# Plan

1. 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. [<1 week]  
  
a. Select a set of Java classes that use logging, say 10 of them, and use standard tooling to extract their ASTs (Eclipse JDT). *[RW: I recall that you did this, but I see no sign of it in the thesis draft.  The details are in the Step1 directory, no?  This can be added to the Background section on ASTs, with the full details in an appendix.]*  
  
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. For an  
interesting subset (say 3 cases), examine the CAST that it produces, noting the choice points and  
where the correspondence is fully specified. Note that the logging calls will have non-zero  
similarities with other elements that are not logging calls. Check that an AST that is compared  
with itself has a similarity of 1. 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. Check that the similarity  
measure makes intuitive sense. Use some sort of graphing tool (like Graphviz) to visualize the  
results so that these do not need to be hand-drawn incorrectly. *[RW: Same comments.]*  
  
2. Determine an algorithm for constructing an anti-unifier from a CAST that pays special attention to logging calls. That is to say, a logging call should either anti-unify with another logging call or should anti-unify with nothing. But the CAST will have other options in there that have to be ignored. Furthermore, if the anti-unifier has eliminated all context around the logging call, this indicates that there is no commonality in that context, and so no pattern is to be found between those two inputs; as a result, this particular antiunifier should be discarded. [<3 weeks, because it is not very different than what has been done before] *[RW: I believe that you have done this in Ch4.]*  
  
3. Implement the algorithm from 2, building atop Jigsaw. Test and debug using the Java classes from 1a. Measure the test coverage of the code, and add additional (artificial) cases to cover missing lines. [about 1 month, because again, it is not that much of a stretch from what has come before] *[RW: While everything you have told me suggests that you must have done this, it is not obvious. I would suggest having a separate section describing the implementation and a demonstration that it works on the examples from 1, measurements of test coverage, etc. This is different from the formal evaluation in Ch5.* ***THIS IS IMPORTANT.****]*  
  
4. Determine an algorithm for anti-unifying a set of ASTs. If the previous steps have been done correctly, this ought to simply be a matter of choosing the right order in which to anti-unify pairs. This will require a bit of research into clustering algorithms; lean on Dr. Denzinger for pointers here. The goal is not to become an expert in clustering algorithms. Depending on how these have to work, it might be possible to use some standard implementation inputting the similarity measurements extracted from Jigsaw, rather than reimplementing anything. I will refine this point in the near future in consultation with Dr. Denzinger. [1 month] *[RW: It’s not clear to me if you have been working towards this.]*  
  
5. Implement the algorithm(s) from 4 atop the Step 3 extension of Jigsaw. Test them on the ASTs from 1a. [2 weeks, probably less] *[RW: It’s not clear to me if you have been working towards this.]*  
  
6. Select at least three full systems that make use of logging. Ideally, these should not be the same systems from which the samples were drawn for Step 1, for the sake of generalization, and they should have very little in common otherwise, again in attempt at generalization. All the classes from each of these systems should be extracted. These classes can be anti-unified via the tooling of Step 5 to determine the patterns on a per-system class-granularity basis. [1 month] *[RW: Based on discussions, it sounds like you have been working towards this, but it is not so obvious.]*  
  
  
[Although a k-fold cross-validation procedure would appear to be a reasonable step at this point, I am unconvinced of its value here: what will come out of Step 6 are not inferred rules about when we ought to expect the presence of a logging call, but rather an abstract characterization of where logging calls actually do occur. There is no inference going on, and so the characterization will be right or wrong. While this could be extended to be interpreted as rules for when we would expect to see logging, I don’t see that as a necessary step here nor necessarily a useful one. You need to be careful to not refer to these as rules.]