Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering

DOCTORAL THESIS

Algorithms for processing of large data sets using distributed architectures and load balancing

Prague 2020 Ing. Ondřej Šubrt

Bibliografický záznam

Autor:	Ing. Ondřej Šubrt
	České vysoké učení technické v Praze
	Fakulta jaderná a fyzikálně inženýrská
	Katedra softwarového inženýrství
Název práce:	Algoritmy pro zpracování velkých objemů dat s využitím distribuovaných architektur a vyvažování zátěže
Studijní program:	Aplikace přírodních věd
Studijní obor:	Matematické inženýrství
Školitel:	Ing. Tomáš Liška, Ph.D.
	České vysoké učení technické v Praze
	Fakulta jaderná a fyzikálně inženýrská
	Katedra softwarového inženýrství
Akademický rok:	2019/2020
Počet stran:	214
Klíčová slova:	COMPASS, systém pro sběr dat, knihovna DIALOG, DAQ Debugger, vyvažování zátěže, dynamické programování, genetický algoritmus, zpětnovazební učení

Bibliographic Entry

Author:	Ing. Ondřej Šubrt
	Czech Technical University in Prague
	Faculty of Nuclear Sciences and Physical Engineering
	Department of Software Engineering

Title of Dissertation:	Algorithms for processing of large data sets using distributed architectures and load balancing
Degree Programme:	Applications of Natural Sciences
Field of Study:	Mathematical Engineering

Supervisor:	Ing. Tomáš Liška, Ph.D.
	Czech Technical University in Prague
	Faculty of Nuclear Sciences and Physical Engineering
	Department of Software Engineering

Academic Year:	2019/2020
Number of Pages:	214
Keywords:	COMPASS, data acquisition system, DIALOG library, DAQ Debugger, load balancing, dynamic programming, genetic algorithm, reinforcement learning

Abstrakt

Moderní experimenty ve fyzice vysokých energií kladou veliké nároky na spolehlivost, efektivitu a rychlost přenosu dat systému pro sběr dat (DAQ). Tato disertační práce se zaměřuje na stabilitu inteligentního systému pro sběr dat založeného na FPGA (iFDAQ) na experimentu COMPASS v CERN. iFDAQ čerpá události vytvořené na úrovni hardwaru a je navržen tak, aby umožňoval čtení dat při maximální rychlosti přenosu dat z experimentu. Vylepšení stability iFDAQ se skládá z několika různorodých úkolů. Nejprve je prezentována nová komunikační knihovna DIALOG pro meziprocesovou komunikaci. Dále byl vyvinut DAQ Debugger sloužící k detekci chyb v iFDAQ. Stabilní iFDAQ dává příležitost k implementaci neptřetržitě běžícího módu pro iFDAQ běžícího 24/7 bez jediného zastavení. Nakonec je řešen problém vyvažování zátěže v iFDAQ pomocí dynamického programování (DP), hladové heuristiky (GH), celočíselného programování (ILP), genetického algoritmu (GA) a zpětnovazebního učení (RL).

Abstract

Modern experiments in high energy physics impose great demands on the reliability, the efficiency, and the data rate of Data Acquisition System (DAQ). This thesis focuses on the stability of the intelligent, FPGA-based Data Acquisition System (iFDAQ) of the COMPASS experiment at CERN. The iFDAQ utilizing a hardware event builder is designed to be able to readout data at the maximum rate of the experiment. The iFDAQ stability improvement consists of several various tasks. Firstly, the new communication library DIALOG for the Inter-Process Communication (IPC) is presented. Secondly, the DAQ Debugger is developed to help with the iFDAQ error detection. Then, the stable iFDAQ gives an opportunity to implement the iFDAQ continuously running mode running 24/7 without any stops. Finally, Load Balancing (LB) of the iFDAQ is solved using Dynamic Programming (DP), Greedy Heuristic (GH), Integer Linear Programming (ILP), Genetic Algorithm (GA) and Reinforcement Learning (RL).

Contents

List of Abbreviations

Abbreviation	Interpretation
BMS	Beam Momentum Station
CASTOR	CERN Advanced STORage manager
CERN	European Organization for Nuclear Research Conseil Européen pour la Recherche Nucléaire
COIN-OR	Computational Optimization Infrastructure for Operations Research
COMPASS	COmmon Muon Proton Apparatus for Structure and Spectroscopy
DAQ	Data AcQuisition system
DATE	Data Acquisition and Test Environment
DC	Drift Chamber
DE	Differential Evolution
DHC	Data Handling Card
DIALOG	Distributed, Inter-process, Asynchronous, Library, Open, General
DIM	Distributed Information Management
DP	Dynamic Programming
DVCS	Deeply Virtual Compton Scattering
DY	polarized Drell-Yan process
ECAL	Electromagnetic CALorimeter
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GEM	Gas Electron Multiplier
GH	Greedy Heuristic
HCAL	Hadronic CALorimeter
IDE	Integrated Development Environment
iFDAQ	intelligent, FPGA-based Data AcQuisition system
ILP	Integer Linear Programming
IPC	Inter-Process Communication
LB	Load Balancing
LP	Linear Programming
MDE	Modified Differential Evolution

Abbreviation	Interpretation
MUX	MUltipleXer
RICH	Ring Imaging Cherenkov counter
RL	Reinforcement Learning
SA	Simulated Annealing
SciFi	Scintillating Fibre
SIGABRT	SIGnal ABoRT
SIGFPE	SIGnal Floating-Point Exception
SIGILL	SIGnal ILLegal instruction
SIGSEGV	SIGnal SEGmentation Violation
SLC	Scientific Linux CERN
SPS	Super Proton Synchrotron
TCS	Trigger Control System

Table 1: List of abbreviations.

Nomenclature

Symbol	Interpretation
$\mathbb{N}$	natural numbers
$\mathbb{N}_0$	natural numbers with zero
$\mathbb{Z}$	the set of integers
$\mathbb{R}$	real numbers
$\mathbb{R}^+$	positive real numbers
$n, m \in \mathbb{N}$	the natural number n, m
$m \in \mathbb{N}$	the number of MUXes
$p \in \mathbb{N}$	the number of ports in each MUX
$n \in \mathbb{N}$	the number of flows, i.e., $n = m \cdot p$
f_i	the i-th flow
$\mathcal{S}_i$	the subset of flow indices allocated to the i-th MUX
F	the theoretical average flow
$\lvert\mathcal{S}\rvert$	the number of elements in the set $\mathcal{S}$
$\lfloor x \rfloor$	the floor function, i.e., $\lfloor x \rfloor = \max\{z \in \mathbb{Z} \mid z \leq x\}$
$\lceil x \rceil$	the ceiling function, i.e., $\lceil x \rceil = \min\{z \in \mathbb{Z} \mid z \geq x\}$
$\varphi(i, \mathcal{S})$	the index function
$\mathcal{R}_i$	the i-th reduced set
w_i	the weight of the i-th item in the Knapsack problem
v_i	the value of the i-th item in the Knapsack problem
W	the capacity of a knapsack in the Knapsack problem
x_i	the assignment of i-th item in the Knapsack problem
W_i	the capacity of the i-th knapsack in the m–Knapsack problem
$\mathcal{R}_i^{\mathrm{w}}$	the reduced set of weights for the i-th knapsack in the m–Knapsack problem
$\mathcal{R}_i^{\mathrm{v}}$	the reduced set of values for the i-th knapsack in the m–Knapsack problem
$\mathcal{U}, \mathcal{U}_1, \mathcal{U}_2$	the initial set, first subset, second subset, respectively of positive integers in the Partition problem
$\langle \cdot \rangle$	the encoding of an input
$\{0,1\}^*$	the set of all binary strings
A	the decision problem A
x	the input x

Symbol	Interpretation
ϕ	the algorithm ϕ
ψ	the verifier ψ
$\|x\|$	the length of the string x (number of bits)
$p(x)$	the polynomial $p(x)$
$A \leq_{\mathcal{P}} B$	the polynomial-time reduction from A to B
$\mathcal{P}$	the complexity class $\mathcal{P}$
$\mathcal{NP}$	the complexity class $\mathcal{NP}$
$\mathcal{NP}$-complete	the complexity class $\mathcal{NP}$-complete
$\mathcal{NP}$-hard	the complexity class $\mathcal{NP}$-hard
$O(\cdot)$	Big O notation
F_n	the n-th member of the Fibonacci sequence
$m[i,w]$	the maximum value that can be attained with weight less than or equal to w using items up to i (first i items)
$m_i[j,w,k]$	the optimal solution when the capacity of the i-th knapsack is w, only the first j items are considered and it is not allowed to put more than k items into the i-th knapsack
x_i, y_i	the decision variable
a_{ij}, e_{ij}	coefficients
b_i	the right-side coefficient
c_i, d_i	the objective coefficients
z	the objective function
$\mathbf{x}$	the solution
z^*	the optimal objective value
$\mathbf{x}^*$	the optimum solution
$\mathcal{M}$	the feasible set
$\mathcal{L}$	the set of active problems
r_j	the reward obtained by the allocation of the j-th flow
x_{ij}	the allocation of the j-th flow to the i-th MUX
$\mathbf{x}$	the vector representing one individual
rand()	the generation of a random number in the range [0,1]
rand(j)	the j-th valuation of function rand() function
s	the counter of iteration of MDE
$s_{\max}$	the maximum number of iteration of MDE
$f(\cdot)$	fitness function of MDE
N_{p}	the number of individuals in each population of MDE
β	the mutation factor
CR	the crossover rate
T	the temperature
α	the rate of reducing the temperature
c_1, c_2, k_1, k_2	constants for MDE
ε	the exploration parameter
α	the learning parameter

Symbol	Interpretation
γ	the discount factor
π	the policy
Π	the policy space
π^*	the optimum policy
x_k	the state at $stage_k$
$\mathcal{X}$	the state space
$\mathcal{X}_k$	the subspace of state space at $stage_k$
a_k	the action at stage $stage_k$
a^{g}	the greedy action
$\mathcal{A}$	the action space
$\mathcal{A}_k$	the subspace of action space at $stage_k$
a'	the action from the next action subspace $\mathcal{A}_{k+1}$
$f(x_k, a_k)$	the function of state transition
$g(x_k, a_k, x_{k+1})$	the reward function
r_k	the reward received for the transition from $stage_k$ to $stage_{k+1}$
$Q(a_k)$	the performance index of action a_k in the N–arm Bandit problem
a^*	the best action (the best arm) in the N–arm Bandit problem
$Q(x, a)$	the Q value of state action pair (x, a)
$\hat{Q}^s(x, a)$	the estimated Q value of state action pair (x, a) in s-th iteration
$Q^*(x, a)$	the optimum Q value of state action pair (x, a)
s	the counter of iteration in learning phase of RL
$s_{\max}$	the maximum number of iterations in learning phase of RL
F_k^{A}	the flow to be allocated at $stage_k$
F_k^{T}	the total flow which has already been allocated in the previous $k-1$ stages
F_k^{Tmin}	the minimum possible total flow already allocated in the previous $k-1$ stages
F_k^{Tmax}	the maximum possible total flow already allocated in the previous $k-1$ stages
$a_k^{\min}$	the minimum value of action a_k
$a_k^{\max}$	the maximum value of action a_k
p^{A}	the number of flows already allocated
$L_{u,v}$	the sum of u smallest flows in last v stages

Table 2: Nomenclature.

Introduction

In general, a state-of-the-art Data Acquistion System (DAQ) in high-energy physics experiments must satisfy high requirements in terms of reliability, efficiency and data rate capability. Utilizing a hardware event builder, the intelligent, FPGA-based Data Acquisition System (iFDAQ) of the COMPASS experiment at CERN is designed to be able to readout data at the average maximum rate of 1.5 GB/s of the experiment.

The thesis is organized as follows. In the first chapter, the thesis gives an introduction to CERN and, especially, to the COMPASS experiment.

The description of iFDAQ is stated in Chapter 2. A very detailed overview of the iFDAQ from the hardware and software point of view is given, followed by a figure of the COMPASS iFDAQ topology which put all views together.

In complex softwares, such as the iFDAQ, having tens of processes communicating with each other, the Inter-Process Communication (IPC) is absolutely essential to satisfy correct synchronization and proper data taking. The DIALOG library is designed and implemented to meet all necessary requirements, especially on high-performance, reliability and robustness. It was fully incorporated to all processes in the iFDAQ during the Run 2016 and improved the stability of iFDAQ significantly.

Chapter 3 presents the IPC. Section 3.1 deals with the design of the DIM library. It gives a description and deeper insight into the DIM library. The motivation for development of a new communication library follows which concludes the previously used communication approach. Section 3.2 is concerned with the implementation of the DIALOG library. It presents all requirements, gives description, integration, robustness and implementation domains. The important subsection is called "Scenarios" discussing all exemplary situations. Section 3.3 deals with the Online Monitoring API for the DIALOG library. It presents how to easily develop own monitoring tools and gives a few examples of monitoring tools. In Subsection 3.3.1, the online monitoring of communication among processes via DIALOG GUI is stated. The DIALOG GUI allows a visualization of all processes involved in the application. The final section, Section 3.4, presents the efficiency measurement and performance of the DIM and DIALOG library.

In the iFDAQ running nonstop 24/7 for most of the calendar year, it is absolutely essential to detect a source of system crashes and problems. The DAQ Debugger encapsulates such features and creates detailed reports concerning system crashes.

The iFDAQ Debugging is presented in Chapter 4. Firstly, a basic concept of con-

ventional debugging is summarized in Section 4.1. The motivation for the DAQ Debugger implementation follows in Section 4.2 and the state-of-the-art error reporting tools are discussed in Section 4.3. Finally, the DAQ Debugger is described in Section 4.4. In order to conclude how the DAQ Debugger is important, the iFDAQ stability over the last few years is shown in Section 4.5.

The improved stability opens up a possibility to keep the system continuously running without interruption of data flow. This feature reduces time consuming synchronization phases at each start and stop of a run. Chapter 5 describes the continuous operation mode of the iFDAQ, the necessity of proper timing and synchronization, and the logic in the affected processes.

In complex systems, such as the iFDAQ, Load Balancing (LB) of data flow is absolutely essential to satisfy a proper data taking. The LB algorithms are designed and implemented to meet all necessary requirements, especially on high-performance, reliability and robustness. LB aims to optimize resource use, maximize throughput and avoid overload of any single resource.

The chapter concerning LB is Chapter 6. The chapter starts with the explanation of the LB problem and at the end, it gives five approaches how to solve it.

Section 6.1 deals with the proper definition of the LB problem. It gives a description and deeper insight into the LB problem. The definition of mathematical terms being useful in next subsections follows which concludes the introduction part to the LB problem. The LB problem complexity is mentioned in Section 6.2.

The first approach how to solve the LB problem is based on Dynamic Programming (DP). Section 6.3 presents the DP approach in more detail. It begins with a short introduction to DP and builds gradually the algorithm solving the LB problem.

Greedy Heuristic (GH) is the second approach given in Section 6.4. GH typically goes through a sequence of steps, with a set of choices at each step, and always makes the choice that looks best at the moment.

Integer Linear Programming (ILP), being a mathematical optimization in which some or all of the variables are restricted to be integers, follows in Section 6.5.

The fourth approach is based on genetic algorithms. Section 6.6 describes Genetic Algorithms (GA) and, especially, Differential Evolution (DE). It follows with the proposal of the Modified Differential Evolution (MDE) meeting requirements of the LB problem. All parts of the MDE algorithm are discussed in an extensive way.

Finally, the last approach, Reinforcement Learning (RL), being the study of how animals and artificial systems can learn to optimize their behaviour in the face of rewards and punishments, is given in Section 6.7.

Several Test Cases are performed in Chapter 7 to demonstrate how particular approaches are successful and efficient in the LB problem solving and to compare them with each other. There are defined three test cases – the Test Case 1, Test Case and Test Case 3 – and all of them are solved using DP, GH, ILP, MDE and RL, respectively. Then, a discussion follows emphasizing the pros and cons of each approach.

Chapter 1

The COMPASS Experiment

COMPASS (COmmon Muon Proton Apparatus for Structure and Spectroscopy) [1] is a high-energy particle physics experiment with fixed-target situated on the M2 beamline of the Super Proton Synchrotron (SPS) particle accelerator at CERN laboratory in Geneva, Switzerland. In Figure 1.1, the COMPASS experiment is located in the North Area. The scientific program of the COMPASS experiment was approved in 1997. The goal was to study the structure of gluons and quarks and the spectroscopy of hadrons using high intensity muon and hadron beams. By the year 2010 the experiment entered it's second phase COMPASS-II [3]. The COMPASS-II program started with a physics run for the study of the polarized Drell-Yan (DY) process in the years 2014 and 2015 followed by a run dedicated to Deeply Virtual

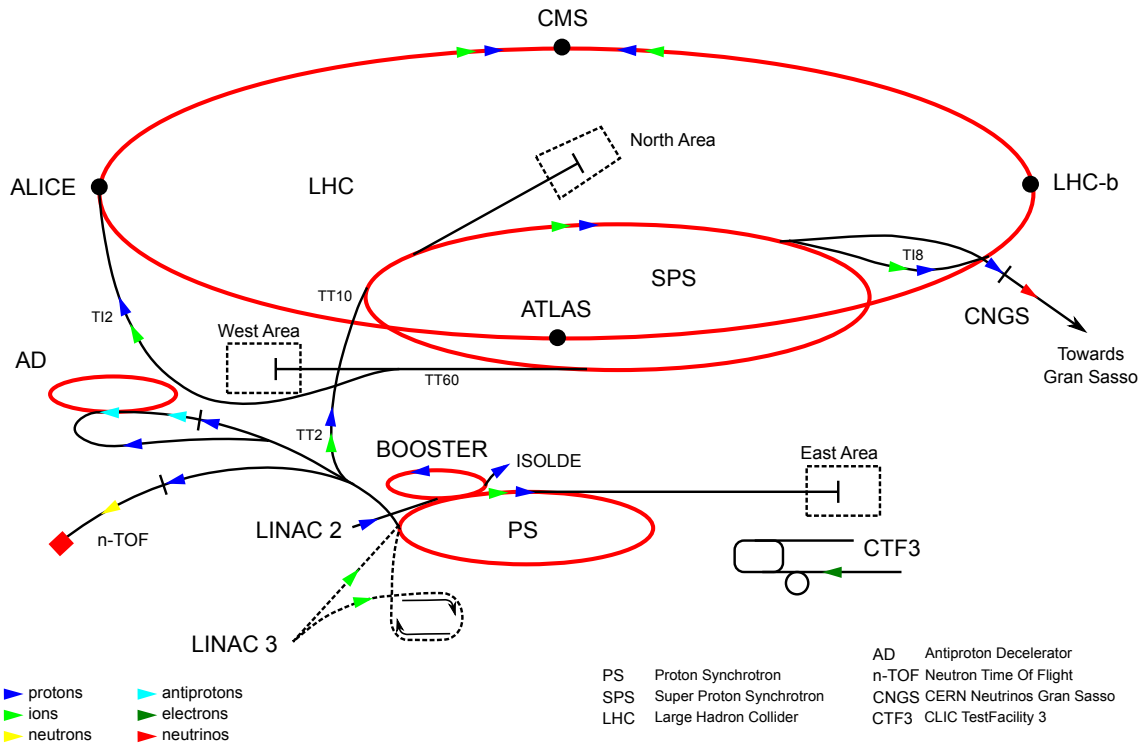

Figure 1.1: COMPASS location within the CERN accelerator complex.

Compton Scattering (DVCS).

The COMPASS spectrometer is designed to perform particle identification, tracking and measurement of its energy. For these purposes the spectrometer is fitted with:

- Ring Imaging Cherenkov Counters (RICH) or muon filters for particle identification
- Gas Electron Multipliers (GEM), Scintillating Fibres (SciFi) and Drift Chambers (DC) for particle tracking
- Hadronic or Electromagnetic Calorimeters (HCAL, ECAL) for calorimetry measurements

Moreover, another detectors are placed along the beamline before the spectrometer to measure momentum and position of the beam (e.g. Beam Momentum Station, BMS). The exact layout of the spectrometer depends on the scientific program and it is still evolving. For illustration, an artistic view on the spectrometer, which is over 60 meters long, composition is shown in Figure 1.2.

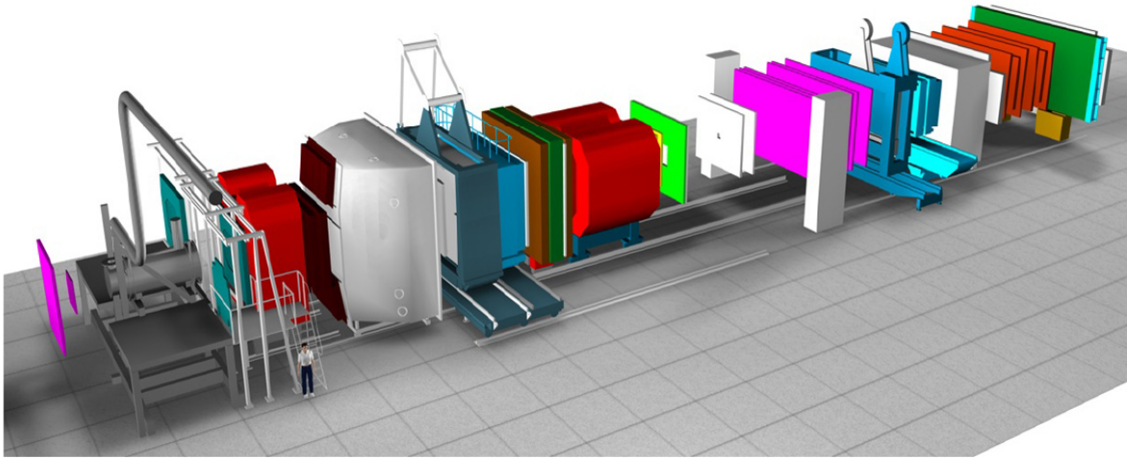

Figure 1.2: COMPASS experimental setup.

During the previous years, it had a usual data rate of approximately 1.5 GB/s during approximately 10 seconds on-spill with the off-spill time between 30 and 50 seconds, depending on SPS super cycle. The original DAQ of the experiment was built between 1999 and 2001. The Data Acquisition and Test Environment (DATE) software [4], originally developed for the ALICE at CERN, was used to control DAQ and event building in old system. Both software package and usage of FPGA-based cards have been widely studied and as the result a design of the new iFDAQ was prepared.

Chapter 2

The iFDAQ Architecture

The COMPASS experiment is in operation since 2002. Since then, the amount of collected data is steadily increasing. The major growth was caused by improvement of the beam intensity and increase of trigger rates and it is supposed to continue to rise in future. Over the years, the electronics of the DAQ was upgraded several times in order to be able to handle such amount of data. Furthermore, the hardware upgrades were getting more and more complicated due to obsolete technology. Consequently, the COMPASS collaboration has decided for the considerable iFDAQ improvement during the shutdown in 2013/2014. Nowadays, the final part of hardware and software replacement finishes.

2.1 Hardware Part

The iFDAQ of the COMPASS experiment consists of several layers [6, 8, 7, 5, 57, 58]. The frontend electronics that form the lowest layer continuously preprocess and digitize analogue data from the detectors. There are approximately 300,000 detector channels; trigger rate can rise up to 50 kHz with 36 kB average event size. SPS accelerator operates in cycles that consist of 10 second long period with beam (called spill) followed by approximately 40 second period without beam. Data from multiple channels are readout and assembled by the concentrator modules called CATCH [14], HGeSiCA [17], and GANDALF [16]. These modules receive signals from the time and trigger system; when the trigger signal arrives, the readout is performed and data are sent over optical connection S-Link [19] to the following layer that is based on special FPGA DHC (Data Handling Card) cards. It is further divided into two layers and is responsible for building of complete events. It comprises eight FPGA (multiplexers (MUXes)) and handles another level of multiplexing. S-Links are also used to connect the first sublayer to the second sublayer which is made up of a single DHC with switch firmware (SWITCH) – this layer handles event building.

This newly designed event building part allows usage of more compact control system. The hardware event builder performs online verification of data consistency. The last layer of the system consists of eight readout engine computers equipped

with spillbuffer cards that readout assembled events and transfer them to the CERN permanent storage (CASTOR) [13].

That is a theoretical description of the iFDAQ full setup. In Figure 2.1, the current state – used in the Run 2016, 2017 and 2018 – is given. It consists of only six FPGA cards (MUXes) on the level of multiplexing and four readout engine computers.

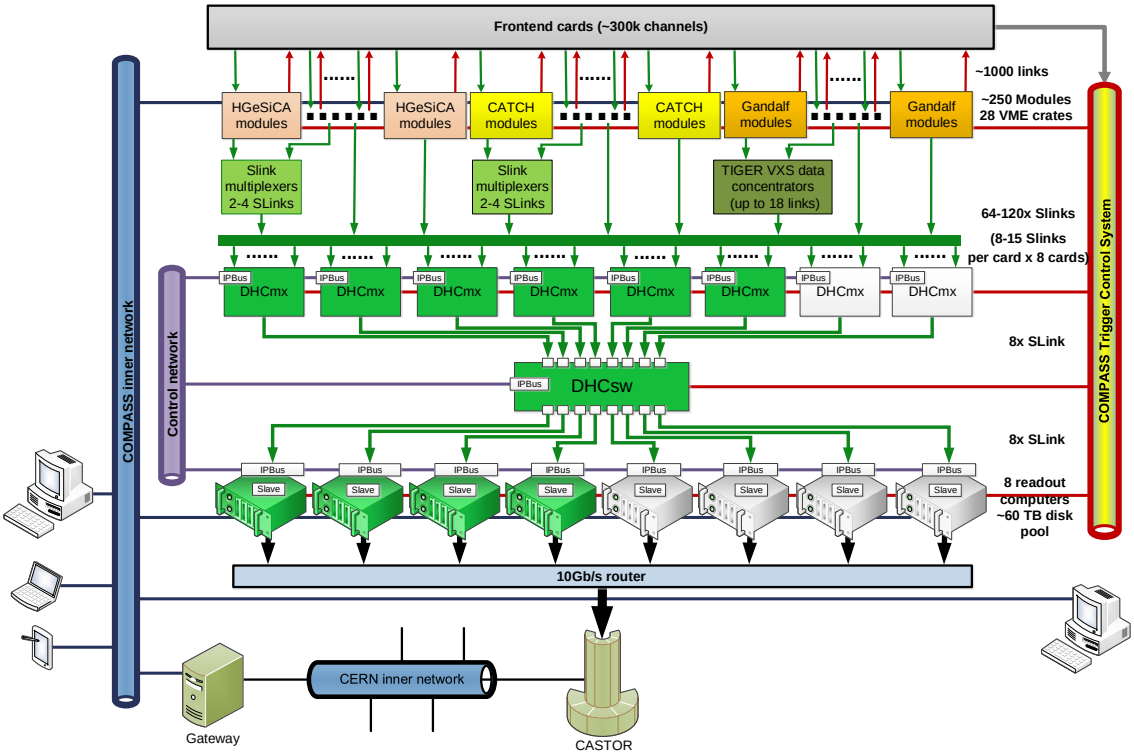

Figure 2.1: The COMPASS iFDAQ topology.

2.2 Software Part

The iFDAQ software [7] is deployed on the readout engine, the individual computers of which run the Scientific Linux CERN 6 (SLC6) operating system [18]. The software is based on C++ and uses the Qt Framework not only for its GUI, but also for its threading. Furthermore, Qt data types and a variety of non-GUI classes are also used in the software. The Qt version used in the iFDAQ software is 5.5.1. Python and Bash script also find use in the iFDAQ, their scripts being particularly useful for starting processes remotely using SSH. Finally, XML is used to describe the hardware configuration of the iFDAQ in XML structure files and the IPBus [48] configuration in XML connection files and address files.

Six main functions are provided by the iFDAQ software: configuration of the hardware, monitoring of the data-taking process, remote control of the hardware, data flow control, logging of information and errors and log browsing. The iFDAQ software also includes a connection to an MySQL database. The database is used to

store, among others: configuration information of the iFDAQ's hardware, information logs and error logs.

There are six types of processes fulfilling these six functions in the iFDAQ [9]: Master, Slave-control, Slave-readout, Runcontrol GUI, MessageLogger, and MessageBrowser. The Master process is responsible for control of the system by retranslation of messages from user to slaves according to configuration loaded from database. It has access to all slaves through DIALOG services and direct access to MySQL database. It also has integrated error recovery functions to cope with problems caused by misbehaving slave processes. The Slave-control process supervises connected FPGA card by accessing registers via IPBus. The full scale system will contain 17 Slave-control processes which will be distributed over the readout computers. The Slave-readout process is the most complex and demands most of CPU resources in the iFDAQ. It is a multi-threaded process that monitors readout activities and checks consistency of accepted data. A Spillbuffer card is used as the data source. The data are transferred between threads via signal-slot connections mechanism of Qt by blocks of about 512 events. Events are distributed to 10 processing threads before final checks and preparation of the final data format. Portion of data is, simultaneously with storing on the HDD, distributed to monitoring outputs. The main graphical user interface is implemented in Qt framework. It has been designed and developed with emphasis on ergonomy and flexibility. It provides iFDAQ status information for expert and non-expert users. It runs in one of two modes: runcontrol and monitoring. There is only one runcontrol GUI allowed in the system; it controls and monitors state of system. The number of running monitoring GUIs is not limited, as they are used only for monitoring. MessageLogger and MessageBrowser are the last two processes to be discussed. The MessageLogger receives messages from all parts of the system and stores them in the database. The MessageBrowser is a visualization tool for browsing through these messages. The Master process and slave processes are based on state machines.

The original DAQ system of the COMPASS was based on the DATE software [4], originally developed for the ALICE experiment at CERN for control of the hardware, therefore, many user programs expect that data files are in the DATE data format. In the iFDAQ, transformation of readout data to DATE data format is needed because of this limitation.

Chapter 3

The Inter-Process Communication

The Inter-Process Communication (IPC) [59] deals with techniques and mechanisms provided by operating systems by which multiple threads in one or more processes exchange data with each other. These processes are not bound to one computer, and can run on various computers connected by network. Therefore, the IPC mechanisms are used to support distributed processing. Applications that split processing among computers using a common network are called distributed applications.

In general, distributed applications using the IPC can be categorized as clients or servers. A client (a requester of a resource) is an application or a process that requests a service from some other application or process. A server (a provider of a resource) is an application or a process that responds to a client request. Many applications act as both a client and a server, depending on the situation.

Firstly, basic characteristics of applications which would benefit from the IPC usage are mentioned. Moreover, to choose the most appropriate the ICP mechanism for a particular application, a developer has to ask key questions in order to meet requirements of an application. On the other hand, it is likely that an application will use several the IPC mechanisms, because some mechanisms partially overlap each other. Afterwards, the obtained specification determines whether an application can benefit by using one, more or combination of the IPC mechanisms.

- Should the application communicate with applications running on different computer?
- Should the application be able to communicate with applications running on other computers that may be running under different operating systems?
- Should the application communicate with any other application or with a cooperating and well-known application only?
- How many processes should participate in the communication?
- Is the communication rather less frequent with larger amount of data or more frequent with smaller amount of data to transmit?

- Is performance a critical aspect of the application? All the IPC mechanisms include some amount of overhead.
- How many computers is connected to network?
- What network bandwidth is currently used?

Processes can communicate with each other in many different ways. The IPC techniques can be divided into various types [59]:

- **Pipes** – The most basic versions of the UNIX operating system gave birth to pipes. It is a unidirectional data channel using the pipe system call, thus creating a pair of file descriptors. Data written to the write end of the pipe is buffered by the operating system until it is read from the read end of the pipe. Two-way data streams between processes can be achieved by creating two pipes utilizing standard input and output.
- **FIFO** – A pipe implemented through a file. A FIFO or "first in, first out" is a one-way flow of data. FIFOs are similar to pipes, the only difference being that FIFOs are identified in the file system with a name. Moreover, multiple processes can read and write to the file as a buffer. In simple terms, FIFOs are "named pipes".
- **Shared memory** – Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other. Therefore, processes communicate by reading and writing to that memory space.
- **Mapped memory** – This method can be used to share memory or files between different processors. A file mapped to RAM and can be modified by changing memory addresses directly instead of outputting to a stream. This mechanism speeds up a file access.
- **Message queues** – Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. A developer can pass messages between processes via a single queue or a number of message queues.
- **Sockets** – A data stream sent over a network interface, either to a different process on the same computer or to another computer on the network. It is byte-oriented, thus, data written through a socket requires formatting to preserve message boundaries. This method facilitates a standard connection that is independent of the type of computer and the type of operating system used.
- **.NET Remoting** – It is a .NET based framework for communication between applications. Objects in .NET are exposed to remote processes, thus allowing IPC. Unfortunately, it requires .NET framework being usually available on Windows platforms only.

Moreover, the interaction style in a communication domain [59] is considered to be a significant criterion too. When selecting an IPC mechanism for a communication, it is useful to think first about how processes interact with each other. There are a variety of interaction styles between processes. They can be categorized along two dimensions. The first dimension is whether the interaction is one-to-one or one-to-many:

- **One-to-one** – Each client request is processed by exactly one server instance.
- **One-to-many** – Each client request is processed by multiple server instances.

The second dimension is whether the interaction is synchronous or asynchronous:

- **Synchronous** – The client expects a timely response from the server and might even block while it waits.
- **Asynchronous** – The client does not block while waiting for a response, and the response, if any, is not necessarily sent immediately.

The following Table 3.1 shows the various interaction styles.

	One-to-one	**One-to-many**
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Table 3.1: The IPC interaction styles.

There are the following kinds of one-to-one interactions:

- **Request/response** – A client makes a request to a server and waits for a response. The client expects the response to arrive in a timely fashion. In a thread-based application, the thread that makes the request might even block while waiting.
- **Notification** – A client sends a request to a server, but no reply is expected or sent.
- **Request/async response** – A client sends a request to a server which replies asynchronously. The client does not block while waiting and is designed with the assumption that the response might not arrive for a while.

There are the following kinds of one-to-many interactions:

- **Publish/subscribe** – A client publishes a notification message which is consumed by zero or more interested servers.

- **Publish/async responses** – A client publishes a request message and then, it waits a certain amount of time for responses from interested servers.

In general, each process typically uses a combination of these interaction styles.

The iFDAQ processes must communicate with each other to satisfy the proper data taking. The communication library must meet all requirements in terms of reliability, efficiency and easy integration to a process. Firstly, the DIM library served as a communication interface among processes in the Run 2014 and 2015. From 2016, the newly developed DIALOG library provides with the communication layer and the previously used DIM library has been fully replaced.

3.1 The DIM Library

DELPHI [32] was one of the largest physics experiments in the world, it's online control system was composed of many different components distributed over many machines. In order to allow for efficient communication among machines and processes a communication system – DIM – was developed.

The processes involved in the DELPHI Online System needed to communicate efficiently and reliably across the different machines. The Online System was responsible for DAQ, trigger, control, monitoring, user interfacing, etc. The DIM (Distributed Information Management) [30] system was proposed and implemented in order to provide the required communication layer.

A generic design and implementation of the DIM offers a wide usage in other platforms and for other applications. For this reason, it provides an opportunity to use it also in other experiments at CERN, e.g., L3, L3 Cosmics and NA50 and by BaBar at SLAC [31].

3.1.1 Design Requirements

All different types of activities in a system define different demands on a communication system. From the DAQ point of view, transfer speed, reliability, handling of large amounts of data and access to all the information available in the experiment are the most important aspects [31]. In order to accomplish the above-mentioned demands the DIM was designed meeting the following requirements [31]:

- **Efficient Communication Mechanism** – A communication system should provide with an asynchronous behaviour in a message exchange among processes to offer a communication in a most efficient way. Once a message is available for sending, the sending procedure is processed. Similar approach should be implemented for the receiving procedure in a process.
- **Uniformity** – All processes should use the same communication mechanism in order to be able to exchange all information within a system. Then, the implementation and support of such a system is more manageable.

- **Transparency** – To satisfy the system independence, a distributed communication system must fulfil transparency. Any running process should be able to communicate with any other process in the system regardless of their current running location.

- **Reliability and Robustness** – In a system with many processes running on many machines connected by network links, it might happen that a process, a machine or a network itself breaks down. The application should be able to deal with the loss of one of these components, i.e., providing for system recovery in a self-recoverable manner from error situations or the migration of processes from one machine to another.

3.1.2 The Motivation for the DIALOG Library Implementation

The DIM library was fully incorporated to all processes for the Run 2014 and 2015. The iFDAQ had to face several problems connected to the DIM library during that time. Messages were sometimes delivered truncated with length multiple 4 B. The iFDAQ solved that by adding artificial spaces to the end of messages. The next problem is more serious, the messages were sometimes not delivered at all.

However, the decision to implement a new communication library came with the last issue. Processes crashed without any obvious reason. Especially, the Master process met this issue quite often. The debugging attempts sometimes terminated in the DIM library.

Unfortunately, the DIM library is a large package and to understand the source code is not a trivial task. The iFDAQ group made a decision to implement their own communication library.

Last but not least, the advantage of understanding the own library also played a key role in the decision making.

3.2 The Communication Library DIALOG

The name DIALOG represents conversation or interview in English, both connected to communication. Moreover, each letter is a first letter of a word characterizing somehow the library itself (distributed, inter-process, asynchronous, library, open, general). The DIALOG library [61] is designed in order to meet the following requirements:

- Any process should be able to access any information it needs in order to perform its processing or display activities.

- The integration to running system requires interface for an easy use.

- The information gathered should be consistent over all processes running at a given moment.
- Any process should be able to move from one machine to another.
- The communication system should be very robust. Any process dying should not disturb the rest system.

3.2.1 Description

In order to provide an easy recovery mechanism from crashes and migrate processes to another machine if necessary and satisfy the requirement for transparency, i.e., a client does not need to know where a server is running, the Control Server was introduced.

Information inside the DIALOG library is handled as named services. A service is a set of data with a name. The data inside a service can be of any type and size, since they are transferred as bytes. The Control Server keeps an up-to-date directory of all the processes and services available in the system.

To control particular process, commands are introduced. They are declared by specifying a name for the command and command with the same name can be registered by more processes. Once the command is delivered to process, generally, some action is taken.

The DIALOG library uses a client/server mechanism. A provider (server) is a process that has information to publish. It sends the list of services it provides to the Control Server at startup. A subscriber (client) is any process that uses a service. When requiring a service the subscriber asks the Control Server which provider provides that service and from then on, it contacts directly the provider. The Control Server knows at any time which services are available in the system and who provides them.

A recovery procedure is started whenever one of the processes (any process or even the Control Server itself) in the system crashes or dies. It includes the notification to remaining processes connected to it about the crashed process and reconnection as soon as a spare process will be available again. Moreover, this feature provides the possibility of balancing the machine load of the different workstations. By stopping a process in the first machine and starting it in the second one, a process can be easily migrated.

3.2.2 Integration

The DIALOG library is designed bearing in mind that it has to be integrated in a running system, so it has to be made as easy to use as possible. The library takes care of all the communications with the Control Server and with the other processes.

It gives all the DIALOG functionality to an existing process just by inserting one or two lines of code. This is possible, because the system is completely asynchronous.

Then, all messages are sent from the OutgoingThread object and delivered to the IncomingThread object. According to the message header, one can recognize the message type easily.

3.2.3 Robustness

The DIALOG library has become the most important means of communication between processes in the iFDAQ, so a very special care has been taken on the recovery from error situations.

The establishment of a communication channel between processes is independent of the order in which they are started. The Control Server keeps track of the subscribers for "non-available" services and contacts them as soon as the providers start up. More generally when any provider or subscriber dies its partners will reconnect as soon as it comes back up.

When the Control Server starts all the providers will re-register their services. And the subscribers will re-request the services they need.

In order to make sure that the processes are in a good state, each process sends a heartbeat to the Control Server, this way the Control Server can disconnect from a process or kill it if its behaviour is anomalous.

The communication between providers and subscribers once established is independent unless the Control Server dies. If the Control Server dies, the processes delete all information about other processes and try to reconnect to the new spare Control Server. Otherwise the behaviour of the whole communication system would become unpredictable without any heartbeat check. Once the Control Server is on again, services are registered and subscribed as in a fresh start.

3.2.4 Implementation

Providers

Providers are processes that have information to provide. A process becomes a provider by declaring any services it can provide and any commands it is willing to accept. It sends this information to the Control Server.

A service is declared by specifying a name for the service which is unique in the DIALOG system scope. A command is declared by specifying a name for the command and a command with the same name can be registered by more processes. Once the command is sent, it is forwarded by the Control Sever to all provider processes that registered it.

Subscribers

Subscribers are processes that need the available information in order to accomplish their tasks that being display, monitoring or processing. In order to become a subscriber a process has to specify the service name it is interested in and requesting for it.

From then on, the subscriber can go on with its work, any service message will automatically be processed whenever a service is received. At any time, a process can send a command to a provider by specifying the command name and the command message.

Any process can be a provider and a subscriber at the same time.

The Control Server

The Control Server keeps an up-to-date list of all the servers and services in the system, it receives registration messages from providers and service requests from subscribers. All processes send heartbeats at regular intervals so that the Control Server can be assured that they are functioning. If a process fails sending heartbeats the Control Server marks its services as not available, send the information concerning the crashed process to processes providing something to it and subscribing something from it. Once a spare process is started, it overtakes the same functionality as the crashed one.

The service uniqueness based on their names is a basic requirement for the system reliability. Any process trying to register a service being already registered is killed by a kill signal from the Control Server.

If the Control Server dies, the processes delete all information about other processes and try to reconnect to the new spare Control Server. Otherwise the behaviour of the whole communication system would become unpredictable without any heartbeat check. When it comes back up all providers re-register all their services (they have been trying at regular intervals) and all the subscribers re-request the services they are waiting for and all connections are then established.

3.2.5 Scenarios

In the following section, the most typical scenarios the DIALOG library is dealing with are presented. Each scenario is displayed in a particular data flow diagram followed by discussion.

In Figure 3.1, data flow diagram shows the connection mechanism to Control Server for each process. Once the connection procedure is successful, the Control Server notifies to the particular process. The messages from processes, which are not connected to the Control Sever, are ignored. It prevents the misleading or malicious behaviour of processes not belonging to the DIALOG at all.

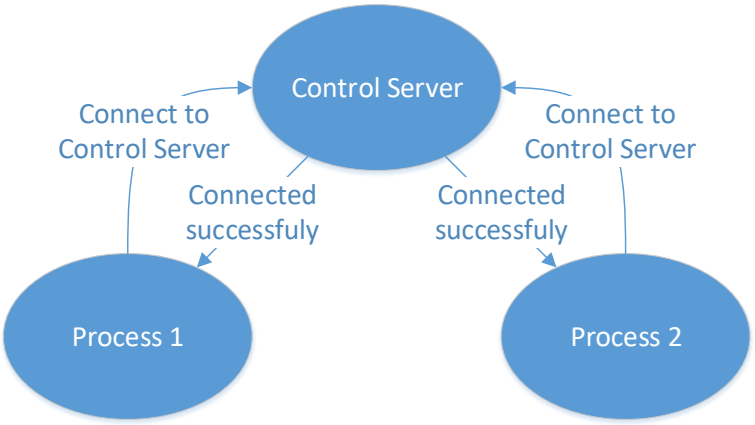

Figure 3.1: The DIALOG connection to the Control Server diagram.

If the process is not connected to the Control Server, nevertheless, it is still sending messages to the Control Server, the Control Server sends the message to the process, that it is not connected and probably it would like to connect.

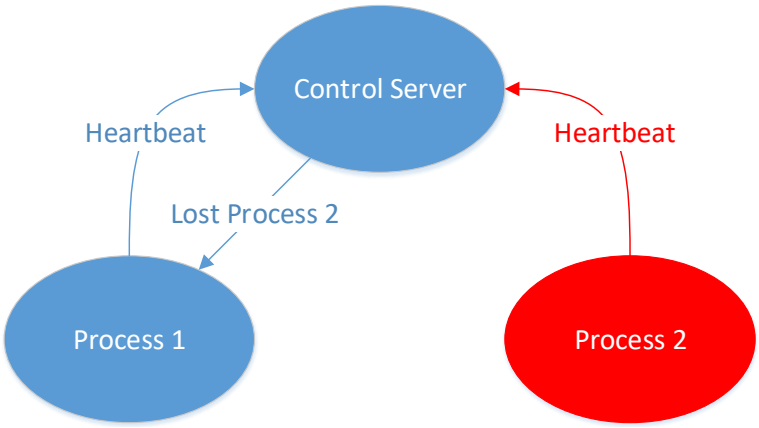

Figure 3.2: The DIALOG heartbeats diagram.

The heartbeat procedure is stated in Figure 3.2. All connected processes are sending heartbeats to the Control Server at regular intervals. The Control Server is checking whether the hearbeat is received in a given checking interval for each process. In Figure 3.2, the red color indicates a process which did not deliver its hearbeat in time. The process could fail or be stuck. Regardless of the real reason of the undelivered heartbeat, the Control Server considers the Process 2 as a lost one and notifies to all remaining processes.

The commands are significant part of the DIALOG library. In Figure 3.3, the data flow diagram for commands is shown. Process 2 is sending a command with the command name and the command message. The command is sent from Process 2 to the Control Server which knows all processes registering the command. The Control Server forwards the command to these processes and each process takes some action after the command delivery.

In Figure 3.4, data flow diagram shows the control and data flow among the basic components of the DIALOG, the Control Server receives service registration

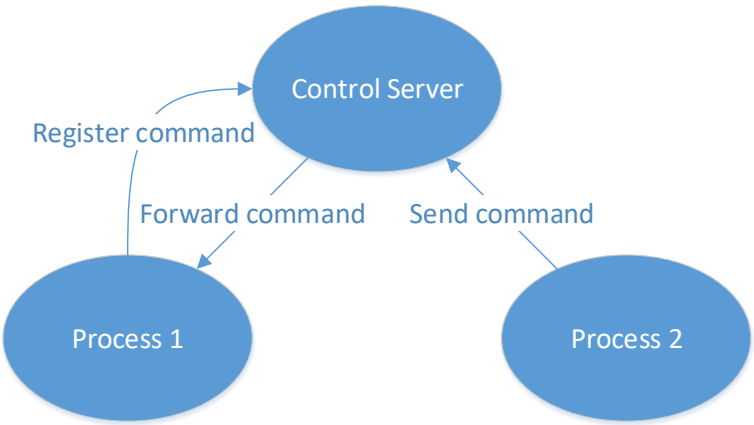

Figure 3.3: The DIALOG command diagram.

messages from providers and service requests from subscribers. Once a subscriber obtains the "Service Info", i.e. the service co-ordinates (hostname and port), from the Control Server it can then subscribe to services provided by provider process. If a subscriber sends a "Request Service" for a service that is not (yet) known to the Control Server a not-yet-provided "Service Info" is sent back to the subscriber, but the request stays queued in the Control Server and when the service is made available a new "Service Info" is then sent to the subscriber and the subscriber proceeds to connecting to the provider.

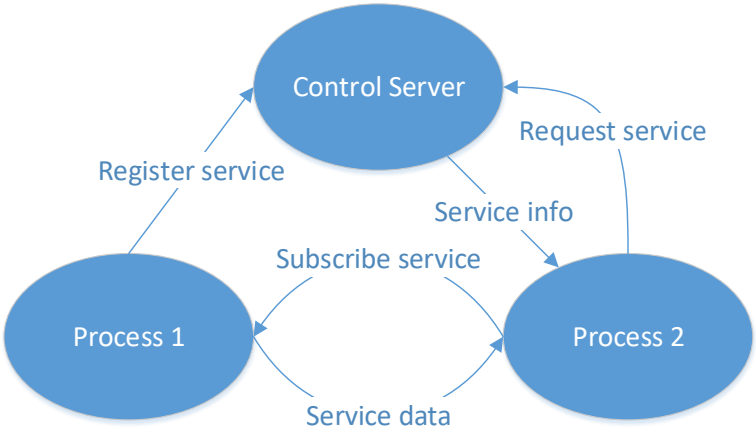

Figure 3.4: The DIALOG service diagram.

In last two diagrams, a deeper look inside a process and the DIALOG integration is given. In Figure 3.5, the threads and communication among them are described. The diagram can be divided into two parts. The first part is sending part, the second part is receiving one.

The Sender, running in the SenderThread, is taken care of dispatching messages among $n \in \mathbb{N}$ threads and load balancing. These n threads are establishing connections to other processes, writing data to sockets and keeping sockets open until timeout. The socket is not closed immediately. It remains open for next messages to the particular process. If no message is sent to the particular process for some time, the socket is closed by timeout. That means, there is always one or no open socket

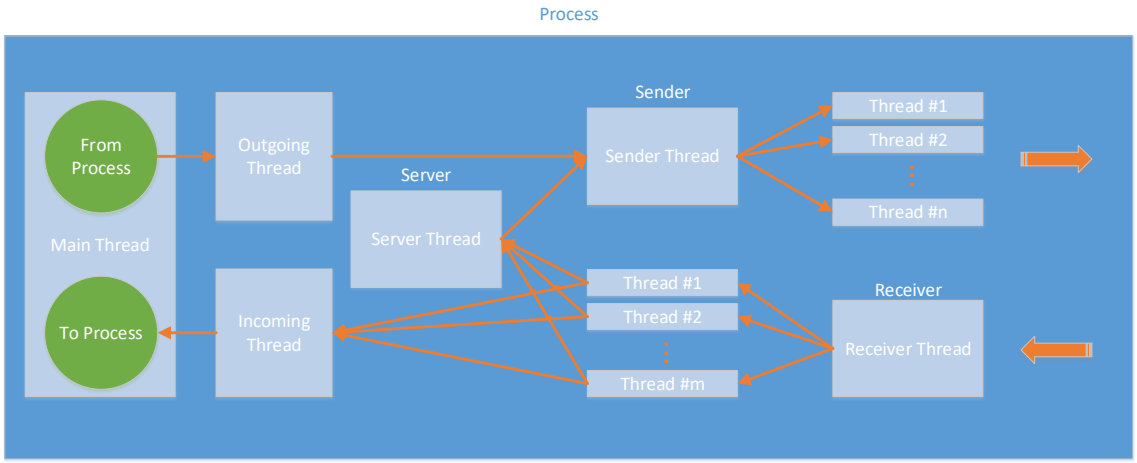

Figure 3.5: The DIALOG process threads diagram.

from one process to the other. If the socket is already closed and new message must be delivered to the particular process, the socket is re-established again.

The message consists of two parts – from message header and message data. It is handling by pointers to them. Once the message is created, it is leaving the process as soon as possible. All these aspects – open socket, pointers to messages, sending as soon as possible – speed up the performance and reduce the latency significantly.

According to the message header, the message types are distinguished and how to deal with them. Messages with header CONNECT_TO_CONTROL_SERVER, REGISTER_SERVICE, REQUEST_SERVICE, REGISTER_COMMAND, SERVICE_MESSAGE and COMMAND_MESSAGE are coming from the OutgoingThread to the SenderThread, to one of n threads and leaving the process. Messages with header HEARTBEAT and SUBSCRIBE_SERVICE are coming from the ServerThread to the SenderThread, to one of n threads and leaving the process.

There are also message headers being sent only from the Control Server, e.g. SUCCESSFULLY_CONNECTED, CONNECTION_LOST, INFO_SERVICE, LOST_SENDER or LOST_RECEIVER.

All these message headers are created only once and used until the process terminates.

The second part is the receiving one. Once the Receiver, running in the ReceiverThread, receives a new socket descriptor, the socket descriptor is dispatched to one of $m \in \mathbb{N}$ threads and the socket is created and opened. The Receiver is taking care of the new sockets only. The already established ones are keeping open until they are closed by sender process. These m threads are responsible for reading data out from sockets. Once a new message is read out, based on its message header, it is sent either to the ServerThread or to the IncomingThread.

The messages with message header CONNECT_TO_CONTROL_SERVER, HEARTBEAT, SUCCESSFULLY_CONNECTED, REGISTER_SERVICE, REQUEST_SERVICE, REGISTER_COMMAND, INFO_SERVICE, CONNECTION_LOST, LOST_SENDER or LOST_RECEIVER are sent to the Server-

Thread. Only SERVICE_MESSAGE and COMMAND_MESSAGE are heading to the IncomingThread directly.

The establishment of communication between two process has not been discussed in a precise way yet. In Figure 3.6, the establishment of communication between two processes is presented. Process 1 is on left and Process 2 is on the right. Process 1 is trying to send a message to Process 2.

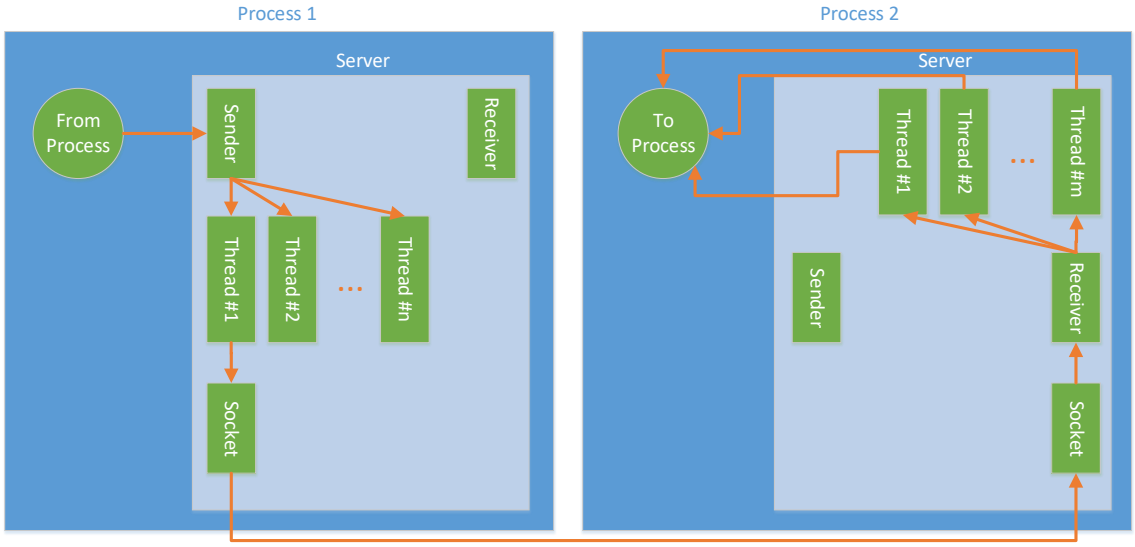

Figure 3.6: The DIALOG communication between processes diagram.

Process 1 sends message to the Sender dispatching it one of n threads. If the connection is not yet established, the object socket is created and opened in Process 1. In Process 2, if the connection between these two processes is not established yet, the Receiver receives the socket descriptor trying to connect to Process 2. The socket descriptor is dispatched to one of $m \in \mathbb{N}$ threads and object socket is created and opened. Then, based on the message header, the message is sent either to the ServerThread or to the IncomingThread.

Socket objects live on both sides until either Process 1 closes it because of timeout, or one of the processes crashes or one of the processes terminates in a correct way. Once the connection is established, Process 2 can write to socket as much as it needs and Process 2 receives and reads out all these messages. If the connection terminates, sockets are deleted on both sides.

The established socket is used only for one direction connection. To send a message in an opposite direction from Process 2 to Process 1, the new socket must be established. That means, there are either two open sockets, or only one open socket or no open socket at all between two processes at one point.

3.3 The DIALOG Online Monitoring API

The DIALOG library distinguishes three types of a process [65]. The process type determines the purpose of a process and how the DIALOG library should deal with

it.

- **ControlServer** – The Control Server keeps an up-to-date list of all processes, services and commands in the system. It receives registration messages from providers and request messages from subscribers. All processes send heartbeats at regular intervals so that the Control Server can be assured that they are working properly.
- **Custom** – It covers all processes they should communicate through the DIALOG library.
- **Monitoring** – Processes being responsible for the monitoring of the DIALOG library. Each of them represents a monitoring tool with a unique purpose. This general concept offers an easy to use API for a developer to implement any monitoring tool.

This section focuses on processes of type *Monitoring*. The *Monitoring* type identifies a process which does not contribute to the communication at all. It does not provide with any service. On the other hand, to be able to start listening to some/all communication, it can subscribe to a service or register a command it is interested in.

Firstly, a monitoring tool connects to the Control Server as well as anyone else. The Control Server recognizes the *Monitoring* type of a process and deals with it like with a monitoring tool during its whole life cycle.

The Control Server sends a monitoring info to a monitoring tool in XML format after successful connection of the monitoring tool to the Control Server. This monitoring info contains list of all connected processes of type *Custom* to the Control Server. The monitoring info in XML format also consists of all services being provided and subscribed by processes and all commands being registered in the Control Server. If something changes (e.g. a lost process, a new service, a new process, etc.), the Control Server will recognize it and re-send a new monitoring info immediately to all monitoring tools currently connected.

