Graph Processing Algorithms to Enhance Search-Based Software Engineering Tools

Carly Jones
Heinz College of Information Systems
and Public Policy
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890
Email: carlyjon@andrew.cmu.edu

Abstract—**The use of graph representations of static code in the search-based software engineering field presents opportunities for leveraging graph processing algorithms to enhance the effectiveness of emerging tools. This paper 1. Describes the challenges posed by using graphs as inputs to a multi-objective search optimization algorithm, 2. Proposes a method for applying a specific graph processing technique to address this challenge, and 3. Demonstrates the feasibility of applying this technique to a particular codebase representation. The paper concludes with a review of proposed validation and testing activities as well as additional opportunities for leveraging graph processing techniques in this space.**

Key concepts: closeness centrality, artificial intelligence, search-based software engineering, refactoring at scale

I. Introduction

Work in progress by the Architecture, Design, Automation and Analysis (ADAA) group at Carnegie Mellon University's Software Engineering Institute uses artificial intelligence and machine learning to support automated software design detection and code improvements. This area of research is referred to as "AI for SE (Software Engineering)".

The ultimate goal of these efforts is to advance the next generation of automated software tools that can support more complex development and maintenance tasks. The efforts expand upon existing tools in the search-based software engineering space, which generally focus on low-level code changes and do not take software design or developer intent into account.[1]

The Untangling the Knot research project is one of two key efforts led by ADAA which uses artificial intelligence to support the common development task of refactoring. The project can be differentiated from existing approaches[2] in that it allows the user to identify a goal, such as harvesting or replacing specific functionality, and recommends a refactoring sequence that effectively isolates the relevant components.[3]

Building from previous research in search-based software engineering[4], the tool frames the challenge of isolating functionality as a multi-objective optimization problem and uses a genetic search algorithm to identify and apply valid refactorings in sequence.

A. The data model

In pre-processing, a static code analysis tool is used to generate a property graph data structure (termed a "codegraph") representing the codebase. In this representation, nodes are code elements such as classes, fields, and methods, and arcs or "relationships" are the structural relations between them, such as declarations, calls, reads, and writes.[3]

Property graphs are chosen as the data model for this application for several reasons. First, graphs are the preferred and most performant choice for modeling complex, many-to-many relationships between large groups of elements. Second, property graphs allow for additional information to be attached to each node or relationship (as either properties or labels), enabling conditional operations to be made on groups of elements. Third, property graphs are inherently directed, which allows for multi-dimensional representations of dependent relationships.

In the case of Untangling the Knot, additional information about developer intent enriches the graph prior to initialization in ways that constrain and guide search to an appropriate solution. For example, if the user wishes to isolate the components that implement a graphical user interface in order to replace them with something new, the nodes which comprise this functionality are labeled as part of the "candidate set", and the relations which inhibit the replacement operation are captured as "problematic couplings".

The idea of a problematic coupling is central to ADAA's research. Experienced developers intuit that when altering the structure of a codebase, not all dependencies can be treated the same. Untangling the Knot leverages novel research which qualifies these dependencies based on specific development scenarios, which in turn supports the goal of generating high quality solutions to large-scale, complex refactoring tasks.

B. The search algorithm

Untangling the Knot uses a multi-objective evolutionary algorithm (MOEA) modeled after NSGA-II to search for pareto-optimal sets of refactorings that successfully isolate the target components.[4] An MOEA was chosen based on previous successful applications of similar algorithms in the search-based software engineering research field.[5][6][7]

Generally, MOEAs deal with two spaces: the decision variable space and the objective space. In basic terms, the decision variable space represents the set of all possible solution choices, and the objective space is the region comprising all possible objective measurements describing the quality of those solutions.

In the case of Untangling the Knot, the decision variable space is the set of valid refactorings that can be applied to a particular codegraph to remove or otherwise change existing component relationships. The objective space, on the other hand, is the hyperplane of k-1 dimensions (k being the number of objectives) representing all possible combinations of code quality metric values describing the solution graph. Currently, the tool has implemented nine code quality metrics, deemed "fitness functions". Search is configurable so that any combination of these functions may be selected as objectives, such as total number of problematic couplings, lines of code, and total number of component types changed.

