SBSE report

Tony Dinh [a1726218@student.adelaide.edu.au](mailto:a1726218@student.adelaide.edu.au)

Task 1:

Mitch:

Tony: I completed Task 1 and Task 2

Task 2:

GIN is a genetic algorithm that is designed to\ optimize and improve the efficient of the program passed as an argument. The fitness function used by this genetic algorithm is to optimize program’s speed. The aim of this experiment is to apply the genetic algorithm GIN to four functions SortBubbleDouble, SortBubbleLoops, SortInsertion and SortCocktail and evaluate their performance upon improvements by the tool. We will compile and run 15 iterations of each of the four sorting functions. The best execution speed from each iteration will be recorded and graphed, this will be completed for all four sorting functions. An evaluation of effectiveness and practicality of the patches will be supplied. Below are the graphs of results obtained from the best execution for all four functions.

The methodology used to obtained the results was done by first downloading the gin-master.tgz folder. Gradle 3.3 and JDK 1.8 was downloaded according to the README file. In the terminal, the build file was compiled by running the command “gradle build”. When built, to obtain the best execution speed the following command was ran inside gin-master folder “java -jar build/gin.jar examples/locoGP/<name of program>”, where name of program is the name of functions with filename .java at the end. The program will then compile and applies numerous patches to the function to try and optimize it. Upon completion, the program will state where the best patch was found, best execution time and achieved speedup. Fifteen iterations of the described method were executed for all four sorting functions, during which the best patches applied and corresponding execution times were recorded. The best execution speeds of each function were graphed separately, the produced results can be seen below.

Fig 1. Optimized functions best execution speeds

From a surface level we can see that the genetic algorithm produces low variance results, having similar best execution speeds. However, the flaw of the GIN genetic algorithm lies in the details. The best patches of each algorithm were recorded and observed. Upon observation we can see that some best patches applied made the algorithm efficient in terms of speed but were not programmable correct. We can see this in four separate cases, the first case is the function Sort Bubble Double the initial nested loop was kept the same while the second nested loop did not contain the if statement inside. The code still compiled and was more efficient than the original algorithm but the empty nested for loop was redundant and computed nothing. In a similar fashion for the Sort Bubble Loops, the first loop in the three nested loop was moved outside the nested loop, creating a redundant for loop. The algorithm still compiled and passed all test cases and provided the best execution speed. In case of the Sort Insertion there was a redundant assignment a[j] = index. Finally, in the Sort Cocktail function the best patch copied the swapped = false line and subsequent for loop and added later into algorithm creating redundant code. The reason for this is due to the fitness function. The fitness function of the Genetic Algorithm GIN only accounts for the execution speed of the optimized function it is not trying to optimized how the correct the code is and as such as long as it provides the best execution speed the correctness of the code is irrelevant.

Task 3:

The aim of task 3 is to analyze the patches that were used to provide the best time execution. Upon each execution of the genetic algorithm, the best final patch was recorded and assessed. The benchmark program chosen for this part was the Sort Cocktail function. The benchmark provides patches more than one edit long. By running this benchmark and recording the patches we can analysis the effect of each patch on multiple runs.

A similar methodology to above was conducted to obtained the results. In the terminal, inside the gin-master folder the command “java -jar build/gin.jar examples/locoGP/SortCocktail.java” was ran and the upon completion of computation the best patches were recorded and analyzed. The program was ran 5 times and 3 unique patches were used to achieve the best speed execution.

1. Best patch found: | MOVE 16 -> 4:0 | 7078500.0(ns)
2. Best patch found: | MOVE 16 -> 4:0 | MOVE 21 -> 3:0 | 7354000.0(ns)
3. Best patch found: | COPY 4 -> 0:2 | MOVE 7 -> 2:0 | COPY 8 -> 2:0 | 6986100 (ns)
4. Best patch found: | MOVE 5 -> 2:0 |\*\*\* New best \*\*\* Time: 7854700 (ns)
5. Best patch found: | DEL 7 | COPY 12 -> 0:0 |\*\*\* New best \*\*\* Time: 8487500.0(ns)

|  |  |  |  |
| --- | --- | --- | --- |
| Edits | MOVE | DEL | COPY |
|  | 5 | 1 | 2 |

Fig 2. Most common patch edits

From these multiple runs we see that the MOVE edit appears most often, while COPY was 2nd most and last was DEL. Many of the positions from the bottom of the program is moved up to the top of the program. An example of this we can from best patch found: | COPY 4 -> 0:2 | MOVE 7 -> 2:0 | COPY 8 -> 2:0 produced the algorithm below:

