Fast and Accurate Determination of Graph Node Connectivity Leveraging Approximate Methods

Robert S. Sinkovits[1]

[1] San Diego Supercomputer Center, University of California San Diego, La Jolla, CA 92093, USA
sinkovit@sdsc.edu

Abstract. For an undirected graph *G*, the node connectivity *K* is defined as the minimum number of nodes that must be removed to make the graph disconnected. The determination of *K* is a computationally demanding task for large graphs since even the most efficient algorithms require many evaluations of an expensive *max flow* function. Approximation methods for determining *K* replace the *max flow* function with a much faster algorithm that gives a lower bound on the number of node independent paths, but this frequently leads to an underestimate of *K*. We show here that with minor changes, the approximate method can be adapted to retain most of the performance benefits while still guaranteeing an accurate result.

Keywords: Graph algorithm, node connectivity, approximation methods, *k*-components.

1 Introduction

Given an undirected graph, what is the smallest number of nodes that need to be removed so that the graph is broken into two or more disjoint components (i.e. there no longer exist paths connecting all possible pairs of nodes)? Like many problems in computer science, the question is easy to pose, but can be difficult to solve. Although the minimum node degree provides an upper bound, even highly connected graphs can become fragmented by the removal of just a few nodes (Fig. 1). The crux of our contribution is recognizing that the expensive vertex disjoint paths calculations underlying the solution of the node connectivity problem can first be estimated using a much faster approximate algorithm. We can easily determine when the approximation is invalid and revert to the more expensive calculation where necessary, thereby guaranteeing that we get the correct result.

The node connectivity problem is not just of theoretical interest, but is highly relevant to a number of fields. It has been applied to the impact of peer groups on juvenile delinquency [1], economic network models [2], clustering in social networks [3], political polarization [4], community structure [5, 6] and neural connectivity [7].

In the remainder section we define the conventions and nomenclature that will be used throughout the paper and describe the current state of the art in the calculation of the graph node connectivity. This is followed by descriptions of our new faster algorithm (section 2), the implementation (section 3), results of computational experiments (section 4) and finally a discussion and future work (section 5). Notation and symbols used repeatedly in the manuscript are summarized in Table 1.

A graph $G = (V, E)$ is specified by a collection of nodes $V(G)$ and the edges $E(G)$ that connect pairs of nodes. An undirected graph is a graph for which the edges do not have a directionality. The numbers of nodes and edges in the graph are denoted by $|V(G)|$ and $|E(G)|$, respectively. Since we are working with a single graph, we simply denote the node and edge counts as $|V|$ and $|E|$. The degree of a node, $d(v)$, is the number of edges that connect to node v and δ is the minimum vertex degree across all nodes in the graph. A set of nodes S whose removal makes the graph disconnected is a cut set and the node connectivity K of a graph is defined as the size of the minimal cut set.

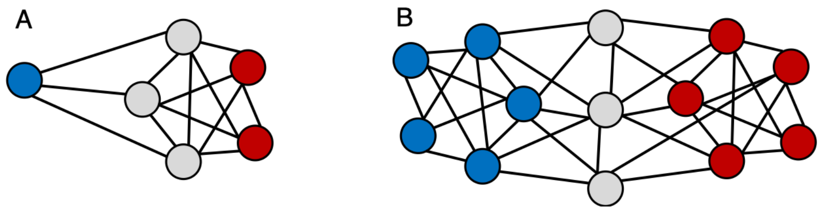

Fig. 1. Removal of gray nodes separates graphs into two disjoint graphs (blue and red nodes). In panel A, the degree of the blue node is the same as the number of nodes that need to be removed. In panel B, although no node has degree less than four, the removal of three nodes is sufficient.

Table 1. Notation and symbols used in manuscript

symbol	definition
v, w	Nodes
v_s	Starting node used in node connectivity algorithms
$d(v)$	Degree of node v
δ	Minimum node degree
E, $\|E\|$	Edges, number of edges
κ	Local node connectivity
κ_{approx}	Approximate local node connectivity based on shortest paths
K	Graph node connectivity
K_{approx}	Graph node connectivity based on approximate algorithm
n_{target}	Number of target nodes tested in search for optimal v_s
n_{trial}	Number of starting nodes tested in search for optimal v_s
t_{approx}	Run time for approximate node connectivity algorithm based on shortest paths
t_{fast}	Run time for fast node connectivity algorithm (presented in this paper)
t_{orig}	Run time for original node connectivity algorithm
$t_{worst\text{-}case}$	Worst-case run time for fast node connectivity algorithm
V, $\|V\|$	Nodes, number of nodes in graph

