LG focused crawler

Our LG focused crawler starts from a set of well-known LG sites and consists of two key components: a crawling procedure a classification procedure. It should be noted that we execute the crawler program based on a host within the Tsinghua University campus network.

./LG seed set

The code and results of retrieving a set of known LG URLs from four public LG portals in April 2020 are included in four subdirectories (./peeringdb, ./traceroute.org, ./bgplookingglass, and ./bgp4). In a subdirectory ./All-LGseedpages, we first merge four sets of known LG URLs into one list (see inputseed.csv) including 2991 unique known LG URLs, and then use the python library Requests to download corresponding LG pages. There are 1,736 html files are downloaded successfully (see urlcontentnew.csv) in April 2020. After a manual check, we obtain 1085 valid html files that are providing looking glass services now (see ./All-LGseedpages/afterManualCheck/isLG.csv).

./Crawling procedure

We implement the hyperlink-guided search and similarity-guided search to locate more effective candidate URLs. A subdirectory **./initial** includes the code and search results of the first iteration that takes the LG seed set as input. Specifically, each search method is implemented as follows:

- **./Title-based guided search**: Firstly, we use the Apriori algorithm to extract frequent words or phrases from a list of LG titles as shared features (see ./frequent itemset mining). Specifically, we run deal.py to obtain all LG title information from the LG seed set and replace names and AS numbers in titles with two virtual words, namely ORG and ASN. The generated file item.csv is used as input of apriori.py (i.e., run python apriori.py -f item.csv -s 0.12). After running, we can obtain frequent itemsets with their corresponding support and record them in rule.csv. For the frequent 3-itemsets, we replace ORG and ASN with the name and the number of every AS on the Internet to construct 97,947 search terms (see URLrecord.csv). Secondly, these search terms are entered into the Bing search engine to collect candidate URLs in July 2020 (see ./crawler). The collected 461,799 candidate URLs are recorded in the file titlesearchURLs.json.
- **./Body-based guided search**: Firstly, we use the TF-IDF weighting model to analyze page bodies of the LG seed set (see ./TF-IDF). Specifically, we run _init_.py to extract informative texts from all valid seed html files and merge them into a document (see LGselectedcontent.json). The document and the 20Newsgroups file are taken as inputs of TF-IDF (implemented in deal.py) to generate 49 words with TF-IDF weights larger than 0.05 (see itemfrequent.csv). secondly, each individual word combined with the word looking glass is entered into the Bing search engine to collect candidate URLs on July 2020 (see ./crawler). The collected 19,793 candidate URLs are recorded in the file bodysearchURLs.json.
- **./URL-based guided search**: We enter all LG seed URLs into Bing to get candidate URLs (see ./crawler). The collected 433,865 candidate URLs are recorded in a file fatherURLs.json.
- **./Hyperlink-guided search**: We directly extract hyperlinks from html files of LG seed URLs as candidate URLs (see ./crawler). The collected 4,436 candidate URLs are recorded in a file LGinlinkurlfirst.json.

Whenever relevant URLs are discovered by the following classification procedure, they can be used as known LG pages to start a new round of iteration. The specific code and search results of later three iterations are included in subdirectories ./second_iteration, ./third_iteration, and ./fourth_iteration. Taking the **./second_iteration** as an example, we introduce details as follows:

- **./Hyperlink guided search**: We conduct ./_init_.py to extract hyperlinks from html files of relevant URLs (obtained from the initial iteration) and conduct ./deal.py to remove duplicates. The obtained candidate URLs are recorded in a file LGinlinkurlfirstnew.json
- **./Similarity guided search:** We enter all relevant URLs (obtained from the initial iteration) into Bing in August 2020 (see ./_init_.py). The collected candidate URLs after removing duplicates are merged into a file fatherURLsnew.json. (see ./deal.py)

./Classification procedure

We develop a two-step classifier, including a URL-based pre-filter and a content-based classifier. The specific code and evaluation results are introduced in the following.

