Sikraken Final Report

Project by: Kacper Krakowiak (C00271692)
Project Supervisor: Dr Chris Meudec

SE TU
Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Introduction:

The Sikraken Optimization Tool aims to address the challenge of manual parameter selection in the Sikraken test case generator. Sikraken's effectiveness in generating test cases for C code depends on these two parameters: [$restarts,$tries]. Originally, finding the optimal values for these parameters required manual trial-and-error, a time-consuming and often mundane process.

This project introduced Machine Learning through implementing a Genetic Algorithm to automatically determine optimal parameter combinations, maximizing code coverage while reducing the time required for parameter tuning. The solution integrates directly with Sikraken and TestCov, leveraging parallel processing techniques to explore the parameters space.

This project features a multitude of different approaches for the numerous problems brought up to make sikraken as efficient as it can be. From managing resources efficiently, through Multi-Cored deployment of the Genetic Algorithm engine to graphing of results for easy visualization, this Final Report will describe in great detail each approach and list the difficulties that came with these approaches:

General Issues:

Problems Encountered and Solutions:

1. Concurrent TestCov Execution Conflicts

Problem: Attempts to run TestCov in parallel (multi-threaded) resulted in file conflicts and corruption of coverage data. Multiple TestCov instances would simultaneously attempt to access and modify the same coverage files, leading to inconsistent results.

Solution: Implemented a thread lock that allows only one TestCov process to run at any given time, while still enabling parallel execution of Sikrakens multiple instances. This approach maintained most performance benefits while ensuring reliable coverage measurement

```
# Run TestCov with a lock to prevent concurrent runs
with self.testcov_lock:
   testcov_cmd = f"cd {self.base_dir} && ./bin/run_testcov.sh
./regression_tests/{self.target_file} -32"
   cov_result = subprocess.run(testcov_cmd, shell=True,
capture_output=True, text=True, timeout=60)
```

2. Parameter Space Exploration

Problem: The parameter space for $restarts, $tries was originally hard limited to values in the ranges 1 -500. For some C code samples this range is simply too small, while for others it's overkill, in which case the algorithm will actually output less accurate results and take longer, wasting processing power on integers that are obviously too large, especially in the earlier generations where the algorithm just begins to learn.

Solution: Give the user an option to choose which range of integers should the algorithm pool from.

3. Timeout Handling

Problem: high parameter combinations can exceed reasonable execution times, wasting processing time and resources

Solution: Implemented an adaptive parameter halving strategy that cuts both individuals by a half when a set timeout occurs. This approach saves the initial generated parent, rather than discarding it and regenerating new values, maintaining better consistency throughout the generations, and saving resources.

```
def halve_parameters(self, individual: List[int]) -> List[int]:
   """Halve both parameters if timeout occurs"""
   return [max(1, x // 2) for x in individual]
```

4. Infinite Timeout Halving

Problem: In extremely rare cases the algorithm would infinitely halve integers until effectively both would reach 0 and an error would occur in the Algorithm. The reasons for this could be that the integers chosen in every halving iteration simply did not suffice sikrakens test generation algorithm. In any case this was a majour bug

Solution: A hardcoded limit to how many times an individual can be halved (set to 5), after which the individual is a simple failure and the algorithm skips it entirely.

5. Visualization Through Graphing

Problem: When a user wanted to see the performance of their chosen $restarts,$tries values, they had to manually keep track of their values (in a notepad or otherwise). This deemed impractical and for researching purposes added toil

Solution: A separate file that will plot a graph (using python's matplotlib package), based on the results and inputs of the Algorithm throughout its processing lifespan. This helps the user to keep track of the algorithm and displays progressions clearly throughout the generations.

Unachieved Goals:

1. Extending the Algorithm to Sikrakens budget mode.
   The budget mode in sikraken is the main mode, while the regression mode is more of a research mode which helps the user visualize patterns and understand the workings and effects minor parameter tweaks may have. In regression mode I am obviously only focusing on two parameters ($restarts,$tries), but in budget mode there are more: (<Budget>,<First_time_out>, <Max_time_out>, <Multiplier>, <Min_time_out>, <Margin>).

