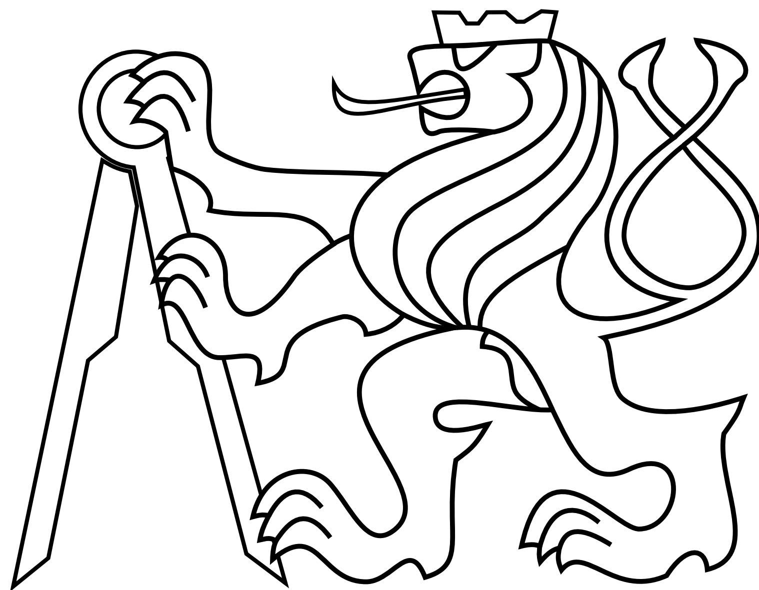
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR THESIS



Kateřina Brejchová

Heuristics for Periodic Scheduling

Department of Cybernetics
Thesis supervisor: **Mgr. Marek Vlk**
May 2019

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Brejchová Kateřina** Personal ID number: **465816**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Heuristics for Periodic Scheduling

Bachelor’s thesis title in Czech:

Heuristiky pro periodické rozvrhování

Guidelines:

1. Familiarize yourself with Ethernet-based periodic scheduling.
2. Study heuristic methods for Time-Triggered scheduling such as described in [1, 2, 3].
3. Propose periodic scheduling algorithms.
4. Implement several different versions of these algorithms.
5. Evaluate proposed methods on benchmark instances.

Bibliography / sources:

[1] Demeulemeester Erik, Herroelen Willy - Project Scheduling: A Research Handbook
[2] Clément Pira, Christian Artigues - Line search method for solving a non-preemptive strictly periodic scheduling problem - Springer Science+Business Media New York 2014
[3] A. Minaeva, D. Roy, B. Akesson, Z. Hanzalek, S. Chakraborty. Efficient Heuristic Approach for Control Performance Optimization in Time-Triggered Periodic Scheduling. IEEE Transactions on Computers, submitted.

Name and workplace of bachelor’s thesis supervisor:

Mgr. Marek Vlk, Optimization, CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **07.01.2019** Deadline for bachelor thesis submission: **24.05.2019**

Assignment valid until: **30.09.2020**

Mgr. Marek Vlk	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Author statement for undergraduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date .... Signature ....

Acknowledgements

I would like to thank Mr. Marek Vlk for supervising this thesis, and members of the Industrial Informatics Research Center for their support. Also, I would like to thank my partner and my family for their never-ending support during my studies.

Abstract

In the past decades, the usage of electronic communication systems that influence all areas of human activities massively increased. Low cost and high effectiveness allow it to be used widely. The massive usage of such systems in different domains such as industry, smart cities, etc. calls for developing scheduling methods that are fast, adjustable and reliable. In this thesis, we formalize the highly critical periodic scheduling problem and design a Java-based framework that allows easy testing of different scheduling methods. The main contribution of this thesis is several heuristics suitable for strictly periodic network communication and comparison of their performance on generated instances.

Abstrakt

V posledních několika desetiletích se masivně zvýšilo využívání elektronických komunikačních systémů, které ovlivňují všechny oblasti lidské činnosti. Díky nízkým nákladům a vysoké efektivitě mohou být tyto modely široce rozšířené. Masivní využívání takových systémů v různých doménách jako je průmysl, chytrá města (smart cities), atd., volá po vývoji rozvrhovacích metod, které jsou rychlé, přizpůsobivé a spolehlivé. V této práci formalizujeme problém vysoce kritického periodického rozvrhování. Dále navrhujeme aplikaci v Javě, která umožnuje jednoduché testování různých rozvrhovacích metod. Hlavní přínos této práce spočívá v několika heuristikách vhodných pro striktně periodické rozvrhování komunikace a porovnání jejich výkonnosti na vygenerovaných instancích.

Contents

List of Figures iii

List of Tables iv

1 Introduction 1

2 Background 3
2.1 Ethernet Scheduling . . . 3
2.2 Heuristic Algorithms . . . 3
2.2.1 Constructive Heuristics . . . 4
2.2.2 Improvement Heuristics . . . 5
2.3 Exact Algorithms . . . 5
2.3.1 Constraint Programming . . . 5
2.3.2 Integer Linear Programming . . . 7
2.4 Related Work . . . 8

3 Problem Statement 9
3.1 Network Model . . . 9
3.2 Communication Model . . . 9
3.3 Scheduling Problem . . . 11
3.4 Scheduling Objective . . . 12

4 Proposed Solution 13
4.1 Schedulability Conditions . . . 13
4.2 Integer Linear Programming . . . 13
4.3 One Pass Heuristics . . . 14
4.3.1 First Fit Methods . . . 14
4.3.2 Priority Queue Ordering . . . 16
4.3.3 Complexity Analysis . . . 17
4.3.4 Random Heuristic . . . 19
4.4 Multiple Pass Heuristics . . . 19
4.4.1 Backjumping . . . 20
4.4.2 Backmarking . . . 20
4.4.3 Heuristic Enhancements . . . 20

5 Program 23
5.1 User Manual . . . 24

6 Experiments 27
6.1 Experiments Setup . . . 27
6.2 Results . . . 31
6.2.1 Structure of Generated Instances . . . 31
6.2.2 One Pass Heuristics . . . 32
6.2.3 Multiple Pass Heuristics . . . 34
6.3 Discussion . . . 35

7 Conclusion **37**

References **39**

A CD Content **41**

B List of Abbreviations **43**

C Average and Maximal Utilization of Links **45**

D Percentage of Scheduled Instances **47**

E Success Rate Based on Link Utilization **49**

F Average Running Time **51**

G Best Objective Value Score **53**

List of Figures

1 Visualization of the network and communication model 9
2 Gantt chart of the sample instance 10
3 Workflow of the proposed program 23
4 Manual step 1 - Choose the file 25
5 Manual step 2 - Choose the solver 25
6 Manual step 3 - View the topology 25
7 Manual step 4 - View the schedule 26
8 Manual step 5 - View the instance 26
9 Sample middle sized TREE and RING topology 27
10 Middle sized LINE topology 28
11 Link utilization of all topologies 32
12 Average and maximal link utilization of small and middle sized topologies . . 45
13 Average and maximal link utilization of large topologies 46
14 Scheduled instances for small and middle sized topologies 47
15 Scheduled instances for large and all topologies 48
16 The percentage of scheduled instances based on average link utilization 49
17 The percentage of scheduled instances based on maximal link utilization . . . 50
18 Average running time for small and medium topologies 51
19 Average running time for large topologies 52
20 Best objective score for small and middle sized topologies 53
21 Best objective score for large topologies 54

List of Tables

1 List of complement criteria for s_j and FFS scheduling 17
2 List of complement criteria for $s_j^{e_k}$ and FFSI scheduling 18
3 Period sets 28
4 Parameters of different topologies 29
5 Number of scheduled instances based on period set 32
6 Performance of One Pass Heuristics and Baseline methods 33
7 Performance of Multiple Pass Heuristics and Baseline methods 34
8 CD Content 41
9 List of abbreviations 43

List of Algorithms

1 Backtracking algorithm for CSP by Russell and Norvig [9] 7
2 One Pass Heuristic 15
3 FFS – First Fit Stream 15
4 FFSI – First Fit Stream Instance 16
5 Heuristic based on Conflict-Directed Backjumping with Backmarking by [12] 22
6 Benchmark generator 29
7 Communication generator 30
8 Placing stream in the schedule 31

1 Introduction

Scheduling is a common optimization problem with many applications. Following definitions in [1, 2], it is a decision-making process used in different areas such as production, transportation, manufacturing and information technology. The goal of a scheduling process is to assign time intervals on resources for given tasks while following the predefined constraints.

Resources represent all the available objects for which the schedule is made. Resources may be of different types like machines, space, communication links, vehicles, school rooms, etc.

Tasks are jobs to be performed in accordance with their specification. Each task may have several dependencies (set of tasks that need to be executed in a predefined order), earliest start time, deadline, duration, and periodicity. Example of such task can be an airplane line, which goes from London to Prague and then from Prague to Budapest every week, and due to other usages, the aircraft can be used only on Monday and Tuesday.

Constraints are necessary conditions which ensure the schedule is plausible and fault-free. Correctly defined constraints are essential for the quality of the solution, and the definitions can be very complex. Example of such real-life constraint can be a parking slot where two cars cannot be parked on the same spot at the same time.

Currently, scheduling is widely used for communication in *cyber-physical systems*. The cyber-physical system consists of physical devices and links connecting the devices. Links between the devices allow them to exchange information. Such an exchange of information between the physical devices is called *stream*. However, the physical links also have their limitations and the more communication is distributed between the devices, the more sophisticated algorithm is necessary to control it.

Moreover, in highly critical systems such as automotive, avionics, etc., additional scheduling requirements such as determinism and guaranteed output are demanded [3]. Any disturbance of these aspects may have severe consequences and cause undesired effects on safety (e.g., if the warning of an incoming person in a self-driving car is not delivered on time, the caused accident can result in people dying).

Typical communication in the network is of a control character – the device is sending information about its current state which is being transmitted periodically in a regular time cycle. Apart from that, there are following specifics resulting from the nature of the highly critical industrial communication:

- **Time-triggered** – any operation in the system is determined by the globally synchronized clock and depends on the predefined schedule
- **No preemption** – critical streams cannot be interrupted
- **Low end-to-end latency** – response time must fit into $[release, deadline]$ time window representing a fraction of the stream period
- **Zero jitter** – the variance of the response time must be zero

- **Time synchronized** – the system is time synchronized with high precision

Apart from the considered scheduler determinism, other used resources like wires, vehicles or machines must also act deterministically, e.g., be resistant to external influences, etc. This aspect is not further discussed in this thesis but is considered a necessary condition [3].

As described in [4] the zero jitter periodic scheduling with at least two different periods is a strongly NP-complete problem. While exact methods like Integer Linear Programming provide a proved optimal solution, the computational time of such solvers prevents it from being used for large scale systems.

On the other hand, heuristic methods can provide a feasible sub-optimal solution within a significantly smaller time frame. The purpose of this thesis is to introduce several heuristic algorithms that address the described form of the periodic scheduling problem and to compare their performance on the generated experimental setups. In addition, a graphical user interface is implemented to easily show schedule, topology, and setup of the solved problem.

The thesis is logically divided into several chapters as follows – Chapter 2 overviews the related literature and state-of-the-art approaches and provides an introduction to heuristic methods, exact methods, and possible industrial application. Chapter 3 formally describes the solved problem. Chapter 4 is the core of this thesis and consists of algorithms designed for the given problem. Chapter 5 briefly describes the implementation (a full description of the code – ReadMe, JavaDoc, API definition and the code itself is provided on the enclosed CD). Chapter 6 describes instance setup, evaluation of the tested algorithms and discussion of the results. Chapter 7 summarizes the thesis contribution.

2 Background

In this section, we describe the concept of Profinet IO IRT [5] standard that is widely used in industrial communication network and represents one of the possible applications of the algorithms described further in this thesis. We also describe the concepts of heuristics and exact methods. In the end, we review heuristic methods published in related literature.

2.1 Ethernet Scheduling

As described in [6] and [3], original Ethernet communication was intended to be event triggered. Event triggered communication tries to pass streams in the system immediately as created. However, since there is no time slot prebooked for the transmission, it can easily happen that the link is already occupied and the task must wait until it can be transmitted.

On the contrary, the usage in hard real-time systems requires deterministic and predictable behavior. Ethernet was not designed to satisfy these requirements. The ability to guarantee the event order is not typically implemented in networking protocols such as TCP/IP. Therefore different standards have been introduced to overcome the hardware shortcomings – here we briefly describe the Profinet IO IRT standard.

Profinet IO IRT is a hard real-time communication protocol using individual static schedules that are distributed among the network nodes. Special hardware (switch) capable of processing these schedules is required. Four different communication classes are defined to serve different requirements: RT Class UDP, RT Class 1, RT Class 2, and RT Class 3. They differ in clock synchronization and real-time capability. The communication cycle is divided between the classes and creates individual communication intervals. Class UDP and Class 1 serve for event-triggered communication and are not suitable for critical traffic. Class 2 is deprecated and not currently used. Class 3 has the highest priority and allows to implement the time-triggered concept which is necessary to satisfy the real-time requirements. With properly synchronized clocks the jitter of the communication cycle is as low as possible.

The algorithms presented in this thesis are implemented for general communication scheduling problem, if applied on Profinet IO IRT standard, it would be necessary to introduce additional "safety margin" parameter separating each transmission frame. [6]

2.2 Heuristic Algorithms

Heuristic algorithms are especially useful for problems with a large solution space. In such problems, searching the whole solution space becomes impossible due to time requirements. Based on [7], the permutation Flow-shop scheduling problem has $n!$ possible flow order setups where n is the number of flows (tasks). Considering it takes ≈ 1 ns to schedule each task order setup, for only 20 tasks it would take around 77 years to compute the solution if exploring every option. Even though efficient space searching can significantly reduce the number of explored task setups, the exact algorithms still spend a lot of time searching for the optimal solution. Therefore, it is often necessary to compromise on optimality to obtain a fast solution.

Based on [8], there are three main performance measures for an algorithm:

- Completeness – whenever a solution exists, the algorithm finds it
- Optimality – the algorithm always returns solution of the best objective value
- Complexity – measures the time and memory requirements of the algorithm

In this thesis, we will focus on the time complexity while keeping the optimality and completeness as close as possible to the solution of exact methods.

According to [1], scheduling heuristic algorithms can be divided into several categories. They can also use different approaches for creating the schedule and different priority rules. The priority rules are used for creating the order in which the tasks are added to the schedule. In this thesis, we follow these paradigms to formally describe the implemented algorithms.

2.2.1 Constructive Heuristics

Constructive heuristics create a schedule from scratch by sequentially adding tasks. Since tasks have different scheduling difficulty (number of repetitions in the hyper period, duration, etc.), priority rules are created to determine the order in which the tasks are added to the schedule. A useful priority rule must maintain precedences – if task a depends on the execution of task b, task b should be placed before the task a in the priority list.

Many different **priority rules** are described in [1]. Here we will present a selection of these rules with corresponding criteria suitable for the scope of this thesis:

- MTS: Most total successors

  The number of successors of the given task.
- EST: Earliest start time

  The earliest possible start time of the task based on its predecessors.
- LST: Latest start time

  The latest possible start time of the task based on its successors.
- $MSLK$: Minimum slack

  The slack in which the task can start = $LST - EST$.
- RED: Resource equivalent duration

  The product of duration and weighted resource requirements.

Two main **schemes** for constructing the schedule are:

- Serial scheduling scheme

  The serial scheduling scheme sequentially adds tasks from the sorted priority lists while keeping the constraints satisfied. It strictly follows the order given by the priority list and assigns each selected task the lowest possible start time. Simply said – it assigns start times to tasks. Further, in the text, we refer to this approach as First Fit.

- Parallel scheduling scheme

  The parallel scheduling scheme works with the sorted priority list as well. In contradistinction to the serial scheme, it iterates over free start times. It selects the lowest possible start time and then searches through the priority list to find the first possible task to be assigned to this start time while keeping the constraints satisfied. Simply said – it assigns tasks to start times.

2.2.2 Improvement Heuristics

Improvement heuristics enhance already created schedule obtained by a constructive heuristics. Different operations depending on the problem specification and optimization criteria are performed to find a local optimum. As in any optimization problem, it is necessary to prevent getting stuck in a loop. Some of the used techniques are genetic algorithms, simulated annealing, and tabu search.

2.3 Exact Algorithms

Exact algorithms are complete and optimal. We will introduce two approaches suitable for solving the scheduling problem – Constraint Programming (CP) and Integer Linear Programming (ILP). Since we use the CP mainly for enhancing our heuristics, we will use its version that is faster but does not guarantee optimality. On the other hand, the introduced ILP model guarantees optimality if provided sufficient time. Hence, we will further use the ILP method to compare the objective value of our other algorithms.

2.3.1 Constraint Programming

Based on [2], scheduling is a Constraint Satisfaction Problem (CSP) which is a problem requiring a search for a feasible solution that satisfies all the predefined constraints. It is defined as a triple of:

- $X = \{x_1, ..., x_n\}$ – a set of decision variables
- $D = \{D_1, ..., D_n\}$ – a set of allowable values for each variable
- $C = \{C_1, ..., C_m\}$ – a set of constraints

Each variable from $x_i \in X$ can be assigned only a value a_i belonging to its domain $D_i \in D$. The value assignment $(x_i = a_i, x_j = a_j, ...)$ is subject to constraints defined in C. For each constraint, we can define a consistency checking function f such that $f_i(x_1, ..., x_n) = 1$ if and only if the constraint C_i is satisfied. An assignment that satisfies all $C_i \in C$ is called *feasible*. An assignment where all variables are instantiated is called *complete*. Otherwise, we call the assignment *partial*. A *solution* of CSP is a feasible and complete assignment.

