RAPORT FINALNY

PRZEDMIOT: PROGRAMOWANIE GENETYCZNE

Autorzy projektu

Katarzyna Stępień
Kacper Stankiewicz

Opiekun projektu

Dr inż. DARIUSZ PAŁKA

EAIiIB / Katedra Informatyki Stosowanej
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Kraków, Polska

01 lutego 2024 r.

Spis treści

Abstract

Niniejsza praca jest ostatecznym raportem z projektu stworzenia algorytmu genetycznego operującego nad własnym, stworzonym specjalnie na te potrzeby językiem.

W raporcie przedstawiona została zarówno implementacja, jak i wyniki testów przeprowadzonych na ostatnecznej wersji systemu.

Praca powstała przy pomocy *A Field Guide to Genetic Programming* autorstwa R. Poli, W. B. Langdon, oraz N. F. McPhee.

1. CEL I ZAKRES PROJEKTU

1.1 Cel projektu

Głównym celem projektu jest dogłęne poznanie mechanik algorytmów genetycznych oraz próba stworzenia jak najlepszej implementacji algorytmu dla własnego języka. Projekt zakłada stworzenie systemu składającego się z następujących komponentów:

- prostego języka programowania, pozwalającego na realizowanie podstawowych operacji;
- interpretera do tego języka;
- algorytmu genetycznego, tworzącego programy w tym języku;
- serii testów algorytmu genetycznego;

1.2 Oczekiwane rezultaty

Oczekiwanym rezultatem projektu jest stworzenie implementacji algorytmu genetycznego zdolnego tworzyć programy w specjalnie stworzonym języku zdolne rozwiązywać problemy zadane w serii testowej. Dodatkowymi cechami implementcji powinny być elastyczność oraz optymalność.

2. REALIZACJA PRAKTYCZNA

2.1 Założenia projektowe

2.1.1 Cel projektu

Celem projektu jest stworzenie algorytmu genetycznego działającego na specjalnie stworzonym, prostym języku.

2.1.2 Zakres projektu

Projekt składa się z prostego języka, zbudowanego dla niego interpretera oraz algorytmu genetycznego tworzącego programy w tym języku. Na stworzonym systemie przeprowadzone zostały różne testy. Implementacja pozwala na użycie systemu do dalszych, własnych testów oraz modyfikację używanej przez algorytm gramatyki (języka).

2.1.3 Ramy czasowe

Projekt był realizowany w ramach przedmiotu Programowanie Genetyczne, na V semestrze kierunku Informatyka i Systemy Inteligentne.

2.2 Oprogramowanie

2.2.1 Python

Do implementacji algorytmu wybrano język programowania Python ze względu na jego prostotę i czytelność, co przekłada się na efektywność w manipulowaniu listami, strukturami danych oraz obsługiwaniem algorytmów. Python jest językiem, który znacząco ułatwia proces tworzenia i zrozumienia kodu, co jest kluczowe w przypadku implemen-

tacji skomplikowanych algorytmów. Ponadto, ogromna społeczność Pythona dostarcza liczne biblioteki i narzędzia, co zwiększa elastyczność i efektywność procesu programowania.

Do wizualizacji statystyk wykorzystana została biblioteka `matplotlib`

2.2.2 ANTLR

ANTLR (ANother Tool for Language Recognition) to narzędzie do generowania analizatorów składniowych (parserów) oraz drzew syntaktycznych w różnych językach programowania. ANTLR umożliwia zdefiniowanie gramatyki za pomocą plików specyfikacji w notacji EBNF (Extended Backus-Naur Form), a następnie generuje kod źródłowy analizatora w wybranym języku programowania, takim jak Python. Wygenerowany analizator jest wykorzystywany do analizy i przetwarzania struktur językowych zgodnych ze specjalnie przygotowaną gramatyką.

2.3 Szczegóły implementacji

2.3.1 Prosty język stworzony na potrzeby algorytmu genetycznego

Na potrzeby algorytmu genetycznego stworzony został bardzo prosty język programistyczny pozwalający na wykonywanie podstawowych operacji: przypisania, wypisania, pętli, instrukcji warunkowej.

Oto krótka charakteryzacja kluczowych elementów tej gramatyki:

- **Tokeny:**
  - Podstawowe operatory arytmetyczne i logiczne: `ADD`, `SUBSTRACT`, `MULTIPLY`, `DIVIDE`, `MOD`, `EQ`, `LTHAN`, `GTHAN`, `EQEQ`, `NOTEQ`, `AND`, `OR` reprezentują powszechne operatory arytmetyczne i logiczne.
  - Nawiasy i klamry: `LPAREN`, `RPAREN`, `LCURL`, `RCURL` oznaczają różne rodzaje nawiasów.
  - Słowa kluczowe sterowania przepływem: `IF` i `WHILE` służą do definiowania instrukcji warunkowych i pętli.

  - Różne tokeny: `NEWLINE`, `PRINT`, `IDENTIFIER`, `INTLITERAL`, `INPUT`, `DOT` i `NOT` reprezentują odpowiednio znaki nowej linii, instrukcje drukowania, identyfikatory, literały całkowitoliczbowe, instrukcje wejścia, separator kropki i operator logicznego NOT.

- **Reguły leksera:**
  - `WS` reprezentuje białe znaki, które są pomijane podczas analizy składniowej.

- **Reguły parsera:**
  - Reguła programu (`prog`): Reprezentuje główną strukturę programu, umożliwiając wiele wyrażeń oddzielonych znakami nowej linii.
  - Reguła wyrażenia (`expr`): Definiuje różne rodzaje wyrażeń, w tym funkcje drukowania, instrukcje warunkowe, przypisania zmiennych, pętle while i znaki nowej linii.
  - Reguła przypisania zmiennej (`variable_assign`): Określa składnię przypisania zmiennych, pozwalając na wartości literałów lub instrukcji wejścia.
  - Reguła funkcji drukowania (`print_function`): Opisuje składnię funkcji drukowania.
  - Reguły operatorów i literałów (`operators`, `literals`): Określają składnię operacji arytmetycznych i logicznych oraz literałów, umożliwiając tworzenie złożonych wyrażeń.
  - Reguła typu porównania (`comparisson_type`): Reprezentuje różne typy operatorów porównania.
  - Reguła instrukcji warunkowej (`if_statement`): Definiuje strukturę instrukcji warunkowej, obejmującą opcjonalne zaprzeczenie (`NOT`).
  - Reguła warunku (`condition`): Określa składnię warunków używanych w instrukcjach warunkowych i pętlach while, umożliwiając logiczne i relacyjne kombinacje.
  - Reguła pętli while (`while_loop`): Opisuje składnię pętli while, obejmującą opcjonalne zaprzeczenie (`NOT`).

Pełna implementacja gramatyki widoczna jest poniżej (lis. 2.1):

```
1 WS: ' ' -> skip;
2 ADD: '+';
3 SUBSTRACT: '-';
4 MULTIPLY: '*';
```

```
5  DIVIDE:  '/';
6  MOD: '%';
7  EQ: '=';
8  LPAREN: '(';
9  RPAREN: ')';
10 LCURL: '{';
11 RCURL: '}';
12 LTHAN: '<';
13 GTHAN: '>';
14 EQEQ: '==';
15 NOTEQ: '!=';
16 AND: '&&';
17 OR: '||';
18 IF: 'if';
19 WHILE: 'while';
20 NEWLINE: [\r\n]+ ;
21 PRINT: 'print';
22 IDENTIFIER: ('a'..'z' | 'A'..'Z') ('0'..'9' | 'a'..'z' | 'A'..'
      Z')*;
23 INTLITERAL: ('-'? ('1'..'9')('0'..'9')*) | '0';
24 INPUT: 'input()';
25 DOT: '.';
26 NOT: '!';
27
28 prog: (expr NEWLINE*)*;
29
30 expr: (print_function DOT
31     | if_statement DOT
32     | variable_assign DOT
33     | while_loop DOT
34     | NEWLINE);
35
36
37 variable_assign:  IDENTIFIER EQ literals | IDENTIFIER EQ INPUT;
38
39 print_function: PRINT LPAREN literals RPAREN;
40
41 operators: MULTIPLY
42             | DIVIDE
43             | ADD
```

```
44              | MOD
45              | SUBSTRACT;
46
47 logic_operators: AND
48                | OR;
49
50 literals: IDENTIFIER | INTLITERAL | literals operators literals
      ;
51
52 comparisson_type: EQEQ
53           | GTHAN
54           | LTHAN
55           | NOTEQ;
56
57 if_statement: IF NOT? LPAREN condition RPAREN LCURL NEWLINE*
58                expr* NEWLINE*
59                RCURL;
60
61 condition: literals comparisson_type literals
62         | condition logic_operators condition;
63
64 while_loop: WHILE NOT? LPAREN  condition RPAREN LCURL NEWLINE*
65         expr* NEWLINE*
66         RCURL;
```

Listing 2.1: Gramatyka ANTLR dla języka użytego w implementacji GP

2.3.2 Klasa pomocnicza - GrammarReader

2.3.2.1 Opis Klasy

Klasa `GrammarReader` została stworzona w celu wczytywania i przetwarzania małych gramatyk. Rozpoznaje reguły tworzące terminale (`self.terminals`) i oddziela je od gramatyki (`self.rules`). Przypisuje liczby całkowite do każdej reguły (`self.TERMINAL_NUMBERS`, `self.RULE_NUMBERS`) oraz tworzy z nich reguły gramatyczne w celu optymalizacji pamięciowej, a także zwraca tabelę LUT *(ang. Look Up Table)* terminali do przyszłego tworzenia składni. Klasa przyjmuje jeden parametr (`self.terminal_start`) w celu kontrolowania dostępnego zakresu przypisywania numerów do reguł. Dzięki temu programy można przechowywać jako tablice

zmiennych typu integer, co drastycznie optymalizuje użycie pamięci przez algorytm, a następnie w prosty sposób przetwarzać je na czytelne dla człowieka słowa.

2.3.2.2 Metody Klasy

Klasa posiada następujące metody:

- `__init__(self, terminal_start: int = 100):`
  Konstruktor klasy, inicjalizuje obiekt klasy `GrammarReader`.
- `read_grammar(self, filename) -> None:`
  Metoda wczytuje gramatykę z pliku o nazwie `filename`. Przetwarza wczytaną gramatykę i inicjalizuje zmienne `self.terminals`, `self.rules`, `self.TERMINAL_NUMBERS`, `self.RULE_NUMBERS` oraz `self.GRAMMAR`.
- `get_start_rule(self) -> str:`
  Metoda zwraca początkową regułę gramatyki. Ignoruje regułę `prog: expr`, ponieważ może zostać uproszczona.
- `get_processed_grammar(self) -> dict:`
  Metoda zwraca przetworzoną gramatykę, składającą się tylko z liczb całkowitych, na podstawie `self.TERMINAL_NUMBERS` i `self.RULE_NUMBERS`.
- `get_terminal_table(self) -> dict:`
  Metoda zwraca tabelę przyporządkowującą numerom całkowitym terminale, przydatną do przyszłego parsowania.

2.3.3 Algorytm genetyczny - BiggerGP

Algorytm genetyczny stanowi kluczowy element systemu, który umożliwia ewolucję syntaktycznych struktur w poszukiwaniu optymalnych rozwiązań. W klasie BiggerGP zaimplementowano zarówno strukturę samego algorytmu, jak i kluczowe mechanizmy jego działania, takie jak selekcja, krzyżowanie i mutacja.

Podczas implementacji zwrócono uwagę na optymalizację procesu ewolucji, uwzględniając jednocześnie elastyczność algorytmu w dostosowywaniu się do różnorodnych problemów. Ponadto, omówiono wybór parametrów algorytmu oraz strategie manipulowania populacją i genotypami w celu osiągnięcia efektywnego przeszukiwania przestrzeni rozwiązań.

Więcej o klasie BiggerGP napisano w rozdziale 3. Rozdział skupia się również na interakcji z innymi modułami systemu, takimi jak analiza gramatyki, ocena dopasowania i generowanie statystyk. Wprowadzone koncepcje i rozwiązania mają na celu stworzenie spójnego i wydajnego środowiska do ewolucji programów komputerowych przy użyciu algorytmu genetycznego.

2.3.4 Testowanie systemu GP

Przeprowadzanie testów nad zbutowanym systemem GP polega na przekazaniu odpowiedniej funkcji fitnessu do algorytmu, który następnie będzie starał się znaleźć jak najbardziej optymalne rozwiązanie. Do przeprowadzania testów zastosowane zostały dwie funkcje pomocnicze:

- `test(function, ex, filename)` - funkcja przeprowadzająca pojedynczy cykl algorytmu dla konkretnej funkcji fitnesu. Jako paramatry przyjmuje: funkcję fitness, treść zadania, nazwę pliku, do którego zostanie zaspisana wizualizacja statystyk.
- `suite(functions, ex, filename)` - funkcja umożliwiająca sekwencyjne przeprowadzanie algorytmu dla kolejnych funkcji fitnessu. Pozwala na szybsze osiąganie lepszych wyników dzięki dostosowywaniu populacji do rozwiązywania kolejnych problemów. Działą tak samo jak `test`, ale przyjmuje listę funkcji fitnessu jako argument, zapisując statystyki dla ostatniego testu (domyślnie uznanego za docelowy).

