Lincoln University Digital Dissertation

Copyright Statement

Optimising locomotive requirements for a pre-planned train schedule

---

A dissertation
submitted in partial fulfilment
of the requirements for the degree of

Bachelor of Applied Computing with Honours

at

Lincoln University

by

Ray Hidayat

Lincoln University

2005

Abstract

Every rail operator wishes to minimise the size of their locomotive fleet in order to reduce costs. This minimum fleet size problem requires a rail operator to allocate locomotives to the trains in a predefined train schedule so that the total number of locomotives required is minimised. The key to this is deciding how and when to transfer locomotives to where they can be better utilised. The rail operator for this hypothetical problem runs approximately 7,200 trains per week involving movements between 780 locations. An integer programming formulation was developed based on the work by Ahuja, Liu, Orlin, Sharma and Shughart (2002)[1] and a solver applied this formulation to a train schedule to find the optimal solution. As the solution process was highly computationally intensive, the largest partial train schedule that was able to be solved by the integer programming solver was 21% of the size of the full train schedule, taking 2½ hours to converge on the optimal solution. An alternative algorithm, called the work unit levels algorithm, was developed. This algorithm schedules locomotives by identifying all valid ways to transfer locomotives between trains, then allocating the train schedule in an order dependent on the possible interconnections between trains. When this algorithm was applied to the largest partial train schedule that could be solved by the integer programming solver, it arrived at a similar solution in 6 seconds. The algorithm took 13 minutes to solve the full problem.

[1] Ahuja, R. K., Liu, J., Orlin, J. B., Sharma, D. & Shughart, L. A. (2002). *Solving real-life locomotive scheduling problems*. Gainesville, FL: University of Florida.

Acknowledgements

I remember myself as a primary school boy with a profound curiosity as to what made things move on the computer screen. I fiddled unsuccessfully for countless hours with Microsoft's QuickBasic programming language trying to teach myself programming. When I finally created my first program – a colourful, ever-changing abstract drawing that looked like a screensaver – I felt like I was flying to the moon. Since then, my enthusiasm, excitement and passion for software development has grown unstoppably.

I owe a special thanks to Dr Lynette Hardie-Wills from the Regional Education of Lincoln University, who took the risk of admitting me to her regional studies program years before I would sit the exams for the University Entrance qualification.

I would also like to thank Arjan van Hasselt, Jade Software Corporation, who recruited me to be a Jade Kid in the year 2000. This started my journey into their programming language, JADE, which has shaped a large part of my life so far and will continue to do so.

I wish to express my gratitude for the excellent guidance of my supervisor, Professor Alan McKinnon, during the course of this honours project. I know that there would be so much missing from this project had it not been for Alan, and in particular, I now have first-hand experience about the importance of literature reviews thanks to him.

I would like to acknowledge with deep gratitude the collaboration of Jade Software Corporation in this project, and I would especially like to acknowledge Dean Cooper, Colin Dixon and Roger Jarquin for providing me with this magnificent window of opportunity to carry out research into the ever-challenging train scheduling problem that is faced by every rail operator.

My thanks are extended to the entire academic staff of the Applied Computing Division at Lincoln University for providing the structure in which I have developed my skills in computing. Even though I have been very much a person who prefers being self-taught, there is knowledge I would never have discovered and fields I would never have looked at if I had not been a part of the Applied Computing Division at Lincoln University.

Lastly, I wish to express my deepest gratitude to my parents, Rudy and Rini Hidajat, for their never ending love and support, and for giving me the freedom to follow my own interests in pursuing my career.

Table of Contents

1. Introduction

With railway being such a fast method to transport large loads of passengers or freight, it is no surprise that rail networks have become a supporting pillar to many of the economies of the world. Rail transport has some major advantages over other methods of land transportation. Railway is the fastest and most energy-efficient method of mechanised land transport, and it is also very safe (Docherty & Shaw, 2003, p. 108). The minimal friction of the train tracks combined with the safety of rail transport enables some trains to travel in excess of 200 km/h to their destination with extremely heavy loads. This makes railway the most effective way to connect towns and cities in dispersed regions that are found in countries such as China and in particular states of the USA.

Railway is also the most space-efficient form of land transport, with a two-rail track being able to carry more commuters and cargo within a given time than a four-laned road (Docherty & Shaw, 2003, p. 108). This is a key reason as to why rail networks are used so often in densely-populated cities such as Tokyo and New York, where congestion found on roads has forced many commuters to make the use of subways part of their lifestyle.

Being so efficient in many ways, in many countries, rail transportation has become closely integrated with ship transport and the use of railway has also become especially prominent in the transportation of coal.

Even with its distinct advantages, trains are not always the most commercially viable means of transport however. In recent times, highways and commercial airlines have replaced much of the need for trains. Rail networks in many areas struggle to achieve profitability and some rely heavily on government funding to survive. For railway to maintain its competitive edge against other technologies, cost reductions are absolutely necessary. With each locomotive costing at least 1.8 million US dollars (Ahuja, Liu, Orlin, Sharma & Shughart, 2002), being able to utilise a rail company's fleet of locomotives more efficiently could induce substantial reductions in the costs incurred by rail companies.

Based on their experience with resource scheduling problems, Jade Software Corporation[2] has constructed a hypothetical train scheduling problem that encapsulates the core challenges that typical rail companies face when deciding how to utilise their locomotive fleet to run their train schedules. The source data for this problem has been extracted from a real system, involving approximately 7,200 trains travelling between 780 locations each week. The objective of this project is to develop an allocation algorithm that allocates locomotives to the trains for a given week. The algorithm should come as close as possible to the minimum number of locomotives within a reasonable amount of running time.

[2] Jade Software Corporation, 19 Sheffield Crescent, PO Box 20 152, Christchurch 8005, New Zealand.
Email: askjade@jadeworld.com. Web: www.jadeworld.com.

2. Description of the Problem

In essence, this problem is a minimum fleet size problem (Bertossi, Carraresi & Gallo, 1987, as cited in Erlebach, Gantenbein, Hürlimann, Neyer, Pagourtzis, Penna, Schlude, Steinhöfel, Taylor & Widmayer, 2001, p. 4), as the goal is to identify a way to run the same set of trains with the smallest fleet possible.

One of the main concepts in this problem is a **consist**. A consist is the name given to one unit that travels through the rail network which is comprised of a number of locomotives and any wagons they pull. Each week, the rail operator has a train schedule that contains the set of trains that have been planned to be run. The word **train** refers to one planned movement of a consist, departing from a specific departure location at a given departure time to a specified arrival location at a given arrival time. The rail operator will have customers who have booked seats or space for freight on particular trains ahead of time, and so it is necessary that the rail operator runs all the trains it has planned to run in a particular week.

The rail operator needs to plan which locomotive or locomotives will run each train. A locomotive can run multiple trains in sequence – once a locomotive has finished one train, it can move on to run the next one. The goal is to use the fewest number of locomotives to run the train schedule.

The main difficulty with the problem is not deciding how to allocate the locomotives to the trains in the schedule, but how to transfer locomotives so that each locomotive can run more trains. If each locomotive runs more trains per week, the full set of trains planned for the week can be run with fewer locomotives. The trouble is that transferring locomotives to where they are needed is not a straightforward process.

Since the tracks on which the locomotive must move are owned by a national company and shared with other rail operators, the rail operator in this problem can only move a locomotive if it has what is called a path on which the locomotive can be transferred. A **path** is a right that is acquired from the national track owner to be able to send one consist from a specific departure location at a given departure time to a specified arrival location at a given arrival time. The rail operator cannot move locomotives from one location to another if it does not have a path to send the locomotives on, as it does not have the right to do so. With other rail operators using the same set of tracks, travelling on the tracks without clearance in the form of a path from the national track owner is very dangerous.

For each of the trains in the train schedule for a particular week, the rail operator will own a path on which the consist for that train can move on. Also in any week, the rail operator will own paths that it is not using for the trains in the train schedule. These paths enable locomotives to be transferred between locations and moved to where they are needed. There are also several other ways the rail operator can move locomotives to different locations if it needs to. These are explained in detail in section 2.3.

Transferring locomotives can allow one locomotive to take on more trains in the same week, which in turn means that the total set of trains in the week can be run with fewer locomotives. The heart of this problem is knowing when and where to transfer a locomotive. The various aspects of this problem will now be expressed in more detail.

2.1. Consists

The word consist refers to a set of locomotives, along with any wagons they pull, that travel together as a unit through the rail network. All consists must have a locomotive in order to be able to move, but not all consists have wagons. A locomotive that is travelling without any wagons is known as a **light engine**.

This problem only considers locomotives; wagons are beyond the scope of this problem. The assumption is that once the locomotives have been arranged to run all the trains in the train schedule, the wagons can be organized in another stage.

2.2. Locations

The point of having consists is to move them to and from locations. A location is a place where a consist can stop and the locomotives or wagons that comprise the consist can be changed. Drivers, passengers or freight on the consist may also be changed when a consist has stopped at a location.

In reality, locomotives have home depots. A home depot is a place where a locomotive should both begin and end its week. For this problem, home depots will not be considered. That is, there are no restrictions on where a locomotive must begin and end its week.

2.3. Movements

In this problem, multiple rail operators send consists on the same set of tracks. The tracks are owned by a national company, and to maintain safety between the various rail operators, the national track owner must verify that each individual rail operator's plans does not clash with the plans of any other rail operator. To make this possible, whenever a rail operator intends to move a consist, the rail operator must have acquired the right to do so from the national track owner.

Rail operators acquire rights to move consists by bidding for the ownership of **paths**. When the national track owner makes a path available to a rail operator, it will have ensured that travelling on that path is safe, given how the other rail operators will use the tracks. As there are other rail operators using the same set of tracks, it is dangerous to move a consist without a path.

There is one special case though. When travelling between close locations, the controllers at the locations involved can wait until the tracks between them are clear and then send a consist on the track without a path. This is called a **location group movement** and will be explained in more detail in section 2.3.2.

2.3.1. Paths

Paths are the primary type of movement that is used. The word **path** refers to a right that is acquired from the national track owner to be able send a consist from a specific departure location at a given departure time to a specified arrival location at a given arrival time.

The rail operator will own paths for each train in the train schedule so that it can meet agreements it has made with customers to run the trains. It will also own paths that are not planned to be used for trains – these are called **unused paths**. These paths can be used to transfer locomotives to where they are needed during the week.

2.3.1.1. VSTPs

VSTP stands for very short term plan. A VSTP is a special path that a rail operator acquires the rights to use only days or hours before the departure time of the path.

When allocating locomotives to the weekly schedule, the rail operator may notice that in a prior week, it used a path that could be useful to transfer a locomotive this week. In this situation, the rail operator may put in a request for a VSTP from the national track owner to acquire that same path for this week. Since the rail operator has not purchased the path ahead of time, there is a risk that the national track owner will not be able to give the rail operator the path it has requested as another rail operator may have acquired a path that clashes with the requested path. That is why in reality, VSTPs are added to the schedule only when they are needed, as they are not always a reliable option. However, this particular problem does not make a distinction between VSTP paths and other paths, all of the paths used in past weeks are assumed to be available to transfer locomotives this week.

2.3.2. Location group movements

When moving between close locations, a locomotive is sometimes able to move without a path. This is called a **location group movement**. For these movements, the controllers at the involved locations just make sure the tracks between them are clear before sending locomotives. Given enough time to wait for the tracks to clear, this is a reliable way to move locomotives and is used quite often.

Location group movements do not have a departure time and arrival time – they are able to occur at any time. So that location group movements can be treated the same as paths, an approximation has been made for this

problem to make it so that location group movements do have departure times and arrival times. For each train in a weekly schedule, one location group movement is created from each close location to the departure location of the train. The location group movement is set so that its arrival time is just before the departure time of the train, and its running time is set to thirty minutes. This enables location group movements to be treated the same as paths.

2.3.3. Dead-in-train

To transfer a locomotive to where it is needed, a locomotive can also "hitch a ride" with another locomotive. In this situation, the locomotive just turns off its engine and is pulled by another locomotive on an existing path. This is called dead-in-train. Dead-in-train can be used to transfer extra locomotives on paths used by trains as well as unused paths.

2.4. Locomotive classes

There are different classes of locomotives that can be used to run the train schedule. In this problem, any locomotive of any class can be transferred on any path or location group movement – there are no restrictions on locomotive classes for transfers. Trains can only be pulled by locomotives of one class, but any class of locomotives can be transferred on a train when using dead-in-train. Trains and their restrictions are explained more in section 2.5.1.

2.5. Train schedule

The entire problem begins from a **train schedule** for a particular week. The train schedule is the input to the problem, it is predefined and unchangeable. Train schedules are made of up paths. Some of these paths will be used by the rail operator to run trains, and all other paths are available to be used for transferring locomotives. Recall from the start of section 2 that a locomotive cannot move unless there is a path on which it can move, and each path has a specific departure location and departure time as well as a specific arrival location and arrival time.

The objective of the system is to take this train schedule and to allocate locomotives to the paths in the schedule in a way that will allow the rail operator to run all of its required trains with as few locomotives as possible. All of this should be executed in reasonable time.

2.5.1. Trains

A **train** is a commitment that the rail operator has made with a customer to move a consist carrying the customer's cargo or commuters from one specific departure location at a given departure time to a specified arrival location at a given arrival time. In this problem, the rail operator cannot cancel trains – all trains must

be run. The rail operator will have acquired paths from the national track owner ahead of time so that all the trains it plans for can be run.

Each train in the train schedule also has information about the number of locomotives required to run the train and what class the locomotives must be. If the train requires more than one locomotive to pull it, all the locomotives must be of the same class. Only that required class of locomotives can be used to pull the train.

2.5.2. Transfers

To be able to run the trains in the train schedule with fewer locomotives, sometimes the rail operator will transfer locomotives to locations where they are needed. Not all of the paths in the train schedule will be used for trains. These extra non-train paths are called unused paths. Unused paths can be used to transfer locomotives to where they can be used more effectively. As well as paths, there are also a few other methods of transferring a locomotive to where it is needed – location group movements and dead-in-train. These were outlined in section 2.1.

A locomotive may sometimes perform a multi-stage transfer, which is when it travels on multiple paths in succession to arrive at the location where it is needed. This is opposed to a single-stage transfer where the locomotive only travels on one path to arrive at the location where it is needed.

2.6. Locomotive diagrams

Once the train schedule has been allocated, each locomotive used in the week will have a locomotive diagram. A locomotive diagram is the sequence of all train and transfer movements that will be run during the week by one locomotive. The rail operator can tell where a particular locomotive is meant to be at any point during the week by looking at its locomotive diagram.

Naturally, each locomotive diagram will contain idling time for the locomotive. This is called **whitespace** in rail terminology. Although whitespace indicates that the locomotive is not being used for productive purposes, whitespace is important to make a schedule flexible and tolerant to situations where the week does not go exactly as planned.

2.7. Logical locomotives

Each of the locomotive diagrams that result from the allocation process do not relate to a specific actual locomotive. For example, one locomotive diagram may tell the rail operator that it requires one locomotive of class 234 to start at location A, from which it will follow a particular sequence of movements during the week according to the locomotive diagram. Any locomotive that is of class 234 and can be at location A at

the start of the week could be used to run that sequence. There is no specific actual locomotive that must be used.

For this reason, when the algorithm outputs which locomotive will run which train, the locomotives that have been allocated are actually called logical locomotives. Logical locomotives exist for the scheduling process, but another process later on must map each logical locomotive to an actual locomotive.

Using the set of logical locomotives that are output from the allocation process, rail operators can see what locomotives they need to run a particular week, not just whether the train schedule of a week is feasible or not. If there are more logical locomotives in the allocated schedule than the number of actual locomotives the rail operator owns, then the rail operator could conclude that the train schedule cannot be run. This gives the rail operator a better indication of how close to feasible or infeasible the train schedule they have planned is, and also gives a measure of how many spare locomotives they will have for backup purposes during the week.

2.8. Problem summary

The problem we are faced with is a minimum fleet size problem (Bertossi *et al*, 1987, as cited in Erlebach *et al*, 2001, p. 4). We begin with a train schedule that contains many paths. Some of the paths are planned to be used to run trains, and the other unused paths can be used for locomotive transfers. The rail operator must plan which locomotives will move on which paths so that all trains are allocated with their required class and quantity of locomotives.

To save costs, the rail operator desires to run all of its trains with as few locomotives as possible, and to be able to do this, the rail operator will try to transfer locomotives to where they are needed. Locomotives can be transferred a number of ways: using dead-in-train, on the unused paths in the train schedule, through the use of location group movements or by using a VSTP.

The objective of this problem is to develop an algorithm that takes a train schedule as input and produces a solution which describes how to use a fleet of logical locomotives to run the train schedule using the fewest number of locomotives possible.

3. Related Work

The minimum fleet size problem is by no means a new problem, and has been heavily researched due to the many commercial benefits that solutions can provide. This chapter explores some of the important problems that have been solved and how the solutions relate our problem.

3.1. Knowledge-based locomotive planning

Scholz (1998) investigated locomotive scheduling for the Swedish railway system. Scholz's problem involved a set of trips that had to be run by locomotives, and the objective was to run the same set of trips with as few locomotives as possible. Every trip had a specified start location, end location and total travel time required, but the trips were not given specific departure times. Instead, each trip had a departure time window, and the trips had to depart at some point during that time window.

After representing how the trips fitted into the schedule in a Gantt chart format, the problem was seen to be similar to a bin packing problem with additional constraints. The Gantt chart in this problem was displayed with each logical locomotive on its vertical axis against time on its horizontal axis. Each trip forms a rectangle of a fixed size in the Gantt chart based on how long the trip is, and so to efficiently plan how to use the locomotives to run the trips in the schedule, one must rearrange the rectangles of the Gantt chart so that as little space as possible is taken along the vertical axis – a bin packing problem.

The Swedish railway system makes room for transfers to occur in the form of passive transfers. The time taken to perform a passive transfer had to be included as part of the allocated schedule, but there was no restriction on when or where passive transfers could be used. Also as part of this problem, when assigning trips to occur at different times during the day, the system also had to consider the locomotives as they travelled on the tracks in the network, making sure there was enough distance between the locomotives on the track, as well as making sure that locomotives did not collide with each other on single-laned tracks by directing them in the same direction or scheduling locomotives to pause at points where other locomotives heading towards them could pass safely. Scholz's solver also had to choose the route that a locomotive could take to get from a trip's start location to its end location.

Scholz devised a multi-stage approach to solving the problem. First, trips that are found to be optimally run together are combined and treated as single trips. This process was called the matching heuristic. Second, a constraint propagation algorithm defines a reduced search space in which a number of heuristics are used to find one good solution to the schedule. Finally, a neighbourhood search improves the solution to a local optimum.

The primary differences between this problem that Scholz investigated and our problem arise from the fundamental difference that Scholz's system simultaneously solves track allocation and locomotive allocation, whereas our problem solely concentrates on locomotive allocation. For that reason, Scholz's solver faces complexities that are not present in our problem, such as choosing routes for trips, and making sure that locomotives do not collide on single-laned tracks. Scholz's problem does have similarities to our problem though. A train in our problem could be considered to be a special case of a trip, where the departure time window is reduced to just one particular point in time, and the travel time of the trip is fixed. However, the major incompatibility of Scholz's problem to ours is the transfers in our problem are much more constrained and need special attention.

