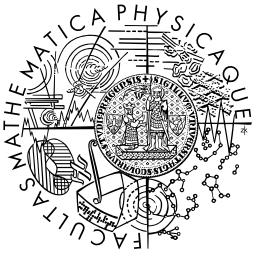

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Ondřej Lakomý

Planning of railway network for Open Transport Tycoon Deluxe

Department of Software and Computer Science Education

Supervisor of the master thesis:	Mgr. Jakub Gemrot, Ph.D.
Study programme:	Computer Science
Study branch:	Computer Graphics and Game Development

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, 5th January 2020 signature of the author

I want to express my gratitude to Mgr. Jakub Gemrot, Ph.D. for his valuable advices throughout the work. I also wish to thank Prof. RNDr. Roman Barták, Ph.D., for his ideas that helped me a lot.

Title: Planning of railway network for Open Transport Tycoon Deluxe

Author: Ondřej Lakomý

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: In a business simulation game Open Transport Tycoon Deluxe, players can play the game alone or can compete against other players or against artificial intelligence. There is plenty of artificial intelligences that can be downloaded and used in the game. Usually the AI is simple and does not build complex transfer routes. Inspired by OpenTTDCoop, a community organisation which specializes on building complex railway networks, new artificial intelligence has been developed to fill the gap in the available artificial intelligence list - TrainsferAI. TrainsferAI is capable of planning, building and expanding its own railway network. It follows a set of rules based on the OpenTTDCoop rules for players. TrainsferAI builds network consisting of stations, routes and crossroads using A* searching algorithm in a couple of variations. The network should be complex and on a human-like level.

Keywords: Transport Tycoon, Open Transport Tycoon Deluxe, OpenTTD, OTTD, AI, artificial intelligence, trains, network, railway, A*, AStar

Contents

1. Introduction

There are a lot of games these days. One of the basic category is strategy games. They are popular among players because of various aspect - a lot of room for creativity, players are also required to think outside of the box sometimes to reach their goals. Playing strategy games is in general quite complex task. Using artificial intelligence (AI) in such games is therefore quite challenging. It needs to cover large variety of game mechanics and usually there are many ways of reaching the goal, there is as well plenty of room for developer of such AI and there are huge differences between good and bad AI agent.

AI is capable of playing strategy games really well (as showed for example when AI called AlphaStar beat two professional players - MaNa and TLO - in game Starcraft 2) (Coldewey, D. [2019]). But those strengths are often more of an efficiency of the algorithms and ability to process huge amount of data in a very short amount of time. The challenge for AI starts when it is not required to be efficient, but to do something else - for example be creative.

The goal of the thesis is to figure out if an AI agent can behave in a creative and human-like way. If we let the AI to face a problem which requires a certain level of creativity, will the AI be capable of solving such a problem? If we let the AI agent face the problem, can the AI behave human-like? Can it build things that player would build? Can we determine if the result is created by human or by AI? Can the AI agent even reach that state?

We will test this on genre of games which is called business simulation games. There are a lot of them since the genre started to pop up, but one particularly important for the history of this genre of games will be used. Its name is Transport Tycoon. It was second game belonging to a group of games called "Tycoon games" (which does include for example Zoo tycoon, Roller coaster tycoon and many more). The player was put in the game as a general manager of a transport company. The main goal was to earn money by transporting various cargoes between industries or passengers and mail between cities. Player was given four types of transport which he could use to transport cargo - road vehicles, aircraft, ships and trains, where each category has its advantages and disadvantages. What really kept players in the game, was building trains. It was the most creative, rewarding and pleasing part of the game. To run trains, player was required to build two stations and railway connection from first station to the second. Only if the stations were connected, the train could then transfer the cargo. If players figured out that they can join those rail connections and make some more complex rail networks, the true fun and challenge did not let the player stop playing. Building railway networks required players not only to know what to connect. It required the player to behave creatively in the game. He needed to figure out how to efficiently join connections and how to ensure the all trains can get to their target stations. So we let the AI agent to build some human-like looking train networks, which are non-trivial - trains from more then one station will travel on the same parts of the network. While building network and fulfilling the human-like aspect, the AI agent still needs to earn money in order not to bankrupt.

1.1 Railway networks

Building a railway network may seem not to be a difficult task. In order to explain its complexity, lets start from the beginning. To build a connection from one station to another station. player will just build some rails leading from the first station to the second station.

The next step in building of network is to connect another station to the existing connection, so the new station will reuse some rails from the first connection (so some money are saved during the construction). However, trains cannot pass through each other - they need to wait for the trains in front of them to go through the rails one by one. This process is controlled by rail signals. Because of this, the basic solution to build a connection is to in fact build two rails - one leading to the station and second going the other way. Then trains going to the station can all take one rail and the trains going back can take the other one.

Important term in network building is capacity of the connection. Each train has its length. If the connection is 100 tiles long and each train has length of 4, we can fit maximum of 25 trains to the connection at any time. We can see that this 2-rails system works quite well if there are not many trains using the roads. But if you want to reach for example a couple of hundreds of trains in the network, the connections will soon not be enough for that traffic.

If we want to increase the capacity of the connection, we can add more rails going in each direction. With each rail added, we need to expand all crossroads or joins in the network for each rail - we want the train to get from each rail to whatever point in the network, so the train can choose any rail. While building more rails on a connection is quite straight-forward, building the crossroads is the real challenge. Trains need not to slow down, otherwise they will get stuck and the network will get jammed. Further different rails are not allowed to cross each other on the same height, otherwise train from one rail would need to wait for the train from the second rail to pass before he could use it, causing jams. We then need to connect each rail with one rail in each direction of the crossroad. If we have 3 directions coming to the crossroad and each has 4 rails going in and 4 rails going out of the crossroad, we need to make 24 connections on quite a small space. For better idea a quite big crossroad is shown on image 1.1.

The construction process of such crossroad is not simple. It often includes many iterations of improving the crossroad and removing its bottlenecks. Other important factor is that while player is building it, he can easily move things around if necessary because player sees the crossroad as a complete object. AI see this same problem as a pathfinding problem and for an AI agent it is very hard to ”look around” to modify different rails. Because of these limitations this thesis will not attempt to build such big crossroads and instead of it it will focus to build simpler versions of the network with limited amount of rails as it still looks human-like and the more complex variants does not add more ”human-like” property to the final network.

1.2 Thesis Structure

On first pages of the thesis, the game Transport Tycoon game will be introduced. Next step will reveal the technology used within the game and the API provided

Figure 1.1: Example of crossroad built by community.

by the game to develop the AI. The main part of the thesis will then describe the TrainsferAI, which is AI for Transport Tycoon developed for this thesis. It will show the specification of the AI and its life cycle in the game. It will describe internal data representation and list of algorithms used. After that a validation of the resulting AI will be showed. By the end of the thesis, future work will be presented, introducing some areas of the AI which could be added or improved in the future.

1.2.1 Scope of the thesis

The main goal of the thesis is to develop an AI agent called TrainsferAI, capable of building human-like railway network structure in the game Transport Tycoon. It will earn money and use it connect other facilities to the network. The AI will use existing network as much as possible, avoiding building different tracks if the connections is already built.

1.2.2 Out of scope of the thesis

The created AI is not considered to be the same type of AI as others - while other existing AIs are often developed with a goal of providing viable opponent to players, this is not a target state for TrainsferAI. It is not a goal to develop an AI which can be played on various world and game settings - there are plenty of AIs with these properties. TrainsferAI is best played as the only company on the map and its sense is in demonstrating its building abilities rather then its competitiveness.

2. OpenTTD

Transport Tycoon was originally released for computers in 1994 by Chris Sawyer. Next version of the game was released year later, called Transport Tycoon Deluxe. It included new landscaping, weather, new map types and vehicles. There were not many computer games and the technology for developing them was very young - the game was written in X86 Assembly language ([Sawyer, C., 2013]).

Transport Tycoon is very old game and author stopped working on it shortly after its release. Players liked the game that much that in March 2004, an open-source version of this game was released and called Open Transport Tycoon Deluxe (OpenTTD). It copied the original game and was developed by OpenTTD Team. Its purpose was to replicate the game and enhance it so players can do particular tasks more easily and more user-friendly. OpenTTD Team is keeping the game up-to-date with updates, adding more functionalities, fixing bugs and adding more detailed graphics. In addition to it, the game community is adding new graphics, styles, vehicles and a lot more to the game [1]. This content is free to download through the game manager. All those features make the game really customizable. OpenTTD has everything original Transport Tycoon could offer and many more, so these days OpenTTD took the place of the original game ([OpenTTD Team, 2019]).

The principle of the game is quite simple - the player is given a transport company and a loan. He is then free to use those money to build vehicles and transport various different cargoes. If the player is not managing his bank balance well, his company may bankrupt and the game ends. On the other hand, if player is able to profit, he can build more transport routes, buy new vehicles and expand his company.

The game offers 4 environments for the player to choose from. The first one is temperate environment. This provides map set which is actually very similar to temperate zone. The second one is snowy mountain environment, which provides map set inspired by mountain landscape. The third one is desert environment. The last one is inspired by children - trees are replaced by big lollypops and you can transport cotton candy for example. This environment is probably the least liked by the players.

2.1 Game goals

OpenTTD game starts at game year 1950. Player then has one set goal of the game - reach game year 2050 without bankrupting his company. For first feel playing, it may not be very clear how to keep the company in green numbers. But after a few attempts, average player will figure out how to earn money and reaching year 2050 is not that difficult. Since for more experienced players this game goal is not that big of a challenge, they started to think up their own goals among the game.

One of the common ones is to earn as much money as possible. One widely used variant of this goal is to earn money as quickly as possible. Player is then

[1] Community-made content can be downloaded to the game by in-game online content manager

forced to figure out the most efficient way of earning money at the beginning of the game and he needs to maximize the cost/income ratio. Similar game goal is to build as many vehicles as possible. Some players do look for other things in games then efficiency. Those players can then opt for building the world to look realistic or aesthetically pleasing. The outcomes of such games differ between each players and that makes the game even more interesting and replayable.

2.2 Vehicles

Vehicles are substantial part of the game. They are used to transport passengers, mail or industrial cargoes from its production point to its accepting point (usually some industries or cities). They are the only way player can earn money in the game. There are four categories of vehicles in the game: road vehicles, aircraft, ships and trains. Each vehicle has some properties - maintenance cost, capacity, reliability, service life, maximal speed and horse power. Those properties influence the choice of the vehicles. As the game progresses further, new types of vehicles are available to the player (usually the later the vehicle is available, the better it is).

2.2.1 Breakdowns and servicing

Breakdown of vehicle is a game mechanic which influences how the vehicle operates. If a vehicle breaks down, it is stopped for a couple of game days. This mechanic is different with aircraft - it does not have a breakdown, but instead an accident occasionally appears and the plane is destroyed completely. The higher reliability of the vehicle, the lower chance that vehicle will break down. Another closely connected game mechanic is servicing. Each vehicle has a servicing interval and after the interval expires, it goes to the closest depot for service. The more days the vehicle has not been serviced, the higher chance of breakdown. The last property that can increase chance of breakdown is service time. It is an amount of years the vehicle can operate without increased breakdown chance. After that amount of years, player should sell the vehicle and buy a new one, or the old vehicle will break more often. Breakdowns and servicing can be turned on and off in the game settings.

2.2.2 Types of vehicles

Road vehicles require road to travel on them and bus/truck stations to load/unload cargo. Each vehicle is very cheap and slow. They have limited capacity so usually do not make much profit. Road vehicles fit best to transport cargo from outskirts to main routes. They are often used only as a secondary way of transport.

Aircraft is supreme transport for passengers. Due to its high speed and quite high transport capacity its very convenient to use aircraft while transporting passengers among cities. Its disadvantage is low capacity while transporting industrial cargo. The only infrastructure it requires is at least two airports, no connections have to be built.

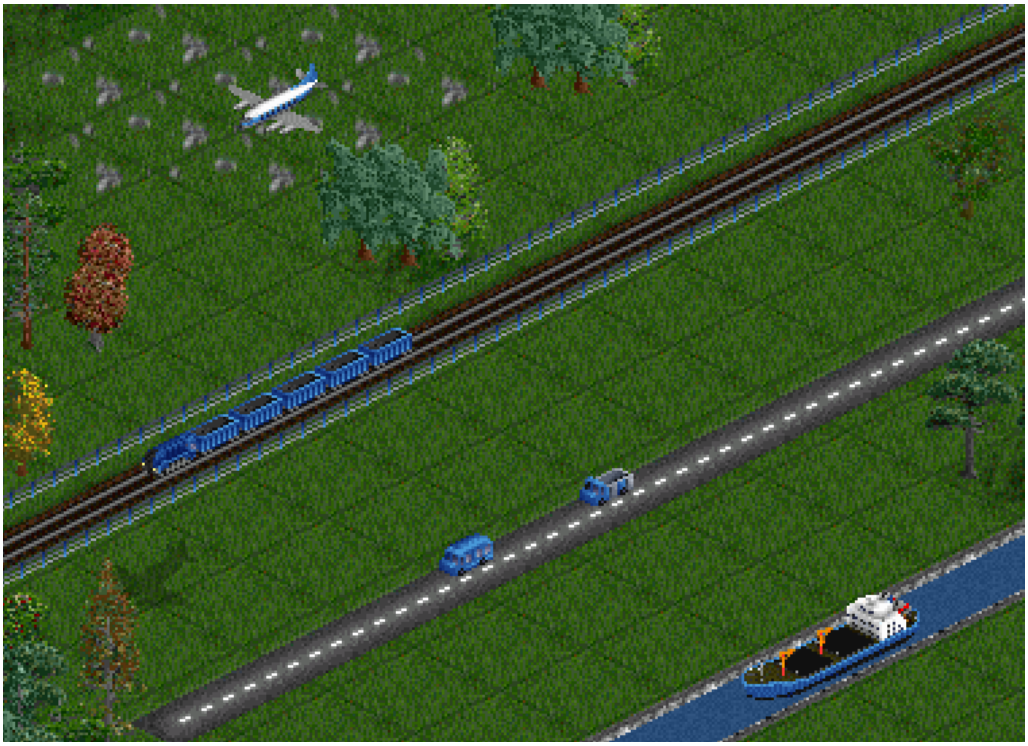

Figure 2.1: Four types of vehicles - airplane, train, road vehicles (bus and truck) and ship

Ships are the slowest and only fit to transfer cargo from the sea to the coastal parts of the map. They have large capacity for industrial cargo. Ships hardly find any real usage in OpenTTD and if they are built, mostly it is for aesthetic or to serve oil rigs. New feature in OpenTTD is river canals which can be built by player and can be used to travelling by boats in the midland. That increase possibilities of using ships but still they are the least used kind of transport.