2. Flexible parameter overhead.
   The algorithm limits the values for $restarts,$tries that can be generated (obviously the algorithm cannot choose from an infinite number pool), however depending on the C code sample some limits may be too low, while for other C code samples the limits are too high and prove to be a waste of resource and time. A parameter overhead could be implemented which will be assigned by the C code sample file size (in MB). The bigger the C code sample, the more likely it will require higher integer value for $restarts, $tries.

3. Cloud Hosting.
   It comes as no surprise that the algorithm is extensively computation heavy, so hooking it up with an external cloud provider could give us extra CPU power to complete the algorithm processing at lightning fast speeds. This isn't essential to our implementation of the Genetic Algorithm, but simply out of own interest to add extra technologies (AWS, Azure Cloud etc), to the project.

Learning Outcomes:

This project proved to be a mountain of knowledge to me, and it sparked a genuine interest in machine learning. Every step, challenge, bug and additional feature taught me a lot in their own way. Just to name a few main learning outcomes, in short summaries that proved most significant throughout the project:

1. **Genetic Algorithm:**
   Hands-on experience of implementing a real Machine Learning algorithm on top of an already existing program/application (sikraken). This project wasn't as simple as from the ground up (as developing an app would have been), as the algorithm has to be suitable to work in tandem with sikraken, hence working closely with my project supervisor (and creator of sikraken), gave a good real world development experience.

2. **Concurrent Programming:**
   I learned how to accelerate my program by utilizing multi-threaded techniques and avoiding parts of the project that do not work in multithreaded (TestCov). To achieve this I learned and implemented parallelization techniques such as locks and synchronization mechanisms, that are available in vanilla python.

3. **Linux Operating System:**
   Sikraken is by design only made to be used on a linux system. In order to even begin work on this project I had to get a linux subsystem and learn how the file management and resource allocations on a linux operating system work. Learning Linux is a whole topic of its own, as in essence it works completely different from Windows, and allows for deeper customization and changing of vital setting (which have proved detrimental in my early development, causing setbacks to restart my operating system)

4. **Process Integration:**
   I learned techniques for integrating with existing tools through command-line interfaces on a Linux system.

5. **Testing Methodologies:**
   I gained a deeper understanding of code coverage metrics and test case generation principles, which added to the basic test cases knowledge I had in the first semester for the Agile Software Development module.

Alternative Approaches:

If starting the project again, I would consider the following alternative approaches:

1. **Additional Algorithm selection:** I would explore other optimization techniques alongside the Genetic Algorithm, to further reinforce and generate more accurate results. Algorithms such as Bayesian optimizers or simulated annealing or otherwise might prove helpful in finding the best parameters (especially when adding more than 2 parameters to search for).

2. **Early Parallelization:** I would incorporate parallelization from the start, which would save extra time when running tests on the algorithm, from the early stages

3. **Optimize for both Sikraken Modes:** I would focus on optimizing both the budget mode and regression mode from the start, to have a fully AI driven sikraken program for users to utilize.

4. **Gcov integration:** I would see whether perhaps it would be possible to change sikraken outputs to fit those of gcov (as TestCov doesn't work in parallel). This change would require major changes in the sikraken codebase itself, and would need to be done with the help of Chris.

5. **Genetic Algorithm further comparison:** I would further compare the Genetic Algorithms abilities, against more sophisticated methodologies/selections than random number generator (which i already have the result for). Other methodologies, such as simply doubling values for $restarts, $tries or otherwise, could give me insights on how well the GA fairs against other, easier solutions of finding best integer combinations.

Technical Issues:

1. **Parallelization strategy:** The original implementation called for a fully multi-threaded solution for all aspects of the test case generation within the Genetic Algorithm. During the implementation it became clear that TestCov by design will not work in concurrency, so I had to adapt to this by only implementing concurrency for sikraken in my algorithm.

