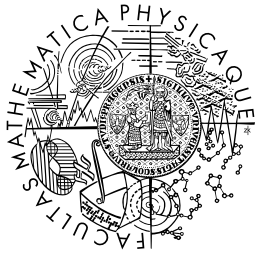


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Pavel Turinský

Visualizing OSPF topology

Department of Applied Mathematics

Supervisor of the bachelor thesis: Mgr. Martin Mareš, Ph.D.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

First, I would like to thank my supervisor Martin Mareš, for being hopeful and patient and for teaching me many useful skills.

For similar reasons, I am grateful for all the presentation and explanation skills the M&M correspondence seminar has taught me.

This thesis would not also be done without my friends, colleagues and family for nudging and convincing me enough to finish my studies.

Last, but not least, I want to thank You, my dear reader, because right now You are making sure this thesis was worth writing.

Title: Visualizing OSPF topology

Author: Pavel Turinský

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., Department of Applied Mathematics

Abstract: Network administrators need to be aware of the state of their network. While link-state routing protocols like OSPF have the required information, there was no easy way to present it to the administrator. To solve this problem, we developed Birdvisu, a program to visualise the topology as seen by the BIRD Internet Routing Daemon. Our approach allows the topology to be easily annotated with other data, providing an extensible way of analysing the topology. While not fully featured yet, Birdvisu helped discover several misconfigurations in a community-run network connecting about 1600 people.

Keywords: OSPF, network visualisation, routing, BIRD

Contents

Introduction

Network administrators need to be aware of the current state of their networks. While routing protocols like OSPF often need to have complete information about the state of the networks, we did not find an existing project that would visualise this data easily. We have therefore created a rather simple program called Birdvisu, which aims to fill this hole.

Apart from visualising the topology, we provide the user with basic analysis tools like showing the shortest path DAGs for routers. Furthermore, the architecture of Birdvisu will allow enhancing the data from the routing protocol with additional data, e.g. the utilization of particular links.

Terminology

In this thesis, we use the word *host* to mean a device connected to a network and capable of processing (sending, receiving, forwarding, ...) IP packets.

An *administrator* is a person or a group of people, who provide the routers with configuration and who make sure that the routers (and other infrastructure) functions correctly.

The word *network*, when used as a noun, will always denote a set of hosts, that can exchange packets "directly", without forwarding, under normal circumstances. While this is somewhat synonymous with the term *network segment*, when the network splits, there may be multiple link-layer segments belonging to the same network.

We say that a router is said to be *incident* to a network when it has an interface into it. For simplicity, we also say that that network is incident to that router (that is, the incidence relation is symmetrical). We say that a router is a *neighbour* of another router, when they are incident to a shared network, and conversely, that two networks are neighbours when they share a router.

We will use the word *network system* (or just *system*) to describe a set of networks managed by a single administrator, which is intended to forward packets across it. This can mean the whole autonomous system[1], but often this is a much smaller part. When we speak about routing using OSPF, a system is only the set of networks that run a single instance of the OSPF protocol.

By the term *IP* we mean the Internet Protocol of either version 4 or 6. When it is important to distinguish, we explicitly write *IPv4* or *IPv6*. IP may also denote an IP address, as in "a router has an IP", but this usage should be clear from the context.

We distinguish between *routing* as the act of finding next hops, and *forwarding*, the act of sending a received packet through an interface according to a *routing table*. (Whether the routing table is distinct from the network-layer forwarding table is an implementation detail.)

A *topology* describes the connections in the network system, i.e. which router is connected to which networks, with what costs etc. We use the term *graph* only when talking about visualisation of the network to distinguish between the real

[1] This is where we borrowed the word "system" from.

state and a virtual one. Moreover, in section 2.5 we will see that a topology is in fact not a graph (as understood by graph theory).

We will be using many other networking terms, which we hope are well-known, but for the sake of completeness, we provide a glossary and a list of abbreviations at the end of this thesis.

Notation

We write filenames and commands in monospace font, like in `rm -rf /bin`. Snippets of code are also written in monospace, but when we speak of a particular class in text, it is only capitalised, as in "The keys of the dictionary are VertexIDs".

For network diagrams, we use the Network topology icons from Cisco [1]. The basic icons are shown on Figure 1.

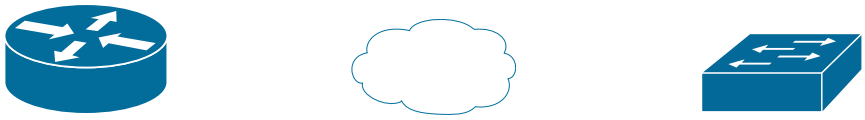


Figure 1: Basic Cisco network icons

Structure of the thesis

In the first chapter, we explore various approaches of visualising and monitoring a network system. Then, in chapter 2, we understand the behaviour of relevant networking technologies. From that, we derive a design for Birdvisu in chapter 3. Chapter 4 explains the usage of the program. At the end, we discuss how the project copes with topologies of network systems of various sizes.

1. Motivation

In this chapter we explore existing solutions for visualising and monitoring network systems and describe the requirements of network administrators. From this we derive a set of properties Birdvisu should fulfil.

1.1 Existing approaches to network monitoring

Several approaches to network monitoring and status visualisation already exist. These can be approximately split into several types: visualisation of existing data, traffic visualisation, host monitoring systems and integrated system management platforms. Here we introduce them shortly and explain their potential disadvantages, compared to visualisation of routing information.

1.1.1 Visualisation of existing data

Tools in this category do not collect any data on their own, rather only focusing on visualising data from other tools. These projects are often small, can be easy to run and allow visualisation of various data, but they require other infrastructure to provide the data. For example, the Network Weathermap [2] project can be used to provide overview based on data in Cacti or other tools. Grafana [3] provides many different methods of visualising data from various databases.

This does not seem to be usable for visualising OSPF state, because we are not aware of any collectors of OSPF state data.

1.1.2 Traffic visualisation

Various software packages can collect and graph utilisation of network links. This often provides information about link state, but requires collection of data on all hosts. Also, in some cases, this might not provide accurate data of the system state, as we will see in section 2.5.1.

These projects usually store a time series of utilisation and therefore need to be deployed on some central server as long-running services.

Examples of this approach include Cacti [4] and Munin [5]. Although both mentioned projects can graph other data, plugins for data collection are available, so this differentiates them from the previous group.

1.1.3 Host monitoring systems

Projects in this category do not necessarily consider the whole network system, but check state of individual hosts. It is possible for them to run locally, as is the case for Plotnetcfg [6], or check the host over the network (CaLStats [7], Icinga [8]), but they do not provide overall picture. The capabilities of these tools also differ, from just checking reachability (CaLStats) to being able to retrieve various details from the hosts (Icinga, plotnetcfg).

This approach can also miss some issues with the system, as described in the section 2.5.1.

1.1.4 Integrated system management platforms

Some network administrators have created platforms for managing the hosts across the network system. These systems usually know various aspects of the system and may provide features like configuration generation or topology visualisation [9]. However, many of these are tailored for the specific system and are therefore not reusable in other environments.

Also, since large number of people need to access such platforms, they are usually server based with web interface [9].

1.1.5 Topolograph

We are aware of only one project that would allow visualisation of OSPF topology, Topolograph [10]. While it does not collect its own data, its companion project, Ospfwatcher [11], is able to retrieve current topology data from a Quagga [12] routing daemon. The deployment involves setting up Logstash [13], so this is only suitable to be run on a server.

Also, we find the interface of Topolograph impractical to use (tested in Firefox version 116.0b2).

1.1.6 Summary of existing tools

As we have seen, various projects are available, but they often have some important disadvantages. Table 1.1 summarizes known approaches.

Approach	CD	RS	CoH	EoD	T
Only visualisation	×	?	×	?	×
Traffic visualisation	✓	✓	✓	×	×
Host monitoring	✓	?	✓	?	×
Integrated management	✓	✓	?	×	✓
Topolograph + Ospfwatcher	✓	✓	×	×	✓
Ideal state	✓	×	×	✓	✓

Table 1.1: Comparison of current approaches. CD: Can collect data on its own, RS: Requires to be deployed on a server, CoH: Collects data on individual hosts, EoD: Easy to deploy, T: Understands the topology of the system.

