Automatic generation and analysis of routing logic

Tunderman, T.M.

Award date:
2014

AUTOMATIC GENERATION AND ANALYSIS OF ROUTING LOGIC

T.M. TUNDERMAN

Department of Mathematics and Computer Science
Architecture of Information Systems

Eindhoven University of Technology

5 September 2014

Graduation supervisor:
dr. N. Sidorova
Eindhoven University of Technology

Graduation tutor:
ir. J. Peeters
Vanderlande Industries B.V.

ABSTRACT

In this thesis we present the problem and solution for automatically generating routing logic for a baggage handling system, used by Vanderlande. Routing logic for a baggage handling system is captured through routing rules. Routing rules indicate at decision points what the preferred course of action is to get baggage to its (intermediate) destination. We want to generate a valid (i.e. complete) and good (i.e. high performance) set of routing rules, which are currently created manually. For a large baggage handling system, creating a complete set of routing rules can take up to two months of man-hours, which is why Vanderlande seeks a way to reduce this.

We present a framework capable of doing so, consisting of three processes: the creation process, the analysis process and the optimization process. Using graph theory as a basis, the creation algorithm applies a modified shortest path algorithm in order to generate an initial set of routing rules. The creation process only regards singular bags, and disregards the effect of multiple bags in the system. The analysis process is used for analysing an arbitrary set of routing rules, in order to indicate the performance of that set. Through the use of flow algorithms, we simulate thousands of bags in the system, being routed using the given set of routing rules. This analysis process is capable of indicating possible bottlenecks in the system due to inefficient routing. The optimization process uses these bottlenecks as input and applies optimization techniques in order to try to resolve or reduce the bottlenecks.

Using two datasets of actual baggage handling systems used by Vanderlande, we analysed the performance of the routing rules generated by our framework. The initial sets of routing rules generated by the creation algorithm already contain 50% to 75% of all manually created routing rules. The performance of these initial sets are around two-thirds worse than the performance of the manually created sets. After analysis and applying optimizations, the performance of these initial sets can be further increased by 15% to 20%.

Further improvements to the framework are still possible by introducing knowledge about the history of baggage, used in the analysis process. This increases the amount of information extracted by the analysis algorithm, and makes it possible to resolve bottlenecks that are the result of multiple inefficient routing decisions spread out across the system.

ACKNOWLEDGMENTS

This thesis is the culmination of six months of blood, sweat and tears spent at Vanderlande in Veghel, the Netherlands. When I started this graduation project, I could not foresee that I would only become more and more interested in the problem given to me. Therefore is consider myself lucky to have stumbled upon a project that in the end still motivates me, and pushes me for better results.

Naturally, lots of people contributed to this project, and without them the results presented in this thesis would look entirely different. I would like to dedicate this relatively tiny space to thank those who helped me.

First of all, I would like to thank my supervisors dr. Natalia Sidorova and ir. Jos Peeters. It was through Natalia Sidorova I acquired this project, by sheer coincidence during the seminar course she taught. Without her, this thesis would have been about a completely different subject. Of course, I would like to thank her for all the feedback sessions and sparking some great ideas on how to tackle some of the problems I encountered. Jos Peeters was the one who kept pushing me to look critically at my own solutions, trying to come up with better ideas, and I would like to thank him for that. Even through his busy schedules, he was always prepared to give me feedback, even during lunch breaks if that was necessary. Also, thanks to them both for proofreading my thesis, and giving me tips on how to rewrite certain sections.

I would like to thank dr. ir. Joaquin Vanschoren and dr. Pieter van Gorp for assuming position on my assessment committee, and Vanderlande for making it possible to do this project. Furthermore, I would like to thank my mother for her continuing support, and my father for proofreading parts of my thesis and giving me advice on how to improve it. Many thanks to the rest of my family and good friends for their interest and support through the years.

Last but not least, I would very much like to thank my girlfriend Leanne for her endless patience with me. This, through all of my busy weekends and even when it resulted in our single opportunity of a vacation being thrown out of the window. Without her support, I could not have done this half as good as I wanted to.

Tristan Tunderman
Eindhoven, 05 September 2014

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

BHS	Baggage Handling System
KPI	Key Performance Indicator
LM	Logistics Manager
NSD	Node Segment Diagram
PC	Process Controller
PLC	Programmable Logic Controller
XML	Extensible Markup Lanuage

INTRODUCTION

Vanderlande specializes (among other things) in baggage handling systems, which nowadays are a must at airports in order to successfully get baggage from point A to point B in ever-growing airport terminals. Such a baggage handling system is responsible for getting passenger baggage from the check-in desk to the loading area of the passenger's aircraft. Not only does the system need to take care of routing baggage through kilometers of underground conveyor belts, but baggage also needs to be scanned, checked and sometimes adjusted manually. One can imagine that these systems are increasingly complex, both because of the amount of routing possibilities, as well as the use of redundancy in order to decrease the possibility of complete failure of the systems.

These baggage handling systems are captured in computer models, which describe such a system in a schematic way. Using these models, all routing possibilities of baggage are predetermined by the system architects. When baggage is inserted, it is known what parts of the system the baggage can possibly be routed through. At any position in the system, it should be known how to route baggage to any of those possible destinations. This knowledge is captured in a routing scheme which consists of routing rules.

Each system has decision points, which can be seen as the intersections on a highway having multiple exits. It is possible that a single bag can take two or more exits, and thus we want to indicate which exit should be taken. Such a decision is a routing rule: for a single intersection indicating what exit is most preferable, given a destination for a bag. As such, these routing rules make sure that a bag can always be routed towards its destination. In cases of failure in the system (e.g. breakdown of a conveyor), different sets of routing rules take over in order to cope with the changed situation in the system. Thus, the routing rules are adaptive towards the state of the entire system: when the system state changes, the routing rules change as well.

Currently, these routing rules are created manually at Vanderlande. Using the experience of employees, routing rules are tailor-made for each system. The goal of this project is to develop a method capable of generating a valid and good set of routing rules based on a given computer model of a baggage handling system. A valid set of routing rules is complete: it is always possible for a bag to be routed towards its destination. A good set of routing rules is optimal: the number of bags the system can handle under peak times is maximized. The developed method should be generic enough to work for any baggage handling system.

We will present a framework capable of generating such a set of routing rules. This framework consists of independent processes which can be used to come to a such a set of valid and good routing rules. This framework is displayed in Figure 1.

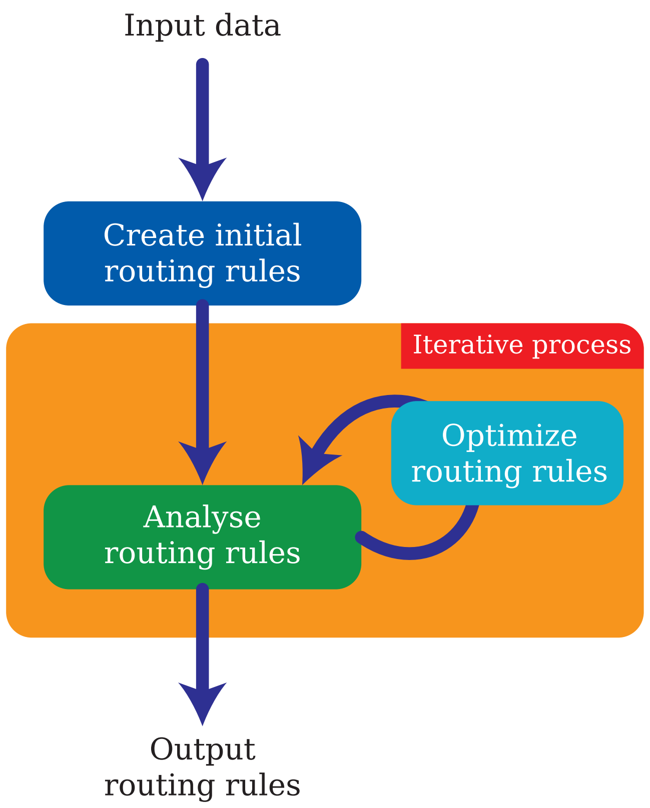


Figure 1: Framework used for reaching the project goal

In this framework, we have defined three distinct processes. Each of these processes will contribute on their own to the project goal.

INITIAL CREATION PROCESS is responsible for creating an initial set of valid (but not necessarily good) routing rules.

ANALYSIS PROCESS can analyse an arbitrary set of routing rules and present bottlenecks that are present in the system.

OPTIMIZATION PROCESS, based on the bottlenecks found by the analysis process, tries to resolve the bottlenecks by modifying the set of routing rules.

We intentionally introduce a distinction between the initial creation of routing rules and the iterative process of analysing and optimizing those rules. There are multiple reasons for this:

- Separation of concerns; we can more easily reason and argue about both processes if we keep them apart;

- Independent execution; having a separate analysis process, it is also possible to analyse and optimize already existing sets of routing rules;
- Different metric specialisations; the creation of the initial set of routing rules is primarily focussed on creating a valid set of routing rules (focus on single bags). The analysis and optimization of routing rules are primarily focussed on creating a good set of routing rules (focus on the entire system, with multiple bags in it). This separation of specialisation makes it possible for us to optimize each method for each metric.

Performance metrics will be introduced, through which we can indicate the performance of a set of routing rules. Extracting the definition of these performance metrics are also part of this graduation project, as they were previously not defined or ready to use for us. Using these metrics, we will provide validation for sets of routing rules, proving the validity (i.e., completeness) of sets. Subsequently, we will analyse the outcome of our framework, and compare the generated set of routing rules to those currently operating on existing baggage handling systems. This shows the performance of rules created with our framework compared to manually created routing rules; we indicate what the differences between the manually created rules and generated rules are, and why these differences exist.

In Chapter 1, we will provide a detailed context description on baggage handling systems and their inner workings. Subsequently, in Chapter 2, we discuss the problem of generating routing rules for a baggage handling system, and what exactly is needed to solve this problem. The approach for solving this problem is given in Chapter 3, where we go in depth on the presented framework. Details on the implementation of this framework are given in Chapter 4, which is followed by an extensive analysis of the presented framework in Chapter 5. We conclude this thesis with the results of our methodology, and a discussion about future and related work in Chapter 6.

1

BAGGAGE HANDLING SYSTEMS

This chapter contains a description of a Baggage Handling System (BHS) and of the general flow of baggage through such a system. Also, we discuss the major controlling parts of such a BHS, and show how routing of baggage is performed.

1.1 BAGGAGE FLOW

Before we can dive into the details about the inner workings of a BHS, and describing what process layers are present, we first need to explain how baggage flows through a BHS. An illustration of this general flow is depicted in Figure 2. The blue dashed lines indicate the main flow through a BHS. About 90% of all bags will only flow through these parts of the process.

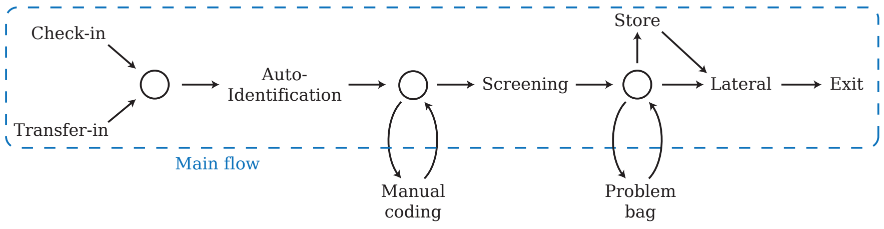


Figure 2: General flow of baggage

Generally, baggage is inserted in the system through either *check-in* stations, or *transfer-ins*. Check-in stations are the well-known front desks of airliners, where passengers report for their flight and drop off their baggage. Transfer-in is the input of baggage which needs to be transferred between flights. This applies to passengers who have landed on an intermediate airport, which is not their end destination, and need to switch aircrafts. Their baggage needs to be transferred from one aircraft to the other; also, sometimes this means the baggage has to be transferred between terminals.

After either of these two input possibilities, the baggage is now in the BHS. A bag is *automatically identified* using the label that is placed on the bag, using ID scanners. Sometimes, the label is unreadable, which results in the bag being diverted to a *manual coding* station. An employee identifies the bag if possible, and returns it to the main line. All bags after this point in the process need to be checked for any dangerous materials, thus all baggage will undergo *screening*. If a bag does not clear in screening, it will be purged from the system. Should the bag be deemed secure, it can enter the rest of the BHS (which is off-limits for unsafe baggage). In case a bag still cannot be identified even after it has been through manual coding, the bag will go to a *problem bag* station (after it has been screened). Here, the bag will be identified by a more thorough identification process, and returned to the main line.

If the passenger has checked his baggage in very early (i.e. a certain amount of hours before the flight), it cannot be stored in its corresponding aircraft yet due to regulations. In order to save valuable space in the system, the bag is then temporarily taken out of the main flow of the BHS, and moved into *storage*. There, it is placed on a rack until it can be processed further. When the corresponding aircraft can be loaded with baggage, these bags will be taken out of storage and reinserted in the main flow of the BHS.

Finally, when baggage needs to be loaded onto an aircraft, it needs to move towards the make-up zone, where all baggage for a specific aircraft is collected and stored inside the aircraft. The make-up zone is divided into *laterals*, which are the final conveyor belts on which the baggage exits the BHS.

Vanderlande strives for a BHS which is operational 100% of the time. In order to guarantee such a high availability, a lot of subsystems in a BHS are redundant. Every station is present multiple times, and important high capacity lines are duplicated. This means that whenever part of a BHS fails, it is still possible for the BHS to operate normally due to the huge amount of redundancy.

1.2 RUNNING EXAMPLE

Before we actually dive into the inner workings of a BHS, we will first introduce a running example which we will use throughout this thesis. Using a running example will enable the reader to visualize aspects of a BHS when they are explained. Also, the running example will provide the reader with a more general feeling of how a BHS works and what layout a BHS could have. We show the running example in Figure 3.

In the running example, we show a fictional BHS. The entire BHS is shown as a graph, with nodes as dots and edges as arrows. The nodes represent either stations where a bag undergoes a procedure (e.g. identification, screening) or where the system can change a bag's direction. The edges represent conveyor belts. One edge does not necessarily represent a single conveyor belt, it can also represent multiple conveyor belts, or part of a single belt.

This model is created specifically for this thesis in order to provide a clear example containing all major parts of a BHS which we want to talk about, more specifically, the most important aspects of the general flow as shown in Figure 2. This fictional BHS is therefore less complex than a real BHS, both in terms of having less functionality as well as less nodes and edges. The layout of this BHS is more streamlined, in order to ease understanding and have functionality immediately clear at a glance.

This running example consists of over 250 nodes and over 300 edges. Real BHSs can very well consist of over 2000 nodes and 2300 edges. This demonstrates the complexity of a real BHS compared to our running example. It should be mentioned, however, that all theory presented in this thesis has been validated and analysed against real BHSs, not fictional ones.

The graph visualization makes reasoning about the BHS easier, as distinct areas are clearly shown. Moreover, the layout of the graph does not necessarily have to be an exact copy of the physical system. This means that we can move around parts of the BHS to increase readability. A downside of this approach is visual deceit: one could instinctively correlate the length of an edge to the length of the conveyor belt it represents. However, a very short edge in this visualization could very well be one of the largest conveyor belts in the physical BHS.

As said, we have almost every aspect of the general flow present in this running example. On the left-hand side of the BHS, shown in blue, we have all input points for the BHS. We have four identical groups of check-in inputs, and two groups of transfer-in inputs. Subsequently, baggage entering the BHS through these input points are collected on the two main loops, indicated in dark red and dark blue.

On these main loops, a bag can undergo screening, shown in purple, or go to manual coding for identification, shown in green. If the bag can be loaded onto the aircraft, it will go to one of the output points on the right-hand side of the graph, shown in pink. If the bag cannot be loaded onto the aircraft yet (for example, because it is too early), the bag can go to the store, shown in yellow. It should be possible to directly go from the store to one of the output points, as such there are dedicated lines from the store to the output area.

The problem bag part of the process is not shown in this running example, mainly because it would complicate the example. Moreover, the problem bag area of a BHS plays a very small role in the entire operations of the system, and is thus not of high importance for our example.

A good thing to notice is that a lot of edges (mainly the loops) are often present in pairs, i.e. there is a lot of redundancy in the system. This is a chief aspect of any BHS made by Vanderlande, and this aspect will play an important role in some of the decisions we make.

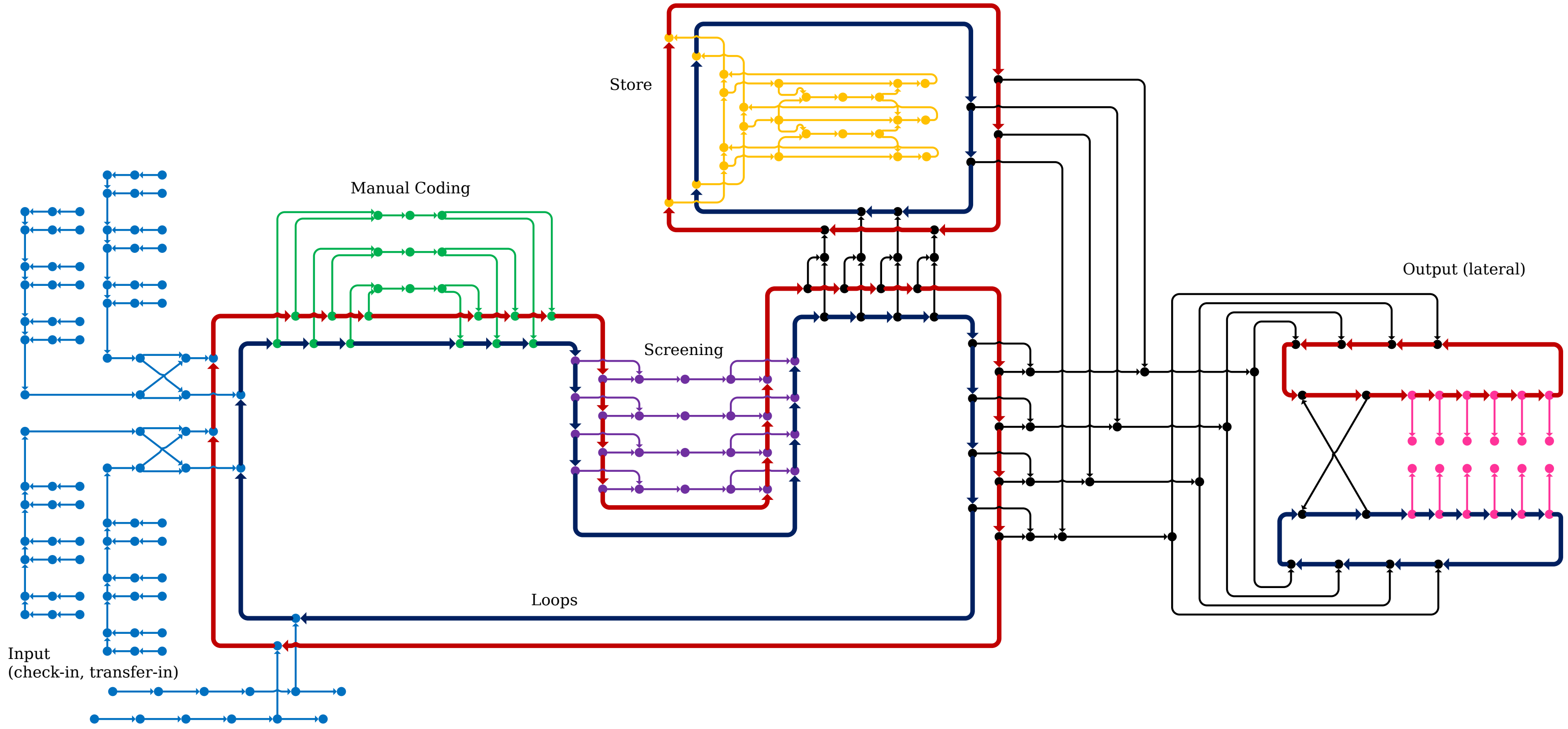


Figure 3: Running example of a BHS

1.3 TERMINOLOGY

In order to easily identify specific parts in the graph representation of a BHS, we introduce a few terms which we will use extensively throughout this thesis.

DIVERT Whenever we have a node with one ingoing edge and two or more outgoing edges, we have a *divert*.

MERGE Symmetrical to diverts, whenever we have a node with two or more ingoing edges and one outgoing edge, we have a *merge*.

Both diverts and merges can change the direction of bags, albeit only diverts can do this actively. Diverts have a single stream of incoming bags, and they have the possibility to propagate bags over one of their outgoing edges. Merges have incoming streams of bags over multiple edges, and (as the name indicates) merge the incoming edges into a single outgoing edge. Diverts are the main decision points in a BHS, because at each divert a bag can be routed to one of its outgoing edges.

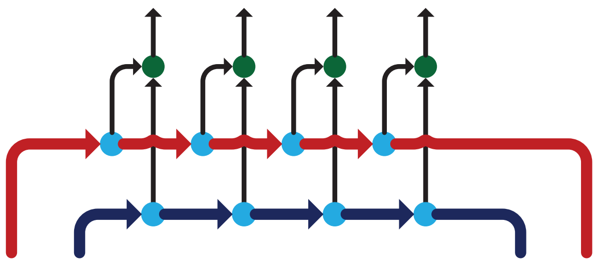


Figure 4: Example of diverts and merges

In Figure 4, a snippet of the running example is shown. More specifically, the transition from the two main loops towards the store. Here, bags can either continue on the loop, or take an exit towards the store. This example shows both diverts and merges, indicated in light blue and dark green, respectively.

STATION A node where a bag can undergo a procedure, like screening or identification.

TASK Procedures like screening or identification are called tasks. Whenever a bag is in the system, it always has a task which it must complete.

DESTINATION Only certain stations can complete tasks for a bag. Thus, all possible stations that can complete a task for a bag will be its possible destinations.

For now, one should assume that a bag always has a task, and that through this task the destination of the bag is defined. In Sections 1.4.2 and 2.5, we will explain in more detail how and by whom these tasks are handed out.

Sometimes we will also discuss the relative position of one area inside the BHS compared to another area. To indicate this, we use the terms *upstream* and *downstream*. For now, one should assume that everything in a BHS generally flows in one direction. An analogy best describes these two terms. Imagine a river; the river always flows in one direction, and at any point on that river, we have an the upstream part of the river (everything against the flow of the river) and the downstream part of the river (everything with the flow of the river). This can also be applied on BHSs.

UPSTREAM Whenever at a specific point in the BHS, everything in reverse direction of the BHS is upstream.

DOWNSTREAM Whenever at a specific point in the BHS, everything in the normal direction of the BHS is downstream.

1.4 LAYERS OF CONTROL

BHSs generally span many square kilometers of conveyor belts, all of which are controlled by multiple layers of logical processors. At any time thousands of bags can be present in a BHS. These bags need to be tracked individually, and specific parts of the system have to react per passing bag.

In order to cope with such a level of complexity, control of the system is divided into layers. For this project, we have to deal with three of those layers: the process layer, the logistics layer, and the controls layer. The process layer consists of the *Process Controller (PC)*, whereas the logistics layer consists of the *Logistics Manager (LM)*. The controls layer consists of smaller components, the *Programmable Logic Controllers (PLCs)*, which are responsible for managing the hardware parts of a BHS. Each layer communicates with other layers in order to retrieve the information it needs. This communication is shown in Figure 5.

The PC is responsible for keeping track of a bag's progress in the higher level process (like in the process displayed in Figure 2). It only knows what a bag is currently doing (e.g. undergoing screening) and what it should do after that is done (e.g. go to a lateral). It has no knowledge of the state of the BHS, this is the main concern of the LM.

The LM receives from the PC what the next task of a bag should be, and where it can possibly be routed towards (i.e. a list of possible destinations). Once the LM has determined a single destination for a bag, it communicates this towards the PLCs who will direct the

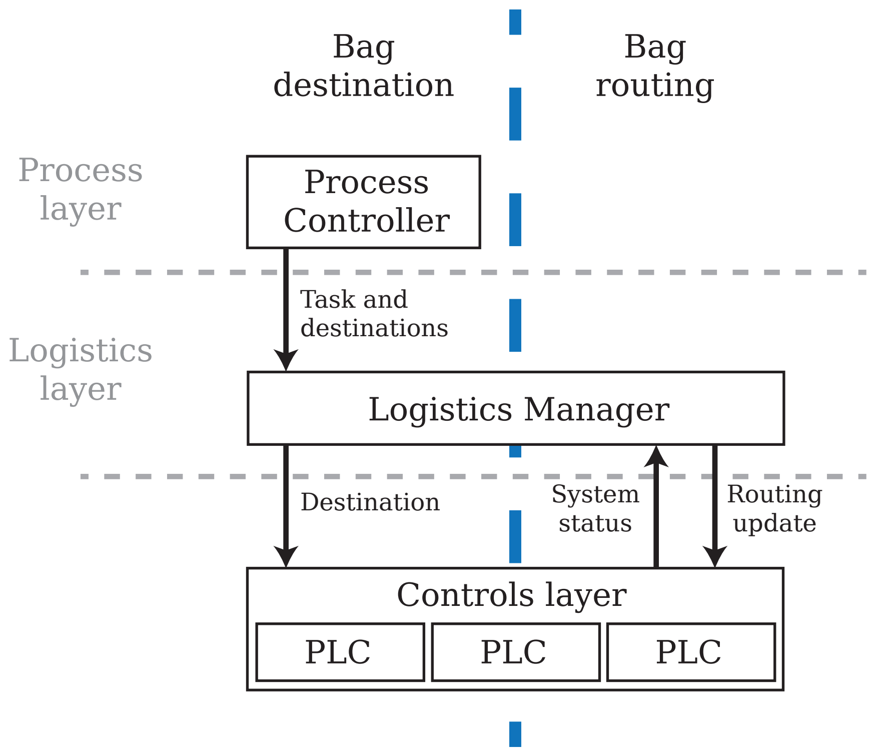


Figure 5: Communication between layers

machinery in order to route the bag. Should there be a failure in the BHS, the PLC managing the failure will notify the LM, who will update the routing of each PLC so that bags in the affected area(s) will be rerouted.

The main focus for this project is the LM. Because we still need the necessary information from the other layers, we will emulate those layers by supplying the necessary information when needed. Whenever we need to emulate information coming from either the process layer or controls layer, we will discuss our methodology for doing so.

1.4.1 *Programmable Logic Controller*

A PLC is responsible for controlling the hardware components which consists of actual conveyor belts, components which can change the direction of baggage, etc. A PLC only knows what direction a bag should go, when it arrives at a decision point in the area it controls. It retrieves this decision information from a routing table, which is created by the LM for each PLC specifically.

Initially, these routing tables are created by the LM, which assumes the entire BHS is completely operational. Whenever there is a failure inside an area which a PLC controls, the PLC will report this failure to the LM. The LM will then, using its own information, create a new routing table for the PLC which will route baggage around the failed part. The actual creation of routing tables, and their use, is outside the scope of this graduation project. We are only interested in the information needed to create these routing tables.

1.4.2 *Process Controller*

The PC keeps track of the progress of each bag in the system. Whenever a bag has completed a task, the PC will know what the next task should be, and it will communicate this towards the LM. Since the PC is only aware of the next task of a bag and it does not know the current state of the BHS, it will present the LM with all possible destinations for that task. It is then the responsibility of the LM to reduce this list of possible destinations to one.

As mentioned, we have a set of stations at which a bag can undergo the task assigned to it. This means only out of this set of stations a single destination can be appointed to the bag, not any other stations or nodes. This simplifies the system, because only a relatively small set of nodes can be a destination. This increases the understanding for all engineers who have to work with the BHS.

All functionality of the PC is outside the scope of this graduation project, and therefore we assume that the LM always receives a task and list of destinations for each bag.

1.4.3 *Logistics Manager*

The main responsibility of the LM is to make sure that a bag has a valid destination, which it can reach. It can reduce the list of possible destinations by using internal heuristics based on the current state of the BHS, which it knows through all of its PLCs. This reduction method is outside the scope of this project, and thus we assume it exists and works properly. Furthermore, the LM needs to make sure each PLC has a routing table which contains a valid routing (i.e. baggage can reach its destination).

An explicit decision has been made for handing out one destination per time, per bag, instead of just handing out the entire list of destinations and calculating the entire route of a bag at once. A BHS is constantly changing, and it is quite easy for one bag to create a snowball effect of events affecting other bags. When a specific station breaks down, all bags that were routed towards that station need to be rerouted towards another station, and this could very well mean that all bags for those stations need to be rebalanced. The bags themselves can also impose changes on the BHS: the flight of a bag can get cancelled, resulting in a change of scheduling, or the label of a bag can become unreadable. The BHS has to react per bag, and because the bags and the BHS itself are constantly changing, all bags in the BHS also have to be routed with these changes in mind. It simply is not efficient to calculate the entire route once if this route needs to change many times throughout the process.