2 Related Work

The determination of K is a computationally challenging problem (for a good overview of both node and edge connectivity algorithms, see [8]). Several approaches have been designed for this purpose that rely on the underlying evaluation of a *max flow* function. This in turn is based on Menger's theorem, which states that for a pair of non-adjacent nodes in an undirected graph, the number of node independent paths between the pair (i.e. paths that have no nodes in common other than the starting and ending points) is equal to the number of nodes that must be removed from the graph so that no paths remain between the pair [9]. Let $\kappa(G, v, w)$ be the local node connectivity function that returns the number of independent paths between vertices v and w in graph G. A brute force approach to finding K would be to simply calculate $\kappa(G, v, w)$ for all non-adjacent pairs of nodes in G and take the minimum value. It should be noted that there are several algorithms that do not rely on *max flow*, either addressing specific problems such as confirming 3-connectivity [10] or 4-connectivity [11] or the general case using random methods [12]. We will limit our discussions to only those that use *max flow*.

Several authors have developed schemes that minimize the number of calls to κ, leading to much better scaling. Even and Tarjan presented an algorithm that required $(K + 1)(|V| - \delta - 1) - ½ K(K + 1)$ calls to the *max flow* function [13]. Esfahanian and Hakimi further reduced the number of calls to $(|V| - \delta - 1) + ½ K(2\delta - K - 3)$ [14]. They noted that for each minimum node cut set in G, there exists at least one node that does not belong to the cut set. A randomly chosen node therefore either belongs to every minimum cut set or is outside of at least one minimum cut set. In the first case, it is sufficient to calculate the number of node independent paths between all pairs of non-adjacent neighbors of the selected node (i.e. between neighbors of the selected node that are not themselves neighbors). In the second case, we need to calculate the node connectivity between the selected node and all non-neighboring nodes. This algorithm, which accounts for both cases, can be expressed as follows.

```
$G = (V, E)$
$K \leftarrow \delta$
select arbitrary node $v_s$ such that $d(v) == K$
for $w$ in $V \setminus \{\{\text{neighbors}(v_s)\} \cup \{v_s\}\}$
    $K \leftarrow \min(\kappa(G, v_s, w), K)$
for $x, y$ in $\{\text{neighbors}(v_s)\}$ such that $x, y$ non-adjacent
    $K \leftarrow \min(\kappa(G, x, y), K)$
```

Improvements to the performance of this graph node connectivity algorithm would require either a further reduction in the number of evaluations of κ or the implementation of a faster method for calculating κ. Esfahanian and Hakimi had already considered the former and described a procedure for limiting the number of target nodes in the case where v_s is not a member of the minimum cut set. Although not considered in their paper, identifying a starting node v_s with the absolute smallest number of non-adjacent neighbors from among the minimal degree nodes could lead to a reduction in number of calls to κ, but for large graphs this would make a negligible difference. Regarding the latter option, although it might be possible to find more efficient ways to calculate κ, multiple algorithms already exist and additional progress would be extremely difficult.

Our approach is described in the next section and, while related to the efficient evaluation of κ, does not require the development of a new or refinement of an existing *max flow* algorithm. Instead, we use a combination of the approximate and exact algorithms for κ to boost performance while still guaranteeing an accurate result for K.

3 Faster Computation of Graph Node Connectivity

An approximate algorithm for calculating the local node connectivity, denoted κ_{approx}, is based on the repeated identification of the shortest path between the pair of nodes [15]. The shortest path is first identified and nodes on this path are marked as visited. The next shortest path involving only unvisited nodes is then determined and the nodes on that path are also marked as visited. The process is repeated until no paths remain between the node pair that involve only unvisited nodes. This approximate algorithm is extremely fast since the shortest path calculations are much more efficient than any known *max flow* algorithm. An approximate method for finding K simply replaces the calls to κ with κ_{approx}. The use of this approximation has also been applied to the problem of finding the k-components of a graph, where a k-component is the maximal subgraph with node connectivity equal to at least k [16]. The obvious downside of using κ_{approx} instead of κ is that we trade accuracy for speed. While the approximate local node connectivity algorithm often agrees with the result of the *max flow* function, a single discrepancy can lead to an incorrect result for K.

Fortunately, there is a way that we can exploit the efficiency of the approximation while still obtaining an exact result. We note that κ_{approx} always returns a lower bound on the true local node connectivity since the *max flow* algorithm can identify node independent paths that are not necessarily shortest paths. The condition $\kappa_{approx}(G, v, w) \leq \kappa(G, v, w)$ holds for all (v, w) in G.