1.2 Target group of users

Birdvisu aims to be a rather simple tool for small to medium-sized network systems, where it is impractical to deploy complex monitoring and visualisation infrastructure. We want to especially help in homelabs and community wireless networks, but any OSPF deployment should be able to make use of our project.

Also, since we do not require any server apart from a routing daemon, which can be running on any laptop, Birdvisu might also be a helpful tool for admins in the field trying to fix broken infrastructure.

The primary motivation for implementing Birdvisu was the need of the author, who has a dynamically routed system that switches uplinks depending on which of them currently work.

1.3 Goals of Birdvisu

We want Birdvisu to:

- Be a simple tool, easy to run on a regular laptop,
- Leverage the knowledge of topology, which is common in the system, without needing to collect data from other hosts,
- Help administrator quickly recognise issues in the network,
- Allow to be enhanced by other data and to export own data to other projects, in the future.

2. Analysis

In order to avoid as many problems as possible when visualising and analysing an OSPF topology, we need to understand several aspects of networking, the OSPF protocol and its implementation in BIRD. However, the aim of this thesis is not to be a complete guide to OSPF nor BIRD, therefore, for the sake of brevity, we skip many details which do not influence our project.

To aid in testing Birdvisu and experimenting with routing, we will also present a small side-project called Gennet in this chapter. Using it, we will understand the behaviour of network splits and multiple links, as seen by the BIRD routing daemon.

2.1 OSPF overview

OSPF [14, 15] is a link-state routing protocol, which means that routers try to understand the whole topology of the network and find the best path using an algorithm for finding the shortest paths. Usually, Dijkstra's algorithm is used. OSPF was designed to provide dynamic routing in an entire autonomous system, but running it on a much smaller scale is also possible.

OSPF requires routers to share information about the state of the topology in messages called *Link State Advertisements* (or LSAs for short). The LSAs contain information about which network segments are incident to which router on which interfaces and with which routers it is possible to communicate on those interfaces. To distinguish between routers, each router is assigned a 32-bit number called *Router ID* by the network administrator. Router IDs are usually written in a quad-dotted notation.

The LSAs are flooded throughout the system in order to let all the routers know about the current topology. To avoid overloading the system just with this routing traffic, OSPF devises several mechanisms of minimising the number of exchanged messages. First, each network elects a *designated router* (or DR) which coordinates exchange of the messages in that network segment. Second, the system can be partitioned into *areas*. That way, the frequent LSAs are only flooded throughout a limited set of networks; the *area border routers* (ABRs) send accumulated LSAs into other areas.[1] These LSAs do not describe the topology of the originating area.

The area 0.0.0.0 is called the *backbone area* and all other areas must be incident to it, so that it has all the routing information. When this is impractical, OSPF allows two routers to be connected using a *virtual link*. From the routing perspective, this is a point-to-point connection between the routers in the backbone area, which allows to forward LSAs through other areas even when these LSAs would not normally leave those areas.

There are several types of networks that emerge in OSPF topologies. The *transit* networks are used for forwarding packets in an area. *Stub* networks only have one router and therefore can only deliver packets originating from or destinated to that network. For representing routes outside of the area and the

[1] These inter-area LSAs are called "Summary LSAs" in OSPFv2, but there is no requirement that the routing information is actually aggregated.

whole system, OSPF recognises *extra-area* and *external* networks respectively. It is also possible for a router to be adjacent to an extra-area router through a point-to-point link. Figure. 2.1 shows the same classification visually.

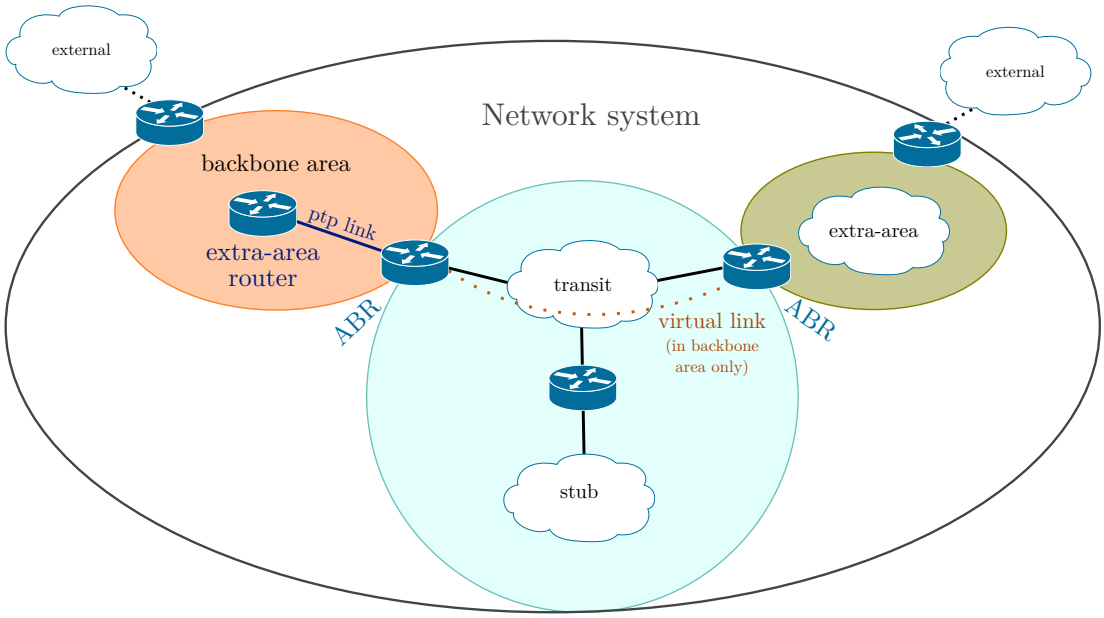


Figure 2.1: Types of networks as seen from the cyan area.

Once the router has complete information about the topology of an area, it constructs a graph representation of the network and calculates the shortest path DAG[2] rooted at that router. This DAG is then used for finding shortest paths to routers and networks in that area, including the external, extra-area and stub networks adjacent to that area. OSPF specifies that the graph has all the networks and routers as vertices, directed edges lead from each router to the incident network with the configured cost and from each transit network to incident routers with cost 0 (except when the two-part metric [16] is implemented). There are no edges starting at the external, extra-area or stub networks, so that the shortest path DAG calculation does not find paths through them.

The cost of the edge to an external network can be of two types. A type 1 cost uses the same units as the internal costs, whereas any type 2 cost is larger than all internal or type 1 costs.

The OSPF family of routing protocols has undergone long evolution since the first specification in 1989 [17]. There are currently two versions of the protocol in use – versions 2 and 3. While the basic idea is still the same, OSPFv2 can only handle IPv4 systems. Although OSPFv3 claims to be network-protocol-independent, it is usually only used with IPv6 systems and, in fact, features like virtual links can only be used with that network protocol [18].

Both OSPF versions have numerous extensions, as can be seen by the number of RFCs that update the base specifications [14, 15]. Therefore, we do not implement the protocol ourself, but rather find a suitable routing daemon to determine the current topology for us.

[2]In the specifications, it is called the shortest path *tree*, but there may be multiple shortest paths and the router is supposed to use all of them.

Many improvements of the protocol only affect the topology construction (e.g. NSSA areas [19]) or change the data exchange between routers (multi-instance extensions [20], authentication [21], ...). By extracting the topology from a routing daemon, we can support many OSPF extensions for free. For this reason, it is mostly sufficient to only consider the base specifications of OSPF.

2.2 Routing daemon selection

While we were mostly determined to use BIRD [22] from the start, because we already had some experience with it, let us present here a short summary of other possibilities. Note that the particular choice does not affect interoperability with other routers as long as the chosen routing daemon supports extensions used in the network system.

There are several implementations of both versions of the OSPF protocol. However, many of them are tied to the specific router hardware, which makes them impractical to connect to a graphical visualisation. Moreover, even evaluating the feasibility would require us to obtain the specific hardware. Therefore, we only consider hardware-independent solutions.

