Gravitas

Gravwell, Artificial Intelligence, and You

Jonas A. Hultén[1], Jappie Klooster[2], Eva Linssen[3], and Deliang Wu[4]

[1] (5742722) jonashu@student.chalmers.se
Chalmers University of Technology, 412 96 Göteborg, Sweden
[2] (5771803) j.t.klooster@students.uu.nl
[3] (3902749) e.linssen@students.uu.nl
[4] (5622182) deliang8307@gmail.com
Universiteit Utrecht, 3584 CS Utecht, The Netherlands

Abstract. In this report, we present an attempt at creating intelligent agents for the board game "Gravwell: Escape from the 9th Dimension." We give a brief explanation for the complexity of the game, in spite of its simple rules. We then construct a computerized version of the game as well as five agents – decision-tree, neuro evolution based, Q-learning, and two forms of random play – to play the game. We show that the decision-tree and Q-learning agents can outperform the random agents. The decision-tree agent, in particular, proves to be particularly strong. We end the report by discussing the known shortcomings and potential future developments of our agents.

1 Introduction

How does one become good at board games? Is there some way to become the best at, say, Monopoly?

In this report, we present and discuss our attempt to create agents to play the board game "Gravwell: Escape from the 9th Dimension." We attempt this by building a software analog of Gravwell and designing several different forms of agents – some learning and some not – to play it. The goal of the agents is to find a good strategy capable of, at the very least, beating an opponent which plays random cards.

In Section 2 we will briefly explain the rules of Gravwell and explain some of the complexity in the game. We also present the theory behind our agents. In Section 3 we will explain how we built "Gravitas", our implementation of Gravwell and the agents that play it. In Section 4 we will explain the tests we used to measure agent performance, and in Section 5 we will present the results thereof. In Section 6 we will discuss what we have learned from this project and put forward ideas for future work before closing the report in Section 7.

2 Theory

Before we look at what we did, it is important to provide a solid founding in theory. In this section, we will begin by presenting the rules of Gravwell and the

complexity of the game, then moving on to introducing the theory behind the agents we developed. We will begin by looking at the theory behind our agent based on reinforcement learning, then decision trees, and finally neural networks.

2.1 Rules of the game

Gravwell is a game that is played by two[5] to four players [8]. The game is played on a board with 55 tiles, where tile 0 is called the "Singularity" and tile 54 is called the "Warp Gate." Each player gets a ship which starts the game in the Singularity. There is also a non-player ship placed on tile 36. If there are three or more players, another non-player ship is placed on tile 26. Throughout this report, we will be calling the non-player ships "hulks." The object of the game is to move your ship from the Singularity to the Warp Gate. The first player to reach the Warp Gate wins.

The structure of the game is straightforward. The game is played in six *rounds*, each of which consists of a *drafting* phase – where players draw cards – and six *turns*. Each turn consists of a *card-picking* phase – when cards are played from the hand – and a *movement* phase – where the played cards are resolved to move the ships on the board. Since the game has a fixed end, it is possible that no player has reached the Warp Gate by then. If that is the case, the player who is closest wins.

A player moves their ship by using a *fuel card*. There are 26 cards and each card has three attributes: *type*, *value*, and *name*. The name is unique to each card and plays an important role, since cards are resolved in alphabetical order – more on that later. The type signifies one of three functions the card has: a *normal* card moves the player *towards* the closest ship, a *repulsor* card moves the player *away* from the closest ship, and a *tractor* card moves *all other ships* – including hulks – toward the player. The number of tiles moved is determined by the value, which is an integer between 1 and 10.

There are, of course, some caveats. With the exception of the Singularity, there can only be one ship per tile. If a ship stops in a tile with another ship in it, the first ship continues moving one tile in the direction it was going. This is repeated as many times as needed until the ship gets a tile of its own. In addition, when determining which ship is closest, all ships in the Singularity are ignored. If a ship has an equal distance to ships both behind and in front of it, the direction with the most ships is the direction of travel. If the number of ships in both directions is also equal, the ship is stuck and cannot move.

Each round starts with a drafting process, when six cards per player are placed on the table in stacks of two with one card visible. The players then take turns drafting a stack into their hand until all stacks are gone. The turn order is determined by distance from the Warp Gate – furthest drafts first. If two or more players are in the Singularity, their order is randomized[6].

[5] The official game rules does allow the game to be played alone, but this involves simulating an opponent which to us means there are two players.

[6] The official game rules say that, in this case, the youngest player drafts first. We randomize since "age" doesn't make sense for computer agents.

After drafting, each player plays a card from their hand face-down on the table. Once all players have played, the cards are turned over at the same time to reveal what was played. The cards are then *resolved* – meaning its effect is applied – in alphabetical order, as previously mentioned. This process of play and move is repeated until all players have played their six cards on hand.

If a player is unhappy with their play – say, their card isn't resolving as early as they had hoped – they can use their "Emergency Stop" card. This card is only available once per round and, when used, cancels the effect of the player's card. Emergency Stop cannot be used to prevent being moved by another player's tractor card.

2.2 Game complexity

In spite of it its relatively simple rules, Gravwell is theoretically complex. It may already be clear that the number of possible games is staggeringly large, but before continuing, we will formalize the complexity of the game with regards to board permutations and play sequences.

Board permutations The board contains information about the position of all player and non-player ships. Each tile of the board – except for the singularity – can contain one ship. We want to distinguish between permutations where player ships may have the same positions but in a different order – e.g. player A being on tile 4 and player B on tile 3 is a different permutation from player B being on tile 4 and player A on tile 3. Since the hulks are indistinguishable from each other, we do not need to distinguish states where only the permutation of the hulks differ.

The formula to determine the number of board permutations can thus be written

$$B_p(n) = \frac{54!}{k!(54-n-k)!} + n!, \quad \begin{cases} \text{if } n = 2 \text{ then } k = 1 \\ \text{if } n > 2 \text{ then } k = 2 \end{cases} \tag{1}$$

where n is the number of players and k is the number of hulks. This means we get the following number of board permutations:

Play sequences The number of possible plays in each game is quite clearly tremendously large. However, since cards are drafted and played each round, independently of other rounds, we can look at the number of possible plays per round instead. This is not to say that the number of possible play sequences in a round is not tremendously large, but it is less than for a complete game.

Firstly, we have to consider that there are 26 cards in the deck. Out of these, each turn uses n cards. The first turn is thus from 26 choose n, the second is from $26 - n$ choose n, and so on.

