Computation of Capacity on Railway Networks

Malin Forsgren

June 4, 2003

SICS

Swedish Institute of Computer Science, Kista

Master of Science Project Report

Supervisor (SICS): Jan Ekman

Supervisor (KTH): Thomas Sjöland

Examiner: Thomas Sjöland

Abstract

Banverket, the Swedish National Rail Administration, is interested in the development of a standard regarding how to assess capacity on railway networks. Jan Ekman and Per Kreuger at SICS AB propose that capacity on a section of a railway network for a specific traffic pattern can be assessed by computing its cycle time.

This report covers an account of my implementation of Ekman and Kreuger's cycle time algorithm. It also describes my suggestion of an algorithm that transforms an intuitive representation of a traffic pattern to a graph that can be used as input to the cycle time algorithm. My major contribution to the project is to facilitate the computation of the cycle time of a traffic pattern by having designed a bridge between the description of a traffic pattern and the format suitable for Ekman and Kreuger's algorithm.

The execution of these two tasks are important for an eventual continuation of the main project. My results constitute an important piece in a work that has the potential of making a real difference in the way in which capacity related issues in this field are handled.

Contents

List of Figures

1 Introduction

Two problems and their solutions constitute the subject for this report. Both are extensions of problems that are introduced and discussed in the report "En analytisk metod för utredning av kapacitet vid signalprojektering"[1] by Jan Ekman and Per Kreuger at SICS AB [9]. The report is a result of the project "Beslutsstöd för utredning av kapacitet vid signalprojektering"[2] for Banverket, the Swedish National Rail Administration.

This report is thus a consequence of that same project, but can be read independently of Ekman and Kreuger's report. Its focus is the implementation of the main results in the latter, as well as an examination into how to transform a description of a traffic pattern into a form suitable for capacity computation.

1.1 Capacity as a concept

There does not yet exist an accepted definition of capacity on railway networks. Capacity related issues in this context are accordingly treated in ad hoc ways.

Historically, capacity on railway networks (in any plausible meaning of capacity) has not always been a problem. There either were tracks connecting points a and b, or there were not. Today, railway traffic in Sweden is largely dependent on what goes on at the so called "wasp waist" – a bottleneck located in Stockholm, the capital of Sweden. It dictates what traffic can run on large portions of the rest of the railway network in the country.

In Sweden, capacity related questions in the context of railway networks have mainly surfaced in the last two decades. This explains why research on capacity issues appears to have been neglected: the problems have not become urgent until very recently. They are yet to be dealt with in more systematic ways. For a brief overview of related research conducted in other countries, see Section 2.2.

1.2 The need for capacity assessment

The laying of new railway tracks to meet future demands is an activity that has to be preceded by a certain amount of planning. The overall requirements are, due to various reasons, often vague. Different alternatives are evaluated by hand or by the use of simulation programs in order to find at least seemingly acceptable solutions.

Solutions done by hand are time-consuming and often hard to verify. Banverket presently relies on simulation[3] to get answers to some of their questions. However, simulation is not suitable for all kinds of questions to which answers are desirable, since the simulation model often addresses specific problems [13].

[1]The title can be translated to "An analytical method for assessing capacity during signal engineering".

[2]"Decision support for capacity analysis in signalling design"

[3]The simulation program used is called SIMON.

Simulation is also a time-consuming task, both when it comes to gathering data needed as input, and the actual run-time on a computer.

In connection to this I should mention that new tracks, and the reorganization of existing ones, are usually measures taken to satisfy a specific time table. Generally, the inevitable future changes to that time table are given little or even no thought.

Apart from the short-term economic reason, this approach has practical reasons. There is no standardized method of assessing capacity. Even verifying timetables is not a clear-cut task. What are the criteria for a timetable that 'works'? If this question cannot get a satisfactory answer, and we cannot say what is meant by capacity, how can we possibly say what a section of a railway network should look like to be adaptable to future changes in the time table? The only tracks built are therefore those that seem to be immediately justified by the time table planned for.

An analytic method of assessing capacity naturally has many advantages over the ad hoc ways used to assess capacity today. Simulation is the only structured method used, and its purpose is mainly to verify time tables. An analytic approach has the potential of being able to provide detailed, accurate answers to specific questions as well as verifying time tables, and doing this relatively fast. Wisely implemented, an analytic method outperforms simulation and opens doors to new and more efficient routines in the various phases of railway traffic planning.

1.3 Capacity and cycle time

For a detailed description of the algorithm that computes the cycle time of a traffic pattern, see Section 7. The purpose of this section is to give an overview of the area, and I assume the reader to be familiar with elementary graph and set theory. A quick review of the graph theory used in this report is nevertheless given in Section 4.1.

Until now, I have used the term *traffic pattern* in its intuitive sense, 'a pattern of traffic'. Ekman and Kreuger introduces a term, *trafikering* (Swedish), to denote the repetition of a pattern of traffic on a railway section [9]. From this point on, traffic pattern will always refer to a repeated pattern of traffic, and will be given a more exact definition later in the text. 'A pattern of traffic' will be used when I need to say something more general about the traffic and its behaviour.

1.3.1 Definition of capacity

Ekman and Kreuger view capacity of a traffic pattern on a section of a railway network as the inverse of the minimum time that has to elapse between the starts of two consecutive cycles of the traffic pattern. According to this point of view, capacity on a railway section is possibly (even most likely) different for different traffic patterns. Capacity is thus defined as *the throughput of trains* *on a particular section of the railway network for a specific, cyclic pattern of traffic.*

1.3.2 Traffic patterns

I will return to Ekman and Kreuger's definition of traffic pattern, and other important definitions, in Section 8.1. For the time being, it is sufficient to describe a traffic pattern as consisting of

- a set of train movements[4], on the specified section of the railway network, that make up *one cycle* of the traffic pattern, *and*
- a partial order on the set of movements that make up *two consecutive cycles* of the traffic pattern, called *the precedence relation*[5], representing how the movements are related to each other with regard to precedence

If two movements are related with regard to precedence, it means that one of them has higher priority, or precedence, than the other. *Partial* refers to the fact that the priorities among two arbitrary movements are not necessarily comparable. The precedence relation is *transitive*, which means that if movements b and c have higher precedence than movement d (denoted $b < d$ and $c < d$), and a has higher precedence than b and c, then a precedes b, c, and d. However, b and c remain unrelated, unless further information is given.

A traffic pattern is a pattern of traffic repeated an indefinite number of times, where the pattern of traffic can be described by one cycle of the traffic pattern. This explains why the definition of the precedence relation is not limited to the movements of just one cycle of the traffic pattern. For all traffic patterns with one or more movements per cycle, any movement in cycle n has higher precedence than its corresponding movement in cycle $n+m$, where $m = 1, 2, \ldots$

If the precedence relation is decided manually, it represents which trains are supposed to wait for which other trains on the railway section, and where. Another useful approach should be to automatically generate a lot of different sets of precedence relations and analyze what happens with the capacity. In particular, experimenting with different locations for the trains to meet or overtake each other, keeping the number and types of trains constant, should give interesting and useful results.

1.3.3 Constructing graphs

A traffic pattern can be expressed in a condensed way by letting the set of nodes[6] of a directed graph represent the set of movements of one cycle of the

[4]A *movement* is, from this point and onward, always a train movement unless otherwise specified, and will be defined formally in Section 8.1.2.

[5]A relation R on a set X is defined as a property which may or may not hold between two arbitrary elements of X. The relation is completely determined by the set of pairs that satisfy it. [5]

[6]*Nodes* are sometimes called *vertices*, and *arc* is a common term for a *directed edge* in a graph. For an overview of the graph theory used in this text, see Section 4.1.

pattern, and the set of arcs of the graph represent the precedence relation on the set of movements. To distinguish precedence over movements in the same cycle from precedence over movements in subsequent cycles, two types of arcs are used: straight arcs stand for the former type and bowed, or curved, arcs for the latter.

As suggested in the previous section, the number of pairs of movements that define the precedence relation in any traffic pattern is theoretically infinite. First of all, the relation is transitive. In addition to this, any movement has precedence over the corresponding ('the same') movements in subsequent cycles. This means that the train undergoing movement a in cycle n must start before the train (possibly physically another train than that of the first cycle) undergoing a in cycle $n+1$ can start, that is $(a, n) < (a, n+1)$ always holds.

An inevitable consequence of movement a in cycle n having higher precedence than a movement b in cycle $n+1$, is that a in cycle n also has higher precedence than movement b in cycle $n+2$. Thus, there is no need to introduce arcs that represent precedences over movements of any other cycles than the following cycle.

A traffic pattern graph represents the traffic pattern. It only shows the so called *direct precedence relation*, meaning that there are no arcs between any two movements for which the precedence relation can be deduced from the transitivity property.

All precedences can easily be derived from a graph displaying only the direct precedence relation. A movement represented by the node a has higher precedence than all other movements whose nodes there are *walks* to, from a, in the graph. Every bowed arc passed means that a new cycle, with respect to the movement that a represents, is entered.

1.3.4 Critical cycles and condition graphs

A graph displaying only the direct precedence relation is an unambiguous and compact way of describing the priorities of the different movements involved in a traffic pattern. Relationships other than direct precedences are expressed in graphs of similar appearances, called *condition graphs* by Ekman and Kreuger. The (weighted) condition graph is the most important type of graph when it comes to computing the *cycle time,* and finding the *critical cycle,* of a traffic pattern.

I now have to summarize some of the assumptions that Ekman and Kreuger make about the train movements if the following account is going to make sense. In short, they assume that all trains always move at the highest allowed (and possible) speed, and that a movement never starts until the train in question can complete the whole movement without any other train blocking its path or in any other way impeding its advance [9].

The nodes of a weighted condition graph represent the movements of the traffic pattern as above, but the arcs serve an additional purpose on top of the one already mentioned. The *weight* of the arc from node a to node b expresses how many time units must pass after movement a has started before movement b is allowed to start, with the assumptions mentioned above taken into consideration.

In other words, an arc in the condition graph represents a *condition* that must be met by the traffic of the traffic pattern. The set of arcs of a condition graph represents all such conditions that must be met.

The *weight* of a *path* is the sum of the arc weights in the path. Ekman and Kreuger define the *cycle time* of a weighted graph representing a traffic pattern to be the *maximum cycle mean* of the cycles in the condition graph, where the cycle mean of each individual cycle is found by dividing the weight of the cycle by the number of bowed arcs in the cycle. The cycle with the greatest cycle mean, or cycles if more than one cycle has the same (maximum) mean, is called the *critical cycle*.

1.4 Problem statements

With the introduction to the area of research fresh in mind, here are the problems I have solved:

1. Write a computer program that, given a weighted condition graph, computes the cycle time of the traffic pattern represented by the graph.

2. Given a traffic pattern, find an algorithm that transforms the description of it to a condition graph, suitable for the algorithm implemented in the previous problem.

1.5 An outline of this report

This introductory section is followed by Section 2 that briefly describes what has already been done in this field, and also mentions some ongoing projects that are related to my work. Methodological considerations are discussed in Section 3. Section 4 provides an overview of the mathematical concepts used to compute the capacity on a railway network.

The description of the implementation (Sections 6 and 7) is preceded by a brief summary of what SICStus Prolog is, and how logic programming differs from other programming paradigms (Section 5). A suggestion on how to deal with the second problem (see Section 1.4) is given in Section 8. An analysis of the solutions – with an emphasis on the second problem – and the conclusions I can draw from this, are presented in Section 9.

2 Background

As I have already mentioned in the introduction, this field of research is fairly young. Analyzing bottlenecks and planning for increased railway traffic and improved capacity on established stretches are however not problems unique to Sweden. Although the method developed by Ekman and Kreuger is brought into line with the needs of Banverket and the operation of railway traffic on the Swedish national railway network (as opposed to for instance the subway network of Stockholm), capacity issues for all traffic carried by tracks share important characteristics.

2.1 Related methods

For small networks that are built for, and used by, a small number of different types of vehicles, capacity analysis might not be hard to carry out by hand. Naturally, the procedure differs from nation to nation (and from company to company), but the basic approach is similar independently of who carries out the analysis.

Capacity analysis and time-tabling walk hand in hand on small networks since it in this case is possible (even desirable) to consider the whole network at once. An assessment of capacity can be done in connection with the simultaneous performance of train assignment and time-tabling by considering an unlimited asset of trains and setting the goal to be 'as heavy traffic as possible'. Note that capacity and 'heavy traffic' needs to be defined before the assessment can be really useful.

When it comes to larger networks, for which capacity analysis, train assignment and time-tabling are all really complicated tasks, efforts have been made in the development of simulation programs. This is a world-wide trend, and Banverket currently uses the simulation program SIMON[7] to assess capacity (among other things).

Simulation in the context of capacity assessment is briefly discussed in "Railway Capacity Assessment, an Algebraic Approach" [8]. The report says that simulation is a strong tool for evaluating time tables, but provides less insight than analytical tools, which makes it less suitable for optimizations than analytical methods. Time consumption is also an issue: the more details required, the more time it takes to model the simulation and the more time the actual simulation takes.

2.2 Related work

The technical development in the railway industry has kept pace with the rest of the technical advances made in the last century. There is a huge difference between steam trains and the train models of today, as well as between the manual operation of train meets of the past and the advanced signalling systems of today.

[7]SIMON is not the same system as Simone, developed in the Netherlands [14].

As opposed to the areas just mentioned, that have received continuous attention and improvements, railway capacity assessment has emerged as an increasingly troublesome problem only in the last 30 years or so. There does not exist any accepted method for capacity assessment. There does not even exist an accepted definition of capacity. Nevertheless, this subject has been addressed several times in the past, with various results.

2.2.1 Railway Research initiated by the EU

The European Union is currently funding a significant number of research projects in the railway sector, of which IMPROVERAIL – IMPROVEed Tools for RAILway capacity and access management – is one. The project spans over two years and is divided into eight work packages. One of the overall objectives of IMPROVERAIL is to provide improved methods for capacity and resource management. One of the work packages, WP5[8], is focused on exactly these questions.

The final report from WP5, a project that should have been finished in the autumn of '02 according to the latest updated schedule, is yet to be delivered. The expected result from WP5, according to the inception report of IMPROVERAIL written in October '01, is "a number of planning methods reflecting best practices and best research results" [21]. This result is to be obtained by examining scientific literature emanating from research about capacity management as well as evaluating capacity management methods that are actually used on railway networks in different countries.

IMPROVERAIL gives an overview of literature and various methods relevant to WP5 in a report called "State of the Art in Railway Infrastructure Capacity and Access Management", delivered in April 2002 [20]. The report establishes that a common objective to very many studies undertaken has been to find a suitable definition of capacity. The definition chosen obviously determines the characteristics and usefulness of the developed method or tool.

The methods examined range from simulation to analytic models, and common to most of them is that they give their results as percentages of total capacity, or in number of trains for a given time period. Although they generally take into account many different parameters, IMPROVERAIL concludes that most seem to fail when it comes to providing a broad perspective of capacity for the whole network. The solution might be to refine and separate the scopes and the different needs of each model, and to use different models for different purposes:

> Concerning capacity assessment, it can be said that long term and medium/short term capacity evaluations lead to two different models, as each planning horizon needs its proper set of information. Moreover, one certainly expects more in-depth results from a short term than from a long term evaluation method. Thus two separate models should be developed within WP5. [20]

[8]The 5th work package, "Methods for capacity and resource management".

2.2.2 Existing tools and methods

Of the different tools reviewed by IMPROVERAIL, there are a few that are used in several countries. VIRIATO (developed by SMA and Partner, 2001) is at least used in former Czechoslovakia, Denmark, the U.K., Portugal, France, Finland, Italy and Estonia. It is mostly a time-tabling tool, but allows the user to determine the level of saturation of a specified line, in percent. CAPRES takes into account junctions and station characteristics, and determines capacity on a railway network, not just on a single line [6]. It is used in Switzerland, France, Italy, and the U.K..

Another tool worth mentioning is RAILCAP, developed by Stratec, a consulting company in Belgium. The initial software that was developed requires detailed description of the tracks (down to block level), the position of switches, crosses, and signals (including their type) as well as the speed limits. The succession of trains is described by their routes across the network, the stops they make, and the length, maximum speed, and acceleration rate of each train. The method measures, through simulation, how much of the available capacity on different sections of the network that is actually used, by computing during how much of the available time the specified section is occupied by trains.

The great detail of analysis of bottlenecks enabled by RAILCAP has one major disadvantage in the fact that the modelling requires a lot of effort. Stratec developed another model in 1999 that makes simulations less time-consuming to prepare. According to Stratec, results from the simplified model have been counterchecked with the more sophisticated Railcap model and proven to come very close to those obtained with the latter [17].

The research program named "Seamless Multimodal Mobility" (SMM) at the Netherlands Research School for Transport, Infrastructure and Logistics (TRAIL) is surprisingly enough not even mentioned in IMPROVERAIL's overview of current research. The main objective of the research program is to "provide tools for the design and operation of attractive and efficient multimodal passenger transport services" [8]. Project 3 of the research program, "Dependable Scheduling", has given rise to several reports, among which "Railway Capacity Assessment, an Algebraic Approach" is the most relevant to my work. SMM is still an active research program, and the latest results from it were recently presented at the TRAIL Congress (November 26, 2002) [8, 10, 11].

The algebraic approach presented by TRAIL is based on maxplus algebra. The authors of "Railway Capacity Assessment, an Algebraic Approach" point out that the algebraic approach can be used as a simulation model, but that it was developed primarily as an analytic tool. Throughput and punctuality are investigated in a deterministic way while the train movements are fixed. Network capacity and robustness are not entirely separated. Instead the former is optimized by finding a set of buffer times that obeys robustness constraints and maximizes the throughput.

Last but not least, there is research that aims to bridge the gap between simulation that requires detailed modelling, and analytic tools that tend to focus on local problems and thus fail to give a broad assessment of capacity. The

Canadian National Railway has developed a parametric model that measures theoretical, practical, used, and available track capacity. These four types of capacity, and numerous parameters identified as affecting capacity, are defined.

Theoretical capacity works as an upper bound of capacity: it "assumes all trains are the same, with the same train consist, equal priority, and are evenly spaced throughout the day with no disruptions. It ignores the effects of variations in traffic and operations that occur in reality" [12]. Some parameters affecting traffic are *Intermediate Signal Spacing Ratio* (ISSR), *Percent Double Track* (% DT), and *Traffic Peaking Factor* (TPF).

All parameters are given fairly exact definitions, although the choice of definition sometimes seems arbitrary. The Traffic Peaking Factor is for instance defined as the ratio between the maximum number of trains dispatched in a 4-hour period and the average number of trains for that time period. According to the Canadian National Railway, this model can be used to identify bottlenecks and sections of excess capacity. I have however not found any evidence of the actual application of the model, and can therefore not say whether the model was abandoned or actually used.

2.3 The intended reader

Although this report can serve as a teaser to Prolog, its intention is not to give a comprehensive introduction to this programming language. Sections 6 and 7, that is, the part of this report that describes the implementation of the results, require some programming skills to be meaningful. The reader's programming experiences do not necessarily have to include Prolog or any other logic programming language, though.

The reader also has to be familiar with linear algebra, and some concepts of discrete mathematics (especially graph theory) in order to fully understand this text.

Furthermore, this report can be read independently of the report written by Ekman and Kreuger, [9]. Areas that are of great interest also to my work, are covered in both reports. Most issues – especially the details – from Ekman and Kreuger's report that are not relevant to my study, have however been omitted in my report. To fully understand the context, I recommend the reader to read their report as well (the report is in Swedish). The problems that I have solved are extensions to the problems tackled by Ekman and Kreuger, and are naturally accounted for only by me.

3 Method

3.1 Implementation

My philosophy when it comes to programming is that the highest priority in any programming project should be a stable and well-documented program. Most optimization considerations can stand aside while this first goal is met: code can always be optimized, but there is no use in optimizing a code that does not do what it is supposed to do.

There is no need to be deliberately careless in the choice of overall approach, however, since this in itself can render the entire (though well-documented) code more or less useless because the most natural and effective optimization would be to change approach altogether.

Once a reasonable approach has been chosen, it might for many purposes be wise to initially make the implementation easy to follow instead of focusing on finding ways to save some running time and space. These potential optimizations might of course become important later in the project, but writing code is to a great extent similar to writing a text: there is always room for improvement, and it is really up to the author to decide when – if ever – the code/text is finished. My experience is that it is better to make a program, or text (to continue the analogy), work as a whole entity, instead of aiming to perfect every part before putting the pieces together.

As a direct consequence of my philosophy described above, a large portion of my implementation is true to the ideas presented by Ekman and Kreuger in their report in an almost literal sense. I chose to use the approach described by Ekman and Kreuger (although I was of course free to use any approach that solved the problem) because it is easy to follow, and because its complexity is low. Even if there might exist an approach with lower overall complexity, the difference between such an approach and Ekman and Kreuger's approach would not be very dramatic, and there was thus no great risk involved with embracing their approach.

3.2 Providing suitable input

3.2.1 The condition graph

There are, as I see it, three major reasons for using and implementing Ekman and Kreuger's approach towards capacity assessment. The one I mention last below is directly connected to one of the problems I was asked to look at (see Section 1.4), and the way in which I approached that problem is described in the next subsection, 3.2.2.

The first reason for favouring Ekman and Kreuger's approach is that it provides an exact result, with respect to the input. Provided the input is 'exact' in a sensible sense, it is straight-forward and unambiguous what the method measures. This is for example not the case with simulation. Note that, in this context, I have adopted Aurell and Ekman's view that simulation is synonymous with a numerical experiment [1]. Their motivation for this assumption is basically that very few people that use simulation programs know exactly what the programs do on a detailed level. Using a simulation program can therefore be likened to performing an experiment.

The second reason is that the complexity of all algorithms used by the approach is fairly low. Time is spent on defining the traffic pattern, feeding the definition of the traffic pattern into the program, and finding the weights of the arcs of the resulting condition graph – not on waiting for the computer program to return a result.

Last, but not least, the definition of the traffic pattern governs how much data, and exactly what data, that needs to be collected. It is obvious that a reliable capacity assessment with Ekman and Kreuger's approach cannot be obtained in any other way than by generating weights of the arcs of the condition graph that correspond to how the train movements in the traffic pattern would affect each other in real life. Finding these arc weights is a laborious task, and it is therefore of uttermost importance that as few of the arcs in the condition graph as possible are redundant – preferably none.

If no arcs (or possibly a negligible number of arcs) are redundant, then it is reasonable that the cycle time of the traffic pattern in question cannot be found with significantly less effort than the effort that is asked from us by this approach.

3.2.2 Identifying redundant arcs

Of course the goal should be to provide a way of translating a description of a traffic pattern to a condition graph representing precisely the conditions – one per arc – that are necessary for the computation at hand. No more and no less. If this goal cannot be met, the only feasible alternative is to generate (possibly too many) arcs while being able to conclude that the ones that are generated form a sufficient set on which a computation of the cycle time can be based.

Finding a set of arcs that is sufficient to correctly compute the cycle time of the traffic pattern is actually not hard at all. The most naive and obvious choice is to let every pair of movements related with regard to precedence (see Section 1.3.2), among all movements in any two consecutive cycles, result in an arc. A sufficient set of conditions is of course represented by the set of arcs generated in this way.

I used a better approach. I noted that certain categories of conditions are at a very early stage easily judged as generating obviously redundant arcs, and can be dismissed immediately during the process in which the first set of arcs for the condition graph is generated. Having done that, it is then possible to examine the relationships between the movements a bit closer and eliminate more redundant arcs.

The more redundant arcs that can be eliminated, the better. But the elimination of redundant arcs must be carried out in such a way that there does not exist any risk of eliminating an arc that must not be eliminated, 'by accident'.