The monitoring info considers changes in the list of processes connected to the Control Server. Once a connected process terminates or crashes, the Control Server recognizes it and re-sends the monitoring info in XML format with the updated list of connected processes. If a new service is provided by any process or some service is subscribed by any process, the monitoring info is re-sent again to all monitoring tools. The last case is a registration of a new command. In this case, current monitoring info is again updated and re-sent. Based on the information from the monitoring info in XML format, a monitoring tool knows everything it needs to know. If a monitoring tool is interested in all communications it subscribes to all services and registers all commands and starts to listen to them. If a monitoring tool is interested in some portion of services, it subscribes to those services and starts to listen to them again.

The monitoring info in XML format begins with an XML declaration describing XML version and encoding. The uppermost tag `<processes>` encapsulates the list

of all processes. Each `<process>` tag is a child element of `<processes>` tag and has attribute *name*, *address*, *port* and *pid*. The tag `<process>` either has no child elements or has the list of services it provides and registered commands it is willing to accept. The tag `<command>` is considered to be an empty-element tag and has only attribute *name*, such as `<command name="commandName" />`. The tag `<service>` has also only one attribute *name* and can contain child elements representing the list or receivers, i.e., the list of processes being subscribed to the service. The tag `<receiver>` consists of attributes *name*, *address*, *port* and *pid*. The sample of monitoring info XML format follows in Listing 3.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <processes>
3      <process name="SR_RE11" address="pccore11.cern.ch" port="57143"
       pid="22640">
4          <service name="INFO_SERVICE_56">
5              <receiver name="MSGBrowser" address="pccorc21.cern.ch"
                  port="33511" pid="24958" />
6              <receiver name="Master" address="pccore15.cern.ch" port
                  ="51523" pid="17376" />
7              <receiver name="MSGLogger" address="pccore15.cern.ch"
                  port="56614" pid="17377" />
8          </service>
9          <command name="RUN_CONTROL_56" />
10     </process>
11 </processes>
```

Listing 3.1: The monitoring info XML sample.

3.3.1 The DIALOG GUI

The DIALOG GUI is one example among many of the monitoring tools using the online monitoring API for the DIALOG library. A behaviour of complex distributed applications can be very difficult to understand without a help of a dedicated tool for online monitoring. The DIALOG GUI allows a visualization of all processes involved in the distributed communication system as shown in Figure 3.7.

A development of a distributed system is quite challenging from a synchronization and robustness point of view. The DIALOG GUI helps with a debugging of misleading functioning so that the system start to work properly. It provides all information from the monitoring info in a well-arranged way. The main widget consists of two parts. The main part contains all necessary information about connected processes to the Control Server with the process type *Custom*. There is the name of each process, the machine where it runs, the port on which it listens to and its process ID. Moreover, each row corresponding to a single process offers the list of provided services, subscribed services and registered commands by the process. If the DIALOG system contains many processes, a user appreciates a filter in the DIALOG GUI for an easy searching.

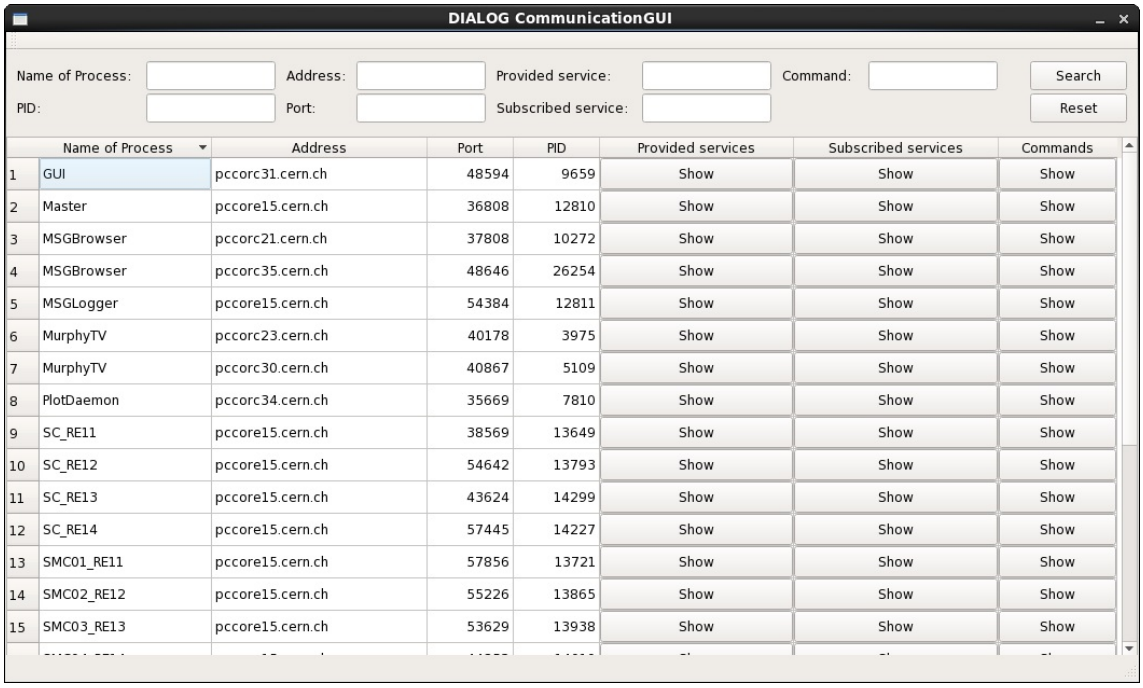

Figure 3.7: The DIALOG GUI.

The provided services widget shows the list of provided services by the selected process. A user can select a service from the list and see the list of all subscribers of the service. Moreover, a user can start to listen to the selected service and see what the service is providing, see Figure 3.8.

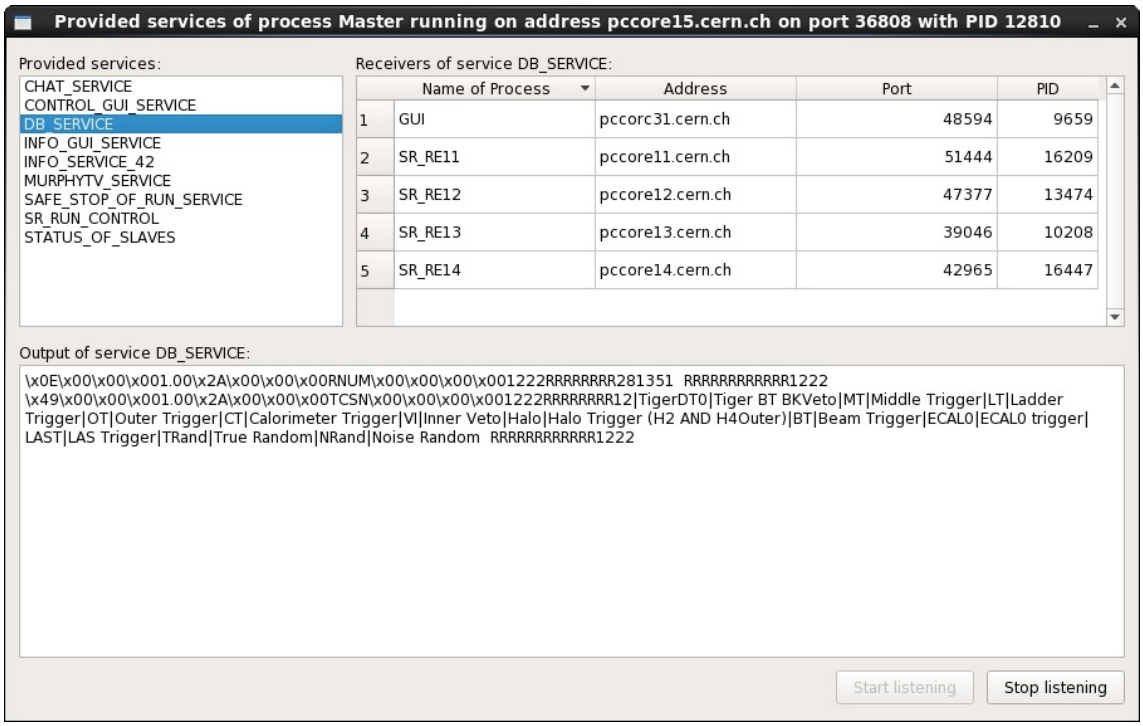

Figure 3.8: The provided services widget of the DIALOG GUI.

The subscribed services widget shows all services being subscribed by the selected process. Basically, it is a list of all services the process is interested in. In the subscribed services widget, a user can select a service from the list of services and see which process is providing the service. Moreover, a user can start to listen to the selected service again. In both provided/subscribed services widget, a user can start to listen to all services at once and see the entire process communication.

The commands widget is corresponding to the selected process and its registered commands it is willing to accept, see Figure 3.9. In the widget, the list of registered commands is stated.

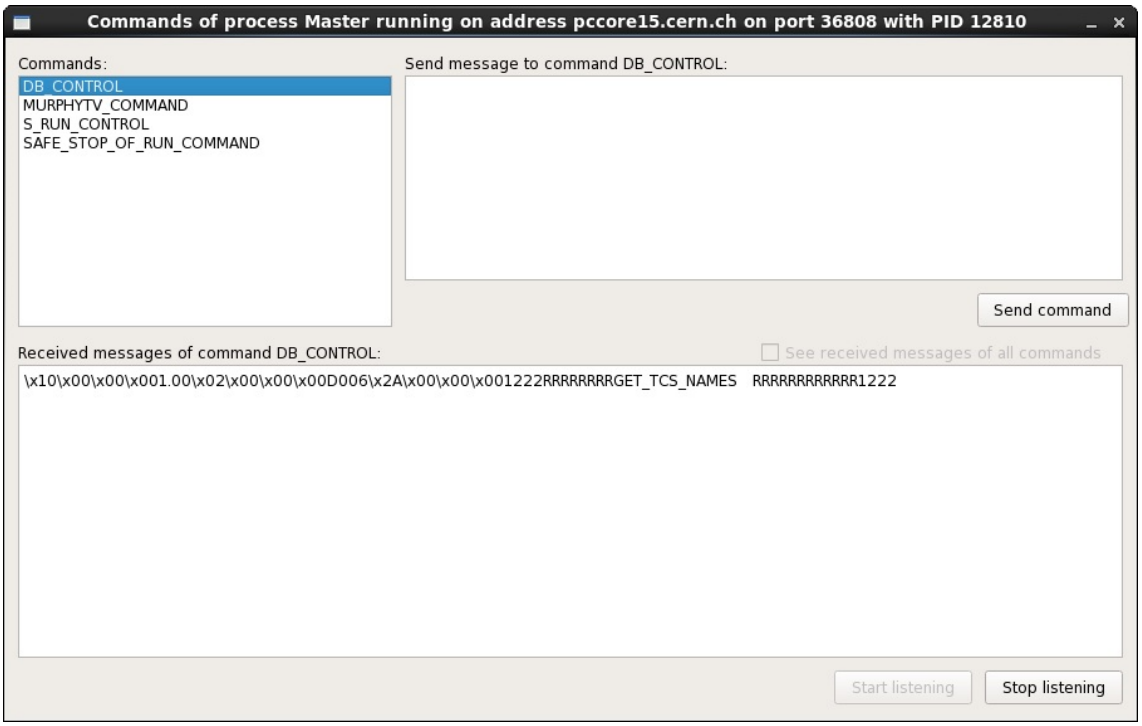

Figure 3.9: The registered commands widget of the DIALOG GUI.

A user can start to listen to the selected one or to all of them. Moreover, there is also possibility to send a command message directly from the DIALOG GUI using the selected command.

3.3.2 The DIALOG POST Daemon

Web-based applications offer a range of business advantages over traditional desktop applications. Web-based applications are:

- **Easier and more cost effective to develop** – Users access the system via a uniform environment (a web browser).
- **More useful for users** – Unlike traditional applications, web systems are accessible anytime, anywhere and via any device with an internet connection.

- **Easier to install, maintain and keep secure** – Once a new version or upgrade is installed on the host server, all users can access it straight away and there is no need to upgrade a device of each potential user.

Considering all above-mentioned aspects, the DIALOG library should provide a layer for communication between a desktop application and a web application. The solution should be general as much as possible, although this section deals only with monitoring tools. On the other hand, such a general solution solves a problem concerning a connection establishment between any web application and any desktop application. It offers a general communication mechanism regardless of a platform and environment where it is running.

There is a wide range of possible solutions in terms of a system architecture. The thesis considers and presents two possible solutions how to communicate between a desktop application and a web application – HTTP GET/POST methods and WebSockets, see in Section 3.3.3.

The DIALOG POST Daemon serves as a middleman between a desktop application side and a web application side. The DIALOG POST Daemon's purpose is to catch all communication being sent among all processes with the process type *Custom* and transmit all data in JSON format using HTTP and its POST method. The POST request is received by a web application for a communication measurement. It is a measurement tool of overall communication and shows statistics and plots related to communication among processes. It can help with understanding of possible bottlenecks and better load balancing of processes among machines.

The main idea is that the DIALOG POST Daemon with the *Monitoring* process type subscribes to all services being provided and registers all commands being already registered on the Control Server. Once a new monitoring info comes, the DIALOG POST Daemon subscribes to new services and registers new commands. Then, all communication is sent to the DIALOG POST Daemon. Basically, the DIALOG POST Daemon is capable to determine the communication between any processes.

A publish frequency for the POST method is set to 1 second and can be easily changed. During 1 second period, all received messages are gathered in JSON format and the collected JSON is published using POST method and freed every 1 second. Each message in JSON consists of information about a sender, i.e., a sender address, a sender port and its process name. To reduce JSON message size, messages are grouped by message itself. That means, if a message has more subscribers, the list of subscribers is only added. Therefore, the list of receivers is added (each entry with receiver address, receiver port and receiver name). There is also the DateTime in simplified extended ISO format (ISO 8601) [41] which is always 24 or 27 characters long (YYYY-MM-DDTHH:mm:ss.sssZ or ±YYYYYY-MM-DDTHH:mm:ss.sssZ, respectively). The timezone is always zero UTC offset, as denoted by the suffix "Z". The proper time format allows to localize the the DateTime in a web application all over the world with respect to users's current time zone.

The rest of record is filled with the MessageHeader and the MessageBody. The MessageHeader distinguishes a message type. Either it can be a service message or a command message. The MessageHeader contains also information about the name

of service or command. The MessageBody is filled with the transmitted information (message) itself. It is encoded using Base64 [39], since a message could be general indeed. It could contain also non-printing characters that would be lost during transmition. Base64 encoding allows to transmit not only messages but also whole files. The sample of JSON format being regularly sent using HTTP POST method follows in Listing 3.2.

```
[
    {
        "Sender": {
            "SenderAddress": "pccore15.cern.ch",
            "SenderPort": "12345",
            "SenderName": "Master"
        },
        "Receivers": {
            "Receiver0": {
                "ReceiverAddress": "pccore13.cern.ch",
                "ReceiverPort": "54698",
                "ReceiverName": "SR_RE13"
            },
            "Receiver1": {
                "ReceiverAddress": "pccore12.cern.ch",
                "ReceiverPort": "54879",
                "ReceiverName": "SR_RE12"
            }
        },
        "DateTime": "2017-09-13T07:20:25Z",
        "MessageHeader": "SERVICE_MESSAGE|STATUS_OF_SLAVES",
        "MessageBody": "textEncodedInBase64" // base64 encoding
    }
]
```

Listing 3.2: The DIALOG POST Daemon JSON sample.

Currently, the DIALOG POST Daemon is prepared and a web application is still under development. The development of web application should be finished in 2018 and deployment is planned in the late 2018.

3.3.3 The DIALOG WebSockets Daemon

The HTTP GET/POST method is suitable for a client-server architecture, when a client (the DIALOG POST Daemon) is publishing something to a server (a web application). On the other hand, it would be useful to have an effective channel to communicate in both directions. That could be obtain using WebSockets among others. WebSockets [40] is an advanced technology that makes it possible to open an interactive communication session between the user's browser and a server. With this API, an application can send messages to a server and receive event-driven responses without having to poll the server for a reply. WebSocket provides full-duplex communication channels over a single TCP connection. Currently, all up-to-date versions of internet browsers (IE, Edge, Firefox, Chrome, Safari, Opera) fully

supports WebSockets.

The idea is to develop the DIALOG WebSockets Daemon as a middleman between a desktop communication system and a web application. The DIALOG WebSockets Daemon should encapsulate a layer with WebSockets and transmit messages from the DIALOG communication interface to a web application. Runcontrol GUI for monitoring and controlling of the iFDAQ as a web application is planned for year 2019. Currently used Runcontrol GUI as a desktop application suffers from a high Qt version-dependency, a precise operating system and an environment setup. A development of the DIALOG WebSockets Daemon providing WebSockets interface is planned for year 2018.

3.4 Tests

Several tests have been conducted to validate system components. The performance of the DIALOG and DIM library has been measured. The system consists of 8 Slave-control processes for 8 MUXes (8 FPGA), 1 Slave-control process for SWITCH (1 FPGA), 8 Slave-control processes for Spillbuffer, and 8 Slave-readout processes (total 25 slaves processes). All slaves are sending messages concerning their status. There is also one Master process incorporated in this test. The Master is receiving all messages from all slaves concerning their status. This setup simulates the behaviour of the iFDAQ full setup.

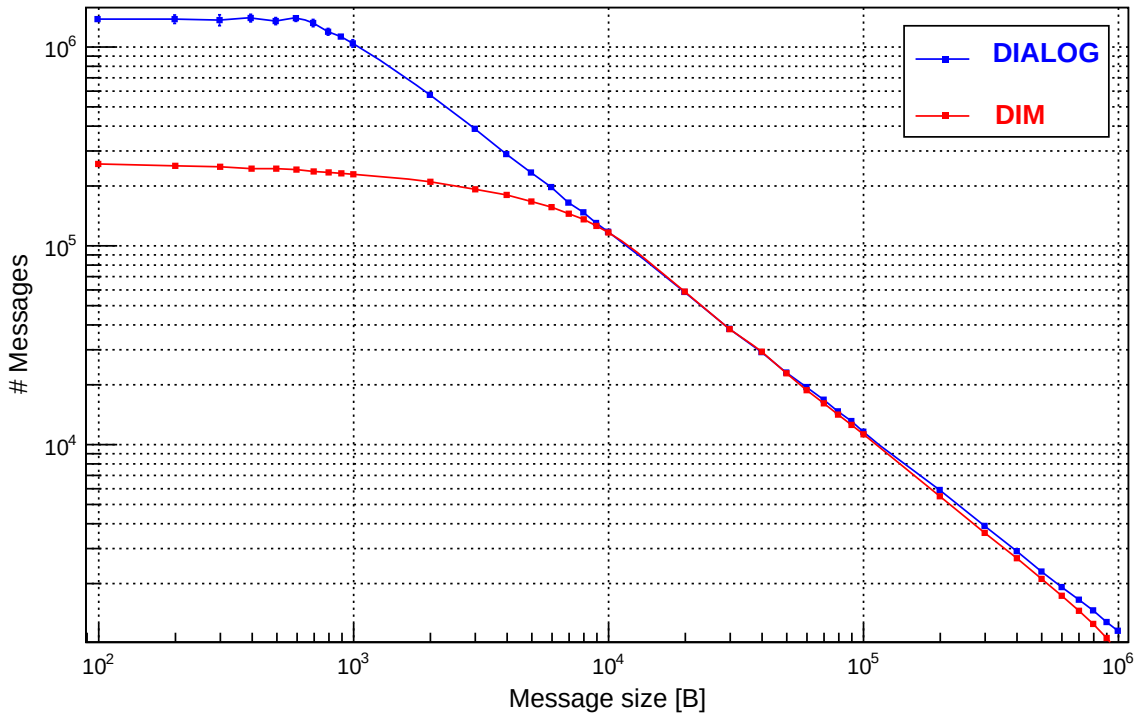

Figure 3.10: Number of messages.

The test measures how many messages can be delivered to one single process in 1 second. The test is conducted for different message sizes and for each message size

is conducted five times to obtain the sufficient statistics. Firstly, the test is using the DIALOG library. Afterwards, the test is performed also with the DIM library.

Before the start of the test, a special attention must be paid to spreading of slaves among machines. Since on machines operating with Linux, if the Master process and slave are running on the same machine, the message is sent directly from slave to the Master process and it is not running through the network at all. If that fact had not been considered, the test results would have been even above the network bandwidth. In the test, the Master process is running on its own machine and nothing else is running there.

For the test, the network bandwidth is 10 Gbps. Based on the bandwidth, the maximum data rate ~ 1.2 GB/s (throughput) can be expected. Moreover, the network bandwidth is not saturated by anything else and is exclusively at test's disposal.

In Figure 3.10, the number of received messages for the particular message size is given. As can be seen, the DIALOG library is significantly more efficient for messages with the message size up to 10 kB. In particular with small messages, it is capable to receive approximately five times more messages than DIM library. Unfortunately based on the plot, no conclusion concerning the performance for the messages with message size over 10 kB can be drawn.

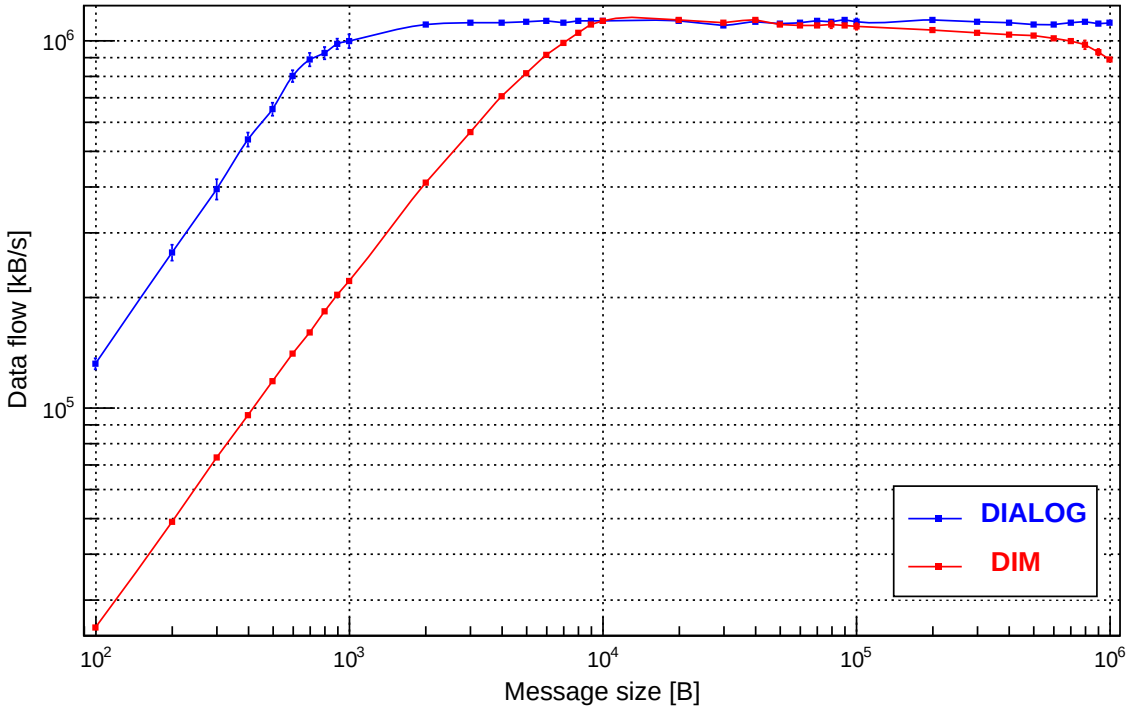

Figure 3.11: Data flow.

To overcome the drawback of the plot in Figure 3.10, Figure 3.11 is given. In Figure 3.11, the data flow for the particular message size is given. As can be seen from the plot, in particular, the DIALOG library is more efficient at the beginning with small message sizes. Taking a deeper look, the biggest difference in data flow is for messages with message size from 100 B to 1 kB. At these points, the DIALOG data flow is approximately five times bigger than the DIM data flow. Both DIALOG and

DIM curves saturate the network bandwidth eventually. Moreover, the DIALOG saturation of the network bandwidth occurs even with smaller message size than the DIM saturation. The DIALOG saturate reaches the maximum network bandwidth already with message size 2 kB. In comparison to DIM, it is much sooner. The DIM library is starting to occupy the whole bandwidth with message size 10 kB. At this moment, it is worth of mentioning the most frequent message size is between 1 kB and 2 kB in the iFDAQ.

Both libraries stand at the maximum network bandwidth with message size from 10 kB to 100 kB. Then, the data flow of both libraries changes again. While the DIALOG data flow stands at the level of the maximum network bandwidth and saturates it regardless the message size until the end, the DIM efficiency starts to decrease. At the level of 100 kB message size, the DIM data flow starts to decline until the end markedly.

From a global perspective, based on both plots in figures, a final conclusion can be drawn. The performance of the DIALOG library is significantly better than the DIM library.

Chapter 4

The iFDAQ Debugging

Nowadays, modern experiments in high energy physics demand a powerful, efficient and reliable DAQ. This chapter presents the development and deployment of a debugging tool called DAQ Debugger for the iFDAQ. In complex softwares, such as the iFDAQ, having thousands of lines of code, the debugging process is absolutely essential to reveal all software issues. Unfortunately, conventional debugging of the iFDAQ is not possible during the real data taking.

The DAQ Debugger is a tool for identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a way to work around it. It provides the layer for an easy integration to any process and has no impact on the process performance. Based on handling of system signals, the DAQ Debugger represents an alternative to conventional debuggers provided by most integrated development environments. Whenever problem occurs, it generates reports containing all necessary information important for a deeper investigation and analysis.

In general, it is considered as an useful tool for bugs identification and backward debugging. The DAQ Debugger is fully incorporated to all processes in the iFDAQ since 2016, including the Run 2016. It helped to reveal remaining software issues and improved significantly the stability of the system in comparison with the previous run. In the chapter, the DAQ Debugger is presented from several insights and discussed in a detailed way.

4.1 Conventional Debugging

In computer programming and engineering, debugging [15, 34, 52] is a multistep process that involves attempt to reproduce the problem and isolating the source of the problem. Then, the second phase of fixing the problem follows. It can be either fully corrected or determined a way to work around it. The final step of debugging is a verification that the fix works and nothing else is broken.

Once an error has been identified, it is essential to detect the error in the source code. Integrated Development Environment (IDE) is usually very useful in error detection.

The state-of-the-art IDEs provide developers with a stand-alone debugger tool or the debugging component helping developers to find the error in the source code.

The standard debugging tool provides the programmer with the capability to examine program states (values of variables, call stack, etc.) and track down the origin of the problem. The control of program execution is assured by setting up a "breakpoint" and run the program until that breakpoint. Once the program meets any breakpoint, the program execution stops and waits. The control of program execution also offers to execute just the next line of code, step into the body of function/method or even change the value of variables.

In software development, debugging is part of the software testing process and is an essential part of the entire software development life cycle. The debugging process starts as soon as a release candidate is implemented and continues step by step to form a final version of software.

4.2 The Motivation for the DAQ Debugger Implementation

The iFDAQ faced several crashes of the Master process and the Slave-readout process per day in the Run 2014 and 2015. Processes crashed without any obvious reason or additional information.

The possibility of conventional debugging during the real data taking is quite limited:

- It would waste the beam time during crash investigation.
- The performance of debugged processes would be lower.
- The conventional debugging process would increase load on readout engine computers.
- The iFDAQ expert would have to be present 24/7 on site.

In sum, the conventional debugging is possible only during so called machine development, i.e., time without beam. Unfortunately, the errors do not occur without the real data taking and all processes are running smoothly. Under the above mentioned circumstances, it gets caught in a vicious circle. Conventional debugging is not usable and effective for the error detection.

4.3 The state-of-the-art error reporting tools

The number of tools generating C++ application crash reports is quite limited. The report must contain the whole stack trace of all threads and produce a memory dump. Such requirements decrease the number of open source tools to a single one, i.e., Google Breakpad [33].

Google Breakpad is a library and tool suite that allows a developer to distribute an application to users with compiler-provided debugging information removed, record crashes in compact "minidump" files, send them back to a developer's server, and produce C and C++ stack traces from these minidumps.

In other words, Google Breakpad is responsible for monitoring an application for crashes (exceptions), handling them when they occur by generating a dump, and providing a means to upload dumps to a crash reporting server. These tasks are divided between the "client" library linked in to an application being monitored for crashes, the "symbol dumper" library reading the debugging information produced by the compiler and producing a symbol file, and the "processor" library reading a minidump file, finding the appropriate symbol files for the versions of the executables and shared libraries the minidump mentions, and producing a human-readable C++ stack trace.

The reports are created based on system signals. However, the integration to an application is not easy-to-use and straightforward. It requires manual integration to a process, manual production of application symbols and manual process of minidump file to produce a final stack trace. All these steps should be done automatically.

Moreover, Google Breakpad requires the most recent version of C++ compiler and Python interpreter. Unfortunately, such a requirement is not possible to meet in the COMPASS experiment.

Therefore, the DAQ Debugger has been implemented. It helped to detect remaining software issues and improved significantly the stability of the system.

4.4 DAQ Debugger

The DAQ Debugger [62] is a library helping with the iFDAQ error detection. The DAQ Debugger was fully incorporated to all processes in the iFDAQ during the Run 2016 and 2017. In general, the integration is very simple to any process. The main goal is to produce a report concerning the process crash. The report must contain as much information as possible. Afterwards, the reports are investigated by iFDAQ experts trying to detect the source of problem. After understanding of a problem, the fix is released and tested. The DAQ Debugger is designed in order to meet the following requirements:

- The integration to running system requires interface for an easy use.
- It does not affect the process performance.
- It does not increase load on readout engine computers.
- It provides with reports in `/tmp` folder containing stack trace of all threads and memory dump.

4.4.1 Description

The DAQ Debugger is a library easily integrated to a process and standing in the background of a running process. If the process is running without any crashes, basically, the DAQ Debugger is only part of the process without any action taken and behaves during the whole process life cycle in this way.

At the operating system level, the fault is caught and a signal is passed on to the offending process, activating the process's handler for that signal. Different operating systems have different signal names to indicate that a fault has occurred. For instance, in case of a segmentation violation, a signal called SIGSEGV (abbreviated from segmentation violation) is sent to the offending process on Unix-based operating systems.

The main idea of action taken in the right instant is based on catching of system signals (SIGSEGV, SIGABRT, etc.). In case of a process crash, the following procedure is started:

- The system signal is caught and forwarded to a signal handler in the DAQ Debugger.
- The memory dump is produced and stored.
- The whole stack trace for each thread is generated with file names and code line numbers.
- The report containing the caught signal and stack trace for each thread is created in `/tmp` folder.
- The process is exiting with the caught signal.

4.4.2 Integration

The DAQ Debugger is designed bearing in mind that it has to be integrated in a running system, so it has to be made as easy to use as possible. To incorporate it to any process, the static initialization method is called in a single line, as you can see in the following example in Listing 4.1 showing the integration of the DAQ Debugger into a Qt-CoreApplication.

```
1 #include <QCoreApplication>
2 #include "daqdebugger.h"
3
4 int main(int argc, char **argv)
5 {
6     QCoreApplication* app = new QApplication(argc, argv);
7     DAQDebugger::init(argv[0]);
8     return app->exec();
9 }
```

Listing 4.1: The integration of the DAQ Debugger into a Qt-CoreApplication.

The first argument of `DAQDebugger::init(argv[0])` is the name of a process. Then, the name of a process is included in the report file name.

Since the DAQ Debugger is a library, it must be located on the system path. Generally, `LD_LIBRARY_PATH` is used to specify directories of libraries. It is also necessary to add the following lines in Listing 4.2 to the Qt-project file (*.pro) in order to create the makefile correctly.

```
# DAQDebugger Start
INCLUDEPATH += PATH_TO_DAQ_DEBUGGER/
DEPENDPATH += PATH_TO_DAQ_DEBUGGER/
LIBS += -L PATH_TO_DAQ_DEBUGGER -lDAQDebugger

QMAKE_CXXFLAGS += -rdynamic -g
QMAKE_LFLAGS += -rdynamic -g

QMAKE_CXXFLAGS +=
-include PATH_TO_DAQ_DEBUGGER/qthreaddaqdebugger.h
QMAKE_CXXFLAGS +=
-include PATH_TO_DAQ_DEBUGGER/qthreaddaqdebugger_macro.h
# DAQDebugger End
```

Listing 4.2: The Qt-project file (*.pro).

GCC flags `-rdynamic` and `-g` enable use of extra debugging information. The `-rdynamic` option instructs the linker to add symbols to the symbol tables that are not normally needed at run time. The `-g` option produce debugging information in the operating system's native format.

The `-include` option processes file as if `#include "file"` appeared as the first line of the primary source file. If multiple `-include` options are given, the files are included in the order they appear on the command line. Demand on `-include` statements is discussed in the implementation subsection in a deeper way.

The process should be compiled without any optimizations, e.g, `-O`, `-O1`, `-O2`, `-O3` or `-Os`, if possible. The default value usually is `-O0` that means do not optimize. It is important for `addr2line` command that is used for detection of an exact file name and a line number crash.

Using optimizations, the compiled source code could be inline and detection of an exact file name and a line number crash is then much harder. Hence, using `addr2line` command on processes compiled with optimizations could lead to shifted or mistaken line numbers.

On the other hand, optimizations are very useful for compilation of libraries due to libraries' shared and linking purpose.

To sum up, turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

4.4.3 Implementation

The system signals to catch are specified in the `DAQDebugger::init(argv[0])` static method. By default, signals SIGABRT, SIGSEGV, SIGILL and SIGFPE are registered. The signals to catch can be added or removed there.

Whenever the registered signal is caught, it is caught in the thread causing the crash and the stack trace of thread can be easily produced. It could be enough in single-threaded processes. Unfortunately, the solution must be more general and considered even multi-threaded processes. The solution must provide the following crash procedure:

- The system signal is caught in the crashed thread.
- All remaining threads are immediately suspended.
- Store memory dump.
- Get stack trace of the crashed thread.
- Get stack traces of suspended threads.
- The crashed thread (whole process) is exiting with the caught signal.

To register a system signal, the following statement in Listing 4.3 is executed.

```
1 // to register a system signal
2 signal(signal, signalHandler);
```

Listing 4.3: The registration of a system signal.

POSIX [38] defines a standard threading library API in order to control and send suspend/resume signals to threads. All important statements are given in the following source code in Listing 4.4 with the explanations in comments.

```
1  // to send a signal to thread ID
2  pthread_kill((pthread_t)threadID, signal)
3
4  // to catch the sent signal in a thread
5  struct sigaction sigActionThreadControlSignal;
6  sigfillset(&sigActionThreadControlSignal.sa_mask);
7  sigdelset(&sigActionThreadControlSignal.sa_mask, signal);
8
9  sigActionThreadControlSignal.sa_flags = 0;
10 sigActionThreadControlSignal.sa_handler = signalHandler;
11 sigaction(signal, &sigActionThreadControlSignal, NULL);
12
13 // to suspend a thread
14 sigset_t sigActionThreadControlSignalMask;
15 sigfillset(&sigActionThreadControlSignalMask);
```

```
16 sigdelset(&sigActionThreadControlSignalMask, signal2);
17
18 sigsuspend(&sigActionThreadControlSignalMask);
```

Listing 4.4: The thread control based on POSIX.

The `signalHandler` is registered in `sigaction` and it is triggered if the `signal` is sent to the thread. Moreover, the thread is suspended by `sigsuspend` with given signal mask and it resumes if the `signal2` is sent to the thread.

At this point, the control of threads is prepared, the DAQ Debugger can obtain the stack trace with file names and line numbers for each thread. Using `backtrace` and `backtrace_symbols`, the stack trace is generated. The `backtrace` command returns the series of currently active function calls for the process. Moreover, using `backtrace_symbols`, the symbolic description of function calls is translated from information obtained by `backtrace` to function names and hexadecimal addresses. Unfortunately, it returns each line of stack trace in a hexadecimal address format and thus, it is not easily readable for a human being. However, to overcome this drawback, the DAQ Debugger is using `addr2line`. It is capable to convert hexadecimal addresses into file names and line numbers. In the following code in Listing 4.5, you can see a short example.

```
1  // storage array for stack trace address data
2  unsigned int max_frames = 63;
3  void* addrlist[max_frames + 1];
4
5  // retrieve current stack addresses
6  unsigned int addrlen = backtrace(addrlist, sizeof
7  (addrlist)/sizeof(void*));
8
9  // resolve addresses into strings containing
10 // "filename(function + address)"
11 char** symbollist = backtrace_symbols(addrlist, addrlen);
12
13 for (unsigned int i = 1; i < addrlen; i++)
14     std::cout << readSystemCommand
15     ("addr2line -e " + processName + " " + getHexAddress
16     (symbollist[i])) << std::endl;
```

Listing 4.5: The conversion of stack trace using file names and line numbers.

The stack trace is one thing, on the other hand, it is still not sufficient for the error detection. To satisfy the comprehensive understanding of crash, the memory dump is absolutely essential. The DAQ Debugger is using `gcore` command for memory dump storage.

Another challenge in the design of the DAQ Debugger is the registration of all threads without violating the concept of easy integration. In order to be able to send signals and to control threads in case of a crash, it is necessary to obtain all thread IDs at the beginning of a process.

Unfortunately, it is not an easy task to obtain IDs of all threads in a process. Moreover, it is even more complex if the integration of DAQ Debugger should be as simple as possible. The only way how to get thread ID is executing the part of code in this thread asking for thread ID. So, it must be ensured the execution of `DAQDebugger::addThreadSlot(thread)` static method in each thread. Each thread must register itself in the DAQ Debugger immediately when it starts its execution. It uses `started()` signal in `QThread` object being emitted when the thread starts executing. The functionality of `QThread` object must be extended in order to cover the registration in the DAQ Debugger and its integration would be still simple to any process. This extension is hidden and added by `-include` statements to Qt-project file (*.pro). File `qthreaddaqdebugger.h` contains the definition of thread satisfying the required functionality and inheriting from `QThread` object. Moreover, file `qthreaddaqdebugger_macro.h` replace all `QThread` objects for `QThreadDAQDebugger` objects by preprocessor definition in the whole process as follows in Listing 4.6.