For each turn, players also get the choice to use Emergency Stop or not. This adds to the number of play sequences in a strange way, since it depends on the

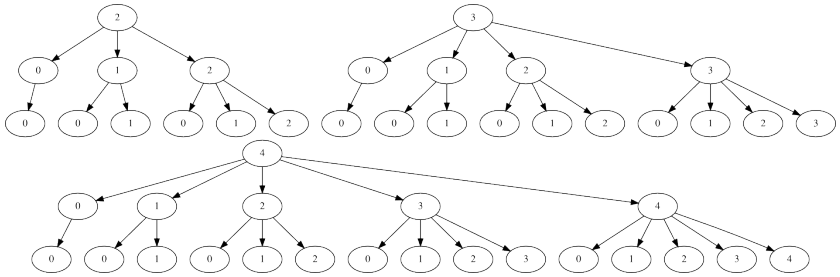

Fig. 1. Tree-graph of number of possible Emergency Stop play sequences for 2, 3, and 4 players and two turns. The numbers indicate how many players are capable of playing Emergency Stop in each state.

number of Emergency Stops available, rather than the number of players. In the first turn, there are $n+1$ outcomes: $n, n-1, \ldots, 1, 0$. This holds for any value of n. In the second turn, however, the number of outcomes becomes more complex – figure 1 shows how the number of outcomes expands rapidly. For the second turn, the number of outcomes can be given by $\sum_{k=0}^{n}(k+1)$, but this, again, does not hold for the third turn.

In order to generalize this, we can use the fact that the number of outcomes for $n = 2$ is the series of triangular numbers, $n = 3$ is the series of tetrahedral numbers, and $n = 4$ is the series of pentatopal[7] numbers. These are defined as $\binom{t+2}{2}$, $\binom{t+3}{3}$, and $\binom{t+4}{4}$ respectively, where t is the turn number. With this, the number of Emergency Stop play sequences after t turns and n players can be written as

$$ES_s(t,n) = \binom{t+n}{n} \tag{2}$$

Having defined that, we can finally define the general formula for the number of possible play sequences as

$$P_s(t,n) = \binom{t+n}{n} \prod_{k=0}^{t-1} \binom{26-kn}{n} \tag{3}$$

By this, we get the following numbers of play sequences:

It should be noted that, while these numbers are clearly very large as is, we do not differentiate between sequences based on who played which card. To do so, we must again look at permutations. This means the first turn is 26 *permute* n, then $26-n$ permute n, and so on.

For Emergency Stop play, the number series is now rather simple. For $n = 2$ – as shown in Figure 2 – it is the series of squares, for $n = 3$ it is the series of cubes, and for $n = 4$ it is the series of tesseractic[8] numbers. Thus, it can be generalized as $(t+1)^n$ outcomes after t turns and n players.

[7] The pentatope is the 4-dimensional form of the tetrahedron.
[8] The tesseract is the 4-dimensional form of a cube.

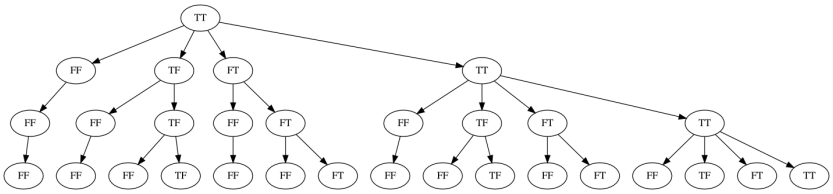

Fig. 2. Tree-graph of number of possible Emergency Stop play sequences for two players and three turns. The letters indicate whether or not a player can (T) or cannot (F) play Emergency Stop in each state.

Finally, using this, we can define the general formula for the number of possible player-distinct play sequences as

$$P_\delta(t,n) = (t+1)^n \prod_{k=0}^{t-1} \frac{(26-kn)!}{(26-(k+1)n)!} \tag{4}$$

which gives the following values:

2.3 Q-learning algorithm

Q-learning is one of the most popular algorithms in reinforcement learning [7]. It is an on-line learning approach which models the learning system including agents and environment. Each agent in the system has its own action set and chooses to play an action from the action set at each discrete-time step. The environment is a finite state world for the agent. The agent will get a reward from the environment by playing an action at any state, and move to another state. Thus, it is a Markov decision process. The task of reinforcement learning is to maximize the long-term reward or utility for the current agent.

In q-learning the decision policy is determined by the state/action value function Q which estimate long term discounted cumulative reward for each state/action pair. Given a current state s and available action set A, a Q-learning agent selects each action $a_i \in A$ with a probability given by the Boltzmann distribution:

$$p(a_i \mid s) = \frac{e^{Q(s,a)/T}}{\sum_{a_i \in A} e^{Q(s,a_i)/T}} \tag{5}$$

where T is the temperature parameter that adjusts the exploration of learning. The agent then executes the selected action, receives an immediate reward r, and moves to the next state s'.

In each time step, the agent updates $Q(s,a)$ by the following update function:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \lambda \max_{b \in A} Q(s',b)) \tag{6}$$

where α is the learning rate, λ is the discount factor of long term payoffs, and r is the immediate reward of executing action a at state s. Note that a specific

$Q(s, a)$ is only updated when taking action a at state s. Selecting actions with a Boltzmann distribution ensures that each action at each state will be evaluated repeatedly and the state/action values will converge to their real value after sufficient updates.

The procedure of Q-learning algorithm is illustrated in Algorithm 1.

Initialize Q(s,a), $\forall s \in S, a \in A(s)$, arbitrarily, and Q(terminal-state,.) = 0
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q
    Take action a, observe r,s'
    $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \lambda \max_{b \in A} Q(s',b))$
    $s \leftarrow s'$
  until s is terminal

Algorithm 1: Q-learning procedure

2.4 Decision tree

A decision tree is a tree with in its nodes input features, in the edges coming from these nodes possible values of these input features, and answers to the decision query in its leaves. By traversing the tree, based on the feature values of the input, a decision can be reached [11]. In our implementation of the decision tree the nodes are filled with state features in the form of Boolean expressions. By answering these 'yes or no' questions based on the current state, starting from the root node and following the tree until a leaf is found, a decision is reached.

2.5 Neural network

Neural networks approximate the functionality of the biological nervous system. In theory they can approximate any continuous function[1]. Neurons or *nodes* are the units that do fundamental processing, when connected together they form a full network. One can think of the network in terms of layers. The first layer is the input layer which can be seen as the "sensor" that receives input from the world. The last layer is the output layer, which is used by the system that contains the network. The layers in between are called "hidden layers".[2]

Darwinian evolution The key concepts of a Darwinian system are:

- Offspring generation
- Competition
- Selection

Offspring can be generated in several ways, either through crossover, which uses multiple parents to create children, or through making plain copies. In both cases mutation can be applied. In our project we will use copies that are mutated.

Competition is used to assess the fitness of the offspring and parents. Usually this is the phase where one compares how well individual solutions work against a problem.

Selection then reduces the population size by keeping only the fitter members. This can be done in multiple ways, such as tournament, truncate or proportional. Where tournament selection creates "tournaments" that only the strongest survive, truncate selection just orders the population on score and deletes the weakest members. Finally, proportional selection allows proportionally more members to live based on their performance. [10]

Neuro evolution Neuro evolution combines the idea of neural networks with Darwinian evolution. Traditionally, neuro evolution methods used fully connected networks and would evolve the weights of connections to approximate functions. Evolving topology however allows you to bypass the problem of guessing how many hidden nodes are required.

The NEAT method starts with a fully connected network and has several mutating operations to modify the network. Every mutation has a unique *innovation number*, which allows for crossover [4].

FS-NEAT goes a step further and uses a sparsely connected initial graph. This allows topology mutations to figure out which of the connections are necessary and which are not [5]. In a more simplified form of FS-NEAT, Jacob Schrum [6] showed that even output nodes can be decided by the evolution process.

