university of groningen / faculty of science and engineering

Using reinforcement learning to fight forest fires: Comparing CMC with CoSyNE

Henric Marijn Roothaert, s2929244, h.m.roothaert@student.rug.nl,
Supervisor: dr. M.A. Wiering

Abstract: Minimising both the economical and physiological damages caused by forest fires is a complex problem. This paper builds upon an existing Reinforcement Learning (RL) based Decision Support System (DSS), which could optimise fire management techniques once fully developed. The aim is to find out which RL-technique is best suited for forest fire control, Connectionist Monte Carlo (CMC) or Cooperative Synapse NeuroEvolution (CoSyNE). Both RL techniques were trained to place sub-goals optimally in all wind directions around the center of a simulated fire. These sub-goals are in turn completed by firefighting agents. By varying the number of agents and the path-finding algorithm used by these agents, different levels of complexity were induced to the fire management problem. Overall, CoSyNE significantly outperformed CMC. There are however considerable improvements to be made to both the implementation of the RL techniques and to the simulation before it can be used to train a reliable DSS.

1 Introduction

Forest fires are a hot topic as of late. As higher temperatures are "linked to increasing probabilities of severe fire weather and fire spread" (van Mantgem et al., 2013), the number and severity of forest fires is likely to increase due to global warming. Next to the well-known health and safety hazards associated with these fires, a forest fire can have an immense economical impact as well. The United States of America alone has an annual budget of 2.4 billion USD dedicated to forest fire control as of 2020 (Department of Agriculture, 2019). Considering that this budget is solely for *preventing* forest fires, the physiological costs of forest fires (i.e. burnt property, increase in healthcare costs, CO2 emissions) should be added to the total costs of these fires. Therefore it is desirable to optimise the forest fire control policies.

There are several ways of combating forest fires. The one most prominently used by firefighters is using a "fire line". This is a natural or man-made barrier which is hard to cross for a fire. The difficulty of placing these fire lines is finding the balance between saving as much resources as possible, and the risk of the fire crossing this fire line. If the fire line is placed at a large distance of the fire, there is plenty of time to construct a barrier which cannot be crossed. However, everything between the barrier and the fire will most likely burn down. Place the fire line too close to the fire and there might not be enough time to finish it or the fire is able to cross it somewhere along the way. In this case the fire will most often grow to an uncontrollable size, resulting in massive losses.

To aid this decision, a few *fire management decision support systems* have been developed. However, most simplify the assumptions made about the fire to such an extent, that the capabilities of assisting in fire management efforts are limited. The others require years of estimation and calibration procedures (Mavsar et al., 2013). Furthermore, all of these decision support systems (DSS) use a policy made by experts as to where to place the fire lines. The DSS will not place these fire lines by itself. This is where a new Machine Learning (ML) based approach could provide an improvement in current forest fire control procedures. By using ML to estimate an optimal policy for placing fire lines, the new DSS could provide out-of-the-box solutions for the fire lines problem.

ML is divided into three different categories, Supervised Learning, Unsupervised Learning and Reinforcement Learning (RL). Supervised learning re-

quires a database with labeled data. While some fire management DSS, such as the Canadian LEOPARDS, the Spanish SINAMI and the American FPA, use historical databases to optimise the predictions given by the DSS, these databases mostly contain information about the behaviour of fire in past events. The effect and placement of fire lines is not included. Therefore a database containing the data required for supervised learning does not exist. Unsupervised learning is mostly used for finding a structure in unlabeled data-sets (Sutton and Barto, 2018, Chapter 1). While it could be useful to classify or group certain forest-fire types, this category of ML is not well suited for approximating the best location for fire lines. RL provides the solution here, as it is able to produce a policy for placing fire lines without requiring a labelled database. It does so by optimising its policy based on experiences obtained in a simulation. Provided that the simulation upon which the policy is optimised is sufficiently true to nature, the policy should be generalizable to real world scenarios.

Some methods for constructing such an RL-based fire management DSS have already been proposed (Wiering and Dorigo, 1998; Wiering et al., 2005). They provide an initial approach on how to set up the reinforcement learning environment and are used as inspiration for the model that will be discussed in detail in section 2.2. They however do not provide any comparison on which RL technique would be best suited for controlling forest fires. This paper therefore proposes a simulation in which multiple techniques are put next to each other, allowing for a direct comparison between those techniques.

The first technique used is a Monte Carlo method, Connectionist Monte Carlo* (*CMC*). Monte Carlo methods require episodic tasks, meaning that an episode has a clear ending after which the total return can be determined. This is applicable to the forest fire simulation as an episode can be defined as a single simulated forest fire. This episode ends after containing the fire, or once the fire has escaped the premises of the simulation. Another reason to implement the Monte Carlo method is because these methods are generally used in environments with "a significant random component" (Sutton and Barto, 2018, Chapter 5). Generally speaking, a forest fire does not start at the same location twice and if so, the propagation of forest fires is affected by a great deal of environmental conditions (Xavier Viegas, 1998). Therefore the environment can be considered a significant random component.

The second technique used is Cooperative Synapse NeuroEvolution* (*CoSyNE*). This is an Evolutionary Neural Network that outperformed the Enforced Sub-Populations (ESP) used in the previous implementation of a fire management DSS (Wiering et al., 2005) in difficult versions of the pole-balancing task (Gomez et al., 2006, 2008). While good performance on the pole-balancing task is no guarantee for a good performance on the forest fire control task, the pole-balancing task is regarded as a benchmark for trainable controllers (Geva and Sitte, 1993). Therefore CoSyNE is used instead of ESP.

To test the different approaches, two testing conditions were manipulated. As fighting forest fires is often a joint effort of multiple fire-fighters/-brigades, it is interesting to know how well the approach handles a multi-agent environment. Therefore the number of agents varies across different runs. The additional level of complexity shows how well the RL technique is able to handle a changing environment, as other agents are modifying the environment to accomplish their own goals.

The path-planning algorithm used by the agents is varied as well. One approach simply goes from its current location to its goal in a straight line. The other takes the fastest route from its current location to its goal. This allows better utilisation of the environment, but could cause the agent to walk a path too close to the fire and is therefore riskier. This level of complexity is meant to give insights on how well the RL techniques deal with the uncertainty of the result of actions made by the agents.