3.2. Multiple Vehicle Depot Scheduling Problem

Loebel (1998) researched what he called the Multiple Vehicle Depot Scheduling Problem (MVDSP). The MVDSP was designed in the context of bus scheduling but the problem exhibits many similar features to train scheduling problems. Provided with a predefined set of timetabled trips, the problem was to efficiently allocate vehicles to the trips. Vehicles start their route from a home depot, and must return to their home depot after they have completed their set of trips. To solve the problem, Loebel formulated it into a multicommodity flow network, which is a network of arcs and nodes on which vehicles (commodities) flow, with an additional constraint that there are different types of vehicles, and some arcs can only accept vehicles of a particular type. The flow network was then solved using Lagrangian relaxation techniques. Loebel's solution was able to handle timetables with up to 49 depots and 25,000 timetabled trips.

In the MVDSP, it is possible to transfer vehicles between trips. Arcs are added into the flow network to represent possible transfers. Although our problem does not involve home depots as in Loebel's problem, the idea of a flow network does apply to this problem, as will be seen in section 4.

3.3. Advanced Locomotive Scheduling System

Ahuja *et al* (2002) faced a train scheduling problem where locomotives must be allocated to a predefined set of timetabled train movements. In their problem, Ahuja *et al* (2002) considered several key constraints:

- Each train must be allocated enough locomotives to fulfil tonnage-pulling requirements.
- Trains can be run by a range of locomotive classes.
- The resulting schedule must be repeatable – all locomotives must end the week at the location where they started the week.

This problem was formulated into a mixed-integer programming model and solved in multiple stages. The stages included the use of linear relaxations, residual flow networks and a neighbourhood search algorithm.

Applying their approach to the train schedules of CSX transportation, a total of 400 locomotives were saved from the existing system used by CSX transportation, which translated to savings of one hundred million dollars annually.

Ahuja *et al* (2002) had a problem very similar to our problem, but their problem involved additional complexity. In our problem, each train has a specific number of required locomotives of one class, while Ahuja *et al* (2002) considered each train to have a range of different locomotive classes that could be used to pull the trains, and each locomotive class had a different tonnage-pulling capacity so different numbers of locomotives may be required to run the same train depending on which classes of locomotives were chosen. Also, each location in their problem had to begin the week with the same number of locomotives as it started with, so the schedule would be repeatable, which is different from our problem.

The mixed-integer programming formulation Ahuja *et al* (2002) used was based on a flow network they called a space-time network. The space-time network summarised all the possible movements locomotives could do through space as well as through time during each week. Based on the space-time network created by Ahuja *et al* (2002), a mathematical formulation of our problem was defined. The formulation created for our problem is described in section 4.

4. Integer programming formulation

The mathematical formulation for this problem is based on the formulation described by Ahuja *et al* (2002). However, the formulation for this problem is a lot simpler than the Ahuja *et al* formulation, as this problem does not require schedules that can be repeated week after week to be produced, and it also does not include several other more complicated features such as multiple locomotive classes being able to run the same trains.

Using this mathematical formulation, any train schedule can be converted into a mathematical description and solved using an integer programming solver. As integer programming solvers can solve such a wide variety of problems, a large quantity of research has been done to develop and improve the algorithms that integer programming solvers use, and so often the solutions they produce are optimal or near-optimal (Fisher, 2004). Some solvers guarantee optimality, and others are able to report whether the optimum has been reached or not. Another reason for constructing this problem into an integer programming formulation is to define the problem using precise mathematical terminology. Having an accurate definition of the problem is essential to the process of finding an effective solution.

4.1. Space-time network

The expression of the locomotive scheduling problem in mathematical terms first involves the construction of a **space-time network**, which summarises all the movements that locomotives can perform through space as well as through time. An example space-time network is shown in Figure 4-1. The space-time network is a kind of multicommodity flow network (Loebel, 1998) made up of nodes and arcs, and is used in many resourcing problems. Each train is represented in the space-time network as a train arc, and the possible transfer options that are available to be used in the train schedule are represented as transfer arcs. In the space-time network, locomotives are created on train arcs when they are needed and the locomotives travel through space and time by flowing down other arcs in the network to reach the end of schedule.

Once this space-time network is constructed, a mathematical description of the train schedule can be built and fed into an integer programming solver. The process of creating this space-time network will now be explained.

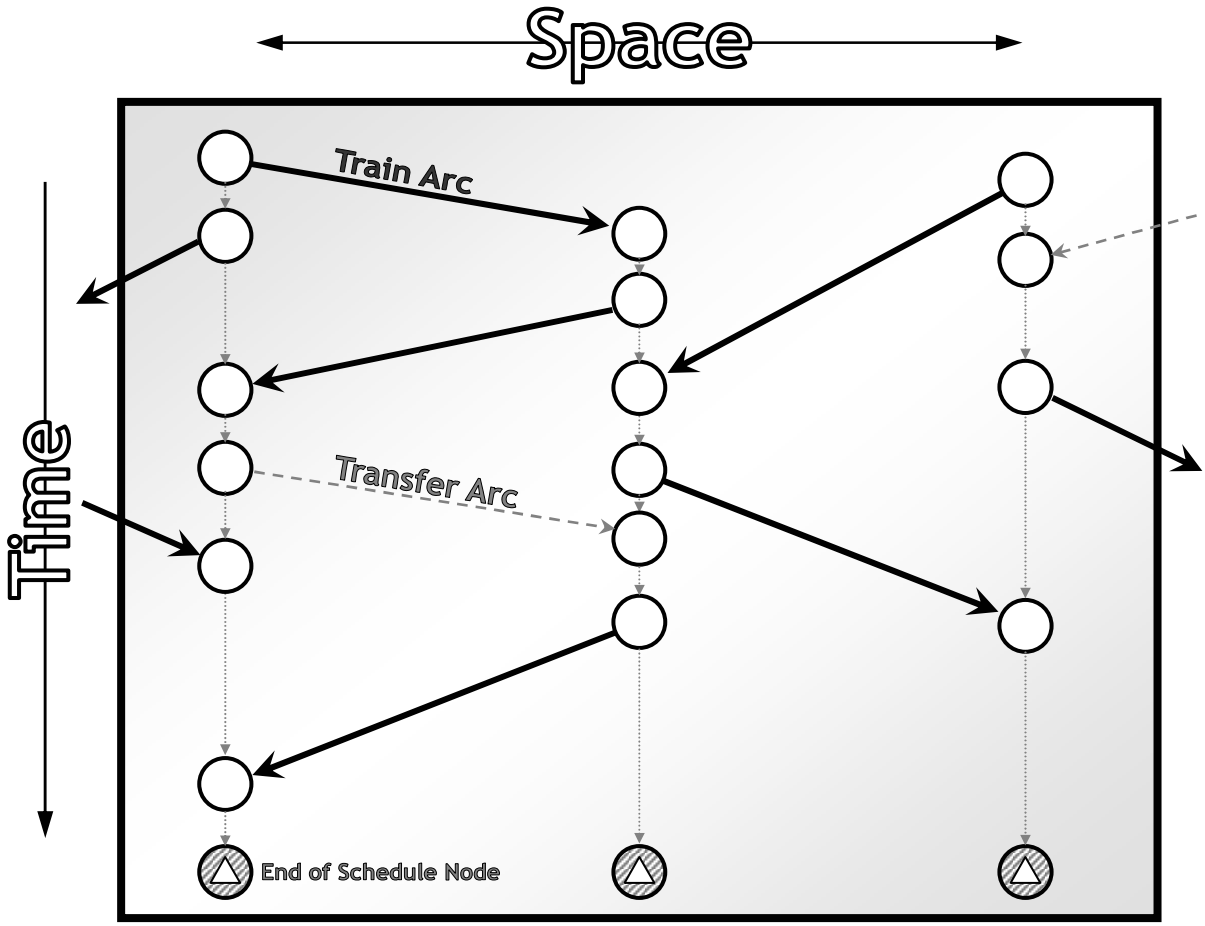


Figure 4-1: An example space-time network. The arcs that cross the borders of the bounding box represent arcs that connect to nodes not displayed in this figure.

First, one **train arc** is created for each train that is scheduled to begin during the scheduled week. At the tail end of each train arc, a **train departure node** is created, and at the head end of each train arc, a **train arrival node** is created. During the construction of the network, both the train departure nodes and train arrival nodes are associated with the location they arrive at or depart from, and the time at which the arrival or departure occurs. This information will be used later on.

Next, one **transfer arc** is created for each unused path, VSTP and location group movement that is available to transfer locomotives during the week. Remember location group movements have start times and end times in this problem the same way paths do (see section 2.3.2). Similar to the train arcs, at the tail end of each transfer arc, a **transfer departure node** is created, and at the head end of each transfer arc, a **transfer arrival node** is created. The nodes are associated with the location and time at which they occur.

In the next stage, each location and its associated nodes are considered in turn. Within each set of nodes associated with the same location, the node with the earliest time is taken and connected to the next earliest node using a **connection arc**. Then the second earliest node is connected to the third earliest node using a

connection arc, the third earliest node is connected to the fourth earliest node, and so on. For the node with the latest time for the location, an **end of schedule node** is created and a connection arc created to connect the node with the latest time to that end of schedule node.

The result is a space-time network that looks similar to the one illustrated in Figure 4-1. In the diagram, the horizontal dimension represents the space dimension and the vertical dimension represents time. The three vertically-aligned sets of nodes correspond to three locations in the network. Building on what has been described in section 2, the train schedule is made up of train arcs, and every train arc needs to have a specific number of locomotives of a particular class allocated to it. The transfer arcs however just represent possibilities for moving a locomotive from one location to another. The connection arcs represent locomotives idling at the same location and allow the locomotives to move from one point in time to the next without changing their location.

4.2. Mathematical representation

Once the space-time network has been constructed, it is converted into a mathematical representation – an objective function and a number of constraints. From there, a mathematical programming tool can be used to solve the space-time network.

4.2.1. Parameters

These parameters will serve as the input to the solver. They need to be specified to define the problem to be solved. At this stage, the space-time network will already have been constructed. The space-time network is used as a parameter to the solver, and so some of the parameters have been inferred from the space-time network instead of taken directly from the train schedule.

Parameter Name	Description
K	Set of all locomotive classes. Known from the train schedule.
AllArcs	Set of all arcs in the space-time network. Inferred from space-time network.
TrainArcs	Set of all train arcs in the space-time network. Train arcs are constructed from the train schedule, but this parameter is inferred from the space-time network. Train arcs are made up of the paths used for trains in the train schedule.

Parameter Name	Description
r_l^k	The number of required active locomotives of class *k* on a particular train arc *l*, where $k \in K, l \in TrainArcs$. Active locomotives are the locomotives that have their engine on and are actively pulling the train. Every train arc has a specific number of locomotives that it requires. Known from the train schedule.
TransferArcs	Set of all transfer arcs in the space-time network. Transfer arcs originate from the train schedule, but this parameter is inferred from the space-time network. A transfer arc indicates there is a possibility of using an unused path, VSTP or location group movement to move locomotives from one location to another.
ConnectionArcs	Set of all connection arcs in the space-time network. Connection arcs are generated solely for the space-time network – they do not have a direct counterpart in the train schedule.
S	Set of all connection arcs that connect to an end of schedule node. These are used to count the number of locomotives used in the week. Inferred from space-time network.
AllNodes	Set of all nodes in the space-time network. Inferred from space-time network.
TrDepartureNodes	Set of all train departure nodes in the space-time network. Train departure nodes may have more locomotives flowing out of them than locomotives flowing in – implying locomotives can be created at train departure nodes. Train departure nodes are the source nodes of the network. Inferred from space-time network.
EOSNodes	Set of all end of schedule nodes in the space-time network. End of schedule nodes may have many locomotives flowing into them but they always have no locomotives flowing out. This makes them the sinks of the network. Inferred from space-time network.
BalancedNodes	Set of all nodes that must have an equal number of locomotives flowing in as the number of locomotives flowing out. Defined as $BalancedNodes = AllNodes - TrDepartureNodes - EOSNodes$. Inferred from space time network.
I[*i*]	Set of all incoming arcs to node *i*, where $i \in AllNodes$. Inferred from space-time network.
O[*i*]	Set of all arcs emanating from node *i*, where $i \in AllNodes$. Inferred from space-time network.

Table 4-1

4.2.2. Variables

The variables are the output of the formulation. During the solving phase, the goal is to find values of these variables that will result in the optimal solution. For this allocation problem, there are two variables – the number of active locomotives and the number of inactive locomotives on each arc.

Variable	Description
x_l^k	This variable indicates the number of active locomotives of class *k* on a particular arc *l*, where $k \in K, l \in AllArcs$. Active locomotives are defined as locomotives that have their engine on. To move a consist on either a train arc or a transfer arc, at least one of its locomotives must be active. All train arcs have a required number of active locomotives (see the parameter r_l^k) and transfer arcs must have one active locomotive if there are locomotives that have been allocated to move on the transfer arc. Connection arcs cannot have active locomotives on them, because when a locomotive is on a connection arc, it is idling at a location.
y_l^k	The variable is a measure of the number of inactive locomotives of class *k* on a particular arc *l*, where $k \in K, l \in AllArcs$. Locomotives are said to be inactive when their engine is turned off. This could mean two things. On a train or transfer arc, a locomotive with its engine turned off would be pulled by another active locomotive – a dead-in-train movement (see section 2.3.3). On the connection arcs, an inactive locomotive is idling – waiting for its next train or transfer movement.

Table 4-2

4.2.3. Objective function

Minimise: $n = \sum_{l \in S} \sum_{k \in K} y_l^k$

The objective of this minimum fleet size problem is to minimise the total number of locomotives, calculated by the objective function expression above. The objective function goes through each arc that connects to an end of schedule node and sums the total number of inactive locomotives of any class allocated to that arc. This counts the total number of locomotives used in the entire system for the following two reasons.

Firstly, the only sink nodes in the network are the end of schedule nodes. That means when locomotives are created at a source node (a train departure node) they must continue flowing on arcs through the network until they reach an end of schedule node. So, all locomotives in the network eventually end up going across a connection arc to reach an end of schedule node as they cannot be lost at any other node in the system. Secondly, because all the arcs that connect to the end of schedule nodes are connection arcs, all locomotives that are allocated to those arcs must be inactive. No locomotive can be active on a connection arc. That is why counting all of the inactive locomotives on the arcs that connect to end of schedule nodes is the same as counting the total number of allocated locomotives in the entire schedule.

4.2.4. Constraints

Constraints:

(a) $x_l^k = r_l^k$ for all $k \in K$, $l \in TrainArcs$

(b) $\sum_{k \in K} x_l^k \times \sum_{k \in K} y_l^k \geq \sum_{k \in K} y_l^k$ for all $l \in TransferArcs$

(c) $x_l^k \geq 0$ for all $k \in K$, $l \in TransferArcs$

(d) $x_l^k = 0$ for all $k \in K$, $l \in ConnectionArcs$

(e) $y_l^k \geq 0$ for all $k \in K$, $l \in AllArcs$

(f) $\sum_{l \in I[i]} (x_l^k + y_l^k) = \sum_{l \in O[i]} (x_l^k + y_l^k)$ for all $k \in K$, $i \in BalancedNodes$

(g) $\sum_{l \in I[i]} (x_l^k + y_l^k) \geq \sum_{l \in (O[i] - TrainArcs)} (x_l^k + y_l^k)$ for all $k \in K$, $i \in TrDepartureNodes$

(h) $\sum_{l \in I[i]} (x_l^k + y_l^k) \leq \sum_{l \in O[i]} (x_l^k + y_l^k)$ for all $k \in K$, $i \in TrDepartureNodes$

Constraint (a) requires that each train has the required number of active locomotives and that they are of the right class. Constraint (b) defines that if a transfer arc has inactive locomotives it must have at least one active locomotive to pull the inactive locomotives, and constraint (c) enforces that a transfer arc must have a non-negative number of active locomotives. Constraint (d) states that no locomotives can be active on a connection arc, and constraint (e) ensures that the system does not allow negative numbers of inactive locomotives to be allocated anywhere in the system. Constraint (f) enforces the rule that a node must have the same number of locomotives going in as it has going out for the relevant balanced nodes. Constraint (g) says that locomotives cannot be created for arcs that are not train departure arcs. Finally, constraint (h) defines that locomotives cannot be removed from the system at a train departure node, only created.

4.3. Formulation example

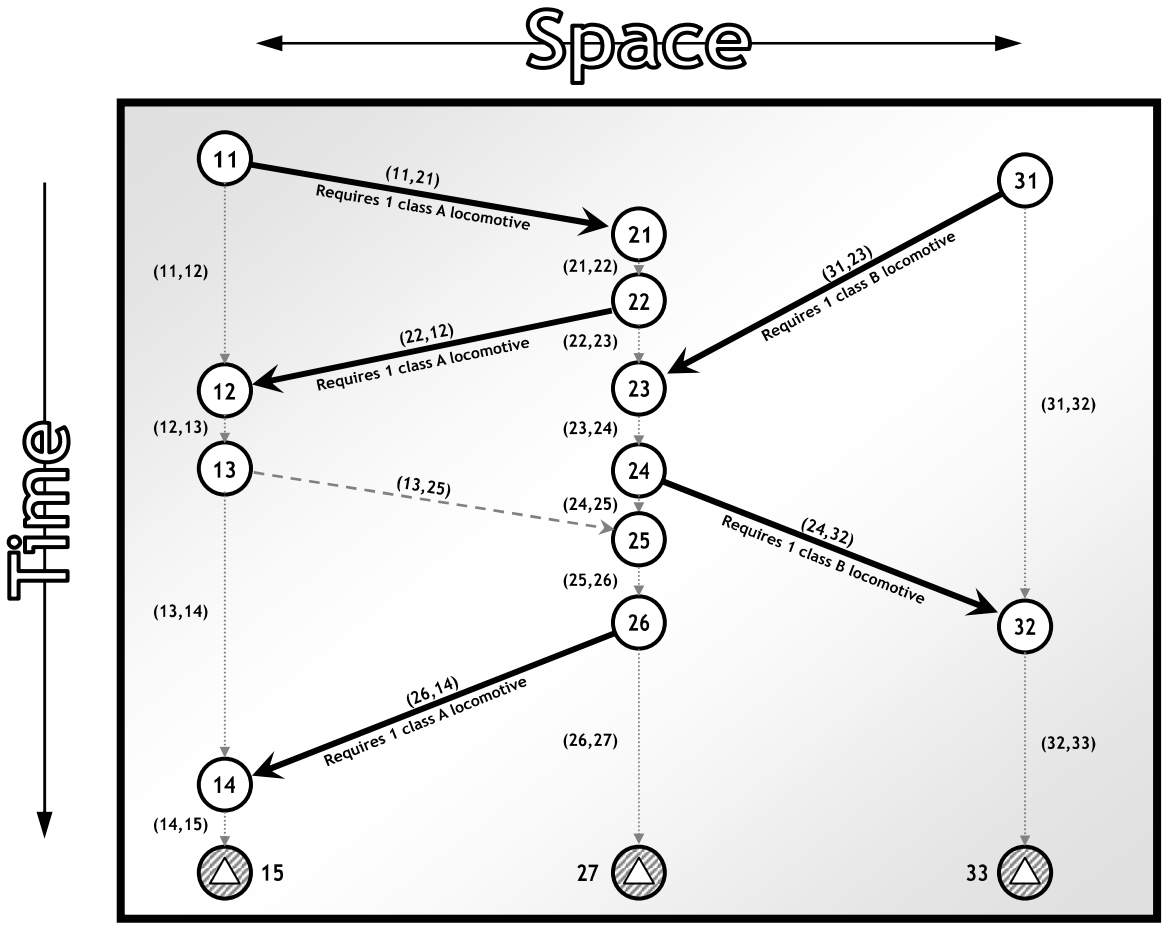


Figure 4-2: An example space-time network with each node and arc named for use in the mathematical formulation.

