Evolving dynamic AI opponents for OpenTTD using Dynamic Scripting and Grammatical Evolution

Frank Bijlsma, 3251411

June 28, 2014

Supervisors: J. M. Broersen & J. J. C. Meyer

Dept. of Information and Computing Sciences
University Utrecht

Abstract

The goal of this research is developing a dynamic AI for OpenTTD using Dynamic Scripting and Grammatical Evolution. First Grammatical Evolution was used to develop a rulebase for the Dynamic Scripting algorithm. Rules used in OpenTTD have to be able to be used in a wide range of circumstances, very different than other games this combination of Dynamic Scripting and Grammatical Evolution has been tried on.

During this process we looked at the effects of grammar and genome structure on convergence. Restricting grammars by incorporating domain knowledge and splitting the genomes into separate parts was found to have a strong effect on convergence. Better programs generated by the Grammatical Evolution leads to a better rulebase.

Using these programs we hand build a rulebase from the generated programs and tested a Dynamic Scripting algorithm. It was found to outperform the evolved programs and beat several hand coded AI's.

Contents

1 OpenTTD **7**
- 1.1 OpenTTD . . . 7
  - 1.1.1 The Basics . . . 7
  - 1.1.2 The World . . . 8
- 1.2 Transportation . . . 9
- 1.3 Other considerations . . . 9
- 1.4 Computational Challenges . . . 10
- 1.5 Agent environment . . . 10

2 Other approaches **12**
- 2.1 PathZilla . . . 12
- 2.2 trAIns . . . 12
- 2.3 ChooChoo . . . 14
- 2.4 AdmiralAI . . . 15

3 Genetic Learning **16**
- 3.1 Grammatical Evolution . . . 18
- 3.2 Grammatical Structure . . . 18
- 3.3 Genotype . . . 18
  - 3.3.1 Backus Naur Form . . . 19
- 3.4 Operators . . . 21
  - 3.4.1 Duplicate . . . 21
  - 3.4.2 Pruning . . . 22
  - 3.4.3 Modulo rule . . . 22
  - 3.4.4 Bucket Rule . . . 23
  - 3.4.5 Positional Effects . . . 24
  - 3.4.6 Crossover . . . 24

4 Grammar and Approach **26**
- 4.1 First Non-Terminal . . . 27
- 4.2 Transportation Modules . . . 27
  - 4.2.1 Build rules . . . 28
  - 4.2.2 Maintain rules . . . 28
- 4.3 Finance . . . 29

4.4 If-statements . . . 29
4.5 Variables . . . 30
4.5.1 Route Variables . . . 30
4.5.2 Financial Variables . . . 31
4.5.3 Extending the Grammar . . . 31

5 Experimental Setup Grammatical Evolution 33
5.1 Genome makeup . . . 33
5.2 Operators . . . 33
5.2.1 Mapping rule . . . 34
5.2.2 Crossover Rule . . . 34
5.2.3 Selection Rule . . . 34
5.2.4 Mutation . . . 34
5.3 Fitness function . . . 34
5.4 Running experiments . . . 35
5.4.1 Population size . . . 35
5.4.2 Mutation probability . . . 35
5.4.3 Stopping criteria . . . 35
5.4.4 Game Variables . . . 36

6 Results Grammatical Evolution 37
6.1 Long vs. Short Grammar . . . 37
6.2 Unrestricting Variables . . . 39
6.3 Programs generated . . . 39
6.3.1 Single transportation grammars . . . 41

7 Constructing the Rulebase 43
7.1 Analyzing the rules . . . 43
7.1.1 Short Grammar . . . 44
7.1.2 Single Transport . . . 44
7.2 Gathering rules . . . 45

8 Dynamic Scripting 46
8.1 Components . . . 47
8.2 Scoring . . . 47
8.3 Rules chosen . . . 48
8.4 Experiment . . . 48
8.5 Results . . . 49
8.6 Future research . . . 50

9 Conclusion 51

A Grammars 53
A.1 Short Grammar . . . 53
A.2 Long Grammar . . . 55
A.3 Unrestricted Variables . . . 55

B Rule database **57**
B.1 Short Grammar Rules 57
B.2 Aircraft only rules 58
B.3 Train only rules 59
B.4 Truck only rules 60

One of the first challenges in game AI research was developing a program capable of beating any person in chess. (Computer) Games are very interesting from an AI perspective. Unlike situations in the real world they have clear rules and a constrained scope of the problem [25]. In a sense they can act as a microcosm of the world. Algorithms or techniques capable of solving nontrivial challenges in games can act as a stepping stone to algorithms or techniques trying to solve a problem in real life.

At the start most AI research looking at games looked at board or card games, like chess, checkers, bridge & poker [25]. However in the last couple of decades a huge industry has arisen surrounding computer games. These games offer new and interesting challenges to AI research. The board games used previously were mostly solved by looking ahead at possible moves and evaluating the resulting board positions. These approaches don't tend to work well for modern computer games however.

Modern computer games are generally evaluated on their entertainment value [29]. In games where players have opponents the AI of those opponents has a big influence on the player experience and therefore the value assigned to the game. This is especially true in games where the opponent has capabilities equal to the human player, like in RTS games. Because of the limitations of most AI's in those type of games players generally prefer human opponents [25], since AI's are incapable of being challenging to most experienced players.

While there are programs capable of beating human players at board games, players generally consider AI players in computer games to be too easy to predict and exhibiting artificial stupidity rather than artificial intelligence [25]. This offers an opportunity to AI research to develop new techniques capable of exhibiting the same complicated behavior human players have. Several different approaches have been developed to create new and better AI opponents. However game publishers are often weary of including online learning in their games. They fear that agents will learn inferior behavior [28] and make the game look bad. Dynamic Scripting is a reinforcement learning technique aimed at fast and reliable adaptation.

Since Dynamic scripting is an online learning technique, it can be used to create adaptive AI opponents. It works with a database of rules, each assigned a weight. Weights represent the probability of that rule being chosen and at the start all weights are the same. All probabilities summed together result in 1. When a set of chosen rules wins, their weight is increased, and when they lose it's decreased. The weights that aren't chosen are adjusted so the total sum of all weights stays 1.

One of the algorithm's main strengths is its capability of quickly adapting to new strategies [20]. Thereby reducing its predictability and making it capable of changing its strategy in response to a player's strategy, creating a more dynamic and interesting opponent. However, its main drawback is that it requires a handmade database of rules. One possible solution to this drawback first used by Ponsen & Spronck [14] is to use a form of Genetic Learning to generate a rule base for the dynamic scripting to use. This combination has been used to create AI's for RTS games like Wargus and World in Conflict [14, 12].

Ponsen et al. [14] used the game Wargus. An RTS game where the player has to construct buildings and raise an army. By constructing those buildings it unlocks new technologies and units to build, which in turn unlock new types of buildings. They split the game into several distinct states, each representing a possible combinations of unlocked buildings, technologies and units. They then generated rules for each of these states. The result is a type of finite state machine where the AI can move from state to state by constructing certain buildings and researching technologies. The state also determines what units it recruits and whether it attacks or defends. Since there is a set number of maps to play on, they found the best set of rules for every map. Because different factors or map sizes reward different strategies. This approach makes sense given the structure of Wargus, however it isn't feasible for all computer games.

A game with a very different structure is OpenTTD. OpenTTD is an open source remake of the game; "Transport Tycoon Deluxe" by Microprose. Players play as a transportation company and can make money by setting up routes and transporting different cargo. OpenTTD is about building and maintaining a transportation network. Compare this to Wargus where the strategy is mostly about switching states in the correct order and building an army to defeat your opponent while doing that. In OpenTTD new units are unlocked at different points in time, irrelevant of player actions. Also unlike Wargus, OpenTTD's maps are randomly generated. This makes the environment highly unpredictable. Because the environment is so unpredictable and the problem is very different from Wargus, a different type of AI needs to be developed. One that is capable of learning general rules of what constitute a good rule, rather than determining a set order in which to take steps.

Apart from the highly unpredictable, stochastic, environment OpenTTD also has a large number of different types of cargo and transportation methods. There are four main transportation types, road vehicles, trains, ships and airplanes. Each of these has dozens of different vehicles for different types of cargo and of different quality. Combined with the fact that every point of interest on the map can be connected to up a dozen other points in the area we end up with a huge search space. This search space is fully observable, since we can see everything that is happening in the map, but so large it's impossible to look at and evaluate every possible action. This situation is further complicated by the sequential nature of the environment. Building a route takes a lot of time and money, but the return on that investment can be years away. This return is also spread out over a long period, and paid out in small increments. Causing opportunity cost play a role. Since there are so many routes, there are always routes that aren't build, and building something now means we might not be able to afford another route we want to build in the future.

Because of these challenges a different approach was taken for the Genetic Learning component of the AI. Since a state based approach doesn't seem to make much sense. Instead we will use a form of genetic learning called grammatical evolution to generate rules for when to build/extend routes and how to maintain them. The approach is described in more detail in Chapter 4.

There are several high performing AI's that have been developed for OpenTTD

by the community. However all of these approaches are hand-coded and static. There hasn't been an attempt to use learning algorithms to create a more adaptive AI for OpenTTD. The use of dynamic scripting would allow for an AI that is dynamic and capable of reacting to the strategies the player uses. Increasing the challenge players face and thereby the entertainment value of the game.

One of the advantages of working with OpenTTD is that it is open source, this means that there is an available AI-API and documentation. There are also several other AI's made by other people and publicly available that can be used for training and comparison as well as several fan made AI development libraries. A modified version of the base OpenTTD game made at the TU Delft in the Parallel and Distributed Group is used. This version allows running games with only AI players without graphics, allowing for quicker testing, while automatically generating results and statistics about AI performance.

Chapter 1

OpenTTD

1.1 OpenTTD

In this chapter we will discuss the game OpenTTD, what are the goals its players try to achieve, what is possible and what are the challenges.

1.1.1 The Basics

In the game the player controls a transportation company. This company can build truck, train, air and ship routes. These routes can earn (and lose) money and the goal of the game is to become the company with the highest net worth. Since ship routes aren't used very often by players unless on very specific maps, they were left out in this project.

The game is open source, so development always continues. Over time a lot of new features have been added to OpenTTD that allow the game to be played in a wide variety of different modes and settings. The most important one for us, is the ability to write custom AI's for the game. These AI's are written in a scripting language called Squirrel. Another addition is multi-player, this means it's possible to build and test AI's to work in a multi-agent environment.

OpenTTD also allows players to completely change the available transportation options, vehicles and industries. In OpenTTD graphics files (partly) determine what vehicles are in the game. The default graphics set is called OpenGFX. It was created by the OpenTTD community to replace the old, still copyrighted, graphics of Open Transport Deluxe. To reduce complexity we will only test with the default vehicle set. However it should be noted that AI code is graphics set agnostic, meaning it would still work in every configuration.

Figure 1.1: Screenshot of OpenTTD at the start of the game, several industries and towns are visible.

1.1.2 The World

The player plays in a randomly generated world. This world is a 2D grid of tiles. For each tile several values are stored. One of these is a height value. This means that the world can have hills and valleys and tiles can be below and above water. On the map there are several industries and towns. These industries and towns have different goods they accept and different goods on offer. For instance Steel Mills offer Steel and accept Iron Ore.

The player has to transport goods (like coal, iron ore or steel, but also people or mail) between different places. For instance it's possible to connect industries to other industries, but also to connect towns to towns and towns to industries.

The player can build stations next to these towns or industries to transport goods between them. Each time one of the player's vehicles drops off some cargo, the player gets paid a certain amount. The amount of money returned is a function of the type of cargo, the distance and the time taken.

Industries keep track of two things, the amount of goods produced and the percentage produced transported. The higher the percentage of produced goods that are transported is, the faster an industry grows. So by actively utilizing an industry we help it grow and since the industries produces more, we can transport more, thereby making more money.

Over time towns slowly grow, but just like industries players can increase this growth when they are active in a town. When towns grow they will build

roads on the tiles surrounding them. A company can build these roads itself and the town will automatically use them and grow even faster.

The goal of a player is to find profitable routes, however industries and towns have a limited number of cargo they produce and accept. This means that we want to start exploiting them before the other players have a chance too. The game typically starts in 1950 and can run up to 2050. A day of in-game time takes about a second of actual time on the default playing speed. It is possible to speed this up, in this mode the game tries to process everything as fast as possible. How fast this actually is depends on the power of the computer used. For testing a modified version of OpenTTD that can be started from the command line is used that has all its graphics components disabled. This way its possible to process games with AI's faster since more computing power is available. Processing an entire 10year game takes about two minutes.

Generated worlds have a certain climate; temperate, sub-arctic, sub-tropical and toyland (fictional children's world). These climates affect exactly what vehicles, goods and industries are available in the world and it affects the way towns grow. In this research we will only be looking at the, default, temperate climate.