1.4.4 *Routing rules*

Routing tables are created by using *routing rules*. These routing rules are used as a basis on which the LM creates routing tables for each PLC, and thus it is important to note that these two definitions are not the same. A routing table is the result of a function based on routing rules.

One could reason why these routing rules exist at all; why such a static method determines the way a bag needs to be routed, instead of just calculating the routing possibilities real-time for each bag. There is one major argument against dynamic routing: understandability. The engineers at Vanderlande who need to work with a BHS need to understand how the BHS routes baggage through its system. If each bag would be routed dynamically, explaining the behaviour of the system would be very difficult. Predictability is in this case more desired than efficiency.

1.4.4.1 *Usages*

At diverts, there are multiple outgoing routes. When a bag arrives at a divert, a decision has to be made what route the bag will be forwarded on, based on the destination of that bag. The first edge of each outgoing route determines this decision, and the information which edge the bag will be forwarded on is supplied in the form of usages. These usages are defined by Vanderlande, and are used internally in multiple layers of control. For each outgoing edge at a divert, four usages are possible:

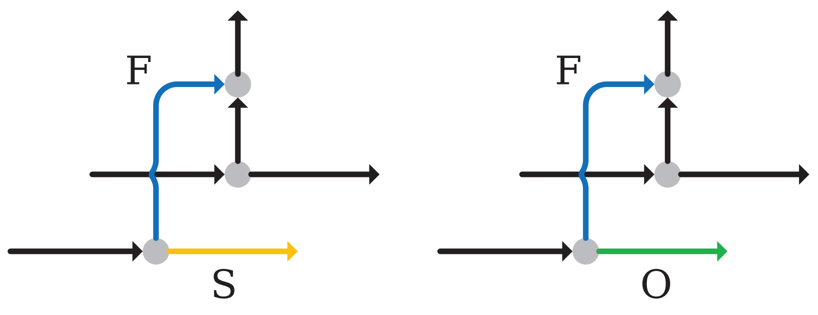
FAVOURITE (F) The preferred route for the destination. At every divert, at least one outgoing route is marked as Favourite.

SUBSTITUTE FAVOURITE (S) In case the Favourite route is unavailable, the route with this Substitute Favourite can be used as an alternative.

OPTION (O) Similar to Substitute Favourite, except that the respective PLC is aware of this route, whereas it is not of routes marked as Substitute Favourite.

NO ROUTE Also noted as the absence of a usage on a route. This means that for the destination, this route is not considered at all.

The abbreviation for each usage (F, S, O) is given for future reference. The No Route usage does not have an abbreviation, as it is defined by the lack of a usage on an edge. For each divert, for each reachable destination, each outgoing route is marked as one of the three usages, or no usage at all. An example of usages on routes is given in Figure 6.



(a) Favourite and Substitute Favourite (b) Favourite and Option

Figure 6: Example of usages on routes

The difference between Substitute Favourite and Option is that in the generated routing tables for the respective PLC, routes with Option are present, whereas routes with Substitute Favourite are not. This means the PLC is aware of routes with Option, but not of routes with Substitute Favourite. The reason behind this is that the PLC can redirect baggage on its own, without intervention of the LM if it has Option routes available.

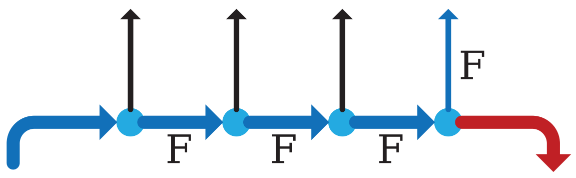
When we have applied a usage on an edge, for a specific destination, we have created a routing rule. These routing rules are nothing more than a function defined on the Cartesian product between edges and destinations. If an edge is important for the routing towards a specific destination, there will be a routing rule indicating what usage is applied on the edge for that specific destination.

Whenever a destination is reachable from a divert, there will always be at least one outgoing edge for which a routing rule is defined. This routing rule will indicate that edge as the Favourite edge for traveling to that destination. This way, we can ensure there will always be a way for a bag to be routed towards its destination.

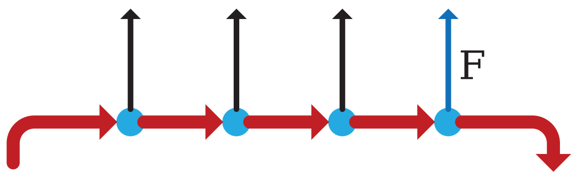
1.4.4.2 *Transport default*

Besides usages, another concept of routing preference is used primarily on high capacity main loops. Because these main loops are a major part of a BHS, they have their own preference system based on the usages described in Section 1.4.4.1. The idea of a transport default is introduced, and the following analogy can best explain this system. On highways, for example, signs indicate each destination for which one should exit the highway. Every other destination is implicit; it is not given, assuming the driver knows that he should stay on the highway until his destination appears on one of the exit signs (in general, at least). The principle is applied for the transport default. Using transport default, we can indicate at each divert for what destinations the baggage should leave the main line. For any other destination, it should stay on the main loop.

A main loop (i.e. transport default) is a physical property of a BHS. Once we have a loop, it is by definition a main loop because everything on it can recirculate. This means that every position on a main loop is reachable over and over without taking a divert, because of the very nature of a loop. This physical property of the loop is captured in a single data field, which indicates for a set of edges whether it is a transport default or not. Indeed, this information can also be extracted by the topology of the BHS, but Vanderlande has chosen to supply additional information, thus losing the necessity of deducing a transport default.



(a) No transport default



(b) Transport default

Figure 7: Example of transport default

An example of the use of transport default is given in Figure 7, which is a simplified version of the situation shown in Figure 4, now with only one main loop. Assume we have a bag which needs to exit the loop at the last divert, shown in orange. If we do not use transport defaults, we need to indicate for each divert along the way what the preferred outgoing edge would be. As is shown in Figure 7a, this would mean that we have three Favourite usages on the main loop, and one Favourite usage indicating the exit. Now, with transport default, the main loop is the implicit 'ongoing' route. If there is no usage for the destination of a bag, just route it over the transport default. As is shown in Figure 7b, we now only need to explicitly state the exit edge on the last divert, the other three diverts are implicit because of the transport default.

1.5 BREAKDOWNS AND REROUTING

Vanderlande strives for a BHS which is operational 100% of the time. In order to achieve this, any BHS is designed towards this high availability concept. As can be seen in the running example in Figure 3, a lot of redundancy is present in a BHS in order to be able to cope with breakdowns of parts of the BHS. When one part of the system fails, another part of the system will function as back-up. In order to cope with failures, several systems have been devised to indicate and respond to these situations.

1.5.1 *Status edges*

Each edge in the system can fail, and usually this means that edges in the direct vicinity can no longer operate as well. An example is given in Figure 8. Assume we have three consecutive edges and the edge in the center fails; now the other two edges before and after the failing edge can no longer correctly transport baggage. In order to group edges together which are failure-sensitive towards each other, we define a *status edge*: a set of edges which all propagate the same failure when one of them in the same set fails. This means that, when one edge fails, all others in the same status edge fail as well.

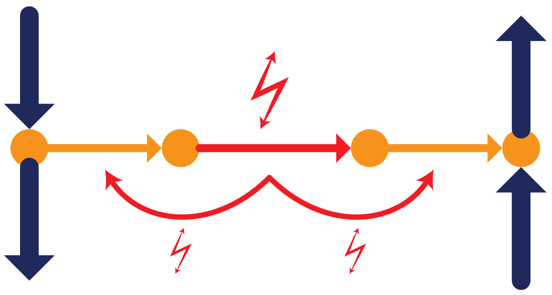


Figure 8: Affected edges when one fails

1.5.2 *Influence rules*

When failures in the system arise, and certain (status)edges become unavailable because of it, the routing tables of the affected PLCs need to be updated. In case of small breakdowns, baggage can no longer be routed over the unavailable edge and rerouting is needed. In case of large breakdowns (whole areas become unavailable), rerouting is necessary to prevent baggage from being routed towards the unavailable area and becoming trapped.

For these failures, influence rules exists. When a part of the system fails, influence rules bound to that part of the system will come into effect. I.e. influence rules are defined on areas of a BHS, and will only be enabled when that specific part fails. An influence rule dictates what routing rules will have to change once it comes into effect, meaning it will change the usages on specific edges during the time the influence rule is enabled. Once the failure has been solved, the influence rule will be disabled and the original routing rule will come back into effect.

An influence rule will only change the usages on edges that are necessary to reroute all baggage in order to circumvent the failing area. It should be noted that it is possible for multiple influence rules to come into effect simultaneously. In this case, should affected routing rules overlap, the influence rule that is enabled last will dictate the changes.

1.6 SUMMARY

In this chapter, we have explained the definition of a BHS and shown how a BHS works internally. We have introduced a running example in Figure 3 which we will use throughout this thesis. We display a BHS as a graph, using nodes for important parts in the system where something can happen to a bag, and edges as abstractions for the conveyor belts.

Diverts and *merges* are special nodes in the system. Diverts are decision points in the system, where the ongoing route of baggage can be altered, with a choice for two or more outgoing edges. Merges are points in the system where multiple ingoing edges come together, and are merged into a single outgoing edge.

Bags always have a *task* which they need to complete, for example to go through screening or manual coding. Each node in the system where a task can be completed is a *station*, and a bag will always be routed towards one of these stations, also known as the *destination* for that bag.

A layered control structure exists, and this control structure consists of a PC, an LM, and multiple PLCs. The PC is responsible for handing out tasks for each bag, but is not aware of the current state of the BHS. A PLC manages a specific area of hardware components, and physically routes baggage through an area by making use of routing tables. These routing tables are created by the LM through the use of *routing rules*.

The LM is the main focus of this graduation project. Sets of routing rules are created based on an extensive layer of information on top of the topology of a BHS. This layer consists of the notion of usages (Favourite, Substitute Favourite and Option), and transport default to indicate what the best possible choice would be at any divert.

Finally, we have extended this notion of routing by introducing additional systems used when parts of a BHS breaks down and rerouting is necessary. We have shown status edges (which group edges together), and influence rules (which are modifications on the routing rules used in a fully functional system).

2 GENERATING ROUTING LOGIC

Using the knowledge of the system as presented in Chapter 1, we can now define the problem we are going to solve in this project. In this chapter, we will explain the basic routing problem, and subsequently add more complexity in order to reach the final definition of the problem. We talk about the conditions in which we are to operate, as well as the performance measurements needed for analysing the problem's solutions. Several interesting scenarios are presented for us to take into account when devising a solution.

2.1 ROUTING

For the LM to be able to make routing decisions, the LM primarily relies on the usages defined on edges (as described in Section 1.4.4.1). These usages are captured in routing rules, and a collection of routing rules enveloping an entire BHS is called a set of routing (logic) rules.

In the end, we want to achieve the most optimal set of routing rules for a given BHS. Due to the complexity of a BHS, automatically generating the most optimal set of routing rules is challenging. For some rules, human insight and knowledge of the system is necessary, something which cannot (or at least, not easily) be emulated through computer algorithms. Thus, we strive towards automatically generating the most cumbersome and repetitive rules, those whom can be deduced through basic logic and are mathematically proven, whilst still trying to optimize the set for the given BHS. We define the core goal of this graduation project as follows:

> To develop a generation method capable of generating for a BHS a valid and good set of routing rules, consisting of routing rules for each divert towards all reachable destinations.

There are two major points of interest in the goal definition given above, namely *valid* and *good*. A valid set of routing rules enables each bag to reach its destination. The bag should not get trapped in an area, even when specific areas of a BHS break down and rerouting is necessary. We will elaborate on this in Section 2.4. A good set of routing rules is subject to the goals which we will describe in Section 2.3, and several performance measurements present in Section 2.7.

2.2 OPERATING CONDITIONS

Before we can actually measure how well a set of routing rules performs, we first need to set the constraints and assumptions for the context of our problem. Vanderlande's main focus with BHSs is to minimize the delay of baggage. Naturally, a system will most likely have trouble performing optimally when under high stress, and in the case of a BHS, this stress will be primarily caused during peak performance. How the system is able to cope with high amounts of incoming bags during peak times is directly related to the amount of bags that will be delayed. In order to recreate these peak performance moments, we assume worst case scenarios throughout this thesis, and define this in a singular worst case scenario for an entire BHS.

To simulate the high stress on the system during peak times, we can look at the input points in the system. If one looks at the general flow of baggage in Figure 2, one can see two input points: check-in and transfer-in. The amount of bags we insert in the system through these input points define the load on the system. For each BHS, Vanderlande has requirements on the performance of that BHS. As such, they have a requirement stating what peak load the system needs to be able to process, i.e. the amount of bags per hour during peak performance the system should be able to process.

For our worst case scenario, we define that the system will receive this peak load to process (which, naturally, differs per BHS). However, there are two assumptions for this worst case scenario:

1. The peak load will be distributed equally over all possible input points (check-in desks, transfer-in points, etc.);

2. The main flow has the most impact on the performance of the system.

For Case 1, equal division of load over all input points is a direct result from the peak load requirement. We do not want to create artificial bottlenecks in the system by excluding input points from the peak load, thus we take all input points into account. For Case 2, we have a main flow through which 90% of all baggage flows. If we are interested in the system's performance, we want to look at the bottlenecks present during peak load in these main flow parts of the system. The remaining 10% processed outside of the main flow will have little effect on the system, for two reasons:

1. The amount of bags passing through the non-main flow parts of the BHS are already much lower;

2. The non-main flow parts of the BHS are often dedicated areas through which main flow baggage does not pass.

Thus, the possible bottlenecks in the non-main flow parts are not of our main concern, and the main flow will primarily have our focus. However, within the bounds of this worst case scenario, we also want to stress the system equally. The BHS has multiple (intermediate) destinations, and we want to spread the load over all these (intermediate) destinations such that the entire BHS will be used through the generated set of routing rules. In other words, we want our routing rules to maximize the coverage inside the BHS. This is something that relaxes the worst case scenario somewhat, because we are not interested in the worst case scenario when it results in completely unrealistic behaviour. This would yield no usable information on the performance of our set of routing rules, and thus we use the maximization of coverage to create a more realistic worst case scenario.

For the remainder of this thesis, we will assume this definition of worst case scenario under all circumstances. Each aspect requiring new constraints and/or assumptions for this worst case scenario will be explained accordingly.

2.3 KEY PERFORMANCE INDICATORS

For Vanderlande to be able to guarantee a customer that the BHS they deliver meets the negotiated standards, there are several Key Performance Indicators (KPIs) to measure the performance of such a BHS. Routing rules form a small part of the very complex system that a BHS is. Still, they can contribute greatly to its performance. If a set of routing rules is badly designed, baggage can get lost or stuck. This all results in the same thing: baggage not arriving on time at the aircraft they are supposed to be.

There are two important goals Vanderlande tries to achieve with all of its BHSs:

1. Reduce the amount of bags delayed to such an extent that they miss their flight;
2. Maximize the throughput of the system during peak times.

Naturally, it becomes obvious that the second goal is tightly coupled with the first goal. These two goals should be kept in mind whilst generating routing rules. Since we cannot directly measure the KPIs related to these goals (and the effects which influences them) due to their complexity, there are several causes to these effects which we can measure and will pay attention to throughout this project.

2.3.1 *Switching*

Both diverts and merges are points in a BHS where the system can physically alter the direction a bag is traveling. It does this by either shoving a bag off a conveyor belt, dumping it on another a belt, or moving an entire conveyor belt into another direction, for example. Whenever a divert[1] needs to alter the direction of a bag, the divert needs to switch from its current position to an other position. Switches are mechanical operations and thus take time, besides wearing on the machine doing the operations.

In our case, the mechanical wear on the machines is not directly important. This, because it is not a short-term problem. Eventually, the wear on the machine will cause it to fail more often, but these long-term problems are outside the scope of our project. We are more interested in the time it takes to switch a divert. Whenever a divert has to switch, the divert is temporarily unable to route baggage, and can thus be considered as disabled. All oncoming bags which need to be routed over this divert will thus have to wait until the divert is done switching for them to continue on their way to their destination. This will result in bags being temporarily delayed, which is directly related to the first goal. Moreover, when during peak times the load on the system is increased, in a worst case scenario switching has to be done equally more often, and thus the second goal is affected as well.

2.3.2 *Exceeding capacity*

Each conveyor belt in a BHS (and thus, also each edge in the graph) has an upper bound on the number of bags it can process per hour: its capacity. Whenever an edge exceeds its capacity, bags that are to enter that edge will have to wait until the edge has space available to process them. This means that any bags that are to go over that edge will have to be rerouted over other edges, in order to still reach their destination.

The result of edges exceeding capacity is two-fold. The edge is unavailable for routing, and bags will have to take an other route in order to get to their destination, which can increase their total travel time (i.e. the total time it takes for the bag to get to its destination, from its starting point). Also, since we have to reroute, other diverts and merges will be stressed more, because they will have to process the rerouted bags as well as their regular flow of bags. This increase in number of processed bags can result in more switch operations for diverts and merges on the rerouted path. This results in this KPI also being related to the switching KPI.

1 Here, every mention of a divert is analogous for merges. We leave out the mentioning of merges for the sake of readability

2.4 ADHERENCE TO PROPERTIES

When we want to generate a set of routing rules, we need to adhere to several properties of a BHS. We will list these properties here and indicate why they are necessary.

Capacity

Obviously, a bag is not alone in a BHS. Even on atomic parts of the system (e.g. a single conveyor belt), multiple bags can reside. This means that we have to take into account the capacity of each of the subsystems of a BHS.

The BHSs on which we have to apply our routing algorithm have already been extensively reworked (by layout architects) such that they will be able to meet the terminal's demands for the next 15 to 20 years. Generally, the main flow of a BHS (as indicated in Figure 2) has the highest capacity. Any alternative routes (which are taken substantially less often) have much lower capacity. These differences in capacity have to be taken into account, such that we do not send huge amounts of bags over low capacity lines, essentially clogging the BHS.

Discourage certain areas

For some parts of a BHS we would like to discourage the routing of bags, because they are simply not meant for bags, or because they are redundant paths which are not capable of processing large quantities of bags.

For example, a BHS could contain storage units for the carts in which baggage is transported in some BHSs. Naturally, these storage units are not meant for baggage, and thus routing through them should be prevented.

Load balancing

Aside from actually getting a bag in time to its final destination, we also need to take into account the equal division of load over the system. In order to explain this equal division of load, we show an applicable scenario in Figure 9.

Here, assume we have a starting node and an ending node (red and blue, respectively) and four parallel paths in between. The paths are equivalent, and it does not matter which of the paths we take in order to get from our starting node to our ending node. If we divide our load from the starting node equally over the four paths, we can process all bags four times faster than if we were to take only one path and propagate everything sequentially.

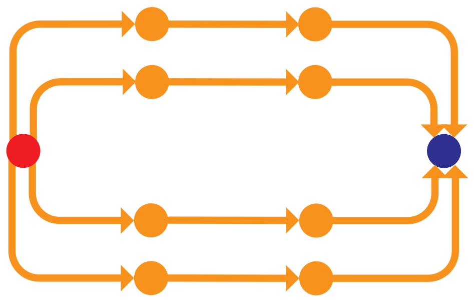


Figure 9: Parallel paths applicable for load division

We mentioned in Section 1.2 that Vanderlande explicitly introduces redundancy in its systems. This redundancy is present primarily as a failsafe system; in case an area breaks down, there are still other ways to circumvent the disabled area. However, when the system is fully operational, those redundant paths can be used to divide the load on the system.

2.5 TASK TOPOLOGY

In Section 1.3, we already mentioned the use of tasks in order to know what operation a bag should complete next. These tasks are handed out by the PC, who has knowledge about these tasks and in what order they should be handed out. Whenever the LM receives a task for a bag, it also receives a set of possible destinations from the PC. Because a bag can only be routed towards one destination at a time, the LM needs to reduce this set of possible destinations to a single destination. For this, the LM uses the *task topology*, and through this task topology, we also emulate a part of the needed functionalities of the PC.

In the task topology, *task groups* are defined. A task group is a set of edges which are related towards each other based on the tasks that can possibly be handed out whilst a bag is on one of those edges. For example, if we define all edges in a main loop as a task group, then all tasks that can be handed out for bags that main loop are the same for all edges in that task group. Additionally, we have sets of stations, called *station groups*. A station group consists of stations that can perform the same task, e.g. all stations in a station group can only perform screening operations.

The task topology is a function of the Cartesian product between a task and a task group, which maps to one or more station groups. Thus, with a specific task and a task group, we get one or more station groups out of the task topology. In a more intuitive sense: we can look up the possible destinations once we know where a bag is in the system and we know what task the bag should complete next.

This resulting set of possible destinations will be intersected with the set of destinations received from the PC, and with further set reduction the LM will come up with a single destination towards which the bag will be routed. This final set reduction method is outside the scope of this project, and one should assume that the LM will always be capable of reducing a set of possible destinations to one. Also, it is important to note that the creation and maintenance of the task topology is outside the scope of this project; we only use the task topology as additional user input.

We show a practical example of the use of the task topology. As said, the next destination of a bag depends on the current position of the bag inside the BHS. In Figure 10, we have a snippet from the running example where we can see the two main loops, and the screening stations reachable from both loops.

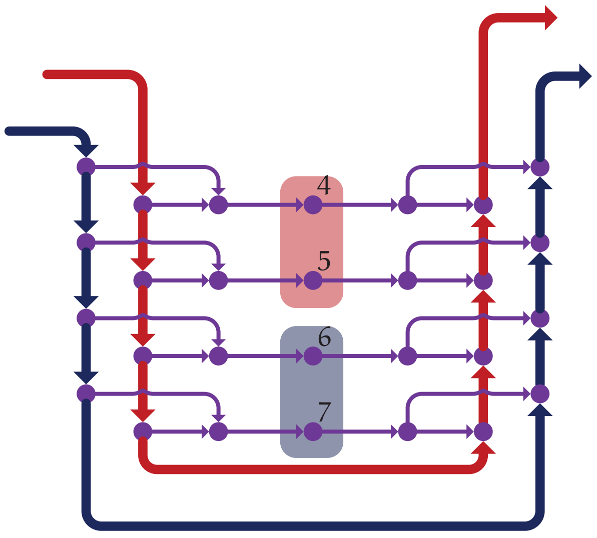


Figure 10: Task topology example on screening

Whenever a bag needs to undergo screening, it can visit one of the four screening stations, from either of the loops. However, we would rather want to dedicate half of the stations to the red loop, and the other half to the blue loop. This would result in two stations for each loop, which again results in less switching at the merges just before the screening stations. This helps reduce the total number of switches as explained in Section 2.3.1.

We can see the result of this separate dedication in Figure 10, where stations 4 and 5 form one station group, and stations 6 and 7 form an other station group. The red loop is a task group, and the blue loop is an other task group. So, if we have the task 'Screening', and look up the station group for the blue task group in the task topology, the set $\{6, 7\}$ will be the resulting station group. Similarly, if we do the same for the red task group, the set $\{4, 5\}$ will be the resulting station group.

Through this task topology, we can further extend our worst case scenario described in Section 2.2. We mentioned wanting to maximize the coverage of our simulations within the bounds of the worst case scenario by using all destinations. We now know all possible destinations that can be handed out at any given point in the system. Thus, we will use all of these possible destinations equally during the generation of a set of routing rules.

2.6 INTERESTING SCENARIOS

If calculating the routing logic were straightforward, this graduation project would not have existed. Therefore, we now show some interesting scenarios in which an optimal routing is not as obvious as one would think.

2.6.1 *Cross*

Redundancy is one of the main aspects of a BHS, and thus there will often be multiple paths leading to a single destination. An example of this is given in Figure 11. This is a snippet from the running example, just before baggage from the input reaches one of the main loops.

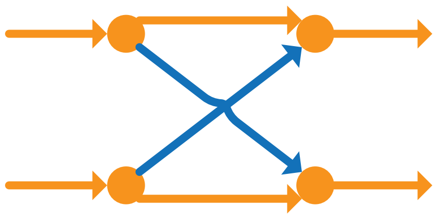


Figure 11: Cross with redundant edges

We have two dedicated paths (shown in orange), both coming from separate groups of input points. The crossing edges (shown in blue) are redundant paths. This division of dedicated/redundant paths is present in the data supplied along with the BHS, and we can retrieve this through the merges. Each merge contains data on what edges are dedicated edges and what are redundant edges.

When further downstream of this cross a path fails, the redundant paths are there to reroute baggage over the other path if necessary. However, these redundant paths often have much lower capacity. There are two possibilities when deciding how to route baggage over this cross, shown in Figure 12.

In Figure 12a, we show the desired routing of baggage. Both dedicated paths are used, the upper path leads to the red loop, the lower path to the blue loop. However, another possibility we do not want is shown in Figure 12b. Here, not the dedicated paths are used, but a re-

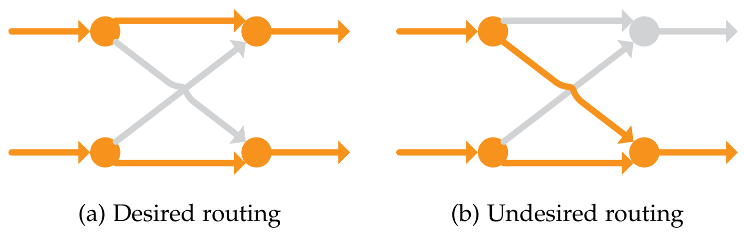

(a) Desired routing (b) Undesired routing

Figure 12: Routing on the cross

dundant path is used. This results in baggage being routed over lower capacity edges. Moreover, two large flows of baggage are merged at the lower merge. Obviously, we want to prevent the latter routing possibility.

2.6.2 *Loops*

Similar to the scenario described in Section 2.6.1, due to redundancy there are often multiple paths in order to route baggage from one loop to another. Again, we show a snippet from the running example in Figure 13. Here, we show the transition from the two main loops to the two storage loops.

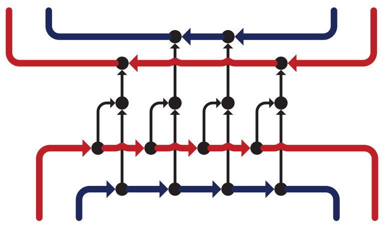


Figure 13: Multiple loops

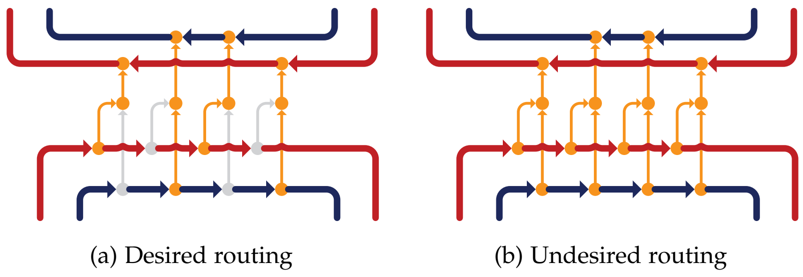

(a) Desired routing (b) Undesired routing

Figure 14: Routing on the loops

We can see four paths leading from the lower two loops to the upper two loops. We want to route baggage from one loop to the corresponding other loop. In this case, we want to route baggage on the red main loop to the red storage loop, the baggage on the blue main loop to the blue storage loop. We would rather not intertwine these flows, because we would need to merge flows coming from two loops, possibly creating a bottleneck.

However, a greedy approach to choosing the best route could very well result in the routing shown in Figure 14b. Here, all possibilities of routing baggage from either of the loops are being used. This would result in unnecessary merging. We do want to dedicate paths between the loops, as shown in Figure 14a. Now, we dedicate paths per loop, decreasing the change of the merges clogging up due to two major flows of baggage that need to be merged.

2.6.3 *Crossing of flow*

One last scenario is that of the merging of multiple flows of baggage. An example is shown in Figure 15. Here, we have a snippet from the running example showing the two main loops, and the two store loops. Both have outgoing edges, that are being merged into a single edge, eventually leading to the output points.