4 The Maximum Mean Cycle Problem

Although there are a few different approaches to computing the cycle time of a traffic pattern, the approach that I have investigated concludes with computing the maximum cycle mean of a so called *collapsed graph.* The maximum cycle mean of a directed graph is, loosely speaking, the average weight of an arc in the cycle with the largest average arc weight in the graph.

4.1 Some graph notation[9]

Let D be a weighted, directed graph (*digraph*) with nodes V, arcs A, and a weight function $w : A \to \mathbb{R}$, where $\mathbb{R}$ denotes the real numbers. A is a set of *ordered pairs* of nodes. In graph literature, a *walk* W in D is commonly defined as an alternating sequence $W = x_1a_1x_2a_2x_3 \ldots x_{k-1}a_{k-1}x_k$ of nodes x_i and arcs a_j from D, in which every arc a_i starts at node x_i and ends at node x_{i+1}. Neither the arcs nor the nodes in W have to be distinct.

The *length* of a walk equals the number of arcs in it. A walk can be of zero length, in which case the walk consists of a single node, $W = x_1$. A node y is *reachable* from a node x in a digraph D if there exists a walk from x to y. A digraph D is *strongly connected* if and only if every node of D is reachable from every other node of D.

A *path* is a walk in which nodes as well as arcs are all distinct. If nodes and arcs are distinct in a walk W except for the starting node and ending node being the same, and the length of W is at least two, W is a *cycle.* A *loop* is a walk of length one consisting of an arc from a node to itself. It is usually not defined as a cycle, but will from this point on, in the context of computing the maximum cycle mean of *collapsed graphs*, be regarded as if were it in fact a cycle. The justification for this is that loops in collapsed graphs generally represent cycles in the associated original graphs, and that we are of course equally interested in all cycles when we compute the cycle means.

A weighted digraph – in this text, all graphs of interest are directed graphs – has a (real numbered) weight associated with every arc. The *weight of a walk* is the sum of the weights of the arcs the walk traverses, the weight of a particular arc counted every time the arc is traversed if the arc occurs more than once in the walk.

4.2 Event graphs

The terminology in this section is taken from an article by Ali Dasdan and Rajesh K. Gupta called “Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis” [7]. Other authors may of course use different names for the properties and quantities defined here, although Dasdan and

[9]This section provides a short summary of graph concepts that I will use in this report. There is nothing unconventional about the notation introduced here, but for reference, I have mainly used [3, 4].

Gupta's terminology seems to coincide with the prevalent terminology in this field.

4.2.1 Discrete event systems

Discrete event systems can be modeled by directed, weighted graphs (*event graphs*) whose nodes and arcs represent the events and the interactions between them. The events are repetitive, and we may assume the graph to be strongly connected without loss of generality[10] .

The arcs and their weights together hold information about when events are allowed to *fire* (for example start) in relation to each other: an arc with weight w starting at node x and ending at node y, says that event y must not fire earlier than w time units (for instance minutes) after event x fired. In some systems, an event can fire when one of its predecessors enables it. For other systems – such as the one in focus for this report – an event has to wait until all its predecessors have enabled it.

In the simple model just referred to, it is not possible to make a firing time of an event dependent on earlier firing times of its predecessors other than just the latest ones. By associating with every arc another value (in addition to the arc weight), called the *occurrence index offset* and denoted by τ, it is possible to let event firings depend on a longer history. To see how this works, compare the different ways of computing the firing times of individual events in the two different models in the sections below.

4.2.2 The maximum cycle mean

The *cycle mean* of a cycle C in a weighted digraph D is defined as the weight of C divided by the length of C[11]. Accordingly, the *maximum cycle mean* is defined as the maximum of the cycle means over all cycles in the digraph D, including the loops. Let $\lambda(C)$ denote the cycle mean of cycle C, and $\lambda(D)$ the maximum cycle mean of a digraph D. Then $\lambda(D)$ can be expressed as

$$\lambda(D) = \max_{\forall C \in D} \{\lambda(C)\} .$$

[10]Any graph G that is not strongly connected consists of a number of strongly connected components that can be considered separately. If the maximum cycle mean of G is to be computed, the answer simply equals the largest maximum cycle mean of the different components of G.

[11]Compare this definition with the definition of cycle mean in Section 1.3.4. They differ, and the reason is that the latter is Ekman and Kreuger's definition of *cycle mean in a condition graph*, which is actually equivalent to the *profit to time ratio*. See Sections 4.2.4 and 4.3 for definitions and the background to this confusion of terms.

4.2.3 Firing times and cycle time

To determine the earliest possible firing time of an event in a system in which every event has to wait for all its predecessors to enable it before it can fire, Dasdan and Gupta provide the *max-causality rule*:

$$s_i(k) = \max_{j \in Pred(i)} \left\{ s_j(k-1) + w_{ji} \right\}, \ k > 0.$$

In the expression above, $s_i(k)$ is the firing time of event i when it fires for the $(k+1)$st time, $Pred(i)$ denotes the set of the *immediate predecessors* of i, and w_{ji} is the weight of the arc from j (a predecessor of i) to i. The immediate predecessors of i are the events whose nodes have arcs to i.

Every event fires independently of each other the first time, so $s_i(0)$ is determined separately. Dasdan and Gupta call this property the *independence rule*.

The *cycle time* of a single event i in a system abiding by the independence and max-causality rules above is defined as

$$T_i = \lim_{k \to +\infty} \frac{s_i(k)}{k}.$$

Loosely speaking, T_i says with what time interval event i fires, on average, in the system described by the event graph. The cycle time T of the whole system is in fact equal to T_i, since $T_i = T_j$ for any events i and j.

Dasdan and Gupta claim that the cycle time of the system that the event graph models is the same as the maximum cycle mean of the event graph. This is part of the reason why the maximum mean cycle problem is of such great importance for performance analysis.

4.2.4 The maximum profit to time ratio

If the model includes the quantity τ introduced in Section 4.2.1, the max-causality rule is given by

$$s_i(k) = \max_{j \in Pred(i)} \left\{ s_j(k - \tau_{ji}) + w_{ji} \right\}, \ k \geq \tau_{ji} \geq 0,$$

which should be compared to the first version of the max-causality rule given in the previous section.

The equivalent of cycle mean in a model governed by the max-causality rule incorporating the τ-quantity is the *profit to time ratio*, defined as

$$\rho(C) = \frac{w(C)}{\tau(C)},$$

where $w(C)$ still is the weight of the cycle C, and $\tau(C)$ is the sum of the τ values around C.

The cycle time of a system modeled with the τ-quantity is now obtained by the *maximum profit to time ratio* $\rho(G)$ of the event graph G, defined by

$$\rho(G) = \max_{\forall C \in G} \left\{ \rho(C) \right\}.$$

4.3 Condition graphs

Condition graphs are not simple event graphs, for which the definition of the maximum cycle mean in Section 4.2.2 applies without modifications. Condition graphs are instead event graphs where every arc a_{ij} is associated with an occurrence index offset τ_{ij} (see Section 4.2.1). The τ values in a condition graph are however restricted to be either 0 or 1.

In other words, the firing of an event i in a condition graph is enabled by events in its own cycle (in cases where τ_{ji} is 0 for the arc from i's predecessor node j to i itself) and/or by events in the preceding cycle (in cases where the τ value in question is 1). This suggests that bowed arcs in the condition graph should be associated with a τ value equal to 1, and straight arcs with the τ value 0, which is of course correct.

Ekman and Kreuger recognize that the cycle time of a traffic pattern, as modeled by the condition graph, equals the maximum profit to time ratio (see Section 4.2.4) when bowed and straight arcs are given τ values 1 and 0 respectively, as indicated above. However, they choose to call the property the *maximum cycle mean of a condition graph* rather than maximum profit to time ratio, since the former term better signals what is being calculated.

Ekman and Kreuger let $m\left(C\right)$ denote the cycle mean of a cycle C in a condition graph G_{con}. $m\left(C\right)$ is defined to be the weight of the cycle C, $w\left(C\right)$, divided by the number of bowed arcs, $b\left(C\right)$, in C, that is

$$m\left(C\right) = \frac{w\left(C\right)}{b\left(C\right)} .$$

The maximum cycle mean of a condition graph G_{con}, $m\left(G_{con}\right)$, is accordingly given by

$$m\left(G_{con}\right) = \max_{\forall C \in G_{con}} \left\{m\left(C\right)\right\} ,$$

and is the same as the cycle time of the traffic pattern represented by the condition graph.

The maximum cycle mean, as defined in Section 4.2.2, is obviously the same as the maximum profit to time ratio with every τ value set to 1 (see Section 4.2.4). Although there are efficient algorithms that compute the maximum profit to time ratio, the corresponding ones that compute the maximum cycle mean are always faster[12].

Ekman and Kreuger's strategy is to transform the condition graph G_{con} to a so called *collapsed graph* G_{col}, in such a way that the maximum cycle mean of the collapsed graph, $\lambda(G_{col})$, equals the maximum cycle mean of the condition graph $m(G_{con})$. The reason is of course that it is then possible to use an algorithm for computing the maximum cycle mean that is known to be very efficient, directly on the collapsed graph, and still get the desired answer.

[12]This is obvious, since timesaving and simplifying assumptions can be made in the latter ones.

Ekman and Kreuger provide a concise proof of why the maximum cycle mean of a collapsed graph is the same as the maximum cycle mean of the condition graph that is the basis for the collapsed graph in [9]. I do not account for it here, but nevertheless gives a convincing argument why it is true in Section 6, in connection with my explanation of the procedure that actually collapses the graph.

4.4 Karp's Algorithm

There exist quite a number of algorithms that can be used to compute the maximum cycle mean, or its dual problem, the minimum cycle mean. Many of these are based on Karp's Algorithm. The basis of Karp's Algorithm and its variants is Karp's Theorem. It provides a handy expression for the maximum cycle mean.

4.4.1 Karp's Theorem

Let D be a strongly connected digraph with n nodes, and select any node in D to be the *source*, or s for short. For all $v \in$ V, and all $k = 0, 1, 2, \ldots, n$, let $D_k(v)$ represent the maximum weight of a walk of length k from s to v. If no walk of length k from s to v exists, $D_k(v)$ is considered to have the value $-\infty$. Now, Karp's Theorem can concisely be expressed as

$$\lambda(D) = \max_{v \in V} \min_{0 \leq k \leq n-1} \frac{D_n(v) - D_k(v)}{n - k}.$$

There is a concise and elegant proof of Karp's Theorem in [2]. I intend to elaborate on that proof and explain every step carefully. Comprehending this proof is important since an essential part of my implementation rests on the fact that Karp's Theorem can indeed be used in the computation of the maximum cycle mean.

4.4.2 Proof of Karp's Theorem

The proof of Karp's Theorem is not dependent on the choice of the source. Any node of D can be selected as source node, and this will be explicitly demonstrated later in the proof.

First, Karp's Theorem is proven for digraphs whose maximum cycle mean is 0. Of course, this is only one possible case out of infinitely many possible ones (since the weights on the arcs are real numbers), but proving this will pave the way for proving Karp's Theorem for strongly connected digraphs having arbitrary maximum cycle means.

Case 1 So, let us assume that the maximum cycle mean of D is 0. This prompts me to show that

$$\max_{v \in V} \min_{0 \leq k \leq n-1} \frac{D_n(v) - D_k(v)}{n-k} = 0\,,$$

for the digraph D in question. Following this assumption, there must be at least one cycle in D whose cycle mean is 0, and no cycles (or loops) in D can have positive weights. Trivially, we can assume that there are arcs in D whose weights are negative, or else all arcs in D have weight 0 which leaves me with nothing to prove.

For every node v of D, there is a walk of maximum weight from the source node s to it, its weight equal to

$$\chi_{sv} = \max_{k=0,1,\ldots} D_k(v)\,.$$

Remember that $D_k(v)$ represents the maximum weight of a walk of length k from s to v, and amounts to $-\infty$ if no such walk exists.

Note that $D_n(v)$ is always smaller than, or equal to, χ_{sv}, for all v, that is

$$D_n(v) \leq \chi_{sv}\,, \forall v \in V.$$

Although not altogether intuitively comprehended, this newly defined value χ_{sv} is finite for every v of D. A walk of greater length than $n-1$ in a digraph D with n nodes, passes through a number of nodes exceeding n, and thus has to contain at least one cycle. This is due to the fact that there is at least one node that gets visited twice in a walk of such a length. The walk must contain at least the one cycle starting and ending at that node. No cycle can add to the weight of the walk, since, according to the assumption, none of these have positive weights. Thus χ_{sv} can be found, for every v of D, by restricting k to non-negative integers smaller than n.

The revised expression now reads: for every node v of D, there is a walk of maximum weight from the source node s to it, its weight equal to

$$\chi_{sv} = \max_{k=0,1,\ldots,n-1} D_k(v)\,.$$

The fact that $D_n(v)$ is always smaller than (or equal to) χ_{sv}, leads to the following expression:

$$D_n(v) - \chi_{sv} = \min_{k=0,1,\ldots,n-1} D_n(v) - D_k(v) \leq 0\,.$$

The replacement of the max operator by the min operator is a natural consequence of the following (trivial) equality:

$$-\left(\max_{k=0,1,\ldots,n-1} D_k(v)\right) = \min_{k=0,1,\ldots,n-1} -\left(D_k(v)\right) .$$

The expression inside the parentheses on the left side of the equality above is of course the definition of χ_{sv}.

Since n is always larger than any k, dividing the inequality by $n-k$ does not alter it. Hence,

$$\min_{k=0,1,\ldots,n-1} \frac{D_n(v) - D_k(v)}{n-k} \leq 0 .$$

Note that equality in the expression above will hold only if $D_n(v) = D_k(v)$. A node v such that this is true, exists in every digraph whose maximum cycle mean is 0. This is really the first and last thing that needs to be proven in order to show that Karp's Theorem is correct when $\lambda(D) = 0$.

Let ζ be a cycle of weight 0 in D. Let l be a node in the cycle ζ, and P_{sl} be a path of maximum weight from s to l. Create a walk W_e by extending P_{sl} with a number of repetitions of the cycle ζ, enough to make the length of W_e greater than or equal to n. ζ has weight 0, so W_e is just like P_{sl} a walk of maximum weight from s to l.

I now claim that the walk consisting of the first n nodes of W_e, ending at node l' – obviously also a node in the cycle ζ – is a walk of maximum weight from s to l'. This is true simply because a subwalk of any walk of maximum weight is of maximum weight itself. Furthermore, this subwalk $W_{sl'}$ of W_e has the same weight as the path $P_{sl'}$ of maximum weight from s to l'. The path $P_{sl'}$ is constructed by appending to P_{sl} those arcs and nodes in the cycle ζ that make l' reachable from l. (Any reader still skeptical should convince him- or herself that $P_{sl'}$ is indeed a path, and that it has maximum weight, before reading on.)

So, without having made any assumptions about the source node s or the nodes l or l', other than that l (and consequently l') is in the cycle ζ, it has in this way been shown that there is, in every strongly connected digraph whose maximum cycle mean is 0, a node v playing the role of l' above, and that $D_n(l') = \chi_{sl'}$. Therefore we get

$$\max_{v \in V} \left[\min_{k=0,1,\ldots,n-1} \frac{D_n(v) - D_k(v)}{n-k} \right] = 0 ,$$

and the first part of the proof, with $\lambda(D) = 0$, is completed.

Case 2 Proving the case when $\lambda(D)$ is any real number can be treated in exactly the same way as when $\lambda(D)$ is 0, making one simple observation. Suppose that a constant c is subtracted from (or added to) the weight of every arc in D. By definition, this will of course change the value of $\lambda(C)$ and thus of $\lambda(D)$ with that same constant. Also, $D_k(v)$ will change with kc for the same reason.

The following explicit steps, show that $\lambda(D)$ does indeed change in the above-mentioned way (using subtraction by c from every arc weight to exemplify):

$$\frac{[D_n(v)-nc]-[D_k(v)-kc]}{n-k} = \frac{D_n(v)-D_k(v)-nc+kc}{n-k} = \ldots$$

$$\ldots = \frac{D_n(v)-D_k(v)}{n-k} - \frac{nc-kc}{n-k} = \frac{D_n(v)-D_k(v)}{n-k} - c\,.$$

Clearly, both sides of the equality in the definition of Karp's Theorem are affected in the same way, that is, reduced by the constant c, if c is subtracted from every arc weight in D. Selecting a c such that $\lambda(D)$ becomes 0 makes the first part of this proof, when $\lambda(D) = 0$, applicable again. Supposing that $\lambda(D)$ is 0 apparently imposes no real restriction on the proof, since every case can be reduced to this. Thus, Karp's Theorem is proven for arbitrary values of $\lambda(D)$.

5 SICStus Prolog[13]

This section provides a very quick introduction to Prolog. Its major aim is to enable readers familiar with programming in general, but not with Prolog programming, to follow the description of my implementation in the following sections. For more comprehensive coverage of the Prolog language, I encourage the interested reader to turn to regular textbooks on the subject.

5.1 The logic programming paradigm

> [Logic programming] is based on the belief that instead of the human learning to think in terms of the operations of a computer [—] the computer should perform instructions that are easy for humans to provide. [16]

Even though I am a fan of the computer language C, I understand why many people who have programmed in C tend to curse pointers. Especially those who have not had the time or interest to really delve deeply into the task of programming, for example those who have attended programming courses as a compulsory part of their education without being genuinely interested in programming, seem to resent pointers and other structures of various programming languages that require knowledge about how the computer program is actually executed at a lower level.

One of the intentions with logic programming was to hide hardware specific details from the programmer, without forgoing the efficiency of programs written in languages of other paradigms. Logic programming, as it stands today, is far from this ideal, since it is desirable – sometimes even necessary – to be aware of how the execution mechanism works to be able to write efficient and correct code.

Nevertheless, in comparison with C and many other common programming languages, programming in Prolog is at least in some aspects at a higher level. Prolog is a declaration-free, typeless language. The contents of an initialized logical variable cannot change, as opposed to the contents of variables in conventional languages. This means that logical variables more closely resemble what we mean by variables in the mathematical sense of the word. Moreover, data manipulation is achieved entirely by what is called unification (see Section 5.4). And the basic approach when writing a program is writing axioms (rules) and defining relationships between objects.

5.2 Prolog basics

The *term* is the single data structure in logic programs. A term is a *constant*, or a *compound term*. A compound term consists of a *functor* and one or more *arguments*. A functor is characterized by its name and *arity* (its number of arguments). A functor with name `f` of arity `n` is denoted `f/n`. Two functors can

[13]The two main sources for this section about Prolog are [16] and [15].

have the same name, but they are considered to be different functors if their arities differ.

A constant is an integer or an *atom.* The latter is symbolized with any sequence of characters as long as the first letter is lowercase (or else the sequence needs to be quoted to show that it is indeed an atom).

Variables begin with an uppercase letter or the underline character _. A variable can be bound to a structure, a constant, or another variable. Terms that do not contain variables are called *ground*, and terms that do are called *nonground.* The name of a functor must be an atom. It cannot be a variable.

A Prolog program is mainly made up of three kinds of basic constructs called *Horn clauses*, or *clauses* for short: *facts, rules, and queries.* Facts are sometimes called *predicates.* A fact, or predicate, holds information about the relation between objects. If there is an arc from a node a to a node b in a graph, this could be represented as

```
arc(a,b).
```

`arc` is the name of the functor of this compound term, and the arity of the functor is 2. The functor can thus be denoted `arc/2`. As in any programming language, the syntax is important. A fact must end with a period.

If the variable is indicated solely by the underline character, the variable is *anonymous.* Anonymous variables in the same clause are treated as distinct variables.

Variables in facts are implicitly universally quantified. Compare the following sets of facts, where each set of facts comprises a separate, possible program (called "Program 1" and "Program 2"):

Program 1

```
times(0,0,0).
times(0,1,0).
times(0,2,0).
times(0,3,0).
...
```

Program 2

```
times(0,X,0).
```

The meaning of the second program is, as summarized by one single fact, that zero times *any number* is zero, which probably is exactly what the writer of the first program wanted to say in the first place.

Queries are syntactically identical to facts, but are used to extract information from logic programs. I subscribe to Sterling and Shapiro's model that distinguishes between facts and queries by ending the former with a period and the latter with a question mark to avoid confusion. Without the period or question mark, the entity is called a *goal.* Thus `P.` says that the goal `P` is true, and `P?` asks whether the goal `P` is true.

The answer to a ground query is *yes* if the query is a logical consequence of the program. Otherwise the answer is *no*. A nonground query is answered with the *substitution* that makes the query a logical consequence of the program, if such a substitution exists.

In a simple program consisting of the single fact `arc(a,b).`, the answer to the query `arc(a,b)?` is for instance *yes*. The answer to `arc(X,b)?` is `X=a`, since the replacement of every occurrence of `X` by `a` in the query results in a match of a fact in the program. `arc(a,X)?` gives `X=b`, `arc(X,Y)?` gives `X=a` and `Y=b`. `arc(X,X)?` simply gives *no*. The variable `X` cannot be both `a` and `b`.

Rules define relationships in terms of existing relationships. Consider the following program:

```
arc(a,b).
arc(b,c).
arc(c,d).
arc(e,f).

connected(X,X).
connected(X,Y) :- arc(X,Z), connected(Z,Y).
```

The second version of `connected/2` is a rule, defined in terms of the arc facts and itself. It displays a simple example of recursion, which I assume the reader is familiar with[14]. `connected(X,Y)` is the *head*, and the conjunction of the two goals `arc/2` and `connected/2` after the head, that is, after "`:-`", is called the *body* of the rule. The goals of the body are separated by commas, and the rule is concluded with a period.

A *procedure* is a collection of rules with the same predicate as head, and is the equivalent of a procedure, or a function, in other programming languages. The base case, or *base fact*, of the procedure `connected/2` is `connected(X,X)`, which says that anything is connected to itself. The procedure says that `X` is connected to `Y` if `X` and `Y` are the same thing or if there is an arc from `X` to `Z`, and `Z` is connected to `Y`.

5.3 Lists

A list in Prolog is a binary structure. The first argument is an element, and the second argument is the rest of the list. The functor for lists is "`.`", and the empty list is denoted `[]`. The list of a single element `a` is thus `.(a,[])`. Thanks to syntactic sugar, the same list can be written as `[a]`. The first element of a list, the *head*, and the rest of the list, the *tail*, are always separated. `.(X,Y)` is denoted `[X|Y]`, where `X` is an element and `Y` another list of elements (possible the empty list)[15]. If `Y` is `[b,c]`, then `[a|Y]` and `[a,b,c]` and `.(a,.(b,.(c,[])))` are three different ways of representing the same list.

[14]See Section 8.3.1 for a short discussion about recursion in a general context.

[15]For readers familiar with Lisp, `[X|Y]` and a cons pair are the same thing. The head and tail in Prolog correspond to *car* and *cdr* in Lisp.

5.4 Unification[16]

A substitution making two terms identical is called a *unifier*, and two terms are said to *unify* if they have a unifier. The substitution `{Y=[b,c]}` thus unifies `[a|Y]` and `[a,b,c]`. `{Y=[b,c]}` also happens to be the *most general unifier*, which is unique[17] (apart from the names of the variables, which may differ).

Loosely speaking, the most general unifier (mgu) comprises the least number of assumptions that make the two terms in question unify. To illustrate this point, consider what makes `foo(X,Y)` and `foo(Y,Z)` unify. The substitution `{X=Y,Y=Z}` is the minimum requirement, and thus a mgu. `{X=Y,Y=Z,X=1}` is also a unifier, but gives more information than what is required to make the two terms unify. Thus it is *not* a mgu.