3 Implementation

In this section we will discuss how we implemented the game of Gravwell in Python. We will also discuss how we created the agents that can play the game of Gravwell autonomously, and how theory was applied to practice.

3.1 Software architecture

The core idea behind our architecture is to separate the control flow, data and representation, similar to how for example the MVC design pattern works.

On the left side of Figure 3 it is clearly visible that the player controllers are separated from the rest of the game. They are almost exclusively known by the factory (with the exception of the evolution, which needs to evolve the neural player). On the bottom we can see programs that "use" the main program in various ways. The upper right side of Figure 3 contains data structures. In the center of Figure 3 is core of the game, consisting of the factory (which handles initialization), the game manager and the engine. The factory is a separate module because initialization of a game became quite complex. The game manager handles the gameplay rules of Gravwell and polls the player controllers for their chosen actions.

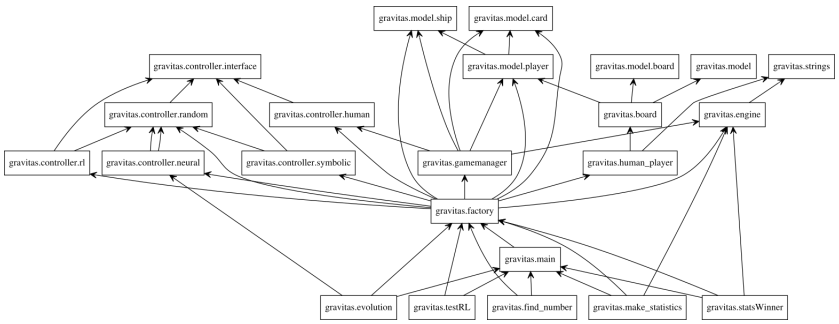

Fig. 3. Architecture overview

Our architecture is single threaded. If a player controller is not able to play yet, it can return None instead of choice. For example the human_player controller first needs to wait on the correct input, which is handled on the same thread elsewhere, so in this case the controller should return None and let the events be handled. Therefore if None is received as a choice the game manager will not progress the game, and the player controller will be asked to make a choice again later. The single threaded design was a very conscious decision because implementing parallelization is hard. We wanted to focus on the AI and the game itself, rather than race conditions and deadlocks.

What is not visible in Figure 3 is how the human controller stands in contact with the human_player class. The latter handles creation of form windows on which the player can select decision input. There is no direct link between the classes because the factory passes a window handle to the controller on construction. This means the human controller itself owns the window, and can therefore directly figure out what actions the human wants to play.

It should also be noted that the game manager is in control of the game state. This is to prevent cheating by the controllers, which is something that was a recurring problem. Since initially the controllers were part of the game state, every player and their hand was visible to each player. This allowed other controllers to read each others' mind, which we did not want to allow.

3.2 Interface

During the initial development of Gravitas, testing hinged on us being able to play the game ourselves. For this reason, we developed a simple graphical interface visible in figure 4. Its primary purpose was to allow the human player to interact with the game in a sensible way but also proved useful for witnessing all-agent runs of the game. Indeed, during the early stages of development, running Gravitas required using the graphical interface – it was only later that a "headless" mode was added, allowing the game to be run *far* faster.

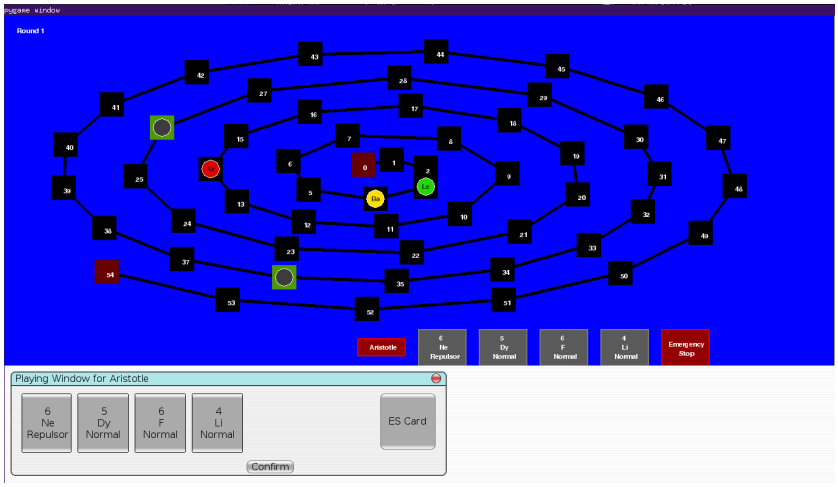

Fig. 4. Screenshot of the interface

3.3 Neuro evolution agent

Neuro evolution can be split up in two distinct parts. The first part is creating a malleable neural network that can play the game. This neural network does not need to be good yet, it only needs to be able to play the game. The second part is doing the evolution on the neural network; here we form opinions about how well the network performs and select changes to improve it.

Since the game of Gravwell consists of several distinct phases (card-pick phase, drafting phase, and movement phase), we would need to either evolve distinct networks for each phase, evolve the outputs nodes in a similar fashion to Jacob Schrum [6], or just pick a phase and hope its important enough. We chose to do the latter, because it can be implemented faster and time was a concern.

Neural Network It is easy to create a neural network for Gravwell because the available input count is well known. In fact, the number of unique input configurations is somewhat on the low side. For example, you have the amount of cards in a hand, which is bounded to a maximum of 6. Then there are the player positions, which can be pre-determined (and are in the original game of Gravwell), and finally the hulk positions. To make this information useful and easily processable by the network we map all the input into numbers. For the player/hulk positions this is easy, because these values are already numerical. For the cards this is a little more difficult, because they each have three distinct properties:

1. Play order $o \in \mathbb{N}$ where $0 < o < 27$, is a number which decides if a card is executed before another card.

2. Power $p \in \mathbb{N}$ where $0 < p < 11$, is a number that indicates the power of a card. What this power represents depends on the type.
3. Type $t \in \mathbb{N}$ where $0 < t < 4$, is a number which indicates the type of a card. This is discussed more thoroughly in the rules 2.1.

Every card in an agent's hand maps to three distinct numerical values for the input nodes of the neural network.

Note that in the board game, type and play order are not indicated by numbers, instead colors and letters are used respectively. For the neural network, however, these are mapped to numbers.

Another important notion is that the player position will get a special input node, in that it will always be the same. This should allow the neural evolution to learn how to do relative comparisons.

The output nodes will be modeled after card "preference". By preference we mean the available card with the highest preference will be played. It is up to the neural network to decide how the preference will be distributed over the cards, and it can do it trough the output nodes. We considered to let all outputs be a single node that would specify card index, however there were doubts if a neural network could figure out the semantic meaning of something like a modulo operation for card index.

To do a concrete implementation of a neural network we use a software framework called TensorFlow. The reason for this choice is that TensorFlow promises speed[3], which is extremely important for doing evolution as we shall see later.

