Princeton University

Computer Science Department

Implementing 3-Edge-Connectivity and Applications in Essential Infrastructure

Independent Work Report

February 2023 - May 2023

Mike Scornavacca

mas23@princeton.edu

Adviser

Prof. Robert Tarjan

ret@princeton.edu

Abstract

This paper explores the implementation and application of a well known 3-edge-connectivity algorithm. We use Tsin's algorithm to analyze, in linear time, the connectivity of essential networks. This includes state-wide road networks, distributed systems, and even the structure of the Internet. Implementations and applications of 3-edge-connectivity are relatively unexplored, and this paper aims to fill a gap in the surrounding academic literature through rigorous explanation and demonstration.

Contents

1 Introduction

Graph theory plays a significant role in modeling a variety of real-world systems, including transportation systems, social networks, and the Internet. Graph theory provides a medium of analysis that allows for otherwise inaccessible insights into these systems and allows for a unique understanding of their structure and organization.

Connectivity is regarded as one of the most fundamental characteristics of a graph, among many others. It refers to the level of connectivity between a graph's vertices, which affects how accessible each vertex is to every other vertex. It provides a succinct metric for the existence of one or more paths, consisting of some sequence of edges, between any pair of vertices in a given graph. Simply assessing a graph's connectivity can reveal some interesting elements of the graph's structure; however, it fails to elaborate on *how* connected a graph is or, in other words, how vulnerable it is to being disconnected.

There are quite a few metrics that measure the *extent* of a graph's connectivity. The one that will be mainly explored throughout this paper is that of a graph's k-edge-connectivity, more specifically, the extent to which a given graph is 3-edge-connected. The intricacies of this are better explained in section 2. Assessing a graph's k-edge-connectivity (not to be confused with k-vertex-connectivity) provides essential information regarding the graph's resilience to edge deletion. This offers a unique perspective on a given network's redundancy in the face of multiple failures.

This can be applied to distributed systems, road networks, or online social networking platforms. For example, GEANT-2, is "Europe's essential terabit network for research and education interconnects Europe's NRENs and links them to over 100 countries in every region of the world" [4]. Downtime for such a network would be catastrophic, so assessing possible points of failure would be essential for continuous uptime. The layout of GEANT-2 is shown in figure 1.1. Even without the use of a 3-edge-connectivity algorithm, some vulnerabilities to simultaneous edge deletions are visually apparent. This paper will explore the applications of 3-edge-connectivity in situations where the individual components cannot simply be visually inspected.

There exist quite a few theoretically proven algorithms for 3-edge-connectivity but little

to no documentation exists for implementation and application. This paper will explain an in-depth implementation of a well-known linear 3-edge-connectivity algorithm and explore applications and related using available networks [8] [14].

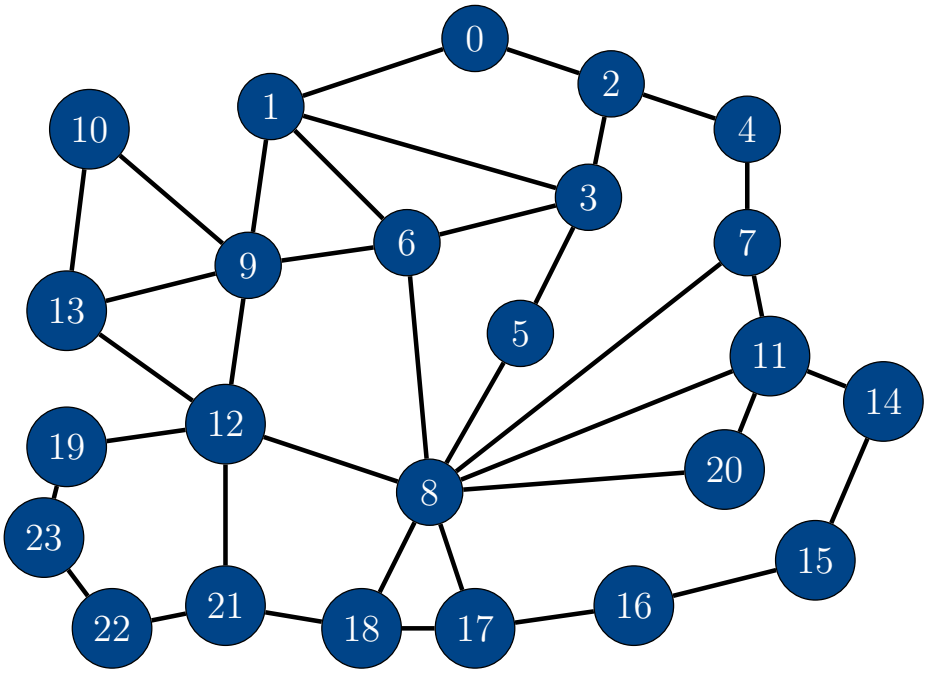


Figure 1.1: The topology of the GEANT2 infrastructure network, which is the pan-European backbone network that supports research and education efforts across the country [1].

The code and implementation are available and thoroughly commented on a public GitHub repository. Instructions for how to set up the associated environment and execute the necessary scripts are succinctly listed in `README.md`.

2 Background

2.1 Definitions

Let an undirected graph, G, consist of a set of vertices, V, and a set of edges, E. In the following diagrams, a vertex is denoted by a circle, and an edge is denoted by a line. An edge always has exactly two endpoints. Two edges that have the same endpoints are considered *parallel edges*. An edge that has both endpoints at the same vertex is called a *self-loop*. A *path* is a sequence of vertices in a graph such that each consecutive pair of vertices in the sequence is connected by an edge.