2. **Integer Overhead limit:** the Genetic Algorithm needs to have a defined pool of integers to choose from for the [$restarts, $tries] values, otherwise it would choose from an infinite value pool, which would be unfeasible. The issue with choosing an overhead appears when unique C code samples require different value ranges. By default the range was hard coded to (1 - 500), but proved to be too small in some C code samples, while for others it was overkill. An easy solution is to give the power to the user to manually select the value range. On a more sophisticated solution, a value range could be imposed based on the C code sample size (in MB).

3. **Double Parameter Fitness Function:** Original idea was to include both code coverage and execution time as the fitness function in the Genetic algorithm, which would score the integer pair not only based on coverage but on execution time also. In practice this proved to be a failure as maintaining a balance between two objects proved too challenging and unfeasible. Instead the time constraint was adopted into the form of a timeout, so any integer pair that goes over time will be halved (not entirely wasted), and that particular initial timeout, discarded from the tournament selection altogether (instead of assigning a lower fitness score).

4. **Initial Resource Scrambling:** It seems that for small c code samples, the first 3 - 4 individuals in a generation (in the early generations most prevalent), those individuals tend to scramble to get into the processing pool, resulting in those 3 - 4 individuals failing every time. Attempts have been made to introduce a more complex and sophisticated semaphore into the Parallelization techniques, but to no avail. Again this issue is only unique to very small C code samples, for which sikraken can generate test cases for in mere seconds.

Areas of Research:

Majority of this project was in fact research based. Sikraken as a standalone software works as expected, the aim was to research and find potentially better (or not), with proof, ways to enhance sikraken, by introducing AI and machine learning. Areas of research also extended to test case generation, and the importance of testing code effectively, as a developer. Very extensive amounts of research was put into this project, some main areas of research were:

1. **Algorithms:** Choosing the right tool for the job was probably one of the most time consuming tasks. With many algorithms that can rate performance and choose best values, Genetic Algorithm is the best fit for sikrakens use case. Sikrakens regression mode with 2 integer pairs (that are independent mathematically from one another) and a single fitness result (coverage), is precisely the kind of problem Genetic Algorithms accelerate in.

2. **Genetic Algorithm:** once the best algorithm for the task was chosen, research on how it actually works had to be conducted. The Genetic Algorithm has 3 key intricacies, Selection, Crossover and Mutation, and the entire process of finding the best integer pairs leans on those 3 aspects. Other side solutions are a mere catalyst and/or resource saviour within the algorithm.

3. **Linux Operating Systems:** in depth understanding of linux and its command line for calling scripts/code was also required. A detailed setup of sikraken and testcov on a linux system also required knowledge, time, trial and error. Linux can be a very complicated system to master (in comparison to windows), but it allows me (the user) to take greater control of the underlying system and access powerful but fragile and crucial settings within it.

4. **Process Management in Python:** investigation of subprocess handling, particularly for managing timeouts, capturing output/input, and handling process failures , to give a detailed error log to make the development process of the algorithm as painless for me as possible.

5. **Third Party Programs for Code Coverage:** Sikraken by design is made to work alongside TestCov to measure coverage, but there are other programs out there that even perhaps work better than TestCov (like Gcov). Learning and understanding these programs was crucial, and making them work together effortlessly was also a learning curve in the grand scheme of developing a machine learning algorithm that will utilize all code test case generation techniques.

6. **Parallelization:** To save time and use resources more effectively, parallelization, with nuances, had to be utilized in the algorithm. Luckily python comes with parallelization built in and is very well documented.

Modules Descriptions:

My implementation consists of several key modules and components:

1. SikrakenOptimizer Class: The main class that implements the Genetic Algorithm and manages the optimization process. It handles population initialization, evaluation, selection, crossover, mutation, and tracking of the best solution.

2. Parameter Evaluation Functions: Specialized functions for running Sikraken and TestCov, capturing their output, and extracting relevant information (particularly coverage percentages), using basic regex.