```
1 #define QThread QThreadDAQDebugger
```

Listing 4.6: The preprocessor definition for replacement of all `QThread` objects for `QThreadDAQDebugger` objects.

In Figure 4.1, you can see the DAQ Debugger class diagram being obtained by the described integration process.

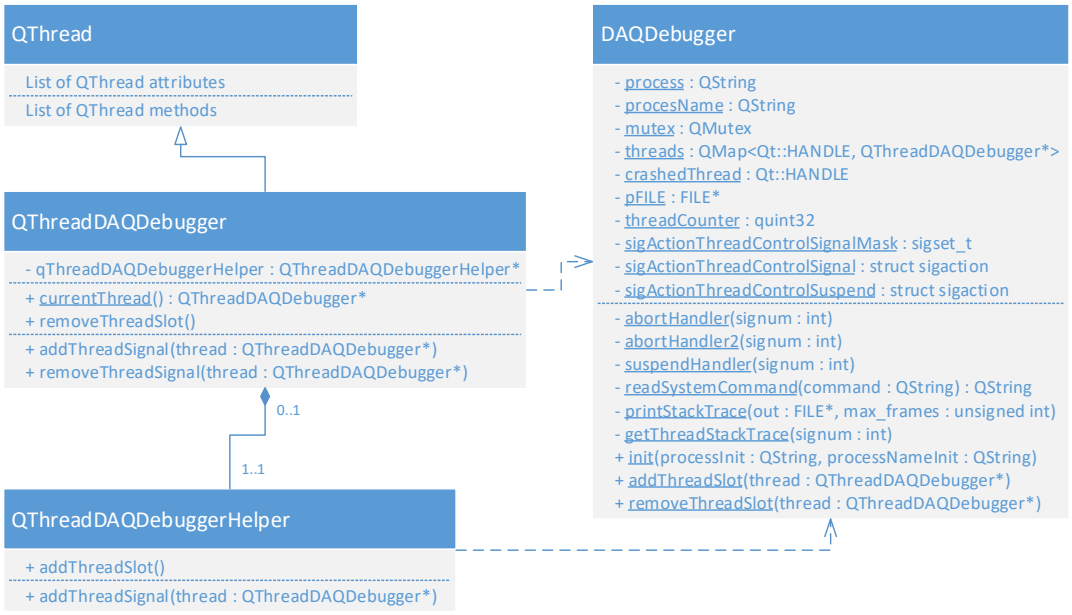

Figure 4.1: Class diagram of the DAQ Debugger.

The way how to introduce the DAQ Debugger to each thread has been described. It remains to discuss all most common scenarios, including the thread registration procedure, in the DAQ Debugger in a deeper way. It is given in the next subsection.

4.4.4 Scenarios

Basic scenarios are emphasized in this subsection dealing with how one or more components interact inside the DAQ Debugger or with the DAQ Debugger itself.

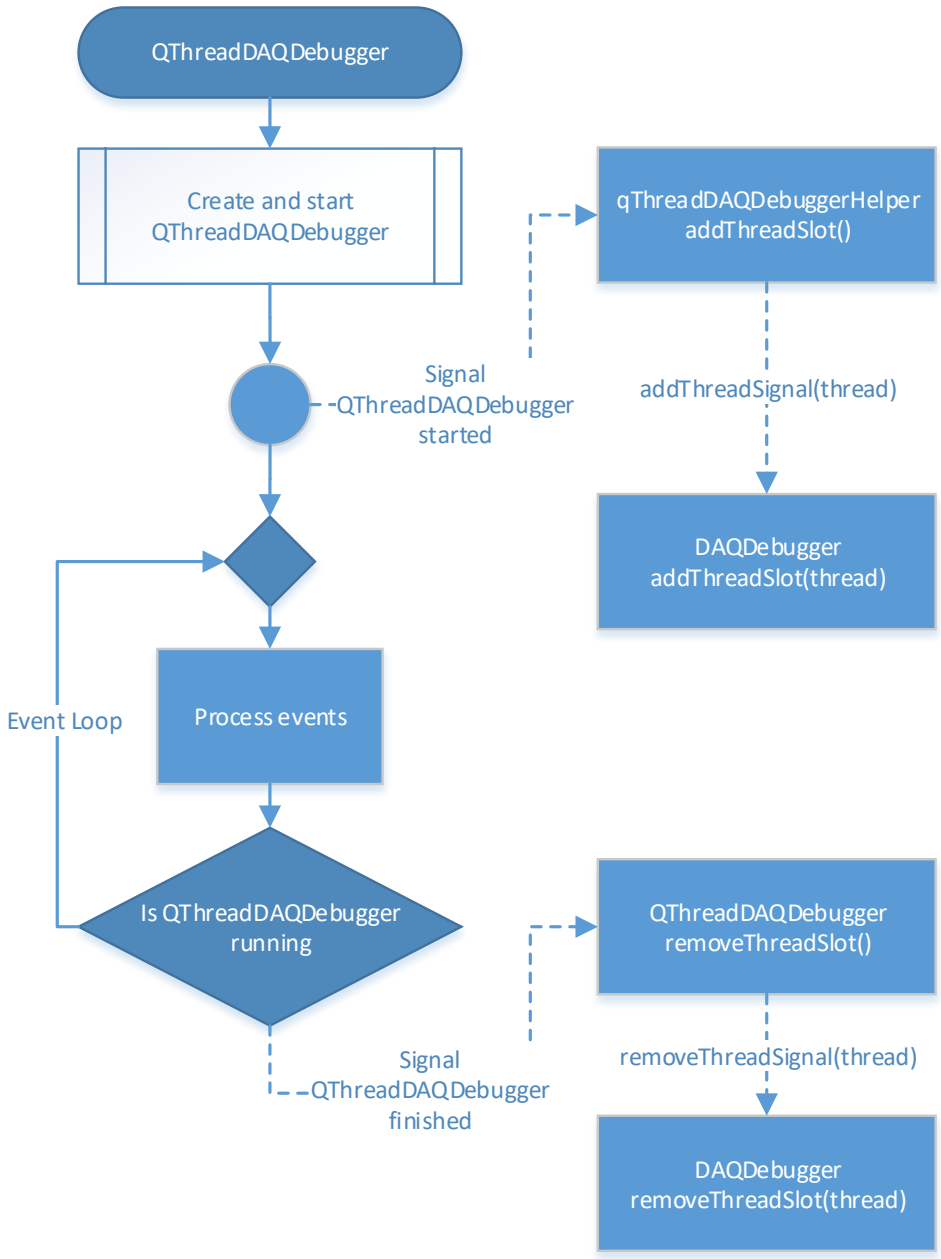

Figure 4.2: Flow diagram of the thread life cycle in the DAQ Debugger.

The description of thread life cycle gives a comprehensive insight from a global point of view. The diagram of the thread life cycle is shown in Figure 4.2.

The `QThreadDAQDebugger` object inheriting from `QThread` object is created and the thread is started. The signal `started()` is emitted and it is connected to the slot `addThreadSlot()` of `qThreadDAQDebuggerHelper` object. This object has already been moved to the thread of `QThreadDAQDebugger` object in the `QThreadDAQDebugger` object constructor, since it must live in this thread. This is the way how to force the execution of `DAQDebugger::addThreadSlot(thread)` static method in this thread and thus, the DAQ Debugger gets thread ID. Finally, the slot `addThreadSlot()` of `qThreadDAQDebuggerHelper` object is emitting the signal `addThreadSignal(thread)` being connected to the `DAQDebugger::addThreadSlot(thread)` static method. This concept ensures the execution of `DAQDebugger::addThreadSlot(thread)` in the thread itself.

Moreover, Figure 4.2 covers the concept when a thread is finishing its execution. The thread must unregister in the DAQ Debugger. When the `QThreadDAQDebugger` object finishes its execution the signal `finished()` is emitted and it is connected to the slot `removeThreadSlot()` of `QThreadDAQDebugger` object. There the signal `removeThreadSignal(thread)` is emitted and it goes directly to the `removeThreadSlot(thread)` of DAQ Debugger. In comparison with the registration procedure, the unregistration procedure is much simpler, since the DAQ Debugger already knows the finished thread. Since the `QThreadDAQDebugger` object lives in the main thread, no one has to worry about handling of emitted `QThreadDAQDebugger` signals after the thread has finished its execution. These emitted signals are handled by the main thread.

To finish the discussion concerning the thread registration procedure properly, the description of registration of $n \in \mathbb{N}$ threads when a process starts is given. In Figure 4.3, the diagram begins with the main thread. First of all, the main thread starts its execution. It registers system signals and registers the main thread in the DAQ Debugger. Whole mentioned functionality is encapsulated in the `DAQDebugger::init(argv[0])` static method. Then, the main thread continues its execution and processes events. All remaining of $n \in \mathbb{N}$ threads are registered in the DAQ Debugger afterwards. The detail of registration procedure for each thread was already mentioned and it triggered with the emitting of signal `started()`. Of course, whenever some of $n \in \mathbb{N}$ threads finish their execution, they are unregistred from the DAQ Debugger. For simplicity reasons, the unregistration procedure is not depicted in the diagram.

Probably the most important scenario is the crash of a process. This situation is described in Figure 4.4. From a process start, the DAQ Debugger is a part of a process and standing in the background of a running process. If the process is running smoothly without any single crash, the DAQ Debugger does not take any action. For this reason, the DAQ Debugger does not affect the process performance and does not increase load on readout engine computers at all.

The system signals are registered, the process continues its execution. Once the crash of process occurs, the DAQ Debugger handles it. The system signal is emitted and it is caught by the signal handler of crashed thread in the DAQ Debugger. At this point, it is important to realize the crashed thread where crash has occurred is responsible for the control of all remaining threads, memory dump storage and

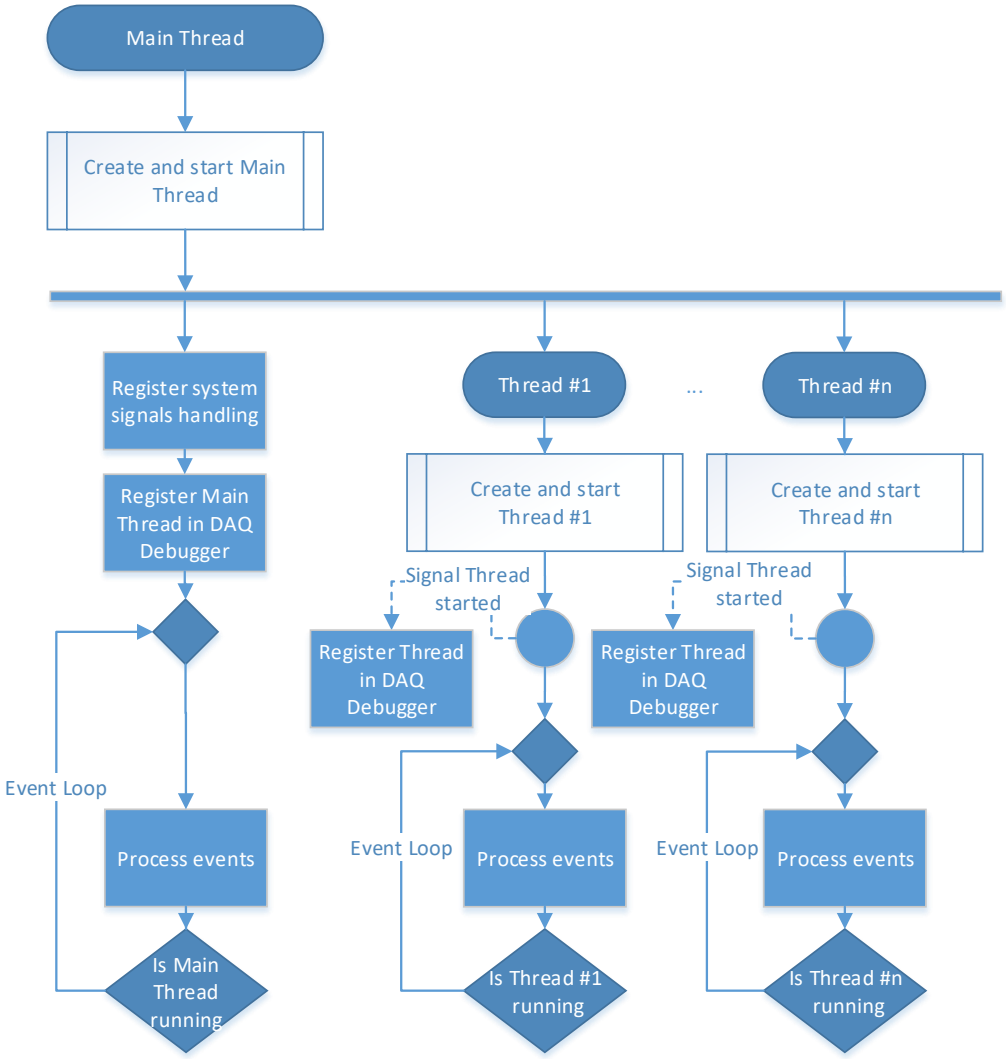

Figure 4.3: Flow diagram of the thread registration procedure in the DAQ Debugger.

creation of a crash report.

Firstly, the crashed thread sends the suspend signal to all remaining threads. It is necessary to suspend them otherwise they would continue their execution and thus, the exact point of crash would be lost. Then, the memory dump is produced and stored. The memory dump can be easily loaded to Qt Creator (Debug → Start Debugging → Load Core File) and memory can be investigated as much as by conventional debugging.

Secondly, the report file is created and open for writing. The crashed thread writes its stack trace to the file.

Afterwards, the control of all suspended threads is started. The crashed thread sends the resume signal to first suspended thread and the crashed thread itself is suspended. The resumed thread writes its stack trace to the file, then sends the

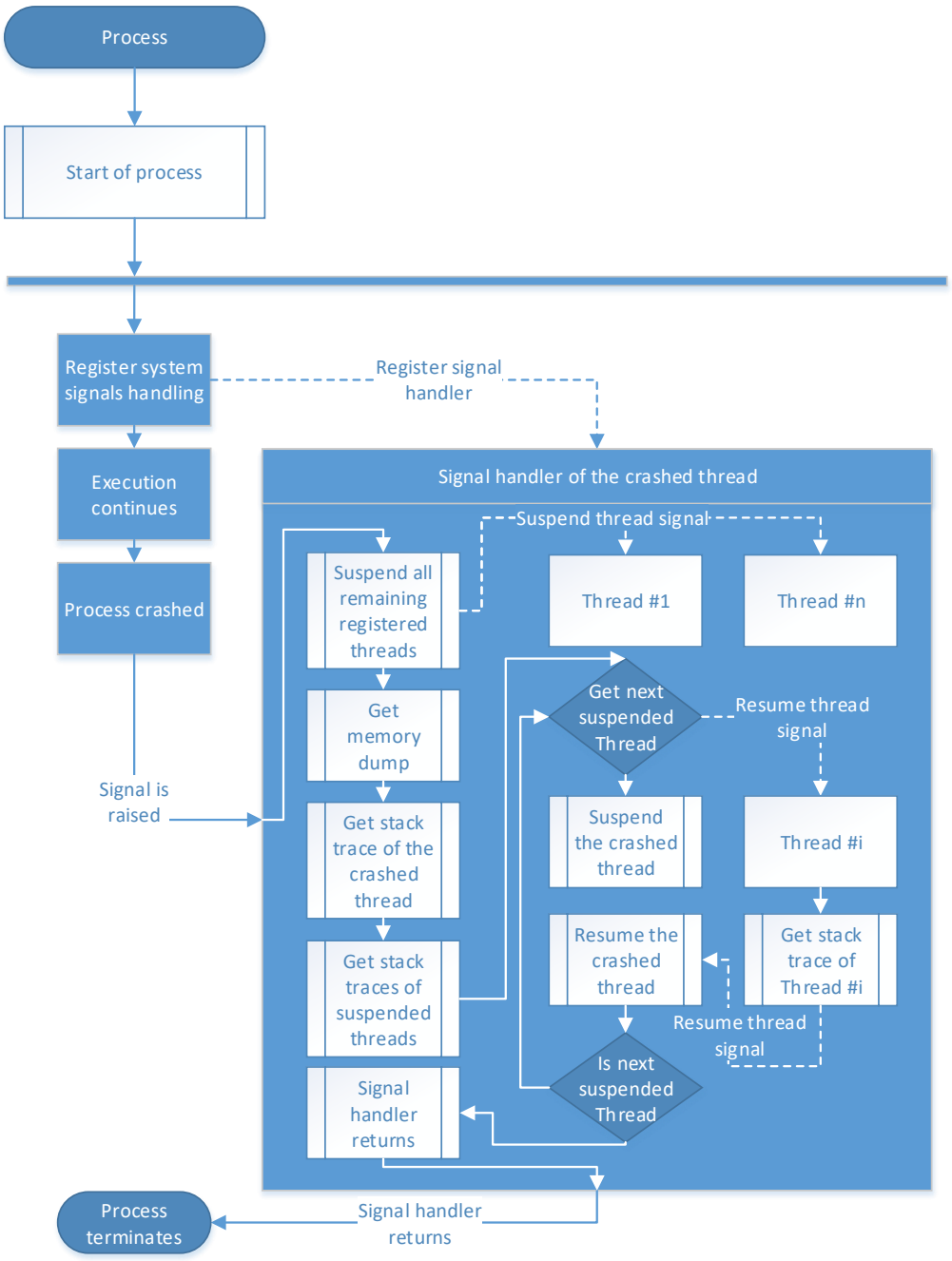

Figure 4.4: Flow diagram of the thread crash caught and handled by the DAQ Debugger.

resume signal to the crashed thread and is suspended again. The resumed crashed thread sends the resume signal to second thread and it is again suspended. The second resumed thread writes its stack trace to the file, then sends the resume signal to the crashed thread and is suspended again. It continues in this way to the last suspended thread. The resumed crashed thread (resumed by the resume signal

sent from $(n-1)$-th thread) sends the resume signal to n-th thread and it is again suspended. The n-th resumed thread writes its stack trace to the file, then sends the resume signal to the crashed thread and is suspended again. This suspend/resume procedure ensures the serial writing to file and proper thread control. Finally, the report file is closed and process is exiting with the caught signal in the crashed thread. The whole control of threads, memory dump storage, opening and closing of report file is controlled by the crashed thread.

4.5 The iFDAQ Stability

Since beam time is highly valuable and it is usually provided 24/7 for most of the calendar year, high reliability of the DAQ system is of major importance to high-energy physics experiments. This section discusses the iFDAQ stability over the last few years.

There are three main sources of instabilities leading to time periods when no physics data can be taken. The overall time period when no physics data can be taken is called the iFDAQ downtime. On the other hand, the iFDAQ uptime denotes the overall time period when the iFDAQ is stable and ready for a proper data taking.

The first source of instability is a memory access error (PCI/DMA) caused by scrambled data being transferred to the RAM of the readout engines. It is the most time-consuming failure, since it requires to reboot all readout engine computers and the recovery procedure takes approximately 10 minutes on average. The second one is an unrecoverable loss of synchronization in the hardware event builder leading to a safe stop of a run. The safe stop of a run might be considered as one of the intelligent elements of the iFDAQ, since a safe stop prevents more serious problems which would lead to the higher downtime. The contribution of these two problems to the overall system downtime decreased during the course of one year due to better commissioning and calibration of detectors and consequently higher data quality.

The third source of the downtime is based on unknown software crashes being not fully understood. The DIALOG library and, especially, the DAQ Debugger have been developed in order to eliminate software errors in the iFDAQ and thus, to improve the iFDAQ software stability.

Figure 4.5 shows the iFDAQ stability in the last few years. The values are calculated by parsing messages that have been stored by the MessageLogger in a database.

The plot shows data from August 2015 to October 2017. Each bar shows the number of hours per month representing the total downtime for this period and, moreover, it is split into three components – PCI/DMA errors (blue), event builder desynchronization (red) and software errors (yellow). In addition, the relative uptime per month is stated in the top of each bar.

In August, September and October 2015, messages that were stored do not include information about the kind of error. Thus, it can not be distinguished between event builder desynchronization and software errors (purple).

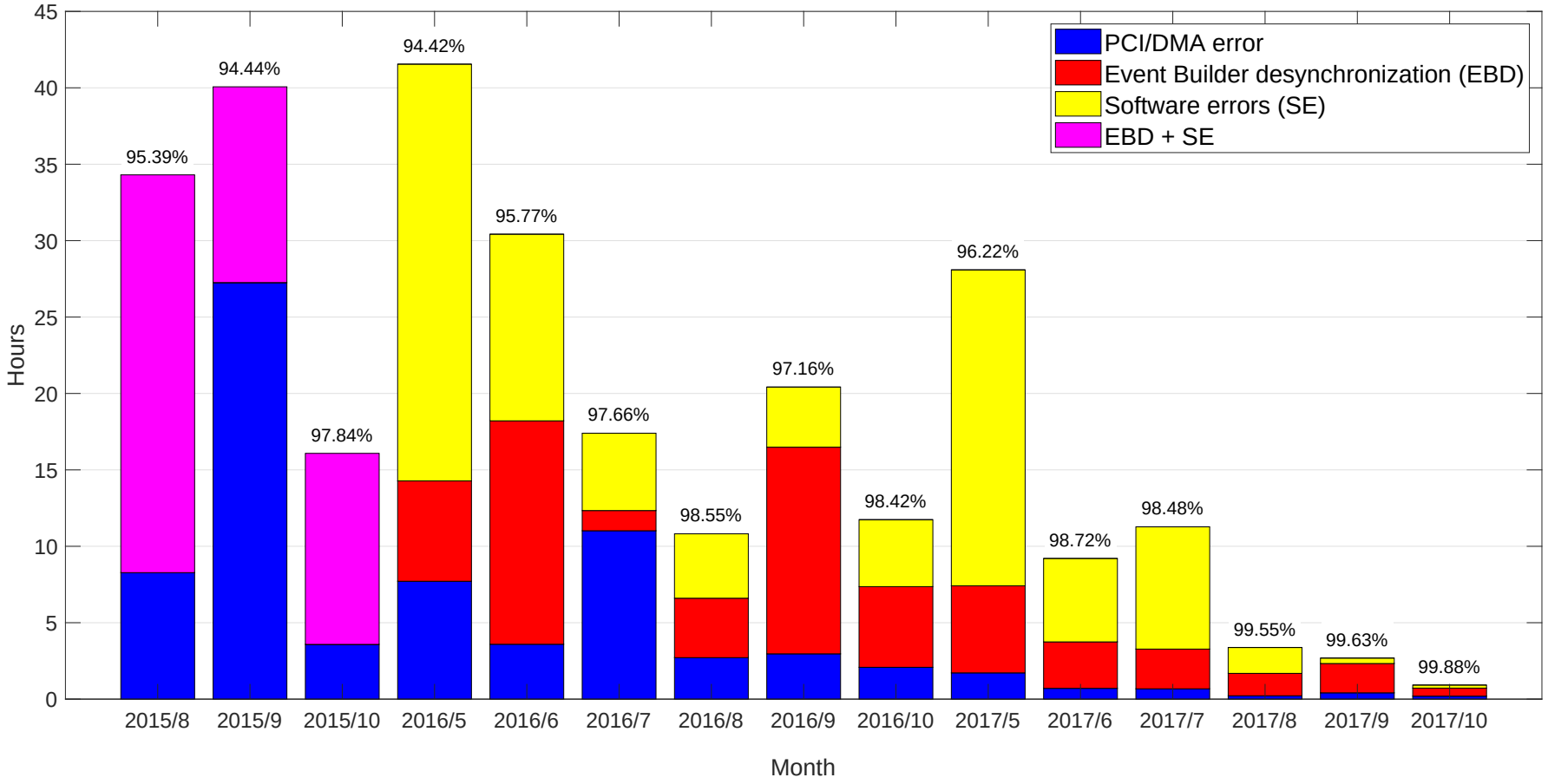

Figure 4.5: Absolute downtime of the iFDAQ per month and corresponding relative uptime.

From the plot, it can be seen that in 2015, most of the 40 hours downtime per month were caused by software errors.

The DIALOG library integration to the iFDAQ is the first big milestone in the iFDAQ stability improvement. The DIALOG library was introduced to the iFDAQ in June and July 2016. The bar plot in Figure 4.5 shows a significant drop in the software errors component of each bar from that time.

The second milestone is located in June 2017 when the DAQ Debugger has been deployed and helped to identify all remaining software errors in the following months.

Finally, the iFDAQ has reached the state when no software errors are present anymore. The last software error was observed at the end of September 2017.

Presently, the iFDAQ reaches a system availability of 99.88% and only PCI/DMA errors and event builder desynchronization appear from time to time. In absolute numbers, the downtime of the iFDAQ has decreased from around 40 hours per month in 2015 to only 1 hour per month in October 2017.

The increase of downtime in May of each year can be explained by the commissioning phase that takes place in the beginning of each year and in which all detectors, the frontend electronics and the iFDAQ do not operate in stable conditions.

Chapter 5

The Continuously Running iFDAQ

Recently, a stability of DAQ has become a vital precondition for a successful data taking in high-energy physics experiments.

In the previous chapters, all aspects which were essential in making the iFDAQ stable were discussed. Firstly, the DIALOG library has been developed and replaced the unreliable DIM library. The implementation followed by the integration to all processes represents the first successful milestone in the iFDAQ stability achievement. The IPC became reliable and robust for the very first time.

Unfortunately, the effort dedicated to the IPC improvement was not enough and the undetected bugs, inconsistencies and improper synchronization leading to the iFDAQ misleading and malicious behaviour were still present in the iFDAQ.

Therefore, a new challenge dealing with the resolution of all remaining issues arose. To detect these issues, the DAQ Debugger offered an alternative to the widespread conventional debugging being not effective and applicable. It helped to reveal all remaining software issues and improved significantly the stability of the system.

Once the iFDAQ stability ensured a smooth data taking, the last challenge came along. It requires such a version of the iFDAQ running in a mode without any stops where no data are lost. DAQ systems fulfilling such requirements reach the efficiency up to 99%. The iFDAQ runs nonstop 24/7 regardless of nights, weekends or bank holidays for most of the calendar year. Thus, it puts stress on reliability and robustness of the system. Every undesirable interruption of data taking results in a possible loss of physics data. Thus, the state with the continuously running iFDAQ is about to achieve [64].

Another significant contribution to the loss of beam time originates from the time that is needed to initiate a synchronized data flow through the event builder. Establishing synchronous processing of data by all involved hardware nodes is achieved by distributing trigger and spill cycle information to all nodes via the Trigger Control System (TCS) [35] and applying reset commands and timeouts. Following section describes how the SPS spill cycle information is used initiate and maintain proper synchronization.

5.1 The Proper Timing and Synchronization

Before the continuously running mode is presented in a deeper way, it is worth of mentioning how to deal with accurate timing and synchronization being absolutely essential for such a mode [64]. To synchronize the iFDAQ processes correctly, the iFDAQ needs to take advantage of some proper timing mechanism. The SPS super cycle is a good candidate offering and ensuring the proper timing.

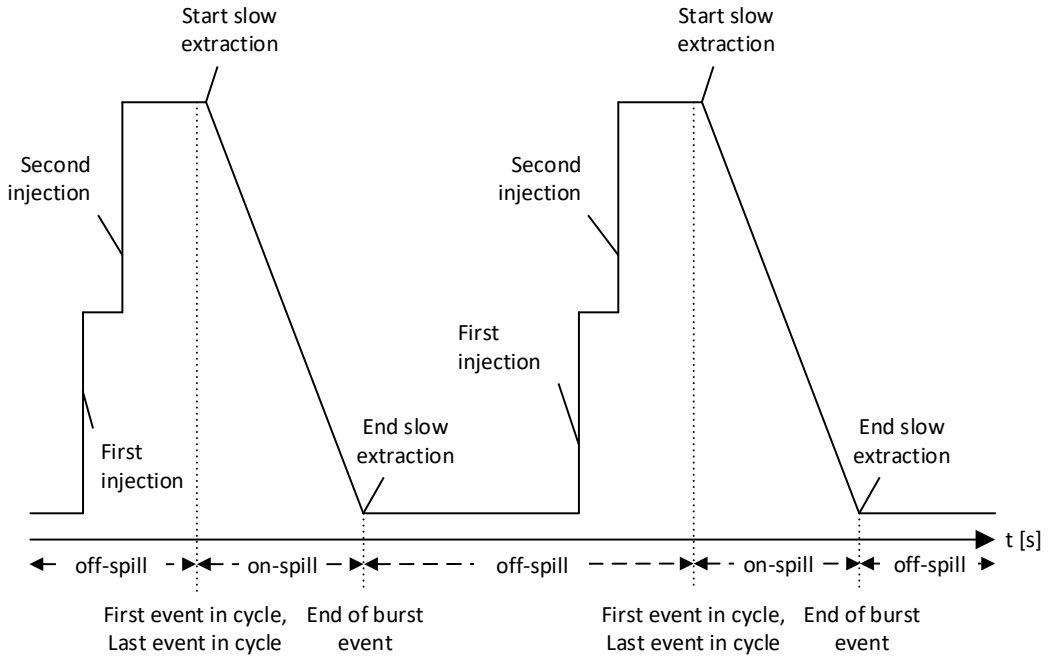

Figure 5.1: Particle intensity in the SPS for an exemplary SPS cycle.

The beam for COMPASS is provided by the SPS. As a circular particle accelerator, the SPS provides beam to experiments in bursts, called spills. Before being able to deliver a particle spill to COMPASS, the SPS has to be filled with proton bunches, the bunches have to be accelerated to the desired energy, and the particle load inside the SPS has to be debunched. Only when the particles circulate homogeneously distributed in the SPS, a spill of stable particle intensity can be delivered. After one filling is completely extracted from the machine, the cycle starts over. Figure 5.1 shows the proton intensity in the SPS during two cycles which together form an exemplary SPS super-cycle.

For COMPASS, the load in the SPS is slowly extracted over the course of 4.8 seconds. After the spill, there are at least 5 seconds without spill: filling the SPS with two injections from the PS takes approximately 2 seconds, and ramping up the energy and debunching (flat top in Figure 5.1) takes another 3 seconds. However, the off-spill period can take significantly longer -– depending on the SPS super-cycle -– when the next charge of the SPS is not conducted to COMPASS but to other recipients (e.g. the LHC).

For a successful synchronization of the iFDAQ to the SPS cycle, dedicated types of triggers have been established. In the iFDAQ, following types of events are distinguished:

- **Start of run event** is first event in a run after it starts for the first time.
- **End of run event** is last event in a run before it terminates.
- **First event in cycle/Start of burst event** is first event in burst (start of beam extraction) and hence in the on-spill period. At the same time, it marks the beginning of a cycle.
- **Last event in cycle** is last event in the off-spill period, it is followed by a reset signal for buffer memory in the frontend electronics ensuring proper synchronization for the next spill.
- **End of burst event** is last event in burst (end of beam extraction and on-spill period).
- **Physics event** contains real physics data and is collected during the on-spill period.
- **Calibration event** can be collected in both the on-spill and off-spill period. It contains calibration data for calorimeters.

Timing and synchronization are based on the artificial events and their correct order. To initiate data flow through the event builder and establish correct timing, three cycles are required – and thus lost – before first physics triggers can be sent to the frontend electronics.

5.2 The Continuously Running Mode

The continuously running mode was introduced to the iFDAQ before the Run 2017 and from then on, it is an integral part of the iFDAQ [64]. To collect more physics data, the iFDAQ with the continuously running mode must take data nonstop without any useless user interventions. Therefore, the continuously running mode has to provide a functionality for a proper transition between two consecutive runs. The transition demands appropriate synchronization, because it must be successfully done within a very short time period (approximately one second).

To safe time-consuming stop and restart of the data flow between two runs, the continuously running mode was introduced. It ensures a smooth transition between two consecutive runs without intervention of the shift crew and without stop of data flow through the event builder. The transition between two runs requires several important things to do. Data files from the previous run must safely be closed and new files for a new run have to be opened in the same time on all machines. The run number has to be increased in the correct time so that data belonging to the previous run and data belonging to the new one are properly distinguished. The

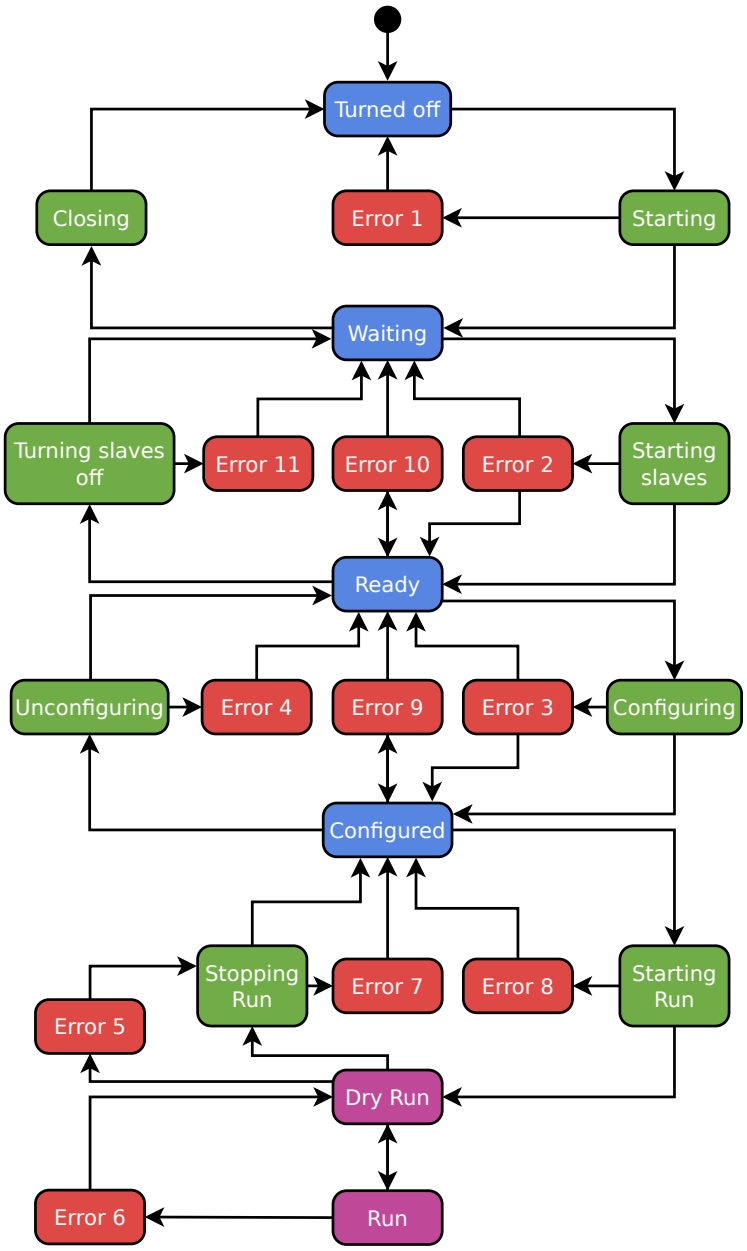

Figure 5.2: The iFDAQ state machine diagram [46].

records in the electronic logbook concerning the previous run have to be filled in and new records concerning the following run have to be created. All these functionalities require proper synchronization.

Before the continuously running mode is discussed in a deeper way, it is necessary to show the iFDAQ state machine, see Figure 5.2, and to give more details to particular states. The iFDAQ can be found in one of the following states (ID represents the value of the given state) [46]:

1. **Turned off** (ID: N/A) – An abstract state which represents that the Master process is turned off.
2. **Starting** (ID: 21) – The Master process is being initialized.

3. **Waiting** (ID: 1) – The Master process is initialized and is ready to receive commands.

4. **Closing** (ID: 22) – The Master process is being turned off (ends threads).

5. **Starting Slaves** (ID: 23) – The Master process is starting the slave processes.

6. **Ready** (ID: 2) – The Master process has received confirmation that the slave processes are turned on.

7. **Turning Slaves off** (ID: 24) – The slave processes are being turned off by The Master process.

8. **Configuring** (ID: 25) – The Master process is retrieving configuration information from the database and configuring the slave processes.

9. **Configured** (ID: 3) – The Master process has received confirmation that the slaves are configured and ready for a run.

10. **Unconfiguring** (ID: 26) – The Master process is unconfiguring the slave processes.

11. **Starting run** (ID: 27) – The Master process has sent the command to initiate a run and is waiting for the slaves to enter the Dry run state.

12. **Stopping run** (ID: 28) – The Master process has sent the command to stop the run and is waiting for the slaves to enter the Configured state.

13. **Dry Run** (ID: 11) – The Master process has received confirmation that the slaves have entered the Dry run state and initiated the data-taking process (verification and debugging purpose).

14. **Run** (ID: 12) – The Master process has received confirmation that the slaves have entered the Run state and initiated the data-taking process (real data taking).

15. **Error** (ID: 41-48) – Error 1-8.

Moreover, the idea of maintaining data flow in the event builder has been extended to periods when no data taking is requested. To do so, a new state, the so called Dry Run (state 11) was introduced. It maintains the data flow through the whole acquisition chain and provides a monitoring data stream to the online monitoring tools, but the acquired events are not written to hard drives. Thus, it serves as verification, monitoring, and diagnostic stage, even in periods when the experiment is not ready for data taking. The consecutive step relevant to real data taking is called Run (state 12) and its start is possible on the next delivered spill, since synchronization is already established. Using the Dry Run state and a smooth transition between runs, the data flow in the iFDAQ is only stopped in case of serious errors (see Section 4.5) or in case of interventions on detectors that require a stop of trigger distribution to the frontends.

At this point, basic functionality requirements are specified. Using the continuously running mode, the Dry Run state is designed in order to meet the following requirements:

- It checks the incoming data stream from the detectors on consistency.
- The transition from Dry Run to Run and vice versa must be fast and smooth.
- It uses the artificial run number in the range from 999,990 to 999,999.
- Starting procedure – The Master process sends a command to initiate a run and waits for the slaves to enter the Dry Run state.
- Terminating procedure – The Master process sends a command to the slaves to stop a run.
- It provides physics data to online monitoring tools.
- No data are stored in files.
- No records in the electronic logbook concerning current run are created.
- The spill number is iterated. After the last spill, the spill counter is reset to 1 in the following spill and the artificial run number is changed.

Similar characteristics can be summarized also for the Run state. Using the continuously running mode, the Run state must meet the following requirements:

- It is started from the active Dry Run state.
- It always uses the last real run number incremented by one.
- Starting procedure – It sends a command to reset the spill counter to 1 and set the real run number after the end of current spill.
- Terminating procedure – A run stops in the Run state and moves to the Dry Run state where a new run starts.
- It provides physics data to online monitoring tools.
- The data are stored in files.
- Records in the electronic logbook concerning current run are created.
- The continuously running mode can be switch on/off in Runcontrol GUI.
- If a run is stopped manually or automatically (by reaching the maximum number of spills set in Runcontrol GUI) in the Run state, the iFDAQ moves to the Dry Run state.

To sum it up, the iFDAQ using the continuously running mode is a self-running data-taking system where decisions related to the continuously running mode are taken based on delivered events. Timing critical actions that have to be taken in transitions from one run to another or from the Dry Run state to the Run state and vice versa are executed on reception of artificial events and depend on their type of trigger:

- **First event in cycle in first spill of a run** opens/closes files and create the electronic logbook records, if necessary.
- **End of burst event in last spill of a run** resets the spill counter and set the next run number.
- **Last event in cycle in last spill of a run** fills in the electronic logbook records and moves to Dry Run/Run, if necessary.

5.3 The Logic in the Master Process

The whole synchronization mechanism in the Master process is based on artificial events, i.e., first event in cycle, end of burst event, last event in cycle.

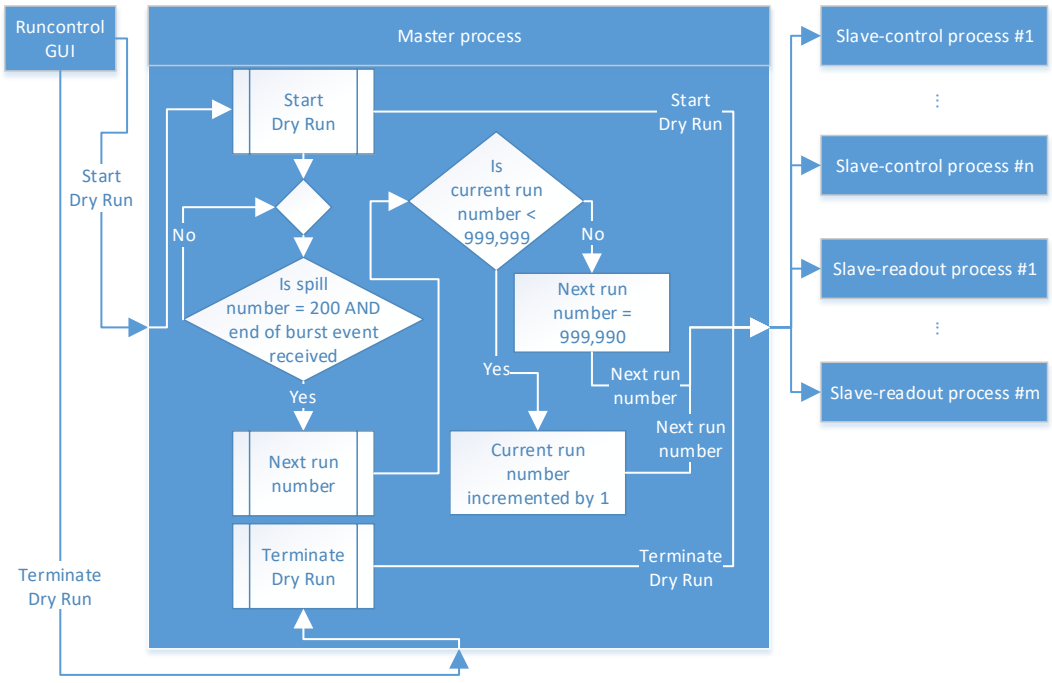

Figure 5.3: The logic in the Master process for the Dry Run state.

A user starts the Dry Run state from the Runcontrol GUI, see Figure 5.3. The command is sent to the Master process where it is forwarded to n Slave-control processes and m Slave-readout processes. Once all processes are in the Dry Run

state, the data are readout. It may remain in the Dry Run state and create runs with 200 spills and with the run number in the range from 999,990 to 999,999. In the Dry Run state, the physics data are processed in online monitoring tools and not stored in files. In addition, no records are created in the electronic logbook. Therefore, the Dry Run state is a useful verification step before the real data taking.

Once a user decides to move to the Run state, it can be either the Run state with the continuously running mode creating runs with the set number of spills or the Run state without the continuously running mode terminating after the set number of spills and moving back to the Dry Run state. The default value for the number of spills is 200.

Firstly, a discussion related to the automatic transition between two consecutive runs is given. The whole logic is stated in the last spill of current run. If the end of burst event occurs in the last spill of current run, the Master process sends the next run number to all slaves and executes a DIM command in order to reset current spill number. Thus, the next spill starts with the spill number 1. If the iFDAQ is in the Dry Run state, then the next run number is in the range from 999,990 to 999,999 and it goes again from 999,990 after 999,999. If the Run state is active, the run number in the Run state is incremented based on the maximum run number stored in the database. Especially because of online monitoring tools, the run number must be changed even in the Dry Run state. All online monitoring tools restart all statistics about current run if the run number is changed. Statistics are not affected by the previous run at all.

The last but not least automatic transition between two consecutive runs remaining to be discussed is actually a transition between two states – from the Run state to the Dry Run state. If the iFDAQ is in the Run state, the continuously running mode is disabled and the end of burst event occurs in the last spill (the set number of spills) of current run, the Master process sends the next run number equal to 999,990 to all slaves and it also sends a DIM command to reset current spill number. Thus, the next spill starts with the spill number 1, with the run number 999,990 and the iFDAQ moves to the Dry Run state.

Secondly, the manual transition between two consecutive runs (the Dry Run state to the Run state and vice versa) is also worth of mentioning. Once a user goes from the Dry Run state to the Run state and vice versa, the transition is done immediately and a DIM command to reset the spill counter is sent. Moreover, the next run number is sent to all slaves. In fact, a new run is started in the next spill with the spill number 1 and the run number equal to the next run number. The run number in the Run state is incremented based on the maximum run number stored in the database. On the other hand, the run number is set to 999,990 in the Dry Run state after the transition from the Run state.

A user starts the Run state from the Runcontrol GUI, see Figure 5.4. In order to move to the Run state, the iFDAQ must be in the Dry Run state already. The command is sent to the Master process where it is forwarded to n Slave-control processes and m Slave-readout processes. In the Run state, the Master process creates a record for the electronic logbook in the database for each run. At the

beginning of each run in the Run state, each Slave-readout process opens a file for writing of physics data. At the end of each run in the Run state, the Master process finishes a record for the electronic logbook in the database and each Slave-readout process closes the file for writing.

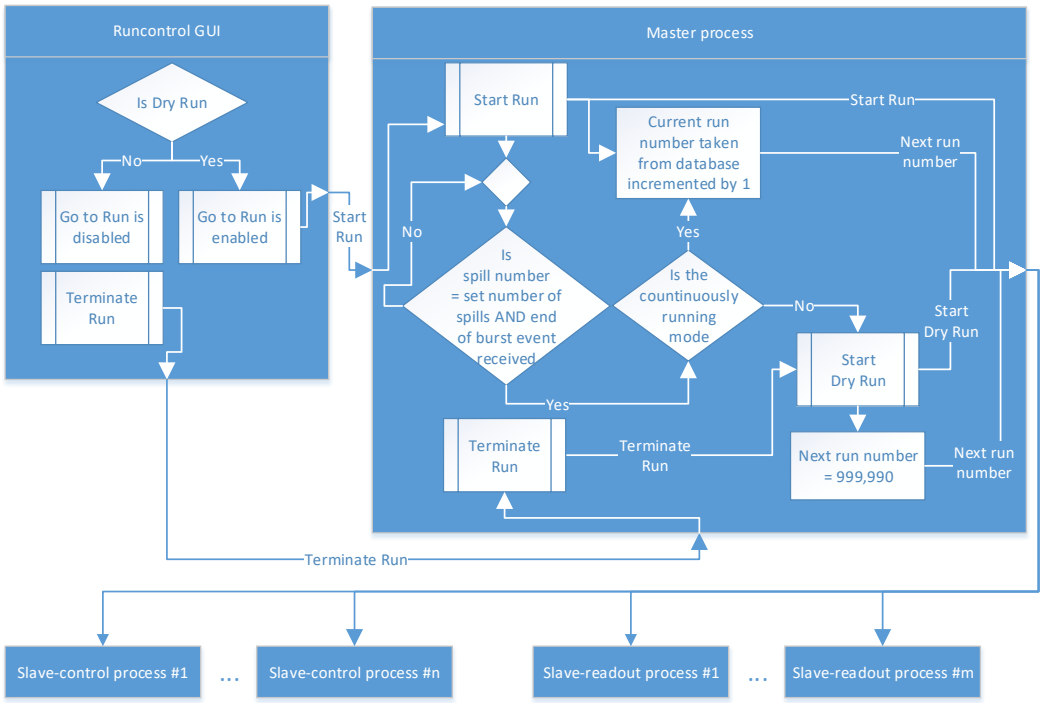

Figure 5.4: The logic in the Master process for the Run state.

Moreover, a user sometimes needs to switch off continuously running mode even during the Run state so that it could terminate after the set number of spills and move to the Dry Run state. It is usually used if a source of error (detectors, frontend electronics, etc.) not being so urgent to be immediately reloaded occurs. The affected equipment can be less important and a user would like to have a full run with the set number of spills. Of course, if a user needs to stop a run running in the continuously running mode in the Run state immediately, it can be done manually and then, it moves to the Dry Run state promptly.

All commands are generated in the Runcontrol GUI and sent to the Master process. The Master process checks consistency and integrity of the iFDAQ. Therefore, it triggers the state transitions in the state machine and controls all slaves via commands. Each slave has its own state machine for the state transitions.

If the continuously running mode is disabled in the Run state, a proper transition from the Run state to the Dry Run state after the set number of spills must be ensured. This synchronization must be precise, since the iFDAQ should stay in the Run state until the last event of the last spill is processed and move to the Dry Run state afterwards.

This particular synchronization procedure is done in two steps. Firstly, a command

indicating to the slaves that they must prepare for a transition from the Run state to the Dry Run state after the end of current spill. Once the last event of the last spill occurs, the Master process sends the final command indicating to the slaves that they must finish the transition procedure from the Run state to the Dry Run state. It ensures that all events readout by the Slave-readout processes are stored in the files, i.e., no event from the Run state is lost and no event from the Dry Run state is stored.

It was already mentioned, a user can change whether to use the continuously running mode during the Run state or not. However, it is not allowed to go from the discontinuously running mode to the continuously running mode in the last spill. In this case, the transition procedure from the Run state to the Dry Run state has already begun.

5.4 The Logic in the Slave-readout Process

The whole logic is designed in order to meet requirements for a transition between two consecutive runs, i.e., the Dry Run state and the Dry Run state, the Dry Run state and the Run state, the Run state and the Run state, the Run state and the Dry Run state.

It was already mentioned, a DIM command is used in order to reset the spill counter, i.e., the spill counter is starting with the spill number 1 in the next spill. Moreover, a command with the next run number is delivered to all slaves too.

The main idea is based on a recognition of the first event of a new run. Unfortunately, the first event in cycle can not be used, because it is delivered to only one readout engine computer.

Instead, it uses the spill number and the event number for a detection of a new run detection. The event number is a unique number for each event in a spill counting from 1. Basically, the Slave-readout process is waiting for the first event with the spill number set to 1 and the event number smaller than the previous event number. This condition ensures a precise recognition of the first event in a new run on each readout engine computer.

The run number is filled in the event header in the Slave-readout process. All physics monitoring tools restart their statistics of collected events when a new run is started. For that reason, the synchronization must be very precise.

Moreover, each event has an attribute *saveToFile*. The attribute indicates whether the events are going to be stored in the files or not. Apparently, it is set to false in the Dry Run state and true in the Run state. Based on the run number and the *saveToFile* attribute, the file for current run is closed and a new one is open for a new run.

No data are stored in files in the Dry Run state. The Master process sends the run number to the Slave-readout process and the Slave-readout process fills in the run number to all events starting from the next spill.

Once a user goes from the Dry Run state to the Run state, this transition is done immediately. The run number for the Run state is delivered to the Slave-readout process from the Master process. The data relevant to the Run state are readout in the next spill starting with the spill number 1 and the new run number is filled in the header of events. It continues until the last spill of current run, when the new run number is again delivered and assigned to all events starting from the next spill.

Once a user needs to return back to the Dry Run state, it is done manually. The data are not stored to files anymore and it is done immediately. For instance, the data from only one half of current spill could be stored in the files in that case.

There is also a possibility of the automatic transition from the Run state to the Dry Run state after the set number of spills. In this case, all events from the set number of spills are stored in the files. It stays in the Run state for the whole last spill. It receives the first command in the last spill from the Master process that it must prepare for the transition to the Dry Run state and it receives the second command when it reads out the last event of cycle and the transition to the Dry Run state is finished.

The possibility of changing the continuously running mode to the discontinuously running mode even during the Run state has already been mentioned. The Slave-readout process is not affected at all, because the continuously running mode is controlled by the Master process. In that case, the procedure of the transition to the Dry Run state is performed in the same way as it would have been set it up even before the start of the Run state itself.

5.5 Contribution of the Continuously Running Mode

Before the incorporation of the continuously running mode, the procedure for starting a new run after another run was successfully finished was always connected to a loss of beam time. Firstly, there is loss due to the already mentioned necessary synchronization phases. The start of run procedure requires three spills for the synchronization of the TCS with the SPS cycle. The terminating procedure takes one spill to end up data taking. Secondly, there is beam time loss due to necessary human intervention. Before the incorporation of the continuously running mode, a run had to be started manually. An inattentive shift crew could hence add a significant part to the beam time loss by not starting the next run right after the previous run is stopped. Eliminating both factors, the continuously running mode contributes to the efficiency of data taking.

In Table 5.1, the contribution of the continuously running mode is demonstrated. In order to eliminate problems unrelated to the iFDAQ, one day with smooth data taking is selected for the Run 2016 (without the continuously running mode) and the Run 2017 (with the continuously running mode). The loss unrelated to the iFDAQ on 2017-07-30 refers to the beam polarity change procedure. In case of smooth data taking, the percentage of collected spills reaches almost 100% in case of the

continuously running mode. Without the continuously running mode, the loss of spills is around 3.5%.

	2016-10-10		**2017-07-30**	
	# spills	**%**	**# spills**	**%**
Delivered from SPS	4,648	—	4,569	—
Recorded	4,487	96.54	4,488	98.23
Loss related to TCS synchronization	112	2.41	4	0.09
Loss related to inattentive shift crew	49	1.05	0	0
Loss unrelated to the iFDAQ	0	0	77	1.69

Table 5.1: The contribution of the continuously running mode.

Chapter 6

Load Balancing

Generally, in computing, Load Balancing (LB) [42] improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives. LB aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Using multiple components with LB instead of a single component may increase reliability and availability through redundancy. LB usually involves dedicated software or hardware, such as a multilayer switch or a Domain Name System server process.

For the iFDAQ, the most challenging task from the LB point of view is load balancing at the multiplexer (MUX) level. The optimization criterion is minimization of the difference between the output flows of the individual multiplexers. This minimization is achieved by remapping the connection of inputs to input ports of the multiplexers. Each input port establishes a connection between a data source (a de-

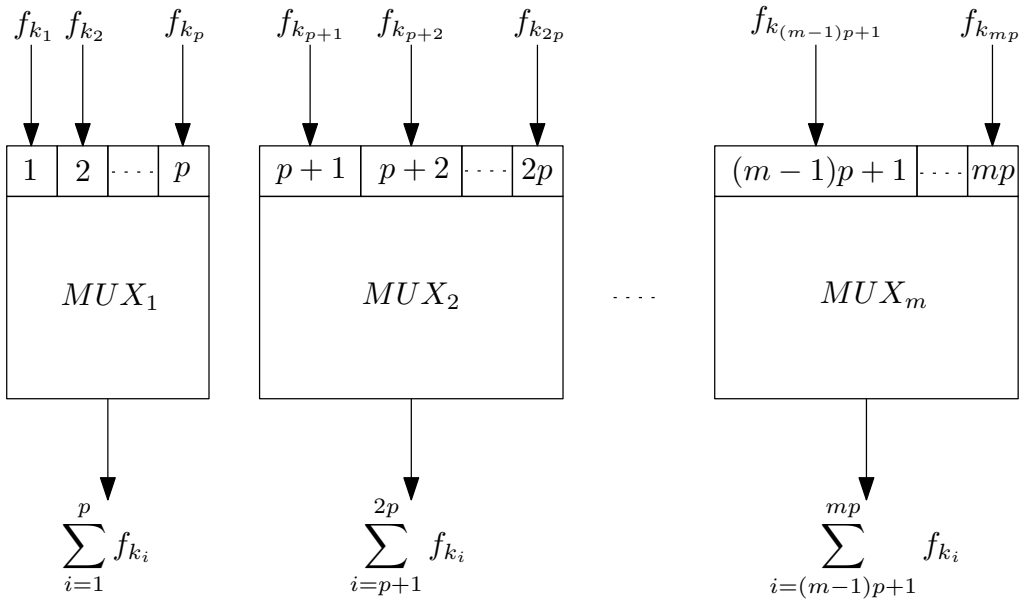

Figure 6.1: Visualization of LB at the MUX level.

tector or a data concentrator) and the MUX level. For the COMPASS experiment, it is necessary to consider flows varying from 0 B to 10 kB for each input port.

In Figure 6.1, a visualization of LB at the MUX level is given. There are m MUXes with p ingoing ports each. Moreover, $n \in \mathbb{N}$ flows $f_{k_1}, f_{k_2}, \ldots, f_{k_{mp}} \in \mathbb{N}_0$, where $n = m \cdot p$, are shown in the figure with indices $k_1, k_2, \ldots, k_{mp} \in \{i \mid 1 \leq i \leq n\} \wedge \forall i, j : k_i \neq k_j$.

Despite the fact that each flow varies from 0 B to 10 kB in the COMPASS experiment, the domain $\mathbb{N}_0$ is used. The motivation comes from a general approach to LB. Moreover, a flow with 0 B can be either a physical connected input port sending no data or an empty input port where no data source is connected to. In brief, there are always $n = m \cdot p$ flows regardless whether all ports are used or not.

6.1 Problem Formulation

This subsection deals with a proper definition of the LB problem and preparation for discussion of the complexity of the LB problem. The Multiple Knapsack problem [43] is useful for the examination of the LB problem complexity as it can be shown that there exists a polynomial reduction from the MK problem to the LB problem. As MK problem is $\mathcal{NP}$-complete, this implies the LB problem is $\mathcal{NP}$-complete.

Definition 1. *Let $m \in \mathbb{N}$ denote the number of MUXes with $p \in \mathbb{N}$ ingoing ports each, i.e., $n = m \cdot p \in \mathbb{N}$ ingoing ports in total and flows $f_1, f_2, \ldots, f_n \in \mathbb{N}_0$. Let $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ be subsets of indices and $F = \left\lceil \sum_{i=1}^{n} f_i / m \right\rceil$ be a theoretical average flow for one MUX. The Load Balancing (LB) problem is an optimization problem such that:*

To minimize

$$\sqrt{\sum_{i=1}^{m} \left(F - \sum_{j \in \mathcal{S}_i} f_j \right)^2}, \tag{6.1}$$

subject to the constraints

- *each flow must be allocated*

$$\bigcup_{i=1}^{m} \mathcal{S}_i = \{i \mid i \in 1, \ldots, n\} \tag{6.2}$$

- *each flow must be allocated at most once*

$$\mathcal{S}_i \cap \mathcal{S}_j = \emptyset \quad \forall i, j = 1, \ldots, m \wedge i \neq j \tag{6.3}$$

- *each MUX has p ports*

$$|\mathcal{S}_i| = p \quad \forall i = 1, \ldots, m \tag{6.4}$$

In this subsection, the index function, the reduced set, the Knapsack problem and the Multiple Knapsack problem are defined too.

Definition 2. *Let* $\mathcal{S} = \{a_{k_1}, a_{k_2}, \ldots, a_{k_n}\}$ *be a set of elements with indices* $k_1, k_2, \ldots, k_n \in \{i \mid 1 \leq i \leq n\} \wedge \forall i, j : k_i \neq k_j$. *Function* $\varphi(i, \mathcal{S})$ *is the index function such that*

$$\varphi(i, \mathcal{S}) = k_i \quad \forall i = 1, \ldots, n. \tag{6.5}$$

Definition 3. *Let* $\mathcal{A} = \{a_{k_1}, a_{k_2}, \ldots, a_{k_n}\}$ *be a set of elements with indices* $k_1, k_2, \ldots, k_n \in \{i \mid 1 \leq i \leq n\} \wedge \forall i, j : k_i \neq k_j$. *Sets* $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{i-1}$ *are subsets of* $\mathcal{A}$ *and* $\bigcap_{j=1}^{i-1} \mathcal{A}_j = \emptyset$. *The* i*-th reduced set* $\mathcal{R}_i$ *is a set such that*

$$\mathcal{R}_i = \{a_{k_1}, a_{k_2}, \ldots, a_{k_n}\} \setminus \bigcup_{j=1}^{i-1} \mathcal{A}_j. \tag{6.6}$$

Definition 4. *Given* $n \in \mathbb{N}$ *items with weight* $w_1, w_2, \ldots, w_n \in \mathbb{N}$, *value* $v_1, v_2, \ldots, v_n \in \mathbb{R}^+$ *and capacity* $W \in \mathbb{N}$. *The Knapsack problem is an optimization problem such that:*

To maximize

$$\sum_{i=1}^{n} v_i x_i, \tag{6.7}$$

subject to the constraints

- *maximum knapsack capacity*

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{6.8}$$

- *assignment of item* i

$$x_i \in \{0, 1\} \quad \forall i = 1, \ldots, n \tag{6.9}$$

Being a generalization of the well-known Knapsack problem [43], the Multiple Knapsack problem represents an extension to m knapsacks.

Definition 5. *Given* $n \in \mathbb{N}$ *items with weights* $w_1, w_2, \ldots, w_n \in \mathbb{N}$ *and values* $v_1, v_2, \ldots, v_n \in \mathbb{R}^+$, *and* $m \in \mathbb{N}$ *knapsacks with capacities* $W_1, W_2, \ldots, W_m \in \mathbb{N}$. *The Multiple Knapsack problem is an optimization problem such that:*

To maximize

$$\sum_{i=1}^{m} \sum_{j=1}^{n} v_j x_{ij}, \tag{6.10}$$

subject to the constraints

- *maximum knapsack capacity*

$$\sum_{j=1}^{n} w_j x_{ij} \leq W_i \quad \forall i = 1, \ldots, m \tag{6.11}$$

- *each item must be allocated at most once*

$$\sum_{i=1}^{m} x_{ij} \leq 1 \quad \forall j = 1, \ldots, n \tag{6.12}$$

- *assignment of item*

$$x_{ij} \in \{0, 1\} \quad \forall i = 1, \ldots, m, \quad \forall j = 1, \ldots, n \tag{6.13}$$

In order to simplify a description of algorithms solving the LB problem, the LB problem can be presented as $m \in \mathbb{N}$ independent Knapsack problems. Thus, the m–Knapsack problem is defined as follows.

Definition 6. *Given $n \in \mathbb{N}$ items with weight $w_1, w_2, \ldots, w_n \in \mathbb{N}$, value $v_1, v_2, \ldots, v_n \in \mathbb{R}^+$ and capacities $W_1, W_2, \ldots, W_m \in \mathbb{N}$. The m–Knapsack problem is an optimization problem of $m \in \mathbb{N}$ independent Knapsack problems using the index function and the reduced set of items such that for the i-th Knapsack problem:*

To maximize

$$\sum_{j=1}^{|\mathcal{R}_i^{\mathrm{v}}|} v_{\varphi(j, \mathcal{R}_i^{\mathrm{v}})} x_j, \tag{6.14}$$

subject to the constraints

- *maximum i-th knapsack capacity*

$$\sum_{j=1}^{|\mathcal{R}_i^{\mathrm{w}}|} w_{\varphi(j, \mathcal{R}_i^{\mathrm{w}})} x_j \leq W_i \tag{6.15}$$

- *assignment of item j*

$$x_j \in \{0, 1\} \quad \forall j = 1, \ldots, |\mathcal{R}_i^{\mathrm{w}}| \tag{6.16}$$

where

$$\mathcal{R}_i^{\mathrm{w}} = \{w_{k_1}, w_{k_2}, \ldots, w_{k_n}\} \setminus \bigcup_{j=1}^{i-1} \bigcup_{k \in \mathcal{S}_j} w_k, \tag{6.17}$$

$$\mathcal{R}_i^{\mathrm{v}} = \{v_{k_1}, v_{k_2}, \ldots, v_{k_n}\} \setminus \bigcup_{j=1}^{i-1} \bigcup_{k \in \mathcal{S}_j} v_k \tag{6.18}$$

and sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{i-1}$ contain indices of items already assigned in the previous i - 1 Knapsack problems. Apparently, $|\mathcal{R}_i^{\mathrm{w}}| = |\mathcal{R}_i^{\mathrm{v}}|$.

Mentioned definitions are useful in the following subsection concerning the problem complexity and the proof of $\mathcal{NP}$-completeness. Moreover, they are used in an extensive way in a description of methods solving the LB problem.

The LB problem is a completely new optimization problem. In general, it is very common approach trying to find a similar well-known optimization problem in the theory of mathematical optimization at first. Such a well-known optimization problem can provide an inspiration how to solve a new optimization problem or information how a new optimization problem might be complex.

Unfortunately, there is no general procedure how to identify similarities with the different well-known optimization problems. It always depends on experience of a solver how well he/she knows all the well-known optimization problems and their specifications. A solver has to start with the abstraction of a new problem and focus on the key aspects. Then, the similarities might be easier to be identified.

The Multiple Knapsack problem being only an extension of the well-known Knapsack problem to m knapsacks has been defined. It is not a coincidence that this fact is used for a solution of the LB problem.

If a solver sets the capacities of m knapsacks to the same values as the theoretical average flow for one MUX, i.e., $F = \left\lceil \sum_{i=1}^{n} f_i/m \right\rceil$ and uses weights as flows, then both problems might be strongly linked. Apparently, the theoretical average flow F for one MUX must be adjusted for the remaining MUXes which have not been allocated yet based on the remaining not yet allocated flows.

Moreover, the level of abstraction might be even higher. In fact, there are always two disjoint sets – the set of allocated flows to the i-th MUX and the set of not yet allocated flows. In addition, if it is considered that all MUXes are supposed to have the similar total flow, then it is comparable with the well-known Partition problem [29] where the task of deciding is whether a given set $\mathcal{U}$ of positive integers can be partitioned into two subsets $\mathcal{U}_1$ and $\mathcal{U}_2$ such that the sum of the numbers in $\mathcal{U}_1$ equals the sum of the numbers in $\mathcal{U}_2$.

Several variants and generalizations of the Partition problem are already known. There is a problem called the 3-Partition problem [27] which is to partition the set $\mathcal{U}$ into $|\mathcal{U}|/3$ triples each with the same sum.

Moreover, the Multi-Way Partition problem [45] generalizes the optimization version of the Partition problem. In this optimization problem, the goal is to divide a set $n \in \mathbb{N}$ integers into a given number $m \in \mathbb{N}$ of subsets, minimizing the difference between the smallest and the largest subset sums. Except the same number of integers $p \in \mathbb{N}$ in all subsets, the analogy with the LB problem is perceptible.

To conclude, taking into consideration that the proof of the Knapsack problem complexity [43] is based on the Partition problem, the optimization problems might be similar and connected at a certain level of abstraction. Thus, the proof of LB problem complexity is derived from the Multiple Knapsack problem.

6.2 Problem Complexity

Definition 7. *The encoding of an input to a problem is denoted by* $\langle \cdot \rangle$.

For example, the input to Linear Programming (LP)

$$\min\{\mathbf{c}^{\mathrm{T}}\mathbf{x} \mid \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0\} \tag{6.19}$$

can be denoted $\langle \mathbf{A}, \mathbf{b}, \mathbf{c} \rangle$.

Definition 8. *The set of all binary strings is defined as* $\{0,1\}^* = \{0, 1, 00, 01, 10, 11, 000, \dots\}$.

Definition 9. *A decision problem is one such that the expected output is either YES or NO. It is represented by a set* $A \in \{0,1\}^*$ *of exactly those inputs whose outputs are YES.*

LP can be seen as a decision problem. Taking into consideration LP defined above, the decision problem can be formulated

$$LP = \{\langle \mathbf{A}, \mathbf{b}, \mathbf{c}, t \rangle : \text{There is a solution } \mathbf{x} \text{ s. t. } \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0, \text{ and } \mathbf{c}^{\mathrm{T}}\mathbf{x} \leq t\}. \tag{6.20}$$

If an optimal solution is desired to be found, it can begin at $t = -\infty$ and decide whether $\langle \mathbf{A}, \mathbf{b}, \mathbf{c}, t \rangle \in LP$, then an optimal solution t^* can be found as the point where the answer "switches" from YES to NO.

Definition 10. *Algorithm* ϕ *runs in polynomial time if there exists a polynomial* p *such that the number of steps of* ϕ *on input* x *is no more than* $p(|x|)$.

Definition 11. x *is a YES instance of a decision problem* A *if* $x \in A$. x *is a NO instance of a decision problem* A *if* $x \notin A$. *An algorithm* ϕ *decides* A *if* $\phi(x)$ *outputs YES iff* $x \in A$.

Definition 12. $|x|$ *is the length of the string* x *(e.g. the number of bits it takes to represent* x*).*

Definition 13. *If a computational problem is defined as* π, *then the set of polynomial-time decision problems, denoted by* $\mathcal{P}$, *is defined as:*

$$\mathcal{P} = \{\pi : \textit{There is an algorithm to decide } \pi \textit{ in polynomial time}\}. \tag{6.21}$$

Definition 14. *A decision problem is in* $\mathcal{NP}$ *if there exists a verifier* $\psi(\cdot,\cdot)$, *polynomials* p_1, p_2 *such that:*

- *for all* $x \in A$ *there exists* $y \in \{0,1\}^*$ *where* $|y| \leq p_1(|x|)$ *such that* $\psi(x,y)$ *outputs YES.*
- *for all* $x \notin A$ *there exists* $y \in \{0,1\}^*$ *where* $|y| \leq p_1(|x|)$ *such that* $\psi(x,y)$ *outputs NO.*

- *the number of steps of* $\psi(x, y)$ *is no more than* $p_2(|x| + |y|)$.

$\mathcal{NP}$ *means non-deterministic polynomial time.*

Definition 15. *For decision problems* A, B*, a polynomial-time reduction from* A *to* B *is a polynomial time algorithm* ϕ *such that* $\phi(x) \in B$ *iff* $x \in A$*. In other words,* $\phi(x)$ *is a yes instance of* B *iff* x *is a yes instance of* A*. It is written as* $A \leq_{\mathcal{P}} B$*.*

For example, the Knapsack problem is $\mathcal{NP}$. A decision problem can be: is the set $\mathcal{K}$ of items that are chosen a solution with $\sum_{i \in \mathcal{K}} v_i \geq V$? It takes polynomial time to compute $\sum_{i \in \mathcal{S}} w_i$ and $\sum_{i \in \mathcal{S}} v_i$.

Definition 16. B *is a* $\mathcal{NP}$*-complete problem if* $B \in \mathcal{NP}$ *and for all* $A \in \mathcal{NP}, A \leq_{\mathcal{P}} B$*.*

In Figure 6.2, relationships between classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete and $\mathcal{NP}$-hard are shown.

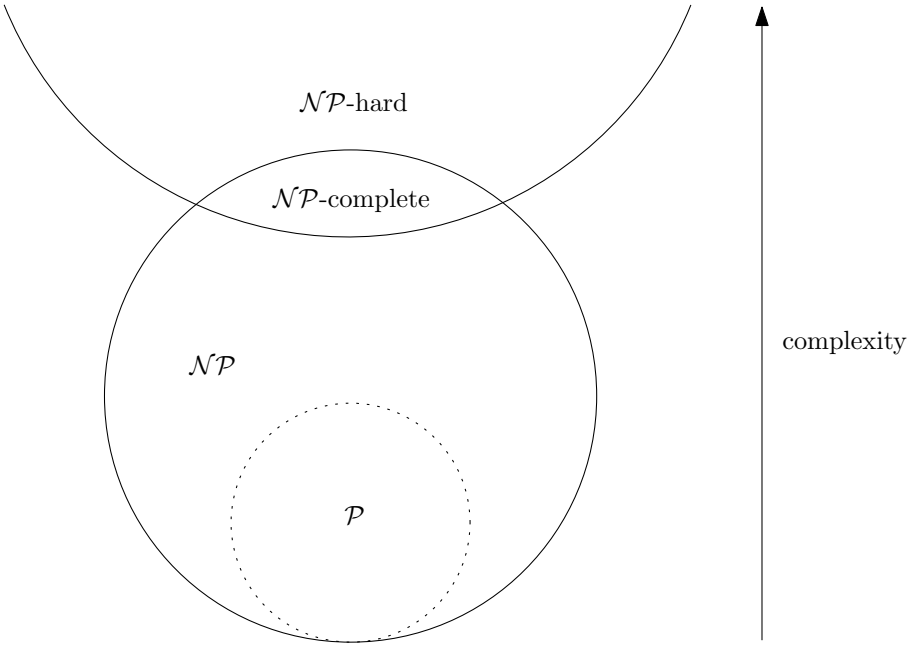

Figure 6.2: Relationships between complexity classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete and $\mathcal{NP}$-hard.

The main complexity classes $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{NP}$-complete have been defined. It remains to discuss the complexity of the LB problem. In the rest of this subsection, it is proven that the LB problem belongs to $\mathcal{NP}$-complete class.

Firstly, the decision version of the Knapsack problem is needed. *Given* $n \in \mathbb{N}$ *items with weight* $w_1, w_2, \ldots, w_n \in \mathbb{N}$*, value* $v_1, v_2, \ldots, v_n \in \mathbb{R}^+$*, capacity* $W \in \mathbb{N}$ *and value* $V \in \mathbb{R}^+$*, is there a subset* $\mathcal{K} \subseteq \{1, \ldots, n\}$ *such that* $\sum_{i \in \mathcal{K}} w_i \leq W$ *and* $\sum_{i \in \mathcal{K}} v_i \geq V$?

Secondly, in order to determine the complexity of the LB problem, the decision version of the LB problem must be defined accordingly. *Given* $m \in \mathbb{N}$ *MUXes*

with $p \in \mathbb{N}$ *ingoing ports each, i.e.,* $n = m \cdot p \in \mathbb{N}$ *ingoing ports in total and flows* $f_1, f_2, \ldots, f_n \in \mathbb{N}_0$. *Are there subsets* $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ *such that* $\bigcup_{i=1}^{m} \mathcal{S}_i = \{i \mid i \in 1, \ldots, n\}$ *and* $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset, \forall i, j = 1, \ldots, m \wedge i \neq j$ *and* $|\mathcal{S}_i| = p, \forall i = 1, \ldots, m$ *and* $\sum_{j \in \mathcal{S}_i} f_j \leq F, \forall i = 1, \ldots, m$ *where* $F = \left\lceil \sum_{i=1}^{n} f_i / m \right\rceil$?

In the proof, the known complexities of the Knapsack problem and the Multiple Knapsack problem are used. Thus, a discussion of their complexity is required beforehand.

Theorem 1. *The Knapsack problem is* $\mathcal{NP}$*-complete.*

Proof. The proof can be found here [43]. □

Theorem 2. *The Multiple Knapsack problem is* $\mathcal{NP}$*-complete.*

Proof. Being a generalization of the Knapsack problem, the Multiple Knapsack problem is $\mathcal{NP}$-complete. The proof can be found here [43]. □

Theorem 3. *The Load Balancing (LB) problem is* $\mathcal{NP}$*-complete.*

Proof. First, the LB problem is a $\mathcal{NP}$ problem. The proof are the subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ of flow indices that are chosen and the verification process is to compute $|\mathcal{S}_i| = p, \forall i = 1, \ldots, m$ and $\sum_{j \in \mathcal{S}_i} f_j \leq F, \forall i = 1, \ldots, m$ which takes polynomial time in the size of input.

Second, it will be shown there is a polynomial reduction from the Multiple Knapsack problem to the LB problem. It suffices to show that there exists a polynomial time reduction $Q(\cdot)$ such that $Q(x)$ is a YES instance to the LB problem iff x is a YES instance to the Multiple Knapsack problem. Suppose there are given $f_1, f_2, \ldots, f_n$, for the LB problem, consider the following Multiple Knapsack problem: $W_i = F, V_i = p, \forall i = 1, \ldots, m$ and $w_i = f_i, v_i = 1 + f_i/h, \forall i = 1, \ldots, n$ where $h > \sum_{i=1}^{n} f_i$ and $\mathcal{K} = \bigcup_{i=1}^{m} \mathcal{K}_i \subseteq \{1, \ldots, n\}$ and $\mathcal{K}_i \cap \mathcal{K}_j = \emptyset, \forall i, j = 1, \ldots, m \wedge i \neq j$ where $\mathcal{K}_i$ represents indices of items assigned to the i-th knapsack. Here, $Q(\cdot)$ is the process converting the Multiple Knapsack problem to the LB problem. It is clear that this process is polynomial in the input size.

If x is a YES instance for the Multiple Knapsack problem, with the chosen sets $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_m$, let $\mathcal{R} = \{1, \ldots, n\} \setminus \bigcup_{i=1}^{m} \mathcal{K}_i$. It follows that $\sum_{j \in \mathcal{K}_i} w_j = \sum_{j \in \mathcal{K}_i} f_j \leq W_i = F, \forall i = 1, \ldots, m$ and it remains to prove there are p items in each knapsack. It follows that $\sum_{j \in \mathcal{K}_i} v_j = \sum_{j \in \mathcal{K}_i} (1 + f_j/h) \geq V_i = p, \forall i = 1, \ldots, m$ and thus, there must be at

least p items in each knapsack to satisfy the inequality. Moreover, $n = m \cdot p$ implies there must be exactly p items in each knapsack and thus, $\mathcal{R} = \emptyset$. Therefore, the sets $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_m$ correspond to sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$, respectively, and x is a YES instance for the LB problem.

Conversely, if $Q(x)$ is a YES instance for the LB problem, there exists $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ such that $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset, \forall i, j = 1, \ldots, m \wedge i \neq j$ and $\bigcup_{i=1}^{m} \mathcal{S}_i = \{i \mid i \in 1, \ldots, n\}$ and $|\mathcal{S}_i| = p, \forall i = 1, \ldots, m$ and $\sum_{j \in \mathcal{S}_i} f_j \leq F, \forall i = 1, \ldots, m$. Let the Multiple Knapsack problem consist of m knapsacks and let the i-th knapsack contain the items corresponding to indices in $\mathcal{S}_i$, and it follows that $\sum_{j \in \mathcal{K}_i} w_j = \sum_{j \in \mathcal{K}_i} f_j \leq W_i = F, \forall i = 1, \ldots, m$ and $\sum_{j \in \mathcal{K}_i} v_j = \sum_{j \in \mathcal{K}_i} (1 + f_j/h) \geq V_i = p, \forall i = 1, \ldots, m$. Therefore, $Q(x)$ is a YES instance for the Multiple Knapsack problem.

This proves the $\mathcal{NP}$-completeness of the LB problem.

□

While a method for computing the solutions to $\mathcal{NP}$-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter $\mathcal{NP}$-complete problems. $\mathcal{NP}$-complete problems are often addressed by using heuristic methods and approximation algorithms. In the following sections, in order to solve the LB problem, five different approaches are proposed, namely, Dynamic Programming (DP), Greedy Heuristic (GH), Integer Linear Programming (ILP), Genetic Algorithm (GA) and Reinforcement Learning (RL).

6.3 Dynamic Programming

Dynamic Programming (DP) [50] is an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure. In general, DP provides a general framework for analyzing many problem types. Within this framework, a variety of optimization techniques can be employed to solve particular aspects of a more general formulation. A creativity is usually required before it can be recognized that a particular problem can be cast effectively as a dynamic program; and often subtle insights are necessary to restructure the formulation so that it can be solved effectively. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force, etc.

In other words, it solves a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

An easy recipe how to proceed using DP to solve a particular optimization problem is as follows [56]:

1. *Embed* a problem in a family of related problems.
2. *Derive* a relationship between the solutions to these problems.
3. *Solve* this relationship.
4. *Recover* a solution to the original problem from this relationship.

In contrast to LP, there does not exist a standard mathematical formulation of "the" DP problem [37]. Rather, DP is a general type of approach to problem solving, and the particular equations used must be developed to fit each situation. Therefore, a certain degree of ingenuity and insight into the general structure of DP problems is required to recognize when and how a problem can be solved by DP procedures. These abilities can best be developed by an exposure to a wide variety of DP applications and a study of the characteristics that are common to all these situations.

The brief introduction to DP was given. In order to understand the above-mentioned principles, the Fibonacci numbers are discussed in the next subsection and how to find the n-th member of the Fibonacci sequence using DP. It follows with demonstration how to solve the Knapsack problem based on DP. Both phases, namely, calculation and retrieval procedures are explained on a simple example. Finally, the last subsection deals with the LB problem.

6.3.1 Fibonacci Numbers

In this subsection, a simple example how to use DP to obtain the Fibonacci numbers is given. The Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones, i.e.,

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots. \tag{6.22}$$

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2} \tag{6.23}$$

with seed values

$$F_0 = 0, F_1 = 1. \tag{6.24}$$

The goal is finding the n-th member of the Fibonacci sequence. A simple method that is a direct recursive implementation mathematical recurrence relation given above is shown in the following Listing 6.1.

