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

Jaemin Jeon (20180576)
Sumin Lee (20200469)
Heechan Yang (20234352)
Hyunsun Hong (20234490)
Team #1

CS454 AI Based Software Engineering
Dec 04, 2023

Contents

Recap: Motivation

- Generally, there are two types of failure in problem solving, i.e. Correctness failure and Time complexity failure.

- Corner case is a test case that shows a hard-to-think counterexample to the wrong program.

- Finding corner cases for algorithmic problem solving is quite difficult for humans.

LeetCode

acm International Collegiate Programming Contest

Time Submitted	Status	Runtime	Memory	Language
10/12/2021 00:24	Wrong Answer	N/A	N/A	python3
10/12/2021 00:21	Wrong Answer	N/A	N/A	python3
10/12/2021 00:20	Wrong Answer	N/A	N/A	python3
10/12/2021 00:01	Time Limit Exceeded	N/A	N/A	java

Recap: Problem Description

Consider 0-1 knapsack problem. Assume that the budget is 5.

General Case

{(10, 4), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1)}

Both correct program(dynamic programming) and incorrect program(greedy) result in 12.

Corner Case

{(4, 4), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1)}

Correct program results in 10 but incorrect program 6.

Genetic Algorithm

Corner Case

{(7, 4), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1)}

Correct program results in 10 but incorrect program 9.

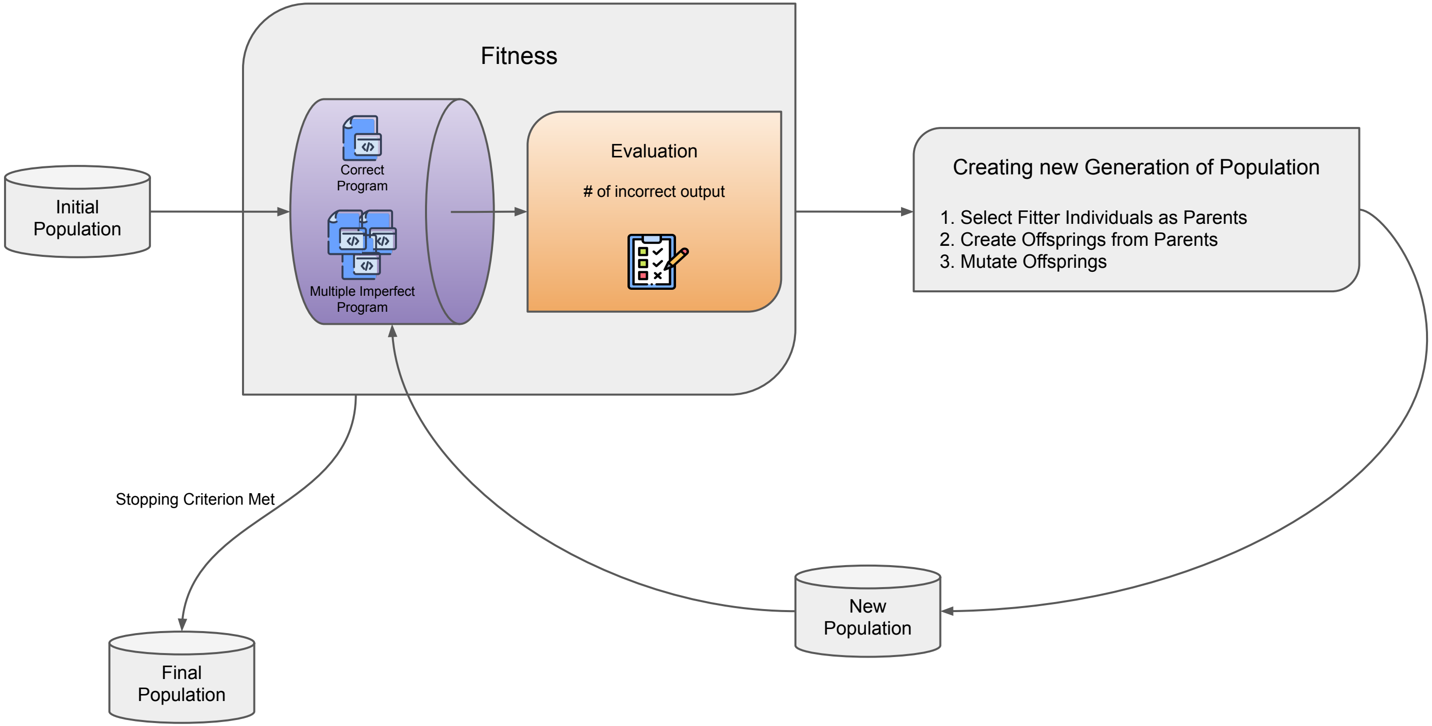
Recap: Naive Approach

- Taking advantage of code-based meta-heuristic search.
- Line/branch coverage, path execution coverage and so on.
- Code-based fitness does not differentiate general test case and corner test case.
- Not code-based, but program-based.