Trains are versatile type of vehicles which fits short routes as much as long ones. Trains consist of an engine (possibly more than one) and wagons. In the basic version of the game, there are four types of rails available: Railroad, electrified railroad, monorail and maglev. Each type has its own engines and wagons and trains built for one type cannot travel on different kind of rails. It is very interesting kind of transportation with very in-depth set of mechanics. Player is required to build station for loading and station for unloading the cargo. In addition he needs to connect those stations with rails. It also probably provides the highest level of creativity for the player. The more wagons the train has, the heavier it is. Loaded wagons are even more heavy. The lower ratio ”horse power / weight of the train” is, the slower train accelerates. In order to run more trains on the same rails, building signals is required. Signals split rail into segments. If train encounters a signal, it checks if the next segment is clear of trains (no train or its part is located on that segment). If segment is clear, the train continues past the signal, in other case train stops and waits until the signal turns greed, meaning the next segment is free. Segments are displayed on image 2.2

Later OpenTTD introduced a track reservation system for trains with new pathfinding system Path Based Signaling (PBS). It allows more trains to enter the same segment under certain circumstances. When train comes to the PBS

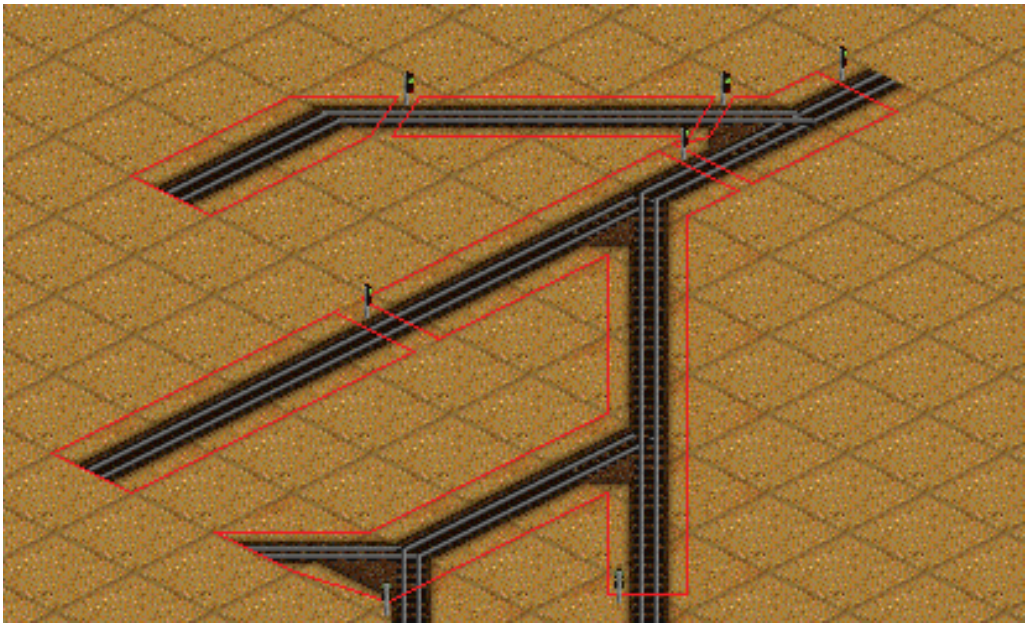

Figure 2.2: Segments of rails divided by signals.

signal, it will try to search path to next signal. If the train successfully reserves a path to next signal (there are no obstacles like other trains), it will reserve that path and while it is going on that path, no other train can reserve any part of the reserved track. This means that as long as new train can still find its way and reserve path to next signal, it can enter rail segment even if there are other trains on it and there will be no collision. This greatly enhances possibilities to build some crossroads or stations (The openttdcoop [2013]).

2.3 Industries and cargoes

Industry is basic unit that produces or accepts particular type of cargo. Industries are placed on the map with restriction that only one industry of each type can be assigned to one city (by default). Cargo is a basic unit that is produced in industries (that is called industrial cargo) or in cities (non-industrial cargo).

2.3.1 Industrial cargoes

Industry and its cargoes are divided into three categories - primary, secondary and tertiary. Primary industry accepts nothing and produces one or more type of cargo. This cargo is then transferred to secondary industry. Secondary industry accept cargo from primary and either produce nothing or produce different type of cargo when the primary cargo is delivered. The last type of industry is tertiary industry, which accepts secondary cargo and produces nothing.

The amount of money the company receives when the cargo is delivered depends on a couple of factors. The bigger distance between source industry/city to target industry/city, the higher reward. On the other hand, the longer it takes for the cargo to be delivered on that distance, the lower reward. In addition, each cargo has its transfer reward rate. That means that each cargo will earn

different amount of money if delivered to the same distance in the same time - it depends of its ”rarity”. Usually primary industry has lower transfer reward rate, secondary has average and tertiary has the highest. But the drop in the rate when the cargo travels more days is not that big by the primary industry then the tertiary. To sum the rules - primary cargo has the lowest reward when delivered very quickly and if delivered late, has still quite the same reward. Tertiary cargo earns the highest income when delivered really quickly, but looses a big part of the reward when delivered slowly. Secondary cargo is somewhere between primary and tertiary while considering income.

Last important thing is that distance traveled is measured in tiles between the industries, it is not considering the actual path that the cargo had traveled. That means if player will build more efficient route, he will earn more money and artificially extending the path does not earn more. Types of industrial cargoes differs depending on the map style (environment) the game is taking place on.

Each single piece of industry has its default production amount. Each game month the production has a chance to increase or decrease for some percents. If the company is serving the industry and transferring the cargo away, the chance to increase the production is higher then chance to decrease. It works the other way around as well. When production reach minimal amount, there is a chance that the production will get closed and disappear from the map. To balance this, sometimes new industries are built randomly throughout the game.

Some environments have unique mechanics, for example in desert map, the most valuable resource is wood. Player earns far more money by transporting it to any other cargo, but at a cost that sawmills producing wood chop down nearby wood in the world so you need to manually replace the trees in order for sawmills to produce wood.

2.3.2 Non-industrial cargoes

Non-industrial cargoes are the same on all four environments - passengers and mail. Both of the cargoes are produced by structures in cities and they are delivered to another cities (or to different parts of the same city). Rules regarding money rewards for the non-industrial cargoes work the same as for the industrial ones. The bigger the city, the more passengers and mail it can produce and the more city building are around the station, the more the production is (but each town has limits depending on its size).

2.4 Technology

As the original game Transport Tycoon was written in Assembly language, with the development of OpenTTD authors have chosen a different technology - C language. The whole game has been reverse engineered into the C language (openttd wiki contributors [2018]). Up to release 0.7.0 (excluded) AI could only be written directly in the C code of the game. Since 0.7.0 release, an API called NoAI API has been added to the game and AI scripts could have been written in language Squirrel, which started the boom of community-made AIs (openttd wiki contributors [2015]).

2.4.1 NoAI API

NoAI API is a long list of classes and functions callable from the scripting language. It gives access to various game data and methods to read information from the game. Finally it provides methods to interact with the world and to change its state. Methods are divided into different categories and the whole API is well understandable and easy to use. The main advantage of the API however is to split compilation of the game from the compilation of the AI scripts. Before the API had been added to the game, the whole source code needed to be compiled in order to see the changes.

Important classes of the API are AIAirport, AIRail, AIRoad and AIMarine which contains all functions connected to that kind of transport. Other important classes are AIIndustry and AITile. They provide information about industries and the tiles on the map. That corresponds with the perception of the human player and allow AI to gather information about the environment.

To showcase usage of the API, lets consider a case that we have X and Y coordinate of a tile and we want to get the index of the tile in the world. We would then write:

```
1    AIMap.GetTileIndex(X, Y)
```

and we would receive a number which is index of that tile. [2]

A useful feature provided by the NoAI API is *AIList* class. It provides a data structure for requesting different types of lists from the game. If AI needs to request all industry on the map, it simply creates an instance of *AIIndustryList* and it contains all industries (their IDs). It has function *Begin()* which gets the first item of the list, function *Next()* which moves the active pointer to next element and gets it, and method *IsEnd()* which returns true if the active pointer is at the end of the list. But what is really useful is function *Valuate(func)*. The *func* parameter is a function which is supplied to the Valuate function. *Valuate()* will then temporarily replace all the values with result of the supplied function, while providing the original ID of the item as first parameter to the supplied function. It looks like the Valuate function would call *func(item)* and then put the result to the original list. *Valuate()* can even take more parameters and they are automatically provided to the *func()*. After *Valuate()* finishes, one can use one of couple of methods for filtering values, for example KeepValue(), *KeepAboveValue()*, *KeepBelowValue()* and others to filter out the result. Values which were kept in the list are then converted back to IDs.

2.4.2 Scripting - Squirrel

OpenTTD uses scripting language Squirrel to script AI to the game. "Squirrel is a high level imperative, object-oriented programming language, designed to be a light-weight scripting language that fits in the size, memory bandwidth, and real-time requirements of applications like video games." (Demichelis, A. [2016]). Syntax is similar to C++ language and its internal functionality is similar to Java (it has its own virtual machine). Scripting in squirrel is quite easy, especially as the language is dynamically typed. The programming in Squirrel is based on

[2] Full NoAI API documentation can be found on link: `https://noai.openttd.org/api/`

using simple data types like integer and float and extending this model with arrays and tables. Even classes in Squirrel are represented as tables. Variables are all declared with a keyword **local**. [3]

2.5 History in AI in OpenTTD

AI has been part of the original Transport Tycoon game. The original AI was very simple - it was trying to build aircraft, trains or road vehicles, but with very low success rate. Very often AI company just bankrupted before the player could even notice it. If a player wanted to block building process of the AI agent, it was enough to build one rail where the AI was building and it got stuck for a long time not knowing what to do. The original AI was simply no challenge.

Later when the NoAI was released, new AIs started to pop up. With the API and Squirrel scripting, the scripts could be compiled on their own. That opened the AI scripting to community and anyone with basic programming knowledge could contribute. It was way easier to develop and script an AI agent to the game so the level of AI started to rise. Through the years, many new AIs have been developed and player can download them through the content manager. Some are specialized for one kind of transportation, others try to be versatile. There is definitely enough possibilities. One common feature of all AIs is that none is too complex - all are just trying to build connection one by one, not any kind of networks or just tiny ones. Exception is one particular AI called ChooChoo, which builds some crossings and reuse same paths for more trains.

[3] Squirrel documentation can be found on link: `http://squirrel-lang.org/squirreldoc/`

3. TrainsferAI

The newly implemented AI will be called TrainsferAI. At the beginning existing AIs will be examined to get an idea what they are capable of. After that the inspiration for the new AI will be introduced and differences between that and existing AIs will be presented. Finally the new AI will be described in details.

3.1 Existing AIs

There is plenty of existing AIs so far downloadable and usable within OpenTTD. It would be useless to implement another of these if it would not differ from all others. To see what aspects can be done differently, we need to check the best of the existing AIs and test what can they do and where are their limits.

3.1.1 trAIns

The main focus of trAIns AI is to bring AI which builds and manages rail networks. It prefers to build rather longer routes and sometimes if a part of the rail can be shared, it will build a junction and share the rail. trAIns use double railway parts for its network so it means for almost all tiles of the rail both directions are next to each other. For a planning algorithms trAIns use standard A* algorithm. It operates not with the single rail track pieces, but with pairs of the rail instead. Occasionally it builds bridges if an obstacle needs to be traversed. The trAIns AI builds so called RoRo stations (Roll-on-Roll-of), where trains enter the station from one side and leave on the other side (Rios, L. and Chaimowicz, L. [2009]).

3.1.2 ChooChoo

ChooChoo is another train-specialized AI. It transports mainly passengers and mail. It uses four way crossings and extends the network in all four directions to nearby towns. When the town is not in the line with the network, new crossing is built and the town is connected through it. If the network segment cannot be extended further, then it picks a random location on the map and begins a new network segment. The crossings are rather inefficient because the AI does not use bridges nor tunnels. That means that one of the trains needs to wait if both of them need to cross the crossroad at the same time. Other than that the AI creates nice networks Michiel [2009].

3.1.3 AdmiralAI

AdmiralAI is slightly different as it does not specialize in trains. Instead one can select which kinds of transport does it use. If it is set to trains only, it creates a lot of tracks which connects to each other sometimes. The network is not really optimized and it is rather randomly connected, but quite aesthetically looking. It builds a lot of trains and creates decent revenue from them. Difference between AdmiralAI and ChooChoo is that network does not look like a grid, but looks more natural with the Admiral.

3.2 Inspiration

All three selected AIs could build trains pretty well. The company could quite reliably grow without bankrupting, they were able to increase train amount over the course of game. To see what can be done better and how can those AIs be extended and improved, we can get inspired by group of players of this game called OpenTTDCoop. It groups a lot of players which are experienced with the game and they arrange public servers for community to cooperate with them. They use trains only and they try to build up effective rail network with loads of trains on it. They sometimes build creatively, sometimes more efficiently, but the outcomes are incredible. An example of what this group is capable of building is shown on image 3.1. The main difference between already existing AIs and the building done by OpenTTDCoop is one crucial element of the gameplay - planning. Existing AIs will not plan the network in the bigger scale, they will decide to add new train line and only in scope of this new line the AI will plan where to connect it. On the other hand, at the beginning of the game OpenTTDCoop players will plan their network in advance and then realize its parts separately, but the network in the end is compact. It is the planning which brings human-like features to the network design.

Planning of the network comes in different layers. First layer of planning is to choosing what trains will be used in the game (train length and engine type). The second, probably the most important planning step, is to plan where the crossroads will be placed on the map. Many factors need to be considered for the best outcome - for example position of the crossroads, distribution of the train traffic among the train lines or terrain difficulties. The last step is to plan where the connections between the crossroads will be placed and which crossroads will be connected to which. After the network is roughly planned, crossroads and connections start to be built and stations are being connected to the network as well.

In TrainsferAI a lot of algorithms and their variations were used for planning of the network. Each of them will be introduced in further sections.

3.3 Bounded rationality

Bounded rationality is a term commonly used in economics. It is an opposite of perfect rationality - if an agent behaves under perfectly rationality, he is rational, only self-interested and choosing optimal choices. While agent is behaving under bounded rationality, reflects the real human beings. The cognitive capacities of humans cannot be fully rational because of some limits - lack of information, limited time for making a decision or limits of human brain for example. The result is that the agent will pick a decision which satisfies his needs with the available information, he does not look for the optimal one. This helps to save time and processing capacities. The difference between bounded rationality and irrationality is that the agent is still trying to maximize its profit and be as rational as possible in given situation (tutor2u [2018]).

TrainsferAI and all actions made by it are limited by bounded rationality. The main reason is that the game itself provides the AI only a part of the assigned processor time. That causes the AI to be quite limited in processing power. As

Figure 3.1: High-end crossroad by the OpenTTDCoop players created on their public server game nr. 332. Games can be found on `http://www.openttdcoop.org/files/publicserver_archive/`

searching specially consumes a lot of processing power, all pathfinding runs several times slower that in a stand-alone program. OpenTTD does not allow any third party program to interact with the AI, so implementing the pathfinding process externally is out of the table. Second limiting factor is terrain in OpenTTD. The height of each corner is dependant of heights of adjacent corners. One cannot just modify the terrain to its needs (it is possible but not guaranteed to success). Modifying the terrain is a complex tasks that is quite easy for a human player. He sees the map as a compact thing and he can easily see that modifying the terrain somewhere else allows him to modify the part he wanted originally. But for an AI agent this is hard to implement and it is out of the scope of this thesis.

Because of those limitations, railroads will not reach levels of complexity that could be seen in OpenTTDCoop. They build the networks for tens of hours, perfecting every single tile of the network to perfection. The main goal of TrainsferAI is not to implement huge networks, but to demonstrate that AI can even do such a task.

