Abstract

For this project we designed and implemented an Artificial Intelligence program that plays *Open Transport Tycoon Deluxe* (OpenTTD or TTD), a business and construction simulation game. In OpenTTD, players attempt to earn profits by transporting passengers and goods using roads, rails, airplanes, and boats. Players are given a random map populated with cities, industries, natural resources, and obstacles, such as trees and water. They must create a transportation network connecting the sources of passengers (cities) and goods (industries and natural resources) in the most efficient and profitable way possible. In addition, they must maintain this network and deal with in-game events such as vehicle aging and breakdown, technology upgrades, offered subsidies, and actions by competitors operating on the same map.

Our goal was to develop an AI for OpenTTD that created an efficient transportation network that was both profitable in the game and theoretically viable in the real world. The AI attempts to design and implement a theoretically efficient network while working within and adapting to the constraints of the OpenTTD environment (ex. construction costs, terrain features, subsidies, and competitor actions). Through the development of this AI we hope to gain a better understanding of the modeling and design of transportation networks and other similar systems.

Introduction and Scope Paper

The AI developed for this project, officially named ZooElite, consists of two major levels of operation. The ‘lower level’ of operation performs tasks that allow for the physical construction of transportation routes between destinations. It determines and optimizes the placement and construction of new bus and rail stations, finds and builds routes between these stations, and creates vehicles to service routes. This level of ZooElite was the primary responsibility of Charlie Croom and is detailed in the first part of this paper.

The second, ‘higher level’ of the AI was the primary responsibility of Cameron Musco and is detailed in this part of the paper. This level of the AI determines which routes to construct and which order to construct them in. It also schedules and coordinates the completion of various in-game tasks, including station construction, route construction, vehicle purchase, vehicle updates, and financial actions, such as loan obtainment and repayment.
In this paper we will first describe the design constraints and final implementation of the route planning portion of ZooElite. We will next explain how route planning was incorporated into and physically implemented by the “lower level” of the AI. We will then describe the overall program structure and task scheduling system implemented in the AI. Finally, we will discuss the performance of and intellectual value of the completed AI program. We will propose possible further improvements and additions to ZooElite, as well as improvements to the *Transportation Tycoon Deluxe* platform.

### 1. The Transportation Network

#### 1.1 Game Mechanics

In *OpenTTD* profits are earned by transporting goods from sources to destinations. There are two basic types of goods – material goods and passengers.

Material goods and resources are transported from specific source locations to specific sink locations. For example, coal may be transported from a mine to a power plant, steel from a mill to a factory, and products from a factory to a town for consumption. In our view, the game mechanics governing the use of material goods are not conducive to creating a realistically efficient transportation network. Pay rates for the delivery of material goods are based on distance as much as quantity, so transporting coal from a mine to a distant power plant can often be much more profitable than transporting to the nearest power plant. Since the game fails to take into account market forces when determining pay rates, the most successful transportation networks bare little resemblance to networks that would be desired in the real world.

Since we wanted our AI to build a network that was both successful in *OpenTTD* and also implemented strategies and design that would be applicable to the real world we focused entirely on the transportation of passengers. Unfortunately the pay rates for passenger delivery are also based on distance as well as quantity. However this effect is less severe than it is with industries. In addition, the CargoDest patch to *OpenTTD* provides each passenger with a specific destination, creating a more realistic model of real world transportation. Future versions of CargoDest are planned to take into account distance when determining demand for destinations (i.e. more passengers would need to travel to close destinations than far destinations). These versions will make *OpenTTD* more realistic. Keeping in mind our goal of creating an AI that can design transportation networks that would be reasonable in the real world, we took these planned future changes into account during the design process.

#### 1.2 The Transportation Network as a Graph

Immediately it seems obvious that the transportation network designed by the AI can be modeled as a graph. Each station represents a node and each route between stations represents an edge with a certain capacity. Resources are generated at nodes located within towns and cities (stations receive passengers from surrounding adjacent populations) and flow along the edges between nodes. Each node has a given capacity to
receive resources from other nodes, as well as a capacity for the through flow of passengers. For example, a station located within a city may serve a sink for passengers, while a regional station may serve simply as a transfer point for passengers traveling to other destinations. Profits are maximized by maximizing the flow of passengers between nodes.

