Autocompletion for Network Configurations

Ahsan Mahmood

Senior Thesis Proposal

Abstract

In this proposal, we state the need for an auto-completion engine for writing network configurations. The tools and technology available today are simply inadequate to help network operators in this process. We propose a simple, yet powerful model inspired by code completion techniques and NLP research. We show how the current state of the model gives encouraging results. We also outline additional work that is required to tune our model specifically for network configurations before we can truly realize our goal. Once completed, we believe our engine will be a strong first step in creating a holistic tool similar to IDEs that can assist network operators.

1 Introduction

A network's backbone is its routing control plane: a set of rules and distributed routing protocols that describe how the network should operate. A control plane is thus defined through configuration files present on every individual routing device in the network. These configurations are written in vendor specific languages (e.g. Cisco and Juniper) and describe very low level behaviours of a particular router. Network operators tasked to configure control planes are required to satisfy various 'policies' that the owning organization wants to enforce: e.g certain devices should always be blocked from communicating with higher privileged devices.

Research has shown that configuring control planes can be extremely complex in modern networks [1]. Consequently, this causes configurations to be prone to errors, most of which are only uncovered during operation after a failure has already dealt significant damage [20].For example, in 2012, failure of a router in a Microsoft Azure data center triggered previously unknown configuration errors on other devices, degrading service in the West Europe region for over two hours [18]. Similarly in 2017, Google made a small error in a protocol configuration which interrupted Japan's Internet for several hours [19]. These examples highlight a need to develop highly resilient configurations that perform reliably.

There are three main areas of research that are trying to enable this resilience: configuration synthesis, configuration development tools and network verification. Synthesis tools like NetComplete [CITE], Zeppelin [CITE] and Propane [CITE] are working towards perhaps the most ambitious solution: automatically generating all configurations without the need of any writing. However, all the existing solutions require comprehensive input from the network operators, outlining the high level requirements of the policies the organization wishes to impose. This can be a tedious task for the operators and they may also be limited by the expressiveness of the language being used to

describe the policies.

Network verification and repair (e.g ARC[CITE]) can be extremely useful to determine whether the current configurations comply with the existing policies of the network. Again, these policies could be defined by the operators or, as in the case of ARC, they could be inferred via a snapshot of the network. However, verification by design is a post development tool i-e it will catch errors after they have already been produced. Between the time the tool is run and a fix is proposed, some damage might already be dealt. It does not help operators avoid producing these errors in the first place. We thus wish to provide a solution that sits in between configuration synthesis and verification. Our penultimate goal would be a "writing assistant" for network configurations, one that can complete entire stanzas (explained in section 2.1), automatically fill in parameters or even suggest more concise syntax. This paper establishes a first step towards this goal: an engine for offering token completions.

We consider the problem of writing network configurations to be analogous to writing software code. Most configurations are written using vendor specific languages, that make use of rules and keywords similar to traditional programming languages. We envision an interactive system inspired by code completion engines that could be invoked by network operators as they are writing router configurations to offer them suggestions for what to put in next, or list the options available from the invocation point.

This paper outlines the necessary details for building a completion engine for network configurations. Section 2 provides background information about network configurations, existing network management tools and code completion techniques. Section 3 describes our token analysis results and explains how our model works. Section 4 discusses our plans for extending the model. Finally, Section 5 reviews all the work pertinent to our research, followed by a conclusion in Section 6.

start background from here?

Currently, network operators try to minimize extraneous features by reusing existing configurations that have been known to work in the past. When creating a network, operators typically write templates containing specific configuration lines that define a base set of behaviours for different router roles [1]. These templates are then used to specialize individual routers to achieve objectives for their respective part of the network. Due to varying router specifications, the template systems used allow network operators to fill in parameters with appropriate information each time the template is used. Writing templates, however, can be an inefficient solution when dealing with special cases that deviate greatly from the predefined archetypal configurations. We thus propose a different approach that can serve to complement existing techniques for writing routing configurations.

