Utrecht University

Masterthesis Computing Science

Using State Evaluation and Artificial Neural Networks to solve the Firefighter Problem

Author
Nikè Lambooy
Studentnumber
4090349

Supervisors
Dr. Erik Jan van Leeuwen
Prof. dr. Hans L. Bodlaender
Daily supervisor
Jelle J. Oostveen, MSc.

July 20, 2022

Abstract

The Firefighter Problem (FFP) on graphs is a model for fighting the spread of a fire through a city. At each time step d nodes can be defended from the fire, before the fire spreads to all undefended nodes adjacent to a burning node. In this thesis we investigate the application of State Evaluation to this problem, creating an ANN called SEANN for this purpose. We also describe a second neural network, called CLANN that solves FFP more directly by classifying which node should be protected at the current time step. To solve the FFP we created several solvers that use some form of State Evaluation, and we compare these to the optimal solution found by solving an ILP model and to a greedy algorithm in case of trees. Our experiments show that State Evaluation works very well for FFP, especially a Greedy State Evaluation algorithm that uses a Greedy Look-ahead algorithm to evaluate potential future states. The CLANN network, however, performs worse than basic greedy heuristics.

Code at: https://git.science.uu.nl/n.l.c.lambooij/firefighter

1 Introduction

The Firefighter Problem (FFP) on graphs, originally proposed by Hartnell in 1995 [13], is a model for the spread of fire through a network. Each time step the fire spreads further, while nodes of the network can be defended by firefighters. Another interpretation of the problem is that it can model the spread of a virus through a population, and this virus can be stopped from infecting new hosts by vaccinating them. The goal is for the fire (or the virus) to consume as few points of the network as possible. Solving this problem can give insight in situations where firefighters or vaccinations are in limited supply, when it is important to most effectively vaccinate people so as to stop the spread of the virus.

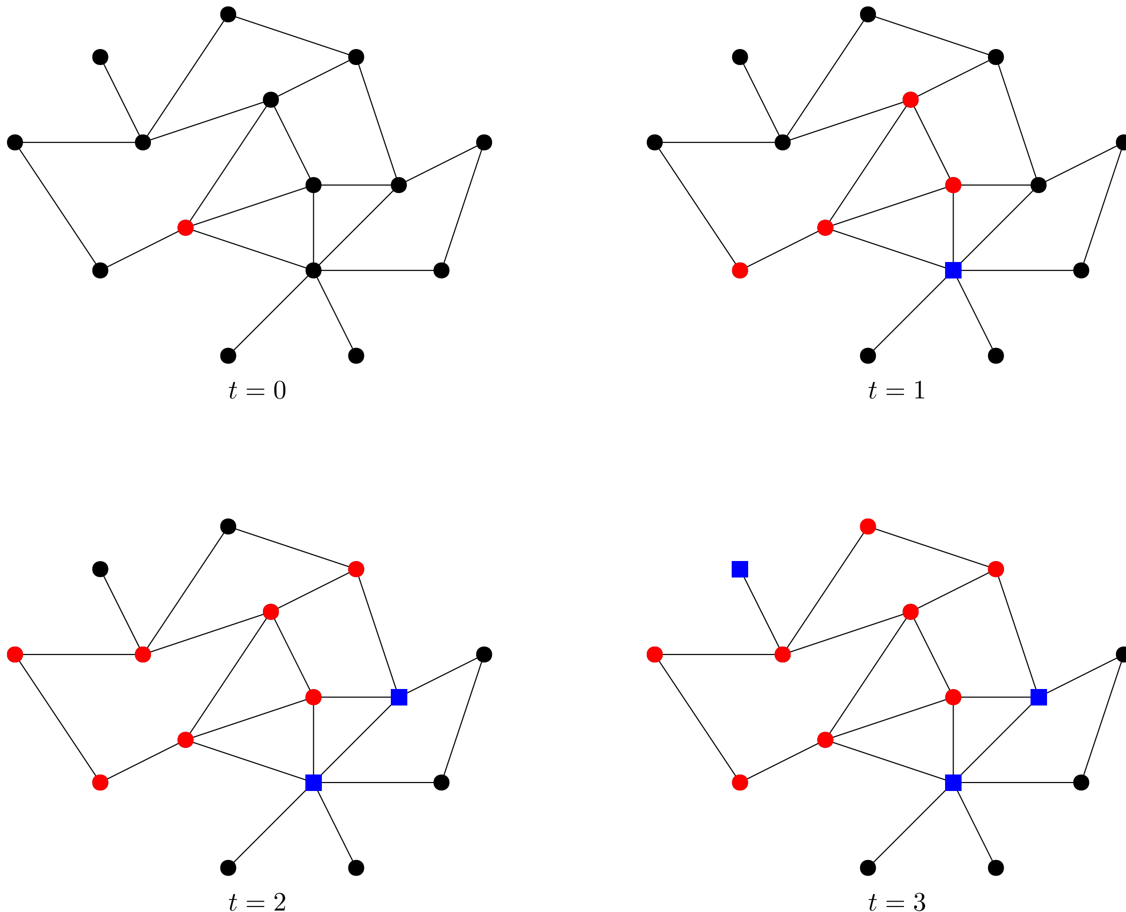

Figure 1: The FFP on a graph, from time step $t = 0$ until the fire can no longer spread. Red nodes are burnt, blue nodes are defended

A formal definition of FFP is the following: Given are a graph $G = (V, E)$, and a set $B \subset V$ of burning vertices at time step 0. Often, $|B| = 1$. All other vertices are *untouched*. At each time step, d untouched vertices can be defended (often, d=1), after which the fire spreads to all untouched vertices adjacent to a burning vertex. This continues until the fire has burnt out: there are no more untouched vertices neighbouring a burning vertex. All vertices that are not burnt, including the defended vertices, are *saved*. An example of FFP playing out on a graph with 15 vertices can be seen in Figure 1.

The objective is generally to save as many vertices as possible, i.e. ensure the set of burning (or burnt) vertices is as small as possible. Other variations of the problem include the weighted version, where vertices have weights that correspond with the priority of saving them, and the same problem on trees, where the root is the single initial burning vertex. For trees, the main objective is also to save as many vertices as possible, but the aim can be to save as many leaves as possible as well. Another variation is the Multi-Objective Firefighter Problem [17], where the weight of a vertex is a vector representing the multiple objectives. For example, in a graph where the nodes represent buildings that need to be saved from a spreading fire, a building can have a high social value, but a low historical value, or vice versa.

The Firefighter Problem is NP-Complete, for, amongst others, cubic graphs [16], and trees with maximum degree three with a root of degree three [9]. The problem is easy for trees with a root of degree two (and trivial with a root of degree one), and these instances can be solved in polynomial time.

The Firefighter Problem can be interpreted as a game, where the player chooses vertices to be protected, attempting to save as many as possible. Following this line of thought, it is an interesting idea to apply Game AI to the problem. For some of the most complex games such as Chess or Go, AI are consistently defeating the best players in the world [24]. Applying some of the same techniques may therefore result in solutions that are very close to optimal. The most fundamental difference to these games and the FFP, however, is that at every time step of the FFP everything is fully known: it is a single player game with full information.

One of the Game AI techniques used for (board)games is State Evaluation, which assigns a value (generally between 0 and 1) to a state, representing how desirable it is for the (current) player. Here, a state is a game state, which ranges from a board with game pieces on it, such as Chess, Checkers or Go, to an agent in an environment that has as objective to move towards some kind of goal. The closer the agent or player is to that goal, the higher the value of the current state. The value of a state can also be calculated with probabilities: how likely is the player to win? In chess, for example, if approximately 54% of matches are won by white when they use a particular starting move, the resulting state has a value of 0.54 for white (and 0.46 for black). The values assigned to potential next states can, among other things, be used to prune the decision tree of the game, or as a heuristic in another algorithm.

For the Firefighter Problem, State Evaluation can be used to find the next best move, or to exclude the worst moves from the search. Essentially, any heuristic does some form of State Evaluation, such as a greedy heuristic that bases the value of a state only on the amount of burnt vertices, or something much more complex that looks ahead multiple steps. When evaluating a state based on possible solution strategies, there are multiple values that can be used: the worst possible outcome from this state, the best, a valuation function that takes into account how many untouched vertices are still present in the graph, or some aggregate of the results of the above. Another well-used method for State Evaluation is applying an Artificial Neural Network to learn the value of a state [25, 21, 26], and these state evaluations may also be used to train a different neural network to find a next move, using the evaluation of the resulting state as feedback.

The above is precisely what we intend to do in this thesis: investigate how effective State Evaluation using Artificial Neural Networks is for FFP. How we proceed to build the neural networks is described in Section 4, as well as how we evaluate performance compared to methods that yield optimal solutions. First we review some preliminary information on Artificial Neural Networks in Section 2, and explore previous research on the FFP and on ANNs in Section 3. In Section 4 we discuss how we acquired our data, the methods we use to evaluate our states, the algorithms we employ to solve FFP, and we describe two Artificial Neural Networks: one for State Evaluation and one used as a solver for FFP. We will conduct several experiments to test the viability of using State Evaluation to help solve the FFP. We analyse the results of our experiments in Section 5. Finally we will draw conclusions and give some ideas for future work in Sections 6 and 7.

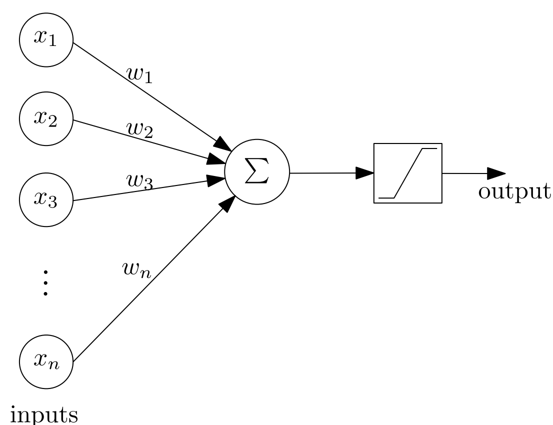

Figure 2: A perceptron with a relu activation function

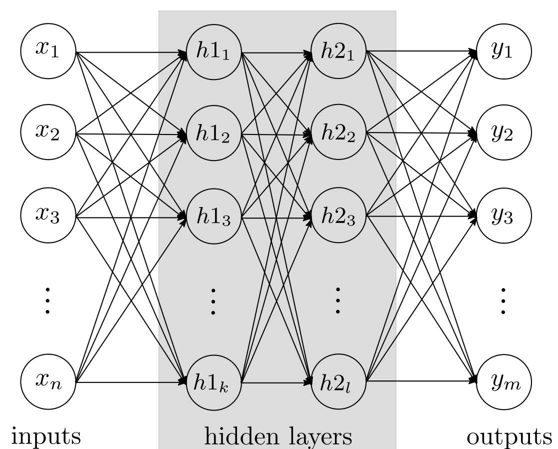

Figure 3: A fully connected network architecture with two hidden layers

2 Preliminaries

2.1 Artificial Neural Networks

In this section we explain how Artificial Neural Networks are built, and how they learn. For more information, see [22].