1.2 Transportation

The player can build routes like; roads, train tracks & canals, or use the ones already present in the map. Different types of routes have different characteristics. Ships move slowly and are expensive, but they can take a lot with them. Airplanes can make a lot of money, but they are expensive and airfields require a lot of space. Vehicles are cheap and easy, but their profit margins tend to be lower. Trains can carry a variable amount (different wagons), but they require a lot of planning to get right.

Building routes and buying vehicles costs money. The player starts with a loan and can borrow more money, but it will have to make most of the money itself. This means it's important to be profitable from the start, but preferably also offer room for growth in the future.

In addition to building roads, canals and train tracks the player has to buy and maintain vehicles to run on these routes. There exist many types of vehicles with different characteristics. So a player not only has to find good routes, it has to try to place the optimal number of vehicles on that route. To few and we might be missing out on profit. Too many and we might turn a profitable route into a net loss, since every vehicle has a maintenance cost associated with it.

1.3 Other considerations

Towns have governments. These governments have an opinion about our company, destroying trees, buildings or roads in a town lowers our rating, while planting trees, providing services (working station) and bribing officials increases

it. If a town's rating of our company drops too much we might be bared from building in or around that town.

As the in-game date increases more and better vehicles start becoming available. This means that an AI capable of playing over a longer period of time has to be able to upgrade/replace its current fleet of vehicles.

Apart from building new routes, the player has to maintain its old routes and manage its finances. A possible solution would be to split the AI up in to different parts, each responsible for managing one side of the company. This is an approach most hard-coded AI's and this research as well. It is described in more detail in Chapter 4.

1.4 Computational Challenges

One of the factors that make designing an AI capable of playing OpenTTD a challenge is the incredibly large number of possible actions to take. Every town or industry can be connected with up to several dozen other ones to create a route. Add to that the fact that good routes might connect more than 2 places together, and there is a very large number of possible routes to build.

There is also a limitation on the number of calculations an AI can make. Every in-game tick the AI is allowed to take 10000 Squirrel actions. This prohibits exploring the entire search-space and it increases the difficulty of using online search algorithms. Next is a more formal description of the environment the agent has to operate in.

1.5 Agent environment

The AI controlling the company acts as an agent in this multi-agent environment. It's formal Russel & Norvig [23] environment classification is strategic, fully observable, sequential, dynamic, discrete and competitive multi-agent[21].

While the game has some stochastic aspects, the rising and falling of industries or when vehicles break down for example. The next game state is determined by the actions the player takes and the unpredictable actions its opponents take. This makes the environment strategic.

Fully observable because the agent can access everything going on in the world at all times. It can view all routes, vehicles, industries and the performance of its competitors. It also has access to the routes the opponent is currently working on.

In OpenTTD building routes or buying vehicles affects the decisions we take in the future. It also creates new maintenance tasks in the future, this means it's a sequential environment.

Since the game map is always changing, industry and towns values change and therefore profit margins on routes, it's a dynamic environment. This is happening while the AI is busy making it's decisions, making fast decision making very important. Someone could see a player is building a route to some industry

or town and decide to quickly connect it to their network before the player has time.

The game map consist of discrete tiles in a 2D map and the time increments in discrete steps as well.

The game can have one or several players and then the goal is to have a more successful company than they do. This makes the environment a competitive multi-agent environment. The resources they fight over are industries, towns and tiles. Industries and towns because they allow us to make money and tiles because once you build something on a tile another company can not, although they can build (expensive) bridges over it. This means possibly lucrative routes for one player might be blocked by another player's route.

Chapter 2

Other approaches

There are about 20 different AI's that have been written for openTTD, these vary in quality, scope and approach. In this chapter some of the more interesting and/or ambitious ones will be discussed. Some of them focus only on a particular type of cargo, some of them on a particular transportation type. These are also the opponents for our Dynamic Scripting algorithm described in Chapter 8.

2.1 PathZilla

PathZilla is an AI that uses graph theory for network planning. First it uses a Delaunay Triangulation on the set of all towns in the map. It then uses this triangulation to find the shortest path tree for some large town in the map. The town is chosen at random from the top 10% largest in the map. Based on this it stores a graph with planned routes. In this graph the towns are nodes and the routes edges. Since it often can't build this network in one go, it will create a queue of routes to complete. The steps in this process are shown in Figure 2.1.

There are two main downsides to this approach. First, since the computation resources allocated to an AI are limited, this approach can take years of in game time to compute. During this time the AI isn't doing anything while it's competitors have often already started building routes. Secondly, because the AI can take such a long time to plan out routes, it's possible that part of the routes it imagined as part of it's network have already been taken over by other players. The AI also doesn't deal with upgrading or accessing routes it has already built.

2.2 trAIns

This AI was developed by Rios & Chaimovicz [21]. Their main goal was creating an AI that was capable of building large and complicated rail networks. They choose an AI specializing in rail because none of the AI's available at the time were good at it. Rail is considered one of the hardest transportation methods

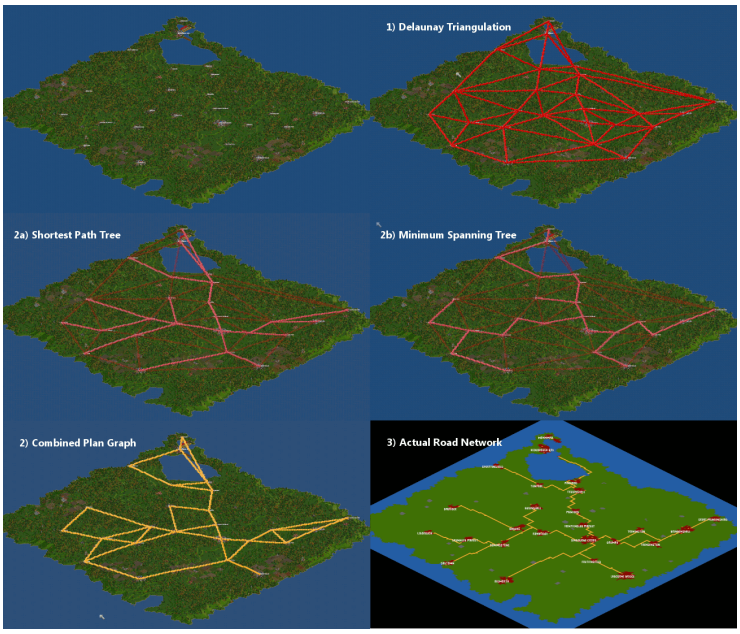

Figure 2.1: The steps taken by PathZilla for determining routes. Images were created by the author of the AI[1].

to get right, because the railway tracks require a lot more planning than for instance roads. This because tracks are one way only, whereas roads are two way streets. Rios & Chaimovicz focused on building double railways, a commonly used player tactic, which at the time no AI except for AdmiralAI[21] was capable of. Since then however several other AI's have incorporated this behavior.

The trAIns AI only builds rail networks. As soon as it receives some money it decides whether to upgrade one of it's current routes or build a new one. Upgrading routes is favored by the AI and can mean increasing/updating the vehicles or the tracks.

For the laying of railway tracks they use an A* algorithm. Rather than the normal single rail tiles they made several combinations with two rail tiles and build with those instead. These combinations allow them to more easily build two-way rail tracks by treating larger groups as single puzzle pieces. When connected a 2-way rail system is left. While the main focus was on allowing the AI to build complex two-way rail systems, it still needs to decide what industries to connect with each other.

It decides using the following steps; for each industry in the game, compute a ratio of the number of stations around the industry. The industry with the highest ratio gets chosen. This ratio is calculated by the untransported production divided by the number of stations around the industry within a certain distance and by the price it would bring for a given distance and delay.

Figure 2.2: Screenshot showing the ChooChoo four-way crossings and using buses to connect to railroads.

However since this distance and delay are always the same value and not based on what would really happen if the route were to be build, we can question whether or not this approach is effective in finding the best possible routes to build.

In their tests they find that trAIns outperforms AdmiralAI[21]. However they only play against AdmiralAI using only rail, while AdmiralAI is capable of many other different transportation options. It's possible that limiting the AI in this way hindered the performance. They did not test trAins against any other AI's either.

2.3 ChooChoo

ChooChoo is another AI focused on trains. It works by building a four-way crossing in a somewhat random location (some restrictions still apply), and extending its network to include nearby towns. Since these rail lines can only be build in a straight line, sometimes nearby town's can be reached directly. If a town is "of to the side" it will place a new crossing and connect to the town's train station and the network from there. Sometimes the station can't be build close enough to the town. In those cases it will use a small bus route to take passengers from the town to the station, Figure 2.2 shows an example of this.

This leads to a growing rail-network of connected towns. If the network can't grow anymore, a new one is started. In this approach only passengers are transported between different cities. The approach here is very much a network-based one. Later updates also added the ability to build cargo networks, although without the same network structure it uses to connect towns.

While the network approach is interesting, it seems like ChooChoo spends much time and resources developing relatively unprofitable routes. Later updates added support for more different types of routes, but their implementation is very simple. For instance cargo train routes consists of a single line of track with a train going back and forth. A route which is impossible to upgrade except for adding more carts on the back of the train.

2.4 AdmiralAI

AdmiralAI is one of the more broad AI's available for OpenTTD. It's developed by one of the main OpenTTD developers. Rios & Chaimovicz [21] used this AI to test their trAIns AI against and they found it worse than their AI. There were some problems with this approach though, namely the fact that they limited AdmiralAI to only rail. AdmiralAI is an AI that is capable of using rail, air and vehicle transports. It also connects routes of all cargo types. It is also capable of planning two-way railtracks, something many AI's struggle with.

The goal of the creator was to develop something fun to play against that used all aspects of the OpenTTD AI framework NoAI. Because the developer who made this AI also contributed to the NoAI framework, this AI contains many examples of how to use the framework.

AdmiralAI is primarily included in this list because it is one of the most complete AI's. While many other AI's focus on one particular transportation method or cargo, AdmiralAI tries to do everything (except ships).

Chapter 3

Genetic Learning

To use Dynamic Scripting, we first have to build a database filled with rules for it to chose from. In the past these rules have been hand written, however this takes a lot of time and is very dependent on the skill or domain knowledge of the rule designer. An alternative might be to use a form of Genetic Learning to create a database of proven rules. There have been several attempts at this approach to acquire domain knowledge which was then used for dynamic scripting[14, 27].

In Ponsen & Spronck[14] a Genetic Algorithm is used to gather domain knowledge about a game called Wargus. The Genetic Algorithm has a set of states. These states coincide with different states in the game. A state is determined by the buildings an AI has built, since they control what it can do. For each state the GA lists a set of moves. Completing these moves in the right order means that the AI moves to another state, with a new set of rules. A new strategy is developed for every different map in the game. So the AI in the game acts like a type of finite state machine where the AI can move from state to state by building certain buildings or developing technologies (which allow it to build new buildings). The genetic learning tries to find a good order of states to move to. There are 20 states in total and the total states and their connections are shown in Figure 3.1.

However this approach does not seem to make sense for openTTD, since in openTTD there aren't really any technology levels. As time increases other vehicles become available, but we can't control the speed at which they appear. Also, openTTD doesn't have a set of maps we can play on, every time we play the game we have a different map, nor are we limited by what we can do based on the buildings we have. Therefore a different genetic algorithm is probably more appropriate. For this case a variant of Genetic Programming, called Grammatical evolution is used.

Grammatical evolution was picked because it has been used to create AI's for games in the past [5] and it creates explicit rules of the kind that can easily be used in dynamic scripting.

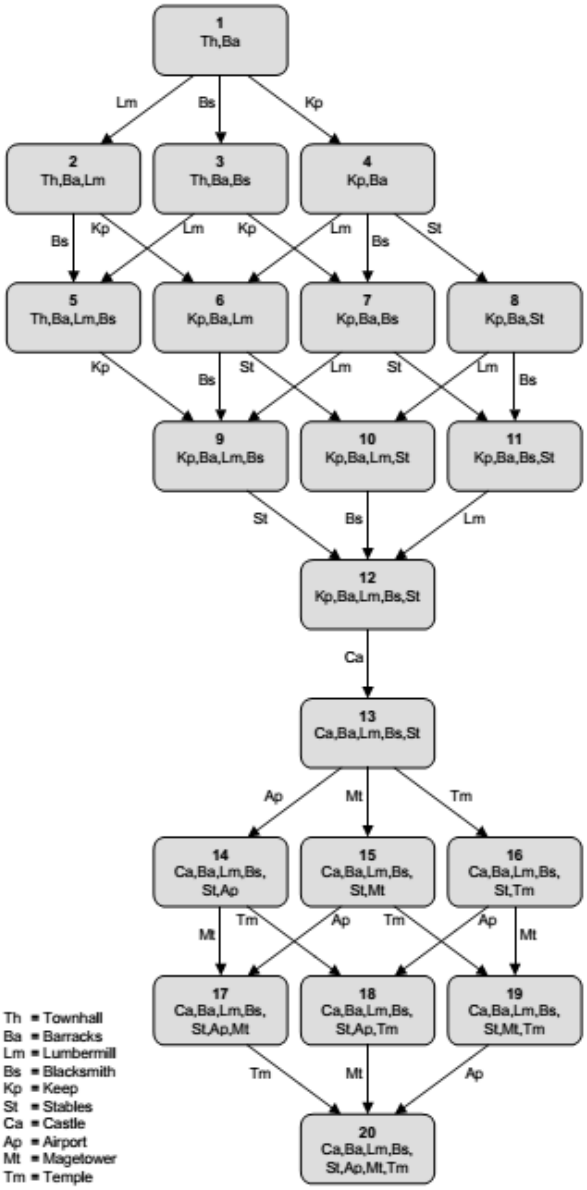

Figure 3.1: Possible states and state changes in Wargus[14]

3.1 Grammatical Evolution

Grammars are one of the core representational structures in the field of Computer Science. They can be used to structure and limit expressions in a language[22]. This allows for restricting the search space and embedding domain knowledge. This also means that constructing the right grammar is crucial to the success of the algorithm.