3. IMPLEMENTACJA KLASY BiggerGP

3.1 Pola klasy BiggerGP

Klada posiada następujące pola:

- `MAX_LEN: int` - maksymalna długość programów - liczba pojedyńczych wyrażeń, z których może składać się program (wyrażenia wewnątrz wyrażeń typu if oraz while nie są liczone);
- `MAX_LOGIC_LEN: int` - maksymalna liczba warunków logicznych (łączonych operatorami logicznymi), jaka może wystąpić w warunku operacji `if` bądź `while`;
- `POP_SIZE: int` - wielkość populacji;
- `DEPTH: int` - maksymalna głębokość programu - liczba wyrażeń występujących "wewnątrz siebie" (w wyrażeniach typu if bądź while);
- `GENERATIONS: int` - liczba generacji, na którą będzie wykonywał się algorytm genetyczny;
- `MATCH_SIZE: int` - wielkość próbki osobników z której będzie wyłaniany zwycięsca w operacji turnieju;
- `MUTATION_RATE: int` - prawdopodobieństwo mutacji pojedynczego osobnika;
- `pop_fitness: list` - lista przechowująca wartości fitness dla każdego osobnika;
- `population: list` - lista przechowująca osobniki;
- `terminal_table: dict` - LUT (look up table) służąca do translacji programu zapisanego w postaci liczb całkowitych na zrozumiałą dla człowieka formę słów, zgodną z gramatyką języka;

- `grammar: dict` - reguły języka zapisane w postaci słownika list liczb całkowitych;
- `variables_start: int` - zmienna sygnalizująca, że liczby większe bądź równe jej są nazwami zmiennych;
- `int_literals_start: int` - zmienna sygnalizująca, że liczby większe bądź równe jej są liczbami całkowitymi użytymi w programie.
- `max_int: int` - największa wartość liczbowa, jaka może zostać użyta przez program;
- `node_starts: list` - lista ze znacznikami wskazującymi na początek pojedynczego wyrażenia w osobniku;
- `start: int` - słowo startowe gramatyki;
- `node_end: int` - znacznik wskazujący na koniec pojedynczego wyrażenia;
- `variables: dict` - słownik przypisujący odpowiednim liczbom całkowitym (większym bądź równym `variables_start`) symbole ASCII
- `variables_buffer: list` - lista przechowująca zmienne, które zostały stworzone w danym osobniku;
- `stats: list` - lista zbierająca statystyki każdej generacji;

3.2 Algorytm genetyczny - funkcja `evolve(fitness_f)`

Funkcja `evolve` jest odpowiedzialna za ewolucję populacji w ramach genetycznego algorytmu ewolucyjnego. Algorytm ten działa na populacji jednostek, każda reprezentowana jako sekwencja genów. Celem jest znalezienie jednostki o jak najwyższym współczynniku przystosowania (*fitness*) w kontekście danego problemu.

Parametry

- `fitness_f` – funkcja oceny przystosowania, przyjmująca jednostkę populacji jako argument.
- `populate` – flaga automatycznie ustawiona na True, dająca znać, czy dla danego cyklu program ma tworzyć nową generację, czy też przeprowadzać działania na już istniejącej. Pozwala to na przeprowadzanie testów sekwencyjnych.

Akcje

1. **Inicjalizacja populacji:** Funkcja rozpoczyna od stworzenia początkowej populacji.
2. **Ewolucja przez generacje:** Dla każdej generacji (od 0 do liczby generacji):
   (a) Obliczenie wartości przystosowania dla każdej jednostki w populacji przy użyciu funkcji `fitness_f`.
   (b) Sprawdzenie, czy problem został rozwiązany na podstawie wartości przystosowania. Jeśli tak, zwracane są statystyki i proces ewolucji kończy się.
   (c) W przeciwnym razie, aktualizacja statystyk i iteracja po populacji:
      i. Mutacja jednostki wyłonionej za pomocą `tournament` lub krzyżowanie dwóch rodziców, również wyłonionych za pomocą tej funkcji.
      ii. Obliczenie wartości przystosowania dla nowo powstałej jednostki.
      iii. Zastąpienie jednostki o najniższym przystosowaniu (wyłonionej za pomocą `negative_tournament()`) nowo powstałą jednostką.
3. Zwrócenie statystyk w przypadku, gdy problem nie został rozwiązany.

Zwracane wartości

W przypadku rozwiązania problemu, zwracane są statystyki ewolucji, w przeciwnym razie zwracane są statystyki końcowe po zadanej liczbie generacji.

Funkcje pomocnicze

Dla lepszego działania algorytmu genetycznego, w `evolve` używane są następujące funkcje pomocnicze:

- `populate_population()` - tworzy populację losowych osobników o wielkości `POP_SIZE`.
- `calculate_pop_fitness(fitness_f)` - oblicza wartość fitnesu dla każdego osobnika.
- `tournament` - losuje `MATCH_SIZE` osobników z populacji. Zwraca osobnika o większej wartości fitness.

- `tournament` - losuje `MATCH_SIZE` osobników z populacji. Zwraca osobnika o mniejszej wartości fitness
- `fitness(individual, fitness_f)` - oblicza wartość fitness dla konkretnego osobnika
- `get_stats(solved, generation)` - zbiera statystyki dla danej generacji, z wiedzą czy znaleziono rozwiązanie, czy nie. Używa klasy pomocniczej Stats.

3.3 Tworzenie nowego osobnika

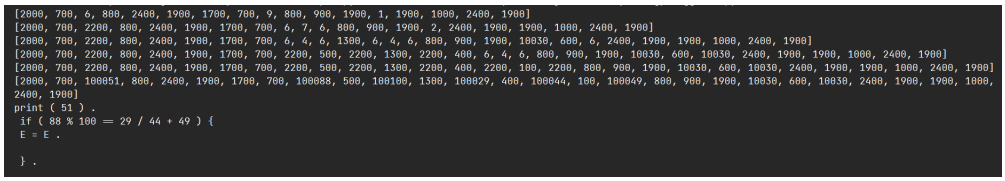
3.3.1 Metodyka działania

Tworzenie nowego osobnika w BiggerGP ma jak najbardziej przypominać wyprowadzenie gramatyczne przeprowadzone przez człowieka - program dostaje słowo początkowe i reguły gramatyki, według których losuje kolejne wyprowadzenia. Dzięki temu algorytm jest niewrażliwy na zmiany w gramatyce, co czyni go nie tylko uniwersalnycm, ale również optymalnym rozwiązaniem podczas dynamiczego tworzenia projektu.

Każde możliwe wyprowadzenie ma jednakowe prawdopodobieństwo bycia użytym przez algorytm, który ograniczony jest jedynie przez `MAX_LEN` oraz `MAX_DEPTH`. Dzięki temu nawet przy tych samych parametrach mogą powstawać programy zarówno mniejsze (rys. 3.1) jak i bardziej rozbudowane (rys. 3.2)

```
[10049, 600, 6, 2400, 1900, 2000, 700, 6, 800, 2400, 1900]
[10049, 600, 2200, 2400, 1900, 2000, 700, 6, 4, 6, 800, 2400, 1900]
[10049, 600, 2200, 2400, 1900, 2000, 700, 6, 4, 6, 500, 2200, 800, 2400, 1900]
[10049, 600, 2200, 2400, 1900, 2000, 700, 2200, 400, 2200, 500, 2200, 800, 2400, 1900]
[10049, 600, 100039, 2400, 1900, 2000, 700, 100052, 400, 100056, 500, 100050, 800, 2400, 1900]
X = 39 .
 print ( 52 / 56 % 50 ) .
```

Rysunek 3.1: Wyprowadzenie gramatyczne losowego osobnika - przykład minimalnego programu



Rysunek 3.2: Wyprowadzenie gramatyczne losowego osobnika - przykład większego programu

3.3.2 Użyte funkcje

3.3.2.1 Funkcja generate_random_individual

`generate_random_individual` jest metodą klasy, która generuje losową jednostkę na podstawie zmiennych.

3.3.2.2 Funkcja grow

`grow` rozbudowuje bufor (przeprowadza wyprowadzenia gramatyczne). Ze względu na powtarzalność część jej iteracji została przeniesiona do `traverse`. Przyjmuje parametr `buffer` – bufor (osobnik), który jest rozbudowywany.

3.3.2.3 Funkcja traverse

`traverse` to iteracja funkcji `grow`, wydzielona ze względu na optymalizację. Rozbudowuje bufer.

3.3.2.4 Funkcja check_new_rule

`check_new_rule` jest metodą klasy, która sprawdza specjalne przypadki podczas tworzenia reguł:

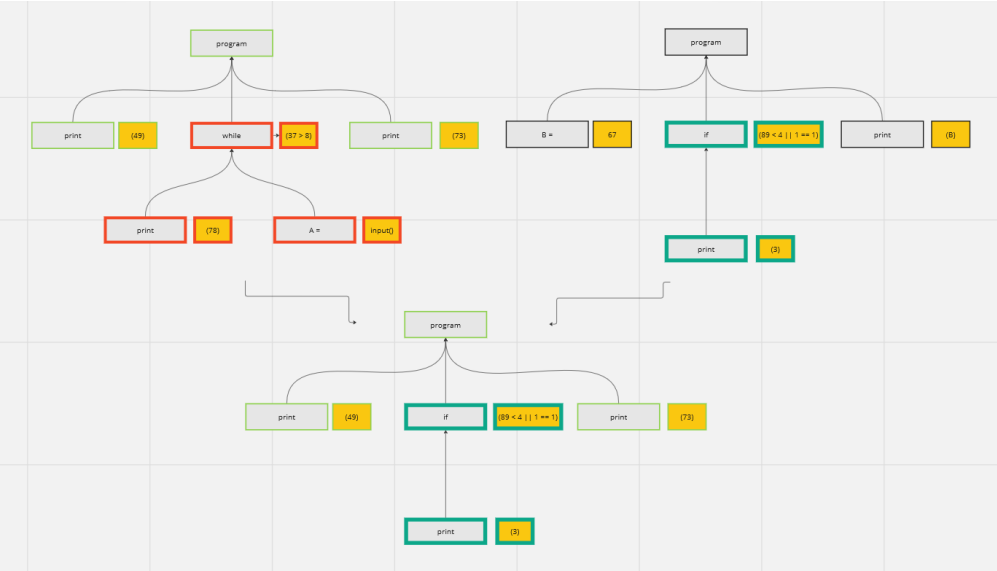
- Jeśli nowa reguła wymaga użycia zmiennej, sprawdzane jest, czy jakiekolwiek zmienne zostały już zadeklarowane przez jednostkę (w takim przypadku znajdują się w `variables_buffer`). Jeśli nie, zmienna zostaje zamieniona na liczbę całkowitą.
- Jeśli reguła tworzy zmienną, wybieramy jej nazwę spośród wszystkich możliwych (self.variables), dodajemy ją do zadeklarowanych zmiennych (self.variables_buffer) i natychmiast umieszczamy w nowej regule.

- Jeśli reguła jest warunkiem, sprawdzamy otoczenie, aby zliczyć, jak długo trwa już instrukcja logiczna. Jeśli jest dłuższa niż self.MAX_LOGIC_LEN, nie pozwalamy na dodanie kolejnych warunków do nowej reguły.

3.4 Operacja crossover

3.4.1 Metodyka działania

Operacja crossover polega na wymianie pojedynczej gałęzi (jednego wyrażenia) rodzica 1 na losową gałąź rodzica 2 (rys. 3.3). W obecnej wersji programu miejsce "przecięcia" w obu rodzicach jest dowolne, jako, że różnorodnosc wielkości w późniejszych generacjach programu została uznana za użyteczną - jednak pilnowanie wielkości bądź konkretnego miejsca krzyżowania nie byłoby problematyczne do zaimplementowania.



Rysunek 3.3: Przykład operacji crossover na niewielkich osobnikach

Gałęzie mogą być podmienione w dowolnym miejscu - również wewnątrz instrukcji typu `if` oraz `while`. Ze względu na poprawność syntaktyczną, algorytm sprawdza dodatkowo występowanie zmiennych - jeżeli w przeszczepionej gałęzi znajdują się obce zmienne, zostają one zamienione na zmienne zadeklarowane w rodzicu 1 bądź zamienione na liczby całkowite, jeżeli ten takowych nie posiada.

3.4.2 Użyte funkcje

3.4.2.1 Funkcja `crossover`

Funkcja `crossover` zwraca osobnika powstałego przez skrzyżowanie dwóch rodziców. Osobnik jest sprawdzany pod względem poprawności syntaktycznej.

3.4.2.2 Funkcja `find_index`

Funkcja `find_index` znajduje miejsce w osobniku, gdzie nastąpi "przecięcie" - używa do tego funkcji `find_closing_dot` oraz `find_closing_parentheses` - funkcje te oznaczają początek oraz koniec wyrażenia na podstawie jego typu. Dzięki temu zachowana jest poprawność syntaktyczna.

3.4.2.3 Funkcja `clean_individual`

Funkcja `clean_individual` zajmuje się "porządkowaniem" zmiennych. Jak zostało już wspomniane powyżej, zmienne przeszczepione z drugiego osobnika zostają zamienione na własne bądź zastąpione liczbami całkowitymi.

