Graph burning and cooling

An interactive, probabilistic and optimization approach

Merel Susanna

Delft University of Technology

TU Delft

Graph burning and cooling

An interactive, probabilistic and optimization approach

by

Merel Susanna

to obtain the degree of Master of Science
in Applied Mathematics specialising in Discrete Mathematics and Optimisation at the faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), at Delft University of Technology,
to be defended on Wednesday, October 22, 2025 at 10:30

Student number:	4907299
Thesis committee:	Dr. Y. Murakami (Supervisor)
	Dr. N.J. Verhulst (Supervisor)
	Dr. ir. L.J.J. van Iersel (Supervisor)
	Dr. J. Komjáthy
Project Duration:	February, 2025 - October, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science

TU Delft

Preface

This thesis, *Graph burning and cooling: an interactive, probabilistic and optimization approach*, was written as part of my graduation from the Master's program Applied Mathematics with the specialization Discrete Mathematics and Optimization at TU Delft. From February 2025 until October 2025, I worked intensively on the subject of graph burning and cooling and composed this thesis.

The research presented in this thesis combines both theoretical and algorithmic results. I very much enjoyed working on this combination, as it allowed me to apply abstract mathematical ideas while also implementing and testing them in practice. The interactive and probabilistic approaches described in this work reflect my interest in programming and algorithmic thinking, while the ILP part represents a more theoretical contribution to the field. Working on both aspects gave me a broad perspective on the topic and taught me valuable skills for future research and work. I would like to acknowledge the use of OpenAI to improve the clarity and grammar of this thesis.

I would like to thank my family and friends for their continuous support, encouragement, and understanding during the writing of this thesis. Their help and patience have been invaluable in helping me complete this project and overcome the more difficult stages.

In addition, I would like to sincerely thank the members of my thesis committee: Yuki Murakami, Nikolaas Verhulst, Leo van Iersel, and Julia Komjáthy. In particular, I am grateful to Yuki and Nikolaas for all the weekly meetings and detailed feedback. Having two weekly supervisors was a great opportunity. I learned a lot from your ways of thinking, your structured approach to hard problems, and your fast pace of reasoning. These experiences have contributed greatly to quality of my work.

Enjoy reading!

Merel Susanna
Delft, October 2025

Abstract

Networks appear in many important areas such as transportation systems, social interactions, and computer infrastructures. Understanding how processes spread across such systems is essential for predicting and controlling events like the transmission of diseases, the diffusion of information, or the spread of computer viruses.

This thesis studies the burning number, a measure that indicates the speed at which a process spreads over a network. While most existing research assumes a standard burning process in which every neighbor of a burning node becomes burned with probability 1, this work extends the model to a different setting. We develop an algorithm that incorporates a probability of spreading less than 1, allowing us to compute the expected number of rounds needed to burn an entire graph for any given sequence of nodes. This approach provides more realistic insights into spreading processes and can be applied to a variety of real-world problems.

In addition to the theoretical contribution, we also design and implement an interactive Python tool that visualizes and simulates the burning process for any user-specified graph. This tool makes it possible to experiment with different graphs and contributes to better understanding, teaching, and exploration of the burning number problem.

Finally, this thesis also addresses the related concept the cooling number. For this problem we introduce an integer linear program (ILP) formulation and propose a 2-approximation algorithm to compute a lower bound of the cooling number quickly. We stated a conjecture about 3-legged spider graphs. This conjecture claims that there exists an optimal cooling sequence such that the source nodes are clustered per leg. As a first step towards the proof of the conjecture, we showed that for a general spider, all the sources of one leg can be clustered at the start of the cooling sequence.

Contents

1

Introduction

Networks are everywhere to be found: transportation networks, social networks, and computer networks. All these networks can be visualized using nodes and edges. The nodes represent train stops, people, or computers, and the edges represent rails, friendships, or data connections. Networks are not only a way to visualize connections but can also be used for analyses or simulations. They help us understand the structure of complex systems and predict how changes in one part of a network can affect the rest.

An example of the usefulness of analyzing a network is the study of the spreading of infectious diseases. As written by L. Danon et al., network theory has great potential in epidemiology [5]. It describes measurable properties of networks that impact disease transmission, such as components, degrees, distributions, correlations, distances, clustering, and subgraphs. Using these properties, researchers can predict which nodes are more likely to be affected, how fast a disease might spread, and which interventions could be most effective.

Another way to study spreading processes is through graph-theoretic measures that capture how quickly something can propagate through a network. One such measure is the burning number. The burning number of a network indicates how fast a contagion, information, or influence can reach all nodes if the spread starts from carefully chosen sources in successive rounds. Although this concept is not widely used in epidemiology yet, it provides an interesting perspective on the speed of spreading processes.

Graph burning is a discrete-time process that models the spread of something, such as fire, information, or disease, across the nodes of a graph. The nodes can represent people, cities, or computers. Throughout this thesis, we use the term burning for consistency. However, it is important to note that in real-world applications, different terminology might be more appropriate depending on the context. For example, when simulating the spread of an infectious disease among people, it may be more suitable to use the term infection rather than burning. In that case, the nodes of the graph represent individuals, and an edge between two nodes indicates that the corresponding individuals have regular contact.

The burning number process is the process of starting with a graph G with all nodes unburned. In the first round, $r = 1$, a node is chosen as a burning source and is immediately burned. In each round $r > 1$, all neighbors of a burning node are burned as well. Additionally, every round a new unburned node is chosen as a burning source. The burning process ends when all the nodes of G are burned. The goal is to pick the sources in such a way that the graph is completely burned in as few rounds as possible.

The concept of the burning number can be applied to various other real-world networks. For example, in transportation networks, the burning number could represent the number of time steps needed for a delay at one station to reach all other stations. In social networks it can indicate how fast information or rumors spread from a few initial users to the entire network. In computer networks, it can describe how quickly a virus or malware might infect all devices. Studying the burning number helps to understand and possibly control spreading processes in many types of networks.

Besides the burning number, there is also the concept of cooling number. This process works exactly the same, however the goal is to pick the sources in such a way that the graph is completely burned as slowly as possible. Instead of saying burning, we use cooling in this context.

1.1. Previous work

It is useful to know the burning number of certain types of graphs, for example a path, a spider, or a cycle. As proven by Bonato et al., finding the burning number of a graph is NP-hard [3]. Bonato et al. also provide properties, characterizations, and bounds for several graph classes [15]. In this paper they work with the concept of covering. A covering of a graph is a set of subsets of the nodes of the graph whose union is the complete set of nodes of the graph. From this concept, a number of upper bounds can be stated.

For some graphs the burning number can easily be found. For a graph P_n, a path on n nodes, and for a cycle graph, the burning number is proven to be

$$b(P_n) = b(C_n) = \lceil n^{1/2} \rceil.$$

For other types of graphs, the exact burning number, or even a sharp bound, is not always clear. For some of these graph classes there are only proven ranges in which the burning number lies.

Lastly, there is a burning number conjecture which states that for any connected graph G of order n,

$$b(G) \leq \lceil n^{1/2} \rceil$$

[15]. Murakami [11] proves that this conjecture is true for trees without degree-2 vertices. Additionally, Ning et al. prove that the burning number conjecture also holds for trees of order n with at most $\lfloor (n-1)^{1/2} \rfloor$ degree-2 vertices [12].

Besides all the theoretical results, there are also integer linear programs (ILPs) and approximation algorithms developed to numerically solve the burning number problem. García-Díaz et al. introduce several ILPs and approximation algorithms. The best algorithm found is proven to be a $\frac{1}{2}$-approximation [7].

Besides the burning number, also the cooling number is studied. However, there is less known about the cooling number than about the burning number. Let G be a graph on n nodes, Bonato et al. proved that

$$CL(G) \leq \lceil \frac{n+1}{2} \rceil$$

and

$$\lceil \frac{\mathsf{diam}(G)+2}{2} \rceil \leq CL(G) \leq \mathsf{diam}(G)+1$$

, where $\mathsf{diam}(G)$ represents the diameter of graph G. The diameter of G is the largest distance between any two vertices in G. These two equations combined show that for a path of order n, P_n, $CL(P_n) = \lceil \frac{n+1}{2} \rceil = \frac{\mathsf{diam}(G)+2}{2}$ [4].

1.2. Information gap

In all the research done so far, the standard burning process is considered. By the standard burning process we mean that in every round all the neighbors of a burning node are also burned with probability 1. In the first part of this thesis, Chapter 3, we introduce a probability of spreading. This results in a more realistic scenario. For example, in the case of disease spreading among a group of people, it is not very realistic that if a person is infected with some disease then all the people related to this person are also infected. One person might have a better immune system than another. In this thesis an algorithm is designed that calculates the expected number of rounds needed to burn an entire graph, given a probability of spreading and a given sequence of nodes which represents the order of the sources. The assumed probability is the same for each edge and the sequence of nodes is a permutation of all nodes of the graph.

Despite the fact that there is still a lot of research to do about the burning number, there is currently no clear, easy, and useful tool for gaining knowledge and intuition about the subject except by drawing many examples by hand or experimenting in a programming language. In Chapter 4 a new, easy-to-use and interactive tool is described and shown. This tool makes it possible to enter any graph and simulate the burning process by selecting sources, simulating the spread, and using supporting tools. This is of great benefit for education and research purposes.

Lastly, in Chapter 5, the topic of the cooling number is addressed. As mentioned in Section 1.1, there already exists an ILP program to find the burning number of a graph. Chapter 5 presents an ILP for the cooling number problem. Additionally, an approximation algorithm is explained that finds a 2-approximation of the cooling number that runs in $\mathcal{O}(nm + n^2)$ time. Finally, we conjecture that the cooling sequence of a spider graphs with three legs can be sorted such that the sources are clustered per leg. Despite the fact that we did not found a proof, we did prove that all the sources on one leg can be clustered at the start of the cooling sequence and this does hold for every spider.

2

Preliminaries and the burning process

2.1. Graph theory

A graph is a mathematical structure that is used to describe relationships between objects. It consists of a set of vertices (also called nodes), written as V, and a set of edges, written as E. In this thesis, we only work with undirected edges, so each edge is an unordered pair of two vertices. So (u, v) and (v, u) represent the same edge. A graph G is usually written as $G = (V, E)$.

The word network is often used as a synonym for graph, especially in real-world problems. Some examples are:

- social networks, where vertices represent people and edges represent friendships,
- transportation networks, where vertices represent cities and edges represent roads,
- computer networks, where vertices represent devices and edges represent data connections.

In this thesis, we study *simple graphs*. A simple graph has no edges from a vertex to itself (called loops), and between two different vertices there can be at most one edge.

A few more important concepts in graph theory are:

- *Path*: a sequence of vertices $(v_0, ..., v_n)$ where the only edges are between each consecutive pair and all vertices are distinct.
- *Distance*: the number of edges of a shortest path between two vertices. This is written as $d(u, v)$ for vertices u and v.
- *Diameter*: the largest distance between any two vertices in the graph.
- *Neighborhood*: the set of vertices that are directly connected to a given vertex. The neighborhood of v is written as $N(v) = \{u \in V : (u, v) \in E\}$. $N_i[v]$ notates the (i)-closed neighborhood of node v, that is all the nodes whose shortest-path distance from v is at most i.

These basic ideas are needed to understand the burning process, since burning spreads along paths, depends on distances between vertices, and always moves through neighborhoods.

2.2. The burning process

Formally, let $G = (V, E)$ be an undirected graph. A node has two states, burned or unburned. Once a node is burned, it stays in this state for the rest of the process. Initially, all nodes are in an unburned state. At each round, which is a discrete time step, two actions take place:

1. Spreading Phase: All nodes that were already burning in the previous time step will spread the fire to all their neighbors. That is, any node adjacent to a burning node becomes burned as well.
2. Source Selection Phase: After the spreading step, a new burning source is selected from the set of remaining unburned nodes. This new source becomes a burned vertex and begins to spread in the following round.

The process then proceeds to the next round and the two actions are repeated. This continues until all nodes in the graph are burned [15].

The main objective in the burning process is to determine the burning number of the graph: the minimum number of rounds required to completely burn the graph when selecting one new source per round. A burning sequence is a sequence $(s_1, s_2, \ldots, s_k)$ where $s_i \in V$. The burning number $b(G)$ of a graph G is the smallest k such that there exists a burning sequence of length k that burns all the vertices of G. Intuitively, a burning source placed at round t will eventually cover all nodes within the distance $(k - t)$, where k is the total number of rounds.

In this thesis, we introduced a slightly different approach to the source selection phase. Instead of demanding an unburned node chosen as a source in each round, we allow nodes that are already burned to be chosen as a source. Since choosing a burned node as the source will not have any effect, we will denote this as a dash, $-$, in the burning sequence. So, for example, instead of a burning sequence $(1, 5, 7, 5, 9)$, we will denote this as $(1, 5, 7, -, 9)$. More formally, we define the burning sequence as a sequence $(s_1, s_2, ..., s_k)$ where $s_i \in V \cup \{-\}$.

It is important to note that in some cases, during the final round of the process, there may no longer be a possibility to choose a new burning source. This can happen when the spreading phase of that round results in all remaining unburned nodes becoming burned. Nevertheless, this final round is still essential as the burning process only completes at the end of it. To account for such situations, the dash is again used to indicate that no new source was selected in that round, even though the round itself contributed to the burning process.

The graph burning process captures important dynamics of spreading phenomena and serves as a useful abstraction for many practical applications. An example is fast spreading an alarm or information via satellite throughout a network. The satellite can only spread the information sequentially (source picking), but it also spreads the information via a cable connection to its neighbors (spreading) [17].

2.2.1. Example

Consider the path graph P_5 on five nodes. Figure 2.1 shows the burning process of this path graph P_5. Initially, at $t = 0$, all nodes are unburned (represented by the grey nodes). At time $t = 1$, node 3 is chosen as a burning source. Next, at time $t = 2$, nodes 2 and 4 are burned by the spreading of burned node 3 and node 1 is chosen as the new burning source. Lastly, node 5 is burned by spreading in round 3. This results in burning sequence $(3, 1, -)$ since in the last round, no new source is chosen. In this case, the burning number of P_5 is $b(G) = 3$. Therefore, $(3, 1, -)$ is an optimal burning sequence. Optimal burning sequences do not have to be unique. For example, $(3, 5, -)$ is an optimal burning sequence as well. To see that 3 is the burning number of the graph is optimal, we should argue that 1 and 2 are not possible

options. Clearly, $b(G) \neq 1$ as that would mean that by only picking one node as a source, the complete graph is burned. This is only the case for a graph with only one node. The maximum number of nodes that could be burned per round due to spreading is two times the number of burned nodes for a path graph. At the start of round 2, there is only one node burning. At the end of the spreading phase maximal three nodes could be burned in total and therefore after the complete round, the maximum number of nodes that could be burned is four. Therefore, $b(P_5) = 3$.