Recent research on software systems has shown that codebases tend to contain regularities, much like natural languages [8]. This has motivated further research on using traditional Natural Language Processing techniques for code completion and token suggestion, resulting in fairly accurate models [8, 15]. We hypothesize a similar regularity for network configurations, especially since they tend to be homogeneous by design, reusing the same set of keywords/tokens. Our analysis of router configurations from a large research university showed that configurations shared between 85% and 99% of tokens across different routers. This prompted us to explore simple NLP techniques that could leverage these token similarities to produce useful suggestions or completions. Our preliminary results show that using an off-the-shelf NLP algorithm with minor modifications, can give us up to 93% accuracy for some configurations. These are encouraging results and in Section 4 we discuss many additional methodologies that we can apply to further improve these

results.

2 Background

Generating token predictions for network configurations requires knowledge from multiple domains. In this section we introduce router configurations and the tools available for writing as well as managing them. We also expand on some code completion techniques that have been successful for programming languages, and form a basis our engine.

2.1 Network Configurations

Router configuration files are often written in a vendor specific language, the popular ones being provided by Cisco and Junyper systems. These files often exist as plain text on the routers and are composed of different types of 'stanzas'. A stanza is defined as the largest contiguous block of commands that encapsulate a piece of the router's functionality. The most important types of stanzas include routing protocol, access-control list (ACL), and interface. Each stanza describes the router's particular role in relation to the stanza type. Network operators will configure these stanzas to define how the routers interact with each other. For example, operators might specify which devices the given router is connected to and what protocol it should follow when communicating with such devices. Additionally, they could enforce security measures by using access-control lists to block certain hosts from entering or leaving a network.

Consider the configuration file for the router with hostname A in Figure 1. We can see an ACL stanza near the top which is configured to deny communication from any IP address beginning with 12. A few interface stanzas follow right below it, which define how this router is connected to other routers with some details about the connections (such as costs associated with using those routes). Lastly, at the bottom, we see a routing protocol stanza which states that the router uses the OSPF protocol to connect to two subnets. Here we can see some inklings of general purpose programming languages. We have certain keywords to set parameters (such as cost) for stanzas and define particular aspects of the router. We also have some notion of variables, where we can reuse predefined values. For example the last line in the first interface stanza, we use the access list with id 1 that was defined in the ACL stanza above it. The similarities with programming languages that we observe prompt us towards code completion techniques to help guide the development of our engine.

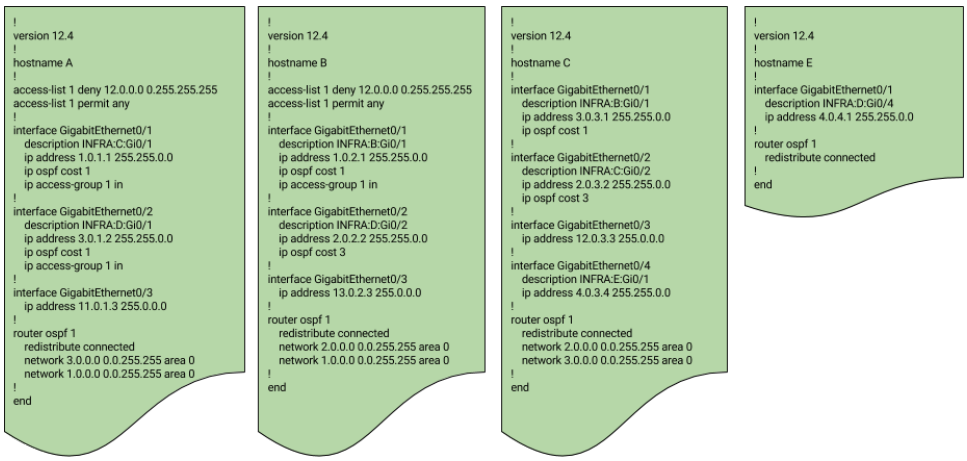

Figure 1: A set of simplified configurations for a small network with four routers employing a single OSPF protocol.