The diagram in Figure 4-2 describes an example node network that might be generated. The next pages will list how a system would take the generated space-time network to define the problem in purely mathematical terms.

4.3.1. Parameter Definitions

From the space-time network, first, the parameters for the problem are generated.

K = {A, B}

TrDepartureNodes = {11, 22, 24, 26, 31}
EOSNodes = {15, 27, 33}
BalancedNodes = {12, 13, 14, 21, 23, 25, 32}
AllNodes = $TrDepartureNodes \cup EOSNodes \cup BalancedNodes$

TrainArcs = { (11, 21), (22, 12), (24, 32), (26, 14), (31, 23) }

TransferArcs = { (13, 25) }
ConnectionArcs = { (11, 12), (12, 13), (13, 14), (14, 15), (21, 22), (22, 23), (23, 24), (24, 25), (25, 26), (26, 27), (31, 32), (32, 33) }
AllArcs = $TrainArcs \cup TransferArcs \cup ConnectionArcs$

S = { (14, 15), (26, 27), (32, 33) }

I[11] = ∅
I[12] = { (11, 12), (22, 12) }
I[13] = { (12, 13) }
I[14] = { (13, 14), (26, 14) }
I[15] = { (14, 15) }
I[21] = { (11, 21) }
I[22] = { (21, 22) }
I[23] = { (22, 23), (31, 23) }
I[24] = { (23, 24) }
I[25] = { (13, 25), (24, 25) }
I[26] = { (25, 26) }
I[27] = { (26, 27) }
I[31] = ∅
I[32] = { (24, 32), (31, 32) }
I[33] = { (32, 33) }

O[11] = { (11, 12), (11, 21) }
O[12] = { (12, 13) }
O[13] = { (13, 14), (13, 25) }
O[14] = { (14, 15) }
O[15] = ∅
O[21] = { (21, 22) }
O[22] = { (22,12), (22, 23) }
O[23] = { (23, 24) }
O[24] = { (24, 25), (24, 32) }
O[25] = { (25, 26) }
O[26] = { (26, 14), (26, 27) }
O[27] = ∅
O[31] = { (31, 23), (31, 32) }
O[32] = { (32, 33) }
O[33] = ∅

$r^A_{(11,21)} = 1$
$r^A_{(22,12)} = 1$
$r^A_{(24,32)} = 0$
$r^A_{(26,14)} = 1$
$r^A_{(31,23)} = 0$

$r^B_{(11,21)} = 0$
$r^B_{(22,12)} = 0$
$r^B_{(24,32)} = 1$
$r^B_{(26,14)} = 0$
$r^B_{(31,23)} = 1$

4.3.2. Objective Function Definition

Minimise: $n = \sum_{l \in S} \sum_{k \in K} y^k_l$

For this problem, expanding the summations results in this expression:
Minimise: $n = y^A_{(14,15)} + y^B_{(14,15)} + y^A_{(26,27)} + y^B_{(26,27)} + y^A_{(32,33)} + y^B_{(32,33)}$

4.3.3. Constraint Definition

(a) **Constraint:**

$x_l^k = r_l^k$ for all $k \in K$, $l \in TrainArcs$

Expanded:

$x^A_{(11,21)} = r^A_{(11,21)} = 1$ [k=A, l=(11,21)] $x^B_{(11,21)} = r^B_{(11,21)} = 0$ [k=B, l=(11,21)]

$x^A_{(22,12)} = r^A_{(22,12)} = 1$ [k=A, l=(22,12)] $x^B_{(22,12)} = r^B_{(22,12)} = 0$ [k=B, l=(22,12)]

$x^A_{(24,32)} = r^A_{(24,32)} = 0$ [k=A, l=(24,32)] $x^B_{(24,32)} = r^B_{(24,32)} = 1$ [k=B, l=(24,32)]

$x^A_{(26,14)} = r^A_{(26,14)} = 1$ [k=A, l=(26,14)] $x^B_{(26,14)} = r^B_{(26,14)} = 0$ [k=B, l=(26,14)]

$x^A_{(31,23)} = r^A_{(31,23)} = 0$ [k=A, l=(31,23)] $x^B_{(31,23)} = r^B_{(31,23)} = 1$ [k=B, l=(31,23)]

(b) **Constraint:**

$\sum_{k \in K} x_l^k \times \sum_{k \in K} y_l^k \geq \sum_{k \in K} y_l^k$ for all $l \in TransferArcs$

Expanded:

$(x^A_{(13,25)} + x^B_{(13,25)}) \times (y^A_{(13,25)} + y^B_{(13,25)}) \geq (y^A_{(13,25)} + y^B_{(13,25)})$ [l = (13,25)]

(c) **Constraint:**

$x_l^k \geq 0$ for all $k \in K$, $l \in TransferArcs$

Expanded:

$x^A_{(13,25)} \geq 0$ [k = A, l = (13, 25)] $x^B_{(13,25)} \geq 0$ [k = B, l = (13, 25)]

(d) **Constraint:**

$x_l^k = 0$ for all $k \in K$, $l \in ConnectionArcs$

Expanded:

$x_{(11,12)}^A = 0$ [k = A, l = (11, 12)] $x_{(11,12)}^B = 0$ [k = B, l = (11, 12)]

$x_{(12,13)}^A = 0$ [k = A, l = (12, 13)] $x_{(12,13)}^B = 0$ [k = B, l = (12, 13)]

$x_{(13,14)}^A = 0$ [k = A, l = (13, 14)] $x_{(13,14)}^B = 0$ [k = B, l = (13, 14)]

…

(e) **Constraint:**

$y_l^k \geq 0$ for all $k \in K$, $l \in AllArcs$

Expanded:

$y_{(11,12)}^A \geq 0$ [k = A, l = (11, 12)] $y_{(11,12)}^B \geq 0$ [k = B, l = (11, 12)]

$y_{(11,21)}^A \geq 0$ [k = A, l = (12, 13)] $y_{(11,21)}^B \geq 0$ [k = B, l = (12, 13)]

$y_{(12,13)}^A \geq 0$ [k = A, l = (13, 14)] $y_{(11,13)}^B \geq 0$ [k = B, l = (13, 14)]

…

(f) **Constraint:**

$$\sum_{l \in I[i]} (x_l^k + y_l^k) = \sum_{l \in O[i]} (x_l^k + y_l^k) \text{ for all } k \in K,\ i \in BalancedNodes$$

Expanded:

$(x^A_{(11,12)} + y^A_{(11,12)}) + (x^A_{(22,12)} + y^A_{(22,12)}) = (x^A_{(12,13)} + y^A_{(12,13)})$ [k = A, i = 12]

$(x^B_{(11,12)} + y^B_{(11,12)}) + (x^B_{(22,12)} + y^B_{(22,12)}) = (x^B_{(12,13)} + y^B_{(12,13)})$ [k = B, i = 12]

$(x^A_{(12,13)} + y^A_{(12,13)}) = (x^A_{(13,14)} + y^A_{(13,14)}) + (x^A_{(13,25)} + y^A_{(13,25)})$ [k = A, i = 13]

$(x^B_{(12,13)} + y^B_{(12,13)}) = (x^B_{(13,14)} + y^B_{(13,14)}) + (x^B_{(13,25)} + y^B_{(13,25)})$ [k = B, i = 13]

$(x^A_{(13,14)} + y^A_{(13,14)}) + (x^A_{(26,14)} + y^A_{(26,14)}) = (x^A_{(14,15)} + y^A_{(14,15)})$ [k = A, i = 14]

$(x^B_{(13,14)} + y^B_{(13,14)}) + (x^B_{(26,14)} + y^B_{(26,14)}) = (x^B_{(14,15)} + y^B_{(14,15)})$ [k = B, i = 14]

…

(g) **Constraint:**

$$\sum_{l \in I[i]} (x_l^k + y_l^k) \geq \sum_{l \in (O[i] \cap TrainArcs)} (x_l^k + y_l^k) \text{ for all } k \in K,\ i \in TrDepartureNodes$$

Expanded:

$0 \geq (x^A_{(11,12)} + y^A_{(11,12)})$ [k = A, i = 11]

$0 \geq (x^B_{(11,12)} + y^B_{(11,12)})$ [k = B, i = 11]

$(x^A_{(21,22)} + y^A_{(21,22)}) \geq (x^A_{(22,23)} + y^A_{(22,23)})$ [k = A, i = 22]

$(x^B_{(21,22)} + y^B_{(21,22)}) \geq (x^B_{(22,23)} + y^B_{(22,23)})$ [k = B, i = 22]

$(x^A_{(23,24)} + y^A_{(23,24)}) \geq (x^A_{(24,25)} + y^A_{(24,25)})$ [k = A, i = 24]

$(x^B_{(23,24)} + y^B_{(23,24)}) \geq (x^B_{(24,25)} + y^B_{(24,25)})$ [k = B, i = 24]

…

(h) **Constraint:**

$$\sum_{l \in I[i]} (x_l^k + y_l^k) \leq \sum_{l \in O[i]} (x_l^k + y_l^k) \text{ for all } k \in K,\ i \in TrDepartureNodes$$

Expanded:

$0 \leq (x^A_{(11,12)} + y^A_{(11,12)}) + (x^A_{(11,21)} + y^A_{(11,21)})$ [k = A, i = 11]

$0 \leq (x^B_{(11,12)} + y^B_{(11,12)}) + (x^B_{(11,21)} + y^B_{(11,21)})$ [k = B, i = 11]

$(x^A_{(21,22)} + y^A_{(21,22)}) \leq (x^A_{(22,12)} + y^A_{(22,12)}) + (x^A_{(22,23)} + y^A_{(22,23)})$ [k = A, i = 22]

$(x^B_{(21,22)} + y^B_{(21,22)}) \leq (x^B_{(22,12)} + y^B_{(22,12)}) + (x^B_{(22,23)} + y^B_{(22,23)})$ [k = B, i = 22]

$(x^A_{(23,24)} + y^A_{(23,24)}) \leq (x^A_{(24,25)} + y^A_{(24,25)}) + (x^A_{(24,32)} + y^A_{(24,32)})$ [k = A, i = 24]

$(x^B_{(23,24)} + y^B_{(23,24)}) \leq (x^B_{(24,25)} + y^B_{(24,25)}) + (x^B_{(24,32)} + y^B_{(24,32)})$ [k = B, i = 24]

…

4.4. AMPL definition of the problem

A standard way to express a mathematical formulation such as this one is through the use of an algebraic modelling language. One such language is AMPL (Fourer, Gay & Kernighan, 1990; Holmes, 1995). AMPL stands for “a mathematical programming language” and many of the common mathematical programming solvers take mathematical problems as input in AMPL form. A full listing of all solvers that are capable of using AMPL can be found on the AMPL website[3]. The formulation for this problem has been converted into AMPL. This can be found in Appendix A.

4.5. Tests of the formulation

To ensure the objective function and constraints were properly defined, several test space-time networks were created and solved using existing integer programming solvers. The solver that was used was the PENNON solver, which is described in more detail in section 6.1.

[3] AMPL website: www.ampl.com
Listing of solvers that use AMPL: www.ampl.com/solvers.html

4.5.1. Transfer arc test

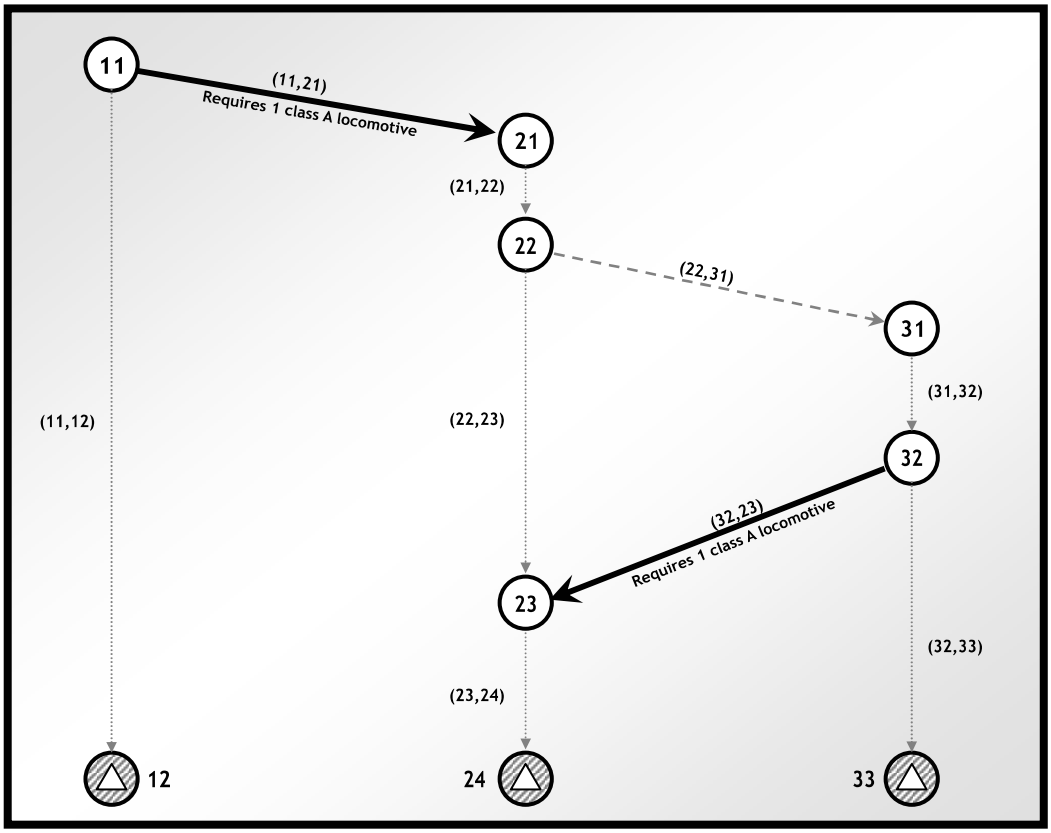


Figure 4-3

The space-time network in Figure 4-3 was used to test that the formulation correctly defined transfer arcs. The entire space-time network in that figure can be run with just one locomotive, but the solver has to use the transfer arc (22, 31) to reach that minimum.

After inputting this space-time network in AMPL form, the PENNON solver successfully allocated one active locomotive to the transfer arc (22, 31) and reached the minimum of one locomotive for this space-time network. The appropriate intermediate connection arcs were also validly allocated with one locomotive each, and the final connection arc (23, 24) was correctly allocated one locomotive.

4.5.2. Dead-in-train test

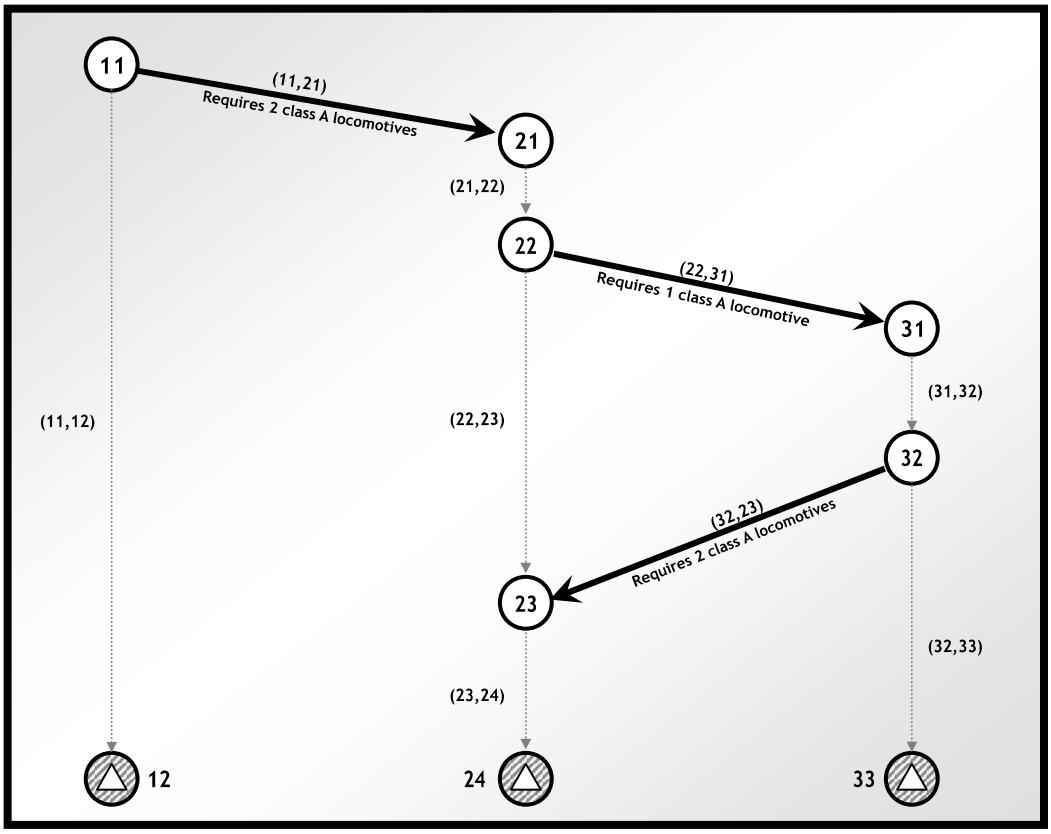


Figure 4-4

Figure 4-4 depicts the network used to test whether the formulation correctly defined dead-in-train movements. The first and the third train both require two locomotives, but the second train requires only one. Using a dead-in-train movement to transfer one inactive locomotive, it is possible to run this space-time network with a minimum of two locomotives. The PENNON solver successfully allocated the dead-in-train movement and reached the minimum of two locomotives. The solver also validly allocated all the connection arcs with the appropriate number of locomotives.

4.5.3. Locomotive classes test

Finally, the space-time network which was used for the example in Figure 4-2 was tested to confirm that the mathematical formulation could handle multiple locomotive classes correctly. The entire space-time network in Figure 4-2 can be validly allocated with a minimum of two locomotives, one of class A and one of class B. To achieve this minimum, the class A locomotive has to run the three trains on the left side of the figure, utilising the transfer movement (13, 25). The class B locomotive has to run the two trains on the right side of the diagram. After supplying this space-time network to the PENNON solver in AMPL form, this minimum result was achieved correctly, using the transfer movement for the class A locomotive and also allocating the appropriate connection arcs in the network to fulfil all the constraints.