The mgu is the substitution that the Prolog engine provides as the result of a unification. For a more comprehensive account of mgu's, I direct the reader to textbooks on Prolog, for example [16].

5.5 Search trees

When a logic program is executed, the Prolog engine basically tries to prove the given goal statements with the aid of the facts and rules of the program at hand. Proving in this context basically means succeeding to unify all relevant terms. This process can be represented by a *search tree*.

Sterling and Shapiro give the following definition: "A reduction of a goal `G` by a program `P` is the replacement of `G` by the body of an instance of a clause in `P`, whose head is identical to the chosen goal" [16]. A search tree of goal `G` is made up of nodes that represent goals, or conjunctive goals, with one goal selected. The root of the tree is goal `G`. Edges represent reductions. Every possible reduction of a goal in the search tree generates an edge.

If variable bindings are created due to the unification operation, these bindings are associated with the edge representing that unification (the reduction). If the selected goal at a node cannot be further reduced, a *failure node* is created, which becomes the leaf of that particular branch. A *proof* (the result of a successful search), is a path from the *root*, the initial goal `G`, to a *success node* in the tree.

A success node is reached if goal `G` is proven. `G` is proven when no more subgoals of `G` need to be reduced. The variable bindings collected along the branch constitute the proof. A proof is the result of the execution, since the execution is equivalent to constructing a proof. Once bindings are created, they apply to the rest of the branch and summarize what must hold for this branch if it is to end with a success node; the proof of a query `G?` is presented to the user as the variable bindings that make the proof valid.

[16]According to Sterling and Shapiro, there are various definitions of unifiers [16]. I base this section on their definition of unification, which they claim is nonstandard. Just like they do, I believe that this simplified definition of unification serves its purpose better here than a more comprehensive exposition would.

[17]I do not intend to prove this here.

5.6 Backtracking

During the actual execution, goals are being reduced in a specific order by the Prolog engine. The process can be visualized as searching for a success node in the search tree using depth-first, left-to-right search with *backtracking*.

Backtracking means that as soon as a failure node is reached, the engine undoes the computation to the last choice made and tries a different path in the search tree. Variables can in this way change state from bound to unbound. Unless the search tree contains an infinite branch[18], all solutions are eventually found, if any solution at all exists. Otherwise the goal `G` fails.

It is possible to prevent backtracking in a controlled way, by using the (non-logical) primitive predicate `!`, called *cut*. For example, it is sometimes possible to know in advance that if a certain condition holds, other alternatives do not even need to be tried since they cannot possibly succeed. A cut in a program commits Prolog to all choices it made from the point at which it matched a query, or subgoal, to the head of the clause containing the cut. Prolog is however free to backtrack at previous choices if the remainder of the clause is not satisfiable.

In the following example, `G`, `C`, `X` and `Y` represent parts of a program. `G` for instance represents the head of a rule, and `C`, `X` and `Y` represent one or more goals each. Here, `Y` will not even be tried if `C` succeeds. Part of the reason for this is the rule order. When approached by the query (or subquery) `G?`, the Prolog engine first tries the first rule named `G`. But without the cut, `Y` might be tried at some point even if `C` succeeded, regardless of rule order.

```
G :- C,!,X.
G :- Y.
```

The interpretation of the clauses above reads: ”To satisfy `G`, if `C` is *true*, then satisfy `X`. Otherwise satisfy `Y`”.

[18]In particular, if the infinite branch is tried before all solutions are found, all solutions will not be found by the Prolog engine.

6 Collapsing a graph

A condition graph G_{con} is transformed into a collapsed graph G_{col} (also a strongly connected digraph) in two steps. First G_{con} is transformed into an acyclic digraph called the *split graph*, $S(G_{con})$. Then $S(G_{con})$ is in turn transformed into the *collapsed graph* G_{col}, in which the existence of an xy-path in the condition graph that includes exactly one bowed arc is represented by an arc a_{xy}. The weight of that arc is the maximum weight of all such paths from x to y (paths from x to y including exactly one bowed arc) in G_{con}.

6.1 The split graph

The condition graph consists of a node set V, a set of straight arcs A_s, a set of bowed arcs A_b and a weight function w associating every arc in A_s and A_b with a real-numbered (possibly negative) weight.

The split graph $S(G_{con})$ has three node types: *base nodes* B, *top nodes* T and *simple nodes* N. $B \cup N$ is exactly the node set V of the original condition graph G_{con}: the base nodes B are those nodes that are hit by bowed arcs in G_{con}, and the simple nodes N are the rest of the nodes in G_{con}. For every base node $x \in B$ there exists in the split graph a clone in the shape of a top node $x' \in T$. The split graph thus has more nodes than the condition graph it is based on.

$S(G_{con})$ has only one kind of arc, here simply denoted A. Every arc a in G_{con}, $a \in A_s \cup A_b$, corresponds to an arc in $S(G_{con})$, and the latter has the same weight w as it has in G_{con}:

- a straight arc from x to y with weight w in G_{con} gives rise to an arc from x to y in $S(G_{con})$ with weight w
- a bowed arc from x to y with weight w in G_{con} gives rise to an arc from x to y' with weight w in $S(G_{con})$, where y' is the top node in the split graph corresponding to y in the condition graph

There are additional arcs in the split graph. An arc from a node x to a node y with weight w in the condition graph, where both x and y are hit by bowed arcs, that is, x and y are base nodes in the split graph, gives rise to not only an arc from x to y with weight w in $S(G_{con})$, but also to an arc from x' to y' with weight w in $S(G_{con})$.

- a straight arc from x to y with weight w in G_{con}, where $x, y \in B$ in $S(G_{con})$, gives rise to an arc from x' to y' in $S(G_{con})$ with weight w, where $x', y' \in T$, in addition to the arc from x to y with weight w in $S(G_{con})$

Figure 1 shows a small condition graph G, and Figure 2 what the split graph $S(G)$ would look like.

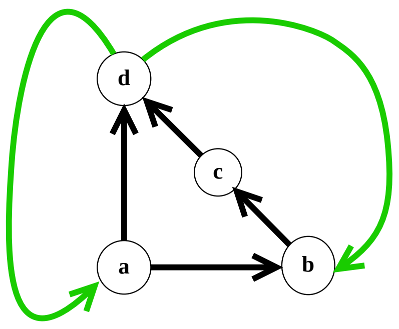

Figure 1: A simple condition graph

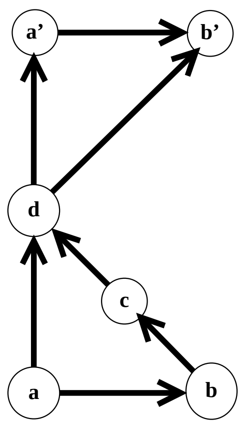

Figure 2: The split condition graph of Figure 1

6.2 split/6

My implementation of `split/6` is straightforward with respect to the definition of the split graph. Here is an overview of what it does. The condition graph is given as input.

1. The base nodes are identified.
2. The top nodes are identified.
3. The arcs that the bowed arcs in the condition graph give rise to are decided.
4. The arcs that the straight arcs in the condition graph give rise to are decided.

The code I have written is available in its entirety only to Banverket and SICS AB. I will not quote, or go into details about, every piece of code that I have written. But for the purpose of explaining my commenting style, I will explain `split/6` fairly thoroughly. This makes the rest of the code easier to follow – both the parts that are included in the report, and the parts that are not.

Figure 3 (see page 28) displays what `split/6` looks like in Prolog code, including my comments about it.

6.2.1 Input and output arguments

First of all, `split/6` has six arguments. The only one that is used as input is the first, called `Graph`. The plus sign in front of the first argument (arg1) means that it is intended as input, as opposed to the last five arguments that are preceded by minus signs to indicate that they are the output arguments of `split/6`. Input arguments should be instantiated, while output arguments must not be instantiated.

Some procedures have arguments that can be either instantiated or uninstantiated depending on what the user wants from the procedure. Such arguments are suitably preceded by a question mark to indicate that the user has this choice.

6.2.2 Association lists

Arg2, arg3, arg4 and arg5 are all represented by lists of names (or definitions) of the nodes, arcs, base nodes and top nodes in question. An arc is denoted by the fact `a(X,Y,W)`, where the existence of such a fact states that there is an arc from node `X` to node `Y` with weight `W`. `BaseTopNodeRelations` is an *association list*, which is a structure provided by the Prolog library. Predicates in the Prolog library are not *built-in*, and needs to be explicitly loaded. In this case, the following line loads the association lists package:

```
:- use_module(library(assoc)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% split(+Graph,-Nodes,-Arcs,-BaseNodes,
%%       -TopNodes,-BaseTopNodeRelations) <--
%%
%%       The graph Graph is defined with four
%%       arguments:
%%       g(Name,Nodes,StraightArcs,BowedArcs).
%%       Graph is a condition graph. The last
%%       five arguments of split/6 specify the
%%       split graph corresponding to Graph.
%%
%% Nodes, Arcs, BaseNodes, and TopNodes are
%% just what they seem. BaseTopNodeRelations
%% is an association list with top nodes as
%% keys and the nodes they originate from as
%% values.  This relation is needed by
%% collapse/7.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

split(G,V,A,B,T,BT) :-
    g(G,Nodes,StrA,BowedA),
    make_base_nodes(BowedA,B),
    make_top_nodes(B,T,BT),
    decide_arcs1(BowedA,BT,A1),
    decide_arcs2(StrA,B,A2),
    append(A1,A2,A),
    append(Nodes,T,V).
```

Figure 3: The code for `split/6`

An association list is a list of *key-value pairs*. In Prolog it is implemented as a balanced tree[19]. Lookup, insertion and deletion in the association list are all $O(log\, n)$ operations (in the worst case). The purpose of `BaseTopNodeRelations` is to keep track of what base node a specific top node corresponds to.

Condition graphs are specified by the fact `g/4`. Various definitions of graphs (including condition graphs) are appropriately kept in a separate file where new graph definitions easily can be inserted or deleted by manually modifying the file. In my case, I call this file 'graphs.prolog'. Arg1 of `g/4` is the name of the graph, and is used as a convenient handle to get hold of a graph definition: `G` unifies with arg1 in the first clause of the body of `split/6`, and thus (if `G` is instantiated with a valid graph name) the nodes (arg2) of the graph with name `G`, and its straight and bowed arcs (arg3 and arg4), unify with `Nodes`, `StrA` and `BowedA` respectively.

6.2.3 Accumulating parameters

Almost all of the procedures I have defined are recursive in nature, but use *accumulating parameters* to achieve better run-time characteristics. An accumulating parameter is passed along all the recursive steps. When the base case is reached, the value of the accumulator is transferred to the output variable.

A straightforward example is `make_top_nodes/5` (Figure 4). `make_top_nodes/3` is called from `split/6`, and `make_top_nodes/3` in turn calls `make_top_nodes/5`: the single purpose of `make_top_nodes/3` is to hide implementation specific details from the procedure that actually calls `make_top_nodes`. All the procedure (in this case `split/6`) cares about is to get the top nodes in question in return[20].

The two extra arguments added in `make_top_nodes/5` are accumulating parameters. Arg2 is an empty list and arg4 an empty association list, represented with `[]` and `t` respectively, as required by the SICStus assoc-library (see Section 6.2.2).

If the condition graph is correctly specified, arg1 is not an empty list when `make_top_nodes/3` is called: there is at least one base node in any condition graph consisting of one or more nodes. The base case is thus not applicable to the first call to `make_top_nodes/5`. Instead, `B` in `[B|Bs]` (in the head of the second version of the rule `make_top_nodes/5`) will unify with the first element in the list of base nodes. `t(B)` is, with the aid of `add_to_relation/4`, added to the association list `AccBT` that keeps track of the base and top node relations, putting `t(B)` as the key and `B` as the value in one of its key-value pairs. `AccBT2` is the updated `BaseTopNodeRelations` association list.

After that, a recursive call on the rest of the list of base nodes, `Bs`, is made, with the top node `t(B)` added to arg2, an accumulator collecting all top nodes that have been identified so far.

[19]Balance in this context is defined by the Adelson-Velskii-Landis balance criterion: "A tree is balanced iff for every node the heights of its two subtrees differ by at most 1" [18].

[20]It is of course a matter of taste how one uses accumulating parameters. I prefer to hide them. I thus avoid adding accumulating parameters directly in the calls from other procedures.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% make_top_nodes(+BaseNodes,-TopNodes,
%%               -BaseTopNodeRelations) <--
%%
%%               Every base node (see the
%%               definition above) has a
%%               corresponding top node. Top
%%               nodes are here defined as
%%               t(Node), i.e., the existence
%%               of the fact t(X) means that
%%               there is a top node
%%               Y == t(X), corresponding to
%%               the node X.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

make_top_nodes(B,T,BT) :-
    make_top_nodes(B,[],T,t,BT).

make_top_nodes([],Top,Top,BT,BT).   % Base case
make_top_nodes([B|Bs],AccT,Top,AccBT,BT) :-
    add_to_relation(t(B),AccBT,B,AccBT2),
    make_top_nodes(Bs,[t(B)|AccT],
                   Top,AccBT2,BT).
```

Figure 4: The code for `make_top_nodes/3` and `make_top_nodes/5`

When arg1 is the empty list, the base case is reached: the call to `make_top_nodes/5` unifies with `make_top_nodes([],Top,Top,BT,BT)`. Here arg3, hitherto uninstantiated, unifies with arg2. The same happens with arg5, which unifies with arg4. Although arg3 and arg4 have merely been passed along in every recursive step, uninstantiated, they are of course the same variables in every call. Once instantiated (by the base case), the result propagates all the way back to the first call to `make_top_nodes/5`, where `T` and `BT` now are known. Thus arg2 and arg3 of `make_top_nodes/3` now also become instantiated.

6.2.4 Exploiting rule order

`decide_arcs2/4`, which is called from `decide_arcs2/3`, exemplifies some important aspects I have pointed out. Consider the code of `decide_arcs2/4` in Figure 5 (page 32).

The code for `decide_arcs2/4` can be read as follows:

1. If arg1 an empty list, then the accumulating list of arcs is the final one, so arg3 and arg4 should unify. This is the base case.
2. If not, we check to see if the start and end nodes, `X` and `Y`, of the arc first in the list of arcs are both base nodes. If so, an arc identical to the one examined is put in the accumulating list together with an arc between the top nodes corresponding to `X` and `Y`, that is, from `t(X)` to `t(Y)`. Then step 1 is applied to the rest of the list of arcs. If `X` and `Y` are not both base nodes, proceed to step 3.
3. If nodes `X` and `Y` are not both base nodes, then as soon as this is discovered (when either `member(X,B)` or `member(Y,B)` fails), the last rule of the procedure is tried. It simply puts the arc as it is in the accumulating list (arg3) and calls `decide_arcs2/4` on the tail of the list of arcs: apply step 1 on the rest of the list.

The order of the rules of the procedure is crucial. The position of the base fact can be arbitrarily chosen in this case, without changing the outcome of a call to the procedure. Interchanging the positions of the other two `decide_arcs2`-rules would, however, render 'wrong' solutions. Wrong in the sense that the solution is not what I intended from the procedure. In this case, it is fairly easy to see why.

The first rule encountered would now, since no test is included in the body of the rule, always succeed as long as arg1 is not an empty list, in which case only the base fact would succeed. Since the solutions in the search tree are found by a depth-first left-to-right search (see Sections 5.5 and 5.6), the first solution suggested by the Prolog engine, arg3 (`Arcs`), would always be identical to the original input argument, arg1 (`StraightArcs`). This solution is not interesting to us.

The 'correct' (intended) solution would sooner or later be found if we (the user) kept asking the Prolog engine for more suggestions to solutions. On this

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% decide_arcs2(+StraightArcs,
%%              +BaseNodes,-Arcs) <--
%%
%%              A straight arc a(X,Y,W),
%%              connecting two base nodes,
%%              gives rise to an arc
%%              a(XTop,YTop,W), where XTop is
%%              t(X), i.e., the top node
%%              corresponding to the node X,
%%              and YTop is t(Y) -- as well
%%              as an arc identical to itself.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

decide_arcs2(S,B,Arcs) :-
    decide_arcs2(S,B,[],Arcs).

decide_arcs2([],_,Arcs,Arcs).   % Base case
decide_arcs2([a(X,Y,W)|As],B,Acc,Arcs) :-
    member(X,B),
    member(Y,B),
    !,
    decide_arcs2(As,B,
                 [a(X,Y,W),a(t(X),t(Y),W)|Acc],
                 Arcs).

%% In case one of the nodes X and Y, or both,
%% are not base nodes, the straight arc in the
%% condition graph simply gives rise to an arc
%% in the split graph that is identical to
%% itself

decide_arcs2([a(X,Y,W)|As],B,Acc,Arcs) :-
    decide_arcs2(As,B,[a(X,Y,W)|Acc],Arcs).
```

Figure 5: The code for `decide_arcs2/3` (and `decide_arcs2/4`)

request, the Prolog engine backtracks to find the next possible solution. In this case, the second rule (the one with the member-tests) would on a second attempt be tried on at least one of the arcs.

At least one branch in the search tree ends with the success node we want, that is, the one along which every arc that starts and ends in a base node has been tried against the rule with the tests. The proof of this success node, that is, the result of the execution, gives the desired output-arcs. If we put the rule with the tests first (as I have in my implementation), this success node will always be suggested first by the Prolog engine. Furthermore, the cut after the tests prevent backtracking at those positions in the search tree where `member(X,B)?` and `member(Y,B)?` have succeeded, so this solution will be the only one suggested.

6.2.5 Red and green cuts

In this very simple example, it is easy to see that the first solution is the only solution we want. In other cases it might be a lot harder to see. And do we really want to be dependent on rule order?

In a case like this, we know that there is a unique solution. An arc either goes from a base node to a base node, or it simply does not. Depending on what applies to a specific arc, we want to take one of two possible actions. Even with the right rule order, the removal of the cut would make the Prolog engine provide additional solutions on backtracking, all different from the single one that is correct. The cut changes the *declarative meaning* of the program since it actually cuts away success branches in the search tree.

Cuts like the one described are called *red cuts*, and should generally be avoided. In a simple if-then-else case like this, when it is trivial to mind one's p's and q's, a red cut is justifiable since the code is still easy to follow and more efficient than the alternatives.

To make the procedure independent of rule order, all rules (except the base fact) would need tests in the body. This would result in a minimum of three rules on top of the base fact: one that checks if both nodes `X` and `Y` in `a(X,Y,W)` are base nodes, one that checks if `X` is a not a base node, and one that checks if `Y` is not a base node.

An example of how this can be done is displayed in Figure 6. Note that the removal of the cuts now would only remove duplicate solutions to the correct one, since there are more than one way to get to the one solution: if none of `X` and `Y` in `a(X,Y,W)` is a base node, two rules can succeed, thus giving rise to two success branches. The result is the same, since the two rules look the same apart from the test. Although success branches are cut away if the cuts are placed as in this code, the cuts here are *green cuts*: only duplicate solutions are being removed.

To make the procedure independent of rule order and to minimize the number of cuts needed, four rules are required (apart from the base fact). All rules need two tests in their bodies, placed before the recursive call on the tail of the list. If A means "A is a base node", and $A*$ means "A is not a base node", the

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% decide_arcs2(+StraightArcs,
%%              +BaseNodes,-Arcs) <--
%%
%% A second version, this time independent of
%% rule order.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

decide_arcs2(S,B,Arcs) :-
    decide_arcs2(S,B,[],Arcs).

decide_arcs2([],_,Arcs,Arcs).      % Base case

decide_arcs2([a(X,Y,W)|As],B,Acc,Arcs) :-
    not(member(X,B)),
    !,
    decide_arcs2(As,B,[a(X,Y,W)|Acc],Arcs).

decide_arcs2([a(X,Y,W)|As],B,Acc,Arcs) :-
    not(member(Y,B)),
    !,
    decide_arcs2(As,B,[a(X,Y,W)|Acc],Arcs).
decide_arcs2([a(X,Y,W)|As],B,Acc,Arcs) :-
    member(X,B),
    member(Y,B),
    !,
    decide_arcs2(As,B,
                 [a(X,Y,W),
                  a(t(X),t(Y),W)|Acc],
                 Arcs).
```

Figure 6: A second version of `decide_arcs2/3`, this time independent of rule order

four different rules should test $A \cap B$, $A \cap B*$, $A * \cap B$ and $A * \cap B*$ respectively. This time, there will be only one success node.

One inefficient way to implement the (almost) cut-free and rule order independent version mentioned in the previous paragraph, is to check $A \cap B$, $A \cap B*$, $A * \cap B$ and $A * \cap B*$ in the same manner as in the code in Figure 6. If A and $A*$ are decided afresh in every version of the rule, the result is a great many unnecessary, costly calls. `member/2` would be run from two times up to eight times per arc instead of between one and two times per arc, as is the case in the implementation I have decided to use (Figure 5).

Figure 7, on the other hand, shows an efficient way to solve this problem. This version is independent of rule order and involves only one cut. The cut is this time used by a helper procedure, `is_member/3` (Figure 8), and it is green.

The number of calls to `member/2` is minimized with the aid of `is_member/3`. The purpose of this helper procedure is obvious. It checks whether a certain element is a member of a certain list or not, and returns the result ('yes' or 'no') to the calling procedure – here `decide_arcs2/4`. The result is then passed on as a separate parameter to `decide_arcs2/6`. Each time the member property has to be checked in a rule, the rule simply consults the boolean instead of calling `member/2`. In this way, `member/2` is called exactly two times per arc instead of between two and eight times, as would be the case if `member/2` were run afresh every time the property needed to be checked.

6.3 The collapsed graph

In the beginning of Section 6, I defined the collapsed graph to be a strongly connected digraph in which the existence of an xy-path in the condition graph that includes exactly one bowed arc is represented by an arc a_{xy}. The weight w of that arc is the maximum weight of all paths from x to y in the condition graph that includes exactly one bowed arc.

Consider what we are really after, and the collapsed graph is almost self-explanatory. If the condition graph is viewed as a discrete event system subject to the second max-causality rule defined in Section 4.2.4, every arc is associated not only with a weight w, but also with a value τ that is either 0 (for straight arcs) or 1 (for bowed arcs). The cycle time of the system is, using Ekman and Kreuger's terminology, the maximum cycle mean of the condition graph, where the mean of a cycle in a condition graph in this context is calculated as the sum of the arc weights along the cycle, divided by the number of bowed arcs in the cycle[21].

The goal of collapsing the condition graph is to get a graph that is equivalent to the condition graph with respect to the cycle time, but simpler in structure since its τ-values are all equal to 1.

[21]As I have pointed out, Ekman and Kreuger's definition of the maximum cycle mean in a condition graph coincides with what others call maximum profit to time ratio. See Section 4.2.4 for a formal definition of this term, and Section 4.3 for an explanation to the confusion of terms.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% decide_arcs2(+StraightArcs,
%%              +BaseNodes,-Arcs) <--
%%
%% A third version, this time both independent
%% of rule order and efficient.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

decide_arcs2(S,B,Arcs) :-
    decide_arcs2(S,B,[],Arcs).

decide_arcs2([],_,Arcs,Arcs).  % Base case
decide_arcs2(S,B,Acc,Arcs) :-
    S=[a(X,Y,_)|_],
    is_member(X,B,Xb),         % Decide whether
                               % X is a base
                               % node or not
    is_member(Y,B,Yb),         % Is Y a base
                               % node?
    decide_arcs2(S,B,Xb,Yb,Acc,Arcs).

decide_arcs2([a(X,Y,W)|As],B,
             Xb,Yb,Acc,Arcs) :-
    Xb==yes,                   % Both X and Y
    Yb==yes,                   % are base nodes
    decide_arcs2(As,B,
                 [a(X,Y,W),a(t(X),t(Y),W)|Acc],
                 Arcs).

%% In case one of the nodes X and Y,
%% or both, are not base nodes, the
%% straight arc in the condition graph
%% simply gives rise to an arc in the
%% split graph that is identical to
%% itself

decide_arcs2([a(X,Y,W)|As],B,Xb,Yb,Acc,Arcs) :-
    Xb==yes,                   % Only X is a
    Yb==no,                    % base node
    decide_arcs2(As,B,[a(X,Y,W)|Acc],Arcs).
decide_arcs2([a(X,Y,W)|As],B,Xb,Yb,Acc,Arcs) :-
    Xb==no,                    % None of X and
    Yb==no,                    % Y is a base
                               % node
    decide_arcs2(As,B,[a(X,Y,W)|Acc],Arcs).
decide_arcs2([a(X,Y,W)|As],B,Xb,Yb,Acc,Arcs) :-
    Xb==no,                    % Only Y is a
    Yb==yes,                   % base node
    decide_arcs2(As,B,[a(X,Y,W)|Acc],Arcs).
```