```
#include <QCoreApplication>

int fibonacci(int n)
{
   if (n <= 1)
      return n;

   return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(int argc, char **argv)
{
    QCoreApplication* app = new QApplication(argc, argv);
    int n = 5;
    int result = fibonacci(n);
    return app->exec();
}
```

Listing 6.1: The Fibonacci numbers using recursion.

It can be observed that this implementation does a lot of repeated work, see the following recursion tree in Figure 6.3.

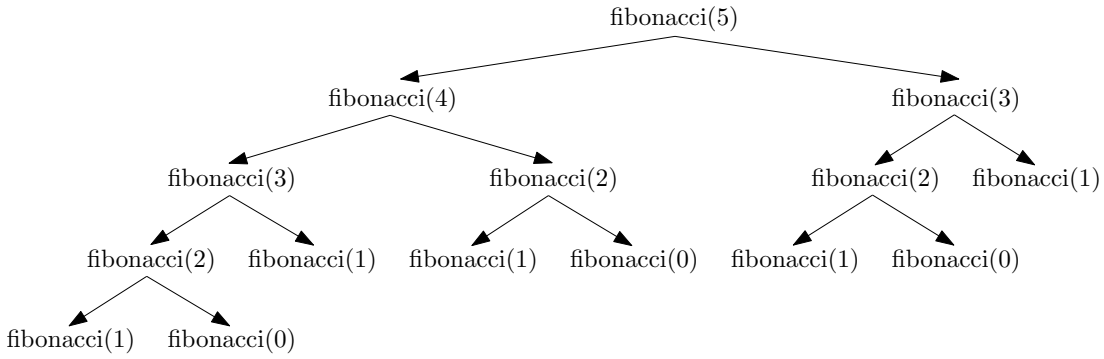

Figure 6.3: The recursion tree for the n-th member of the Fibonacci sequence.

Using recursion is not efficient. Time complexity of such a naive approach is exponential, namely, $O(2^n)$. The repeated work done in the approach using recursion can be avoided by storing the Fibonacci numbers calculated so far. The resulting function requires only $O(n)$ time instead of exponential time, but requires $O(n)$ space. The code using DP is given in the following Listing 6.2.

```
#include <QCoreApplication>

int fibonacci(int n)
{
    int f[n + 1];
    f[0] = 0;
    f[1] = 1;

    for (int i = 2; i <= n; i++)     // use the calculated and
        f[i] = f[i - 1] + f[i - 2]; // stored Fibonacci numbers
```

```
11
12      return f[n];
13  }
14
15  int main(int argc, char **argv)
16  {
17      QCoreApplication* app = new QApplication(argc, argv);
18      int n = 5;
19      int result = fibonacci(5);
20      return app->exec();
21  }
```

Listing 6.2: The Fibonacci numbers using DP.

6.3.2 The Knapsack Problem

Given weights and values of $n \in \mathbb{N}$ items, put these items in a knapsack of capacity $W \in \mathbb{N}$ to get the maximum total value in the knapsack. In other words, given two integer arrays $v_1, v_2, \ldots, v_n \in \mathbb{R}^+$ and $w_1, w_2, \ldots, w_n \in \mathbb{N}$ which represent values and weights associated with n items respectively. Also given an integer W which represents a knapsack capacity. The goal is to find out the maximum value subset of $v_1, v_2, \ldots, v_n$ such that sum of the weights $w_1, w_2, \ldots, w_n$ of this subset is smaller than or equal to W.

Firstly, $m[i, w]$ can be defined to be the maximum value that can be attained with a weight less than or equal to w using items up to i (first i items) where $w \in \{0, 1, \ldots, \sum_{i=1}^{n} w_i\}$ and $i \in \{0, 1, 2, \ldots, n\}$.

In general, $m[i, w]$ can be defined recursively as follows [56]:

- initialization

$$\begin{aligned} m[0, w] &= 0 \quad \forall w \in 0, 1, \ldots, \sum_{i=1}^{n} w_i \\ m[i, 0] &= 0 \quad \forall i \in 0, 1, \ldots, n \end{aligned} \tag{6.25}$$

- recursive step

$$m[i, w] = \begin{cases} m[i-1, w] & \text{if} \quad w_i > w \\ \max(m[i-1, w], v_i + m[i-1, w-w_i]) & \text{if} \quad w_i \leq w \end{cases} \tag{6.26}$$

The solution can then be found by calculating $m[n, W]$. In order to do this efficiently, a table to store the previous computations can be used.

In order to demonstrate and describe algorithms, a simple example is given. It considers a knapsack with capacity $W = 13$ and 8 items with weights $w_1 = 8, w_2 = 8, w_3 = 3, w_4 = 7, w_5 = 7, w_6 = 2, w_7 = 2, w_8 = 5$ and values $v_1 = 8, v_2 = 8, v_3 = 3, v_4 = 7, v_5 = 7, v_6 = 2, v_7 = 2, v_8 = 5$. The following code in Listing 6.3 computes and stores values in a table.

```
static const int n = 8;
static const int W = 13;

int weights[n] = { 8, 8, 3, 7, 7, 2, 2, 5 };
int values[n] = { 8, 8, 3, 7, 7, 2, 2, 5 };

int m[n + 1][W + 1];

// m(i, w) means the value of the best knapsack
// with capacity w using the first i items
for (int i = 0; i <= n; i++)
{
    for (int w = 0; w <= W; w++)
    {
        if (i == 0 || w == 0)
            m[i][w] = 0;
        else if (weights[i - 1] > w)
            m[i][w] = m[i - 1][w];
        else
            m[i][w] = max(m[i - 1][w],
                values[i - 1] + m[i - 1][w - weights[i - 1]]);
    }
}
```

Listing 6.3: The calculation of values and storing them in a table for the Knapsack problem using DP.

Time complexity of algorithm based on DP is $O(nW)$ where $n \in \mathbb{N}$ is the number of items and $W \in \mathbb{N}$ is the capacity of knapsack.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	8	8	8	8	8	8
2	0	0	0	0	0	0	0	0	8	8	8	8	8	8
3	0	0	0	3	3	3	3	3	8	8	8	11	11	11
4	0	0	0	3	3	3	3	7	8	8	10	11	11	11
5	0	0	0	3	3	3	3	7	8	8	10	11	11	11
6	0	0	2	3	3	5	5	7	8	9	10	11	12	13
7	0	0	2	3	4	5	5	7	8	9	10	11	12	13
8	0	0	2	3	4	5	5	7	8	9	10	11	12	13

Table 6.1: The values stored in a table for the Knapsack problem using DP.

The values stored in a table are shown in Table 6.1. The total value of items being in the knapsack is located in the bottom-right corner. Based on the table, it is possible to retrieve which items are assigned to the knapsack and which are not.

The last question is how to obtain items based on the stored table being in the knapsack. Using backtrace, the retrieval procedure starts in the bottom-right corner. There is 13 in row 8 and column 13. It continues up and looks for the last row with value of 13 in the same column. It finds row 6 as the last row with value of 13.

That means, the first item in the knapsack is the item 6. It decreases the capacity of knapsack by $w_6 = 2$, i.e., $w = W - w_6 = 11$.

Then, it moves to the cell in row 6 and column 11 with value of 11. It looks for the last row with value of 11 in the same column again. It finds row 3 as the last row with value of 11. Thus, the item 3 belongs to the knapsack. The capacity of knapsack is decreased by $w_3 = 3$ and results in the remaining capacity equal to 8.

Finally, it moves to the cell in row 3 and column 8 with value of 8. The last row with value of 8 in the same column is row 1. The item 1 belongs to the knapsack too. The weight of item 1 $w_1 = 8$ is equal to the remaining capacity of the knapsack, i.e., there is no free space in the knapsack, the algorithm terminates.

To sum it up, the knapsack contains the items 1, 3, 6 with weights $w_1 = 8, w_3 = 3, w_6 = 2$ and has the total weight of 13 being equal to the initial knapsack capacity $W = 13$. The following code in Listing 6.4 summarizes the retrieval procedure.

```
int best = m[n][W];

// true/false if the i-th item is/is not in the knapsack
int solution[n];
for (int i = 0; i < n; i++)
    solution[i] = 0;

// backtrace
int a = best;
int i = n;
int w = W;

while (a > 0)
{
    while (m[i][w] == a)
        i = i - 1;

    solution[i] = 1; // this item has to be in the knapsack
    w = w - weights[i];
    a = m[i][w];
}
```

Listing 6.4: The retrieval procedure based on values in the stored table for the Knapsack problem using DP.

6.3.3 The Load Balancing Problem

Given $m \in \mathbb{N}$ MUXes with $p \in \mathbb{N}$ ingoing ports each, i.e., $n = m \cdot p \in \mathbb{N}$ ingoing ports in total and flows $f_1, f_2, \ldots, f_n \in \mathbb{N}_0$. The goal is to find out the best allocation of n flows to m MUXes based on the LB problem, see Definition 1.

The idea how to solve the LB problem using DP is based on the DP approach for the Knapsack problem [66]. The first approximation is taken from the fact that the LB problem is very similar to the m–Knapsack problem, see Definition 6. If

these two problems were the same, the algorithm for the Knapsack problem using DP would only be used m-times considering a reduced set of flows $\mathcal{R}_i = \{f_j \mid j \in \{1,\ldots,n\} \setminus \bigcup_{k=1}^{i-1} \mathcal{S}_k\}$ for the i-th MUX where a subset $\mathcal{S}_i$ encapsulates flow indices of the i-th MUX. Finally, it would obtain the solution for the LB problem.

Unfortunately, these two problems are only similar and not the same. The constraint considering p ingoing ports for each MUX makes the difference. Using the m–Knapsack problem terminology, the m–Knapsack problem has no constraint or restriction dealing with exactly p items in the i-th knapsack. That means, the i-th knapsack could have consisted of either less, equal or more items than p.

For the i-th knapsack, a stored table is built where $m_i[j,w]$ contains an optimal value when the overall capacity of the i-th knapsack equals w and only the first j items are considered. If W_i is the overall capacity of the i-th knapsack and there are in total $|\mathcal{R}_i^{\mathrm{w}}|$ items, an optimal solution is given by $m_i[|\mathcal{R}_i^{\mathrm{w}}|, W_i]$. Using backtrace, the items being in the i-th knapsack can be retrieved.

However, considering the exact number of items p in the i-th knapsack, the Knapsack problem algorithm based on DP can not be used. The reason is that the recurrence formula in Equation 6.26 does not take into account different combinations of items [66]:

- Firstly, if the following condition is satisfied

$$m_i[j-1,w] < m_i[j-1, w - w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})}] + v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})}, \tag{6.27}$$

  then

$$m_i[j,w] = m_i[j-1, w - w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})}] + v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})} \tag{6.28}$$

  so that the j-th item is inserted into the i-th knapsack and the exact number of items p is not considered at all. It might lead to a violation of the constraint. On the other hand, the recurrence formula in Equation 6.26 could be improved by keeping track of the number of items inserted at each step and not adding others if the number of items inserted into the i-th knapsack exceeds p. However, a new problem would rise up, see the next point.

- Secondly, assuming the recurrence formula in Equation 6.26 keeps track of the number of items inserted at each step. If the following condition is satisfied

$$m_i[j-1,w] > m_i[j-1, w - w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})}] + v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})}, \tag{6.29}$$

  then

$$m_i[j,w] = m_i[j-1,w] \tag{6.30}$$

  so that the j-th item is not inserted into the i-th knapsack. It might be a mistake in case an optimal solution $m_i[j-1,w]$ already consists of the exact number of items p to be inserted into the i-th knapsack. The source of a problem is that the comparison is not done in a proper way. On one hand, to preserve an optimal solution consisting of p items selected among the previous $j-1$ items, on the other hand, to insert the j-th item and, additionally, consider the best subset with $p-1$ items among the previous $j-1$ items.

Moreover, such a strict constraint is hard to fulfil. Therefore, two mechanisms are introduced ensuring the exact number of items p in the i-th knapsack. Firstly, the mechanism for the upper bound – a maximum number of items p in the i-th knapsack – is proposed. Secondly, the mechanism for the lower bound – a minimum number of items p in the i-th knapsack – is presented. Finally, both mechanisms together form the constraint on the exact number of items p in the i-th knapsack.

Thus, the mechanism for the upper bound consists of adding the third dimension, i.e., $m_i[j, w, k]$ is an optimal solution when the capacity of the i-th knapsack is w, only the first j items are considered and it is not allowed to insert more than k items into the i-th knapsack [66].

In general, $m_i[j, w, k]$ can be defined recursively as follows:

- initialization

$$\begin{aligned} &m_i[0, w, k] = 0 \quad \forall w \in 0, 1, \ldots, W_i, \quad \forall k \in 0, 1, \ldots, p \\ &m_i[j, 0, k] = 0 \quad \forall j \in 0, 1, \ldots, n, \quad \forall k \in 0, 1, \ldots, p \\ &m_i[j, w, 0] = 0 \quad \forall j \in 0, 1, \ldots, n, \quad \forall w \in 0, 1, \ldots, W_i \end{aligned} \tag{6.31}$$

- recursive step

$$m_i[j, w, k] = \begin{cases} m_i[j-1, w, k] \\ \quad \text{if} \quad w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})} > w \\ \max(m_i[j-1, w, k], v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})} + m_i[j-1, w - w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})}, k]) \\ \quad \text{if} \quad w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})} \leq w \wedge j \leq k \\ \max(m_i[j-1, w, k], v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})} + m_i[j-1, w - w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})}, k-1]) \\ \quad \text{if} \quad w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})} \leq w \wedge j > k \end{cases} \tag{6.32}$$

It remains to discuss the lower bound mechanism [66]. Actually, it is based on values $v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})}, \forall j = 1, \ldots, |\mathcal{R}_i^{\mathrm{v}}|$ for the i-th knapsack. The main idea relies on an efficient and clever form of the values. On one hand, it should motivate the algorithm to add as much items as p into the i-th knapsack, on the other hand, it should still keep in mind that a higher flow deserves better reward. For this reason, values are introduced in the following form

$$v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})} = 1 + f_{\varphi(j,\mathcal{R}_i)}/h \quad \forall j \in 1, \ldots, |\mathcal{R}_i^{\mathrm{v}}| \tag{6.33}$$

where $h > \sum_{k=1}^{n} f_k$. The reward "1" is gained by inserting of one item into the i-th knapsack and the fraction $f_{\varphi(j,\mathcal{R}_i)}/h$ assures items with a higher flow to be rather inserted into the i-th knapsack. Moreover, the fraction $f_{\varphi(j,\mathcal{R}_i)}/h$ ensures that a reward for allocation of a single flow is always higher than a reward gained only based on the sum of size of all flows themselves. Such a form of values has already been used in the proof of the LB problem complexity, see Theorem 3.

In order to finalize the problem transformation from the LB problem to the m–Knapsack problem and vice versa, it is necessary to use weights $w_{\varphi(j,\mathcal{R}_i^{\mathrm{w}})} =$

$f_{\varphi(j,\mathcal{R}_i)}, \forall j = 1,\ldots,|\mathcal{R}_i^{\mathrm{w}}|$, values $v_{\varphi(j,\mathcal{R}_i^{\mathrm{v}})} = 1 + f_{\varphi(j,\mathcal{R}_i)}/h, \forall j = 1,\ldots,|\mathcal{R}_i^{\mathrm{v}}|$ and capacity $W_i = \left\lceil \sum_{j=1}^{(m-i+1)p} f_{\varphi(j,\mathcal{R}_i)}/(m-i+1) \right\rceil$ for the i-th knapsack. In addition, the constraint on the exact number of items p in the i-th knapsack results in $|\mathcal{R}_i^{\mathrm{v}}| = |\mathcal{R}_i^{\mathrm{w}}| = |\mathcal{R}_i| = (m-i+1)p$.

The following code in Listing 6.5 computes and stores values in a three-dimensional table for the i-th MUX of the LB problem [66].

```
double m_i[flowCount + 1][F + 1][portCount + 1];

// m_i(j, f, k) means the value of the best i-th knapsack
// with capacity f using the first j items and
// considering maximum number of items k
for (int j = 0; j <= flowCount; j++)
{
    for (int f = 0; f <= F; f++)
    {
        for (int k = 0; k <= portCount; k++)
        {
            if (j == 0 || f == 0 || k == 0)
                m_i[j][f][k] = 0;
            else if (flows[j - 1] > f)
                m_i[j][f][k] = m_i[j - 1][f][k];
            else
            {
                if (j <= k)
                    m_i[j][f][k] =
                        max(m_i[j - 1][f][k], values[j - 1] +
                            m_i[j - 1][f - flows[j - 1]][k]);
                else
                    m_i[j][f][k] =
                        max(m_i[j - 1][f][k], values[j - 1] +
                            m_i[j - 1][f - flows[j - 1]][k - 1]);
            }
        }
    }
}
```

Listing 6.5: The calculation of values and storing them in a table for the i-th MUX of the LB problem using DP.

Time complexity of the LB problem algorithm based on DP is $O((m-i+1)pW_ip)$ where $(m-i+1)p \in \mathbb{N}$ is the number of items, $W_i \in \mathbb{N}$ is the capacity of the i-th knapsack and p is the exact number of items in the i-th knapsack. A solution can be found by calculating $m_i[|\mathcal{R}_i^{\mathrm{w}}|, W_i, p]$. Using backtrace in the stored table, the allocated items can then be retrieved. The following code in Listing 6.6 summarizes the retrieval procedure for the i-th MUX of the LB problem [66].

```
double best = m_i[flowCount][F][portCount];

// true/false if the j-th item is/is not in the i-th knapsack
int solution[flowCount];
```

```
for (int j = 0; j < flowCount; j++)
    solution[i] = 0;

// backtrace
double a = best;
int j = flowCount;
int f = F;
int k = portCount;

while (a > 0)
{
    while (m_i[j][f][k] == a)
        j = j - 1;

    solution[j] = 1; // this item has to be in the i-th knapsack
    f = f - flows[j];
    k = k - 1;
    a = m_i[j][f][k];
}
```

Listing 6.6: The retrieval procedure based on values in the stored table for the i-th MUX of the LB problem using DP.

The algorithms for both phases – computing the stored table and the retrieval procedure – have been discussed. Both algorithms are written for the LB problem of the i-th MUX in a general way. In order to combine the whole LB problem together, a proposal of an algorithm considering all m MUXes is required. The idea is to compute and retrieve $\mathcal{S}_1$ for the first MUX, then reduce a set of flows $\mathcal{R}_1$ by the allocated flows and get a set of flows $\mathcal{R}_2$ for the second MUX and so forth. Generally, the stored table is computed for the i-th MUX, $\mathcal{S}_i$ is retrieved and $\mathcal{R}_i$ is reduced by $\mathcal{S}_i$ giving $\mathcal{R}_{i+1}$ for the $(i+1)$-th MUX. Regarding the last MUX, the m-th MUX, a set $\mathcal{R}_m$ determines directly $\mathcal{S}_m$. The proposed algorithm [66] is stated in Algorithm 6.7.

Algorithm 6.7 The complete LB problem algorithm considering m MUXes using DP

1: load flows $\mathcal{R}_1$
2: **for** $i = 1 \rightarrow m-1$ **do** ▷ get LB of the i-th MUX
3: $\quad F = \left\lceil \sum_{j=1}^{(m-i+1)p} f_{\varphi(j,\mathcal{R}_i)}/(m-i+1) \right\rceil$
4: $\quad m_i = \text{computeTheStoredTable}(\mathcal{R}_i, F)$ ▷ based on Listing 6.5
5: $\quad \mathcal{S}_i = \text{retrievalPhase}(\mathcal{R}_i, F, m_i)$ ▷ based on Listing 6.6
6: $\quad \mathcal{R}_{i+1} = \{f_j \mid j \in \{1,\ldots,n\} \setminus \bigcup_{k=1}^{i} \mathcal{S}_k\}$
7: **end for**
8: $\mathcal{S}_m = \{\varphi(j,\mathcal{R}_m) \mid \forall j = 1,\ldots,p\}$

6.4 Greedy Heuristic

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using DP to determine the best choices is overkill. Therefore, simpler and more efficient algorithms sometimes offer an interesting alternative. A greedy algorithm [20] always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will eventually lead to a globally optimal solution. However, generally, greedy algorithms do not provide globally optimized solutions.

In other words, greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. An optimization problem can be solved using a greedy strategy if the problem has the following property: *At every step, a choice can be made that looks best at the moment, and the optimal solution of the complete problem is obtained.*

In many problems, a greedy strategy does not in general produce an optimal solution, however, a Greedy Heuristic (GH) may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

At this point, a discussion about some of the general properties of greedy methods is started. In general, greedy algorithms have five components [20]:

1. A candidate set from which a solution is created.
2. A selection function which chooses the best candidate to be added to the solution.
3. A feasibility function that is used to determine if a candidate can be used to contribute to a solution.
4. An objective function which assigns a value to a solution, or a partial solution.
5. A solution function which will indicate when a complete solution has been discovered.

Moreover, a greedy algorithm development goes through the following steps [20]:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if the greedy choice is made, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

The key question being still remaining not answered is how a solver can tell whether a greedy algorithm will solve a particular optimization problem. No way works all the time, but the *greedy-choice property* and *optimal substructure* are the two key ingredients [20]. If it can be demonstrated that the problem has these properties, then it is well on the way to developing a greedy algorithm for that problem.

The first key ingredient is the *greedy-choice property* [20]. A globally optimal solution can be assembled by making locally optimal (greedy) choices. In other words, when it is considering which choice to make, it makes the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from DP. In DP, a choice is made at each step, but the choice usually depends on the solutions to subproblems. Consequently, DP problems are solved typically in a bottom-up manner, progressing from smaller subproblems to larger subproblems. In a greedy algorithm, whatever choice seeming best at the moment is made and then, it solves the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it can not depend on any future choices or on the solutions to subproblems. Thus, unlike DP, solving the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. A DP algorithm proceeds bottom up whereas a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

In other words, a greedy algorithm never reconsiders its choices. This is the main difference from DP which is exhaustive and is guaranteed to find the solution. After every stage, DP makes a decision based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to a solution.

The second key aspect is the *optimal substructure* [20]. A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of DP as well as greedy algorithms.

A more direct approach regarding optimal substructure is usually used when applying it to greedy algorithms. Thus, there is the luxury of assuming that it arrived at a subproblem by having made the greedy choice in the original problem. All it really needs to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

To conclude, greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. Therefore, greedy algorithms can be characterized as being "shortsighted", and also as "non-recoverable". They are ideal only for problems which have *optimal substructure*. Despite this, for many simple problems, the best suited algorithms are greedy algorithms.

6.4.1 The Partition Problem

Given a set of $n \in \mathbb{N}$ positive integers, a task is to separate them into two subsets. It may put as many or as few numbers as it pleases in each of the subsets, but it must make the sums of the subsets as nearly equal as possible. Ideally, the two sums would be exactly the same, but this is feasible only if the sum of the entire set is even. In the event of an odd total, the best you can possibly do is to choose two subsets that differ by 1. Accordingly, a perfect partition is defined as any arrangement for which the "discrepancy" – the absolute value of the subset difference – is no greater than 1.

```
#include <QCoreApplication>
#include <list>

using namespace std;

int main(int argc, char **argv)
{
    QCoreApplication* app = new QApplication(argc, argv);

    list<int> U = { 4, 6, 7, 5, 8 };
    list<int> U1;
    list<int> U2;
    int sumU1 = 0;
    int sumU2 = 0;

    U.sort();
    U.reverse();

    for (list<int>::iterator it = U.begin(); it != U.end(); it++)
    {
        if (sumU1 <= sumU2)
        {
            U1.push_back(*it);
            sumU1 += *it;
        }
        else
        {
            U2.push_back(*it);
            sumU2 += *it;
        }
    }

    return app->exec();
}
```

Listing 6.8: The greedy algorithm for the Partition problem.

More formally, the Partition problem is the task of deciding whether a given set $\mathcal{U}$ of positive integers can be partitioned into two subsets $\mathcal{U}_1$ and $\mathcal{U}_2$ such that the sum of the numbers in $\mathcal{U}_1$ equals the sum of the numbers in $\mathcal{U}_2$.

The GH algorithm for the Partition Problem starts with a sorting of the numbers

in descending order, and then assigns each number in turn to the subset with the smaller sum so far [45]. For example, given the numbers $\mathcal{U} = \{8,7,6,5,4\}$, it would assign the 8 and 7 to different subsets, the 6 to the subset with the 7, the 5 to the subset with the 8, and finally the 4 to either subset, yielding for example the partition $\mathcal{U}_1 = \{8,5,4\}$ and $\mathcal{U}_2 = \{7,6\}$, with a subset difference of 17 - 13 = 4. The average solution quality of this heuristic is based on the order of the smallest number.

However, if $\mathcal{U}$ is divided into the subsets $\mathcal{U}_1 = \{7,8\}$ and $\mathcal{U}_2 = \{4,5,6\}$, the sum of the numbers in each subset is 15, and the difference between the subset sums is zero which is optimal.

Time complexity of the Partition problem algorithm based on GH is $O(n \log n)$. This heuristic works well in practice, however, as it can be seen in the particular example, generally, it is not guaranteed to produce the best possible partition. To conclude, the above-mentioned code in Listing 6.8 summarizes the greedy algorithm for the Partition problem.

6.4.2 The Load Balancing Problem

The LB problem can be seen as a generalization of the Partition problem with m subsets [66] and such a problem is called the Multi-Way Partition problem. However, such a generalization is not enough. In general, the goal is to divide a set of $n \in \mathbb{N}$ integers into a given number $m \in \mathbb{N}$ of subsets, minimizing the difference between the smallest and the largest subset sums.

Unfortunately, the Multi-Way Partition problem does not keep in mind the constraint on the same number of integers $p \in \mathbb{N}$ in all subsets. In other words, the Multi-Way Partition problem would be equal to the LB problem if and only if the Multi-Way Partition problem also held the same number of integers p in all subsets.

Therefore, the code for the Partition problem can be generalized to m subsets and adjusted so that it does not consider "full" subsets (p integers have been already allocated to the subset) anymore, see Listing 6.9.

Due to a temporary assignment of `INT_MAX` to subset sum for all "full" subsets, the algorithm does not consider "full" subsets for an allocation anymore. The remaining "not-full" subsets have smaller subset sum each. Thus, a subset with the smallest sum is selected for the allocation of the integer.

The algorithm proceeds in this way and in the last step, there remains only one subset for the allocation of the last integer. The other subsets are "full" subsets with p integers each. Finally, all subsets have the same number of integers p at the end of the algorithm.

```
1 static const int MUXCount = 6;
2 static const int MUXPortCount = 15;
3 static const int portCount = MUXCount * MUXPortCount;
4
```

```
5  static const int maxPortSize = 10000;
6  static const int minPortSize = 1;
7
8  int f[portCount];
9
10 // load flows in the range from 1 B to 10 kB
11 loadFlows(f, portCount, minPortSize, maxPortSize)
12
13 int MUXes[MUXCount][MUXPortCount];
14 int flowSum[MUXCount];
15 int flowSumTemp[MUXCount];
16 int flowAssignedCount[MUXCount];
17 for (int i = 0; i < MUXCount; i++)
18 {
19     flowSum[i] = 0;
20     flowSumTemp[i] = 0;
21     flowAssignedCount[i] = 0;
22 }
23
24 sortDescending(f, portCount);
25
26 for (int i = 0; i < portCount; i++)
27 {
28     for (int j = 0; j < MUXCount; j++)
29     {
30         if (flowAssignedCount[j] < MUXPortCount)
31             flowSumTemp[j] = flowSum[j];
32         else
33             flowSumTemp[j] = INT_MAX;
34     }
35
36     // get index of list with smallest sum so far
37     int minSumIndex = getMinSumIndex(flowSumTemp, 0, MUXCount - 1);
38
39     MUXes[minSumIndex][flowAssignedCount[minSumIndex]] = f[i];
40     flowSum[minSumIndex] += f[i];
41     flowAssignedCount[minSumIndex]++;
42 }
```

Listing 6.9: The greedy algorithm for the LB problem.

6.5 Integer Linear Programming

Integer Linear Programming (ILP) [12] is a mathematical optimization dealing with a class of constrained optimization problems in which some or all variables are integers and all mathematical functions in the objective and constraints are conventionally linear. The insertion of integer variables in Linear Programming (LP) [51] enables much more rich and realistic representations of decision situations.

This integrality restriction may seem rather innocuous, but in reality, it has far reaching effects. On one hand, modeling with integer variables has turned out to be useful far beyond restrictions to integral production quantities. With integer

variables, one can model logical requirements, fixed costs, sequencing and scheduling requirements, and many other problem aspects.

The downside of all this power, however, is that problems with as few as 40 variables can be beyond the abilities of even the most sophisticated computers. While these small problems are somewhat artificial, most real problems with more than 100 variables are not possible to solve unless they show specific exploitable structure. Despite the possibility (or even likelihood) of enormous computing times, there are methods that can be applied to solving integer programs.

LP is a method to achieve the best outcome in a mathematical model whose requirements are represented by linear relationships. In other words, LP is a special case of mathematical programming (mathematical optimization). Using LP, it searches a set of values for continuous variables $(x_1, x_2, \ldots, x_n)$ that maximizes or minimizes a linear objective function z, while satisfying a set of linear constraints (a system of simultaneous linear equations and/or inequalities). Mathematically, LP is expressed as follows [51]:

To maximize

$$z = \sum_{j=1}^{n} c_j x_j, \tag{6.34}$$

subject to the constraints

$$\begin{aligned} \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad & \forall i = 1, \ldots, m \\ x_j \geq 0 \quad & \forall j = 1, \ldots, n. \end{aligned} \tag{6.35}$$

The ILP problem is the LP problem in which at least one of the variables is restricted to integer values. In the past two decades, there has been an increasing use of an alternate term – the mixed integer linear programming problem – for LP problems with integer restrictions on some or all of the variables. For clarity, a term the pure integer linear programming problem is sometimes used to emphasize the ILP problem whose variables are all restricted to be integer valued.

The term "programming" in this context means planning activities that consume resources and/or meet requirements, as expressed in the $m \in \mathbb{N}$ constraints. The resources may include raw materials, machines, equipments, facilities, workforce, money, management, information technology, and so forth. In the real world, these resources are usually limited and must be shared with several competing activities. Requirements may be implicitly or explicitly imposed. The objective of the LP/ILP problem is to allocate the shared resources, and responsibility to meet requirements, to all competing activities in an optimal (best possible) manner.

Mathematically, the ILP problem is defined as [12]:

T maximize

$$z = \sum_{j=1}^{n_1} c_j x_j + \sum_{k=1}^{n_2} d_k y_k, \tag{6.36}$$

subject to the constraints

$$\begin{aligned} \sum_{j=1}^{n_1} a_{ij} x_j + \sum_{k=1}^{n_2} e_{ik} y_k \leq b_i \quad & \forall i = 1, \ldots, m \\ x_j \geq 0 \quad & \forall j = 1, \ldots, n_1 \\ y_k \in \mathbb{N}_0 \quad & \forall k = 1, \ldots, n_2. \end{aligned} \tag{6.37}$$

If $n_1 = 0$, then the ILP problem can be called as the pure integer linear programming problem. Then, the ILP problem is basically the LP problem if $n_2 = 0$. Finally, if $n_1 > 0$ and $n_2 > 0$, then the ILP problem can be called as the mixed integer linear programming problem.

However, only pure integer linear programming problems are assumed in the rest of this section, i.e., $n_1 = 0$. Moreover, for the ILP problem, an integer decision variable is denoted as x_i, an integer solution as $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and an integer optimum solution as $\mathbf{x}^*$ in the rest of this section.

This section gives a brief description of two classical approaches for solving ILP problems, namely, branch-and-bound and cutting-plane methods [12]. Although both approaches are capable of solving ILP problems, their degrees of success vary in software implementation. The cutting-plane approach, when used as a stand-alone solver, has potential to solve ILP problems of limited size, but may not work well in a large-scale application.

However, the valid inequality cuts generated by the cutting-plane approach can be useful when combined with the branch-and-bound method to yield a powerful branch-and-cut approach [12].

For over three decades, the branch-and-bound had been the prevailing solution method until the emergence of the branch-and-cut in early 1990s. The branch-and-cut combined the branch-and-bound with the generated cutting planes into a much more efficient "hybrid" approach.

As a whole, extracting the strengths of the cutting-plane method and injecting them into the branch-and-bound may greatly increase the modern solution power for ILP problems. In what follows, it introduces the concepts and background of these two solution approaches, and then exploit the potential strengths of each approach.

In contrast with ILP, all other approaches, like DP, GH, Genetic Algorithm (GA) and Reinforcement Learning (RL), being used in this thesis require own implementation for each particular optimization problem in general. Thus, all of them represent a general type of an approach to problem solving and the particular equations used must be developed to fit each situation.

In other words, there does not exist any standard mathematical formulation of "the" optimization problem for any of them and ILP is an exception. Therefore, it gives an opportunity to use some standard solvers. In this thesis, to find a solution of ILP problems, a standard MATLAB function `intlinprog` [49] and very powerful COIN-OR (Computational Optimization Infrastructure for Operations Research) project [70] written in C++ based both on the branch-and-cut method are used.

To sum it up, generally, the branch-and-bound, cutting-plane and branch-and-cut methods are well-known and widely used. For this reason, a description of these methods is not going into details and it rather focuses on `intlinprog` and COIN-OR project themselves.

Moreover, it takes into account that the simplex method and dual simplex method used for the LP optimization [51] are generally known and it refers to them without any additional explanation.

6.5.1 Branch and Bound Method

The branch-and-bound approach [12] can be viewed as a divide-and-conquer approach to solving the ILP problem in which a branching process represents dividing and a bounding process stands for conquering. Throughout the algorithm, a series of LP subproblems are systematically generated and solved. Then, upper and lower bounds are progressively tightened on the objective value of the original ILP problem.

A typical way to represent such a process is via a branch-and-bound tree which is a specialized enumeration tree for keeping track of how LP subproblems are generated and solved. The branch-and-bound tree by convention is drawn upside down with its root node at the top. The root node that represents the LP problem relaxation of the original ILP problem (in mathematics, the relaxation of the ILP problem is the problem that arises by removing the integrality constraint of each variable) is solved. If the LP problem optimum solution satisfies the integer requirement, the ILP problem is solved. Otherwise, the LP problem objective value becomes the initial upper bound on the ILP problem optimal objective value and the root node is partitioned into two successor nodes (subproblems) by two branches. These branches are valid cuts in terms of simple inequality constraints that have the following properties [12]:

- They cut off the current non-integer LP problem optimum point and other fractional region.
- The two successor nodes are mutually exclusive and their union contains the same integer feasible region as that of their predecessor, i.e., no integer points are eliminated.

The solution of the LP problem relaxation on a node provides information about [12]:

- Whether a further branching from this node is needed (or whether the node can be pruned).
- A better lower bound (for maximization problem) on the objective of the original ILP problem.

There are three cases indicating that a node can be pruned [12]:

1. **Pruned by infeasibility** – The subproblem has no feasible LP problem solution.
2. **Pruned by optimality** – The subproblem has an integer optimum solution.
3. **Pruned by bound** – The upper bound of the subproblem optimum is less than or equal (for maximization problem) to the lower bound of the original problem.

If a node is pruned by optimality, its optimum solution can be used to increase the lower bound on the objective value of the original ILP problem. Whenever an integer solution to a subproblem is obtained, it is a candidate optimum to the original ILP problem. In the solution process of the branch-and-bound, the best integer solution found so far is continuously updated.

6.5.2 Cutting Plane Method

The term cutting plane [12] is often used for an equality or inequality constraint that can cut off a fractional part of the LP problem feasible region, without excluding any integer feasible solution. In the cutting plane approach, one or more such cutting planes are added to the current LP problem simplex tableau which in turn are resolved for a new LP problem optimum. This process is repeated until the prescribed integer requirements are satisfied.

The method is running in the following steps [12]:

Step 1 – the LP problem
Solve the ILP problem as if it were the LP problem. If it is infeasible, so is the LP problem and then stop. Else if an LP problem optimal solution satisfying the integer requirements is found, then the ILP problem is solved. Otherwise, go to Step 2.

Step 2 – a generating row (or source row) selection
Select a row, having the non-integer right side with the highest non-integer violation $b_i - \lfloor b_i \rfloor$ in simplex tableau, to be a generating row (or source row) from the LP problem optimum simplex tableau.

Step 3 – a new Gomory's cut
Derive a cut constraint from the generating row and augment it to the current tableau, resulting in a primal infeasible solution. Given $m \in \mathbb{N}$ rows in current simplex tableau (there are $m \in \mathbb{N}$ basic variables and $k \in \mathbb{N}$ non-basic variables [51]), the i-th row is selected and a new Gomory's cut is added, then a new constraint can be formulated as:

$$x_{m+k+1} + \sum_{j=1}^{m+k} (\lfloor a_{ij} \rfloor - a_{ij}) x_j = \lfloor b_i \rfloor - b_i \tag{6.38}$$

where x_{m+k+1} is a new Gomory's slack variable associated with the cut. For clarity, the result of sum in Gomory's cut does not contain m basic variables

at all, because m basic variables have coefficients a_{ij} either equal to 1 or 0 in current simplex tableau , i.e., $\lfloor 1 \rfloor - 1 = 0$ and $\lfloor 0 \rfloor - 0 = 0$.

Step 4 – the dual simplex method
Apply the dual simplex method to reoptimize the augmented LP problem. If a new LP problem optimum satisfies the integer requirements, the original ILP problem is solved. Otherwise, go to Step 2.

The main difference among various methods of cutting plane is how a cut constraint is generated. The main requirement is that a generated cut constraint must be valid, meaning that its addition will result in cutting off the current LP problem optimal point, but will not eliminate any integer feasible solution. In other words, every valid cut has two properties [12]:

- The current optimal solution to the LP problem relaxation will violate the cut constraints.
- Any feasible point to the corresponding ILP problems will satisfy the cut constraint.

6.5.3 Branch and Cut Method

Conceptually, the branch-and-cut method [12] can be viewed as a generalization of the branch-and-bound method. Basically, it builds upon the same branch-and-bound framework with additional cuts generated and imposed on each node of the branch-and-bound tree, prior to pruning and branching processes.

Although both methods solve a series of LP problem relaxation at various nodes, their solution philosophies are different. The branch-and-bound applies two simple bound cuts at each node and takes advantage of fast reoptimization of the LP problem at each node. The branch-and-cut philosophy is to do as much work as necessary to get a "tight bound" at the node before pruning and branching. The work at each node may include generating strong cuts, improving formulations, problem preprocessing, and applying a primal heuristic. In practice, many cuts may be added at each node which may slow down the reoptimization. For a given large-scale problem, an empirical investigation is usually used to determine the proper number of cuts to be imposed on the root and other nodes.

The branch-and-cut method can be summarized in the following steps [12]:

1. Add the initial ILP problem to $\mathcal{L}$ – the set of active problems.
2. Set $\mathbf{x}^* = \text{null}$ and $z^* = -\infty$.
3. While the $\mathcal{L}$ is not empty.
   (a) Select and remove a problem from $\mathcal{L}$.
   (b) Solve the LP relaxation of the problem.

(c) If the solution is infeasible, then go back to Step 3. Otherwise, denote the solution by $\mathbf{x}$ with the objective value z.

(d) If $z \leq z^*$, then go back to Step 3.

(e) If $\mathbf{x}$ satisfies the integer requirements, then set $\mathbf{x}^* = \mathbf{x}$, $z^* = z$ and go back to Step 3.

(f) If desired, then search for the cutting planes that are violated by $\mathbf{x}$. If any are found, then add them to the LP relaxation of the problem and return to Step 3b.

(g) Branch to partition the problem into new problems with the restricted feasible regions. Add these problems to the set $\mathcal{L}$ and go back to Step 3.

4. Return $\mathbf{x}^*$.

6.5.4 The Load Balancing Problem

This subsection focuses on a standard MATLAB function `intlinprog` and the COIN-OR project written in C++. They both provide an interface for solving of ILP problems based on the branch-and-cut method.

However, it is necessary to define the LB problem in terms of ILP firstly. Given $m \in \mathbb{N}$ MUXes with $p \in \mathbb{N}$ ingoing ports each, i.e., $n = m \cdot p \in \mathbb{N}$ ingoing ports in total and flows $f_1, f_2, \ldots, f_n \in \mathbb{N}_0$. $F = \left\lceil \sum_{i=1}^{n} f_i/m \right\rceil$ is a theoretical average flow for one MUX. Moreover, the rewards $r_j \in \mathbb{R}^+, \forall j \in 1, \ldots, n$ obtained by the allocation of the j-th flow f_j are defined.

To maximize

$$\sum_{i=1}^{m} \sum_{j=1}^{n} r_j x_{ij}, \tag{6.39}$$

subject to the constraints

- MUX flow limit

$$\sum_{j=1}^{n} f_j x_{ij} \leq F \quad \forall i = 1, \ldots, m \tag{6.40}$$

- each MUX has p ports

$$\sum_{j=1}^{n} x_{ij} = p \quad \forall i = 1, \ldots, m \tag{6.41}$$

- each flow must be allocated

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \forall j = 1, \ldots, n \tag{6.42}$$

- allocation of the j-th flow to the i-th MUX

$$x_{ij} \in \{0,1\} \quad \forall i = 1,\ldots,m, \quad \forall j = 1,\ldots,n \tag{6.43}$$

In the LB problem using ILP, the rewards are set to be 1 for each flow allocation and thus, they are not set in any sophisticated way. That means, the optimal objective value z^* can be achieved at the level of $n = m \cdot p$.

Firstly, it is showed how to solve the LB problem in MATLAB using `intlinprog` [49]. Function `intlinprog` is a mixed integer linear programming solver. In contrast with the ILP definition, the `intlinprog` finds the minimum of the ILP problem.

If function `intlinprog` considers a minimization problem, say $\min f(\mathbf{x})$ subject to $\mathbf{x} \in \mathcal{M}$ where $\mathcal{M}$ is a feasible set, then an equivalent maximization problem is $\max -f(\mathbf{x})$ subject to $\mathbf{x} \in \mathcal{M}$. That is, minimizing $-f$ is the same as maximizing f. Any solution to the minimization problem is a solution to the maximization problem and vice versa. For clarity, the value of the maximization problem is -1 times the value of the minimization problem.

Function `intlinprog` uses the following basic strategy in order to solve mixed integer linear problems. It can solve the problem in any of the stages. If it solves the problem in a stage, `intlinprog` does not execute the later stages [49]:

1. Reduce the problem size using the LP problem preprocessing.
2. Solve an initial relaxed (non-integer) problem using LP.
3. Perform mixed integer problem preprocessing to tighten the LP relaxation of the mixed integer problem.
4. Try cut generation to further tighten the LP relaxation of the mixed-integer problem.
5. Try to find integer-feasible solutions using heuristics.
6. Use the branch-and-bound algorithm to search systematically for the optimal solution. This algorithm solves LP problem relaxations with restricted ranges of possible values of the integer variables. It attempts to generate a sequence of updated bounds on the optimal objective function value.

In the following Listing 6.10, the code of algorithm for the LB problem using MATLAB function `intlinprog` based on ILP is given. In order to control the optimization process, it is highly recommended to adjust solve options. In the LB problem, it is required to adjust `LPMaxIter` in order to prevent an early termination and `TolInteger` determining the maximum deviation from integer that a component of the solution $\mathbf{x}$ can have and still be considered an integer.

```
1 MUXCount = 6;
2 MUXPortCount = 15;
3 portCount = MUXCount * MUXPortCount;
```

```
variableCount = portCount * MUXCount;

maxPortSize = 10000;
minPortSize = 1;

f = loadFlows(portCount, minPortSize, maxPortSize);
r = ones(variableCount, 1) .* (-1); % rewards
F = ceil(sum(f)/MUXCount);

A = zeros(MUXCount, variableCount);
for i = 1 : MUXCount
    for j = 1 : portCount
        A(i, ((i - 1) * portCount) + j) = f(j); % MUX flow limit
    end
end

Aeq = zeros(MUXCount + portCount, variableCount);
for i = 1 : MUXCount
    for j = 1 : portCount
        % each MUX has p ports
        Aeq(i, ((i - 1) * portCount) + j) = 1;
    end
end
for i = 1 : portCount
    for j = 1 : MUXCount
        % each flow must be allocated
        Aeq(MUXCount + i, ((j - 1) * portCount) + i) = 1;
    end
end

b = ones(MUXCount, 1) .* F; % MUX flow limit

beq = zeros(MUXCount + portCount, 1);
for i = 1 : MUXCount
    beq(i) = MUXPortCount; % each MUX has p ports
end
for i = 1 : portCount
    beq(MUXCount + i) = 1; % each flow must be allocated
end

intcon = 1 : variableCount; % all variables are integer
lb = zeros(variableCount, 1); % variable lower bounds
ub = ones(variableCount, 1); % variable upper bounds
[x, fval, exitflag, output]
    = intlinprog(r, intcon, A, b, Aeq, beq, lb, ub);
```

Listing 6.10: The algorithm for the LB problem using a standard MATLAB function `intlinprog` based on ILP.

Secondly, a discussion how to solve the LB problem in C++ using the powerful COIN-OR project based on the branch-and-cut method [70] is given. The COIN-OR project is an open source for the operations research community. In 2018, it consists of 53 projects, most of them are written in C++ and the rest in Python or Java. For instance, it includes tools for LP (e.g., CLP), nonlinear programming (e.g., Ipopt), ILP (e.g., CBC, BCP and SYMPHONY) and more.

The COIN-OR project is provided for Windows, Linux and Mac OS X with a deployment guide. For Windows, the solution CoinAll in Visual Studio is prepared containing all COIN-OR projects and after a compilation of the whole solution, it is ready to solve optimization problems.

For Linux, the installation is also very straightforward. It requires to clone the COIN-OR project from repository and follow installation instructions in `INSTALL` file. In brief, there is prepared a script to download dependencies, a configure script a it is finished by the project compilation using `make` command.

Once the optimization environment is ready to be used, a solver implements his/her particular optimization problem and includes the compiled COIN-OR projects. The algorithm for the LB problem using the COIN-OR project is shown in the following Listing 6.11. It uses `OsiClpSolverInterface` to solve LP problem relaxations and `CbcModel` to apply the branch-and-cut method. Moreover, it requires to specify its own model describing the LB problem. The `CelModel` consists of all LB problem constraints and the objective function, both described by a standard COIN-OR class `CelExpression`. In order to require exclusively integer variables, the `CelModel` uses `CelIntVar` for all variables with lower and upper bounds set to 0.0 and 1.0, respectively.