An Artificial Neural Network (ANN) is built from smaller components called a node or a perceptron. Such a perceptron has one or more numerical inputs, generally between 0 and 1, which are weighted. These weights are also values between 0 and 1, and generally give an indication of the importance of the input value. These weighted inputs are then added together, and if this value exceeds a set threshold, the neuron outputs 1, and otherwise 0. Instead of a threshold, a sigmoid or relu function can also be used, in which case the output is some value between 0 and 1. A schematic image of a perceptron can be seen in Figure 2.

The ANN consists of multiple layers of these perceptrons: First an input layer, the nodes of which each have a single input value, and no adjustable weight (the weight is 1). This input layer is the size of the actual input, for example, the amount of pixels of an image. Next comes some number of hidden layers, generally at least one. In fully connected networks all input nodes are connected to all nodes of the second layer, and all nodes of layer i are connected to all nodes of layer $i+1$. The last hidden layer of the network is fully connected to the output layer. This output layer can consist of a single node, or multiple nodes, each outputting a value between 0 and 1. These nodes represent the shape of the output in some way, for example, if an ANN is meant to classify handwritten digits 0-9, the output layer would have 10 output nodes. A diagram of this is visible in Figure 3.

The purpose of the network is to give the 'correct' output, given some input - whatever that may be. To be able to do this, the network needs to train, that is, adjust its weights such that feeding an input into the network results in the correct output. The training happens by giving the network training data: many examples together with the correct answer. In a classification network with the task of recognising the digits 0-9 this means that given a picture of a number 5, the fifth output node ideally outputs 1, and the rest of the nodes output 0. The output of the network is correct if the fifth node has the highest output. This does not necessarily have to be 1. The next step is to calculate the error: how far off was the network with its guess? This error can be the Mean Squared Error (MSE) or something like the Sparse Categorical Crossentropy, and is given as feedback to the network. Using the error and the learning rate, the weights are updated with backpropagation: The weights leading to the output node are updated first, then the weights in the layer before that, etcetera. The new weight is calculated according to the following formula:

$$w'_{ij} = w_{ij} + \eta\delta_j$$

where w_{ij} is the current weight between nodes i and j, w'_{ij} is the new weight, η is the learning rate, δ_j is the error of node j. This means that the weight is adjusted more when the error is higher, and less when the error is low. The learning rate also influences how much the weight changes, with a higher learning rate allowing for quicker learning, but a lower learning rate resulting in more precise adjustments.

The above method is a gradient descent method of optimizing the weights of the network. Gradient descent methods work very well as optimizers for ANNs: they adjust the weights of the network in small steps in an attempt to find an optimal value. When this optimal value is found, the error has been minimized, hence gradient *descent*. Stochastic Gradient Descent (SGD) [2] is an adaptation where not the entire data set is used to calculate the gradient, but a random subset. This strongly reduces computation time, and most other optimizers are based on it, such as those offered by the Tensorflow Keras package that we use in our experiments.

In more complex networks not all layers are fully interconnected, which may save training time. The nodes that are connected tend to have some problem-specific reasoning behind it, often with multiple hidden layers to aggregate the information from earlier layers. The output layer can have multiple nodes, or just a single node. When classifying, the output layer has as many nodes as there are 'options', e.g. 10 nodes when we are trying to recognize single digit handwritten numbers. Another alternative is to have a single output node, which outputs a value between 0 and 1. This value describes something about the input, such as the value of the current state, hence the use of ANNs for State Evaluation. An ANN can also be pruned [4], whereby a percentage of the weights are set to 0. This is a technique to combat overfitting, and will also decrease the amount of time necessary for training. Overfitting is what happens when the network fits very well on the training data, but performance on the test data is going down. The network is learning how to deal with the individual cases of the training data, rather than general rules applicable to all inputs.

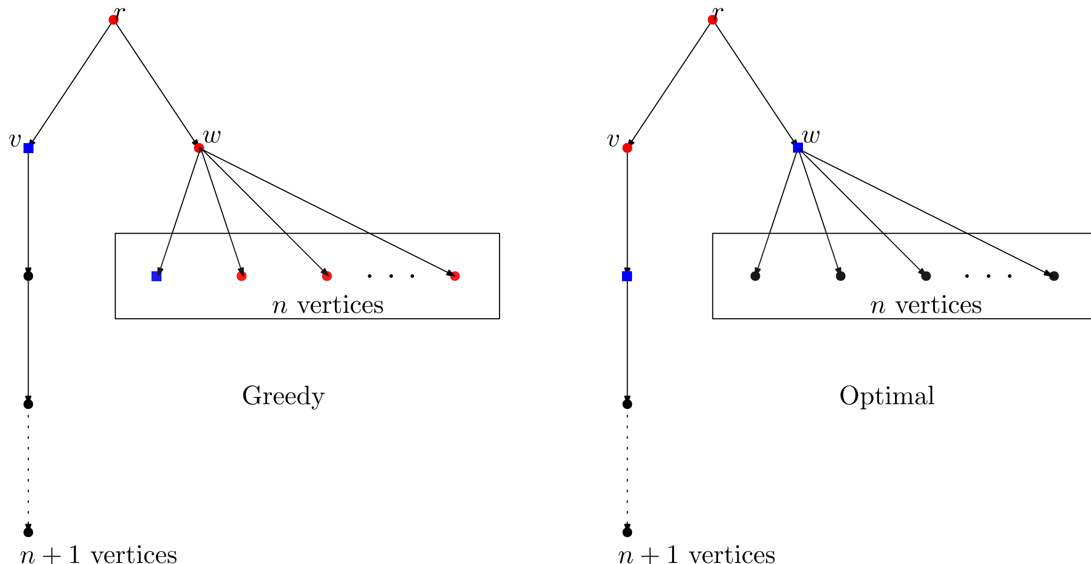

Figure 4: Optimal strategy vs. greedy strategy

3 Previous Work

3.1 Firefighting on Trees

For the Firefighter Problem on trees, it is generally assumed that the root of the tree is the single burning vertex at the start. At each progressive time step, the fire burns one layer deeper than the previous, continuing until no more vertices can be burnt. For the Firefighter Problem on trees, it is always optimal to save nodes that are adjacent to burning nodes, as any node that is not adjacent to a burning node can be saved by protecting its ancestor that is adjacent to the fire.

The Firefighter Problem was first suggested by B. Hartnell [13], who subsequently investigated how well a simple greedy algorithm (always protect the vertex with the most descendants) performed on trees [14]. He proves that this strategy saves at least half as many nodes as an optimal strategy, and that this bound is tight. The proof is constructed as follows:

In the first move, the greedy algorithm saves at least as many vertices as the optimal move, since the greedy algorithm saves as many vertices as possible. At any subsequent time step, either the above is the case again, or the optimal move protects more vertices because the greedy algorithm already defended an ancestor of the optimal vertex.

Define S_A to be the set of vertices saved by the optimal moves on which the greedy move saves at least as many vertices as the optimal move. Define *Greedy* as the set of nodes saved by the Greedy algorithm. It follows that $|Greedy| \geq |S_A|$. Define S_B as the set $S_{opt} \setminus S_A$ where S_{opt} is the set of nodes saved by making only optimal moves.

As the moves resulting in S_B are optimal, the greedy algorithm would choose those moves if they were available. If the greedy algorithm cannot choose the optimal vertex to save, it saves fewer vertices on its corresponding move. The only reason for a move not to be available is for an ancestor to have already been defended. Thus, the vertices of set S_B have already been saved by the greedy algorithm. It follows that $|Greedy| \geq |S_B|$. From here we can conclude: $|Greedy| + |Greedy| \geq |S_A| + |S_B| \Leftrightarrow 2|Greedy| \geq |S_{opt}| \Leftrightarrow |Greedy| \geq \frac{1}{2}|S_{opt}|$.

The proof of the tightness of this bound shows a worst-case scenario, see also Figure 4: a root node r is burning, and one of its two children, v and w must be protected. w has n children, all leaves, and thus n descendants. v has one single child, which has one single child, etcetera, forming a chain of $n+1$ vertices, excluding v, giving vertex v $n+1$ descendants. This means that v will be protected by the greedy algorithm. In the next step, vertex w will burn, after which only one of the children of w can be protected. This strategy therefore saves $n+2$ vertices in total. The optimal strategy, however, defends vertex w first, and in the next time step it will defend the single child of v. This saves $2n$ vertices in total, making this a fraction $\frac{n+2}{2n}$ of optimal, which for large numbers approaches $\frac{1}{2}$ [14].

Since then, both exact approaches and approximation algorithms have been investigated. An incredibly useful resource for this is the recent survey by Wagner [27], which expounds on what has been happening in the field since the last survey by Finbow and MacGillivray [10]. This 2021 survey touches

on the results of the research done on the complexity of FFP, discusses approximation algorithms, Fixed Parameter Tractability (FPT) of FFP, heuristic approaches, surviving rates of graphs, and variants of the problem.

In their 2008 paper, Cai et al. [6] study the Firefighter Problem on trees, and present a $(1 - 1/e)$-approximation algorithm. They base their approach on the LP relaxation of the problem, giving the nodes fractions of firefighters. They then give each node a probability of being saved, based on these fractions and using randomized rounding. They also give several FPT algorithms.

Bazgan et al. [3] continue in the line of Cai et al., and investigate parameterizations of the Firefighter Problem on general graphs and trees where $d \geq 1$ nodes can be defended per time step. They investigate the following parameterizations: saving k vertices, protecting k vertices, and saving all but k vertices. Their results show that for each of these cases the problem is W[1]-hard.

3.2 Population Based Search methods

The most common representation of a solution to FFP is represented as a permutation of nodes, the order of which determines which node gets protected when. In population based search methods the population consists of many such lists of nodes, and both simple and more complex mutation and crossover operators can be executed on them. As the entire solution space is incredibly large, we need some clever way to search this space. One such ways is to apply Ant Colony Optimization, as Blum et al. do in their 2014 paper [5]. Their algorithm starts with many initial solutions and improves upon those by using heuristic information. This approach is subsequently hybridized with CPLEX, by giving CPLEX the result from ACO as initial solution. This yields better results than either CPLEX or ACO on their own.

In his papers [17], [18], [19] and [20], Krzysztof Michalak focuses mainly on solving the multi-objective Firefighter Problem by using evolutionary algorithms. The genetic operators used in [17] are designed to generate list permutations, ensuring that no node can occur twice in the same solution. He investigates the effectiveness of the operators compared to each other, using auto-adaptation: Operators that perform well get a higher and higher probability of being used. He continues his line of research with the Sim-EA Algorithm in [18], again using Operator Auto-adaptations for the Multi-Objective Firefighter Problem. The way the Sim-EA Algorithm differs from the previous evolutionary algorithm is that the Sim-EA algorithm makes use of multiple populations of specimens between which specimens might migrate to exchange information, instead of a single large population. This algorithm is then compared to the MOEA/D algorithm, which is a different evolutionary algorithm.

A third paper [19] by the same author investigates the performance of a heuristic local search algorithm, ED-LS, on the multi-objective Firefighter Problem. The approach is hybridized with Evolutionary Algorithms, using ED-LS to improve upon the solutions found by the EA.