Genetic algorithms have been used to generate code for computer programs. This code is based on certain grammars, that determine what is and what isn't valid. The most well known approach is Koza's Genetic Programming[11]. Koza used Lisp, but others have done similar things with other languages. Some examples are Whigham's[30] where the initial population was created by a context free grammar and Wong et al.[31] who used Prolog definite clause grammars to learn first order relations. All of these approaches use trees to model code structures.

3.2 Grammatical Structure

The grammatical structures consist of a genotype and a phenotype. The genotype is the structure that we change using our mutation and crossover operators. The phenotype is the behavior exhibited when the individual is executed[22]. There are two ways in which we can structure our genotype for use in genetic algorithms. It's possible to store the tree structure directly, or alternatively we can use a linear representation and use a string representing the tree structure. A linear representation has the advantage that many different crossover and mutation operators from evolutionary strategies and genetic algorithms are available. With a linear representation we require an algorithm to map the genotype, a string, to an intermediate state before we're able to check what the phenotype looks like. This genotype-to-phenotype mapping algorithm ensures that we can transform any string of numbers to a syntactically valid phenotype (program)[17].

One of the most wildly used systems is called Grammatical Evolution (GE) developed by Ryan et al. [24]. GE has been used in many things, from finding rules for expert systems [7, 32], to evolving a strategy to play ms Pacman. In their article Lopez et al. [6] describe an arcade game from the 1980's called ms Pacman. One of the challenging aspects of this game is that there is a strong non-deterministic element to the game, just like openTTD has. In their article they describe how they evolve "if <condition> then perform <action>" rules. Generated rules of this type can easily be reused in the Dynamic Scripting rule database and our approach is modeled on theirs.

3.3 Genotype

In the original article by Ryan[24] the genotype was coded as a list codons. Each of these codons was made up of an 8-bit binary number. The length of

the genotype is variable. The grammar that is used is in the Backus Naur Form (BNF). The BNF is a notation technique to describe a grammar and it's production rules for the creation of programs.

3.3.1 Backus Naur Form

A Backus Naur Form grammar consists of terminals and non-terminals. Terminals are items that appear in the grammar, for instance operators like bigger-than or equal-to. Non-terminals are items that can be expanded into either one or multiple terminals and non-terminals.

A grammar can be represented using a tuple {*N, T, P, S*}. Here the *N* is the set of non-terminals, *T* the set of terminals, *P* the set of production rules mapping N to T and finally *S* the start symbol. BNF production rules have the following form <non-terminal> ::= <expression>. <expression> can be one or multiple different sets of terminals and non-terminals. Each set representing a possible choice. By evaluating these rules one by one we can build the program tree. In their paper Ryan et al.[24] give an example grammar. This grammar is included below. The goal of this grammar is to evolve a program capable of solving a symbolic regression problem[17] where we try to find the function that maps a set of input point to a set of output points.

N = {expr, op, preop}
T = {Sin, Cos, Tan, Log, +, -, /, *, X,()}
S = <expr>

The *N, T & S* values are defined above, that leaves *P* whose rules are used to map the genotype to the phenotype, the first of P's rules is listed here:

(1) <expr>::=<expr><op><expr>|(<expr><op><expr>)|
<pre-op>(<expr>)|<var>

Here the non-terminal <expr> can produce four different results:

<expr><op><expr> (0)

<expr><op><expr> (1)

<pre-op>(<expr>) (2)

<var> (3)

This is the rest of *P*:

(2)<op>::= +(0) | -(1) | /(2) | * (3)

(3)<pre-op>::= Sin(1) | Cos(2) | Tan(3) | Log(4)

(4)<var>::= X

The genome consists of codons and each codon is a number. Every step the algorithm takes the next available number and uses it to determine the next production step. For this we use a mapping rule. The mapping rule used here takes the modulo of the number on the genotype and the number of different possibilities for this rule. For example if we take the rule (1), given earlier, to be evaluated and the number on the genotype was 18, the algorithm would use **<pre-op>(<expr>)** since 19 modulo 4 is 2.

It should be noted that while the modulo mapping rule was the first one to be implemented for Grammatical Evolution there are some issues with it. The final mapping rule used in this research is a different one, the bucket rule. This rule and the issues with the modulo rule will be described in more detail in Section 3.4.4

The rest of the grammar is listed below.

<func>::= <header>;

<header>::= float symb(float X) <body> ;

<body>::= <declarations><code><return>;

<declarations>::= float a;

<code>::= a = <expr>;

<return>::= return(a);

What would a program generated from this grammar look like? Let's consider the following individual:

220 203 17 3 109 215 104

The 8-bit codon values have been converted to integers for clarity. The first steps of our grammar require no input. They always result in the following form.

```
float symb( float x ){
    a = <expr>;
    return(a);
}
```

The only thing left to expand is the **<expr>** expression, for this the first rule given is used. There are 4 different possibilities. The first value on the genome is 220 and 220 modulo 4 = 0. This means we will expand **<expr>** to

<expr> <op> <expr>

Next we have to expand an **<expr>** expression again. 203 modulo 4 = 3 so we get **<var>** turning our expression into

<var> <op> <expr>

<var> can only be turned into X so that is the next step. Note that we don't have to look at our genome for this, since X is the only possibility. Below are the next steps the algorithm will take. If the algorithm were to run out of codons while generating the phenotype, it simply wraps around.

Expression	Rule	Gene
X <op> <expr>	na	na
X + <expr>	2.1	17
X + pre_op(<expr>)	2.3	19
X + Sin(<expr>)	3.1	109
X + Sin(<var>)	1.3	215
X + Sin(X)	4	na

The end result of the genotype to phenotype algorithm is the following

```
float symb( float x ){
    a= X + Sin(X);
    return(a);
}
```

While the system given here is only capable of generating a single line of code. It can easily be modified to allow it to build multi-line programs. For instance by adding a line like **<code>::= a = <expr> | a = <expr> <code>**

3.4 Operators

All the standard operators, mutation and crossover, are available with genetic evolution. However there are two other additional operators, duplicating and pruning. In this section we will discuss these operators and their effects on evolution.

3.4.1 Duplicate

Duplicating means making a copy of a gene or genes and placing it somewhere else on the genome. This is a phenomenon that is observed in nature and can be beneficial.[24] In Grammatical Evolution the duplication operator works by taking a random number of genes and copying them. These new copied genes get placed on the last position of the genome. It can be beneficial because either

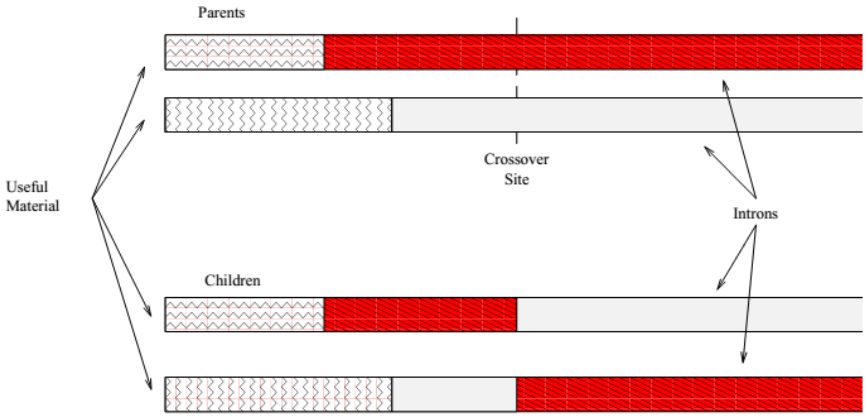

Figure 3.2: Showing the reduced effect of crossover when a genome has too many introns [24].

it provides copies of a beneficial gene or because it allows new functionality to be obtained[18]. New functionality can be obtained because the same set of genes can lead to different impressions based on the rule the first element is interpreted with.

3.4.2 Pruning

It's possible that a genome has more genes than it uses for the creation of the program associated with the genotype. These inactive genes are called introns and they serve to protect the effective genes from the effects of crossover [24]. However having too many introns could hamper our evolution, because individual no longer change. Figure 3.2 shows the reduced efficacy of a crossover when a genome has a lot of introns.

A solution to this problem is the pruning operator. Every individual that does not use all of its genes has a probability of having the prune operator applied to it. This prune operator removes all unused genes. Initially Ryan et al. claim that the effect of pruning is a faster, better convergence[24]. However in a later paper they cite research indicating that introns have been shown to beneficial in other genetic algorithms[18]. They still use the operator in later research[18], but with a much lower probability of 0.01. So the positive effects of this operator are debated.

3.4.3 Modulo rule

The modulo rule is the first mapping rule developed for Grammatical Evolution. When the modulo rule is used the codon values are drawn from [0,255]. This leads to many codon values always mapping to the same rules. In the case of

the grammar listed below there are only 2 possible values for each rule. This means the result is determined by the least important bit of the codon[9], since all even codons become 0 and all uneven 1.

<bitstring> ::= <bit> | <bit> <bitstring>

<bit> ::=0 | 1

Codon Value	<bit>	<bitstring>
0	0	<bit>
1	1	<bit> <bitstring>

Because the rule chosen only depends on the least significant bit, a linkage between different production rules is introduced. Since even codons will always pick <bit> or 0 and uneven codons 1 or <bit> <bitstring>. This linkage is undesirable since it inhibits the GE's intrinsic polymorphism[9]. Since this bias is dependent on the layout of the rules, the way rules are laid out might have an effect an the search efficiency, however this is effect can't be predicted. This was shown by Keijzer et al. [9] who showed that two similar grammars with different orderings converged to different mean fitness values. The solution offered by Keijzer et al. is to change the rule we use for picking production rules, this new rule is called the bucket rule.

3.4.4 Bucket Rule

The bucket rule is an alternative mapping rule instead of the modulo rule developed by Keijzer et al.[9]. When given a set of n non-terminals with production rules [c*1*,...,cn] and given the current symbol r. Condon values are taken from the interval $[0, \prod_{i=1}^{n} c_i]$ instead of [0,255]. The mapping rule used is[9]:

$$\textbf{choice(r)} = \frac{codon}{\prod_{i=1}^{r-1} c_i} \textbf{ mod } cr$$

Keijzer et al. showed the effectiveness of this change in their experiments. In the example from Section 3.4.3 that would mean codon values are chosen from the interval [0,3]. Below is a table showing their new encoding values. Now production rule results are not automatically linked, but the relative proportions of the values chosen are still intact.

Codon Value	<bit>	<bitstring>
0	0	<bit>
1	1	<bit> <bitstring>
2	0	<bit> <bitstring>
3	1	<bit>

3.4.5 Positional Effects

An often-mentioned issue with Grammatical Evolution is that a small change in the genotype might completely change the interpretation of all the following codons [4]. This would mean that mutation or crossover events that happens near the front of the genotype would be far more destructive than those occurring near the back. To test if this is true Castle et al.[4] ran a GE for different problems; a Santa Fe trail, 6-bit multiplexer and a symbolic regression problem. For each (one point) crossover and mutation they compared the parent(s) and the offspring to see if they improved or not. They found that mutation operators have a bigger effect, both positive and negative, when they change codons at the start of the genotype. The effect of the crossover was found to be more likely to be positive and less likely to be negative the further back it occurred.

3.4.6 Crossover

Standard crossover in Grammatical Evolution consists of a 1-point crossover. First two individuals are selected, then a point is chosen randomly for each individual. Next we swap everything to the right of the chosen point between the two individuals. Despite the fact that this is a common approach in string based genetic algorithms. Some research warns against using this crossover in GE algorithms due to it's destructive nature [19] or because they claim it isn't an improvement on randomly generating new subtrees and using those instead [3]. This last claim seems unlikely however and in research by ONeill [19] the standard 1-point crossover clearly outperforms both "headless chicken" crossover, which replaces random codons with a new random value, and using no crossover operator.

An often noticed effect in genetic programming is the rapid increase of size in a population. It has been theorized that is this arises to combat the harmful effects of crossover, by creating a buffer so the crossover has no effect on the phenotype[16]. This effect is called bloat.