```
static const int MUXCount = 6;
static const int MUXPortCount = 15;
static const int portCount = MUXCount * MUXPortCount;
static const int variableCount = portCount * MUXCount;

static const int maxPortSize = 10000;
static const int minPortSize = 1;

int* f = new int[portCount];

// load flows in the range from 1 B to 10 kB
loadFlows(f, portCount, minPortSize, maxPortSize)
int F = (int)ceil(sum(f) / ((double)MUXCount));

OsiClpSolverInterface *solver = new OsiClpSolverInterface();
CelModel model(*solver);

CelIntVar** x = new CelIntVar*[variableCount]; // variables
for (int i = 0; i < variableCount; i++)
    x[i] = new CelIntVar("x" + i, 0.0, 1.0); // integer (0.0 - 1.0)

CelExpression objective;
for (int i = 0; i < variableCount; i++)
    objective += (*x[i]);

model.setObjective(objective); // objective

for (int i = 0; i < MUXCount; i++)
{
    CelExpression constraint; // MUX flow limit
    for (int j = 0; j < portCount; j++)
        constraint += f[j] * (*x[i * portCount + j]);
```

```
33      model.addConstraint(constraint <= F);
34  }
35
36  for (int i = 0; i < MUXCount; i++)
37  {
38      CelExpression constraint; // each MUX has p ports
39      for (int j = 0; j < portCount; j++)
40          constraint += (*x[i * portCount + j]);
41      model.addConstraint(constraint == MUXPortCount);
42  }
43
44  for (int i = 0; i < portCount; i++)
45  {
46      CelExpression constraint; // each flow must be allocated
47      for (int j = 0; j < MUXCount; j++)
48          constraint += (*x[j * portCount + i]);
49      model.addConstraint(constraint == 1);
50  }
51
52  solver->setObjSense(-1.0); // maximization
53  model.builderToSolver();
54  solver->setLogLevel(0);
55  solver->initialSolve(); // solve initial LP relaxation
56
57  CbcModel cbcModel(*solver);
58  cbcModel.setLogLevel(1);
59  cbcModel.solver()->setHintParam(OsiDoReducePrint, true, OsiHintTry);
60  cbcModel.branchAndBound(); // invoke the branch-and-cut algorithm
```

Listing 6.11: The algorithm for the LB problem using the COIN-OR project based on ILP.

6.6 Genetic Algorithm

Belonging to the Evolutionary Algorithms (EA) class [28], a Genetic Algorithm (GA) [21] is a heuristic technique inspired by the process of natural selection and evolutionary biology. It attempts to simulate evolutionary principles in order to find estimate solutions of complex problems for which any algorithm ensuring an optimum retrieval is not effective, applicable or is not known at all. These algorithms use techniques and strategies simulating processes well-know from nature (biology and genetics) – heredity, mutation, natural selection and crossover – for a selective breeding. Thus, the best solution of a given problem can be found.

GA uses Mendel's theory of genetics and Darwin's theory of natural selection as a theoretical background. Mendel's theory of genetics builds up the fundamental laws of heredity being applied in heredity of characters that are not influenced by gender. The theory of natural selection says that stronger and more powerful individuals have bigger chance to survive in a dynamic and constantly changing environment than weaker individuals. In general, such a stronger individual is usually faster, more intelligent or more powerful than its competitors. Weaker individuals survive in their

environment rather due to their luck than due to their characteristics. Therefore, the weaker individuals participate in a production of next generation in a less significant way than stronger individuals.

The main principle of the GA process is gradual production of stronger generations containing individuals representing different solutions of a given optimization problem. In a optimization process, a new population is created in each generation and each individual in a population represents just one solution of a given problem. As a population evolves, solutions improve.

From the optimization process point of view, a population in first generation consists of random individuals $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{N_\mathrm{p}}$ at the beginning of evolution where $N_\mathrm{p} \in \mathbb{N}$. Their elements fulfil the constraints of a problem. In a transition to a new generation, *fitness* value (value of *fitness* function) is counted for each individual where *fitness* value represents a quality of solution given by a particular individual and it is considered as the objective function f of a optimization problem. Based on *fitness* values, individuals are selected being modified using mutation and crossover and resulting in a production of a new population. This process is repeated iteratively. Thus, quality of solutions in population should improve gradually. An algorithm terminates if the stopping criterion is satisfied, e.g., a threshold of solution quality or the maximum number of iterations.

Algorithm can be summarized in the following steps:

Step 1 – initialization
Random initialization of the entire individuals of population (first generation)

Step 2 – evolution step
Using a particular selection strategy (partially random), several high-quality individuals from current population are selected. To produce a new generation, it generates new individuals using the selected high-quality individuals based on the following operators:

- crossover – swapping of elements between a few individuals
- mutation – random change of element of individual so that it still fulfils all constraints of a optimization problem
- reproduction – selection of individual without any change to a new generation

Evaluation of quality of new individuals.

Step 3 – stopping criterion
If the stopping criterion is not satisfied go to Step 2.

Step 4 – termination
An individual with the highest quality (best evaluation) represents the best solution of a optimization problem.

In this section, it is denoted a solution by individual and vice versa. Its genes (characteristics) are called elements. For each individual, the i-th element represents

the characteristic of the same kind. An individual of size $n \in \mathbb{N}$ (n elements) is denoted as $\mathbf{x} = (x_1, x_2, \ldots, x_n)^{\mathrm{T}}$.

To create a new offspring, crossover is a operator used for a recombination simulating a random change of information contained in parents. An application of crossover operator should result in better individuals, i.e., better chromosomes.

Mutation is a reproductive operator being used with small probability for a random change of elements in a particular individual. In other words, it changes value of some elements in some individuals. Mutation operator is able to return back value of element already lost. Moreover, it prevents from an early convergence and offers possibility of searching in a neighbourhood of current population (it is necessary to understand a neighbourhood as a set of individuals being different from individuals in current population in the minimum number of elements). Thus, mutation operator helps to abandon from a local extreme of the *fitness* function.

The improvement of entire population is performed iteratively. Just created individuals become a new generation and replace the old generation completely. It regards to the simplest strategy in which a former population dies out entirely. This process can be called as a generational change. After a generational change finalization, the whole cycle is repeated until the stopping criterion is satisfied.

The stopping criterion is used for an algorithm termination. For instance, it can be represented by the maximum number of iterations, the finding of satisfactory solution, the negligible change of the best so far found solution in last generations, etc.

6.6.1 Differential Evolution

Differential Evolution (DE) [60, 24] is a stochastic, population-based search strategy developed based on the same principles as GA. However, it differs significantly in mutation step, crossover operator and following selection mechanism. DE differs from GA in that mutation is applied first to generate a trial vector which is then used within the crossover operator to produce one offspring, and mutation step sizes are not sampled from a prior known probability distribution function. In DE, mutation step sizes are influenced by differences between individuals of current population.

Algorithm runs in the following steps:

Step 1 – parameter setup

To determine size of population $N_{\mathrm{p}} \in \mathbb{N}$ (number of individuals in each population), the boundary constraints of decision variables representing each gene of an individual, the mutation factor β, the crossover rate CR and stopping criterion.

Step 2 – initialization of population

To initialize randomly the entire individuals of population within the given upper and lower limits.

Step 3 – evaluation of population

To evaluate *fitness* of each individual according to objective function.

Step 4 – mutation operation

The DE mutation operator produces a trial vector for each individual of current population. This trial vector will then be used by the crossover operator to produce offspring. For each parent $\mathbf{x}_i(s)$ from current population generates the trial vector $\mathbf{u}_i(s)$ as follows:

$$\mathbf{u}_i(s) = \mathbf{x}_{i,1}(s) + \beta(\mathbf{x}_{i,2}(s) - \mathbf{x}_{i,3}(s)) \tag{6.44}$$

where $\mathbf{x}_{i,1}(s)$, $\mathbf{x}_{i,2}(s)$ and $\mathbf{x}_{i,3}(s)$ are randomly selected individuals from current population for the parent $\mathbf{x}_i(s)$. β is the scaling factor, controlling the amplification of the differential variation. Theoretically $\beta \in (0; \infty)$, but it is usually taken from the range $[0.1, 1]$.

Step 5 – crossover (recombination operation)

The DE crossover operator implements a discrete recombination of the trial vector $\mathbf{u}_i(s)$ and the parent vector $\mathbf{x}_i(s)$ to produce offspring $\mathbf{x}'_i(s)$. The crossover is implemented as follows:

$$x'_{i,j}(s) = \begin{cases} u_{i,j}(s) & \text{if} \quad \text{rand}(j) \leq CR \\ x_{i,j}(s) & \text{if} \quad \text{rand}(j) > CR \end{cases} \tag{6.45}$$

where $x_{i,j}(s)$ refers to the j-th element of the vector $\mathbf{x}_i(s)$. Elements $u_{i,j}(s)$ and $x'_{i,j}(s)$ are defined in the same way and refer to j-th element of vectors $\mathbf{x}'_i(s)$ and $\mathbf{u}_i(s)$, respectively. CR is the crossover or recombination rate in the range $[0, 1]$.

Step 6 – selection

To construct the population of the next generation, deterministic selection is used: the offspring replaces the parent if the *fitness* of the offspring is better than its parent; otherwise the parent survives to the next generation. In the case of minimization problems, selection is implemented as follows:

$$\mathbf{x}_i(s+1) = \begin{cases} \mathbf{x}'_i(s) & \text{if} \quad f(\mathbf{x}'_i(s)) < f(\mathbf{x}_i(s)) \\ \mathbf{x}_i(s) & \text{otherwise} \end{cases} \tag{6.46}$$

where $f(\cdot)$ indicates the objective function of DE. This mechanism ensures that the average *fitness* of the population does not deteriorate.

Step 7 – stopping criterion

If the stopping criterion is not satisfied go to Step 3, else return the individual with the best *fitness* as the solution. Here, the maximum number of iterations is selected as the stopping criterion.

6.6.2 Modified Differential Evolution

The Modified Differential Evolution (MDE) [63] is a heuristic algorithm based on GA [21] and has new mutation operation, crossover operator and selection mechanism.

Selection mechanism is inspired by Simulated Annealing (SA) [26, 44]. In this section, an individual is represented by vector with n elements $\mathbf{x} = (x_1, x_2, \ldots, x_n)^{\mathrm{T}}$.

At first, all parts are presented in more detail. Namely, *fitness* function, mutation operator, crossover operation, and selection mechanism are described. Consequently, the whole MDE algorithm is presented.

Objective Function

The objective function of the LB problem is directly used as the *fitness* function for the MDE [63], i.e.,

$$\sqrt{\sum_{i=1}^{m} \left(F - \sum_{j \in \mathcal{S}_i} f_j \right)^2} \tag{6.47}$$

where $f_1, f_2, \ldots, f_n \in \mathbb{N}_0$ are flows, $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ are subsets of flow indices for $m \in \mathbb{N}$ MUXes and $F = \left\lceil \sum_{i=1}^{n} f_i / m \right\rceil$ is a theoretical average flow for one MUX.

Moreover, the objective function restricted to the i-th MUX only is used as the i-th MUX *fitness* function for the MDE [63], i.e.,

$$\left(F - \sum_{j \in \mathcal{S}_i} f_j \right)^2. \tag{6.48}$$

Mutation Operator

The mutation operator [63] produces a trial vector for each individual of the current population. This trial vector will then be used by the crossover operator to produce offspring.

Let the first $(i-1)$ MUXes be already allocated. In general, the mutation operator tries to mutate the i-th MUX only. The trial vector $\mathbf{u}_j(s)$ is created based on a random individual $\mathbf{x}_j^r(s)$ selected from the current population for a parent $\mathbf{x}_j(s)$ for the i-th MUX in iteration s. Firstly, all n elements from the random individual $\mathbf{x}_j^r(s)$ are copied to the trial vector $\mathbf{u}_j(s)$. Actually, the flows represented by elements with indices $1, \ldots, (i-1)p$ have already been allocated to the first $(i-1)$ MUXes. Therefore, the mutation operator does not consider elements with indices $1, \ldots, (i-1)p$ and leaves them as they are copied from the random individual $\mathbf{x}_j^r(s)$. Otherwise, the so far achieved solution would have been damaged or completely lost.

Thus, the mutation operator considers only elements with indices $(i-1)p+1, \ldots, mp$. It performs $\lceil \beta p \rceil$ swaps. In more detail, it randomly selects one element from the elements with indices $(i-1)p+1, \ldots, ip$ and one element from elements with indices $ip+1, \ldots, mp$ in each swap from $\lceil \beta p \rceil$ swaps and swaps them.

To sum it up in a well-arranged way, a detailed diagram of the mutation operator is shown in Figure 6.4.

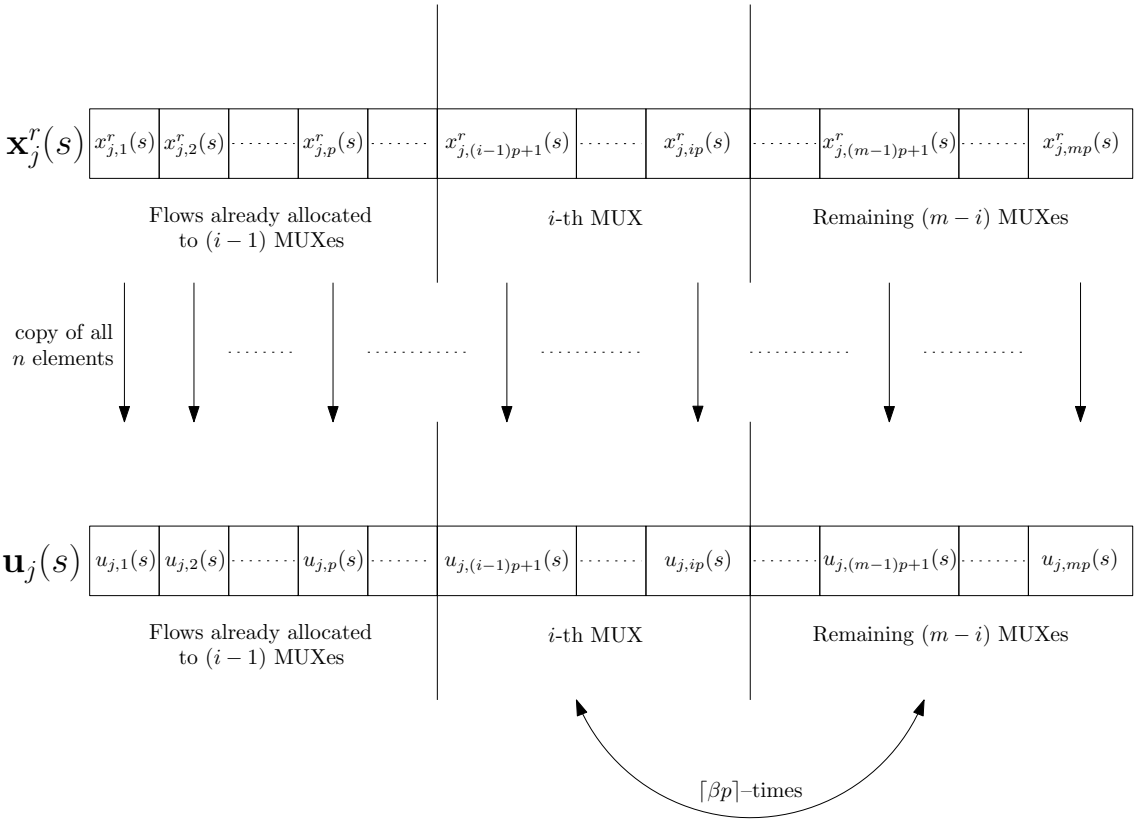

Figure 6.4: The mutation operator used to produce a trial vector $\mathbf{u}_j(s)$ from a random individual $\mathbf{x}_j^r(s)$ selected from current population for a parent $\mathbf{x}_j(s)$ for the i-th MUX in iteration s using $\lceil \beta p \rceil$ swaps.

β is the scaling factor, controlling the amplification of the differential variation. Theoretically $\beta \in (0, \infty)$, but it is usually taken from the range $[0.1, 1]$.

Crossover Operator

The MDE crossover operator [63] implements a discrete recombination of the trial vector $\mathbf{u}_j(s)$ and the parent vector $\mathbf{x}_j(s)$ to produce offspring $\mathbf{x}'_j(s)$. $x_{i,j}(s)$ refers to the j-th element of the vector $\mathbf{x}_i(s)$. Elements $u_{i,j}(s)$ and $x'_{i,j}(s)$ are defined in the same way and refer to the j-th element of vectors $\mathbf{x}_i(s)$ and $\mathbf{u}_i(s)$, respectively. CR is the crossover or recombination rate in the range $[0, 1]$.

The approach is similar to the mutation operator. Let the first $(i-1)$ MUXes be already allocated. In general, the crossover operator tries to cross the i-th MUX only. Firstly, all n elements from the parent $\mathbf{x}_j(s)$ are copied to the offspring $\mathbf{x}'_j(s)$. Actually, flows represented by elements with indices $1, \ldots, (i-1)p$ have already been allocated to the first $(i-1)$ MUXes. Therefore, the crossover operator does not consider elements with indices $1, \ldots, (i-1)p$ and leaves them as they are copied from the parent $\mathbf{x}_j(s)$. The reason is the same as for the mutation operator – the so far achieved solution would be damaged or completely lost.

Thus, the crossover operator considers only the elements with indices $(i-1)p +$

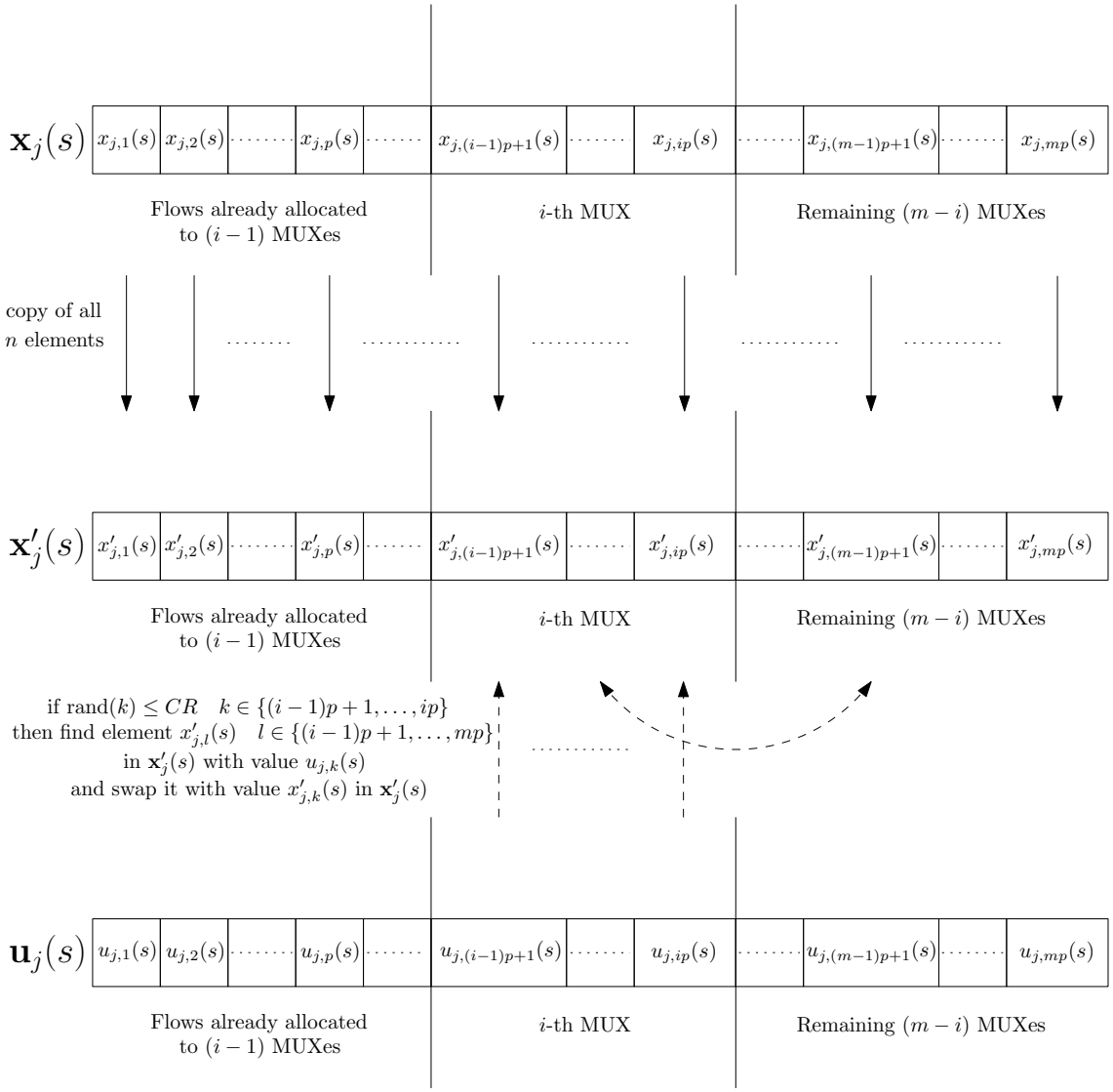

Figure 6.5: The crossover operator used to produce an offspring $\mathbf{x}'_j(s)$ from a parent $\mathbf{x}_j(s)$ and a trial vector $\mathbf{u}_j(s)$ for the i-th MUX in iteration s.

$1,\ldots,mp$. Then, for each element

$$x'_{j,(i-1)p+1}(s),\ldots,x'_{j,ip}(s), \tag{6.49}$$

if

$$\text{rand}(k) \leq CR \quad k \in \{(i-1)p+1,\ldots,ip\}, \tag{6.50}$$

then find element

$$x'_{j,l}(s) \quad l \in \{(i-1)p+1,\ldots,mp\} \tag{6.51}$$

in $\mathbf{x}'_j(s)$ with value $u_{j,k}(s)$ and swap the values of element $x'_{j,k}(s)$ and element $x'_{j,l}(s)$ in $\mathbf{x}'_j(s)$. If the condition in Equation 6.50 is not satisfied, then do nothing.

In Figure 6.5, a diagram of the crossover operator is given. The dashed arrows represent actions being subjected to the condition in Equation 6.50. Once the condition is not satisfied for a given element, it does nothing and continues to the next element.

Adaptive Parameters

The scaling factor β and recombination rate CR affect the exploration and exploitation of the algorithm [63, 22, 53]. Exploration is the algorithm ability to cover and explore different areas in the feasible search space while exploitation is the ability to concentrate only on promising areas in the search space and to enhance the quality of the potential solution in the promising region. The scaling factor β controls the amplification of the differential variations. The smaller the value of β, the smaller the mutation step sizes, and the longer it will take for the algorithm to converge. Larger values of β facilitate exploration, but may cause the algorithm to overshoot good optima. The value of β should be small enough to allow differentials to explore tight valleys, and large enough to maintain diversity. As the population size increases, the scaling factor should be decreased. In this thesis, an adaptive scaling factor is adopted to achieve a favorable compromise between exploration and exploitation. For increasing exploration, a large initial value of β is chosen. Then, it is reduced linearly along the iterations for good exploitation:

$$\beta(s) = c_1 - c_2 \frac{s}{s_{\max}} \tag{6.52}$$

where $s_{\max}$ is the maximum number of iterations and c_1, c_2 are constants.

In this way, the mutation operator performs a wider search in the solution space at the early stages of the evolution and at the later stages, the search is restricted around the local area, resembling a hill-climbing operator [55].

The probability of recombination, CR, has a direct influence on a diversity of the MDE. The higher the probability of recombination, the more variation is introduced in a new population, thereby increasing diversity and exploration. Increasing CR often results in faster convergence, while decreasing CR increases search robustness. In the MDE, an adaptive CR is similarly adopted. CR is changed along the evolution process like β as follows:

$$CR(s) = k_1 - k_2 \frac{s}{s_{\max}} \tag{6.53}$$

where $s_{\max}$ is the maximum number of iterations and k_1, k_2 are constants.

Selection Mechanism

In the MDE, a probabilistic selection mechanism [23] is used instead of the deterministic selection of the original DE [63]. The selection mechanism has been inspired by SA [26, 44]. SA uses a random search strategy which not only accepts new solutions that decrease the objective function value (assuming a minimization problem), but may also accept new solutions that rather increase the objective function value based on a predetermined probability distribution function. Exponential probability distribution function is normally used for this purpose. Based on this idea, the selection mechanism of the MDE can be described as follows:

$$\mathbf{x}_j(s+1) = \begin{cases} \mathbf{x}'_j(s) & \text{if} \quad f(\mathbf{x}'_j(s)) \leq f(\mathbf{x}_j(s)) \\ \mathbf{x}'_j(s) & \text{if} \quad f(\mathbf{x}'_j(s)) > f(\mathbf{x}_j(s)) \wedge h(\mathbf{x}_j(s), \mathbf{x}'_j(s)) > \text{rand}() \\ \mathbf{x}_j(s) & \text{otherwise} \end{cases} \tag{6.54}$$

$$h(\mathbf{x}_j(s), \mathbf{x}'_j(s)) = \exp\left(\frac{f(\mathbf{x}_j(s)) - f(\mathbf{x}'_j(s))}{f(\mathbf{x}_j(s))T}\right) \tag{6.55}$$

where T is the temperature, as defined in the SA technique.

Algorithm 6.12 The MDE algorithm

load flows $\mathcal{R}_1$
set size of population N_p, the maximum iteration s_max, constants $c_1, c_2, k_1, k_2, \alpha$, initial value of the temperature T
initPopulation()
$i = 1$ ▷ current MUX
$F = \left\lceil \sum_{j=1}^{n} f_j/m \right\rceil$
for $s = 1 \rightarrow s_\mathrm{max} \wedge i < m$ **do** ▷ loop for the MDE iterations
    set $\beta = c_1 - c_2 \cdot s/s_\mathrm{max}$, $CR = k_1 - k_2 \cdot s/s_\mathrm{max}$, $T = \alpha \cdot T$
    for $j = 1 \rightarrow N_\mathrm{p}$ **do** ▷ update the population by evolutionary operators
        $xParent$ = getIndividual(j)
        $xRandom$ = getRandomIndividual()
        $uTrial$ = mutation($j, xRandom, k, \beta$)
        $xOffspring$ = crossover($j, uTrial, xParent, CR$)
        $xResult$ = selection($xParent, xOffspring, T$)
        setIndividual($j, xResult$)
    end for
    $bestMUXIndividual$ = getBestMUXIndividual() ▷ based on Equation 6.48
    if fitnessMUX($bestMUXIndividual$) = 0 **then** ▷ based on Equation 6.48
        for $j = 1 \rightarrow N_\mathrm{p}$ **do**
            x = getIndividual(j)
            $x = bestMUXIndividual$ ▷ assign $bestMUXIndividual$
            x = shuffle($x, m - i$) ▷ shuffle elements in the remaining $m - i$ MUXes
            setIndividual(j, x)
        end for
        $\mathcal{S}_i$ = getMUXFlows($i, bestMUXIndividual$)
        $\mathcal{R}_{i+1} = \{f_j \mid j \in \{1, \ldots, n\} \setminus \bigcup_{k=1}^{i} \mathcal{S}_k\}$
        $F = \left\lceil \sum_{j=1}^{(m-i)p} f_{\varphi(j,\mathcal{R}_{i+1})}/(m-i) \right\rceil$
        $i = i + 1$
    end if
end for
$bestIndividual$ = getBestIndividual() ▷ based on Equation 6.47
$error$ = fitness($bestIndividual$) ▷ based on Equation 6.47

Here, the temperature T is adaptively changed in the evolution process as follows:

$$\begin{aligned} T(s+1) &= \alpha T(s) \\ T(0) &= T_0. \end{aligned} \tag{6.56}$$

The parameter α is the rate of reducing the temperature ($\alpha < 1$). T_0 is the initial temperature. A normalized difference between the parent and offspring objective functions has been considered in Equation 6.55 to eliminate the effect of different ranges of objective functions (the adjustment of the temperature T becomes independent from the range of objective function). The selection mechanism begins with a large value for the initial temperature. In other words, at the beginning of the evolution process, many new worse solutions $\mathbf{x}_j(s)$ have chance to be selected to increase the exploration of the MDE. However, by evolving the individuals, the temperature T decreases along the iterations and so the probability of selecting worse solutions is decreased.

The MDE Algorithm

At first, the parameters are set. Then, the initialization of population is executed. To create a new population, the mutation operator, the crossover operator and the selection mechanism, respectively, are applied. The algorithm tries to allocate flows to MUXes gradually, i.e., it always deals with one particular MUX only. Once the allocation procedure for such a particular MUX is finished, it moves to the next MUX and tries to allocate flows from remaining flows to this MUX. If the stopping criterion is not satisfied go to produce a new population by evolutionary operators, else return the individual with the best *fitness* as the solution. Here, the maximum number of iterations $s_{\max}$ is selected as the stopping criterion. The MDE algorithm [63] is described in Algorithm 6.12.

6.7 Reinforcement Learning

Reinforcement Learning (RL) is a study of how animals and artificial systems can learn to optimize their behaviour in the face of rewards and punishments [68, 69, 47]. One way in which animals acquire complex behaviours is by learning to obtain rewards and to avoid punishments. Learning of a baby to walk, a child acquiring the lesson of riding bicycle, an animal learning to trap his food, etc., are some examples. During this learning process, the agent interacts with the environment. At each step of interaction, on observing or feeling the current state, an action is taken by the learner (agent). Depending on the goodness of the action at the particular situation, it is tried at the next stage when the same or similar situation arises [54, 10, 11].

The learning methodologies developed for such learning tasks originally combine two disciplines: Dynamic Programming (DP) and Function Approximation. DP is a field of mathematics that has been traditionally used to solve a variety of optimization

problems [54, 10, 36, 11]. However, DP in its pure form is limited in size and complexity of the problems it can address. Function Approximation methods like Neural Networks learn the system by different sets of input–output pairs to train the network. In RL, the goal to be achieved is known and the system learns how to achieve the goal by trial and error interactions with the environment.

In the conventional RL framework, the agent does not initially know what effects its actions have on the state of the environment and also what the immediate reward he will get on selecting an action. It particularly does not know what action is best to do. Rather, it tries out the various actions at various states, gradually learns which one is the best at each state so as to maximize its long term reward. Thus, the agent tries to acquire a control policy or a rule for choosing an action according to the observed current state of the environment. One of the most natural ways to acquire the above mentioned control rule would be the agent to visit each and every state in the environment and try out the various possible actions. At each state it observes the effect of the actions in terms of rewards. From the observed rewards, the best action at each state or the best policy is manipulated. However, this is not practically possible, since planning ahead involves accurate enumeration of possible actions and rewards at various states which is computationally very expensive.

The concept of the RL problem and action selection is explained with the N–arm Bandit problem in the next subsection. Then, the multi-stage decision making tasks are explained. The Grid World problem is a multi-stage decision making problem.

6.7.1 Single-Stage Decision Making Problem

The N–arm Bandit Problem

The N–arm Bandit is a game based on slot machines [68]. The slot machine has a number of arms or levers. For playing the game, one has to pay a fixed fee. The player will obtain a monetary reward by playing an arm of his choice. The monetary reward may be greater or lesser than the fee he had paid. Also the reward from each arm will be around a mean value with some value of variance. The aim of the player is to obtain maximum reward by playing the game. The play on an arm is considered as an action or decision and the objective is to find the best action from the action set (set of arms). Since the reward is around a mean value, the problem is to find the arm with highest mean value which can be called as the best arm.

In order to introduce the notations used in the thesis, an action of choosing an arm is denoted by a. The goodness of choosing an arm or quality of an arm is the mean value of the reward and is denoted by $Q(a)$. If the mean of all arms are known the best arm is given by the equation

$$a^* = \arg\max_{a \in \mathcal{A}} Q(a) \quad \mathcal{A} = \{1, 2, \ldots, N\}. \tag{6.57}$$

As mentioned earlier, the problem is that the $Q(a)$ values are unknown. One simple and direct method is to play each arm a large number of times. Let the reward

received in playing an arm in the k-th trial be $r^k(a)$. Then, an estimate of $Q(a)$ after s trials is obtained using the equation

$$\hat{Q}^s(a) = \frac{1}{s}\sum_{k=1}^{s} r^k(a) \tag{6.58}$$

and by the Law of large numbers

$$\lim_{s\to\infty} \hat{Q}^s(a) = Q(a). \tag{6.59}$$

In order to update the value of $Q(a)$ (the mean value of reward) in current iteration, the following equation is used

$$\hat{Q}^{s+1}(a) = \hat{Q}^s(a) + \alpha[r^{s+1}(a) - \hat{Q}^s(a)]. \tag{6.60}$$

Above equation tells that the new estimate based on the $(s+1)$-th observation $(r^{s+1}(a))$ is old estimate $Q^s(a)$ plus a small number times the error $(r^{s+1}(a) - Q^s(a))$.

Algorithm 6.13 Algorithm for the N–arm Bandit problem using RL [68]

1: set the learning parameter α and the greedy factor ε
2: set the maximum iteration $s_{\max}$
3: **for all** $a \in \mathcal{A}$ **do** ▷ initialize Q values to zero
4: $\quad Q^0(a) = 0$
5: **end for**
6: **for** $s = 1 \rightarrow s_{\max}$ **do** ▷ learning phase
7: $\quad a$ = getGreedyAction(Q, ε)
8: $\quad r^s(a)$ = getReward(a)
9: $\quad Q$ = updateQ($Q, r^s(a)$) ▷ based on Equation 6.60
10: $\quad$ set the greedy factor $\varepsilon = 1 - s/s_{\max}$
11: **end for**
12: *bestAction* = getBestAction() ▷ retrieval phase

In order to improve the learning phase, an efficient action selection strategy is required. One method would be to take an action with the uniform probability. In this way, one will play all the arms almost equal number of times, i.e., throughout the learning the action space is explored.

Instead of playing all arms several times, it makes sense to play the arms which may be the best arm. One such an efficient algorithm for the action selection is the ε–greedy algorithm. In this algorithm, the greedy arm is played with a probability $(1-\varepsilon)$ and one of the other arms with a probability ε. The greedy arm (a^g) corresponds to the arm with the best estimate of the Q value, i.e.,

$$a^g = \arg\max_{a\in\mathcal{A}} Q^n(a). \tag{6.61}$$

It may be noted that if $\varepsilon = 1$, the algorithm will select one of the actions with uniform probability and if $\varepsilon = 0$, the greedy action will be selected. Initially, the

estimates $Q^s(a)$ may be far from their true value. However as $s \to \infty$, $Q^s(a) \to Q(a)$, the information contained in $Q^s(a)$ becomes increasingly exploitable. So in the ε–greedy algorithm, initially ε is chosen close to 1 and as s increases, ε is gradually reduced.

Finally, a proper balancing of exploration and exploitation of the action space ultimately reduces the number of trials needed to find the best arm. The complete algorithm for the N–arm Bandit problem is given in Algorithm 6.13.

6.7.2 Multi-Stage Decision Making Problem

The Grid World Problem

In the previous subsection, the N–arm Bandit problem has only one state. In many practical situations, the problem might be to find the best action for different states. In order to make the characteristics of such general RL problems clearer, and to identify the different parts of RL, the shortest path problem is considered in this section. Consider the Grid World problem as given in Figure 6.6.

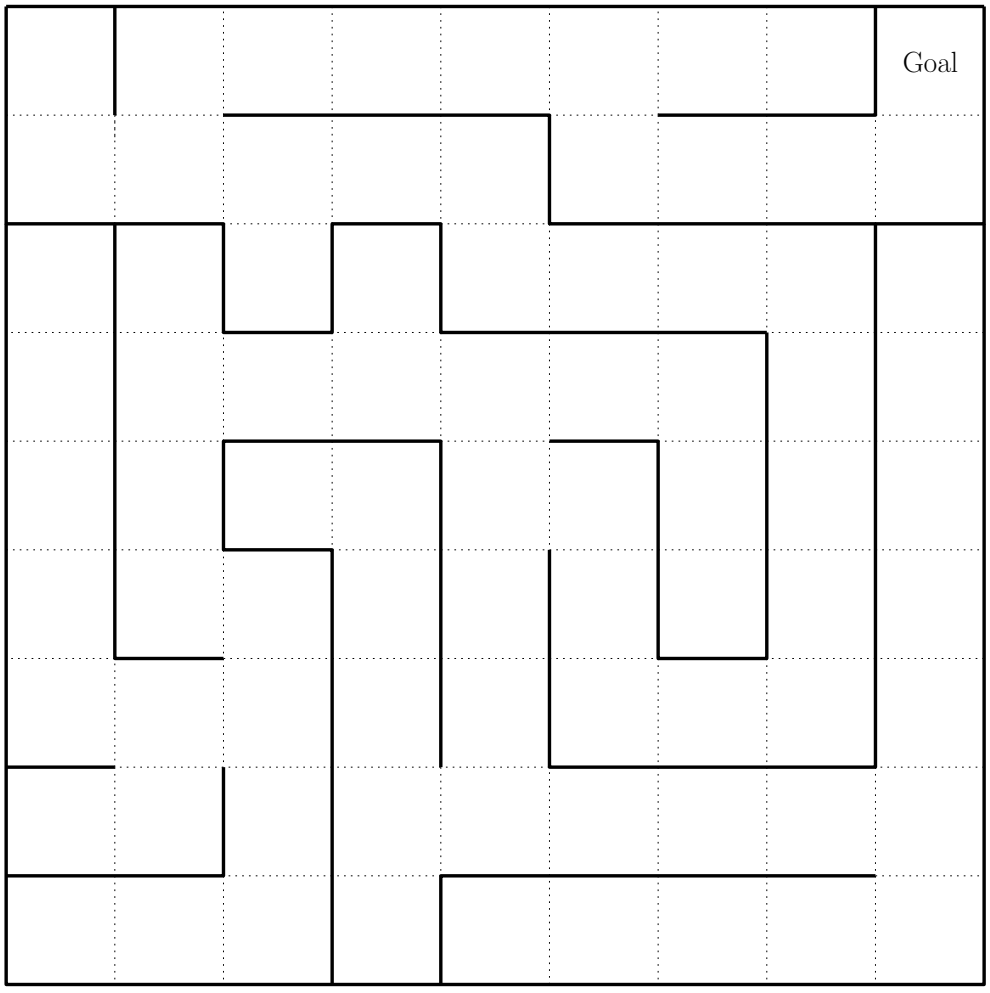

Figure 6.6: The Grid World problem.

The Grid World problem is represented by a maze, see Figure 6.6. The maze is considered as a grid consisting of 81 cells arranged in 9 rows and 9 columns and separated by dotted lines. A robot can be at anyone of the possible cells at any time instant. *Goal* denotes the goal cell to which the robot aim to reach and the solid lines denote walls of the maze. There is a cost associated with each cell transition while the cost of passing through a wall is much higher compared to other transitions. Starting from any initial position in the grid, robot can reach the goal cell by following different paths and correspondingly cost incurred will also vary. The problem is to find an optimum path to reach the goal starting from anyone of the initial cell position. Apparently, the optimum path is the shortest one.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
1	1	*w*	1	1	1	1	1	1	1	1	1	1	1	1	1	*w*	1
2	1	*w*	1	*w*	*w*	*w*	*w*	*w*	*w*	*w*	1	*w*	*w*	*w*	*w*	*w*	1
3	1	1	1	1	1	1	1	1	1	*w*	1	1	1	1	1	1	1
4	*w*	*w*	*w*	*w*	1	*w*	*w*	*w*	1	*w*	*w*	*w*	*w*	*w*	*w*	*w*	*w*
5	1	*w*	1	*w*	1	*w*	1	*w*	1	1	1	1	1	1	1	*w*	1
6	1	*w*	1	*w*	*w*	*w*	1	*w*	*w*	*w*	*w*	*w*	*w*	*w*	1	*w*	1
7	1	*w*	1	1	1	1	1	1	1	1	1	1	1	*w*	1	*w*	1
8	1	*w*	1	*w*	*w*	*w*	*w*	*w*	1	*w*	*w*	*w*	1	*w*	1	*w*	1
9	1	*w*	1	*w*	1	1	1	*w*	1	1	1	*w*	1	*w*	1	*w*	1
10	1	*w*	1	*w*	*w*	*w*	1	*w*	1	*w*	1	*w*	1	*w*	1	*w*	1
11	1	*w*	1	1	1	*w*	1	*w*	1	*w*	1	*w*	1	*w*	1	*w*	1
12	1	*w*	*w*	*w*	1	*w*	1	*w*	1	*w*	1	*w*	*w*	*w*	1	*w*	1
13	1	1	1	1	1	*w*	1	*w*	1	*w*	1	1	1	1	1	*w*	1
14	*w*	*w*	1	*w*	1	*w*	1	*w*	1	*w*	*w*	*w*	*w*	*w*	*w*	*w*	1
15	1	1	1	*w*	1	*w*	1	1	1	1	1	1	1	1	1	1	1
16	*w*	*w*	*w*	*w*	1	*w*	1	*w*	*w*	*w*	*w*	*w*	*w*	*w*	*w*	*w*	1
17	1	1	1	1	1	*w*	1	*w*	1	1	1	1	1	1	1	1	1

Table 6.2: The grid represents the maze with a transition cost for each cell where $w = 1000$.

In Figure 6.6, the maze consists of 81 cells arranged in 9 rows and 9 columns. The treasure is denoted by *Goal* cell which is stated in top-right corner of the maze. The maze is represented by a grid in Table 6.2 where cell transition costs are listed.

All walls of the maze have to be considered. Thus, the grid in Table 6.2 have 17×17 cells where the cells with the odd number of row and column represent the cell itself and cells with the even number of row and column represent walls or gaps for passing. The task is to reach the goal state from any initial position in the grid with the lowest costs.

State Space

The cell number can be taken as a state of the robot at any time. The possible state the robot can occupy at any instant is coming from the entire cell space. In RL

terminology, it is termed as state space. State space in the RL problem is defined as the set of possible states the agent (learner) can occupy at different instants of time. At any instant, the agent will be at anyone of the state from the entire state space. The state of the robot at instant k can be denoted as x_k. The entire state space is then taken as $\mathcal{X}$ so that at any instant k, $x_k \in \mathcal{X}$. In order to reach the goal state *Goal* from the initial state x_0, the robot has to take a series of actions $a_0, a_1, \ldots a_{N-1}$.

Action Space

At any instant k, the robot can take any of the action a_k from the set of permissible actions in the action set or action space $\mathcal{A}_k$. The permissible set of actions at each instant k depends on the current state x_k of the robot. If the Robot stays in any of the cells in the first column, "move to Left" is not possible. Similarly for each cell in the grid, there is a set of possible cell movements or state transitions. The set of possible actions or cell transitions at current state x_k is denoted as $\mathcal{A}_{x_k}$.

State–Evolution Model

On taking an action, the robot proceeds to the next cell position which is a function of the current state and action. In other words, the state occupied by the robot in $k+1$, x_{k+1} depends on x_k and a_k, i.e.,

$$x_{k+1} = f(x_k, a_k). \tag{6.62}$$

The aim of a robot in the grid is to reach the goal state starting from its initial position or state at minimum cost. At each step, it takes an action which is followed by state transition or movement in the grid. The actions which make state transitions to reach the goal state at minimum cost points out the optimum solution. Therefore, the shortest path problem can be stated as finding the sequence of actions $a_0, a_1, \ldots a_{N-1}$ starting from any initial state such that the total cost for reaching goal state *Goal* is minimum.

Policy

As explained in the previous section, whenever an action a_k is taken in a state x_k, state transition occurs governed by Equation 6.62. Ultimate learning solution is to find out a rule by which an action is chosen at any of the possible states. In other words, a good mapping from the state space $\mathcal{X}$ to action space $\mathcal{A}$ has to be derived.

$$\pi : \mathcal{X} \rightarrow \mathcal{A} \tag{6.63}$$

In RL problems, any mapping from state space to action space is termed as policy. The optimum policy at any state x is denoted as $\pi^*(x)$.

Q–learning

Q–learning is a RL algorithm that learns the values of the function $Q(x,a)$ to find an optimal policy [68]. The value of the function $Q(x,a)$ indicates how good is to perform action a at the given state x.

At each iteration of the algorithm, from the current state x, it chooses an action a based on some strategy and reaches the new state y and obtains reward $g(x,a,y)$ which is used for updating the Q value of the state–action pair as

$$Q^{s+1}(x,a) = Q^s(x,a) + \alpha[g(x,a,y) + \gamma \min_{a' \in \mathcal{A}} Q^s(y,a') - Q^s(x,a)] \tag{6.64}$$

where $\alpha \in [0,1]$ is the learning parameter and determines the extent of modification of the Q value at each iteration of the learning phase. Discount factor $\gamma \in [0,1]$ indicates the real goodness of an action. It may not be reflected by its immediate reward. Value of γ is decided by the problem environment to account how much the future rewards to be discounted to rate the goodness of the policy at the present state. A value 1 indicates that all the future rewards are having equal importance as the immediate reward. In this shortest path problem, since all the costs are relevant to the same extent, γ is taken 1.

When the learning parameter α is sufficiently small and if all possible (x,a) combinations of state and action occur sufficiently often, then the above iteration given by Equation 6.64 will result Q^s converging to Q^*. The complete algorithm for the Grid World problem is given as follows.

Algorithm 6.14 Learning algorithm for the Grid World problem using RL [68]

```
set the learning parameter α, the discount factor γ and the greedy factor ε
set the maximum iteration s_max
for all x ∈ X, a ∈ A_x do                          ▷ initialize Q values to zero
    Q^0(x,a) = 0
end for
for s = 1 → s_max do                               ▷ learning phase
    x_0 = getRandomInitState()
    k = 0
    while isNotGoalState(x_k) do
        x_k = getCurrentState()
        a_k = getGreedyAction(Q, ε)
        x_{k+1} = getNextState(x_k, a_k)           ▷ based on Equation 6.62
        r_k = g(x_k, a_k, x_{k+1})
        Q = updateQ(Q, r_k)                        ▷ based on Equation 6.64
        k = k + 1
    end while
    set the greedy factor ε = 1 − s/s_max
end for
```

The algorithm described so far gives only the learning phase. To get the best path in the maze, the retrieval phase of the algorithm is also needed. The complete algorithm of the retrieval phase for getting the path is given as follows.

Algorithm 6.15 Policy retrieval algorithm for the Grid World problem using RL

