

Praca dyplomowa

Napisanie aplikacji dobierającej współczynniki równania z wykorzystaniem biblioteki PyGAD

Cezary Żmuda

Numer albumu 282556

promotor

mgr Paweł Chodkiewicz

# Warszawa 2022

**Streszczenie**

Celem niniejszej pracy dyplomowej jest stworzenie programu z graficznym interfejsem użytkownika wykorzystującej bibliotekę otwartego oprogramowania PyGAD i zweryfikowanie jej możliwości do rozwiązywania problemów kombinatorycznych, takich jak dobór parametrów dla prostych funkcji matematycznych. Przeanalizuję stworzoną przeze mnie aplikację do poszukiwania parametrów równania liniowego, wielomianowego i sinusoidalnego. Celem jest osiągnięcie optymalnego dopasowania modelu do funkcji eksperymentu bazującego na danych wprowadzonych przez użytkownika. Został zaprezentowany cały proces stworzenia aplikacji w języku programistycznym python w oparciu o biblioteki otwartego oprogramowania jak i kalibrowanie parametrów algorytmu w celu uzyskania dokładniejszych rozwiązań. Praca składa się z teoretycznego wprowadzenia czytelnika w zagadnienia związane z algorytmiką i dokładnym opisem algorytmu genetycznego. W kolejnych rozdziałach zostało opisane środowisko programistyczne, wykorzystane biblioteki i szczegółowy opis algorytmu genetycznego z biblioteki PyGAD. Przedstawiono również szczegółowe działanie programu dobierającego parametry, wykresy z wynikami i przykłady kalibrowania algorytmu dla funkcji liniowej, wielomianowej i sinusoidalnej. W ostatniej części jest omówiony program z graficznym interfejsem użytkownika i jego funkcjonalnościami.

**Abstract**

The purpose of this thesis is to create a graphical user interface program using the PyGAD open source library and verify its ability to solve combinatorial problems such as parameter selection for simple mathematical functions. I will analyze the application I’ve created for finding the parameters of a linear, polynomial, and sinusoidal equation. The goal is to achieve the optimal fitting of the model to the experimental function based on user input. The whole process of creating the application in python programming language based on open source libraries as well as calibrating the algorithm parameters to obtain more accurate solutions has been presented. The paper consists of a theoretical introduction of the reader to issues related to algorithmics and a detailed description of the genetic algorithm. The following chapters describe the development environment, the use of libraries and a detailed description of the genetic algorithm from the PyGAD library. The detailed operation of the parameter selection program, graphs with results and examples of calibrating the algorithm for linear, polynomial and sinusoidal functions are also presented. In the last section is discussed the program with a graphical user interface with all functionalities.

**Spis treści**

1. Cel i opis problemu 3
2. Wstęp 4
3. Metaheurystyki 6
   1. Czym są metaheurystyki? 6
   2. Popularne metaheurystyki 10
      1. Algorytmy ewolucyjne……………………………………………………10
      2. Inteligencja roju…………………………………………………………..12
      3. Symulowane wyżarzanie………………………………………………..13
      4. Przeszukiwanie tabu……………………………………………………..14
      5. Sztuczny układ odpornościowy…………………………………………15
      6. Metaheurystyki hybrydowe……………………………………………...16
      7. Algorytmy mrówkowe……………………………………………………17
   3. Algorytmy genetyczne 20
      1. Selekcja…………………………………………………………………...22
      2. Krzyżowanie………………………………………………………………23
      3. Mutacja…………………………………………………………………….24
      4. Wskaźnik mutacji…………………………………………………………25
      5. Współczynnik krzyżowania……………………………………………...26
      6. Rozmiar populacji………………………………………………………...26
      7. Funkcja fitness……………………………………………………………26
      8. Opis działania algorytmu genetycznego……………………………….27
4. Implementacja algorytmu 30
   1. Język programowania i środowisko programistyczne 30
   2. Biblioteki i technologie 31
      1. Klasa pygad.GA………………………………………………………….34
      2. Metody generowania wykresów w klasie pygad.GA…………………37
      3. Metody wyboru rodziców………………………………………………..37
   3. Programowanie zorientowane obiektowo 38
   4. Przykładowa implementacja algorytmu 41
   5. Algorytm genetyczny dla funkcji wielomianowej 45
   6. Algorytm genetyczny dla funkcji sinus 57
5. Graficzny interfejs użytkownika 72
6. Podsumowanie 79
7. Bibliografia 80
8. Spis rysunków 83
9. Spis tabel 87
10. **Cel i opis problemu**

Głównym celem stworzonego przeze mnie programu będzie zaprezentowanie możliwości wykorzystania algorytmu genetycznego w zadaniu numerycznym. Aplikacja będzie spełniać zadanie optymalizacyjne, które przedstawię pokrótce w niniejszym rozdziale. Wizja jaka przyświecała mi w tworzeniu aplikacji, ewoluowała w trakcie głębszego zagłębiania się w badaną dziedzinę. Docelowo aplikacja została napisana w sposób zorientowany – obiektowo, co ma niewątpliwą zaletę w przypadku wielokrotnego uruchamianie aplikacji, dla różnych wartości parametrów, z kolei w przypadku optymalizacji algorytmu odgrywa kluczową kwestię, bo daje nam możliwość dobrania wielu parametrów w oparciu o szybkość działania, jak i skuteczność algorytmu. Oczywiście to nie wyczerpuje wszystkich możliwości optymalizacyjnych, bo ich liczba jest ograniczona przez wyobraźnię czytelnika.

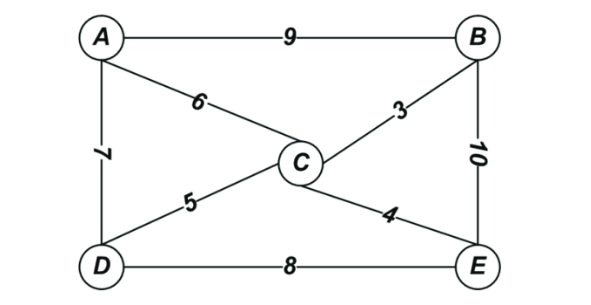
Problem, na podstawie którego zostanie zaprezentowany sposób działania algorytmu genetycznego tyczy się funkcji wielomianowej () i funkcji sinusoidalnej (), który polega na dobraniu wartości współczynników odpowiadających modelowi eksperymentalnemu. Dane eksperymentalne są podawane przez użytkownika. Następnie algorytm będzie przeszukiwał tablicę losowo wygenerowanych liczb w celu znalezienia najbardziej optymalnych rozwiązań, czyli takich, które są najbliższe funkcji podanej przez użytkownika i jednoczenie ich czas przeszukiwania będzie dużo niższy niż w przypadku znalezienia najlepszego możliwego rozwiązania. W późniejszych rozdziałach zostanie zaprezentowany dokładny opis funkcjonowania aplikacji, jak i samego algorytmu, poparty licznymi wykresami prezentującymi jego skuteczność i wpływ poszczególnych parametrów na rezultat końcowy.

Jak wiadomo przyczyną powstania wielu metod heurystycznych, w tym algorytmu genetycznego, był problem komiwojażera, polegający na znalezieniu najkrótszej drogi przechodzącej przez wiele miast. Dużym impulsem był również rewolucyjny wzrost mocy obliczeniowej komputerów osobistych. Z wielu badań, które zostały przeprowadzone na przestrzeni lat, jasno wynika, że algorytm genetyczny charakteryzuje się bardzo wysoką skutecznością w rozwiązywaniu problemów kombinatorycznych, stąd też moje zainteresowanie do eksploracji go w innych obszarach zadań optymalizacyjnych.