3.5 Mutacja

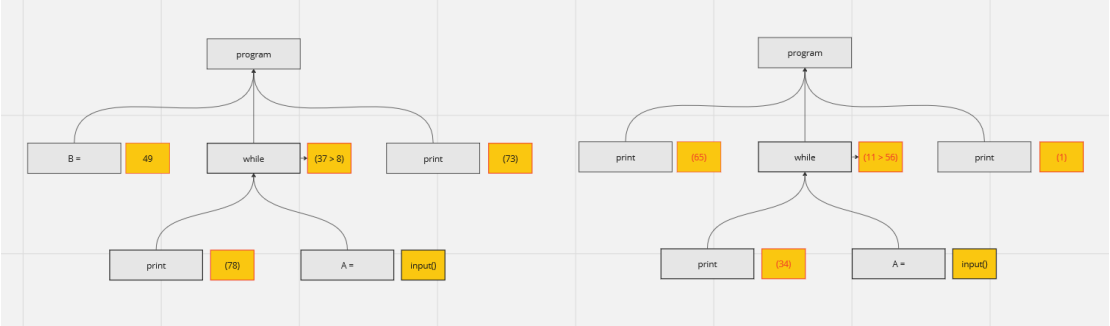
Mutacja zmienia losowy fragment osobnika. Aby zwiększyć różnorodność genetyczną, w programie zaimplementowane zostały cztery różne możliwe typy mutacji.

3.5.1 Mutacja nr. 0

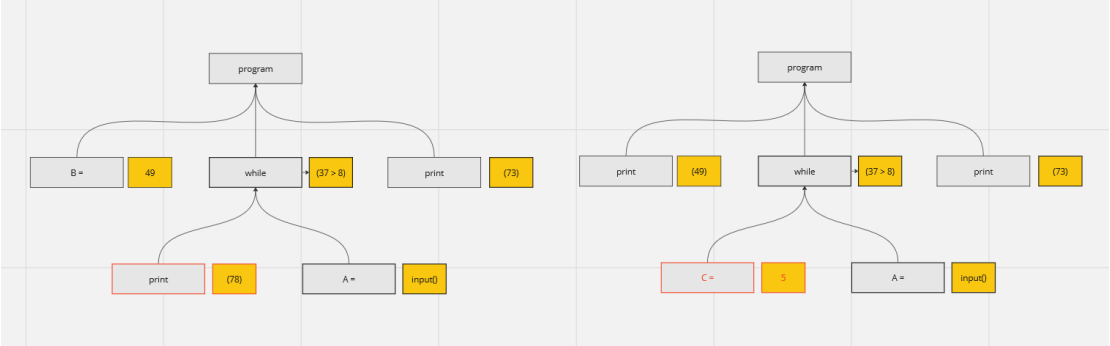
Mutacja nr. 0 polega na zmutowaniu wszystkich użytych wartości liczbowych w programie na nowe (rys. 3.4). Dzięki temu algorytmy szukające programów o odpowiednich działaniach matematycznych mają większe szanse na powodzenie.

3.5.2 Mutacja nr. 1

Mutacja nr. 1 podmienia losową gałąź programu na nową (rys. 3.5), z naciskiem na użycie jedynie tych zmiennych, które zostały stworzone przed punktem podmiany, bądź nowozadeklarowanych.



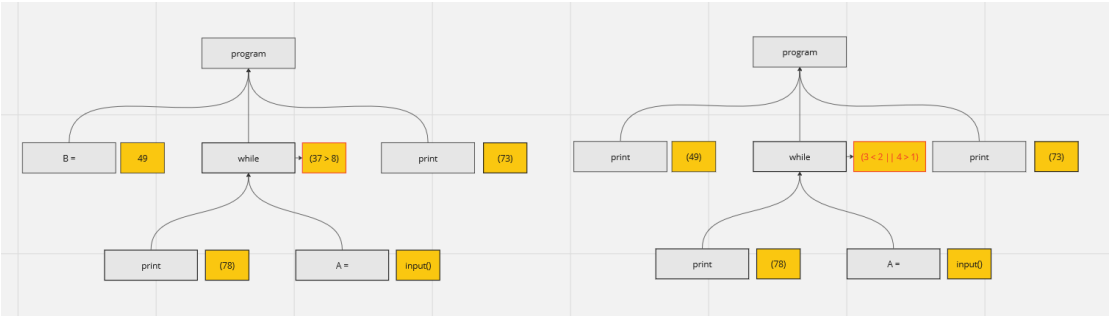
Rysunek 3.4: Mutacja - zamienienie wartości liczbowych



Rysunek 3.5: Mutacja - podmienienie gałęzi

3.5.3 Mutacja nr. 2

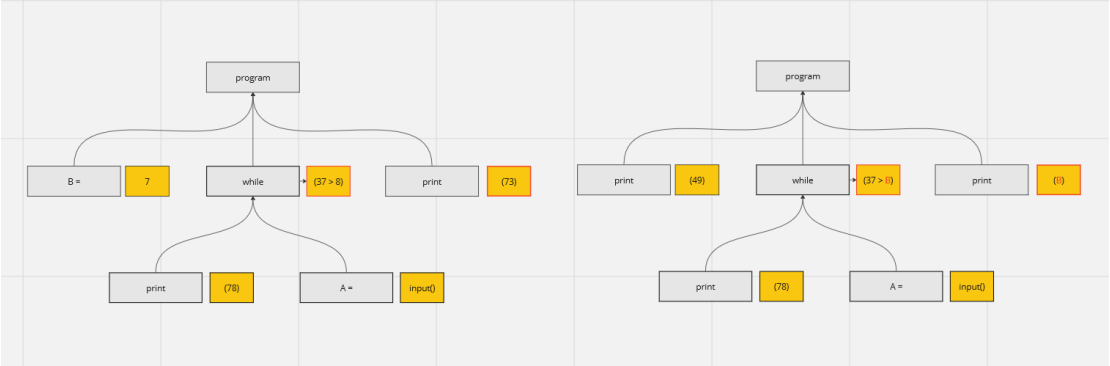
Mutacja nr. 2 podmienia zawartość wybranych losowo warunków logicznych oraz wyrażeń `print` (rys. 3.6) - jest to słowem mutacja "nawiasowa".



Rysunek 3.6: Mutacja - podmienienie zawartości nawiasu

3.5.4 Mutacja nr. 3

Mutacja nr. 3 zamienia losowo wybrane wartości liczbowe użyte w programie na zadeklarowane w nim zmienne, jeżeli takowe istnieją (rys. 3.7).



Rysunek 3.7: Mutacja - podmienienie wartości liczbowych na zmienne

3.6 Usprawnienia dla użytkownika

3.6.1 Metoda `toString()`

Metoda `toString()` zamienia osobniki zapisane w postaci list liczb całkowitych na czytelny dla człowieka program, używając operacji LUT, przy pomocy słownika `terminal_table`. Dzięki temu przechowywanie dużej ilości osobników jest bardziej optymalne pamięciowo.

3.6.2 Klasa pomocnicza `Stats`

`Stats` to niestandardowa klasa służąca do zbierania statystyk dla programu BiggerGP. Na jej podstawie generowane są wizualizacje statystyk działania algotymu przy pomocy biblioteki `matplotlib`.

Parametry

- `solved` – Flaga wskazująca, czy problem został rozwiązany (typ: `bool`).
- `generation` – Numer pokolenia (typ: `int`).
- `best_fit` – Najlepsza wartość dopasowania (typ: `float`).
- `best_indiv` – Najlepsza jednostka w danym pokoleniu (typ: `str`).
- `avg_fit` – Średnia wartość dopasowania w danym pokoleniu (typ: `float`).
- `pops` – Rozmiar populacji (typ: `int`).
- `d` – Głębokość (typ: `int`).

Metoda `to_string`

Metoda zwracająca tekstową reprezentację obiektu klasy `Stats`.

4. PODSTAWOWE TESTY SYSTEMU GP

Niniejszy rozdział skupia się na podstawowych testach zaimplementowanego systemu GP.

4.1 Wypisywanie liczb

4.1.1 Test 1.1.A - Program powinien wygenerować na wyjściu (na dowolnej pozycji w danych wyjściowych) liczbę 1. Poza liczbą 1 może też zwrócić inne liczby.

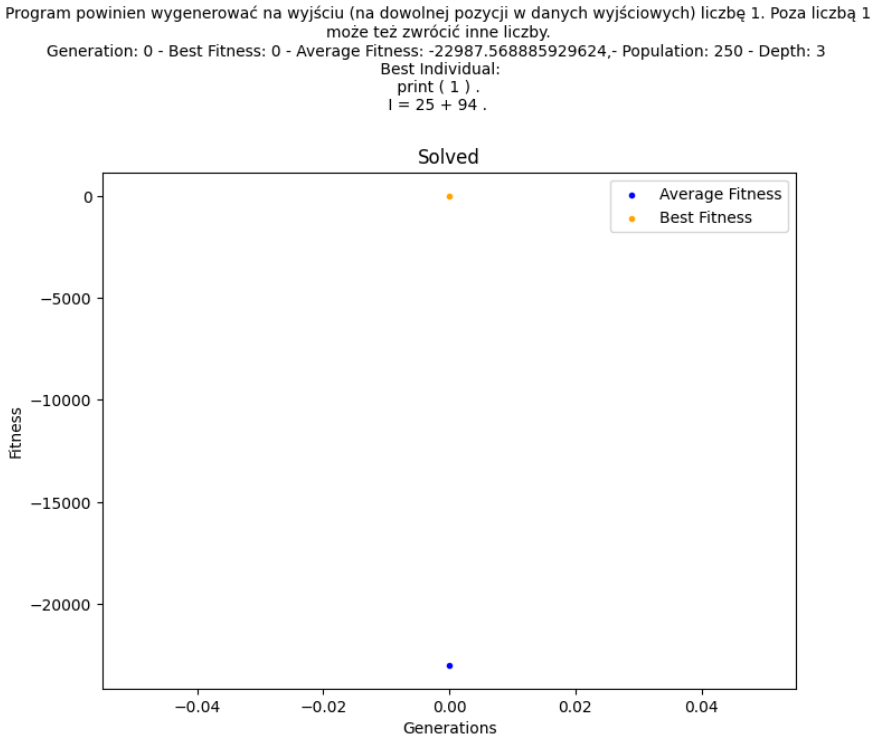
Jest to jeden z najprostszych testów. Algorytm znalazł odpowiedź natychmiastowo, w pierwszej generacji (rys. 4.1), w bardzo małej populacji - zostało użyte jedynie 250 osobników.

W teście użyto następującej funkcji fitness:

```
def fitness_function_1(program):
    try:
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        visitor = ExprVisitor()
        output, _ = visitor.visit(tree)
        if 1 in output:
            return 0
        else:
            return -10 * (abs(min(output)) + 1)
    except Exception as e:
        return -1000
```

Zadany test nie wymagał wyznaczania etapów uczenia. Programy otrzymywały fitness

wprost proporcjonalny do bliskości najmniejszej wartości z wyjścia do jedynki.



Rysunek 4.1: Statystyki testu 1.1.A

4.1.2 Test 1.1.B Program powinien wygenerować na wyjściu (na dowolnej pozycji w danych wyjściowych) liczbę 789. Poza liczbą 789 może też zwrócić inne liczby.

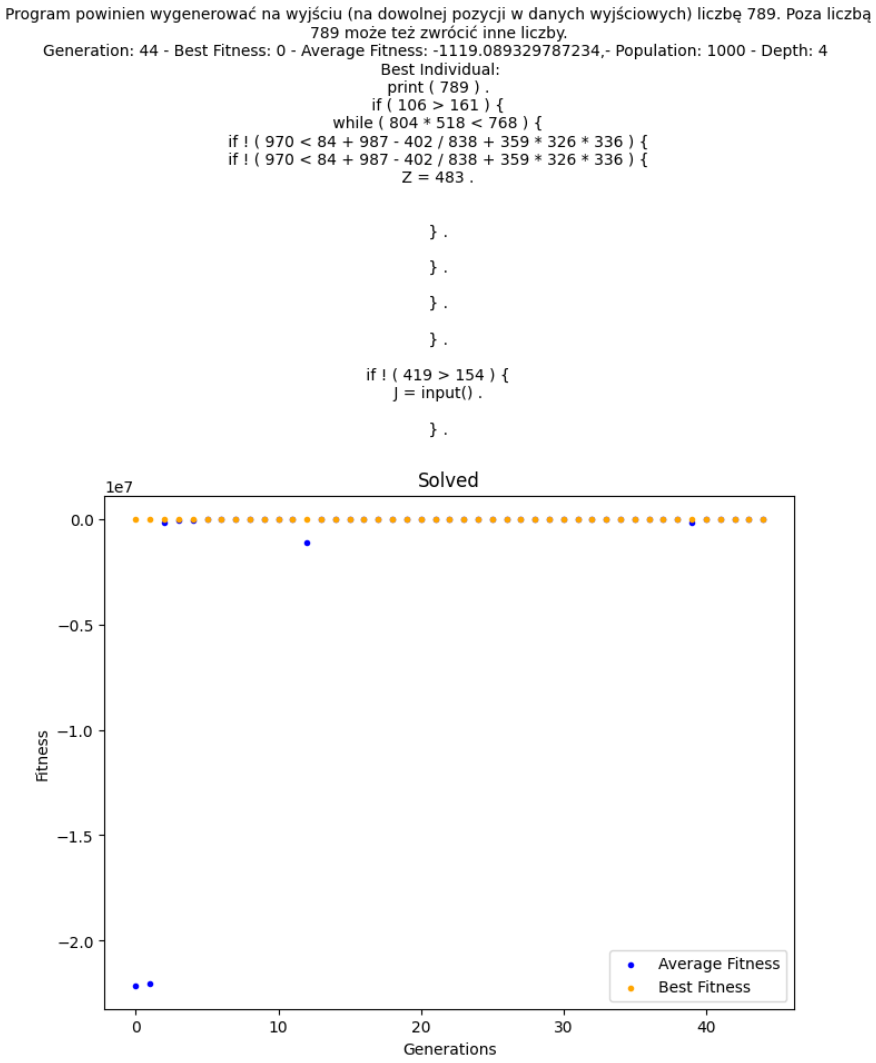
Ze względu na większy zakres możliwych wartości liczbowych stosowanych w programach, algorytm szukał zdecydowanie dłużej, niż w przypadku testu 1.1.A. Wciąż jednak osiągnął pożądany wynik w stosunkowo szybkim czasie i małej populacji - tysiąc osobników (rys. 4.2)

W teście użyto następującej funkcji fitness:

```
def fitness_function_2(program):
    try:
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        visitor = ExprVisitor()
        output, _ = visitor.visit(tree)
```

```
9         if 789 in output:
10            return 0
11        else:
12            if len(output) == 0:
13                return -10000
14            else:
15                return -10 * (sum([abs(x - 789) for x in output
    ]) / 10 + 1)
16     except TypeError as e:
17        return -1000
```

W teście wyznaczony został jeden dodatkowy etap uczenia - programy, które nie zwracały nic na wyjście dostawały najmniejszy fitness. Pozostałe były oceniane za bliskość sumy z wyjścia do punktu 789.