3.4 Railway network structure

Railway network is a rail structure which allow efficient train traffic. Good railway network has as little redundant rail connections as possible and allow trains to go quickly. It should not contain paths which are far longer then the distance between two points the path is connecting, neither trains traveling through it should be forced to wait too long. Constructing network is not a simple task as many factors are affecting the outcome. TrainsferAI's network building is inspired by the OpenTTDCoop networks. They proved that their concept of network building

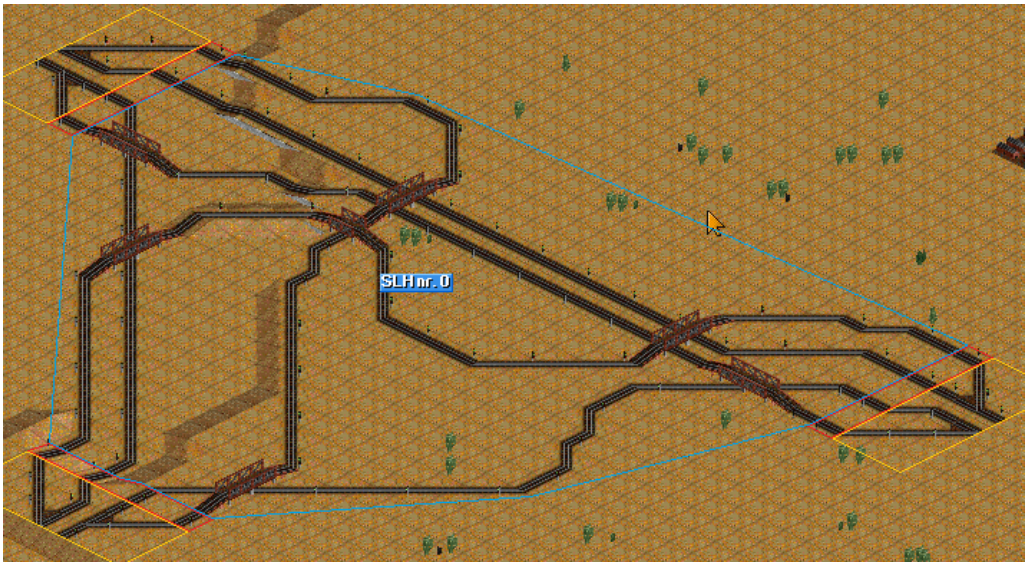

Figure 3.2: Crossroad scheme. Yellow parts are the entrance parts, red parts are entrance points and the blue part is the inner crossroad made from inner connections.

can handle huge amount of traffic if done precisely.

The basic structure of the network uses two elements - crossroads and connections. Crossroad is a basic node of the network which can be connected to other crossroads or can have free connection to the around world which some side-lines can connect to. Crossroad has entrances, which are points when the crossroad itself ends and the rails start to form a connection and from there are connected to another point of the network. Each crossroad has number starting from 1 and incremented for each crossroad. Internally the crossroad is made by connections, which connects the crossroad entrances from inside the crossroad - shown on figure 3.2. Crossroads can be three way or four way ones. Three way crossroads have three entrances and four way ones have four. A big part of all crossroads are three way because when they are expanded, it is easier to fit all connections in them onto the given space. Connections are rail routes connecting entrances of crossroads with another crossroads. Those connections are called main lines. They usually contain the highest traffic and they are built according to it. Depending on the number of trains, some can reach even 6 rails in one direction (12 total). Another type of connections are side-lines, which usually runs from primary industry station to the free entrance of a crossroad, effectively connecting the industry to the network.

Networks have a set of rules which builders follow. Those rules make the whole process easier and help to keep the traffic as fluent as possible. The basic rule is that a new industry is connected to the network only via crossroads. Connections between crossroads can never connect industries. They are designed in a way that there should be no slowdowns. Side-lines which connect the industries with the crossroad merge if close to each other to create bigger side-lines. Then the result big side-line is connected to the free crossroad entrance.

Distance between turns	km/h
0 (90°turn)	61
1 (2x45°turn)	88
2	111
3	132
4	151
5	168
6	1835
7	196
8	207
9	216
10	223
11	228
12+	231

Source: `https://wiki.openttd.org/Game_mechanics`

Table 3.1: Table of distances between the two turns and its maximal speed limit.

3.4.1 Turn length

Turn length is an important feature when designing all rails if the efficiency and fluency is the goal. If train lies on more then one turn to the same side, that train will receive maximum speed limit. If it travels faster, its speed will be reduced to the limit value. If the train goes slower, its speed remains unchanged, but the train can never accelerate to higher speed then the limit, while still located on two turns at the same time. The closer the turns are to each other, the lower the speed limit is. The exact number are shown in table 3.1. The table works for the basic type of rail, with monorail the speed limit is multiplied by 1.5, for maglev the multiplier is 2.

3.4.2 Crossroad types

There are total of three types of crossroad in the network - Sideline hub (SLH), Mainline hub (MLH) and Backbone hub (BBH). The terminology is taken from the OpenTTDCoop. Each crossroad has different purpose in the network and all types are important. [1]

SLH is the basic crossroad which connects primary industries to the network. It consists of one entrance used for connecting primary industries and two or more entrances which connect the crossroad with other crossroads. Number of SLHs in the network depends on player choice and on distribution of the primary industry.

MLH is the second crossroad type. It is similar to SLH but instead of one connection leading to primary industries, it has one connection leading to one secondary/tertiary industry. The number of MLHs in the network then corresponds with the number of secondary/primary industries which are being used among the network.

[1] More information about crossroad types and its features can be found at `http://blog.openttdcoop.org/2010/07/10/advanced-building-revue-06-hubs/`

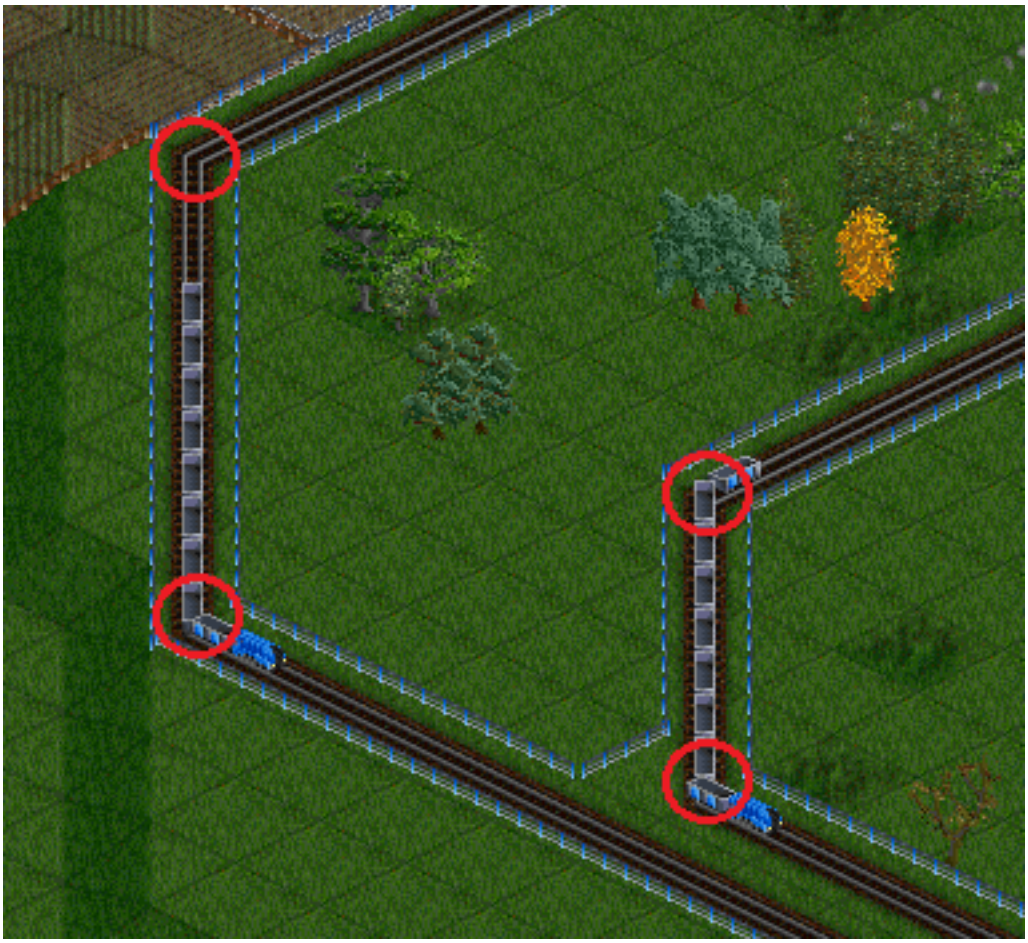

Figure 3.3: Train on the left is located only on one turn to the left, it will not slow down. Train on the right is located on both turns to the left therefore it will slow down.

BBH is the last type of crossroad. It connect more then two main lines together. It is usually built in a strategic places between other crossroads to ensure that there is a short connection between them. It is possible to create a network with no BBH if the player insists of it, or the network can be full of them. It all depends on design of the network and distribution of the crossroads. Good layout usually has not much of them, but having couple does no harm.

3.5 Data representation

Majority of the data required for the run of the AI agent is gathered directly from the game data through the API. This method is quick, ensures that the data received is up-to-date in the time of the request and does not require any saving/loading of the data manually in the AI, so it is cheap. In this category belongs several kinds of tasks - requesting information about certain tiles of the map, size of the map, requesting information about industry (for example industry location from its ID), checking if tile can be built on or getting information about terrain, those requests are all directed to the API.

The only information which the AI needs to store manually are information about its previous decisions/calculations. Because TrainsferAI plans the network, it needs to store the network model. It also stores some additional information about industry which needs to be stored for further planning.

3.5.1 Industry model

Industry model is a class storing information about industry. Industry has two flags which can be assigned to them by AI - blacklisted and used. Those flags are internal values for the TrainsferAI, they are not used anywhere in the game. Blacklisted industry is industry which failed to be included to the network. The reason may be that there was no valid path found from the industry to its target destination, that it is already served by another company, that there is no secondary industry on the map that accepts its cargo or maybe it is in the middle of the city and there is no room for station to be built. If industry is blacklisted, the AI will no more try to build station around it, transfer its cargo or connect it to the network. Used flag is set to industries which have already been served by the TrainsferAI (in a money making or regular route). The industry model contains two tables - *blacklistedIndustries* and *usedIndustries*, each for its respective flag. When an industry is about to be blacklisted/used, to the corresponding table a boolean value of ”true” is inserted with key being the industry id *industryId*:

```
1 function IndustryModel::BlacklistIndustry(industryId) {
2     blacklistedIndustries[industryId] <- true;
3 }
```

If AI checks whether the given industry is blacklisted/used, it simply checks if the table contains any value with the given key (*industryId*):

```
1 function IndustryModel::IsIndustryBlacklisted(industryId) {
```

```
2     return blacklistedIndustries.rawin(industryId);
3 }
```

3.5.2 Rail model

Rail model stores information about trains and rails. First information is the *engine* chosen for the game. For the maximum flow in the network, it is ideal when trains travel all with the same speed - no train has to slow down because it is faster then the train in front of it. When first creating train, engine is selected from all available engines by its maximum speed (the fastest is chosen). Then the engine id is stored in the rail model. Money making routes are stored in the rail model as well. Each time there new money making route is constructed, it is stored in this array. It has also a table called *railRegistrations*, which stores information about on which tile there is a rail and what crossroad it leads to. The most important variable in rail model is array *crossroads*, which stores all crossroads currently created on the map.

Crossroad class has properties *type*, which is enum { SLH, MLH, BBH, VIRTUAL }, *number*, which is number of the crossroad, *index*, which is index of tile where the center of the crossroad is located, *assignedIndustries*, which is array storing IDs of all industries assigned to that crossroad, *networkConnections*, which is array of connections of the crossroad to other crossroads/to industry (class *NetworkConnection*), and boolean *isBuilt*, which is self-explanatory.

NetworkConnection is a class storing information about one particular connection. It has properties *source*, which is number of crossroad from which the connection starts, *target*, which is number of crossroad to which the connection leads, and then structures for storing inner and outer entrance points for entrance at the source crossroad and target crossroad. The structures are *inConnectionsFrom*, which is storing points going from the connection to the source crossroad, *outConnectionsFrom*, which is storing points going from the source crossroad to the connection, *inConnectionsTo*, which is storing points going from the connection to the target crossroad and *outConnectionsTo*, which is storing points going from the target crossroad to the connection. Those four are storing the inner connection points, on image 3.2 those are the red zones. *Connection* also needs to store the outer points, which are located in the yellow zones and are the two points furthest from the crossroad center. Those points are stored in variables *inOuterConnectionFrom* (point going from the connection to source crossroad), *outOuterConnectionFrom* (point going from the source crossroad to the connection), *inOuterConnectionTo* (point going from the connection to the target crossroad) and *outOuterConnectionTo* (point going from the target crossroad to the connection).

3.5.3 Building restrictions manager

Special feature of the TrainsferAI are building restrictions. It is a mechanic that allow the AI to reserve some tiles for building particular thing in the future and no other construction is allowed to happen on those tiles.

Building restrictions use string tags and principle of restricting is that some building tags can be active at given time and if the tile is restricted to no tags

Figure 3.4: Restricted area around the planned crossroad center. All tiles inside the red rectangle are restricted and only crossroad 1 can be built on them.

or to subset of tags which are currently active (in other words, if at least all tags the tile is restricted to are active at that time), the tile is available for building. Otherwise the tile is treated as non-buildable.

If the AI wants to restrict a tile, it needs to supply an array of tags with it. Then the tile is restricted with that supplied tags. The *BuildingRestrictionManager* has a variable *buildingTags*, which is array of currently active building tags. This array can be modified throughout the run of the AI depending on what is being built.

When for example crossroad 1 is being planned, all tiles around that crossroad center are restricted with tags *"C1"*. The tag *"C1"* is active only if crossroad 1 is being built. If anything else would want to build on any of those tiles, the restriction tag *"C1"* is never active so those tiles will never be used for building - this situation is displayed on image 3.4.

This feature is very convenient for plan-based building because it ensures earlier planned construction will have room to be built and the AI does not need to look for another spot if it accidentally built something on that tiles.

Before each pathfinding, This feature is used in the planner - if a crossroad is planned at a particular location, range around that tile is restricted.

3.6 Life cycle

Each AI contains a *Start()* function which is the main function of the AI. If the *Start()* function ends or the execution of the script is interrupted (for example because of an error), the AI will stop. That means that the already existing vehicles of the company will continue to run, but the AI will no longer be running,

so no new vehicles/infrastructure will be built.

At the beginning of the life cycle of TrainsferAI, data model instances are created and stored as global for later access (more about data models in 3.5). Then the TrainsferAI moves to its first phase - money making phase. After that network planning phase starts, followed by network building phase. The final phase of the AI is industry connecting phase in which the AI remains until the end of the game. All of the tasks proposed in each phase will be further examined and explained in section 3.7.

3.6.1 Money making phase

Purpose of the money making phase is to provide the company starting budget to build the network and make some normal trains. Money making route is route straight from the selected primary industry to the selected secondary industry. It has its own rail tracks and straight connection between the stations. Number of money making routes can be set in the AI settings. When the right amount of routes is built, the AI moves to the network planning phase.