1. **Wstęp**

Początek badań nad **problemem komiwojażera** (Travelling Salesman Problem lub TSP) nie jest jasny. Wspomina o nim podręcznik z 1832, który zawiera przykładową trasę po Niemczech i Szwajcarii, lecz nie zawiera żadnych matematycznych uzasadnień. W 1859 irlandzki matematyk William Rowan Hamilton sformułował problem istnienia cyklu o długości n w grafie n-wierzchołkowym. Za pierwszego autora, który sformalizował matematycznie problem komiwojażera uznaje się austriackiego matematyka Karla Mengera, który zdefiniował go w 1930 zwracając szczególną uwagę na trudność w obliczeniu rozwiązania. Niezależnie od niego ten sam problem poruszył w 1934 Hassler Witney na wykładzie w Princeton University. Natomiast pierwsza próba rozwiązania problemu miała miejsce w 1937, gdy Merrill Flood pracował nad rozwiązaniem wyznaczania tras dla autobusów szkolnych. Z uwagi na bardzo prosty opis problemu oraz opinię o bardzo trudnym obliczeniowo procesie optymalizacji, problem komiwojażera stał się bardzo popularny. Fascynacja ta trwa od lat pięćdziesiątych XX wieku do dziś, zarówno wśród amatorów jak i profesjonalistów. [19]

W zwykłej postaci TSP, sprzedawca otrzymuje mapę miast i musi odwiedzić wszystkie miasta tylko raz, aby zakończyć trasę w taki sposób, żeby długość trasy była jak najkrótsza spośród wszystkich możliwych tras dla tej mapy. Dane składają się z wag przypisanych krawędziom skończonego grafu zupełnego, a celem jest znalezienie cyklu Hamiltona, czyli cyklu przechodzącego przez wszystkie wierzchołki grafu, który ma minimalną wagę całkowitą. W kontekście TSP cykle hamiltonowskie są powszechnie nazywane trasami. Na przykład, biorąc pod uwagę mapę pokazaną na rysunku 2.1.1, najmniej kosztowną trasą byłaby (A, B, C, E, D, A), z kosztem 31. Ogólnie rzecz biorąc TSP obejmuje dwa różne rodzaje: Symetryczny TSP i Asymetryczny TSP. W postaci symetrycznej znanej jako STSP istnieje tylko jedna droga pomiędzy dwoma sąsiadującymi miastami, tzn. odległość pomiędzy miastami A i B jest równa odległości pomiędzy miastami B i A (Rys. 1). Natomiast w ATSP (Asymmetric TSP) nie ma takiej symetrii i możliwe jest występowanie dwóch różnych kosztów lub odległości pomiędzy dwoma miastami. Stąd liczba tras w ATSP i STSP na *n* wierzchołkach (miastach) wynosi odpowiednio . Należy pamiętać, że grafy reprezentujące te TSP są grafami zupełnymi. W tym rozdziale rozpatrujemy głównie STSP. Wiadomo, że TSP jest często używany do testowania algorytmów optymalizacyjnych. Rozwiązywanie TSP było interesującym problemem w ostatnich dekadach. Prawie każde nowe podejście do rozwiązywania problemów inżynierskich i optymalizacyjnych było testowane na TSP jako ogólnym wzorcu testowym. Pierwszymi krokami w rozwiązywaniu TSP były metody klasyczne. Metody te składają się z metod heurystycznych i dokładnych. Metody heurystyczne, takie jak płaszczyzny tnące oraz rozgałęzienia i połączenia (Padherg & Rinaldi, 1987), mogą optymalnie rozwiązywać tylko małe problemy, podczas gdy metody heurystyczne, takie jak 2-opt, 3-opt, łańcuch Markowa, symulowane wyżarzanie oraz przeszukiwaniu tabu są dobre dla dużych problemów. Ponadto, niektóre algorytmy oparte na zasadach zachłanności, takie jak najbliższy sąsiad i drzewo rozpinające mogą być również postrzegane jako skuteczne metody. Niemniej jednak, klasyczne metody rozwiązywania TSP zwykle powodują wykładniczą złożoność obliczeniową. Dlatego też potrzebne są nowe metody, aby przezwyciężyć to niedociągnięcie. Metody te obejmują różnego rodzaju techniki optymalizacyjne, algorytmy optymalizacyjne oparte na naturze, algorytmy optymalizacyjne oparte na populacji i inne. [17]



Rys.2.1.1 Mapa odwiedzanych miast z najkrótszą ścieżką (A, B, C, D, E). [18]

**Problem plecakowy** jest szeroko badanym problemem optymalizacji kombinatorycznej. Szczególne zainteresowanie wynika z licznych zastosowań w życiu codziennym, na przykład w logistyce i harmonogramowaniu. Podstawowy problem polega na wybraniu z danego zbioru elementów takiego podzbioru, aby łączna waga (lub rozmiar) podzbioru nie przekraczała danego limitu (pojemności plecaka), a łączna korzyść z podzbioru była maksymalna (przegląd (deterministycznych) problemów plecakowych. [18]

Naukowcy badają problemy plecakowe (KP) od czasu, gdy Dantzig po raz pierwszy przedstawił je w 1957 roku. W ostatnich kilkudziesięciu latach KP znalazły zastosowanie w wielu dziedzinach, takich jak teoria obliczeń czy zarządzanie finansami, i nadal są klasycznymi problemami optymalizacji kombinatorycznej. Klasyczne KP można opisać następująco:

gdzie i reprezentują odpowiednio wagę i zysk każdego elementu ; całkowita pojemność wagowa każdego plecaka jest mniejsza niż ; a jest zmienną decyzyjną każdego elementu . Celem KP jest znalezienie rozwiązania dla maksymalnego całkowitego zysku. Jeśli element jest zapakowany to wynosi jeden; w przeciwnym razie jest równe zero. Przyjmuje się, że wszystkie zyski i wagi są dodatnie. Zakłada się, że suma wszystkich wag jest mniejsza od pojemności . [18]

1. **Metaheurystyki**

**3.1 Czym są metaheurystyki?**

Optymalizacja jest częścią natury i nieuchronnie, integralną częścią ludzkiego życia. Każda podejmowana przez nas decyzja jest próbą znalezienia optymalnego lub prawie optymalnego rozwiązania. Z ogólnego punktu widzenia, każdy problem optymalizacyjny może być traktowany jako problem decyzyjny, a pytanie brzmi, czy istnieje lepsze rozwiązanie problemu niż to, które znaleźliśmy. Innymi słowy, optymalizacja oznacza osiągnięcie rozwiązania tak dobrego, jak to tylko możliwe, aby doprowadzić nas do lepszej wydajności rozważanego systemu. Zgodnie z opisem Beightlera, optymalizacja jest trzystopniowym procesem decyzyjnym: (1) modelowanie problemu na podstawie wiedzy o problemie, (2) znalezienie miar efektywności lub funkcji celu oraz (3) metoda optymalizacji lub teoria optymalizacji. Można śmiało powiedzieć, że cała dziedzina optymalizacji, a szczególnie ten ostatni etap, zyskała jedynie dzięki rozwojowi i udoskonalaniu komputerów, które rozpoczęło się w połowie lat 40. Istnienie metod optymalizacji można prześledzić za czasów Newtona, Bernoulliego, Lagrange’a, Cauchy’ego i Gibbsa, w których analiza matematyczna została stworzona na podstawie rachunku wariacji. Takie metody optymalizacji są ogólnie znane jako programowanie matematyczne i obejmują wiele zaawansowanej literatury na przestrzeni dziesięcioleci. Wraz z pojawieniem się komputerów pojawiła się potrzeba optymalizacji bardziej skomplikowanych i z natury nieliniowych problemów optymalizacyjnych, co zaowocowało nowymi osiągnięciami w teorii optymalizacji. Inny obszar teorii optymalizacji, znany jako metaheurystyka, został opracowany w połowie lat czterdziestych XX wieku, a w ostatnim półwieczu poświęcono mu wiele uwagi. [3]

Jako naukowcy, inżynierowie i menedżerowie zawsze musimy podejmować decyzje. Ponieważ świat staje się coraz bardziej złożony i konkurencyjny, więc podejmowanie decyzji musi odbywać się w sposób racjonalny i optymalny. Możemy wyszczególnić następujące etapy procesu decyzyjnego:

* **Sformułowanie problemu**: W tym pierwszym kroku identyfikuje się problem decyzyjny. Jest wykonane wstępne sformułowanie problemu, które może być nieprecyzyjne. Czynniki wewnętrzne i zewnętrzne oraz cel problemu są nakreślone. W formułowanie problemu może być zaangażowanych wielu decydentów.
* **Modelowanie problemu**: W tym ważnym kroku budowany jest abstrakcyjny model matematyczny dla danego problemu. Model może być inspirowany podobnymi w literaturze. W ten sposób problem zostanie sprowadzony do dobrze zbadanych modeli optymalizacyjnych. Zazwyczaj modele, które rozwiązujemy są uproszczeniem rzeczywistości. Zawierają one przybliżenia, a czasem pomijają procesy, które są skomplikowane do przedstawienia w modelu matematycznym.
* **Optymalizacja problemu**: Po zamodelowaniu problemu, procedura rozwiązywania generuje „dobre” rozwiązanie. Rozwiązanie to może być optymalne lub suboptymalne. Należy zauważyć, że znajdujemy rozwiązanie dla abstrakcyjnego modelu, a nie dla oryginalnie sformułowanego problemu rzeczywistego. W związku z tym, uzyskane wydajności rozwiązań są orientacyjne, gdy model jest dokładny.
* **Wdrożenie rozwiązania**: Uzyskane rozwiązanie jest testowane praktycznie i jest wdrażane, jeśli jest „akceptowalne”. Pewna wiedza praktyczna może zostać wprowadzona do rozwiązania, które ma być wdrożone. Jeśli rozwiązanie jest nieakceptowalne, model i/lub algorytm optymalizacyjny musi zostać poprawiony i proces decyzyjny jest powtarzany. [4]