A *connected component* in G is a subset of V, such that every vertex in this subset is reachable via some set of edges in E to every other vertex in the subset. In the example in figure 2.1, it is not possible to get from a blue vertex to a yellow vertex via any set of edges, guaranteeing their existence in separate connected components. However, every blue vertex is reachable from every other blue vertex, and the same for all yellow vertices.

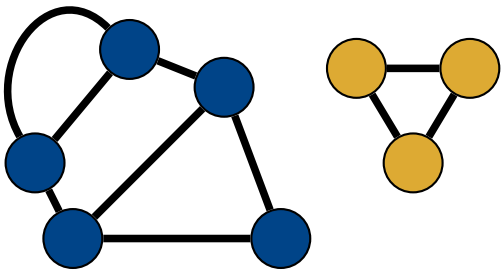


Figure 2.1: An example graph of *two* connected components, where the **blue** vertices are a connected component, and the **yellow** vertices are another connected component.

A 2-*edge-connected component* in G is a maximal subset of V, such that every vertex in this subset is reachable via at least two edge-distinct paths in E to every other vertex in the subset. That is, removing any **single** edge in G cannot possibly disconnect the vertices in a 2-edge-connected component. This results from the fact that removing 1 edge can only possibly break one of the two viable paths established by the definition of 2-edge-connectivity. This is visible in figure 2.2, which demonstrates a simple, 4-vertex 2-edge-connected component (shown on

the far left). Removing any edge in the graph (shown by the dashed lines) does not disconnect the graph, demonstrating its 2-edge-connectivity.

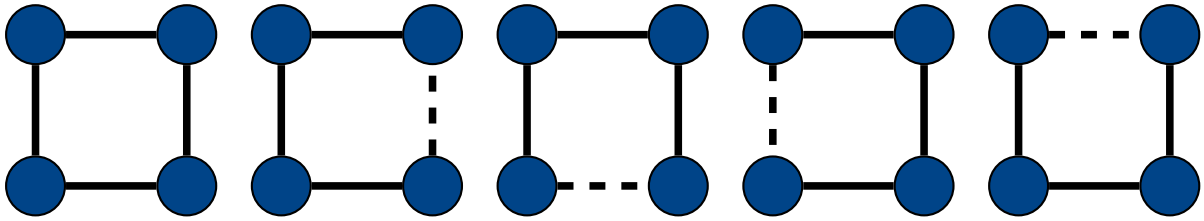


Figure 2.2: An example of a 2-edge-connected graph. Given the graph on the left, removing any edge does not disconnect any pair of vertices.

A 3-*edge-connected component* follows from the definition of a 2-edge-connected component. A 3-edge-connected component in G is a maximal subset of V, such that every vertex in this subset is reachable via at least **three** edge-distinct paths in E to every other vertex in the subset. That is, removing any **two** edges in G cannot possibly disconnect the vertices in a 3-edge-connected component. This results from the fact that removing two edges can only possibly break two of the three viable paths established by the definition of 3-edge-connectivity. These are the types of components that will be analyzed over the course of this paper. In figure 2.3, two distinct 3-edge-connected components exist. Removing any two edges in E cannot possibly disconnect the vertices within the same 3-edge-connected component. However, there are two edges that can be removed that can disconnect the two components, which shall be called *critical edges*. It is due to these critical edges that figure 2.3 is not a single 3-edge-connected component. Notice that the 3 edge-distinct paths that guarantee 3-edge-connectivity simply need to be from E, not necessarily the edges connecting the vertices within the component.

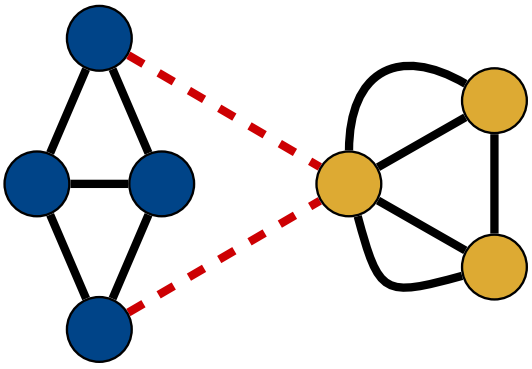


Figure 2.3: An example of two 3-edge-connected components, denoted by blue and yellow vertices. The two edges that can be removed to disconnect the two components are denoted by red-dashed lines (critical edges).

Components are *vertex-distinct*. That is, a given vertex, u, is guaranteed to be in exactly *one* component and cannot be shared by *two* or more components.

To abstract this away from graph theory and demonstrate a possible application, the following is a concrete example in the context of roadways:

> Suppose a set of roads and intersections. Intersections join multiple roads, and roads connect intersections together. This creates a graph (G), consisting of a set of vertices (V) and a set of edges (E) where the vertices are intersections and the edges are roads that connect these intersections.
>
> The set of intersections is connected if from *any* intersection, paths of roads exist to travel to any other intersection.
>
> The set of intersections is 2-*road-connected* if, from *any* intersection, there exist *two* road-independent paths to every other intersection. That is, any road can be demolished, and all intersections are still guaranteed to be connected.
>
> The set of intersections is 3-*road-connected* if, from *any* intersection, there exist *three* road-independent paths to every other intersection. Any **two** roads can be demolished, and all intersections are guaranteed to be connected.