2.2 Code Completion

Traditional completion techniques, such as those seen in IDEs [10], generate context-aware models of program histories. This means the code completion engines have to be aware of the grammar of the programming language and make suggestions based off that. Additionally, they maintain a record of past objects, their types and method calls invoked on them. If the user starts typing code with a similar type structure, they use this model to generate and rank the suggestions. These solutions offer fairly respectable accuracies but come with their idiosyncrasies. Popular IDEs, such as IntelliJ [9] or Eclipse [4], use relatively simple type based inferential techniques to suggest all methods available for an object, usually sorted in alphabetical order. Researchers, on the other hand, have proposed more intelligent forms of code completion techniques in the past. Early work started by adopting rule based approaches where a database of predefined rules could be continuously queried to carry out possible completion tasks [11]. Other researchers explored how to make use of program history to offer suggestions based on what users had done in the past [16].

Eventually people started applying machine learning techniques, such as K-Nearest Neighbours, to extract patterns from existing code bases and building models that could be used to rank possible predictions for a given input vector [2]. All these techniques, however, require some form of context extraction, so that information about the codebase can be stored e.g. in form of a feature vector.They heavily leverage the existing code structure and require knowledge about the grammar of the programming language. A similar methodology for network configurations would require more input from our end to ensure that the context of the tokens was properly understood. We go into a little more detail about some code completion techniques in Section 5.2.

Natural Language Processing (NLP) techniques, on the other hand, can generate predictions based on token usage and do not need to be explicitly aware of the grammar. This allows us to use these techniques independent of vendor specific configuration languages. Our inspiration for using NLP techniques primarily came through Hindle,*et al.* 2012 [8]. This paper provides an excellent insight into the regularity of software code. The authors draw parallels between natural languages and codebases, and show that software is just as predictable as many human languages. They used an n-gram model to demonstrate high regularity in a dataset of Java projects, compared to an English corpus. They also proved that these results arose directly from the natural regularity of the

codebases rather than being an artifact of the programming language being syntactically simpler than English. Similar to Hindle,*et al.*, Raychev *et al.* 2015 [15] took an NLP inspired approach to generating code completions. They reduced the problem to predicting probabilities of sentences, performing static analysis on the code and feeding the results to two statistical language models: N-gram and Recurrent Neural Networks (RNN). They collect a history of method calls and treat them as sentences to synthesize suggestions. Interestingly, using RNNs had negligible effect on their accuracies even though it increased the training time by many folds. Consequently, we have decided to use N-gram models as they seemed to work surprisingly well for both Raychev *et al.* and Hindle,*et al.*. We detail this model in Section 3.

2.3 Network Management Tools

It is important for us to recognize existing tools for managing networks and their shortcomings. One of the main motivations for pursuing this research is the lack of resources available for network operators to write configurations. Most routers offer some form of simple built-in Command Line Interface (CLI), where operators can use vendor specific languages to update router configurations. Often, these CLIs will offer rudimentary tab completion, where they will alphabetically suggest all the options available for a token from the invocation point. These are sometimes unhelpful as the user then has to search for the desired completion.

Enterprise network management tools, on the other hand, are built to assist network operators as they design, maintain and monitor large sets of router configurations. These tools often also offer various functionality to help write router configurations. NetMRI [13], for example, allows users to use existing templates or write their own scripts to automate simple changes across the network. These changes might include adding ACLs, updating the router OS etc. SolarWinds [17], a similar product, claims to simplify and standardize complex configuration changes by creating a single vendor-neutral script that can be scheduled and executed on multiple devices.

A recurring drawback of these tools is that they focus mostly on updating existing configurations. They do not provide any additional functionality for writing new configurations other than utilizing templates. Even in the latter case, the operators will have to fill in the templates appropriately or write their own custom templates for specialized router roles. Our work acknowledges that in practice no networks functionality can be captured by templates alone. Thus, there is a need for an engine that can distinguish itself from these existing tools by being agnostic towards where it is used in the network development life cycle. We expect our engine to perform consistently whether invoked while writing new configurations or updating existing ones.

