SBSE Assignment 3

Investigating genetic improvement using GIN

Mitchell Follett a1770751

Tony Dinh a1726218

## Task 1:

Mitch: I completed all code changes and analysis for tasks 4 and 5, as well as setting up the Git and editing and formatting the report

Tony: I completed Task 2, Task 3, proof read and final edits.

## Task 2:

GIN is a genetic algorithm that is designed to optimize and improve the efficiency of the program passed as an argument. The fitness function used by this genetic algorithm is to optimize the 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 obtain the results was setup by first downloading the gin-master.tgz folder from the assignment sheet. Gradle 3.3 and JDK 1.8 were used in all compilation and running in this assignment, in accordance 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 run, and numerous patches are applied to the function in an attempt at optimization. Upon completion, the output 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 were 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 obtain 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 5 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 |
| 0 | 5 | 1 | 2 |

Table 1. Most common patch edits

From these multiple runs we see that the MOVE edit appears most often, while COPY was second most and last was DEL. Many of the positions from the bottom of the program is moved up to the top of the program. This is observable in best patches 1, 2, 3 and 5. 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, as the edits were at different lines and vastly different. An example we have chosen to shorten was the most interesting patch, patch 3. Patch 3 contains the following patch, best patch found: | COPY 4 -> 0:2 | MOVE 7 -> 2:0 | COPY 8 -> 2:0 which produced the algorithm below:

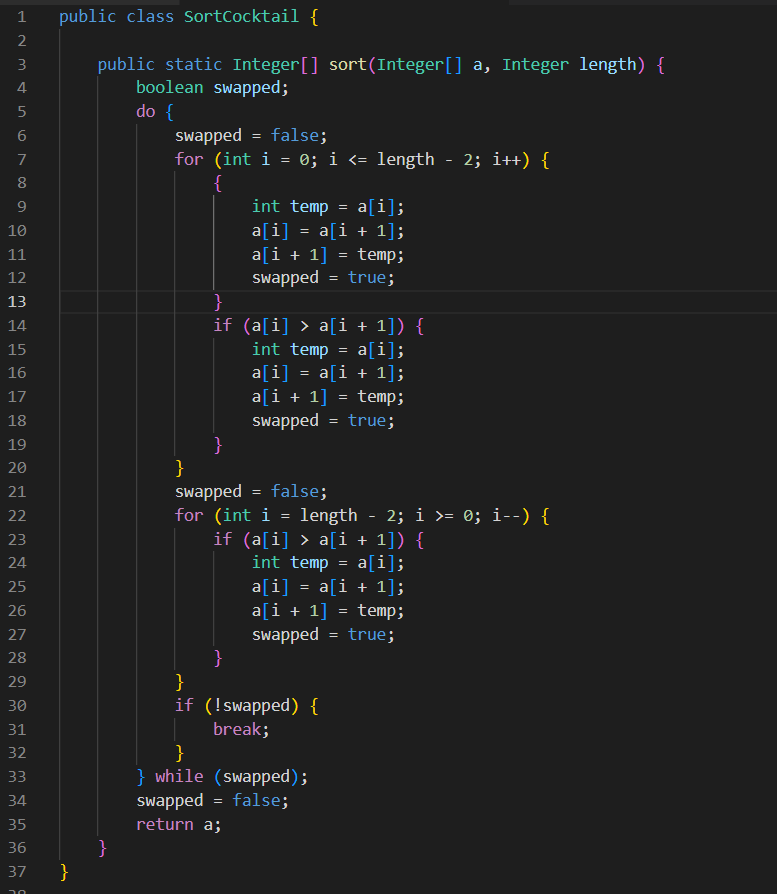


Fig 2. Optimized Sort Cocktail Algorithm using best patch 3

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 then produce the algorithm below.