This means that we can first perform the approximate calculation and then repeat the exact calculation only if κ_{approx} is less than the value currently assigned to K, resulting in a modified algorithm that can leverage the better performance of the approximation while still guaranteed to yield the correct result. This is illustrated below.

```
$K \leftarrow \delta$
select arbitrary node $v_s$ such that $d(v) == K$
for $w$ in $V \setminus \{\{\text{neighbors}(v_s)\} \cup \{v_s\}\}$
    if $\kappa_{approx}(G, v_s, w) < K$
        $K \leftarrow \min(\kappa(G, v_s, w), K)$
for $x, y$ in $\{\text{neighbors}(v_s)\}$ such that $x, y$ non-adjacent
    if $\kappa_{approx}(G, x, y) < K$
        $K \leftarrow \min(\kappa(G, x, y), K)$
```

We define the following conventions used in the remainder of the paper. The *original* algorithm refers to Esfahanian and Hakimi, which exclusively uses κ for the local node connectivity calculations. The *fast* algorithm, described above, uses a combination of κ and κ_{approx} while ensuring the correct result. The *approximate* algorithm (introduced in [15]) relies entirely on κ_{approx} and is not guaranteed to give the correct answer.

The performance benefits depend on the relative costs of evaluating κ_{approx} and κ, along with the number of calls to κ that can be avoided. Let N be the number of calls to κ in the original algorithm. The computational expense is then $N\langle t(\kappa)\rangle$, where $\langle t(\kappa)\rangle$ is the average time needed to evaluate κ. In the best case, where $\kappa_{approx}(G, v, w)$ is greater than or equal to the initial value assigned to K for all node pairs (a, b) that are tested, no calls need to be made to κ and the run time of the new algorithm will be $N\langle t(\kappa_{approx})\rangle$. In the worst case, where $\kappa_{approx}(G, v, w)$ is always less than the running value for K, the run time will be $N\langle t(\kappa_{approx})\rangle + N\langle t(\kappa)\rangle$. In general, $\kappa_{approx}(G, v, w)$ will be less than K for a fraction p of the calls and the computational expense is $N\langle t(\kappa_{approx})\rangle + pN\langle t(\kappa)\rangle$. Note that in order to avoid having to calculate κ, it is not necessary that κ_{approx} return the correct result. Rather, we only require that $\kappa_{approx}(G, v, w)$ be greater than or equal to the running value for K.

Assuming that $\langle t(\kappa_{approx})\rangle$ is small compared to $\langle t(\kappa)\rangle$, the scaling of the fast algorithm with $|V|$, K and δ will be the same as that for the original algorithm but with overall run time multiplied by a factor of p.

There can be significant variation in the value of p for different choices of the starting node v_s. Finding with absolute certainty the best node would incur considerable expense since it entails determining K for each node of minimal degree. Our solution is to consider a subset of the minimal degree nodes and evaluate κ_{approx} between the trial starting node and a random set of non-neighboring nodes. We then choose as v_s the node that satisfies the condition $\kappa_{approx} < \delta$ with the lowest frequency.

```
$G = (V, E)$
for $v$ in $X \subseteq \{x : d(x) == \delta\}$;  where $|X| == n_{trial}$
    $c(v) \leftarrow 0$
for $w$ in $Y \subseteq V \setminus \{\text{neighbors}(v)\} \setminus \{v\}$; where $|Y| == n_{target}$
    if $\kappa_{approx}(G, v, w) < \delta$
        $c(v) \leftarrow c(v) + 1$
$v_s \leftarrow \{v : c(v) == \min(c(v)\}$
```

We want to choose the numbers of trial nodes (n_{trial}) and test evaluations of κ_{approx} for each trial node (n_{target}) to be large enough to obtain a sufficiently small value for p, but not so large that we expend significant computational effort in the search for v_s.

4 Implementation

As our starting point, we use the Esfahanian and Hakimi algorithm as deployed in the *node_connectivity* function of the NetworkX Python package. This enables us to measure the benefits of our modified algorithm in the context of a state-of-the-art implementation and to take advantage of the full NetworkX framework for our benchmarks. The NetworkX version also builds an auxiliary digraph and residual network that are passed as additional arguments to the local node connectivity function, leading to improved performance. The *node_connectivity* function can accept one of several *max flow* functions and we use the current default (Edmonds-Karp in NetworkX 2.1). For the evaluation of κ_{approx} we use the approximate *node_connectivity* function, which can be called after importing the NetworkX approximation module.