The model of the transportation network as a graph is complicated however since nodes are not given, but must be placed strategically according to the geography of the randomly generated map. In addition, the choice of routes is unlimited – routes may flow between any nodes, may have any capacity, and may be constructed at any time. So, although many techniques and algorithms from graph theory may be used to design the network, this problem has degrees of freedom not typical in more theoretical graph theory and network optimization problems.

In order to simplify the problem we first break nodes up into classes depending on relative capacity. This method is seen in real world transportation networks where different nodes have widely varying capacities and functions – from a local bus stop to an international airport. In our graph one level of node is immediately obvious. The population of the map is not evenly distributed but concentrated in towns and cities of varying population. This pattern can be seen in figure 1.2.1 below:

![Figure 1.2.1: Five towns on a small OpenTTD map](image)

We serve each town with a cluster of bus stations that serve as sub-nodes and are connected by the existing road network of the town. The sub-nodes are placed and built by the lower level of the ZooElite AI in order to maximize the population in each town with access to a bus station. The cluster of bus stations forms a complete graph that can be treated as a single node of our network located in the town.

So, at our level of AI the lowest node level is the town. Theoretically, we could form a network of inter-town connections and attempt to maximize the flow of
passengers along this network. However, this leads to a variety of problems. First, computationally, considering the many possible routes between different towns is quite burdensome, especially with the slow run time of Squirrel, the language that the AI is programmed in. Computational time costs can have a real effect in a game where quick decision and action is necessary. In addition, an optimized network with all nodes located at towns would not be an overall optimal network. It can often be advantageous to place nodes in centrally located areas that are not towns. And, as we did with bus stations in towns, to form small node clusters that are connected to the wider network with higher capacity routes.

We implement this strategy by dividing towns into regions. Each region consists of a small number of adjacent towns, or for higher-level regions, a small number of adjacent sub-regions. A regional network has the following advantages (some mentioned above):

1) Allows for highly optimized networks – similar to those seen in real world transportation networks. Stations need not be placed at population centers but may be placed at intermediate locations to act as transfer points for traffic.

2) Allows for computational efficiency, as the route structure of each region can be determined separately from the route structure of the graph as a whole. This avoids having to precalculate our transportation network design, which would lead to a high front end time cost.

3) Allows the network to be profitable at each stage of construction. In the same way that we can calculate the route structure in regions separately, we can build each region separately, creating a profitable sub-network that can then be connected to the overall network. With sub-regions, we can recursively build up our network by starting at the smallest level regions and working our way up.

4) Allows for an easy determination of route capacity, with routes between higher level regions having higher capacity than routes between lower level regions.

A general layout of the regional network configuration is shown in figure 1.2.2 below:
Although the regional layout may have an unlimited hierarchy of regions and sub-regions, since the map size that we operate on is limited, it was never necessary to implement a many level regional system. We instead simply construct small ‘base regions’ that are connected to local towns through buses. Each base region contains a rail station (‘base station’) that allows it to be connected directly to other base regions. Our system however, could easily be generalized to a higher-level regional set up, with each base region serving as the analog of a town and being incorporated into higher-level regions connected with higher capacity routes. Figure 1.2.3 illustrates an example of our configuration, showing two base regions connected by a rail line.

1.3 Identifying Regions

The first step in creating a network based on the above regional layout is to identify regions. In ZooElite this was done by implementing a clustering algorithm, to identify distinct groups of closely placed towns (or subregions). A variety of clustering algorithms were implemented and tested:

*K-Means Clustering:* In this algorithm, the number of regions is preset to a constant, k, which is determined by the number of towns and the size of the map (or the number of subregions). The k regions are randomly positioned on the map and a loop is entered. In each iteration of the loop, each town is assigned to the region it is closest to. The region centers are then rebalanced to the center of masses of the towns they contain. This loop is repeated until the region set up stabilizes, meaning that reasonable regions have been
found. The k-means clustering algorithm gave very good results on our maps. However, due to its random nature, the fact that it was difficult to specify k without processing the configuration of the map, and the fact that our data set (the number of towns) was reasonably small (25-75), it would occasionally produce inefficient configurations. Since the regional configuration can have a very large effect on future profitability, we decided to go with a deterministic, rather than random, algorithm.