5. Description of Data

Three approaches were taken to solve the problem, and they are outlined in sections 6, 7 and 8. The solution approaches were all applied to the same set of train schedules in order to compare their performance. These train schedules were extracted from a real system and transformed to fit the hypothetical problem.

5.1. Size measures of the train schedule

Table 5-1 lists several measures of train schedule size and their rounded mean values over all the train schedules that were tested.

Measure	Rounded mean value
Total number of train arcs	7 200
Total number of transfer arcs	39 700
Total number of paths	10 700
Number of paths used for trains	7 200
Number of paths not used for trains	3 500
Number of unique VSTPs available	13 200
Number of location group movements created	23 000
Maximum number of simultaneous trains at a particular instant of the week	310
Number of locations	780
Number of locomotive classes	18

Table 5-1

5.2. Train schedule class diagram

The train schedule data is stored in an object-oriented JADE database and supplied as input to the algorithm. The class diagram for the train schedule data is presented in Figure 5-1.

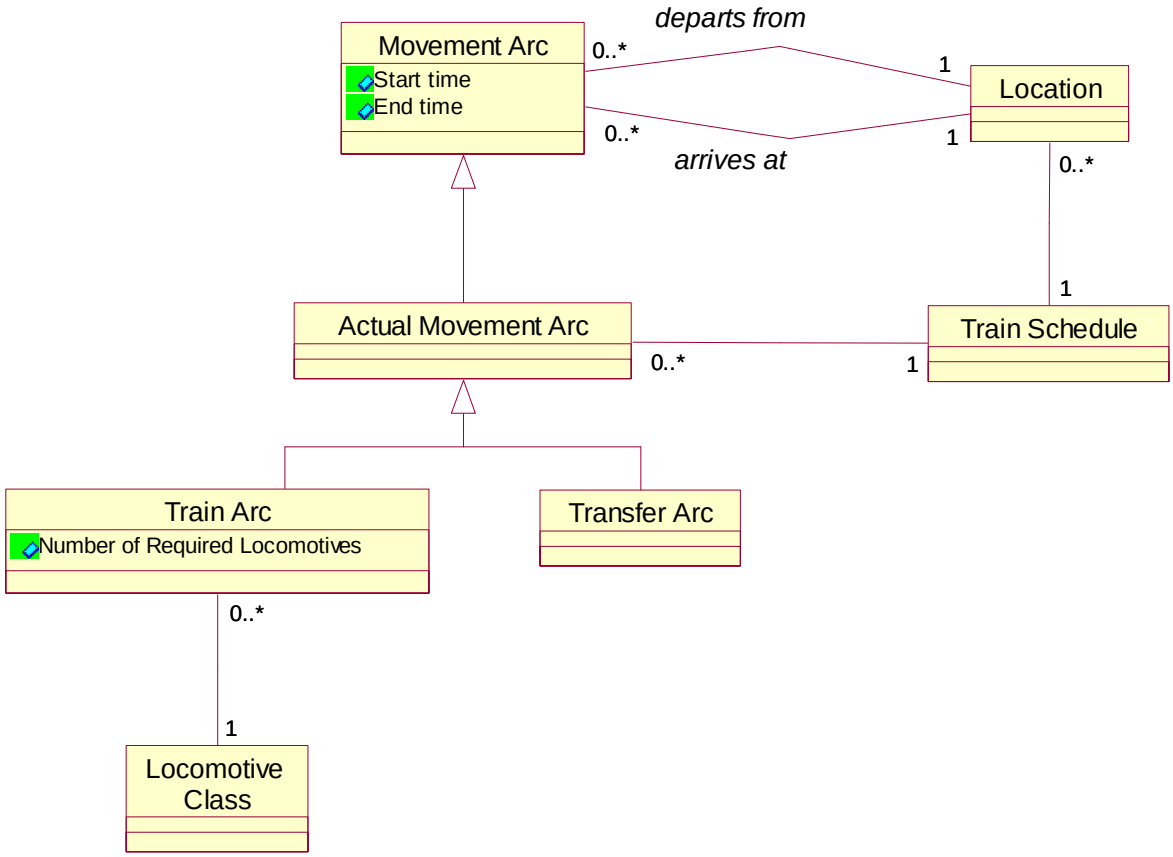


Figure 5-1

The train arc and transfer arc class both inherit from actual movement arc instead of directly from movement arc in anticipation of additional classes being created to inherit from movement arc. As will be seen in sections 7 and 8, multi-stage transfers will have a departure location and time as well as an arrival location and time, so it makes sense if these multi-stage transfers can be represented as a subclass of movement arc without interfering with any existing relationships.

6. Using Existing Integer Programming Solvers

The use of integer programming solvers is a standard approach to solving minimisation problems. An integer programming solver approach was applied to this problem to find out whether this standard approach would be enough to solve the problem, or whether a new direction had to be taken.

6.1. Solver Used

There is a wide range of integer programming solvers available, each implementing different algorithms designed to solve different types of problems. To solve this problem, the NEOS server[4] (Gropp & Moré, 1997; Czyzyk, Mesnier & Moré, 1998; Dolan, 2001) was used, which is a server provided free to the public to solve mathematical programs by the Argonne National Laboratory[5]. A wide range of solvers can be used via the NEOS server.

After testing a range of solvers, the solver that was chosen for this problem was the PENNON solver, as its algorithm was able to converge to a solution for train schedules of substantial size, while other solvers quickly ran out of memory or were only able to converge on a solution for train schedules of a trivial size. PENNON stands for **pen**alty method for **non**linear and semi-definite programming, and is aimed to solve large-scale problems with a sparse data structure. PENNON uses an algorithm based around the generalized augmented Lagrangian method. A conference paper describing the algorithm in full has been published by the creators of PENNON, Kocvara and Stingl (2001).

PENNON takes mathematical formulations via AMPL input, and so the model in Appendix A was used to input train schedules to PENNON. Additionally, the PENNON solver reports whether it has converged on the optimal solution or a suboptimal solution.

6.2. Method for testing

A full week's train schedule could not be solved within the four-hour solve time limit imposed by the NEOS server, and so only partial train schedules were tested. A number of partial train schedules of varying sizes were formulated by taking slices of the same train schedule. Each partial train schedule contained the train and transfer arcs for a specified number of minutes from the start of the schedule. The number of minutes in each slice was increased in 30-minute intervals to generate progressively larger partial train schedules, ranging from the 30 minutes to the 2880 minutes (48 hours) of the train schedule. In total, 96 partial train schedules were generated and sent to the solver.

[4] NEOS Server: www-neos.mcs.anl.gov
[5] Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439. Web: www.anl.gov

The 96 partial train schedules were generated using a script made in JADE. The script generated partial train schedules so that they had the same data structure as full train schedules (the data structure of train schedules was described in section 5.2). This allowed the partial train schedules to be used for not only the integer programming solver but also the other solution approaches without changing any code. This would be useful for comparing multiple solution approaches. For each of the generated partial train schedules, the system would then generate AMPL data files which could be sent to the PENNON solver along with the AMPL model file presented in section Appendix A. Commands were specified to the PENNON solver so that it would output the time taken to solve each train schedule, and also the number of locomotives required for each solution it generated. These values were collated with other measures that were output from the script to produce the results.

6.3. Results

All of the partial train schedules that were solved were able to be solved to optimality by the PENNON solver. The largest partial train schedule that was able to be solved included only the first 2160 minutes (36 hours) of the full train schedule. Partial train schedules larger than this were unable to be solved and caused out-of-memory errors or reached the four-hour solve time limit. Partial train schedules smaller than 600 minutes were extremely trivial and took less than one-hundredth of a second to solve, and because the solve times were reported to the nearest hundredth of a second, these partial train schedules were discarded from the result set as their time was not measured accurately enough. To present an indication of the results gained from the full set of partial train schedules, the results for every fifth partial train schedule between 600 and 2100 minutes along with the largest 2160-minute partial train schedule are shown in Table 6-1. The full set of results is shown in Appendix B.

Minutes of Schedule	Locomotives Required	Solve Time (seconds)	Train Arcs	Transfer Arcs	Locations
600	8	0.01	8	64	51
750	10	0.26	11	124	75
900	16	0.34	20	187	95
1050	26	2.05	33	263	119
1200	40	12.69	51	390	163
1350	56	58.54	80	573	215
1500	66	411.89	102	793	261
1650	75	337.43	122	1044	328
1800	90	1967.93	163	1413	408
1950	109	5036.16	211	1835	483
2100	134	4663.08	274	2316	536
2160	147	8869.35	307	2503	548

Table 6-1: The results found from using the solver PENNON to solve partial train schedules ranging in size from the first 600 to 2160 minutes of the full train schedule, including only the results for every 150 minutes.

As expected, the solve time increases rapidly as the size of the integer program increases. Figure 6-1 illustrates how the total solve time increases as the number of train arcs increases.

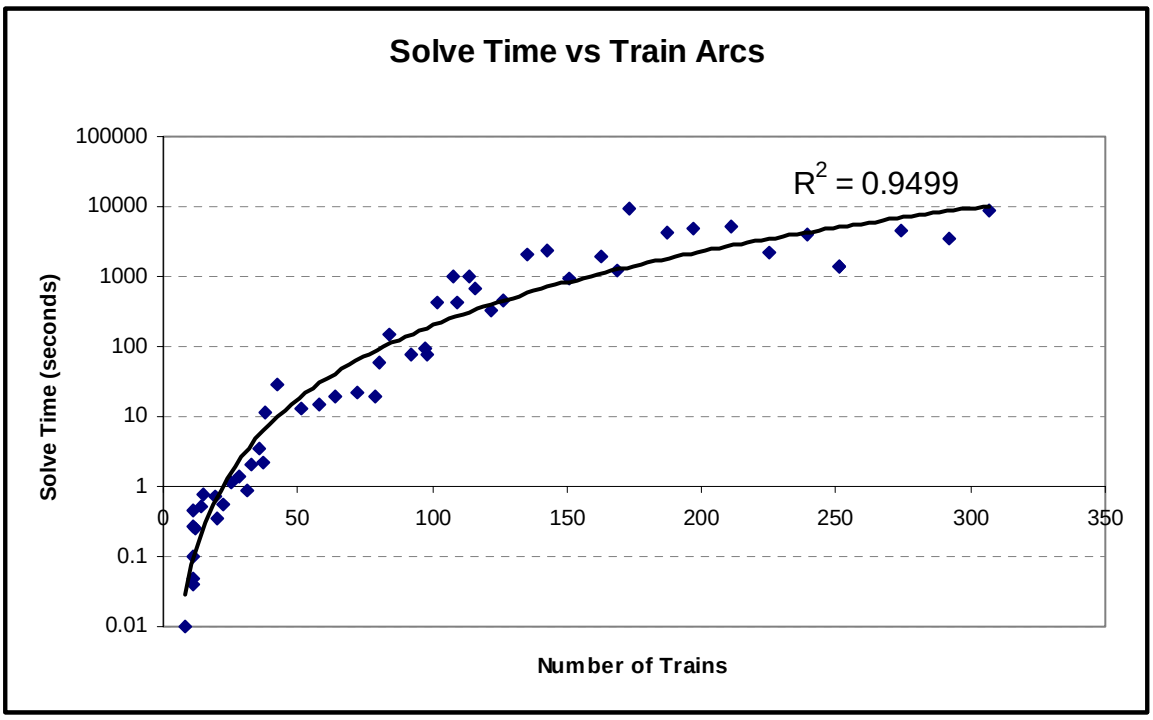


Figure 6-1: The solve time versus the number of train arcs in the train schedule

Notice that Figure 6-1 was drawn with a logarithmic scale because of the rate at which the solve time increases. A power law regression line is drawn on the graph as it was found to be the type of regression that fitted that data best. Given that the regression has a coefficient of determination of 0.9499, it is clear that the train schedule solve time increases at an escalating rate with the size of the train schedule when using an integer programming solver. The full scale train schedule could not be solved with any of the available integer programming solvers. This leads into the exploration of new approaches to solve this minimum fleet size problem.

7. Same-location Merge Algorithm

The same-location merge algorithm was developed to allocate an entire train schedule in reasonable time. For this algorithm, the concept of the locomotive diagram is very important. Recall from section 2.6 that a locomotive diagram is the sequence of trains and transfers that will be run by one logical locomotive during a train schedule's week. This algorithm runs in multiple phases. The first phase of this algorithm produces an initial solution which is good but not optimal. The next phases of the algorithm then incrementally improve the solution by combining locomotive diagrams. The three phases of the algorithm are summarised as follows:

1. **Same-location connection phase** – connects each departing train with the most recent unallocated arriving train at the same location. No transfers are used in this phase.

2. **Different-location connection phase** – finds and connects situations where the end of one locomotive diagram can be connected to the start of another locomotive diagram using a transfer movement.

3. **Different-location merge phase** – attempts to reallocate the trains in each locomotive diagram into whitespaces (described in section 2.6) of other locomotive diagrams.

7.1. Same-location connection phase

Within this phase, locomotives are assigned to trains so that each train departing from a particular location reuses a locomotive that has previously arrived at that location, if one is available. If there is no locomotive available for the train to reuse, then a new logical locomotive is created to begin its week with that particular train. No transfers of any kind are used in this phase. Since this phase connects train arrivals to train departures at the same location, this phase is called the same-location connection phase.

If the algorithm gets to a particular train that departs from a location at such a time that there are multiple locomotives available for it to reuse, it will take the locomotive that has most recently arrived. This last-in-first-out method of allocation is commonly used in practice because it creates varying amounts of whitespace between movements in locomotive diagrams, with some locomotives having long waiting times and others having very short waiting times. To illustrate this, imagine one departing train. This train would take the most recent arrival to the departure location. Imagine a second departing train that departs directly after the first. This second departing train must take a locomotive that has been waiting for much longer than the locomotive used for the first departing train, as the locomotive for the first departing train has already been allocated and cannot be used for the second departing train. The second locomotive in this example will have

an extra long wait which could not occur if allocation was done on a first-in-first-out basis, as that would mean the locomotive that has been waiting the longest would be allocated first.

Having locomotives with long waiting times is useful to the rail operator, as the rail operator will have spare locomotives for longer periods which can be used to cover trains of other locomotives if the schedule does not go to plan. So, using last-in-first-out enables the algorithm to produce a more flexible schedule.

7.1.1. Same-location connection phase algorithm

The pseudocode for this phase of the allocation algorithm is presented below.

```
Set unconnected-trains-list := empty list;
Set train-list := list of all trains in the train schedule sorted by their departure time;
For each current-train in train-list do
    Set departure-location := departure location of current-train;
    Set unconnected-arriving-train-list := list of all trains in unconnected-trains-list that arrive at
        departure-location;
    If unconnected-arriving-train-list is not empty then:
        Set last-unconnected-train := the train that has the last arrival time in unconnected-arriving-train-list;
        Connect last-unconnected-train to current-train;
        Remove last-unconnected-train from unconnected-trains-list;
    End if;
    Add current-train to unconnected-trains-list;
End for;
```

Algorithm 7-1

7.2. Different-location connection phase

Once a solution has been produced by the same-location connection phase, this next phase begins to use transfers to combine multiple locomotive diagrams into single locomotive diagrams. This phase identifies situations where there is one locomotive diagram that ends earlier than another locomotive diagram starts. Looking at the two locomotive diagrams, the algorithm searches for a possible transfer that can be made from the end of the first locomotive diagram to the start of the second locomotive diagram. If a transfer is found, then the two locomotive diagrams are combined into one, using the transfer movement to bridge the gap between the first locomotive diagram and the second.

7.2.1. Different-location connection phase algorithm

This algorithm connects the end of one locomotive diagram to the start of another locomotive diagram wherever there is a transfer movement that is available to make a connection possible.

Set *diagrams-by-start-time* := list of all locomotive diagrams **sorted by** their first train's start time;
Set *diagrams-by-end-time* := list of all locomotive diagrams **sorted by** their last train's end time;
For each *first-diagram* **in** *diagrams-by-end-time* **do**
    Set *first-end-time* := end time of *first-diagram*;
    For each *second-diagram* **in** *diagrams-by-start-time* **do**, beginning iteration from the diagram that has a start time after *first-end-time*:

        Set *transfer-sequence* := **Find** a sequence of transfer movements which can be traversed to get from the end location and end time of *first-diagram* to the start location and start time of *second-diagram*; // see algorithm in section 7.2.2

        If tr*ansfer-sequence* was found **then**
            Add all transfer movements **in** *transfer-sequence* **to** *second-diagram*;
            Reallocate all movements in *first-diagram* **to** *second-diagram*;
            Discard *first-diagram*; // *first-diagram* is empty
        End if;
    End for;
End for;

Algorithm 7-2

7.2.2. Transfer sequence search algorithm

This algorithm performs a breadth-first search for a sequence of transfer movements that can transfer a locomotive from one location to another location between a transfer departure time and transfer arrival time.

```
Parameter target-departure-location := location at which the transfer sequence should start from;
Parameter target-departure-time := time at which the transfer sequence should start at;
Parameter target-arrival-location := location at which the transfer sequence should arrive at;
Parameter target-arrival-time := time at which the transfer sequence must be completed by;

Set unextended-transfer-sequences-list := empty list;

Set initial-transfer-sequence := empty transfer sequence;
Set end location of initial-transfer-sequence := target-departure-location;
Set end time of initial-transfer-sequence := target-departure-time;
Add intial-transfer-sequence to unextended-transfer-sequences-list;

While unextended-transfer-sequences-list is not empty do
    Set transfer-sequence := retrieve and remove first transfer sequence in unextended-transfer-sequences-
        list;

    Set movement-extensions-list := all paths or location group movements departing from the end location of
        transfer-sequence after the end time of transfer-sequence;
    For each movement-extension in movement-extensions-list do
        Set extension-end-location := end location of movement-extension;
        Set extension-end-time := end time of movement-extension;

        If extension-end-time is after target-arrival-time then
            Skip to next for loop iteration; // since there is no point in the locomotive continuing the transfer
                sequence when the target arrival time has already passed.
        Else if extension-end-location is the target-arrival-location then
            Set found-transfer-sequence := copy of transfer-sequence;
            Add movement-extension to found-transfer-sequence;
            Return found-transfer-sequence;
        Else if extension-end-location has not been visited or the earliest visit to extension-end-location was
            after extension-end-time then

            Set new-transfer-sequence := copy of transfer-sequence;
            Add movement-extension to new-transfer-sequence;

            Set end location of new-transfer-sequence := extension-end-location;
            Set end time of new-transfer-sequence := extension-end-time;
            Add new-transfer-sequence to unextended-transfer-sequences-list;

            Mark extension-end-location as visited;
            Set earliest visit time of extension-end-location := extension-end-time;
        End if;
    End for;
End while;
```

Algorithm 7-3

7.3. Different-location merge phase

The previous phase of the algorithm combines locomotive diagrams that do not have a time overlap. This phase attempts to combine locomotive diagrams that do have a time overlap by reassigning trains to be run within the whitespaces of other locomotive diagrams. Recall from section 2.6 that whitespace is times in locomotive diagrams where locomotives are idling and could be used for more productive tasks such as running trains.