3.6.2 Network planning phase

Network planning phase is second phase of the AI life cycle. In this phase, nothing is being built, everything AI does is plan. That makes room for the money making routes to earn some money and prepare for the building phase. First step in the network planning is finding locations for future SLHs. When those points are found, then the AI will find MLH spots on the map. The last step of planning the network is connecting the crossroads. When the connections between crossroads are set, AI moves to the building phase.

3.6.3 Network building phase

In this phase the AI takes the network model and builds it. It starts with crossroads. For each crossroad entrance points are found. When the entrance points are set, the crossroad is getting built. After building the inner connections between the entrances, the entrances themselves are constructed. When all crossroads are built, connections between each crossroad is build. When the whole network is finished and only connections to industries are left not built, network building phase is finished and AI moves to the final phase.

3.6.4 Connecting phase

In final phase of the AI the agent picks one primary industry and secondary industry where the cargo needs to be delivered. It builds connection to SLH entrance and make station for loading and unloading. When the infrastructure is complete, it builds some trains and start running them. This sequence repeats as long as there are free entrances to the SLHs for industry connecting.

3.7 Algorithms

There was a lot of algorithms used in TrainsferAI to solve particular sub-tasks of the problem. We will look at each algorithm in details.

3.7.1 A*

A* (AStar) algorithm is widely used path-finding algorithm as it is very efficient and fast. It is an extension of Dijkstra's algorithm - it introduces a heuristic function into the search. It is calculated as

$$f(n) = g(n) + h(n),$$

where:

- f(n) is total estimated cost of path through node n
- g(n) is real cost to reach node n
- h(n) is estimated cost from node n to goal (Brilliang.org [2016])

A* is a standard pathfinding algorithm in OpenTTD and there is even a library containing the algorithm for free use in AI development. For purpose of this thesis the library A* was not enough. The main problem were the turns - if a node would be expanded and the turn to left would not be possible because it would be to close to previous turn to the left, the node would not be expanded. But after that we could enter the node from different side and the turn would be fine - in that case we could expand the node. But with normal A* the node would already be open so it would not be possible to open it again. It was necessary to modify the algorithm to achieve this functionality.

The main problem encountered in the thesis is called multi-agent pathfinding. It is basically a problem of finding multiple paths in a common space which does not cross each other. The basic version of this problem operates in three dimensions - x, y and time dimension, where x and y being the standard 2D dimensions. In the case of the thesis, it only needs to operate in 2D (x and y). In the original problem the paths are actually a real-time positions of the agents, so the conflict of two standing on the same tile can only happen in one timestamp. If an agent steps on tile where another agent has been in the past, it does not matter and this situation is not a conflict. That is not the case here - the path is considered whole - from start point to the end point. The path cannot go through a tile where another path has been in the past because there will be rails of that path. So the time dimension is discarded from this problem.

A basic solver for this problem is a path-searching algorithm which is ran multiple times in a row. This is very time-consuming to find optimal solution because the search space is enormous and scales exponentially by the number of agents. If we satisfy with sub-optimal solution however, this problem is solvable in reasonable time as each search only needs to find a path, not the optimal one.

Alternative approach to problem of finding multiple paths in common space is called Conflict-based search. It defines a conflict as two agents being at the same place at the same time. It operates at two levels - at the higher level the solver searches in a tree of conflicts between two agents. At the low level, it takes one

agent and performs a search, but it gives the search a constraint that the search cannot go through the node of the conflict at the time of the conflict. That can raise new conflict which are then resolved in the same way Sharon, G., Stern, R., Felner, A., & Sturtevant, N. [2012]. To adapt this algorithm to the problem in the thesis, we need to discard the time dimension from the search. That increases the number of conflicts a lot because the "locations" of the agents are not a single tile, but the whole path. The conflict-based search is optimized for as low number of conflicts as possible, not fitting for an open-space path finding problems, but more likely for corridors and narrow spaces. The time complexity of the algorithm increases for each encountered problem.

For the purpose of the A*, priority queue has been implemented to store open nodes by the priority. The basic data structure Node contain *tileIndex*, which is index of the tile the node is located on, *edge*, which is the side of the tile which the node is on (numbered 0 to 3 and starting top-right 0 and ending bottom-right 3 anti-clockwise), *parentNode*, which is reference to the parent node, *cost*, which is total cost of the path up to the node, *length*, which is length of the node (diagonal has 0,5, straight has 1 - it is used for distance between signals), *lastTurnLeft*, which is distance to last turn to the left and *lastTurnRight* analogically.

The first step of the A* is preparation. The A* needs to find path coming from a concrete direction so the turns are built correctly. For this, we find a set of tiles called "area near goal". It is a list of tiles which are close to the goal and which allow the path to come from the right direction, as shown on image 3.5. The goal tile there is the rail and the path has to come from the top-right direction. The blue shows correct area where there is possible to find the connection. The red area is forbidden as from there path cannot be found to the goal tile from the specified direction. Variable *areaNearGoal* stores this area (the blue part) and each time node close to the goal is opened, it is checked if it belongs among the allowed tiles. If not (that means it is on the wrong side), the tile will not be expanded. Pseudocode is showed in algorithm 3.1.

The main cycle of the algorithm (3.2) then checks if *maxSteps* not reached, in that case it returns no path. it pops the top node from the queue and opens all its neighbours. For each of them, it checks if the neighbour is goal - it has found the path in that case, otherwise it checks if the neighbour is available. If true, it calculates its priority and pushes it to the queue (3.3). After all neighbours have been checked, it puts the node to closed nodes.

```
1   goalTile = to.tileIndex;
2   goalEdge = to.edge;
3   fromTile = from;
4
5   //set area near goal
6   areaNearGoal = GetGoalNearbyArea();
7
8   queue.Push(0, from);
9   lastNode = null;
10
11  local step = 0;
```

Algorithm 3.1: A* preparation

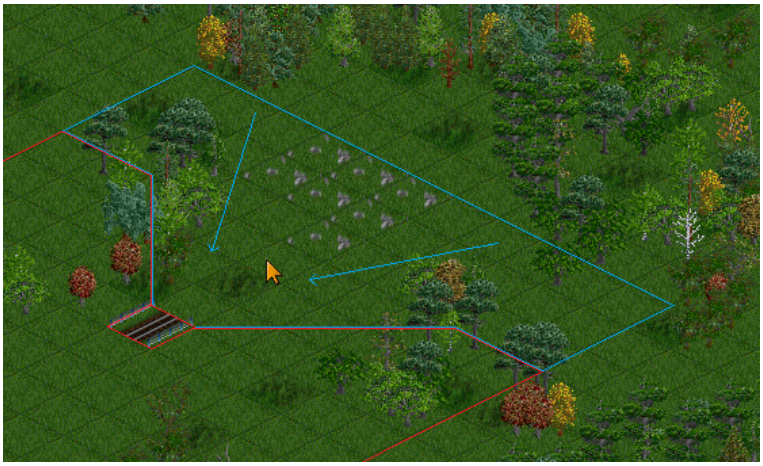

Figure 3.5: The area around goal - the blue part is the correct area and the red area is the forbidden one

```
function AStar::FindPath(from, to) {
  PrepareAstar();

  while(!queue.IsEmpty()) {
    if(maxSteps > 0 && step >= maxSteps) return null;

    prio = queue.GetMin();
    current = queue.Pop();

    //add all neighbours to the queue and put the current
   node to closed
    neighbours = GetNeighbours(current);
    for(i = 0; i < neighbours.len(); i++) {
      currentNeighbour = neighbours[i];
      CheckNeighbour(currentNeighbour);
    }

    //if the goal has been found
    if (lastNode) {
      break;
    }

    //put node to closed
    if(isDetailed) {
      closed[current.GetDetailedIdentifier()] = current;
    } else {
      closed[current.GetIdentifier()] = current;
    }

    step++;
  }

  //reconstruct the path
```

```
33
34    return ReconstructPath();
35  }
```

Algorithm 3.2: A*

```
1  //check goal reached
2  isGoal = isToGetClose ? IsNearGoal(currentNeighbour) :
       IsGoal(currentNeighbour);
3
4  if (isGoal) {
5    lastNode = currentNeighbour;
6    break;
7  }
8
9  //if neighbour is not an available tile for building, skip
       it
10 if (!IsAvailable(current, currentNeighbour)) {
11   continue;
12 }
13
14 priority = GetPriority(current, currentNeighbour);
15
16 queue.Push(priority, currentNeighbour);
17
18 if(isDetailed) {
19   open[currentNeighbour.GetDetailedIdentifier()] =
       currentNeighbour;
20 } else {
21   open[currentNeighbour.GetIdentifier()] = currentNeighbour;
22 }
```

Algorithm 3.3: A* CheckNeighbour function

If the path has been found, it needs to be reconstructed. The algorithm stores parent node for each node. For reconstruction, it will simply take the last node and reconstruct the path by traversing through the parent nodes until there is no parent node (and we found the starting node). The path will be reversed before building (so it looks better that the path is building from the beginning to the end, not the other way around (3.4).

```
1  //if no last node, path has not been found
2  if(lastNode == null) return null;
3
4  //push the last tile
5  path = [];
6  n = lastNode;
7
8  while(n != null) {
9    path.push(n);
10   n = n.parentNode;
```

Figure 3.6: The upper rails were the first path. If that rail would go straight, the lower rails would fail to build as there is a house in the way. The pathfinder was able to avoid it.

```
11 }
12
13 paths = [];
14 paths.push(path);
15
16 return paths;
```

Algorithm 3.4: A* ReconstructPath function

A* has a possibility to find a double way. That means that for each node it will check node to the left of it and checks if that node is available as well as shown on image 3.6. That ensures that along the found path can be built another path, effectively making two paths instead of one. For that reason, at the end of the algorithm the resulting path is returned as array (in case of double way search, it returns two paths in the array, otherwise only one). The double way search is not shown in the code.

Open and closed lists are implemented as tables. If a node is put into the table, it checks if A* is set to detailed mode or not. If not in detailed mode, the identifier of the node which is put into closed/open list will be just his index and edge. If the detailed mode is on, the identifier will contain information about how far back there was turn to left and turn to right. The node can then be opened more times if the distance to last turn to left or turn to right differs from the previous instances of the same node. That is convenient when finding path with correct turns on smaller area so there are less tiles that can be expanded, but the algorithm needs to find a path there anyway. The best example of the difference between detailed and not detailed modes is finding connection between crossroads and finding connection inside the crossroad. While the first case usually looks like a long lane leading to another crossroad and the algorithm usually does not try to open one node more times, in second case the algorithm will repeatedly open some nodes until potentially correct turn will appear and path will be found.

The distance from the last turn to left and turn to right can be big numbers. We do not need to store the full number because if we hit the number at which the path can be turned to the right, we do not need to count more. We can then limit

the number to the length of the trains. That will lower the possible combinations of one node that can be opened significantly.

Due to space searches being time-consuming and AI having only limited processor time, some A* performance improvements needed to be done in order for the A* to speed up. First of all, The detailed mode was enabled only while building crossroads. The connection usually find other way and go forward almost all the time so the reopening of the node would not benefit them. The other problem that occurred when the A* was finding connection in detailed mode was that if it reached a dead end, it would usually never get from that point.

3.7.2 A* heuristic function

Heuristic function is very important for a successful A* searching. Heuristic function in TrainsferAI is enhanced with additional factors (algorithm 3.5). If the search is not detailed (for example connections), it has higher heuristic multiplier of 3. That means the remaining estimated cost is multiplied by 3 - the estimated cost has far higher value then the already travelled cost. That means A* behaves more "hungry", expanding more likely nodes which are very closer to the goal even if they have higher cost themselves. This helps to get from the dead ends faster. When the A* is detailed (finding crossroads), it has a multiplier of 1.2. This enables to pass obstacle and get to a point from more different angles more easily as nodes which are further from the goal but has low cost are more likely to be expanded. This leads to a fact that this way A* can find bridging over another part of the crossroads much easier.

```
function AStar::GetPriority(nodeFrom, nodeTo) {
    cost = nodeTo.cost;
    prevCost = nodeFrom.cost;
    diffCost = cost - prevCost;

    distance = GetEuclideanDistance(goalTile, nodeTo.
   tileIndex);

    //estimated cost to the goal node
    if(isDetailed) {
        multiplier = detailedHeuristicMultiplier;
    } else {
        multiplier = heuristicMultiplier;
    }

    estimated = (DistanceManhattan(nodeTo.tileIndex,
   goalTile)*1.8 + DistanceMax(nodeTo.tileIndex, goalTile)
   *0.2) / 2;

    //penalty for beeing a bridge
    bridgePenalty = nodeTo.type == NodeType.BRIDGE ?
   BRIDGE_PENALTY : 0;

    //penalty for requiring a terraforming
```

```
21      terraformingPenalty = IsTileSloppedCorrectly(nodeFrom,
    nodeTo) ? 0 : TERRAFORMING_PENALTY;
22
23      closeToGoalMultiplier = 1;
24      if(isDetailed && distance < CLOSE_TO_GOAL) {
25          closeToGoalMultiplier = 0.6;
26      }
27
28      accessSideMultiplier = GetAccessSideMultiplier(nodeTo.
    tileIndex, goalTile, goalEdge)
29
30      return (prevCost + bridgePenalty + terraformingPenalty)
    + (estimated + diffCost) * multiplier *
    closeToGoalMultiplier * accessSideMultiplier;
31  }
```

Algorithm 3.5: Heuristic function

In the heuristic function there are many of parts which evaluate certain features. All parts has their own reason to get counted into the final heuristic.

Estimated cost takes two parts - manhattan distance and max distance. Manhattan distance is the sum of difference of the X and Y coordinates of two points. This only has produced weird looking connections that would go diagonally to some point and then go straight which did not look right. So the max distance has been added. Max distance is simply the greater from difference of the X and Y coordinates of the points. That caused to expand points which reduces both distances by 1 to be at first. That would get rid of the diagonal part of the rail and would make it more squared and fluent (not so many sharp turns). The coefficients were set that the max distance only has small influence compared to the manhattan distance, which works the best. The whole *extimated* cost needs to be roughly equal to the distance, so we need to divide the result by 2 (because the sum of the both distances are roughly 2*distance as well).

Bridge penalty gives a slight penalty to nodes which create a bridge. The penalty given is almost zero becuase the only real effect this factor gave to the search was longer searching time.

Terraforming penalty is a slight penalty given to nodes which required to modify the terrain in order to be built.

CloseToGoalMultiplier is a very important one. When the path gets closer to the goal, those closer nodes have priority over the further by multiplying its multiplier by 0.6. That way its easier for the search to get finished when getting closer to the goal, but it preserves its ability to get from dead ends (which is granted by the higher multiplier for the estimated part of the path).

AccessSideMultiplier is another way how to get to the goal faster. It penalizes nodes which are on the opposite side compared to the goal direction. Those nodes would need to go around the goal tile in order to come from the right direction. The further the node is from the goal tile while on the wrong side, the bigger is the multiplier. Its max value is size of the map / 50, so for map 512*512 the multiplier can get to value of 10.

The final heuristic value is calculated as cost of the node + penalties + (*estimated cost* + cost difference) * all multipliers.