2.2 Algorithm

2.2.1 Depth-First Search (DFS)

Tsin's algorithm takes advantage of a well-known process called depth-first search (DFS). It is a fundamental algorithm to graph exploration and involves starting at a given vertex and exploring a particular branch as long as possible before backtracking. The recursive version of the algorithm (as used in Tsin's algorithm) is as follows, beginning with picking a starting vertex, u, to begin the depth-first search and calling `dfs(u)`:

1. Mark u as visited.
2. Call `dfs(v)` on all unvisited neighbors of u.

If all neighbors are visited, then the algorithm will backtrack to the previous vertex and begin to explore its unvisited neighbors. This repeats until all vertices have been explored. The *pre-order number* of a vertex, u, is the order in which u was marked as visited. A *tree-edge* is an

edge, $u \to v$, that was traversed from u in order to reach an unvisited vertex, v. A *non-tree-edge* is simply any other edge that is not a tree-edge, that is, it was not traversed in order to reach an unvisited vertex during DFS. The *low* value of a vertex, u, is the lowest pre-order number reachable by any number of tree edges, followed by, at most, one non-tree edge.

2.2.2 Absorption (Contraction)

A few essential operations are critical for understanding Tsin's algorithm intuitively and the challenges surrounding its implementation [14]. It is encouraged for a full understanding of this paper that Tsin's algorithm is thoroughly reviewed. The main graph modification operation that is used is `absorb()`[1]. This involves a given vertex "absorbing" another vertex and gaining all of its incident edges. The endpoints of the incident edge change, but the algorithm is dependent on the "embodiments" of edges, which is best explained via an example. In figure 2.4, v_1 is being absorbed into v_2. The orange edge is between v_1 and v_3, and the blue edge is between v_1 and v_4. The red dotted edge between v_1 and v_2 will be deleted as it becomes a self-loop in the `absorb` operation. **After the absorb operation**, the blue edge between v_2 and v_4 is the *embodiment* of the original blue edge between v_1 and v_4; the orange edge between v_2 and v_3 is also the *embodiment* of the original orange edge between v_1 and v_3. Since Tsin's algorithm relies on storing data on the edges (such as whether or not a given edge is a tree edge), it is necessary to track the embodiments of edges throughout `absorb()` operations.

2.2.3 Tsin's Algorithm

Tsin's algorithm leverages the ordering of depth-first search in order to find 3-edge-connected components. The algorithm is inherently very complex, but an intuitive, high-level explanation will be provided in order best to assist the understanding of the rest of this paper.

Each vertex, u, is associated with a corresponding 3-edge-connected component, C_u, which is a set of vertices. The algorithm begins with the original graph, and each C_u is initialized to a single vertex each, u. Vertices in a set C_u are *guaranteed* to be in a 3-edge-connected component with other vertices in the same C_u and will never be removed throughout the course of the algorithm. Calculating the correct C_u for every vertex involves maximally joining together components through a process called "absorption" (as mentioned above). Notice that all C_u will be disjoint sets.

[1] In graph theory, this operation is commonly referred to as contraction.

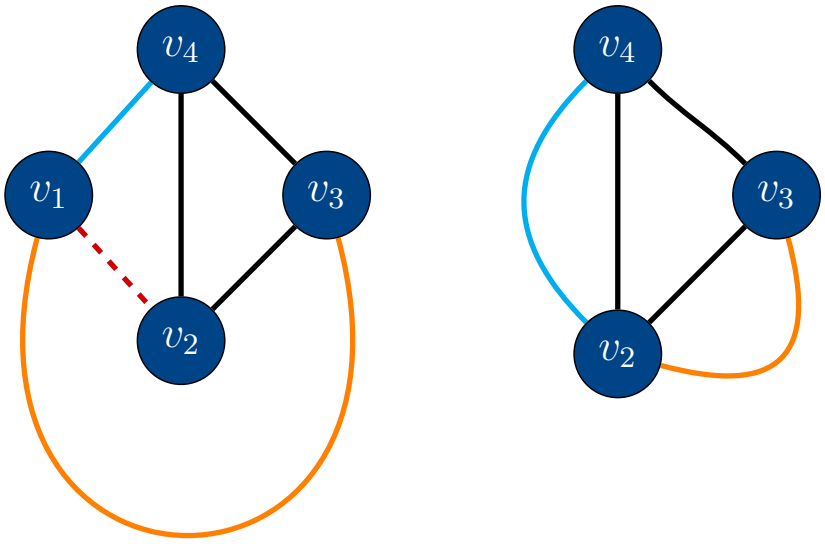


Figure 2.4: An example of the absorb operation, where v_1 is absorbed by v_2. The red-dotted edge becomes a self-loop and is removed, and the orange and blue edges become parallel edges with the corresponding existing edges.

The absorption process has two possible operations: (1) absorb and (2) absorb-eject. In (1) absorb, the vertices, u and v, are merged, and their corresponding components, C_u and C_v, respectively, are combined (indicating that C_u and C_v are part of the same 3-edge-connected component). In (2) absorb-eject, the vertices, u and v, are merged, but v is "ejected" with no incident edges, and their corresponding components, C_u and C_v respectively, are *not* combined (indicating that C_v is a final 3-edge-connected component, and will never be absorbed in the future). As a result, a vertex, v, is only *absorbed* into another vertex, u, if there is an absolute certainty that all of the vertices in C_v and all of the vertices in C_u are a part of the same 3-edge-connnected component; and a vertex, v, is only *absorb-ejected* if there is an absolute certainty that C_v contains all of the correct vertices in its 3-edge-connected component.

