Studying and Visualizing Contagion on Graphs

Connor Anderson, Akshaj Balasubramanian, Henry Poskanzer

Worcester Polytechnic Institute

Advisors:

Gábor N. Sárközy
Daniel Reichman

A Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for
the Degree of Bachelor of Science in Computer
Science and Mathematical Sciences.

Abstract

In this report, we investigate the NP-complete minimum contagious set problem in bootstrap percolation. We present the following: three new bounds providing approximations for the size of minimum contagious sets, efficient and optimal algorithms for special types of graphs, and an optimal algorithm for any graph.

Furthermore, we created a publicly available software product named 'Booper', a web application that can help mathematicians research and visualize bootstrap percolation. Users of Booper are able to upload a custom graph, find the minimum contagious set of a graph, perform iterations of bootstrap percolation with user-defined threshold and probability parameters, and much more.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recently, there has been increasing interest in learning about spread in a network, whether it be a contagious disease like COVID-19 in a city or fake news in a social media network. There have been a multitude of mathematical models and processes developed to quantify, analyze, and understand the spread of contagious processes over networks. One such approach is to use graph theory to model and study a process on a network. By representing a social network as a graph, we can use existing results in graph theory to formulate new insights into real-world applications.

In our project, we used this approach in studying bootstrap percolation. Bootstrap percolation is a process which was originally invented by physicists to study magnetism [1]. The field has grown since, and nowadays it has been used to study viral marketing [2] and gossip spread in social networks [3]. We believe that it could also be used to study pandemics. The problems we studied have implications in the three aforementioned applications, and potentially more. One such problem of major interest in bootstrap percolation is finding the minimum contagious set: the smallest initially infected group of people needed to infect their whole network. We studied the minimum contagious set problem across a variety of graphs.

Given that these topics are actively used in real-world applications, it is important that their associated problems are solved efficiently. The topics we addressed in this project are constrained by the currently known limits of computational complexity. The minimum contagious set problem is NP-hard [4]; most computer scientists believe it is impossible to find an efficient and optimal solution that works for all graph inputs. Therefore, solutions to this problem can be efficient, optimal, or generic —they cannot have all three of these desirable qualities. However, we can make incremental improvements by writing new algorithms and theorems that achieve two of these desirable qualities.

In this project, we did precisely that. First, we created efficient, optimal algorithms for certain special types of graphs. Second, we proved new upper bounds that approximate the size of a minimum contagious set for any graph. Third, we created optimal algorithms which find a minimum contagious set of any graph at the cost of efficiency. In addition, we considered how our proof techniques can be applied to other topics in graph theory: independent sets and k-degenerate subgraphs. Then, we implemented our new bounds and algorithms. We used this code to run empirical tests on random graphs generated by the

Erdős-Rényi model, as well as graphs representing real-world networks, determining how much our results improve upon previously known results. One of our new bounds turned out to be a major improvement on both random graphs and real-world graphs.

Bootstrap percolation is a very active research field. To further aid this research, in addition to the above theoretical and practical results, we developed Booper, a web application that can help users visualize bootstrap percolation. Booper can display a user-defined graph and seed set, and show which vertices are activated in each iteration of the bootstrap percolation process. It can also compute minimum contagious sets and contagious sets from a greedy algorithm. As far as we know, there is currently no other application for visualizing bootstrap percolation; we filled this gap in the available software. Once we made Booper, we shared it with potential users, students, and faculty at Worcester Polytechnic Institute, as well as professional software engineers and mathematicians who specialize in bootstrap percolation. It was received extremely well; an overwhelming majority of respondents greatly liked the layout of the application and understood the functionality of the features. Booper can help scholars in the field research bootstrap percolation more efficiently, and it can be beneficial to the field in the long run.

In order to state our results, we first need some background on graph theory and the specific topics studied in this project.

Chapter 2

Background

This chapter provides the context behind graph theory, bootstrap percolation, independent sets, and related algorithms which are referenced and built upon in our project.

2.1 Context

This section introduces bootstrap percolation informally, in a way that can serve as a baseline for formally understanding the concepts later.

First, we introduce the concept of a **node**. A node can be any entity or thing, but for the purposes of the following examples, we can think about each node as a person.

Next, we introduce the concept of an **edge**. An edge connects two different nodes. We can think about this as a person-to-person relationship, like a friendship.

Figure 2.1: Two friends represented as two nodes and an edge.

The above figure is an example of a **graph**, a mathematical structure that can model communities and networks. One problem of interest to us is how something can spread through a graph. A relevant application of this would be modelling how a disease can spread through a group of people.

For example, in Figure 2.2, Ben is a close contact of both Avi and Caleb. On day 1, Avi, who is sick with a contagious disease, infects Ben. Ben becomes a carrier for the disease on day 2, and infects Caleb. By day 3, all of them have contracted the disease. This day-by-day contagious spread is a simple example of an **activation process**.

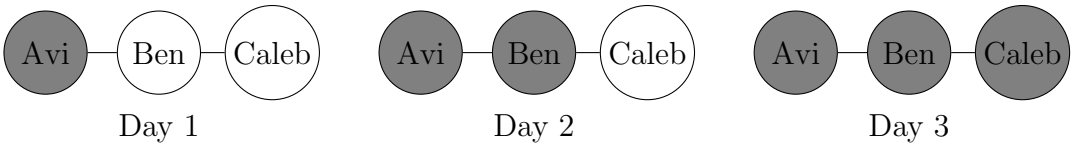

Figure 2.2: Day-by-day graphs modelling how a sickness can spread across three friends. The disease is represented as a grey coloring.

In the above model, we assumed that someone would be infected in the next day if they are a close contact with at least 1 person with the disease. However, consider the case where the disease was not so contagious, and we wanted to model a process whereby someone would be infected in the next day if they have at least 2 infected close contacts. To do this, we establish a **threshold**: The smallest number of infected close contacts a person would need to have to contract the disease in the next day. In the above model, we assumed a threshold of 1. *Unless otherwise stated, this report assumes a threshold of 2 for activation processes.*

Another assumption we made in the above model is that a person cannot re-contract a disease. This assumption is often accurate because a person can build immunity to a disease after contracting it. Visually, this means that a node that has been colored to indicate infection will remain colored for the duration of the process.

Representing infectious processes among communities as an iterative process (percolation) on a graph leads us to an interesting problem: what is the minimum number of initial disease carriers in a community needed to eventually infect the whole community? Formally defined, this is known as the **minimum contagious set** problem [5]. Studying the size of the minimum contagious set for a graph can inform us of how dangerous diseases could be, giving us a motivation to study it. This problem can also be extended to the realm of information sharing in social media, where we can find the smallest number of news sources required to fully disseminate fake news through a social network.

In Figure 2.3, we have a community of six people. Two people are initially sick. The assumption for this infectious spread is a threshold of 2. The two people that are initially sick form a minimum contagious set since they infect the whole community and there is no smaller configuration of initially sick people that would eventually infect the whole community.

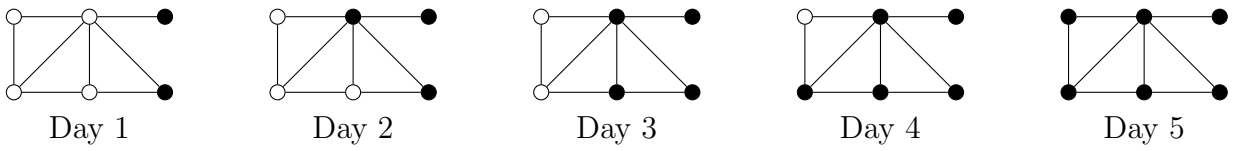

Figure 2.3: A graph of a community of people. Day 1 is starting day, with the initial disease carriers colored. For every day after, the person contracts the disease if they were linked to two disease carriers.

After this informal introduction, we provide the formal definitions that are needed later in this report.

2.2 Graph Theory

In this section, we define core concepts in graph theory, using [6]. First, a **graph** is a mathematical structure consisting of a set of elements named **vertices** (or **nodes**), and a set of **edges**, which are pairs of vertices. The set of vertices in a graph is denoted by V and the set of edges in a graph is denoted by E. Thus, the graph can be denoted as $G = (V, E)$. In this report, our discussion is limited to **finite graphs**, which are graphs with a finite number of vertices and edges.

When there is an edge between two vertices, we say that the vertices are **adjacent** to each other and **incident** to the edge. The **degree** of a vertex v, denoted $d(v)$, is the count of how many edges are incident to it. The **neighborhood** of a vertex v, denoted $N(v)$, is the set of vertices adjacent to v.

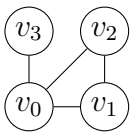

Figure 2.4: A graph with $V = \{v_0, v_1, v_2, v_3\}$ and $E = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_3\}, \{v_1, v_2\}\}$. The vertex v_1 has degree 2, and its neighborhood, $N(v_1)$, is $\{v_0$,$v_2\}$.

Graphs can be used to model community structures intuitively. Each vertex can be associated with a person, and each edge can represent a friendship/connection between two persons.

2.2.1 Graph Properties and Types of Graphs

There are different concepts and properties of graphs we can study to categorize them (see for example [6] for these concepts). One important concept is a **path**: a finite sequence of vertices, where each vertex is adjacent to its direct predecessor, and no vertex is repeated (except if the path starts and ends at the same vertex). In the case where the starting and terminal vertices of a path are the same, the path is called a **cycle**.

Figure 2.5: Left: A graph with a path from v_3 to v_6. Right: A graph with a cycle formed by the sequence of edges $\{(v_0, v_3), (v_3, v_9), (v_7, v_9), (v_1, v_7), (v_0, v_1)\}$.

Graphs are said to be **connected** if every pair of vertices has a path connecting them, and **disconnected** otherwise. Every disconnected graph can be expressed as a **union** of

disjoint connected graphs, which is the graph formed by the union of the vertex sets and the union of the edge sets from the individual graphs. A union of graphs does not add edges between the individual graphs, yielding a disconnected graph. In a union of connected graphs, each of these connected graphs is called a **component** of the resulting graph.

Alternatively, components can be defined as equivalence classes. Consider an equivalence relation on a vertex set. Two vertices are equivalent if and only if there is a path from one to the other. A component is an equivalence class of this relation. Furthermore, a graph is connected if and only if it has exactly one component.

Figure 2.6: Left: A connected graph. By definition, all connected graphs have 1 component. Right: A disconnected graph with 3 components. It is worth noting that the single-vertex graph is connected.

In general, a graph may have multiple edges between the same two vertices. These redundant edges are called **multiedges**. Additionally, an edge can join a vertex to itself; we call this edge a **loop**. However, these kinds of edges do not make sense in many graph theory problems. **Simple graphs** are graphs that do not have multiedges or loops. *Unless otherwise stated, all graphs in this report are assumed to be simple graphs.*

So far, we have discussed undirected graphs, in which each edge is an unordered pair of vertices and represents a mutual connection. **Directed graphs** are graphs whose edges are ordered pairs, and the connection only goes in one direction. The directed edge (v_0, v_1) means that v_1 can be reached from v_0, but not necessarily that v_0 can be reached from v_1. For directed graphs, instead of the generic degree parameter of a vertex, we consider the **in-degree** (number of incoming edges to that vertex) and the **out-degree** (number of outgoing edges to that vertex). In a similar vein, the vertices adjacent to a vertex v through its incoming edges are called its **incoming neighbors**, $N_{in}(v)$, and the vertices adjacent through outgoing edges are called its **outgoing neighbors**, $N_{out}(v)$. One application of directed graphs is to represent social media community data, where directed edges can be assigned from a user to all the people they follow. Since following is not necessarily mutual, directed graphs are appropriate here.

Figure 2.7: Left: A graph with an edge from v_0 to v_1. Right: A graph with edges from v_0 to v_1 and v_1 to v_0. Combining these edges is equivalent to a bidirectional edge.

Graphs with at least one cycle are called **cyclic**, while graphs with no cycles are called **acyclic**. Graphs that are both undirected and acyclic are called **forests** and graphs that are both directed and acyclic are simply called **directed acyclic graphs** (DAGs). DAGs are

particularly interesting and well studied because they can be **topologically ordered**; that is, the vertices can be arranged such that for every directed edge (v_i, v_j), v_i appears before v_j in the ordering [7].

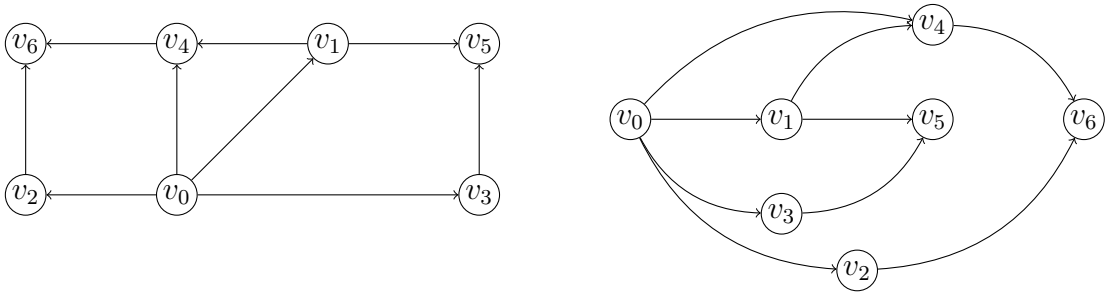

Figure 2.8: Left: A directed acyclic graph. Right: The left-to-right topological sort of the same directed acyclic graph.

An $M \times N$ **rectangular grid graph** is a graph representation of a grid with M vertices along the horizontal direction and N vertices along the vertical direction. The vertices can be arranged in a grid such that each vertex is adjacent to the vertices around it.

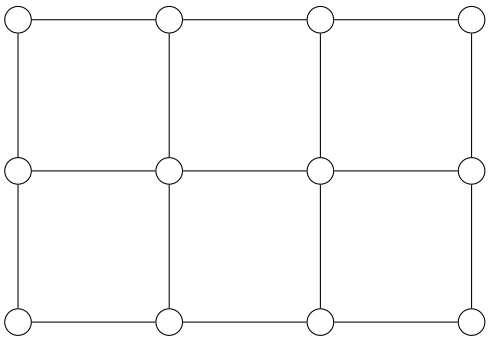

Figure 2.9: A 3×4 grid graph

A **triangular grid graph** T_n is defined as the graph interpretation of an order-$(n+1)$ triangular grid [8]. T_0 is the graph with a single vertex and no edges, and T_1 is the triangle graph.

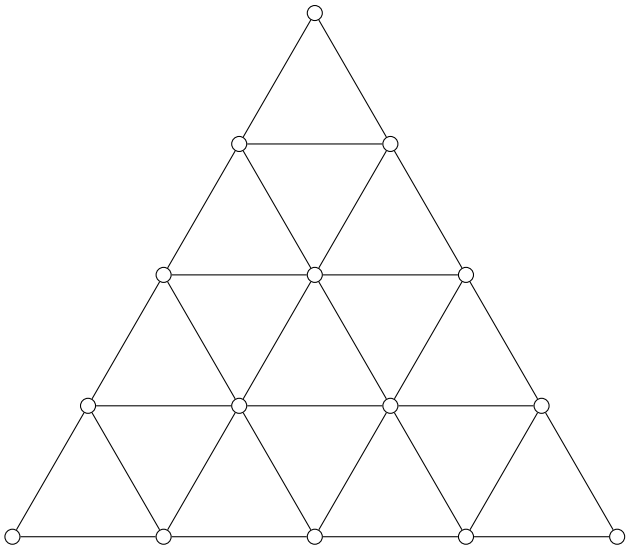

Figure 2.10: T_4

2.3 P vs. NP

A major problem in contemporary computer science is the **P** vs. **NP** problem, introduced by Cook in 1971 [9]. The problem is one of the seven Millennium Prize Problems posed by the Clay Mathematics Institute in 2000, paired with a writeup by Cook on its status [10]. There is a one million dollar prize offered to the first correct solution to the problem. Not only is the problem a major point of research, it is also so critical that its inclusion is typical in introductory texts in algorithms (for example, in Chapter 34 of [7]).

The problem involves two classes, P and NP, of **decision problems**, which are problems in which we receive an input and through some algorithm derive a yes or no answer. These problems are split into **complexity classes** depending on the amount of time it takes to solve and verify the solutions of the problems given the size of the input. The class P contains the problems which can be solved in polynomial time. These are the problems we consider efficiently solvable. NP contains the problems which can be verified in polynomial time. It is clear that $\mathrm{P} \subseteq \mathrm{NP}$, but it is not known whether or not $\mathrm{NP} \subseteq \mathrm{P}$. This is precisely the P vs. NP problem. Most computer scientists today believe $\mathrm{NP} \nsubseteq \mathrm{P}$ [11], and thus that there is no hope in finding efficient algorithms for some problems in NP.

NP-complete problems are the hardest problems in NP. That is, a problem is NP-complete if it is NP and all NP problems are **polynomial-time reducible** to it. To understand polynomial-time reduction, assume that we have two problems, *Problem A* and *Problem B*, and a solution for *Problem B* called *Solution B*. We say *Problem A* is polynomial-time reducible if there is a polynomial-time transformation of inputs of *Problem A* to inputs of *Problem B* and the number of times *Solution B* has to be called with these inputs is polynomial. Polynomial-time reductions are used to compare problems. If problem A is polynomial-time reducible to problem B, then problem B is at least as hard as problem A.

NP-hard problems are at least as hard as NP-complete problems. By definition, a problem is NP-hard if and only if all NP problems are polynomial-time reducible to it. Therefore, all NP-complete problems are NP-hard, but not all NP-hard problems are NP-complete.

As an example for a problem in P, we will consider a decision problem related to the activity selection problem. We are given a positive integer k and a set S of activities which each have a start time and an end time. We have to decide if there are k activities whose schedules do not overlap. This problem is in P. A polynomial-time algorithm is to repeatedly select the activity that ends the earliest, and remove it and all overlapping activities from S [7]. If we select at least k activities before S is empty, then the answer is yes, otherwise the answer is no.

In contrast, consider a decision problem related to the maximum clique problem. We are given a positive integer k and a simple undirected graph G. We have to determine if there is a set of k vertices in G such that each pair of vertices is adjacent. Such a set is called a clique. This decision problem is in NP since if we are given k, G, and a set $C \subseteq V(G)$, then we can decide in polynomial time whether or not C is a clique by checking each pair of vertices in C. This is an example of verifying a solution in polynomial time. Furthermore, this decision problem is NP-hard because another NP-hard problem, the 3-CNF satisfiability problem, has been polynomial-time reduced to the decision problem related to the maximum clique problem [7]. Since this decision problem is both NP and NP-hard, it is NP-complete. There is certainly an exponential-time algorithm, in which we consider every set of vertices and check if it is a clique, but it is not known if there is a polynomial-time algorithm for it.

Since the decision problem related to activity selection is in P, it is supposedly polynomial-time reducible to the NP-complete decision problem related to the maximum clique problem. Indeed, given an integer k and a set S of activities, we construct a graph G by adding one vertex for every activity and an edge between each pair vertices whose corresponding activities do not overlap. Then, we solve the decision problem related to the maximum clique problem with inputs k and G. G has a clique of size k if and only if there are k non-overlapping activities in S.

Notice that our discussion pertains mostly to decision problems related to some other problem. This is because these complexity classes are formally defined for decision problems, whereas the activity selection and maximum clique problems are both **optimization problems**. In an optimization problem, we have a set of solutions which each have a value, and we have to find one solution with either maximum or minimum value. In the maximum clique problem, the solutions are cliques, and the value of a solution is the number of vertices in it. In the activity selection problem, the solutions are sets of non-overlapping activities, and the value of a solution is the number of activities in it. Both of these optimization problems require a solution of maximum value. Every optimization problem has a related decision problem: given a value k and other inputs, the decision problem is to determine whether or not there is a solution with value k. In fact, the related decision problem is polynomial-time reducible to the original optimization problem; we could solve the optimization problem, compute the value of the optimal solution, and compare this value to k.

2.4 Bootstrap Percolation

k-neighbor bootstrap percolation is an activation process defined for an undirected graph $G = (V, E)$ given an integer $k \geq 1$ called the **threshold** [1]. Suppose we have an initial set of vertices $A_0 \subseteq V$. This set is called the **seed set**, and the vertices within them **seeds**. The recursive formula of bootstrap percolation is, for positive i,

$$A_i = A_{i-1} \cup \{v \in V : k \leq |N(v) \cap A_{i-1}|\}.$$

At step i, a vertex v is considered **active** if $v \in A_i$, and it is otherwise considered **inactive**. An inactive vertex becomes active if it has at least k active neighbors; when a vertex is activated this way, we say it is infected.

Given a seed set A_0, the set $\langle A_0 \rangle = \bigcup_i A_i$ is the set of vertices which the seed set A_0 eventually activates. A **contagious set** is a seed set for which $\langle A_0 \rangle = V$, meaning it eventually activates every vertex in the graph. Trivially, the entire vertex set is a contagious set, but a graph may have smaller contagious sets as well.

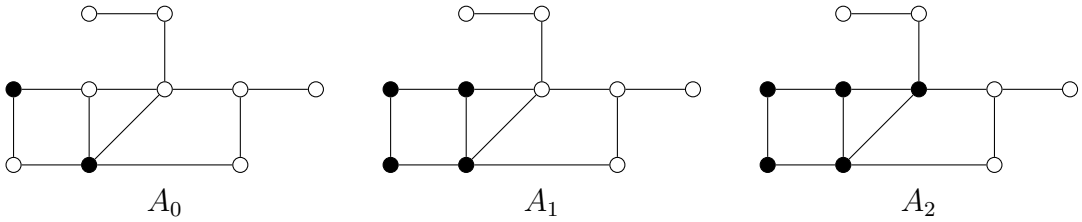

Figure 2.11: An example of 2-neighbor bootstrap percolation, with A_0, A_1, and $A_2 = \langle A_0 \rangle$ in black.

Bootstrap percolation has real-world applications in the modeling of spread. For example, it has been used to model the spread of behavior in social media in [2] as desired in viral marketing. In this application, social media users are represented as vertices, and their friendships, collaborations, etc. are represented as edges. Suppose a marketing team wants to spread an idea or opinion through this network. They could send advertisements to a contagious set, and allow the social media connections to spread the idea to the rest of the network. By choosing a minimum contagious set, the team can minimize their marketing costs.

2.4.1 Minimum Contagious Sets

Given a graph $G = (V, E)$ and a threshold k, let $m(G, k) = \min\{|A| \mid \langle A \rangle = V\}$ denote the minimum cardinality of a contagious set of vertices in G. We call a contagious set **minimum** if it has cardinality $m(G, k)$.

