Bachelorarbeit

Update Your Network - Loop Free

Verfasserin oder Verfasser

Tabea Reichmann

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2021

Studienkennzahl lt. Studienblatt:	UA 521
Fachrichtung:	Informatik - Scientific Computing
Betreuerin / Betreuer:	Dr. Kathrin Hanauer, MSc, BSc Dr. Dr. Dipl.-Math. Dipl.-Inform. Klaus-Tycho Förster

Contents

Abstract

In Software Defined Networking, updating forwarding rules should be done as efficiently as possible - many rules should be updated in a short amount of time. The resulting rules must not have a cycle, as packets should be able to reach their destination and not get dropped. Due to these limitations, a heuristic approach is needed, as updating loop-free optimally, using the fewest rounds, is an NP-complete problem. As the most updates should be made in each round, and the number of rounds should be reduced, therefore only few rules should not be added, this is an application of the Feedback Arc Set Problem. This thesis compares algorithms and combinations of those algorithms based on heuristics that approximate solutions to the Feedback Arc Set Problem to find out how current techniques can be improved.

1 Motivation

With the rise of *Software Defined Networking*, flexibility in routing can be achieved. This is valuable, as, due to maintenance or link and node failures, new rules sometimes need to be computed. As a Software Defined Network is a distributed system, there are other challenges regarding the updating procedure. Delays in the communication between switches and the controller can make it inconsistent when updating all nodes at once. Solutions to this problem have been studied before, an example are the usage of version tags, see [23], or only updating a certain subset of nodes, so loop-freedom can be maintained, see [7, 9, 20]. The latter approach is also studied in this thesis.
While a node is not updated, its outgoing old arc cannot be removed, so packets can still be routed to the destination, without them being dropped. As loop-freedom needs to be preserved, there are limits to which rules can be added per round. The goal is to minimize the number of rounds needed for the updates, while maximizing the updates that are possible per round. That an optimal 3-round update schedule is NP-hard has been shown in [7].
The state-of-the-art algorithm is a greedy approach, in which the computed new rules are tried to be added to the old rules. The problem is an application of the *Feedback Arc Set Problem*, which is another NP-complete problem in general, except for a few special graphs. Therefore, there exist a number of approximation algorithms that can be adapted to fit the context of network updates, e.g. the *Eades-Lin-Smyth* algorithm [5], or the *1-Opt* algorithm [13].

This thesis adapts the aforementioned Eades-Lin-Smyth and 1-Opt algorithm and compares these with the state-of-the-art greedy approach, as well as adapting the greedy approach, which uses a random insertion order, by sorting the arcs by the indegree of their tails. Further, the 1-Opt algorithm and the greedy algorithm are used as post processing techniques of the Eades-Lin-Smyth algorithm. The aim is to improve the current techniques and to find out advantages and disadvantages of the different algorithms.

2 Preliminaries

2.1 Definitions and Terminology

In this section, the terminology further used in the thesis is covered. When using standard terminology, the definitions from [10, 16, 21, 25] are used.
First to some definitions in the field of graph theory, that are important over the course of the thesis:
A *graph* is defined as $G = (V, A)$, with V denoting the vertices (or nodes) and A the arcs (or edges) of the graph. A *digraph* $D = (V, A)$ is a graph with directed arcs. In a directed graph, an arc is written as $a = (v_1, v_2)$, where the node v_1 is called the *tail* and v_2 the *head* of the arc a. The arc a is an *incoming arc* into v_2 and an *outgoing arc* from v_1.

In a directed graph $D = (V, A)$, the indegree of v is the number of incoming arcs into a vertex v, and is denoted by $d^-(v)$. The outdegree is the number of outgoing arcs from a vertex v and is written as $d^+(v)$.

Planar graphs are graphs that can be drawn in a plane without its edges crossing.

A *linear ordering* of the nodes V in a graph is linear sequence, ranking the nodes. It also can be seen as a permutation of the vertices. If the vertex v_1 is ranked before v_2, we write $v_1 \prec v_2$. The linear ordering is denoted as σ, and $\sigma(i)$ defines the position of the node i in the linear ordering.

A forward arc is an arc, with its tail having a lower position in the linear ordering than its head. A back arc is an arc, with its head having a lower position in the linear ordering than its tail.

Further, also the *Feedback Arc Set Problem* (further also written as *FASP*) has to be defined. There exists a definition as an optimization problem, and a definition for the decision problem. First to the definition of the optimization problem:
The Feedback Arc Set Problem is defined for a directed graph $D = (V, A)$, where the aim is to find an arc set B of minimum cardinality, so that the resulting graph $\hat{D} = (V, A \backslash B)$ is acyclic. It can also be defined for a weighted graph, in which the subset B is of minimum weight. [21]
The Feedback Arc Set Problem can also be defined as a decision problem: For an for a directed graph $D = (V, A)$ and a positive integer $K \leq |A|$ is there a subset $B \subseteq A$ with the property $|B| \leq K$ such that B contains at least one edge from every directed cycle of D?

Next, *Software Defined Networking (SDN)* has to be defined. It is an alternate approach to implementing routing functionality in networks to the traditional approach, where each routing component runs a routing algorithm and therefore forwarding and routing is done within the router. SDN physically separates

routers and the controlling functionality which computes and distributes the forwarding tables to each router in the network.

When now an update has to be made, e.g. due to maintenance work, or link / node failures, a new set of rules gets computed by the routing algorithm. During the updates, the network has to stay consistent. Therefore, there should not be any loops when adding the new rules to the set of old rules, as a simultaneous update cannot be guaranteed and otherwise there can be inconsistencies. While *Strong Loop-Freedom* means that forwarding rules stored in the switches must always be loop-free, *Relaxed Loop-Freedom* means that only the rules in switches along the path from the source to the destination have to be loop-free. [7]
This thesis focuses on strong loop-freedom, therefore this definition is used when talking of loop-freedom.

2.2 Equivalent and Related Problems to the FASP

There exist several equivalent or related problems to the Feedback Arc Set Problem. One equivalent problem is the *Acyclic Subdigraph Problem (ASP)*. The optimization problem is defined as follows:
For a digraph $D = (V, A)$ find a subset $B \subseteq A$, which contains no cycles and has maximum cardinality, or, in case of a weighted digraph, maximum weight. The optimal solution is trivially equivalent to the Feedback Arc Set Problem.
A related problem to the Feedback Arc Set Problem is the *Feedback Node Set Problem* or *Feedback Vertex Set Problem (FNSP)*. Again, the definition of the optimization problem is to find a subset $W \subseteq V$ with minimum cardinality or weight of the nodes or arcs, such that $(V \backslash W, A(V \backslash W))$, with $A(V \backslash W)$ denoting all arcs that do not include vertices from W but only from V, does not contain a cycle.
As a decision problem, the Feedback Vertex Set Problem can defined as follows: For a digraph $D = (V, A)$ and an integer $K \leq |V|$, is there a subset $W \subseteq V$ with the property $|W| \leq K$, such that W contains at least one vertex from each directed cycle in D?

Both the FASP and FNSP are NP-complete problems and can be reduced from the vertex cover [14]. The FASP also remains NP-complete for graphs with nodes with a total in- and outdegree of 3 [11].
While the FNSP remains NP-complete also for planar graphs, the FASP is then optimally solvable in polynomial time [17]. Further, the FASP is also solvable in polynomial time for reducible flow graphs, as their arcs can be uniquely split into forward and back arcs [22].

2.3 Topology of the Networks

The network topologies, which were constructed to evaluate the algorithms, share a specific characteristic: All nodes have exactly one outgoing arc, except for the *destination node*, which only has incoming arcs. The initial rules will

further be called *old rules* and the resulting digraph denoted as $G_1 = (V, A_1)$.
A network update may now happen due to e.g. maintenance work, and link / node failures, why a new topology is constructed, having the same properties. The rules, to which the network should now be updated will be called *new rules* and the resulting digraph denoted as $G_2 = (V, A_2)$. Both resulting graphs are trees.
The goal is now to as efficiently and fast as possible update the network, but, as loop-freedom has to be preserved, only a certain subset of $A_2^* \subseteq A_2$ can be added to the set A_1, so that the graph $G^* = (V, A_1 \cup A_2^*)$ remains acyclic and therefore inconsistencies can be avoided.
In the next step, the nodes v_i can be updated, of which the outgoing arcs are $(v_i, j_i) \in A_2^*$. The old outgoing arcs can then be deleted.
This process is iterated until all nodes are updated, therefore the resulting graph G' is equal to the graph $G' = G_2 = (V, A_2)$.
In order for this problem to be as efficiently done as possible, the fewest possible arcs should be in the set $A_2 \backslash A_2^*$, therefore being an application of the Feedback Arc Set Problem. Further, the number of iterations, called rounds, should be reduced.
To compute the feedback arc set, the graph $G = (V, A_1 \cup A_2)$ is constructed, that therefore is constructed of two trees put on top of each other.

Even for networks with the old rules and new rules just being simple paths from one source to one destination node, the problem of minimizing the number of nodes, that are not updated in the first round, has been proven to be NP-hard, see [2], where the authors use a reduction from the *Minimum Hitting Set Problem*, and [8], for networks with a single or multiple destinations. For this proof a reduction from the Feedback Arc Set Problem is used.
In [7] it has been shown that this problem is NP-complete for 3-round updates for two simple paths as old and new rules, by reducing the problem to selecting edge subsets and modifying the 3-SAT problem to an edge selection problem. The authors first classify the nodes into categories, when the nodes can be updated. While for some of the nodes it is clear when to update them, for one type the updates can be done in the first or the third round, and finding this distribution is NP-hard.

3 Related Work

The following sections outlines several algorithms used for approximating a minimum feedback arc set, as well as discusses work on network updates in Software Defined Networking.

3.1 Feedback Arc Set Problem

There are several algorithms that approximate a minimum feedback arc set, and one of the most popular ones is the Eades-Lin-Smyth algorithm [5]. The algorithm was published in 1993, and its goal is to find a linear ordering that has a small set of back arcs. To achieve this, the algorithm identifies sinks and sources of the graphs, then removes them from the graph, then identifying resulting sinks and sources. When no more sinks and sources can be found, the algorithm finds nodes with the highest $\delta = d^+ - d^-$, treating them as a sources. This procedure then moves the found sources to the front of the linear ordering, the sinks to the back. The paper states a running time of $O(m)$, with m denoting the number of arcs.

Another algorithm that computes a linear ordering of the nodes, with its back arcs then being the feedback arc set, is the 1-Opt algorithm, that, starting out from a linear ordering σ, goes through each node and tries to optimize the position of each node, setting the node to the position with the smallest number of back arcs $b = b^- + b^+$.

