Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Alexander Hedges

Grigori: Does the network work as I expected?

Semester Thesis, Spring Semester 2018
February 2018 to May 2018

Tutor: Prof. Dr. Laurent Vanbever
Supervisor: Ahmed El-Hassany

Abstract

While today network configurations are still largely written manually which tedious and prone to errors, there is a development towards using network configuration synthesizers that automate this process by taking a higher level specification of the network and synthesizing configurations from it. But with increasing reliance on network configuration synthesizers for production networks, it is also important that these tools are bug-free or at least rigorously tested.
We present Grigori, a tool for automatically checking network configurations in a synthesizer-agnostic way and for generating inputs for the NetComplete network configuration synthesizer to exercise its internal OSPF and BGP model. It is capable of generating inputs randomly that cover a large subset of the valid input space and can also generate inputs that exhaust the OSPF and BGP model which NetComplete uses internally.
We implemented Grigori and by checking the configurations generated by NetComplete from our inputs, it helped uncover numerous bugs in NetComplete including one bug in the BGP model code itself. The checking part of the implementation scaled well to all configurations which NetComplete could synthesize in a reasonable time and it is parallelizable if a further performance increase is needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

NetComplete [2] is a tool for synthesizing networking configurations from existing partial configurations (the sketch) and routing requirements. The resulting configuration has to be a strict superset of the given configuration sketch and when run on a Cisco router, it has to forward packets according to the routing requirements.
Where Grigori comes in is checking that the synthesized router configuration is correct. In addition to that Grigori generates input for NetComplete that covers its internal model. By testing NetComplete with this input, bugs can be found and ultimately some correctness guarantees can be made about its operation.
The NetComplete functionality we want to test is OSPF and BGP. The idea for showing that it performs the way we expect it for those protocols is to first extract a model of how we believe OSPF and BGP is implemented in NetComplete. In essence the model is the structure of the SMT that is generated by the tool. To cover all the model, input is generated so that all states in the model are covered once and the positive states (states where the requirements can be met) can be synthesized by NetComplete from partial sketches. After showing that the tool generates configurations for the cases where it should and returns a message saying no configuration exists for the cases where it shouldn't, we then test whether the produced configuration is actually implementing all requirements by running it in a emulator which runs Cisco router images. If we can show this, NetComplete correctly implements the extracted model which in turn is a strict subset of the Cisco model.
We will start by giving a high level overview of Grigori in the following chapter.

Chapter 2

Overview

In this chapter first the necessary terminology is defined in section 2.1 which in our case is the term routing requirement. We motivate the problem and show the workflow of our solution in section 2.2. Finally in section 2.3 we give a short outline of the rest of the report.

2.1 Definitions

2.1.1 Routing Requirements

Requirements are an important part of Grigori and NetComplete as they are the operator's way of expressing his intent for the behavior of the network.
A requirement specifies what should be reachable from where and via which path (and consequently also what should not be reachable). Each requirement has a destination that specifies which IP network should be reachable and a path which specifies the ways in which to reach the destination. The path is, depending on the requirement type, either a sequence of router identifiers or an sequence of requirements. Requirements also have a protocol which specifies over which protocol the path should be learned.
There are four types of requirements currently supported by NetComplete:

Simple routing requirement A requirement says the destination should be reachable via the given sequence of router identifiers.

ECMP routing requirement ECMP stands for equal cost multi path. The paths are an unordered list of simple routing requirements. If unobstructed, all paths should be available at the same time to choose from to reach the destination.

Ordered routing requirement The paths of an unordered routing requirements is an ordered list of simple routing requirements that should be preferred in that order. The first available path has to always be taken to the destination.

Any-Path routing requirement For this requirement the paths are an unordered list of simple routing requirements. If at least one path is available, the path taken to the destination has to be in the paths list.

2.2 Big picture

As seen in Figure 2.1, Grigori integrates with NetComplete at multiple points.
NetComplete is a tool that takes requirements, the network topology and a partial configuration file for each router. It fills in the missing parts such that the network in which each router uses the completed configuration files behaves as expected. Or does it?
This is where Grigori comes in. In order to make sure that the network forwards according to the requirements it has a component that simulates the network and reports any mismatches between the forwarding tables and the requirements. Grigori can also generate inputs for NetComplete. The goal is to show that NetComplete does the correct thing for all possible inputs and is therefore operating correctly.

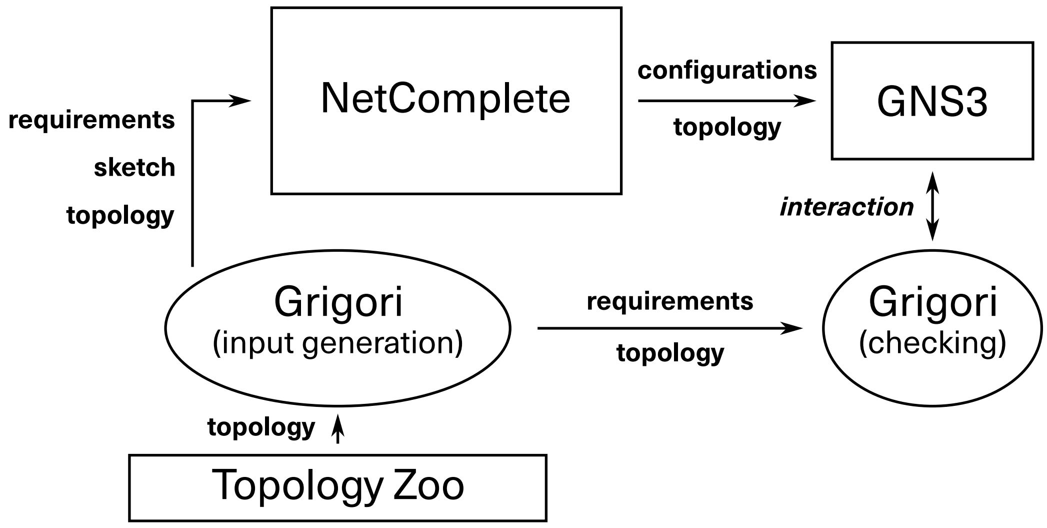


Figure 2.1: Overview of how Grigori integrates with Netcomplete

The reason why Grigori can show correctness beyond what unit tests can do is because it also tests its own assumptions (the model) against the ground truth which is the emulated Cisco router image.
Since the input space for NetComplete is infinite, testing all inputs is not possible. To get as close as possible to those guarantees a combination of random and smart (model covering) inputs are used here to test both networks that use OSPF or BGP for path selection.

2.3 Outline

In this report we will proceed by first exploring the design space a bit, before showing the design choices that we picked for Grigori.
To put that in context, the design part (chapter 3) will be followed by an evaluation in chapter 5 where we look at the concrete results we got from it in terms of coverage and bug reports. Also performance of the implementation will be touched upon briefly in section 5.3.
Having learned a lot during this project, before the concluding remarks in chapter 7, chapter 6 will talk about all the things we would do differently next time and all the things that could still be done in the future to improve upon this work.

