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. I have written a new function, “equate1” which matches the semantics of union in DST. This function is the first function in hw2.ml.

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 \(i, j,\) and \(k\) 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 \(i\)?