The deployment of our new algorithm involves minimal changes to the original version. In the most basic implementation, this only requires the addition of several statements to evaluate $\kappa_{approx}(G, v, w)$ and test whether the result is less than the running value of K. A small amount of extra code allows one to choose between using a particular starting node v_s that is passed as an input argument or automatically selecting v_s from among candidate nodes of minimal degree. We also provide an option that forces the algorithm to simulate the worst case by evaluating κ_{approx} and ignoring the result regardless of how it compares to the running value for K, leading to both κ and κ_{approx} being called for each node pair. Although this option would not be used in practice, it is useful for testing purposes and providing a bound on the worst case.

It should be noted that the NetworkX package specifically or a Python implementation in general will likely have lower performance than an implementation in a compiled language such as C++. We are aware of these limitations, but as described earlier, using NetworkX provides the convenience of being able to work in a complete graph analytics package. The simplicity of our modifications to the original algorithm make them easy enough to employ in an implementation in any language where the user has access to the necessary graph libraries. The performance gains should be comparable to what we see using NetworkX, but the exact speedup will depend on the relative computational expense of the *max flow* and shortest path calculations.

5 Results

Here we compare the performance of our algorithm against the original and approximate algorithms for several types of graphs. We start with a collection of random graphs of varying sizes and densities generated using three different well-known models. We then apply our method to a set of graphs generated from a real-life anonymized social network, starting with the largest embedded 3-component and ending with the largest embedded 7-component. This social network example includes several intermediate graphs generated during the iterative process used to find these embedded components.

In all these benchmarks, the reported times for our method include the overhead associated with choosing an appropriate starting node v_s. We used up to 10 trial nodes (n_{trial}) and 100 target nodes (n_{target}). If there are fewer than 10 nodes with $d(v)$ equal to δ, all minimum-degree nodes are used.

We also address two issues raised earlier. The first regards the dependence of the run time on the choice of the source node v_s. The second concerns the overhead associated with the worst case where κ_{approx} is always less than the running value for K and the evaluation of the exact algorithm for the local node connectivity can never be avoided. All timings are obtained running the benchmarks with Python 3.6.4 and NetworkX 2.1 on a 1.8 GHz Intel Core i5 with 8 GB 1600 MHz DDR3.

5.1 Random Graphs

We present benchmark results for graphs generated using the Barabasi-Albert [17], Erdös-Renyi [18] and Watts-Strogatz [19] models for a range of graph sizes and edge counts in Tables 2-4, respectively. The Barabasi-Albert model generates scale-free networks in which the degree distribution of the nodes follows a power-law of approximately k^{-3}. This is accomplished using a preferential attachment model whereby nodes of higher degree are more likely to be assigned new edges. The Erdös-Renyi model produces random graphs where each potential edge in the graph is independently created with a fixed probability. The Watts-Strogatz model creates graphs having small-world properties by starting with a ring lattice where each node is connected to a given number of neighbors and then rewiring the edges with a given probability. Keep in mind that rerunning the examples will produce different results since new graphs will be randomly generated and there is a variation in run time even when working with the same graphs.

Table 2. Performance of original and fast algorithms for random Barabasi-Albert graphs using seed equal to 1456789356 + n + m. The model parameters n and m are the number of nodes and number of edges from the newly added node to existing nodes as the graph is grown, respectively. $|E|$ is the number of edges in graph, K is the node connectivity, t_{approx}, t_{fast} and t_{orig} are the times taken by approximate, fast and original algorithms, respectively, to find the node connectivity and speedup is the ratio t_{orig} / t_{fast}. All times reported in seconds.

n	m	$\|E\|$	K	t_{approx}	t_{fast}	t_{orig}	speedup
1,000	4	3,984	4	0.10	0.36	15.31	42.53
1,000	6	5,964	6	0.17	0.45	21.06	46.80
1,000	8	7,936	8	0.24	0.75	28.07	37.43
1,000	10	9,900	10	0.32	1.30	31.14	23.95
2,000	4	7.984	4	0.31	0.75	52.31	69.75
2,000	6	11,964	6	0.46	1.08	69.72	64.56
2,000	8	15,936	8	0.63	1.43	98.15	68.64
2,000	10	19,900	10	1.30	1.99	120.26	60.43
4,000	4	15,984	4	0.91	1.67	238.77	142.98
4,000	6	23.964	6	1.30	2.68	332.70	124.14
4,000	8	31,936	8	1.91	3.15	373.17	118.47
4,000	10	39,900	10	2.68	4.86	446.49	91.87
8,000	4	31.984	4	2.88	3.97	851.88	214.58
8,000	6	47,964	6	2.65	4.48	1,170.44	261.26
8,000	8	63,936	8	4.86	6.92	1,470.57	212.51
8,000	10	79,900	10	5.81	8.95	1,747.80	195.28
16,000	4	63,984	4	5.90	8.22	3,496.00	425.30
16,000	6	95,964	6	7.97	11.31	4,692.47	414.90
16,000	8	127,936	8	15.83	20.85	5,941.14	284.95
16,000	10	159,900	10	14.05	19.52	7,250.68	371.45