To do this, each locomotive diagram is examined and attempts are made to reallocate all of its trains to other locomotive diagrams. If only some of the trains can be reassigned to other locomotive diagrams, the reallocation is cancelled for that locomotive diagram and no trains are reallocated. This is because the goal of the algorithm is to minimise the number of locomotives, and if only some of the trains in one locomotive diagram can be reassigned to other locomotive diagrams, then the first locomotive is still needed. So in this case, there is no gain to be made by reallocating just part of the first locomotive's diagram as the same number of locomotives is still required. In fact, it is better to have the first locomotive with its original locomotive diagram as it balances the number of commitments each locomotive has, which means the train schedule has more flexibility.

7.3.1. Different-location merge algorithm

This algorithm attempts to reassign trains in one locomotive diagram into another locomotive diagram using transfer movements.

```
For each current-diagram in all locomotive diagrams do
    Set initial-allocated-train-schedule := checkpoint of the allocated train schedule that can be reverted to;
    Set is-merge-possible := true;

    For each current-diagram-train in all train movements in current-diagram do
        Set is-merge-of-train-possible := false;
        For each other-diagram in all locomotive diagrams do
            If other-diagram is not the current-diagram and current-diagram-train does not clash with a train
                movement in other-diagram then

                Set other-diagram-train-before := the latest train that ends before current-diagram-train in
                    other-diagram;
                Set other-diagram-train-after := the earliest train that begins after current-diagram-train in
                    other-diagram;

                Set transfer-sequence-before := find a sequence of transfer movements that can be used to get
                    from the end of other-diagram-train-before to the start of current-diagram-train;
                If transfer-sequence-before was not found then
                    Skip to next for loop iteration;
                End if;

                Set transfer-sequence-after := find a sequence of transfer movements that can be used to get
                    from the end of current-diagram-train to start of other-diagram-train-after;
                If transfer-sequence-after was not found then
                    Skip to next for loop iteration;
                End if;

                Set inner-transfer-movements-list := list of all transfer movements between other-diagram-
                    train-before and other-diagram-train-after on the other-diagram;
                Remove all transfer movements in inner-transfer-movements-list from other-diagram;

                Add transfer-sequence-before to other-diagram;
                Reallocate current-diagram-train to other-diagram;
                Add transfer-sequence-after to other-diagram;

                Set is-merge-of-train-possible := true;
                Exit for;
            End if;
        End for;

        If is-merge-of-train-possible is false then
            is-merge-possible := false;
            Exit for;
        End if;
    End for;

    If is-merge-possible is true then
        Discard current-diagram; // all trains run by current-diagram will have been reassigned.
    Else
        Rollback allocated train schedule to initial-allocated-train-schedule;
    End if;
End for;
```

Algorithm 7-4

7.4. Implementation

This algorithm was implemented in JADE, the programming language created by Jade Software Corporation. The implementation uses the class diagram shown in Figure 7-1.

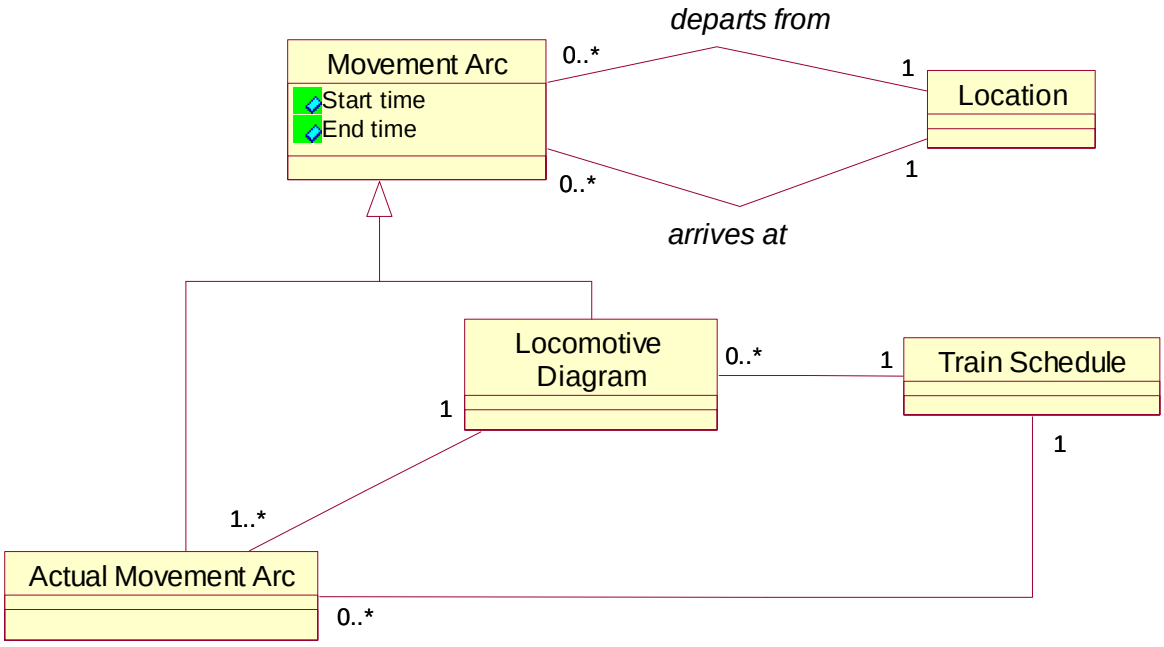


Figure 7-1

The locomotive diagram class in Figure 7-1 stores a sequence of train arcs and transfer arcs for one logical locomotive. Its inherited properties of start location, start time, end location and end time are taken from the start and end of the full sequence of train arcs and transfer arcs in the locomotive diagram. This information is useful when connecting multiple locomotive diagrams together. The same-location connection phase of the algorithm first generates an initial set of locomotive diagrams which are later combined by the different-location connection and different-location merge phases of the algorithm.

7.5. Results

The same-location merge algorithm was applied to the same train schedule that the integer programming solver was applied to, except the same-location merge algorithm worked on all the data. Overall, all the phases of the algorithm took approximately 2.5 hours, and created an allocated schedule that required 1048 locomotives. The results after each phase are displayed below:

Phase	Cumulative Execution Time	Locomotives Required after Execution
Same-location connection phase	53 seconds	1703
Different-location connection phase	18 minutes, 6 seconds	1058
Different-location merge phase	2 hours, 17 minutes, 19 seconds	1048

Remember the output of each phase is a fully allocated train schedule – the different-location connection phase and the different-location merge phase just improve the allocations from the last phase. This means that not all phases have to be run. For example, the rail operator might decide that it is not be worth running the different-location merge phase as it adds about two hours to the solve time of the algorithm but only saves ten locomotives.

7.6. Drawbacks of the Same-location Merge Algorithm

Fundamentally, solving the locomotive allocation problem really comes down to just finding the optimal way to connect trains to previous trains. In other words, the problem is about finding ways to use a locomotive that is already used for one train to pull another train.

The first phase in the same-location merge algorithm does this by trying to use locomotives from previous trains that arrive at the same location the current train departs from. Transfer movements of any kind are not considered by the first phase in the same-location merge algorithm. Restricting the solution space to ignore transfer movements means that the same-location connection phase of the same-location merge algorithm alone can never find the optimal solution.

The other two phases of the algorithm attempt to remedy this by taking the resulting schedule generated by the same-location connection phase and improving it to utilise the transfer movements that the same location connection phase ignores. Using those transfer movements, the other phases analyse one locomotive diagram and attempt to reassign the trains that it runs into the white spaces of other locomotive diagrams. But the

downfall of this is the algorithm only ever considers merging two locomotive diagrams at any one time – one source locomotive diagram with the trains that the algorithms are trying to reallocate, and another destination locomotive diagram which the algorithm tries to reallocate the trains to. Sometimes, swapping trains between only two locomotive diagrams at once is not enough.

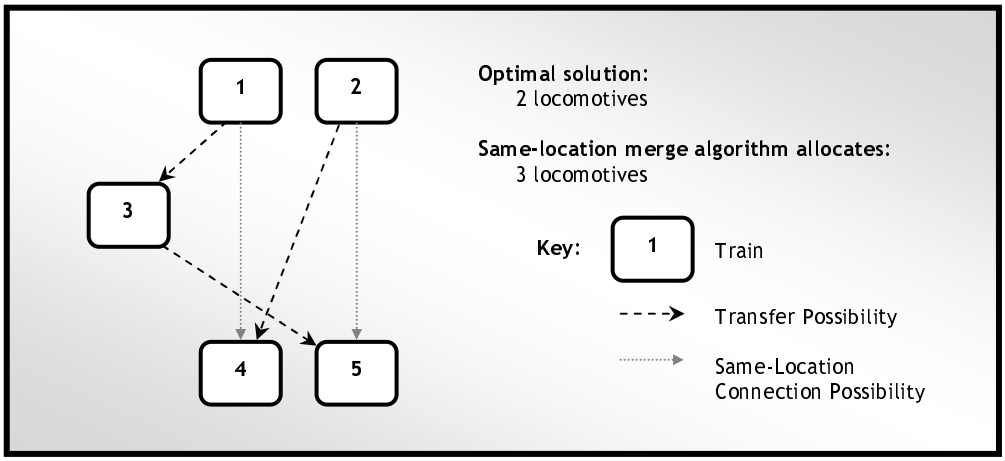


Figure 7-2: The same-location merge algorithm can never find the optimal solution on some networks.

Consider the example in Figure 7-2. There are five trains in this train schedule. Optimally, this train schedule can be satisfied with only two locomotives by connecting trains 1-3-5 and 2-4, where the hyphen indicates a connection between trains which may involve a transfer movement. The same-location merge algorithm will begin with its first phase connecting locomotives at the same location, resulting in a three-locomotive solution: 1-4, 2-5 and 3. When the other phases of the same-location merge algorithm run, they cannot bring the number of locomotives required down to the optimal two locomotives. They would try to move train 3 into another locomotive's diagram. To do this, first they would test the possibility of pushing train 3 into the 1-4 locomotive diagram to get 1-3-4. This is impossible as there is no way to connect train 3 to train 4, as you can see in the diagram. The algorithm would then try to push train 3 into the 2-5 locomotive diagram to get 2-3-5. This is also impossible as there is no way to connect train 2 to train 3. This example shows that there are some networks for which the same-location merge algorithm could never find the optimal solution, and the reason for that is, when it comes to transfer movements, the same-location merge algorithm does not consider all the possibilities.

8. Work unit levels algorithm

Knowing the drawbacks of the same-location merge algorithm, a new algorithm was sought to overcome the weaknesses of the same-location merge algorithm. The result was the work unit levels algorithm, which takes a completely new approach to solving the locomotive allocation problem.

The algorithm has four phases:

1. **Work unit determination** – trains in the schedule are grouped into short sequences called work units. Trains in a work unit are run optimally when they are run together in sequence - the system does not need to perform any optimisation on the trains within the work units as they are already optimally connected.

2. **Identification of resourcing options** – the algorithm finds every possible connection that can be made from one work unit to all other work units.

3. **Level assignment phase** – the train schedule is divided into what are called levels in this phase. Levels are an extension of the idea that there is an order to the work units based on how one work unit can resource other work units later in the schedule. Each of the levels can be solved individually in the next phase.

4. **Possibilities network construction and solution** – the work units are added and solved level-by-level in what is called the possibilities network for the train schedule. The result is a number of work unit chains, each of which represents a locomotive diagram.

8.1. Work unit determination phase

The first phase of the algorithm is to combine trains into short sequences called work units. Work units are similar to the matching heuristic used by Scholz (1998). All the trains in a work unit will be run by the same set of one or more locomotives in sequence. When trains are connected together to form a work unit, it means that they are run optimally when run by the same set of locomotives. In other words, reconnecting the trains in the work unit to other trains in the schedule besides the ones in the work unit will not lead to a more optimal solution. Because of this, work units are the atomic unit of the schedule as far as the algorithm is concerned, as no optimisation needs to be made to reconnect the trains within a work unit. Combining trains into optimal work units like this assists the next phases of the algorithm to find a better solution, as it removes some of the suboptimal options from the solution space.

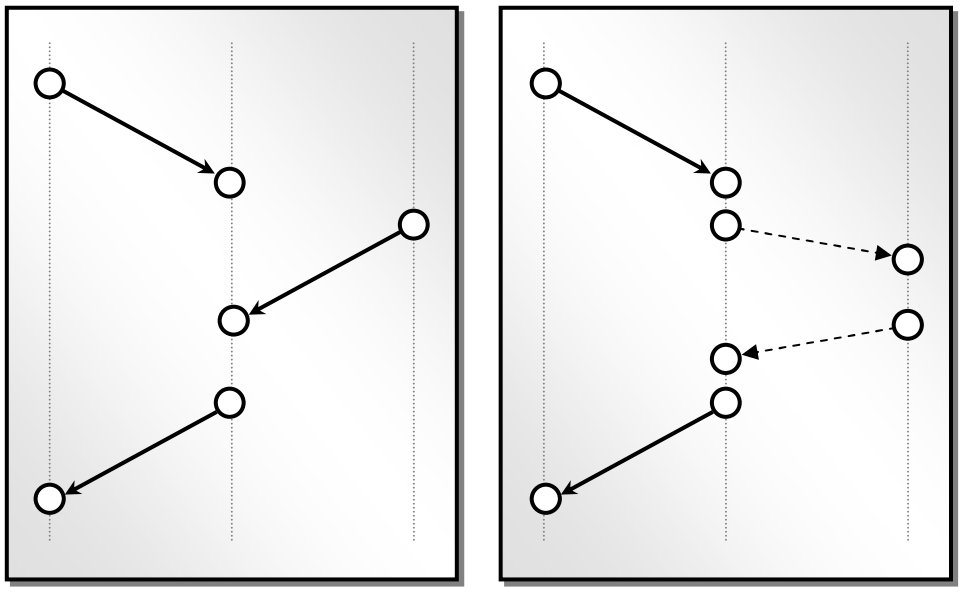
Single trains that cannot be connected to other trains at the work unit determination stage become single-train work units so that they can be treated in the same way as other work units. Once work units are identified, the identification of resourcing options phase and the level assignment phase perform preprocessing on the work units to prepare them for the possibilities network phase in which the entire schedule is solved.

Since the possibilities network phase is already very good at connecting trains, the work unit determination phase connects trains into work units only when it is certain that it is optimal to do so.

8.1.1. Identifying work units

The algorithm initially looks at each train arrival node and the next train departure node at the train arrival node's location. It then examines all the nodes between the train arrival node and the train departure node, and decides whether the train arrival node should be connected to the train departure node in a work unit.

When deciding whether to connect a particular train arrival node to a particular train departure node, the algorithm tests for the negative case. In other words, it looks for reasons why the two nodes should be left open for the possibilities network stage to connect them. Some situations that the algorithm looks for are illustrated in Figure 8-1.



(a) (b)

Figure 8-1: The first and last trains in each of these configurations will match one of the conditions for not forming a work unit.

The algorithm looks at each train arrival node and examines only the nodes between the train arrival node and the next train departure node. The train arrival node's train arc is called the arriving train, and the train departure node's train arc is called the departing train. The algorithm then performs a number of checks to find reasons why the arriving train and the departing train should not be connected. If after running the checks the algorithm has not found a reason not to connect the two trains, then the two trains will be connected. The exclusion conditions that the algorithm checks for not connecting two trains into a work unit are described below.

(1) If the arriving train is of a different locomotive class from the departing train, do not connect the trains as their locomotives are incompatible.

(2) Do not connect the trains if the arriving train requires a different number of locomotives than the departing train. If these trains were connected into a work unit, then in the middle of that work unit, the departing train may be required to be connected to trains outside the work unit to supply additional locomotives, or the arriving train might be connected to trains outside the work unit to allow surplus locomotives to run other trains. A work unit is the atomic unit of the schedule, and so it is invalid to have connections to and from other trains in the middle of a work unit. Work units must be self-contained.

(3) As depicted in Figure 8-1(a), the arriving train should not be connected to the departing train if the arriving train is not the latest train to arrive before the departing train departs, and at least one of the intermediate arriving trains is of the same locomotive class as the departing train. The reason for having this condition is, the departing train here has multiple resourcing options from which it could take a locomotive. It could take a locomotive from the first arriving train, or it could take a locomotive from one of the intermediate arriving trains. The algorithm has to choose one of the multiple arriving trains to connect to the departing train. As with the same-location merge algorithm, when there are multiple locomotives to choose from, locomotives are allocated on a last-in-first-out basis. See section 7.1 for an explanation of this. The first arriving train is not the "last in" for this case, and so no connection is made between the first arriving train and the departing train.

(4) As depicted in Figure 8-1(b), if between the arriving train and departing train there is a transfer departure node followed by a transfer arrival node, leave the trains unconnected. The transfer departure node allows all unallocated locomotives at the location to leave the location to run other trains, and the following transfer arrival node allows extra locomotives to return to the location to resource the departing train. The arriving train and departing train are left unconnected in this case so that the possibilities network phase can decide whether to resource the departing train with the arriving train's locomotive or with a locomotive that has been transferred from another location.

Given those four negative conditions which are tested to identify when two trains are not part of a work unit, we can deduce configurations trains that are connected into a work unit by this algorithm.

Each of the three panels in Figure 8-2 illustrates a configuration of trains and transfers which may appear in the schedule. All the trains in the figure are of the same locomotive class and require the same number of locomotives. The two trains in each of those panels would be run optimally when they are run by the same locomotive, and so they would be combined into a work unit. The reasons why the two trains in each of those panels should be combined into a work unit will now be explained.

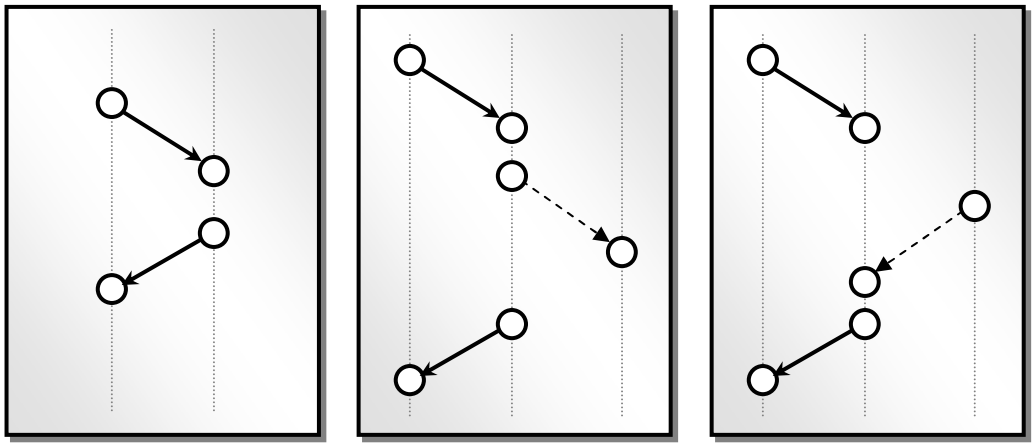


Figure 8-2: Configurations of train and transfer arcs that may be found in a space-time network. The two trains in each of these configurations all form one work unit.