Homologous Crossover

An alternative crossover method that was developed is called Homologuos Crossover[19]. This crossover method was developed to try to keep the context of the mapping process in mind while recombinating. The crossover works as follows: while the algorithm is running a history of rules chosen is stored for each individual. When two individuals are selected for crossover this history is compared and the *region of similarity* is determined. This region is the overlap in the rules chosen at the start of both individuals. The first crossover point is chosen to be at the end of the region of similarity. For both individuals the second crossover point will be chosen somewhere at random in the *region of dissimilarity*. Note that the selections swapped between individuals don't have to be of equal size.

The homologous crossover operator was developed by O'Neill et al. [19] and they tested it performance compared to the typical 1 and 2 point crossover

operators. Both a homologous operator that allowed for differing sizes and one with constant sizes were compared. While the homologous crossovers sometimes found very large improvements, the 1-point crossover still outperformed it in the long run. A possible explanation for this is that the homologous crossover starts acting more like a local search the further it evolves. This in contrast with the 1-point operator which keeps searching on a global scale[19].

Ripples

Clearly the 1-point crossover has some advantages. It outperforms using no crossover, replacing random codons and the homologous crossover. The question however is why. When the genome is processed to create the final program it's possible that not all codons are used due to the bloat effect. This last group of (unused) codons on the right of the genome is called the tail. The string of codons in the genome is used to build a program tree. When a crossover event occurs it's as if we remove one of the subtrees of our tree and replace it with the subtree from another genome[10]. These subtrees are also known as *ripple trees*.

Due to the intrinsic polymorphism of the subtrees[19], every codon can take every value, changes in one codon ripple out and change the interpretation of the following codons. These ripples can even occur between subtrees. Since a change in one tree could mean it now has codons left over or it's lacking them, the following trees would then respectively gain or lose codons. So despite 1-point crossover not taking the tree structure into account when performing crossover, due to the intrinsic polymorphism tree structures stay intact.

Chapter 4

Grammar and Approach

In OpenTTD the player has to perform several different tasks. They are the following:

- Upgrade and repair vehicles
- Find new profitable routes
- Scrap unprofitable routes
- Manage money (how much to borrow, invest, etc.)

The approach used by all AI's discussed in Chapter 2 and by Rios [21] is to split the AI into different submodules, each responsible for a different task. Each transportation method used get its own module and there is a separate finance module in charge of managing the AI's finances. These tasks can be subdivided into several different sub-grammars, each capable of generating rules relevant to the task. Figure 4.1 shows an overview.

First the first non-terminal, the start of the program, will be discussed. From this non-terminal its determined what sub-grammar will be used. Since the rules for the different transportation modes have very similar structures they will be discussed next. After that the finance rules will be discussed. Parts of the grammar will be replicated here, for the full grammar please see Appendix A.1. The grammar presented here is cleaned up for readability, the changes made are described in Appendix A.

In Chapter 3 grammatical evolution and the effect of grammar and production rules on the efficacy of the algorithm were discussed. Changes in one codon affect how the following codons are interpreted. O'Neill et al [19] refer to this as the *ripple effect*. In order to minimize the destructive effect these ripples the grammars where designed with the goal of having similar structures. This means that the changes in one codon will still cause following codons to be interpreted in a similar manner as they were before. This will hopefully preserve the structure of the genotypes when performing a crossover or mutation.

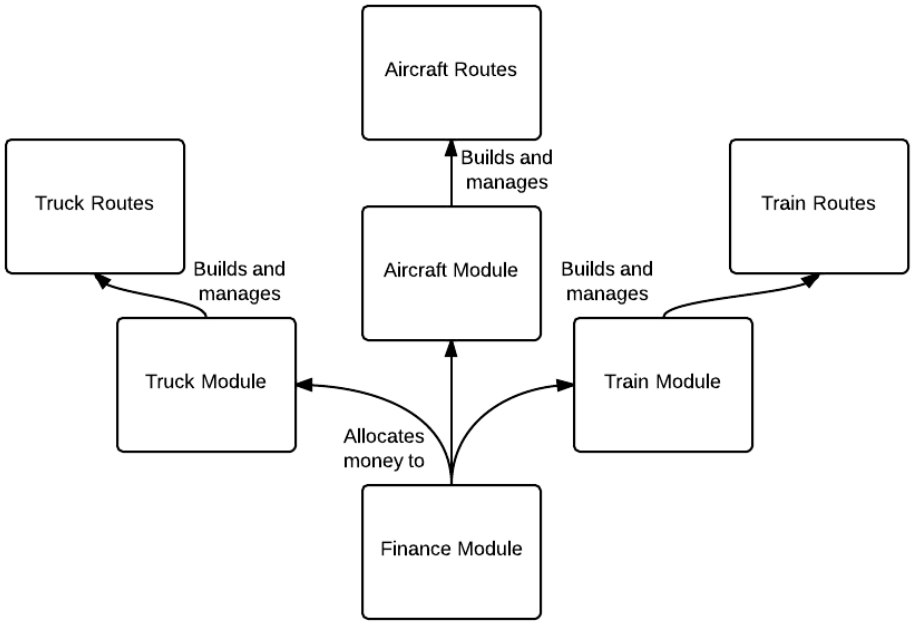

Figure 4.1: Overview of the different modules.

4.1 First Non-Terminal

The first non-terminal we use is **<rule>**, its grammar is listed below. Since this is the first non-terminal in our grammar, it's the first to be evaluated. What this non-terminal does is select for what module a rule will be generated for. There is some similarity between this approach and the gene-encoding used by Ponsen et al.[14]. They subdivide the genome into several different states and have the first element, a letter while the rest are numbers, of the sub-state determine what type of rule is being encoded.

<rule> ::= <money> | <truck> | <rail> | <aircraft>

4.2 Transportation Modules

In this section the sub-grammar for the truck module will be discussed. It should be noted however that the grammars for the other transportation mechanisms have the same structure, only the **truck** is replaced. Again, the full grammar can be found in Appendix A.1. After determining that the type of rule we are using is a road transportation rule, the next step is determining what type of rule we will use. Build rules are for adding new routes and maintain rules for maintaining the ones we have. The <build> and <maintain> non-terminals

have the prefix **truck.** this to distinguish the different transportation modules. The <train> and <aircraft> have the prefixes **train.** and **aircraft.**.

<truck> ::= if (truck<route_var> <equals> <num>) truck.<build> |

if (truck<route_var> <equals> <num>) truck.<maintain>

One important difference between the aircraft module and the other modules is that the aircraft module ignores cargo designations. This because all airplanes can only carry passengers and mail, nothing else. The distance variables work as regular. The aircraft module still has a cargo designator so that when a rule switches from aircraft to some other cargo type, all other variables are still interpreted in the same way.

4.2.1 Build rules

<build> rules allow for two possibilities, find a new set of two industries to connect, or extend an already existing network by adding a location to it. The finding and extending nonterminals refer to functions the AI has access to at runtime. These functions both require the same three variables. The first, **<cargo>**, is the cargo type. The second two **<distance>** variables refer to one of several possible distance values. The lowest of these two values is used as a lower bound and the highest as an upper bound on the distance the new route can be. These rules have an if statement at the start, determining when they are used. If statements are expanded upon in Section 4.4. In addition to an if statement, build rules also have a hard rule build in for when a route can be build. The corresponding transportation module has to have enough money available in it's budget to build the route.

<build> ::= <find> | <extend>

<find> ::= find (<cargo> <distance> <distance>)

<extend> ::= extend (<cargo> <distance> <distance>)

4.2.2 Maintain rules

In addition to building new routes the AI also has to maintain old ones. There are two options for maintaining old routes. These are adding new vehicles/upgrading old ones on existing routes and removing unprofitable ones. Both the upgrade and the remove non-terminals refer to functions the AI has access to. The remove function tries to find the worst performing route and removes it. UpgradeVehicles tries to upgrade vehicles on all the routes, if this isn't possible, it will try to add vehicles instead, but only when routes have cargo waiting.

<maintain> ::= <upgrade_vehicle> | <remove_unprofitable>

<upgrade_vehicle> ::= UpgradeVehicles()

<remove_unprofitable> ::= RemoveUnprofit( )

4.3 Finance

The finance module acts as the accountant of the company. Its role is to assign money to the budget of the various transportation modules and to take on or pay off debt. For this it has two rules a **<debt>** one and a **<increase_budget>** one. The debt rule increases or decreases debt by a certain amount based on the variables given. The boolean determines whether it is a return payment (0) or an additional loan (1). The num non-terminal is a normalized value between 0.1 and 1. The actual value given is a percentage of the budget available, where 1 is 25% of all available money. Rules can maximally spend 25% of all available money to prevent one rule from dominating all other money rules by the virtue that it's evaluated first.

The increase budget rule requires the type of transport to finance and again a <num> variable. The num variable is handled the same as with debt and can maximally spend 25% of the budget at once. The money is added to the chosen transportation module's budget.

<money> ::= if (<fin_var> <equals> <num>) game.<debt> |

if (<fin_var> <equals> <num>) game.<increase_budget>

<debt> ::= debt (<boolean>, <num>)

<increase_budget> ::= add_budget(<num>, <trans_budget>)

4.4 If-statements

The rules used by the different modules are preceded by if-statements that then determine when the rules should be applied. In this section we will look into those if statement in more detail. The if-statement takes three arguments. A variable, an equality variable ($<$, $>$, $\leq$ or $\geq$), and a number variable. The num non-terminal is still a value from 0.1 to 1, and the variable values are all normalized to be within the interval [0,1].

Variables are normalized to keep rules relevant in different stages of the game. Since the amounts of money and vehicles in the later years of the game can be a 100 times bigger than at the start. This makes it very difficult to have a rule that can be relevant in all years of the game if the numbers are set. How the variables are normalized will be discussed in section 4.5. There are two types of if statements in the grammar, those looking at route variables <route_var>

about the performance of different transportation companies. And there are financial variables <fin_var>, looking at things like profitability and size of budgets or loans. Two examples from the grammar are given below.

if (truck<route_var> <equals> <num>)

if (<fin_var> <equals> <num>)

4.5 Variables

As mentioned in Section 4.4 there are two types of variables, route variables and financial variables. They are each used in the if statements of various rules. Here in this section we will go into more detail about them. First we will look at the route variables.

4.5.1 Route Variables

The route variables are selected for by the non-terminal <route_var>. Looking at Section 4.2 we can see that the instances of <route_var> are preceded by truck. This is to make sure we look at the corresponding variable for all truck routes. For train or aircraft rules it's replaced with train or aircraft respectively. So while the <route_var> non-terminal ends has five possible choices, in the end there are fifteen different variables, because we monitor five variables for each of the three different transportation types.

<route_var> ::= _waiting | _pr_prof | _rt_build | _profit | _budget

For each transportation type we record five different variables. Below each variable is explained in more detail.

Variable	Complete name	Explanation
_waiting	Waiting	Total amount of cargo waiting to be transported relative to a set value per route.
_pr_prof	Percetage profitable	Percentage of all routes that is profitable
_rt_build	Route build	Time since the last route was build, where 0 is 0 days ago and 1 is 2 years ago.
_profit	Profit	Total profit routemanager relative to maximum allowed loan
_budget	Budget	Total budget relative to maximum allowed loan

Some of these variables are set relative to maximum allowed loan. This is because the maximum allowed loan scales in game relative to the company

value, and will always be higher than the value given, while not being so much higher that the value stays very low. This way the requirements will scale up as the company grows. The _waiting variable looks at the number of routes and the total amount of cargo waiting. Depending on the type of transportation company used we take a set amount of maximum cargo waiting per route. The bigger the amount of cargo vehicles of a certain type can take with them, the bigger the value is. For trucks it's set at 200, for trains at 300 and for aircraft at 400. So if we have five truck routes, and a total of 300 cargo waiting, our truck_waiting variable would be 0.3, since 300 / (200 * 500) = 0.3.

4.5.2 Financial Variables

Rules for the financial manager use financial variables in their if statements. They are selected by the non-terminal <fin_var>.

> <fin_var> ::= truck_profit | aircraft_profit | train_profit | game.balance_left | game.debt_taken | truck_budget | train_budget | aircraft_budget

These variables can be split into three different groups. First the _profit variables. These values are the same as those discussed in Section 4.5.1. The second group are the _budget variables, they too are the same as discussed in Section 4.5.1. The third and final group are the game.balance_left and game.debt_taken. They are the balance available, i.e. money not allocated to the budget of a transportation module and the debt taken, respectively. Each of these variables are normalized by making them relative to the maximum amount we can loan.

4.5.3 Extending the Grammar

Now all the sub-grammars used have been described. However our current grammar only allows for the creation of a single rule. One possible solution would be to extend the grammar to allow multiple rules to be captured by it. For instance like this:

> <rule> ::= <money> <rule> | <truck> <rule> | <rail> <rule> | <aircraft> <rule>

In GE this is the general approach. A <rule> is generated using a particular sub-grammar and then the algorithm continues making rules using the rest of the genome. However due to the earlier described ripple effect we theorize that this would result in very unstable genomes. One change has the possibility of changing all the rules present in a genome. This is undesirable of course. It's manageable with small grammars, where the final program might be 5 or 6 lines of pseudo code, for instance when generating an AI for ms Pacman[5]. In this case however we decided to allow each AI to use at least 20 different rules, making the original approach seem less feasible.

