Lab 6: Delta

Fall Semester 2021

Due: 22 November, 8:00 a.m. Eastern Time

Corresponding Lecture: Lesson 10 (Delta Debugging)

Objective

In this lab, you will implement the Delta Debugging algorithm using Java 1.8 (already installed on class VM) to find 1-minimal failing test cases for buggy text encoder programs using both line and character granularity.

Resources

1. Delta Debugging webpage: http://www.st.cs.uni-sb.de/dd/

Caution

Dr. Zeller, the author of the Delta Debugging technique, and several other people have published implementations of the Delta Debugging algorithm. While we encourage you to research and read more about the algorithm, looking at someone else's implementation, will almost certainly cause your implementation to be similar to theirs. Therefore, make sure that you do not view or reference any implementations or code of the algorithm.

Setup

1. Download `delta.zip` from Canvas and decompress it. It will produce the directory `delta` where you should find:
    - `DeltaDebug.java` - skeleton file for you to implement your algorithm in
    - `DeltaDebugTest.java` - Program to run and test your program
    - `SecretCoder` - first buggy encoder program
    - `MessageCoder` - second buggy encoder program
    - `long_failing_text.txt` - an input text file on which each buggy encoder program fails.
    - A folder called `expected_output` containing the following:
        - `[prg]_min_failing_text_line.txt`: the correct minimal test input that should be produced by your Delta Debugging program on `SecretCoder` ("sc") or `MessageCoder` ("mc") for part 1 (line granularity)
        - `sc_min_failing_text_char.txt`: the correct minimal test input that should be produced by your Delta Debugging program on `SecretCoder` for part 2.

2. Each time you change the `DeltaDebug.java` and `DeltaDebugTest.java` files, you need to compile the program with:
   ```
   javac *.java
   ```
3. After compiling the files, you may execute the `DeltaDebugTest` program as follows:
   ```
   java DeltaDebugTest
   ```

Part 1: Line Granularity

Program Description

`SecretCoder` and `MessageCoder` implement a Caesar (or shift) cipher with a right shift of 98 places. Instead of the regular alphabet, these programs are intended to encode the 128 characters in the ASCII character set (value 0 to 127). When the programs run successfully, they will print out a message that indicates the program executed successfully.

The programs can be executed directly in the command line, followed by a single input that is the path to the input file, as shown below:

```
./SecretCoder <input file>
```

Note: In the VM, you may need to change the access mode of the executables in order to run them:

```
chmod +x <PROGRAM(s)>
```

Problem Summary

The bug in `SecretCoder` occurs when `long_failing_text.txt` contains a non-ASCII character. The program is using the UTF-8 character set to determine the number of characters in the file and creates a char array to store the encoded values for printing. It does not take into account that non-ASCII characters in the UTF-8 character set could be included into the file and these will take up more than one byte. When a non-ASCII character is read into the file as two or more bytes, the char array runs out of room and throws a `java.lang.ArrayIndexOutOfBoundsException`.

`MessageCoder` is able to handle all characters, but the bug in the encoding program `MessageCoder` occurs when `long_failing_text.txt` contains a specific pattern: a four-letter pattern of alternating caps starting with a capital letter (e.g. “JaVa”, or “HaHa”). When this pattern is detected, the following exception occurs: `java.lang.IllegalArgumentException`

Your task is to implement the delta debugging algorithm presented in the lecture to obtain the minimal test case on which `SecretCoder` fails for `java.lang.ArrayIndexOutOfBoundsException` and the minimal test case on which `MessageCoder` fails for `java.lang.IllegalArgumentException`. In your implementation, do not hardcode the program names and error messages. Your program should be able to run for a program of any name and any error message.

To accomplish this task, we have provided a framework and you will need to implement the `deltaDebug` method in `DeltaDebug.java`. For part one of the lab, we will use line granularity. We have provided a `DeltaDebugTest` program that you can use to run `DeltaDebug` to test your changes.

Note that `DeltaDebugTest` will call the `deltaDebug` method in `DeltaDebug.java`, which requires 5 parameters:

```
deltaDebug(Boolean char_granularity, String program,
     String failing_file, String error_msg,
     String final_minimized_file)
```

- char_granularity - If false, implement line granularity only
- program - this is the path to the program being tested
- failing_file - this is the path to the file with the large failing test case
- error_msg - this is the error message that will be used to determine if the program is failing for the same reason as the original file or not
- final_minimized_file - this is the name of the final file that your program should print with the minimized input.

Similarly, the grading script will call the `deltaDebug` method directly in `DeltaDebug.java`. **Therefore, it is very important that you do not make any changes to the type or method signature of `deltaDebug` in `DeltaDebug.java`.**

Ensure you are using the *failing_file* and *final_minimized_file* parameters for your file names in `DeltaDebug.java` instead of hardcoding files because the grader will be using additional files besides the ones provided in this lab.