Figure 8-2(a) illustrates a simple configuration where a train departure node immediately follows the train arrival node. In this configuration, the arriving train should be connected to the departing train because at the time of the train departure node, there will be one unallocated locomotive at the location from the train arrival node. This locomotive is the “last in” (see section 7.1) and so it is optimal to connect the two trains.

In Figure 8-2(b), even with the transfer departure node between the arriving train and departing train, it is still optimal to connect to the arriving train and departing train. This is because, no matter how many locomotives are transferred down the intermediate transfer arc in Figure 8-2(b), at least one locomotive must remain behind at the location to run the departing train as there is no other way to get a locomotive to the departing train. Since the locomotive for the arriving train is the “last in,” it is optimal to make it so that it is the arriving train’s locomotive that is left behind to run the departing train. So these two trains are connected into a work unit by the algorithm.

In the train configuration depicted in Figure 8-2(c), there is a transfer arrival node between the arriving train and the departing train. The two trains in this configuration still should be connected into a work unit. This is in view of the fact that at the point in time of the train departure node, there will always be one locomotive

available from the train arrival node. Since it is certain that the location will have enough locomotives to run the departing train, there is no reason to leave the configuration open for the possibilities network to decide whether to use the transfer arc for the departing train. So in this case, the arriving train will be connected to the departing train to form a work unit.

8.1.2. Work unit determination algorithm

The pseudocode for the work unit determination phase is displayed below:

```
For each location in set of all locations in the train schedule do
    Set node-list := list of all train and transfer nodes at location, in ascending time order;

    Set current-train-arrival-node := nothing;
    Set has-seen-transfer-departure-node := false;

    For each node in node-list do

        If node is a train arrival node then
            // exclusion condition (3) is checked here.
            /* the latest train arrival node will overwrite the previous train arrival node before it gets a chance
                to connect to the train departure node. */

            Set current-train-arrival-node := node;
            Set has-seen-transfer-departure-node := false;

        Else if node is a train departure node then
            Set current-train-departure-node := node;

            If current-train-arrival-node is not nothing then
                // exclusion conditions (1) and (2)
                If current-train-arrival node requires the same number of locomotives and the same locomotive
                    class as current-train-departure node then

                    Connect current-train-arrival-node to current-train-departure-node;
                End if;
            End if;

            Set current-train-arrival-node := nothing;
            Set current-train-departure-node := nothing;
            Set has-seen-transfer-departure-node := false;

        Else if node is a transfer departure node then
            If current-train-arrival-node is not nothing then
                Set has-seen-transfer-departure-node := true;
            End if;

        Else if node is a transfer arrival node then
            If has-seen-transfer-departure-node is true then // exclusion condition (4) matches.
                Set current-train-arrival-node := nothing;
                Set has-seen-transfer-departure-node := false;
            End if;
        End if;
    End for;
End for;
```

Algorithm 8-1

8.2. Identification of resourcing options phase

Before the possibilities network can be constructed, first the system needs to identify all the possibilities to connect the work units. In this phase, a resourcing options schedule is generated, which indirectly specifies all the possible **source work units** that may provide their locomotives to a **receiver work unit**.

The resourcing options schedule is set up as follows. For each work unit, the algorithm identifies all of the locations that the work unit's locomotives are able to transfer to after the work unit has been run. Transfers may be multi-stage transfers. The final location of the source work unit is also included as a transferable location. The algorithm then finds the earliest time that the source work unit's locomotives can arrive at each of the transferable locations. Other receiver work units that depart from one of these transferable locations after the earliest transfer arrival time at that location will have the option of reusing the locomotive of the source work unit. The source work unit is called a **resourcing option** for those receiver work units, and the earliest transfer arrival time at one of the transferable locations is called a **resourcing option node**. Each resourcing option node will have a sequence of transfers which will allow the source work unit's locomotive to arrive at the location of the resourcing option node at the earliest transfer arrival time. These resourcing option nodes are added to the resourcing options schedule.

Work units of different locomotive classes cannot connect to each other, and so there is one resourcing options schedule for each locomotive class in the system. Each resourcing options schedule contains a separate list for each location. Each of these lists contains all the resourcing option nodes for that location, sorted by their arrival time. Figure 8-3 is an illustration of the data contained in one resourcing options schedule.

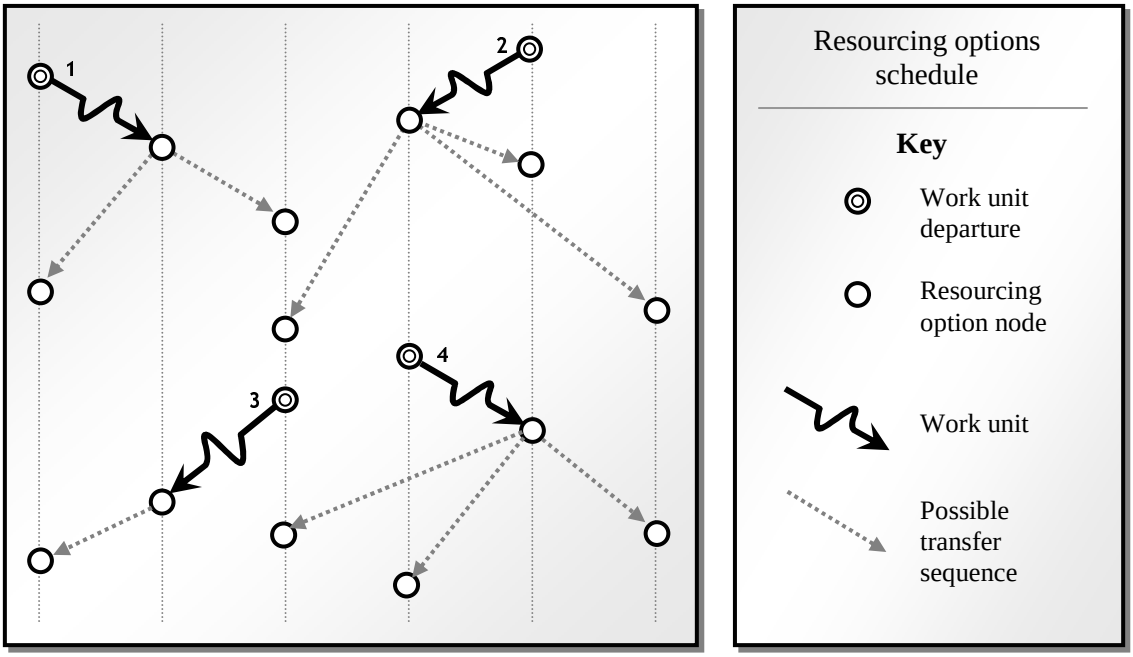


Figure 8-3: A diagrammatic form of the resourcing options schedule. Like the space-time network, time increases in the downward direction, and space is represented by the horizontal dimension.

Once the resourcing options schedule has been generated, it is possible for the algorithm to quickly identify all of its resourcing options for each work unit. For example, in the resourcing options schedule depicted in Figure 8-3, it can be seen that work unit 3 can be resourced by both work units 1 and 2. This is because both have resourcing option nodes at the departure location at work unit 3 that occur before the departure time of work unit 3. What this means is, work units 1 and 2 can be transferred to the departure location of work unit 3 before the required departure time. In the same way, we can see that work unit 4 can be resourced by work unit 2, as there is a resourcing option node before the beginning of work unit 4 at its departure location.

8.2.1. Generating the resourcing options schedule

To generate the resourcing options schedule, the algorithm uses a breadth-first search for each work unit. The algorithm begins with the arrival location of the work unit, and then expands to all the locations that can be transferred to from that arrival location. It then expands from the second layer of locations that have just been found, and so on.

The algorithm differs slightly from a breadth-first search. This is because a traditional breadth-first search is executed on a tree in which each node has only one parent, and in the train schedule, there may be multiple transfer sequences that could arrive at the same location. The algorithm handles this by remembering the current resourcing option node for each location. If the algorithm finds a new sequence of transfer

movements that can transfer a locomotive to a location earlier than the current resourcing option node for that location, the resourcing option node for that location is replaced with the earlier resourcing option node and the location is examined once again to see if there are new transfers that have spawned off this change.

A limit may be placed on the maximum number of consecutive transfers the algorithm allows. This is useful for two reasons. Firstly, it reduces the time required to solve the schedule, at the cost of the solution being less optimal. Secondly, rail operators usually avoid having too many consecutive transfers, as it only takes one late locomotive to break the chain of transfers. Setting a maximum of three consecutive transfers is a reasonable practical restriction.

To be efficient, the work units are processed in reverse end-time order. This is so that as locations are repeatedly visited to find all of the earliest possible movements from the locations, the system can cache and reuse the earliest transfers that are found at each location.

8.2.2. Identification of resourcing options algorithm

The pseudocode for the identification of resourcing options phase is below:

```
Set resourcing-options-schedule := empty resourcing options schedule;

Set last-start-time := latest start time of a work unit in the train schedule;
Set work-unit-list := list of all work units in order of their end time going from latest end time to earliest end
    time;
Remove all work units that have an end time after last-start-time from work-unit-list;

For each work-unit in work-unit-list do
    Set resourcing-options-to-expand := empty list;
    Set work-unit-resourcing-option-nodes := empty list;

    Set initial-resourcing-option-node := New resourcing option node;
    Set arrival location of initial-resourcing-option-node := end location of work-unit;
    Set arrival time of initial-resourcing-option-node := end time of work-unit;
    Set transfer sequence of initial-resourcing-option-node := empty transfer sequence;

    Add initial-resourcing-option to resourcing-option-nodes-to-expand;

    While resourcing-option-nodes-to-expand is not empty do
        Set current-resourcing-option-node := Retrieve and remove the first resourcing option node in
            resourcing-option-nodes-to-expand;

        Set movement-extensions-list := Get the list of all train and transfer arcs that depart from the arrival
            location of current-resourcing-option-node after the arrival time of current-resourcing-option-
            node;

        For each movement-extension in movement-extensions-list do
            Set new-transfer-sequence := Copy of the transfer sequence of current-resourcing-option-node;
            Append movement-extension to new-transfer-sequence;
```

```
        Set movement-extension-location := arrival location of movement-extension;
        Set movement-extension-time := arrival time of movement-extension;

        Set resourcing-option-node-to-location := Find a resourcing option node in
            work-unit-resourcing-option-nodes that arrives at movement-extension-location;

        If resourcing-option-node-to-location was not found then
            Set resourcing-option-node-to-location := New resourcing option node;
            Set arrival location of resourcing-option-node-to-location to movement-extension-location;
            Set arrival time of resourcing-option-node-to-location to movement-extension-time;
            Set transfer sequence of resourcing-option-node-to-location to new-transfer-sequence;

            Add resourcing-option-node-to-location to work-unit-resourcing-option-nodes;
            Add resourcing-option-node-to-location to resourcing-option-nodes-to-expand;
        Else
            If the arrival time of resourcing-option-node-to-location is after movement-extension-time then

                Set arrival location of resourcing-option-node-to-location := movement-extension-location;
                Set arrival time of resourcing-optio-noden-to-location := movement-extension-time;
                Set transfer sequence of resourcing-option-node-to-location := new-transfer-sequence;

                Add resourcing-option-node-to-location to resourcing-option-nodes-to-expand;
            End if;
        End if;
    End for;
End while;

    Add all resourcing options in work-unit-resourcing-option-nodes to resourcing-options-schedule;
End for;
```

Algorithm 8-2

8.3. Level assignment phase

The key to solving the possibilities network is to split the network into levels. A work unit in a particular level can only be resourced by work units in the levels that come before it. More specifically, the work units in the first level cannot be resourced by other work units, the work units in the second level can only be resourced by work units in the first level, the work units in the third level can only be resourced by work units in the first and second level, and so on.

The method for splitting work units into levels is similar to an existing algorithm called the topological sort (Black, 2004). Imagine a directed acyclic graph, where all work units are nodes and each arc represents a possible connection that can be made from a source work unit to a receiver work unit. All of the nodes that do not have any incoming arcs are added to level one, as each of these work units cannot be resourced by others. Now those level one nodes and all of their connections are removed from the network, and the algorithm again finds all the nodes that do not have incoming arcs anymore and adds them to level two, as they are the work units that could only be resourced by the level one work units which have now been

removed from the network. Following the same pattern, all level two nodes are removed and then the algorithm again finds the nodes that do not have any incoming arcs. These nodes are added to level three. Level three nodes are removed from the network, and the process continues until all nodes have been removed from the network, and all work units have been assigned levels.

8.4. Possibilities network phase

The possibilities network is a directed acyclic graph that presents which other work units can resource a particular work unit. As was introduced in the previous section, one of the fundamental ideas behind the possibilities network is the way it is organised and solved in levels.

The nodes in the possibilities network are work units. A work unit can be present in the possibilities network multiple times if it requires more than one locomotive. The number of times a work unit appears in the possibilities network corresponds to the number of locomotives required by the work unit. This allows the same work unit to be run by more than one locomotive if it is required.

The arcs between the various work units in the possibilities network represent connection possibilities – situations where the locomotive of a source work unit can be transferred to be used for a receiver work unit. The algorithm can find out what connections can be made between the work units by using the resourcing options schedule introduced in section 8.2.

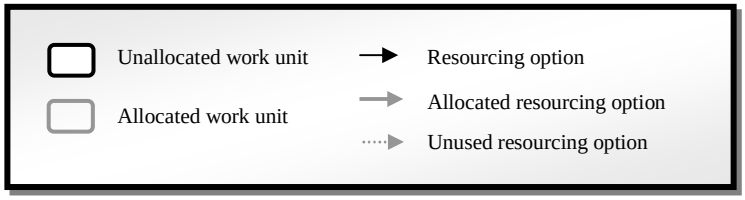
To construct and solve the possibilities network, the algorithm iterates over all the levels in the train schedule. At each level, all of the work units for the current level are added to the network. They will be the **receiver work units** being considered for that level. The **source work units** will be all of the work units in previous levels that are available to be allocated to one of the receiver work units. The algorithm will then solve the level by attempting to connect the receiver work units to the source work units. The method of solving each level is explained in section 8.4.2.

Note that work units are only receiver work units when their level is being solved. This means that if a receiver work unit cannot be connected to a source work unit when its level is being solved, there will be no more chances for the receiver work unit to be provided with a locomotive later on in the solution process. In this case, a new logical locomotive must be created for the receiver work unit.

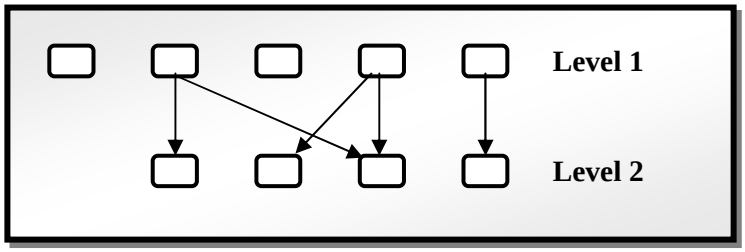
Once the possibilities network phase is complete, the algorithm will have many connected chains of work units. Each one of these connected chains is a locomotive diagram, and together locomotive diagrams form an allocated train schedule, which is the final output of the algorithm.

8.4.1. Possibilities network solution example

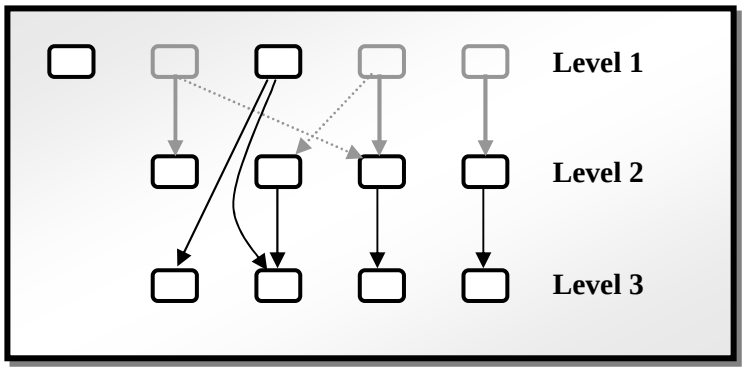
An example of the overall steps taken in the possibilities network phase of the work unit levels algorithm is described in this section.



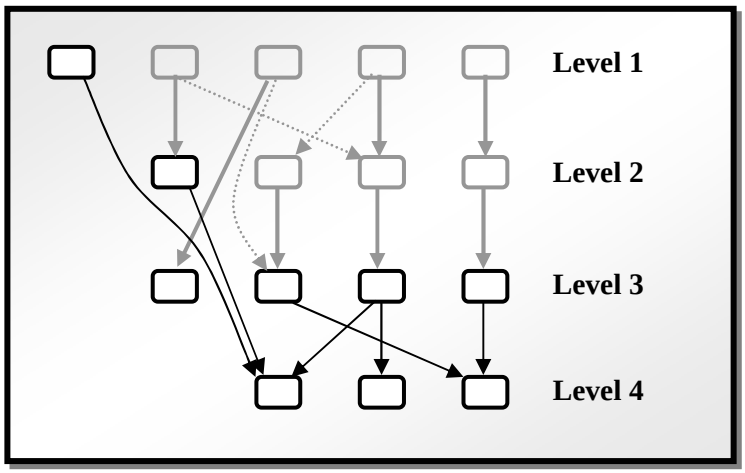
This is the key for the diagrams in this section.



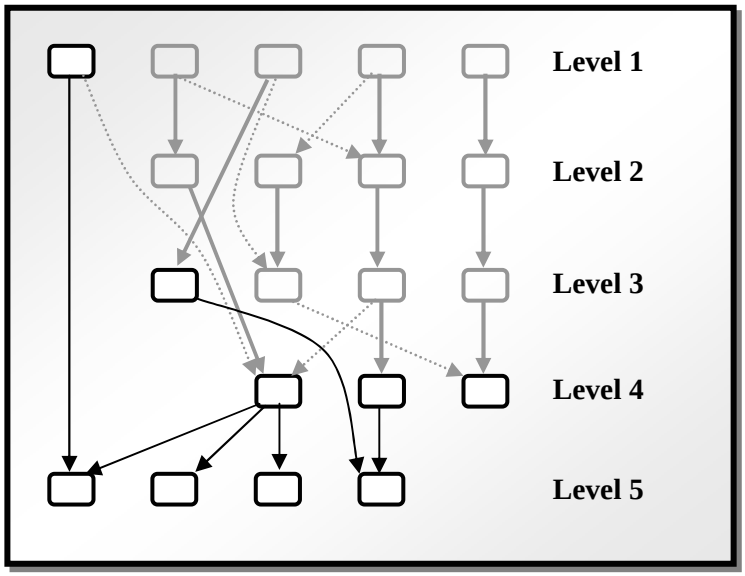
First the work units for level one and level two are placed into the network and resourcing option arcs are added.



The algorithm solves the level by connecting work units. Level three work units are added and resourcing option arcs for level three are placed.



The algorithm connects level three work units to work units in previous levels. Level four work units are placed and resourcing option arcs for level four are added.



The process continues in the same manner until all the levels have been processed.