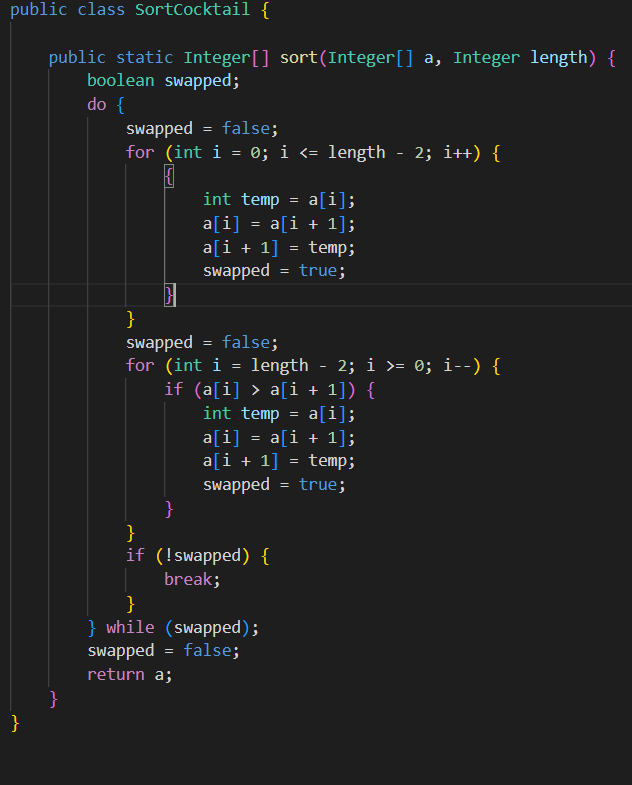


Fig 3. Optimized Sort Cocktail using shorten best patch

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

## Task 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 2: 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 run 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:

### The chosen program:

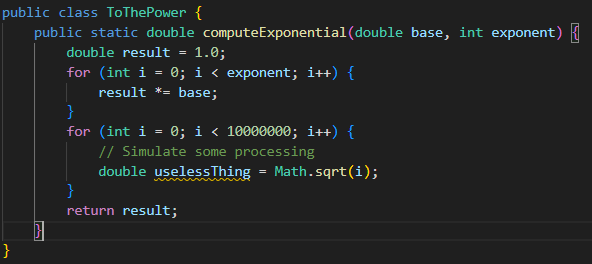
The program that was chosen for this case study, to optimize, was a custom program named ToThePower.java (implementation found in appendix part5). This program takes two numbers and produces the result of one to the power of the other. However, to add inefficiencies we add a for loop that hogs a bunch of processing time, by just doing a square root operation 1000000 times. Removing this will not remove any functionality to the program, so this is what we aim to have gin remove when optimizing the program.

Fig 4: The implementation of the case study algorithm “ToThePower”

The program itself is very simple, using an iterative loop to iteratively multiply the 2 numbers to get the total exponential answer. Without the extra baggage the program should run very fast. This program is tested using some equally simple Junit tests, which test a few simple cases and a few that produce more iteration and a larger number. These test cases can be found in appendix under part5.

### Results:

Using the same methodology from part 4, we are able to gather results for the best patches and equivalent patches for our “ToThePower” program. As can be seen from the table below, almost all of the best patches have | DEL 5 | involved in them. Upon debugging the code, it was found that the “5” was the ‘FOR’ loop that was extraneous to the program, and most other changes that were found to be faster basically made no difference in comparison to only deleting the ‘FOR’ loop.



Table 3: Result from running the updated search with examples/locoGP/ToThePower.java

The above method achieved a speedup of around 45%, meaning that the program ran in just under half of the time of the original run, with no changes. As well as these runs, another test was conducted, where the slow portion was much slower than that of the previous iteration. The updated code can be seen in the below figure.

