# Step 1 Report

Demonstrate what Jigsaw does and how to use it, at least to the extent that is pertinent for this project. The higher-order part of Jigsaw must be used. The following explanation is about what jigsaw does (based on my understanding) to the extend that is useful for our purpose:

1. Jigsaw takes ASTs of two input seeds (copy& paste seeds)
2. Jigsaw determines all possible candidate correspondence between AST nodes in the two input seeds.
   1. Each node in the AST of input seeds maintains a list of its correspondence connections or elements, in which each connection in the list represents an anti-unifier.
   2. Candidate correspondences between AST nodes are stored in an augmented form of AST, which is called the CAST.
   3. Each CAST could represent multiple anti-unifiers since every node of it holds a list of candidate correspondences. However, we should construct one single anti-unifier that is helpful to solve our problem; thus, our algorithm should select an anti-unifier of the best fit to our application. In general, there is no unique optimal fit, but our desire is to find one of the best fits to approximate the optimal one that can sufficiently solve our problem.
   4. Jigsaw computes structural correspondence between ASTNodes (~~it compares nodes of identical types~~) to determine how much similar they are. If it is above zero a correspondence connection will be created between these AST nodes and their similarity value will be recorded in that connection of the CAST. Jigsaw compares nodes of identical types or nodes that does not have identical types but have similar semantical meaning.
   5. Jigsaw applies a set of equivalence equations on higher-order extended structures to indicate which structures can be compared.
   6. ~~Jigsaw uses equivalence equations to denote~~ ~~the structures that are semantically similar.~~
3. Since there could be many possible anti-unifiers, Jigsaw tries to filter them to recognize the anti-unifiers of the best fit by defining three different threshold values.
4. Select a set of Java classes that use logging, say 10 of them, and use standard tooling to extract their ASTs (Eclipse JDT).

The sample set of Java classes that contains logging calls (10 examples from JEdit 4.2 pre15) is mentioned in the LoggingClasses.txt file. Our plug-in uses the Eclipse JDT to extract ASTs of these classes.

b.

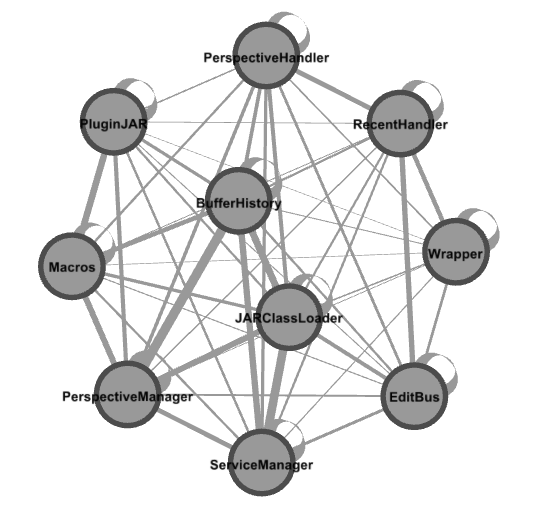
* Use Jigsaw to determine the correspondences between these in a pairwise manner (55 cases in total, including self-comparisons). Measure the Jigsaw similarity in each of these cases.
* Use some sort of graphing tool (like Graphviz) to visualize the results so that these do not need to be hand-drawn incorrectly.

For every comparison, our Plug-in gives a pair of ASTNodes of these Java classes as input seeds to jigsaw. The structural correspondence between the ASTs of Java classes, calculated by Jigsaw, in a pairwise manner is brought in Comparisions.txt file for 55 comparisons. The results are visualized in Figure 1.

Problem: I investigate whether the similarity, computed by jigsaw, between ASTs of the Java Classes is appropriate for our application or not. I came to the conclusion that it is not since what jigsaw considers for extracting similarity of type declaration ASTNodes is on the basis of their names and type hierarchies, and also for method declaration nodes it only considers method signatures; Jigsaw does not check the body of classes and methods containing logging calls for calculating similarities. However, the body and context of these classes are important for our purpose since we want to describe the location the logging classes inside the classes.

Suggested solution: I think we have to re-calculate the similarity based on the body of classes and methods containing log statements and also the location of log statements, and then we can decide to discard the anti-unifiers if the similarity between two logged structures is below a pre-determined threshold. I believe that the best anti-unifier can be constructed in a button-top approach starting from the first block containing log statement.

Each node in the CAST of an input seed maintains a list of its corresponding connections (if there is any). Each element in a node’s list represents an anti-unifier. We can choose one anti-unifier for each node inside the block that is the best fit for our purpose. Our choice can be based on the location of corresponding node compared to the location of log statement inside the seed (before or after the log statement and being at the same level as the selected node) and the highest similarity value compared to the other options (to choose the best fit). Then we can pursue this approach in a bottom-up approach until reaching to the root node. (I have to think more about it to propose an algorithm for step 2)



*Figure 1. Visualizing the comparison results using Gephi. Nodes are the Java Classes ASTNodes; edge thicknesses are based on the value of similarity, extracted from Jigsaw, between these ASTNodes.*

* Note that the logging calls will have non-zero similarities with other elements that are not logging calls.

Yes, as it is shown in node1.txt, it has non-zero similarity with other MethodInvocation nodes (In node1.txt, two logged java Classes are given as input seeds to jigsaw, and then we extract the list of correspondence connections that it produces for the log method invocation node and the value of similarity for each element). However, similarity value is higher for elements that their paste nodes are log method invocations as well (0.46, 0.51, 0.61, 0.7) compared to the other MethodInvocation nodes.

* For an interesting subset (say 3 cases), examine the CAST that it produces, noting the choice points and where the correspondence is fully specified.

Node2.txt file shows generated corresponding elements in the CAST for the following node from a pair of input seeds of Java classes selected from the sample set:

for (int i=0; i < classes.length; i++) {

classHash.put(classes[i],this);

}

As you can see, Jigsaw detects correspondence between for statement with both for statements and while statements in the paste seed (it detects that these two structures are semantically similar); however, the for statement node has higher similarity with a for statement node in the paste seed (0.86) compared to while statements.

Node3.txt has shown generated corresponding elements in the CAST for the following InfixExpression AST node:

action == null

As you can see, this node has the highest similarity value (0.72) with “ settingsDirectory == null” node, which is more similar to the selected node compared to the other corresponding InfixExpression nodes, meaning that similarity measurement extracting from Jigsaw makes intuitive sense.

* Check that an AST that is compared with another AST that is

utterly dissimilar has a similarity of 0, if such a scenario is practical.

This scenario is not practical since Jigsaw compares nodes of identical type to indicate how similar they are. For example, I tried to extract similarity between the following ASTNodes:

Node1 : return entry;

Node2 : int i=0;

Since the node’s types are different and they are not even detected semantically similar using equivalence theories (similarity is not above zero), not any correspondence connections are created for these nodes. We can conclude that the similarity value is zero, thus no connection is created in the CAST.

* Check that an AST that is compared with itself has a similarity of 1.

The sample set contains 10 Java classes, thus we have 10 self-comparisons. As it is shown in Comparison.txt, the similarity values for these comparisons are 1, meaning that the similarity extracted from Jigsaw ranges from 0 to 1.