1. Implement the DST congruence closure algorithm described in lecture 5. You will probably want to make use of the partial function and partition data types we discussed in class (the examples we did in class are posted on the web in the OCaml directory).

Note that the “equate” function used to implement “union” in lib.ml doesn’t quite match the semantics of union in DST. hw2.ml (available from the course website) contains a new function, “equate1”, which matches the semantics of union in DST.

The rest of hw2.ml contains code for creating and testing congruence closure examples. The function “cc1” runs Harrison’s implementation of the basic congruence closure algorithm and gives the result along with the elapsed time.

Your goal should be to create a new version (maybe “cc2”) which does the same thing but runs much faster. I suggest you use the examples in hw2.ml to test your algorithm. The “buildex” function is useful for creating a wide variety of test examples (try lots of values for \(i, j, k\) and make sure your results match the result of cc1).

You may also want to consult the original paper, “Variations on the Common Subexpression Problem.” The citation is at the beginning of lecture 5.

2. Use the “buildex” function to test the performance of a suite of examples using cc1 and cc2. The suite should leave \(j\) and \(k\) fixed at 47 and 29 respectively while varying the value of \(i\).

What can you say about the relative performance of the two algorithms as a function of \(i\)?

3. One important problem in propositional satisfiability solving is that of finding minimal unsatisfiable cores. Given an unsatisfiable set of propositional clauses \(C\), an unsatisfiable core is a set \(C' \subseteq C\) that is still unsatisfiable; a minimal unsatisfiable core is an unsatisfiable core \(C'\) such that any proper subset of \(C'\) is satisfiable. Naturally, such a minimal clause set is only a local minimum.

A simple way to construct a minimal unsatisfiable core is simply to remove clauses, testing satisfiability at each step. More sophisticated methods analyze the set of clauses to determine a cause of unsatisfiability. (In first-order logic, the situation is more complicated, as discussed in class; however, when we consider conjunctions of theory terms, as in “buildex”, we can do something analogous: we can give a short explanation of the inconsistency. This has important applications that we’ll see later in the semester.)

- In the theory of equality, say we have sets of constraints \(\Gamma\) and \(\Delta\) such that they are inconsistent under congruence closure. Write an OCaml function to minimize these sets of constraints, finding a minimal inconsistent subset, by making calls to a congruence closure algorithm.

- Think about your implementation of the DST algorithm and what is necessary to analyze inconsistency. What can be done in DST to analyze a result of inconsistent? Modify your implementation of DST to be a bit more clever than the simple approach, so that in the case of inconsistency it returns new sets \(\Gamma' \subseteq \Gamma\) and \(\Delta' \subseteq \Delta\), such that \((\Gamma', \Delta')\) is an inconsistent subset of \((\Gamma, \Delta)\) (it need not necessarily be minimal). You may want to keep additional data structure(s). Can you find better inconsistent “cores” using this approach? Can you find them faster?