If we use a graph to model a social community, then the size of a minimum contagious set would indicate the potential danger of a contagious disease, such as COVID-19. In this application, a small minimum contagious set would indicate great danger because it would

only take a few disease carriers to infect the entire population; likewise, a large minimum contagious set would indicate low danger.

The minimum contagious set problem is NP-hard [4], thus it is believed that it is unlikely that there is an efficient, optimal algorithm for all inputs.

Since computing the exact value of $m(G, k)$ can be costly, we strive to find efficient algorithms for upper bounds; these can be thought of as approximations of $m(G, k)$. Several upper bounds have been found recently. Ackerman, Ben-Zwi, and Wolfovitz proved an upper bound of particular interest [12]. Later, Reichman improved this bound by providing a constructive proof in [13] through a polynomial-time algorithm for finding contagious sets.

Theorem 2.1 (Reichman, [13])**.** *For an undirected graph $G = (V, E)$ and a threshold k,*

$$m(G, k) \leq \sum_{v \in V} \min \left\{ 1, \frac{k}{d(v) + 1} \right\}.$$

This bound is derived through the following algorithm.

Algorithm 1: A greedy algorithm for finding contagious sets.

Input: a graph G and a threshold k
Output: a contagious set of G

while $\exists v \in V(G) : d(v) \geq k$ **do**
  Choose $v \in V(G) : d(v) = \min \{d(u) : u \in V(G), d(u) \geq k\}$;
  $G \leftarrow G - \{v\}$;
end
return $V(G)$

This algorithm iterates for as long as there is a vertex of degree at least our threshold k. Out of these vertices, we choose one with minimal degree, and remove it from our graph. The operation $G - \{v\}$ creates a new graph by removing vertex v and incident edges from G. Since this deleted vertex had at least k neighbors before its deletion, we know that it would be activated if the rest of the graph were active, so it is safe to remove. This holds for every iteration, so the vertex set of the resulting graph is a contagious set; the vertices which are not in this set will be activated in precisely the opposite order from which they were removed.

Note that this algorithm is **greedy**. During each iteration, it makes progress towards a solution by deleting a vertex. It uses a heuristic to decide which vertex to delete. Namely, it chooses the vertex with the least degree that can be removed. Removing a vertex with the smallest degree possible is typically a good choice because low-degree vertices can infect fewer neighbors than high-degree vertices. The use of heuristics is the defining characteristic of a greedy algorithm. Thus, greedy algorithms are generally fast, but they do not always produce optimal solutions.

Consider the example iteration for $k = 2$ in Figure 2.12.

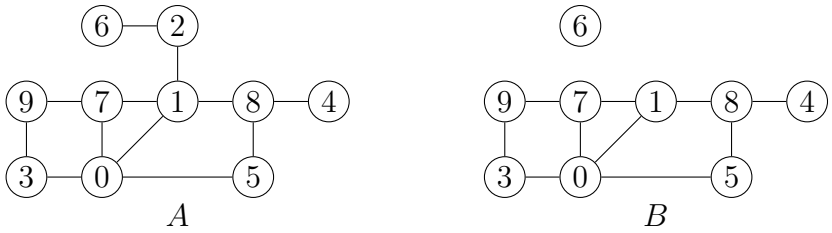

Figure 2.12: An example iteration of Algorithm 1. A is the graph before the iteration, and B is the graph after the iteration. Note that the vertex labelled 2 is removed.

The initial set of vertices of degree at least k is $\{0, 1, 2, 3, 5, 7, 8, 9\}$, and the vertices of minimal degree in this set are 2, 3, and 5. For consistency between iterations, and to ensure a deterministic result, we choose to remove the vertex whose label is the least. In 2.12, this vertex was 2. Continuing like so, the result of our final iteration is displayed in Figure 2.13.

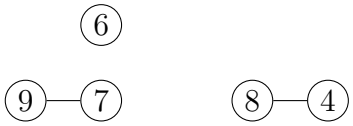

Figure 2.13: The output of the final iteration of our run-through of Algorithm 1.

Thus the set $\{4, 6, 7, 8, 9\}$ is a contagious set in our graph. Note, however, that this is not a minimum contagious set; Algorithm 1 does not guarantee its output is minimum, but can output minimum contagious sets. The set $\{1, 4, 6, 9\}$ is contagious and has cardinality one less, and is in fact minimum. By performing the same process described on our initial graph, but selecting the vertex whose label is the greatest, we can in fact arrive at this contagious set by Algorithm 1.

If the input graph has n vertices, then Algorithm 1 terminates in at most $n - k$ iterations. In the i^{th} iteration, it compares the degrees of $n - i + 1$ vertices when deciding which vertex to remove. So, the total running time of Algorithm 1 is indeed a polynomial.

$$\sum_{i=1}^{n-k}(n - i + 1) = \sum_{i=1}^{n-k}(k + i) = O(n^2)$$

2.4.2 Variations of Bootstrap Percolation

Bootstrap percolation need not be limited to undirected graphs. With small modifications, we can define it for a directed graph $G = (V, E)$ given a threshold k. Once more, pick an initial set of vertices $A_0 \subset V$. The activation process follows the equation

$$A_i = A_{i-1} \cup \{v \in V : k \leq |N_{in}(v) \cap A_{i-1}|\}$$

That is, a vertex is activated if it has at least k active incoming neighbors. This variant of bootstrap percolation allows us to study graphs where edges may be unidirectional, such as a

graph of a social media network like Twitter. A node in this graph (a Twitter user) would have outgoing neighbors representing their followers and incoming neighbors representing their followings. This successfully models the modern "social media influencer": a celebrity with a lot of followers, who can easily spread information (like fake news) to their large group of followers.

Furthermore, the threshold used need not be limited to a single integer. By defining a **threshold function** $k : V \to \mathbb{N}$, one can associate specific threshold values to each vertex in a graph. Our percolation process then becomes

$$A_i = A_{i-1} \cup \{v \in V : k(v) \leq |N(v) \cap A_{i-1}|\}$$

Now, a vertex v is activated if it has at least $k(v)$ active neighbors. A threshold function can add even more specificity to the bootstrap percolation process. Keeping with the same example from above, if we wanted to represent users who are more susceptible to fake news than others, we can assign these users a lower threshold value. This means that it would take fewer active neighbors to activate the node corresponding to that user. In other words, this user is more eager to believe the information shared by the sources they follow.

Both of these variations are generalizations of the model in which the graph is undirected, and the threshold is a constant. Consider a directed graph in which for every edge, there is another edge between the same two vertices in the opposite direction. This graph behaves like an undirected graph during bootstrap percolation. Additionally, a threshold function can behave like a constant threshold if it is a constant function. These variations can model more scenarios than the simple bootstrap percolation model.

2.5 Independent Sets

Consider a simple, undirected graph $G = (V, E)$. We say that $I \subseteq V$ is an **independent set** if and only if no two vertices in I are adjacent. The cardinality of the maximum independent set of G is known as the **independence number**, usually denoted as $\alpha(G)$. The independence number of a graph is closely related it's density/sparseness.

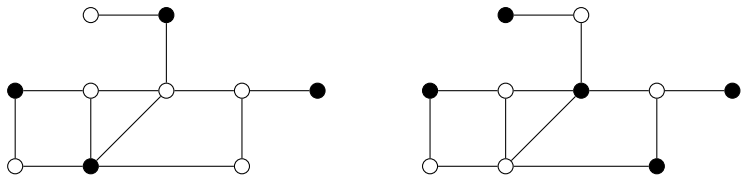

Figure 2.14: Two independent sets of the same graph, shown in black. The one on the right is a maximum independent set.

Finding an independent set is trivial. Choose an arbitrary vertex from each component, for example. Furthermore, the empty set is the minimum independent set for every graph. However, finding a maximum independent set is another NP-complete problem [14].

Maximum independent sets are useful for many applications, usually in choosing multiple options that cannot conflict with each other. For example, suppose that a company wants to open many retail stores in a region such that no two stores are close enough to compete for business. We could model this region as a graph, where each vertex represents a potential store location, and two vertices are adjacent if they are too close to each other. Then, a maximum independent set would dictate where to build retail stores.

Alternatively, suppose we have several tasks to complete. For efficiency, we want to parallelize these tasks as much as possible, but some tasks cannot be completed concurrently because they require a common resource. Maximum independent sets can aid in this situation as well. We can create a graph with one vertex for each necessary task. Then, we add an edge between any two vertices whose corresponding tasks cannot be completed in parallel. The independence number of this graph is the maximum number of tasks that can be done simultaneously.

Since the maximum independent set is an NP-complete problem, we do not have an efficient solution. The best we can get is an exponential-time algorithm. For example, a brute-force algorithm would have to check as many as 2^n vertex sets to find a maximum independent set. There are better algorithms available with a running time of $O(2^{\epsilon n})$ for some $0 < \epsilon < 1$ [15]. They cleverly avoid checking every possible vertex set and are much more efficient than their brute-force alternatives, but they do not run in polynomial time, as is desired.

In addition, there are some polynomial-time approximation algorithms that produce a (usually large, but often not maximum) independent set. We present two examples below. These algorithms are both greedy, and they are based on the idea that we can most likely maximize the independent set by including low-degree vertices.

Algorithm 2: A greedy algorithm for finding independent sets by picking vertices of minimum degree.

Input: a graph G
Output: an independent set of G

$I \leftarrow \emptyset$;
while $V(G) \neq \emptyset$ **do**
    Choose $v \in V(G) : d(v) = \delta(G)$;
    $I \leftarrow I \cup \{v\}$;
    $G \leftarrow G - \{v\} - N(v)$;
end
return I

Algorithm 3: A greedy algorithm for finding independent sets by picking a vertices of maximum degree.

Input: a graph G
Output: an independent set of G

while $E(G) \neq \emptyset$ **do**
  Choose $v \in V(G) : d(v) = \Delta(G)$;
  $G \leftarrow G - \{v\}$;
end
return $V(G)$

Many lower and upper bounds have been proven for the independence number. The Caro-Wei bound is particularly famous [16, 17]. Caro and Wei discovered this bound while working independently.

Theorem 2.2 (Caro-Wei, [16, 17])**.** *For a graph $G = (V, E)$,*

$$\alpha(G) \geq \sum_{v \in V} \frac{1}{1 + d(v)} \tag{2.1}$$

This bound has been improved many times, but it continues to receive a lot of attention because its formula and proof are both very simple. Additionally, Wei proved that Algorithm 2 achieves this bound [17], and Griggs proved that Algorithm 3 achieves it, too [18].

Algorithms 2 and 3 each terminate in at most n iterations. In the i^{th} iteration, they have to compare the degrees of at most $n - i + 1$ vertices to find a minimum/maximum degree vertex. So, the total running time of both algorithms is

$$\sum_{i=1}^{n}(n - i + 1) = \sum_{i=1}^{n} i = O(n^2).$$

2.6 k-Degenerate Graphs

A graph $G = (V, E)$ is said to be **k-degenerate** if and only if every subgraph of G has a vertex of degree at most k. Equivalently, k-degenerate graphs can be defined in terms of induced subgraphs. An **induced subgraph** is a subgraph formed from the original graph by keeping a subset of its vertices and all the edges spanned by these vertices. Any two vertices in an induced graph are adjacent if and only if they are adjacent in the original graph. Alternatively, we can create an induced subgraph by deleting a subset of the vertices and all incident edges, but keeping the edges that are not incident to deleted vertices. We can therefore say a graph is k-degenerate if and only if every induced subgraph of G has a vertex of degree at most k. An example of a 2-degenerate graph is shown in Figure 2.15. We also have the following result [13, 19].

Theorem 2.3 (Alon, Kahn, Seymour [19])**.** *Every graph $G = (V, E)$ has a k-degenerate induced subgraph with at least the following number of vertices:*

$$\sum_{v \in V} \min(1, \frac{k}{1 + d(v)}).$$

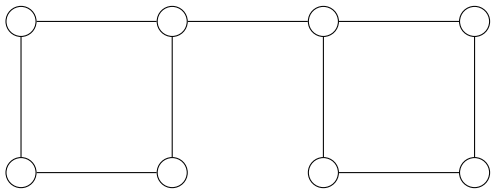

Figure 2.15: A 2-degenerate graph.

2.7 The Probabilistic Method

The **probabilistic method** is a proof technique often applied to proving bounds on graph parameters. Paul Erdős introduced it in the context of graph theory and combinatorics [20, 21]. This is a powerful proof technique since it is **non-constructive**, meaning that it can prove the existence of a mathematical object without having to provide an example or construction of said object. Furthermore, the conclusion derived from this method is certain, even though the method itself is probabilistic. The alternative to a non-constructive proof is a **constructive** proof, which explicitly shows how to construct the mathematical object.

The main idea for the probabilistic method is as follows: suppose we have a collection of objects which may or may not have a desired property, and we choose an object from this collection at random. If the probability that the selected object has the desired property is strictly positive, then there must exist a mathematical object with the desired property.

2.8 The Erdős-Rényi Random Graph Model

The **Erdős-Rényi model** is a model that generates graphs with a fixed number of nodes [22]. There are two variants of this model: $G(n, M)$ (fixed number of edges) and $G(n, p)$ (probabilistic number of edges).

The $G(n, M)$ model generates a graph with n labeled vertices and M edges. The graph generated by the model is picked uniformly at random out of all simple graphs of n labeled vertices and M edges. For example, the possible graphs from the Erdős-Rényi model $G(3, 2)$ can be seen in Figure 2.16.

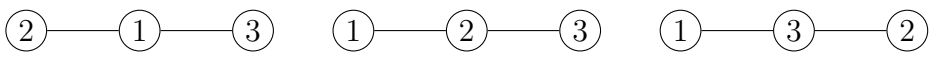

Figure 2.16: All possible $G(3, 2)$ graphs. This version of the model will yield one of these three graphs at random.

The $G(n, p)$ model generates a graph with n labelled vertices and a probabilistic number of edges. A graph on n vertices has $\binom{n}{2}$ possible edges. In this model, each edge is included in the graph with probability p ($0 \leq p \leq 1$). Therefore, the expected number of edges in a $G(n, p)$ graph is $p \cdot \binom{n}{2}$. Since this method is probabilistic, any graph on n vertices, ranging from the graph with no edges to the complete graph, can be generated by this variant of the model if $0 < p < 1$. Examples of possible $G(3, 0.5)$ can be seen in Figure 2.17.

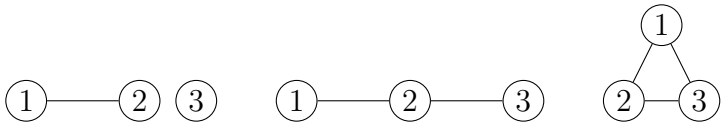

Figure 2.17: Possible $G(3, 0.5)$ graphs.

Both variants of this model can be used to generate large numbers of random graphs for testing and evaluation of results. They can also be used in the probabilistic method to derive expected values.

While these graphs are useful, they do have some limitations. One such limitation is that the Erdős-Rényi Model is unlikely to produce graphs with **community structure**. In many applications, graphs tend to be divided into communities which each have high internal connectivity, but low connectivity with other communities. Graphs with community structure, like those often produced by the **stochastic block model** [23], can replicate social network connections or real-life friendships. These models can be better for problems involving studying results on a real-life network.

2.9 Web Applications

Web applications have become an increasingly popular form of application software. JavaScript, the most popular programming language used to create web applications, has been the most popular language on GitHub (an Internet-hosted service for creating version-controlled repositories) since 2014 [24]. With rapid developments in technology rendering web applications faster and easier to use, it is no surprise that the use cases for these applications are quickly expanding. Since web applications run on a web server instead of having to be installed, they present an alternative to standard, desktop applications. Currently, they are being used as a tool for industries like education. Furthermore, web applications have been researched to be an effective supplement for education [25].

2.9.1 Popular Libraries

Multiple libraries exist to create responsive, interactive, and reactive applications. These types of applications can provide a satisfying user experience (**UX**) while maintaining good user interface (**UI**) designs. Developers often use multiple libraries to create a single product in web development. Here we will discuss a few select libraries.

React.js

React was created by Jordan Walke, a software engineer at Facebook. Initially released in 2013, it has since blossomed into one of the most used web frameworks worldwide [26]. It is currently maintained by Facebook, and has continued to be a prevalent framework, receiving frequent updates, patches, and developments.

React is noted for its use of a virtual **Document Object Model (DOM)**. A DOM is an interface that represents an HTML document as a tree, where each node in the tree corresponds to an object in the document. Since the data in an HTML document can quickly change based on user input, the application has to be able to re-render any changes as fast as possible. This is the goal of the virtual DOM.

At a high-level, React creates a cache containing the DOM. It calculates differences between the old and new DOM to quickly re-render the page. This saves time on re-rendering any unchanged styling on the page. In addition to its use of a virtual DOM, React also leverages components, a way of creating a reusable UI design for any element. This helps achieve regularity within the UI.

Material-UI

Material-UI is a JavaScript library that unifies React and its component-based structure with Google's well-researched Material Design system to create a visually appealing, structured application with desirable UI and UX features. It is used by a variety of software companies, including Spotify, Netflix, and Amazon [27]. The components available from Material-UI cover a wide range of possible use cases, with purposes including layout components, input components, and navigation components. It also includes layout structures and breakpoints to develop different layouts for different resolutions (e.g., one layout for a phone/tablet, one layout for a desktop). In addition, a *Theme* variable can be added and rendered on Material-UI components. This makes it possible to abstract the development of a good UI to a Theme file, which would include relative sizing, typography, and custom coloring. Themes help achieve consistency across the application by applying omnipresent style rules, and can greatly simplify the development process.

D3.js

D3.js is a powerful data visualization library that can create a multitude of charts and graphs. D3 manipulates the DOM to create stylistic solutions that combine HTML, CSS, JavaScript and Scalable Vector Graphic (SVGs) to create smooth, interactive data displays [28].

2.9.2 Graph Visualization Websites

The frameworks and libraries explained above are tools to create an interactive website for graph visualization and education. One such website is d3gt, an interactive graph theory tutorial website. The website, developed by Avinash Pandey, utilizes D3 for its graph visualization, in addition to JavaScript libraries Bootstrap and jQuery [29]. It covers 22

different units in graph theory, starting from the basics of vertices and edges and building up to more sophisticated concepts like spanning trees. While the graph certainly provides a helpful and intuitive visualization to aid learning, user engagement with the graph is limited to the context of the lesson it's aiding. It's not possible for users to upload and study their own graphs on this website.

Surprisingly, this was the only example we could find of an interactive web application for graph theory with significant use of the graph library provided through D3. Other websites required the user download something in order to use their graph visualization software.

Despite extensive searching, we could not find a graph visualization web application for bootstrap percolation. We filled this absence by creating a graph theory application that makes it easy for the user to choose what graphs they study, find various contagious sets in those graphs, and perform the percolation process.

Chapter 3

Theoretical Results

In this chapter, we present our new theoretical results. Since the minimum contagious set problem is NP-complete, there are three different ways to go about solving it. In this project, we used all three methods to find several different solutions. First, we created efficient, optimal algorithms for graphs with special structural properties. Second, we proved new upper bounds to approximate the size of the minimum contagious set of any graph. Third, we created an algorithm that finds the optimal solution, at the cost of efficiency, for the minimum contagious set of any graph. In addition, we reapplied our proof techniques to the maximum independent set problem and the problem of finding a maximum k-degenerate subgraph.

3.1 Optimal Solutions for Specific Graphs

In this section, we explore algorithms that would efficiently find a minimum contagious set in the following graphs: directed acyclic graphs, rectangular grid graphs, and triangular grid graphs.

3.1.1 Directed Acyclic Graphs

We discovered a new algorithm for the minimum contagious set of a directed acyclic graph.

Theorem 3.1. *For a directed acyclic graph $G = (V, E)$ and a threshold function $k : V \to \mathbb{N}$, the unique minimum contagious set is*

$$\{v \in V : d_{in}(v) < k(v)\}.$$

Proof. Let $A = \{v \in V : d_{in}(v) < k(v)\}$, and let $a \in A$. a cannot be infected by its in-neighbors, even if they were all active, because it has too few in-neighbors. Rather, the only way for a to be active is if it were a seed. Any set that does not contain a is not contagious because it does not activate a. Therefore, a is in every contagious set, and A is a subset of every contagious set.

Every directed acyclic graph has a topological order, as proved in [7]. We prove by induction on a particular topological order that A activates every vertex in G. Suppose that A activates the first r vertices of the particular topological order. In the base case, this is trivially true for $r = 0$. For the inductive step, let c be the $(r+1)^{\text{th}}$ vertex. There are two possible cases here.

Case 1. $c \in A$

A activates c simply because c is a seed.

Case 2. $c \notin A$

From the definition of A, we know that c has at least $k(c)$ in-neighbors. Since G is sorted topologically, these in-neighbors precede c. By the inductive hypothesis, the in-neighbors are activated by A. Since c has at least $k(c)$ active in-neighbors, c is infected.

In both cases, A activates c. By induction, A activates every vertex in G, and A is contagious.

Since A is contagious, and A is a subset of every contagious set, A is the unique minimum contagious set. □

This theorem gives us an optimal, polynomial-time algorithm, which is a major improvement over the exponential time required for the generic minimum contagious set problem. This algorithm is simple enough that we express it in a concise formula. In addition, the uniqueness of the minimum contagious set is shown. Many graphs have multiple minimum contagious sets, and determining how many could necessitate a lot of computation. Now, the number of minimum contagious sets of a directed acyclic graph requires no computation at all; there is always exactly one.

To apply this theorem, we would have to consider every vertex in a directed acyclic graph. Depending on the implementation of the graph and set data structures, we could compute the degree of a vertex and add it to the contagious set in constant time. Therefore, this algorithm for directed acyclic graphs can be executed in linear time.

We note that the existence of a topological order is important to this proof. However, when computing the minimum contagious set of a particular directed acyclic graph, the topological order is not necessary, or even useful. Rather, we can consider the vertices in any order we like because we can determine whether a certain vertex is in the minimum contagious set without knowing if its predecessors are included. All we need is the in-degree of each vertex.

3.1.2 Rectangular Grid Graphs

We proved another optimal, efficient algorithm for a minimum contagious set of a rectangular grid graph. It is a generalization of an algorithm for square grid graphs from [30]. We recognize that the following results have already been proved in [31], but we present a simpler proof.

We define an $m \times n$ grid graph as a grid graph with height m and width n, where $n \geq m$. For $1 \leq i \leq m$ and $1 \leq j \leq n$, let $v_{i,j}$ denote the vertex at the i^{th} row and j^{th} column.