Chapter 3

Design

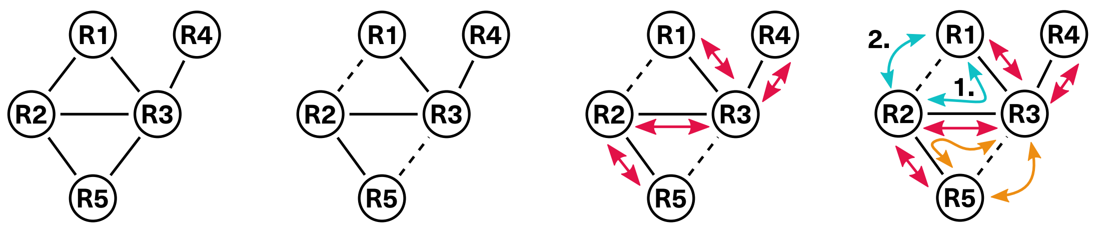
In this chapter we discuss the design space and decisions, separated into the design decisions concerning OSPF in section 3.1 and the design decisions concerning BGP in section 3.2.
In general the design decisions for generating random requirements involve weighting coverage (how much of the possible input can be reached) vs false positives (where the input is not valid or consistent). In the ideal setting, the whole space would be covered and the checker could enforce that invalid input does not lead to output. But since finding out if an input is valid is quite hard so trade-offs have to be made.
For smart inputs (inputs which cover the internal model completely) one of the challenges is striking the balance between a simple and useful model. Another challenge is covering a given model with inputs that trigger the internal state.

3.1 Generating OSPF input

The model for OSPF is quite simple. The OSPF works is that it performs Dijkstra on a weighted directed graph[3]. From this follows that the corresponding SMT constraints for each requirement are that the total costs of the paths (which is the sum of the link costs) covered by the requirement are higher than the costs of all other remaining paths. For ECMP requirements there is an additional requirement that the cost of the covered paths have to all be equal and for ordered requirements the costs have to be decreasing in the order in which they are mentioned in the paths array.
So to test OSPF, it suffices to cover the input space of requirements and edge weights since it does not interact with the config in another way.

3.1.1 Random OSPF requirements

From the beginning we made the assumption that the cost of simulating the network and checking it is proportional to the amount of configurations we need to check given a fixed network. As seen in the results (section 5.3) this assumption holds.
So assume we have two sets of requirements. We could test both sets independently, but this does not tell us anything about whether the configuration is correct when it is synthesized from the union of the two sets. So to show correctness we don't get around feeding the union of the requirements to the synthesizer.
Inversely, if the union of requirement sets is tested (this requires that there exists a configuration where all those requirements are satisfied simultaneously), we can, under some reasonable assumptions, say that giving any of those sets to the synthesis method as input would also produce a valid configuration. The intuition behind this is that adding a requirement to the input can never make the tool's job easier. This has the added benefit that by testing a configuration that was generated from n requirements, we don't need to test synthesis on all the 2^n strict subsets of those n requirements. Summarizing, it is best to combine as many requirements as possible (which don't conflict) in the input to the synthesis, because this maximizes the coverage to computing time ratio (assuming that the computing time is proportional to the checked configurations).



(a) The initial topology

(b) Generate a random spanning tree

(c) Add simple routing requirements for all edges in the tree

(d) For the remaining edges, randomly add ordered or any-path requirements

Figure 3.1: The different steps of filling a graph with OSPF requirements until saturation

Guided by this principle we sought to combine as many requirements as possible when synthesizing the graph. To do this, first a spanning tree was generated. On the spanning tree all edges were covered by a simple routing requirement in both directions. Since it is a spanning tree those simple routing requirements are always satisfiable. If the original topology was a spanning tree, this is also the only applicable requirement type. The remaining edges in the graph are used for other types of requirements (either any-path or ordered requirements) and the preferred path is always the path that goes along the spanning tree. An example of this can be seen in figure 3.1. First the a possible spanning tree is generated from the graph.

Because for a given topology there is usually an exponential amount of possible spanning trees we can't possibly generate all spanning trees and all requirements. For a medium sized graph of about 30 routers there are approx. 4 Billion spanning trees while the largest graph in topology zoo has around 10^{100} trees (this number can be computed as done in the code using Kirchoff's Matrix-Tree Theorem [4, p. 28]).

Given it is not feasible to cover the whole search space, the next best thing is to uniformly sample random requirements from the space (à la Monte-Carlo). Of course, here uniformly means uniform with respect to our own parametrisation for generating spanning trees, if we could parametrize the requirement space according to model coverage we wouldn't be doing this. To randomly generate a spanning tree, first the cycle basis is computed. This is a set of cycles in the graph from which all cycles can be generated via hopping to the same node on another cycle. A necessary but not sufficient condition for having a spanning tree is to cut all the cycles in the basis once. Therefor we simply cut all cycles once at a random position and see if we get a spanning tree. If not we repeat.

This way of generating spanning trees is well suited for random generation. If we want to generate all possible spanning trees, this method does not work well as one could generate the same tree more than once with this method. Because at first the plan was to generate trees exhaustively I also wrote a method that generates all possible spanning trees by doing a depth-first backtracking search (where edge inclusion is the decision), but as mentioned before this is not feasible even for small examples.

For the remaining non-spanning edges a random bit vector is generated which decided whether it is part of an ordered or any-path requirement.

How can this cover the case of arbitrary requirements (where for instance an ordered requirement has two paths which both have more than length 1)? The key is to think in terms of equivalent requirements. For instance assume that we have an ordered requirement with two paths. The less preferable path has length 1. In terms of requirements this is stricter than having the destination moved towards the start along the longer path (see figure 3.2). If the the first requirement holds, the second one will always hold since removing links and adding them to the less preferred path will always make the preferred path more preferable (assuming non-negative link weights).

ECMP requirements are handled that the start and end is chosen at random from the nodes and then a random selection of simple paths from the start to the end are used in the requirement. Note that this only tests the model for ECMP superficially. The reason that ECMP is handled separately is because you can't apply this requirement implication reasoning from above as for

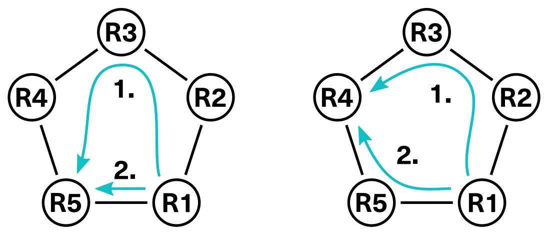
ordered or any-path requirements. This has the small benefit that later for BGP requirement generation, where ECMP requirements are not allowed, we can simply reuse the requirement generation routine without modification.
While this is a form of random requirement generation and we can use that for testing input, there are a myriad of drawbacks:

- First of all an undirected graph is always assumed in all operations above. But the network graph we have is directed and it is not quite clear how to expand the idea above to work with directed graphs.
- Given the large amount of spanning trees and requirement type combinations, the fact that 32 bit seeds are used means that a lot of combinations will never be generated.
- The combination of ECMP with other requirement types was never tested.
- A whole dimension of the input is completely ignored, which is edge weights.