The absorb-path operation (as listed on lines 32-40 of appendix A) calls (1) absorb consecutively on a list of vertices, $[x_0, x_1, ..., x_i]$. It absorbs x_1 into x_0, then x_2 into x_0, and so on until x_i is absorbed into x_0.

2.3 Datasets

This paper stands to offer both a thorough explanation of implementation as well as a demonstration of possible applications of theoretical algorithms. There are three datasets being used for the analysis of infrastructure, all obtained from the Stanford Large Network Dataset Collection (SNAP) [8]:

1. The Gnutella peer-to-peer network, where vertices are computers/clients and edges are queries/query hits [2].

2. The road networks of Pennsylvania, where vertices are intersections/endpoints and edges are connecting roads.

3. The topology of the internet as obtained from traceroutes in 2005, where vertices are any device with an IPv4 address and edges are the paths of IP packets. An example traceroute can be seen in figure 2.5.

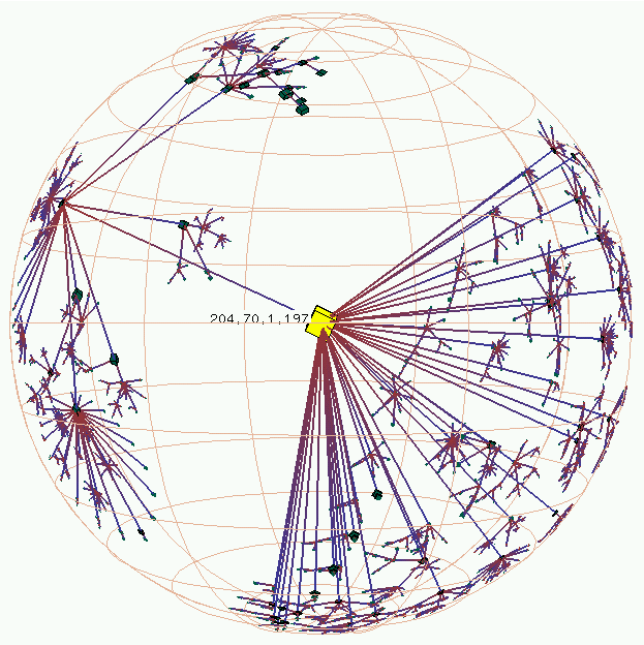


Figure 2.5: An example traceroute from the Skitter tool [10].

3 Related Work

Connectivity, specifically k-edge-connectivity and k-vertex-connectivity, are both calculable metrics available for measuring a graph's connectivity. However, linear time algorithms are only available for $k \leq 3$, for both k-edge-connectivity and k-vertex-connectivity problems.

A linear time algorithm for 2-vertex-connectivity was originally proposed by Tarjan [12]. It, and the algorithms proposed to follow, all utilize depth-first search. This algorithm, in turn, also solves the 2-edge-connectivity problem. A linear time algorithm for 3-vertex-connectivity was proposed by Hopcroft and Tarjan [6]. It was the first to accomplish this problem in linear time and was fundamental to the academic literature surrounding k-connectivity. It, however, suffered from a few critical errors that were later resolved by Gutwenger and Mutzel [5].

The first solution to 3-edge-connectivity, as will be the type of algorithm mainly discussed throughout this paper, was a reduction to Hopcroft and Tarjan's 3-vertex-connectivity conducted by Galil and Italiano [3]. Despite its correctness, the algorithm was rather complicated and very difficult to implement. It also adopted the issues from Hopcroft and Tarjan's algorithm 3-vertex-connectivity algorithm. There were many that improved the simplicity of the algorithm, but all required multiple passes of DFS, resulting in a lot of overhead and thus increasing the difficulty of implementation [9] [11].

Tsin was not the first to propose a linear time algorithm for 3-edge-connectivity, but his algorithm is among the most efficient and elegant [14]. It only uses one pass through DFS and only one main graph manipulation function, `absorb`. It is roughly 30 lines of pseudo-code, but there is no documentation of implementation beyond the pseudo-code.

There has been little to no available work on the implementation or applications of *any* 3-edge-connectivity algorithm. A few public repositories on GitHub have implemented some 3-vertex-connectivity algorithms, but the implementations are rather undocumented and outdated [7] [15]. These implementations have little to no code for testing correctness and do not provide any demonstration for application.

4 Approach

1. **Implement Tsin's algorithm [14].**

   This involves writing an accessible set of Python scripts that allow for Tsin's algorithm to be run on a vanilla Python dictionary of lists. Since the original algorithm proposed by Tsin is recursive, which will be limited by the size of the recursive stack, this step also involves revising the original algorithm to be iterative.

2. **Test on known graphs.**

   Since there are no other implementations to test against, the most efficient way to roughly test for the correctness of the algorithm is to run it on graphs with known results. On smaller graphs, it is rather simple to identify 3-edge-connected components, and there are a few established examples in Tsin's paper [14]

3. **Analyze varying infrastructures' 3-edge-connectivity.**

   Using the Stanford Network Analysis Project (SNAP), a wide collection of interesting networks is available to analyze [8]. In order to demonstrate the possibility of applications, a set of networks from a wide variety of disciplines will be used. This includes three main large datasets: (1) the road network for the state of Pennsylvania, (2) the topology of the peer-to-peer file sharing protocol, Gnutella, and (3) the topology of the entire internet from 2005 [2].