Furthermore, many publications describe the problem in a tournament context, see [1], [4] and [12]. A *tournament* T in A is defined as a subset of arcs that for each pair of nodes i, j in V contains either the arc (i, j) or (j, i).
One of the most prominent algorithms is the *KwikSort* algorithm [1], that adapts the idea of the *QuickSort* algorithm for sorting numbers. The KwikSort algorithm uses a divide and conquer strategy, starting out with a random vertex $i \in V$. The algorithm goes through each vertex j of the other vertices, and, if (j, i) is in the set of arcs, the vertex j is added to the left partition of V, denoted as V_L. If $(i, j) \in A$, j is added to the right partition of V_R. The linear ordering then is returned as $\sigma = (KwikSort(V_L), i, KwikSort(V_R))$, recursively calling the routine on each of the partitions.
While in [4] kernelization is used in the algorithm, [12] gives, inter alia, algorithms based on randomization for bipartite tournaments, using the properties of this graph structure.
Kudelić et al. also make use of randomization in their publication [15], developing an ant colony inspired Monte Carlo algorithm.

Another algorithm that is often used for the approximation of the minimum feedback arc set, is the Berger-Shor algorithm [3], which was developed to find the maximum acyclic subgraph. Comparing the in- and outdegree of each vertex, either the incoming arcs or the outgoing arcs are added to the subgraph.

There are two versions of this algorithm, a randomized one, and one, in which the order of vertices is chosen to get an as high as possible expected value of the subgraph size.
There is also a special case of the feedback arc set, namely the *Subset Feedback Edge Set*, for which only certain cycles are needed to be intersected [6].

3.2 Software Defined Networking and Network Updates

With the rise of Software Defined Networking the flexibility of routing has increased. However, as Software Defined Networks can be seen as distributed systems, the updating process can still be a challenge, as delays between the controller and the switches make the communication to take longer and inconsistencies can arise. Several approaches for updating can be found in the literature, e.g. the usage of version tags, or updating only a set of nodes, for which loop-freedom can be preserved.
The model, using version tags is described in [23] by Reitblatt et al. In the first step all new rules are communicated to the nodes, making it possible that only some packets, that have a certain tag saying it belongs to the new route, are forwarded along the new route. Other packets are forwarded along the old rules. When all nodes communicate that the updating process has been done, all packets get tagged as well, forcing the packets to get forwarded according to the new rules. One disadvantage is, that all nodes have to be updated before the new rules can properly be used.
Another approach is only updating a certain subset of nodes, for which it is known, that loop-freedom can be maintained. In [20] Mahajan and Wattenhofer describe the nodes that can be updated using the structure of a dependency forest, where the destination is the root node. Children must wait with their update until their parents switched from the old rules to the new rules, therefore tagging is not needed. The authors also discuss an algorithm, that greedily tries to update as many nodes as possible.
This model is also discussed in [9], where the algorithm is outlined.
In [7], also relaxed loop-freedom is studied, where only the rules in switches along the path from the source to the destination have to be loop-free. This approach is used, as loops cannot provide inconsistencies when they are not between the source and destination. This has advantages over strong loop-freedom, as a schedule for $O(logn)$-round always exists [18], making it computationally tractable.
Further, the authors conclude that the greedy approach in some cases can have a bad performance when it comes to the number of rounds needed for the updating process. While there are solutions with $O(1)$ rounds, the greedy approach can take $\Omega(|V|)$ rounds.
There is also literature on route updates, that are not destination based, see [19], but describe an arbitrary route. In the paper it is also studied, how *Waypoint Enforcement* can be maintained, in which the packet has to traverse a checkpoint.

4 Algorithms

In this section, a closer look is taken at heuristics, that provide good results for the general Feedback Arc Set Problem, and are adapted to the problem of network updates. When adapting the algorithms, it has to be taken into account that there is a set of fixed rules, that cannot be part of the feedback arc set. Starting out from a set of old rules, and a set of new rules, rules from the set of old rules are deleted, while rules from the set of new rules are added in each round. These are then deleted from the set of new rules, while the arcs, outgoing from an already updated node, are then removed from the old and fixed rules. The set of fixed rules in each round is further also called *current rules*. The feedback arcs can only be part of the the set of new rules, but never the set of current rules.
The data structure of the graph used in the implementation is in an adjacency list representation[1], therefore look ups and deletions can be done in constant time.
This chapter further examines the algorithms used in the thesis. The time complexity analysis uses the properties of the adjacency list representation.

4.1 Variation of the Eades-Lin-Smyth-Algorithm

For this thesis, a variation of the Eades-Lin-Smyth-Algorithm was used. It takes the ideas of the original algorithm and makes adaptations to fit the topic of network updates, as there is a fixed set of rules that cannot be deleted, therefore cannot be part of the resulting feedback arc set.
The original algorithm first identifies sinks in the graph, then removes the found ones from the graph and places them in the back of the linear ordering. This can result in creating new sinks, which then will be removed and placed in the linear ordering as well.
The next step is to find sources, place them in the front of the linear ordering, remove them from the graph, which again can result in creating new sources.
When no more sinks and sources can be found, the algorithm goes through all nodes that are still in the graph and chooses the node with the highest $\delta = d^+ - d^-$, place them in the linear ordering after the sources and remove the node from the graph. Then, the algorithm begins with finding sinks and sources again, iterating until the graph is empty. The following pseudo code outlines the steps of the algorithm:

```
procedure ELS(digraph D = (V, A)) [5]
    s1, s2 = list(), list()
    while D ≠ ∅ do
        while sink in D do
            prepend sink to s2
            remove sink from D
```

[1] `https://networkx.org/documentation/stable/reference/introduction.html`

```
        end while
        while source in D do
            append source to s1
            remove source from D
        end while
        find node with maximum δ = d+ − d−
        append node to s1
        remove node from D
    end while
    append s2 to s1
end procedure
```

The ELS algorithm can make use of a special data structure to be able to find sinks, sources and nodes with the highest $\delta = d^+ - d^-$ efficiently. It uses a structure of bins to save the nodes according to their δ (from $-(n-3)$ to $n-3$, with n denoting the number of nodes), and in the case of an in- or outdegree of 0, a special bin for sinks or sources. The δ of the bin with the highest available δ gets saved. If this bin now is empty, this value is decremented, until a non-empty bin is found. If an arc is deleted, the value can be maximally incremented by 1. Therefore, the value can only be incremented m (denoting the number of arcs) times and decremented $(2(n-3)+m)$ times over the whole algorithm, making the look ups for the nodes possible in $O(m)$, and therefore the algorithm possible in linear time.
To adapt the algorithm to fit the problem of network updates, an additional step has to be added - a look up for nodes without incoming fixed arcs.
While the original algorithm can add any arc to the feedback arc set, the adaptation cannot. Therefore, when arranging the linear ordering, nodes without incoming fixed arcs can be appended to the list of sources, as no cycle can then occur.
The following pseudo code outlines the process of the algorithm:

```
procedure ELS(old_rules, new_rules)
    both_rules ← old_rules ∪ new_rules
    current_rules ← old_rules
    s1 ← ∅, s2 ← ∅
    while not all nodes updated do
        while both_rules ≠ ∅ do
            while sinks in both_rules do
                prepend sink to s2
                remove sink from both_rules
            end while
            while sources in both_rules do
                append sink to s1
                remove source from both_rules
            end while
```

```
            find node with maximum δ = d⁺ − d⁻ and no incoming fixed arcs
            add node to s1
            remove node from both_rules
        end while
        append s2 to s1
        current_rules ← update nodes according to linear order
    end while
end procedure
```

Each round of the Eades-Lin-Smyth-Algorithm can be done in $O(|V|)$.

Lemma 4.1. The time complexity of the ELS algorithm, adapted for the problem of network updates, is $O(|V|)$ per round.

Proof. Let $D = (V, A_{1,2})$ be the graph to represent the constructed digraph that includes both the set of new rules and the set of old rules $A_{1,2} = A_1 \cup A_2$. We know that $m := |A_1| = |A_2|$, and $n := |V|$.
Building a similar data structure as described in [5] and saving sinks, sources and nodes without incoming fixed arcs according to their δ, takes $O(n)$ time. A look up for sinks and sources can be done in constant time. The maximum δ can be saved to a value x. Some of the nodes only get eligible as other nodes are deleted, while in the original algorithm all nodes can be used. This problem can be solved by e.g. having the bin point to two different lists of nodes, one with fixed incoming, and one without fixed incoming arcs. Only when the fixed arcs are removed from the graph, the node can be transferred to the other list. Therefore, finding the node with the highest delta can lead to a jump (by incrementing the value x) from one bin to another. The maximum distance for this is $d^-(v)+2$, as the maximum δ a node can have (without it being a source) using the topology of the graphs used for this thesis, is 1, and the minimum δ for each vertex is $1-d^-(v)$ (without it being a sink). Afterwards, the maximum distance that can be needed to go to a resulting node without an incoming fixed arc is $d^-(v)+2$ again. Over the whole algorithm this means that the value x needs to be de- and incremented at most $\sum_v 2(d^-(v)+2) = \sum_v 2d^-(v) + 4 = 2m + 4n$ times, as the sum of indegrees is the number of arcs. As, due to the network topologies, $m = n - 1$, the resulting time complexity is $O(n) = O(|V|)$ per round. □

4.2 1-Opt

The original 1-Opt algorithm tries to optimize the position of each node by calculating the amount of back arcs $b = b^+ + b^-$, with b^- denoting the number of incoming back arcs and b^+ the number of outgoing back arcs.
The procedure of the original algorithm is outlined in the following pseudo code:

```
procedure OneOpt(digraph D = (V, A), linear ordering σ) [13]
    for v ∈ V do
        b_0 = b⁺(v) + b⁻(v) according to current position
```

```
        find position p that minimizes b
        b_n = b^+(v) + b^-(v) according to p
        if b_n < b_0 then
            move v to position p
        end if
    end for
end procedure
```

This approach is extended to fit the problem of network updates by only looking at positions that are valid for each node, meaning the positions are bounded by their incoming and outgoing fixed arcs and the positions of their heads and tails in the topologically sorted list of nodes, or, if used as post processing, the resulting linear ordering of the post processed algorithm.
The following pseudo code outlines the process of the algorithm:

```
procedure OneOpt(old_rules, new_rules)
    current_rules ← old_rules
    while not all nodes updated do
        top_sort(current_rules)
        for node in current_rules in top_sort order do
            Find valid positions for the node
            for position i in valid positions do
                calculate b_i = b_i^+ + b_i^-
            end for
            Move node to position i with the smallest b_i
        end for
        current_rules ← update according to linear order
    end while
end procedure
```

For showing the time complexity of the 1-Opt algorithm, we need the following lemma:

Lemma 4.2. [24] The time complexity of topological sorting a digraph $D = (V, A)$ is $O(|V| + |A|)$.