Instead the genome has been split into 20 parts. Each part has 11 codons, enough to generate any rule in the grammar. These parts are independent, so changes in one part will only affect a single rule. Also rather than the whole genome having a tail containing possible subtrees, each part has its own tail. This is discussed in more detail in Section 5.1. In Chapter 5 we will compare these two approaches.

Our hope is that this increases the locality of the crossover operator. A high locality means that solutions that have a similar genotype will have a similar phenotype and score. It's been shown to be an important factor in convergence towards a good solution [8]. Because it allows offspring to keep the qualities that made their parents perform well.

Chapter 5

Experimental Setup Grammatical Evolution

In Chapter 4 we discussed the grammar we will use for our genetic learning. In this Chapter the setup of the experiment will be discussed as well as the operators used by the GE algorithm.

5.1 Genome makeup

The genome we use is cut into different parts. The size of these parts is predetermined and is based on the minimum size required to always form a complete rule in the grammar. For the standard grammar discussed in Chapter 4 this value is 11. At the start of the algorithm a variable is passed along declaring how many rules each genome should contain. The genome length L is then determined as follows:

$$L = n * s$$

Where n is the number of rules and s is the size per rule. For instance if rule length is 11 and 10 rules are required genome length is set to 110. Rather than one long tail for the whole genome, with this approach each rule has it's own small tail. The generated AI's each have 20 rules, with the length of each segment of the genome being 11. This means that the genomes have a length of 220 codons.

5.2 Operators

The Genetic Learning algorithm requires the use of several operators to change the population and increase fitness. The operators needed are a mapping rule, a crossover rule, a selection rule and a mutation rule.

5.2.1 Mapping rule

The mapping rule is the rule used to map codon values to programs. Based on the grammar the mapping rule determines what choice we make when unpacking a non-terminal and it has more than one possibility. The bucket rule, discussed in Section 3.4.4 will be used as the mapping rule. It was picked because it disables the linkage between different production rules present in the modulo mapping rule, these linkages have been shown to have unwanted effects on convergence [9].

5.2.2 Crossover Rule

The standard 1-point crossover will be used, since it has proven more effective than other, more complicated rules, have [19, 10]. A random point is chosen on the genomes from the parents and everything to the right of that is swapped. Since our genomes consist of a set of rules, encoded one after the other, most rules survive the crossover intact. This will hopefully keep children somewhat similar to their parents, helping with convergence. To preserve rules both parents have their genome cut in the same place.

The algorithm is elitist, this means that every generation we perform the crossover 1 time, and then replace the lowest scoring genomes in the population. The genome is only replaced if it's score is worse than the score of the new genome.

5.2.3 Selection Rule

The parents are picked using tournament selection. A tournament selection of n means that we pick n random genomes population and pick the m with the highest score to be one of the parents. The higher we set our n value, the higher the selection pressure. Causing the population to converge faster. However we don't want to to converge to soon, since it could end up in a local optimum. A n value of 3 and an m value of 1 were used. These values are the same as those used in the research on Wargus[14].

5.2.4 Mutation

The standard 1 point mutation is used. Each genome in the population has a chance p to be mutated. A mutation means that a single codon's value is replaced with another value. Since we split the genome into different parts for different rules, only a single rule is affected by the mutation. This means the effect of a single mutation might be rather limited.

5.3 Fitness function

Individuals in the population need to be scored. For this the following function F is used:

$$F = m + v - l$$

Here m is the total amount of cash the AI has in it's coffers, v is the company value, the total value of all the vehicles and stations it possesses. Finally l is the amount of money the company has loaned. By adding the money and value variables we determine how successful the company was, since it represents how much it was able to earn and invest. By subtracting the loan value we punish AI's that only have a high company valuation by borrowing lots of money without doing anything with it. The fitness function allows both positive and negative values to be returned.

5.4 Running experiments

The goal of these experiments is two-fold. First look at the normal grammar defined in Chapter 4 and compare it to the extended version of the grammar defined in Section 4.5.3. However before we can start comparing these two grammars some other values have to be established. The second goal is to generate a rule database for use in our dynamic scripting algorithm. Each test consists of 50 complete runs of the algorithm, results about the population are stored every 25 generations.

5.4.1 Population size

The size of the population has a big effect on the efficacy of the genetic learning algorithm, and finding the optimal size is the subject of debate [2]. Larger populations tend to give better results, but they also carry a larger computational cost with them. For this experiment we chose a population size of 50, since early tests indicated it offered a balance between computational resources and results. This the same value as used by Ponsen et al.[14] for their experiments with Wargus. A major practical limit are computational resources, each genome has to be scored. This scoring takes a couple of minutes per genome so huge populations of a 1000 like Keijzer et. al. [9] use are infeasible.

5.4.2 Mutation probability

In Section 5.2 we discussed the mutation operator. O'Neill et al. [19] showed that for different problems, when a 1-point crossover is used mutation has a beneficial effect. They use a mutation rate of 1% in their research. However they also have much larger populations, in the 100's, therefore for this research we used a higher mutation rate, to help the smaller population escape local optima. Each genome has a 3% chance of mutating.

5.4.3 Stopping criteria

The algorithm has to stop learning at some point. There are two different criteria that determine when it stops. The maximum allowed generations and

the maximum allowed generations without change. The first value, maximum allowed generations, is set at 400. This means that the algorithm wil stop after 400 generations.

The second value is maximum allowed generations without change, this one is set at 150. Every generation we check if we found a new best solution, if this is the case we store this. At the end of every generation the algorithm checks how many generations ago we last found a new optimum, if this is more than 150 generations ago the algorithm stops running. Here we assume that 150 generations without a change means that the algorithm has stopped evolving and found an optimum.

5.4.4 Game Variables

The genomes were tested on a standard temperate climate map. The map was 512x512 tiles big. During the evolution the generated AI was the only AI in the game. The best performing AI's will be tested against other AI's later. Games were run for 20 years, after which the results were analyzed logged.

Chapter 6

Results Grammatical Evolution

In this Chapter we will discuss the results of the experiment discussed in Chapter 5 comparing the long and short form of the grammar. We will also analyze the programs generated from the different grammars and describe which rules we chose for our Dynamic Scripting experiment. As well as describe some auxiliary experiments run and their results.

6.1 Long vs. Short Grammar

In Chapter 4 our grammar was described, with a possible modification described in Section 4.5.3. From now on the original grammar will be called the short grammar and the modification the long grammar.

When working with the short grammar the program will split the genome into different parts, each part corresponding with a single rule. The final program is then created by placing the different rules one after the other. With the long grammar the genome isn't split into rules, rather the first non-terminal <rule> can result in several different choices, but all of them contain a new <rule> statement. Because of this the program continues adding rules until it runs out of codons, any incomplete rules are ignored. The full grammars can be viewed in Appendix A.1.

In Chapter 4 we theorized that the short form grammar would perform better, since it would be more stable. Because the genome is split into independent parts changes like a mutation or a crossover can no longer cause the whole phenotype to change, disrupting the locality of the offspring. Rather smaller changes are made to a part of the phenotype.

Figure 6.1 shows the highest score in the population as the number of generations increases. These results were obtained by running each the algorithm 50 times for each grammar with the experimental setup described in Chapter 5, the results were then averaged.

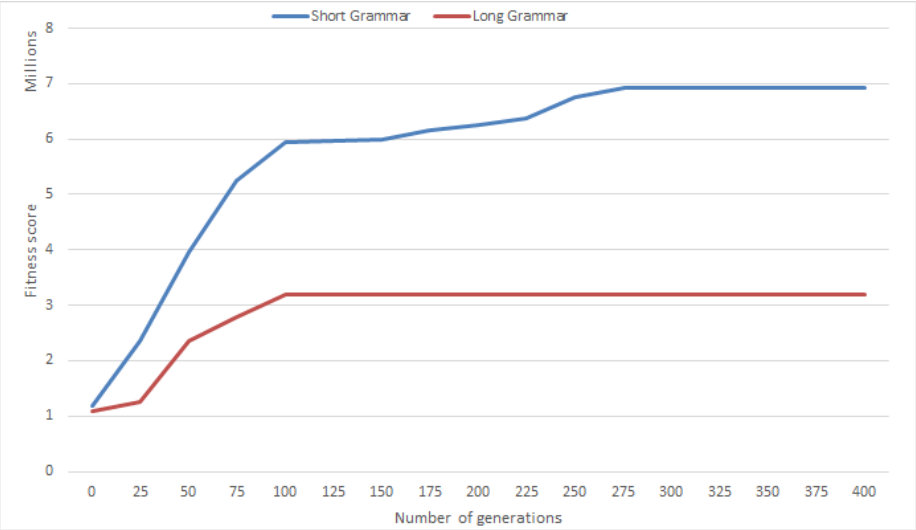

Figure 6.1: Figure showing the best score in the population over time, the x-axis has the number of generations and the y-axis the fitness score in millions

The two grammars don't differ significantly at the start. However it's clear that populations using the short grammar increase their performance at a much higher rate than those using the long grammar. After 100 generations the short grammar performs more than twice as good on average. The difference between these two approaches is significant($p < 0.001$) after 50 generations. Since every phenotype the short-grammar can encode for can also be encoded by the long grammar, and vice versa, the difference in performance isn't caused because one solution is possible in the short grammar, but not in the long. Rather it's because of the difference in structure in the genotype and phenotype used. Since the genotype is split in parts, each encoding for a part of the phenotype, the disruptive effect crossover and mutation can have is lessened.

The maximum allowed number of generations is 400, however both the long and the short grammar results have stopped growing before then. This is an indication that the populations already reached a local optimum before reaching 400 generations and that our stopping criteria, described in Section 5.4.3, were liberal enough not to get in the way of better results.

Both grammars are shown to increase their fitness score up to around 100 generations, after that however the long grammar seems to get stuck in a local optimum. While the short grammar keeps increasing for a while until around 275 generations.

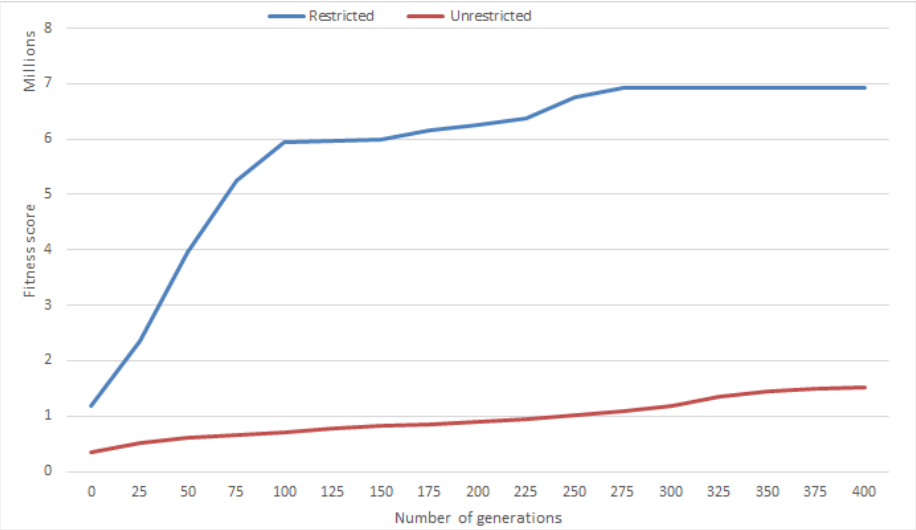

Figure 6.2: This figure shows the best score in the population, averaged over 50 runs, comparing the restricted and unrestricted versions of the grammar.

6.2 Unrestricting Variables

Assumptions were made while designing the original grammar. One of those was that the final programs would be of a higher quality if variables were automatically linked with regard to transportation module. So if the final action has to do with the truck module, the variable in the if statement also says something about truck module. Similarly we have different non-terminals for financial and transportation modules and the rules generated for them, <fin_var> and <route_var> respectively, this difference was kept intact. Because the variables are unrestricted the grammar as a whole can be simplified. The grammar can be found in Appendix A.3.

To test if this assumption was correct we compared the short grammar with the new grammar, where there is no structure imposed on the variables. The results are in Figure 6.2. It is clear from these results that restricting the variables is necessary to encourage convergence. While the genomes with unrestricted variables show some evolution, it is very very slow. In fact most populations were still evolving when hitting the 400 generations mark, indicating that the grammar might be able to achieve somewhat comparable solutions if given more time.

6.3 Programs generated

The goal of the genetic learning algorithm is to develop a database of rules for use in the dynamic scripting. Since the rules evolved by the short grammar with

restricted variables perform the best, they will act as the main source. However we won't just look at the best one, but also other well performing individuals, since they might also give us insight into what rules perform well.