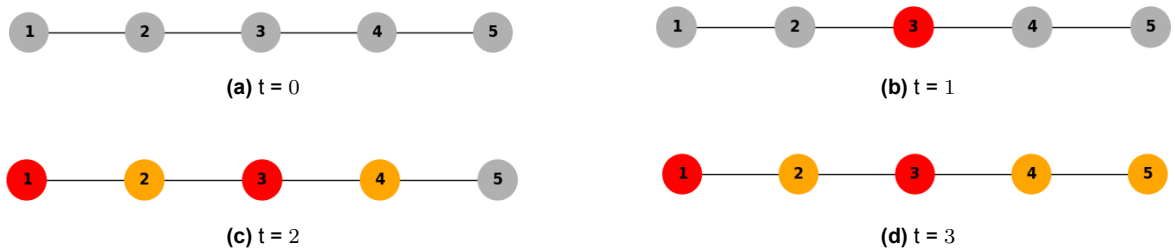


Figure 2.1: We show the burning process on the path graph P_5. The red nodes indicate the burning sources and the orange nodes are the nodes that are burned in the spreading phase. The burning sequence corresponding to this example is $(3, 1, -)$.

2.3. The burning number problem

The burning number problem consists of the following question: what is the minimum number of rounds needed to burn the entire graph when the burning spreads according to the burning process? More technically, given a graph G, what is the burning number $b(G)$? Equivalently, what is the minimum length of the burning sequence to burn the entire graph?

At first sight, the problem may appear simple: one might expect that it is always best to place new burning sources in the part of the graph where there are a lot of unburned nodes not connected to a burned node. However, finding an optimal sequence is far from trivial, especially in larger or more complex graphs. In fact, the burning number problem is known to be computationally challenging. As proven by Bonato et al., the burning number is NP-hard [3].

Because of this difficulty, researchers are interested not only in finding the exact burning number for specific classes of graphs, but also in designing approximation algorithms and proving bounds that apply more generally. One of the conjectures that is not proven yet is the claim that for every connected graph G of order n, $b(G) \leq \lceil n^{\frac{1}{2}} \rceil$. Bastide et al. provides the best-known bound on a connected graph of order n, which is $b(G) \leq \sqrt{\frac{4}{3}n} + 1$ [1].

2.4. Integer linear program

As the burning number problem is an optimization problem, an Integer Linear Program, hereafter mentioned as ILP, can be used to solve this problem. An ILP consists of an objective function and constraints for variables. As the burning number of a graph is the minimum number of rounds needed to burn the entire graph, the burning number problem can be defined as a minimization problem. Garcia-Diaz et al. present several ILP's for solving the graph burning problem [7]. For one of the ILP's the burning number problem is formulated as a coverage problem. This ILP is formulated by equations (2.1) - (2.7). Binary variables $x_{i,j}$ and b_j are used. $x_{i,j} = 1$ when node j is chosen as a source at time i and zero otherwise and $b_j = 1$ if node j is burned and zero otherwise. The objective function minimizes the number of nodes chosen as a burning source after the time steps reach the upper bound. Equation (2.2), (2.3) and (2.7) imply that every time step one source is chosen. Constraint (2.4) handles the spreading.

Lastly, equation (2.5) forces all nodes to be burned at the end [7].

It is important to note that the variable U represents an upper bound on the burning number of the graph. A trivial upper bound for U is the total number of nodes in the graph, however this could be improved by theoretical known upper bounds.

$$\text{min.} \quad \sum_{i\in[1,U]} \sum_{v_j\in V} x_{i,j} \tag{2.1}$$

$$\text{s.t.} \quad \sum_{v_j\in V} x_{i,j} \leq \sum_{v_j\in V} x_{i-1,j} \qquad \forall i \in [1,U] \tag{2.2}$$

$$\sum_{v_j\in V} x_{i,j} \leq 1 \qquad \forall i \in [1,U] \tag{2.3}$$

$$b_j \leq \sum_{i\in[1,U]} \sum_{v_k\in N_{i-1}[v_j]} x_{i,k} \qquad \forall v_j \in V \tag{2.4}$$

$$\sum_{v_j\in V} b_j = n \tag{2.5}$$

$$\text{where} \quad x_{i,j}, \quad b_j \in \{0,1\} \qquad \forall i \in [1,U], \forall v_j \in V \tag{2.6}$$

$$\sum_{v_j\in V} x_{0,j} = 1 \tag{2.7}$$

In some cases it would be beneficial to minimize the number of sources. For example in the case of information spreading across a network of servers. Then it would be better if some information should be installed on as few different servers as possible. This minimization corresponds to minimizing the amount of different sources, while still maintaining the minimum number of rounds needed. Therefore, an integer linear program (ILP) introduced which minimizes the number of rounds needed and at the same time minimizing the number of sources. The ILP can be adjusted such that simultaneously the number of rounds and the number of sources are minimized. Such an ILP is given by equations (2.8) - (2.16). Gurobi can have multiple objective functions. These objective functions can be combined using weights or in a hierarchical way [8]. Among the optimal burning sequences that result in the burning number, the one with the least number of distinct nodes should be chosen. Therefore, (2.8) is the primary objective function and (2.9) is the secondary objective function in the hierarchical way of solving this ILP. c_j is a newly introduced binary variable that equals 1 if a node v_j appears in the burning sequence. Constraint (2.14) forces c_j to be 1 when node v_j is in the burning sequence for some timestep i. The secondary objective function (2.9) minimizes the number of distinct nodes appearing in the burning sequence.

$$\text{min.} \quad \sum_{i\in[1,U]} \sum_{v_j\in V} x_{i,j} \tag{2.8}$$

$$\text{min.} \quad \sum_{v_j\in V} c_j \tag{2.9}$$

$$\text{s.t.} \quad \sum_{v_j\in V} x_{i,j} \leq \sum_{v_j\in V} x_{i-1,j} \qquad \forall i \in [1,U] \tag{2.10}$$

$$\sum_{v_j\in V} x_{i,j} \leq 1 \qquad \forall i \in [1,U] \tag{2.11}$$

$$b_j \leq \sum_{i\in[1,U]} \sum_{v_k\in N_{i-1}[v_j]} x_{i,k} \qquad \forall v_j \in V \tag{2.12}$$

$$\sum_{v_j\in V} b_j = n \tag{2.13}$$

$$c_j \geq x_{i,j} \qquad \forall i \in [1,U], \forall v_j \in V \tag{2.14}$$

$$\text{where} \quad x_{i,j}, \quad b_j, \quad c_j \in \{0,1\} \qquad \forall i \in [1,U], \forall v_j \in V \tag{2.15}$$

$$\sum_{v_j\in V} x_{0,j} = 1 \tag{2.16}$$

3

The expected number of rounds to burn a graph

In the classical version of the burning problem, a burning node always spreads the fire to all of its neighbors in the next round. In other words, the fire spreads with probability 1. However, this assumption may not reflect certain real-world situations. For example, in the case of an infectious disease, the chance of transmission can vary depending on the disease and the immune systems of different people. Some infections are highly contagious, while others spread more slowly or are less reliable.

To better model such processes, this chapter introduces a probabilistic variation of the burning number problem. In this probabilistic approach, the fire spreads from a burning node to each of its neighbors with a fixed probability p. Specifically, each edge ab in the graph is assigned the same probability p, which represents the chance that node b will catch fire from node a, assuming that a is already burned. The burning along different edges occurs independently. That is, for each burning node at each time step, the unburned neighbors become burned with probability p, independently of other neighbors and independently across time steps.

The goal is no longer to compute the exact number of rounds required to burn the graph, but instead to determine the expected number of rounds until all nodes are burned, given a fixed spreading probability p. Besides the probability, also a sequence with a permutation of the graph nodes is needed which tells the order in which the nodes should be chosen as a source.

In this chapter, we present an algorithm that estimates the expected number of rounds needed to burn the entire graph under this probabilistic spreading approach.

3.1. Method

In this section, we explain the general idea behind the algorithm that incorporates probabilities into the burning process. The goal of this algorithm is to compute the expected number of rounds required to burn the entire graph when a sequence is given which represents the order in which nodes are chosen as a source. In other words, instead of simply simulating a deterministic spreading process, the algorithm takes into account the uncertainty that arises when spreading does not always succeed with probability 1.

In the traditional burning problem, the spreading behavior is fully deterministic. We define a *state* as the set of nodes that are currently burned at a given time step. If a node i is burned at time j, then all of its neighbors will certainly be burned at time $j + 1$. At the same time, a new burning source is selected. This means that the transitions from one state to the next are completely fixed: given the current state, the next state is uniquely determined. Thus, in the classical model, the system always moves forward in a predictable way, and the process of moving between states can be described with probability 1.

When a probability of burning is introduced, however, the situation becomes more complex. From any given state, multiple possible outcomes can occur. For example, even if a node has several burning neighbors, it might fail to catch fire in the next round with some probability. As a result, the process no longer has a single deterministic state but instead branches into several possible future states, each with its own probability. To handle this uncertainty in a systematic way, the algorithm makes use of the concept Markov chains.

A *Markov chain* is a stochastic process that evolves in discrete time steps. The defining property of a Markov chain is that the probability of moving to the next state depends only on the current state, and not on the history of previous states [14]. In the burning problem with probabilities, two different types of transitions must be considered. The first type corresponds to the spreading phase, where currently burning nodes may or may not spread the fire to their neighbors depending on the probability p. The second type corresponds to the source-picking phase, where a new burning source is chosen. The probabilities of moving between states due to spreading are recorded in a matrix known as the transition matrix. The rows and columns of this matrix correspond to the possible states of the system, and an entry $T_{i,j}$ represents the probability of transitioning from state i to state j. Each row of this matrix sums to 1, reflecting the fact that from any state the system must move to some next state. The source picking is done in a matrix called the addition matrix. This matrix only consists of zeros and ones. Each column contains two ones and the rest are zeros. There are only two ones since there are two ways to end up in a state, say $\{x_1, x_2, ..., x_n\}$. Assume node a is chosen as a source. Either a is already in the starting state, so nothing changes or the starting state is $\{x_1, x_2, ..., x_n\} \setminus a$.

The algorithm then alternates between the transition and the addition matrices as the process unfolds over time. By combining the effects of probabilistic spreading and source selection, it is able to compute the expected number of rounds required to burn the entire graph using a burning sequence. This approach allows us to generalize the classical burning model and provides a way to analyze more realistic scenarios where the spread is not guaranteed, such as the spread of information, diseases, or influence in real-world networks.

3.2. An algorithm for the expected number of rounds needed to burn a graph

To improve readability and modularity, the algorithm is divided into several smaller functions. Each function is responsible for a specific step in the overall process. This modular structure not only makes the algorithm easier to understand, but it also makes it possible to test or optimize individual components separately. The complete process can then be seen as a sequence of well-defined building blocks.

The algorithm starts by determining, for any given state of the system, the other states that can be reached after one round of the spreading phase. Recall that a state is defined as the set of nodes that are burned. For each such set of nodes, the algorithm explores all possible outcomes that may arise when the fire spreads to their neighbors with probability p. For example, if a node a has three neighbors $\{b, c, d\}$, each of them may independently

catch fire or remain unburned, which leads to several possible successor states. In this case the states are: $\{\{a\},\{a,b\},\{a,c\},\{a,d\},\{a,b,c\},\{a,b,d\},\{a,c,d\},\{a,b,c,d\}\}$. These states, along with their associated probabilities, are collected and stored for later use.

Once the possible successor states have been identified, the algorithm calculates the probability of transitioning from the current state to each of them. This is the most important part of the probabilistic model: unlike in the classical burning problem, where the spreading step is deterministic, here every transition has an associated probability that depends on the chosen value of p. These probabilities are stored in a data structure that records, for each pair of states (a,b), the probability of moving from state a to state b. The transition probabilities are calculated in algorithm 1. The final probabilities are calculated with the formula (3.1)

$$\prod_{i\in\text{added}} 1-(1-p)^{|N_b(i)|} \cdot \prod_{j\in\text{unburned}} (1-p)^{|N_b(j)|} \tag{3.1}$$

This formula is the probability that the nodes that are added to the state are burned via some burning neighbor times the probability that the nodes that should not be burned stay unburned. Algorithm 1 shows the steps needed for programming to perform the explanation above.

With this information, the algorithm constructs two different transition matrices. The first is the transition matrix, which describes how the fire spreads from one state to another according to the given probabilities. Each entry in this matrix represents the probability of moving between two states during the spreading phase. The second matrix is the addition matrix, which captures the effect of adding a new burning source according to the chosen burning sequence. Unlike spreading, this step is deterministic: if a new source is added to a state, the system moves to the corresponding expanded state with probability 1. The steps to calculate these matrices are given in Algorithms 2 and 3.

The main part of the algorithm then alternates between these two matrices. In each round except for the first round, the transition matrix is applied first, followed by the addition matrix, which changes every round. This alternation simulates one complete time step in the burning process. By repeating this procedure across multiple rounds, the algorithm terminates when all nodes are burned. The first round is different. In this round a row vector is initialized with all zeros except for the state only containing the first node of the given sequence.

Finally, the algorithm computes the expected number of rounds needed for the process to finish. To do this, it tracks how the probability distribution evolves over across the states. In particular, it monitors the probability that the system reaches the “fully burned” state at each round. These probabilities are then combined to calculate the expected total number of rounds. The outcome is a precise measure of how long the burning process is expected to take under the given sequence and probability. This last part is given by Algorithm 4.

This approach provides a probabilistic extension of the burning number problem. By separating the spreading and source selection steps into different matrices, the algorithm is flexible. It allows us to evaluate different burning sequences and probabilities in a systematic way. The complete process implemented in python is given in Appendix A.