3 Preliminary Work

In this section we detail all the work that has been done on building our model which is the backbone of our engine. We start by describing the intuition behind using Natural Language Processing, then introduce N-gram models as our NLP technique of choice and close by describing how we incorporated that into our completion engine.

3.1 Token Analysis

Prior research in computer networking hints that we should expect network configurations to share a common set of tokens. As Hindle,*et al.* 2012 [8] pointed out, regularities in bodies of texts

can be easily exploited by NLP techniques. We hope to find and use such regularities in network configurations to generate suggestions.

In [1] researchers identified a few key design decisions commonly made by network operators. Network configurations are designed to be homogeneous as a means of easy maintenance, where some operators start off with common configuration templates with varying parameters. They may then tweak these templates to achieve specialized routing roles if needed. Thus one can posit that configurations across devices in a given network may share a lot of the same tokens, subnets and sometimes even complete stanzas (such as Access Control Lists).

To confirm our hypothesis, we took configurations from a large university network and split up each configuration file into a list of tokens. Tokens included all keywords and subnets with punctuation and newline characters stripped off. For every file we then plotted the percentage of tokens that exist in other router configuration files.

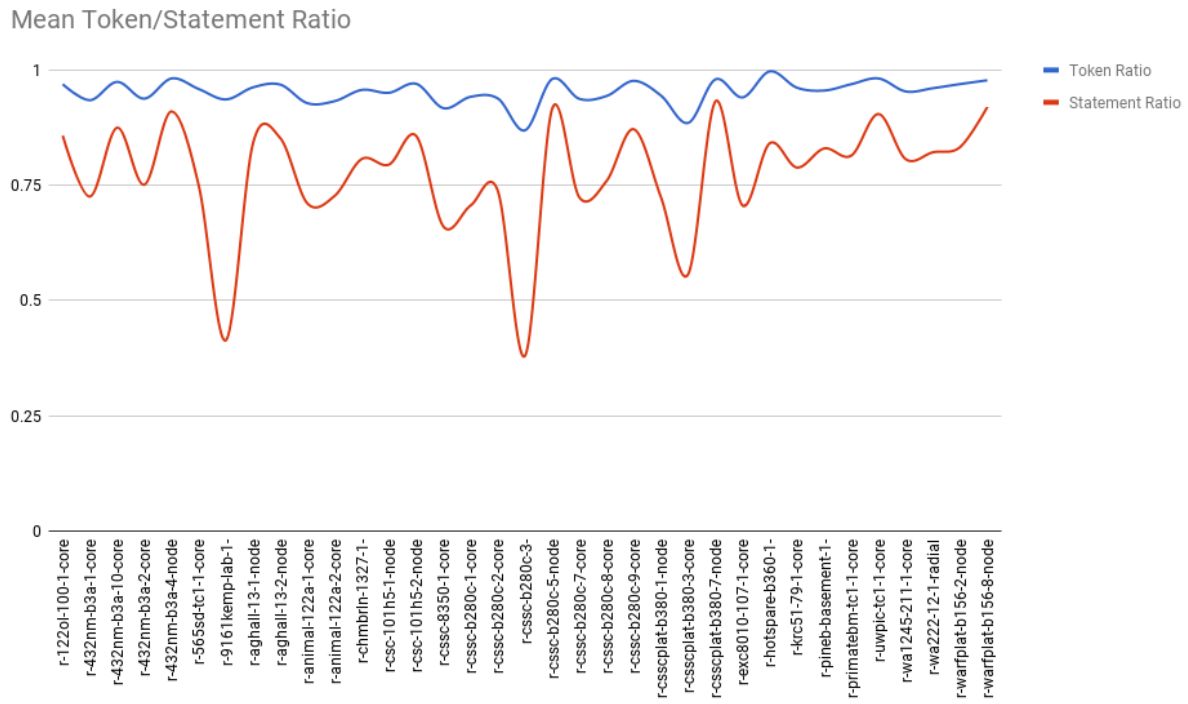

Figure 2: The plot shows how many tokens and statements a router configuration holds in common with the rest of the network. The data was taken from a large research university.