While we are aware of several software implementations, many of these do not seem to be developed anymore (Quagga [12], XORP [23], OpenOSPFd [24]). Apart from BIRD, we only found FRRouting [25] to be maintained, meaning that a new version has been released in the past year. While being maintained is not a strict requirement, it would allow us to use that implementation in case OSPF is extended again.

However, even BIRD does not implement all the extensions, for example, the two-part metric [16].

2.3 BIRD interface

The BIRD daemon is controlled through a UNIX domain socket using a text line-based protocol slightly resembling SMTP. The client may send commands to the daemon, which provides responses. The response may be long and possibly formatted into a table. This interface is primarily aimed at human users, so a rather simple client, `birdc`, is provided in the BIRD's package.

While there is a note of a machine-readable protocol in the `doc/roadmap.md` file in BIRD's source code [26], it has not been implemented, so we will need to interface using the socket. This has following consequences, most of which are not very pleasant:

- The responses to different commands often have completely different formats. This necessitates creating a dedicated parsing routines for each kind of command we want to use.
- There is no guarantee that the output will not change between versions. We might need to follow BIRD's development in order to be aware of possible changes.

- The output does not contain all details of BIRD's state. For example, we can not retrieve the shortest path DAG directly from BIRD, nor see the details of the individual LSAs.
- There is no way to get notified when the topology changes.

BIRD provides only a few commands that deal with OSPF:

- `show ospf` shows a simple summary of the running instances of OSPF, like which areas are they incident to or how many LSAs does BIRD currently consider.
- `show ospf interface` describes the current status of the individual local interfaces: their configuration, designated routers for the incident network, etc.
- `show ospf neighbors` provides details about the state of communication with neighbour routers.
- `show ospf lsadb` returns the details about known LSAs. Unfortunately, this contains low-level information like checksums and sequence numbers, but not details about networks or routers.
- `show ospf state` shows an overall view of the OSPF graph representation: present routers and networks, costs of links, distances, ...
- `show ospf topology` seems to only provide a subset of the output of `show ospf state`. For example, it does not provide information about any non-transit networks.

Even though some of the commands can have more parameters, parsing the output of the `show ospf state` command is still the only the viable option of getting a topology description. The following subsection describes the syntax of the response to this command.

2.3.1 Retrieving the OSPF state

Let us look in depth at the `show ospf state` command, since we will be using it and the format of its output a lot.

The command has two optional parameters. First, the flag `all` may be added to show details not only about the reachable part of the system, but from all the known and non-expired LSAs. The difference between the topologies can be used to discover network problems even without configuring the expected state.

The second parameter is a name of the OSPF instance. It is only required when BIRD is running multiple instances simultaneously. This is unfortunately quite common, because in dual-stack systems there need to be two separate instances of OSPF, each configured for different IP version.

The output of the command is a tree of lines representing the topology itself. Children of a directive are indented by one more tab. An example output is shown in listing 2.1.

```
1
2  area 0.0.0.1
3
4          router 203.0.113.1
5                  distance 20
6                  network 203.0.113.0/26 metric 10
7                  xnetwork 203.0.113.64/26 metric 10
8                  xrouter 203.0.113.42 metric 10
9
10         router 201.0.113.2
11                 distance 0
12                 network 203.0.113.0/26 metric 20
13                 external 0.0.0.0/0 metric 60
14                 stubnet 201.0.113.128/25 metric 200
15
16         network 203.0.113.0/26
17                 dr 203.0.113.1
18                 distance 20
19                 router 201.0.113.1
20                 router 201.0.113.2
```

Listing 2.1: Example OSPFv2 state output

The tree as output by BIRD[3] has three levels, we call them top-level, level-2 and level-3. The top level only contains directives of the form `area AreaID`, with the AreaID being written in the quad-dotted notation.

On level-2 are mentioned all the routers and networks in the area. This is different for OSPFv2 and OSPFv3. While routers are always mentioned by their router IDs (again, quad-dotted), networks in OSPFv2 dumps are addressed using their IPv4 addresses (CIDR notation), but by the designated router ID and interface number in OSPFv3 ones: `network [203.0.113.1-23]`.

The third level describes details of the respective router/network and all the incident objects. There is always the distance from us (i.e. the router we asked for the dump), or the word `unreachable` if it is not reachable.

There is also a level-3 line for each incident host and network. Overview of the "tags" (the first words) and parameters is provided by table 2.1. Note that networks can only be incident to routers, while routers may be incident to anything. The incidences of routers have also a metric, after the word `metric`, or, in case of type 2 external cost, `metric2`.

For networks, additional details are also provided on level-3. For OSPFv2, the designated router ID is given in the `dr` directive, similarly, OSPFv3 may provide the networks with zero or more `address` lines with CIDR addresses. An example of a network block in OSPFv3 is in listing 2.2.

One of the nice properties of BIRD's output is that whenever there is a level-3 incidence line for object B in a level-2 block of object A, there exists an edge from A to B in the topology used by the Dijkstra's algorithm. This fact will later simplify parsing.

[3]The format was determined by experimentation and inspecting of `proto/ospf/ospf.c` in BIRD's source code [26].

Incidence type	tag	parameter
Transit network	`network`	Same as on level-2
Router	`router`	Router ID
Extra-area router	`xrouter`	Router ID
Extra-area network	`xnetwork`	IP address (CIDR)
External network	`external`	IP address (CIDR)
Stub network	`stubnet`	IP address (CIDR)
Virtual link	`vlink`	Peer router ID

Table 2.1: Level-3 lines describing incidences

```
network [198.51.100.1-16]
        distance 10
        router 198.51.100.1
        router 198.51.100.2
        address 2001:db8:b00:7::/64
```

Listing 2.2: Example OSPFv3 network block

2.4 Test network system: Gennet

To help test Birdvisu and understand network behaviour, we created a simple set of scripts called Gennet. Since it was mainly written to aid Birdvisu, we provide it as attachment A.2 of this thesis.

Gennet is a network generator. Using a hard-coded configuration and a set of Jinja2 [27] templates, it provides a semi-automatic way of creating several virtual machines (their disk images and startup scripts) and configuration to connect them using software bridges. This will allow changing the state from the host operating system, simulating various network conditions.

We fully admit that Gennet is really just a quick hack. However, since it was created specifically to aid the development of Birdvisu and because it provides a reproducible environment, we think it makes sense to attach it to this thesis. The particular choice of technologies (Jinja2, Python, Bash, QEMU and Alpine Linux) is driven solely by our previous experience with them and should not affect the behaviour of the generated system in any way.

Using Gennet generally involves creating a base disk image and configuration for individual machines, embedding this configuration into the base image, configuring the bridges and finally starting the required machines. The process is explained in detail in Gennet's `README.md` file.

When used without changing Gennet's configuration, it creates a topology of 10 routers (A–I, X) and 7 networks (numbered), as shown in figure 2.2. We expect the user to provide some network connectivity to network 7 and configure the machine X manually. We use this exact topology as a base for our experimentation.

