Institute for Parallel and Distributed Systems

Bachelorarbeit

Independent Colorful Vertex Sets in large dynamic Vertex Colored Conflict Graphs

Sauter Simon

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Kurt Rothermel

Supervisor: Heiko Geppert

Commenced: 21. October 2021

Completed: 21. April 2022

Abstract

Time-Sensitive Networks are a substantial part of the Industrial Internet of Things. Extensive computation is required to generate sufficient schedules, in order to allow for real-time communication within them. This is due to a time-constraint that each communication has. These communications can vary from various logistical operations to heavy machinery. As such, not meeting these time-constraints can incur financial costs, or even human harm. Most existing solutions for this problem are ill-suited for the dynamic scenario which often appears in real-life. Modern factories are rarely static, since changes to devices, and therefore their communications, occur as new ones are added, or old ones are modified.

We built upon a previous approach, using a vertex colored conflict graph model and searching for an independent colorful set, to solve the time-triggered flow scheduling problem. Specifically, we introduce two new algorithms that dynamically generate an independent colorful set, and do so in a fraction of the time. We compare our algorithms to the original one, concluding that a combined approach might lead to the best outcome in terms of runtime and resulting scheduling quality.

Kurzfassung

Time-Sensitive Networks sind ein wesentlicher Bestandteil des Industrial Internet of Things. Umfangreiche Berechnungen sind erforderlich, um Zeitpläne zu generieren, die eine Kommunikation darin ermöglichen. Dies liegt daran, dass jede Kommunikation eine Zeitbeschränkung hat. Diese Kommunikation kann von verschiedenen logistischen Operationen bis hin zu Schwermaschinen variieren. Daher kann die Nichteinhaltung dieser Zeitbeschränkungen zu finanziellen Kosten oder sogar zu Personenschäden führen. Die meisten bestehenden Lösungen für dieses Problem sind für das dynamische Szenario, das in der Realität üblich ist, schlecht geeignet. Moderne Fabriken sind selten statisch, da sich die Geräte, die miteinander kommunizieren müssen, ändern, wenn neue hinzugefügt oder alte modifiziert werden.

Wir haben auf einem existierenden Ansatz aufgebaut, der ein gefärbtes Konfliktdiagrammmodell verwendet und nach einem unabhängigen farbigen Menge gesucht haben, um das zeitgesteuerte Flussplanungsproblem zu lösen. Insbesondere führen wir zwei neue Algorithmen ein, die dynamisch ein unabhängiges buntes Set erzeugen, und zwar in einem Bruchteil der Zeit. Wir vergleichen unsere Algorithmen mit dem Original und kommen zu dem Schluss, dass ein kombinierter Ansatz zu den besten Ergebnissen in Bezug auf Laufzeit und resultierende Scheduling-Qualität führen könnte.

Contents

1 Introduction **11**

2 Preliminaries and Problem Statement **13**
2.1 Vertex Colored Graphs 13
2.2 Independent Colorful Vertex Set 13
2.3 Problem Statement 14

3 Related Work **15**

4 Application Domain **17**
4.1 Greedy Flow Heap 18

5 Dynamic Independent Colorful Set **21**
5.1 DynGFH 21
5.2 WRandomGFH 23

6 Evaluation **27**
6.1 Evaluating DynGFH Parameters 27
6.2 Evaluation of WRandomGFH Parameters 28
6.3 Evaluation against the state of the art 30

7 Conclusion **33**
7.1 Future Work 33

Bibliography **35**

List of Figures

6.1 Results of comparing Recalculating vs Caching. Compared configurations start with 200 colors. The left diagram depicts the time each method took in ms. The caching-method has a slightly increased runtime. The left diagram shows, that, as expected, both methods result in similar ICSS. 28
6.2 Results of comparing different amounts of repeats. The left chart illustrates the different execution times. As expected, the higher the number of repeats, the longer the execution time. The left chart displays how many colors ended up in the resulting ICSS each time step. The parameters we used lead to no major differences between the different amounts of repeats. 29
6.3 Results of comparing different ways of selecting a vertex. The left diagram indicates a higher runtime for the degree-weighted algorithm and very similar runtimes for the random and random-below-average ones. In the left diagram, again, a similar result for all of them is expected. 30
6.4 A more detailed comparison of the different vertex-selection algorithms, in later time steps. As expected, the degree-weighted algorithm, on average, ends up with the highest amount of colors in the ICSS. Followed by random-below-average and random. 30
6.5 Results of GFH, DynGFH and WRandomGFH, when run on a scenario with 100 initial flows. 31
6.6 Results of GFH, DynGFH and WRandomGFH, when run on a scenario with 200 initial flows. 31
6.7 Results of GFH, DynGFH and WRandomGFH, when run on a scenario with 500 initial flows. 32

List of Algorithms

4.1 GFH: main algorithm loop 19
5.1 DynGFH: main algorithm loop 22
5.2 WRandomGFH: main algorithm loop 24
5.3 Weighted Random Color 25

1 Introduction

The industrial internet of things (IIOT) is a modern concept, where industrial devices are all interconnected [BHCW18]. Examples are production lines in the automobile industry, medical equipment in a hospital and many more. A Jeep production facility, for example, has conveyor belts, security sensors and robot arms in various configurations to enable the part by part construction of a car or car part. All these devices communicate with each other and with various managerial implements, to enable a smooth operation. These communications can be time-sensitive and are essential for the facility to operate. Problems of failures can range from significant financial cost to potential human injury or even casualty [Fra21]. To solve the problem of enabling this time-sensitive communication between all connected devices, the concept of a time-sensitive network (TSN) was introduced. Two communications intersecting at some point within this network lead to delays. The Problem therefore becomes, when and how to route different communications, so that no intersections happen. Different solutions for this problem have been widely studied, such as integer linear programming [SDT+17], satisfiability modulo theory [COA17], using constraint programming [GP20] or by mapping the constraints to a graph and finding an independent set [FGD+21]. What most of those solutions have in common is that they are static. Dynamic solutions to this problem are also vitally important, since many use-cases of time sensitive networks are in a dynamic setting [RPGS17; SALC21]. However, many of these dynamic solutions are slow.

A previously proposed method is the Greedy Flow Heap Heuristic (GFH) [FGD+21]. GFH uses a conflict graph of different communications, called flows, and calculates an independent set in it. The vertices in the conflict graph represent different flow configurations and edges between them represent a conflict between two flows, aka. using both flow configurations leads to a violation of the time constraint. As such, an independent set represents a set of flow configurations that do not interfere with each other within the original TSN. In a dynamic setting, devices are added and removed, leading to constraints shifting. That leads to a previously calculated result of GFH loosing validity, since newly added devices are not included in the result anymore. The only way to solve that using GFH, is to completely recompute a solution, using the new conflict graph. Most changes to such a network tend to be relatively small. It is likely, that only a few flows are added or removed. The conflict graph changes only marginally, but still requires a complete rerun of GFH. If these changes happen more frequently, GFH quickly becomes inefficient. That is because GFH is a static algorithm, no information of a previous result is used to potentially increase efficiency. Changes to a network also add another constraint, since adding a new device should not result in previously working connections to suddenly cease working. We want to prioritize already active connections when updating a network. GFH supports this prioritization by doing just that, any newly added connection has a significantly lower priority than an already existing one. However, it does still require a complete rerun of GFH.