1. if (game.truck_waiting < 0.6) truck.UpgradeVehicles()
2. if (game.aircraft_rt_build <= 0.1) aircraft.RemoveUnprofit( )
3. if (game.train_budget >= 0.1) game.debt(0.3, 1)
4. if (game.balance_left <= 0.9) game.add_budget(1.0, "aircraft")
5. if (game.truck_pr_prof <= 0.7) truck.find("GRAI", 500, 500)
6. if (game.aircraft_profit <= 0.6) game.debt(0.2, 1)
7. if (game.train_rt_build < 0.2) train.RemoveUnprofit( )
8. if (game.truck_pr_prof >= 0.9) truck.UpgradeVehicles()
9. if (game.aircraft_budget > 0.3) aircraft.find("IRON", 400, 100)
10. if (game.truck_profit <= 0.7) truck.extend("COAL", 500, 500)
11. if (game.train_profit <= 0.5) game.debt(0.3, 1)
12. if (game.aircraft_waiting < 0.6) aircraft.RemoveUnprofit( )
13. if (game.aircraft_profit < 0.6) game.add_budget(0.2, "aircraft")
14. if (game.aircraft_profit >= 0.3) game.add_budget(0.4, "aircraft")
15. if (game.aircraft_profit >= 0.1) game.debt(0.7, 1)
16. if (game.train_rt_build >= 0.6) train.extend("OIL_", 400, 75)
17. if (game.train_budget < 0.3) train.find("WOOD", 75, 100)
18. if (game.truck_budget > 1.0) truck.RemoveUnprofit( )
19. if (game.train_profit < 0.9) train.UpgradeVehicles()
20. if (game.truck_rt_build >= 0.9) truck.RemoveUnprofit( )

Included above are the 20 generated rules for of the best performing genomes. First lets look at the spread of the rules across the modules. Table 6.1, shows how many rules are assigned to each module. What we see is that the aircraft module has only three rules. One of them building new routes between 100 and 400 long and the other two remove unprofitable ones. The truck module has 6 rules, so looking at this table it would seem the truck module is the dominant one. However when we look at a game in which this AI plays we see something curious. After a couple of years the entire map is filled with airports and airplanes flying everywhere, although there are also some truck and train

Module	Number of rules
truck	6
train	4
aircraft	3
finance	7

Table 6.1: Table shows how many rules are assigned to each module.

routes. Despite having only one build rule, there are a lot of rules adding money to the aircraft budget, allowing it the AI to constantly build aircraft routes at an apparently favorable distance. Other AI's in the same population showed similar behavior.

In fact, the grammar often converged on behavior where the majority of the money was spend on aircraft. One possible reason for this, is that air routes are a lot quicker to build. This because while truck and train routes require a pathfinder algorithm to find an actual route across tiles, the aircraft route only requires two airfield. The AI then only has to give the order, and the planes fly to the other airfield on their own. Pathfinding can be very computationally intensive, especially over longer distances, and AI's are limited to 10.000 op-codes in OpenTTD. That means that sometimes for weeks in the game the AI does nothing except calculate a path. This approach skips that and just builds a ton of airfields instead. While this approach may be profitable, it's not very fun to play against. Since a lot of the AI's from the short grammar tended to focus on aircraft we decided to explore changes in the grammar to encourage other play styles and gather different behavior for our database.

Some of the rules generated here are dead, they will never be triggered. An example is **"if (game.truck_budget > 1.0) truck.RemoveUnprofit( )"** this rule will never trigger, since the truck_budget variable can never be higher than 1. Another rule that is dead in a way is **"if (game.truck_pr_prof <= 0.7) truck.find("GRAI", 500, 500)"** since this rule will only accept routes between stations exactly 500 tiles apart from each other, it will almost never build, despite being triggered.

6.3.1 Single transportation grammars

To broaden the rule database we ran the experiment described in Chapter 5 with 3 different grammars. All of them were restricted to only a single transportation module, so only trucks, trains or aircraft were used. Based on the analysis in Section 6.3 it would seem likely that the aircraft-only grammar will perform best. Figure 6.3 shows that this is indeed the case. The aircraft only grammar quickly rises and converges to a value not significantly different from the short grammar, although lower. Clearly for this type of grammar aircraft are the most profitable.

Trains perform the worst with trucks coming in second. This would confirm our hypothesis that the increased performance of the aircraft module is due to

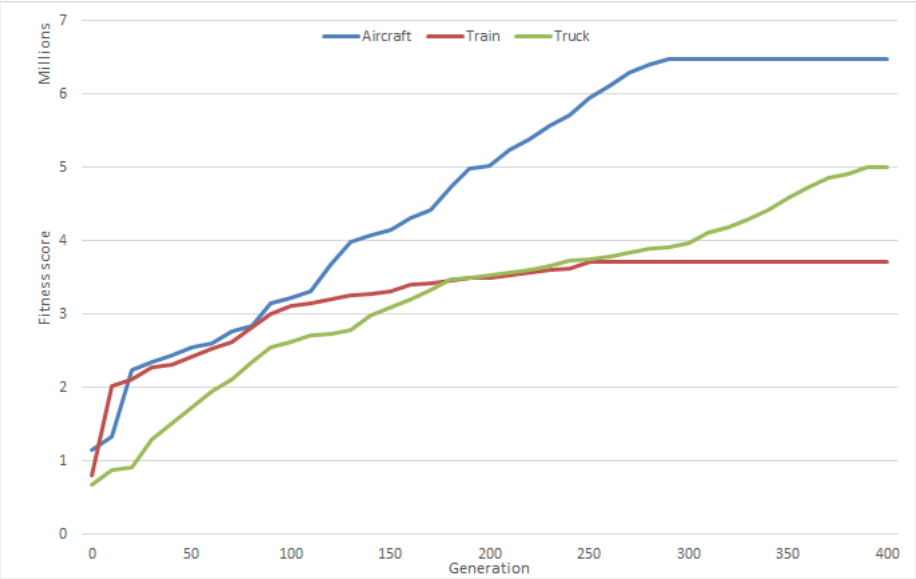

Figure 6.3: This figure shows the best score in the population, averaged over 50 runs, comparing the grammars restricted to one transportation module

the lack of pathfinding, since the pathfinding for trucks is a lot simpler than for trains. Trucks run on roads which are two-way streets, so a single line of road tiles is enough. Train tracks are one-way so they require two separate, non-overlapping, paths, which is more computationally expensive.

Chapter 7

Constructing the Rulebase

Our dynamic scripting algorithm requires a database of rules, from now on referred to as the rulebase. The rulebase will be split into different groups, to ensure that all actions needed are available. Since our goal isn't necessarily to make the most profitable AI possible, but also something that is interesting to play against, we generated different rulesets using different grammars. This was done using a grammatical evolution algorithm, a form of evolutionary learning. The process is described in Chapter 6.

We gathered the programs generated by the final genomes in populations evolved using several different grammars. They were taken from the short grammar and the single transportation grammars.

7.1 Analyzing the rules

We want all transportation modules to be included in the dynamic scripting, so for all three transportation systems we need rules for finding, extending and managing the network, as well as rules for dealing the with budgets of the modules and the debt of the AI as a whole. We analyzed the different programs by looking at the actions taken in them, and how often they occurred. A small script was written which sorted actions and labeled how often they appeared and what the average fitness score was of individuals using them. It would also list examples from rules where this action was used. Note that here we distinguish between the action which would be taken, and the rule as a whole (including if statement). We assume that rules used in many genomes and associated with a high fitness score are good rules. An example of the output from the script is below:

train.find("WOOD", 75, 100) 6350212 20

if (game.train_budget < 0.1) train.find("WOOD", 75, 100) 5729243 1

if (game.train_budget < 0.3) train.find("WOOD", 75, 100) 6382894 19

Here the program lists all cases where the action was trying to find a train route between 75 and 100 tiles long transporting wood. There are 20 rules in total with this action, and their average score is 3350212. This action occurs in two rules, listed next to the rules is how often they occur and the average score of the genomes they occur in. In this case the first rule only occurred once, while the second one occurred 19 times. The average fitness score of the programs in which the second rule occurred was higher than the first one. These two facts combined make the second rule a more likely candidate for inclusion in the dynamic scripting rulebase. Rules with a very high average score, but who only occur in a few genomes are also interesting, since they might be the reason those genomes scored so good.

We found that the combined genomes from the aircraft grammar had fewer unique rules in the their populations. On average about 50, while the truck grammar had around 60 and the train around 80. Possibly this convergence to fewer rules played a role in these genomes performing better.

7.1.1 Short Grammar

The best performing actions in the short grammar add money to the aircraft module budget. This makes sense, since the short grammar strongly favors the aircraft. From the short grammar we took several rules regarding the management of loans and truck and airplane routes, the train module is rarely used in this grammar.

7.1.2 Single Transport

Looking at the rules for the financial module evolved with a single transportation module, the modules appear to be very greedy. Most of the high scoring rules increase the budget for the transportation module with the maximum or a near maximum amount. This probably happened because there was only one transportation module. Since any money not allocated to the module couldn't be used to make routes, it makes sense to transfer most available money to the budget of the transportation module.

The high performing finance rules usually have an if statement with very generic variables. They rarely look at the performance of the module, but more at factors like debt taken and balance left. One exception to this is low budgets, there are several rules that increase the budget for a given transportation module when it drops below a certain (generally low) value.

Looking at the rules for finding new routes there is a clear difference between the aircraft module and the others. The aircraft module has only a handful of distinct find actions, while there are dozens in the truck or train only populations. This probably occurred because the aircraft module is agnostic to cargo, since it can only transport passengers and mail. So the only distinction between the rules is distance used.

The train module on the other hand shows a nice spread across different cargo types and ranges, same as the truck module. Although the truck module

seems to have a preference for transporting coal. Most variables used in the if statements of these rules are _waiting, which have to do with cargo available at the stations, and _rt_build which increases the longer ago it is that a new route was build by that corresponding module.

If-statements for actions regarding the upgrading of routes and removing unprofitable ones generally looked at the profitability of the routes and how much cargo was waiting. It seems that action's value at a certain moment is strongly linked to only a few variables. Future research might look into what the effect of restricting the variables even more would be on the fitness scores achieved by the genetic learning algorithm.

7.2 Gathering rules

We want to create a balanced AI, which uses several different transportation types. Therefore we have gathered the best performing rules for each transportation module. For each module we tried to find a balanced spread of rules. This means we gather several different rules that find or extend routes for different cargoes. We also gathered rules for maintaining our routes, upgrading or removing, and managing the finances, increasing and paying off debt. The final rulebase is shown in Appendix B.

Using this approach we try to avoid "dead" rules. Rules that are never used, for instance a rule trying to extend the oil transporting network, when there are no rules creating an oil transporting network. If the network is never build, it can't be extended, and the AI is burdened. This because the rule takes a spot, but never does anything. It could be replaced by a rule that might be beneficial. A good mix of rules could help prevent this from happening.

For each single transportation grammar we extracted the best 4 rules for the following actions; add_budget, (increase) debt, find (new route), extend (old route), remove unprofitable and upgrade vehicles. So 24 per transportation type, to this we added 15 rules from the short grammar, giving us 89 rules in total.

Chapter 8

Dynamic Scripting

Dynamic scripting is an unsupervised online learning technique [20]. It was first developed by Spronck et al. [20] for use in the Computer RolePlaying Game Neverwinter Nights. They gave four requirements they wanted their algorithm to fulfill;

1. Fast, this excludes most model-based learning.
2. Effective, meaning it's at least as challenging as manually designed scripts.
3. Robust, capable of dealing with randomness.
4. Efficient, it should require few trials.

In another paper four additional, functional, requirements are listed [26]:

1. Clarity, results must be easily interpretable.
2. Variety, the AI must be capable of a variety of different behavior.
3. Consistency, the AI results should have little variance.
4. Scalability, the AI must scale with the results of the human player.

To meet these requirements a 'high performance' algorithm is required [15]. High performance algorithms require two things [15]; exclusion of randomness, and domain-specific knowledge. Dynamic scripting was developed to be able to meet these requirements.

The algorithm is a reinforcement learning technique. It has been adapted however, since traditional reinforcement learning doesn't satisfy the efficiency criteria [20], simple reinforcement learning systems generally don't perform well in games. It uses on-policy value iteration to learn state-action values. These values are exclusively based on a reward signal, maximising immediate rewards is it's only concern [28] and it does not try to model opponents.

Action	Description	Transport module
add_budget	Add money to the budget of a transportation module	*
find	Finds a new route	*
extend	Extends an old route	*
RemoveUnprofitable	Removes unprofitable routes	*
UpgradeVehicles	Upgrades vehicles on routes	*
debt	Increases or decreases company debt	

8.1 Components

The algorithm maintains a rule database, a rulebase, split into rules for different components. Everytime an AI is needed it is created by combining rules from these different components. In our research we split rules according to the action they trigger. A balanced AI should have at least one rule triggering each action. Since all of them represent a vital game function. The following groups have been created, those marked with a * have different versions for each transportation module:

In the rulebase each of these rules has a weight. The probability that a rule is selected for play is equal to its weight. The weights of all rules in a component group sum to 1. By changing the weight values in the rulebase the algorithm is able to adapts to changing circumstances. For this approach to work, all, or at least most, rules need to define sensible behavior.