5 Implementation

> All code is available on this public GitHub repository. All explanations and comments are located in the files themselves.

5.1 API

There are two classes, `ThreeEdgeConnectRecursive` and `ThreeEdgeConnectIterative`, which use the recursive and iterative versions of Tsin's algorithm respectively. The input is an undirected graph, which is a vanilla Python dictionary, where the key is a vertex, and the value is a list of its adjacent vertices. The 3-edge-connected components are then calculated in the constructor of the implementation class. They can be retrieved via a call to `get()` and are returned in a list of sets, where each set is a 3-edge-connected component. An example use of `ThreeEdgeConnectIterative` is shown below:

```
from triconnect.edge.iterative import ThreeEdgeConnectIterative
from triconnect.edge.recursive import ThreeEdgeConnectRecursive

# Key is the vertex, and value is list of adjacent vertices
graph: Dict[int, List[int]]

implementation = ThreeEdgeConnectIterative(graph)
# or implementation = ThreeEdgeConnectRecursive(graph)

# Get components from implementation
components: List[Set[int]] = implementation.get()
```

Both classes inherit from `ThreeEdgeConnectBase`, which defines common functions between the two implementations.

5.2 Recursive

The original 3-edge-connectivity algorithm, as defined by Tsin, relies heavily on recursion as its medium for depth-first search [14]. That is, instead of manually defining a stack, Tsin's algorithm takes advantage of the recursive call stack in order to achieve DFS.

5.2.1 Challenges

1. **Representation of edges**
   In Tsin's algorithm, edges can have properties. If they are the edge used to reach a vertex for the first time, they are marked as a tree edge. Traversing a tree edge versus a back edge have different implications in the algorithm, so it is necessary to know if the edge in question is a tree or a non-tree edge. Since parallel edges are allowed, this makes it quite difficult to simply store the values of the two endpoints to denote a tree edge. This dilemma is clearly apparent in figure 5.1, where there are three parallel edges from v_2 to v_3, but only one of them is marked as a tree edge. Thus, edges must be *objects* such that they can each store data.

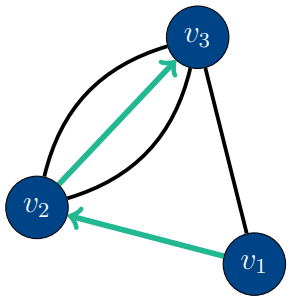


Figure 5.1: An example of a DFS traversal in a three vertex network with parallel edges. The root of the tree is at v_1, and then to v_2, and finally to v_3. Notice how only one edge from v_2 to v_3 is marked as a tree edge.

2. **Embodiments of edges**
   Whilst iterating through a vertex's incident edges, the algorithm mandates that the graph is modified. Adding, removing, and changing edges whilst iterating through a list is largely problematic and can cause issues; thus, most languages throw an error when doing so. Due to the recursive nature of this implementation, it is simply impossible to change the graph in the middle of the iteration. So for every edge, there is an *embodiment* of that edge that is changed when the graph is modified. The process is to iterate through the

original graph, and prior to traversing any edge, getting the *embodiment* of that edge and visiting that vertex instead. This is done through a disjoint set union with path compression[2]. Note that the original edges store the properties of the embodied edge, as shown in figure 5.2.

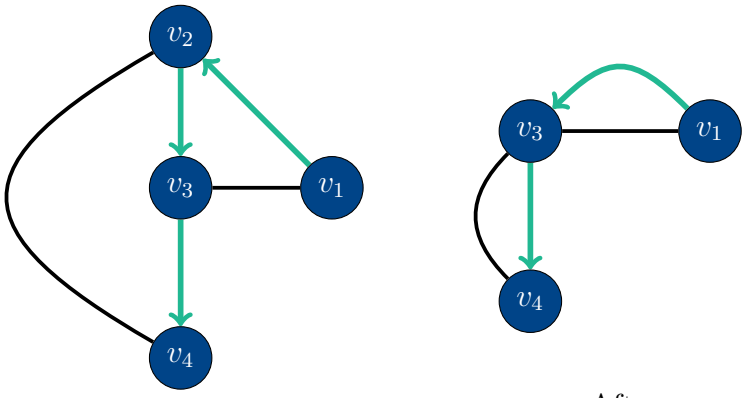


Figure 5.2: An example of an *absorb* of v_2 into v_3, and the embodiment of tree edges still being maintained in the resulting graph.

3. **Size of recursive stack**

   As the size of V approaches $10,000$, Python's default recursive stack size of $1,000$ is easily exceeded. This problem can be quickly patched by simply increasing the recursive stack size, but this quickly becomes an issue as the recursive stack consumes more and more memory. If the recursive stack size is set too high, the algorithm quickly encounters segmentation faults. For inputs where $|V| > 1,000,000$, it becomes unreasonable to have a recursive stack size large enough to support the algorithm. This essentially holistically prevents the recursive implementation from being run on large datasets (such as the Pennsylvania road network, which has $1,088,092$ vertices).

[2]Note that the runtime of the disjoint set union that was used is not truly constant and has runtime proportional to the Ackermann function, but the difference is negligible. $O(1)$ implementations do exist, but they are extremely complicated to implement but have been proven in COS423 here at Princeton University [13].