We propose some changes to GFH, that lead to a more efficient computation in a dynamic scenario. In this paper, we make the following contributions:

- Two adaptations, DynGFH and WRandomGFH, to GFH employing dynamic conflict graph updates
- An empiric evaluation of our approaches to find feasible configuration parameters
- An extensive comparison against the GFH algorithm on multiple network topologies

Chapter 2 introduces our notation, as well as various concepts we use in our paper. It also introduces the problem statement, as a more formal description of the problem. In Chapter 3 we present the related work, that has been done on time-sensitive-networks, conflict-graphs and independent sets. Chapter 4 explains the application domain of TSN and scheduling problems. This chapter also explains GFH in more detail, showing its functionality and implementation. In Chapter 5, we introduce our two approaches to solving the problem. We explain our thought-process as well as the implementation of DynGFH and WRandomGFH. Following that, in Chapter 6 we evaluate DynGFH and WRandomGFH. We inspect our algorithms for optimal parameters and compare them to GFH. Chapter 7 summarizes our findings and discusses future work.

2 Preliminaries and Problem Statement

In the following, we introduce our notation of graphs and independent vertex sets, as well as their specialized forms, the vertex colored graph and independent colorful vertex set.

2.1 Vertex Colored Graphs

A Graph $G = (V, E)$ consists of a set of vertices V and edges $E \subseteq V \times V$. Edges from E connect two vertices of V. An edge $e \in E$ can also be written as a tuple (v_i, v_j) with $v_i, v_j \in V$, which contains the two vertices that e connects.
A graph can be directed or undirected. Directed graphs contain edges which can only be traversed in one direction, and edges in undirected graphs can be traversed in both. Formally, this means that in a directed graph, $(v_i, v_j) \neq (v_i, v_j)$ and in an undirected one $(v_i, v_j) = (v_i, v_j)$. Graphs in this paper are purely undirected.
Vertex colored graphs are a special case, where each vertex inside the graph has a color. So, every vertex v in V additionally contains information on what color it belongs to. To do this, we can extend the model of graph G to $G = (V, E, C)$, where $C = \{C_1, C_2, \ldots, C_n\}$ is a set of sets. These sets represent the colors of vertices: $\forall C_i \in C : C_i \subseteq V$ and $\forall C_a, C_b \in C : C_a \cap C_b = \emptyset$ and lastly $\bigcup_{i=0}^{n} C_i = V$. This model limits each vertex to being a member of exactly one color. We use a function $c(v) : V \rightarrow C$, where $v \in V$ and, $c \in C$ to return the color a vertex belongs to.
To model a dynamically changing graph, we add another parameter to the graph tuple. This parameter is a positive number denoting a timestep. A graph $G = (V, E, C, t)$ describes the graph at the time t. This means that changes to G can be modelled by increasing t incrementally and changing V, E, C. As a shorthand for the edges, vertices and groups of a graph at the timestep t we use V^t, E^t and C^t. Then, to describe a graph at a given time, we use $G_t = (V^t, E^t, C^t) = (V, E, C, t)$. To model the change in the set of vertices between two timesteps, we use $tmodV(t_1, t_2) : \mathbb{N} \times \mathbb{N} \rightarrow V_{t_1} \cap V_{t_2}$. The function $tmodV()$ produces the set of vertices that changed between the two given timesteps. The given changes between two time steps t_1 and t_2 are the differences
$V_{t_1}^{t_2} = V^{t_2} \backslash V^{t_1}$,
$E_{t_1}^{t_2} = E^{t_2} \backslash E^{t_1}$,and
$C_{t_1}^{t_2} = C^{t_2} \backslash C^{t_1}$.

2.2 Independent Colorful Vertex Set

Given a graph $G = (V, E)$ and a subset of vertices $V' \subseteq V$, we say that V' is independent, if $\forall v_1, v_2 \in V' : (v_1, v_2) \notin E$. So, an independent vertex set is a set, where no two vertices inside it have an edge in E connecting them. Extending this to an independent colorful vertex set of a vertex colored graph, means that the independent set has some amount of colors represented by

its vertices. We differentiate between the amount of colors represented. We call a set tropical, if $\forall c \in C : \exists v \in V'$ with $cOf(v) = c$. If this only holds for a subset of all colors $C' \subset C$, then the set is called colorful. Every vertex colored graph has a maximum colored set, meaning an independent colored vertex set in which the most colors possible are represented. Not every graph has a tropical set, as the restriction of containing every color is sometimes impossible to meet.

2.3 Problem Statement

Let G be a dynamic vertex colored conflict graph and S an Independent Colored Solution Set (ICSS) for G at the time of t. Assuming G now changes and is now at the time step of $t + 1$, how can we dynamically calculate the new ICSS?
This can be summarized in the following optimization problem:
Let $G_t = (V^t, E^t, C^t)$, S ⊆V an ICSS for G_t.

Let $C_r \subset C^{t+1}$ be the remaining colors used in the old ICSS S. We want to find a new ICSS S' that maximizes the objective:

$$\max_{S'} \sum_{v \in S' \cap C_r} 1 + \sum_{v \in S' \setminus C_r} \frac{1}{|C^{t'}|} \tag{2.1}$$

The factor $\frac{1}{|S'|}$ in Eq. 2.1 accounts for the relative importance of previously existing vertices in an ICSS to newly added ones. In general, this equation boils down to prioritizing colors that already existed in the previous ICSS (C_r). Prioritizing previously used colors is necessary in some applications. Using the network flow as an example, removing a color between two time-steps means that a previously possible communication suddenly is not included anymore.

3 Related Work

This paper addresses algorithms to dynamically generate an independent colorful vertex set in a vertex-colored graph. While generating independent vertex sets has been widely studied already [KW85], the special case of generating them in vertex-colored graphs has not.
A naive approach to generating a non-colored independent set is going through all possible combinations of vertices. This has an exponential complexity and is thus not suited for larger problems, due to lack of scalability. In the following, we show a brief overview of discovered solutions for the independent vertex set problem.
The first implementations of algorithms that have a complexity of $O(2^n)$, with n being the number of vertices, were developed in the early 70s [BS73; NT75]. In 1977 Tarjan et. al. first showed an algorithm to generate an independent vertex set while being a little more efficient than the previous examples [TT77]. They show that their method has a complexity of $O(2^{\frac{n}{3}})$, which allows for three times the problem size of earlier implementations. Later research revealed sequential algorithms that solve the problem in as little as $O(1.1996^n * n^{O(1)})$ time [BEPR12; XN17]. These solve the problem bottom-up, meaning they firstly consider a smaller subset of a graph and iteratively increase the problem-space to get the maximum solution. Even still, this method has more of a theoretical significance, since an algorithm of that runtime has little use since the complexity still grows exponentially.
Later, in the 80s, parallel algorithms were introduced that could speed up the process of finding an independent set by optimizing calculations to use parallelization [KW85; Lub86]. These implementations were modified Monte Carlo Algorithms that had runtimes of $O(log(n))$ up to $O(log(n)^2)$ while using m up to $n^2 * m$ processors, with m being the number of edges.
Another approach appeared in the 90s, that approximated a maximum independent set. These methods cannot guarantee to find the maximum independent set, but approximate them through various methods [BH92; BS92; BSK09]. The outcome of these approximations can still be large and useful independent sets in most scenarios.
Algorithms based on approximation generally operate on a greedy method. They apply some form of heuristic that assigns a value to nodes and then pick the node with the highest (or lowest, depending on the heuristic) value. With repeated application of this heuristic and greedy choice, an end point is reached and the resulting set is guaranteed to be independent, but not guaranteed to be the maximum independent set of the graph. This is because the greedy method can occasionally make choices that exclude the true maximum independent set by prioritizing a local maximum in one step, when a lower value could have opened up a much larger value later on. How often this occurs and the general impact on efficiency and accuracy depends on the given heuristic. None of the methods mentioned consider colors of vertices.
We now show some solutions that include the additional attribute of colored vertices as well. K. Kurita et. al. search for tropical dominating sets in graphs [KWAU21]. Due to dominating sets being only vaguely related to independent sets, their research results aren't applicable to our problem here. Research on the specific problem of calculating colorful independent sets has been studied as well. Italiano et. al. managed to solve it for some classes of graphs such as complements of

