CRACK: AutomatiC SeaRch for Corner Test CAses on 0-1 KnapsaCK Problem

Team No. 1
Team member: 홍현선, 양희찬, 전재민, 이수민
Github link: https://github.com/yheechan/crack

1. Proposal

The CRACK project stands at the forefront of algorithmic testing innovation, aspiring to validate the effectiveness of employing Genetic Algorithms (GAs) in the search for corner test cases within the complex domain of the 0-1 Knapsack Problem. The primary objective of CRACK is to experimentally assess the viability of utilizing Genetic Algorithms in the realm of algorithmic problem solving. Through empirical programs and rigorous experimentation, the project seeks to demonstrate the utility of GAs in uncovering elusive corner cases that may challenge traditional algorithmic solutions. By systematically refining and evolving potential test scenarios, CRACK aims to shed light on the efficiency of this automated approach, paving the way for more resilient and adaptive algorithms in the face of intricate computational challenges.

1.1 Problem Description

A corner test case, also known as a boundary or edge case, is a specific test scenario or input data set that represents the extreme or limit conditions of a system or program. These cases are designed to explore the behavior of a system at the boundaries of its input space. Corner test cases are particularly valuable because they often reveal vulnerabilities, errors, or unexpected behavior that might not be apparent during regular testing with typical inputs.

In the context of algorithmic problem solving, a corner test case serves as a critical and challenging input that tests the limits of the algorithm's capabilities. It aims to expose potential weaknesses or incorrect assumptions in the algorithm by exploring scenarios that are less common or may not be immediately obvious.

The "CRACK" project specifically focuses on the 0-1 Knapsack Problem, a classic optimization problem where items have weights and values, and the goal is to determine the combination of items to maximize the total value within a given weight constraint. This problem is known for its complexity, and identifying corner cases for optimal algorithmic solutions is particularly challenging.

The essence of the "CRACK" project lies in its application of Genetic Algorithms for the systematic generation of corner test cases in the context of the 0-1 Knapsack Problem. This project stands as a strategic endeavor to deploy Genetic Algorithms with a specific focus on uncovering challenging scenarios that push the boundaries of algorithmic solutions. By

forging its own path in this innovative approach, CRACK aims to enhance the robustness and effectiveness of solutions tailored to the complexities of the 0-1 Knapsack Problem.

1.2 Naive Approach: Code-Based Approach

In our pursuit of effective strategies for generating corner test cases in the 0-1 Knapsack Problem, our initial approach was rooted in leveraging code-based meta-heuristic information. This involved tapping into details such as line/branch coverage and path execution coverage within the code. However, a crucial limitation emerged: the code-based fitness failed to distinguish between general and corner test cases.

Algorithm 1 Greedy(items, budget)

```
1: result ← 0
2: Sort items in descending order based on value.
3: for each item u ∈ items do
4:     if u.cost ≤ budget then
5:         budget ← budget − u.cost
6:         result ← result + u.value
7:     end if
8: end for
9: return result
```

Figure 1. Greedy Algorithm for 0-1 Knapsack Problem

As illustrated by the pseudo-code of a misguided greedy algorithm for the 0-1 Knapsack Problem, the approach relied on sorting items by value in descending order and subsequently including them in the knapsack if deemed acceptable. This process occurred uniformly, irrespective of the proximity to a corner test case. In essence, there was no discernible difference in terms of line coverage, branch coverage, or path execution coverage between the handling of general and corner cases within this naive approach.

Recognizing this inherent limitation, we pivoted our strategy to a program-based approach, seeking to address the shortcomings of the initial code-centric method. This shift reflects our commitment to refining our corner test case generation methodology and underscores the importance of adapting our strategies to the intricacies of the 0-1 Knapsack Problem. Through this evolution, we strive to enhance the precision and effectiveness of our approach, ensuring that our corner test cases truly push the boundaries of algorithmic solutions.

2. Empirical Program-Based Approach

2.1 Genetic Algorithm for CRACK

The Empirical Program-Based Approach within the CRACK project unfolds as a sophisticated mechanism designed to harness the power of Genetic Algorithms for the purpose of discovering corner test cases in the challenging domain of the 0-1 Knapsack Problem.