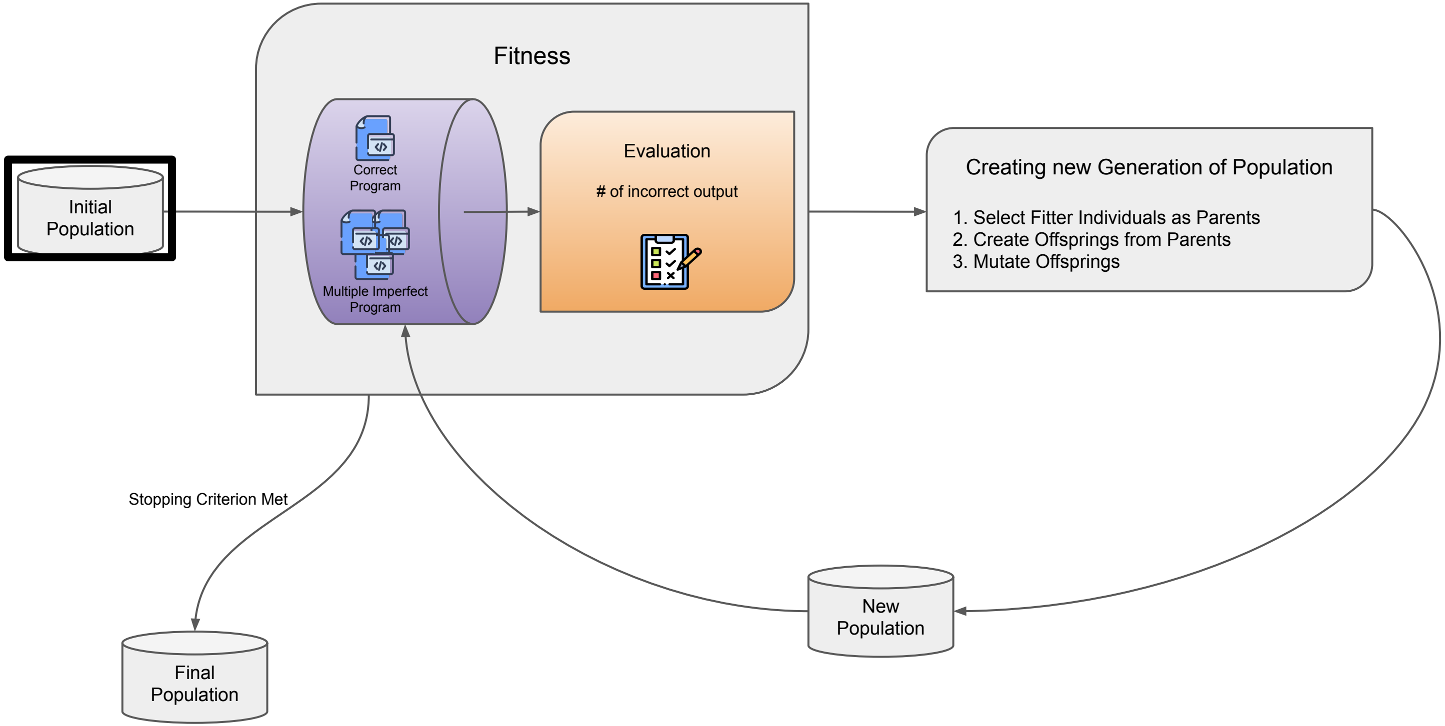
Algorithm 1 Greedy(items, budget)

```
result ← 0
Sort items in descending order based on value.
for each item u ∈ items do
    if u.cost ≤ budget then
        budget ← budget − u.cost
        result ← result + u.value
    end if
end for
return result
```