3.3 Using Artificial Neural Networks for State Evaluation in games

In their 2009 paper, Tomizawa et al. [26] use State Evaluation for the game Go, applying an Artificial Neural Network, which they have called 'TG361G'. The network gives an expected winning probability for either black or white, and is trained using game states from human expert's game records, with as feedback the actual result of the game (1 for winning, 0 for losing). The resulting expected winning probability can then be used to find a good next move. The architecture of the network makes use of local patterns of stones that can be good or bad, and each hidden unit is assumed to have its receptive 4×4 patch on the go board (with overlapping patches). This will reduce the number of weights that would have to be learned if the network were fully connected. The resulting network has a 95% accuracy, using a training data set of about 600 000 examples.

Artificial Neural Networks for State Evaluation are also used for General Game Playing, which is concerned with playing classes of games, instead of one specific game. This means that no game-specific information can be used to train or tweak the Neural Network. In the paper 'Neural Networks for State Evaluation in General Game Playing' by Daniel Michulke and Michael Thielscher [21], they initialize the ANN in such a way that it is correctly able to classify goal conditions without training. This is then used to identify the similarity of non-terminal states to goal states, which gives a value to those states. Their method works very well for individual games, but when applied to general games performance goes down, which indicates overfitting.

3.3.1 Integer Linear Programming

A different way to represent the Firefighting Problem is to model it as an ILP, as Develin and Hartke do in their 2007 paper 'Fire containment in grids of dimension three and higher' [7]. It is more comprehensively described in Blum et al. [5] as follows:

ILP_{FFP} :

$$b_{v,t} = \begin{cases} 1 & \text{if vertex } v \in V \text{ at time step } t \text{ where } 0 \leq t \leq T \text{ is burnt} \\ 0 & \text{otherwise} \end{cases}$$

$$d_{v,t} = \begin{cases} 1 & \text{if vertex } v \in V \text{ at time step } t \text{ where } 0 \leq t \leq T \text{ is defended} \\ 0 & \text{otherwise} \end{cases}$$

$$\max |V| - \sum_{v \in V} b_{v,T}$$

subject to

$$b_{v,t} + d_{v,t} - b_{v',t-1} \geq 0 \; \forall v \in V,\, v' \in N(v) \text{ and } 1 \leq t \leq T \tag{1}$$
$$b_{v,t} + d_{v,t} \leq 1 \; \forall v \in V \text{ and } 1 \leq t \leq T \tag{2}$$
$$b_{v,t} - b_{v,t-1} \geq 0 \; \forall v \in V \text{ and } 1 \leq t \leq T \tag{3}$$
$$d_{v,t} - d_{v,t-1} \geq 0 \; \forall v \in V \text{ and } 1 \leq t \leq T \tag{4}$$
$$\sum_{v \in V} (d_{v,t} - d_{v,t-1}) \leq D \; \forall 1 \leq t \leq T \tag{5}$$
$$b_{v,0} = 1 \; \forall v \in B_{init} \tag{6}$$
$$b_{v,0} = 0 \; \forall v \in V \backslash B_{init} \tag{7}$$
$$d_{v,0} = 0 \; \forall v \in V \tag{8}$$
$$b_{v,t}, d_{v,t} \in 0, 1 \; \forall v \in V \text{ and } 1 \leq t \leq T \tag{9}$$

Here, T denotes the upper bound for the amount of time steps until the fire burns out, $N(v)$ denotes the set of vertices adjacent to v, D denotes the amount of firefighters that can be used per time step, and B_{init} denotes the set of vertices that is burning at time step 0. Constraint 1 ensures the burning of vertices and constraint 2 ensures that a burning vertex cannot be defended and vice versa. Constraints 3 and 4 ensure that a burnt vertex and a defended vertex stay that way, respectively. Constraint 5 limits the amount of defended vertices per time step to D. Constraints 6, 7 and 8 all describe the initial conditions of the graph: the vertices of set B_{init} are burnt, and no others, and none of the vertices are defended. Finally, constraint 9 ensures that the variables $b_{v,t}$ and $d_{v,t}$ have either value 0 or 1. With this model, an ILP-solver like CPLEX [15] or Gurobi [11] can be used to solve the problem.

In our experiments, we have included one additional constraint:

$$\sum_{v \in V} d_{v,t} - d_{v,t-1} \leq \sum_{v \in V} b_{v,t-1} - b_{v,t-2} \text{ for } 0 \leq t \leq T \tag{10}$$

This constraint ensures that no more vertices are defended after the graph is burnt out. This is not a necessity to ensure correctness of solutions, but to know how many time steps the graph took to burn out. This was especially useful for figuring out how varied our generated graphs and trees are.

4 Methods

The main area of research of this thesis consists of two aspects: State Evaluation for the Firefighter Problem, and several ways of solving the Firefighter Problem using State Evaluation.

4.1 Hardware and software

The hardware used for all experiments is a desktop computer, with a Intel(R) Core(TM) i5-8600K CPU @ 3.60 GHz with 16GB RAM and a NVIDIA GeForce GTX 1070 graphics card. Our experiments are coded in Python, using the gurobipy package from Gurobi [12] for the solving of ILPs, and using the Tensorflow [1] Keras packages to construct, train and use two ANNs.

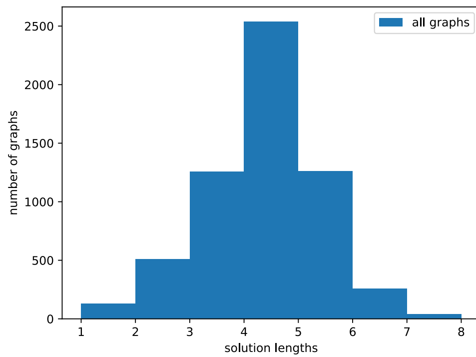

Figure 5: Solution lengths of optimal solutions of graphs of 25 nodes

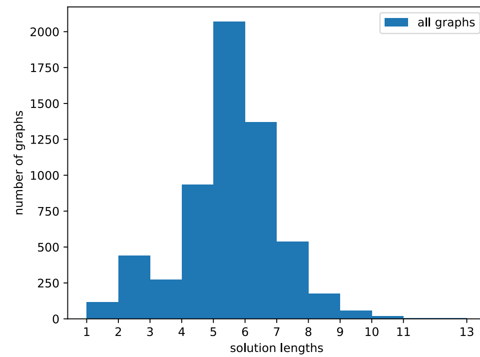

Figure 6: Solution lengths of optimal solutions of graphs of 50 nodes

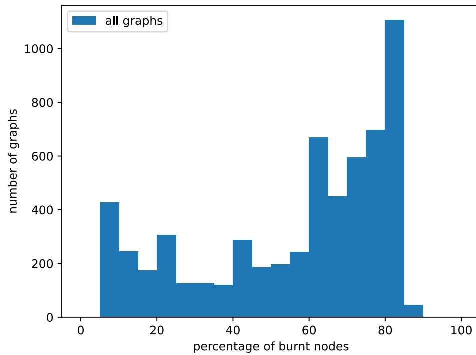

Figure 7: Percentage of burnt nodes of graphs of 25 nodes when solved optimally

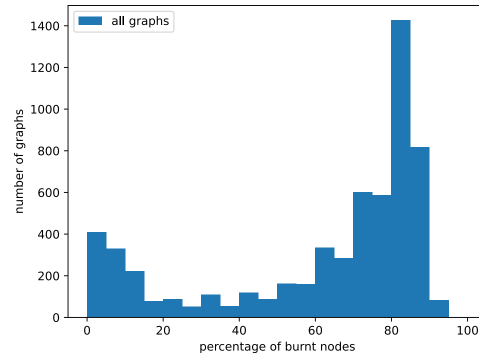

Figure 8: Percentage of burnt nodes of graphs of 50 nodes when solved optimally

4.2 Generating Data

To train and test an ANN, we need data: trees and graphs to learn from and test the ANN with. This data set consists of 30000 randomly generated trees and the same amount of graphs. These 30000 trees and graphs are divided equally in six different types, to ensure a varied data pool. We will generate a separate test set of 6000 trees in the same manner. We will now describe the generated data set in more detail.

4.2.1 Initial Graphs

The generated graphs are Erdös-Renyi graphs, with expected degrees ranging from 2.5 to 5 with a step size of 0.5. This creates a large variety of graphs, where some are not fully connected, and some will be so dense that most nodes will burn very quickly. The initial burning node is chosen at random from the nodes of the graph with degree at least 2. Figures 5 and 6 show a histogram of how many time steps

it takes for the graphs of 25 and 50 nodes to burn out. For most graphs of 25 nodes this takes 4 time steps, and for most graphs of 50 nodes it takes 5 time steps. Figures 7 and 8 show a histogram of the percentage of the vertices of graphs of 25 and 50 nodes burn. Most graphs burn for more than 80%, with another peak at around 10%.

4.2.2 Initial Trees

The first random trees we generated were Prüfer trees [23], choosing a random node with at least three children as root node, which is also the node burning at time step 0. The resulting trees did not provide enough variation. These trees were mostly solvable in two time steps, with very few vertices burning. While it is a good thing to have a portion of trees that burn very little, for those 'easy' instances to make up the entire data set is not representative of trees as a whole. Therefore we also generated trees of n nodes using Algorithm 1.

Algorithm 1: Generating random trees

input : number of nodes n
output: A randomly generated tree

1 Create a tree T with a root with 3 children
2 **for** $i = 4$ *to* $i = n - 1$ **do**
3 | Attach a new node i to one of the existing nodes depending on probability distribution P
4 **return** T

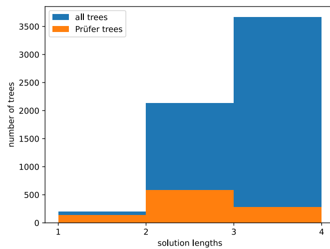

Figure 9: Solution lengths of optimal solutions of trees of 25 nodes

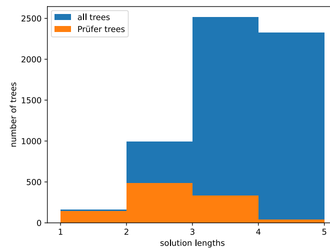

Figure 10: Solution lengths of optimal solutions of trees of 50 nodes

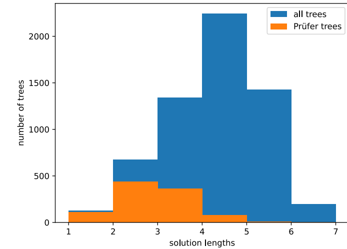

Figure 11: Solution lengths of optimal solutions of trees of 100 nodes

For all of these trees, the root node is the initial burning node. The probability distribution P in line 3 of Algorithm 1 is one of the following, where M is the current set of nodes of the graph, k_m the current degree of node $m \in M$, and d_m the depth of node $m \in M$, where the depth of the root node is 1:

1. $P_m = \frac{1}{|M|}$ $\forall m \in M$. The new nodes are attached to the existing nodes randomly.
2. $P_m = \frac{k_m}{\sum_{v \in M} k_v}$ $\forall m \in M$. The new nodes are attached with a probability proportional to the amount of neighbours an existing node already has. New nodes are thus more likely to be attached to nodes that already have a lot of neighbours.
3. $P_m = \frac{1}{\sum_{v \in M} \frac{k_m}{k_v}}$ $\forall m \in M$. Here the probability of attaching a new node to a node m is reverse proportional to the amount of children an existing node already has, meaning that new nodes are more likely to be attached to nodes that have very few neighbours.
4. $P_m = \frac{d_m}{\sum_{v \in M} d_v}$ $\forall m \in M$. Here the deepest nodes are more likely to receive children, creating a deep tree.
5. $P_m = \frac{1}{\sum_{v \in M} \frac{d_m}{d_v}}$ $\forall m \in M$. Here the nodes closest to the root are more likely to receive children, creating a shallow tree.