Theorem 3.2. *For an $m \times n$ rectangular grid graph G and a threshold $k = 2$, the minimum contagious set has cardinality*

$$\left\lceil \frac{m+n}{2} \right\rceil.$$

Proof. We can construct a contagious set of G as follows. Let X denote the unique $m \times m$ square grid subgraph containing $v_{1,1}$, the top left vertex. Let the vertices on the diagonal of X be seeds. On the bottom row, moving right from $v_{m,m}$, let every other vertex be a seed. If $v_{m,n}$ is not chosen during this process, then let it be a seed, too.

For graphs where $n + m$ is even, $v_{m,n}$ is made a seed while choosing every other vertex in the bottom row. In this case, the resulting seed set is

$$S = \{v_{i,i} \mid 1 \leq i \leq m\} \cup \{v_{m,m+2i} \mid 1 \leq i \leq \frac{n-m}{2}\} \tag{3.1}$$

For graphs where $n + m$ is odd, $v_{m,n}$ is made a seed in addition to every other vertex in the bottom row. The resulting seed set is

$$S = \{v_{i,i} \mid 1 \leq i \leq m\} \cup \{v_{m,m+2i} \mid 1 \leq i \leq \left\lfloor \frac{n-m}{2} \right\rfloor\} \cup \{v_{m,n}\} \tag{3.2}$$

In both cases, S is contagious. From [30], we know that the diagonal of X activates all of X. As for the rest of the grid graph, the seeds on the bottom row activate the entire bottom row. Then, the bottom row together with the rightmost column of X activate the second-to-last row, then the third-to-last row, and so on. This process continues until the whole graph is active.

The cardinality of the set S is

$$\left\lceil \frac{m+n}{2} \right\rceil$$

This serves as an upper bound for the minimum contagious set.

To understand why $\left\lceil \frac{m+n}{2} \right\rceil$ is a lower bound, we consider a $m \times n$ chessboard. Each vertex in G corresponds to the square in its respective location on the chessboard. Each edge in G corresponds to the boundary between the two squares that represent the edge's endpoints. We apply an invariant method similar to that used in [30]. Consider the perimeter of the active squares. This value cannot increase during the percolation process. Suppose that each square has side length 1. Then, when the entire graph is active, the perimeter of the active region is $2m + 2n$. So, the perimeter of a contagious set is at least $2m + 2n$. Dividing this by 4, the perimeter of a single square, we see that we need at least $\frac{m+n}{2}$ seeds. Since the cardinality of a seed set must be an integer, we round this number up:

$$\left\lceil \frac{m+n}{2} \right\rceil$$

Note that the ceiling is only necessary when $m + n$ is odd.

The cardinality of a minimum contagious set is bounded above and below by $\lceil \frac{m+n}{2} \rceil$, so this is the cardinality of a minimum contagious set. □

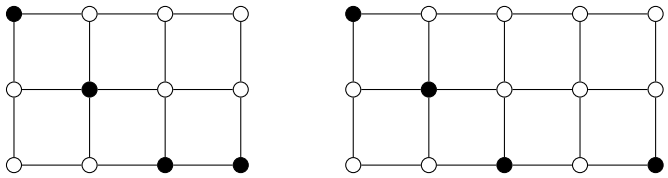

Figure 3.1: Left: a minimum contagious set of a 3×4 rectangular grid graph. Right: a minimum contagious set of a 3×5 rectangular grid graph.

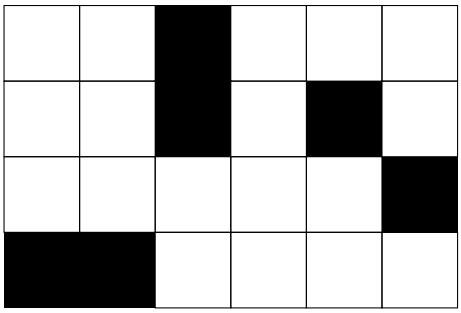

Figure 3.2: A chessboard corresponding to a 4×6 rectangular grid graph with 6 active vertices. The perimeter of the active region is 20.

While the proof for Theorem 3.2 is constructive, the minimum contagious set it presents is usually not unique. Except for some special cases when $m = 1$, every grid graph has multiple minimum contagious sets. For example, in the proof of Theorem 3.2, we could have let vertices on the counterdiagonal of X be seeds, instead of the diagonal.

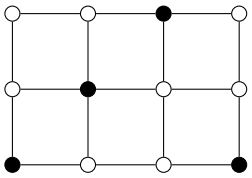

Figure 3.3: A minimum contagious set of a 3×4 rectangular grid graph.

Given an $m \times n$ grid graph, we could use Theorem 3.2 to find a minimum contagious set in $O(m + n)$ time. We simply apply (3.1) if $m + n$ is even or (3.2) if $m + n$ is odd, selecting the $\lceil \frac{m+n}{2} \rceil$ necessary vertices. With a data structure specifically for grid graphs, selecting a vertex could be done in constant time, so this algorithm is indeed $O(m + n)$.

3.1.3 Triangular Grid Graphs

In like manner, we proved a theorem that completely describes minimum contagious sets of triangular grid graphs. First, we present a necessary lemma.

Lemma 3.1. *For any triangular grid graph G with more than 1 vertex and a threshold $k = 2$, any two adjacent vertices form a minimum contagious set.*

Proof. Let any two adjacent vertices v_1 and v_2 be seeds, with the seed set denoted by S. For any vertex v other than v_1 and v_2, we can prove that v is activated by S. Let us induct on the distance l between v and S; that is, the length of the shortest path between v and the farthest vertex in S. If $l = 1$, then v forms a triangle with v_1 and v_2, so v is infected. Now suppose that all vertices with distance less than l from S are infected. Either v has two neighbors that are distance $l - 1$ from S, or v has one such neighbor and another neighbor which in turn has two such neighbors. In both cases, v is infected. By induction, every vertex in G is activated by S.

The cardinality of S is 2, which is also the minimum cardinality for a contagious set with $k = 2$ percolation. Therefore, S is a minimum contagious set for G. □

An example of the inductive step from the proof of Lemma 3.1 can be seen in Figure 3.4. A seed set is shown in black. Vertex 1 is distance 3 from the seed set, and it has two neighbors (2 and 3) which are distance 2 from the seed set. Vertex 4 is distance 3 from the seed set, and it has one neighbor (2) which is distance 2 from the seed set and one neighbor (1) which in turn has two such neighbors.

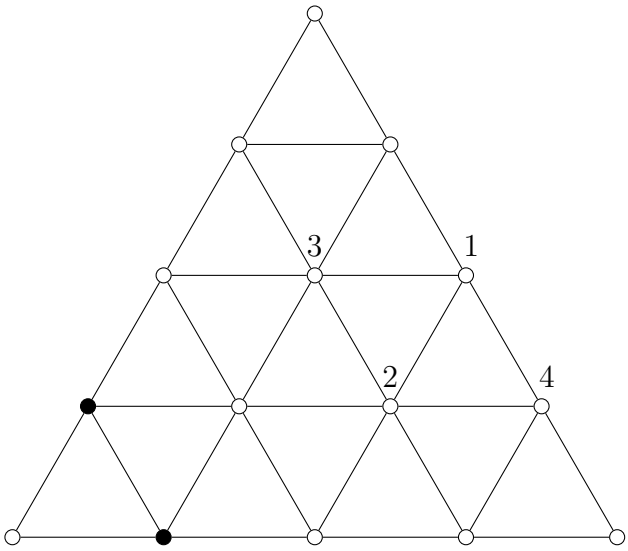

Figure 3.4: An illustration of the proof for Lemma 3.1.

With Lemma 3.1 in mind, here is our main result for triangular grid graphs.

Theorem 3.3. *For any triangular grid graph G with more than 1 vertex and a threshold $k = 2$, any two vertices that share a neighbor form a minimum contagious set.*

Proof. Pick any two vertices v_1 and v_2 that share a neighbor v_3, and let v_1 and v_2 be seeds, denoted by S. v_3 is infected in the first iteration. Then, v_1 and v_3 are adjacent active vertices. By Lemma 3.1, v_1 and v_3 infect the rest of the graph. So, S is contagious.

The cardinality of S is 2, which is also the minimum cardinality for a contagious set with $k = 2$ percolation. Therefore, S is a minimum contagious set for G. □

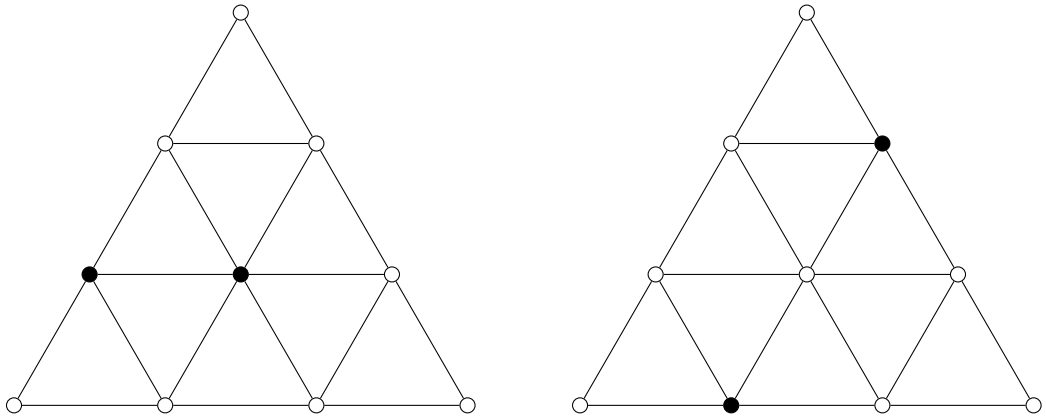

Figure 3.5: Two minimum contagious sets of a triangular grid graph.

In proving Theorem 3.3, we completely described minimum contagious sets for triangular grid graphs for $k = 2$. Every such set must be outlined by Theorem 3.3: two vertices that share a neighbor. If they don't share a neighbor, then they cannot percolate at all.

We can use Theorem 3.3 to find a minimum contagious set of a triangular grid graph in constant time. We could choose an arbitrary vertex, and then choose another arbitrary vertex from the first vertex's adjacency list. These operations could both be done in constant time, so the overall running time of this algorithm is also constant.

Note that Lemma 3.1 and Theorem 3.3 together are redundant because any two adjacent vertices in a triangular grid graph also share a neighbor. That said, both results are valuable. We used Lemma 3.1 to prove Theorem 3.3, and Theorem 3.3 is a generalization of 3.1.

3.2 Our New Bounds for All Graphs

In this section, we present bounds that approximate the cardinality of a minimum contagious set, the cardinality of maximum k-degenerate subgraphs, and the cardinality of maximum independent sets for all graphs.

3.2.1 Contagious Sets

Reichman presented Algorithm 1 in [13]. Given a graph $G = (V, E)$ and a threshold $k \geq 1$, it finds a contagious set of G with cardinality at most

$$\sum_{v \in V} \min\{1, \frac{k}{d(v)+1}\} \tag{3.3}$$

Here we prove that there is actually a stronger upper bound that applies to the output of Algorithm 1 (see the discussion below, after the proof about why it is stronger in certain cases).

Theorem 3.4. *The output of Algorithm 1 has cardinality at most*

$$\sum_{v \in V} \begin{cases} 1 & d(v) < k \\ \frac{k}{d(v)+1} & d(v) \geq k \wedge \exists u \in N(v) : d(u) \geq k \\ 0 & d(v) \geq k \wedge \forall u \in N(v),\ d(u) < k \end{cases} . \tag{3.4}$$

Proof. For a graph $G = (V, E)$ and a threshold $k \in \mathbb{N}$, let

$$w(G) = \sum_{v \in V} \begin{cases} 1 & d(v) < k \\ \frac{k}{d(v)+1} & d(v) \geq k \wedge \exists u \in N(v) : d(u) \geq k \\ 0 & d(v) \geq k \wedge \forall u \in N(v),\ d(u) < k \end{cases}$$

We will show that $w(G)$ does not increase during the iterations of the algorithm. At any step of the algorithm, let v be the vertex that is removed. There are two possible cases here.

Case 1. $\exists u \in N(v) : d(u) \geq k$

The summation term corresponding to v is $\frac{k}{d(v)+1}$. When we remove v, we both remove this term from the summation and change the terms corresponding to v's neighbors. Let $u_1, u_2, ...u_l$ be the neighbors of v that have degree at least k. Each u_i has a neighbor (v) that has degree at least k, so its summation term changes from $\frac{k}{d(u_i)+1}$ to at most $\frac{k}{d(u_i)}$ (this includes the case $d(u_i) = k$). The summation terms corresponding to other neighbors of v are 1 before and after v is removed. When v is removed, the summation becomes

$$\begin{aligned} w(G - \{v\}) &\leq w(G) - \frac{k}{d(v)+1} + \sum_{i=1}^{l} (\frac{k}{d(u_i)} - \frac{k}{d(u_i)+1}) \\ &= w(G) - \frac{k}{d(v)+1} + \sum_{i=1}^{l} (\frac{k}{d(u_i)(d(u_i)+1)}) \end{aligned}$$

By the definition of the algorithm, $d(v) \leq d(u_i)$. Therefore, $w(G - \{v\})$ can be bounded above by replacing $d(u_i)$ with $d(v)$.

$$\begin{aligned} w(G - \{v\}) &\leq w(G) - \frac{k}{d(v)+1} + \sum_{i=1}^{l} \left(\frac{k}{d(v)(d(v)+1)}\right) \\ &= w(G) - \frac{k}{d(v)+1} + \frac{lk}{d(v)(d(v)+1)} \end{aligned}$$

Since $l \leq d(v)$, this expression can be bounded above by replacing l with $d(v)$.

$$\begin{aligned} w(G - \{v\}) &\leq w(G(A)) - \frac{k}{d(v)+1} + \frac{d(v)k}{d(v)(d(v)+1)} \\ &= w(G) - \frac{k}{d(v)+1} + \frac{k}{d(v)+1} \\ &= w(G) \end{aligned}$$

Case 2. $\forall u \in N(v),\ d(u) < k,$

The summation term corresponding to v is 0. Removing this term does not affect the value of w(G). The summation terms corresponding to v's neighbors are all 1 both before and after v is removed. Therefore, $w(G - \{v\}) = w(G)$.

In both cases, $w(G)$ does not increase after an iteration of the algorithm. Let I be the output of the algorithm for a graph G. By induction, $w(G|_I) \leq w(G)$, where $G|_I$ is the subgraph of G induced by I. By the definition of the algorithm, each vertex in $G|_I$ has degree less than k. Therefore, the summation terms in $w(G|_I)$ are all 1, and $w(G|_I) = |I|$. By transitivity, $|I| \leq w(G)$. □

For all graphs and thresholds, the value of our new upper bound is at most the value of the bound in (3.3). Most terms are equal between the two summations, but for vertices with degree at least k and with neighbors all having degree less than k, the term in the new bound is strictly less than the term in the old bound. This reduction can lead to an improvement in the upper bound for certain graphs. For example, consider a union of m disjoint stars with k leaves each. The value of the new upper bound is mk, which is tight. The value of the old bound is $m(k + \frac{k}{k+1})$.

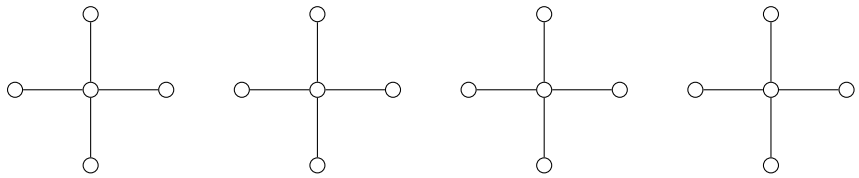

Figure 3.6: A union of 4 disjoint stars with 4 leaves each. With a threshold of 4, the bound from (3.3) is $\frac{96}{5}$, while the bound from (3.4) is only 16.

We can prove another upper bound with a slight modification to the probabilistic proof that appears in [12, 13]. This proof is non-constructive as well, but the resulting bound is much stronger than even (3.4) for certain graphs.

Theorem 3.5. *Let $G = (V, E)$ be an undirected graph. Let $k \geq 1$ be a threshold. Let us define functions*

$$A, B : V \to \mathcal{P}(V)$$
$$A(v) = \{u \in N(v) : d(u) < d(v)\}$$
$$B(v) = \{u \in N(v) : d(u) \leq d(v)\}$$

Then, there is a contagious set of G with cardinality at most

$$\sum_{v \in V} \begin{cases} 1 & |B(v)| < k \\ \frac{k - |A(v)|}{1 + |B(v) \setminus A(v)|} & |A(v)| < k \leq |B(v)| \\ 0 & k \leq |A(v)| \end{cases}. \tag{3.5}$$

Proof. For a graph $G = (V, E)$ and a threshold $k \geq 1$, choose a permutation of the vertices uniformly at random from the permutations in which the degrees of the vertices are non-decreasing. That is, $\forall u, v \in V$, if u precedes v in the permutation, then $d(u) \leq d(v)$. Let

$$L = \{v \in V : v \text{ has fewer than } k \text{ neighbors that precede } v \text{ in the permutation}\}$$

We argue by contradiction that L is contagious.

Suppose that L is not contagious. Then there must be at least one vertex in the graph that is not activated by L. Let v be the first such vertex in the permutation. We know that $v \notin L$, otherwise L would trivially activate v. Therefore, v has at least k neighbors that precede v in the permutation. L activates all of these neighbors because v is the first vertex that L does not activate. Since v has at least k active neighbors, v is infected. This is a contradiction, so L must be contagious.

Now we will compute the expectation $\mathbb{E}[|L|]$. The probability that a particular vertex is in L depends on its degree and the degrees of its neighbors. We will discuss each possible case in detail. Formally, let $v \in V$. If $|B(v)| < k$, then v has fewer than k neighbors that precede v in every permutation for which the degrees are non-decreasing.

$$P(v \in L \mid |B(v)| < k) = 1$$

If $|A(v)| < k \leq |B(v)|$, then it is possible but not certain that $v \in L$. Every vertex in $A(v)$ precedes v, and every vertex that precedes v is in $B(v)$. Therefore, $v \in L$ if and only if at least $k - |A(v)|$ vertices in $B(v) \setminus A(v)$ precede v. v can appear in $1 + |B(v) \setminus A(v)|$ locations in the permutation relative to its neighbors. In $k - |A(v)|$ of these locations, v has fewer than $k - |A(v)|$ neighbors in $B(v) \setminus A(v)$ that precede v. Therefore,

$$P(v \in L \mid |A(v)| < k \leq |B(v)|) = \frac{k - |A(v)|}{1 + |B(v) \setminus A(v)|}$$

If $k \leq A(v)$, then v must have at least k neighbors that precede v in the permutation. Therefore, $v \notin L$.

$$P(v \in L \mid k \leq |A(v)|) = 0$$

By the linearity of expectation, $\mathbb{E}[|L|]$ is the sum of these probabilities for each vertex.

$$\mathbb{E}[|L|] = \sum_{v \in V} P(v \in L) = \sum_{v \in V} \begin{cases} 1 & |B(v)| < k \\ \frac{k - |A(v)|}{1 + |B(v) \setminus A(v)|} & |A(v)| < k \leq |B(v)| \\ 0 & k \leq |A(v)| \end{cases}$$

By expectation properties, there is a permutation of the vertices for which $|L| \leq \mathbb{E}[|L|]$. Hence, there is a contagious set that obeys this upper bound. □

While performing empirical evaluations of the bound from (3.5), we found that it was usually much weaker than our bounds from (3.3) and (3.4). We reasoned that if we reversed the order of the permutation in the proof of Theorem 3.5, then we would probably prove a much stronger bound. Accordingly, we present a third new upper bound on contagious sets. The formula and proof for this bound follow the same structure as (3.5), but we will see in a discussion afterwards that either bound can be stronger than the other for various graphs. Later, in Chapter 4, we will see that this new bound is indeed much stronger than other previously known bounds for both random graphs and graphs representing real-world scenarios.

Theorem 3.6. *Let $G = (V, E)$ be an undirected graph. Let $k \geq 1$ be a threshold. let us define functions*

$$C, D : V \to \mathcal{P}(V)$$
$$C(v) = \{u \in N(v) : d(u) > d(v)\}$$
$$D(v) = \{u \in N(v) : d(u) \geq d(v)\}$$

Then, there is a contagious set of G with cardinality at most

$$\sum_{v \in V} \begin{cases} 1 & |D(v)| < k \\ \frac{k - |C(v)|}{1 + |D(v) \setminus C(v)|} & |C(v)| < k \leq |D(v)| \\ 0 & k \leq |C(v)| \end{cases} \tag{3.6}$$

Proof. For a graph $G = (V, E)$ and a threshold $k \geq 1$, choose a permutation of the vertices uniformly at random from the permutations in which the degrees of the vertices are non-increasing. That is, $\forall u, v \in V$, if u precedes v in the permutation, then $d(u) \geq d(v)$. Let

$$L = \{v \in V : v \text{ has fewer than } k \text{ neighbors that precede } v \text{ in the permutation}\}$$

By the same argument that we presented in Theorem 3.5, L is contagious in G.

Now we will compute the expectation $\mathbb{E}[|L|]$. The probability that a particular vertex is in L depends on its degree and the degrees of its neighbors. We will discuss each possible case in detail. Formally, let $v \in V$. If $|D(v)| < k$, then v has fewer than k neighbors that precede v in every permutation for which the degrees are non-increasing.

$$P(v \in L \mid |D(v)| < k) = 1$$

If $|C(v)| < k \leq |D(v)|$, then it is possible but not certain that $v \in L$. Every vertex in $C(v)$ precedes v, and every vertex that precedes v is in $D(v)$. Therefore, $v \in L$ if and only if at least $k - |C(v)|$ vertices in $D(v) \setminus C(v)$ precede v. v can appear in $1 + |D(v) \setminus C(v)|$ locations in the permutation relative to its neighbors. In $k - |C(v)|$ of these locations, v has fewer than $k - |C(v)|$ neighbors in $D(v) \setminus C(v)$ that precede v. Therefore,

$$P(v \in L \mid |C(v)| < k \leq |D(v)|) = \frac{k - |C(v)|}{1 + |D(v) \setminus C(v)|}$$

If $k \leq C(v)$, then v must have at least k neighbors that precede v in the permutation. Therefore, $v \notin L$.

$$P(v \in L \mid k \leq |C(v)|) = 0$$

By the linearity of expectation, the expectation $\mathbb{E}[|L|]$ is the sum of these probabilities for each vertex.

$$\mathbb{E}[|L|] = \sum_{v \in V} P(v \in L) = \sum_{v \in V} \begin{cases} 1 & |D(v)| < k \\ \frac{k - |C(v)|}{1 + |D(v) \setminus C(v)|} & |C(v)| < k \leq |D(v)| \\ 0 & k \leq |C(v)| \end{cases}$$

By expectation properties, there is a permutation of the vertices for which $|L| \leq \mathbb{E}[|L|]$. Hence, there is a contagious set that obeys this upper bound. □

This new upper bound can indeed be less than our previously known bounds. For example, consider the complete bipartite graph $K_{k,r}$ for some $r > k$. The value of the upper bound from (3.6) is k, which is tight. However, the value of the upper bound is from (3.5) is r. The value of the upper bound from (3.4) is $\frac{k^2}{r+1} + \frac{rk}{k+1} \geq \frac{k^2}{r+1} + \frac{k(k+1)}{k+1} > k$. Theorem 3.6 provides the strongest bound for contagious sets of $K_{k,r}$.