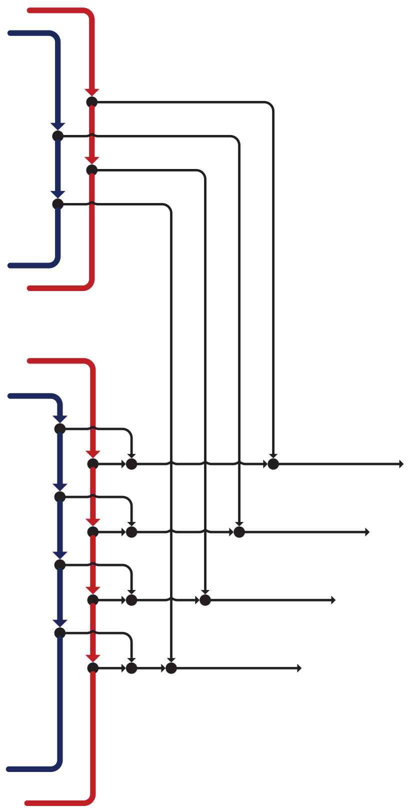


Figure 15: Multiple loops merged into one

Assume both sets of loops contain many bags, and that merging those two flows of bags has to be done carefully. If the two flows are merged bluntly, as is shown in Figure 16b, severe bottlenecks could be present at the merges. This, because the single edges after the merge are not capable of handling such a large amount of bags. Moreover, the merges have to switch between the two incoming edges all the time, which affects the switching KPI as described in Section 2.3.1.

More desirable is the routing as shown in Figure 16a. Here, we dedicate two of the lines to each of the sets of loops. This means that the merges do not have to switch anymore, as there is just a single incoming edge per merge that actually has a flow. Note that the desired routing still has some leeway, as we can also further dedicate the outgoing edges from the main loops. It could also be reasonable to dedicate one edge per main loop instead of two, but that is a discussion other than that of the crossing of flow, and is a combination of the cross and loop scenario shown earlier.

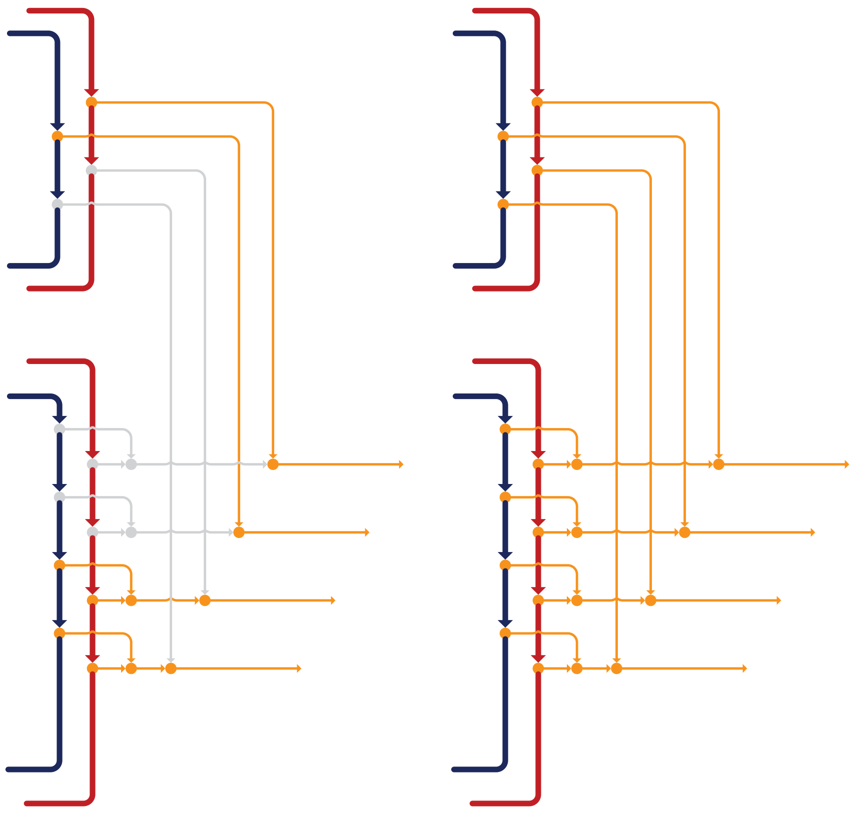

(a) Desired routing (b) Undesired routing

Figure 16: Routing on the crossing of flow

The interesting scenarios presented are one of the most common constructions in BHSs where straightforward routing cannot be achieved without some tailor made constructs. This is why we will pay special attention to these scenarios throughout this project and the solutions we present.

2.7 PERFORMANCE MEASUREMENT

In Section 2.1, we mentioned we want to achieve a set of 'good' routing rules. Now that we have defined our KPIs, explained which parts of the BHS affect these, and what interesting scenario's we need to take into account, we can define a performance measurement on a set of routing rules.

We want the peak performance of the system to be as high as possible, i.e. the system should be able to handle high amounts of load for a reasonable amount of time. In order to effectively measure this, all of the following performance measurements take the worst case scenario into account. In Section 2.2, we already explained the context of this worst case scenario and how it affects our system. For each performance measurement, the worst case scenario has a different impact, and each impact will be explained accordingly.

2.7.1 *Amount of capacity bottlenecks*

As described in Section 2.3.2, we want to use as much as possible of the capacity given to us, but we do not want to exceed this capacity, as it would result in clogging the BHS. In order to measure the performance of a set of routing rules on this point, we can simulate a large amount of bags traversing through the BHS. We divide the bags equally over the available destinations, and check if the routing rules would result in any bottleneck where we approach or exceed the given capacity of the system.

2.7.2 *Amount of switches*

For each divert and merge, we have bags that need to travel over these diverts and merges. In a worst case scenario, these bags arrive at a divert or merge in the least efficient way possible. An example is given in Figure 17. Here, we see a divert and merge in their worst case scenario.

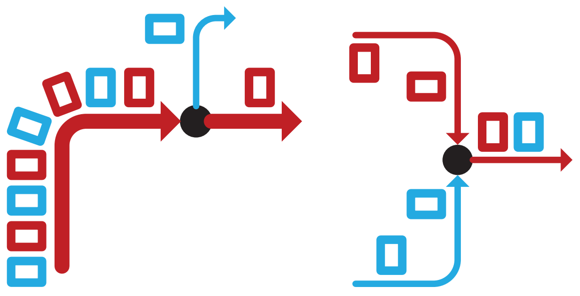

(a) Divert (b) Merge

Figure 17: Switching worst case scenario

In Figure 17a, the divert has incoming baggage, alternating between the two outgoing edges. This means the divert has to switch from its current direction to the other direction each time a bag arrives.

In Figure 17b, incoming baggage over both incoming edges is just far enough apart that each time a new bag arrives, it will be over the other ingoing edge as opposed to the edge over which the previous bag arrived. So the merge will have to switch over to the other bag in order to process it.

The performance of such switching situations will be optimal if only one ingoing edge (for diverts) or one outgoing edge (for merges) is being used. This would result in no switching whatsoever, thus reducing the amount of switching time to zero. This relates directly to the KPI described in Section 2.3.1.

2.7.3 *Severity of switching*

Not only do we want to know what switching bottlenecks we have, but the severity of such a bottleneck also plays a role. Additional to the capacity of an upper bound of bags per hour which the edge can process, we also have the amount of bags which can (at one time) reside on the edge, which is defined as the maximum holding capacity. Where the capacity as Vanderlande has defined it is more like a throughput, this maximum holding capacity is the actual number of bags which can physically stay on the edge should the edge break down.

This relates to the impact a switching operation has on the system. When a divert needs to switch a lot, we can look at the maximum holding capacities of the incoming edges. When these are very large, the impact of such a high switching point is less severe than when the edges cannot hold that many bags. For example, we have a divert, where one edge is the main loop, and the other leads to a screening machine. When this divert needs to switch a lot, and the edge to the screening machine has a maximum holding capacity of 3, the divert does not have a lot of leeway before the edge to the screening machine is full and bags are forced to skip the exit and travel along the main loop. In such a situation, the impact on the system of a divert which needs to switch a lot is much higher.

2.7.4 *Amount of crossing flows*

Related to reducing the amount of switches, we also want to prevent the merging of large flows as much as possible, as was given the example in Section 2.6.3. This is a cumulation of all previous performance measurements, because the impact of the crossing of main flows is the highest. This, due to the large amount of necessary switches, the probability that the resulting merge will exceed capacity, and that the ingoing edges do not have a high maximum holding capacity. I.e. on all previous three performance measurements, the probability that those will be affected here is high.

2.8 SUMMARY

In this chapter, we have defined the problem which we are going to solve. We have shown that we want to generate routing logic, such that baggage can traverse in a BHS from point A to point B. We have defined this routing logic as a configuration for the LM, and call this configuration a *set of routing rules*.

We introduced the Key Performance Indicators (KPIs) Vanderlande uses in their BHSs, of which one KPI is the most important: the amount of bags delayed to such an extent that they miss their flight. Because the delay of a bag can have multiple causes, we explained two other KPIs which more directly relate to the delay of bags: *switching* and *exceeding of capacity*. Switching is the amount of mechanical operations a divert or merge has to make in order to process bags. Worst case a divert or merge has to constantly alternate between its possible switching positions. Each edge in the system also has a capacity, which is the amount of bags per hour that can be processed over that edge. When we exceed that capacity, the edge will be disabled and rerouting is necessary; this results in other edges being more stressed, as well as other diverts and merges having to switch more often.

We defined the main *goal* of this graduation project, which is as follows:

> To develop a generation method capable of generating for a BHS a valid and good set of routing rules, consisting of routing rules for each divert towards all reachable destinations.

We indicated what the properties of a BHS is, to which we need to make sure our generated set of routing rules adhere. We introduced the *task topology*, which enables the LM to determine what the possible destinations are to which a bag can be routed, given an certain position in the system.

We have shown multiple interesting scenarios, to which we pay special attention throughout the rest of this thesis due to their complexity. These interesting scenarios consist of:

- Preventing the use of redundant paths which will result in unwanted merges;
- Using redundant paths which could lead to spreading the load on the system;

- Preventing crossing main flows which could result in severe bottlenecks.

Finally, we explained how a set of routing rules could be measured performance-wise. These measurements are a direct result of the KPIs, and they amount to:

- Amount of capacity bottlenecks; the number of times an edge exceeds the number of bags per hour it can ultimately process;
- Amount of switching; the number of times a divert or merge has to switch between one of its switching positions;
- Severity of switching; the relation between the amount of switches and the impact of a high amount of switching, due to the maximum number of bags an edge can physically hold when it is disabled;
- Amount of crossing flows; a cumulation of the previous three measurements, with the highest impact on the performance of the entire BHS.

3 APPROACH

In this chapter, we will discuss our approach for solving the problems as discussed in Chapter 2. As a foundation for this approach, we will introduce a framework which divides the approach in three distinct processes. Subsequently, a graph definition is presented, which will be used throughout this thesis. Using the framework as a basis, we will describe each of the processes, explaining their contribution to solving the project goal. First, we create an initial set of routing rules, followed by an analysis method, and finally an optimization method for finetuning of the routing rules.

3.1 FRAMEWORK

In order to generate a valid and good set of routing rules, as was defined in Section 2.1, we introduce a framework to do so, displayed in Figure 18. We can separate the goal in two distinct parts: the creation of a valid set of initial routing rules, and the creation of a good set of routing rules. This distinction can also be seen in the framework.

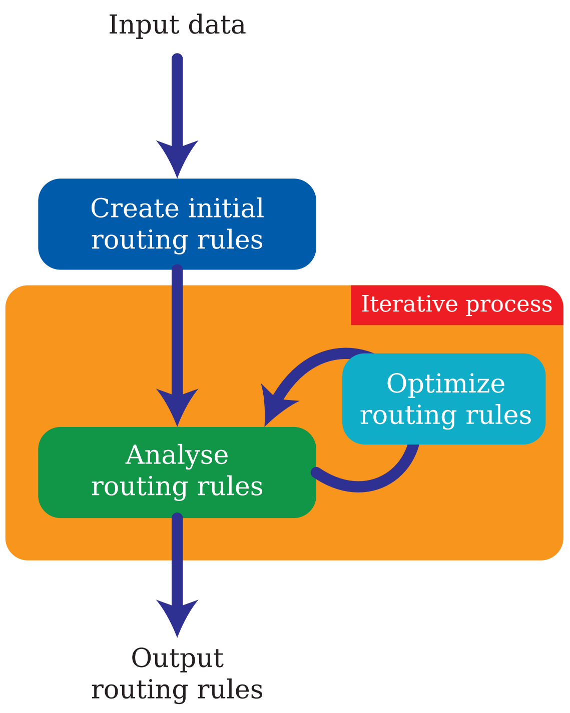


Figure 18: Framework used for reaching the project goal

We have a set of input data, which contains all relevant data of the BHS for us to work on. With this data, we create an initial set of routing rules. This initial creation aspect will be concerned with generating a valid set of routing rules. We will analyse the performance of this set of routing rules separately. The result of this analysis could indicate that there are aspects of the set of routing rules which are applicable for improvement. In order to achieve these improvements, we will modify the routing rules in order to achieve these improvements. The modified set of routing rules will again be analysed in order to check if improvements were actually achieved. As a result, this is an iterative process: after analysing a set of routing rules, we try to optimize the routing rules, reanalyse to check the performance, possibly modify again, and so on.

In the initial creation aspect, we focus on single bags. We are not interested in whether or not the routing rules we are creating are feasible. We are only interested in creating a complete set of rules which makes it possible for all baggage to reach their destination regardless of the situation. In the iterative analysis and optimization aspects, we look at the BHS as a whole, and try to modify the rules such that they are more optimal for the entire system. Here, we do take into account capacities and the interesting scenarios which we described in Section 2.6.

In this presented framework, we have three distinct aspects in the process of generating a valid and good set of routing rules:

- Creating a valid set of routing rules (creation algorithm);
- Analysing this (or a) set of routing rules (analysis algorithm);
- Improving the performance by modifying the routing rules and analysing them again (optimization algorithm).

In this chapter, we will show our solution for each of these aspects. We talk about the creation of a valid set of routing rules in Section 3.3, the analysis of a set of routing rules in Section 3.4, and finally the optimization of these rules in Section 3.5.

3.2 GRAPH DEFINITION

Before we can actually solve the problems, we first have to define a graph to which we can map a BHS. In Section 1.2, we already mentioned that a BHS can be represented as a graph. As such, we define a directed graph $G = (V, E)$, whereas V is the set of nodes, and $E \subseteq V \times V$ is the set of edges between these nodes. In terms of a BHS, one should think of the nodes as important parts in the system where something can actually happen to a bag (baggage scanners, diverts, manual coding, etc.) and the edges as conveyor belts. Though, it is not necessarily the case that a single edge represents a single conveyor belt, it can also represent multiple (connected) conveyor belts or even parts of a single conveyor belt.

An edge $e \in E$ is defined as $e = (v, w)$ where $v, w \in V$. Thus, we have a directed edge e which starts in v and ends in w. In order to more easily talk about the start and end node of an edge we define functions $\sigma, \tau : E \to V$, such that $\sigma((v, w)) = v$ and $\tau((v, w)) = w$. Self-loops are not possible, so $\forall(v, w) \in E : v \neq w$. For each $v \in V$, $\delta^-(v)$ denotes the indegree of v, $\delta^+(v)$ the outdegree of v; the set of ingoing edges and outgoing edges, respectively. The number of edges in these sets are depicted as $|\delta^-(v)|$ or $|\delta^+(v)|$.

In the set V, we define special nodes, which play a vital role in a BHS. We define the set of destination nodes $T \subseteq V$, which are nodes in V to which bags can be explicitly directed. At any time in a BHS, a bag always has a destination $t \in T$ to which it will be routed. A destination is not necessarily a final destination, i.e. the bag does not necessarily leave the system after it has arrived at such a t. A t can also be an intermediary destination, thus it is not necessarily the case that $\delta^+(t) = 0$. We do however state that $\forall t \in T : \delta^-(t) \geqslant 1$.

For the routing logic, diverts are decision points in a BHS. Because of this, we define a special set of divert nodes $D = \{d \in V \mid \delta^+(d) \geqslant 2\}$, i.e. each divert node has two or more outgoing segments. Because diverts cannot be destinations, we can say that $D \cap T = \emptyset$.

Merging also plays an important role when considering combining large flows, and thus we want to indicate these as well. We do this by defining a set of merge nodes $M = \{m \in V \mid \delta^-(m) \geqslant 2\}$, which is symmetrical to D. No node can be both a divert and a merge, thus $D \cap M = \emptyset$. Destinations cannot be merges as well, $T \cap M = \emptyset$.

Two edges are consecutive (denoted by $\twoheadrightarrow$) if the end-node of one edge matches the start-node of the other edge: $\forall e_0, e_1 \in E : e_0 \twoheadrightarrow e_1 \Rightarrow \tau(e_0) = \sigma(e_1)$. We define a path p, which is a sequence of consecutive edges $p = \{e_0, \ldots, e_{n-1}\}$. By this definition, it is possible to traverse the path from $\sigma(e_0)$ to $\tau(e_{n-1})$ without going against the direction of the edges. Like with edges, we define functions $\sigma_p, \tau_p : \mathbb{P}(E) \to E$, where σ_p indicates the first edge in a path p, and τ_p the last edge in a path p. In the given example p, $\sigma_p(p) = e_0$ and $\tau_p(p) = e_{n-1}$. Cycles are possible in a path, so edges can occur multiple times in a path. The length of the path is defined by $|p|$, which is denoted by the number of edges p contains. Also, we denote a function *distance* $d(v, w)$ where $v, w \in V$, which denotes the length of the shortest path from v to w. If there is no path available, $d(v, w) = \infty$.

Additionally, we introduce multiple edge functions in order to perform graph operations later on. We define two functions $c, tt : E \to \mathbb{R}^+$. Here, c indicates the capacity of a single edge, i.e. the maximum number of bags per hour an edge can process, and tt indicates the travel time of a single edge, i.e. the time in seconds it takes for a bag to traverse over the edge.

3.3 CREATING VALID ROUTING RULES

The first aspect in our framework is to create a set of valid routing rules. At this moment, we are primarily concerned with the validity of the set of routing rules, not with its performance. For this we use a graph as defined in Section 3.2. What we want is for every divert $d \in D$ to define a routing rule for each reachable destination $t \in T$. This means that for each edge $e \in \delta^+(d)$ we need to set a usage, being either Favourite, Substitute Favourite or Option, for the specific destination t.

3.3.1 *Calculating best path*

At first glance, the situation and problem very much tend towards applying a single-source shortest path algorithm. However, in a BHS, the shortest path is almost never the preferred path. The reasons for this will become apparent in this section. Although we cannot use a plain shortest path algorithm, we can use the shortest path algorithms that are available as a basis for our own algorithm. For our application we use Dijkstra's well-known single-source shortest path algorithm as the basis for our algorithm[Dijkstra, 1959; Korte and Vygen, 2000].

We choose Dijkstra's algorithm, due to its simplicity. It is a straightforward solution for a single-source shortest path problem, and is quite easily implemented and checked. Due to our graph being a sparse graph (where $|E| \approx |V|$), there are also some more benefits which contributed to our choice for (a modification of) Dijkstra's algorithm, but these will become more apparent in Chapter 4.

Since Dijkstra's single-source shortest path algorithm is common knowledge, we will only glance over the algorithm itself. For more detail, we refer to the paper of Dijkstra[Dijkstra, 1959]. We start in our source node. Every iteration, we take the node with the smallest cumulative cost to the source node, and expand from that node to all its neighbours. In the next iteration, we again take the node with the smallest cumulative cost to the source node, except we now also include the neighbour nodes in the search space. This way, we keep on expanding, until we have eventually reached the target node and we know that there is no shorter path possible.

3.3.2 *Cost function*

For the cost of an edge, we define a cost function $w : E \to \mathbb{R}$. With a shortest path approach, generally one has a predefined weight on an edge, for each edge in the graph. The shortest path is based on the cumulative cost of this weight. However, because we are not interested in the shortest path, we define our own cost function to be used with the algorithm. As the base for this cost function we take the travel time of an edge.

The travel time of an edge corresponds to the time a bag takes to get from the beginning of an edge to the end of it. Our reason for choosing this as the base of our cost function is because it is tightly coupled with our KPIs. Since we want to reduce the time the bag is in the system, time itself naturally plays a very important role in this. The travel time of an edge determines for a large amount the time a bag is in the system, and thus we will use this as the base for the cost function of an edge.

The resulting cost function is as follows, where use the travel time function introduces in Section 3.2:

$$w(e) = \mathrm{tt}(e) \tag{1}$$

However, the cost function as given in Equation (1) will result in nothing more than a fastest path calculation. There are several special cases in our graph which we want to take into account, and for this we need to augment our cost function. These special cases will be explained in the following sections.

3.3.2.1 *Prevent unwanted shortcuts*

Throughout the graph, we have destinations that are not necessarily the destination to which we want to route the bag. In the case of intermediary destinations (e.g. Screening, Manual Coding, not Laterals), we can still route baggage through those destinations in order to get to the destination of our bag. However, these intermediary destinations often have their own dedicated areas in a BHS, where the capacity is lower than the main loops in the system. This is, because these areas should only be used to route baggage towards one of those destinations, and thus we do not want bags in those areas that do not belong there.

An example on this scenario is given in Figure 19. Here, we have the snippet from the running example showing the Screening stations and part of the Store loops. Assume we have a bag which needs to be routed towards the Store. Due to our simplistic cost function shown in Equation (1), it is very well possible that the bag is routed through one of the Screening stations in order to get to the Store (shown in orange).

The result of this routing is that Screening stations receive bags that do not have to be there. The consequence of this is that bags that actually have to be at the Screening stations have to wait, because of this increased load on those stations. Thus, this is directly related to the capacity KPI as defined in Section 2.3.2.

The solution for this problem is to augment the cost function with a case where the cost is increased to infinity if we want to route through a station that is not our destination t, i.e. route over an edge $e \in E$

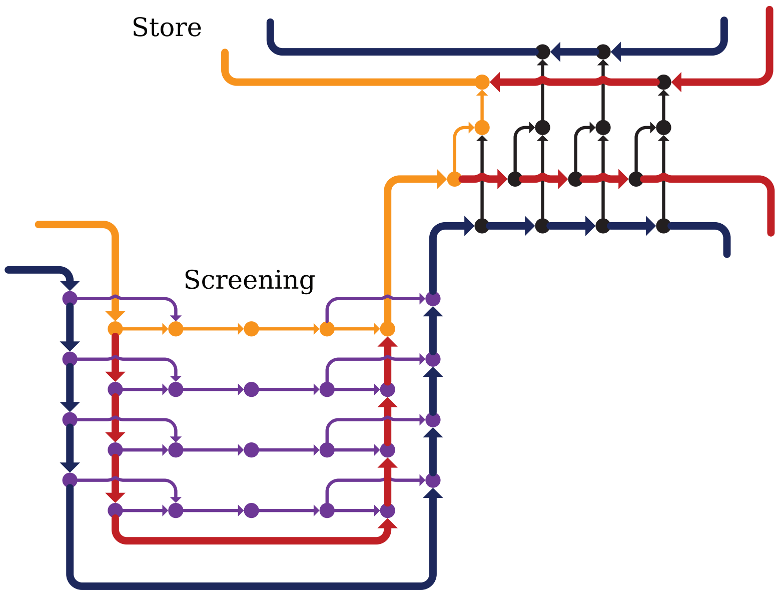


Figure 19: Unwanted routing through Screening

where $\tau(e) \in T \wedge \tau(e) \neq t$. The resulting augmented cost function is given in Equation (2)

$$w(e) = \begin{cases} \infty & \tau(e) \in T \wedge \tau(e) \neq t \\ tt & \text{else} \end{cases} \tag{2}$$

3.3.3 *Setting the usages*

Once we have calculated the best path from a specific divert d to a specific destination t, we can determine the usages for d. As a result, we mark the first edge of the best path as the Favourite for the route from d to t.

In Figure 20, we show a snippet from the running example, showing part of the output area. We have a divert d, shown in light blue, and a destination t, shown in pink. At d, we have two possible acyclic paths to get to t, either beginning with edge 105 or 102. Along the edges, their cost is given. When we look at both possible paths, we get a total cost of $4+6+9+15+2 = 36$ when starting with edge 105, and a total cost of $5+6+9+20+9+15+2 = 66$ when starting with edge 102. Thus, the path starting with edge 105 is the best path, so we will use that in order to define our routing rule.

As described in Section 1.4.4.1, we also have the possibility to indicate alternative paths. A divert has two or more outgoing edges, so once we have determined one of those edges as the Favourite edge, we should also take the remaining edges into account for alternative routing. Suppose we have edge $e_f \in \delta^+(d)$ marked as Favourite for t.

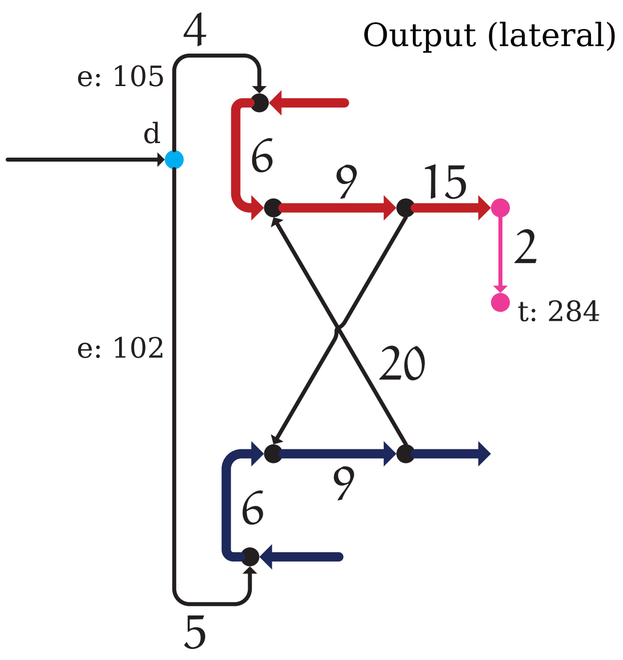


Figure 20: Best path calculation from d to t

We should then see if edges $e \in \delta^+(d) \setminus \{e_f\}$ are also capable of routing a bag towards t, i.e. is t also reachable through the other edges? If this is the case, we have edges on which we can apply usages for alternative routing.

There is a tradeoff between having an alternative path, and actually using one by setting a usage. This tradeoff is the total cost of alternative paths. We do not want to indicate alternative paths significantly longer than the best path, because the effect of switching to the alternative path in case the best path becomes unavailable could very well be negative. An analogy: one can travel from Amsterdam to Paris by train in a fairly straight manner. However, one can also travel from Amsterdam to Paris by first going through Moscow. Obviously, such a detour is unwanted due to the enormous increase in travel time. Similarly, we want to prevent such alternative paths here as well. We do this by setting a threshold on the increased total cost of alternative paths. If we have a path p, then $w(p) = \sum_{e \in p} w(e)$. For all alternative paths p', we indicate that they only qualify for setting a usage when $w(p') < 1.10 \cdot w(p_{\text{best}})$, i.e. when the total cost of the alternative path lies within 10 percent of the total cost of the best path. This percentage threshold can change per BHS, due to the nature of the BHS; as such, this will also become part of the user input. We intentionally do not choose for a K best paths approach[Yen, 1971; Eppstein, 1998], due to the possible large gap between the first and subsequent best paths. With a percentage approach, we can control how close the alternative paths lie in terms of cost. With a K best path approach this is also possible, but requires additional methodology to use these algorithms and thus unnecessarily complicates the overall methodology.

There are two possible usages for alternative edges, namely Substitute Favourite and Option. Assume we have an edge $e_r \in \delta^+(d) \setminus \{e_f\}$ through which t is reachable, then the usage on e_r will become:

- Substitute Favourite if e_r lies on a status edge which is a loop,
- or Option otherwise.

The rationale behind this distinction is due to the difference between the Substitute Favourite and Option usages, as was also explained in Section 1.4.4.1. Only the LM can determine routing over loops, as it cannot be left to a PLC. The reason for this is that loops are always running, and an isolated PLC does not have the overview of the entire loop, and thus cannot make an accurate decision on whether it is a good choice to route over the loop or not. The LM always has an overview of the entire BHS, and thus the LM can make such a decision. This is why we want to leave alternative routing over loops to the LM, and thus indicate those edges as Substitute Favourite. All other alternative paths that do not lie on a loop can receive the Option usage.

In our example with edges 102 and 105, the remaining edge 102 does not lie on a loop, and thus it could receive the Option usage. However, since the total cost of the alternative path does not adhere to our threshold of 10 percent ($36 \cdot 1.10 < 66$), we will not indicate the alternative path as such. The resulting routing rules are shown in Table 1. In order to show a more realistic set of routing rules, we have added multiple destinations for which edges 102 or 105 (or neither) are the Favourite starting point.