It does need to be noted that TensorFlow itself is not aimed at doing evolution through topology changes (although our implementation proves you can). Instead, it is intended for the field of machine learning, where the networks are designed rather than evolved. Because of this the graph provided by TensorFlow is immutable once constructed. To work around this constraint, we decided to use a builder pattern, similar to a string-builder, for example. The builder object keeps track of a symbolic representation and can produce a concrete graph instance once modification is done.

This symbolic representation can also used to store the neural network to a file after evolving. Although TensorFlow can also write to file, it is more aimed at storing "Variables" rather than complete graphs. So instead, we use Python's pickling library [9].

Evolution Evolving the network is done by means of a simplified FS-NEAT method [6]. The initial population consists of random inputs being the outputs. So for example a player position may be directly be a preference for a card to be played.

To create an offspring we create plain copies of the parents and apply mutation. During the mutation step we select an output to be modified and add a randomly chosen operation to it. It is known how many arguments the operation requires, so if the selected output has not enough as input for the new operation,

we keep on adding other nodes that come strictly a layer *before* the current one. This ensures a feed forward network.

Competition has been done is several different ways, since there were issues with finding the right scoring mechanism. The first attempt pitted the neural networks against random AIs. The performance metric used was the final position. However doing this for 4 players requires a run count $r = 280$ to get an error margin of $l = 0.05$ (see 4.2). This proved to be to many runs to do any effective mutation upon.

We then tried pitting the mutated copies against their parents, basically doing a repeated death match for r run count amount of times. This had the advantage that scores aren't cached and we can evolve more quickly. Now we can put three random mutations and the original against each other and apply heavy selection. This tournament supports the results of comparing performance 4.2 better than the previously discussed methodology, because the averages still tended to vary quite heavily. Parents that had "lucky" averages would stagnate innovation because results were cached. However this methodology was even slower, because the neural networks take quite some extra time to execute compared to random AIs.

The final attempt was something which is practically foolproof. We pre-seed the tournaments each generation, so that the random players, the cards, and the play order would be the same in every competition. Then we'd let the parents and children play against these pre-seeded random AIs. In this method, the only thing differing between tournaments would be the playing strategy. Thus, we didn't have to worry about stochastic differences (l), since the stochasticity would be the same.

Mutation The process of mutation starts by selecting the output that is going to be mutated. This gives us a position in the graph to start with. Then its decided if a node needs to be added or deleted to the network. This decision is made randomly.

If a node needs to be added, we first need to calculate the new position. This is done with help of the previously selected output node, we say its the selected output node layer +1, and then get the node count for that layer to find the index of that layer. After that the new operation to add needs to be decided, where the available operations are $\{add, subtract, multiply\}$. Division and modulo used to be part of the operation family, however they tended to cause division-by-zero errors, which TensorFlow translates into segfaults. After operation selection the inputs need to be decided. We already know one input, namely the selected output node. The other inputs are added randomly; as long as they're in a layer before the new node, they can be selected.

If we want to delete a node, the first thing we do is check if the selected output node is a basic input node, such as "ship position" or "card 1 type". We can't delete those, so instead we select a another input node as the new output. If this is not the case, we go trough the entire input tree of the selected output node and add it to a list. The output node is also added to the list. Then we

randomly select one node from the list to delete and go trough the following cases:

- If the output node is selected for deletion, we select a random input as the new output.
- If an input node is selected for deletion, we go trough all the used-by operations and randomly attach other input nodes instead of the current one. Finally we reset the used-by attribute to an empty array. Thus, after "deleting", the input node is no longer used, but still available for later mutations. This entails that input nodes are never deleted.
- If a hidden node is selected, go trough all used-by nodes, and attach random inputs of the hidden node to them instead of the current node. Then we delete the node from the graph.

3.4 Q-learning agent

There are three decision making processes for a player in Gravwell: drafting cards, playing cards, and using the emergency stop card. It is hard to evaluate the decision when drafting because there is no immediate reward. So, in the current version of the Q-learning agent, we only implemented q-learning for playing cards and using the ES card.

Learning to play cards There are at most six rounds in a game and six turns in each round. The agent needs to learn to play a card at each turn. The goal is moving towards the Warp Gate as far as possible. The key points of Q-learning are:

1. State representation, which represents the game environment with finite states.
2. Action representation, which represents the action set for each player.
3. Reward function, which calculate the reward for each $\{state, action\}$ pair.

It is rather complex to fully represent the whole state space for each player (see section 2.2). We do not have enough time and resources to train such huge state space in the Q-learning model. To simplify the state representation function, the current version of the Q-learning agent only considers the relative position between itself and its closest opponents in both directions. The reason is that the reward of each move is more likely to be affected by the closest opponents rather than opponents far away.

The simplified state representation for the Q-learning agent is illustrated in Figure 5. The state representation for the agent is the visual field after applying a sensation limitation, the agent can "see" the exact positions of opponents in the range of $-10 \sim 10$. There is also an integer for showing no ships within that distance limit, and another integer meaning there are no ships in that direction even outside of that limit. This adds up to 12 options per direction, which there are two of, so 12x12 options in total. Therefore, the size of simplified state space

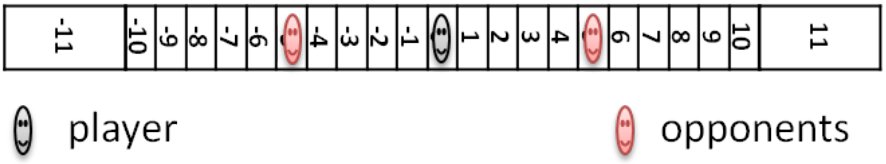

Fig. 5. The simplified state sensation for Q-learning player

is $12 \times 12 = 144$. To further simplify the implementation, the position of hulks were also be ignored because the hulks are only moved when a player plays a tractor card.

Because there are six turns in a round, the player may have different strategies at different turns, even though the relative position to the opponents is the same. Thus, the possible state space may need to be connected with the turn number. The size of the state space is then extended to $6 \times 144 = 864$. So, we have two state representations: 144 states and 864 states. We will further discuss the performance comparison between them in section 4.

When learning to update Q values, there are 26 possible cards that can be played. So, the size of action space is 26. When choosing a card to play, the action set is the cards in hand. Because the game is an imperfect information game, the cards held by opponents and which card an opponent will play are unknown. Also, it is hard to use fictitious play to predict opponents' strategies. So, we make the Q-learning agent ignore the actions taken by the opponents by treating them as non-stationary environment.

The player chooses to play an action a at state s, we need to calculate a reward r. If the player moves forward, it gets positive reward, otherwise it gets negative reward. We tried two different reward functions:

1. Calculate the reward for each turn, the immediate reward for each turn was decided by the direction and distance of movement.
2. Calculate rewards for each round, the immediate reward for each turn was set to 0, and the total reward of each round is accumulated at the end of each round as the reward for the last state. So in this reward function, only long term reward will be considered.

Let l be the spaces the player's ship moves, d be the direction of player's ship moves, where -1 is away from the Warp Gate and 1 is towards it, the reward r is given as:

$$r = l \times d \times \gamma \tag{7}$$

where γ is a negative reward reinforcement factor such that, when $d = 1$, $\gamma = 1$ and when $d = -1$, γ can be set from 1 to 10.