3. Genetic Algorithm Operators: Implementation of tournament selection, single-point crossover, and random mutation operators customized for the parameter optimization problem.

4. Parallel Processing Framework: Thread pool management for parallel evaluation of parameters, for Sikraken only. A thread lock solution to refrain TestCov from attempting to access parallelization (as it's not designed to do so).

5. A matplotlib (python library) implementation to graph the Genetic Algorithm processing on, for easier understanding/viewing of results.

6. User Interface Module: Command-line interface for configuration, file selection, and results presentation.

Data Structures:

1. **Individual Representation:** where parameters are represented as a list of two integers, [$restarts,$tries]:
```
individual = [restarts_value, tries_value]  # e.g., [350, 200]
```

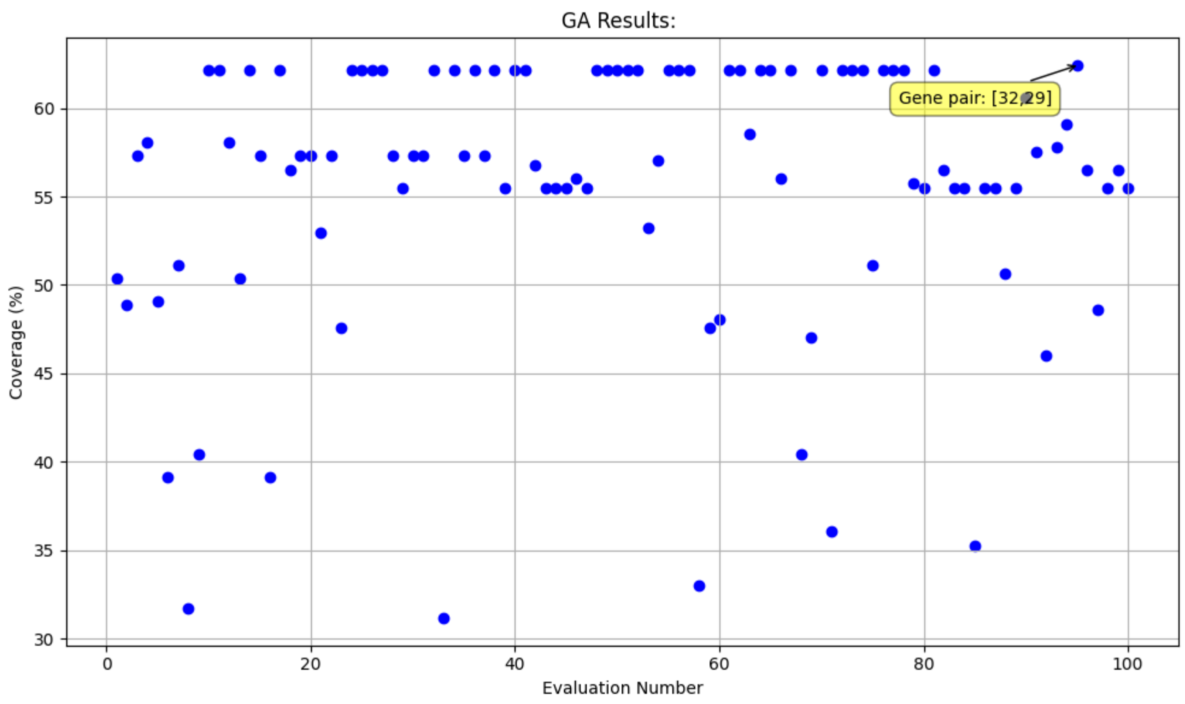
2. **Population:** essentially a list of individuals maintained across generations:
```
   def initialize_population(self) -> List[List[int]]:
       return [self.create_individual() for _ in
range(self.pop_size)]
```

