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 networks 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 may also be 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 [14].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 [12]. Similarly in 2017, Google made a small error in a protocol configuration which interrupted Japan's Internet for several hours [13]. These examples highlight a need to develop highly resilient configurations that perform reliably.

Network operators thus 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. 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.

Recent research on software systems has shown that codebases tend to contain regularities, much like natural languages [5]. This has motivated further research on using traditional Natural Language Processing techniques for code completion and token suggestion, resulting in fairly accurate models [5, 9].We hypothesize a similar regularity for network configurations, especially since they tend to be homogeneous by design, reusing the same set of keywords/tokens. Some of our work over the summer tried to quantify this similarity between configurations. We analyzed router configurations from a large research university and calculated the average number of tokens shared by a particular router with the rest of the network. Our preliminary results 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. We plan to train an n-gram model using existing configurations and evaluate the accuracy of the suggestions generated.

2 Background

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 files on the routers and can be thought of as a static rule base for the device. A configuration file is composed of different sections which are called stanzas. Stanza types could be router, access-control list, interface etc. 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.

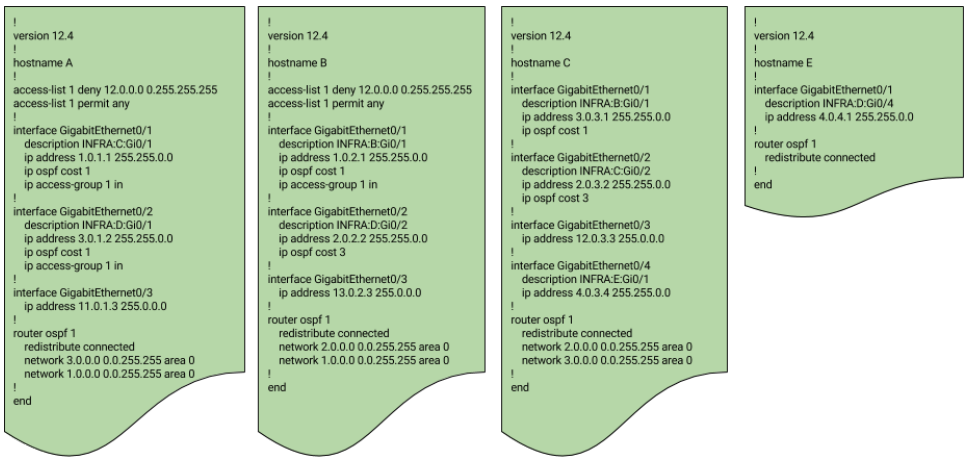

Figure 1: Here is what a set of simplified configurations for a small network employing a single OSPF protocol

2.2 Network Management Tools

Network management tools are built to assist network operators as they design and manage router configurations. Most of these tools offer some form a Command Line Interface, where the 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.

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.

2.3 Code Completion

Traditional completion techniques, such as those seen in IDEs, generate context aware models of program histories. In doing so, code completion engines often have to be aware of the grammar of the programming language and make suggestions based off that. These solutions offer fairly respectable accuracies but come with their idiosyncrasies. Popular IDEs, such as IntelliJ or Eclipse, 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 [6]. Other researchers explored how to make use of program history to offer suggestions based on what users had done in the past [10].

Eventually people started applying machine learning techniques, such as KNNs, to extract patterns from existing code bases and building models that could be used to rank possible predictions for a given input vector. 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. However, NLP techniques 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.

2.4 N-gram 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. These probabilities would be estimated by counting the frequency by which a given pair occurs in our training data. 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.

2.5 Likelihood estimators

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 [7].

3 Preliminary Work

3.1 Token Analysis

Prior research in computer networking hints that we should expect network configurations to share a common set of tokens. 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 few sample networks 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.