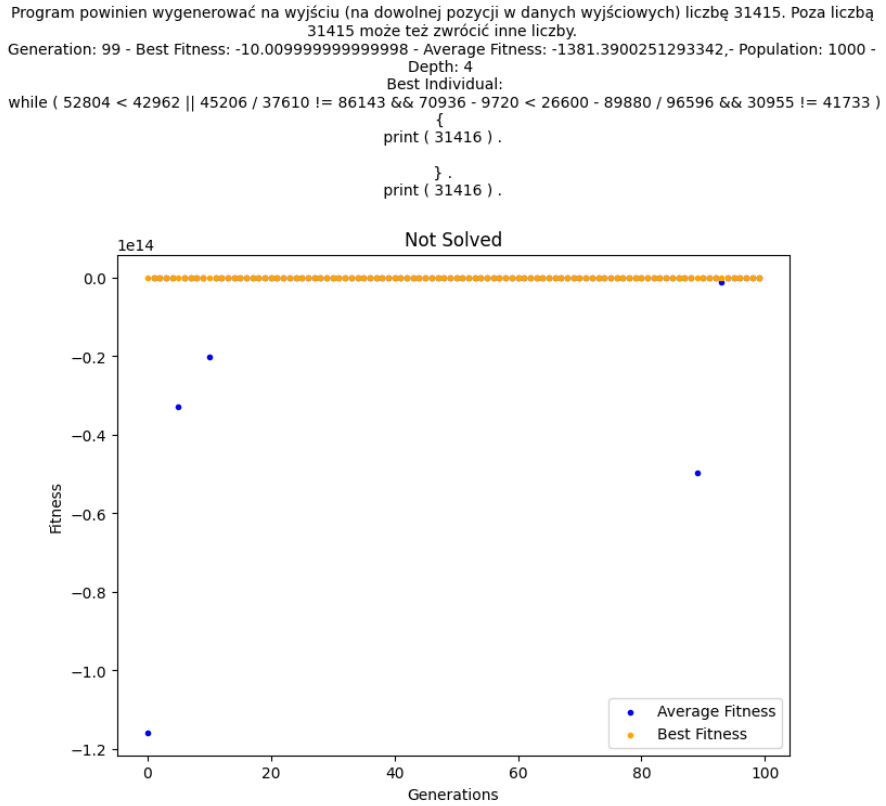
Rysunek 4.2: Statystyki testu 1.1.B

4.1.3 Test 1.1.C Program powinien wygenerować na wyjściu (na dowolnej pozycji w danych wyjściowych) liczbę 31415. Poza liczbą 31415 może też zwrócić inne liczby.

Algorytm nie znalazł idealnego rozwiązania, pomylił sie o 1 i nie udało nam się powtórzyć testu tak, żeby dostać oczekiwana wartość

```
    def fitness_function_3(program):
    try:
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        visitor = ExprVisitor()
        output, _ = visitor.visit(tree)
        if 31415 in output:
            return 0
        else:
            if len(output) == 0:
                return -1000
            else:
                return min(-10 * (min([abs(abs(x) - 31415) for
x in output]) + 1), -1000)
     except Exception as e:
        # print(e)
        return -1000
```

(rys. 4.3)



Rysunek 4.3: Statystyki testu 1.1.C

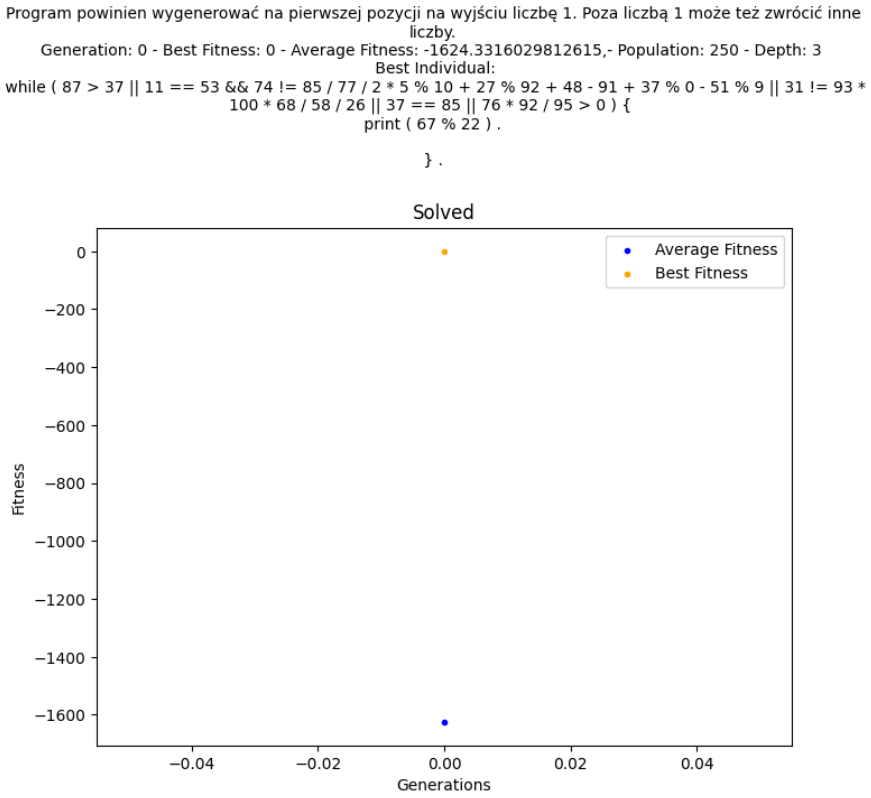
4.1.4 Test 1.1.D Program powinien wygenerować na pierwszej pozycji na wyjściu liczbę 1. Poza liczbą 1 może też zwrócić inne liczby.

Algorytm juz w 1 generacji (zerowej) znalazł oczekiwana wartość poprzez operacje modulo.

```
1    def fitness_function_4(program):
2    try:
3        lexer = ExprLexer(InputStream(program))
4        stream = CommonTokenStream(lexer)
5        parser = ExprParser(stream)
6        tree = parser.prog()
7        visitor = ExprVisitor()
8        output, _ = visitor.visit(tree)
9        if output[0] == 1:
```

```
            return 0
        else:
            return -10 * (abs(min(output)) + 1)
    except Exception as e:
        # print(e)
        return -1000
```

(rys. 4.4)



Rysunek 4.4: Statystyki testu 1.1.D

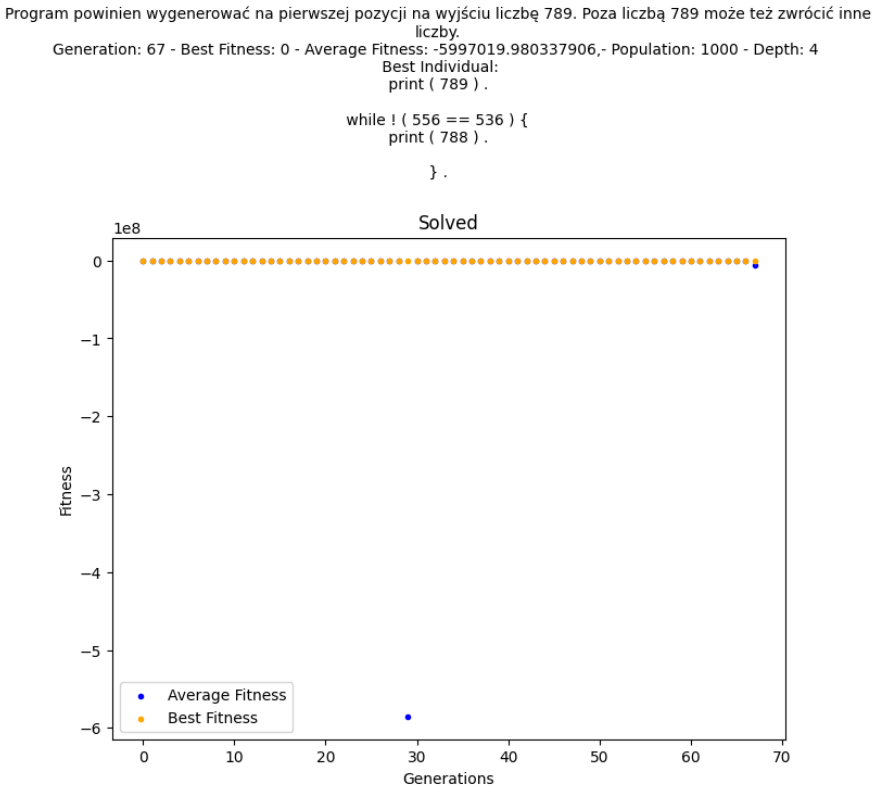
4.1.5 Test 1.1.E Program powinien wygenerować na pierwszej pozycji na wyjściu liczbę 789. Poza liczbą 789 może też zwrócić inne liczby.

Funkcja fitnessu w tym teście odrzucała automatycznie wszystkie wyniki, które nie miały liczby 789 na pierwszym wyniku.

```
    def fitness_function_5(program):
    try:
        lexer = ExprLexer(InputStream(program))
```

```
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        visitor = ExprVisitor()
        output, _ = visitor.visit(tree)
        if output[0] == 789:
            return 0
        else:
            if len(output) == 0:
                return -10000
            else:
                return -10 * (abs(output[0] - 789) / 100 + 1)
    except Exception as e:
        # print(e)
        return -1000
```

(rys. 4.5)



Rysunek 4.5: Statystyki testu 1.1.E

4.1.6 Test 1.1.F Program powinien wygenerować na wyjściu liczbę jako jedyną liczbę 1. Poza liczbą 1 NIE powinien nic więcej wygenerować.

Funkcja fitnessu karała algorytm za zwracanie więcej niż jednej wartości. Jeśli była tylko jedna wartość, ale różna od jedynki to był po prostu "mniej karany"

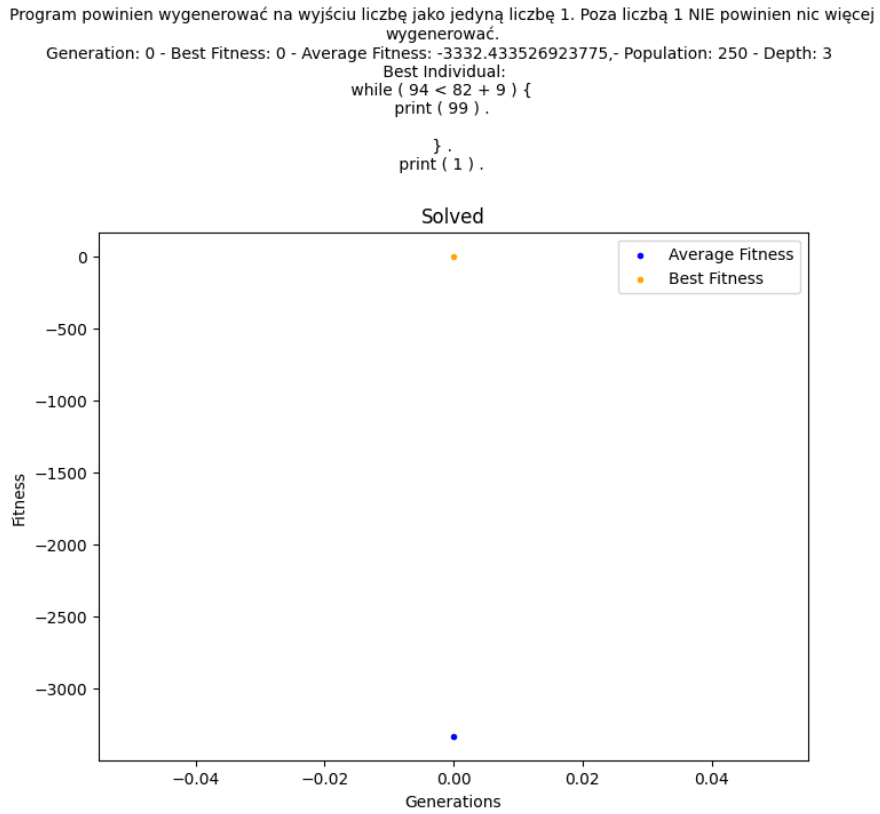
```
    def fitness_function_6(program):
    try:
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        visitor = ExprVisitor()
        output, _ = visitor.visit(tree)
        # print("Output: ", output)
        if output[0] == 1 and len(output) == 1:
            return 0
        elif len(output) == 1:
            return -10 * (abs(min(output)) + 1)
        else:
            return -100 * (abs(min(output)) + 1)
    except Exception as e:
        # print(e)
        return -1000
```

(rys. 4.6)

4.2 Proste działania arytmetyczne

4.2.1 Test 1.2.A Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) ich sumę. Na wejściu mogą być tylko całkowite liczby dodatnie w zakresie [0,9]

(rys. 4.7) Funkcja Fitnessu w tym przypadku było już troche bardziej skomplikowana. Żeby pomóc algorytmowi znaleźć odpowiedni algorytm nagradzaliśmy go za odpowiednie wyniki. Na przykład jeśli w wynikowym algorytmie był dwa $input()$ to algorytm był nagradzany, tak samo jak za użycie $print()$ z operacją dodawania w środku.