Figure 7: A third version of `decide_arcs2/3`, this time efficiently implemented with a helper procedure `is_member/3`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% is_member(+Element,+List,-Boolean) <--
%%
%%            Boolean is set to 'yes' if
%%            Element is a member of the
%%            list List, and to 'no' if it
%%            is not.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

is_member(X,B,Boolean) :-
    member(X,B),
    !,
    Boolean = yes.
is_member(_,_,Boolean) :-
    Boolean = no.
```

Figure 8: The code for `is_member/2`

6.3.1 A naive graph-collapsing algorithm

A naive way of collapsing a graph, would be to:

1. Find all distinct subpaths and cycles in the condition graph that contain exactly one bowed arc
2. Find the weights of the subpaths (and cycles) found in step 1
3. For every ordered pair of nodes x and y (x, y) in the condition graph, where $x = y$ is allowed, for which there exists at least one subpath from x to y found in step 1, make note of the weight of the heaviest one from x to y (found in step 2)
4. Create a new, collapsed, graph with the same node set as that of the condition graph, and with an arc from x to y with weight w if and only if there exists a subpath from x to y from step 1. The weight w is that of the heaviest subpath from x to y that has been noted in step 3

The first crux is: how do we find all distinct subpaths involving exactly one bowed arc in an efficient way?

The second question at issue is that it is obvious that there will be redundant arcs if the approach above is used. In step 4, all cycles involving only one bowed arc will invariably give rise to one arc (a loop) per node in the cycle. Some of these – all but one if the cycle in question is a critical cycle – will clearly be redundant. Consider for example a traffic pattern involving only two movements, a and b, for which $(a, n) < (b, n)$ holds[22]. Its condition graph

[22]Recall that $(a, n) < (b, n)$ means that movement a of cycle n has higher precedence than movement b of cycle n (see Section 1.3.2).

consists of nodes a and b, the straight arc a_{ab} with weight w_1 and the bowed arc a_{ba} with weight w_2. It is easy to see that one of the arcs created in step 4 is redundant. Two loops, a_{aa} and a_{bb}, spring out of the same cycle, and their weights are of course identical, that is, $w_1 = w_2$.

6.3.2 Improving the graph-collapsing algorithm

One single, important observation modifies step 1 and makes it less time-consuming to perform. First I have to say that from this point on, I will use the term *base nodes* also to refer to nodes *in the condition graph* that are hit by bowed arcs, not just to the nodes in the split graph that correspond to the nodes that are hit by bowed arcs in the condition graph.

The observation made, is that every cycle must include at least one bowed arc. The consequence of this is that every cycle also must include at least one base node: the one (or ones) hit by that bowed arc (or those bowed arcs). It is thus possible to have base nodes as points of departure for the subpaths identified in a modified version of step 1 in Section 6.3.1 above.

Note that a cycle can include a greater number of base nodes than bowed arcs. The opposite is however not possible. In the following, when I reason about a specific cycle and talk about base nodes, I am only interested in the base nodes that are hit by bowed arcs that are *included in the cycle in question.* Let $B(C)$ denote the base nodes of cycle C that are hit by the bowed arcs that are included in C to avoid confusion[23].

The only thing we really require from the collapsed condition graph is the following: we need to be certain that the critical cycle (see Section 1.3.4) with mean m in the condition graph G_{con} will be represented by a critical cycle in the collapsed condition graph G_{col} with the same mean m.

Let C_{crit} be a critical cycle in G_{con}. Then there are two possible cases. The first possibility is that C_{crit} includes exactly one base node $b \in B(C)$. The other possibility is that C_{crit} includes two or more base nodes, $b_1, \dots, b_n \in B(C)$, $n = 2, 3, \dots$.

Case 1 C_{crit} includes exactly one base node $b \in B(C)$. Compute the mean of this cycle m_{crit} by accumulating the arc weights along the cycle[24] , starting with the weight of the arc that leaves b. Represent the base node b in G_{con} with a node v in G_{col}. The cycle C_{crit} in the condition graph G_{con} is then represented by a loop from the node v in G_{col}. The weight of the arc (the loop) is of course m_{crit}.

[23]Do not confuse this designation with $b(c)$, that Ekman and Kreuger use to denote the number of bowed arcs in a cycle c. The number of bowed arcs in a cycle, $b(c)$, and the number of base nodes, $|B(C)|$, in a cycle C is of course the same if $c = C$, so these properties are however related.

[24]The mean m_{crit} of the cycle is actually the result of adding all arc weights of the arcs along the cycle and then dividing the result with one, the number of bowed arcs in the cycle.

Case 2 C_{crit} includes two or more base nodes, $b_1, \ldots, b_n \in B(C)$, $n = 2, 3, \ldots$. b_1 can be arbitrarily chosen among the base nodes $B(C)$ in the cycle. The base nodes are named in order of appearance so that b_1 is 'first', b_2 'next', etc. Compute the weights of the paths from b_i to b_{i+1}, $i = 1, 2, ..., n$, as well as the weight of the path from b_n to b_1. The weights of the paths are equal to the respective sums of the weights of the included arcs in the paths. Moreover, these paths include exactly one bowed arc each. Let the base nodes $b_1, \ldots, b_n$ in G_{con} be represented by nodes $v_1, \ldots, v_n$ in G_{col}, and let every path mentioned above be represented by an arc in G_{col} according to the following principle: a path from b_i to b_j in G_{con} with weight w corresponds to an arc in G_{col} from v_i to v_j with weight w.

It is now clear that it is sufficient to find all paths *between base nodes* that include exactly one bowed arc. This is true because no specific assumptions about the critical cycle or the base node (or base nodes) have been made in the reasoning above. If every path between two base nodes results in an arc in the collapsed condition graph (between the nodes that correspond to the base nodes in the condition graph), at least one of the cycles in the collapsed graph will be a critical cycle. The critical cycle (or cycles) in the collapsed graph will have the same weight as the critical cycle (or cycles) in the condition graph.

The new wording of step 1 in the naive algorithm in the beginning of Section 6.3.1 becomes:

- Find all distinct paths in the split graph that start with a base node and end with a top node.

If there are more than one critical cycle in the condition graph, they all have the same weight (per definition). At least one of them will have a direct correspondence in the form of a critical cycle in the collapsed graph. And this is exactly what we need: no more, no less.

The purpose of the split graph is now obvious: the heaviest path between any pair of base nodes in the condition graph that includes exactly one bowed arc is represented by the heaviest path between the corresponding base node-top node pair in the split graph.

The split graph is acyclic. The bowed arcs are in the split graph represented by arcs to top nodes. Every top node corresponds to exactly one base node. A path from any base node to any top node in the split graph corresponds to a path in the condition graph that includes exactly one bowed arc. The starting node in the condition graph is the base node, and the ending node is the base node that the top node corresponds to.

Not only step 1 can be simplified. It is possible to combine steps 2 and 3 in the algorithm above. A couple of definitions are needed for the following reasoning.

Definition The *height of a node* v, denoted $height(v)$, in an acyclic graph G is the length of the longest path P in G that ends with v.

Let w_{xy} denote the weight of the arc from x to y.

Definition The maximum weight of a path P in the split graph $S(G_{con})$ from node x to node y is denoted $w_{path}(x, y)$ and is defined recursively in the following way.

- $w_{path}(x, y)$ is 0 if $x = y$. This is the base case.
- If $x \neq y$, for every arc a_{zy} that hits y where z is reachable from x, compute the sum $w_{path}(x, z) + w_{zy}$. $w_{path}(x, y)$ is the maximum of these sums.

All paths from x to y in $S(G_{con})$ are finite in length since the split graph is acyclic. $w_{path}(x, y)$ is thus well-defined as long as y is reachable from x. Otherwise $w_{path}(x, y)$ is undefined.

If $w_{path}(x, y)$ is decided for every node y that is reachable from x in the split graph *in height order*, the required $w_{path}(x, y)$-values are always explicitly known when they are needed.

To find all the required paths and their weights, all $w_{path}(x, y)$-values for every base node x in the split graph are needed.

Reformulation of step 2 and 3 gives

- For every base node x in the split graph, find $w_{path}(x, y)$ for every top node y.

The last step is now to create a collapsed graph, using the information gathered in the improved steps 1 and 2, together with the original input (the condition graph itself).

The wording now reads

- Let the base nodes of the condition graph/split graph form the node set of the collapsed graph. Let every $w_{path}(x, y)$-value computed in step 2, where x is a base node and y is a top node, result in an arc from x to the base node that y corresponds to, with weight corresponding to $w_{path}(x, y)$.

6.4 `collapse/7`

To keep track of the $w_{path}(x, y)$-values, I introduce a new structure in `collapse/7` (the code for `collapse/7` can be found in Figure 9). Conceptually it can be viewed as a matrix, called $W[i, j]$, in which every base node corresponds to a row, and every node (including the base nodes) corresponds to a column. The $w_{path}(x, y)$-values can be found in the matrix by translating node names to rows and columns: the maximum weight of the path from base node a to the simple node[25] b, $w_{path}(a, b)$, can be found in the row that corresponds to a and the column that corresponds to b.

[25] A simple node is any node in the split graph that is not a base node or a top node, see Section 6.1.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% collapse(+Nodes,+Arcs,+BaseNodes,
%%          +TopNodes,+BaseTopRelations,
%%          -Nodes,-ArcsColl) <--
%%
%% Output from split/6 is suitably used as
%% input to collapse/7. Nodes and ArcsColl are
%% the lists of nodes and arcs that specify the
%% collapsed graph. (BaseNodes and Nodes are
%% actually identical lists.)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

collapse(V,A,B,T,BT,B,ArcsColl) :-
    init_wp_info(B,WpPrel),
    list_heights(V,A,Heights),
    process_base_nodes(B,V,A,
                       T,BT,
                       WpPrel,
                       WpInfo),
    find_arcs(B,T,BT,WpInfo,ArcsColl).
```

Figure 9: The code for collapse/7

All entries in the matrix are not necessarily needed. It is hard to say whether the matrix generally will be sparse or dense, but my guess is that it is a bit more likely to be sparse – at least for large condition graphs. My implementation of the $w_{path}(x, y)$-structure is basically an adjacency-list representation, although I use SICStus' association lists instead of linked lists[26].

The structure is called wp_info/2 in my program, where arg1, MapNtoE, is an association list that maps node names to element positions in arg2, an array that I call ArrayOfAL. ArrayOfAL stores an association list corresponding to a base node Node at every valid position: one position for every base node in the split graph. Every path from Node to any other node in the split graph gives rise to a key-value pair, where the key Key is a node name and the value Value is the maximum weight of all paths from Node to Key in the split graph, that is, the $w_{path}(Node, Key)$-value.

I have chosen to 'initialize' wp_info/2 at the start of collapse/7 with the procedure init_wp_info/2. Declarations of structures are not at all required in Prolog, and this 'initialization' is only a help for me, the programmer, since the structure is more easy to use once it is prepared in the way init_d_info/2 prepares it.

list_heights/3 returns a list of node-height pairs, corresponding to nodes and their respective heights in the split graph. It is sorted according to the

[26]See Section 6.2.2 for an account of what association lists are.

height-values in the node-height pairs, with the smallest height first. This list is needed for the recursive computation of the $w_{path}(x, y)$-values, where the nodes reachable from x need to be processed in height order.

`process_base_nodes/8` takes care of one base node at a time, filling in all the values in that base node's particular row of the $W[i, j]$-matrix before moving on to the next base node.

Finally, `find_arcs/5` sifts out the relevant entries in $W[i, j]$, that is, the values in the columns that correspond to top nodes. The $w_{path}(x, y)$-value for a top node y reachable from a particular base node x gives rise to an arc in the collapsed graph, with weight $w_{path}(x, y)$, from the base nodes x to the base node that the top node y corresponds to.

6.4.1 `process_base_nodes/8`

A few words can be said about the clause in `collapse/7` that 'administrates' the filling of the $W[i, j]$-matrix: `process_base_nodes/8`.

`process_base_nodes/8` calls `max_weight_paths/4` on every base node in the split graph, one at a time.

`max_weight_paths/4` does the following with a base node b:

- Sets $W[b, b]$ to 0, since the $w_{path}(x, y)$-value is zero if $x = y$. b is now the source node, s, for this call to `max_weight_paths/4`.
- Identifies what nodes there are paths to from the source node s in the split graph, and in what order their $w_{path}(x, y)$-values should be computed (with $x = s$), that is, determines the heights of the nodes reachable from s.
- Calls `find_paths/5`

`find_paths/5` does the following to each node v that there is a path to from the source node s of the current call to `max_weight_paths/4`:

- Identifies all immediate predecessors of v that are reachable from s, that is, every node who is the start node of an arc that ends in v and to which there at the same time exists a path from s.
- Calls `process_predecessors/6`

`process_predecessors/6` does the following with every immediate predecessor *pre* of node v of `find_paths/5`:

- If *pre* is the first immediate predecessor of v to be treated by `process_predecessors/6`, $W[s, v]$ has not yet got even a preliminary value. $W[s, v]$ is thus simply set to $W[s, pre] + w_{pre,v}$ for the time being, where $w_{pre,v}$ is the weight of the arc from *pre* to v.

- If $W[s,v]$ has a preliminary value, meaning that another immediate predecessor of v has already been treated, the sum $W[s,pre]+w_{pre,v}$ is computed and compared with the preliminary value in $W[s,v]$. If the sum is greater than the preliminary value, $W[s,v]$ is updated with it. Otherwise the old value in $W[s,v]$ is kept.

`process_predecessors/6` is done when all immediate predecessors to v that are reachable from s have been processed, or treated, in the above-mentioned way. The value in $W[s,v]$ is then the final $w_{path}(x,y)$-value for $x = s$ and $y = v$.

`process_predecessors/6` then returns the updated `wp_info/2`-structure to `find_paths/5`: one element in the $W[i,j]$-matrix has been determined. `find_paths/5` calls itself on the rest of the nodes in the list of nodes that should be treated, to determine yet another element in the $W[i,j]$-matrix with $i = s$.

When all $w_{path}(x,y)$-values for $x = s$ have been determined by `find_paths/5` (and its 'helper', `process_predecessors/6`), the updated `wp_info/2`-structure is returned by `find_paths/5` to `max_weight_paths/5`. When this occurs, the row corresponding to the source node s in the $W[i,j]$-matrix contains the final values. `max_weight_paths/5` returns this to `process_base_nodes/8`.

`process_base_nodes/8` calls itself on the rest of the list of base nodes, to determine the values in the row corresponding to another base node in the split graph, until all rows have been filled with their final values. Then `find_arcs/5` is called. `find_arcs/5` uses the relevant $w_{path}(x,y)$-values in the `wp_info/2`-structure to create the arcs of the collapsed graph (see page 42).

7 Cycle time computation

Computing the maximum cycle mean of the collapsed graph is the only thing left to do to find the cycle time of the traffic pattern, that is, the inverse of the capacity as the term is defined by Ekman and Kreuger (see Section 1.3.1): the maximum cycle mean of the collapsed graph is the cycle time of the condition graph (see Sections 4.2.3 and 4.3).

There are, as I mention in the beginning of Section 4.4, many algorithms that compute the maximum cycle mean of a strongly connected digraph. I have chosen to use a variant of Karp's algorithm that is suggested by Dasdan and Gupta as basis for my implementation [7]. Karp's Algorithm is one of the most commonly used algorithms for the maximum (and minimum) mean cycle problem, and also one of the fastest. The main reason for my choice of Dasdan and Gupta's algorithm is their use of a technique they call *unfolding* (see Section 7.1). It improves the running time of Karp's Algorithm, and the result is a very efficient algorithm for this problem.

The algorithm that computes the maximum cycle mean of G_{col} resembles `collapse/7` (Section 6.4) up to a certain point. Once again, certain entries in a matrix need to be determined. The values of these entries have similar origin to those of the $W[i,j]$-matrix in the algorithm that collapses the split graph, except that they say what the maximum weight of *a walk of a certain length* from a certain node to another node is. Another difference is that the origin of all such walks in this algorithm is the same node, the *source node s*, so the rows and columns of the matrix now correspond to the lengths of the walks and the nodes of the graph respectively. The source node s can be chosen arbitrarily, but if it is selected with care, further running time can be saved (see Section 7.2.4).

This time, the data collected in the matrix (at least it is conceptually collected in the matrix) is of course not used to create arcs in a collapsed graph – it is used to find the maximum cycle mean via Karp's Theorem. A proof of Karp's Theorem is given in Section 4.4.2.

The values needed for Karp's Theorem are the $D_k(v)$-values mentioned in Section 4.4.1, where $D_k(v)$ represents the maximum weight of a walk of length k from s to v. Let $D[k,v]$ denote the matrix that holds these values. Let G be a digraph – for example a collapsed condition graph – and let V denote the node set of G, and n the number of nodes in G. Then Karp's Theorem says that

$$\lambda(G) = \max_{v \in V} \min_{0 \leq k \leq n-1} \frac{D_n(v) - D_k(v)}{n-k},$$

where $\lambda(G)$ is the maximum cycle mean of G.

$D_k(v)$ is computed recursively in a way similar to $w_{path}(x,y)$ (see page 40). $D_k(v)$ can be determined if the $D_{k-1}(v_p)$-values for all predecessors v_p of v, reachable from s, are determined. Generally, not all entries in the $D[k,v]$-matrix need to be determined in order to find $\lambda(G)$. To be able to know exactly what entries that are actually needed for the computation, the principle of unfolding (se Section 7.1) can be used.

7.1 Unfolding

The original version of Karp's algorithm considers every entry in $D[k, v]$, regardless of whether there is a walk from s of length k to v or not. The sparser the graph, the more unnecessary work the algorithm performs. Unfolding, the technique explained below, ensures that only relevant entries in D demand action by the algorithm[27].

First a few words about notation: if there exists an arc from node x to node y, y is a *successor* of x. Let G denote a cyclic, weighted digraph with node set V and arc set A. Let any node $v \in V$ be the source node, s, in the following reasoning. Let Q denote a queue that initially is empty. Let D be a $[n, n]$-matrix, whose entries are all initialized to $-\infty$, where n is the number of nodes in G.

Put the level-node pair $< 0, s >$ in the queue Q, meaning that the source is at level 0. The level k of a node is the same thing as the number of arcs away from s the node is when the current walk is considered. Set $D[s, 0]$ to 0 in D, meaning that $D_0(s) = 0$: the weight of the heaviest walk from s to s of length 0 is 0.

Dequeue Q and let $< k, v >$ denote the dequeued pair (when Q is dequeued for the first time, $v = s$ and $k = 0$). Do the following to every successor v_s of v if $k < n$, which is referred to as *unfolding* the graph G.

- If $D[v_s, k + 1] = -\infty$, then no walk from s to v_s of length $k + 1$ has previously been discovered. Set $D[v_s, k+1]$ to $D[v, k] + w_{v,v_s}$, where w_{v,v_s} is the weight of the arc from v to v's successor v_s. Enqueue $< k+1, v_s >$.
- Else, if $D[v_s, k + 1]$ has a value due to a previously discovered walk from s to v_s of length $k + 1$, call this value α and compute $D[v, k] + w_{v,v_s}$. If this sum is greater than α, update $D[v_s, k + 1]$ with α and enqueue $< k + 1, v_s >$. Otherwise, let α stay as the value of $D[v_s, k + 1]$.

When all successors v_s of v have been treated, dequeue Q and let the new $< k, v >$-pair be the basis for the treatment described above, that is, treat all successors of the newly dequeued v in the above-mentioned way.

Break when the first$< k, v >$-pair with $k = n$ is dequeued, since at this point, D holds all the valid $D_k(v)$-values in G for $k = 0, 1, \ldots, n$. It is quite easy to be convinced that unfolding correctly computes the $D_k(v)$-values for each $v \in V$ and $k = 0, 1, \ldots, n$. The following is an outline of a proof by induction.

All nodes at a certain level k are treated before any node at level $k + 1$ (or any other level $k + m$ where $m > 1$) is treated. Every level k is thus *exhausted* before the next level, $k+1$, is considered. To determine a value $D_k(v)$, $D_{k-1}(v_i)$ for all immediate predecessors v_i of v *that there are walks to from* s must have been determined: $D[v_i, k - 1]$, where all v_i are reachable (see Section 4.1) from s and v_i are immediate predecessors of v, must hold the final $D_{k-1}(v_i)$-values before $D_k(v)$ can be computed. How do we know that they do that?

[27]This is true except for the fact that every entry in the matrix is initialized to $-\infty$.

There can be only one node at level 0, that is, the source node s. $D_0(s)$ is of course 0. For all nodes at level $k = 1$, we thus know that all relevant $D[v, k-1]$-entries are final, since $D_0(s)$ is 0 and final. A relevant $D[v, k-1]$-entry in this context is an entry corresponding to a node that is reachable from s, and a level that is one less compared with the level we are currently looking at. For a node v to be at level k, v has to be a successor of a node at level $k - 1$[28].

Let V_k denote the node set of all nodes at level k in a graph G. All nodes $v \in V_1$ are thus successors of s. Their $D_1(v)$-values are equal to the arc weights of the arcs going from s to the respective node[29], since the first term in the sum $D[s, 0] + w_{s,v_1}$ is 0. All $D_1(v)$-values are obviously final. Level 1 is in this way exhausted before level 2 is considered. All nodes $v \in V_2$ are successors of nodes at level 1 whose $D_k(v)$-values are final. $D_2(v)$ for a specific $v \in V_2$ is $\max\left\{D_1(v_p) + w_{v_p,v} \mid v_p \in V_1 \,\&\, a_{v_p v} \in A\right\}$, where A is the set of arcs of the graph G.

Generally,

$$D_k(v) = \max\left\{D_{k-1}(v_p) + w_{v_p,v} \mid v_p \in V_{k-1} \,\&\, a_{v_p v} \in A\right\} .$$

$D_k(v)$ can always be determined if the $D_{k-1}(v_p)$-values have already been determined for all predecessors v_p of v that are reachable from s. The base case of the induction is that $D_0(v)$ is 0, and that only s is at level 0.

7.2 DG1

Dasdan and Gupta call the algorithm that I have chosen to use for the maximum cycle mean computation DG1[30]. Pseude-code for the relevant parts of DG1 is given in Figure 10, extracted from their article [7].

7.2.1 `dg1/3`

`dg1/3` is the overall rule that manages the computation of the maximum cycle mean of a collapsed graph in my implementation. arg1 of `dg1/3` is the node set V of the graph, arg2 the arc set A. arg3 is the output argument, that is, the maximum cycle mean of the graph defined by V and A. `dg1/3` is suitably called with V and A equal to the output arguments, arg6 and arg7, of `collapse/7`.