**QT Clustering:** This algorithm requires specifying a maximum cluster diameter but not a set number of clusters, making it preferable to k-means clustering. A maximum diameter calculation can be based on the type and capacity of transportation planned to be used in the region rather than complicated geographic parameters and configurations. In the algorithm each town starts at a candidate cluster. The cluster around the town is defined as all towns within the maximum region diameter of the town. The candidate cluster with the most towns is fixed and the process repeated until there are no remaining candidate clusters and all towns have been assigned to regions. Unfortunately on our map, since towns are not heavily clustered naturally, this algorithm led to inefficient results. Large central clusters were identified early in the algorithm, leaving many small marginal clusters scattered throughout the map.

**Agglomerative Clustering:** In agglomerative clustering (AggCluster), each town (or subregion) starts as a region. The two closest regions are then combined into one region. This is repeated, reducing the number of regions until the algorithm is terminated. We terminate the algorithm when the newest region exceeds a maximum diameter, allowing the control that is possible with QT clustering. AggCluster gave superior results to our other options and was eventually implemented in our AI. Figure 1.3.1 illustrates an example region configuration determined by AggCluster.
1.4 Route Selection

After identifying regions it is necessary to determine the routes between those regions. With our two-tier set up we have two route problems to solve: connecting towns within base regions and connecting base regions to each other.

Intra-Region Routes:

This problem is relatively simple. We want each town to be connected directly to its base station with buses. Towns are indirectly connected to the surrounding towns in their region through the bus stops at the base station. This set up is motivated by the fact that transportation to the base station is more important than intra-region transport since efficient transport to the base station leads to efficient inter-region transport, which is both high volume and long distance. In order to minimize the time that it takes for passengers to travel from a town to the base station, we locate base stations at the center of mass of the populations of the regions. Route pathing is done by Charlie’s part of the AI, using an A* pathfinding algorithm for roads. At their terminuses, routes are connected to bus stops placed in towns by Charlie’s code, and bus stops connected directly to the base rail station. These bus stops are built along with, and adjacent to, the rail station.

Inter-Region Routes:

In placing inter-region routes we want to create a long-run overall efficient network and also to have a viable and profitable network at each stage of development. This second constraint turns out to be of great importance, as building profitable routes at the beginning of the game is necessary for the AI company to stay in business and continue developing its network. Due to the importance of this constraint, we implemented a greedy route choosing algorithm – we determined a metric to measure route profitability and, at each step of development, build the inter-region route that maximizes our metric. The metric we chose is “flow”, which is an approximation of the traffic along a route. This traffic is an approximation for revenue, the true value that we are attempting to maximize. Flow between two regions is calculated as:

\[ F_{i,j} = \frac{1}{\min\left[distance(r_i,r_j)\right]} \times \left[ \sqrt{\text{pop}(r_i)} + \sqrt{\text{pop}(r_j)} \right] \]

This metric approximates the number of passengers that can “flow” between \( r_i \) and \( r_j \) over time. The 1/distance parameter accounts for the fact that if \( r_i \) and \( r_j \) are closer, traffic can flow at a higher rate. The populations of the regions are square rooted since, due to the game mechanics, it is more profitable to connect regions with similar population sizes. The concave square root function diminishes the weight of very large regions, causing the AI to avoid connecting regions with extremely disparate populations. The minimum distance between \( r_i \) and \( r_j \) is calculated as the minimum distance traveling along routes in our network to get from \( r_i \) to \( r_j \). If \( r_i \) and \( r_j \) are directly connected, this distance is length(\( R_{ij} \)), where \( R_{ij} \) is the route between the regions. Otherwise, this distance must be calculated using Dijkstra’s shortest path algorithm and updated each
time a new route is added to the network. If \( r_i \) and \( r_j \) are not connected in the network, the distance between them is set to infinity, and so the flow between them is 0.

In order to assess the effects of adding routes to our network, we must look at flow over the entire network, defined as:

\[
F_{\text{Network}} = \sum_{i,j} F_{i,j}
\]

Each route that is added to the network will increase this metric in two ways: It will connect regions that were not previously connected, allowing flow between them. In addition, it will decrease the distances between previously connected regions. The route that most increases \( F_{\text{Network}} \) is added to our graph at each step of the greedy algorithm. The distances and flows are then recalculated and the process repeated. Practically, construction never terminates since the length of time needed to construct a comprehensive network between regions (theoretically a complete graph) exceeds the typical time period of the game. However, it is important to note that a complete graph would have maximum flow over our set of regions. In addition, although such a graph would be very heavily connected, it would be theoretically desirable in the long term, as route construction is a one time fixed cost, while route revenue is generated perpetually after a route is constructed. Increasing flow, even incrementally, is always desirable.