```
1: x_0 = Start
2: addStateToPath(x_0)
3: k = 0
4: while isNotGoalState(x_k) do
5:     x_k = getCurrentState()
6:     a_k = getGreedyAction(Q, ε)
7:     x_{k+1} = getNextState(x_k, a_k)        ▷ based on Equation 6.62
8:     addStateToPath(x_{k+1})
9:     k = k + 1
10: end while
```

Load Balancing as Multi-Stage Decision Making Problem

In order to view the LB problem as a multi-stage decision making problem, the various stages of the problem are to be identified [67]. Consider a system with $n \in \mathbb{N}$ flows $f_1, f_2, \ldots, f_n \in \mathbb{N}_0$ committed for allocation to $n = m \cdot p$ ports. Then, the LB problem involves selecting $p \in \mathbb{N}$ flows to be allocated to the first MUX from flows $\mathcal{R}_1 = \{f_i \mid i \in \{1, \ldots, n\}\}$, i.e., determined by subset $\mathcal{S}_1$. For the second MUX, p flows are selected from flows $\mathcal{R}_2 = \{f_i \mid i \in \{1, \ldots, n\}\} \backslash \mathcal{S}_1\}$ and described by $\mathcal{S}_2$. The last, i.e., the m-th MUX is occupied by remaining p flows $\mathcal{R}_m = \{f_i \mid i \in \{1, \ldots, n\} \backslash \bigcup_{j=1}^{m-1} \mathcal{S}_j = \mathcal{S}_m\}$ and in fact, there is no selection procedure at all and the subset $\mathcal{S}_m$ is determined directly.

In general, the i-th MUX selects p flows from flows $\mathcal{R}_i = \{f_j \mid j \in \{1, \ldots, n\} \backslash \bigcup_{k=1}^{i-1} \mathcal{S}_k\}$ and a subset $\mathcal{S}_i$ contains flow indices of the i-th MUX.

The problem statement follows. Initially, there are p flows to be allocated in the i-th MUX chosen from $n - (i-1)p$ flows, see Figure 6.7. In this formulation, a flow to be allocated at $stage_k$ is denoted as F_k^{A} and is based on an action a_k. In RL terminology, the action a_k corresponds to a flow allocation either $f_{\varphi(k,\mathcal{R}_i)}$ or 0 to the i-th MUX at $stage_k$, i.e.,

$$F_k^{\mathrm{A}} = \begin{cases} 0 & \text{if} \quad a_k = 0 \\ f_{\varphi(k,\mathcal{R}_i)} & \text{if} \quad a_k = 1. \end{cases} \tag{6.65}$$

Therefore, the action set $\mathcal{A}_k$ consists of either 2 possibilities (allocate $a_k = 1$ and do not allocate $a_k = 0$) or 1 possibility (allocate $a_k = 1$ or do not allocate $a_k = 0$) at $stage_k$. That is,

$$\mathcal{A}_k = \{a_k^{\min}, \ldots, a_k^{\max}\}, \tag{6.66}$$

$a_k^{\min}$ being the minimum possible action at $stage_k$ and $a_k^{\max}$ being the maximum possible action at $stage_k$. Values of $a_k^{\min}$ and $a_k^{\max}$ depend on the total flow which has already been allocated at the previous $k-1$ stages, the number of flows already allocated, a flow at the k-th stage and flows that can be allocated at the remaining $(n - (i-1)p) - k$ stages.

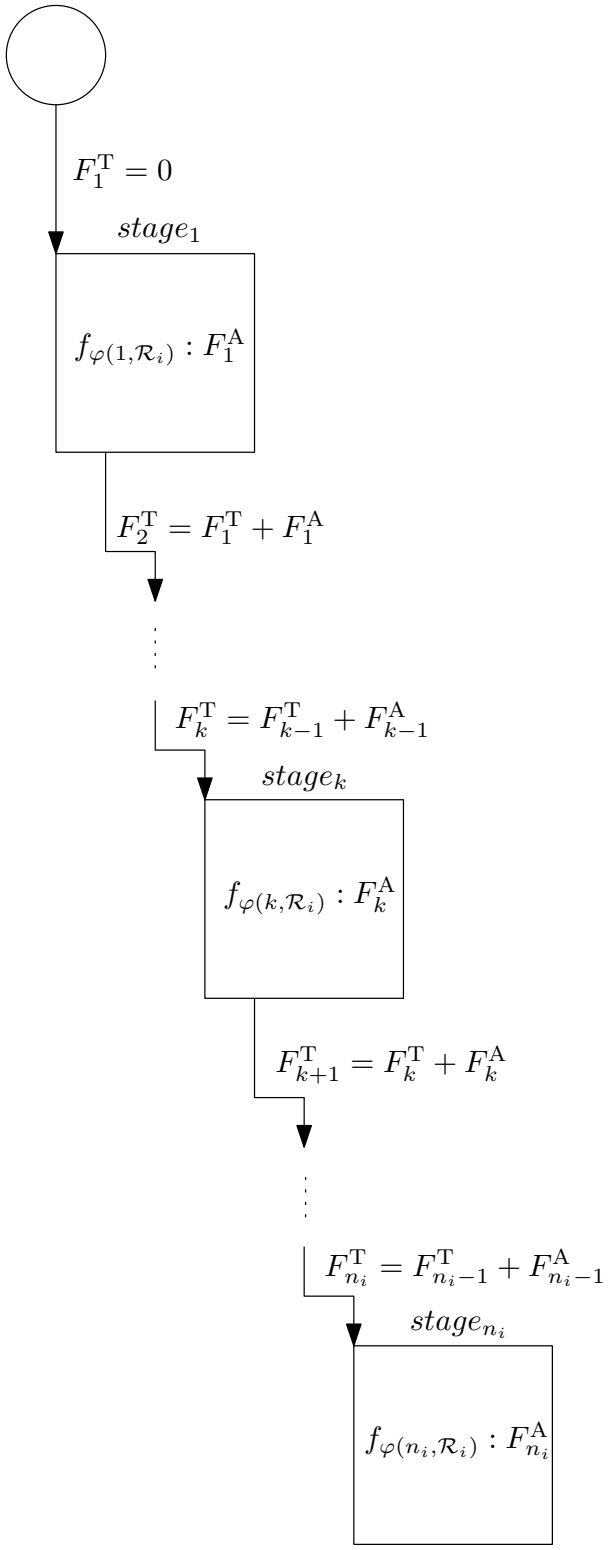

Figure 6.7: Visualization of p flows allocation to the i-th MUX where $n_i = n-(i-1)p$.

The initial state is denoted as $stage_1$. At $stage_1$, a decision is made on whether a flow $f_{\varphi(1,\mathcal{R}_i)}$ is allocated or not. This action is denoted as a_1 and corresponds to F_1^{A} allocation at $stage_1$.

Upon having made this decision, $stage_2$ is reached. The expression $(F_1^{\mathrm{T}} + F_1^{\mathrm{A}})$ represents the total flow which has already been allocated at the previous stages, i.e., $stage_1$. At $stage_2$, a decision a_2 is made on whether a flow $f_{\varphi(2,\mathcal{R}_i)}$ allocates or does not allocate. Generally at $stage_k$, a decision is made on whether a flow f_k is allocated or not. Finally, $stage_{n-(i-1)p}$ is reached and a decision $a_{n-(i-1)p}$ is made on whether a flow $f_{\varphi(n-(i-1)p,\mathcal{R}_i)}$ is allocated or not.

Each state at any $stage_k$ can be defined as a tuple (k, F_k^{T}) where k is the stage number and F_k^{T} is the total flow which has already been allocated at the previous $k-1$ stages.

Thus, for $k = 1$, the state information is denoted as $(1, F_1^{\mathrm{T}})$ where F_1^{T} is equal to 0, since no decision concerning any flow allocation has been made so far. The algorithm for the LB problem selects one among the permissible set of actions and either allocates or does not allocate a flow $f_{\varphi(1,\mathcal{R}_i)}$ at $stage_1$ so that it reaches the next stage $k = 2$ with the total flow already allocated and $(n-(i-1)p)-1$ flows for an allocation decision. Transition from $(1, F_1^{\mathrm{T}})$ on performing an action $a_1 \in \mathcal{A}_1$ results in the next state reached as $(2, F_2^{\mathrm{T}})$ where

$$F_2^{\mathrm{T}} = F_1^{\mathrm{T}} + F_1^{\mathrm{A}}. \tag{6.67}$$

Generally at $stage_k$, from a state x_k on performing an action a_k reaches a state x_{k+1}, i.e., state transition is from (k, F_k^{T}) to $(k+1, F_{k+1}^{\mathrm{T}})$ where

$$F_{k+1}^{\mathrm{T}} = F_k^{\mathrm{T}} + F_k^{\mathrm{A}}. \tag{6.68}$$

This repeats until the last stage. Therefore, a state transition can be denoted as

$$x_{k+1} = f(x_k, a_k) \tag{6.69}$$

where $f(x_k, a_k)$ is the function of state transition defined by Equation 6.68.

Thus, the algorithm for the LB problem can be treated as one of finding an optimum mapping from the state space $\mathcal{X}$ to the action space $\mathcal{A}$. The algorithm design for the LB problem is finding or learning a good or optimal policy (flows allocation) which is the optimum allocation at each stage. Such an allocation can be treated as elements of an optimum policy π^*. For finding the cost of allocation, it cumulates the costs at each of the $n-(i-1)p$ stages of the problem. These costs can be treated as a reward for performance of an action in the perspective of the LB problem. The cost of generation on following a policy π can be treated as a measure of goodness of that policy. The Q–learning technique is employed to cumulate costs and thus find out the optimum policy.

For updating the Q value associated with the different state–action pairs, one should cumulate the total reward at different stages of allocation. In the LB problem, the reward function $g(x_k, a_k, x_{k+1})$ can be chosen as $-F_k^{\mathrm{A}}$ at $stage_k$. The rewards are negative, since Q–learning is considered as a minimization problem. In the RL terminology, the immediate reward is

$$r_k = g(x_k, a_k, x_{k+1}). \tag{6.70}$$

Since the aim is to allocate as large as possible a total flow, the estimated Q values of the state–action pair are modified at each step of learning as

$$Q^{s+1}(x_k, a_k) = Q^s(x_k, a_k) + \alpha[g(x_k, a_k, x_{k+1}) + \gamma \min_{a' \in \mathcal{A}_{k+1}} Q^s(x_{k+1}, a') - Q^s(x_k, a_k)]. \quad (6.71)$$

Here, α is the learning parameter and γ is the discount factor. When the system comes to the last stage of decision making, there is no need of accounting the future effects and then, the estimate of Q value is updated using the equation

$$Q^{s+1}(x_k, a_k) = Q^s(x_k, a_k) + \alpha[g(x_k, a_k, x_{k+1}) - Q^s(x_k, a_k)]. \quad (6.72)$$

For finding an optimum policy, a learning algorithm is designed. It iterates through each of the $n - (i-1)p$ stages at each step of learning. As the learning steps are carried out a sufficient number of times, the estimated Q values of state–action pairs will approach the optimum so that the optimum policy $\pi^*(x)$ corresponding to any state x can be easily retrieved.

6.7.3 RL Algorithm for the LB Problem using ε–greedy Strategy

In the previous subsection, the LB problem is formulated as a multi-stage decision making problem. To find the best policy or the best action corresponding to each state, the RL technique is used. The solution consists of two phases, namely the learning phase and the policy retrieval phase [67].

To carry out the learning task, one issue concerns how to select an action from the action space. In this subsection, the ε–greedy strategy of exploring action space is used.

For solving this multi-stage problem using RL, the first step is fixing of state space $\mathcal{X}$ and action space $\mathcal{A}$ precisely. The whole concept for the i-th MUX is explained in a general way where the number of flows is $n - (i-1)p$.

The fixing of state space $\mathcal{X}$ [67] primarily depends on the number of flows and the possible values of the total flow in the i-th MUX (which in turn directly depends on the minimum and maximum values of each flow). Since there are $n - (i-1)p$ stages for solution of the problem, the state space is also divided into $n-(i-1)p$ subspaces. Thus, if there are $n - (i-1)p$ flows to be allocated, then

$$\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \ldots \cup \mathcal{X}_{n-(i-1)p}. \quad (6.73)$$

The allocation problem should go through $n - (i-1)p$ stages for making decision to allocate or not to allocate for each of the $n - (i-1)p$ flows. At any $stage_k$, the part of state space to be considered $\mathcal{X}_k$ consists of the different tuples having the stage number as k and the total flow already allocated varying from F_k^{Tmin} to F_k^{Tmax} where F_k^{Tmin} is the minimum possible total flow already allocated and F_k^{Tmax} the maximum possible total flow already allocated at the previous $k-1$ stages. Thus,

$$\mathcal{X}_k = \{(k, F_k^{\mathrm{Tmin}}), \ldots, (k, F_k^{\mathrm{Tmax}})\} \quad (6.74)$$

where F_k^{Tmin} is the minimum possible total flow already allocated at the previous $k-1$ stages, i.e.,

$$F_k^{\mathrm{Tmin}} = 0 \tag{6.75}$$

and F_k^{Tmax} is the maximum possible total flow already allocated at the previous $k-1$ stages, i.e.,

$$F_k^{\mathrm{Tmax}} = \sum_{j=1}^{k-1} f_{\varphi(j,\mathcal{R}_i)}. \tag{6.76}$$

At each step, the LB problem algorithm will select an action from the permissible set of actions and forward the system to one among the next permissible states. Therefore, the action set $\mathcal{A}_k$ is a dynamically varying one, depending on the flows already allocated at the previously considered stages. As the number of MUXes or number of ports in each MUX increases, the number of states in the state space increases. Thus, state space and action space are both discrete.

The action set $\mathcal{A}_k$ [67] consists of either 2 possibilities (allocate $a_k = 1$ and do not allocate $a_k = 0$) or 1 possibility (allocate $a_k = 1$ or do not allocate $a_k = 0$) at $stage_k$. At the current state x_k, the action set $\mathcal{A}_k$ depends on the total flow already allocated F_k^{T}, the number of flows already allocated p^{A}, a flow at the k-th stage $f_{\varphi(k,\mathcal{R}_i)}$ and flows that can be allocated at the remaining $(n-(i-1)p)-k$ stages, i.e., $f_{\varphi(k+1,\mathcal{R}_i)}, \ldots, f_{\varphi(n-(i-1)p,\mathcal{R}_i)}$. Therefore, the action set $\mathcal{A}_k$ is dynamic in nature in the sense that it depends on the total flow already allocated up to that stage and also the number of flows already allocated at the previous $k-1$ stages. If F_k^{T} is the total flow already allocated, p^{A} is the number of flows already allocated, $f_{\varphi(k,\mathcal{R}_i)}$ is a flow at $stage_k$, the minimum value and the maximum value of action a_k are defined as

$$a_k^{\min} = \begin{cases} 0 \text{ if } p = p^{\mathrm{A}} \\ 0 \text{ if } F - F_k^{\mathrm{T}} < f_{\varphi(k,\mathcal{R}_i)} \\ 0 \text{ if } p - p^{\mathrm{A}} < (n-(i-1)p) - k \quad \wedge \\ \qquad F - F_k^{\mathrm{T}} < L_{p-p^{\mathrm{A}}-1,(n-(i-1)p)-k} + f_{\varphi(k,\mathcal{R}_i)} \\ 0 \text{ if } p - p^{\mathrm{A}} < (n-(i-1)p) - k \quad \wedge \\ \qquad F - F_k^{\mathrm{T}} \geq L_{p-p^{\mathrm{A}},(n-(i-1)p)-k} \\ 1 \text{ otherwise} \end{cases}$$

$$a_k^{\max} = \begin{cases} 0 \text{ if } p = p^{\mathrm{A}} \\ 0 \text{ if } F - F_k^{\mathrm{T}} < f_{\varphi(k,\mathcal{R}_i)} \\ 0 \text{ if } p - p^{\mathrm{A}} < (n-(i-1)p) - k \quad \wedge \\ \qquad F - F_k^{\mathrm{T}} < L_{p-p^{\mathrm{A}}-1,(n-(i-1)p)-k} + f_{\varphi(k,\mathcal{R}_i)} \\ 1 \text{ otherwise} \end{cases} \tag{6.77}$$

where $L_{u,v}$ denotes the sum of the u smallest flows at last v stages. In order to illustrate the situation from action's point of view, a visualization in Figure 6.8 is given. Below, Equation 6.77 is discussed in more detail.

The conditions $F - F_k^{\mathrm{T}} < f_{\varphi(k,\mathcal{R}_i)}$ and $p = p^{\mathrm{A}}$ are common for both $a_k^{\min}$ and $a_k^{\max}$. Clearly, if a flow $f_{\varphi(k,\mathcal{R}_i)}$ at $stage_k$ is greater than $F - F_k^{\mathrm{T}}$ (the rest what remains to allocate), it does not allow the flow $f_{\varphi(k,\mathcal{R}_i)}$ to be allocated at $stage_k$. Similarly, if

the number of flows already allocated p^{A} is equal to the number of ports p, there is no free port available for a flow allocation. Therefore, both $a_k^{\min}$ and $a_k^{\max}$ are equal to 0 if the condition is true.

The second condition $p-p^{\mathrm{A}} < (n-(i-1)p)-k \wedge F-F_k^{\mathrm{T}} < L_{p-p^{\mathrm{A}}-1,(n-(i-1)p)-k} + f_{\varphi(k,\mathcal{R}_i)}$ is also common for both $a_k^{\min}$ and $a_k^{\max}$. The condition examines whether the flow $f_{\varphi(k,\mathcal{R}_i)}$ has to be necessarily allocated or not. The first part, $p-p^{\mathrm{A}} < (n-(i-1)p)-k$, checks whether there are enough flows at the remaining $(n-(i-1)p)-k$ stages to fill remaining free ports if the flow $f_{\varphi(k,\mathcal{R}_i)}$ is not to be allocated. The second part, $F-F_k^{\mathrm{T}} < L_{p-p^{\mathrm{A}}-1,(n-(i-1)p)-k} + f_{\varphi(k,\mathcal{R}_i)}$, checks if the flow $f_{\varphi(k,\mathcal{R}_i)}$ together with the sum of the $p-p^{\mathrm{A}}-1$ smallest flows at the last $(n-(i-1)p)-k$ stages are higher than $F-F_k^{\mathrm{T}}$, ensuring that the total i-th MUX flow is less or equal than F. If both parts are satisfied, both $a_k^{\min}$ and $a_k^{\max}$ are equal to 0.

The last condition deals with $a_k^{\min}$ only. The condition $p-p^{\mathrm{A}} < (n-(i-1)p)-k \wedge F-F_k^{\mathrm{T}} \geq L_{p-p^{\mathrm{A}},(n-(i-1)p)-k}$ consists of two parts. The first part, $p-p^{\mathrm{A}} < (n-(i-1)p)-k$, checks again whether there are enough flows at the remaining $(n-(i-1)p)-k$ stages to fill remaining free ports if the flow $f_{\varphi(k,\mathcal{R}_i)}$ is not to be allocated. The second part, $F-F_k^{\mathrm{T}} \geq L_{p-p^{\mathrm{A}},(n-(i-1)p)-k}$, determines whether the sum of the $p-p^{\mathrm{A}}$ smallest flows at the last $(n-(i-1)p)-k$ stages is less than $F-F_k^{\mathrm{T}}$, ensuring that the total i-th MUX flow is less or equal than F. If both parts are satisfied, the flow $f_{\varphi(k,\mathcal{R}_i)}$ does not necessarily have to be allocated, since all LB problem conditions can be still satisfied at the remaining $(n-(i-1)p)-k$ stages.

Clearly, if none of the above-mentioned conditions is satisfied, both $a_k^{\min}$ and $a_k^{\max}$ are equal to 1 and the flow $f_{\varphi(k,\mathcal{R}_i)}$ has to be allocated in the i-th MUX.

To conclude, if $F-F_k^{\mathrm{T}} < f_{\varphi(k,\mathcal{R}_i)}$ or $p = p^{\mathrm{A}}$ is satisfied, the choice can only be made from one action – do not allocate $a_k = 0$. There is also no choice if $p-p^{\mathrm{A}} < (n-(i-1)p)-k \wedge F-F_k^{\mathrm{T}} < L_{p-p^{\mathrm{A}}-1,(n-(i-1)p)-k} + f_{\varphi(k,\mathcal{R}_i)}$ is true, since it leads to only one possible action again – do not allocate $a_k = 0$.

The situation when two actions are feasible (allocate $a_k = 1$ and do not allocate $a_k = 0$) depends on the condition $p-p^{\mathrm{A}} < (n-(i-1)p)-k \wedge F-F_k^{\mathrm{T}} \geq L_{p-p^{\mathrm{A}},(n-(i-1)p)-k}$. The decision to allocate the flow $f_{\varphi(k,\mathcal{R}_i)}$ can be seen as a substitution of the $(p-p^{\mathrm{A}})$-th smallest flow at last $(n-(i-1)p)-k$ stages for the flow $f_{\varphi(k,\mathcal{R}_i)}$.

All other situations lead to both $a_k^{\min}$ and $a_k^{\max}$ being equal to 1 and the only one possible decision is to allocate the flow $f_{\varphi(k,\mathcal{R}_i)}$ in the i-th MUX.

The learning procedure can now be summarized [67], see Algorithm 6.16. Initially, the total flow already allocated is set to 0 at $stage_1$. Then, an action is performed that either allocates or not the flow at $stage_1$ and then, it proceeds to the next stage ($k = 2$) with the total flow already allocated. This proceeds until all the $n-(i-1)p$ flows are either allocated or not. At each state transition step, the estimated Q value of the state–action pair is updated using Equation 6.71.

As the learning process reaches the last stage, a flow at $stage_{n-(i-1)p}$ is either allocated or not. Then, the Q value is updated using Equation 6.72. The transition process is repeated a sufficient number of times (iterations) and each time the allo-

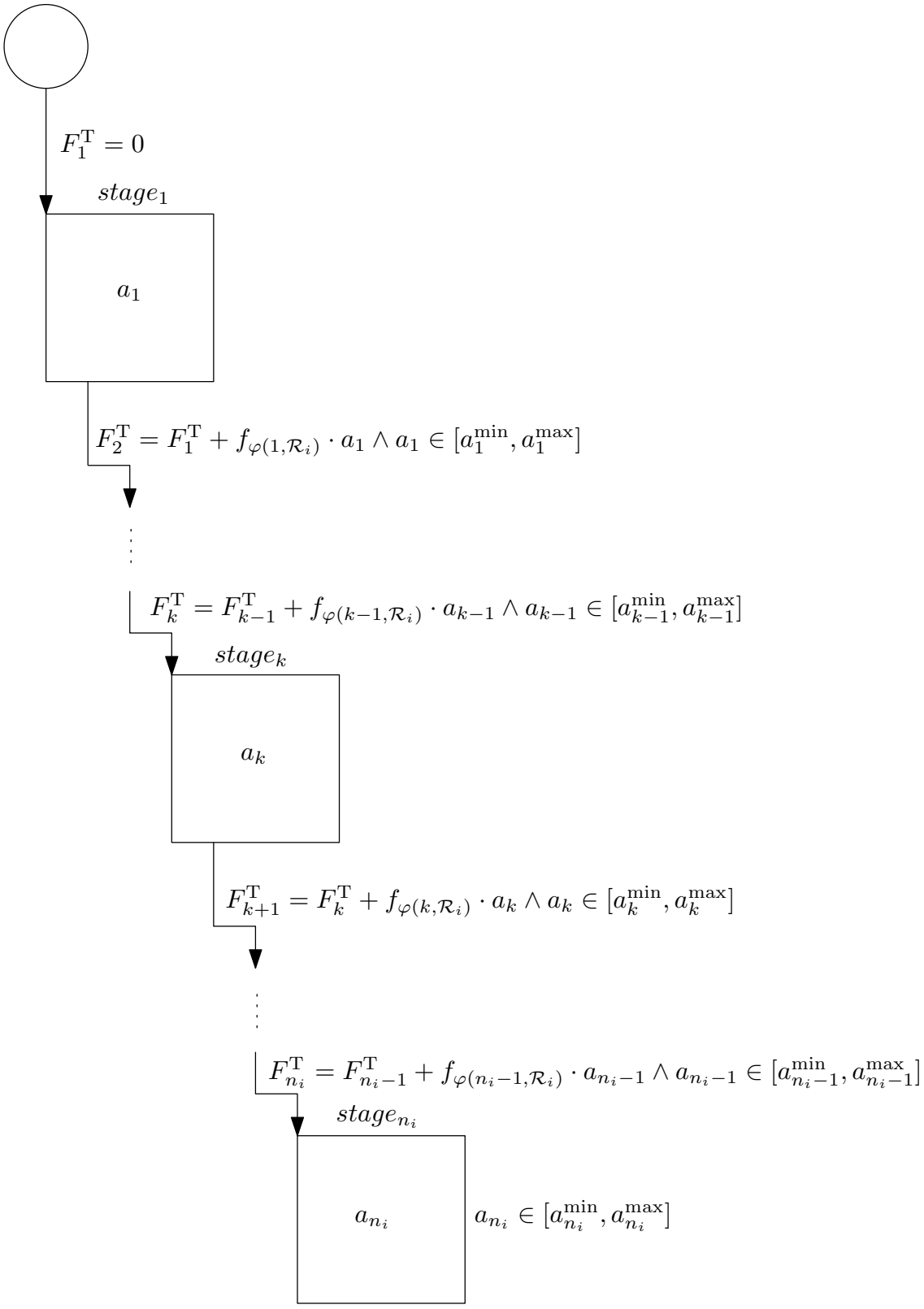

Figure 6.8: Visualization of p flows allocation to the i-th MUX from action's point of view, $n_i = n - (i-1)p$. Corresponding $a_k^{\min}$ and $a_k^{\max}$ are based on (6.77).

cation process goes through all the $n-(i-1)p$ stages.

Algorithm 6.16 Learning algorithm for the LB problem of the i-th MUX using ε–greedy strategy

load flows $\mathcal{R}_i$, average flow F
set the learning parameter α, the discount factor γ and the greedy factor ε
set the maximum iteration $s_{\max}$
set Q values to zeros
for $i = 1 \rightarrow n-(i-1)p$ **do** ▷ stages initialization
    $\mathcal{X}_i$ = initStage(i)
    for all $x_k \in \mathcal{X}_i$ **do** ▷ states initialization
        $a_k^{\min}$ = getMinAction() ▷ based on Equation 6.77
        $a_k^{\max}$ = getMaxAction() ▷ based on Equation 6.77
        setStatePermissibleActions$(x_k, a_k^{\min}, a_k^{\max})$
    end for
end for
for $s = 1 \rightarrow s_{\max}$ **do** ▷ learning phase
    $F_1^{\mathrm{T}} = 0, p^{\mathrm{A}} = 0$
    for $k = 1 \rightarrow n-(i-1)p$ **do**
        x_k = getCurrentState(k, F_k^{T})
        $\mathcal{A}_k$ = getActions$(k, x_k, F, p^{\mathrm{A}})$ ▷ based on Equation 6.77
        a_k = getGreedyAction$(Q, \mathcal{A}_k, \varepsilon)$
        $p^{\mathrm{A}} = p^{\mathrm{A}} + a_k$
        $F_{k+1}^{\mathrm{T}} = F_k^{\mathrm{T}} + f_{\varphi(k,\mathcal{R}_i)} \cdot a_k$
        if $k < n-(i-1)p$ **then**
            Q = updateQ(Q, x_k, a_k) ▷ based on Equation 6.71
        else
            Q = updateQ(Q, x_k, a_k) ▷ based on Equation 6.72
        end if
    end for
    set the greedy factor $\varepsilon = 1 - s/s_{\max}$
end for

6.7.4 Policy Retrieval

After the learning phase is done, the retrieval phase begins. The system is learnt and the learnt values are stored in a lookup table. The retrieval phase accesses the lookup table in order to retrieve the required results. The learnt values in the lookup table can be used unless the parameters of the system change. In this case, the system must be learnt again by triggering a new run of learning phase.

As the learning proceeds, and the Q values of state-action pairs are updated, Q^s approaches Q^*. Next, the optimum Q^s values are used to obtain the optimum allocation. The retrieval algorithm [67] is summarized in Algorithm 6.17. At $stage_1$, $F_1^{\mathrm{T}} = 0$ is initialized and thus, the state of the system is $(1, F_1^{\mathrm{T}})$. The algorithm finds the greedy action at this stage as a_1^{g} which is the best allocation F_1^{A} for $stage_1$. The

Algorithm 6.17 Policy retrieval algorithm for the LB problem of the i-th MUX using RL

load flows $\mathcal{R}_i$, average flow F, Q values
$F_1^{\mathrm{T}} = 0, p^{\mathrm{A}} = 0$
$\mathcal{S}_i = \{\}$
for $k = 1 \rightarrow n - (i-1)p$ **do** ▷ retrieval phase
  $x_k = \mathrm{getCurrentState}(k, F_k^{\mathrm{T}})$
  $\mathcal{A}_k = \mathrm{getActions}(k, x_k, F, p^{\mathrm{A}})$ ▷ based on Equation 6.77
  $a_k^{\mathrm{g}} = \arg\min_{a \in \mathcal{A}_k} Q(x_k, a)$
  if $a_k^{\mathrm{g}} = 1$ **then**
    $\mathcal{S}_i = \mathcal{S}_i \cup \varphi(k, \mathcal{R}_i)$
  end if
  $p^{\mathrm{A}} = p^{\mathrm{A}} + a_k^{\mathrm{g}}$
  $F_{k+1}^{\mathrm{T}} = F_k^{\mathrm{T}} + f_{\varphi(k,\mathcal{R}_i)} \cdot a_k^{\mathrm{g}}$
end for

retrieval algorithm reaches the next state as $(2, F_2^{\mathrm{T}})$ where $F_2^{\mathrm{T}} = F_1^{\mathrm{T}} + a_1^{\mathrm{g}} \cdot f_{\varphi(1,\mathcal{R}_i)}$ and finds the greedy action corresponding to $stage_2$ as a_2^{g}. This proceeds up to $stage_{n-(i-1)p}$. Finally, a set of actions (allocations) $a_1^{\mathrm{g}}, a_2^{\mathrm{g}}, \ldots, a_{n-(i-1)p}^{\mathrm{g}}$ is obtained which is the optimum allocation $F_1^{\mathrm{A}}, F_2^{\mathrm{A}}, \ldots, F_{n-(i-1)p}^{\mathrm{A}}$ for the i-th MUX and thus, it builds up the subset $\mathcal{S}_i = \{\varphi(j, \mathcal{R}_i) \mid F_j^{\mathrm{A}} \neq 0 \quad \forall j = 1, \ldots, n - (i-1)p\}$.

Algorithm 6.18 The complete LB problem algorithm considering m MUXes using RL

load flows $\mathcal{R}_1$
for $i = 1 \rightarrow m - 1$ **do** ▷ get LB of the i-th MUX
  $F = \left\lceil \sum_{j=1}^{n-(i-1)p} f_{\varphi(j,\mathcal{R}_i)} / (m - i + 1) \right\rceil$
  $Q = \mathrm{learningPhase}(\mathcal{R}_i, F)$ ▷ based on Algorithm 6.16
  $\mathcal{S}_i = \mathrm{retrievalPhase}(\mathcal{R}_i, F, Q)$ ▷ based on Algorithm 6.17
  $\mathcal{R}_{i+1} = \{f_j \mid j \in \{1, \ldots, n\} \setminus \bigcup_{k=1}^{i} \mathcal{S}_k\}$
end for
$\mathcal{S}_m = \{\varphi(j, \mathcal{R}_m) \mid \forall j = 1, \ldots, p\}$

6.7.5 Complete Algorithm

Finally, an algorithm considering all $m \in \mathbb{N}$ MUXes is proposed [67], see Algorithm 6.18. The idea is to learn and retrieve $\mathcal{S}_1$ for the first MUX, then reduce a set of flows $\mathcal{R}_1$ by the allocated flows and get a set of flows $\mathcal{R}_2$ for the second MUX, etc. Generally, the i-th MUX is learnt, $\mathcal{S}_i$ is retrieved and $\mathcal{R}_i$ is reduced by $\mathcal{S}_i$ giving $\mathcal{R}_{i+1}$ for the $(i+1)$-th MUX. Regarding the last MUX, the m-th MUX, a set $\mathcal{R}_m$ determines directly $\mathcal{S}_m$.

Chapter 7

Load Balancing – Numerical Results

All algorithms have been implemented in C++ and Matlab (R2018a, 64-bit) on a personal computer equipped with Intel(R) Core(TM) i7-8750H CPU (@2.20 GHz, 6 Cores, 12 Threads, 9M Cache, Turbo Boost up to 4.10 GHz) and 16 GB RAM (DDR4, 2 666 MHz) memory. The LB problem solving methods are examined on three test cases and numerical results are compared with each other in each section of this chapter.

Flow	**[B]**	**Flow**	**[B]**
f_1	9,282	f_{16}	1,474
f_2	5,176	f_{17}	2,531
f_3	1,948	f_{18}	4,930
f_4	123	f_{19}	507
f_5	9,577	f_{20}	1,993
f_6	5,930	f_{21}	8,273
f_7	7,764	f_{22}	6,070
f_8	9,480	f_{23}	1,241
f_9	7,989	f_{24}	9,620
f_{10}	565	f_{25}	9,052
f_{11}	6,287	f_{26}	3,319
f_{12}	466	f_{27}	9,036
f_{13}	3,815	f_{28}	2,712
f_{14}	1,360	f_{29}	5,057
f_{15}	5,205	f_{30}	5,338

Table 7.1: Flows in the Test Case 1.

7.1 Test Case 1 – Formulation

In this section, the Test Case 1 of the LB problem is introduced and investigated. It consists of $m = 2$ MUXes with $p = 15$ ingoing ports each and therefore, in terms

of the number of MUXes, it corresponds to the smallest possible task when LB can be applied.

The task is very similar to the well-known Partition problem. However, the constraint considering 15 ingoing ports for each MUX makes a difference. The Partition problem has no constraint or restriction dealing with exactly 15 positive integers in both subsets.

In Table 7.1, the flows used in the Test Case 1 are stated. There are $n = m \cdot p = 2 \cdot 15 = 30$ flows with values randomly generated in the range from 0 B to 10 kB.

7.2 Test Case 1 – Results

In the following subsections, the results related to the Test Case 1 are given. The Test Case 1 is solved by DP, GH, ILP, MDE and RL, respectively. All the listed solution methods are implemented in C++ and Matlab. Finally, the results are investigated with respect to the error and computation time. The error is defined as the objective function of the LB problem, see Equation 6.1.

Taking into consideration the sum of all flows in the Test Case 1, the best possible LB can be reached with the total flow allocation of 7,360 B to the first MUX and 7,360 B to the second MUX. Therefore, the best possible error is equal to 0 based on Equation 6.1.

7.2.1 Dynamic Programming

As the DP approach is designed, DP does not bring any partially stochastic problem solving. However, all executions are following the same computational process. In other words, the DP method is strictly deterministic in every single step and thus, the optimization process is easily predictable.

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	0.00	1,378	0.00	2,004
2	0.00	1,376	0.00	1,894
3	0.00	1,362	0.00	1,883
4	0.00	1,362	0.00	1,870
5	0.00	1,357	0.00	1,893
6	0.00	1,363	0.00	1,899
7	0.00	1,381	0.00	1,915
8	0.00	1,368	0.00	1,884
9	0.00	1,367	0.00	1,893
10	0.00	1,394	0.00	1,886

Table 7.2: Results for the Test Case 1 using DP in each execution.

The mentioned determinism can be illustrated in Table 7.2 where the results produced in C++ and Matlab for the Test Case 1 using DP in each execution are stated. The error is equal to 0 in all executions regardless of a programming language.

The zero error means a global optimum has been reached, since it is not possible to acquire a lower value of the LB objective function. Unfortunately, the DP approach does not give the answer whether there is one and only one global optimum or multiple different global optima with the same error exist, however, it is not an important question from the LB point of view. A certainty given by the LB objective function form – a better solution than the current one is not possible to gain – is sufficient.

The detailed DP results containing the best flow allocation for the Test Case 1 are given in Appendix A in Table A.1 for C++ and in Table A.2 for Matlab. Regardless of a programming language, the DP method has always reached the same solution. Therefore, the detailed DP results in Table A.1 and Table A.2 correspond to all executions stated in Table 7.2.

In both Tables A.1 and A.2, the total flow allocated to the first MUX is 73,060 B and to the second MUX is 73,060 B. Therefore, the mutual comparison reaches $73,060/73,060 = 100.00\%$. To conclude, DP prepared the best possible LB for the Test Case 1.

It remains to compare C++ and Matlab executions with respect to the computational time. The C++ executions are slightly faster than the executions in Matlab, however, the difference is quite low.

7.2.2 Greedy Heuristic

Based on the greedy principle, the GH approach tends to solutions with the higher error, i.e., solutions might be less precise in terms of LB.

Execution	**C++**		**Matlab**	
	Error	**Time [μs]**	**Error**	**Time [μs]**
1	107.48	1	107.48	70
2	107.48	1	107.48	67
3	107.48	1	107.48	38
4	107.48	1	107.48	39
5	107.48	1	107.48	41
6	107.48	2	107.48	65
7	107.48	1	107.48	51
8	107.48	1	107.48	38
9	107.48	1	107.48	33
10	107.48	1	107.48	34

Table 7.3: Results for the Test Case 1 using GH in each execution.

In general, GH does not intend to find the best solution, but it terminates in a

reasonable number of steps. Finding an optimal solution to such a complex problem – like the LB problem – typically requires unreasonably many steps. It might even happen GH terminates and produces the *unique worst possible solution*.

Greedy algorithms mostly (but not always) fail to find a globally optimal solution, because they usually do not operate exhaustively on all the data. However for the LB problem, a probability of such a situation is significantly reduced by the strictly given range from 0 B to 10 kB for each flow. In other words, there can not be one flow being several times higher than the others which would potentially tend to the *unique worst possible solution*.

On the other hand, GH usually requires only few steps to terminate and therefore, it requires a significantly smaller amount of the computational time.

In Table 7.3, the results produced in C++ and Matlab for the Test Case 1 using GH in each execution are given. The error is the same in each execution, since GH does not use any stochastic approach and is strictly deterministic. The error is equal to 107.48 which is higher in comparison with the DP approach where a global optimum has been reached.

The detailed GH results containing the best flow allocation for the Test Case 1 are given in Appendix A in Table A.3 for C++ and in Table A.4 for Matlab. Regardless of a programming language, the GH method has always reached the same solution. Therefore, the detailed GH results in Table A.3 and Table A.4 correspond to all executions stated in Table 7.3.

In both Tables A.3 and A.4, the total flow allocated to the first MUX is 73,136 B and to the second MUX is 72,984 B. Therefore, the mutual comparison reaches $72,984/73,136 \approx 99.79\%$. To conclude, GH is less precise than the DP approach, however, the obtained GH precision is still remarkable.

Moreover, the GH computational time is worth of mentioning. To gain a solution being not so far away from a global one, it requires only few microseconds. This observation offers a good candidate for a real-time LB solver where the low computational time is crucial. It remains to compare C++ and Matlab executions with respect to the computational time. The C++ executions are significantly faster than the executions in Matlab.

7.2.3 Integer Linear Programming

In Table 7.4, the results produced in C++ and Matlab for the Test Case 1 using ILP in each execution are stated. The error is equal to 0 giving a global optimum in each execution.

The detailed ILP results containing the best flow allocation for the Test Case 1 are given in Appendix A in Table A.5 for C++ and in Table A.6 for Matlab. However, the solutions might differ, since the implementations in both programming languages use the intern built-up ILP solving libraries. Therefore, the detailed ILP results in Table A.5 correspond to all C++ executions stated in Table 7.4 and the detailed ILP results in Table A.6 correspond to all Matlab executions stated in Table 7.4.

In both Tables A.5 and A.6, the total flow allocated to the first MUX is 73,060 B and to the second MUX is 73,060 B. Therefore, the mutual comparison reaches $73,060/73,060 \approx 100.00\%$. To conclude, ILP prepared the best possible LB for the Test Case 1.

As the ILP approach is proposed, ILP is not based on a stochastic problem solving. Therefore, every ILP execution always runs in the same way for the C++ implementation and the Matlab implementation, however, both are different from each other (a standard MATLAB function `intlinprog` and the COIN-OR project written in C++ based both on branch-and-cut method are used).

At first glance, the ILP method seems to be very efficient, since the computational time is remarkably low, especially in Matlab. Nevertheless, larger test cases – the Test Case 2 and Test Case 3 – show how poor performance the ILP approach actually might have, and even the ILP approach is not often capable to find a feasible solution at all.

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	0.00	2,625	0.00	705
2	0.00	2,351	0.00	703
3	0.00	2,252	0.00	709
4	0.00	2,360	0.00	700
5	0.00	2,485	0.00	709
6	0.00	2,405	0.00	705
7	0.00	2,344	0.00	696
8	0.00	2,326	0.00	702
9	0.00	2,341	0.00	725
10	0.00	2,317	0.00	699

Table 7.4: Results for the Test Case 1 using ILP in each execution.

7.2.4 Modified Differential Evolution

The proposed MDE algorithm is partially stochastic and hence, it might produce different solutions in every execution. Nevertheless, the goal remains still the same, i.e., to reach as the lowest LB error as possible and thus, to perform proper LB. The parameters used for the MDE algorithm to solve the Test Case 1 are given in Table 7.5.

Parameter	N_p	s_max	c_1	c_2	k_1	k_2	T_0	α
Value	50	20,000	0.6	0.4	0.3	0.1	1	0.7

Table 7.5: Parameters used for the MDE.

In Table 7.6, the results produced in C++ and Matlab for the Test Case 1 using the MDE in each execution are stated. The error is equal to 0 giving a global optimum in

Execution	C++		Matlab	
	Error	Time [ms]	Error	Time [ms]
1	0.00	96	0.00	3,540
2	0.00	154	0.00	692
3	0.00	60	0.00	10,115
4	0.00	47	0.00	267
5	0.00	89	0.00	2,545
6	0.00	40	0.00	6,821
7	0.00	184	0.00	2,301
8	0.00	55	0.00	3,941
9	0.00	172	0.00	1,349
10	0.00	99	0.00	1,426

Table 7.6: Results for the Test Case 1 using the MDE in each execution.

each execution, however, the solutions might be different – multiple global optima are possible. The detailed MDE results containing the best flow allocation for the Test Case 1 are given in Appendix A in Table A.7 for C++ corresponding to Execution 6 and in Table A.8 for Matlab corresponding to Execution 4.

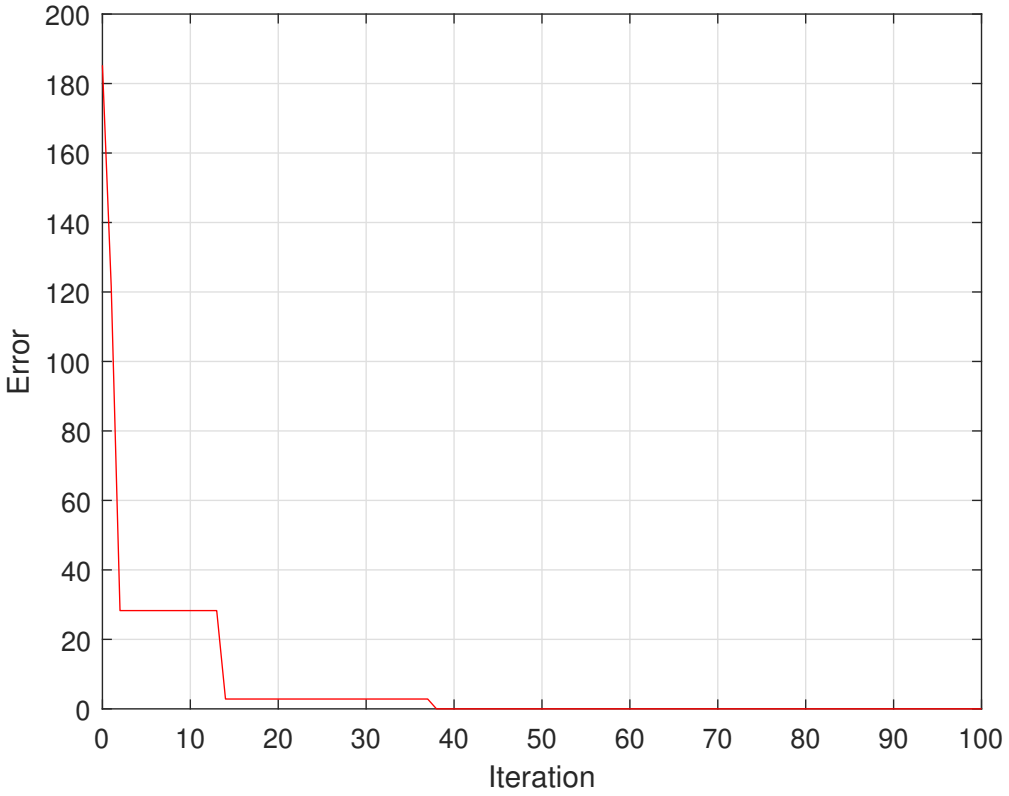

Figure 7.1: The evolution process of the best Test Case 1 flow allocation based on the MDE produced in Matlab corresponding to Execution 4.

In both Tables A.7 and A.8, the total flow allocated to the first MUX is 73,060 B

and to the second MUX is 73,060 B. Therefore, the mutual comparison reaches $73,060/73,060 = 100.00\%$. To conclude, MDE prepared the best possible LB for the Test Case 1.

The computational time is very low for the C++ implementation. The implementation makes a difference. The methods DP and GH being so far examined on the Test Case 1 are rather based on operations with matrices where Matlab is well-optimized, and no copying of memory is necessary. Therefore, the Matlab implementations of DP and GH methods being so far examined on the Test Case 1 are not so much slower than the C++ implementations.

On the contrary, the MDE and RL algorithms require more frequent copying of memory. Thus, it opens up a new perspective for a pointers usage and fast memory access in C++.

In order to demonstrate how well-designed the MDE algorithm is proposed, the evolution process of the best Test Case 1 flow allocation produced in Matlab corresponding to Execution 4 is shown in Figure 7.1.

At the beginning of the evolution process, it shows a fast convergence of the proposed MDE algorithm. A global optimum is reached after 35 iterations and the error is equal to 0.

7.2.5 Reinforcement Learning

The last results of the Test Case 1 remaining to be discussed are produced by the RL approach. As the only one representative of the machine learning algorithm class, RL offers quite a different point of view. RL is based on taking suitable action to maximize reward in a particular situation. The parameters used for the RL approach to solve the Test Case 1 are given in Table 7.7.

Parameter	$s_{\max}$	α	γ	ε
Value	100,000	0.1	1	1

Table 7.7: Parameters used for RL.

Unlike the supervised learning, where the training data set has the answer key so the model is trained with the correct answer itself, RL has no training data set in advance. In RL, there is no answer provided together with the model, however, the agent decides what to do to perform the given task. In the absence of training data set, it is bound to learn from its experience.

In Table 7.8, the results produced in C++ and Matlab for the Test Case 1 using RL in each execution are given. Every execution gives a unique solution, since RL is a strongly stochastic approach.

The detailed RL results containing the best flow allocation for the Test Case 1 are stated in Appendix A in Table A.9 for C++ corresponding to Execution 6 and in Table A.10 for Matlab corresponding to Execution 10.

Execution	C++		Matlab	
	Error	Time [ms]	Error	Time [ms]
1	4.24	755	1.41	12,410
2	7.07	747	1.41	12,499
3	0.00	752	7.07	12,452
4	4.24	760	7.07	12,469
5	7.07	749	7.07	12,421
6	0.00	747	1.41	12,422
7	7.07	752	7.07	12.445
8	7.07	756	4.24	12,483
9	7.07	757	7.07	12,445
10	7.07	756	0.00	12,538

Table 7.8: Results for the Test Case 1 using RL in each execution.

In both Tables A.9 and A.10, the total flow allocated to the first MUX is 73,060 B and to the second MUX is 73,060 B. Therefore, the mutual comparison reaches $73,060/73,060 = 100.00\%$. To conclude, RL prepared the best possible LB for the Test Case 1.

However, it has to be taken into account that only two out of ten executions reaches the error equal to 0 in C++ and one out of ten in Matlab implementation. It depends how $s_{\max}$ is set and such a setting requires experience of an executor. On the other hand, even all the other executions have the error close to 0.

The C++ computational time is several times lower than Matlab needs to calculate an optimal solution. The high computational time for Matlab comes from a type of operation required in the optimization process.

In RL, main mathematical operation are performed on submatices of matrices, e.g., a sum of elements, a minimum or maximum from elements of submatrix.

In C++, a usage of pointers is very efficient way how to implement such mathematical operations. Unfortunately, there are no pointers in Matlab and therefore, the submatrix must be always copied to perform a particular mathematical operation.

7.3 Test Case 2 – Formulation

In this section, the Test Case 2 of the LB problem is introduced and investigated. It consists of $m = 6$ MUXes with $p = 15$ ingoing ports each and therefore, in terms of the number of MUXes, it corresponds to the iFDAQ setup used in the Run 2016, 2017 and 2018.

In Table 7.9, the flows used in the Test Case 2 are stated. There are $n = m \cdot p = 6 \cdot 15 = 90$ flows with values randomly generated in the range from 0 B to 10 kB.

Flow	[B]	Flow	[B]	Flow	[B]
f_1	668	f_{31}	9,136	f_{61}	8,159
f_2	7,870	f_{32}	2,783	f_{62}	9,293
f_3	5,280	f_{33}	9,037	f_{63}	5,795
f_4	4,988	f_{34}	9,894	f_{64}	9,692
f_5	8,109	f_{35}	9,854	f_{65}	3,429
f_6	2,825	f_{36}	3,235	f_{66}	4,021
f_7	8,998	f_{37}	5,743	f_{67}	7,589
f_8	6,853	f_{38}	7,950	f_{68}	3,579
f_9	7,151	f_{39}	4,067	f_{69}	2,341
f_{10}	714	f_{40}	6,721	f_{70}	425
f_{11}	3,501	f_{41}	3,097	f_{71}	4,370
f_{12}	9,164	f_{42}	3,876	f_{72}	9,034
f_{13}	1,757	f_{43}	8,043	f_{73}	7,278
f_{14}	5,267	f_{44}	2,354	f_{74}	5,637
f_{15}	5,767	f_{45}	5,015	f_{75}	6,888
f_{16}	1,485	f_{46}	7,160	f_{76}	2,345
f_{17}	6,757	f_{47}	4,619	f_{77}	8,753
f_{18}	6,429	f_{48}	2,743	f_{78}	4,977
f_{19}	7,647	f_{49}	5,318	f_{79}	3,606
f_{20}	8,204	f_{50}	6,399	f_{80}	9,868
f_{21}	4,149	f_{51}	2,806	f_{81}	7,468
f_{22}	7,720	f_{52}	1,422	f_{82}	2,028
f_{23}	5,028	f_{53}	6,683	f_{83}	1,710
f_{24}	7,313	f_{54}	8,703	f_{84}	1,241
f_{25}	8,007	f_{55}	854	f_{85}	5,921
f_{26}	7,766	f_{56}	9,920	f_{86}	64
f_{27}	4,504	f_{57}	710	f_{87}	4,079
f_{28}	6,632	f_{58}	5,745	f_{88}	4,727
f_{29}	7,065	f_{59}	9,333	f_{89}	3,626
f_{30}	8,789	f_{60}	4,753	f_{90}	2,636

Table 7.9: Flows in the Test Case 2.

7.4 Test Case 2 – Results

In the following subsections, the results related to the Test Case 2 are given. The Test Case 2 is solved by DP, GH, ILP, MDE and RL, respectively. All the listed solution methods are implemented in C++ and Matlab. Finally, the results are investigated with respect to the error (see Equation 6.1) and computation time.

Taking into account the sum of all flows in the Test Case 2, the best possible LB can be acquired with the total flow allocation of 82,494 B for one MUX and 82,493 B for the remaining five MUXes each. Therefore, the best possible error is equal approximately to 2.24 based on Equation 6.1.

7.4.1 Dynamic Programming

Taking into account a determinism of the DP approach, all executions are following the same computational process and thus, DP always generates the same best solution. Moreover, DP produces solutions at the best possible error level.

In Table 7.10, the results produced in C++ and Matlab for the Test Case 2 using DP in each execution are given. The error is equal approximately to 2.24 in all executions regardless of a programming language. A global optimum has been reached, since it is not possible to acquire a lower value of the LB objective function.

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	2.24	10,997	2.24	17,299
2	2.24	10,880	2.24	17,149
3	2.24	10,758	2.24	17,078
4	2.24	10,811	2.24	17,109
5	2.24	10,884	2.24	17,156
6	2.24	10,764	2.24	17,128
7	2.24	10,737	2.24	17,178
8	2.24	10,714	2.24	17,132
9	2.24	10,784	2.24	17,068
10	2.24	10,724	2.24	17,141

Table 7.10: Results for the Test Case 2 using DP in each execution.

The detailed DP results containing the best flow allocation for the Test Case 2 are given in Appendix B in Table B.1 for C++ and in Table B.2 for Matlab. Regardless of a programming language, the DP method has always reached the same solution. Therefore, the detailed DP results in Table B.1 and Table B.2 correspond to all executions stated in Table 7.10.

In both Tables B.1 and B.2, the total flow allocated to the first MUX is 82,494 B and to the remaining five MUXes is 82,493 B each. Therefore, the mutual comparison reaches $82,493/82,494 \approx 100.00\%$. To conclude, DP provided the best possible LB for the Test Case 2.

It remains to discuss the Test Case 2 from the computational time point of view. The C++ executions are significantly faster than the executions in Matlab. Though, a comparison of the computational time needed for the Test Case 1 and Test Case 2 requires a deeper insight. There is a significant increase in the computational time for the Test Case 2 related to the number of MUXes. A size of the Test Case 2 causes an increase of the computational time eventually leading to an elimination of DP as a real-time LB solver.

On the other hand, if no on-the-fly LB adjustment is demanded, a very price LB solution might be sometimes useful for the long-term LB setup.

7.4.2 Greedy Heuristic

In Table 7.11, the results produced in C++ and Matlab for the Test Case 2 using GH in each execution are given. The error is the same in each execution, since GH does not use any stochastic approach and is strictly deterministic. The error is equal approximately to 243.89 being higher in comparison with the DP approach where a global optimum has been reached.

The detailed GH results containing the best flow allocation for the Test Case 2 are given in Appendix B in Table B.3 for C++ and in Table B.4 for Matlab. Regardless of a programming language, the GH method has always reached the same solution. Therefore, the detailed GH results in Table B.3 and Table B.4 correspond to all executions stated in Table 7.11.

Based on Tables B.3 and B.4, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes might be performed. The fourth MUX has the lowest total flow allocation with a value of 82,323 B and the fifth MUX has the highest total flow allocation with a value of 82,640 B. The mutual comparison reaches $82,323/82,640 \approx 99.62\%$. To conclude, GH is less precise than the DP approach being in line with the same observation in the Test Case 1.

Execution	**C++**		**Matlab**	
	Error	**Time [μs]**	**Error**	**Time [μs]**
1	243.89	6	243.89	2,577
2	243.89	6	243.89	140
3	243.89	6	243.89	140
4	243.89	6	243.89	141
5	243.89	7	243.89	386
6	243.89	7	243.89	103
7	243.89	6	243.89	109
8	243.89	7	243.89	108
9	243.89	7	243.89	108
10	243.89	6	243.89	114

Table 7.11: Results for the Test Case 2 using GH in each execution.

With a reference to the Test Case 1, the GH computational time is again worth of mentioning. GH is capable to keep the computational time needed for the Test Case 2 at the same level as for the Test Case 1. In order to gain a solution not being so far away from a global one, it requires only few microseconds. It shows that a choose of the GH approach as a real-time LB solver might be a meaningful decision.

It remains to compare the computational time of the C++ and Matlab executions with each other. The C++ executions are significantly faster than the executions in Matlab. It corresponds with an efficient way how C++ uses its fast memory access.

7.4.3 Integer Linear Programming

It was already indicated in the Test Case 1, ILP might have a problem to find even a feasible solution for larger test cases in the reasonable computational time. In fact, based on the ILP model for LB as it is described in Subsection 6.5.4, ILP is not capable to find a feasible solution for the Test Case 2 at all.

Therefore, it is necessary to adjust slightly the ILP model for LB to be less strictly. The only one possibility is to relax the *MUX flow limit* condition described by Equation 6.40. Unfortunately, it is not possible to change a form of other two conditions – Equation 6.41 and Equation 6.42 – since it would lead to a violation how the LB problem is formulated.

As a result, the adjusted condition has the following relaxation:

$$\sum_{j=1}^{n} f_j x_{ij} \leq F + 10 \quad \forall i = 1, \ldots, m. \tag{7.1}$$

Then, considering the adjusted condition resulting in the less strictly ILP model of LB, ILP is able to find a feasible solution and even a global optimum of the Test Case 2. In Table 7.12, the results produced in C++ and Matlab for the Test Case 2 using ILP in each execution are shown.

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	21.19	18,558	23.81	1,545
2	21.19	17,819	23.81	1,541
3	21.19	17,251	23.81	1,484
4	21.19	17,205	23.81	1,473
5	21.19	17,424	23.81	1,487
6	21.19	17,266	23.81	1,465
7	21.19	17,138	23.81	1,514
8	21.19	17,154	23.81	1,498
9	21.19	17,484	23.81	1,480
10	21.19	17,127	23.81	1,466

Table 7.12: Results for the Test Case 2 using ILP in each execution.

The detailed ILP results containing the best flow allocation for the Test Case 2 are given in Appendix B in Table B.5 for C++ and in Table B.6 for Matlab. However, the solutions might differ, since the implementations in both programming languages use the intern built-up ILP solving libraries. Though, based on the ILP determinism, every ILP execution always runs in the same way for the C++ approach and the Matlab approach, however, both are different to each other.

Therefore, the detailed ILP results in Table B.5 correspond to all C++ executions stated in Table 7.12 and the detailed ILP results in Table B.6 correspond to all Matlab executions stated in Table 7.12.

Based on Table B.5, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by C++ might be performed. The sixth MUX has the lowest total flow allocation with a value of 82,481 B and the fourth MUX has the highest total flow allocation with a value of 82,504 B. The mutual comparison reaches $82,481/82,504 \approx 99.97\%$.

Based on Table B.6, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by Matlab might be performed. The fifth MUX has the lowest total flow allocation with a value of 82,479 B and the second MUX has the highest total flow allocation with a value of 82,504 B. The mutual comparison reaches $82,479/82,504 \approx 99.97\%$.

Besides different implementations in both programming languages using the intern built-up ILP solving libraries, the fact that the C++ and Matlab implementations find different global optima for the same ILP model of LB might be also caused by the adjustment of `TolInteger` option in Matlab. A value of 0.001 is used for the `TolInteger` option.

The computational time is quite low in Matlab, however, that might be an exception. The Test Case 3 shows how long it may take to find a solution of the LB problem using ILP. In addition, taking into account the higher error, ILP is not a suitable approach for the LB problem solving.