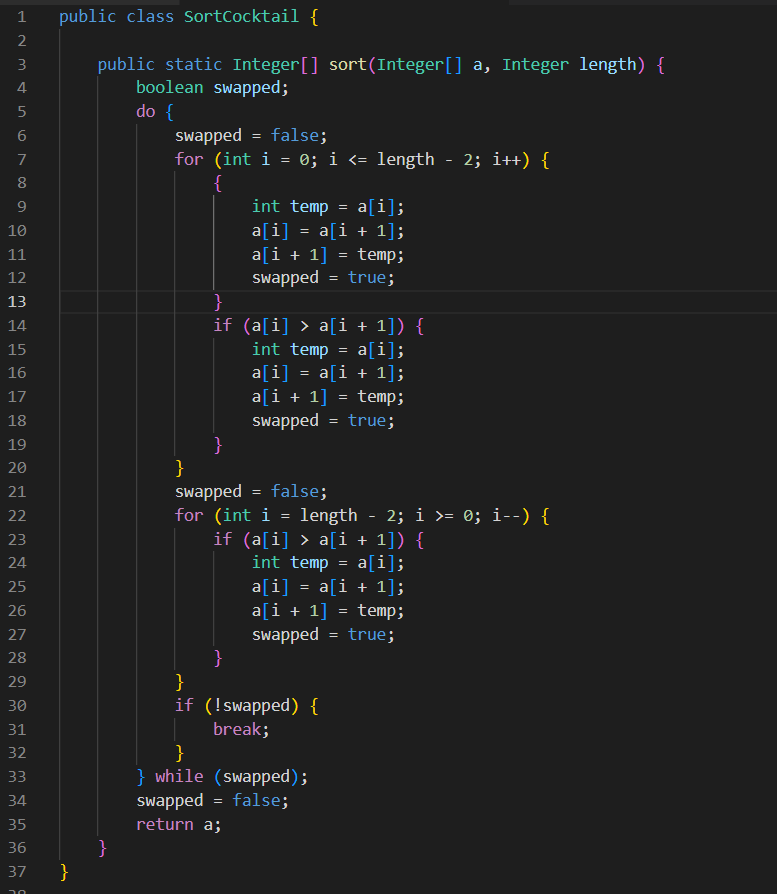


Fig 4. Optimized Sort Cocktail Algorithm using best patch 3

From what can be seen from the other patches it can be seen that edits do not have the same effect as the edits in other patches. From my observation it seems that the genetic algorithm is favoring a move edit over delete and copy edits. A shorten patch that can be observed when assessing patch 3 is to remove the last copy edit and only have the initial copy and move. The new batch can be seen as batch : |COPY 4 -> 0:2 | MOVE 7 -> 0:2 |. Which will the produce the algorithm below.

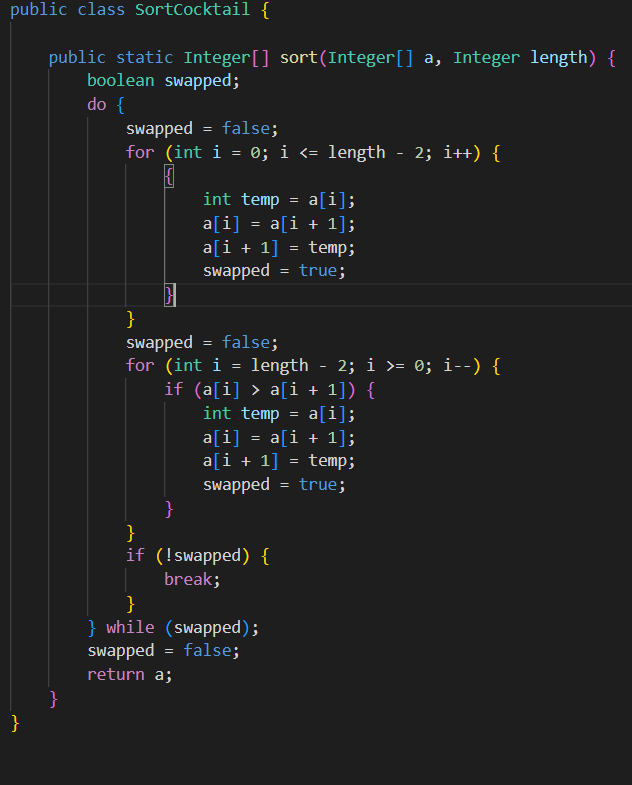


Fig 5. Optimized Sort Cocktail using shorten best patch

Resulting in a more optimized algorithm then compared to the previous patch, where there was a redundant swap statement, from the last copy statement.