Table 1: Routing rules after best path calculation

		Destination							
		14	15	38	284	285	286	287	288
Edges	102	F	F				F		F
	105				F	F		F	

3.3.4 *Validity*

In Sections 2.1 and 2.4 we stated that we want our set of routing rules to be valid, and adherent to certain properties. Actually, the properties stated in Section 2.4 are not our primary concern when creating an initial set of routing rules. This, because they are mainly concerned with the actual performance of a set of routing rules; how well the system is able to process baggage with the given set of routing rules. With the creation algorithm, we are primarily interested in creating a valid set of routing rules and, as said in Section 2.2, to operate according to the worst case and maximize our coverage.

In order to prove that the creation algorithm generates a valid set of routing rules, we need to make some assumptions about the properties of any BHS and therefore about the graph representation of those BHSs:

1. The topology is complete, and there are no unreachable areas from any point in the graph which should be reachable;
2. Edges in the graph have a finite travel time;
3. All destinations are reachable through non-destination nodes from any node in the system (both destinations as non-destinations).

These assumptions are made on the functionality of the BHS, and as such do not belong to the scope of this project. If a BHS does not adhere to these properties, the system is in itself dysfunctional and generating a valid set of routing rules could be impossible.

Our definition of a valid set of routing rules was that each bag should be able to reach its destination. This can also be interpreted as: at each point in the system, we should be able to get to all possible destinations. We intentionally talk about possible destinations, because due to the nature of the BHS we cannot reach all areas from every other area in the BHS. When we want to prove the validity of our created set of routing rules, we want to prove that we have generated routing rules for each possible tuple of edges and destinations.

Trivially, every edge not coming out of a divert already enables bags to reach all destinations. This, because nodes connected to those edges are either simple nodes that can only propagate bags in one way, or they are merges and they cannot actively change the direction of bags at all.

This leaves us with proving that all diverts enable bags to reach all possible destinations. We are using a modification of Dijkstra's single-source shortest path algorithm, and it has already been proven that Dijkstra's algorithm terminates and if there is a shortest path, it will find it[Korte and Vygen, 2000]. Our modified cost function incorporates travel time, and with assumption Case 2, we know that if there is a best path, it will not have an infinite cost due to all edges in the path having a cost lower than infinity. However, we raise the cost to infinity ourselves whenever we reach a destination that is not the destination towards which we want to route. This is negated by assumption Case 3: if there is at least one path possible from a divert to a possible destination, at least one of those paths will not contain any destination nodes. So whenever our modified Dijkstra's algorithm finds a best path, it will be a valid one.

Now, since we run our modified Dijkstra's algorithm for each tuple of diverts and destinations $(d, t), d \in D \wedge t \in T$, and with our assumption Case 1, we know that if there is a best path between divert d and destination t, our algorithm will find it and associate a routing rule with it. Thus, for each divert we will create a routing rule for each possible destination, and as such we will validate that at any point in the system a bag can reach any of its possible destinations.

3.4 ANALYSIS OF ROUTING RULES

Generating a set of routing rules without ever checking its performance is obviously undesirable. We want to know what the effects of a set of routing rules on a BHS are, using the performance measurements as described in Section 2.7. At this moment, full analysis of routing rules at Vanderlande is done using extensive simulations. These simulations are expensive, both in time and cost, which is why we want to eliminate errors in the routing rules as early on as possible. Thus, when we can indicate the performance of a set of routing rules early on in the design process, less errors will be encountered later on. As such, time and costs can be saved during the simulation phase at the end of the entire design process.

So, we need a way to approach these full fledged simulations, without actually performing them. A simulation is done by taking into account the entire BHS, and simulating thousands of bags inside the BHS. Each bag is tracked by the simulation, and every important variable that is part of the system is taken into account. The costly nature of simulations lies in the fact that they have to be performed real-time. Simulations are not sped up, in order to accurately simulate the entire BHS. This means that time alone is a very important factor for us to save when eliminating as much errors as possible, early on in the design process. Additionally, a simulation is heavy performance-wise, because every single bag in the system has to be accurately simulated. This means that, while keeping track of each and every single bag in a flow of thousands of bags, simulations have to be performed on multiple servers, dedicated to the calculations of these simulations. We, however, can abstract from these simulations by not taking into account every single bag, but larger groups of bags.

The idea is to use flow algorithms to perform more crude simulations compared to the actual simulations. These flow algorithms take into account larger groups of bags, and by assuming a worst case scenario during the operation of the system, we can indicate whether we have conflicting flows or bottlenecks in the system. Of course, it is impossible to have an airtight proof on the performance of a set of rules without doing full fledged simulations. With these more crude simulations, we intend to give insight early on in the process about the performance of a set of routing rules, and feed information back into the creation algorithm which can adjust its parameters in order to circumvent the found bottlenecks. This is also shown in the framework in Figure 18, at the start of this chapter. Additionally, the end user can take the given information into account, and make decisions based on this information. These could for example be changes to the routing rules requiring human insight, or intentional changes that are suboptimal, but with a predefined reason.

In comparison with the full fledged simulation Vanderlande performs, our simulation will lose some precision. This is mainly due to the assumptions we make throughout this project, and the loss of physical properties of the system. We only regard the most major influences the physical system has on the actual routing of baggage, whereas a Vanderlande simulation takes all these properties into account. However, the results from our simulations should be at least close to the results a full simulation by Vanderlande outputs.

3.4.1 *Flow*

Throughout this thesis, we have been sporadically dropping the term *flow*. We used the term in a more natural sense, in order to talk about entire groups of bags flowing through the system. Due to the BHS behaving very much like a network, where almost all bags move from one direction to the other (and seldom backwards), the formal term flow from graph theory quickly springs to mind. Describing this flow of baggage by making use of the appropriate flow algorithms is one solution which we will now explore.

The aspect of flow throughout a network is a field of graph theory which always has been of much interest to researchers. One of the most researched topics are those concerned with calculating the maximum possible flow throughout a network, where the flow adheres to the capacity of the network[Ford and Fulkerson, 1955; Edmonds and Karp, 1972; Denic, 1970]. Another topic is an extension of the maximum flow problem, where we are also concerned with flow taking the least costly paths inside the network, the so called minimum-cost network flow problem. The network is augmented with weighted edges, and solutions for this problem often take into account some shortest path algorithm[Fulkerson, 1961; Ford and Fulkerson, 1962; Goldberg and Tarjan, 1990].

We will briefly touch upon these two main problems, describe the problems and their solutions in a generic manner, and follow up with their applicability for this project.

3.4.1.1 *Maximum flow problem*

The maximum flow problem is a thoroughly researched topic in the field of graph theory, with solutions dating back as early as 1955[Ford and Fulkerson, 1955]. We have a directed graph $G = (V, E)$, which is augmented to a network by adding two special nodes, source α and sink ω, thus $V \leftarrow V \cup \{\alpha, \omega\}$. The notion of source node α and sink node ω will be used consistently throughout the rest of this thesis.

To explain the idea of flow throughout a network, we use an analogy; that of a network of pipes through which water flows. Water has to come from a single point, which is the source α. After it flows through the system, it needs to exit at some point. There can be multiple exits, but for simplicity we assume we have a single exit: the sink ω. Each water pipe has a certain amount of water it can process. It can hold water up to this amount, after which it is completely filled and no more water can enter the pipe. This upper bound on the amount of water is the capacity of the pipe. The flow in this network of pipes is the amount of water that can move from α to ω.

Now, if we map this back to our actual network G, we have a source of flow α, a sink ω where flow terminates, and edges $e \in E$ which have a capacity $c : E \to \mathbb{R}^+$, and a flow $f : E \to \mathbb{R}^+$. The solution for the maximum flow problem is to create a total amount of flow N in G, from α to ω, adhering to the capacity of each edge in E, such that N is maximum. Also, we want to impose that for each $v \in V$ (except α and ω) the ingoing flow is equal to the outgoing flow. More formally:

$$\begin{aligned}
&\text{Maximize } N = \sum (f(e) : e \in E) \\
&\text{subject to} \\
&f(e) \geqslant 0, \forall e \in E \\
&\sum (f(e) : e \in \delta^-(v)) = \sum (f(e) : e \in \delta^+(v)), \forall v \in V \setminus \{\alpha, \omega\}
\end{aligned}$$

Throughout the years, many solutions have been given for this problem. Each solution lowered the running time, or presented a solution for special graphs (for example, sparse graphs[Sleator, 1981; Gabow, 1985], or for implementing parallel calculations[Shiloach and Vishkin, 1982]).

3.4.1.2 *Minimum-cost flow problem*

An extension of the maximum flow problem is that of the minimum-cost flow problem. Here, in addition to the given graph G, we also have a cost function $w' : E \to \mathbb{R}$. Each edge in E has a certain cost for each unit of flow passing through it. We want to minimize the total cost of a given flow throughout G. More formally:

$$\begin{aligned}
&\text{Minimize } \sum (f(e)w'(e) : e \in E) \\
&\text{subject to} \\
&f(e) \geqslant 0, \forall e \in E \\
&w(e) \geqslant 0, \forall e \in E \\
&\sum (f(e) : e \in \delta^-(v)) = \sum (f(e) : e \in \delta^+(v)), \forall v \in V \setminus \{s, t\}
\end{aligned}$$

Close observers can see that this is indeed an extension of the maximum flow problem, or rather, the maximum flow problem is a subclass of this problem. This, because if we set the weights for each edge to 1, we exactly have the maximum flow problem.

3.4.1.3 *Applicability*

We now look at the applicability for analysing a set of routing rules, using one of the two main flow problems explained in Sections 3.4.1.1 and 3.4.1.2.

An initial set of routing rules has already been created using a best path algorithm (see Section 3.3). This means that the aspect of minimizing the cost of a flow has already been done throughout this best path algorithm. Using the minimum-cost flow problem as a basis for the analysis of a set of routing rules would indicate that we take the aspect of cost into account twice. Also, we want to have a separation of concern between generating and analysing a set of routing rules. Both processes come with their own set of demands according to what output they should deliver, so separating these would increase the understandability of both approaches. Most importantly, if we separate both processes, we can run them separately. This way, our analysis algorithm can be applied on already existing routing rules (those currently operating on existing BHSs), and indicate potential bottlenecks in those rules.

So, with these arguments, we can cross off the minimum-cost flow problem, and apply the maximum flow problem for our analysis.

3.4.2 *Push-relabel maximum flow algorithm*

To solve the maximum flow problem for our analysis, we use Goldberg and Tarjan's push-relabel flow algorithm[Goldberg and Tarjan, 1988]. Their algorithm is currently the fastest algorithm with a running time of $O(nm\log(n^2/m))$ using a dynamic trees data structure[Sleator and Tarjan, 1983, 1985]. Aside from the running time, the elegance and simplicity of their algorithm also plays an important role for the choice of using it in our analysis. It is quite an intuitive algorithm, which we will explain in this section. For a more detailed explanation, with proofs of correctness and termination, we refer to the paper itself.

3.4.2.1 *Flow and preflow*

The authors use Karzanov's notion of *preflow*[Karzanov, 1974]. Before we can explain what preflow is, we first need to give the definition of flow:

$$f(v,w) \leqslant c(v,w) \qquad ,\forall(v,w) \in E \tag{3}$$

$$f(v,w) = -f(w,v) \qquad ,\forall(v,w) \in E \tag{4}$$

$$\sum_{u \in V} f(u,v) = 0 \qquad ,\forall v \in V \setminus \{\alpha, \omega\} \tag{5}$$

Here, we have in Equation (3) the capacity constraint, in Equation (4) the antisymmetry constraint and in Equation (5) the flow conservation constraint. Equations (3) and (5) were already given, although in some other form, in Section 3.4.1.1.

3.4.2.2 *Relaxation of constraint*

The authors extend the capacity function with the notion that $c(v,w) = 0$ for all $(v,w) \notin E$. The definition of preflow is similar to that of flow, but with the flow conservation constraint in Equation (5) weakened:

$$\sum_{u \in V} f(u,v) \geqslant 0 \qquad ,\forall v \in V \setminus \{\alpha, \omega\} \tag{6}$$

This means that we no longer have the constraint that the ingoing flow of a node must be equal to the outgoing flow of a node. Now, the ingoing flow can be larger than the outgoing flow. The authors define a *flow excess* function indicating the total value of the ingoing flow minus the total value of the outgoing flow, i.e. the amount of flow which 'overflows' the node.

3.4.2.3 *Pushing flow*

The idea behind the push-relabel algorithm is that we take into account all nodes which still have excess flow, and *push* that flow towards sink ω. If there is no possibility of pushing excess flow towards ω, excess is pushed back towards source α. Eventually, there will be no more nodes with an excess flow; the network has then reached a state where a normal flow is present. Even more, that normal flow will then be the maximum flow.

In order to move flow throughout the network, we need to know when we can push flow, and if so, how much flow. The authors introduce the notion of *residual capacity*, which is nothing more than the amount of flow that still can be pushed over a certain edge: $r_f(v,w) = c(v,w) - f(v,w)$. Whenever we have a node with excess flow, and it has an edge e with $r_f(e) > 0$, we can push flow over e in order to reduce the excess flow at v.

Additionally, we need to make sure we are pushing flow towards ω. The authors do this by applying a *label* to each node $v \in V$, where

$d(v, \omega)$ is set as the label. Initially, the labeling can be set to $d(\alpha) = |V|$, $d(\omega) = 0$ and $d(v) = 0$.

3.4.2.4 *Pushing and relabeling*

The push-relabel algorithm received its name due to the two main operations the algorithm consists of: pushing, and relabeling. The algorithm keeps running whilst there are still *active* nodes. A node v is active when $v \in V \setminus \{\alpha, \omega\} \wedge d(v) < \infty \wedge e(v) > 0$, i.e. a node which is not the source or sink of the graph, which has a valid labeling and still has excess flow. The algorithm will keep on applying push and relabel operations, until the set of active nodes is empty.

When the algorithm applies a push operation on the edge (v, w), it increases $f(v, w)$ and $e(w)$ by $\delta = \min(e(v), r_f(v, w))$, and decreases $f(w, v)$ and $e(v)$ by the same δ. So we push either all excess we have, or only that which the capacity of the edge still allows us to. The resulting excess will eventually be pushed back towards the source.

When the algorithm applies a relabel operation on node v, it sets the label of v to the minimum label of all its outgoing neighbouring nodes, plus one: $d(v) \leftarrow \min\{d(w) + 1 \mid (v, w) \in \delta^+(v) \wedge r_f((v, w)) > 0\}$. As mentioned, the relabeling is necessary to know whether we are pushing towards ω or towards α. If we again use the analogy of water, one can see the labeling as pushing from a high placed node to a lower placed node, by using gravity. It will then only be possible to push 'downwards', i.e. by making use of the gravity, to lower labeled nodes.

3.4.3 *Maximum overflow*

With the push-relabel algorithm as explained in Section 3.4.2, we can calculate the maximum flow in our graph G. This is, however, too restrictive for the information we want. When we want to analyse the performance of a set of routing rules, we want to check its performance under a worst case scenario. This means that all input points of the graph will operate under (near-)maximum capacity. Whenever we calculate a maximum flow, what we in essence do is calculating what the maximum flow is that the BHS can handle, and adjust the flow of the input points accordingly (because, outgoing flow of α must equal ingoing flow of ω). This is contradictory when looking at the real-life scenario, where the input of a BHS is one thing that cannot easily be turned off, especially in a worst case scenario.

What we do want to calculate, is to set the flow of all input points at their maximum capacity (or close to it), and push flow through the entire system. The input flow has to be seen as an unstoppable force, and it can only exit at ω. We do not want to adhere to the systems capacity (at least, for the analysis). We want to 'drown' the system in its input flow and see if and where the bottlenecks are present. Thus, we talk about *maximum overflow* instead of maximum flow.

3.4.4 *Modified push-relabel algorithm*

The push-relabel algorithm calculates the maximum flow of a network, but with some modifications, we can calculate the maximum overflow of a network. These modifications are relaxations on some operations and/or constraints of the algorithm. This, because we remove an important constraint: we do not want the flow to adhere to the capacity of the BHS. Only if we ignore the capacity, we can push high amounts of flow through low capacity parts of the system, and thus we can later on indicate that this would be a problem if the analysed set of routing rules were to be used in the real BHS.

We no longer have need for the capacity constraint (Equation (3)), because we do not want to adhere to the capacity of the system. So, for our relaxation, we remove this constraint entirely. This means that flow on an edge can be higher than the capacity, which will be an indication that we have a bottleneck (see Section 2.3.2). Since we have removed this constraint, we also no longer need the notion of residual capacity, because the capacity in essence has become infinity. This means that it will always be possible to push flow over an edge, because $r_f(v, w) = \infty - f(v, w) = \infty$.

In the original algorithm, as soon as $\forall e \in \delta^+(v) : r_f(e) = 0$ we would push flow back towards source α because we could no longer push flow towards sink ω. The removal of the residual capacity (actually, the residual capacity always being ∞) has made it possible to always push flow towards ω.

3.4.5 *Steering flow*

In its current state, the analysis algorithm is capable of simulating an 'unstoppable' input flow, which needs to be pushed throughout the entire BHS. However, the flow itself is uncontrolled. Whenever flow arrives at a divert, it is divided over all outgoing edges, disregarding the destinations to which flow (or in our case, baggage) needs to travel. This means that flow is sent to otherwise impossible destinations, and the routing rules which we have created as a result of the algorithm described in Section 3.3 are completely ignored. In its current state, the analysis algorithm is only capable of analysing the topology of a BHS, ignoring any routing defined on that BHS.

After some research, we did not find any solutions or algorithms capable of directing flow. This means that we have to modify the algorithm to be able to cope with directing flow. What we want to achieve, is to steer the flow such that it will take into account the routing rules which we want to analyse. So, we need a mechanism which divides the flow at a divert, according to the routing rules (and, as appears later on, the task topology). The solution for this is two-fold. We use the main flow of baggage, as is shown in Figure 2, alongside with a combination of routing rules and the task topology.

In the following sections, we will explain how we will steer flow according to the routing rules. We will discuss each topic separately, and in the end we will provide an overall example showing a combination of all discussed topics.

3.4.5.1 *Main flow*

Each BHS is different, and as of such, extracting the main flow out of such a BHS is not straightforward. Due to it not belonging to the actual generation of routing rules, we leave this out of the scope of this project. Because of this, we define a user input which indicates what the main flow in a BHS is. Each major area of the BHS will be indicated, as well as the connections between these areas. An example is given in Figure 21, here we can see that we have process flow similar to Figure 2. We define a set of main flow nodes V_m and a set of main flow edges E_m whose functionality is identical to those in the graph G definition described in Section 3.2. The areas in a BHS can be seen as main flow nodes with tasks, and the main flow edges indicate the subsequent tasks that are possible after a task has been completed.

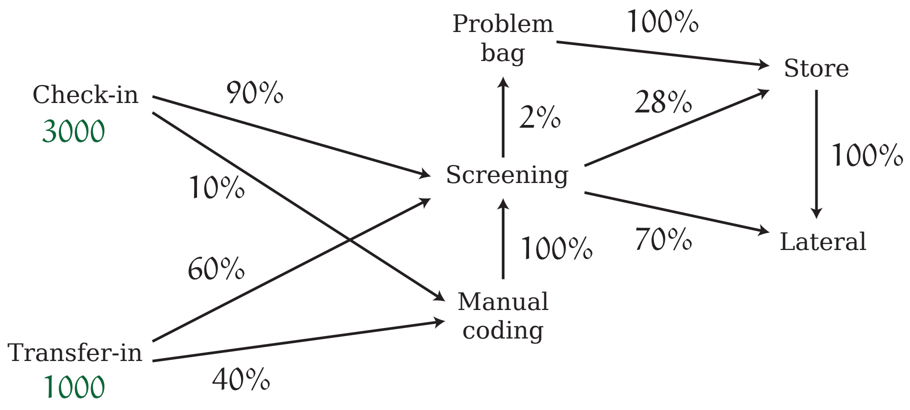


Figure 21: Example of user input on flow through main process

By using this main flow, we can analyse each single main flow edge one at a time. For example, we know (through the main flow) that of the 3000 bags arriving at the check-in input points, 90% continues towards Screening. Then, we can calculate how many bags this is (2700), set up the flow algorithm for this, and push this amount through the graph.

If we calculate the maximum overflow for each main flow edge, we are only interested in the two main flow nodes connected through this main flow edge. This means that we can create a subgraph out of G, using these two main flow nodes. If we, for example, want to calculate the maximum overflow from Manual Coding to Screening, we only need to take into account parts of G that make it possible for bags to travel from all Manual Coding stations to all Screening stations. The rest of the graph is irrelevant, and thus we can leave it out.

Using subgraphs for each single main flow edge cuts back the calculation time of our algorithms, because the subgraph can be much smaller than the whole graph G. If we now calculate, for each main flow edge, the maximum overflow between two main flow nodes in the main flow, we can merge all these maximum overflow subgraphs into one single maximum overflow graph. The result is again the entire graph, with one maximum overflow which holds for the entire main flow.

Not all BHSs will use the exact same distribution as shown in Figure 21. Though, there will be common elements almost every BHS will need, like the check-in and screening areas. Thus, the core of a distribution can be identical in many BHSs. However, due to the simplicity of a distribution, they can be interchangeable between BHSs, and it is not necessary for a distribution to cover all areas of a specific BHS. If one wants to test a specific part of a BHS, he can leave out all irrelevant parts in the distribution, and the algorithms will be able to cope with such a minimal distribution.

3.4.5.2 *Destination of flow*

Our graph G has been augmented to a network by adding source α and sink ω. As such, sink ω serves as a termination node for all flow in the graph. This means that all nodes that are connected to ω are the actual end points of the graph. Not each end point will be used equally, because of the use of a main flow distribution. If we start with 10000 bags at the check-in nodes, and we only send 10% towards Manual Coding compared to 90% towards Screening, we need some way to direct only 10% of the flow towards Manual Coding, instead of equally dividing all flow over Screening and Manual Coding. Manual Coding also consists of multiple stations, so we also need to divide the 10% of the flow over these stations, and for this we use the task topology.

As was described in Section 2.5, we can retrieve all possible destinations given a task and an edge in the BHS. When we create flow at each of the input points of a graph, we determine what destinations can possibly be handed out by the PC. Subsequently, we divide flow over all these possible destinations.

3.4.5.3 *Division of flow*

What remains is to steer all flow at every divert. This means that we have to keep track of flow, and where it needs to go. We can no longer have one generic amount of flow which spreads itself throughout the graph. Instead, from the moment we create flow, we need to know where it goes, and how it needs to get there.

We solve this by augmenting the notion of flow: introducing *destination flow* $f_t \in \mathbb{R}^+ \times T$. In the basic algorithm, we push real-valued flow throughout the graph, which is indistinguishable. For example: at merges, two incoming flows are merged, but these consist only of real-valued units, thus we can no longer distinguish them from each other once they have been added up. This makes routing towards a specific destination impossible. So, with our destination flows, we combine the real-valued unit with a destination.

Each f_t contains a certain amount of flow, which is still real-valued. However, we add a destination $t \in T$. Instead of pushing singular units of flow, we push tuples of destination flows through the graph. Whenever a destination flow f_t needs to be pushed from a divert $d \in D$ there are two possibilities, given the routing rules for d:

1. There is only one edge $e \in \delta^+(d)$ which leads to ω;
2. There are two or more edges $e \in \delta^+(d)$ which lead to ω.

With Case 1, we do not have any choice other than to push f_t over the single available edge e. With Case 2, we need to divide the destination flow over all possible edges. Assume we have n possible edges which lead to ω. If we have, for example, an f_t where the amount of flow is 18, then we create n new destination flows f'_t, and each f'_t will have a flow equal to $18/n$. All f'_t will still have the same destination ω. Now, we can divide each f'_t over each n outgoing edge, over which ω is reachable.

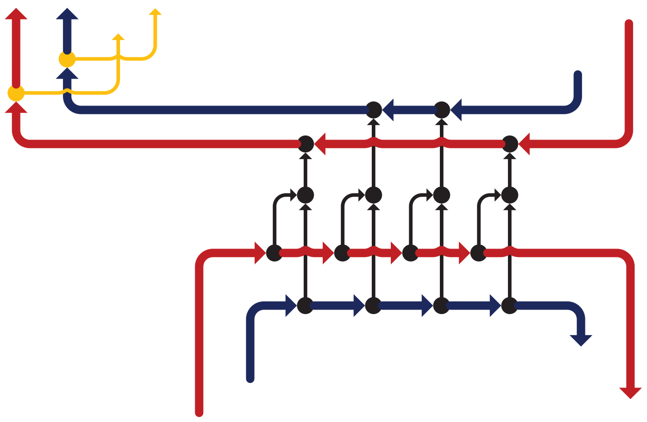


Figure 22: Load balancing when dividing flow

A special case when we have to split up destination flows, is when we have multiple possible routes which are all eligible for load balancing. This is only the case when we have a loop, and multiple routes starting from this loop can all lead to the same destination. An example is shown in Figure 22. Here, we have a snippet from the running example where we have baggage going from either of the two main loops to the Store loops. As can be seen, there are multiple paths leading to the Store loops. This is a practical version of the (interesting scenario) balancing problem shown in Section 2.6.2.

In this case, there are four parallel paths all eligible for load balancing. So, we divide a single destination flow f_t in four new destination flows f_t', all with the same destination. Still, a destination flow needs to know which path it needs to take, because up to now it only knew its end destination, and relied on the routing rules in order to get there. Our solution is to include a 'pass through' divert $d_\nu \in D$ in the destination flow. The destination flow has to travel through d_ν in order to get to its destination, and has to disregard any other diverts that can possibly lead to the destination. This way, we can force destination flows to be only pushed over diverts which we indicate, thus forcing the load balancing. After it has traveled through d_ν, d_ν is reset and the destination flow can continue along its way using the routing rules.

We intentionally apply load balancing in our analysis algorithm, and not during the initial creation of the routing rules. Due to the creation algorithm being a modification of Dijkstra's shortest path algorithm, we could use K-shortest path algorithms in order to determine the $K-1$ best paths[Yen, 1971; Eppstein, 1998]. Through these K best paths, we could check if there are multiple paths leading to the same destination, and use some of them for load balancing. However, there are two main arguments against this approach.

The first argument is that of separation of concern. As we said, we want to use the creation algorithm to calculate all valid paths from any divert to any reachable destination. If we are also taking into account load balancing, we are shifting this concern towards generating a good set of routing rules as well, which is the main concern of the analysis algorithm.

The second argument is that of simplicity. During our analysis, we are already taking into account several performance measurements, and load balancing is part of this. It would be contra-intuitive to take something which fits very well inside this aspect, and move it to a separate aspect.

3.4.5.4 *Combined example*

We shall now provide a single example in which we will show the practical result of the theory explained above. Assume we want to calculate the maximum overflow from Manual Coding to Screening, which can be seen in Figure 21. For this, we will create a subgraph from the running example, in which we take all possible input points (all Manual Coding stations) and all possible output points (all Screening stations). We remove the ingoing edges from the input points, and remove the outgoing edges from the output points. Subsequently, we add the source α and sink ω, and match them to the input and output points respectively. The resulting subgraph is shown in Figure 23.

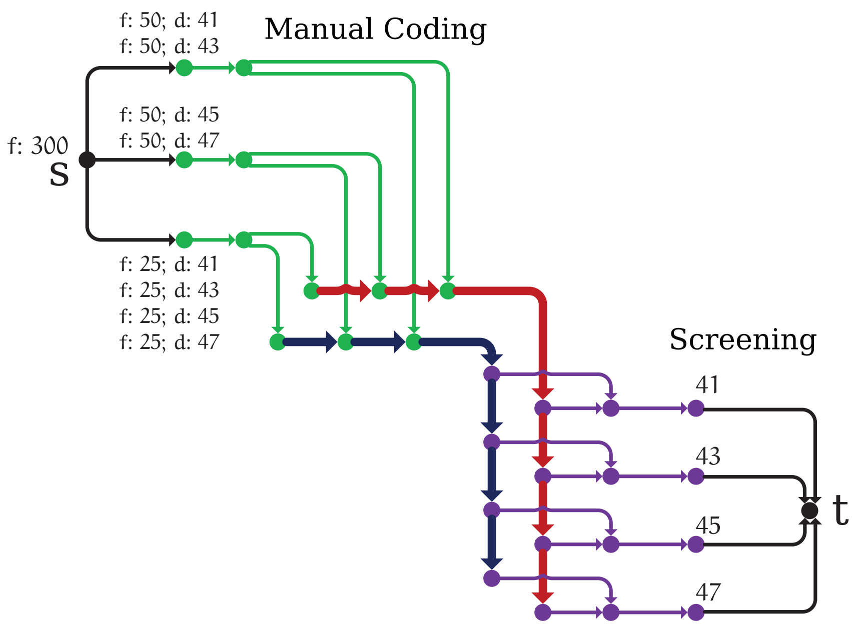


Figure 23: Flow from Manual Coding to Screening

Assume we want to send a flow of 300 bags over this subgraph. There are four possible destinations (Screening stations), with ids 41, 43, 45, 47. We have three Manual Coding stations which are the input points, so we divide the flow of 300 over these input points. This means that each Manual Coding will have a starting flow of 100.