We show the time complexity for the 1-Opt algorithm.

Lemma 4.3. The time complexity for each round of the adapted 1-Opt algorithm is $O(|V|)$.

Proof. Let $D = (V, A_{1,2})$ be the graph to represent the constructed digraph that includes both the set of new rules and the set of old rules $A_{1,2} = A_1 \cup A_2$. We know that $m := |A_1| = |A_2|$, and $n := |V|$. Topologically sorting the nodes (only taking the old rules into account) takes $O(n + m)$ time. Finding the optimal position for each node v takes $O(d(v))$ time. As is is known that $\sum d(v) = 2|A|$, finding the optimal position for each node is done in $O(m)$. Therefore each

round of the algorithm takes $O(n + m + m)$ time, which can be simplified to $O(n + m)$. As, due to the network topologies, $m = n - 1$, the resulting time complexity is $O(n) = O(|V|)$ per round. □

The 1-Opt algorithm can further optimize the linear ordering resulting from the ELS algorithm. The idea of the algorithm stays the same, only that the initial ordering is not computed by the topological sort algorithm, but the ELS algorithm.

4.3 Greedy Approach

The greedy approach simply goes through all arcs in the set of new rules and tries to insert the arc if acyclicity can be guaranteed. For checking for acyclicity again the topological sort algorithm can be used, as if it fails, the graph is cyclic. [24]
The following pseudo code outlines the process of the algorithm:

```
procedure GREEDY(old_rules, new_rules)
    current_rules ← old_rules
    nodes_to_update ← ∅
    while not all nodes updated do
        for rule in new_rules do
            if current_rules ∪ rule is acyclic then
                current_rules ← current_rules ∪ rule
                add tail of rule to nodes_to_update
                remove rule from new_rules
            end if
        end for
        update nodes in nodes_to_update
    end while
end procedure
```

We show the time complexity for the greedy algorithm.

Lemma 4.4. The time complexity for each round of the greedy algorithm is $O(|V|^2)$.

Proof. Let $D = (V, A_{1,2})$ be the graph to represent the constructed digraph that includes both the set of new rules and the set of old rules $A_{1,2} = A_1 \cup A_2$. We know that $m := |A_1| = |A_2|$, and $n := |V|$. Going through each arc in the set of new rules takes $O(m)$. For each arc the topological sorting algorithm is called which leads to a time complexity of $O(n + m)$ for each arc. This results in a time complexity of $O(n + m^2)$. As, due to the network topologies, $m = n - 1$, the resulting time complexity is simplified to $O(m^2) = O(n^2) = O(|V|^2)$ per round. □

In some cases, see Section 7, the greedy algorithm performs better, when the insertion order is not random, but sorted by the indegree of the tails. This

version of the greedy approach will be further called the *sorted greedy algorithm*. For this, the outgoing arcs from nodes with a higher indegree are inserted first. The idea behind this is, that in the round after, it is possible for more nodes to update, as the possibility of a cycle outgoing from this node is eliminated.
An example is shown in Figure 1. The nodes 3, 4, 5 and 6 can only be updated, once the node 1 is updated. The node 1 can only update, when the node 2 is not updated in the same round, therefore, when choosing the node 2 to be updated first, the node 1 can only update in the next round and the nodes 3, 4, 5 and 6 in the round after. This takes 3 rounds. When first updating the node 1 though, the node with the highest indegree, the nodes 2, 3, 4, 5 and 6 can all be updated in the next round, leading to 2 rounds in total. The node 8 can be either way updated in the first round, the node 7 does not need to be updated, as the outgoing arc stays the same.

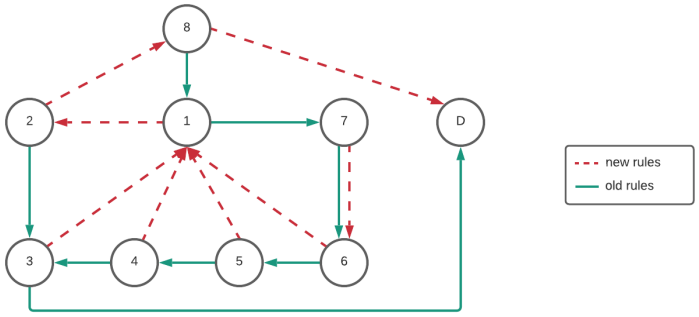


Figure 1: Scenario, where the sorted greedy algorithm can perform better

The greedy algorithm can further be used as a post processing method by greedily trying to add resulting back arcs, while maintaining acyclicity with both the forward arcs and set of fixed rules.

5 Performance of the Algorithms

This chapter examines advantages and disadvantages of the algorithms and compares them to each other for some special cases. The first section further analyses the number of rounds that are needed in the worst case.

5.1 Worst Case Bounds

The number of rounds for each of the algorithms can be trivially bounded by the number of rules to update, for the case in which only one arc per round can be updated. This is the case, if the graphs containing the old rules A_1 and new rules A_2 have the property that each arc (i, j) of the old rules is (j, i) in the set of new rules, except for the arc leading to the destination node. A visualisation of this special case can be seen in Figure 2.

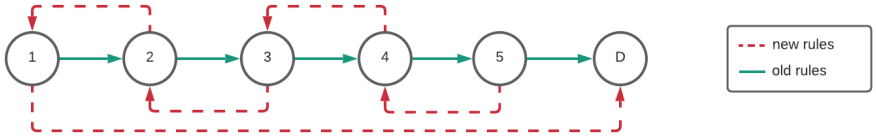


Figure 2: Special case, where all arcs are reverse (except for the one leading to the destination node

As can be trivially seen, in the first round the arc $(1, D)$ can be added to the set of old rules while preserving acyclicity. Therefore, the node 1 can be updated, leading to the deletion of the arc $(1, 2)$ from the set of old rules, as can be seen in Figure 3.

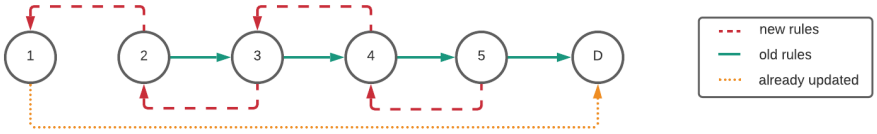
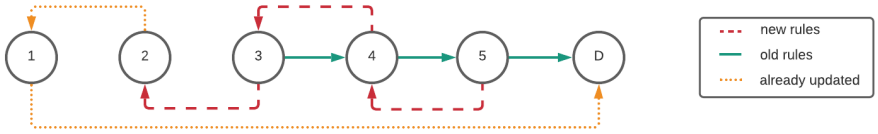
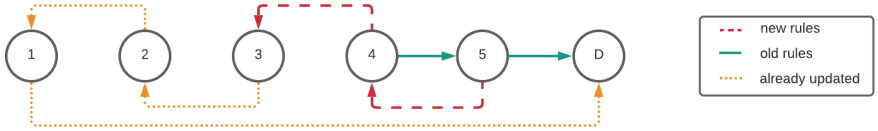


Figure 3: Scenario after the first update

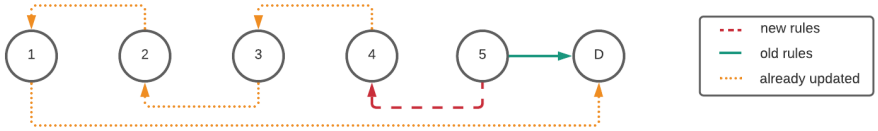
Now it is possible to add the arc $(2, 1)$ while maintaining acyclicity. Therefore, node 2 can be updated. The update of node 2 further results in a deletion of the arc $(2, 3)$, which then leads to the possibility of updating node 3, then the node 4 and finally the node 5, visualized in Figure 4. So in each round, there is only one node that can be updated, resulting in a bound of a maximum of $|A_1| = |A_2|$ rounds, in which each of the algorithms terminates.
While the greedy approach here tries to add each edge to the set of fixed rules and checks for acyclicity, the ELS algorithm in each round identifies the destination node and recursively, when deleting all incoming arcs into the destination



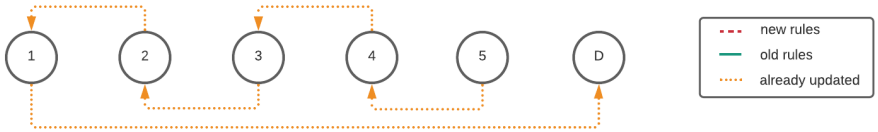
(a) After round 2



(b) After round 3



(c) After round 4



(d) Final result after round 5

Figure 4: Scenario after rounds 2 - 5

node, further resulting sinks. Sources and nodes without incoming fixed rules cannot be found. The 1-Opt algorithm moves, in each round, each node forward right before the node that is the head of the outgoing fixed rule. As more than just one update per round is not possible while maintaining acyclicity, each of the algorithms performs the same in terms of the number of rounds needed. Another worst case, only for the 1-Opt algorithm, is discussed in Section 5.2.3, as the 1-Opt algorithm can have circumstances, where the algorithm does not terminate at all.

5.2 Advantages and Disadvantages of the Algorithms

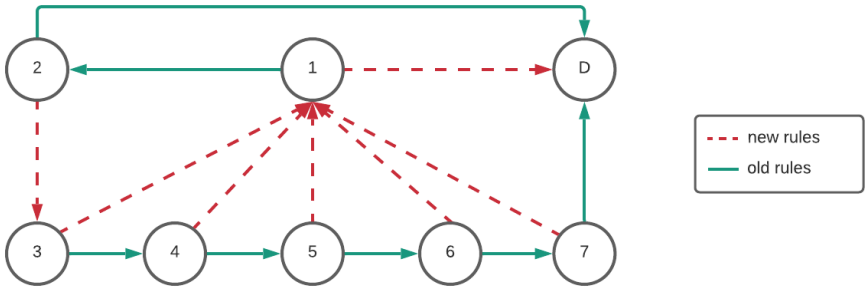
In this section cases are observed, where some of the algorithms show their weaknesses, while others perform well. This makes the disadvantages and advantages clear, that the different heuristics have. For each of the algorithms, one case is shown, where it explicitly encounters a problem that can be more efficiently solved by the other algorithms.