Building up our network using the above greedy algorithm leads to the following advantages:

1) Early routes connect large regions and have high traffic, allowing for profitability despite the lack of a comprehensive transportation network.
2) The network tends to remain connected rather than fragmented into many disjoint components. This is due to the fact that adding a route that connects to a previously built component of the network will increase the flow to and from all regions connected to this component. Although the AI remains flexible to choose to build the graph in disjoint components if this is preferable, it usually maintains a completely connected graph throughout the build process.
3) Route construction is flexible. The physical constraints of the map may at times lead to route failure if a base station cannot be built in a region or a route cannot be built between two regions. Our greedy algorithm does not precalculate a network, so adapting to these failures is possible. A function is implemented that is invoked after a route failure to return the network model to its previous state and calculate a new flow-maximizing route. A single failure or a number of failures will not significantly decrease the overall efficiency of the network designed by the AI.
4) Route calculation occurs throughout construction rather than at the beginning. This avoids a high front end calculation cost and allows calculation to be interlaced with the construction activities of the AI. For example, the AI may calculate the next route to be built while waiting for money to build a new base station.
The logical layout of a reasonably well-developed network is shown in figure 1.4.1. The routes and regions are marked but have not been physically constructed:

2. Physical Implementation of Routes

Although it is possible to develop an abstract plan for our transportation network, we must overlay this plan on a concrete map with various obstructions, limitations, and changing variables. Intra-regional road networks are relatively easy to construct using a simple A* pathfinding algorithm. Inter-regional rail routes however, are significantly more complicated. First, we decided to use a system of double rail lines – with each route having a line traveling from \( r_i \) to \( r_j \) and a parallel line traveling from \( r_j \) to \( r_i \). This set up allows trains to run in a loop between the regions and allows multiple trains to run along a single route without interfering with each other. However, the double track system also makes pathing much more difficult. We modified open source code developed for another AI, TrAIns, which hard-codes in many of the rail structures and configurations needed to build double track routes. This code includes a basic junction building system to allow rail routes to combine so that they can run into a single base station. Although a logical network with many routes running into one region seems relatively simple, a single train station cannot simply accept any number of incoming routes. These routes must combine at the station in an efficient and logical way, with signals being built to control traffic and prevent train collisions and bottlenecks. In order to increase the likelihood of route building success, we implemented various strategies:
1) Since at times base stations may be constructed in non-ideal locations, with one of their two possible exits abutting an obstruction, such as a town, we attempt to construct routes from both station exits. We first choose the exit facing the direction of the desired route. If pathing fails (which is determined by a time-out constant that terminates the path finding algorithm if it takes too long and is unlikely to find a reasonable path) we attempt to find a route from the other exit.

2) We introduced a ‘fudge’ factor to the A* pathfinding algorithm. The A* algorithm evaluates a tree of possible paths by using a (distance + cost) heuristic function, $f(x) = g(x) + h(x)$, where $g(x)$ is the cost to make it to the current node in the path and $h(x)$ is the distance from the current node to the final destination (an estimate of future cost). If the pathfinder times out in attempting to find a path, we set $f(x) = g(x) + \text{fudge} \times h(x)$, with the fudge coefficient specified when calling the pathfinder. This fudge factor causes the algorithm place less of a weight on cost and to favor paths that are closer to reaching the goal, allowing for faster, although more costly pathfinding.

3) The NoAI framework allows an AI to operate in test mode, carrying out operations that do not affect the actual map. This mode allows the AI to determine the costs of various construction actions before they are committed. In order to insure that route construction does not fail due to a lack of necessary funds, we interlace route calculation with partial route construction in test mode. Since route calculation requires a large time cost, this interlacing allows our routes to be built efficiently but also successfully, and to not fail do to lack of construction funds.

4) Since pay rates for passenger transportation are higher for longer distance routes and since it is necessary to generate a large amount of revenue at the outset of the game in order to fund future construction, we place a minimum length criteria on the first two routes we choose. Although this leads to a less ideal network in the long run, it allows the physical construction of routes to get up and running since these initial routes are able to generate large revenues even before being connected to a wider network.

