

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit

Dynamic Vertex Colored Conflict Graphs for Time-sensitive Networks

Nico Holtwerth

Course of Study: Informatik
Examiner: Prof. Dr. Kurt Rothermel
Supervisor: Heiko Geppert, M.Sc.

Commenced: November 10, 2021
Completed: May 10, 2022

Abstract

Industrial IoT has a growing demand for dynamically changing real-time communication networks, e.g., dynamic production systems. Thereby, a trend to the *Plug-and-Produce* paradigm is evident. Using IEEE Time-sensitive network standards is a common way to uphold real-time requirements. Usually, global traffic schedules for networks are computed statically, e.g., using Linear Programs. A conflict graph based scheduler is introduced to fulfill the requirements of a dynamic real-time communication network. The conflict graph stores flow configurations as vertices and conflicts (an incompatibility of two configurations) as edges. The scheduler solves an independent set problem to compute a global traffic plan for a network. However, constructing the conflict graph constitutes a significant bottleneck. Therefore, we present approaches to optimize the conflict graph construction phase. We discuss different graph data structures storing the conflict graph efficiently. Due to the efficient insert operations, we argue that implementing the conflict graph as an adjacency list is the most efficient conflict graph data structure for this use case. Further, we present approaches to reduce the conflict computations for new flow configurations by storing meta information. Roughly, a naive insertion is an $O(|V|)$ operation checking if the new configuration conflicts with all other configurations. To reduce the complexity, we apply the *Potential Conflicts* approach by omitting conflict computations for flow configurations, which use disjoint paths. The approach shows to be effective for network topologies with many disjoint paths, e.g., Erdős-Rényi, but less efficient on topologies like trees. The runtime performance improvement achieved by the *Potential Conflicts* approach is up to a factor of 1.2-2 depending on the network topology. Another approaches we present are the *Recurrence Conflicts* and *Recurrence Non-Conflicts* checks. These checks recognize conflict reappearances that are shifted over time. The *Recurrence Conflicts* check decreases the runtime performance, however, checking for *Recurrence Non-Conflicts* improves the runtime performance immensely in cost of an exponential memory overhead. Combining the *Potential Conflicts* and *Recurrence Non-Conflicts* approaches provides the most efficient improvement. The runtime performance of the conflict graph construction phase increases up to a factor of 4-8, depending on the network topology.

Kurzfassung

Im Industrie 4.0 Kontext besteht ein wachsender Bedarf an sich dynamisch verändernden Echtzeit-Kommunikationsnetzen, z.B. bei dynamischen Produktionssystemen. Dabei zeichnet sich ein Trend zum Plug-and-Produce Paradigma ab. Die Verwendung von IEEE-Standards für zeitsensitive Netzwerke ist eine gängige Methode zur Einhaltung von Echtzeitanforderungen. Normalerweise werden globale Verkehrspläne für Netzwerke statisch berechnet, z.B. mit Hilfe linearer Programme. Um die Anforderungen eines dynamischen Echtzeit-Kommunikationsnetzes zu erfüllen, wurde ein Ansatz für einen Planer eingeführt, der auf Konfliktgraphen basiert. Der Konfliktgraph speichert Flusskonfigurationen als Eckpunkte und Konflikte (zwei inkompatible Konfigurationen) als Kanten. Um einen globalen Netzwerkplan zu berechnen, löst der Planer ein Independent Set Problem. Die Konstruktion des Konfliktgraphen erweist sich dabei jedoch als sehr aufwendig. Daher stellen wir Ansätze zur Optimierung der Konstruktionsphase vor. Wir diskutieren verschiedene Graph Datenstrukturen, die den Konfliktgraphen effizient speichern. Außerdem zeigen wir, dass die am besten geeignete Datenstruktur für den Konfliktgraphen in unserem Anwendungsfall die Adjazenz Liste ist. Dies kommt vor allem durch die effizienten Einfüge Operationen. Zusätzlich stellen wir Ansätze vor, die unterschiedliche Metainformationen speichern, um die Anzahl der Konfliktberechnungen zu reduzieren. Grob gesagt ist das Einfügen eine $O(|V|)$ Operation, die prüft, ob die neue Konfiguration mit allen anderen Konfigurationen in Konflikt steht. Dabei wenden wir den Ansatz von *Potential Conflicts* an, indem wir die Konfliktberechnungen für Flusskonfigurationen auslassen, die disjunkte Pfade verwenden. Der Ansatz erweist sich als effektiv für Netzwerktopologien mit vielen disjunkten Pfaden, z.B. die Erdős-Rényi Topologie. Topologien wie Bäume zeigen sich eher als ineffizient. Die Laufzeitverbesserung, die mit dem *Potential Conflicts* Ansatz erreicht wird, beträgt je nach Netzwerktopologie bis zu einem Faktor von 1,2-2. Ein weiterer Ansatz, den wir vorstellen, sind die *Recurrence Conflicts* und *Recurrence Non-Conflicts* Prüfungen. Diese Prüfungen erkennen wiederkehrende Konflikte, die zeitlich versetzt sind. Dabei verschlechtert die Prüfung auf *Recurrence Conflicts* die Laufzeit. Die Prüfung auf *Recurrence Non-Conflicts* hingegen verbessert die Laufzeit, jedoch auf Kosten einer exponentiellen Speichernutzung. Die Kombination der beiden Ansätze *Potential Conflicts* und *Recurrence Non-Conflicts* ist am effizientesten und verbessert die Laufzeit der Konstruktionsphase des Konfliktgraphens je nach Netzwerktopologie um einen Faktor von 4-8.

Contents

1	Introduction	13
2	Background	17
2.1	Graph Models	17
2.2	Graph Data Structures	18
3	Related Work	23
4	System Model	27
5	Research Problem	29
5.1	Conflict Graph	29
5.2	Problem Formulation	31
6	Optimize Conflict Graph Construction	33
6.1	Graph Data Structure Optimization Analysis	34
6.2	Conflict Computation Analysis	36
7	Evaluation	41
7.1	Environment and Scenarios	41
7.2	Conflict Graph Behavior for Different Network Topologies	42
7.3	Graph Data Structure Optimization Evaluation	44
7.4	Conflict Computation Evaluation	49
7.5	Discussion	61
8	Conclusion and Outlook	65
	Bibliography	67

List of Figures

2.2	Three basic computational representations of the graph in Figure 2.1a.	19
7.1	Used network graph topologies with 20 switches (yellow) and 10 end devices (purple)	42
7.2	Graph behavior on the different network topologies for scenario <i>Dynamic Small</i> with 20 switches and 10 end devices.	43
7.3	Graph behavior on the different network topologies for scenario <i>Dynamic Large</i> with 20 switches and 10 end devices.	43
7.4	Comparing the optimization goal for CSR and the basic adjacency list implementation for the <i>Dynamic Small</i> scenario.	45
7.5	Comparing the optimization goal components for CSR and the basic adjacency list implementation for the <i>Dynamic Small</i> scenario.	45
7.6	Comparing the optimization goal for the different adjacency list implementation alternatives for the <i>Dynamic Large</i> scenario.	47
7.7	Comparing the optimization goal components for the different adjacency list implementation alternatives for the <i>Dynamic Large</i> scenario.	47
7.8	Comparing the optimization goal for different flow configuration (vertex information) implementation alternatives for the <i>Dynamic Large</i> scenario.	48
7.9	Comparing the optimization goal components for different flow configuration (vertex information) implementation alternatives for the <i>Dynamic Large</i> scenario.	48
7.10	Comparing the optimization goal for <i>PIC</i> , <i>PFCI</i> and the basic adjacency list implementation for the <i>Dynamic Large</i> scenario.	50
7.11	Comparing the number of conflict computations for <i>PIC</i> , <i>PFCI</i> and the basic adjacency list implementation for the <i>Dynamic Large</i> scenario.	50
7.12	Comparing the memory usage for <i>PIC</i> and <i>PFCI</i> data structures in addition to the cumulative conflict graph size using the indices for the <i>Dynamic Large</i> scenario.	50
7.13	Conflict computation ratio for <i>PIC</i> and <i>PFCI</i> in proportion to the naive conflict computation approach for the <i>Dynamic Large</i> scenario.	51
7.14	Comparing the optimization goal for <i>RCC</i> , <i>RNCC</i> and the basic adjacency list implementation for the <i>Dynamic Large</i> scenario.	54
7.15	Comparing the number of conflict computations for <i>RCC</i> , <i>RNCC</i> and the basic adjacency list implementation for the <i>Dynamic Large</i> scenario.	54
7.16	Comparing the memory usage for <i>RCC</i> and <i>RNCC</i> data structures in addition to the cumulative conflict graph size using the indices for the <i>Dynamic Large</i> scenario.	54
7.17	Conflict computation ratio for <i>RCC</i> and <i>RNCC</i> in proportion to the naive conflict computation approach for the <i>Dynamic Large</i> scenario.	55
7.18	Comparing the optimization goal for <i>RNCC_PFCI</i> , <i>RNCC_PIC</i> and <i>RNCC</i> for the <i>Dynamic Large</i> scenario.	57
7.19	Comparing the number of conflict computations for <i>RNCC_PFCI</i> , <i>RNCC_PIC</i> and <i>RNCC</i> for the <i>Dynamic Large</i> scenario.	57

7.20	Comparing the cumulative memory usage for <i>RNCC_PFCI</i> , <i>RNCC_PIC</i> and <i>RNCC</i> for the <i>Dynamic Large</i> scenario.	57
7.21	Conflict computation ratio for <i>RNCC_PFCI</i> , <i>RNCC_PIC</i> and <i>RNCC</i> in proportion to the naive conflict computation approach for the <i>Dynamic Large</i> scenario. . . .	58
7.22	Comparing the optimization goal for <i>RNCC_PFCI</i> , <i>PFCI</i> and the basic adjacency list implementation for the <i>Dynamic Large</i> scenario.	58
7.23	Comparing cumulative memory usage for <i>RNCC_PFCI</i> , <i>PFCI</i> and the basic adjacency list implementation for the <i>Dynamic Large</i> scenario	58
7.24	Comparing the optimization goal for <i>RNCC_PFCI</i> , <i>PFCI</i> with different <i>cpf</i> values for the <i>Compare CPF</i> scenario.	60
7.25	Comparing the cumulative memory usage for <i>RNCC_PFCI</i> , <i>PFCI</i> with different <i>cpf</i> values for the <i>Compare CPF</i> scenario.	60
7.26	Comparing the optimization goal for <i>RNCC_PFCI</i> , <i>PFCI</i> adding different number of flows for the <i>Compare Flows</i> scenario.	60
7.27	Comparing the cumulative memory usage for <i>RNCC_PFCI</i> , <i>PFCI</i> adding different number of flows for the <i>Compare Flows</i> scenario.	60
7.28	Comparing the number of conflicts for the two growing dimension scenarios <i>Compare CPF</i> and <i>Compare Flows</i>	61
7.29	Comparing the optimization goal for <i>RNCC_PFCI</i> and <i>PFCI</i> with different number of candidate paths per flow for the <i>Dynamic Small</i> scenario.	61

List of Tables

7.1 Scenario Parameters 41

Acronyms

- CNC** Centralized Network Controller. 27
- DBMS** Database Management System. 24
- FNV** Fowler-Noll-Vo. 35
- GB** Gigabyte. 55
- GFH** Greedy Flow Heap. 14
- GUB** General upperbound. 23
- ID** Identifier. 18
- ILP** Integer Linear Programm. 23
- MB** Megabyte. 52
- PTP** Precision Time Protocol. 27
- QoS** Quality of Service. 27
- SDN** Software-defined Network. 27
- TAS** Time-Aware Shaper. 13
- TSN** Time-sensitive Network. 13

1 Introduction

Dynamic production environments in many industrial use-cases are still a challenge for many companies [Pro21]. Especially controlling dynamic production systems with real-time constraints is demanding. An automated (smart) factory can have multiple machines running in a tandem mode in parallel and depend on each other [Old19]. The output or information of one machine must reach another machine within a time limit. Also, robotic control systems communicate with real-time constraints to fulfill their tasks. The communication delay limit for these systems is some milliseconds [Old19]. In both cases, if the data does not arrive within the communication delay limit, one or multiple machines or even the whole manufacturing process may fail. This can be expensive and a safety risk for the working environment, including the fabric employees. One concrete example is a logistic center where a robot collision caused a fire [Wun21]. The communication between the robots was not in a certain timely tolerance, and the necessary information was not present when needed. This not only caused a fire, safety risk for employees, and material damage but also had legal consequences. The automotive industry also researches real-time communication within the car [HMKS19]. Modern cars have many sensors which need to deliver data in time. For example, an autonomous driving car must react to unpredictable situations, which can only be ensured by a real-time sensor data delivery to recognize such a situation.

To tackle this problem, a trend of using IEEE Time-sensitive Network (TSN) for real-time communication scenarios is growing. TSN introduces deterministic communication delay in IEEE 802.3 (Ethernet) networks by extending the Ethernet standard. For that the TSN Task Group published standards, e.g., the Time-Aware Shaper (TAS) (cf. IEEE 802.1Q [Soc18]) to achieve the determinism in Ethernet networks. TAS provides the mechanism to schedule traffic on switches for prioritized network packets. This enables the computation of a global traffic plan to ensure a given end-to-end delay through the network [FGD+22]. 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].

Scheduling scenarios can achieve a tremendous scaling factor where multiple thousands of devices communicate in a network. For instance, one real-world manufacturing reliably produces a vehicle body of different versions and sizes every 77 seconds [KUK16]. Thereby, 259 robots and 60,000 devices are linked together and communicate. Research literature defines the requirements for future Industry 4.0 scenarios regarding scalability regarding the number of devices in one network to several hundred or thousand [DPST18]. Such scaling problems are currently solved by knowing the needed communication flows beforehand and computing a static global traffic schedule before deploying the network [FGD+22]. A network change would require a new computation of a static traffic plan. With many alternative schedules, a traffic plan re-computation cannot be performed ad-hoc, and storing pre-computed traffic plans becomes impractical for too many alternatives. Additionally, each scenario has to be known, which reduces the flexibility for network changes or failures. Industry 4.0 requires scaling scenarios for dynamically changing networks.

One goal for Industry 4.0 is the *Plug-and-Produce* paradigm, similar to the *Plug and Play* paradigm connecting hardware devices to a desktop computer. Thereby, *Plug-and-Produce* describes the ability to connect a new field device in a manufactory that connects to the network and works on the fly [Zve17]. Currently, changing the topology and the number of field devices in a factory includes a high amount of manual work, which should be minimized with *Plug-and-Produce* [Zve17]. Therefore, research on the topic dynamic reconfiguration of mostly Ethernet networks is concluded, e.g., by [FGD+22][RPGS17]. Additional research literature defines the requirement to reconfigure the network during runtime dynamically. Therefore, as mentioned before, the network must be highly scalable in terms of network devices [DPST18].

For industrial use-cases, wireless communication in factories has become an interesting field. Since TSN extends the Ethernet standard and cannot be replaced by wireless connections, research has started for combining the 5th generation wireless communication system (5G) and real-time communication using the TSN standards [GHR+20]. The wireless connections must meet the high demands of the industrial landscape, including time synchronization and real-time requirements [GHR+20]. This comes with challenges in the form of dynamic networks and regular changing network topologies. However, solving wireless-related problems for this use case is another research topic.

Falk et al. present a conflict graph based solution to solve the problem of reconfigure dynamic networks [FDR20][FGD+22]. This research focuses on dynamically re-configuring a TSN based network to provide a real-time data plane by using actual state information of a current network state. However, to provide a *Plug-and-Produce* feature, the control plane of an industry network should compute a new traffic plan in a suitable time. Falk et al. construct a conflict graph including different flow configurations for a given set of flows and simulate adding and removing flows for different points in time. Thereby, the Greedy Flow Heap (GFH) heuristic generates the traffic schedule. The thesis covers optimization approaches to construct the conflict graph efficiently to provide a fast schedule reconfiguration.

The thesis contribution is a conflict graph construction analysis for computing a global traffic plan for TSN with the approach presented by Falk et al. [FGD+22]. We are discussing different graph data structures. Furthermore, we present, analyze and evaluate different index data structures storing meta information for an efficient conflict computation between different flow configurations in a network. We compare the optimizations with a basic implementation without storing additional meta information and discuss the memory usage overhead.

The following listing summarizes the contributions:

- Comparing and discussing different data structure implementations storing a conflict graph for a use case in the TSN topic
- Analyzing different index data structures storing meta information to optimize the efficiency to compute conflicts
- Evaluating the conflict graph data structure and meta information indices implementations

The thesis is organized as follows: we will introduce the theoretical background, including graph models and well-known graph data structures in Chapter 2. Chapter 3 discusses related work on graph data structures and other scenarios which are using a conflict graph-based approach. For that, we outline the different requirements for our use case. Chapter 4 introduces the system

model we assume during this thesis. The system model defines different components relevant for understanding and introducing the assumed network behavior. Chapter 5 discusses the research problem in detail. Thereby, the focus is on the conflict graph construction and the complexity. Additionally, we formulate our research question with an optimization goal. Chapter 6 presents and discusses different optimization approaches we are suggesting. We are discussing the runtime complexity and memory usage, and we outline the optimization improvements and the respective trade-off. In Chapter 7 we evaluate the presented optimizations with different scenarios on different network topologies. At the end of the evaluation, we summarize the achieved knowledge. Finally, in Chapter 8, we discuss relevant questions for future work and conclude the thesis.

2 Background

We start with describing important graph models for this thesis and well-known data structures to constitute a graph in computer programs.

2.1 Graph Models

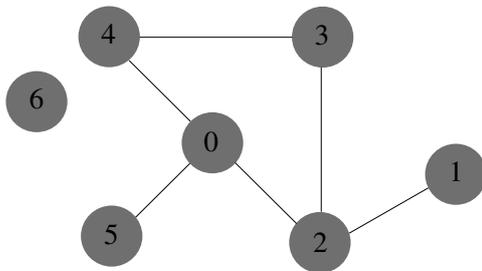
The basic graph model is a generic data structure that puts different entities into a relationship. One well-known example is the social media graphs, e.g., Facebook, where friends are connected. We name the entities *vertices* and the connections *edges*. A user's profile is a vertex in the social media graph, and the friendship is an edge. Formally, we define a graph as follows:

Definition 2.1.1 (Graph)

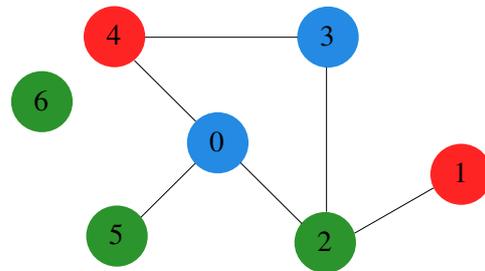
A graph $G = (V, E)$ is a tuple with a set of vertices V of size $n = |V|$ and a set of edges $E \subseteq V \times V$ of size $m = |E|$, whereby $(u, v) \in E$ and $u, v \in V$.

Two vertices are called *adjacent* to each other if they are connected with an edge. We distinguish between directed graphs where the edge tuple is ordered, e.g., $(u, v) \neq (v, u)$ are different edges and undirected graphs where the edge tuple is unordered, e.g., $(u, v) = (v, u)$ express the same edge. Literature draws a directed graph with arrows on the edge and undirected graphs without an arrow. The *degree* of a vertex $v \in V$ in the graph is the number of edges adjacent to v . Additionally, directed graphs have an in-degree and an out-degree for incoming and outgoing edges, respectively. A graph is called dense if $|E| \approx |V|^2$ or, in other words, if most of the vertices are adjacent to each other [CFV21]. Additionally, a graph is called sparse if $|E| \ll |V|^2$, or in other words, if most of the vertices are not adjacent to each other [CFV21].

Figure 2.1a shows an undirected graph $G(V, E)$ with $V = \{0, 1, 2, 3, 4, 5, 6\}$ and $E = \{(3,4), (0,4), (0,2), (1,2), (2,3), (0,5)\}$. Vertex 0 has three adjacent vertices, namely $neigh(0) = \{2, 4, 5\}$ and therefore, the degree $deg(0) = 3$.



(a) Example of a graph $G(V, E)$



(b) Example of a colored graph $G(V, E, C)$

An extension to the basic graph model is a colored graph where colors are assigned to vertices. We define a colored graph as follows:

Definition 2.1.2 (Vertex Colored Graph)

A colored graph $G = (V, E, C)$ with a set of colors C and a function $f : V \rightarrow C$ with $f(v) = c$ and $v \in V, c \in C$.

Figure 2.1b illustrates the example graph as a colored graph with $C = \{\text{red, green, blue}\}$. For example, f assigns vertex 0 blue with $f(0) = \text{blue}$. In practice, colors can be used to express domains, e.g., spectrums in the radio spectrum management domain [ZZW+].

