Rapid Analysis of Network Connectivity

Scott Freitas  
Arizona State University  
Tempe, Arizona 85281  
scott.freitas@asu.edu

Nan Cao  
Tongji University  
Shanghai, China  
nan.cao@nyu.edu

Hanghang Tong  
Arizona State University  
Tempe, Arizona 85281  
hanghang.tong@asu.edu

Yinglong Xia  
Huawei  
Santa Clara, CA 95050  
yinglong.xia@huawei.com

ABSTRACT
This research focuses on accelerating the computational time of two base network algorithms (k-simple shortest paths and minimum spanning tree for a subset of nodes)—cornerstones behind a variety of network connectivity mining tasks—with the goal of rapidly finding network pathways and trees using a set of user-specific query nodes. To facilitate this process, we utilize: (1) multi-threaded algorithm variations, (2) network re-use for subsequent queries and (3) a novel algorithm to reduce the network search space. Key Neighboring Vertices (KNV). The proposed KNV algorithm serves a dual purpose— (a) to reduce the computation time for algorithmic analysis and (b) to identify key vertices in the network (context). Empirical results indicate this combination of techniques significantly improves the baseline performance of both algorithms. We have developed a web platform utilizing the proposed network algorithms that enables researchers and practitioners to both visualize and interact with their datasets (PathFinder: http://www.path-finder.

1. INTRODUCTION

Motivation. With the advent of big data era and the emergence of network science, large-scale networks appear across many disciplines, from medicine and epidemiology to advertising and marketing. As a result, there is an exponential amount of network data being generated at an unprecedented rate. How we translate this large network data into meaningful knowledge with limited computational resources is the challenge before us.

Problem. How can we rapidly explore, analyze and visualize a set of user-specific query nodes in relation to a dataset? We envision that tree, pathway and context are the three key components to answer this question. Formally, given a graph G = (V, E) and a set of user-specific query nodes Q, we seek to find (1) the relationship between each query node in Q (i.e. tree detection), (2) a subset of paths R ⊂ G such that R contains only vertices and edges that provide key path information between different query nodes in Q (i.e. pathway detection) and (3) a subset of important vertices, C and edges, S, such that C ⊂ V and S ⊂ E (i.e. context detection).

Related Work. There has been significant research related to our proposed algorithms. In addition, the concept of user-specific query nodes (seed nodes) has been an active research topic. For example, Staudt et. al. used user-specific query nodes for community detection via “selSCAN” [2] and Akoglu et. al. used them to find connection pathways [1].

K-Simple Shortest Paths (pathway): A theoretical analysis has been laid out by Ruppert [4] along with the theoretical and experimental work by Guerriero et. al. using a shared memory model [5]. More recently, Singh et. al. used GPUs along with CUDA to parallelize the algorithm, resulting in impressive speedup [6]. In addition, two representative works for connection subgraph identification (i.e. pathway detection) can be seen in [1] and [3].

Shortest Paths MST (tree): Related work towards creating a minimum spanning tree for a subset of nodes has been done by Cenek et. al. Instead of creating a shortest paths distance matrix, they proposed to insert the shortest paths into the tree dynamically [8].

Key Neighboring Vertices (context): Sulieman et. al. proposed a semantic social breadth first search algorithm which takes the top N vertices with the highest centrality degree and attempts to find the nearby influential players [7].

Contributions. Our main contributions are three-fold: (1) the development of two multi-threaded algorithms: k-simple shortest paths (KSSP) and minimum spanning tree for a subset of nodes (Shortest Paths MST); (2) the creation of a novel algorithm, Key Neighboring Vertices (KNV), for reducing network search space and identifying key vertices; (3) the development of a web platform, utilizing these algorithms, for researchers and practitioners to upload their own network data for analysis and visualization.

1. K-Simple Shortest Paths (pathway): Our approach to the k-simple shortest path algorithm offers three new features: (a) instead of creating a multi-threaded single source shortest path algorithm [4][5][6], we parallelize each single source shortest path computation required to find a path in k (for details see section 3.1); (b) we pre-process the network data to reduce the search space using the KNV algorithm; (c) the generated pathways and surrounding context nodes
can be re-queried to find additional paths or trees.

