Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Weighted Independent Colorful Sets in large Vertex Colored Conflict Graphs for timetriggered Flow Scheduling

Lorenz Rönsch

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Rothermel
Supervisor:	Heiko Geppert, M.Sc.
Commenced:	April 4, 2022
Completed:	October 4, 2022

Abstract

The need for time sensitive communication on networks is increasing more and more, especially due to Industrial internet of things and Industry 4.0. With the appearance of graphics-based network participants in time-critical networks, such as VR glasses, the absolute amount of traffic that needs to be scheduled over the network increases strongly. The most common method to realize real time communication is using the IEEE Time-Sensitive Network (TSN) and the Time-Aware Shaper (TAS). However, the TSN schedule calculation is not standardized. There are several approaches, such as SMT solver, integer linear programming and calculating a conflict graph to calculate time-triggered flow schedules. But none of them are tackling the problem of maximizing traffic. In our work, we extend the time-triggered flow scheduling problem to include the component of maximum traffic. For this purpose, we modify an existing heuristic, called Greedy Flow Heap Heuristic, so that we can adapt the scheduling to our problem. The results that our version provides compared to the original heuristic are very promising. On all our evaluation data, we achieved an average improvement of 81.91% in terms of maximum network traffic. We also developed an alternative non-deterministic approach based on a genetic algorithm. In our work we investigate different variants of the algorithm with the goal to provide better results with different adaptations of the algorithm. In our repair version, we manage to beat our benchmark algorithm the Greedy Flow Heap Heuristic on every circle based conflict graph.

Kurzfassung

Der Bedarf an zeitsensitiver Kommunikation in Netzwerken nimmt immer mehr zu, insbesondere aufgrund von dem industriellen internet der Dinge und Industrie 4.0. Mit dem Auftreten von grafikbasierten Netzwerkteilnehmern in zeitkritischen Netzwerken, wie z.B. VR-Brillen, steigt auch die absolute Menge des Netzwerkverkehr, der über das Netzwerk geplant werden muss, stark an. Die gängigste Methode zur Realisierung von Echtzeitkommunikation ist die Verwendung des IEEE Time-Sensitive Network (TSN) und des Time-Aware Shaper (TAS). Das Problem ist, dass es nicht standardisiert ist, wie die Zeitpläne für TSN zu berechnen sind. Es gibt verschiedene Ansätze wie SMT-Löser, ganzzahlige lineare Programmierung und die Berechnung eines Konfliktgraphen zur Berechnung von zeitgesteuerten Ablaufplänen. Aber keiner von ihnen befasst sich mit dem Problem der Maximierung des Netzwerkverkehr. In unserer Arbeit erweitern wir das Problem der zeitgesteuerten Verkehrsflussplanung um die Komponente des maximalen Netzwerkverkehr. Zu diesem Zweck modifizieren wir eine bestehende Heuristik, die so genannte Greedy Flow Heap Heuristic, zur Maximierung der Netzwerkteilnehmer bei der Verkehrsplanung, so dass wir die Planung an unser Problem anpassen können. Die Ergebnisse, die unsere Variante im Vergleich zur ursprünglichen Heuristik liefert, sind sehr vielversprechend. Bei all unseren Evaluierungsdaten erreichten wir eine durchschnittliche Verbesserung von 81.91% in Bezug auf den maximalen Netzwerkverkehr. Wir haben auch einen nicht-deterministischen Ansatz entwickelt, der auf einem genetischen Algorithmus basiert. In unserer Arbeit untersuchen wir verschiedene Varianten des Algorithmus mit dem Ziel, mit verschiedenen Anpassungen des Algorithmus bessere Ergebnisse zu erzielen. In einigen Varianten gelingt es uns auch, unseren Benchmark-Algorithmus, die Greedy Flow Heap Heuristik, zu schlagen.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The industry is undergoing a revolution in which the focus is no longer just on the computer, but on the internet and digitization. The basis is the so-called Industrial Internet of Things (Industrial IoT). The core concern of IIoT is to make production machines smarter by letting them communicate with each other [Sud17]. Big players in the industry like Airbus, Jeep and Mercedes are already operating IIoT environments to improve the efficiency of their factories. The IIoT can be used in many different areas of the industry, for example in manufacturing companies, logistic companies, agriculture, energy sector or the health care sector.

Airbus has launched its smart factory called "Factory of the Future" where machines are equipped with sensors to provide information about their status and workers are given smart glasses to interact with those machines [Ian17]. This reduces errors and increases safety in the factory which is also reflected in the efficiency [Sud17].

Whenever we create a cyber physical system, which means a system in which information and software technology is connected to mechanical components in the real world using real time communication. It is important that communication is safe, because message loss between those machines can cause devastating accidents, like harming humans. To achieve safe communication between system participants, short and guaranteed transmission times and the avoidance of data loss due overload are crucial. A solution to gain real time data delivery in a network is using the IEEE Time-sensitive Network (TSN). The TSN introduces deterministic communication delay in IEEE 802.3 (Ethernet) networks by extending the Ethernet standard. Therefore, the TSN Task Group published standards, e.g. the Time-Aware Shaper (TAS) to achieve determinism in Ethernet networks. TAS provides a mechanism to schedule traffic over the network with a given end-to-end delay[Hol22]. How to calculate schedules for TSN is not standardized. Intensive research has been carried out in this area for years.

For the scheduling of a traffic plan, different parameter like frequency of transmissions, amount of data per transmission and network topology must be taken into account [FDR20]. Computing a network-wide traffic plan is a well-known NP-hard problem, e.g. related to the Job Shop Scheduling Problem [DN16; FGD+22]. There are several approaches to solve the NP-Hard scheduling problem like integer linear programming [PRCS16], SMT solver [Ste10], constraint programming [VHT21] and building a conflict graph and finding a independent set [FDR20]. The main goal of the approaches is to accommodate the maximal amount of connections of the network into the traffic plan. For example, the Greedy Flow Heap Heuristic [FGD+22] centers on the approach of taking connections who have fewer points of conflict with other connections, rather than those who have a high number of connections in order to increase the maximum number of connections. But there are scenarios where it is not possible to schedule all connections and we have to decide which we should schedule. In this case, the traffic should then be maximized.

Since virtual reality (VR) and augmented reality (AR) glasses have already found their way into the factories, it is important to adapt in this domain as well. Because VR glasses and any other graphics-based device require a large amount of network traffic, it is even more important to achieve the highest possible data throughput while keeping the number of participants high. A high traffic throughput is important for such devices, otherwise there may be delays in the streamed data, which can lead to a loss of efficiency of the application. This was not considered in the previous approaches like GFH by Falk and Geppert [FGD+22]. It should be mentioned that most VR and AR glasses do not work wired but via wireless transmission. However, the consideration of wireless connections is out of scope for this thesis and we focus instead on the backbone networks.

In this thesis we will adapt the GFH heuristic to the traffic property and develop further non-deterministic algorithms for finding the weighted independent set like GFH does on conflict graphs which can be used to solve the scheduling problem. Note that the traffic parameter can theoretically represent any other parameter according to which scheduling is required, e.g. priorities based on domain knowledge. This thesis makes the following contribution:

- a GFH traffic version where we change the sorting function to maximize the total traffic of the network.
- a non-deterministic algorithm with evolutionary traits. We present different versions for this algorithm to improve the resulting solution at the cost of higher runtimes.
- an evaluation of the presented algorithms including a comparison to the existing state-of-the-art algorithms.

In Chapter 2 we introduce our notations and graph theoretical concepts as well as our problem statement. Chapter 3 focuses on networks and the underlying GFH algorithm. Chapter 4 provides the related work that has been done on finding maximum independent sets.
Chapter 5 presents our algorithms, including the GFH traffic sorting approach and different versions of the non-deterministic algorithm. Followed by the empirical evaluation of the approaches in Chapter 6. We compare how the variants perform in comparison to the original GFH algorithm. In Chapter 7 we draw our conclusions followed by possible future work.

2 Preliminaries

In this chapter, we explain the graph theory basics that are necessary to understand the topic. Furthermore we describe the mathematical problem behind the thesis and different kinds of independent sets.

2.1 Graph and Independent Sets

The section covers graph concepts like the basic graph as well as colored and weighted graph concepts. We also address independent sets with rainbow and weighted variants.

2.1.1 Basic graph concept

A graph G = (V,E) is defined by a set V which contains all nodes of the graph and a set of tuples $E \subseteq \{(u, v)|v, u \in V\}$. The tuple $(u, v) \in E$ represent an edge between two nodes u and v. A node u has a set of neighbors N with $N \subseteq \{V|(u, n) \in E, u, n \in V\}$. If a node has no neighbors, we call it a solitary node. In our thesis we assume undirected graphs $(u, v) = (v, u)$ with $u \neq v$. Which can represent a network graph i.e. nodes represent computer or switches and edges a physical or wireless connection between them. Modeling Networks with graphs is discussed in Section 3.1.

2.1.2 Vertex colored graph

To define a colored graph, we add to the basic graph from 2.1.1 an additional color set C. Let G = (V,E,C) be a vertex colored graph with a ink function $I : V \longrightarrow C$ which maps a vertex to its color. Colors of vertices represent different attributes of a vertex. It can be a device type like switch and end system or in our case in the conflict graph an assignment of a possible flow configuration to its flow.

2.1.3 Vertex weighted graph

A vertex weighted graph G = (V,E,W) is a graph where every vertex has an attribute called weight. We define a heft function H which returns the weight w of a vertex v with $H : V \longrightarrow W$. A weight can represent different attributes of a vertex, for example a buffer of memory or possible incoming flows at the same time.

2.1.4 Independent set

An independent set of a graph is a subset of vertices where none of the vertex are adjacent to each other. The maximum independent set represent the set of vertices which is not a subset of another independent set which means it has the maximum possible amount of vertices in it. We call the problem of finding such a set the maximum independent set problem, which is NP-hard in general graphs [HL05]. An independent set is defined as the following formula $IS(V) = \{S \subseteq V | \forall v, u \in S(u, v) \notin E\}$. The maximum independent set is defined as $MIS = argMax(|IS(V)|)$.

2.1.5 Rainbow independent set

For a colored graph G = (V,E,C) and $C = \{0, 1, 2, ..., t - 1\}$ a rainbow independent set (RIS) is a independent set of vertices from G but with the addition that every vertex in the set has an unique color, $RIS(V) = \{S \subseteq V | \forall v, u \in S \notin E \wedge I(v) \neq I(u)\}$. Thus, the rainbow independent set can contain at most t elements. A rainbow independent set is also an independent set, but is not equivalent. A maximum rainbow independent set is the set of vertices which is not a subset of another rainbow independent set which means it has the maximum amount of colors in it. We define the maximum rainbow independent set function as followed: $MRIS = argMax(|RIS(V)|)$.

2.1.6 Weighted independent set

There are cases where not every element of a set has the same prioritization as others. This can be modeled using weights. To do so we attach an *weight* attribute to every element and to the independent set a attribute *total weight*. We call this a Weighted independent set $WIS(V) = \{S \in V | \forall v, u \in S(v, u) \notin E \wedge \forall v \in S, H(v) = w, w \in W\}$. Where H is the heft function. The total weight (TW) is calculated by summing up all weight attributes from the vertices of the set S, $TW(S) = \sum_{v \in S} H(v)$. The maximum weighted independent set is the independent set with the greatest total weight attribute, MWIS = argMax$|WIS(S)|$. The maximum independent set mentioned above is a special case of the maximum weighted independent set where all weights equal to 1.