Algorithm 1 TransitionProbabilities

Require: Graph $G = (V, E)$, probability p
Ensure: Mapping $(\text{start\_state}, \text{end\_state}) \mapsto$ probability: evolvedProb, and power set of V: allStates
  $n \leftarrow |V|$
  allStates $\leftarrow$ power set of V
  evolvedStates $\leftarrow$ empty mapping
  for each subset $\in$ allStates **do**
    evolution $\leftarrow$ {subset}
    options $\leftarrow$ all neighbors of nodes in subset, excluding nodes in subset
    subsetOptions $\leftarrow$ power set of options
    for each option $\in$ subsetOptions **do**
      possibleNextState $\leftarrow$ subset $\cup$ option
      Add possibleNextState to evolution
    end for
    Associate subset with evolution in evolvedStates
  end for
  evolvedProb $\leftarrow$ empty mapping
  for each startState and its evolutions in evolvedStates **do**
    for each targetState $\in$ evolutions **do**
      addedNodes $\leftarrow$ targetState $\setminus$ startState
      unburnedNodes $\leftarrow V \setminus$ targetState
      for each node $i \in V$ **do**
        Find $N_b(i)$, the set of neighbors of node i
      end for
      Compute

$$\text{probability} = \Big(\prod_{i \in \text{addedNodes}} [1 - (1-p)^{|N_b(i)|}] \Big) \cdot \Big(\prod_{j \in \text{unburnedNodes}} (1-p)^{|N_b(j)|} \Big)$$

      Associate probability with the pair (targetState, startState) in evolvedProb
    end for
  end for
  return evolvedProb, allStates

Algorithm 2 TransitionMatrix

Require: Transition probabilities from TransitionProbabilities
Ensure: Transition matrix T
  Obtain evolvedProb and allStates from TransitionProbabilities
  Assign an index to each state in allStates
  stateIndex $\leftarrow$ the mapping from each state to its corresponding index
  $T \leftarrow$ a zero matrix of size |allStates| $\times$ |allStates|
  for each pair $(\text{rowState}, \text{colState})$ with probability p in evolvedProb **do**
    rowIndex $\leftarrow$ index of rowState in stateIndex
    colIndex $\leftarrow$ index of colState in stateIndex
    Set $T[\text{rowIndex}, \text{colIndex}] \leftarrow p$
  end for
  return T

Algorithm 3 AdditionMatrix

Require: Transition probabilities from TransitionProbabilities, node a chosen as a source
Ensure: Addition matrix A
  Obtain allStates from TransitionProbabilities
  Assign an index to each state in allStates
  stateIndex $\leftarrow$ the mapping from each state to its corresponding index
  $A \leftarrow$ a zero matrix of size |allStates| $\times$ |allStates|
  for each state with index i in allStates **do**
    newState $\leftarrow$ state $\cup\{a\}$
    $j \leftarrow$ index of newState in stateIndex
    Set $A[i, j] \leftarrow 1$
  end for
  return A

Algorithm 4 ExpectedNumberRounds

Require: Graph $G = (V, E)$, burning sequence seq, probability p
Ensure: Expected number of rounds to burn the entire graph: expectedRounds
  Obtain evolvedProb and allStates from TransitionProbabilities(G, p)
  $T \leftarrow$ TransitionMatrix(evolvedProb)
  burnedNodes $\leftarrow$ {first element of seq}
  Assign an index to each state in allStates
  stateIndex $\leftarrow$ the mapping from each state to its corresponding index
  Initialize initialVector as a zero vector of length |allStates|
  Set the entry corresponding to burnedNodes in initialVector to 1
  Reshape initialVector into a $1 \times$ |allStates| vector
  totalSetIndex $\leftarrow$ index of state V in stateIndex
  expectedRoundsMap $\leftarrow$ an empty mapping
  expectedRoundsMap[1] $\leftarrow$ initialVector[0, totalSetIndex]
  product $\leftarrow$ initialVector
  for $i = 2$ to $|V|$ **do**
    $M \leftarrow$ AdditionMatrix(evolvedProb, seq[$i - 1$])
    product $\leftarrow$ product $\cdot T \cdot M$
    expectedRoundsMap[i] $\leftarrow$ product[0, totalSetIndex]
  end for
  cumulative $\leftarrow 0$
  for each key in expectedRoundsMap **do**
    expectedRoundsMap[key] $\leftarrow$ expectedRoundsMap[key] $-$ cumulative
    cumulative $\leftarrow$ cumulative + expectedRoundsMap[key]
  end for
  expectedRounds $\leftarrow 0$
  for each key, value in expectedRoundsMap **do**
    expectedRounds $\leftarrow$ expectedRounds + key $\cdot$ value
  end for
  return expectedRounds

3.3. Worked-out example of the algorithm

This section gives a worked-out example of how the steps of the approximation algorithm are applied. For this example we use the graph in Figure 3.1. Assume that the probability of spreading equals 0.3. This probability applies to each edge. The sequence we use is $(0, 1, 2, 3)$. Because this is a complete graph, the result does not depend on the sequence. For other graphs this is not the case. For simplicity, some numbers in this example are rounded.

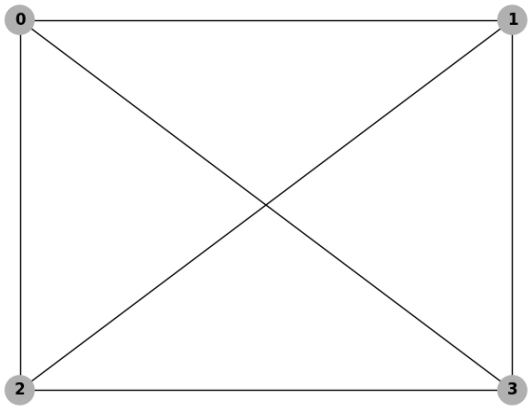


Figure 3.1: Complete graph with 4 nodes, K_4.

First we create the dictionary of possible next states corresponding to each initial state. For this example graph the dictionary looks as follows:

```
evolvedStates = {
 (0,): {({0, 2, 3}), ({0, 1, 2}), ({0, 3}), ({0, 1}),
        ({0, 2}), ({0, 1, 2, 3}), ({0, 1, 3}), ({0})},
 (1,): {({0, 1, 2}), ({1, 2}), ({0, 1}), ({1}),
        ({0, 1, 2, 3}), ({1, 2, 3}), ({0, 1, 3}), ({1, 3})},
 (2,): {({2}), ({2, 3}), ({0, 2, 3}), ({0, 1, 2}),
        ({1, 2}), ({0, 2}), ({0, 1, 2, 3}), ({1, 2, 3})},
 (3,): {({2, 3}), ({0, 2, 3}), ({0, 3}), ({3}),
        ({0, 1, 2, 3}), ({1, 3}), ({1, 2, 3}), ({0, 1, 3})},
 (0, 1): {({0, 1}), ({0, 1, 3}), ({0, 1, 2}), ({0, 1, 2, 3})},
 etc.}
```

As can be seen from this dictionary, if initially nodes 0 and 1 are burned, the possible next burned states are still $\{0, 1\}$ if there is no spread or $\{0, 1, 3\}, \{0, 1, 2\}, \{0, 1, 2, 3\}$ if some spreading occurs.

Next we evaluate all transition probabilities. This results in a dictionary with a tuple as key. The first element of the tuple is the start state and the second element is the state to which we transition. The corresponding value is the transition probability. For the same case as before, where initially nodes 0 and 1 are burned, the transition probabilities to the possible next states are 0.24 (no spread), 0.25, 0.25 and 0.26, respectively. These probabilities depend on the chosen spreading probability (0.3 in this example). For example, if the burned nodes are $\{0, 1\}$ and in the next state the burned nodes are still $\{0, 1\}$, the transition probability equals $(1 - 0.3)^4 = 0.24$.

```
evolvedProb = {
 (({0}), ({0, 2, 3})): 0.063,
 (({0}), ({0, 1, 2})): 0.063,
 (({0}), ({0, 3})): 0.147,
 (({0}), ({0, 1})): 0.147,
 ...
 (({0, 1}), ({0, 1})): 0.2401,
 (({0, 1}), ({0, 1, 3})): 0.2499,
 (({0, 1}), ({0, 1, 2})): 0.2499,
 (({0, 1}), ({0, 1, 2, 3})): 0.2601,
 ...
}
```

After calculating the transition probabilities, the results are transformed into a transition matrix. The order of the rows/columns is $\{0\}, \{0,1\}, \{0,1,2\}, \{0,1,2,3\}, \{0,1,3\}, \{0,2\}, \{0,2,3\}, \{0,3\}, \{1\}, \{1,2\}, \{1,2,3\}, \{1,3\}, \{2\}, \{2,3\}, \{3\}$. The value in the first row and second column corresponds to the transition probability from burned node 0 to burned nodes $(0,1)$.

```
transition_matrix = [
0.343 0.147 0.0630 0.0270 0.0630 0.147 0.0630 0.147 0.0 ...
0.0 0.2401 0.2499 0.2601 0.2499 0.0 ...
...
]
```

This transition matrix is used for the calculation of the probabilities during the spreading phase. In addition, a source selection must be represented as an addition matrix. The addition matrix has the same size as the transition matrix and consists only of zeros and ones. Because the sequence is known from the start, it is clear which node is chosen as a source in each round. This choice is represented in the matrix. Assume for example that node 2 is chosen as a source and A represents the addition matrix. Then $A_{i,j} = 1$ if and only if the set corresponding to column j is equal to the set corresponding to row i together with the set $\{2\}$. If this is not the case, $A_{i,j} = 0$.

Now that all matrices are calculated, the multiplications can be performed to get the final value.

Initially, at $t = 0$, we start with a row vector consisting of all zeros except for the place corresponding to the first node in the sequence, in this case node 0. The initial vector is

$$v = \begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix}.$$

At $t = 1$, we multiply the vector with the transition matrix T and with the addition matrix A:

$$M = v \cdot T \cdot A.$$

This process is repeated for the number of nodes. After each round we evaluate the probability that the graph is fully burned by looking at the value at the index of the complete set of nodes, and this is saved in a dictionary. To get the expected number of rounds, we multiply the number of rounds with the probability of finishing in that round. In the case of sequence $(0,1,2,3)$, the dictionary equals

$$\{1 : 0.0, 2 : 0.090, 3 : 0.68787, 4 : 1\}.$$

These probabilities are cumulative, so they need to be adjusted to calculate the probability of finishing in exactly a given number of rounds:

$$\{1 : 0.0, 2 : 0.090, 3 : 0.59787, 4 : 0.31213\}.$$

From this the expected number of rounds equals

$$1 \cdot 0 + 2 \cdot 0.090 + 3 \cdot 0.59787 + 4 \cdot 0.31213 = 3.2213.$$

4

An interactive graph to simulate the burning process

Understanding the concept of the burning number of a graph can be quite challenging, especially without visual support. While Python and related libraries already allow researchers to generate visualizations of the burning process, such visualizations are often static or require programming knowledge to adapt. The interactive tool developed in this thesis is more user-friendly and easier to use without great programming skills. Users can interact directly with the graph by selecting nodes, entering guesses, and observing the burning process step by step, which provides a clearer understanding of how the burning number evolves. This makes the tool useful for researchers, students and the general public who wish to build intuition about spreading processes in graphs.

For academics and students, the tool offers an engaging and educational way to learn about the burning number. It clearly shows how the process of selecting sources and spreading fire across a graph works, step by step. This makes it much easier to grasp the logic behind the concept. Especially when dealing with large graphs, finding the burning number manually can be very time-consuming. This tool helps researchers by simplifying the process and allowing them to focus more on patterns and insights rather than on calculations.

One of the key features of the tool is the ability to add your own graph. This opens the door for experimentation and may lead to the discovery of new patterns or ideas related to graph burning. By visualizing different graph structures and burning sequences, researchers can explore new hypotheses or improve existing ones.

In addition to its academic value, the tool is also designed to be user-friendly for the public. Because it is so easy to use, it invites citizens to take part in the exploration of the burning number. This kind of citizen science can be very powerful. When more people get involved, more data and insights can be gathered, leading to a better understanding of how certain graphs behave during the burning process [19].

In short, this interactive graph tool has the potential to play an important role in both education and research. It makes a difficult concept more accessible, encourages experimentation, and opens new possibilities for collaboration between researchers and the public.

In this chapter, the features of the interactive graph are explained, visualizations are shown and lastly, the implementation method is explained.

4.1. Features of the interactive graph tool

This section provides an overview of the main features included in the interactive graph tool. Each feature has been designed with usability, educational value, and research support in mind. By describing the functionality in detail, we aim to demonstrate how the tool can help users in understanding the burning number problem and contribute to further exploration in this subject.

The features explained in this chapter cover both the user interface elements and the interactive components that guide the user through the burning process. Visual references are provided to clarify how each part of the tool functions in practice.

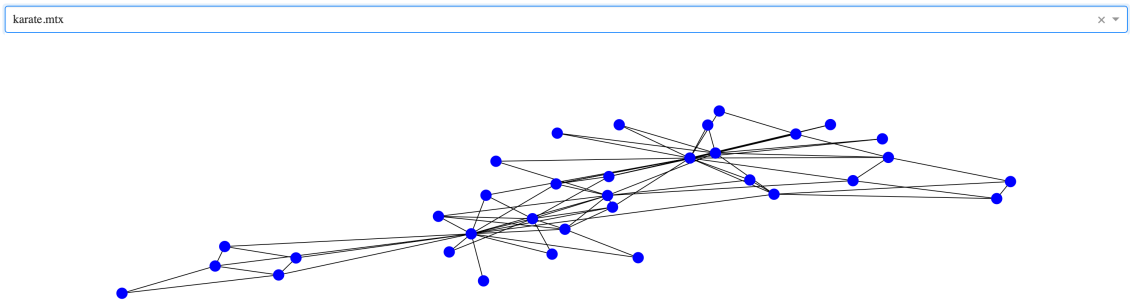
The first thing that pops up when you start the program is a screen where you need to enter your name as shown in Figure 4.1.3. Now the graph burning can begin.