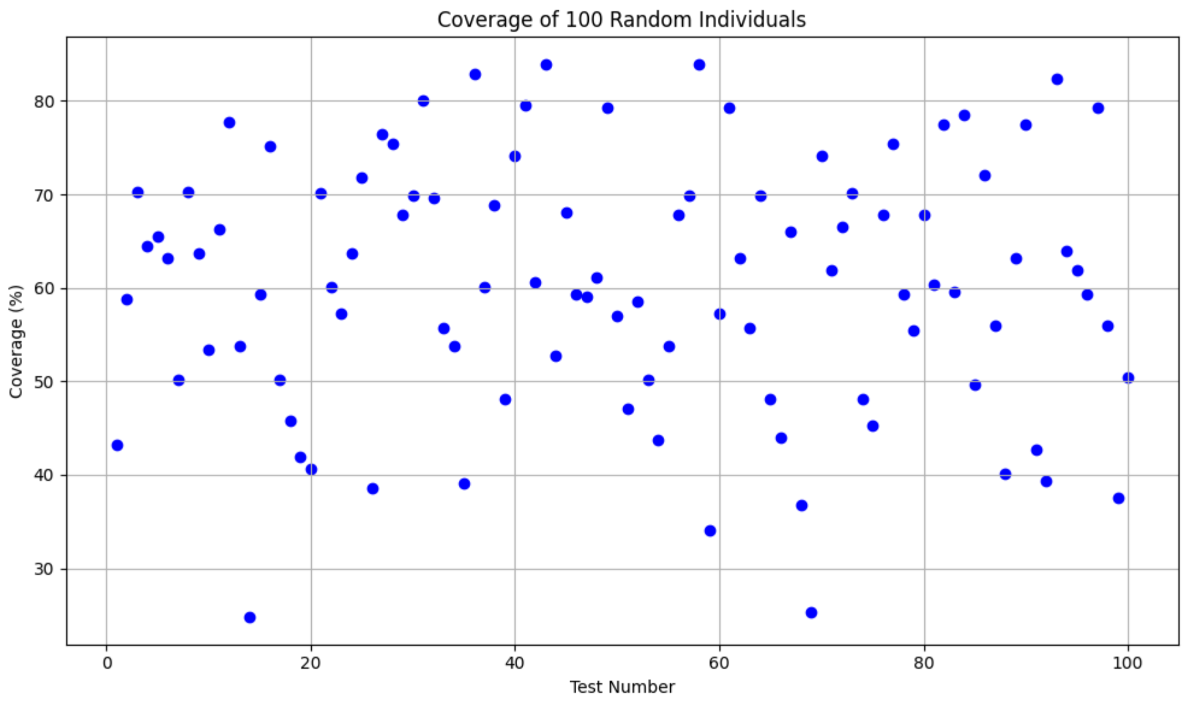
3. **Fitness:** parallel arrays of individuals and their coverage (fitness):
```
# Evaluate population in parallel
fitnesses = [0.0] * len(population)  # fitness list
```

4. **Result Storage:** A object of storage which contains params (or the individuals), coverage, elapsed time and error (if any)

Testing Methodologies:

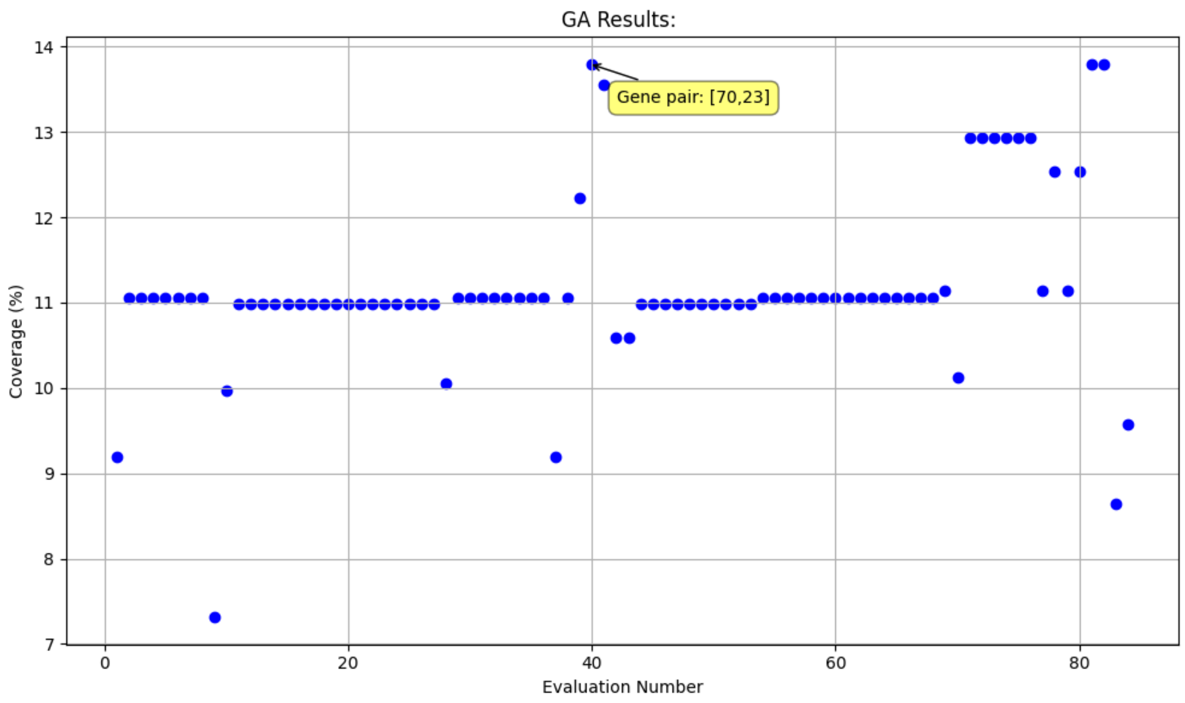
1. **Cross-File Validation:** The algorithm was executed on a plethora of different C code samples to ensure cross capabilities of the algorithm
2. **Comparison Testing:** The algorithm was compared against a random integer generator to prove its efficiency. Results for problem16 (~1700 lines of code):





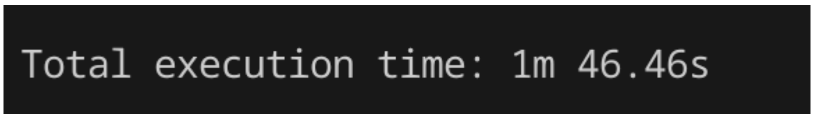
This shows that the algorithm learns over-time. The initial 2 generations (between 0 - 20 individuals, 10 individuals per generation) are scattered enough (showing essentially randomness), but the further down the generations the algorithm explores, we see more consistency between points. In comparison to the second graph (Random), no consistency, points scattered on the graph.

Additional results for Problem06, after parallelization was implemented, so keep that in mind some individuals may be staggered in their positions, however the generations do remain the same (i.e a set of 10 individuals is 1 generation):



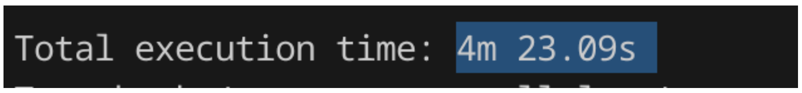
For this one it is clear that the algorithm only began to get the better coverage (of 14%), in the last 2 generations. Keeping in mind that probelm06 is considerably larger (~9500 lines of code) than Problem16, and getting a higher coverage seems to be impossible for this sample (even when inputting values for $restarts$tries manually).

3. **Reliability Testing:** Ran the algorithm on the same sample with the same parameters twice to ensure that the results remained in the general vicinity of the previous test.
4. **Parallelization Performance:** Proved that Parallelization saves considerate amount of time in my algorithm (up to 60% time saved with parallelization of sikraken alone). Parallelization saves about 3x - 4x time confirmed.
   Parralelized result for Arrays03-ValueResticstsIndex-2.c, a small C sample:



```
Total execution time: 1m 46.46s
```

vs
Non Parallelized:



```
Total execution time: 4m 23.09s
```

That's a difference of over 2 and a half minutes saved.