At each round, a player has only one opportunity to use an Emergency Stop(ES) card. It is best to use an ES card when the player's ship has to move backward. The player needs to learn how to balance between immediate rewards

and long term rewards, because using an ES card for going back 2 tiles is a waste, since you may need to use it later, but for 10 tiles its probably a good investment. One of the Markov Decision Processes for learning to use ES card is illustrated in Figure 6.

The state of learning to use ES cards can be represented by the tuple $\{turn, moveDirection, moveDistance\}$. The range of turn is $1 \sim 6$. $moveDirection$ is the direction the player will move after resolving, and has 2 possible values $\{forward, backward\}$. $moveDistance$ is the tiles the player will move after resolving, the range of $moveDistance$ is $1 \sim 10$. So, the total size of the state space of learning to use ES card is $6 \times 2 \times 10 = 120$.

For playing the ES card there are only 2 actions available in the action space: $\{useES, donotUseES\}$. If the player chooses not to use an ES card in current turn, then the immediate reward is 0. If the player chooses to use an ES card, then the immediate reward is decided by the current state $\{moveDirection, moveDistance\}$. Assuming $moveDirection = -1$ if the player's ship will move backward, and $moveDirection = 1$ if the player's ship will move forward. Then, the reward r for the action of using ES card is $r = -moveDirection \times moveDistance$.

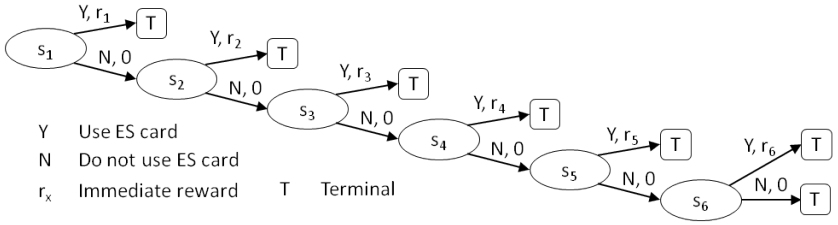

Fig. 6. One of Markov decision processes for learning to use ES card

3.5 Decision tree agent

The decision tree is an easy way to let an agent make choices based on properties of the game state. The state space of Gravwell is rather big and as a result the amount of possible decisions based on state properties is also huge. Because of this we tried to keep the decision trees simple, just catching the more general strategies. In this section the reasoning behind the design of the decision trees for the drafting phase, card-picking phase and the move phase are explained.

Drafting cards During the drafting phase the cards that are to be used in the card-pick phase are chosen. There is an intuition that having a varied hand might be helpful, considering the future game states are not predictable. This unpredictability is caused by the configuration of the opponents' hands only being known for half of the cards, and only after the drafting phase.

To implement this intuition into the decision tree and to make it both simple and useful for general cases, we have divided the possible cards into 5 non-overlapping sets of cards. The first division is based on types, then the group of normals is subdivided based on values. This means there are the Tractors, Repulsors, High normals, Low normals and Remaining normals. Currently a normal card c is considered to be High if $Value(c) \geq 8$, Low if $Value(c) \leq 3$ and, as follows from the name, Remaining if $3 < Value(c) < 8$, but it could be argued to change these boundaries.

We choose a card only if the hand holds no card of the same group yet. Since there are more normal cards than tractors or repulsors, all stack choices have a relatively high chance to have a normal card as the hidden card. Because of that, card-stacks featuring these special types are preferred. Since our goal is a varied hand, we also want both high and low valued normal cards. When the special cards are gone from the drafting field, these high and low normal cards are the preferred choice. As a low value card can do less harm in unfavorable situations, between high and low, the latter is preferred. When choosing from within a group early (low alphabetical value) cards are preferred, because as they resolve early they help in keeping the future state as predictable as possible during the card-pick phase. All these considerations together results in the decision tree as shown in Figure 7.

Note that this might not result in the varied hand we wanted, since the choosing of cards is depended on the available card choices and also on the choices of the opponents. Special cards might not be visible on the stacks, or opponents might choose them before the player can.

Playing cards A general notion of playing cards is to use a normal card when the closest ship is in front of you, and a repulsor when its behind you. Both these card choices are intended to propel the player ship forward. It is not that simple however. The state of the game at the moment that your played card gets resolved is almost surely different from the current one, with only exception being the card "Ar" which is always resolved first. As you only know half of the

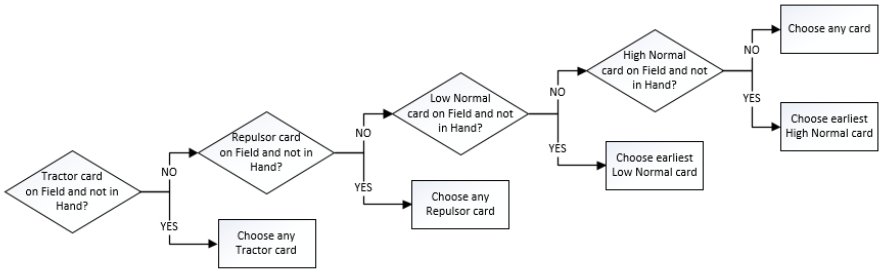

Fig. 7. Decision tree for choosing a card stack during the drafting phase.

cards your opponents have, you cannot feasibly predict these future state. This makes it difficult to handle within the decision tree and as such is not considered. Instead, we have chosen to let the drafting phase prefer early-resolving cards when choosing within a card group. This only helps us a little in that account, because half of the cards you get are not taken by choice.

This general notion creates some edge cases cases we need to take into account. For example, what if the ship is stuck? In this case you might want to use a tractor, but those are scarce. Also, the resolution order might mean you are not stuck anymore at resolve time. Our choice not to take future states into account forces us to make a random choice here.

Another example of an edge case is when you are about to move backwards but have no repulsor card in your hand. Ignoring future state-changes this can be solved as the follows; either a tractor is chosen if the player has one in their hand, or the lowest normal card is used. Solving forward movement but owning no normal cards works similar: either a tractor (if available) or the lowest repulsor is chosen.

These tree design choices result in a decision tree as shown in Figure 8. Of course, if the chosen card turns out to have a very unfavorable effect in the move phase, the emergency stop can be used to negate that effect.

Use of Emergency Stop Choosing whether or not to use the Emergency Stop is based on move direction and card type. If the effect of a card means the ship position is decreased, this is considered a negative effect and the Emergency Stop is used. This results in a decision tree as shown in Figure 9.

3.6 Random agents

For reference purposes, we also developed two random agents. The implementation of these is very simple: every choice it makes is drawn from a uniform random distribution. This means random cards are drafted and random cards are played.