Rysunek 4.6: Statystyki testu 1.1.F

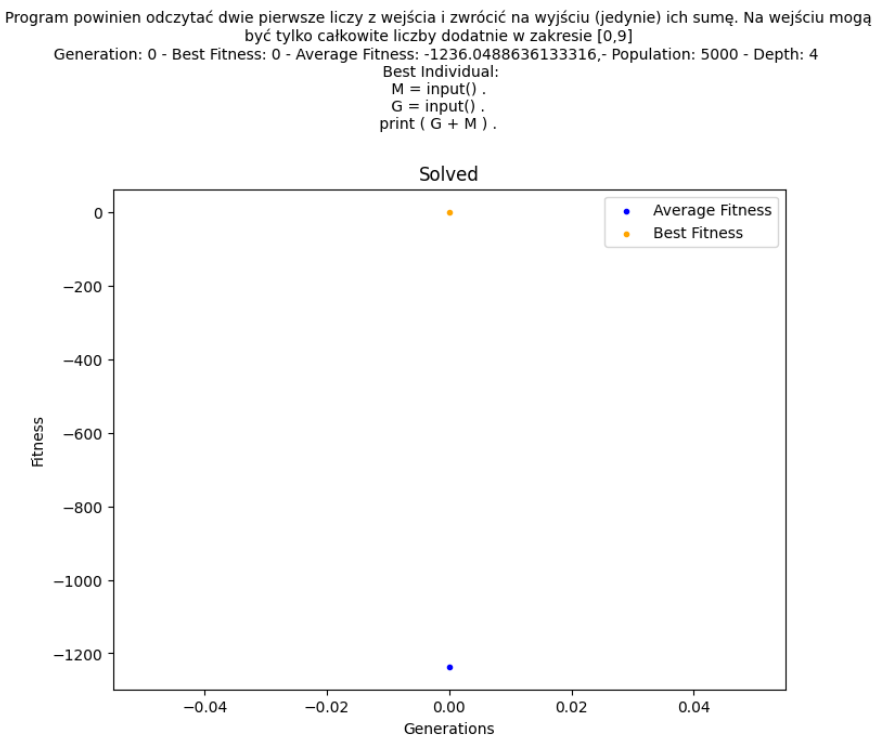
Ta sama funkcja fitnessu została użyta dla każdego testu 1.2.A, 1.2.B, 1.2.C, 1.2.D, 1.2.E, zmieniając tylko odpowiednie wartości w konstruktorze ExprVisitor i operator działania na odpowiedni zadany w poleceniu (linia numer 15 w kodzie)

```
1     def fitness_function_7(program):
2     fitness = 0
3     value = -1000
4     checks = 1
5     inputs_count = program.count("input()")
6     if inputs_count == 2:
7         value += 100
8         checks += 1
9     else:
10        return -1000
11    split_program = program.split(".")
12    if "input()" in split_program[0] and "input()" in
    split_program[1]:
13        value += 500
14        checks += 1
```

```
15      if any(["print" in x and "+" in x for x in split_program
    [1:]]):
16          value += 200
17          checks += 1
18      value = value / checks
19
20      try:
21          lexer = ExprLexer(InputStream(program))
22          stream = CommonTokenStream(lexer)
23          parser = ExprParser(stream)
24          tree = parser.prog()
25          num_readings = program.count("input()")
26          for _ in range(4):
27              visitor = ExprVisitor(20, -9, 9)
28              output, program_input = visitor.visit(tree)
29              # print("Output: ", output)
30              if output[0] == program_input[0] + program_input[1]
    and num_readings == 2 and len(output) == 1:
31                  print("inputs:", program_input)
32                  print("Outputs: ", output)
33                  fitness += 0
34              elif len(output) == 1:
35                  fitness += (value - len(program) / 10000)
36              else:
37                  fitness += 2 * (value - len(program) / 10000)
38          return fitness
39      except Exception as e:
40          # print(e)
41          return 8 * (value - len(program) / 10000)
```

4.2.2 Test 1.2.B Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) ich sumę. Na wejściu mogą być tylko całkowite liczby w zakresie [-9,9]

(rys. 4.8)



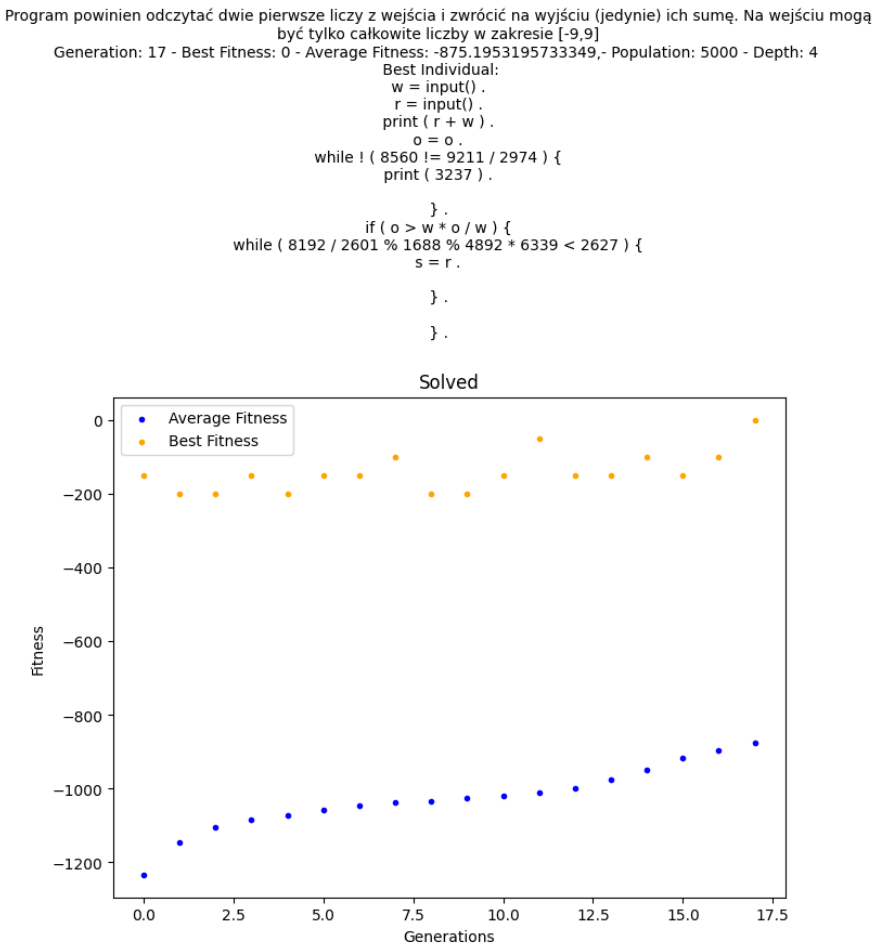
Rysunek 4.7: Statystyki testu 1.2.A

4.2.3 Test 1.2.C Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) ich sumę. Na wejściu mogą być tylko całkowite liczby dodatnie w zakresie [-9999,9999]

(rys. 4.9)

4.2.4 Test 1.2.D Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) ich różnicę. Na wejściu mogą być tylko całkowite liczby dodatnie w zakresie [-9999,9999]

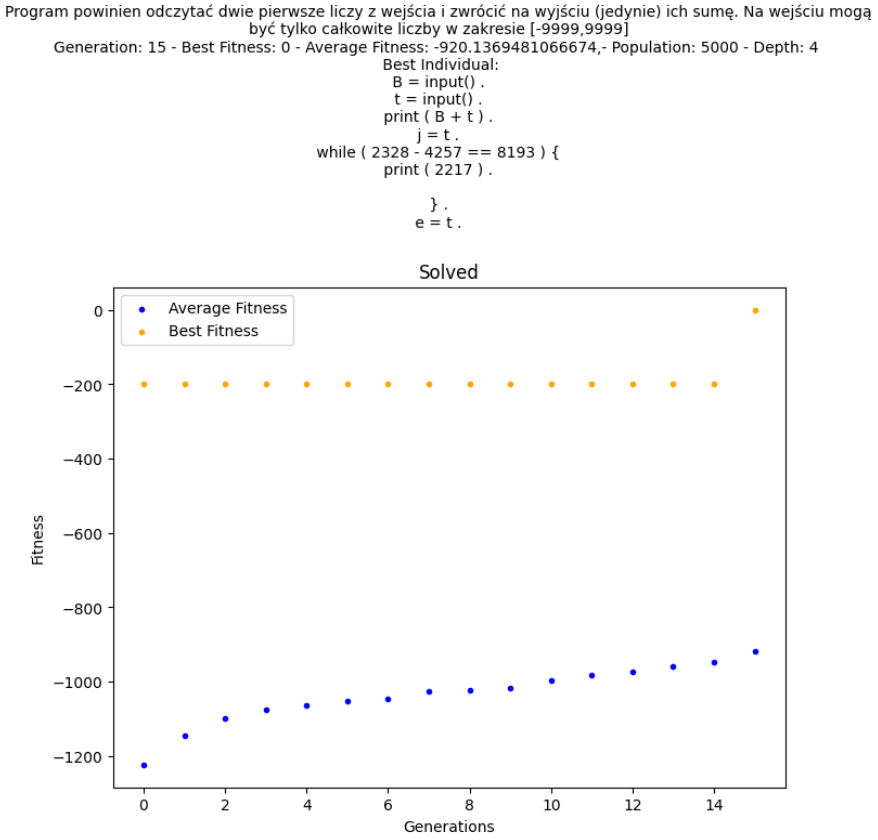
(rys. 4.10)



Rysunek 4.8: Statystyki testu 1.2.B

4.2.5 Test 1.2.E Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) ich iloczyn. Na wejściu mogą być tylko całkowite liczby dodatnie w zakresie [-9999,9999]

(rys. 4.11)



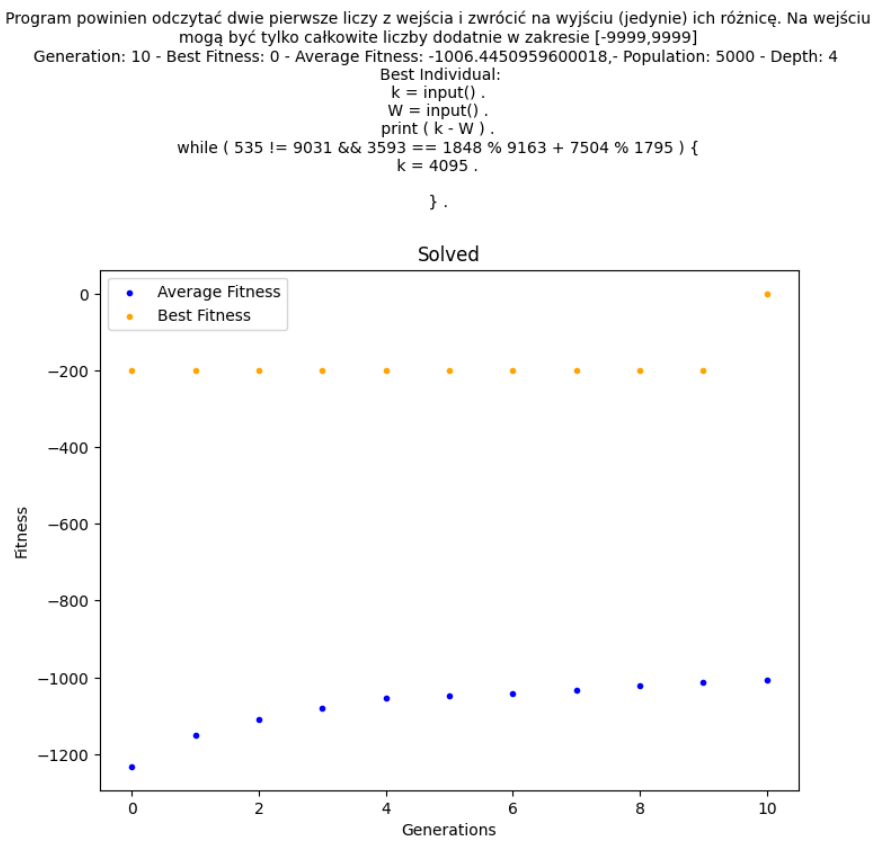
Rysunek 4.9: Statystyki testu 1.2.C

4.2.6 Test 1.3.A Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) większą z nich. Na wejściu mogą być tylko całkowite liczby dodatnie w zakresie [0,9]

(rys. 4.12) Funkcja fitnesuu nagradzała w tym teście odpowiednie wyniki algorytmu. Nagradza była odpowiednia ilość $input()$, brak pętli, warunek if z operatorem porównania. Karany był za nieopdowiednia ilość $inputs$, więcej lub mniej niz 1 if, brak zgodności z patternem (regexem), który odpowiadał za warunek if z porównaniem zmiennych i printem jednej z tych zmiennych w środku ifa. Taka sama funkcja fitnessu byla zastosowana dla testu 1.3.B

Do zdobycia odpowiednich wyników zasotsowane zostało uczenie sekwencyjne - najpierw uczyliśmy program, żeby znajdował pożądane struktury (while, input()).

```
1    def fitness_function_10(program):
2    fitness = 0
3    value = -1700
```