8.2 Scoring

At the end of a game the score of the AI is evaluated using the fitness function defined in Section 5.3. The AI plays against an opponent, so the opponent's fitness score is then determined in the same way. After this the final score is calculated by subtracting the opponent's score from the AI's score.

The score is then normalized to the interval [0,1]. Here 0 represents a final score of -1.000.000 and 1 a score of 2.000.000. The game is considered a win if the AI scored higher than it's opponent, resulting in a positive final score.

After the score is determined, the weights of the rules are adjusted. This is done using the following algorithm:

$$w = \begin{cases} max(0, W_{org} - MP * \frac{b-F}{b}) & F < b \\ min(W_{org} + MR * \frac{F-b}{1-b}, 1)) & F \geq b \end{cases}$$

F is the normalized score, b is the threshold for winning. B is set at 0.3 in our research. When the AI made more profit than it's opponent F will be bigger than 0.3, triggering the bottom rule, otherwise the top one is used. MP is the maximum penalty and MR the maximum reward, both are set at 0.2. W_{org} is the original weight of the rule and W the new one.

After updating the activated rule, the other rules in the same component need to be updated as well. This way we ensure that the sum of all weights stays 1. The following function is used:

$$w = \begin{cases} W_{org} + (c * \frac{W_{org}}{\sum_{i=1}^{n} w_i}) & F < b \\ W_{org} - (c * \frac{W_{org}}{\sum_{i=1}^{n} w_i}) & F \geq b \end{cases}$$

c is the (absolute) change in weight the activated rule had, $\sum_{i=1}^{n} w_i$ is the sum of weights of all n unused rules.

8.3 Rules chosen

In Chapter 7 we discussed how we assembled the rulebase for use in dynamic scripting. For each of groups listed in Table 8.1 we have at least four rules in the rulebase, and we pick one from each group each game. The only exception is the debt group, there we pick 2 rules. This leads to a total of 17 rules. Comparable to the AI's generated by the grammatical evolution described in Chapter 6. Those AI's had 20 rules generated, however they often also contained several dead rules. Causing them to have fewer than 20 effective rules.

8.4 Experiment

To asses the adaptive performance of the algorithm we let it play against several static AI's developed for OpenTTD. All the AI's discussed in Chapter 2 will be used as opponents. Here a quick recap of the opponents:

PathZilla, uses graph algorithms to find a good path between cities.

trAIns, sophisticated use of trains.

ChooChoo, build huge connected railway networks

AdmiralAI, strives to be an AI capable of using everything.

While using players would be the ideal way to evaluate how dynamic the AI is, due to time constraints we have chosen to test them against strong static opponents instead. This is a common strategy for testing dynamic scripting algorithms [14, 28, 13].

Every round the AI played against an opponent the score was logged. If the average score over the last 10 games was over 3.000.000, the algorithm was stopped, since it was capable of reliably beating the opponent. Otherwise the algorithm was allowed to run for 50 runs before being stopped.

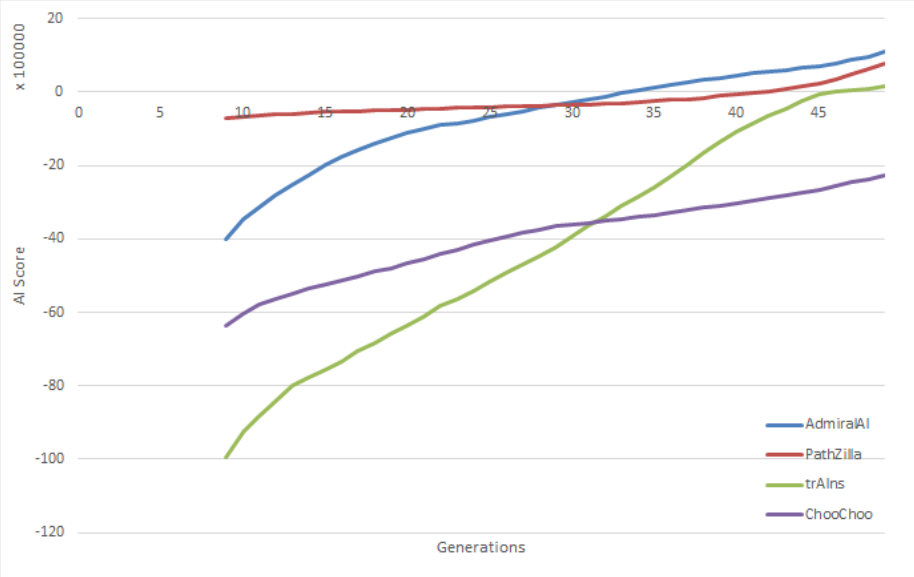

Figure 8.1: Average score from the last 10 generations. Averaged over 50 runs of the algorithm.

8.5 Results

Experiments were run 50 times each, the result were then averaged. We tracked the AI's final score after each round, this is the score we calculate by subtracting it's opponent's game score from the AI's. These results are shown in Figure 8.1. Against all the four different opponents the generated AI's show progress and in all but one case it beats them in the end.

We don't know for sure why it's unable to beat ChooChoo, but we theorize it's because ChooChoo builds large railnetworks, possibly blocking routes the AI would otherwise take. This would result in very long pathfinding, since the pathfinding algorithm tries to go around obstacles before going over them, because building over other roads/tracks is very expensive. The AI is unable to use the strategy that was evolved in Chapter 6, where it just build lots of aircraft. This because the program generated by the dynamic scripting always has transport rules for every type.

The other three AI's the algorithm can beat. Although it's only barely capable of beating the trAIns AI. Since trAIns is an AI that is focused on trains, it could be because trAIns also blocks a lot of routes, similar to ChooChoo. The other two AI's which are not as trains focused it can beat reliably, with AdmiralAI being the easiest one to beat. It's worth noting that none of the evolved programs from Chapter 5 were able to beat any of the static approaches. They probably contained to many dead rules to be competitive.

8.6 Future research

While the AI's generated by dynamic scripting were capable of beating 3 out of 4 static AI's, there is still room for improvement. One possible direction is offered by research of Spronck et. al [28]. They compared different rule ordering mechanisms. The default rule ordering by Ponsen et al.[14], also used here, was compared with two other mechanisms.

The first is called Relation-Weight Ordering, in this approach a relation-weight table is stored. For each combination of rules it stores whether they have a positive or a negative effect on each other. Rules are ordered so as to maximize the weights from the table.

The second was Relation-Weight Ordering with Selection Bonus. This mechanism works the same as regular relation-weight ordering, except for the rule selection. If a rule is chosen, we look at the rule in the relation-weight table with whom it has the best relation (highest weight). If this rule is not yet added to the grammar, it will receive a bonus on its probability of being picked.

In their research Ponsen et al. showed these techniques to be effective and their is reason to believe they would work in this domain. Since in the game many rules depend on each other, for instance an extend rule for coal is useless if a coal rule is never build. A form of relation-weight ordering could help find these links.

Chapter 9

Conclusion

The goal of this thesis is to use dynamic scripting and evolutionary learning to develop an dynamic AI for the game OpenTTD. The database of rules required by the dynamic scripting algorithm was created using a Grammatical Evolution algorithm. Grammatical Evolution is a type of Evolutionary Learning where genomes represent programs. During the development of the database we looked at the effects of differently structured grammars and genomes.

We found that using domain knowledge to restrict variables had an enormous effect on learning. Reinforcing the notion that domain knowledge is very important when developing these types of algorithms [15]. Possible future work could look into restricting variables even more by tying specific variables to specific actions.

Another important change we made was to the structure of the genome. Traditional grammatical evolution solutions view the genome has a single program, our approach was to split the genome into parts. Each part encoding a single rule. Our hope was that this would increase the locality of the genomes, by protecting most of the rules in the program during crossover or mutation. Despite both approaches being capable of generating the same programs, the version with a split genome performed more than twice as good.

Finally we compared different single transport grammars, where only one type of transportation was allowed. While none of the resulting programs performed better than the version with all transportation modules, the resulting grammars were a valuable source of rules for our rulebase.

We used the resulting programs to develop a rulebase. Currently these rules were handpicked. While picking them rules we analyzed the rules in the programs by looking at how often they occurred across different genomes, and the average fitness score of those genomes. Rules that had a high average fitness, or occurred in many programs were considered to play a role in their program's success. Rules were picked looking at a combination of these two factors and human judgment about the suitability of this rule. Future research could look into developing algorithms to automate or at least formalize this process.

After constructing our rulebase we tested our dynamic scripting algorithm

against four different static opponents. The algorithm was capable of beating three. Although this is a good score there was room for improvement. In Section 8.6 we discuss an alternative rule ordering mechanism that might improve performance.

However our goal was not just to develop the highest scoring AI financially but rather develop an AI that is fun to play against. We feel that dynamic scripting offers a higher entertainment value since the AI is capable of constantly changing it's tactics, keeping players on edge. One interesting development is using dynamic scripting to make automatically scaling opponents for human players [26]. This could be done by changing the fitness function used and is an avenue for possible future research.

Appendix A

Grammars

The grammars represented below are cleaned up for readability. The actual programs run have truck, train and aircraft replaced with game._truck_manager, game._train_manager and game._aircraft_manager respectively. Similarly all variables have game. appended to them. The game variable passed along allows the AI to send commands to and read variables from various parts of the AI program.

A.1 Short Grammar

<rule> ::= <money> | <truck> | <rail> | <aircraft>

<truck> ::= if (truck<route_var> <equals> <num>) truck.<build> | if (truck<route_var> <equals> <num>) truck.<maintain>

<rail> ::= if (train<route_var> <equals> <num>) train.<build> | if (train<route_var> <equals> <num>) train.<maintain>

<aircraft> ::= if (aircraft<route_var> <equals> <num>) aircraft.<build> | if (aircraft<route_var> <equals> <num>) aircraft.<maintain>

<build> ::= <find> | <extend>

<find> ::= find (<cargo> <distance> <distance>)

<extend> ::= extend (<cargo> <distance> <distance>)

<maintain> ::= <upgrade_vehicle> | <remove_unprofitable>

<upgrade_vehicle> ::= UpgradeVehicles()

<remove_unprofitable> ::= RemoveUnprofit()

<money> ::= if (<fin_var> <equals> <num>) game.<debt> | if (<fin_var> <equals> <num>) game.<increase_budget>

<debt> ::= debt (<boolean> <amount>)

<increase_budget> ::= add_budget(<num>, <trans_budget>)

<cargo> ::= "PASS" | "IRON" | "OIL_" | "WOOD" | "GRAI" | "COAL"

<fin_var> ::= truck_profit | aircraft_profit | train_profit | game.balance_left | game.debt_taken | truck_budget | train_budget | aircraft_budget

<route_var> ::= _waiting | _pr_prof | _rt_build | _profit | _budget

<num> ::= 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0

<equals> ::= < | > | ≤ | ≥

<distance> ::= 25 | 50 | 75 | 100 | 150 | 200 | 300 | 400 | 500

<boolean> ::= 0 | 1

<trans_manager> ::= truck | train | aircraft

<trans_budget> ::= "truck" | "train" | "aircraft"

This is the short grammar, the Grammar itself is described in more detail in Chapter 4.