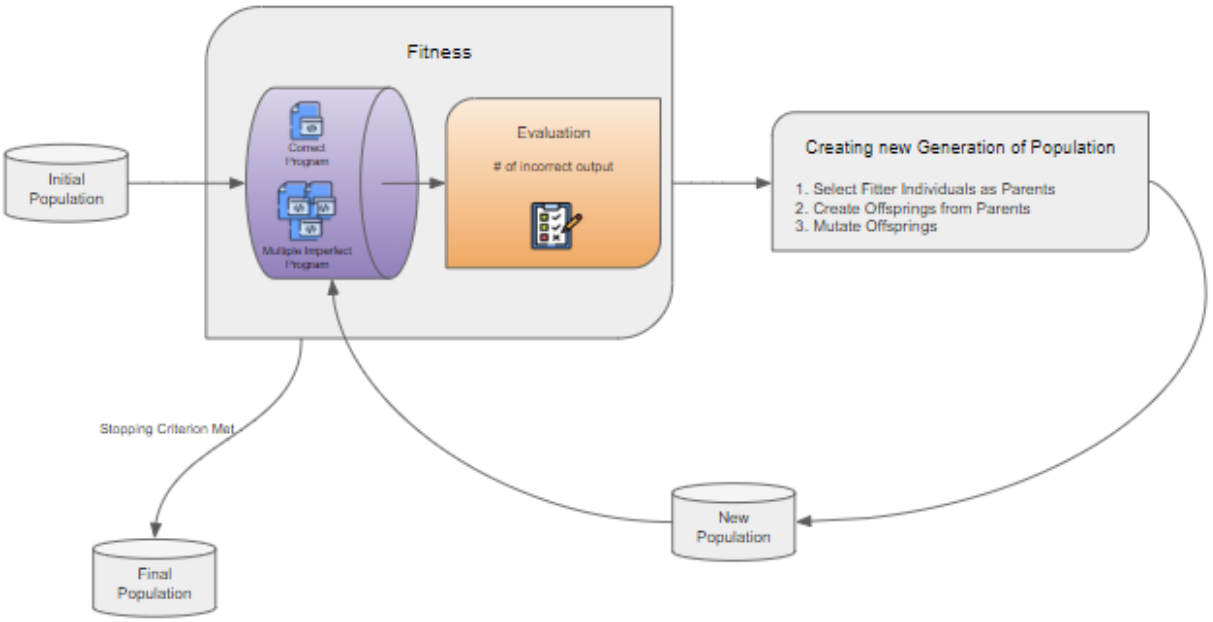


Figure 2. Overall Procedure for Genetic Algorithm

The process begins by initializing an initial random population of seeds. Each seed undergoes rigorous evaluation of its fitness, gauging the proximity of its associated corner case. Here, the ingenuity of this approach becomes evident as it assembles a diverse set of programs, featuring a correct implementation based on dynamic programming algorithms alongside multiple imperfect programs, such as those derived from greedy algorithms.

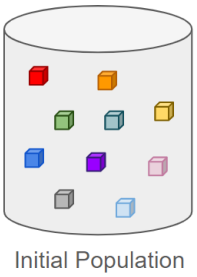
The programs, encompassing both the correct and imperfect variants, are executed with the entire initial population of inputs, laying the groundwork for a comprehensive evaluation. The distinctive strength of this approach lies in the subsequent fitness assessment, which measures the difference in output among the correct and imperfect programs. This differential analysis is based on the count of imperfect programs yielding incorrect outputs.

With this wealth of information, the fitness of each seed in the population is calculated, setting the stage for the application of Genetic Algorithms (GA). The GA operates iteratively, incorporating key genetic principles such as selection, crossover, and mutation, optimizing the population towards the identification of corner test cases. This cyclical process repeats until the stopping criterion is met, culminating in the finalization of the genetic algorithm.

Crucially, the success of the Empirical Program-Based Approach relies on the inclusion of imperfect programs, sourced from reputable algorithmic problem-solving online judge websites like LeetCode or Baekjoon Online Judge. In this context, 35 imperfect programs were initially collected, and through a meticulous clustering process, 21 distinct imperfect programs were extracted to ensure diversity while eliminating redundancy among programs with identical results. This methodical approach enhances the richness of the program pool, contributing to the effectiveness and adaptability of the Genetic Algorithm in uncovering corner test cases for the 0-1 Knapsack Problem.

2.2 Initial Population

The initial population begins with randomly initiated individuals. With population size set to 5, the objective is to evolve the population so that the individuals may kill all the existing 21 imperfect programs. An individual initiated at random means that the number of elements a test case holds, the budget value, and the cost and weight values for each element are set at a random value within the constraints of a valid test case.