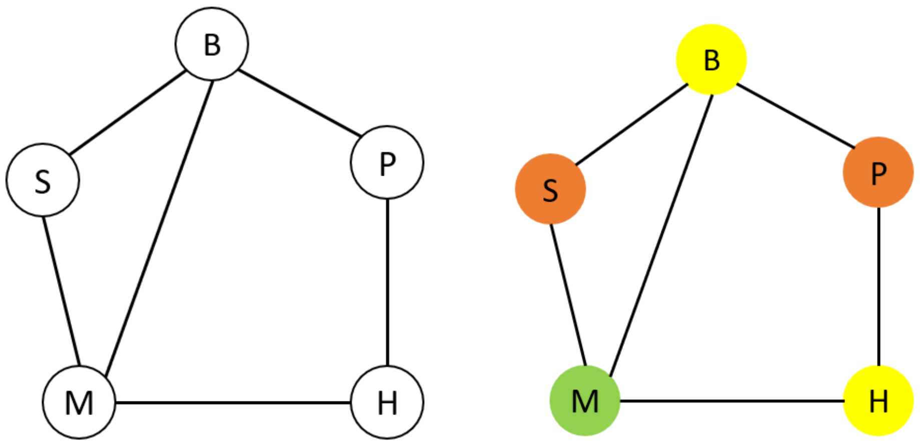
2.1.7 Rainbow weighted independent set

In the following we merge the rainbow property (cf. Section 2.1.5) and the weighted property (cf. Section 2.1.6) together creating a rainbow weighted independent set (RWIS). Let G = (V,E,C,W) be a weighted colored graph. A rainbow weighted independent set is an independent set where every vertex has an unique color, and each set has an attribute total weight (TW). The maximum rainbow weighted independent set is the set of vertices with the greatest total weight attribute while the definition of the rainbow independent set holds true. We define the function $MRWIS = argmax(TW(RWIS(V)))$.

2.1.8 Conflict graph

A conflict graph G is a graph where we can model relationships between different entities [ANS00]. With a conflict graph it is possible to solve different problems, like an scheduling problems where we going to have a closer look on. The idea of the scheduling problem is to allocate resources among a number of time slots, for example timetable scheduling or aircraft scheduling. To model a conflict graph for a problem, we need so called constraints. Which define conflicts between resources. Lets take the timetable scheduling problem as an example. We want to schedule courses of students for a college. Possible constraints could be that we can not schedule courses at the same time slot if one student is registered in both. By building a conflict graph with this constraint we create vertices for every course and draw edges between courses where at least one student is registered parallel. A possible conflict graph could look like Figure 2.1a. Where at least one student is in mathematics (M) and biology (B) registered at the same time. With such a conflict graph we can solve the scheduling problem by coloring the nodes with an graph coloring algorithm, see Figure 2.1b. The number of colors represent the number of minimal time slots we need to schedule every course with out conflicts. This means for our little example we could schedule biology and history at the same time. After this timeslot we can schedule sports and physic. And in the end we can schedule mathematics.

In our thesis we use already colored conflict graphs to solve the scheduling problem for time-sensitive network traffic by finding an maximum weighted independent set in it. In our thesis a conflict graph is based on the colored graph concept (cf. Section 2.1.2) and the weighted vertex graph (cf. Section 2.1.3) and represent a logical relation between flow configurations [FGD+22]. Each flow gets its own color, nodes colored the same represent a flow configuration which represent a possible pathing through the network. Edges represent conflicts between flow configurations.



(a) A conflict graph representing the constraint of at least one student is registered in both courses.

(b) A colored conflict graph, with 3 different colors which means we need 3 independent timeslot so schedule all courses.

Figure 2.1: Two graphs with 5 nodes and 6 edges. Nodes represent courses (**S**port, **B**iology, **P**hysic, **H**istory, **M**athematics)

2.2 Problem statement

In this section we formally define the problem of finding an maximum weighted independent set of a graph.

For a given weighted and colored graph we want calculate the (maximum) rainbow weighted independent set. A problem of this calculation is that we have to make a trade of between the maximum amount of colors in the set and the maximum total weight of the set. In our thesis we focus on maximizing the total weight by holding the properties of the color independent set true. We also want to define a formula to evaluate how well a set solves the problem. A second goal is to keep the runtime until we find a solution as short as feasible, in relation to the quality of the resulting solution, to solve even large-scale problems. To describe our problem formaly we define our optimization function as followed, $OF = argmax_S\{\sum_{v \in S} w(v) | \forall v_1, v_2 \in S \subseteq V : (v_1, v_2) \notin E \wedge S \subseteq V\}$. Where S is our selection set and contains at most a single vertex from each color V_i and w is our weight function. Our problem statement is used to solve the scheduling problem or more specific the flow scheduling problem. We expand on the combination of TSN scheduling and MRWIS in Chapter 3.

3 Network domain

In this section, TSN scheduling of time-triggered flows is discussed as a possible application domain for the RWIS problem. First we describe the underlying system model of the network problem. Then we present a reduction of the network problem to the RWIS problem. In the end we review some solution approaches and techniques.

3.1 Network graph

A network can be modeled as a undirected graph. Thereby, nodes can be defined as end nodes or switches. End nodes feed data packets periodically into the network and also receive them from other nodes. Network switches do not insert traffic into the network, they just forward packets as defined by the traffic plan. We assume that all switches are store-and-forward switches. And working equally fast, so we talk about homogeneous networks. Connections, full-duplex links, between network participant can be defined as undirected edges between the nodes. We only consider wired networks.

3.2 Time-triggered Flow

A Time-triggered Flow, or short a flow, is a directed communication channel from a transmitter node to a receiver node. Active flows feed a single data packet from their transmitter into the network per period. We can say the communication channel is build between two end nodes (transmitter, receiver) and switches to connect them. A flow contains a transmitter and receiver node, a packet size, and an end-to-end deadline that ensure that a packet sent by a transmitter node always arrives within this time limit. A flow can have different configurations e.g. with different routes through the network or different offsets [SCO18]. In this thesis we only consider such time-triggered flows.

3.3 Modeling traffic planning with a conflict graph

To solve the traffic planing for time-triggered traffic in a data network, we have to fulfill some conditions. No collision may occur on the network. Switches are not allowed to temporary safe data packets (zero-queuing). Time sensitive flows deliver there data in time and no packet drops under normal operation conditions.

To model these constraints we can build a conflict graph. Where nodes represent possible configurations of routes through the network for a flow. Each flow is given its own color. The nodes are colored by the corresponding flow. Edges between nodes represent our constrains such as collisions and queuing. Which means when 2 configurations have a conflict, we insert an edge between them. To build such conflict graphs we use the algorithm of N. Holtwerth [Hol22].

By performing an maximum rainbow independent set search on the conflict graph, we are able to solve the traffic planing problem. Because by picking only configurations which have no conflict no constraints of the traffic planing problem are violated. The result of the MRIS search can be a partial solution to the problem. Which means we find a solution which is good but does not contain all configurations. If we find a solution that includes all colors, i.e. takes every flow into account, we solve the original traffic planning problem in total [FDR20]. To add new flows into our existing solution, we need to extend the previous conflict graph and search for a new MRIS.

There are different approaches for handling new flows, see Section 3.4. In our paper we are going to search not only for traffic plans by getting a maximum amount of flows into it, we aim to maximize the data rate going through the network. This can be reached by adding every color a weight, in our case the data rate. Since our generated graph has no weights, we need to extract them from the meta data provided from the algorithm of Holtwerth. To find such a set of vertices, we not only have to search for a MRIS we have to find a MRWIS on the conflict graph.

3.4 Offensive vs. defensive planing

There are different ways to react to new flows during the traffic planing phase. Two common ways are defensive and offensive planing. During a defensive planing phase, we do not allow changing configurations of active flows in the traffic plan in order to add new ones. Which means, if a configuration of an active flow is fixed we schedule and route the new upcoming flows around them. In a offensive planing approach we permit to change fixed configurations of active flows in our traffic plan to gain better utilization of network resources, but always schedule those active flows. Rescheduling active flows can entail a Quality of Service degeneration by introducing jitter which can be bounded [FGD+22]. In this thesis we work with unbounded reconfiguration jitter, since jitter optimization is not the focus of this work.

3.5 Greedy Flow Heap Heuristic algorithm

The Greedy Flow Heap Heuristic algorithm (GFH) is an iterative greedy approach to solve the MRIS problem on colored graphs.

First we take a look on the parameters of the algorithm, the algorithm receives two sets of flows. The first one represents the active flows (ActiveF), they stand for active flow configurations which are already included into the solution. Because the GFH follows an offensive planing approach we are allowed to choose other configurations existing flow configurations in the planing plan. The second parameter represents new flows which have to be included into the traffic plan configuration. During the algorithm a heap is build which is ordered by the flow with the least eligible (not shadowed) configurations on top. Than the root node is picked of the heap and it is searched for the best configuration of the flow based on a configuration rating function. After this the first flow of the heap is removed and the heap is reordered. The next flow is picked until no flow is left in the heap.

With the re-run mechanism the algorithm try to improve the solution, means the number of flows admitted [FGD+22].

4 Related Work

In this chapter we take a look at the research that has already been done in the area of various types of independent sets. Finding an independent set of a graph is a complex task with different possible approaches. To approximate the best solution, we need to trade off computational efficiency in large graphs and the accuracy of the solution.

All independent set problems presented in Chapter 2 such as MRIS and MWIS are not less complex than the MIS. We can reduce the problem of finding a maximum independent set problem to our problem of finding a maximum rainbow weighted independent set. To do this we have to color every node of the conflict graph with another color and the weights are all set to 1. By doing this, we can say our problem of finding an MRWIS is at least NP-hard because the MIS problem is known to be a NP-hard problem and we are able to reduce it [MP18]. There are some graph structures like claw free graphs and P_5 free graphs where we can solve the MIS efficient in polynomial time [HL05]. However, we can not assume our conflict graphs are in such format.

Balaji et al. present an algorithm called Vertex support algorithm (VSA) using vertex cover which solves the maximum independent set on general graphs with a complexity of $O(2^{0.304n})$. The performance is much better than other heuristics found in the literature like SQUEEZE or KLS [Jia86]. The problem is that VSA only solves the independent set problem and does not cover weighted and rainbow sets.

In 2003 Sakai et al. presented three simple greedy algorithms which solve the maximum weighted independent set problem [STY03]. They start with two basic greedy algorithms which solve the MIS problem and extend them to solve the MWIS problem. The first approach is build on the basic algorithm GMIN, an algorithm selecting the vertex of minimum degree and removes the neighbors until the graph is empty. The second approach is build on the basic algorithm GMAX, which selects the vertex with the maximum degree and deletes it from the graph until there is no remaining edges left. In both approaches Sakai et al. change the vertex selecting rule to suite the MWIS Problem. For the third greedy algorithm they change the vertex selection rule of the extended GMIN. The problem of the presented algorithms is that they just cover the weighted side of our problem not the rainbow side.

In the field of Rainbow Independent Sets there is a lack of general purpose solutions to solve it. Manoussakis and Pham proposed a solution for cluster graphs and trees [MP18]. The conflict graphs in this thesis have a different structure. So the solving algorithms can not help us to solve our underlying problem.

Falk et al. [FDR20; FGD+22] published two articles in 2020 and 2022 where they used (M)RIS heuristics to solve traffic planning via conflict graphs. Conflict graphs represent the relation between different flows, where vertices represent a specific configuration of a flow and edges a conflict between two flow configurations. After building those conflict graphs, they use a heuristic called *Greedy Flow Heap Heuristic* (GFH) [FGD+22] to find an independent rainbow set of the conflict graph or in other words a solution for the traffic routing problem.

The GFH is designed to find a solution with as many colors as possible. For our purpose the GFH does not suite in total, because the GFH searches for as many colors as possible it is more likely to pick flows with small traffic or short routes. Since those will cause less conflicts with other flows.