Figure 2.1 An example of a junction set up outside of two regional stations.
3. Overall AI Structure

Due to the choice to divide up the responsibility for AI functions between two programmers, developing an overall AI structure that could integrate our portions of the code proved difficult. The most important part of the overall program structure is that it schedules tasks for the AI to complete in a coherent and efficient order. In addition, the AI must manage its finances, a key part of the OpenTTD game. Each vehicle and piece of infrastructure built is associated with a cost and the AI must insure that it generates enough revenue to continue building its transportation network. There is no in game stop to overspending, so if the AI attempts to build infrastructure without the requisite cash, the construction will simply fail, leaving us with an incomplete and fragmented network. If the AI builds a route but then fails to build a train along it because it cannot afford the new vehicle, then the route will be wasted and will incur property taxes without earning revenue.

In order to have a flexible program structure, we decided to have the main backbone of the AI be an infinite loop. Although immediately a task queue would seem to be a logical backbone for the AI, due to the repetitive nature of many of the tasks that the AI completes, a loop was preferable. Each task that the AI is responsible for - managing money, building new routes, or updating vehicles along routes - is assigned a “repeat value” for the loop. This value dictates how often a task will be completed. For example, the AI will manage funds in each cycle of the loop, will update vehicles every other cycle, and will attempt to construct a new route every third cycle.

In addition to the main loop, an important part of the overall AI structure was a money management function. It is important that the AI not run low on funds during the execution of atomic functions such as station construction and route finding. A money management loop was implemented that is called before each atomic construction function. This loop checks if the AI has enough money in the bank to complete the construction. If not, it attempts to take out a loan or waits until incoming revenues replenish the bank balance to the necessary level. Unfortunately, since this function was implemented after the two portions of the AI were completed, it is not optimized. It often has no way of checking how much a construction action will cost so has to use rough cost estimates hard coded into the program in place of actual cost functions. Ideally, a comprehensive money management system would be integrated into each construction function, with the AI entering test mode to calculate actual costs and ensuring the existence of sufficient funds during each game defined atomic construction function (the construction of a single piece of track, a single vehicle, or a single piece of station infrastructure), rather than during each AI defined atomic construction function (the construction of a route, an entire regional station, or a group of vehicles to service a region).
4. Improvements and Intellectual Value of Project

4.1 Improvements to ZooElite and OpenTTD

During the course of developing and testing our AI we came upon many improvements that we would like to eventually implement. In addition, in working with the OpenTTD platform we found many ways in which the platform and game mechanics could be altered in order to better simulate real world transportation systems.

*Improvements to Our AI:*
- Find a balance between the flexibility and heuristic benefits of greedy route selection and the advantages of preplanning a network. Possibly implement a “look ahead” strategy in which routes are selected greedily but with some forward looking effects of selection considered.
- Actually implement a recursive multilevel region finding algorithm. Although never necessary with the size maps we were working on, it would be interesting to actually implement a multilevel regional network. This would require determining how super-regions should be connected (from their centers, their closest sub-regions, etc.) and how the structure of the network in a sub-region should be altered in order to fit more efficiently into the higher level regional set up.
- As mentioned above, implement a better money management system.
- Although ambitious, possibly implement a route finding method in which a final destination is not rigidly specified. In our AI, regional stations are constructed and then connected with rail routes found by a pathfinder. These stations may be located in less than ideal locations for pathfinding, for example, abutting cities, adjacent to cliffs or water, or simply near a high number of obstructions. Implementing a pathfinder that could place the station as it finds a route to a region would solve these problems.
- Implement a pathfinder with a higher level view of the map. The pathfinder currently considers routes tile by tile, and could be made much more efficient if it was able to consider higher level geographic “chunks.”

*Improvements to OpenTTD:*
- A more realistic pay rate model that does not base pay rates simply off of distance, but rather on an approximation of market demand. This would allow the goals of creating a profitable AI in the game and a realistic transportation network to converge.
- An improved passenger distribution algorithm. Although the CargoDest patch to OpenTTD (which we used while developing our AI) makes a great step towards realistic passenger distribution, with passengers having specific destinations, the distribution system could be improved. Currently, demand for destinations is based off of destination population and network connectivity between the source and destination. These factors could be expanded to include distance (with a higher demand to travel to nearby locations) and other factors, such as the locations of industries. In addition, making approximate demand functions available to the AI platform would allow the AI to better incorporate demand modeling into network planning so that a truly efficient network could be constructed.
4.2 Intellectual Value of Project