5.2.1 Greedy Approach

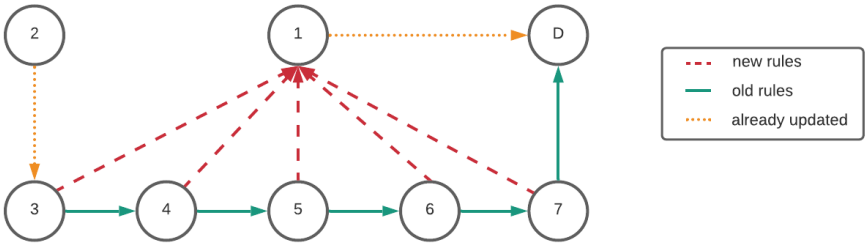
The greedy approach in general randomly tries to add arcs from the set of new rules to the set of fixed rules. Therefore, some cases can be found in which the order of the insertion process can make a difference when it comes to the amount of nodes that can be updated in one round.
When looking at the situation in Figure 5a (example taken from [9]), the order of iterating through the arcs and trying to insert them to the set of fixed rules can make a huge difference. E.g. when trying to first add the arc $(2, 3)$, only one other arc can be added to ensure acyclicity, see Figure 5b. But, when trying to first add e.g. the arc $(3, 4)$, all except for the arc $(2, 3)$ can be added without constructing a cycle, see Figure 5c. This results in different outputs depending on the order of the insertion process. The sorted greedy algorithm does not perform better in this case, as, except for node 1, all other nodes have the same indegree.
This instability is not observed for the Eades-Lin-Smyth algorithm. Identifying the destination node as sink and then further identifying the node 1 as a node without incoming fixed arcs due to the deletion of the arcs $(1, D)$, $(7, D)$ and $(2, D)$ from the union of the fixed and new rules, then further identifying the node 7 as a sink, etc., always leads to the linear ordering of $\sigma = (1, 2, 3, 4, 5, 6, 7, D)$ leading to the same scenario as the worst case performance of the greedy approach as seen in 5b. This can be improved by using the post processing method of applying the 1-Opt algorithm of the resulting linear ordering σ resulting in the linear ordering $\sigma' = (3, 4, 5, 6, 7, 1, 2, D)$, leading to the same result as seen in Figure 5c. The 1-Opt algorithm alone will also return $\sigma' = (3, 4, 5, 6, 7, 1, 2, D)$, therefore having the best performance in this case.

5.2.2 Eades-Lin-Smyth

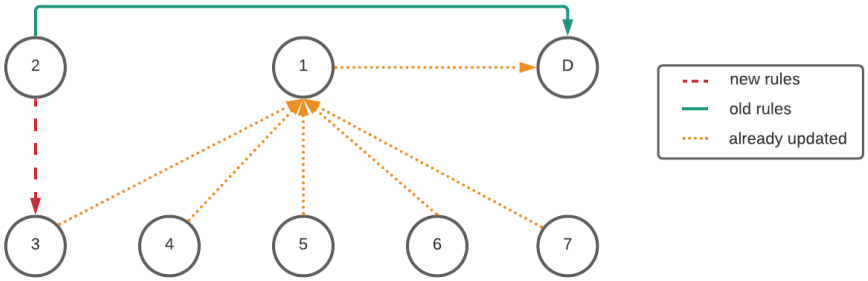
As also seen in the case before, the Eades-Lin-Smyth algorithm can have the problem, that, due to the ordering of those nodes that have no incoming fixed rule, there can be back arcs, that actually do not create cycles when inserting them to the set of current rules. Looking at the new and old rules in Figure 6a, we know that each algorithm will identify the same rules as possible to update, resulting in the situation shown in Figure 6b. The disadvantage of the ELS algorithm is visible when looking at the updates that are identified as possible in the second round. The ELS algorithm first finds the sinks D and 3, and the source 2, as there are no incoming arcs, see Figure 6b. Now there are two possibilities that arise for choosing the node with the highest $\delta = d^+ - d^-$. Both



(a) Initial situation



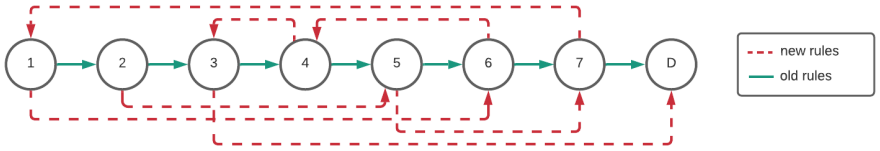
(b) When adding arc $(2, 3)$



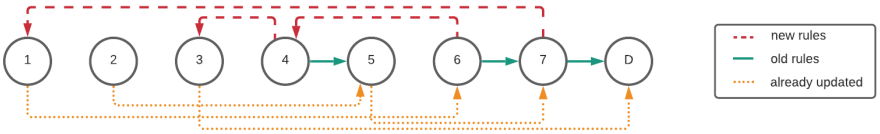
(c) When not adding arc $(2, 3)$

Figure 5: Different performance of the greedy algorithm due to the order of insertion, example taken from [9]

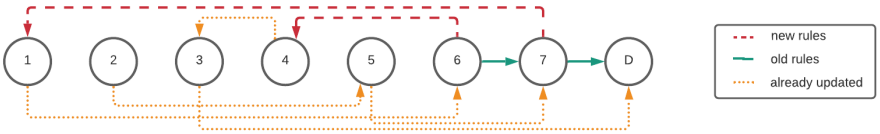
the nodes 4 and 1 have a $\delta = 0$, but when the algorithm chooses the node 4 first, it results in the linear ordering $\sigma = (2, 4, 5, 1, 6, 7, 3, D)$. This means the arc $(6, 4)$ cannot be inserted, even though acyclicity would be maintained, see Figure 6c.
The 1-Opt algorithm (also when using it only as post processing for the ELS algorithm) and the greedy approach both update the node 6 as well, leading to one round less than the ELS algorithm needs.



(a) Initial situation



(b) Situation after first round



(c) When first choosing 4 as the node with maximum δ

Figure 6: Disadvantage of the ELS algorithm

5.2.3 1-Opt

The 1-Opt algorithm has the problem, that in some cases, it may not terminate. One example is shown in Figure 7, were already a few rounds were computed. There are still two rules to update: $(12, 10)$ and $(10, 5)$. The computed linear ordering of the fixed rules at this point, by calling topological sort, is $\sigma = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, D)$. The arc $(10, 5)$ can be inserted without creating a cycle, but, due to the positions that are possible for the node 10, it cannot be moved to a position, where the number of back arcs can be reduced. This is because all nodes that were looked at before, do not have back arcs anymore, therefore not changing their positions, except for node 5, which cannot be positioned after node 6 though, therefore staying at its position as well. As 10 needs to be positioned between the nodes 9 and 11, it cannot be moved.
This problem is solved by trying to insert resulting back arcs greedily when no node to update is found, making it possible for the algorithm to terminate.
Both the ELS algorithm and the greedy algorithm terminate and take the same number of rounds to do so.

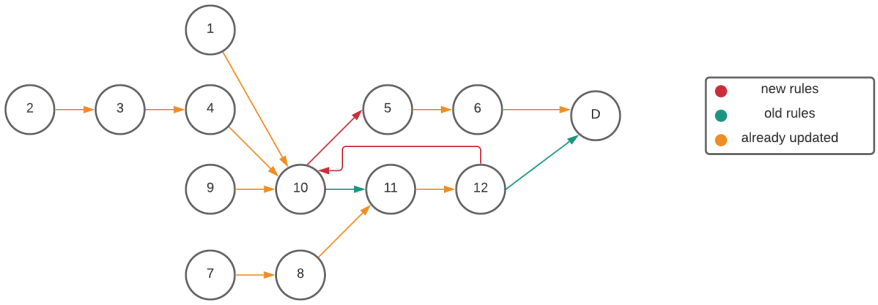


Figure 7: Scenario, where 1-Opt does not terminate

6 Implementation

For the implementation Python 3.8 and the library *NetworkX*[2] (version 2.5.1), which is a network analysis tool, were chosen. The graph data structure needed for the thesis, along with implementations of the topological sort algorithm, a check for acyclicity and planarity, that were needed for the implementation and evaluation of the algorithms, were covered in the *NetworkX* library.
The software for evaluating the performance of the algorithms in a practical approach consists of two parts: The algorithms for the construction of the evaluated instances and the algorithms approximating the minimum feedback arc set, see the class diagram in Figure 8.

6.1 Algorithms

As each of the algorithms uses the same parameters and share some methods, there is a parent class `Algorithm`, from which the different algorithms inherit. Each algorithm class has a `run()` method, that starts the algorithm. There further are helper methods that can be called by the `run()` method, to make the code more modular and clear.
Further, for the ELS algorithm, it is necessary to state a post processing type, which then results in the algorithm being optimized by further trying to greedily add back arcs to the updated rules, or using the 1-Opt approach to further maximize the number of updates, or even both.
Just like the post processing type for the ELS algorithm, for the greedy algorithm it can be specified, whether the rules should be inserted in a random or sorted order.

The second part of the software is the construction of the topologies, found in the packet *topologies*. It consists of the converter, that converts either *CSV* files containing the networks from *Rocketfuel*, or *GML* files from *Topology Zoo*[3], that, similar to *Rocketfuel*, also collects network topologies.

6.2 Construction of the Topologies

Further, the packet holds the subpacket *topology_construction*, that takes care of handling the choice, which topology to construct (depending on the arguments given) and the runs one of the algorithms for constructing the topologies. This can be either shortest path trees (using the Dijkstra algorithm), or random paths, see Section 7. A subfolder *files* also contains the files that were used for the evaluation.

[2] `https://networkx.org`
[3] http://www.topology-zoo.org

6.3 Controller

The controlling function comes from the `main` function in the `main.py` file. It starts, depending on the arguments from the command line, the experiments. There are some predefined experiments in the `experiments.py` file. For simplification, the parameters are not shown in the class diagram. Further, the different experiments call functions from the `plotting.py` file or directly print the results to the command line.

For running the software, the command `python main.py <experiment_type> <topology_type> <instance>` has to be entered. The experiment type can be a single algorithm to evaluate (greedy, sorted greedy, etc.), a comparison experiment, that compares the performance for each of the algorithms for a certain topology and instance, the *mean* experiment, which runs 10 different experiments per instance and statistically evaluates the results, and one experiment, that visualizes the effects of different path lengths for the Random Path topologies.