Another essential concept is a conflict graph. A vertex-colored graph can be interpreted as a conflict graph with vertices as domain-specific entities and edges as conflicts between two entities. Colors separate the domain-specific entities into groups. Regarding a scheduling problem in railway networks, a conflict graph can model the solution space by expressing vertices as possible schedule alternatives. Thereby, edges are conflicts between alternatives, and colors express different train lines that have to be scheduled. The colored graph in Figure 2.1b can also be interpreted as a conflict graph.

In this thesis, we are using the conflict graph to model the solution space of a traffic planning problem in a TSN. After introducing essential graph models, the next section is primarily concerned with different graph model presentations with well-known data structures which can be used for implementing a graph.

2.2 Graph Data Structures

Computational graph representations can be implemented with different data structures depending on the use case and graph model. Each data structure comes with advantages and disadvantages. The choice of a data structure is affected by space limitation, read performance, and update performance [TV17]. Also, cache locality can be an optimization factor. The remainder of this section discusses three basic data structures commonly used to represent graphs, and their limits, namely, adjacency matrix, adjacency list, and compressed sparse row [CFV21][FD20]. In a basic scenario, vertex Identifiers (IDs) can be array indices or a pointer to a vertex structure. Generally, vertex Identifiers are positive continuous integer numbers starting at index 0 and ending at index $|V| - 1$. However, the valuable information of a graph is the edges and how the vertices are connected. Since we want to model a conflict graph in this thesis, we focus on undirected graphs. In practice, undirected graphs are often represented as a directed graph with edges in both directions. We will use the representation in the following as well.

2.2.1 Adjacency Matrix

An intuitive computational representation of edges is a $n \times n$ matrix A_{ij} . Existing edges are marked with a '1' and absent edges with '0'. This comes to the following matrix formulation:

$$A_{i,j} = \begin{cases} 1 & \text{if there is a directed edge from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

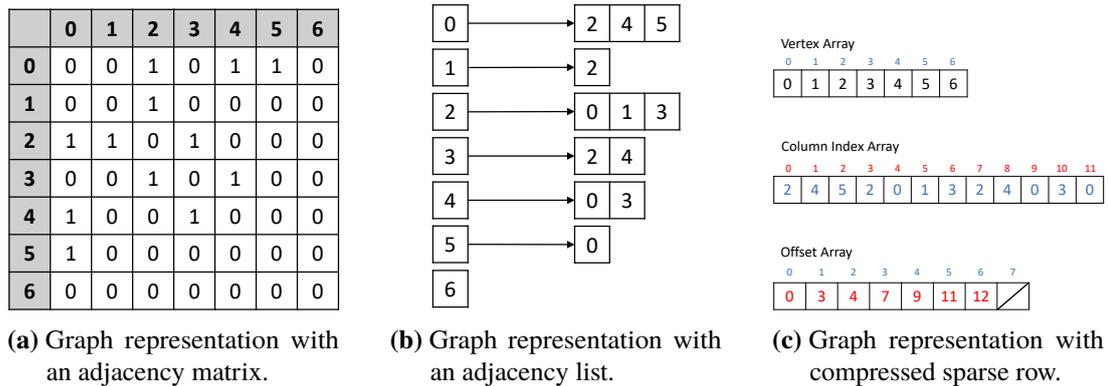


Figure 2.2: Three basic computational representations of the graph in Figure 2.1a.

The name adjacency matrix comes from the matrix indicating the adjacent vertices. Column and row ID are linked to vertex ID, e.g., an entry $a_{i,j} = 1$ expresses a directed edge $(i, j) \in E$ in graph $G(V, E)$. Figure 2.2a illustrates an example of an adjacency matrix representation of the graph in Figure 2.1a.

The adjacency matrix advantage is the fast lookup if two vertices are adjacent to each other, which is a $O(1)$ operation. However, this performance strength introduces a vast memory overhead. An adjacency matrix requires $|V|^2$ storage which becomes impractical for larger graphs. On the other hand, inserting additional vertices needs an extension of the matrix dimension. Additionally, deleting vertices introduces another overhead introducing unused memory spaces or changing the matrix dimension and adjusting the vertex Identifiers. However, inserting edges into the matrix is efficient by changing the marker at the specific position from '0' to '1'. If a graph is static and dense, adjacency matrices are suitable since the edges have to be stored regardless, and the graph can benefit from the fast lookup performance. However, if a graph is sparse, adjacency matrices store a high rate of unnecessary information on absent edges. Additionally, computing the vertex degree is an $O(|V|)$ operation by iterating a row and counting the number of '1', which is costly in sparse graphs due to the many '0' in a row. This can be optimized by using a Laplacian matrix [BKP01] which combines the adjacency matrix with a diagonal matrix storing the vertex degrees in the diagonal entries. Therefore, computing the degree is only one read operation on the diagonal entry. However, the Laplacian matrix representation requires the absence of self-loops, e.g., $(i, i) \notin E$. Additionally, for weighted graphs, one can store the edge weights into the adjacency matrix instead of only marking adjacency vertices with '1', marking them with the weight [BKP01].

All these matrix representations have an overhead storing absence edges. Therefore, if a graph is sparse, another data structure should be used, e.g., an adjacency list.

2.2.2 Adjacency List

An adjacency list stores a list of all adjacent vertices for each vertex. Compared to the adjacency matrix, the absence of markers for the non-existing edges in memory is the difference. Figure 2.2b illustrates an example how to represent the graph displayed in Figure 2.1a with an adjacency list. The advantage compared to an adjacency matrix is the storage requirement which is $O(|V| + |E|)$ by storing a list of vertices and a list of vertex neighbors, respectively. However, storing the edges

into an adjacency list decreases the lookup complexity compared to an adjacency matrix from $O(1)$ to $O(deg_{max})$ in the worst case. To compute the vertex degree or find an adjacent vertex, one must iterate the adjacency list. This is an $O(deg_{max})$ operation, with deg_{max} the maximal vertex degree over all vertices in the graph. Since sparse graphs have a high rate of missing edges, an adjacency list is a suitable data structure for representing sparse graphs with a lower memory overhead.

Additionally, for a dynamic graph, inserts in adjacency lists are a $O(1)$ or $O(m)$ operation depending on the adjacency list implementation. For example, one can implement the adjacency list with linked lists or vectors. Usually, unsorted vectors provide the advantage of inserting an element at the end of the vector in $O(1)$ because the data is stored consecutively in memory and, therefore, the last memory position can be easily computed. This applies only if the vector has enough capacity to store new elements. If not, additional memory must be allocated, which decreases the time complexity to $O(m)$ since most vector implementations copy the existing vector to the newly allocated memory position. Unsorted linked lists have a time complexity of $O(m)$ when inserting a new element at the end of the list. The end of the list must be located to update the last element's pointer. The iteration through the list is necessary because the elements are located at arbitrary memory locations. However, to optimize the operation insertion at the end of a list, the pointer to the last element can be stored, which also results in time complexity of $O(1)$ by changing the pointer of the last element to the memory location of the added element.

The deletion operation of an undirected edge has time complexity $O(deg_{max})$ by looking up the position in the adjacency lists and deleting the element, which can be performed in $O(1)$ in both vectors and linked lists depending on the implementation. Deleting an element in vectors can include additional costs by shifting elements to close the gap or move the last element into the gap. However, deleting a vertex in an undirected graph is an $O(deg_{max}^2)$ operation by iterating over the whole neighbor list and performing the deletion operation, explained above, for all adjacent vertices.

The before-mentioned discussion assumed unsorted adjacency lists. Sorting the lists can improve the time complexity for looking up or deleting a vertex because one can iterate the list, e.g., by binary search instead of linear iteration. However, inserting a new edge into a sorted list introduces the sorting overhead. Therefore, the insertion performance runtime depends on the list length and is not a constant operation.

2.2.3 Compressed Sparse Row

CSR is another graph representation that compactly stores existing edges continuously in memory. For that, CSR stores data in two, or with edge weights, in three arrays. The first array stores all column indices from the adjacency matrix, which are marked with '1'. We call this array the column index array. All indices are stored in one array continuously. Therefore, this array stores all existing edges in the graph and has the size of $\mathcal{O}(|E|)$. A second array stores the index where each row from the adjacency matrix starts in the column index array. For example, the first row starts at index 0. We call this array the offset array. Therefore, the offset array stores an offset value for each vertex ID. The memory usage for the offset array can be estimated with $\mathcal{O}(|V| + 1)$. For example, Figure 2.2c shows an example of the graph Figure 2.1a represented as CSR. The adjacent vertices of vertex 0, $neigh(0) = 2, 4, 5$ are stored on the first three positions in the column index array. If one compares the adjacency matrix representation of this graph in Figure 2.2a, one can see the row with index 0 marks these three columns with a "1". The offset array stores the value 0 at position

0, which means the adjacency vertices of vertex ID 0 are stored at index 0 in the column index array. Respectively, the second entry describes the adjacency vertices of vertex ID 1 are stored at position 3 in the column index array. Let o_{id} be the offset value stored in the offset array at index id. To retrieve all neighbors of a vertex id, one can lookup the column index array in the interval of $[o_{id}, o_{id+1})$. The interval is the reason why the offset array has the size of $\mathcal{O}(|V| + 1)$ to lookup the neighbors of the last vertex id. The third array mentioned before can be used to store weights or other information about edges.

A crucial requirement for CSR is that the vertex Identifiers are consistent, such that there is no gap. Storing the information in arrays brings high runtime performance on reading operations which can also benefit from cache locality since all information is stored in memory blocks nearby. However, update operations, which dynamic graphs frequently use, are inefficient because the arrays have to be rearranged, and a single change results in cascading updates on every update operation. For example, adding a directed edge into the column index array needs a shift operation for all entries to the right of the position where the edge is inserted. Additionally, the offset array has to be updated for all vertex Identifiers greater than the vertex ID where the edge is inserted by incrementing their offset value. Therefore, adding an edge is a $\mathcal{O}(|V| + |E|)$ operation since both arrays have to be rearranged. Similarly, deleting an edge or a vertex is also a $\mathcal{O}(|V| + |E|)$ operation by shifting elements in the column index array to the left and updating the offset values in the offset array.

Since arrays are a static-sized data structure, vectors are used to store the data in practice. Also, vectors are limited in space capacity, and if the maximum capacity is reached, the full vector will be copied into a larger memory block. To improve CSR insertion operation, additional memory can be reserved beforehand to avoid shifting operation on each insertion [FTL+20]. Alternatives are versioning approaches or batch insertion [FTL+20][MMMS15]. Each of these optimizations is a trade-off of runtime performance and memory usage.

Comparing the adjacency list with CSR, CSR has a runtime performance advantage over the adjacency lists when reading a graph [TV17]. Especially, there is no cache locality benefit in implementing the adjacency list with linked lists. Implementing the adjacency list with vectors results in a constant runtime performance penalty compared to CSR [TV17] but with an increased read performance compared to linked lists. In contrast, CSR has a vast performance overhead for update operations. The *Simple Update* protocol explained by Valiyev [TV17] is the same explained above, and the performance measurements show the performance overhead adding new nodes and edges.

3 Related Work

In this chapter, we describe related work in the area of conflict graphs, how others use the approach to solve problems, and data structures to store the graphs. Additionally, we investigate different dynamic graph data structures to store a conflict graph. We will provide a high-level overview of the problem we investigate in the thesis and show that additional work is needed.

Most important to notice are the papers from Falk et al. [FDR20] [FGD+22] which describe the work on which this thesis is based. They describe a conflict graph approach to solve a traffic planning problem in strong real-time networks with cyclic time-triggered traffic by dynamically adding and removing flows in the network. At this point, we give a short overview of the approach. Chapter 4 and Chapter 5 will explain the system model and problem statement in detail. To solve the suggested traffic planning problem variant, Falk et al. use the GFH heuristic to find an independent subset of flow configurations for a given network. The flow configurations are stored in the proposed conflict graph as vertices. They store information on which path a flow sends a packet and the point in time when a packet will be sent in each period. Falk et al. compute conflicts between different flow configurations with the flow information (e.g., packet size, the sending interval, maximum end-to-end delay) and network knowledge, e.g., the bandwidth of the links. These conflicts are stored as undirected edges between the flow configurations. Falk et al. evaluated the efficiency of this approach using a conflict graph in combination with GFH to find a schedule for flows in a given network. However, constructing the conflict graph is a significant runtime performance bottleneck because the more flow configurations are in the conflict graph, the more conflict computations must be performed when adding additional flow configurations. That is the case because each new configuration checks if it has a conflict with all configurations in the graph. A complete conflict graph with all possible flow configurations is not necessarily needed to find a feasible solution. However, more flow configurations are increasing the probability of finding a solution. Therefore, we analyze how to store the conflict graph efficiently in terms of handling the bottleneck during the construction process and providing fast read operation since GFH does many read operations on the conflict graph. In the remainder of this chapter, we describe other solutions for conflict graphs and used data structures, including a general investigation of dynamic graph data structures.

First to mention are conflict graphs applied in Integer Linear Programm (ILP) solvers [ANS00][BS21]. As also Falk et al. mention [FDR20], a general approach to solve a traffic planning problem is to use ILP formulations. For that, constraints are formulated within linear equations. Most approaches know static constraints beforehand. In contrast, our conflict graph has to adapt to new or removed previously unknown flows. The static knowledge can analyze the constraints and find conflicts more efficiently. Conflict graphs solving an ILP formulation is practical for a static problem-solving approach by converting the constraints into a graph formulation. However, it is impractical for dynamic problem-solving approaches since changing the problem's input requires updating the constraints. Using existing conflict graph approaches applied to ILP needs improvement in applying them to dynamic scenarios beforehand. Atamturk et al. [ANS00] uses an additional data structure to store General upperbound (GUB) constraints for ILP formulations

to compute the conflicts ad-hoc and do not store edges during the conflict graph construction. In our network scenario, an ad-hoc conflict computation would shift the runtime performance penalty from the conflict graph construction to the GFH execution. Further, the runtime performance would decrease massively since GFH uses degree computations which would need to be computed every time by iterating a significant part of the conflict graph.

Conflict graphs are also often used in concurrency control [DN19][Cha04]. Chaitin uses a conflict graph and graph coloring for register allocation using an adjacency list as a graph data structure. Durner and Neumann applied a multi-core parallel working conflict graph approach for transaction management, e.g., in a Database Management System (DBMS). Their conflict graph uses transactions as vertices, where each has a sequence of reading and writing operations. A directed edge will be inserted into the conflict graph if one transaction blocks another. They used the approach to find acyclic conflict graphs for conflict serialization efficiently. Compared to our approach, the conflict graph represents a current state in the system but does not try to represent a solution space. The goal is to find an acyclic graph that requires directed edges. Moreover, the size of the conflict graph is equal to uncommitted operations in the system; therefore, transactions can be removed from the graph after they finish their commit. Therefore, unlike the conflict graph proposed by Falk et al., vertices in the graph are removed regularly without a semantic impact. However, Durner and Neumann maintain an additional data structure to store the access history for each data element to find possible transactions which can conflict with each other efficiently. They ignore transactions that do not access the same data elements, and therefore, the transactions do not conflict with each other. This approach can also improve the conflict computation in our approach by storing routing information for each path and linking flow configurations to the paths they use.

Another conflict graph use case is to plan a schedule for railway network [DPH07][Won21]. Wonner uses the same approach as Falk et al. as the basis of his work. He tries different approaches to generate flow configurations, adding only promising configurations into the conflict graph and reducing the needed size of the graph to find a schedule. This part of the research is not the focus of this thesis, but both works can be combined. However, the idea of potential conflicts, which is similar to the additional access history data structure of Durner and Neumann [DN19], can be used to reduce the number of computations to find all conflicts and therefore improve the conflict graph construction efficiency.

Zhou et al. [ZZW+] are using a conflict graph for radio spectrum management. Their research in this paper is mainly on conflict estimation. They collect frequency data to build a conflict graph on measured data and build another graph with estimated conflicts. Afterward, they compared both graphs and evaluated their differences to evaluate their estimation.

Until now, we presented research with conflict graphs in other fields. These works cannot be applied directly to our approach since most of the presented work solves other problems and has different requirements for constructing a conflict graph. Our approach builds a dynamic solution space compared to other works using static scenarios. Another difference is that other problems allow regular removal of most of the vertices from the graph. However, we can use some ideas of the approaches, e.g., additional data structures to store certain meta information. In the next part of this chapter, we investigate literature related to the implementation of the concrete conflict graph data structures.

Firmly et al. [FTL+20][FD20] characterize dynamic graph mutations into three groups: In-place update, batching, and delta maps. In-place updates are the most memory-efficient mutation since the updates are performed on the data which is already in the memory. The most promising approach to adapting a data structure to dynamic graphs using in-place updates is to reserve additional space. Thereby, no additional memory space has to be allocated for each insertion [AAPO20][FTL+20][PPP02][MB09]. Examples for in-place updates are Awad et al. [AAPO20] who are using an adjacency list implemented with hash tables. Firmly et al. tried to extend CSR for dynamic graphs, and Ediger et al. implemented neighbor lists with arrays pointing to each other similar to a distributed CSR [ERBM11]. Further, batch insertion allows for pre-processing of inserted nodes which can improve the insertion runtime performance, e.g., applying concurrency [DN19][MB09] or using cache capabilities [WX18]. Moreover, delta maps or snapshots efficiently store updates on the graph because the graph itself must not be changed, but the delta is stored in an extra data structure. One example which applies this approach is LLAMA [MMMS15]. The pitfall of delta maps is the memory consumption and the performance of reading operations since every read operation has to take each delta into account. This can be improved by regularly merging the delta maps, which is nevertheless an overhead. Since our conflict graph approach tends to have a large graph, this approach could use too much memory for our problem.

Additional work has done for efficient dynamic graph data structure using parallel processing [DBS17] and cache oblivious data structures [WX18][PPP02]. Combined with batching, this approach can have the potential to speed up the conflict graph construction and also perform fast read operations.

The thesis's contribution is to apply different presented methods and data structures to the traffic planning problem proposed by Falk et al. to find a schedule in a dynamic network with hard real-time constraints. Promising approaches are in-place update data structures [WX18], optimizing CSR for dynamic graphs [ERBM11] or using hash tables for adjacency lists [AAPO20]. Additionally, techniques to reduce the number of conflict computations proposed by Wonner [Won21] and [DN19] using an additional data structure storing path information will be evaluated. We expect the most remarkable runtime improvement by reducing the conflict computations, e.g., skipping those where flow configurations use non-intersecting paths. However, the data structure storing the graph information is essential for the runtime performance since we physically add and delete edges and vertices regularly and also have a large number of reading operations due to the GFH. Therefore, the evaluation will include both aspects and how they correlate.

4 System Model

In this chapter, we describe the used system model, which is similar to the system model of Falk et al. [FGD+22]. Mainly, we are in the context of TSN. The network consists of hosts being source and destination nodes, switches, and routers. It is packet-switched with full-duplex point-to-point links, and all network nodes are synchronized, e.g., with protocols like Precision Time Protocol (PTP). We are using the isochronous TSN network traffic type, which is characterized by sending packets periodically (cyclic) and time-triggered with the awareness of hard deadlines [IEE]. For simplicity, we assume all switches, routers, and links have the same physical constraints, e.g., all links have the same bandwidth and switching delay. Each switch has one separate egress queue for each outgoing link and one ingress queue for each incoming link. Flows in the network are sequences of packets from a source to a destination node. Since flows are central units to compute schedules for a network, we define a flow with its parameters like Falk et al. [FGD+22] as follows. Let H be the set of all hosts in the network.

Definition 4.0.1 (Flow)

A flow f is defined by the immutable tuple $f = (s, t, \text{packet size}, t_{\text{cycle}}, t_{e2e})$ with $s, t \in H$.

A flow is defined by a source node, destination node, packet size, transmission cycle (t_{cycle}), end-to-end delay t_{e2e} which do not change after entering the domain. Transmission cycle t_{cycle} expresses the period in which a source node sends one packet. Because we aim for hard real-time traffic, flows are under constraints related to specified Quality of Service (QoS), namely, an upper bound to end-to-end delay. The end-to-end delay includes processing delays on switches, transmission delays, and propagation delays. To meet this requirement, we need specific network abilities like IEEE 802.Qbv switches to enable time-triggered network traffic with cyclic time schedules. Computing the schedules to program the switches is well-known to be an NP-hard problem, e.g., related to the Job Shop problem [DN16]. We consider a centralized controller, similar to the controller in Software-defined Networks (SDNs) or the IEEE 802.1Qcc Centralized Network Controller (CNC) in TSN, which performs the global schedule computation. The controller has a global view of the physical network and its current state and can additionally include domain-specific logic, which can simplify the scheduling. However, in this thesis, we assume no domain-specific logic. To understand the computation of a global traffic plan, we dive into the network's time-triggered traffic behavior in the following.