Overview: GA for Finding Corner Test Cases

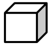


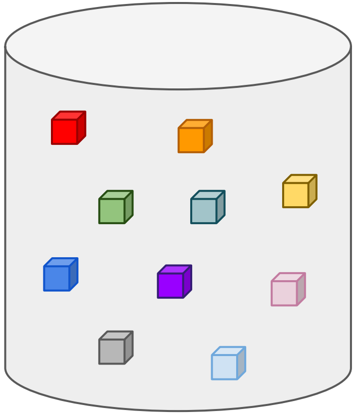
Initial Population



Initial Population

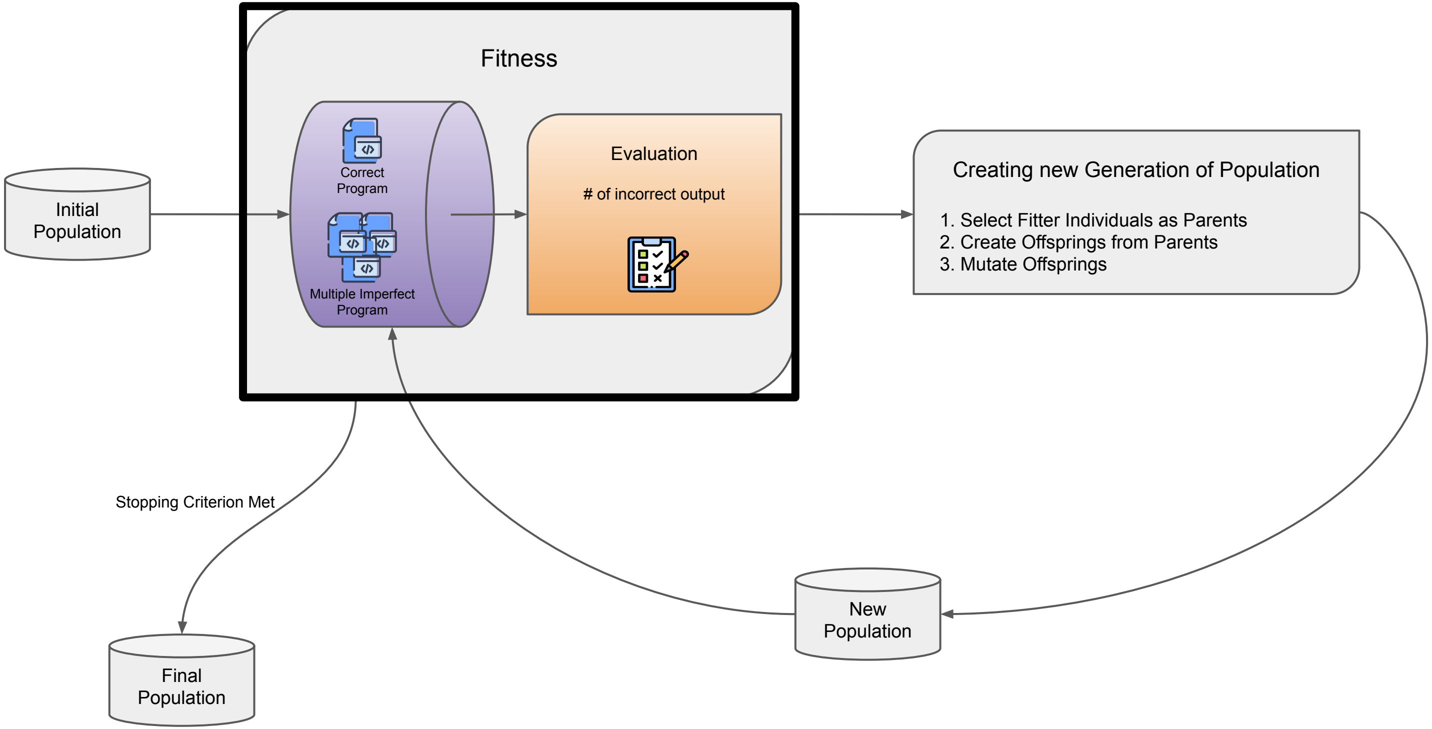
- Generate Random Inputs with Constraints
  - # of elements: 1 <= N <= 50
  - Budget: 1 <= B <= 50000
  - Weight Value: 1 <= weight_val <= 50000
  - Cost Value: 1 <= cost_val <= 5000
- Initial Population: total of 10 randomly generated input case

 = Single Input Case



Initial Population

Fitness

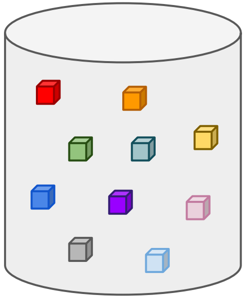


Initial Population
Fitness
Correct Program
Multiple Imperfect Program
Evaluation
# of incorrect output
Creating new Generation of Population
1. Select Fitter Individuals as Parents
2. Create Offsprings from Parents
3. Mutate Offsprings
Stopping Criterion Met
Final Population
New Population