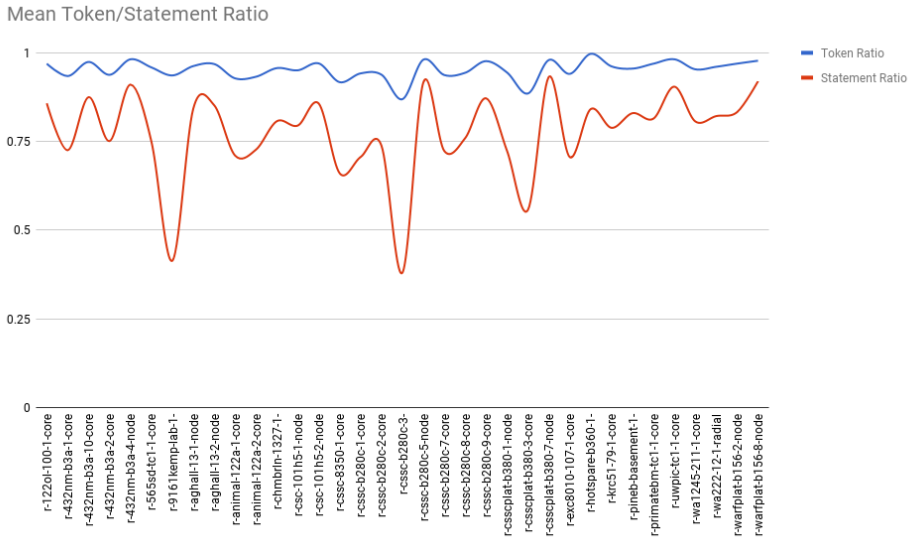

Figure 2: The plot shows how many tokens and statements a router configuration holds in common with the rest of the network

Our results show that most files 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 Model Construction

During the semester, we developed a program in Python which builds a bigram model out of input files. We use the NLTK package [cite] to build the model. The package 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. 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. 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 the future work section, we consider other approaches to help suggest IP addresses.

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 set 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.

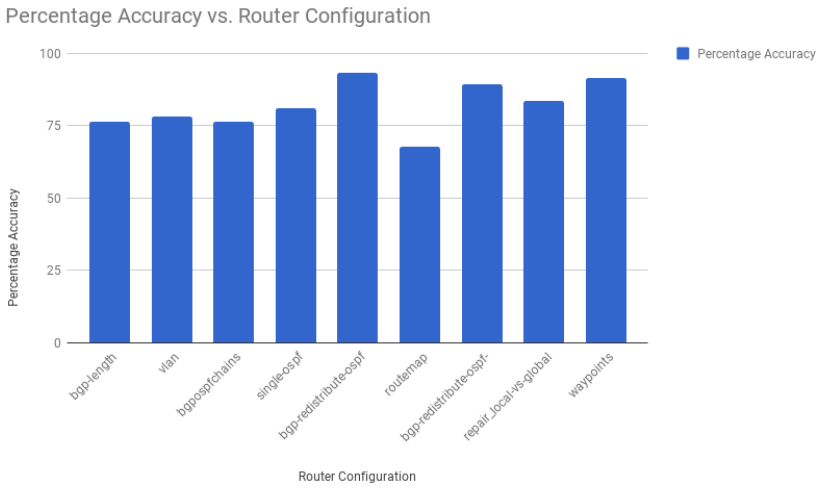

Figure 3: The bar charts show the accuracy of the model for each set or router configurations used as the validation set

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. Our results are promising and we see accuracies of up to 93%, and an average of 81%.

4 Future Work

There are many directions in which we can take 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.

As we mentioned earlier in the Preliminary Work section, 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 [4] 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

5.1 Network Management Tools

The aims and goals of network management tools most closely align with ours. They are used by network operators in large organizations to help monitor and maintain computer networks. These tools often also offer various functionality to help write router configurations. NetMRI [8], 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 [11], 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. As mentioned in our Background section, these tools tend to offer CLI solutions with simple tab completion.

5.2 Code Completion Tools

There are many code completion engines for software codebases. We gave a brief overview in the Background section 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.

Our inspiration for using NLP techniques primarily came through Hindle,*et al.* 2012. 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 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.*.

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 [3] 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.

5.4 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 which [2], as we mentioned in our introduction, are the leading cause for network outages.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

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

[13] 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/.

[14] 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.