In the scheduling problem, the variables represent the tasks that are to be scheduled, and the domains represent their possible start times. Hence, the domains are finite sets of discrete values. The constraints are binary and work over each pair of tasks.

The CSP is typically solved via a tree search algorithm where each node represents a partial assignment of variables and reduced domains $D' = \{D'_1, ..., D'_n\}$ where $D'_i \subseteq D_i$. Each time a variable is assigned, the node creates a new branch and performs a consistency check. In case the consistency check fails, the node is not further explored. The algorithm stops when it finds the first complete and feasible assignment or if all nodes were explored without finding such an assignment.

It is important to note that the naïve algorithm has a big branching factor – if all domains D_i had the same size then the number of all different complete assignments would be $|D_i|^n$ where n is the number of variables.

The CSP tree search algorithm is complete because it finds the solution each time it exists (assuming we have enough time and resources for the computations). However, it is not optimal since it returns the first found solution. If needed, the algorithm can be improved in a way that allows adding an objective function [2], but we do not use this technique and rather use the techniques allowing us to find the first complete solution as quickly as possible.

There are many different CSP techniques helping to speed up the naïve tree search. Further, we will describe – Backtracking, Backjumping, and Backmarking.

Backtracking is used for a depth-first search that assigns values to variables one by one and when a conflict occurs (the domain of some unassigned variable is empty), the algorithm backtracks to the most recently assigned variable. The pseudocode is shown in Algorithm 1. Line 9 of the algorithm performs a forward checking. Forward checking is a procedure in which we reduce domains of variables that have not yet been assigned based on the currently assigned variable's value. In case this step results in some of the variables having an empty domain, a feasible solution does not exist for the current partial assignment, and the algorithm needs to backtrack.

Conflict-Directed Backjumping (CBJ) is a more efficient version of backtracking. While in backtracking, the algorithm returns to the previous level of the tree, the backjumping method can jump right to the assignment that caused the current failure. More specifically, we create a conflict set $conf(x_i)$ for each variable $x_i \in X$. Each time we try to assign x_i some value from its domain and the assignment fails due to consistency checks, we add the variable x_j that caused conflict to the conflict set of x_i. After we unsuccessfully try to assign x_i every value from its domain, we backjump to the most recently assigned variable x_h from

Algorithm 1 Backtracking algorithm for CSP by Russell and Norvig [9]

```
1: procedure BACKTRACKING-SEARCH(csp)
2:     return BACKTRACK({}, csp)
3: procedure BACKTRACK(assignment, csp)
4:     if assignment is complete then return assignment
5:     var ← select-unassigned-variable(csp)
6:     for value ∈ order-domain-values(var, assignment, csp) do
7:         if value is consistent with assignment then
8:             add{var = value} to assignment
9:             inferences ← inference(csp, var, value)
10:            if inferences ≠ failure then
11:                add inferences to assignment
12:                result ← BACKTRACK(assignment, csp)
13:                if result ≠ failure then return result
14:        remove {var = value} and inferences from assignment
15:    return failure
```

the conflict set of x_i and update the conflict set of x_h:

$$conf(x_h) \leftarrow conf(x_h) \cup conf(x_i) \setminus \{x_h\}$$

This update helps us to keep information about the conflicting variables. [9]

Backmarking is a method introduced by Gaschnig [10]. In this method the results of consistency checks are saved and reused after the algorithm backtracks; this helps to avoid unnecessary constraints rechecking.

As introduced in Kondrak and van Beek [11], both of the methods can be combined. Moreover, in [12], Vlk shows an iterative version of the combined *Conflict-Directed Backjumping with Backmarking* (*CBJ_BM*). Later on, in Section 4.4, we will elaborate on this exact algorithm while creating a more sophisticated heuristics. In the mentioned chapter, we also describe the *CBJ_BM* more thoroughly and provide pseudocode.

2.3.2 Integer Linear Programming

Integer Linear Programming (ILP) is a special case of Constraint Satisfaction Problem where all variables are discrete and constraints linear. The ILP is given by matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and vectors $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{c} \in \mathbb{R}^n$. The goal is to find a vector $\mathbf{x} \in \mathbb{Z}^n$ such that $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{c}^T\mathbf{x}$ is maximized. [1]

Even though the exact methods are not the core of this thesis, an ILP model was implemented as a reference solution for the heuristics and is further described in Section 4.2.

2.4 Related Work

In [13], Pira proposes an improvement heuristic following the game theory paradigm. As in game theory problem where two players take turns, changing their strategies based on the current knowledge, the algorithm updates schedule partitions to optimize their quality.

Minaeva et al. [14] address the problem by creating a constructive heuristic that sequentially adds tasks to the schedule based on priority order. In the case of the infeasibility of the given task, a reason graph is used to help with the backtracking. After this step, the schedule is optimized by local neighborhood search using ILP.

Syed and Fohler [15] present a search-tree pruning heuristic based on job response-time. The heuristic searches for groups of symmetrical sub-schedules and uses only one sub-schedule from the group in the searching process (tree pruning technique). They emphasize the fact that pruning of infeasible search-tree paths is as important as looking for new feasible ones and use Parallel Iterative Deepening A* to create the schedule.

Finally, Bansal creates a divide-and-conquer heuristic in his master thesis [16]. Contrary to most of the algorithms in the literature, he proposes a decentralized approach aiming mostly at large-scale networks. Following the divide-and-conquer paradigm, the schedule for each link is created separately, and then the schedules are completed together. However, from the text, it is unclear how the algorithm treats conflicts among the pre-created schedules.

There are several different approaches to consider when designing a heuristic algorithm. From the literature described above, it is clear that any introduction of backtracking significantly increases the time for which the solver runs. Therefore, we decided to focus on one pass heuristics that don't use any backtracking and their power lies in creating the order in which the streams are added to the schedule. This concept was partially introduced in [14] but was not deeply explored. Further, we apply the knowledge gained from the one pass heuristics and design Conflict-Directed Backjumping and Backmarking search method proposed in [12] which is using dynamic granularity to speed up the searching.

3 Problem Statement

The problem statement is motivated by communication in industrial networks and is formulated as follows: Scheduling strictly periodic transmission over network links where each stream has a predefined path and links have different weights corresponding to transmission speeds. To adapt to the real-life situation we use time lags on nodes and links modeling the transmission and fabric switching delays in the industrial systems [6]. Note that the domains of the parameters, such as transmission duration, are integral which corresponds to real scheduling of production where the model is discretized, usually to a unit corresponding to a common network clock granularity.

3.1 Network Model

The network is modeled as a simple connected directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes (network devices) and $\mathcal{E}$ is a set of links representing connections between the nodes.

Each node $v_i \in \mathcal{V}$ is specified by its time lag $v_i.l \in \mathbb{N}_0$. Each link $e_k = (v_a, v_b) \in \mathcal{E}$ is specified by its time lag $e_k.l \in \mathbb{N}_0$, weight $e_k.w \in \mathbb{N}$, origin $e_k.from \in \mathcal{V}$, and target $e_k.to \in \mathcal{V}$. The link weight represents a link speed coefficient. The network is full duplex, meaning that $(v_a, v_b) \in \mathcal{E} \iff (v_b, v_a) \in \mathcal{E}$.

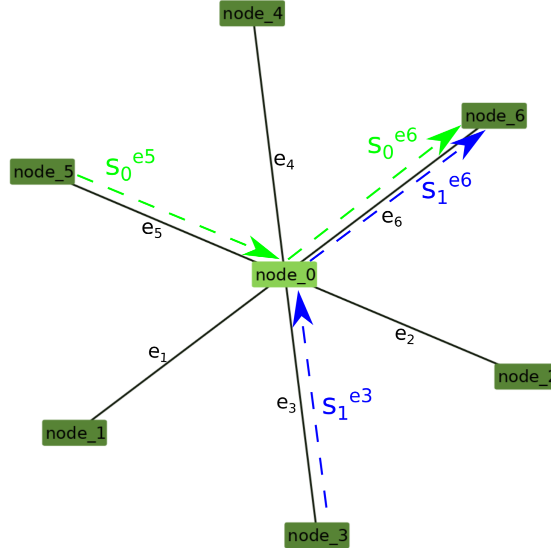


Figure 1: Visualization of the network and communication model

3.2 Communication Model

A stream is a periodic transmission from the origin node to the target node throughout the network. A set of all streams is denoted by S. Each stream $s_j \in S$ is specified by its duration $s_j.p \in \mathbb{N}$, release date $s_j.r \in \mathbb{N}_0$, deadline $s_j.\tilde{d} \in \mathbb{N}$, period $s_j.T \in \mathbb{N}$, origin $s_j.org \in \mathcal{V}$,

and target $s_j.trg \in \mathcal{V}$. Streams may have different periods resulting in common hyper period $HP \in \mathbb{N}$ which is the least common multiple (LCM) of all periods.

We denote the route of the stream s_j as R_j where $R_j = (e_{j^1}, ..., e_{j^n})$ is a sequence of links that are visited on the way from $s_j.org$ to $s_j.trg$. The last edge from the sequence R_j is denoted $R_{j,last}$. The stream instance $s_j^{e_k}$ represents the stream s_j routed through the link e_k.

Figure 1 depicts sample network communication. There are seven nodes and six links in the network. Two streams s_0 and s_1 are sent from $node_5$ to $node_6$ and from $node_3$ to $node_6$ respectively. This results in having 4 stream instances $\{s_0^{e_5}, s_0^{e_6}, s_1^{e_3}, s_1^{e_6}\}$ in the network. Routes of the streams are equal to $R_0 = (e_5, e_6)$ and $R_1 = (e_3, e_6)$.

The scope of the schedule is a hyper period. This results in $HP/s_j.T$ periodic repetitions of the stream instance. After completing the first hyper period, the schedule repeats itself, and it would be redundant to enlarge the scheduler time scope.

Since we are dealing with zero jitter scheduling, it is not theoretically necessary to use other granularity of the stream because each periodic repetition of the stream instance is determined by its first occurrence. However, for the clearness of the notation, we will denote the reoccurred stream instance as $s_{j,l}^{e_k}$ where $l \in \{0, ..., \frac{HP}{s_j.T} - 1\}$.

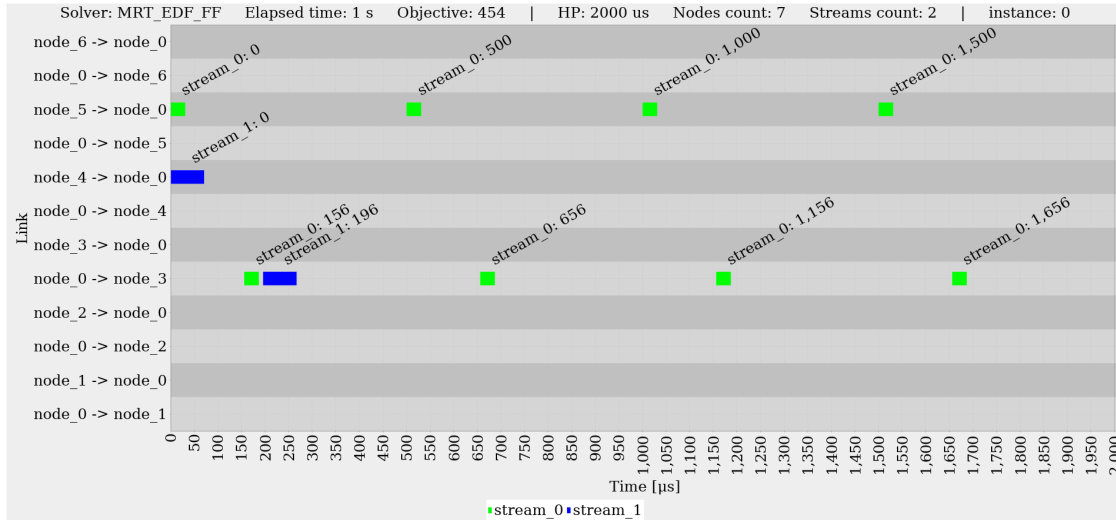


Figure 2: Gantt chart of the sample instance

Figure 2 shows a sample schedule of the instance proposed above. In this example, $s_0.T = 500\ \mu\ s$ and $s_1.T = 2000\ \mu s$ resulting in a hyper period equal to 2000 μs. In the vertical axis, each line of the Gantt chart represents the schedule of one network link. The horizontal axis represents discrete time in microseconds. In the schedule, we can see reoccurred stream instances depicted as single colored rectangles. In total, there are eight reoccurred stream instances for s_0 (it repeats four times in the hyper period and transmits over two links) and 2 reoccurred stream instances for s_1 (it occurs only once in the hyper period and transmits over two links).

3.3 Scheduling Problem

The goal is to find a periodic schedule for each link in the network. Our approach is to assign a valid value to each reoccurred stream instance marking its start time $s_{j,l}^{e_k}.\phi \in \mathbb{N}_0$. The start time assignment is subject to several constraints described below.

1. **Zero Jitter Constraint**

   All reoccurred stream instances of each stream are strictly periodic, meaning there is zero jitter between period instances.

$$\forall s_j \in S, \forall l \in \{1, ..., HP/s_j.T - 1\}, \forall e_k \in R_j : s_{j,l}^{e_k}.\phi = s_{j,l-1}^{e_k}.\phi + s_j.T \tag{1}$$

2. **Link Constraint**

   No link can be occupied by more than one stream at the moment.

$$\forall e_k \in \mathcal{E}, \forall s_{j,l}^{e_k}, s_{h,i}^{e_k}, (j,l) \neq (h,i) : \\ s_{j,l}^{e_k}.\phi + s_j.p \cdot e_k.w \leq s_{h,i}^{e_k}.\phi \ \vee s_{h,i}^{e_k}.\phi + s_h.p \cdot e_k.w \leq s_{j,l}^{e_k}.\phi \tag{2}$$

3. **Precedence Constraint**

   The stream can be processed only after it is fully prepared on the current node.

$$\forall s_j \in S, \forall h \in \{1, ..., len(R_j) - 1\} : \\ s_{j,0}^{e_j^{h-1}}.\phi + s_j.p \cdot e_j^{h-1}.w + e_j^{h-1}.l + (e_j^{h-1}.to).l \leq s_{j,0}^{e_j^h}.\phi \tag{3}$$

4. **Release & Deadline Constraint**

   The transmission interval for each reoccurred stream instance must fit into the $[release, deadline]$ interval for the given period. Taking into consideration the precedence constraint (3), it is enough to check whether the first stream instance of the stream s_j starts after the release time and the last stream instance of s_j finishes before the deadline. Moreover, due to the zero jitter constraint (1), it is enough to check it only in the first period of s_j.

$$\forall s_j \in S : s_{j,0}^{R_{j,0}}.\phi \geq s_j.r \tag{4}$$

$$\forall s_j \in S : s_{j,0}^{R_{j,last}}.\phi + s_j.p \cdot R_{j,last}.w + R_{j,last}.l \leq s_j.\tilde{d} \tag{5}$$

3.4 Scheduling Objective

We are minimizing the sum of the end-to-end latencies of all streams.

$$\min \sum_{s_j \in S} s_{j,0}^{R_{j,last}}.\phi + s_j.p \cdot R_{j,last}.w + R_{j,last}.l - s_{j,0}^{R_{j,0}}.\phi \tag{6}$$

4 Proposed Solution

In this chapter, we describe the proposed algorithms and provide pseudocode for better understandability. Two baseline algorithms were implemented to measure the quality of the algorithms – ILP (exact method) and Random Heuristic.

The algorithms are evaluated by two different quality measures – *schedulability* and *objective value* reached in the given time limit. Schedulability describes whether a consistent solution was found, meaning that all reoccurred stream instances have start time assigned and all constraints are satisfied. The usage of the time limit is especially important for the exact method to ensure that it finishes within a reasonable time. The calculation of the objective value is shown in the equation (6). The goal of the proposed algorithms is to be faster than the exact ILP method and to yield better results (with respect to schedulability or objective value) than the random method.

It is important to understand that ILP is a complete method – in case it proves that the instance is not schedulable, no other method can find a solution. Similarly, in case the instance is schedulable, and ILP proves optimality, no other solution would have lower objective value. However, in the case a time limit is set, and ILP is preempted, the currently best solution is yielded. Since it did not search the whole solution space, it gives us no guarantees about the optimality of the solution.

Paths for all streams were precomputed by breadth-first search algorithm to correspond to the first found paths with the lowest number of links.

4.1 Schedulability Conditions

A necessary condition for the schedulability of the problem is that utilization of no link in the network exceeds 100 %. Let us denote S^{e_k} as a set of all streams routed through the link e_k. Then the following condition must apply:

$$\forall e_k \in \mathcal{E} : \text{util}(e_k) \leq 1$$

where

$$\text{util}(e_k) = \sum_{s_i \in S^{e_k}} \frac{(e_k.w \cdot s_i.p)}{s_i.T} \tag{7}$$

4.2 Integer Linear Programming

As described in Section 2.3.2, Integer Linear Programming solves optimization problem on discrete variables and linear constraints. To apply it to our problem, all the constraints must be linearized to be in a form $\mathbf{a}^T\mathbf{x} \leq b$ where b is a constant, $\mathbf{a}$ is a vector of constant values and $\mathbf{x}$ is a vector of variables.

Constraints (1), (3), (4) and (5) are already in a linear form. The only constraint that needs to be linearized is *Link Constraint* (2) which is in a disjunctive form. It can be modeled using a big-M (a positive large enough constant) and a binary variable $y \in \{0, 1\}$ so that it can "switch off" one of the inequalities.