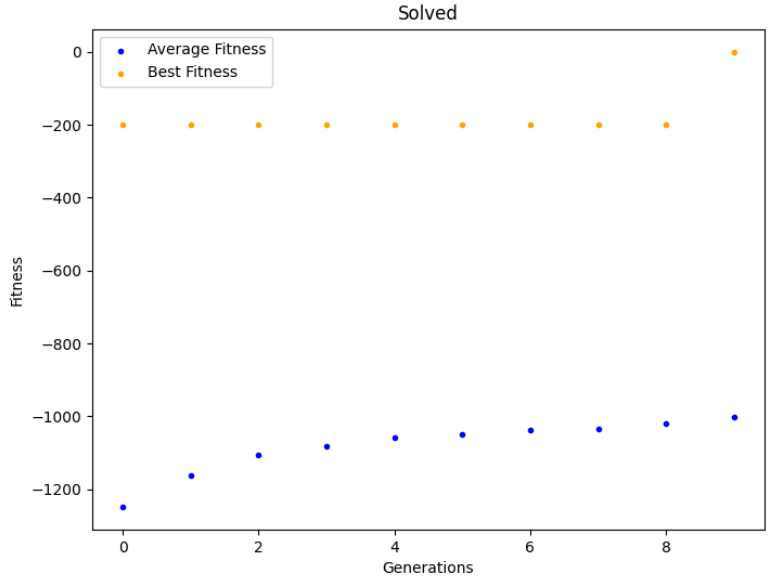


Rysunek 4.10: Statystyki testu 1.2.D

```
    checks = 1
    inputs_count = program.count("input()")
    if inputs_count == 2:
        value += 100
        checks += 1
    else:
        return -3000
    split_program = program.split(".")
    variables = [line.strip()[0] for line in split_program if "input()" in line]
    if len(variables) != 2:
        return -2700
    if "input()" in split_program[0] and "input()" in split_program[1]:
        value += 500
        checks += 1
    if any([variables[0] in x and variables[1] in x and "=" in x and "while" not in x and "if" not in x for x in
            split_program[1:]]):
```

Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) ich iloczyn. Na wejściu mogą być tylko całkowite liczby dodatnie w zakresie [-9999,9999]
Generation: 9 - Best Fitness: 0 - Average Fitness: -1002.873489406668,- Population: 5000 - Depth: 4
Best Individual:

```
j = input() .
z = input() .
while ! ( 2983 - 6808 != 3567 * 5128 || 4943 > 1166 && 6828 / 3958 * 3231 - 4293 * 9589 / 9994 + 8470 == 9361
|| 4467 + 4723 > 2303 ) {
if ! ( 8041 % 7013 / 3578 % 9340 + 3712 - 3835 % 51 - 7330 != 4641 ) {
if ( 7252 % 7370 == 4697 * 6879 && 4794 - 6096 > 9898 || 1511 - 7093 * 2039 - 6791 / 7871 * 2475 != 2047 ||
5645 != 9174 ) {
while ! ( 8673 / 4949 > 5032 / 4006 ) {
while ! ( 6357 == 2916 && 4992 > 5756 && 3505 != 2750 - 8399 / 3378 % 310 * 6701 / 5568 % 2261 ) {
while ( 6182 / 1948 < 9278 || 6500 < 411 % 3424 && 7461 + 4277 - 5519 % 2002 / 6584 / 3218 < 7181 ) {
print ( 9835 + 5252 ) .

} .

} .

} .

} .

} .

} .
while ! ( 4806 + 3482 * 7847 < 84 % 5039 % 1337 * 4337 * 8276 && 481 * 4257 + 8814 / 6169 - 2452 % 7171 !=
4962 % 9278 * 4294 ) {
if ( 2393 + 1949 * 5076 - 1834 / 4221 % 1450 + 2984 > 9942 ) {
print ( 3868 - 4397 ) .

} .

} .
if ( 2168 < 5928 ) {
print ( z * j ) .

} .
while ( z - j - 7794 * j % 2359 + 2259 > 9466 || z == j ) {
while ! ( 9937 < 3232 * 8573 ) {
print ( 1654 * 967 ) .

} .

} .
```



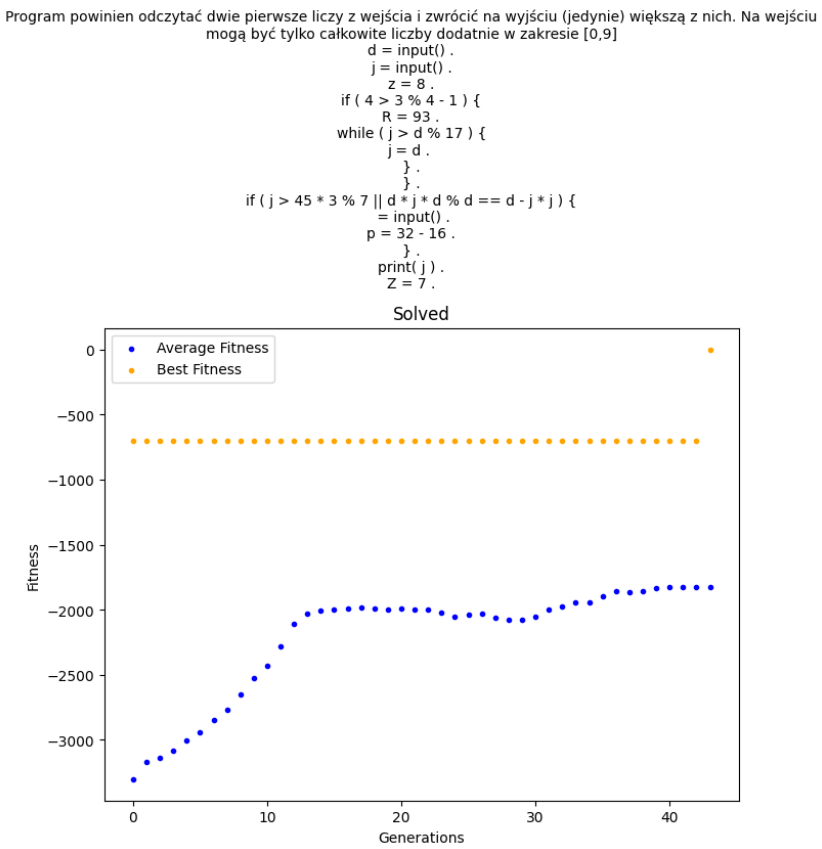
Rysunek 4.11: Statystyki testu 1.2.E

```
20          value += 500
21          checks += 1
22      if any(["print" in x and (variables[0] in x or variables[1]
        in x) for x in split_program[1:]]):
23          value += 100
```

```
24          checks += 1
25      value = value / checks
26
27      try:
28          lexer = ExprLexer(InputStream(program))
29          stream = CommonTokenStream(lexer)
30          parser = ExprParser(stream)
31          tree = parser.prog()
32          num_readings = program.count("input()")
33          for i in range(15):
34              visitor = ExprVisitor(2, 0, 9, [abs(9 - i), abs(i
    -6)])
35              output, program_input = visitor.visit(tree)
36              # print("Output: ", output)
37              if output[0] == max(program_input[0], program_input
    [1]) and num_readings == 2 and len(
38                      output) == 1:
39
40                  fitness += 0
41              elif len(output) == 1:
42                  fitness += (value - len(program) / 10000)
43              else:
44                  fitness += 2 * (value - len(program) / 10000)
45          return fitness
46      except Exception as e:
47          # print(e)
48          return 8 * (value - len(program) / 10000)
```

4.2.7 Test 1.3.B Program powinien odczytać dwie pierwsze liczy z wejścia i zwrócić na wyjściu (jedynie) większą z nich. Na wejściu mogą być tylko całkowite liczby w zakresie [-9999,9999]

Dla zwiekszonego zakresu wartości kod wygląda dokładnie tak samo jak na rysunku 1.3.A 4.12



Rysunek 4.12: Statystyki testu 1.3.A

4.2.8 Test 1.4.A Program powinien odczytać dziesięć pierwszych liczy z wejścia i zwrócić na wyjściu (jedynie) ich średnią arytmetyczną (zaokrągloną do pełnej liczby całkowitej). Na wejściu mogą być tylko całkowite liczby w zakresie [-99,99]

Pomimo prób dla testu 1.4.A i 1.4.B nie udało się wyznaczyć nic sensownego. Funkcja fitnessu która stosowaliśmy jak we wcześniejszych testach nagradzała za odpowiednią ilość inputów, za znaki dodawania i przyrównywania (w celu zwiekszenia sumy). Wyglądała ona następująco:

```
    def fitness_function_11(program):
    fitness = 0
    value = -2500
    checks = 1
    inputs_count = program.count("input()")
    value += max(inputs_count * 100, 1000)
    split_program = program.split(".")
```

```
    variables = [line[0] for line in split_program if "input()" in line]
    if all(["input()" in x for x in split_program[:10]]):
        value += 500
        checks += 1
    if any(["=" in x and "+" in x and all([v in x for v in variables]) and "while" not in x and "if" not in x for x in split_program[10:]]):
        value += 500
        checks += 1
    for line in split_program:
        value += line.count("+")
    if any(["print" in x for x in split_program[10:]]):
        value += 100
        checks += 1
    value = value / checks

    try:
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        num_readings = program.count("input()")
        for _ in range(15):
            visitor = ExprVisitor(20, -99, 99)
            output, program_input = visitor.visit(tree)
            if output[0] == sum(program_input) and num_readings == 10 and len(output) == 1:
                print("inputs:", program_input)
                print("Outputs: ", output)
                fitness += 0
            elif len(output) == 1:
                fitness += (value - len(program) / 10000)
            else:
                fitness += 2 * (value - len(program) / 10000)
        return fitness
    except Exception as e:
        # print(e)
        return 8 * (value - len(program) / 10000)
```

```
Best Individual:
P = input() .
t = input() .
while ! ( P < P ) {
if ! ( P == P + P - 3 / P / 54 ) {
f = input() .

} .

} .
while ! ( 47 == P + 82 || 78 < f % 97 ) {
print ( 45 ) .

} .
if ! ( 21 / f / f < 41 ) {
while ( f - f - 19 + f % 50 < f && f > 59 + f + f * 1 - 7 && f * P != 58 % 66 ) {
if ( P != P + 21 - 95 && P == f / P / 26 && 44 + P * 40 % f < 45 ) {
while ! ( 54 * f > f ) {
while ! ( P - 79 < 55 ) {
while ! ( 21 != 53 || 56 - P - P - P + 66 % 100 > f && f + f > P ) {
if ( P < 96 % 15 ) {
if ! ( 58 < 49 ) {
t = f .

} .

} .

} .

} .

} .
```

Rysunek 4.13: Enter Caption

Test 1.4.A nie dał nam żadnego sensownego rezultatu, dlatego nie było większego sensu próby uzyskania algorytmu 1.4.B który bazował na algorytmie 1.4.A.

5. BENCHMARKI

5.1 Benchmark 1 - Given an integer and a float, print their sum

5.1.1 Wersja bez liczb typu float zaimplementowanych do języka

Na potrzeby tego testu użyty został input w postaci: liczba całkowita, licznik, mianownik. Dane wejściowe mieszczą się w zakresie [1,9].

W teście użyto następującej funkcji fitness:

```
def benchmark1(program):
    fitness = 0
    value = -1500
    inputs_count = program.count("input()")
    if inputs_count == 3:
        value += 100
    split_program = program.split(".")
    variables = [line[0] for line in split_program if "input()"
    in line]
    if "input()" in split_program[0] and "input()" in
   split_program[1] and "input()" in split_program[2]:
        value += 600
    if any(["print" in x and "/" in x for x in split_program
   [2:]]):
        value += 100
    if any(["print" in x and "+" in x for x in split_program
   [2:]]):
        value += 100
    if any(["print" in x and "+" in x and "/" in x for x in
   split_program[2:]]):
        value += 200
```

```
    if any(all(v in x for v in variables) and "print" in x for x in split_program[2:]):
        value += 100
    index = 0
    for i in range(0, len(split_program)-1):
        if "print" in split_program[i]:
            index = i
            break
    value = value - index

    try:
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        num_readings = program.count("input()")
        for _ in range(8):
            visitor = ExprVisitor(20, 1, 9)
            output, program_input = visitor.visit(tree)

            if output[0] == program_input[0] + program_input[1]/program_input[2] and num_readings == 3 and len(output) == 1:
                print("inputs:", program_input)
                print("Outputs: ", output)
                fitness += 0
            elif len(output) == 1:
                fitness += (value - len(program) / 10000)
            else:
                fitness += 2 * (value - len(program) / 10000)
        return fitness
    except Exception as e:
        # print(e)
        return 8 * (value - len(program) / 10000)
```