None of the presented algorithms cover the MRWIS problem. In our literature search there was no approach which covered our problem statement in total. To tackle the presented problem statement in our thesis, we take the underlying approach of Falk et al. because it covers a big part of our problem already. We are going to change the GFH to not only maximize the amount of colors but also maximize the traffic caused by the flows.

5 Algorithmic approaches

In this chapter we are presenting the various algorithmic approaches we developed to find an RWIS in a conflict graph. First we introduce the GFH traffic sorting approach which is based on the original GFH outlined in Section 3.5 with a modified flow sorting function. Than we present different genetic algorithm approaches based on a evolution method. In the chapter we introduce variations based on the basic version to boost the quality of the algorithms solution. Finally we introduce a combined approach, merging GFH and the genetic algorithm.

5.1 GFH traffic sorting

We explain in this Section our adjustments to the GFH algorithm to fit more into our problem statement. The original GFH algorithm schedules with a min-heap according to the remaining eligible configurations. Ties are broken with the total degree of the color. This approach does not fit the (M)RWIS problem, because GFH optimizes the number of admitted flows. The admitted traffic obviously correlates with the number of admitted flows. However, in cases of network congestion this changes. This means that the difference in metrics only comes into play if we cannot schedule all flows, because if we accommodate all flows, we will also achieve maximum traffic. Our GFH variation works as the original GFH algorithm, but uses a different heap sorting function. Our heap sorting function first sorts by the traffic of the respective flow. Hence, high-traffic flows are handled earlier and are more likely to be included in the solution. Ties are broken using the original sorting function (cf. Algorithm 5.1).

Algorithm 5.1 Pseudo Code of heap sort function

```
1: procedure HeapSort($flowA$, $flowB$)
2:     if Traffic of both flows are equal then
3:         if Remaining Configs of both flows are equal then
4:             if Total Degree of both flows are equal then
5:                 return flow with smaller Color          // every color is represented as a number
6:             else
7:                 return flow with highest total degree
8:         else
9:             return flow with less Remaining Configs
10:    else
11:        return flow with higher traffic
```

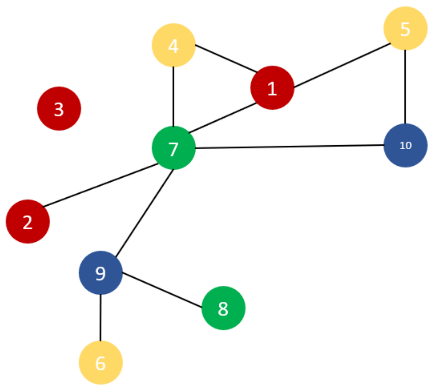


Figure 5.1: Example conflict-graph with 4 colors and 10 nodes where node 3 is solitary.

5.2 Genetic Algorithm Approach

The GFH approach delivers to every input graph the same result because it is a deterministic algorithm. This can be a curse and a blessing at the same time. Constant calculation time is definitely desirable but as we always use the same heuristic, we can never expect better results. In this section we present an approach relying on nondeterminism. Our genetic algorithm approach is an algorithm based on mathematical chance, which can lead to a better result but on the other hand may has an enormous calculation time. Further, better results are not guaranteed. The approach follows a genetic algorithm basic principle where a population of genes is mutated and their fitness value is determined. A larger fitness value represents a better population. For every newly found population, we simply decide, by following the *natural selection* of Charles Darwin also called "Survival of the fittest", whether to use it or return to the previous population. This means that we only use the new population if it is better than the previous one.

5.2.1 Basic Version

In this subsection we introduce our basic version of the genetic algorithm. It follows the basic principle of an genetic algorithm and uses as a fitness value the number of flows admitted to the solution. For our problem the population is an array with the size equal to the number in colors of our conflict graph. We are going to call it from now on the mutation array. Initially, we create a population set for every color. A population set contains all possible configurations of the color and a value for marking a color as not selected in our solution array. In our case it is the value "-1". If a flow has solitary nodes, the population set consists only of those, because solitary nodes enhance our solution without any compromise. Next, we initialize our mutation array, solitary nodes which do not have any conflicts are placed in our array, all other array elements have the value "-1". A mutation array example based on the conflict graph in Figure 5.1 can be seen in Figure 5.2a. Then we start iterating through the array and decide for every index if we mutate its content. The mutation chance is based on a mutation value called α, preliminary evaluations suggested a value of $\alpha = 3\%$ to be promising. With a chance of α we mutate an array index. If our algorithm decides we mutate an array index we choose a random value of the population set for the color. After we iterated

3	-1	-1	-1

(a) The mutation array after the initialization for the example graph Figure 5.1. Fitness = 1

3	-1	7	-1

(b) A possible mutation of the mutation array. Fitness = 2

3	4	7	-1

(c) A invalid mutation of the mutation array. Nodes 4 and 7 are adjacent, resulting in a violation of RWIS properties. Fitness = 0

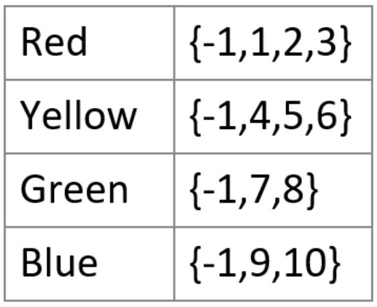
3	4	8	10

(d) A perfect possible final mutation. Fitness = 4

Figure 5.2: Cases of the mutation array which can occur during the mutation loop.

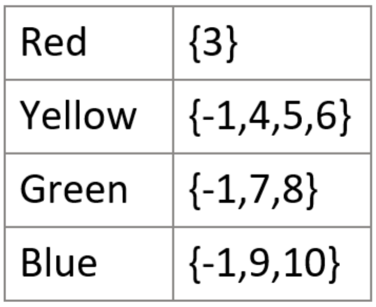
through the array, we have to validate if the array fulfills the RIS constraints. The quality of the mutation array is depicted by a fitness score, which is zero in case the array is invalid, e.g., contains two neighboring vertices. In the basic variant of the genetic algorithm we choose our fitness value as followed: For every valid mutation array we count the entry of the array $\neq -1$. This fitness value represents the amount of colors we can accommodate in our set. The mutation phase is repeated several times, every run is called iteration. The time we need for the algorithm can be bounded by selecting a specific number of iterations. However, a small number can lead to a significantly poorer result.

For our example graph (cf.Figure 5.1) we have 4 colors which leads to a array of size 4. In the following we assume our first array index represent the color red, the second the color yellow, third the color green and the fourth the color blue. Before we start with the mutation loop, we have to initialize the Global set hash map. As we can see in Algorithm 5.2 we need two steps for this, since in line 3 we initialize the population set with the information we get from the adjacent list plus we add the "not picked" value -1. The result can be seen in Figure 5.3a. Afterwards, we trim our population sets with the function in line 4, which searches for solitary nodes. If a population contains a solitary node we have to remove all other configurations of the population set. For our example we have to remove all configurations of red except vertex 3 since it is a solitary node. The result for our example conflict graph is shown in Figure 5.3b. In case we mutate the array index 0 the function can only pick configuration 3 for red. After we are finished with the global set we can start with the mutation loop, iterating through the array and mutating with a very low rate α. Let's assume we iterated through the array and thereby only mutated index 2 (yellow), picking configuration 7 (cf. Figure 5.2b). Our fitness value is now 2, because two array fields are not filled with -1. It is better than the previous fitness value which means we keep the new mutation array. During our mutation phase, it can happen that invalid arrays are created. An example is shown in Figure 5.2c where after the mutation of the array, array index 1 and 2 are in conflict because node 4 and 7 are adjacent in our example graph (cf. Figure 5.1). This means the mutation array violates the properties of a RIS resulting to a fitness value of zero. In this case we discard the new mutation array and go a step back to the previous one. A perfect solution of our example graph is shown in Figure 5.2d, where we can achieve a fitness score of 4 by filling every array field with a configuration without violating the RIS properties.



Red	{-1,1,2,3}
Yellow	{-1,4,5,6}
Green	{-1,7,8}
Blue	{-1,9,10}

(a) Global set after the init_global_sets() function



Red	{3}
Yellow	{-1,4,5,6}
Green	{-1,7,8}
Blue	{-1,9,10}

(b) Global set after the find_solitaires function

Figure 5.3: Global set of the color and the related configurations

Algorithm 5.2 Pseudo Code of the Genetic Algorithm

```
procedure GENETICMAIN($adjList, iterations$)
    int fitness = 0;
    initGlobalSets();
    findSolitarys();          // Finds the solitary nodes in the graph and trims the global set.
    mutationArray = initMutationArray();
    fitness = findFitness(mutationArray);
    for int i = 0; iterations > i; i++ do
        mutationArrayNew = mutateFunction(mutationarray);
        if (checkIfMutationIsLegal(mutationArrayNew)) then
            if (findFitness(mutationArrayNew) > fitness) then
                mutationArray = mutationArrayNew;
                fitness = findFitness(mutationArrayNew);
```

5.2.2 Traffic Version

In the following we discuss a variation of the genetic algorithm. This variation takes the traffic of the flows into account, adapting the algorithm for the RWIS problem.

In the basic variation of the genetic algorithm we calculated the fitness value based on the amount of colors we are able to accommodate in our set, similar to the GFH, which searches for a RIS. This leads us to a problem, as we search for a solution for a RWIS. Hence, we are searching for a set which is color unique and generates the maximum weight. Every color has a weight, in our case it is the traffic of the flows associated with the color. There are cases where its not the best solution to take the maximum possible amount of nodes without conflicts into the solution set because flows with a large traffic can be outdone by smaller flows which generate a smaller traffic together than the single flow. To prevent this scenario, we need to adapt our fitness function. In the genetic algorithm with traffic we do not count the number of picked colors within the mutation array but we iterate over the mutation array and sum up the induced traffic of the picked colors. The traffic of a color can be obtained from the flow's information (cf. Figure 5.4). For our previous example, depicted

Red	5 Kilobytes/s
Yellow	4 Kilobytes/s
Green	9 Kilobytes/s
Blue	11 Kilobytes/s

Figure 5.4: Example traffic table for the example conflict graph.

Red	{3}
Yellow	{-1,4,4,5,5,6,6,6,6}
Green	{-1,7,8,8,8,8}
Blue	{-1,9,10,10}

Figure 5.5: Weighted GlobalSets for example Graph.

in Figure 5.2b, a fitness value of 14 KBps is achieved. This is accomplish by adding up all the traffic values of the accommodated flows, 5 KBps + 9 KBps = 14 KBps. In the optimal case, e.g., Figure 5.2d, we receive a fitness values of 29 KBps.

5.2.3 Weighted Chance Version

In the following we discuss a weighted chance variation of the genetic algorithm to improve the quality of the solution set. In the previous variation of the Genetic Algorithm we choose configurations in the mutation phase without evaluating the impact on the solution set. The result is a high chance of invalid arrays during the mutation phase causing less progress towards finding a better solution. In this approach, we look at how configurations have a negative impact on the solution set. The goal is that configurations which are considered as good, because they have less conflicts with other configurations, are more likely to be chosen during the mutation phase than configurations which harm the solution. We archive this by adding good configurations multiple times into the population set. This means that configurations that have little influence on other configurations are more likely to be included in the solution than those that have a greater influence. To implement this weighted chance we take an already known heuristic at hand, namely the Shadow-Rating from the Greedy Flow Heap Heuristic (cf. Section 3.5). At the start of our algorithm we calculate the shadow rating of every configuration of each flow. Based on the shadow rating we fill our population set shown in Algorithm 5.3. As we can see the sets have changed. For example the configuration 8 of the green flow is added 4 times into the set because of its shadow rating of 1, we get $\lfloor \frac{AmountFlows}{ShadowRating} \rfloor = \lfloor \frac{4}{1} \rfloor = 4$. In our mutation phase it is now more likely to pick configuration 8 for the green flow with an chance of $\frac{4}{6}$ than configuration 7 or not admitting the flow with a probability of $\frac{1}{6}$, respectively.
Solitary nodes are still highly preferred which means we still use the find_solitaires function. For our example in Figure 5.1 we get a weighted global set shown in Figure 5.5.