Some of these problems might be fixable (see discussion in section 6.2.1), but only at the cost of a lot of extra complexity and some not at all.
So essentially all problems here are in terms of coverage meaning that even if we exhaustively generate all inputs that this method gives us, feed them to the checker and they verify, we still can't guarantee that OSPF is correctly implemented and works as we expect it to. So seeing the limitations of this approach, we chose to try to achieve our primary goal by a different means, which is all the next section is about.

3.1.2 Smart OSPF inputs

In order to say something about the correctness of the OSPF synthesizer we need to first model OSPF. While the model described at the beginning of the section is useful to understand how it works, to efficiently cover it we need a model that is closer to the actual implementation. If we can show that the implementation works according to the model and that for all points in the model the synthesized output performs as expected in the Cisco emulator, we can guarantee that our model of OSPF is a strict subset of the model that Cisco has for OSPF. In other words: our model is a useful simplification of the real thing.
Note that this implementation specific model can not be used to test other implementations of OSPF synthesizers as it assumes the verifier treats same cases the same. Assume a program that hard-codes solutions to all of the finite set of problems. As the model space for OSPF (graphs, weights) is infinite and we can only test a finite set of cases, we can never distinguish it from a correct program. From a practical perspective though this is not such a large problem because the missing guarantees can be achieved via unit testing and our model was more or less directly translated from the SMT-generating code in NetComplete so it is reasonable to assume



(a) Requirement 1 (b) Requirement 2

Figure 3.2: Requirement 1 is stricter than Requirement 2. If it holds, Requirement 2 automatically also hold.

```
if $type$(req) $=$ simple then
    $\forall p \in \text{paths}_{\text{other}} : cost(p) < cost(\text{req.path})$
else if $type$(req) $=$ any-path then
    $\forall p \in \text{paths}_{\text{other}}, \forall p' \in \text{req.paths} : cost(p) < cost(p')$
else if $type$(req) $=$ ordered then
    $\forall p \in \text{paths}_{\text{other}}, \forall p' \in \text{req.path} : cost(p) < cost(p')$
    $\forall p, p' \in \text{req.paths} : index(p) < index(p') \implies cost(p) < cost(p')$
else if $type$(req) $=$ ECMP then
    $\forall p \in \text{paths}_{\text{other}}, \forall p' \in \text{req.path} : cost(p) < cost(p')$
    $\forall p, p' \in \text{req.paths} : cost(p) = cost(p')$
end if
```

Figure 3.3: Our model of OSPF

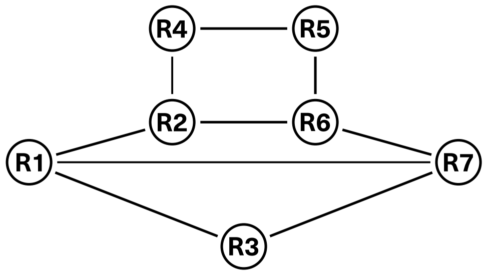
that the model reflects the implementation. In addition to that the randomness in the random tests from the previous section should get rid of the fear of hard-coded solutions.

The OSPF model that was extracted from the implementation can be seen in figure 3.3. It treats each requirement type differently and adds constraints for each requirement that is added. In the following the start and destination of the requirement will be called A and B respectively. For simple routing requirements it adds constraints for all paths leading from A to B that the mentioned path has the lowest weight globally. For ordered paths it adds constraints that all the mentioned paths have a lower weight than all the paths from A to B that are not mentioned and the weights of the mentioned paths are strictly monotonically increasing. any-path constraints are similar, but without the explicit ordering within the group of mentioned paths. ECMP adds restrictions that each of the mentioned paths has the same weight and that no other path from A to B has a lower weight.

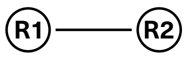
From the given model we can see that there are multiple cases that exercise the generated SMT. First of all, for each requirement type, there is the case that there are no other paths between the source and the destination. Then of course there are the cases where there are other paths than the ones mentioned in the requirement. For the ordered requirements the paths are also treated differently according to their order in the paths list. So all we need to do here is basically to show that the synthesizer can enforce the given partial path ordering for every type of requirement in both cases where there are additional paths and where there are none.

The best way to show if a certain configuration can be generated by the synthesizer is to have the concrete link weights set in a way that the remaining symbolic weights can only take one possible value.

In order to cover the model points, a particular graph was chosen with multiple different paths between two nodes (see figure 3.4). For each of the non-simple requirements the cases of covering all possible paths and not all possible paths were covered. So for instance for ordered requirements there is a case where a certain ordering is enforced on all paths and a case where a certain ordering is only enforced on a subset of all paths. Also the preset weights were set in a



(a) A flexible topology with 4 distinct paths from R1 to R7. Two paths are partially overlapping.



(b) Graph used to test a simple routing requirement with no other paths.

Figure 3.4: The two graphs used to test the OSPF model

way that there is only one possible way of setting the remaining weights so that the requirements can be met.
For simple requirement a graph with two nodes and one edge was done to check the case where the simple requirement is on the only path and one case was done where the simple path was one of the possible paths in the graph from above. Again, here the graph was given weights so that there is only one way to set the weight.

3.2 Generating eBGP inputs

The way in which BGP works is that at every router BGP advertisements are received from neighboring routers and those advertisements are filtered after being received and before being sent. Each router can only forward one advertisement per destination so if multiple advertisements with the same destination make it past the input filter, the router selects one advertisement based on attributes of the advertisements. This function is called the select function.
In our model the select function compares the advertisement pairwise and once it finds a point in the select function hierarchy where they differ it selects one of the paths according to the current point. The select function hierarchy is:

1. Highest local-pref value is selected
2. Shortest AS-path is selected
3. Lowest MED is selected (for routes coming from the same AS)
4. Advertisements received via eBGP are preferred over ones received via iBGP
5. Lowest IGP metric is selected
6. Lowest router id acts as a final tie-breaker

Note that the tool also supports a rule which would be situated between point 2 and 3 in the select function which selects paths with the lowest origin type (in ascending order: IGP, EGP and incomplete), but unless the route is redistributed from OSPF (and hence its origin type incomplete) nowadays all route origin types are IGP. NetComplete cannot synthesize configurations that specifically trigger this rule as it does not know how to redistribute routes from OSPF to BGP. Therefore this step is left out in our select function model.
It is important to keep in mind that the announcement originates from the destination, so the path and consequently the packets go in the reverse direction of the announcement.
Before and after the select function the input function and output function respectively is applied which consists of route maps. A route map is a sequence of match predicates that the advertisement is evaluated against consecutively until a match is found. If no match is found the advertisement is dropped. If a predicate matches the associated actions are performed. Actions can drop an advertisement or modify its attributes.
The concatenation of input function with the select function and then the output function will be referred to as the BGP function here.