For each Manual Coding station, we look at the task topology in order to determine what destinations we should route all bags to. Assume the task topology tells us that the upper Manual Coding station needs to route towards either 41 or 43, the middle station towards either 45 or 47, and the lower station can route towards all four. As such, we divide the flow of each Manual Coding station according to the task topology, and this results in two destination flows for the upper two Manual Coding stations (both containing destination flows with 50 flow each), and four destination flows for the lower Manual Coding station (with 25 flow each).

3.4.5.5 *Stateless flow*

One of the properties of the way flow is propagated through the network is its statelessness. Flow does not know where it is coming from, only where it is going, and even then it does not know how it is supposed to get there. This is a deliberate choice which we are making.

It creates a more abstract way of talking about flow, since we are only concerned with the current position of flow. This simplifies our modified push-relabel flow algorithm, as we never have to take into account the history of any destination flow.

The downside of stateless flow is just that, we are missing information about the history of destination flow. Naturally, it is still possible to generate routing rules with stateful flow, and we will discuss this methodology in Section 6.1.3.

3.4.6 *Analysing bottlenecks*

Once we have calculated the entire maximum overflow, using the main flow as user input, we can start to analyse what the bottlenecks in the graph are. We already explained how we can measure the performance of a set of routing rules in Section 2.7, and now we are going to apply these measurements using the maximum overflow we have calculated.

There are three possible ways for us to indicate a bottleneck:

1. A divert has to switch too often;
2. A merge has to switch too often;
3. An edge has more flow than the capacity of that edge.

The term 'too often' is a subjective measurement which the end user can determine. It differs per divert and merge how often is 'too often', and it is outside the scope of this project to determine this automatically. Each bottleneck indication is in itself a valid way for determining whether a bottleneck is present in a BHS, but it does not give us a way for comparing bottlenecks with each other and ultimately comparing sets of routing rules on the same BHS. This is why we introduce a way for incorporating all bottleneck indications into a single measurement.

3.4.6.1 *Defining bottlenecks*

We essentially have two ways for indicating a bottleneck, through switching and through the exceeding of capacity. However, when we have two diverts both switching equally often, it can still be the case that one divert has a bigger impact because of the capacity of its surrounding edges. So, we need a way to combine both the switching and capacity KPI. We do this, by calculating a *capacity reduction* on ingoing edges for both merges and diverts. The capacity reduction is the result of the amount of switching operations a divert or merge has to perform, combined with the capacity of each ingoing edge. A switch takes time, and the higher the amount of switching operations a divert or merge has to perform, the more time is lost as a result of these switches (see Section 2.3.1). For each second a divert or merge is busy switching, the incoming edges can handle less bags per second. This effectively lowers the capacity of all incoming edges, because the effect of switching takes place upstream of that switching node.

Applying the capacity reduction differs between diverts and merges. This, because the calculation of the amount of switches differs between diverts and merges. Worst case, a node n has to switch as much as the lowest amount of flow it needs to process (i.e. alternating between each direction) when $|\delta^+(n)| + |\delta^-(n)| = 3$, and as much as the lowest two amounts when $|\delta^+(n)| + |\delta^-(n)| = 4$. The total amount of switches for a node n with $|\delta^+(n)| + |\delta^-(n)| = 3$ becomes:

$$\#\text{switches}(n) = \begin{cases} \min(\{f(e) \mid e \in \delta^+(n)\}) & n \in D \\ \min(\{f(e) \mid e \in \delta^-(n)\}) & n \in M \end{cases} \tag{7}$$

And for a node n with $|\delta^+(n)| + |\delta^-(n)| = 4$ becomes:

$$e_{\max} = \begin{cases} \{e \mid e \in \delta^+(n) \land \exists e' \in \delta^+(n) : f(e') < f(e)\} & n \in D \\ \{e \mid e \in \delta^-(n) \land \exists e' \in \delta^-(n) : f(e') < f(e)\} & n \in M \end{cases} \tag{8}$$

$$\#\text{switches}(n) = \begin{cases} f(e_{\max}) + \max(\{f(e) \mid e \in \delta^+(n) \setminus \{e_{\max}\}\}) & n \in D \\ f(e_{\max}) + \max(\{f(e) \mid e \in \delta^-(n) \setminus \{e_{\max}\}\}) & n \in M \end{cases} \tag{9}$$

On average, a switch takes 3 seconds to perform, regardless whether it is a divert or a merge. Thus, the total amount of busy waiting for a node is $\#\text{switches}(n) \cdot 3$. During this busy waiting period, the node cannot process bags, and thus the capacity of each incoming edge is reduced by the amount of bags it could have processed in this period of busy waiting. This results in the following capacity reduction $c_{\text{redux}}(e)$ for all $e \in \delta^-(n)$:

$$c_{\text{redux}}(e) = \#\text{switches}(n) \cdot 3 \cdot \frac{c(e)}{3600} \tag{10}$$

We divide the capacity of e by 3600 because the capacity is denoted in bags per hour, and our busy waiting time is denoted in seconds. If we subtract this capacity reduction on all edges $e \in M \cup D$, we have a more realistic performance measurement on the impact a switch has on the system. This means that we have eliminated the skewed comparison between two switches who both had an equal switching ratio, but different capacities to cope with.

Whenever an edge $e \in E$ has a flow higher than the reduced capacity of that edge, $f(e) > c(e) - c_{\text{redux}}(e)$, we have a bottleneck. This bottleneck is caused by the node at the end of this e, due to its switching behavior. Thus, with a bottleneck edge $e_b \in E$, we have a corresponding bottleneck node $\tau(e_b) = n_b \in V$. The amount of overflow on the reduced capacity of the bottleneck edge indicates the severity of the bottleneck. This way, we can compare bottlenecks within the same set of routing rules, and even between different sets of routing rules for the same BHS. Whenever we want to resolve a bottleneck edge, we can only do this by looking at its corresponding bottleneck node. As such, for the remainder of this thesis, whenever we talk about resolving bottlenecks, we talk about bottleneck nodes.

3.4.7 *Analysis overview*

When we look at all aspects we have used for the analysis algorithm so far, we can clearly identify each of the layers of control we discussed in Section 1.4. Figure 24 contains a schematic of the relationship between each layer when we are analysing a set of routing rules.

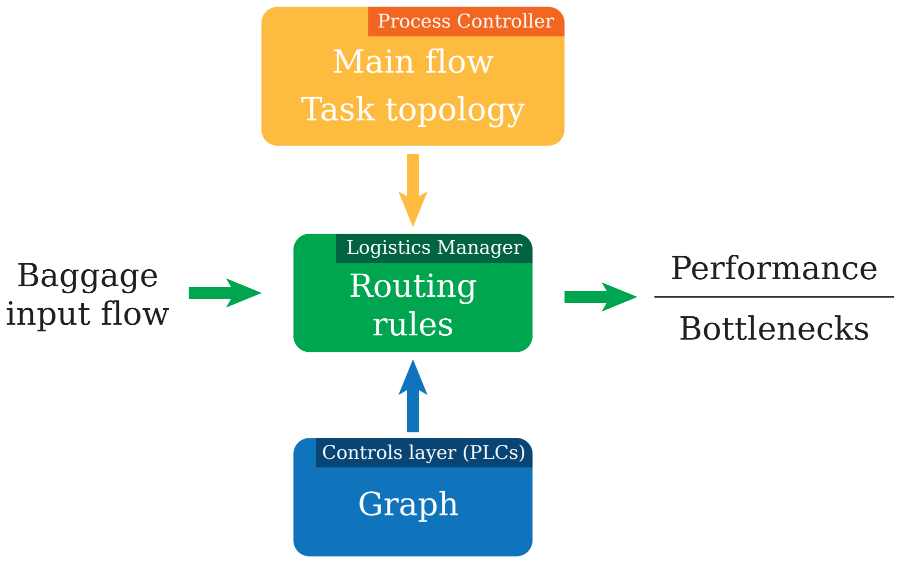


Figure 24: Overview of all layers during analysis

Naturally, we have the routing rules. The routing rules are an integral part of the LM and, as it is the main focus of this project, they play a central role in our analysis. The routing rules are based on the functionality of the graph, which in turn mimics the functionality of the PLCs (i.e., the controls layer). We emulate each conveyor belt through edges, and each junction through nodes, and thus we simulate the entire controls layer through this graph. Finally, we need information from the PC in order for us to simulate the LM. We do this by emulating the PC through the use of a main flow, which emulates the process logic the PC contains (what tasks a bag should complete), and we use the task topology to emulate what possible destinations could be handed out.

As we have said in Section 1.4, we needed to emulate parts of the BHS in order for us to receive all necessary information. We do this through all of the methodologies described in this section. Naturally, more realistic results could be possible if we have actual simulations of the PC and/or PLCs, but due to lack thereof, we try to approach these simulations as close as possible.

Important to note is that whenever we are analysing a BHS, we use the main flow given to us (by the user). This means that everything we analyse and, more importantly, optimize is dependent on this main flow. Everything not included in the main flow will therefore not be analysed and optimized.

One way of resolving this is to do a two step analysis/optimization cycle. First, one uses a main flow covering the entire functionality of the BHS, rather called a 'complete' flow. Next, when all rules have been optimized globally, one can use an actual main flow in order to focus on specific parts of the BHS that require special attention. For this thesis, we only focus on the use of a main flow and not an additional complete flow, for the sake of conciseness.

3.5 ITERATIVE OPTIMIZATION

After analysing a set of routing rules, whether it is generated by our creation algorithm or it is an existing set of routing rules, we can try to improve the set by applying optimizations. For this, we take the bottlenecks we have found as a result of our analysis, and change the routing rules in order to resolve these bottlenecks. We then re-analyse these modified set of routing rules in order to see if we have improved the performance.

3.5.1 *Ordering bottlenecks*

Before we can resolve bottlenecks, we first need to determine in what order we are going to resolve them. The main flow in our system determines how different areas in a BHS are connected, and in what direction baggage flow runs. This means that we also need to look at the main flow in order to determine the ordering of bottlenecks. For example, when we have a bottleneck in a Check-In area, and a bottleneck in a Screening area, it is very well possible that the bottleneck in the Screening area is the result of the bottleneck in the Check-In area, because Screening is done after Check-In. As a result of this, we want to first order the bottlenecks based on the main flow of the BHS in which they are present.

Each node in the graph can be mapped back to a single main flow node. This indicates that for each node, we have a corresponding main flow node and thus can determine the ordering based on the main flow. In Section 3.4.6.1, we defined bottlenecks as edges whose flow exceeds its corresponding reduced capacity, and their corresponding nodes causing the bottleneck edges. Because bottlenecks edges are the result of switching behaviour downstream, whenever we have a bottleneck edge $e_b \in E$, the node causing that bottleneck is its downstream node $\tau(e_b) = v_b \in V$. This node v_b will thus be the bottleneck node of e_b. Through n_b, we can determine what main flow node n_b can be mapped to. If we do this for all bottlenecks, we have an initial ordering of bottlenecks based on the main flow.

Should we try to resolve bottleneck nodes, we start with the bottlenecks that can be mapped to the first main flow nodes in the main flow, i.e. a main flow node $n_m \in V_m$ whereas $|\delta^-(n_m)| = 0$. There is a special case on nodes that lie on the edges of two areas defined by two distinct main flow nodes. These nodes can be mapped to both main flow nodes; in this case we map the node to the main flow node that has the outgoing edge towards the other main flow node. More formally, we can possibly map node $n \in V$ to two main flow nodes $n'_m \in V_m$ and $n''_m \in V_m$. If there is an edge $e_m \in E_m$ such that $\sigma(e_m) = n'_m$ and $\tau(e_m) = n''_m$, then we map n to n'_m, else we map n to n''_m.

Once we have a collection of sets of bottlenecks that all map to their own main flow node, we still need an ordering inside these sets of bottlenecks. For instance, if we take all bottlenecks that map to the Check-In main flow node, we need to know how we order the bottlenecks inside this set. For this, we use a distance measurement inside the area defined by the main flow node, using the distance function $\delta(v, w)$ with $v, w \in V$ as described in Section 3.2.

Because each main flow node defines a subgraph $G_m \subseteq G$, we can determine the distance in this subgraph measured from its starting nodes to its ending nodes. Whenever we have a bottleneck node $n_b \in E$, we calculate the shortest distance from the start of G_m. This distance measurement can then be used to order the bottlenecks in a way that will provide us with enough information to be able to overall solve the bottlenecks from the 'front' of the graph to the 'back' of the graph (or, from upstream to downstream).

Now that we have defined an ordering on the bottlenecks, we can start trying to resolve the bottlenecks. In this ordering, we take the first bottleneck node $n_b \in V$, and match the type of n_b against a few cases in order to determine how we are going to try to resolve this bottleneck:

- Divert
  - On a loop
  - Not on a loop
- Merge
  - On a loop
  - Not on a loop

Additionally, in some cases it can be feasible to try to resolve multiple equivalent bottlenecks simultaneously. Assume we have a set of bottleneck nodes $B \subset V$. For each bottleneck $v_b \in B$, we can determine non-identical bottleneck equivalent nodes, i.e. other nodes $v \in V : v \neq v_b$ that map to the same main flow node. For an edge v to be bottleneck equivalent to v_b, they need to have identical reachable destinations. This means that it does not matter for a bag whether it is at v or v_b, it can still get to all of its destinations from either of the two. Whenever we have multiple equivalent bottlenecks, we have more options on spreading the load of the bottlenecks over the several bottlenecks, with an increased probability of reducing and/or solving the bottleneck.

3.5.2 *Resolving bottlenecks*

Whenever we have a bottleneck, it is either present on a divert, or a merge. Indirectly, bottlenecks at merges are the effect of inefficient routing rules at diverts upstream (because merges themselves cannot reroute baggage), or the result of unavoidable merging. As will become apparent, resolving the bottlenecks at merges will be done through bottlenecks further upstream.

There are four possible scenarios for a bottleneck. Either it is a divert or a merge, and then it can either lie on a loop or not. In this section, we will describe each possible scenario for a bottleneck, and how we are going to try to resolve this. For each solution, we assume that, whenever we are dealing with a bottleneck, it will be the first bottleneck present in the system, given that we traverse the graph from the begin nodes (Check-In, Transfer-In, etc.) downstream to the end nodes (Laterals, etc.). Through this assumption, we know there are no other possible bottlenecks upstream that may be influencing the bottleneck we are trying to resolve.

The general solution for resolving bottlenecks will be to modify the routing rules that could be affecting the bottlenecks. A bottleneck is always a case of high load on a node or edge (or both), and thus we want to reroute (part of) this high load. Our best effort to do this will be to modify large quantities of routing rules, and this can easily be achieved by looking at station groups. We explained the notion of station groups in Section 2.5. They are sets of task equivalent stations (destinations), which are generally considered altogether when handing out a specific destination. Due to their equivalence, we can try to reroute part of a high load by modifying the routing rules of all stations in a station group, and preferable the largest station group that is reachable at the bottleneck. This rerouting is done by modifying the routing rules affecting the bottleneck.

Other possibilities when rerouting is to look at single destinations, or all destinations. Looking at single destinations means modifying a single routing rule per optimization iteration. The subsequent analysis step for checking whether the small modification is a (small) improvement takes too much time. Thus, looking at single destinations is infeasible and takes too much computational time. Taking all destinations into account is too coarse a step, because one either modifies all routing rules, or one modifies none. This results in solutions that are alternating between two extremes.

3.5.2.1 *Modifying routing rules*

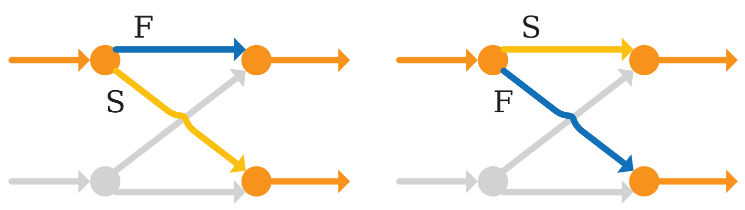
Modifying a set of routing rules in order to optimize them relies on the alternative paths we have discovered during the initial creation of the set of routing rules, as described in Section 3.3, or the presence of those paths in any other (not created by us) set of routing rules. This means that aside from Favourite usages, we also need Substitute Favourite and Option usages in the routing rules, otherwise we have no knowledge of alternative paths.

During each of the possible bottleneck scenarios we will talk about 'modifying the routing rules'. Because these modifications are the same each time, we elaborate on this operation here. It also became apparent that we only modify the routing rules themselves, meaning that we only do this at diverts.

Whenever we modify a single routing rule, we have for a specific destination an already existing routing rule indicating that for a certain divert $d \in D$, we have a certain outgoing edge $e \in \delta^+(d)$ with a Favourite usage. Modifying this routing rule such that we try to resolve a bottleneck means 'flipping' the routing rule over: we regard one of the other outgoing edges as Favourite, create a routing rule for that and remove the already existing routing rule, i.e. the Favourite usage flips over to an other outgoing edge.

Whenever divert d has an outdegree size of 2, $|\delta^+(d)| = 2$, flipping the routing rule over is quite easy. We have an edge e which is the current Favourite edge for a specific destination, and should the other edge $e' \in \delta^+(d) : e \neq e'$ have a Substitute Favourite or Option usage for the same destination, we just flip the usages around. Now e' will become Favourite, and e will become Substitute Favourite or Option, regarding the situation before the flip. An example of this operation is shown in Figure 25.

Should $|\delta^+(d)| = 3$, we have two choices for the usage to flip over to (two remaining outgoing edges). In this case, we take the outgoing edge that has the lowest amount of flow. This way, we minimize the chance that we create a new bottleneck downstream by flipping to an already highly stressed edge.



(a) Before flip (b) After flip

Figure 25: Flipping usages

3.5.2.2 *Divert; not on a loop*

Whenever the bottleneck is a divert $d \in D$ not residing on a loop, we have the most straightforward scenario to resolve. We know (by assumption) that there are no bottlenecks upstream, thus the only way we can possibly try to resolve this bottleneck is by alleviating the load on the edge with the highest overflow. Here, the bottleneck is the divert node itself. When we have no bottleneck equivalent nodes, we will look at the largest station group present at this bottleneck, and modify the routing rules for each station in this station group.

If we do have bottleneck equivalent nodes, we have a set of bottleneck equivalent nodes B. We order B based on the severity of the bottleneck, i.e. on the amount of capacity reduction of the ingoing edge for each bottleneck $b \in B$, from high to low. Subsequently, we divide B into two (roughly) same sized sets of bottlenecks B' and B'', such that $|B'| = \left\lceil \frac{|B|}{2} \right\rceil$ and $|B''| = \left\lfloor \frac{|B|}{2} \right\rfloor$. B' will contain the first $|B'|$ elements of B based on the ordering on bottleneck severity, whereas B'' will contain the last $|B''|$ elements of B based on the same ordering.

Now, we will modify the routing rules of the largest station group for the bottlenecks in B' only, not for the bottlenecks in B''. This will put more pressure on the bottlenecks in B'', and relieve the bottlenecks in B' from their high load.

3.5.2.3 *Divert; on a loop*

The scenario when the bottleneck node is a divert residing on a loop is almost identical to the non-loop scenario, except we now have the transport default property of the loop to consider. Whenever we modify routing rules, and we want to reroute baggage over the loop, we first need to check if there is actually an alternative path present. An alternative path is a path that does not include the same divert again (i.e. we do not want a full traversal over the loop to wind back up at the bottleneck again).

If there is such an alternative path, we can apply the same scenario as when we have a divert that does not lie at a loop. If there is no possible alternative path, we cannot resolve this bottleneck. In that case, the divert is the only possible way for all of its routing rules, and thus it is impossible for us to reroute baggage in order to relieve the load on the divert.

3.5.2.4 *Merge; not on a loop*

Whenever the bottleneck is a merge $m \in M$ not residing on a loop, we need to trace the problem back upstream. We do this by taking the incoming edge $e_{max} \in \delta^{-}(m)$ with the highest overflow on the reduced capacity: $\neg\exists\, e' \in \delta^{-}(m) : f(e') - (c(e') - c_{redux}(e')) > f(e_{max}) - (c(e_{max}) - c_{redux}(e_{max}))$. We take the edge with the highest overflow on the reduced capacity, because the effect of resolving this bottleneck will be the highest.

We traverse upstream (i.e., backwards) through e_{max} until we reach the first divert $d \in D$. If we reach d, this means that we have a path p from d to m such that both m and d are in p ($m \in p \wedge d \in p$), and that there is no other divert in p: $\neg\exists\, d' \in D : d' \neq d \wedge d' \in p$. Now that we have traced the problem back to divert d, we will treat d similar as to it being a bottleneck itself. Thus, we take the largest station group, and modify all routing rules containing the first edge of p.

This is the solution for resolving a single bottleneck m. If we have a set of bottleneck equivalent merges B, we trace each bottleneck $b \in B$ back to its respective divert, in an identical way as when we have a singular merge bottleneck. Now, we have a set of bottleneck diverts B_d, identified through each of the merge bottlenecks in B. We divide B_d in sets of bottleneck equivalent diverts. For each subset $B_{de} \in \mathbb{P}(B_d)$, we apply the same non-loop divert bottleneck solution as presented in Section 3.5.2.2, depending on the size of B_{de} accordingly.

Merge; on a loop

Should the bottleneck be a merge $m \in M$ which resides on a loop, we can perform almost the same resolve technique as whenever m is not on a loop. However, instead of traversing back over the incoming edge with the highest capacity reduction, we take the edge that does not lie on the loop. This, because whenever m itself is a bottleneck, the probability that this is caused due to baggage being inserted on the loop is very high, as opposed to baggage that is already on the loop itself. Should there be two edges that are not on the loop (i.e., we have three ingoing edges in total), then we still take the non-loop edge with the highest capacity reduction.

Whenever we traverse back over (one of) the incoming non-loop edge(s), we can apply the rest of the resolving technique identical to the non-loop merge bottleneck technique.

Prevent flow crossing

Whenever we have a merge bottleneck, there is the possibility that the bottleneck is present due to crossing flows, as we have described in Section 2.6.3. For both the merge bottleneck cases, we should also look for the prevention of these flow crossings.

If we look at the main flow, we can identify possible flow crossing points. Each time a main flow node $n_m \in V_m$ has two or more ingoing edges (i.e. $|\delta^-(n_m)| > 1$), there is a possibility that we have crossing flow. So, if we have a merge bottleneck that maps back to such a main flow node n_m, we need to check if we can possibly reroute baggage such that we prevent the crossing of flow.

In addition to the techniques we already discussed for resolving merge bottlenecks, if we have a flow crossing possibility, we need to divide the station groups per main flow edge over the bottlenecks. This means that, if we have multiple diverts, we want to dedicate each flow to a specific set of diverts, as we have also shown in the crossing flows interesting scenario in Figure 16a. We want to prevent overlap between these sets of diverts, as this would again result in crossing of flow. So, whenever we are modifying the routing rules in a merge case, we also want to dedicate the main flows whenever we are dealing with a (possible) flow crossing scenario.

3.5.3 *Re-analysis and re-iteration*

Each time after an attempt to resolve a bottleneck, we need to reanalyse the entire graph in order to make sure that we indeed did resolve the bottleneck and thus have further optimized the routing rules. This is the iterative part of the optimization algorithm. We do not want to try to resolve multiple non-related bottlenecks at once, as this makes it difficult to indicate what optimization(s) did indeed optimize the routing rules. Thus, we only focus on a single bottleneck at once, or multiple if we have bottleneck equivalency, and analyse after each modification step.

After re-analysing with the modified routing rules, several situations can become apparent:

1. The bottleneck has been reduced or entirely resolved;

2. The bottleneck has not been reduced;

3. The bottleneck has been increased;

4. The bottleneck has been either reduced, increased or remains identical, but one or more new bottlenecks have been introduced as well.

We will only regard Case 1 as an optimization/improvement of the set of routing rules. All other cases we will regard as a decline in the performance of the set of routing rules. Should the outcome of the re-analysis be any other than Case 1, we will roll back the modification to the state of the set of routing rules as they were before we started modifying them. Even whenever we have reduced (or resolved) the bottleneck, but introduced one or more new bottlenecks (Case 4), we will roll back the modifications. This, because the effects of modifications are hard to keep track of when we try to resolve multiple layers of bottlenecks introduced with each layer of optimization. Thus, we stick to one layer of optimization only, and only if we can optimize within that layer without introducing new bottlenecks, we will continue with the modified set of routing rules.

Once we have acknowledgement that we have indeed further optimized the set of routing rules (i.e. the re-analysis reports that we have a Case 1 modification result), we will use the modified set of routing rules as a new basis for further optimization. We will again run the optimization algorithm on the remaining bottlenecks, further trying to resolve those as well. However, should we have to perform a rollback (Cases 2, 3 and 4), we will still keep track of the tried modifications. This is to prevent the optimization algorithm from trying to apply the same modifications each iteration. This also enables us to keep track of all tried modifications per bottleneck. Since each divert has a finite number of station groups to which it can possible route, we also have a finite number of modifications we can apply on the routing rules. After we have tried all modifications, we have exhausted all possibilities of resolving the bottleneck. Should it still be present (either fully present or in reduced form) we will regard the bottleneck as 'dried out', i.e. we can no longer try to optimize it and will move on to other bottlenecks.

3.5.4 *Maintaining validity*

After each optimization iteration, we will have a modified set of routing rules. Especially if we start optimizing a set of routing rules that we created initially through our creation algorithm, we need to prove that we still adhere to the 'valid' property of that initial set of routing rules. In Section 3.3.4, we have proven the validity of our initial set of routing rules. Now, we are going to prove that all modifications the optimization algorithm performs still keep this validity intact.

The creation algorithm not only generates a set of routing rules for primary paths with Favourite usages, it also generates routing rules for alternative paths using Substitute Favourite and Option usages. When we are modifying a routing rule, we flip the Favourite usage if and only if there is an alternative path indicated with a Substitute Favourite or Option usage. This simplifies our proof to the question whether or not the alternative paths generated by the creation algorithm are valid, alongside the primary paths.

Whenever the creation algorithm finds a best path between a divert $d \in D$ and a destination $t \in T$, it will also look for alternative paths that lie within a certain (user-defined) percentage of that best path, cost-wise, as described in Section 3.3.3. These alternative paths are all n^{th} best paths: there are $n - 1$ better paths between d and t. Still, the path finding algorithm for alternative paths is exactly the same as for the best path. This means that (given the proof in Section 3.3.4), whenever there is an alternative path, that path will be a valid one. Using that alternative path when modifying the routing rules thus replaces a valid routing rule with an alternative but still valid rule. The resulting modified set of routing rules will still be valid, and thus we have proven our optimized set to also be valid at any time.

3.6 SUMMARY

In this chapter, we showed the theoretical solutions for solving the routing generation problems as presented in Chapter 2. We presented a framework as the foundation for these solutions, shown in Figure 18. In this framework, we have three algorithms: the *creation algorithm*, the *analysis algorithm*, and the *optimization algorithm*. Each algorithm is responsible for a distinct aspect in the process of generating a valid and good set of routing rules:

- Creating a valid set of routing rules (creation algorithm);
- Analysing this (or a) set of routing rules (analysis algorithm);
- Improving the performance of the analysed set by modifying the routing rules and analysing them again (optimization algorithm).

We have explained the graph notation which we will use throughout the rest of this thesis. Subsequently, we explained each of the aspects present in the framework.

The creation algorithm is primarily concerned with creating a valid set of routing rules, where the performance of the set of routing rules is not of direct importance. We use Dijkstra's single-source shortest path algorithm as the basis for the creation algorithm. We apply modifications to the cost function, incorporate travel time and unwanted shortcut prevention in order to come to a best path cost function.

With the modification of Dijkstra's shortest path algorithm into a best path algorithm, we can create an initial set of routing rules. We calculate, for each divert, towards each possible destination for that divert, the best path available. The first edge of that best path will become a routing rule with the Favourite usage. All non-Favourite edges can still be applicable for alternative paths, so we keep on checking for second and third best paths, which we could possible have as an alternative path. Should there be such an alternative path, it will get a Substitute Favourite or Option usage, depending on the context of the edge.

The analysis algorithm is used for analysing the performance of a set of routing rules. This set of routing rules does not necessarily have to be created by the creation algorithm, it can also be an already existing set, for example on a actual operating BHS. Simulations which Vanderlande runs are too costly for us, which is why we introduce a new way of analysing the performance of a set of routing rules: flow algorithms.