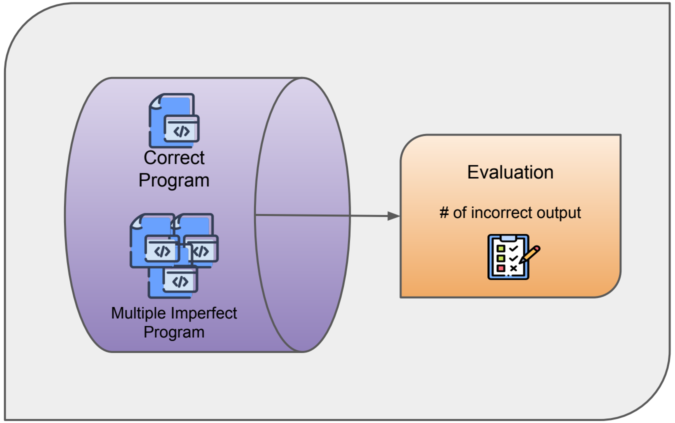
Fitness

- How many imperfect programs has the population killed?
    - imperfect program returning incorrect results (compared against correct program)

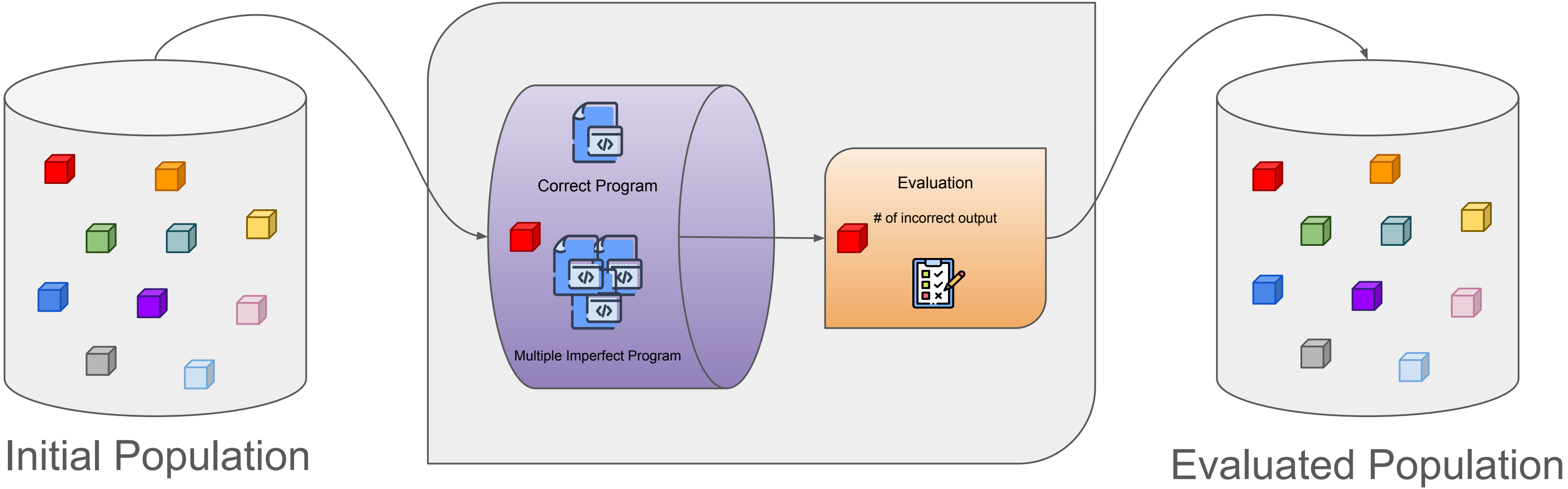
$$Fitness = \frac{\#\ of\ killed\ programs}{total\ \#\ of\ imperfect\ programs}$$