5.3 Iterative

As mentioned above, a fatal flaw of the recursive implementation is its innate use of the recursive stack. In order to use Tsin's algorithm on larger and more interesting datasets, the use of the call stack needs to be extrapolated to use an explicit stack data structure. DFS can be written both recursively and iteratively using these methods, however, rewriting this algorithm to support large datasets comes with some difficult challenges.

5.3.1 Challenges

1. **Replication of recursive functionality**
   The recursive implementation is *significantly* easier to understand, and there are quite a few challenges in switching the algorithm from recursive to iterative. One, in particular, has to do with the nature of depth-first search. Once all adjacent vertices have been visited from a vertex, u, Tsin's algorithm will back-track along the tree-edge, $u \leftrightarrow v$, and execute a function from u regarding the properties of v, which we will call the *post-analysis* of u from v (located on **10-20** of appendix A). In figure 5.3, v_1 will need to execute this "post-analysis" 4 times in total, one for each v_2, v_3, v_4, v_5. Normally, when conducting DFS iteratively, the current vertex is popped from the top of the stack upon visiting, and all of the unvisited neighbors are added to the top of the stack. However, in order to accomplish Tsin's algorithm iteratively, this implementation will *not* pop anything from the top of the stack unless all "post-analysis" operations have run. It checks whether or not it needs to execute post-analysis every time v_1 is on top of the stack. If all edges have been explored from v_1 (and thus this "post-analysis" has been called 4 times), only then will it finally be popped from the stack.

2. **Dynamic modification of graph structure**
   In order to maintain $O(n)$, it is no longer possible to use the disjoint set union structure used for the recursive implementation since vertices can be at the top of the stack "multiple" times, as mentioned above in item (1). Whenever looking for the next "unexplored" vertex in the neighbors of u, it is simply too inefficient to iterate through all of the neighbors again and skip over those already explored. This was not an issue for the recursive implementation, as the natural ordering of the call stack mandated that an edge, $u \leftrightarrow v$, would only ever be looked at once. Thus, the graph must be modified whilst iterating in

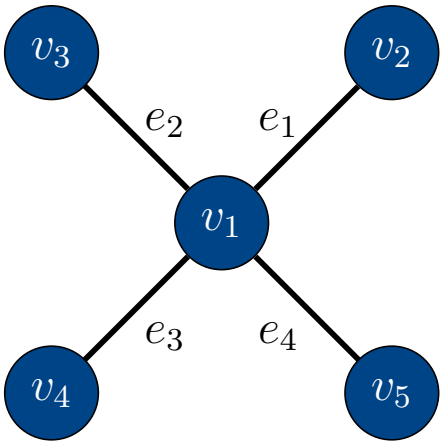
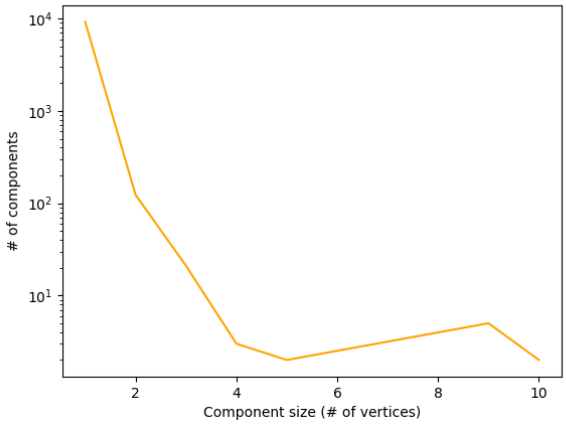


Figure 5.3: A simple graph structure, in which the root is at v_1, and DFS travels to v_2, v_3, v_4, and then v_5.

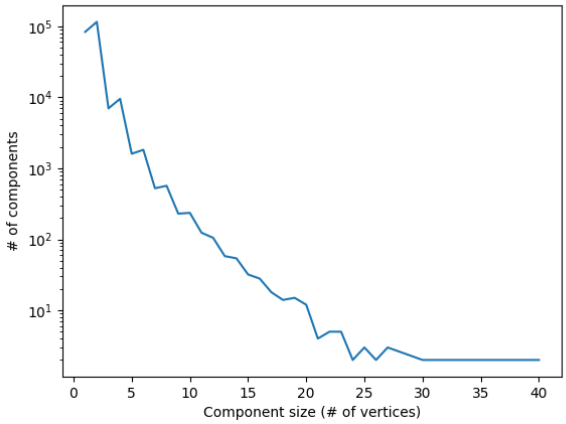
order to maintain linear runtime. For every edge, there is a single mutable `Edge(u,v)` object created, which is stored in the incident edges of u ($I(u)$) and the incident edges of v ($I(v)$). When an edge is "redirected," let's say an edge $u \leftrightarrow v$ becomes $u \leftrightarrow x$, two actions occur: (1) the changed endpoint (v in this case) is modified to x in `Edge(u,v)` and (2) it is removed from its respective list incident edges ($I(v)$ in this case) and moved to its changed endpoint's list ($I(x)$ in this case). This mandates that the embodiments of edges are maintained, and the object `Edge(u,v)` is always and solely accessible from both u and v through their incident edge lists, $I(u)$ and $I(v)$ respectively.