Heurystyka (z greckiego heuriskein lub euriskein, „szukać”) jest metodologią rozumowania w rozwiązywaniu problemów, która umożliwia rozwiązanie problemu metodą prób i błędów. Z kolei metaheurystyka jest algorytmem zaprojektowanym do rozwiązywania w przybliżeniu szerokiego zakresu trudnych problemów optymalizacyjnych bez konieczności szczegółowego dostosowywania się do każdego problemu. Grecki przedrostek „meta”, obecny w nazwie, jest używany do wskazania, że te algorytmy są heurystykami „wyższego poziomu”, w przeciwieństwie do heurystyk specyficznych dla danego problemu. Metaheurystyki są wykorzystywane do rozwiązywania problemów optymalizacyjnych poprzez proces poszukiwania optymalnych rozwiązań dla danego problemu. Proces poszukiwania może być przeprowadzony przy użyciu wielu agentów, którzy zasadniczo tworzą system ewoluujących rozwiązań przy użyciu zestawu reguł lub równań matematycznych podczas wielu iteracji. Te iteracje trwają do momentu, gdy znalezione rozwiązanie spełnia jakieś z góry określone kryterium. To ostateczne rozwiązanie (rozwiązanie bliskie optymalnemu) jest uważane za rozwiązanie optymalne i uznaje się, że system osiągnął stan zbieżny. Z tego powodu zostały opracowane metaheurystyki, aby właśnie znaleźć rozwiązanie „wystarczająco dobre” w czasie obliczeń, który jest „wystarczająco krótki”. [2]

Metaheurystyki zostały udowodnione przez środowisko naukowe jako realna, a często lepsza, alternatywa dla bardziej tradycyjnych (dokładnych) metod optymalizacji wielomianowej. Szczególnie w przypadku skomplikowanych problemów lub dużych instancji, metaheurystyki są często w stanie zaoferować lepszy kompromis pomiędzy jakością rozwiązania, a czasem obliczeń. Co więcej, metaheurystyki są bardziej elastyczne niż metody dokładne na dwa ważne sposoby. Po pierwsze, ponieważ struktury metaheurystyczne są zdefiniowane w sposób ogólny, algorytmy metaheurystyczne mogą być dostosowane do potrzeb większości rzeczywistych problemów optymalizacyjnych pod względem oczekiwanej jakości rozwiązania i dopuszczalnego czasu obliczeń, który może być bardzo różny dla różnych problemów i różnych sytuacji. Po drugie, metaheurystyki nie nakładają żadnych wymagań na sformułowanie problemu optymalizacyjnego (jak wymóg, aby ograniczenia lub funkcje celu były wyrażone jako liniowe funkcje zmiennych decyzyjnych). Jednakże, ta elastyczność jest kosztem tego, że wymaga znacznego dostosowania do konkretnego problemu, aby osiągnąć dobrą wydajność. [6]

Niezależnie od bogatej literatury na temat modyfikacji obecnych metaheurystyk z wykorzystaniem różnych mechanizmów i strategii, które są w toku, dziedzina ta jest świadkiem pojawiania się nowych metaheurystyk, być może raz na miesiąc. Historię metaheurystyki można opisać w pięciu odrębnych okresach:

* okres przed-teoretyczny (do ok. 1940 r.), w którym heurystyki, a nawet metaheurystyki były używane, ale nie były formalnie przedstawione,
* okres wczesny (ok. 1940-1980 r.), w którym pojawiają się pierwsze formalne opracowania heurystyk,
* okres metodo – centryczny (ok. 1980 – 2000 r.), w którym dziedzina metaheurystyk nabiera rozpędu i proponuje się wiele różnych metod,
* okres strukturo – centryczny (ok. 2000 r.- teraz), podczas którego wzrasta świadomość, że metaheurystyki są bardziej użyteczne jako struktury, a nie jako metody.
* okres naukowy (przyszłość), w którym projektowanie metaheurystyk staje się nauką, a nie sztuką.

Przed rokiem 2000 istniała pilna potrzeba rozwiązywania skomplikowanych problemów optymalizacyjnych. W tym okresie powstały metaheurystyki oparte na ewolucji (najbardziej znane to: strategie ewolucyjne, programowanie ewolucyjne, algorytmy genetyczne, programowanie genetyczne i ewolucja różnicowa) oraz metaheurystyki oparte na trajektorii (takie jak wspinaczka na wzniesienie, symulowane wyżarzanie, przeszukiwanie tabu, iterowane wyszukiwanie lokalne, zmienne wyszukiwanie sąsiedzkie). Okres przejściowy miał miejsce około roku 2000, kiedy to wynaleziono najbardziej udane i znane metaheurystyki oparte na roju (optymalizacja rojem cząstek i optymalizacja kolonii mrówek). W tym okresie potrzebne były bardziej wydajne metaheurystyki oraz nowe rozwiązania dla przeszukiwania lokalnego i globalnego. [3]

Istnieje kilka różnych kryteriów klasyfikacji metaheurystyk, z których najczęstszym jest wyszukiwanie oparte na populacji versus wyszukiwanie oparte na pojedynczym rozwiązaniu. Zdefiniowanie tych dwóch klas może być przydatne do zapoznania się z metaheurystyką. Metaheurystyki oparte na pojedynczym rozwiązaniu manipulują i przekształcają pojedyncze rozwiązanie w celu uzyskania rozwiązania optymalnego, stosując iteracyjną procedurę wyszukiwania (generowanie i zastępowanie). Generowanie oznacza tworzenie jednego rozwiązania kandydującego lub zbioru rozwiązań kandydujących z bieżącego rozwiązania w oparciu o struktury lub mechanizmy wyższego rzędu. Zastępowanie oznacza, że nowo wygenerowane rozwiązanie lub odpowiednie rozwiązanie z wygenerowanego zbioru jest wybierane do zastąpienia bieżącego rozwiązania w celu wejścia w obiecujący region przestrzeni poszukiwań. Ten iteracyjny proces jest kontynuowany do momentu spełnienia kryterium stopu. W metaheurystykach populacyjnych również stosuje się iteracyjną procedurę wyszukiwania (zawierającą generowanie i zastępowanie), ale w tym typie metaheurystyki zbiór rozwiązań rozciąga się na przestrzeń poszukiwań. Po pierwsze, inicjalizowany jest zbiór rozwiązań zwany populacją początkową. Inicjalizacja może być przeprowadzona przy użyciu różnych strategii, ale najczęściej stosowaną jest losowe generowanie agentów w przestrzeni poszukiwań. Algorytm manipuluje bieżącym zbiorem rozwiązań wielokrotnie, w oparciu o nadrzędne struktury lub mechanizmy wyszukiwania, w celu wygenerowania nowych i zastąpienia starych rozwiązań nowo wygenerowanymi przy użyciu określonej strategii. Proces ten trwa do momentu spełnienia kryterium stopu. Najczęstszymi kryteriami stopu są: ustalona liczba iteracji algorytmu, maksymalna liczba iteracji bez postępu w funkcji celu oraz minimalna wartość funkcji celu. [3]

Metaheurystyki oparte na pojedynczym rozwiązaniu, zwane również metodami trajektorii. W przeciwieństwie do metaheurystyk opartych na populacji, zaczynają one od jednego rozwiązania początkowego i oddalają się od niego, opisując trajektorię w przestrzeni poszukiwań. Niektóre z nich mogą być postrzegane jako „inteligentne” rozszerzenia algorytmów przeszukiwania lokalnego. Metody trajektorii obejmują głównie metodę symulowanego wyżarzania, przeszukiwanie tabu, przeszukiwanie zmiennego sąsiedztwa, kierowane przeszukiwanie lokalne, iterowane przeszukiwanie lokalne i ich warianty. [5]

Metaheurystyki oparte na populacjach zajmują się raczej zbiorem (tj. populacją) rozwiązań niż pojedynczym rozwiązaniem. Najbardziej zbadane metody oparte na populacji są związane z obliczeniami ewolucyjnymi („Evolutionary Computation”) oraz inteligencją roju („Swarm Intelligence”). Algorytmy ewolucyjne są inspirowane teorią ewolucji Darwina, gdzie populacja osobników jest modyfikowana poprzez operatory rekombinacji i mutacji. W inteligencji roju, ideą jest wytworzenie inteligencji obliczeniowej poprzez wykorzystanie prostych analogi interakcji społecznych, a nie czysto indywidualnych zdolności poznawczych. [5]