Genetic Algorithm Engine Detailed Operations:

The core of this project is obviously the genetic algorithm that evolves parameters through multiple generations to find optimal values for [$restarts,$tries]. This section provides a detailed walkthrough of how individuals progress through a single generation cycle.

1. **Population Initialization**
   Each run begins with a randomly generated population of 10 individuals. For example:
   Individual 1: [47, 125]
   Individual 2: [218, 93]
   …
   Individual 10: [156, 302]

   Each individual is a specific combination of $restarts and $tries values, within a user-specified range.

2. **Fitness Evaluation**
   Every individual in the population undergoes evaluation:
   a) Running Sikraken with these parameters
   b) Measuring code coverage using TestCov
   c) Recording the coverage percentage as that individual's fitness score

   For example:
   Individual 1: [47, 125] = 87.5% coverage
   Individual 2: [218, 93] = 100% coverage
   ...
   Individual 10: [156, 302] = 75% coverage

   The evaluation phase takes up the most computational resources and is parallelized.

3. **Parent Selection via Tournament**
   Once all individuals are evaluated, the algorithm selects parents for breeding using tournament selection:

   a) For each selection, the algorithm randomly picks 3 individuals (tournament_size= 3)
   b) The individual with the highest fitness from this group becomes a parent
   c) This process repeats to select a second parent

   For example, if tournaments select:
   Tournament 1: Individuals #2, #5, #8 = Individual #2 wins with 100% coverage
   Tournament 2: Individuals #1, #4, #7 = Individual #1 wins with 87.5% coverage

   The selected parents ([218, 93] and [47, 125]) proceed to the crossover phase.

4. **Crossover Operation**
   With a probability of 85% (crossover_rate = 0.85), the selected parents exchange genomes:

   a) A random crossover point is selected between the two genes
   b) The Genomes are swapped at this point to create two new offspring

   For our example with parents [218, 93] and [47, 125]:
   - Crossover creates: Child 1 = [218, 125], Child 2 = [47, 93]

   If crossover doesn't occur (15% chance), the children are exact copies of their parents, which helps reinforce results as in the off chance the same individual ran twice, can output slightly different results

5. **Mutation Operation**
   Each gene in every child has a 15% chance (mutation_rate = 0.15) of mutating:

   a) If mutation occurs, the gene value is replaced with a completely new random value from the valid range
   b) Mutation introduces new genetic material that might not have been present in the original population

6. **Replacement Strategy**
   The newly created children replace the original population entirely (generational replacement):

   a) This process repeats until we create 10 new individuals for the next generation
   b) The best solution found so far is kept separately (elitism)

   The next generation then begins with these new individuals, and the cycle repeats.

7. **Results and patterns**
   Over successive generations, the population typically converges toward optimal parameter values:

   a) Early generations (1-3) show high variability and mostly random exploration
   b) Middle generations (4-7) only begin showing patterns of improved fitness
   c) Later generations (8-10) often converge on a smaller range of high-performing values

   This is clearly visible in the results graph, where later generations cluster around high-performing parameter combinations, unlike the scattered distribution in early generations.

For Sikraken parameter optimization, this approach proves particularly effective because the relationship between parameter values and code coverage is complex and non-linear. Traditional optimization methods would struggle with this problem space, while the genetic algorithm can efficiently navigate it without requiring mathematical models of the underlying relationships.