Our results show that most of each file could be rebuilt from existing statements in routing configurations due to the amount of tokens they share. Given our results, and the observations made by [1] about how networks are configured, we can confidently hypothesize that most token suggestions can be generated from analyzing other existing configurations. This effectively makes all router configuration histories a part of the search space for our NLP model.

3.2 N-gram Models

Once we were confident that token regularities existed in network configurations, we began exploring NLP techniques to make use of token histories. As we mentioned in Section 2.2, we have decided on n-gram models as a basis of our engine. Here we briefly describe the theory behind these models.

Consider a sequence of tokens in a body of text (in our case, network configurations). We can statistically model how likely tokens are to follow other tokens. We accomplish this by calculating the conditional probabilities of certain tokens appearing in the text. Given a sequence of tokens $a_1, a_2, a_3, ..., a_n$, we can calculate the probability of a_2 occurring given that a_1 has already occurred i-e $p(a_2|a_1)$. We continue by calculating the probability of a_3 given a_2, and so on. Since we looked at two tokens at a time, this is called a bigram model. More generally, predicting how likely a token is to show up based on the previous $n-1$ tokens is called an n-gram model. In our work, we plan to use bigram and trigram models.

3.3 Likelihood Estimators

The probabilities for suggesting a token pair are estimated by using some scoring function. Often a function may simply count the frequency by which a given pair occurs in the training data and calculate the probability as a ratio of frequency of the pair to the total token count. However, we utilize likelihood estimators as they provided good accuracies for Hindle, *et al.* and have been known to work well for n-gram models [12].

Likelihood ratios are one approach to hypothesis testing. The two hypotheses in our case are:

$$\text{Hypothesis 1: } P(w_2|w_1) = p = P(w_2|\neg w_1)$$
$$\text{Hypothesis 2: } P(w_2|w_1) = p_1 \neq p_2 = P(w_2|\neg w_1)$$

A likelihood estimator is simply a number that tells us how much more likely one hypothesis is than the other. They also have an added advantage of generally being more appropriate for sparse data than other tests. Our system internally uses the Manning and Schutze (5.3.4) version of likelihood estimators [12].

3.4 Model Construction

N-gram models provided us with a strong foundation on which we could build a specialized completion engine. We developed a program in Python which reads in configuration files and builds a bigram model, using the NLTK package [14]. NLTK also allows us to easily incorporate likelihood ratios as a means of scoring the bigrams. Our script was run on sample configurations from the ARC package [7]. These configurations are simple in nature but mimic what deployed network configurations would look like. Each set of configurations emulates a small network employing a different routing policy or design. This allows for a wide breadth of network configuration types to be considered for our model.

Next, we incorporated some preprocessing steps to clean up the data. Since IP addresses and subnets tend to vary a lot and add noise to the data, we replaced them with placeholders. In Section 4, we consider other approaches to help suggest IP addresses. Additionally, our initial analysis showed that the engine was trying to predict what the user would enter after a complete configuration statement. This greatly affected our accuracies, and thus we altered our model to only suggest tokens that appear on the same line.

To test the accuracy of our model, we perform Leave One Out (LOO) Cross Validation. This form of cross validation involves usingone observation as the validation set and the remaining observations as the training set. This is repeated for all combinations of training sets, allowing every observation to act as a validation set. For our analysis an observation is one set of configurations. For example, consider five sets of configurations: A through E. We pick A as the validation set and train the model on all other configurations. Our program will now "walk through" rebuilding configuration A, starting from the first keyword. At every step, we invoke our model and compare our predictions against the actual tokens in A. If the model generates the correct prediction within the top three results, we mark a token completion to be successful.