This method ensures that there are many varied trees in the data set, with varying solution lengths. Figures 9, 10, and 11 show histograms of the solution lengths of optimal solutions of trees of 25, 50, and

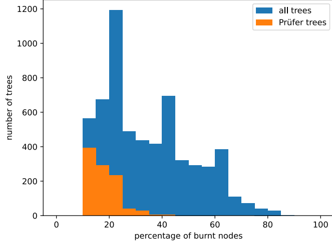

Figure 12: Percentage of burnt nodes of trees of 25 nodes when solved optimally

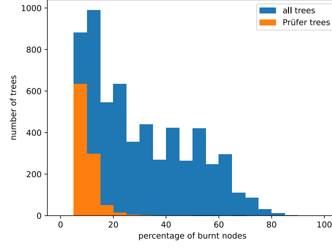

Figure 13: Percentage of burnt nodes of trees of 50 nodes when solved optimally

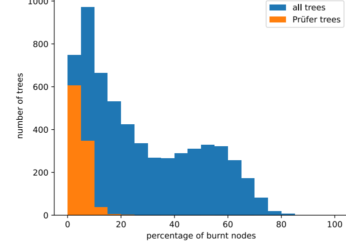

Figure 14: Percentage of burnt nodes of trees of 100 nodes when solved optimally

100 nodes respectively. The difference between Prüfer trees and the combined data set of trees is clear: Prüfer trees are generally solvable in fewer time steps, and are therefore easier to solve at all. This also becomes clear in Figures 12, 13, and 14, which show histograms of the percentage of burnt nodes of trees of 25, 50 and 100 nodes respectively when they are solved optimally. The peak of the 'all trees' graph around 10% is largely caused by the peak of the Prüfer trees around this percentage. This is especially visible for trees with 50 and 100 nodes, where the histograms are much flatter above 30%, showing a far more even division of burnt nodes across the instances.

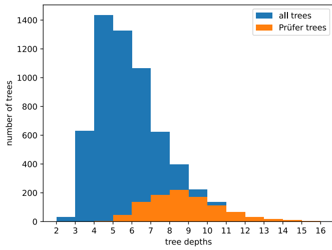

Figure 15: Depth of trees of 25 nodes

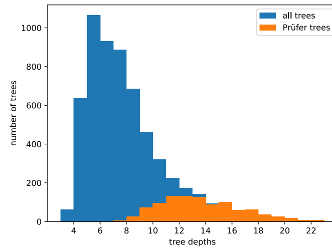

Figure 16: Depth of trees of 50 nodes

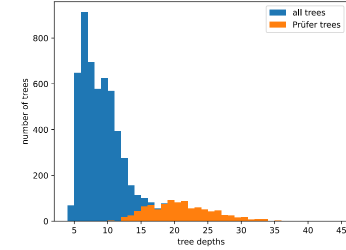

Figure 17: Depth of trees of 100 nodes

The distribution of the depth of the trees can be seen in Figures 15, 16, and 17. The depth distribution of Prüfer trees is much more even compared to that of all trees. This ties in with the low solution lengths: if a tree is very deep, then it does not branch as much. The fire will thus spread to fewer vertices each time step compared to a tree where nodes have many children, and is overall easier to extinguish.

4.2.3 Extending the Graphs and Trees

Having generated the initial graphs and trees, we extend them into (partially) solved graphs and trees, to enable SEANN (see Section 4.4) to learn from sub-optimal and partially solved FFP graphs. We do this according to Algorithm 2. We start with one of the initially generated graphs, and at each time step we defend a node, burn the graph, calculate its current value, and take a snapshot, until the graph is burnt out. To calculate the value of the snapshot, we solve the graph optimally, which gives us the number of nodes burnt in the optimal case: b. We then use the following formula to find the value of the snapshot: $v = (n - b)/n$, where n is the amount of nodes in the graph. This function does not take into account any worst-case scenarios, or how many time steps it would take to reach the optimal solution. We chose this function because it is the objective truth: it is the value of the best possible state reachable from the current state.

The optimal solution used to calculate the true evaluation value is acquired by using Gurobi to solve ILP_{FFP} (see Section 3.3.1). Each snapshot, together with its value, is saved in our database of examples for SEANN to learn from.

In step 5 we choose a node to defend. This node is not necessarily the optimal node, because we want SEANN to learn from graphs that have vastly different values. To this end, we must choose sub-optimal nodes to defend, and for each snapshot this 'node to defend' is chosen in one of the following four ways:

Algorithm 2: Extending generated graphs

input : initial graph G
output: A set of partially solved versions of G, with their value v'_G

```
1  S ← ∅
2  v_G ← the value of graph G
3  S ← S ∪ (G, v_G)
4  while G is not burnt out do
5  |  n ← node to defend
6  |  G' ← G where:
7  |          node n is defended
8  |          all untouched nodes adjacent to a burning node in G, are burnt
9  |  v_G' ← the value of graph G'
10 |  S ← S ∪ (G', v'_G)
11 |  G ← G'
12 return S
```

1. Optimal. Every time step the optimal node is chosen, resulting in the optimal solution. This node is found by solving ILP_{FFP}.

2. Randomly. Every time step a node (that is not yet burnt or defended) is chosen at random to be the next defended node.

3. Highest degree. Every time step the node with the highest degree (that is not yet burnt or defended) is chosen to be the next defended node. Ties are broken arbitrarily. This will result in solutions that are occasionally quite good, but rarely optimal.

4. Lowest degree. Every time step the node with the lowest degree (that is not yet burnt or defended) is chosen to be the next defended node. Ties are broken arbitrarily. Choosing the node with the lowest degree will always result in a bad strategy: the lowest degree vertices are vertices with a single connection, and saving the vertex they are connected to (if possible) is always better, since this saves two vertices instead of one.

These four ways of choosing which node to defend result in very varied partial solutions. SEANN should be able to learn to recognize bad states, good states, and every kind of state in between. Unfortunately, to create a sufficient amount of trees of 200 nodes or graphs of 100 nodes, it would take more than 2000 hours with the current setup. Thus, in our experiments, the largest graphs we examine have 50 nodes, and the largest trees have 100 nodes.

The extended graphs and trees also have an ordered version, where the first node is the node with the most neighbours, the second has the second most neighbours, and so on. This is to test whether such an ordering has any effect on the ability of a neural network to learn from these trees. The graphs and trees are also all extended into random versions, where the nodes are ordered randomly, to pad out the data set even more. The resulting graphs and trees are, of course, isomorphic to their original version, but the resulting adjacency matrix will be very different.

4.2.4 Classification Data

To train CLANN (see Section 4.5), yet another data set is required, because the labels of the examples for the network need to contain the evaluation values of the nodes that can be chosen to be defended by the network. For this we adapt the extended data set. For all of the graphs with partial solutions in this data set, we use SEANN to find the value of each possible next state of the graph. This results in a list of n values: one value for each node i in case node i is defended in the current time step. If node i cannot be defended, because it is already burnt or defended, the corresponding value is 0. The node with the highest value is (theoretically) the node that should be defended.

This classification data set can be used in two ways, and the way they differ is in the feedback given to the network. The network can be given the feedback that all output nodes but the one corresponding to the best vertex should output 0 and the node corresponding to the best vertex 1. The second option is to use the State Evaluation values directly, giving those as feedback to the network. This way the network learns that there are more options that are good. Unfortunately, the results from training with

the State Evaluation values directly were less than satisfactory: the network could not find any patterns in the data. This means that in our research we only give CLANN the feedback on which node should have the highest output. Since we only tell the network which is the best node to protect, we can also use absolute data: Instead of using SEANN to tell us which node is best, we can solve the instance optimally, and take the first node of the solution as the best node, and thus the node to protect. The results of this experiment can be found in Section 5.2.1.

4.3 State Evaluation

To evaluate the state of an FFP instance, there are many options: one can use a simple and fast algorithm to find a solution to a particular state of the problem. This solution s can be used as an indicator of the value of the state: we know that the optimal solution to the instance is at least as good as s. Another way to do state evaluation is to use an Artificial Neural Network. The network receives many examples of states with varying values, and learns characteristics associated with higher and lower values. In this thesis we will use four different algorithms to determine the value of a state in the Greedy State Evaluation (GSE) algorithm. GSE is a simple solver that uses a second algorithm to determine the value of possible next states, choosing the move that results in the state with the highest value. The full pseudocode is displayed in Algorithm 3. These are the algorithms used in line 12:

1. The Greedy algorithm, as described by Hartnell in [14]. This algorithm can only be used on trees, and thus we will only evaluate tree instances of the FFP with it. The combination of the GSE solver with the greedy algorithm as state evaluator will be called GSE-Greedy.
2. The Greedy Look-ahead algorithm (GLA), also only usable on trees. It is based on the greedy algorithm, but instead of looking only at the vertices that will be burnt in the next step, we also look at the vertices that will be burnt in the step after. The full algorithm is shown in Algorithm 4. The resulting solver is GSE-GLA.
3. The first ANN, SEANN, see Section 4.4, resulting in GSE-SEANN.
4. A random algorithm, that simply picks a random untouched node to protect. We are using this as a control group, to see how well the other state evaluators perform. The resulting solver will be called GSE-Random.

Algorithm 3: Greedy State Evaluation (GSE)

input : An instance tree of the FFP
output: An ordered list of vertices that form a solution to the input

```
1  S ← Initialize an empty ordered list
2  while true do
3      frontLine ← get all the untouched nodes adjacent to a burning node
4      if length(frontLine) = 0 then
5          return S
6      if length(frontLine) = 1 then
7          v ← frontLine[0]
8          S.append(v)
9          return S
10     values ← Initialize an empty list
11     for v ∈ frontLine do
12         value ← StateEvaluation(S ∪ {v})
13         values.append(node : value)
14     bestNode ← argmax(values)
15     defend vertex bestNode
16     S.append(bestNode)
17     burn all untouched nodes adjacent to a burning node
```

Algorithm 4: Greedy Look-ahead (GLA)

input : An instance tree of the FFP
output: An ordered list of vertices that form a solution to the input

```
1  S ← Initialize an empty ordered list
2  while true do
3      frontLine ← get all the untouched nodes adjacent to a burning node
4      if length(frontLine) = 0 then
5          return S
6      if length(frontLine) = 1 then
7          v ← frontLine[0]
8          S.append(v)
9          return S
10     greedyPick ← vertex from the frontLine with maximum weight
11     secondLine ← the next layer of nodes
12     if length(secondLine) < 2 then
13         S.append(greedyPick)
14         defend vertex greedyPick
15     else
16         bestWeightYet ← weight(greedyPick)
17         bestNodeYet ← greedyPick
18         for n1 ∈ frontLine do
19             for n2 ∈ secondLine do
20                 if n1 is not the parent of n2 then
21                     if weight(n1) + weight(n2) > bestWeightYet then
22                         bestNodeYet ← n1
23                         bestWeightYet ← weight(n1) + weight(n2)
24         S.append(bestNodeYet)
25         defend vertex bestNodeYet
26     burn all untouched nodes adjacent to a burning node
```

4.4 The first ANN: SEANN

The first ANN is used for State Evaluation, and is hereafter named SEANN. The input SEANN receives is a graph, represented as an adjacency matrix, with for every node two extra $[0, 1]$ markers indicating whether a node is burnt or defended. This input is then fed through two layers of hidden nodes: a larger first hidden layer and a smaller second one. To determine the optimal sizes of those layers, the best optimizer and learning rate, we have done extensive parameter tuning, see Section 4.4.1. SEANN is trained using the set of generated training data, and tested with the set of generated test data, see Section 4.2. For SEANN we also investigate the effects of pruning and ordering the data.