Table 3. Performance of original and fast algorithms for random Erdös-Renyi graphs using seed equal to 1456789356 + n + 1000*p. The model parameters *n* and *p* are the number of nodes and probability of edge creation, respectively. |*E*| is the number of edges in graph, *K* is the node connectivity, t_{approx}, t_{fast} and t_{orig} are the times taken by approximate, fast and original algorithms, respectively, to find the node connectivity and speedup is the ratio t_{orig} / t_{fast}. All times reported in seconds.

n	*p*	|*E*|	*K*	t_{approx}	t_{fast}	t_{orig}	speedup
1,000	0.01	4,965	3	0.10	0.32	16.86	52.69
1,000	0.02	9,988	8	0.31	0.52	34.84	67.00
1,000	0.04	15,004	13	0.59	1.24	58.35	47.06
2,000	0.01	20,019	6	0.61	1.30	113.60	87.38
2,000	0.02	40,007	21	3.60	5.55	311.78	56.18
4,000	0.03	59,761	35	8.58	13.07	648.37	49.61
4,000	0.01	80,443	18	7.24	10.33	986.54	95.50
4,000	0.02	159,306	51	58.99	64.49	3,648.42	56.57
4,000	0.03	240,767	89	111.60	216.73	11,576.66	53.42

Table 4. Performance of original and fast algorithms for random Watts-Strogatz graphs using seed equal to 1456789356 + *n* + *k*. The model parameters *n* and *k* are the number of nodes and number of neighbors to join in ring topology, respectively. Rewiring probability was set to 0.10 for all graphs. |*E*| is the number of edges in graph, *K* is the node connectivity, t_{approx}, t_{fast} and t_{orig} are the times taken by approximate, fast and original algorithms, respectively, to find the node connectivity and speedup is the ratio t_{orig} / t_{fast}. All times reported in seconds. Incorrect result returned by approximate algorithm marked with *.

n	*k*	|*E*|	*K*	t_{approx}	t_{fast}	t_{orig}	speedup
1,000	5	2,000	2	0.23	0.38	11.00	28.95
1,000	7	3,000	3	0.35	0.42	13.45	32.02
1,000	9	4,000	5	0.50	0.75	18.72	24.96
2,000	5	4,000	2	0.54	1.03	38.35	37.23
2,000	7	6,000	3	0.81	0.84	50.69	60.35
2,000	9	8,000	5	1.20	2.19	67.96	31.03
4,000	5	8,000	2	1.23	1.95	129.42	66.37
4,000	7	12,000	3	2.25	2.48	177.92	71.74
4,000	9	16,000	5	3.73	5.71	215.24	37.70
8,000	5	16,000	2	6.22	7.70	592.69	76.97
8,000	7	24,000	3	6.09*	7.80	699.72	89.71
8,000	9	32,000	4	9.34	8.27	841.16	101.71
16,000	5	32,000	2	10.77	13.24	2,167.33	163.70
16,000	7	48,000	3	15.98	16.77	2,488.67	148.40
16,000	9	64,000	4	23.78	21.84	3,259.02	149.22

For the Barabasi-Albert graphs, we observe speedups ranging from 24x to 425x, with the speedup increasing with the number of nodes and showing a weak dependence on the parameter *m*, which specifies the number of edges from the newly added node to existing nodes during the construction of the graph. For the Erdös-Renyi graphs, we obtained speedups ranging from 47x to 95x, with weak dependence on the graph size and reduced benefits at larger values of the edge creation probability. For the Watts-Strogatz graphs, speedups ranged from 25x to 149x. We obtained noticeably weaker gains for smaller graphs, but this might reflect the overhead in searching for v_s. The performance gains also decrease as the parameter k, which sets the number of neighbors to join in the ring topology, is increased.

Although there is considerable variation in the speedup when applying our algorithm to graphs created using different models and parameters, we consistently observe significant gains of at least one order of magnitude reduction in run time.

For the random graphs, we found that the approximate algorithm nearly always returned the correct result while only taking 30-80% as long as our fast algorithm. While it is tempting to simply rely on the approximate algorithm, there is no guarantee of accuracy as seen in Table 4 and demonstrated in the next section.