Znaczny wzrost wydajności komputerów w ostatnich dziesięcioleciach umożliwił modelowanie, analizowanie i projektowanie rzeczywistych problemów tak precyzyjnie, jak to tylko możliwe w różnych zastosowaniach i dziedzinach badawczych. Prawie wszystkie problemy świata rzeczywistego mogą być uważane za skomplikowane problemy optymalizacyjne. Złożoność ta wynika z różnych nieodłącznych cech, takich jak dyskretna lub mieszana przestrzeń rozwiązań, wysoce nieliniowa i/lub wielokryteriowa ocena efektywności problemu, duża przestrzeń przeszukiwania potencjalnych rozwiązań oraz liczne ograniczenia projektowe. Z drugiej strony, pojawiające się istotne zalety metaheurystyk sprawiają, że są one pierwszorzędnym wyborem, który zyskuje coraz większe zainteresowanie w optymalizacji rzeczywistych problemów optymalizacyjnych. Metaheurystyki są wykorzystywane w problemach optymalizacyjnych w różnych dyscyplinach, itp. w inżynierii, ekonomii i naukach ścisłych. Należy zaznaczyć, że metaheurystyki znalazły również zastosowanie w rozwiązywaniu wielu problemów optymalizacji kombinatorycznej, problemów rozwiązywalnych w czasie wielomianowym oraz różnych problemów wyszukiwania, takich jak selekcja cech, automatyczne grupowanie i uczenie maszynowe. W dziedzinie inżynierii można odnieść się do następujących zastosowań:

* optymalne projektowanie statków powietrznych i pojazdów kosmicznych w inżynierii lotniczej i kosmicznej
* optymalizacja szacowania kosztów i czasu w inżynierii oprogramowania
* bezpieczeństwo sieci i komputerów
* optymalne projektowanie konstrukcji i infrastruktury w inżynierii lądowej i wodnej
* optymalne projektowanie systemów mechanicznych, motoryzacyjnych i automatycznych
* optymalizacja procesów w inżynierii
* optymalne planowanie i harmonogramowanie produkcji w inżynierii przemysłowej i systemowej
* przetwarzanie sygnałów i obrazów
* optymalne projektowanie sieci w elektrotechnice oraz wiele innych zastosowań w różnych interdyscyplinarnych dziedzinach inżynierii. [3]

Jednym z najbardziej interesujących trendów w ostatnich latach są hybrydowe metody optymalizacji. Coraz więcej publikacji dotyczy hybrydyzacji metaheurystyki z innymi technikami optymalizacyjnymi. Hybrydyzacja nie ogranicza się do łączenia różnych metaheurystyk, ale obejmuje również stosowanie algorytmów hybrydowych, które łączą wyszukiwanie lokalne lub algorytmy dokładne i metaheurystyki. Ponadto, połączenie koncepcji z różnych metaheurystyk i różnych obszarów badawczych może prowadzić do ciekawych nowych podejść, które łączą logikę rozmytą i kilka technik optymalizacyjnych. Takie hybrydyzacje mogą wykorzystywać mocne strony każdego algorytmu w celu poprawy wydajności dla bardziej efektywnego i skutecznego rozwiązywania problemów. [5]

* 1. **Popularne metaheurystyki**

Rozdział ma na celu zaprezentować parę popularnych metaheurystyk, aby pokazać mnogość różnych metod optymalizacyjnych, jak i ich różnorodność podchodzenia do rozwiązywania problemów. Dzięki szybkiemu rozwojowi technologicznemu mamy do czynienie z dużą grupą algorytmów, które powstały w oparciu o przykłady zjawisk zaczerpniętych z natury. Najpopularniejsze z nich zostaną zaprezentowane i pokrótce omówione w dalszych podrozdziałach.

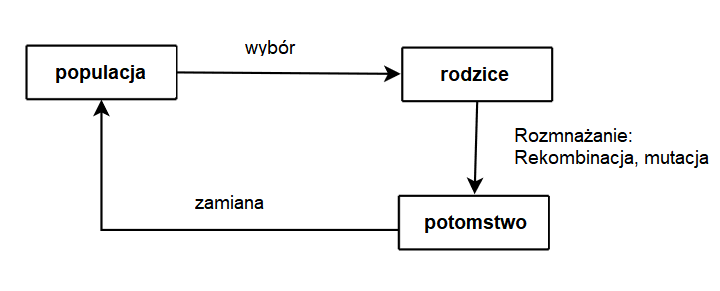
* + 1. **Algorytmy ewolucyjne**

Obliczenia ewolucyjne (Evolutionary Computation, w skrócie EC) jest ogólnym terminem dla kilku algorytmów optymalizacyjnych, które są inspirowane darwinowskimi zasadami zdolności natury do ewolucji istot żywych dobrze przystosowanych do swojego środowiska. Zazwyczaj pod pojęciem algorytmów EC (zwanych również algorytmami ewolucyjnymi ) kryją się takie dziedziny jak: algorytmy genetyczne, strategie ewolucyjne, programowanie ewolucyjne oraz programowanie genetyczne. [5]

Można powiedzieć, że optymalizacja jest wykonywana przy użyciu algorytmów ewolucyjnych (EA). Różnica pomiędzy tradycyjnymi algorytmami, a EA polega na tym, że EA nie są statyczne, lecz dynamiczne, gdyż mogą ewoluować w czasie. Algorytmy ewolucyjne mają trzy główne cechy:

* **Populacyjne:** Algorytmy ewolucyjne polegają na optymalizacji procesu, w którym aktualne rozwiązania są niszczone w celu wygenerowania nowych, lepszych rozwiązań. Zbiór aktualnych rozwiązań, z których mają być generowane nowe rozwiązania nazywany jest populacją.
* **Zorientowane na wartość dopasowania (fitness):** Jeśli istnieje kilka rozwiązań, to jak stwierdzić, że jedno rozwiązanie jest lepsze od drugiego? Z każdym rozwiązaniem związana jest wartość fitness obliczana na podstawie funkcji fitness. Taka wartość odzwierciedla jak dobre jest dane rozwiązanie.
* **Wariacyjne:** Jeśli w aktualnej populacji nie ma rozwiązania akceptowalnego według funkcji fitness obliczonej na podstawie każdego osobnika, należy podjąć działania w celu wygenerowania nowych, lepszych rozwiązań. W rezultacie, poszczególne rozwiązania będą poddawane wielu wariacjom w celu wygenerowania nowych rozwiązań. [42]

Algorytmy ewolucyjne są stochastycznymi równoległymi metaheurystykami, które zostały z powodzeniem zastosowane do wielu rzeczywistych i złożonych problemów (wielomodalnych, wielobodźcowych i silnie ograniczonych). Są to najlepiej przebadane algorytmy oparte na populacji. Ich sukces w rozwiązywaniu trudnych problemów optymalizacyjnych w różnych dziedzinach (optymalizacja ciągła lub kombinatoryczna, modelowanie i identyfikacja systemów, planowanie i sterowanie, projektowanie inżynierskie, eksploracja danych i uczenie maszynowe) przyczynił się do rozwoju dziedziny znanej jako obliczenia ewolucyjne (evolutionary computation). Reprezentują one klasę iteracyjnych algorytmów optymalizacji, które symulują ewolucję gatunków. Opierają się one na ewolucji populacji osobników. Początkowo, populacja ta jest zwykle generowana losowo. Każdy osobnik w populacji jest zakodowaną wersją wstępnego rozwiązania. Funkcja celu kojarzy wartość dopasowania z każdym osobnikiem, wskazując jego przydatność w rozwiązaniu problemu. Na każdym etapie osobniki są wybierane na rodziców zgodnie z paradygmatem selekcji, w którym osobniki o lepszej kondycji są wybierane z większym prawdopodobieństwem. Następnie wybrane osobniki są powielane przy użyciu operatorów wariacji (krzyżowanie, mutacja) w celu wygenerowania nowych osobników potomnych. W końcu, schemat zamiany jest stosowany do określenia, które osobniki z populacji przetrwają z potomstwa i rodziców. Ta iteracja reprezentuje jedno pokolenie (rys. 3.2.1). Proces ten jest powtarzany aż do momentu, gdy spełnione zostanie kryterium stopu. W algorytmach ewolucyjnych genotyp reprezentuje kodowanie, podczas gdy fenotyp reprezentuje rozwiązanie. W związku z tym genotyp musi zostać zdekodowany, aby wygenerować fenotyp. Operatory wariacji działają na poziomie genotypu, natomiast funkcja dopasowania wykorzystuje fenotyp skojarzonego osobnika. Dopasowanie osobnika mierzy jego zdolność do przetrwania w środowisku. W przypadku zastosowania kodowania bezpośredniego, genotyp jest podobny do fenotypu. W przeciwnym wypadku (tj. w przypadku kodowania pośredniego) genotyp i fenotyp są strukturami odmiennymi. W rzeczywistości do przekształcenia genotypu w fenotyp używana jest funkcja dekodująca. [4]



Rys. 3.2.1 Wykres przedstawia generowanie pełnego cyklu jednego pokolenia w algorytmach ewolucyjnych [4]