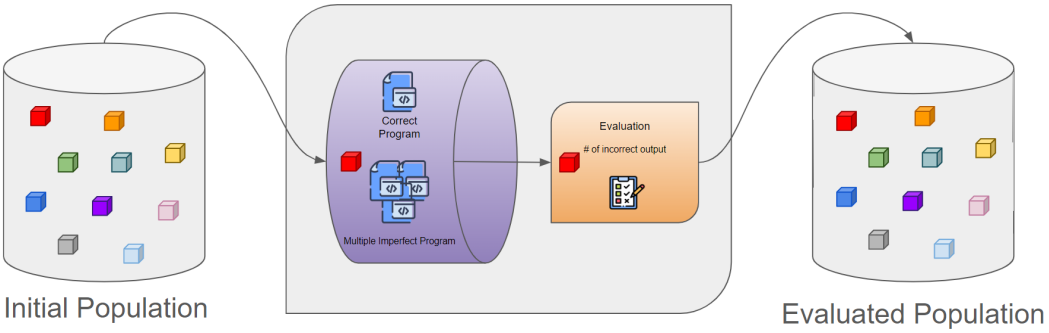


Initial Population

2.3 Fitness

Each individual in the population is given a fitness score. An individual is a test case (input) for the existing programs. Therefore, the fitness score for each individual is assigned by calculating how many imperfect the individual killed after running the individual on each imperfect program. The main purpose for assigning a fitness score to an individual is to evolve the population (test suite) to a more fit population for the next generation.

$$Fitness = \frac{\#\ of\ killed\ programs}{total\ \#\ of\ imperfect\ programs}$$



2.4 Creating New Generation of Population

Creating a new generation of population is a pivotal step in the evolutionary process of refining test cases through the Genetic Algorithm within the CRACK project. After the fitness assessment of individuals in the current population, the generation of a new population unfolds through a strategic combination of crossover and mutation.

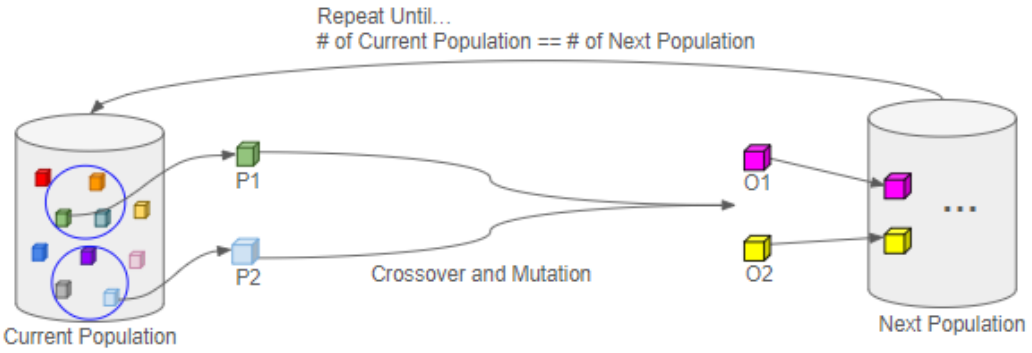


Figure 3. Procedure for Creating New Population

The procedure for generating a new population is systematically executed as follows: it commences with the existing population, followed by the random selection of two subsets from the current population. From each subset, a parent is chosen to undergo the crucial processes of crossover and mutation, resulting in the creation of two offspring. These offspring are then seamlessly integrated into the next population. The iterative nature of this process continues until the size of the current population aligns with the size of the next population, at which point the procedure is halted.