2 System Description

The system which is built to test the different RL techniques is categorised in four main parts: the forest-fire simulation, the sub-goal management system, the used RL techniques and how the data is structured to enable proper analysis of the performance of the two RL techniques.

*The details of this technique are discussed in section 2.3

Table 2.1: Relation between local parameters and square categories

Category	Tree	Grass	Dirt	Road
Amount of fuel	High	Medium	-	-
Ignition temperature	High	Low	-	-
Clear costs	High	Low	-	-
Move costs	High	Low	Low	Very low

2.1 The simulation

The main functionality of the simulation is to enable the agents to move around in an environment in which they can ultimately control a fire. To add realism to the simulation, there are a few global and local variables. Different combinations of these variables create different terrain types and weather conditions.

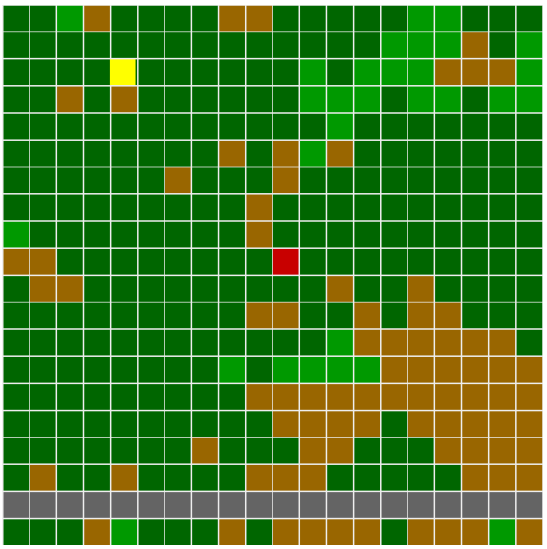

Figure 2.1: Example of testing environment. The fire has started in the middle and an agent can be seen in the upper left corner.

The environment. The simulation consists of a 20×20 grid in which each square is classified into one of four groups: Tree, Grass, Dirt or Road. The difference between these categories is how much fuel is stored in the square, at what temperature the square will catch fire, how much energy it takes to clear the square (i.e. turn it into dirt) and how much it costs for an agent to travel across them. An overview of the relation between these parameters and categories is shown in table 2.1.

When generating the grid, squares of a certain category are clustered. The size and amount of these clusters depend on global parameters related to humidity and urbanisation, giving the possibility to simulate a specific climate (higher humidity results in less dirt and more trees) and human activity (the number of roads) in the area. An example of a moderately humid and moderately urbanised environment is shown in figure 2.1. The effect of climate and human activity are beyond the scope of this experiment and thus the parameter settings were not manipulated over the course of the experiment.

Fire Dynamics. The fire propagates through the production of heat. If a square is on fire, it will slowly turn its fuel into heat which is radiated to adjacent tiles. The direction and intensity of this radiation is determined by the fire intensity, wind-speed and wind-direction. Once a square has reached its individual ignition threshold, it will ignite and start producing heat as well. Although this implementation is not entirely true to nature (Xavier Viegas, 1998), it requires little computational power.

2.2 Sub-goal management

After initialising the environment, the agents can start controlling the fire. The *Incident Commander* (IC) controls the sub-goals that need to be reached in order to contain the fire. Once these have been set, these sub-goals are assigned to ground agents. These agents will in turn use a *path finding algorithm* to determine which path to take to reach their goals.

Incident Commander. In real life situations, the IC conjures a strategy on how to optimally divide the available resources (Molina et al.). The same principle is used in the simulation. The IC determines where to place the sub-goals which are in turn executed by the ground agents. There are a total of eight sub-goals, placed in all wind-directions

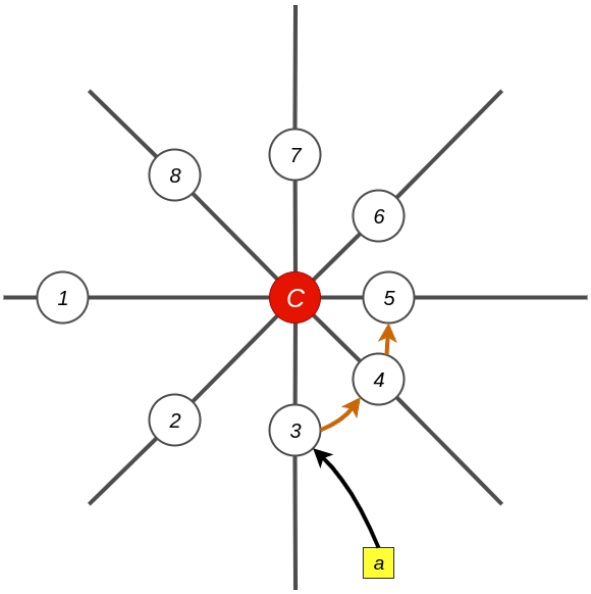

Figure 2.2: Placement of the sub-goals. The distance between sub-goals *1* to *8* and the center of the fire *C* is determined through RL. After initialisation, agent *a* will first move to the closest sub-goal, in this case number *3*. From there it will cut a fire line from *3* to *4*, *4* to *5*, etc.

around the center of the fire. The offset to the center of the fire is ultimately determined through RL, which is discussed in section 2.3.

In single agent environments, the agent is assigned the closest sub-goal once the initial offset of all sub-goals is determined with respect to the position of this agent. This agent will then cut a fire line in a counter-clockwise fashion to the next sub-goal as shown in figure 2.2. When the agent reaches a sub-goal, the next sub-goal is re-evaluated. By doing so, the IC is able to account for the growth of the fire and possibly changing weather conditions.