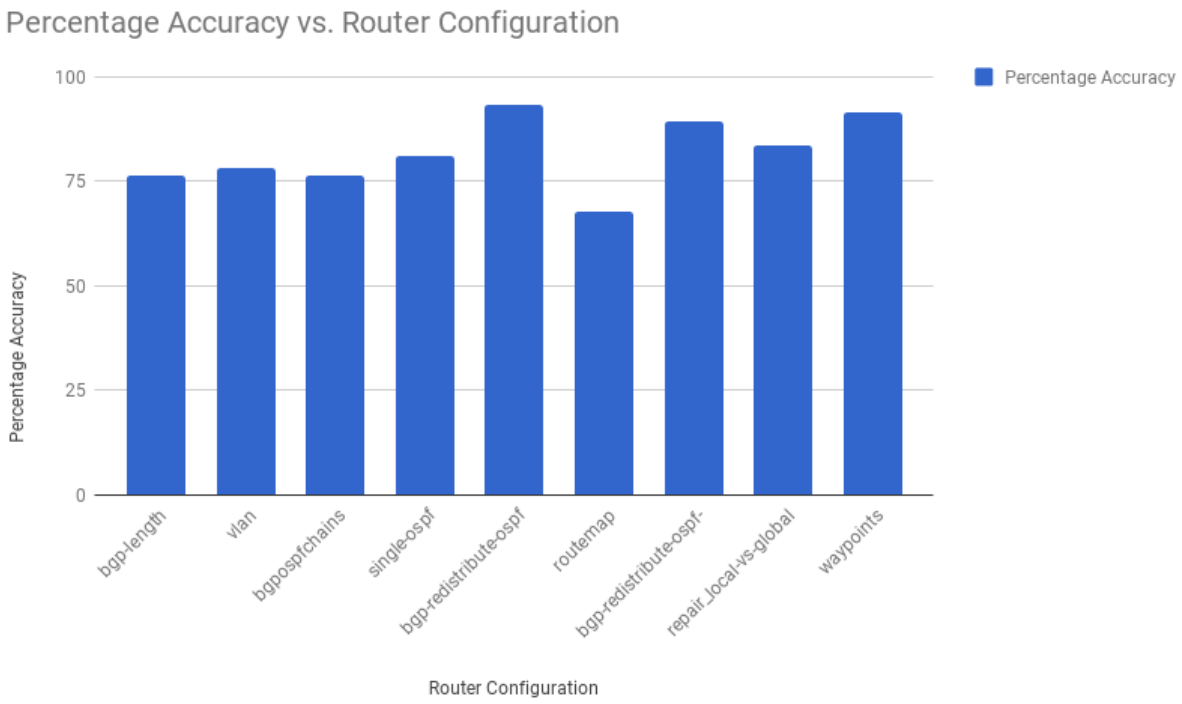

Figure 3: The bar charts show the accuracy of the model for each set of network configurations used as the validation set.

Since we had 10 sets of configurations at our disposable, we performed 10 LOOs and took the average of the accuracies for a final accuracy measure. Initially, without any preprocessing and subnet removal, we observed a maximum accuracy of 85% and an average of 65%. Our results in the figure above are after preprocessing the data and we now see accuracies as high as 93%, and an average of 81%. These results are very promising as we saw a significant jump in accuracy from simple refinements to the model. We are thus optimistic about trying new approaches for further improving accuracies that we describe in the next section.

4 Future Work

There are many directions in which we can expand on this work. An obvious next step would be to test the accuracy of our trigram models. When using N-gram models, researchers will often start

at a higher number and fallback on lower order N-grams. To that effect, we could first use the trigram model to generate suggestions, and if the results are unsatisfactory we could invoke the bigram model. A combination of the two results should result in improved performance. It should also be relatively easy to add additional placeholders in the preprocessing step, such as for VLAN numbers and interface numbers. Additionally, we acknowledge that we have limited ourselves to n-gram models as we deemed it an excellent starting point. However, it is imperative for us to explore other code completion and NLP techniques (such as Recurrent Neural Networks) before we can confidently declare n-grams as the final choice for our engine. In doing so, we also leave open the possibility of developing a hybrid model which offers the best of individual ones.