Let us substitute

$$\begin{aligned} x_1 &\leftarrow s_{j,l}^{e_k}.\phi \\ x_2 &\leftarrow s_{h,i}^{e_k}.\phi \\ a &\leftarrow s_j.p \cdot e_k.w \\ b &\leftarrow s_h.p \cdot e_k.w \end{aligned}$$

The variables x_1, x_2 represent start times for reoccurred stream instances. Then *Link Constraint* (2) for one valid triplet of $\{e_k, s_{j,l}^{e_k}, s_{h,i}^{e_k}\}$ is represented as $x_1 + a \leq x_2 \vee x_2 + b \leq x_1$. Using a big-$M$ notation:

$$\begin{aligned} x_1 + a &\leq x_2 + M \cdot y \\ x_2 + b &\leq x_1 + M \cdot (1 - y) \end{aligned}$$

and the derived *Link constraint* is:

$$\begin{aligned} &\forall e_k \in \mathcal{E}, \forall s_{j,l}^{e_k}, s_{h,i}^{e_k}, (j,l) \neq (h,i) : \\ &z_{j,l,h,i}^{e_k} \in \{0, 1\} \\ &s_{j,l}^{e_k}.\phi + s_j.p \cdot e_k.w \leq s_{h,i}^{e_k}.\phi + M \cdot z_{j,l,h,i}^{e_k} \\ &s_{h,i}^{e_k}.\phi + s_h.p \cdot e_k.w \leq s_{j,l}^{e_k}.\phi + M \cdot (1 - z_{j,l,h,i}^{e_k}) \end{aligned}$$

Since the scope of the schedule is a hyper period (HP), no start time can be larger than the hyper period and we can use it as big-M ($M = HP$).

4.3 One Pass Heuristics

One Pass Heuristics are methods that perform one iteration to create a schedule based on a priority sequence of streams (Algorithm 2). In such heuristics, streams are sequentially added to the schedule, and in case any of the streams cannot be placed to the schedule, the heuristics discard the instance and yield *"no solution."* A well-sorted sequence of the streams is crucial for the method success rate. Since the minimized objective is the end-to-end latency, Equation (6), we are attempting to place the streams as early to the schedule as possible which corresponds to the serial scheduling scheme described in Section 2.2.1.

4.3.1 First Fit Methods

There are two main approaches to adopt for the first fit method: *scheduling streams* and *scheduling stream instances*. The first approach is called First Fit Streams (further referred

Algorithm 2 One Pass Heuristic

```
1: procedure SCHEDULE(problemInstance)
2:     PQ ← priorityQueueInit(problemInstance)
3:     schedule ← createScheduleFirstFit(PQ)
4:     return schedule
```

to as FFS) and is described in Algorithm 3. It creates a priority queue containing the streams (the prioritizing rules will be discussed in the text below) and then polls (removes from the top of the queue) streams from the priority queue one after another and places all stream instances of the currently processed stream to the first available time slot in the schedule.

The second approach is called First Fit Stream Instances (further referred to as FFSI) and is described in Algorithm 4. It also starts by creating the priority queue but this time the queue consists of stream instances, not the whole streams. This means that stream instances of the same stream are not necessarily placed right next to each other in the queue. However, as mentioned in Chapter 2, it is necessary to keep precedences in between the stream instances. If the stream instance a is dependent on stream instance b, the stream instance b must be placed above the stream instance b in the priority queue. On the other hand, scheduling single stream instances provides more variability, allowing us to simply (with not much computational cost) update the priority queue. FFSI proceeds similarly as FFS – it takes stream instances from the priority queue and places them to the first available slot in the schedule. Additionally, each time a stream instance is placed to the schedule, the priority queue is updated (in the case the currently placed stream instance affected the criteria value of other stream instances in the queue).

Both of the approaches use list `freeSlots` while searching for an empty space for the current stream instance. Each link has one instance of this list that corresponds to the sequence of free intervals, e.g. $\{0-10, 40-150, 180-200\}$. Considering this sample list, it would mean that we can schedule stream instance of duration at most 10 to the first slot, stream instance of duration at most 110 to the second slot, etc. In the worst case, the length of the list is $HP/2$ which corresponds to unit-length slots separated by unit-length space.

Algorithm 3 FFS – First Fit Stream

```
1: procedure CREATESCHEDULEFIRSTFITSTREAM(PQ)
2:     schedule ← initialize empty schedule for each link
3:     freeSlots ← initialize all links available throughout whole hyper period
4:     while !PQ.empty() do
5:         stream ← PQ.pop()
6:         for streamInstance ∈ stream do
7:             start = findFirstSlotForAllPeriods(streamInstance)
8:             if start == -1 then return null
9:             addToSchedule(streamInstance, start, stream.period)
10:            removeFromFreeSlots(streamInstance, start, stream.period)
11:    return schedule
```

Algorithm 4 FFSI – First Fit Stream Instance

```
1: procedure createScheduleFirstFitStreamInstance(PQ)
2:     schedule ← initialize empty schedule for each link
3:     freeSlots ← initialize all links available throughout whole hyper period
4:     while !PQ.empty() do
5:         streamInstance ← PQ.pop()
6:         start = findFirstSlotForAllPeriods(streamInstance)
7:         if start == -1 then return null
8:         addToSchedule(streamInstance, start, stream.period)
9:         removeFromFreeSlots(streamInstance, start, stream.period)
10:        updatePQ(streamInstance, start, PQ)
11:    return schedule
```

4.3.2 Priority Queue Ordering

As we mentioned above, the way in which the priority queue is sorted is very important – simply for the reason that one stream instance that cannot fit into the schedule causes the whole heuristic to fail. The priority queue is sorted by the lowest value of the first criterion and then in case of a draw by the lowest value of the second criterion. Since we quickly noticed that the deadline of the stream strongly affects the scheduling process, we decided that it will always play a role in one of the criteria.

More specifically, we designed two slightly different criteria that are using the deadline. The EDF (earliest deadline first) criterion corresponds to the value of $s_j.\tilde{d}$, and DF (deadline first) corresponds to the value of the deadline with lowered granularity $\lceil s_j.\tilde{d}/100 \rceil$. Lowering the granularity of the deadline enables aggregating values that are close by to the same criterion value, and then the heuristic rule can more likely decide by the second criterion.

As a complement to these two deadline criteria, six other criteria were implemented. These criteria are formally described in Tables 1 and 2. The criteria values are calculated based on different parameters like stream period, duration, utilization of the resources, number of precedences or earliest/latest start time. Method $est(s_j^{e_k})$ is used to calculate the earliest start time of stream instance $s_j^{e_k}$ with respect to its predecessors and release time. Method $lst(s_j^{e_k})$ is used to calculate the latest start time of $s_j^{e_k}$ with respect to its successors and deadline. Both of the methods are defined recursively using the stream path definition $R_j = (R_{j,0}, ..., R_{j,i}, ...R_{j,last})$.

$$est(s_j^{R_{j,i}}) = \begin{cases} est(s_j^{R_{j,i-1}}) + s_j.p \cdot R_{j,i-1}.w + R_{j,i-1}.l + R_{j,i-1}.to.l & \text{if } i \geq 1 \\ s_j.r & \text{if } i = 0 \end{cases}$$

$$lst(s_j^{R_{j,i}}) = \begin{cases} lst(s_j^{R_{j,i+1}}) - R_{j,i}.w \cdot s_j.p - R_{j,i}.l - R_{j,i}.to.l & \text{if } i \neq last \\ s_j.\tilde{d} - R_{j,last}.w \cdot s_j.p - R_{j,last}.l & \text{if } i = last \end{cases}$$

In total, we have two criteria suitable for FFS and four criteria suitable for FFSI. The following criteria use streams for calculating the criterion value. The criterion Most Required Time (MRT) calculates the minimal end-to-end latency (the minimal time it takes to transfer

the stream from the origin to the target node throughout the network). The minimal end-to-end latency is subtracted from the hyper period to ensure that the stream with the largest end-to-end latency has the highest priority. The criterion Resource Equivalent Duration (RED) is similar to MRT with the difference that the stream transmission duration on each link is multiplied by a utilization coefficient determined by the given link.

The following criteria use stream instances for calculating the criterion value. The criterion Most Total Successors calculates the number of stream instances of the same stream that need to be scheduled after the current stream instance. The value is subtracted from the total number of links to ensure that the stream instance with the most successors has the highest priority. Since this value does not change throughout the scheduling process, the priority queue is not being updated. The formal definition uses stream instance id which is calculated based on stream path definition as $id(s_j^{R_{j,i}}) = i$. The criterion Earliest Start Time (EST) calculates the earliest possible start time based on the release time of the stream and duration of the preceding stream instances. This value is updated each time a preceding stream instance is scheduled. The criterion Latest Start Time (LST) calculates the latest possible start time based on the deadline of the stream, the stream instance duration and the duration of the succeeding stream instances. The criterion Minimum Slack (MSLK) calculates the length of the interval during which the stream instance can start based on EST and LST.

As already mentioned above, the heuristic rule is created by two criteria and has one of the three following structures:

- 1. EDF, 2. Complement Criterion
- 1. Complement Criterion, 2. EDF
- 1. DF, 2. Complement Criterion

When creating all the possible rules corresponding to these structures (3 possible structures, 6 possible complement criteria) we end up with $6 \cdot 3 = 18$ one pass heuristics in total. We do not consider the rule structure *1. Complement Criterion 2. DF* since it would yield very similar results as *1. Complement Criterion, 2. EDF* rule structure.

Shortcut	**Rule name** **Criterion calculation**
MRT	Most Required Time $HP - HP/s_j.T \cdot \sum_{e_k \in R_j} (e_k.w \cdot s_j.p + e_k.l + e_k.to.l) + R_{j,last}.to.l$
RED	Resource Equivalent Duration $10 \cdot HP - \sum_{e_k \in R_j} \lceil 10 \cdot util(e_k) \rceil \cdot (e_k.w \cdot s_j.p + e_k.l)$

Table 1: List of complement criteria for s_j and FFS scheduling

4.3.3 Complexity Analysis

For analyzing the complexity, we use common knowledge that the complexity of adding an element to priority queue is $\mathcal{O}(\log n)$ and retrieving an element from the queue is $\mathcal{O}(1)$.

Shortcut	Rule name	Criterion calculation
MTS	Most Total Successors	$\lvert\mathcal{E}\rvert - (\lvert R_j\rvert - id(s_j^{e_k}))$
EST	Earliest Start Time	$est(s_j^{e_k})$
LST	Latest Start Time	$lst(s_j^{e_k})$
MSLK	Minimum Slack	$lst(s_j^{e_k})$ - $est(s_j^{e_k})$

Table 2: List of complement criteria for $s_j^{e_k}$ and FFSI scheduling

Further, we use graph diameter d which corresponds to the maximum eccentricity in the graph, i.e., the length of the longest shortest path between any two nodes in the network graph. Other complexity parameters are the number of streams in the network $|S|$, hyper period HP and the maximal period of any stream in the network T. Since we will often work with expression $|S| \cdot d$ corresponding to the maximal total number of stream instances in the network, we will substitute this expression as $|SI|$.

The complexity analysis of FFSI follows the pseudocode described in Algorithm 4. The steps of the algorithm that require non-linear time are: priority queue initialization and iterating over all stream instances while searching for a free slot and updating the priority queue. We will calculate the complexity of the algorithm from the complexity of these steps.

Initialization of the *schedule* and $freeSlots$ has linear complexity and is negligible compared to the priority queue initialization. In the iteration cycle, popping an element from the priority queue is $\mathcal{O}(1)$ and methods $addToSchedule(...)$ and $removeFromFreeSlots(...)$ work with time slots already found by method $findFirstSlotForAllPeriods(...)$ and their time complexity is constant.

The complexity is calculated as follows:

- **Priority queue init**: $|SI| \cdot \log |SI| + |SI| \approx \mathcal{O}(|SI| \cdot \log |SI|)$
  The member $|SI| \cdot \log |SI|$ corresponds to inserting all stream instances to the priority queue. The member $|SI|$ corresponds to calculating the criteria value which is linear with respect to the total number of stream instances in the network.
- **Iterate over all stream instances**: $|SI|$
  - **Find free slots**: $(T/2) \cdot (HP/2) \approx \mathcal{O}(T \cdot HP)$
    The method goes through all the free slots in the list corresponding to the first periodical occurrence of the stream instance – the largest possible start time is at most $T - 1$. The number of visited slots while searching for the correct slot is at most $T/2$. Then it goes through the rest of the time slots in the lists and checks if there are the required free time slots available for the other reoccured stream instances in the hyper period. The total number of the time slots checked can be at most $HP/2$.
  - **Update priority queue**: $2 \cdot (d-1) + (d-1) \cdot \log |SI| \approx \mathcal{O}(d \cdot \log |SI|)$
    The method removes all the predecessors or successors (based on criterion type) of the currently scheduled stream instance from the priority queue. Since there are at most d stream instances in each stream, the total number of such predecessors

or successors can be at most $d-1$. Then we update the criterion value of these removed stream instances which is again linear. Finally, we return them to the priority queue with complexity $\mathcal{O}((d-1) \cdot \log |SI|)$.

Then the complexity of FFSI:

$$\begin{aligned}
&\mathcal{O}(|SI| \cdot \log |SI| + |SI| \cdot (T \cdot HP + d \cdot \log |SI|)) \\
&\approx \mathcal{O}(|S| \cdot d \cdot (\log(d \cdot |S|) + T \cdot HP + d \cdot \log(d \cdot |S|))) \\
&\approx \mathcal{O}(|S| \cdot d \cdot (T \cdot HP + d \cdot \log(d \cdot |S|)))
\end{aligned}$$

For FFS algorithm, the queue has only S members. The number of iterations is then S, and finding of free slots runs d times because we need to assign a start time to each stream instance of the stream. This results in the time complexity of $\mathcal{O}(|S| \cdot (\log |S| + d \cdot T \cdot HP))$.

The complexity of the algorithms depends on the values of T, HP, d and $|S|$. Hence, the algorithms belong to pseudo-polynomial complexity class. However, we must mention that especially the step *find free slots* works with a very pessimistic upper bound on the size of the `freeSlots` list. In reality, the runtime of these methods is very low as described in Chapter 6.

4.3.4 Random Heuristic

The random heuristic is the second baseline method used for evaluating the performance of the other heuristics. The implementation is rather straightforward, it uses the First Fit Stream (FFS) method and priority queue where the criterion is equal to a random number.

4.4 Multiple Pass Heuristics

Multiple pass heuristics are methods that perform several iterations over the sequence of streams, usually by using some form of backtracking. We based our multiple pass heuristics on the *CBJ_BM* exact method described in Section 2.3.1. Moreover, we enhanced the method by several techniques introduced in the text above – *priority rules*, *est* and *lst* functions, and enlarged granularity of the time units. The multiple pass heuristic is shown in Algorithm 5. The contributions to the original code are marked by a comment in the Algorithm. In the following paragraphs, we will first describe the specifics of the *CBJ_BM* iterative method proposed by [12] and secondly show the applied heuristical enhancements in detail. Furthermore, we have implemented both the original version of the algorithm *CBJ_BM* and the multiple pass heuristics and compared their performance in Section 6.2.3.

The core of the *CBJ_BM* algorithm is the while cycle, which is iterating over the sequence of stream instances $sInsts$ until all of them are scheduled or the solution is not found. The algorithms workflow is similar to the CSP backtracking algorithm shown in Algorithm 1 – stream instances are sequentially being assigned start times from their domains, and in case there is a collision, the algorithm backtracks to the previously assigned stream instance. Additional features specific to *CBJ_BM* are applied. The original values of variables that are not initialized in the pseudocode are zero or empty set.

4.4.1 Backjumping

The field $conf[siID]$ consists of the stream instances that conflicted with $sInsts[siID]$. In case the algorithm finds a conflict, the set $conf[siID]$ is updated (line 21), and the next $startTime$ is selected for the current stream instance. In case it is necessary to backtrack (line 31), we select the highest-indexed stream instance j from the conflict set and backjump to it. This backjump allows us to skip the backtracking steps that would result in a redundant search; but at the same time no feasible solution is skipped. However, the reason why $sInsts[siID]$ could not be scheduled may be even some earlier scheduled stream instance than $sInsts[j]$. Therefore the conflict set $conf[j]$ is updated to contain all the plausible causes.

In case there is no stream instance in the conflict set of the currently scheduled stream instance, there may be two different scenarios. Firstly, the domain of the stream instance was empty, then the problem instance is infeasible. Secondly, we backtracked to $siID = 0$ and could not find a start time that would result into a feasible solution. Then, if we are working with $step = 1$, the scheduling problem is infeasible. Otherwise, if $step > 1$, the problem may be infeasible or we may skip some solution.

4.4.2 Backmarking

The field $Mark$ is a $|sInsts| \times HP$ array initialized to 0. Each time there is a conflict we update the array so that the $Mark[siID][startTime]$ corresponds to the lowest-indexed stream instance preventing the $startTime$ from being assigned to the $sInsts[siID]$.

The field $BackTo$ is $|sInsts| \times HP$ array initially set to 0. The field $BackTo[siID][startTime]$ corresponds to the lowest-indexed stream instance which was reassigned after the current $sInsts[siID]$ was assigned with $startTime$.

These two fields help to reduce the number of consistency checks. The condition on the line 12 is called *type A saving*, it checks for a particular start time and stream instance, whether there is a variable for which the consistency checks already failed and still has the same value. In such a case, it would not make sense to continue because we know there is a conflict. The for loop on the line 19 is called *type B saving*. In this case, the $BackTo[siID][startTime]$ field represents up to which index the consistency checks already passed and we do not need to recheck them.