7.4.4 Modified Differential Evolution

The proposed MDE algorithm is partially stochastic and hence, it might produce different solutions in every execution. The parameters used for the MDE algorithm to solve the Test Case 2 are given in Table 7.5. The presented results in the Table 7.13 have been published in [63].

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	2.24	1,601	2.24	71,413
2	2.24	1,652	2.24	25,558
3	2.24	1,997	2.24	23,688
4	2.24	2,220	2.24	25,378
5	2.24	1,382	2.24	43,370
6	2.24	2,099	2.24	90,298
7	2.24	862	2.24	78,666
8	2.24	1,901	2.24	71,448
9	2.24	2,379	2.24	57,260
10	2.24	1,803	2.24	28,210

Table 7.13: Results for the Test Case 2 using the MDE in each execution.

In Table 7.13, the results produced in C++ and Matlab for the Test Case 2 using the MDE in each execution are stated. The error is equal approximately to 2.24 giving a global optimum in each execution, however, the solutions might be different

– multiple global optima are possible. The detailed MDE results containing the best flow allocation for Test Case 2 are given in Appendix B in Table B.7 for C++ corresponding to Execution 7 and in Table B.8 for Matlab corresponding to Execution 3.

In both Tables B.7 and B.8, the total flow allocated to the first MUX is 82,494 B and to the remaining five MUXes is 82,493 B each. Therefore, the mutual comparison reaches $82,493/82,494 \approx 100.00\%$. To conclude, the MDE prepared the best possible LB for the Test Case 2.

The computational time is very low in C++ due to the fast memory access provided by C++ and the usage of pointers trying to avoid any copying of memory. Hence, the MDE algorithm represents a good candidate for a real-time LB solver.

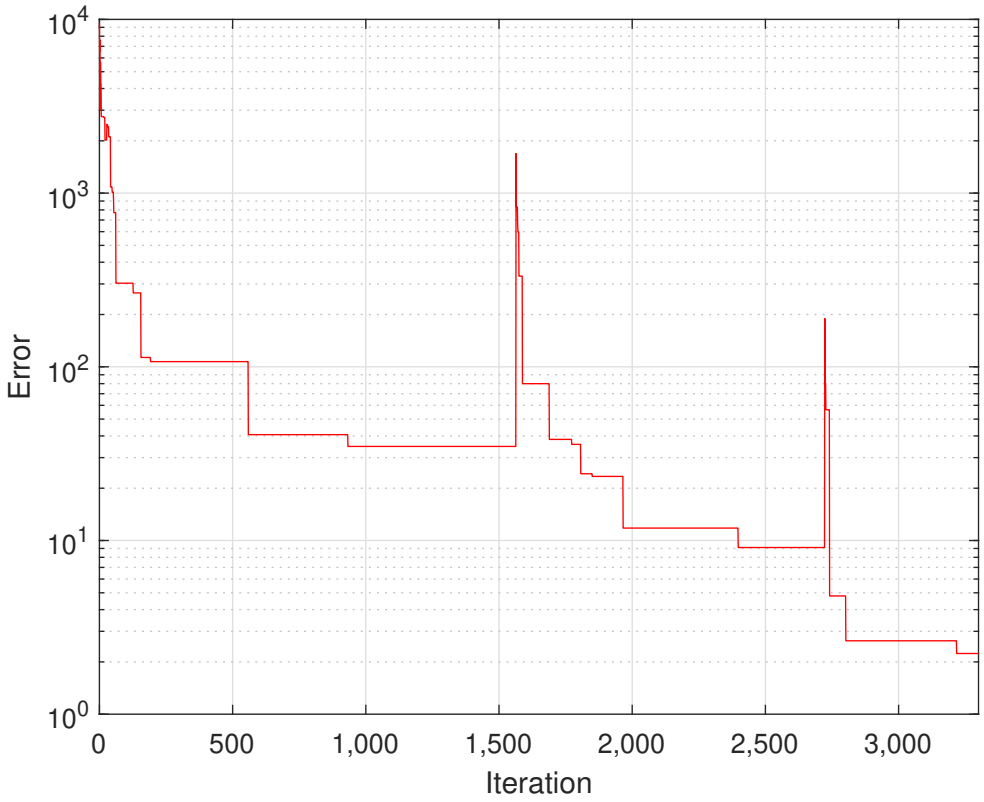

Figure 7.2: The evolution process of the best Test Case 2 flow allocation based on the MDE produced in Matlab corresponding to Execution 3.

To demonstrate the behaviour of the MDE algorithm, the evolution process of the best Test Case 2 flow allocation based on the MDE produced in Matlab corresponding to Execution 3 is shown in Figure 7.2. At the beginning of the evolution process, it shows a fast convergence of the proposed MDE algorithm. It reached a solution close to a global optimum after just 150 iterations.

Nevertheless, there are several short-term deteriorations of the error in evolution which are caused by selection mechanism based on SA. The selection mechanism sometimes selects individuals with a worse *fitness* value. They have chance to show

their potential to produce a new population. However, this feature weakens at the end of the evolution process.

Finally, a global optimum is reached after 3,400 iterations and the error is equal approximately to 2.24.

7.4.5 Reinforcement Learning

In this subsection, the results produced by the RL approach related to the Test Case 2 are given. The proposed RL algorithm is strongly stochastic and hence, it produces a unique solution in every execution. The parameters used for the RL algorithm to solve the Test Case 2 are given in Table 7.7. The quality of the results depends on $s_{\max}$ directly controlling how well state–action pairs are learnt represented by Q values. Therefore, the $s_{\max}$ setup requires experience of an executor for the learning phase.

In Table 7.14, the results produced in C++ and Matlab for the Test Case 2 using RL in each execution are stated. The error is equal approximately to 10.44 giving a solution close to a global optimum in each execution.

The presented results in the Table 7.14 have been published in [67]. However, the C++ executions in the Table 7.14 are several times faster than C++ results published in [67]. After the paper publication, the C++ implementation has been enhanced in terms of an efficient memory allocation and deallocation.

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	10.44	8,180	10.44	117,253
2	10.44	8,073	10.44	117,252
3	10.44	8,093	10.44	117,106
4	10.44	8,125	10.44	119,307
5	10.44	8,104	10.44	119,113
6	10.44	8,114	10.44	117,690
7	10.44	8,052	10.44	118,236
8	7.14	8,073	10.44	118,427
9	10.82	8,095	10.44	118,242
10	9.00	8,085	10.82	117,918

Table 7.14: Results for the Test Case 2 using RL in each execution.

The detailed RL results containing the best Test Case 2 flow allocation are stated in Appendix B in Table B.9 for C++ corresponding to Execution 8 and in Table B.10 for Matlab corresponding to Execution 3.

Based on Table B.9, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by C++ might be performed. The second MUX has the lowest total flow allocation with value of 82,489 B and

on the other hand, the sixth has the highest total flow allocation with a value of 82,497 B. The ratio of the flows is $82,489/82,497 \approx 99.99\%$.

Based on Table B.10, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by Matlab might be performed. The third MUX has the lowest total flow allocation with value of 82,486 B and on the other hand, the fifth and sixth MUX have the highest total flow allocation with a value of 82,498 B each. The ratio of the flows is $82,486/82,498 \approx 99.99\%$.

In order to calculate an optimal solution, Matlab consumes several times more computational time than a version implemented in C++. The high computational time for Matlab comes from a type of operation required in an optimization process. In RL, the main mathematical operations are performed on submatices of matrices, e.g., sum of elements, minimum or maximum from elements of a submatrix. In C++, pointers are a very efficient way how to implement such mathematical operations. Since pointers are absent in Matlab, a submatrix must be always copied to perform a particular mathematical operation.

However, the computational time for both C++ and Matlab is quite high, resulting in an exclusion of the RL algorithm as a real-time LB solver. On the other hand, the error is quite small. Therefore, the RL approach can be considered for the long-term LB setup where no frequent changes in the flows are expected.

On the other hand, faster and more precise algorithms have already been mentioned, e.g., DP and MDE, and thus, there is no meaningful reason for giving a priority to RL.

In addition, the RL approach might lead to a quite high RAM memory consumption during execution to store values of each state, since the problems can be quite complex.

7.5 Test Case 3 – Formulation

In this section, the Test Case 3 of the LB problem is introduced and investigated. It consists of $m = 8$ MUXes with $p = 15$ ingoing ports each and therefore, in terms of the number of MUXes, it corresponds to the iFDAQ full setup. However, the iFDAQ full setup has never been in operation for the COMPASS experiment, since it was not required by any physics program.

In Table 7.15, the flows used in the Test Case 3 are stated. There are $n = m \cdot p = 8 \cdot 15 = 120$ flows with values randomly generated in the range from 0 B to 10 kB.

7.6 Test Case 3 – Results

In the following subsections, the results related to the Test Case 3 are given. The Test Case 3 is solved by DP, GH, ILP, MDE and RL, respectively. All the listed

solution methods are implemented in C++ and Matlab. Finally, the results are investigated with respect to the error (see Equation 6.1) and computation time.

Taking into consideration the sum of all flows in Test Case 3, the best possible LB can be reached with the total flow allocation of 68,401 B for four MUXes each and 68,400 B for the remaining four MUXes each. Therefore, the best possible error is equal to 2 based on Equation 6.1.

Flow	[B]	Flow	[B]	Flow	[B]	Flow	[B]
f_1	9,056	f_{31}	7,346	f_{61}	6,618	f_{91}	1,182
f_2	582	f_{32}	6,000	f_{62}	815	f_{92}	834
f_3	4,409	f_{33}	249	f_{63}	5,325	f_{93}	5,374
f_4	3,140	f_{34}	458	f_{64}	4,428	f_{94}	245
f_5	5,035	f_{35}	5,194	f_{65}	2,400	f_{95}	8,219
f_6	3,906	f_{36}	7,196	f_{66}	8,104	f_{96}	7,350
f_7	2,149	f_{37}	667	f_{67}	739	f_{97}	3,354
f_8	2,266	f_{38}	2,519	f_{68}	2,772	f_{98}	2,485
f_9	4,150	f_{39}	3,066	f_{69}	4,335	f_{99}	6,642
f_{10}	6,473	f_{40}	9,404	f_{70}	9,233	f_{100}	5,575
f_{11}	6,533	f_{41}	2,599	f_{71}	3,059	f_{101}	1,421
f_{12}	7,415	f_{42}	2,946	f_{72}	9,179	f_{102}	398
f_{13}	146	f_{43}	7,434	f_{73}	3,547	f_{103}	5,577
f_{14}	5,115	f_{44}	6,401	f_{74}	690	f_{104}	3,741
f_{15}	7,845	f_{45}	151	f_{75}	8,215	f_{105}	3,043
f_{16}	380	f_{46}	8,114	f_{76}	4,673	f_{106}	4,063
f_{17}	4,718	f_{47}	8,088	f_{77}	5,523	f_{107}	8,303
f_{18}	8,525	f_{48}	7,084	f_{78}	5,343	f_{108}	6,451
f_{19}	3,685	f_{49}	6,835	f_{79}	4,646	f_{109}	6,100
f_{20}	2,324	f_{50}	1,888	f_{80}	7,832	f_{110}	9,792
f_{21}	7,464	f_{51}	5,232	f_{81}	7,857	f_{111}	1,046
f_{22}	3,629	f_{52}	431	f_{82}	5,433	f_{112}	4,540
f_{23}	1,091	f_{53}	1,929	f_{83}	5,549	f_{113}	4,533
f_{24}	3,684	f_{54}	2,784	f_{84}	4,278	f_{114}	1,408
f_{25}	7,432	f_{55}	9,259	f_{85}	3,020	f_{115}	391
f_{26}	7,223	f_{56}	2,438	f_{86}	520	f_{116}	9,104
f_{27}	2,888	f_{57}	925	f_{87}	6,314	f_{117}	7,525
f_{28}	1,360	f_{58}	8,885	f_{88}	646	f_{118}	8,968
f_{29}	2,795	f_{59}	7,258	f_{89}	7,716	f_{119}	8,360
f_{30}	3,165	f_{60}	2,841	f_{90}	5,676	f_{120}	2,491

Table 7.15: Flows in the Test Case 3.

7.6.1 Dynamic Programming

The DP determinism can be illustrated in Table 7.16 where the results produced in C++ and Matlab for the Test Case 3 using DP in each execution are stated. The

error is equal to 2 in all executions regardless of a programming language.

The certainty that the DP approach eventually finds a global optimum is absolutely crucial for the long-term LB setup where no on-the-fly changes are required.

Execution	C++		Matlab	
	Error	Time [ms]	Error	Time [ms]
1	2.00	15,341	2.00	25,161
2	2.00	15,362	2.00	24,738
3	2.00	15,344	2.00	25,027
4	2.00	15,526	2.00	24,771
5	2.00	15,557	2.00	24,782
6	2.00	15,310	2.00	24,768
7	2.00	15,336	2.00	24,729
8	2.00	15,507	2.00	24,790
9	2.00	15,455	2.00	24,880
10	2.00	15,350	2.00	24,831

Table 7.16: Results for the Test Case 3 using DP in each execution.

The detailed DP results containing the best flow allocation for the Test Case 3 are given in Appendix C in Table C.1 for C++ and in Table C.2 for Matlab. Regardless of a programming language, the DP method has always reached the same solution. Therefore, the detailed DP results in Table C.1 and Table C.2 correspond to all executions stated in Table 7.16.

In both Tables C.1 and C.2, the total flow allocated to the first four MUXes is 68,401 B each and to the remaining four MUXes is 68,400 B each. Therefore, the mutual comparison reaches $68{,}400/68{,}401 \approx 100.00\%$. To conclude, DP prepared the best possible LB for the Test Case 3.

It remains to discuss the Test Case 3 with respect to the computation time. The C++ executions are significantly faster than the executions in Matlab.

Based on the Test Case 1, Test Case 2 and Test Case 3, the DP approach is significant for the long-term LB setup planning, since a guarantee of providing the best possible solution is priceless. On the other hand, DP is not suitable for the real-time LB adjustments because of the high computational time being necessary for a solution searching.

7.6.2 Greedy Heuristic

It was already mentioned, GH is probably the best candidate for a real-time LB solver and the results of the Test Case 3 confirm this statement.

In Table 7.17, the results produced in C++ and Matlab for the Test Case 3 using GH in each execution are stated. The error is the same in each execution, since GH does not use any stochastic approach and is strictly deterministic. The error is

Execution	C++		Matlab	
	Error	Time [μs]	Error	Time [μs]
1	224.22	9	224.22	3,053
2	224.22	10	224.22	184
3	224.22	10	224.22	183
4	224.22	10	224.22	182
5	224.22	10	224.22	427
6	224.22	10	224.22	136
7	224.22	10	224.22	137
8	224.22	10	224.22	164
9	224.22	9	224.22	286
10	224.22	10	224.22	150

Table 7.17: Results for the Test Case 3 using GH in each execution.

equal approximately to 224.22 which is higher in comparison with the DP approach where a global optimum has been reached.

The detailed GH results containing the best flow allocation for the Test Case 3 are given in Appendix C in Table C.3 for C++ and in Table C.4 for Matlab. Regardless of a programming language, the GH method has always reached the same solution. Therefore, the detailed GH results in Table C.3 and Table C.4 correspond to all executions stated in Table 7.17.

Based on Tables C.3 and C.4, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes might be performed. The seventh MUX has the lowest total flow allocation with a value of 68,285 B and the fourth MUX has the highest total flow allocation with a value of 68,488 B. The mutual comparison reaches $68,285/68,488 \approx 99.70\%$. The error persists still at the same level as for the Test Case 1 and Test Case 2.

In order to acquire a solution not being so far away from a global one, it requires only few microseconds. Thus, GH is the best real-time solver of the LB problem.

7.6.3 Integer Linear Programming

The Test Case 3 verifies how difficult is to find a solution of the LB problem for ILP. ILP usually has no problems with small test cases like the Test Case 1 where only two MUXes are considered. As soon as test cases become larger, ILP has a problem to provide an optimal solution of the LB problem.

It was already discussed in the Test Case 1 and Test Case 2, ILP might have problems to find even a feasible solution for larger test cases in the reasonable computational time. In fact, based on the ILP model for LB as it is described in Subsection 6.5.4, ILP is not capable to find a feasible solution for the Test Case 3 at all.

Therefore, by analogy to the Test Case 2, it is again necessary to adjust slightly the ILP model for LB to be less strictly. The relaxation has been already described in

the Test Case 2 and it deals with Equation 6.40.

As a result, the adjusted condition has the following relaxation:

$$\sum_{j=1}^{n} f_j x_{ij} \leq F + 35 \quad \forall i = 1, \ldots, m. \tag{7.2}$$

Then, with the adjusted condition, ILP is able to find a feasible solution and even a global optimum of the Test Case 3 considering the less strictly model. In Table 7.18, the results produced in C++ and Matlab for the Test Case 3 using ILP in each execution are shown.

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	194.43	52,415	134.61	96,635
2	194.43	51,483	134.61	95,542
3	194.43	50,989	134.61	95,630
4	194.43	50,941	134.61	95,556
5	194.43	50,963	134.61	95,268
6	194.43	50,910	134.61	95,714
7	194.43	49,992	134.61	95,596
8	194.43	50,041	134.61	96,168
9	194.43	50,335	134.61	95,251
10	194.43	49,927	134.61	96,264

Table 7.18: Results for the Test Case 3 using ILP in each execution.

The detailed ILP results containing the best flow allocation for the Test Case 3 are given in Appendix C in Table C.5 for C++ and in Table C.6 for Matlab. However, the solutions might differ, since the implementations in both programming languages use the intern built-up ILP solving libraries. Though, based on ILP determinism, every ILP execution always runs in the same way for the C++ approach and the Matlab approach, however, both are different to each other.

Therefore, the detailed ILP results in Table C.5 correspond to all C++ executions stated in Table 7.18 and the detailed ILP results in Table C.6 correspond to all Matlab executions stated in Table 7.18.

Based on Table C.5, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by C++ might be performed. The eighth MUX has the lowest total flow allocation with a value of 68,221 B and the first MUX has the highest total flow allocation with a value of 68,435 B. The mutual comparison reaches $68,221/68,435 \approx 99.69\%$.

Based on Table C.6, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by Matlab might be performed. The fourth MUX has the lowest total flow allocation with a value of 68,282 B and the sixth MUX has the highest total flow allocation with a value of 68,430 B. The mutual comparison reaches $68,282/68,430 \approx 99.78\%$.

As it has been already observed and explained in the Test Case 1 and Test Case 2, the C++ and Matlab implementations find different global optima.

Finally, the computational time is quite high and therefore, ILP is not suitable for the real-time LB solving. To conclude, the high computational time, together with the higher error caused by the relaxation of the model, excludes the ILP approach from any usage in production. Moreover, ILP has significant problems to find a feasible solution and thus, a necessity of the relaxation degrades applicability too.

7.6.4 Modified Differential Evolution

The proposed MDE algorithm is partially stochastic and hence, it might produce different solutions in every execution. The parameters used for the MDE algorithm to solve Test Case 3 are given in Table 7.5. The presented results in the Table 7.19 have been published in [63].

Execution	**C++**		**Matlab**	
	Error	**Time [ms]**	**Error**	**Time [ms]**
1	15.30	6,050	2.00	87,064
2	2.00	2,507	2.00	67,601
3	2.00	1,911	2.00	104,993
4	2.00	3,266	2.00	61,987
5	2.00	2,358	2.00	94,706
6	2.00	2,659	2.00	80,280
7	2.00	5,398	4.24	151,224
8	2.00	3,153	2.00	108,325
9	2.00	2,718	2.00	37,530
10	2.00	5,100	2.00	82,067

Table 7.19: Results for the Test Case 3 using MDE in each execution.

In Table 7.19, the results for the Test Case 3 using MDE in each execution are given for C++ and Matlab. The error is nearly always equal to 2.00 giving a global optimum in each execution. Nevertheless, there are few executions which did not find a global optimum like the others. It is caused by the stopping criterion. Here, the maximum number of iterations is selected as the stopping criterion. In other words, the MDE algorithm would have needed more iterations for some executions to find a global optimum. Thus, fundamental experience with heuristics of an executor is required to adjust all parameters in a right way.

The detailed MDE results containing the best flow allocation for the Test Case 3 are given in Appendix C in Table C.7 for C++ corresponding to Execution 3 and in Table C.8 for Matlab corresponding to Execution 9.

In both Tables C.7 and C.8, the total flow allocated to the first four MUXes is 68,401 B each and to the remaining four MUXes is 68,400 B each. Therefore, the mutual comparison reaches $68,400/68,401 \approx 100.00\%$. To conclude, MDE prepared the best possible LB for the Test Case 3.

The computational time is very low in C++ due to the fast memory access provided by C++ and the usage of pointers trying to avoid any copying of memory. Hence, the MDE algorithm represents a good candidate for a real-time LB solver.

To demonstrate the behaviour of the MDE algorithm, the evolution process of the best Test Case 3 flow allocation produced in Matlab corresponding to Execution 9 is shown in Figure 7.3. At the beginning of the evolution process, the figure shows the fast convergence behavior of the proposed MDE algorithm.

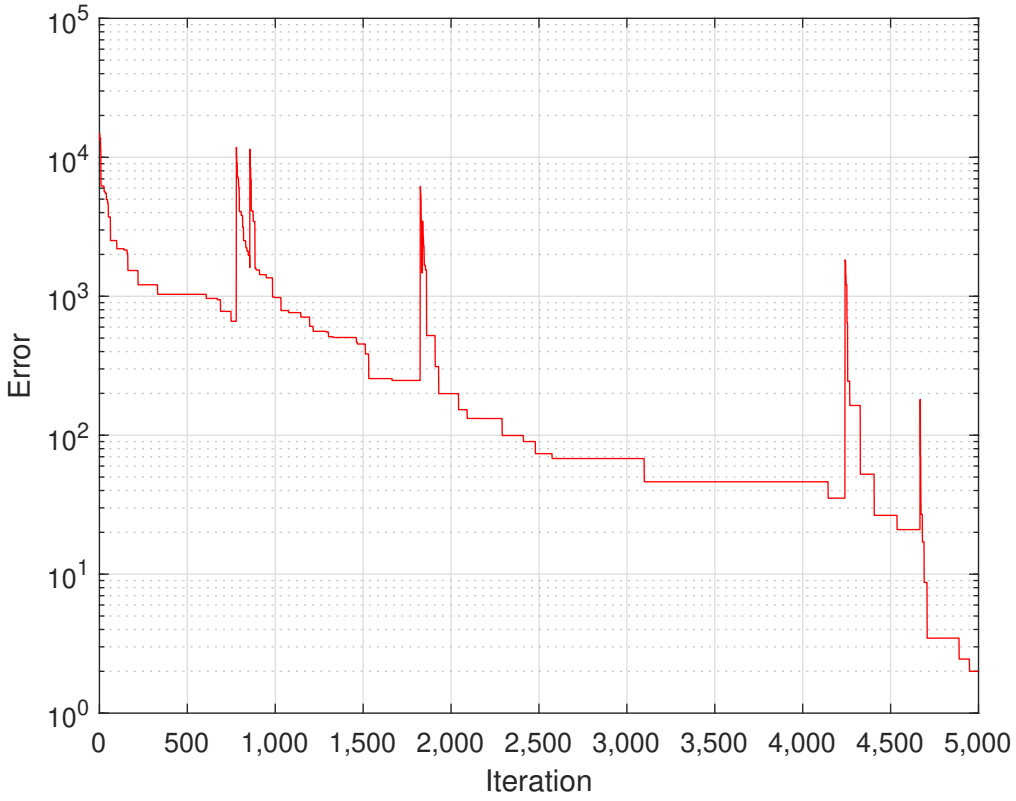

Figure 7.3: The evolution process of the best Test Case 3 flow allocation based on the MDE produced in Matlab corresponding to Execution 9.

Nevertheless, there are many short-term deteriorations of error in evolution which are caused by selection mechanism based on Simulated Annealing (SA). It sometimes selects individuals with a worse *fitness* value. They have chance to show their potential to produce next population. However, this feature weakens at the end of the evolution process. Finally, a global optimum is reached after 4,900 iterations and the error is equal to 2.00.

7.6.5 Reinforcement Learning

The last results of the Test Case 3 to be discussed are produced by the RL approach. In Table 7.20, the results produced in C++ and Matlab for the Test Case 3 using RL in each execution are stated. In almost each execution, a unique solution is

retrieved giving very precise LB with the small error. The parameters used for the RL approach to solve Test Case 3 are given in Table 7.7.

The presented results in the Table 7.20 have been published in [67]. However, the C++ executions in the Table 7.14 are several times faster than C++ results published in [67]. After the paper publication, the C++ implementation has been enhanced in terms of an efficient memory allocation and deallocation.

Execution	C++		Matlab	
	Error	Time [ms]	Error	Time [ms]
1	11.31	14,279	11.66	206,358
2	5.10	14,133	5.29	203,017
3	7.87	13,911	6.48	203,287
4	11.14	14,195	11.22	204,427
5	6.93	14,331	6.32	207,476
6	5.10	14,365	11.22	208,410
7	11.49	13,980	2.45	204,824
8	6.63	14,008	6.16	205,160
9	5.29	13,998	11.49	204,299
10	7.07	14,028	11.14	204,525

Table 7.20: Results for the Test Case 3 using RL in each execution.

The detailed RL results containing the best flow allocation for the Test Case 3 are stated in Appendix C in Table C.9 for C++ corresponding to Execution 2 and in Table C.10 for Matlab corresponding to Execution 7.

Based on Table C.9, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by C++ might be performed. The sixth MUX has the lowest total flow allocation with a value of 68,398 B and the eighth has the highest total flow allocation with a value of 68,404 B. The mutual comparison reaches $68,398/68,404 \approx 99.99\%$.

Based on Table C.10, a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes generated by Matlab might be performed. The fourth MUX has the lowest total flow allocation with a value of 68,399 B and the first, third, fifth, sixth and seventh have the highest total flow allocation with a value of 68,401 B. The mutual comparison reaches $68,399/68,401 \approx 100.00\%$.

All aspects related to RL have already been mentioned in the Test Case 2. More precise and faster algorithms have been proposed – like DP and MDE – for the LB problem solving. Moreover, a consumption of RAM memory needed for RL is quite high.

7.7 Summary based on Numerical Results

Test cases have been discussed in an extensive way. It is useful to put all observations to a single table and draw conclusions based on them.

In Table 7.21, the overview of main features – the accuracy and computational time – is shown. Moreover, there are also the real-time and long-term LB flags indicating which method is suitable for real-time and long-term LB.

Method	Accuracy	Time	Real-time LB	Long-term LB
DP	~ 100.00%	4		**x**
GH	> 99.60%	1	**x**	
ILP	> 99.60%	5		
MDE	~ 100.00%	2	**x**	**x**
RL	~ 100.00%	3		

Table 7.21: Summary based on numerical results.

The accuracy is shown based on a comparison of the lowest total flow allocation and highest total flow allocation for the respective MUXes. Following the results of the C++ implementation, the computational time column orders methods based on the computational time needed in executions where 1 is equal to the fastest method and 5 is equal to the slowest method.

Based on the summary in Table 7.21, DP, the MDE and RL are best candidates for the long-term LB setup solving. However, RL is generally considered to generate results with the higher error than DP and the MDE and thus, RL is excluded. MDE is faster than DP, however, DP has a flag in the long-term LB column too, since it may be considered as an alternative to the faster MDE.

Real-time LB has only two possibilities – GH and the MDE. However, GH is rather being applicable for real-time LB than the MDE, since the computational time of GH takes only few microseconds and does not increase together with the size of a given test case. Moreover, the GH accuracy is still kept above 99.60%.

Conclusion

The iFDAQ was successfully deployed and commissioned in 2014, allowed to successfully take data for nominal Drell-Yan conditions during the Run 2015 and followed by runs dedicated to DVCS in 2016 and 2017, and to Drell-Yan again in 2018.

Firstly, the DIALOG library has been presented. The DIALOG library is a new communication library for iFDAQ of the COMPASS experiment at CERN. It is a replacement for the DIM library. The DIALOG library provides efficient and reliable IPCs across different platforms. Its communication mechanism is based on the publish/subscribe method and allows for asynchronous communications, task parallelism and multiple destination updates. Its characteristics of efficiency and reliability have considerably improved the performance and robustness of the complete iFDAQ. It was fully incorporated to all processes in the Run 2016.

The DIALOG is responsible for basically all communications inside the iFDAQ, in this environment it makes available around 100 services provided by 30 servers.

The DIALOG Online Monitoring API provides an easy use interface for the monitoring of whole communication system. Its general implementation enables a development of various monitoring tools. The DIALOG GUI is found to be very useful not only for monitoring of the system state, but also to determine which services are available at a given time. Furthermore, the DIALOG POST Daemon and the DIALOG WebSockets Daemon establish a connection between any desktop application and any web application. That makes any communication system based on the DIALOG library less dependent in terms of an operating system and environment.

Moreover, the efficiency measurement has shown the performance of the DIALOG library is significantly better than the DIM library. It uses and saturates the network bandwidth in a more efficient way.

Secondly, the DAQ Debugger has been implemented. The DAQ Debugger has been incorporated to all processes of the iFDAQ in August 2016 and since then, it helps with the error detection. It does not affect the process performance and does not increase load on readout engine computers.

Based on crash reports created by the DAQ Debugger, all remaining software issues in the iFDAQ have been identified and fixed. Since October 2017, the iFDAQ is stable and without any single crash. The DAQ Debugger fulfilled initial demands and purpose.

The improved iFDAQ stability gave an opportunity to introduce the continuously

running mode to the iFDAQ. Without any stops, the iFDAQ runs 24/7 regardless of nights, weekends or bank holidays for most of the calendar year. It helped to collect more physics data in the Run 2017 and 2018.

This thesis has introduced the LB problem of the iFDAQ of the COMPASS experiment at CERN. $\mathcal{NP}$-completeness of the LB problem makes optimization more challenging. Five approaches how to deal with such a problem have been proposed.

The first way is based on DP. It is partially inspired by Knapsack problem using DP. Moreover, it introduces a third dimension to keep a proper number of ports in one MUX.

GH is the second approach. The algorithm has been implemented for the LB problem solving and it is partially inspired by greedy algorithm for the Partition problem. It is generalized to $m \in \mathbb{N}$ partitions and it does not consider "full" partitions ($p \in \mathbb{N}$ integers have been already allocated to the partition) anymore.

The next approach is based on ILP. The general form of ILP opened up a new possibility to use some standard solvers for the LB problem. Thus, a standard MATLAB function `intlinprog` and the powerful COIN-OR project written in C++ based both on branch-and-cut method have been used.

The fourth approach is based on a GA. The proposed MDE has new crossover and mutation operator and its selection mechanism is inspired by SA.

Finally, the last approach, RL refers to a kind of machine learning method in which an agent receives a delayed reward in the next time step to evaluate its previous action. The RL algorithm runs in order to learn and retrieve flow allocation for the i-th MUX, move to the $(i+1)$-th MUX, etc. In this manner, it allocates flows to all m MUXes.

Finally, all mentioned approaches solving the LP problem have been evaluated using three test cases – the Test Case 1, Test Case 2 and Test Case 3. Based on the results, the GH approach is the most suitable method for the real-time LB solving due to the small computational time and reasonable error. For the long-term LB setup purpose, DP and MDE match the requirements in terms of the best error and ability to find a global optimum.

Due to iFDAQ versatility and scalability, the iFDAQ is also suitable for other high-energy physics experiments. Recently, it has been chosen by the second experiment at the SPS which is searching for light dark matter – the experiment NA64.

Nowadays, the COMPASS typical data rate is 1500 MB/s during spill which is collected from more than 100 front-end modules. The maximum aggregated throughput of the designed system is 1.5 GB/s, but taking into account accelerator duty cycle and significant local memory resources, it has a safety margin of 200-300% and possibility of future improvements. Thus, it fulfilled initial demands and its development continues.

Acknowledgement

The work on this thesis has been supported in part by the following grants of the Ministry of Education, Youth, and Sports of the Czech Republic: LM2015058, RVO14000, SGS 14/210, and SGS 17/198. It has also being supported by the Maier-Leibnitz-Labor, Garching and the DFG cluster of excellence "Origin and Structure of the Universe", and the CERN Doctoral Student Programme.

I would like to thank to my supervisor, Ing. Tomáš Liška, Ph.D., for his useful remarks, advice and suggestions concerning this thesis. My sincere thanks belongs to doc. Ing. Miroslav Virius, CSc. who offered me an opportunity to participate in the COMPASS experiment at CERN. I also wish to express my gratitude towards M.Sc. Vladimir Frolov and Ing. Josef Nový who kindly introduced me into the Data Acquisition System (DAQ) of the COMPASS experiment.

The work on this thesis would not be possible without continuous support of my family. My thanks belong especially to my mother and sister who provided me a home whenever I was coming home for couple of days from CERN. Last but not least, I would like to thank to my family for their love.

Very special thanks is dedicated to my beloved girlfriend, Anna, who patiently waited for me so many long months I spent at CERN without any objections.

Thank you!

Ing. Ondřej Šubrt

Bibliography

[1] P. Abbon et al. *The COMPASS experiment at CERN.* Nucl. Instrum. Methods Phys. Res. (2007), 455 – 518.

[2] M. Affenzeller, S. Wagner, S. Winkler, and A. Beham. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications.* Taylor & Francis Ltd, first edition, (2018).

[3] V. Y. Alexakhin et al. *COMPASS-II Proposal.* The COMPASS Collaboration, (May 2010). CERN-SPSC-2010-014, SPSC-P-340.

[4] T. Anticic et al. *ALICE DAQ and ECS Users Guide.* CERN, EDMS 616039 (January 2006).

[5] Y. Bai et al. *Overview and Future Developments of the FPGA-based DAQ of COMPASS.* Journal of Instrumentation **11** (2016), C02025.

[6] M. Bodlak et al. *Developing Control and Monitoring Software for the Data Acquisition System of the COMPASS Experiment at CERN.* Acta polytechnica: Scientific Journal of the Czech Technical University in Prague (2013).

[7] M. Bodlak et al. *New data acquisition system for the COMPASS experiment.* Journal of Instrumentation **8** (2013), C02009.

[8] M. Bodlak et al. *FPGA based data acquisition system for COMPASS experiment.* Journal of Physics: Conference Series **513** (2014), 012029.

[9] M. Bodlak et al. *Development of new data acquisition system for COMPASS experiment.* Nuclear and Particle Physics Proceedings **273** (2016), 976 – 981. 37th International Conference on High Energy Physics (ICHEP).

[10] F. Borrelli, A. Bemporad, and M. Morari. *Predictive Control for Linear and Hybrid Systems.* Cambridge University Press, Cambridge, UK, first edition, (June 2017).

[11] L. Busoniu, R. Babuska, B. D. Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators.* CRC Press, Boca Raton, Florida, (2010).

[12] D.-S. Chen, R. G. Batson, and Y. Dang. *Applied Integer Programming: Modeling and Solution.* John Wiley & Sons, Inc., first edition, (2010). ISBN 978-0-470-37306-4.

[13] CASTOR — CERN Advanced Storage manager [online]. http://castor.web.cern.ch. Accessed 2020-09-01.

[14] Electronic developments for COMPASS at Freiburg [online]. https://twiki.cern.ch/twiki/pub/Compass/Detectors/FrontEndElectronics/catch-userguide.ps. Accessed 2020-09-01.

[15] Debugging definition [online]. http://searchsoftwarequality.techtarget.com/definition/debugging. Accessed: 2020-09-01.

[16] The GANDALF Module [online]. https://gandalf-framework.web.cern.ch/. Accessed 2020-09-01.

[17] iMUX/HGESICA module [online]. https://twiki.cern.ch/twiki/pub/Compass/Detectors/FrontEndElectronics/imux_manual.pdf. Accessed 2020-09-01.

[18] Linux at CERN [online]. http://linux.web.cern.ch/linux/scientific6/. Accessed 2020-09-01.

[19] S-Link — High Speed Interconnect [online]. http://hsi.web.cern.ch/HSI/s-link/. Accessed 2020-09-01.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, USA, third edition, (2009). ISBN 978-0-262-03384-8.

[21] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, first edition, (1989).

[22] S. Das, A. Konar, and U. K. Chakraborty. Two Improved Differential Evolution Schemes for Faster Global Search. In 'GECCO 2005 – Genetic and Evolutionary Computation Conference', 991–998, (January 2005).

[23] S. Das, A. Konar, and U. K. Chakraborty. Annealed Differential Evolution. In '2007 IEEE Congress on Evolutionary Computation', 1926–1933, (September 2007).

[24] S. Das, S. S. Mullick, and P. N. Suganthan. *Recent Advances in Differential Evolution -– An Updated Survey*. Swarm and Evolutionary Computation **27** (2016), 1–30.

[25] S. Das and P. N. Suganthan. *Differential Evolution: A Survey of the State-of-the-Art*. IEEE Transactions on Evolutionary Computation **15** (February 2011), 4–31.

[26] D. Delahaye, S. Chaimatanan, and M. Mongeau. *Simulated Annealing: From Basics to Applications*, 1–35. Springer International Publishing, (2019).

[27] M. Dell'Amico and S. Martello. *Reduction of the Three-Partition Problem*. Journal of Combinatorial Optimization **3** (July 1999), 17 – 30.

[28] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* Springer-Verlag Berlin Heidelberg, second edition, (2015).

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, first edition, (1979). ISBN 0-7167-1045-5.

[30] C. Gaspar and M. Dönszelmann. *DIM — A Distributed Information Management System for the DELPHI Experiment at CERN.* Proceedings of the 8th Conference on Real-Time Computer applications in Nuclear, Particle and Plasma Physics (June 1993). Vancouver, Canada.

[31] C. Gaspar, M. Dönszelmann, and P. Charpentier. *DIM — A Distributed Information Management System for the DELPHI Experiment at CERN.* International Conference on Computing in High Energy and Nuclear Physics (February 2000). Padova, Italy.

[32] C. Gaspar and J. J. Schwarz. *A Highly Distributed Control System for a Large Scale Experiment.* 13th IFAC workshop on Distributed Computer Control Systems — DCCS'95 (September 1995). Toulouse, France.

[33] Google. Documentation of Google Breakpad [online]. `https://github.com/google/breakpad/`. Accessed: 2020-09-01.

[34] T. Grötker, U. Holtmann, H. Keding, and M. Wloka. *The Developer's Guide to Debugging.* CreateSpace Independent Publishing Platform, second edition, (April 2012). ISBN 1-4701-8552-0.

[35] B. Grube. *A Trigger Control System for COMPASS and a Measurement of the Transverse Polarization of Lambda and Xi Hyperons from Quasi-Real Photo-Production.* PhD thesis, Technical University Munich, (2006).

[36] P. Hansen and N. Mladenoviă. *A Separable Approximation Dynamic Programming Algorithm for Economic Dispatch with Transmission Losses.* Yugoslav Journal of Operations Research (December 2002), 157–166.

[37] F. S. Hillier and G. J. Lieberman. *Introduction To Operations Research.* McGraw-Hill, Inc., New York, NY, seventh edition, (2001). ISBN 0072321695.

[38] IEEE. POSIX — Standards [online]. `http://standards.ieee.org/develop/wg/POSIX.html`. Accessed: 2020-09-01.

[39] IETF. *The Base16, Base32, and Base64 Data Encodings.* Internet Engineering Task Force, (October 2006). RFC 4648.

[40] IETF. *The WebSocket Protocol.* Internet Engineering Task Force, (December 2011). RFC 6455.

[41] ISO. *ISO 8601:2000. Data elements and interchange formats — Information interchange — Representation of dates and times.* International Organization for Standardization, (2000).

[42] H. Kameda, J. Li, C. Kim, and Y. Zhang. *Optimal Load Balancing in Distributed Computer Systems*. Springer-Verlag London, first edition, (1997).

[43] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Heidelberg, first edition, (2004). ISBN 978-3-540-24777-7.

[44] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. *Optimization by simulated annealing*. Science (May 1983).

[45] R. E. Korf. *Multi-Way Number Partitioning*. Proceedings of the 21st International Joint Conference on Artificial Intelligence (July 2009), 538 – 543.

[46] A. Kveton. State machines of the data acquisition system of the COMPASS experiment at CERN. Bachelor's thesis, Prague, CTU, (2015).

[47] M. Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing, Birmingham, UK, (2018). ISBN 1788834240.

[48] C. G. Larrea, K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea, and T. Williams. *IPbus: a flexible Ethernet-based control system for xTCA hardware*. Journal of Instrumentation **10** (2015), C02019.

[49] MathWorks. Documentation of MATLAB function intlinprog [online]. `https://ch.mathworks.com/help/optim/ug/intlinprog.html`. Accessed: 2020-09-01.

[50] J. A. Momoh. *Electric Power System Applications of Optimization*. CRC Press, Boca Raton, Florida, second edition, (2017). ISBN 9781351834841.

[51] K. G. Murty. *Linear Programming*. John Wiley & Sons, Inc., first edition, (1983). ISBN 978-0471097259.

[52] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, Inc., third edition, (November 2011). ISBN 978-1-118-03196-4.

[53] N. Amjady and H. Sharifzadeh. *Solution of non-convex economic dispatch problem considering valve loading effect by a new Modified Differential Evolution algorithm*. Elsevier — Electrical power and energy system (January 2010).

[54] W. B. Powell. *Approximate Dynamic Programming*. Willey-Interscience, New York, (2007). ISBN 0470171553.

[55] J. M. Renders and H. Bersini. *Hybridizing genetic algorithms with hill-climbing methods for global optimization: two possible ways*. Proceedings of the first IEEE conference on evolutionary computation (June 1994), 312 – 317.

[56] M. Sniedovich. *Dynamic Programming: Foundations and Principles*. CRC Press, Boca Raton, Florida, second edition, (2010). ISBN 978-0-8247-4099-3.

[57] D. Steffen et al. *Overview and Future Developments of the intelligent, FPGA-based DAQ (iFDAQ) of COMPASS*. Proceedings of Science **ICHEP2016** (02 2016), 912.

[58] D. Steffen et al. *Intelligence Elements and Performance of the FPGA-based DAQ of the COMPASS Experiment.* Proceedings of Science **TWEPP-17** (March 2018), 127.

[59] W. R. Stevens. *Unix Network Programming: Interprocess Communications.* Prentice Hall, second edition, (1998). ISBN 978-0132974295.

[60] R. Storn and K. Price. *Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces.* Journal of Global Optimization **23** (01 1995).

[61] O. Subrt et al. *The Communication Library DIALOG for iFDAQ of the COMPASS Experiment.* International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering **11** (September 2017), 372 – 381. 19th International Conference on High Energy Physics, Paris, France.

[62] O. Subrt et al. *The DAQ Debugger for iFDAQ of the COMPASS Experiment.* International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering **11** (December 2017), 448 – 456. 19th International Conference on Engineering, Computational and Technological Innovative Sciences, Amsterdam, the Netherlands.

[63] O. Subrt et al. *Modified Differential Evolution in the Load Balancing Problem for the iFDAQ of the COMPASS Experiment at CERN.* IJCCI 2019 – Proceedings of the 11th International Joint Conference on Computational Intelligence (September 2019), 213–220.

[64] O. Subrt et al. *The Continuously Running iFDAQ of the COMPASS Experiment.* EPJ Web Conf. **214** (2019), 8. 23rd International Conference on Computing in High Energy Physics (CHEP 2018), Sofia, Bulgaria, July 9-13, 2018.

[65] O. Subrt et al. *The Online Monitoring API for the DIALOG Library of the COMPASS Experiment.* EPJ Web Conf. **214** (2019), 8. 23rd International Conference on Computing in High Energy Physics (CHEP 2018), Sofia, Bulgaria, July 9-13, 2018.

[66] O. Subrt et al. *Dynamic Programming and Greedy Heuristic in the Load Balancing Problem for the iFDAQ of the COMPASS Experiment at CERN.* ISOCI 2020 – Proceedings of the 2020 International Symposium on Computational Intelligence (September 2020). Forthcoming, accepted for publication.

[67] O. Subrt et al. *Reinforcement Learning in the Load Balancing Problem for the iFDAQ of the COMPASS Experiment at CERN.* ICAART 2020 – Proceedings of the 12th International Conference on Agents and Artificial Intelligence (February 2020), 734–741.

[68] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, (1998). ISBN 0262193981.

[69] C. Szepesvari. *Algorithms for Reinforcement Learning.* Morgan and Claypool Publishers, San Rafael, California, (2010). ISBN 1608454924.

[70] The COIN-OR Foundation. Documentation of the COIN-OR project [online]. https://www.coin-or.org/Doxygen/CoinAll/. Accessed: 2020-09-01.

List of Figures

List of Tables

List of Algorithms and Listings

List of Publications

First Author Papers

In addition, the papers listed below have been presented by myself as a speaker on the corresponding conferences.

1. O. Subrt et al. *Dynamic Programming and Greedy Heuristic in the Load Balancing Problem for the iFDAQ of the COMPASS Experiment at CERN*. 12 pages. Proceedings of the 2020 International Symposium on Computational Intelligence (ISOCI 2020), Berlin, Germany, September 16 – 18, 2020. Forthcoming, accepted for publication.

2. O. Subrt et al. *Reinforcement Learning in the Load Balancing Problem for the iFDAQ of the COMPASS Experiment at CERN*. SciTePress (February 2020), 734 – 741. Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART 2020), Valletta, Malta, February 22 – 24, 2020. ISBN 978-989-758-395-7.

3. O. Subrt et al. *Modified Differential Evolution in the Load Balancing Problem for the iFDAQ of the COMPASS Experiment at CERN*. SciTePress (September 2019), 213 – 220. Proceedings of the 11th International Joint Conference on Computational Intelligence (IJCCI 2019), Vienna, Austria, September 17 – 19, 2019. ISBN 978-989-758-384-1.

4. O. Subrt et al. *The Continuously Running iFDAQ of the COMPASS Experiment*. EPJ Web of Conferences **214** (September 2019), 8 pages. 23rd International Conference on Computing in High Energy Physics (CHEP 2018), Sofia, Bulgaria, July 9 – 13, 2018.

5. O. Subrt et al. *The Online Monitoring API for the DIALOG Library of the COMPASS Experiment*. EPJ Web of Conferences **214** (September 2019), 8 pages. 23rd International Conference on Computing in High Energy Physics (CHEP 2018), Sofia, Bulgaria, July 9 – 13, 2018.

6. O. Subrt et al. *The DAQ Debugger for iFDAQ of the COMPASS Experiment*. International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering **11** (December 2017), 448 – 456. 19th International Conference on Engineering, Computational and Technological Innovative Sciences, Amsterdam, the Netherlands. ISSN 1307-6892.

7. O. Subrt et al. *The Communication Library DIALOG for iFDAQ of the COMPASS Experiment.* International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering **11** (September 2017), 372 – 381. 19th International Conference on High Energy Physics, Paris, France. ISSN 1307-6892.

8. O. Subrt and V. Merunka. *The Algorithmizable Modeling of the Object-Oriented Data Model in Craft.CASE.* Lecture Notes in Business Information Processing **272** (June 2016), 98 – 110. EOMAS 2016: Enterprise and Organizational Modeling and Simulation. CAiSE 2016 conference, EOMAS workshop, Ljubljana, Slovenia. ISBN 978-3-319-49454-8.

9. O. Subrt, K. Macek and M. Virius. *Modified Differential Evolution in Load Dispatch Optimization Problem.* 20th Annual Conference Proceeding's Technical Computing Bratislava 2012 (November 2012). Technical Computing Bratislava 2012, Bratislava, Slovakia.

Other Papers

The papers listed below have been published either by the COMPASS collaboration or another member of the COMPASS DAQ group.

1. M. G. Alexeev et al. *Antiproton over proton and K^- over K^+ multiplicity ratios at high z in DIS.* Physics Letters B. Elsevier **807** (August 2020), 19 pages. DOI 10.1016/j.physletb.2020.135600.

2. J. Agarwala et al. *Contribution of exclusive diffractive processes to the measured azimuthal asymmetries in SIDIS.* Nuclear Physics B. Elsevier **956** (July 2020), 13 pages. DOI 10.1016/j.nuclphysb.2020.115039.

3. M. G. Alexeev et al. *Measurement of the cross section for hard exclusive π^0 muoproduction on the proton.* Physics Letters B. Elsevier **805** (June 2020), 9 pages. DOI 10.1016/j.physletb.2020.135454.

4. A. Kveton et al. *A multi-purpose user interface for the iFDAQ of the COMPASS experiment.* 7 pages. 24th International Conference on Computing in High Energy Physics (CHEP 2019), Adelaide, Australia, July 4 – 8, 2019. Forthcoming, accepted for publication.

5. R. Akhunzyanov et al. *Transverse extension of partons in the proton probed in the sea-quark range by measuring the DVCS cross section.* Physics Letters B. Elsevier **793** (June 2019), 188 – 194. DOI 10.1016/j.physletb.2019.04.038.

6. M. G. Alexeev et al. *Measurement of P_T-weighted Sivers asymmetries in leptoproduction of hadrons.* Nuclear Physics B. Elsevier **940** (March 2019), 34 – 53. DOI 10.1016/j.nuclphysb.2018.12.024.

7. M. Aghasyan et al. *Light isovector resonances in $\pi^- p \to \pi^-\pi^-\pi^+ p$ at 190 GeV/c*. Physical Review D **98** (November 2018), issue 9. DOI 10.1103/PhysRevD.98.092003.

8. R. Akhunzyanov et al. *K^- over K^+ multiplicity ratio for kaons produced in DIS with a large fraction of the virtual-photon energy*. Physics Letters B. Elsevier **786** (November 2018), 390 – 398. DOI 10.1016/j.physletb.2018.09.052.

9. C. Adolph et al. *Azimuthal asymmetries of charged hadrons produced in high-energy muon scattering off longitudinally polarised deuterons*. The European Physical Journal C. Springer Science **78** (November 2018), 952. DOI 10.1140/epjc/s10052-018-6379-7.

10. M. Aghasyan et al. *Search for muoproduction of $X(3872)$ at COMPASS and indication of a new state $\widetilde{X}(3872)$*. Physics Letters B. Elsevier **783** (August 2018), 334 – 340. DOI 10.1016/j.physletb.2018.07.008.

11. M. Aghasyan et al. *Longitudinal double-spin asymmetry A_1^p and spin-dependent structure function g_1^p of the proton at small values of x and Q^2*. Physics Letters B. Elsevier **781** (June 2018), 364 – 472. DOI 10.1016/j.physletb.2018.03.044.