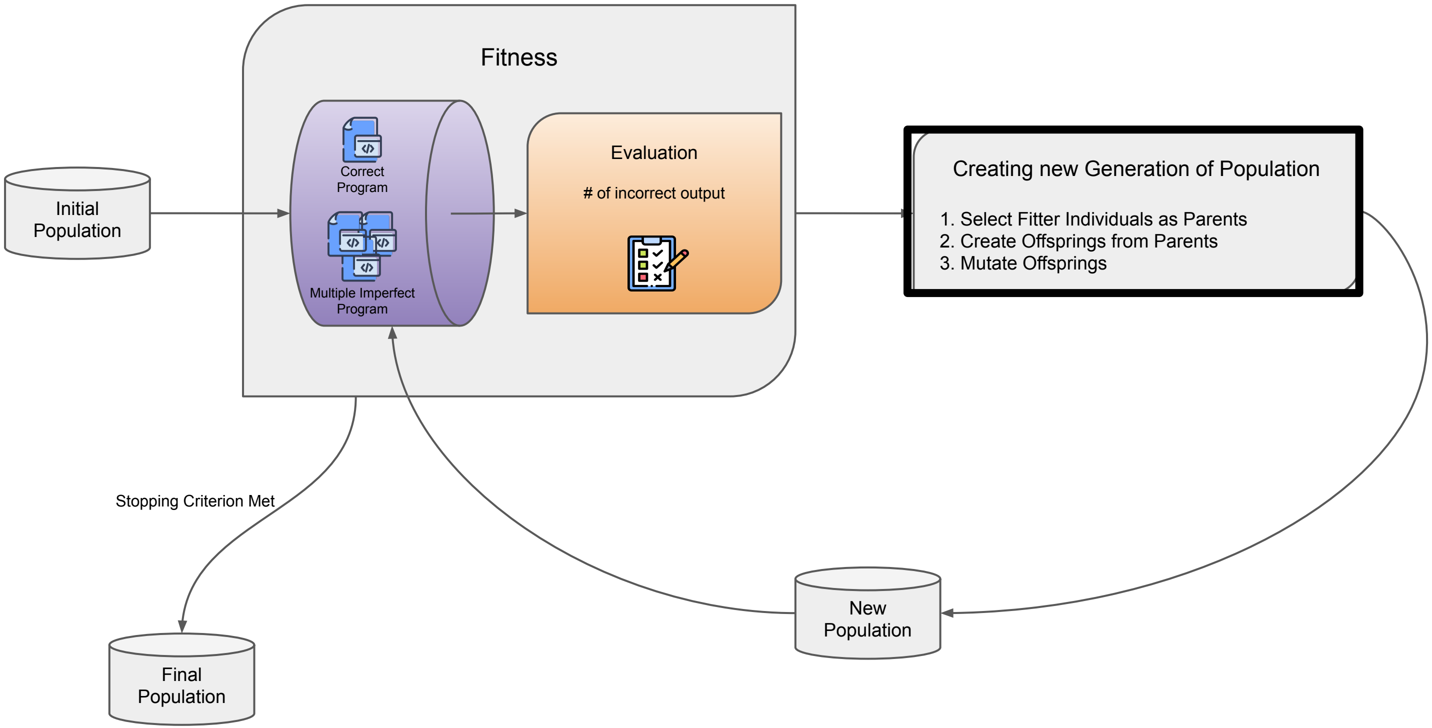
Initial Population



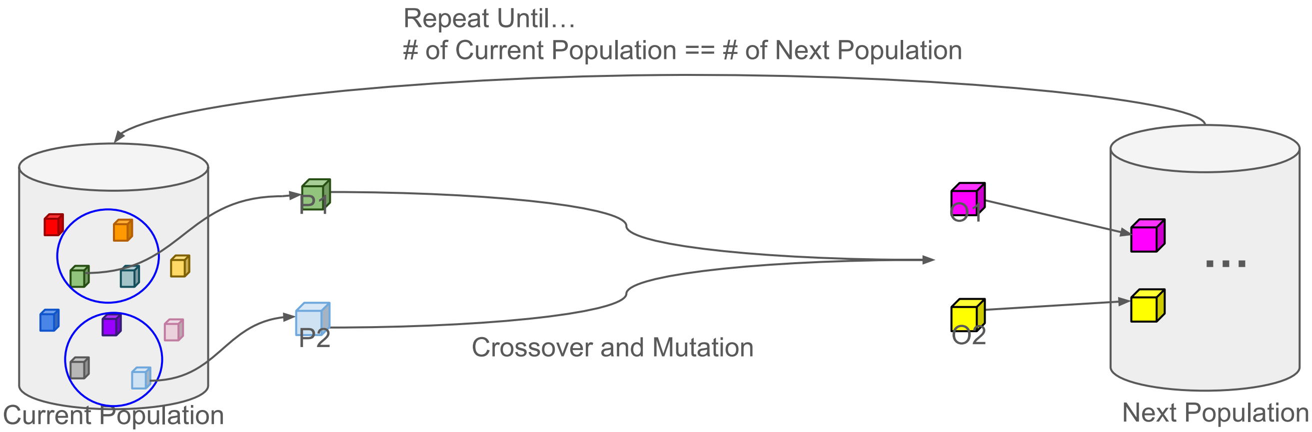
Fitness


Correct Program
Multiple Imperfect Program
Evaluation
# of incorrect output
Initial Population
Evaluated Population