* + 1. **Inteligencja roju**

Inteligencja rojowa (Swarm Intelligence, w skrócie SI) jest innowacyjnym, rozproszonym, inteligentnym paradygmatem rozwiązywania problemów optymalizacyjnych, który czerpie inspirację z kolektywnych zachowań grup społecznych kolonii owadów oraz innych społeczeństw zwierzęcych. Systemy SI składają się zazwyczaj z populacji prostych agentów (jednostek zdolnych do wykonywania pewnych operacji) oddziałujących lokalnie między sobą i ze swoim otoczeniem. Te jednostki o bardzo ograniczonych zdolnościach indywidualnych mogą wspólnie (kooperacyjnie) wykonywać wiele złożonych zadań niezbędnych do ich przetrwania. Chociaż zazwyczaj nie ma scentralizowanej struktury kontrolnej dyktującej jak poszczególni agenci powinni się zachowywać, lokalne interakcje pomiędzy takimi agentami często prowadzą do wyłonienia się globalnego i samoorganizującego się zachowania. Zaproponowano kilka algorytmów optymalizacyjnych inspirowanych zachowaniem roju w przyrodzie. Optymalizacja kolonii mrówek, optymalizacja roju cząstek, optymalizacja żerowania bakterii, optymalizacja kolonii pszczół, sztuczne systemy immunologiczne i optymalizacja oparta na biogeografii są przykładami tej metaheurystyki. [5]

Algorytmy te uzyskują optymalne rozwiązanie poprzez współpracę i ewolucję, dążąc do osiągnięcia skomplikowanych i wydajnych sposobów poszukiwania, składających się z prostych osobników i ograniczonej komunikacji. Algorytmy optymalizacyjne inteligencji roju stopniowo otrzymują coraz więcej uwagi badaczy akademickich i inżynierów zajmujących się optymalizacją z uwagi na ich istotne zalety:

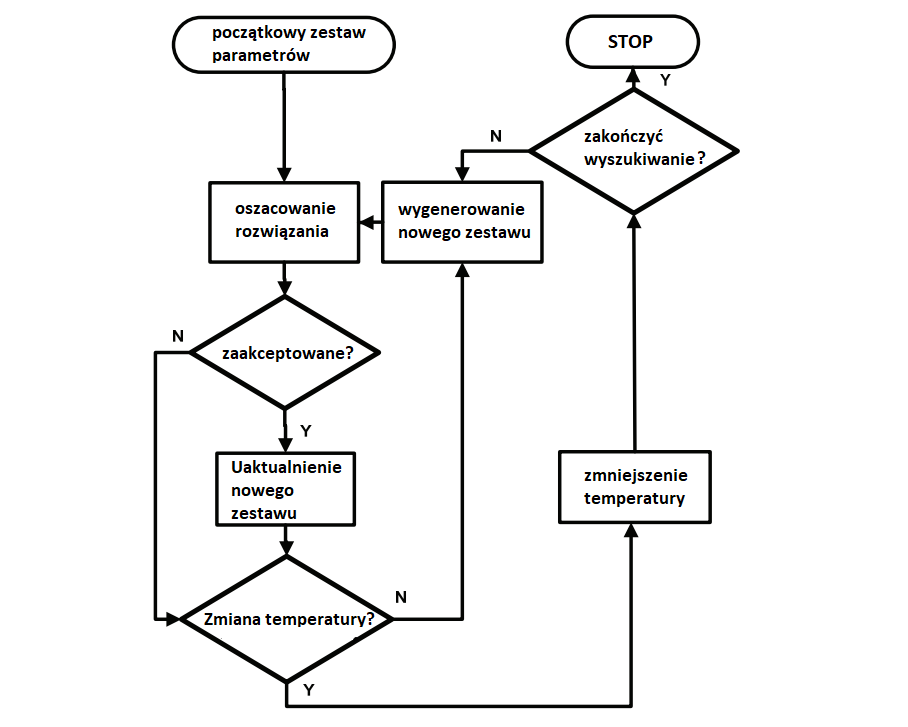
* Niskie wymagania dotyczące funkcji celu.
* Prostota.
* Obliczenia równoległe i możliwość rozproszenia. [10]

Algorytmy inteligencji roju charakteryzują się prostotą, niepewnością, interaktywnością, rozproszonym paralelizmem, odpornością, skalowalnością i samoorganizacją. Obecnie badania nad algorytmami inteligencji roju obejmują głównie teorię, algorytm i zastosowanie. Tendencje rozwojowe dotyczą opracowywania algorytmów hybrydowych, nowych ulepszonych algorytmów i analizy teoretycznej, a także rozwiązywania problemów wielkoskalowych (zastosowanie big data). Ogólnie rzecz biorąc, algorytmy inteligencji roju mogą rzucić światło na przełamanie klątwy braku darmowych obiadów\* (NFL), co pokazuje, że głębokie badania mogą dać nam wystarczającą antycypację motywującą coraz więcej naukowców do zaangażowania się w badania algorytmów inteligencji roju i ich zastosowań. [11]

\* teoria braku darmowych obiadów (No free lunch in search and optimization) - W złożoności obliczeniowej i optymalizacji twierdzenie o braku darmowych obiadów stwierdza, że ​​w przypadku niektórych rodzajów problemów matematycznych koszt obliczeniowy znalezienia rozwiązania, uśredniony dla wszystkich problemów w klasie, jest taki sam dla każdej metody rozwiązania. [12]

* + 1. **Symulowane wyżarzanie**

Wyżarzanie jest procesem metalurgicznym polegającym na ogrzewaniu ciała stałego, a następnie powolnym chłodzeniu, aż do jego krystalizacji. Atomy tego materiału mają wysoką energię przy bardzo wysokich temperaturach. To daje atomom dużą swobodę w ich zdolności do restrukturyzacji. W miarę jak temperatura jest obniżana, energia tych atomów maleje, aż do osiągnięcia stanu minimalnej energii. W kontekście optymalizacji symulowanego wyżarzania (simulated annealing, w skrócie SA) dąży do naśladowania tego procesu. SA rozpoczyna się w bardzo wysokiej temperaturze, gdzie wartości wejściowe mogą przyjmować duży zakres zmienności. W miarę postępu algorytmu temperatura zacznie spadać. Ogranicza to stopień w jakim dane wejściowe mogą się zmieniać. To często prowadzi algorytm do lepszego rozwiązania, tak jak metal osiąga lepszą strukturę krystaliczną poprzez proces wyżarzania. Tak więc, tak długo jak temperatura jest obniżana, na wejściach powstają zmiany prowadzące do lepszych rozwiązań. SA może być użyty do znalezienia minimum funkcji celu i oczekuje się, że algorytm znajdzie optymalne wartości wejściowe, gdy temperatura będzie bliska zero. Tak więc, funkcja celu jest wyrażeniem, które mierzy błąd pomiędzy danymi eksperymentalnymi i symulowanymi. Główną cechą algorytmu SA jest zdolność do unikania uwięzienia w lokalnym minimum. Dzieje się tak, pozwalając algorytmowi akceptować nie tylko rozwiązania lepsze, ale również gorsze z określonym prawdopodobieństwem. Główną wadą, która jest powszechna w stochastycznych algorytmach przeszukiwania lokalnego jest to, że definicja niektórych parametrów kontrolnych (temperatura początkowa, szybkość chłodzenia itp.) jest w pewnym sensie subiektywna i musi być określona na podstawie badań empirycznych. Oznacza to, że algorytm musi być dostrajany w celu zmaksymalizowania jego wydajności. [8]

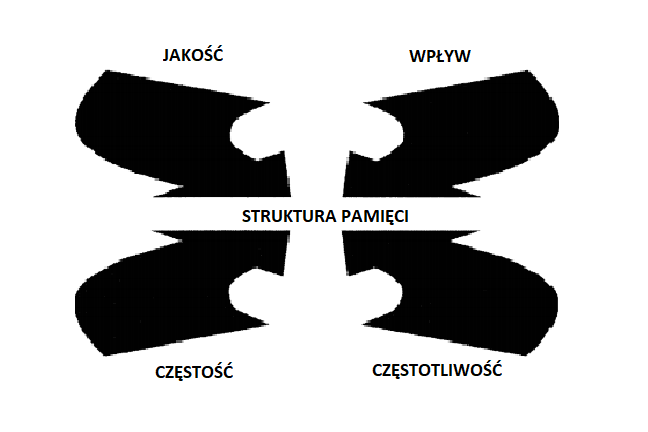


Rys. 3.2.2 Rysunek przedstawia algorytm symulowanego wyżarzania w postaci schematu blokowego [8]