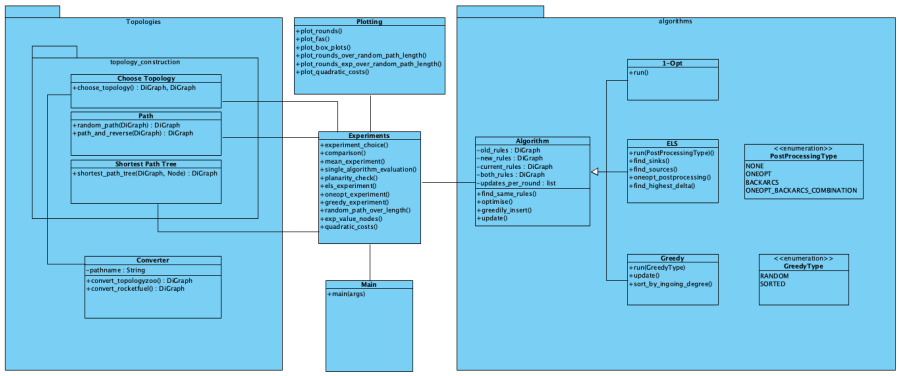


Figure 8: Class diagram of the software for evaluating the algorithms

7 Evaluation and Discussion

Experiments were run for different *Rocketfuel*[4], an ISP topology mapping engine, instances: the *1221*, *1239*, *1755*, *3257*, *3967* and *6461* instances.
Due to the topological properties of the networks, described in Section 2.3, there are two types of topologies that were constructed to experimentally evaluate the algorithms. One is a randomly generated path. Each node has one outgoing and one incoming arc, except for the source, which has no incoming arc, and the destination node, which has no outgoing arc. The other topology evaluated is a shortest path tree from each node to the destination node. Therefore, the nodes can have more than one incoming arc, but the outdegree is still 1.
For plotting and evaluating the results the libraries *Pandas*[5], *Statistics*[6] and *Matplotlib*[7] were used. Further libraries that were used were the *NumPy*[8], the *Random*[9] library, the *Enum*[10] library, and, the *CSV*[11] library.

7.1 Topologies

This section takes a deeper look into the construction of the evaluated topologies and illustrates how these differ from each other.

7.1.1 Shortest Path Trees

The first experiments were done with shortest path trees. For this, a random source node had to be chosen from the nodes in the respective instance. Then, a shortest path tree, from the chosen source node to all other nodes, was computed using the Dijkstra algorithm. Next, the paths were oriented to the source node, so it becomes a destination node. As the Dijkstra algorithm computes this on basis of edge weights, random weights were assigned using the Gamma-distribution, when converting the files. This distribution was experimentally chosen to have a certain complexity in the updating process, as other distributions[12] assigned weights in a way that no cycles existed, or all algorithms terminated within about two rounds, making it impossible to compare the different results. The constructed graph is depicted in Figure 9. Converting was done twice (with the random seeds 74 and 19 for reproducibility), so that there exist two different shortest path trees (using the same source node), that

[4] `https://research.cs.washington.edu/networking/rocketfuel/`
[5] `https://pandas.pydata.org`
[6] `https://docs.python.org/3/library/statistics.html`
[7] `https://matplotlib.org`
[8] `https://numpy.org`
[9] `https://docs.python.org/3/library/random.html`
[10] `https://docs.python.org/3/library/enum.html`
[11] `https://docs.python.org/3/library/csv.html`
[12] Other distributions that were tried out were the uniform, normal and exponential distributions.

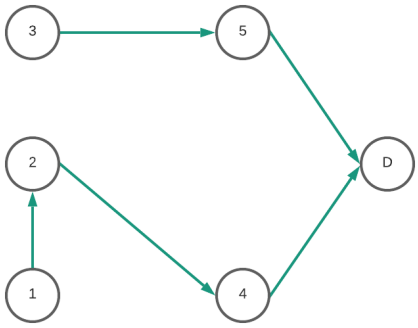


Figure 9: Shortest Path Tree - old rules

could be put on top of each other, constructing the final topology on which the feedback arc set was computed, graphically illustrated in Figure 10.

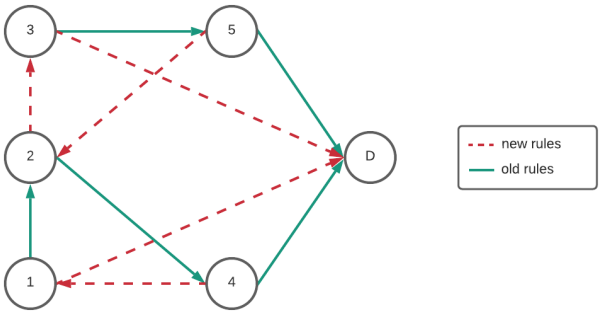


Figure 10: Shortest Path Tree - old and new rules

7.1.2 Random Paths

In the second experiments the different algorithms were compared by taking a look at random paths through the instances. First, a random permutation (using the seed 33 for reproducibility) of nodes was chosen for the construction of a path, where each node was connected to the node before (as outgoing arc) and after (as incoming arc), coming from a source node and leading to a destination node. In Figure 11 this is graphically illustrated, with the node D denoting the destination node.
Further, the new rules needed to be chosen, which also were constructed as a random permutation of nodes, where the last node again is connected to the same destination node as before. When combining those two digraphs, as shown in Figure 12, it can be seen that circles can exist. Therefore, only a certain subset

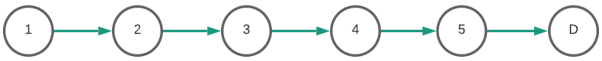


Figure 11: Random Path - old rules

of the arcs in the new rules can be updated in the first round.

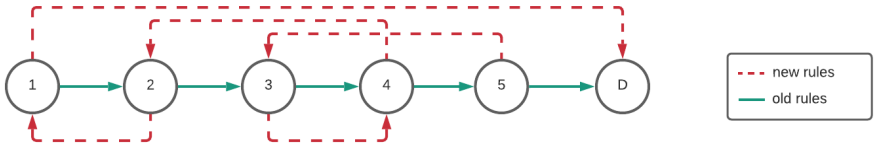


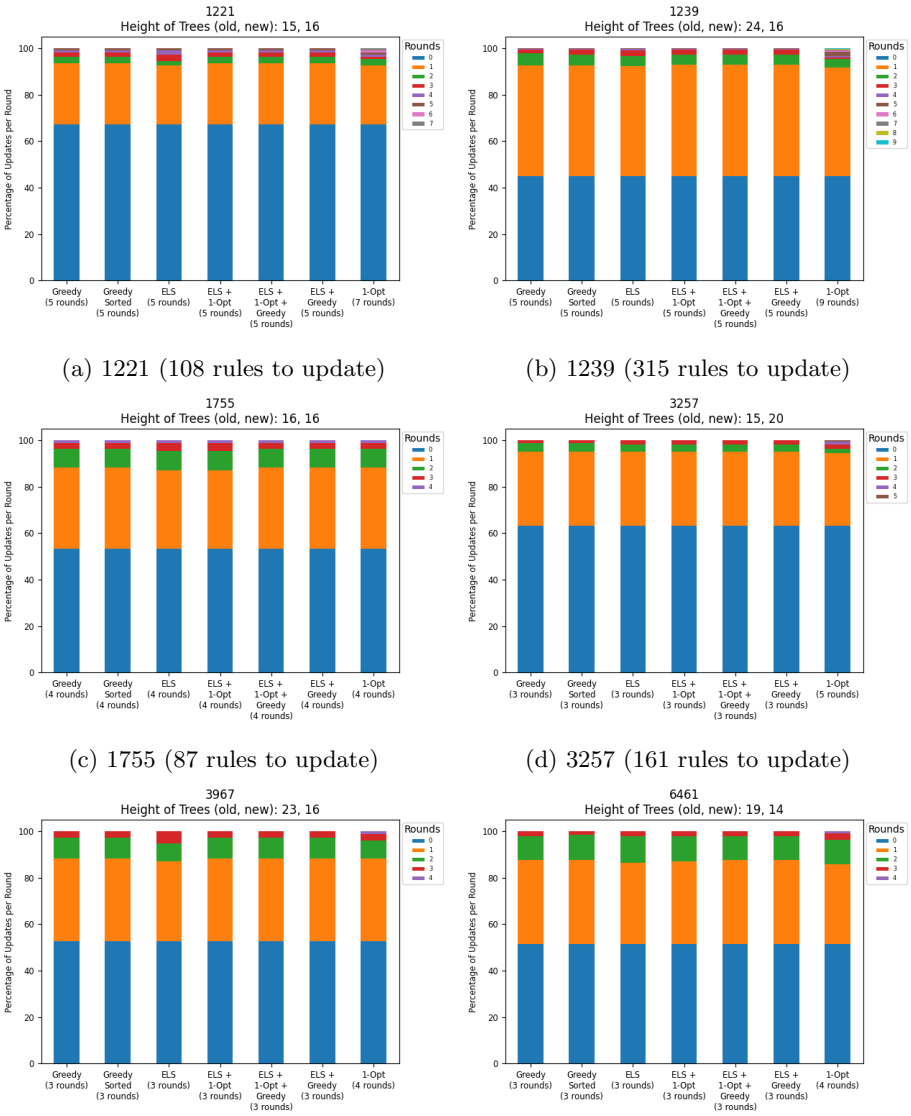
Figure 12: Random Path - old and new rules

7.2 Results

This section presents the results of the different experiments and summarizes and discusses the performance of the different algorithms.

7.2.1 Shortest Path Trees

The construction of the Shortest Path Tree lead to the fact that the intersection of the two resulting trees is at around 50 to 70 percent. Therefore, many nodes do not need to be updated at all. Further, no instance in the first set of experiments of the evaluated topologies was planar.
Looking at the comparison of the algorithms for the different instances in Figure 13, for all algorithms most of the updates can be done in the first round and they, in general, perform similarly, although the 1-Opt algorithm alone needs more rounds until it terminates. The round in which each node is expected to be updated can be seen in Table 1. Also here, the results are very similar for each algorithm. As many nodes stay the same, the value is below 1. In general it can be seen, that the expectancy is mostly higher for the ELS algorithm without post-processing and the 1-Opt algorithm. The difference between the algorithms is better visualized in Figure 14. It can be seen, that the size of the feedback arc set for the 1-Opt algorithm is in most cases bigger than for the algorithms. Further, as can be seen for the *3967* instance in Figure 14e, in the first two rounds the size of the feedback arc set found by the ELS algorithm is bigger, but still the number of rounds is then smaller than for the 1-Opt algorithm. For the case of the *6461* network, the greedy algorithms have the best performance in each round, the sorted greedy algorithm even more so, but still needing the same amount of rounds to terminate as the different versions of the ELS algorithm. A difference between the ELS, post processed by greedily



(a) 1221 (108 rules to update)

(b) 1239 (315 rules to update)

(c) 1755 (87 rules to update)

(d) 3257 (161 rules to update)

(e) 3967 (79 rules to update)

(f) 6461 (141 rules to update)

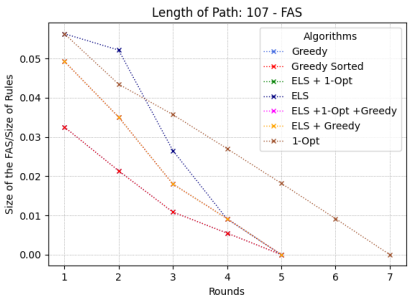
Figure 13: Shortest Path Trees: Evaluation of the different algorithms for different networks

adding back arcs, and post processed by the 1-Opt and greedily adding back arcs could not be found in these experiments.
The box plots, seen in Figure 15, in which 60 different experiments are evaluated, show that most of the algorithms perform similarly when it comes to the number