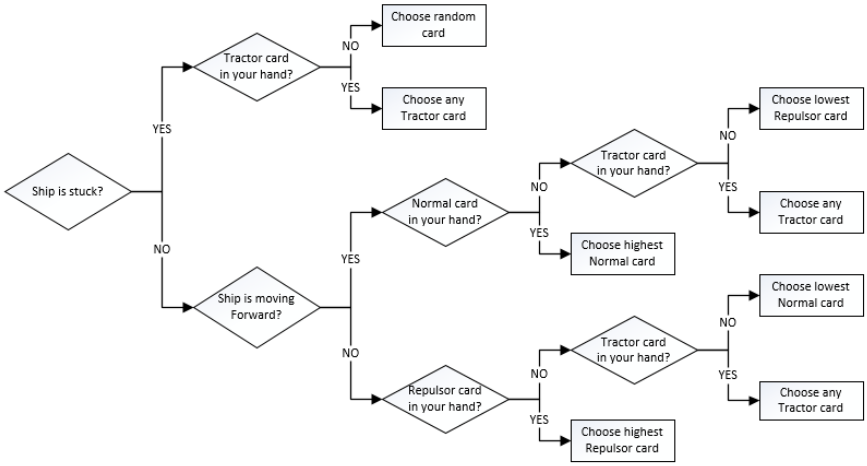

Fig. 8. Decision tree for choosing a card during the card-pick phase.

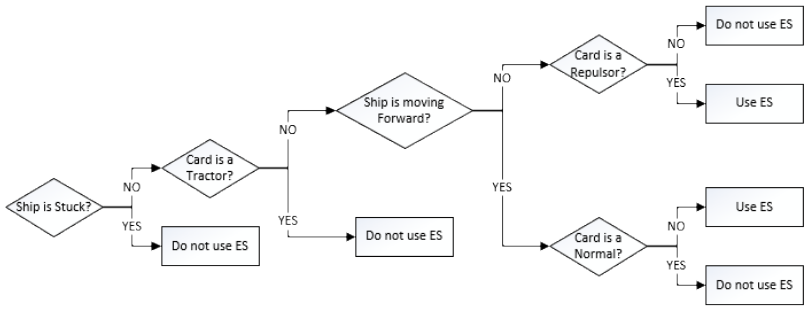

Fig. 9. Decision tree for choosing whether or not to use the Emergency Stop during the move phase.

The difference between the two agents is that one will also play Emergency Stop with a 50% chance, whereas the other will never play Emergency Stop. This second agent was deemed necessary after the first proved too disruptive to evolution of the neural network. This is the case since the neural network agent builds on the random agent, but only applies its logic during the card-pick phase. Using the first random agent, a well-chosen play by the neural network could be randomly discarded by the underlying random agent. To eliminate this problem, the second random agent was constructed and the neural network agent built on top of that instead.

4 Tests

In order to evaluate our results, it was important to properly design tests and metrics. In this section, we describe how and why we designed the tests we did.

4.1 Q-learning agent

The performance of the Q-learning agent is largely affected by the implementation and learning parameters. In this part, there will be a comparison in the performance of Q-learning player in different parameters by running the game 2000 times. The agent configurations for all performance comparisons in this section are set to three random agents vs. one Q-learning agent. and other default parameters can be found in table 1

Table 1. Default run configuration parameters

Setups/Parameters	Value
Learning whole round	*No*
Turn in state representation	*No*
Negative reward reinforcement factor γ	10
Disable using ES card	*No*
Learning rate α	0.7
Discount factor λ	0.2

Learning whole round vs. learning each turn When learning the whole round, the immediate reward for first 5 turns was set to 0, and the total reward is accumulated to the end of each round as the reward for the last turn state/action. So, only long term reward will be considered by the Q-learning agent in this setup. When learning each turn, the immediate reward for each turn was decided by the moving direction and distance after resolving the game each turn. The run configuration can be found in table 1, where learning whole round is overwritten by this test.

Table 2 shows the performance comparison between learning whole round and learning each turn when calculating reward for learning to play cards. One may think that long-term turn reward gets better results compared with learning reward in each turn. But the results suggest that this is not the case. The Q-learning player who learns the whole round did not outperform the player who learned rewards each turn, it was even slightly worse.

Table 2. Learning whole round vs. learning each turn performance

	No. of Wins			
	Q-learning Player	Random P1	Random P2	Random P3
Learning Whole Round	797	399	404	401
Learning Each Turn	818	388	400	395

Turn in state representation Players may choose to use different strategy for different turns even if the state is same. So, we compared the performance between the player who includes turn into the state representation and the player who does not includes turns into the state representation. The run configuration can be found in table 1, where turn in state representation is overwritten by this test. The performance comparison result is shown in Table 3. The result shows that considering turns into the state representation does not improve performance.

Table 3. Including turn into state representation vs. not performance

	No. of Wins			
	Q-learning Player	Random P1	Random P2	Random P3
States don't include turn	818	388	400	395
States include turn	803	410	404	384

The influence of negative reward reinforcement factor Players should also avoid backward moving. So, we introduced negative reward reinforcement factor γ to train the Q-learning player. The run configuration can be found in table 1, where γ is overwritten by this test. The performance result is illustrated in Table 4. The result shows that different settings of γ does not influence performance much.

Enable ES card vs. disable ES card Properly using ES card has significant influence to the result when a human plays the game. So, we also compared performance between the Q-learning agent with ES card enabled and the player

Table 4. Performance about influence of negative reward reinforcement factor

	No. of Wins			
	Q-learning Player	Random P1	Random P2	Random P3
$\gamma = 10$	818	388	400	395
$\gamma = 5$	802	399	404	395
$\gamma = 1$	798	410	427	366

with ES card disabled. The run configuration can be found in table 1, where ES usage is overwritten by this test. The performance result is shown in Table 5. The result shows that though the performance of player with ES card disabled still outperform random player, but significantly loses out to the player with ES card enabled.

Table 5. Performance about enable ES card vs. disable ES card

	No. of Wins			
	Q-learning Player	Random P1	Random P2	Random P3
Enable ES Card	818	388	400	395
Disable ES Card	584	467	474	476

4.2 Measuring performance empirically

Because the game is quite random we wanted to make sure we could compare different game strategies in a reliable way. For neuro evolution, in which we have to compare strategies often because of selection, doing as few runs as possible is necessary. What needed a number $r \in \mathbb{N}$ of runs from which we can almost surely say that comparisons are statistically relevant. To find this number we pitted the random AI's against each other. The expectation is that, as the random AIs should play with the same strength, they will eventually converge to the same performance. For example, in case of p random AIs, each random AI will win $\frac{1}{p}$ of the time. We increase our guess for r until the number of wins has converged to $\frac{r}{p}$ for each random AI. To make sure the found number r is not a lucky result, the experiment was repeated 30 times.

We initially experimented to find r using win-count. The win-count w_i for player i where $0 \leq i < p$ is the result of the following function:

$$w_i = \sum_{x=0}^{r} \begin{cases} 1 & \text{if game } x \text{ was won by } i \\ 0 & \text{if game } x \text{ was lost by } i \end{cases}$$

The constraint we wanted to have was:

$$\forall i, w_i = \frac{r}{p}$$

however this constraint, proved to be to harsh: r quickly became bigger than we could practically do experiments on. Therefore leeway $l \in \mathbb{R}$ where $0 < l < 1$ was introduced. This number allows for some deviation between the found win-count and the desired constraint:

$$\forall i, \frac{r}{p}(1-l) < w_i < \frac{r}{p}(1+l)$$

Having a smaller l has the effect that r becomes bigger, but it also allows more precise perception of change in the AIs.