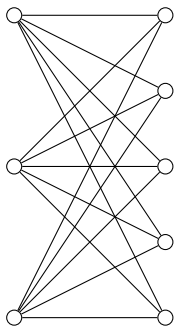

Figure 3.7: The complete bipartite graph $K_{3,5}$. With a threshold of 3, the bound from (3.4) is $\frac{21}{4}$, the bound from (3.5) is 5, and the bound from (3.6) is 3.

On the other hand, consider a clique of r vertices which each have k additional neighbors of degree 1. If $1 \leq r < k$, then the value of the bound from (3.5) is rk, which is tight. The value of the bound from (3.6) is $rk + r$. The value of the bound from (3.4) is $rk + \frac{rk}{r+k}$. The bound from (3.5) is the strongest for this graph.

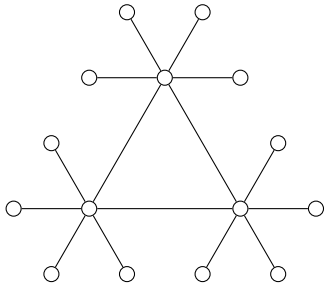

Figure 3.8: A clique of 3 vertices, which are each adjacent to 4 other vertices of degree 1. With a threshold of 4, the bound from (3.4) is $\frac{96}{7}$, while the bound from (3.5) is only 12, and the bound from (3.6) is 15.

However, the bound from (3.4) is often more valuable than our other two bounds because it can be achieved by a deterministic efficient algorithm. These three bounds all have different strengths and weaknesses, making them useful in different situations.

3.2.2 k-Degenerate Graphs

The technique and result of Theorem 3.5 is also applicable to approximating the cardinality of k-degenerate graphs.

Theorem 3.7. *Let $G = (V, E)$ be and undirected graph, and let $k \in \mathbb{N}$. Let A and B be the functions defined in Theorem 3.5. G has a k-degenerate subgraph with cardinality at least*

$$\sum_{v \in V} \begin{cases} 1 & |B(v)| < k \\ \frac{k - |A(v)|}{1 + |B(v) \setminus A(v)|} & |A(v)| < k \leq |B(v)| \\ 0 & k \leq |A(v)| \end{cases}.$$

Proof. Choose a permutation of V, and let $L \subseteq V$ as defined in the proof for Theorem 3.5. Let $U \subseteq L$, and let $u \in U$ such that u succeeds all other vertices of U in the permutation. u has fewer than k neighbors in U, otherwise $u \notin L$. Therefore, the subgraph induced by L is k-degenerate. We know that

$$\mathbb{E}[|L|] = \sum_{v \in V} \begin{cases} 1 & |B(v)| < k \\ \frac{k - |A(v)|}{1 + |B(v) \setminus A(v)|} & |A(v)| < k \leq |B(v)| \\ 0 & k \leq |A(v)| \end{cases}$$

By expectation properties, there is a permutation for which $|L| \geq \mathbb{E}[|L|]$. Therefore, there is induced subgraph of G that is k-degenerate and has a cardinality at least $\mathbb{E}[|L|]$. □

3.2.3 Independent Sets

While our new bounds for contagious sets are valuable in their own right, the proof technique we used for Theorem 3.5 can be reapplied to improve upon the Caro-Wei bound (Theorem 2.2) for independent sets.

Theorem 3.8. *For a graph $G = (V, E)$, let us define a function*

$$A : V \to \mathcal{P}(V)$$

$$A(v) = \{u \in N(v) : d(u) = d(v)\}.$$

G has an independent set of cardinality at least

$$\sum_{v \in V} \begin{cases} 0 & \exists u \in N(v) : d(u) < d(v) \\ \frac{1}{1+|A(v)|} & \forall u \in N(v),\ d(v) \leq d(u) \end{cases}. \tag{3.7}$$

Proof. Choose a permutation of V uniformly at random for which the degrees of the vertices are increasing. Let

$$I = \{v \in V : \forall u \in N(v), v \text{ precedes } u \text{ in the permutation } \}$$

Let $v \in I$ and $u \in N(v)$. We know that $u \notin I$ because u has a neighbor (v) that precedes u in the permutation. Therefore, I is an independent set.

Let us compute the expectation $\mathbb{E}[|I|]$. Let $v \in V$. If v has a neighbor whose degree is less than that of v, then this neighbor precedes v in every possible permutation. In this case, it is certain that $v \notin I$.

$$P(v \in I \mid \exists u \in N(v) : d(u) < d(v)) = 0$$

If v does not have such a neighbor, then it is possible that $v \in I$. Consider the set $\{v\} \cup A(v)$. $A(v)$ might be empty, but this possibility does not affect our argument. $v \in I$ if and only if v is the first vertex from $\{v\} \cup A(v)$ to appear in the permutation. v can appear in $1 + |A(v)|$ locations relative to its neighbors, and there is one such location for which $v \in I$.

$$P(v \in I \mid \forall u \in N(v),\ d(v) \leq d(u)) = \frac{1}{1 + |A(v)|}$$

Notice that every vertex must satisfy exactly one of these two cases. We can use the probabilities to compute the expectation $\mathbb{E}[|I|]$.

$$\mathbb{E}[|I|] = \sum_{v \in V} P(v \in I) = \sum_{v \in V} \begin{cases} 0 & \exists u \in N(v) : d(u) < d(v) \\ \frac{1}{1+|A(v)|} & \forall u \in N(v),\ d(v) \leq d(u) \end{cases}$$

By expectation properties, there is a permutation of V for which $|I| \geq \mathbb{E}[|I|]$. This particular I is an independent set that satisfies the lower bound. □

A classic proof for the Caro-Wei bound involves choosing a permutation of the vertices uniformly at random. We reasoned that it would be helpful to sort the permutation by degree, so that low-degree vertices are more likely to be included in I than high-degree vertices. As we can see from Algorithms 2 and 3, including low-degree vertices instead of high-degree vertices usually helps us maximize independent sets.

Indeed, our new bound can be larger than the Caro-Wei bound. Consider the following graph consisting of an independent set I of size r, a clique C of size r, and all vertices in I are adjacent to all vertices in C. Then for every $v \in I$, all of v's neighbors have degree greater than that of v. Hence our bound for this graph is r, which is tight. On the hand, the Caro-Wei bound is $\frac{r}{r+1} + \frac{1}{2} < 2$.

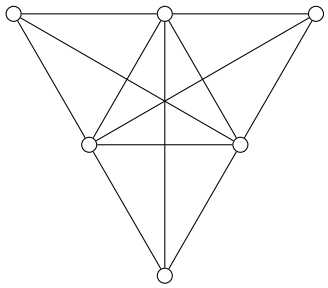

Figure 3.9: A clique of 3 vertices, and an independent set of 3 vertices which are each adjacent to every vertex in the clique. The bound from (3.7) is 3. The Caro-Wei bound is $\frac{5}{4}$.

3.3 An Exact Algorithm for All Graphs

We developed an algorithm for the minimum contagious set problem. It is optimal, and it can be applied to any undirected graph. Even though this problem is NP-complete, making it improbable for anybody to find a polynomial-time algorithm, we strived to make our algorithm as efficient as we could. We drew inspiration from [15], which used recursion to write maximum independent set algorithms that run in $O(2^{\epsilon n})$ time for some $0 < \epsilon < 1$. We applied various methods to reduce the height of the recursion tree and avoid entire branches.

Given an input graph $G = (V, E)$, consider a specific vertex v. Every Contagious set falls into exactly one of two categories: sets that contain v, and sets that do not contain v. We recursively find a minimum among the contagious sets that contain v and a minimum among the contagious sets that do not contain v. Then, we compare the results of the two recursive calls and return the smaller set.

Now we consider the subproblem of finding a minimum among the contagious sets that contain v. This subproblem is easiest to solve when the threshold is a function, rather than a constant, as described in subsection 2.4.2. However, a threshold function is capable of representing a constant threshold, so we do not lose any generality here. To solve this subproblem, we remove v from the graph, and we decrement the thresholds of v's neighbors, saturating to zero if necessary. We reasoned that if v must be a seed, then its presence would have the same effect as decrementing the thresholds of its neighbors. While decrementing

thresholds, it is possible that some thresholds become zero. When this phenomenon occurs, we interpret that these vertices are activated by the seeds that we have already chosen. We can remove these vertices in the same way that we remove v. This additional processing might seem costly, but it ultimately makes the algorithm much more efficient. Every time we remove an additional vertex, we reduce the height of the recursion tree, which has a much stronger impact on efficiency than processing the vertices that are infected by our chosen seeds. When all of this processing is finished, we recursively find a minimum contagious set of the graph and add v to this result.

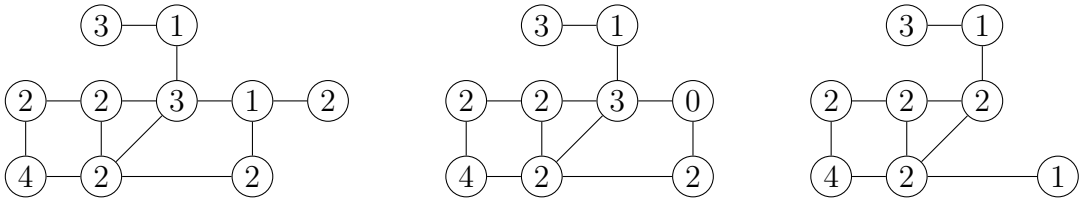

Figure 3.10: An example of Algorithm 6. We delete the far right vertex because it is the input v, and we decrement the threshold of its neighbor. Since the neighbor now has a non-positive threshold, we delete it, too, and we decrement the thresholds of its neighbors. We stop here because all vertices have positive thresholds.

Next, let us focus on the subproblem of finding a minimum among the contagious sets that do not contain v. Within each recursive call, we maintain a set of vertices $A \subseteq V$ and require that the output be a subset of A. Initially, $A = V$. When we add the restriction that v is not in the output set, we simply remove v from A and recurse. Note that if A is not contagious in G, then there is no subset of A that is contagious in G. In this case, our algorithm returns a special symbol, *null*.

Since *null* is a possible return value from recursive calls, we have to adapt our other solutions within this algorithm. While solving the first subproblem, if its recursive call returns *null*, then this subproblem call returns *null* as well, passing it up the recursion tree. While solving the overall problem, if both recursive calls return contagious sets, then we return the minimum of the two. If one recursive call returns a contagious set, and the other returns *null*, then we return the contagious set. If both recursive calls return *null*, then we return *null*.

Notice that if there is a contagious set that does not contain v, then there is also a contagious set that contains v. This property of contagious sets provides an opportunity for an efficiency increase. The subproblem of finding the minimum among the contagious sets that contain v must be solved first. If the result is *null*, then we can skip solving the other subproblem because its solution would certainly be *null*. In this way, we eliminate large portions of the recursion tree, resulting in a major improvement in efficiency.

Since this algorithm is recursive, we need to consider the base case. Before we solve the two subproblems, we have to choose a vertex $v \in A$. So, we decided that the base case is when there are no vertices to choose. That is, $A = \emptyset$. In this case, if G has no vertices then $\emptyset$ is its minimum contagious set, so we return $\emptyset$. However, if G has vertices, then there is no subset of A that is contagious in G, so we return *null*.

We included one more improvement to the efficiency. If there are vertices whose degrees are less than their respective thresholds, then these vertices must be in every contagious set. It is better to process them up front, rather than recurse on them. Before we begin recursive calls, we delete these vertices in the same way that we delete v in the subproblem of searching for a minimum among the contagious sets that contain v. Then, we find a minimum contagious set of the graph and add the deleted vertices. Every vertex that we remove in this processing reduces the maximum height of the recursion tree by 1, resulting in a major improvement if several vertices are removed up front. We do not have to repeat this processing during the recursive calls. If we already removed every vertex whose degree is less than its threshold, then no more vertices attain this property. Every time we remove a vertex, decrementing the degrees of its neighbors, we also decrement the thresholds of its neighbors.

Now, we present pseudocode, summarizing the algorithm described in this section. Due to the complexity of the algorithm, we find it easiest to present it as three mutually recursive subroutines.

Algorithm 4: minimumContagiousSet

Input: An undirected graph $G = (V, E)$ and a threshold function $k : V \to \mathbb{N}$
Output: A minimum contagious set of G

$B \leftarrow \emptyset$;
for $v \in V$ **do**
  if $d(v) < k(v)$ **then**
    $k(v) \leftarrow 0$;
    $B \leftarrow B \cup \{v\}$;
  end
end
while $\exists v \in V : k(v) = 0$ **do**
  for $u \in N(v)$ **do**
    $k(u) \leftarrow \max(0, k(u) - 1)$;
  end
  $G \leftarrow G - \{v\}$;
  $k \leftarrow k - \{(v, k(v))\}$;
end
return $B \cup$ restrictedMinimumContagiousSet(G, k, V)

Algorithm 5: restrictedMinimumContagiousSet

Input: An undirected graph $G = (V, E)$, a threshold function $k : V \to \mathbb{N}$, and a set of vertices $A \subseteq V$

Output: A minimum subset of A that is contagious in G, or *null* if there is no such set

```
if A = ∅ then
    if V = ∅ then
        return ∅
    else
        return null
else
    Choose v ∈ A;
    B ← doublyRestrictedMinimumContagiousSet(G, k, A, v);
    if B = null then
        return null
    end
    else
        C ← restrictedMinimumContagiousSet(G, k, A − {v});
        if C = null or |B| < |C| then
            return B
        else
            return C
```

Algorithm 6: doublyRestrictedMinimumContagiousSet

Input: An undirected graph $G = (V, E)$, a threshold function $k : V \to \mathbb{N}$, a set of vertices $A \subseteq V$, and a vertex $v \in A$

Output: A minimum subset of A that contains v and is contagious in G, or *null* if there is no such set

```
k(v) ← 0;
while ∃u ∈ V : k(u) = 0 do
    for w ∈ N(u) do
        k(w) ← max(0, k(w) − 1);
    end
    G ← G − {u};
    k ← k − {(u, k(u))};
    A ← A − {u};
end
B ← restrictedMinimumContagiousSet(G, k, A);
if B = null then
    return null
else
    return B ∪ {v}
```

Lastly, we analyze the time cost of this algorithm. We focus on Algorithm 5 because it is the only subroutine that makes two recursive calls. Moreover, we focus on the size of the input, $n = |A|$. In the call to Algorithm 6, which ultimately leads to a recursive call of Algorithm 5, we delete at least one vertex in A. In the next recursive call to Algorithm 5, we remove exactly one vertex from A. All the other steps in Algorithm 5 and Algorithm 6 can be completed in quadratic time. So, the recurrence relation is

$$T(n) \leq 2T(n-1) + O(n^2)$$

The solution to the recurrence relation is

$$T(n) = O(2^n)$$

Hence, Algorithm 5 is $O(2^n)$.

Hoping to prove a stronger bound, we attempted one other method of analyzing the time cost of Algorithm 5. This time, we focus on the thresholds of the input,

$$s = \sum_{v \in A} k(v)$$

In the call to Algorithm 6, we reduce this sum by deleting a vertex and decrementing the thresholds of its neighbors. In the worst case, the threshold of the deleted vertex is 1, and all neighbors are not in A, so s is decreased by 1. In the next recursive call to Algorithm 5, we decrease s by removing a vertex from A. In the worst case, the removed vertex has threshold 1, and s decreases by 1. The recurrence relation is

$$T(s) \leq 2T(s-1) + O(s^2)$$

and its solution is

$$T(s) = O(2^s).$$

With the same running time, one could easily iterate through each of 2^n vertex subsets of the given graph. The benefit of this algorithm, however, is that it exploits opportunities for major efficiency increases. Furthermore, since this algorithm is recursive, we can cache the result of calls to each subroutine, providing another speedup in our search.

Chapter 4

Empirical Evaluation

Here we present our empirical evaluation of our theoretical results. These results include the outcome of experiments that compared new and known upper bounds for the minimum contagious set of random graphs.

4.1 Our New Upper Bounds

In evaluating our new bounds (Theorem 3.4, Theorem 3.5, and Theorem 3.6), we generated 10000 Erdős-Rényi graphs, denoted $G(n, p)$ [22], for $n = 40$ and 55 different values of p. We compared the means of the cardinality of the minimum contagious set and the bounds described in Theorems 3.4, 3.5, and 3.6 for a threshold of $k = 2$. The result is described in Table 4.1.

Table 4.1: A comparison of the mean of upper bounds on minimum contagious sets for several thousand $G(40, p)$ for varying p.

p	$m(G, 2)$	Thm 2.1	Thm 3.4	Thm 3.5	Thm 3.6
0.005	39.42	39.77	39.48	39.45	39.99
0.010	38.13	39.17	38.49	38.35	39.91
0.015	36.47	38.32	37.43	37.15	39.67
0.020	34.48	37.23	36.32	35.92	39.15
0.025	32.43	36.06	35.25	34.8	38.36
0.030	30.26	34.78	34.12	33.7	37.29
0.035	28.01	33.43	32.94	32.62	35.87
0.040	25.79	32.09	31.74	31.6	34.28
0.045	23.59	30.75	30.5	30.61	32.46
0.050	21.49	29.46	29.29	29.68	30.54
0.055	19.37	28.17	28.05	28.78	28.48
0.060	17.4	26.97	26.89	27.96	26.47
0.065	15.46	25.78	25.73	27.12	24.45

p	$m(G,2)$	Thm 2.1	Thm 3.4	Thm 3.5	Thm 3.6
0.070	13.63	24.66	24.63	26.38	22.5
0.075	11.91	23.6	23.58	25.65	20.64
0.080	10.31	22.56	22.55	24.91	18.85
0.085	8.908	21.62	21.61	24.26	17.26
0.090	7.667	20.71	20.7	23.58	15.78
0.095	6.581	19.83	19.83	22.98	14.37
0.100	5.72	19.05	19.05	22.42	13.23
0.105	4.948	18.26	18.26	21.81	12.1
0.110	4.361	17.57	17.57	21.31	11.19
0.115	3.877	16.9	16.9	20.75	10.34
0.120	3.497	16.28	16.28	20.28	9.673
0.125	3.179	15.71	15.71	19.81	9.034
0.130	2.909	15.14	15.14	19.32	8.466
0.135	2.704	14.63	14.63	18.89	7.983
0.140	2.55	14.14	14.14	18.49	7.545
0.145	2.417	13.67	13.67	18.06	7.19
0.150	2.313	13.25	13.25	17.69	6.878
0.155	2.234	12.84	12.84	17.31	6.589
0.160	2.167	12.43	12.43	16.9	6.314
0.165	2.13	12.05	12.05	16.55	6.1
0.170	2.101	11.73	11.73	16.22	5.928
0.175	2.073	11.42	11.42	15.92	5.74
0.180	2.051	11.09	11.09	15.57	5.554
0.185	2.036	10.8	10.8	15.26	5.401
0.190	2.027	10.52	10.52	14.98	5.273
0.195	2.021	10.24	10.24	14.68	5.162
0.200	2.014	9.992	9.992	14.42	5.046
0.250	2.0	7.997	7.997	12.05	4.21
0.300	2.0	6.663	6.663	10.26	3.715
0.350	2.0	5.713	5.713	8.883	3.349
0.400	2.0	5.001	5.001	7.768	3.057
0.450	2.0	4.446	4.446	6.877	2.845
0.500	2.0	4.001	4.001	6.144	2.666
0.550	2.0	3.637	3.637	5.508	2.524
0.600	2.0	3.332	3.332	4.976	2.403
0.650	2.0	3.077	3.077	4.538	2.295
0.700	2.0	2.856	2.856	4.106	2.214
0.750	2.0	2.667	2.667	3.775	2.146
0.800	2.0	2.5	2.5	3.445	2.088
0.850	2.0	2.353	2.353	3.151	2.045
0.900	2.0	2.222	2.222	2.888	2.013
0.950	2.0	2.105	2.105	2.631	2.0

We also plotted these results graphically, with linear interpolation between points. The result of this is shown in Figure 4.1, plotting for each $p \in \{0.05, 0.10, \ldots, 0.95\}$, and Figure 4.1, plotting for each $p \in \{0, 0.005, 0.01, \ldots, 0.20\}$.

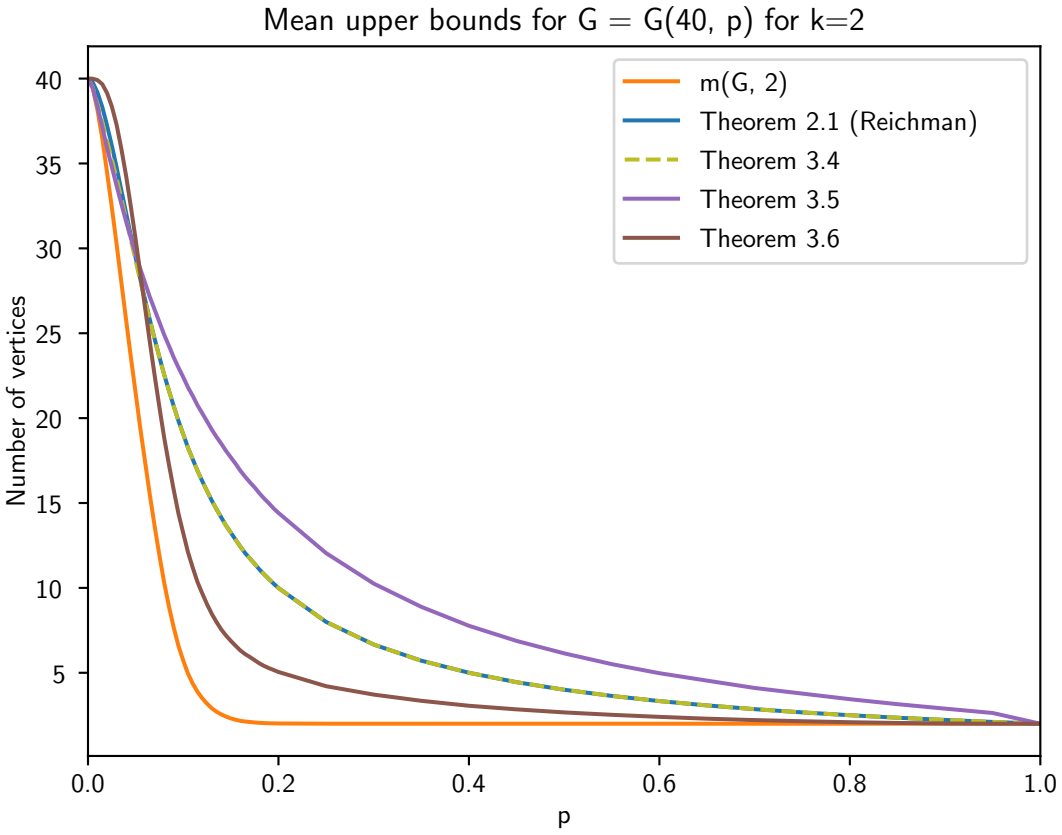

Figure 4.1: A graphical comparison of the mean of upper bounds on minimum contagious sets for several thousand $G(40, p)$ for various p from 0 to 1.

