Report on Parallel Betweenness Centrality Algorithm (ParBC)

Rui Qiu (rq2170), Hao Zhou (hz2754)

Dec 2021

1 Introduction

Currently, there is a boosting trend of users size in social applications because of the popularization of personal smart devise. This constructs huge social networks containing considerably large quantities of interactions among those users, which is worth further investigation for more accurate and intelligent future functionality. One task is to measure how significant, within a graph, a vertex is. Many algorithms are reported including Betweenness Centrality which our project focuses on.

However, in reality, the network is highly likely vast in size. This could be changing to apply Betweenness Centrality algorithm on those real-world data. Therefore, the project aims to provide a parallel implementation of Betweenness Centrality Algorithm to make the it efficient to perform centrality analysis in real-world social network.

2 Problem Definition

Betweenness Centrality is formulated by Freeman [1] as shown below.

$$Betweenness(k) = \sum_{i \neq k \neq j} (\frac{\sigma_{i,j}(k)}{\sigma_{i,j}})$$

Where $\sigma_{i,j}(k)$ is the number of shortest path between i, j containing the node k and $\sigma_{i,j}$ is the total number of shortest path between i, j. This could, to some extend, reflect the centrality of a node within a network. That is because a node on the shortest

could be regarded as a bridge connecting those two vertices and its significance should be proportional to the total number of shortest path passing through it. For example, with regards to social network, a node at the center of the whole network or some major components or a node acting like a gateway between two major groups will have higher betweenness centrality.

3 Implementation and Algorithm Design

3.1 Notation

3.1.1 Graph Presentation

To take advantage of the sparse nature of social networks, instead of using adjacent matrix, adjacent lists are used. It is implemented by Map in Haskell, presenting like:

$$\{source : [neighbour_1, neighbour_2, ...]\}$$

where the key *source* is an integer identifying a node and the value is a list of integers identifying all its neighbours.

3.1.2 Algorithm Design

We started with the original algorithm we proposed at the beginning. The algorithm is divided into two steps. First step is calculating pair-wise shortest distances and the second is quantifying the times of occurrence of a node on those shortest paths. The heavy calculation inside each function can be implemented in a parallel fashion while two steps should be strictly sequential to ensure correctness of algorithm.

We are targeting at analyzing the betweenness centrality of github network which is inherently a unweighted and undirected graph with our ParBC algorithm in the end. We get insights from Breath-First Search for calculating the pair-wise shortest path. Every BFS-like shortest path calculation can be applied to a single node which provides great feasibility of parallelizing. To acquire the betweenness for each node, the most heavily computational part is calculating the number of shortest path passing through a particular node. It's really expensive if we trying to record every path and extract betweenness from that. Instead, we gain insights from Floyd–Warshall algorithm [2]. More explicitly, for a node v, if the summation of shortest path length from i to v and shortest path length from v to j is equal to the shortest path length from i to j, then we can conclude the shortest path from i to j pass through v. However,

there might be two major limitation. First, not all things can be fully parallelized. The execution of betweenness calculation is depend on the result from shortest path length calculation. This forces the threads to be synchronised and wait in this loop. Second, its complexity is $O(n^3)$ where n is the number of nodes. However, in reality, social network is edge-sparse but the algorithm could not take a good advantage of this feature. Therefore, we turn to Brandes' algorithm [3], which utilizes sparsity of social network quite well. Comparing to $O(n^3)$ for original proposal, it has $O(nm)$, where m is the number of edges in the graph (where $n << m$)

To quantify the times of occurrence of a node on those shortest paths, we could use map-and-reduce fashion to perform this calculation in a parallel manner where, node id is the key and count is the value. This could make a good use of parallelism to calculate the final betweenness centrality for all the nodes in a networks.

3.1.3 Brandes' Algorithm

To describe Brandes' Algorithm [3] and our further adjustment, its notation and precisely formulation would be shown in the following two paragraphs.