4.4.3 Heuristic Enhancements

To reduce the domains of the variables we use the $est(s_j^{e_k})$ and $lst(s_j^{e_k})$ functions introduced in Section 4.3.2. This reduction allows us to speed up the searching because only plausible start times (with respect to constraints (3), (4) and (5)) are contained in the domains of the stream instances. The variable $nextStart[siID]$ represents the first plausible start time from the domain that can be assigned to stream instance with $siID$ and is equal to either $est(s_j^{e_k})$ or the latest successfully assigned $startTime$ with added $step$ for the given stream instance $sInsts[siID]$. The upper bound on stream instance variable domain is set by $lst(s_j^{e_k})$. Since

we are trying to minimize the end-to-end latency, we do not try to apply any domain ordering, because similarly as in the first fit methods introduced in the text above, we are trying to place the stream instance to the first available time slot.

When we calculate the $startTime$ (line 5), the algorithm either just backtracked to this place or we proceeded forward from the previously assigned variable. In case the algorithm backtracked, we already calculated the start time before and we will now use its updated value that is saved in $nextStart[siID]$. In case we proceeded forward, the $nextStart[siID]$ variable is initialized to zero and we need to calculate the earliest meaningful start time to avoid the redundant searches. The start time is set to $est(sInsts[siID])$, which may be further influenced by the already assigned preceeding stream instances of the same stream. We add such stream instances to the conflict set of $siID$.

At the beginning of the algorithm, we sort the stream instances based on one of the well performing one pass heuristics. The corresponding criteria would be *DF* and *MRT*. Since *MRT* is criterion suitable for scheduling streams and in the *CBJ_BM* algorithm we are scheduling stream instances, we added the third criterion equal to the id of the stream instance in the stream instance sequence for the given stream $id(s_j^{e_k})$. This ensures that the stream instance precedences are kept in the created priority queue. Such sorting positively impacts the scheduling process because it helps to reduce the search time (see Section 6.2.3).

To speed up the search (which is necessary especially for large problems), we use a larger time granularity introduced by variable *step*. This implies a domain reduction that can possibly skip some solution (the algorithm is not complete) but on the other hand, it enables us to search the solution space faster. We evaluated three different methods for calculating the *step* size based on:

1. stream instance $s_j^{e_k}$ duration: $\lceil e_k.w \cdot s_j.p/100 \rceil$ – we call this method *CBJ_BM_D*
2. stream instance $s_j^{e_k}$ period: $\lfloor s_j.T/500 \rfloor$ – we call this method *CBJ_BM_P*
3. scheduling progress: $1 + \lfloor 30 \cdot siID/|sInsts| \rfloor$ – we call this method *CBJ_BM_ID*

As a result of the applied enhancements, the consistency checking can be vastly reduced. The only constraint we need to check is the link overlapping, Constraint (2). Constraint (1) can be skipped because we are scheduling the whole stream instance at a time, which is enforcing the zero jitter by deriving the start times of the respective reoccurred stream instances from the first periodical occurrence. Constraint (3) can be skipped because of the fact that we are always scheduling stream instances of the same stream in order that is keeping the precedences and we are using the $est(s_j^{e_k})$ function for domain reduction. Constraint (4) does not need to be checked because it is already integrated in the $est(s_j^{e_k})$ function used for domain reduction. Similarly, Constraint (5) is also part of the domain reduction in the $lst(s_j^{e_k})$ function.

Additionally, we implemented the algorithm with $step = 1$ (called *CBJ_BM*) and also the algorithm based on the original pseudocode in [12] (referred to as *CBJ_BM_NOH* where *NOH* stands for "no heuristic") to compare the performance of the multiple pass heuristics with their exact version. The *CBJ_BM_NOH* algorithm uses random sorting that keeps stream instance precedences, and the stream instances domains are represented by the interval $[s_j.r, s_j.\tilde{d}]$.

Algorithm 5 Heuristic based on Conflict-Directed Backjumping with Backmarking by [12]

```
1: procedure CBJ-BM-BASED-HEURISTICS(sInsts) returns the solution or false
2:     sInsts ← sort(sInsts, {DF, MRT, ID})                    ▷ One pass sorting
3:     siID ← 0
4:     while siID < |sInsts| do
5:         if nextStart[siID] == 0 then
6:             startTime, conflicts ← getEstAndConflicts(sInsts, siID)
7:             confSet[siID] ← confSet[siID] ∪ conflicts
8:         else
9:             startTime ← nextStart[siID]
10:        successful ← false
11:        while ¬successful ∧ startTime ≤ lst(sInsts[siID]) do    ▷ Reduce domains
12:            if Mark[siID][startTime] < BackTo[siID][startTime] then
13:                confSet[siID] ← confSet[siID] ∪ {Mark[siID][startTime]}
14:                step ← getStep()                             ▷ Larger time granularity
15:                startTime ← startTime + step
16:                continue
17:            sInsts[siID] ← startTime
18:            fail ← false
19:            for j ← BackTo[siID][startTime] to siID − 1 do
20:                if ¬Consistent(sInsts[j], sInsts[siID]) then
21:                    confSet[siID] ← confSet[siID] ∪ {j}
22:                    Mark[siID][startTime] ← j
23:                    fail ← true
24:                    break
25:            if ¬fail then
26:                Mark[siID][startTime] ← siID − 1
27:                successful ← true
28:            BackTo[siID][startTime] ← siID
29:            step ← getStep()                                 ▷ Larger time granularity
30:            startTime ← startTime + step
31:        if ¬successful then
32:            if confSet[siID] == ∅ then return false
33:            j ← Max(confSet[siID])
34:            confSet[j] ← confSet[j] ∪ confSet[siID] \ {j}
35:            for k ← j + 1 to |sInsts| − 1 do
36:                for v ← 0 to HP − 1 do
37:                    BackTo[k][v] ← Min(BackTo[k][v], j)
38:            while siID > j do
39:                nextStart[siID] ← 0                              ▷ Reduce domains
40:                confSet[siID] ← ∅
41:                siID ← siID − 1
42:        else
43:            nextStart[siID] ← startTime
44:            siID ← siID + 1
45:    return sInsts
```

5 Program

The algorithms were implemented in Java 8 and designed in a way that aggregates methods common for several solvers and hence allows easy adding of new solver algorithms. General workflow of the implemented program is shown in Figure 3. Proposed algorithms implement the step `Schedule`; otherwise, the framework is common for all solvers.

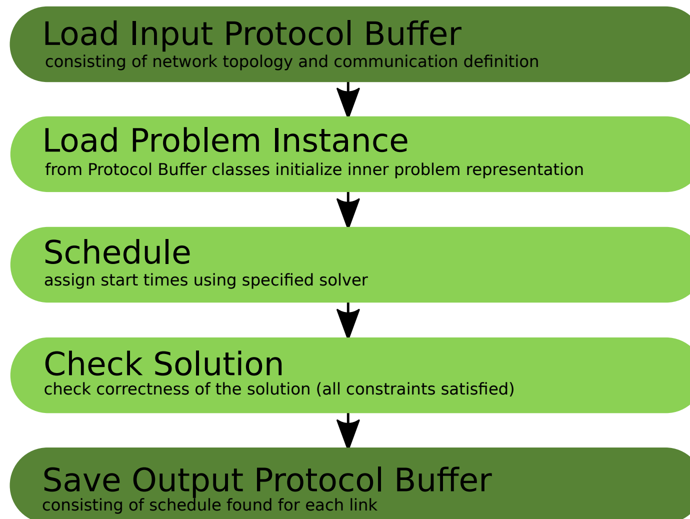


Figure 3: Workflow of the proposed program

The ILP model was implemented using the Gurobi Optimizer [18] which is a mathematical programming solver known for good performance and easily understandable API. This solver also provides useful outputs about the quality of the solution such as upper bound on distance from the optimum.

The Maven build system is used to build the project. Both input and output data are kept in the Protocol Buffer format, which is a language-neutral tool for serializing structured data. The Protocol Buffer definitions were compiled using `protoc 3.7` compiler into Java classes. The JavaDoc documentation was autogenerated using Idea IntelliJ. Since the JavaDoc does not support Protocol Buffer format, tool `protoc-gen-doc` was used for API documentation.

Suitable Java graphical environments for plotting custom graphs are rather scarce. Most of the available libraries are outdated, not very well documented or contain too advanced features for a simple GUI. In the end, Graph Stream library was used for viewing the network topology, and JFreeChart library was used for the implementation of the Gantt chart. Otherwise, the GUI is based on Java Swing.

Java Lombok plugin (allowing auto-generation of methods like getters, setters, etc.) was used to make the implementation more transparent. To install the program, it should be sufficient to have Java 8, Gurobi 8.1 (licensed) and Maven installed and build the code using the enclosed `pom.xml`.

To process the results, we created a Python script `new_stats.py` with automated figure plotting. The script assumes folders aggregating instances with the same meta parameters (the folders are indexed in ascending order). Example usage of the script would be `python new_stats.py 3000 3360 28 > stats.txt`, which would process the results of 28 different solver methods from folders with indexes from 3000 to 3360 and redirect the text output to file `stats.txt`. The figures would be saved to folder `fig/`. The code is written in Python 3.7, and used libraries are Matplotlib and Pandas.

Further, we show the package structure of the code and describe the content of the packages. Detailed documentation can be found on the enclosed CD in JavaDoc format. There are five runnable classes – `Main` for running the scheduler, `ParallelScheduler` running larger experiments, `InstanceGenerator` for local generation of a few instances, `ParallelGenerator` for a parallel generation of experiment datasets and `GUI` which allows solving one selected instance by one selected method and view its schedule, topology, and setup.

```
cz.cvut.ciirc ...............main classes like ProblemInstance and Scheduler
├── data_format ................ autogenerated classes from .proto definitions
├── data_format_definitions ................protocol buffer definitions of API
├── exceptions ...........................................custom exceptions
├── generator ....................instance generator and parameters constants
├── gui .................runnable GUI class and all the necessary components
├── helper ...................common static methods such as IO handling, etc.
├── network_and_traffic_model .......classes for inner problem representation
└── scheduling .......................abstract solver classes, solution class
    ├── baseline_methods ....................ILP, RandomHeuristic, exact CBJ_BM
    ├── helper_classes .................... classes such as LinkTimeSlots, etc.
    ├── multi_pass .................................all multi pass heuristics
    └── one_pass .......................................all one pass heuristics
```

5.1 User Manual

This brief user manual will guide the user throughout the program usage without any need to modify the code. Please note that the following path definitions follow Linux convention, adapt them to your system accordingly. To be able to run the ILP solver, you must have Gurobi installed.

1. Create a custom named folder `CUSTOM_FOLDER` and place the `scheduler.jar` into this folder. Create folder `CUSTOM_FOLDER/instances/instance_dirID`.

2. In the created folder, define you protocol buffer input file called `instance_fileID.pb`. As of May 2019, the protocol buffer can be generated in Java, Python, Objective-C, C++, Dart, Go, Ruby, and C#. The input file must follow the `.proto` definitons contained in `data_format_definitions` and described in `API_documentation.html`.

3. Run `java -jar scheduler.jar` from your `CUSTOM_FOLDER` to start the program.

4. Select the instance you would like to solve and press *Open*.

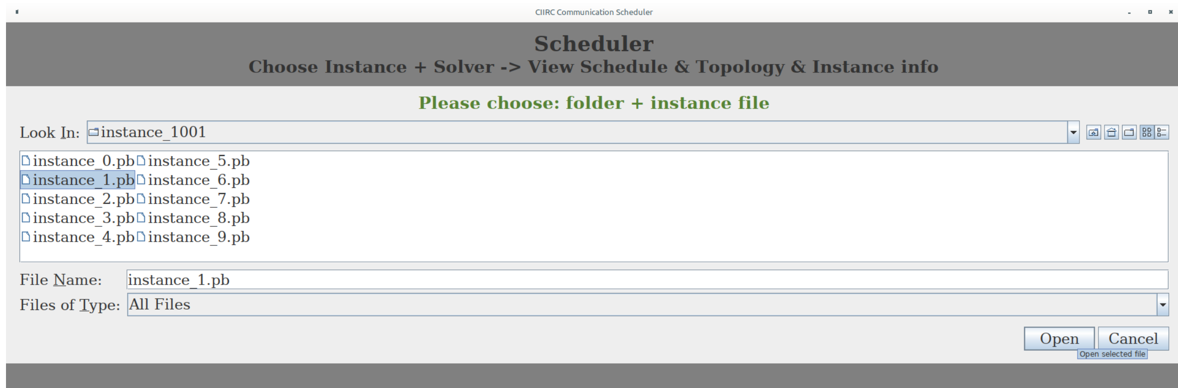


Figure 4: Manual step 1 - Choose the file

5. Select the desired solver from the list and press *Solve*. Wait until the instace is solved.

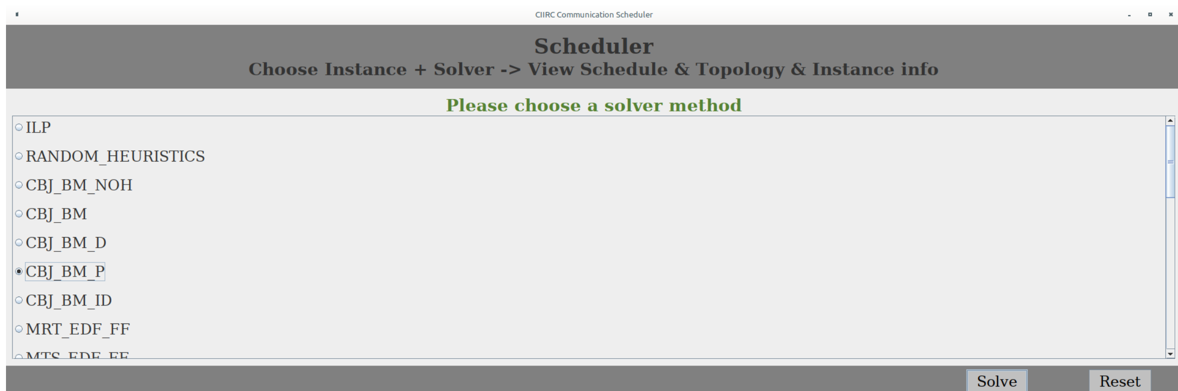


Figure 5: Manual step 2 - Choose the solver

6. Press *Topology* to view the network topology of the instance.

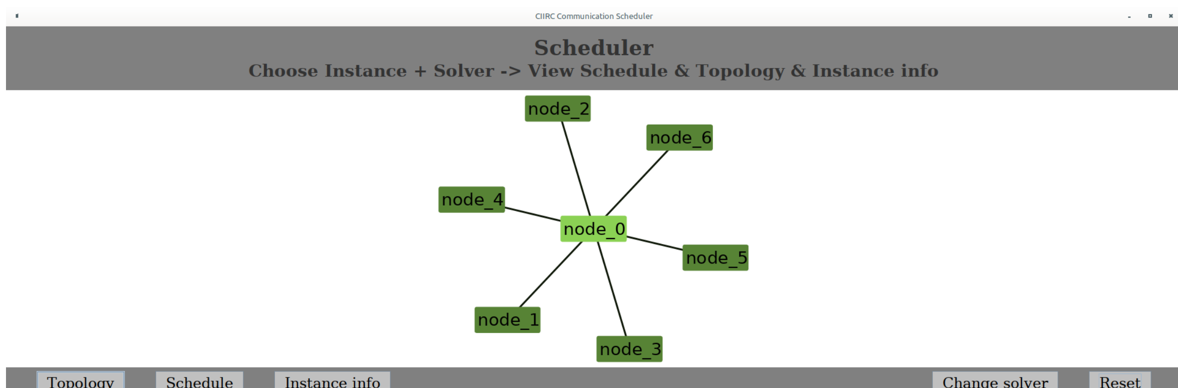


Figure 6: Manual step 3 - View the topology

7. Press *Schedule* to view the schedule of the instance. Zoom in the Gantt chart to have a closer look. If desired it is possible to have labels added to each frame in the Gantt chart as shown in Figure 2. To do so set `GUIConstants.showLabels = true` in the code.