The algorithm is designed to minimize all objective values, the most important of which being the total number of problematic couplings. A score of zero for this objective indicates that A) no inhibiting dependencies exist between the candidate set and the rest of the codebase and, B) depending on the scenario, either the harvested component or remaining codebase remains a stable build.

C. Challenges posed by the use of graphs

Untangling the Knot's use of graph data structures as inputs to search poses several challenges not inherent in other MOEA use cases.

Evolutionary algorithms are often applied to problems for which the objective space is non-convex, hilly, disjoint, or otherwise "difficult". This is because evolutionary algorithms employ both informed and random heuristic functions to guide exploration of the search space. Informed heuristics, such as a selection function which calculates the total number of problematic couplings associated with the target node of each available refactoring in the decision variable space in order to choose the one which will result in biggest reduction, support the "greedy" or exploitative pursuit of a locally optimal solution. Random heuristics, such as a selection or mutation function which chooses a refactoring from the decision variable space to be applied at random, enforces broader exploration and prevents search from getting "stuck" in local, but perhaps not global, optima.

The main challenge posed by performing heuristically-guided search over a graph data structure is that a new instance of a graph must be created for every solution. In addition, because sequenced solutions are built over many iterations as refactoring options are applied to each graph, both the constrained set of decision variables and the solution graph change continuously. Both objective functions and additional heuristic functions must be calculated at every iteration for every solution, leading to a large degree of computational complexity.

Ensuring broad exploration of the search space is key to improving the performance of an optimization algorithm. Introducing exploitative heuristic strategies improves convergence time, allowing us to reach a good solution quickly. The challenge, therefore, is to find a mechanism for balancing exploration and exploitation that does not require graph-global computations to be performed for every solution at every iteration during search.

II. Proposed Method

If the goal is to find a method for influencing the progression of search without requiring additional within-search computation, our best and perhaps only option is to generate additional information to assist with the initialization of search.

Currently, search is initialized by selecting a set of refactorings at random to apply to the initial state graph to generate the first "population" of solution graphs. These solution graphs are then processed and additional refactorings are selected and applied according to available heuristics.

However, if a feature which described the overall "connectedness" of each available refactoring were available to inform the initialization process, we could theoretically seed search in a way that would support broader exploration.

A. Wasserman and Faust closeness centrality

To effectively inform search initialization, the proposed feature should accurately represent each target node's accessibility to the rest of the codegraph. Given that this measurement is graph-global, an appropriate graph processing algorithm is needed.

The selected algorithm should calculate a score that is representative of the overall connectedness of each node in the initial state codegraph. In essence, this requires looped computation of the total number of nodes dependent upon each node in the target set of nodes, and on and on until maximum depth is reached or iteration is terminated. The calculation should only consider allowable refactoring options, and sets of nodes need not be mutually exclusive.

Closeness centrality is a common metric for identifying the most influential nodes in a network. Closeness is measured as the average farness (or inverse distance) to all other nodes in a group and is calculated using the following formula:

$$C(u) = \frac{1}{\sum_{v=1}^{n-1} d(u,v)}$$

Where $\mathtt{u}$ is a node, $\mathtt{n}$ is the total number of nodes in a group, and $\mathtt{d}$ is the shortest distance between two nodes $\mathtt{u}$ and $\mathtt{v}$. For each node, the closeness centrality algorithm calculates the sum of it's distances to all other nodes, based on calculating the shortest paths between all pairs of nodes. The resulting sum is then inverted to determine the closeness centrality score for that node. The node with the highest measure of closeness, therefore, is the node which has the shortest distance to all other nodes in the group.[8]