Figure 4.1: At the beginning of the interactive tool you need to enter your name. This is used for the high-scores, which is explained in section 4.1.3

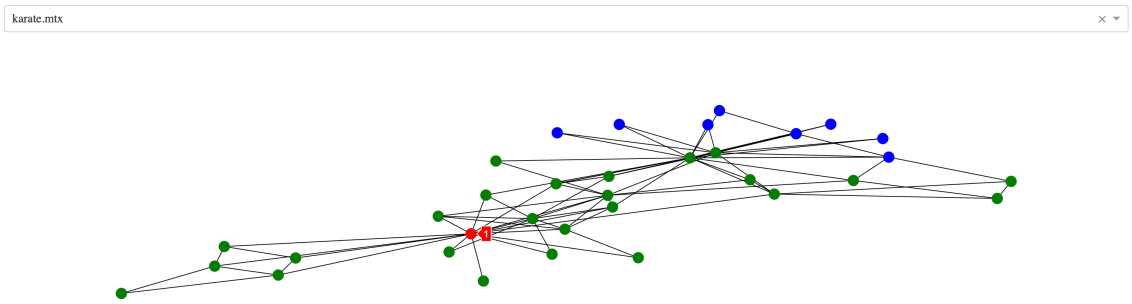
4.1.1. Simulation of the burning process

As we explain in Chapter 2, the burning process consists of two actions that take place at each round. First, a burning source is chosen. Then, in each subsequent time step, the fire spreads to all unburned neighbors of previously burned nodes, and a new burning source is selected. At the start, the graph is displayed with all nodes in blue, representing unburned nodes. Clicking on a node selects it as a burning source, changing its color to red. This is visualized in Figure 4.2. After selecting a source, pressing the 'Next time step' button increases the time by one. At this point, the fire spreads: all unburned neighbors (blue neighbors) of burned nodes turn orange to indicate they are now burned. The process then repeats, another source can be selected, followed by pressing the "Next time step" button. This continues until all nodes are either red or orange, meaning the entire graph has been burned.

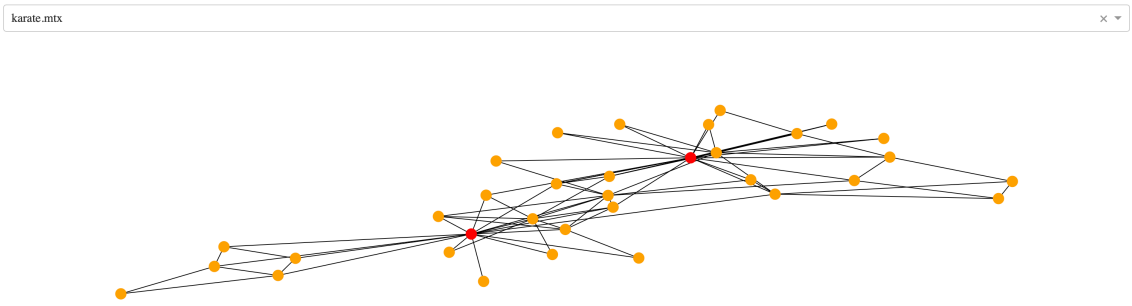
As the user burns the graph, all nodes that are chosen as a source are collected in the burning sequence. If a blue node is clicked, it is added at the end of the sequence. When no node is clicked in that round, a dash, $-$, will appear in the sequence. Then no new source is added in a round. This ensures that every source is placed correctly in the sequence. When a source node is clicked, the source will disappear from the sequence and will no longer be red. A new source can be selected as the source that replaces the deleted one. It is then not added at the end of the sequence, but at the place of the source it is replaces. All the nodes that are only burned by the removed source will disappear, and all the nodes that would be burned by the new source in the current amount of rounds will be burned (orange).



(a) Initial state of the graph.



(b) Selecting a source.



(c) Completely burned graph.

Figure 4.2: Initially, all nodes are colored blue (unburned). Nodes selected as a source, in this case node number 1, are colored red and all nodes selected due to the guess appear in green. In this figure, the network soc-karate is shown [16]. At the end all nodes are either red or orange.

To support experimentation and testing of strategies or theories, the tool includes an 'Undo' button. This button reverses only the most recent action, not the entire round. For example, if the fire has spread and a new source is chosen, pressing 'Undo' will remove only the newly selected source. If a complete reset is desired, the button "Reset" should be pressed. Then all the nodes are unburned and the burning sequence is empty again. Besides the selected network, the buttons and more information are given as shown in Figure 4.3.

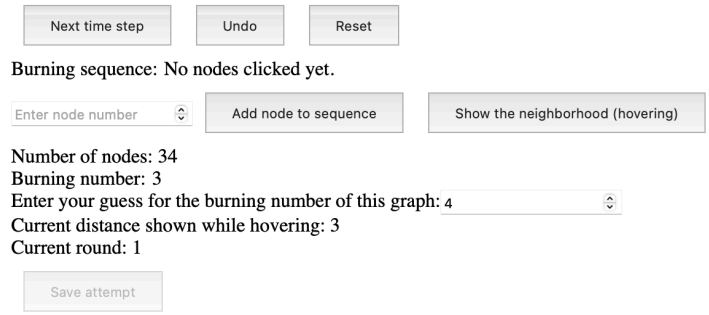


Figure 4.3: There are 3 buttons present: Next time step, Undo, Reset. Besides the buttons, there is some information about the graph available and the possibility to enter a guess. This is based on the network soc-karate [16].

4.1.2. Finding the burning number of a graph

Now that it is clear how the interactive graph tool can be used to simulate the burning process, this section explains the additional features that could help with finding the burning number of a graph and/or the optimal burning sequence(s).

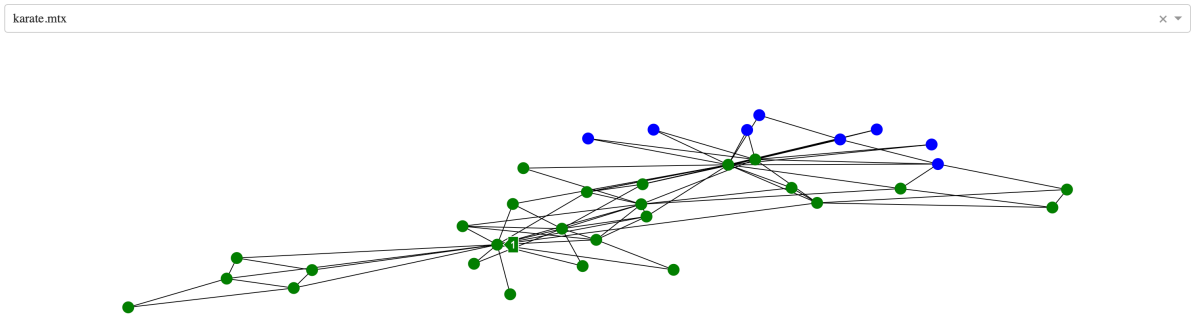
The number of nodes is shown as well as the burning number of the graph, if this number is known. If the burning number is not known, a question mark is shown. Showing the burning number, if it is known, adds value for educational purposes. It provides immediate feedback on how close the proposed solution is to the actual minimum, and it sets a clear target for users. This makes the experience more engaging and motivating, as it introduces a game-like challenge.

A feature of the interactive tool is the incorporation of a game element that allows users to guess the burning number of a given graph. Users can enter a guess for the burning number, denoted by x, before starting the burning process. The default value for this guess is set to 4. All these features are shown in Figure 4.3.

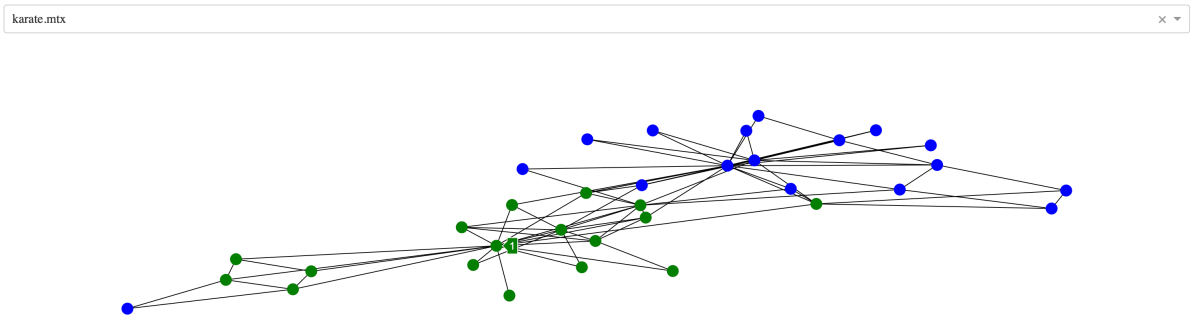
Once a guess is entered, the visualization provides feedback during the simulation to support intuitive understanding of the burning number. As the user hovers over nodes in the graph, the hovered node is highlighted in green. In addition, all nodes within a distance of $x - r$ from the hovered node are also highlighted in green, where r is the current round number. This visualization indicates the subset of nodes that would be burned in the remaining $x - r$ rounds, if the hovered node is chosen as a source. The underlying idea comes from the definition of burning: any newly selected source has only a limited number of remaining rounds to spread fire, so it can only reach nodes within that corresponding distance [2]. For example, if the user guesses the burning number to be $x = 3$, then at round 1, any hovered node highlights all nodes within distance $3 - 1 = 2$ in green. These nodes form the radius-t neighborhood of the hovered node, where $t = x - r$, and represent the part of the graph that could still be burned from that position in the remaining rounds. This feature, shown in Figure 4.4, helps users strategically decide where to place burning sources to reach unburned nodes within the guessed time.

As the simulation progresses and the round number increases, the highlighted radius in green decreases. This is because each subsequent burning source has fewer time steps left to spread the fire. Once the round number exceeds the guessed burning number, $r > x$, the green hover effect disappears. At this point, the user has surpassed their own prediction for the burning number, indicating that their guess was too low to complete the burning within the allowed time with the chosen burning sources.

This interactive feedback mechanism not only adds a game-like element to the tool but also reinforces the user's understanding of the concept of distance-based spreading and the optimization challenge behind finding the burning number. The user can afterwards try another sequence of the same length, or try for a larger burning number, to allow for larger radius when burning.



(a) The entered guess for the burning number is 3. Node 1 is hovered and the green nodes are all the nodes that would be burned by spreading in the remaining rounds if node 1 is chosen as a source in the current (first) round.



(b) The entered guess for the burning number is 2. Node 1 is hovered and the green nodes are all the nodes that would be burned by spreading in the remaining rounds if node 1 is chosen as a source in the current (first) round.

Figure 4.4: The hovering in combination with the guess is very helpful to gain insight into which nodes will be burned within the given number of rounds, and to check whether the chosen guess for the burning number is realistic. In particular, if the highlighted nodes cover the entire graph when using the guessed value, this suggests that the guess could indeed be the correct burning number or that the guess is a bit to high. This is the network soc-karate [16].

4.1.3. High-scores

Another valuable feature of the interactive tool is its ability to record and display user attempts. This functionality not only aids users in experimenting with different burning sequences, but also connects to broader principles from citizen science and gamification. Much like the game *FoldIt* [20], where non-experts contribute to solving complex protein-folding problems, this tool invites users to actively engage with open graph theory questions. By allowing users to iteratively test ideas and receive feedback, the tool encourages exploration and learning in a playful way.

To support this process, users can save a completed attempt after successfully burning the entire graph. The 'Save attempt' button becomes active only when all nodes have been burned, ensuring that only valid and complete solutions are stored. After saving, users can begin a new attempt by clicking the 'Reset' button. Each saved attempt appears in a list of the five most recent tries, allowing users to reflect on their progress and strategies.

In addition to showing recent attempts, the tool highlights the three best performances. An attempt is considered better if it uses fewer rounds to burn the graph. If multiple attempts take the same number of rounds, preference is given to the one with dashes in its sequence. When there are several attempts with dashes, the one with the dash the earliest in the sequence is ranked higher than a sequence with a dash later in the sequence. This ordering is called lexicographical. A *lexicographical ordering* compares sequences element wise. A source is translated to a 0 and a dash to a 1. Then the sequence where a dash occurs the earliest is ranked higher as $1 \geq 0$. A dash indicates a round in which no new source was chosen, demonstrating a more efficient spread from previous sources — a desirable outcome, as discussed in section 2.4. Figure 4.5 shows the high-score list.

To maintain flexibility in the user experience, the top three attempts are only displayed if the user chooses to view them by clicking the 'Show the high scores' button. This feature supports both self-guided exploration and social comparison, potentially motivating users to optimize their solutions and contribute to broader insights about the burning number problem.

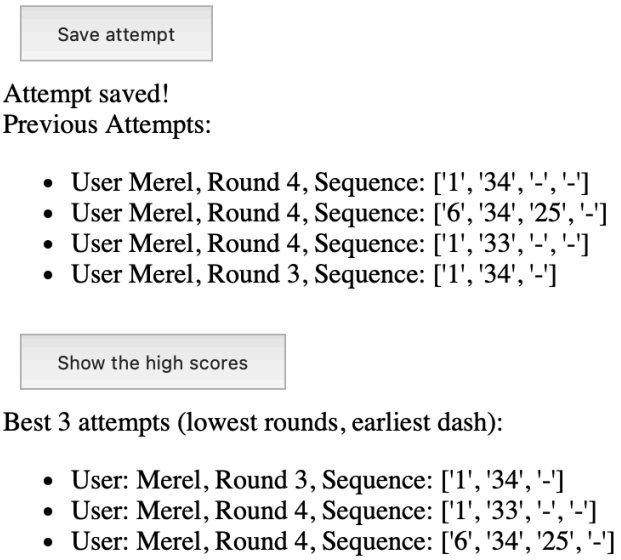


Figure 4.5: The two buttons, Save attempt and Show the high scores. After a valid burning sequence, the Save attempt button is available and the attempt can be saved. The most recent attempts are shown. If the Show high scores button is pressed, the best 3 attempts are visible. Press the button again and the list disappears. This is based on the network soc-karate [16].

4.1.4. Adding your own graph

The last feature of the interactive tool is the possibility to create and work with your own graph. While the tool initially presents a drop-down menu containing several predefined graphs, the final option in this menu is labeled “Create your own graph.” When this option is selected, users are given the opportunity to enter their own graph data and assign it a custom name, shown in Figure 4.7. Once submitted, the graph is saved and added to the drop-down menu, allowing users to interact with their own graph just like with the predefined ones.

To ensure the tool can correctly read the custom graph, the data must follow a specific format. The file should be structured as follows:

- Line 1: The number of nodes in the graph.
- Line 2: The number of edges.
- Line 3: Either the known burning sequence (the numbers of the source nodes separated by a comma), or a question mark “?” if the burning number is unknown.
- Line 4 and onward: Each line represents an edge in the graph, consisting of two integers indicating the endpoints of the edge.