4.4.1 Parameter tuning for SEANN

The parameter tuning for SEANN determines the learning rate, the optimizer, and the size of the layers to use. In both SEANN and CLANN we use two layers, the respective sizes of which are $L1$ and $L2$. To find out which values to use for these parameters, we tried out all combinations of values found in Table 1. To do this, we used a training data set of extended trees with 25 nodes, consisting of 460 874 examples. This data set was further cut into five equal parts, to be used for cross-validation: SEANN is trained on four fifth of the data set, and tested on the remaining examples. This is done for each fifth of the data set, and the result are aggregated. This is done to ensure that the separation of the data set into training data and test data cannot be especially 'lucky' or 'unlucky', resulting in a network that performs worse or better than it might have with a different part of the data set.

Before tuning with these parameters, we tried multiple other optimizers, but many of these showed the problem of vanishing gradients: the weights of the network became so low that the output for all inputs was zero. The Adagrad (from ADAptive GRADient descent) optimizer adapts the learning rate for every weight. This means that the network can be both quick and precise in adapting its weights. If a weight is far from its optimal value, it can be adapted much quicker, and if it is close, the adjustments should be small, so as not to overshoot the mark. The learning rate given to Adagrad is the initial

Table 1: Parameter tuning of SEANN: values of variables, where n is the amount of nodes of the graphs currently being trained on.

Optimizer	Learning rate	$L1$	$L2$
Adadelta	0.5	$n^2 * 1$	$n * 10$
Adagrad	0.4	$n^2 * 2$	$n * 20$
	0.3	$n^2 * 3$	$n * 30$
	0.2	$n^2 * 4$	$n * 40$
	0.1	$n^2 * 5$	$n * 50$
	0.05		
	0.01		

learning rate, the starting point from which they are adapted. For more information on Adagrad, see [8]. The Adadelta optimizer is an extension of Adagrad that seeks to optimize the adaptation of the learning rate even further. This optimizer technically does not need an initial learning rate, but we found that setting one anyway improved its performance. More information on Adadelta can be found in [28].

4.4.2 Results of parameter tuning for SEANN

SEANN has the following parameters: optimizer, learning rate, and layer sizes, as discussed in the methods section. The effect of these parameters are measured in the loss and accuracy of the network. The loss is the Mean Absolute Error (MAE), and this value should be as low as possible. The accuracy is 1− the Mean Squared Error (MSE), and should be as high as possible. These errors are calculated by comparing the network output to the ground truth value that is acquired from the optimal solution of the input instance.

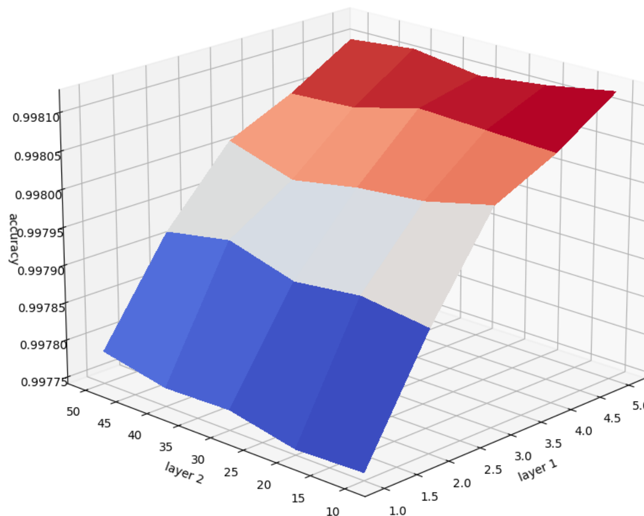

Figure 18: Parameter tuning SEANN: layer sizes

As visible in Figure 18, layer size $L1$ has the biggest impact on the accuracy of the network, as the slope along this axis is much steeper than that of the second layer. For the size of the second layer of the network, smaller is better. Even though larger values for $L1$ have a higher performance, we did not attempt a larger layer size than $n^2 * 5$, because of the impact on the size of the network, and the ability of the hardware to handle it. Which optimizer to use with what learning rate was more difficult to determine. Using the found layer sizes of $L1 = n^2 * 5$ and $L2 = n * 10$ for the second, Adagrad with a learning rate of 0.3 and Adadelta with a learning rate of 0.5 seem to perform equally well, both reaching an accuracy of 0.9985, as is visible in Figure 19. When looking at the loss instead of the accuracy, as in Figure 20, it is clear that Adagrad with a learning rate of 0.3 performs slightly better, while Adadelta performs better overall, even with suboptimal learning rates. The differences between the two best optimizer/learning rate combinations are very small, and as ANNs deal with a degree of randomness, this difference is not large enough to give a definitive answer on which performs better. For my experiments, however, we have

Table 2: Network sizes SEANN

Graph size	Dataset size	$L1$ size	$L2$ size
25	500000	$n^2 * 5$	$n * 10$
50	250000	$n^2 * 5$	$n * 10$
100	25000	$n^2 * 1.5$	$n * 5$

chosen to use the Adagrad optimizer with a learning rate of 0.3, because even though Adadelta might perform marginally better with the optimal learning rate, the difference is so small it hardly matters. We conclude that the following parameters will be used in our experiments with SEANN:

Optimizer:	Adagrad
Learning rate:	0.3
$L1$:	$n^2 * 5$
$L2$:	$n * 10$

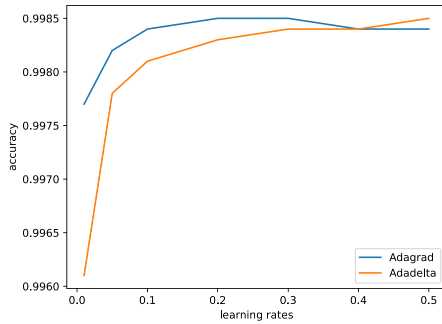

Figure 19: Parameter tuning SEANN: learning rates
(accuracy: 1-MSE)

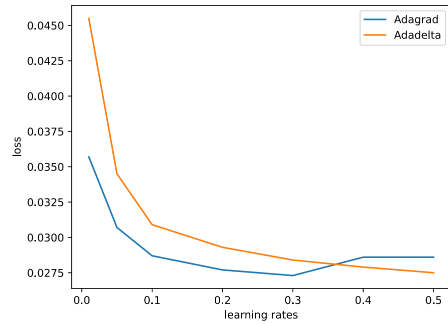

Figure 20: Parameter tuning SEANN: learning rates
(loss: MAE)

Due to the limitations of our hardware, we were forced to adjust our parameters for larger graphs. As the graphics card could not handle the large size of the network combined with the size of the data set, these were adjusted according to Table 2.

4.5 The second ANN: CLANN

The second ANN is used for classification, as it 'classifies' which node to protect given an instance of FFP that is not yet burnt out. This network is hereafter named CLANN. CLANN receives the same input as SEANN, and also has two hidden layers, but the output is a layer of n nodes: one node for each vertex of the input graph. The node with the highest output will be the node that is protected in the current time step. If this node cannot be protected because it is already protected or burnt, the second best node is protected, and so on.

For the feedback used for CLANN there are three options:

1. Using absolute data: the actual best node according to the optimal solution has a feedback value of 1, and the rest of the nodes have a value of 0.

2. Using SEANN: the node with the highest evaluation value has a feedback value of 1, the rest of the nodes have a value of 0.

3. Using SEANN: The feedback of the network consists of the SE value SEANN gives every node.

Options 1 and 2 give a standard feedback to a classification network, but option 3 is more complex. Our attempts to make this work were unsuccessful: the performance of the network remained very low, and often tried to defend nodes that were already defended or even burnt. Using absolute data is then the obvious choice, and Section 5.2.1 supports the use of this measure.

4.5.1 Parameter tuning for CLANN

The purpose of CLANN is fundamentally different from the purpose of SEANN. It stands to reason that the parameters that work well for SEANN may not work as well for CLANN. The first of these parameters to test is the optimizer. Of all the optimizers available in the Tensorflow Keras package, the Adagrad and SGD optimizers show no vanishing gradients, exploding gradients or cases where the network outputs the same values regardless of input data.

We then varied the learning rates, and with the optimal learning rate found, we varied the layer sizes. Table 3 displays all the values investigated.

Table 3: Parameter tuning of CLANN: values of variables, where n is the amount of nodes of the graphs currently being trained on.

Optimizers	Learning rates	hidden layer 1	hidden layer 2
Adagrad	0.01	$n^2 * 4$	$n * 10$
SGD	0.05	$n^2 * 5$	$n * 20$
	0.1	$n^2 * 6$	$n * 30$
	0.2		$n * 40$
	0.3		$n * 50$
	0.4		

4.5.2 Results of parameter tuning for CLANN

CLANN has the same four parameters as SEANN: optimizer, learning rate, and the sizes of layers 1 and 2. The search for optimal values for these parameters is somewhat less extensive, and some similar results can be found. The effect of the parameters is measured in the loss: Sparse Categorical Crossentropy (SCC) and the accuracy: Sparse Categorical Accuracy (SCA). Sparse Categorical Crossentropy uses the following equation to calculate the loss:

$$\text{Loss} = -\sum_{i=0}^{n} y_i * \log \hat{y}_i,$$

where n is the size of the output, y_i is the target value for output node i, and $\hat{y}_i$ is the network output. In SCC it is not necessary to give the network all values for the output nodes, only to indicate which node should have output 1. The other output nodes are automatically set to having 0 as output value. The Sparse Categorical Accuracy is calculated by dividing the number of correct predictions by the total number of predictions. Again, the loss should be as low as possible, whereas the accuracy should be as high as possible.

For CLANN we first determined what the optimizer and learning rate of the network should be. Our expectations after tuning the parameters for SEANN were that these variables would have the most impact on the network. We start our process of parameter tuning using size $L1 = n^2 * 6$ for the first layer, and $L2 = n * 20$ for the second layer. These values were both just one step larger than the optimal layer sizes for SEANN, and as the output layer of CLANN is size n, we expected the layer sizes need be larger as well. As is displayed in Figure 21, 0.05 is the best learning rate for this network for both optimizers. The optimizers themselves perform similarly, with Adagrad having a slightly higher accuracy with the chosen learning rate and layer sizes. The SGD optimizer outperforms Adagrad on higher learning rates.