bipartite chain graphs and complete multipartite graphs [IMTP18]. More specifically, they solved the problem of finding a maximum colored clique in vertex colored graphs, which can be reduced to our independent set problem. This could be remedied by applying their algorithm on the inverse of a graph. However, their algorithm can only be applied to a limited selection of graph types and as such does not fit our specific generalized problem.
Manoussakis and Phang show that finding maximum independent colorful graphs is NP-hard and provide polynomial algorithms for cluster and tree graphs [MP18]. This limit to the two certain types of graphs limits their applicability to the problem in this paper.
Another method of obtaining an ICSS is the Greedy Flow Heap Heuristic (GFH) [FGD+21]. This heuristic takes an incremental greedy approach to find an independent colored set. It also allows for weighing of colors, meaning that some colors can have a higher priority to be in the resulting set than others. When applied to a graph where a change has occurred, this can be used to calculate a new ICSS that prioritizes colors that were in the ICSS previously. A use case for this is Traffic planning, which requires this option, because this can make sure that active flows do not get deactivated because new traffic flows appear. A problem with GFH is, that it does not dynamically calculate this ICSS. It allows for previous iterations of ICSS's to influence newer ones, with the prioritization of colors. However, it does not include data of previous solutions and recalculates an entirely new solution. This means that every change to the structure of a conflict graph leads to a potentially large amount of computation. A frequent number of small changes to a conflict graph would be the worst case, since the change from a previous solution to a new one is small, but the calculation is similar to making a large change.

4 Application Domain

The following chapter discusses time-sensitive networks (TSN) as an application domain, where scheduling problems can be resolved by solving an independent colorful set problem. TSN describe real-world networks that have a time-sensitive aspect. These networks usually consists of various end devices, that communicate with each other via a series of interlinked switches. The topology of TSN-networks can vary, however some are more common (like circles). In this paper, we assume that every end device only has one connection, meaning that every end device has one access point into the network of switches.

The idea of TSN's sprung out of the industrial internet of things (IIOT), in which machines sometimes have to communicate with each other to allow for complex processes. These machines and the various switches connecting them make up a TSN network. The time-sensitive aspect comes into play when end devices have to send packets of information to each other periodically, which have to arrive before a deadline. Violating that requirement can have outcomes that vary from hampering functionality to potentially bringing harm to humans and as such, are vital to uphold. This need to periodically communicate between two end devices can be summarized in the concept of flows. A flow includes how much has to be communicated, called the payload and how frequently is has to be communicated, called the period. Conventional methods of routing usually cannot even guarantee that packets arrive at all. This is because they can lead to bottlenecks appearing, due to two or more flows needing to traverse the same link between two switches at the same time, in the same direction. One switch then has to store one of the flow's information and delay its transmission. Since this can happen multiple times across the path of a packet, retaining the time-sensitive nature becomes less and less likely. Therefore, TSN usually has to include zero-queuing, where the networking does not allow for queues to form. Zero-queue networking predefines every flow in such a way, that none of these collisions previously mentioned occur. Depending on the network, the number of flows and their requirements, implementing such a queue-less networking can be impossible, but even knowing if it is possible is not trivial and turns out to be a NP-hard problem.

A method to calculate these flows, is to generate a conflict graph and derive the actual scheduling from the computed independent set. This conflict graph describes the conflicts that exist within our network graph. A vertex within it represents a possible configuration of a flow. What flow it belongs to is represented by a color. Two configurations are in conflict with each other, i.e. the use of both would lead to a queue forming at some point, they are in conflict with each other. Any two configurations that are in conflict with each other are connected with an edge, unless they belong to the same color. An independent colorful set in this conflict graph leads to a set of configurations of flows, that will never lead to a queue forming, and thus allows for potential use inside a TSN.

Since we are only given a network of end devices and switches, as well as a number of flows, we first have to generate a conflict graph from them. Then we have to calculate an independent set in this conflict graph. A method of calculating this independent set is the Greedy Flow Heap heuristic, which we cover in 4.1.

A problem with GFH and many other ways of calculating a solution for the zero-queue network is that most networks are not static and change over time. That means that new flows are added, and old flows are removed. GFH does not react dynamically to these sorts of changes, which can lead to decreased efficiency. This is what we strive to improve in this paper.

4.1 Greedy Flow Heap

The Greedy Flow Heap Heuristic (GFH) [FGD+21] is an iterative greedy algorithm, that heuristically solves the independent colorful set problem. GFH takes a colored conflict graph and returns an independent colorful set, though not necessarily the maximum independent colorful set.

GFH has an iterative procedure, in which every iteration tries to add a vertex to the independent set that has been calculated so far. To keep the runtime feasible, some form of rating has to occur, so that the algorithm can "greedily" pick the best option in the next step. GFH introduces a "shadow rating", this rating is calculated via a ratio of shadowed and eligible neighbors every neighbor of a vertex has. Shadowed vertices are vertices, that were removed as potential candidates in a previous step, because one of their neighbors has been added to the independent set. Adding a shadowed vertex in a later iteration would lead to a conflict. Eligible vertices are all the candidates that are not shadowed or already in the set. Every iteration adds the vertex with the minimum shadow rating, until all colors are covered, or no eligible candidates remain of the missing colors. The result of GFH is a set of vertices that make up the independent set. The input to GFH is a conflict graph and two sets that make up all colors in that graph. All colors in the active set are prioritized when generating a new independent colorful set, because previously accepted flows have to still be accepted, though they do not necessarily have to follow the same path. Colors in the required set thus have a lower priority.

GFH does not use any previously calculated independent sets in any way. In a dynamic setting, changes to the underlying network graph may occur or the amount and setup of flows may change. This leads to changes in the conflict graph and thus can invalidate a previously calculated independent colorful set. Using GFH in such a dynamic setting is only doable, by recalculating the entire independent set, every time the conflict graph is changed. So, depending on the frequency or scale of changes, GFH's efficiency can decrease drastically.

The overall structure of GFH can be seen in algorithm 4.1. The input variable n determines the amount of times this main loop will be carried out. A higher amount of loops can potentially increase the accuracy of the end result, but increases the runtime linearly. Each loop calculates a viable output set and after all iterations are done, the best set is returned. It computes four different sets out of the given active flows (ActiveF) and requested flows (ReqF). These are the accepted active flows (aActiveF), not accepted active flows (naActiveF) and similarly aReqF and naReqF. In the first iteration, both aActiveF and aReqF are empty, these sets only come into play in later iterations. GFH then first adds any unconnected vertices to a potential result set, removing the flows those belong to from consideration. It then prioritizes adding configurations of flows from the active flows first. The way addConfigPerFlow adds configs, is in a hierarchical, iterative process. A heap of all the flows given to addConfigPerFlow is sorted by the number of remaining eligible configurations a flow has. The flow with the fewest eligible configurations is processed first. What specific configuration is used is determined by assigning a shadow-rating to each and taking the one with the lowest. The shadow-rating of a configuration a is calculated by checking what percentage

Algorithm 4.1 GFH: main algorithm loop

procedure GFH // Input: active flow set ActiveF(p), new flow set ReqF(p), n re-runs
$C \leftarrow \emptyset$
while $n > 0$ **do**
$naActiveF \leftarrow ActiveF(p)\ C$
$aActiveF \leftarrow ActiveF(p) \bigcap C$
$naReqF \leftarrow ReqF(p)\ C$
$aReqF \leftarrow ReqF(p) \bigcap C$
$C \leftarrow \{v \in V : degree(v) == 0\}$ // C gets solitary configurations
$C \leftarrow addConfigPerFlow(naActiveF, C)$
$C \leftarrow addConfigPerFlow(aActiveF, C)$
$C \leftarrow addConfigPerFlow(naReqF, C)$
$C \leftarrow addConfigPerFlow(aReqF, C)$
if C is Maximum Independent set **then**
return C
else
cache C
end if
n–
end while
end procedure

of remaining eligible configurations different flows would lose, if the configuration a was chosen. The higher the percentage, the higher the shadow-rating, with a huge penalty for eliminating all remaining configurations of a flow. This setup results in always choosing the configuration that has the least impact on other flows, potentially allowing for more flows to be included in the end result.