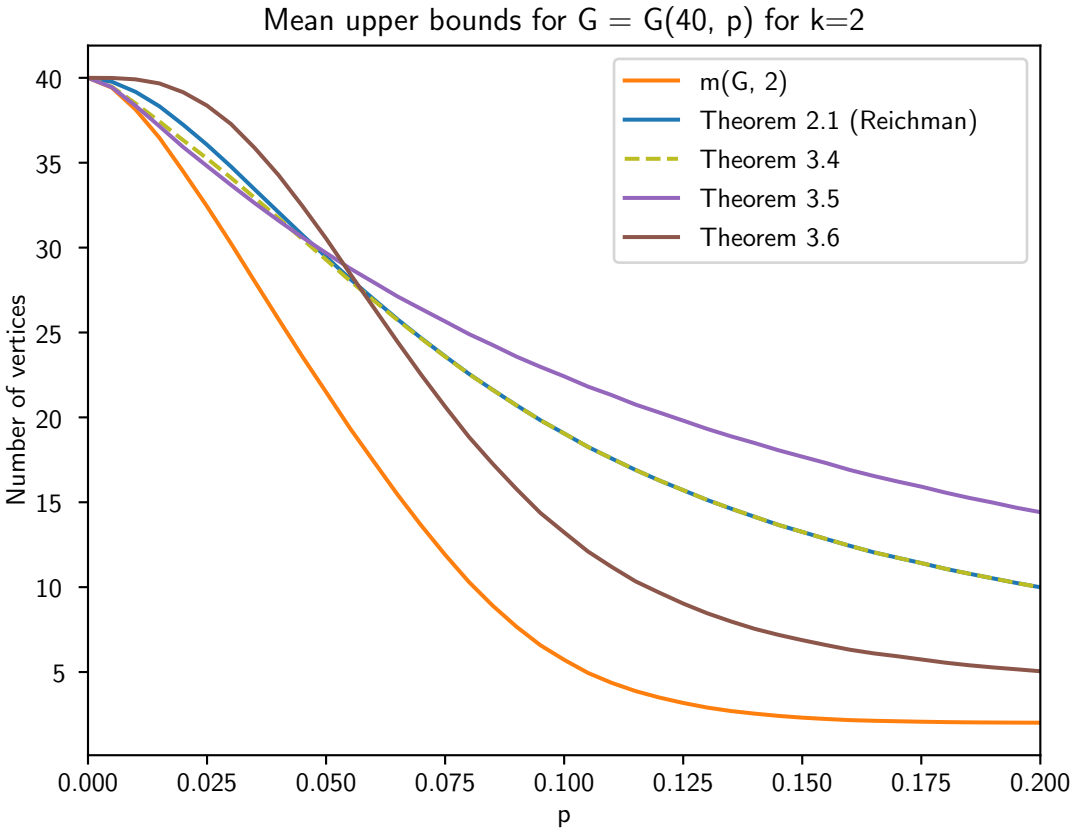

Figure 4.2: A graphical comparison of the mean of upper bounds on minimum contagious sets for several thousand $G(40, p)$ for various p near and below $\frac{1}{\sqrt{40}}$.

We see that for $k = 2$ our bounds presented in Theorem 3.4 and Theorem 3.6 on average beat the existing bound given in Theorem 2.1, and the bound from Theorem 3.6 does so by a substantial margin for graphs generated with $n = 40 and p > 0.060$.

Since our upper bounds performed well on random graphs, we continued our evaluation by comparing the bounds on graphs of real-world data.

4.1.1 Evaluation on Real-World Data

While the Erdős-Rényi model helped us evaluate our new bounds for generic graphs, these graphs usually do not resemble graphs that are used in real-world applications. In order to evaluate our bounds on real-world data, we used several undirected graphs from the Stanford Network Analysis Project (SNAP) [32] representing communities in social networks.

Our first such graph is the Facebook Ego network graph from [33]. Its vertices represent Facebook users and its edges represent a mutual friendship between two users. The result is presented in Figure 4.3.

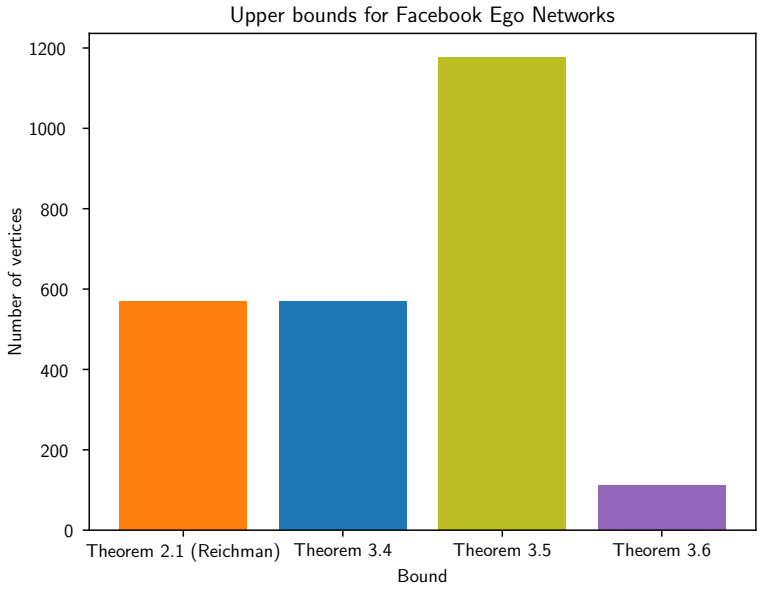

Figure 4.3: A comparison of upper bounds for the Facebook Ego Networks graph.

Our second dataset is the GEMSEC Facebook page dataset [34] (from here on shortened to GEMSEC FB). It is made up of eight graphs, each corresponding to a category of Facebook page. The categories of pages are the following: Artists, Athletes, Companies, Governments, New Sites, Politicians, Public Figures, and TV Shows. Each vertex represents a Facebook page and each edge represents a mutual like between two pages.

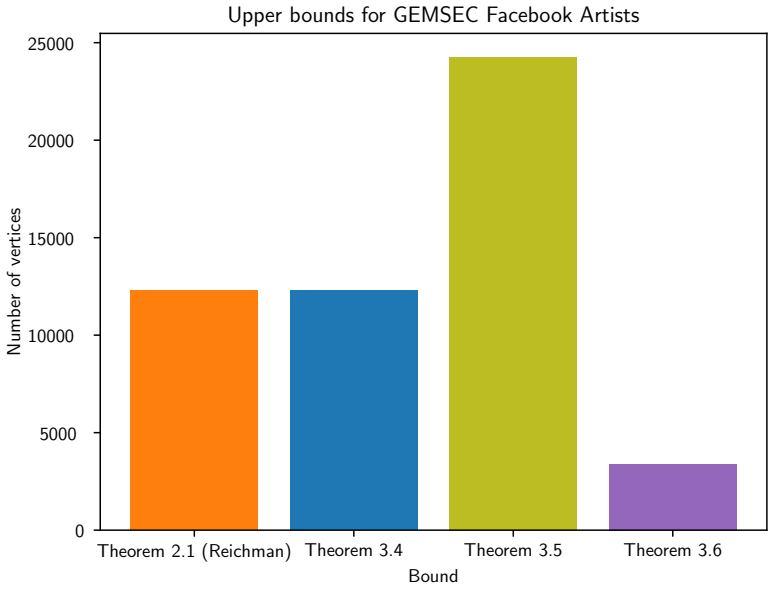

Figure 4.4: A comparison of upper bounds for the GEMSEC FB Artists graph.

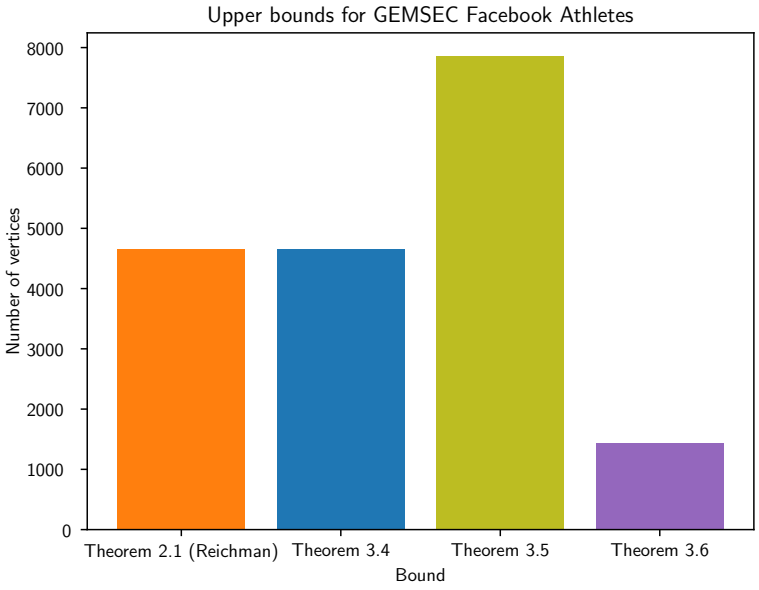

Figure 4.5: A comparison of upper bounds for the GEMSEC FB Athletes graph.

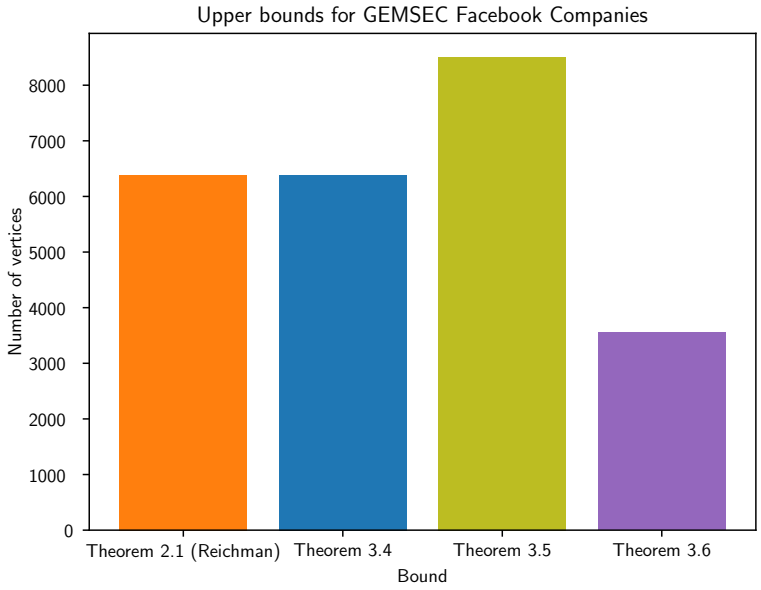

Figure 4.6: A comparison of upper bounds for the GEMSEC FB Companies graph.

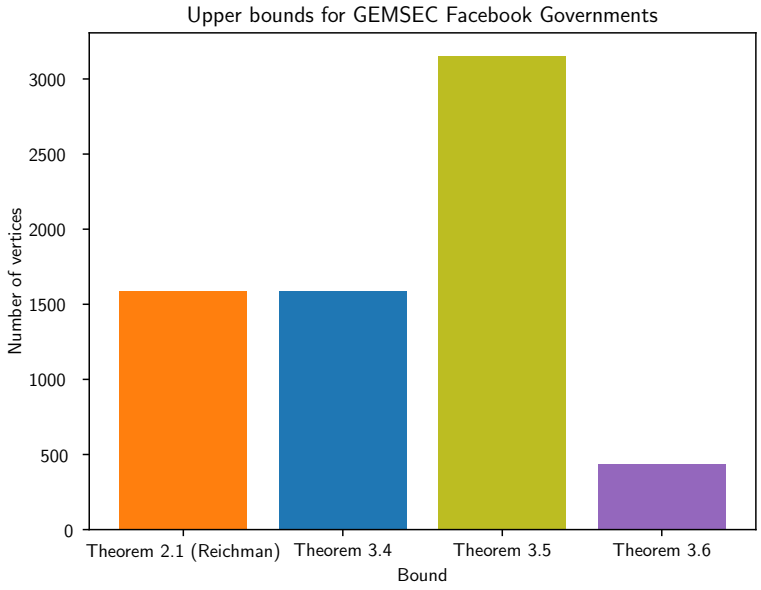

Figure 4.7: A comparison of upper bounds for the GEMSEC FB Governments graph.

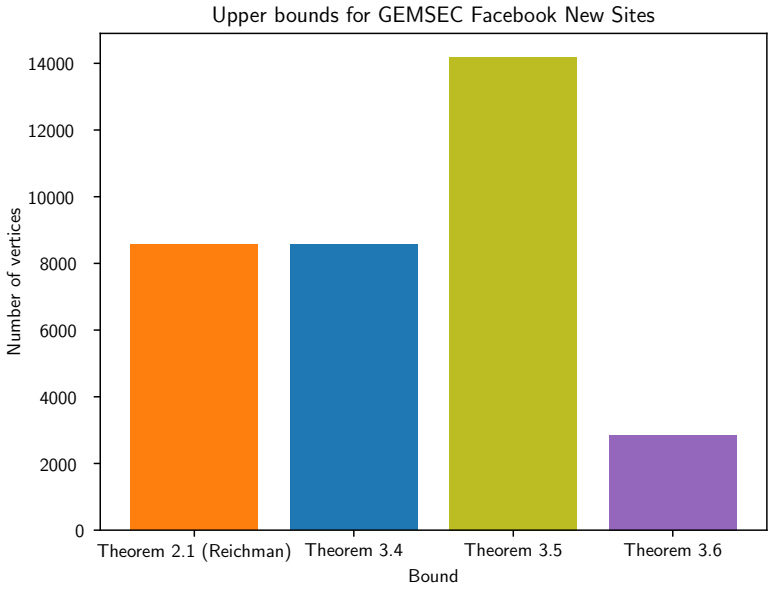

Figure 4.8: A comparison of upper bounds for the GEMSEC FB New Sites graph.

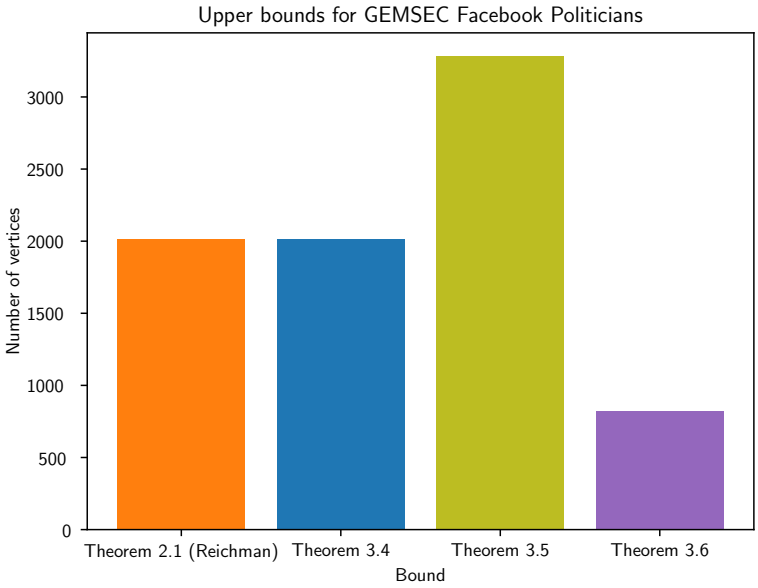

Figure 4.9: A comparison of upper bounds for the GEMSEC FB Politicians graph.

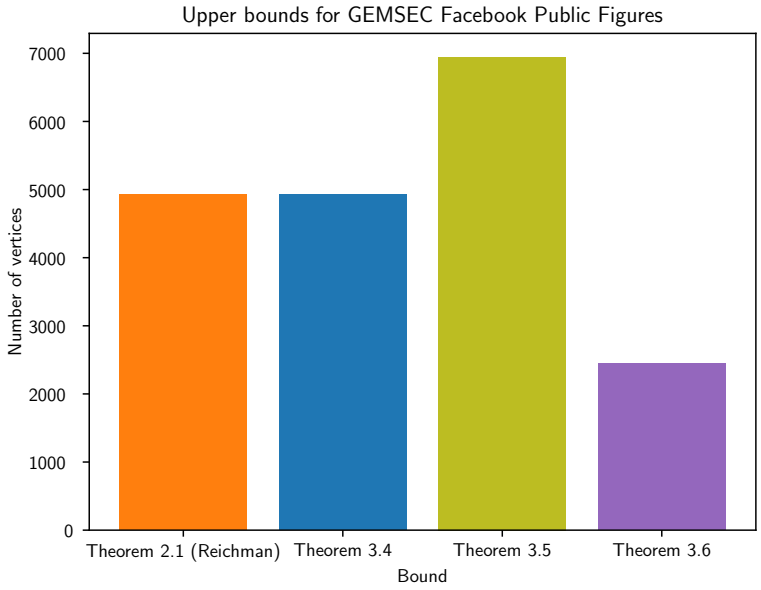

Figure 4.10: A comparison of upper bounds for the GEMSEC FB Public Figures graph.

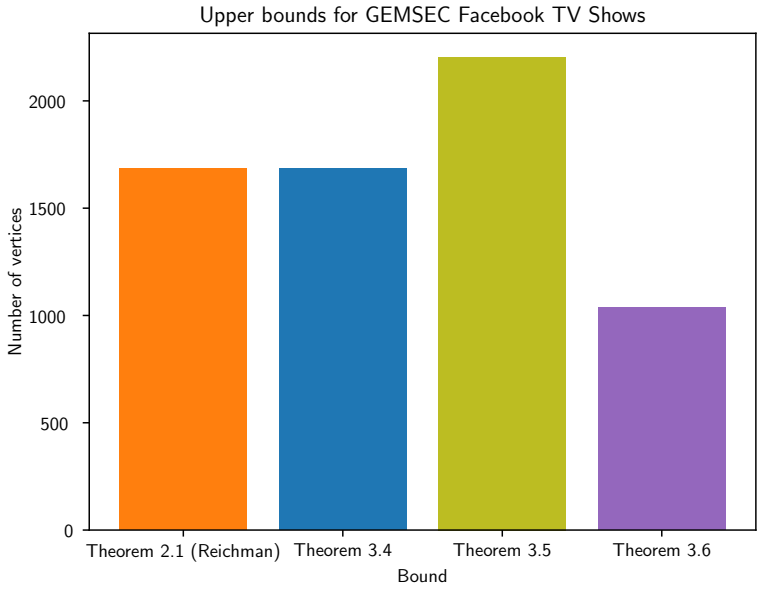

Figure 4.11: A comparison of upper bounds for the GEMSEC FB TV Shows graph.

Our third dataset used is a subset of that from MUSAE [35]. It is made up of seven graphs, one whose vertices represent GitHub users who have starred at least 10 repositories and whose edges represent a mutual follower relationship between two users, and six whose

vertices represent Twitch users who stream in a certain language and whose edges represent a mutual friendship between two users. The results are displayed in Figures 4.12 to 4.18.

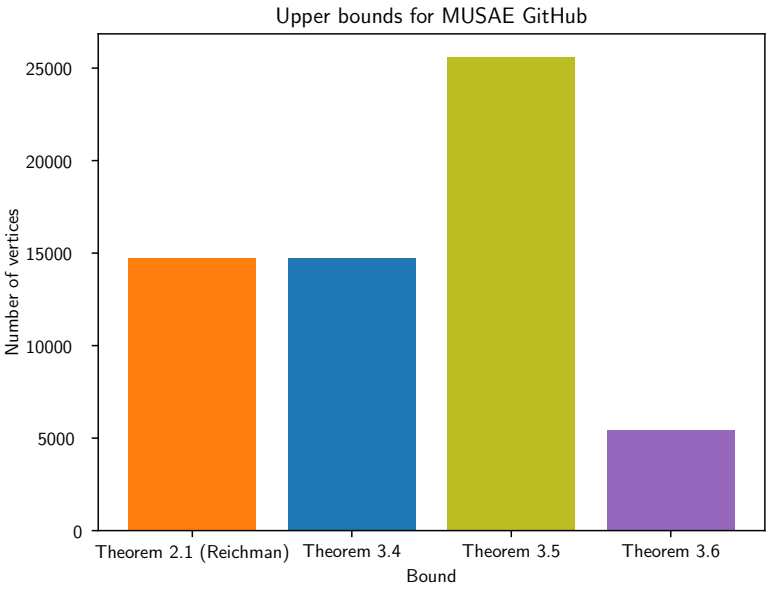

Figure 4.12: A comparison of upper bounds for the MUSAE GitHub graph.

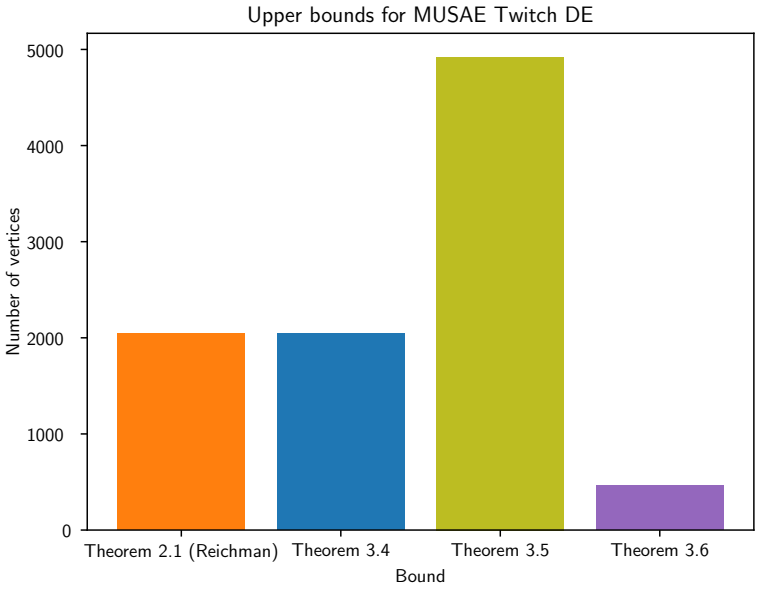

Figure 4.13: A comparison of upper bounds for the MUSAE Twitch DE graph.