We take the notion of flow and Goldberg and Tarjan's push-relabel flow algorithm to apply it on our analysis method. The actual algorithm is too restrictive to be applicable for us, which is why we relax some of its constraints. The intuitive idea behind the algorithm is that we push flow from the input of the graph to the output of the graph. We introduce the notion of maximum overflow, to indicate flow exceeding the capacity of its edge. Through this overflow, we can show where in the BHS the routing rules are creating bottlenecks.

Methodology to 'steer' flow in our graph is presented, and we first introduce the notion of a main flow. Main flow indicates the major areas of importance in the BHS, and in such a main flow graph we can indicate how these major areas are interlinked. Through this main flow, and the task topology introduced in Section 2.5, we can divide flow throughout the graph in order to maximize our coverage of the BHS.

Once the flow algorithm is done, we can start analysing the resulting maximum overflow graph in order to indicate bottlenecks in the system. We recognize three different type of bottlenecks in the system:

1. A divert has to switch too often;
2. A merge has to switch too often;
3. An edge has more flow than the capacity of that edge.

We introduced the notion of *capacity reduction* in order to be able to compare bottlenecks in the same set of routing rules, and between different sets of routing rules. This capacity reduction is based on the amount of switching of a node, resulting in a decreased capacity of all edges leading to that node. The more the node has to switch, the higher the capacity reduction will be.

Once we have analysed a set of routing rules and know what the bottlenecks are, we can use our optimization algorithm to try and improve the set of routing rules. We look at each bottleneck, try to shift around with the routing rules, analyse the set again and see if the bottleneck has been resolved.

In order to efficiently try to resolve bottlenecks, we imposed an ordering based on the main flow of that BHS. Each bottleneck can be mapped to a main flow node, and thus we have an initial ordering. Within this initial ordering, we order all nodes that map to a single main flow node based on the distance measurement of a node. This is the distance from the starting nodes to the bottleneck, and through this we have an ordering sufficient enough for us to start trying to resolve bottlenecks from upstream to downstream.

Subsequently, we defined four cases for the actual resolving of a bottleneck. Either a bottleneck is a divert or a merge, and then it can either reside on a loop or not. We introduce the definition of bottleneck equivalency, indicating that nodes are bottleneck equivalent if they are distinct nodes, but have an identical set of reachable and possible destinations. Next, we defined resolving techniques for each bottleneck case:

- *Divert; not on a loop* We modify the routing rules for the largest station group whenever we have a single bottleneck, and modify the routing rules for the largest station group for half of the bottlenecks if we have a set of bottleneck equivalent bottlenecks;
- *Divert; on a loop* Almost identical to the previous case, however we also have to check if there is an alternative path present;
- *Merge; not on a loop* Bottlenecks at merges are caused by bottlenecks upstream, so we search for the first divert upstream through the edge with the highest overflow on the reduced capacity and try to apply the divert resolving case on it. Should there be multiple bottlenecks, we match each bottleneck to its divert, and divide those diverts in sets of bottleneck equivalent diverts. We then solve it simultaneously, as with the divert case;
- *Merge; on a loop* Almost identical to the previous case, however, we now only look at incoming edges not part of the loop.

Modifying a (set of) routing rule(s) means that we 'flip' the routing rule over to other possible edges. We look at all non-Favourite edges, and check if we have a Substitute Favourite or Option. If so, we switch the usages on both edges around, such that the original Favourite edge now becomes a Substitute Favourite or Option, and the non-Favourite edge now becomes a Favourite.

Once we have modified all routing rules in a single iteration, we reanalyse the resulting modified set of routing rules. Should we have reduced or completely removed the bottleneck, we continue with the modified set of routing rules. If this is not the case, we roll back the modifications, and try other resolving techniques.

Because we are only operating within the bounds that are created by the creation algorithm, we can also prove that our modifications still result in a valid set of routing rules. This, because each alternative path generated by the creation algorithm is also valid, as opposed to the best path. Since we are only using these alternative paths, and no other paths, as a substitute for the best path, we still have a valid set of routing rules.

4 IMPLEMENTATION

In this chapter, we will expand the solutions as given in Chapter 3. Firstly, we discuss the coherence of each solution in a bigger picture. We will show what the interaction of the end-user is, and how both algorithms will come to a solution. Secondly, each problem will be discussed with a more in-depth elaboration on the used solution. For this, we provide pseudocode in Appendix A, and the necessary detail on the implementations in this chapter.

4.1 INPUT DATA

There are several datasets available as input data for our framework. As the foundation of all this data, we have the Node Segment Diagram (NSD). This NSD is the graphical representation of the BHS, as used internally at Vanderlande. This NSD is a Visio file, with nodes and segments to visualize a BHS. Each node and segment, both defined as *shapes*, have shape data. This shape data is both partially present in the NSD, as well as separately (and more extensively) in an Extensible Markup Lanuage (XML) file.

We can map the NSD to our graph, defined in Section 3.2. Both the nodes and segments in the NSD can be directly mapped to nodes and edges in our graph, respectively. After mapping this NSD to our graph, we have the following data available to us.

Node

ID Unique identifier.

TYPE Type of the node (Workstation, Exit, Merge, Divert, etc.).

Destination

ID Unique node identifier.

DESTINATION_ID Unique destination identifier.

TYPE Type of station (e.g. lateral, HBS, AutoScan).

MAXIMUM_CAPACITY Bags per minute the station can at most handle. This is convenient for setting the input of a subgraph when running the analysis algorithm.

Edge

ID Unique identifier.

NODE_ID_START The id of the starting node.

NODE_ID_END The id of the ending node.

IND_TRANSPORT_DEFAULT Indicates whether the segment is a transport default (can be the case with loops).

SYSTEM_CAPACITY Bags per hour the segment can at most transport.

TRAVEL_TIME Time in seconds for a bag to traverse over the segment.

STATUS_SEGMENT_ID The id of the status segment to which this route segment belongs.

Status Edge

Status segments are the equivalent of status edges in our graph, i.e. they are sets of edges that are failure sensitive towards each other.

ID Unique identifier.

TRANSPORT_TYPE General type of the edges it contains (e.g. tubtrax, bagtrax).

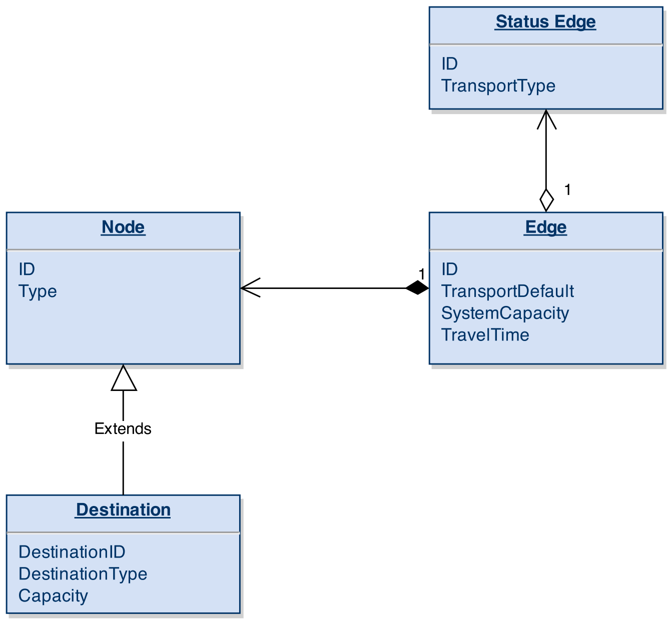


Figure 26: Class diagram of the input data

After mapping the data to our graph, we have the class diagram as shown in Figure 26.

4.2 utilization

An important factor in the creation of a tool which uses the framework shown in Figure 18, is how it is used: the utilization of such a tool. The end goal of the tool is the same as the goal of this graduation project: to generate a good and valid set of routing rules given a BHS.

Here, we will explain how the user will use the framework, what input and output data is expected for each aspect of the framework, and what user interaction is necessary.

4.2.1 *Initial creation algorithm*

For this we have, as explained, the mapping of the NSD to our graph. Additionally, we determined in Section 2.5 that we need the task topology as user input, in order to retrieve all possible destinations. Naturally, the task topology that is given has to match the BHS on which the NSD is based.

The user can indicate whether or not edges in the graph are disabled or not. It could be possible that some edges in the graph should never be considered during routing, or functionality of areas in the BHS is incomplete and generating routing rules for those areas is unnecessary. All disabled edges will be ignored during the initial creation of routing rules.

Once the user ran the initial creation algorithm, it will have generated routing rules for each divert towards each reachable and possible destination. These routing rules can be exported to a file by the user in the form of an $E \times D$ matrix. If there is a usage present for a tuple $(e, d), e \in E \wedge d \in D$, it will be indicated as either *F*, *S*, or *O* for Favourite, Substitute Favourite or Option, respectively. If there is no usage present, the tuple will still be present in the matrix, but the usage will be empty.

4.2.2 *Analysis algorithm*

The input data for the analysis algorithm is identical to the input of the creation algorithm, plus the output data of the creation algorithm: a dataset of routing rules in the form of an $E \times D$ matrix. This dataset does not necessarily have to be the output data of an execution of the creation algorithm. It can also be extracted from an already existing set of routing rules present in a BHS.

Additionally, we need the main flow of the given BHS as user input. This main flow will be a graph as well, as is shown in Figure 21. This main flow will have a node for each major area present in the BHS, or a subset of this set of possible nodes. In our graph $G = (V, E)$, each node $n \in V$ can be mapped to one node in the main flow. This means that each main flow node can be mapped to a set of nodes $Q \subseteq V$, but there is is no overlap with the resulting set from an other main flow node. Each main flow node will have at least one edge, connecting it to an other node. Each edge indicates the possibilities for a bag at a specific node. Each outgoing edge determines what areas the bag can be routed towards.

After the analysis algorithm has been run, the graph has been augmented with more data. Each edge will have a certain amount of flow and a capacity reduction, and each node will have a switch ratio. These values will be used to indicate whether or not a node or edge is a bottleneck. A single performance measurement will also be given, indicating on a whole how the routing rules perform.

4.2.3 *Optimization algorithm*

Once we have a graph augmented with data from the analysis algorithm, the user can run the optimization algorithm. The optimization algorithm will make decisions based on the intensity of the bottlenecks, in order to resolve them. Each iteration of the optimization algorithm will try to solve one bottleneck. After an iteration the algorithm will have, if it has found a more optimal set of routing rules, modified the input set of routing rules. Again, the user can export this set of routing rules in the same way as is possible after the initial creation of routing rules.

It could be possible for the user to manually execute iterations in the optimization algorithm, or to let it run until it converges towards a solution where improvements are difficult to get or the improvements are only marginally better.

4.2.4 *Implementation framework*

Now that we have established the input and output of each algorithm, we can take the framework shown in Figure 18 and extend it. This results in an implementation framework, shown in Figure 27.

We have the task topology as user input for both the creation and analysis algorithm, and the main flow is user input for the analysis algorithm. The creation algorithm additionally takes the BHS data, in order to generate a set of routing rules. These routing rules are the main dataset on which all three algorithms operate. The creation algorithm generates them, the analysis algorithm analyses them and generates KPIs, and the optimization algorithm modifies the rules based on the generated KPIs.

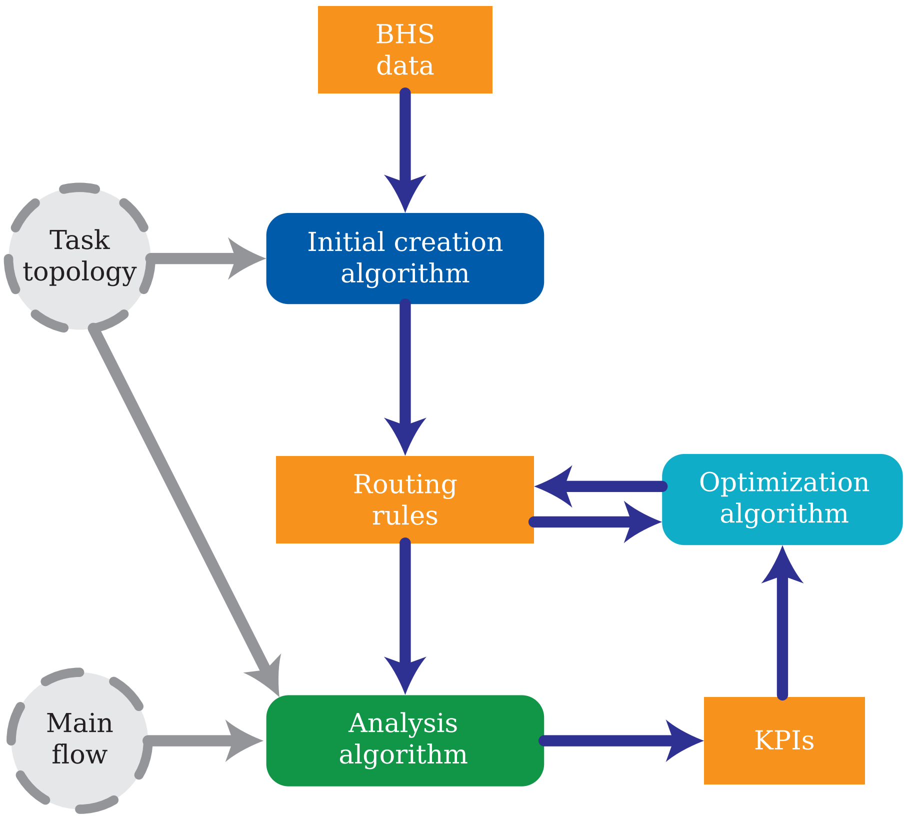


Figure 27: Implementation framework

4.3 INITIAL CREATION ALGORITHM

Even though the creation algorithm is only used for initial generations of routing rules, it is still a vital part of the entire framework. It uses Dijkstra's single-source shortest path algorithm[Dijkstra, 1959] as its basis. Dijkstra's running time is $O(n^2)$, where $n = |V|$ given a graph $G = (V, E)$[Korte and Vygen, 2000]. However, this is only the running time for a single instantiation of Dijkstra's. Since we need to calculate for every divert, for every reachable and relevant destination wether or not we have a path, the worst case running time is increased by a factor n: $O(n^3)$. This results in a running time which is too high for our application, we simply have too much nodes —on average there are thousands of nodes present in a BHS.

Our solution for reducing this running time is by using a modification of Dijkstra's shortest path algorithm, which uses a data structure called *Fibonacci Heaps*[Fredman and Tarjan, 1987]. Our graph is a sparse graph, where $|E| \approx |V|$. The proof for this is straightforward: every node has at least one edge, but never more than four, due to constraints on the BHS determined by Vanderlande. So, worst case we have $|E| = 4 \cdot |V|$, which is still very sparse compared to dense graphs where the maximum number of edges is achieved ($|E| = |V| \cdot |V - 1|$, when we do not take loops into account). The implementation of Dijkstra's with Fibonacci Heaps is most effective on sparse graphs[Korte and Vygen, 2000], which is an additional argument for the use of this implementation.

These Fibonacci Heaps are capable of performing deletion operations on an n-item heap in $O(\log n)$ worst case running time. However, all other heap operations can be completed in $O(1)$ worst case running time, significantly reducing the running time of Dijkstra's algorithm from $O(n^3)$ to $O(n^2 \log n + nm)$.

Practical test results showed that the running time of the creation algorithm on a real BHS with around 2000 nodes went down from 45 minutes to under 2 minutes due to the use of Fibonacci Heaps as the underlying data structure.

4.3.1 *Finding a best path*

In Algorithm 1, we present the algorithm for finding the best path between a source node s and a target node t. The algorithm uses Dijkstra's functionality as explained in Section 3.3.1, however here it is modified to be used with Fibonacci Heaps and the heap nodes that are part of the data structure. The SCAN function plays a vital part in the algorithm, for it is the function which traverses through the heap nodes stored in the heaps. For further explanation of the specific operations on the Fibonacci Heaps, we refer to the excellent explanation present in the author's paper[Fredman and Tarjan, 1987].

In Algorithm 2, we show the SCAN function used in the Algorithm 1. Each scanning operation takes the minimum node of all heaps (since that is our most valuable candidate for the best path), and traverses the heap.

Finally, the cost function as we have defined it in Section 3.3.2 used in the algorithm is shown in Algorithm 3.

4.3.2 *Creating initial set of routing rules*

Now that we have established a way to calculate the best path from any node to any other node, we can use this algorithm to create an initial valid set of routing rules. The resulting algorithm is shown in Algorithm 4. This algorithm is the final algorithm which represents the entire generation scope in the framework.

We loop over every divert $d \in D$. For that divert, we retrieve the taskgroup to which the divert belongs, using the task topology. Through this taskgroup, we know what the possible destinations T_p from that divert are. Subsequently, we run for each $t \in T_p$ the BESTPATH algorithm. Should the BESTPATH algorithm return a path, we know that this path is the best path for the given combination of nodes. Thus, we create a routing rule indicating a usage Favourite for that combination. Still, we need to calculate all alternative paths besides the best path. We do this by disabling the first edge of the best path, and again running the BESTPATH algorithm for each remaining edge.

After the algorithm has completed, we have a $E \times T$ matrix *RoutingRules* containing all routing rules.

4.4 ANALYSIS ALGORITHM

With the analysis algorithm, we take a matrix $E \times T$ containing routing rules. This matrix does not necessarily have to be created by our creation algorithm, it can also be extracted from already existing BHS configurations. Additionally, we use the same data input as the creation algorithm; the BHS data, and the task topology. Finally, we also add the main flow graph, defined by the user. This main flow graph defines part of the BHS our algorithm cannot easily extract. An example of this graph is shown in Figure 21.

The actual implementation of the analysis algorithm is very complicated, mostly due to the special cases necessary to properly push flow over a BHS. Since stating this here would require multi-page pseudocode examples, we will only show the most straightforward cases and leave out the special cases.

4.4.1 *Calculating flow through a (sub)graph*

We begin with the initialisation of our graph $G = (V, E)$ with the initialization code shown in Algorithm 5. We simply add a source α and sink ω to V, and connect those with all relevant nodes. Subsequently, we already start setting the flow of the input points. We assume we have some sort of integer amount of input flow called *inputFlow*. This amount will be divided over all input points, and we will create flowsets for each group of flow.

It should be noted that graph G does not necessarily have to be the complete graph as used in the creation algorithm. It can also be a subset of the graph. This is relevant in our case of main flow. Since we do not calculate the flow of our entire graph at once, we use the main flow to divide the calculation into pieces. Each edge in the main flow graph will be a separate calculation, using the appropriate starting nodes and ending nodes. The input flow of this subgraph will be the flow as has been iteratively calculated from the starting nodes of the main flow, up to the point where we have our input flow.

After initialisation, we can start calculating the flow over the augmented graph. We do this by calling the DISCHARGE method shown in Algorithm 6. This algorithm is the core of our analysis algorithm. It contains a queue Q of active nodes, and for each node will keep on applying PUSHRELABEL operations until the node become inactive. Should the label of the node become higher than the total amount of nodes in the graph, we know that we have flow which cannot get to its destination, and we will abort the operation.

The PushRelabel method shown in Algorithm 7 consists mainly of pushing flow around over all possible edges of v. Should a node w become active during this pushing process, it will be added to the queue Q. Due to the size of the PushRelabel operation, both the Push operation, as well as the ApplicableForPush method are shown separately in Algorithms 8 and 9.

The Push method is concerned with pushing flow from v over edge e to node w. It takes into account the routing rules RR, or other aspects of the system like transport default edges. It should be noted that, although we talked about 'via diverts' d_v in Section 3.4.5.3, we left the implementation out in the pseudocode. We do show how to balance flow in Algorithm 10, but the actual use of via diverts is too repetitive to show here. Implementation is quite trivial: if a flowset has a via divert set, it should ignore any other diverts until it reaches the via divert. Once it has traveled over this divert, the via divert should be reset in the flowset.

After the Discharge method has completed, graph each edge $e \in E$ has been augmented with integer amounts of flow, indicating what the amount of flow was that traveled over e during the calculation.

4.4.2 *Indicating bottlenecks*

Once we have a collection of subgraphs, each containing a flow calculation for an edge in the main flow, we can start indicating the bottlenecks in our system. First, we need to merge all subgraphs back into one single graph containing all flow calculations. This is quite trivial, as it only requires a summation of all edges in each subgraph, and as such we will not show that here.

What we are interested in, however, is the calculation of switch ratios and capacity reductions. This is shown in Algorithms 11 and 12, respectively.

4.5 OPTIMIZATION ALGORITHM

Once we have a list of bottlenecks presented by the analysis algorithm, we can use the optimization algorithm to try and resolve these bottlenecks. In order to save time during the implementation phase, we have not implemented the bottleneck ordering system as described in Section 3.5.1. The ordering itself is not relevant for the actual performance of the optimization algorithm, and as such we have emulated the ordering by manually following the exact same ordering as was described in Section 3.5.1.

Due to the straightforward nature of the optimization algorithm, we will not go into detail about aspects of the algorithm that are trivial to implement, like the case distinction between the type of bottleneck and how to solve these, as well as how to determine the equivalency of bottlenecks, as this relies on the task topology.

In Algorithm 13, we show the flipping of routing rules. Important to notice when implementing this is to keep track of the current state of usages on the edges. One needs to take care not to switch a Favourite usage when there is no alternative path available, as this could potentially invalidate the set of routing rules.

4.6 visualization

As both the routing rules and KPIs are essential data for the end-user, both need to be visualized. For this, we use the input graph as the basis for our visualization. For each destination, we can display all routing rules which are affected by this destination. As such, when the user selects a single destination, we can visualize each matching routing rule by coloring all outgoing edges for each divert in the graph. This way, a user can have a quick glance over all routing rules through a familiar representation.

Both KPIs can be visualized by using this same graph representation, but by slightly altering the visualization.

4.6.1 *Colormaps*

One of the most straightforward ways to convey data on large intervals to a user is through the use of coloring. We choose to use a colormap, due to the simplicity of implementing it, and getting effective results in terms of quickly visualizing data.

A colormap defines colors on an interval specified by the user. Specific values within the interval are connected to a color, and the colormap will calculate the appropriate color for any value in between the interval. This is achieved by dividing the interval in smaller intervals, and calculating the correct color for such a subinterval; this result is called a lookup table. This way, we only need a function to map a value to any of those subintervals, and the color is predefined for us; this saves calculation time.

We will show an example using one of the applicabilities in our implementation. We have edges with flow and capacity, and we want to show to the user in what manner the flow adheres to the capacity (i.e. if the flow exceeds capacity). We do this by defining a colormap, which is shown in Figure 28.

We have a colormap ranging from 0% to 125%. The reason behind an interval exceeding 100% is to also be able to indicate how much flow exceeds capacity. Should we cap the interval at 100%, we would not be able to show this. The general idea is that we do not want flow exceeding capacity, but also not lacking the use of the capacity (e.g. an edge with a capacity of 6000 with a flow of only 10 bags). This means that we need to differentiate between the two extremes on our interval. For this, we use a double-ended colormap, ranging from blue to gray to red[Borland, 2007]. Blue indicates overcapacity, red indicates undercapacity (too little flow and too much flow, respectively).

In Figure 28, we also show the division of the main colors red, green and blue. Naturally, we need full blue and red at either end of the colormap, so both colors gradually decrease and increase, respectively, over the course of the interval $[0, 125]$. In order to create gray in the center of the colormap, we gradually increase green to 0.5 intensity towards the center of the interval, and gradually decrease it afterwards.

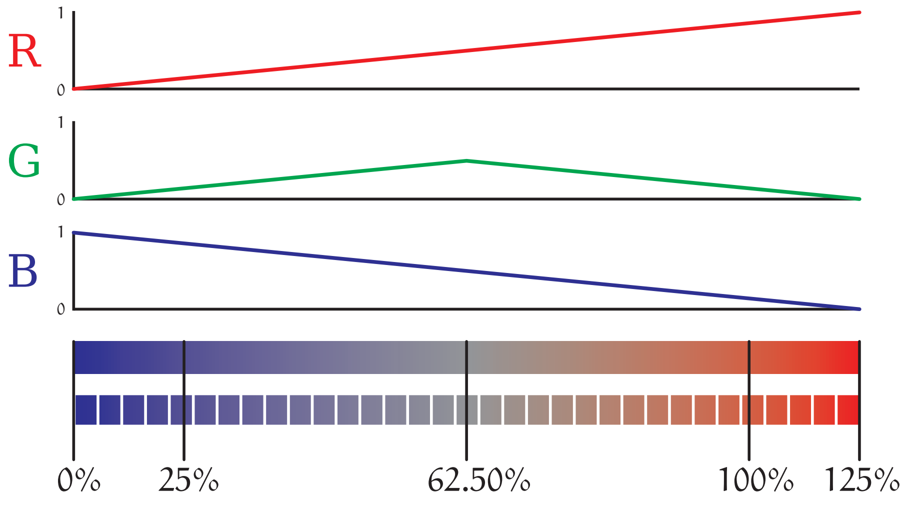


Figure 28: Double-ended blue-gray-red colormap

If we have an interval $i = [i_{min}, i_{max}]$ for which we want to create a colormap, we first need to divide the interval in equal parts for the lookup table. The granularity of this division determines how smooth the transition from one part to the other is. The reason for this is that all values that fall inside a subinterval will have the same color, so naturally, if we have few subintervals we get harsh transitions from one color to the other.

Given a value $s_i \in i$, in order to retrieve a color from the lookup table (with size n) matching s_i, we use the index calculation shown in Equation (11)

$$i = \begin{cases} 0 & s_i < i_{min} \\ n-1 & s_i > i_{max} \\ \left\lfloor (n-1)\left(\frac{s_i - i_{min}}{i_{max} - i_{min}}\right)\right\rfloor & \text{else} \end{cases} \tag{11}$$

Now, in order to calculate the colors that are matching a value $s_i \in i$ in the blue-gray-red colormap, we use Equations (12) to (14).

$$\mathrm{red}(s_i) = \begin{cases} 0 & s_i < i_{\mathrm{min}} \\ 1 & s_i > i_{\mathrm{max}} \\ \frac{s_i - i_{\mathrm{min}}}{i_{\mathrm{max}}} & \mathrm{else} \end{cases} \tag{12}$$

$$\mathrm{green}(s_i) = \begin{cases} 0 & s_i < i_{\mathrm{min}} \\ 0 & s_i > i_{\mathrm{max}} \\ \frac{s_i - i_{\mathrm{min}}}{i_{\mathrm{max}}} & s_i \geqslant i_{\mathrm{min}} \wedge s_i \leqslant \frac{i_{\mathrm{max}} - i_{\mathrm{min}}}{2} \\ 1 - \frac{s_i - i_{\mathrm{min}}}{i_{\mathrm{max}}} & \mathrm{else} \end{cases} \tag{13}$$

$$\mathrm{blue}(s_i) = \begin{cases} 1 & s_i < i_{\mathrm{min}} \\ 0 & s_i > i_{\mathrm{max}} \\ 1 - \frac{s_i - i_{\mathrm{min}}}{i_{\mathrm{max}}} & \mathrm{else} \end{cases} \tag{14}$$

4.6.2 *Flow and capacity*

As said in Section 4.6.1, when we visualize the flow on an edge, we want to visualize the adherence to the capacity of that edge. For this, we introduce the colormap already shown in Figure 28. When we have an edge e, the ratio $\phi = f(e)/c(e)$ indicates the percentual use of the flow on the capacity. When $\phi > 100$, we have more flow than capacity and thus the edge exceeds its capacity.

We want to indicate primarily the edges exceeding their capacity (red), and the edges barely using the capacity given to them (blue). Any edges in between these two extremes are not directly to our interest (gray).

5 ANALYSIS

In this chapter, we will analyse the practical performance of the theory presented in Chapter 3, using the implementation presented in Chapter 4. We will present the datasets of two distinct, real BHSs which we will use for analysis.

5.1 DATASETS

For the analysis we are going to perform, we present two datasets. Each dataset is a complete and operational BHS as used by Vanderlande. Due to confidentiality we cannot disclose any specific information about their locations or what functionality is present, but fortunately that information is also unnecessary for performing such an analysis. The properties of the used datasets (and, additionally, that of the running example), are shown in Table 2.