Using these two optimizers with learning rate 0.05, we investigated what the best layer sizes would be. In Figures 22 and 23 the six plots show the combinations of optimizer + first layer size, with $L2$ on the x axis and loss or accuracy on the y axis. It is clear that Adagrad performs better in terms of accuracy, with the best two options being Adagrad with $L1 = n^2 * 5$ and Adagrad with $L1 = n^2 * 6$. Looking at the graph of the loss as well, we chose to use the Adagrad optimizer with $L1 = n^2 * 5$, because while $L1 = n^2 * 6$ might perform slightly better for layer size $L2 = n * 50$, a smaller network is preferable. The size of the second layer needs to be greater for CLANN than for SEANN, and either $L2 = n * 40$ or $L2 = n * 50$ seems to be optimal. As some of the accuracy starts going slightly down with $L2 = n * 50$, and the loss goes slightly up, $L2 = n * 40$ is the value of choice, especially as a smaller network will train slightly faster and take up less RAM and less memory on the graphics card.

Due to the limitations of our hardware, we were forced to adjust our parameters for larger graphs. As the graphics card could not handle the large size of the network combined with the size of the data set, these were adjusted according to Table 4.

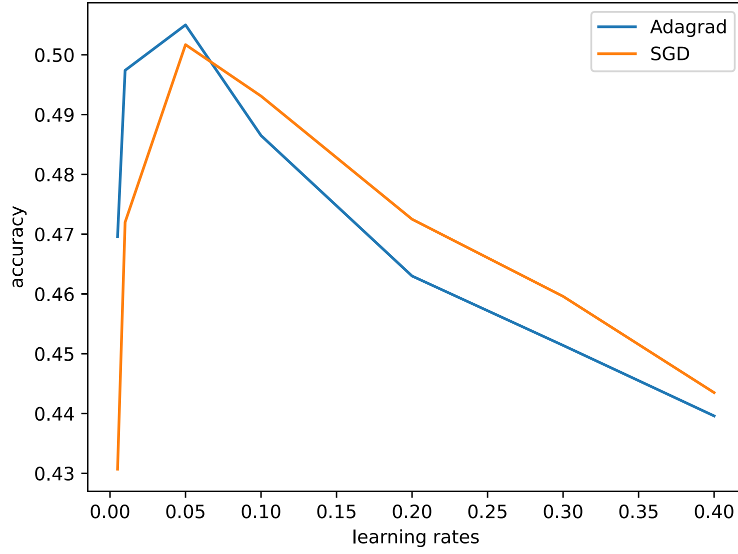

Figure 21: Parameter tuning CLANN: learning rates (accuracy: SCA)

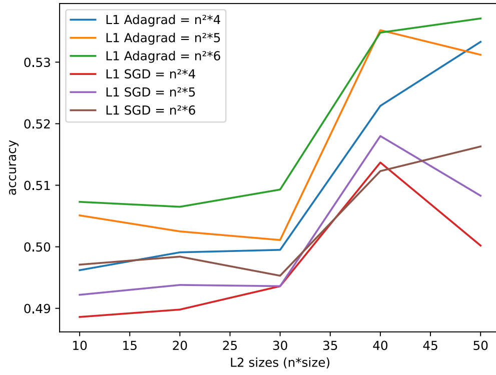

Figure 22: Parameter tuning CLANN: layer sizes (accuracy: SCA)

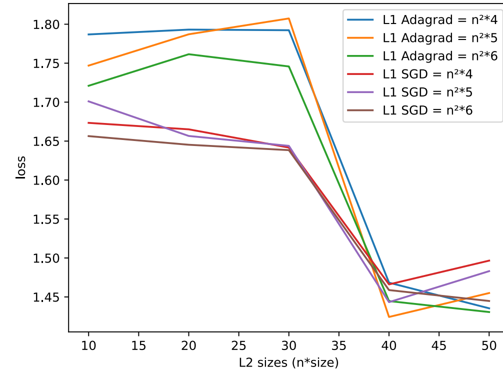

Figure 23: Parameter tuning CLANN: layer sizes (loss: SCC)

4.6 Solving the FFP

We now have several ways to solve the FFP:

1. Solving ILP_{FFP}, see Section 3.3.1.
2. Using the GSE algorithm, see Section 4.3, Algorithm 3.
   (a) GSE-SEANN: GSE with SEANN as evaluator.
   (b) GSE-Greedy: GSE with the Greedy algorithm as evaluator.
   (c) GSE-GLA: GSE with Greedy Look-ahead as evaluator.
   (d) GSE-Random: GSE with the random solver as evaluator.
3. The Greedy algorithm (in case of trees).
4. The Greedy Look-ahead algorithm (in case of trees), see Section 4.3 Algorithm 4.
5. Using a second ANN: CLANN, see Section 4.5.

Table 4: Network sizes CLANN

Graph size	Dataset size	Layer 1 size	Layer 2 size
25	500000	$n^2 * 5$	$n * 40$
50	250000	$n^2 * 5$	$n * 40$
100	25000	$n^2 * 2$	$n * 5$

At first glance, the GSE-Greedy and Greedy Look-ahead algorithms seem to do the same thing: both look one layer ahead and evaluate that layer. However, GSE-Greedy applies the Greedy algorithm to each untouched node adjacent to a burning node, from which an evaluation value can be calculated. The Greedy Look-ahead algorithm on the other hand, evaluates how many nodes can be saved in the current move and next move combined. The evaluation of the nodes is thus purely based on their weight, and not on what nodes may be saved in steps beyond the current step and the one after that. This means that the GSE-Greedy algorithm makes its decision based on more complete information, and therefore we expect its performance to be slightly better than that of GLA. We expect the GSE-GLA algorithm to perform even better, as the evaluation values of GLA are more accurate than those of GSE-Greedy. With more accurate State Evaluation, the GSE algorithm has a higher performance.

The GSE-Random algorithm, which is an objectively 'bad' evaluator, is added to investigate the influence of a 'good' evaluator. As with the other GSE variants, it will give a value to all possible next states, but these values are unreliable. We can however expect that better states will have a higher value and worse states a lower one, but not that the best state has the highest value. Especially in trees, GSE-Random will likely pick nodes to defend that are close to the fire, for these save the most vertices in the current move. In this, the GSE-Random solver has some similarity to the Greedy algorithm. We therefore expect the performance of GSE-Random to be on average only slightly worse than that of the Greedy algorithm.

5 Results

Using the discussed state evaluators and solvers for the FFP problem, we will conduct several experiments. We will compare several methods of State Evaluation, the effects of pruning the network and ordering the graphs. We will also compare the ways of solving the FFP, and evaluate CLANN. Here, too we will study the effect of ordering the graph, and we will look at how much of a difference the use of absolute data makes. Lastly, we will compare graphs and trees, looking at the performance of the GSE-SEANN and CLANN solvers when trained on a mixed data set, and when trained on graphs but tested on trees, and vice versa.

5.1 State Evaluation

5.1.1 Different evaluators

Table 5: Loss and Accuracy of State Evaluators on trees

Treesize	Evaluator	Loss (MAE)	Accuracy (1-MSE)
25	SEANN	0.0216	0.99911
	Greedy	0.003	0.9998
	GLA	**0.0011**	**0.9999**
	Random	0.0993	0.9741
50	SEANN	0.0255	0.99863
	Greedy	0.0043	0.9998
	GLA	**0.0014**	**0.9999**
	Random	0.1269	0.9597
100	SEANN	0.0434	0.99667
	Greedy	0.0047	0.9998
	GLA	**0.0018**	**0.9999**
	Random	0.1275	0.959

We use four different evaluators in the GSE algorithm. In Table 5 we can see that the Greedy evaluator performs very well, and even better than SEANN. The random evaluator has the lowest performance, though the accuracy is still very high. The GLA evaluator performs very close to optimal. This is likely due to the fact that the state evaluation algorithm looks at the layer one step away from the fire, and the GLA algorithm looks a further two steps ahead. Since most trees of 25 nodes do not go deeper than four layers, root included, this is almost equivalent to a brute force approach. This does not explain why the GLA evaluator performs equally well on larger trees. For all tree sizes both Greedy and GLA perform very well. It is important to note that the differences between the evaluators are very small. From this high level of accuracy we can draw the conclusion that a state of FFP is quite easy to evaluate. As the evaluation process takes very little time in case of the Greedy, GLA and Random evaluators, this technique may prove very fruitful in the solving of the problem. Unfortunately, it takes SEANN much longer to evaluate a state of FFP, especially as the network needs to be trained before any evaluations can take place. We hypothesize that SEANN will make different mistakes than Greedy or GLA, and thus will likely perform better on instances that are particularly difficult for these algorithms.

Interestingly, the Random evaluator performs rather well. What this tells us is that the current move makes much more of a difference to the solution than any future moves.

5.1.2 Effects of pruning

Table 6: Effect of pruning on Loss and Accuracy of SEANN

Graph Type	Pruning?	Loss (MAE)	Accuracy (1-MSE)
trees (25)	yes	0.0266	0.99863
	no	**0.0251**	**0.99866**
graphs(25)	yes	**0.0435**	**0.99481**
	no	0.045	0.99445
trees(50)	yes	**0.0362**	**0.99743**
	no	0.0363	0.99744
graphs(50)	yes	**0.0492**	0.99247
	no	0.052	**0.99248**
trees(100)	yes	**0.0737**	**0.99003**
	no	0.0801	0.98865

We pruned our model using the pruning step available in the Keras package of Tensorflow, with an initial sparsity of 50%, and a final sparsity of 80%. These values are given in the example shown in the documentation of the package, and we found they work well for our purposes. The most prevalent effect of pruning SEANN is the speed with which the network learns. For most graph types the loss decreases and the accuracy improves, see Table 6. This happens because pruning is a method used to combat overfitting, and this positive effect, though small, is visible in our data as well. The positive effect of pruning seems more prevalent with larger trees and with graphs. This may be the case because it is more difficult for the network to learn from these, as can be observed in Table 5. With larger trees, the accuracy of SEANN goes down slightly. Another factor is the decreasing size of the dataset the network learns from with increasing graph size. Overfitting becomes more of a problem with a smaller dataset, and thus measures against overfitting are more effective in these cases.

5.1.3 Effects of ordering the data

When comparing the loss and accuracy of SEANN using the original data set to that of the ordered data set, it is immediately clear that ordering the vertices by degree has a positive effect on both. This is indicated in Table 7, where we can see improvements in both the loss and accuracy of SEANN when it is trained and tested on ordered graphs. This effect increases with the size of the graphs, showing that the degree of a vertex in combination with its status (burnt/defended/untouched) is an important indicator for the network of the value of the input graph. This also gives rise to the idea that incorporating more metadata in the input of both SEANN and CLANN may be useful to help the network learn.