3.7.3 Finding a station

When a rail station needs to be found, function *FindSimpleStationWithConnection()* is called (3.6). Prior to this, a pathfinding is ran and points close to the potential stations are found. Those points need to be connected to the station for a successful construction. First step is to get directions between the industry where the station will be built and the entry points of the connection. Then a direction of the station is picked to the station faces the connection points. Then up to four times the algorithm will try to find a station with that given direction. If it does not find viable station in a direction, it changes the direction and tries again. For each direction it will iterate over all points in an area around the industry (variable i corresponding to x coord and variable j corresponding to y coord). If the tile is not close to industry (that means the station will not be in the industry range and will not be provided by cargo or will not accept it), current tile is discarded. Then method *TryToFindStation()* is called which checks if the station can be built starting from the specified tile. It checks for correct terrain and if there are no obstacles for the station itself and for the station entrance as well. If there is a free room for the station, it then checks if a connection from the possible station entrance can be connected to the given points from the connection. If all station, its entrance and connection can be built, then pair of (station, entrance) is returned. Because of the bounded rationality, function can return null - in that case the industry is blacklisted as there is no room for a station to serve it and another industry is found.

```
1  function StationFinder::FindSimpleStationWithConnection(
       tileIndexFrom, tileIndexTo, cargo, loading, entryPoint,
       exitPoint) {
2
3      //at first, try all stations with directions leading
       towards the target points
4    directions = MapHelper.GetDirections(tileIndexFrom,
       entryPoint.tileIndex);
5    directionFrom = GetDirectionFrom(tileIndexFrom, entryPoint
       .tileIndex);
6    directionTo = GetDirectionTo(tileIndexFrom, entryPoint.
       tileIndex);
7
8    x = AIMap.GetTileX(tileIndexFrom);
9    y = AIMap.GetTileY(tileIndexFrom);
10   moveToOtherDirection = false;
11   direction = directionFrom;
12
13   for(dir = 0; dir < 4; dir++) {
14     for(i = x-MIN_DIFF; i < x+MAX_DIFF; i++) {
15       for(j = y-MIN_DIFF; j < y+MAX_DIFF; j++) {
16         //continue only if the tile accepts/produces the
       required cargo
17         if(!CheckTileIsCloseToIndustry(GetTileIndex(i, j),
       cargo)) continue;
18
```

```
19          //try to build station
20          station = TryToFindStation(i, j, direction,
    tileIndexFrom);
21          if(station != null) {
22            //now we want to find the path to the given points
    , if success, return the station, if not, continue
23            pf = PathFinder();
24            paths = pf.FindPath(entryPoint, station.
    entryPoints[0], true/*double path*/, 200/*max steps*/);
25            return [station, paths];
26          }
27
28          if(paths != null && paths.len() == 2) {
29            //lets return paths and the station
30            return [station, paths];
31          } else {
32            moveToOtherDirection = true;
33          }
34
35          //else we continue;
36          }
37
38          if(moveToOtherDirection) {
39            break;
40          }
41        }
42
43        if(moveToOtherDirection) {
44          break;
45        }
46      }
47
48      local newDir = (direction + 1) % 4;
49      direction = newDir;
50    }
51
52    return null;
53  }
```

Algorithm 3.6: Finding a station

3.7.4 Finding SLHs

First step of the planning of the network is to find spots for SLHs. At the beginning, list of available primary industries is gathered from the game. Available industry is an industry which is not served by any other transportation services. For the SLH spot searching, a k-means algorithm is used. The algorithm objective is to ”group similar data points together and discover underlying patterns. To achieve this objective, k-mean looks for a fixed number (k) of clusters in the dataset.” (Dr. Garbade, Michael J. [2018])

The dataset in this case is the list of available industries and the purpose of the algorithm is to divide the industries to clusters and then find the centers of each cluster. That set of center points would then create center points for the SLHs. Because the SLH distribution is not required to be precisely in center of the clusters, 3 iterations of the k-means algorithm are enough to get a rough distribution of the points.

One more step is done after the k-means is finished. A square around the center is made and another eight of same squares are created around the original square. Then, for each crossroad center point, for each square from the 9, a buildable score is calculated. The buildable score is number of tiles in the square which are buildable (has no obstacle on them) and at the same time are not a water tile. Then all 9 buildable scores are compared and center of the square with the highest buildable score is selected as center of the SLH. This helps to preserve building place for the crossroad around its center and prefers the "clearest" area.

3.7.5 Finding MLHs

Second step after SLH spots are found is to find MLHs. The AI iterates over all cargo types of the game. For each cargo type it gets a list of industries that accept the cargo. For each industry function *GetNewMLH()* is called which tries to find MLH spot for the industry accepting the selected cargo type (3.7). The function at first checks if the industry is not too close to any existing crossroads to prevent the network to be collapsed and too dense on some spots. If this condition is satisfied, it will try to find suitable spot for MLH around the selected industry - function FindCrossroadNearIndustry() (3.8) is called. If this function returns a crossroad, it is returned. Otherwise it continues to check other industries.

```
1  function CrossroadPlanner::GetNewMLH(crossroads,
       availableIndustries) {
2      newCrossroad = null;
3
4      //until we found crossroad or all industries has been
       searched through
5      chosen = availableIndustries.Begin();
6      while(!newCrossroad && !availableIndustries.IsEnd()) {
7          location = AIIndustry.GetLocation(chosen);
8          isTooClose = false;
9
10         //check all crossroads if the industry is not too
       close to one
11         for(i = 0; i < crossroads.len(); i++) {
12             if(EuclideanDistance(chosen, crossroads[i].index
       ) < INDUSTRY_TOO_CLOSE_TO_CROSSROAD) {
13                 isTooClose = true;
14             }
15         }
16
17         if(isTooClose) {
18             chosen = availableIndustries.Next();
```

```
19              continue;
20          }
21
22          //try to get a crossroad near this industry
23          possibleCrossroad = FindCrossroadNearIndustry(
    crossroads, chosen);
24
25          if(possibleCrossroad != null) {
26              newCrossroad = possibleCrossroad;
27              break;
28          }
29
30          chosen = availableIndustries.Next();
31      }
32
33      return newCrossroad;
34  }
```

Algorithm 3.7: Finding a MLH spot

Function FindCrossroadNearIndustry() tries to find MLH around a given industry. At first it creates a virtual square around the industry. Then it excludes all points which are too close to the industry itself from the square. Then two random numbers xRand and yRand are drawn and other two numbers controlling if the xRand and yRand numbers will be negative or positive. Then the xRand and yRand numbers are added to the industry location coordinates and a potential tile for crossroad is found. The point is then checked if it is not too close to any existing crossroads. If it is, the point is discarded. If that condition is satisfied, buildable score of the area around the point is calculated. If it is higher then the threshold, the point is automatically selected as new crossroad center and new crossroad is returned. If not, its score is remembered and next point is being drawn. After each 50 unsuccessful tries, the multiplier is increased by 0.4. This ensures that if the industry is failing to be built in a spot, the radius of the crossroad center is getting bigger, offering more possible spots for the crossroad to be successfully built. But for most cases, the crossroad is built somewhere near the original point.

```
1   function CrossroadPlanner::FindCrossroadNearIndustry(
    crossroads, industry) {
2       multiplier = 1, bestBuildableScore = -1, bestIndex = -1;
3
4       for(i = 0; i < ATTEMPTS_TO_FIND_CROSSROAD_NEAR_INDUSTRY;
        i++) {
5           if(i % 50 == 0) {
6               multiplier += 0.4;
7           }
8
9           xRand = AIBase.RandRange((MAX_DISTANCE_OF_CROSSROAD
    * multiplier) - INDUSTRY_TOO_CLOSE_TO_CROSSROAD) +
    INDUSTRY_TOO_CLOSE_TO_CROSSROAD;
10          yRand = AIBase.RandRange((MAX_DISTANCE_OF_CROSSROAD
```

```
* multiplier) - INDUSTRY_TOO_CLOSE_TO_CROSSROAD) +
INDUSTRY_TOO_CLOSE_TO_CROSSROAD;
11          isXNegative = AIBase.RandRange(2);
12          isYNegative = AIBase.RandRange(2);
13
14          if (isXNegative == 0) { xRand = -xRand; }
15          if (isYNegative == 0) { yRand = -yRand; }
16
17          //x and y coord of the potentional crossroad tile
18          x = AIMap.GetTileX(AIIndustry.GetLocation(industry))
 + xRand;
19          y = AIMap.GetTileY(AIIndustry.GetLocation(industry))
 + yRand;
20
21          //protecting the overflow and underflow from the map
22          if (x < 0 || y < 0 || x >= AIMap.GetMapSizeX() || y
>= AIMap.GetMapSizeY()) {
23              continue;
24          }
25
26          isTooClose = false;
27          for(j = 0; j < crossroads.len(); j++) {
28              crossroad = crossroads[j];
29              if(abs(x - crossroad.GetX()) <
CROSSROADS_TOO_CLOSE &&
30                  abs(y - crossroad.GetY()) <
CROSSROADS_TOO_CLOSE) {
31                  isTooClose = true;
32              }
33          }
34          if(isTooClose) {
35              continue;
36          }
37
38          buildableScore = GetCrossroadBuildableScore(
crossroads, x, y);
39          if(buildableScore > bestBuildableScore) {
40              bestBuildableScore = buildableScore;
41              bestIndex = AIMap.GetTileIndex(x, y);
42          }
43
44          //check if a good MLH was not already found, if it
was, return it right now
45          if(bestBuildableScore >= MIN_BUILDABLE_SCORE) {
46              return Crossroad(CrossroadType.MLH, crossroads.
len(), bestIndex, [industry]);
47          }
48      }
49
50      if(bestIndex == -1) {
51          return null;
```

```
52      }
53
54      return Crossroad(CrossroadType.MLH, crossroads.len(),
    bestIndex, [industry]);
55  }
```

Algorithm 3.8: Finding a MLH near selected industry

3.7.6 Connecting the network

After SLH and MLH locations are found, the last piece to finish main part of the network is to connect crossroads together. Function *ConnectNetwork()* does that task. It iterates only through all MLH crossroads (because SLH crossroads does not require any other crossroad to be connected to them to unload cargo, SLH crossroads contain only loading stations). All other steps apply to each MLH. The algorithm calls method *GetConnectionPoints()* which will iterate over all crossroads and check which crossroads need to be connected to this MLH. Then it calculates distances from the MLH to each of the crossroads about to be connected. If the algorithm has found crossroads and its distances, for each one of them it calls *FindAndConnect()* method. This method will find path in the partially-connected network. It can create new one, but the already existing parts of the network have its cost multiplied by 0.4 so it prefers to reuse existing connections if possible. A* is used as a search algorithm for the path lookup. After this step, there is a valid path between each crossroad which needed to be connected and current MLH (path is created as a virtual connection, nothing is planned on the map or built yet. The only information the connection contains so far is "Crossroad number 1 will be connected to crossroad number 3" and so on.) It can happen that a crossroad will have more then 3 connections. In that case the crossroad building would be too complicated. We need to limit the number of connections each crossroad has. Method UniteConnections() takes care of that (detailed in 3.10). After that all crossroads are in valid state - they are connected to all crossroads they need to be and they have at least 2 and maximum of 3 connections (SLHs and MLHs have one connection less as it is reserved for connecting of the industry).

Finally the algorithm will once more iterate over all crossroads, selecting only SLHs and MLHs this time. For each, it will generate a random point on the map and then creates a virtual crossroad there. Virtual crossroad is not an existing crossroad and its only purpose is to create entrance pointing to it. The direction is random and industries connected to that crossroad use the entrance created with this virtual crossroad.

After all crossroads are iterated over, the array with crossroads is returned.

```
1
2   function ConnectionPlanner::ConnectNetwork(crossroads) {
3       //create list of all cargos which is not primary cargo (
    does not come out from SLHs)
4       secondaryCargoList = ConnectionPlanner.
    GetSecondaryCargos(crossroads);
5
```

```
 for (i = 0; i < crossroads.len(); i++) {
     //we want to connect the points to the MLH so we consider only MLHs here
     if(crossroads[i].type != CrossroadType.MLH) { continue; }
     currentMLH = crossroads[i];

     //only crossroads which we want to connect to the current MLH
     connectionPoints = GetConnectionPoints(crossroads, secondaryCargoList, currentMLH);

     //distances of crossroads
     distances = EuclideanDistances(connectionPoints, currentMLH);

     //for each crossroad, connect to the most convenient crossroad
     for (i = 0; i < distances.len(); i++) {
         FindAndConnect(crossroads, distances[i]);
     }

     crossroadCount = crossroads.len();
     //now we need to unite connections at nodes which has more than allowed connections
     for(i = 0; i < crossroadCount; i++) {
         if(crossroads[i].networkConnections.len() > GetCrossroadMaxConnections(crossroads[i].type)) {
             //more connections, unite them, creating a new BBH
             crossroads = UniteConnections(crossroads, crossroads[i]);
         }
     }
 }

 //at this point we connect SLHs and MLHs to virtual crossroad
 for(local i = 0; i < crossroads.len(); i++) {
     if(crossroads[i].type != CrossroadType.SLH && crossroads[i].type != CrossroadType.MLH) continue;

     local targetX = AIBase.RandRange(AIMap.GetMapSizeX());
     local targetY = AIBase.RandRange(AIMap.GetMapSizeY());

     virtual = Crossroad(CrossroadType.VIRTUAL, -1, AIMap.GetTileIndex(targetX, targetY));
     crossroads.push(virtual);

```

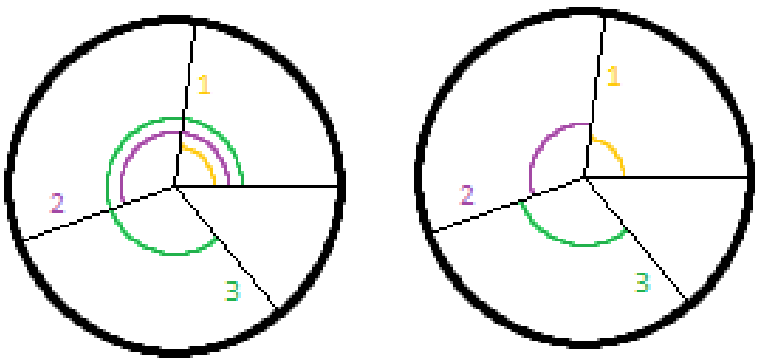

Figure 3.7: Total angles (left) and Local angles (right)

```
42          crossroads[i].Connect(virtual);
43      }
44
45      return crossroads;
46  }
```