Algorithm 5.3 Pseudo Code of the weighted GlobalSet initialization

```
1: procedure fillGlobalSet($adjList, iterations$)
2:     int ColorMax = Amount of Flows;
3:     for flow : AllFlows do
4:         for j = 0; j < Math.round(ColorMax/ShadowRating(configuration)); j++ do
5:             flowSet = Node                                  // Add node to flowSet;
6:         avgShadowRating += ShadowRating(configuration);
7:         for i = 0; i < Math.round(ColorMax/ShadowRating(configuration)); i++ do
8:             flowSet = -1                 // Adds the option of non selection of a flow
9:         globalSet = (flow, flowSet)
```

5.2.4 Delete Version

In the previous variation we tried to improve the selection of configurations during the mutation phase. However, the problem of conflicts within the array causing an invalid array still persists. The goal of this variant is that we never have to discard an array because it is invalid. During our mutation phase we choose random configurations of the global sets and add them to the solution array. Then we check if the array is valid or not. At this point we want to intervene. Instead of dropping a invalid array, we will delete conflicts of the array, by not admitting the conflicting flow, to make it valid. For example in Figure 5.2c our mutation phase produced an array with a conflict. Now we iterate over the array and delete conflicts until we get a valid array. In the example we would delete the Configuration 4 by setting the value of the array index 1 to -1 which leads to the array shown in Figure 5.2b. After this process, we get a valid array and can proceed with the fitness validation.

5.2.5 Repair Version

The repair version follows a similar idea as the delete variation (cf. Section 5.2.4). But instead of deleting conflicts of the array we try to solve them. If there is no possible way to solve the conflict we fall back to deleting configurations.

The reason we try to repair the array instead of simply deleting the conflict is that deleting a flow from the solution lowers the potential fitness value by the deleted flow. When we resolve a conflict using our repair variant, we do not reduce the fitness value. The biggest problem with this variant is how we repair the array. If there is a simple and efficient way to choose the correct configuration to repair a array, the problem of finding RWIS would not be NP-hard. For our repair variation we choose a simple but inefficient way to find a solution - the brute force method. After we have identified a conflict, we try to solve the conflict-causing array indexes by choosing another configuration of the population set. If this does not resolve our conflict, we try the next configuration until there is no one left, resulting in our fall back (cf. Algorithm 5.4). A solving configuration is defined by solving the resulting conflict and not causing a new conflict with other flows. This method is performed for every conflict in the array. For large conflict graphs with large population sets this method can have drastic performance issues.

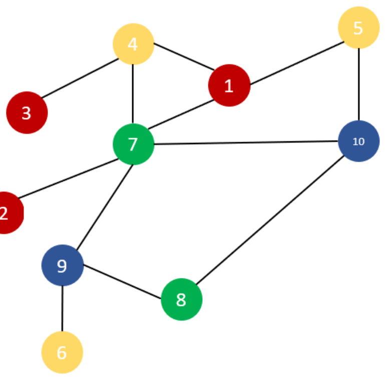


Figure 5.6: Second example conflict-graph with 4 colors and 10 nodes.

Algorithm 5.4 Pseudo Code of the repair function

```
1: procedure REPAIRARRAY($indexOfConflict, mutationArray$)
2:     solvingCandidates = globalSet(indexOfConflict); // Returns the List of Configs. of the flow
3:     for entry : solvingCandidates do
4:         if entry != -1 then                    // Not selecting the flow is not a option
5:             if entry is solving Conflict then
6:                 mutationArray[indexOfConflict] = entry;
7:                 return;
8:     mutationArray[indexOfConflict] = -1          // no solution found -> delete conflict
```

For our example in Figure 5.7a based on the conflict graph depicted Figure 5.6 we first detect the conflicts. In our example we have two conflicts. Then we start solving the first one by testing the possible configurations, we swap randomly configuration 3 with 1. It is causing an invalid repair attempt and we try the next configuration of the population set. We insert coincidentally configuration 2, shown in Section 5.2.5 which leads us to a valid repair attempt. The second conflict is between array index 2 and 3. By swapping randomly configuration 8 with 7, we generate an invalid repair attempt. As there is no other configuration left to test, we have to delete the flow by inserting -1.

3	4	8	10

(a) Example array after a mutation phase with two conflicts. Index 0 is in conflict with 1 and index 3 with 4.

1	4	8	10

(b) Invalid repair attempt. Violation of a solving configuration by producing a new conflict with configuration 4.

2	4	8	10

(c) Valid repair attempt, configuration 2 is solving the conflict.

2	4	-1	10

(d) Valid array with out a conflict by deleting the third flow.

Figure 5.7: Repair function example.

6 Evaluation

In this chapter we discuss our evaluation methodology and results. First we give a brief overview about our system environment and used input data. Further, we will compare our developed algorithms in relation to the scenarios and compare our results with the GFH algorithm.

6.1 Environment and conflict graphs

Our algorithms are implemented with Java version 14.0.1 and build with gradle. Our evaluations are performed on a server machine with Ubuntu 20.04.4, equipped with two AMD EPYC 7413 24-Core processors. In total our server has 96 threads with a cache size of 512KB each and a total of 256 GB RAM. In our implementation we do not use parallel programming, which leads to a single-threaded program execution.

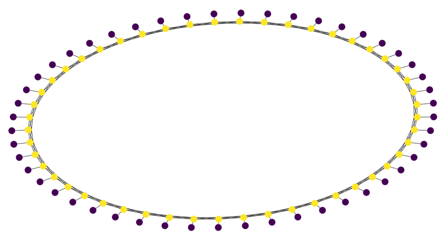
For our purpose of finding a MRWIS mentioned in Section 2.2 we need graphs as input data. For our evaluation we use conflict graphs which represent TSN scheduling problems (cf. Section 3.3). Those conflict graphs are generated by a program based on the master thesis of N. Holtwerth [Hol22]. The program uses network graphs and use case scenarios as input data to generate a conflict graph. Our scenarios just contain a single time step which means we do not add and delete flows from the network in several passes. To generate our 15 Conflict Graphs for our evaluation, we use 5 Network Graphs (cf. Figure 6.1 and Table 6.1). One circle network graph, one mesh network graph and 3 random network graphs. We decided to use 3 different random graphs with the same number of switches and end devices to detect and interpret anomalies in our evaluation. We also needed 3 scenarios with 500, 600 and 700 flows for each network. From this data we then generated our Conflict graphs(cf. Table 6.2).

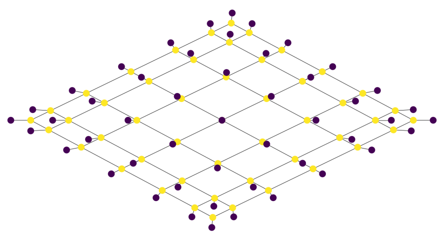
6.2 Evaluation of GFH approaches

In the following section we will evaluate the GFH algorithm and the GFH variant we developed. We will look at how far the results of the two algorithms differ in terms of traffic and admitted flows.

topology	network end devices	network switches	network edges
circle	49	49	147
mesh	49	49	133
random	49	49	random

Table 6.1: Overview of networks used for the evaluation.


(a) Triple Linked Ring Network


(b) Mesh Network

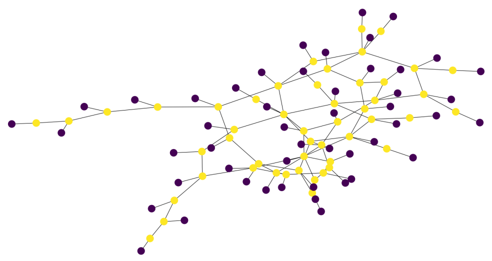

(c) Random Network

Figure 6.1: Networks used for evaluation. Yellow nodes represent switches and blue nodes end devices.

network	number of initial flows	resulting conflict graph
circle	500	circle500
	600	circle600
	700	circle700
mesh	500	mesh500
	600	mesh600
	700	mesh700
random1	500	random500_1
	600	random600_1
	700	random700_1
random2	500	random500_2
	600	random600_2
	700	random700_2
random3	500	random500_3
	600	random600_3
	700	random700_3

Table 6.2: Overview of the scenario parameters and resulting conflict graphs used as evaluation data.

conflict graph	traffic (KBps)	calculation time (ms)	admitted flows	admitted flows in percent
circle500	416.75	3842	379	75.80%
circle600	370.1875	9155	406	67.67%
circle700	443.375	17706	460	65.71%
mesh500	512.9375	24362	428	85.60%
mesh600	371.9375	32844	424	70.67%
mesh700	464.6875	47503	449	64.14%
random500_1	375.625	77705	374	74.80%
random500_2	345.75	150446	356	71.20%
random500_3	407.1875	232016	411	82.20%
random600_1	311.9375	79847	359	59.83%
random600_2	200.125	147187	286	47.67%
random600_3	344.8125	220004	398	64.83%
random700_1	321.9375	113425	411	58.71%
random700_2	222.75	203900	336	48.00%
random700_3	349.1875	237298	400	57.14%

Table 6.3: Results of the GFH Algorithm for all 15 Conflict graphs.

6.2.1 Benchmark Algorithm

We have chosen the GFH algorithm (cf. Section 3.5) by Falk and Geppert [FGD+22] as our benchmark algorithm. In previous evaluations it was able to deliver good and consistent results in a short time. The results of the GFH algorithm for our 15 conflicts graph is shown in Table 6.3. As we can see, GFH does not manage to accommodate all flows for any of the 15 conflict graphs. For the conflict graphs with 500 flows, GFH was able to admit between 71.20% – 82.20% of the 500 flows. The range for the 600 flows conflict graphs is between 47.67% and 70.67%. GFH was able to admit 48.00% – 65.71% of the flows for our conflict graphs with 700 flows. We can see a significant drop in admitted flows as more flows have to be accommodated. This can be explained by the fact that an increasing number of flows poses a greater challenge for scheduling. It is also interesting that the time we need for the calculation is not only dependent on the number of flows to be handled but also on the underlying network topology. We can observe a higher computation time for the random topologies compared to the circle and mesh topologies.

6.2.2 GFH Traffic sorting

We evaluate in this subsection the results of the GFH traffic sorting approach (cf. Section 5.1. The idea of adapting the GFH heap to a traffic sorting heap was to schedule flows with a higher traffic output first in order to receive the maximum amount of traffic in the network.

conflict graph	traffic (KBps)	calculation time (ms)	admitted flows	traffic compared to original GFH (percentage difference)
circle500	637.375	2878	378	52.93%
circle600	677.3125	7380	403	82.96%
circle700	780.6875	14533	434	76.08%
mesh500	603.8125	20921	406	17.72%
mesh600	618.4375	26022	416	66.27%
mesh700	798.9375	36382	438	71.93%
random500_1	566.0625	53424	368	50.70%
random500_2	478.375	113606	323	38.36%
random500_3	632.0	165047	382	55.21%
random600_1	602.625	63284	319	93.19%
random600_2	516.75	127302	259	158.21%
random600_3	605.875	132277	368	75.71%
random700_1	714.9375	72449	377	122.07%
random700_2	585.1875	156771	266	162.71%
random700_3	714.625	155123	370	104.65%

Table 6.4: Results of the GFH traffic sorting for all 15 Conflict graphs and the percentage difference compared to original GFH (cf. Section 3.5)

.