Source nodes send one packet for a flow per cycle. The needed time to transmit the packet onto a link is denoted by t_{trans} , namely, the transmission delay. The transmission delay can be computed by $t_{\text{trans}} = \frac{\text{packet size}}{\text{bandwidth}}$. Additionally, the needed time to propagate the packet via a link to another host in the network is denoted by t_{prop} , namely, the propagation delay. Every router and switch has a processing delay receiving a packet and placing it into the correct egress queue, which is denoted by t_{proc} , namely, the processing delay. Since the switches apply store-and-forward switching, the whole packet must be received before being forwarded. A packet will be sent each

$t_0 + k * t_{cycle} + \phi$, $k \in \mathbb{N}$ where t_0 is a fixed reference point in time. Thereby, ϕ expresses the phase in the interval $[0, t_{cycle} - t_{trans}]$, the time shift, when the packet will be sent in t_{cycle} but at the latest that a packet transmission does not exceed the interval.

Time schedules assign configurations to flows consisting of a phase ϕ and a path π . A centralized controller, e.g., the CNC, computes the schedule. Additionally, the controller computes candidate paths for each flow. We are computing the k-shortest paths to limit the time of a packet in the network. With the k-shortest path computation, we choose the most promising paths not to exceed the end-to-end delay. Therefore, we do not compute all possible paths with a small k. However, other strategies to compute candidate paths could be possible. The assigned configurations are called flow configurations and are defined as follows.

Definition 4.0.2 (Flow Configuration)

A flow configuration $c_{\pi, f}^{\phi}$ is an assignment of a phase ϕ and a path π to a flow f .

Limiting the number of paths reduces the solution space since for each phase ϕ , only a limited number of paths π can be chosen. Falk et al. also simplify the path selection by choosing the k-shortest candidate paths [FGD+22]. An active flow schedule has one active configuration per flow, which the controller chooses to solve the traffic planning problem. The controller maintains a list of active flows and takes a list of requested flows that should be scheduled or removed. Falk. et al. [FGD+22] aim to provide guarantees on a real-time data plane but not a real-time control plane. This implies no time guarantee for how long the controller computes a new global traffic plan.

For computing a traffic plan, we also apply the zero-queuing constraint. Intuitively, it describes that no queuing delay occurs in the system, and packets always enter an empty egress queue. With this constraint, we ensure that multiple packets are not sent via one link simultaneously.

Definition 4.0.3 (Zero queuing constraint)

Packets in the network must not be buffered on network nodes.

A switch or router forwards an arriving packet in a store-and-forward manner without a queuing delay. With the aid of this constraint, we can compute the position of a packet in the network at each point in time. For that we accumulate the delay for each hop $t_{hop} = t_{trans} + t_{prop} + t_{proc}$. The end-to-end delay consists of $t_{src} + t_{trans} + t_{prop} + l * t_{hop} + t_{dst}$, $l \in \mathbb{N}$, where t_{src} and t_{dst} denotes the processing delay of the source and destination node respectively, and l denotes the number of hops through the network. This supports the conflict computation between different flow configurations pairwise. The controller can efficiently compute if the zero queuing constraint is violated at the intersection points of the used paths.

The controller has to compute packet collisions to recognize conflicts between flow configurations to schedule the flows with specified QoS. Changing a global traffic plan to a new plan requires changes on the switches to configure new or updated flow configurations. Deploying a traffic plan update to the network has different challenges, e.g., blackholing, packet loops, or congestion [NCC17]. To prevent network update errors, Falk et al. [FGD+22] describe a technique to solve such problems and how the controller performs a network update, which is out of the scope of this thesis. Therefore, we omit the problem of updating the global traffic plan physically on the switches and focus on computing a new global traffic plan with the conflict graph approach. In the remainder of this thesis, we refer to the described controller as the scheduler.

5 Research Problem

In this chapter, we explain the conflict graph construction, the conflict computation, and the operation complexities. Additionally, we derive the problem statement with a research question and an optimization goal.

5.1 Conflict Graph

Falk et al. published a solution for solving the dynamic traffic planning problem with a graph-based approach [FDR20]. Typically, other researchers solve the problem by formulating constraints and finding a solution, e.g., by executing an ILP. These approaches work on the routing and scheduling level [SDT+17], e.g., by formulating constraints on individual network devices or links. The result is a tight coupling between single flow configurations and the global traffic plan. In a static planning case, a solver computes the global traffic plan once and finds feasible configurations for each flow, which will be installed on the network devices. Changing routing or scheduling variables of a stream or the network itself can result in a cascading change in the global traffic plan.

Falk et al. are constructing a conflict graph to resolve this tight coupling, where vertices are flow configurations, and edges indicate conflicts between two configurations. Each vertex is colored according to the related flow. Therefore, the conflict graph is a colored graph with color set C whereby $|C| = |flows|$. Generating a global traffic plan from the conflict graph is done by solving the colorful independent vertex set problem [FGD+22], where the vertex set contains one vertex for each flow. If the solution of the colorful independent vertex set problem contains a flow configuration for each flow, the size of the vertex set is equal to the number of flows since only one configuration per flow will be installed in the network. However, if the scheduler cannot find a feasible schedule for all new flows, these flows will be rejected. The scheduler only rejects flows in new requests and never rejects already scheduled flows. However, the scheduler can change the configuration of a scheduled flow which is referred to offensive planning which allows a better network utilization [FGD+22].

We use the conflict graph to perform offensive traffic planning and compute a new global traffic plan during runtime as fast as possible. In contrast, to solve an ILP every time to compute a new plan and find a feasible solution, the conflict graph does not provide the whole solution space. However, it iteratively adds new flow configurations and tries to solve the colorful independent vertex set problem using a subspace of the solution space. This approach yields significant runtime performance benefits since we are looking for a feasible solution in a given subset by reusing state information. Additionally, we can change already scheduled flow configurations to adjust the feasible schedule for newly added flows.

5.1.1 Construction and Modification

The next part will describe and analyze the conflict graph construction. This serves as the basis for our analysis to optimize the construction through different data structures and optimization approaches. First, we explain the general procedure to admit a new flow to the active flows. An overview of the high-level steps are given in the following enumeration:

1. examine candidate paths, e.g., with Yen's k-shortest path algorithm [Yen71]
2. generate flow configurations for the conflict graph
3. add flow configurations to the conflict graph and examine pairwise with each flow configuration in the conflict graph if they are in conflict
 - a) compute the hypercycle of the two flows in question
 - b) check if the zero-queuing constraint is violated in one of the network devices
 - c) if true, add edge between these two flow configurations

The scheduler performs step 1 once at the beginning and computes a set of paths that will be taken into account to generate possible flow configurations. Like Falk et al., we use a static upper bound on the number of candidate paths which we compute with Yen's shortest path algorithm [FGD+22][Yen71]. A limitation of path length is implicitly given by a short end-to-end delay. Thereby, scenarios with high end-to-end delays and many path choices without an upper bound could be impractical for many candidate paths which increases the solution space size and, therefore, the conflict graph size.

Step 2 generates flow configurations for requested flows to admit or for active flows. Different strategies result in various conflict graph sizes, which impacts the runtime performance. As an example, Falk et al. traverse the route-phase space with a phase step size of the 75-th percentile of the transmission duration of all flows [FGD+22]. Wonner optimizes this strategy within the context of line scheduling in train networks [Won21].

This thesis focuses on step 3 to efficiently update the conflict graph. The scheduler checks a generated flow configuration for a conflict with all flow configurations already inserted into the conflict graph. Practically, one can compute a conflict, e.g., by discretizing the time and using an array for each network device. The time slots when a packet is using the device is marked for one hypercycle of the two flows in question [FDR20]. Another approach to check for a conflict is to compute the time intervals where the packets use the same link between two switches or routers. If intervals overlap, a buffer delay will occur, and the flow configurations have a conflict [FGD+22]. For performance reasons, the scheduler does not build the graph from scratch every planning iteration but keeps the old graph in memory. The benefit is to have a state which can be used for the next planning iteration.

5.1.2 Complexity of constructing the conflict graph

Further, we discuss the conflict graph construction complexity. Thereby, we ignore the implementation complexities like the used data structure.

The most complex part is computing and inserting the conflicts as edges into the conflict graph. Inserting a new configuration without any edges is a constant operation in addition to the complexity of the underlying data structure. As discussed before, inserting the resulting conflicts as edges to the conflict graph requires a pairwise conflict check of the new flow configuration with all flow configurations in the conflict graph. This includes computing the hypercycle for each flow configuration pair and looking for a zero-queuing constraint violation during the computed hypercycle. Additionally, we have to compute the path intersection points to find a zero-queuing constraint violation. Therefore, let P be the set of paths and H be the set of all hypercycles. We say $l(p)$ with $p \in P$ and $l(h)$ with $h \in H$ is the length of a path or hypercycle, respectively. The maximum of all path length is defined by $\hat{P} = \max l(p) \in P$, as well as, $\hat{H} = \max l(h) \in H$ is the maximum hypercycle. Computing all conflicts by comparing a new flow configuration with all flow configurations in the conflict graph has the worst case complexity of $O(|V| * \hat{H} * \hat{P})$. The complexity of computing a conflict between two flow configurations depends on the size of their hypercycle $l(h)$ and the path lengths $l(p)$, since the longer the paths, the more conflicting points can exist. The analysis is conservative and intuitive. Path checks beforehand can reduce the factor \hat{P} by only checking the path intersection points, where packets can collide. Let us denote the number of path intersection points as $pi(p)$, whereby $pi(p) < l(p)$. If $pi(p) = l(p)$, the paths would be the same. Each path is stored once and can be used by multiple flows.

Until now, we discussed the complexity to compute a conflict. The next step is to insert the resulting edge into the conflict graph data structure in memory. This complexity depends on the used data structure. As an example, the time complexity inserting a conflict into an unsorted adjacency list implemented as linked list, depends on the degree of the conflicting configurations which is a $O(deg(c_0) + deg(c_1)) = O(d)$ operation. This comes from the fact that by inserting an element into a trivial linked list, the end of the list must be found by iterating the list. For each conflict, we insert two edges because of the notion of an undirected edge. In contrast, the time complexity of inserting a conflict into an unsorted adjacency list implemented as a vector is a constant operation most of the time (if no additional space has to be allocated in memory) by pushing the conflict ID to the end of the vector. Therefore, the complexity can be estimated with $O(push(c_0) + push(c_1)) = O(c)$. To compare the complexities, $O(d)$ depends on the degree of a neighbor list, and $O(c)$ is a constant amortized operation. We formally describe the dependency of the graph data structure, denoting the complexity of inserting a conflict with $O(i)$. The time complexity of building the conflict graph increases to $O(|V| * H * P * i)$.

5.2 Problem Formulation

Constructing the conflict graph as explained in Section 5.1.1 has a bottleneck since inserting a configuration with all its conflicts has a significant overhead, which tends to be slow while the graph is growing. This comes from the conflict computation effort to find all conflicts by checking all flow configurations in the conflict graph. We focus on the concrete implementation to reduce the complexity of constructing the conflict graph to solve the traffic planning problem. The aim

is to find and evaluate new data structures and optimization approaches to improve the runtime performance of update operations, like deleting or inserting new flow configurations. Therefore, we derived the thesis research question as follows:

RQ: Which data structures and techniques improve the construction and update performance of the conflict graph adapting to changing flows?

Interesting evaluations are the runtime performance and memory usage while the conflict graph mutates. An additional measure is also the GFH execution time to evaluate the reading behavior on the conflict graph to compute a new schedule. The thesis mainly focuses more on runtime performance to limit the impact of the bottleneck of constructing the conflict graph. The goal of RQ is to minimize the overall time to construct and update the conflict graph, as well as run an algorithm like GFH on the conflict graph. Our scheduler executes three phases for each request to insert or delete new network flows. First, remove the requested flows. Second, insert the new flows and the configured number of flow configurations for each flow, and third, run an independent set algorithm which will be GFH in our case. The insertion operation adding new flow configurations with the resulting conflicts into the graph is expected to be the most expensive part. Deletion is expected to be faster since no conflict computation has to be performed, and deletion can be optimized by postponing the actual deletion process to a scheduler idle phase by marking the respective configurations. Additionally, the runtime of GFH can be limited by the rounds to execute the heuristic, which impacts the outcome. However, optimizing GFH is another research topic that is not covered by this thesis. We will use the implementation described by Falk et al. with a default limit of 3 rounds [FGD+22]. From the discussion above, we derive the following optimization goal:

$$\min \sum \left(deletion_{time} + construction_{time} + GFH_{time} \right)$$

In other words, we aim to improve the runtime of deleting and inserting flow configurations and run an algorithm on the conflict graph and use the sum of the runtime of all the operations to compare our optimizations. We will analyze optimization approaches of the conflict graph construction in two dimensions: first, analyze data structures to store the conflict graph, and second, reduce the effort to compute conflicts for the flow configurations. We denote the dimensions as *Data Structure dimension* and *Conflict Computation dimension* respectively. Both dimensions can be combined and do not exclude each other. Additionally, we will discuss the memory trade-off for the optimization approaches, especially in the *Conflict Computation dimension*. We collect analysis data by executing scenarios that define time steps. The scheduler performs all operations defined in the optimization goal in each time step. We will evaluate all of the named operations for each time step depending on the graph size on different network topologies.

6 Optimize Conflict Graph Construction

This chapter discusses graph data structures and optimization approaches to efficiently construct the conflict graph. First of all, we provide an overview of the optimization strategy. Afterward, we explain the optimizations and data structures and provide intuition on their efficiency. As mentioned in Section 5.2, we analyze optimizations in two dimensions, first, the *Data Structure dimension*, and second, the *Conflict Computation dimension*. Both dimensions are not disjoint and can be combined.

The *Data Structure dimension* has the challenge of finding a data structure that provides a suitable trade-off between fast insertion of flow configurations and conflicts, fast reads, e.g., for an algorithm like GFH, and deleting flow configurations. Currently, deletes are only performed if there is a delete request for a flow, which deletes all its flow configurations and their conflicts. In further optimizations, deleting probably unnecessary flow configurations to reduce the graph size can be considered. However, the actual deletion from the graph can be postponed, e.g., by setting a deletion flag. Therefore, deleting flow configurations from the graph is not as critical as insertion. However, our evaluations will delete the flows and flow configurations directly without marking them. Another critical aspect for an independent set algorithm like GFH is the efficiency of providing the conflict graph data, including flow configurations and their conflicts. Thereby, GFH often iterates over conflicts of a particular flow configuration. Preliminary evaluations at this point are showing that GFH is not a critical performance factor. GFH and deletion both require fast access to flow configurations for a specific flow. Therefore, all graph data structure implementations will maintain an index that links flows to flow configurations for a fast lookup in which flow configuration is stored for a particular flow. This affects the construction performance since there is another overhead to build the index. However, maintaining the index enormously improves the GFH runtime performance. In contrast, to naively search through all flow configurations, the overhead is worth taking.

The *Conflict Computation dimension* will deal with the question of how to reduce the computation effort to find a conflict. We analyze to store different meta information to reduce the conflict computation effort. We will compare the runtime with the naive conflict computation, as explained in Section 5.1.1. Thereby, we are interested in the runtime improvement and the additional runtime and memory overhead storing the meta information. The challenge for this dimension is a trade-off between runtime and memory. Therefore, we evaluate the memory usage for each index. Additionally, this dimension depends significantly on the network topology. For example, the size of a path intersection index depends on how many paths overlap. Moreover, the more intersecting paths, the more conflict computations have to be performed, increasing the runtime.

6.1 Graph Data Structure Optimization Analysis

We start with analyzing the *Data Structure dimension* and discuss the general behavior of the conflict graph data structure and the resulting optimizations. The requirements of the conflict graph are defined in the optimization goal in Section 5.2 and include fast insert, delete, and GFH execution. In the remainder of this section, we compare static and dynamic data structures and discuss their suitability.

6.1.1 CSR and Adjacency Lists

First of all, we are comparing two well-known data structures: adjacency lists (cf. Section 2.2.2) and CSR (cf. Section 2.2.3). CSR is known for a high read performance through cache locality and therefore decreases the execution time of the GFH compared to adjacency lists. However, CSR has a slow construction time compared to adjacency lists since, for each insertion of a conflict, the offset array and column index array have to be adjusted. This results in many shifting operations which have a significant runtime performance overhead and, as mentioned in Section 2.2.3, has a complexity of $O(|V| + |E|)$. Shifting operations are necessary since the conflicts in CSR are stored continuously. In contrast, the adjacency lists are stored at separate memory locations, and therefore, adding a conflict in an unordered adjacency list is a single push operation. Similarly, the deletion of a flow configuration and the related conflicts in CSR results in adjusting the two arrays with shifting operations to close the resulting memory gaps. Depending on the implementation of the adjacency list, deletion from an unordered list is also a constant operation. Using a hash set for a neighbor list implementation is straightforward. A vector-based implementation can be optimized by using a *swap remove* operation, which takes the last conflict in the list and puts it in the place of the deleted conflict. A *swap remove* operation prevents shifting the whole vector from the place of the deleted conflict to the end of the vector to close the gap. Overall, CSR is not suitable for our use case since insertion and deletion operations are too expensive and result in an inappropriate overhead. Adjacency lists are a promising data structure for our conflict graph use case. With some optimizations, it is expected to efficiently perform the insert, delete, and read operations on the conflict graph. Since the main focus is the construction phase improvement, optimizing CSR seems not to be goal-oriented. The literature discussed in Chapter 3 has a more considerable overhead for all implementations compared to the constant amortized insertions and deletions using an adjacency list. Therefore, we analyze different adjacency list implementations.

6.1.2 Adjacency List Optimizations

As described in Section 2.2.2, an adjacency list can be implemented among other alternatives using vectors or linked lists. We omit the implementation of a linked list because it has no runtime performance advantages over a vector implementation [TV17]. Since we want to reduce the conflict graph construction overhead, we store the adjacency lists unsorted and therefore take advantage of the constant amortized delete and insertion operations as discussed in Section 6.1. Additionally, we do not have overhead sorting the adjacency list, which can decrease the insertion performance depending on the adjacency list's size. GFH shows a minor impact on our optimization goal. Sorting

an adjacency list could only improve the read performance during GFH execution through cache locality and faster lookup. Therefore, at this point, we omit a deeper analysis of data structures for sorted adjacency lists, like implementing the lists as a binary search tree.

We have two points to optimize storing adjacency lists. First, we optimize the data structure storing all neighbor lists, and second, we optimize the neighbor list itself. The data structure to store all lists can be intuitively implemented as a hash map, whereby the key is the vertex ID and the value the neighbor list. A drawback of this implementation is the hashing overhead. For example, at the time of writing, Rust's standard hash map provides a collision-resistant hash function (SipHash 1-3) at the cost of longer runtime for hashing a key. An alternative hash map implementation uses a Fowler-Noll-Vo (FNV) hash function, which performs better on small keys like integers but does not provide protection against collision attacks in contrast to SipHash 1-3 [Net22]. A third option is the *txhash* function, which was extracted from the Rust compiler. *TxHash* has speedup of 6% [Net22] compared to the FNV based implementation and also does not provide protection against collision attacks. Since our implementation does not use user-crafted keys for the hash map, we do not need protection against collision attacks. Therefore, we use the fastest hash function to reduce the hashing overhead.

Comparing a hash map with a vector implementation for the data structure storing the neighbor lists is interesting. The vector implementation stores the neighbor lists into a vector at the positions of the vertex Identifiers. The drawback is a high amount of gaps between vertex Identifiers which can appear due to deleting flow configurations. This results in unused positions in the vector and wasting space. However, this drawback can be limited by reusing vertex Identifiers which introduces additional bookkeeping overhead. On the other hand, a hash map with a low load factor also wastes space due to unused positions in the map. A hash map with a high load factor has fewer unused spaces than one with a low load factor.

We expect the GFH execution runtime performance is faster for the vector implementation, which does not have the hash function execution overhead. Therefore, the constant operation for the hash map depends on the used hash function and its performance. We are also expecting deleting a neighbor list to have the same runtime performance since no adjustments in the hash map have to be done, and for the vector implementation, we are only marking a gap. The lookup is fast for both approaches depending on the hash function.

The second point of optimization is the data structure of the neighbor list itself. Thereby, we also want to analyze a vector implementation against a hash set to verify the overhead of the hashing process and the iteration efficiency. A hash set operates like a hash map, only storing keys but no values. Inserting an element into the hash set is a constant amortized operation amortized when no re-hashing happens. Therefore, the runtime performance depends on the hash function. We are also using the *txhash* implementation for the hash set, which is the fastest widely used hash function for Rust's hash maps and hash sets. Inserting a conflict into a neighbor list implemented with a hash set includes the hash function overhead compared to the vector. Additionally, a hash set provides all stored elements by iterating the whole allocated space since each position can contain an entry. In contrast, a vector iterates only over the elements since they are stored continuously, and the end of the vector is known. Therefore, we expect that the neighbor lists vector implementation is faster executing the GFH than the hash set. On the other side, the hash set will be more efficient in deleting conflicts from a neighbor list since deletion is a constant operation. In a vector, the element's position has to be found beforehand.

6.2 Conflict Computation Analysis

For the *Conflict Computation dimension* we are analyzing how we can reduce the number of conflict computations between the flow configurations. We are using three strategies in this section. First, we are skipping computations between flows using disjoint paths. Second, reduce the number of iterations over the stored flow configurations to such which have a potential conflict. Third, recognizing conflict reappearances.