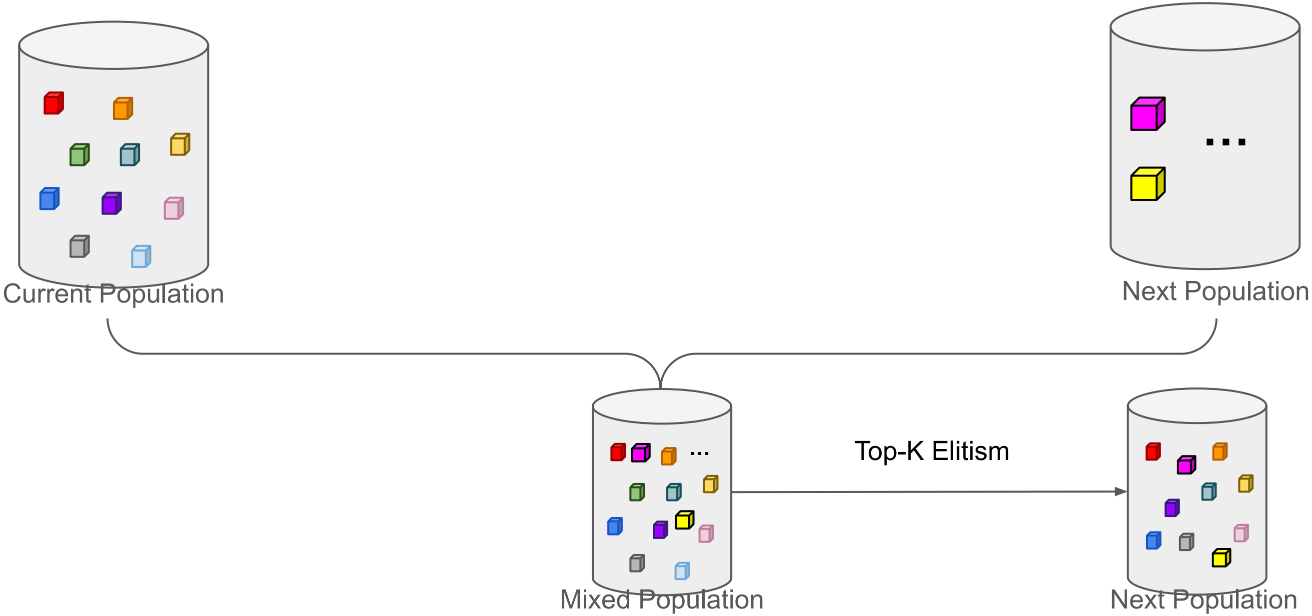
Crossover / Mutation



Creating New Generation of Population



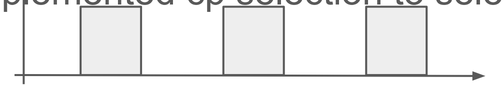
Creating New Generation of Population



Crossover

- In crossover, each parent was split into two parts, which were then recombined.
- The crossover point(cp) determines crossover functionality.
  - For example,

    cp1 cp2
  - Parent 1 : (V_{11}, W_{11}), (V_{12}, W_{12}), (V_{13}, W_{13}) and Parent 2 : (V_{21}, W_{21}), (V_{22}, W_{22}), (V_{23}, W_{23})
  - Off 1 : (V_{11}, W_{11}), (V_{21}, W_{21}), (V_{22}, W_{22}) and Off 2 : (V_{12}, W_{12}), (V_{13}, W_{13}), (V_{23}, W_{23})
- We assume ensuring continuity of chromosome would enhances offspring effectiveness.
  - Implemented cp selection to select mid-point via discrete normalized distribution.



cp	0	1	2
P(cp)	0.33	0.33	0.33

Discrete Uniform Distribution

cp	0	1	2
P(cp)	0.25	0.5	0.25

Discrete Normalized Distribution

Crossover

- Assign offspring budgets proportionally to their chromosome ratio.
  - Assume
  - B_1, B_2 as budget of parent1 and parent 2.
  - N_1, N_2 as # of items of parent 1 and parent 2.
  - cp_1, cp_2 as the crossover point of parent 1 and parent 2.
  - b_1, b_2 as the budget of offspring 1 and offspring 2.

$$\boldsymbol{b_1 = B_1(cp_1 + 1)/N_1 + B_2(N_2 - (cp_2 + 1))/N_2}$$
$$\boldsymbol{b_2 = B_1(N_1 - (cp_1 + 1))/N_1 + B_2(cp_2 + 1)/N_2}$$