An example of such a file for a path of length 9 might look like this:

```
9
8
3,7,9
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
```

Note that the nodes might be labeled differently, so it is not necessary that the nodes are labeled $1, ..., n$. The result will be a figure like Figure 4.6

Figure 4.6: This is a path graph of length 9 resulting from the above example file.

In addition to entering this graph structure, the user must also provide a name for the graph. This name is used to save the file in the folder with all other graphs and will appear as a new option in the drop-down menu for future use. The graphs are stored in the dataset folder and can be found on GitHub [18].

This feature is especially useful for users who wish to experiment beyond the predefined examples. By allowing the creation of custom graphs, the tool supports deeper exploration and testing of specific hypotheses or scenarios. Users can construct graphs with unique structures, test burning sequences, and investigate how different configurations influence the burning number. This freedom to create and explore makes the tool valuable not only for education, but also for research and experimentation.

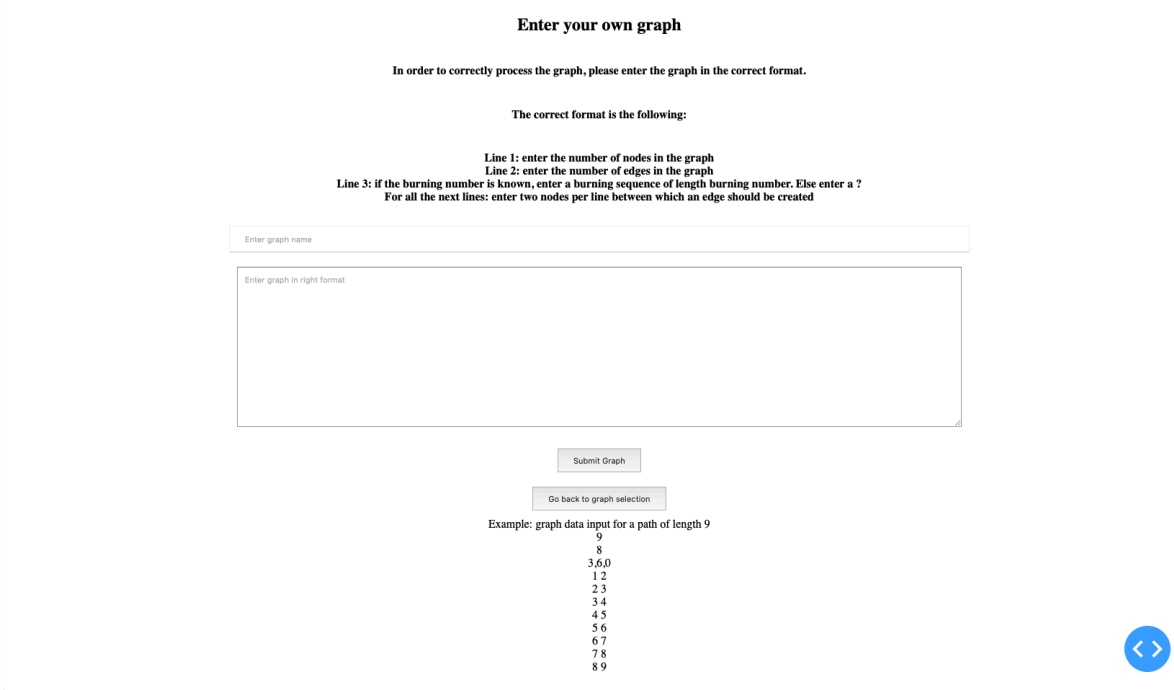


Figure 4.7: On this page a new graph can be entered. There is an explanation on the mandatory format and there is an example present.

4.2. Implementation

The interactive tool was implemented using Python, relying primarily on the Dash framework by Plotly. Dash is framework for building data apps in Python and provides a flexible structure for interactive components.

The graph data is processed using the NetworkX library. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [9]. All calculations such as calculating distances are also done by using NetworkX.

The application is structured as a web app, where user interactions, such as clicking, hovering, or selecting a node, trigger callbacks that update the visualization and the state of the system. The layout of the web application is built using standard Dash components like buttons, input fields, drop-down menus, and containers. The interface is designed to be intuitive and clean, supporting both first-time users and researchers experimenting with more complex graphs. All graphs, including user-generated ones, are stored as text files in a predefined format. This ensures that the tool remains extensible and allows users to easily add and explore new graph structures.

Overall, the combination of Dash, Plotly, and NetworkX provides a robust and flexible foundation for building an easy to use, educational and exploratory tool for graph burning.

For now, the program is hosted locally due to the time limitations of this thesis. The code is available on GitHub [18]. Hosting the program online would make it more accessible to a wider audience. Moreover, an online version would allow users to share their results, making the high-score list meaningful, since scores from all users would be visible and comparable.

5

The cooling number

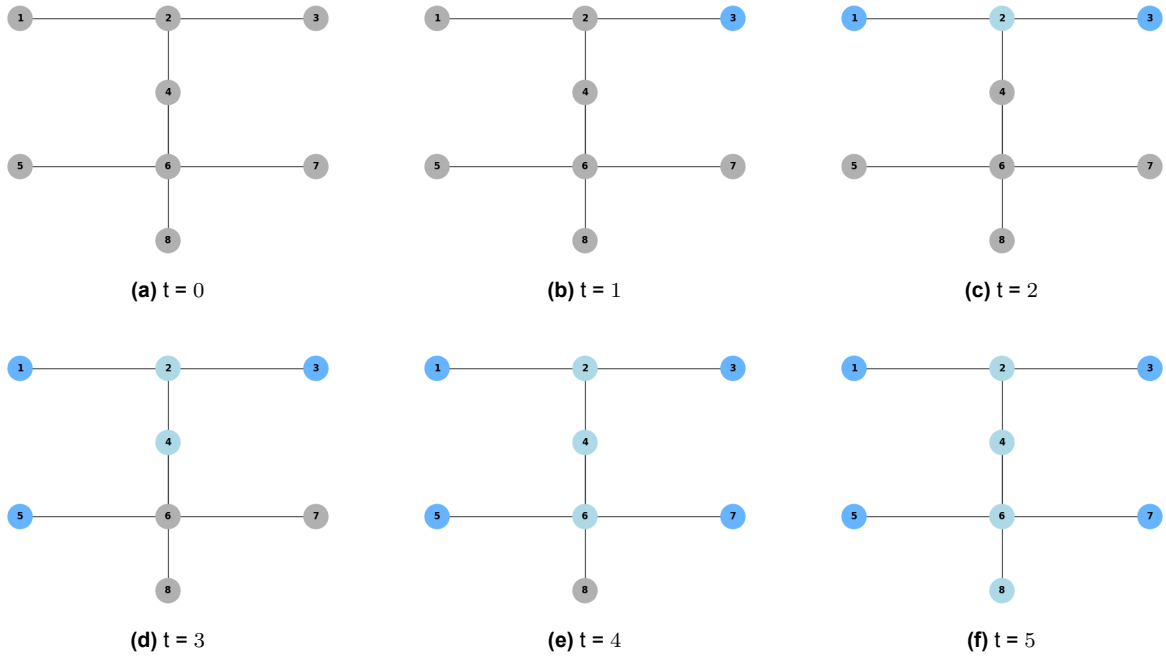
In Chapter 2, the burning number is introduced. While the burning number measures how quickly a fast-spreading contagion can take over a graph, the cooling number does the opposite. It measures how slowly a slow-moving process or contagion can spread through the graph. In this case, a lower cooling number means the contagion spreads faster, and a higher cooling number means it spreads slower [4].

The cooling process works in rounds. At the start, all nodes in the graph are uncooled. In the first round, one node is chosen as a cooling source. Just like in the burning process, in each round, all neighbors of cooled nodes also become cooled. After this spreading step, one new source is selected for the next round. This continues until all nodes are cooled.

However, there is an important difference from the burning number. In the burning problem, the goal is to minimize the number of sources needed to burn the graph [7]. In the cooling problem, the goal is to maximize the number of time steps it takes to cool the entire graph [4]. A cooling sequence for a graph is a sequence that records the cooling sources in the exact order they are chosen during the cooling process.

More formally, the cooling number of a graph G, CL(G), is the maximum number of rounds for the cooling process needed to cool the entire graph. The cooling sequence is the sequence of cooling sources selected during the cooling process [4].

A visual presentation of a cooling process of a graph is given in Figure 5.1. At time $t = 1$, only a cooling source is chosen. Every next round, first the spreading is happening, and then a new cooling source is chosen. For example, at time $t = 2$, first node 2 is cooled since it is a neighbor of node 3. Then node 1 is chosen as a new cooling source. The cooling number of Figure 5.1 equals 5 with corresponding cooling sequence $(3, 1, 5, 7, -)$, where $-$ indicates that in the last round no additional source is chosen. In the cooling sequence, a dash can only appear in the last place and not in any earlier place. This example gives a lower bound on the cooling number, so Figure 5.1 shows that the cooling number of that graph is ≥ 5. However, for this graph it is true that there is no longer cooling sequence.



(a) t = 0 **(b)** t = 1 **(c)** t = 2

(d) t = 3 **(e)** t = 4 **(f)** t = 5

Figure 5.1: The cooling process of a tree with 8 nodes. The darker blue nodes are the cooling sources and the light blue nodes are cooled by the propagation process. The cooling number of the graph is 5. The corresponding cooling sequence is $(3, 1, 5, 7, ' - ')$.

5.1. Integer Linear Program

In order to calculate the optimal cooling number of a graph, we will introduce an ILP. The upcoming sections present the ILP to calculate the cooling number of a given graph. First, the variables and constraints are introduced and explained. After that, the complete ILP is given.

5.1.1. Variable declaration

In order to explain the ILP in a clear way, we first define the used variables. We use i to indicate the nodes in the graph, so $i \in V$. j is used to indicate the round number, so $j \in 1, ..., U$. U represents the upper bound on the number of rounds needed to cool the entire graph. An example of an upper bound is the number of nodes in the graph. However, the higher the upper bound, the more redundant rounds are calculated and the higher the running times are. Bonato et. al. provide an upper bound that can be used for the graphs studied in this work, $\text{CL}(G) \leq \lceil \frac{n+1}{2} \rceil$ [4]. All the variables used are:

- y_j is a binary variable. $y_j = 1$ if there exists an uncooled node at round j and it equals 0 else.
- z_j is a binary variable. $z_j = 1$ if all the nodes are cooled at the beginning of round j and it equals 0 else. Note that z_j is the opposite of y_j.
- $s_{i,j}$ is a binary variable. $s_{i,j} = 1$ if node i is chosen as a cooling source at round j.
- $c\_startround_{i,j}$ is a binary variable. $c\_startround_{i,j} = 1$ if node i is cooled at the beginning of round j.
- $c\_afterspread_{i,j}$ is a binary variable. $c\_afterspread_{i,j} = 1$ if node i is cooled after the spreading phase in round j.
- $c\_aftersource_{i,j}$ is a binary variable. $c\_aftersource_{i,j} = 1$ if node i is cooled after the source picking phase in round j.

5.1.2. Complete ILP with explanation

$$\text{max.} \quad \sum_{j\in[1,U]} y_j \tag{5.1}$$

$$\text{s.t.} \quad \sum_{i\in[n]} s_{i,1} = 1 \tag{5.2}$$

$$s_{i,j} \leq 1 - c\_afterspread_{i,j} \qquad \forall i \in [n], j \in [1,U] \tag{5.3}$$

$$s_{i,j} \leq c\_aftersource_{i,j} \qquad \forall i \in [n], j \in [1,U] \tag{5.4}$$

$$c\_afterspread_{i,j} \geq c\_startround_{i,j} \qquad \forall i \in [n], j \in [1,U] \tag{5.5}$$

$$c\_aftersource_{i,j} \geq c\_afterspread_{i,j} \qquad \forall i \in [n], j \in [1,U] \tag{5.6}$$

$$\sum_{i\in[n]} s_{i,j} \geq 1 - c\_afterspread_{i,j} \qquad \forall i \in [n], j \in [1,U] \tag{5.7}$$

$$\sum_{i\in[n]} s_{i,j} \leq 1 \qquad \forall j \in [1,U] \tag{5.8}$$

$$z_j \geq \sum_{i\in[n]} c\_aftersource_{i,j-1} - n + 1 \qquad \forall j \in [2,U] \tag{5.9}$$

$$c\_afterspread_{i,j} \geq c\_aftersource_{k,j-1} \qquad \forall j \in [2,U], k \in N(i) \tag{5.10}$$

$$c\_aftersource_{i,j-1} = c\_startround_{i,j} \qquad \forall i \in [n], j \in [2,U] \tag{5.11}$$

$$y_j + z_j = 1 \qquad \forall j \in [1,U] \tag{5.12}$$

$$y_j, z_j \in \{0,1\} \qquad \forall i \in [n], j \in [1,U] \tag{5.13}$$

$$s_{i,j} \in \{0,1\} \qquad \forall i \in [n], j \in [1,U] \tag{5.14}$$

$$c\_startround, c\_afterspread, c\_aftersource \in \{0,1\} \qquad \forall i \in [n], j \in [1,U] \tag{5.15}$$

As the goal is to maximize the number of rounds until there are no more uncooled nodes, the objective function is given by equation (5.1).

- Constraint (5.2) forces exactly one source in the first round.
- Constraints (5.3) and (5.4) ensure that a cooled node cannot be chosen as a cooling source and that each source is immediately cooled.
- Constraints (5.5), (5.6) and (5.11) ensure that once a node is cooled, it stays cooled.
- Constraint (5.7) forces at least one source in round j if there is a uncooled source after the spreading phase. The fact that there can only be one source is forced by constraint (5.8).
- If all nodes are cooled at the end of round $j-1$, the cooling process is done and there is no need for an extra round. This step is done in constraint (5.9).
- Constraint (5.10) represents the spreading phase. If in round j, node i has a neighbor that is cooled at the end of round $j-1$, then node i is cooled after the spreading phase in round j.
- Lastly, constraint (5.12) shows the connection between y and z.