* + 1. **Przeszukiwanie tabu**

Przeszukiwanie Tabu (ang. Tabu Search, w skrócie TS) jest metaheurystyką, która prowadzi lokalną procedurę heurystyczną do eksploracji przestrzeni rozwiązań. Procedura lokalna jest wyszukiwaniem, które wykorzystuje operację zwaną *ruch* ( move ) do określenia sąsiedztwa danego rozwiązania. Jednym z głównych elementów TS jest wykorzystanie pamięci adaptacyjnej, która tworzy bardziej elastyczne zachowanie podczas wyszukiwania. Strategie oparte na pamięci są więc cechą charakterystyczną metod przeszukiwania tabu. Słownik PWN definiuje tabu jako „zakaz stykania się z pewnymi przedmiotami, osobami, zwierzętami lub dokonywania pewnych czynności, którego naruszenie miało powodować karę sił nadnaturalnych” [50]. Poszukiwanie tabu ledwie odnosi się do łamania głębokich zakazów fundamentalnych, bądź kulturowych, ale zamiast tego dotyczy nakładania ograniczeń, aby kierować procesem wyszukiwania w celu pokonania trudnych regionów. Ograniczenia te funkcjonują w kilku formach, zarówno poprzez bezpośrednie wykluczenie alternatywnego wyszukiwania sklasyfikowanego jako „zakazane”, jak również przez przekształcenie w zmodyfikowane oceny i prawdopodobieństwo wyboru. Ograniczenia są nakładane lub tworzone poprzez odwoływanie się do struktur pamięci, które są zaprojektowane do tego konkretnego celu. [9]

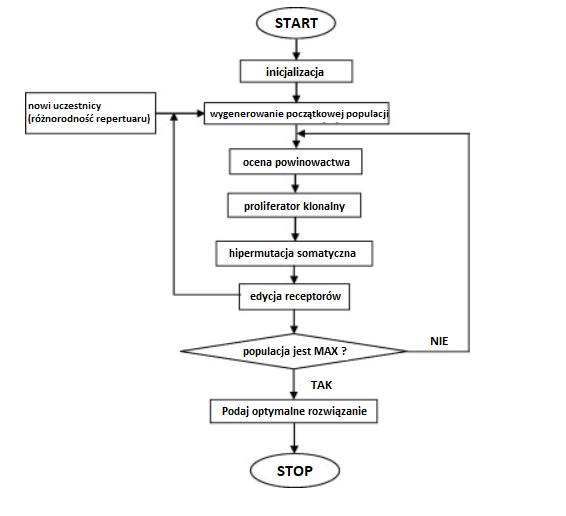
TS opiera się na założeniu, że rozwiązywanie problemów, aby mogło zostać uznane za inteligentne, musi zawierać *pamięć adaptacyjną* i *eksplorację responsywną*. Dobrą analogią jest wspinaczka górska, gdzie wspinacz musi selektywnie zapamiętywać (pamięć adaptacyjna) kluczowe elementy przebytej drogi i musi być w stanie dokonywać strategicznych wyborów (eksploracja responsywna) po drodze (rysunek 1.2). Cecha pamięci adaptacyjnej TS, której analogia do wspinaczki górskiej sugeruje znaczenie analizowania bieżących alternatyw w odniesieniu do poprzednich wejść na podobny teren. Umożliwia implementację procedur, które są w stanie przeszukiwać przestrzeń rozwiązań ekonomicznie i efektywnie. Ponieważ wybory lokalne dokonywane są na podstawie informacji zebranych podczas wyszukiwania, TS kontrastuje z konstrukcjami z mniejszą ilością pamięci, które w dużym stopniu polegają na procesach losowych. Pamięć adaptacyjna kontrastuje również z konstrukcjami o pamięci sztywnej typowymi dla strategii rozgałęzień i ograniczeń. Nacisk na eksplorację responsywną (i stąd cel) w przeszukiwaniu tabu, czy to w deterministycznej lub probabilistycznej implementacji, wywodzi się z założenia, że zły wybór strategiczny może przynieść więcej informacji niż dobry wybór losowy. (W systemie, który używa pamięci, zły wybór oparty na strategii może dostarczyć użytecznych wskazówek o tym, jak strategia może zostać poprawiona. Nawet w przestrzeni z dużą losowością – która na szczęście nie jest na tyle wszechobecna, by wygasić wszystkie pozostałości porządku w większości problemów świata rzeczywistego – przemyślany projekt może być bardziej sprawny w odkrywaniu śladów struktury, a tym samym w dawaniu szansy na wykorzystanie warunków, w których losowość nie jest wszechogarniająca). Eksploracja responsywna integruje podstawowe zasady inteligentnego wyszukiwania (tj. wykorzystywanie dobrych cech rozwiązania podczas eksploracji nowych, obiecujących obszarów). Przeszukiwanie Tabu zajmuje się znalezieniem nowych i bardziej efektywnych sposobów wykorzystania mechanizmów związanych zarówno z pamięcią adaptacyjną jak i eksploracją responsywną. Rozwój nowych konstrukcji i kombinacji strategicznych sprawia, że TS jest bogatym obszarem do badań i studiów empirycznych. Struktury pamięci w wyszukiwaniu tabu działają w odniesieniu do czterech głównych wymiarów, na które składają się: częstość, częstotliwość, jakość i wpływ (Rysunek 3.2.3). [9]



Rys.3.2.3 Rysunek przedstawiający cztery wymiary wyszukiwania tabu. [9]

* + 1. **Sztuczny układ odpornościowy**

Naturalny układ odpornościowy służy do ochrony organizmu przed szkodliwymi organizmami (zwanymi antygenami), a funkcję tę pełnią komórki odpornościowe limfocyty (głównie limfocyty B i T). Kiedy antygen wdziera się do organizmu, tylko nieliczne komórki odpornościowe mogą rozpoznać peptydy najeźdźcy. Rozpoznanie to stymuluje proliferację i różnicowanie się komórek, które wytwarzają pasujące do siebie klony (przeciwciała). Proces ten nazywany jest ekspansją klonalną, która generuje dużą populację komórek produkujących specyficzne dla danego antygenu przeciwciała. W wyniku klonalnej ekspansji komórek odpornościowych dochodzi do zniszczenia lub neutralizacji antygenu. Biologiczne zasady generowania klonów, proliferacji i dojrzewania są naśladowane i włączone do algorytmu obliczeniowego, który jest nazywany Sztucznym Systemem Odpornościowym (Artificial Immune System, w skrócie AIS). AIS jest wielomodelowym algorytmem optymalizacji funkcji, który imituje naturalny system odpornościowy. W oparciu o kilka zasad działania układu odpornościowego opracowano modele obliczeniowe, takie jak model sieci immunologicznej, algorytm selekcji negatywnej, algorytm selekcji pozytywnej i algorytm selekcji klonalnej. Udane zastosowania AIS obejmują wykrywanie anomalii, optymalizację, diagnozowanie błędów, rozpoznawanie wzorców itp. W szczególności AIS zapewnia doskonałą wydajność w rozwiązywaniu wielomodalnych, kombinatorycznych, zależnych od czasu problemów optymalizacyjnych. [13]

Rys. 3.2.4 Schemat blokowy proponowanego algorytmu AIS. [13]

* + 1. **Metaheurystyki hybrydowe**

W ostatnich latach pojawiło się wiele algorytmów, które nie są zgodne z paradygmatem pojedynczej tradycyjnej metaheurystyki. Wręcz przeciwnie, łączą one w sobie różne składniki algorytmiczne, często pochodzące z algorytmów innych obszarów badań nad optymalizacją. Podejścia te są powszechnie określane mianem metaheurystyk hybrydowych. Brak precyzyjnej definicji tego terminu był niekiedy przedmiotem krytyki. Należy jednak zauważyć, że stosunkowo otwarty charakter tego określenia może być pomocny, gdyż ścisłe granice pomiędzy pokrewnymi dziedzinami badań są często przeszkodą dla twórczego myślenia i eksploracji nowych kierunków badawczych. Główną motywacją stojącą za hybrydyzacją różnych algorytmów jest wykorzystanie komplementarnego charakteru różnych strategii optymalizacyjnych. Wybór odpowiedniej kombinacji komplementarnych koncepcji algorytmicznych może być kluczem do osiągnięcia najwyższej wydajności w rozwiązywaniu wielu trudnych problemów optymalizacyjnych. Niestety, opracowanie efektywnego podejścia hybrydowego jest na ogół trudnym zadaniem, wymagającym wiedzy z różnych dziedzin optymalizacji. Ponadto literatura pokazuje, że nie jest to trywialne do uogólnienia, to znaczy, że pewna hybryda może działać dobrze dla określonych problemów, ale może działać słabo dla innych. Niemniej jednak istnieją typy hybrydyzacji, które okazały się skuteczne w wielu zastosowaniach. Mogą one służyć jako wskazówki dla nowych rozwiązań. [7]