Algorithm 3.9: Connecting the crossroads in the network

3.7.7 Optimization of crossroad connection number

Each crossroad can have up to 3 connections, in order for the network to be not cluttered together. If in the connecting of crossroads any of the crossroad did have more then 3, the *UniteConnections()* function has been called (3.10). Its purpose was to instead of some of the connections, create a new BBH and reconnect part of the connections so that all crossroads have maximum of 3. At the beginning x and y coordinate of selected crossroad is stored. The total angles are calculated by the GetTotalAngles() method (3.11) and right after that local angles are calculated as well (3.12). Total angle is an angle measured between pure east direction and the direction of the given point. On the other hand, Local angles are measured from the angle of the previous point to the angle of the current point as shown in image 3.7. Then the local angles get sorted and the lowest local angle is selected (connections which has the lowest local angle between themselves. Those connections get united and one connection is created instead of those two. New BBH is created - details about this method in 3.7.8. Crossroads sharing the two original connections are disconnected, new BBH is inserted to the network and the source MLH and two target crossroads are all connected to the new BBH. This algorithm can repeat more times until the correct conditions are met. Finally *AngleSortFunction* is function which sorts angles by their value in an ascending order.

```
1  function ConnectionPlanner::UniteConnections(crossroads,
       selectedCrossroad) {
2      //now we need to order the connections by the total
       angle they have to the 0 angle (means a "connections"
       straight east
```

```
3       //then we calculate the angle between each two neighbour
        connections
4       //finally we process them
5
6       //x and y of the selected crossroad
7       x = selectedCrossroad.GetX();
8       y = selectedCrossroad.GetY();
9
10      totalAngles = GetTotalAngles(selectedCrossroad);
11
12      //we have the array of angles, sort it
13      totalAngles.sort(AngleSortFunction);
14
15      localAngles = GetLocalAngles(totalAngles);
16
17      //now we have the array of local angles, sort it again
18      localAngles.sort(AngleSortFunction);
19
20      //now we have localAngles sorted
21      crossroad1 = localAngles[0].crossroad1;
22      crossroad2 = localAngles[0].crossroad2;
23
24      //TODO find a good spot for the new BBH
25      newBBHLocation = FindBBHSpot(crossroads, crossroad1,
    crossroad2, selectedCrossroad);
26      if(newBBHLocation == null) {
27          return crossroads;
28      }
29
30      //make new BBH
31      newBBH = Crossroad(CrossroadType.BBH, crossroads.len(),
    newBBHLocation, []);
32      crossroads.push(newBBH);
33
34      //reconnect crossroads
35      newBBH.Connect(selectedCrossroad);
36      selectedCrossroad.Connect(newBBH);
37      newBBH.Connect(crossroad1);
38      newBBH.Connect(crossroad2);
39      crossroad1.Connect(newBBH);
40      crossroad2.Connect(newBBH);
41
42      selectedCrossroad.RemoveConnection(crossroad1);
43      selectedCrossroad.RemoveConnection(crossroad2);
44      crossroad1.RemoveConnection(selectedCrossroad);
45      crossroad2.RemoveConnection(selectedCrossroad);
46
47      return crossroads;
48  }
```

Algorithm 3.10: Uniting the connections and creating new BBH

```
1
2  function ConnectionPlanner::GetTotalAngles(selectedCrossroad
   ) {
3      totalAngles = [];
4
5      for(i = 0; i < selectedCrossroad.networkConnections.len
   (); i++) {
6          //use atan2 for angle
7          //we need to get the [0,0] point which is the
   selected crossroad and then diff of the other crossroads
   from the selected.
8          //those two values are then used to the atan2
9
10         secondCrossroad = selectedCrossroad.
   networkConnections[i].target;
11
12         //x and y of the second crossroad
13         x2 = secondCrossroad.GetX();
14         y2 = secondCrossroad.GetY();
15
16         //this is the actual atan2 argument point
17         xDiff = x2 - x;
18         yDiff = y2 - y;
19
20         //angle in radians
21         angle = atan2(xDiff, yDiff);
22
23         //angle in degrees, need to be normalized to [0,
   360) degrees
24         tmpDegrees = angle / PI * 180;
25
26         //normalized degrees
27         degrees = tmpDegrees > 0 ? 360 - tmpDegrees : -
   tmpDegrees;
28
29         //add angle to the array
30         totalAngles[i] = CrossroadTotalAngle(degrees,
   secondCrossroad);
31     }
32
33     return totalAngles;
34 }
```

Algorithm 3.11: Method for calculating total angles

```
1
2  function ConnectionPlanner::GetLocalAngles(totalAngles) {
3      localAngles = [];
4      for(i = 1; i < totalAngles.len(); i++) {
```

```
5           localAngles.push(CrossroadAngle(totalAngles[i].angle
    -totalAngles[i-1].angle, totalAngles[i].crossroad,
    totalAngles[i-1].crossroad));
6       }
7
8       //push the angle between last and first crossroad as
    well
9       localAngles.push(CrossroadAngle(totalAngles[0].angle +
    360 - totalAngles[totalAngles.len()-1].angle, totalAngles
    [0].crossroad, totalAngles[totalAngles.len()-1].crossroad
    ));
10  }
```

Algorithm 3.12: Method for calculating local angles.

3.7.8 Finding BBHs

The algorithm for finding spot for new BBH is quite similar to finding MLH spot (3.7). The only real difference between the two algorithms is the selection of the initial center point. In case of BBH finding, we have three points related to the BBH center finding - center of the original MLH and centers of two crossroads where the two united connections lead to. The initial center point x is calculated as sum of the x coordinate of the previous three points, same goes for y coordinate. Then the initial center point of the crossroad is [x,y].

3.7.9 Finding in and out points of the crossroad

Finding entrance points of a crossroad is a problem which is influenced the most by the bounded rationality. There are plenty of problems that can occur while finding entrance points. Because of that, if any of these problems occur, the points are simply discarded and new points are being generated. Sometimes it happens that crossroad is stuck in a place that there is not possible to fit entrances anywhere. For that case, on each iteration except the first even the crossroad center itself can be slightly moved against the original position (still cannot be too close to another crossroad). Approach of those two modifications has very high chance of success and because of the bounded rationality problems, we cannot simply iterate over all combinations and find the best one because that would be too expensive for the processor time.

The function GetInAndOutPoints() (3.13) finds angles of the connections at the beginning. Then it tries to move the crossroad center a bit. Then it repeatedly tries to find entrance points until it is successful. In first step it takes the local angles of the connections and randomize them. After that they get corrected so they are not close together. Finally points are retrieved from the angles via the GetRecalculatedPoints() method. It simply calculates point based on the crossroad center, radius and angle provided by the method for each angle there is. As we have only one point per connection so far (first point out), then each point is expanded and first point in is added to each entrance. If not successful, points are discarded and new iteration begins. After that those two points for each connection are expanded further if 3-way crossroad (it requires two in and

two out points per connection as one point will lead to the second connection and the other one to the third connection). Sometimes it happens that points are located in the corner of the square around the crossroad center. Those points are discarded as well.

Last thing that is checked is if the entrance is clear. Few rows of tiles in front of the entrance tiles and behind the entrance tiles are tested if they are buildable. If all are buildable, then the entrance points are found and the cycle ends. Because of a lot of randomization and moving of the points, it is possible that entrances assigned to each connection are in a sub-optimal combination - they are leading other way then to the target point. Those connection then look weird when built if the connection needs to for example run around the whole crossroad because it was built on the other way then it needed. When the points are found, the algorithm will reassign the points to the connections. That means that all combinations of assignment of each points are checked and difference between angles of the entrance and the connection target is calculated. For further improvement of the result, the differences are squared. That ensures that if a difference in angles of one entrance is very big, it is seen as less viable solution then if all three entrances has the difference low.

After the reassignment the points are set to the crossroad and it is returned from the method.

```
function ConnectionPlanner::GetInAndOutPoints(crossroad) {
    angles = GetAngles(crossroad);

    for(i = 0; i < MOVE_CROSSROAD_CENTER_TRIES; i++) {
        if(i > 0) {
            newCrossroad = ConnectionPlanner.
   MoveCrossroadCenter(crossroad);
            crossroad = newCrossroad;
        }

        for (local j = 0; j < MAX_ENTRANCE_POINT_TRIES; j++)
    {
            entranceNotClear = false;

            //find random angles close to original angles
            localAngles2 = FindRandomAngles(angles);

            //correct the angles so they are not close to
   each other (not lower then threshold)
            localAngles3 = CorrectAngles(localAngles2);

            //get points from angles
            points = GetRecalculatedPoints(crossroad,
   localAngles3);

            newPoints, invalid = GetAllBasicPoints(points);

            if (invalid) { continue; }

```

```
26              entranceClear = true;
27              expandedPoints = ExpandInAndOutPoints(crossroad,
    newPoints);
28              points = expandedPoints;
29
30              if (!CheckNotInCorners(newPoints)) {
31                  continue;
32              }
33
34              for (k = 0; k < points.len(); k++) {
35                  if (!CheckEntranceFree(points[k], crossroad)
    ) {
36                      entranceClear = false;
37                      break;
38                  }
39              }
40
41              if (entranceClear) {
42                  //now reassign the entrances to different
    targets as they could have been move and not be assigned
    optimally
43
44                  reassignedPoints = ReassignInAndOutPoints(
    crossroad, expandedPoints);
45                  points = reassignedPoints;
46
47                  //set the points
48                  SetPoints(crossroad, points);
49              }
50          }
51      }
52
53      return crossroad;
54  }
```

Algorithm 3.13: Method for finding in and out points of the crossroad

3.7.10 Creating a crossroad

A problem of building a crossroad has been solved by serial runs of A* algorithm to find each paths. If the crossroad had two entrances, only two A* instances were run, from point *AIn* to point *BOut* and from point *BIn* to point *AOut*. If crossroad had three entrances however, there were total of 6 A* instances as shown on image 3.8. Point *AIn1* connects to *COut1*, *AIn2* to *BOut2*, Bin1 to *AOut1*, *BIn2* to *COut2*, *CIn1* to *BOut1* and *CIn2* to *AOut2*. At first the *AIn1*, *BIn1* and *CIn1* are connected so the inner connections has less problems being built. After that the outer connections are built as well. While finding paths in the crossroad, if something fails, the paths are cancelled and the search will start again, but with different entrances. Algorithm for finding entrances 3.13 contains element of randomness so finding new entrances avoid the unsuccessful solution to be tried

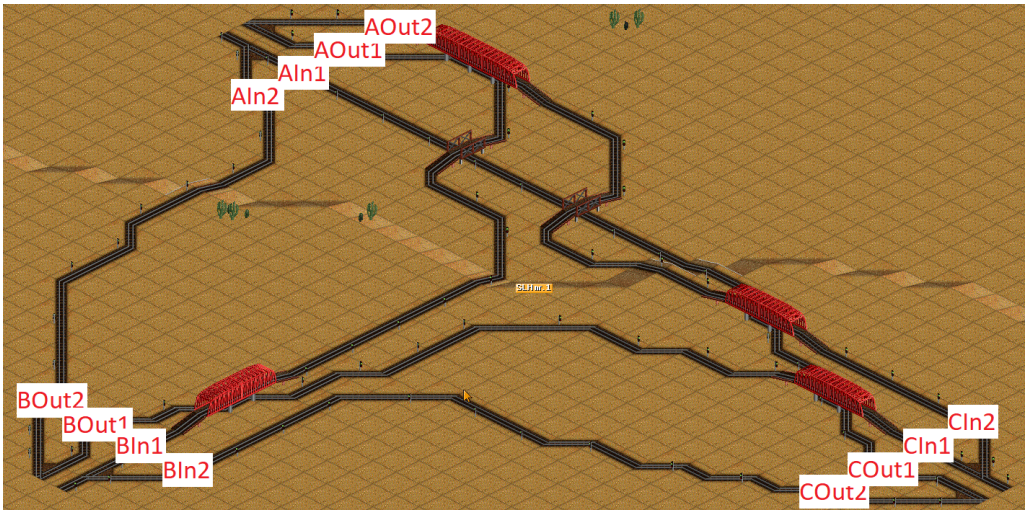

Figure 3.8: Crossroad entrance points

again. When in building phase, building of the crossroad can fail as well due to terrain or environment changing. In that case, all rails built on the crossroad so far are deleted and new entrances are found, new paths are found and built.

If the building of crossroad is successful, entrance points will be built. Before that terrain under the entrance is leveled so it has the same height. After that, entrance tiles are built (yellow section on image 3.2). When planning entrances, the algorithm checks those entrance tiles for being buildable, so there are no problems with building them.

3.7.11 Trains

The last algorithm of the TrainsferAI is algorithm for building trains on a route. There are two depot spots by the station. Each station will get one depot built on one of those spots. When building trains, a number of trains is determined. At first distance and production industry is retrieved. Wagon count is calculated as

$$\textbf{wagon count} = \textbf{train length} * 2 - 1$$

because two wagons fit on each tile and -1 is for engine. Then wagon capacity is retrieved and with that value a train capacity is calculated as

$$\textbf{train capacity} = \textbf{wagon count} * \textbf{wagon capacity}$$

. Number of days one train will travel to the unloading station and back is calculated as

$$\textbf{days traveling} = \textbf{distance} * 2 \ / \ (\textbf{speed} * \textbf{FIELD\_PER\_DAY\_PER\_KMH})$$

FIELD_PER_DAY_PER_KMH is a constant calculated manually which means how many tiles a train will travel in one day if it would have a speed of 1 km/h. Then cargo produced while train is gone is calculated as

cargo produced while train is gone = (days traveling / 30) * cargo produced in one month

As this calculation does not need exact number, days in month are approximated to 30. Finally train count is calculated as

trains = cargo produced while train is gone / train capacity

That means that the amount of days a train is gone is the loop of the cargo. Approximately after that amount of days the first train will arrive again to the station. During this time some cargo has been produced. This amount of cargo then needs to be divided by the capacity of the train to figure out how many trains is enough to cover the cargo until the first train will come back to the station. Note that if the train count is lower then 2, it is set to 2. For the industry to increase production, certain percentage of the cargo each month has to be transferred away. If there would be only one train, some months the cargo would not be transferred because the train would be away, thus the industry would probably not increase its production.For regular routes the minimum train is set to 7 and the train count is doubled so it covers the industry production increase in the future.

3.8 Preferred game settings

The game should have some settings set up for the TrainsferAI to be functioning properly. All of those settings correspond to what members of OpenTTDCoop have set up in most of their games as well. The first is *Vehicles - Physics - Train acceleration model: Realistic*. It ensures that the acceleration model is set up correctly for the network - single turns do not slow, double turns slow. The next one is *Disasters/Accidents - Vehicle breakdowns: None*. This prevents the vehicles to breakdown, slowing all the traffic in the network.

TrainsferAI is optimized for running in the desert environment (third one in the game menu). For correct results the AI should be ran in that environment.

If the settings are not set up correctly, the AI will still work. But the efficiency of the network will be much lower because one breakdown will stop all trains behind the broken one, leading to massive jams if more trains are in the network.

4. Validation

Validation of the AI has three main parts - performance validation, success rate validation and aesthetic validation. All three parts will be examined in details, providing evaluation of the whole AI agent.

The evaluation was done on 512x512 maps in the first round, on 1024x1024 maps in the second round. 5 moneymaking routes was used at the beginning to earn money. on 512x512 maps, only 3 SLHs were build as the map was not quite big enough to have more. On the 1024x1024 map, 10 SLHs were set to be built.

4.1 Performance validation

The game does not provide much processor time to the AI agents. Due to this, a performance issues were expected at the beginning. Space search is usually quite time-consuming and the state space is not very small. The minimal turn length condition greatly increases the search complexity as well. During the development of AI, many performance improvements have been made to decrease the time needed to plan and build as described in 3.7.2. The end result was that the AI would earn approximately the same amount of money as the maximum loan set to the highest possible value per year. That is a lot more then average player can earn during first game year of the game. During the run of the AI the agent was able to spend all money he had to build the network and the construction was often even waiting for that money. Therefore the performance did not influence the run of the AI.