Table 7: Effect of ordering of graphs on Loss and Accuracy of SEANN

Graph Type	Ordered?	Loss (MAE)	Accuracy (1-MSE)
trees (25)	no	0.0266	0.99863
	yes	**0.0216**	**0.99911**
graphs(25)	no	0.0435	0.99481
	yes	**0.037**	**0.99615**
trees(50)	no	0.0362	0.99743
	yes	**0.0255**	**0.99863**
graphs(50)	no	0.0492	0.99247
	yes	**0.0359**	**0.99515**
trees(100)	no	0.0737	0.99003
	yes	**0.0434**	**0.99667**

Table 8: Effect of using absolute data on the solutions acquired using CLANN

Graph Type	Absolute data?	Average % burnt	Average runtime (s)
trees (25)	no	39.61	0.102
	yes	**38.1**	0.102
graphs(25)	no	72.39	0.143
	yes	**72.23**	0.142
trees(50)	no	41.89	0.209
	yes	**40.71**	0.2
graphs(50)	no	82.62	0.25
	yes	**82.36**	0.272

5.2 Solving the FFP

5.2.1 Using absolute data

Table 8 shows the results of training CLANN with absolute data or with the data acquired by using SEANN. Here, we only use feedback options 1 and 2 discussed in Section 4.5. As is clear, the difference in the average percentage of burnt vertices is quite minimal. For trees there is around 1% difference, and for graphs it hardly seems to make any difference at all. This means that SEANN does a very good job of evaluating the states of the examples used to train CLANN, as will be further confirmed in Section 5.2.2. Because using the absolute data is slightly better, and because we do not have to generate classification data for larger graphs, we use absolute data to train CLANN in the rest of our experiments.

5.2.2 Different Solvers

Tables 9, 10, and 11 show the performance of the different Solvers on graphs with 25, 50 and 100 nodes respectively. The average runtime is the runtime of the algorithm excluding the time it takes to train SEANN or CLANN in case of the GSE-SEANN and CLANN solvers. The ILP solver always gives the optimal solution, so this is what we will compare performance against in terms of how many vertices are burnt. For all three graph sizes, CLANN performs worst, and its performance goes down with the size of the graph. CLANN is also the slowest of all the solvers. The solver with the best results is GSE-GLA, performing very close to optimal for all graph sizes. In addition to this, it is also a rather fast solver, faster than Gurobi and faster than either of the ANN solvers. The fastest solvers are Greedy and GLA, solving instances as large as trees with 100 nodes in less than one thousandth of a second, and both are quite good at solving FFP. In all cases, GLA burns less than 1% more than optimal on average, with Greedy following close behind.

The GSE-Random solver exceeds all expectations in terms of performance, outperforming GSE-SEANN on trees with 50 nodes and trees with 100 nodes. GSE-Random is relatively fast, but slightly slower than the ILP solver. The reason for this is likely suboptimal coding: random number generation is computationally quite heavy, and generating a large amount of random numbers at the same time is much quicker than generating them individually. The latter is what happens in the GSE-Random algorithm, as only a single graph is solved at a time, and only a single node is chosen to be protected at a time. This makes it difficult to generate multiple random numbers in one go.

The Greedy solver is only $\frac{1}{2}$OPT in theory, but in our data set we see that the Greedy algorithm is

Table 9: Results of the algorithms on trees of size 25

Solver	Average%burnt	Average runtime (s)
ILP	**33.96**	0.007
GSE-GLA	**33.96**	0.004
GSE-Greedy	34	0.004
GLA	34.17	**0.0**
Greedy	35.32	**0.0**
GSE-SEANN	35.34	0.098
GSE-Random	35.59	0.006
CLANN	38.1	0.102

Table 10: Results of the algorithms on trees of size 50

Solver	Average%burnt	Average runtime (s)
ILP	**29.3**	0.021
GSE-GLA	29.31	0.013
GSE-Greedy	29.37	0.012
GLA	29.58	**0.0**
Greedy	30.84	**0.0**
GSE-SEANN	31.21	0.257
GSE-Random	31.09	0.022
CLANN	40.71	0.2

Table 11: Results of the algorithms on trees of size 100

Solver	Average%burnt	Average runtime (s)
ILP	**26.73**	0.076
GSE-GLA	26.74	0.046
GSE-Greedy	26.81	0.044
GLA	27.04	**0.0**
Greedy	28.26	**0.0**
GSE-SEANN	29.69	0.816
GSE-Random	28.45	0.091
CLANN	57.37	0.396

much better than that at solving a variety of random graphs. This means that the worst-case scenario (see Section 3) does not occur very often in our data set, or that when it occurs, it is in a sub-tree of the full tree. This mitigates the damage that choosing a sub-optimal node to defend can do.

We further investigate the performance of the State Evaluation algorithms, especially compared to the Greedy algorithm, by looking at the 1% of graphs on which the Greedy algorithm has the worst performance. Tables 12, 13 and 14 show the percentages of how many of the solutions to these graphs were optimal, equal to Greedy, better than Greedy (but not optimal), or worse than Greedy. GSE-Greedy and GSE-GLA perform best, each giving the most optimal solutions to all these 'difficult' graphs. GSE-GLA gives optimal solutions for graphs with 50 and 100 nodes as well.

CLANN is the solver that most often gives a worse solution than Greedy on graphs of 25 nodes, followed by GSE-SEANN and GSE-Random. These solvers are also the only ones to give solutions that are worse than Greedy. This is not surprising, as all other algorithms are specifically guarded against Greedy's weakness by looking ahead in some way.

Both GSE-SEANN and CLANN perform better on smaller trees, but on trees with 100 nodes even the GSE-Random algorithm outperforms them. This bodes ill for even larger graphs, and to be considered a viable way of solving FFP, CLANN needs much improvement in this area.

Table 12: Results of the algorithms on the 1% of trees with 25 nodes on which the Greedy algorithm has the worst performance

Solver	Optimal(%)	Equal to Greedy(%)	Better than Greedy(%)	Worse than Greedy(%)
GLA	88.33	1.67	10	0
CLANN	60	0	30	10
GSE-Greedy	100	0	0	0
GSE-GLA	100	0	0	0
GSE-SEANN	28.33	56.67	8.33	6.67
GSE-Random	40	46.67	11.67	1.67

Table 13: Results of the algorithms on the 1% of trees with 50 nodes on which the Greedy algorithm has the worst performance

Solver	Optimal(%)	Equal to Greedy (%)	Better than Greedy (%)	Worse than Greedy(%)
GLA	85	3.33	11.67	0
CLANN	20	5	50	25
GSE-Greedy	98.33	0	1.67	0
GSE-GLA	100	0	0	0
GSE-SEANN	35	45	11.67	8.33
GSE-Random	18.33	45	25	11.67

Table 14: Results of the algorithms on the 1% of trees with 100 nodes nodes on which the Greedy algorithm has the worst performance

Solver	Optimal(%)	Equal to Greedy (%)	Better than Greedy (%)	Worse than Greedy(%)
GLA	83.33	1.67	15	0
CLANN	0	0	1.67	98.33
GSE-Greedy	96.67	0	3.33	0
GSE-GLA	100	0	0	0
GSE-SEANN	0	23.33	25	51.67
GSE-Random	15	36.67	23.33	25

As can be seen in Table 13, there is a single graph where GSE-Greedy does not perform optimally. The graph in question can be seen in Figure 24. It is no surprise that this particular instance features as one that the Greedy algorithm performs badly on: it shows exactly the worst-case scenario for the Greedy algorithm. The Greedy solution defends nodes

$$2, 6, 13, 39$$

in order, saving 36 nodes. An optimal solution would be to protect nodes

$$1, 6, 7, 32$$

in order. This would save 43 nodes, and GSE-Greedy saves 41 nodes. Where GSE-Greedy goes wrong is with the evaluation of node 1. The Greedy algorithm will choose to protect node 5 before node 6, resulting in a lower overall evaluation. When evaluating node 3 the Greedy algorithm will choose to protect node 5 also, but this has a lower impact on the evaluation value, because node 1 has fewer children. This also explains why GSE-GLA does solve this graph optimally: it can look one more step ahead, and thus protects node 7 in time step 3, allowing node 6 to be protected in time step 2.

5.2.3 Graphs vs. Trees

The results of our experiments with training SEANN and CLANN on trees, graphs or a mix of both, and testing on graphs and/or trees are reported in Tables 15 and 16. The most visible difference between the graphs and trees is that in the graphs, almost twice as many nodes burn as in trees. This is unsurprising, as trees have the advantage of being non-cyclical: from the root to any node there is only ever one path. Training on graphs and subsequently testing on trees and vice versa is not as detrimental to the performance of the ANN solvers as expected. It is even the case that the network trained on both (‘mixed’ in the tables) slightly outperforms the network that is both trained and tested on the same graph type.

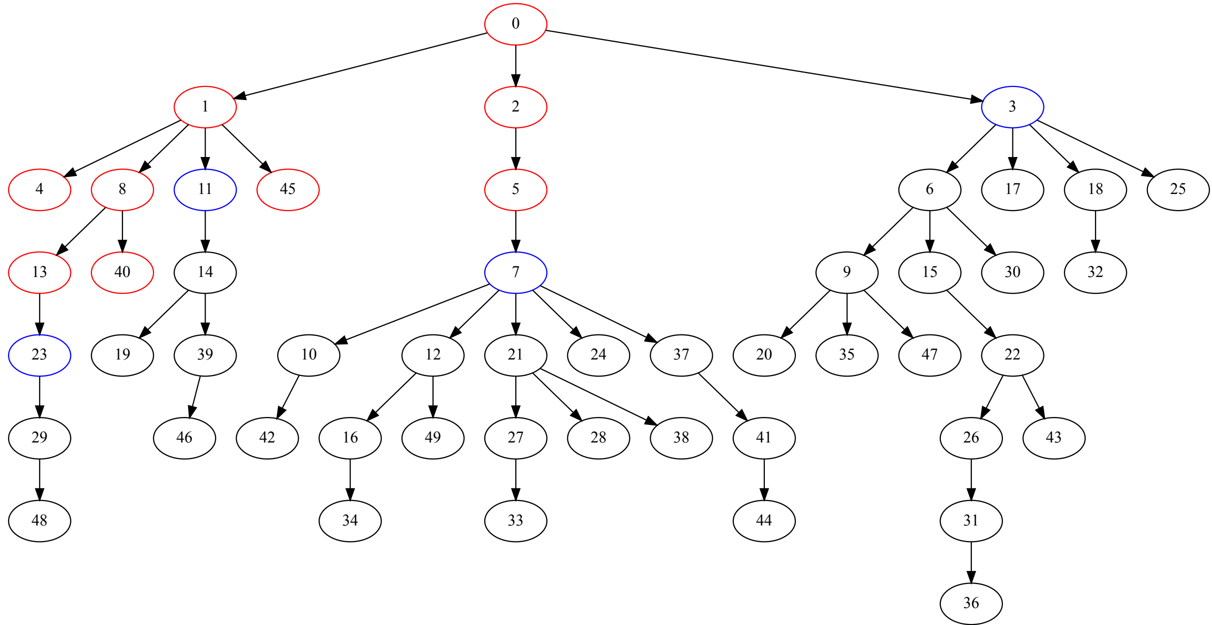

Figure 24: Tree with GSE-Greedy performing sub-optimally. Red nodes are burnt, blue nodes are defended. Greedy solution: 2, 6, 13, 39. Optimal solution: 1, 6, 7

5.2.4 Effects of ordering the data

As seen in Section 5.1, ordering the graphs that SEANN learns from has a positive effect on the ability of the network to evaluate the given state. The same is the case for CLANN, showing a decrease of burnt vertices when using the CLANN Solver on trees and graphs of increasing sizes, as can be seen in Table 17. The effect of ordering the data increases slightly with the size of the graphs, from 2.56% on trees with 25 nodes, to 5.11% on trees with 100 nodes. When looking at the results of the GSE-SEANN solver, which can be seen in Table 18, we do not see such a difference. There seems to be a mild improvement in case of graphs, but for the trees the algorithm seems to perform slightly worse. From this we can conclude that while it makes a difference to the accuracy of both CLANN and SEANN, for the GSE-SEANN algorithm the state evaluations of ordered graphs do not differ very much from those of non-ordered graphs, and when they do, they may even cause the wrong next state to be given the highest value.