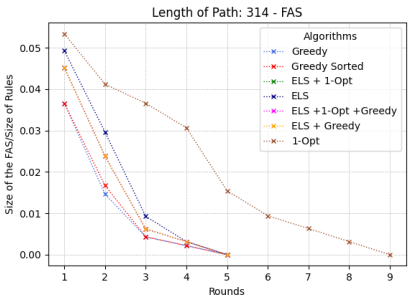
Algorithm/ Instance	**1221**	**1239**	**1755**	**3257**	**3967**	**6461**
Greedy	0.4579	0.6561	0.6279	0.4313	0.6154	0.6286
Sorted Greedy	0.4579	0.6592	0.6279	0.4313	0.6154	0.6214
ELS	0.4953	0.6720	0.6512	0.4375	0.6538	0.6429
ELS + 1-Opt	0.4579	0.6561	0.6512	0.4375	0.6154	0.6357
ELS + Greedy	0.4579	0.6561	0.6279	0.4375	0.6154	0.6286
ELS + 1-Opt + Greedy	0.4579	0.6561	0.6279	0.4375	0.6154	0.6286
1-Opt	0.5421	0.7834	0.6279	0.4875	0.6410	0.6714

Table 1: Round, in which each node is expected to be updated

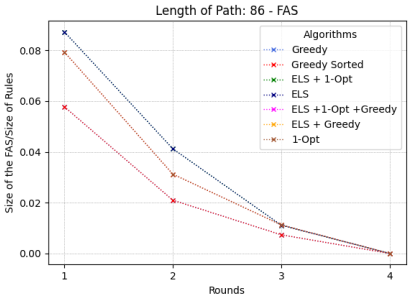
of rounds needed, until the update is finished. Also here it can be seen that the 1-Opt algorithm alone takes the most number of rounds until terminated. All of the quartiles, as well as the minima and maxima, are the same. The difference for the different topologies here can be seen evaluating the geometric mean in Table 2. In four of the 60 different experiments, the graph $G = (V, A_1 \cup A_2)$, containing both the set of old and new rules, was planar.



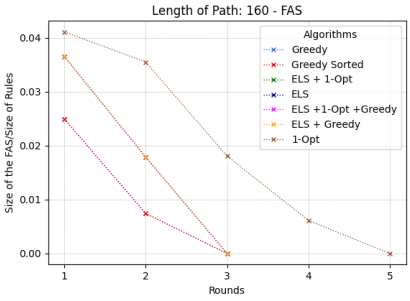
(a) 1221 (107 rules to update)



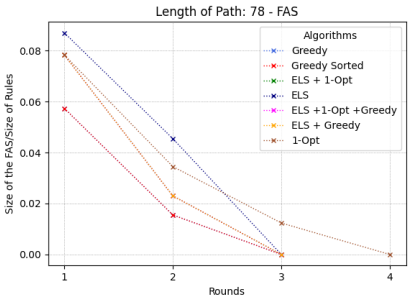
(b) 1239 (314 rules to update)



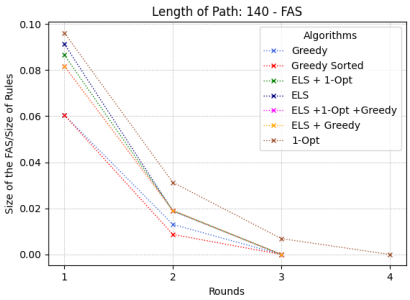
(c) 1755 (86 rules to update)



(d) 3257 (160 rules to update)



(e) 3967 (78 rules to update)



(f) 6461 (140 rules to update)

Figure 14: Shortest Paths: Evaluation of the different algorithms for different networks - Size of the FAS relative to the size of all rules

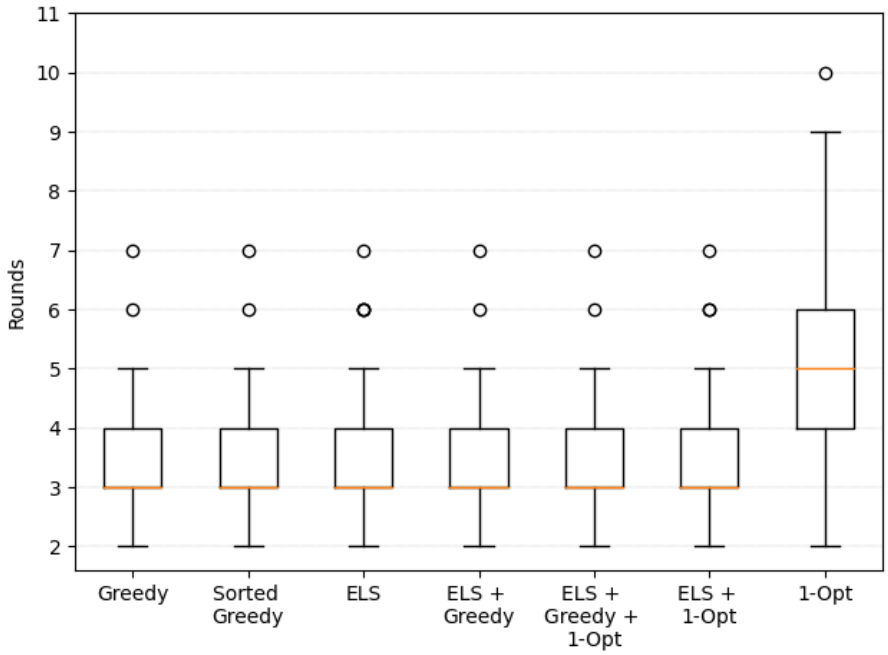


Figure 15: Shortest Paths: Statistical evaluation of the different algorithms for 60 different experiments

7.2.2 Random Paths

The random path topologies were evaluated using the nodes (but randomly generated arcs based on permutations of the nodes) of several networks from *Rocketfuel* to be able to make a comparison to the topology of the Shortest Path Trees. The experiments lead to different results for the different algorithms. Interestingly, in the first round, all algorithms can update the same number of rules, and in fact the same rules. As the linear ordering of the old rules is strict and cannot be changed, only new rules, that are forward arcs when adding them to the set (including those that stay the same), can be part of the update, as there is no other possible permutation of the nodes, that does not have fixed arcs as back arcs. All further arcs are part of the feedback arc set. The order, in which nodes are chosen for the update does not matter, therefore all algorithms perform the same in the first iteration. Afterwards, some differences can be seen. Figure 16 illustrates in which round which percent of rules can be updated. As the data sets used have a different number of nodes, the results for each data set also varies.

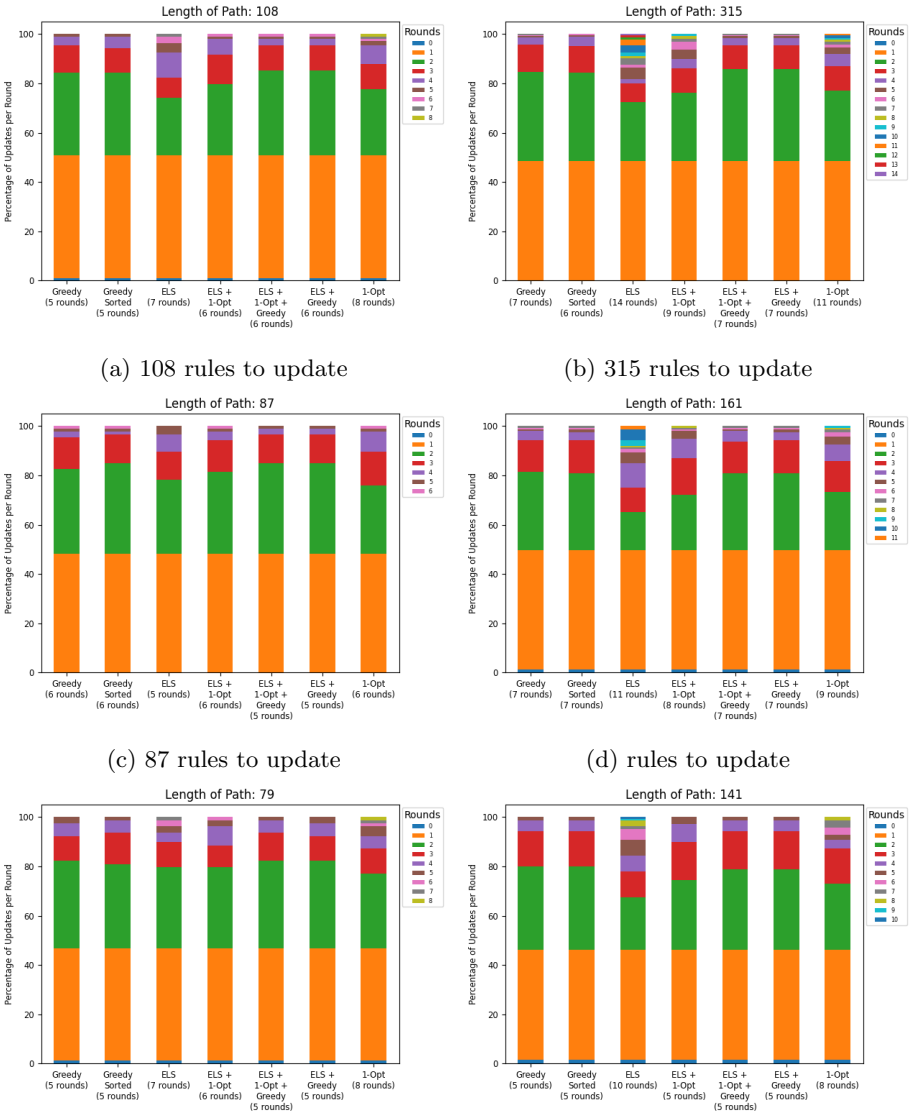
For the topology, having the same nodes as the *1221* data set, which has 108 nodes, plus one further added node, the destination node, and for which therefore 108 nodes needed to be updated, the different algorithms needed 5 to 8 rounds. The greedy approach, in which the nodes where inserted in a random order, as well as the sorted version, perform the same. Both these algorithms and the ELS algorithm that uses greedily adding back arcs, perform the best in this situation. Also, the 1-Opt algorithm performs similarly for this case. The ELS algorithm without post processing performs worst in terms of rounds for these sets of rules. This can be also examined when looking at the other examples. Using greedily adding back arcs, or even the 1-Opt algorithm as a post processing method, increases its performance.

For all other networks, except for the one with 108 nodes to update, the 1-Opt algorithm alone does not perform as well as the greedy approaches, or the Eades-Lin-Smyth algorithm when post processing the linear ordering.