8.4.2. Level solution algorithms

At each level, the algorithm attempts to connect work units of the current level to work units of previous levels. An algorithm has been developed that is designed to connect as many work units as possible in each level. If a work unit is not connected by the algorithm, then it is not being resourced with an existing locomotive, and so it requires a new locomotive to be added to the train schedule. The goal is to minimise the total number of locomotives, and to achieve this, the algorithm has to connect as many receiver work units as possible in each level that is solved.

To connect work units, the algorithm performs a number of steps, which are described as follows:

1. The algorithm first connects the source work units that can resource only one receiver work unit in the current level. This step and the next step eliminate work units which the algorithm has no choice about connecting.

2. Connect receiver work units that have only one resourcing option. If the algorithm is able to make a connection in this second step, it returns back to the first step, because some source work units may now have only one receiver work unit they can connect to after the second step runs.

3. Using a heuristic, choose one of the unconnected source work units that should be connected. Various heuristics can be used and some are explored in section 8.4.3. The source work unit that is chosen is then connected to any one of the receiver work units that it can connect to. There is no difference between choosing one receiver work unit and another receiver work unit. This will be explained later in this document.

4. Once the heuristic's chosen source work unit has been connected to a receiver work unit, the level solving part of the algorithm goes back to its first step, connecting any work units that only have one resourcing option as a consequence of this connection. The algorithm does not go to step one if there are no remaining source work units available to connect. If that is the case, the level is solved and the algorithm moves onto solving the next level.

For step three, there is no difference between choosing one receiver work unit over another receiver work unit to connect to a particular source work unit in terms of the final solution. This is because all of the receiver work units in the current level will become source work units in the next levels, independent of whether they have been connected or not. This means that, when the later levels are being allocated, the algorithm will still have the same choices regardless of what receiver work unit is chosen to connect to the source work unit. In the same way, no options are being closed off in the current level, as every receiver work unit will have at least one other source work unit that it can connect to. If there was a receiver work unit that could only connect to the source work unit chosen by the heuristic, then that connection would have been made in the previous steps.

In other words, when a source work unit has many receiver work units that it can resource, all options will result in the same number of total locomotives required in the end, assuming all other work units are connected the same way. Even though all the options in this case have the same end result, some options may create a more flexible train schedule, such as the last-in-first-out rule seen in section 7.1, and also the algorithm may choose to connect the receiver work unit that departs from the same location the source work unit arrives at so that a locomotive does not have to be transferred between locations.

Independent of the quality of the heuristic, the current level will always have the maximum number of receiver work units connected, if the connections of previous levels are taken as fixed. The choices of the heuristic only affect the levels downstream from the current level. This occurs because the algorithm constantly checks for and connects any work units that only have one resourcing option before moving onto the third step where source work units with multiple options are connected. Whenever the third step of the algorithm runs, the algorithm will never inadvertently remove all the resourcing options a receiver work unit has, because all the receiver work units will have at least two connection possibilities, and only one source work unit is being allocated each time.

A receiver work unit can end up unconnected due to an allocation made while the algorithm connects receiver work units that only have one resourcing option however. Each source work unit can only connect to one receiver work unit, and there may be multiple receiver work units that can only be resourced by one source work unit in the second step. If the work units are in a configuration that causes this situation to arise, then the algorithm can do nothing about this at the current level. The only way to change this would be to change the source work units available to the current level, and that would mean the heuristic making better decisions about source work units in previous levels. So overall, the algorithm is guaranteed to connect the maximum number of receiver work units when considering all the connection decisions of previous levels to be fixed. This means the heuristic is the sole factor that determines how well this algorithm performs.

8.4.3. Heuristics

When solving the possibilities network, the algorithm will sometimes have to choose which unconnected source work unit will be allocated a receiver work unit. It is important to make a good choice in this case, because once a source work unit has been connected to a receiver work unit, the algorithm has closed off the option of other receiver work units in later levels connecting to that source work unit. In the case where a receiver work unit can only connect to that one source work unit that is now already allocated, the receiver work unit would be required to use a new locomotive, increasing the number of locomotives required to run the train schedule. So, a good heuristic will choose to connect the source work units that will not require a receiver work unit in another level to use a new locomotive.

Fortunately, it is not too difficult to construct a good heuristic for this algorithm since all the possible resourcing options for each work unit are already known. This allows a simple heuristic to be built to find source work units that are least likely to force other receiver work units downstream to use a new locomotive. It should be stated why this situation cannot be seen ahead of time when all the resourcing options for every work unit is known. Most receiver work units, especially the ones in later levels, will have many resourcing options distributed throughout the train schedule. At each level, some of those resourcing options are used up by other receiver work units. The only way to detect whether a receiver work unit will have one source work unit option left when its level is solved is to know how the levels before it have been allocated, and this cannot be known until it comes to the time to solve the level of the receiver work unit. This means that a heuristic must be used. Two heuristics have been devised – the most-connected heuristic and the least-connected heuristic.

The most-connected heuristic chooses the source work unit that can be used to resource the most number of receiver work units, not only in the current level being solved, but in all the levels in the train schedule. The least-connected heuristic is the opposite, it chooses the source work unit that can be used to resource the least number of receiver work units in all levels of the train schedule.

Counting the number of possible receiver work units a source work unit has is relatively straightforward with the resourcing options schedule, as was explained in section 8.2. The only difference is, all receiver work units that have already been allocated a source work unit are ignored.

Applying and comparing both heuristics, it was found that the least-connected heuristic generated allocated train schedules that required the least number of locomotives. This was in line with expectations, as the source work unit that has the least number of possible receiver work units has less of a chance of being the only option for a receiver work unit in another level.

8.5. Implementation

An implementation of the work unit levels algorithm was created in JADE. The algorithm takes a train schedule as input using the class diagram described in section 5.2. This section will present the class diagrams used in each phase of the work unit levels algorithm. The class diagrams in this section contain only the relevant classes.

8.5.1. Work unit determination phase

The first phase of the algorithm is to determine the work units in the train schedule (see section 8.1 for a description of this phase). The relevant classes for this phase are illustrated in Figure 8-4.

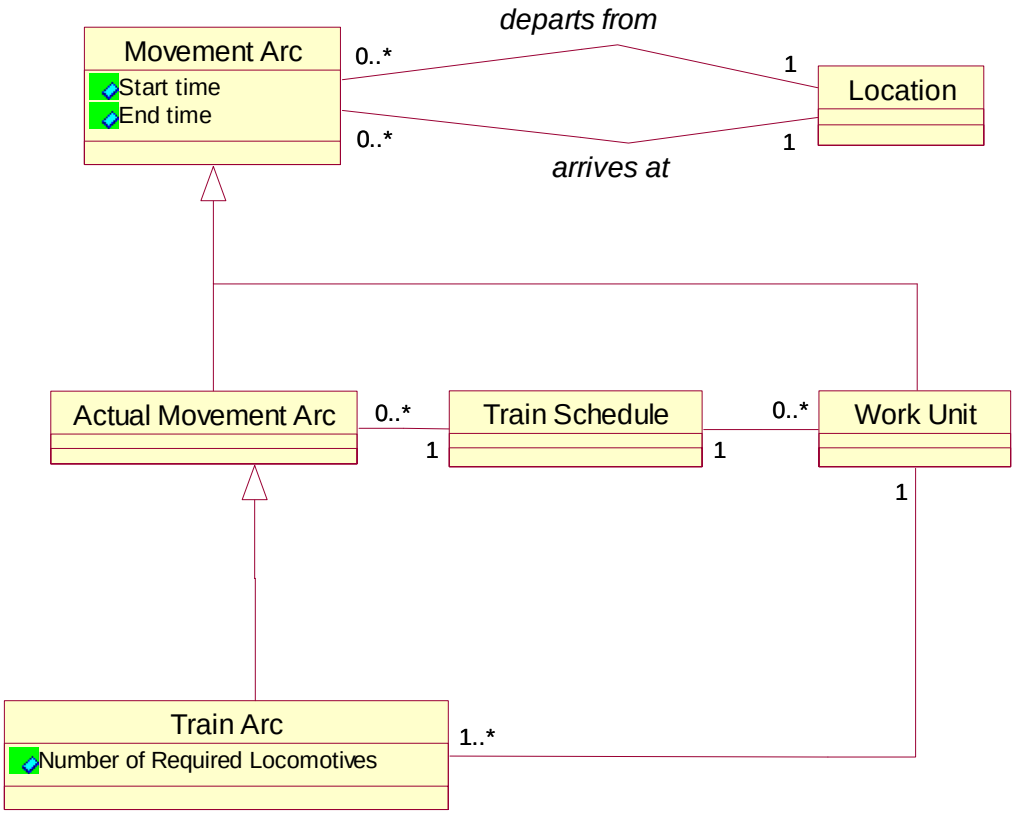


Figure 8-4: Class diagram for work unit determination phase.

When this phase completes, a number of work units will be created as part of the train schedule, represented by the one-to-many relationship between train schedules and work units. Each work unit is made up of a number of train arcs which are run optimally when they are run together by the same set of locomotives, and this corresponds to the one-to-many relationship between work units and train arcs. Work units inherit from the movement arc class, which means they have a departure time and location as well as an arrival time and

location. The departure properties are taken from the first train in the work unit, and the arrival properties are taken from the last train in the work unit.

8.5.2. Identification of resourcing options phase

In the next phase, the resourcing schedule is created, as explained in section 8.2. The classes for this phase are shown in Figure 8-5.

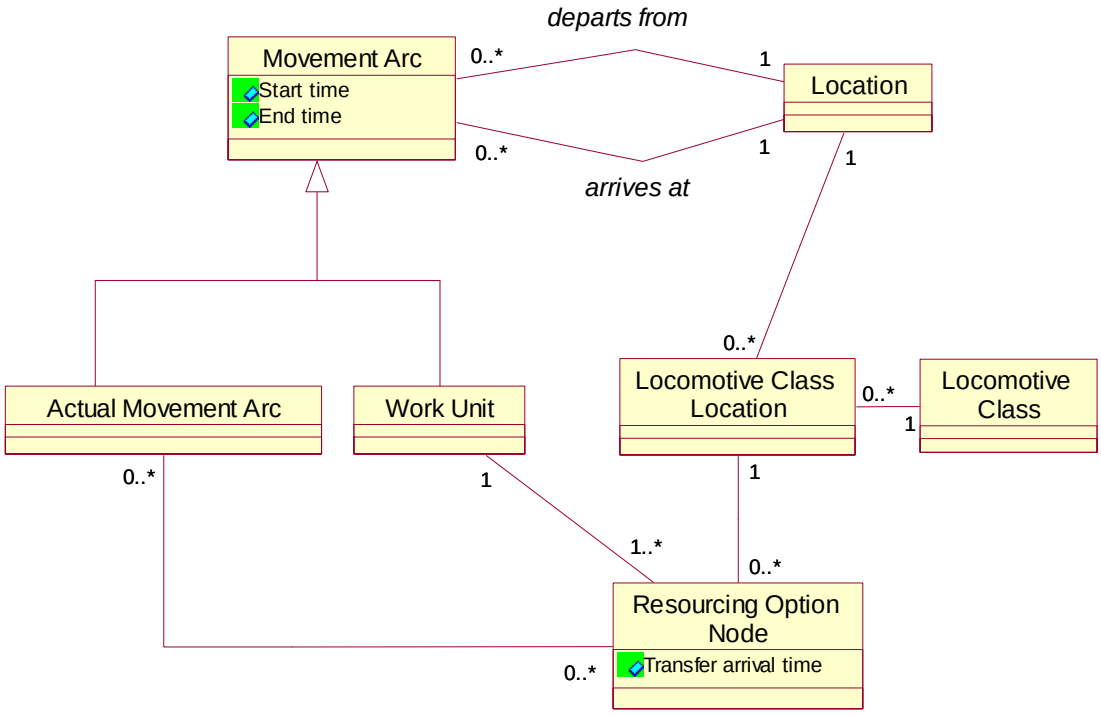


Figure 8-5

Recall that a resourcing option node contains the earliest possible time of the week that a work unit can transfer to a location. The transfer arrival time property of the resourcing option node class contains this earliest arrival time. Other work units that depart the location of the resourcing option node after the transfer arrival time could reuse one of the locomotives of the work unit for that resourcing option node, as long as both work units use the same locomotive class. The class called **Locomotive Class Location** refers to a particular combination of a location and a locomotive class. Each resourcing option node generated in this phase has a reference to the Locomotive Class Location object which contains the location the resourcing option node transfers to, and the locomotive class of the source work unit for that resourcing option node.

Each resourcing option node will have a collection of movements which can be followed by the work unit's locomotives to arrive at the destination location of the resourcing option node at the transfer arrival time. This is represented by the relationship between the resourcing option node class and the actual movement arc

class. The transfer sequence can contain dead-in-train movements on train arcs, or active movements on transfer arcs, so the relationship is to the actual movement arc class and not to one of its subclasses. Every work unit will have at least one resourcing option node created for the end time and location of the work unit, and this resourcing option node will not require any transferring of the work unit's locomotive. This explains why there is a cardinality of zero on the association relationship from resourcing option node to actual movement arc.

8.5.3. Level assignment phase

Using the resourcing option nodes found in the previous phase, the level assignment phase divides the work units into levels. The classes involved in this phase are presented in Figure 8-6.

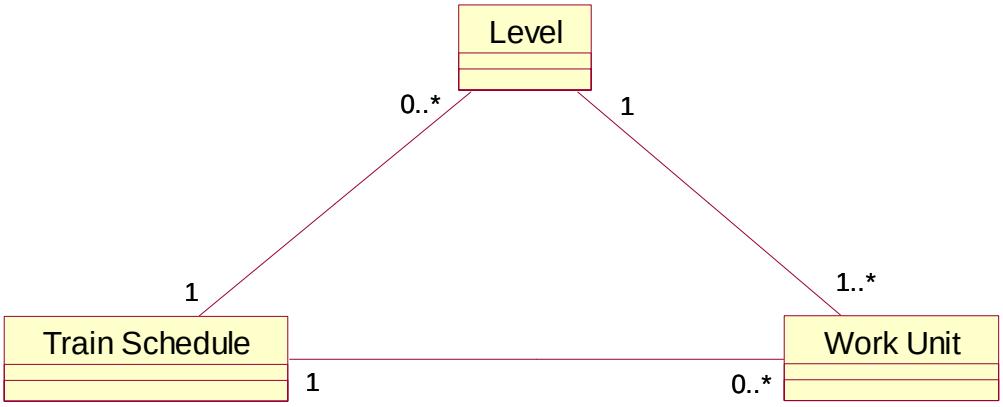


Figure 8-6

8.5.4. Possibilities network phase

The implementation of the possibilities network phase uses the class diagram shown in Figure 8-7.

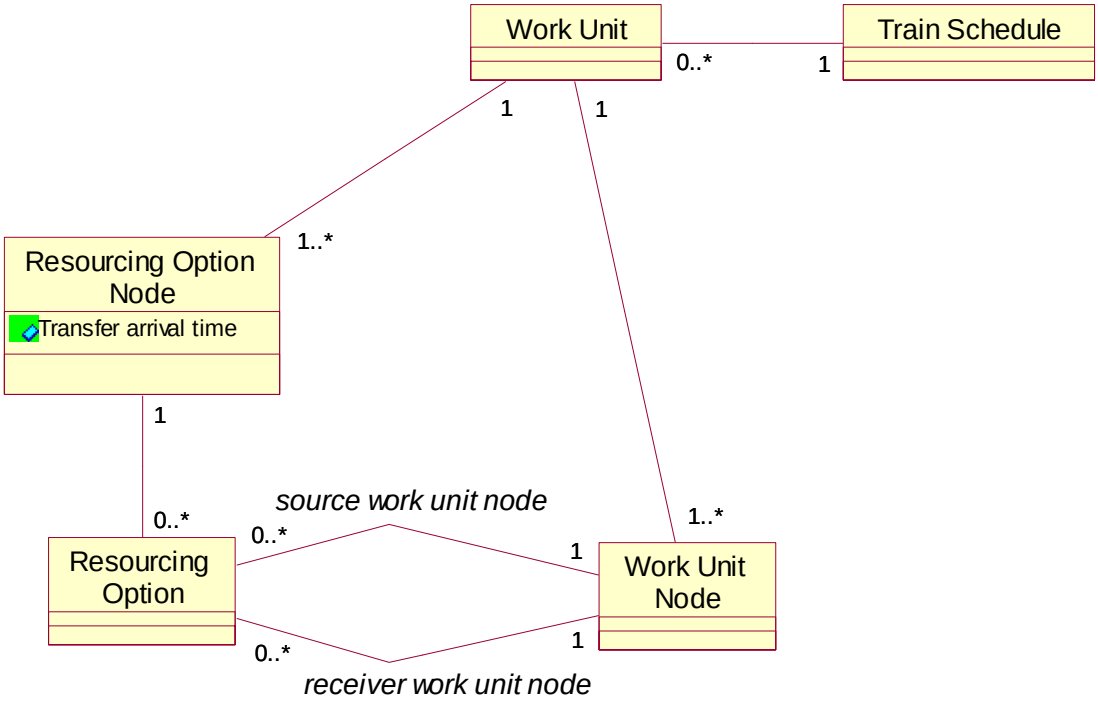


Figure 8-7

To begin the possibilities network phase, work unit nodes are created. Recall from the description of the possibilities network phase in section 8.4 that if a work unit requires multiple locomotives, it may have multiple nodes in the possibilities network. The number of work unit nodes each work unit has is equal to the number of locomotives it requires, so that the correct number of locomotives can be allocated to each work unit.

Resourcing options are represented as a class in the class diagram. Each resourcing option describes a connection that can be made between a source work unit node and a receiver work unit node. Each resourcing option also links to the resourcing option node that that stores the transfer sequence from which to get from the source work unit to the receiver work unit of the resourcing option.

As was described in section 8.4, the algorithm creates chains of work unit nodes by connecting work unit nodes at each level to work unit nodes in later levels. The algorithm does this level-by-level, where work units at the current level are called receiver work units and work units are previous levels are called source work units. Before a receiver work unit node has been connected, it will have a collection which may contain multiple resourcing options. After a receiver work unit node is connected, it will be set so that it contains

only one resourcing option in its collection, and that will be the resourcing option that connects the receiver work unit to the chosen source work unit. JADE's inverse references feature mirrors this on the other side of the relationship so that the collection of outgoing resourcing options from the chosen source work unit node is reduced to just the resourcing option that connects the source work unit node to the receiver work unit node. After a level has been solved, any unallocated receiver work unit nodes will have all of their incoming resourcing options removed to indicate that they were unable to be connected. These receiver work unit nodes are stored in a collection since they form the beginning of a new work unit node chain. On completion of this phase, the algorithm can generate full locomotive diagrams by traversing the chains of connected work unit nodes starting from each of the work unit nodes in this collection. Each sequence of trains arcs and transfers arcs in each work unit node chain will form one locomotive diagram.

8.6. Results

The work unit levels algorithm was applied to the same train schedule as the other solution approaches were. At first, multi-stage transfers of an unlimited length were permitted. The results of applying the algorithm under this condition are given in Table 8-1 and Table 8-2.