5.1.3. Run time analysis

Appendix B.1 contains the code for the complete implemented ILP given in section 5.1.2. There are several factors that could influence the time it takes the ILP given in section 5.1.2 to find an optimal solution. These factors are among other the number of nodes and the edge

density. *Edge density* is the ratio between edges in the graph and the maximum number of edges in the graph [6]. The formula for the edge density of a graph G with $|V|$ number of vertices and $|E|$ number of edges is

$$\text{edge density of } G = \frac{2 \cdot |E|}{|V|(|V|-1)}.$$

In order to analyze the run time of the cooling ILP, we created connected graphs with different number of nodes and different edge densities. The amount of nodes in the graphs are $[10, 20, 50, 80, 100, 150, 200, 300]$ and for each of these amount of nodes, we used $[0.2, 0.4, 0.6, 0.8, 1.0]$ as edge densities. The running times are also depended on the device used to collect the data. This data is collected on a MacBook Pro with an Apple M4 chip. The upperbound for the ILP is set to $\lceil \frac{n+1}{2} \rceil$. Figure 5.2 shows the run time versus the number of nodes for different edge densities. It can be seen that the run time increases fast when the number of nodes increases. The figures shows mixed effect of the the edge densities. For higher amount of nodes, the edge density of 0.4 takes significantly more time. Based on the empirical results the program runs between quadratically and cubically.

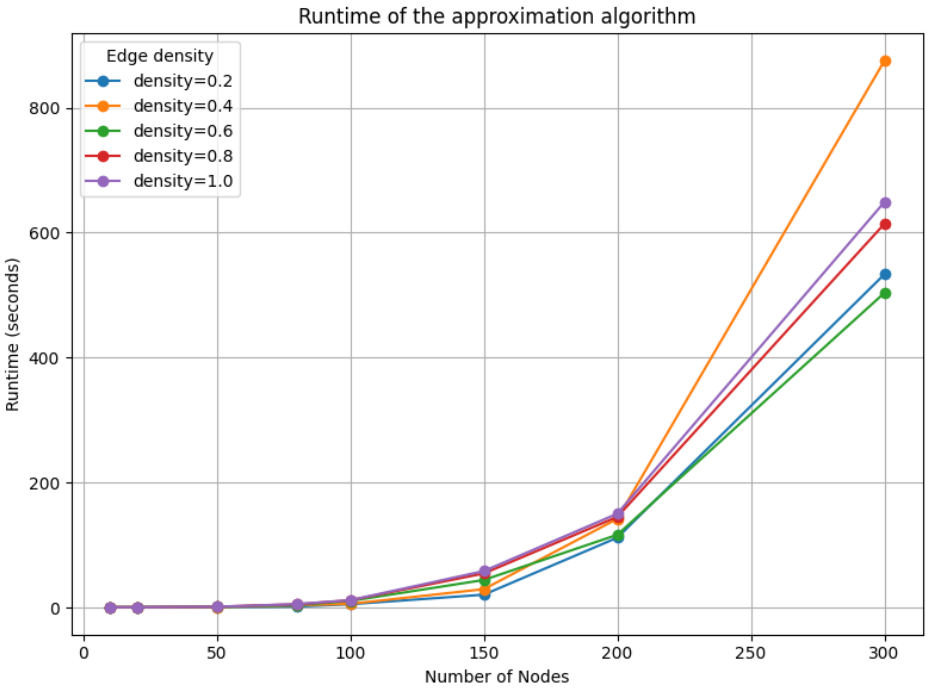


Figure 5.2: Running times of the cooling ILP with $U = \lceil \frac{n+1}{2} \rceil$ for graphs with variable number of nodes and different edge density

Table 5.1 shows the exact running times for the different amount of nodes and edges. There are two column for the running times. The first column contains the running times corresponding to Figure 5.2. The last column contains the running times corresponding to the same ILP but then with an upper bound of n. The big differences in the running times show that finding a tight upper bound could really benefit the running times.

$\|V\|$	$\|E\|$	edge density	run time for $U = \lceil \frac{n+1}{2} \rceil$ (seconds)	run time for $U = n$ (seconds)
10	10	0.2	0.417	0.418
10	14	0.4	0.399	0.410
10	27	0.6	0.412	0.403
10	41	0.8	0.408	0.417
10	45	1.0	0.403	0.404
20	35	0.2	0.420	0.438
20	83	0.4	0.422	0.445
20	119	0.6	0.435	0.472
20	152	0.8	0.433	0.471
20	190	1.0	0.433	0.487
50	235	0.2	0.675	0.881
50	488	0.4	0.801	1.441
50	709	0.6	0.969	1.613
50	967	0.8	1.148	2.332
50	1225	1.0	1.227	2.250
80	638	0.2	1.512	5.232
80	1221	0.4	3.605	8.422
80	1936	0.6	3.843	8.923
80	2510	0.8	5.070	11.312
80	3160	1.0	5.391	15.490
100	979	0.2	5.222	12.428
100	1963	0.4	5.802	14.972
100	2967	0.6	10.494	28.688
100	3955	0.8	11.566	27.307
100	4950	1.0	11.688	32.389
150	2252	0.2	20.476	91.952
150	4425	0.4	29.319	63.039
150	6648	0.6	44.035	95.756
150	8957	0.8	54.483	109.803
150	11175	1.0	58.282	151.221
200	4008	0.2	111.754	276.910
200	8035	0.4	141.907	294.241
200	11841	0.6	116.392	242.529
200	16013	0.8	145.075	315.648
200	19900	1.0	150.161	388.518
300	8902	0.2	533.295	1003.795
300	18045	0.4	875.065	908.562
300	27010	0.6	503.553	1312.065
300	35948	0.8	614.107	1721.110
300	44850	1.0	648.960	$>$3600

Table 5.1: Run time of the ILP for random connected graphs of varying sizes, edge densities and two different upper bounds.

5.2. Approximation algorithm

It is strongly suspected that the cooling number problem is NP-hard. However it is not proven that the problem is actually NP-hard. In section 5.1.3 is the run time of the ILP explained. As can be seen, the run times are increasing rapidly when the number of nodes increase. Therefore, this section provides a 2-approximation algorithm for the cooling number of a graph.

5.2.1. Approximation algorithm

The approximation algorithm designed in this thesis is based on the diameter of the graph. Let G be the graph of which we want to get an approximation of the cooling number. First, the diameter is found, together with all the paths in G which are of diameter length, called the diameter paths.

The algorithm is done for a random choice among all the diameter paths, let this path be $(v_1, ..., v_d)$. v_1 is chosen as cooling source in the first round. Then the the cooling spreads to all neighbors of v_1. After that the source picking starts. Then all nodes with largest distance to v_d are found, lets say set S contains all these nodes. If there exists a node in S that is not contained in $(v_1, ..., v_d)$, this node is chosen as a source. If there is no of diagonal option, every node in S can be selected as a source. In both cases, if there are multiple nodes that can be

chosen as a source, the choice is made random. This process is repeated until all nodes are cooled. The Python code that is used to find the diameter and all paths with this length is given in section B.3. The full Python implementation of the algorithm is given in section B.2, and the pseudocode is shown in Algorithm 5.

Algorithm 5 Approximation algorithm

Require: Graph $G = (V, E)$
Ensure: A cooling sequence of the graph
  (diameter, diameterPaths) $\leftarrow$ DiameterAndPath(G)
  Pick $(n_1, n_2, ..., n_k)$ $\in$ diameterPaths
  cooledNodes $\leftarrow \emptyset$
  sources $\leftarrow \emptyset$
  round $\leftarrow 1$
  distances $\leftarrow \{(u, d(u, n_k)) : u \in V\}$
  while |cooledNodes| $< |V|$ **do**
    uncooledNodes $\leftarrow |V| \setminus$ cooledNodes
    for node $\in$ uncooledNodes **do**
      if neighbors(node) $\cap$ cooledNodes $\neq \emptyset$ **then**
        cooledNodes $\leftarrow$ cooledNodes $\cup$ {node }
      end if
    end for
    if round $= 1$ **then**
      source $\leftarrow n_1$
    else
      uncooledDistances $\leftarrow \{(u, d(u, n_k)) : u \in V \setminus$ cooledNodes$\}$
      if uncooledDistances $\neq \emptyset$ **then**
        candidates $\leftarrow$ all vertices $u \in V \setminus$ cooledNodes such that $d(u, n_k)$ is maximal
        offPath $\leftarrow$ all $u \in$ candidates such that $u \notin (n_1, n_2, ..., n_k)$
        if there exists such a u in offPath **then**
          $r \leftarrow$ a random vertex of offPath
          source $\leftarrow r$
        else
          $r \leftarrow$ a random vertex of candidates
          source $\leftarrow r$
        end if
      else
        source $\leftarrow \emptyset$
      end if
    end if
    cooledNodes $\leftarrow$ cooledNodes $\cup$\{source\}
    sources $\leftarrow$ sources $\cup$\{source\}
    round $\leftarrow$ round $+1$
  end while
  return round, sources

5.2.2. Precision of the approximation algorithm

Now that the algorithm is explained, it is time to analyze its performance. First, the run time is shown to be polynomial and after that it is proven that the cooling sequence as a result of Algorithm 5 is at least half the optimal length.

Theorem 1. *Algorithm 5 runs in polynomial time. More precisely, the run time is* $\mathcal{O}(nm + n^2)$.

Proof. Fixing a path of length equal to the graph's diameter ensures that the algorithm runs in polynomial time. From now on, we let $n = |V|$ and $m = |E|$. The first step involves computing the shortest distances from every node to the last node of the fixed path. This is done using a function from NetworkX, *single_source_shortest_path_length*. This function uses Breadth-First Search and has a complexity of $\mathcal{O}(n + m)$.

After this pre-processing step, the main part of the algorithm begins. This consists of a while-loop that continues until all nodes have been cooled. In the worst case, this loop can run up to n times, since in each round at least one new node is cooled.

The first operation inside the while-loop is the spreading phase, where the algorithm iterates through all uncooled nodes and checks whether any of their neighbors are already cooled. This step requires scanning the neighbor lists of uncooled nodes, which in the worst case leads to a total time complexity of $\mathcal{O}(nm)$ over all rounds.

Next is the source selection phase. In each round, the algorithm selects a new source node to continue the cooling process. This involves filtering out already cooled nodes, identifying the nodes farthest from the endpoint of the path, and randomly picking one of them. Each of these steps requires at most $\mathcal{O}(n)$ time per round, resulting in an overall cost of $\mathcal{O}(n^2)$ across all rounds.

Putting these parts together, the total time complexity of the algorithm, when a single fixed path is used, is: $\mathcal{O}(m + n + nm + n^2)$, which simplifies to $\mathcal{O}(nm + n^2)$. □

Now that it is clear that the algorithm runs in polynomial time, the precision of the algorithm can be proven. An algorithm is a 2-approximation for the cooling number problem if and only if for every input x of the problem, $\frac{\text{OPT}(x)}{2} \leq \text{ALG}(x) \leq \text{OPT}(x)$. Here OPT($x$) is the cooling number and ALG(x) is the result of Algorithm 5 [10].

Theorem 2. *Algorithm 5 is a* 2*-approximation algorithm for the cooling number problem.*

Proof. The first condition for an algorithm to be an approximation algorithm is that it runs in polynomial time. For Algorithm 5, this is shown in Theorem 1. It remains to prove that Algorithm 5 is a 2-approximation. Let OPT(x) be the optimal solution for the cooling number problem for instance x. This optimal solution is the result of the ILP. Let ALG(x) be the solution for the cooling number problem for instance x, resulting from Algorithm 5.
There are two bounds to prove: ALG$(x) \leq$ OPT(x) and $\frac{\text{OPT}(x)}{2} \leq$ ALG(x).
The first bound follows from the fact that ALG(x) results in a feasible solution of the ILP. Since OPT(x) is defined as the maximum over the feasible region, it holds that ALG$(x) \leq$ OPT(x). The solution produced by ALG(x) belongs to the feasible region, since in every round exactly one source is chosen (except potentially in the final round), and in each round, all nodes adjacent to a cooled node are also cooled through spreading.
It then remains to show that $\frac{\text{OPT}(x)}{2} \leq$ ALG(x) holds. Let $(u, ..., v)$ be the diameter path. First, note that the first source is fixed to be u, and in every round, the node w such that $d(w, v)$ is maximal is chosen as a source. Since $d(w, v)$ is at most Diam$(G) + 1$ steps away, we have OPT$(x) \leq$ Diam$(G) + 1$ [4]. In the worst case, each round cools only two new nodes on the diameter, one being the selected source, and one being cooled through spreading. Therefore,

at least $\frac{\text{Diam}(G)+1}{2}$ rounds are required. It follows that $\text{ALG}(x) \geq \frac{\text{Diam}(G)+1}{2}$. Combining these, it can be seen that $\frac{\text{OPT}(x)}{2} \leq \text{ALG}(x)$. Therefore, Algorithm 5 is a 2-approximation. □

5.2.3. Run time analysis

In order to analyze the run time of the approximation algorithm, we again created connected graphs with different numbers of nodes and different edge densities. The numbers of nodes were chosen to be $\{10, 20, 50, 80, 100, 150, 200, 300\}$, and the edge densities were $\{0.2, 0.4, 0.6, 0.6, 0.8, 1.0\}$. These numbers are exactly the same as for the ILP. This it better for the comparison between the two. Figure 5.3 shows the run time for these graphs with the different edge densities. For the smaller graphs, there is not a significant different between the approximation algorithm and the ILP. However, for all the bigger graphs the approximation algorithm is way faster than the ILP. The empirical running times shown in Table 5.2 are consistent with Theorem 1.