- **./URL-based pre-filter**: In terms of datasets, we randomly divide the LG seed set and the four candidate URL files (obtained from the above crawling procedure) into three subsets and manually label URLs in the validation dataset (1\%) and testing dataset (1\%) (see ./training_validation_datasets). Besides, the code and classification results of our URL-based pre-filter are included in a subdirectory ./PUbagging_model. Specifically, we implement the PU Bagging model in baggingPU.py. To choose appropriate hyperparameters, we run _init_.py to train multiple URL-based pre-filters under different hyperparameters, and run deal.py to calculate their AUC performance on the validation dataset. Additionally, Figure 2(a) which draws the distribution of TPR and FPR of the trained pre-filter under different T in our paper is also drawn with deal.py. To evaluate the generalization ability of the trained pre-filter with the optimal hyperparameters, we run deal.py to calculate the TPR and FPR performance of the pre-filter on the test set. It is worth noting that the classification results of classifying all candidate URLs collected from the above first iteration of the crawling procedure are recorded in a file baggingresultnew.csv.
- **./Get pre-filtered URL**s: According to the above classification results, we filter out irrelevant URLs and download html files of the remaining 129,926 pre-filtered URLs (see ./initial/_init_.py). There are 77,113 pre-filtered URLs (recorded in getcontenturl.csv) that respond successfully and we get their html files in August 2020.
- **./Content-based classifier**: In terms of datasets, we randomly divide the 1,085 valid seed html files and the successfully downloaded pre-filtered URLs into three subsets and manually label URLs in the validation dataset (3\%) and testing dataset (3\%) (see ./training_validation_datasets). As inputs of the PU-Bagging model, we run ./Extracting_features/_init_.py to extract texts and attributes inside input, select, and button elements and the page title of each html file. As for the code and classification results of the content-based classifier, they are included in a subdirectory ./PUbagging_model. To choose appropriate hyperparameters, we train multiple content-based classifiers under different hyperparameters and calculate their AUC performance on the validation dataset using _init_.py. Figure 2(b) which draws the distribution of TPR and FPR of the trained classifier under different T in our paper is also drawn with _init_.py. The classification results of the trained classifier with the optimal hyperparameters on classifying the above pre-filtered URLs are recorded in a file relevanceprediction.csv. To evaluate the generalization ability of the trained classifier, we also implement the function of calculating the TPR and FPR performance on the test set in _init_.py.
- **./Get relevant URLs**: According to the above classification results, we filter out irrelevant URLs and record 4,226 relevant URLs with their html files (see ./initial/_init_.py). These files can be used as input of the second round of iteration to search more candidate URLs.

Candidate URLs collected from a new round of iteration will be further classified by the trained pre-filter and classifier. Taking the second round of iteration as an example, the specific code and classification results are as follows:

- **./Get pre-filtered URLs/second_iteration**: Using the above trained URL-based pre-filter, we run ./_init_.py classify new candidate URLs into relevant or not (a file baggingresultnew.csv records the classification results). According to the above classification results, we obtain unique pre-filtered URLs (see ./prefilteredurl.csv). Then, we run ./crawler.py to download html files of the pre-filtered URLs. There are 69,774 pre-filtered URL (recorded in ./getcontenturl.csv) that respond successfully and we get their html files (see urlcontent.csv) in September 2020.
- **./Get relevant URLs/second_iteration**: Firstly, we run ./deal.py to extract features from html files of the above pre-filtered URLs. Then, we run ./_init_.py to use the trained classifier to classify the pre-filtered URLs into relevant or not. The relevant URLs and their html files are recorded in ./relevanceLG.csv and ./LGallcontent.csv, respectively.

./Practical_applications

We develop a tool to retrieve automatable LG VPs and show practical values of automatable LG VPs. The retrieved automatable LG VPs can be publicly available at **https://zhuangshuying18.github.io/discover_obscure_LG/**.