5.2 Social Network Example

Table 5 contains the results of benchmarks for subgraphs extracted from a large social network. The performance gains, while still significant, are less dramatic than those obtained when applying our fast algorithm to the random graphs. They range from a 5x speedup for the largest 7-component to 31x for the largest pre-5-component, the penultimate graph generated during the iterative search for the largest 5-component [20]. The fast algorithm did extremely well for the most challenging problem, reducing the time to find the connectivity for the largest 3-component from nearly eight hours to about 17 minutes. Most importantly though, our algorithm produced the correct result whereas the approximate algorithm underestimated the connectivity for seven out of the eight graphs tested.

Table 5. Performance of original and fast algorithms on subgraphs of large social network. Naming convention for graphs: X-comp is the largest k-component in graph and pre-X is the largest graph found in last step of iterative process during search for k-component. K is the true node connectivity and K_{approx} is the connectivity reported by the approximate algorithm; $|V|$ (equivalent to n in the random graph models in tables 2-4) and $|E|$ are the number of nodes and edges; t_{approx}, t_{fast} and t_{orig} are the times taken by approximate, fast and original algorithms; speedup is the ratio t_{orig} / t_{fast}. All times reported in seconds.

graph	K	K_{approx}	$\|V\|$	$\|E\|$	t_{approx}	t_{fast}	t_{orig}	speedup
7-comp	7	5	542	3,704	0.37	2.75	14.20	5.16
pre-7	6	6	573	3,889	0.77	1.02	15.82	15.51
6-comp	6	4	3,089	19,002	13.97	69.59	468.29	6.73
pre-6	5	3	3,636	22,166	12.50	45.51	520.58	11.44
5-comp	5	3	9,864	53,732	38.39	210.85	2653.61	12.59
pre-5	4	3	9,923	53,996	37.02	79.87	2508.79	31.41
4-comp	4	2	19,948	94,630	41.15	518.87	9540.75	18.39
3-comp	3	1	37,938	150,453	109.03	1017.04	28351.48	27.88

5.3 Dependence on Choice of Starting Node

The performance of the fast algorithm depends critically on the frequency with which the exact local node connectivity calculation can be avoided. We found, particularly for the subgraphs extracted from the social network, that this frequency in turn depends on the choice of starting node v_s.

Let $n_\kappa(v_s)$ be the number of evaluations of κ given v_s. Figure 2 shows the run time as a function of $n_\kappa(v_s)$ for the fast algorithm applied to the largest 6-component using all 451 minimal degree nodes ($d = 6$) as starting points. The time varies from 51 seconds to 552 seconds, with the corresponding values of $n_\kappa(v_s)$ equal to 199 and 2,996, respectively. Although the run time tracks closely with $n_\kappa(v_s)$, there is still some variation. For example, using the starting node that maximized $n_\kappa(v_s)$ with a value of 3,079, the run time was only 479 seconds. While long compared to the best cases, this is still considerably less than the longest run time. At the other end of the spectrum, there can be a 30% variation in run time for starting nodes with the same or nearly the same value for $n_\kappa(v_s)$. This spread reflects the fact that the overall run time depends not only on $n_\kappa(v_s)$, but also in the variation in the time needed to execute κ for different node pairs.

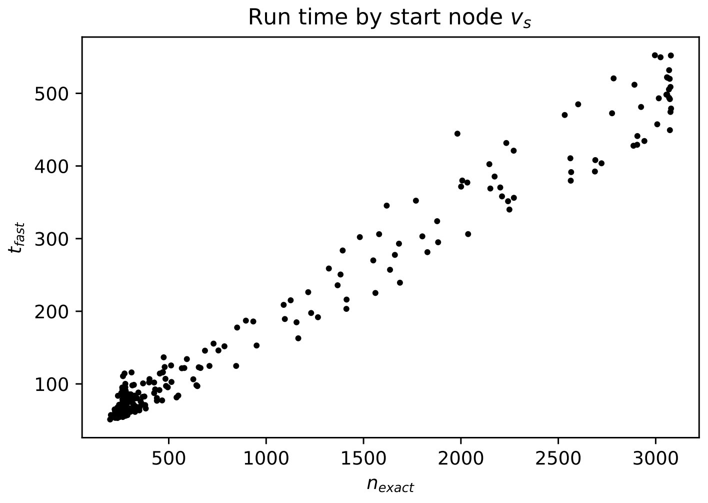

Fig. 2. Dependence of the fast algorithm performance on choice of starting node (v_s) when applied to the largest 6-component in large social network. Each marker represents one of the 451 minimal degree ($d = 6$) nodes. n_{exact} is the number of calls to κ and t_{fast} is the time required by the fast algorithm.

These results reaffirm the importance of carefully selecting v_s. In this particular case, nearly 80% of the starting nodes lead to performance that is at least 4x better than the original algorithm. Evaluating even a small number of choices for v_s before launching the full calculation drastically reduces the likelihood of a poor outcome.