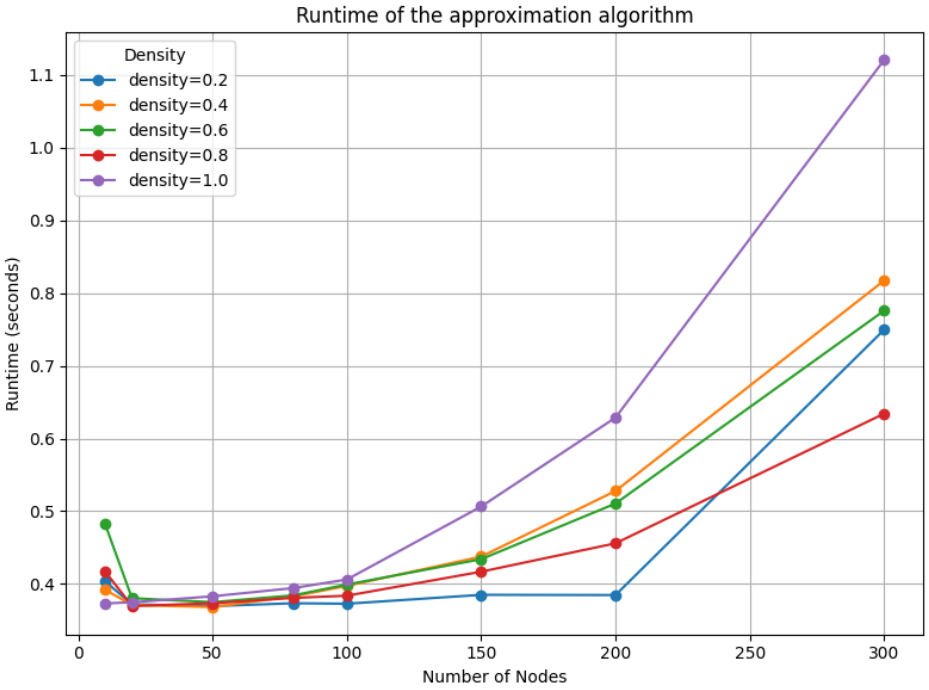


Figure 5.3: Plot of runtime of the approximation algorithm for graphs with variable number of nodes and different edge density

$\|V\|$	$\|E\|$	edge density	run time (seconds)
10	10	0.2	0.404
10	17	0.4	0.392
10	29	0.6	0.482
10	38	0.8	0.417
10	45	1.0	0.373
20	36	0.2	0.372
20	70	0.4	0.371
20	105	0.6	0.380
20	151	0.8	0.370
20	190	1.0	0.375
50	236	0.2	0.370
50	474	0.4	0.368
50	743	0.6	0.375
50	947	0.8	0.374
50	1225	1.0	0.383
80	655	0.2	0.374
80	1303	0.4	0.383
80	1882	0.6	0.384
80	2498	0.8	0.381
80	3160	1.0	0.394
100	1001	0.2	0.373
100	2029	0.4	0.397
100	2927	0.6	0.399
100	3978	0.8	0.384
100	4950	1.0	0.406
150	2211	0.2	0.385
150	4532	0.4	0.438
150	6827	0.6	0.434
150	8980	0.8	0.417
150	11175	1.0	0.507
200	3893	0.2	0.385
200	7945	0.4	0.528
200	11916	0.6	0.511
200	15979	0.8	0.456
200	19900	1.0	0.629
300	8945	0.2	0.750
300	18098	0.4	0.817
300	26799	0.6	0.776
300	35859	0.8	0.634
300	44850	1.0	1.121

Table 5.2: Runtime of the approximation algorithm for random connected graphs of varying sizes and densities.

5.3. Optimal cooling sequence of spider graphs with 3 legs

A specific type of graph is a spider graph. A *spider* is a tree where there is one node, the head, with degree greater than 2. All other nodes in the spider have degree 1, called a leaf, or degree 2 [13]. The degree of the head corresponds to the number of legs of the spider. A *leg* of a spider is a path from the head to a leaf. In this section, we focus only on spiders with 3 legs, so the degree of the head is 3. An example of such a spider is given in Figure 5.4. In this example the legs of the spider are $\{1, 2, 3, 4, 5, 6, 7\}, \{1, 8, 9, 10, 11, 12, 13, 14, 15\}$ and $\{1, 16, 17, 18, 19, 20\}$.

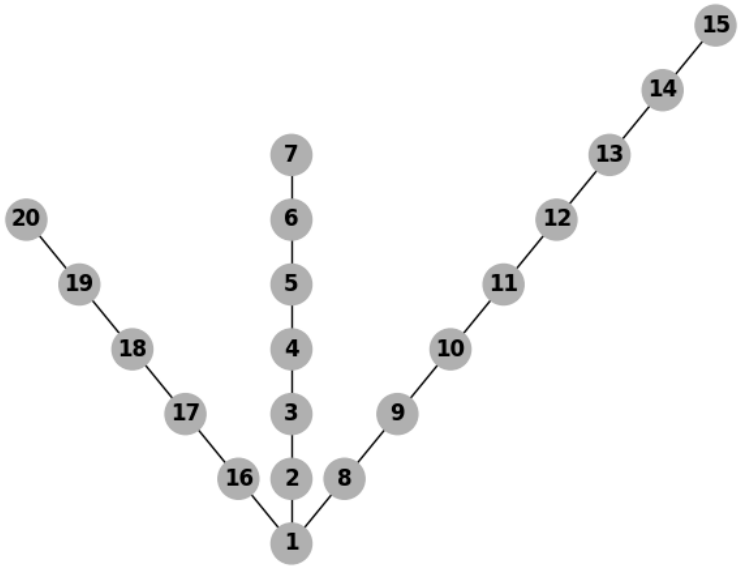


Figure 5.4: This is a spider graph with 3 legs. The head node is node 1.

We will first define some notation. Let $S = (s_1, ..., s_n)$ be a cooling sequence for a spider graph G. We denote S_i to be the number of sources in S that lie on leg L_i and l_i is the number of edges on L_i.
We define L_1 to be the leg which contains source s_1. Let L_1 consists of nodes $x_1, x_2, ..., x_{l_1}$ where x_1 is the leaf node of the leg and x_{l_1} is the neighbor of the head.
To indicate the *concatenation* of sequences A and B, we will use notation $A \parallel B$.

First, we prove a theorem that states that all the sources at the beginning of the cooling sequence can lie on a single leg. This provides structure on how the cooling process can proceed in a spider. After that we formulate a conjecture that in a 3-leg spider, also the sources on the remaining two legs can be clustered per leg.

Lemma 3. *Sequence $S' = (x_1, x_3, ..., x_{2S_1-1}) \parallel ((s_1, ..., s_n) \backslash L_1)$ is a prefix of a valid cooling sequence for spider graph G, that is there exists a valid cooling sequence that starts with S'.*

Proof. First, we need to prove that in sequence S', the head is cooled no earlier than in the original sequence S. In order to show this, we define two cases:

1. Case 1: in sequence S', the head cools not via L_1, any other leg. This means that one or multiple of the other legs are completely cooled, which then cools the head, while the node on L_1 adjacent to the head is uncooled.
2. Case 2: in sequence S' the head cools via L_1. This means that L_1 is completely cooled, which then cools the head, while the nodes on the other legs adjacent tot the head remain uncooled.

First we consider case 1. Assume the head is not cooled via L_1 with cooling sequence S'. Since S' is altered such that the sources on L_1 are pushed to the beginning in S', the other sources, the ones on the other legs are pushed to later. If the sources on L_1 are pushed to the beginning of S', the head is definitely not cooled by L_1 earlier than it would in S. As the sources on the other legs are pushed to a later round in S', they cannot cool the head earlier

than in S. From this we can conclude that in this case the head is not cooled in an earlier round by S' than by S.

Now we consider the second case. Assume the head is cooled via L_1. Then we need to show that the first round in which L_1 is completely cooled with S' is larger or equal than when using S.

Define r as the first round in which leg L_1 is completely cooled under sequence S and r' is the first round in which leg L_1 is completely cooled under sequence S'.
First we look at sequence S. We know that at round r the number of nodes cooled by spreading together the number of sources is equal to l_1. The number of sources equals S_1. The number of nodes cooled by spreading is greater or equal to $r-1$. This is true since each round, the number of nodes cooled by spreading is at least one except for the first round. In the first round there is no node cooled by spreading. So we get

$$S_1 + r - 1 \leq l_1$$

For sequence S' we can do the same. At round r' the number of nodes cooled by spreading together with the number of sources is equal to l_1. Again the number of sources equals S_1, but now we also know that the number of nodes cooled by spreading is exactly $r'-1$ as the sources are pushed to the end of the leg and therefore each new source is added next to an already cooled node.

$$S_1 + r' - 1 = l_1$$

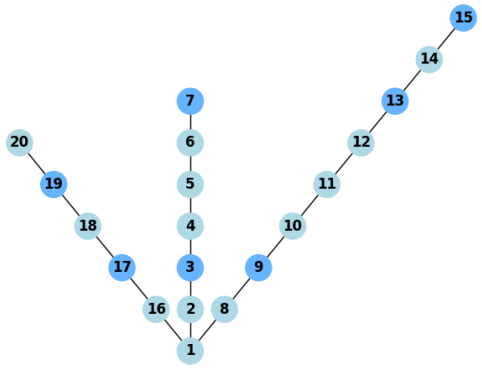
So if the head cools via L_1 using S', we combine these two equations and get $l_1 = S_1 + r' - 1 \geq S_1 + r - 1$, so $r' \geq r$. So we can conclude that the head is cooled not earlier using sequence S' instead of S.

Now that we have proven that the head doesn't cool earlier, the sequence
$S' = (x_1, x_3, ..., x_{2S_1-1}) \parallel ((s_1, ..., s_n) \backslash L_1)$ is a valid cooling sequence if it cools all the nodes of the graph. In the case that that is not true, there should be sources added to the end of the sequence and this addition creates a valid cooling sequence. □

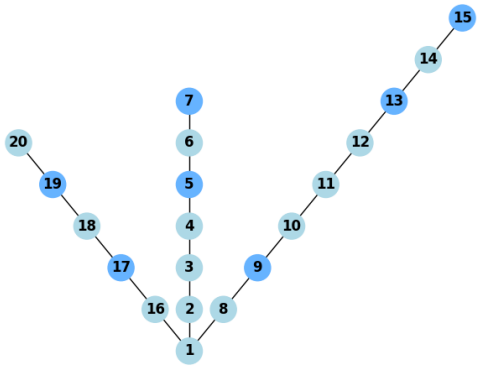
Figure 5.5 visualizes the result of Lemma 3. In Figure 5.5a the cooling sequence is $S = (7, 15, 13, 3, 9, 19, 17)$ and the head is cooled in round 6. The modified sequence is visualized in Figure 5.5b, where the cooling sequence is $S' = (7, 5, 15, 13, 9, 19, 17)$ and the head is also cooled in round 6.

Now that it is proven that the sources at the front of the sequence could be clustered on a single leg, we strongly suspect that this also holds for the sources on the remaining two legs. This statement is formulated by Conjecture 4.

Conjecture 4. *Let G be a spider graph with 3 legs, L_1, L_2, L_3. There always exists an optimal cooling sequence such that the source nodes are clustered per leg. This says, there always exists an optimal cooling sequence $s = (s_1, .., s_a, s_{a+1}, ..., s_b, s_{b+1}, ..., s_c)$ where $s_1, ..., s_a \in L_1$, $s_{a+1}, ..., s_b \in L_2$ and $s_{b+1}, ..., s_c \in L_3$.*



(a) In the sequence $S = (7, 15, 13, 3, 9, 19, 17)$, the sources are for example placed according to this figure. The dark blue nodes are the sources.



(b) In the sequence $S' = (7, 5, 15, 13, 9, 19, 17)$, the sources are pushed to the end of the leg 1. The dark blue nodes are the sources.

Figure 5.5: Visualization of clustering all sources on one leg at the beginning of the sequence and keeping the rest of the sources the same. 5.5a shows the initial cooling process and 5.5b shows the modified cooling process of the spider in Figure 5.4.

6

Conclusion

This goal of this thesis was to investigate the burning number and cooling number process. This is done by implementing a probabilistic approach for the burning number problem. Besides the interactive tool also the probabilistic approach, also an interactive tool was created using Python. And lastly, an optimization program for the cooling number was developed.

The original graph burning problem consists of two steps each round: a spreading phase and a source picking phase. In this thesis we introduce a probabilistic approach, in which the source picking phase is the same, but the spreading phase is slightly altered. Originally, spreading of burned nodes occurs with probability 1, so each round every node that is a neighbor of a burned node in the previous round is burned in the current round. As this is not very accurate in real-life problems, it is valuable to add a probability of spreading. By adding this probability, there is no way to calculate the exact number of rounds needed to burn the entire graph. Therefore, an algorithm is developed that outputs the expected number of rounds needed to burn the entire graph. The algorithm makes use of Markov Chains for the calculations of the two phases in the burning process.

The proposed interactive tool to simulate the burning process was completed successfully in Python. The program makes it possible to select any graph that a user wishes to burn. For that selected graph the burning process can be simulated by making use of button, clicking nodes or entering nodes. There is an option of entering a guess which helps to develop an intuition for finding the burning number by showing nodes that are burned in the remaining number of rounds. To incorporate a game like process, burning attempts could be saved. The interactive tool keeps track of previous attempts and high-scores. The attempts are sorted based on the number of rounds needed to burn the entire graph, the lower the better, and the amount of rounds without a chosen burning source. A burning sequence with a round without a burning source in an earlier round is preferred over a sequence without a burning source in a later round. The last feature which is a great addition is the possibility to easily add your own graph. By giving the information of the graph in the correct format, the graph is immediately created and the user can work on this graph directly.

In the last part of this thesis the cooling number is researched. As finding the cooling number by hand is obviously not that straightforward for larger graphs, the development of an integer linear program is very useful. This ILP is implemented in Python and consists of an objective function which maximize the time for which not all nodes are cooled. The constraints simulate the two different phases, by forcing every round to have a source and force every node in the

neighborhood of a cooled node to be cooled. The run time of the ILP is rapidly increasing when the number of nodes and the density are increasing. For a graph with 300 nodes, the run time is already about 500 seconds when the edge density is 0.6. As the running time of the ILP can increase fast for larger graphs, we gave a 2-approximation algorithm for the cooling number problem. The algorithm runs in polynomial time and is an 2-approximation. The run time of this approximation algorithm is significantly shorter than the ILP for most of the graphs. For graphs with a lower edge density, the run time of the approximation algorithm is about the same as the ILP.

Lastly, we stated a conjecture claiming that the sources in a cooling sequence of a 3-leg spider can be clustered per leg. We were not able to prove this conjecture. However, we did show that all the sources on one leg can be moved to the front of the sequence. Therefore, a cooling sequence can be divided into two blocks: the first block contains all the sources on one leg, and the second block contains all the sources on the remaining legs.

7

Discussion

In this thesis we explored graph burning and cooling from an implementation and a theoretical perspective. we introduced a probabilistic method to calculate the expected burning number for a given probability. we also created an interactive tool to simulate the burning process in a clear way using Python. Finally, we created an integer linear program to calculate the cooling number of a graph and provided a quicker 2-approximation algorithm. Together, this gives a variety of new findings regarding the burning and cooling number.

The introduction of a probabilistic approach creates a more realistic scenario. However, in our assumption all the spreading occurs with the same probability for each edge. Instead of a constant probability it could be helpful to adjust the probabilities for different edges. Additionally, the given algorithm calculates the expected number of rounds for the given probability and sequence. This sequence consists of a permutation of all the node numbers. For further research it could be beneficial to adapt the algorithm so that a sequence with length shorter than the number of nodes could be entered or even with dashes in the sequence.