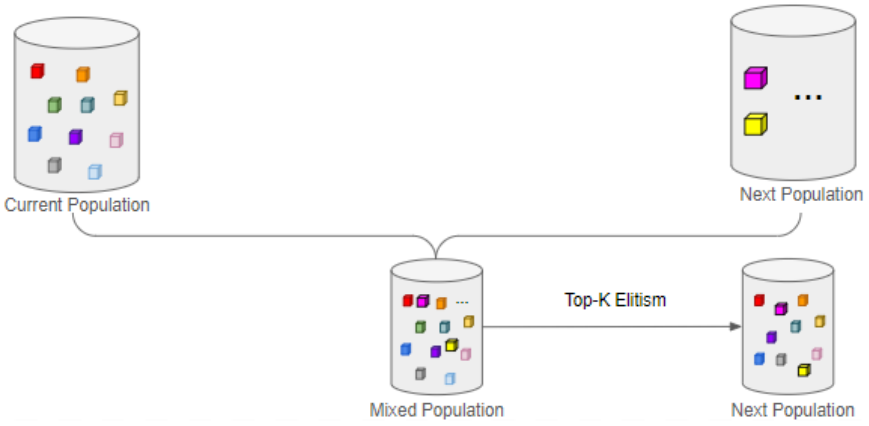


Figure 4. top-K Elite Selection

At this juncture, the project holds two distinct populations: the current and the next. Subsequently, these populations are merged to consolidate the results of the evolutionary process. The culmination of this merging process is a combined pool of potential test cases from both populations.

Following this consolidation, a selection process unfolds. The top-K best test cases, identified based on their fitness values, are singled out from this amalgamated pool. This elite selection forms the nucleus for creating the subsequent generation, ushering in a new wave of potential corner test cases that inherit the strengths and adaptability refined through the Genetic Algorithm.

2.5 Crossover/Mutation

Let's explore the Crossover phase, a critical step in the CRACK project's Genetic Algorithm. In simple terms, Crossover involves combining parts from two parents to create offspring. For instance, assume that

Parent 1 with (V11, W11), (V12, W12), (V13, W13)
Parent 2 with (V21, W21), (V22, W22), (V23, W23)

And the offspring are like

Offspring 1: (V11, W11), (V21, W21), (V22, W22)
Offspring 2: (V12, W12), (V13, W13), (V23, W23)

Offspring inherit characteristics from both parents.

We naturally believed that preserving continuity in this procedure is vital for the efficiency of the crossover, as a randomly mixed chromosome could lead to the creation of nonsensical test cases. To accomplish this, we employ a Crossover Point (cp) chosen through a discrete normalized distribution, identifying the central point for recombination, rather than utilizing a uniform distribution.

Once the offspring is created, we move to budget allocation. Budgets are assigned proportionally to the chromosome ratio of each parent. Let's consider factors like the budgets of Parent 1 (B1) and Parent 2 (B2), the number of items for each parent (N1 and N2), and their crossover points (cp1 and cp2). The resulting budgets for Offspring 1 and Offspring 2 are denoted as b1 and b2.

Here are the simple formulas for budget assignment:

$$b_1 = cp_1/N_1 * B_1$$
$$b_2 = (N_2 - cp_2)/N_2 * B_2$$

To manage budget allocation, we ensure that if the assigned budget exceeds a specified limit (b_max = 50000), it is then capped at the maximum value using the formula:

$$b = min(b_{max}, b)$$

This approach ensures effective distribution and control of resources for the offspring, contributing to the success of the Crossover phase in the Genetic Algorithm.

Now, let's delve into the Mutation phase. During this phase, both budget and item properties, including value and weight, undergo random mutations, introducing variations to enhance the evolutionary process.

In the Mutation process, we apply random changes by either adding or subtracting 1, and this is done with a probability of 20%. Additionally, there is an 80% probability of selecting a random value within a specified range. If a mutation causes a value to exceed its predefined limits, we reverse the mutation to ensure the feasibility of the solution. This strategic approach to Mutation introduces randomness and diversity into the genetic makeup of potential test cases, contributing to the overall adaptability and effectiveness of the Genetic Algorithm.

2.6 Stopping Criterion

In the exploration of Genetic Algorithms, a central focus of our investigation centers around the establishment of a robust stopping criterion. This critical point, marking the end of the algorithm's execution, plays a crucial role in evaluating the effectiveness of our approach. Our defined stopping criterion consists of two essential components.

First, iteration count termination. The algorithm concludes its execution when the iteration count reaches the predetermined maximum of 100 cycles. This limitation allows for a comprehensive evaluation of the algorithm's performance within a finite scope.

Second, fitness threshold termination. The stopping criterion also includes a fitness threshold of 1. This signifies that our test suite has produced an incorrect output for every candidate knapsack program. By integrating this dual stopping criterion, we gain insights into the percentage of test suites that failed even after 100 iterations. Additionally, it helps distinguish the necessary iterations for rejecting every problem within a given random initial population.

3. Experiment

3.1 Experiment Setup

To make the experiment better, understanding of the variables involved in our experimental setup is essential. These variables fall into two broad categories: independent and controlled. Independent variables can be adjusted to observe different outcomes, primarily focusing on the number of iterations until the stopping criterion is met. Controlled variables include population size, selection size, crossover rate, mutation rate, knapsack problem specifications, and parameters for the stopping criterion. Maintaining the consistency of these variables ensures the reliability of our experimentation.