5.4 Worst-case Performance

Here we compare the worst-case performance of our algorithm against the standard algorithm for a collection of random graphs. As noted in the previous section, the choice for v_s affects the run time in two ways – the number of calls that must be made to κ and the variation in the computational expense of evaluating κ for different node pairs. To allow for a fair comparison, each pair of runs using the original and fast algorithms is done using the same choice for v_s.

The results are shown in Table 6. To simulate the worst case, we evaluate κ_{approx} as usual, but then ignore the result and require that κ still be executed. We find that the fast algorithm takes at most a few percent longer than the original algorithm, although in several instances we measured it to be slightly faster. Since the worst case basically involves running the original algorithm plus the overhead associated with the execution of κ_{approx}, this finding can be attributed to the natural variability in runtime.

Table 6. Performance of original and worst-case fast algorithm. K is the node connectivity; $t_{worst\text{-}case}$ and t_{orig} are the times of the worst case for the fast algorithm and original algorithm, respectively. For the worst-case, the fast algorithm was deliberately modified so that the exact local node connectivity was calculated for every node pair that was considered, even if the approximate node connectivity was greater than or equal to the running value for graph node connectivity. For each choice of model and parameters, test was run on five different random graphs. All times reported in seconds.

Model	K	$t_{worst\text{-}case}$	t_{orig}	speedup
Barabasi-Albert (n=2000, m=4)	4	67.25	68.10	1.01
Barabasi-Albert (n=2000, m=4)	4	67.48	66.44	0.98
Barabasi-Albert (n=2000, m=4)	4	67.65	67.19	0.99
Barabasi-Albert (n=2000, m=4)	4	68.08	66.78	0.98
Barabasi-Albert (n=2000, m=4)	4	68.82	67.96	0.99
Erdos-Renyi (n=1000, p=0.03)	13	57.87	57.45	0.99
Erdos-Renyi (n=1000, p=0.03)	15	62.67	60.97	0.97
Erdos-Renyi (n=1000, p=0.03)	12	55.81	55.29	0.99

6 Discussion

The main challenge going forward is improving the selection of the starting node v_s. As we described earlier, finding the optimal choice would generally be prohibitively expensive since it requires running the full algorithm for every minimal degree node. Furthermore, it's not entirely obvious that a minimal degree node is the best option. Esfahanian and Hakimi made this decision in order to minimize the number of calls to κ. Choosing a starting node of higher degree might minimize the frequency with which κ_{approx} is less than the running value for K, but these benefits could be offset by a larger number of calculations. Even within our current scheme, additional progress is possible. Our default is to select 10 trial starting nodes and evaluate κ_{approx} for 100 different non-neighboring nodes. Increasing either n_{trial} or n_{target} improves the odds of finding a good choice for v_s, but also increases the overhead costs. To further complicate the situation, the values that strike the right balance between the extra overhead and likelihood of finding an optimal, or close to optimal, choice are probably dependent on the properties of the graph.

Our work focuses on improving the performance of the general algorithm, but it should be kept in mind that there are several simple steps that should be taken before undertaking the more expensive calculations. Graphs with δ equal to two have a maximum K of two and testing for articulation points – nodes whose removal results in a disconnected graph – quickly identifies graphs with K equal to one.

There are several additional avenues for future work. Our proposed improvements are not limited to the Esfahanian and Hakimi algorithm and can easily be incorporated into other schemes that rely on the *max flow* calculation for determining node connectivity [13, 21] or deciding if the node connectivity is at least k [21, 22]. We are also considering parallelization of the algorithm since the local node connectivity calculations can be done independently. One complication is that determining the validity of the shortest paths-based approximation depends on the running value for the overall graph node connectivity. This could be addressed by periodic synchronization between threads and backtracking when required.

Another possible application is to the more difficult problem of identifying the k-components of a graph. The method described by Moody and White [23] relies on the repeated execution of two computationally demanding steps: determining the connectivity of a graph or subgraph and finding all minimum sized cut sets using Kanevsky's algorithm [24]. Replacing the former with our faster implementation can yield immediate performance gains, although the magnitude will depend on the relative amounts of time spent in the two steps. Our preliminary benchmarks indicate that the improvements are modest, generally around 10-15%, but additional work remains to be done. Nonetheless, given the interest in k-components, especially from researchers in the social sciences [1-6], even small improvements will be welcome.

In conclusion, our new graph node connectivity algorithm effectively leverages the performance of the fast approximation to the local node connectivity (κ_{approx}) while still being guaranteed to give the correct result. Although there is significant variation in performance relative to the original algorithm, depending on the size and complexity of the graph, we find that in every instance there are unambiguous benefits. The worst-case scenario, which we have not encountered in any test and that we can simulate only by forcing κ_{approx} and κ to be calculated for every node pair considered, only results in at most a few percent degradation in performance.