First, we follow the notation from the definition of Betweenness Centrality where $\sigma_{i,j}(k)$ is the number of shortest path between i, j containing the node k and $\sigma_{i,j}$ is the total number of shortest path between i, j. In addition, predecessor nodes is defined as:

$$pred_s(v) = \{w \in V | \{w, v\} \in E, d(s, v) = d(s, w) + \omega(w, v)\}$$

Given that the graph is unweighted, $\omega(w, v) = 1$. Therefore, as mentioned in Section 3.1.2, BFS could be utilized to calculate $\sigma_{i,j}$ with $pred_s(v)$ as shown below:

$$\sigma_{i,j} = \sum_{w \in pred_i(j)} \sigma_{i,w}$$

In addition, $pred_s(v)$ helps in calculating betweenness. To simplify the formulation, it denotes the number of shortest path from s to t passing through node v and edge v, w is $\sigma_{s,t}(v, \{v, w\})$ and:

$$\delta_{s,t}(v, \{v, w\}) = \frac{\sigma_{s,t}(v, \{v, w\})}{\sigma_{s,t}}$$

$$\delta_s(v) = \sum_{t \in V} \sum_{w \in Pred_s(v)} \delta_{s,t}(v, \{v, w\}) = \sum_{w \in Pred_s(v)} \sum_{t \in V} \delta_{s,t}(v, \{v, w\})$$

so if $w \neq t$

$$\delta_{s,t}(v, \{v, w\}) = \frac{\sigma_{s,v}}{\sigma_{s,w}} \frac{\sigma_{s,t}(w)}{\sigma_{s,t}}$$

if $w = t$

$$\delta_{s,t}(v, \{v, w\}) = \frac{\sigma_{s,v}}{\sigma_{s,w}}$$

Therefore

$$\delta_s(v) = \sum_{w \in Pred_s(v)} \sum_{t \in V} \delta_{s,t}(v, \{v, w\}) = \sum_{w \in Pred_s(v)} (\frac{\sigma_{s,v}}{\sigma_{s,w}} + \sum_{t \in V, t \neq w} \frac{\sigma_{s,v}}{\sigma_{s,w}} \frac{\sigma_{s,t}(w)}{\sigma_{s,t}}) = \sum_{w \in Pred_s(v)} \frac{\sigma_{s,v}}{\sigma_{s,w}} (1 + \delta_s(w))$$

$$Betweenness(v) = \sum_{s \in V} \delta_s(v)$$

As demonstrated above, rather than do a nested loop to calculate $\sigma_{s,t}(v)$ for Betweeness (which costs $O(n^2)$ for a single node s), $pred_s(v)$ helps in calculating betweenness by propagating $\delta_s(v)$ (which costs $O(m)$ for a single node s where m is the number of edges in the graph which is small given the network is edge-sparsed)

Therefore, it helps to reduce the time complexity from $O(n^3)$ to $O(nm)$

3.2 Sequential Solution

According to the formulation above the sequential version of the algorithm could be drafted as Algorithm 1.

3.3 Algorithm Adjustment and Parallel Solution

To make the algorithm fit for Parallelism, the global variable $Betweenness$ is divided by nodes as:

$$Betweenness(v) = \sum_{s \in V} Betweenness_s(v)$$

Therefore, it could perform in a map-reduce manner, where map the calcualtion of $Betweenness_s(v)$ to the whole list of nodes and reduce all the $Betweenness_s(v)$ by summing up by keys. With this approach, we eliminate the strong step-wise independence of the previous algorithm and implement it a virtually pure parallel fashion.

In addition, to reduce the memory cost for each thread, the BFS is calibrated into

a layer-base implementation. Therefore, the distances are not needed to be tracked within a thread and this memory could be saved for its $Betweenness_s(v)$. Moreover, by deploying layer-based BFS, each branch could be further divided into more thread which make it even suitable for parallelism. However, It turns out that the granularity is so small that the overhead of this further parallelism overwhelm the benefit of itself

Algorithm 1 Sequential Style Implementation

```
0: G ← the graph
0: Betweenness ← {n : 0|∀n ∈ G}
0: for s in G do
0:   pred ← {n : [ ]|∀n ∈ G}; dist ← {n : −1|∀n ∈ G}; sigma ← {n : 0|∀n ∈ G}; S ← [ ]
0:   dist[s] = 0; sigma[s] = 1
0:   queue ← [s]
0:   while queue is not empty do
0:     v = queue.get()
0:     S.append(v)
0:     for w in G.neighbours(v) do
0:       if dist[w] == −1 then
0:         dist[w] = dist[v] + 1
0:         queue.put(w)
0:       end if
0:       if dist[w] == dist[v] + 1 then
0:         sigma[w]+ = sigma[v]
0:         pred[w].append(v)
0:       end if
0:     end for
0:     delta ← {n : 0|∀n ∈ G}
0:     for w in reverse(S) do
0:       for v in pred[w] do
0:         delta[v]+ = sigma[v]/sigma[w] ∗ (1 + delta[w])
0:         if w ≠ s then
0:           Betweenness[w]+ = delta[w]/2
```