3.2.1 Random eBGP requirements

Similar to the random OSPF requirement the first idea was also to generate random BGP requirements in order to catch bugs. This is harder than doing the same thing for OSPF because the model is more complicated and in addition to the topology and requirements, the sketch is an important part of the input, which adds another set of dimensions to the possible input.
Because the general problem of truly random BGP input seems to be complex I decided to use one general semi-symbolic sketch for all inputs and combine it with topology-zoo sourced topologies and random requirements.
To increase the amount of things that are tested and venture into the unit testing area, the method that generates the general semi-symbolic sketch takes an optional random number generator which causes it to instantiate the route map randomly.
Despite its simplicity, this quite basic testing already uncovered a few bugs and helped us better understand the problems that come up when generating configuration sketches. For instance

we found out that the method of checking if the requirements are implementable which was called before calling the actual solver gave false positives as well as false negatives.
To generate the BGP requirements, the random (non-ECMP) OSPF requirements from the previous section were taken and put through a procedure that changed the protocol of the requirement and the destination. Apart from that the topology had to be changed to put each router in a separate AS and connect the routers with eBGP sessions instead of OSPF connections.
The main challenge in using OSPF requirements directly is that BGP and OSPF implementable requirements are not a strict subset or superset of each other. For instance, BGP does not support ECMP while OSPF is in general less powerful. In our case ECMP is not a problem since those requirements are generated separately in the OSPF requirement generation process. The cases where a set of requirements can only be implemented using BGP are not discussed here since the OSPF-requirement generation method does not generate them, this is one of the limitations of the BGP random requirement generation process.
Right now, whenever NetComplete fails to produce a working configuration, it eliminates requirements using a coin flip for each one. This reduces the requirements by about half on average and decreases the chance of a conflict due to requirements. That NetComplete fails to synthesize a configuration can happen due to many reasons, one being existing bugs (in both Grigori and NetComplete) but also due to the fact the randomly chosen sketch happened to not be sufficiently general. Only producing a warning avoids producing too many hard errors for the same condition and also avoids false negatives.
The problem with the random BGP requirements is similar to the OSPF case: coverage. While the method of generating the input is simple, the generated input only covers a very small part of the input space and the model. With the given setup only a limited number of route maps are created and mostly it suffices to set the *local pref* in the route map in order to achieve the requirements. This means the select function is seldomly exercised beyond point 1.

3.2.2 Smart eBGP inputs

To test/verify the BGP select function is implemented correctly we have to show that each stage can be triggered, that NetComplete can synthesize all stages correctly and that the ordering of the stages is correct. The select function is checked at a single router so because of that the topology centers around that router and it has multiple neighboring routers it can receive from and one client AS (router) it announces its selected route to.
The different setups can be seen in figure 3.5. For the setups with simple requirements the tested router 'R1' has two neighbors, one from which it receives the announcement and the other which it announces it to. A variation of the same topology with additional connected but non-participating routers is also tested. These tests are to make sure that the case where the select function does not have to be invoked works. For the other case the central router has two or three iBGP neighbors (which are virtually connect full mesh) each of which have one eBGP neighbor from the same AS. This setup allows enough flexibility in the sketch to test all of the stages of the select function. In the case when stage 4 of the select function has to be tested the topology is slightly modified such that one of the internal routers is omitted and 'R1' is directly connected to its eBGP neighbor.
To test that each stage can be triggered we have to use the sketch to force the decision to be made at that level. This can be easily done by having the decision factors at all other stages set to equal except at the tested one. To test that NetComplete can correctly synthesize the configuration at each level the parts of the sketch that are needed to set the decision variable are left symbolic for NetComplete to set correctly such that the requirements are satisfied. In the implementation, only this second method is used because the check will afterwards automatically determine if the symbolic parts were set correctly (in which case they are set to the same values as we would have set manually in the sketch).
This test alone does not yet guarantee that the ordering is correct. Each configuration can be mapped to a vector of length 6 that indicates whether the comparison at the bit index would be in favor of the preferred path, not in favor or equal for both paths. To guarantee that the ordering of the select function is implemented correctly one would have to check all possible combinations (3^6) and determine if the expected result is produced (alternatively to only check the ordering assuming the other tests succeed, testing only a subset of those combinations would also work).
This is not done here (we assume the ordering is correct from what we see in the code where

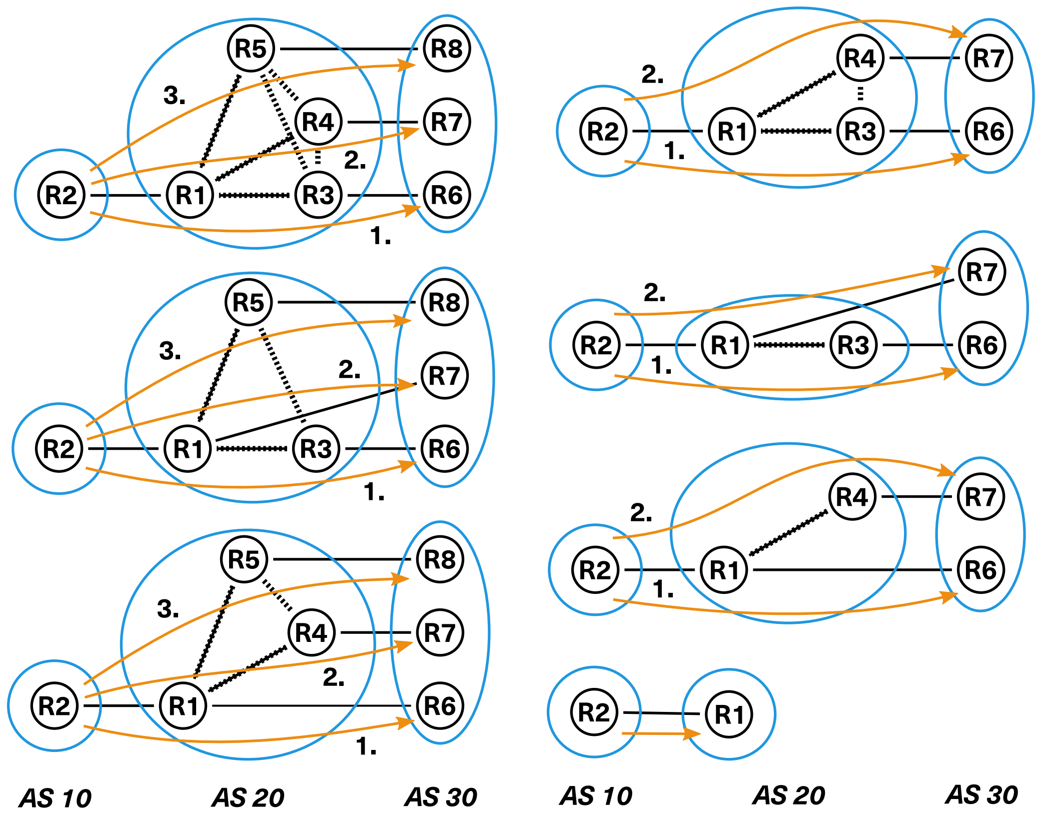


Figure 3.5: The different topologies used to test the BGP select function. The missing routers in AS 20 are needed to test the rule that announcements are preferred when received via eBGP over iBGP.