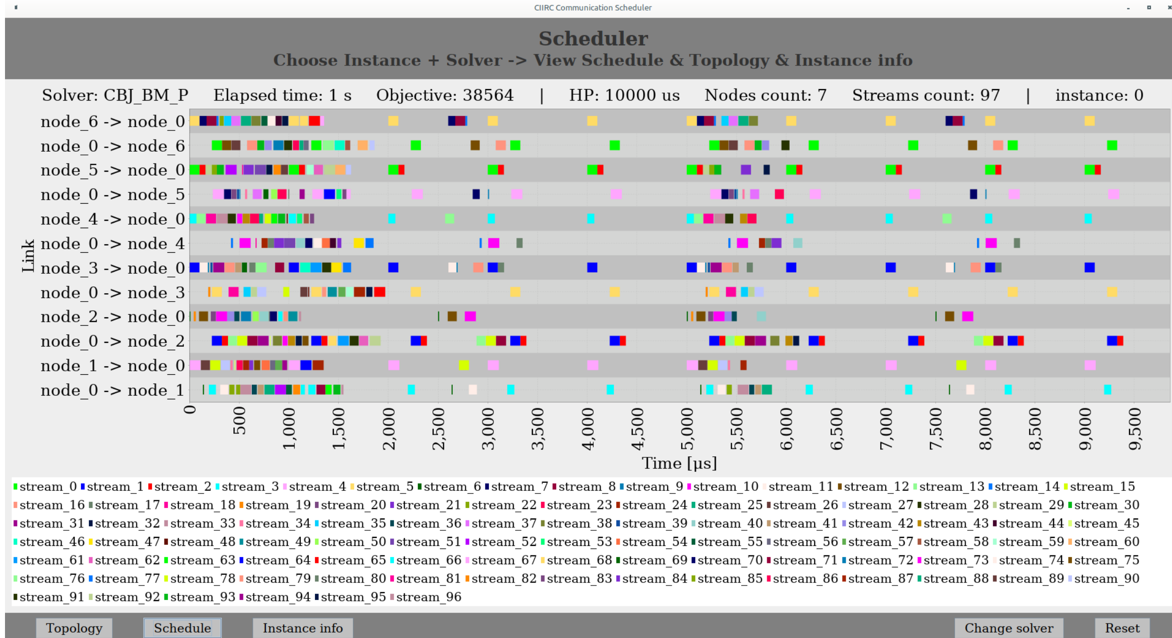


Figure 7: Manual step 4 - View the schedule

8. Press *Instance info* to view the overview of the input data.

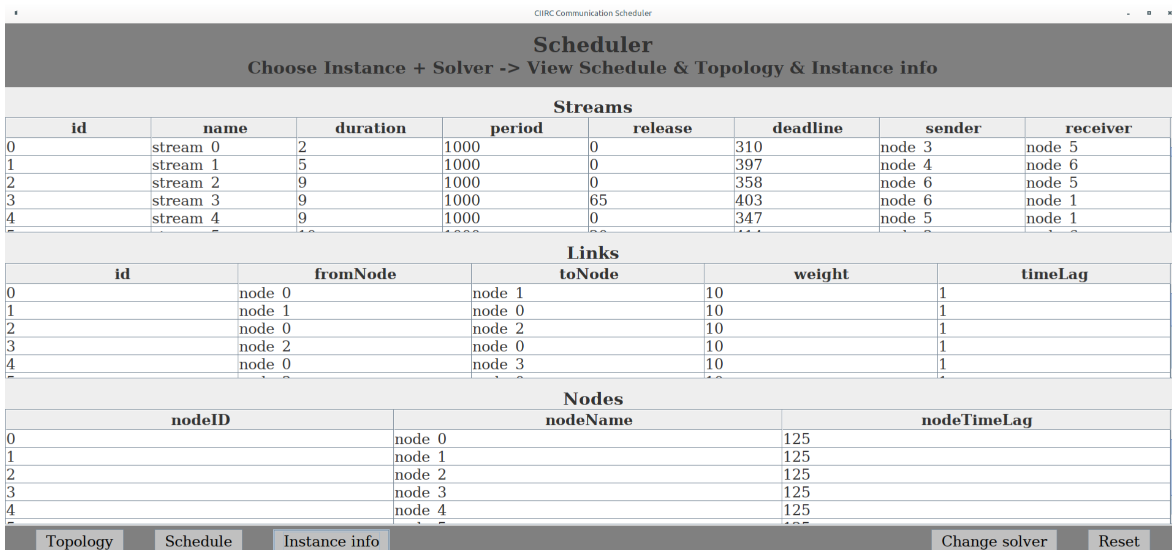


Figure 8: Manual step 5 - View the instance

9. Press *Reset* to change the instance or press *Change solver* to use different solver for the same instance.

6 Experiments

To evaluate the proposed algorithms we conducted experiments on artificially generated data. The generation methodology is described in the following section. Moreover, we compare the performance of selected methods (with a high enough percentage of scheduled instances).

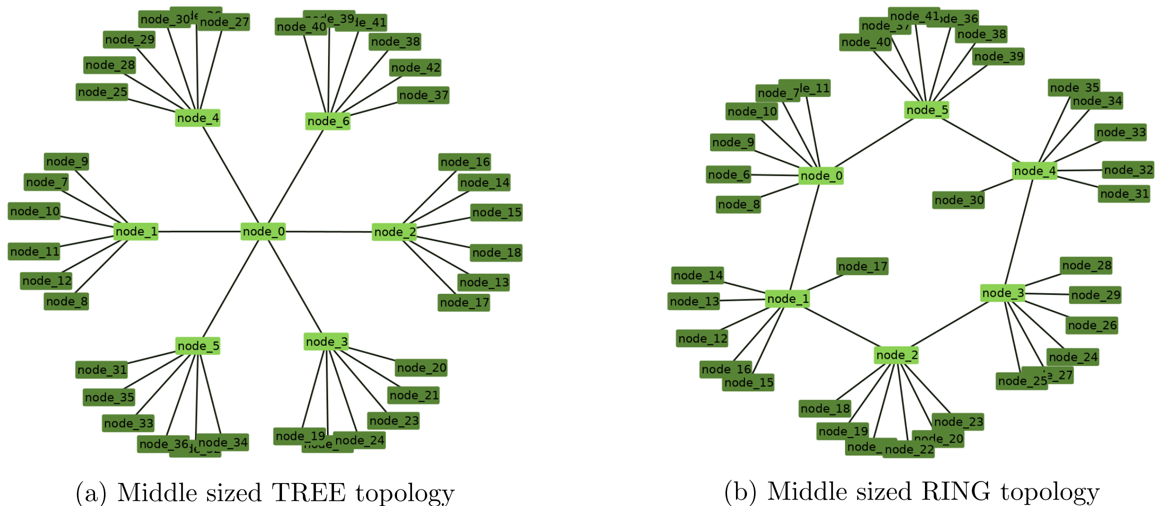
6.1 Experiments Setup

We evaluated the proposed algorithms on randomly generated instances suitable for highly critical Ethernet communication. We generated schedulable instances of various sizes and topologies with the aim to have instances with sequentially growing average link utilization allowing us to test both easily and hardly schedulable instances.

In this chapter we will use additional terminology for the network nodes – *end systems* are network nodes that are connected to the rest of the network by one duplex link only (e.g., leaf of a tree graph), *switches* are any other network nodes that are not end systems (i.e., node acting as an intermediate for other nodes). Time units used in experiments are microseconds.

The instance generation can be simplified into two main steps – *topology generation* of all the network nodes and connections between them and *communication generation* of all streams that are sent between the end systems.

The *topology generation* was inspired by Craciunas et al. [19] who designed several industrial-sized topologies for time-triggered scheduling in distributed systems. We run the experiments on three topology sizes, SMALL, MEDIUM, and LARGE, ranging from a couple of switches to several tens of switches (see Table 4). The topologies are of three different types – TREE, RING, and LINE. Example middle sized topologies of each type are depicted by our GUI in Figures 9a, 9b and 10 respectively. In total, we have nine different topologies to test on.



(a) Middle sized TREE topology

(b) Middle sized RING topology

Figure 9: Sample middle sized TREE and RING topology

Parameters of nodes and links in the network are chosen to suit the Ethernet communication problem and are similar to [19]. Link weight $e_k.w$ is set to 1 for links between two switches and 10 for links between the switch and end system, representing physical Ethernet link of speed 1 Gbit/sec and 100 Mbit/sec respectively. Similar as in [6], the link time lag $e_k.l$ is set to 1 μs for all links, representing the propagation delay which is equal to *wire length/speed of light* and the node time lag is set to 10 μs for all nodes.

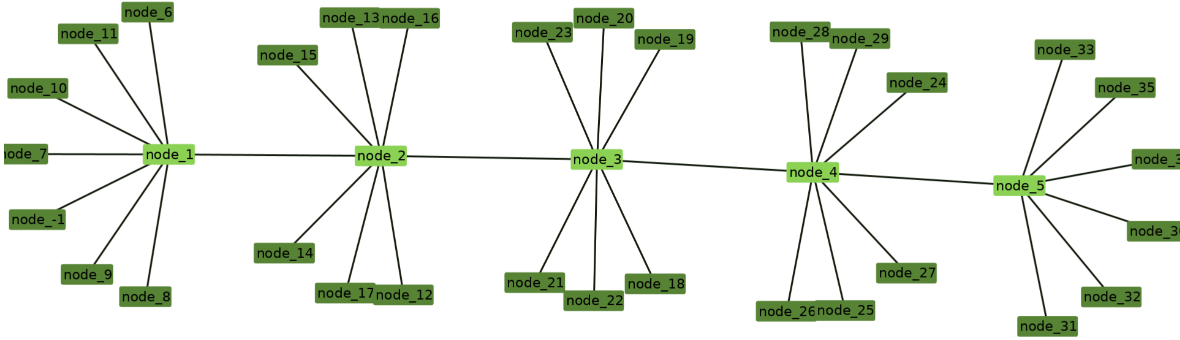


Figure 10: Middle sized LINE topology

The *communication generation* depends on the topology – Streams are generated to be sent between the end systems. In RING and TREE topology, a stream can be sent between any two end systems. In LINE topology the communication takes place only between one specified end system (control unit) and the other end systems.

The streams have a different period $s_j.T$ which is uniformly chosen from one of the three predefined period sets (for one generated instance, one period set is used) with values ranging from 1 ms to 16 ms and hyper period not larger than 16 ms. All period sets are shown in Table 3.

periodSet	**periods** (μs)	**HP** (μs)
P_1	$\{1000, 2500, 5000, 10000\}$	10000
P_2	$\{5000, 7500\}$	15000
P_3	$\{2000, 4000, 8000, 16000\}$	16000

Table 3: Period sets

To model communication of different complexity, the total number of reoccurred stream instances rsi was adjusted.

$$rsi = \sum_{s_j \in S} \frac{HP}{s_j.T} \cdot |R_j| \tag{8}$$

We set upper and lower bound on rsi for each topology type and size as shown in Table 4 and iterate from lower to upper bound with a step size equal to the interval size divided by 20 resulting in 20 different communication complexities for each topology. Upper bound ub on rsi was set experimentally for each combination of topology size and topology type as the maximum number of frames for which the generator yielded a schedulable instance in a reasonable time (in order of hundreds of seconds). Lower bound lb was set as $lb = \frac{ub}{10}$ for each setting.

topMode	topSize	numSwitches	numEndSystems	lb on rsi	ub on rsi
TREE	SMALL	1	6	60	600
TREE	MEDIUM	7	36	1600	16000
TREE	LARGE	21	64	2000	20000
RING	SMALL	2	6	200	2000
RING	MEDIUM	6	36	1600	16000
RING	LARGE	14	70	2000	20000
LINE	SMALL	1	4	160	1600
LINE	MEDIUM	5	31	800	8000
LINE	LARGE	13	66	1800	18000

Table 4: Parameters of different topologies

The stream duration is set randomly to correspond to an Ethernet packet of size 125–1500 bytes transmitted through the 1Gbit/s network. Release time $s_j.r$ and deadline $s_j.\tilde{d}$ are randomly set so that the interval $[s_j.r, s_j.\tilde{d}]$ covers 15–40 % of the period $s_j.T$ and the interval is large enough to cover the sum of transmission durations over all reoccurred stream instances on the path of the stream.

Summarized we have four different parameters defining one instance folder `{topologySize, topologyMode, periodSet, rsi}` resulting in $3 \cdot 3 \cdot 3 \cdot 20 = 540$ folders in total. Each folder consists of 100 instances with the same quadruple of these parameters – the instances differ in the generated communication which is partially random. The main generation cycle is shown in Algorithm 6.

Algorithm 6 Benchmark generator

1: **procedure** GenerateBenchmarks
2: **for all** $topologySizes$ **do**
3: **for all** $topologyModes$ **do**
4: **for all** $periodSets$ **do**
5: $lb, ub, step \leftarrow get(topologyMode, topologySize)$
6: **for** $rsi \leftarrow lb, ub, step$ **do**
7: **for** $i \leftarrow 1, numInstances$ **do**
8: $topology \leftarrow$ GenTopology($topologyMode$, $topologySize$)
9: $streams \leftarrow$ GenStreams($topology$, $periodSet$, rsi)
10: **if** $!valid(streams)$ **then**
11: $discard(streams, topology)$

The pseudocode for communication generation is described in Algorithm 7. The streams are generated using a first fit heuristic approach described in 2.2.1. The generator keeps an online schedule of so far generated streams and updates it each time a new stream is generated. The streams are generated until the desired rsi bound is reached or the instance generation is not possible for given parameters (and generated random numbers).

A priority queue `fromToPQ` (Algorithm 7, line 6) is used for choosing origin and target nodes of the streams. Items in the queue consist of quadruple origin, destination, current period and the last usage. Before the communication generation starts, the `fromToPQ` is ini-

Algorithm 7 Communication generator

1: **procedure** GenStreams($topology, periodSet, rsi$)
2: $rsiID \leftarrow 0$
3: $streams \leftarrow \{\}$
4: $HP \leftarrow lcm(periodSet)$
5: $schedule \leftarrow$ initialize empty schedule for all links
6: $fromToPQ \leftarrow$ initialize priority structure
7: **while** $rsiID < rsi$ **do**
8: $curP \leftarrow$ select period based on $rsiID$
9: **if** $fromToPQ.peek().period > curP$ **then**
10: **return** null
11: $curPeekPQ \leftarrow fromToPQ.pop()$
12: $path \leftarrow$ find a path from $curPeekPQ.org$ to $curPeekPQ.trg$
13: $success \leftarrow$ TryToPlace($curP$, $path$, $schedule$, $streams$, $topology$)
14: **if** $success$ **then**
15: $rsiID \leftarrow rsiID + (HP/curP) * length(path)$
16: $fromToPQ.insert(curPeekPQ, curP, rsiID)$
17: **else**
18: $nextP \leftarrow$ select next period from the periodSet
19: $fromToPQ.insert(curPeekPQ, nextP, rsiID)$
20: **return** streams

tialized with all possible combinations of end systems, the lowest period from the period set and negative random number (for the last usage). The top of the priority queue is an element with the lowest current period and in the case of a draw the element with the lowest last usage. The lowest current period represents the lowest period for which the stream of given origin and target can theoretically be placed to the schedule. The last usage is a timestamp (represented by a current number of reoccurred stream instances in the network) marking the last usage of the given origin – target combination. Initializing the last usage to a random number ensures that the priority queue is different each time it is initialized. The `fromToPQ` structure guarantees that the communication generation stops in a finite time (Algorithm 7, line 10) and that the streams are as uniformly distributed as possible between different combinations of origin and target end systems.

Periods are distributed uniformly with respect to the number of reoccurred stream instances `rsi`. The period set is sorted from the lowest to the largest value. The period `curP` for the currently generated stream is selected based on the progress of already placed reoccurred stream instances in the schedule which is corresponding to the ratio $rsiID/rsi$. Please note that since rsi is calculated as in Equation (8), the actual number of added reoccurred stream instances of the period $T \in P_i$ is not equal to $rsi/|P_i|$ but belongs to the interval

$$\left(\frac{rsi}{|P_i|} - \frac{HP}{T} \cdot d, \frac{rsi}{|P_i|} + \frac{HP}{T} \cdot d\right)$$

where d is the longest path between any two end systems in the network.

In case there is no combination of origin and target nodes that would allow us to place the stream with period `curP`, the instance is discarded. The stream with parameters origin, target, period is valid only if it is possible to place it to the current schedule using the first fit approach 2.2.1 so that it meets all constraints (2)–(5) – in pseudocode procedure `TryToPlace` (Algorithm 8). In case the stream placement is possible, the online schedule is updated. Otherwise, the stream payload is being iteratively decreased. After reaching the lower bound for the stream transmission duration, the current stream is discarded and different combination of origin and target is chosen from the priority structure.

Algorithm 8 Placing stream in the schedule

1: **procedure** TryToPlace($curP, path, schedule, streams, topology$)
2: $duration \leftarrow randomStreamDuration(1, 12)$
3: **while** $duration > 0$ **do**
4: $schedule \leftarrow placeFirstFit(curP, path, schedule, streams, topology, duration)$
5: **if** $valid(schedule)$ **then**
6: **return** true
7: **else** $duration \leftarrow duration - 1$
8: **return** false

6.2 Results

To test the proposed algorithms on generated instances, we implemented all of them in Java 8, and for the ILP model we used Gurobi solver version 8.1. To ensure the same environment for both ILP and the other algorithms, the number of threads that Gurobi is allowed to use was set to one. The experiments were run on a system with 4x Intel® Xeon® CPU E5-2690 v4 @ 2.60GHz with 14 cores (56 cores in total) and in total 251GB of RAM. We set the time limit to 60 seconds per one solver. Note that the problem instance initialization (data loading, etc.) is not included in this time limit and is done in order of seconds. The problem instances were run in parallel on 56 threads. The parallelization was done in a way that allows the solvers to run without any common resources, only the call to create Gurobi environment is locked for synchronization safety reasons. In general, compared to single thread execution, the delay of one solver resulting from the parallelization is negligible.

6.2.1 Structure of Generated Instances

Due to the large number of input parameters, the difficulty of each instance is hard to estimate. However, we can still point out some trends based on the results of the first experiment. Each instance is influenced by several factors.

Firstly, we point out the topology type and size which influences the importance of bottleneck link. We call the link a bottleneck when it gathers significantly more communication traffic (i.e., the link has high utilization) than most of the links. Such bottleneck link then determines the throughput of the whole network. For LINE topology, the bottleneck is easy to determine. Since all communication includes the control unit, the bottleneck link is the

one connecting the control unit to the rest of the network. This topology is extreme in a way that all streams in the network cross the bottleneck link. For TREE and RING topologies, the bottleneck link is not that clear and depends on the distribution of streams among end systems. In Figure 11, we can see the average and maximal link utilization of all instances. We can see that the average and maximal utilization do not have the same distribution. This is caused by the fact that some of the links transfer more traffic than the other links. For this reason, we will need to analyze each topology type and size separately because the number of rsi in the network does not necessarily correspond to the instance difficulty. In Appendix C we can see the utilization for each topology type and size separately.