4.2 Success rate validation

Second validated aspect of the AI was success rate. Because of the bounded reality, some problems can occur meaning the network will sometimes partially fail to build. Therefore, tests have been done to evaluate the success rate of the whole network. 50 games were run at 512x512 maps, 50 were run at 1024x1024 maps. On the resulting graph we can see that 1024x1024 maps have higher success rate. The reason behind this is probably the connection success rate which is much higher on bigger map. The crossroads are placed more spread out and the connection between them is easier to find without problems. The crossroad building success rate is similar on both types of the map.

Most of the problems which occurred during the testing were missing pieces of rails caused by being unable to correctly modify the terrain. That causes part of the network to not work correctly. Some fails were caused by industry being destroyed before the AI would build the connection there. Reacting on the world events was left for future work as it is quite complex mechanic which has a lot of different aspect to be covered. Those kinds of fails were included to the ”Connection building failed” category.

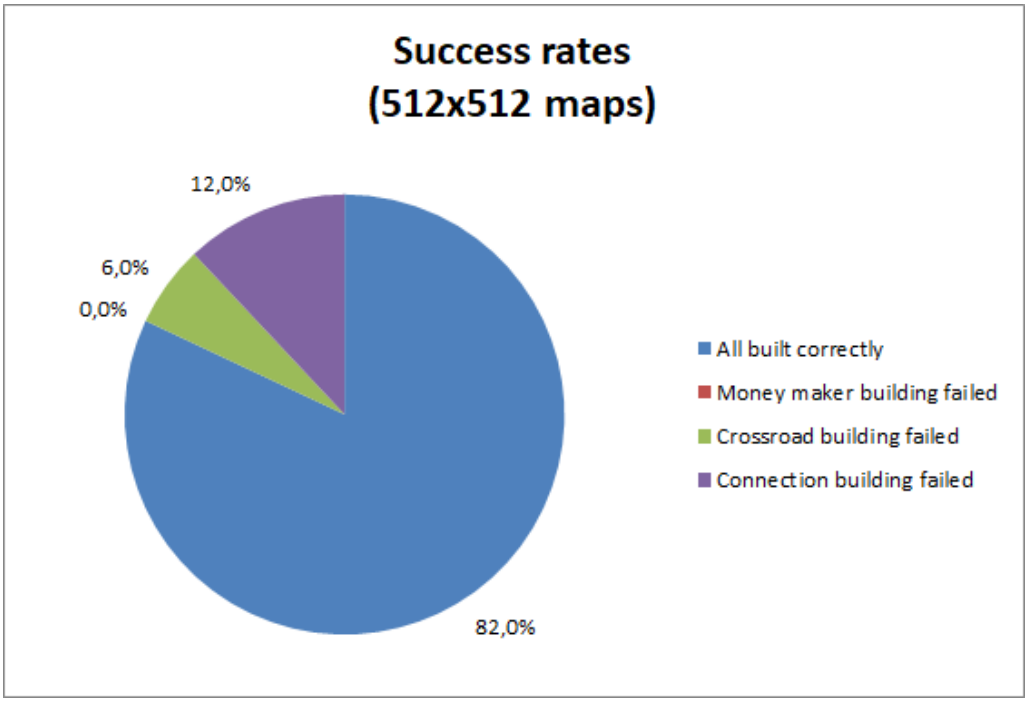

Figure 4.1: Success rate results for 512x512 maps (50 games)

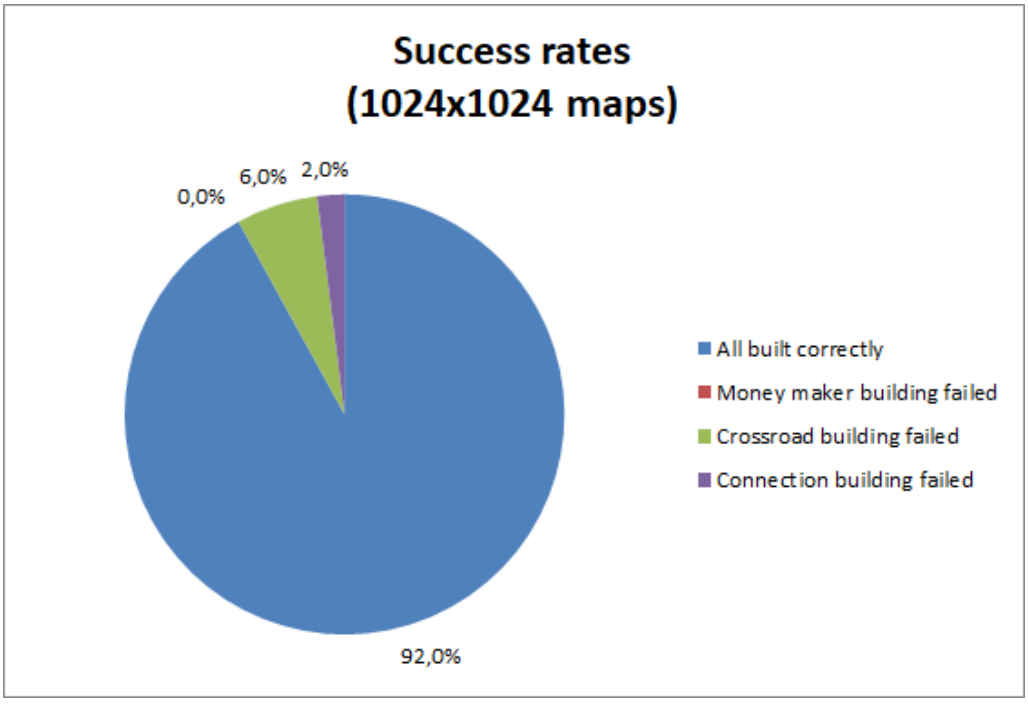

Figure 4.2: Success rate results for 1024x1024 maps (50 games)

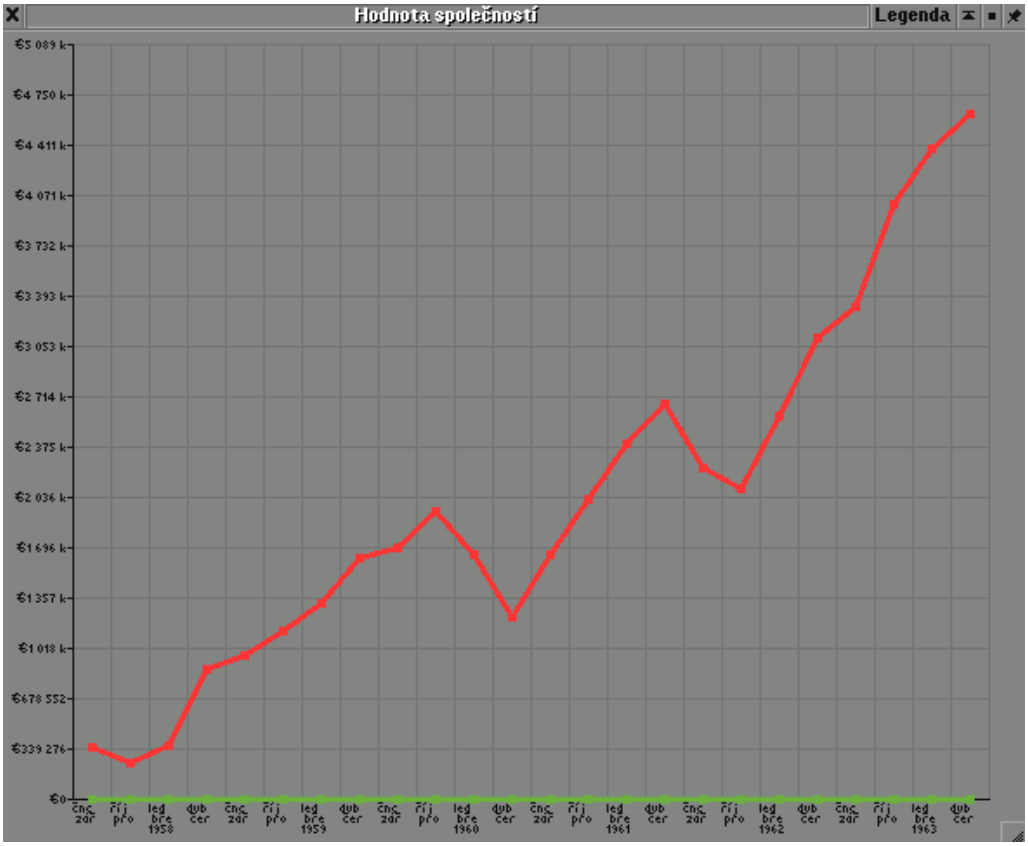

Figure 4.3: Company value of TrainsferAI on 1024x1024 map

4.3 Bank account and company value validation

During the run of TrainsferAI bank account was usually around 0 euro. This is due to the AI building slightly faster then the money income from the moneymakers. In fact keeping bank account as low as possible is good because the more money is spent on building new infrastructure and new routes, the earlier will the routes earn more money. The important feature of the company is Company value. Company value is the value of bank account + sum of all things the company has built so far. So it approximates the economical growth of the company. TrainsferAI reached good results when increasing company value. During 7 game years, the company value was usually around 5 mil euro, while the starting maximal load was 1 mil euro. The company value fluctuated around 5 mil euro, sometimes it was lower - it depends on the time spent on building the network which differed quite a lot. Graph showcasing the company value progress over the 7 years is displayed on image 4.3.

4.4 Aesthetic

Evaluating aesthetic features was done through comparing the network structure similarities with the OpenTTDCoop networks from public servers. The number of rails, number of stations and trains were neglected in the comparison as it is out of the scope of the thesis to create such a huge networks. Instead the layouts and the resulting scheme on the map was compared. The structure looks quite similar

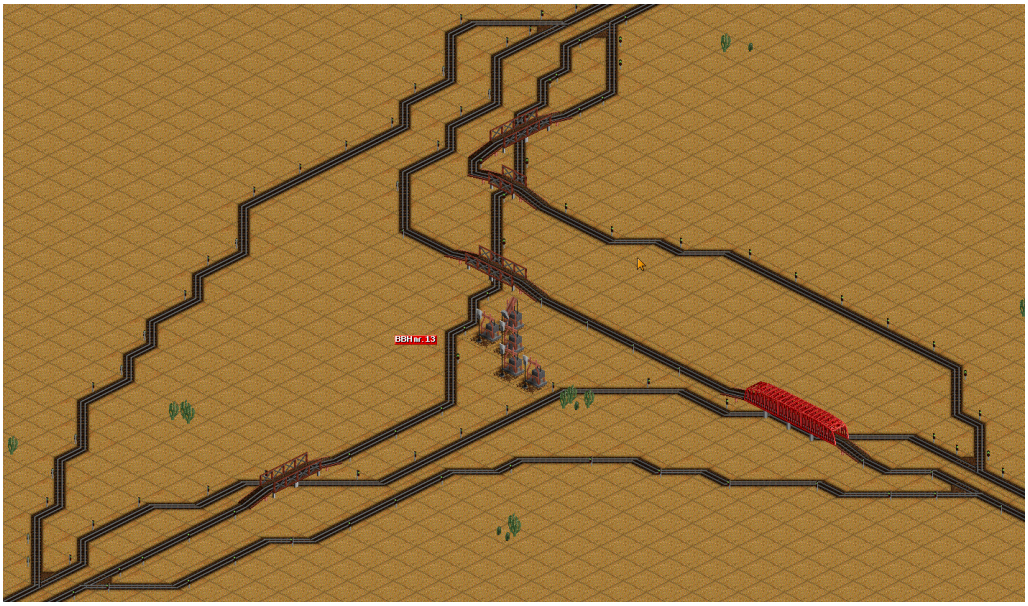

Figure 4.4: Crossroad going around industry

even though there are some differences - the entrances are not on the correct places every time so the connection sometimes has to go around part of the crossroad. The networks built by TrainsferAI are more squared, while the OpenTTDCoop networks usually go better along the terrain, using its features better.

4.5 Results

Results can be best seen as a series of pictures. At first a few pictures of interesting crossroad designs will be shown 4.4, 4.5, 4.6, 4.7, followed by images of the network layouts shown on the in-game minimap 4.8, 4.9.