4 Evaluation

4.1 Setting

Experiments are performed on Dell G3 with CPU (8th Gen) i7-8750H Hexa-core 2.20GHz and 16GB DDR4 RAM

4.2 Benchmark

The project focuses on undirected and unweighted social network. To be more specific, It would target on the networks listed in SNAP [4]. It should solve the github network [5] where there are 37,700 nodes and 289,003 edges. Each node denotes a github developer who have at least 10 repositories. Between two nodes, an edge exists if there is a mutual follower. According to our observation, this graph is too large for the sequential implementation. Typically, it would take about 4 days to calculate the result. However, the parallel implementation could solve it in 10 hours which is considerably comparable to a professional network analysis library called NetworkX [6] which is based on a python library SciPy [7] on top of NumPy [8] and C++.

In addition, to incur sequential implementation into the comparison, two edge-spars social networks, Simulate1000 and Simulate 2000 are randomly generated for simulation testing.

4.3 Test Result

To assure the correctness of our implementation, we tested different simulation network with a relatively small size ($n = 10, 50, 100, 500$) against the NetworkX library [6], The sum of the differences is lower than 10^{-13}

Table 1: Correctness Testing Graph Set

Graph Name	Number of Nodes	Number of Edges
Simulate20	20	96
Simulate50	50	389
Simulate100	100	884
Simulate200	200	1878
Simulate500	500	4870

For performance, we performed both sequential and parallel version of the algorithm

with 3 medium and large size graphs.

Table 2: Performance Testing Graph Set		
Graph Name	Number of Nodes	Number of Edges
Simulate1000	1000	9859
Simulate2000	2000	19844
SNAP	37700	289003

Table 3: Experiment Results for Simulate1000						
N	time(s)	converted	gc'd	fizzled	total	Speedup
seq	6.012	N/A	N/A	N/A	N/A	N/A
2	3.620	1998	1	1	2000	X1.66
3	2.590	1998	1	1	2000	X2.32
4	2.110	1998	1	1	2000	X2.85
5	1.920	1998	1	1	2000	X3.13
6	1.780	1998	1	1	2000	X3.38

Table 4: Experiment Results for Simulate2000						
N	time(s)	converted	gc'd	fizzled	total	Speedup
seq	54.10	N/A	N/A	N/A	N/A	N/A
2	33.62	3998	1	1	4000	X1.61
3	27.76	3998	1	1	4000	X2.06
4	23.36	3998	1	1	4000	X2.32
5	21.07	3998	1	1	4000	X2.49
6	20.23	3998	1	1	4000	X2.67

Table 5: Experiment Results for SNAP		
Method	time	Speedup
seq	44h 43 mins	N/A
ParBC-N6	18h 38 mins	X2.40
NetworkX	16h 23 mins	X2.73

4.4 Performance Analysis

From the results, we can conclude is much more faster than the sequential version and fairly comparable to NetworkX.

We can also observe that when N = 6, which is equal to the number of cores, the performance of ParBC is the best, about 3.38 times . If N is set to be larger, we can

definitely get better performance since the adjusted version of algorithm can be implemented in a pure parallel manner. Additionally, the computation for each sparks is growing heavier as the size of the graph grows, and when we target at social networks which are inherently large graphs(most of them are even larger than SNAP [4]), the performance improvement of ParBC are guaranteed on those input since the computation heaviness grow faster than parallel overhead .