Table 2: Properties of the analysed datasets

	$\|V\|$	$\|E\|$	Stations	Complexity
Dataset $\mathcal{A}$	133	152	40	Low
Dataset $\mathcal{B}$	1507	1779	385	High
Running example	252	314	55	Medium

One should note that the perceived level of complexity is a mere indication of the amount of technical properties a BHS has. It is not quantifiable, but is only present to give the reader an indication how Vanderlande perceives the BHS in a given dataset. Additionally, the running example will not be analysed, and is presented only as a comparison factor. This, because it is the only way for the user to know how the running example compares to the actual datasets (through the level of complexity). The running example in itself is only used in this thesis for reference; it is infeasible to extend the running example with all relevant data in order to be able to analyse it, as this would require too much work. Naturally, analysing real world examples results in more reliable indications on the applicability of the presented theory, and it should again be noted that all theory in this thesis is based on those real world examples.

5.2 METHODOLOGY

For the analysis, we will use a tool that implements the theory presented in Chapter 3, using the implementation techniques presented in Chapter 4. This tool will also serve as a proof of concept, where we show that the theory is indeed applicable for generating a set of valid and good routing rules.

As mentioned, each dataset represents an actual, operational BHS, and because of this all datasets have an operational set of routing rules. These routing rules have been created manually by Vanderlande employees. We are going to compare these manually created routing rules to those generated by our tool.

There are three ways we are going to compare our generated set of routing rules to those created manually:

1. Difference between the sets based on the matching and mismatching of routing rules;

2. Total number of bottlenecks present; i.e. total number of edges $e \in E$ where the flow exceeds the reduced capacity (overflow), $f(e) > c(e) - c_{\text{redux}}(e)$;

3. Total sum of overflow; i.e. $\sum f(e) - (c(e) - c_{\text{redux}}(e)), \forall e \in E : f(e) > c(e) - c_{\text{redux}}(e)$.

For Case 1, we express the difference between two sets in percentages. However, this measurement does not incorporate the actual performance of a set of routing rules. Even when the generated set of routing rules is completely different, the generated set can still outperform the manual set because it has less bottlenecks or a lower sum of overflow. So, Case 1 is purely for us to have some sort of grasp how the generated set of routing rules compare to the manually created one.

We use Cases 2 and 3 to actually measure the performance of a set of routing rules. Given a certain input flow, we calculate the number of bottlenecks present, and the total sum of these bottlenecks. If we calculate these measurements over a certain interval of input flows, we can see how sensitive a set of routing rules is. We can see when the system starts to give in under the increasing pressure of the input flow, and we can see how large these impacts will be. This will give us a comparison method between sets of routing rules.

For each dataset, we will use a main flow distribution determined by Vanderlande employees as feasible for that corresponding dataset. We regard the input flow of those distributions as the demanded input flow for which we want to minimize the number of bottlenecks. For the sensitivity analysis, we define an interval (which differs per dataset) which starts from the lowest feasible amount of bags in which we have no bottlenecks. We increase the input flow steadily, dividing the increase equally over all input nodes, until we reach the wanted input flow (defined by the main flow distribution) and push past that to see how the system operates at above-capacity levels.

The differences (and similarities) between the manually created routing rules and the (by our tool) generated routing rules will be presented. We will discuss why these difference are present, and in what parts of the datasets they manifest. Furthermore, we will revisit the interesting scenarios presented in Section 2.6 to see how the algorithms cope with those.

5.3 RESULTS

We will discuss the results of each dataset separately. This, due to the functional differences between each dataset; comparing the results between the datasets themselves does not contribute any valuable information.

5.3.1 *Dataset $\mathcal{A}$*

Dataset $\mathcal{A}$ is the smallest dataset we are analysing. It is a fairly straightforward BHS with not a lot of functionality. This means that the set of routing rules is simple, as there are not a lot of possibilities for baggage to traverse the BHS.

5.3.1.1 *Percentage difference*

In Table 3, we present the percentage difference between the (by Vanderlande employees) manually created set of routing rules and our set of generated routing rules. As can be seen, our initial created set of routing rules (before any optimizations) has a difference of 25% compared to the manually created set. Due to the small size of the BHS, there are not a lot of routing rules possible. The main difference can be explained by one destination. The manual set deliberately decides to route baggage for that destination over a loop, whilst the initial set regards that path as too costly and takes a shortcut.

Table 3: Percentage difference between manual and generated sets of routing rules on Dataset $\mathcal{A}$, per optimization iteration

	Initial	Opt #1
Dataset $\mathcal{B}$	25.00	25.00

After one optimization step, we still have the same difference percentage. A few routing rules have been flipped, but the flipped versions are still mismatching compared to the manual set, hence the equal difference. Due to the small size of the dataset, we only have one successful optimization step. Any other optimization step resulted in reduced performance of the set.

5.3.1.2 *Performance analysis*

In Figure 29 (and Table 4) and Figure 30 (and Table 4), we can see the performance analysis on the number of bottlenecks and the summation of overflow present in Dataset $\mathcal{A}$. An input flow of 6000 bags per hour is considered as the demanded input flow. To indicate this, we have marked this with a gray dotted line. A range of analysis runs have been made, starting at an input flow of 1000 bags per hour, increasing it with 1000 bags per run and ending with 8000 bags per hour.

Table 4: Number of bottlenecks presented in Figure 29

	1000	2000	3000	4000	5000	6000	7000	8000
Manual	0	0	1	3	22	35	36	40
Initial	0	0	8	12	25	29	32	34
Opt #1	0	0	1	5	21	28	32	39

Table 5: Sum of overflow presented in Figure 30

	1000	2000	3000	4000	5000	6000	7000	8000
Manual	0	0	494	1593	5761	16768	30269	45174
Initial	0	0	3134	10435	21713	36185	52462	70144
Opt #1	0	0	520	2322	8263	17852	29444	42968

As one can see, our initial set continually has a higher amount of bottlenecks compared to the manual set, until the demanded 6000 bags per hour mark, where it drops below the manual set. Performance under the demanded mark is of higher importance than the performance above it, due to peak loads. The load above the demanded mark will only be present in bursts, and thus will have a short duration. However, the load under the demanded mark will be a more consistent load, hence the duration will be longer. Also, even though the number of bottlenecks drops below that of the manual set, we can see that the sum of overflow is continually far above that of the manual set. So even though our initial set comes fairly close to the manual set, some improvements can still be made.

These improvements become apparent after the first (and only) optimization step is made. During this optimization step, a cross is optimized. This cross is present just after some major input points, result-

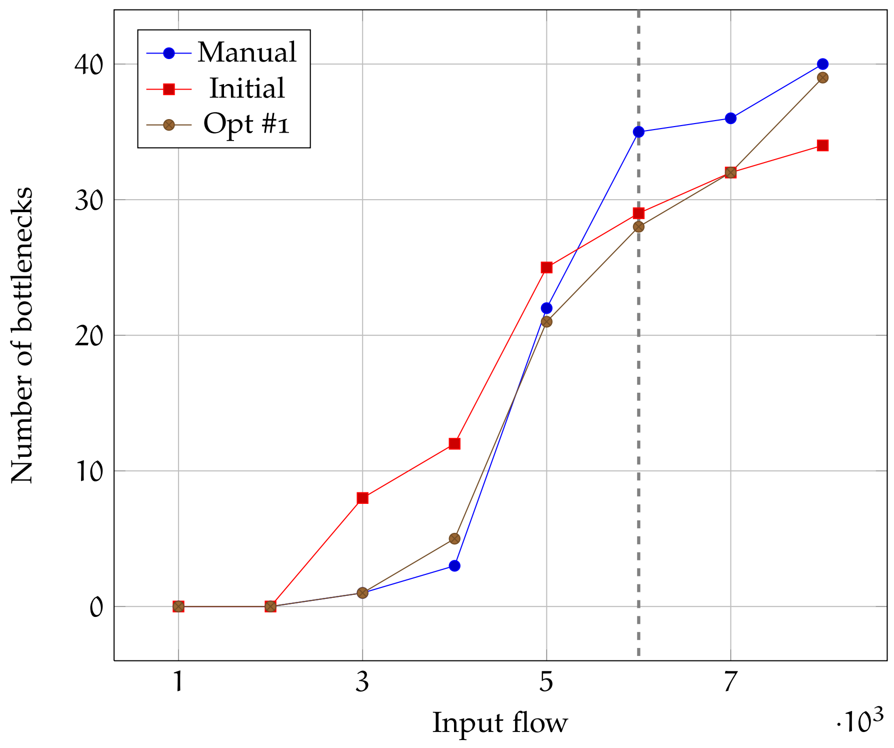


Figure 29: Number of bottlenecks on Dataset $\mathcal{A}$ per different sets of routing rules

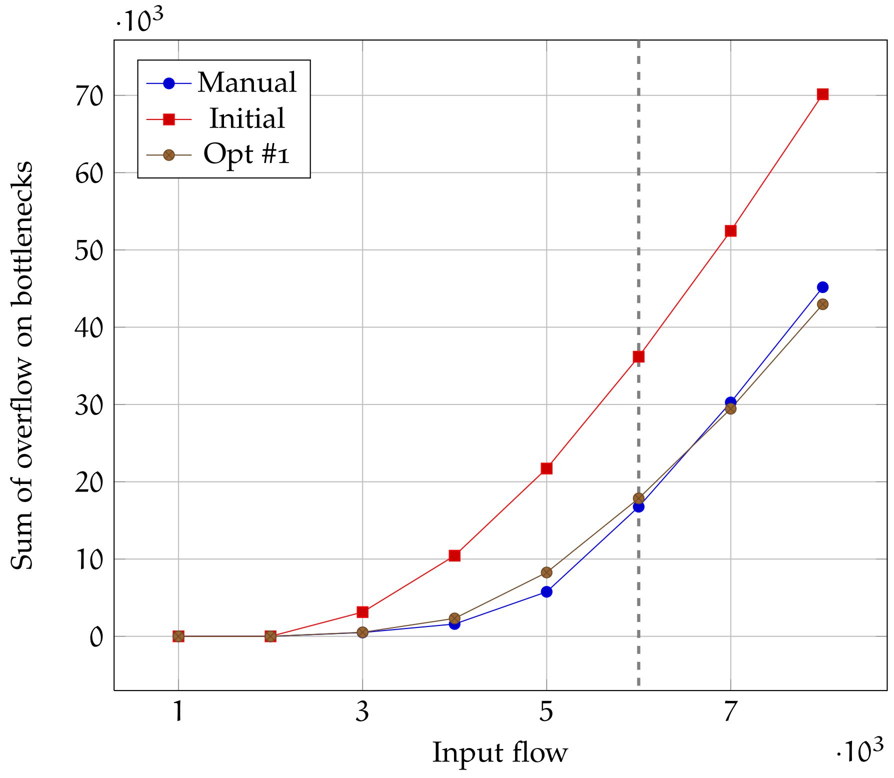


Figure 30: Sum of overflow on bottlenecks on Dataset $\mathcal{A}$ per different sets of routing rules

ing in the cross being a major bottleneck due to inefficient routing. We can see that the sum of overflow of the opt #1 set comes very close to that of the manual set. It is apparent that this bottleneck was the major bottleneck that throttled the entire system, and now that it is solved the system is more capable of handling a high amount of input flow.

One interesting fact is that the opt #1 set still has less bottlenecks than the manual set, as much as 7 distinct bottlenecks less on the demanded input flow. This does not necessarily results in a better performance of the opt #1 set compared to the manual set. Even though the manual set has more bottlenecks, the sum of overflow is the same, which means that on average overflow could be spread more evenly over all present bottlenecks.

5.3.2 *Dataset $\mathcal{B}$*

Dataset $\mathcal{B}$ is substantially more complex than $\mathcal{A}$. It can be seen in Table 2 that $\mathcal{B}$ contains almost ten times more stations than $\mathcal{A}$, and one can imagine that all those stations also need the corresponding edges and paths in order to be able to route to them. As the topology of $\mathcal{B}$ is more common than $\mathcal{A}$, we are very interested in the results of the analysis on this dataset.

5.3.2.1 *Percentage difference*

In Table 6, we present the percentage difference between the manual set of routing rules and the generated set of routing rules. What immediately becomes apparent, is the high difference percentage of 55% in the initial created set of routing rules. This is mainly due to one major feature of this dataset, it has a (highly complex) storage area. This storage area has its own set of tailored routing rules, and these routing rules do not necessarily follow the conventions of the rest of the graph because the area itself is fully automated—it does not work with conveyor belts. After some consideration, this area was regarded as out of scope of this project, due to its complexity. Though, the routing rules for this storage area are present in the manual set. The algorithms do try to take the storage area into account, but the generated routing rules for that area (and all rules leading to that area) will not be complete, and thus we have a huge mismatch between the two sets of routing rules explaining the large percentage difference.

After one optimization step (opt #1), several crosses have been optimized. Similarly as with Dataset $\mathcal{A}$, we had several crosses close to the input points of the BHS. Due to inefficient routing, a lot of baggage was being routed over a single loop (whilst there is more than one loop). Contrary to Dataset $\mathcal{A}$, the modifications in opt #1 did re-

Table 6: Percentage difference between manual and generated sets of routing rules on Dataset $\mathcal{B}$, per optimization iteration

	Initial	Opt #1	Opt #2
Dataset $\mathcal{B}$	55.9	54.02	54.00

sult in routing rules now identical to those in the manual set, thus we get a difference decrease of 1.88 percentage point.

After the second optimization step (opt #2), a few diverts leading to loops have been optimized. Originally, all diverts routed baggage to the same loop, whereas there are two loops present. One loop went over capacity, whilst the other loop still had some capacity left. The optimization algorithm flipped half of the diverts over to the other loop, resulting in similar behaviour as the manual set. However, since we are only talking about 8 edges in a dataset containing 1779 edges, the difference percentage only decreases by 0.02 percentage point.

5.3.2.2 *Performance analysis*

In Figure 31 (and Table 7) and Figure 32 (and Table 7), we can see the performance analysis on the number of bottlenecks and the summation of the overflows present in Dataset $\mathcal{B}$. An input flow of 11000 bags per hour is considered as the demanded input flow. To indicate this, we have marked this with a gray dotted line. A range of analysis runs have been made, starting at an input flow of 3000 bags per hour, increasing it with 1000 bags per run and ending with 14000 bags per hour.

Contrary to the initial set of $\mathcal{A}$, the initial set in $\mathcal{B}$ continually performs worse than the manual set, both in terms of number of bottlenecks as well as the sum of overflow. This is mainly due to the much larger scale of $\mathcal{B}$ compared to $\mathcal{A}$. It is much harder for the creation algorithm to immediately generate a good performing set of routing rules, even when the amount of input flow is low.

Two noticeable peaks are present, both in the number of bottlenecks as the sum of overflow, at an input flow of 10000 and 12000 bags per hour. The first peak (10000 bags per hour) can be explained by the saturation of screening stations. Their capacity is reached at 9000 bags per hour, the additional 1000 bags per hour increase results in all ingoing routes clogging up. This remains steady until 12000 bags, when a major bottleneck increase is noticeable. This is the result of the entire output of the system buckling under the extremely high input flow. Two major flows are being combined just before the laterals, resulting in a lot of overflow on these lines.

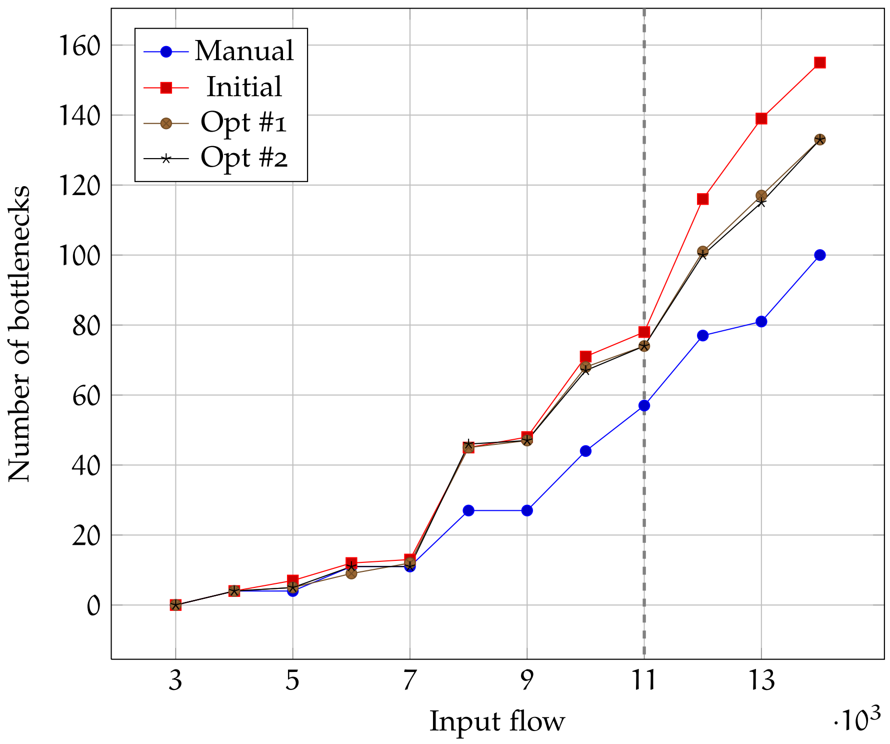


Figure 31: Number of bottlenecks on Dataset $\mathcal{B}$ per different set of routing rules

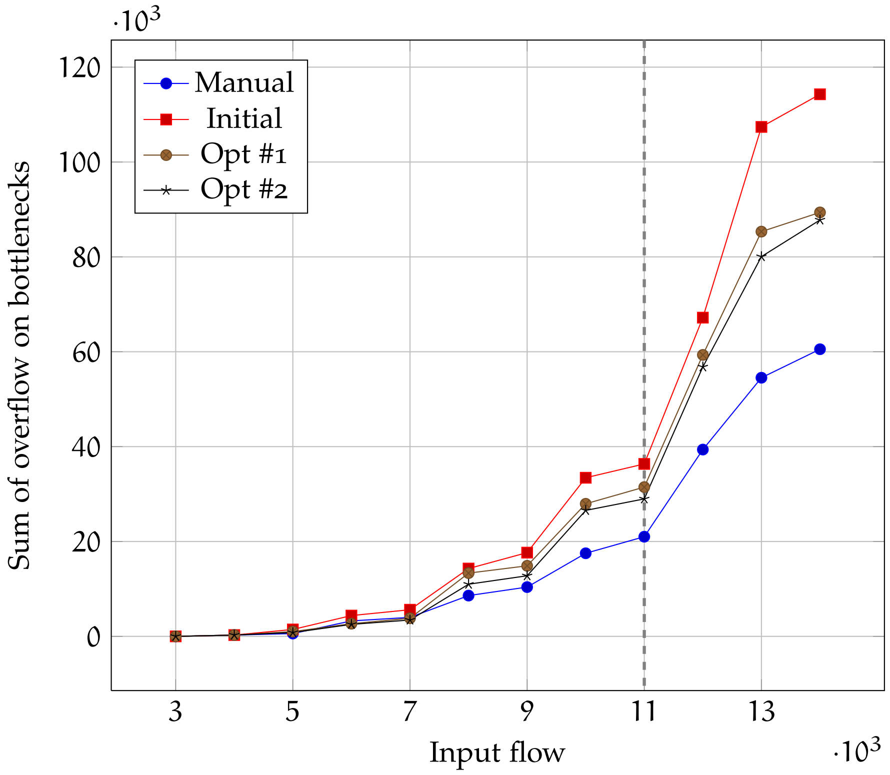


Figure 32: Sum of overflow on bottlenecks on Dataset $\mathcal{B}$ per different sets of routing rules

Table 7: Number of bottlenecks presented in Figure 31

	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000
Manual	0	4	4	11	11	27	27	44	57	77	81	100
Initial	0	4	7	12	13	45	48	71	78	116	139	155
Opt #1	0	4	5	9	12	45	47	68	74	101	117	133
Opt #2	0	4	5	11	11	46	47	67	74	100	115	133

Table 8: Sum of overflow presented in Figure 32

	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000
Manual	0	272	592	3257	4004	8594	10386	17514	21010	39378	54523	60532
Initial	0	272	1455	4383	5615	14296	17653	33437	36349	67192	107396	114280
Opt #1	0	272	1002	2666	3819	13332	14871	27946	31468	59336	85327	89376
Opt #2	0	272	802	2525	3446	10992	12774	26553	28978	56808	80058	87780

After the first optimization step (opt #1), multiple cross bottlenecks are optimized, similar to the cross bottleneck present in the initial set of $\mathcal{A}$. However, in $\mathcal{B}$, multiple crosses are present, and all are inefficiently routed in the initial set. We will elaborate on the changes made on the cross in Section 5.4.1. With the changes, load is now more evenly directed to one of the loops. Because the loops themselves have high capacities, the performance improvements are only noticeable when the input flow nears the demanded input flow (i.e. when it gets high).

After the second optimization step (opt #2), diverts leading to loops have been optimized. Again, we will elaborate on the changes in Section 5.4.2. What can be seen is that the number of bottlenecks in opt #1 and opt #2 are roughly the same. The diverts that have been changed are already going over capacity after 9000 bags (with and without the optimizations), and since the number of bottlenecks do not make a distinction between small or large overflows, the number of bottlenecks does not change. However, if we look at the sum of overflows, we can see that the bottlenecks themselves have been reduced by around 20% at 9000 bags per hour and above, compared to the initial set.

5.4 INTERESTING SCENARIOS IN PRACTICE

We have shown interesting scenarios for our theory in Section 2.6, and paid special attention to solving those interesting scenarios during our approach in Chapter 3. In this section, we will revisit those interesting scenarios using the results we have retrieved through the analysis on datasets $\mathcal{A}$ and $\mathcal{B}$.

5.4.1 *Cross*

In Dataset $\mathcal{B}$, we had a crossing problem similar to the interesting scenario presented in Section 2.6.1. In Figure 33, we can see the routing rules present before and after optimization. In Figure 34, we can see the same cross visualized and annotated with the remaining capacity. After we push flow through, we have a certain amount of remaining capacity, i.e. for an edge $e \in E$ the reduced capacity minus the flow: $(c(e) - c_{redux}(e)) - f(e)$. If this number is negative, we have more flow going over the edge than the reduced capacity can take. In other words: we have overflow, and thus a bottleneck. The coloring is the direct result of the colormap we have described in Section 4.6.1.

As can be seen in Figure 33a, the routing rules originally routed all traffic over the Favourite edges through the lower merge. The downside of this routing can be seen very clearly in Figure 34a. Both edges going into the lower merge are red hot, both with an overflow of almost 700 bags per hour. The upper merge has almost no traffic to process, resulting in a very idle merge node with remaining capacities almost identical to the actual capacity of 1800 bags per hour.

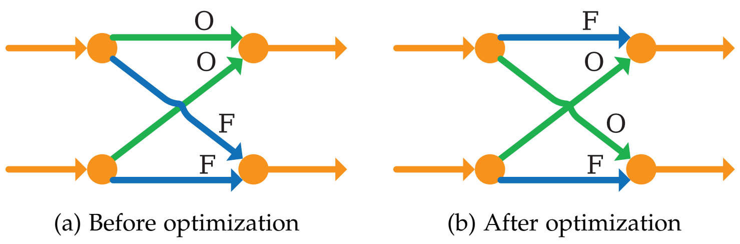


Figure 33: Optimization of routing rules on a cross

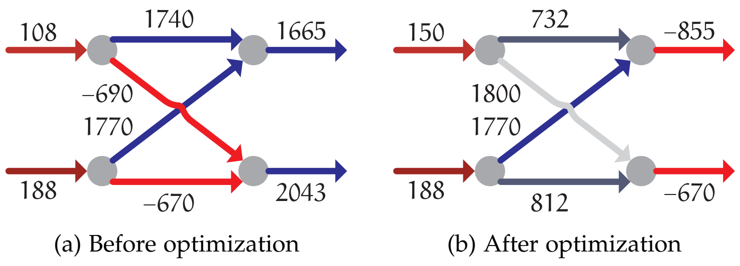


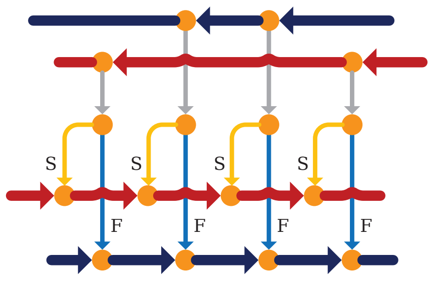
Figure 34: Remaining capacity of routing rules on a cross

After the actual optimization step, we can see the routing rules have been flipped over, resulting in the situation shown in Figure 33b. We now send flow over their dedicated paths instead of the redundant paths, and the result of this is very clearly shown in Figure 34b. The redundant path is relieved of its stress, and is actually no longer used (we have no flow going over it, hence why the remaining capacity is equal to its actual capacity). Because the lower merge no longer has to process a lot of bags, the lower dedicated path is also no longer a bottleneck.

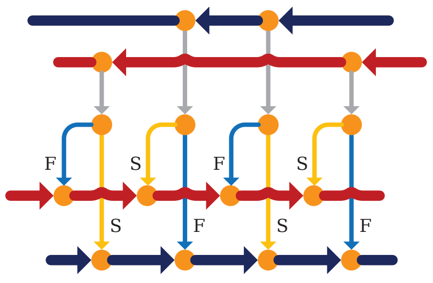
The input flow is different, due to this cross not being the only one in the system that is optimized in the same optimization step (because of bottleneck equivalency). The outgoing flow can now pose as a problem, because of its negative remaining capacity, but this can be due to an other bottleneck downstream, or due to an input flow which is too high.

5.4.2 *Loops*

In Dataset $\mathcal{B}$, we had a scenario similar to the interesting scenario presented in Section 2.6.2. Diverts that were leading to loops were routing all baggage to one loop, instead of spreading the load over multiple loops. In Figure 35, we have created a similar situation in order to explain how the optimization algorithm dealt with this.



(a) Before optimization



(b) After optimization

Figure 35: Optimization of routing rules on loops

In Figure 35a, we can see four diverts all leading to either of the two main loops. However, all diverts have routing rules resulting in baggage only being routed towards the blue loop. This resulted in bottlenecks on the blue loop, due to the high flow of baggage that was being routed over it. After the optimization step (done in opt #2), half of the routing rules were flipped over to the red loop, resulting in the situation displayed in Figure 35b. This reduced the bottlenecks on the blue loop, and increased the flow over the red loop. This reduced the overall sum of overflow, as is apparent in Figure 32.

5.4.3 *Crossing of flow*

What became apparent during all optimization iterations (both successful and unsuccessful iterations) was the difficulty of determining crossing of flow and actually solving it. The 'crossing of flow' example we presented in Section 2.6.3 is a very simple and straightforward one, where we have a single point of failure on the merging of multiple main flows.

In Dataset $\mathcal{B}$, we have one major issue near the output where multiple main flows are being merged. Because there are multiple bottlenecks present, it is hard for the optimization algorithm to exactly pinpoint the direct cause of those bottlenecks. This results in many failed optimization attempts due to lack of information.

The primary reason for this lack of information is the use of stateless flow. Whenever we have a bottleneck, we only know that the bottleneck exists at that point, not how it is formed. We do not have any history on the flow upstream, and how those decisions might have influenced the bottleneck currently present. Should we have stateful flow, we could use the history of flow to extract useful knowledge about upstream behavior. This could result in better optimization techniques in order to prevent crossing of flow.

5.5 EVALUATION

With the results presented in Section 5.3, and the analysis of interesting scenarios in Section 5.4, we can now evaluate the performance of the algorithms, and ultimately, the theory itself.

When we look at the initial sets of routing rules generated, it is apparent that already a lot of the correct routing rules are generated. With a difference of 25% in Dataset $\mathcal{A}$ and 56% in Dataset $\mathcal{B}$, we already prevent the manual creation of 50% of all routing rules. This, in itself, is already a major improvement regarding the amount of man-hours needed for creating the same amount of routing rules manually.

On average, creating a full set of routing rules for BHSs as big as Dataset $\mathcal{B}$ takes around two months of man-hours[1]. Even if we assume that the initial set only contains the most straightforward routing rules, we could reduce this by two to two-and-a-half weeks of man-hours.

The optimization techniques presented in Section 3.5.2 further increase the performance of sets of routing rules by 15% to 20% on Datasets $\mathcal{A}$ and $\mathcal{B}$. Two of the three interesting scenarios are indeed recognized and taken into account when regarding bottlenecks, effectively reducing the bottlenecks or completely resolving them. The remaining interesting scenario (crossing of flow) appears difficult to resolve due to lack of information, resulting in only the capability of resolving bottlenecks caused by local routing problems.

Additionally, the performance of the set of routing rules do not only rely on the efficiency of the set, but also on the efficiency of the task topology. Because baggage flow is simulated through the information extracted out of the task topology, the task topology itself has a great influence on how baggage is pushed through the BHS. If the task topology is inefficient, the routing rules will appear inefficient as well, though, this may not necessarily be the case.