6.2.1 Potential Path Conflicts

The first optimization, also described by Wonner [Won21], is to store tuples of conflicting paths and skip a conflict computation if two flow configurations use disjoint paths. For a fast lookup, we store paths in a path map and address a path with the path ID. We denote the set of all paths in the system with P . One path $p \in P$ is a list of network devices that describe the path in an ordered way. A potential conflict PC exists if two configurations use overlapping paths by using the same link. Formally we describe PC as a function with $PC : P^2 \rightarrow \{True, False\}$ as follows:

$$PC(p_1, p_2) = \begin{cases} True & p_1.links \cap p_2.links \neq \emptyset \\ False & \text{otherwise} \end{cases}$$

When two flow configurations are using overlapping paths, we say that they have a *Potential Conflict*. In Section 5.1.2 we already provided an overview of the conflict graph construction complexity. The construction runtime depends on the conflict graph size. Therefore, each added flow configuration decreases the construction phase runtime performance. With the *Potential Conflict* check, we reduce the number of conflict computations to $|PC| \leq |V|^2$. To find all *Potential Conflicts*, we have to check if a newly generated path has at least one intersection point with all other paths. Thereby, an intersection point is one link used by two different paths. We call this check the *Path Intersection Check (PIC)*. The resulting index data structure is a minor conflict graph storing overlapping paths. We implement this data structure as an adjacency list with a hash map storing the lists with a hash set. The key is the path ID, and the neighbor lists contain the conflicting path Identifiers.

As also mentioned by Wonner [Won21], this approach is a trade-off between runtime and memory. The worst case memory usage depends on the number of paths and can be estimated with $O(|P|^2)$. However, in practice, memory usage depends on the network topology. For a tree topology like in Figure 7.1b, many *Potential Conflicts* can be estimated because most of the paths have an intersection point. For an Erdős-Rényi topology like in Figure 7.1c, we can expect many disjoint paths. The memory usage depends on the number of candidate paths and the number of *Potential Conflicts* which differs between network topologies. Additionally, the second trade-off is between the runtime of computing all flow configuration conflicts and the runtime constructing the *PIC* index. The construction time of the *PIC* index depends on the average path length $l(\bar{P})$ and the new path length $l(p_{new})$ and can be estimated by $O(l(\bar{P}) * l(p_{new}) * |P|)$ for each newly inserted path. In other words, for all paths, we check for a path intersection by iterating through both paths. Because the index is a minor conflict graph, the construction performance depends on the number of paths in the path map and decreases by inserting new paths, which results in more path intersection computations.

In a network with many primarily disjoint candidate paths, we can reduce the conflict computation runtime because the set of *Potential Conflicts* is small. Therefore, we can skip many conflict computations between two flow configurations. However, if many *Potential Conflicts* exist, we must execute many conflict computations. Therefore, we expect minor improvement for *PIC* the more overlapping paths exists.

To reduce the iteration overhead over all vertices in the conflict graph and check for a *Potential Conflicts* beforehand, we will explain an approach with a second index to reduce the number of iterations over the vertices.

6.2.2 Path Flow Configuration Index

To optimize the *Potential Conflict* check, we add a data structure storing the flow configurations using a particular path. *PIC* has only a directional connection from the flow configurations to the paths. We have only the information which path a flow configuration uses. Therefore, we optimize this approach by introducing a bidirectional connection between flow configurations and paths. This enables the possibility of efficiently looking up all flow configurations using a particular path. We introduce a data structure additional to the *PIC* index implemented by using a hash map with the path ID as key and a list of flow configuration Identifiers as value. In combination with the *PIC* index, we can determine which flow configurations have a *Potential Conflict* with a particular flow configuration. We name the combination the *Path Flow Configuration Index* (PFCI). The benefit of this approach is to reduce the iteration overhead over the flow configurations stored in the conflict graph. With all approaches before, we have the complexity estimation of $O(|V| * \hat{H} * \hat{P})$ as explained in Section 5.1.2. With *PFCI* we reduce the complexity to $O(|PC| * \hat{H} * \hat{P})$, since we can expect that $|PC| < |V|$ due to spatial isolation of paths in the network. This is, in most networks, the case. However, we expect the speed up scales differently depending on the network topology. The dependency on the network topology is the same as explained in Section 6.2.1 since we are using the *PIC* index as the basis.

The previously discussed optimizations (*PIC* and *PFCI*) tried to reduce the number of conflict computations and flow configuration comparisons. However, preliminary evaluations show a bottleneck in the conflict computation of *Potential Conflicts* which are not conflicting. The following optimization addresses the problem and reduces the number of conflict computations of flow configuration pairs that are not in conflict.

6.2.3 Recurrence Conflicts

The *Recurrence Conflicts* idea recognizes conflict reappearances that are shifted over time and therefore skipping conflict computations. In the following, we are defining the conflict notation to explain the idea:

Definition 6.2.1 (Conflict Notation)

Let c be a flow configuration as defined in Definition 4.0.2. Then, $c_{\pi', f'}^{\phi'} \circ c_{\pi'', f''}^{\phi''}$ denotes a conflict between the two given flow configurations.

If two flow configurations with phase ϕ' and ϕ'' have a conflict, the same conflict occurs for any time shift Δ for all configuration pairs using the same path and flow combination with phase $\phi' + \Delta$ and $\phi'' + \Delta$. As an example, w.l.o.g. we assume a conflict between two flow configurations $c_{p_1,A}^{t_0}$ and $c_{p_2,B}^{t_0}$ at the first egress port in each path. Additionally, we assume the same bandwidth on all links in the network. Both flow configurations send packets of the same size, w.l.o.g. at time t_0 . At the first egress port, both packets enter the queue for the link simultaneously, and therefore, one of the packets is buffered. This violates the zero-queuing constraint, and the flow configurations conflict. A similar situation occurs, when we generate flow configurations $c_{p_1,A}^{t_0+2}$ and $c_{p_2,B}^{t_0+2}$ which send their packets at time $t_0 + 2$. We can derive the conflict from $c_{p_1,A}^{t_0} \circ c_{p_2,B}^{t_0}$, since both flow configurations send their packets in the same time distance shifted by two time units. Therefore, a *Recurrence Conflict* can be found by comparing the phase differences of two flow configurations and checking if we already know a conflict between two flow configurations using the same path and flow combination having the same phase offset. Let P be the set of all paths and F the set of all flows in the system. In the following, we define a *Recurrence Conflict* formally.

Definition 6.2.2 (Recurrence Conflict)

A *Recurrence Conflict* is defined by the implication $c_{p_1,A}^{t_1} \circ c_{p_2,B}^{t_1'} \implies c_{p_1,A}^{t_1+\Delta} \circ c_{p_2,B}^{t_1'+\Delta}$ with $p_1, p_2 \in P$ and $A, B \in F$ and $A \neq B$ and $\Delta \in \mathbb{Z}$

A suitable data structure has to store the phase offsets for conflicting flow configurations depending on the flow Identifiers and the used path to tackle the optimization in practice. Storing the flow ID information is necessary to know the packet size and period. Additionally, one flow can send the packets via multiple candidate path alternatives. Therefore, the combination of this information is needed to derive a *Recurrence Conflict*. In benchmarks, we observed that the conflict computation time for flow configurations that do not conflict with each other is the most expensive conflict computation scenario. To compute a conflict, we must compute the time when a packet from both flow configurations uses the link of each path intersection. This must be computed for the whole hypercycle of the two flows. In contrast, two flow configurations with a conflict can end the computation when a zero-queuing constraint violation has been found. To reduce this behavior, we will additionally analyze the *Recurrence Non-Conflicts* approach to derive non-conflicting flow configurations. Thereby, Definition 6.2.2 can also be applied to *Recurrence Non-Conflicts* by changing the Definition 6.2.1 denoting a non-conflict between two flow configurations. *Recurrence Non-Conflicts* uses the same idea as explained above with the difference to store phase offsets of non-conflicting flow configurations.

The data structure's worst case memory size can be estimated by $O(|\text{flows}|^2 * |\text{paths}|^2 * \#\text{offsets})$ which is an exponential space usage. However, in practice the factor $O(|\text{paths}|^2)$ for the *Recurrence Conflicts* will be reduced to $O(|\text{conflicting paths}|^2)$. Additionally, not every flow must be in conflict with all other flows, therefore, the factor $O(|\text{flows}|^2)$ will be reduced to $O(|\text{conflicting flows}|^2)$. The memory usage for *Recurrence Non-Conflicts* cannot be limited as described above since flow configurations of non-conflicting flows and flow configurations using disjoint paths are non-conflicts. Nevertheless, combined with the *Potential Conflicts* approach, we can limit the *Recurrence Non-Conflicts* checks to overlapped paths and do not store unnecessary information for disjoint paths.

The *Recurrence Conflict* efficiency depends on three factors. First, the network topology has a similar impact on the efficiency as on the *Potential Conflicts* explained in Section 6.2.1. A network topology with many disjoint paths will have fewer *Recurrence Conflicts* than a topology with many

overlapping paths. In contrast, *Recurrence Non-Conflicts* will store more non-conflicting offsets for network topologies with many disjoint paths. The second is the hypercycle size of two flow configurations using intersecting paths. The larger the hypercycle, the more *Recurrence Conflicts* and *Recurrence Non-Conflicts* can potentially occur. The third is the flow configuration generation strategy. Since, in this thesis, we iterating the route-phase space as Falk et al. [FGD+22], we can expect to generate many *Recurrence Conflicts* because, for each route, the multiple phase values will be generated in an ordered way. Thereby, the phase value step size is the same and, in our use case, the 75-th percentile of the transmission duration of all flows. For example, we generate phase values for each path by iterating the interval $[0, t_{cycle} - t_{trans}]$ with step size two time units. W.l.o.g. assume, two flow configurations using the phase value 0 have a conflict. Another two flow configurations using a phase value of 2 have a *Recurrence Conflict*. With these two information we can expect that all generated flow configurations for the respective paths and flows will all have a *Recurrence Conflict*. The more flow configurations are generated for each flow, the higher the possibility of finding *Recurrence Conflicts* and *Recurrence Non-Conflicts*. Wonner describes the configuration generation approach explained above as *Bruteforce Strategy* [Won21]. In contrast, using a flow configuration generation strategy with more diverse phase values can reduce the amount of *Recurrence Conflicts* or *Recurrence Non-Conflicts*, e.g., the *DivideAndConquer Strategy* [Won21].

7 Evaluation

In this chapter, we discuss the empirical evaluations and results. In the beginning, we introduce the evaluation environment and data. Further, we will investigate the impact of the network graph topology on the resulting conflict graphs. Then the impact of our optimizations from Chapter 6 will be examined, and the computation techniques will be inspected.

7.1 Environment and Scenarios

The evaluation program is implemented with Rust version 1.59.0, and Cargo creates the release build. We conducted our evaluations on a machine with two AMD EPYC 7401 processors, each having 24 cores with two threads respectively and running under Ubuntu 20.04. Overall the machine has 96 threads, each having a cache size of 512 KB. Additionally, the machine has 256 GB of RAM available. The Rust evaluation program is executed single-threaded since the evaluation focuses on the different data structure implementations and optimization techniques.

As mentioned before, we are running scenarios to compare the different implementations of the conflict graph. A scenario consists of time steps, and in each time step, flows are added and removed. We add a configurable number of flow configurations for each flow to the conflict graph, which we denote with *cpf* (config per flow). Flow configurations are only deleted when a flow will be removed, which is part of the scenario execution. When executing GFH and a flow will not be accepted, the flow with its flow configurations stay in the system in contrast to Falk et al. [FGD+22]. We also do not implement another extension phase to extend the number of flow configurations for an existing flow. With this setup, we can control the exact number of flow configurations in the conflict graph with the *cpf* parameter. This helps us to compare different optimization approaches accurately with our optimization goal defined in Section 5.2. The following table Table 7.1 describes the different performed scenarios. We mainly use the *Dynamic Large* scenario to compare our optimizations. It includes operations to insert and delete flow configurations, as well as the execution of the GFH.

name	no. initial flows	no. add flows	no. delete flows	cpf	timesteps
Dynamic Large	100	250	100	100	10
Dynamic Small	20	50	20	100	10
Compare CPF	0	100	0	100-1300	1
Compare Flows	0	100-1300	0	100	1

network parameters: processing delay $t_{proc} = 2 \mu\text{s}$, propagation delay $t_{prop} = 1 \mu\text{s}$

Table 7.1: Scenario Parameters

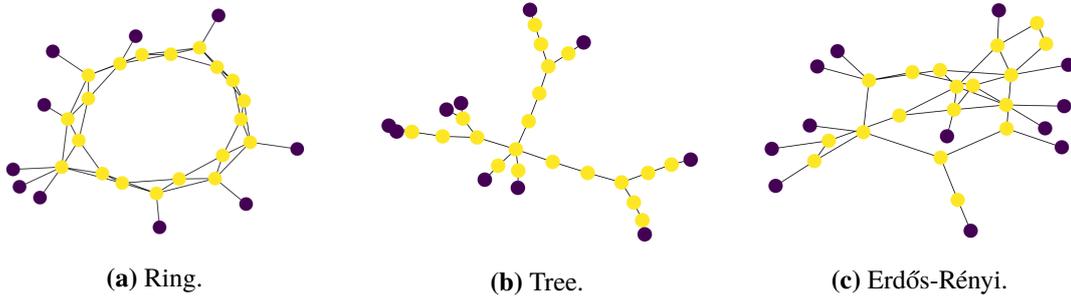


Figure 7.1: Used network graph topologies with 20 switches (yellow) and 10 end devices (purple)

For each scenario, we assume a link bandwidth of 1 GB/s and a propagation delay of 1 μ s. Each network node has a processing delay of 2 μ s. The k-shortest path algorithm computes at most three paths for each source and destination for which flows will be inserted.

We also compare the scenarios on different network topologies. Moreover, we will execute the Rust evaluation tool on a tree, ring, and Erdős-Rényi topology. Figure 7.1 shows these three different network topologies. Thereby, the purple nodes are end devices sending the packets, and the yellow nodes are network devices like IEEE 802.Qbv switches. The different properties of the network topologies also result in conflict graphs of different sizes. Additionally, a different number of paths will be generated on different topologies. For example, the tree topology only has one path that can send packets from one end device to another. In contrast, the Erdős-Rényi topology usually provides many alternative candidate paths. This behavior impacts the number of conflicts and the number of conflict computations. We discuss these differences in Section 7.2.

7.2 Conflict Graph Behavior for Different Network Topologies

We start the evaluation by analyzing the three mentioned network topologies. As described in Table 7.1, the *Dynamic Small* scenario adds 50 flows with 100 configurations per flow in each time step, resulting in adding 5000 flow configurations (vertices) per time step. Additionally, 20 flows will be deleted and, therefore, all their 2000 flow configurations. Construction and deletion operations are measured in the Rust implementation independently. For each time step, first, the configurations will be deleted, and second, the flow configurations for the requested flows will be inserted. This results in a growing conflict graph of 3000 flow configurations each time step from 2000 to 32000. All new flow configurations are generated before the conflicts (edges) are computed and inserted, which is not part of the construction phase. For the *Dynamic Large* scenario, we are adding and deleting five times more flows with the same *cpf*. Therefore the conflict graph in the *Dynamic Large* scenario grows from 10.000 up to 160.000 flow configurations with a step size of 15.000 per time step. Figure 7.2 and Figure 7.3 shows the resulting conflict graph sizes and number conflict for the *Dynamic Small* and *Dynamic Large* scenario respectively.

Especially the number of conflicts in the conflict graph is an interesting measure, and Figure 7.2c illustrates these for all three network topologies. From all theoretically possible conflicts, which would be $|V|^2$, less than 1% are indeed conflicts in the *Dynamic small* scenario for all three network topologies. In the *Dynamic Large* scenario, the tree topology has 2-8% of all theoretically conflicts

7.2 Conflict Graph Behavior for Different Network Topologies

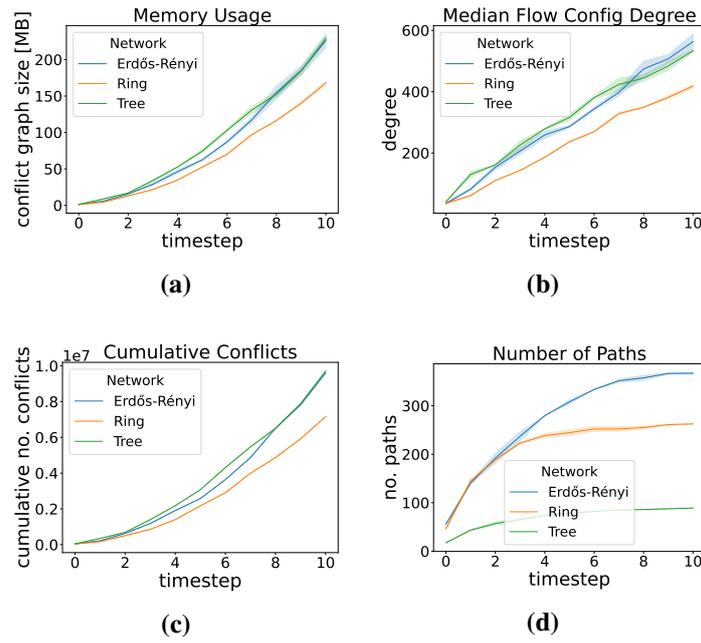


Figure 7.2: Graph behavior on the different network topologies for scenario *Dynamic Small* with 20 switches and 10 end devices.

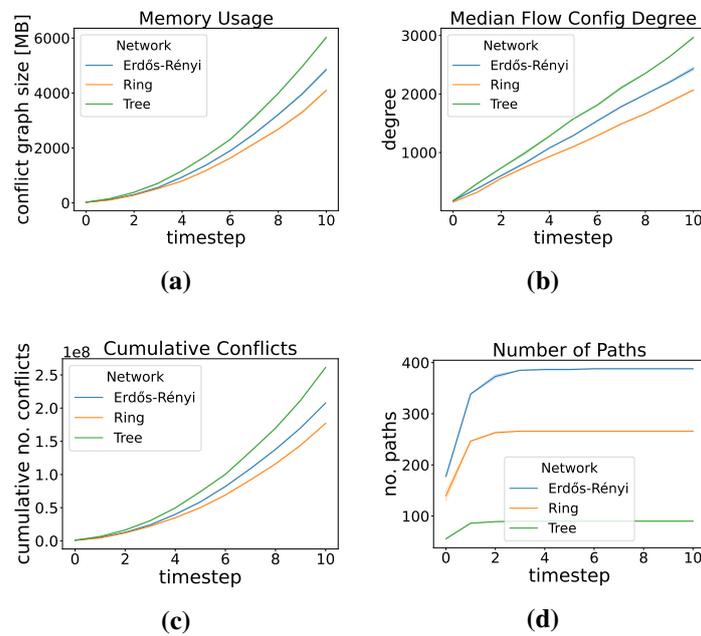


Figure 7.3: Graph behavior on the different network topologies for scenario *Dynamic Large* with 20 switches and 10 end devices.

depending on the time step and the other topologies 1-3%. Therefore, the tree topology has the most conflicts, followed by the Erdős-Rényi and ring topology. Also, the tree topology has more conflicts per flow configuration than the other two, as shown in Figure 7.2b. These two facts about the conflict graph will lead to a different construction, GFH, and deletion times. Since the tree has fewer candidate paths and only one communicating between two hosts, many flow configurations use the same sub-paths. This results in many conflicts which need more memory, as shown in Figure 7.2a. A tree mostly has a few main sub-paths where all the traffic is sent over without an alternative. This is a relevant difference between the ring and Erdős-Rényi topology. Our ring topology, one network node has four neighbor nodes, and therefore, a network node has, in most cases, three alternatives to send packets received via the fourth link. Similar to the Erdős-Rényi topology, where network nodes are randomly connected, and therefore, in most cases, multiple candidate paths are available. The number of reused path is small for ring and Erdős-Rényi compared to the tree topology. Therefore, Figure 7.2d shows that the Erdős-Rényi has the most, the ring has second-most, and the tree has the fewest candidate paths.

The amount of conflicts is, by nature, a relevant dependency on runtime performance of the three different operations (construction, GFH, deletion). Additionally, the number of paths will be a relevant factor for the indices explained in Sections 6.2.1 to 6.2.3. These optimizations depend on the number of path intersections which are more likely if more paths are in the system.

For choosing a suitable data structure, the degrees of the flow configurations are interesting. Figure 7.2b and Figure 7.3b show the median degree of all flow configurations for the different network topologies. For example, using a sorted adjacency list, the runtime performance depends on the size of the neighbor lists, which is the degree of a flow configuration. Also, for GFH, the size of each neighbor list has an impact since it iterates through the neighbor lists.

7.3 Graph Data Structure Optimization Evaluation

At the beginning of the evaluation, we analyze the measured data for the different conflict graph implementations. Thereby, we are comparing the optimization goal defined in Section 5.2 and the three components contributing to the optimization goal. Since we aim to reduce the optimization goal sum, evaluations that are faster are preferred.