Using Plotly was a great choice for the interactive tool since Python is a programming language that is widely known. However, at the start of this thesis we did not know anything about Plotly itself. Therefore, the program could be optimized more, possibly making it faster. There is a limit on the size of the graphs that can be used in this program. One problem is the loading time of the graphs. NetworkX is the package that is used for loading of the graph. For larger graphs this takes a lot of time. Besides the loading time, the display of larger graphs is also not that clear. This could be a beneficial improvement. For the score ordering a lexicographical approach was chosen. In this ordering, the earlier a dash appears in the sequence, the better the burning sequence is considered. This is based on the idea that an early dash reflects more efficient source placement, as the chosen sources already cover a large part of the graph without requiring additional sources. At an earlier stage of this project, the number of dashes alone was used to determine the ranking of attempts. An additional advantage of having multiple dashes, especially later in the sequence, is that they indicate that several rounds did not require a new source at all, which demonstrates strong overall efficiency of the chosen sequence even if the first dash appeared later. More research into the influence of dashes on the burning process of a graph could help creating a more accurate distinction between different burning sequences.

In the chapter on the cooling number we developed several complementary approaches. First, we formulated an integer linear program that can compute the cooling number of a graph

exactly. Because ILPs quickly become time-consuming for larger instances, we also proposed a simple 2-approximation algorithm that gives a guaranteed upper bound much faster. We stated a conjecture about the clustering of cooling sources in spider graph with three legs. We did not manage to completely prove this conjecture, however we did manage to prove that at least all the sources on one leg can be clustered at the start of the cooling sequence. Future work could focus on improving the efficiency of the ILP or tightening the approximation factor. Additionally, proving the Conjecture 4 is a nice result and could help to give insights on the cooling sequence for more graphs.

References

[1] Paul Bastide et al. *Improved pyrotechnics : Closer to the burning graph conjecture*. 2022. arXiv: 2110.10530 [math.CO]. url: https://arxiv.org/abs/2110.10530.

[2] A. Bonato, J. Janssen, and E. Roshanbin. *Burning a Graph as a Model of Social Contagion*. 2014. url: https://doi.org/10.1007/978-3-319-13123-8_2.

[3] Anthony Bonato, Jeannette Janssen, and Elham Roshanbin. *Burning a Graph is Hard*. 2015. arXiv: 1511.06774 [math.CO]. url: https://arxiv.org/abs/1511.06774.

[4] Anthony Bonato et al. *How to cool a graph*. Jan. 2024. url: https://arxiv.org/abs/2401.03496.

[5] L. Danon et al. "Networks and the Epidemiology of Infectious Disease". In: *Interdisciplinary Perspectives on Infectious Diseases* (2011). Accessed: 2025-09-15. doi: 10.1155/2011/284909. url: https://doi.org/10.1155/2011/284909.

[6] Subham Datta. *Graph Density*. https://www.baeldung.com/cs/graph-density. 2024. url: https://www.baeldung.com/cs/graph-density.

[7] J. García-Díaz, J. Cornejo-Acosta, and J. Sánchez. *A greedy heuristic for graph burning*. 2024. arXiv: 2401.07577 [cs.DM]. url: https://arxiv.org/abs/2401.07577.

[8] Gurobi. *GUROBI OPTIMIZATION*. 2025. url: https://docs.gurobi.com/projects/optimizer/en/current/features/multiobjective.html (visited on 02/21/2025).

[9] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[10] Lalla Mouatadid. *Introduction to Approximation Algorithms*. https://www.cs.toronto.edu/~lalla/373s16/notes/Intro2Approx.pdf. Accessed: 2025-06-05. 2016.

[11] Yukihiro Murakami. *The Graph Burning Conjecture is true for trees without degree-2 vertices*. 2023. arXiv: 2312.13972 [math.CO]. url: https://arxiv.org/abs/2312.13972.

[12] Jiajun Ning, Xian'an Jin, and Meiqiao Zhang. *The burning number conjecture holds for trees of order n with at most $\lfloor\sqrt{n-1}\rfloor$ degree-2 vertices*. 2025. arXiv: 2509.03144 [math.CO]. url: https://arxiv.org/abs/2509.03144.

[13] Abdullah O, Arif E, and Firas Fawzi. "Dividing Graceful Labeling of Certain Tree Graphs". In: *Tikrit Journal of Pure Science* 25 (Aug. 2020), pp. 123–126. doi: 10.25130/tjps.v25i4.281.

[14] N. Privault. *Understanding Markov Chains: Examples and Applications*. Springer, 2018. isbn: 978-981-13-0659-4.

[15] Elham Roshanbin, Jeannette Janssen, and Anthony Bonato. "How to Burn a Graph". In: *Internet Mathematics* 12.1 (Mar. 2016). doi: 10.1080/15427951.2015.1103339.

[16] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *AAAI*. 2015. url: https://networkrepository.com.

[17] M. Šimon et al. “Heuristics for Spreading Alarm throughout a Network”. In: *Applied Sciences* 9.16 (2019), p. 3269. doi: 10.3390/app9163269. url: https://doi.org/10.3390/app9163269.

[18] Merel Susanna. *Burning Number*. https://github.com/MerelSusanna/BurningNumber. GitHub repository. 2025.

[19] Delft University of Technology. *What Is Citizen Science?* https://www.tudelft.nl/en/citizen-science/what-is-citizen-science/. Accessed: 2025□08□04. n.d.

[20] University of Washington Center for Game Science and Department of Biochemistry. *Foldit*. https://fold.it. Accessed: 2025□08□04. n.d.

A

Algorithm expected number of rounds to burn the entire graph

```
def transition_probabilities(G, p):
    n = G.number_of_nodes()
    nodes = G.nodes()
    allStates = []

    for i in range(n):
        for subset in itertools.combinations(nodes, i+1):
            allStates.append(tuple(sorted(subset)))

    evolvedStates = {}
    for subset in allStates:
        evolution = {frozenset(subset)}

        options = {neighbor for node in subset for neighbor in G.neighbors(node)}
            - set(subset)
        subsets_options = []
        for i in range(len(options)):
            for opt_subset in itertools.combinations(options, i+1):
                subsets_options.append(opt_subset)
        for option in subsets_options:
            evolution.add(tuple(sorted(set(subset).union(option))))

        evolvedStates[subset] = evolution

    evolvedProb = {}
    for key, value in evolvedStates.items():
        frozen_key = frozenset(key)
        for subset in value:
            target_set = frozenset(subset)
            new_key = (tuple(sorted(frozen_key)), tuple(sorted(target_set)))

            added_nodes = target_set - frozen_key
            mult1 = 1
            mult2 = 1

            for node in added_nodes:
                neighbors_node = set(G.neighbors(node))
                burning_neighbors_node = neighbors_node.intersection(frozen_key)
                mult1 *= 1 - (1-p)**len(burning_neighbors_node)
```

```
39
40              not_burned_nodes = G.nodes() - target_set
41              for node in not_burned_nodes:
42                  neighbors_node = set(G.neighbors(node))
43                  burning_neighbors_node = neighbors_node.intersection(frozen_key)
44                  mult2 *= (1-p)**len(burning_neighbors_node)
45
46              evolvedProb[new_key] = mult1 * mult2
47
48      return evolvedProb, allStates
49
50  def transition_matrix(evaluation_prob, allStates):
51      allStates = [tuple(sorted(s)) for s in allStates]
52      stateIndex = {state: index for index, state in enumerate(allStates)}
53      matrix = np.zeros((len(allStates), len(allStates)))
54
55      for (row_state, col_state), prob in evaluation_prob.items():
56          r = tuple(sorted(row_state))
57          c = tuple(sorted(col_state))
58          if r in stateIndex and c in stateIndex:
59              matrix[stateIndex[r], stateIndex[c]] = prob
60
61      return matrix
62
63  def addition_matrix(evaluation_prob, allStates, a):
64      stateIndex = {state: i for i, state in enumerate(allStates)}
65      matrix = np.zeros((len(allStates), len(allStates)))
66
67      for state in allStates:
68              new_state = tuple(sorted(set(state) | {a}))
69              if new_state in stateIndex:
70                  matrix[stateIndex[state], stateIndex[new_state]] = 1
71
72      return matrix
73
74  def exact_prob(G, sequence, p):
75      evaluation_prob, all_states = transition_probabilities(G, p)
76      T = transition_matrix(evaluation_prob, all_states)
77      burned_nodes = set()
78      burned_nodes.add(sequence[0])
79      burned_nodes_tuple = tuple(sorted(burned_nodes))
80
81      stateIndex = {state: index for index, state in enumerate(all_states)}
82      initial_vector = np.zeros(len(all_states))
83      initial_vector[stateIndex[burned_nodes_tuple]] = 1
84      initial_vector = initial_vector.reshape(1, len(all_states))
85      total_set_index = stateIndex.get(tuple(G.nodes()))
86
87      expectedNumberRounds_dict = {i: 0 for i in range(1, G.number_of_nodes() + 1)}
88      expectedNumberRounds_dict[1] = initial_vector[0, total_set_index].item()
89
90      product = initial_vector.copy()
91      for i in range(2, G.number_of_nodes() + 1):
92          M = addition_matrix(evaluation_prob, all_states, sequence[i - 1])
93
94          extra = np.matmul(product, T)
95          product = np.matmul(extra, M)
96
97          expectedNumberRounds_dict[i] = product[0, total_set_index].item()
98
99      cumulative = 0
```

```
100     for key in expectedNumberRounds_dict.keys():
101         expectedNumberRounds_dict[key] -= cumulative
102         cumulative += expectedNumberRounds_dict[key]
103 
104     expectedRounds = 0
105     for key, value in expectedNumberRounds_dict.items():
106         expectedRounds += key * value
107 
108     return expectedRounds
```

B

Cooling number

B.1. Cooling ILP

```
G = create_graph(inputfile)
n = G.number_of_nodes()
U = math.ceil((n+1)/2)

m = Model("Graph␣Cooling␣Problem␣ILP")

#Variables
y = [m.addVar(vtype=GRB.BINARY, name=f"y_{j}") for j in range(U)]
z = [m.addVar(vtype=GRB.BINARY, name=f"z_{j}") for j in range(U)]
s = [[m.addVar(vtype=GRB.BINARY, name=f"s_{i}_{j}") for j in range(U)] for i in
    range(n)]
c_startround = [[m.addVar(vtype=GRB.BINARY, name=f"cstartround_{i}_{j}") for j in
    range(U)] for i in range(n)]
c_afterspread = [[m.addVar(vtype=GRB.BINARY, name=f"caftersprea_{i}_{j}") for j
    in range(U)] for i in range(n)]
c_aftersource = [[m.addVar(vtype=GRB.BINARY, name=f"caftersource_{i}_{j}") for j
    in range(U)] for i in range(n)]


# 5.1
m.setObjective(quicksum(y[j] for j in range(U)), GRB.MAXIMIZE)

# Constraints
# 5.2
m.addConstr(quicksum(s[i][0] for i in range(n)) == 1)

# 5.3 and 5.4
for j in range(U):
    for i in range(n):
        m.addConstr(s[i][j] <= 1 - c_afterspread[i][j])
        m.addConstr(s[i][j] <= c_aftersource[i][j])

# 5.5 and 5.6
for i in range(n):
    for j in range(1, U):
        m.addConstr(c_afterspread[i][j] >= c_startround[i][j])
        m.addConstr(c_aftersource[i][j] >= c_afterspread[i][j])

# 5.7
for j in range(1, U):
```

```
    for i in range(n):
        m.addConstr(quicksum(s[i][j] for i in range(n)) >= 1 - c_afterspread[i][j
            ])

# 5.8
for j in range(U):
    m.addConstr(quicksum(s[i][j] for i in range(n)) <= 1)

# 5.9
for j in range(1, U):
    m.addConstr(z[j] >= quicksum(c_aftersource[i][j - 1] for i in range(n)) - n +
        1)

# 5.10 and 5.11
for j in range(1, U):
    for i in range(n):
        neighbors = list(G.neighbors(i))
        neighbors.append(i)
        for k in neighbors:
            m.addConstr(c_afterspread[i][j] >= c_aftersource[k][j-1])
            m.addConstr(c_afterspread[i][j] >= s[k][j-1])

# 5.12
for j in range(U):
    m.addConstr(y[j] + z[j] == 1)

m.optimize()
```

B.2. Approximation algorithm

```
def approx_algorithm(G):
    diameter, diameter_paths = diameter_and_path(G)
    path = random.choice(diameter_paths)

    cooled_nodes = set()
    sources = []
    rounds = 0
    end = path[-1]

    distances = nx.single_source_shortest_path_length(G, end)

    while len(cooled_nodes) < G.number_of_nodes():
        # Spreading
        uncooled_nodes =set(G.nodes()) - cooled_nodes
        cooled_nodes_copy = cooled_nodes.copy()
        for node in uncooled_nodes:
            neighbors = G.neighbors(node)
            if len(set(neighbors).intersection(cooled_nodes_copy)) > 0:
                cooled_nodes.add(node)

        if rounds == 0:
            source = path[0]
        else:
            uncooled_distances = {node: dist for node, dist in distances.items()
                if node not in cooled_nodes}
            if uncooled_distances:
                max_value = max(uncooled_distances.values())
                max_distance_end = [node for node, value in uncooled_distances.
                    items() if value == max_value]
                possible_sources_off_diagonal = [node for node in max_distance_end
                     if node not in path]
```

```
                if possible_sources_off_diagonal:
                    source = random.choice(possible_sources_off_diagonal)
                else:
                    source = random.choice(max_distance_end)
                cooled_nodes.add(source)
            else:
                source = None  # all nodes cooled, no source needed

        cooled_nodes.add(source)
        sources.append(source)
        rounds += 1

    return path, rounds, sources
```

B.3. Find diameter and diameter paths

```
def diameter_and_path(G):
    all_lengths = dict(nx.all_pairs_shortest_path_length(G))
    diameter = nx.diameter(G)
    diam_pairs = []

    for u in G.nodes():
        for v in G.nodes():
            d = all_lengths[u][v]
            if d == diameter:
                diam_pairs.append((u, v))

    paths = []
    for u, v in diam_pairs:
        path = nx.shortest_path(G, u, v)
        paths.append(path)

    return diameter, paths
```