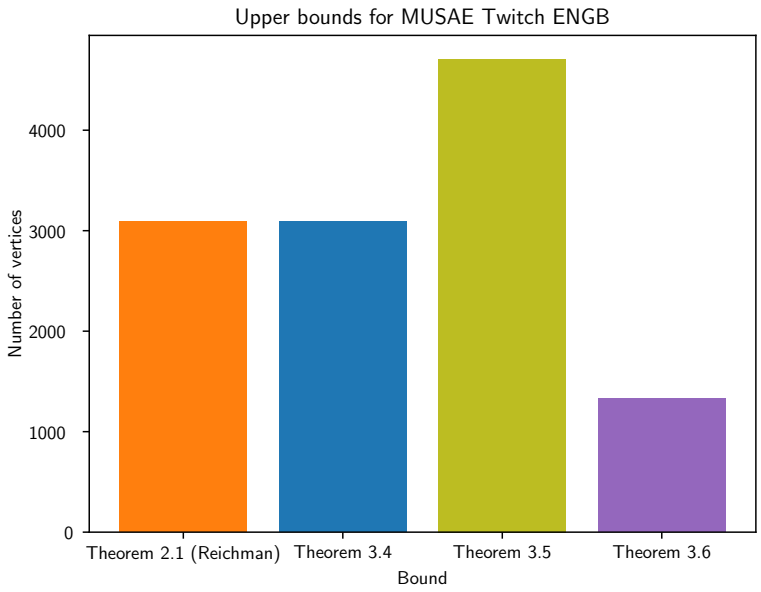

Figure 4.14: A comparison of upper bounds for the MUSAE Twitch ENGB graph.

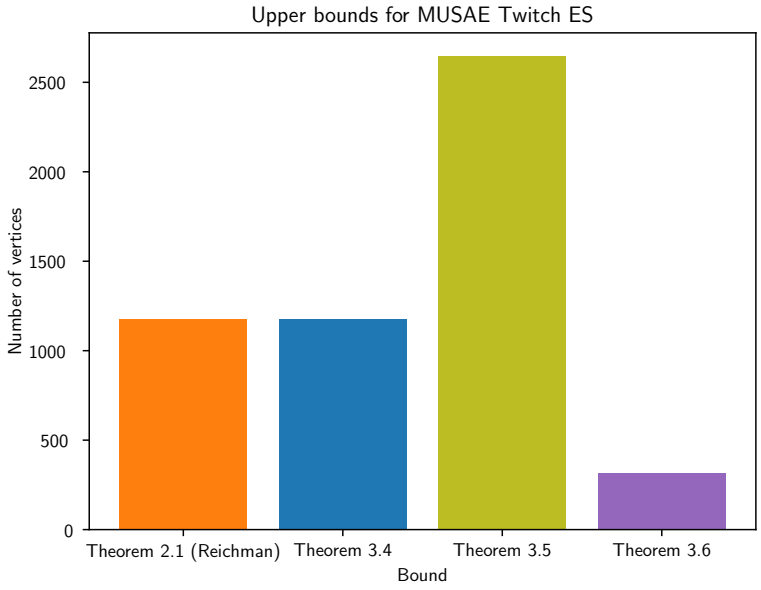

Figure 4.15: A comparison of upper bounds for the MUSAE Twitch ES graph.

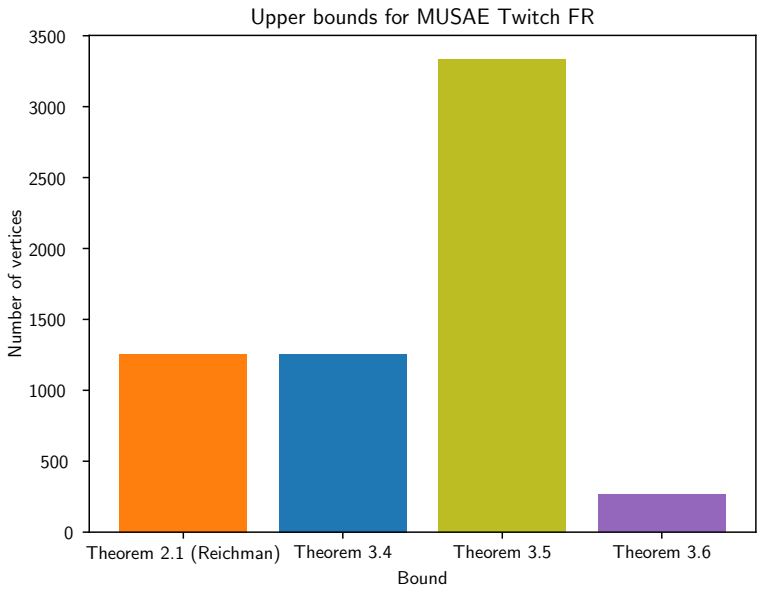

Figure 4.16: A comparison of upper bounds for the MUSAE Twitch FR graph.

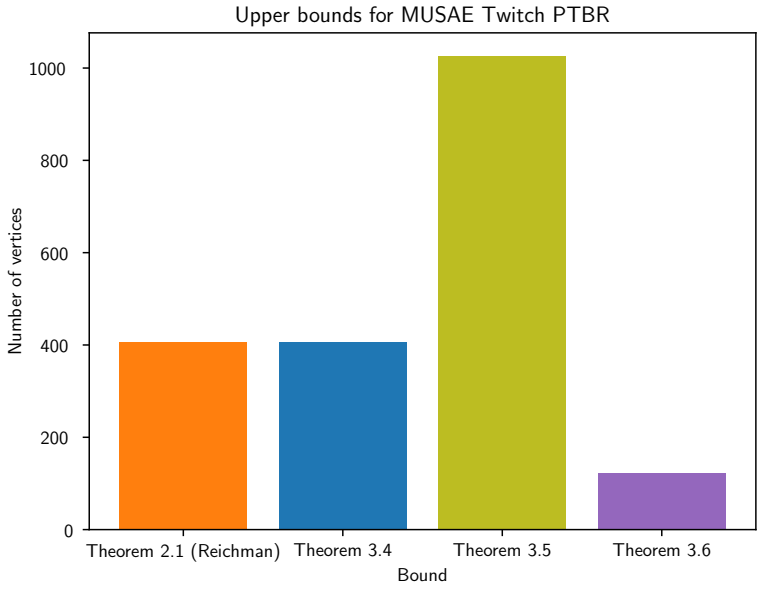

Figure 4.17: A comparison of upper bounds for the MUSAE Twitch PTBR graph.

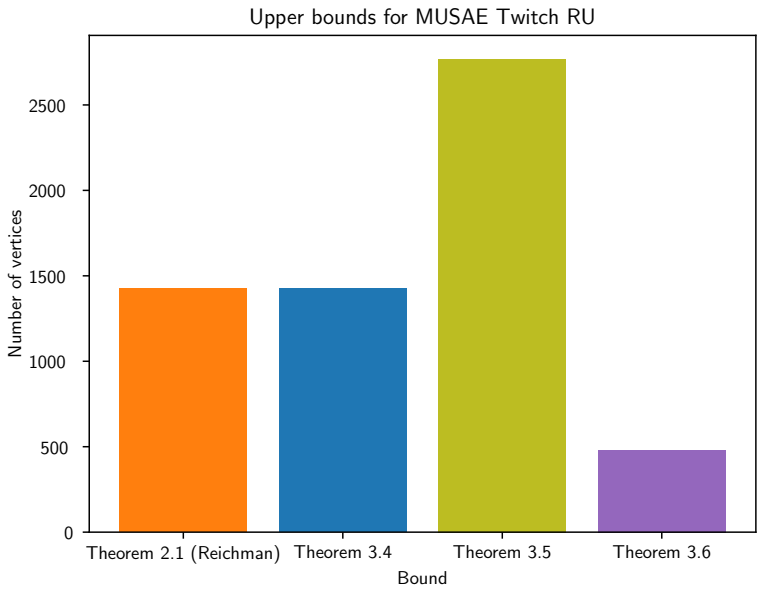

Figure 4.18: A comparison of upper bounds for the MUSAE Twitch RU graph.

Finally, we summarize the results numerically in table form in Table 4.2. On these graphs, our nonconstructive bound presented in Theorem 3.6 was by far the strongest of those tested.

Dataset	$\|V\|$	Thm 2.1	Thm 3.4	Thm 3.5	Thm 3.6
Facebook Ego Networks	4039	569.7	569.7	1177	110.9
GEMSEC FB Artists	50520	12290	12290	24260	3396
GEMSEC FB Athletes	13870	4652	4652	7852	1437
GEMSEC FB Companies	14110	6388	6388	8506	3551
GEMSEC FB Governments	7057	1590	1590	3149	437.3
GEMSEC FB New Sites	27920	8568	8568	14190	2851
GEMSEC FB Politicians	5908	2014	2014	3280	819.1
GEMSEC FB Public Figures	11560	4930	4930	6945	2445
GEMSEC FB TV Shows	3892	1683	1683	2205	1038
MUSAE GitHub	37700	14740	14740	25570	5409
MUSAE Twitch DE	9498	2042	2042	4922	463.5
MUSAE Twitch ENGB	7126	3095	3095	4708	1327
MUSAE Twitch ES	4648	1175	1175	2644	317.5
MUSAE Twitch FR	6549	1256	1256	3336	265
MUSAE Twitch PTBR	1912	405.6	405.6	1025	121.5
MUSAE Twitch RU	4385	1427	1427	2768	480.5

Table 4.2: A summary of our findings on real-world graphs.

Chapter 5

Booper Implementation

This chapter describes the methods used in the development of Booper and the user study performed during its development.

5.1 Booper

We created Booper to address the lack of availability of an interactive web application for bootstrap percolation. Booper incorporated many basic bootstrap percolation processes, providing users a graph visualization of the percolation process and a dashboard consisting of the iteration number and the tallies of active and inactive vertices. The user is able to step through the percolation process while referring to the graph. In addition, we implemented a skip-to-end feature that shows the final state of the percolation and updates the dashboard accordingly.

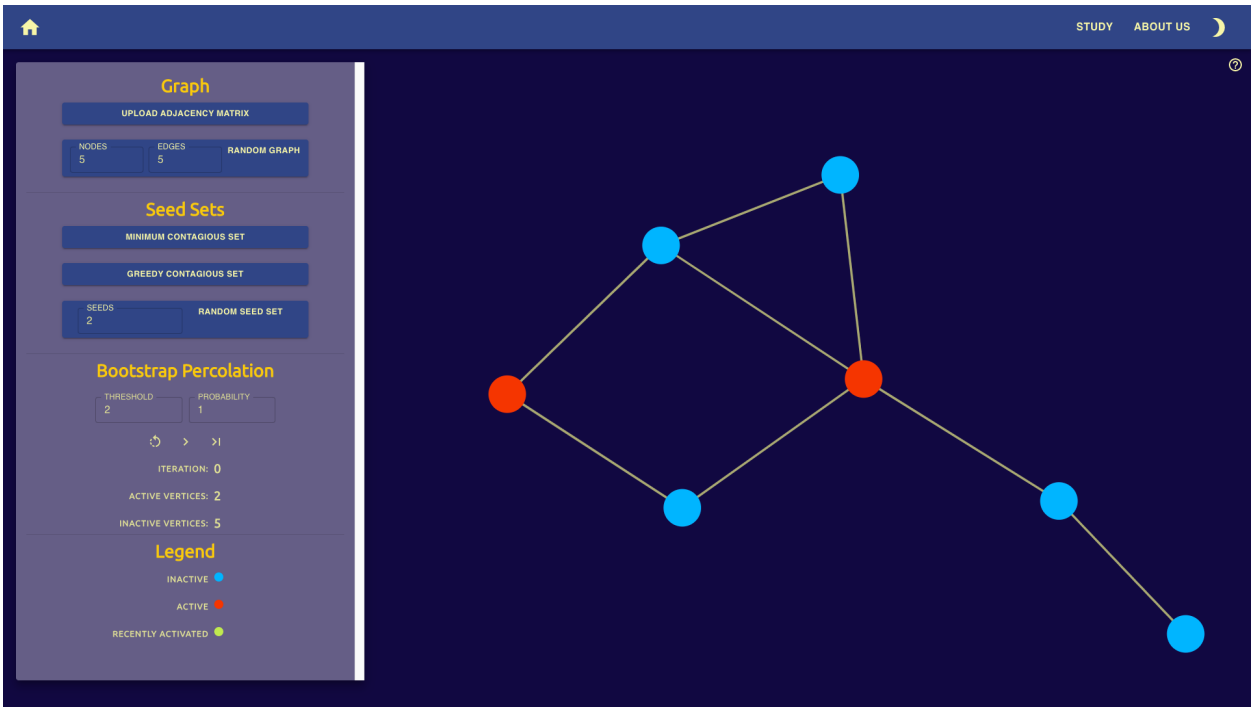

Figure 5.1: The Booper Study Page UI

Additionally, our application offers users the opportunity to find contagious sets based on our algorithms, or create a random seed set with a given number of seeds. We included one button to find the minimum contagious set of a graph and one button to find a contagious set using a greedy algorithm. Users can also specify their own seed sets and graphs by uploading a file that defines the desired inputs. This can be used in a scenario like misinformation analyses, where a user can upload a graph of their social network with predetermined nodes (accounts) being labelled as active for spreading misinformation.

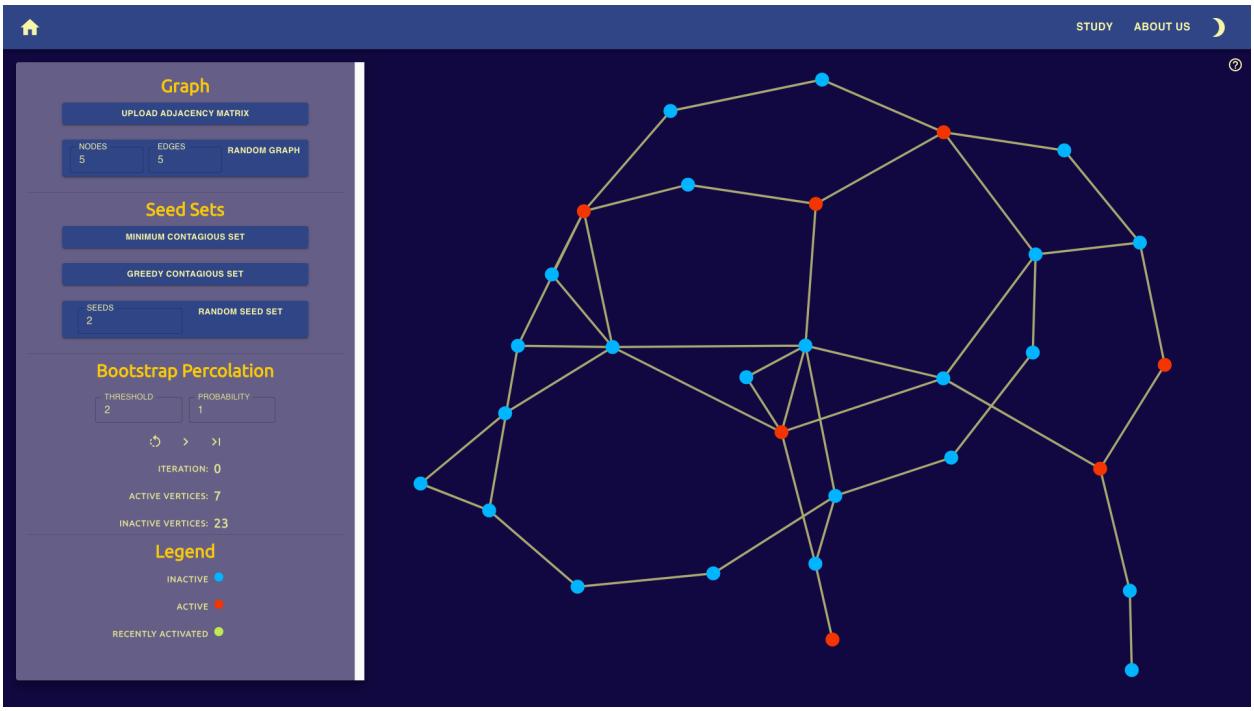

Figure 5.2: A minimum contagious set found by Booper on a 30-node graph

Users also have control over the percolation process. They can specify the activation threshold and probability of activation. These parameters can be modified prior to the start of the percolation or during the percolation, again allowing for user customization of the process. This flexible model can be used to address nuances in applications of bootstrap percolation. For example, during the COVID-19 pandemic, the probability of infection is likely to vary as time passes by, due to the efficacy of measures like social distancing and mask-wearing. The user can accordingly increase or decrease the infection probability after a certain number of iterations to model this effect.

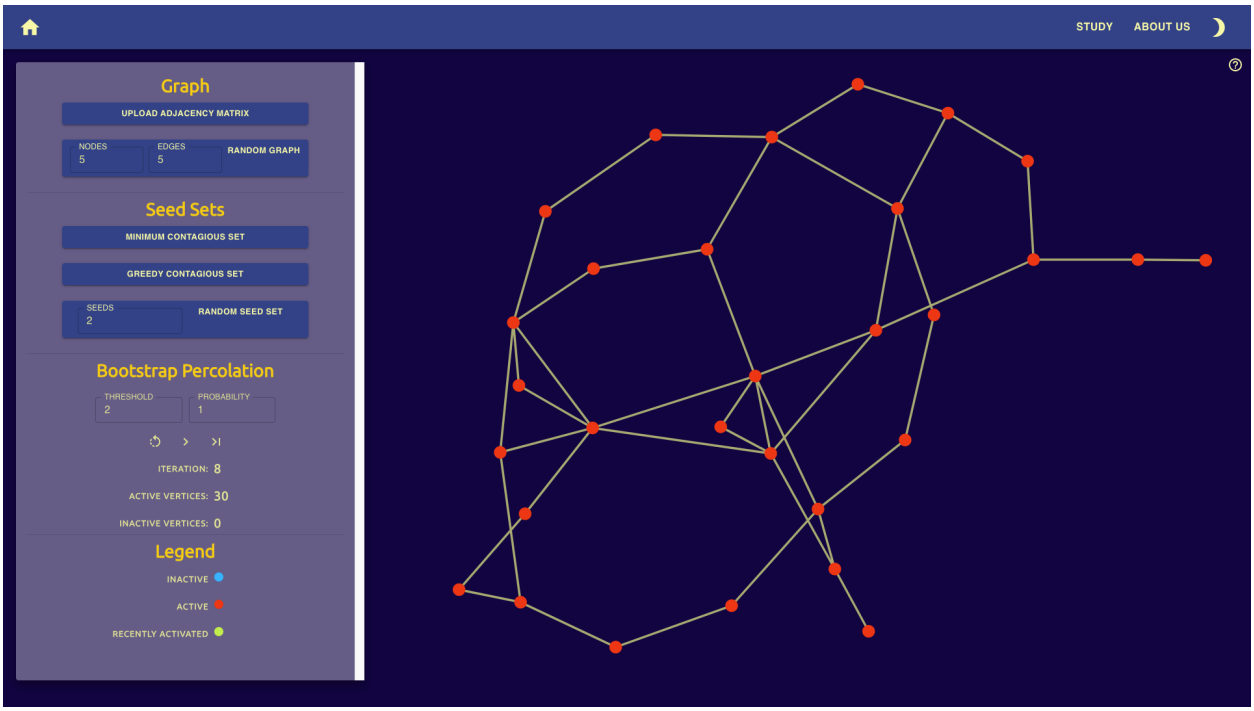

Figure 5.3: A skip-to-end view of a probability 1, threshold 2 percolation of the minimum contagious set. This took 8 iterations.

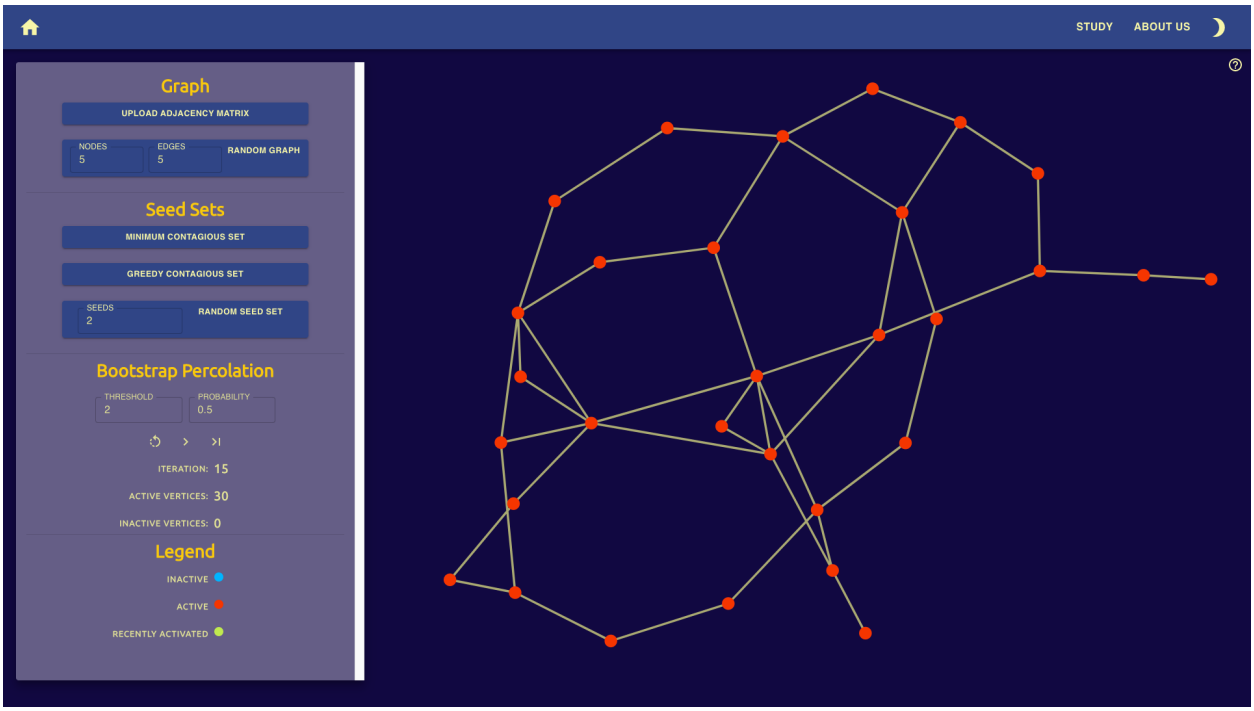

Figure 5.4: A skip-to-end view of a probability 0.5, threshold 2 percolation of the minimum contagious set. This took 15 iterations.