The default Gennet assigns addresses as follows: The networks are given addresses 172.23.n.0/24 and fdce:73a4:b00:n::/64, where n is the number of the network. Routers are assigned router IDs of 172.23.100.r, where r is the lexicographic order of that router (A gets a 1, B is 2, ..., I becomes 9, X is 10). The IP addresses of the routers have the same number in the last octet (e.g., the IPv6

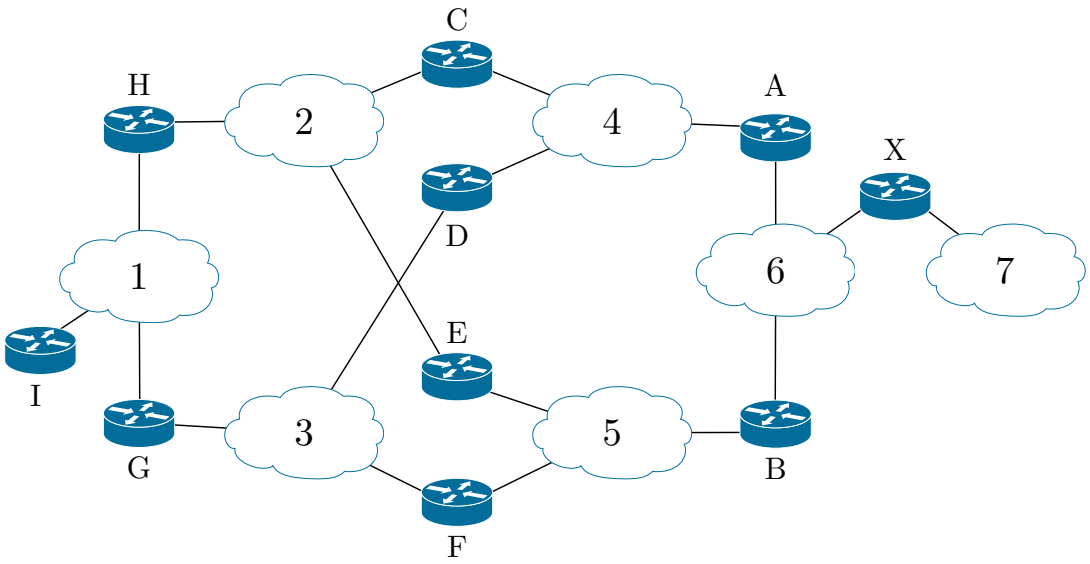


Figure 2.2: The topology of the default Gennet

of router X in network 6 is fdce:73a4:b00:6::a), and all routers have an IP address in each incident network. The costs of all links are 10 by default.

2.5 Unusual network states

Now that we have basic understanding of BIRD and a network system for testing, we see how both versions of OSPF react to various unusual conditions in the system.

We focus on behaviour of BIRD in following scenarios:

- Network split: Hosts in the same network stop being able to communicate with some other host in the same network. This is often caused by a broken cable or switch malfunction.
- Multiple links to the same network: It might make sense to connect a single router to the same network using multiple links, possibly with different costs. This provides redundancy and helps e.g. avoid network splits.
- Network with multiple addresses: Sometimes, a network may have more than one address (prefix). This may be either intentional or a result of accidental joining of networks which should be separate.

Unfortunately, the behaviour of BIRD in these states is often very different depending on the version of OSPF.

2.5.1 Network splits

Network splits are of particular interest to administrators. Not only are they symptoms of a broken part of the infrastructure, but also every netsplit inherently means that some addresses from the split network can not be reached by some hosts, because there is no hint which segment a packet should be delivered to.

Splits can also be tricky to spot from systems using topology-unaware approaches, because if a link connecting two switches fails, all ports on hosts are still up and the traffic in the split network might not change, when no traffic is routed through the broken link.

When a split occurs, at least one segment stops being able to communicate with the network's designated router. Either this segment has only one router and becomes stub, or has more routers and then a new designated router is elected. (OSPF cannot detect when a network splits and there is no router in a separated segment. Monitoring of host reachability is therefore still useful.)

The representation of split network in BIRD's output is straightforward: some routers may become attached to a `stubnet`, instead of a `network` and more level-2 network blocks can appear, one for each segment that has elected a DR.

For OSPFv3, this forms a valid topology, since the level-3 network directives are derived from the designated router ID and its interface number, identifying the network uniquely. However, since OSPFv2 identifies a network using the shared address in the output, it is not immediately obvious, which of the segments the router is connected to. Luckily, this can be deduced from level-2 network blocks, because they provide information both about the segment's DR and about incident routers.

Consequence: A network address is not sufficient to identify a network or stubnet. To do that, we either need to also know a designated router ID, in case of transit networks, or which router the network is connected to, for stubnets.

2.5.2 Multiple links

BIRD's implementation[4] of both versions of OSPF seems to announce two copies of the same network throughout the area, if the designated router is connected to that network using multiple links. This is not an issue for routing, because using any of the copies results in the packet being sent to the right network, but is an unfortunate behaviour for visualisation. In the OSPFv2 dumps there is no way to differentiate the two copies, since the DR's interface number is not exposed, so we can only merge them into one network, solving the problem.

On the other hand, in OSPFv3 the interface number may be the only information used to differentiate different networks, since the networks do not need to have any addresses assigned. The only safe way is therefore to visualise both copies just in case.

(There also seems to be a bug, when OSPFv3 dump does not contain the level-2 block of the multiply-connected router on a neighbouring router. We did not explain this behaviour, but it does not seem to propagate to other routers nor affect packet forwarding, so we decided to ignore this peculiarity and debug it later. A simple workaround is to add another router between the actual network and Birdvisu, which is always possible by using unnumbered networks.)

2.5.3 Multiple addresses in a single network

In OSPFv3, a single network can be assigned zero or more addresses. Therefore, from its point of view it is not an unusual state. *Consequence:* For OSPFv3, the

[4]We are not sure whether this is the correct behaviour

set of all addresses must be considered to determine whether the network has changed or not.

OSPFv2 treats each of the address as a separate network, ignoring other routers. When this is intended, it should not cause problems, and unintended merges of networks will not interact. However, we cannot detect this state across the area, unless there is another change in topology (for example, if this is caused by a cable connected to a wrong port, the network will probably be stub).

2.6 Area structure

It is also worth to consider the expected size and structure of an OSPF areas where Birdvisu might run. While it is up to the administrator and they may be very creative, there are some limits to such creativity.

The largest system which can be spanned by a single OSPF instance is the whole autonomous system (AS). The largest ASes only have about several hundred thousand routers [28]. The average degree also seems to be rather low.

We can derive another limit from IPv4 address allocations. A /8 block (i.e. Class A) has 16 777 216 addresses. Very few ISP would be assigned such a large block, but they might be using the 10.0.0.0/8 private block. Even if an ISP wants to use all those addresses, majority of them will likely not be assigned to routers, but to some end devices that actually provide "useful" services. Those devices are also likely to be grouped into non-trivial networks, thus reducing the number of vertices (i.e. routers and networks) in the OSPF topology. (While IPv6 allocates many more addresses, we assume that the overall topology will not be different from the IPv4 one.)

It is probably not practical to have a single OSPF area span all the routers in a large AS, since any link state change results in an LSA being flooded throughout the area.

While we cannot be sure about particular administrative decisions, given the observations above, we expect that a single area contains at most few thousand vertices and probably much less than that.

3. Design

We now explain the design of Birdvisu in depth. First, we explain some important decisions and present the overall structure of the project, then we look into individual parts of the program.

Birdvisu is implemented in Python [29], using PySide6, the official bindings for Qt6 [30], for drawing on screen. We decided to use Qt, because it provides a lot of pre-made widgets and tools and since it is widely used, it is easy to find help for it on the Internet. The decision to use Python was not hard either. Not only Qt has official bindings for it, but we use the language very often and thus are comfortable writing software in it. We do not expect the potential slowness of Python to be an issue, because for handling graphics we are using Qt, which is written in C++. Also, as we have analysed in section 2.6, we expect the topologies to be quite small.

The project comprises of three main parts: data collection, annotation and presentation part. The data collection part is tasked with finding out the current topology and creating a usable representation of such topologies and their combinations. In the annotation part, we add additional information to the topologies like the difference from the expectation or graph properties of the topology. Finally, when we have all the needed information, we draw the topology on the screen, highlighting the relevant information.

3.1 Recurring and general patterns

Birdvisu's data structures make heavy use of dictionaries and sets, because we do not handle much data that would need to be processed in any particular order. While this allows us to perform set operations quickly, it requires us to provide hashable keys.

We have decided to embrace this requirement and use rather complex frozen dataclasses, which can hold as much of the required data as possible, as long as we can re-create that data.

This can be illustrated on our usage of vertices in topology. There are two objects: a VertexID, and the Vertex itself. VertexID is the hashable part and Vertex provides additional information, like incident edges, which are not hashable. The topology then has a dictionary from the VertexIDs to Vertices, providing the complete data.