7.3.1 CSR vs Adjacency List

We start the evaluation by comparing the static graph data structure CSR with adjacency lists. The adjacency list is implemented with neighbor lists as vectors, and the list contains all neighbor lists also with a vector. Since the graph size is unknown before the execution of the evaluation tool, the CSR arrays are also implemented with vectors. Nevertheless, the data is stored continuously in memory, with a reference overhead compared to an array stored on the stack.

Figure 7.4 shows the optimization goal evaluation, where we can observe, that CSR is not suitable for our conflict graph implementation. Figure 7.5 shows, that the inefficient operations of CSR are the insertion (cf. Figure 7.5a) and the flow deletion operation (cf. Figure 7.5c). Using an adjacency list significantly improves the construction phase runtime to some seconds, including the conflict computations and inserting them into the graph. In contrast, CSR needs multiple minutes in most

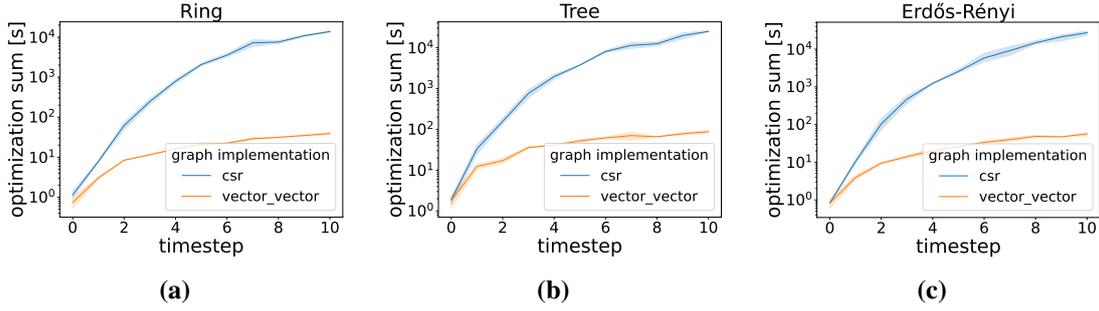


Figure 7.4: Comparing the optimization goal for CSR and the basic adjacency list implementation for the *Dynamic Small* scenario.

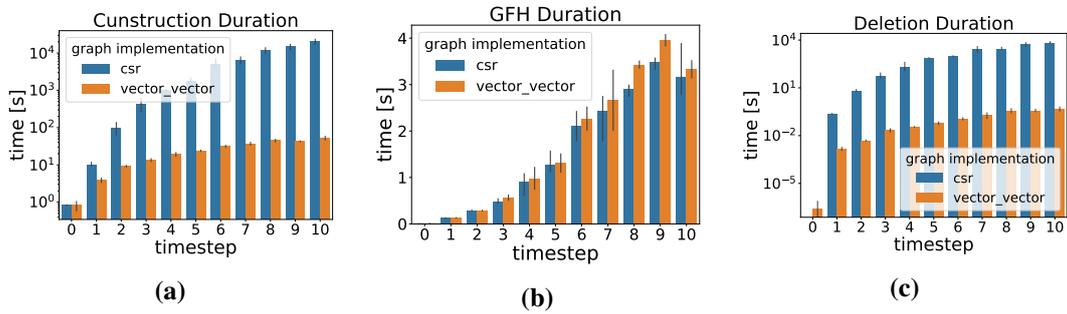


Figure 7.5: Comparing the optimization goal components for CSR and the basic adjacency list implementation for the *Dynamic Small* scenario.

time steps. Adding more flow configurations and conflicts to the conflict graph implementing CSR increases the construction phase runtime significantly compared to adjacency list. The conflict computations have only a small overhead compared to the insertion operations since the runtime for the conflict computations is the same using the adjacency list data structure. This comes from the fact that the column index array, storing the conflicts, needs more expensive shifting operations the more flow configurations are added to the conflict graph. For each added flow configuration, the number of shifting operations is the degree of this configuration in the conflict graph at the time of insertion. Since newly added configurations are appended to the CSR arrays, we also have the same number of appending operations that do not include any shifting operation. Additionally, the deletion operation comes with a high overhead for CSR. As for the insertion, the deletion operation executes many shifting operations in the column index array, which impacts the runtime performance. For each deleted flow configuration, the number of the shifting operations is two times the degree of the deleted configuration. This explains the high overhead of CSR compared to the adjacency list. The adjacency lists work with partitioned neighbor lists, which prevent expensive shifting operations. Executing a shifting operation on the CSR column index array can move $|E|$ elements in worst case, whereby shifting operation on the neighbor lists only moves $\text{degree}(v)$ with $v \in V$ elements in worst case. However, we also use optimization for the neighbor list by swapping the last element into the place of the deleted conflict in the list. We explained this operation in Section 6.1.

Taking the before described overhead improves the GFH runtime because CSR has a higher cache locality compared to adjacency lists. Figure 7.5b illustrates the improved GFH runtime performance using CSR. However, the benefit is not worth it compared to the deletion and insertion overhead. For the *Dynamic Small* scenario, the GFH executions need some seconds. Therefore, the GFH runtime does not impact the optimization goal significantly.

Overall, the optimization goal evaluation shows the significant difference between a data structure for static graph implementation (CSR) and one for a dynamic graph (adjacency list). We observe the significant impact of the construction phase. Therefore, as a result of this evaluation, we consider CSR as unsuitable.

7.3.2 Adjacency List Implementation Evaluation

After comparing CSR against adjacency lists, we will evaluate the different adjacency list implementation alternatives. As explained in Section 6.1.2, we compare the implementation with vectors and hash maps or hash sets. We compare the behavior for different implementation approaches for the adjacency list storing the neighbor lists (denoted as *storing list* in the following) and for the neighbor list itself. Thereby, we denote the implementations with the pattern $\langle \textit{storing list} \rangle _ \langle \textit{neighbor lists} \rangle$. For example, the adjacency list implementation using a vector for the *storing list* and a hash set for the neighbor lists is denoted with *vector_set*. A special case is the sorted neighbor list implementation denoted by *vector_vector_sorted*, which uses a vector for the *storing list* and the neighbor lists. As mentioned in Section 6.1.2, we do not perform a deeper analysis on ordered neighbor lists since we primarily improve the construction phase, which needs most of the time. Additional ordering operations introduce another overhead compared to the constant operations using an unsorted neighbor list. However, we include the sorted vector into the evaluation to show that there is no benefit of a sorted neighbor list for executing GFH (cf. Figure 7.7b). The sorting overhead during the construction phase is small for the *Dynamic Large* scenario comparing *vector_vector* and *vector_vector_sorted* in Figure 7.7a. The higher the degree of a neighbor list, the higher the sorting overhead when inserting or deleting conflicts from the neighbor list. The conflict graph has a median flow degree of 2000-3000 depending on the network topology after executing all time steps of the *Dynamic Large* scenario (cf. Figure 7.3b).

Figure 7.6 shows the optimization goal sum for all three network topologies as defined in Section 5.2. Since the optimization goal is a minimization, low values are preferred. We can observe that the optimization goal sum for all three network topologies is similar. The sorted vector implementation has the worst optimization goal sum. This illustrates the sorting overhead comparing *vector_vector* and *vector_vector_sorted*. Implementing the *storing list* with a hash map or vector has a similar efficiency. Therefore, Figure 7.7 shows the three different metrics contributing to the optimization goal for the Erdős-Rényi topology in detail. The other two topologies behave similarly. Figure 7.7a visualize the time to compute conflicts and insert them into the conflict graph data structure. We can observe a notable overhead difference for the implementations using a hash function. Since, at this point, we do not have any conflict computing optimizations, we can argue that the conflict insertion into a vector is faster than inserting them into a hash set. This comes from the hashing overhead. Additionally, we can see that the overhead for *vector_vector_sorted* appears especially during deletion. In Figure 7.7b the runtime executing GFH is similar for *vector_vector_sorted* and *vector_vector*.

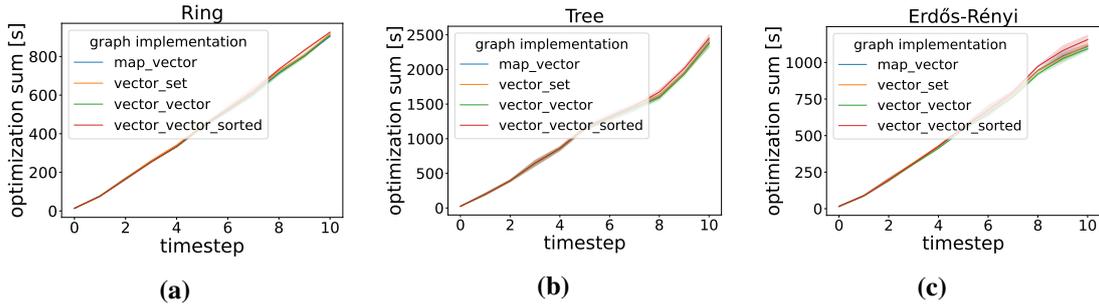


Figure 7.6: Comparing the optimization goal for the different adjacency list implementation alternatives for the *Dynamic Large* scenario.

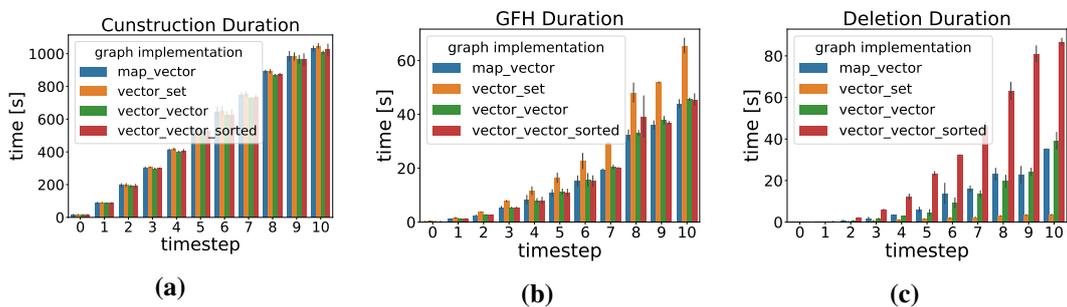


Figure 7.7: Comparing the optimization goal components for the different adjacency list implementation alternatives for the *Dynamic Large* scenario.

The efficiency executing GFH on the different implementations differs more significant for *vector_set* compared to all other implementations. Using the *adjl_set* slows down the GFH runtime by a factor of around 1.5 compared to the *adjl_vector* implementation. The slow down factor 1.5 can be explained by the nature of a hash set, that the elements are not stored side by side compactly in the memory. All elements in a hash set are stored in a memory region with possible spaces. The whole memory region must be iterated to ensure that no element is missed to iterate over all elements. Iterating through a vector is faster because there is no unused memory space between the elements in contrast to a hash set. GFH often iterates over the neighbor lists to compute the *shadow rating* [FGD+22]. This leads to the overhead, which increases, the more frequent iterations over the neighbor lists are performed. However, the overhead depends on the hash set's load factor. The higher the load factor, the less unused space is between the elements, which results in fewer empty memory hits during the iteration. Therefore, the worst case number of iterations for a hash set is $O(\text{capacity})$, and the iteration duration is proportional to the capacity. The elements in the vector are stored continuously, and the end of a vector can be identified independently of the vector's capacity. Therefore, the number of iterations through a vector is always $O(\text{no. elements})$ and the iteration duration is proportional to the length of the vector. This performance advantage can be seen in Figure 7.7b for the *vector_vector* and *vector_vector_sorted* implementation. The *map_vector* implementation has a slight runtime performance disadvantage compared to *vector_vector*, which comes from the hashing overhead when accessing a neighbor list.

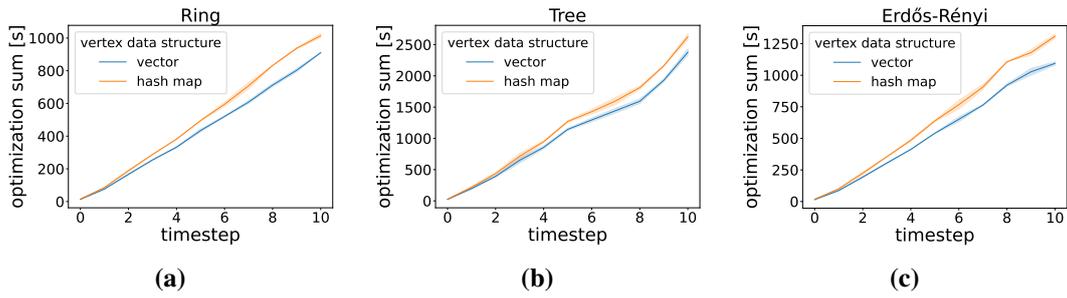


Figure 7.8: Comparing the optimization goal for different flow configuration (vertex information) implementation alternatives for the *Dynamic Large* scenario.

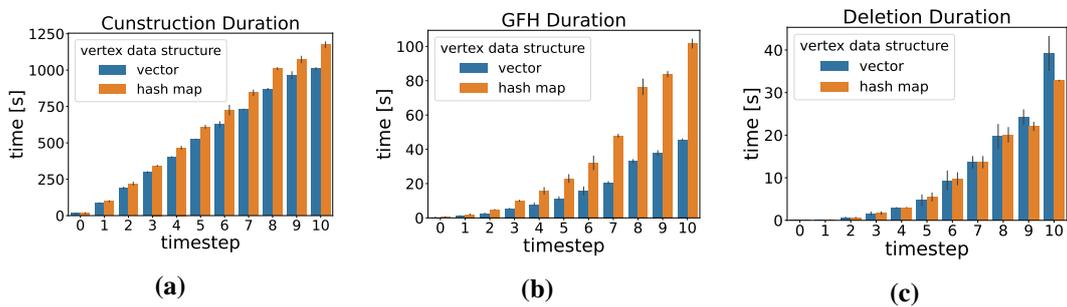


Figure 7.9: Comparing the optimization goal components for different flow configuration (vertex information) implementation alternatives for the *Dynamic Large* scenario.

From the described fact, implementing an adjacency list using a hash set based data structure for the neighbor lists comes with an iterating overhead depending on the capacity of the hash set. Therefore, one has to pay attention to the load factor of the hash set. If the load factor is too small, e.g., through removing elements, the hash set should be reallocated and the values copied into one with lower capacity when the scheduler is idle. In our implementation, we are not using such a technique, and the hash set capacities would only grow until a neighbor list is deleted. Managing a suitable load factor adds another overhead when using hash maps or hash sets. However, as mentioned above, this overhead can be outsourced to the scheduler’s idle time. Overall, using GFH for the independent set computation, a hash set implementation for the neighbor lists is not suitable due to the iteration frequency. A hash set becomes suitable when the independent set algorithm looks for specific entries in the hash set.

Overall, we can observe, that the *vector_vector_sorted* has the slowest delete performance and the *vector_set* the fastest. Naturally, finding the element to delete in a hash set is faster than a vector. A vector implementation must find the index of the element to delete by iterating the vector beforehand. This explains the overhead for the two *vector_vector* implementations. The larger the degree, the more iterations are needed on average. The *vector_vector_sorted* implementation keeps the order in the neighbor list after deleting an element which results in a high overhead compared to the other implementations.

Additionally, we are using a vector-based flow configuration data structure for all the implementations above. The flow configuration data structure stores the vertex information of the conflict graph. Figure 7.8 illustrates the performance advantage for the vector compared to a hash map implementation. The relevant overhead originates during the construction and GFH execution phase as shown in Figure 7.9a and Figure 7.9b respectively.

In summary, implementing the neighbor list and the vertex list with a vector is the most efficient solution. The *storing list* can be implemented with a vector or hash map, which both performs similarly but with the hash map having a small overhead. However, a hash map manages the memory spaces of deleted neighbor lists better than a vector. A hash set reuses the current space. In contrast, the vector only grows and must be rearranged by removing the empty spaces to reuse memory. For the GFH heuristic, only an efficient iteration data structure is suitable. Therefore, a vector is the most efficient neighbor list data structure from our evaluation. Additionally, the construction phase is very efficient as well as the deletion phase has a reasonable overhead. Sorting the neighbor lists does not show any advantages. In the remainder of the thesis, we will use the *vector_vector* implementation as the base conflict graph data structure.

We can observe that the construction phase has no significant runtime performance differences between the adjacency list implementations. From a theoretical point of view, adjacency lists have the fastest insertion operation, which is constant amortized, so we are not expecting any performance improvements for other conflict graph data structures.

7.4 Conflict Computation Evaluation

As explained in Section 6.2, we are using three different strategies to reduce the number of conflict computations to improve the construction phase runtime performance, which takes most of the time. Thereby, we will evaluate *Potential Conflicts* to reduce the number of conflict computations by omitting computations between two flow configurations that are using disjoint paths. Additionally, *Recurrence Conflicts* and *Recurrence Non-Conflicts* will be discussed. Finally, we evaluate the combination of the optimization approaches. In the following, we will discuss the evaluations with the memory runtime trade-off for the *Dynamic Large* scenario.

7.4.1 Potential Conflicts

Potential Conflicts aim to decrease conflict computations and reduce the construction phase overhead. Since *Potential Conflicts* depend on the paths in the system, we will explain the related implementation details before. The paths are implemented with vectors storing integer numbers indicating the network nodes. All paths are stored in a hash map for a fast lookup via the path ID. To find a path intersection, we are iterating over two paths, checking for the same node ID. If we find a node intersection, we also check if both paths are using the same link by checking the predecessor or successor node. This check has a complexity of $O(m * n)$ with m and n denoting the path length, respectively. Since the path length in a computer network is not significantly long, this trivial check is suitable and introduces only a tiny negligible overhead. The maximal path length we measured for all networks we are using was 13. Therefore, we are also not using an index that

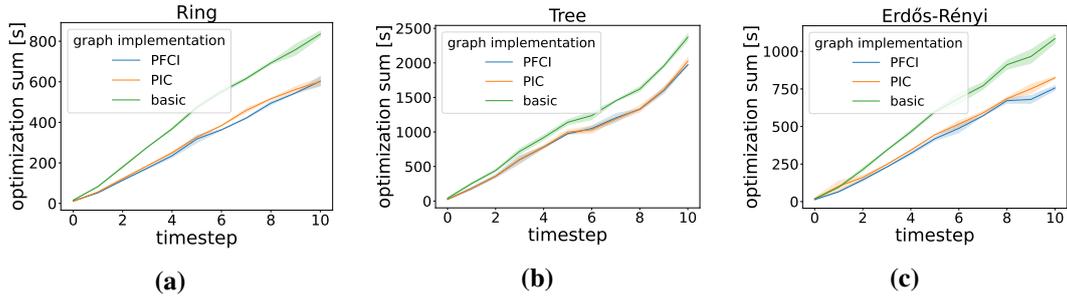


Figure 7.10: Comparing the optimization goal for *PIC*, *PFCI* and the basic adjacency list implementation for the *Dynamic Large* scenario.

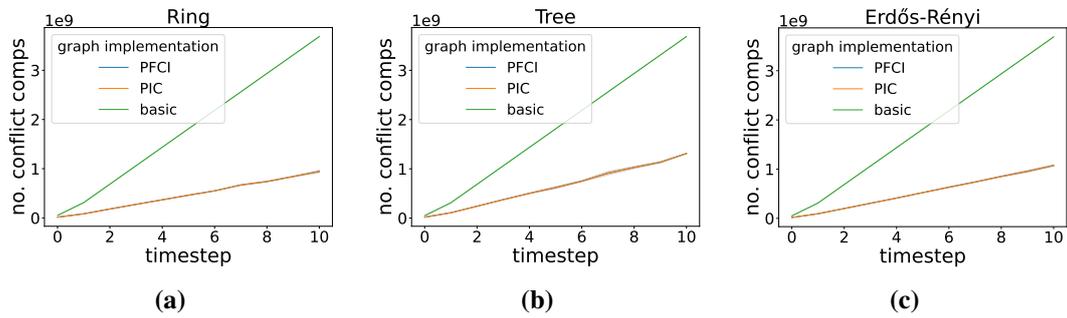


Figure 7.11: Comparing the number of conflict computations for *PIC*, *PFCI* and the basic adjacency list implementation for the *Dynamic Large* scenario.

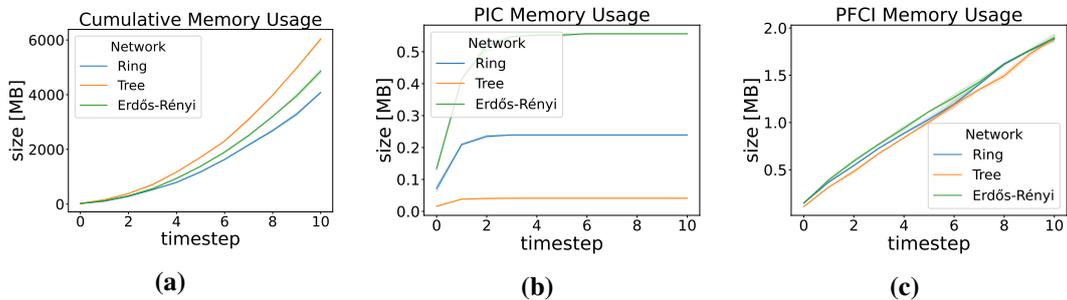


Figure 7.12: Comparing the memory usage for *PIC* and *PFCI* data structures in addition to the cumulative conflict graph size using the indices for the *Dynamic Large* scenario.

lists intersection points between two paths as explained in Section 5.1.2. This kind of optimization should be addressed when having longer paths. Paths will not be deleted from our system and can be re-used.