Our problem statement aims to allow for a more dynamic approach. It is similar to the problem statement that lead to GFH, but includes the additional aspect of time steps. Instead of looking at the problem as calculating an independent colorful set out of a conflict graph, we add the additional variables of the previous conflict graph and its calculated independent set. Our algorithm thus has additional access to the previous conflict graph, as well as the previous independent set.

5 Dynamic Independent Colorful Set

This chapter discusses how we construct our two dynamic algorithms, DynGFH and WRandomGFH. DynGFH caches how additions to an ICSS affect other vertex candidates and uses that data to update an ICSS instead of completely recalculating it. WRandomGFH uses weighted randomness to add new colors iteratively, with colors weighing more, the lower the total degree of all their vertices is.

The Greedy Flow Heap Heuristic (c.f. Section 4.1) is an example of a static algorithm. It takes an input graph and generates an Independent Colorful Set. When the graph is changed, e.g., flows are added or removed, GFH does not react to it, instead it needs to be reapplied to the changed graph. Since real-life applications are usually factories with many cyber physical systems, most changes boil down to a few new flows being added or some flows being removed. This means that these networks usually remain relatively static, with only small changes occurring over time.

These types of changes are the worst case for GFH, since every minor change in the conflict graph requires a complete rerun that recalculates the entire independent colorful set. A Dynamic solution would react to only the changes and their affected areas. This means that the input to our algorithm has to change, instead of getting a set of active and required flows, we need a set that encapsulates the change that happened in the new time step, as well as the changed conflict graph.

5.1 DynGFH

One way we thought to implement a dynamic solution, is by only modifying GFH to what we call DynGFH, for Dynamic GFH. DynGFH functions almost identically to GFH, but we take the additional input of all the colors that were removed in the time step, and use them to update the configuration we calculated in the previous time step. This way, the first iteration of the GFH algorithm already has a configuration set to work with. DynGFH tries to jump ahead in a typical GFH execution, by assuming that removing a few colors would not change the resulting set significantly. We therefore assume, that removing a few colors is a faster method than recalculating with all colors to end up with a similar result.

Removing colors from configuration sets can be done in various ways. One method is to recalculate the shadowed vertices of a configuration based on the remaining admitted vertices of each color. Another approach we considered, was to cache additional information during construction of an ICSS, that could be used in the next time step to remove colors more dynamically. Every time we add a vertex to an ICSS, we save which other vertices were shadowed by it. The cached data can be updated when removing colors. When a shadowed vertex is not shadowed anymore, it becomes an eligible vertex again. How much this setup reduces runtime depends on the ratio of removed colors to total colors, since we now iterate over all removed colors. This method also adds a lot of

overhead to simply adding a vertex into the ICSS. For this method to be feasible, the additional overhead needs to impact the performance less, than what the dynamic approach improves. If this is the case is not obvious and will be discussed in Chapter 6.

Algorithm 5.1 DynGFH: main algorithm loop

Input: added flow set AddF, set of removed flows RemF, number of repeats n
Output: An ICSS

```
1:  procedure DynGFH
2:      ▷ Changes start here
3:      C ← config calculated in previous time step
4:      C ← remove flows in RemF from C     // Here is where we recalculate or use cached data
5:      ActiveF(p) ← acctepted flows in C
6:      ReqF(p) ← AddF
7:      ▷ Changes end here
8:      naActiveF ← ActiveF(p) C
9:      aActiveF ← ActiveF(p) ⋂ C
10:     naReqF ← ReqF(p) C
11:     aReqF ← ReqF(p) ⋂ C
12:     C ← {v ∈ V : degree(v) == 0}        // Not strictly neccessary anymore
13:     C ← addConfigPerFlow(naActiveF, C)
14:     C ← addConfigPerFlow(aActiveF, C)
15:     C ← addConfigPerFlow(naReqF, C)
16:     C ← addConfigPerFlow(aReqF, C)
17:     return C
18: end procedure
```

Algorithm 5.1 shows how we changed the GFH algorithm to work more dynamically. In line 3, the previous configuration is updated, any accepted flows that have now been removed, need to be accounted for. That means, the configuration no longer accepts them, and any vertices that were shadowed by that flow need to be recalculated as well. Here is where we either recalculate completely, based on the reduced number of flows, or implement the previously introduced caching method. We do not need ActiveF as input anymore, since we have the solution of the previous time step and can get them there. Instead, we set ActiveF and ReqF in lines 4 and 5. ReqF is still the set of added flows, while ActiveF is now the remaining accepted flows in our configuration.

Next, DynGFH continues in the same fashion as GFH. We calculate ActiveF and ReqF from the input, ActiveF from the previous config and ReqF are just all added colors. GFH has an additional loop, where an input variable dictates how many times a valid config is calculated, each with slightly different priorities. In the original algorithm, this was to potentially generate a better solution due to the different priorities. In our case, this additional looping would greatly reduce the impact of having the updated solution of the previous time step, since each loop completely generates a new config. As such, we only gain an advantage in the first iteration. We will evaluate in Chapter 6, whether additional reruns have a significant enough impact, to warrant implementation.

Due to DynGFH's defensive approach to scheduling, integrating our problem statement Section 2.3 is trivial. Since no vertex is removed, unless the color no longer exists, all previously accepted colors are guaranteed to be in any generated ICSS. We rate an ICSS based on the amount of colors in it.

5.2 WRandomGFH

We also implemented a different algorithm, WRandomGFH. The way GFH and DynGFH choose a vertex can lead to pitfalls. A greedy choice can prevent the algorithm from reaching better results, due to lack of "foresight". Therefore, WRandomGFH has a probabilistic approach, instead of relying on deterministic sorting and evaluation, like GFH and DynGFH. It is a weighted, randomized algorithm, which works off the first calculated GFH configuration. The first step cannot be done in a dynamic way, since no previous data exists, and therefore could be done with any algorithm that generates an ICSS. But since we are evaluating against GFH, it lends itself to use for calculating the first ICSS. WRandomGFH tries to find a fitting ICSS for a new time step, by locking the remaining configurations of the previous time step, then iteratively and randomly choosing a color to add to the set. This still takes into account the shadowed and therefore not eligible vertices, but removes the computation of shadow ratings. Instead, we use weights to increase the chances of more important colors and vertices to be chosen. In each iteration, a color is chosen, then a random vertex of that color is chosen. A color, whose total combined degree of all its vertices is higher than that of another color, has a lower weight. When a color has been chosen, a vertex' weight depends on its degree, with a higher degree leading to a lower weight. We base these weights on the degree, because the higher the degree of a color or degree, the more neighbors are potentially shadowed. This makes it more likely that colors, that had few possible vertices to choose from, get eliminated entirely. Since we want to include as many colors into our solution as possible, we have to "penalize" colors and vertices that are likely to limit it.

This algorithm is randomized, and unlike GFH or DynGFH it might provide bad solutions by chance. GFH and DynGFH on the other hand might generate bad solutions due to problematic inputs. They put a lot more computational effort into choosing a vertex each iteration, whereas WRandomGFH only looks at the total degree. Therefore, we use the fact that this algorithm runs a lot faster than either of those, and just repeat it a number of times, taking the best result.