Given that codegraphs may contain disconnected subgraphs, the above formula may overestimate measurements of closeness for elements in these subgraphs. The Wasserman and Faust variation of closeness centrality addresses this issue by normalizing the measurement by the ratio of the number of nodes in a subgraph to the total number of nodes:

$$C_W F(u) = \frac{n-1}{N-1}\left(\frac{1}{\sum_{v=1}^{n-1} d(u,v)}\right)$$

In this variation, `n` is the number of nodes in the subgraph and `N` is the total number of nodes. The result is a score that is more representative of the closeness of nodes to the entire graph, rather than simply its subgraph.[9]

III. Exploration of Proposed Method

To assess the feasibility of calculating Wasserman and Faust closeness centrality for each node in the candidate set to inform search initialization, exploratory analysis was completed using the MissionPlanner open source codebase and Neo4j graph database platform.

A. Setup

MissionPlanner is one of 14 open source codebases used for testing the Untangling the Knot prototype. After instantiating a configured Neo4j database, MissionPlanner nodes and relationships were imported from csv files. The ingested MissionPlanner codegraph is comprised of the following elements:

```
NODES                    [82148]
    Namespaces =            357
    Classes =              4522
    Interfaces =            244
    Structs =               463
    Fields =              27176
    Properties =           9875
    Methods =             34507
    Delegates =             357
    Events =                279
    Enums =                 489
    Files =                3879

RELATIONSHIPS           [589388]
    Calls =              101788
    Reads =              202676
    Writes =              78082
    Inherits =             2227
    Implements =            750
    Locations =           78253
    Declares =            78093
    Type Uses =           47519

CALLS RELATIONS         [101788]
    Methods =            101635
    Classes =                 0
    Fields =                153
```

After creating the codegraph, the candidate set was manually labeled using an imported list of target nodes for the "LoggerD" scenario. "Logger" refers to the MissionPlanner components which implement logging functionality, and "D" indicates that the goal is to replace this functionality with new components.

```
LOAD CSV from 'file:///logger.csv'
    AS candidates
MATCH (n)
WHERE n.longname in candidates
SET n:Candidate
RETURN n.longname, labels(n)
    AS labels
```

Given that LoggerD is a replace scenario, relationships between candidate nodes and nodes in the rest of the codegraph are only problematic if they break the build *outside of the candidate set*. For example, if a method outside of the candidate set calls a method within the candidate set, and that method is removed, the code will not compile. With this context in mind, the following query produces a visualization of the candidate set and all problematic couplings in the initial state codegraph.

```
MATCH p=
    (c:Candidate)<-
    [r:CALLS|IMPLEMENTS|INHERITS_FROM|
   READS|USES_TYPE|WRITES]
    -(n)
RETURN p
```

MissionPlanner LoggerD

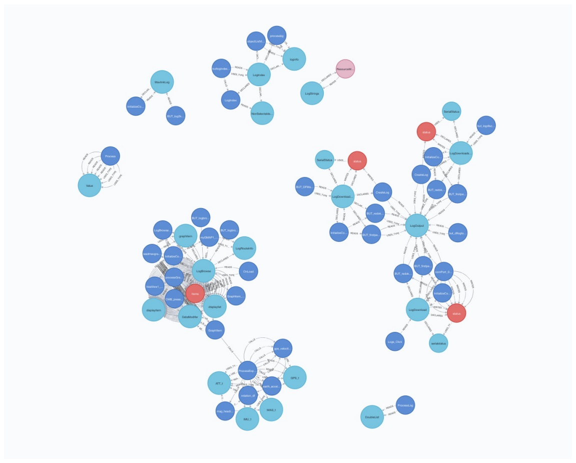

Fig. 1. A visualization of all problematic couplings and their source and target nodes in the initial state MissionPlanner LoggerD codegraph.

B. Running the algorithm

The Neo4j Graph Data Science (GDS) library provides implementations of many of the popular graph processing algorithms, including the Wasserman and Faust variation of closeness centrality.

Calling the GDS closeness function requires a projection of the graph to be replicated in memory for processing. For this projection, all nodes and the subset of relationships of interest (that is, those which indicate problematic couplings) were selected.