In multi-agent environments, the assignment of sub-goals becomes more complicated. In these situations, sub-goal assignment is done by iterating over all agents in the environment. First, the sub-goals are initialised with respect to the position of the first agent. Afterwards, the closest sub-goal is assigned to that agent. This sub-goal is in turn labelled as "occupied". Next, all non-occupied sub-goals are re-evaluated with respect to the second agent. This agent is assigned the closest non-occupied sub-goal. This continues until all agents are assigned a sub-goal. Once an agent reaches a sub-goal, there are three options: the next sub-goal is not occupied, the next sub-goal is occupied, or another agent has already cut a fire-line towards the next sub-goal meaning that it has been completed. If the next sub-goal is not occupied, this sub-goal is re-evaluated and the agent starts cutting towards it. If the next sub-goal is occupied but not completed, the agent will cut towards it without re-evaluating this sub-goal. If the next sub-goal has already been completed, the agent will re-evaluate all remaining non-occupied sub-goals and navigate to the closest one.

Path-finding Algorithms. As mentioned in the introduction, two path-planning algorithms are implemented to reach the set sub-goals. The implemented algorithms are the A* algorithm (Hart et al., 1968) and a modified version of Bresenham's algorithm (Bresenham, 1965). A* uses a heuristic $h(n)$ to estimate the lowest cost from the current node n to the goal node and adds to this the current path costs $g(n)$. In this implementation, each node represents a square in the 20×20 grid. By selecting the node with the lowest expected costs each time, $min(f(n))$ where $f(n) = g(n) + h(n)$, the algorithm will explore node by node and eventually find the shortest path. The only condition is that $h(n)$ is an admissible heuristic, meaning that it will never over-estimate the total costs.

In order to avoid burning squares, an additional heuristic function is used. This function applies to $g(n)$ and is given function 2.1. When moving from node n to n', $g(n')$ increases by 9999 compared to $g(n)$ if the square at node n' is burning. If the square at n' is not burning, the regular move costs $c_m(n')$ is added to $g(n)$.

$$g(n') = \begin{cases} g(n) + 9999, & \text{if } burning(n') \\ g(n) + c_m(n'), & \text{otherwise} \end{cases} \tag{2.1}$$

Note that this heuristic only helps when determining the path. Once a path has been set, the agents will continue on this path regardless of how the fire spreads.

A modified version of Bresenham's algorithm is used to determine straight paths as it requires minimal computational resources (Bresenham, 1965). The reason for this is that the algorithm only uses subtraction, addition and multiplication, all

of which are relatively easily computed by a CPU. The only modification made to the original algorithm resulted in one additional square being added to each line segment. This results in each subsequent square to square: (x, y) is placed at either (x, y+1) or (x+1, y) instead of (x+1, y+1) or (x+1, y), generating a closed path between the starting- and end-point. Note that this path-finding algortihm does not take burning squares into account.

2.3 RL techniques

As mentioned in section 2.2, the IC uses RL to determine the off-set of the sub-goals. For both RL techniques, the same feature vector was used. This feature vector consists of sub-goal dependent variables and agent-dependent variables. To capture the size, distance and direction of the fire with respect to the agent, the corners of an imaginary square capturing the fire are used. These corners are expressed in an x direction, a y direction and the absolute distance, resulting in 12 variables. To capture the wind direction and speed, the speed of the wind relative to the sub-goal axis is expressed in a single value and added to the input vector. The final parameter used is the number of sub-goals already reached. This provides a measurement on the progress of containing the fire.

Both RL techniques had a training phase of 2500 episodes, where an episode is defined as a single simulated forest fire. CMC utilised these episodes by having 2500 episodes in combination with back-propagation after each episode, while CoSyNE utilised these episodes by having 50 generations with a population size of 50 as well ($50 \times 50 = 2500$). After this phase, the 10 best performing CoSyNE MLPs and the MLPs used in the final 10 episodes of the CMC training phase continue to the testing phase. In the testing phase, all MLPs that reached this phase are tested in the same 10 maps. This results in $10 * 10 = 100$ additional episodes for each run. As the difficulty of controlling the fire is highly influenced by the map that is generated, using the same set of maps for all MLPs should result in high costs for both RL techniques in environments that are "difficult" to control and low costs in "easy" environments. This entire procedure is repeated 20 times for each testing condition (i.e. varying number of agents and path-finding algorithm).

2.3.1 CMC

The Connectionist Monte Carlo (CMC) approach is a combination of every-visit Monte Carlo (MC) estimation of action values as defined in (Sutton and Barto, 2018, Chapter 5) and a Multi Layer Perceptron (MLP).

The psuedocode of every-visit MC estimation of action values is shown in algorithm 2.1. It starts of by initialising an expected return value Q for all state-action pairs (s, a) within state space $\mathcal{S}$ and the set of possible actions $\mathcal{A}$. For all these state-action pairs (s, a), an empty list *Returns* is initialised, meant for storing the returns received by taking action a at state s. Next, a policy π is initialised for all states s in the state space $\mathcal{S}$. After the initialisation process, policy π is optimised for E episodes. The first step of this optimisation process, is selecting an initial state S_0 and action A_0. If this is done in such a way that all state-action pairs have a $probability > 0$, it is easy to see that all pairs are visited if E approaches infinity. Afterwards, an episode is generated from the initial state-action pair (S_0, A_0) by following policy π. The action A_t taken at state S_t yields a reward of R_{t+1} and advances the episode to the next state S_{t+1}. In this state, the policy $\pi(S_{t+1})$ returns action A_{t+1}, yielding a reward of R_{t+2}. This continues until the action is taken that results in reaching the terminal state at $t = T$, meaning that the episode is completed. Now the assignment of return values to state-action pairs starts by first setting the discounted return G to 0. By going over all steps t in reverse order from T to 0, the discounted return value G for each step t is determined by multiplying the old discounted return from the actions taken after action A_t with the discount factor γ and adding the reward from the immediate following step. This value is appended to the *Returns* list of state-action pair (S_t, A_t). The second to last step is updating the expected return value Q of state-action pair (S_t, A_t) by taking the average of all experienced returns at state S_t while taking action A_t stored in *Returns*. Finally, the policy π for state S_t is updated by selecting the action with the maximum expected return at state S_t.

