1 OPTIMIZATIONS

This report accompanies the paper **Active Learning of Discriminative Subgraph Patterns for API Misuse Detection**. In this report, we describe optimizations made for ALP's subgraph mining module, which builds on top of the gSpan [5] frequent subgraph mining algorithm. As input, gSpan accepts a collection of graphs and a parameter, *min_sup*, which indicates the minimum support that a frequent subgraph should have. As output, gSpan returns a collection of frequent subgraphs. Mining frequent subgraphs is slow, but gSpan only traverses subgraphs in a canonical space, without explicitly checking for subgraph isomorphism, which is NP-complete and significantly slows down the subgraph mining process.

In ALP, we try to mine discriminative subgraphs. Two filters are used to find discriminative subgraphs from frequent subgraphs.

- Test that a subgraph appears significantly more often in graphs of one label than another
- The CORK criterion [3]

The second filter is used at the end of the subgraph mining process, but the first filter is done during the enumeration of frequent patterns. As such, the first filter enables further heuristics for speeding up the process. In order to use the first filter, we compute four quantities. They are C_H, the support (number of 'hits') of the subgraph among C, examples of correct usage, C_M, the number of graphs in C that does not contain the subgraph ('misses'), and M_H and M_M, similarly defined for M, the examples of misuses.

Using best-case significance Another reason for gSpan's speed is that gSpan [5] employs a heuristic to prune branches during the traversal of the code tree. Branches corresponding to supergraphs are pruned if a subgraph has frequency below the minimum support, *min_sup*, as any supergraph has a lower frequency than its subgraph. Similar to existing subgraph mining algorithms [3], ALP's extends the branch-and-prune approach in the canonical search space to speed up subgraph mining. As our focus is to identify subgraphs that occur more significantly for one label, we compute the upper-bound of significance that a supergraph can have in this particular branch. For any subgraph, the best case is that one of C_H or M_H is maximized while the other is 0. This indicate that the subgraph is a perfect discriminator of the two labels. Within the code tree, as we traverse from a subgraph to its supergraph, a supergraph feature is most informative when one of C_H or M_H reaches 0 while the other (M_H or C_H) is maintained at its current value. As such, we compute the best p-value (as given by the Chi-Square test) that a supergraph can achieve on a traversal of a particular branch. If this score is insignificant, then we prune this branch. This allows us to considerably speed up the subgraph mining process.

Favouring smaller subgraphs Unlike many existing subgraph mining techniques, we favor the selection of smaller subgraphs over longer subgraphs. This decision is motivated by two reasons. Firstly, we invoke Occam's Razor and suggest that finding small and simple patterns, that is more understandable, is in itself a goal [1], Secondly, empirical evidence suggests that API usages are typically short [6]. When traversing from one significant subgraph to its supergraph, if both the supergraph and subgraph have the same level of significance, ALP adds only the subgraph to the feature set. Not only does this prevent the addition of perfectly correlated features, but this enables another heuristic to further prune the search space. Once the traversal has reached a subgraph feature where either C_H or M_H has reached 0, the traversal of this branch will no longer lead to the discovery of new features (any supergraph can do no better than its subgraph at which C_H or M_H already reached 0). Therefore, these branches are pruned to improve performance. in ALP, we only consider subgraph patterns with size up to 6 edges to keep running time reasonable. At its slowest, for our datasets, the subgraph mining process runs for up to 1 hour.

Why not LEAP search? While there are alternatives to gSpan, such as LEAP search [4], that can mine frequent subgraphs quickly, we find that they rely on assumptions that may not be true of API misuses. LEAP search relies on the observation that, often, for a given quality measure, high quality subgraphs are among the most frequent subgraphs. In this study, as we try to mine informative examples beyond the most frequent patterns, it is likely that the subgraph features that we want to discover are not among the most frequent subgraphs.

Why not [2]? The work of Kong et al. [2] assumes that graphs with the same label should be near each other in the vector space. For API misuse, as there are many correct/alternative usage patterns, this assumption is likely to be incorrect. Graphs using a different (but equally correct) usage pattern may be far away from each other in the vector space, therefore, we do not use their framework.

REFERENCES

[1] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1990. Occam's razor. *Readings in machine learning* (1990), 201–204.

[2] Xiangnan Kong, Wei Fan, and Philip S Yu. 2011. Dual active feature and sample selection for graph classification. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 654–662.

[3] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S Yu, Xifeng Yan, and Karsten Borgwardt. 2009. Near-optimal supervised feature selection among frequent subgraphs. In *Proceedings of the 2009 SIAM International Conference on Data Mining*. SIAM, 1076–1087.

[4] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S Yu. 2008. Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 433–444.

[5] Xifeng Yan and Jiawei Han. 2002. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings*. IEEE, 721–724.

[6] Hao Zhong and Hong Mei. 2017. An empirical study on API usages. *IEEE Transactions on Software Engineering* 45, 4 (2017), 319–334.