A.2 Long Grammar

<rule> ::= <money> <rule> | <truck> <rule> | <rail> <rule> | <aircraft> <rule>

The rest of the grammar is the same as for the Short Grammar

A.3 Unrestricted Variables

<rule> ::= <money> | <if> <trans_manager>.<build> | <if> <trans_manager>.<maintain>

<if> ::= if (<fin_var> <equals> <num>) | if (<route_var> <equals> <num>)

<build> ::= <find> | <extend>

<find> ::= find(<cargo>, <distance>, <distance>)

<extend> ::= extend(<cargo>, <distance>, <distance>)

<maintain> ::= <upgrade_vehicle> | <remove_unprofitable>

<upgrade_vehicle> ::= <trans_manager>.upgradeVehicles()

<remove_unprofitable> ::= <trans_manager>.removeUnprofitable( )

<money> ::= <if> game.<debt> | <if> game.<increase_budget>

<debt> ::= debt(<num>, <boolean>)

<increase_budget> ::= add_budget(<num>, <trans_budget>)

<cargo> ::= "PASS" | "IRON" | "OIL_" | "WOOD" | "GRAI" | "COAL"

<fin_var> ::= truck_profit |aircraft_profit | train_profit | balance_left
| debt_taken | truck_profit | train_profit | aircraft_profit

<route_var> ::= truck_waiting | train_waiting | aircraft_waiting | truck_pr_prof
| train_pr_prof | aircraft_pr_prof | aircraft_rt_build | train_rt_build |
truck_rt_build | truck_profit |aircraft_profit | train_profit | truck_profit
| train_profit | aircraft_profit

<num> ::= 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0

<equals> ::= < | > | <= | >=

<distance> ::= 25 | 50 | 75 | 100 | 150 | 200 | 300 | 400 | 500

<boolean> ::= 0 | 1

<trans_manager> ::= train | truck | aircraft

<trans_budget> ::= "road" | "train" | "aircraft"

Appendix B

Rule database

B.1 Short Grammar Rules

if (manager.balance_left > 0.6) manager.debt(0.6, 1)

if (manager.debt_taken < 0.2) manager.debt(0.4, 0)

if (manager.truck_budget < 0.4) manager._truck_manager.find("OIL_", 150, 100)

if (manager.train_rt_build < 0.1) manager._train_manager.find("WOOD", 750, 50)

if (manager.aircraft_budget > 0.3) manager._aircraft_manager.find("IRON", 400, 100)

if (manager.truck_budget < 1.0) manager._truck_manager.find("COAL", 300, 50)

if (manager.aircraft_budget >= 0.7) manager._aircraft_manager.extend("WOOD", 400, 300)

if (manager.truck_profit <= 0.7) manager._truck_manager.extend("COAL", 1000, 750)

if (manager.aircraft_budget > 0.5) manager._train_manager.extend("COAL", 150, 75)

if (manager.train_rt_build < 0.2) manager._train_manager.RemoveUnprofit( )

if (manager.aircraft_waiting < 0.6) manager._aircraft_manager.RemoveUnprofit( )

if (manager.truck_profit <= 0.4) manager._truck_manager.RemoveUnprofit( )

if (manager.train_profit > 0.9) manager._train_manager.UpgradeVehicles()

if (manager.truck_waiting > 0.3) manager._truck_manager.UpgradeVehicles()

if (manager.aircraft_pr_prof >= 0.9) manager._aircraft_manager.UpgradeVehicles()

B.2 Aircraft only rules

if (manager.balance_left > 0.4) manager.add_budget(0.6, "aircraft")

if (manager.balance_left <= 0.3) manager.add_budget(0.1, "aircraft")

if (manager.balance_left <= 0.4) manager.add_budget(0.7, "aircraft")

if (manager.aircraft_budget >= 0.1) manager.add_budget(0.5, "aircraft")

if (manager.aircraft_budget > 0.9) manager.debt(0.9, 0)

if (manager.balance_left >= 0.7) manager.debt(0.8, 0)

if (manager.balance_left >= 0.2) manager.debt(0.1, 0)

if (manager.debt_taken < 0.1) manager.debt(0.2, 1)

if (manager.aircraft_rt_build < 0.4) manager._aircraft_manager.find("COAL", 300, 200)

if (manager.aircraft_pr_prof <= 0.6) manager._aircraft_manager.find("GRAI", 100, 500)

if (manager.aircraft_budget <= 0.6) manager._aircraft_manager.find("GRAI", 300, 500)

if (manager.aircraft_rt_build < 0.4) manager._aircraft_manager.find("COAL", 300, 75)

if (manager.aircraft_budget < 0.4) manager._aircraft_manager.extend("WOOD", 500, 300)

if (manager.aircraft_profit <= 0.3) manager._aircraft_manager.extend("PASS", 100, 25)

if (manager.aircraft_budget <= 0.2) manager._aircraft_manager.extend("IRON", 100, 75)

if (manager.aircraft_budget <= 0.7) manager._aircraft_manager.extend("GRAI", 500, 200)

```
if ( manager.aircraft_profit > 0.6 ) manager._aircraft_manager.RemoveUnprofit(
)

if ( manager.aircraft_rt_build < 0.2 ) manager._aircraft_manager.RemoveUnprofit(
)

if ( manager.aircraft_budget < 0.5 ) manager._aircraft_manager.RemoveUnprofit(
)

if ( manager.aircraft_pr_prof >= 0.2 ) manager._aircraft_manager.RemoveUnprofit(
)

if ( manager.aircraft_rt_build <= 0.2 ) manager._aircraft_manager.UpgradeVehicles()

if ( manager.aircraft_budget >= 0.1 ) manager._aircraft_manager.UpgradeVehicles()

if ( manager.aircraft_waiting < 0.6 ) manager._aircraft_manager.UpgradeVehicles()

if ( manager.aircraft_profit < 0.3 ) manager._aircraft_manager.UpgradeVehicles()
```

B.3 Train only rules

```
if ( manager.train_budget <= 0.4 )  manager.add_budget( 0.1, "train" )

if ( manager.debt_taken < 0.9 )  manager.add_budget( 0.9, "train" )

if ( manager.balance_left > 0.9 )  manager.add_budget( 0.9, "train" )

if ( manager.debt_taken < 0.7 )  manager.add_budget( 0.3, "train" )

if ( manager.debt_taken < 0.7 )  manager.debt( 0.7, 1 )

if ( manager.balance_left < 0.8 )  manager.debt( 0.3, 1 )

if ( manager.debt_taken >= 0.6 )  manager.debt( 0.1, 0 )

if ( manager.train_profit <= 0.1 )  manager.debt( 0.7, 1 )

if ( manager.train_waiting <= 0.3 ) manager._train_manager.find( "PASS",
50, 300 )

if ( manager.train_rt_build < 0.4 ) manager._train_manager.find( "WOOD",
75, 300 )

if ( manager.train_waiting <= 0.2 ) manager._train_manager.find( "COAL",
400, 100 )

if ( manager.train_rt_build < 0.8 ) manager._train_manager.find( "OIL_",
100, 150 )

if ( manager.train_budget <= 0.4 ) manager._train_manager.extend(
"COAL", 25, 400 )
```

if (manager.train_waiting >= 0.4) manager._train_manager.extend("WOOD", 25, 500)

if (manager.train_waiting < 0.2) manager._train_manager.extend("PASS", 150, 200)

if (manager.train_waiting >= 0.6) manager._train_manager.extend("OIL_", 75, 300)

if (manager.train_rt_build >= 0.1) manager._train_manager.RemoveUnprofit( )

if (manager.train_waiting > 0.5) manager._train_manager.RemoveUnprofit( )

if (manager.train_waiting <= 0.6) manager._train_manager.RemoveUnprofit( )

if (manager.train_profit < 0.9) manager._train_manager.RemoveUnprofit( )

if (manager.train_profit >= 0.3) manager._train_manager.UpgradeVehicles()

if (manager.train_rt_build > 0.2) manager._train_manager.UpgradeVehicles()

if (manager.train_waiting <= 0.3) manager._train_manager.UpgradeVehicles()

if (manager.train_pr_prof >= 0.1) manager._train_manager.UpgradeVehicles()

B.4 Truck only rules

if (manager.truck_budget <= 0.2) manager.add_budget(0.9, "truck")

if (manager.debt_taken > 0.2) manager.add_budget(0.9, "truck")

if (manager.debt_taken < 0.6) manager.add_budget(0.8, "truck")

if (manager.debt_taken <= 0.4) manager.add_budget(0.4, "truck")

if (manager.balance_left >= 0.6) manager.debt(0.5, 0)

if (manager.balance_left >= 0.4) manager.debt(0.1, 1)

if (manager.debt_taken <= 0.3) manager.debt(0.2, 1)

if (manager.debt_taken >= 0.5) manager.debt(0.8, 1)

if (manager.truck_waiting > 0.4) manager._truck_manager.find("COAL", 25, 100)

if (manager.truck_rt_build > 0.2) manager._truck_manager.find("GRAI", 25, 100)

if (manager.truck_budget >= 0.5) manager._truck_manager.find("PASS", 100, 50)

if (manager.truck_rt_build <= 0.5) manager._truck_manager.find("PASS", 25, 75)

if (manager.truck_rt_build >= 0.6) manager._truck_manager.extend("PASS", 500, 75)

if (manager.truck_waiting > 0.4) manager._truck_manager.extend("GRAI", 500, 100)

if (manager.truck_profit < 0.7) manager._truck_manager.extend("COAL", 500, 300)

if (manager.truck_waiting <= 0.5) manager._truck_manager.extend("PASS", 25, 75)

if (manager.truck_waiting <= 0.9) manager._truck_manager.RemoveUnprofit( )

if (manager.truck_waiting <= 0.2) manager._truck_manager.RemoveUnprofit( )

if (manager.truck_pr_prof < 0.2) manager._truck_manager.RemoveUnprofit( )

if (manager.truck_budget >= 0.3) manager._truck_manager.RemoveUnprofit( )

if (manager.truck_budget >= 0.3) manager._truck_manager.UpgradeVehicles()

if (manager.truck_budget < 0.3) manager._truck_manager.UpgradeVehicles()

if (manager.truck_waiting > 0.4) manager._truck_manager.UpgradeVehicles()

if (manager.truck_waiting > 0.1) manager._truck_manager.UpgradeVehicles()

Bibliography

[1] Pathzilla (v6) - a networking ai. `http://www.tt-forums.net/viewtopic.php?f=65&t=38645`. [Online; accessed 05-June-2014].

[2] Chang Wook Ahn and Rudrapatna S Ramakrishna. A genetic algorithm for shortest path routing problem and the sizing of populations. *Evolutionary Computation, IEEE Transactions on*, 6(6):566–579, 2002.

[3] Peter J Angeline. Subtree crossover: Building block engine or macromutation. *Genetic Programming*, 97:9–17, 1997.

[4] Tom Castle and C. G. Johnson. Positional effect of crossover and mutation in Grammatical Evolution. In Anna I. Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and Sima Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, Istanbul, April 2010. EvoStar, Springer.

[5] Edgar Galván-López, John Mark Swafford, Michael OâĂŹNeill, and Anthony Brabazon. *Evolving a ms. pacman controller using grammatical evolution*, pages 161–170. Springer, 2010.

[6] Edgar Galván-López, John Mark Swafford, Michael O?Neill, and Anthony Brabazon. Evolving a ms. pacman controller using grammatical evolution. In *Applications of Evolutionary Computation*, pages 161–170. Springer, 2010.

[7] Andreas Geyer-Schulz and A Geyer-Schulz. *Fuzzy rule-based expert systems and genetic machine learning*. Physica-Verlag Heidelberg, 1997.

[8] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Number 2. Addison-Wesley, Reading, MA, 1989.

[9] Maarten Keijzer, Michael OŃeill, Conor Ryan, and Mike Cattolico. *Grammatical evolution rules The mod and the bucket rule*, pages 123–130. Springer, 2002.

[10] Maarten Keijzer, Conor Ryan, Michael OâĂŹNeill, Mike Cattolico, and Vladan Babovic. *Ripple crossover in genetic programming*, pages 74–86. Springer, 2001.

[11] John R Koza. *Genetic programming*. Citeseer, 1992.

[12] David Lindh. Adaptive combat ai in strategy games - an approach based on dynamic scripting.

[13] Jeremy Ludwig and Arthur Farley. Examining extended dynamic scripting in a tactical game framework. In *AIIDE*, 2009.

[14] P. Spronck M. Ponsen. Improving adaptive game ai with evolutionary learning. *Computer Games: Artificial Intelligence, Design and Education*, pages 389–396, 2004.

[15] Zbigniew Michalewicz and David B Fogel. *How to solve it: modern heuristics*. Springer, 2004.

[16] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. *Advances in Genetic Programming*, chapter Explicitly Defined Introns and Destructive Crossover in Genetic Programming, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

[17] Michael O'Neil and Conor Ryan. Grammatical evolution. In *Grammatical Evolution*, pages 33–47. Springer, 2003.

[18] Michael O'Neill and Conor Ryan. Evolving multi-line compilable c programs. In *Genetic Programming*, pages 83–92. Springer, 1999.

[19] Michael O'Neill, Conor Ryan, Maarten Keijzer, and Mike Cattolico. Crossover in grammatical evolution. *Genetic programming and evolvable machines*, 4(1):67–93, 2003.

[20] Ida Sprinkhuizen-Kuyper Pieter Spronck and Eric Postma. Online adaptation of game opponent ai in simulation and in practice. *International Journal of Intelligent Games and Simulation*, 3(1):45–53, March/April 2004.

[21] Luis Henrique Oliveira Rios and Luiz Chaimowicz. trains: An artificial inteligence for openttd. In *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*, pages 52–63. IEEE, 2009.

[22] Peter Alexander Whigham Yin Shan Michael O?Neill Robert I (Bob) McKay, Nguyen Xuan Hoai. Grammar-based genetic programming : a survey.

[23] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, 1995.

[24] Conor Ryan, JJ Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, pages 83–96. Springer, 1998.

[25] Jonathan Schaeffer. A gamut of games. *AI Magazine*, 22(3):29, 2001.

[26] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game ai with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.

[27] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Enhancing the performance of dynamic scripting in computer games. pages 296–307, 2004.

[28] Timor Timuri, Pieter Spronck, and H Jaap van den Herik. Automatic rule ordering for dynamic scripting. *AIIDE*, 1:49–54, 2007.

[29] Paul Tozour. The evolution of game ai. *AI game programming wisdom*, 1:3–15, 2002.

[30] Peter A Whigham et al. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pages 33–41. Citeseer, 1995.

[31] Man Leung Wong and Kwong Sak Leung. Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180, 1997.

[32] Yufei Yuan and Huijun Zhuang. A genetic algorithm for generating fuzzy classification rules. *Fuzzy sets and systems*, 84(1):1–19, 1996.