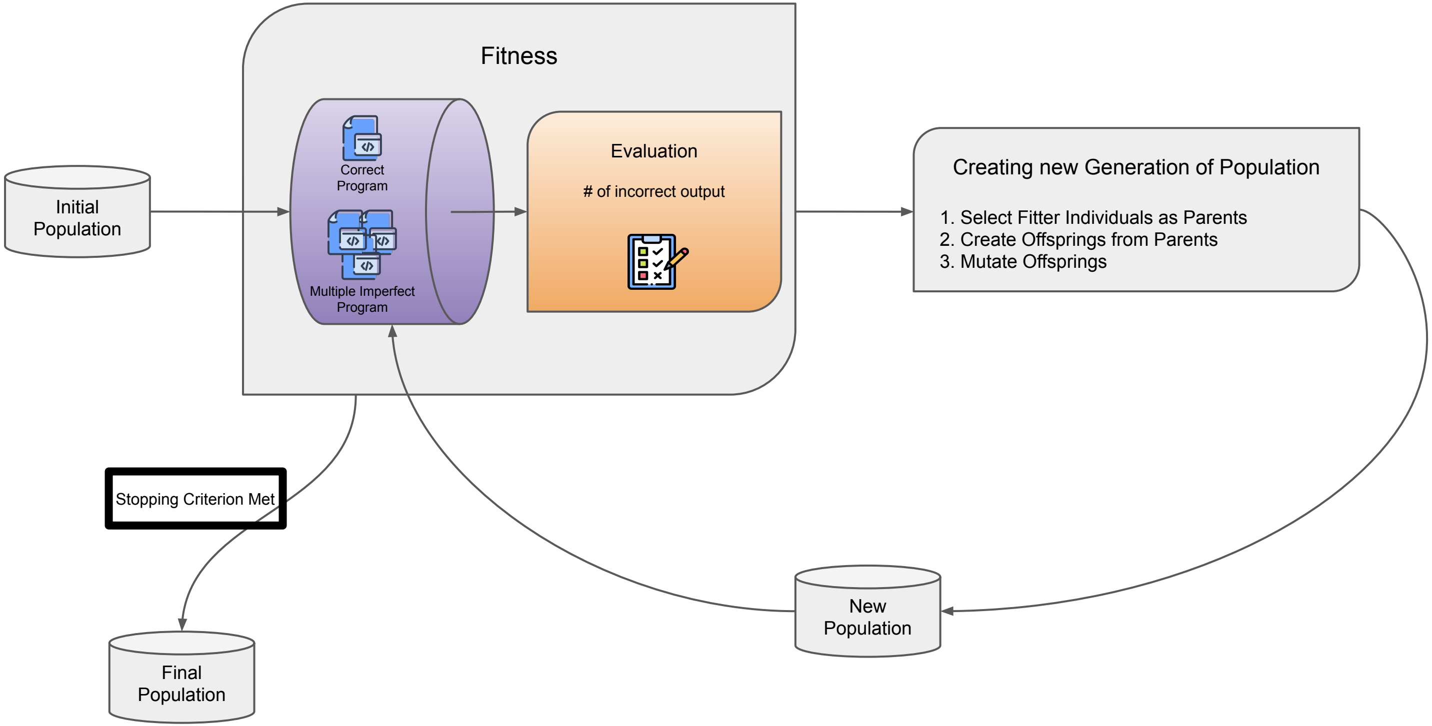
- If the assigned budget get out of limit, it is assigned with max budget value. (b_max = 50000)
  - b = min(b_max, b)

Mutation

- Mutate both budget and item properties (value and weight) randomly.
    - Apply +1 or -1 with a 20% probability.
    - Select a random value within the range with 80% probability.

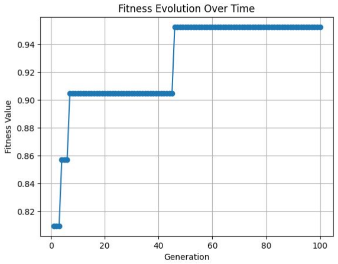
- Undo mutation if the value exceeds limits.

Stopping Criterion

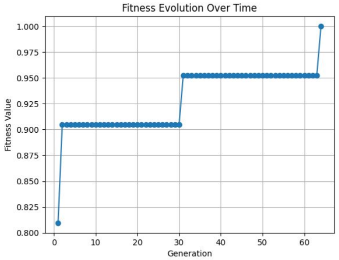


Stopping Criterion

- Maximum limit reached



- Fitness reaches 1



WHY?