The differences between the greedy and sorted greedy algorithm are minimal, but for the case of the network with 161 nodes to update, sorting made a difference and increased the performance in terms of number of rounds.

In Figure 17, the size of the feedback arc set relative to the size of the set of arcs of the fixed and new rules is shown. Also here it can be seen that the greedy algorithm (when inserting the arcs in a random or sorted order), and using greedily adding back arcs as post processing of the ELS algorithm, perform the best as the size of the feedback arc set found using these algorithms is smaller than when using the ELS algorithm, the 1-Opt algorithm or the ELS algorithm with the 1-Opt algorithm as a post processing step. Using both the greedy approach and the 1-Opt algorithm as post processing for the ELS algorithm does not have an advantage in most cases.

In all cases about 50 percent of all updates where made in the first iteration of the algorithm, this is also the case when using other seeds. Further experiments can be seen in Figure 22, and Lemma 7.1. Running the experiment with ten



(a) 108 rules to update

(b) 315 rules to update

(c) 87 rules to update

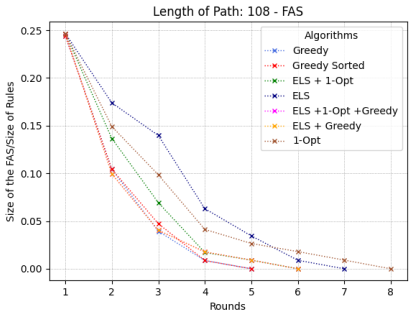
(d) rules to update

(e) 79 rules to update

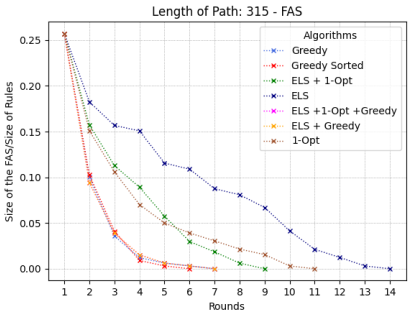
(f) 141 rules to update

Figure 16: Random Paths: Evaluation of the different algorithms for different networks

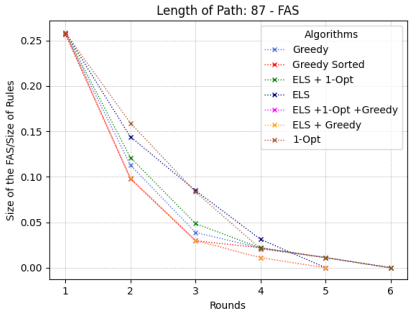
different seeds (from 33 to 42) for each of the data sets and evaluating the performance statistically, shows that the greedy approach (in sorted or random insertion order) on its own or as a combination with the ELS algorithm performs best, the median lies at 6 rounds, see Figure 18. A difference between the sorted



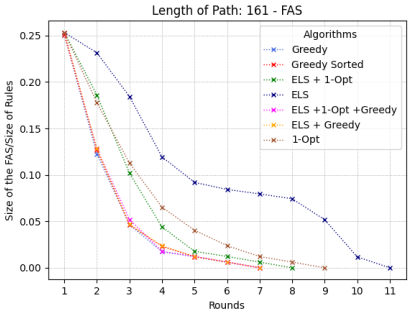
(a) 108 rules to update



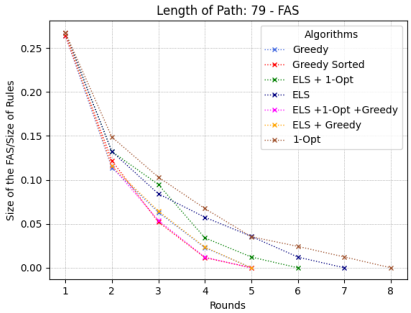
(b) 315 rules to update



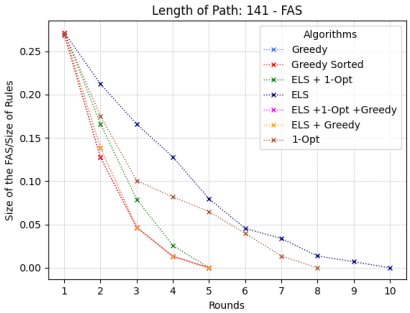
(c) 87 rules to update



(d) 161 rules to update



(e) 79 rules to update



(f) 141 rules to update

Figure 17: Random Paths: Evaluation of the different algorithms for different networks - Size of the FAS relative to the size of all rules

and the random version can be seen. Where 8 rounds is an outlier for the sorted version, 9 rounds is still in the span for the random version, therefore sorting can make the algorithm more stable. Similarly, only using the greedy approach as a post processing method for the ELS algorithm performs a bit better than

the greedy algorithm (in random insertion order) does. The 1-Opt algorithm and the ELS algorithm have a higher range and in general need more rounds as the minimum lies at 6 rounds for both of the algorithms, while for the others it lies at 4 rounds.

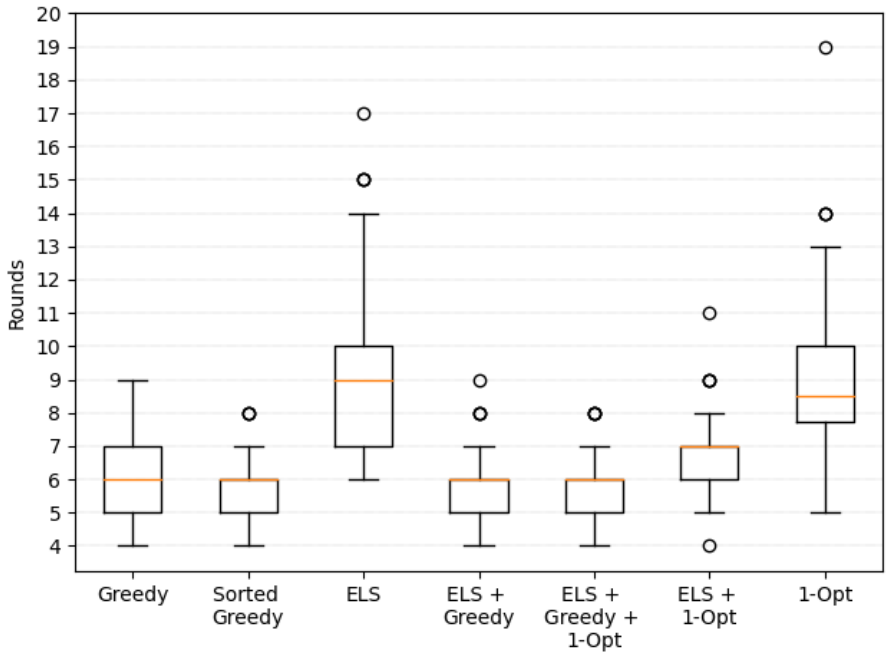


Figure 18: Random Paths: Statistical evaluation of the different algorithms for 60 different experiments

Further evaluations were made with random paths of a certain length, see Figure 19. For this experiment 50 different iterations were done for each path length and the arithmetic mean was calculated. It can be seen, that for smaller topologies, especially for those of lengths that are smaller than 50, it takes less rounds until the updating process is finished (even observing some planar instances), with the general tendency of a logarithmic rise.

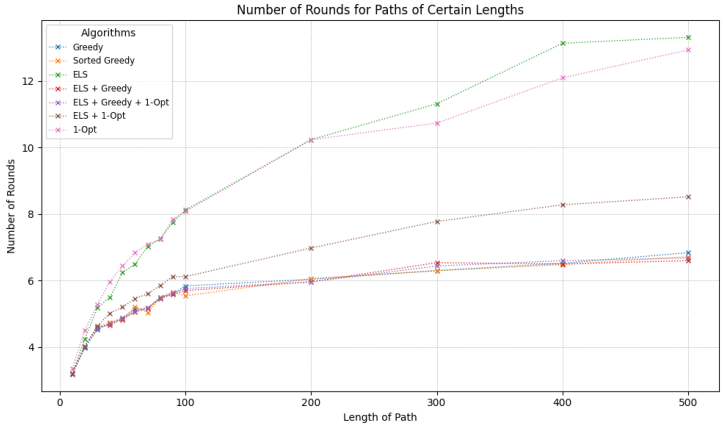


Figure 19: Random Paths: Evaluation of number of rounds for different lengths of paths

Furthermore, it was evaluated, in which round each node is expected to be updated. For most algorithms, the expectancy lies at less than two rounds, the ELS and 1-Opt algorithm also perform worse in this regard. The best performances can be observed for the greedy approaches or the ELS algorithm that uses the greedy algorithm as a post processing method, see Figure 20. When plotting the quadratic costs per node, a significant difference between the greedy algorithms or the ELS, when using the greedy approach as a post processing method, still cannot be observed, see Figure 21.
Also for these experiments, about 50 percent of the nodes can be updated in the first round, see Figure 22. The fluctuation for the experiments with a path length of less than 100 can be explained by the fact, that, due to a smaller sample size, outliers have a bigger impact on the result. Still, it can be seen that the number of updated nodes in the first round (including those that do not need to be updated), is about 50 percent of the nodes in the graph. We can now prove the following lemma:

Lemma 7.1. The expected value for the number of nodes updated in the first round (including those, that do not need to be updated) is 1/2 of all nodes, or $\frac{n+1}{2}$ nodes, with n being the length of the path and therefore the number of updates that need to be done.