Multi Threading Architecture Detailed Operations:

The parallelization strategy uses Python's native concurrent.futures.ThreadPoolExecutor, which manages a pool of worker threads that execute tasks concurrently.
The core of the parallel implementation is in the run() method:

```
            with
concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as
executor:
                # Submit all evaluation tasks
                future_to_idx = {}
                for i, (ind, gen_num, ind_num) in enumerate(eval_tasks):
                    future = executor.submit(self.evaluate, ind,
gen_num, ind_num)
                    future_to_idx[future] = i

                # Collect results as they complete
                for future in
concurrent.futures.as_completed(future_to_idx):
                    idx = future_to_idx[future]
                    try:
                        fitnesses[idx] = future.result()
                    except Exception as exc:
                        print(f"Evaluation {idx} generated an exception:
{exc}")
                        fitnesses[idx] = 0.0
```

The number of worker threads is automatically adjusted based on system capabilities:

```
        # Determine optimal number of workers based on CPU cores
        max_workers = min(self.pop_size, (os.cpu_count() or 4))
        print(f"Using {max_workers} worker threads for parallel
evaluation")
```

In essence each evaluation task returns a “future” object that represents a placeholder for a soon to be result. The actual results are then collected as they become available, hence why the individual numbers are staggered within the generation, as naturally some will finish faster than others. The basic exception handler prevents an individual task failure from crashing the entire evaluation process.

As mentioned before, sikraken instances can run in parallel, however testCov operates on a strict synchronized access model to the test cases. This poses a small inconvenience that is handled through a lock mechanism:

```
                    with self.testcov_lock:
                        testcov_cmd = f"cd {self.base_dir} &&
./bin/run_testcov.sh ./regression_tests/{self.target_file} -32"
                        cov_result = subprocess.run(testcov_cmd,
shell=True, capture_output=True, text=True, timeout=60)
```

Through all these solution, a relevant bug still resides for small c code samples that take the algorithm seconds to process, when the initial 3 - 4 individuals per generation scramble for thread-pool access, and as a result they return as failed (the subsequent individuals however recover from this and the program carries on). Attempts have been made to introduce semaphores, but those proved to be inconsistent, and those initial individuals would still be present. Due to this issue being isolated mainly to small C code samples and has a negligible effect on the whole algorithm, it has been pushed for now.

Conclusion:

The Sikraken optimization project touches on multiple fronts of computer related topics. The underlying algorithm itself was developed from ground up (rather than using a library) and proved to be the favorable method of developing machine learning for very specific use cases (such as integrating algorithms with other technologies such as sikraken/TestCov). Developing from scratch gives me more control over the variables and fine tune the algorithms settings to work effortlessly with the given programs. The Algorithm itself only proved to be a major part of the learning process, other topics such as Operating Systems, Concurrency, Data, Python played the other major role in this very involved and at times quite difficult project.

Through this project I have demonstrated that evolutionary computation techniques can effectively solve software engineering problems within code/applications, particularly in the context of test case generation. While there are some outstanding features I didn't have time to complete (such as integrating the algorithm into sikrakens second mode), I definitely have made the core functionality a major and valuable addition for sikraken.

Throughout this project I have also kept a rough diary, that kept a more detailed timeline of the work that was put into this project: Research Notes

References:

Natural-Selection_Wikipedia(2024). Research strategy [online]. Available from: https://en.wikipedia.org/wiki/Natural_selection#:~:text=He%20defined%20natural%20selection%20as,likely%20to%20survive%20and%20reproduce.[accessed 6 December 2024].

Sikraken user and development guide.(2024). Research Startegy [online]. Available from: Sikraken Development and User Guide [Accessed 29 March 2024].

What-Are-Genetic-Algorithms_YoutTube(2024). Research strategy [Video]. Available What are Genetic Algorithms? 6 December 2024].

Genetic-Algorithms-Explaine-By-Example_YoutTube(2024). Research strategy [Video]. Available Genetic Algorithms Explained By Example 6 December 2024].

Knapsack-Problem_Wikipedia(2024). Research strategy [online]. Available https://en.wikipedia.org/wiki/Knapsack_problem[accessed 6 December 2024].

Python Software Foundation. (2023). concurrent.futures — Launching parallel tasks. Python 3.11 Documentation. https://docs.python.org/3/library/concurrent.futures.html [accessed 29 March 2024].

Python Software Foundation. (2023). threading — Thread-based parallelism. Python 3.11 Documentation. https://docs.python.org/3/library/threading.html [accessed 29 March 2024].