However, the VertexID already contains information like what version of IP it belongs in, whether it represents a router and all the possible IP addresses and identifiers related to the vertex. It is sufficient for Vertex objects to only contain sets of edges and references to the related topology and VertexID. (In the next section, we will see that a type of the vertex is also stored in Vertex, but that is really everything.)

The other thing we decided to reuse was the format of BIRD's topology output. We call the format "ospffile" and have extended it by allowing comments (after an octothorpe, i.e. `#`). Also, empty lines do not seem to be of relevance. These are quality-of-life improvements for cases when ospffiles are edited by hand.

Apart from storing topologies, we intend to use ospffiles for description of basic styles. Therefore, our implementation in `birdvisu.ospffile` only constructs the tree of strings and does not try to understand it. Our module provides API similar to the one of `json` or `marshall` modules, even though it cannot represent arbitrary types.

3.2 Data collection: providers and parsing

This part of the project deals with processing topologies. The core object of this part is a TopologyV3[1]. While the Topologies can be created manually by adding the vertices and edges, we expect that retrieving topologies from other sources like saved ospffiles or running BIRD processes. This is made possible by implementing a TopologyProvider.

Representing a topology turns out to be a bit complicated problem for the following reasons:

- The topology edges need to be directed. OSPF allows a shortest path from A to B to be different to the other direction.
- It can have a very general shape, so we cannot rely on common patterns. For example, routers can be connected to other routers using point-to-point or virtual links, not just networks.
- The objects are shape-shifting. A transit network may become stub or change the designated router and we want to be able to understand the change as best as possible.
- The topology is not necessarily a graph, because multiple links may lead from a single router to the same network. However, we strongly believe that the maximum number of parallel edges is quite low, so most of the theory for simple graphs is still applicable.
- For completeness, we note here again that the shortest paths from a single vertex form a DAG, even though the OSPF specifications speak of them as of trees. (Negative edges are, fortunately, not permitted.)

Given the above requirements and lessons learned in section 2.5, we need to find a representation of vertices, that is both powerful enough to uniquely describe a particular vertex, and flexible to allow us easily detect its metamorphoses. The table 3.1 shows, which information we can use for each type of object. We see that networks in particular are hard to represent, because the ID of the DR may change and it might be the only distinguishing property in case of a split network.

We decided to aim for correctness, so whenever any of the attributes of an object change, we consider it to be a a separate object. This may create some false positives, but we think that is the better case than potential false negatives, which could hide some issues. Also, when the infrastructure works correctly, the designated router should only change in case of outage. Therefore, it might

[1]The "V3" suffix is sometimes impractical to keep, so we will sometimes shorten the class name only to "Topology". It denotes the same object.

Object	Address	RID	DR ID	IF ID	Notes
`router`	–	✓	–	–	
`xrouter`	–	✓	–	–	
`vlink`	–	✓	–	–	Peer is a `router`
`network`	v2:✓,v3:∗	–	•	v3:•	
`external`	✓	–	–	–	
`xnetwork`	✓	–	–	–	
`stubnet`	✓	✓	–	–	

Table 3.1: Information determining each object of a topology. ∗ means it may or may not be known, • denotes an attribute that may change. Columns in order: whether it has assigned a address, relevant router ID, ID of designated router, interface number of the DR.

actually be useful to notice the user when a network has an unexpected designated router even when it is otherwise healthy. However, we provide a way to find objects by partial information, using the VertexFinder objects, so this allows heuristics to match different objects.

The information mentioned in table 3.1 serves as the main part of the VertexID. However, we want the VertexID to identify the same object even after it transforms to another kind of object, so instead of using the object type, we only note whether the object is a router or a network, since this property stays the same even for changed objects. The code is also oblivious to the fact that the interface ID is a number and what it means – we use it as an opaque "discriminator" and do not even bother with parsing it from a string.

The VertexIDs are supposed to be descriptors of objective vertex state, so they do not belong to any particular TopologyV3. Instead, they can be used to track actual Vertices across multiple Topologies.

Apart from VertexIDs, the TopologyV3 also consists of the additional data in Vertex objects and Edges. The Vertex objects, as noted above, contain only a set of incoming and outgoing edges, references to their TopologyV3 and VertexID objects and the actual type of the object the vertex represents (i.e. the first column of the table).

An Edge knows the source and target VertexID, its cost and the number of parallel edges with these properties. If the Edge was determined by a virtual link, it is marked as virtual. This is needed, because the both Vertices are regular routers, so the information about the virtual link cannot be stored in them. Note that an Edge does not need to belong to any Topology, since it only contains factual data. The information, whether an Edge is in the topology, is stored only in the incident Vertices.

A Topology can be marked as "frozen". This denotes an intent that it really should not be modified, because other code might rely on the particular shape of the Topology. However, making the Topology truly immutable would be impractical in Python, so we opted for this approach. In case our solution turns out to be prone to accidental modification of the Topology, we will deploy additional countermeasures against that.

Frozen Topologies also allow us to stack them, creating a union of the original Topologies. This way, a single Topology can be used in the visualisation, while

keeping the original information. This mechanism is fully generic, but was mainly invented to allow merging the reference (expected) topology with the actual one (i.e. the current state of the system). The ancestors are stored by a string label in a dictionary of the Topology. While subclassing TopologyV3 into a StackedTopology would probably be a cleaner design, since the only difference is a state of one dictionary, we did not employ this approach.

The TopologyProviders are not very interesting, but are important nevertheless. There are a few caveats with parsing topologies from the ospffile format. First, the edges from routers to networks can only be resolved after the networks are known, since network's level-2 block contains information not present in the level-3 directive for the router (namely, the designated router for OSPFv2 networks and the set of addresses for OSPFv3).

Since BIRD may be running more than one instance of OSPF, the BirdSocketTopologyProvider contains an ad-hoc parser of the response to the `show protocols` command, which seems to be a reliable way to list running instances.

Moreover, BIRD does not seem to expose any way to determine the version of OSPF. So far, we think it is sufficient to guess from the `network` directives, since they seem to contain a hyphen if and only if the dump is from an OSPFv3 instance. (The source code of BIRD suggests that under some circumstances, brackets can appear even in OSPFv2 dump, so that is not a possibility.)

3.3 Annotations

Once a TopologyV3 is obtained, it may be annotated. An Annotator may create an Annotation, which is then stored as a part of an AnnotatedTopology. We now explore design of these objects in detail.

An Annotation is essentially only a holder for any "tags" that are to be attached to the topology. These are represented by a dictionary holding annotations for Vertices, another dictionary for annotating Edges, and a single field allowing the attachment of a tag to the entire Topology. The keys of the dictionaries are VertexIDs and Edges, respectively.

The Annotation can only attach one tag to each vertex and edge, but there are little restrictions of what the tag is allowed to be. The intention is to allow Annotators to provide any useful data they can collect. However, we think that our AnnotatedTopologies could be utilised in other projects, so the Annotation objects ought to be easy to serialise into JSON or other formats.

Annotations do not need to take other Annotations into account, because AnnotatedTopology stores Annotations from different Annotators separately.

The Annotators are a tiny bit more interesting. While these objects are basically wrappers around the `annotate()` method, which takes an AnnotatedTopology and returns an Annotation, there are few twists to it.

First, an Annotator object is intended to be created by the respective AnnotatedTopology in order for it to keep track of all the Annotators. To describe an Annotator, an AnnotatorID is used, which is a re-creatable and hashable recipe for creating that Annotator. It is also used as a handle to reference and scope the resulting Annotation. The AnnotatorID is a pair of the type object of the particular Annotator, and an optional hashable parameter, which is passed to the Annotator's initialiser.

Second, an Annotator might require another Annotator to have already run. We make this possible by allowing Annotators to request another Annotator to be run by the AnnotatedTopology (provided the AnnotatorID), as long as there is not a dependency cycle. This is the recommended method of implementing dependencies of Annotators.

Furthermore, an Annotator can be declared to be idempotent. This affects what happens when the same Annotator is invoked on the same Topology in the same way (that is, using the same AnnotatorID) multiple times. For idempotent Annotators, we know that their output will not change, so the Annotator is not really run. For non-idempotent Annotators, the previous Annotation is removed and the Annotator is run again.