1 This is an estimation made by Vanderlande employees

6 CONCLUSION

In this thesis we have introduced the problem of generating routing logic for Baggage Handling Systems (BHSs). Routing baggage through BHSs is done making use of pre-defined routing rules, which indicate for each decision point what the possible next step should be, in order to get the bag to its (intermediate) destination. These routing rules are currently created manually at Vanderlande, and the goal of this graduation project was to automatically generate a valid (i.e. complete) and good (i.e. high performance) set of routing rules.

In order to generate these routing rules, we introduced a framework capable of doing so. This framework consists of three processes: the creation process, the analysis process and the optimization process. The creation process generates an initial set of routing rules through a modified shortest path algorithm, by taking into account all the possibilities of the BHS. This initial set only focusses on baggage, and disregards the entire BHS. This is why we make use of the analysis process to analyse this (or an arbitrary) set of routing rules. The analysis process takes into account mechanical properties of the BHS, and uses flow algorithms to simulate baggage flowing through the BHS. The analysis process produces possible bottlenecks that are the direct result of inefficient routing. Using these bottlenecks as a basis, the optimization process tries to resolve the bottlenecks by modifying the routing rules possibly responsible for the bottlenecks.

We analysed the performance of this entire framework by comparing all generated sets of routing rules with those created manually, on datasets from actual BHSs. The initial sets generated by the creation algorithm already contain 50% to 75% of all manual routing rules. This already decreases the number of man-hours needed to create a complete set of routing rules by around one-third. The performance of these initial sets are roughly two-thirds worse than the manual set. After applying optimization steps, the optimization techniques we presented further increased the performance by an additional 15% to 20%.

Further optimizations are still possible, but impossible without introducing stateful flow instead of stateless flow. Due to the latter being used in this graduation project, information on the history of flow is missing. This makes it difficult to accurately predict the exact cause of bottlenecks whenever these bottlenecks are not the cause of local routing inefficiencies.

6.1 FUTURE WORK

Optimization processes are almost always a never ending process. One has to determine the tradeoff between the amount of effort and the results achieved. The existence of this project proves that with some abstraction, one can still provide reasonable arguments on the performance of a set of routing rules. However, further optimization is always possible, and in this section we will provide methods for this. We will show how we can improve the processes we have described in this thesis, but also discuss the possibilities of taking into account a broader scope for the project itself.

6.1.1 *Influence rules*

In Section 1.5.2, we discussed the notion of influence rules. How they are present to take over whenever something in a BHS becomes unavailable (either through breakdowns, high loads, or other reasons). They change routing rules temporarily in order to reroute baggage in such a way that the unreachable area is circumvented.

We deliberately chose to leave out the calculation of influence rules in this project. Even though the calculations themselves are quite straightforward (trivial, even), analysing all those influence rules and taking them into account when optimizing is quite complex. This was simply infeasible for the duration of this graduation project. However, we can elaborate on the possible ways to calculate these influence rules.

For each possible breakdown, one has to translate this breakdown into a set of edges $E_d \subset E$ that will be disabled as an effect of the breakdown. As such, in effect we have a new graph $G_d = (V, E \setminus E_d)$ on which we run the initial creation algorithm. Each edge in E_d will not exist for the creation algorithm, and as such it will not take them into account. The resulting initial set of routing rules will thus route all baggage around the disabled set of edges E_d. If we take the difference between the initial set of routing rules on G and that on G_d, we will have our set of influence rules which temporarily take over whenever all edges in E_d are disabled due to a breakdown.

The analysis and optimization algorithm can still be used on that set of routing rules for G_d. Though, two assumptions we made in Section 3.3.4 cannot be guaranteed anymore. We do not know for certain that the topology is complete, and we do not know for certain that we can route through non-destination nodes. This would endanger the validity of the routing rules that are in effect when the influence rules take over. However, one can state that all breakdowns which can occur always make it possible for baggage to be rerouted as Vanderlande also incorporates this when designing a BHS.

6.1.2 *Take into account holding capacity*

Edges have an additional property we can use when analysing a set of routing rules. Besides the actual capacity of an edge, indicating the amount of bags it can process per hour, there is also the notion of holding capacity. This indicates the physical capacity of the conveyor belt which the edge represents, defining the maximum number of bags that can reside on the edge itself. This introduces the actual length of a conveyor belt into the equation.

Using the holding capacity, we can distinguish between a physically long and short edge. If we have two edges $e, e' \in E$ with an equal capacity of $c(e) = c(e') = 1500$ bags per hour, but e has a holding capacity of 10 bags, whilst e' has a holding capacity of 500 bags, the short term impact of a mutual edge downstream breaking down is much more severe compared to the breakdown of edge e'. One can see the holding capacity as a buffer: if it is low, there is not much buffer space and the edge will clog up fast if it needs to temporarily store baggage.

We can extend the capacity reduction equation which was shown in Section 3.4.6.1. For example, we can decrease the capacity reduction whenever the holding capacity, defined by $c_{\text{hold}} : E \to \mathbb{R}^{+}$, increases:

$$c_{\text{redux}}(e) = \frac{1}{c_{\text{hold}}(e)} \cdot \#\text{switches}(n) \cdot 3 \cdot \frac{c(e)}{3600} \tag{15}$$

However, the impact of the holding capacity can be too steep if used as in Equation (15). This is something that should be investigated, and adjusted accordingly.

6.1.3 *Introduce stateful flow*

As described in Section 3.4.5.5, we have deliberately chosen for stateless flow in our approach. A possible extension on this work would be to replace this stateless flow with stateful flow. For this, one would have to keep track per destination flow the previously visited nodes. This way, once we have a bottleneck in a BHS, more information can be retrieved on the possible flows that were responsible for this bottleneck.

For example, assume we have a bottleneck node. When using stateful flow, we can backtrack through the list of passed nodes for each destination flow. With these lists of passed nodes, we can check if there are multiple nodes present in history lists of different destination flows. Should we have such a node present, we can try to shift the routing rules for that node around. In more direct terms: we can directly pinpoint the common ancestor for multiple bottlenecks, and here we have the most change of successfully altering the routing rules such that we resolve the bottleneck.

This also makes it easier to detect whether we have main flows crossing through the system. As was already evaluated in Section 5.5, history on destination flows are necessary in order to make optimizations based on non-local bottleneck problems.

6.1.4 *Incorporate Process Controller logic*

In this project, the LM and the PC are considered as separate entities, only communicating with each other the necessary information. We assume we receive the correct information from the PC (where, in the main flow, a bag needs to go) and react on that. The logic of the PC, i.e. how it reacts on the position of a bag, or the conditions of the system, can be extracted and captured in logic rules.

These logic rules of the PC can then be combined with the methodology presented in this thesis to create more realistic routing rules. In this project, we only know the possible destinations for each point in the system, by using the task topology. If we incorporate the logic rules from the PC in this project, we can use this logic to generate rules tailored on these possible decisions. We achieve this because we increase the knowledge about the entire BHS using these logic rules. We can now know what the possible future steps for a bag are, at a certain position. Currently, we only know the current step a bag has to undertake, not its possible future steps.

This knowledge can be valuable for generating routing rules, because we can look into creating the best possible path for a bag to take from its first destination to its final destination. We can take into account where a bag has been, and more importantly, where it needs to go. This way, we can try to generate routing rules which allows for a more seamless flow from input to output points for each bag.

6.2 RELATED WORK

This graduation project uses many aspects of graph theory and other fields of research. There are many past and possible future research topics closely related to this graduation project, and in this section we will discuss these research topics. Both in and out of the scope of this graduation project.

6.2.1 *Realtime shortest path calculation in simulations*

In 2001, a previous graduation project was performed at Vanderlande[de Jongh, 2001]. The goal of this project was to investigate if it is possible to calculate routing logic in a realtime way for environments, whilst also taking into account balancing. The similarities to this current graduation project are high, with the exception that the graduation project by de Jongh was focussed on actual simulation models, whereas this graduation project focusses on pre and post design models.

The conclusion of the graduation project was the infeasibility (at that time) of realtime calculation of routing logic, as it would take too much time to do so. Moreover, errors in the created routing logic was present, as it was only using a shortest path calculation methodology. This is also the reason why in this current graduation project we do not only rely on the shortest path methodology for creation routing rules, but also on optimizing these rules. Finally, several load balancing methods were considered, but none of them were found to be optimal for the systems Vanderlande uses.

6.2.2 *Retrieve correlated destinations*

As mentioned in Section 2.5, the task topology enables the LM to determine the possible destinations to which it can route a bag, given the bag's current position in the system. These sets of destinations (station groups) are predefined in the task topology, i.e. correlated destinations are combined manually into sets. Originally, the use of these station groups were to ease the work of the system architect; this way he does not have to redefine a destination multiple times for many positions in the system, but he can rather define a whole group of destinations and use this group over and over. However, through this project it became apparent that the functionality of these station groups is not only useful for the system architects, but they also provide valuable information on the relationship between destinations, which we have used as well.

Currently, these station groups are created through the insights of a system architect. Usually, destinations are grouped whenever they are all considered equally whenever handing out a task which they all perform. E.g. when it does not significantly matter which destination we choose, they are all feasible.

This process of creating station groups can also be automated. For this, one should look into methodologies for finding correlated nodes in a graph. After some research, it seems that the notion of assortative mixing could be a primary candidate for solving this problem[Newman, 2003]. Through assortative mixing, assortativity can be applied for each combination of nodes, expressing the attraction between those nodes. If one uses an assortative function which is able to express the similarity between destinations, one can create a new graph composed of similar destination nodes. Through this graph, it is then possible to apply graph clustering techniques in order to retrieve sets of similar destinations which could be translated into station groups[Schaeffer, 2007].

6.2.3 *Generate task topology*

The task topology we use (explained in Section 2.5) is also manually created by Vanderlande employees. It is based on the possible decisions that the PC can make, and takes the topology of the BHS into account in order to know what destinations would be most efficient given any point in the system.

This task topology can also be generated. The process for doing this could be complex enough to create an other graduation project. One could again use the BHS as a graph, but through the calculation of reachability graphs indicate what destinations are possible at any point in the system. These reachability graphs can be used as a basis for clustering techniques in order to determine what station groups would be most efficient to hand out at a given point in the system. This information can be sufficient to generate a basic task topology which experts can use as a starting point whenever a new BHS needs to be implemented.

6.2.4 *Route logic in other fields*

Defining routing through graphs and networks is a broadly researched topic within graph theory, as it adds a layer of complexity where one needs to take into account actually reaching the destination within the graph, just as we also needed to do in this graduation project. Routing logic can be applied on any graph with the appropriate data, hence the interest not only in autonomous systems like Vanderlande's BHS, but also in car navigation technology[Flinsenberg, 2004; Duckham and Kulik, 2003], with added complexity by applying turn restrictions[Winter, 2002] and even when considering electric distribution networks[Nguyen et al., 2010].

A BHS is a static collection of conveyor belts, they do not dynamically alter such that extra routing possibilities can appear. This is similar to a network of roads. One could apply the theory presented in this thesis for generating and optimizing the flow of cars in a city. The layout of the city can be represented as a graph with roads as edges and intersections as nodes. If we apply turn restrictions, we can also prohibit cars from making illegal turns. The capacity and travel time of roads can be mapped in a similar way as we did with conveyor belts. Major destinations can be marked, and subsequently we can calculate an optimal routing schema.

For normal routing behaviour a city can be too complex for the theory presented, due to the large diversity of destinations each car has. However, in case of special events in a city, a substantial amount of cars will have the same (small set of) destination(s). This way, we can calculate the most optimal routing to get all cars in and out of the city in the most efficient way.

BIBLIOGRAPHY

Borland, D. Rainbow Color Map (Still) Considered Harmful. *Computer Graphics and Applications*, 27(2):14–17, 2007.

de Jongh, S. Dynamic Routing and Balancing in Material Handling Systems. Master's thesis, Delft University of Technology, 2001.

Denic, E.A. Algorithm for Solution of a Problem of Maximum Flow in Network with Power Estimation. *Soviet Mathematics Doklady*, 11: 1277–1280, 1970.

Dijkstra, E.W. A Note On Two Problems In Connexion With Graphs. *Numerische Mathematik*, 1:269–271, 1959.

Duckham, M. and Kulik, L. "Simplest" Paths: Automated Route Selection for Navigation. *COSIT*, 2825:169–185, 2003.

Edmonds, J. and Karp, R.M. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19: 248–264, 1972.

Eppstein, D. Finding the K Shortest Paths. *Siam Journal on Computing*, 28(2):652–673, 1998.

Flinsenberg, I.C.M. *Route Planning Algorithms for Car Navigation*. PhD thesis, Eindhoven University of Technology, 2004.

Ford, L.R. and Fulkerson, D.R. Maximal Flow Through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1955.

Ford, L.R. and Fulkerson, D.R. *Flows in Networks*. Princeton University Press, 1962.

Fredman, M.L. and Tarjan, R.E. Fibonacci Heaps and Their Uses in Improved Network Optimization Problems. *Journal of the ACM*, 34 (3):596–615, 1987.

Fulkerson, D.R. An Out-of-Kilter Method for Minimal-Cost Flow Problems. *Journal of the Society for Industrial and Applied Mathematics*, 9:18–27, 1961.

Gabow, H.N. Scaling Algorithm for Network Problems. *Journal of Computer And System Sciences*, 31(2):148–168, 1985.

Goldberg, A.V. and Tarjan, R.E. A New Approach to the Maximum-Flow Problem. *Journal of the ACM*, 35:921–940, 1988.

Goldberg, A.V. and Tarjan, R.E. Finding Minimum-Cost Circulations by Successive Approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

Karzanov, A.V. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Mathematics Doklady*, 15:49–52, 1974.

Korte, B. and Vygen, J. *Combinatorial Optimization: Theory and Algorithms*. Springer, 2000.

Newman, M. E. J. Mixing Patterns in Networks. *Physical Review Letters*, 89(20), 2003.

Nguyen, P.H., Kling, W.L., Georgiadis, G., Papatriantafilou, M., Tuan, L.A, and Bertling, L. Distributed Routing Algorithms to Manage Power Flow in Agent-based Active Distribution Network. In *Innovative Smart Grid Technologies Conference Europe (ISGT Europe), 2010 IEEE PES*, pages 1–7, Oct 2010.

Schaeffer, S.E. Survey: Graph Clustering. *Computer Science Review*, 1 (1):27–64, 2007. ISSN 1574-0137.

Shiloach, Y. and Vishkin, U. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

Sleator, D.D. *An $O(nm \log n)$ Algorithm for Maximum Network Flow*. PhD thesis, Stanford University, Stanford, CA, USA, 1981.

Sleator, D.D. and Tarjan, R.E. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

Sleator, D.D. and Tarjan, R.E. Self-adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

Winter, S. Modeling Costs of Turns in Route Planning. *GeoInformatica*, 6(4):345–361, 2002.

Yen, Y. Finding the K-Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, Juli 1971.

A PSEUDOCODE

In this chapter we will provide the pseudocode accompanying Chapter 4. We have divided the chapter in the same sections and subsections as present in Chapter 4, with the same section titles.

We will provide no further explanation per algorithm, as they are already discussed in Chapter 4. It should be noted that, due to readability, the pseudocode we show only the most relevant parts needed for implementation. In any implementation, practical issues will arise that are not relevant for the theory itself, and as such we have left them out here as well. The pseudocode in this section is fairly complex and concise, and we suggest one to be very familiar with the theory presented in this thesis before implementing based on this pseudocode.

A.1 INITIAL CREATION ALGORITHM

A.1.1 *Finding best path*

Algorithm 1 Retrieve the best path between source s and target t

```
function BestPath(s, t)
    $w_{tot} \leftarrow 0$
    bestPath $\leftarrow \emptyset$
    for all $v \in V$ do
        $v$.HeapNode $\leftarrow$ null
        $v$.Previous $\leftarrow$ null
        $v$.Marking $\leftarrow$ Unlabeled
    end for
    heaps $\leftarrow \emptyset$
    heaps.Insert(s, 0)
    while heaps.MinimumNode $\neq$ null do
        Scan
    end while
    if t.PreviousNode $\neq$ null then
        b $\leftarrow$ t
        while b $\neq$ s do
            bestPath $\leftarrow$ (b.Previous,b) $\cup$ bestPath
            $w_{tot} \leftarrow w_{tot}$ + Cost((b.Previous,b))
            b $\leftarrow$ b.Previous
        end while
    end if
    return {bestPath, $w_{tot}$}
end function
```

Algorithm 2 Expand the searching from the minimum node in a heap

```
function Scan
    $m \leftarrow heaps$.MinimumNode
    $m$.Marking $\leftarrow$ Scanned
    $heaps$.DeleteMin()
    for all $e \in \delta^+(m) : \neg e$.Disabled do
        if $t(e) = null$ then
            $t(e)$.HeapNode $\leftarrow heaps$.Insert($t(e)$,$m$.Key + Cost($e$))
            $t(e)$.PreviousNode $\leftarrow m$
        else
            if $t(e)$.Marking $\neq$ Scanned
                        $\wedge$ $m$.Key + Cost($e$) $<$ $t(e)$.Key then
                $heaps$.DecreaseKey($m$.Key + Cost($e$), $t(e)$)
                $t(e) \leftarrow$ Labeled
                $t(e)$.PreviousNode $\leftarrow m$
            end if
        end if
    end for
end function
```

Algorithm 3 Return the cost of an edge $e = (v, w)$

```
function Cost(($v, w$))
    if $w \in D \wedge w \neq t$ then
        return $\infty$
    else
        return $(v, w)$.TravelTime
    end if
end function
```

A.1.2 *Creating initial set of routing rules*

Algorithm 4 Generate all valid routing rules

```
1: RoutingRules ← $\emptyset$
2: for all $d \in D$ do
3:     $tg \leftarrow t \in$ TaskGroups : $e \in \delta^{-}(d)$ : e.TaskGroup $= t$
4:     $T_p \leftarrow$ tg.Destinations                         ▷ All possible destinations
5:     for all $t \in T_p$ do
6:         path ← BestPath(d,t)
7:         if path $\neq \emptyset$ then
8:             RoutingRules[s(path), t] ← Favourite
9:             $w_t \leftarrow$ path.Cost
10:            s(path).Disabled ← true
11:            for all $e_o \in \delta^{+}(d) : e_o \neq s(path)$ do
12:                altPath ← BestPath(d,t)                ▷ Calculate alternative paths
13:                if altPath $\neq \emptyset \wedge$ altPath.Cost$/w_t < 1.10$ then
14:                    if altPath $\in$ loops then
15:                        RoutingRules[s(altPath), t] ← Substitute Favourite
16:                    else
17:                        RoutingRules[s(altPath), t] ← Option
18:                    end if
19:                end if
20:                s(altPath).Disabled ← true
21:            end for
22:        end if
23:    end for
24: end for
```

A.2 ANALYSIS ALGORITHM

A.2.1 *Calculating flow through a (sub)graph*

Algorithm 5 Initialize graph $G = (V, E)$

```
1: $V \leftarrow V \cup \{s, t\}$                                   ▷ Create source and sink node
2: for all $v \in V : \delta^-(v) = \emptyset$ do                  ▷ All input nodes
3:     $E \leftarrow E \cup (s, v)$
4: end for
5: for all $v \in V : \delta^+(v) = \emptyset$ do                  ▷ All output nodes
6:     $E \leftarrow E \cup (v, t)$
7: end for
8: for all $e \in E$ do
9:     e.Flow $\leftarrow 0$
10: end for
11: for all $e \in \delta^+(s)$ do
12:     e.Capacity $\leftarrow$ inputFlow/$|\delta^+(s)|$
13:     e.Flow $\leftarrow$ inputFlow/$|\delta^+(s)|$
14: end for
15: for all $v \in V$ do
16:     $v$.Label $\leftarrow 0$
17:     $v$.Excess $\leftarrow \emptyset$
18: end for
19: $s$.Label $\leftarrow |V|$
20: for all $e \in \delta^+(s)$ do
21:     $tg \leftarrow e$.TaskGroup
22:     $T_p \leftarrow tg$.Destinations                            ▷ All possible destinations
23:     for all $t_p \in T_p$ do                                  ▷ Create flowsets for each possible destination
24:         $t(e)$.Excess $\leftarrow t(e)$.Excess $\cup$ $(t_p, e.\text{Flow}/|T_p|)$
25:     end for
26: end for
```

Algorithm 6 Discharge

```
1: $Q \leftarrow V$
2: while $Q \neq \emptyset$ do
3:     $v \leftarrow Q[0]$
4:     while $v$.Excess $\neq \emptyset$ do
5:         if $v$.Label $> |V|$ then
6:             return fail                                       ▷ Recirculating flow present
7:         end if
8:         PushRelabel($v$)
9:     end while
10: end while
```

Algorithm 7 Push relabel on v

```
1:  $e \leftarrow v$.CurrentEdge
2:  $w \leftarrow w \in e \wedge w \neq v$                         ▷ Get the other node in $e$
3:  wIsActive $\leftarrow$ IsActive($w$)
4:  if ApplicableForPush($e$) then Push($e$)
5:  else
6:      if $v$.CurrentEdge.Next $\neq \emptyset$ then
7:          $v$.CurrentEdge $\leftarrow v$.CurrentEdge.Next
8:      else
9:          $v$.CurrentEdge $\leftarrow v$.CurrentEdge[0]
10:         $v$.Label++
11:     end if
12: end if
13: if $\neg$wIsActive $\wedge$ IsActive($w$) then        ▷ $w$ was not active, but is now
14:     $Q \leftarrow Q \cup w$
15: end if
```

Algorithm 8 Applicability of push for $e = (v, w)$

```
1:  if IsActive($v$) $\wedge e$.Capacity $> 0$ then
2:      if $v \notin D \vee |\delta^+(v)| = 1$ then
3:          return false
4:      else
5:          other $\leftarrow 0$
6:          for all $fs \in v$.Excess do
7:              if RR[$e, fs$.Destination] = Favourite then
8:                  return true
9:              else
10:                 for all $e_o \in \delta^+(v) : e_o \neq e$ do
11:                     if RR[$e_o, fs$.Destination] = Favourite then
12:                         other $\leftarrow$ other + $fs$.Amount
13:                     end if
14:                 end for
15:             end if
16:         end for
17:         if $e$.TransportDefault then
18:             if other $= \sum_{fs \in v.\text{Excess}} fs$.Amount then
19:                 return false
20:             else
21:                 return true
22:             end if
23:         end if
24:     end if
25: end if
26: return false
```

Algorithm 9 Push operation on $e = (v, w)$

```
1:  balance ← false
2:  if $v \notin D \vee |\delta^+(v)| = 1$ then
3:      if $v \notin \text{loops} \wedge w \in \text{loops}$ then
4:          balance ← true
5:      end if
6:      $e.\text{Flow} \leftarrow \sum_{v.\text{Excess}} fs.\text{Amount}$
7:      $w.\text{Excess} \leftarrow w.\text{Excess} \cup v.\text{Excess}$
8:      $v.\text{Excess} \leftarrow \emptyset$
9:  else
10:     for all $fs \in v.\text{Excess}$ do
11:         if RR[$e$, $fs$.Destination] = Favourite
12:                                 $\wedge(d_v = \text{null} \vee d_v = v)$ then
13:             if $v \notin \text{loops} \wedge w \in \text{loops}$ then
14:                 balance ← true
15:             end if
16:             $e.\text{Flow} \leftarrow e.\text{Flow} + fs.\text{Amount}$
17:             $w.\text{Excess} \leftarrow w.\text{Excess} \cup \{fs\}$
18:             $v.\text{Excess} \leftarrow v.\text{Excess} \setminus \{fs\}$
19:         else
20:             if $e$.TransportDefault then
21:                 if $\neg\exists e_o \in \delta^+(v) : e_o \neq e \wedge \text{RR}[e_o, fs.\text{Destination}] = \text{Favourite}$ then
22:                     $e.\text{Flow} \leftarrow e.\text{Flow} + fs.\text{Amount}$
23:                     $w.\text{Excess} \leftarrow w.\text{Excess} \cup \{fs\}$
24:                     $v.\text{Excess} \leftarrow v.\text{Excess} \setminus \{fs\}$
25:                 end if
26:             end if
27:         end if
28:     end for
29: end if
30: if balance then
31:     BalanceFlow($w$)
32: end if
```

Algorithm 10 Balance flow on $e = (v, w)$

```
1:  balanceFlows ← $\{fs \mid fs \in v.\text{Excess} \wedge fs.\text{Via} = \text{null}\}$
2:  balanceDiverts ← $\{d \mid d \in D \wedge d \in e.\text{StatusEdge}\}$
3:  $v.\text{Excess} \leftarrow \emptyset$
4:  for all $fs \in$ balanceFlows do
5:      $D_b \leftarrow \{d_b \mid d_b \in$ balanceDiverts
6:                  $\wedge \exists e_o \in \delta^+(d_b) : \text{RR}[e_o, fs.\text{Destination}] = \text{Favourite}\}$
7:      for all $d_b \in D_b$ do
8:          $fs' \leftarrow (fs.\text{Destination}, fs.\text{Amount}/|D_b|, d_b)$
9:          $v.\text{Excess} \leftarrow v.\text{Excess} \cup \{fs'\}$
10:     end for
11: end for
```

A.2.2 *Indicating bottlenecks*

Algorithm 11 Switch ratio for a node $v \in V$

```
if $v \in D$ then
    if $\exists e \in \delta^+(v)$ : e.TransportDefault then
        return 0
    else
        return $\min(\delta^+(v).\mathrm{Flow})/\max(\delta^+(v).\mathrm{Flow})$
    end if
else if $v \in M$ then
    if $v \in$ loops then
        $e_{main} \leftarrow e \in \delta^-(v) : e \in \mathrm{loops}$
        $e_{in} \leftarrow e \in \delta^-(v) : e \notin \mathrm{loops}$
        $cap \leftarrow \max(\delta^+(v).\mathrm{Capacity})$
        return $|(e_{in} + (e_{main}/cap)^2 \cdot e_{in}))/(cap - e_{main})|$
    else
        return $\min(\delta^-(v).\mathrm{Flow})/\max(\delta^-(v).\mathrm{Flow})$
    end if
end if
```

Algorithm 12 Capacity reduction for a node $v \in V$

```
if $v \in D$ then
    $f_m \leftarrow \min(\delta^+(v).\mathrm{Flow})$
    $e_i \leftarrow \exists^1 e \in \delta^-(v)$
    redux ← CalcCapRedux($f_m$, $e_i$.Capacity)
    $e_i$.CapacityRedux ← redux
else if $v \in M$ then
    if $v \in$ loops then
        $e \leftarrow \{e \mid e \in \delta^-(v) \land e \notin \mathrm{loops}\}$
        $f_m \leftarrow \min(\delta^-(v).\mathrm{Flow})$
        redux ← CalcCapRedux($f_m$, e.Capacity)
        e.CapacityRedux ← redux
    else
        for all $e \in \delta^-(v)$ do
            $f_m \leftarrow \min(\delta^-(v).\mathrm{Flow})$
            redux ← CalcCapRedux($f_m$, e.Capacity)
            e.CapacityRedux ← redux
        end for
    end if
end if
function CalcCapRedux(flow, capacity)
    return $\mathrm{flow} \cdot 3 \cdot \mathrm{capacity}/3600$
end function
```

A.3 OPTIMIZATION ALGORITHM

Algorithm 13 Flip routing rules for diverts $D_f \subseteq D$

```
1:  $D_h \leftarrow \{d \in D_f \mid |D_h| = \lceil \frac{|D_f|}{2} \rceil\}$
2:  for all $d \in D_h$ do
3:      $T_{pos} \leftarrow$ PossibleDestinations(d)
4:      $T_s \leftarrow T_s \subseteq T_{pos} \implies \neg\exists T_s' \subseteq T_{pos} : |T_s'| > T_s$
5:      $e_c \leftarrow e_c \in \delta^+(d) \implies \neg\exists e' \in \delta^+(d) : f(e') - c(e') - c_{redux}(e') >$
6:                  $f(e_c) - c(e_c) - c_{redux}(e_c)$
7:      if $T_s$ = null then
8:          return                                                    ▷ Dry node
9:      end if
10:     for all $t \in T_s$ do
11:         if $Usages[e_c, t] \neq$ "" then
12:             if $\exists e_o \in \delta^+(d) \setminus e_c : Usages[e_c, t] \neq$ "" then
13:                 oldUsage $\leftarrow Usages[e_c, t]$
14:                 $Usages[e_c, t] \leftarrow Usages[e_o, t]$
15:                 $Usages[e_o, t] \leftarrow oldUsage$
16:             end if
17:         end if
18:     end for
19: end for
```