As we can see in our results, our idea is paying off. We outperform the original GFH version in every conflict graph in terms of traffic on the network. Especially in the randomized network topologies, we achieve improvements of up to 162.07 %. In the average of all conflict graphs, we achieve a higher traffic level of 81.91%.

The case we mentioned in Section 5.1, that it can happen to generate a higher traffic even if we accommodate less flows, we can now see well in our results. For example admitted GFH for the *random700_2* conflict graph 336 flows with a total of 222.75 KBps as traffic (cf. Table 6.3). The GFH traffic sorting approach admitted for the *random700_2* conflict graph only 266 flows but generated with 70 flows less, compared to the original GFH, a traffic of 162.71 KBps. This is an improvement of 162.71%. In general, however, we also see that we admit less flows compared to the GFH. This can be explained by our new sorting function in the heap. Where we no longer have the maximum number of flows in the solution as the first priority, but the highest traffic generated.

6.3 Evaluation of the genetic algorithms

In the following section we have a look on our developed genetic algorithms and do not just compare them to each other further more we compare them with GFH and GFH_Traffic as well. First, we start by evaluating our mutation parameter α to investigate which value seems promising. Followed by the evaluation of our traffic version of the genetic algorithm. Further we investigate our weighted random version. Finally we evaluate the delete version and the repair version.

6.3.1 Mutation parameter

In this section we investigate the impact of the mutation parameter. The Genetic algorithms use a array that represents the current solution. The mutation parameter indicates how high the probability is that we mutate our array element to another value. The mutation parameter can be a value between 0.0 and 1.0. Where 0.0 means we never mutate and 1.0 means we always mutate. Our hypothesis was that rare mutations cause less conflicts but lower our chance to find a good solution. To evaluate this we run our genetic traffic implementation on all conflict graphs with a variable mutation parameter. For every mutation value between 0 and 1 with a step size of 0.01 we take 1000 samples with 10000 mutation iterations for each sample. We represent the average of the highest fitness values of each sample to smooth out peaks. For our use case, peaks are not that considerable because we want to achieve a certain consistency in our solution.

We take Figure 6.2a as an example, on the x axis of the diagram we can see the mutation parameter α and the average fitness value on the y-axes after 10 000 Iteration of all 1000 samples. As we can see in Figure 6.2 for every shown conflict graph we only generate a fitness value unequal zero between an α of 0.0 and 0.15. This suggests that a low mutation rate will give us better results. If we look at figure 3, we see the diagram of circle500 plotted on the important range of values. This supports our hypothesis that a low mutation rate is leading us to less conflicts but on the other side it do not seems to be harming our chance to calculate a good solution. The reason that a high mutation rate is leading to those results is, that if we mutate many array indices there is a higher chance that we run into conflicts of configurations. The genetic traffic algorithm discards such arrays and goes back to the old array, what harm our progress in term of finding a good solution drastically.

For the evaluation of the mutation parameter, we consider not only the fitness value but also the time required. As we can see in Figure 6.3, all graphs exhibit similar behavior. We find that a low mutation rate increases the running time of the algorithm compared to a high mutation rate. This can be traced back to the validation check where we check if our new array is filled with a valid combination of configurations. With a higher mutation rate, we detect invalid arrays earlier and more often during this check, which saves some runtime. For the scheduling problem via conflict graphs the runtime is not that impotent because we need the most amount of time creating a conflict graph. Also our primary goal is the solution quality and the runtime is just a secondary objective. With this knowledge, we chose an alpha of 1% for our research.

6.3.2 Evaluation of the Traffic Version

Now that we have selected our α parameter, we will evaluate our genetic traffic version. Here for we take the best alpha value, i.e. 1%, which we found in Section 6.3.1. As for the two previously evaluated algorithms, we again look at how our algorithm version performs on our 15 conflict graphs. However, since this is a non-deterministic algorithm, we can no longer just let the algorithm compute once and look at its results. To generate a representative sample of results, we ran the algorithm 1000 times. Each run with 10000 iterations, i.e. the mutation array went through the mutation phase 10000 times.

As we can see in Figure 6.6 we have a very distributed point cloud for every conflict graph. The diagram relates the fitness value to the required iterations. We can still see some structure in the fixed topologies (cf. Figure 6.6a - Figure 6.6f). But this falls away completely with the randomized

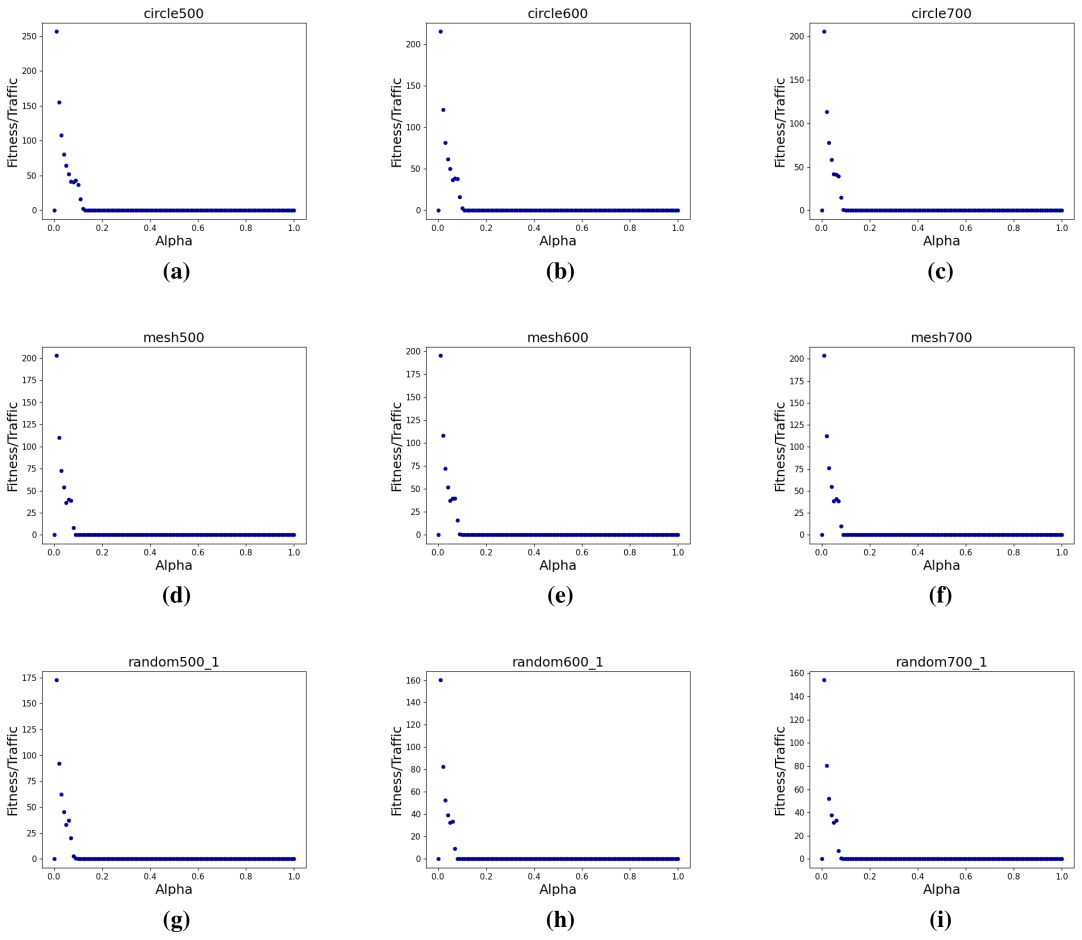


Figure 6.2: Evaluation data of 9 Conflict graphs for the mutation parameter performing the genetic traffic version (cf. Section 5.2.2). The graphs show the fitness value achieved when running the algorithm with a given α.

topologies(cf. Figure 6.6g - Figure 6.6o). The distributed point clouds are a high indicator of a lot of randomness. If we have a look on Table 6.5 we can see the generated traffic of the algorithm. The min traffic column represents the run of the algorithm with the the smallest produced traffic. The max traffic is the traffic value a run was able to generate in 10000 iterations. As we can see, the algorithm does not even come close to our benchmark algorithm in terms of traffic. This can be attributed to the fact that we are very much based on chance. In addition, mutation arrays that do not contain a valid configuration are simply deleted. This leads to poor progress in the mutation phase and explains our low results. This behavier can be seen in Figure 6.5a were the point cloud is represented as a staircase function. Here we can already see that certain runs of the algorithm do not find improvements until the end and remain unchanged over several iterations. This becomes clearer in Figure 6.5b, where we only plotted 10 runs of our results set. For the lowest brown function, for example, we have a fitness improvement at 4145 iterations from 66.1875 to 73.375. Followed by another one at 6127 iterations by 2.125 to 75.5 Kbps. Followed by almost 4000 iterations in which no better fitness value could be found. These many iterations, which do not improve our results but cost us time, should be avoided at all costs.

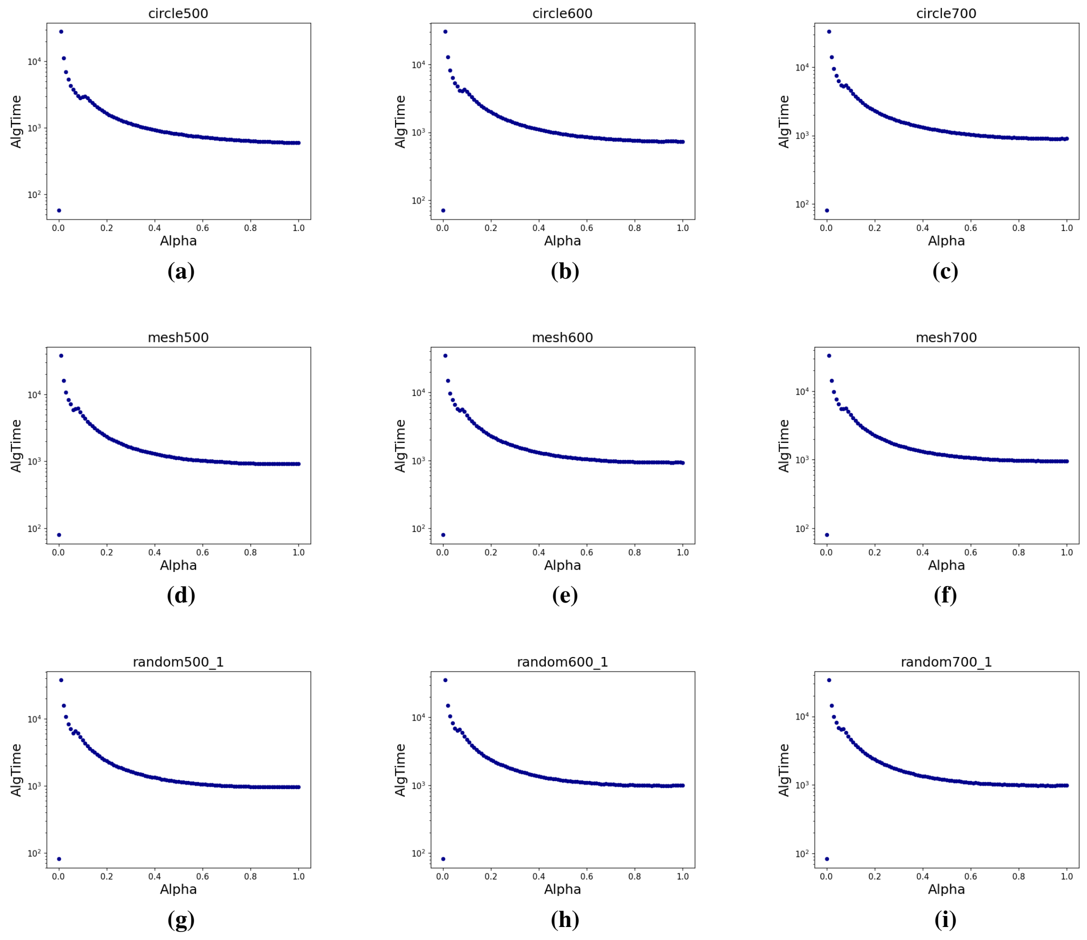


Figure 6.3: Evaluation data of 9 Conflict graphs for the mutation parameter performing the genetic traffic version (cf. Section 5.2.2). The diagrams show the time required to execute the algorithm with a given α.