Annotators may not alter the AnnotatedTopology in any way. They are only allowed to return an Annotation, which will be added to the AnnotatedTopology. As with frozen Topologies, this is not enforced by the code.

Annotators may be used for various tasks, including but not limited to performing analysis of the Topology, enhancing it with additional data (e.g. ping response times from other system), or specifying parameters for visualisation. As a part of Birdvisu itself, we ship several annotators: TopologyDifference outputs the differences between the reference and current Topology, and ShortestPathTree marks the edges of the shortest path DAG. The next section describes how Annotators aid visualising the data.

The last important object related to annotation is the AnnotatedTopology. It serves as a coordinator for running Annotators. It does two main things: detects dependency cycles between Annotators, and keeps the Annotations.

The Annotations in the AnnotatedTopology are stored in a dictionary indexed by the respective AnnotatorID. For vertices and edges, only sets of AnnotatorIDs are stored. This way, both iterating Annotations for a Vertex or Edge and examining individual Annotations is fast. Also, our approach isolates unrelated Annotations by putting them into different scopes by AnnotatorID.

However, by using the Annotator's type in AnnotatorID, this design enforces a rather tight coupling between Annotators and users of Annotations, because the consumers of Annotations need to understand the precise format of the particular Annotation. This could be solved by implementing support for "interface-annotators", so that various Annotators may provide Annotations in a commonly understood format.[2]

AnnotatedTopology does not expose a way to delete old Annotations. While we do not expect this to cause big memory leaks, in case it does, an LRU-like strategy might be employed to tame the memory usage. Also, the Annotators could be run dynamically when the Annotation is requested, but our current approach does not need this functionality, so it is not implemented at the moment.

[2]Preliminary work on implementing this approach is present in the `ann_interfaces` branch, but the interaction of implementers of the same interface is not decided yet.

3.4 Visualisation

The visualisation is split into two parts: computing the appearance and actually showing the result. For the former we reuse the Annotator infrastructure. The latter is handled by Qt's Graphics view framework.

The appearance is described by a styling dictionary. For vertices, it contains a position and a highlighting colour. Edges can have a colour, line width and a highlighting colour. However, more styling properties can be defined in the future.

To provide those styling dictionaries, a subclass of Annotators is created, StyleAnnotator. StyleAnnotators only differ from regular Annotators in that they only tag vertices and edges with styling dictionaries. This provides something similar to an interface, helping to uncouple the style from the specific Annotator that provided the respective data. Each Annotator which provides data worth showing has a companion StyleAnnotator to provide the respective style. When drawing, we pick one StyleAnnotator and highlight the graph according to it.

The current approach avoids mixing styles from multiple Annotators, which might be required for more advanced use in the future. We considered using stylesheets similar to CSS, but we think that approach is too heavy-weight. Rather, assigning priorities to the StyleAnnotators could allow a flexible order of applying styles, but at this point this also seems like a unnecessary complication of the project.

We let the user decide, where the vertices should be placed, because they might have some idea or understanding of the system that is not present in the topology. For this reason, we also ignore classical metrics of graph drawing, like the crossing number of the layout. This can be demonstrated on the default Gennet topology: while it forms a planar graph, it makes more sense to let the edges cross as on figure 2.2, because the layered structure is more important.

To store the placement, we reuse the ospffile format. An example is shown in listing 3.1. The top-level contains a `visualisation` directive, so that other information may be stored in the same file in the future. Level-2 contains vertex specification in the same format as in dumps from BIRD. On level-3 there is a `position` directive with the coordinates, but for transit networks, additional details (DR or address) can be provided to specify the correct network. Similarly, we allow a `router` level-3 directive to be used in the `stubnet` block. This format allows using BIRD's output as the basis for the placement file and could be extended by other directives if needed in the future.

```
1 visualisation
2         router 192.0.2.14
3                 position 200 200
4         network 192.0.2.0/28
5                 position 0 1500
6                 dr 192.0.2.14
```

Listing 3.1: Vertex placement description

We try to place vertices without known position in proximity to already placed neighbours, so that the user can easily locate them. Since the neighbours can also

have unknown position, BFS is used: we place the vertices of known positions, then their neighbours in their proximity, then the neighbours' neighbours and so on. When there is a completely unplaced component, we place one of its vertices at random. However, disconnected topologies are of little interest to us.

We tried using Graphviz [31] for laying out the vertices, but we were not satisfied with its result. To demonstrate, the listing 3.2 describes the topology of our home network with Gennet attached. Figure 3.1 then shows how each of Graphviz's layout engines draws the topology. While it could be possible to tweak the engine settings, we believe the user still knows better, so we did not continue exploring this idea.

```
graph "This is not good" {
        rou_x -- {net_7 net_6};
        rou_a -- {net_6 net_4};
        rou_b -- {net_6 net_5};
        rou_c -- {net_4 net_2};
        rou_d -- {net_4 net_3};
        rou_e -- {net_5 net_2};
        rou_f -- {net_5 net_3};
        rou_g -- {net_3 net_1};
        rou_h -- {net_2 net_1};
        rou_i -- {net_1};
        pm -- {internet vl16};
        qs -- {internet vl16};
        tr -- {internet vl15 vl16 vl2 vl42};
        zr -- {net_7 vl101 vl16};
}
```

Listing 3.2: Author's home topology

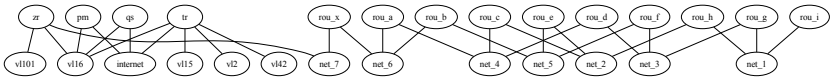
In order to display the topology, we convert it to a simple undirected graph. The set of vertices stays the same, the edges are only reduced to a pair of VertexID without having any kind of cost associated with it. The only tricky part is deciding the style of the new edge when the StyleAnnotator returned multiple styles for the unified edge. We decided to pick the style of the lightest positive-cost edge, since it is the most relevant in most cases: zero-cost edges are almost always network-to-router edges[3] and heavier edges are not used, because it is always cheaper to use the light edge. (We are aware that this is not true for asymmetrically configured point-to-point links, but we do not think they are commonly deployed.)

Since we want to be able to use edges in sets, we need a canonical hashable representation. For that, we implement a total ordering on VertexIDs, which allows us to use pairs of the VertexIDs in ascending order to reference the edge. There are currently no specific requirements for the ordering to satisfy.

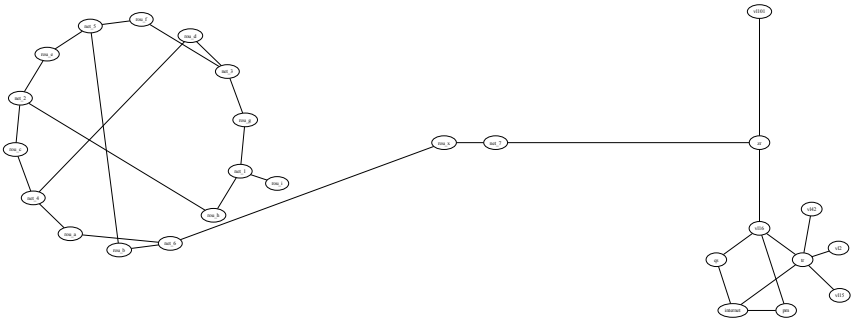
The vertices and edges of the simple graph have a one-to-one mapping to the actual graphics items shown to the user. The Graphics view framework allows us to create nested items, which we use for highlighting: a highlight is just a bigger rectangle in a lower layer than the displayed object. Moreover, the framework has built-in support for dragging and right-clicking objects, which simplifies creating the UI.

[3]We are not sure whether other zero-cost edges are permitted.

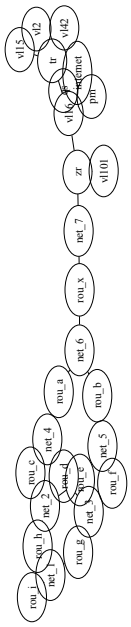
Currently, the MainWindow object acts as a coordinator of everything that needs to happen, from loading topologies and annotating them to putting them on the scene and allowing the user to interact with them.