5.2.5 ANN-solvers vs. GSE-Random

When looking at the performance of the GSE-SEANN and CLANN solvers, we see in all above tables that their performance is worse than that of simple greedy algorithms. While CLANN does perform better than GSE-Random for smaller graphs, we would not recommend its usage the way it is now, especially with faster, more accurate approximations such as GSE-GLA. The GSE-SEANN solver faces similar problems. While its performance is markedly better than that of CLANN, it does not exceed the performance of GSE-Random either, though by less of a margin than the CLANN solver. The most significant drawback of both methods is the average time they take to solve a single tree, even excluding the time it takes to train either network. This makes both methods unusable, as the point of an approximation method is to be faster than an exact algorithm.

Table 15: Comparing graphs and trees of 25 nodes

Solver	Testdata	Trainingdata	Average%burnt
GSE-SEANN	Trees	mixed	**35.28**
		Trees	35.34
		Graphs	35.69
	Graphs	mixed	**64.1**
		trees	64.65
		graphs	64.24
CLANN	Trees	mixed	39.58
		trees	**38.1**
		graphs	48.99
	Graphs	mixed	**71.96**
		trees	72.08
		graphs	72.23

Table 16: Comparing graphs and trees of 50 nodes

Solver	Testdata	Trainingdata	Average%burnt
GSE-SEANN	Trees	mixed	31.29
		Trees	**31.21**
		Graphs	31.48
	Graphs	mixed	72.93
		trees	**72.69**
		graphs	73.28
CLANN	Trees	mixed	42.1
		trees	**40.71**
		graphs	53.85
	Graphs	mixed	**82.36**
		trees	82.76
		graphs	**82.36**

6 Conclusion

Several conclusions can be drawn from the results of our experiments. Firstly, it is possible to use techniques from Game-AI to solve the FFP. The state evaluation works well, especially the GSE-GLA solver, and SEANN can learn to evaluate states of an FFP instance with a high accuracy. Unfortunately, CLANN does not perform as well as hoped, and thus we can conclude that modelling the question of 'which node to defend at the current time step' as a classification problem does not yield good solutions for FFP. We did find that it does not make a large difference to SEANN or CLANN whether the input graphs are trees or not. In fact, both networks seem to benefit from a combination of both, giving the network more varied examples to learn from. Another factor that helps the network make sense of the data is the ordering of the nodes of the input graphs. This shows that SEANN and CLANN benefit from metadata, and that the degree of a vertex is important in determining whether it should be protected.

When comparing our approximation methods to the results of the ILP solver, we find that the average run time of Gurobi is quite small, at least for the sizes of graphs and trees that we tested. For larger graphs, this becomes much more of an issue, and approximation algorithms are thus more relevant. This area is were the most headway can be made in the future. When considering the run time of the different solvers, simple greedy algorithms are clearly superior to any other methods, at least on our data set.

Our data set is quite varied, but even more variation would likely have improved performance of SEANN and CLANN even further. The variation present in the graphs were not optimal, as increasing the average degree of the nodes mostly just caused more of the graph to burn, rather than providing unique challenges as the variety of trees did.

An unexpected find is that the Greedy algorithm is not as bad in practice as it is in theoretical analysis. On our database of random graphs it performs exceedingly well, saving less than two percent fewer vertices compared to optimal. This result is far from the expected 'half as good as optimal' that the analysis of the algorithm predicts. This means that on a data set of random trees, the 'worst-case scenario' is rather rare, and this makes the Greedy algorithm a very valid method of acquiring an approximated solution of FFP quickly.

Table 17: Effect of ordering of graphs on performance of CLANN

Graph Type	Ordered?	Average % burnt	Average runtime (s)
trees (25)	no	38.1	0.102
	yes	**35.54**	0.099
graphs(25)	no	72.23	0.142
	yes	**70.64**	0.143
trees(50)	no	40.71	0.2
	yes	**37.84**	0.188
graphs(50)	no	82.36	0.272
	yes	**80.71**	0.23
trees(100)	no	57.37	0.396
	yes	**52.28**	0.384

Table 18: Effect of ordering of graphs on performance of GSE-SEANN

Graph Type	Ordered?	Average % burnt	Average runtime (s)
trees (25)	no	**35.34**	0.098
	yes	35.7	0.098
graphs(25)	no	64.24	0.159
	yes	**63.9**	0.154
trees(50)	no	**31.21**	0.257
	yes	32.06	0.187
graphs(50)	no	73.28	0.379
	yes	**71.22**	0.305
trees(100)	no	**29.69**	0.816
	yes	30.23	0.78

7 Future Work

For future research we would suggest investigating ways to make SEANN and CLANN applicable to larger graphs. The current model is not very scalable - though a better graphics card with more memory would help. Increasing the amount of examples the network trains on may also yield better results, as well as creating even more variation in the data set. Giving the ANNs more metadata, such as incorporating the depth of nodes or their distance from the fire, to work with may also improve their performance.
Another idea is sampling: taking a sample (or multiple samples) of the graph and applying state evaluation to a partial problem. This will largely solve the issue of scalability, but creates the question of how to sample such that the sample is (or samples are) representative of the entire network.
As State Evaluation has been proven successful, finding more applications is an obvious next step. SE may for example be used as part of a heuristic, or to find starting solutions to speed up an optimal solver. Additionally, using smarter, more complex (approximation) algorithms than GSE may also prove fruitful. Due to the positive results, further investigation into the GSE-GLA algorithm is of interest, especially for larger graphs. For graphs with 25-100 nodes performance barely decreased when the graph size increased, and it is worth investigating how far this trend may go.
Some of the SE algorithms proposed in this research, such as GSE-GLA, are fully deterministic. It is easy to see that this solver runs in polynomial time, but we have made no attempt to investigate the exact approximation ratio of this algorithm. This will be especially interesting as its performance on our database is very good, and we ourselves have not encountered any obvious downside to the method, such as is the case for the Greedy algorithm.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Shun-ichi Amari. A theory of adaptive pattern classifiers. *IEEE Trans. Electron. Comput.*, 16(3):299–307, 1967.

[3] Cristina Bazgan, Morgan Chopin, Marek Cygan, Michael R. Fellows, Fedor V. Fomin, and Erik Jan van Leeuwen. Parameterized complexity of firefighting. *J. Comput. Syst. Sci.*, 80(7):1285–1297, 2014.

[4] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the state of neural network pruning? In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[5] Christian Blum, Maria J. Blesa, Carlos García-Martínez, Francisco J. Rodríguez, and Manuel Lozano. The firefighter problem: Application of hybrid ant colony optimization algorithms. In Christian Blum and Gabriela Ochoa, editors, *Evolutionary Computation in Combinatorial Optimisation - 14th European Conference, EvoCOP 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers*, volume 8600 of *Lecture Notes in Computer Science*, pages 218–229. Springer, 2014.

[6] Leizhen Cai, Elad Verbin, and Lin Yang. Firefighting on trees: (1-1/e)-approximation, fixed parameter tractability and a subexponential algorithm. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation, 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, volume 5369 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2008.

[7] Mike Develin and Stephen G. Hartke. Fire containment in grids of dimension three and higher. *Discret. Appl. Math.*, 155(17):2257–2268, 2007.

[8] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[9] Stephen Finbow, Andrew D. King, Gary MacGillivray, and Romeo Rizzi. The firefighter problem for graphs of maximum degree three. *Discret. Math.*, 307(16):2094–2105, 2007.

[10] Stephen Finbow and Gary MacGillivray. The firefighter problem: a survey of results, directions and questions. *Australas. J Comb.*, 43:57–78, 2009.

[11] LLC Gurobi Optimization. Gurobi optimization, 2019.

[12] LLC Gurobi Optimization. gurobipy, the gurobi python interface, 2019.

[13] Bert Hartnell. Firefighter! an application of domination. In *the 24th Manitoba Conference on Combinatorial Mathematics and Computing, University of Minitoba, Winnipeg, Canada, 1995*, 1995.

[14] Bert Hartnell. Firefighting on trees: how bad is the greedy algorithm? *Congressus Numerantium*, 145:187–192, 2000.

[15] IBM. Ibm ilog cplex optimizer, 2022.

[16] Andrew D. King and Gary MacGillivray. The firefighter problem for cubic graphs. *Discret. Math.*, 310(3):614–621, 2010.

[17] Krzysztof Michalak. Auto-adaptation of genetic operators for multi-objective optimization in the firefighter problem. In Emilio Corchado, José Antonio Lozano, Héctor Quintián, and Hujun Yin, editors, *Intelligent Data Engineering and Automated Learning - IDEAL 2014 - 15th International Conference, Salamanca, Spain, September 10-12, 2014. Proceedings*, volume 8669 of *Lecture Notes in Computer Science*, pages 484–491. Springer, 2014.

[18] Krzysztof Michalak. The sim-ea algorithm with operator autoadaptation for the multiobjective firefighter problem. In Gabriela Ochoa and Francisco Chicano, editors, *Evolutionary Computation in Combinatorial Optimization - 15th European Conference, EvoCOP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings*, volume 9026 of *Lecture Notes in Computer Science*, pages 184–196. Springer, 2015.

[19] Krzysztof Michalak. ED-LS: a heuristic local search for the firefighter problem. In Hernán E. Aguirre and Keiki Takadama, editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, pages 25–26. ACM, 2018.

[20] Krzysztof Michalak. Solving the parameterless firefighter problem using multiobjective evolutionary algorithms. In Manuel López-Ibáñez, Anne Auger, and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 1321–1328. ACM, 2019.

[21] Daniel Michulke and Michael Thielscher. Neural networks for state evaluation in general game playing. In Wray L. Buntine, Marko Grobelnik, Dunja Mladenic, and John Shawe-Taylor, editors, *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2009, Bled, Slovenia, September 7-11, 2009, Proceedings, Part II*, volume 5782 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2009.

[22] Ian Millington and John Funge. *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann, 2009.

[23] Adolfo J Quiroz. Fast random generation of binary, t-ary and other types of trees. *Journal of Classification*, 6(1):223–231, 1989.

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[25] Sebastian Thrun. Learning to play the game of chess. In Gerald Tesauro, David S. Touretzky, and Todd K. Leen, editors, *Advances in Neural Information Processing Systems 7, [NIPS Conference, Denver, Colorado, USA, 1994]*, pages 1069–1076. MIT Press, 1994.

[26] Hiroki Tomizawa, Shin-ichi Maeda, and Shin Ishii. Learning of go board state evaluation function by artificial neural network. In Chi-Sing Leung, Minho Lee, and Jonathan Hoyin Chan, editors, *Neural Information Processing, 16th International Conference, ICONIP 2009, Bangkok, Thailand, December 1-5, 2009, Proceedings, Part I*, volume 5863 of *Lecture Notes in Computer Science*, pages 598–605. Springer, 2009.

[27] Connor Wagner. A new survey on the firefighter problem. Master's thesis, University of Victoria, 2021.

[28] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.