the BGP model is implemented), but would be an obvious next step in our work.
Note the currently no any-path requirements are used at all in testing of the select function.
In the current implementation, Grigori only tests the select function. Testing the route maps would also have to be done for complete check of the BGP function. This is left for future work.
Once the BGP function is verified, it remains to show that if each router correctly implements the BGP function in the sense of the requirements, that the requirements are globally met. This can be shown via the following inductive reasoning.

Inductive reasoning why it suffices to test the eBGP implementation on one router

A set of requirements is valid if there is no subset that directly causes another subset to not be realizable on the given topology. In other words, non-conflicting: requirements on which a correct synthesis method would return sat.
We say a network is correct, if for all valid sets of requirements and all possible ways of link failures: the requirements are simultaneously met by the announcements that arrive at the respective requirement destination.
We say a router is correct, if for all valid sets of requirements and all possible ways of link failures: applying the internal BGP function to all incoming announcements produces an outgoing set of announcements that entail the requirements.
We have to show that a network consisting of correct routers implies a correct network.

Base case (-1) In a network with one router, all requirements would have path size 0 and therefore there are no possible requirements which means the network is correct by definition.

Base case (0) Take a network with two routers. there are four possible requirement combinations: none, a-b simple, b-a simple and both. Since the routers are correct they only announce the path to them if the corresponding requirement exists. If the connecting link fails, there is no connectivity and all requirements are also met. The whole network is therefor correct.

Inductive step Now take a correct network of n routers. By adding a correct router R_new and connecting it to an arbitrary non-zero set of routers, the new valid requirement sets will be a super-set of the existing valid requirement sets. Since the routers are correct they will ignore the added topology for the set of requirements that was valid before the adding of R_new and will work as before, hence be correct.
All requirements that contain R_new in one of its paths can be rewritten as the requirements with the path containing R_new going up to R_new and an extra routing requirement that has the paths going from R_new or from the start if R_new is not in the path. For all those requirements whose paths end in R_new the advertisement reaches the start of the path since if it reaches the router before R_new because the network is correct. Since R_new is correct, it sends an announcement to its neighbor where it is not dropped. For the requirements with paths starting in R_new the advertisement reaches R_new since it reaches the neighboring router and from there is forwarded to R_new because it is correct.
For link failures we are only interested in links that were added with R_new (other link failures are covered by the correctness of the old network via the equivalency before). If the link fails there, the relevant path is obviously broken and not relevant for the requirement any more. The advertisement also doesn't reach the start of the path any more, but this is expected behavior.

Chapter 4

Implementation

This chapter looks at the practical side of implementing a system that checks if a network fulfills certain requirements. The overview in section 4.1 provides a general summary of the implementation and the technologies used. This is followed by section 4.2 which goes into more detail about how the forwarding state is extracted from the routers during simulation and how it is checked if a set of requirements are met on a given forwarding state.

4.1 Overview

The project was implemented in python in order to integrate well with NetComplete (also as at the time of writing NetComplete does not have a non-programmatic user interface). The actual implementation was done in about 1500 lines of code with another 800 lines of code for unit tests.
For the graph library, NetComplete's `NetworkGraph` was used which is a subclass of the `DiGraph` class from the NetworkX library. For the communication with the router the `pexpect` library was used.
The Cisco routers were emulated using `dynamips` which can run Cisco router images.

4.2 Instrumentation

Checking if a given setup runs correctly involves extracting a forwarding state from the emulated routers and making sure that all requirements hold. Then all possible link failure combinations are induced and for each one conformance to the requirements is checked again.
There are, however, some subtleties that have to be taken into account. First of all, we treat BGP requirements as strict, meaning that any prefix which is advertised due to a BGP requirement is not allowed to be advertised by BGP on routes which are not covered by requirements with that destination. For prefixes that are not a destination of any requirement, advertising them via BGP emits a warning. This is supposed to encode the idea that for BGP it is undesirable if prefixes are advised without the knowledge of the AS owner, because in practice this constitutes a BGP-Hijack. But we can't treat this case as an error since the given configuration sketch might already contain unrelated parts that advertise different prefixes. This means the network operator is not forced to write requirements for the whole existing configuration but can do it gradually.
In order to extract the forwarding state the output from the Cisco management consoles has to be parsed for all involved routers. In order to extract all the relevant information the commands `show ip cef` and `show ip route` were used. While `show ip cef` provides a nice machine-readable representation of the forwarding state, it does not contain extra information like via which protocol the information was learned. To get this auxiliary information the `show ip route` command was used and the information was combined.
From a practical standpoint the Cisco management console complicates some tasks by for instance printing all previous interactions with the console at every login (which posed the problem of reading non-fresh information) or by outputting some non-printable characters when paging through a list, but this was not a major hurdle.

The other part of the instrumentation was concerned with taking down and restoring links. To have a correct timing, first the forwarding tables were read. Then the link failure for the next to be checked state was introduced (restoring the taken down link from the current session) and then the conformance check was done. To ensure sufficient time between checks so that the link failures have enough time to propagate, the time of the start of the operation is recorded and after the check the program waits until enough time has passed. In practice the time we need to wait between failures is quite long (currently 1.5 minutes) so the program always has to wait. Because we assume the time to read out the forwarding states is more or less constant, we have a constant time between the link failure and reading the states for the checks.
The part that checks if the forwarding is correct does two things. For one it traverses the routers one by one to see if it can get from the source to the destination of the path by following the prefix. Note that the advertised prefix has to be larger or equal to the prefix that is searched for in order for the checker to be satisfied. In addition to that it splits the prefixes of the BGP advertisements it sees at each router on the path into the destination network and the rest. The destination network is marked as belonging to a requirement. This allows the tool to handle subnet aggregation. If there are any non-marked advertisements left, those will be reported as warnings or errors according to the policy mentioned above.

Chapter 5

Results

In this chapter Grigori is evaluated based on its checking results. The first part concerns the general results in section 5.1 of applying the checking function to what NetComplete made from our generated input. From a practical standpoint, section 5.2 where the found bugs are discussed, is surely important. Finally, no evaluation would be complete without talking about performance, which is what section 5.3 is about.

5.1 Coverage

All of the tests were run with link fail depth 1. This means for testing as described in the instrumentation section at first everything was run without failures and then each link was taken down individually but never more than one at a time. For correctness guarantees the link failure depth would have to be increased (for a more detailed discussion see section 6.2.1).