Measure failure percentage, speed of reaching desired fitness

Experiment Setup - Variables

Variables

- Independent Variables
  - Place of Crossover
  - Method for Processing Budget during Crossover

- Controlled Variables
  - Stopping Criterion
  - Population Size
  - Selection Size
  - Crossover Rate
  - Mutation Rate
  - Knapsack Problem Variables

Experiment Setup - Randomization & Replication

- Randomization
  - Initial Population
    - Randomly generated based on the constraints of Knapsack Problem
  - Selection for Crossover and Mutation
    - Elements are randomly selected for crossover and mutation
    - Priority given to elements with higher fitness
  - Crossover and Mutation
    - Inherently random process applied to selected elements

Replication

Entire process is repeated for replication

Experiment Setup

- Measurement Tools
  - Fitness function
  - Runtime Analysis

- Data Collection Procedures
  - Collect data consistently at each generation into json file
  - Record state of each candidate problem and fitness score


```
{"ts_1_100": {"generation": 1, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, false, false], "fitness": 0.7142857142857143}}
{"ts_2_100": {"generation": 2, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_3_100": {"generation": 3, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_4_100": {"generation": 4, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_5_100": {"generation": 5, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_6_100": {"generation": 6, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_7_100": {"generation": 7, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_8_100": {"generation": 8, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_9_100": {"generation": 9, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, false], "fitness": 0.8571428571428571}}
{"ts_10_100": {"generation": 10, "tc_num": 5, "data_num": 7, "state": [true, true, true, true, true, true, true], "fitness": 1.0}}
```


Simplified example

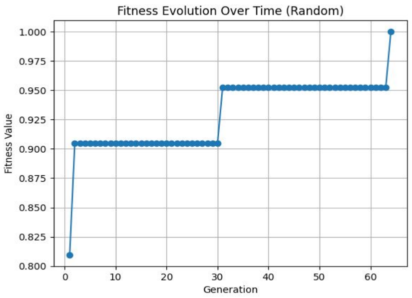
Result/Evaluation

- We conducted 10 experiments for each method.
  - Fitness 1 reach ratio before 100 generations: (4) > (1) = (2) > (3)
  - Average number of generations to reach fitness 1: (3) < (2) < (4) < (1)
  - Average number of generations reduced a lot when we applied (2) or (3).
    - Faster convergence

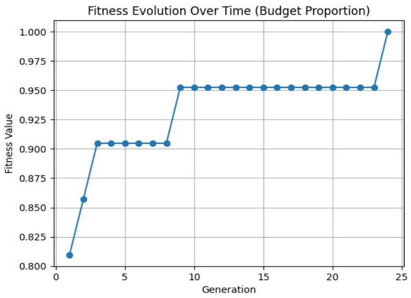
Method	(1) Random	(2) Budget Proportion	(3) Norm CP select	(4) Both (2) and (3)
Fitness 1 Reach Ratio Before Gen_Limit (%)	70%	70%	60%	80%
Average Generations to Reach Fitness 1	56.43 Gen	43.71 Gen	40.84 Gen	45.25 Gen

Result/Evaluation

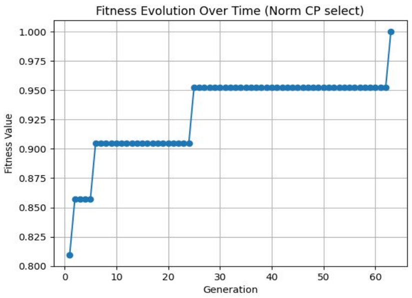
- Fitness evolution over time
  - Genes of all four methods have been evolved in the right direction.



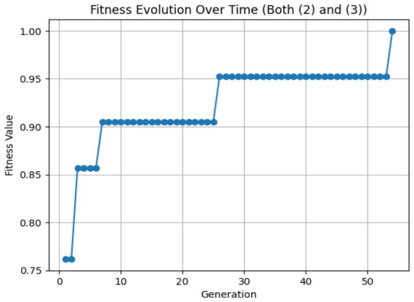
(1) Random



(2) Budget Proportion



(3) Norm CP select



(4) Both (2) and (3)

Conclusion

- We have shown that it is possible to find a set of edge cases for 0-1 knapsack using genetic algorithm.
- Evolving good inputs → find a set of inputs that can kill more programs
- Fitness function was the ratio of incorrect answers, and we can generate the set of edge cases with fitness 1.0 within the 100 generations.
- And in crossover, we proposed two new methods: Budget proportion & Norm CP select. These two methods make convergence faster.
- This study was limited to the 0-1 Knapsack problem. However, in the same way, edge cases can be found in other problems.

Thank You.