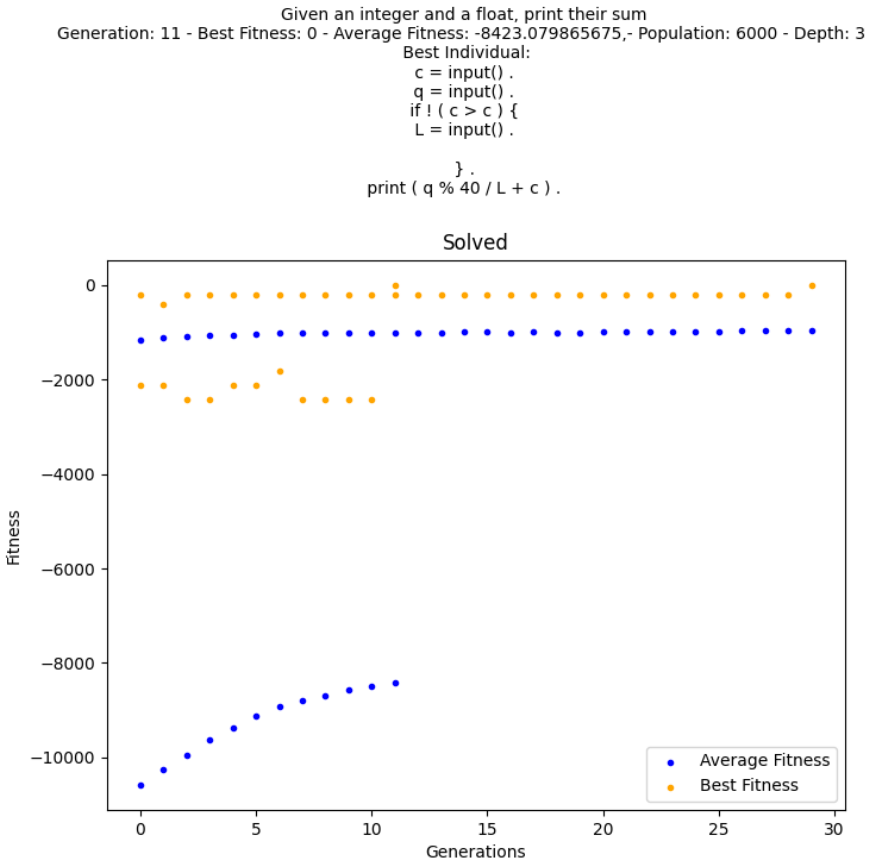
Jest to bardzo rozbudowana funkcja - program jest nagradzany za bycie podobnym do rozwiązania na różne sposoby:

- odpowiednia ilość czytania z wejścia
- odpowiednia długość wyjścia

- odpowiednie operacje arytmetyczne przy wypisywaniu na wyjście
- użycie wszystkich zadeklarowanych zmiennych przy wypisywaniu na wyjście
- jak szybko cokolwiek zostanie wypisane na wyjście
- długość

Złożony system nagród pozwalał na szybsze osiąganie pożądanych wzorców w populacji.

Ze względu na zaskakująco duże trudności w uzyskaniu pożądanego efektu w teście zastosowane zostało uczenie sekwencyjne - algorytm dostał najpierw za zadanie znalezienie sumy dwóch liczb całkowitych. Nawet mimo to znalezienie rozwiązania (rys. 5.1) zajęło algorytmowi kilka podejść.



Rysunek 5.1: Statystyki pierwszego benchmarka - wspólny wykres pierwszego i drugiego uczenia

Najciekawszym zachowaniem algorytmu zaobserwowanym podczas przeprowadzaina testów był fakt, że algorym stosunkowo wcześnie zaczynał oscylowac wokół pożądanego wzorca - a + b/c (rys. 5.2), nawet pomimo dość niskiego współczynnika dostoso-

wania - faktyczne wyniki zwracane na wyjście przez program były skrajnie różne od celu, co skutkowało karami.

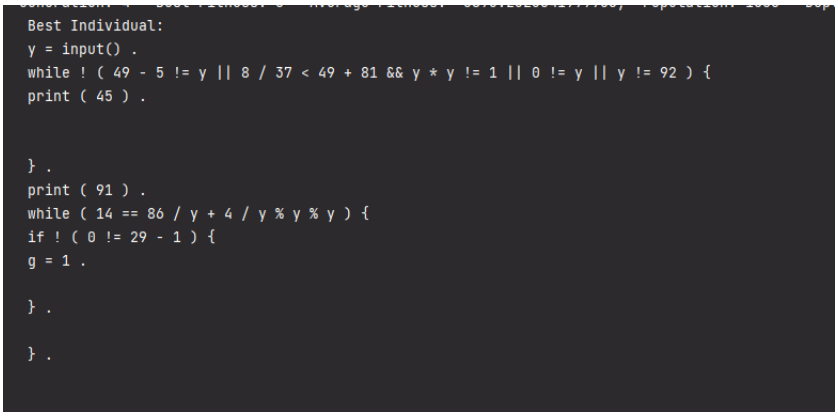
```
GENERATION  33
inputs: [7, 7, 7]
Outputs:  [8.0]
Generation: 33 - Best Fitness: -2121.0510999999997 - Average Fitness: -8482.589687191537,- Population: 6000 - Depth: 3
 Best Individual:
 U = input() .

 q = input() .
 q = input() .
 print ( q + q / U ) .

GENERATION  34
Generation: 34 - Best Fitness: -3224.0584 - Average Fitness: -8491.433961799794,- Population: 6000 - Depth: 3
 Best Individual:
 A = input() .
 q = input() .
 Y = input() .
 print ( 12 + 4 / 71 ) .
```

Rysunek 5.2: Algorytm znajdujacy programy bardzo bliskie rozwiązaniu

5.1.2 Benchmark 2 - Given the integer n, return the sum of squaring each integer in range[1, n]

Niestety ten benchmark nie został zaliczony. Nie udało się znaleźć nic sensownego. Funkcja fitnessu nagradzała za odpowienią ilosc *input* i znaki mnozenia.

```
    def benchmark2(program):
    split_program = program.split(".")
    value = -1000
    checks = 1
    if program.count("inputs()") != 1:
        value -= 600
    else:
        value += 200
        checks += 1

    if any(["print" in x and "*" in x for x in split_program
[1:]]):
        value += 200
        checks += 1
    if "input()" in split_program[0] and "input()" in
split_program[1] and "input()" in split_program[2]:
        value += 200

    value = value / checks

    try:
```

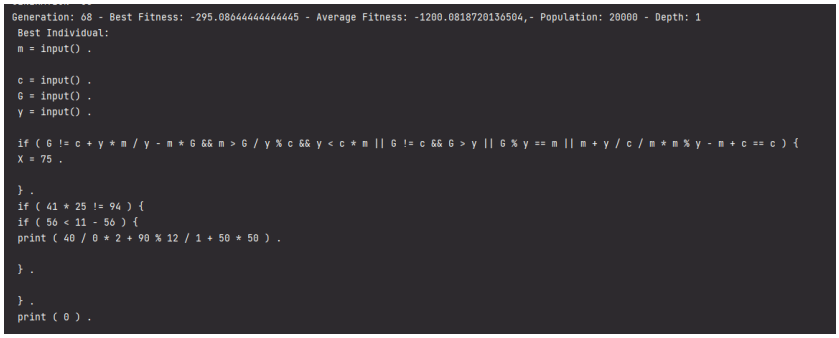


Rysunek 5.3: Enter Caption

```
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
        visitor = ExprVisitor()
        fitness = -1000
        if program.count("while") == 0:
            return -10_000
        for _ in range(3):
            output, inputs = visitor.visit(tree)
            num_reading = program.count("input()")
            proper_value = 0
            for i in range(1, inputs[0] + 1):
                proper_value += i ** 2

            if num_reading == 1 and proper_value in output:
                print("Inputs: ", inputs)
                print("Outputs:", output)
                print("Proper val: ", proper_value)
                return 0
            else:
                fitness += (value - len(program) / 10000)
        return fitness
    except Exception as e:
        return -10_000
```

5.1.3 Benchmark 3 - Given 4 intergers find smallest of them

Rysunek 5.4: Proby uzyskania wyniku zgodnie z benchmarkiem nr 3



Rysunek 5.5: Enter Caption

Rysunek 5.6: Dalsze proby uzyskania wyniku zgodnie z benchmarkiem nr 3

```
def benchmark3(program):
    fitness = 0
    value = -1000
    checks = 1
    ands_count = program.count("&&")
    less_than_count = program.count("<")

    split_program = program.split(".")

    if "input()" in split_program[0] and "input()" in
   split_program[1] \
            and "input()" in split_program[2] and "input()" in
   split_program[3]:
        value += 200
        checks += 1
    else:
        return -1200
```

```
    if any(["if" in x and ("<" in x or ">" in x) and (x.count("
<") == 3 or x.count(">") == 3) for x in
            split_program[4:6]]):
        value += 100
        checks += 1

    if any(["print" in x for x in split_program[4:7]]):
        value += 100
        checks += 1

    if ands_count == 3:
        value += 50
        checks += 1

    if less_than_count == 3:
        value += 50
        checks += 1
    if len(split_program) < 12:
        value += 50
        checks += 1

    for x in split_program[4:7]:
        if "if" in x and ("<" in x or ">" in x):
            for index, character in enumerate(x):
                if character == "<" or character == ">":
                    try:
                        int(x[index - 2])
                        value -= 200
                    except:
                        value += 33
                        checks += 1


    value = value / checks

    try:
        lexer = ExprLexer(InputStream(program))
        stream = CommonTokenStream(lexer)
        parser = ExprParser(stream)
        tree = parser.prog()
```

```
55          num_readings = program.count("input()")
56          for _ in range(4):
57              visitor = ExprVisitor(20, -9, 9)
58              output, program_input = visitor.visit(tree)
59              if len(program_input) == 4:
60                  value += 50/(checks + 1)
61
62              # print("Output: ", output)
63              if output[0] == min(program_input) and num_readings
    == 4 and len(output) == 4:
64                  print("inputs:", program_input)
65                  print("Outputs: ", output)
66                  fitness += 0
67              elif len(output) == 1:
68                  fitness += (value - len(program) / 10000)
69              else:
70                  fitness += 2 * (value - len(program) / 10000)
71          return fitness
72      except Exception as e:
73          # print(e)
74          return 8 * (value - len(program) / 10000)
```

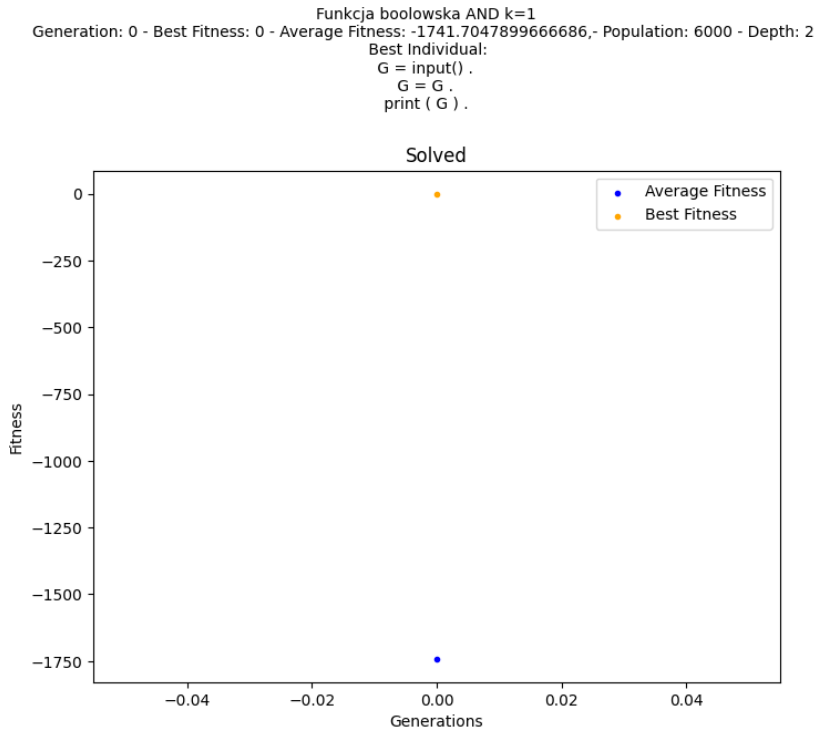
Algorytm do benchmarku numer 3 został ułożony tak aby nagradzać odpowiednie fragmenty kodu, czyli pierwsze 4 linie to przypisanie zmiennych z $input()$, jesli w liniach 4-7 jest if zawierajacy operatory porownania to również system jest nagradzany. Tak samo dzieje sie w przypadku jest w ostatich 2 liniach jest $print()$
Niestety nie udało sie znaleźć odpowiedniego programu mimo wielu prób, generacji i zmiany parametrów.

5.1.4 Benchmark 4 - Odwzorowanie funkcji boolowskiej AND

Do uczenia algorytmu zastosowana została pomocnicza klasa `TruthTableGenerator`. Klasa `TruthTableGenerator` zawiera metody do generowania tabel prawdy dla operacji logicznych: AND, OR oraz XOR. Generuje tabelę prawdy dla operacji logicznej AND/OR/XOR dla zadanej liczby zmiennych logicznych.

Odwzorowanie funkcji boolowskiej dla k = 1 (rys. 5.7) jest przypadkiem trywialnym.