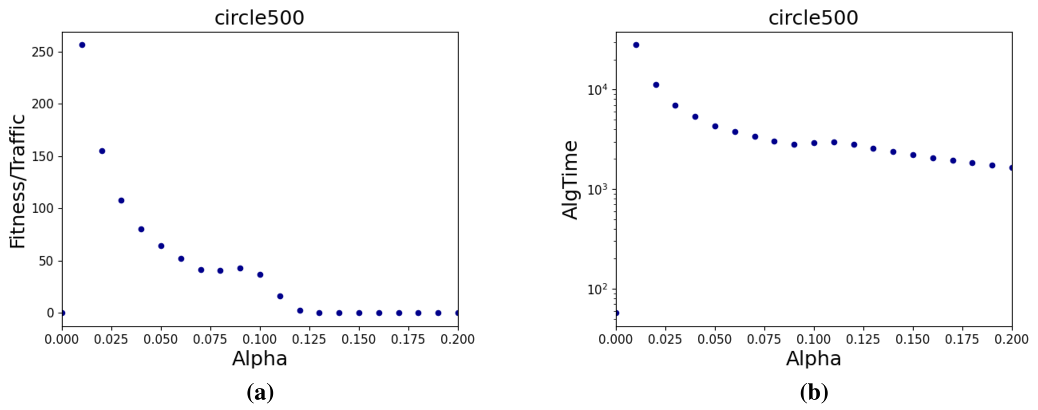


Figure 6.4: Evaluation data of the conflict graph circle500 in the range of 0.0 to 0.2.

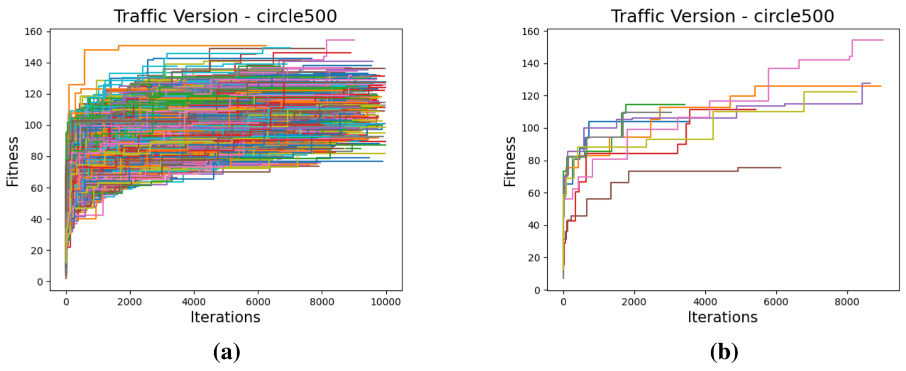


Figure 6.5: Visualization of the results as a staircase function for the conflict graph circle500.

conflict graph	**min traffic (KBps)**	**max traffic (KBps)**	**mean traffic (KBps)**
circle500	66.375	154.1875	107.26
circle600	49.875	137.75	82.31
circle700	39.3125	127.25	76.09
mesh500	31.0625	133.75	72.48
mesh600	34.5	112.25	71.29
mesh700	36.3125	111.8125	75.15
random500_1	28.875	105.4375	60.72
random500_2	14.5	82.0	38.50
random500_3	13.5625	87.0625	42.38
random600_1	21.625	105.5	53.25
random600_2	10.5	66.75	32.65
random600_3	19.8125	82.4375	45.55
random700_1	23.25	92.9375	53.30
random700_2	6.25	65.125	30.58
random700_3	22.6875	95.125	54.07

Table 6.5: Results of the Genetic traffic version for all 15 Conflict graphs

.

6.3.3 Evaluation of the Weighted Chance Version

In the previous evaluation, we found that a purely random-based algorithm yielded poor results. In the weighted variant, we therefore introduced a weighted random. This ensures that configurations that could negatively influence the solution are picked less often than those that do so less often. If we take look at our diagrams for the generated results of the weighted version algorithm (cf. Figure 6.7), we can see that we still have point clouds with a large distribution. Only a closer comparison with the results of the traffic version (cf. Figure 6.6) reveals that we now have fewer outliers up and down from the overall structure. To verify this assumption, we took a closer look at the final results of all iterations of both versions and presented the percentage difference in Table 6.6.

conflict graph	min traffic (KBps)	max traffic (KBps)	mean traffic (KBps)	min traffic compared to traffic version	max traffic compared to traffic version
circle500	69.875	166.9375	112.32	5.27%	8.27%
circle600	44.375	152.5	86.98	-11.03%	10.71%
circle700	43.75	125.4375	80.24	11.29%	-1.42%
mesh500	32.9375	125.8125	76.08	6.19%	-5.93%
mesh600	38.375	117.1875	74.78	11.23%	4.40%
mesh700	42.875	132.5	78.65	18.07%	18.50%
random500_1	34.4375	103.375	64.161	19.26%	-1,96%
random500_2	16.0625	75.5625	40.86	10.78%	-7.85%
random500_3	20.75	93.5	45.64	53.00%	7.39%
random600_1	24.875	101.5625	57.65	15,03%	-3.73%
random600_2	10.3125	70.625	34.41	-1.79%	5.81%
random600_3	22.3125	94.5625	49.39	12.62%	14.71%
random700_1	30.6875	101.25	59.04	31,99%	8.94%
random700_2	8.5	65.3125	33.18	36.00%	0.29%
random700_3	24.75	102.6875	58.94	9.09%	7.95%

Table 6.6: Results of the weighted version for all 15 Conflict graphs and the percentage difference compared to the results of the traffic version (cf. Table 6.5)

.

We note a significant improvement in the lower threshold. In concrete terms, this means that our weighted version achieved a better minimum value than the traffic version in 13 out of 15 conflict graphs. Only in two cases did the minimum value decrease by -1.79% and -11.03%.

Even if it is better to get a smaller range of values for our problem, it is still our goal to improve the maximum value. And here our algorithm could only show slight to no improvement. I.e., we were able to achieve an improvement between 0.29% and 18.5% in 10 cases. However, we also encountered a deterioration in 5 cases from 1.42% to 7.85%.

If we compare our achieved traffic values with those of our benchmark algorithm, we have to conclude that GFH continues to outperform us strongly. For each conflict graph, GFH generates a solution that is 129.5Kbps to 387.125 KBps better.

6.3.4 Evaluation of the Delete Version

The delete version of the genetic algorithm tries to tackle the problem where we cannot find a better solution over many iterations. The goal of the algorithm was to create less invalid mutation arrays in order to have more possibilities during the iterations to expand the traffic value (cf. Section 5.2.4). If we look at our result diagrams in which we show the fitness value in relation to the respective iteration (cf. Figure 6.8), we can directly see a change compared to the previous evaluations. First, we have less outliers and therefore our results are like in a tube. Also, we can observe a significant increase of the fitness value in the first 1000 iterations followed by a more continuous

conflict graph	min traffic (KBps)	max traffic (KBps)	mean traffic (KBps)	time for max traffic calculations (ms)	max traffic compared to GFH
circle500	343.125	399.875	373.99	223233	-4.05%
circle600	335.25	392.3125	361.85	319334	5.98%
circle700	352.8125	413.1875	386.145	460516	-6.81%
mesh500	304.4375	352.25	330.84	222992	-31.33%
mesh600	307.9375	355.5	333.25	321410	-4.42%
mesh700	348.5	417.5	377.41	461142	-10.15%
random500_1	237.8125	307.25	272.97	229244	-18.20%
random500_2	191.3125	243.125	217.99	284818	-29.68%
random500_3	278.5625	335.0	309.38	246421	-17,73%
random600_1	255.5	320.375	285.10	335817	2.70%
random600_2	200.8125	265.8125	230.30	424275	32.82%
random600_3	252.0625	314.6875	288.38	344894	-8.74%
random700_1	285.4375	353.625	318.29	497972	9.84%
random700_2	209.8125	277.25	247.92	640650	-5.81%
random700_3	283.0	336.75	311.25	517189	-18.95%

Table 6.7: Results of the delete version for all 15 Conflict graphs and the percentage difference compared to GFH (cf. Table 6.3)

.

growth. The red horizontal line in the graphs represents the traffic generated by GFH. As we can see, the blue point cloud crosses this line in conflict graph circle600 (cf. Figure 6.8b), random700_1 (cf.Figure 6.8i), random600_2 (cf. Figure 6.8k) and random700_2 (cf. Figure 6.8l). This means that fitness values greater than those of GFH were found.

If we take a look at our results in Table 6.7, we can see that in the cases where our maximum traffic is greater than GFH's, we can see a traffic improvement of 2.7% up to 32.82%. But in the 11 cases where GFH is better, we have to note a traffic loss of 4.05% to 31.33%.

We also achieved our goal of handle no improvement in fitness towards the end of the mutation phase. This can be seen very nicely in Figure 6.5b where all 10 functions generating progress still the end.

Comparing the time required by our benchmark algorithm to our algorithm for calculating the respective maximum traffic value, we unfortunately have to conclude that the delete version takes much longer. In fact, we need between 264449 ms and 341201ms, which is about 4.4 to 5.68 minutes, more time than our benchmark algorithm. Also, we must note that here we are only talking about the maximum traffic that occurred during 1000 executions of the algorithm. This means that it also happens that we do not manage to beat GFH for these Conflict graphs in certain executions. Only for conflict graph random600_2 we could achieve this for each execution.

conflict graph	**max traffic (KBps)**	**time for max traffic calculations (ms)**	**max traffic compared to delte version**	**max traffic compared to GFH**
circle500	417.0625	1452378	4.5%	0.24%
circle600	411.8125	2596536	4.8%	11.08%
circle700	448.0625	4069858	8.47%	1.13%
mesh500	378.3125	1705836	7.39%	-26.17%
mesh600	382.75	2595213	7.60%	2.96%
mesh700	428.9375	4398980	2.64%	-4.07%
random500_1	315.1875	2363659	2.61%	-16.00%
random600_1	333.5	4287643	4.06%	7.07%
random700_1	365.5	6028597	3.40%	13.70%

Table 6.8: Results of the repair version for 9 Conflict graphs and the percentage difference compared to delete version (cf. Table 6.7) and GFH (cf. Table 6.3)

.

6.3.5 Evaluation of the Repair Version

In the following we will evaluate our approach from Section 5.2.5 which does not only delete the conflicts but tries to solve them in order to achieve a greater traffic improvement. Our results for the repair version can be seen in Figure 6.10. As we can see, we only used 9 conflict graphs for this evaluation instead of our 15. This is due to the significantly longer runtime of the algorithm and a limited time for the evaluation. In our diagrams , we can see that for certain runs we outperform both GFH (red line) and our Delete version (yellow line). Only in diagram Figure 6.10d, Figure 6.10f and Figure 6.10g can we see that we do not beat our benchmark algorithm (GFH). In our table with absolute values, we can confirm this. We outperformed GFH in 6 out of 9 Conflict graphs. Furthermore, we perform better than our previous approach in all 9 conflict graphs. The better result, however, affects the total computing time. This is due to our Bruce force approach. This means that for example for *circle_500* we need about 550% more time to get the maximum result. In concrete terms, we are talking about about 20 minutes more calculation time for the for the *circle_500* conflict graph. This enormous difference in computing time makes our small improvement look a bit less good. Especially since we perform even more drastically in comparison with the calculation time of GFH.