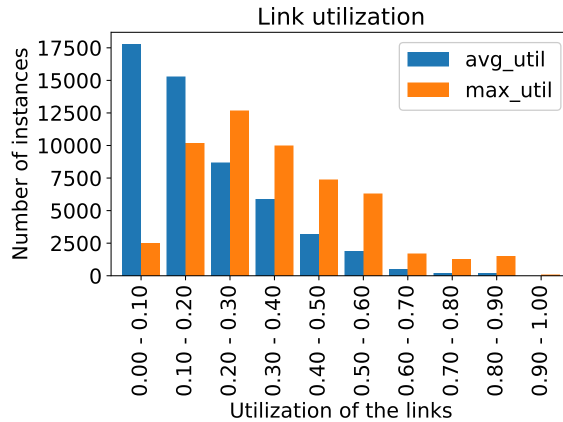


Figure 11: Link utilization of all topologies

Secondly, the period set can be described by the number of periods, the average value of the period, the hyper period and if the periods are harmonic or not. All of these factors influence the instance difficulty. Table 5 shows the dependence of the number of scheduled instances on the period set. We say that the instance is scheduled if there was at least one method that found a solution. The most successful was the harmonic set P_3 with 99.37 % of scheduled instances. On the other hand, set P_1 which is not harmonic and has four different periods, had only 71.27 % of scheduled instances.

periodSet	**periods** (μs)	**HP** (μs)	**scheduled**	**scheduled** (%)
P_1	$\{1000, 2500, 5000, 10000\}$	10000	12829	71.27
P_2	$\{5000, 7500\}$	15000	16012	88.96
P_3	$\{2000, 4000, 8000, 16000\}$	16000	17886	99.37

Table 5: Number of scheduled instances based on period set

6.2.2 One Pass Heuristics

The first experiment was performed on one pass heuristics and the baseline methods. In Table 6 we can see the results. Each heuristic is called by its criteria – e.g., *RED_EDF* is a heuristic where the first criterion is the Resource Equivalent Duration and the second criterion is the Earliest Deadline First.

method	scheduled	avg time [s]	best obj
EDF_MRT	40345	0.03	9836
EDF_RED	40343	0.06	9706
DF_RED	40267	0.06	3556
DF_MRT	40199	0.03	3742
DF_EST	20918	0.01	807
DF_LST	20871	0.01	199
LST_EDF	20804	0.01	190
EDF_EST	20796	0.01	324
EDF_LST	20788	0.01	301
EDF_MTS	20785	0.01	315
DF_MTS	20385	0.01	413
ILP	18885	48.53	18769
MTS_EDF	12811	0.01	242
MRT_EDF	9102	0.00	7
EST_EDF	7631	0.01	133
RED_EDF	3657	0.03	1
RANDOM	3352	0.00	197
DF_MSLK	136	0.00	0
EDF_MSLK	117	0.00	1
MSLK_EDF	96	0.00	0

Table 6: Performance of One Pass Heuristics and Baseline methods

Three performance measures were taken – the number of scheduled instances, the average running time and the number of best objective values. As a reminder, we must mention that all of the instances were generated in a way that ensures they are schedulable (i.e., some solution exists for each one of them).

In total, we generated 54000 instances, which means that the most successful method *EDF_MRT* solved 74.7 % of the instances. The other similarly successful methods (74.4–74.7 %) were alternations of the aforementioned method. The most successful method that has a complement criterion as the first priority value was *LST_EDF* with 38.5 % of scheduled instances. The only criterion that seems useless for our problem is *MSLK*. This may be caused by the fact that it diminishes the release time and the deadline of the stream. Our baseline methods ILP and Random Heuristic scheduled fewer instances than most of the heuristic methods. The success rate of *ILP* was 34.9 %, and the success rate of *RANDOM* was 6.2 %.

The number of best objective values is calculated as a sum of the instances where the method obtained the best objective value among others. If more methods obtained the best objective value for some instance, all of these methods get a score point. As expected, the *ILP* did very well in this performance measure – for 34 % of instances, it found a solution with the lowest objective value. As we mentioned earlier, the *ILP* is a complete and optimal method. However, we can see that there is a 0.9 % difference between the scheduled and best objective instances. This is caused by the Gurobi solver, which may return a sub-optimal solution in case the time limit is reached. The performance of the heuristics with respect to the best

objective metric was similar as for the number of scheduled instances – none of the heuristics was exceptional in this performance measure.

The average running time for all heuristics was in the order of hundredth of a second. The average running time for the *ILP* was 48.53 seconds. However, it is important to point out that as opposed to heuristics, the *ILP* did not return the first feasible solution but instead continued in searching for the optimal one. This is partially causing the larger running time of the *ILP*. However, it is clear that the heuristics run much faster.

6.2.3 Multiple Pass Heuristics

We based our multiple pass heuristics on the best performing one pass heuristic *DF_MRT*. Table 7 compares the original *CBJ_BM_NOH* implementation with the enhanced implementation *CBJ_BM* and the implementations skipping some domain values *CBJ_BM_D*, *CBJ_BM_P* and *CBJ_BM_ID*.

method	scheduled	avg time [s]	best obj
CBJ_BM_D	44981	15.32	17026
CBJ_BM	44931	15.38	9340
CBJ_BM_P	41780	14.35	507
CBJ_BM_ID	40731	15.35	456
ILP	18885	48.53	18738
CBJ_BM_NOH	4637	55.01	466
RND	3352	0.0	12

Table 7: Performance of Multiple Pass Heuristics and Baseline methods

The most scheduled instances were obtained by *CBJ_BM_D* heuristics. It found a solution for 83.3 % instances. The complete version *CBJ_BM* obtained comparable results with 83.2 % of scheduled instances. The reason why these two methods behave similarly is that the step size for *CBJ_BM_D* is one for stream instances passing between switches. Hence, these two methods act the same on stream instances between switches. The step size for stream instances passing between end systems and switches is between one and twelve – the step size is affected by the stream transmission duration and the link speed. The other two heuristics do not perform significantly worse with 75.4–77.4 % of scheduled instances. However, they do not show better results than the *CBJ_BM_D* heuristic for any combination of instance parameters. The original implementation without the heuristical enhancement found a solution for 8.58 % instances.

The best objective values were (apart from ILP) obtained by *CBJ_BM_D* heuristics. If we remove all methods except for *CBJ_BM_D* and *CBJ_BM* from the statistics, we obtain best objective values 26313 (48.7 %) and 22615 (41.9 %) respectively. This is significantly better than the difference in the number of scheduled instances. The rationale may be that larger step size sometimes skips the first available time slot for the given stream instance. The larger the transmission duration of the stream instance is, the more available time slots it may skip. This allows shorter (in transmission duration) stream instances to fit into the empty gap.

Hence, the end-to-end latency of the shorter stream instances may be lower. Nevertheless, this hypothesis would need a deeper exploration of the results to confirm it.

The time performance for all heuristic *CBJ_BM* based methods is similar (14–16 s). The *CBJ_BM_NOH* method timed out on most of the instances, and it is also reflected on the average solving time 55 s.

6.3 Discussion

To have a clear visual interpretation of the results, we have chosen to plot only results for the best performing one pass heuristic *EDF_MRT*, the best performing multiple pass heuristic *CBJ_BM_D* and the *ILP* method. However, it is possible to run the enclosed plotting script `python new_stats.py 7000 7540 25 > results.txt` on any other combination of methods if needed. The figures showing the performance measures of the selected methods for each topology type and size can be found in Appendices D – G.

Another interesting performance measure can be found in Appendix E where we show the dependence of success rate (percentage of scheduled instances) on the average and maximal link utilization of the instance. We can see that the maximal link utilization is a slightly better indicator of instance difficulty.

Generally speaking, ILP performs very well on small instances with low maximal link utilization (up to 40 %). Since it also finds the optimal solution, it does not make much sense to compete with ILP on such instances. On the other hand, for larger or more utilized networks it is beneficiary to use the heuristic methods. We can see that for large instances, the *CBJ_BM_D* success rate 85.2 % was significantly better than the ILP success rate of 12.0 %.

Further, we compare objective values of *ILP* and *CBJ_BM_D* on instances where *ILP* found an optimal solution. The average objective value of the solution found by *CBJ_BM_D* was 101.7 % higher than of the solution found by *ILP*. We could improve this measure by introducing a version of *CBJ_BM_D* that is optimal and it will be one of the directions for the future work.

If we compare the results of the best multiple pass (83.3 % success rate) and the best one pass heuristics (74.7 % success rate), we find out that the introduction of backtracking allows us to have 8.6 % better difference in the percentage of scheduled instances. In Table 6 we see that the run time of one pass heuristics is negligible as opposed both to *ILP* and *CBJ_BM* based methods. Hence, we could run all of the implemententen one pass heuristics and then choose the one that yielded the best solution for the given instance. In such case, the success rate of the combined one pass heuristics would be 78.7 %.

7 Conclusion

The aim of this thesis was to propose several heuristics for highly critical periodic scheduling and to compare their performance. The main focus has been taken on developing methods that are fast and reliable as well as on developing an easily extensible code. We proposed 25 different methods that can be divided into three categories. Firstly, we developed an exact ILP-based method which allowed us to compare the performance of the proposed heuristics as well as to double-check their correctness. Secondly, we designed several one pass heuristics based on the first fit approach and the order in which the schedule is created. Lastly, we applied the knowledge gained from the one pass methods and constructed backtracking methods based on Conflict-Directed Backjumping with Backmarking.

The experimental results have shown that the proposed heuristics report more than two times better results in the number of scheduled instances than the baseline *ILP* method. Especially for harder instances, with respect to the topology size and the total number of rsi in the network, they have shown their importance. The main shortcoming of the experiment would be that it was performed on artificially generated data. Even though we tried to design the instance generator as unbiased as possible, it would certainly be beneficial to run the experiments on a real dataset.

Further, we developed a graphical user interface that allows to intuitively run the framework on a single instance and to visually display the results. We defined an API for both input and output data format. The API together with GUI allows the framework to be used as a standalone program. Another important part of the code is the postprocessing script which interprets the obtained results of experiments and automatically plots figures. The proposed script will ease up the future work on the project.

In the future, we would like to further develop the proposed multiple pass heuristics and to speed up the search. We could enhance the implemented *CSP* techniques for example by deeper forward checking. We could also explore other combinations of the one pass heuristics and *CBJ_BM*, some of the proposed one pass rules allow dynamical priority queue sorting which could be even more beneficial when combined with the *CSP* backtracking. The goal would be to find out if the overhead of the additional techniques is effective with respect to the decreased search space size. A deeper exploration should be given to determining the step size in *CBJ_BM* based heuristics.

Another interesting area of focus would be the analysis of the instance difficulty. More specifically, defining the input parameters that influence the chance of the instance to be scheduled the most. The attention given to the deeper exploration of input parameters could also result in developing a metaheuristic that would choose the solver method based on the instance parameters.

References

[1] Erik L. Demeulemeester and Willy S. Herroelen. *PROJECT SCHEDULING, A Research Handbook*. KLUWER ACADEMIC PUBLISHERS, 2002.

[2] Michael L. Pinedo. *Scheduling - Theory, Algorithms, and Systems*. Springer, fifth edition, 2016.

[3] Brendan Galloway and Gerhard P. Hancke. Introduction to Industrial Control Networks. *IEEE Communications Surveys & Tutorials*, 15(2):860–880, 2013.

[4] J.H.M. Korst, E.H.L. Aarts, J.K. Lenstra, and J. Wessels. *Periodic multiprocessor scheduling*. Memorandum COSOR. Technische Universiteit Eindhoven, 1990.

[5] Profibus Internationa. Application layer protocol for decentralized periphery and distributed automation, specification for profinet, iec 61158-6- 10/fdis. Technical report, Profibus International, 2007.

[6] Zdeněk Hanzálek, Pavel Burget, and Přemysl Šůcha. Profinet IO IRT Message Scheduling. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 57–65, July 2009.

[7] Zdeněk Hanzálek, Přemysl Šůcha, and et al. Course presentations for Combinatorial Optimalization. `https://cw.fel.cvut.cz/wiki/courses/ko/start`, mar 2019.

[8] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[9] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010.

[10] John Gary Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1979.

[11] Grzegorz Kondrak and Peter van Beek. A theoretical evaluation of selected backtracking algorithms. *Artif. Intell.*, 89(1-2):365–387, 1997.

[12] Marek Vlk. Dynamic scheduling. Master's thesis, MatFyz CUNI, 2014.

[13] Clément Pira and Christian Artigues. Line search method for solving a non-preemptive strictly periodic scheduling problem. *Journal of Scheduling*, 19(3):227–243, June 2016.

[14] Anna Minaeva, Debayan Roy, Benny Akesson, Zdeněk Hanzalek, and Samarjit Chakraborty. Control Performance Optimization in Time-Triggered Periodic Scheduling. IEEE Transactions on Computers, submitted, 2019.

[15] Ali Syed and Gerhard Fohler. Efficient offline scheduling of task-sets with complex constraints on large distributed time-triggered systems. *Real-Time Systems*, 2018.

[16] Bharat Bansal. Divide-and-Conquer Scheduling for Time-sensitive Networks. Master's thesis, University of Stuttgart, 2018.

[17] Thomas H. Cormen and Thomas H. Cormen, editors. *Introduction to algorithms.* MIT Press, Cambridge, Mass, 2nd ed edition, 2001.

[18] Gurobi. Gurobi optimizer 8.1. `http://www.gurobi.com`, mar 2019.

[19] Silviu S. Craciunas and Ramon Serna Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Real-Time Syst.*, 52(2):161–200, 2016.

Appendices

Appendix A CD Content

Table 8 lists names of all root directories and files present on enclosed CD.

Name	Description
experiment_1	Instances and results of the performed experiment
instances	Sample instances for experimenting with the GUI, processing scripts
JavaDoc	Documentation of the source code
program	Location of scheduler.jar which is a runnable GUI
thesis-code	Source code of the project
API_documentation.html	API documentation of input and output ProtoBuffer files
Brejchova_BP.pdf	Text of the bachelor thesis
ReadMe.html	Guide for project installation

Table 8: CD Content

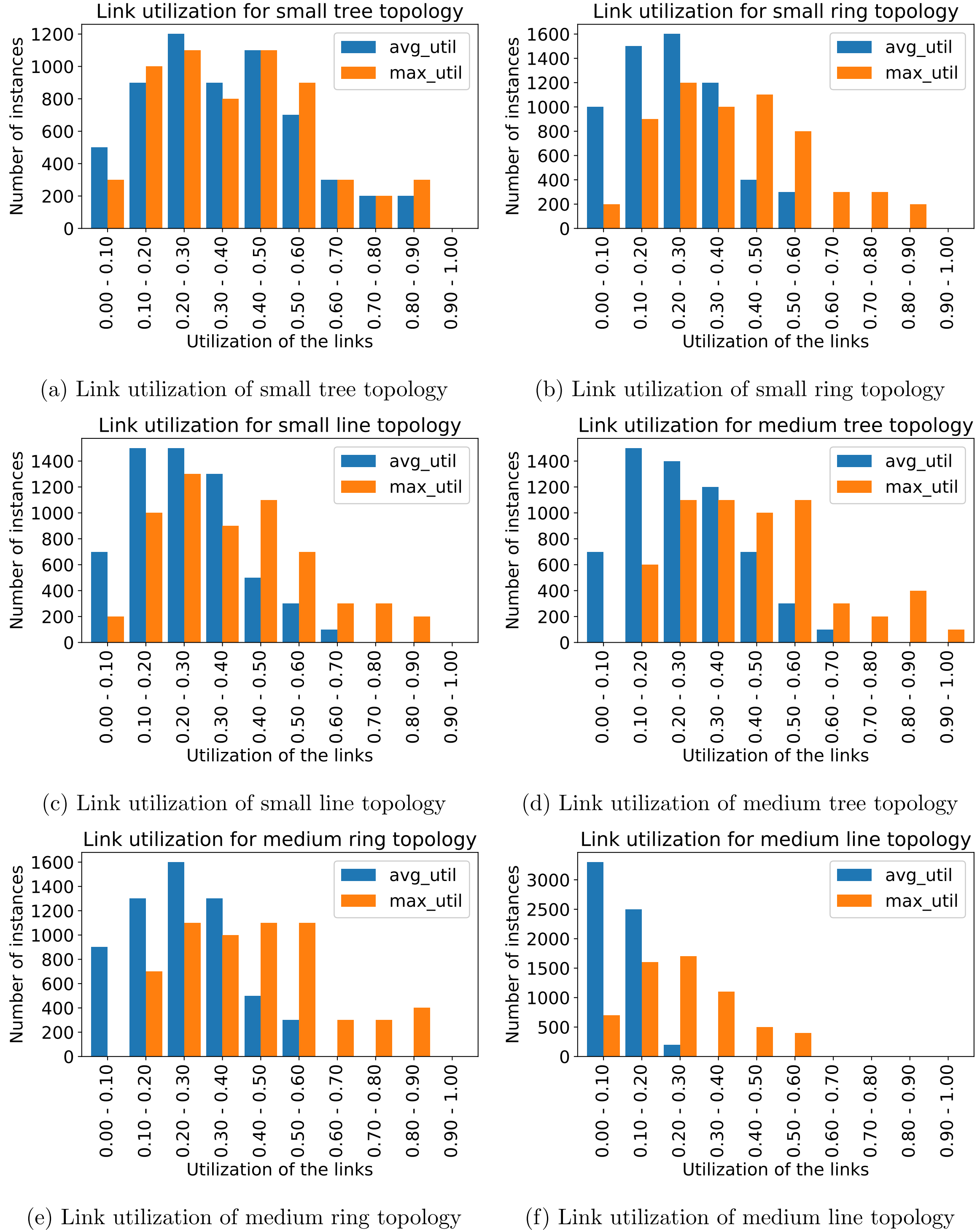
Appendix B List of Abbreviations

Table 9 lists abbreviations used in this thesis.