The code of retrieving automatable LG VPs is included in a subdirectory **./Automatable_tool**. According to sources of relevant URLs, we divide the subdirectory into ./initial, ./second_iteration, ./third_iteration and ./forth_iteration. Taking retrieving automatable LG VPs from relevant URLs that collected from the first round of iteration, we introduce **./initial** as follows:

- **./matchtemplate**: Firstly, we run ./_init_.py to extract form elements of each LG seed URL and each relevant URL and record the element information in seedLGleafXpath.json and RelevantLGleafXpath.json. Secondly, we run ./_init_.py to retrieve input interface information about VPs from LG seed URLs and relevant URLs by matching their form element information with known templates.
- **./matchkeyword**: As for the remaining relevant pages, we run ./_init_.py retrieve input interface information about VPs from LG seed URLs and relevant URLs by matching their form element information with specific keywords.
- **./request_response**: Firstly, with the above interface information, we can run ./_init_.py to translate ping measurement requests into the action of filling in the input fields of the corresponding form element with specific values (e.g., ./pingcmd.csv). Then, we run ./_init_.py to automatically issue ping measurement requests to ask the VP to ping a controlled machine (IP is 112.124.15.132) in October 2020. The controlled machine which runs tcpdump successfully records all incoming ICMP packages (see resultnew.cap and resultnewbs.cap). To parse these packets and determine automatable VPs, we need to run dealcap.py. In the meanwhile, information about the automatable VPs is recorded in files (see seedLGlist.json and RelevantLGlist.json).

The code and results of analyzing measurement capabilities, geographic and network coverage of the automatable LG VPs are included in a subdirectory **./Analysis**. The details are as follows:

- **./obsecureLGlist.py**: We run the script and conclude there are 1,446 automatable VPs from the seed LG pages, and 910 obscure automatable VPs from our crawler.
- **./measurementcapacities.py**: We run the script to calculate the number of obscure and known automatable VPs that support each command.

- **./coverage**: Under this subdirectory, we implement ./VP2AS and ./VP2Geo to learn the location of each VP from its IP using IP2AS and geolocation databases. The more accurate information about the automatable VPs are recorded in files ./VP2AS/Relevant3LGlistupdaterevise1.json, ./VP2AS/Relevant2LGlistupdaterevise1.json, ./VP2AS/Relevant1LGlistupdaterevise1.json, ./VP2AS/RelevantLGlistupdaterevise1.json, and ./VP2AS/seedLGlistupdaterevise1.json.

The code and results of conducting a case study in improving topology completeness are included in a subdirectory **./Application**. The details are as follows:

- **./BGPdata**: As a comparison, we construct AS topologies based on BGP data collected from two popular BGP collector projects: RIPE RIS and RouteViews in October 2020.
- **./collectASpath**: We first run ./_init_.py to translate {\em show ip bgp summary} measurement requests into the action of filling in the input fields of the corresponding form element with specific values (see bgpcmd.csv and bgpseedcmd.csv). By automatically issuing {\em show ip bgp summary} measurement requests to each VP, we can obtain each generated webpage that gives the status of every BGP connection with the VP and the ASN and IP address of its neighboring BGP router for each BGP connection (see bgp_resulls.csv and bgpseed_resulls.csv). Then, we extract the hyperlink of each neighbor IP (see bgpneighbor.json and bgpseedneighbor.json), and visit the hyperlink to ask the VP to run show bgp neighbor ip command to collect detailed information about the neighbor IP, including a hyperlink for showing its advertised (or received) routes. By further extracting the hyperlinks (see bgpneighbor1.json and bgpseedneighbor1.json), we send requests to the VP to run the command to collect advertised (or received) BGP routes. The obtained AS links are recorded in bgplink.json and bgpseedlink.json.

./Evaluation

The effectiveness of the two-step classifier has been evaluated in the subdirectory **./classification procedure**. In this subdirectory, we run ./calculate.py to evaluate the effectiveness of the similarity-guided search.