Proof. Due to the strict ordering that is given in the first round, it is restricted, that only those new rules that are forward arcs according to the linear ordering

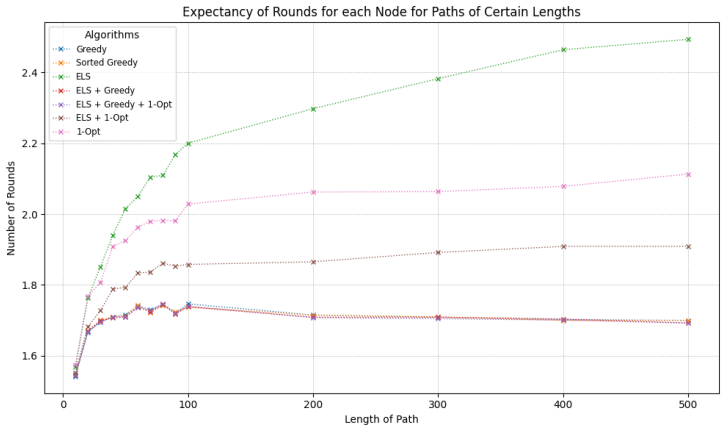


Figure 20: Random Paths: Expected value for the round in which each node is updated, for different lengths of paths

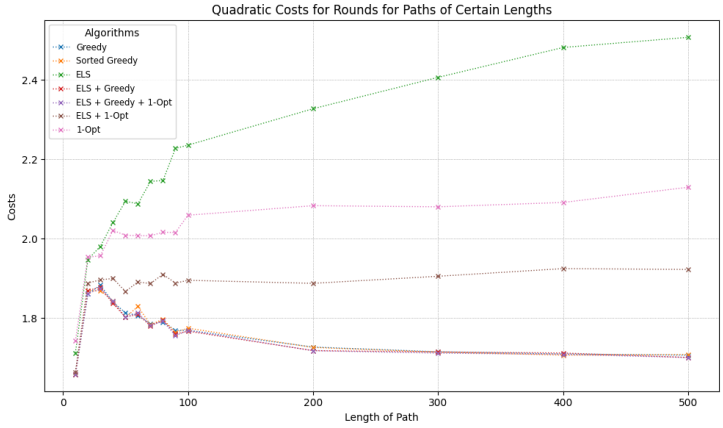


Figure 21: Random Paths: Quadratic costs per node, for different lengths of paths

returned by the topologically sorting the old rules can be updated. Given the nodes $v_0, v_1, v_2, ..., v_{n-1}, v_n$, we know that for each node v_i the probability of the new outgoing arc being a forward arc is $(n-i)/n$, as there are i places in the linear ordering, that would create back arcs, and therefore $n-i$ possibilities for creating forward arcs. The node v_n does not have any outgoing arcs, as it is the destination node. Therefore, the expected value is given by $\frac{n}{n} + \frac{n-1}{n} + ... + \frac{n-(n-1)}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{n+1}{2}$, with n denoting the number of updates that need to be done and $n+1$ being the number of nodes. □

Similarly it can be proven, how many arcs stay the same.

Lemma 7.2. The number of arcs, that stay the same, is expected to be 1.

Proof. For each vertex v_i, the probability, that the vertex after stays the same in the two different permutations, is $1/n$, as there are n different possibilities, where the vertex that was directly after v_i in the first permutation, can be in the second permutation. Therefore this can be summed up: $\sum_{i=0}^{n-1} \frac{1}{n} = n\frac{1}{n} = 1$. □

This can also be seen in Figure 16, were only a small amount of nodes (or no nodes at all), belong to "round 0", which denotes the number of nodes, that do not need to be updated.

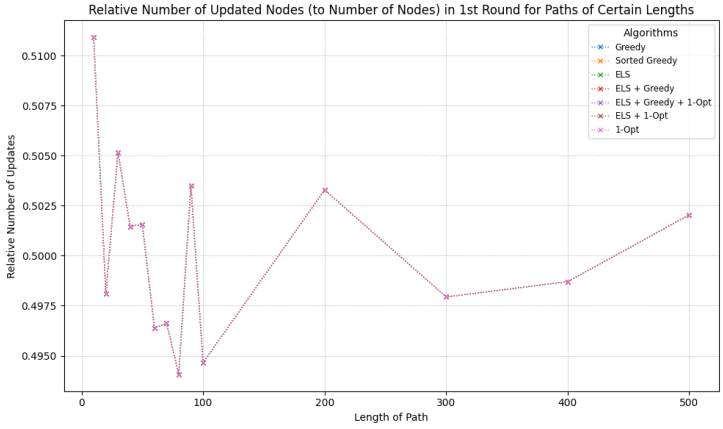


Figure 22: Random Paths: Percentage of updates done in the first round, for different lengths of paths

7.2.3 Summary

In the case of the random paths, a difference between the algorithms is easier visible than for the case of the shortest path trees, as can be seen in the Figures 16, 17 and 18. While in the first round, all updates that can be made are the same for all of the algorithms, a visible difference is found for the second round, in which the ELS algorithm alone in most cases can make the least amounts of updates, followed by the 1-Opt algorithm. Especially for the instance of the network with 161 nodes to update, while the ELS algorithm has updated about 65 percent of the nodes, the greedy algorithm (sorted or in random insertion order), has updated about 80 percent of its rules already. Also when it comes to the number of rounds needed until the updating process is finished, the ELS algorithm alone performed worse than the other algorithms. Looking at the box plots from Figure 18, the ELS algorithm has the highest maximum counting the number of rounds, with the 1-Opt algorithm having a slightly lower median and maximum, but a higher outlier. This can also be seen in Table 2 below, where the geometric mean of the different algorithms for the different topologies is shown.
For the shortest path trees, it can be seen that the ELS alone, as well as with the post processing method of running the 1-Opt algorithm on the resulting linear ordering, perform slightly worse than the greedy algorithm or when using the greedy algorithm as post processing method for the ELS algorithm. The sorted greedy algorithm performs slightly better than all the other algorithms in the experiments. In this case, the 1-Opt algorithm is the least favorable algorithm though, as its median, maximum, and geometric mean, counting the number of rounds, all are the highest of the evaluated experiments.
While the state-of-the-art algorithm is the greedy approach, in the evaluated experiments the sorted greedy algorithm was favorable, also using the ELS with greedily adding back arcs performed equally or even slightly better, in terms of number of rounds, than the greedy algorithm alone, for both of the topologies. When looking at the size of the feedback arc set, e.g. in Figure 14a, comparing the ELS with the greedy algorithm as a post processing method, and the greedy algorithms, the greedy algorithms perform slightly better.

Algorithm/Topology	**Shortest Path Tree**	**Random Path**
Greedy	3.47	5.88
Sorted Greedy	3.46	5.78
ELS	3.58	8.95
ELS + 1-Opt	3.54	6.59
ELS + Greedy	3.47	5.78
ELS + 1-Opt + Greedy	3.47	5.74
1-Opt	4.66	8.89

Table 2: Geometric mean of the number of rounds needed by the algorithms until the updating process is finished for the different topologies for 60 experiments

8 Conclusions and Future Work

There are many possible ways to approximate the feedback arc set, and due to the special case of network updates, where a set of fixed rules never can be part of the feedback arc set, adaptations have to be made. Some of the algorithms that provide good results for the general feedback arc set problem were adapted and evaluated. The experiments in this thesis have shown that the state-of-the-art greedy algorithm has been proven to be performant and to terminate within a small number of rounds, while also updating a high number of nodes in the first rounds. While this algorithm on its own already performed well, sorting the rules beforehand can make it even more efficient, though the differences are quite small. Also using the greedy approach as a post processing method for the Eades-Lin-Smyth algorithm showed that it can further increase the number of rounds for a few cases.
The ELS algorithm or the 1-Opt algorithm on their own are not recommended to use when trying to update the networks in a small number of rounds, especially observed in the case of the random path topologies, that in general were the instances, that were harder to update.

As this thesis only highlights the qualitative aspects of the algorithms, studies on the runtime performance in a practical setting is an open question for research. Other open questions include trying out different approximation algorithms for the feedback arc set and applying them to the topic of network updates, as well as experimenting with variations of the algorithms used for this thesis.
Further, as some planar instances were observed, adapting the algorithms for optimally solving the feedback arc set problem is an interesting task for future work.

References

[1] Ailon, N., Charikar, M., and Newman, A. Aggregating inconsistent information: Ranking and clustering. *Journal of the ACM 55*, 5 (2008), 1–27.

[2] Amiri, S. A., Ludwig, A., Marcinkowski, J., and Schmid, S. Transiently consistent sdn updates: Being greedy is hard. In *Structural Information and Communication Complexity* (Cham, 2016), J. Suomela, Ed., Springer International Publishing, pp. 391–406.

[3] Berger, B., and Shor, P. W. Approximation alogorithms for the maximum acyclic subgraph problem. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 1990), SODA '90, Society for Industrial and Applied Mathematics, p. 236–243.

[4] Bessy, S., Fomin, F. V., Gaspers, S., Paul, C., Perez, A., Saurabh, S., and Thomassé, S. Kernels for feedback arc set in tournaments. *Journal of Computer and System Sciences 77*, 6 (2011), 1071–1078.

[5] Eades, P., Lin, X., and Smyth, W. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters 47*, 6 (1993), 319–323.

[6] Even, G., (Seffi) Naor, J., Schieber, B., and Sudan, M. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica 20*, 2 (1998), 151–174.

[7] Foerster, K.-T., Ludwig, A., Marcinkowski, J., and Schmid, S. Loop-free route updates for software-defined networks. *IEEE/ACM Transactions on Networking 26*, 1 (2018), 328–341.

[8] Forster, K.-T., and Wattenhofer, R. The power of two in consistent network updates: Hard loop freedom, easy flow migration. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)* (2016), pp. 1–9.

[9] Förster, K.-T., Mahajan, R., and Wattenhofer, R. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops* (2016), pp. 1–9.

[10] Garey, M. R., and Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[11] Gavril, F. Some np-complete problems on graphs. In *11th Conference on Information Sciences and Systems* (1977), pp. 91–95.

[12] Gupta, S. Feedback arc set problem in bipartite tournaments. *Information Processing Letters 105*, 4 (2008), 150–154.

[13] Hanauer, K. Linear ordering of sparse graphs, 2018.

[14] Karp, R. M. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 1972, pp. 85–103.

[15] Kudelić, R., and Ivković, N. Ant inspired monte carlo algorithm for minimum feedback arc set. *Expert Systems with Applications 122* (2019), 108–117.

[16] Kurose, J. F., and Ross, K. W. *Computer networking : a top-down approach*, global edition, seventh edition. ed. Global edition. Pearson, Boston München, 2017.

[17] Lucchesi, C. Minimax equality for directed graphs, 1976.

[18] Ludwig, A., Marcinkowski, J., and Schmid, S. Scheduling loop-free network updates: It?s good to relax! In *ACM Symposium on Principles of Distributed Computing (PODC)* (2015).

[19] Ludwig, A., Rost, M., Foucard, D., and Schmid, S. Good network updates for bad packets waypoint enforcement beyond destination-based routing policies. In *13th ACM Workshop on Hot Topics in Networks (HotNets)* (2014).

[20] Mahajan, R., and Wattenhofer, R. On consistent updates in software defined networks. HotNets-XII, Association for Computing Machinery.

[21] Marti, R., and Reinelt, G. *The linear ordering problem. Exact and heuristic methods in combinatorial optimization*, vol. 175. Springer-Verlag Berlin Heidelberg, 01 2011.

[22] Ramachandran, V. A minimax arc theorem for reducible flow graphs. *SIAM journal on discrete mathematics 3*, 4 (1990), 554–560.

[23] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, Association for Computing Machinery, p. 323–334.

[24] Rivest, R., Stein, C., Molitor, P., Leiserson, C. E., and Cormen, T. H. *Algorithmen - Eine Einführung*. De Gruyter, 2013.

[25] Trudeau, R. J. *Introduction to graph theory*, dover ed., 1. publ.. ed. Dover Publ., New York, NY, 1993.