Algorithm 5.2 presents WRandomGFH's workflow as pseudocode. The first time step is the same as DynGFH or GFH, we simply calculate an ICSS for the first time step, since no dynamic calculation is applicable. In all other time steps, we have a loop, that repeatedly generates a random potential ICSS configuration and finally outputs the best one. Each loop iteratively chooses a vertex of a color and adds it to the potential configuration until no colors remains in the pool of available colors. Colors leave that pool, when they have been chosen, or they have no eligible vertices remaining.

First, all colors that contain at least one solitary vertex are added, since these do not need any further calculation and can just be added to the set. Then a weighted random algorithm is used to determine what vertices are chosen. A random color is chosen, based off a weight that is calculated with their total-color-degree. The total-color-degree of a color is the total degree of all vertices inside with a color. A higher total-color-degree leads to a lower weight of the color. To facilitate this weight behavior, we use a number to divide by the total-color-degree. The number we use, is the amount of edges inside the conflict graph times two, called $weightDecider$, i.e. $color - weight = \frac{weightDecider}{total-color-degree}$. Edges get multiplied by two, because the calculation of the degree of a vertex ignores if an edge has been counted by a different vertex before. This leads to the minimum weight being 2, when a color's vertices encompass all edges within the conflict graph. Since a color never has edges between its own vertices, the max total color degree is the

Algorithm 5.2 WRandomGFH: main algorithm loop

```
Input: added flow set AddF, set of removed flows RemF, number of repeats n
Output: An ICSS
procedure WRandomGFH
    if Its the first time step then return DynGFH(AddF, RemF)
    end if
    C ← config calculated in previous time step
    C ← remove flows in RemF from C
    C ← add flows in AddF with solitary vertices
    bestConfig ← C
    while n > 0 do
        AddF_Copy ← AddF
        potentialConfig ← C
        while AddF_Copy ≠ ∅ do
            f ← randomly choose a color, based on its total degree
            v ← randomly choose a vertex of color f that is eligible, based on its degree
            potentialConfig ← add v
            AddF_Copy ← remove f
        end while
        if potentialConfig > bestConfig then
            bestConfig ← potentialConfig
        end if
    end while
    return bestConfig
end procedure
```

amount of edges in the conflict graph. The maximum color weight would be $weightDecider$, when a color only has one edge going from all its vertices, since colors without edges, have already been accepted.

After a color has been randomly chosen, a vertex belonging to that color is chosen. We looked at three different options here:

- randomly choosing a vertex of the color
- choosing a random vertex of a color, but the vertex has to have below the average degree of the color
- similar to how we choose the color, take a random vertex with the weight depending on its degree

The weight divider for the third option would be the total color degree. We evaluate these options in Chapter 6. Algorithm 5.3 contains the pseudocode for choosing a color based on weight. It details the procedure for calculating weights of colors and how we implemented the weighted random selection.

Similarly to DynGFH, WRandomGFH's ICSS is guaranteed to contain all remaining colors from the previous time step. We rate an ICSS based on the amount of colors within, which automatically maximizes the formula we introduced in our problem statement 2.3.

Algorithm 5.3 Weighted Random Color

Input: Set C of colors, graph G that contains those colors
Output: A random color in C, with probabilities proportional to their weights

1: **procedure** WRColor
2: $weightDecider \leftarrow |E_G| * 2$ // number of vertices in G times two
3: $weights \leftarrow \emptyset$
4: ▷ calculate the weights of colors in C
5: **for** color c in C **do**
6: $total_color_degree \leftarrow \Sigma_{v \in c} degreeOf(v)$
7: $weights[c] \leftarrow \frac{weightDecider}{total_color_degree}$
8: **end for**
9: ▷ Choose a random color
10: $total_weight \leftarrow \Sigma_{w \in weights} w$ // The total weight of all colors added together
11: $random_number \leftarrow$ a random number between 0 and $total_weight$
12: $chosencolor \leftarrow$ first color in C
13: **while** $random - number > weights[chosencolor]$ **do**
14: $random_number \leftarrow random_number - weights[chosencolor]$
15: **end while**
16: **return** $chosencolor$
17: **end procedure**

6 Evaluation

This chapter evaluates our the algorithms WRandomGFH, DynGFH and GFH, and compares them to each other. We investigate the impact of the parameters we used for WRandomGFH, DynGFH on their performance. Further, we explain what how we generated the datasets used in the evaluation, and the specifications of the system they were executed on.

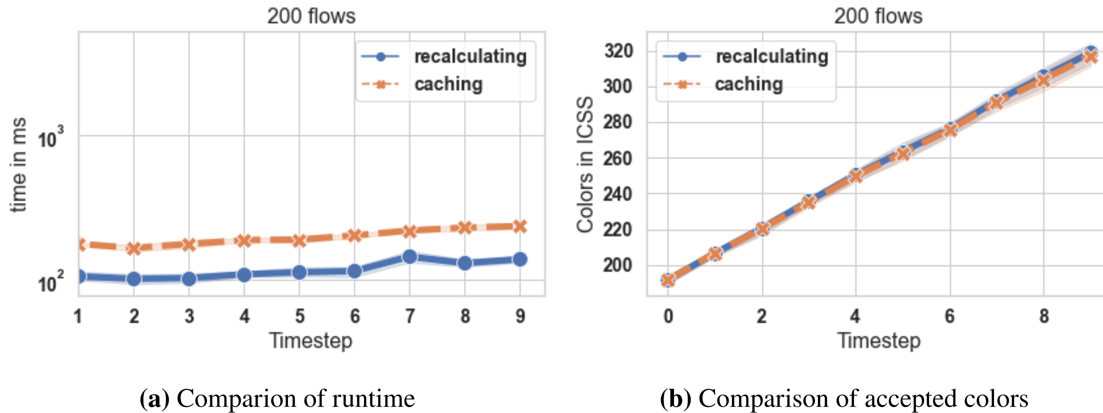
All evaluations were performed on a server machine. This machine runs on Ubuntu 20.04, has two AMD EPYC 7401 24-Core processors (including hyper-processing, that leads to 96 threads), and has 128 GB of RAM. We wrote GFH, DynGFH and WRandomGFH in C#, using dotnet version 6.0.100, compiling to netcoreapp 5.0. To get the data necessary to compare all of these algorithms, we used python to generate network graphs and evaluation scenario files that contain flow configurations for that network graph, but used C# to compute the resulting conflict graph.

Two types of network graphs were used: a circle and random. The network graphs were made to simulate real world scenarios, so they are made up of switches and end devices. The switches make up the actual network graph structure, all end devices are only connected to this circle via one edge. The circle was generated with each switch connected to its next two neighbors. The randomly generated network was generated with twice its switches as edges, using the Erdös & Rényi method. This network is used to generate the scenario file of flows in the python script. It randomly generates a number of flows that go from one end device to another and chooses a random package size and period, from a pool of choices, for each. Lastly, it simulates time steps, by randomly removing a number of flows followed by adding new ones. Parameters are: the number of flows to start with, the number of time steps to generate, the number of flows that are added or removed per time step, as well as candidates for package size and period that can be chosen.

We used 100 switches and 50 end devices for our network graph. We compared various setups for flow configurations, mainly differentiating them on the number of flows they start with. Every scenario uses 10 time steps and removes one tenth of the starting number of flows each time step, while adding one fifth, so a scenario with 100 starting flows would remove 10 and add 20 each time step. We compared scenarios starting with 100, 200 and 500 flows. Each flow has a period of 250, 500, 1000 or 2000 μs and a package size of 125, 250, 500, 750, 1000 or 1500 bytes. We evaluated ten different scenarios that were generated with the same configurations, i.e. we had 10 different scenarios that started with 100 flows, ten with 200 and ten with 500.