Part 4:

Code changes:

The code changes for part 4 aim to alter the way that the search function from gin works. During the original run of the search function, we keep track of any patches that compiled and passed all tests. After the original run we look at the best patch that the program found, this has information relating to the changes that the patch makes. We then go through the list of patches that passed and find the first one that has no changes compared to the best patch, this is our first example of the best patch. The code that reflects this is given in the appendix.



Table X: Result from running the updated search with examples/locoGP/SortInsertion.java

The above table shows the results from running the updated gin search program. This data was ran for ten tests and the best patch, along with when the best patch was found was recorded, along with the equivalent patch, and when that was found. As can be seen from the table above, often the exact patch, or an equivalent was found a lot earlier than when the best patch was found.

Part 5:

From my understanding, we need to create a benchmark program that is inefficient. We then run GIN against this benchmark program to improves it efficiency. We can alter a couple of lines in the program to make it efficient and see if GIN can pick up on it. In the report we write where the inefficiency comes from and run against GIN and presents the results we obtain.

Appendix:

Code changes for part 4:

The Search() function within the search return class becomes:

    private SearchReturn search() {

        // start with the empty patch

        Patch bestPatch = new Patch(sourceFile);

        double bestTime = 10000000; // testRunner.test(bestPatch, WARMUP\_REPS).executionTime;

        double origTime = bestTime;

        int bestStep = 0;

        ArrayList<PatchWIthIndex> passingPatches = new ArrayList<>();

        PatchWIthIndex betterPatch;

        System.out.println("Initial execution time: " + bestTime + " (ns) \n");

        for (int step = 1; step <= NUM\_STEPS; step++) {

            System.out.print("Step " + step + " ");

            Patch neighbour = neighbour(bestPatch, rng);

            // System.out.print(neighbour);

            TestRunner.TestResult testResult = testRunner.test(neighbour);

            if (!testResult.patchSuccess) {

                System.out.println("Patch invalid");

                // failedChanges.add(neighbour.toString());

            }

            if (!testResult.compiled) {

                System.out.println("Failed to compile");

                // failedChanges.add(neighbour.toString());

                continue;

            }

            if (!testResult.junitResult.wasSuccessful()) {

                System.out.println("Failed to pass all tests");

                // failedChanges.add(neighbour.toString());

                continue;

            }

            passingPatches.add(new PatchWIthIndex(step, neighbour));

            if (testResult.executionTime < bestTime) {

                bestPatch = neighbour;

                bestTime = testResult.executionTime;

                bestStep = step;

                System.out.println("\*\*\* New best \*\*\* Time: " + bestTime + "(ns)");

                // failedChanges.clear();

            } else {

                System.out.println("Time: " + testResult.executionTime);

            }

        }

        System.out.println("\nBest patch found: " + bestPatch);

        System.out.println("Found at step: " + bestStep);

        System.out.println("Best execution time: " + bestTime + " (ns) ");

        System.out.println("Speedup (%): " + (origTime - bestTime)/origTime);

        bestPatch.writePatchedSourceToFile(sourceFile.getFilename() + ".optimised");

        SourceFile checkFile = bestPatch.apply();

        betterPatch = new PatchWIthIndex(0, bestPatch);

        System.out.println("print is equal working" + bestPatch.apply().isEqual(bestPatch.apply()));

        for (PatchWIthIndex str : passingPatches) {

            if (str.patch.apply().isEqual(checkFile)) {

                betterPatch = str;

                break;

            }

        }

        return new SearchReturn(betterPatch, bestPatch);

    }

Added is equal function to SourceFile.java, this compares two files so we know if the sourcefile is the same

    public boolean isEqual(SourceFile sf2) {

        boolean ret = (this.filename == null ? sf2.filename == null : this.filename.equals(sf2.filename))

            && this.getNumberOfBlocks() == sf2.getNumberOfBlocks()

            && this.getStatementCount() == sf2.getStatementCount()

            && (this.blockList() == null ? sf2.blockList() == null : this.blockList().equals(sf2.blockList()))

            && (this.statementList() == null ? sf2.statementList() == null : this.statementList().equals(sf2.statementList()));

        return ret;

    }

These were helped by two simple data structures that were created to store the data efficiently:

package gin;

public class PatchWIthIndex {

    public int index;

    public Patch patch;

    // Constructor

    public PatchWIthIndex(int i, Patch p) {

        index = i;

        patch = p;

    }

}

package gin;

public class SearchReturn {

    public PatchWIthIndex betterPatch;

    public Patch patch;

    // Constructor

    public SearchReturn(PatchWIthIndex ss, Patch p) {

        betterPatch = ss;

        patch = p;

    }

}