We have provided an outline for some helper methods in `DeltaDebug.java` that you may find useful (there are comments in the code to explain what these do). You can modify these methods or add new helper methods in `DeltaDebug.java` in any way you see fit as long as your algorithm is invoked by calling `deltaDebug` with the original parameters. As a warning, since there are test cases looking for exact text matches, it is in your best interest to keep changes to the `writeToFile()` method minimal since the method appends a new line at the end of the file by design; the grader will be expecting this new line.

If the `deltaDebug` is written correctly, executing the test program will create a 1-minimal test case text file (named by your `final_minimized_file` variable). The contents should be identical to the provided expected `[prg]_min_failing_text_line.txt` when running the algorithm with line-granularity.

You may verify the files are identical (*including whitespace & new lines*) using the diff command (replace 'prg' with 'mc' or 'sc' depending on which files you are comparing):

```
diff my_min_failing_text_line.txt [prg]_min_failing_text_line.txt
```

For grading, we will also be testing your algorithm on hidden input files. For this section, we are looking for an exact text match since the test cases will have at most 1 line with a failing character or pattern. The hidden test cases will be similar in format to the provided one, but will have a different file name, different number of lines, different failing character or pattern, different position for failing character or pattern, or other changes appropriate for a text file. You can assume that there will be a single failing character or pattern in the file. We encourage you to come up with different inputs to test how your algorithm handles different cases.

Your algorithm should run in less than 3 minutes on the course VM.

If you add print statements for debugging, make sure to remove these before submission.

Part 2: Character Granularity

Now you will extend your algorithm to perform minimization at the character granularity level when the flag is set. Make sure to minimize the file by lines as much as possible, and then switch to character granularity when line granularity has reached its minimum.

If the `deltaDebug` is written correctly, executing the test program will create a 1-minimal test case text file (named by your `final_minimized_file` variable). The 1-minimal failing test case for `SecretCoder` should be identical to the provided `sc_min_failing_text_char.txt`, and you can verify with `diff`. Due to the nature of the failure in `SecretCoder`, we are expecting an exact text match.

The 1-minimal failing test case for `MessageCoder` will vary depending on the partition strategy implemented in your method, so there will not be an exact text matching for this portion. The grading criteria for these test cases is as follows:

- The final minimized text must be a *subsequence* of your output from the line-granularity of `MessageCoder`, or else no credit. This means if your line-granularity was "abcdefgh", your char-granularity can be "acg", but it cannot be "gah". The characters must be found in the same order as it appears in the original string. The algorithm for delta handles this implicitly, so if your output is not a proper subsequence, something is wrong with your algorithm.

- The char-granularity output must also fail when passed into the buggy encoder program, or else no credit. If you run the `MessageCoder` program with your char-granularity output, you should still get the `java.lang.IllegalArgumentException` failure. This means that at minimum, there should be some upper-lower-upper-lower pattern in your output.
- The output is 1-minimal, which means that you cannot refine your granularity any further. Removal of any single character will result in a non-failing input.

If your output meets all three criteria, full credit for that test case will be awarded. Partial credit will be awarded if and only if you meet the first 2 criteria but not the last criteria.

The same grading notes and 3 minute timeout limit apply as in part 1.

Items to Submit

Submit the following file into Gradescope.

- `DeltaDebug.java`

Make sure the spelling of the filenames and the extensions are exactly as indicated above. Misspellings in filenames may result in a deduction to your grade. Also double-check that you are not accidentally submitting `DeltaDebugTest.java`.

Grading

Part 1:

- 8% of grade: For `SecretCoder`, `DeltaDebug.java` correctly minimizes `long_failing_text.txt` to `sc_min_failing_test_case_line.txt`
- 12% of grade: For `MessageCoder`, `DeltaDebug.java` correctly minimizes `long_failing_text.txt` to `mc_min_failing_test_case_line.txt`
- 30% of grade: Other test cases

Part 2:

- 8% of grade: For `SecretCoder`, `DeltaDebug.java` correctly minimizes `long_failing_text.txt` to `sc_min_failing_test_case_char.txt`
- 12% of grade: For `MessageCoder`, `DeltaDebug.java` correctly minimizes `long_failing_text.txt` to some reduced failing text.
- 30% of grade: Other test cases

How Do We Use It?

Delta Debugging is an expanded binary search. There are multiple implementations available for most common programming languages. Researchers have even extended the technique into test case minimization. Even without implementing the technique in code, it can be used to help quickly diagnose problems with failing inputs in cases where the error messages available do not provide sufficient information for isolating issues. For example, it is possible to debug a program that reads in multiple records and does not give status on how many succeeded when it fails by using a Delta Debugging-influenced technique to isolate a failing record.

Note: Every semester, we have students asking if they can implement binary search: the answer is that you *can* implement it for the line granularity portion of this lab, but you will not be able to use binary search for char granularity. Ultimately, it is up to you whether you want to implement only *one* algorithm that will work for both parts, or *two* separate algorithms.