Kiedy badacze po raz pierwszy rozważali hybrydyzację preferowanej metaheurystyki z inną techniką optymalizacji większość z nich zaczęła szukać możliwych kombinacji z heurystykami lub innymi metaheurystykami. Obecnie hybrydyzacja różnych metaheurystyk jest bardzo rozpowszechniona, szczególnie jeśli chodzi o wykorzystanie metod przeszukiwania lokalnego w ramach metod populacyjnych. Zarówno obliczenia ewolucyjne jak i optymalizacja z wykorzystaniem kolonii mrówek często wykorzystują procedury wyszukiwania lokalnego do udoskonalania rozwiązań, które są generowane podczas procesu wyszukiwania. Można to przypisać faktowi, że te inspirowane naturą metody są dobre w eksploracji przestrzeni poszukiwań i identyfikacji obszarów o wysokiej jakości rozwiązań. Przy inicjalizacji starają się one na ogół uchwycić globalny obraz przestrzeni poszukiwań. Następnie w trakcie procesu wyszukiwania sukcesywnie skupiają poszukiwania na bardziej obiecujących regionach przestrzeni poszukiwań. Zwykle jednak nie są one tak efektywne w zakresie wykorzystania zgromadzonego doświadczenia do znajdowania najlepszych rozwiązań w tych wysokiej jakości obszarach. Mocną stroną przeszukiwania lokalnego jest możliwość szybkiego znalezienia lepszych rozwiązań w pobliżu danych rozwiązań początkowych. Podsumowując, metody populacyjne są dobre w identyfikowaniu obiecujących obszarów przestrzeni przeszukiwania, w których metody przeszukiwania lokalnego mogą następnie szybko wyznaczyć najlepsze rozwiązania. Dlatego też tego typu hybrydyzacja jest zazwyczaj bardzo udana. Algorytmy ewolucyjne wykorzystujące metody przeszukiwania lokalnego są czasami oznaczane jako algorytmy memetyczne (algorytm będący połączeniem algorytmu genetycznego i metod lokalnej optymalizacji). [7]

* + 1. **Algorytmy mrówkowe**

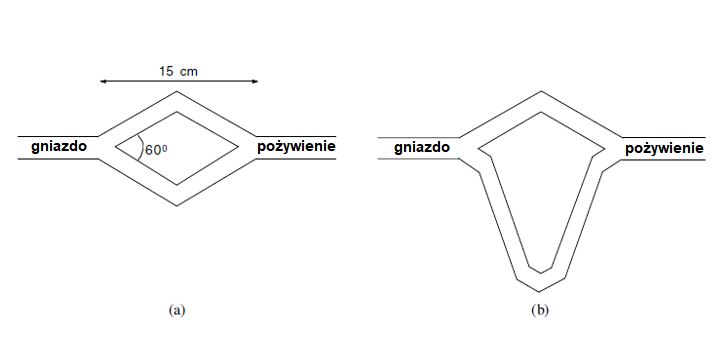
Pierwsza optymalizacja z wykorzystaniem kolonii mrówek (Ant Colony Optimization, w skrócie ACO) została zainspirowana badaniem zachowania mrówek w 1991 roku przez Macro Dorigo i współpracowników. Kolonia mrówek jest wysoce zorganizowaną grupą, w której jedne osobniki oddziałują z innymi za pomocą feromonów w doskonałej harmonii. Problemy optymalizacyjne mogą być rozwiązywane poprzez symulację zachowania mrówek. Od czasu zaproponowania pierwszego algorytmu nastąpił znaczny rozwój w dziedzinie ACO. [16]

Mrówki i inne owady żyjące w kolonii, takie jak pszczoły, termity i osy mogą być postrzegane jako systemy rozproszone, które pomimo prostoty każdego osobnika prezentują wysoki poziom społecznej organizacji. Niektóre przykłady możliwości kolonii mrówek to: podział pracy i przydzielanie zadań, sortowanie czerwiu, transport kooperacyjny i znajdowanie najkrótszej drogi pomiędzy dwiema (lub więcej) lokalizacjami (często między źródłem pokarmu a gniazdem). Pierwszy opracowany algorytm ACO został początkowo zastosowany do problemu komiwojażera. Algorytm ten opierał się na zdolności kolonii mrówek do znajdowania najkrótszej ścieżki pomiędzy źródłem pożywienia, a gniazdem. Algorytm wykorzystuje sztuczne mrówki, które współpracują w poszukiwaniu rozwiązania problemu poprzez komunikację, w której pośredniczą sztuczne szlaki feromonowe. Podczas poruszania się po grafie, sztuczne mrówki umieszczają feromon na krawędziach wyznaczających ścieżkę, którą mogą podążać inni członkowie kolonii, wzmacniając feromon na tej ścieżce. Dzięki tej stygmeri mrówki koordynują swoje działania. To samoorganizujące się zachowanie skutkuje w samowzmacniający się proces, który prowadzi do powstania ścieżki o wysokim stężeniu feromonu. Podczas gdy ścieżki, które są mniej używane, mają tendencję do zmniejszania się poziomu feromonu z powodu parowania. Koncepcja ta może być zastosowana do każdego problemu optymalizacji kombinatorycznej, dla którego można zdefiniować heurystykę. Proces konstruowania rozwiązań może być traktowany jako spacer po grafie konstrukcyjnym, gdzie każda krawędź grafu reprezentuje możliwy krok, który może wykonać mrówka. Algorytmy ACO były początkowo stosowane do rozwiązywania problemów optymalizacji kombinatorycznej zainspirowane zachowaniem związanym z wyznaczaniem ścieżek, a później zastosowane do wielu innych problemów. [15][14]

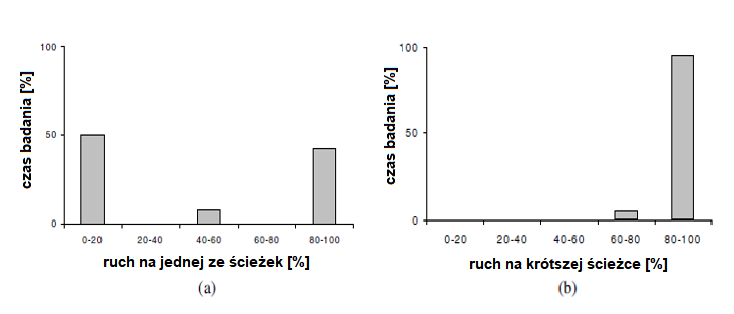
Zostało przeprowadzonych wiele badań na temat ścieżek feromonowych i podążających za nimi mrówkami. Jednym z nich jest eksperyment przeprowadzony przez Deneubourg i współpracowników (Deneubourg, Aron, Gross, Pasteels, 1990), którzy stworzyli ścieżkę od mrowiska do źródła pożywienia, przechodzącą przez dwa mosty. Eksperyment przeprowadzili w dwóch modyfikacjach, pierwsza zakładała stosunek długości obu mostów równy jeden (rys.1.1a). Druga z kolei zawierała stosunek długości mostów dwa do jednego (rys.1.1b). [1]

Na początku pierwszego eksperymentu, mrówki miały wolny wybór jednej z dwóch równych ścieżek. Procent mrówek podążających jedną z dwóch ścieżek zmieniał się w czasie. Wynik końcowy był taki (rys.1.2a), że pomimo wolności wyboru jednej z dwóch ścieżek, finalnie wszystkie mrówki poruszały tą samą ścieżką. Można to wytłumaczyć następująco, kiedy pierwsze mrówki rozpoczęły eksplorację trasy nie było jeszcze naniesionego feromonu na obu ścieżkach. Stąd stosunek prawdopodobieństwa wyboru jednej ze ścieżek wynosił 50:50. Można się tutaj odnieść do rozkładu prawdopodobieństwa przy rzucie monetą, w której możemy otrzymać dwa wyniki, orła lub reszkę. Stosunek będzie taki sam jak w przypadku naszego eksperymentu, ze względu na liczbę możliwych wyników do uzyskania. Natomiast w krótkich seriach może się zdarzyć pewne odchylenie od średniej, w postaci serii tych samych wyników. Na szczęście w długich seriach, nie musimy się o to martwić, gdyż funkcjonuje zasada powrotu do średniej, która gwarantuje nam otrzymanie prawdopodobieństwa na poziomie 50%. Natężenie pozostawionego feromonu będzie rozłożone na korzyść jednej z nich, co będzie w konsekwencji skutkowało chętnym wyborem tej ścieżki przez pozostałe mrówki. Co dobitnie pokazuje samodyscyplinę kolonii i komunikację poprzez stygmerię, mrówki kontrolują swoją aktywność wykorzystując komunikację pośrednią na którą wpływa środowisko w którym się poruszają. [1]