`dg1/3` calls `dg1/4`, where the extra argument is n, the number of nodes V in the graph $G = (V, A)$. n provides the condition of when to break and apply

[28]A node v can of course be at many levels at the same time, since there can be walks of different lengths from s to v. Nevertheless, it is always true that v has to be a successor of a node at level $k - 1$ to be at level k.

[29]Multiple arcs between the same two nodes, in the same direction, are not allowed.

[30]I suppose that D stands for Dasdan, and G for Gupta. They also propose a second variant, DG2, that they claim to be faster than DG1 in practice. They say that the approach in DG2 can be better optimized by a compiler. DG1 is however faster in theory (according to them), without taking compiler optimization into consideration. I chose to implement DG1 since it is easier to see what it does and since it is not obvious that the compiler optimization applies to my Prolog code anyway: optimizing the code has not been given highest priority in my implementation (see Section 3.1).

```
/* Head */
for each node v ∈ V do
   LastLevel[v] ← -1
s ← FindSource(G)
D[0,s] ← 0
Valid[0,s] ← -1
LastLevel[s] ← 0
Enqueue(Q,<0,s>)

/* Body */
<k,v> ← Dequeue(Q)
do
   for each successor node u ∈ AdjOut[v] do
      if (LastLevel[u] < k+1) then
         Enqueue(Q,<k+1,u>)
         Valid[k+1,u] ← LastLevel[u]
         LastLevel[u] ← k+1
         D[k+1,u] ← −∞
      if (D[k+1,u] < D[k,v]+w(v,u)) then
         D[k+1,u] ← D[k,v]+w(v,u) /* max */
   <k,v> ← Dequeue(Q)
while (k < n)

/* Tail */
λ ← −∞
for each node v ∈ V do
   if (LastLevel[v] = n) then
      M[v] ← +∞ /* the identity for min */
      k ← Valid[n,k]
   while (k > -1) do
      if(M[v]>(D[n,v]-D[k,v])/(n-k)) then
         M[v] ← (D[n,v]-D[k,v])/(n-k) /*min*/
      k ← Valid[k,v]
   if (λ < M[v]) then
      λ ← M[v] /* max */
return λ
```

Figure 10: Pseudo-code for DG1

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% dg1(+Nodes,+Arcs,+N,-MeanCycle) <--
%%
%%     N is the number of nodes in the list
%%     Nodes, and is used to signal to
%%     process_graph/12 when to stop and
%%     return a result.  dg1/4 does a few
%%     initializations, creates a queue and
%%     then passes on to process_graph/12 and
%%     compute_mean/6 to do the rest of the
%%     work.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dg1(V,A,N,Meancycle) <-
    select_source(V,S),   % S can be any
                          % node in the
                          % node set V
    ...
    %% initializations %%
    ...
    create_queue(Q),
    process_graph(S,0,V,A,N,Q,
                  LLPrel,DPrel,ValidPrel,
                  LL,D,Valid),
    compute_mean(V,N,LL,D,Valid,Meancycle).
```

Figure 11: Prolog-like pseudo-code for `dg1/4`

Karp's Theorem to the $D_k(v)$-values collected. `dg1/4` first initializes the structures that are needed to keep track of all that must be documented in order to get hold of the $D_k(v)$-values and compute the maximum cycle mean efficiently. Once again: declarations are not at all required in Prolog. Initializations are however helpful, and I choose to initialize these structures although there are other ways to achieve the same functionality[31]. `dg1/4` then calls `process_graph/12` to compute all relevant $D_k(v)$-values, and finally `compute_mean/6` that basically applies Karp's Theorem on the result of `process_graph/12`.

Figure 11 shows the basic design of `dg1/4`, in a Prolog-like pseudo-code. The row denoted "`initializations`" refers to the initializations of several structures that play important roles in `process_graph/12` (see below).

A few words can be said about the structures that are introduced by Dasdan and Gupta in DG1.

[31] Instead of initializing every entry in (a structure simulating) a matrix to the atom `nil` to indicate the lack of a valid value in that particular position – an atom that could be called `null`, or `mumbojumbo` for that matter, since nil/null is not defined in Prolog – there are other ways of finding out that a certain entry has not yet received a valid value that I could have chosen, but that I did not choose.

Q A queue with elements made up of level-node pairs. The queue is denoted `Q` in my implementation.

LastLevel A structure holding key-value pairs, with the keys being node names (of nodes in `V`) and the values the last level at which the node corresponding to the key has so far been processed, or treated. Every node at a particular level k is treated at that level before any node is treated at level $k+1$. This structure is denoted `LL` in my implementation.

D This structure can be viewed as a matrix $D[k, v]$, keeping track of triplets, each consisting of a level k, a node v and a value, or weight, w, where $w = D[k, v]$. The semantic reading is that "node v has weight w at level k". The level represents how many arcs this particular walk from the source node (see `dg1/4`) to the node v consists of. The weight is the maximum weight of all walks of length k from the source node to v so far discovered, if there are more than one. `D` in my implementation is the same as $D[k, v]$.

Valid A structure similar to D in its construction, but holding triplets consisting of a level k, a node v, and a value val with quite a different meaning from $D[k, v]$ above. val, that is, $Valid[k, v]$, is a value that states whether the node v at level k is 'valid' or not. Once $Valid[k, v]$ is set, it means that the node v actually has a value in D at level k. The actual value val, $Valid[k, v]$, is the previous level at which node v has a value in D. $Valid[k, v]$ is set equal to -1 if k is the first level for which $D[k, v]$ has a value[32]. Using $Valid$, we can easily find all valid entries in D. This structure is denoted `Valid` in my implementation.

7.2.2 `process_graph/12`

The procedure that actually *unfolds* the graph (see Section 7.1) is `process_graph/12`. The code – this time real code, not pseudo-code – is given in Figure 12. Here follows a thorough explanation of it.

Input to `process_graph/12` is, among other things, a node `Node` and a level `K`, corresponding to a newly dequeued $< k, v >$-pair from the queue `Q`. `process_graph/12` does the work that is called "the body" in DG1 (see Figure 10).

`process_graph/12` first calls `list_successors/4` on v to get a list of all successors of `Node` in the graph, and is the equivalent of `AdjOut[v]` in DG1. The list of successors of `Node` is passed to `do_successors/12`, that treats every successor, updates relevant structures and enqueues new $< k, v >$-pairs in the manner previously described in the context of the unfolding of a graph (see Section 7.1). `do_successors/12` thus basically does the work of the for-loop in the body of DG1.

[32] Any value, or atom, that is not a possible level number could of course be used.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% process_graph(+Node,+K,+V,+A,+N,+Queue,
%               +LLPrel,+DPrel,+ValidPrel,
%               -LastLevel,-D,-Valid) <--
%
% LLPrel, DPrel and ValidPrel are the
% structures updated with the latest
% information, due to the <k,u>-pair latest
% dequeued from Queue.  (The meaning of K is
% made clear from the description  of the
% structure D above).  N is simply the number
% of levels to exhaust (see above) before we
% can be sure that LLPrel, DPrel and ValidPrel
% reflect the contents of the whole graph.  In
% the base case, these three arguments (arg
% 7-9) unify with the output arguments,
% LastLevel, D and Valid (arguments 10-12).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

process_graph(_,N,_,_,N,_,     % The base case
              LL,D,Valid,
              LL,D,Valid).
process_graph(Node,K,V,A,N,Q,
              LLPrel,DPrel,ValidPrel,
              LL,D,Valid) :-
    list_successors(Node,V,A,Successors),
    do_successors(Successors,Node,A,K,
                  Q,LLPrel,DPrel,ValidPrel,
                  QTemp,LLUpd,DUpd,ValidUpd),
    dequeue(i(KNew,NodeNew),QTemp,QNew),
    process_graph(NodeNew,KNew,V,A,N,QNew,
                  LLUpd,DUpd,ValidUpd,
                  LL,D,Valid).
```

Figure 12: The code for `process_graph/12`

When the preliminary `D`-, `LL`- and `Valid`-structures have been updated due to the successors of `Node`, a new $< k, v >$-pair is dequeued, and `process_graph/12` is called on the new `Node` and `K`.

`process_graph/12` is a recursive procedure (with accumulating parameters) that treats one $< k, v >$-pair per call. `V` is the node set and `A` the arc set of the graph. `LLPrel`, `DPrel` and `ValidPrel` are preliminary versions of the final structures `LL`, `D` and `Valid`, updated with the information gained by processing the last dequeued $< k, v >$-pair from `Queue`. The base case is reached when `N` levels have been exhausted, where `N` is the number of nodes in the graph. At that point, `LLPrel`, `DPrel` and `ValidPrel` reflect the contents of the whole graph, and their values are passed on to the output arguments `LL`, `D` and `Valid`.

7.2.3 `compute_mean/6`

The only thing `compute_mean/6` does, is pass on its input arguments to a call to `compute_mean/8`. Of the input arguments to `compute_mean/8`, `M` and `Lambda` are the extra arguments compared with the input arguments of `compute_mean/6`, and they are also the only ones that have not already been used earlier in the algorithm. `M` is an association list used to keep track of node-value pairs; $M(v)$ is the outcome of the minimization operation of Karp's Theorem applied to v.

$$M\,(v) = \min_{0 \leq k \leq n-1} \frac{D_n(v) - D_k(v)}{n-k} \,.$$

`Lambda` simply holds the maximum $M(v)$-value discovered so far. When all nodes in the graph have been processed by `compute_mean/8`, `Lambda` (arg7) is equal to the maximum cycle mean and is in the base case made to unify with the output argument (arg8). This corresponds to the maximization operation in Karp's Theorem (see Section 4.4.1).

The code for `compute_mean/8` is shown in Figure 13. `compute_mean/8` takes care of one node at a time, and does the work of "the tail" in DG1 (see pseudecode in Figure 10). The first version involves a test that checks whether the last level at which the current node was processed in `process_graph/12` was equal to `N`, the number of nodes in the graph, or not. If this test succeeds, the minimization operation of Karp's Theorem is applied to that node. Otherwise, the call to `compute_mean/8` succeeds only in the second version (the cut in the first version prevents the second version from being tried on backtracking on calls for which the first version succeeds), whose task is to only call `compute_mean/8` on the rest of the nodes of the graph.

Why does not $M(v)$ need to be computed if there is no walk from the source node to v of length n, where n is the number of nodes in the graph?

The reason can be found in the formulation of Karp's Theorem. If $D_n(v)$ is $-\infty$ for a particular v, $M(v)$ is automatically $-\infty$ for that v. To see this, note that the computation of $M(v)$ involves a minimization operation. Nothing can be smaller than $-\infty$, which means that we can be sure that $M(v)$ will be $-\infty$ if the fraction $(D_n(v) - D_k(v))/(n-k)$ takes on this value for the current node v for at least one k between 0 and $n-1$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% compute_mean(+Nodes,+N,+LastLevel,
%%              +D,+Valid,+M,+Lambda,
%%              -MeanCycle) <--
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

compute_mean([],_,_,_,_,     % The base case
            _,Res,Res).
compute_mean([V|Vs],N,LL,D,Valid,
             M,Lambda,Res) :-
    get_value(V,LL,
              LLValue),     % Find the last
                            % level at which
                            % node V was
                            % processed.
    LLValue == N,           % Apply the min
                            % operation of
                            % Karp's Theorem on
                            % node V only if
                            % LLValue is equal
                            % to N
    !,
    karps_theorem(V,D,Valid,N,
                  Lambda,M,
                  Lambda2,M2),
    compute_mean(Vs,N,LL,D,Valid,   % Process
                 M2,Lambda2,Res).   % next
                                    % node
                                    % in the
                                    % list.

%% If the last level at which node V was
%% processed by process_graph/12 was not N
%% (i.e., the last level exhausted), do NOT
%% apply the min operation of Karp's Theorem on
%% V.  Just process the next node in the
%% list.

compute_mean([_|Vs],N,LL,D,Valid,
             M,Lambda,Res) :-
    compute_mean(Vs,N,LL,D,Valid,
                 M,Lambda,Res).
```

Figure 13: The code for `compute_mean/8`

The only way for the fraction to equal anything else than $-\infty$ is if $D_k(v)$ also is $-\infty$, for some k, in which case the fraction would possibly (but not necessarily) take on a real value[33]. $D_k(v)$ is probably $-\infty$ for several values on k in an average graph, but $M(v)$ will however only possibly take on a real value if $D_k(v) = -\infty$ for *all* possible values on k. And this is not possible since the underlying graph is strongly connected. In any strongly connected graph, there must be at least one walk from any node to any other node of length n or shorter. If the walk is not of length n, then it is imperative that there must be a walk of shorter length between the two nodes in the graph. This ensures that $D_k(v)$ has a real value for at least one k between 0 and $n-1$ even when $D_n(v)$ does not, which in turn makes the fraction for that k equal $-\infty$. Thus, $M(v)$ is $-\infty$ if $D_n(v) = -\infty$.

The maximum cycle mean of a strongly connected digraph G is the maximum $M(v)$-value of G. As long as there are arcs in G, that value is real. Any real value is greater than $-\infty$, so if $M(v)$ is $-\infty$ for all nodes v in the graph G for which $D_n(v)$ is $-\infty$, these $M(v)$-values do not affect the maximum cycle mean of G.

Therefore, the fraction $(D_n(v)-D_k(v))/(n-k)$ does not need to be computed when $D_n(v) = -\infty$. The same applies to the case when $D_n(v) \neq -\infty$ and $D_k(v) = -\infty$, since the fraction will take on the value $+\infty$, the identity for the minimization operation of $M(v)$, in those cases.

The purpose of the structure $Valid$ is to keep track of those level-node combinations for which $D_k(v)$ is a real number. The while-loop of the tail of DG1 from the previous section, here shown again, shows how the fraction is only computed for the level numbers corresponding to levels at which $D[k, v]$ contains a real number.

```
k:=Valid[n,v]
while (k > -1) do
   fraction:=(D[n,v]-D[k,v])/(n-k)
   if(M[v] > fraction) then
      M[v]:=fraction
   k:=Valid[k,v]
```

Remember that the value of $Valid[k, v]$ is equal to the 'next' (previous) value of k for which $D_k(v)$ is defined, with $Valid[k, v]$ equal to -1 if k is the first level for which $D[k, v]$ has a value. *Valid* and *LastLevel* are defined in Section 7.2.1.

In my implementation, the while-loop above corresponds directly to the procedure `karp/7`. Figure 14 gives the code for `karps_theorem/8`, called from `compute_mean/8`.

7.2.4 Selecting a suitable source

As can be noted, the correspondence between Dasdan and Gupta's DG1 [7] and my implementation is almost total. What is left is to write a procedure

[33]This is because $-\infty - (-\infty)$ is not well-defined.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% karps_theorem(+V,+D,+Valid,+N,
%%               +Lambda,+M,
%%               -LambdaNew,-MNew) <--
%%
%%               karps_theorem/8 initializes
%%               M[V] to nil before sending
%%               the node V along with
%%               relevant structures to
%%               karp/7.  karp/7 finds the
%%               appropriate value for M[V]
%%               Lambda, a variable that holds
%%               the so far largest value
%%               encountered in the structure
%%               M, is then passed to
%%               update_lambda/4 that updates
%%               Lambda, if needed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

karps_theorem(V,D,Valid,N,
              Lambda,M,
              Lambda2,M2) :-
    add_to_relation(V,M,nil,MNew), % nil is
                                   % used to
                                   % represent
                                   % −∞
    get_value(Valid,N,V,K),
    karp(V,D,Valid,N,K,MNew,M2),
    get_value(V,M2,M2value),
    update_lambda(Lambda,M2value,Lambda2).
```

Figure 14: The code for karps_theorem/8

that selects a suitable source node. Choosing an arbitrary node works. My implementation just picks the first node in the list of nodes. This is of course not the most efficient way of computing the maximum cycle mean, since the unfolded graph will contain a different number of arcs depending on the choice of source node. The fewer, the better.

If finding the best source, or at least a good one, takes more time than executing the algorithm with a badly chosen source, just picking any node to be the source node is of course preferable. Dasdan and Gupta resorts to a fast heuristic. They unfold the graph from each node for a given number of iterations, and choose the node that leads to the fewest arcs.

It is of course possible to use another approach for the implementation of DG1. Just like `collapse/7` does, `dg1/4` could use a procedure that computes the "heights" of all nodes with regard to an arbitrarily chosen node, and then use this information to optimize the rest of the algorithm. Then a queue would not be necessary, and less unnecessary work would need to be performed. The concept of height would however first need to be modified to agree with cyclic graphs.

Ekman and Kreuger state that the complexity of finding the heights of the nodes of the split graph $S(G_{con})$ is $O(m)$, where m is the number of arcs in $S(G_{con})$ [9]. Computing the heights of the nodes of a graph is thus not particularily expensive, but not for free either. I have deemed it to be definitely worthwhile in `collapse/7` since the result of the one computation is used in every call to `max_weight_paths/4.` In `dg1/3`, the technique of unfolding the graph, with the potential of being combined with the identification of a really suitable source, will at least suffice for now.

8 The condition graph

This section deals with the second problem I was asked to look at: how can a description of a traffic pattern be translated into a graph suitable for the algorithm that computes the cycle time (the inverse of the capacity) of the traffic pattern? In other words, my task has been to work out an algorithm that finds the condition graph from a description of a traffic pattern.

In the following, note that the model does not take any operational issues into consideration. No train movement is unpredictable in any way. The assessment of capacity assumes a perfect, undisturbed, flow of trains.

8.1 Describing a traffic pattern

The first issue to resolve is what actually is required from the description of the traffic pattern in order to build a 'correct' condition graph. I will suggest, along the lines that Ekman and Kreuger have set up (see [9]), a set of requirements that allows me to define a clear-cut, although simplified, algorithm that should be useful for Banverket.

8.1.1 Signals and track segments

The advance of a train through a track system is controlled by signals. The main purpose of a railway signalling system is to maintain a safe distance between trains, and to ensure the safe movement of trains over the entire network – even in the event of equipment or system failure [19]. Safety is, and should be, the highest priority when placing signals. The locations of the signals on a railway section are however vital for the capacity on it.

The influence of one train on another is communicated through signals. The advance of a specific train through a certain railway section can be uniquely determined by specifying what signals that train passes and what track segments the train traverses. The track segments alone, and the order in which they are occupied by the train, do not necessarily specify the advance of the train uniquely. This is due to the fact that a track segment can serve as a switch. Since the actual path is determined by which one of the branches of the switch the train uses, it is clearly not always enough to specify the track segments to describe the path of a train.

Ekman and Kreuger define the term *path* in terms of both signals and track segments. It is important to keep in mind what branch of a switch a specific movement of the traffic pattern uses when it is time to find the arc weights of the condition graph. Finding the actual arc weights is a completely different problem from finding what arc weights that will be needed. For the latter problem, a simplified definition of the term path suffices, since I am only interested in whether a certain track segment is occupied by a specified movement or not. What branch of a switch a specified movement uses is therefore not interesting.

With Ekman and Kreuger's definition slightly modified, a *path* consists of a finite number of track segments $t_1, \ldots, t_n$ and two signals. t_1, the *beginning* *segment*, is preceded by the *beginning signal*, and t_n, the *closing segment*, has a *closing signal* at its very end. The closing signal thus precedes the track segment that succeeds the closing segment. If the path consists of a single track segment, the beginning and closing segments coincide. Beginning and closing segments will sometimes be referred to as *end segments.* A path has only one end segment if and only if it consists of a single track segment.

I have no deeper knowledge of signal engineering. Instead of trying to speak a language I do not know, I have chosen to look at the problem in an abstract way. I realized that certain rules apply if a number of feasible assumptions can be made. These rules are well-defined (they make the problem and its solution unambiguous), and more importantly, *the assumptions are compatible with signal engineering.* I base that conclusion on the fact that I only give a *suggestion* on exactly how one determines that two movements are 'in conflict' (see page 66). It is possible to introduce another definition (based on signals, for instance), and the rest of my proposed algorithm will still be valid and applicable.

8.1.2 Cleared paths

A stop and its duration are always planned, and should be distinguished from the *waiting* that a train can be forced to do when another train is given precedence, that is, has a higher priority on a section of the track that both trains want access to at the same time.

Waiting point is an important concept. A waiting point can be viewed as a conceptual point coinciding with the seam between two specified track segments. A specific train is only allowed to wait for trains with higher precedence at waiting points that are associated with the movement that the train is undergoing.

Physically, 'at a waiting point' means that a train has to come to a full stop *before* the specified seam, the waiting point, between the two track segments is reached. Then the train must stand there and wait for other trains, whose movements have higher precendence, to first occupy sections of the track that it will need in the future.

The following defines Ekman and Kreuger's term *complete train movement*: a train path through a specified railway section, a train type, the track segments on which the train makes (planned) stops and the durations of those stops. A *train path*, or *path* for short, is in my report defined as a sequence of track segments, in the order in which the train undergoing the specified movement traverses them. The activity of a traffic pattern is represented by the set of complete train movements of the traffic pattern.

The path of a complete train movement is partitioned by its waiting points into several, consecutive paths, called *cleared path*s. The path of a complete train movement that has zero waiting points is also a cleared path. A train movement on a cleared path is from this point on simply referred to as a *movement.* Note that a cleared path is, in my report, merely defined as a succession of track segments. This means that two different movements of the same cycle might use the same cleared path, or the same cleared paths. Situations where different movements traverse the same cleared path will come up in every traffic pattern, since a movement x in cycle n is a different movement from x in cycle $n+1$ although they traverse the same track segments in the same order.

Note that a train can only wait *before* a certain movement starts. In other words, a movement is not allowed to start if any other train will force the train to wait somewhere along the cleared path. The movement can start earliest at the instant at which the whole path of the movement, the cleared path, is 'cleared' in a conceptual sense: it does not need to be cleared of trains, but it must be in such a state that the movement that is about to start start traversing it can regard the path as if it were in fact cleared of trains. All the time a train spends standing still, waiting for the cleared path that it is about to traverse to be 'cleared' in the way described above, is classified as waiting.

Contrast the definition of a cleared path with that of a train route. A *train route* is a part of the railway section that must not be occupied by more than one train at the same time. A cleared path might be occupied by more than one train at the same time, as long as they do not interfere with each other in any way.

The above correctly suggests that waiting occurs between movements. If this were not the case, it would be impossible to find the weights of the arcs in the condition graph, since they would include indeterminate waiting times[34].

The physical location of a train that is waiting is considered to be on the cleared path associated with the movement of the same train that immediately *precedes* the movement that is waiting to start (see page 63 for a definition of predecessor movements). During the waiting, at least the front of the train occupies the last track segment of the cleared path associated with the movement that concludes at the waiting point, and no part of the train occupies the track segment that succeeds the waiting point. This is compatible with the idea that a train is not considered to be on the railway section before it is allowed to enter it.

I illustrate with a simple example. Let a denote the path that is associated with the complete movement undergone by train a. It should be obvious from the context when a refers to the path and when it refers to the train. Imagine a being divided into two cleared paths, here denoted a_1 and a_2, in that order, by a waiting point called $a_1 : a_2$. It is straight-forward what the weight of arc $a_{a_1 a_2}$ means: the time it takes for train a (in any cycle) to traverse, in the specified order, all track segments included in the cleared path a_1, including the planned duration times of all stops by train a on a_1, but *excluding* any waiting time for train a at the waiting point $a_1 : a_2$.