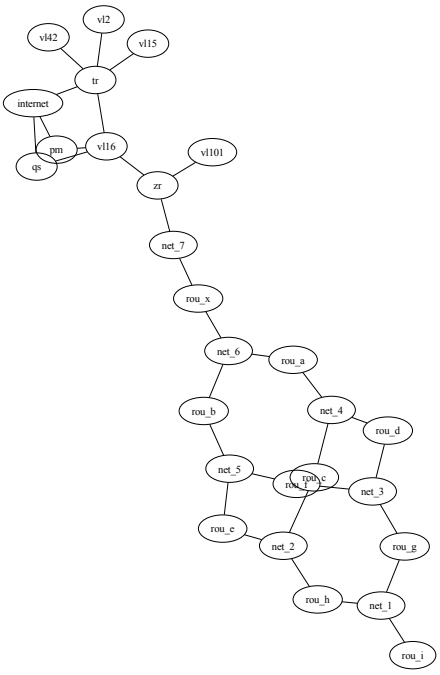
dot



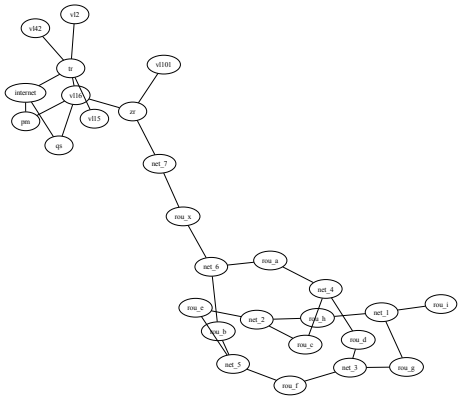
circo



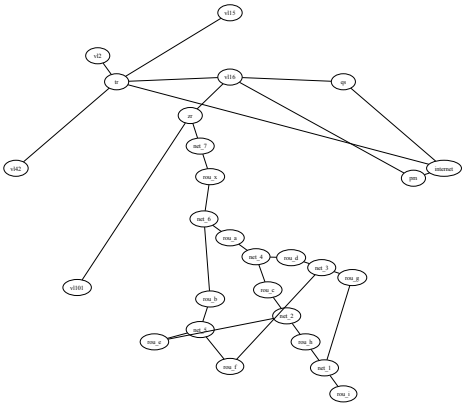
sfdp (rotated for clarity)



neato



fdp



twopi

Figure 3.1: The unpleasant results of Graphviz's layout engines

4. Usage

In this chapter we describe the program from the user's point of view and present its features.

4.1 Running Birdvisu

First, the user needs to obtain Birdvisu from somewhere. Either, the attachment A.1 can be extracted, or the project can be downloaded from its repository at `https://gitea.ledoian.cz/LEdoian/birdvisu`, which will also contain any future improvements.

The only strict dependency of Birdvisu is Python3.10 or newer. While the project depends on the PySide6 library for the user interface, it can be downloaded as a wheel from PyPI. Of course, using a system-wide installation of PySide6 is also possible.

Birdvisu can read the topology information from files, so it is not necessary to have a BIRD running and configured if only showing static data is required.

We only support running the project on Linux. However, Birdvisu might be able to run on other types of operating systems, since we have only used cross-platform libraries.

There are several ways to start the program. The recommended method is to install the project into a Python virtual environment and using the `visu` command. Running the `visu.py` script also works when PySide6 is installed in the system, without requiring the installation step.

Once the program is started, the user is presented with an empty canvas. Using the *Topology* menu, it is possible to load both a reference and current topologies. Both can be loaded from a file, the current one can also be retrieved directly from a running BIRD via its socket. In the latter case, the user is provided with a dialog to pick an area and OSPF instance.

Once both topologies are loaded, the graph of the topology appears. Now, the vertices can be moved around to the user's liking. Alternatively, the *Positions* menu provides a way to load the positions of vertices from a file.

The program expects the topology files to have the `.ospf` extension, positioning files may end in `.visu`. Nevertheless, any files may be used, they just are not offered by default.

Several highlighting ways are available. In the *Highlight* menu, the user can choose to show differences between the reference and current topology, or set the widths of edges according to their costs. Alternatively, by right-clicking any vertex, it is possible to highlight its shortest path DAG. If the user wants to find a specific route, showing the DAG also serves as selecting the start vertex. The context menus for vertices then allow finding the path to those vertices. Note that stub and external networks are sinks in the OSPF topology, so it is expected that the program draws an empty DAG for them.

Finally, there is an option to reload the shown graph in the Topology menu. We decided that the graph should not change unexpectedly, because that could be unpleasant to use.

As a part of Birdvisu we ship the files `gennet.ospf`, `gennet-changed.ospf` and `gennet.visu` for demonstration purposes. These files provide a topology of the default Gennet, the same topology with several manual changes, and a placement of vertices like in figure 2.2.

5. Evaluation

To show that Birdvisu is a usable project, we have tried to use it in several network systems, both synthetic and already deployed. Here we summarise our experience with using it.

Because our project is mainly an interactive program, we do not present any performance or other measurements. Therefore, we rather focus on the subjective feeling of the project and evaluate the design decision from chapter 3.

5.1 Gennet

Naturally, Gennet, as described in section 2.4 is supported quite well. Since we had complete control of the whole network, we could test e.g. multi-area OSPF deployment, which is not common in real network systems, as far as we are aware.

However, only small networks can be simulated in Gennet, because each router requires disk and memory, which may be quite scarce.

5.2 Our home network

At home, the author relies on a dynamic routing to switch between falling uplinks and to provide a simple way to address virtual machines across the network system. This was one of the reasons we decided to look for visualisations of OSPF topology state.

Since it only spans a single room, this system is even smaller than Gennet, containing only three routers and less than 10 networks (the exact number depends on which devices are up at the time). We believe that major issues do not show up at this scale.

Naturally, Gennet can be connected to the home network. At that point, our approach to laying out vertices starts feeling suboptimal, because edges cross unnecessarily often. The connections are clear, however, and this can be alleviated by using a fixed layout in a file. In the future, this could be addressed by using a force-based approach for the automatic layout.

5.3 Department of Applied Mathematics

The department which advised this thesis also uses OSPF in its infrastructure. Again, the main purpose is to address containers and virtual machines in the system. The topology consists of 5 routers and about 27 networks, most of which are stub.

As in the previous case, the main issue is the automatic vertex layout. Also, since most of the networks are stub (and often only contain a single host), the graph could position these networks near each other, or even collapse them into one vertex. Birdvisu does not unfortunately support this at the moment.

5.4 Czela.net

Czela.net [32] is a network system run by a community of network enthusiasts in Čelákovice and the surrounding area. There are about 1600 people connected to Czela.net. This is the typical OSPF deployment, whose main task is to dynamically provide fallbacks in case of outages. It is also an example of a larger network, with 45 routers, 32 transit networks and 178 external routes[1].

Unfortunately, their infrastructure does not use BIRD anywhere and we were not able to connect our instance to any of their transit networks. Therefore we only tested displaying the topology based on a dump from one of their MikroTik routers, which we manually converted to mimic BIRD's format. We believe this should not affect the performance much, because the retrieval of data from BIRD is only occasional and once BIRD dumps the topology, the procedure is the same as with loading the topology from file.

This turned out to be helpful for both us and Czela.net. We discovered that our method of highlighting of costs does not work well with too big range of the costs, and on the other hand, we found several misconfigurations in their network (some routers had external routes to transit networks, a few networks with routing-unaware clients were used as transit), even without any other knowledge of their system.

We did not notice any performance issues when dealing with the topology.

Unfortunately, we did not have the opportunity to test in a large network (the Czela.net's one is the largest we tested), so we do not really know the limits of Birdvisu. Those are even hard to guess, because while we are trying to use rather fast algorithms, they are implemented in Python, which can sometimes be quite slow, and more importantly, does not support threads well. The heavy use of hash tables and indirection can also impair performance.

Overall, the testing did not discover any important problems with the design of Birdvisu, we are only aware of issues related to small parts of the project.

[1] Most of these would be stub routes in other deployments, but there is little difference from the topology perspective.