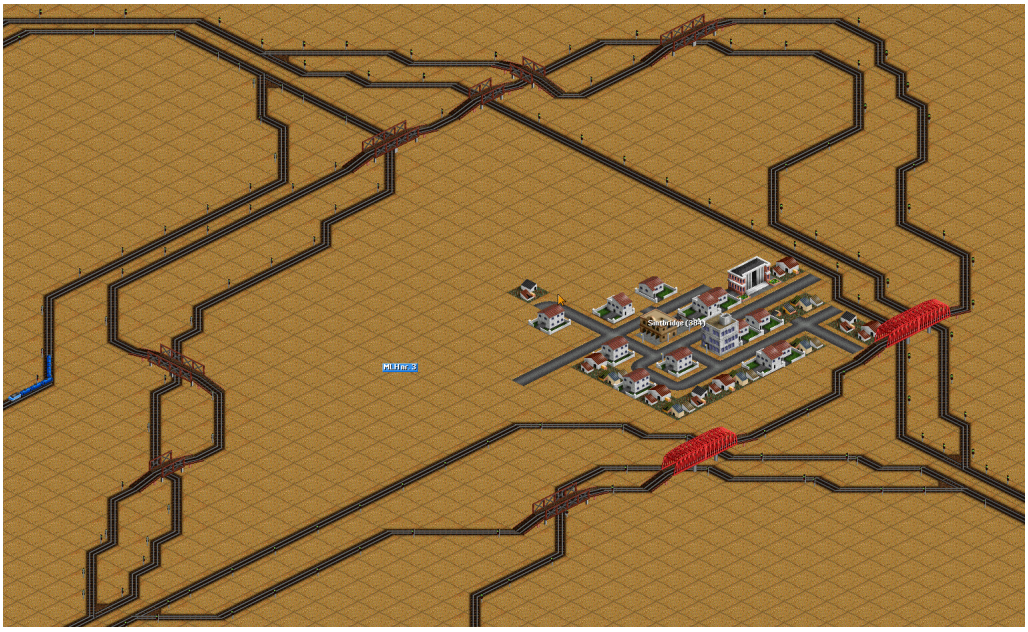

Figure 4.5: Crossroad going even around a city

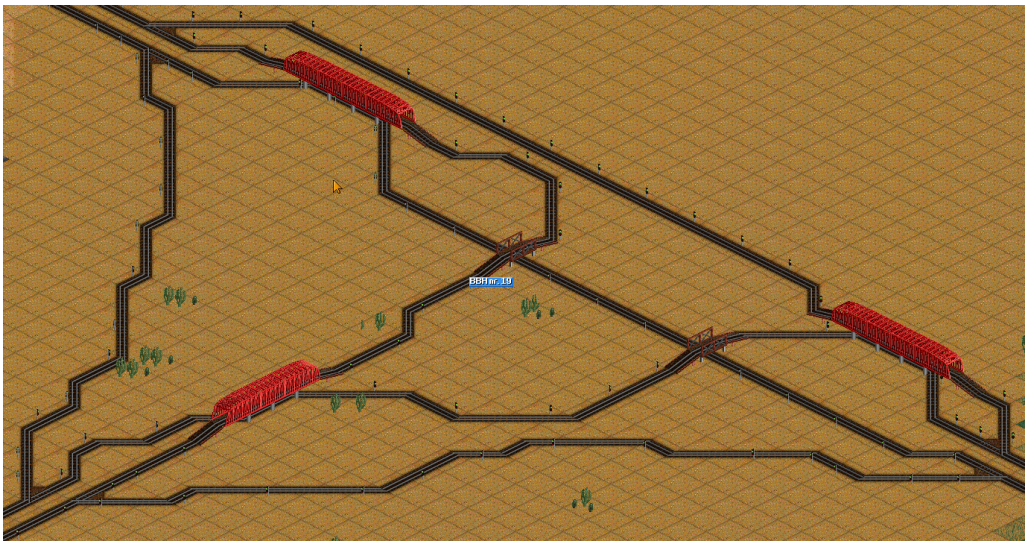

Figure 4.6: Another crossroad

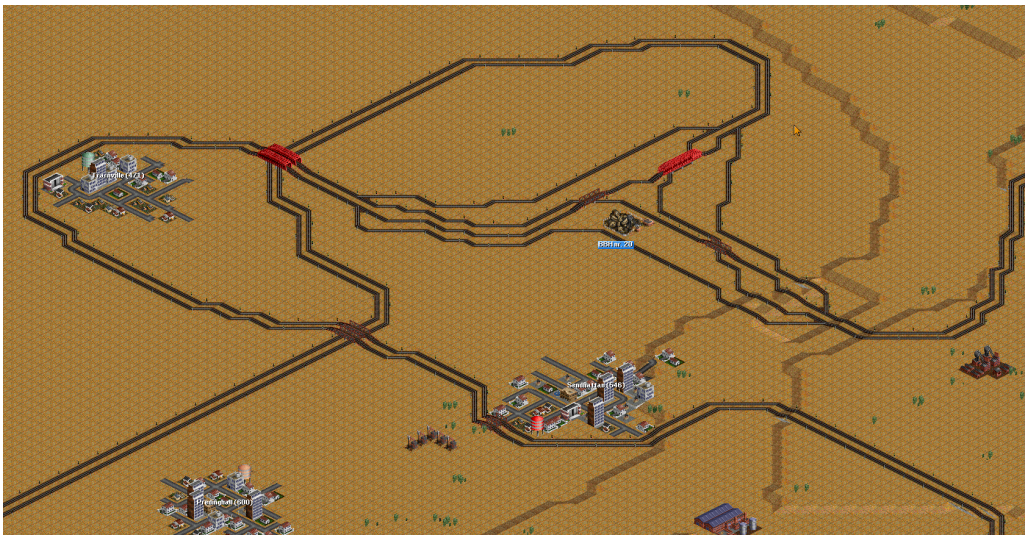

Figure 4.7: Bigger crossroad area

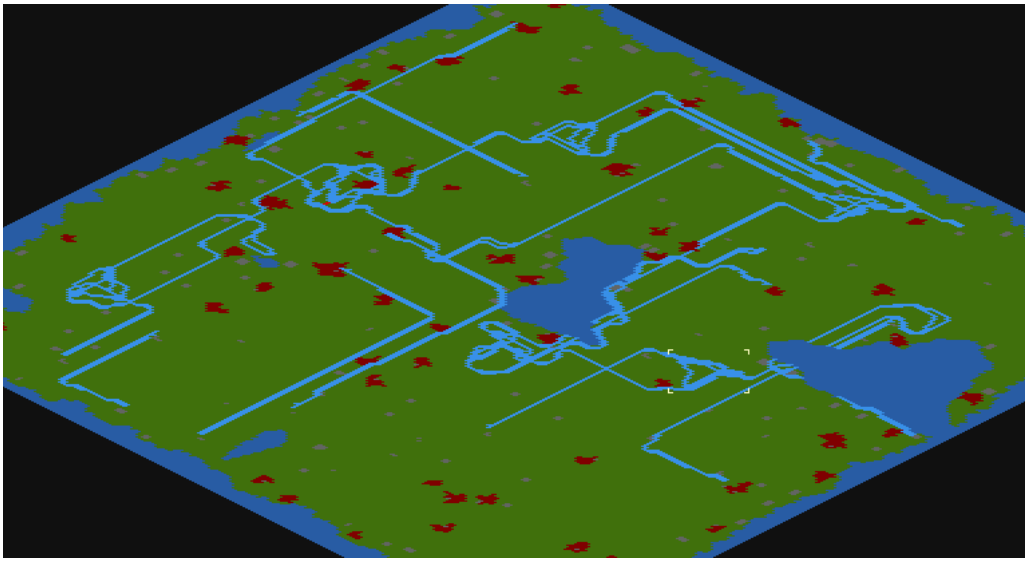

Figure 4.8: Result scheme from game 14 on 512x512 map

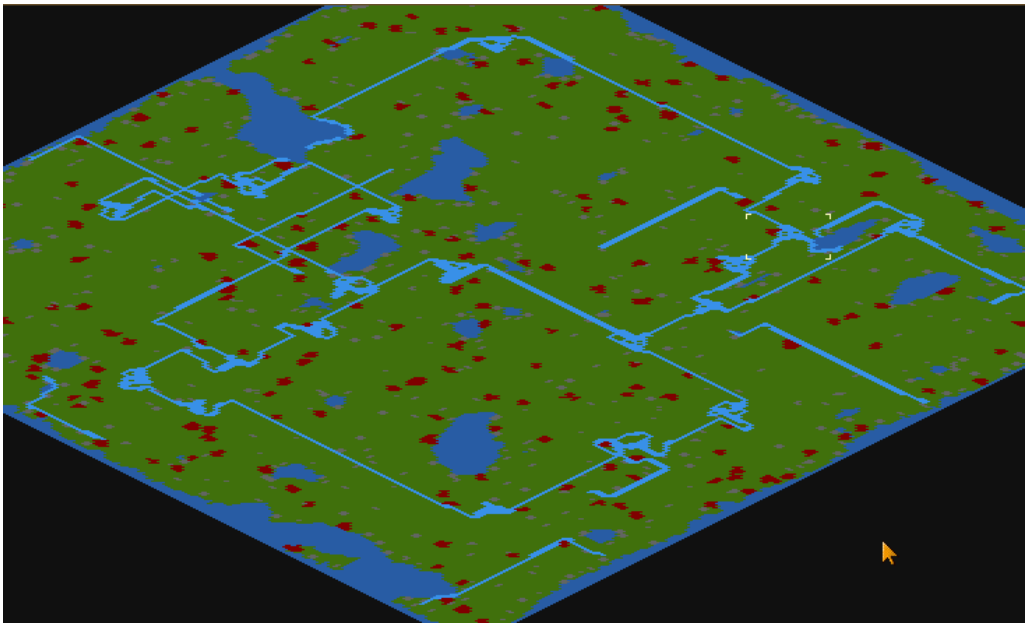

Figure 4.9: Result scheme from game 31 on 1024x1024 map

5. Future work

The human-like construction in games is a big topic and many of the things were out of the scope of the thesis. Many of the features would greatly enhance its impact. Elements of the AI that could be extended are many, but the most important ones are network extendibility, train management, network management and multiplayer.

5.1 Network extendibility

So far the basic scale of the network has been implemented. Each connection has one rail to one direction and one rail to the other direction. Each crossroad has one rail leading to each other side of the crossroad. Network scaled like this can run couple of tenths, maybe even hundred of trains depending on the map size. The network could be extended to support more trails in each direction for connections and more rails to each other entrance of crossroads. It would require far more time for the paths to be built and some additional modification of the code. Networks of OpenTTDCoop group sometimes has even 5 or more rails in one direction. That is way out of the possibilities of the OpenTTD AI as it is right now because the time it would spent planning the network and building would be huge. A possible solution would be to modify source code of the OpenTTD itself so it would support connecting of third party program to the game and enable AI to connect to it. Then the planning could be done externally and the performance would be much better.

Another thing that could be extended is connecting more stations to one crossroad (specially for SLH). That would need a lot of extra work because the AI would need to somehow solve where to connect additional stations. If this feature would be implemented, it would greatly increase the amount of trains that the AI would build on a map. This feature is dependable on the previous point because so far the network would not be capable of running so many trains anyway.

5.2 Train management

Another interesting thing to be extended is train management system. So far the only thing the AI is doing about trains is that it calculates the estimated number of trains required to run a route. But the industry production is changing and the AI could react to that. It would sell some trains if the industry would greatly decrease its production or it could buy more trains if the production would go up. This feature is again dependant on the network extendibility, because too many trains would flood the network and cause it to collapse.

5.3 Network management

Very important thing which could be added to the AI is network management. Calculating if the network is big enough for the amount of trains that are running on it is very important if the amount of trains start to go above hundred. Network

management is greatly connected to previous two points because the management would determine when to extend parts of the network, but it would probably not be needed if the amount of trains would stay low.

5.4 Multiplayer

One feature which is not supported by the TrainsferAI is multiplayer. And not only playing versus a player opponent, but another AI as well. So far the AI agent plans parts of the network and builds them afterwards. But there is a time delay between those steps and if something has built over the tiles where the network is planned to be, the AI is going to have a hard time. It would require basically to add validation after each step the AI is doing. It would validate if the step could have been done and if not, it would execute steps to deal with the new situation. This is not contained in the scope of the thesis nor the original purpose of the AI, but it could be handy to implement it because that way the AI could attend some AI matches and competitions.

5.5 Covering more environment

As for now, TrainsferAI runs correctly in the desert environment. For a future work it would be handy to cover all other environments. The industry rules are slightly different there so it would need some work.

6. Conclusion

In this thesis, we tried to implement an AI agent which would be run in business simulation game Open Transport Tycoon Deluxe. It should be able to run a company, earn money and mainly build railway network with a human-like features. It was inspired by OpenTTDCoop group, which builds huge train networks in the game.

An AI agent has been implemented. At the beginning it builds some money making routes to earn money for the network construction. Then it plans the network based on real-time data from the game. When the network is planned, the agent will gradually build it. When the network is completed, it starts to find industries and starts to connect them to the network. When a connection is created, it builds some trains and run them on the network.

The evaluation was done in few steps, evaluating different aspects of the AI. At first, performance was evaluated. Next important factor of evaluation was success rate. Then money management was evaluated and finally the aesthetic part of the building.

Bibliography

Brilliang.org. A* Search, 2016. URL https://brilliant.org/wiki/a-star-search/. Accessed 15 December 2019.

Coldewey, D. StarCraft II-playing AI AlphaStar takes out pros undefeated, 2019. URL https://techcrunch.com/2019/01/24/starcraft-ii-playing-ai-alphastar-takes-out-pros-undefeated/. Accessed 11 December 2019.

Demichelis, A. Squirrel - The Programming Language, 2016. URL http://squirrel-lang.org/. Accessed 15 December 2019.

Dr. Garbade, Michael J. Understanding K-means Clustering in Machine Learning, 2018. URL https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1. Accessed 2 Januar 2020.

Michiel. ChooChoo, a train network AI - Transport Tycoon Forums, 2009. URL https://www.tt-forums.net/viewtopic.php?t=44225. Accessed 18 December 2019.

OpenTTD Team. OpenTTD — About, 2019. URL https://www.openttd.org/about.html. Accessed 11 December 2019.

openttd wiki contributors. AI:Main Page - OpenTTD, 2015. URL https://wiki.openttd.org/AI:Main_Page. Accessed 15 December 2019.

openttd wiki contributors. OpenTTD - OpenTTD, 2018. URL https://wiki.openttd.org/OpenTTD. Accessed 13 December 2019.

Rios, L. and Chaimowicz, L. trains: An artificial inteligence for openttd. *2010 Brazilian Symposium on Games and Digital Entertainment*, 0:52–63, 01 2009. doi: 10.1109/SBGAMES.2009.15.

Sawyer, C. Transport Tycoon, 2013. URL http://www.transporttycoon.com/history. Accessed 11 December 2019.

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. Conflict-Based Search For Optimal Multi-Agent Path Finding, 2012. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/viewFile/5062/5239. Accessed 18 November 2019.

The openttdcoop. PBS - #openttdcoop wiki, 2013. URL https://wiki.openttdcoop.org/PBS. Accessed 12 December 2019.

tutor2u. Bounded rationality — Topics — Economics — tutor2u, 2018. URL https://www.tutor2u.net/economics/topics/bounded-rationality. Accessed 28 December 2019.

List of Figures

List of Tables

List of algorithms

List of Abbreviations

CPU - Central processing unity

CSV - Comma-separated values

DOTS - Data oriented tech stack

ECS - Entity component system

GPU - Graphics processing unit

IL - Intermediate language

IR - Intermediate representation

RAM - Random access memory

SIMD - Single instruction multiple data

UI - User interface

SLH - Sideline hub

MLH - Mainline hub

BBH - Backbone hub

AI - Artificial intelligence

A. Instalation guide

For installation of the package of this thesis one needs to install the game itself at first. Game downloads are available at webpage `https://www.openttd.org/downloads/openttd-releases/testing.html`. After installing the game, the game creates a data folder (default location is at "C:/Users/user_name/Documents/OpenTTD" - for later references the "default location" would mean this folder). To install the AI package, we need to copy the .zip file to the folder "default location"/content_download/ai. We can extract the folder or leave it in the .zip file. We then close the game if it has been open, and start it again. To bring the new AI to new game, in the main game menu, we click the "AI/Game Scripts Settings" (A.1). Then a pop-up window show up with settings of the AI for the game (A.2). We set the maximum number of opponents to a number greater equals 1 and we set the first opponent to TrainsferAI - we do that by clicking the second row (below "Human player") and click "Select AI" in the left bottom of the window. Then we select the TrainsferAI as the AI. After the TrainsferAI has been selected, we close the dialog window. Then in the main menu, we select the desert environment (the third one) and start the game by clicking "New game" in the main menu.

Figure A.1: Main menu of the game

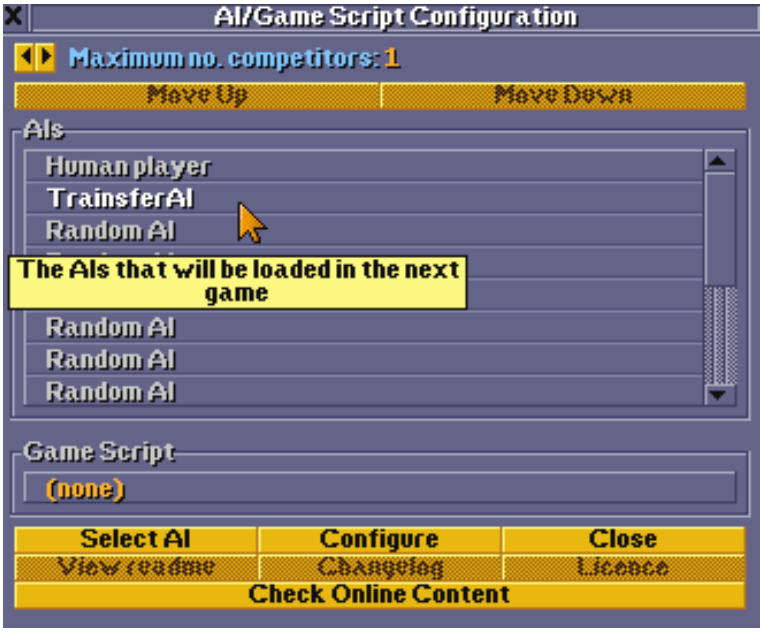

Figure A.2: Pop-up dialog for selecting the AI opponents