8.1.3 The boundaries of a railway section

Where a complete train movement (defined in the previous section) actually starts and ends is of course a dubious question. It is only obvious when a closed

[34]Recall that the weight of the arc from node a to node b expresses how many time units must pass after movement a has started before movement b is allowed to start.

system is considered, in which every complete train movement starts where it ends. The method for capacity assessment proposed by Ekman and Kreuger is not intended for closed systems. Ekman and Kreuger's suggestion on how to define the boundary of the railway section that is about to be analyzed gives rise to some difficulties that I have to resolve.

The number of movements of one cycle of a traffic pattern is finite. The railway section concerned by the traffic pattern is made up of at least the track segments that are included in a path of a complete train movement in the traffic pattern. The railway section is deliberately vaguely specified by Ekman and Kreuger. The reason is that slightly different patterns of traffic should be made comparable by being said to concern the same railway section although the paths of the different sets of complete train movements involved do not necessarily cover the exact same track segments. It is however essential to my algorithm to know exactly what track segments of the railway section will be occupied by trains, and exactly what trains any of these individual track segment will be occupied by.

The trains of every complete train movement must both *enter* the railway section and *leave* it, and they can only do so at a limited number of locations. As intuitive as this might seem, it is possible that a track segment serves as an end segment for only some of the paths of which it is a part.

See Figure 15 for an example of this. It displays three track segments. The middle one serves as a switch. There are three points at which a train can enter or leave this railway section, marked L1, L2 and R in the figure. The segment in the middle might serve as an 'interior' track segment for a train entering at L1 and leaving at R (or vice versa). But for trains either entering or leaving the railway section at L2, it serves as an end segment.

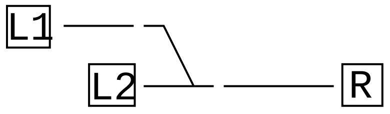

Figure 15: A simple railway section, illustrating the problematic term *end segment*

Ekman and Kreuger define the boundaries of a railway section by identifying its *main signals*. Every complete train movement must pass exactly two main signals on its way through a railway section: one on its way in, and another on its way out. A main signal serving as a beginning signal for one complete train movement, cannot serve as a closing signal for another movement.

A complication of this definition is that a movement might physically traverse track segments that are not part of its path-definition, but part of a path of an oppositely directed movement and therefore one of the track segments used by the traffic pattern. See Figure 16.

In Figure 16, sig1 is the beginning signal of movement x, and sig2 its closing signal. sig3 and sig4 are the beginning and closing signals of movement y.

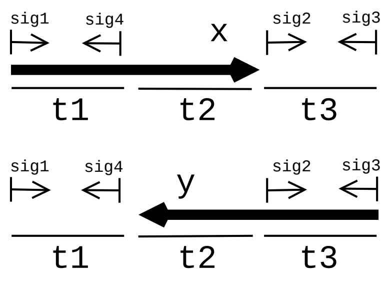

Figure 16: Problematic placement of beginning and closing signals

The cleared path of movement x consists of track segments t_1 and t_2[35], and the cleared path of y consists of t_3 and t_2. The train undergoing movement x must traverse t_3, and the train undergoing movement y must traverse t_1, but these track segments are still *not formally included* in the paths of their movements.

The track segments of a traffic pattern For the purpose of finding a sufficient set of conditions, I need an exact definition of the boundaries of a railway section. I require the main signals to be placed in such a way that all track segments of the traffic pattern that a movement uses are always included in the path definition of the movement. This requires a clear-cut definition of what I mean by the track segments of a traffic pattern.

The track segments of a traffic pattern are exactly those track segments that are used by at least one complete train movement of the traffic pattern. It always suffices to look at the movements of one cycle of the pattern to determine what track segments are used by the it.

Moreover, an end segment of a path of a complete train movement must serve as end segment for every other path whose movement uses it, unless it is a switch. If it is a switch, it might serve as an end segment for one complete train movements and as an interior segment for another only if they use different branches of the switch. See Figure 15.

Note that the restriction applies to *end segments of paths of complete train movements*, not to end segments of just any cleared path. The distinction is very important, since segments often serve as end segments of some cleared paths but not for others.

The purpose of imposing this restriction is simply to force the user to decide what track segments are used by the traffic pattern, and let all movements that use a specified track segment have their path definitions include that track segment. This is summarized by the following assumption.

[35]Track segments are denoted `t1`, `t2`, ... instead of $t_1, t_2, \ldots$ in the figure only because the program generating the figures could not handle subscripts.

Important assumption I will assume that the main signals are always placed in such a way that every path consists of all track segments of the traffic pattern that the train that uses the path actually traverses.

The traffic pattern illustrated in Figure 16 can for example be defined in this way (Figure 17).

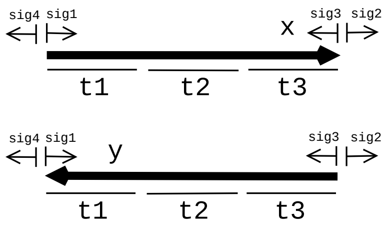

Figure 17: A traffic pattern exemplifying where the main signals will always be assumed to be located

The only difference from the previous figure is the location of the main signals, that now adheres to the above-mentioned assumption: the position of a main signal, acting as the beginning signal for a complete train movement in one direction, is used also for a main signal acting as a closing signal for other complete train movements, traversing the railway section in the other direction.

In the example, the two trains x and y still traverse the same track segments as before, but now both paths formally include all three track segments instead of two each (with only one coinciding). This shows more accurately what track segments are used by what movements, and simplifies the following discussions significantly.

8.1.4 Notational issues

Before proceeding to the next topic, I need to resolve some notational issues.

The path of complete train movements will from this point on be denoted by lowercase letters. A complete train movement is not associated with any specific cycle of the traffic pattern. The existence of a complete train movement using path x, merely states that in every cycle of the traffic pattern in question, a train of a specified type will occupy the track segments that are specified by x, make stops of specified durations on specified track segments, and be allowed to wait for other trains at specified locations (waiting points).

Waiting points partition the paths of complete train movements into cleared paths. The cleared paths whose 'parent' is the path of a complete train movement x, will be named $x_1, x_2, \ldots, x_n$, in the order of appearance in one cycle of the traffic pattern: x_1 starts with the beginning segment of the path of the complete train movement x, and x_n ends with the closing segment of x. This means that a cleared path can have many names – one per complete train movement that uses the path.

A movement that uses a cleared path x_i is denoted by (x_i, n), meaning the movement using the cleared path x_i in cycle n of the traffic pattern. It is often superfluous to specify the cycle of a movement when discussing properties that apply to a movement using x_i of any cycle. Therefore, I will sometimes use x_i to denote (x_i, n), since it makes the account more easy to read. The movements $x_1, x_2, \ldots, x_n$ will sometimes be referred to collectively as the *x-movements*. It will be clear from the context when x_i refers to the cleared path and when it refers to the movement using the cleared path x_i. (x_i, n) always refers only to the movement of cycle n using the cleared path x_i.

Regardless of whether certain things apply to the movement using x_i of any cycle or not, the movements (x_i, n) and (x_i, m) are always different unless $n = m$. Different movements might be identical in every possible way, that is, use the same cleared paths, make stops on the same track segments and with the same duration, concern the same train type, and end or start at the same waiting points. As a rule of thumb, two movements are considered to be different as soon as it is physically possible that two different trains are used for them.

Waiting points A seam between two track segment that serves as a waiting point for one complete train movement of the traffic pattern, does not necessarily serve as a waiting point for any other complete train movement (see Section 8.1.2 for a discussion about waiting points). A waiting point that serves as a waiting point for only one complete train movement of the traffic pattern is *simple.*

Definition Let a path a of an arbitrary complete train movement be divided by waiting points into the cleared paths $a_1, \ldots, a_n$. If the seam between a_i and a_{i+1} serves as a waiting point for the complete train movement using a, but not for any other complete train movement in the traffic pattern, $a_i : a_{i+1}$ is a *simple waiting point.* The movements a_i and a_{i+1} are said to *share* a waiting point.

Simple waiting points can be compared with joint waiting points. A *joint waiting point* is a seam between track segment that serves as a waiting point for more than one complete train movement of the traffic pattern.

Definition Let two different, arbitrary complete train movements use paths a and b. The paths are divided by waiting points into the cleared paths $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ respectively. Imagine that some of the cleared paths originating from a and b share end segments. Let one of the waiting points that partitions a be also a waiting point of b. Let this waiting point be located at the seam between the closing segment of a_i and the beginning segment of a_{i+1}. If this seam corresponds to the seam between b_j and b_{j+1}, then $a_i : a_{i+1}$ and $b_j : b_{j+1}$ form a *joint waiting point.* It can be called either $a_i : a_{i+1}$ or $b_j : b_{j+1}$. The movements a_i, a_{i+1}, b_j and b_{j+1} are all said to *share* a (joint) waiting point.

Successor and predecessor movements I will sometimes refer to movements as the successor or predecessor movements of another specified movement. Successor and predecessor movements *do not* refer to movements that simply happen 'before' or 'after' a specified movement. (a_i, n) is indeed initiated before $(a_i, n+1)$, but the first will not be referred to as a predecessor movement of the latter.

Definition Let a complete train movement use the path a, and let a be divided into $a_1, \ldots, a_n$ by $n-1$ waiting points. a_i is a *predecessor movement of* a_j if and only if $i < j$; a_i is a *successor movement of* a_j if and only if $i > j$. A movement cannot be a predecessor or successor movement of a movement if the two movements do not both originate from the same complete train movement, or if they concern two different cycles.

8.1.5 Precedence

As soon as a situation in which the activity of one train affects the activity of another train in the traffic pattern comes up, precedence has to be given to one of the trains. If three or more trains all influence each other in an interfering manner in the same railway section, a "pecking order" has to be decided, saying what train is allowed to occupy the area of conflict first, second and third, and so on. It is of course always enough to consider every pair of movements separately, and the pecking order – a mathematically well-defined *partial order* on the set of movements – will crystallize as a side effect.

In the previous paragraph, I have taken for granted not only that a pecking order exists. I have also taken for granted that there is such a thing as a 'first' movement. Alternatively that a whole set of movements that theoretically can be undergone simultaneously can be thought of as occurring 'before' all others, but in no particular mutual order.

The traffic pattern is cyclic. This means that any movement (or set of mutually independent movements, as described above) in the pattern can be arbitrarily chosen to be 'first'. The 'first' movement (or set of 'first' movements) does however not play any other role apart from simply being a reference point, indicating where the (arbitrarily chosen) cycle boundary is. This reference point is needed in the following discussion.

Definition Let x_i and y_j be two arbitrary movements among the movements originating from the set of complete train movements describing the activities of the traffic pattern. A set of *first movements* F of the traffic pattern can be chosen in any way, as long as the following criteria are met

- $F \neq \emptyset$
- If $x_i \in F$, then $y_j \in F$ if and only if the pecking order between x_i and y_j is undefined.

Recall that the set of complete train movements of a traffic pattern describes the activities of the traffic pattern by representing what happens in every cycle of it.

8.1.6 Graph representation

The precedence relation on the set of movements of a traffic pattern can always be found by separately considering all pairs of movements of two consecutive cycles of it. Every two movements are not necessarily related, in which case the pair in question does not give any new information about the pecking order.

Every movement of one repetition (one cycle) of the traffic pattern corresponds to a node in the graph, whether it is the traffic pattern graph or the condition graph. The precedence relation on the set of movements of just one cycle, n, of the traffic pattern is represented by straight arcs in the graphs. Bowed arcs represent precedences where a movement has higher precedence than a movement *in the next cycle.*

A path in the graph, consisting of only straight arcs (possibly just one arc), from node x to node y (where $x \neq y$), means that the movement corresponding to x has higher precedence than the movement corresponding to y – *in the same cycle.*

If, when looking at the pecking order among the movements of only one cycle, movement x has higher precedence than y, denoted $x < y$[36], but lower precedence than z, denoted $z < x$, then the following is true

- $(z, n) < (x, n) < (y, n)$
- $(x, n) < (x, n + 1)$
- $(z, n) < (y, n + 1)$

etc. The notation (x, n) is explained in section 8.1.4, *Notational issues.*

The purpose of the traffic pattern graph is to summarize the precedence relation on the movements of a traffic pattern in a condensed way. A precedence is therefore not represented by an arc in the traffic pattern graph if it is implied by other precedences. For instance, $(z, n) < (y, n)$ is implied by $(z, n) < (x, n) < (y, n)$ above, and there should not be an arc from the node corresponding to z to the node corresponding to y in the traffic pattern graph, although $(z, n) < (y, n)$ surely holds. A precedence of the type $(z, n) < (y, n)$ above is instead represented by a path (of length greater than 1) in the traffic pattern graph.

All precedences of a traffic pattern can thus be derived from a traffic pattern graph, since (x, n) has higher precedence than all movements whose nodes are reachable from the node representing x, where every bowed arc that is passed in the walk from x means that a cycle boundary has been crossed.

[36] I use the notational convention introduced by Ekman and Kreuger for precedences. $x > y$ might seem to be a more intuitive representation of x having higher precedence than y, but I do not want to confuse readers that are already familiar with Ekman and Kreuger's terminology by reversing the symbol. "$<$" in $x < y$ indicates that the train undergoing movement x must enter the area of conflict at an earlier point in time than the train undergoing movement y.

8.1.7 Definition of a traffic pattern

Now I have defined everything to be able to explain what kind of description of the traffic pattern that is required in order to uniquely define a condition graph based on the pattern.

- A set of complete train movements of (one cycle of) the traffic pattern and their respective paths, describing all the activities of the traffic pattern.
- All waiting points of the traffic pattern, the resulting movements and their cleared paths. Every movement must 'know its parent', that is, it is essential to keep track of what complete train movement a certain (sub-)movement is associated with.
- A set of first movements F.
- The "pecking order" among the movements of two consecutive cycles of the traffic pattern, that is, a partial order on the set of movements that uniquely determines what movement is allowed to go first if any two movements from two consecutive cycles need access to the same track section at the same time.

The pecking order can be represented by a traffic pattern graph (see Section 8.1.6). I will always try to choose a set of first movements that is as small as possible (preferably consisting of a single movement).

8.2 Traffic pattern graphs vs. condition graphs

8.2.1 Equivalent condition graphs

A *weighted condition graph* is defined by Ekman and Kreuger as a weighted digraph in which the nodes represent the movements of a certain traffic pattern, and the arcs and their respective weights represent conditions on the relative starting times of the involved movements. There is a one-to-one correspondence between the movements of one cycle of the traffic pattern in question and the nodes in the condition graph.

As I said in Section 8.1.5, many different sets of movement in the pattern can be chosen to be 'first', although once the decision is made, the set must not be changed. The cycle time of the traffic pattern is obviously independent of that choice. Luckily, the cycle time of the condition graph is also independent of this. The choice only affects what arcs in the condition graph that are straight and what arcs are bowed, since the bowed arcs indicate a cycle boundary. A node representing a 'first' movement of the pattern will be hit by at least one bowed arc.

It is important to point out that any single cycle in a condition graph will contain the same number of bowed arcs (with a minimum of one) regardless of what set of movements is considered to be first in the pattern. Different choices merely displaces the bowed arcs of the cycles. Note that if this were not the case, the cycle time would not be the same for different choices of sets of first movements, which would of course be absurd. The choice made only affects what type – straight or bowed – any individual arc will belong to. The number of arcs, their starting and ending nodes, and their respective weights will be the same irrespective of the choice of set of first movements.

8.2.2 Arc weights

How to find the arc weights of the condition graph is beyond the scope of this report. Ekman and Kreuger suggest in their report how arc weights can be approximated. A summary of their account on this subject does not serve any purpose here. I will in this section simply make it clear exactly what the weights of the arcs of the condition graph represent.

Consider a precedence denoted by $x < y$. The relationship implicitly represents a condition, denoted $c(x < y)$. The condition says that movement y must not start in such a way it impedes movement x, and vice versa. Moreover, track sections that x and y share, must be occupied by the train undergoing movement x before they can be occupied by the train undergoing y.

Let $start(x)$ denote the point of time when movement x starts. Then the arc representing $c(x < y)$ has weight w, where

$$start(y) \geq start(x) + w$$

Note that y is *not forced* to start at time $start(x) + w$ – it is *allowed* to start *earliest* at this point of time.

8.3 Finding a set of sufficient conditions

The purpose of this section is to present a step-by-step algorithm that will generate a set of conditions that can serve as a starting point for the arc set of the final condition graph. The algorithm basically assesses every feasible condition and lets every such condition give rise to an arc unless it is absolutely clear that the arc in question is redundant. In this way, the algorithm never misses a condition that should give rise to an arc, but returns a result that may include some redundant arcs as well.

Feasible conditions are conditions that involve movements whose cleared paths are 'in conflict'. The exact definition of 'in conflict' is actually up to the user of this algorithm. I give one suggestion of a definition below, on which I base all my examples. But it is merely one of many possible definitions; the algorithm is applicable irrespective of the exact formulation.

Proposed definition Two movements are *in conflict* if their cleared paths coincide, or if their cleared paths share a part of the railway section. Two cleared paths share a part of the railway section if they have at least one track segment in common, or share a simple or joint waiting point (see Section 8.1.4).

8.3.1 Divide and Conquer

Divide and Conquer is a common concept that refers to a method of solving problems in general, and a method of designing algorithms in particular. Algorithms in this category split the instance of the problem to be solved into smaller sub-instances of the same problem, and solve them independently. The solutions to the sub-instances are then combined to form a solution for the original problem instance.

The reason for dividing the problem instance into smaller parts is of course that the smaller the problem is, the easier it is to solve. Recursion is a technique that exploits this idea, and the problem instances are continuously divided until a *base case* is reached. A base case is often the smallest possible problem instance, and is trivial to solve.

Part of my algorithm is based on the idea behind divide and conquer. A pure divide-and-conquer algorithm would work, but would treat too many obviously irrelevant conditions as if they were relevant, since it is often essential to consider larger contexts to discover that they are not relevant.

8.3.2 Subsections of a railway section

I will now define what I call *subsections of a railway section*, or simply *subsections*. They can in some sense be viewed as constituting the base cases of my algorithm.

Recall that every traffic pattern can be said to concern a well-defined set of track segments: the track segments of the traffic pattern (see Section 8.1.3). I want to be able to consider smaller sections on which the precedence relation, on the set of movements using the track segments of the section, is a total order that applies to every track segment of the section in question.

This goal could be satisfied by separately considering each track segment that is concerned by the traffic pattern. But there is often no need to consider every single track segment of the railway section separately. Preferably, the railway section is divided into as large sections as possible, for which the above still holds.

Definition A *subsection* is the *largest possible* part of a railway section on which every movement that uses the subsection occupies every track segment that is part of the subsection.

Practical consequences and notational issues From the definition follows that every movement that uses a specified subsection s is related to every other movement that uses the subsection with regard to precedence. The same total order applies to every track segment of s.

Every movement x_i of one cycle of the traffic pattern that uses the subsection, does so precisely once: if x_i uses the subsection, it means that (x_i, n) with $n = 1, 2, 3, \ldots$ all use the subsection exactly once.

The movements that use the subsection are from this point on referred to as *the subsection's movements*, movements *belonging* to the specified subsection, or the movement of a particular subsection. The track segments that make up the subsection of the railway are similarly referred to as *belonging* to the subsection, or being the track segments of the subsection.

A track segment in the set of track segments of the traffic pattern (see Section 8.1.3) belongs to precisely one subsection. A single movement can belong to many subsections, and in fact belongs to exactly those subsections that together contain exactly the track segments that make up the path of the movement.

The 'first' and 'last' track segments of a subsection are called *border segments*. Since waiting points are not parts of cleared paths, they will need to be considered separately. My algorithm presupposes the division of the railway section into subsections, and waiting points (or *waiting point-subsections*). All subsections and waiting points will be viewed both separately and as parts of a larger context.

Every waiting point borders on exactly two subsections. The end segment of a complete train movement marks the border of precisely one subsection. A track segment serving as a switch will generally – unless extraordinary circumstances prevail (see below) – form a border segment if more than one of its branches are used by the movements that use the subsection. If every movement uses the same branch of a switch, the fact that the track segment in question can serve as a switch can be ignored.

One of the 'extraordinary circumstances', mentioned above, refers to the situation illustrated in Figure 18. In the traffic pattern summarized by the figure showing one cycle of the pattern, neither one of the track segments serving as a switch is a border segment. The two cleared paths a and b are actually identical according to my definition[37], since they consist of the same track segments, in the same order. The railway section from `L` to `R` is in itself a single subsection with regard to this traffic pattern.

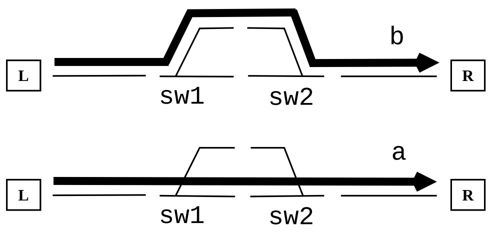

Figure 18: Two identical cleared paths a and b

A subsection of a traffic pattern will in the following be described by

- The track segments that it consists of.

[37]See page 56 for a discussion on why a and b in this example are considered to be the same cleared path.

- The set of movements M of one cycle of the traffic pattern, belonging to the subsection, and the direction (one of two possible) of each movement in M.
- The precedence relation – a total order – on M.
- Any neighbouring subsections, and what end of the subsection each one of them borders on. If the border segment serves as a switch, the exact branch that the neighbouring subsection borders on must be specified.

Most of the time, the descriptions will be graphical. Also, I will of course not call any part of the railway section a subsection, if the 'subsection' in question does not agree with the definition given above.

8.3.3 Examining subsections

The algorithm rests on few but very important principles. One of them is the division of the railway section into subsections. The other principles will be explained soon enough, but first I need to say a few words about how some steps of the algorithm are carried out. The exact steps of the algorithm are given in Section 8.3.8.

The algorithm basically considers every movement of the traffic pattern one at a time, in "pecking order", starting with one of the movements in the set of 'first' movements (see Section 8.1.5). For every movement that is scrutinized, the subsections that the movement in question belongs to, and all waiting points that it borders on, are *examined*, in chronological order.

The purpose of examining a subsection (possibly a so-called waiting point-subsection) s, with regard to a specific movement (x_i, n), is to get answers to the following questions:

- what movement (y_j, m) immediately precedes (x_i, n) on s?
- is the condition $c((y_j, m) < (x_i, n))$ relevant on s?

The first question is trivial to answer, since the movements are totally ordered on a subsection with regard to precedence. m in (y_j, m) is either n or $n-1$, depending on where the cycle boundary is. $y_j = x_i$ is a possible case, but only if $m = n - 1$.

Note that since immediate precedence on a subsection s is such an important concept, the fact that a movement y immediately precedes another movement x on s will from now on be denoted $y \ll_s x$.

The second question is trickier to answer, and I will not yet tell you exactly how this will be done. All that can be said at this point is that for every subsection, for every movement that uses it, precisely one condition is looked at. Only if it is deemed relevant, the condition gives rise to an arc in the condition graph that the algorithm outputs.

The algorithm is not perfect, and might in some cases deem a condition relevant on s even if its associated arc will be redundant in the output condition graph. The opposite, that a condition whose arc will not be redundant in the condition graph is missed, never occurs. Why this is the case, will be made clear later in the text (see Section 8.4).

Example In Section 8.3.9, I will illustrate how the algorithm is used by applying it to a traffic pattern that is used as the major example in Ekman and Kreuger's report [9]. I describe the traffic pattern here, together with a (mainly graphical) description of the subsections, to highlight the concepts that I have introduced so far.

Let W and E be two stations. Between them is a railway section with three other stations, mainly used to facilitate overtaking and train meets. The three complete train movements of the traffic pattern are w, e, and f. One cycle of the traffic pattern can be described by Figure 19. Movement w_1 (denoted `w1` in Figure 19), that is, the movement using the cleared path w_1, is considered to be 'first'. There are two simple waiting points: $w_1 : w_2$ and $e_1 : e_2$.