W przypadku większej liczby zmiennej alorytm dość szybko zorientował się, że operację AND dla wartości 0-1 można bardzo łatwo osiągnąć przy pomocy mnożenia,

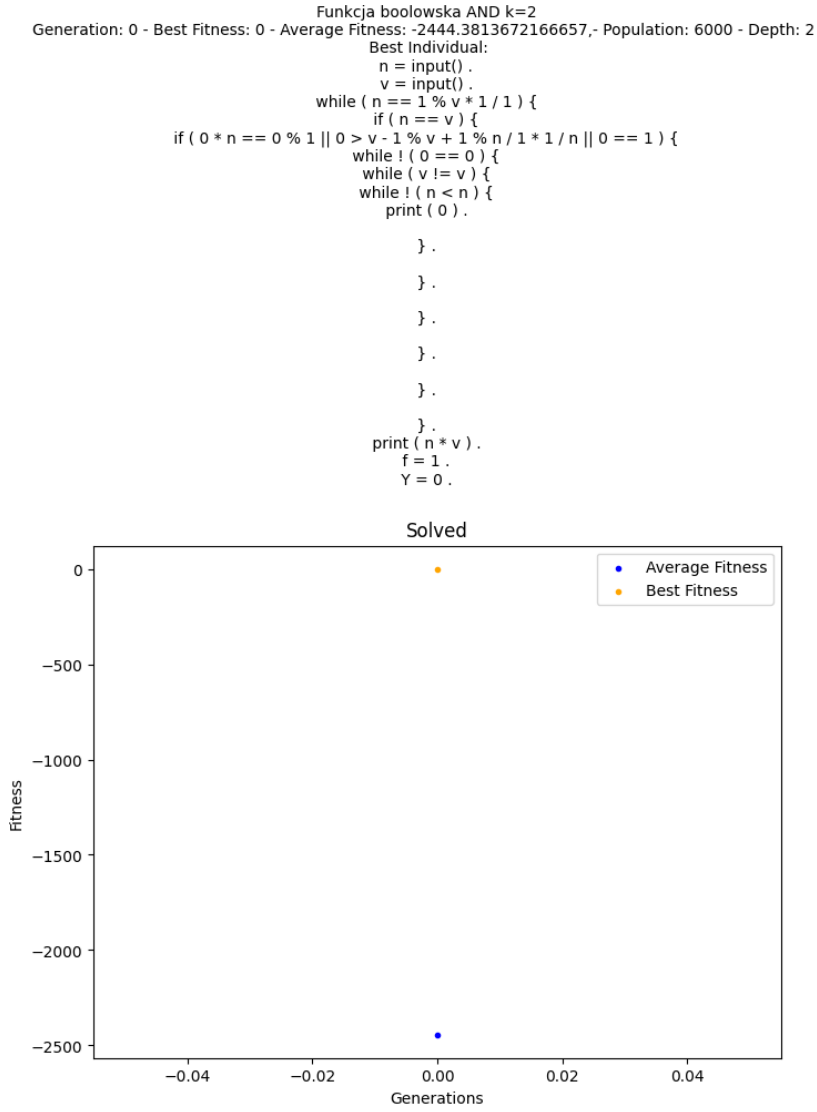


Rysunek 5.7: Program uzyskany dla funkcji boolowskiej k = 1

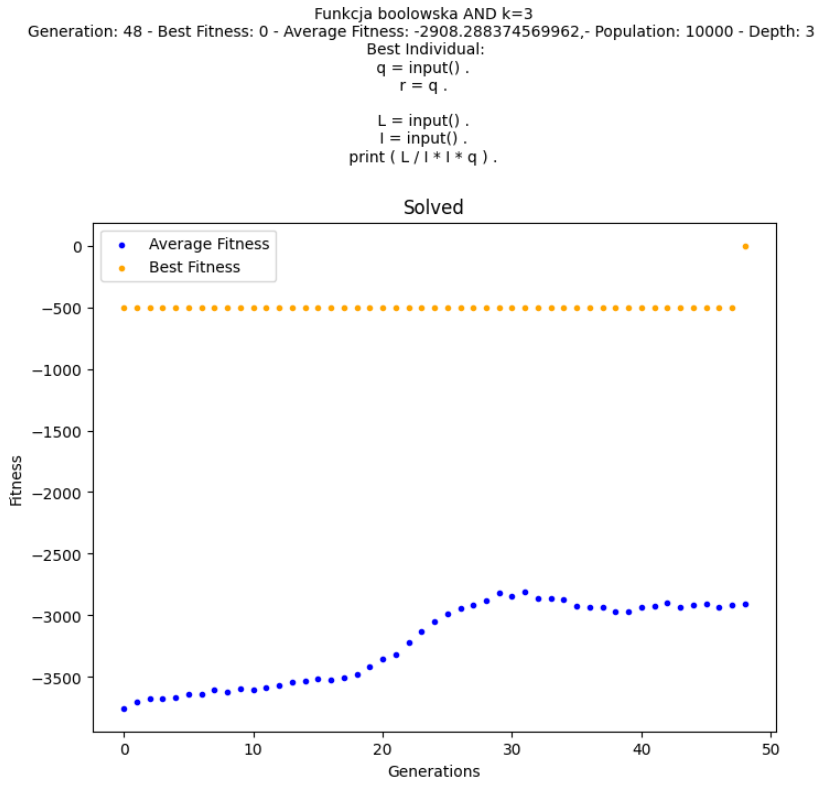
szczególnie przy małych ilościach zmiennych k=2, k-3 (rys. 5.8, 5.9). W przypadku 7 zmiennych algorytm znalazł dość uniwersalny algorytm, ograniczony maksymalną ilością iteracji pętli (ograniczony do ilości zmiennych) (rys. 5.10)

Do uczenia algorytmu użyta została następująca funkcja:

```
def bool_fitness_and(program):
    k = 7
    fitness = 0
    value = -1200
    checks = 1
    inputs_count = program.count("input()")
    if inputs_count == k:
        value += 300
        checks += 1
    else:
        return -3600
    split_program = program.split(".")
    variables = [line.strip()[0] for line in split_program if "
   input()" in line]

    if len(set(variables)) != k:
```

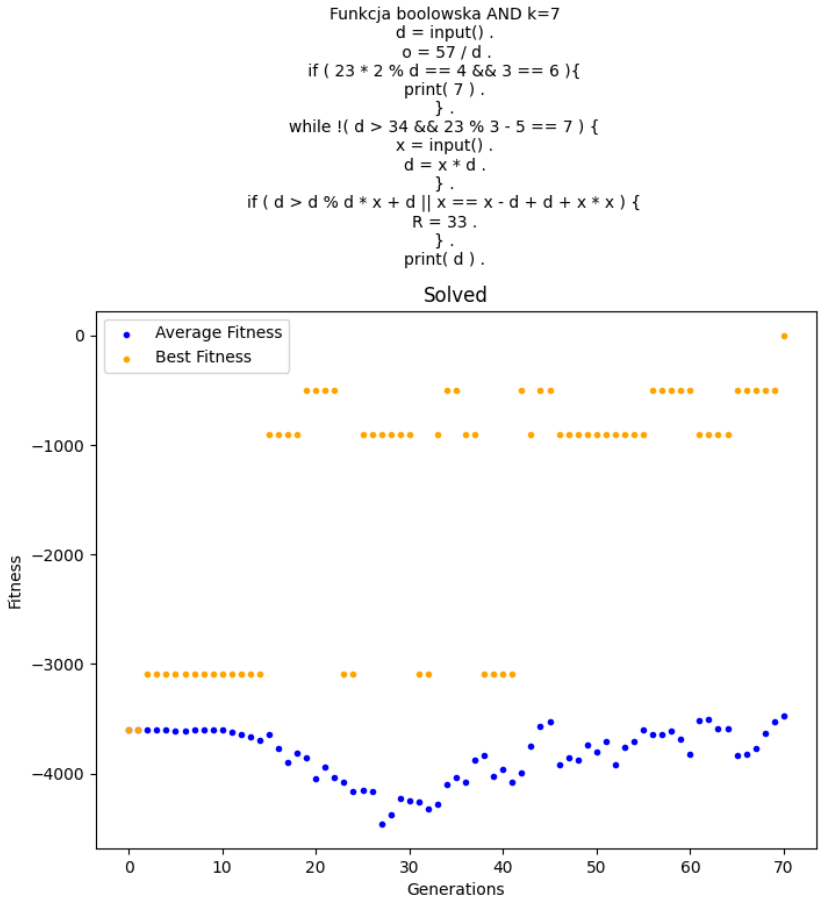


Rysunek 5.8: Program uzyskany dla funkcji boolowskiej k = 2



Rysunek 5.9: Program uzyskany dla funkcji boolowskiej k = 3

```
16          return -3300 + len(variables)*30
17      if "if" in split_program and "&" in split_program:
18          value += 400
19          checks += 1
20      if any(["*" in line and "print" in line for line in
    split_program]):
21          value += 400
22          checks += 1
23
24      tgen = TruthTableGenerator()
25      tables = tgen.and_truth_table(k)
26      try:
27          lexer = ExprLexer(InputStream(program))
28          stream = CommonTokenStream(lexer)
29          parser = ExprParser(stream)
30          tree = parser.prog()
31          for table in tables:
32              visitor = ExprVisitor(20, 1, 9, table[:-1])
33              output, program_input = visitor.visit(tree)
34              if (output[0] == table[-1] or (len(output) == 0 and
```



Rysunek 5.10: Program uzyskany dla funkcji boolowskiej k = 7

```
    table[-1] == 0)) and len(output) == 1 and len(program_input
) == k:
                fitness += 0
            elif len(output) == 1:
                fitness += (value - len(program) / 10000)
            else:
                fitness += 2 * (value - len(program) / 10000)
        return fitness
    except Exception as e:
        # print(e)
        return 8 * (value - len(program) / 10000)
```

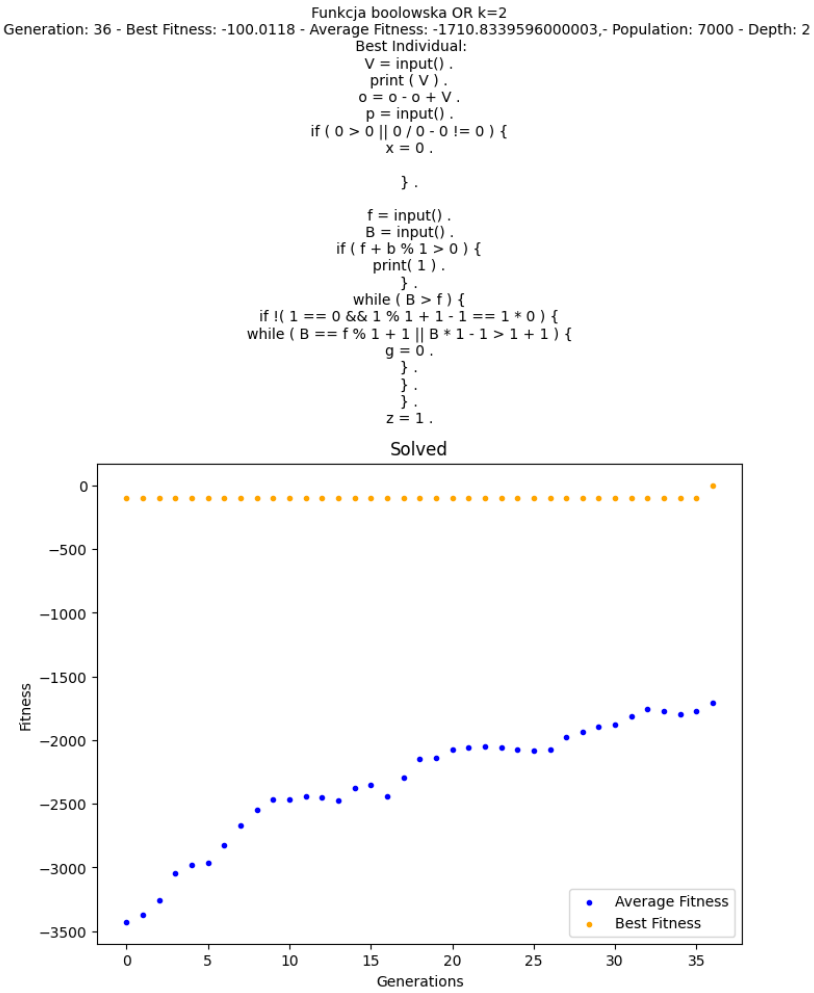
5.1.5 Benchmark 4 - Odwzorowanie funkcji boolowskiej OR

Odwzorowanie funkcji boolowskiej dla k = 1 (rys. 5.7) jest przypadkiem trywialnym.

W przypadku fukcji OR algorytm męczył się dużo bardziej niż w przypadku funkcji AND. Udało mu się wyprodukować sprawny program dla k = 2 (rys. 5.11

Do uczenia algorytmu użyta została następująca funkcja analogiczna dla funkcji AND:

```
def bool_fitness_or(program):
    k = 2
    fitness = 0
    value = -1200
    checks = 1
    inputs_count = program.count("input()")
    if inputs_count == k:
        value += 300
        checks += 1
    else:
        return -3600
    split_program = program.split(".")
    variables = [line.strip()[0] for line in split_program if "
input()" in line]

    if len(set(variables)) != k:
        return -3300 + len(variables)*30
    if any(["if" in line and "||" in line for line in
split_program]):
        value += 300
```



Rysunek 5.11: Program uzyskany dla funkcji boolowskiej k = 2

```
19          checks += 1
20          if any(["if" in line for line in split_program]):
21              value += 100
22              checks += 1
23      if any(["print" in line for line in split_program]):
24          value += 400
25          checks += 1
26
27      tgen = TruthTableGenerator()
28      tables = tgen.or_truth_table(k)
29      try:
30          lexer = ExprLexer(InputStream(program))
31          stream = CommonTokenStream(lexer)
32          parser = ExprParser(stream)
33          tree = parser.prog()
34          for table in tables:
35              visitor = ExprVisitor(20, 1, 9, table[:-1])
36              output, program_input = visitor.visit(tree)
37              if (output[0] == table[-1] or (len(output) == 0 and
     table[-1] == 0)) and len(output) == 1 and len(program_input
    ) == k:
38                  fitness += 0
39              elif len(output) == 1:
40                  fitness += (value - len(program) / 10000)
41              else:
42                  fitness += 2 * (value - len(program) / 10000)
43          return fitness
44      except Exception as e:
45          # print(e)
46          return 8 * (value - len(program) / 10000)
```