5.1.1 OSPF

Apart from the issue with ECMP, no other problems were detected with OSPF. All of the smart OSPF tests were run without a failure. Using the tool to generate random OSPF requirements I generated 33 configurations (this means that 7 cases which tested ECMP failed to synthesize). The configurations are for two topologies, the first of which is very small with 7 routers. The second which is slightly larger at 14 routers. Using any topology larger than that would have not been feasible, already a topology with 34 router would have taken hours per configuration for the non-ECMP requirements.
All of the 33 test cases succeeded. This does not guarantee that everything is correct and leaves the possibility that we didn't look hard enough or in the right places i.e. the generated tests don't exercise the part the contains the bugs. But the fact that only 1 OSPF related bug was found with two independent methods of generating tests suggests that there is a chance of having a correct OSPF implementation.

5.1.2 BGP

Testing BGP was a way more bumpy road than OSPF. In the beginning it was not clear how to correctly set up the sketch and what NetComplete needed versus what it already provided. Also there were instances where it would not complain when the configuration was slightly off.
After a lot of trial and error (and some bug fixing) we managed to get all smart BGP test cases to succeed. As of writing, MED selection is not implemented yet and there is a bug where the synthesizer won't generate the correct OSPF requirements. Because of that the test cases which cover the MED stage are not correct and the test cases that depend on ECMP OSPF requirements being synthesized fail to generate configurations. Otherwise all the test cases that are expected to produce a configuration, produce one.
With the random BGP routing requirements we trade some security that it will synthesize for a bit more coverage via the randomized route maps. Since the randomization is relatively crude,

this means that we waste a lot of effort trying to synthesize configurations that can never satisfy the requirements due to the structure of the sketch.
When generating the random requirements, in expectation (for the non-trivial topologies) about two out of ten configurations could be synthesized for a subset of the requirements. So after some synthesizing of random requirements for different topologies we ended up with 18 configurations to test, 10 of which were for a very simple topology where we might find fewer bugs, but the synthesis takes less time. The other topologies were the same as used for random OSPF requirement synthesis.
From those 18 tested eBGP configurations, sadly, none of them passed the test. Most likely the tool is still missing some checks to ensure the requirements can be satisfied, and most of the failures are probably due to NetComplete not complaining enough (getting the input sketch for BGP right is surprisingly tricky, as experience with writing the smart requirement generation has shown). It is an ongoing work to debug the reasons for failure and we expect to find at least one or two actual synthesis bugs this way.

5.2 Found bugs

Apart from some trivial bugs and regressions a total number of 15 bugs where found in the implementation. Some bugs were fixed as of writing, other bugs are currently being investigated. Here are the bugs that were fixed roughly sorted by time of discovery:

1. The ECMP settings were left at the default. In Cisco this means that a maximum of four parallel ECMP paths are allowed [1]. This hard limit was never enforced by NetComplete. As a temporary fix the limit of ECMP paths was increased to the maximum (32) and it was made sure that the random requirements would not have more than 32 paths. The complete fix would be to encode that restriction in SMT and setting the largest needed value in the configuration.

2. There was a bug in a preliminary check before BGP synthesis where it crashed because an assumption was baked in about the content of a list.

3. A bug was that when calculating the OSPF graph, it would include all routers (even ones in other ASs). This was fixed by only considering routers where OSPF was enabled.

4. Another bug was that the ordering of the requirement was not preserved before generating the SMT for the select function. This could mean that it would sometimes synthesize the constraints that lead to the exact opposite order.

5. When propagating announcements, the original AS path was replaced by an AS path containing only the originating AS, destroying a potential initial AS path length (by replacing [10, 10, 10] with [10]) and the initial hops.

6. The most important bug found was a bug in the actual model code. The problem was that the SMT written in a way that the tool was wishfully looking for **any** match in the select stages instead of stopping at the first point where the preferred path lost a comparison to the less preferable path and return unsat. This bug has been fixed.

7. Before the fix, the IGP cost was not correctly calculated and there was no way to get the OSPF requirements that had to be fed to the OSPF synthesizer in order to get the desired result (see select function stage 5).

8. In some cases eBGP routes were not preferred over iBGP routes.

9. During OSPF requirement generation a bug was uncovered where NetComplete crashed if the paths list of an ECMP requirement only contained one simple routing requirement.

10. There was a bug where some partially evaluated state was not discarded between calls to synthesize so calling it twice for the same graph sometimes resulted in synthesis failures.

11. Network strings where not correctly handled in prefix lists leading to extra prefix lengths being appended in the configuration.

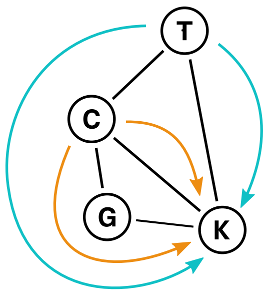


Figure 5.1: A graph in which one any-path requirement (turquoise) forces the other any-path routing requirement (orange) to act as an ordered routing requirement

The following bugs were reported and but not yet fixed so they might contain duplicates, false negatives, or collection of multiple bugs.

12. The preliminary check turned out to be too strict for some cases. This was in the case where two requirements seemed to conflict but didn't. Take for instance the example in figure 5.1, from the requirements it seems that the path T $\rightarrow$ C $\rightarrow$ K would be possible, but it should not be taken since BGP requirements are strict. To be satisfiable this forces the second any-path requirement (orange) to become an ordered requirement where the path C $\rightarrow$ G $\rightarrow$ K should be preferred over C $\rightarrow$ K. The check failed every time something like this was encountered.

13. A bug was that it NetComplete does not provide a warning if the given configuration is not synthesizable without route maps and no route maps are given. This bug might be the consequence of another bug that is reported here.

14. When getting the needed OSPF requirements from the BGP synthesizer it won't distinguish the case where the OSPF requirements are needed for tie breaking (emitting an ordered routing requirement) or simply to make sure that the select function is triggered below the OSPF stage (emitting an ECMP routing requirement) and always returns ordered routing requirements.

15. All the tested random BGP requirements that produced a configuration didn't verify. This problem is most likely a combination of providing wrong input to NetComplete, missing sanitization in NetComplete and bugs in the synthesis. But because we don't know the exact reason yet, these failures are listed collectively as a bug.

5.3 Performance

Since the project consists of two separate components there are many distinct possible performance bottlenecks. For checking small graphs (< 20 routers), which should be enough to check NetComplete exhaustively the performance is good enough.
The non-random tests have topologies with few routers and links and can therefore be checked efficiently. The configuration synthesis is also quite fast for these cases.
Surprisingly, for larger problems the times for both the synthesis (of BGP as well as OSPF configurations) and checking increases sharply, but due to different reasons. The problem here is that the time to actually generate the configuration is not negligible, which contradicts our assumption from the beginning. As a consequence, with this extra knowledge we would have probably made different design decisions (for details see section 6.2.2).