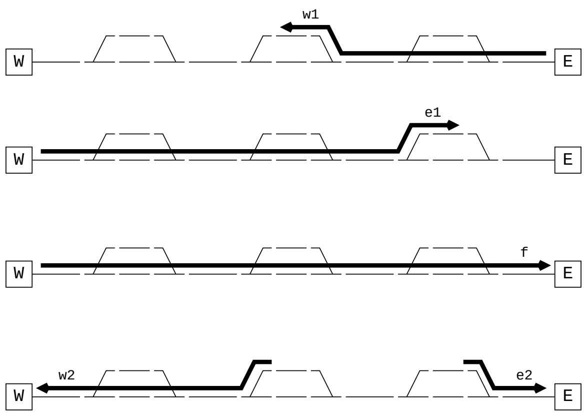

Figure 19: The traffic pattern of a comprehensive example

The subsections How the railway section is divided into subsections is indicated by Figure 20, where the seams now show where one subsection starts and ends. I have labled the subsections $s_1, s_2, \ldots, s_7$ so that I can refer to them more easily later on. In the figure, they are denoted `s1,s2,...,s7`.

Note that only necessary details of the subsections are shown. One of the stations thus 'disappears', that is, appears as part of a single line in Figure 20, since all trains in the traffic pattern occupy the same track segments of the station. Also, at this point there is no need to keep track of where individual track segments of the subsections start and end. That information is not needed until the arc weights of the condition graph are about to be determined.

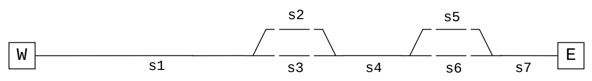

Figure 20: The subsections of the traffic pattern in Figure 19

The pecking order These are the movements belonging to the subsections 1-7, and their movements in pecking order.

1. $e_1 < f < w_2$
2. w_1
3. $e_1 < f$
4. $w_1 < e_1 < f$
5. e_1
6. $w_1 < f$
7. $w_1 < f < e_2$

Note that the pecking orders refer to the movements' mutual precedences within a single cycle, with the set of first movements as a reference point. The pecking order on for example s_1 should however be read as $(e_1, n) < (f, n) < (w_2, n) < (e_1, n+1) < \ldots$, since the pattern on any subsection is cyclic. The pecking orders on the other subsections should of course be interpreted in this way too.

The simple waiting points give rise to the following precedences: $(w_2, n) < (w_1, n+1)$, $(e_2, n) < (e_1, n+1)$, $(w_1, n) < (w_2, n)$ and $(e_1, n) < (e_2, n)$.

The traffic pattern graph These pecking orders together give the traffic pattern graph of Figure 21.

The procedure The algorithm will in this case start off by looking at movement w_1, since it is the 'first' movement of the traffic pattern. The first subsection that it will examine is s_7. It would, due to the pecking order displayed above, conclude that $(e_2, n-1)$ immediately precedes (w_1, n), denoted $(e_2, n-1) \ll_{s_7} (w_1, n)$, whereupon it would decide whether the condition $c((e_2, n-1) < (w_1, n))$ is relevant or not and take appropriate action. Next, the algorithm examines s_6, since s_6 is the subsection occupied next after s_7 by w_1, the movement which we are currently scrutinizing.

For a complete description of how the algorithm treats the traffic pattern of this example, see Section 8.3.9. Before I can fully describe the algorithm, I both need to tell how waiting points are viewed by the algorithm, and tell how conditions are deemed relevant.

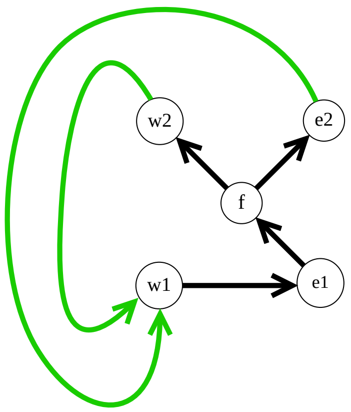

Figure 21: The traffic pattern graph of the traffic pattern in Figure 19

8.3.4 Waiting points

As far as the algorithm goes, waiting points work almost like subsections. Of course a train that waits at a waiting point has to move out of the way, that is, leave the waiting point, before another train can occupy it. In this regard, a waiting point and a subsection are equivalent. The only difficulty involved with waiting points at this stage, is the one resulting from the fact that a train is considered to occupy part of the cleared path *preceding* the waiting point during the actually waiting.

Simple waiting point A simple waiting point $x_i : x_{i+1}$, that is, a waiting point that does not serve as the waiting point of any complete train movement other than the one using path x (see page 62), gives rise to two conditions: one per movement involved in the waiting point.

When (x_{i+1}, n) is scrutinized, the waiting point $x_i : x_{i+1}$ gives rise to a relevant condition, $c((x_i, m) < (x_{i+1}, n))$, where (x_i, m) is the movement that immediately precedes (x_{i+1}, n). m is either $n - 1$ or n, depending on where the cycle boundary is. No movement other than x_i of cycle m can immediately precede (x_{i+1}, n), since no overtaking is possible at a waiting point. The associated condition $c((x_i, m) < (x_{i+1}, n))$ is always relevant, since the train that is about to physically undergo (x_{i+1}, n) must of course complete movement (x_i, m) before (x_{i+1}, n) can start!

The other condition is the one that is considered when the simple waiting point is examined during the scrutinizing of (x_i, n). The previous train x (of cycle m) must have left the waiting point $x_i : x_{i+1}$ before a new train x (of cycle n), can occupy it. This is summarized by the condition $c((x_{i+1}, m) < (x_i, n))$, which might be deemed as relevant depending on whether the condition is an immediate consequence of the relationships between other movements in the traffic pattern or not. How this is judged will be explained in Section 8.3.6.

Joint waiting point A joint waiting point has multiple names. Joint waiting points are not considerably more complicated than simple ones, since trains must leave waiting points in the same order as they arrive at them due to the fact that only one train at a time can occupy a waiting point.

The perspective from which a joint waiting point has to be viewed, depends on what movement is currently being scrutinized. Let x_i be the movement currently being scrutinized, and let x_i start at a waiting point that is shared between two different complete train movements using paths x and y. The resulting joint waiting point can be called either $x_{i-1} : x_i$ or $y_{j-1} : y_j$. To decide what conditions might be needed due to this joint waiting point, create a *waiting point-subsection* for movement x_i.

The waiting point-subsection for a movement that starts at a waiting point is always created by thinking of the movement in question, x_i, and all movements that belong to the subsection *preceding*[38] the waiting point, as being the movements of an ordinary subsection, which is then treated *in the same way as all the other subsections*. In this case, at least x_{i-1}, x_i and y_{j-1} will belong to the waiting point-subsection if x_i and y_j have the same direction. If they are oppositely directed, the movements of the waiting point subsection will instead be (at least) x_{i-1}, x_i and y_j. Read Section 8.1.4, *Notational issues*, again if this confuses you.

If a movement ends at a waiting point, the movements belonging to the waiting point-subsection are the movement being scrutinized and all movements belonging to the subsection *succeeding* the waiting point. The procedure above is the same regardless of the number of complete train movements that share the waiting point. It also works for simple waiting points.

8.3.5 Irrelevant conditions

In a traffic pattern, some relationships among movements are more easily analyzed than others. Precedences regarding oppositely directed movements belong to this category.

Consider the schematic sketch of a traffic pattern in Figure 22. The traffic pattern can be informally described as train a entering the railway section 'from the left' (at `L`), making a stop on the track segment between the track segment serving as switch 1 (`sw1`) and the track segment serving as switch 2 (`sw2`), at waiting point $a_1 : a_2$, then leaving the railway section at `R`. Train b is then allowed to enter at `R`, traversing the path from `R` to `L` without making any stops. The traffic pattern involves four subsections, described by Figure 23.

The information about the stops is actually not essential to make the following observation, since it applies with or without stops by trains a and b: b cannot enter the railway section at `R` before a has left it (at `R`). Moreover, the a-train of a cycle cannot enter the railway section at `L` until b belonging to the previous cycle has completed its movement and left the railway section at `L`.

[38]Preceding the waiting point with respect to the direction of the movement that is currently being scrutinized.

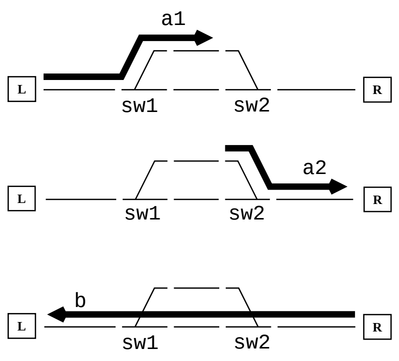

Figure 22: A simple traffic pattern involving oppositely directed movements

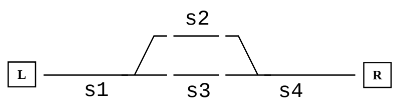

Figure 23: The subsections of the traffic pattern in Figure 22

The waiting point $a_1 : a_2$ is superfluous in this case, since train a does not wait for any train at this particular spot in this simple traffic pattern. But nothing prevents the user from introducing superfluous waiting points, and the algorithm must work even if such exist.

The key to identifying irrelevant conditions is the fact that once the condition $c((a_2, n) < (b, n))$ has been established in a case similar to the one above, we can be sure that a condition saying that movement (b, n) has lower precedence than any predecessor movement of (a_2, n) cannot be a relevant condition: $c((a_1, n) < (b, n))$ *automatically holds* in this case. The condition graph for the traffic pattern in question is shown in Figure 24.

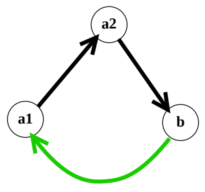

Figure 24: The condition graph of the traffic pattern in Figure 22

As can be seen in the condition graph (Figure 24), there is an arc from a_2 to b, but not from $a_1$to b, although $a_1 < b$ surely holds as well as $a_2 < b$. The fact that $a_1 < a_2$ and $a_2 < b$ together imply $a_1 < b$ is however not alone sufficient to deem the condition $c(a_1 < b)$ irrelevant.

The reason for $c(a_1 < b)$ being irrelevant in this case is that a and b have opposite directions. Since a_2 cannot start until a_1 has finished, and b cannot start until a_2 has finished[39], it is possible to exclude the possibility that a train undergoing movement a_1 occupies subsection s_1 at the time when the train undergoing movement b needs it. Thus the arc representing the condition $c(a_1 < b)$ is redundant in the presence of the other arcs given in the condition graph in Figure 24.

Applying the principle In the example above, movement (b, n) has to wait for (a_2, n) to complete before it can start. s_4 is the first subsection in b's path that b shares with a (in this case a_2). Since s_4 is the first subsection in b's path that b and a share, s_4 is also the only subsection that can give rise to a relevant condition involving a-movements[40] when b-movements (in this case only b itself) are scrutinized.

In the traffic pattern of the example above, the only relevant condition on the form $c((a_i, m) < (b_j, n))$ arises due to the first subsection that a and b share, s_4, and is of course $c((a_2, n) < (b, n))$.

In the algorithm, the movements are considered in pecking order. For every movement, the subsections it passes through are examined, in chronological order. Due to this, it is relatively simple to keep track of what subsection is the first that two oppositely directed movements share – first with respect to the movement that is currently being scrutinized. I will tell exactly how this can be done in Section 8.3.7.

8.3.6 Relevant conditions

As usual, the purpose of examining a subsection s during the scrutinizing of a movement (x_i, n) is to see whether the movement (y_j, m) that immediately precedes (x_i, n) on s gives rise to a relevant condition or not. Now it is time to say exactly what deems a condition relevant on s.

Consider a subsection s and two movements (x_i, n) and (y_j, m). Let $(y_j, m) \ll_s (x_i, n)$ hold. The interesting condition at this point is therefore $c((y_j, m) < (x_i, n))$. It is deemed *relevant* if and only if the following is true:

- x_i and y_j are oppositely directed, *and*
- s, with respect to x-movements, is the first subsection that x- and y-movements share

or

- x_i and y_j have the same direction, *and*

[39]All movements a_1, a_2 and b referred to here are of course in the same cycle, which is implicit since I have not denoted the movements on the form (x_i, n).

[40]Movements whose 'parent' is the complete train movement that uses the path a – in this case a_1 and a_2.

- s, with respect to (x_i, n), is the first subsection on which (y_j, m) is the movement that immediately precedes (x_i, n).

Observe that the last case is equivalent of saying that the condition $c((y_j, m) < (x_i, n))$ can only be relevant if it has not already been considered and deemed relevant at another subsection. In that case, the condition would already have played its part.

I assume that a movement (x_i, n) is preceded (and succeeded) by other movements of the subsection, that is, that n is never to be viewed as the first cycle. It is then always possible to identify the two movements (x_i, n) and (y_j, m) on any subsection s. For instance, a possible scenario is that, during one cycle of the traffic pattern, s is used by only one movement, x_i. In that case, (y_j, m) in the reasoning above is equal to $(x_i, n-1)$.

8.3.7 Log covered conditions

To be able to deem a condition relevant while scrutinizing a movement (x_i, n), the algorithm must keep track of what other movements have been encountered on the various subsections that have already been examined on (x_i, n)'s account. Without a log of some kind, it is for instance impossible to say if the current subsection is the first that an x-movement (a movement using the path x of a complete train movement) shares with a y-movement.

Let R denote a set that will be used as a log by the algorithm. It is empty when the algorithm starts off. The elements of R will eventually be so called covered conditions, that is, conditions that have played their parts.

The purpose of R is to enable the algorithm to just have to look in R when a subsection s is examined while scrutinizing x_i on which $y_j \ll_s x_i$ holds, and deem the condition $c(y_j < x_i)$ relevant if and only if $c(y_j < x_i) \notin R$.

When, and why is a condition $c(y_j < x_i)$ put in R? Generally, a condition is put in R when it is clear that it has played its part. A condition is for instance always put in R when it has been deemed relevant on a subsection s, after it has given rise to an arc. This prevents the same condition from giving rise to a duplicate arc.

Similarly, once a condition $c(y_j < x_i)$ *involving two oppositely directed movements* x_i and y_j has been deemed relevant, not only $c(y_j < x_i)$ is put in R. Due to the reasoning in Section 8.3.5, we know that *all conditions* on the following form are irrelevant from this point on:

- $c(y_{j-q} < x_{i+p})$, where $p = 0, 1, \ldots$, and $q = 0, 1, \ldots$

All such conditions are thus immediately put in R when $c(y_j < x_i)$ is deemed relevant and x_i and y_j are oppositely directed.

The last case that puts conditions in R occurs when $z_k < y_j < x_i$ holds on a subsection s for any movements z_k, y_j and x_i during the scrutinizing of x_i, while z_k and x_i are oppositely directed, and s is the first subsection that z- and x-movements share (with respect to the x-movements). All conditions on the form $c(z_{k-q} < x_{i+p})$, where $p = 0, 1, \ldots$, and $q = 0, 1, \ldots$ are immediately put in R, even though $c(z_k < x_i)$ is clearly not deemed relevant by the algorithm on s ($z_k \ll_s x_i$ does not hold). The reason is that no other subsection, during the scrutinizing of x_i can be the first subsection that z- and x-movements share, since s is the first. Thus no condition on the form just mentioned will ever be deemed relevant when x-movements are scrutinized.

8.3.8 The algorithm

The following algorithm will be thoroughly exemplified in the next section.

Input: a traffic pattern

Output: a condition graph (without arc weights), representing a sufficient set of conditions

The procedure The algorithm scrutinizes the movements of one cycle, cycle n, of the traffic pattern. Every movement will be completely scrutinized before the algorithm goes on to the next movement in line. The order in which the movements are scrutinized is any order that does not violate the partial order on the set of movements of cycle n (with regard to precedence).

Let R denote the set of covered conditions, A_s the set of straight arcs, and A_b the set of bowed arcs. $R = A_s = A_b = \emptyset$ when the algorithm starts.

Let (x_i, n) denote the movement currently being scrutinized. Examine the subsections belonging to (x_i, n), and waiting points bordering on (x_i, n), in order of appearance. Create a waiting point-subsection for every waiting point bordering on (x_i, n). Examining a subsection s belonging to (x_i, n), including waiting point-subsections, involves the following:

- Identify the movement (y_j, m) for which $(y_j, m) \ll_s (x_i, n)$ holds.
- For every movement (z_k, l) for which $(x_i, n-1) < (z_k, l) < (y_j, m) < (x_i, n)$ holds on s, and z_k and x_i are *oppositely directed*, check to see if $c((z_k, l) < (x_i, n)) \in R$. If not, put all conditions on the form $c((z_{k-q}, l) < (x_{i+p}, n))$ in R, where $p = 0, 1, \ldots$, and $q = 0, 1, \ldots$.
- If $c((y_j, m) < (x_i, n)) \notin R$, then
  - add the arc $a_{y_j x_i}$ to the appropriate arc set: A_s if $m = n$, and A_b if $m = n - 1$.
  - *If and only if x_i and y_j are oppositely directed*, put all conditions on the form $c((y_{j-q}, m) < (x_{i+p}, n))$, where $p = 0, 1, \ldots$, and $q = 0, 1, \ldots$, in R.
  - *If x_i and y_j have the same direction*, put $c((y_j, m) < (x_i, n))$ in R.
- Proceed to the next subsection (or waiting point-subsection) in the path of movement (x_i, n).

When all subsections (including waiting point-subsections) belonging to (x_i, n) have been examined, the algorithm is done with (x_i, n), and the next movement according to the pecking order is scrutinized.

When all movements of cycle n have been scrutinized, the algorithm outputs the condition graph consisting of a node set V with one node per movement that was scrutinized, and the arc sets A_s and A_b.

8.3.9 An example

I will now apply the algorithm to the traffic pattern described in Section 8.3.3. All five movements will be scrutinized in the overall pecking order, starting with w_1 since it was arbitrarily chosen to be 'first' in the description of the traffic pattern. w_2 and e_2 can be scrutinized in any order, so I simply decide that I will scrutinize w_2 before e_2.

Below is an account of what arcs the algorithm suggests for every scrutinized movement, and the reasoning that leads to the suggestions. In the following, A_s is the set of straight arcs of the output, and A_b is the set of bowed arcs. If a subsection s_k "gives $a_{x_i y_j} \in A_s$", it means that the straight arc $a_{x_i y_j}$ is added to the output. I will not show the reasoning as detailed for all movements. After a while, the procedure will be familiar.

Initializations R, A_s, and A_b are all empty sets when the algorithm starts.

(w_1, n) Subsection s_7 is examined first. $(e_2, n-1)$ immediately precedes (w_1, n) on s_7, that is, $(e_2, n-1) \ll_{s_7} (w_1, n)$ holds. Since $R = \emptyset$, the bowed arc $a_{e_2 w_1}$ is suggested: s_7 gives $a_{e_2 w_1} \in A_b$. Moreover, all movements of s_7 that occur between $(w_1, n-1)$ and (w_1, n) that are oppositely directed w_1, receive special treatment. The movements in question are $(f, n-1)$ and $(e_2, n-1)$. The result of the 'special treatment' is that the following conditions are put in R:

- $c((f, n-1) < (w_1, n))$
- $c((f, n-1) < (w_2, n))$
- $c((e_2, n-1) < (w_1, n))$
- $c((e_2, n-1) < (w_2, n))$
- $c((e_1, n-1) < (w_1, n))$
- $c((e_1, n-1) < (w_2, n))$

Of these, $c((e_2, n-1) < (w_1, n))$ was deemed relevant on s_7 and gave rise to an arc due to s_7. But the circumstances now tell us that the conditions above should not give rise to arcs *during the rest of the procedure.* Putting them in R thus correctly prevents them from being deemed relevant in the future.

Next, s_6 is examined. $(f, n-1) \ll_{s_6} (w_1, n)$ holds, but $c((f, n-1) < (w_1, n)) \in R$. The condition is thus not deemed relevant, and no new arc is added to the output. There is no movement on s_6 occurring between $(w_1, n-1)$ and (w_1, n) which is oppositely directed w_1, other than $(f, n-1)$. Since $c((f, n-1) < (w_1, n))$ is already in R, no further action is taken due to s_6.

s_4 is next in line. The pecking order of this subsection, $w_1 < e_1 < f$, introduces a new movement: $(e_1, n-1)$. $(f, n-1) \ll_{s_4} (w_1, n)$ holds on s_4 just like $(f, n-1) \ll_{s_6} (w_1, n)$ did on s_6, but is ignored once again since the associated condition is still in R. $c((e_1, n-1) < (w_i, n)) \in R$ also holds, so no new conditions need to be put in R.

s_2 gives $a_{w_1 w_1} \in A_b$, and $c((w_1, n-1) < (w_1, n))$ is put in R.

Finally, waiting point $w_1 : w_2$ needs to be examined. A waiting point-subsection, consisting of movements (w_1, n), $(w_2, n-1)$, $(e_1, n-1)$ and $(f, n-1)$, is created: the movement being scrutinized, (w_1, n), and all movements from cycle n and $n-1$ that precede (w_1, n) and belong to the subsection *succeeding* the waiting point: $(w_2, n-1)$, $(e_1, n-1)$, and $(f, n-1)$. What movement, in this waiting point-subsection, immediately precedes (w_1, n)? The answer is $(w_2, n-1)$. Since $c((w_2, n-1) < (w_1, n)) \notin R$, $a_{w_2 w_1} \in A_b$ is suggested. $(e_1, n-1)$ and $(f, n-1)$ are both oppositely directed (w_1, n), but the conditions associated with them are already in R, so we do nothing more at this stage.

This concludes the scrutinizing of (w_1, n).

(e_1, n) First, s_1 is examined. $(w_2, n-1) \ll_{s_1} (e_1, n)$ holds. $c((w_2, n-1) < (e_1, n)) \notin R$, so the bowed arc $a_{w_2 e_1}$ is added to A_b. Since e_1 and w_2 are oppositely directed and $c((w_2, n-1) < (e_1, n)) \notin R$, the conditions $c((w_2, n-1) < (e_1, n))$, $c((w_2, n-1) < (e_2, n))$, $c((w_1, n-1) < (e_1, n))$, and $c((w_1, n-1) < (e_2, n))$ are all put in R. No other oppositely directed movements that occur between $(e_1, n-1)$ and (e_1, n) belong to s_1.

On s_3, $(f, n-1) \ll_{s_3} (e_1, n)$. $c((f, n-1) < (e_1, n)) \notin R$, which gives $a_{f e_1} \in A_b$. $c((f, n-1) < (e_1, n))$ is then put in R.

s_4 gives $a_{w_1 e_1} \in A_s$ since $c((w_1, n-1) < (e_1, n)) \notin R$. $c((w_1, n-1) < (e_1, n))$ is then put in R, along with $c((w_1, n-1) < (e_2, n))$.

s_5 gives $a_{e_1 e_1} \in A_b$, and $c((e_1, n-1) < (e_1, n))$ is put in R.

Now only the waiting point $e_1 : e_2$ remains before the algorithm is through with (e_1, n). The waiting point-subsection this time consists of movements $(f, n-1)$, $(e_2, n-1)$, (w_1, n) and (e_1, n). $(w_1, n) \ll_{e_1 : e_2} (e_1, n)$ holds. The associated condition is in R, so no new arc is suggested. No more conditions need to be put in R.

(f, n) s_1 gives $a_{e_1 f} \in A_s$. Put in R: $c((e_1, n) < (f, n))$, $c((w_2, n-1) < (f, n))$ and $c((w_1, n-1) < (f, n))$.

s_3 gives nothing new.