Outside of the implementations for the mathematical processes, we created a theme switcher as a quality-of-life feature. This feature switches between applying a dark theme, which is the default theme, and a light theme. Although this does not have a scientific utility, it still provides the opportunity for additional user engagement and customization.

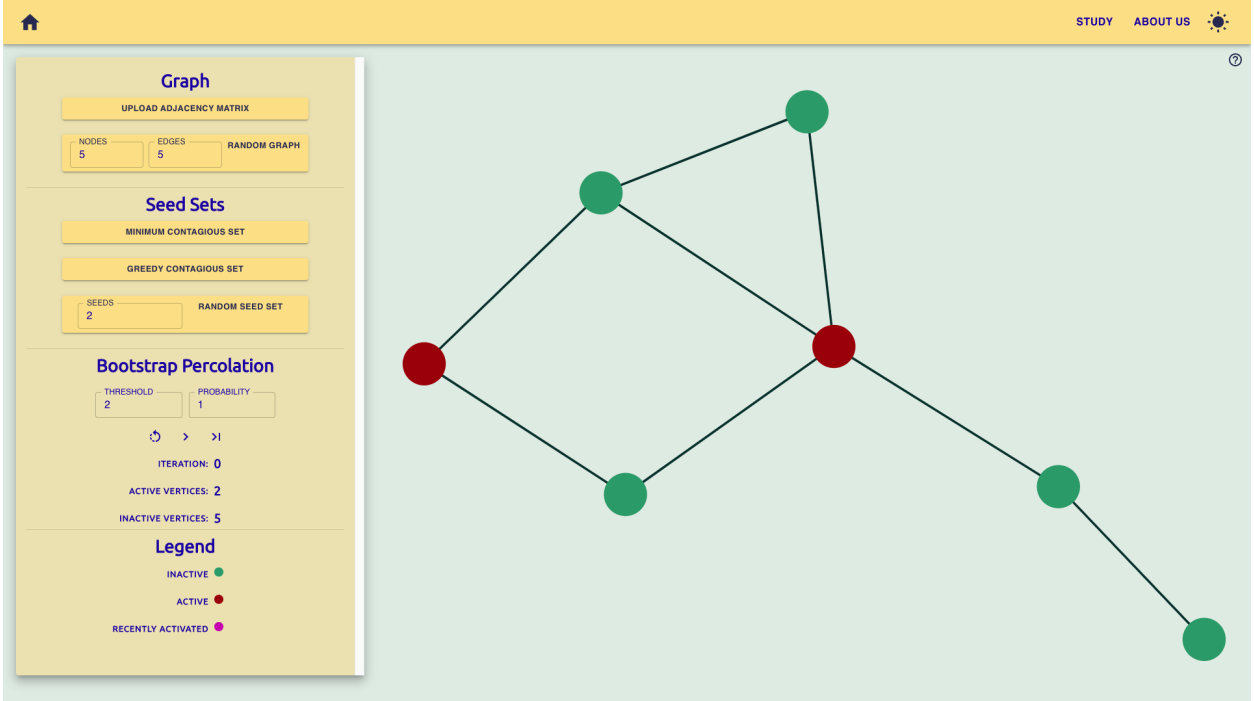

Figure 5.5: Booper with the Light Theme

5.2 Development

This section will describe the implementation details, technologies, and techniques behind the development of Booper.

5.2.1 Initial Research and Development Choices

The first step in creating the application was to formally define what goal we wanted it to fulfil. We started by brainstorming an **epic**, a definition of a product's purpose in a way that can be broken down into individual pieces of work called **stories**. In our initial meeting, we developed the following epic:

> This application will help people learn about bootstrap percolation and key results in it. It will also serve as a tool for people researching bootstrap percolation.

From there, we created multiple user stories. Each one referenced the use of an implementable feature in the application. Some stories included:

> As a Booper user studying bootstrap percolation, I want to be able to upload my own graphs and visualize the notable results for my graph.

> As a Booper user studying bootstrap percolation, I want to be able to generate random graphs and analyze results.
>
> As a Booper user studying bootstrap percolation, I want to be able to modify the parameters for percolation.

We created the name "Booper" from the first three letters of the words "bootstrap" and "percolation".

Next, we had to choose a framework. To do that, we needed to decide what type of application we wanted Booper to be. The two main choices here were a web application or a standalone application. We chose to make Booper a web application because of the increasing popularity of frameworks for web applications. In addition, web applications are easily accessible by anyone with access to the internet running a modern browser, and they do not require installation. Therefore, this choice allowed us to fill in the gap in publicly available applications mentioned in Section 2.9.2 most accessibly.

To ensure we could all contribute to Booper and to maintain the software's version control, we used Git. We chose GitHub to provide cloud hosting for the repository we created.

As one of the aforementioned increasingly popular frameworks for web applications, React was our choice for a front-end library. Further research also led us to finding Material-UI, a library with interactive components for user interface design, and D3.js, a data visualization library with functionality for dynamic graph visualizations. With the combination of these libraries, we could create an interactive UI with an intuitive user experience. We developed a UI mock-up, pictured in 5.6, to establish a vision for our design.

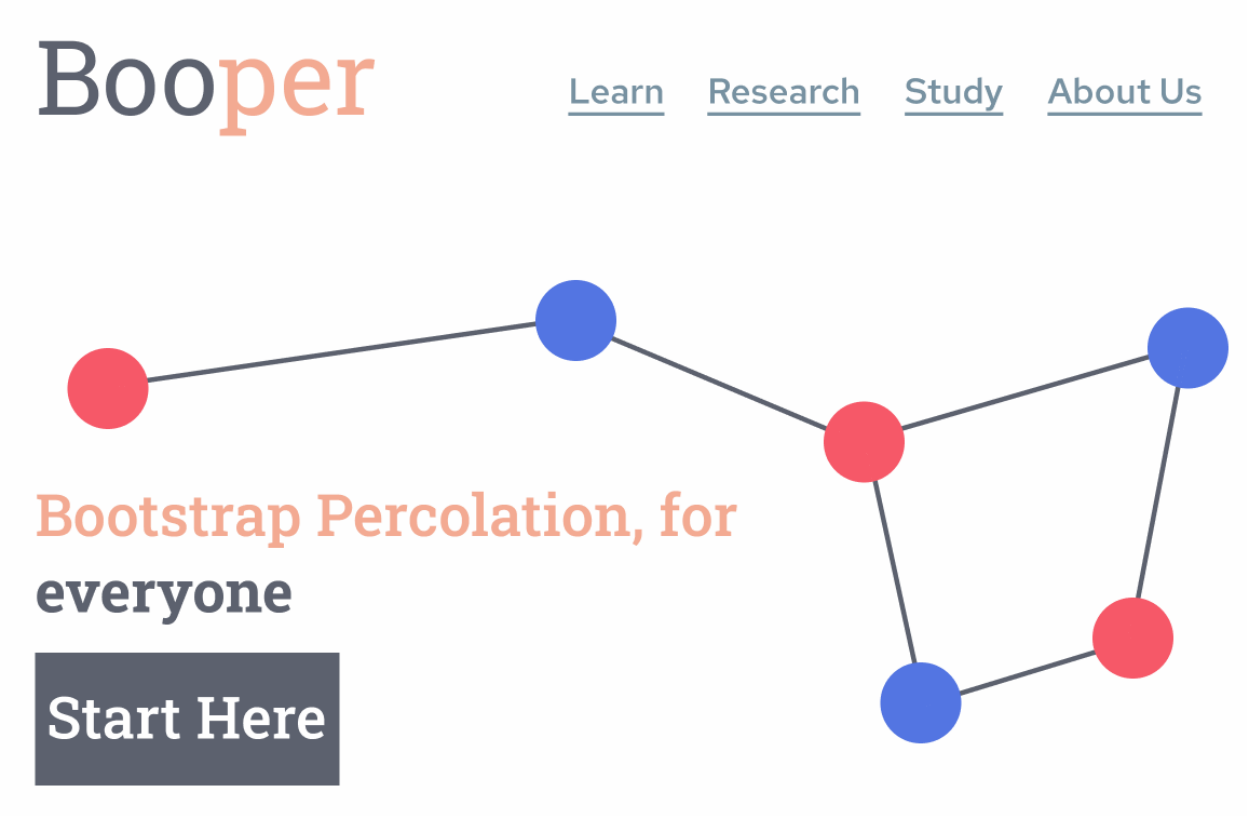

Figure 5.6: An initial mockup of Booper

5.2.2 Project Management and Development Processes

To structure our development of Booper, we decided to adopt **agile** project management practices. This involved week-long **sprints**; at the beginning of the week, we decided which features should be implemented, and we set a goal to complete them by the end of the week.

We used GitHub Projects to manage our project and track the sate of all our user stories. We created four lists in the agile board: "To Do", "In Progress", "Ready for Review", and "Done". These lists contained **notes**, which are ideas we devised for potential features or fixes, and **issues** (also called **tickets**), which are notes that were formalized into implementable tasks. Every user story we developed, whether it be through a brainstorming session or feedback from a meeting, was initially written as a note in the "To Do" list.

At the start of each development sprint, we conducted a **sprint planning** session using the storyboard. This involved converting notes to issues, ranking the issues based on level of priority, and assigning the issues. We would also move any tickets being undertaken into the "In Progress" list.

Later, when a ticket had an implemented solution, we moved it to the "Ready for Review" list. We used **pull requests** to review each other's code and merge it together. These code

reviews were essential for catching mistakes, maintaining a consistent style, and generally keeping each other in check. Then, when a solution to a ticket was merged into our main branch, we moved the ticket to the "Done" list in the storyboard.

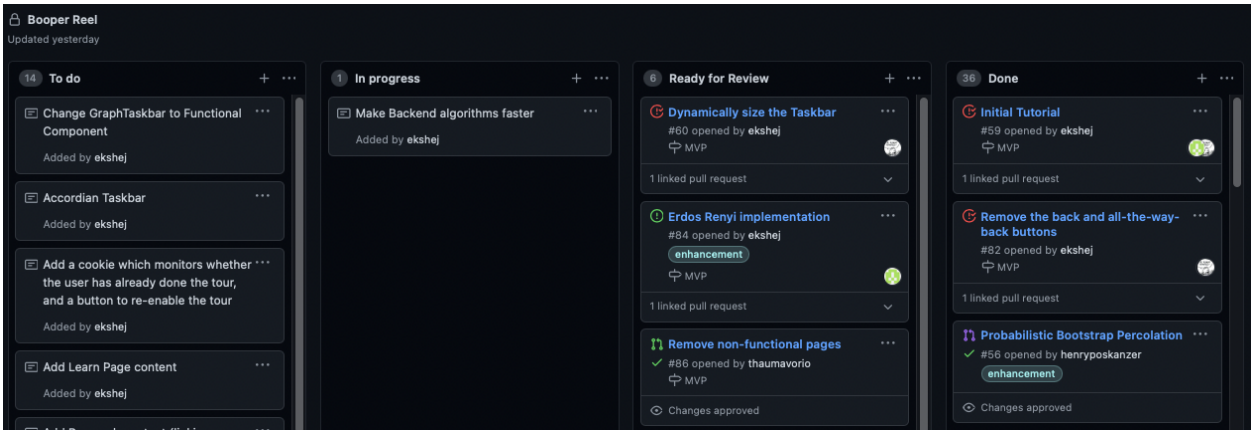

Figure 5.7: A snapshot of the Booper storyboard during a development iteration.

To check in on each other's progress each week, we held at least three **scrum** meetings each week. During this time we could share any bugs we faced or any ideas we had. In addition, we had weekly meetings with our advisors that functioned like **sprint review** sessions. This gave us a chance to showcase our features and improvements from the prior week, while also being a useful time for ideation and feedback. After every sprint review session, we would meet within the next two days to complete a **sprint retrospective** meeting. During these, we evaluated both what went well and what could be improved about the past sprint.

Our development process was integrated with our storyboard in a way that allowed us to link an issue, or multiple issues to a solution. The structure of the git repository was made to emphasize that the code in the "master" branch was always functional, and that initial fixes should be pushed to the "dev" branch first before going to the master branch. To develop a fix, a branch was created from the dev branch. The branch was named starting with one of three descriptors: "bugfix", "feature", and "techdebt", and ending with the name of the issue. Branches with the bugfix descriptor were intended to solve a bug, while branches with the feature descriptor were intended to introduce a new feature to Booper. Branches with the techdebt descriptor involved documenting code or cleaning up its structure and layout.

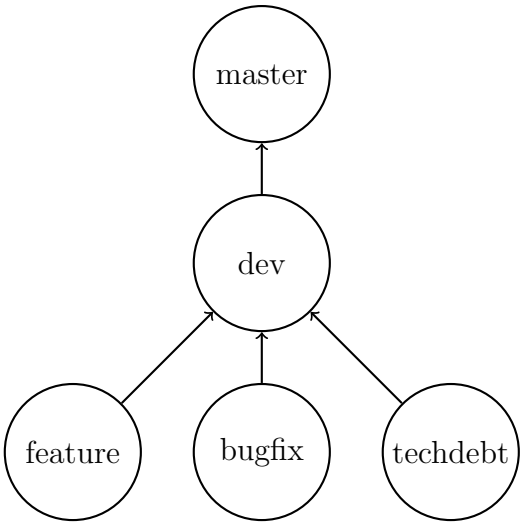

Figure 5.8: A visual showing how we structured our git repository.

Once the solution is implemented and ready to merge with the dev branch, a pull request was submitted. GitHub allows for linking an issue, or multiple issues, to a pull request. Therefore, it was clear what exactly the pull request was supposed to solve. For each pull request, the two other members of the team were assigned to review the pull request, and required to approve it for the pull request to be merged. However, they occasionally requested changes to a pull request instead, requiring the author to rewrite parts of the pull request so that it could be reviewed again.

5.2.3 Implementation Details

This subsection will cover the code and packages used to develop Booper. As mentioned in Section 2.9, JavaScript-based web applications are becoming increasingly popular. Booper, a JavaScript-based web application written using the React framework, is part of this trend.

Server

After implementing our algorithms for use in experiments and evaluation, we reused this code in a web back end, written in Haskell, presenting a REST API with functions for finding the minimum contagious set of a graph (using Algorithms 4, 5, and 6), and finding a small (but not necessarily minimum) contagious set using Algorithm 1.

Classes

The back end of Booper has one class, the *Graph* class. This class represents graphs using the adjacency list implementation. It contains a **Map** data structure, where the keys are vertices in the graph, and the values are the neighborhoods of the keys. The vertices were

represented by numbers, and the neighborhoods were represented by **Set** data structures containing other vertices.

This class also has methods to retrieve data from the aforementioned server. This was achieved through **POST** requests, which returned **promises**. A POST request is a way of sending a request of arbitrary size to a server. A promise is an object, representing the completion, and possible failure, of an asynchronous task (here, accessing a web API). In this method, a POST request was made to a URL corresponding to the desired algorithm (e.g. the *minimumContagiousSet* function made a POST request to a URL corresponding to the *minimumContagiousSet* function in the server, which performed Algorithm 4) that returned a promise. When the promise was fulfilled, the data containing the active and inactive vertices for the graph was sent in a response object to be rendered.

Components

React is a framework based on the use of components, independent pieces of code that can be reused to help achieve regular patterns in a UI. Booper uses components to define the application's pages: the home page, Study page, and About Us page. The pages themselves are also components.

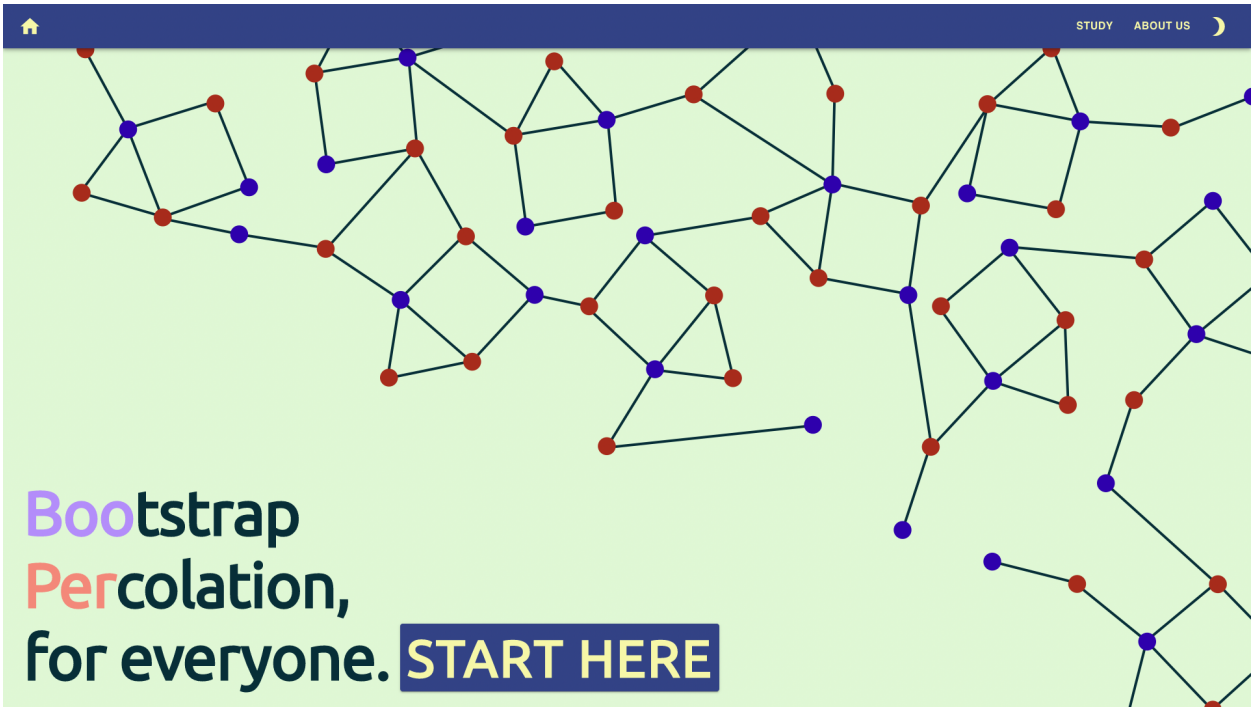

Figure 5.9: The Booper Home Page. This component is partly assembled of transition and typography components from Material-UI.

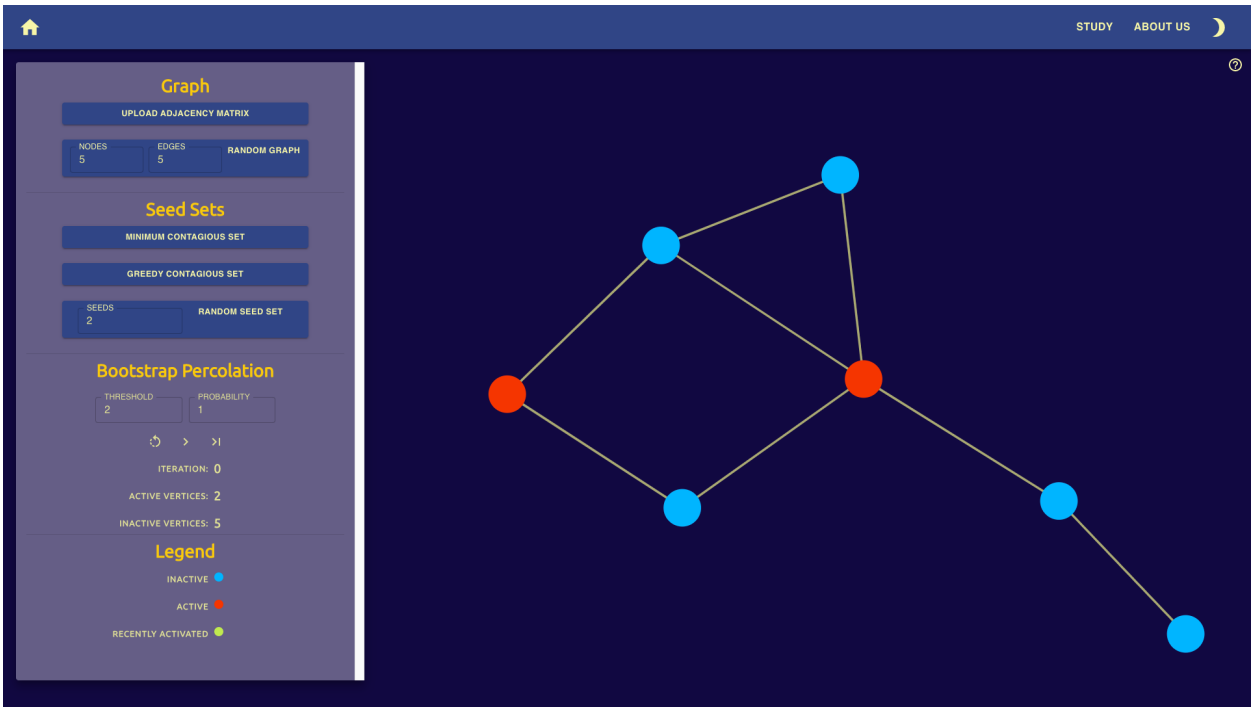

Figure 5.10: The Booper Study Page. This component solely consists of the *ForceGraph* component, which contains components for the taskbar and the d3-based graph implementation.

Figure 5.11: The Booper About Us Page. This page contains the authors' information as well as a link to the GitHub repository.

The most important components in Booper are the *ForceGraph2D* component and the *GraphTaskbar* component, which are combined to form the ForceGraph component.

The *ForceGraph2D* component was imported from a package that represented a graph data structure using the d3 force engine. It also implemented graph zooming, panning, and node dragging features. The result is an interactive graph UI that allows for intuitive movement and analysis.

The *GraphTaskbar* component is a component whose button operations can update the *ForceGraph2D* component, which in turn updates the visual display. It can also render new graphs through either a file upload or generating a random graph from the fixed edges variant of the Erdős-Rényi Model. There are buttons to find and display the minimum contagious set, the contagious set from Algorithm 1, and a random seed set based on user-specified parameters. These are accompanied by a button group to iterate through a bootstrap percolation step (with a user-specifiable threshold and activation probability), skip to the end (the iteration where there are no more vertices that can be activated), and a reset button that makes all vertices inactive.

5.3 User Study

After the creation of a minimum viable product, we planned a user study to investigate how we could improve Booper for its users. The goal of this study was to gather opinions on the appearance, ease of use, accessibility, and desired features in Booper. These opinions helped us evaluate the overall quality of Booper, and they can help future project teams further improve the design of Booper.