Conclusion

We developed a simple and standalone tool for visualising OSPF topologies. The program is functional and can visualise network systems of medium size. The project's design will allow additional features to be added in the future.

The project also meets the goals stated in chapter 1:

- Birdvisu utilises BIRD to collect data about the whole network system from a single host,
- the project can be easily run from any Linux machine, not requiring any server-based setup, and
- its topology-centric approach has successfully discovered minor issues with an existing OSPF deployment.

The only stated goal Birdvisu fails to fulfil at this time is the ability to exchange data with other projects. However, the project's design expects such functionality, so enhancing Birdvisu in this way should be easy in the future.

Approach	CD	RS	CoH	EoD	T
Only visualisation	×	?	×	?	×
Traffic visualisation	✓	✓	✓	×	×
Host monitoring	✓	?	✓	?	×
Integrated management	✓	✓	?	×	✓
Topolograph + Ospfwatcher	✓	✓	×	×	✓
Birdvisu	✓	×	×	✓	✓

Table 6.1: Comparison of Birdvisu's approach compared to other known approaches. See also table 1.1.

Future work

There are many ways Birdvisu can be expanded in the future. Apart from the stated ones (exports, better vertex placements, ...), it could be possible to support other routing daemons and even other link-state protocols like Babel.

Another interesting possible use case could be running Birdvisu headless. When combined with the export feature, this could allow using the project as a data source for other visualisation tools. While this is currently not possible, because the GUI controls all the parts of the program, it might be possible to separate the GUI and the coordination part.

More Annotators could also be written to provide other functions, like detecting single points of failure. And last, but not least, the user interface can definitely be prettier.

Bibliography

[1] Cisco Systems Inc. Network Topology Icons. Available at https://www.cisco.com/c/en/us/about/brand-center/network-topology-icons.html. Accessed: 2023-07-17.

[2] Howard Jones. Network Weathermap. Available at https://www.network-weathermap.com/. Accessed: 2023-07-17.

[3] Grafana Labs. Grafana. Available at https://grafana.com/grafana/. Accessed: 2023-07-17.

[4] Inc. The Cacti Group. Cacti – the complete RRDTool-based graphing solution. Available at https://www.cacti.net/. Accessed: 2023-07-17.

[5] The Munin project. Munin monitoring. Available at https://munin-monitoring.org/. Accessed: 2023-07-17.

[6] Jiří Benc. plotnetcfg. Available at https://github.com/jbenc/plotnetcfg. Accessed: 2023-07-17.

[7] Jan Krupa. CaLStats: Computer and line statistics. Available at https://jankrupa.com/projects/calstats/. Accessed: 2023-07-17.

[8] Icinga GmbH. Icinga 2. Available at https://icinga.com/. Accessed: 2023-07-17.

[9] Jernej Kos, Mitar Milutinović, and Luka Čehovin. nodewatcher: A substrate for growing your own community network. *Computer Networks*, 93:279–296, December 2015.

[10] Vadims06. Topolograph. Available at https://topolograph.com/. Accessed: 2023-07-17.

[11] Vadims06. OSPF Topology Watcher. Available at https://github.com/Vadims06/ospfwatcher. Accessed: 2023-07-17.

[12] The Quagga team. Quagga routing suite. Available at https://www.nongnu.org/quagga/. Accessed: 2023-07-17.

[13] Elasticsearch B.V. Logstash. Available at https://www.elastic.co/logstash/. Accessed: 2023-07-17.

[14] John Moy. OSPF Version 2. RFC 2328, April 1998.

[15] Dennis Ferguson, Acee Lindem, and John Moy. OSPF for IPv6. RFC 5340, July 2008.

[16] Zhaohui (Jeffrey) Zhang, Lili Wang, and Acee Lindem. OSPF Two-Part Metric. RFC 8042, December 2016.

[17] John Moy. The OSPF Specification. RFC 1131, October 1989.

[18] Michael Barnes, Sina Mirtorabi, Rahul Aggarwal, Abhay Roy, and Acee Lindem. Support of Address Families in OSPFv3. RFC 5838, April 2010.

[19] Dr. Patrick W. Murphy. The OSPF Not-So-Stubby Area (NSSA) Option. RFC 3101, January 2003.

[20] Acee Lindem, Abhay Roy, and Sina Mirtorabi. OSPFv2 Multi-Instance Extensions. RFC 6549, March 2012.

[21] M Fanto, Randall Atkinson, Michael Barnes, Vishwas Manral, Russ White, Tony Li, and Manav Bhatia. OSPFv2 HMAC-SHA Cryptographic Authentication. RFC 5709, October 2009.

[22] CZ.NIC Labs. The BIRD Internet Routing Daemon. Available at `https://bird.network.cz`. Accessed: 2023-07-11.

[23] XORP, Inc. eXtensible Open Router Platform. Available at `http://xorp.org/`. Accessed: 2019-02-15.

[24] OpenBSD Project. OpenOSPFD. Available at `http://www.openbgpd.org/`. Accessed: 2023-07-17.

[25] FRRouting Project. FRRouting project. Available at `https://frrouting.org/`. Accessed: 2023-07-17.

[26] CZ.NIC Labs. BIRD source code. Available at `https://gitlab.nic.cz/labs/bird`. Accessed: 2023-07-17.

[27] Pallets project. Jinja. Available at `https://jinja.palletsprojects.com/en/3.1.x/`. Accessed: 2023-07-17.

[28] M Abdullah Canbaz, Khalid Bakhshaliyev, and Mehmet Gunes. *Router-Level Topologies of Autonomous Systems*, pages 243–257. February 2018.

[29] Python Software Foundation. Python. Available at `https://www.python.org/`. Accessed: 2023-07-20.

[30] The Qt Company. Qt 6. Available at `https://www.qt.io/product/qt6`. Accessed: 2023-07-20.

[31] The Graphviz Authors. Graphviz. Available at `https://graphviz.org/`. Accessed: 2023-07-20.

[32] czela.net z.s. czela.net: Metropolitní síť v Čelákovicích a okolí. Available at `https://www.czela.net/`. Accessed: 2023-07-20.

Glossary

A *bridge* is a networking device, which forwards link-level frames between interfaces.

The term *community wireless network* describes a system of often wireless networks, which is managed by a community rather than by an ISP.

Dual-stack networks can forward and process both IPv4 and IPv6 packets.

When a program is run without being able to display graphical elements, it is said to be run *headless*.

A *homelab* is a small infrastructure, on which a network enthusiast can experiment and thus improve their networking skills.

Netsplit is just a short form for "network split".

A *next hop* is the name for the following router a packet is forwarded to.

In Linux distributions, a *package* is a common way to distribute software.

Quad-dotted notation denotes writing a 32-bit number as four decimal numbers representing individual bytes, with dots between them. The numbers are written in the network order, also known as big-endian.

A *routing daemon* is a program running on a router that exchanges routing information with other routers.

Python's current way of distributing compiled software is called *wheel*.

List of Abbreviations

ABR – Area border router
API – Application programming interface
AS – Autonomous system
BIRD – BIRD Internet Routing Daemon – BIRD Internet Routing Daemon Inter...
CIDR – Classless inter-domain routing
DAG – Directed acyclic graph
DR – Designated router
GUI – Graphical user interface
IP – Internet protocol
ISP – Internet service provider
LSA – Link state advertisement
OSPF – Open shortest path first
PTP – Point-to-point
PyPI – Python package index
RFC – Request for comments
SMTP – Simple mail transfer protocol
UI – User interface
UNIX is not an abbreviation, but rather a trademark of The Open Group.

A. Attachments

All attachments to this thesis are only in the electronic version. Readers of the paper version can find them on the internet

A.1 Birdvisu

The Birdvisu project is located in the `birdvisu` directory of the attached ZIP file.

Attached as of commit `4abc3cc8`.

Available online at `https://gitea.ledoian.cz/LEdoian/birdvisu`.

A.2 Gennet

The Gennet project is located in the `gennet` directory of the attached ZIP file.

Attached as of commit `4d604955`.

Available online at `https://gitea.ledoian.cz/LEdoian/gennet`.