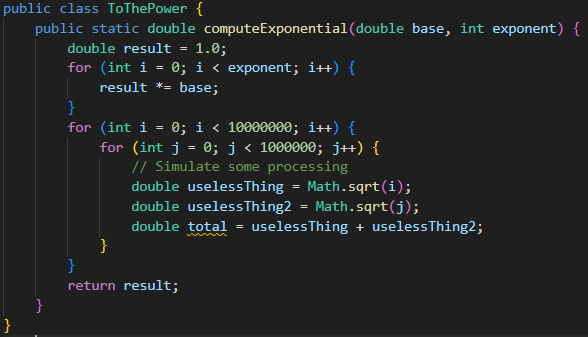
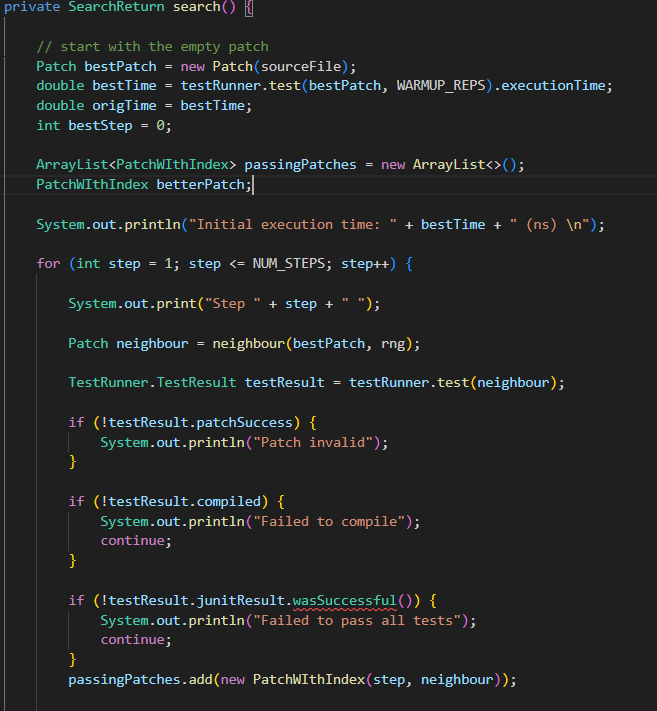


Fig 5: The implementation of the case study algorithm “ToThePower” extended for extra slowness

When using this approach, the slow portion should be, in theory, around 1 million times slower (this may not be exactly true due to the compiler doing optimization and caching of operations). However, when running this program in the same way as the previous one, although the speedup is already much more, the program often gets stuck. This is due to the random copying of the slow section, meaning that it will unfortunately timeout before completing. Before this happened, the run time was seen to be around 10% of the original. This shows that given an even slower program, the speedup will increase proportionally, to around the same compilation speed as the faster program, as they should reduce to effectively the same program.

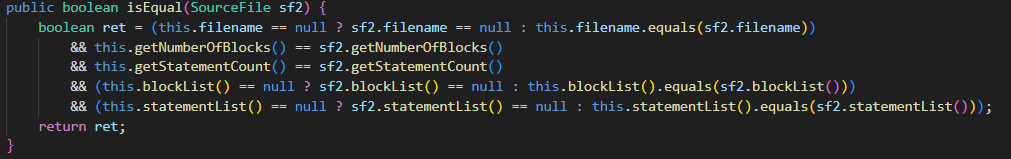
## Appendix:

### Code changes for part 4:

LocalSearch.java: Search() function becomes:



SourceFile.java: Added isEqual method to allow for checking against other sourcefile to see if they are the same



PatchWithIndex.java: Added data structure to easily keep track of a patch and it’s index

A screen shot of a computer code

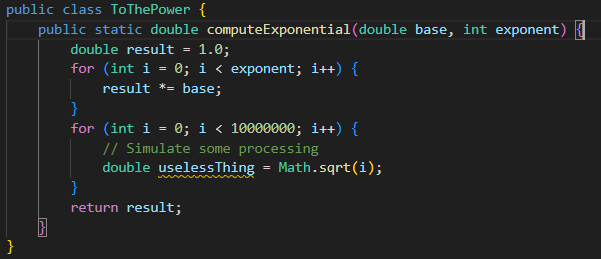
Description automatically generated

SearchReturn.java: Added search return for a custom type to return form the search function, giving both the best Patch, and the first instance of an equivalent patch.

A screen shot of a computer program

Description automatically generated

### Code changes for part 5:

ToThePower.java: Added ToThePower.java as the case study program  


ToThePowerTest.java: Added simple test suite for the ToThePower program (ignore junit errors)  