**Shortest Paths MST (tree):** Our approach to solving the minimum spanning tree for a subset of nodes (Shortest Paths MST) is centered around (a) parallelizing the shortest path computations to run simultaneously; (b) pre-processing the network data to reduce the search space of the algorithm; (c) re-querying the generated tree and surrounding context nodes to find additional trees or paths.

2. Key Neighboring Vertices (context): Our approach is similar to Suliceman et. al., but with three key differences. (a) Instead of exploring the network based on the top N vertices with highest degree centrality, we seed the map with a set of user-specific query nodes; (b) we propose using above average network degree centrality as the metric for including nearby neighboring vertices for further exploration in the search list; (c) we allow additional parameters that let the user control the exploration process.

3. Platform: We have developed PathFinder, a web platform to assist users in mining network connectivity from large networks. PathFinder begins by taking an input network uploaded by the user or selection from a pre-loaded dataset. Depending on the user’s expertise with the program there are two sets of controls: basic and advanced. The basic controls allow the user to start without an understanding of the algorithms, while the advanced controls allow the user to fine-tune their queries and obtain information that may not be available with a basic search. Visualizations are generated using vis.js and GraphViz. A video demo of the platform is available at: https://youtu.be/PxQVd-Gmkuc

2. PLATFORM FUNCTIONALITY

Each part of Figure 1 highlights some of the platforms core functionality. Figure 1(1) shows a sample visualization using the pathway detection algorithm on the DBLP network. Figure 1(2) allows the user to enter the algorithm parameters and select a network for analysis. Figure 1(3) allows the user to enter nodes and edges to be removed from the graph search, select whether or not to re-use the current graph results for further analysis and change the configuration of network style parameters, including: node-edge color scheme and node size. Figure 1(4) is a zoomed in portion of Figure 1(1). The red nodes represent the start and end query vertices, orange for intermediate path vertices, blue for one hop away critical vertices and purple for two hop away critical vertices. The blue and purple vertices surrounding the the path vertices are determined by the KNV algorithm.

**Engaging the Audience.** We expect that our demo will primarily attract two audiences, (1) practitioners who are interested in exploring the connectivity between key nodes in large networks, and (2) information management and data mining researchers who develop new algorithms and tools.

3. TECHNICAL DETAILS

All algorithms perform on an undirected, unweighted, adjacency list graph representation, $G=(V, E)$. Nonetheless, we note that the proposed platform is flexible to admit alternative algorithms and graph types with mild changes.

3.1 Pathway Detection

**Problem definition:** Given two or more pre-marked vertices, $x, y \in V$ from the graph $G=(V, E)$, this algorithm will find $k$-simple shortest paths from $x$ to $y$.

**Algorithm description.** We adopt the $k$-simple shortest path threaded and search reduced algorithm (Algorithm 1) to detect key pathways that connect the query nodes, which can be viewed in 7 steps:

1. Run the Key Neighboring Vertices (KNV) algorithm to detect key pathways that connect the query nodes, which can be viewed in 7 steps:
2. Run the Key Neighboring Vertices (KNV) algorithm to detect key pathways that connect the query nodes, which can be viewed in 7 steps:
3. Run the Key Neighboring Vertices (KNV) algorithm to detect key pathways that connect the query nodes, which can be viewed in 7 steps:
4. Optional: Re-query the generated network with different paths.