5.3.1 Generating the testing configurations

Generating the configurations for testing takes a significant part of the total amount of time to check a configuration.
For one, NetComplete takes quite some time to generate OSPF configurations with a lot of requirements. As an example, generating one configuration for the graph *Geant2009* (about 30 nodes) with a lot of requirements takes several hours. This could surely be optimized as for the

smaller examples it can be seen in the logs that the time spent in z3 is on the order of milliseconds while the time spent in processing the requirements is on the order of seconds.
The problem is that the synthesis time depends strongly on the number of requirements that are given and the random requirement generation process is geared towards generating as many requirements as possible for the given topology. How the synthesis time of NetComplete depends on the amount of given requirements is something that would have to be determined.
For BGP random requirements, the picture is similar even though configuration synthesis of BGP is slightly faster then the OSPF counterpart. What makes the BGP synthesis perform so badly in our case is the fact that there are many retries and also a lot of wasted effort on non-satisfiable inputs. The way the retries happen is that if the synthesis fails for the given set of requirements it roughly halves the size of input requirements leading to a maximum of $\log r$ tries per attempted configuration synthesis, where r is the amount of requirements initially given as the input to NetComplete. Further, due to the extra OSPF synthesis step sometimes needed this number doubles: every time the configuration can't be synthesized with the *enable_igp* flag turned off, it retries with the flag turned on.
It is important to note that generating the requirements themselves (the part that Grigori does) takes only a fraction of a second even for larger graphs.

5.3.2 Simulating the Network and checking the requirements

Because extracting the forwarding state from all routers takes relatively little time compared to the time the network takes to converge in the following discussion the time for checking a network under a given set of link failures is constant (the concrete value is 1.5 minutes). Since the link fail depth is 1 this means that checking a configuration currently takes $n \cdot 1.5$ minutes plus some setup cost for the simulation. Starting the simulation itself takes about 15 seconds per router (as a new router is started every 15 seconds) and then the network needs time to converge initially (the concrete value the simulation waits is 2 minutes). To test the network with link fail depth n takes $O(2^n)$ time and since most used topologies are not trees, n is usually a lot larger than the amount of nodes. As an example, a very small graph with 7 nodes that was used for some of the tests has 13 edges which means testing one configuration on that graph with link-fail depth takes $14 \cdot 1.5 + 7 \cdot 0.25 + 2$ (which is about 25) minutes. For link fail depth $n = 13$ there are about 8000 failure states that need to be ckecked, so if checking each failure takes 1.5 minutes this means the simulation would have to run for 200 hours which is more than a week. For larger graphs this number grows accordingly. Of course this testing can be parallelized, but it has to be taken into account that only one instance of `dynamips` will ever run at a time, so one cannot trivially exploit thread level parallelism for the simulation. One would have to isolate the different `dynamips` instances using either VMs or containers.
Tables 5.1 and 5.2 show some statistics on the actual computation time of checking a link failure. While the time measurement are coarse (seconds) and they aggregate the times for both reading the forwarding state and checking the requirement conformance, they still show the big picture.
First and foremost they show that the checking times are dominated by connecting to the routers and reading the tables. This is visible by the fact that there is no significant time difference in checking a single ECMP requirement (in the test cases > 9) compared to checking the non-ECMP test cases which contain a lot more requirements (the first Compuserve test case has 22 requirements).
The tables also show that the time to check a failure is proportional to the amount of routers in the graph, with about 1 second per router. This seems like a large number but there is no reason to be concerned as we would expect that the 1.5 minute mark is surpassed at about 70 routers which is currently a lot more than Grigori can handle. Also the growth is linear while, as seen, most other times grow much faster with graph size so in practice this should never pose a problem.
The python script to read the values from the saved output of running `network.py` is provided in the Grigori source tree. Note that the script subtracts 5 seconds from the measurement for every error that happens on connection establishment. This is because sometimes the connection to the router fails and the retry timeout in `pexpect` is set to 5 seconds. In the processed values the standard deviation is roughly the measurement precision, which makes sense. The reason the connection fails is because sometimes the router spits out a status update during

Test	Median	5th Percentile	95th Percentile
0	11	9	11
1	10	9	11.5
2	10	9.5	11.5
3	11	10	11
4	11	9.5	11.5
5	11	9.5	11.5
6	10	9.5	11
7	11	9	11.5
8	11	9	11.5
9	10	9.5	11.5
10	11	9.5	11.5
11	11	9.5	11.5
12	11	9.5	11.5
13	11	9.5	11.5
14	11	10	11.5
15	10	9.5	12
16	10	10	11.5
17	10	10	11.5
18	11	9.5	11.5
19	10	9.5	11.5

Table 5.1: Measurements of the time to check the forwarding state for the different configurations in the Heanet topology (`var/OSPF_Heanet(40454456)`_*Test*/)

Test	Median	5th Percentile	95th Percentile
0	17	15.8	19.4
1	17	16	19.2
2	17	16	19.4
3	17	15	19.6
4	18	16	19.4
5	17	16	18.2
6	18	16.8	19.4
7	18	16.8	19.2
8	18	16	20
9	18	16	19.2
17	18	17	19.4
19	17	17	19.8

Table 5.2: Measurements of the time to check the forwarding state for the different configurations in the Compuserve topology (`var/OSPF_Compuserve(3080100503)`_*Test*/)

the time when we connect which causes `pexpect` to see an unexpected output.
Due to the fact that it is unfeasible to synthesize configurations for large topologies (see preceding section) the effect of increasing topologies on runtime of the requirement checking could not be determined precisely here. A rough estimate is that the total complexity should be in the order of $O((n + m) \cdot 2^n)$ where m is the amount of nodes and n the amount of edges in the network.

Chapter 6

Further areas for improvement

Since the scope of this project is quite limited, there are a lot of things that are left open for future work and investigation. In the following sections the most important places for improvement are grouped together by the part of Grigori they are in and highlighted in a few paragraphs each.

6.1 Instrumentation Improvements

6.1.1 Testing a configuration with up to n simultaneous link failures

Right now, while taking down links in the simulation, only one link is taken down at a time. For the result of the test to be correct in the sense of “for all link failures” also combinations of link failures have to be taken into account. This means that all 2^n link failure combinations have to be tested instead of just the $n + 1$ that are currently tested.

Alternatively, instead of inducing up to n link failure where n is the amount of edges, when testing requirements with a maximum of m paths, one could induce 2^m link failures and have a correct result (under some reasonable assumptions).

Right now the code is written in a way that takes a list of link failures so it would be relatively easy to extend it to fill that list with more than one link failure at a time. Note that this would add to the testing time which was one reason why it was not implemented.