To ensure the dependability, generalizability, consistency, and variability of our results relies on careful randomization and replication techniques. Within our Genetic Algorithm framework, randomness is introduced through various stages, including the initialization of the initial population, selection for crossover and mutation, and the application of crossover and mutation techniques. This randomness is further emphasized through 10 iterations for each set of independent variables, providing a robust replication technique.

Exploring the methodology behind our data collection process during successive generations, two key metrics guide our measurements: the fitness function and runtime analysis.

Our data collection involved the careful recording of results for every candidate program and the corresponding fitness values at each generation. The visual representation provided below offers a simplified overview of our results, obtained from executing seven distinct knapsack programs with intentional inaccuracies. These detailed setups serve as the foundation for executing our programs and extracting meaningful results.

3.2 Result and Evaluation

We conducted an ablation study to measure the effects of the two methods, ‘budget proportion’ and ‘Normal Crossover Point select’, proposed in our study. Thus, we performed 10 experiments for each method, using two criteria for evaluation: ‘fitness 1 reach ratio before generation_limit’ and ‘average generations to reach fitness 1’.

Method	Random	Budget Proportion	Norm CP select	Both B.P & Norm CP
Fitness 1 Reach Ratio Before Gen_Limit (%)	70%	70%	60%	80%
Average Generations to Reach Fitness 1	56.43 Gen	43.71 Gen	40.84 Gen	45.25 Gen

Table 1. Result of ablation study

Firstly, the results of measuring the ratio of reaching fitness 1 before the generation limit(in this case, one hundred generations) showed that applying both methods yielded the highest rate of 80%. In contrast, applying no method (‘random’) and applying only ‘Budget proportion’ resulted in rates of 70%, and ‘Norm cp select’ had a rate of 60%. As we can see from these results, the percentage of each method reaching fitness 1 before the generation limit is similar, and all are above half, indicating that our genetic algorithm (CRACK) is effective at finding edge cases.

In addition, we measured the ‘average number of generations to reach fitness 1’ for all methods, and found that it took at least 10 more generations to reach fitness 1 using nothing than using any of the ‘Norm CP select’ or ‘Budget Proportion’ methods proposed in this report. This shows that the two methods proposed in this report can speed up the convergence to the correct solution.

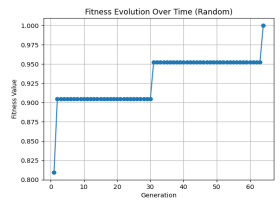
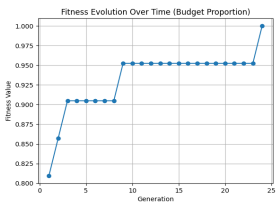
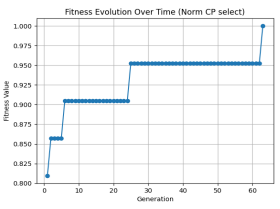
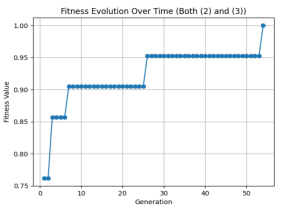
random	Budget proportion	Norm CP select	Both B.P & Norm CP
			

Table 2. Fitness evolution over time

Moreover, we observed the changes in fitness over time. As shown in Table 2, we can see that the genes of all four methods evolve in the right direction over time.

4. Conclusion

In this paper, we propose CRACK, a genetic algorithm-based 0-1 knapsack edge case finder. CRACK is based on the idea that a good set of inputs can be created by evolving each input to be better. It evolves to kill more programs by using the proportion of killed programs as the fitness function. In addition, crossover proposed two methods to improve the performance of the program: ‘Budget proportion’ and ‘Norm CP select’. We conducted 10 experiments for each method to evaluate the results of CRACK. The experimental results show that fitness 1.0 can be obtained within 100 generations in more than half of the experiments, indicating that CRACK can be used to effectively find the edge case of the 0-1 knapsack problem. We also found that the convergence of the solution is faster when applying ‘Budget proportion’ and ‘Norm CP select’.

Our study is limited to the 0-1 knapsack problem. However, we believe that CRACK is capable of identifying edge cases for other problems. In the future, we would like to extend our work to solve other problems.