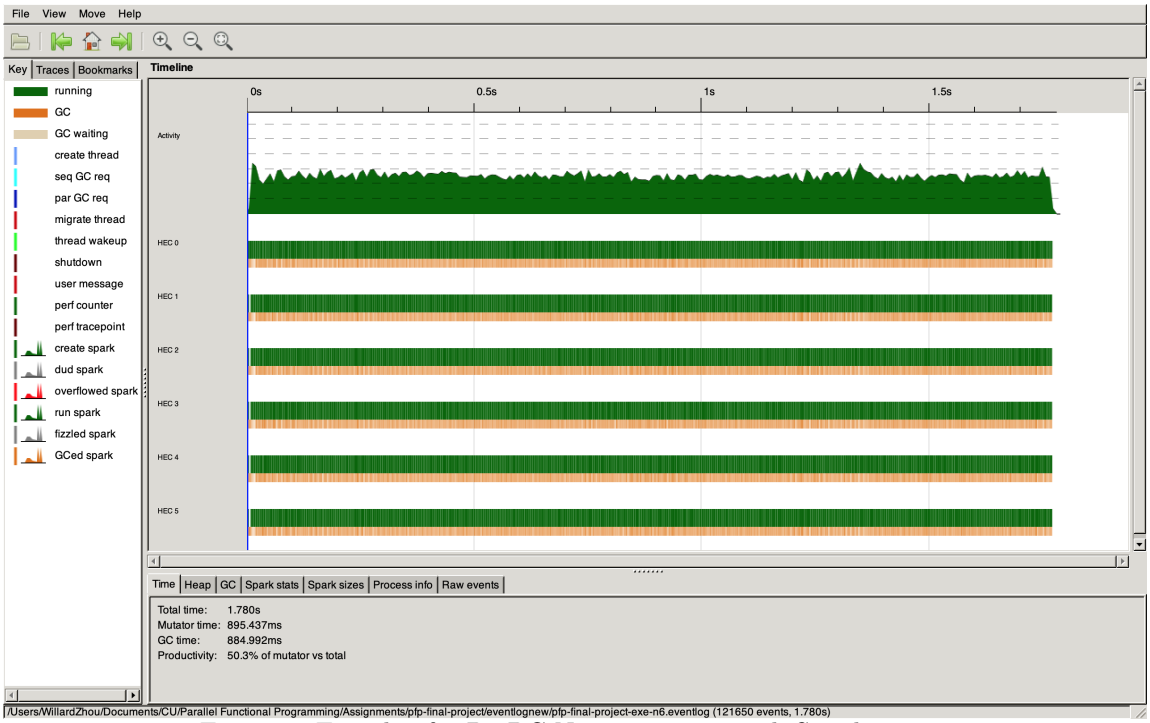

Figure 1: Eventlog for ParBC-N6 experiment with Simulate1000

5 Future work

Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

Maximum heap size:	12.0	MiB	12,582,912	bytes
Maximum heap residency:	4.0	MiB	4,199,816	bytes
Total allocated:	49.4	GiB	53,094,739,480	bytes
Allocation rate:	1.5	GiB/s	1,563,175,044	bytes per second (of mutator time)
Maximum slop:	117.0	KiB	119,816	bytes

Figure 2: Sequential solution heap usage with Simulate2000

Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

Maximum heap size:	49.0	MiB	51,380,224	bytes
Maximum heap residency:	14.1	MiB	14,826,432	bytes
Total allocated:	49.5	GiB	53,148,045,456	bytes
Allocation rate:	5.3	GiB/s	5,647,139,855	bytes per second (of mutator time)
Maximum slop:	266.5	KiB	272,888	bytes

Figure 3: ParBC-N6 heap usage with Simulate2000

During the performance test, we found that our ParBC algorithm reach a bottleneck for performance improvement. We expect better performance speedup as scale of the graph grows, however, it becomes worse when we start to test with Simulate2000 and bigger graphs. After the investigation with heap usage, we found memory size become a major restriction as the figure showed.As the figures suggest, Sequential solution have larger heap space than ParBC-N6 with regarding to available memory per-core wise.

Therefore, for the future work, we will try to investigate and optimize the memory usage with library we covered in the lecture and all other memory-optimization libraries to deal with the obstacle. Hopefully, we can get good performance as expected.

6 Conclusion

In conclusion, ParBC provides a parallel betweenness centrality calculation tool. It calibrates and hence makes itself more suitable for parallelism by deploying layer-wise BFS, dividing total BC into independent sub-BC with regards to different source nodes. This considerably speeds up the sequential implementation and even could be comparable to a C-kernel professional benchmark NetworkX [6]. However, we also notice that the memory size and utilization becomes the bottleneck against the performance, which would be further investigated in our future work.