6.1.2 Test conformance to a sketch

Making sure that the resulting sketch corresponds to the original sketch modulo the symbolic parts is another low-hanging fruit in this area. For this to work one would have to compare the internal representation of the configuration from before and afterwards. Right now this test would fail because for some things extra parts are added to the configuration like export route maps (which also makes sense). Maybe that could be handled in a way that adding things to the sketch could produce a warning and removing things from the sketch would produce an error.

6.1.3 Reusing the instrumentation code

The instrumentation part that checks correctness of a configuration is NetComplete-agnostic and can be used to check configurations generated by any Network Synthesis tool or even manually generated ones, given routing requirements are provided in the NetComplete format.

It would be imaginable to integrate this component into other tools that need to ensure a correct network forwarding state and where the user can provide it with a set of routing requirements.

Right now the error reporting of the checker could also be made more useful by automatically getting the relevant information from the router (for instance exacting BGP session information) and pinpointing the the error more closely by traversing the path in reverse order and showing the exact location of the announcement drop.

6.2 OSPF checking improvements

6.2.1 Finding out how to expand the model coverage for OSPF

Right now it is a open question how one could best improve on the current OSPF random requirement generation. There are multiple shortcomings that would need to be fixed to make it practical, the most pressing of which are:

- ECMP would have to be mixable with the other requirements
- The directed graph property would have to be taken into account. Requirements don't have to be bidirectional.
- Currently ordered and any-path routing requirements never have more than two paths. It should be possible to combine multiple of the current requirements that start at the same node into equivalent requirements with more than two paths (or possibly choose whether they should be combined or not)

It is possible that most of these objectives could be met by having a function that can quickly check if adding a certain constraint makes the configuration unsat. This could then be used to randomly add (directed) requirements with decreasing probability (sample the amount of requirements with a geometric distribution). If adding a requirement makes the requirements unsat, backtrack and keep going. One could also randomly pre-populate some edges with weights.

6.2.2 Determining the actual cost of testing for a given input

As seen, the implicit assumption that the time NetComplete takes to synthesize a configuration is negligible, is not correct. To make the best possible design decision is has to be understood exactly how the checking cost behaves with variations in input. Currently only small graphs are used (with amounts of requirements that are dependent on the size of the graph).
All we know is that increasing graph size is very bad for performance as the synthesis time of NetComplete grows a lot. It is not clear if this is due to the amount of requirements passed in, or if the synthesis time was a lot smaller how much time the checking component of Grigori would take. All we know is that using a lot of requirements saves us $2^{|\text{reqs}|}$ configuration generations, that the synthesis time increases with the amount of requirements and that the checking time increases exponentially with the amount of edges.
By knowing which term explodes first, and why, one might be able to tune the requirement generation to be able to test larger topologies (or test the current topologies in less time).
It is left open for further research to systematically gather data on how much time the combined system of NetComplete and Grigori (the checking component) takes as a function of the input requirements and the graph.

6.2.3 Testing negative examples in the smart requirement generation part

To make sure that OSPF works correctly one would also have to show that for impossible requirements NetComplete returns unsat. The smart way of generating requirements could be extended that (where applicable) for every existing test case there is a test case where the graph is populated with edge weights that make synthesis impossible.

6.3 eBGP checking improvements

6.3.1 Finding out whether BGP can implement more requirements that OSPF

While it is clear that BGP is more powerful than OSPF, it is not immediately clear that there are requirements that can be implemented using BGP but not using OSPF. The thing is that BGP is very good at restricting access (dropping advertisements) but might not be better at providing connectivity. My current experience would let me venture a guess that they can actually implement the same requirements (apart from ECMP), but that would have to be proven.

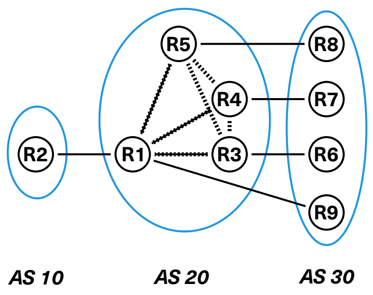


Figure 6.1: A graph in which all stages of the select function can be tested. This could significantly reduce the implementation complexity of `bgp_ver.py`.

6.3.2 A better design of the smart BGP requirement generation

The current design of the functions that generate the smart requirements is very poor. The cyclomatic complexity is through the roof and debugging it is a pain. The main problem stems from having different topologies for each configuration and handing them all in one function.
A better design would be to have one graph where there are four paths from 'R1' to the AS where the destination is advertised from (see figure 6.1). One of them would be a direct connection while the other ones would be via iBGP neighbors. Then one could have one function per select function stage that either sets the comparison to "greater than", "less than" or "equal to" for that stage and then those functions could be composed in a reusable way.
This would also allow to generate testing configurations for arbitrary configuration vectors (in which at each index the result of the comparison function at the corresponding stage is given) as described in section 3.2.2.

6.3.3 Any-Path requirements as smart BGP requirements

While the any-path requirements are not restrictive enough to actually test the stages of the select function, there should still be a test case where symbolic values are concretized by having any-path requirements. While no matter which path is selected, the requirement will always be satisfied, it is still important to test that everything works with any-path requirements.

6.3.4 Adding OSPF to BGP checks

Right now to verify the BGP implementation, OSPF is only involved as a side product of stage 5 of the select function. Having independent OSPF requirements mixed with the BGP requirements would be a good thing to do to test the overall system. Of course this adds extra dimensions to the input ...

Chapter 7

Conclusion

In conclusion, Grigori was useful in its purpose of finding bugs in the synthesis of NetComplete. While it proved to be harder than expected to synthesize good random requirements for both ECMP and BGP, a valid point in the design space was explored by balancing coverage and synthesis success probability.
The satisfying part about the testing result is that a number of medium to high profile bugs were found and fixed. Another aspect of the evaluation showed that the current performance it not great, but when taking the synthesis speed of NetComplete into account, the overall overhead of the checking is acceptable. Also the fact that huge topologies cannot be checked efficiently by Grigori is not a showstopper because the purpose of it is to debug NetComplete, not a specific network configuration.
Looking forward, a few interesting possibilities to expand the project either in in scope or improve it in quality open up. One example would be to test the combination of BGP and OSPF with random requirement or to use the checking part of Grigori in another context and maybe improve its usefulness when it comes to helping a human debug the situation.

7.1 Acknowledgments

The author would like to thank the supervisor Ahmed El-Hassany for his tireless efforts in fixing bugs and for the continued support when dealing with NetComplete's BGP synthesis. He would also like express his gratitude to the Tutor Prof. Dr. Laurent Vanbever for taking time out of his busy schedule to regularly review the progress of the project and providing valuable guidance.

Bibliography

[1] *MPLS: Layer 3 VPNs Configuration Guide, Cisco IOS XE Release 3S (Cisco ASR 900 Series)*, chapter ECMP Load Balancing. Cisco.

[2] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[3] J. Moy. OSPF Version 2. RFC 2328, RFC Editor, April 1998.

[4] H. W. Vildhøj and D. K. Wind. Supplementary notes for graph theory i. 01 2012.