12. M. Aghasyan et al. *New analysis of $\eta\pi$ tensor resonances measured at the COMPASS experiment*. Physics Letters B. Elsevier **779** (April 2018), 364 – 472. DOI 10.1016/j.physletb.2018.01.017.

13. M. Aghasyan et al. *Transverse-momentum-dependent Multiplicities of Charged Hadrons in Muon-Deuteron Deep Inelastic Scattering*. Physical Review D **97** (February 2018), issue 3. DOI 10.1103/PhysRevD.97.032006.

14. M. Aghasyan et al. *First measurement of transverse-spin-dependent azimuthal asymmetries in the Drell-Yan Process*. Physical Review Letters **119** (September 2017), issue 11. DOI 10.1103/PhysRevLett.119.112002.

15. C. Adolph et al. *First measurement of the Sivers asymmetry for gluons using SIDIS data*. Physics Letters B. Elsevier **772** (September 2017), 854 – 864. DOI 10.1016/j.physletb.2017.07.018.

16. C. Adolph et al. *Sivers asymmetry extracted in SIDIS at the hard scales of the Drell–Yan process at COMPASS*. Physics Letters B. Elsevier **770** (July 2017), 138 – 145. DOI 10.1016/j.physletb.2017.04.042.

17. C. Adolph et al. *Final COMPASS results on the deuteron spin-dependent structure function g_1^d and the Bjorken sum rule*. Physics Letters B. Elsevier **769** (June 2017), 34 – 41. DOI 10.1016/j.physletb.2017.03.018.

18. C. Adolph et al. *Multiplicities of charged kaons from deep-inelastic muon scattering off an isoscalar target*. Physics Letters B. Elsevier **767** (April 2017), 131 – 141. DOI 10.1016/j.physletb.2017.01.053.

19. D. Steffen et al. *Intelligence Elements and Performance of the FPGA-based DAQ of the COMPASS Experiment.* Proceedings of Science (March 2018), 127 – 131. Topical Workshop on Electronics for Particle Physics (TWEPP 2017), Santa Cruz, California, USA, September 2017.

20. D. Steffen et al. *Overview and Future Developments of the intelligent, FPGA-based DAQ (iFDAQ) of COMPASS.* Proceedings of Science (February 2017), 912 – 915. 38th International Conference on High Energy Physics (ICHEP 2016), Chicago, IL, USA, August 2016.

Appendix

Appendix A

Test Case 1 – Detailed Results

A.1 Dynamic Programming

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_3	1,948	1	f_1	9,282
2	f_7	7,764	2	f_2	5,176
3	f_8	9,480	3	f_4	123
4	f_9	7,989	4	f_5	9,577
5	f_{10}	565	5	f_6	5,930
6	f_{11}	6,287	6	f_{13}	3,815
7	f_{12}	466	7	f_{14}	1,360
8	f_{16}	1,474	8	f_{15}	5,205
9	f_{18}	4,930	9	f_{17}	2,531
10	f_{19}	507	10	f_{22}	6,070
11	f_{20}	1,993	11	f_{23}	1,241
12	f_{21}	8,273	12	f_{26}	3,319
13	f_{24}	9,620	13	f_{27}	9,036
14	f_{25}	9,052	14	f_{29}	5,057
15	f_{28}	2,712	15	f_{30}	5,338
Σ		**73,060**	**Σ**		**73,060**

Table A.1: The best Test Case 1 flow allocation produced in C++ using DP.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_3	1,948	1	f_1	9,282
2	f_7	7,764	2	f_2	5,176
3	f_8	9,480	3	f_4	123
4	f_9	7,989	4	f_5	9,577
5	f_{10}	565	5	f_6	5,930
6	f_{11}	6,287	6	f_{13}	3,815
7	f_{12}	466	7	f_{14}	1,360
8	f_{16}	1,474	8	f_{15}	5,205
9	f_{18}	4,930	9	f_{17}	2,531
10	f_{19}	507	10	f_{22}	6,070
11	f_{20}	1,993	11	f_{23}	1,241
12	f_{21}	8,273	12	f_{26}	3,319
13	f_{24}	9,620	13	f_{27}	9,036
14	f_{25}	9,052	14	f_{29}	5,057
15	f_{28}	2,712	15	f_{30}	5,338
Σ		**73,060**	**Σ**		**73,060**

Table A.2: The best Test Case 1 flow allocation produced in Matlab using DP.

A.2 Greedy Heuristic

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{24}	9,620	1	f_5	9,577
2	f_1	9,282	2	f_8	9,480
3	f_{25}	9,052	3	f_{27}	9,036
4	f_{21}	8,273	4	f_9	7,989
5	f_{11}	6,287	5	f_7	7,764
6	f_{22}	6,070	6	f_6	5,930
7	f_{30}	5,338	7	f_{15}	5,205
8	f_2	5,176	8	f_{29}	5,057
9	f_{18}	4,930	9	f_{13}	3,815
10	f_{28}	2,712	10	f_{26}	3,319
11	f_{17}	2,531	11	f_{20}	1,993
12	f_{16}	1,474	12	f_3	1,948
13	f_{14}	1,360	13	f_{23}	1,241
14	f_{10}	565	14	f_{19}	507
15	f_{12}	466	15	f_4	123
Σ		**73,136**	**Σ**		**72,984**

Table A.3: The best Test Case 1 flow allocation produced in C++ using GH.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{24}	9,620	1	f_5	9,577
2	f_1	9,282	2	f_8	9,480
3	f_{25}	9,052	3	f_{27}	9,036
4	f_{21}	8,273	4	f_9	7,989
5	f_{11}	6,287	5	f_7	7,764
6	f_{22}	6,070	6	f_6	5,930
7	f_{30}	5,338	7	f_{15}	5,205
8	f_2	5,176	8	f_{29}	5,057
9	f_{18}	4,930	9	f_{13}	3,815
10	f_{28}	2,712	10	f_{26}	3,319
11	f_{17}	2,531	11	f_{20}	1,993
12	f_{16}	1,474	12	f_3	1,948
13	f_{14}	1,360	13	f_{23}	1,241
14	f_{10}	565	14	f_{19}	507
15	f_{12}	466	15	f_4	123
Σ		**73,136**	**Σ**		**72,984**

Table A.4: The best Test Case 1 flow allocation produced in Matlab using GH.

A.3 Integer Linear Programming

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_3	1,948	1	f_1	9,282
2	f_6	5,930	2	f_2	5,176
3	f_9	7,989	3	f_4	123
4	f_{11}	6,287	4	f_5	9,577
5	f_{15}	5,205	5	f_7	7,764
6	f_{16}	1,474	6	f_8	9,480
7	f_{17}	2,531	7	f_{10}	565
8	f_{20}	1,993	8	f_{12}	466
9	f_{21}	8,273	9	f_{13}	3,815
10	f_{22}	6,070	10	f_{14}	1,360
11	f_{23}	1,241	11	f_{18}	4,930
12	f_{25}	9,052	12	f_{19}	507
13	f_{26}	3,319	13	f_{24}	9,620
14	f_{27}	9,036	14	f_{29}	5,057
15	f_{28}	2,712	15	f_{30}	5,338
Σ		**73,060**	**Σ**		**73,060**

Table A.5: The best Test Case 1 flow allocation produced in C++ using ILP.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_3	1,948	1	f_1	9,282
2	f_5	9,577	2	f_2	5,176
3	f_6	5,930	3	f_4	123
4	f_8	9,480	4	f_7	7,764
5	f_{12}	466	5	f_9	7,989
6	f_{13}	3,815	6	f_{10}	565
7	f_{16}	1,474	7	f_{11}	6,287
8	f_{18}	4,930	8	f_{14}	1,360
9	f_{19}	507	9	f_{15}	5,205
10	f_{20}	1,993	10	f_{17}	2,531
11	f_{21}	8,273	11	f_{22}	6,070
12	f_{23}	1,241	12	f_{24}	9,620
13	f_{25}	9,052	13	f_{26}	3,319
14	f_{27}	9,036	14	f_{28}	2,712
15	f_{30}	5,338	15	f_{29}	5,057
Σ		**73,060**	**Σ**		**73,060**

Table A.6: The best Test Case 1 flow allocation produced in Matlab using ILP.

A.4 Modified Differential Evolution

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_6	5,930	1	f_9	7,989
2	f_{16}	1,474	2	f_{27}	9,036
3	f_{22}	6,070	3	f_{15}	5,205
4	f_7	7,764	4	f_1	9,282
5	f_{25}	9,052	5	f_{20}	1,993
6	f_3	1,948	6	f_{19}	507
7	f_5	9,577	7	f_{21}	8,273
8	f_2	5,176	8	f_{26}	3,319
9	f_{23}	1,241	9	f_{11}	6,287
10	f_{14}	1,360	10	f_4	123
11	f_{13}	3,815	11	f_{30}	5,338
12	f_{17}	2,531	12	f_{12}	466
13	f_8	9,480	13	f_{10}	565
14	f_{28}	2,712	14	f_{24}	9,620
15	f_{18}	4,930	15	f_{29}	5,057
Σ		**73,060**	**Σ**		**73,060**

Table A.7: The best Test Case 1 flow allocation produced in C++ using the MDE.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{22}	6,070	1	f_7	7,764
2	f_{11}	6,287	2	f_8	9,480
3	f_{12}	466	3	f_{24}	9,620
4	f_{28}	2,712	4	f_{21}	8,273
5	f_{27}	9,036	5	f_{30}	5,338
6	f_{20}	1,993	6	f_4	123
7	f_{19}	507	7	f_{10}	565
8	f_{25}	9,052	8	f_{16}	1,474
9	f_5	9,577	9	f_2	5,176
10	f_1	9,282	10	f_9	7,989
11	f_{23}	1,241	11	f_{15}	5,205
12	f_{29}	5,057	12	f_{13}	3,815
13	f_6	5,930	13	f_{14}	1,360
14	f_{17}	2,531	14	f_{18}	4,930
15	f_{26}	3,319	15	f_3	1,948
Σ		**73,060**	**Σ**		**73,060**

Table A.8: The best Test Case 1 flow allocation produced in Matlab using the MDE.

A.5 Reinforcement Learning

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{24}	9,620	1	f_1	9,282
2	f_5	9,577	2	f_9	7,989
3	f_8	9,480	3	f_{11}	6,287
4	f_{25}	9,052	4	f_{22}	6,070
5	f_{27}	9,036	5	f_6	5,930
6	f_{21}	8,273	6	f_{30}	5,338
7	f_7	7,764	7	f_{15}	5,205
8	f_{13}	3,815	8	f_2	5,176
9	f_3	1,948	9	f_{29}	5,057
10	f_{16}	1,474	10	f_{18}	4,930
11	f_{14}	1,360	11	f_{26}	3,319
12	f_{10}	565	12	f_{28}	2,712
13	f_{19}	507	13	f_{17}	2,531
14	f_{12}	466	14	f_{20}	1,993
15	f_4	123	15	f_{23}	1,241
Σ		**73,060**	**Σ**		**73,060**

Table A.9: The best Test Case 1 flow allocation produced in C++ using RL.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{24}	9,620	1	f_1	9,282
2	f_5	9,577	2	f_9	7,989
3	f_8	9,480	3	f_{11}	6,287
4	f_{25}	9,052	4	f_{22}	6,070
5	f_{27}	9,036	5	f_6	5,930
6	f_{21}	8,273	6	f_{30}	5,338
7	f_7	7,764	7	f_{15}	5,205
8	f_{13}	3,815	8	f_2	5,176
9	f_3	1,948	9	f_{29}	5,057
10	f_{16}	1,474	10	f_{18}	4,930
11	f_{14}	1,360	11	f_{26}	3,319
12	f_{10}	565	12	f_{28}	2,712
13	f_{19}	507	13	f_{17}	2,531
14	f_{12}	466	14	f_{20}	1,993
15	f_4	123	15	f_{23}	1,241
Σ		**73,060**	**Σ**		**73,060**

Table A.10: The best Test Case 1 flow allocation produced in Matlab using RL.

Appendix B

Test Case 2 – Detailed Results

B.1 Dynamic Programming

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_5	8,109	1	f_1	668	1	f_2	7,870
2	f_7	8,998	2	f_3	5,280	2	f_{16}	1,485
3	f_{10}	714	3	f_4	4,988	3	f_{23}	5,028
4	f_{13}	1,757	4	f_6	2,825	4	f_{33}	9,037
5	f_{14}	5,267	5	f_9	7,151	5	f_{38}	7,950
6	f_{15}	5,767	6	f_{11}	3,501	6	f_{41}	3,097
7	f_{19}	7,647	7	f_{17}	6,757	7	f_{44}	2,354
8	f_{20}	8,204	8	f_{18}	6,429	8	f_{53}	6,683
9	f_{21}	4,149	9	f_{34}	9,894	9	f_{54}	8,703
10	f_{22}	7,720	10	f_{40}	6,721	10	f_{55}	854
11	f_{27}	4,504	11	f_{43}	8,043	11	f_{58}	5,745
12	f_{29}	7,065	12	f_{45}	5,015	12	f_{60}	4,753
13	f_{32}	2,783	13	f_{46}	7,160	13	f_{65}	3,429
14	f_{37}	5,743	14	f_{48}	2,743	14	f_{74}	5,637
15	f_{39}	4,067	15	f_{49}	5,318	15	f_{80}	9,868
Σ		**82,494**	**Σ**		**82,493**	**Σ**		**82,493**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{25}	8,007	1	f_8	6,853	1	f_{12}	9,164
2	f_{26}	7,766	2	f_{35}	9,854	2	f_{24}	7,313
3	f_{30}	8,789	3	f_{42}	3,876	3	f_{28}	6,632
4	f_{36}	3,235	4	f_{47}	4,619	4	f_{31}	9,136
5	f_{50}	6,399	5	f_{61}	8,159	5	f_{52}	1,422
6	f_{51}	2,806	6	f_{62}	9,293	6	f_{59}	9,333
7	f_{56}	9,920	7	f_{64}	9,692	7	f_{67}	7,589
8	f_{57}	710	8	f_{66}	4,021	8	f_{69}	2,341
9	f_{63}	5,795	9	f_{68}	3,579	9	f_{72}	9,034
10	f_{77}	8,753	10	f_{70}	425	10	f_{79}	3,606
11	f_{78}	4,977	11	f_{71}	4,370	11	f_{81}	7,468
12	f_{83}	1,710	12	f_{73}	7,278	12	f_{82}	2,028
13	f_{85}	5,921	13	f_{75}	6,888	13	f_{86}	64
14	f_{87}	4,079	14	f_{76}	2,345	14	f_{88}	4,727
15	f_{89}	3,626	15	f_{84}	1,241	15	f_{90}	2,636
Σ		**82,493**	**Σ**		**82,493**	**Σ**		**82,493**

Table B.1: The best Test Case 2 flow allocation produced in C++ using DP.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_5	8,109	1	f_1	668	1	f_2	7,870
2	f_7	8,998	2	f_3	5,280	2	f_{16}	1,485
3	f_{10}	714	3	f_4	4,988	3	f_{23}	5,028
4	f_{13}	1,757	4	f_6	2,825	4	f_{33}	9,037
5	f_{14}	5,267	5	f_9	7,151	5	f_{38}	7,950
6	f_{15}	5,767	6	f_{11}	3,501	6	f_{41}	3,097
7	f_{19}	7,647	7	f_{17}	6,757	7	f_{44}	2,354
8	f_{20}	8,204	8	f_{18}	6,429	8	f_{53}	6,683
9	f_{21}	4,149	9	f_{34}	9,894	9	f_{54}	8,703
10	f_{22}	7,720	10	f_{40}	6,721	10	f_{55}	854
11	f_{27}	4,504	11	f_{43}	8,043	11	f_{58}	5,745
12	f_{29}	7,065	12	f_{45}	5,015	12	f_{60}	4,753
13	f_{32}	2,783	13	f_{46}	7,160	13	f_{65}	3,429
14	f_{37}	5,743	14	f_{48}	2,743	14	f_{74}	5,637
15	f_{39}	4,067	15	f_{49}	5,318	15	f_{80}	9,868
Σ		**82,494**	**Σ**		**82,493**	**Σ**		**82,493**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{25}	8,007	1	f_8	6,853	1	f_{12}	9,164
2	f_{26}	7,766	2	f_{35}	9,854	2	f_{24}	7,313
3	f_{30}	8,789	3	f_{42}	3,876	3	f_{28}	6,632
4	f_{36}	3,235	4	f_{47}	4,619	4	f_{31}	9,136
5	f_{50}	6,399	5	f_{61}	8,159	5	f_{52}	1,422
6	f_{51}	2,806	6	f_{62}	9,293	6	f_{59}	9,333
7	f_{56}	9,920	7	f_{64}	9,692	7	f_{67}	7,589
8	f_{57}	710	8	f_{66}	4,021	8	f_{69}	2,341
9	f_{63}	5,795	9	f_{68}	3,579	9	f_{72}	9,034
10	f_{77}	8,753	10	f_{70}	425	10	f_{79}	3,606
11	f_{78}	4,977	11	f_{71}	4,370	11	f_{81}	7,468
12	f_{83}	1,710	12	f_{73}	7,278	12	f_{82}	2,028
13	f_{85}	5,921	13	f_{75}	6,888	13	f_{86}	64
14	f_{87}	4,079	14	f_{76}	2,345	14	f_{88}	4,727
15	f_{89}	3,626	15	f_{84}	1,241	15	f_{90}	2,636
Σ		**82,493**	**Σ**		**82,493**	**Σ**		**82,493**

Table B.2: The best Test Case 2 flow allocation produced in Matlab using DP.

B.2 Greedy Heuristic

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{56}	9,920	1	f_{34}	9,894	1	f_{80}	9,868
2	f_{7}	8,998	2	f_{72}	9,034	2	f_{33}	9,037
3	f_{20}	8,204	3	f_{61}	8,159	3	f_{54}	8,703
4	f_{38}	7,950	4	f_{43}	8,043	4	f_{26}	7,766
5	f_{19}	7,647	5	f_{81}	7,468	5	f_{46}	7,160
6	f_{40}	6,721	6	f_{29}	7,065	6	f_{9}	7,151
7	f_{53}	6,683	7	f_{85}	5,921	7	f_{63}	5,795
8	f_{3}	5,280	8	f_{58}	5,745	8	f_{15}	5,767
9	f_{4}	4,988	9	f_{23}	5,028	9	f_{14}	5,267
10	f_{27}	4,504	10	f_{88}	4,727	10	f_{87}	4,079
11	f_{89}	3,626	11	f_{68}	3,579	11	f_{39}	4,067
12	f_{11}	3,501	12	f_{41}	3,097	12	f_{36}	3,235
13	f_{76}	2,345	13	f_{90}	2,636	13	f_{44}	2,354
14	f_{16}	1,485	14	f_{84}	1,241	14	f_{13}	1,757
15	f_{57}	710	15	f_{55}	854	15	f_{70}	425
Σ		**82,562**	**Σ**		**82,491**	**Σ**		**82,431**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{35}	9,854	1	f_{64}	9,692	1	f_{59}	9,333
2	f_{31}	9,136	2	f_{12}	9,164	2	f_{62}	9,293
3	f_{5}	8,109	3	f_{77}	8,753	3	f_{30}	8,789
4	f_{25}	8,007	4	f_{22}	7,720	4	f_{2}	7,870
5	f_{67}	7,589	5	f_{73}	7,278	5	f_{24}	7,313
6	f_{17}	6,757	6	f_{8}	6,853	6	f_{75}	6,888
7	f_{28}	6,632	7	f_{18}	6,429	7	f_{50}	6,399
8	f_{49}	5,318	8	f_{74}	5,637	8	f_{37}	5,743
9	f_{45}	5,015	9	f_{78}	4,977	9	f_{60}	4,753
10	f_{71}	4,370	10	f_{21}	4,149	10	f_{47}	4,619
11	f_{42}	3,876	11	f_{66}	4,021	11	f_{79}	3,606
12	f_{6}	2,825	12	f_{51}	2,806	12	f_{65}	3,429
13	f_{48}	2,743	13	f_{32}	2,783	13	f_{69}	2,341
14	f_{82}	2,028	14	f_{83}	1,710	14	f_{52}	1,422
15	f_{86}	64	15	f_{1}	668	15	f_{10}	714
Σ		**82,323**	**Σ**		**82,640**	**Σ**		**82,512**

Table B.3: The best Test Case 2 flow allocation produced in C++ using GH.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{56}	9,920	1	f_{34}	9,894	1	f_{80}	9,868
2	f_{7}	8,998	2	f_{72}	9,034	2	f_{33}	9,037
3	f_{20}	8,204	3	f_{61}	8,159	3	f_{54}	8,703
4	f_{38}	7,950	4	f_{43}	8,043	4	f_{26}	7,766
5	f_{19}	7,647	5	f_{81}	7,468	5	f_{46}	7,160
6	f_{40}	6,721	6	f_{29}	7,065	6	f_{9}	7,151
7	f_{53}	6,683	7	f_{85}	5,921	7	f_{63}	5,795
8	f_{3}	5,280	8	f_{58}	5,745	8	f_{15}	5,767
9	f_{4}	4,988	9	f_{23}	5,028	9	f_{14}	5,267
10	f_{27}	4,504	10	f_{88}	4,727	10	f_{87}	4,079
11	f_{89}	3,626	11	f_{68}	3,579	11	f_{39}	4,067
12	f_{11}	3,501	12	f_{41}	3,097	12	f_{36}	3,235
13	f_{76}	2,345	13	f_{90}	2,636	13	f_{44}	2,354
14	f_{16}	1,485	14	f_{84}	1,241	14	f_{13}	1,757
15	f_{57}	710	15	f_{55}	854	15	f_{70}	425
Σ		**82,562**	**Σ**		**82,491**	**Σ**		**82,431**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{35}	9,854	1	f_{64}	9,692	1	f_{59}	9,333
2	f_{31}	9,136	2	f_{12}	9,164	2	f_{62}	9,293
3	f_{5}	8,109	3	f_{77}	8,753	3	f_{30}	8,789
4	f_{25}	8,007	4	f_{22}	7,720	4	f_{2}	7,870
5	f_{67}	7,589	5	f_{73}	7,278	5	f_{24}	7,313
6	f_{17}	6,757	6	f_{8}	6,853	6	f_{75}	6,888
7	f_{28}	6,632	7	f_{18}	6,429	7	f_{50}	6,399
8	f_{49}	5,318	8	f_{74}	5,637	8	f_{37}	5,743
9	f_{45}	5,015	9	f_{78}	4,977	9	f_{60}	4,753
10	f_{71}	4,370	10	f_{21}	4,149	10	f_{47}	4,619
11	f_{42}	3,876	11	f_{66}	4,021	11	f_{79}	3,606
12	f_{6}	2,825	12	f_{51}	2,806	12	f_{65}	3,429
13	f_{48}	2,743	13	f_{32}	2,783	13	f_{69}	2,341
14	f_{82}	2,028	14	f_{83}	1,710	14	f_{52}	1,422
15	f_{86}	64	15	f_{1}	668	15	f_{10}	714
Σ		**82,323**	**Σ**		**82,640**	**Σ**		**82,512**

Table B.4: The best Test Case 2 flow allocation produced in Matlab using GH.

B.3 Integer Linear Programming

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{21}	4,149	1	f_1	668	1	f_9	7,151
2	f_{22}	7,720	2	f_2	7,870	2	f_{11}	3,501
3	f_{27}	4,504	3	f_7	8,998	3	f_{19}	7,647
4	f_{32}	2,783	4	f_{16}	1,485	4	f_{28}	6,632
5	f_{33}	9,037	5	f_{30}	8,789	5	f_{35}	9,854
6	f_{37}	5,743	6	f_{34}	9,894	6	f_{40}	6,721
7	f_{38}	7,950	7	f_{56}	9,920	7	f_{43}	8,043
8	f_{49}	5,318	8	f_{57}	710	8	f_{53}	6,683
9	f_{68}	3,579	9	f_{60}	4,753	9	f_{63}	5,795
10	f_{71}	4,370	10	f_{61}	8,159	10	f_{69}	2,341
11	f_{72}	9,034	11	f_{65}	3,429	11	f_{73}	7,278
12	f_{74}	5,637	12	f_{67}	7,589	12	f_{84}	1,241
13	f_{78}	4,977	13	f_{70}	425	13	f_{85}	5,921
14	f_{79}	3,606	14	f_{76}	2,345	14	f_{86}	64
15	f_{87}	4,079	15	f_{81}	7,468	15	f_{89}	3,626
Σ		**82,486**	**Σ**		**82,502**	**Σ**		**82,498**
$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_4	4,988	1	f_8	6,853	1	f_3	5,280
2	f_5	8,109	2	f_{10}	714	2	f_6	2,825
3	f_{12}	9,164	3	f_{13}	1,757	3	f_{14}	5,267
4	f_{31}	9,136	4	f_{15}	5,767	4	f_{18}	6,429
5	f_{39}	4,067	5	f_{17}	6,757	5	f_{23}	5,028
6	f_{41}	3,097	6	f_{20}	8,204	6	f_{24}	7,313
7	f_{42}	3,876	7	f_{25}	8,007	7	f_{45}	5,015
8	f_{44}	2,354	8	f_{26}	7,766	8	f_{46}	7,160
9	f_{47}	4,619	9	f_{29}	7,065	9	f_{50}	6,399
10	f_{48}	2,743	10	f_{36}	3,235	10	f_{51}	2,806
11	f_{55}	854	11	f_{54}	8,703	11	f_{52}	1,422
12	f_{58}	5,745	12	f_{62}	9,293	12	f_{75}	6,888
13	f_{59}	9,333	13	f_{66}	4,021	13	f_{77}	8,753
14	f_{64}	9,692	14	f_{83}	1,710	14	f_{80}	9,868
15	f_{88}	4,727	15	f_{90}	2,636	15	f_{82}	2,028
Σ		**82,504**	**Σ**		**82,488**	**Σ**		**82,481**

Table B.5: The best Test Case 2 flow allocation produced in C++ using ILP.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{21}	4,149	1	f_3	5,280	1	f_4	4,988
2	f_{23}	5,028	2	f_5	8,109	2	f_{13}	1,757
3	f_{34}	9,894	3	f_8	6,853	3	f_{14}	5,267
4	f_{37}	5,743	4	f_9	7,151	4	f_{17}	6,757
5	f_{42}	3,876	5	f_{15}	5,767	5	f_{32}	2,783
6	f_{45}	5,015	6	f_{16}	1,485	6	f_{35}	9,854
7	f_{46}	7,160	7	f_{18}	6,429	7	f_{36}	3,235
8	f_{49}	5,318	8	f_{22}	7,720	8	f_{41}	3,097
9	f_{56}	9,920	9	f_{28}	6,632	9	f_{44}	2,354
10	f_{57}	710	10	f_{38}	7,950	10	f_{48}	2,743
11	f_{58}	5,745	11	f_{47}	4,619	11	f_{59}	9,333
12	f_{60}	4,753	12	f_{50}	6,399	12	f_{63}	5,795
13	f_{74}	5,637	13	f_{70}	425	13	f_{64}	9,692
14	f_{85}	5,921	14	f_{79}	3,606	14	f_{78}	4,977
15	f_{89}	3,626	15	f_{87}	4,079	15	f_{80}	9,868
Σ		**82,495**	**Σ**		**82,504**	**Σ**		**82,500**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_1	668	1	f_7	8,998	1	f_{19}	7,647
2	f_2	7,870	2	f_{12}	9,164	2	f_{20}	8,204
3	f_6	2,825	3	f_{24}	7,313	3	f_{25}	8,007
4	f_{10}	714	4	f_{31}	9,136	4	f_{27}	4,504
5	f_{11}	3,501	5	f_{33}	9,037	5	f_{43}	8,043
6	f_{26}	7,766	6	f_{39}	4,067	6	f_{51}	2,806
7	f_{29}	7,065	7	f_{52}	1,422	7	f_{53}	6,683
8	f_{30}	8,789	8	f_{61}	8,159	8	f_{54}	8,703
9	f_{40}	6,721	9	f_{62}	9,293	9	f_{65}	3,429
10	f_{55}	854	10	f_{66}	4,021	10	f_{69}	2,341
11	f_{67}	7,589	11	f_{68}	3,579	11	f_{73}	7,278
12	f_{72}	9,034	12	f_{71}	4,370	12	f_{76}	2,345
13	f_{75}	6,888	13	f_{84}	1,241	13	f_{77}	8,753
14	f_{81}	7,468	14	f_{86}	64	14	f_{82}	2,028
15	f_{88}	4,727	15	f_{90}	2,636	15	f_{83}	1,710
Σ		**82,479**	**Σ**		**82,500**	**Σ**		**82,481**

Table B.6: The best Test Case 2 flow allocation produced in Matlab using ILP.

B.4 Modified Differential Evolution

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{15}	5,767	1	f_{24}	7,313	1	f_{2}	7,870
2	f_{59}	9,333	2	f_{3}	5,280	2	f_{12}	9,164
3	f_{88}	4,727	3	f_{8}	6,853	3	f_{74}	5,637
4	f_{65}	3,429	4	f_{42}	3,876	4	f_{44}	2,354
5	f_{61}	8,159	5	f_{39}	4,067	5	f_{7}	8,998
6	f_{52}	1,422	6	f_{40}	6,721	6	f_{67}	7,589
7	f_{60}	4,753	7	f_{76}	2,345	7	f_{5}	8,109
8	f_{56}	9,920	8	f_{35}	9,854	8	f_{79}	3,606
9	f_{73}	7,278	9	f_{31}	9,136	9	f_{25}	8,007
10	f_{28}	6,632	10	f_{19}	7,647	10	f_{13}	1,757
11	f_{47}	4,619	11	f_{41}	3,097	11	f_{36}	3,235
12	f_{14}	5,267	12	f_{38}	7,950	12	f_{55}	854
13	f_{50}	6,399	13	f_{11}	3,501	13	f_{78}	4,977
14	f_{87}	4,079	14	f_{6}	2,825	14	f_{17}	6,757
15	f_{57}	710	15	f_{82}	2,028	15	f_{68}	3,579
Σ		**82,494**	**Σ**		**82,493**	**Σ**		**82,493**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{34}	9,894	1	f_{48}	2,743	1	f_{54}	8,703
2	f_{45}	5,015	2	f_{49}	5,318	2	f_{21}	4,149
3	f_{53}	6,683	3	f_{69}	2,341	3	f_{81}	7,468
4	f_{62}	9,293	4	f_{46}	7,160	4	f_{23}	5,028
5	f_{10}	714	5	f_{4}	4,988	5	f_{66}	4,021
6	f_{89}	3,626	6	f_{86}	64	6	f_{83}	1,710
7	f_{63}	5,795	7	f_{85}	5,921	7	f_{1}	668
8	f_{84}	1,241	8	f_{29}	7,065	8	f_{71}	4,370
9	f_{43}	8,043	9	f_{37}	5,743	9	f_{58}	5,745
10	f_{9}	7,151	10	f_{70}	425	10	f_{16}	1,485
11	f_{20}	8,204	11	f_{18}	6,429	11	f_{26}	7,766
12	f_{90}	2,636	12	f_{22}	7,720	12	f_{32}	2,783
13	f_{75}	6,888	13	f_{77}	8,753	13	f_{33}	9,037
14	f_{51}	2,806	14	f_{72}	9,034	14	f_{64}	9,692
15	f_{27}	4,504	15	f_{30}	8,789	15	f_{80}	9,868
Σ		**82,493**	**Σ**		**82,493**	**Σ**		**82,493**

Table B.7: The best Test Case 2 flow allocation produced in C++ using the MDE.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{50}	6,399	1	f_{67}	7,589	1	f_{32}	2,783
2	f_{21}	4,149	2	f_{53}	6,683	2	f_{39}	4,067
3	f_{80}	9,868	3	f_{86}	64	3	f_{56}	9,920
4	f_{9}	7,151	4	f_{83}	1,710	4	f_{87}	4,079
5	f_{41}	3,097	5	f_{12}	9,164	5	f_{30}	8,789
6	f_{79}	3,606	6	f_{48}	2,743	6	f_{43}	8,043
7	f_{55}	854	7	f_{14}	5,267	7	f_{33}	9,037
8	f_{28}	6,632	8	f_{8}	6,853	8	f_{1}	668
9	f_{20}	8,204	9	f_{77}	8,753	9	f_{63}	5,795
10	f_{19}	7,647	10	f_{76}	2,345	10	f_{36}	3,235
11	f_{74}	5,637	11	f_{47}	4,619	11	f_{84}	1,241
12	f_{10}	714	12	f_{5}	8,109	12	f_{6}	2,825
13	f_{71}	4,370	13	f_{3}	5,280	13	f_{17}	6,757
14	f_{75}	6,888	14	f_{62}	9,293	14	f_{85}	5,921
15	f_{73}	7,278	15	f_{66}	4,021	15	f_{59}	9,333
Σ		**82,494**	**Σ**		**82,493**	**Σ**		**82,493**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{42}	3,876	1	f_{46}	7,160	1	f_{60}	4,753
2	f_{57}	710	2	f_{13}	1,757	2	f_{15}	5,767
3	f_{7}	8,998	3	f_{11}	3,501	3	f_{65}	3,429
4	f_{72}	9,034	4	f_{88}	4,727	4	f_{31}	9,136
5	f_{27}	4,504	5	f_{78}	4,977	5	f_{70}	425
6	f_{64}	9,692	6	f_{81}	7,468	6	f_{29}	7,065
7	f_{58}	5,745	7	f_{44}	2,354	7	f_{26}	7,766
8	f_{82}	2,028	8	f_{51}	2,806	8	f_{69}	2,341
9	f_{38}	7,950	9	f_{34}	9,894	9	f_{35}	9,854
10	f_{18}	6,429	10	f_{25}	8,007	10	f_{37}	5,743
11	f_{90}	2,636	11	f_{45}	5,015	11	f_{49}	5,318
12	f_{23}	5,028	12	f_{2}	7,870	12	f_{89}	3,626
13	f_{40}	6,721	13	f_{61}	8,159	13	f_{4}	4,988
14	f_{52}	1,422	14	f_{24}	7,313	14	f_{68}	3,579
15	f_{22}	7,720	15	f_{16}	1,485	15	f_{54}	8,703
Σ		**82,493**	**Σ**		**82,493**	**Σ**		**82,493**

Table B.8: The best Test Case 2 flow allocation produced in Matlab using the MDE.

B.5 Reinforcement Learning

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{34}	9,894	1	f_{56}	9,920	1	f_{54}	8,703
2	f_{80}	9,868	2	f_{33}	9,037	2	f_{61}	8,159
3	f_{35}	9,854	3	f_{72}	9,034	3	f_{43}	8,043
4	f_{64}	9,692	4	f_{7}	8,998	4	f_{25}	8,007
5	f_{59}	9,333	5	f_{30}	8,789	5	f_{38}	7,950
6	f_{62}	9,293	6	f_{77}	8,753	6	f_{2}	7,870
7	f_{12}	9,164	7	f_{20}	8,204	7	f_{22}	7,720
8	f_{31}	9,136	8	f_{26}	7,766	8	f_{73}	7,278
9	f_{6}	2,825	9	f_{76}	2,345	9	f_{41}	3,097
10	f_{55}	854	10	f_{82}	2,028	10	f_{51}	2,806
11	f_{10}	714	11	f_{13}	1,757	11	f_{32}	2,783
12	f_{57}	710	12	f_{83}	1,710	12	f_{48}	2,743
13	f_{1}	668	13	f_{16}	1,485	13	f_{90}	2,636
14	f_{70}	425	14	f_{52}	1,422	14	f_{44}	2,354
15	f_{86}	64	15	f_{84}	1,241	15	f_{69}	2,341
Σ		**82,494**	**Σ**		**82,489**	**Σ**		**82,490**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{5}	8,109	1	f_{29}	7,065	1	f_{18}	6,429
2	f_{19}	7,647	2	f_{75}	6,888	2	f_{50}	6,399
3	f_{67}	7,589	3	f_{8}	6,853	3	f_{85}	5,921
4	f_{81}	7,468	4	f_{17}	6,757	4	f_{63}	5,795
5	f_{24}	7,313	5	f_{40}	6,721	5	f_{15}	5,767
6	f_{46}	7,160	6	f_{53}	6,683	6	f_{58}	5,745
7	f_{9}	7,151	7	f_{28}	6,632	7	f_{37}	5,743
8	f_{45}	5,015	8	f_{23}	5,028	8	f_{74}	5,637
9	f_{39}	4,067	9	f_{60}	4,753	9	f_{49}	5,318
10	f_{89}	3,626	10	f_{47}	4,619	10	f_{3}	5,280
11	f_{79}	3,606	11	f_{71}	4,370	11	f_{14}	5,267
12	f_{68}	3,579	12	f_{21}	4,149	12	f_{4}	4,988
13	f_{11}	3,501	13	f_{87}	4,079	13	f_{78}	4,977
14	f_{65}	3,429	14	f_{66}	4,021	14	f_{88}	4,727
15	f_{36}	3,235	15	f_{42}	3,876	15	f_{27}	4,504
Σ		**82,495**	**Σ**		**82,494**	**Σ**		**82,497**

Table B.9: The best Test Case 2 flow allocation produced in C++ using RL.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{34}	9,894	1	f_{56}	9,920	1	f_{7}	8,998
2	f_{80}	9,868	2	f_{33}	9,037	2	f_{20}	8,204
3	f_{35}	9,854	3	f_{72}	9,034	3	f_{43}	8,043
4	f_{64}	9,692	4	f_{30}	8,789	4	f_{25}	8,007
5	f_{59}	9,333	5	f_{77}	8,753	5	f_{38}	7,950
6	f_{62}	9,293	6	f_{54}	8,703	6	f_{2}	7,870
7	f_{12}	9,164	7	f_{61}	8,159	7	f_{26}	7,766
8	f_{31}	9,136	8	f_{5}	8,109	8	f_{75}	6,888
9	f_{6}	2,825	9	f_{76}	2,345	9	f_{41}	3,097
10	f_{55}	854	10	f_{82}	2,028	10	f_{51}	2,806
11	f_{10}	714	11	f_{13}	1,757	11	f_{32}	2,783
12	f_{57}	710	12	f_{83}	1,710	12	f_{48}	2,743
13	f_{1}	668	13	f_{16}	1,485	13	f_{90}	2,636
14	f_{70}	425	14	f_{52}	1,422	14	f_{44}	2,354
15	f_{86}	64	15	f_{84}	1,241	15	f_{69}	2,341
Σ		**82,494**	**Σ**		**82,492**	**Σ**		**82,486**

$\mathbf{MUX_4}$			$\mathbf{MUX_5}$			$\mathbf{MUX_6}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{22}	7,720	1	f_{29}	7,065	1	f_{73}	7,278
2	f_{19}	7,647	2	f_{8}	6,853	2	f_{28}	6,632
3	f_{67}	7,589	3	f_{17}	6,757	3	f_{63}	5,795
4	f_{81}	7,468	4	f_{40}	6,721	4	f_{15}	5,767
5	f_{24}	7,313	5	f_{53}	6,683	5	f_{58}	5,745
6	f_{46}	7,160	6	f_{18}	6,429	6	f_{37}	5,743
7	f_{9}	7,151	7	f_{50}	6,399	7	f_{74}	5,637
8	f_{49}	5,318	8	f_{85}	5,921	8	f_{3}	5,280
9	f_{21}	4,149	9	f_{60}	4,753	9	f_{14}	5,267
10	f_{89}	3,626	10	f_{27}	4,504	10	f_{23}	5,028
11	f_{79}	3,606	11	f_{71}	4,370	11	f_{45}	5,015
12	f_{68}	3,579	12	f_{87}	4,079	12	f_{4}	4,988
13	f_{11}	3,501	13	f_{39}	4,067	13	f_{78}	4,977
14	f_{65}	3,429	14	f_{66}	4,021	14	f_{88}	4,727
15	f_{36}	3,235	15	f_{42}	3,876	15	f_{47}	4,619
Σ		**82,491**	**Σ**		**82,498**	**Σ**		**82,498**

Table B.10: The best Test Case 2 flow allocation produced in Matlab using RL.

Appendix C

Test Case 3 – Detailed Results

C.1 Dynamic Programming

MUX_1			MUX_2			MUX_3			MUX_4		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_2	582	1	f_8	2,266	1	f_4	3,140	1	f_{16}	380
2	f_3	4,409	2	f_9	4,150	2	f_{19}	3,685	2	f_{23}	1,091
3	f_7	2,149	3	f_{24}	3,684	3	f_{32}	6,000	3	f_{29}	2,795
4	f_{15}	7,845	4	f_{25}	7,432	4	f_{33}	249	4	f_{61}	6,618
5	f_{17}	4,718	5	f_{39}	3,066	5	f_{35}	5,194	5	f_{69}	4,335
6	f_{27}	2,888	6	f_{53}	1,929	6	f_{36}	7,196	6	f_{72}	9,179
7	f_{30}	3,165	7	f_{58}	8,885	7	f_{44}	6,401	7	f_{79}	4,646
8	f_{31}	7,346	8	f_{60}	2,841	8	f_{46}	8,114	8	f_{82}	5,433
9	f_{45}	151	9	f_{71}	3,059	9	f_{49}	6,835	9	f_{89}	7,716
10	f_{47}	8,088	10	f_{74}	690	10	f_{54}	2,784	10	f_{100}	5,575
11	f_{48}	7,084	11	f_{80}	7,832	11	f_{57}	925	11	f_{106}	4,063
12	f_{52}	431	12	f_{95}	8,219	12	f_{59}	7,258	12	f_{108}	6,451
13	f_{63}	5,325	13	f_{96}	7,350	13	f_{76}	4,673	13	f_{111}	1,046
14	f_{64}	4,428	14	f_{101}	1,421	14	f_{83}	5,549	14	f_{112}	4,540
15	f_{110}	9,792	15	f_{103}	5,577	15	f_{102}	398	15	f_{113}	4,533
Σ		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**

MUX_5			MUX_6			MUX_7			MUX_8		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_1	9,056	1	f_5	5,035	1	f_{26}	7,223	1	f_{12}	7,415
2	f_{34}	458	2	f_6	3,906	2	f_{56}	2,438	2	f_{28}	1,360
3	f_{38}	2,519	3	f_{10}	6,473	3	f_{65}	2,400	3	f_{37}	667
4	f_{41}	2,599	4	f_{11}	6,533	4	f_{67}	739	4	f_{40}	9,404
5	f_{50}	1,888	5	f_{13}	146	5	f_{73}	3,547	5	f_{43}	7,434
6	f_{51}	5,232	6	f_{14}	5,115	6	f_{81}	7,857	6	f_{84}	4,278
7	f_{62}	815	7	f_{18}	8,525	7	f_{90}	5,676	7	f_{85}	3,020
8	f_{66}	8,104	8	f_{20}	2,324	8	f_{93}	5,374	8	f_{87}	6,314
9	f_{68}	2,772	9	f_{21}	7,464	9	f_{94}	245	9	f_{88}	646
10	f_{70}	9,233	10	f_{22}	3,629	10	f_{97}	3,354	10	f_{92}	834
11	f_{75}	8,215	11	f_{42}	2,946	11	f_{104}	3,741	11	f_{98}	2,485
12	f_{77}	5,523	12	f_{55}	9,259	12	f_{105}	3,043	12	f_{99}	6,642
13	f_{115}	391	13	f_{78}	5,343	13	f_{107}	8,303	13	f_{114}	1,408
14	f_{116}	9,104	14	f_{86}	520	14	f_{109}	6,100	14	f_{117}	7,525
15	f_{120}	2,491	15	f_{91}	1,182	15	f_{119}	8,360	15	f_{118}	8,968
Σ		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**

Table C.1: The best Test Case 3 flow allocation produced in C++ using DP.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$			$\mathbf{MUX_4}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_2	582	1	f_8	2,266	1	f_4	3,140	1	f_{16}	380
2	f_3	4,409	2	f_9	4,150	2	f_{19}	3,685	2	f_{23}	1,091
3	f_7	2,149	3	f_{24}	3,684	3	f_{32}	6,000	3	f_{29}	2,795
4	f_{15}	7,845	4	f_{25}	7,432	4	f_{33}	249	4	f_{61}	6,618
5	f_{17}	4,718	5	f_{39}	3,066	5	f_{35}	5,194	5	f_{69}	4,335
6	f_{27}	2,888	6	f_{53}	1,929	6	f_{36}	7,196	6	f_{72}	9,179
7	f_{30}	3,165	7	f_{58}	8,885	7	f_{44}	6,401	7	f_{79}	4,646
8	f_{31}	7,346	8	f_{60}	2,841	8	f_{46}	8,114	8	f_{82}	5,433
9	f_{45}	151	9	f_{71}	3,059	9	f_{49}	6,835	9	f_{89}	7,716
10	f_{47}	8,088	10	f_{74}	690	10	f_{54}	2,784	10	f_{100}	5,575
11	f_{48}	7,084	11	f_{80}	7,832	11	f_{57}	925	11	f_{106}	4,063
12	f_{52}	431	12	f_{95}	8,219	12	f_{59}	7,258	12	f_{108}	6,451
13	f_{63}	5,325	13	f_{96}	7,350	13	f_{76}	4,673	13	f_{111}	1,046
14	f_{64}	4,428	14	f_{101}	1,421	14	f_{83}	5,549	14	f_{112}	4,540
15	f_{110}	9,792	15	f_{103}	5,577	15	f_{102}	398	15	f_{113}	4,533
Σ		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**

$\mathbf{MUX_5}$			$\mathbf{MUX_6}$			$\mathbf{MUX_7}$			$\mathbf{MUX_8}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_1	9,056	1	f_5	5,035	1	f_{26}	7,223	1	f_{12}	7,415
2	f_{34}	458	2	f_6	3,906	2	f_{56}	2,438	2	f_{28}	1,360
3	f_{38}	2,519	3	f_{10}	6,473	3	f_{65}	2,400	3	f_{37}	667
4	f_{41}	2,599	4	f_{11}	6,533	4	f_{67}	739	4	f_{40}	9,404
5	f_{50}	1,888	5	f_{13}	146	5	f_{73}	3,547	5	f_{43}	7,434
6	f_{51}	5,232	6	f_{14}	5,115	6	f_{81}	7,857	6	f_{84}	4,278
7	f_{62}	815	7	f_{18}	8,525	7	f_{90}	5,676	7	f_{85}	3,020
8	f_{66}	8,104	8	f_{20}	2,324	8	f_{93}	5,374	8	f_{87}	6,314
9	f_{68}	2,772	9	f_{21}	7,464	9	f_{94}	245	9	f_{88}	646
10	f_{70}	9,233	10	f_{22}	3,629	10	f_{97}	3,354	10	f_{92}	834
11	f_{75}	8,215	11	f_{42}	2,946	11	f_{104}	3,741	11	f_{98}	2,485
12	f_{77}	5,523	12	f_{55}	9,259	12	f_{105}	3,043	12	f_{99}	6,642
13	f_{115}	391	13	f_{78}	5,343	13	f_{107}	8,303	13	f_{114}	1,408
14	f_{116}	9,104	14	f_{86}	520	14	f_{109}	6,100	14	f_{117}	7,525
15	f_{120}	2,491	15	f_{91}	1,182	15	f_{119}	8,360	15	f_{118}	8,968
Σ		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**

Table C.2: The best Test Case 3 flow allocation produced in Matlab using DP.

C.2 Greedy Heuristic

MUX_1			MUX_2			MUX_3			MUX_4		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{110}	9,792	1	f_{40}	9,404	1	f_{55}	9,259	1	f_{70}	9,233
2	f_{66}	8,104	2	f_{46}	8,114	2	f_{75}	8,215	2	f_{95}	8,219
3	f_{43}	7,434	3	f_{89}	7,716	3	f_{15}	7,845	3	f_{47}	8,088
4	f_{36}	7,196	4	f_{12}	7,415	4	f_{59}	7,258	4	f_{48}	7,084
5	f_{49}	6,835	5	f_{108}	6,451	5	f_{11}	6,533	5	f_{10}	6,473
6	f_{82}	5,433	6	f_{103}	5,577	6	f_{100}	5,575	6	f_{90}	5,676
7	f_{14}	5,115	7	f_{93}	5,374	7	f_{78}	5,343	7	f_{35}	5,194
8	f_{79}	4,646	8	f_{69}	4,335	8	f_{64}	4,428	8	f_{113}	4,533
9	f_{73}	3,547	9	f_{106}	4,063	9	f_{104}	3,741	9	f_{22}	3,629
10	f_{97}	3,354	10	f_{85}	3,020	10	f_{71}	3,059	10	f_{30}	3,165
11	f_{41}	2,599	11	f_{38}	2,519	11	f_{60}	2,841	11	f_{29}	2,795
12	f_{65}	2,400	12	f_{56}	2,438	12	f_{7}	2,149	12	f_{8}	2,266
13	f_{92}	834	13	f_{111}	1,046	13	f_{28}	1,360	13	f_{91}	1,182
14	f_{62}	815	14	f_{88}	646	14	f_{34}	458	14	f_{86}	520
15	f_{16}	380	15	f_{94}	245	15	f_{102}	398	15	f_{52}	431
Σ		**68,484**	**Σ**		**68,363**	**Σ**		**68,462**	**Σ**		**68,488**

MUX_5			MUX_6			MUX_7			MUX_8		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{72}	9,179	1	f_{116}	9,104	1	f_{1}	9,056	1	f_{118}	8,968
2	f_{107}	8,303	2	f_{119}	8,360	2	f_{18}	8,525	2	f_{58}	8,885
3	f_{80}	7,832	3	f_{81}	7,857	3	f_{117}	7,525	3	f_{21}	7,464
4	f_{96}	7,350	4	f_{26}	7,223	4	f_{25}	7,432	4	f_{31}	7,346
5	f_{87}	6,314	5	f_{61}	6,618	5	f_{99}	6,642	5	f_{44}	6,401
6	f_{109}	6,100	6	f_{83}	5,549	6	f_{77}	5,523	6	f_{32}	6,000
7	f_{17}	4,718	7	f_{51}	5,232	7	f_{63}	5,325	7	f_{5}	5,035
8	f_{76}	4,673	8	f_{112}	4,540	8	f_{3}	4,409	8	f_{84}	4,278
9	f_{19}	3,685	9	f_{24}	3,684	9	f_{6}	3,906	9	f_{9}	4,150
10	f_{4}	3,140	10	f_{39}	3,066	10	f_{105}	3,043	10	f_{42}	2,946
11	f_{54}	2,784	11	f_{27}	2,888	11	f_{68}	2,772	11	f_{120}	2,491
12	f_{20}	2,324	12	f_{53}	1,929	12	f_{50}	1,888	12	f_{98}	2,485
13	f_{23}	1,091	13	f_{114}	1,408	13	f_{101}	1,421	13	f_{57}	925
14	f_{2}	582	14	f_{74}	690	14	f_{37}	667	14	f_{67}	739
15	f_{115}	391	15	f_{13}	146	15	f_{45}	151	15	f_{33}	249
Σ		**68,466**	**Σ**		**68,294**	**Σ**		**68,285**	**Σ**		**68,362**

Table C.3: The best Test Case 3 flow allocation produced in C++ using GH.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$			$\mathbf{MUX_4}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{110}	9,792	1	f_{40}	9,404	1	f_{55}	9,259	1	f_{70}	9,233
2	f_{66}	8,104	2	f_{46}	8,114	2	f_{75}	8,215	2	f_{95}	8,219
3	f_{43}	7,434	3	f_{89}	7,716	3	f_{15}	7,845	3	f_{47}	8,088
4	f_{36}	7,196	4	f_{12}	7,415	4	f_{59}	7,258	4	f_{48}	7,084
5	f_{49}	6,835	5	f_{108}	6,451	5	f_{11}	6,533	5	f_{10}	6,473
6	f_{82}	5,433	6	f_{103}	5,577	6	f_{100}	5,575	6	f_{90}	5,676
7	f_{14}	5,115	7	f_{93}	5,374	7	f_{78}	5,343	7	f_{35}	5,194
8	f_{79}	4,646	8	f_{69}	4,335	8	f_{64}	4,428	8	f_{113}	4,533
9	f_{73}	3,547	9	f_{106}	4,063	9	f_{104}	3,741	9	f_{22}	3,629
10	f_{97}	3,354	10	f_{85}	3,020	10	f_{71}	3,059	10	f_{30}	3,165
11	f_{41}	2,599	11	f_{38}	2,519	11	f_{60}	2,841	11	f_{29}	2,795
12	f_{65}	2,400	12	f_{56}	2,438	12	f_{7}	2,149	12	f_{8}	2,266
13	f_{92}	834	13	f_{111}	1,046	13	f_{28}	1,360	13	f_{91}	1,182
14	f_{62}	815	14	f_{88}	646	14	f_{34}	458	14	f_{86}	520
15	f_{16}	380	15	f_{94}	245	15	f_{102}	398	15	f_{52}	431
Σ		**68,484**	**Σ**		**68,363**	**Σ**		**68,462**	**Σ**		**68,488**

$\mathbf{MUX_5}$			$\mathbf{MUX_6}$			$\mathbf{MUX_7}$			$\mathbf{MUX_8}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{72}	9,179	1	f_{116}	9,104	1	f_{1}	9,056	1	f_{118}	8,968
2	f_{107}	8,303	2	f_{119}	8,360	2	f_{18}	8,525	2	f_{58}	8,885
3	f_{80}	7,832	3	f_{81}	7,857	3	f_{117}	7,525	3	f_{21}	7,464
4	f_{96}	7,350	4	f_{26}	7,223	4	f_{25}	7,432	4	f_{31}	7,346
5	f_{87}	6,314	5	f_{61}	6,618	5	f_{99}	6,642	5	f_{44}	6,401
6	f_{109}	6,100	6	f_{83}	5,549	6	f_{77}	5,523	6	f_{32}	6,000
7	f_{17}	4,718	7	f_{51}	5,232	7	f_{63}	5,325	7	f_{5}	5,035
8	f_{76}	4,673	8	f_{112}	4,540	8	f_{3}	4,409	8	f_{84}	4,278
9	f_{19}	3,685	9	f_{24}	3,684	9	f_{6}	3,906	9	f_{9}	4,150
10	f_{4}	3,140	10	f_{39}	3,066	10	f_{105}	3,043	10	f_{42}	2,946
11	f_{54}	2,784	11	f_{27}	2,888	11	f_{68}	2,772	11	f_{120}	2,491
12	f_{20}	2,324	12	f_{53}	1,929	12	f_{50}	1,888	12	f_{98}	2,485
13	f_{23}	1,091	13	f_{114}	1,408	13	f_{101}	1,421	13	f_{57}	925
14	f_{2}	582	14	f_{74}	690	14	f_{37}	667	14	f_{67}	739
15	f_{115}	391	15	f_{13}	146	15	f_{45}	151	15	f_{33}	249
Σ		**68,466**	**Σ**		**68,294**	**Σ**		**68,285**	**Σ**		**68,362**

Table C.4: The best Test Case 3 flow allocation produced in Matlab using GH.

C.3 Integer Linear Programming

MUX_1			MUX_2			MUX_3			MUX_4		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_4	3,140	1	f_1	9,056	1	f_{21}	7,464	1	f_2	582
2	f_7	2,149	2	f_6	3,906	2	f_{25}	7,432	2	f_{10}	6,473
3	f_8	2,266	3	f_{19}	3,685	3	f_{32}	6,000	3	f_{11}	6,533
4	f_{12}	7,415	4	f_{38}	2,519	4	f_{34}	458	4	f_{26}	7,223
5	f_{16}	380	5	f_{51}	5,232	5	f_{37}	667	5	f_{29}	2,795
6	f_{18}	8,525	6	f_{54}	2,784	6	f_{43}	7,434	6	f_{30}	3,165
7	f_{27}	2,888	7	f_{58}	8,885	7	f_{44}	6,401	7	f_{31}	7,346
8	f_{28}	1,360	8	f_{64}	4,428	8	f_{57}	925	8	f_{39}	3,066
9	f_{40}	9,404	9	f_{68}	2,772	9	f_{70}	9,233	9	f_{41}	2,599
10	f_{66}	8,104	10	f_{80}	7,832	10	f_{87}	6,314	10	f_{42}	2,946
11	f_{71}	3,059	11	f_{85}	3,020	11	f_{92}	834	11	f_{59}	7,258
12	f_{75}	8,215	12	f_{97}	3,354	12	f_{96}	7,350	12	f_{61}	6,618
13	f_{81}	7,857	13	f_{105}	3,043	13	f_{102}	398	13	f_{99}	6,642
14	f_{91}	1,182	14	f_{115}	391	14	f_{108}	6,451	14	f_{101}	1,421
15	f_{120}	2,491	15	f_{117}	7,525	15	f_{111}	1,046	15	f_{104}	3,741
Σ		**68,435**	**Σ**		**68,432**	**Σ**		**68,407**	**Σ**		**68,408**

MUX_5			MUX_6			MUX_7			MUX_8		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{15}	7,845	1	f_9	4,150	1	f_3	4,409	1	f_{13}	146
2	f_{23}	1,091	2	f_{20}	2,324	2	f_5	5,035	2	f_{33}	249
3	f_{36}	7,196	3	f_{22}	3,629	3	f_{14}	5,115	3	f_{45}	151
4	f_{47}	8,088	4	f_{48}	7,084	4	f_{17}	4,718	4	f_{46}	8,114
5	f_{50}	1,888	5	f_{52}	431	5	f_{24}	3,684	5	f_{49}	6,835
6	f_{53}	1,929	6	f_{55}	9,259	6	f_{35}	5,194	6	f_{56}	2,438
7	f_{65}	2,400	7	f_{62}	815	7	f_{60}	2,841	7	f_{74}	690
8	f_{67}	739	8	f_{72}	9,179	8	f_{63}	5,325	8	f_{77}	5,523
9	f_{86}	520	9	f_{73}	3,547	9	f_{69}	4,335	9	f_{78}	5,343
10	f_{89}	7,716	10	f_{84}	4,278	10	f_{76}	4,673	10	f_{82}	5,433
11	f_{98}	2,485	11	f_{88}	646	11	f_{79}	4,646	11	f_{83}	5,549
12	f_{112}	4,540	12	f_{94}	245	12	f_{93}	5,374	12	f_{90}	5,676
13	f_{113}	4,533	13	f_{100}	5,575	13	f_{103}	5,577	13	f_{95}	8,219
14	f_{116}	9,104	14	f_{107}	8,303	14	f_{109}	6,100	14	f_{106}	4,063
15	f_{119}	8,360	15	f_{118}	8,968	15	f_{114}	1,408	15	f_{110}	9,792
Σ		**68,434**	**Σ**		**68,433**	**Σ**		**68,434**	**Σ**		**68,221**

Table C.5: The best Test Case 3 flow allocation produced in C++ using ILP.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$			$\mathbf{MUX_4}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_6	3,906	1	f_3	4,409	1	f_{16}	380	1	f_{23}	1,091
2	f_9	4,150	2	f_4	3,140	2	f_{33}	249	2	f_{28}	1,360
3	f_{10}	6,473	3	f_{14}	5,115	3	f_{38}	2,519	3	f_{41}	2,599
4	f_{17}	4,718	4	f_{15}	7,845	4	f_{42}	2,946	4	f_{44}	6,401
5	f_{18}	8,525	5	f_{39}	3,066	5	f_{48}	7,084	5	f_{50}	1,888
6	f_{19}	3,685	6	f_{40}	9,404	6	f_{49}	6,835	6	f_{55}	9,259
7	f_{22}	3,629	7	f_{52}	431	7	f_{54}	2,784	7	f_{70}	9,233
8	f_{24}	3,684	8	f_{62}	815	8	f_{60}	2,841	8	f_{72}	9,179
9	f_{56}	2,438	9	f_{69}	4,335	9	f_{63}	5,325	9	f_{82}	5,433
10	f_{73}	3,547	10	f_{71}	3,059	10	f_{77}	5,523	10	f_{86}	520
11	f_{84}	4,278	11	f_{74}	690	11	f_{81}	7,857	11	f_{87}	6,314
12	f_{95}	8,219	12	f_{76}	4,673	12	f_{93}	5,374	12	f_{108}	6,451
13	f_{97}	3,354	13	f_{79}	4,646	13	f_{99}	6,642	13	f_{109}	6,100
14	f_{104}	3,741	14	f_{80}	7,832	14	f_{112}	4,540	14	f_{111}	1,046
15	f_{106}	4,063	15	f_{118}	8,968	15	f_{117}	7,525	15	f_{114}	1,408
Σ		**68,410**	**Σ**		**68,428**	**Σ**		**68,424**	**Σ**		**68,282**

$\mathbf{MUX_5}$			$\mathbf{MUX_6}$			$\mathbf{MUX_7}$			$\mathbf{MUX_8}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_5	5,035	1	f_2	582	1	f_1	9,056	1	f_{12}	7,415
2	f_7	2,149	2	f_{27}	2,888	2	f_{46}	8,114	2	f_{13}	146
3	f_8	2,266	3	f_{30}	3,165	3	f_{47}	8,088	3	f_{20}	2,324
4	f_{11}	6,533	4	f_{32}	6,000	4	f_{57}	925	4	f_{21}	7,464
5	f_{35}	5,194	5	f_{37}	667	5	f_{58}	8,885	5	f_{25}	7,432
6	f_{36}	7,196	6	f_{53}	1,929	6	f_{67}	739	6	f_{26}	7,223
7	f_{51}	5,232	7	f_{61}	6,618	7	f_{75}	8,215	7	f_{29}	2,795
8	f_{64}	4,428	8	f_{66}	8,104	8	f_{85}	3,020	8	f_{31}	7,346
9	f_{78}	5,343	9	f_{68}	2,772	9	f_{92}	834	9	f_{34}	458
10	f_{88}	646	10	f_{83}	5,549	10	f_{94}	245	10	f_{43}	7,434
11	f_{90}	5,676	11	f_{89}	7,716	11	f_{101}	1,421	11	f_{45}	151
12	f_{100}	5,575	12	f_{98}	2,485	12	f_{102}	398	12	f_{59}	7,258
13	f_{103}	5,577	13	f_{116}	9,104	13	f_{107}	8,303	13	f_{65}	2,400
14	f_{105}	3,043	14	f_{119}	8,360	14	f_{110}	9,792	14	f_{91}	1,182
15	f_{113}	4,533	15	f_{120}	2,491	15	f_{115}	391	15	f_{96}	7,350
Σ		**68,426**	**Σ**		**68,430**	**Σ**		**68,426**	**Σ**		**68,378**

Table C.6: The best Test Case 3 flow allocation produced in Matlab using ILP.

C.4 Modified Differential Evolution

MUX_1			MUX_2			MUX_3			MUX_4		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{52}	431	1	f_{99}	6,642	1	f_{40}	9,404	1	f_{111}	1,046
2	f_{116}	9,104	2	f_{113}	4,533	2	f_{4}	3,140	2	f_{5}	5,035
3	f_{65}	2,400	3	f_{71}	3,059	3	f_{64}	4,428	3	f_{49}	6,835
4	f_{77}	5,523	4	f_{83}	5,549	4	f_{2}	582	4	f_{59}	7,258
5	f_{35}	5,194	5	f_{54}	2,784	5	f_{103}	5,577	5	f_{117}	7,525
6	f_{19}	3,685	6	f_{94}	245	6	f_{45}	151	6	f_{1}	9,056
7	f_{10}	6,473	7	f_{93}	5,374	7	f_{75}	8,215	7	f_{87}	6,314
8	f_{66}	8,104	8	f_{120}	2,491	8	f_{78}	5,343	8	f_{39}	3,066
9	f_{69}	4,335	9	f_{104}	3,741	9	f_{68}	2,772	9	f_{16}	380
10	f_{41}	2,599	10	f_{15}	7,845	10	f_{76}	4,673	10	f_{89}	7,716
11	f_{79}	4,646	11	f_{70}	9,233	11	f_{34}	458	11	f_{91}	1,182
12	f_{23}	1,091	12	f_{46}	8,114	12	f_{18}	8,525	12	f_{106}	4,063
13	f_{8}	2,266	13	f_{63}	5,325	13	f_{58}	8,885	13	f_{73}	3,547
14	f_{80}	7,832	14	f_{86}	520	14	f_{82}	5,433	14	f_{51}	5,232
15	f_{17}	4,718	15	f_{42}	2,946	15	f_{62}	815	15	f_{13}	146
Σ		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**

MUX_5			MUX_6			MUX_7			MUX_8		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{88}	646	1	f_{27}	2,888	1	f_{60}	2,841	1	f_{96}	7,350
2	f_{84}	4,278	2	f_{21}	7,464	2	f_{36}	7,196	2	f_{119}	8,360
3	f_{81}	7,857	3	f_{47}	8,088	3	f_{20}	2,324	3	f_{37}	667
4	f_{6}	3,906	4	f_{56}	2,438	4	f_{11}	6,533	4	f_{61}	6,618
5	f_{90}	5,676	5	f_{22}	3,629	5	f_{112}	4,540	5	f_{110}	9,792
6	f_{24}	3,684	6	f_{55}	9,259	6	f_{25}	7,432	6	f_{29}	2,795
7	f_{95}	8,219	7	f_{32}	6,000	7	f_{105}	3,043	7	f_{108}	6,451
8	f_{3}	4,409	8	f_{115}	391	8	f_{74}	690	8	f_{28}	1,360
9	f_{109}	6,100	9	f_{114}	1,408	9	f_{98}	2,485	9	f_{85}	3,020
10	f_{92}	834	10	f_{100}	5,575	10	f_{12}	7,415	10	f_{33}	249
11	f_{53}	1,929	11	f_{107}	8,303	11	f_{67}	739	11	f_{57}	925
12	f_{102}	398	12	f_{38}	2,519	12	f_{101}	1,421	12	f_{14}	5,115
13	f_{31}	7,346	13	f_{7}	2,149	13	f_{26}	7,223	13	f_{72}	9,179
14	f_{9}	4,150	14	f_{50}	1,888	14	f_{48}	7,084	14	f_{30}	3,165
15	f_{118}	8,968	15	f_{44}	6,401	15	f_{43}	7,434	15	f_{97}	3,354
Σ		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**

Table C.7: The best Test Case 3 flow allocation produced in C++ using the MDE.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$			$\mathbf{MUX_4}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{81}	7,857	1	f_{13}	146	1	f_{5}	5,035	1	f_{41}	2,599
2	f_{108}	6,451	2	f_{58}	8,885	2	f_{20}	2,324	2	f_{72}	9,179
3	f_{67}	739	3	f_{8}	2,266	3	f_{36}	7,196	3	f_{94}	245
4	f_{49}	6,835	4	f_{106}	4,063	4	f_{52}	431	4	f_{93}	5,374
5	f_{26}	7,223	5	f_{119}	8,360	5	f_{2}	582	5	f_{22}	3,629
6	f_{76}	4,673	6	f_{98}	2,485	6	f_{77}	5,523	6	f_{9}	4,150
7	f_{100}	5,575	7	f_{23}	1,091	7	f_{59}	7,258	7	f_{86}	520
8	f_{68}	2,772	8	f_{38}	2,519	8	f_{15}	7,845	8	f_{39}	3,066
9	f_{55}	9,259	9	f_{27}	2,888	9	f_{12}	7,415	9	f_{117}	7,525
10	f_{60}	2,841	10	f_{89}	7,716	10	f_{3}	4,409	10	f_{116}	9,104
11	f_{4}	3,140	11	f_{63}	5,325	11	f_{91}	1,182	11	f_{80}	7,832
12	f_{88}	646	12	f_{61}	6,618	12	f_{99}	6,642	12	f_{11}	6,533
13	f_{31}	7,346	13	f_{118}	8,968	13	f_{112}	4,540	13	f_{42}	2,946
14	f_{33}	249	14	f_{6}	3,906	14	f_{24}	3,684	14	f_{84}	4,278
15	f_{29}	2,795	15	f_{30}	3,165	15	f_{69}	4,335	15	f_{101}	1,421
Σ		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**

$\mathbf{MUX_5}$			$\mathbf{MUX_6}$			$\mathbf{MUX_7}$			$\mathbf{MUX_8}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{87}	6,314	1	f_{75}	8,215	1	f_{45}	151	1	f_{74}	690
2	f_{18}	8,525	2	f_{62}	815	2	f_{47}	8,088	2	f_{115}	391
3	f_{1}	9,056	3	f_{97}	3,354	3	f_{103}	5,577	3	f_{83}	5,549
4	f_{34}	458	4	f_{70}	9,233	4	f_{79}	4,646	4	f_{25}	7,432
5	f_{65}	2,400	5	f_{92}	834	5	f_{17}	4,718	5	f_{85}	3,020
6	f_{46}	8,114	6	f_{43}	7,434	6	f_{113}	4,533	6	f_{95}	8,219
7	f_{102}	398	7	f_{28}	1,360	7	f_{10}	6,473	7	f_{37}	667
8	f_{56}	2,438	8	f_{35}	5,194	8	f_{14}	5,115	8	f_{48}	7,084
9	f_{7}	2,149	9	f_{90}	5,676	9	f_{73}	3,547	9	f_{53}	1,929
10	f_{120}	2,491	10	f_{109}	6,100	10	f_{66}	8,104	10	f_{96}	7,350
11	f_{50}	1,888	11	f_{44}	6,401	11	f_{64}	4,428	11	f_{110}	9,792
12	f_{107}	8,303	12	f_{104}	3,741	12	f_{51}	5,232	12	f_{111}	1,046
13	f_{78}	5,343	13	f_{19}	3,685	13	f_{16}	380	13	f_{105}	3,043
14	f_{21}	7,464	14	f_{57}	925	14	f_{114}	1,408	14	f_{54}	2,784
15	f_{71}	3,059	15	f_{82}	5,433	15	f_{32}	6,000	15	f_{40}	9,404
Σ		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**	**Σ**		**68,400**

Table C.8: The best Test Case 3 flow allocation produced in Matlab using the MDE.

C.5 Reinforcement Learning

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$			$\mathbf{MUX_4}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{110}	9,792	1	f_{58}	8,885	1	f_{66}	8,104	1	f_{116}	9,104
2	f_{40}	9,404	2	f_{18}	8,525	2	f_{47}	8,088	2	f_{81}	7,857
3	f_{55}	9,259	3	f_{119}	8,360	3	f_{15}	7,845	3	f_{43}	7,434
4	f_{70}	9,233	4	f_{107}	8,303	4	f_{80}	7,832	4	f_{96}	7,350
5	f_{72}	9,179	5	f_{95}	8,219	5	f_{89}	7,716	5	f_{59}	7,258
6	f_{1}	9,056	6	f_{75}	8,215	6	f_{117}	7,525	6	f_{26}	7,223
7	f_{118}	8,968	7	f_{46}	8,114	7	f_{21}	7,464	7	f_{108}	6,451
8	f_{101}	1,421	8	f_{90}	5,676	8	f_{25}	7,432	8	f_{65}	2,400
9	f_{86}	520	9	f_{92}	834	9	f_{91}	1,182	9	f_{20}	2,324
10	f_{102}	398	10	f_{67}	739	10	f_{23}	1,091	10	f_{8}	2,266
11	f_{16}	380	11	f_{37}	667	11	f_{111}	1,046	11	f_{7}	2,149
12	f_{33}	249	12	f_{2}	582	12	f_{57}	925	12	f_{53}	1,929
13	f_{94}	245	13	f_{34}	458	13	f_{62}	815	13	f_{50}	1,888
14	f_{45}	151	14	f_{52}	431	14	f_{74}	690	14	f_{114}	1,408
15	f_{13}	146	15	f_{115}	391	15	f_{88}	646	15	f_{28}	1,360
Σ		**68,401**	**Σ**		**68,399**	**Σ**		**68,401**	**Σ**		**68,401**

$\mathbf{MUX_5}$			$\mathbf{MUX_6}$			$\mathbf{MUX_7}$			$\mathbf{MUX_8}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{12}	7,415	1	f_{61}	6,618	1	f_{11}	6,533	1	f_{10}	6,473
2	f_{31}	7,346	2	f_{44}	6,401	2	f_{83}	5,549	2	f_{35}	5,194
3	f_{36}	7,196	3	f_{87}	6,314	3	f_{77}	5,523	3	f_{5}	5,035
4	f_{48}	7,084	4	f_{109}	6,100	4	f_{82}	5,433	4	f_{76}	4,673
5	f_{49}	6,835	5	f_{32}	6,000	5	f_{93}	5,374	5	f_{79}	4,646
6	f_{99}	6,642	6	f_{103}	5,577	6	f_{78}	5,343	6	f_{112}	4,540
7	f_{17}	4,718	7	f_{100}	5,575	7	f_{63}	5,325	7	f_{113}	4,533
8	f_{39}	3,066	8	f_{51}	5,232	8	f_{14}	5,115	8	f_{64}	4,428
9	f_{29}	2,795	9	f_{71}	3,059	9	f_{19}	3,685	9	f_{3}	4,409
10	f_{68}	2,772	10	f_{105}	3,043	10	f_{24}	3,684	10	f_{69}	4,335
11	f_{41}	2,599	11	f_{85}	3,020	11	f_{22}	3,629	11	f_{84}	4,278
12	f_{38}	2,519	12	f_{42}	2,946	12	f_{73}	3,547	12	f_{9}	4,150
13	f_{120}	2,491	13	f_{27}	2,888	13	f_{97}	3,354	13	f_{106}	4,063
14	f_{98}	2,485	14	f_{60}	2,841	14	f_{30}	3,165	14	f_{6}	3,906
15	f_{56}	2,438	15	f_{54}	2,784	15	f_{4}	3,140	15	f_{104}	3,741
Σ		**68,401**	**Σ**		**68,398**	**Σ**		**68,399**	**Σ**		**68,404**

Table C.9: The best Test Case 3 flow allocation produced in C++ using RL.

$\mathbf{MUX_1}$			$\mathbf{MUX_2}$			$\mathbf{MUX_3}$			$\mathbf{MUX_4}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{110}	9,792	1	f_{116}	9,104	1	f_{75}	8,215	1	f_{80}	7,832
2	f_{40}	9,404	2	f_{58}	8,885	2	f_{66}	8,104	2	f_{43}	7,434
3	f_{55}	9,259	3	f_{18}	8,525	3	f_{81}	7,857	3	f_{25}	7,432
4	f_{70}	9,233	4	f_{119}	8,360	4	f_{15}	7,845	4	f_{31}	7,346
5	f_{72}	9,179	5	f_{107}	8,303	5	f_{89}	7,716	5	f_{59}	7,258
6	f_{1}	9,056	6	f_{95}	8,219	6	f_{117}	7,525	6	f_{26}	7,223
7	f_{118}	8,968	7	f_{46}	8,114	7	f_{21}	7,464	7	f_{36}	7,196
8	f_{101}	1,421	8	f_{17}	4,718	8	f_{96}	7,350	8	f_{97}	3,354
9	f_{86}	520	9	f_{57}	925	9	f_{91}	1,182	9	f_{20}	2,324
10	f_{102}	398	10	f_{67}	739	10	f_{23}	1,091	10	f_{8}	2,266
11	f_{16}	380	11	f_{88}	646	11	f_{111}	1,046	11	f_{7}	2,149
12	f_{33}	249	12	f_{2}	582	12	f_{92}	834	12	f_{53}	1,929
13	f_{94}	245	13	f_{34}	458	13	f_{62}	815	13	f_{50}	1,888
14	f_{45}	151	14	f_{52}	431	14	f_{74}	690	14	f_{114}	1,408
15	f_{13}	146	15	f_{115}	391	15	f_{37}	667	15	f_{28}	1,360
Σ		**68,401**	**Σ**		**68,400**	**Σ**		**68,401**	**Σ**		**68,399**

$\mathbf{MUX_5}$			$\mathbf{MUX_6}$			$\mathbf{MUX_7}$			$\mathbf{MUX_8}$		
#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**	#	**Flow**	**[B]**
1	f_{47}	8,088	1	f_{11}	6,533	1	f_{90}	5,676	1	f_{100}	5,575
2	f_{12}	7,415	2	f_{10}	6,473	2	f_{103}	5,577	2	f_{78}	5,343
3	f_{48}	7,084	3	f_{108}	6,451	3	f_{83}	5,549	3	f_{51}	5,232
4	f_{49}	6,835	4	f_{44}	6,401	4	f_{77}	5,523	4	f_{35}	5,194
5	f_{99}	6,642	5	f_{87}	6,314	5	f_{82}	5,433	5	f_{76}	4,673
6	f_{61}	6,618	6	f_{109}	6,100	6	f_{93}	5,374	6	f_{112}	4,540
7	f_{14}	5,115	7	f_{32}	6,000	7	f_{63}	5,325	7	f_{113}	4,533
8	f_{27}	2,888	8	f_{73}	3,547	8	f_{5}	5,035	8	f_{64}	4,428
9	f_{54}	2,784	9	f_{30}	3,165	9	f_{79}	4,646	9	f_{3}	4,409
10	f_{41}	2,599	10	f_{105}	3,043	10	f_{19}	3,685	10	f_{69}	4,335
11	f_{38}	2,519	11	f_{85}	3,020	11	f_{24}	3,684	11	f_{84}	4,278
12	f_{120}	2,491	12	f_{42}	2,946	12	f_{22}	3,629	12	f_{9}	4,150
13	f_{98}	2,485	13	f_{60}	2,841	13	f_{4}	3,140	13	f_{106}	4,063
14	f_{56}	2,438	14	f_{29}	2,795	14	f_{39}	3,066	14	f_{6}	3,906
15	f_{65}	2,400	15	f_{68}	2,772	15	f_{71}	3,059	15	f_{104}	3,741
Σ		**68,401**	**Σ**		**68,401**	**Σ**		**68,401**	**Σ**		**68,400**

Table C.10: The best Test Case 3 flow allocation produced in Matlab using RL.