7 Appendix

Listing 1: Main.hs

```
module Main where

import System.Exit(die)
import System.Environment(getArgs, getProgName)
import qualified Data.Map.Strict as Map

import BCsequential
import BCparallel

main :: IO ()
main = do args <- getArgs
          case args of
            [version,filename] -> do
              contents <- readFile filename
              let inputMap = Map.fromList rawList
                  rawList = map transfromSingleLine rawLines
                  rawLines = lines contents -- [String] -> [(int,[int])
                     ]
              case version of
                   "sequential" -> do print version
                                      print $ length $ bcSolver
                                         inputMap
                   "parallel" -> do print version
                                    print $ length $ bcSolverPar
                                       inputMap
                   _ -> die $ "Usage: Choose correct version (
                      sequential / parallel)"
            _ -> do pn <- getProgName
                    die $ "Usage: "++pn++"<version> <filename>"


transfromSingleLine :: String -> (Int,[Int])
transfromSingleLine str = (read node, map read neighbors)
    where
      (node:neighbors) = words str
```

Listing 2: BasicType.hs

```
module BasicType where

import qualified Data.Map.Strict as Map
{-
Sample graph presented by adjacency list
```

```
{
1: 2
2: 1,3
3: 2,4
4: 3
}
-}


type Graph = Map.Map Int [Int]

sampleG :: Graph
sampleG = Map.fromList [(1,[2]),(2,[1,3]),(3,[2,4]),(4,[3])]
```

Listing 3: BCsequential.hs

```
module BCsequential where

import BasicType
import qualified Data.Map.Strict as Map
import qualified Data.Set as Set

sg :: Graph
sg = sampleG

shortestPathMap :: Map.Map Int [Int] -> Map.Map (Int, Int) Int
shortestPathMap g = Map.fromList [((s,e),shortestPath g s e)|s <- Map.
    keys g, e <- Map.keys g, (/=) s e]

shortestPath :: Graph -> Int -> Int -> Int
shortestPath g s e = bfs e g (Set.fromList [s]) [] 0


bigG :: Graph
bigG =  sampleG


bfs:: Int -> Graph -> Set.Set Int -> [Int] -> Int -> Int
bfs target g frontier explored depth
    | Set.member target frontier = depth
    | otherwise = bfs target g newFrontier newExplored (depth+1)
      where
        newFrontier = Set.fromList (concatMap findchild (Set.toList
            frontier))
        newExplored = explored ++ Set.toList frontier
        findchild :: Int -> [Int]
        findchild i
```

```
        | Map.lookup i g == Nothing = error "invalid key for lookup"
        | otherwise = filter (\neighbor-> notElem neighbor explored)
          adjList
          where (Just adjList) = Map.lookup i g


calculateSigmaAndSoOn :: Graph -> Set.Set Int -> [Int] -> Set.Set Int
  -> Map.Map Int [Int] -> Map.Map Int Int -> (Map.Map Int [Int], Map.
  Map Int Int, [Int])
calculateSigmaAndSoOn g frontier s explored prede sigma
  | Set.null frontier = (prede, sigma, s)
  | otherwise = calculateSigmaAndSoOn g newFrontier newS newExplored
    newPred newSigma
  where
    newFrontier = Set.fromList (concatMap findchild (Set.toList
      frontier))
    newS = s ++ Set.toList frontier
    newExplored = Set.union explored frontier
    (newPred, newSigma) = updatePredSigma prede sigma (Set.toList
      frontier)
    findchild :: Int -> [Int]
    findchild i
      | Map.lookup i g == Nothing = error "invalid key for lookup"
      | otherwise = filter (\neighbor-> Set.notMember neighbor
        newExplored) adjList
        where (Just adjList) = Map.lookup i g
    updatePredSigma :: Map.Map Int [Int] -> Map.Map Int Int -> [Int
      ] ->(Map.Map Int [Int], Map.Map Int Int)
    updatePredSigma p1 s1 [] = (p1, s1)
    updatePredSigma p1 s1 (v:vs) = updatePredSigma p1New s1New vs
      where
        (p1New, s1New) = updateForSingleParent (findchild v) p1
          s1
        updateForSingleParent :: [Int] -> Map.Map Int [Int] ->
          Map.Map Int Int -> (Map.Map Int [Int], Map.Map Int Int
          )
        updateForSingleParent [] p2 s2 = (p2,s2)
        updateForSingleParent (w:ws) p2 s2 =
          updateForSingleParent ws p2New s2New
          where
            p2New = Map.insertWith (++) w [v] p2
            s2New = Map.insertWith (+) w sig s2
              where (Just sig) = Map.lookup v s2
```

```
accumulateCB :: Map.Map Int Double -> Int -> Map.Map Int Double -> Map.
    Map Int [Int] -> Map.Map Int Int -> [Int] ->  Map.Map Int Double
accumulateCB cb _ _ _ _ [] = cb
accumulateCB cb start delta prede sigma (w:ws) = accumulateCB cbNew
    start deltaNew prede sigma ws
    where
      (Just deltaW) = Map.lookup w delta
      cbNew = if (/=) w start then Map.insertWith (+) w deltaW cb else
          cb
      deltaNew = updateDelta predList delta
        where
          (Just predList) = Map.lookup w prede
          updateDelta :: [Int] -> Map.Map Int Double -> Map.Map Int
              Double
          updateDelta [] d = d
          updateDelta (v:vs) d = updateDelta vs dNew
            where
              dNew = Map.insertWith (+) v dValue d
              dValue :: Double
              dValue = (fromIntegral sigmaV) / (fromIntegral sigmaW) *
                  (1.0 + deltaW)::Double
              (Just sigmaV) = Map.lookup v sigma
              (Just sigmaW) = Map.lookup w sigma



calculatePerNode :: Graph -> [(Int, [Int])] -> [(Int, Int)] -> [(Int,
    Double)] -> Int ->Map.Map Int Double
calculatePerNode g iniListPred iniListSigma iniListIntDouble node = Map
    .map (/ 2.0) resultMap
    where
      resultMap = accumulateCB (Map.fromList iniListIntDouble) node (
          Map.fromList iniListIntDouble) prede sigma (reverse s)
      (prede,sigma,s) = calculateSigmaAndSoOn g (Set.fromList [node])
          [] Set.empty (Map.fromList iniListPred) (Map.insert node 1 (
          Map.fromList iniListSigma))


bcSolver :: Graph -> Map.Map Int Double
bcSolver g = foldl (Map.unionWith (+)) Map.empty bcMapList
    where
      bcMapList::[Map.Map Int Double]
      bcMapList = map (calculatePerNode g iniListPred iniListSigma
          iniListIntDouble) nodelist
      nodelist :: [Int]
      nodelist = Map.keys g
```

```
        iniListPred :: [(Int, [Int])]
        iniListPred = map (\x -> (x,[])) nodelist
        iniListSigma :: [(Int, Int)]
        iniListSigma = map (\x -> (x,0)) nodelist
        iniListIntDouble :: [(Int, Double)]
        iniListIntDouble = map (\x -> (x,0.0)) nodelist
```

Listing 4: BCparallel.hs

```
module BCparallel where

import BCsequential
import BasicType

import qualified Data.Map.Strict as Map
import Control.Parallel.Strategies

myparMap :: (a -> b) -> [a] -> Eval [b]
myparMap _ [] = return []
myparMap f (a:as) = do b <- rpar (f a)
                       bs <- myparMap f as
                       return (b:bs)

bcSolverPar :: Graph -> Map.Map Int Double
bcSolverPar g = foldl (Map.unionWith (+)) Map.empty bcMapList
    where
      bcMapList = parMap rpar singleSolver nodelist
      --bcMapList = runEval $ myparMap singleSolver nodelist
      singleSolver :: Int -> Map.Map Int Double --solve Bc from one
          node
      singleSolver = calculatePerNode g iniListPred iniListSigma
          iniListIntDouble
      nodelist :: [Int]
      nodelist = Map.keys g
      iniListPred :: [(Int, [Int])]
      iniListPred = map (\x -> (x,[])) nodelist
      iniListSigma :: [(Int, Int)]
      iniListSigma = map (\x -> (x,0)) nodelist
      iniListIntDouble :: [(Int, Double)]
      iniListIntDouble = map (\x -> (x,0.0)) nodelist
```

Listing 5: random graph generation

```
import networkx as nx
from collections import defaultdict

g = nx.generators.random_graphs.powerlaw_cluster_graph(2000,10,0.2,
    seed = 0)
```

```
c=0
es = defaultdict(list)
for e in g.edges:
    s = e[0]
    t = e[1]
    if s == t:
        continue
    es[s].append(str(t))
    es[t].append(str(s))
    c+=1

for e in es.keys():
    print(str(e) + " "+ " ".join(set(es[e])))
print(c)
```

References

[1] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

[2] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[3] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[4] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[5] B. Rozemberczki, C. Allen, and R. Sarkar, "Multi-scale attributed node embedding," 2019.

[6] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[7] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2