6 Results

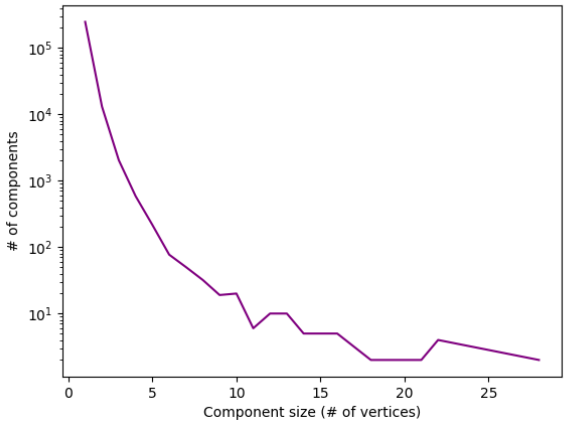
The 3-edge connectivity of infrastructure, no matter how essential, is *not* guaranteed. Albeit a large proportion of the analyzed networks are 3-edge-connected, there was a surprisingly significant portion that could easily be disconnected via the removal of two or more edges (whether that be roads, network connections, etc.).



(a) Gnuetlla P2P



(b) Pennsylvania road network



(c) 2005 Internet

Figure 6.1: (x-axis): Number of vertices in the component. (y-axis): Number of components of this size. Categories with only one component were excluded in the pursuit of more understandable plots.

We can see in figure 6.1, that most components have a relatively small number of vertices[3]. Components of size one are generally endpoints (in the context of road networks, a dead-end), and normally not even 2-edge-connected.

In table 6.1, there is evidence that the majority of the provided networks are in the same 3-edge-connected component. We calculate two essential metrics: **proportion** and **core proportion**. Proportion is calculated by the proportion of the network that is covered by the largest component. However, this includes numerous components that are only of size one, and as mentioned above, these are likely endpoints in the original graph. Thus, we calculate **core proportion**, which is the proportion of the network, *excluding* components of size one, covered by the largest component. This will offer a metric that is more accurate as to the connectivity of the graph as a whole.

Dataset	**Results**			
	Size	**Largest component size**	**Proportion**	**Proportion (core)**
Gnutella	62586	52931	0.846	0.992
PA Roads	1088092	675359	0.621	0.672
2005 Internet	1696415	1411056	0.832	0.973

Table 6.1: A table that emphasizes the proportion of the graph that are a part of the largest component.

We notice that the core proportion of the peer-to-peer file-sharing system, Gnutella, of 0.992 is incredibly high. Indicating that besides endpoints (devices solely involved in *receiving* files, rather than sending them, or vice versa), the network as a whole is incredibly resistant to dual simultaneous edge failure. The Skitter dataset also noticed similar proportions. Besides the endpoints (sole, unidirectional clients), the internet (as seen by these trace routes) as a whole is also incredibly resistant to dual simultaneous edge failure.

However, in contrast to this, the Pennsylvania road networks have a significantly lower proportion and core proportion (0.621 and 0.672, respectively) than that of the Gnutella system. Past the largest component of size 675359, what was incredibly surprising was the next largest component size was only a *little less than* 100. This may be a result of how the road networks were translated into a quantitative graph format, but assuming accuracy on the dataset, it is concerning how easily particular parts of the state could be entirely disconnected.

[3]Note that the largest components are excluded from the graph, and are referenced in table 6.1.

7 Conclusions and Future Work

Despite the thorough research around the theory of 3-edge-connectivity and 3-vertex-connectivity, this paper contributes a few missing aspects regarding implementation and application:

1. An open-source implementation of Tsin's simple 3-edge-connectivity algorithm with pre-processed datasets.

2. The first demonstration of application for such 3-edge-connectivity algorithms and commentary on the connectivity of established networks.

The main goal of this paper was not to underline the need for greater connectivity in infrastructure but rather to propose an easily understood implementation and demonstrate its possible applications. This paper was never intended to present new groundbreaking truths on these pieces of infrastructure, yet still emphasize how the implementation of theoretical algorithms could be beneficial to analyzing essential networks in everyday life.

Future work largely lies in the availability of datasets. Due to the short nature of this project, collecting extensive data on large networks of $1,000,000+$ vertices was simply not feasible. Thus, the demonstration of the application was largely dependent on the availability of network datasets in SNAP [8]. Given more time and greater access to resources, it would be interesting to possibly explore a tool that allows this algorithm to be dynamically run on a network, assessing vulnerabilities as failures occur. Given the linear runtime of the algorithm, it would provide an efficient yet comprehensive analysis of the integrity and resilience of the network. A combination of a distributed snapshot algorithm to gather the status of vertices and edges and then performing this locally on one of the nodes could be a viable approach to this.

Python is not the most *efficient* choice in terms of language and was largely chosen due to its ease of use. It would be beneficial for performance to reprogram this implementation in a more efficient, compiled language such as *C++* or *Go*.