Path Intersection Check

The first optimization is the *Path Intersection Check (PIC)*, which stores path tuples which are intersecting each other. In Section 6.2.1 we described the data structure as a minor conflict graph storing conflicting paths. We construct the *PIC* index and add paths after the computation of the

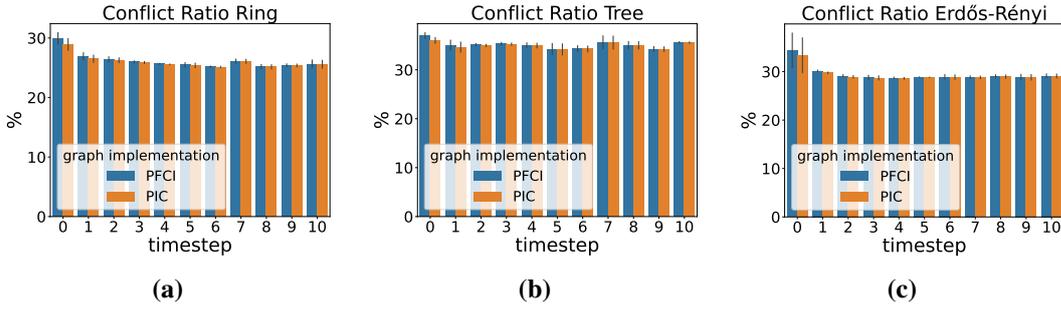


Figure 7.13: Conflict computation ratio for *PIC* and *PFCI* in proportion to the naive conflict computation approach for the *Dynamic Large* scenario.

k-shortest path algorithm. New paths are checked against each existing path for an intersection. To store the graph tuples, we are using a hash map with the path ID as key and a hash set with all conflicting path Identifiers as value. In contrast to our flow configuration conflict graph on which we perform many iteration operations during the GFH execution, on the *PIC* conflict graph, we perform many lookup operations. The construction overhead of the index depends on the number of paths in our system. Finding an intersection between two paths is performed by the trivial check explained above. However, we only want to know if there is a path intersection during the index construction. Therefore, we can stop the check after finding the first intersection point. In contrast, the conflict computation operation checks all intersection points.

Figure 7.10 displays the optimization evaluation for *PIC* and *PFCI* compared to the basic adjacency list without any optimizations. Thereby, *PFCI* implements the idea explained in Section 6.2.2 which will be discussed in Section 7.4.1 in detail. *PIC* impacts the optimization goal sum for inserting conflicts. Deletion and GFH execution is the same as the *vector_vector* implementation discussed in Section 7.3.2. Thereby, Figure 7.11 shows the significant conflict computation reduction for all three network topologies. The optimization goal sum shows a benefit for all three network topologies. *PIC* and *PFCI* have the same values since both optimizations compute the conflicts between flow configurations using intersecting paths. However, the tree topology (cf. Figure 7.10b) has only a small runtime performance increase compared to the ring and Erdős-Rényi topology illustrated in Figure 7.10a and Figure 7.10c respectively. The tree topology has only a few candidate paths which are likely to overlap. This results in many conflict computations and fewer computations that can be skipped. Figure 7.13 visualizes the conflict computations ratio for *PIC* proportional to the basic adjacency implementation. Figure 7.13b shows, that the scheduler executes 30-40% of the conflict computations with *PIC* in the tree topology. In contrast, the ring (cf. Figure 7.13a) topology performs around 20-30% and Erdős-Rényi (cf. Figure 7.13c) around 25-35% conflict computations compared to the naive approach. The ring and Erdős-Rényi topology have multiple disjoint paths. Therefore, the optimization delta between *PIC* and the basic implementation is more significant for the tree topology compared to the other two.

One great advantage of the *PIC* optimization is the memory usage. The index size depends on the computed paths in the network. Naturally, these are more paths in an Erdős-Rényi than in a tree topology as illustrated in Figure 7.2d and Figure 7.3d. Therefore, the actual memory usage depends on the number of paths and intersecting paths. Theoretically, the possible path intersections grow quadratic with the number of hosts. If each host communicates with all other hosts, the limit of paths in the system would be $O(|H|^2 * k)$ with H being the set of hosts in the network and k

the maximal number of candidate paths per flow. Practically, if multiple flows exist between two hosts, more than k paths between these hosts can exist because, for each additional flow, another alternative can be computed. We can see this behavior, e.g., in Figure 7.3d for the Erdős-Rényi topology in which we computed more than 300 paths. However, Figure 7.2d and Figure 7.3d give an intuition that each network with a limited number of paths for each flow and a constant number of hosts, the total number of paths is practically limited. The more flows are added, the more paths are computed, and therefore, all suitable paths are computed at some point. As explained before, the limit depends on the number of hosts and how many candidate paths were computed between each host. New paths can be generated between all other hosts with each added host. This is an interesting factor for dynamically changing networks. We can observe another practical limit for the tree topology in Figure 7.2d and Figure 7.3d. When there are only a limited number of alternative candidate paths, the number of computed paths is also limited. However, a suitable candidate path could also be a longer path if the end-to-end delay constraint is satisfied. We visualized the indices memory overhead for the *Dynamic Large* scenario in Figure 7.12b. We can observe the practical limit and the network topology dependency in the form of how many paths are computed. After time step 3, the index does not significantly grow for all topologies. At this point, all suitable candidate paths are computed with the limit of 3 candidate paths per flow. The tree topology has only one path alternative between two hosts. Increasing the candidate path limit per flow does not have an impact in the tree topology. Therefore, the memory usage is less compared to the other two topologies. Increasing the number of candidate paths per flow results in more computed paths in the ring and Erdős-Rényi topology, resulting in a larger *PIC* index. The index grows until the scheduler computes all suitable paths. In general, the index size correlates with the number of hosts and candidate paths comparing Figure 7.12b and Figure 7.3d. Therefore, memory usage strongly relies on the network topology.

We can also observe the independence between the scenario and the memory usage. The interesting fact is that the *PIC* index uses the same amount of memory size maximal around 0.5 Megabyte (MB) for each performed scenario with the same number of candidate paths per flow. Therefore, the only dependency that remains is the network size and the number of hosts sending packets. The *PIC* index has a small memory overhead compared to the cumulative scheduler size including the conflict graph and the *Potential Conflict* indices visualized in Figure 7.12a. In our implementation, the *PIC* index size uses less than 1% of the scheduler's needed memory.

Overall, the *PIC* optimization is suitable for a slight runtime performance increase to compute and insert new conflicts into the conflict graph with efficient memory usage. The conflict computation reduction is around 60-75%. We will use *PIC* as a basis for further optimization approaches.

Path Flow Configuration Index

The *PFCI* optimization uses the *PIC* index in combination with an index storing which flow configuration uses which path. The *PFCI* works only in combination with the *PIC* since potentially conflicting flow configurations are determined by using paths that intersect each other. Without *PIC*, we cannot determine the subset of potentially conflicting paths used by the flow configurations beforehand.

We implement the *PFCI* with a hash map containing path Identifiers as key and flow configuration Identifiers stored in vectors as value. With this information, we can look up all flow configurations that are potentially in conflict and reduce the number of iterations over the conflict graph vertices. Thereby, the reserved memory space is two-dimensional. First, the hash map size depends on the number of paths, and second, the vector sizes depend on the number of flow configurations using a particular path. Therefore, the *PFCI* size grows linearly with the number of flow configurations in the conflict graph and paths in the system. Figure 7.12c shows the *PFCI* memory consumption, illustrating the linear behavior. For the *Dynamic Large* scenario *PIC* uses approximately 25% of the memory in the last time step compared to *PFCI*. Both indices use significantly less memory than the conflict graph itself. The linear growth does not significantly affect the cumulative scheduler size since the conflict graph grows exponentially.

Figure 7.11 illustrates, that the number of conflict computations are the same as for *PIC* and therefore, *PFCI* has the same conflict computation ratio as shown in Figure 7.13 with a small delta which comes from a small difference in the implementation. However, the difference is that during a conflict computation, we are not iterating over each flow configuration contained in the conflict graph but only over the subset using the conflicting paths. In comparison, the *PIC* only skips non-potential conflicts but technically iterates overall flow configurations. The runtime performance benefit of this approach is visualized in Figure 7.10 for all network topologies. The ring and Erdős-Rényi topologies (cf. Figure 7.10a and Figure 7.10c) have a better goal optimization sum than the tree (cf. Figure 7.10b). A tree topology has more conflicts between paths as shown in Figure 7.2 and Figure 7.3. The conflict graph contains many flow configurations which are potentially in conflict. Therefore, the subset of flow configurations that must be checked for a conflict with a new flow configuration is larger than the subset in the ring or Erdős-Rényi topologies. Nevertheless, all three topologies show a runtime performance benefit compared to using only the *PIC*. However, for the tree topology, the advantage is not significant.

To summarize the evaluation of *Potential Conflicts*, we could reduce the conflict computations significantly. Thereby, with the *PIC* optimization, we skipped the computations. *PFCI* optimizes this approach by only iterating over the potential conflicting flow configurations. Overall, both indices are memory efficient, whereby the *PIC* index has a memory usage benefit by limiting the stored information depending on the network size. In contrast, the *PFCI* memory usage depends on the number of paths and flow configurations which constantly increases the more the solution space is covered. However, both indices grow less than the conflict graph itself.

7.4.2 Recurrence Conflicts and Non-Recurrence Conflicts

As explained in Section 6.2.3, the *Recurrence Conflicts* idea tries to reduce the number of conflict computations by skipping computations for reappearing conflicts that are shifted over time. To find these conflicts, we are storing the phase offsets with signed integer into a hash set when we compute a conflict, which is not known as *Recurrence Conflict*. To link the known offsets to the flow ID and path combination as explained in Section 6.2.3, we store the hash set into a hash map. The hash map key is a 4-tuple consisting of two tuples, each containing the flow ID and path ID of the conflicting flow configurations respectively, e.g., ((flow ID, path ID), (flow ID, path ID)). This key is more extensive and brings a more considerable hash function execution overhead because

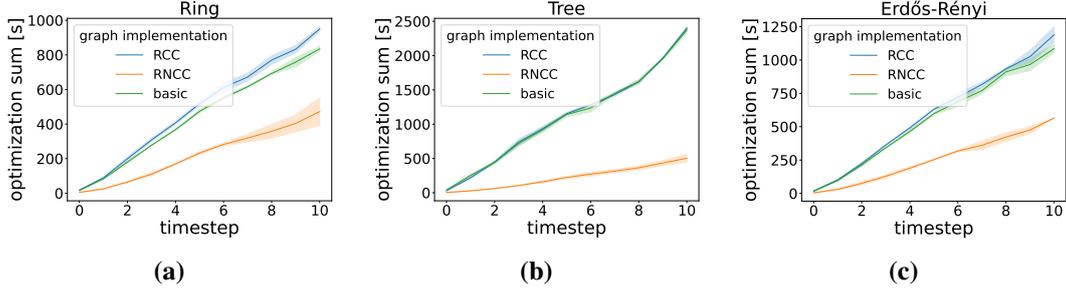


Figure 7.14: Comparing the optimization goal for *RCC*, *RNCC* and the basic adjacency list implementation for the *Dynamic Large* scenario.

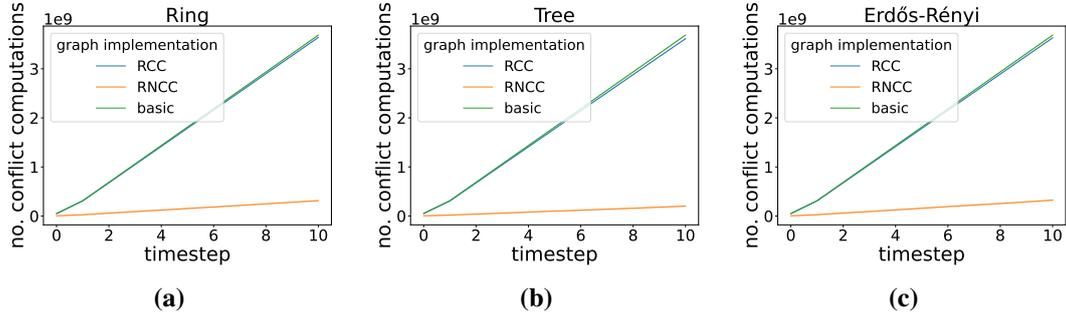


Figure 7.15: Comparing the number of conflict computations for *RCC*, *RNCC* and the basic adjacency list implementation for the *Dynamic Large* scenario.

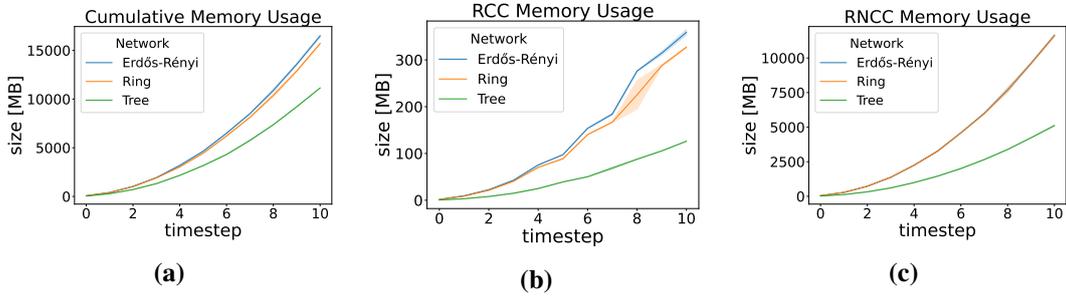


Figure 7.16: Comparing the memory usage for *RCC* and *RNCC* data structures in addition to the cumulative conflict graph size using the indices for the *Dynamic Large* scenario.

four integers are hashed. Overall, the hash map serves as a cache or scratchpad to remember which phase offsets result in *Recurrence Conflicts*. Each flow combination f_1, f_2 is only stored once by ordering the tuple key depending on the flow ID as follows:

$$\text{construct_key}(t_1, t_2) = \begin{cases} (t_1, t_2) & t_1.\text{flow_id} < t_2.\text{flow_id} \\ (t_2, t_1) & \text{otherwise} \end{cases}$$

Therefore, storing the offsets with signed integers is necessary to know the relative starting point to each other and not only the absolute offset. Before each conflict computation, we compute the phase offset of the two flow configurations and check if we have already computed a conflict with

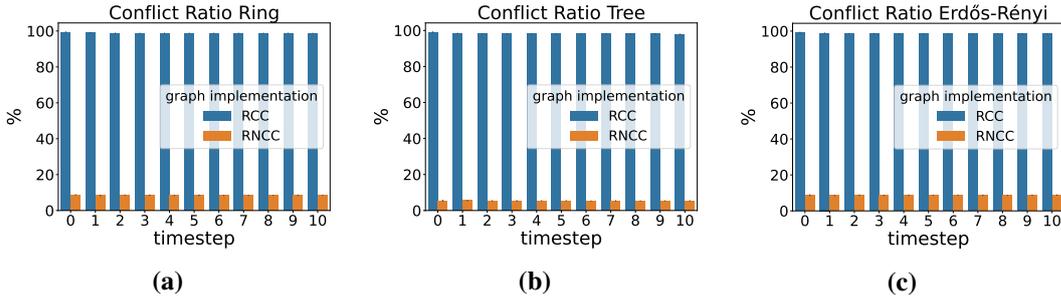


Figure 7.17: Conflict computation ratio for *RCC* and *RNCC* in proportion to the naive conflict computation approach for the *Dynamic Large* scenario.

this offset. In the first step, we check if the combination of path Identifiers and flow Identifiers is known in the hash map, and second, if the hash set contains the computed phase offset. If both conditions hold, we can skip the conflict computation because we found a *Recurrence Conflict*.

Figure 7.14 shows the optimization goal sum of the *Recurrence Conflicts Check* optimization (*RCC*) compared to the basic adjacency list implementation as discussed in Section 7.3.2 (cf. *vector_vector*). For the ring and Erdős-Rényi topology, we can observe in Figure 7.14a and Figure 7.14c respectively, that the overhead to check for *Recurrence Conflicts* and storing phase offsets is slower compared to the basic implementation. Comparing the optimization goal sum for the tree topology in Figure 7.14b shows the same runtime performance as the basic implementation. The hashing overhead is high due to the large key. Additionally, Figure 7.15 shows the conflict computation reduction for *RCC* is similar compared to the basic implementation and not many conflict computations can be skipped. Therefore, checking for *Recurrence Conflicts* provides no runtime performance benefits.

Additionally, we investigated the inverse idea, searching and caching *Recurrence Non-Conflicts* (*RNCC*). The idea is to take the same hash map but store all offsets that are not in conflict depending on the same tuple key. Preliminary evaluations have shown that conflict computations for flow configurations without a conflict take much time, as explained in Section 6.2.3. Only 1-3% for the ring and Erdős-Rényi and 2-8% for the tree topology of all possible conflicts are actually conflicts. Therefore, *RNCC* reduces the conflict computations as shown in Figure 7.15, especially the ones, which are more inefficient. Figure 7.17 visualizes the conflict computation ratio and how many conflict computations are actually performed in proportion to the naive approach. Only less than 10% of the conflict computations must be performed to compute all conflicts. *RNCC* skips many of the expensive conflict computations which compare two non-conflicting flow configurations. The significant conflict computation reduction increases the optimization goal sum for all three network topologies, as displayed in Figure 7.14. The best runtime performance is achieved for the tree topology where most of the paths overlap; therefore, many non-conflicting phase offsets can be cached.

However, this comes with vast memory usage as Figure 7.16 illustrates. The memory demand for *RNCC* is exponential. Remarkable at this point is the memory usage dimension compared to *Potential Conflict* indices. Comparing Figure 7.12 and Figure 7.16 the *Potential Conflict* indices are using at most 2 MB of memory for the *Dynamic Large* scenario which is much fewer than the *RCC* index (cf. Figure 7.16b) which is using tenth to hundreds of MB memory. The *RNCC* needs much more memory and uses more than 10 Gigabyte (GB) in the *Dynamic Large* scenario as illustrated in Figure 7.16c. Comparing the memory usage for the *RNCC* index (cf. Figure 7.16c)

with the cumulative memory usage (cf. Figure 7.16a), *RNCC* needs much more memory than the conflict graph itself. In Section 6.2.3 we explained that the memory usage from a theoretical point of view is $O(|\text{flows}|^2 * |\text{paths}|^2 * \#\text{offsets})$. The exponential growth in practice comes from a growing number of flows in the system. The more flows are in the system, the more of them can be in conflict, growing exponentially in the worst case. In practice, we can observe that the memory usage factor of conflicting paths is not growing exponentially as visualized with the *PIC* memory usage in Figure 7.12b. At some point in time, all conflicting paths are found using the k-shortest path computation approach. *RNCC* has a memory disadvantage with the memory overhead to store the phase offsets for non-conflicting flow configurations using disjoint paths. Therefore, we can observe in Figure 7.16c that the memory usage is similar for all three topologies. *RNCC* stores much unnecessary information. We tackle this problem in Section 7.4.3 by combining *RNCC* with *PIC*.

To analyze the optimization goal sum illustrated in Figure 7.14, we can observe a significant runtime performance increase for all three topologies. Thereby, the tree topology has a better improvement than ring and Erdős-Rényi. We can explain this improvement by skipping relatively more non-conflicting computations due to fewer candidate paths. The more path combination we have, the more phase offsets can be found, and therefore, more conflict computations that result in non-conflict will be performed.

To summarize, the *Recurrence Conflict* optimization is only suitable when checking for non-conflicts. However, this comes with a high memory overhead. To run the scheduler with *RNCC* has a vast storage demand which grows exponentially depending on the number of flows and network size. Therefore, *RNCC* is only suitable for small use cases and networks.

We further relax the practical memory demand and improve the runtime by combining our explained optimizations in the next section.