As we mentioned earlier in Section 3, there are some techniques that we could explore to generate custom completions for IP addresses and subnets. Currently, the model will generate all the addresses that it has seen before during training. It would be possible to improve these results if we could store a mapping of all the subnets that the router is known to be connected to. Then if a network operator wants to add an IP address we would be able to suggest only those addresses that are relevant to that particular router.

One extremely useful addition to the model would be context awareness. We could generate customized completions for different stanza types in the configuration files. For example, a routing interface stanza uses certain keyword like neighbor and network, more often than other stanzas. Our engine should then weight these keywords higher if it is invoked within a routing stanza. Existing configuration parsers like Batfish [6] already have the functionality to be context-aware of stanzas. We would like to explore ways in which we can extract information using such parsers and incorporate it into our engine.

Lastly, we have additional plans for evaluating our model. It should be relatively straightforward to train and test on real-world configurations. We already have access to a dataset from two universities and it should be simple to scrape additional ones from online data sources such as router vendor documentations and publicly available Internet configurations. Additionally, we would like to ascertain the extent to which our model is generalizable. This would require multiple analyses across configurations that vary with time, owners, device types etc. This will allow us to see whether our model is confounded when tested on sources that are different in nature to the training set.

5 Related Work

In this section we discuss areas of research that are pertinent to our work.

5.1 Configuration Complexity

Perhaps the most relevant work on this topic is done by Benson *et al* 2009. In this paper, the authors develop a family of complexity models and metrics that describe the complexity of the design and configuration of an enterprise network.They describe three key metrics for measuring the complexity of designing networks: referential complexity, inherent complexity, and router roles. Referential complexity is the measure of the number of reference links from one router to other routers in the network. Inherent complexity tries to measure how policies dictating the function of a router impact the performance of the network. However, the metric most pertaining to our work is router roles. The authors recognized that while creating a network, the operators will

define a base set of behaviours that will be present across all routers in the control plane. They proceed to argue that networks become more complex to manage as router roles increase and as routers start to play multiple roles. As we have mentioned before in this papers introduction, a common cause for network outages is configuration errors, which stem from the complexity of the network. If we can facilitate the building of new configurations, then perhaps we can also help reduce common mistakes. Highly complex routing designs leave more opportunity for configuration errors to occur [3] which, as we mentioned in our introduction, are the leading cause for network outages.

5.2 Code Completion Tools

There are many code completion engines for software codebases. We gave a brief overview in Section 2 but here we go into a little detail about how they work.

Kaiser *et al.* 1988 was one of the earliest works done on auto-completion for code. The authors present an architecture that provides intelligent assistance by automating certain tasks like compiling or completing missing parameters to a function. The assistant maintains of a database of all the entities in the software system, such as modules, procedures, types etc., and a comprehensive collection of rules that define the conditions in which the assistant tools may carry out a possible task. Overall, the architecture presented in this paper takes a rule-based approach reminiscent of expert systems. If a rule is not implemented by the developers, then the engine is unable to provide any assistance. This architecture seems as a precursor to modern software development technologies which offer much of the same assistance. IDEs such as IntelliJ and Eclipse also offer tab completion, which ranks all possible completions from the invocation point in alphabetical order. These IDEs often use type inference to collect all possible methods available to a certain variable.

Robbes *et al.* 2008 and Bruch *et al.* 2009 showcase how program history can be used for maintaining a model of existing code. They store information about changes made in groups of work that are written close together in time. The researchers used various algorithmic techniques to generate code completions. Usually, these algorithms employ some form of variable contextualization. Robbes *et al.* 2008 makes use of static type inference with a clever session based history which first finds similar code segments and then recommends what was written in the same time frame. Bruch *et al.* 2009 on the other hand, extracts the context of variables and encodes them as a feature vector for those particular variables. Features may include the type of the variable, methods already invoked on it, the method in which it is called etc. Their algorithm (which is similar to KNN) can then use these feature matrices along with the input context from the user to retrieve possible recommendations which are determined by their distances from the input vector. Completion techniques involving program histories offer fairly respectable accuracy results but come with their idiosyncrasies.