The original goal of this project was to explore transportation networks by developing a program that could plan and build them from the ground up. Throughout the project we came upon many issues and challenges that led us to think about various issues in transportation network design and implementation. Some of these issues that I was specifically interested in and maybe will choose to explore in the future are:

1. Why is it so much easier for an AI to perform certain abstract tasks that are difficult for human players, such as large-scale network planning, but so difficult to perform tasks that are relatively simple for human players, such as pathfinding, station placement, and station construction? Although the answer to this question may seem obvious – the later tasks are complicated by many small, random factors that are difficult to deal with in a computer program - it is interesting to think about why these tasks are relatively easy for humans. Is it that we rely on our physical and visual intuitions from performing tasks outside of playing the game? How could these intuitions be programmed into an improved AI program? Would the best AI program for OpenTTD also be able to play similar strategy games, or even perform other, seemingly unrelated physical tasks?

2. Is there a theoretical underpinning to the efficiency of the region-based network? I spent a lot of time during this project looking at graph theory algorithms and theorems and much of this work focuses on networks with less degrees of freedom than the network we designed with our AI. How can you define efficiency for a graph with completely free node and edge placement (especially since a complete graph, with every node connected to every other node would seem to the most efficient graph possible but is usually not feasible and rarely implemented in the real world.) Would there be a theoretically optimal way to go about constructing a transportation network as we did?
Appendix: Outline of Major Pieces of Code

Below is an outline of the major pieces of code that make up the AI.

‘Higher Level’ Code:

**Main.nut**
Start() – calls the loop function that runs the AI. Deals with user defined game options and defines various global variables that are used throughout the AI.

**Looper.nut**
Loop() – runs an infinite loop that schedules tasks for the AI to complete.

**RouteChooser.nut**
Contructor() – takes as input a set of regions for the map and calculates all possible routes and initial flow improvements.
getNextRoute() – greedily selects the next route of the network to build. Updates the graph that represents the network to reflect the new construction.
unGetRoute() – if a route cannot be built physically, this function resets the network so that the AI may choose an alternative route.

**RoutePlanner.nut**
builtRegions() – uses agglomerative clustering to choose the regions of the map that we will use for our network.

Other methods: implements all necessary utility methods for RouteChoose.nut, including the clustering algorithm, Dijkstra’s minimum path algorithm, and various algorithms for updating flow.

**Helper.nut**
Implementes various helper functions for the AI, including many used in the ‘lower level’ of the AI. Implements the Money management loop, GetMoney().

/Objects
**Station.nut** – station class that stores station information including location, connected routes, and serviced cities.
**Town.nut** – town class that stores town information including bus stops, bus routes, and the regional station the town is connected to.
**Route.nut** – route class that stores route information including connected stations, the physical path of the route, and vehicles running along the route.

/Rail
This folder includes a large portion of the raw code for the project. Files adapted from the open source TrAIns AI implement: the A* path finding algorithm for rail lines, actual rail line construction, rail junction finding and construction, signal construction for rail lines, and depot construction for rail lines. Files original to our AI include:
Finder.nut – mostly the responsibility of Charlie, implements station construction. Also includes functions that combine the lower level and higher level of the AIs including:

- **BuildBaseStation()** - takes a region as input, and builds a regional train station in the region.
- **ConnectBaseStation()** – takes a regional station as input and connects the towns in the region to the station.

‘Lower Level’ Code:

**Road/**

Files which relate to the building of bus stations, purchasing busses within a city, and the routing of busses.

- **Placer.nut** - contains the functions that locate the optimal station configuration.
- **Builder.nut** - responsible for building the stations, handling errors, and building other support infrastructure, such as road connections, vehicle depots, etc.
- **Vehicle.nut** contains all the bus building and routing methods.

Finder.nut – implements all rail station construction.

**Appendix: Useful Resources**

- NoAI Homepage: [http://noai.openttd.org/](http://noai.openttd.org/)
- NoAI API Documents: [http://noai.openttd.org/docs/trunk/](http://noai.openttd.org/docs/trunk/)
- Info on CargoDest Mod: [http://wiki.openttd.org/Passenger_and_cargo_distribution](http://wiki.openttd.org/Passenger_and_cargo_distribution)