On s_4, $(e_1, n) \ll_{s_4} (f, n)$ holds, just like $(e_1, n) \ll_{s_1} (f, n)$ did on s_1. This time, $c((e_1, n) < (f, n))$ is in R, so the condition is not relevant on s_4. $(w_1, n) < (f, n)$ holds on s_4, (w_1, n) and (f, n) are oppositely directed and $c((w_1, n) < (f, n)) \notin R$. $c((w_1, n) < (f, n))$ is therefore put in R[41], but (w_1, n) have no

[41]Previously, $c((w_1, n-1) < (f, n))$ has been put in R, but this does not prevent $c((w_1, n) <$

predecessors – and (f, n) no successors – so no more conditions need to be put in R right now.

s_6 and s_7 do not give any new arcs although $(w_1, n) \ll_{s_6} (f, n)$ and $(w_1, n) \ll_{s_7} (f, n)$ hold, since $c((w_1, n) < (f, n)) \in R$ due to s_4.

(f, n) is not involved in any waiting point, so the algorithm is done with it.

(w_2, n) This movement starts at a waiting point. The waiting point-subsection of $w_1 : w_2$ consists of (w_2, n) and (w_1, n), which of course gives $a_{w_1 w_2} \in A_s$. Remember that any movement x_i that starts at a waiting point gives rise to the arc $a_{x_{i-1} x_i}$ – straight or bowed depending on the cycle boundary – since the movement x_i cannot possibly start before movement x_{i-1} has been completed.

The condition that gave rise to the arc is put in R. (w_2, n) only belongs to one subsection, s_1. It gives rise to $a_{f w_2} \in A_s$. Put in R: $c((f, n) < (w_2, n))$ and $c((e_1, n) < (w_2, n))$.

(e_2, n) Last but not least, (e_2, n) is scrutinized. $e_1 : e_2$ gives $a_{e_1 e_2} \in A_s$ (see the discussion about $(w_1 : w_2)$ above). s_7 gives $a_{f e_2} \in A_s$.

The output The following precedences of the traffic pattern give rise to straight arcs:

- $w_1 < e_1$
- $e_1 < f$
- $w_1 < w_2$
- $f < w_2$
- $e_1 < e_2$, and
- $f < e_2$

The following precedences of the traffic pattern give rise to bowed arcs:

- $(e_2, n-1) < (w_1, n)$
- $(w_1, n-1) < (w_1, n)$
- $(w_2, n-1) < (w_1, n)$
- $(w_2, n-1) < (e_1, n)$
- $(e_1, n-1) < (e_1, n)$, and
- $(f, n-1) < (e_1, n)$

This means that the condition graph that the algorithm suggests is the one in Figure 25.

$(f, n))$ from being put there. Had $(w_1, n) \ll_{s_4} (f, n)$ held, the condition in question would of course have given rise to an arc. Do not confuse $(w_1, n-1)$ with (w_1, n)!

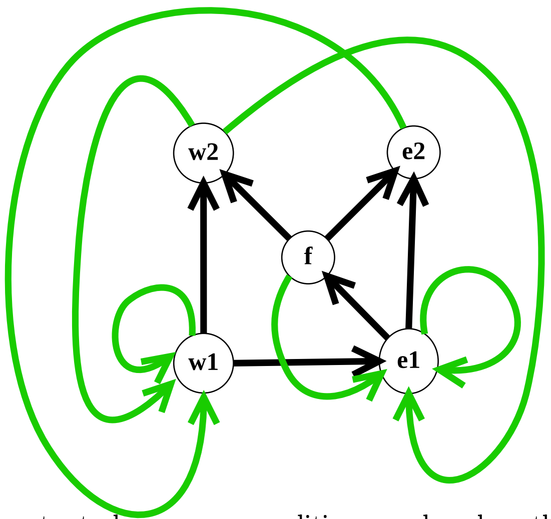

Figure 25: The output shown as a condition graph, when the algorithm of Section 8.3.8 is run on the traffic pattern of Section 8.3.3

8.4 Eliminating redundant arcs

A set of infinitely many conditions – one per every possible pair of movements defined by the precedence relation of the traffic pattern, of which infinitely many exist – is reduced to a manageable set of sufficient conditions by the algorithm of Section 8.3.8. Due to special circumstances, the arc sets of straight and bowed arcs output from the algorithm possibly contain redundant arcs.

I will not provide any exhaustive algorithms for how to eliminate every redundant arc from the output, since I cannot see how this can be done. Instead, I will reason about possible ways of finding them, and give a couple of examples. Before I do this, I need to define a few terms.

I claim that *a set of sufficient arcs* with regard to the traffic pattern used as input, is output by the algorithm that I give in Section 8.3.8, since the algorithm only dismisses conditions that would give rise to *obviously redundant* arcs. A condition belonging to the set of sufficient conditions represented by an arc in the condition graph output by the algorithm is simply a condition that *at one point in the algorithm was deemed relevant.*

I have not yet properly defined what must apply to a condition that belongs to the set of sufficent conditions.

Definition With respect to a specific traffic pattern, defined according to the rules in Section 8.1.7, *a set of sufficient conditions* S is a set of conditions for which the following statements all hold

- if $c \in S$, c is a condition that represents a relationship between two movements of the traffic pattern that is in accordance with the precedence relation

- if $c \notin S$ and c is a condition that represents a relationship between two movements of the traffic pattern that is in accordance with the precedence relation, then c is an immediate consequence of the conditions in S
- if c is not a condition that represents a relationship between two movements of the traffic pattern that is in accordance with the precedence relation, then $c \notin S$

Note that nothing in this definition says that the set of sufficient conditions has to be the smallest possible set for which the three requirements of the definition hold. Potentially, all precedences (infinitely many!) can be represented by conditions in S. This leads me to define redundancy in the context of conditions and sufficient conditions.

Definition If c is an immediate consequence of the other conditions in S, then the arc representing such a condition in the condition graph is *redundant.*

The cycle time of the traffic pattern can be computed from the set of sufficient conditions, as I have defined the term above. This follows from what Ekman and Kreuger calls the main statement of their report: the cycle time of a traffic pattern is the maximum cycle mean of the condition graph of the traffic pattern [9]. They provide a strict proof of their main statement. For my purposes, it is sufficient to say that there is an available proof of this, and I refer the interested reader to their report. A set of sufficient conditions enables the computation of the cycle time of the underlying traffic pattern.

Ekman and Kreuger also point out that it is difficult to identify every redundant arc, that is, all conditions that are immediate consequences of a smaller set of sufficient conditions, but that the smaller the set of sufficient conditions is, the better. Every condition, on which the computation of the cycle time will be based, requires data from the user in terms of at least the time it takes for the trains to travel certain distances. Redundancy should be avoided if possible.

8.4.1 Dismissing arcs

In the following, the arcs I refer to are the arcs that are output by the algorithm given in Section 8.3.8, unless specified otherwise. Likewise, 'the railway section' of course refers to the railway section of the traffic pattern, used as input to the algorithm when the arcs were produced.

One way of eliminating redundant arcs is by considering larger portions of the railway section than the subsections. Sometimes interesting relationships between the movements can be found from this 'bird perspective' – relationships that render the arcs representing certain conditions redundant. In other words, an approach that focuses on subsections can unfortunately deem conditions, whose arcs in the condition graph are redundant, relevant.

An example of this is the bowed arc $a_{e_1 e_1}$ in the example of Section 8.3.9. It is quite easy to convince oneself that this arc is redundant. The reason is basically the same that renders the bowed arc $a_{e_2 e_1}$ redundant. The movement (w_1, n) needs to wait for $(e_2, n-1)$ to finish before it can start. $(e_2, n-1)$ in turn needs to wait for $(e_1, n-1)$ to finish before it can start. (e_1, n), however, cannot be allowed to occupy the part of the track that I call subsection 4, s_4, before $(w_1, n-1)$ has occupied it.

The effect is that it is impossible to have a train still undergoing movement $(e_1, n-1)$ in s_5 when the next train e during (e_1, n) needs access to s_5. s_5 is the subsection that is occupied by e_1 alone, and therefore gave rise to the arc in question. From this perspective, we can see that $a_{e_1 e_1} \in A_b$ is redundant, although the algorithm missed to identify the associated condition as irrelevant.

Why did the algorithm miss this one? Because on s_5, $(e_1, n-1) \ll_{s_5} (e_1, n)$ certainly holds, since e_1 is the only movement ever using s_5, and because $c((e_1, n-1) < (e_1, n))$ is not in R at the time when (e_1, n) is scrutinized by the algorithm and s_5 is examined on behalf of (e_1, n).

Additional rules How do we discover redundant arcs of this kind? Are there many other situations that are as obvious as this one? I cannot give a definitive answer to the last one of these questions. There could be, but I have in that case not been able to identify all of them. To the first question, I have a suggestion.

Let the path x of a complete train movement of a particular traffic pattern include at least three subsections: s_h, s_t, and s_u. A train using path x occupies these three subsections in the order specified above. Let y be the path of another complete train movement of the same traffic pattern, and let y include the subsections s_h and s_u, but not s_t. A train using path y occupies subsection s_u before it occupies s_h. This means that x- and y-movements on s_h and s_u are oppositely directed.

Now, suppose that $(y_j, m) < (x_i, n)$ holds on s_h, where $m = n-1$ or $m = n$. Suppose also that $(y_{j-q}, m) < (x_{i+p}, n)$ holds on s_u, where $p = 0, 1, \ldots$ and $q = 0, 1, \ldots$. Then we can conclude that a train undergoing the complete train movement on path x cannot possible occupy subsection s_t at a point in time when a train undergoing the same complete train movement *of the next cycle* needs access to it. The reason is that $(x_{i+p}, n-1)$ must be completed on s_u before (y_{j-q}, m) can start on s_u. And before (x_i, n) is allowed to start on s_h, (y_j, m) has to finish on s_h.

In the reasoning above, s_t represents an arbitrary subsection that is part of the path of x but not part of y. All conditions that arise on subsections like s_t, physically located between s_h and s_u, implying that any x-movement governs when another x- movement can start, can be disregarded. The associated conditions, $c((x_{i+p}, n-1) < (x_{i+p}, n))$ and $c((x_{i+p+1}, m) < (x_{i+p}, n))$ (where $m = n-1$ or $m = n$), need to give rise to arcs only if the conditions are deemed relevant on subsections that are not physically located between s_h and s_u, and for which the above situation does not apply.

Refining the algorithm If this rule were part of the algorithm, it would be necessary to incorporate a way of looking back on what actions any specific subsection has previously demanded. Then the wording could be the following.

Suppose the following is true when (x_i, n) is scrutinized:

- $c((y_j, m) < (x_i, n))$ is placed in R due to a subsection s_h and for any reason, where $m = n - 1$ or $m = n$.
- (y_j, m) and (x_i, n) are oppositely directed.
- In a later subsection s_u (with respect to (x_i, n)), $(y_{j-q}, m) < (x_{i+p}, n)$ holds, where $p = 0, 1, \ldots$ and $q = 0, 1, \ldots$.

Then no immediate precedence, ***considered due to any subsection s_t physically located between s_h and s_u*** (with regard to the x-movements), and on the following form (see below), could have given rise to a needed arc. The associated arcs, generated due to any such subsections, should be dismissed.

- $(x_{i+p}, n-1) \ll_{s_t} (x_{i+p}, n)$, where $p = 0, 1, \ldots$
- $(x_{i+p}, m) \ll_{s_t} (x_{i+p-1}, n)$, where $p = 0, 1, \ldots$ and $m = n$ or $m = n - 1$

Note that it is possible that other subsections call for the arcs in question. Such arcs must not be dismissed. Only if the above holds for ***all subsections*** on which the arc in question is given rise to, the arc is considered redundant.

The situation above of course applies to w_1, e_1 and e_2 in the example of Section 8.3.9. If $w_1 = x_i$, $e_2 = y_j$, $s_h = s_7$, and $s_u = s_4$, the rule above says that arcs based on immediate precedences on a certain form – of which $(e_1, n-1) \ll_{s_5} (e_1, n)$ is one example – given rise to due to subsections physically located between s_7 and s_4 are redundant, although such arcs can be suggested by the algorithm. The bowed arc $a_{e_1 e_1}$ from s_5 is in other words redundant.

There is also another, similar case to the one above, where conditions that are deemed relevant by the algorithm afterwards can be identified as irrelevant, and their associated arcs as redundant.

Suppose the following is true:

- a traffic pattern concerns, among others, three complete train movements using paths x, y and z. The movements on x and y are oppositely directed on the subsections that they share, while those on x and z have the same direction.
- $(x_i, n) < (y_j, m) < (z_k, l)$ holds on a subsection s_h, and $(x_{i+p}, n) < (y_{j-q}, m) < (z_{k+r}, l)$ holds on s_u, where $p = 0, 1, \ldots$, $q = 0, 1, \ldots$ and $r = 0, 1, \ldots$.

Then the following conclusion can be drawn: whenever a condition considered due to a subsection s_t physically located between s_h and s_u is deemed relevant, it will give rise to a redundant arc if it is on the following form:

- $c((x_{i+p}, n) < (z_{k+r}, l))$, where $p = 0, 1, \dots$ and $r = 0, 1, \dots$.

The reason is that y-movements (possible only one y-movement) of cycle m must precede z-movements of cycle l, and succeed x-movements of cycle n, *on both s_h and s_u*. On a subsection s_t located between s_h and s_u, $(x_{i+p}, n) \ll_{s_t} (z_{k+r}, l)$ might hold, with $p = 0, 1, \dots$ and $r = 0, 1, \dots$.. But we know that any arc, given rise to on s_t and going from a node representing an x-movement to a node representing a z-movement (straight or bowed does not matter), would be redundant, since an x-train cannot possibly linger in s_t when a z-train needs access to it.

An example on which this case applies is displayed in Figure 26. The four subsections are shown in Figure 27.

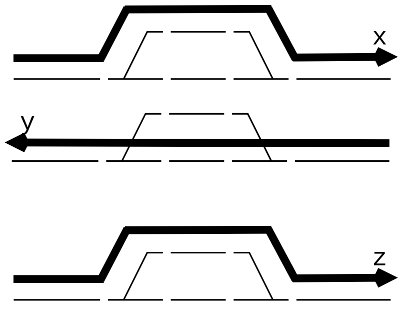

Figure 26: A traffic pattern that renders redundant arcs when the algorithm of Section 8.3.8 is applied

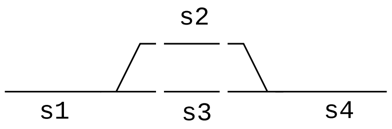

Figure 27: The subsections of the traffic pattern in Figure 26

Subsection s_2, on which $(x, n) \ll_{s_2} (z, n)$ holds, gives rise to an arc in the algorithm. This is a situation for which the case just explained holds. Since (y, n) has to precede (z, n) and succeed (x, n) on both s_4 and s_1, the additional rule says that any condition associated with $(x, n) \ll_{s_t} (z, n)$ when $t = 2$ or $t = 3$ is irrelevant in this case. The train undergoing movement (x, n) has to leave s_4 before (y, n) can start, and the train undergoing movement (y, n) has to leave s_1 before (z, n) can start. Thus, the start time of (z, n) cannot directly be governed by (x, n).

It is, as I said in the beginning of this section, possible that there are more rules of this kind that can be applied to the arc set that the algorithm outputs.

I have not been able to prove that these rules eliminate all redundant arcs. I will have to leave the quest for a perfect algorithm or set of rules to a future project.

8.4.2 Verifying arcs

A similar, but still very different approach that can be used concerns the verification of conditions whose arcs are not redundant. The problem with this approach is that it is risky to use it to actually eliminate arcs. The principle is that a 'suspect' arc can be examined by trying to prove that it is *not* redundant. If this fails, the arc is probably redundant. But there is of course a possibility that you only failed to prove something that is indeed provable, so you cannot be completely sure that such an arc can be eliminated.

This principle should only be used when it might be easier to prove that an arc is needed than it is to see that it is redundant, and the user desperately wants the arc to be redundant – for instance because the arc weight of the arc is very hard to calculate or approximate. *If* the user can be convinced that the arc is definitely needed, he or she can at least stop trying to prove that it is redundant and focus on gathering the needed data for its weight.

Conditions that generate loops seem hard for the algorithm of Section 8.3.8 to analyze. The necessity of these arcs is fairly easy to verify (for arcs that are indeed not redundant). A relevant loop-condition $c((x_i, n+1) < (x_i, n))$ gives rise to a bowed arc that starts and ends at the same node, implying that a train about to start its movement x_i of cycle $n+1$ might need to wait for the train undergoing movement x_i of cycle n to move out of its way.

I assume that the following is true, without attempting to prove it. If the cycle time of a traffic pattern is computed without taking a specific loop-condition on a movement x_i into consideration, and the cycle time can be shown to be independent of, say, the durations of a possible stop by the train undergoing movement x_i, the missing loop-condition $c((x_i, n+1) < (x_i, n))$ is obviously necessary for the computation of capacity. Otherwise, it would be possible to let the duration of a stop by x_i exceed the cycle time, which is of course absurd.

Generally, if a certain condition is excluded from the set of sufficient conditions as output from the algorithm, and this makes the cycle time independent of the duration of any possible stop by any train in the traffic pattern, the arc representing the condition is not redundant. This is unfortunately not often as easily shown as with the loop-condition in the example above, but is nevertheless true.

9 Analysis and conclusions

The method that Ekman and Kreuger have suggested can be used to assess capacity on a railway network. It is analytical and provides exact answers with respect to its input; there are no indeterministic aspects involved. Full-scale, practical use of the method will not be utilized until there exists some kind of computerized tool based on the method.

I have implemented the part of the method that performs the final computation of the cycle time of the traffic pattern, that is, the inverse of the capacity. I have deliberately made the code easy to modify and optimize if this is desirable before it is eventually incorporated in a future tool for capacity assessment; the documentation of the code as given in this report is thorough.

The problem of translating an informal definition of a traffic pattern into a condition graph suitable as input to my implementation showed to be harder than I first thought. As I have mentioned, the algorithm given in Section 8.3.8 outputs arcs that represent a set of sufficient conditions that enables the computation of capacity. This means that if all arcs are given their appropriate weights, the result is a condition graph that can successfully be used as input to the maximum mean cycle algorithm, DG1, given in Section 7.2.

Depending on what data of traversal times of trains and similar information that is available, it might be valuable to prune the arc sets (bowed and straight arcs) output by the algorithm further. Some redundant arcs can luckily be eliminated quite easily, and this is illustrated in Section 8.4.1.

I have not been able to find a grammar that incorporates every aspect of finding the smallest possible set of sufficient conditions. This was not my assignment, but I admit that it was what I aimed at. I cannot be sure of how many different kinds of redundant arcs my algorithm and the extra rules given in Section 8.4.1 miss. With ad hoc reasoning, any traffic pattern of reasonable size can be carefully examined and verified by hand. I have carried out numerous such verifications, and they have all shown that my algorithm and extra rules seem to eliminate at least almost all redundant arcs.

The major result accounted for in this report is without doubt the approach of finding a suitable input for the algorithm that performs the cycle time computation. It is not perfect, but definitely seems good enough. With it, Ekman and Kreuger's method should be of great use if a tool that uses it is appropriately designed.

References

[1] Erik Aurell and Jan Ekman. Kapacitet hos enskilda bangårdar. Technical Report L4i-99/341, Industrilogik, 1999. FoU-rapport, Banverket.

[2] F.L. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems.* Wiley: Chicester, 1992.

[3] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications.* Springer-Verlag, London, 2000. `http://www.imada.sdu.dk/Research/Digraphs/`.

[4] Norman L. Biggs. *Discrete Mathematics.* Oxford University Press, New York, revised edition, 1989.

[5] Peter Jephson Cameron. *Combinatorics: topics, techniques, algorithms.* Cambridge University Press, Cambridge, U.K., 1994.

[6] Anne Churchod and Luigi Lucchini. Capres, general description of the model. Online publication, June 2001. Swiss Federal Railways, document LITEP 788/5_e.

[7] Ali Dasdan and Rajesh K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[8] Robert-Jan van Egmond. *Railway Capacity Assessment, an Algebraic Approach.* Number S99/2 in TRAIL Studies in Transportation Science. Delft University Press, Delft, The Netherlands, July 1999.

[9] Jan Ekman and Per Kreuger. En analytisk metod för utredning av kapacitet vid signalprojektering. Technical report, SICS AB, Kista, 2003. FoU-rapport, Banverket.

[10] R.M.P Goverde and G. Soto Y. Koelemeijer. *Performance Evaluation of Periodic Timetables: Theory and Algorithms.* Number S2000/2 in TRAIL Studies in Transportation Science. Delft University Press, Delft, The Netherlands, September 2000.

[11] A.F. de Kort, B. Heidergott, R.J. van Egmond, and G. Hooghiemstra. *Train Movement Analysis at Railway Stations: Procedures & Evaluation of Wakob's Approach.* Number S99/1 in TRAIL Studies in Transportation Science. Delft University Press, Delft, The Netherlands, February 1999.

[12] Harald Krueger. Parametric modeling in rail capacity planning. In P.A. Farrington, H.B. Nembhard, D.T. Sturrock, and G.W. Evans, editors, *Proceedings of the 1999 Winter Simulation Conference*, pages 1194–2000, 1999. `http://www.informs-cs.org/wsc99papers/173.PDF`.

[13] Robert H. Leilich. Application of simulation models in capacity constrained rail corridors. In D.J. Medeiros, E.F. Watson, and M.S. Carson, J.S.and Manivannan, editors, *Proceedings of the 1998 Winter Simulation Conference*, pages 1125–1133, 1998. http://www.informs-cs.org/wsc98papers/153.PDF.

[14] Dick Middelkoop and Michiel Bouwman. Simone: Large scale train network simulation. In B.A. Peters, J.S. Smith, D.J. Medeiros, and M.W. Rohrer, editors, *Proceedings of the 2001 Winter Simulation Conference*, pages 1042–1047, 2001. http://www.informs-cs.org/wsc01papers/140.PDF.

[15] Thomas Sjöland. Course notes for logic programming, 2g1121. Laboratory of Electronic and Computer Science, Kista, August 2001.

[16] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press series in logic programming. The MIT Press, Cambridge, Massachusetts, second edition, 1986. Fourth printing.

[17] Stratec. Railcap, a computer tool for studying capacity problems of railway networks. Online publication. http://www.stratec.be/PlanGBrailcap.htm.

[18] Sicstus prolog – association lists. Website. http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_16.html#SEC190.

[19] The world of railway signalling. Website. http://www.railwaysignals.com/.

[20] Peran van Reeven, Jan-Jaap de Vlieger, Robert Offermans, Henning Tegner, Peter Danzer, Dimitrios Tsamboulas, Yves Putallaz, José Viegas, Rosário Macário, Fernando Crespo, Marques Carlos, Susana Neves, Dominique Bouf, Pierre-Yves Peguy, Kjell Werner Johansen, and Bjørnar Andreas Kvinge. State of the art in railway infrastructure capacity and access management. Deliverable 1 of IMPROVEd tools for RAILway capacity and access management, April 2002. http://www.tis.pt/proj/improverail/Downloads/D1Final.pdf.

[21] José Viegas, Rosário Macário, Fernando Crespo Dui, Peran van Reeven, Robin Hirsch, Will Adeney, Richard Anderson, Paola Cossu, Dimitrios Tsamboulas, Antoaneta Ormandjieva, Robert Rivier, Yves Putallaz, Henning Tegner, and Kjell Werner Johansen. Inception report. Deliverable 0 of IMPROVEd tools for RAILway capacity and access management, October 2001. http://www.tis.pt/proj/improverail/Downloads/D0Final.pdf.