However, even with $l = 0.1$ does $r = 320$ not get through the 30 test, using the win-count as comparison and player count $p \in \mathbb{N}$. Increasing r much more would result in the run count becoming to big for doing neuro evolution. Therefore we used the position on the board at the end of the the game as a comparison method. The average end position after r games then needs to become similar in between the random AIs. This method produced an r that was stable during 30 tests for $p = 2$ players and $l = 0.05$ at $r = 60$ and for $p = 4$ and $l = 0.05$ at $r = 280$. Those r are small enough that using them to compare different neural networks is feasible. This means that now our neuro evolution implementation should grow fast enough that it actually improves itself.

Measuring performance with rigged randomness It turned out, however, that $r = 280$ still took too much time. Getting to 50 mutations would take about 12 hours, using a 4-way tournament selection between neural networks. To reduce this amount of time we rigged the randomness of the game per generation and pitted the random AIs against the neural networks. Every generation gets a seed from the main process. Because of the equal seed used and the fact that both the random AI and the neuro evolution AI draft cards randomly, the AIs would get the same hand for each turn of the equally seeded game runs 3.3. This assures that the random AIs also play the same cards against each neural network of that generation; the only thing that changes is the neural networks themselves, and thus the order they play their cards in. This way, the better strategy would always get a better score. Since the network always face the same odds, any r should now be acceptable.

At the start of the neuro evolution we set $r = 20$, then if a neural network manages to win more than 50% of the time, the r gets increased by 2. To break ties between neural networks, we used the following formula:

$$s = w + d/t$$

Where s is the score the neural network gets, w is the win-count, d is the ending position, and t is the total tile count, which is 54. Slowly increasing r allows for rapid evolution in the beginning, when the strategy is still "simple". Later in the evolution, when the strategies have become more complex, it allows for more precise measurement.

5 Results

In this section we will discuss the main experimental results we got from pitting the agents against each other. Each test is called a tournament, which consists of $n \in \mathbb{N}$ game runs. The results are measured through win-count per AI as a percentage of the total rounds.

5.1 Random tournaments

In this tournament (Figure 10) we pitted the random agents against each other for 30.000 games. This result supports the intuition that if random AI's are competing in enough games, they indeed will have the same strength. However if we pit a random AI that ignores emergency stop against the default random AI (Figure 11) we can see that ignoring the emergency stop in combination with random play improves performance ever so slightly. This supports the results of the Q-learning tests, that ES usage can have significant impact on play (see Section: 4.1)

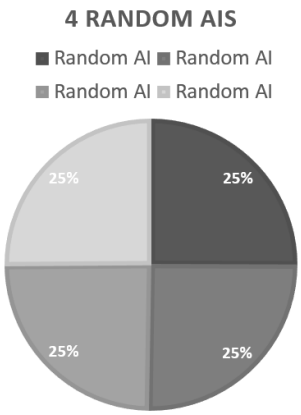

Fig. 10. Percentages of times won between 4 Random AIs after 30000 games.

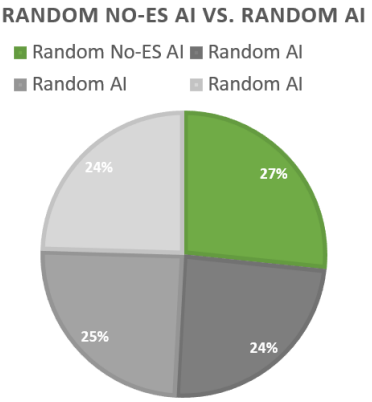

Fig. 11. Percentages of times won between a Random No-ES AI and 3 Random AIs after 10000 games.

5.2 Thinking AI tournaments

For the actual thinking AIs we created, we got more impressive results. First of all, the Q-learning agent (Figure 12) managed to play a lot better than the random baseline. We can therefore say beyond reasonable doubt that Q-learning is better at the game than random play, which means Q-learning actually learned some kind of strategy for playing this game. The decision tree did even better (Figure 13); it managed to win more than half of the games.

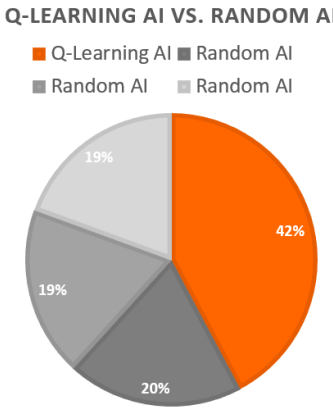

Fig. 12. Percentages of times won between a Q-learning AI and 3 Random AIs after 10000 games.

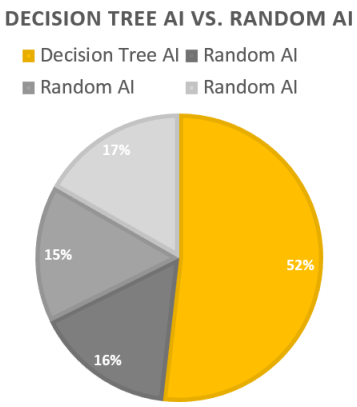

Fig. 13. Percentages of times won between a Decision tree AI and 3 Random AIs after 10000 games.

5.3 The neuro evolution tournament

The biggest disappointment was neuro evolution, as seen in Figure 14. In contrast to previously shown tournaments, this one was ran against 3 Random No-ES AIs. We did this because of our decision to also ban ES usage for the Neuro Evolution AI; the slight improvement with respect to the normal Random AI would not have come from the learned strategy but from not using the ES. In Figure 14 you can see that the Neuro Evolution AI preform a little less well than the Random No-ES AIs, (just like the Random AIs do against the Random No-ES AI in Figure 11), so one could say our Neuro Evolution AI performed similar to a Random AI. In the discussion, in Section 6.1, we explain why we think this agent had such a bad performance.

5.4 The Final Tournament

Finally, we tried pitting the different kinds of agents directly against each other to see who would win most (figure 15). This tournament basically shows what we already saw before: the decision tree wins most of the time, after comes Q-learning, limping behind that comes neuro evolution, and finally the random AI.

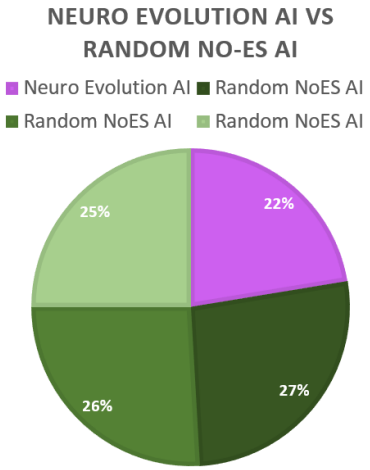

Fig. 14. Percentages of times won in a between 3 Random No-ES AIs and a Neuro evolution AI after 1000 games.