There are however two major shortcomings that need to be solved in order to be able to use MC estimation of action values as a controller in the simulation. The first shortcoming is the necessity

Algorithm 2.1 Pseudocode of every-visit MC estimation of action values

Initialise $Q(s,a) \in \mathbb{R}, \forall s \in \mathcal{S}, a \in \mathcal{A}$
Initialise $Returns(s,a) \Leftarrow \{\emptyset\}, \forall s \in \mathcal{S}, a \in \mathcal{A}$
Initialise $\pi(s) \in \mathcal{A}, \forall s \in \mathcal{S}$
for $e = 1$ **to** E **do**
  Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}$ randomly such that each pair has probability> 0
  Generate an episode from S_0, A_0, following π : $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
  $G \Leftarrow 0$
  for t=T-1, T-2, ..., 0 **do**
    $G \Leftarrow \gamma G + R_{t+1}$
    Append G to $Returns(S_t, A_t)$
    $Q(S_t, A_t) = average(Returns(S_t, A_t))$
    $\pi(S_t) = argmax_a Q(S_t, a)$
  end for
end for

to store the expected returns for all state-action pairs. The memory required to store all of these expected returns increases exponentially with each variable added to the state representation. Another undesirable consequence of using such a look-up table is that all variables in the state representation need to be discrete, otherwise there would be an infinite amount of states. The second shortcoming is that running an infinite amount of episodes is impossible. Therefore a different type of exploration method is needed to assure that sufficiently many state-action pairs are tested.

To bypass the issue of storing the expected return values for all state-action pairs, the CMC algorithm replaces this look-up table with an MLP. By using the state representation as a feature vector for the MLP, similar states will have similar output values of the MLP. By back-propagating the error in the prediction made by the MLP, this prediction will improve as the number of episodes increases. Therefore the effect will be similar compared to having a massive look-up table, while requiring only a fraction of the storage.

Diminishing ϵ-greedy is used to solve the exploration problem. The algorithm of diminishing ϵ-greedy is explained in Appendix A. Diminishing ϵ-greedy yielded good results as exploration method for an agent playing Bomberman which was controlled though Connectionist Q-Learning

Algorithm 2.2 Pseudocode of CMC

Initialise $M(s,a)$
$\pi(s) = argmax_a M(s,a)$
for $e = 1$ **to** E **do**
  Choose $S_0 \in \mathcal{S}$
  Generate an episode from S_0, following π with diminishing ϵ-greedy: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
  $G \Leftarrow 0$
  for t=T-1, T-2, ..., 0 **do**
    $G \Leftarrow \gamma G + R_{t+1}$
    $M \Leftarrow BackProb(M, S_t, A_t, G)$
  end for
end for

(Kormelink et al., 2018). While CMC and Connectionist Q-Learning are in essence different RL techniques, they show strong similarities as they both use a MLP to estimate expected return values of state-action pairs.

Combining the two modifications with every-visit MC estimation of action values, results in the CMC algorithm used in this thesis and the psuedocode is shown in algorithm 2.2. The main difference to regular MC and CMC is that instead of initializing expected return values for each state-action pair, an MLP M is initialised. This MLP processes the state information in the form of a feature vector s and produces the expected return of action a. The policy of CMC is then defined as the action a for which the expected return produced by M for state s is maximum. After the initialisation process, policy π is optimised for E episodes. This time it is not possible to choose a state action pair such that each pair has a $probability > 0$ of being chosen, as there are an infinite number of state-action pairs, while E is finite. Therefore an arbitrary state S_0 is chosen from the state space $\mathcal{S}$. Once the initial state is determined, an episode is generated by following π with diminishing ϵ-greedy as exploration method. Afterwards the discounted reward value G is set to 0. At this point the updating process starts by looping over all time-steps. For each step t, G is updated and M is updated by back-propagation of the error between the original prediction of the return value made by M in state S_t for action A_t and the discounted reward G.

Adaptation to Sub-goal Management. The CMC algorithm described above is implemented in the following way to facilitate sub-goal management. An action is defined as placing a sub-goal with an offset of x, where $x \in \mathbb{N}$ and $x \leq X$ with X being the maximum distance at which the sub-goal can be placed. The state representation used by the MLP to base this decision on, is the feature vector discussed at the start of section 2.3. If a sub-goal is re-evaluated before this sub-goal is reached by a fire-line cutting agent, the old feature vector is replaced by the new feature vector and the action which has not been completed is replaced by the new action. Once the episode has terminated, the total costs of the fire are used as the return value for the entire episode. As the placement of all sub-goals contributed equally to containing the fire, the discount-factor γ is set to 1.

General Parameter Settings. An MLP with a hidden layer of 30 neurons and a sigmoid activation function gave sufficient results in a single agent environment with a 20×20 grid of grass squares. As the tests were conducted on a 20×20 grid, the maximum offset to the center of the fire was 10 and therefore the number of output neurons was set to 10 as well, one for each grid-cell between the center of the fire and the maximum offset. These output neurons had a ReLU activation function, allowing them to predict the expected costs of placing the sub-goal at the offset associated with that output neuron. The other parameters required by the algorithm are summarized in table 2.2.

Table 2.2: Parameters of CMC

Parameter	Value
Learning-discount (γ)	1
Learning-rate MLP (α)	0.05
Starting exploration-rate (ϵ)	0.3

2.3.2 CoSyNE

Cooperative Synapse NeuroEvolution, or CoSyNE for short, is an ENN which evolves at the level of weights (Gomez et al., 2006). The psuedocode is described in algorithm 2.3. It starts of by initialising a population $\mathcal{P}$ consisting of n sub-populations, where n stands for the number of synaptic weights

Algorithm 2.3 Pseudocode of CoSyNE(n, m, Ψ)

```
Initialise P = {P_1, ...., P_n}
for g = 1 to G do
  for j=1 to m do
    x_j ⇐ (x_{1,j}, ...., x_{n,j})
    Evaluate(x_j, Ψ)
  end for
  O ⇐ Recombine(P)
  for k=1 to l do
    x_{i,m-k} ⇐ o_{i,k}
  end for
  for i=1 to n do
    Permute(P_i)
  end for
end for
```

that need to be evolved in the user-specified network architecture Ψ. Each sub-population contains m uniformly distributed real numbers in the interval $[-\alpha, \alpha]$. Thus, $\mathcal{P}$ is represented by an $n \times m$ matrix.

The implementation of CoSyNE used in this experiment uses a set number of generations G instead of the more common approach of stopping when a sufficiently good network has been found, as this allows for a better comparison with the CMC algorithm. At the start of each of these generations, the network chromosomes $\mathrm{x}_j \Leftarrow (x_{1,j}, ..., x_{n,j})$ are initialised from the rows in $\mathcal{P}$. Each of the m resulting chromosomes are transformed into networks by inserting their weights into the corresponding synapses defined in Ψ and are in turn assigned a fitness once evaluated.

The offspring pool $\mathcal{O}$ is created by randomly selecting two parents from the top 25% performing chromosomes for each synaptic weight i, $p_{1,i}$ and $p_{2,i}$. These parent pairs form a new chromosome o by randomly selecting a weight from either $p_{1,i}$ or $p_{2,i}$ for each synaptic weight o_i. Furthermore, there is a 5% chance that synaptic weight o_i is changed to a random value between $[-\alpha, \alpha]$. This process is repeated l times. Afterwards, for each child k in offspring pool $\mathcal{O}$, each weight of the chromosome $o_{i,k}$ is added to $\mathcal{P}$ by replacing the worst performing weight in their corresponding sub-population $x_{i,m-k}$.

To *coevolve* the weights, each sub-population is permuted. This results in new combinations

of weights that would not have been generated through recombination alone. The resulting chromosomes will therefore not only have the weights passed down by the parents, but also some weights from other chromosomes in the population. This will make the network less greedy, as weights that were not part of the parent chromosomes have a chance to reproduce.

Adaptation to Sub-goal Management. It is no longer necessary to predict the costs associated with a specific offset. This makes it possible to reduce the output-layer of the MLP to a single neuron. The activation of the neuron is normalised through a sigmoid function to get a final output out_i for sub-goal i between $[0, 1]$. This output is in turn mapped to an offset r_i by multiplying it with the maximum offset R. This results in the formula shown in 2.2.

$$r_i = out_i \times R_i \tag{2.2}$$

General Parameter Settings. The hidden layer uses a sigmoid activation function as well. Because the MLP for the CoSyNE implementation needs to produce just a single value, the complexity of the MLP can be minimised. Initial tests on a 20×20 grid of grass squares, single agent environment, yielded sufficient results with only 3 hidden neurons. During these initial tests, the parameter settings shown in table 2.3 were established as well. The number of synaptic weights is derived by adding the number of connections between the input- and hidden-layer to the number of connections between the hidden- and output-layer. This results in $14 * 3 + 3 * 1 = 45$ synaptic weights.

Table 2.3: Parameters of CoSyNE

Parameter	Value
# of synaptic weights (n)	45
Weight-spread (α)	3
Size offspring pool (l)	5
Size sub-populations (m)	50
# of generations (G)	50

Table 2.4: Exact values of burn costs, clear costs and move costs for each square category

Category	Tree	Grass	Dirt	Road
Burn costs	50	20	-	-
Clear costs	5	2	-	-
Move costs	5	2	2	1

2.4 Performance Analysis

The performance of the different RL approaches under these different testing conditions was measured by adding the total move costs of all agents to the total damage (or burn costs) of the fire. The exact costs are shown in table 2.4 and are based on the relation between local parameters and square categories shown in table 2.1. Additionally, there is a large penalty of 5000 for each agent that was lost in the fire.

To see whether or not the RL techniques are improving over the course of the training phase, the costs of the training episodes of both CMC and CoSyNE in a single agents environment with Bresenham as path-finding algorithm are compared with a *random walk* under the same conditions on the same map. This random walk is a controller with a policy of placing the sub-goals with a random offset. It does not use any form of RL and will therefore not improve upon this policy. Therefore the difference between the costs produced under random walk and the RL technique should increase as the number of training episodes increases. As the costs made in the environment is effected heavily by the environment itself, the costs curve is highly irregular. To produce a smooth curve, the costs over 250 episodes are averaged. This means that each data point under CoSyNE is the average of 5 generations of each 50 episodes.

To analyse the testing phase, the performance of the MLPs of a single run on a single map is averaged, creating 10 data points for each run. As the set of maps on which the testing phase is conducted is the same for all runs, the results of CoSyNE and CMC in episodes under the same testing conditions can be paired.

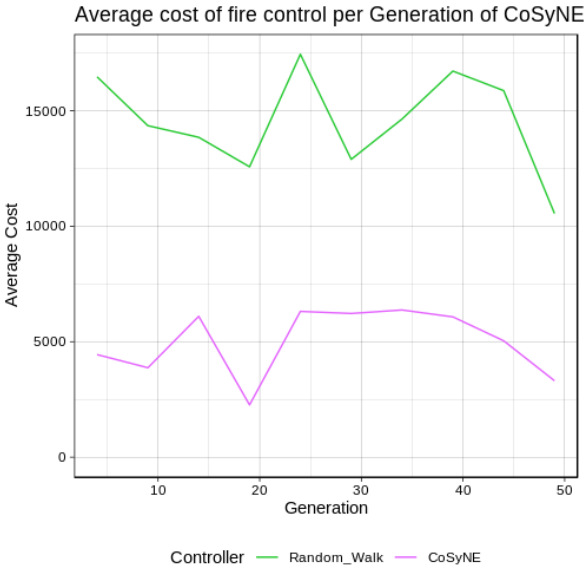

Figure 3.1: Learning curve of CoSyNE controller in a single agent environment with Bresenham path-finding

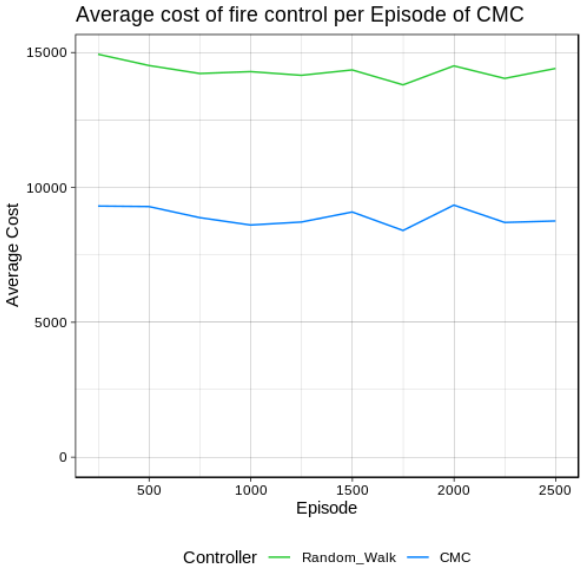

Figure 3.2: Learning curve of CMC controller in a single agent environment with Bresenham path-finding

3 Results

The learning curves of both CoSyNE and CMC in a single agent environment with Bresenham as path-finding algorithm are shown in figure 3.1 and 3.2 respectively. While the difference between Random-Walk and the RL techniques is substantial, it does not seem to increase over the course of the training phase. This puts serious doubts on the effectiveness of the implementation of both RL techniques. However, it does not mean that a comparison between the two RL techniques has no value.

This final comparison is done by analysing the performance in the testing-phase of both techniques over all testing conditions. The results of 20 runs with 10 test-maps are shown in figure 3.3 and show that the median total costs of CoSyNE is lower compared to CMC under all testing conditions. To test whether or not the difference in distributions over all testing conditions is statistically significant, the non-parametric Friedman test is used. This test was chosen for two reasons:

- **Dependency of the data.** As the environment at each episode was the same for all testing conditions, the results from CMC and CoSyNE are paired. In other words, there is a positive dependency within the gathered data.
- **Distribution of data.** The Kolmogorov-Smirnov test on normality showed that 8 out of 8 testing conditions under CMC produced a cost distribution that did not differ significantly from a normal distribution ($D(200) < 0.09, p = N.S.$ for all distributions). The same Kolmogorov-Smirnov test on normality showed that 4 out of 8 testing conditions under CoSyNE did produce a costs distribution that significantly differs from a normal distribution:
  - CoSyNe in a 4 agent environment with A* ($D(200) = 0.16, p < 0.01$)
  - CoSyNe in a 2 agent environment with Bresenham ($D(200) = 0.23, p < 0.01$)
  - CoSyNe in a 1 agent environment with Bresenham ($D(200) = 0.31, p < 0.01$)
  - CoSyNe in a 1 agent environment with A* ($D(200) = 0.16, p < 0.01$)

  Under to other 4 testing conditions, the costs distribution did not differ significantly from a normal distribution ($D(200) < 0.09, p = N.S.$

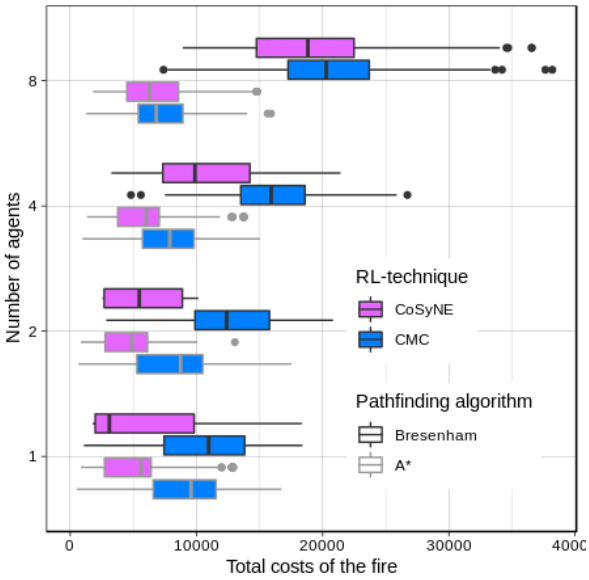

Figure 3.3: Box-plot of the total costs under varying testing conditions

Table 3.1: Results of Wilcoxon Signed-Rank test on the distribution of total costs made by CoSyNE and CMC under varying testing conditions.

Number of Agents	Path-finding Algorithm	Z-Value	p-Value
8	Bresenham	2.21	0.027
	A*	2.00	0.045
4	Bresenham	8.41	<0.001*
	A*	5.28	<0.001*
2	Bresenham	9.80	<0.001*
	A*	8.80	<0.001*
1	Bresenham	7.09	<0.001*
	A*	8.67	<0.001*

*Costs distribution of CoSyNE significantly lower than CMC

for all remaining distributions). An overview of the exact D-values, along with the median and CI of the median, is given in Appendix B. Because some but not all distributions approximate a normal distribution, the assumption of equal distributions does not hold and a non-parametric test is preferred.

The non-parametric Friedman test yielded a Chi-square value of 220.52 which was statistically significant ($p < 0.01$).

For post-hoc analysis within each group of the testing-conditions, the Wilcoxon Signed-Rank test is used. The results are shown in table 3.1. This test is used for the same reasons as for why the non-parametric Friedman test is used for the entire data set. The results of each Wilcoxon rank-sum test is adjusted for multiple comparisons through the Bonferonni correction. Therefore, the result of a comparison is deemed significant if the p-value is below $0.05/8 = 0.0063$.

Table 3.1 shows that the difference in costs distributions was significant under most conditions, the only exception being the 8-agent environment. While the median cost under CoSyNE was lower compared to the median cost under CMC both when using A* and Bresenham as path-finding algorithm, this difference was not significant.

A few observations are worth noting. The first one being that using A* as opposed to Bresenham as path-finding algorithm seems to improve performance of CMC and CoSyNE in a similar fashion. The second observation worth noting is that using more agents does not necessarily result in lower costs. The contrary seems to be the case, as the highest cost distributions are found in the 8-agent environments. This could be explained by an increase in movement costs and agent deaths. Eight agents are likely to produce more movement costs compared to a single agent and having more agents in an environment means that more agents are able die. Figure 3.4 shows that the burn costs without the movement costs and death penalty decreases as the number of agents increases. This time, CoSyNE only produces less costs in the single and 2-agent environment. The difference in the 4- and 8-agent environment seems to be insignificant. Finally, it is interesting to note that using Bresenham as opposed to A* produces less burn-costs.

4 Conclusion

This experiment aimed to find out which RL technique performs better at the task of forest-fire control, CMC or CoSyNE. The comparison was made by looking at the performance of both techniques controlling a simulated fire in a randomly gener-

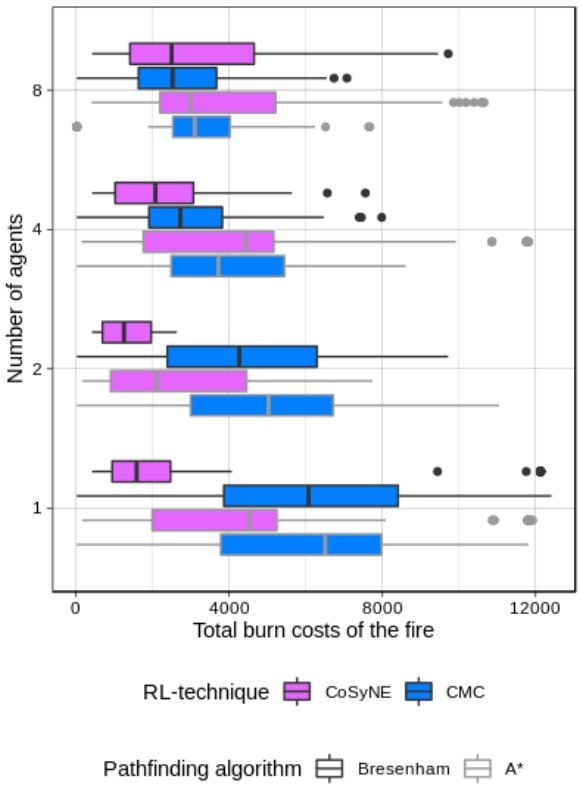

Figure 3.4: Box-plot of the burn costs under varying testing conditions

ated environment. By varying the number of fire-line cutting agents and the path-finding algorithm used by these agents, different testing conditions were induced upon the simulation.

Unfortunately, neither RL technique showed a notable decrease in total costs over the course of the training phase. This suggest that the implementation of these techniques needs improvement. One of these improvements could be:

- **Incremental learning:** It has already been shown that implementation of incremental learning helps in finding solutions for more complex forest fire simulations (Wiering et al., 2005). It is very well possible that the current control problem is too complex and therefore the RL techniques are not able too improve themselves.
- **Extended feature vector:** Another reason for the RL techniques not being able to improve themselves, is that the data they receive does not include all necessary information. Like driving without rear-view mirrors, valuable information could be missing. Without this information, "driving a car" or controlling the fire could be close to impossible.
- **Different parameter settings:** While the parameter settings in the current environment yielded sufficient results in simple, controlled single-agent environment (a 20×20 grid filled with grass squares), it is possible that these settings do not work for more complex environments.

Nonetheless, it is shown that CoSyNE significantly outperformed CQL under most testing conditions, the only exception being the 8-agent environment. Using A* as path-finding algorithm as apposed to Bresenham seems to improve the performance of both CoSyNE and CQL in a similar fashion. Increasing the number of agents however results in unexpected behaviour as this does not necessarily decrease the costs made. This could be explained by the increase of move costs combined with the increase of agents lost in the fire. Looking at only the burn costs of the fire, CoSyNE still outperforms CMC in single- to few-agent environments. Therefore, it is concluded that CoSyNE is the preferred RL technique when dealing with single- to few-agent environments.

This provides an important start when building an RL-based fire management DSS. With CoSyNE at the basis of the fire management DSS, the simulation can be improved to increase the resemblance with nature. The main deviations with a real-life forest-fire control scenario, and possible solutions for including them, are discussed below.

Fire Dynamics. For the sake of simplicity and the saving of computational power, concessions were made in the fire propagation model. Only wind and different fuel types are included in the current fire propagation model. Additional factors that should be included in a true-to-nature fire-propagation model are: the regime of propagation (i.e. is the fire limited to the surface level such as grass and bushes, or has it reached the crown level, burning the tree canopies as well?) and meteorological conditions such as air temperature, air humidity, solar radiation and atmospheric stability. For a

more in depth analysis of the effect of each of these factors, see (Xavier Viegas, 1998).

Priority Areas. While destroyed forests and massive greenhouse gas emissions as a result of forest fires form a major problem, the real problem occurs when the fire spreads towards highly urbanised areas or areas with cultural or natural significance. A prerequisite to be able to respond to prioritised areas is that additional information on the surrounding environment is added to the feature vector used by the RL techniques.

Natural and Preexisting Fire Lines. The simulation included man-made fire lines in the form of roads. In real life situations, fire fighters often utilise natural fire-lines such as rivers or mountain ridges as well to minimize the resources needed to control the fire. The current implementation of the DSS is however not able to utilise the preexisting fire lines to the fullest. The DSS is programmed to set the eight sub-goals in all wind directions of the center of the fire, while a much simpler solution could be found by cutting a short fire line between for instance a river-bank and a road. If the implementation of the DSS can be improved such that it can utilise these fire lines, it is also possible to look at the effect preventive measures such as pre-cut fire lines.

Height Information. The final adjustment recommended to make the simulation as true to nature as possible, is the addition of height differences in the environment. This is valuable information as these height differences can be translated to slopes and these slopes highly influence the propagation speed and direction of forest fires (Xavier Viegas, 2004). The fire dynamics should therefore be updated as well to act accordingly to this additional information. The height information influences the movement costs of the fire-line cutting agents as well. One can imagine that moving uphill requires more energy compared to moving downhill.

Geographic Information System. The improvements stated above would contribute a great deal to the resemblance of real life forest fires. The only downside is the amount of details required when applying the new DSS to the real world. Some DSS import this information from a Geographic Information System (GIS). A GIS stores information on vegetation, urbanisation, climate, water bodies and height differences of environments across the globe. By importing the relevant information from a GIS in case of a newly discovered fire, the new DSS could quickly generate an optimal management strategy for that specific fire.

Acknowledgements

I thank the following individuals for their contributions in designing and constructing the simulation:

- Ivo de Jong, for creating the framework of the simulation and implementing the CoSyNE algorithm.
- Roel Rotteveel, for designing and creating the randomly generated maps.
- Dirk Jelle Schaap and Travis Hammond, for designing and implementing the fire propagation model. Their efforts in parameter tuning are not forgotten either.

References

Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1): 25–30, 1965.

United States Department of Agriculture. *FY 2020, budget justification.* U.S. Gov. Printing Office, 2019.

Shlomo Geva and Joaquin Sitte. A cartpole experiment benchmark for trainable controllers. *IEEE Control Systems Magazine*, 13(5):40–51, 1993.

Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In *European Conference on Machine Learning*, pages 654–662. Springer, 2006.

Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, 2008.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Joseph Groot Kormelink, Madalina M Drugan, and Marco A Wiering. Exploration methods for Connectionist Q-learning in Bomberman. In *ICAART (2)*, pages 355–362, 2018.

Robert Mavsar, Armando González Cabán, and Elsa Varela. The state of development of fire management decision support systems in America and Europe. *Forest Policy and Economics*, 29:45–55, 2013.

Domingo Molina, Marc Castellnou, Daniel García-Marco, and António Salgueiro. Improving fire management success through fire behaviour specialists. *Towards Integrated Fire Management Outcomes of the European Project Fire Paradox*, pages 105–119.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Phillip J van Mantgem, Jonathan CB Nesmith, MaryBeth Keifer, Eric E Knapp, Alan Flint, and Lorriane Flint. Climatic stress increases forest fire severity across the western United States. *Ecology letters*, 16(9):1151–1156, 2013.

Marco A Wiering and Marco Dorigo. Learning to control forest fires. In *Proceedings of the 12th International Symposium on Computer Science for Environmental Protection (UI'98)*, pages 378–388. Metropolis Verlag, 1998.

Marco A Wiering, Fillipo Mignogna, and Bernard Maassen. Evolving neural networks for forest fire control. In *Proceedings of the 14th Belgian-Dutch Conference on Machine Learning*, pages 113–120, 2005.

Domingos Xavier Viegas. Forest fire propagation. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 356(1748):2907–2928, 1998.

Domingos Xavier Viegas. Slope and wind effects on fire propagation. *International Journal of Wildland Fire*, 13(2):143–156, 2004.

A Diminishing ϵ-greedy

Exploration methods are required to guide the learning process of a Q-Learning based controller. These methods provide a means to control the ratio of exploration vs. exploitation. If a controller always chooses the best option according to its current believes, it might get stuck in a local optimum. It could very well be possible that a different option would in reality provide better results, but as the controller never experienced these better results, does not know of this better option and will therefore never choose it (low exploration vs. high exploitation). On the other hand, if the controller is always looking for better options, it will at some point no longer find any improvements. At this point it makes no sense to continue its exploration efforts and is better of exploiting the knowledge already obtained (high exploration vs. low exploitation).

ϵ-greedy is one of the simplest exploration methods. Instead of always taking the action under the policy at state x ($\pi(x)$), there is a chance of ϵ that a random action is chosen. This exploration method is however not GLIE (Greedy in the Limit with Infinite Exploration) as it will always have a chance of not selecting an action according to π. This is undesirable considering that towards the end of a training phase, π will approach the optimal policy π^* and therefore any deviation from π will most likely result in a worse performance.

Diminishing ϵ-greedy solves this problem by gradually decreasing ϵ as the number of iterations progresses. There are several ways to implement this gradual decrease. It can be implemented by defining different phases in the learning process in which the ratio between exploration and exploitation is varied, by multiplying the current ϵ with a value between $(0, 1)$ or by subtracting a fixed number x from ϵ at each iteration until it reaches 0. The latter option is chosen and the formal definition is given is formula A.1. In this formula, ϵ_i is the exploration rate at iteration i. ϵ_{start} represents the starting exploration rate.

$$\epsilon_i = \epsilon_{start} - x * i \tag{A.1}$$

The reason for implementing this version is that it adds only one additional parameter that needs tuning, namely ϵ_{start}. By defining $x = \frac{\epsilon_{start}}{I_{max}}$, where I_{max} is the maximum number of iterations in the training phase, ϵ reaches 0 at the end of the training phase.

B Overview of the results

Table B.1 shows the median, the 95% CI of the median and the results of the Kolmogorov Smirnov test on normality of the costs distribution under each testing condition with varying RL techniques. The median is preferred over the mean of the distributions as the Kolmogorov Smirnov test showed that not all distributions are approximately normal. The CI of the median is determined through the use of a normal bootstrap with replacement over 10.000 resamples.

Table B.1: The propability of being equal to a normal distribution according to the Kolmogorov Smirnov test, the median distribution and the 95% CI of the median

Number of Agents	Path-finding Algorithm	RL technique	D(200)	p-Value	Median	95% CI
8	Bresenham	CoSyNE	0.09	0.09	18 832.4	[17637 , 20314]
		CMC	0.06	0.39	20 282.4	[19402 , 21290]
	A*	CoSyNE	0.09	0.10	6278.45	[5677 , 6798]
		CMC	0.08	0.13	6817.3	[6401 , 7193]
4	Bresenham	CoSyNE	0.08	0.15	9883.35	[8991 , 10650]
		CMC	0.04	0.96	15 921.55	[15219 , 16449]
	A*	CoSyNE	0.17	<0.01	6014.4	[5598 , 6538]
		CMC	0.05	0.71	7889.6	[7339 , 8466]
2	Bresenham	CoSyNE	0.23	<0.01	5471.95	[2815 , 7505]
		CMC	0.05	0.66	12 391.15	[11586 , 13126]
	A*	CoSyNE	0.09	0.09	4861	[4185 , 5908]
		CMC	0.05	0.59	8742.7	[8125 , 9433]
1	Bresenham	CoSyNE	0.31	<0.01	3103.85	[2713 , 3560]
		CMC	0.07	0.25	10 968.95	[10002 , 11908]
	A*	CoSyNE	0.16	<0.01	5613	[5350 , 6113]
		CMC	0.06	0.55	9572.9	[8970 , 10251]