7.4.3 Combining Potential Conflicts and Recurrence Conflicts

The *RNCC* index memory usage storing *Recurrence Non-Conflicts* is a significant disadvantage. *RNCC* stores too much unnecessary information because we are also storing the offsets for disjoint paths. Therefore, we combine *RNCC* with *PIC* which we denote as *RNCC_PIC*. With this combination, we can skip conflict computations for non-intersecting paths which are always not conflicting with each other and additionally reduce the memory usage for the *RNCC* scratchpad. Through not storing each non-conflicting phase offset for each path-flow combination, we observe for the *Dynamic Large* scenario in Figure 7.20a and Figure 7.20c the cumulative memory usage reduction with a factor of around 4-5 for the ring and Erdős-Rényi topology. For the tree topology, visualized in Figure 7.18b, the cumulative memory usage improvement is around a factor of 2.5. This behavior and the reduction factors can also be observed for the *Dynamic Small* scenario with different *cpf*. The conflict computation reduction is visualized in Figure 7.19 for all three topologies. Figure 7.21 shows that the conflict computations ratio proportional to the basic adjacency list implementation can be halved with the *RNCC_PIC* combination compared to *RNCC*. Approximately 4% of the conflict computations for the ring and Erdős-Rényi topology and 3% for the tree topology must be performed. Additionally, the runtime performance improvement visualized in Figure 7.18 with our optimization goal improves similar to the improvement between the basic adjacency list implementation and *PIC* explained in Section 7.4.1.

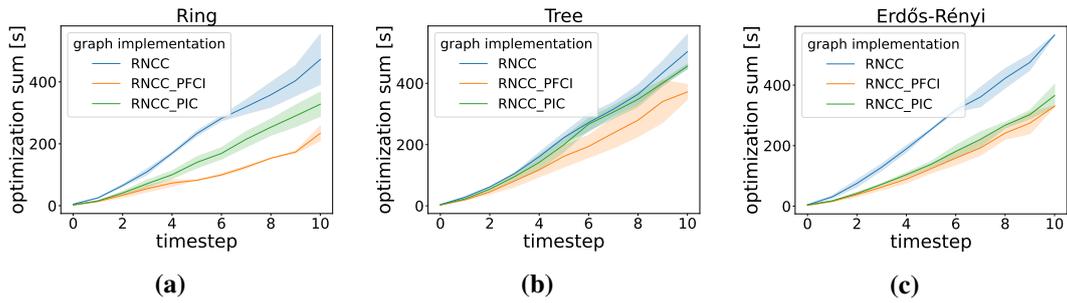


Figure 7.18: Comparing the optimization goal for *RNCC_PFCI*, *RNCC_PIC* and *RNCC* for the *Dynamic Large* scenario.

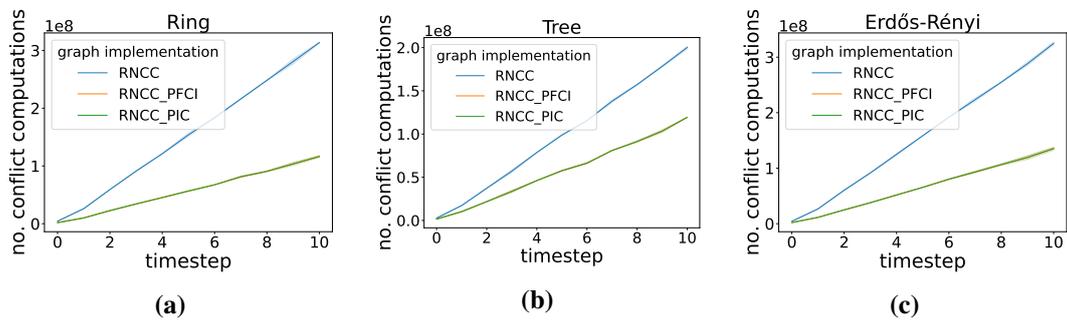


Figure 7.19: Comparing the number of conflict computations for *RNCC_PFCI*, *RNCC_PIC* and *RNCC* for the *Dynamic Large* scenario.

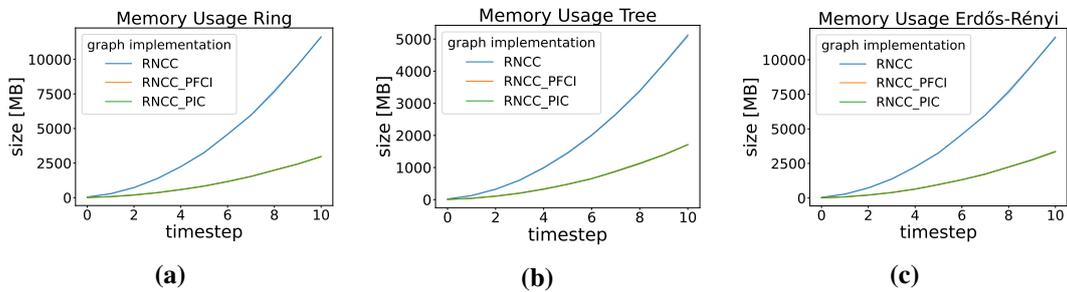


Figure 7.20: Comparing the cumulative memory usage for *RNCC_PFCI*, *RNCC_PIC* and *RNCC* for the *Dynamic Large* scenario.

Combining *PFCI* and *RNCC* denoted by *RNCC_PFCI* improves the optimization goal (cf. Figure 7.18) with the same offset as explained in Section 7.4.1. Skipping many conflict computations and only iterating potentially conflicting flow configurations results in an efficient conflict computation and insertion into the conflict graph. Overall, *RNCC_PFCI* reduces the runtime of the construction phase to double or triple the *GFH* runtime. *RNCC_PFCI* has almost the same memory usage overhead as *RNCC_PIC* because the memory overhead of *PFCI* and *PIC* are negligible compared to the cumulative memory usage (cf. Figure 7.20).

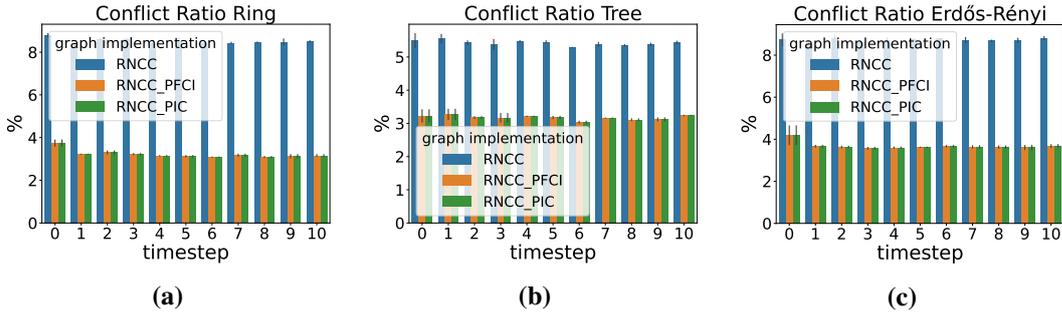


Figure 7.21: Conflict computation ratio for *RNCC_PFCI*, *RNCC_PIC* and *RNCC* in proportion to the naive conflict computation approach for the *Dynamic Large* scenario.

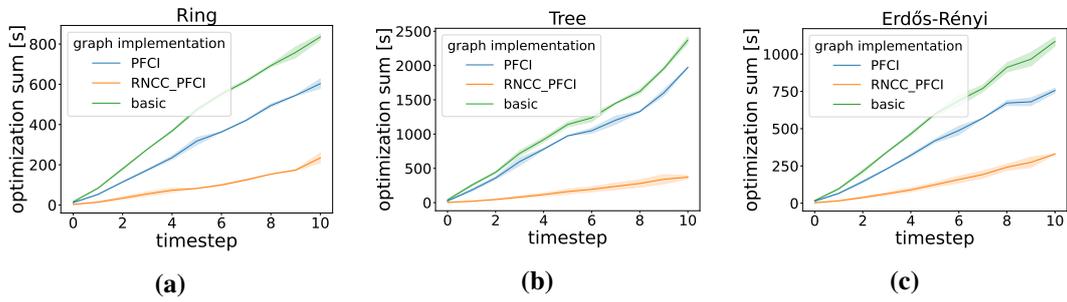


Figure 7.22: Comparing the optimization goal for *RNCC_PFCI*, *PFCI* and the basic adjacency list implementation for the *Dynamic Large* scenario.

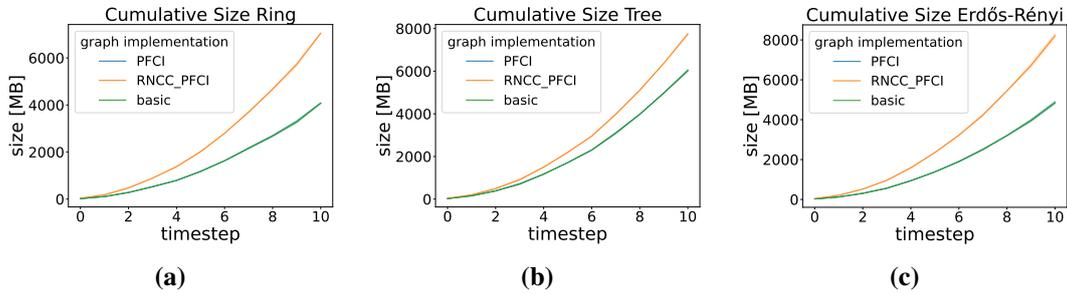


Figure 7.23: Comparing cumulative memory usage for *RNCC_PFCI*, *PFCI* and the basic adjacency list implementation for the *Dynamic Large* scenario

7.4.4 Conflict Computation Optimization Evaluation

Up to here, we evaluated three optimization approaches to reduce the conflict computations and improve the runtime performance. We evaluated the *Potential Conflict* evaluation, where *PFCI* provides the best performance improvement. Further, the *Recurrence Conflict* evaluation shows the *Recurrence Non-Conflicts* check (*RNCC*) provides the most improvement, however, with an exponential memory overhead using more memory than the conflict graph itself. Finally, we evaluated the behavior combining these approaches.

Figure 7.22 compares the optimization goal for the most efficient implementations derived from the evaluations before. We omit *RNCC* at this point, since *RNCC_PFCI* is better in all factors. It runs faster and uses less memory. However, comparing *PFCI* and *RNCC_PFCI* is a runtime-memory trade-off. Figure 7.23 shows the cumulative memory usage for the *Dynamic Large* scenario, including all used indices and the conflict graph. We can observe that *PFCI* has negligible memory overhead compared to the basic implementation, which does not use any optimization indices. The basic implementation's memory usage only consists of the conflict graph. In contrast, *RNCC_PFCI* has a more significant memory overhead compared to the basic implementation. Comparing the runtime between *RNCC_PFCI* and *PFCI* in the *Dynamic Large* scenario, shows a benefit of factor approximately 3 for the ring (cf. Figure 7.22a) and Erdős-Rényi (cf. Figure 7.22c) topologies, and a factor approximately 4 for the tree topology (cf. Figure 7.22b). We can observe that the larger the conflict graph, the better is the *RNCC_PFCI* runtime performance compared to *PFCI*. This result can also be seen in the analysis comparing different graph sizes by growing the conflict graph by inserting more flows or with a higher *cpf* value.

We analyzed a conflict graph growing in two dimensions to compare the memory usage depending on the size of the conflict graph. One growth dimension is to extend the conflict graph by adding more flow configurations for each flow, or in other words, increase the *cpf* value. The analysis is performed with the scenario *Compare CPF* constructing a conflict graph with 100 flows in one time step with *cpf* values from 100 to 1300. Another dimension is to grow the graph by adding more flows which are performed in the *Compare Flows* scenario. Thereby, we have a constant *cpf* value of 100 and construct a conflict graph from scratch with 100 to 1300 flows in one time step. Both scenarios are summarized in Table 7.1, and the resulting conflict graph has the same number of flow configurations. The results comparing the optimization goal and the memory size for the different graph sizes are displayed in Figure 7.24 and Figure 7.25 respectively for the *Compare CPF* scenario and in Figure 7.26 and Figure 7.27 for the *Compare Flows* scenario. The number of conflicts are compared in Figure 7.28.

Analyzing the different graph size scenarios, we can observe the benefit for *RNCC_PFCI* compared to *PFCI* in Figure 7.24 and Figure 7.26 for all three network topologies. Both scenarios show a constantly decreasing runtime performance for *PFCI*. The exponential decrease for the *Compare Flows* scenario comes from the significant more number of conflicts when adding more flows compared to use a higher *cpf* value (cf. Figure 7.28). This comes from our use case, which defines no conflicts between flow configurations of the same flow. In contrast, *Compare CPF* contains only 100 flows, and therefore, fewer flows can conflict in total compared to *Compare Flows*. However, an analysis comparing a constant *cpf* with a growing number of paths shows a significant runtime reduction for *RNCC_PFCI* compared to *PFCI* (cf. Figure 7.29). We can derive the fact that using fewer flow configurations on a path constantly decreases the *RNCC_PFCI* construction phase runtime performance. Figure 7.29a and Figure 7.29c shows a decreasing optimization goal sum which is mainly impacted by the construction phase. The tree topology in Figure 7.29b shows a consistent runtime performance since the limit of available candidate paths is reached with a $k = 1$ using the *k*-shortest path algorithm. With fewer flow configurations on a path, less *Recurrence Non-Conflicts* can be found.

Additionally, Figure 7.25 shows that adding more flow configurations per flow increases the memory overhead with a constant factor. In contrast, Figure 7.27 illustrates the memory growth is exponentially with a growing delta between *RNCC_PFCI* and *PFCI*. This comes from the number of

7 Evaluation

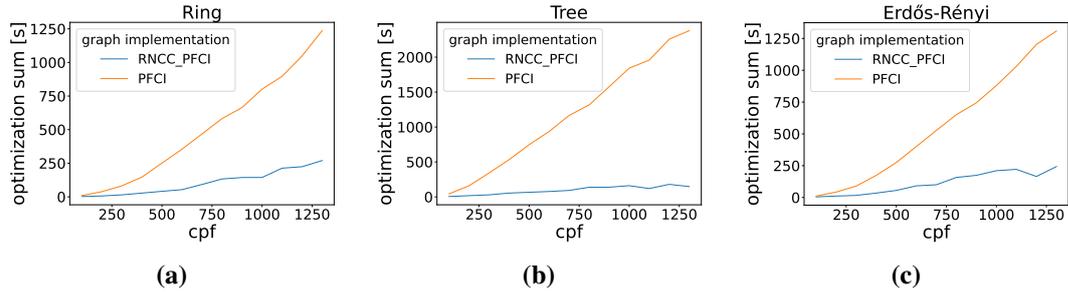


Figure 7.24: Comparing the optimization goal for *RNCC_PFCI*, *PFCI* with different *cpf* values for the *Compare CPF* scenario.

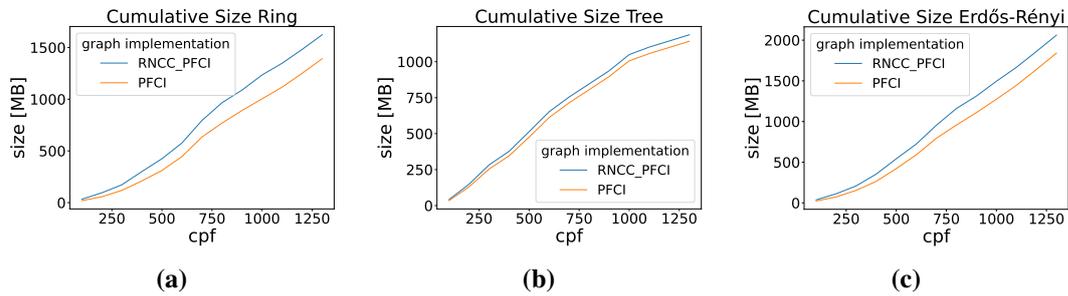


Figure 7.25: Comparing the cumulative memory usage for *RNCC_PFCI*, *PFCI* with different *cpf* values for the *Compare CPF* scenario.

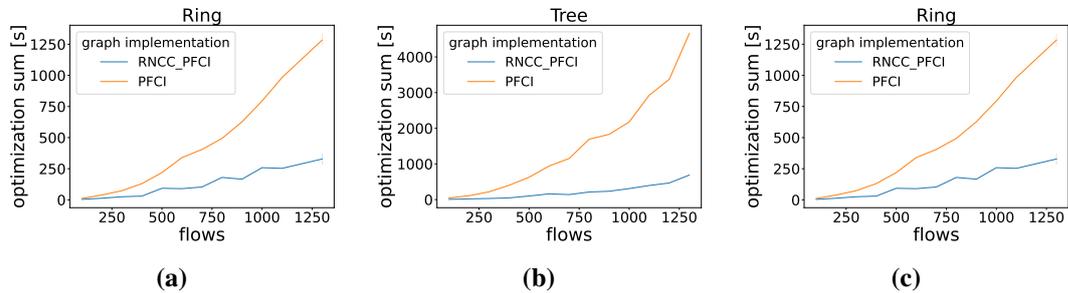


Figure 7.26: Comparing the optimization goal for *RNCC_PFCI*, *PFCI* adding different number of flows for the *Compare Flows* scenario.

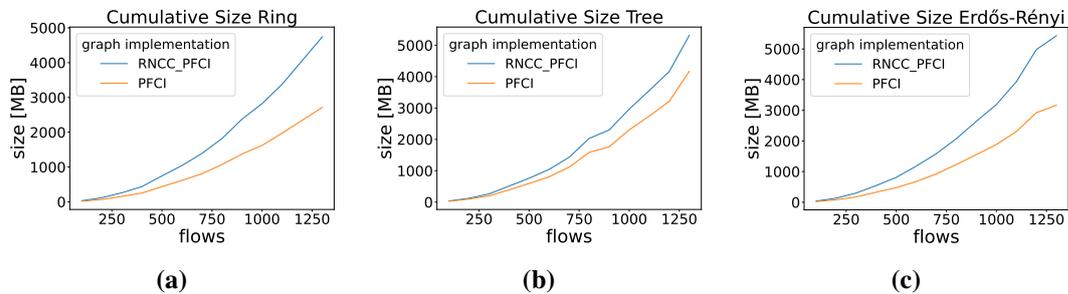


Figure 7.27: Comparing the cumulative memory usage for *RNCC_PFCI*, *PFCI* adding different number of flows for the *Compare Flows* scenario.

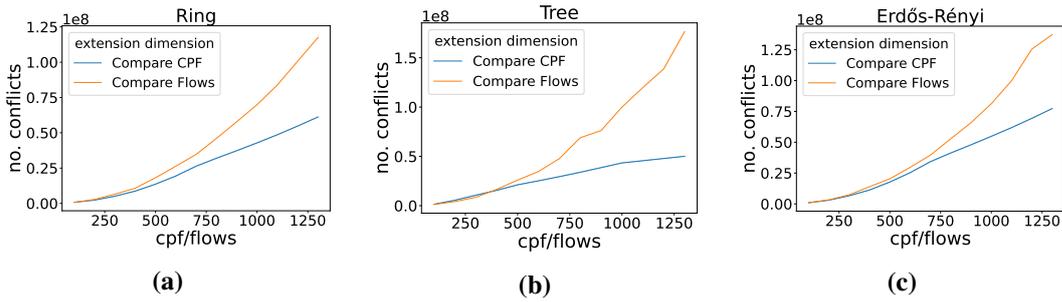


Figure 7.28: Comparing the number of conflicts for the two growing dimension scenarios *Compare CPF* and *Compare Flows*.

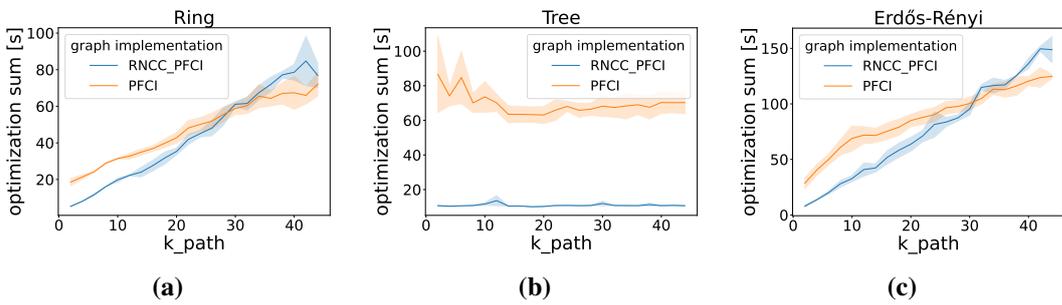


Figure 7.29: Comparing the optimization goal for *RNCC_PFCI* and *PFCI* with different number of candidate paths per flow for the *Dynamic Small* scenario.

conflicts for each scenario (cf. Figure 7.28) and the *RNCC_PFCI* hash map key, which depends on the flow ID. The more flows are in the system, the more flow ID combinations can conflict with each other on multiple paths.

To summarize the comparison between *PFCI* and *RNCC_PFCI*, we can observe the combination of *PFCI* and *RNCC* has the best performance. *PFCI* uses less memory with negligible overhead. However, *RNCC_PFCI* provides a significant runtime advantage which improves the more flow configurations are added to the conflict graph compared to *PFCI*. However, *RNCC_PFCI*'s runtime depends heavily on the number of flow configurations used by a path. Both approaches provide a significant runtime performance improvement compared to the naive basic implementation. Moreover, *RNCC_PFCI* performs on all network topologies similarly. *PFCI* shows better performance on the ring and Erdős-Rényi compared to the tree topology. We observe a runtime performance improvement in the *Dynamic Large* scenario depending on the network topology of factor 4-8 for *RNCC_PFCI* and of factor 1.2-2 for *PFCI* compared to the basic implementation.