Phase	Cumulative Time Taken
Work unit determination	00:00:36
Identification of resourcing options	12:08:49
Level assignment	20:34:05
Possibilities network	30:36:40

Table 8-1

Measure	Value
Work units	6023
Levels	102
Number of locomotives required	496

Table 8-2

These results show that, the full algorithm took approximately 31 hours to complete and its final solution required a total of 496 locomotives. One observation that was made during the execution of the algorithm was that approximately 1.47 gigabytes worth of data had to be swapped out onto the hard disk, which would

have added an extra bottleneck to the execution of the algorithm, and so if the algorithm were run on a computer with a greater memory capacity, it would run substantially faster.

Having run the full algorithm with no limit on the number of consecutive transfers allowed, the maximum transfer sequence length was set to 3, as this is around the maximum length of a multi-stage transfer that a typical rail operator would use. The results for the algorithm this time are shown below:

Phase	Cumulative Time Taken
Work unit determination	00:00:22
Identification of resourcing options	00:05:47
Level assignment	00:06:31
Possibilities network	00:13:01

Table 8-3

Measure	Value
Work units	4841
Levels	95
Number of locomotives required	595

Table 8-4

Imposing a maximum length for multi-stage transfers resulted in the total solve time decreasing from 31 hours to 13 minutes, while the number of locomotives required increased to 595 from 496. Also this time, the system did not swap any data onto the hard disk.

The work unit levels algorithm produces very good train allocations with very few locomotives. However the real test is to see how this algorithm compares with the other solution approaches.

9. Comparisons of Techniques

Three quite different solution approaches were developed and tested in this project.

Solution Method	Solve Time	Required Locomotives
Integer programming solver	(unable to solve full train schedule)	
Same-location merge algorithm (one phase only)	0:00:53	1703
Same-location merge algorithm (two phases)	0:18:06	1058
Same-location merge algorithm (three phases)	2:17:19	1048
Work unit levels algorithm (max. 3 consecutive transfers)	0:13:01	595
Work unit levels algorithm (unlimited consecutive transfers)	30:36:40	496

Table 9-1: Comparison of all solution approaches

In Table 9-1, it is evident that the work unit levels algorithm is able to generate the solution with the fewest number of locomotives. Limiting the work unit levels algorithm to explore a maximum of three consecutive transfers allows it to generate a very good solution within reasonable time. It is also better than not limiting the number of consecutive transfers because it takes less time to execute, and most rail operators avoid scheduling more than three transfers in a row because planning too many transfer movements in a row means the schedule will become very sensitive to a locomotive arriving late and breaking the chain of transfer movements.

Even if it was feasible to apply the integer programming solver to the full train schedule, it would be very difficult, if not impossible, to constrain the integer programming solver so that only a certain number of consecutive transfer movements are allowed. Such a constraint is very difficult to express mathematically. This is one other advantage that the work unit levels algorithm has over the integer programming approach.

The work unit levels algorithm was built to overcome the primary weakness of the same-location merge algorithm, which was that the same-location merge algorithm has a restricted search space and does not consider all the possibilities. The work unit levels algorithm has succeeded, and although the first phase of

the same-location merge algorithm is able to produce a result in only 53 seconds, overall, the same-location merge algorithm has been made obsolete by the work unit levels algorithm, as the work unit levels algorithm can produce better solutions in less time.

9.1. Work Unit Levels versus Integer Programming Solver

Even though the integer programming solver was unable to solve the entire train schedule, the partial train schedules that it was capable of solving were also solved using the work unit levels algorithm to benchmark the performance of the work unit levels algorithm against the true optimum. In this case, the work unit levels algorithm has one restriction – only a maximum of 25 consecutive transfers have been allowed, which may mean the solution provided by the integer programming solver is more optimal than the work unit levels algorithm simply because of this additional constraint.

The results of the two solution approaches are compared in Table 9-2.

Minutes of Full Schedule	Optimal Locomotives from IP Solver	Work Unit Levels Locomotives	IP Solver Solve Time (seconds)	Work Unit Levels Solve Time (seconds)
600	8	8	0.01	0.22
750	10	10	0.26	0.38
900	16	16	0.34	0.53
1050	26	26	2.05	0.70
1200	40	40	12.69	1.04
1350	56	56	58.54	1.06
1500	66	66	411.89	2.23
1650	75	75	337.43	4.73
1800	90	92	1967.93	4.23
1950	109	110	5036.16	4.16
2100	134	137	4663.08	5.42
2160	147	150	8869.35	6.04

Table 9-2

For the smallest train schedules, the work unit levels algorithm requires fractions of a second more time than the integer programming solver to solve the train schedule. This is likely to be due to the amount of preprocessing that is performed as part of the work unit levels algorithm. The work unit levels algorithm is just as good as the integer programming solver for the partial train schedules that only contain the 1650 minutes worth of data or less, which is approximately the first 16% of the train schedule. At the point where the integer programming solver's limit had been reached with the 2160-minute partial train schedule, the work unit levels algorithm allocated only three more locomotives than the integer programming solver, but took only seconds to arrive at that solution. The 2160-minute partial train schedule was about 21% of the full train schedule.

This gap between the optimum and the solution produced by the work unit levels algorithm is likely to increase as the train schedule considered gets larger. However, the rate at which this gap increases with train schedule size cannot be easily determined. It is expected that the work unit levels algorithm will continue to allocate good solutions for larger train schedules, because the only part of the algorithm that can lead the solution away from the optimal is the heuristic. The least-connected heuristic that is used has all the information about how work units can be connected to other work units in the train schedule, and so it should be able to guide the algorithm to produce a near-optimal solution for much larger train schedules.

10. Conclusions and Future Work

There is increasing pressure on rail operators to reduce costs as they endeavour to compete against other available forms of transportation. The foundation of this project was a hypothetical train scheduling problem constructed by Jade Software Corporation, made to encapsulate the core challenges that a typical rail operator faces when allocating locomotives to their preplanned train schedules. The motivation behind the problem was to be able to assist in reducing the total number of locomotives a rail operator requires to run its train schedule each week. In resource scheduling, the problem is known as a minimum fleet size problem, as the objective of the project was to develop an algorithm that could allow a rail operator to run a train schedule with as few locomotives as possible.

A mathematical formulation was constructed to express the objective function and constraints. This precise mathematical definition served as a base for the three approaches that were taken to solve the problem. The solution approaches would be compared by applying all approaches to the same train schedule that contains approximately 7,200 trains and about 39,200 transfer possibilities.

The first approach was to use an integer programming solver. Using the mathematical formulation, a train schedule could be transformed into a mathematical representation and solved using an integer programming solver. It was found that the full problem was too large to be solved using the integer programming solver, but sections of the train schedule were small enough to be solved. This made it necessary to explore new ways of solving the full train schedule.

This lead to the development of the same-location merge algorithm. The primary characteristic of this algorithm was it makes trains use locomotives that are already at a location before using new locomotives, and attempts to improve solutions by merging locomotive diagrams. The major downfall of same-location merge algorithm was that if a train schedule was organised in a particular way the same-location merge algorithm would never be able to find the optimal solution to the train schedule as it would not consider all the possibilities for connecting trains.

To overcome the weaknesses of the same-location merge algorithm, the work unit levels algorithm was created. This algorithm combines trains into optimal sequences called work units, finds all the ways in which work units can be connected to each other, sorts the work units into levels according to their interconnection possibilities and finally connects the work units level-by-level in a possibilities network.

After executing the work unit levels algorithm, it was evident that it was a much better algorithm as it was capable of producing allocated train schedules that required fewer locomotives and also took less time to produce. The work unit levels algorithm was able to produce a solution to a train schedule within 13 minutes

that used 595 locomotives, which was a huge reduction from the 1058 produced by the same-location merge algorithm after an 18-minute solving process. Also when compared to the integer programming solver, the work unit levels algorithm demonstrated that it produced optimal solutions for train schedules up to 16% of the size of the full train schedule, and near-optimal solutions for the partial train schedules up to 21% of the full train schedule. Unfortunately, the integer programming tool was unable to solve more than 21% of the train schedule so the work unit levels algorithm could not be compared to see how close to optimality the solution it produced for the full train schedule was.

The work unit levels algorithm that was created for this project exhibits great potential to reduce the total number of locomotives that a rail operator requires to run its train schedules, but there are still areas in which further work could be beneficial.

The heuristic used by the work unit levels algorithm is the determinant of the quality of the solution produced by the algorithm, and so improvements to this heuristic would be highly beneficial to the rail operator. It would also be useful if the work unit levels algorithm were compared against other solutions to the problem or against real world train schedule allocations to benchmark the algorithm's scheduling capability.

Since this project explored a hypothetical problem which considered only the core features of the locomotive allocation problem, another direction for future work could be to add additional complexity to the problem, and investigate how the solutions used for this problem can be adapted. Some features which could be added to the problem could be allowing multiple locomotive classes to run the same train or to allow location group movements to be handled differently from paths.

Rail operators would also benefit greatly if the algorithm provided feedback on the way trains have been planned in the current train schedule, and was able to suggest how to plan the trains better for future weeks.

The success of the work unit levels algorithm in this project will be another step forward to reducing the costs of rail operators, and will strengthen the competitive edge railway has in the transportation market where other methods of transportation have become uncompromising competitors.

11. Appendix A: AMPL representation of space-time network

AMPL is a standard programming language used to express mathematical problems so that they can be solved by mathematical programming solvers. To input a problem to a solver using AMPL, users must provide two source files. The AMPL model file describes the variables, objective function and constraints to the problem. The AMPL data file expresses the values of the parameters for the model. Fourer, Gay, and Kernighan (1990) wrote a full description of the AMPL programming language, and Holmes (1995) has summarised the key features of AMPL in his paper.

11.1. AMPL model

The AMPL model for this problem is expressed below:

```
set K;

set TrDepartureNodes;
set EOSNodes;
set BalancedNodes;
set AllNodes := TrDepartureNodes union EOSNodes union BalancedNodes;

set TrainArcs within (AllNodes cross AllNodes);
set TransferArcs within (AllNodes cross AllNodes);
set ConnectionArcs within (AllNodes cross AllNodes);
set AllArcs := TrainArcs union TransferArcs union ConnectionArcs;

set S within AllArcs;

set I{AllNodes} within AllArcs;
set O{AllNodes} within AllArcs;

param r{k in K, (s,e) in TrainArcs} >= 0;

var x{k in K, (s,e) in AllArcs} >= 0;
var y{k in K, (s,e) in AllArcs} >= 0;

minimize locomotivesUsed: sum {k in K, (s,e) in S} y[k,s,e];

subject to ActiveLocosOnTrainArcs {k in K, (s,e) in TrainArcs}:
      x[k,s,e] = r[k,s,e];

subject to ActiveLocosOnTransferArcs {(s,e) in TransferArcs}:
      ((sum {k in K} x[k,s,e]) * (sum {k in K} y[k,s,e])) >= (sum {k
      in K} y[k,s,e]);

subject to ActiveLocosOnConnectionArcs {(s,e) in ConnectionArcs, k in
      K}: x[k,s,e] = 0;

subject to BalancedNodesConstraint {i in BalancedNodes, k in K}:
      (sum {(s,e) in I[i]} (x[k,s,e] + y[k,s,e])) = (sum {(s,e) in
      O[i]} (x[k,s,e] + y[k,s,e]));

subject to NonTrainDepartureArcsConstraint {i in TrDepartureNodes, k
      in K}:
      (sum {(s,e) in I[i]} (x[k,s,e] + y[k,s,e])) >= (sum {(s,e) in
      (O[i] diff TrainArcs)} (x[k,s,e] + y[k,s,e]));

subject to TrainDepartureArcsCreationConstraint {i in
      TrDepartureNodes, k in K}:
      (sum {(s,e) in I[i]} (x[k,s,e] + y[k,s,e])) <= (sum {(s,e) in
      O[i]} (x[k,s,e] + y[k,s,e]));
```

11.2. Example AMPL data file

Using the AMPL model, a space-time network can be input to any integer programming solver by specifying the data in an AMPL data file. An example AMPL data file for the space-time network presented in Figure 4-2 is expressed below:

```
set K := A B;

set TrDepartureNodes := 11 22 24 26 31;
set EOSNodes := 15 27 33;
set BalancedNodes := 12 13 14 21 23 25 32;

set TrainArcs := (11, 21) (22, 12) (24, 32) (26, 14) (31, 23);
set TransferArcs := (13, 25);
set ConnectionArcs := (11, 12) (12, 13) (13, 14) (14, 15) (21, 22)
      (22, 23) (23, 24) (24, 25) (25, 26) (26, 27) (31, 32) (32, 33);

set S := (14, 15) (26, 27) (32, 33);

set I[11] := ;
set I[12] := (11, 12) (22, 12);
set I[13] := (12, 13);
set I[14] := (13, 14) (26, 14);
set I[15] := (14, 15);
set I[21] := (11, 21);
set I[22] := (21, 22);
set I[23] := (22, 23) (31, 23);
set I[24] := (23, 24);
set I[25] := (13, 25) (24, 25);
set I[26] := (25, 26);
set I[27] := (26, 27);
set I[31] := ;
set I[32] := (24, 32) (31, 32);
set I[33] := (32, 33);

set O[11] := (11, 12), (11, 21);
set O[12] := (12, 13);
set O[13] := (13, 14), (13, 25);
set O[14] := (14, 15);
set O[15] := ;
set O[21] := (21, 22);
set O[22] := (22,12), (22, 23);
set O[23] := (23, 24);
set O[24] := (24, 25), (24, 32);
set O[25] := (25, 26);
set O[26] := (26, 14), (26, 27);
set O[27] := ;
set O[31] := (31, 23), (31, 32);
set O[32] := (32, 33);
set O[33] := ;

param r :=
[A, 11, 21] 1  [B, 11, 21] 0
[A, 22, 12] 1  [B, 22, 12] 0
[A, 24, 32] 0  [B, 24, 32] 1
[A, 26, 14] 1  [B, 26, 14] 0
[A, 31, 23] 0  [B, 31, 23] 1;
```

12. Appendix B: Integer programming solver results

These are the expanded results from section 6.3.

Minutes	Required Locomotives	Solve Time	Trains Arcs	Transfers Arcs	Locations
600	8	0.01	8	64	51
630	10	0.04	11	81	58
660	10	0.05	11	86	59
690	10	0.1	11	101	65
720	10	0.46	11	115	72
750	10	0.26	11	124	75
780	10	0.25	12	134	80
810	12	0.51	14	145	81
840	12	0.77	15	163	87
870	15	0.71	19	179	94
900	16	0.34	20	187	95
930	18	0.55	22	197	95
960	19	1.17	25	209	100
990	22	1.4	28	228	103
1020	25	0.9	31	250	111
1050	26	2.05	33	263	119
1080	30	3.57	36	283	129
1110	30	2.24	37	299	136
1140	31	11.44	38	325	148
1170	34	28.22	42	353	155
1200	40	12.69	51	390	163
1230	43	15.03	58	423	170
1260	47	19.17	64	452	179
1290	50	21.77	72	489	185
1320	53	19.22	79	541	205
1350	56	58.54	80	573	215
1380	57	150.78	84	628	229
1410	62	76.93	92	672	236
1440	63	91.8	97	705	239
1470	64	76.08	98	753	251
1500	66	411.89	102	793	261
1530	68	1019.74	108	843	271
1560	68	425.68	109	881	287
1590	72	1003.91	114	938	304
1620	73	684.64	116	980	311
1650	75	337.43	122	1044	328
1680	76	445.46	126	1111	341
1710	80	2033.69	135	1171	357
1740	83	2341.83	143	1238	373
1770	87	926.71	151	1316	393
1800	90	1967.93	163	1413	408
1830	92	1188.36	169	1491	423
1860	95	9655.18	173	1547	438
1890	100	4249.62	187	1648	454
1920	103	4992.61	197	1747	469
1950	109	5036.16	211	1835	483
1980	117	2254.04	225	1921	495
2010	119	4097.28	239	2014	508
2040	125	1398.68	251	2129	518
2100	134	4663.08	274	2316	536
2130	143	3412.57	292	2409	545
2160	147	8869.35	307	2503	548

13. References

Ahuja, R. K., Liu, J., Orlin, J. B., Sharma, D. & Shughart, L. A. (2002). *Solving real-life locomotive scheduling problems*. Gainesville, FL: University of Florida.

Bertossi, A. & Carraresi, P. (1987). *On some matching problems arising in vehicle scheduling models*. Networks, 17, 271-281.

Black, P. E. (2004, December 7). Topological sort. In Black, P. E. (Ed.), *NIST dictionary of algorithms and data structures*. Retrieved October 26, 2005, from http://www.nist.gov/dads/HTML/topologcsort.html

Czyzyk, J., Mesnier, M., and Moré, J. (1998). The NEOS server. *IEEE journal on computational science and engineering*, 5: 68-75.

Docherty, I. & Shaw, J. (2003). *A new deal for transport?* Malden, MA: Blackwell Publishing.

Dolan, E., (2001). *The NEOS server 4.0 administrative guide*. (Technical Memorandum ANL/MCS-TM-250). DuPage County, IL: Mathematics and Computer Science Division, Argonne National Laboratory.

Erlebach, T., Gantenbein, M., Hürlimann, D., Neyer, G., Pagourtzis, A., Penna, P., Schlude, K., Steinhöfel, K., Taylor, D. S. & Widmayer, P. (2001). On the complexity of train assignment problems. *Lecture notes in computer Science*, 2223: 390-402.

Fisher, M. L. (2004). The Lagrangian relaxation method for solving integer programming problems. In *Management Science*, 27(1): 1-18.

Fourer, R., Gay, D. & Kernighan, B. (1990) *AMPL: A mathematical programming language*. San Francisco, CA: Scientific Press.

Gropp, W. and Moré, J. (1997). Optimization Environments and the NEOS Server. In Buhmann, M. D. & Iserles, A. (Eds.) *Approximation Theory and Optimization*, pp. 167-182. Cambridge University Press.

Holmes, D. (1995). *AMPL (A mathematical programming language) at the University of Michigan, Documentation (Version 2)*. Retrieved October 12, 2005, from http://www-personal.engin.umich.edu/~murty/510/ampl.pdf

Kočvara, M. & Stingl, M. (2001). PENNON, a generalized augmented Lagrangian method for semidefinite programming. In Di Pillo, G. & Murli, A. (Eds.), *High performance algorithms and software for nonlinear optimisation*, pp. 303-321. Erice: Italy.

Kočvara, M. & Stingl, M. (2003). PENNON, a code for convex nonlinear and semidefinite programming. In *Optimization Methods and Software*, 18(3): 317-333.

Kokott, A. & Lobel, A. (1996). *Lagrangean relaxations and subgradient methods for multiple-depot vehicle scheduling problems*. (Technical report SC 96-22). Berlin, Germany: Konrad-Zuse-Zentrum fur Informationstechnik Berlin (ZIB).

Loebel, A. (1998). *Optimal vehicle scheduling in public transit*. Ph.D. thesis, Technical University of Berlin, Berlin, Germany. Aachen: Shaker-Verlag.

Scholz, V. (1998). *Knowledge-based Locomotive Planning for the Swedish Railway*. (Technical report T2000-05). Sweden: Swedish Institute of Computer Science.