The following call produced a table of node names and their corresponding closeness scores, in descending order, which were then exported to a csv file for tabulation and further analysis.

```
CALL gds.alpha.closeness.stream({
     nodeProjection: "*",
     relationshipProjection: [
          "CALLS",
          "IMPLEMENTS",
          "INHERITS_FROM",
          "READS",
          "USES_TYPE",
          "WRITES"],
     improved: True})
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).longname
     AS node.longname, centrality
ORDER BY centrality DESC;
```

TABLE I
THE TEN HIGHEST SCORING CANDIDATE NODES

node.longname	centrality
MissionPlanner.Log.LogBrowse	0.110511
MissionPlanner.Log.LogOutput	0.110424
MissionPlanner.Log.LogBrowse.DataModifer	0.105939
MissionPlanner.Log.LogIndex.loginfo	0.105456
MissionPlanner.Log.MavlinkLog	0.104308
MissionPlanner.Log.MatLab.DoubleList	0.103088
MissionPlanner.Log.LogDownloadMavLink	0.099355
MissionPlanner.Log.LogDownloadscp	0.098839
MissionPlanner.Log.LogDownload	0.098794
MissionPlanner.Log.LogBrowse.displayitem	0.097486

TABLE II
THE TEN LOWEST SCORING CANDIDATE NODES

node.longname	centrality
MissionPlanner.Log.Scan	0.000000
MissionPlanner.Log.MatLabForms	0.000000
MissionPlanner.Log.LogMap	0.000000
MissionPlanner.Log.test.MotorFailure	0.000000
MissionPlanner.Log.LogStrings	0.083034
MissionPlanner.Log.LogIndex.NonSel...	0.084447
MissionPlanner.Log.DFLogScript	0.086919
MissionPlanner.Log.test.MotorFailure.Value	0.089407
MissionPlanner.Log.LogDownload.serialstatus	0.092147
MissionPlanner.Log.LogBrowse.LogRouteInfo	0.092862

C. Analyzing the output

A preliminary review of the Wasserman and Faust closeness centrality scores calculated for each candidate node seems to indicate that these scores could be effective if used to select the most connected candidate nodes to seed the initial population of refactoring solutions.

The scores for the 31 candidate nodes ranged from `0.110511` at the high end to `0.000000` at the low end. Visual analysis of the candidate set and initial state problematic couplings indicates that the two highest scoring nodes, `LogBrowse` and `LogOutput`, are, in fact, the most connected at depth one.

MissionPlanner.Log.LogBrowse

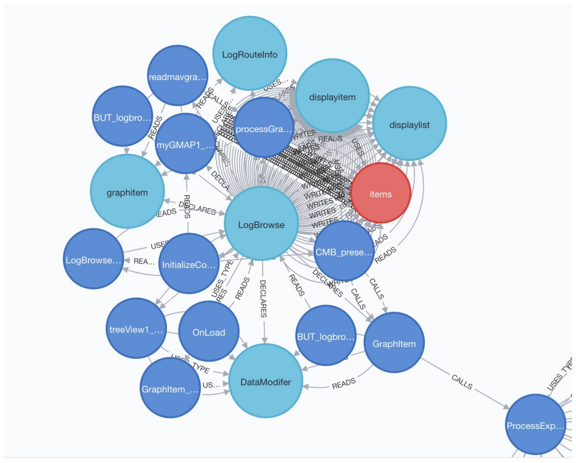

Fig. 2. The LogBrowse candidate node and its problematic couplings.

MissionPlanner.Log.LogOutput

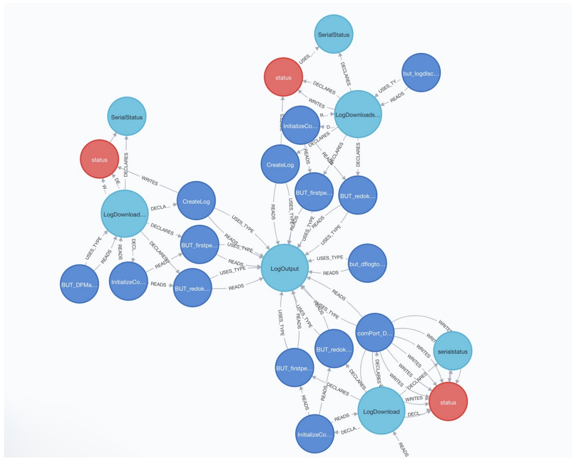

Fig. 3. The LogOutput candidate node and its problematic couplings.