5.3.1 Survey Design

On the first page of our survey, we defined bootstrap percolation in informal terms, and we explained what Booper is. Then, we provided instructions for how to access and use Booper, requesting that survey participants spend at least five minutes exploring Booper before answering our survey questions.

The rest of the survey was made up of 5-option Likert scales [36] and free response questions. Likert scales allowed us to collect participant opinions quantitatively, while requiring a small amount of effort from participants. Free responses allowed us to collect these opinions in a detailed form; participants could write in specific terms what they liked or disliked about Booper. The survey questions are available in Appendix A.

We are interested in how feedback about Booper might be related to the demographics of users. So, we included a few questions at the end of the survey asking participants about their background and occupation. We also wanted to protect the privacy of our survey participants. We made it explicit that these demographic questions were completely optional, and they did not include any personally identifying information.

We designed and administered our survey using Qualtrics. Qualtrics provided customizable survey elements and a professional-looking theme. Furthermore, Qualtrics automatically analysed of our survey, suggesting improvements and estimating the time it takes to complete the survey.

5.3.2 Survey Administration

We emailed our survey to undergraduate students, graduate students, and faculty in the computer science and mathematical sciences departments at WPI. In addition, we emailed it to several professionals in the software industry and a few mathematicians who specialize in bootstrap percolation. No part of this survey involved face-to-face interaction between us and the survey participants; it was administered entirely online.

In keeping with the Institutional Review Board's policies, we also sent out a consent form. Survey participants were asked to read and electronically sign this form before completing the survey itself.

We incentivized participation in this survey with an optional raffle for six $25 Amazon gift cards. Since we needed a way to contact the raffle winners, we included a survey question allowing participants to provide an email address. This information was used solely for the purpose of the raffle, and was excluded from any data used in our user study. Participants

were notified of the purposes of collecting email addresses. They were not required to provide their emails; if they decided against it, they could still send us feedback about Booper which would be used in our study, but they were not entered into the raffle.

5.3.3 Survey Results

Our survey received a total of 42 responses.

86% of respondents felt that they understood the application's features at least moderately well. The responses to this question are plotted in Figure 5.12. 94% of respondents with prior exposure to graph theory felt that they understood the features at least moderately well while only 43% of those with up to minimal exposure to graph theory felt so.

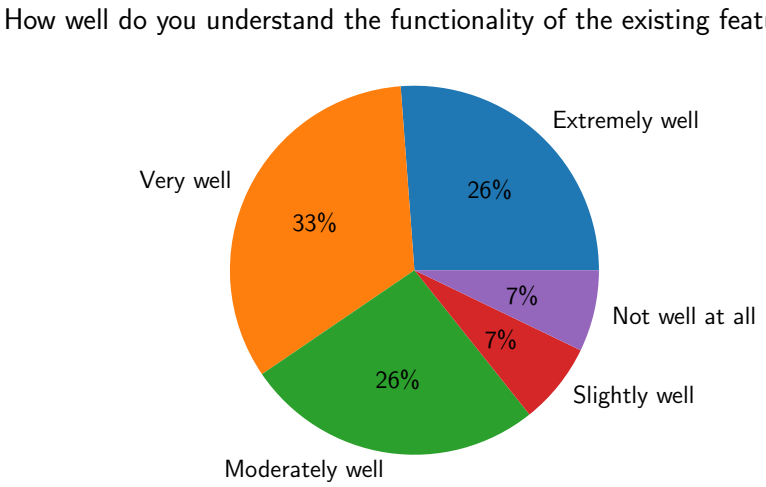

Figure 5.12: A summary of how well respondents claimed to understand the application's features.

95% of respondents followed the initial tutorial. Of these respondents, 95% found the tutorial helpful. Their responses are displayed graphically in Figure 5.13. 94% of respondents with prior exposure to graph theory who completed the initial tutorial found the tutorial helpful while 100% of those with up to minimal exposure to graph theory found it so. 55% of respondents who completed the tutorial felt that it had precisely the right amount of detail, 30% that it had slightly too much, and 15% that it had slightly too little. No respondents found that it had far too much or far too little. This is summarized graphically in Figure 5.14.

With the far majority of respondents able to understand our tutorial and find it helpful, the current tutorial is certainly a success. Where our application does not yet fully deliver is in its explanation of features to those who do not have graph theory experience, as the majority of respondents who reported having minimal exposure to graph theory felt that they did not understand Booper's features.

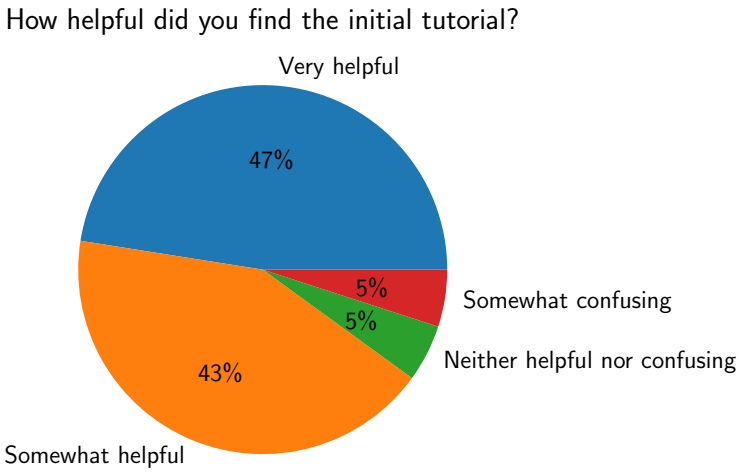

Figure 5.13: A summary of the respondents' opinions on the helpfulness of the tutorial

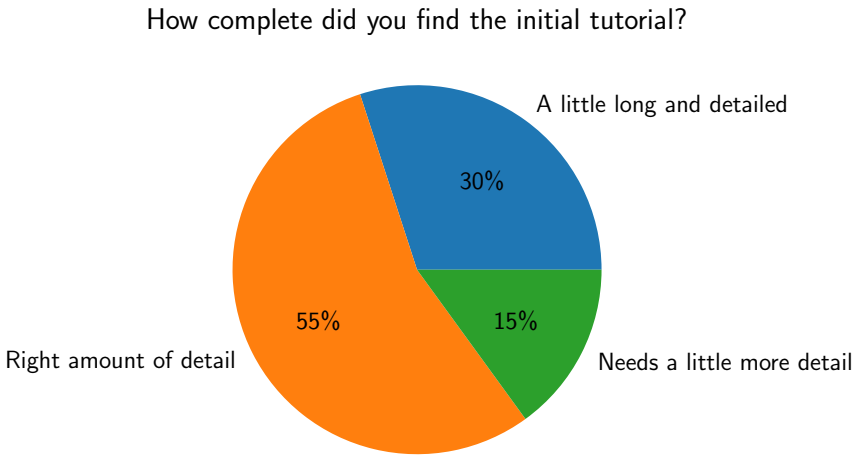

Figure 5.14: A summary of the respondents' opinions on the completeness of the tutorial

As for aesthetics, 98% of respondents liked the layout of the application, 45% liking it somewhat and 52% liking it a great deal. 90% of respondents found the dark theme pleasing and 49% of respondents found the light theme pleasing. The responses on the dark and light theme are summarized in Figure 5.15 and Figure 5.16. A common criticism of the light theme, however, was that the color of edges was too dark and made it difficult to differentiate between vertex types and positions.

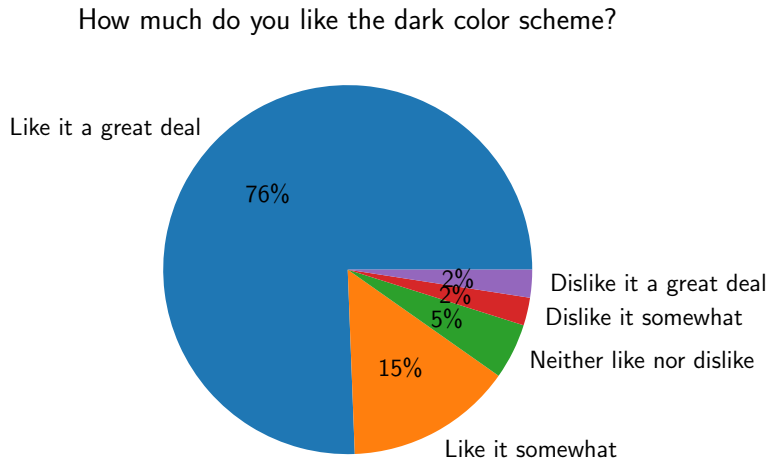

Figure 5.15: A summary of how respondents found the dark theme.

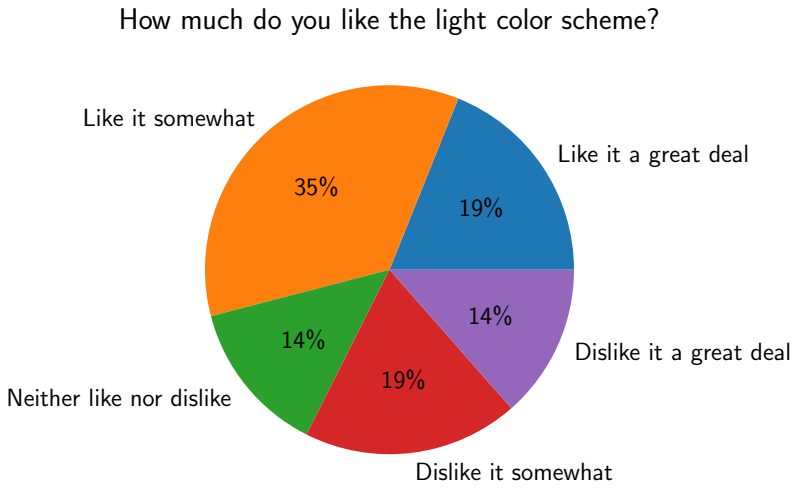

Figure 5.16: A summary of how respondents found the light theme.

The most commonly mentioned issue with the aesthetics of Booper was the existence of the scroll bar on the task bar. Two possible fixes for this would be to dynamically size and space the contents of the task bar or to create an accordion layout of its contents.

We made sure to provide space for users to suggest features they felt were missing from Booper during their evaluation. Among the most highly recommended features were in-application graph editing, clicking to mark nodes as active or inactive, and pinning the graph in place.

The in-application graph editing feature would make for a good pairing with the graph file upload feature. The file format required is unique to Booper, and may be difficult to edit

by hand for large graphs. Thus, allowing for in-application graph editing, as well as exporting a graph file would likely create a satisfying feature set for graph description. Additionally, allowing for users to click to mark nodes as active and inactive would likely pair well with this. The feature could serve both as a method for simply editing the graph, and also as a one-off utility for users to fix possible mistakes in an already loaded graph.

The respondents' recommendation to be able to pin the graph in place in reality was split in two; while some requested to be able to pin single nodes, some requested to be able to pin the entire graph, or to stop any sort of non-interactive graph movement from occurring at all. Therefore this request requires two separate features: the pinning of single nodes, and the pinning of entire graphs. Together, these two features would satisfy the respondents' recommendations.

Chapter 6

Conclusion and Future Work

The overarching goal of this project was to create mathematical results and a software solution relevant to bootstrap percolation. Based on our background research and reading, we chose to further investigate the minimum contagious set problem to find mathematical results. To complement these mathematical findings we decided to create, to our knowledge, the only web application for bootstrap percolation on the internet.

We devised novel theoretical results for the NP-complete minimum contagious set problem using three approaches: creating efficient and optimal algorithms for special types of graphs, proving new upper bounds (or approximations) for the minimum contagious set for any graph, and creating optimal algorithms to find the minimum contagious set for any graph at the cost of efficiency. We elucidated the importance and efficacy of our results, particularly Theorem 3.6, by analyzing the results of our bounds on the minimum contagious set for both random graphs and real-world data. Furthermore, the ideas we used in proofs on new bounds for the minimum contagious set for all graphs were also applicable to k-degenerate graphs and independent sets.

We created Booper to address the gap of a publicly available research tool for bootstrap percolation, and to provide visualizations of our results. We employed agile development processes to maintain product ideas and and establish the direction of our development. In addition, we used popular technologies and libraries like React and Material-UI to ensure the look of the product was modern. The vision of this product was to educate people about bootstrap percolation, and how it can serve as a model for contagion. Booper does exactly that, significantly decreasing barriers to initially understanding bootstrap percolation by providing a user interface that makes it both visually satisfying and appealing to analyze contagion on graphs. To further inform our future development of the application, we collected information and feedback from potential users through a survey. In keeping with the vision, the source code repository for Booper is public and available for anyone to contribute.

Fusing our mathematical results and software solution together in this project makes it both holistic and accessible to a wider audience. This is beneficial since the applications of bootstrap percolation are topical (e.g. modeling COVID-19 contagion, modeling the spread of fake news) and will continue to be relevant so long as understanding spread in a community is relevant.

While our project has made several useful contributions to the field of bootstrap percolation, there is always room for improvement. Regarding optimal solutions for specific graphs, we recommend considering more thresholds. We proved efficient, optimal solutions for the minimum contagious set of grid graphs with $k = 2$, but we think that similar solutions could also be proven for greater thresholds. For example, the minimum contagious set of a triangular grid graph is non-trivial when $k \leq 6$. In addition, characterizing all graphs with a minimum contagious set of size 2 for a threshold of 2, such as those in triangular grid graphs, can be an interesting task to undertake.

Our upper bounds for contagious sets could be improved as well. Since Theorem 3.6 usually provides the strongest upper bound, we recommend focusing attention on this bound. We think that it could be improved further with different constraints on the permutation that appears in its proof. Even more importantly, an attempt should be made to derandomize this bound, since constructive proofs are superior to non-constructive proofs.

Furthermore, Algorithms 4, 5, and 6 could be made faster. There could be various heuristics for choosing a vertex in Algorithm 5 that would speed up the algorithm in the average case. However, there may be a more fundamental improvement in reducing the order of growth of this algorithm's running time. For example, one could require that Algorithm 5 chooses a vertex with threshold at least 2, while extending the base case to include all situations in which every vertex in A has threshold 1. If there is an efficient algorithm for this base case, then the order running time of Algorithm 5 would be reduced to $O(2^{\epsilon s})$ where $0 < \epsilon < 1$ and s is the sum of the thresholds in A.

Future work on Booper includes the development of new features that make Booper easier to use. For instance, some users would find it helpful to be able to create and edit a graph using features in Booper, rather than writing a file defining the graph. There are two other major features that should be added. The first is to implement directed graph functionality and the second is to allow the threshold to be defined by a function instead of a constant. Both of these features would require a major overhaul of existing code, but they would make Booper useful in a much wider variety of scenarios involving research on bootstrap percolation.

Bibliography

[1] J. Chalupa, P. L. Leath, and G. R. Reich, "Bootstrap percolation on a bethe lattice," *Journal of Physics C: Solid State Physics*, vol. 12, no. 1, pp. L31–L35, 1979. [Online]. Available: https://doi.org/10.1088/0022-3719/12/1/008

[2] P. Shakarian and D. Paulo, "Large social networks can be targeted for viral marketing with small seed sets," 2013.

[3] R. Bhansali and L. P. Schaposnik, "A trust model for spreading gossip in social networks," 2019.

[4] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 137–146. [Online]. Available: https://doi.org/10.1145/956750.956769

[5] N. Chen, "On the approximability of influence in social networks," *SIAM Journal on Discrete Mathematics*, vol. 23, no. 3, pp. 1400–1415, 2009. [Online]. Available: https://doi.org/10.1137/08073617X

[6] R. Wilson, *Introduction to Graph Theory*. Longman, 1996.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Intorduction to Algorithms*, 3rd ed. Cambridge, Massachusetts: The MIT Press, 2009.

[8] E. W. Weisstein, "Triangular grid graph," https://mathworld.wolfram.com/TriangularGridGraph.html.

[9] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158. [Online]. Available: https://doi.org/10.1145/800157.805047

[10] ——, "The P versus NP problem," 2000.

[11] W. Gasarch, "The P=?NP poll," *SIGACT News*, vol. 33, pp. 34–47, 01 2002.

[12] E. Ackerman, O. Ben-Zwi, and G. Wolfovitz, "Combinatorial model and bounds for target set selection," *Theoretical Computer Science*, vol. 411, no. 44, pp. 4017–4022, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397510004561

[13] D. Reichman, "New bounds for contagious sets," *Discrete Mathematics*, vol. 312, no. 10, pp. 1812–1814, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0012365X12000301

[14] M. R. Garey and D. S. Johnson, *Computers and intractability.* W. H. Freeman, 1979.

[15] H. S. Wilf, *Algorithms and Complexity*, 1994. [Online]. Available: https://www2.math.upenn.edu/~wilf/AlgoComp.pdf

[16] Y. Caro, "New results on the independence number," Tel-Aviv University, Tech. Rep., 1979.

[17] V. K. Wei, "A lower bound on the stability number of a simple graph," Bell Laboratories, Tech. Rep., 1981.

[18] J. R. Griggs, "Lower bounds on the independence number in terms of the degrees," *Journal of Combinatorial Theory, Series B*, vol. 34, no. 1, pp. 22–39, 1983. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0095895683900035

[19] N. Alon, J. Kahn, and P. D. Seymour, "Large induced degenerate subgraphs," *Graphs and Combinatorics*, vol. 3, pp. 203–211, 1987.

[20] P. Erdős, "Graph Theory and Probability," *Canadian Journal of Mathematics*, vol. 11, p. 34–38, 1959.

[21] ——, "Graph Theory and Probability. II," *Canadian Journal of Mathematics*, vol. 13, p. 346–352, 1961.

[22] P. Erdős and A. Rényi, "On random graphs. I," *Publ. Math.*, vol. 6, pp. 290–297, 1959.

[23] P. W. Holland, K. B. Laskey, and S. Leinhardt, "Stochastic blockmodels: First steps," *Social Networks*, vol. 5, no. 2, pp. 109–137, 1983. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0378873383900217

[24] GitHub, "The 2020 state of the octoverse," 2020. [Online]. Available: https://octoverse.github.com

[25] E. Alsadoon, "Motivating factors for faculty to use web applications in education," *TOJET : The Turkish Online Journal of Educational Technology*, vol. 17, no. 3, 2018, copyright - Copyright Sakarya University 2018; Last updated - 2019-11-23.

[26] StackOverflow, "2020 developer survey," 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020

[27] Material-UI, "About us," 2020. [Online]. Available: https://material-ui.com/company/about/

[28] E. Mustafaraj, "D3.js - an introduction," 2020. [Online]. Available: https://cs.wellesley.edu/~mashups/pages/am5d3p1.html

[29] A. Pandey, "D3 graph theory," 2018. [Online]. Available: https://d3gt.com/index.html

[30] J. Balogh and G. Pete, "Random disease on the square grid," *Random Structures & Algorithms*, vol. 13, 1998.

[31] R. Zhao, "Extremal problems about bootstrap percolation on graphs," 2020. [Online]. Available: https://mspace.lib.umanitoba.ca/bitstream/handle/1993/35278/zhao_ruoxin.pdf

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

[33] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 539–547.

[34] B. Rozemberczki, R. Davies, R. Sarkar, and C. Sutton, "Gemsec: Graph embedding with self clustering," in *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2019*. ACM, 2019, pp. 65–72.

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

[36] L. Rensis, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 140, p. 1–55, 1932.

Appendix A

Booper User Study Survey

Hi, we're glad to have you using our application! We developed Booper to help people study bootstrap percolation, a mathematical model that we've been studying. Bootstrap percolation is a visually intuitive process that can model the spread of disease in a population. If this model is unfamiliar to you, that's completely fine! Our goal with this study is to see how users of all levels of familiarity with the concept engage with our application. If you want to know more about bootstrap percolation, you can read our brief explanation.

Prior to responding to our survey questions, we ask that you explore Booper and spend at least 5 minutes using the Study page. All the instructions for how to use this tool will be given to you in a tutorial that pops up as soon as you open the Study page. One feature allows you to upload a file defining a graph, and it draws this graph on the screen. If you want to try this feature, you can use one of the files we provide below, or you can write one yourself following the format defined in the feature tooltip.

Booper is available here. We recommend viewing it fullscreen on a laptop or desktop computer. It's a very graphical application, so it is difficult to use in small windows/screens.

Responses to this survey may be anonymous. Respondents must be at least 18 years old and will have a chance to enter a raffle for one of six $25 Amazon gift cards.

1. Are you at least 18 years old?
    - Yes
    - No (if a participant responds no, the survey ends)
2. Please check the box below before proceeding.
3. How well do you understand the functionality of the existing features?
    - Extremely well
    - Very well
    - Moderately well
    - Slightly well

- Not well at all

4. Are there any features that you feel are missing from Booper?
5. How helpful did you find the initial tutorial?
    - I found it very helpful
    - I found it somewhat helpful
    - I did not find it helpful or confusing
    - I found it somewhat confusing
    - I found it very confusing
    - I skipped the initial tutorial
6. How complete did you find the initial tutorial?
    - It was way too long and detailed
    - It was a little long and detailed
    - It had the right amount of detail
    - It needs a little more detail
    - It needs a lot more detail
    - I skipped the initial tutorial
7. Do you have any additional feedback about the initial tutorial?
8. How much do you like the layout of Booper?
    - I like it a great deal
    - I like it somewhat
    - I neither like nor dislike it
    - I dislike it somewhat
    - I dislike it a great deal
9. Do you have any additional feedback about the layout?
10. How much do you like the light color scheme?
    - I like it a great deal
    - I like it somewhat
    - I neither like nor dislike it
    - I dislike it somewhat

- I dislike it a great deal
- I didn't try light mode

11. How much do you like the dark color scheme?

    - I like it a great deal
    - I like it somewhat
    - I neither like nor dislike it
    - I dislike it somewhat
    - I dislike it a great deal
    - I didn't try dark mode

12. Do you have any additional feedback about the color scheme(s)?

13. How familiar were you with graph theory before you started participating in this study?

    - I study graph theory
    - I knew a few theorems
    - I knew what a graph is
    - I had heard of it
    - I had never heard of it

14. How familiar were you with bootstrap percolation before you started participating in this study?

    - I study bootstrap percolation
    - I knew a few theorems
    - I knew what bootstrap percolation is
    - I had heard of it
    - I had never heard of it

15. What is your occupation?

    - Undergraduate student
    - Graduate student
    - University faculty
    - Software engineer
    - Other

16. If you answered student or faculty to the previous question, what do you study / in which department are you?

    - Computer science
    - Mathematics/mathematical sciences
    - Other

17. If you would like to be entered into a raffle for one of six $25 Amazon gift cards, please enter your email below, and we will contact you if you win. Your email will only be used for the raffle and will not be used in our project. When the raffle is finished, we will remove email addresses from our data sets.