5.3 Configuration Synthesis

In recent years, researchers have developed network configuration synthesis tools that tackle the problem of generating network-wide configurations. These tools can build router configurations entirely from scratch by using high level policies the owning organization wants to enforce, which are provided by network operators as input. Configuration synthesizers can be extremely useful to

avoid bugs, guarantee policy compliance, and sometimes even offer network resilience.

SyNET [5] and Zeppelin are two prominent examples of such tools. SyNet accomplishes this by modeling the routing protocols as a stratified Datalog program, and synthesizing inputs such that they satisfy certain policies or path requirements that comply with the operators requirements. Zeppelin on the other hand, employs a two phase solution: first synthesizing policy compliant paths, and then generating configurations guided by those concrete paths. Zeppelin offers increased connectivity resilience over configurations generated by SyNet as minimizing the number of static routes used in the configurations. There are other tools such as Propane and Cocoon that similarly use high level specification to generate low level configurations, though we did not study them extensively.

These systems demonstrate that it is possible to use policy constraints to guide configuration creation. However, they require well-defined and thorough policies to be made by the operators, which may be a tedious task for larger networks. Additionally, depending on the complexity of the networks and the policies , synthesis from scratch can take long periods of time and would have to be repeated whenever a new policy is introduced or an existing one is changed. Finally, these systems require the replacement of the entire current network control plane, which can incur significant overhead and network downtime.

It is perhaps possible for these systems to be used in conjunction with code completion techniques to provide a specialized network configuration completion engine. It would require us to develop some intermediate system between the policy definitions and the synthesis techniques proposed by SyNet and Zeppelin. For now, we simply consider configuration synthesis as a potential area to explore future improvement to our engine.

6 Conclusion

Compared to software developers, network operators are often left neglected when it comes to development tools. Our work tries to bridge that gap by providing a simple completion engine that could be incorporated into a more extensive tool. Our initial findings show that we can get fairly respectable accuracies with off-the-shelf NLP techniques. However, we propose a myriad of refinements to the model that could potentially improve our accuracies to desirable numbers. We plan to implement these features as part of a complete senior thesis.

7 Acknowledgments

I would like to thank Professor Aaron Gember-Jacobson for his continuing guidance and support throughout the semester. I would also like to thank the Colgate Computer Science department for allowing this independent study to be carried out (and hopefully continued).

References

[1] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. pages 335–348, 2009.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the*

ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[3] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting edge of ip router configuration. *SIGCOMM Comput. Commun. Rev.*, 34(1):21–26, Jan. 2004.

[4] Eclipse. Eclipse oxygen. `https://www.eclipse.org/`.

[5] A. El-Hassany, P. Tsankov, L. Vanbever, and M. T. Vechev. Network-wide configuration synthesis. *CoRR*, abs/1611.02537, 2016.

[6] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, 2015. USENIX Association.

[7] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. August 2016.

[8] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016.

[9] JetBrains. Intellij. `https://www.jetbrains.com/idea/`.

[10] JetBrains. Intellij autocompletion documentation. `https://goo.gl/1MrE8o`.

[11] G. E. Kaiser and P. H. Feiler. An architecture for intelligent assistance in software development. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 180–188, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[12] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[13] NetMRI. `https://www.infoblox.com/products/netmri/`.

[14] NLTK. `http://www.nltk.org/`.

[15] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6):419–428, June 2014.

[16] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[17] SolarWinds. `https://www.solarwinds.com/network-management-software`.

[18] Y. Sverdlik. Microsoft: misconfigured network device led to azure outage. `https://goo.gl/Y5sDov`.

[19] Web. Google made a tiny error and it broke half the internet in japan. `https://thenextweb.com/google/2017/08/28/google-japan-internet-blackout/`.

[20] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. pages 159–172, 2011.