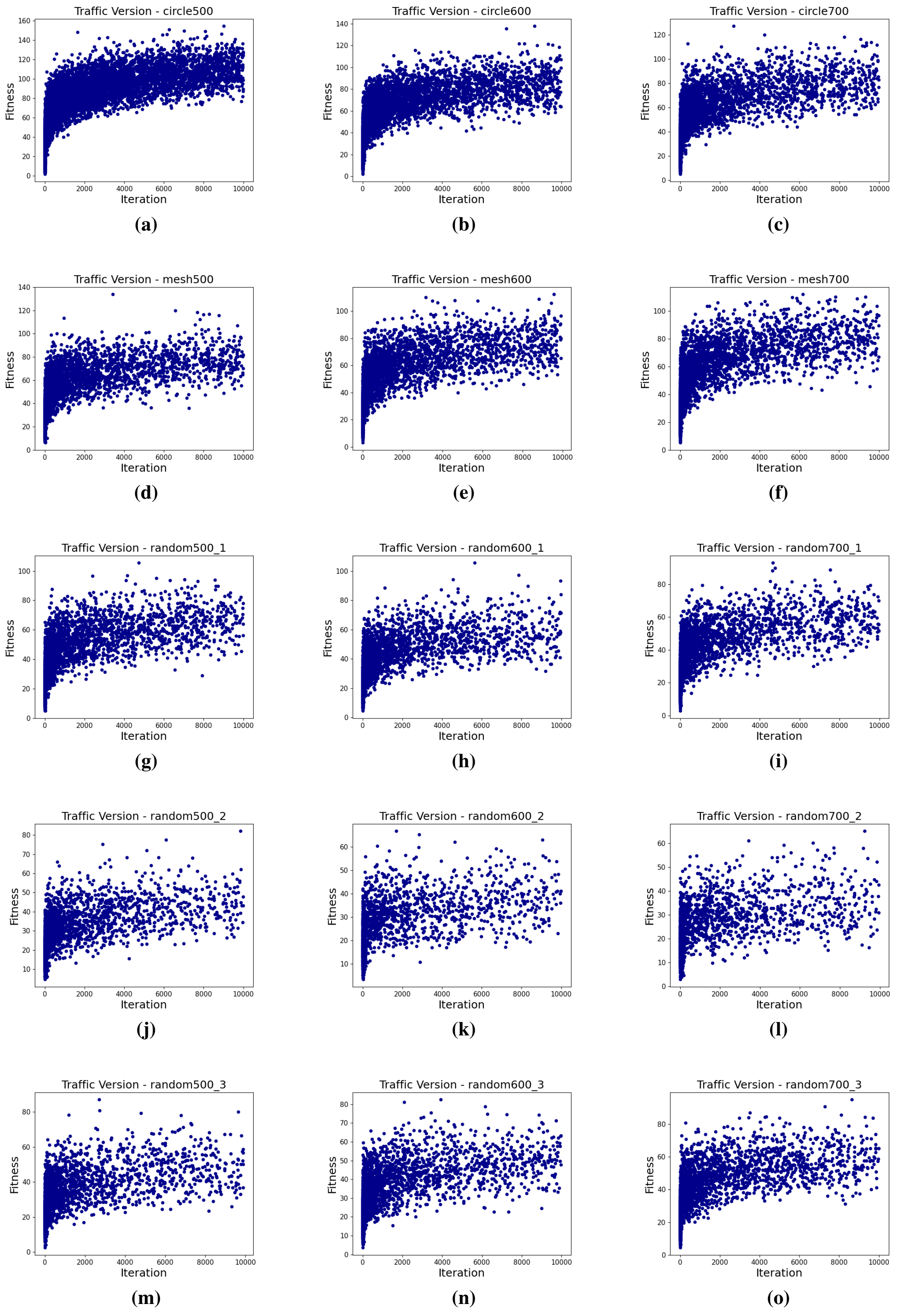


Figure 6.6: Evaluation data of the 15 Conflict graphs for the genetic traffic version.

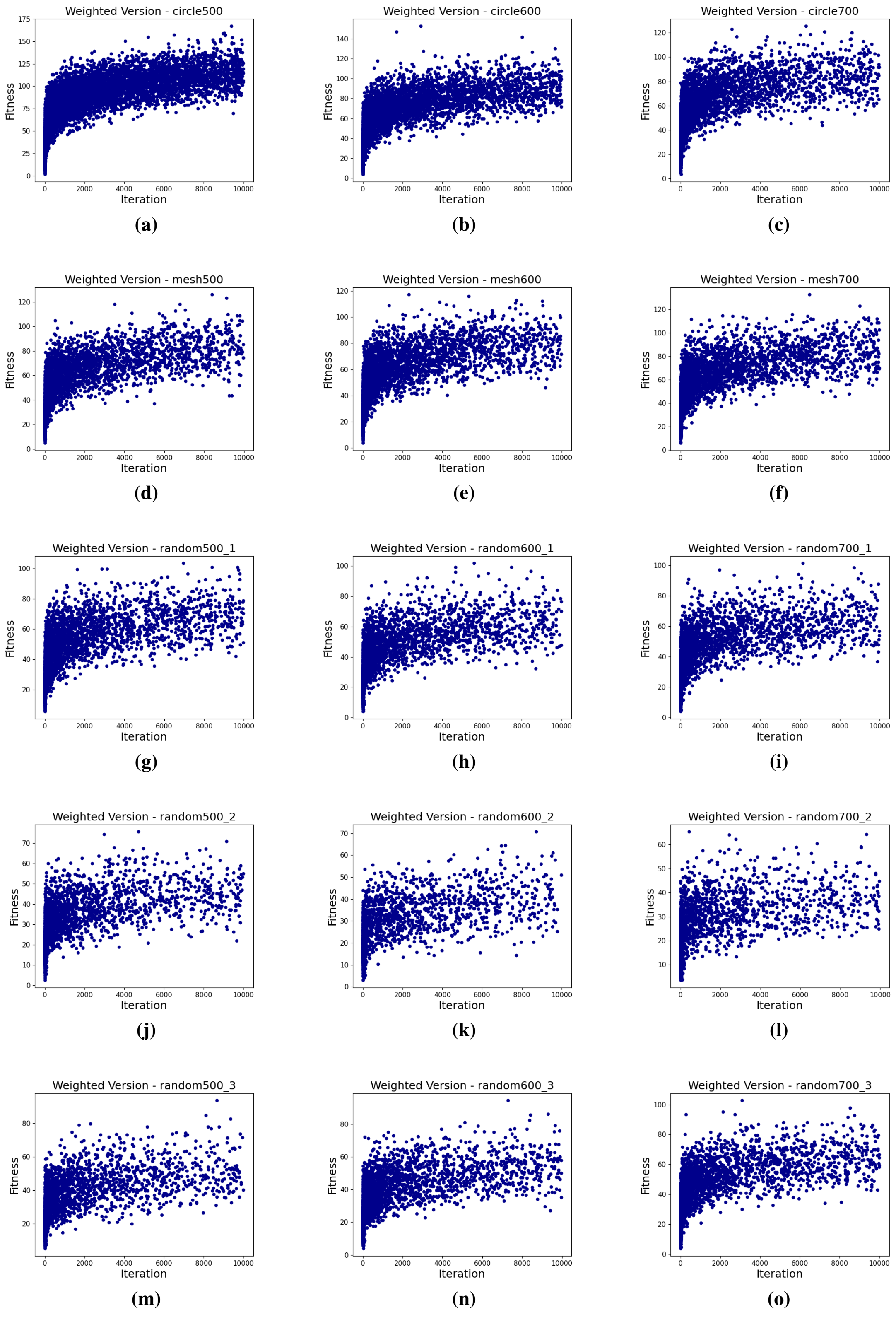
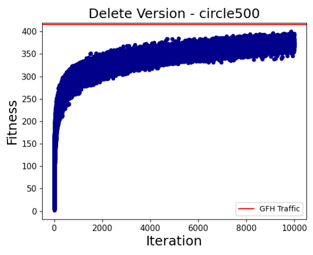
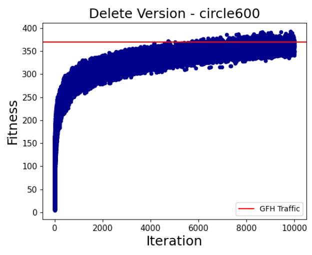


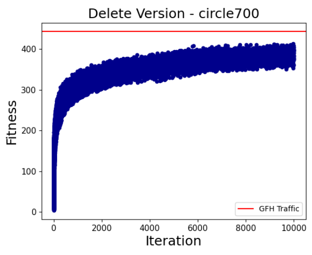
Figure 6.7: Evaluation data of the 15 Conflict graphs for the genetic weighted version.



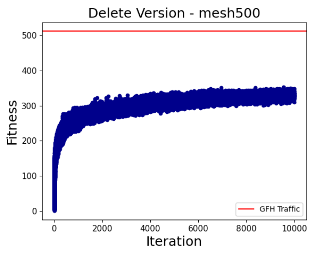
(a)



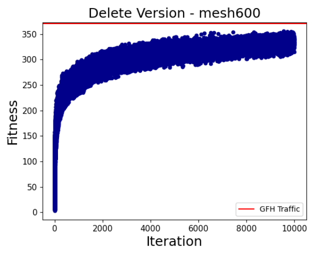
(b)



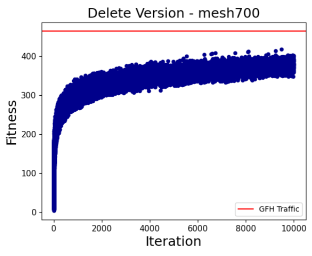
(c)



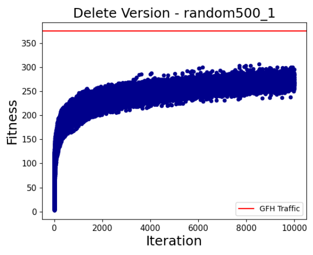
(d)



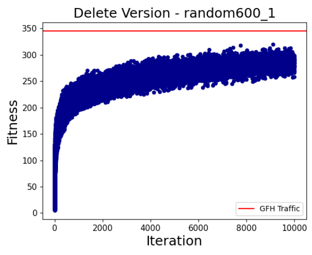
(e)



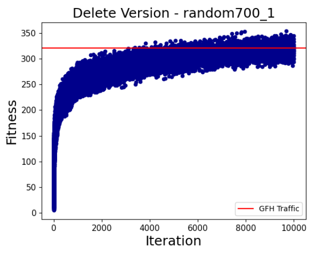
(f)



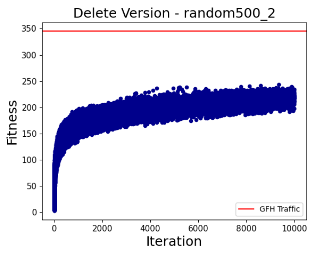
(g)



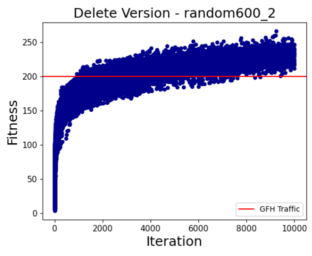
(h)



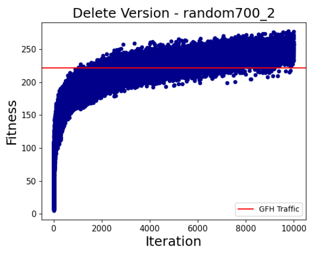
(i)



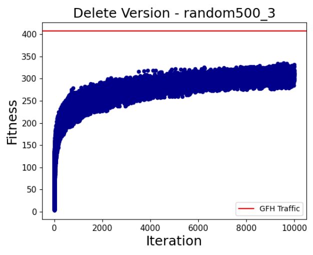
(j)



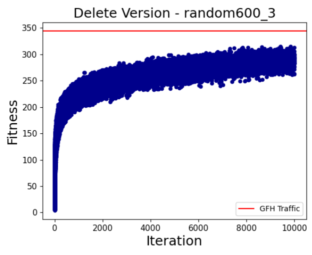
(k)



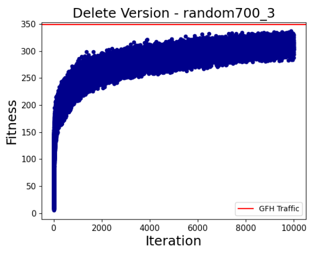
(l)



(m)



(n)



(o)

Figure 6.8: Evaluation data of the 15 Conflict graphs for the genetic delete version.