Abbreviation	Meaning
API	Application programming interface
CBJ	Conflict-Directed Backjumping
CBJ_BM	Conflict-Directed Backjumping with Backmarking
CBJ_BM_D	CBJ_BM with step size based on the duration
CBJ_BM_ID	CBJ_BM with step size based on scheduling progress
CBJ_BM_NOH	CBJ_BM implementation based on the original pseudocode
CBJ_BM_P	CBJ_BM with step size based on stream period
CPU	Central processing unit
CSP	Constraint satisfaction problem
DF	Criterion earliest deadline with lower granularity
EDF	Criterion earliest deadline
EST	Criterion earliest start time
FFS	First fit streams algorithm
FFSI	First fit stream instances algorithm
GUI	Graphical user interface
HP	Hyper period
ILP	Integer Linear Programming
IRT	Isochronous real time
LST	Criterion latest start time
MSLK	Criterion minimum slack
MTS	Criterion most total successors
PQ	Priority queue
RED	Criterion resource equivalent duration
RSI	Total number of reoccurred stream instances
RT	Real-time
UTIL	Link utilization

Table 9: List of abbreviations

Appendix C Average and Maximal Utilization of Links



(a) Link utilization of small tree topology

(b) Link utilization of small ring topology

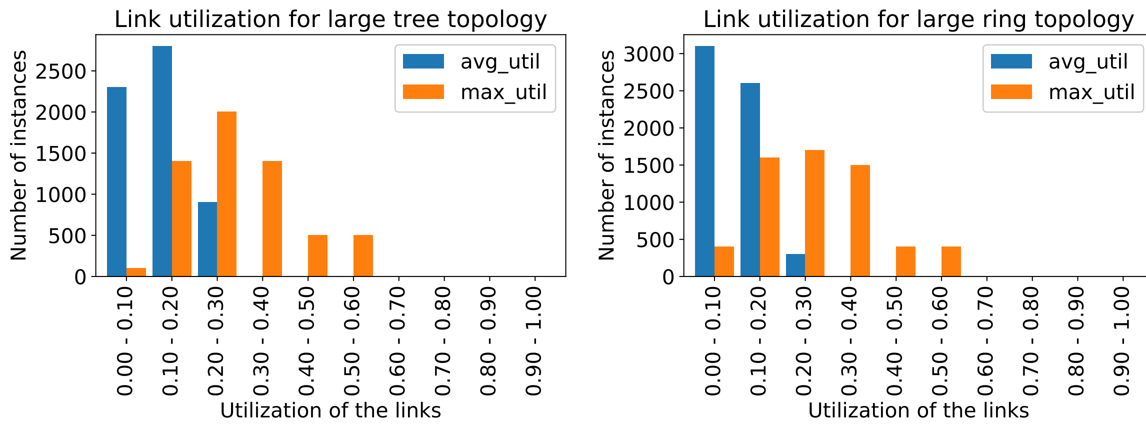
(c) Link utilization of small line topology

(d) Link utilization of medium tree topology

(e) Link utilization of medium ring topology

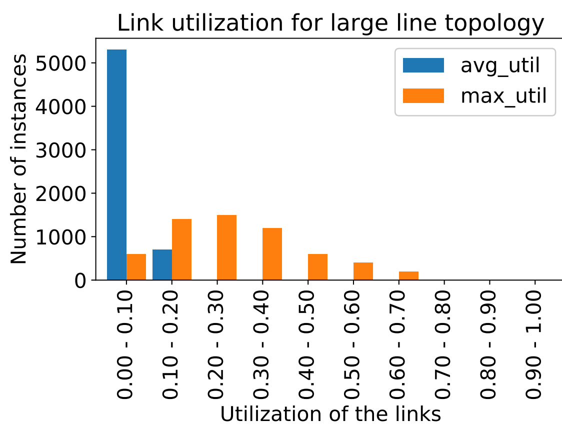
(f) Link utilization of medium line topology

Figure 12: Average and maximal link utilization of small and middle sized topologies



(a) Link utilization of large tree topology

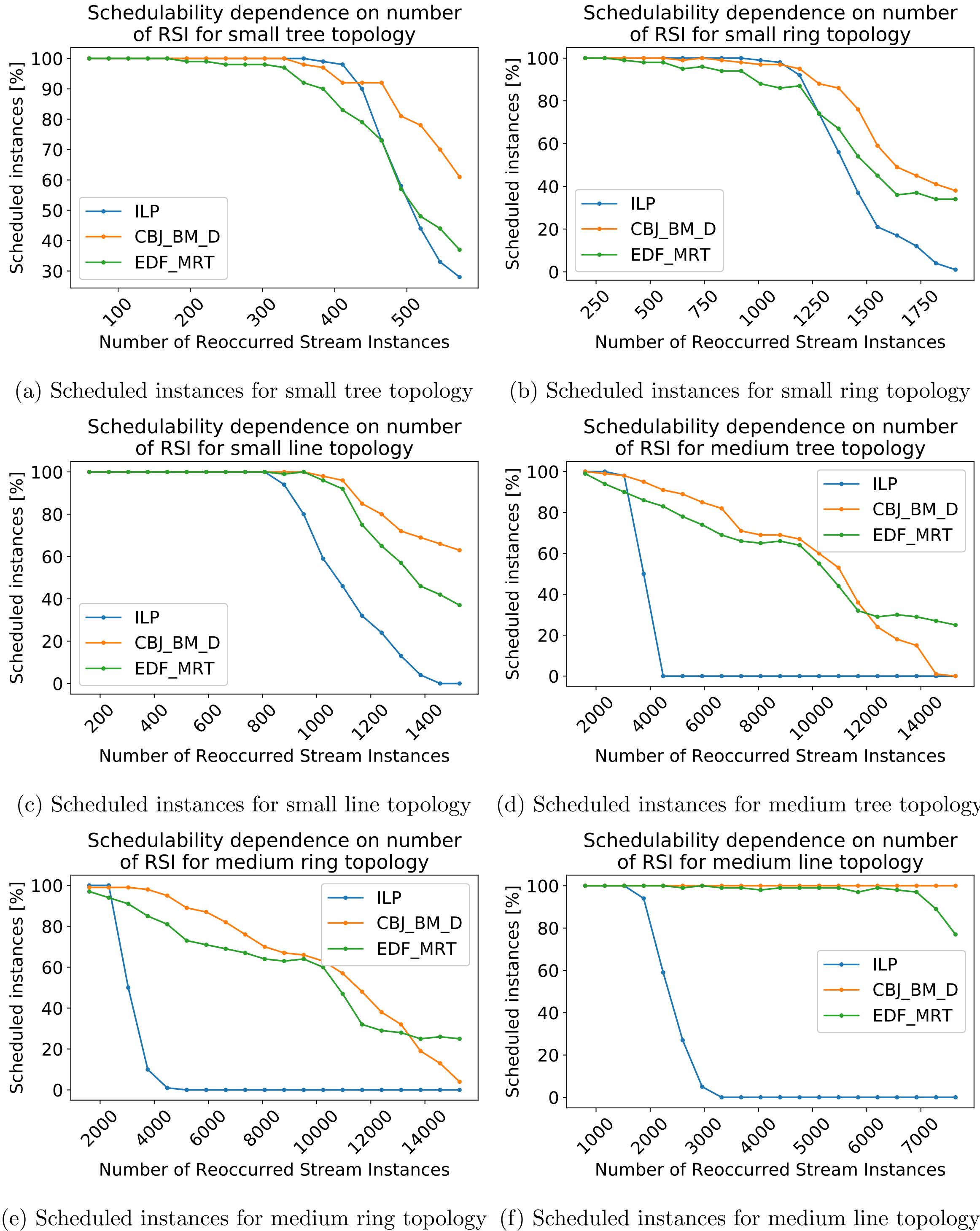
(b) Link utilization of large ring topology



(c) Link utilization of large line topology

Figure 13: Average and maximal link utilization of large topologies

Appendix D Percentage of Scheduled Instances



(a) Scheduled instances for small tree topology

(b) Scheduled instances for small ring topology

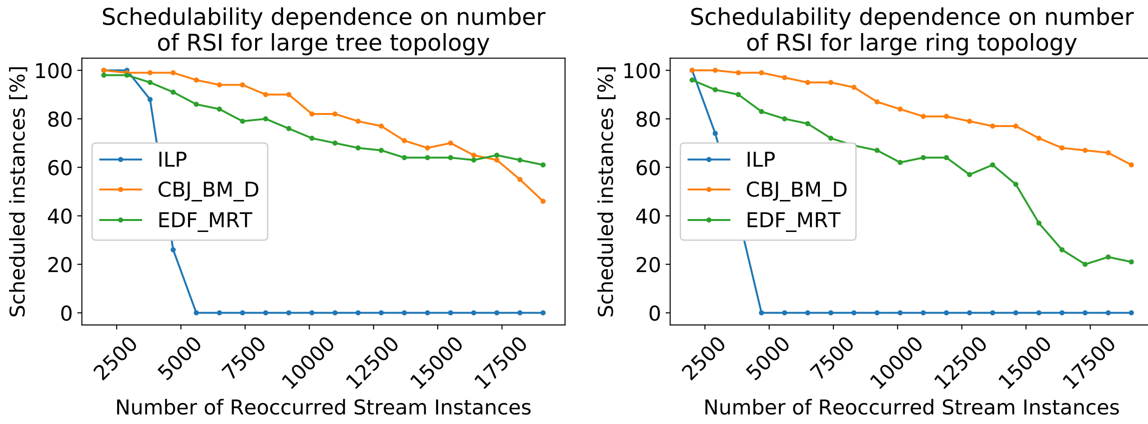
(c) Scheduled instances for small line topology

(d) Scheduled instances for medium tree topology

(e) Scheduled instances for medium ring topology

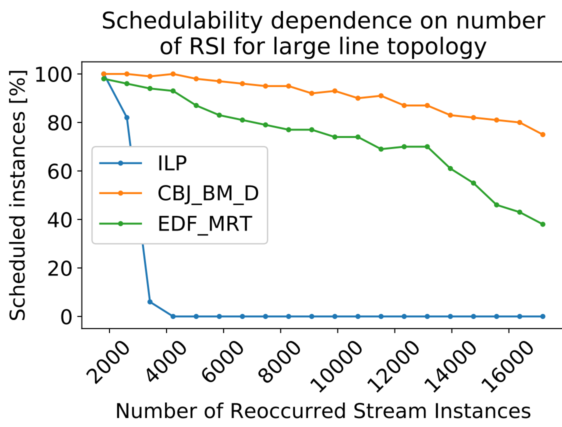
(f) Scheduled instances for medium line topology

Figure 14: Scheduled instances for small and middle sized topologies



(a) Scheduled instances for large tree topology

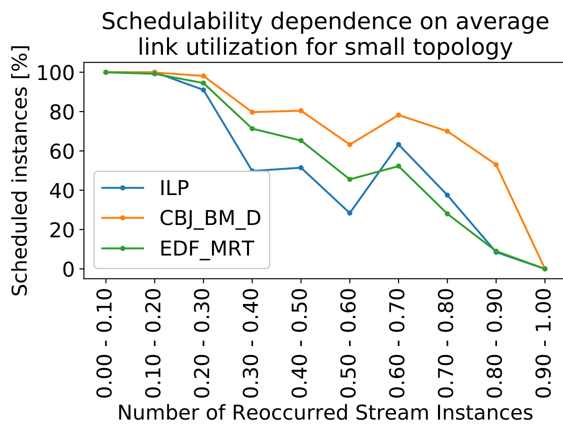
(b) Scheduled instances for large ring topology



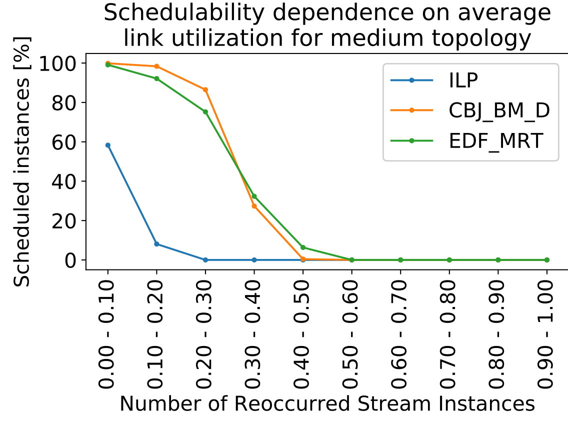
(c) Scheduled instances for large line topology

Figure 15: Scheduled instances for large and all topologies

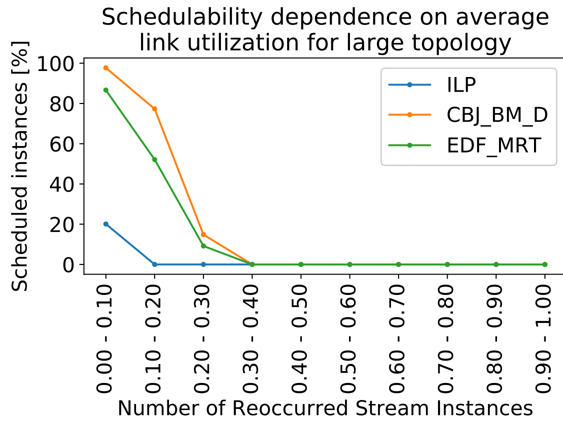
Appendix E Success Rate Based on Link Utilization



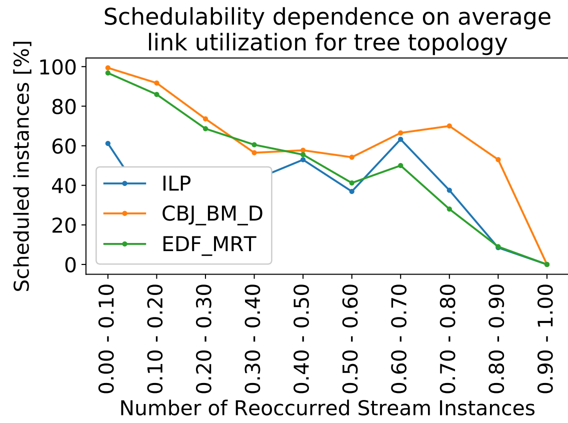
(a) Success rate for avg utilization – small topology



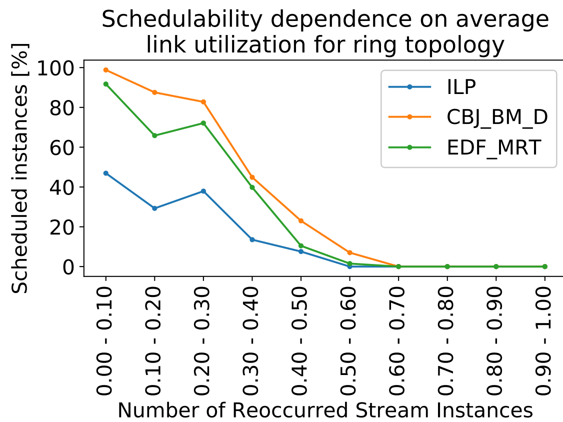
(b) Success rate for avg utilization – medium topology



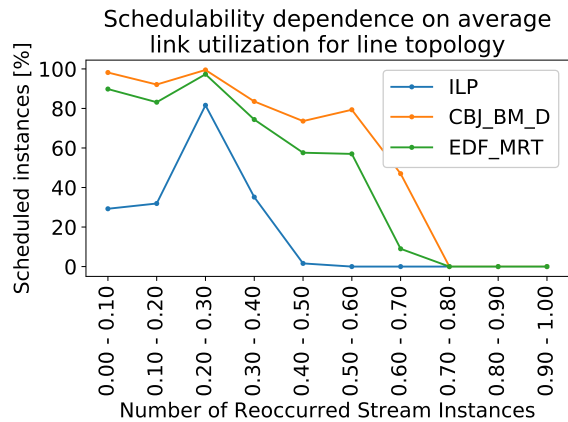
(c) Success rate for avg utilization – large topology