The improved algorithm, test data and Jupyter notebook for running the benchmarks can be downloaded at https://github.com/sinkovit/node-connectivity-fast.

7 Acknowledgments

The author thanks Doug White for many useful discussions and for introducing him to the challenging problems in node connectivity and the identification of k-components in large graphs. Early stages of this work were supported in part by National Science Foundation grants OCI #0910847 Gordon: A Data Intensive Supercomputer and

ACI#1053575 XSEDE: eXtreme Science and Engineering Discovery Environment (XSEDE) through the ECSS program.

References

1. Kreager, D.A., K. Rulison, and J. Moody, Delinquency and the structure of adolescent peer groups. Criminology, 2011. 49(1): p. 95-127.
2. Mani, D. and J. Moody, Moving beyond stylized economic network models: The hybrid world of the Indian firm ownership network. AJS; American Journal of Sociology, 2014. 119(8): p. 1629.
3. Moody, J. and J. Coleman, Clustering and cohesion in networks: Concepts and measures. International Encyclopedia of Social and Behavioral Sciences, 2014.
4. Moody, J. and P.J. Mucha, Portrait of political party polarization. Network Science, 2013. 1(01): p. 119-121.
5. Newman, M.E., Modularity and community structure in networks. Proceedings of the National Academy of Sciences, 2006. 103(23): p. 8577-8582.
6. Porter, M.A., J.-P. Onnela, and P.J. Mucha, Communities in networks. Notices of the AMS, 2009. 56(9): p. 1082-1097.
7. Sporns, O., Graph theory methods for the analysis of neural connectivity patterns, in Neuroscience databases. 2003, Springer. p. 171-185.
8. Esfahanian, A.H., Connectivity algorithms. URL: http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf, 2013.
9. Menger, K., Zur allgemeinen kurventheorie. Fundamenta Mathematicae, 1927. 10(1): p. 96-115.
10. Hopcroft, J.E. and R.E. Tarjan, Dividing a graph into triconnected components. SIAM Journal on Computing, 1973. 2(3): p. 135-158.
11. Kanevsky, A. and V. Ramachandran. Improved algorithms for graph four-connectivity. in Foundations of Computer Science, 1987., 28th Annual Symposium on. 1987. IEEE.
12. Henzinger, M.R., S. Rao, and H.N. Gabow, Computing vertex connectivity: New bounds from old techniques. Journal of Algorithms, 2000. 34(2): p. 222-250.
13. Even, S. and R.E. Tarjan, Network flow and testing graph connectivity. SIAM journal on computing, 1975. 4(4): p. 507-518.
14. Esfahanian, A.H. and S. Louis Hakimi, On computing the connectivities of graphs and digraphs. Networks, 1984. 14(2): p. 355-366.
15. White, D.R. and M. Newman, Fast approximation algorithms for finding node-independent paths in networks. Santa Fe Institute Working Papers Series. Available at SSRN: ssrn.com/abstract_id=1831790, June 29, 2001.
16. Torrents, J. and F. Ferraro, Structural Cohesion: Visualization and Heuristics for Fast Computation. Journal of Social Structure, 2015. 16(8): p. 1-35.
17. Barabási, A.-L. and R. Albert, Emergence of scaling in random networks. science, 1999. 286(5439): p. 509-512.
18. Erdős, P. and A. Rényi, On the strength of connectedness of a random graph. Acta Mathematica Hungarica, 1961. 12(1-2): p. 261-267.
19. Watts, D.J. and S.H. Strogatz, Collective dynamics of 'small-world'networks. nature, 1998. 393(6684): p. 440.
20. Sinkovits, R.S., J. Moody, B.T. Oztan, and D.R. White, Fast determination of structurally cohesive subgroups in large networks. Journal of Computational Science, 2016. 17: p. 62-72.

21. Galil, Z. and G.F. Italiano. Fully dynamic algorithms for edge connectivity problems. in Proceedings of the twenty-third annual ACM symposium on Theory of computing. 1991. ACM.
22. Even, S., An algorithm for determining whether the connectivity of a graph is at least k. SIAM Journal on Computing, 1975. 4(3): p. 393-396.
23. Moody, J. and D.R. White, Structural cohesion and embeddedness: A hierarchical concept of social groups. American Sociological Review, 2003: p. 103-127.
24. Kanevsky, A., Finding all minimum-size separating vertex sets in a graph. Networks, 1993. 23(6): p. 533-541.