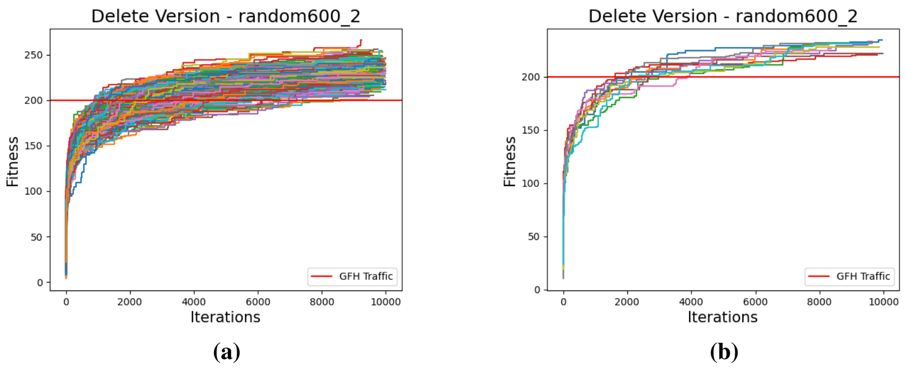


Figure 6.9: Visualization of the results as a staircase function for the conflict graph random600_2.

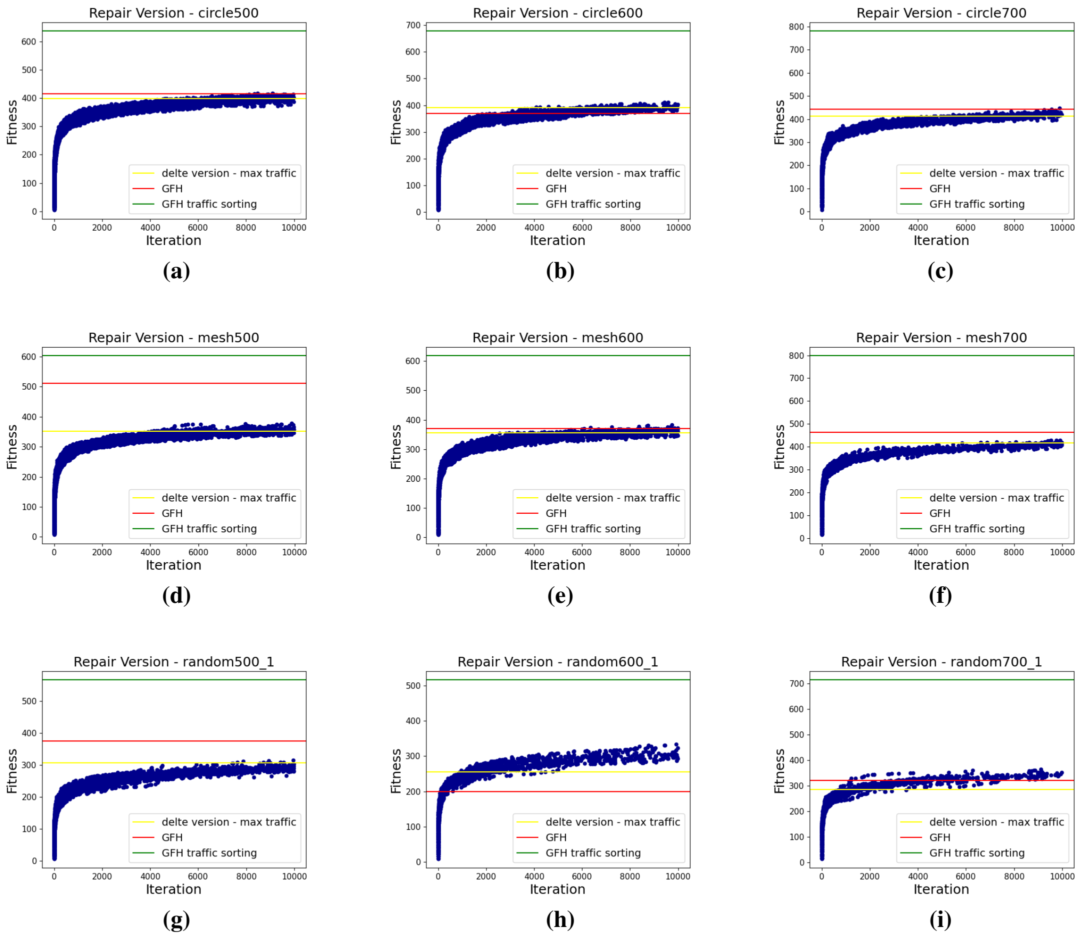


Figure 6.10: Evaluation data of 9 Conflict graphs for the genetic repair version.

7 Conclusion and Outlook

In this thesis, we have investigated the problem of time-triggered flow scheduling with the help of conflict Graphs. Since we expect a higher data traffic of flows with respect to IIoT and Industry 4.0, we aimed to accommodate as much traffic as possible over the network when scheduling. Instead of focusing only on the number of admitted flows as well-known approaches do. For our approaches, we took the solution approach for the scheduling problem with conflict graphs and finding a rainbow independent set. To adapt our problem to the underlying solution we had to find a rainbow weighted independent set instead of a rainbow independent set, which is a subset of RIS. Our first approach was an adaptation of the GFH algorithm of Falk et al. [FGD+22]. With our adaptation to the heap sort function, we were able to deliver very good results compared to GFH. More precisely, we have been able to achieve a higher traffic value for our evaluations conflict graphs from 17.72% up to 162.71%. Whereas the 17.72% is rather the exception and we can talk about an average improvement of 81.91% for our 15 conflict diagrams. Another improvement is the computation time, here we need a little less time than the original Heuristic.

Our second algorithm is an alternative way to find a weighted independent rainbow set which follows the approach of a genetic algorithm. Here we developed different approaches to increase the quality of the solution. We achieved this successfully with the delete and repair variant. Our evaluation data showed that it is more important to generate valid arrays during the mutation phase rather than to improve the choice of configurations. Due to the long computation time of the repair variant, large scaling conflict graphs will require a very high computation time. Here, it should be considered to create a combination of GFH and the Repair version. Where we use the result of the GFH as input for our algorithm to save calculation time. This can also be considered for the Delete variant.

Outlook

Future work can focus on implementing a multi-threaded solution for our genetic algorithm to improve the time needed during the mutation phase. In general, a language such as C, would be worth considering since our algorithms operate on large datasets. To improve access times to arrays, hashmaps, etc. In the case of the deletion variant, it might be possible to optimize the solution by selecting the best configuration for deletion. I.e., instead of always deleting the conflicting configurations from left to right on the array, a function could be implemented that estimates which configuration is the worse for the overall solution and then deletes it. As already mentioned before, a combination of both algorithms, s.o. GFH and repair version (or delete version), would be a promising approach to optimize the solution.

Bibliography

[ANS00] A. Atamtürk, G. L. Nemhauser, M. W. Savelsbergh. “Conflict graphs in solving integer programming problems”. In: *European Journal of Operational Research* 121.1 (2000), pp. 40–55 (cit. on p. 17).

[DN16] F. Dürr, N. G. Nayak. “No-wait packet scheduling for IEEE time-sensitive networks (TSN)”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 203–212 (cit. on p. 13).

[FDR20] J. Falk, F. Dürr, K. Rothermel. “Time-Triggered Traffic Planning for Data Networks with Conflict Graphs.” In: *RTAS*. 2020, pp. 124–136 (cit. on pp. 13, 20, 23).

[FGD+22] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. “Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows in the Real-time Data Plane”. In: *IEEE Transactions on Network and Service Management* (2022) (cit. on pp. 13, 14, 17, 20, 21, 23, 24, 35, 49).

[HL05] A. Hertz, V. V. Lozin. “The maximum independent set problem and augmenting graphs”. In: *Graph Theory and Combinatorial Optimization*. Springer, 2005, pp. 69–99 (cit. on pp. 16, 23).

[Hol22] N. Holtwerth. “Dynamic Vertex Colored Conflict Graphs for Time-sensitive Networks”. In: (2022) (cit. on pp. 13, 20, 33).

[Ian17] Ian Wright. *Airbus Uses Smart Glasses to Improve Manufacturing Efficiency*. Mar. 2017. URL: `https://www.engineering.com/story/airbus-uses-smart-glasses-to-improve-manufacturing-efficiency` (cit. on p. 13).

[Jia86] T. Jian. “An $O(2^{0.304n})$ Algorithm for Solving Maximum Independent Set Problem”. In: *IEEE Transactions on Computers* C-35.9 (1986), pp. 847–851. DOI: `10.1109/TC.1986.1676847` (cit. on p. 23).

[MP18] Y. Manoussakis, H. P. Pham. “Maximum colorful independent sets in vertex-colored graphs”. In: *Electronic Notes in Discrete Mathematics* 68 (2018), pp. 251–256 (cit. on p. 23).

[PRCS16] P. Pop, M. L. Raagaard, S. S. Craciunas, W. Steiner. “Design optimisation of cyber-physical distributed systems using IEEE time-sensitive networks”. In: *IET Cyber-Physical Systems: Theory & Applications* 1.1 (2016), pp. 86–94 (cit. on p. 13).

[SCO18] W. Steiner, S. S. Craciunas, R. S. Oliver. “Traffic planning for time-sensitive communication”. In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 42–47 (cit. on p. 19).

[Ste10] W. Steiner. “An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks”. In: *2010 31st IEEE Real-Time Systems Symposium*. IEEE. 2010, pp. 375–384 (cit. on p. 13).

[STY03] S. Sakai, M. Togasaki, K. Yamazaki. “A note on greedy algorithms for the maximum weighted independent set problem”. In: *Discrete applied mathematics* 126.2-3 (2003), pp. 313–322 (cit. on p. 23).

[Sud17] SudipSingh. *Industrie 4.0 und das Industrial Internet of Things (IIoT) – eine Einordnung*. Aug. 2017. URL: `https://www.industry-of-things.de/industrie-40-und-das-industrial-internet-of-things-iiot-eine-einordnung-a-629710/` (cit. on p. 13).

[VHT21] M. Vlk, Z. Hanzálek, S. Tang. “Constraint programming approaches to joint routing and scheduling in time-sensitive networks”. In: *Computers & Industrial Engineering* 157 (2021), p. 107317 (cit. on p. 13).

All links were last followed on October 4, 2022.

Declaration