The lowest scoring nodes, on the other hand, appear to be the least connected. Each of the `Scan`, `MatLabForms`, and `LogMap` candidate nodes have only one declaration, each to the `Log` namespace. Similarly, `MotorFailure` only declares the `Log.Test` namespace.

IV. Future Work

A. Validation and testing

The above approach generates closeness centrality scores which can be used rank candidate nodes by how easily they can reach all other nodes in their subgraph, normalized by the size of that subgraph.

To validate the effectiveness of using these scores to seed search, additional controlled validation activities should be conducted.

Ideally, follow-on testing would include controlled experiments comparing the distribution of objective values and convergence time for solutions generated by a set of randomly initialized search runs to the distributions of objective values and convergence time for solutions generated by a search run with biased initialization.

For a set of at least 30 runs each, the metrics of average number of problematic couplings and average search time could be used to evaluate performance and efficiency, respectively.

Additional research could be conducted to examine which initial refactorings are featured in the best solutions generated by random initialization, to compare to those which score highest in closeness centrality.

B. Additional opportunities

Opportunities to use graph processing algorithms to enrich initial state codegraphs is not limited to supporting search initialization. Graph-global analyses could provide answers to important questions about the structure of the underlying static code, such as:

- How uniformly connected is the data? Can we describe the overall distribution of the graph?
- How densely connected is the data? Are most relationships many to many?
- Can the graph be divided into subgraphs? How many are there?

In addition, graph processing algorithms could support feature engineering to make machine learning applications, such as those being developed in ADAA, more effective.

V. Conclusion

Property graph data structures may be the only data structures able to adequately represent the complexity of dependent relationships inherent in large-scale codebases. Although graph representations pose challenges to the efficient use of AI-enabled automation tools, thoughtful application of graph processing algorithms may be able to address these challenges and support complex development tasks at scale.

Acknowledgments

The author would like to thank James Ivers, Lena Pons, and Chris Seifried of the Architecture, Design, Automation and Analysis (ADAA) group.

References

[1] J. Ivers, I. Ozkaya and R. L. Nord, "Can AI Close the Design-Code Abstraction Gap?," 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), San Diego, CA, USA, 2019, pp. 122-125, doi: 10.1109/ASEW.2019.00041.

[2] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. O. Cinnéide, and K. Deb, "On the Use of Many Quality Attributes for Software Refactoring: A Many-objective Search-based Software Engineering Approach," Empir. Softw. Eng., pp. 2503-2545, December 2015.

[3] J. Ivers, I. Ozkaya, R. L. Nord, C. Seifried. 2020. Next Generation Automated Software Evolution: Refactoring at Scale. In Proceedings of the ACM 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20). ACM, November 8-13, 2020 Virtual Event, USA. https://doi.org/10.1145/3368089.3417042

[4] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, April 2002, doi: 10.1109/4235.996017.

[5] M. Harman and L. Tratt: Pareto Optimal Search Based Refactoring at the Design Level. GECCO 2007, pp. 1106-1113, 2007.

[6] M. Abebe and C.-J. Yoo: Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review. Int. J. Softw. Eng. Appl. 8(6), pp. 299–318, 2014.

[7] T. Mariani and S. R. Vergilio: A Systematic Review on Search-based Refactoring. Information Software Technology 83, pp. 14-34, March 2017.

[8] M. Needham and A. E. Hodler. *Graph Algorithms*. Sebastopol, CA: O'Reilly Media. 2019.

[9] S. Wasserman, and K. Faust. Social Network Analysis: Methods and Applications. Cambridge; Cambridge University Press, 1994.