A Tsin's Algorithm

This is a commented and line-numbered version of Tsin's 3-edge-connectivity algorithm. The algorithm listed in Tsin's paper has a few alignment and indentation issues, and for the purposes of this paper, it is significantly easier to explain if it is rewritten. The proof of correctness can be seen in Tsin's paper [14].

```
1:  count ← 1;                                   ▷ Initialize preorder number to be 1
2:  DFS(root)                                    ▷ Begin algorithm from any vertex
3:  procedure DFS(u)
4:      pre(u) ← count
5:      count ← count + 1
6:      low(u) ← pre(u); P_u ← u
7:      for v ∈ N(u) do                          ▷ Go through all adjacent vertices to u
8:          if v is unvisited then
9:              DFS(v)                           ▷ Explore unvisited vertex
10:             if degree of v is 2 then
11:                 G ← G/(u ↔ v)                ▷ Absorb then eject u, thus making it (and σ(u)) as
                                                   a 3-edge-connected component of G
12:                 P_v = P_v − v
13:             end if
14:             if low(u) ≤ low(v) then
15:                 absorbPath(P_v)
16:             else
17:                 low(u) ← low(v)
18:                 absorbPath(P_u)
19:                 P_u ← u + P_v
20:             end if
21:         else if u ↔ v is not a tree edge then
22:             if pre(v) < low(u) then
23:                 absorbPath(P_u)
24:                 low(u) ← pre(v)
25:                 P_u ← u
26:             else if pre(u) < pre(v) then
27:                 absorbPath(P_u[u...v])       ▷ Partially absorb P_u, only from where u is in
                                                   the path up until v.
28:             end if
29:         end if
30:     end for
31: end procedure
```

```
32: procedure absorbPath(P)
33:     if P is not null then
34:         Let P = x_0, x_1, ..., x_k                ▷ x_0, ..., x_i are all connected via tree edges.
35:         for i ∈ [1, ..., k] do
36:             G ← G/(x_0 → x_i)                     ▷ Absorb x_i into x_0, where the edge x_0 ↔ x_i is the
                                                        embodiment of x_{i-1} ↔ x_i.
37:             σ(x_0) ← σ(x_0) ∪ σ(x_i)              ▷ "Adopt" x_i's list into x_0's list.
38:         end for
39:     end if
40: end procedure
```

Bibliography

[1] Fernando Barreto, Emilio Wille, and Luiz Nacamura. "Fast Emergency Paths Schema to Overcome Transient Link Failures in OSPF Routing". In: *International Journal of Computer Networks Communications* 4 (Apr. 2012). DOI: 10.5121/ijcnc.2012.4202.

[2] Fernando Bordignon and Gabriel Tolosa. "Gnutella: Distributed System for Information Storage and Searching Model Description". In: (Aug. 2001).

[3] Zvi Galil and Giuseppe F. Italiano. "Reducing edge connectivity to vertex connectivity". en. In: *ACM SIGACT News* 22.1 (Mar. 1991), pp. 57–61. ISSN: 0163-5700. DOI: 10.1145/122413.122416. URL: https://dl.acm.org/doi/10.1145/122413.122416 (visited on 04/25/2023).

[4] *GÉANT*. May 2019. URL: https://geant.org/ (visited on 04/25/2023).

[5] Carsten Gutwenger and Petra Mutzel. "A linear time implementation of SPQR-trees". In: *Graph Drawing* (2001), pp. 77–90. DOI: 10.1007/3-540-44541-2_8.

[6] J. E. Hopcroft and R. E. Tarjan. "Dividing a Graph into Triconnected Components". en. In: *SIAM Journal on Computing* 2.3 (Sept. 1973), pp. 135–158. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0202012. URL: http://epubs.siam.org/doi/10.1137/0202012 (visited on 04/24/2023).

[7] Chenyang Huang. *TriconnectedComponent*. 2016. URL: https://github.com/chenyangh/TriconnectedComponent.

[8] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[9] Hiroshi Nagamochi and Toshihide Ibaraki. "A linear time algorithm for computing 3-edge-connected components in a multigraph". en. In: *Japan Journal of Industrial and Applied Mathematics* 9.2 (June 1992), pp. 163–180. ISSN: 0916-7005, 1868-937X. DOI: 10.1007/BF03167564. URL: http://link.springer.com/10.1007/BF03167564 (visited on 04/26/2023).

[10] *Skitter data using Hypviewer: Internet Atlas Gallery*. Aug. 2019. URL: https://www.caida.org/projects/internetatlas/gallery/hypskit/.

[11] Satoshi Taoka, Toshimasa Watanabe, and Kenji Onaga. "A Linear-Time Algorithm for Computing All 3-Edge-Connected Components of a Multigraph". In: *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences* E75-A.3 (Mar. 1992), pp. 410–424. ISSN: 0916-8508. (Visited on 04/26/2023).

[12] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". en. In: *SIAM Journal on Computing* 1.2 (June 1972), pp. 146–160. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0201010. URL: http://epubs.siam.org/doi/10.1137/0201010 (visited on 04/24/2023).

[13] Robert E. Tarjan and Jan Van Leeuwen. "Worst-case Analysis of Set Union Algorithms". en. In: *Journal of the ACM* 31.2 (Mar. 1984), pp. 245–281. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/62.2160. URL: https://dl.acm.org/doi/10.1145/62.2160 (visited on 04/27/2023).

[14] Yung H. Tsin. "A Simple 3-Edge-Connected Component Algorithm". en. In: *Theory of Computing Systems* 40.2 (Feb. 2007), pp. 125–142. ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-005-1269-4. URL: http://link.springer.com/10.1007/s00224-005-1269-4 (visited on 04/24/2023).

[15] Yung H. Tsin. "Decomposing a Multigraph into Split Components". In: *CRPIT* 128 (). DOI: 10.5555/2523693.2523694. URL: https://dl.acm.org/doi/pdf/10.5555/2523693.2523694.