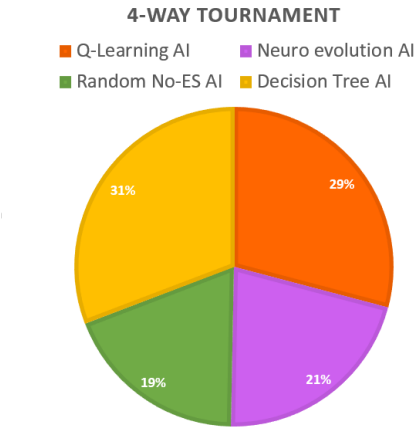

Fig. 15. Percentages of times won in a 4-way tournament with a Random No-ES AI, a Decision tree AI, a Q-learning AI and a Neuro evolution AI after 10000 games.

6 Discussion

Having now covered our implementation and the results we could get from it, it is important to consider what could have been done differently to improve these result. This section will cover ideas we have about future work .

6.1 Neuro evolution

The most disappointing result was from the neuro evolution. There are several possible explanations of why it may have failed. The first and foremost is probably because of bad selection. The mutations that were made to the network apparently had too little effect to be statistically relevant. This is either because Gravwell is so random that you need to play many more games for results to converge reliably, or because playing against random players enhances the random Gravwell seems to have. It might have been to hard for neuro evolution to figure out a strategy *against random play* in the low number of runs we used for evolution. In fact, one of the ideas we have is to let neuro evolution train against the q-learning or decision tree AIs.

This might have affected Q-learning less because its reward cycle is more fine-grained: where Q-learning can determine the reward after playing a card, the neuro evolution had to observe entire game results. So if both neuro evolution and Q-learning played 1000 games, then Q-learning would have gotten 1000*6*6 rewards and neuro evolution just 1000. An improvement of the neuro evolution could be to learn by playing only one round instead of a whole game.

Another explanation could be that the card-pick phase is just not that relevant to the gameplay. Maybe the real battles get decided during the drafting and the movement phases. We implemented the neuro-network for the card-pick phase only, since the other phases require altering the input. We could test the relevance of the choices made during the card-pick phase by extending the Decision Tree AI, overwriting its card-pick phase function to choose random, and seeing how this impacts its performance. Maybe timely playing of the emergency stop or selecting the right cards in the drafting phase could improve play by itself. A very obvious extension would be to allow a neural network to also be in control of the other phases, either as separate networks or as a singular evolved one.

Thirdly, the network did not make use of all available information. For example, during the drafting phase players can see which player picks what cards. Adding this information as input to the network could improve the play. This information only became available to us late in the development cycle, so we have not implemented it.

A final explanation for neuro evolution's disappointing results could be that not all mutations proposed by FS-NEAT were implemented. Our implementation misses mutations such as the adding and removing connections. The idea was to replace existing connections with TensorFlow constants. This way, they could be mutated as weights together with the multiplication operation. Right now every connection simply has a weight weight of 1.

6.2 Decision tree

While having the best results in comparison to our other AI implementations, the decision tree can still be improved. The current trees are rather egocentric seeing as they do not take into account what effects their choices have on the opponents. For example, moving backwards is not always a bad thing if it means the ship lands just behind an opponent; this means you both set yourself up for moving forwards and the opponent for moving backwards in the next turn. Accepting a small setback could be agreeable if it means you put your opponents at a disadvantage.

Aside from that there are still properties of the state that are not considered in the trees, like the turn number and availability of the ES. During the play phase we could decide that at a low turn number the ES should not be chosen against a card that would result in only a small decrease in position. As there are still some turns left which could result in bigger negative effects, it would be better to save the ES for later.

Also, the play phase should take into account whether or not the ES is still available. It could be considered to play cards a little more carefully when the ES has already been played. "Carefulness" could be implemented, for example, by choosing a low normal card whenever the players are all fairly close together, since close proximity would mean a high chance of a changed move direction at resolve time. The relative positions of the opponents could be calculated in the same way we simplified the state for the Q-learning: two numbers indicating the closest ship in the two directions.

6.3 Q-learning

There are a few things we could do to improve our Q-learning AI. The positions of the hulks are not taken into consideration at this moment, but as they can influence the moving direction of the player they should definitely be considered. Next to that, we could improve our methods to simplify the state for the card-pick and move phase. The current state representation methods are still too weak to fully represent the real states for player. Some neural network based state representation methods can be introduced.

An interesting possibility would be using joint-action learning, since the reward of each player is significantly effected by the actions taken by other players. The cards held in opponents' hand are not completely unknown: as by design of the drafting phase, half of all cards are public information at the start of the card-pick phase. We can further apply this information to minimize the size of the joint-action space.

7 Conclusion

The primary conclusion to be drawn is that Gravwell is too large a game to be trivially learned by learning agents. As such, any hope of finding an optimal

solution will require extensive abstraction or the use of some other method to minimize the state/action space.

Beyond that, we can conclude that it is almost always beneficial to implement a decision tree AI before starting on more complex algorithms. It proved to be significantly stronger than the learning AIs while it took a lot less effort to implement.

We can also conclude that it is useful to have a random AI, especially as a bench-marking baseline. It was also useful for figuring out a rough bound of stochasticity of the game, although this bound was not tight enough to do neuro evolution upon. This leads us to another conclusion: neuro evolution is a lot harder than it sounds.

Source code

The source code is available at `https://github.com/DrSLDR/mgmag-proj`. Note that, due to compatability issues, the neural networking agent resides on a separate branch (neural). There is presently no known way to use this agent on a non-Unix operating system.

References

1. G. Cybenko: Approximation by Superpositions of a sigmoidal Function, Mathematics of Control, Signals, and Systems, (1989), 2(4):303–314
2. Kenneth O. Stanley: Efficient Evolution of Neural Networks through Complexification, Artificial Intelligence Laboratory, The University of Texas at Austin, (2004)
3. Martin Abadi et all: Large-Scale Machine Learning on Heterogeneous Distributed Systems, Google Research, (2015)
4. Kenneth O. Stanley and Risto Miikkulainen: Evolving Neural Networks trough Augmenting Topologies, Evolutionary Computation, 10(2):99–127, (2002).
5. S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl: Automatic feature selection in neuroevolution, in Proceedings of the Genetic and Evolutionary Computation Conference, (2005).
6. Jacob Schrum and Risto Miikkulainen: Evolving Multi-modal Behavior in NPCs In IEEE Symposium on Computational Intelligence and Games (CIG 2009), 325–332, Milan, Italy, September 2009. (Best Student Paper Award).
7. Richard S. Sutton and Andrew G. Barto: Reinforcement Learning:An Introduction, The MIT press,London, England :129-149 (2012).
8. Cryptozoic Entertainment: Gravwell: Escape from the 9th Dimension, Rulebook. Available at `https://www.cryptozoic.com/sites/default/files/icme/u2793/gvw_rulebook_final.pdf`, Retrieved 2016-06-25, (2013).
9. Python Software Foundation: Python object serialization. Available at `https://docs.python.org/3/library/pickle.html`, Retrieved on 2016-06-25
10. Gareth Jones: Genetic and evolutionary algorithms. Online available at `http://www.wiley.com/legacy/wileychi/ecc/samples/sample10.pdf` Retrieved on 2016-06-26.
11. David Poole and Alan Mackworth: Artificial Intelligence: Foundations of Computational Agents, Section 7.3.1. Available at `http://artint.info/html/ArtInt_177.html`, Retrieved on 2016-06-26, (2010).