6.1 Evaluating DynGFH Parameters

We now interpret our evaluation of the two methods we could use to remove sets from a DynGFH config. Our two options are either completely recalculating a config after all colors have been removed, or the caching method we introduced in Section 5.1. We compare them on a circle graph, with 10 different flow configurations. DynGFH was executed on each configuration with



(a) Comparion of runtime

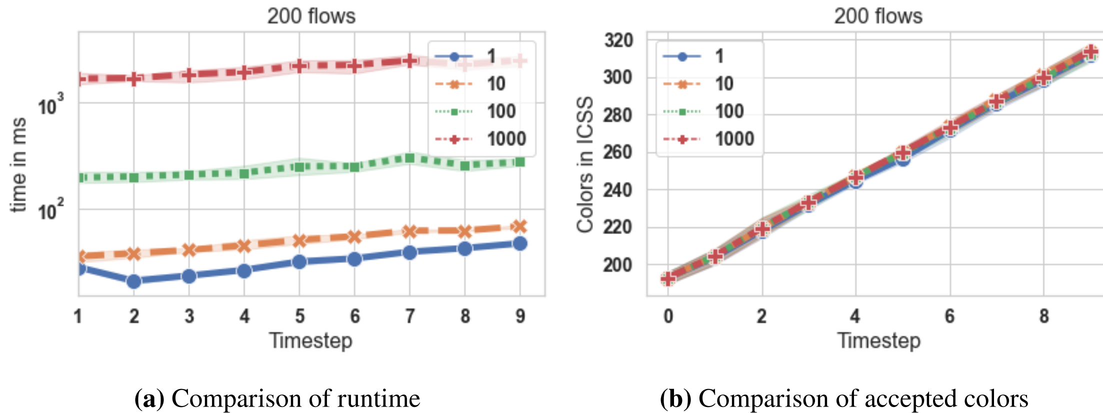
(b) Comparison of accepted colors

Figure 6.1: Results of comparing Recalculating vs Caching. Compared configurations start with 200 colors. The left diagram depicts the time each method took in ms. The caching-method has a slightly increased runtime. The left diagram shows, that, as expected, both methods result in similar ICSS.

and without the caching method. The results can be seen in Figures 6.1. The first graph 6.1a illustrates caching having a higher runtime, while the second graph 6.1b shows a slightly reduced number of colors for caching. This might be due to inefficiencies in our implementation, as the difference minor and does not change. Since Figure 6.1a has a logarithmic scale, the shown runtime difference is quite small. Some optimizations might be able to decrease the runtime difference substantially. We can conclude, that our implementation of caching shadowed vertices is not worth the additional overhead it generates, with the parameters we used. Therefore, we will use the recalculation approach in the remainder of the evaluation.

6.2 Evaluation of WRandomGFH Parameters

In this section, we evaluate different parameters of WRandomGFH and compare their impact. The specific parameters we have to consider here are: First, the number of candidate configurations that should be calculated. Every additional potential configuration increases the runtime linearly, but also adds another chance to randomly generate a better configuration. Second, how a vertex is chosen. We considered three different options here in Chapter 5: random, random-below-average and degree-weighted. We started by comparing different amounts of candidate configurations. We generated a network graph with a circle topography, and generated five different flow scenarios. On each, we used four different amounts of candidate configurations, going from 1 candidate to 10, 100 and 1000 candidates. Figure 6.2 displays the results of our comparison. As expected, a higher amount of repeats leads to longer execution times, as seen in the chart 6.2a. However, in our scenario, the different amounts of repeats seem to have little impact on the amount of colors in the resulting ICSS. This can be attributed to the defensive way that WRandomGFH operates. It is likely, that without changing the already accepted colors in the last time step, even just one repeat already reaches the maximum amount of colors possible, with this method. Given our starting amount of 200 flows, i.e. 200 colors, each time step adds 40 colors and removes 20. The maximum the amount of colors can that can be reached in any time step t, is $200 + t * 20$. Additionally, WRandomGFH



(a) Comparison of runtime

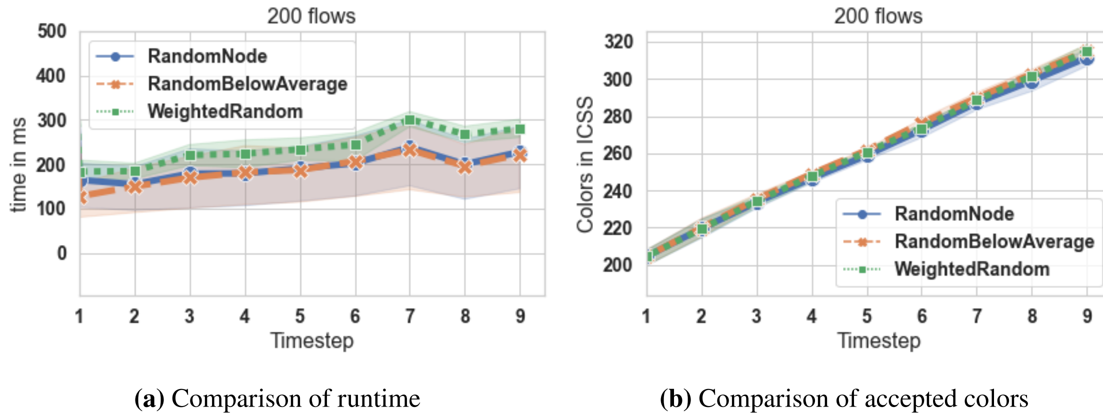
(b) Comparison of accepted colors

Figure 6.2: Results of comparing different amounts of repeats. The left chart illustrates the different execution times. As expected, the higher the number of repeats, the longer the execution time. The left chart displays how many colors ended up in the resulting ICSS each time step. The parameters we used lead to no major differences between the different amounts of repeats.

ignores any colors that were not accepted in the last time step, as such greatly lowering that ceiling. All this leads to, is repeats having less of an impact when little changes between time steps. To reach a middle ground between runtime and potential gains from repeats, we choose 100 repeats, since the runtime difference to 1 or 10 is negligible in comparison to GFH runtimes and still leaves room for benefits in higher starting flow numbers.

Next, we compare the different methods to choose a vertex. Similarly, we generated five different flow configurations for a network and then executed WRandomGFH using the three ways of selecting a vertex. These are: One, completely randomized. Two, randomized from all vertices that have a degree below the average of the color. And three, randomized, but with weights depending on the inverse of their weights. Figure 6.3 provides an overview on how they compare against each other. The degree-weighted algorithm has the longest runtime, but also presents better results, when inspected closely in Figure 6.4. The random-below-average is a good approximation and comes close to degree-weighted, while random relies heavily on chance. It is also worth mentioning, that the random algorithm has the highest gain from the previously inspected repeats. The nature of the degree weighted algorithm makes repeats less and less impactful the more there are. Here again, random-below-average is in the middle ground, because repeats have more of an impact than in degree-weighted and less of one than in random.

From these result, we conclude that 100 repeats is an adequate amount to allow random chance to generate better results, while not drastically increasing the runtime. Furthermore, we select the degree-weighted algorithm to choose vertices, since the runtime difference is negligible, while still providing better results on average.



(a) Comparison of runtime

(b) Comparison of accepted colors

Figure 6.3: Results of comparing different ways of selecting a vertex. The left diagram indicates a higher runtime for the degree-weighted algorithm and very similar runtimes for the random and random-below-average ones. In the left diagram, again, a similar result for all of them is expected.

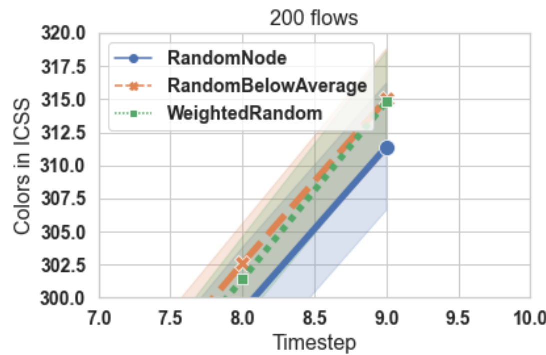
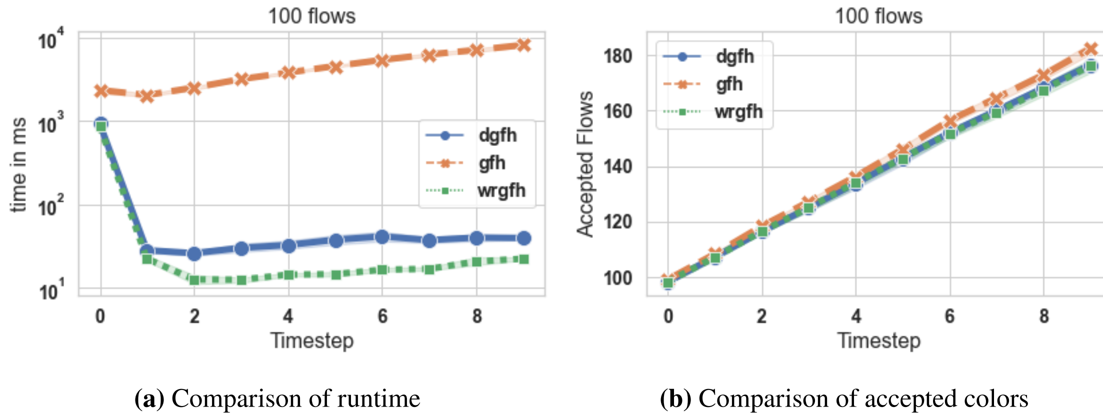


Figure 6.4: A more detailed comparison of the different vertex-selection algorithms, in later time steps. As expected, the degree-weighted algorithm, on average, ends up with the highest amount of colors in the ICSS. Followed by random-below-average and random.

6.3 Evaluation against the state of the art

We now evaluate GFH, DynGFH and WRandomGFH and compare them based on runtime and average number of colors in their ICSS. Ten graphs are generated, and three flow scenarios are built for each of them, containing ten time steps, starting from zero. The first scenario built has 100 flows starting out. The next has 200 and the last has 500. Each scenario uses parameters we introduced in the beginning of this chapter. We run GFH, DynGFH and WRandomGFH on ten variations of each scenario.

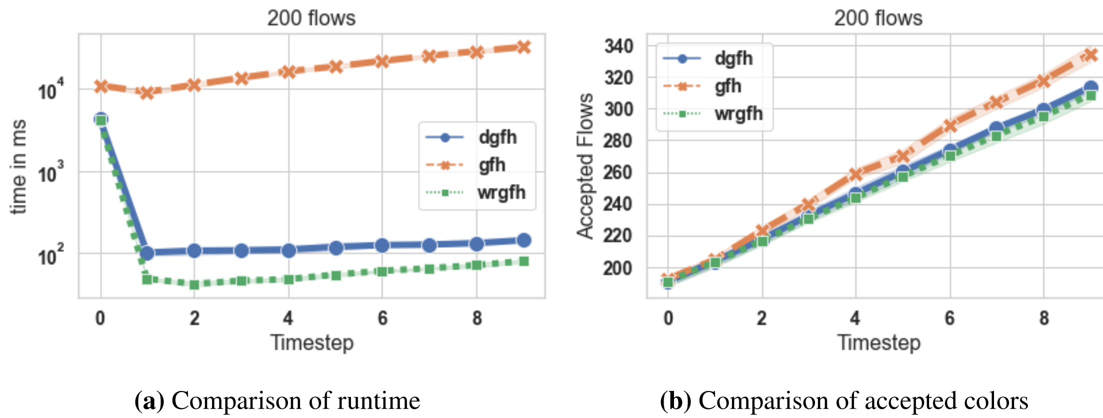
Figures 6.5, 6.6 and 6.7 present our results. We can see that there is little difference between all three algorithms in the first step. This is because all of them use GFH to calculate the first step, producing the same ICSS. The only difference is the slightly lower runtime of DynGFH and WRandomGFH, because our reimplementation of GFH inside DynGFH's code is slightly more efficient. The later steps show a clear advantage for WRandomGFH and DynGFH when it comes to



(a) Comparison of runtime

(b) Comparison of accepted colors

Figure 6.5: Results of GFH, DynGFH and WRandomGFH, when run on a scenario with 100 initial flows.



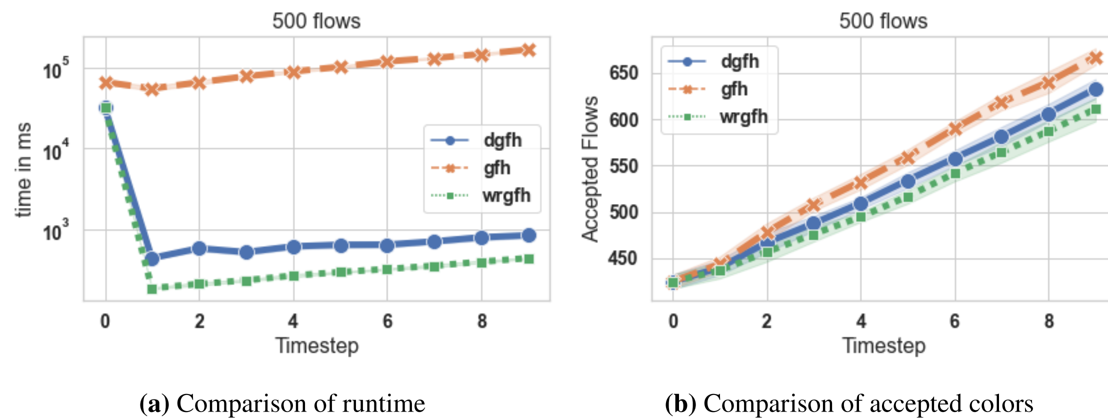
(a) Comparison of runtime

(b) Comparison of accepted colors

Figure 6.6: Results of GFH, DynGFH and WRandomGFH, when run on a scenario with 200 initial flows.

runtime. Due to both using a defensive approach, they only operate on the newly given colors each time step. However, GFH has the advantage when it comes to accepted flows each time step. This is most prominent in Figure 6.7b. WRandomGFH performed the worst, when it comes to accepted flows, with DynGFH performing better on average. We can deduce, the higher the initial amount of colors is, the more of an advantage GFH gets. With our parameters, the amount of change in every time step scales with the initial flow amount. The more changes occur per time step, the more DynGFH and WRandomGFH suffer in terms of size of ICSS. Each time step, their defensive planning foregoes correcting now inefficient vertices and the more vertices are added, the more likely those become.

In summary, DynGFH has potential as a dynamic algorithm for ICSS calculation. It generates larger ICSS than a randomized application like WRandomGFH with a minimal increase in runtime. Its defensive nature results in worse ICSS than GFH generates, but takes a fraction of the time.



(a) Comparison of runtime

(b) Comparison of accepted colors

Figure 6.7: Results of GFH, DynGFH and WRandomGFH, when run on a scenario with 500 initial flows.

7 Conclusion

With the advent of the industrial internet of things (IIOT), new scheduling problems arose, due to the need to reliably and efficiently communicate between devices. As such, TSNs make up the backbone of many IIOT implementations. However, most of them do so statically and require massive time and resource investment every time a change has to be made. The Greedy Flow Heap Heuristic (GFH), an existing static algorithm, abstracts the scheduling problem into an independent set problem of a conflict graph. These sets are called independent colorful set (ICSS).

In this paper, we implemented two different dynamic algorithms for scheduling in TSNs. We built on GFH and introduced time steps, a variable depicting the changes to the underlying TSN over time. One algorithm, DynGFH, is deterministic, similar to GFH, and the other one, WRandomGFH, is randomized. DynGFH uses a defensive scheduling approach, meaning the results of the previous time steps are fixed. GFH is then executed on only the changes that occurred in each time step, to generate a result. WRandomGFH uses GFH to calculate the first time step. Following time steps are calculated, by iteratively adding colors to the ICSS in a randomized way. We base this random choice on various aspects, leading to the most likely colors being ones, which put the least amount of strain on future additions.

Our evaluation indicates, that while our algorithms generate smaller ICSS, compared to GFH, they take a fraction of the time. The larger the network of flows becomes, the more this difference is evident. The large disparity in runtime, compared to the smaller disparity in ICSS size, allows for many quality-runtime tradeoffs. Especially with smaller networks, the reduced runtime could allow for more frequent changes, that might be too much for GFH's algorithm to calculate in time. That same trade-off exists between DynGFH and WRandomGFH, although to a lesser degree. WRandomGFH has an even shorter runtime than DynGFH, while not reaching the sizes of DynGFH's ICSS. In both runtime and ICSS size, this difference is a lot smaller than to GFH. Each algorithm has its own use-case, where either the biggest ICSS or the fastest runtime are the priority. Alternatively, DynGFH could be used in combination with GFH to achieve a tradeoff between GFH's results and DynGFH's short runtimes.

7.1 Future Work

There are various additions to DynGFH and WRandomGFH that could result in a better performance. Their low runtime lends itself to multiple calculations per time step. For example, instead of executing the defensive algorithm of DynGFH on the last two to three time steps could result in better ICSS, while only multiplying the runtime by two or three. That would still put it at a fraction of GFH's runtime, while giving the algorithm a higher number of colors to optimize decisions.

A combination of DynGFH and GFH can also greatly reduce runtime, while barely affecting resulting ICSS. By using GFH every n'th time step, any faulty decisions made in previous ones would be mitigated, while reducing overall runtime substantially. Even using GFH every other time step, would almost reduce overall runtime by half.

A possible extension to DynGFH would be to calculate the impact of the changes that occurred in each time step. For example, checking, whether a vertex that is currently in the ICSS has a greatly increased degree due to changes and selectively removing it. This would allow for potentially removing the “bad” vertices, that the complete recalculation of GFH wouldn't consider, while not requiring said expensive recalculation.

Bibliography

[BEPR12] N. Bourgeois, B. Escoffier, V. T. Paschos, J. M. van Rooij. “Fast algorithms for max independent set”. In: *Algorithmica* 62.1 (2012), pp. 382–415 (cit. on p. 15).

[BH92] R. Boppana, M. M. Halldórsson. “Approximating maximum independent sets by excluding subgraphs”. In: *BIT Numerical Mathematics* 32.2 (1992), pp. 180–196 (cit. on p. 15).

[BHCW18] H. Boyes, B. Hallaq, J. Cunningham, T. Watson. “The industrial internet of things (IIoT): An analysis framework”. In: *Computers in industry* 101 (2018), pp. 1–12 (cit. on p. 11).

[BS73] E. Balas, H. Samuelsson. *Finding a Minimum Node Cover in an Arbitrary Graph.* Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP, 1973 (cit. on p. 15).

[BS92] P. Berman, G. Schnitger. “On the complexity of approximating the independent set problem”. In: *Information and Computation* 96.1 (1992), pp. 77–94 (cit. on p. 15).

[BSK09] S. Balaji, V. Swaminathan, K. Kannan. “Approximating maximum weighted independent set using vertex Support”. In: *International Journal of Computational and Mathemtical Sciences1* 3.8 (2009), pp. 406–411 (cit. on p. 15).

[COA17] S. S. Craciunas, R. S. Oliver, T. Ag. “An overview of scheduling mechanisms for time-sensitive networks”. In: *Proceedings of the Real-time summer school LÉcole dÉté Temps Réel (ETR)* (2017), pp. 1551–3203 (cit. on p. 11).

[FGD+21] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. *Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows with Conflict Graphs*. 2021. arXiv: 2105.01988 [cs.NI]. URL: https://arxiv.org/abs/2105.01988 (cit. on pp. 11, 16, 18).

[Fra21] Frank Wunderlich-Pfeiffer. *Golem News*. online. 2021. URL: https://www.golem.de/news/automatisierung-brand-in-lagerhaus-nach-roboterkollision-2107-158254.html (cit. on p. 11).

[GP20] V. Gavriluţ, P. Pop. “Traffic-type Assignment for TSN-based Mixed-criticality Cyber-physical Systems”. In: *Acm Transactions on Cyber-physical Systems* 4.2 (2020), pp. 1–27 (cit. on p. 11).

[IMTP18] G. F. Italiano, Y. Manoussakis, N. K. Thang, H. P. Pham. “Maximum colorful cliques in vertex-colored graphs”. In: *International Computing and Combinatorics Conference*. Springer. 2018, pp. 480–491 (cit. on p. 16).

[KW85] R. M. Karp, A. Wigderson. “A Fast Parallel Algorithm for the Maximal Independent Set Problem”. In: *J. ACM* 32.4 (Oct. 1985), pp. 762–773. ISSN: 0004-5411. DOI: 10.1145/4221.4226. URL: https://doi.org/10.1145/4221.4226 (cit. on p. 15).

[KWAU21] K. Kurita, K. Wasa, H. Arimura, T. Uno. "Efficient enumeration of dominating sets for sparse graphs". In: *Discrete Applied Mathematics* 303 (2021), pp. 283–295 (cit. on p. 15).

[Lub86] M. Luby. "A simple parallel algorithm for the maximal independent set problem". In: *SIAM journal on computing* 15.4 (1986), pp. 1036–1053 (cit. on p. 15).

[MP18] Y. Manoussakis, H. P. Pham. "Maximum colorful independent sets in vertex-colored graphs". In: *Electronic Notes in Discrete Mathematics* 68 (2018), pp. 251–256 (cit. on p. 16).

[NT75] G. L. Nemhauser, L. E. Trotter. "Vertex packings: Structural properties and algorithms". In: *Mathematical Programming* 8.1 (1975), pp. 232–248 (cit. on p. 15).

[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)*. IEEE. 2017, pp. 1–6 (cit. on p. 11).

[SALC21] A. A. Syed, S. Ayaz, T. Leinmüller, M. Chandra. "Dynamic Scheduling and Routing for TSN based In-vehicle Networks". In: *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*. 2021, pp. 1–6. DOI: 10.1109/ICCWorkshops50388.2021.9473810 (cit. on p. 11).

[SDT+17] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, 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. Grenoble, France: Association for Computing Machinery, 2017, pp. 8–17. ISBN: 9781450352864. DOI: 10.1145/3139258.3139289. URL: https://doi.org/10.1145/3139258.3139289 (cit. on p. 11).

[TT77] R. E. Tarjan, A. E. Trojanowski. "Finding a Maximum Independent Set". In: *SIAM Journal on Computing* 6.3 (1977), pp. 537–546. DOI: 10.1137/0206038. eprint: https://doi.org/10.1137/0206038. URL: https://doi.org/10.1137/0206038 (cit. on p. 15).

[XN17] M. Xiao, H. Nagamochi. "Exact algorithms for maximum independent set". In: *Information and Computation* 255 (2017), pp. 126–146 (cit. on p. 15).

All links were last followed on April 11, 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