**Algorithm 1 Pathway Detection: K-Simple Shortest Paths Threaded and Search Reduced**

**Input:** Graph $A = (V, E); sv, ev \in V$

**Output:** Array of paths: sPaths[]

**Initialization:** sPaths[], pHolder[], eHolder[]; cPath := 0

Graph B = Key Neighboring Vertices(A, sv, ev)

sPaths[cPath] = Dijkstra(sv, ev, B)

cPath++

Graph B = Key Neighboring Vertices(A, sPaths)

while cPath < numPaths do

pIndex := 0

for each edge $e \in sPaths[cPath-1]$ do

  Graph C = B

  C.removeEdge(e)

  eHolder[pIndex] = e

  pHolder[pIndex] = new_thread(Dijkstra(sv, ev, C))

  pIndex++

shortestPaths[cPath] = pHolder.getMinPath

B.removeEdge(eHolder[cPath])

cPath++

3.2 Tree Detection

**Problem definition:** Given two or more pre-marked vertices, $2 \ldots x \in V$ from the graph $G=(V, E)$, this algorithm will find a MST that is constructed from a combination of single source shortest paths.

**Algorithm description.** We adopt the Shortest Paths Minimum Spanning Tree algorithm (Algorithm 2) to determine the relationship between the user-specified query nodes. This can be viewed in 4 steps:

1. Each pre-marked node, $v_i$ will run a single source shortest path algorithm against every other pre-marked vertex. The single source shortest path algorithms run in parallel.
2. Sort the resulting paths in ascending order of path length.
3. Run Kruskal’s algorithm to determine which of these shortest paths form the MST.
4. Optional: Re-query the generated network with differ-
Figure 1: An illustrative example of our platform to find the key pathways. Start and end vertices are in red.

Algorithm 2 Tree Detection: Shortest Paths MST

Input: Graph A = (V, E); Array of integers: vertices[]
Output: Array of paths: sPaths[]
Initialization: struct PathInfo {Vertex v1, v2}, PathInfo paths[], Two integers: pathCount, pathsFound := 0
for each unique pair of vertices p1, p2 ∈ vertices do
    paths[pathCount].v1 = p1, paths[pathCount].v2 = p2
    pathCount++
Graph B = Key_Neighboring_Vertices(A, vertices)
for i := 0 to pathCount do
    sPaths[pathsFound] = new thread(Dijkstra(B, paths[i]))
    pathsFound++
Sort Ascending Order(sPaths)
Kruskal’s Algorithm(sPaths)

3.3 Context & Speed-up

For both tree and pathway detection, we propose an efficient algorithm to detect key neighboring vertices to reduce the search space using a combination of three techniques to identify critical nodes: (1) vertex centrality, (2) edge connection to a pre-marked node and (3) breadth first search. This allows us to create a reduced graph \( R = (V, E) \), that is a subset of \( G, R \subseteq G \). Through this process we implicitly assume that vertices with high centrality are key hubs in the graph and are therefore important ‘players’ in the network.

The proposed key neighboring vertices algorithm can be viewed in 4 steps:

1. Determine if the current vertex has an edge connection to one of the ‘key’ vertices and has above average vertex centrality. If both conditions are met, place the current vertex into a bin of that key vertex.
2. Sort each bin in descending order of vertex centrality.
3. From each bin, take the top \( x \) neighboring nodes as important vertices at that depth level and add them to the reduced adjacency list.
4. From each bin, a percentage of the top \( x \) nodes will become ‘key’ vertices and recursively undergo the process until the specified depth level is reached.

Algorithm 3 Context & Speed-up: Key Neighboring Vertices

Input: Graph A = (V, E); Array of integers: vertices[]; Six integers: cDepth, depth, avgCentrality, numVertex, numVerticesNextIter, numVerticesCritical
Output: Graph R = (V, E)
Initialization: bins[][]
for i := 0 to A.size do
    if A[i].centrality > avgCentrality then
        for k := 0 to vertices.size do
            if vertices[k] ∩ A[i] then
                bins[k] += i
        for each array, a, in bins do
            Sort Descending_Order_Vertex_Centrality(a);
        for i := 0 to vertices.size do
            for k := 0 to bins[i].size do
                if !vertices[i] ∩ R[bins[i].at(k)] then
                    R.addEdge(vertices[i], bins[i].at(k))
        if cDepth < depth then
            numVertex = numVerticesNextIter
            vertices = new vertices[]
            cDepth++
        for each array, a, in bins do
            for i := 0 to a.size and i < numVerticesCritical do
                vertices.insert(a[i])
        key_Neighboring_Vertices(vertices, numVertex, cDepth)

3.4 Empirical Evaluation

We used the DBLP co-authorship and the LiveJournal social network from the Stanford SNAP network to gather empirical data on the platform. The two measures we aim to quantify are speed and accuracy. Since the Shortest Paths MST and K-Simple Shortest Paths algorithm utilize the same search space reduction algorithm and multithreading techniques, we use the KSSP algorithm to represent the Shortest Paths MST in terms of accuracy and run time. All data was collected locally and does not account for any additional run time caused by using the web-platform.

To compare the accuracy and run time, we ran three variations of the K-Simple Shortest Path algorithm (KSSP). The first variation (v.1) contained only the core KSSP algorithm with no search space reduction or multithreading. The second variation (v.2) ran the KSSP algorithm with multithreading (KSSPT). The third variation (v.3) ran the KSSP algorithm with multithreading and the search space...
reduction algorithm (KSSPR). The run time and accuracy of the three KSSP variations can be seen in Figure 2 and Figure 3-4 respectively. It should be noted that (v.1) and (v.2) of the KSSP algorithm will always find the shortest paths available, while the same guarantee cannot be extended to (v.3). The reasoning behind the possible suboptimal path(s) for (v.3) is due to the nature of the search space reduction algorithm applied to the graph. The KNV algorithm uses a tradeoff between accuracy and run time, which can be varied depending on the parameters. In trials for Figure 2 we applied parameters to the KNV algorithm that retained accuracy at the cost of speed. However, even with this additional ‘cost’, trial one results show (v.3) over \(2.5x\) faster than (v.1) and \(1.7x\) faster than (v.2) with no loss of accuracy with respect to the full network (last data point). It can be seen in Figure 4 that the run time and path length for the variations is dependent upon the start and end vertices.

Figure 2: Run time results for KSSP, KSSPT and KSSPR. Start Vertex: 61, End Vertex: 70591, # of paths: 6. No data for KSSPR on first two data points due to chosen KNV parameters.

Figure 3: KSSPR and MST Shortest Paths algorithms w/ and w/o the re-query feature. Each data point ranges from 20% to 100% of the Live-Journal network in 1/5th intervals. KSSPR: Start Vertex: 35521, End Vertex: 286345, # of paths 8. MST Shortest Paths: Vertices: 0, 58, 9558, 34343.

In order to better access the abilities of the platform we ran both the KSSPR and Shortest Paths MST algorithms on the LiveJournal network. To put it in perspective, the LiveJournal network has approximately 38.5 million edges and vertices compared to the DBLP network of 1.3 million. In figure 3, the labels KSSPM and MSTM represent re-query results compared to Algorithm 1 and 2 with no re-query. Comparing the full network (5th interval) run time for KSSPR vs KSSPM we see an \(5.3x\) speed-up and a \(5.6x\) speed-up for the MSTM vs MST. It should be noted that (a) parameters were held constant when re-querying the network and that only information generated from the previous run is analyzed when re-querying and (b) shorter paths can be found when re-querying the graph since it’s saved based on visualization parameters not search space reduction.

4. CONCLUSIONS

The goal of this work is to rapidly analyze network connectivity. We believe the computational speedup obtained will be of interest to information management and data mining researchers. In addition, the web platform PathFinder allows users to quickly and intuitively determine network connectivity between a set of user-specific query nodes. An operational prototype is online: http://path-finder.io and source code will be made publicly available by the conference date.

5. ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under Grant No. IIS-1651203, IIS-1715385 and IIS-1743040, by DTRA under the grant number HDTRA1-16-0017, by Army Research Office under the contract number W911NF-16-1-0168, and gifts from Huawei and Baidu.

6. REFERENCES


Figure 4: Each data point represents # of paths found: 2-14 in intervals of 4. Trial 1: Start Vertex: 35521, End Vertex: 286345. Trial 2: Start Vertex: 9790, End Vertex: 26073.