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 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. [cite] 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.  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 [source]. [**google example**]. 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 [Benson, 2011]. 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.[source?] 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.[Naturalness] This has motivated further research on using traditional Natural Language Processing techniques for code completion and token suggestion, resulting in fairly accurate models.[Raychev][Naturalness]. We hypothesise a similar regularity for network configurations, especially since they tend to be homogenous 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.

NW managenet 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 network’s 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.

Background

NW 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. Network operators will use these configurations 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. EXPLAIN STANZAS [Here is what a set of nw configs would look like of they were abstracted away into pseudocode]

CC engines

Code completion engines used in 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. However, researchers 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 [intelli assistance].  Other researchers explored how to make use of program history to offer suggestions based on what users had done in the past [Robbes,2008].  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. Context abstraction like these are a little more difficult for network configurations where we do not have an explicit notion of ‘type’ or variables, and would have presented a range of its own challenges. Instead, we opted for a more straightforward solution.

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. 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.

N-gram Models

Consider a sequence of tokens in a document (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 a n-gram model. In our work, we plan to use bigram and trigram models. [could be more technical but make it accessible]

Likelihood estimators

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

H1

H2

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.

Preliminary work

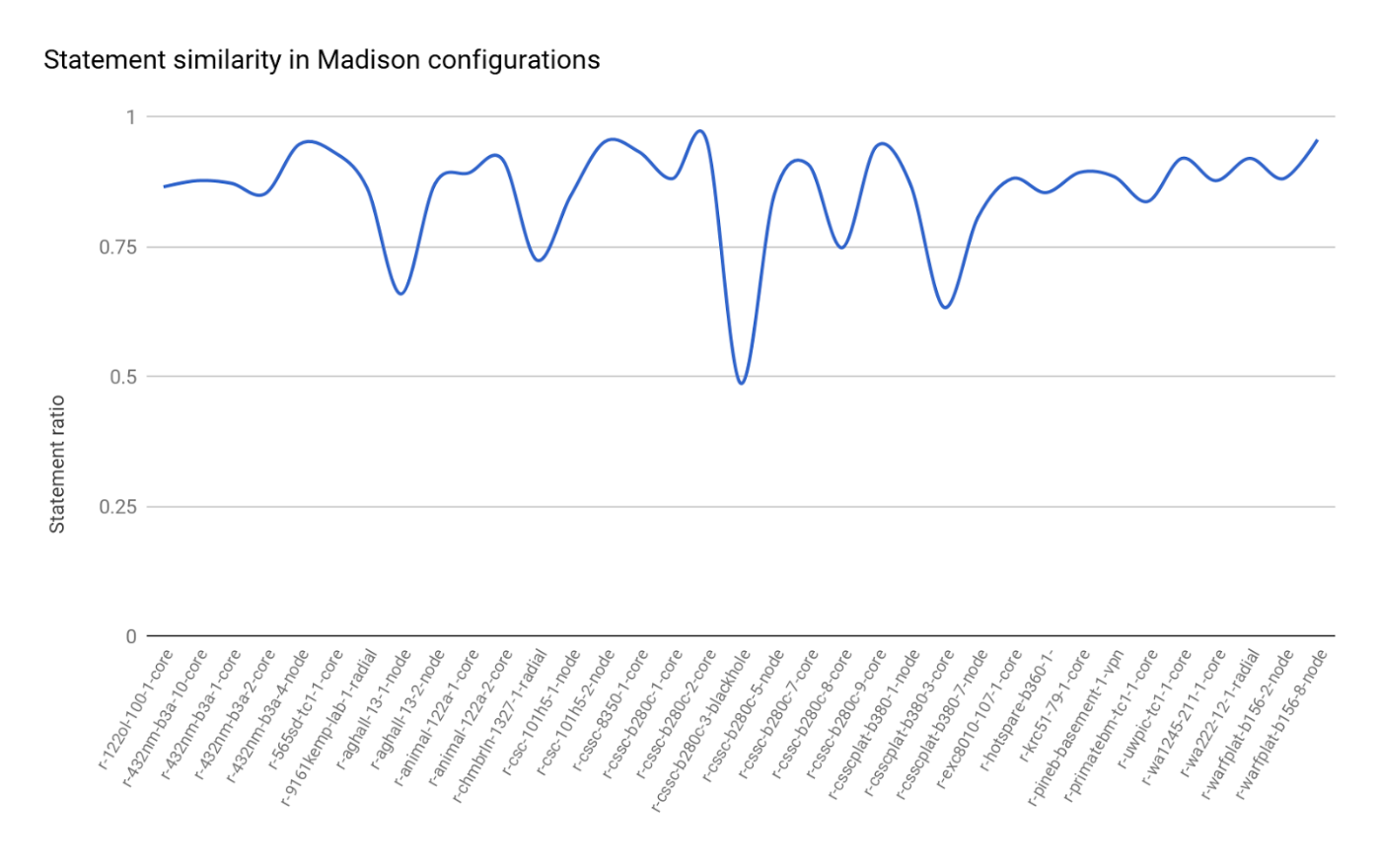
Summer work

In order for an evolutionary algorithm like Genetic Programing to work, we need a viable non-infinite sized search space. In [GP paper], the designers used the program code base as the search space to fix bugs in one location of the program, a decision that relied on the assumption that any missing functionality on the program could be adapted from another location. Prior research in computer networking hints that we should expect a similar trend to hold for network configurations as well. In [NW Complexity] 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.

Ratio of configuration shared with the rest of the network:

|  |  |  |  |
| --- | --- | --- | --- |
| Configuration | Token Ratio | Statement Ratio | Unmatched Tokens |
| B | 0.921 | 0.714 | Hostnames,  Interface Descriptions,  Subnets |
| C | 0.931 | 0.750 |
| D | 0.886 | 0.6 |
| E | 0.857 | 0.625 |



Our results show that most files could be rebuilt from existing statements in routing configurations due to the amount of tokens they share. We realize that some fixes that require completely new rules, such as a unique access list or additional interfaces, would be impossible to build when we restrict ourselves to statement level mutations. It could, however, be possible to generate these fixes from different tokens and intelligent parameter variation. This paper does not explore that possibility but it does make note of it as a future improvement.

Given our results, and the observations made by [NW complexity] about how networks are configured, we can confidently hypothesize that most broken policies can be generated from other existing configurations. This effectively makes all router configuration histories a part of the search space for our NLP model.

Semester

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 using one 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.

Since we had NUMBER sets of configurations at our disposable, we performed NUMBER LOOs and took the average of the accuracies for a final accuracy measure. Our results are promising and we see accuracies of up to 89%. [Figure]

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 fist 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 use 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[cite] 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.

Conclusion

In this proposal we stated the need for an autocompletion 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 shows 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 wholistic tool similar to IDEs that assist network operators.