(d) Success rate for avg utilization – tree topology

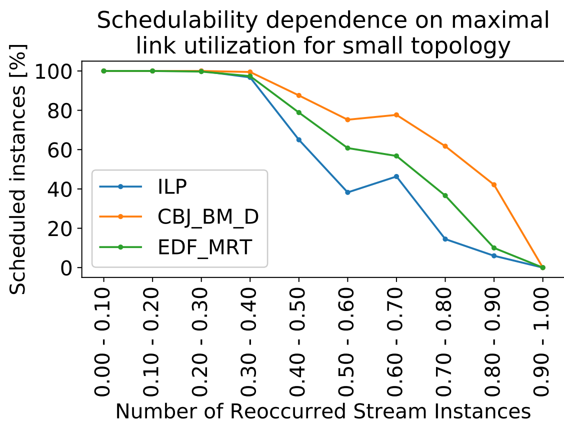


(e) Success rate for avg utilization – ring topology

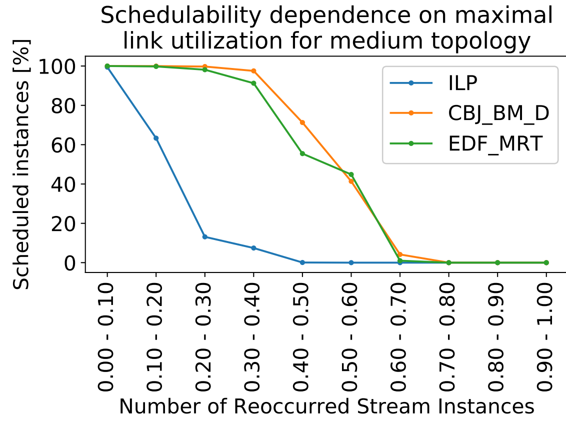


(f) Success rate for avg utilization – line topology

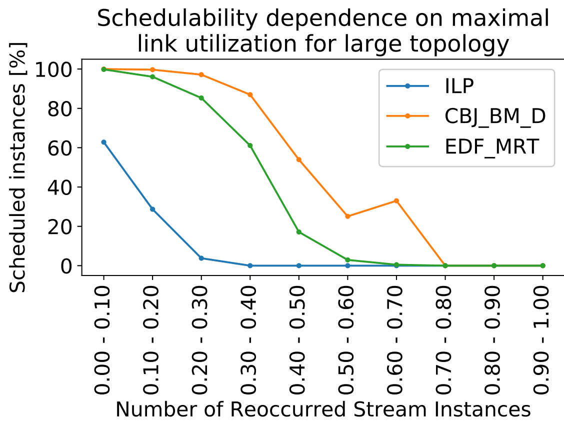
Figure 16: The percentage of scheduled instances based on average link utilization



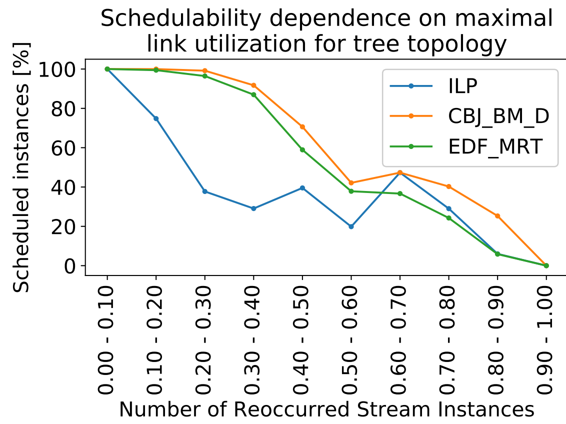
(a) Success rate for max utilization – small topology



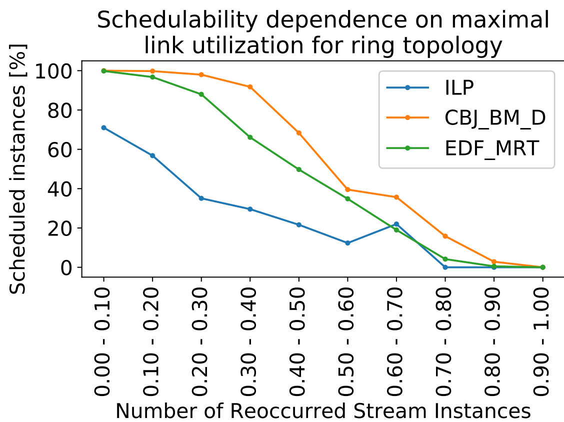
(b) Success rate for max utilization – medium topology



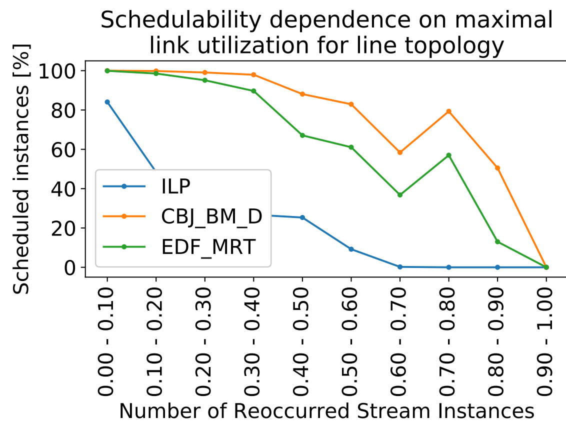
(c) Success rate for max utilization – large topology



(d) Success rate for max utilization – tree topology



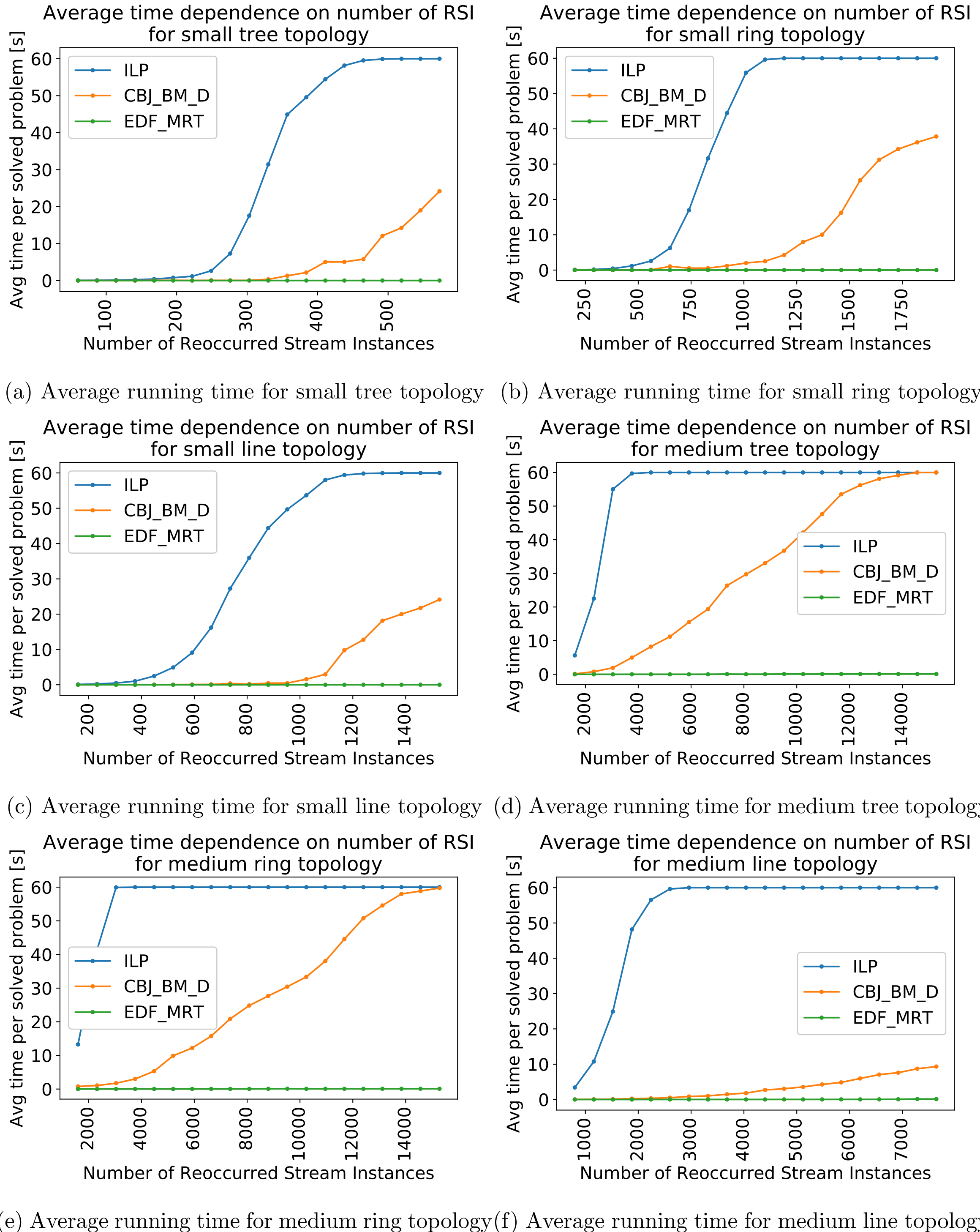
(e) Success rate for max utilization – ring topology



(f) Success rate for max utilization – line topology

Figure 17: The percentage of scheduled instances based on maximal link utilization

Appendix F Average Running Time

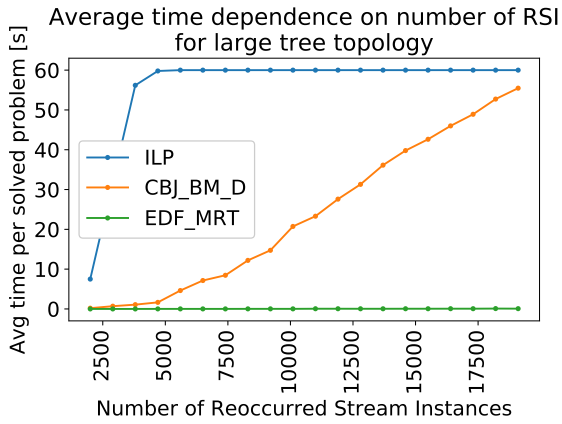


(a) Average running time for small tree topology (b) Average running time for small ring topology

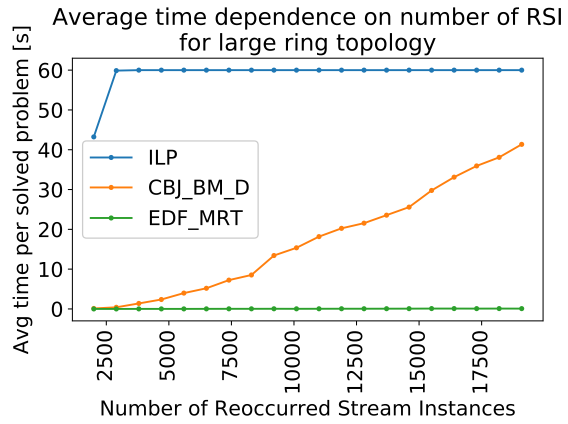
(c) Average running time for small line topology (d) Average running time for medium tree topology

(e) Average running time for medium ring topology (f) Average running time for medium line topology

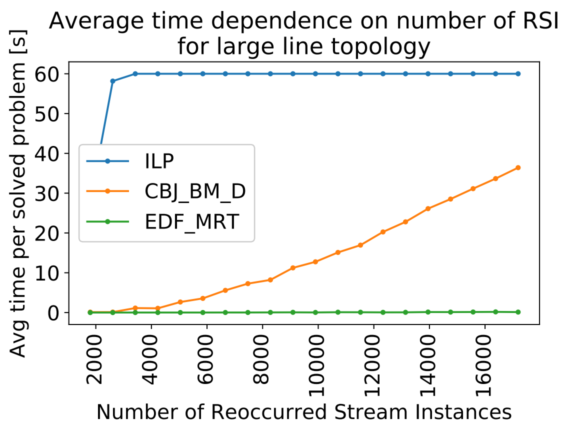
Figure 18: Average running time for small and medium topologies



(a) Average running time for large tree topology



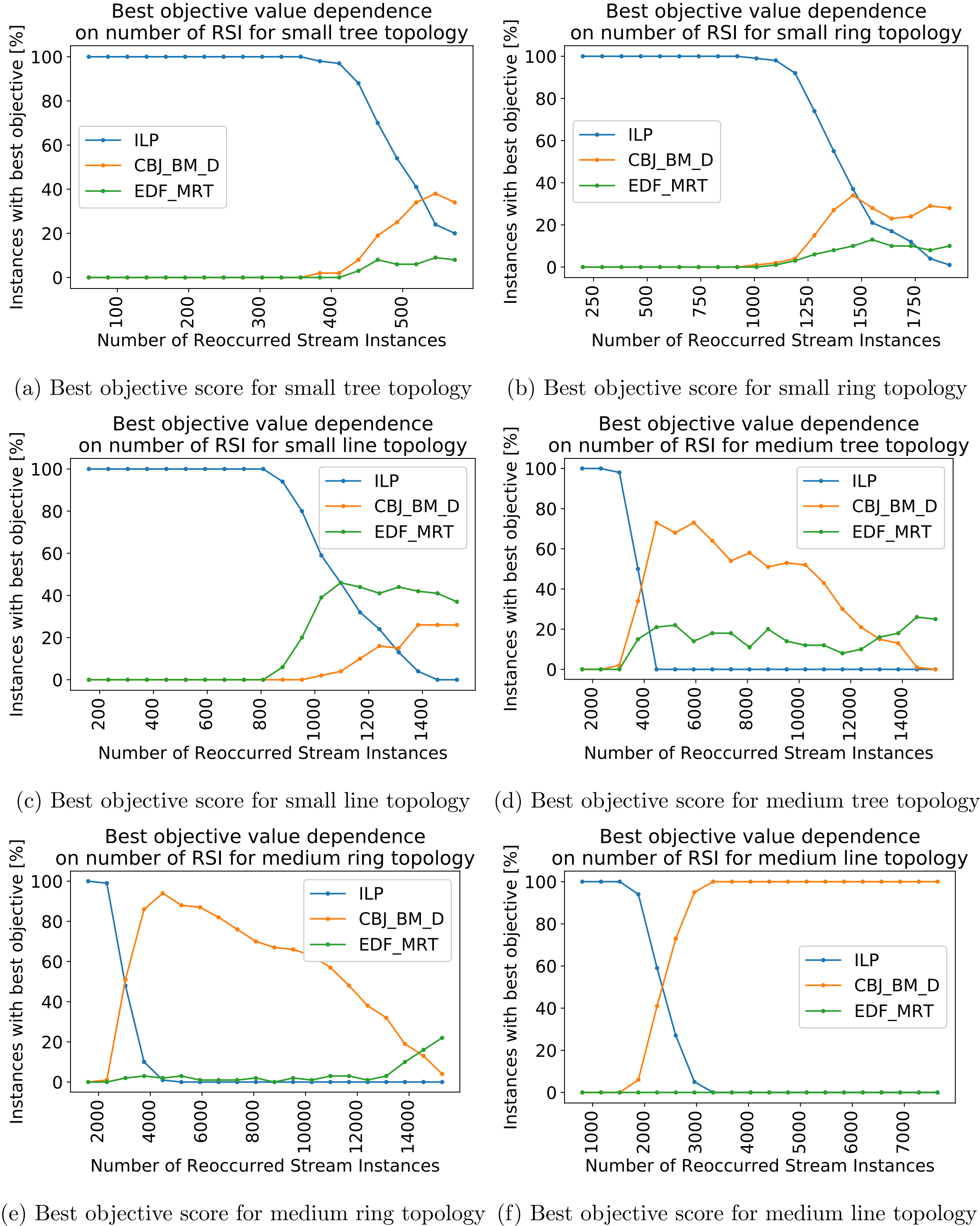
(b) Average running time for large ring topology



(c) Average running time for large line topology

Figure 19: Average running time for large topologies

Appendix G Best Objective Value Score

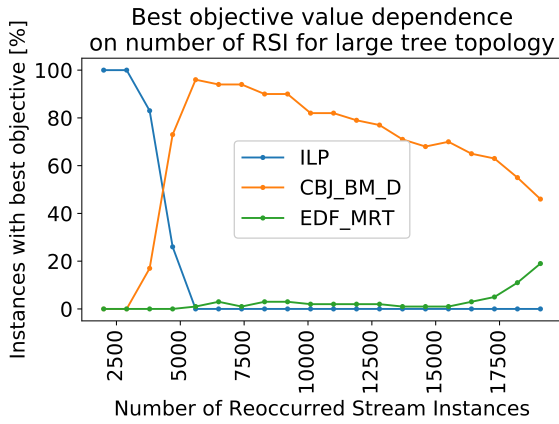


(a) Best objective score for small tree topology (b) Best objective score for small ring topology

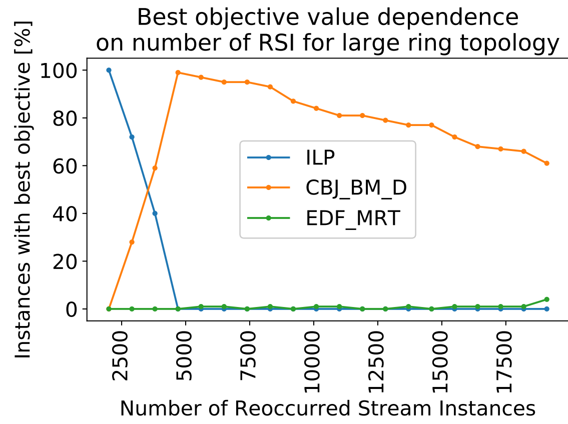
(c) Best objective score for small line topology (d) Best objective score for medium tree topology

(e) Best objective score for medium ring topology (f) Best objective score for medium line topology

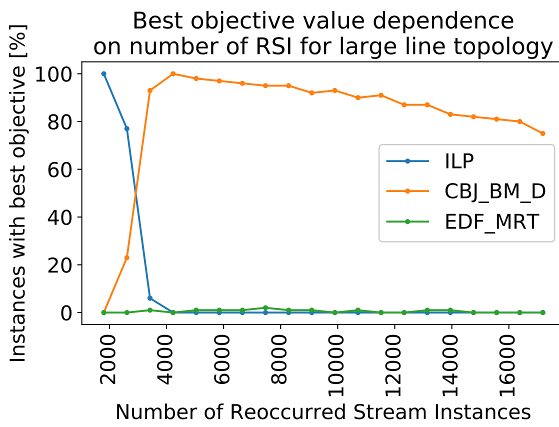
Figure 20: Best objective score for small and middle sized topologies



(a) Best objective score for large tree topology



(b) Best objective score for large ring topology



(c) Best objective score for large line topology

Figure 21: Best objective score for large topologies