7.5 Discussion

In the following, we are discussing the evaluation results. We will consider the limits of the evaluated approaches and discuss the scalability of our approaches applied to larger networks and scenarios.

First of all, we discussed and evaluated the adjacency list as the most promising data structure for the conflict graph applied to the scheduling problem described by Falk et al. [FGD+22]. Since the bottleneck for this use case is the conflict graph construction phase, an adjacency list provides the best runtime performance by adding conflicts and flow configurations with constant amortized operations. Data structures benefiting from cache locality improve the GFH execution runtime, which is only a fraction compared to the construction phase runtime. However, the dimension of cache locality advantage is a trade-off factor in the construction phase. Comparing CSR and adjacency lists, CSR has higher cache locality advantage in cost of a significant construction phase overhead. Further, different adjacency list implementations have advantages and disadvantages, but the best implementation for our use case was implementing all lists using vectors.

After analyzing the base conflict graph data structure, we focused on efficiently computing all conflicts. Thereby we used two approaches, namely, *Potential Conflicts* and *Recurrence Conflicts*. *Potential Conflicts* focused on ignoring non-conflicting paths and reducing the number of iterations over the flow configurations stored in the conflict graph. Furthermore, *Recurrence Conflicts* reduces the number of conflict computations noticing conflicts that are shifted over time. We recognized the runtime performance from *Recurrence Non-Conflicts* check was significantly more efficient than checking for *Recurrence Conflicts*. Therefore, the most promising optimizations are *PFCI* and *RNCC* for the two approaches, respectively. *RNCC* appeared to be much better according to our optimization goal compared to *PFCI* but uses much memory. We combined these two approaches (cf. *RNCC_PFCI*) to increase the optimization goal and reduce memory usage. *RNCC_PFCI* combines the benefits of the *Potential Conflicts* and *Recurrence Non-Conflicts* approaches but also has a considerable memory overhead. Due to the memory overhead, *RNCC* and *RNCC_PFCI* are limited to small network topologies. However, the whole conflict graph approach is limited by the memory usage since comparing the memory usage of *RNCC_PFCI* with the basic implementation shows the conflict graph needs the most memory in our scenarios. Scaling the scenarios will most likely result in using more memory than the conflict graph.

To answer the research question, our evaluation illustrates the best optimization goal for *RNCC_PFCI*. For the *Dynamic Large* scenario, our evaluation tool measured removing configurations, extending the conflict graph, and executing three GFH rounds require approximately 300 seconds for the tree, 250 seconds for the Erdős-Rényi and 200 seconds for the ring topology in the last of the ten time steps. In comparison, the naive approach needs up to 2300 seconds for the tree, up to 1200 seconds for the Erdős-Rényi, and up to 900 seconds for the ring topology. The construction phase only needs 2-3 times more time than the GFH execution. For the naive approach, the construction phase is 20-25 times slower than the GFH. The larger the conflict graph, the better *RNCC_PFCI* performs but uses exponential memory space. All network topologies used 10 end devices and 20 switches. Since our evaluation tool runs single-threaded, the runtime performance can further improved by applying multi-threading. Additionally, we performed evaluations on the three network topologies with 20 end devices and 50 switches with initial 1000 flows, adding 100 and deleting 50 flows each time step for 10 time steps. The results show the same optimization goal behavior. Thereby, the optimization goal sum of *RNCC_PFCI* for the ring and Erdős-Rényi topologies are less than 1400 seconds and for the tree around 3000 seconds. The *RNCC_PFCI* memory usage for all three topologies was from 40 to 50 GB. Moreover, we also performed analysis on networks with 260 end devices and 300 switches where our evaluation tool ran out of the 250 GB available memory.

Considering our evaluation results, our optimization approaches are suitable for small networks with a limited number of flows. Both optimization approaches should be combined to compute all conflicts efficiently. The outcome of our evaluation is a significant construction phase efficiency improvement.

8 Conclusion and Outlook

In the scope of this thesis, we conducted research on data structures and optimization approaches to efficiently construct a conflict graph based on the research previously done by Falk et al. [FGD+22][FDR20]. We formulated the research problem to find an efficient approach and data structure for performing the construction, deletion, and GFH operations on the conflict graph as soon as possible. Thereby, we focused on analyzing how to implement an adjacency list tailored for this scenario efficiently. Due to the vector's iteration behavior in GFH and the insertion performance by pushing elements into the vector, an implementation with all lists as vectors provides the best performance. Using a data structure with an even higher cache locality has a performance increase for the GFH execution, but the overhead for the construction phase is significant. For that, we analyzed CSR. There are more efficient implementations for CSR as we discussed in Chapter 3. However, we focused on improving the conflict graph construction phase, which takes the most time.

Further, we applied optimization approaches to compute all conflicts efficiently. Thereby, two promising approaches are *Potential Conflicts* and *Recurrence Conflicts*. For the *Potential Conflicts* approach, we analyzed *PIC*, which only computes conflicts between flow configurations that use overlapping paths. To enhance *PIC*, we implemented *PFCI*, which only iterates over the flow configurations which use these overlapping paths. Both optimizations need a hash map to store the meta information. Our analysis has shown that the additional memory overhead is negligible. Moreover, *PFCI* has a better runtime performance than *PIC*. However, both optimizations perform better on a network topology with many disjoint paths.

Additionally, the *Recurrence Conflicts* optimization stores meta information on conflicting phase offsets between flow configurations using the same path and flow combination. In our analysis, we observed the optimization becomes efficient when storing non-conflicting phase offsets and, therefore, checking for *Recurrence Non-Conflicts*. The *Recurrence Non-Conflicts* checks recognize many non-conflicting flow configurations, and therefore, many conflict computations can be skipped. This idea shows a vast performance improvement at a significant memory overhead cost. We combined the *Potential Conflicts* idea with *Recurrence Non-Conflicts* checks, which increases the construction phase runtime performance and decreases the memory usage. According to our analysis, the combination is the most efficient overall presented optimizations and has a better runtime performance the larger the conflict graph becomes. The optimization combination has an exponential memory demand, but the conflict graph requires even more memory for our analyzed scenarios.

Outlook

Future work can focus on implementing a multi-threaded solution using the suggested optimization approaches. Additionally, the *RNCC_PFCI* hash map key is implemented as a tuple of two tuples which both include a combination of flow ID and path ID. We showed that depending on the network topology and path computation strategy, the number of paths in the network is limited. A growing number of flow Identifiers causes exponential memory usage. Implementing the key as a combination of packet sizes, period, and path ID could limit the exponential memory demand. Preliminary evaluations showed a more extensive memory usage. However, we did not analyze the key in detail. Another approach to limit the memory usage of *RNCC_PFCI* can be achieved by not storing all phase offsets but offset intervals.

Another interesting topic for future work is handling network failures. Thereby, a solution could use the *PFCI* index in combination with an index storing meta information linking network nodes with their used paths. When a network node fails, the new index efficiently finds all affected paths, and the *PFCI* index efficiently finds all affected flow configurations with the path information. These flow configurations can then be marked as temporarily invalid, and if needed, new paths and flow configurations can be computed. If a network node becomes available again, the affected flow configurations can be marked as valid in the same manner as described before. To recognize a reappearance of a network node needs a data structure that stores the known failed devices.

A final interest aspect is researching more extensive dynamic networks to satisfy industrial needs. One main problem is the conflict graph memory demand. Thereby, interesting topics to solve this are how to limit the needed information, e.g., by finding an approach to only store relevant parts of the solution space. This can depend on domain knowledge. Therefore, research on a specific industrial use case could also be interesting to find approaches to use domain-specific knowledge, e.g., as already conducted by Wonner [Won21] in the railroad scheduling domain. Additionally, a distributed solution of the conflict graph can be interesting in solving the memory problem.

Bibliography

- [AAPO20] M. A. Awad, S. Ashkiani, S. D. Porumbescu, J. D. Owens. “Dynamic Graphs on the GPU”. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ISSN: 1530-2075. May 2020, pp. 739–748. DOI: [10.1109/IPDPS47924.2020.00081](https://doi.org/10.1109/IPDPS47924.2020.00081) (cit. on p. 25).
- [ANS00] A. Atamturk, G. Nemhauser, M. Savelsbergh. “Conflict graphs in solving integer programming problems”. In: *European Journal of Operational Research* 121 (Feb. 2000), pp. 40–55. DOI: [10.1016/S0377-2217\(99\)00015-6](https://doi.org/10.1016/S0377-2217(99)00015-6) (cit. on p. 23).
- [BKP01] R. Bapat, S. Kirkland, S. Pati. “The perturbed Laplacian matrix of a graph”. In: *Linear and Multilinear Algebra* 49 (Dec. 2001), pp. 219–242. DOI: [10.1080/03081080108818697](https://doi.org/10.1080/03081080108818697) (cit. on p. 19).
- [BS21] S. S. Brito, H. G. Santos. “Preprocessing and cutting planes with conflict graphs”. en. In: *Computers & Operations Research* 128 (Apr. 2021), p. 105176. ISSN: 0305-0548. DOI: [10.1016/j.cor.2020.105176](https://doi.org/10.1016/j.cor.2020.105176). URL: <https://www.sciencedirect.com/science/article/pii/S0305054820302938> (visited on 01/28/2022) (cit. on p. 23).
- [CFV21] M. E. Coimbra, A. P. Francisco, L. Veiga. “An analysis of the graph processing landscape”. In: *Journal of Big Data* 8.1 (Apr. 2021), p. 55. ISSN: 2196-1115. DOI: [10.1186/s40537-021-00443-9](https://doi.org/10.1186/s40537-021-00443-9). URL: <https://doi.org/10.1186/s40537-021-00443-9> (visited on 01/31/2022) (cit. on pp. 17, 18).
- [Cha04] G. Chaitin. “Register allocation and spilling via graph coloring”. In: *ACM SIGPLAN Notices* 39.4 (Apr. 2004), pp. 66–74. ISSN: 0362-1340. DOI: [10.1145/989393.989403](https://doi.org/10.1145/989393.989403). URL: <https://doi.org/10.1145/989393.989403> (visited on 01/25/2022) (cit. on p. 24).
- [DBS17] L. Dhulipala, G. Blelloch, J. Shun. “Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing”. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’17. New York, NY, USA: Association for Computing Machinery, July 2017, pp. 293–304. ISBN: 978-1-4503-4593-4. DOI: [10.1145/3087556.3087580](https://doi.org/10.1145/3087556.3087580). URL: <https://doi.org/10.1145/3087556.3087580> (visited on 12/15/2021) (cit. on p. 25).
- [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*. RTNS ’16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 203–212. ISBN: 978-1-4503-4787-7. DOI: [10.1145/2997465.2997494](https://doi.org/10.1145/2997465.2997494). URL: <https://doi.org/10.1145/2997465.2997494> (visited on 03/14/2022) (cit. on pp. 13, 27).

- [DN19] D. Durner, T. Neumann. “No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System”. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. ISSN: 2375-026X. Apr. 2019, pp. 734–745. doi: [10.1109/ICDE.2019.00071](https://doi.org/10.1109/ICDE.2019.00071) (cit. on pp. 24, 25).
- [DPH07] A. D’Ariano, M. Pranzo, I. A. Hansen. “Conflict Resolution and Train Speed Coordination for Solving Real-Time Timetable Perturbations”. In: *IEEE Transactions on Intelligent Transportation Systems* 8.2 (June 2007). Conference Name: IEEE Transactions on Intelligent Transportation Systems, pp. 208–222. issn: 1558-0016. doi: [10.1109/TITS.2006.888605](https://doi.org/10.1109/TITS.2006.888605) (cit. on p. 24).
- [DPST18] P. Danielis, H. Puttnies, E. Schweissguth, D. Timmermann. “Real-Time Capable Internet Technologies for Wired Communication in the Industrial IoT—a Survey”. In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. ISSN: 1946-0759. Sept. 2018, pp. 266–273. doi: [10.1109/ETFA.2018.8502528](https://doi.org/10.1109/ETFA.2018.8502528) (cit. on pp. 13, 14).
- [ERBM11] D. Ediger, J. Riedy, D. Bader, H. Meyerhenke. “Tracking Structure of Streaming Social Networks”. In: June 2011, pp. 1691–1699. doi: [10.1109/IPDPS.2011.326](https://doi.org/10.1109/IPDPS.2011.326) (cit. on p. 25).
- [FD20] S. Firmli, C. Dalila. “A Review of Engines for Graph Storage and Mutations”. In: Jan. 2020, pp. 214–223. ISBN: 978-3-030-36777-0. doi: [10.1007/978-3-030-36778-7_23](https://doi.org/10.1007/978-3-030-36778-7_23) (cit. on pp. 18, 25).
- [FDR20] J. Falk, F. Dürr, K. Rothermel. “Time-Triggered Traffic Planning for Data Networks with Conflict Graphs”. In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. ISSN: 2642-7346. Apr. 2020, pp. 124–136. doi: [10.1109/RTAS48715.2020.00-12](https://doi.org/10.1109/RTAS48715.2020.00-12) (cit. on pp. 14, 23, 29, 30, 65).
- [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). Conference Name: IEEE Transactions on Network and Service Management, pp. 1–1. ISSN: 1932-4537. doi: [10.1109/TNSM.2022.3150664](https://doi.org/10.1109/TNSM.2022.3150664) (cit. on pp. 13, 14, 23, 27–30, 32, 39, 41, 47, 62, 65).
- [FTL+20] S. Firmli, V. Trigonakis, J.-P. Lozi, I. Psaroudakis, A. Weld, C. Dalila, S. Hong, H. Chafi. “CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure”. In: Dec. 2020. doi: [10.4230/LIPIcs.OPODIS.2020.17](https://doi.org/10.4230/LIPIcs.OPODIS.2020.17) (cit. on pp. 21, 25).
- [GHR+20] M. Gundall, C. Huber, P. Rost, R. Halfmann, H. D. Schotten. “Integration of 5G with TSN as Prerequisite for a Highly Flexible Future Industrial Automation: Time Synchronization based on IEEE 802.1AS”. In: *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*. ISSN: 2577-1647. Oct. 2020, pp. 3823–3830. doi: [10.1109/IECON43393.2020.9254296](https://doi.org/10.1109/IECON43393.2020.9254296) (cit. on p. 14).
- [HMKS19] T. Häckel, P. Meyer, F. Korf, T. C. Schmidt. “Software-Defined Networks Supporting Time-Sensitive In-Vehicular Communication”. In: *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)* (Apr. 2019). arXiv: 1903.08039, pp. 1–5. doi: [10.1109/VTCSpring.2019.8746473](https://doi.org/10.1109/VTCSpring.2019.8746473). URL: <http://arxiv.org/abs/1903.08039> (visited on 04/19/2022) (cit. on p. 13).
- [IEE] IEEE. *IEC/IEEE 60802 TSN Profile for Industrial Automation* |. URL: <https://1.ieee802.org/tsn/iec-ieee-60802/> (visited on 02/03/2022) (cit. on p. 27).

- [KUK16] KUKA. *Hello industrie 4.0 _we go digital*. 2016 (cit. on p. 13).
- [MB09] K. Madduri, D. A. Bader. “Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis”. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. ISSN: 1530-2075. May 2009, pp. 1–11. DOI: [10.1109/IPDPS.2009.5161060](https://doi.org/10.1109/IPDPS.2009.5161060) (cit. on p. 25).
- [MMMS15] P. Macko, V.J. Marathe, D. W. Margo, M. I. Seltzer. “LLAMA: Efficient graph analytics using Large Multiversed Arrays”. In: *2015 IEEE 31st International Conference on Data Engineering*. ISSN: 2375-026X. Apr. 2015, pp. 363–374. DOI: [10.1109/ICDE.2015.7113298](https://doi.org/10.1109/ICDE.2015.7113298) (cit. on pp. 21, 25).
- [NCC17] T. D. Nguyen, M. Chiesa, M. Canini. “Decentralized Consistent Updates in SDN”. In: *Proceedings of the Symposium on SDN Research*. SOSR ’17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 21–33. ISBN: 978-1-4503-4947-5. DOI: [10.1145/3050220.3050224](https://doi.org/10.1145/3050220.3050224). URL: <https://doi.org/10.1145/3050220.3050224> (visited on 04/08/2022) (cit. on p. 28).
- [Net22] N. Nethercote. *Hashing - The Rust Performance Book*. Apr. 2022. URL: <https://nnethercote.github.io/perf-book/hashing.html> (visited on 04/04/2022) (cit. on p. 35).
- [Old19] M. Olding. *Wie TSN die Sicherheit in der Fabrik erhöhen kann*. de. Feb. 2019. URL: <https://www.exorint.com/de/blog/wie-tsn-die-sicherheit-in-der-fabrik-erh%C3%B6hen-kann> (visited on 04/19/2022) (cit. on p. 13).
- [PPP02] J.-S. Park, M. Penner, V. Prasanna. “Optimizing graph algorithms for improved cache performance”. In: Feb. 2002, pp. 32–41. ISBN: 978-0-7695-1573-1. DOI: [10.1109/IPDPS.2002.1015509](https://doi.org/10.1109/IPDPS.2002.1015509) (cit. on p. 25).
- [Pro21] F.-I. für Produktionstechnologie IPT. *5G-Technologie für zuverlässigere Echtzeit-Kommunikation zwischen Maschinen, Anlagen und Cloud-Systemen dank Time Sensitive Networking - Fraunhofer IPT*. de. Feb. 2021. URL: <https://www.ipt.fraunhofer.de/de/presse/Pressemitteilungen/210211-5g-technologie-fuer-zuverlaessigere-echtzeit-kommunikation-dank-time-sensitive-networking.html> (visited on 04/19/2022) (cit. on p. 13).
- [RPGS17] M. L. Raagaard, P. Pop, M. Gutiérrez, W. Steiner. “Runtime reconfiguration of time-sensitive networking (TSN) schedules for Fog Computing”. In: *2017 IEEE Fog World Congress (FWC)*. Oct. 2017, pp. 1–6. DOI: [10.1109/FWC.2017.8368523](https://doi.org/10.1109/FWC.2017.8368523) (cit. on p. 14).
- [SDT+17] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjeglą, G. Mühl. “ILP-based joint routing and scheduling for time-triggered networks”. In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. RTNS ’17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 8–17. ISBN: 978-1-4503-5286-4. DOI: [10.1145/3139258.3139289](https://doi.org/10.1145/3139258.3139289). URL: <https://doi.org/10.1145/3139258.3139289> (visited on 03/25/2022) (cit. on p. 29).
- [Soc18] I. C. Society. “IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks”. In: *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)* (July 2018). Conference Name: IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014), pp. 1–1993. DOI: [10.1109/IEEESTD.2018.8403927](https://doi.org/10.1109/IEEESTD.2018.8403927) (cit. on p. 13).

- [TV17] M. V. Tum, M. Valiyev. *Graph Storage : How good is CSR really ?* en. 2017. URL: <https://www.semanticscholar.org/paper/Graph-Storage-%3A-How-good-is-CSR-really-Tum-Valiyev/57abed66216252569de8c7227bdb6e0aaf6e9647> (visited on 01/25/2022) (cit. on pp. 18, 21, 34).
- [Won21] S. Wonner. “Efficient and complete conflict graph generation schemes for railroad scheduling”. en. In: (2021). Accepted: 2022-03-15T16:29:36Z ISBN: 9781797862491. DOI: 10.18419/opus-12025. URL: <http://elib.uni-stuttgart.de/handle/11682/12042> (visited on 05/02/2022) (cit. on pp. 24, 25, 30, 36, 39, 66).
- [Wun21] F. Wunderlich-Pfeiffer. *Automatisierung: Brand in Lagerhaus nach Roboterkollision - Golem.de*. July 2021. URL: <https://www.golem.de/news/automatisierung-brand-in-lagerhaus-nach-roboterkollision-2107-158254.html> (visited on 04/19/2022) (cit. on p. 13).
- [WX18] B. Wheatman, H. Xu. “Packed Compressed Sparse Row: A Dynamic Graph Representation”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. ISSN: 2377-6943. Sept. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547566 (cit. on p. 25).
- [Yen71] J. Y. Yen. “Finding the K Shortest Loopless Paths in a Network”. In: *Management Science* 17.11 (1971). Publisher: INFORMS, pp. 712–716. ISSN: 0025-1909. URL: <https://www.jstor.org/stable/2629312> (visited on 01/31/2022) (cit. on p. 30).
- [Zve17] Zvei. *Industrie 4.0 Plug-and-Produce for Adaptable Factories*. de. June 2017. URL: <https://www.plattform-i40.de/IP/Redaktion/DE/Downloads/Publikation/Industrie-40-20Plug-and-Produce.html> (visited on 04/20/2022) (cit. on p. 14).
- [ZZW+] X. Zhou, Z. Zhang, G. Wang, X. Yu, B. Y. Zhao, H. Zheng. “Practical Conflict Graphs for Dynamic Spectrum Distribution”. en. In: (), p. 12 (cit. on pp. 18, 24).

All links were last followed on May 08, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature