Resumé

Vi arbejder med Software Defined Networking, hvor vi kigger på, hvordan man kan opdatere et netværk uden at overbelaste det. Netværk kan modelleres som grafer med knuder og kanter, hvor hver knude er en router og hver kant er en forbindelse mellem to routere. En mængde data der bliver sendt igennem netværket fra én router til en anden kalder vi en strøm, og hver strøm følger nogle bestemte stier. Hvis man vil ændre de stier som strømmen følger, så kan man opdatere strømmen.

Problemerne opstår når man vil opdatere flere strømme på samme tid. Netværk er decentraliserede, hvilket betyder, at tidssynkronisering mellem routerere er praktisk set umuligt, så hvis man prøver at opdate flere routers på samme tidspunkt, så ved man ikke hvilke, der bliver opdateret først. Det kan være et problem, for så risikerer man, at data bliver tabt under opdateringen.

En måde at håndtere dette problem på er, at gennemtvinge en bestemt rækkefølge som routerne skal opdateres i, hvor man da kan garantere at ingen data bliver tabt, og denne tilgang er velstuderet. Vi arbejder med en variant af denne idé ved at tillade, at man kan opdatere strømme delvist, så noget af dataen følger den både den gamle og den nye sti, og ved at man kan dele opdateringerne op i mindre dele på denne måde, så kan man finde opdateringssekvenser som undgår overbelastning, som man ikke ville kunne finde, hvis man opdaterede hele strømme på én gang.

Vi beviser, at dette problem kan løses i polynomialtid sammen med en *monoton* variant af problemet. Ydermere beviser vi, at en *atomisk* variant af problemet, hvor man kun kan opdaterere en strøm ad gangen, er NP-komplet, og vi beviser også, at en *fastsat granularitets*-variant, hvor man kun kan opdatere strømme i inkrementer af en vis størrelse, er NP-svær.

Vi reducerer det generelle problem til en lineær programmeringsmodel. Det der hovedsaligt adskiller vores arbejde fra andet literatur på området er, at vi ikke bare finder bare en løsning som er 'god nok', vi finder den optimale løsning, der giver minimal belastning på netværket. Vi implementerer vores lineære programmeringsmodel, og kører en række eksperimenter med denne. Vores eksperimenter viser, at den lineær programmeringsmodel i sig selv giver gode resultater, men ved at bruge diverse teknikker til at løse problemet, såsom at fjerne overflødige strømme og kanter, så kan vi i mange tilfælde mindske køretiden til en tiendedel.

Optimizing Link Utilization During Network Migration

Magnus H. Lundbergh and Benjamin N. Nielsen
mlundb18@student.aau.dk, bnni18@student.aau.dk

Department of Computer Science, Aalborg University, Aalborg, Denmark

Abstract. We study consistent network updates with the focus on minimizing transient congestion. While we migrate the network from its initial configuration to its final configuration, we allow each flow to split its data between the old and the new path. The goal is to find a sequence of at most n split ratios for each flow that allows it to transition to the final configuration while minimizing congestion. We study the computational complexity of different variants of this problem, and we find that the most general variant of our problem can be solved in polynomial time, and we show how to reduce it to a linear programming problem. While most literature that studies congestion-free updates simply find a solution that is good enough, our model is capable of finding an optimal solution that minimizes the maximum link utilization. In order to improve the scalability of our approach, we propose multiple techniques, and we run experiments which show that these techniques can improve computation time by an order of magnitude.

1 Introduction

Computer networks are everywhere. From the ubiquitous Internet to data centers and IoT, we interact with networks in every part of our lives. But as the world has become more connected, the amount of data traffic has increased immensely; from 2016 to 2021, the data generated globally increased from 218 ZB to 847 ZB [5]. The enormous amount of data can be difficult for networks to manage, because traditionally, each switch and router controls path selection in a fully distributed manner, which scales poorly when networks are large and complicated [6]. When traffic demands change often, distributed control makes sense, because it has good fault tolerance, but when this is not the case, there is a compelling argument for removing control from the switches in exchange for a centralized controller [4].

Software Defined Networking (SDN) [6] [17] is a growing paradigm for controlling network infrastructure, where the control plane and data plane are separated, so a centralized controller is responsible for path selection and update scheduling, while the switches are only responsible for forwarding data. When the controller has determined paths for each data flow, it sends the new rules to each switch and router, which then update their tables.

However, it is difficult to synchronize clocks in a distributed system, so what happens if one switch updates its tables before the other switches are ready? In the worst case, this can lead to packets looping, packet drops, and link congestion. Even if this loss is temporary, it might be undesirable, which has brought about the field of consistent migration [11], which is about constructing update schedules such that switches are updated with an ordering that guarantees certain properties for networks during migration.

There are multiple avenues to explore in this field. *Foerster et al.* [11] have made what is, at the time of writing, the most comprehensive survey in the field of consistent updates for SDN, and they define three categories of transient consistency properties: connectivity consistency, policy consistency, and capacity consistency.

Connectivity consistency is about ensuring that packets have a path to their destination. There are two major topics in this category, the first of which is loop-freedom [8][10], which is about guaranteeing that, during the update process, switches do not send packets in back and forth between each other, and this is the oldest area of study in the field of consistent updates. The other topic is blackhole freedom [10], which is about avoiding packets ending up in a dead end before arriving at their destination. In general, these problems are NP-hard, though there exist special cases where the problems can be solved in polynomial time [2].

Policy consistency is about ensuring that packets fulfill some policy requirements, for example that all packets should go through a certain switch before reaching their destination, called a waypoint policy [18], which can be important in the context of firewalls and network security. The oldest policy consistency property is called per-packet consistency, which is introduced by Reitblatt et al. [23]. When flows are updated from using some initial paths to some different paths, per-packet consistency is the guarantee that each packet can only follow the initial or the final paths, but never an intermediary path. This simplifies an update process, because each packet only ever follows one set of forwarding rules. Our network model is based on the assumption that we can guarantee per-packet consistency with a process called two-phase commit [23], where we stamp each packet with information that determines whether it follow the old or the updated path. The primary downside to this method is that it can introduce unnecessary memory overhead on the switches because they need to know both the initial and final configuration, the memory requirements are effectively doubled.

The final property is capacity consistency, where the goal is to update switches in a sequence that avoids sending more data through links than they have capacity for, to avoid congestion. There are two main areas of study here, where the newest is about link latency, as it has been found that link latency can have an effect on congestion. For example, when a flow that initially uses high-latency links is updated to use low-latency links, this can cause the flow to congest itself [9]. Recent work [25] [26] [7] has found techniques to avoid conges-

tion from link latency, e.g. by using timed SDN. This paper does not consider the impact of link latency on link utilization during updates.

Instead, we study the other area of capacity consistency, which is about directly avoiding congestion by updating switches in the right ordering. There is already much literature on the subject [11][23][14][12][4][3], but there is still room for more research in this area, both in the theoretical formulation of the problems and the design of efficient algorithms to solve the problems. Typically, studies on capacity consistency use two-phase commits as the lowest-level operation, where whole data flows are updated as an atomic operation, thereby avoiding dealing with singular switch updates, and we also follow this paradigm.

Our main contribution is that we find a linear programming solution that can determine an optimal update sequence to minimize the maximum utilization, given a length bound for the update sequence. Second, we adapt and improve the simplification techniques from [19], which we later show has a significant effect on the computation time of finding a solution. We also come up with our own technique, where we remove some of the smallest flows, and instead directly calculate the worst-case utilization they could provide, giving us an overapproximation of the optimal maximum utilization. Thirdly, we find the complexity of our problem, and multiple variants of the problem. Finally, we implement a solver for our linear programming reduction of our problem, and show that by applying multiple techniques, we can efficiently find solutions for topologies with sizes ranging from 5 nodes to 197 nodes.

2 Network Model

We use a directed and capacitated graph as a representation of a network where the nodes represent switches, the edges represent connections between switches, and a capacity function on the edges represents bandwidth.

Definition 1 (Capacitated network). A *capacitated network* is a directed graph $G = (V, E, \mathit{capacity})$ where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, and $\mathit{capacity} : E \to \mathbb{N}_{\geq 1}$ is an assignment of each edge to its capacity.

We use paths to model how data travels through a capacitated network. Intuitively, a path is a connected sequence of edges.

Definition 2 (Path). A *path* is a finite or infinite ordered sequence of edges $(v_0, v_1), (v_1, v_2), (v_2, v_3), \ldots$ where $(v_i, v_{i+1}) \in E$ for all $i \geq 0$. The empty path is denoted ϵ, and the set of all paths is denoted $\mathcal{P}$. We write the number of times a link e is present in a path p as $\#e\langle p \rangle$.

A flow is a representation of a certain amount of data being sent from one specific switch to another. Since we are interested in network updates, we assign two paths to each flow, one, initial, path representing the edges the data travels through before the update, and the other, final, path representing the edges the data travels through after the update. The amount of data that is sent in a flow is called the traffic demand of the flow.

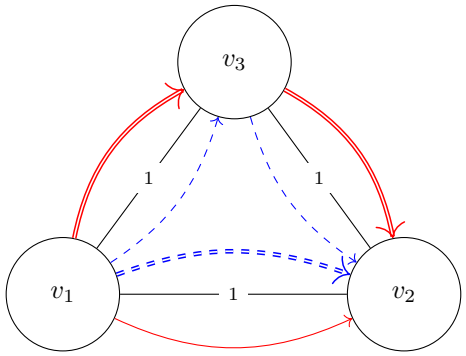


Fig. 1: A small traffic system. The circles v_1, v_2, and v_3 are nodes and the solid, black line between each node is an edge in both directions with a capacity of 1. There are two flows, so $F = \{f_1, f_2\}$, and f_1 is the red, solid arrows, while f_2 is the blue, dashed arrows. The initial path of each flow is the single-lined arrows, while the final paths are the double-lined arrows. Each flow has a traffic demand of 1.

Definition 3 (Traffic system). A *traffic system* is a 5-tuple $(G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$ where G is a capacitated network, F is a set of flows, the function $\mathit{initial} : F \to \mathcal{P}$ is an assignment of each flow to its initial path, the function $\mathit{final} : F \to \mathcal{P}$ is an assignment of each flow to its final path, and the function $\mathit{demand} : F \to \mathbb{N}_{\geq 1}$ is a mapping of each flow to its traffic demand.

For the rest of the paper, we fix a generic traffic system $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$.

One example of a traffic system is Figure 1. The capacitated network is a fully connected graph with three nodes, where each edge has a capacity of 1. There are two flows, f_1 and f_2 that each have a traffic demand of 1, where the initial path of f_1 is (v_1, v_2) and the final path of f_1 is $(v_1, v_3), (v_3, v_2)$, while the initial path of f_2 is $(v_1, v_3), (v_3, v_2)$ and the final path of f_2 is (v_1, v_2). The idea is that both flows initially send data through their respective initial paths, and we wish to update the flows such that the data is sent through their final paths.

The key concept we use for modelling updates is a split ratio. A split ratio determines how much of the data traffic of a flow is sent through the initial path and final path. If a split ratio of a flow is 0, then all the data travels through the initial path, and if it is 1, then all the data travels through the final path.

Definition 4 (Split ratio). A *split ratio* $\mathit{split} : F \to \{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$ over F maps each flow $f \in F$ to the fraction of the traffic demand that is carried through $\mathit{final}(f)$.

With a split ratio, we can determine the load on each edge, which is how much data is sent through that edge.

Definition 5 (Load). Let *split* be a split ratio over F. The *load* on a edge $e \in E$ is given by $load(split, e) = \sum_{f \in F} \big((1 - split(f)) \cdot \#e\langle initial(f)\rangle + split(f) \cdot \#e\langle final(f)\rangle\big) \cdot demand(f)$.

Note that a path can contain the same edge multiple times, which this notion of load takes into account.

If we divide the load on an edge with the capacity of that edge, we get the utilization of the edge. If the utilization is greater than 1, it means that more data is sent through the edge than it can process, and packets then risk being dropped.

Definition 6 (Utilization). Let *split* be a split ratio over F. We denote the *utilization* of a edge $e \in E$ by $utilization(split, e) = \frac{load(split,e)}{capacity(e)}$.

As an example, consider Figure 1. If the split ratio of f_1 is 0.5 and the split ratio of f_2 is 0, then only half of the traffic demand of f_1 and none of the traffic demand of f_2 is sent through (v_1, v_2), which means that the load on (v_1, v_2) is 0.5. Since (v_1, v_2) has a capacity of 1, the utilization of (v_1, v_2) is 0.5. Likewise, we find that the utilization of both (v_1, v_3) and (v_3, v_2) is 1.5. In this example, the greatest utilization of any edge is 1.5, and we call this the maximum utilization of this split ratio.

Definition 7 (Maximum utilization of a split ratio). Let *split* be a split ratio over F. The *maximum utilization* of *split* is given by $maximum\_utilization(split) = \max_{e \in E} utilization(split, e)$.

We use split ratios to model updates. Specifically, we model an update as a transition from one split ratio to another. As a split ratio is a function on all flows, an update can change the split ratio for multiple flows at once. However, real networks cannot reliably change routing protocols for multiple routers at the exact same time due to the difficulty of time synchronization on distributed systems. There is a risk that, at some moments during the update, some flows are updated, and other flows are not updated, so when we update the split ratio for multiple flows, we have to assume that the flows can be updated in any ordering. We accomplish this by finding the combination of updated- and unupdated flows that give the worst possible maximum utilization.

Definition 8 (Update utilization). Let *split* and *split*$'$ be two split ratios over F. The *update utilization* when moving from *split* to *split*$'$ is given by $update\_utilization(split, split') = \max_{s \in S} maximum\_utilization(s)$ where $S = \{split'' \mid \forall f \in F.\ (split''(f) = split(f) \vee split''(f) = split'(f))\}$.

We assume that we initially send all the data through the initial paths, and we want to perform updates so that all the data is sent through the final paths. An update sequence is a sequence of split ratios that does exactly this.

Definition 9 (Update sequence). An *update sequence* $\pi = split_0, \ldots, split_m$ over F is an ordered finite sequence of *split*-functions over F such that for all $f \in F$ we have that $split_0(f) = 0$ and $split_m(f) = 1$.

The key measure that we use is the maximum utilization of an update sequence. We calculate it by finding the greatest update utilization for any update in the update sequence.

Definition 10 (Maximum utilization of an update sequence). Let $\pi = split_0, \ldots, split_m$ be an update sequence over F. The *maximum utilization* of an update sequence is given by

$$maximum\_utilization(\pi) = \max_{0 \leq i < m} update\_utilization(split_i, split_{i+1}) .$$

Some update sequences are special cases. An update sequence is atomic if it only changes the split ratio for one flow in each iteration, is monotonic if the split ratio only increases over iterations, and has a fixed granularity if the split ratio only changes in increments of a specific fraction.

Definition 11 (Update sequence variants). Let $\pi = split_0, \ldots, split_m$ be an update sequence over F. The update sequence π

- is *monotonic* if $split_i(f) \leq split_j(f)$ for all $f \in F$ and all i, j s.t. $0 \leq i \leq j \leq m$,
- has a *fixed granularity* of $g \in \mathbb{Q}$ if for all $f \in F$ there exists an $h \in \mathbb{N}_{\geq 0}$ such that $split_i(f) = h \cdot g$ for all i where $0 \leq i \leq m$, and
- is *atomic* if for all i where $0 \leq i < m$ there exists an $f \in F$ such that $split_i(f) \neq split_{i+1}(f)$ and $split_i(f') = split_{i+1}(f')$ for all $f' \in F \setminus \{f\}$.

The update sequence decision problem is central problem for this paper. The problem is to determine whether there exists an update sequence such that it is at most length n and has a maximum utilization of at most u.

Definition 12 (Update sequence decision problem). Let $n \in \mathbb{N}_{\geq 0}$ and $u \in \mathbb{Q}_{\geq 0}$. The *update sequence decision problem* is the question of whether there exists an update sequence $\pi = split_0, \ldots, split_m$ over F such that $m \leq n$ and $maximum\_utilization(\pi) \leq u$.

If we constrict π to be monotonic, we call it the *monotonic* update sequence decision problem, and if we constrict π to have a granularity of g for some $g \in \mathbb{Q}$, we call it the *fixed granularity* update sequence decision problem. If instead we constrict π to be atomic, we call it the *atomic* update sequence decision problem.

To distinguish the update sequence decision problem with no constrictions from the other problems, we call it the general update sequence decision problem when it would otherwise be unclear.

In Definition 12, we have an n that can bound the length of the update sequence, because not all networks have an optimal update sequence in terms of minimizing the maximum utilization, as we can see in Lemma 1.

Lemma 1. *There exists a traffic system such that for any update sequence π, there exists another update sequence π' such that $maximum\_utilization(\pi') < maximum\_utilization(\pi)$.*

Proof. Let the traffic system $\mathcal{T} = (G = (V, E, capacity), F, initial, final, demand)$ be as defined by Figure 1. Assume by contradiction that there exists an update sequence $\pi = split_0, \ldots, split_m$ over F such that there is no update sequence π' where $maximum\_utilization(\pi') < maximum\_utilization(\pi)$. We know that $maximum\_utilization(split_0) = 1$ and $maximum\_utilization(split_m) = 1$, and we know that there has to be some n such that $0 \leq n < m$ where $update\_utilization(split_n, split_{n+1}) > 1$, because increasing the split ratio for either flow by any fraction increases utilization above 1, which in turn means that $maximum\_utilization(\pi) > 1$. This means that we can define an $\epsilon \in \mathbb{Q}_{>0}$ such that $maximum\_utilization(\pi) = 1 + \epsilon$.

We can construct an update sequence with a maximum utilization of $1+\frac{\epsilon}{2}$. Let $\pi' = split'_0, \ldots, split'_l$ be an update sequence where $l = \frac{2}{\epsilon}$ such that $split'_i(f) = \frac{\epsilon}{2} \cdot i$ for all $f \in F$ and for all i such that $0 \leq i \leq l$. We know this is an update sequence because it is finite in length, $split'_0(f) = 0$, and $split'_l(f) = 1$ for all $f \in F$. The maximum utilization of all split ratios in π' is 1, because all split ratios in π' split the demand equally. When the load on each link is 1, updating in increments of $\frac{\epsilon}{2}$ can only increase the load with at most $\frac{\epsilon}{2}$, so $update\_utilization(split'_i, split'_{i+1}) = 1+\frac{\epsilon}{2}$ for all i such that $0 \leq i < l$. This means that $maximum\_utilization(\pi') = \frac{\epsilon}{2}$ which contradicts our assumption. □

3 Flow Pruning

We can simplify a traffic system by removing certain flows, while guaranteeing no change in maximum utilization. The general idea is that we can statically detect that some flows cannot change the maximum utilization, and can thus be updated at any time with no impact. We provide an algorithm for pruning flows, and we argue why we can safely do so.

3.1 Threshold

The central concept for pruning flows is the threshold, which is the utilization of the edge with the greatest utilization where either (i) the split ratio for each flow is 0 or (ii) the split ratio for each flow is 1. We can find it using Algorithm 1.

We use this threshold as a lower bound for the maximum utilization, since any update sequence assigns 0 to every flow and 1 to every flow at some point, so we know that the maximum utilization of any update sequence will have at least the same value as the threshold.

Algorithm 1 Find Threshold

Input: A traffic system $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$
Output: A threshold t

1: $t \leftarrow 0$
2: **for** $e \in E$ **do**
3: $\quad a \leftarrow 0$
4: $\quad b \leftarrow 0$
5: $\quad$ **for** $f \in F$ **do**
6: $\quad\quad a \leftarrow a + \frac{\#e\langle \mathit{initial}(f)\rangle \cdot \mathit{demand}(f)}{\mathit{capacity}(e)}$
7: $\quad\quad b \leftarrow b + \frac{\#e\langle \mathit{final}(f)\rangle \cdot \mathit{demand}(f)}{\mathit{capacity}(e)}$
8: $\quad$ **end for**
9: $\quad$ **if** $a > t$ **then**
10: $\quad\quad t \leftarrow a$
11: $\quad$ **end if**
12: $\quad$ **if** $b > t$ **then**
13: $\quad\quad t \leftarrow b$
14: $\quad$ **end if**
15: **end for**
16: **return** t

Lemma 2. *The threshold t found by Algorithm 1 is equal to* $\max(\mathit{maximum\_utilization}(\mathit{split}), \mathit{maximum\_utilization}(\mathit{split}'))$ *where* $\mathit{split}(f) = 0$ *and* $\mathit{split}'(f) = 1$ *for all* $f \in F$.

Proof. Obvious by analysis of Algorithm 1. □

If we only care about deciding whether there exists a solution under a certain level of maximum utilization u, then we can change line 1 in Algorithm 1 to $t \leftarrow u$. Then, the threshold is at least u, even if there is no edge that has a utilization of u in initial or final.

3.2 Flow Pruning Algorithm

When we have found a threshold for a traffic system, we can now safely remove flows where no edge from either their initial paths or their final paths can in any way achieve a utilization that is at least as the same value as the threshold. We present the algorithm for flow pruning in Algorithm 2.

At line 6, the fraction $\frac{\max(\#e\langle \mathit{initial}(f)\rangle, \#e\langle \mathit{final}(f)\rangle) \cdot \mathit{demand}(f)}{\mathit{capacity}(f)}$ gives the worst-case utilization that f can contribute to e, and since we do that for all flows, a finds the worst-case utilization of e. At line 9, we add a flow f to F', which means that we keep the flow, if either the initial or the final path of f contains e and if $a \geq t$, meaning that the worst-case utilization of e is greater than the threshold. What we end up with is an F' which only has the flows that can affect the maximum utilization of an update sequence.

An example of the flow pruning process can be found in Figure 2.

Algorithm 2 Flow Pruning

Input: A traffic system $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$
Output: A reduced traffic system $\mathcal{T}' = (G, F', \mathit{initial}, \mathit{final}, \mathit{demand})$

1: $t \leftarrow$ the threshold from Algorithm 1 applied to $\mathcal{T}$
2: $F' \leftarrow \emptyset$
3: **for** $e \in E$ **do**
4: $\quad a \leftarrow 0$
5: $\quad$ **for** $f \in F$ **do**
6: $\quad\quad a \leftarrow a + \frac{\max(\#e\langle \mathit{initial}(f)\rangle, \#e\langle \mathit{final}(f)\rangle) \cdot \mathit{demand}(f)}{\mathit{capacity}(e)}$
7: $\quad$ **end for**
8: $\quad$ **if** $a \geq t$ **then**
9: $\quad\quad$ **for** $f \in F$ **do**
10: $\quad\quad\quad$ **if** $\#e\langle \mathit{initial}(f)\rangle \geq 1$ or $\#e\langle \mathit{final}(f)\rangle) \geq 1$ **then**
11: $\quad\quad\quad\quad F' \leftarrow F' \cup \{f\}$
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end for**
14: $\quad$ **end if**
15: **end for**
16: $\mathcal{T}' = (G, F', \mathit{initial}, \mathit{final}, \mathit{demand})$
17: **return** $\mathcal{T}'$

Theorem 1. *Let $\mathcal{T}$ and $\mathcal{T}'$ be traffic systems where $\mathcal{T}'$ is the output of Algorithm 2 with $\mathcal{T}$ as input. There exists an update sequence of length n with a maximum utilization of u for $\mathcal{T}$ if and only if there exists an update sequence of length n with a maximum utilization of u for $\mathcal{T}'$.*

Proof. Let $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$ and $\mathcal{T}' = (G = (V, E, \mathit{capacity}), F', \mathit{initial}, \mathit{final}, \mathit{demand})$ be two traffic systems such that $\mathcal{T}'$ is the output of Algorithm 2 applied to $\mathcal{T}$, and let $u \in \mathbb{Q}_{\geq 0}$ and $n \in \mathbb{N}_{\geq 2}$. We first prove that if there exists an update sequence of length n with a maximum utilization of u for $\mathcal{T}$, then there exists an update sequence of length n with a maximum utilization of u for $\mathcal{T}'$.

Assume that there exists an update sequence $\pi = \mathit{split}_0, \ldots, \mathit{split}_{n-1}$ for $\mathcal{T}$ with a maximum utilization of u. On line 1 in Algorithm 2, we assign the threshold from Algorithm 1 to t, which we know from Lemma 2 is given by $\max(\mathit{maximum\_utilization}(\mathit{split}_0), \mathit{maximum\_utilization}(\mathit{split}_{n-1}))$. By definition of maximum utilization of an update sequence, we know that $t \leq u$.

As it iterates through the edges, the for-loop from line 3 to line 15 in Algorithm 2 adds a flow $f \in F$ to F' if there is an edge in $\mathit{initial}(f)$ or $\mathit{final}(f)$ such that the edge has a worst-case utilization of at least the threshold. The worst-case utilization is represented by the variable a, which we see in lines 4 to 7.

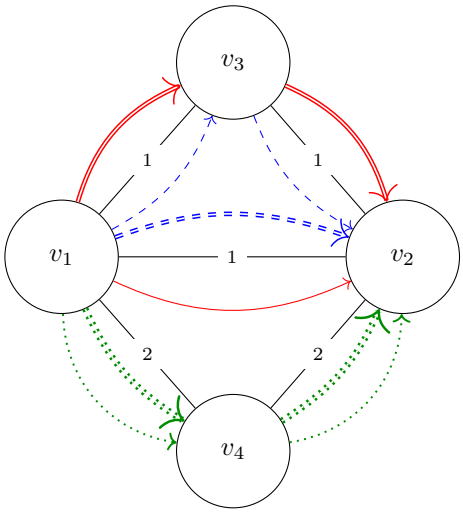
After iterating through all the edges, we have that $F' \subseteq F$, and for every $f \in F \cap \overline{F'}$, we know that every edge in $\mathit{initial}(f)$ or $\mathit{final}(f)$ has a worst-case utilization w_f such that $w_f < t$, which means that $w_f < u$. The maximum

utilization of an update sequence is, by definition, based solely on the edge $e_\top$ with the greatest update utilization for any pair of split ratios, and we know that the worst-case utilization of $e_\top$ is at least u, so every flow that contains $e_\top$ must be in F'.

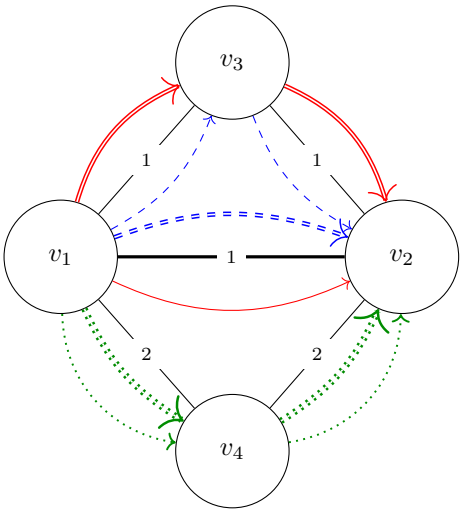
We construct an update sequence $\pi' = split'_0, \ldots, split'_{n-1}$ such that $split'_i(f) = split_i(f)$ for all $f \in F'$ and for all i such that $0 \leq i < n$. Since all flows that use $e_\top$ are in F', and since we know that it is exactly these flows that give $e_\top$ an update utilization of u in π, then, since π' assigns the same split ratios to the those flows, $e_\top$ also has a utilization of u in π'. And because of how we construct π', and because $F' \subseteq F$, we know that the maximum utilization of π' cannot be greater than the maximum utilization of π. Thus, the maximum utilization of π' must be equal to the maximum utilization of π.

Now we prove that if there exists an update sequence of length n with a maximum utilization of u for $\mathcal{T}'$, then there exists an update sequence of length n with a maximum utilization of u for $\mathcal{T}$. Assume that there exists an update sequence $\pi' = split'_0, \ldots, split'_{n-1}$ for $\mathcal{T}'$ with a maximum utilization of u. We construct an update sequence $\pi = split_0, \ldots, split_{n-1}$ such that $split_i(f) = split'_i(f)$ for all $f \in F'$ and for all i such that $0 \leq i < n$, and $split_0(f) = 0$ and $split_i(f) = 1$ for all $f \in F \cap \overline{F'}$ and for all i such that $0 < i < n$.

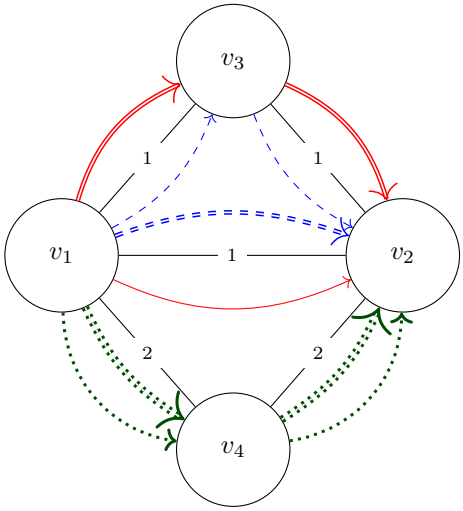
By the same logic as before, we know that every flow in $F \cap \overline{F'}$ only uses edges where the worst-case utilization is less than t, which is in turn at most u. Therefore, we know that the edge $e_\top$ that gives the greatest maximum utilization u in F' is not affected by flows in $F \cap \overline{F'}$ in F either, so since we construct π with the same split ratios as in π' for the flows in F', and only the flows in F' affect the utilization in $e_\top$, we know that, if $e_\top$ is still the edge that gives the greatest maximum utilization in π, then the maximum utilization of π is at least u. Since none of the flows in $F \cap \overline{F'}$ can affect an edge with a greater worst-case utilization than $e_\top$, $e_\top$ must be the edge in F that gives the greatest maximum utilization. Thus, the maximum utilization of π must be equal to the maximum utilization of π'. □



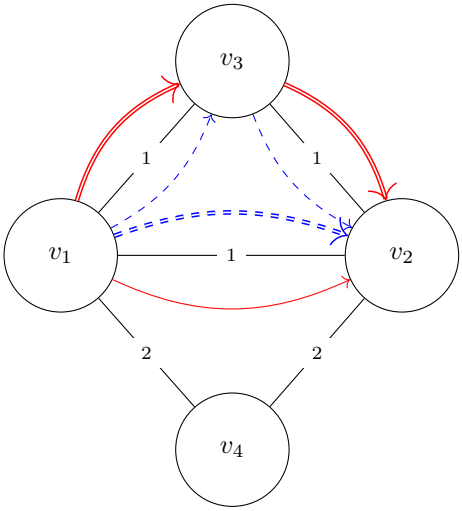
(a) Here, we have a similar setup to Figure 1, but with an extra node v_4 and an extra, green, flow. Note that the edges between v_4 and its neighbors have a capacity of 2.



(b) The first step of pruning flows is finding the threshold. The greatest utilization of any edge when all flows follow their initial or final paths is 1, which can be found in (v_1, v_2), so the threshold is 1.



(c) Now we find all flows where all edges in its paths have a worst-case utilization of less than 1, here f_3.



(d) Finally, we remove the flows that we found in the last step, and this is the traffic system that we get.

Fig. 2: Example of flow pruning. There are three flows, so $F = \{f_1, f_2, f_3\}$, which are solid red, dashed blue, and dotted green arrows, respectively. The initial paths of the flows are single-lined, while the final paths are double-lined.

4 Problem Complexity

Now that we have defined the different variants of the update sequence decision problem, we look at their respective complexity classes. We find an overview of the complexity results in Table 1.

4.1 Complexity of the atomic and fixed granularity variants

First, we look at the complexity of the atomic and fixed granularity problems.

Theorem 2. *The fixed granularity update sequence decision problem and the atomic update sequence decision problem are NP-hard.*

Proof. We prove this by polynomial-time reduction from the NP-complete partition problem [1]. Let $S = \{s_1, \ldots, s_n\}$ be a set of positive integers with at least 2 elements and $S_\sum$ be the sum of all numbers in S. The partition problem is to determine whether there exists two disjoint sets $S_1 \subseteq S$ and $S_2 \subseteq S$ such that $S_1 \bigcup S_2 = S$ and $\sum_{s_1 \in S_1} s_1 = \sum_{s_2 \in S_2} s_2 = \frac{S_\sum}{2}$.

We use the network described in Figure 3 where the capacity of each link is $S_\sum$. We construct a set of flows $F = \{f_i \mid s_i \in S\} \bigcup \{b\}$. The function *initial* assigns path (v_1, v_2) to all f_i and the path $(v_1, v_3), (v_3, v_2)$ to b, while *final* assigns the path $(v_1, v_3), (v_3, v_2)$ to f_i and the path (v_1, v_2) to b. Finally, $demand(f_i) = s_i$ for all f_i, and $demand(b) = \frac{S_\sum}{2}$. From this, we make a fixed granularity update sequence decision problem where $u = 1$, $n = 4$, and the update sequence has a granularity $g = 1$, and we make an atomic update sequence decision problem where $u = 1$ and $n = |F| + 1$.

We shall now see that the partition problem has a solution if and only if the corresponding fixed granularity update sequence decision problem has a solution. First, assume that the partition problem has a solution. From this assumption, there exists a subset S_1 of S such that $\sum_{s_1 \in S_1} s_1 = \frac{S_\sum}{2}$ and another subset $S_2 = \{s \in S \mid s \notin S_1\}$. Because of how we defined the network, we can make a corresponding partition of F into $F_1 = \{f_i \mid s_i \in S_1\}$ and $F_2 = \{f_i \mid s_i \in S_2\}$. As $demand(b) = \frac{S_\sum}{2}$ and every edge has a capacity of $S_\sum$, we know we can immediately update every flow in F_1 since the sum of demands for the flows in F_1 is equal to $\frac{S_\sum}{2}$. With this knowledge, we can construct an update sequence $split_0, split_1, split_2, split_3$ where $split_1$ updates all the flows in F_1 which gives sufficient room that we can update b in $split_2$ and then the rest of the flows in $split_3$. As this update sequence is a solution to the fixed granularity update sequence decision problem, we see that if we assume that the partition problem has a solution, then the fixed granularity update sequence decision problem also has a solution.

To show that it also holds in the other direction, assume that the fixed granularity update sequence decision problem has a solution. This assumption means that there exists an update sequence π of at most length 4 that solves the

Problem variant	Complexity class	Proven in
General	in P	Theorem 5
Monotonic	in P	Theorem 5
Atomic	NP-complete	Theorem 2 and 3
Fixed granularity	NP-hard and in PSPACE	Theorem 2 and 4

Table 1: Overview of the complexity results of the variations of the update sequence decision problem.

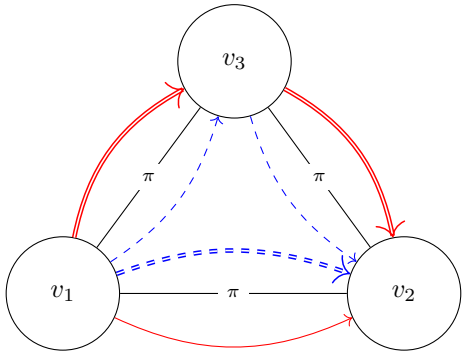


Fig. 3: The traffic system we construct a partition problem. The circles v_1, v_2, and v_3 are nodes and the black line between each node is an edge in both directions with a capacity of $S_\sum$. There is a flow for each $s \in S$ and an additional flow b, so $F = \{f_1, f_2, \ldots, f_{|S|}, b\}$. The path for each flow except b follows the red, solid arrows, and b follows the dashed, blue arrows. The initial paths are single-lined, while the final paths are double-lined. The traffic demand for $f_i = s_i$ for all i where $1 \leq i \leq |S|$, and the traffic demand for b is $\frac{S_\sum}{2}$

problem. As the first split ratio is 0 for every flow and since the flows in F initially use the whole capacity of the edge (v_1, v_2), b cannot be updated in the second split ratio unless $S_\sum = 0$ which would trivially contradict our assumption. As b can at the earliest be updated in the third split ratio, not all flows in F can be updated before fourth split ratio, as there is insufficient capacity before b is updated. Therefore, π must have a length of exactly 4.

Since b is updated in the third split ratio, we know that there must exist an $F' \subset F$ that is updated in the second split ratio such that the sum of all demands for the flows in F' is $\frac{S_\sum}{2}$, and because of how we define the network, there must exist a set $S' = \{s_i \in S \mid f_i \in F'\}$ such that $\sum_{s' \in S'} s' = \frac{S_\sum}{2}$. We can then define a set $S_1 = S'$ and another set $S_2 = \{s \in S \mid s \notin S_1\}$ which solve the partition problem, so if the fixed granularity update sequence decision problem has a solution, then the partition problem also has a solution.

We use the same techniques for the atomic update sequence decision problem. Because $n = |F| + 1$, there is exactly enough room for each flow to be updated

once, and since every flow needs to be updated, we effectively get a granularity of 1. □

We can also find the upper bound of the atomic problem by reduction to integer programming.

Theorem 3. *The atomic update sequence decision problem is in NP.*

Proof. By polynomial-time reduction to integer programming [15]. Assume an update sequence decision problem where the solution can at most be of length $n \in \mathbb{N}_{\geq 0}$ and where the maximum utilization of the solution can be at most $u \in \mathbb{Q}_{\geq 0}$. We define $|F| \cdot n$ variables, each of the form $x_f^i \in \mathbb{R}$.

$$\begin{matrix} x_{f_1}^0 & x_{f_1}^1 & \dots & x_{f_1}^{n-1} \\ x_{f_2}^0 & x_{f_2}^1 & \dots & x_{f_2}^{n-1} \\ \dots & \dots & \dots & \dots \\ x_{f_{|F|}}^0 & x_{f_{|F|}}^1 & \dots & x_{f_{|F|}}^{n-1} \end{matrix}$$

This set of variables is equivalent to an update sequence. A split ratio $split_i$ assigns a value to each flow, which corresponds to an array $x_{f_1}^i, x_{f_2}^i, \dots, x_{f_{|F|}}^i$ of which we have n.

Furthermore, we define another set of variables, each of the form $y_f^i \in \{0, 1\}$

$$\begin{matrix} y_{f_1}^0 & y_{f_1}^1 & \dots & y_{f_1}^{n-1} \\ y_{f_2}^0 & y_{f_2}^1 & \dots & y_{f_2}^{n-1} \\ \dots & \dots & \dots & \dots \\ y_{f_{|F|}}^0 & y_{f_{|F|}}^1 & \dots & y_{f_{|F|}}^{n-1} \end{matrix}$$

We use y_f^i to encode atomicity, so if $y_f^i = 1$, then that means that x_f^{i+1} can be different from x_f^i, and if $y_f^i = 0$, then $x_f^{i+1} = x_f^i$.

Finally, we also define the variable α which we use to represent the overall maximum utilization, and this is the value that we optimise for. We reduce the atomic update sequence decision problem to:

Minimize

$$\alpha \tag{1}$$

such that

$$\text{for all } f \in F$$
$$x_f^0 = 0,\ x_f^{n-1} = 1 \tag{2}$$

$$\text{for all } f \in F \text{ and } i \text{ such that } 0 \leq i \leq n-1$$
$$0 \leq x_f^i \leq 1,\ y_f^i \in \{0,1\} \tag{3}$$

$$\text{for all } i \text{ such that } 0 \leq i \leq n-1$$
$$\sum_{f \in F} y_f^i = 1 \tag{4}$$

$$\text{for all } f \in F \text{ and } i \text{ such that } 0 \leq i \leq n-2$$
$$-y_f^i \leq x_f^i - x_f^{i+1} \leq y_f^i \tag{5}$$

$$\text{for all } e \in E \text{ and } i \text{ such that } 0 \leq i \leq n$$
$$\alpha \geq \sum_{f \in F} \frac{((1 - x_f^i) \cdot \#e\langle \mathit{initial}(f)\rangle + x_f^i \cdot \#e\langle \mathit{final}(f)\rangle) \cdot \mathit{demand}(f)}{\mathit{capacity}(e)} \tag{6}$$

The atomic update sequence decision problem has a solution if and only if $\alpha \leq u$. To prove this, we first assume that the atomic update sequence decision problem has a solution. From this assumption, there must be an atomic update sequence $\pi = \mathit{split}_0, \ldots, \mathit{split}_m$ of at most length n such that $\mathit{maximum\_utilization}(\pi) \leq u$. We assign the value $\mathit{split}_i(f)$ to x_f^i for all $f \in F$ where $0 \leq i \leq m$ and we assign the value 1 to x_f^i for all $f \in F$ where $m < i \leq n-1$. Additionally, when $\mathit{split}_i(f) \neq \mathit{split}_{i+1}(f)$, we assign the value 1 to y_f^i, and otherwise we assign the value 0 to y_f^i.

Since π is atomic, (4) and (5) are clearly upheld by the y_f^i we just defined. By definition of a split ratio, (3) is also upheld, while (2) is upheld by definition of an update sequence. The inequality (6) corresponds to the load on each edge, so this inequality says that α should be greater than or equal to the greatest load on any edge for any split ratio in π. Since the set S in Definition 8 is a singleton for any pair of split ratios in an atomic update sequence, the maximum utilization of π is also the greatest load on any edge for any split ratio in π, so (6) is also upheld. Because the integer programming problem minimizes α, we know that $\alpha = \mathit{maximum\_utilization}(\pi)$, so if the atomic update sequence decision problem has a solution, then the integer programming problem also has a solution such that $\alpha \leq u$.

In the other direction, assume that the integer programming problem has a solution such that $\alpha \leq u$. Let $\pi = \mathit{split}_0, \ldots, \mathit{split}_{n-1}$ be an update sequence

such that $split_i(f) = x_f^i$ for all $f \in F$ where $0 \leq i \leq n-1$. Because of (4) and (5), π must be atomic, because only x_f^i can be different from x_f^{i+1} for one $f \in F$, which corresponds to our notion of atomicity. The inequality (3) enforces the domain of a split ratio, and (2) ensures that the first and last split ratio assigns 0 and 1 to all flows. As previously mentioned, the maximum utilization of π is the greatest load on any edge for any split ratio, but since (6) ensures that α is greater than the greatest load on any edge for any split ratio, we know that $maximum\_utilization(\pi) \leq \alpha$, so if the integer programming problem has a solution such that $\alpha \leq u$, then there exists an update sequence that solves the atomic update sequence decision problem. □

Since the atomic problem is NP-hard and in NP, the problem is NP-complete.

Corollary 1. *The atomic update sequence decision problem is NP-complete.*

Proof. Follows from Theorem 2 and Theorem 3. □

Now that we found the upper bound for the complexity of the atomic problem, we only need the upper bound for the fixed granularity problem.

Theorem 4. *The fixed granularity update sequence decision problem is in PSPACE.*

Proof. Assume a fixed granularity update sequence decision problem with a fixed granularity of g where the solution can at most be of length $n \in \mathbb{N}_{\geq 0}$ and where the maximum utilization of the solution can be at most $u \in \mathbb{Q}_{\geq 0}$. To see that this problem is in PSPACE, we write a nondeterministic algorithm that can solve the problem while using polynomial space, which we find in Algorithm 3. In Algorithm 3, we can nondeterministically guess split ratios, because by definition of fixed granularity, a split ratio can only assign a finite set of values to each flow. For the algorithm to accept, there needs to exist a sequence of split ratios such that

- The first split ratio for each flow is 0,
- The last split ratio for each flow is 1,
- The sequence has a fixed granularity of g,
- The sequence contains at most n split ratios, and
- Each pair of split ratios $split_i, split_{i+1}$ has an update utilization below u.

This corresponds exactly to our notion of a solution for the fixed granularity update sequence decision problem, so this algorithm can clearly decide whether a solution exists. We note that the algorithm only uses polynomial space, since it only needs to remember at most 2 split ratios at once, and can forget the old ones as it iterates through them. Hence, we see that the fixed granularity update sequence decision problem is in PSPACE. □

Algorithm 3 Nondeterministic Solver for the Fixed Granularity Problem

Input: A traffic system $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$, a fixed granularity g, a bound on the length n, and a utilization limit u

Output: **accept** if a solution exists, otherwise **reject**

$x \leftarrow \mathit{split}_0$ ▷ split_0 assigns 0 to all flows
$i \leftarrow 1$
while $i < n$ **do**
  nondeterministically guess a split ratio y with a fixed granularity of g
  if $\mathit{update\_utilization}(x, y) \geq u$ **then**
    reject
  end if
  if $y(f) = 1$ for all $f \in F$ **then**
    accept
  end if
  $x \leftarrow y$
  $i \leftarrow i + 1$
end while
reject

4.2 Complexity of monotonic and general variants

Finally, we look at which complexity class the monotonic and general problems belong to.

Theorem 5. *The update sequence decision problem and the monotonic update sequence decision problem are in P.*

Proof. By polynomial-time reduction to P-complete linear programming [21]. First, we look at the (non-monotonic) update sequence decision problem. Like in the proof for Theorem 3, we use x_f^i to represent split ratios.

$$\begin{matrix} x_{f_1}^0 & x_{f_1}^1 & \dots & x_{f_1}^{n-1} \\ x_{f_2}^0 & x_{f_2}^1 & \dots & x_{f_2}^{n-1} \\ \dots & \dots & \dots & \dots \\ x_{f_{|F|}}^0 & x_{f_{|F|}}^1 & \dots & x_{f_{|F|}}^{n-1} \end{matrix}$$

We have another set of variables $y_{f,e}^i$, which we use to find the update utilization from Definition 8. For each edge, we have $|F| \cdot n - 1$ variables, which can be visualized as

$$\begin{matrix} y_{f_1,e_1}^0 & y_{f_1,e_1}^1 & \dots & y_{f_1,e_1}^{n-2} \\ y_{f_2,e_1}^0 & y_{f_2,e_1}^1 & \dots & y_{f_2,e_1}^{n-2} \\ \dots & \dots & \dots & \dots \\ y_{f_{|F|},e_1}^0 & y_{f_{|F|},e_1}^1 & \dots & y_{f_{|F|},e_1}^{n-2} \end{matrix} \quad \dots \quad \begin{matrix} y_{f_1,e_{|E|}}^0 & y_{f_1,e_{|E|}}^1 & \dots & y_{f_1,e_{|E|}}^{n-2} \\ y_{f_2,e_{|E|}}^0 & y_{f_2,e_{|E|}}^1 & \dots & y_{f_2,e_{|E|}}^{n-2} \\ \dots & \dots & \dots & \dots \\ y_{f_{|F|},e_{|E|}}^0 & y_{f_{|F|},e_{|E|}}^1 & \dots & y_{f_{|F|},e_{|E|}}^{n-2} \end{matrix}$$

The key principle is to use each $y^i_{f,e}$ to find the utilization of edge e given either x^i_f or x^{i+1}_f, whichever gives the greater utilization. By summation of $y^i_{f,e}$ over f, we get a value that is the combination of updated- and unupdated flows that gives the greatest utilization in the edge e and the transition from i to $i+1$, which corresponds to update utilization if we maximize across all edges. We minimize our last variable, α, which is equivalent to the maximum utilization of π.

Minimize

$$\alpha \tag{7}$$

such that

$$\begin{gathered}\text{for all } f \in F \\ x^0_f = 0,\ x^{n-1}_f = 1\end{gathered} \tag{8}$$

$$\begin{gathered}\text{for all } i \text{ where } 0 \le i \le n-1 \\ 0 \le x^i_f \le 1\end{gathered} \tag{9}$$

$$\begin{gathered}\text{for all } e \in E \text{ and } i \text{ where } 0 \le i \le n-2 \\ \alpha \ge \sum_{f \in F} y^i_{f,e}\end{gathered} \tag{10}$$

$$\begin{gathered}\text{for all } f \in F \text{ and } e \in E \text{ and for all } i \text{ where } 0 \le i \le n-2 \\ y^i_{f,e} \ge \frac{\left((1-x^i_f)\cdot \#e\langle \mathit{initial}(f)\rangle + x^i_f \cdot \#e\langle \mathit{final}(f)\rangle\right)\cdot \mathit{demand}(f)}{\mathit{capacity}(e)}\end{gathered} \tag{11}$$

$$\begin{gathered}\text{for all } f \in F \text{ and } e \in E \text{ and for all } i \text{ where } 0 \le i \le n-2 \\ y^i_{f,e} \ge \frac{\left((1-x^{i+1}_f)\cdot \#e\langle \mathit{initial}(f)\rangle + x^{i+1}_f \cdot \#e\langle \mathit{final}(f)\rangle\right)\cdot \mathit{demand}(f)}{\mathit{capacity}(e)}\end{gathered} \tag{12}$$

To prove that the reduction holds, we prove that the update sequence decision problem has a solution if and only if this linear programming problem has a solution such that $\alpha \le u$. We first assume that the update sequence decision problem has a solution $\pi = \mathit{split}_0, \ldots, \mathit{split}_{n-1}$ such that $\mathit{maximum\_utilization}(\pi) \le u$. Let each x^i_f correspond to $\mathit{split}_i(f)$. Then, (8) and (9) hold by definition of an update sequence. In conjunction, (11) and (12) constrain each $y^i_{f,e}$ to be at least the load of $\mathit{split}_i(f)$ and $\mathit{split}_{i+1}(f)$ on edge e. If we assume that each $y^i_{f,e}$ is minimized, then $\max\limits_{e \in E} \sum\limits_{f \in F} y^i_{f,e}$ for some iteration i gives exactly the update utilization of split_i and split_{i+1}. Since the objective function minimizes α, inequality (10) means that $\alpha = \max\limits_{0 \le i \le n-2} \max\limits_{e \in E} \sum\limits_{f \in F} y^i_{f,e}$, which then corresponds

exactly to the maximum utilization of π. However, we cannot assume that each y_f^i is minimized, but we *can* assume that the greatest $y_{f_1}^i, y_{f_2}^i, \ldots, y_{f_{|F|}}^i$ when summed over f is minimized, because that minimizes our objective function α, so we still get that $\alpha = \max_{0 \leq i \leq n-2} \max_{e \in E} \sum_{f \in F} y_{f,e}^i$. Therefore, if the update sequence decision problem has a solution, then the linear programming problem has a solution such that $\alpha \leq u$.

To prove it in the other direction, assume that the linear programming problem has a solution such that $\alpha \leq u$. Let $\pi = \mathit{split}_0, \ldots, \mathit{split}_{n-1}$ be an update sequence such that $\mathit{split}_i(f) = x_f^i$. We know that π follows the definition of an update sequence because (9) corresponds to the co-domain of the split functions and (8) means that $x_f^0 = 0$ and $x_f^{n-1} = 1$ for each $f \in F$. The update utilization of any pair of split ratios split_i and split_{i+1} is at most α because of (10), (11), and (12), since (11) and (12) constrain any y_f^i to be at least the utilization of edge e given x_f^i and x_f^{i+1}, and (10) constrains α to be at least $\sum_{f \in F} y_{f,e}^i$ for all edges and iterations except $n-1$, which means that the minimal α is exactly the greatest update utilization of any split_i and split_{i+1}. As this is also the definition of maximum utilization for an update sequence, we know that $\alpha = \mathit{maximum\_utilization}(\pi)$, so if there exists a solution for the linear programming problem such that $\alpha < u$, then there also exists a solution for the update sequence decision problem.

For the monotonic update sequence decision problem, we use the same technique, but we add the inequality

$$\text{for all } f \in F \text{ and for all } i \text{ where } 0 \leq i \leq n-2$$
$$x_f^i \leq x_f^{i+1} \tag{13}$$

This corresponds exactly to our notion of monotonicity, since it means that the x_f^i-variables can only increase over iterations. □

5 Towards a Practical Linear Programming Solution

In Theorem 5, we defined a linear programming reduction from the general update sequence decision problem. By applying an LP-solver to the model, we can find a solution to the update sequence decision problem. However, the number of variables in the linear programming model increases with the number of flows, the number of edges, and the length bound for the solution, so especially for larger networks, we need to reduce the number of variables to make the linear programming solution practical.

To this end, we have multiple techniques that we can use. The flow pruning from Section 3.2 is one such technique, but specifically for the linear programming model, we can also safely prune some of the edge constraints, and we can also remove some of the smallest flows by trading precision for computation time.

Algorithm 4 Edge Pruning

Input: A traffic system $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$
Output: A pruned set of edges E'

$t \leftarrow$ the threshold from Algorithm 1 applied to $\mathcal{T}$
$E' \leftarrow \emptyset$
for $e \in E$ **do**
    $a \leftarrow 0$
    for $f \in F$ **do**
        $a \leftarrow a + \frac{\max(\#e\langle \mathit{initial}(f)\rangle, \#e\langle \mathit{final}(f)\rangle) \cdot \mathit{demand}(f)}{\mathit{capacity}(f)}$
    end for
    if $\max(\#e\langle \mathit{initial}(f)\rangle, \#e\langle \mathit{final}(f)\rangle) \geq 1$ and $a \geq t$ **then**
        $E' \leftarrow E' \cup \{e\}$
    end if
end for
return E'

5.1 Edge Pruning

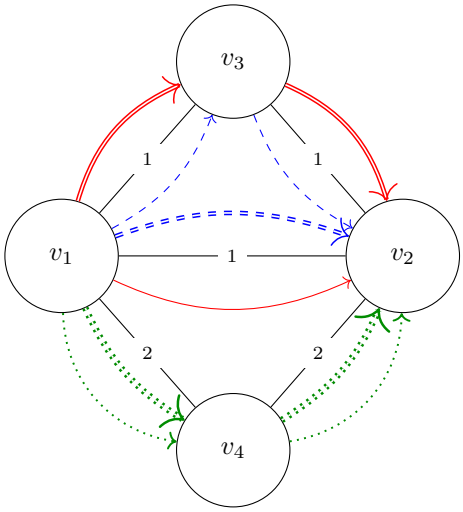
Some edges cannot affect the maximum utilization, and we can find those edges by applying the same technique we use for flow pruning in Section 3.2. If an edge cannot, under any circumstances, achieve a utilization of at least the value of the threshold, then we can remove the constraints on that edge without affecting the maximum utilization. We present the algorithm for edge pruning in Algorithm 4.

When we have applied Algorithm 4 to a traffic system $\mathcal{T}$, we then use the new set of edges E' for the constraints in the linear programming model, because the edges that are not in E' do not matter when looking for the maximum utilization.

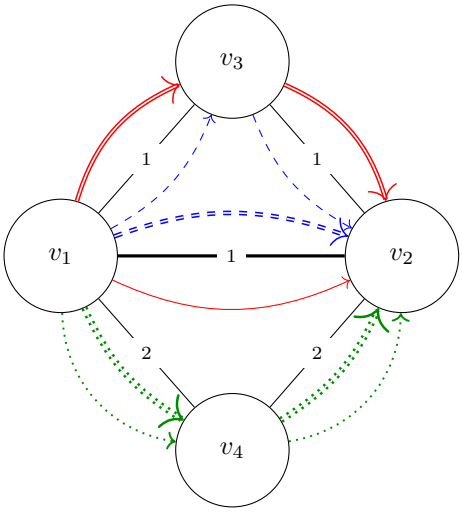
An example of the edge pruning process can be found in Figure 4.

Theorem 6. *Let $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$ be a traffic system and E' be the output of Algorithm 4 with $\mathcal{T}$ as input. The linear programming model from Theorem 5 gives the same optimal solution α whether E or E' is used for the constraints* (10)*,* (11)*, and* (12)*.*

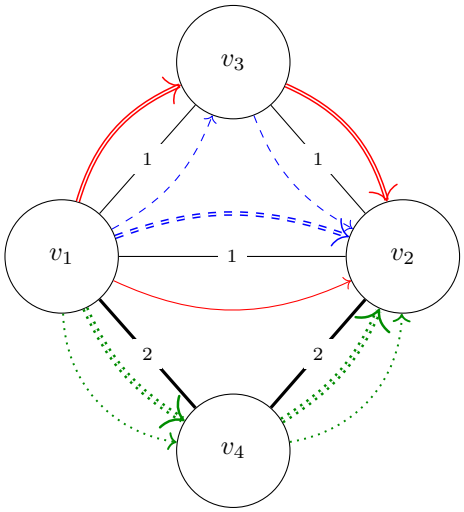
Proof. Using the same logic as the proof for Theorem 1, we see from analysis of Algorithm 4 that the pruned edges always have a utilization that is less that α. Since the value of α is found from the edge that gives the greatest maximum utilization, removing the constraints that use the pruned edges can never change α. □



(a) Here, we have a similar setup to Figure 1, but with an extra node v_4 and an extra, green, flow. Note that the edges between v_4 and its neighbors have a capacity of 2.



(b) The first step of pruning edges is finding the threshold. The greatest utilization of any edge when all flows follow their initial or final paths is 1, which can be found in (v_1, v_2), so the threshold is 1.



(c) Now we find all edges where the worst-case utilization is less than 1, here (v_1, v_4) and (v_4, v_2). These edges are then removed from the constraints of the LP model.

Fig. 4: Example of edge pruning. There are three flows, so $F = \{f_1, f_2, f_3\}$, which are solid red, dashed blue, and dotted green arrows, respectively. The initial paths of the flows are single-lined, while the final paths are double-lined.

Algorithm 5 Removing Smallest Flows Overapproximation

Input: A traffic system $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$ and a fraction $q \in Q_{\geq 0, \leq 1}$

Output: A smaller set of flows F and a function a that assigns demand to edges

$F' \leftarrow F$
$r \leftarrow q \cdot$ total demand of all flows in F
$f \leftarrow$ flow with the smallest traffic demand in F'
$a(e) \leftarrow 0$ for all $e \in E$
while $r - \mathit{demand}(f) \geq 0$ **do**
    $f \leftarrow$ flow with the smallest traffic demand in F'
    $r \leftarrow r - \mathit{demand}(f)$
    for $e \in \mathit{initial}(f)$ and $e \in \mathit{final}(f)$ **do** $a(e) \leftarrow a(e) + \frac{\mathit{demand}(f)}{\mathit{capacity}(e)}$
    end for
    $F' \leftarrow F' \setminus \{f\}$
end while
return F' and a

5.2 Overapproximation by Removing the Smallest Flows

When we solve our linear and integer programming models, each flow adds about an equal amount of computation time. However, not all flows are equally important. In many real-life cases, traffic demands do not all share the same size. Some flows are very large, for example if they connect two major cities, and other flows are miniscule in comparison.

If we are willing to trade precision in exchange for reducing the number of constraints in the linear programming model, we can make an overapproximation by removing some of the smallest flows, and add the traffic demand of the flows directly to the load on each edge in both their initial and final paths. This always gives at least the same maximum utilization as not removing any flows, but it can give give a greater maximum utilization.

If there is a significant difference in the size of flows, and if we only apply this technique to some of the smallest flows, then, in theory, the precision should not decrease much, as the loss in precision is proportional to the amount of traffic demand that we remove, but the computation time could improve substantially, as the gain is proportional to the number of flows removed. We show how to remove the smallest flows in Algorithm 5

By using Algorithm 5, we get a smaller set of flows F' and a function on edges a. Now, instead of (10), we use (14) for the linear programming model.

$$\text{for all } e \in E \text{ and for all } i \text{ where } 0 \leq i \leq n-2$$
$$\alpha \geq \sum_{f \in F} y^i_{f,e} + a(e) \tag{14}$$

We then use F' instead of F for all the constraints, including the new one.

If there exists a solution to $\mathcal{T}$ after applying Algorithm 5, then there also exists a solution before applying it, though the opposite does not hold.

Theorem 7. *Let $\mathcal{T} = (G = (V, E, \mathit{capacity}), F, \mathit{initial}, \mathit{final}, \mathit{demand})$ be a traffic system, LP be the linear programming model from Theorem 5 reduced from $\mathcal{T}$, and F' and the function a be the output of Algorithm 5 with $\mathcal{T}$ as input. Let LP' be the linear programming model reduced from $\mathcal{T}$ where we use* (14) *instead of* (10) *and where we use F' instead of F for* (8)*,* (11)*,* (12)*, and* (14)*. Then, LP has a solution $\alpha \leq \alpha'$ if LP' has a solution α'.*

Proof. By analysis of Algorithm 5, we see that for each flow f we remove, the function $a(e)$ assigns the utilization that the traffic demand of f would cause to every edge in the initial and the final path of f. Since the removed flows can never produce a greater utilization on the edges than the assignment of a on the edges does, we can consider a to be the worst-case utilization that the removed flows could have caused on each edge. In (14), we then add this worst-case utilization to our constraints, so α' must be greater than or equal to α. □

6 Experimental Results

We are interested in how our linear programming model performs in practice, so in this section, we perform a series of experiments to evaluate its performance.

6.1 General Experimental Setup

We implement the linear programming model from Theorem 5 for the general update sequence decision problem, to show that we can use it to solve the problem, and see how it performs in terms of efficiency and scalability. We make both a decision and an optimization variant for the linear programming model, where the decision variant simply determines if a solution exists, while the optimization variant determines the best possible maximum utilization. We implement this using the Python library PuLP [20], and our implementation can be found at "https://github.com/mlundb18/Optimizing-Link-Utilization-During-Network-Migration.git".

We use the Internet Topology Zoo dataset [16] for networks, and generate our own flows and edge capacities. For our experiments, we give each edge a capacity of 100,000. For each flow, we generate the initial and final paths by

1. Randomly selecting a start node,
2. Randomly selecting a connected end node for the initial path,
3. Randomly generating the initial path from the start node to the end node,
4. Randomly selecting a connected end node for the final path, and
5. Randomly generating the final path from the start node to the end node.

We use a gravity model[24] to generate the traffic demands. To do this, we randomly assign the square of a number between 1 and 10 to each node, which is the gravity of the node, and the traffic demand is given by multiplying the gravity of the start and end nodes of each flow's initial path. In the generated dataset, about half of the traffic systems have a maximum utilization of more than 1, while the remaining have a maximum utilization of less than 1.

Of the topologies from the Internet Topology Zoo, the largest has 245 edges and 197 nodes, while the smallest topology has 4 edges and 5 nodes. For each topology, we generate an amount of flows equal to 10 times the nodes of the respective topology.

The experiments are run on the DEIS-MCC Slurm cluster [13] at Aalborg University, with a memory limit of 200 GB RAM.

6.2 Length bound experiment

Purpose of experiment For the experiments, we wish to know what an appropriate length bound for the update sequences are, such that we have a good balance between computation time and maximum utilization. We hypothesize that, because of Lemma 1, the greater the length bound on the update sequences are, the better the maximum utilization will generally be. However, we also believe that this leads to a slower computation time, as it increases the number of constraints for the linear programming model.

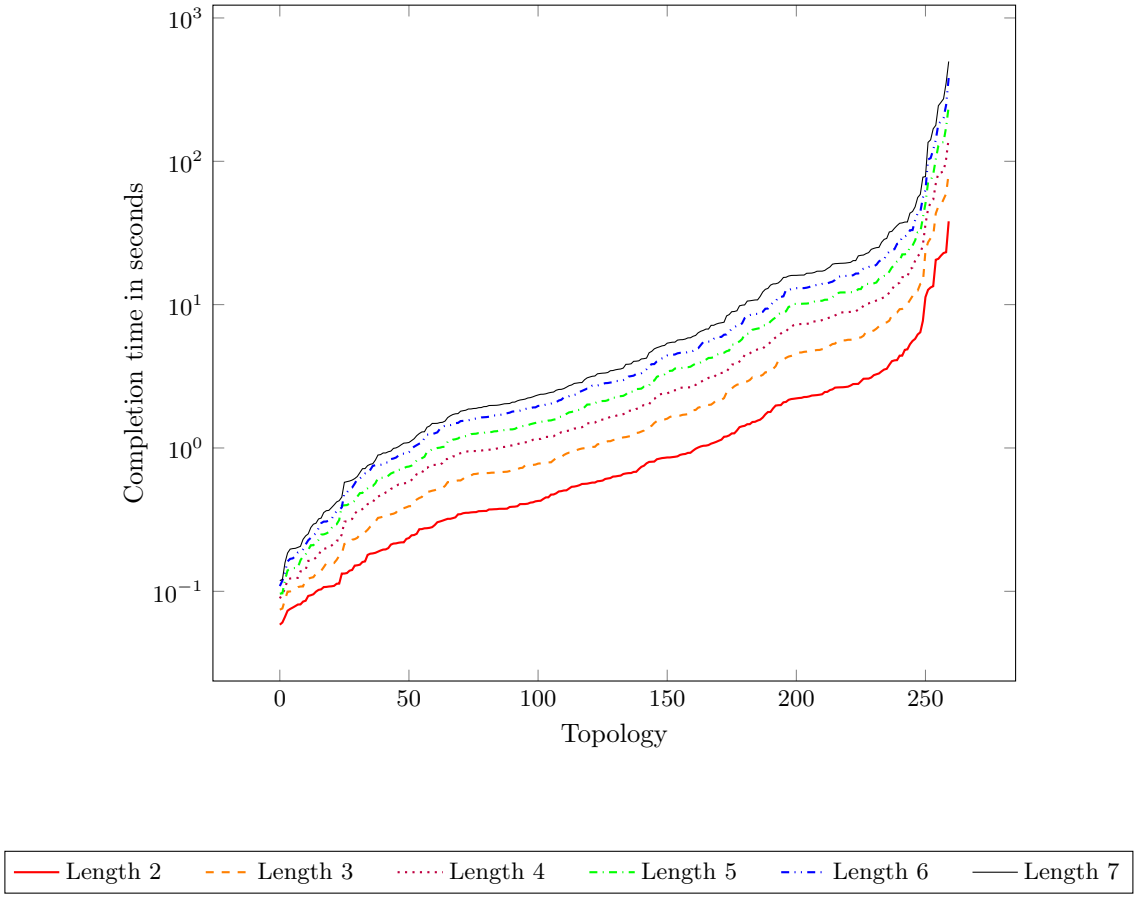


Fig. 5: Cactus plot that shows the comparison of the linear programming completion time for different length bounds. The completion time shows that the time taken overall increases as the length bound of the update sequences increases.

Setup of experiment We setup the experiment by implementing and running an unmodified version of the linear programming model from Theorem 5. We solve the model for each network in the dataset, with length bounds for the update sequence ranging from 2 to 7.

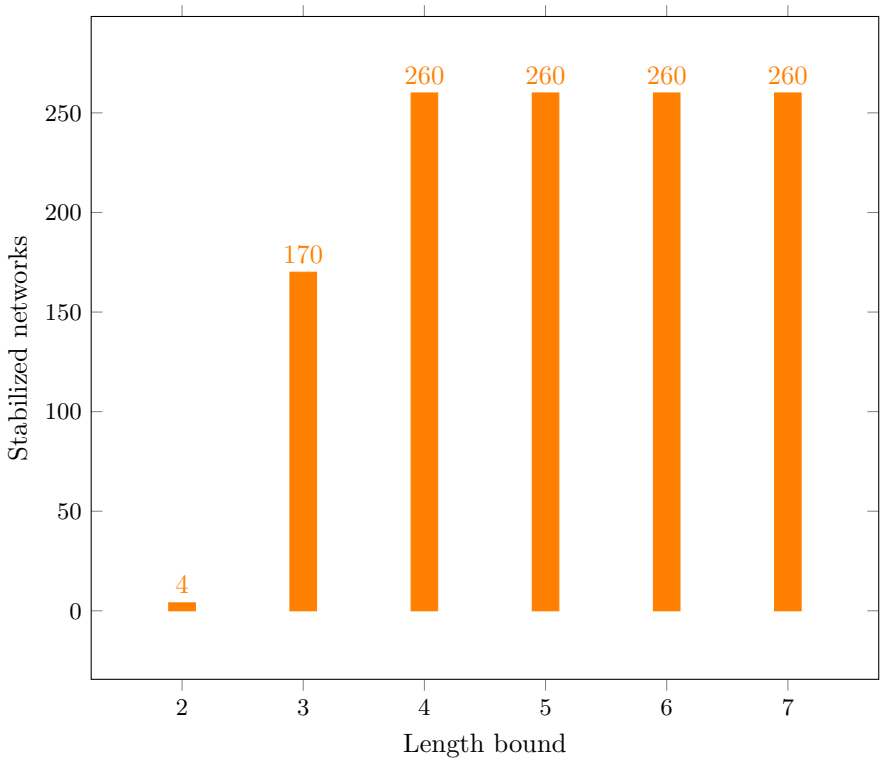


Fig. 6: Graph where the amount of stabilized networks are shown for each length bound. A stabilized network is when the maximum utilization does not change as the length bound increases, i.e. we have found the optimal solution for any length bound. The figure shows that with a length bound of 2, only 4 networks have stabilized, with a length bound of 3 170 networks have stabilized, and at a length bound of 4 and up, all networks have stabilized.

Discussion of Experiment In the results of the experiment, shown in Figure 5, we observe that for every increase in the length bound, the computation time increases. This confirms our hypothesis that a longer length bound will lead to a slower computation time. We also observe in the results of Figure 6 that the maximum utilization decreases when going from a length bound of 2 to a length bound of 3, as well as from a length bound of 3 to a length bound of 4. However the maximum utilization does not decrease after a length bound of 4, which is contrary to our hypothesis that a greater length bound leads to less maximum utilization. This could be because the dataset we generated does not have any cases like what is described in Lemma 1. With these results in mind, we run the remaining experiments with a length bound of 4.

6.3 Decision LP Experiment

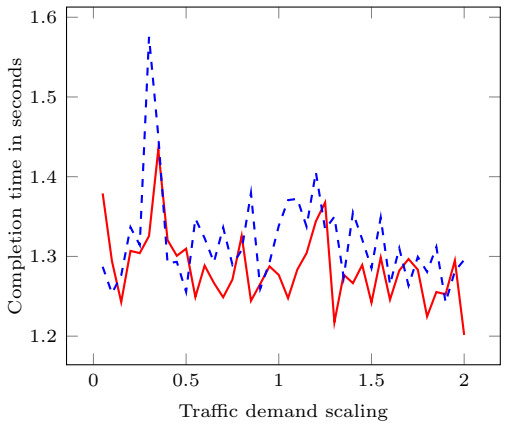
Purpose of experiment The linear programming model optimizes the solution, but we hypothesize that, in some cases, it is faster to not look for an optimal solution, but just find a feasible solution where the maximum utilization is less than 1. This could especially be the case for topologies where the demand is much smaller or much larger than what there is capacity for, since an LP solver might not have to explore many options before confirming or denying whether a solution exists.

Setup of experiment We setup this experiment by implementing both and unmodified version of the optimization linear programming model from Theorem 5, as well as a modified linear programming model which only checks if there is a feasible solution to the update sequence decision problem. We do this by removing the optimization function (1) and instead using constraint (15).

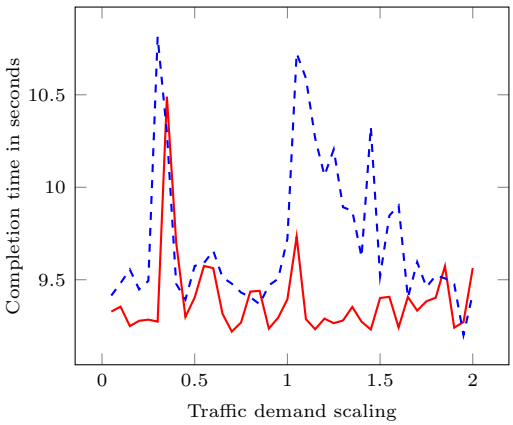
$$\alpha \leq 1 \tag{15}$$

To test the effect of demands of different sizes on the decision model, we hand pick 4 topologies and choose a link capacity such that the maximum utilization is 1. These topologies are Aarnet, Bellcanada, Darkstrand, and Zamren. We run the experiment over 40 iterations for each test, where we set the demand of each respective flow f to $\mathit{demand}(f) \cdot \mathit{iteration} \cdot 0.05$ for each topology. The results thus cover different levels of maximum utilization, ranging from 0.05 to 2.00.

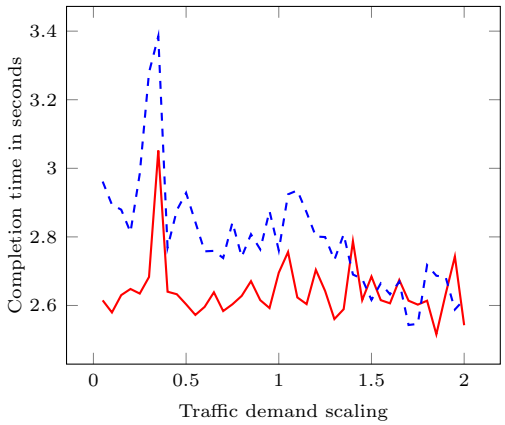
Discussion of experiment In the results of the experiments, shown in Figure 7, we observe that, though there are differences and fluctuations, the difference between the optimization and decision models is small. We do however observe that there is a general tendency for the optimization model to have a slightly faster computation time than the decision model.



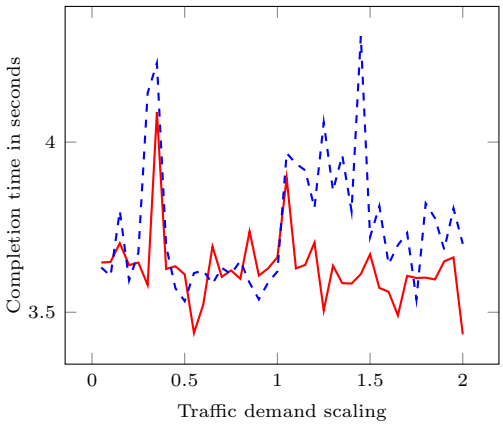
(a) Plot of the decision LP experiment for the Aarnet topology.



(b) Plot of the decision LP experiment for the Bellcanada topology.



(c) Plot of the decision LP experiment for the Darkstrand topology.



(d) Plot of the decision LP experiment for the Zamren topology.

Fig. 7: The traffic demand scaling is a multiplier on the traffic demand of each flow. The x-axis consists of 40 demand scalings from 0.05 to 2. A solution exists when the scaling is at most 1. This experiment is run on 4 different topologies, and in each case, a solution exists if the traffic demand is at most 1.

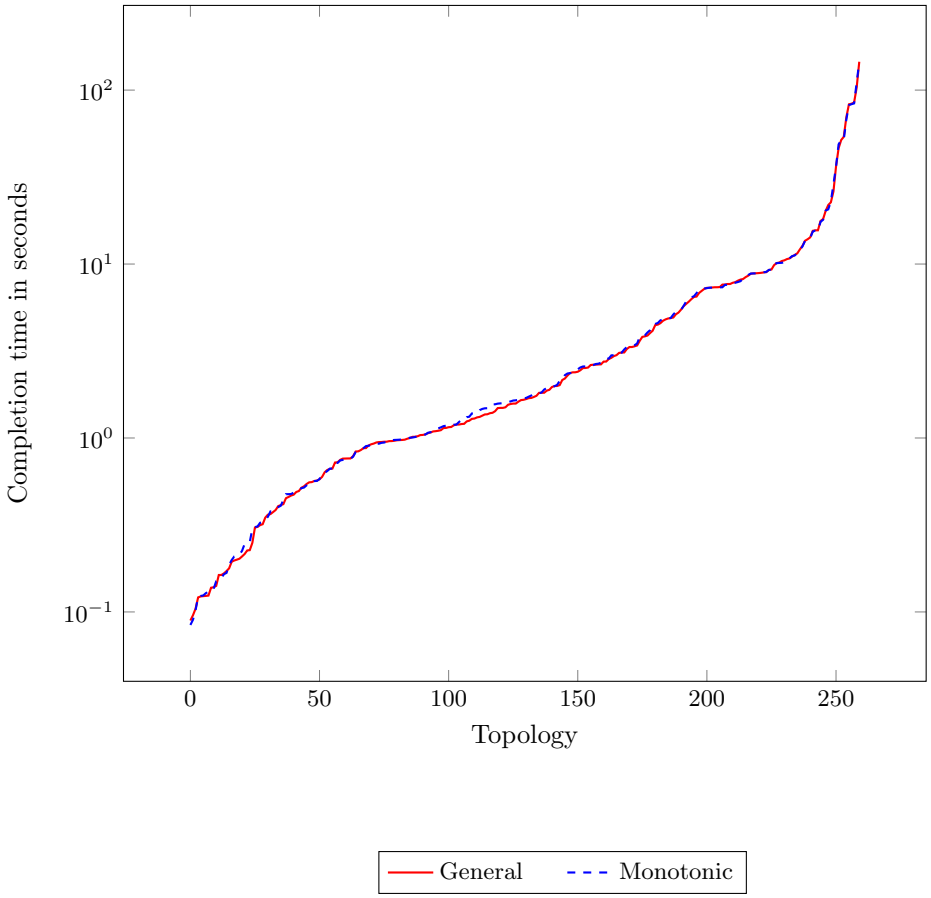


Fig. 8: Cactus plot that depicts the runtime of solving both the general problem and the monotonic problem with linear programming. In every case, the maximum utilization was exactly the same for the general and monotonic models.

6.4 Monotonicity Experiment

Purpose of experiment In Theorem 5, we find that the complexity of both the general and monotonic update sequence decision problems are the same. Since we have a linear programming model for both, we can observe how they perform against one another. The purpose of this experiment is twofold. First, is there a difference in computation time? Second, does the monotonic variant find the same solutions as the general one?

Setup of experiment We implement and test both an unmodified version of the optimization linear programming model, which we call 'General', as well as the monotonic linear programming model from Theorem 5, which we call

'Monotonic', on the dataset we generated. The difference between the two models is that the monotonic linear programming model has (13) in addition to the constraints of the general model.

Discussion of experiment We observe in Figure 8 that 'General' and 'Monotonic' both have almost the same computation time. Furthermore, the same maximum utilization was found for the general and monotonic models for all of the 260 topologies. For this reason, we propose Conjecture 1.

Conjecture 1. For a traffic system $\mathcal{T}$, there exists an update sequence of length n with a maximum utilization of u if and only if there exists a monotonic update sequence of length n with a maximum utilization of u.

6.5 Pruning Experiment

Purpose of experiment In Section 3.2 and Section 5.1, we respectively present techniques for pruning flows and edges, which can reduce the number of constraints in the linear programming model. We know that applying these techniques will not change the maximum utilization, but we hypothesize that applying these techniques will speed up the computation time as a result of fewer constraints.

Setup of experiment For comparison, we use the linear programming model from Theorem 5, which we call the baseline. Then, we also implement and test a modified version of the linear programming model, where the edge pruning and flow pruning from Algorithm 4 and Algorithm 2 are applied. Because they use the same logic, the techniques can be used in combination with almost no additional cost compared to using one of them separately, so we also test using both in combination.

Discussion of experiment In the results of the experiments, shown in Figure 9, we confirm our hypothesis that the computation time is reduced for both techniques. We particularly note that the edge pruning technique produces a more significant speed up over the flow pruning technique, and that the combination of both produce an even greater speed up.

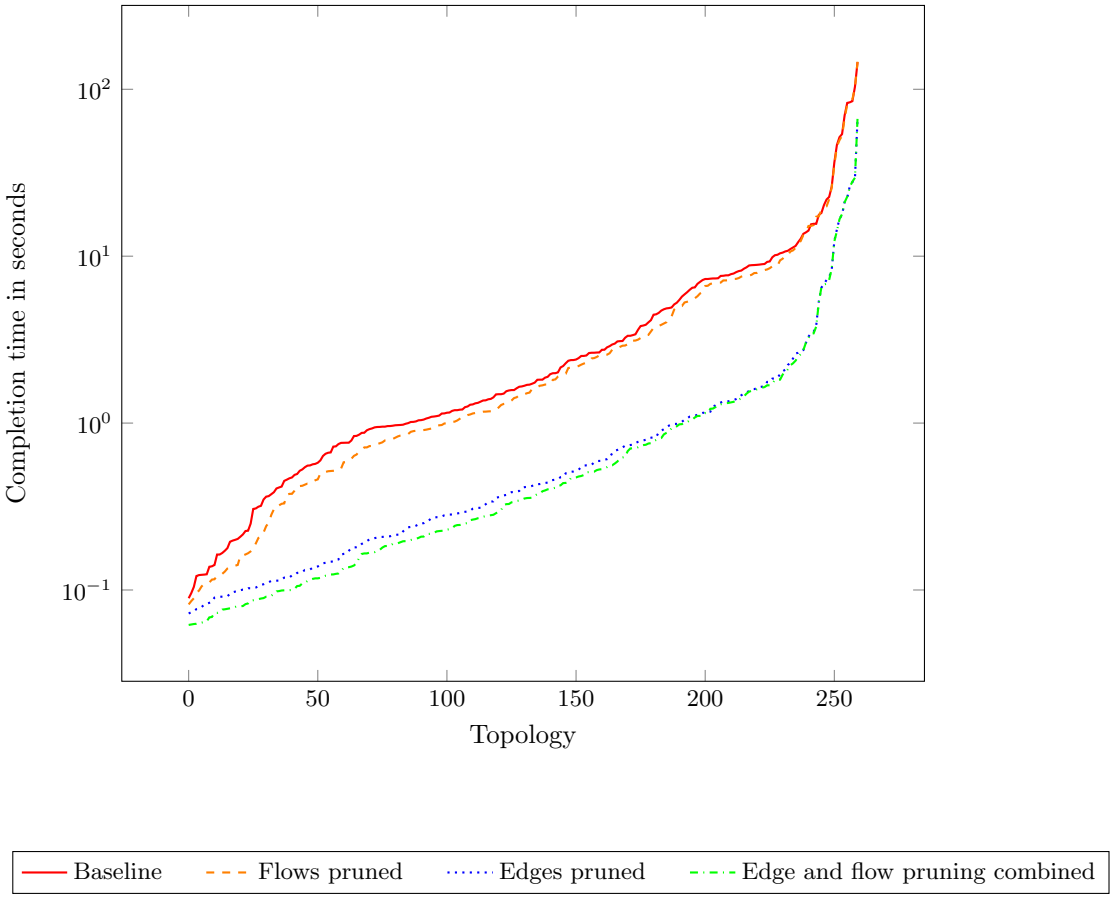
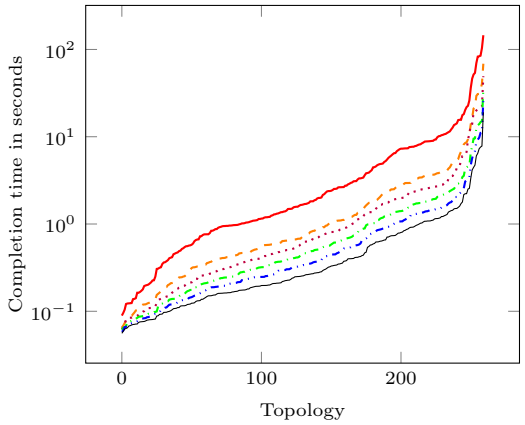


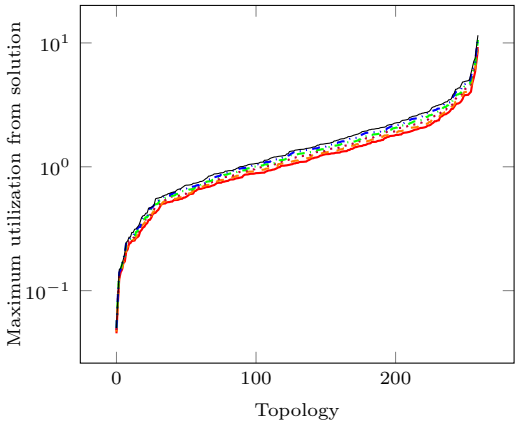
Fig. 9: Cactus plot that depicts the runtime in tests where we apply the flow pruning from Section 3 and the edge pruning from Section 5. The Baseline label indicates that no reductions have been applied. In all of the solutions, the maximum utilization was the same for each test

6.6 Flow Removal Experiments

Purpose of experiment We apply the technique from Section 5.2 to speed up computation time at the cost of precision, by removing some of the smallest flows corresponding to a fraction of the total demand. We can remove any fraction of the total demand, but our assumption is that removing more demand will lead to less computation time at the cost of a greater maximum utilization.



(a) Logarithmic cactus plot of completion time for each topology



(b) Logarithmic cactus plot of the maximum utilization for each topology

Fig. 10: Cactus plot of the technique from Section 5.2 applied with to the Internet Topology Zoo. x% removed means that the smallest flows corresponding to x% of the total demand are removed. 'Baseline' means that no flows are removed.

Setup of experiment For comparison, we use the linear programming model from Theorem 5, which we call the baseline. We then run a series of tests where we respectively remove 10%, 20%, 30%, 40%, and 50% of the total demand by using Algorithm 5.

Discussion of experiment We confirm our hypothesis by observing the results from Figure 10, where we can see that the computation time decreases as flows are removed. As expected, we observe that, in general, removing more demand increases maximum utilization. Interestingly, in Figure 11, we see that even after removing flows equal to 50% of the demand, there is some network where the maximum utilization is the same as in the Baseline. This could be due to how the dataset is generated on particularly small topologies. The average difference shows that for each 10% demand removed, the maximum utilization increases by about 5%. We also see that the maximum difference generally increases about 10% for each 10% demand removed.

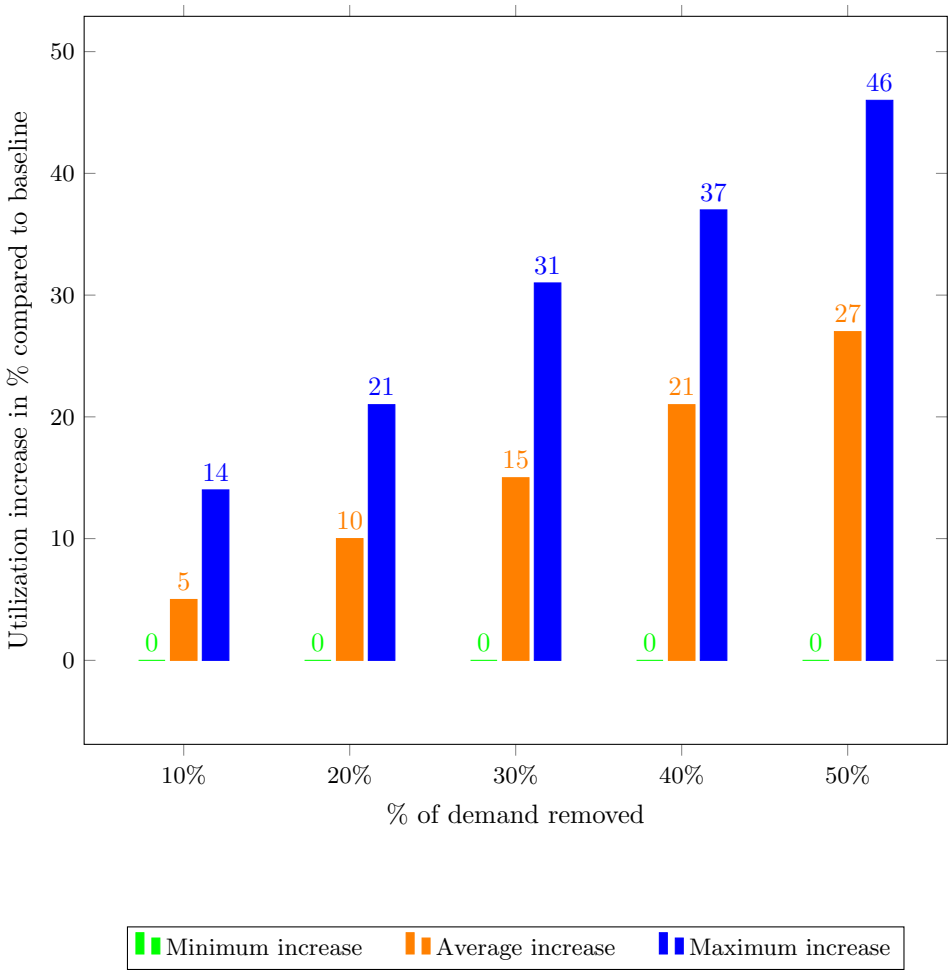


Fig. 11: Depiction of the difference in maximum utilization between the results from the baseline LP model, and the models where we have removed some of the smallest flows. For each element on the x-axis, we have three bars, respectively showing the smallest, the average, and the greatest increase in maximum utilization among all networks.

6.7 Combination of Pruning and Flow Removal Experiment

Purpose of experiment Applying the pruning techniques in Section 6.5 and the demand removal from Section 6.6 shows promising results. Our assumption is that using both at the same time can lead to an even faster computation time, while still only increasing the maximum utilization slightly.

Setup of experiment For comparison, we use the linear programming model from Theorem 5, which we call the baseline. We also implemented and ran the linear programming model where we removed flows equal to 10% of the demand, from Section 6.6, the linear programming model where we pruned both the edges and flows from Section 6.5 as well as a linear programming model that applies both of these techniques. We applied these techniques sequentially, first using the pruning techniques before removing the smallest flows corresponding to 10% of the demand.

Discussion of experiment In the results of Figure 12, we confirm our hypothesis that combining the techniques leads to an overall speedup in computation time. We also observe among that of the two techniques, pruning and removing flows, the pruning techniques lead to an overall faster computation time, though that could change if more than 10% of the demand is removed. When all techniques are applied, we see that, for most topologies, computation time is improved by about a factor of 10.

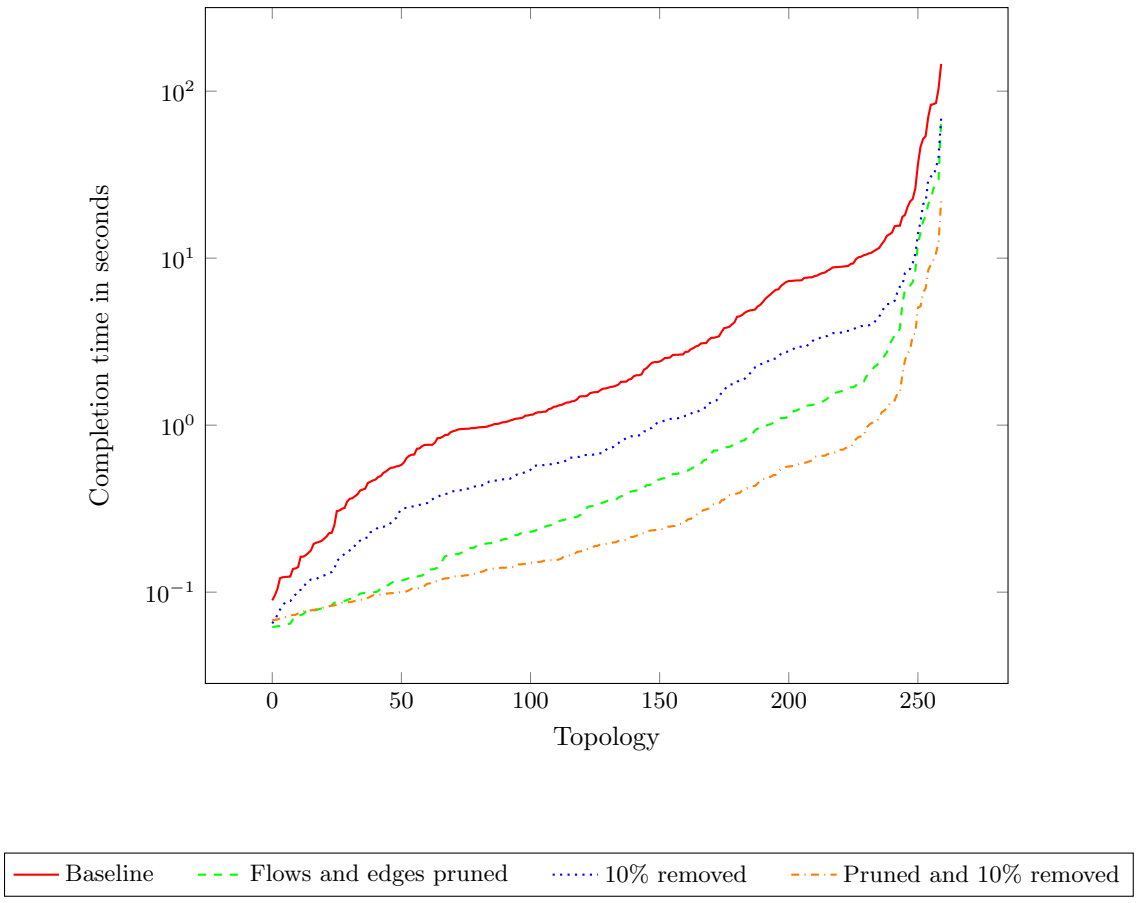


Fig. 12: Cactus plot that depicts the runtime in tests where we apply the pruning techniques from section 3.2 and section 5.1, as well as the flow removal technique from section 5.2. The 'Baseline' label indicates that no reductions have been applied, and 10% removed means that the smallest flows corresponding to 10% of the total demand has been removed

7 Related work

The subject of avoiding congestion during network updates has already been studied. Some notable work in this area is Dionysus [14], Atomip [19], SWAN [12], Brandt et al. [4], and ez-Segway [22], that each have their own techniques to solve the problem. The main difference that sets our work apart is that we are capable of finding an optimal solution, while they look for any solution that provides a maximum utilization of at most 1.

Dionysus [14] is a system for updating networks in a congestion-free manner. In their model, flows are unsplittable, so each flow sends its data along either its initial configuration or its final configuration, but not a mix of both. They use dependency graphs that represent potential updates and available network resources, which can then be used to determine the possible updates. They find that using these graphs to find update schedules that do not create congestion is NP-complete. One of the main advantages of Dionysus is that it is able to adapt its scheduling at runtime depending on which switches have been updated, and while our problem definition is more general than theirs, as we allow flows to be split between their initial and final configurations, the update schedule that we produce cannot adapt.

Atomip [19] uses mixed integer linear programming to provide a solution for updating SDN networks with an ordering that minimizes transient congestion. The solution uses a heuristic method to solve the integer programming problem, and experimental results show that it is able to solve the problems in sub-seconds. One of the techniques Atomip uses to quickly create update schedules is to remove what it calls non-critical flows and edges, and this is the main inspiration for the flow and edge pruning in Section 3.2 and Section 5.1. What they do is to remove all flows and edges that they know definitely cannot cause congestion, meaning a utilization of at least 1, and they can check this by determining if a link has more capacity than what all flows can bring through it. The main difference with our techniques is that we do not check if edges and flows can cause congestion, but instead if they can affect the maximum utilization, which means that our technique can be useful even if all edges and flows are critical by their measure, and even if the goal is to find the optimal maximum utilization.

SWAN [12] is an influential system for providing a congestion-free update schedule. When they update a flow, they find the measure of slack on each affected link, which is how much capacity there is left for the flows, and they use that to determine how many rounds of updates it takes to update the flows. This can be powerful, because in many cases, it allows them to bound the length of an update sequence to the necessary length to find a solution. After finding the length, they use linear programming to determine if such a solution exists, and if there is slack, a solution can always be found. If there is no slack, then it may still be that a solution exists, but if it does, then this method cannot be used to find out how many update rounds are needed. We do not use any mechanism similar to slack, though it could easily be used in conjunction with our linear programming model. However, in Section 6, we see that using more than 4 update rounds does not affect the maximum utilization for any networks

in our dataset, so using slack to determine the length of the update sequences would not improve results for our dataset.

Brandt et al. [4] builds upon the work of SWAN and Dionysus introduce the first polynomial-time algorithm that can decide whether there exists an update sequence that updates one flow configuration to another while avoiding congestion in the network. Their definition of flows allows for multiple paths for both the initial and final configuration of each flow, while we only allow one initial path and one final path, which means that they are more general than us in this regard. They show a polynomial-time algorithm for their problem, but they have not implemented it, so it is unknown how well it performs in practice.

ez-Segway [22] presents a decentralized mechanism for fast and consistent network updates. The main idea is that when a centralized controller makes changes to the network flows, each switch is given a set of conditions for when it can update. The switches communicate their state to each other, so the switches can update without additional input from the controller. This method can result in deadlocks, where groups of routers wait for each other to update. Ez-Segway aims to avoid these situations by using flow segmentation, where different segments of flows can be updated independently, and splitting volume, which means that a flow can split the traffic between its initial and final paths. Our notion of split ratios in Section 2 is inspired by splitting volume.

8 Conclusion

We studied how to update flows from their initial configuration to their final configuration while avoiding link congestion. We showed that, by allowing flows to be split between their configurations, and by bounding the length of the update sequence, we could decide whether there exists a solution in polynomial time by reduction to a linear programming model. We also found that, if we fixed a granularity for the update sequence or update flows one at a time, i.e. atomic updates, then the problem was NP-hard. To simplify the problem, we showed how to prune flows and edges that would not affect the maximum utilization in any way, and we found a technique to remove some of the smallest flows without giving incorrect solutions, though those solutions could be suboptimal.

Where most tools in the domain of congestion-free network updates are limited to finding any update sequence that limits the maximum utilization of a network's links to less than 100%, our linear programming model found an optimal update sequence with the least possible maximum utilization. We implemented our model to conduct a series of experiments, where we found that an update sequence with a length of 4 was sufficient to find the best possible maximum utilization for our dataset, though we know that there are networks where more update rounds always provide a better maximum utilization. We also saw that, for our system, finding an optimal solution was generally slightly faster than determining if a solution existed. Finally, we discovered that applying all of our techniques to simplify the problem, in many cases, improved computation time by more than a factor of 10.

9 Future Work

There are multiple avenues of further study that could be followed based on our findings. Here, we present some of the theoretical and experimental possibilities for further research.

9.1 Multiple Paths for Initial and Final Flow Configurations

The flow model in [4] allows for both their initial and their final configuration to have multiple paths. We only allow for one path in either configuration, but there is little reason why our work could not be extended in this direction. This is an interesting direction to explore, because it could make our model more general, especially since many routing strategies, like ECMP, use multiple paths.

9.2 Techniques to Determine Update Sequence Length

The concept of slack from [12] could be adapted to our problem, which could, in some cases, help us to determine a length for the update sequences. While we found that a length of 4 was sufficient, we believe that this is dependent on the dataset, so using this technique could help ensure that our solution is useful in more situations. However, we note that the slack technique is only useful for decision problems, so it is insufficient if the goal is to find an optimal solution. We propose exploring other options for determining a length bound for update sequences that could be used when optimizing.

9.3 Unsolved Computational Complexity Questions

In Table 1, we see that we have not found a lower bound for general and monotonic update sequence problems. This is interesting, because if the problems belong to a lower complexity class than P, then they may be possible to parallelize. We believe the problem to be P-complete, but as we have not been able to prove so.

In addition, the fixed granularity update sequence decision problem is NP-hard and in PSPACE, but we have not discovered the exact complexity of the problem.

9.4 Monotonic Conjecture Proof

In Section 6.4, we proposed Conjecture 1. We believe the conjecture to be true, but we have yet to prove it, so we propose it as an obvious approach for future work.

9.5 Additional Experiments

Increased Amount of Flows In Section 6, we used a generated dataset for flows and capacities, where the amount of flows per topology was scaled to 10 times the amount of nodes for each topology. However, our approach could be improved to be more realistic, for example by using a number of flows corresponding to the square of the number of nodes.

Real world dataset While generating a more realistic dataset would improve the validity of our results, it would be even better if we could test our linear programming model on a real-world dataset.

Measuring Computation Time for Pruning Flows and Edges and Removing Flows In Section 6.5 and Section 6.6 we applied techniques to reduce the size of the traffic systems. While they can drastically speed up the computation time for solving the linear programming model, we have not measured the computation time of the techniques themselves, and without that, we do not know for sure how much they improve the total computation time.

Measuring Computation Time for Linear Programming Constraint Instantiation In Section 6, we tested the computation speed of solving the linear programming model, but we did not measure the time it took to instantiate all the constraints. This means that the total time of running our solution might be longer than what is indicated by our experimental results.

Bibliography

[1] Partition problem (Nov 2022), URL https://en.wikipedia.org/wiki/Partition_problem

[2] Amiri, S.A., Ludwig, A., Marcinkowski, J., Schmid, S.: Transiently consistent sdn updates: Being greedy is hard. In: Structural Information and Communication Complexity: 23rd International Colloquium, SIROCCO 2016, Helsinki, Finland, July 19-21, 2016, Revised Selected Papers 23. pp. 391–406. Springer (2016)

[3] Brandt, S., Foerster, K.T., Wattenhofer, R.: Augmenting flows for the consistent migration of multi-commodity single-destination flows in sdns. Pervasive and Mobile Computing 36, 134–150 (2017)

[4] Brandt, S., Förster, K.T., Wattenhofer, R.: On consistent migration of flows in sdns. In: IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications. pp. 1–9. IEEE (2016)

[5] Cao, K., Liu, Y., Meng, G., Sun, Q.: An overview on edge computing research. IEEE Access PP, 1–1 (01 2020)

[6] Feamster, N., Balakrishnan, H., Rexford, J., Shaikh, A., Van Der Merwe, J.: The case for separating routing from routers. In: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture. pp. 5–12 (2004)
[7] Foerster, K.T.: On the consistent migration of splittable flows: Latency-awareness and complexities. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). pp. 1–4. IEEE (2018)
[8] Foerster, K.T., Ludwig, A., Marcinkowski, J., Schmid, S.: Loop-free route updates for software-defined networks. Ieee/acm Transactions on Networking 26(1), 328–341 (2017)
[9] Förster, K.T.: Don't disturb my flows: Algorithms for consistent network updates in software defined networks. Ph.D. thesis, ETH Zurich (2016)
[10] Förster, K.T., Mahajan, R., Wattenhofer, R.: Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In: 2016 IFIP Networking Conference (IFIP Networking) and Workshops. pp. 1–9. IEEE (2016)
[11] Hong, C.Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., Wattenhofer, R.: Achieving high utilization with software-driven wan. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM. pp. 15–26 (2013)
[12] Hong, C.Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., Wattenhofer, R.: Achieving high utilization with software-driven wan. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM. pp. 15–26 (2013)
[13] Jensen, P.G.: Deis-mcc (2023), URL https://github.com/DEIS-Tools/DEIS-MCC
[14] Jin, X., Liu, H.H., Gandhi, R., Kandula, S., Mahajan, R., Zhang, M., Rexford, J., Wattenhofer, R.: Dynamic scheduling of network updates. SIGCOMM Comput. Commun. Rev. 44(4), 539–550 (aug 2014), URL https://doi.org/10.1145/2740070.2626307
[15] Karp, R.M.: Reducibility among combinatorial problems. Springer (2010)
[16] Knight, S., Nguyen, H., Falkner, N., Bowden, R., Roughan, M.: The internet topology zoo. Selected Areas in Communications, IEEE Journal on 29(9), 1765 –1775 (october 2011)
[17] Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. Proceedings of the IEEE 103(1), 14–76 (2014)
[18] Ludwig, A., Rost, M., Foucard, D., Schmid, S.: Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In: Proceedings of the 13th ACM Workshop on Hot Topics in Networks. pp. 1–7 (2014)
[19] Luo, L., Yu, H., Luo, S., Zhang, M.: Fast lossless traffic migration for sdn updates. In: 2015 IEEE International Conference on Communications (ICC). pp. 5803–5808 (2015)
[20] Mitchell, S., Kean, A., Mason, A., O'Sullivan, M., Phillips, A., Peschiera, F.: Pulp (2009), URL https://coin-or.github.io/pulp/

[21] Miyano, S., Shiraishi, S., Shoudai, T.: A list of p-complete problems (1989)
[22] Nguyen, T.D., Chiesa, M., Canini, M.: Decentralized consistent updates in sdn. In: Proceedings of the Symposium on SDN Research. p. 21–33. SOSR '17, Association for Computing Machinery, New York, NY, USA (2017), URL https://doi.org/10.1145/3050220.3050224
[23] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstractions for network update. ACM SIGCOMM Computer Communication Review 42(4), 323–334 (2012)
[24] Roughan, M.: Simplifying the synthesis of internet traffic matrices. ACM SIGCOMM Computer Communication Review 35(5), 93–96 (2005)
[25] Zheng, J., Chen, G., Schmid, S., Dai, H., Wu, J., Ni, Q.: Scheduling congestion-and loop-free network update in timed sdns. IEEE Journal on Selected Areas in Communications 35(11), 2542–2552 (2017)
[26] Zheng, J., Li, B., Tian, C., Foerster, K.T., Schmid, S., Chen, G., Wux, J.: Scheduling congestion-free updates of multiple flows with chronicle in timed sdns. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). pp. 12–21. IEEE (2018)