W drugim eksperymencie, jedna z dróg była dwukrotnie dłuższa, co powodowało po jakimś czasie wybór w większości przypadków krótszej trasy. Tak jak w pierwszym eksperymencie, mrówki opuszczając swoje mrowisko i zbliżając się do punktu podjęcia decyzji o wyborze jednej z dwóch ścieżek, które na początku bez naniesionego feromonu wydawały się identyczne, podejmowały czysto losową decyzję. Tak jak już wcześniej było wspomniane, rozkład prawdopodobieństwa w długim terminie pozostaje na poziomie 50%, tak w krótkim terminie będą występować anomalie powodujące faworyzowanie jednej ze ścieżek względem drugiej. Z kolei otrzymane wyniki z przebiegu doświadczenia wykazują znaczącą różnicę w stosunku do poprzedniego badania (rys.1.2.b), ponieważ jedna ze ścieżek jest krótsza od drugiej. Mrówki wybierające krótszą trasę szybciej zdobędą pożywienie i wrócą do mrowiska. Następnie te same mrówki podejmujące ponownie decyzję miedzy krótszą, a dłuższą ścieżką, będą się kierowały natężeniem feromonu, które będzie większe na tej ścieżce, która w tym samym czasie zostaną pokonana przez większą liczbę mrówek. Mamy tu do czynienia ze zjawiskiem zwanym autokatalizą, która charakteryzuję się wzrostem szybkości reakcji chemicznej pod wpływem wzrostu stężenia produktu będącego katalizatorem. Jeżeli porównamy ten eksperyment do eksperymentu z mostami o tej samej długości, to zauważymy jak mocno jest zredukowany wpływ początkowej losowej fluktuacji wybory jednej ze ścieżek, poprzez stygmerię, autokatalizę i różnicę długości mostów. Jeszcze wart odnotowania jest fakt, że mimo iż długość jednego z mostów jest dwukrotnie większa, to mały procent mrówek będzie wybierał tą dłuższa. Może to być intepretowane jako „ścieżki eksploracyjne”. [1]

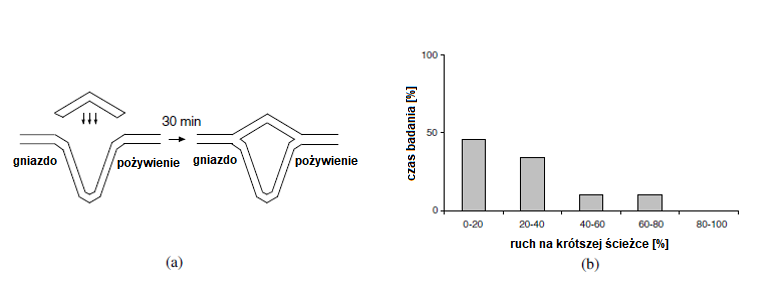


Rys. 3.2.5 Tor do przeprowadzenia eksperymentu podwójnego mostu. (a) Mosty są równej długości. (b) Jeden z mostów jest dwukrotnie większy od drugiego. [1]



Rys. 3.2.6 Wykresy przedstawiające wyniki przeprowadzonych eksperymentów. (a) Wynik dla przypadku obu mostów tej samej długości, na wykresie widać, że mrówki wybierały jedną z dwóch ścieżek na zmianę. (b) Wynik dla przypadku, gdy jeden z mostów jest dokładnie dwukrotnie dłuższy od drugiego, wykres obrazuje dominację krótszej ścieżki. [1]

Również interesujące jest przyjrzenie się sytuacji, w której między gniazdem i pożywieniem dodamy krótszą ścieżkę, w trakcje eksploracji terenu przez mrówki. Ten przypadek był badany jako eksperyment, w którym początkowo była udostępniona tylko jedna, dłuższa droga i po 30 minutach została dodana krótsza trasa (rys.1.3a). W tym przypadku, krótsza trasa była wybierania sporadycznie i cała kolonia mrówek poruszała się dłuższą trasą. Możemy to wytłumaczyć dużym natężeniem feromonu na dłuższej ścieżce i przez wolne odparowywanie feromonu. Większość mrówek wybierała dłuższą trasę ze względu na duże natężenie feromonu, które było kontynuowane autokatalitycznym zachowaniem wzmacniającym dłuższą ścieżkę, nawet pomimo pojawienia się lepszej, krótszej ścieżki. Parowanie feromonu, który służy eksploracji nowych ścieżek, jest zbyt wolne, co uniemożliwia kolonii mrówek „zapomnienie” optymalnej ścieżki, aby nowa, krótsza ścieżka mogła zostać odkryta. [1]



Rys.3.2.7 W tym eksperymencie tylko dłuższa ścieżka była dostępna od początku. Po 30 minutach, kiedy uformował się feromon na jedynej dostępnej ścieżce, została dodana krótsza ścieżka. (a) Początkowa faza przeprowadzania badania i zaktualizowana po 30 minutach, poprzez dodanie krótszej ścieżki. (b) W większości przypadków, po dodaniu krótszej ścieżki, większość mrówek wciąż chętniej wybierała dłuższą trasę. [1]

* 1. **Algorytmy genetyczne**

Na przestrzeni kilku pokoleń organizmy biologiczne ewoluują w oparciu o zasadę naturalnej selekcji "przetrwanie najsilniejszych", aby osiągnąć pewne niezwykłe cechy. Doskonałe kształty albatrosa, sprawność i podobieństwo między rekinami i delfinami, są najlepszymi przykładami osiągnięć przypadkowej ewolucji nad inteligencją. Działa ona tak dobrze w naturze, że w rezultacie powinno być interesujące symulowanie naturalnej ewolucji i opracowanie metody, która rozwiązuje konkretne poszukiwane problemy optymalizacyjne. Osobniki w populacji konkuruje ze sobą o pożywienie, schronienie itd. Również w tym samym gatunku jednostki konkurują aby przyciągnąć innych do reprodukcji. Ze względu na tę selekcję, słabo działające jednostki mają mniejsze szanse na przeżycie, a osobniki najlepiej przystosowane produkują stosunkowo dużą liczbę potomstwa. Po kilku generacjach gatunki ewoluują spontanicznie stając się coraz lepiej przystosowane do środowiska, w którym żyją. W 1975 roku Holland rozwinął tę ideę w swojej książce " Adaptacja w systemach naturalnych i sztucznych". Opisał jak zastosować zasady naturalnej ewolucji do problemów optymalizacyjnych i zbudował pierwsze algorytmy genetyczne. Teoria Hollanda została rozwinięta i obecnie Algorytmy Genetyczne (Genetic Algorithms, w skrócie GA) stanowią potężne narzędzie do rozwiązywania problemów wyszukiwania i optymalizacji. Algorytmy genetyczne oparte są na zasadzie genetyki i ewolucji. Siła matematyki leży w transferze technologii: istnieją pewne modele i metody, które opisują wiele różnych zjawisk i rozwiązują szeroką gamę problemów. GA są przykładem transferu technologii matematycznej: symulując ewolucję można rozwiązywać problemy optymalizacyjne z wielu różnych źródeł. [37]

Klasyczny Algorytm Genetyczny bazuje na zbiorze rozwiązań kandydujących do rozwiązania problemu optymalizacyjnego. Rozwiązanie jest potencjalnym kandydatem na optimum problemu optymalizacyjnego. Jego reprezentacja odgrywa istotną rolę, gdyż to ona decyduje o wyborze operatorów genetycznych. Reprezentacje są zazwyczaj listami wartości, a bardziej ogólnie opierają się na zbiorach symboli. Jeśli są one ciągłe to nazywamy je wektorami, jeśli składają się z bitów to nazywamy je ciągami bitowymi. Przykładem jest reprezentacja trasy w przypadku problemu komiwojażera. Operatory genetyczne produkują nowe rozwiązania w wybranej reprezentacji i umożliwiają spacer po przestrzeni rozwiązań. Zakodowanie rozwiązania jako reprezentacji, która podlega procesowi ewolucji jest nazywane genotypem lub chromosomem. [38]

Algorytmy genetyczne mogą być stosowane w problemach, w których rozwiązanie może być w przybliżeniu poprawne. Oznacza to, że odpowiedź ma pewien stopień błędu. W większości problemów optymalizacyjnych wiemy jak odpowiedź mogłaby wyglądać, ale nie znamy konkretnej wartości, więc zaczynamy od stosunkowo złej odpowiedzi i powoli pracujemy nad drogą do optymalnej. Dobrym przykładem jest algorytm stochastycznego zejścia gradientowego. W przypadku użycia algorytmu genetycznego do rozwiązania problemu optymalizacyjnego:

* Za możliwe rozwiązanie można uznać *osobnika*.
* *Populacja* będzie zbiorem możliwych rozwiązań.
* *Miara fitness* jest sposobem pomiaru jak dobre jest dane rozwiązanie.
* *Pokolenie* to poziom zaawansowania w kierunku pożądanego dopasowania.
* *Operacja wariacyjna* służy przybliżeniu do lepszego rozwiązania.

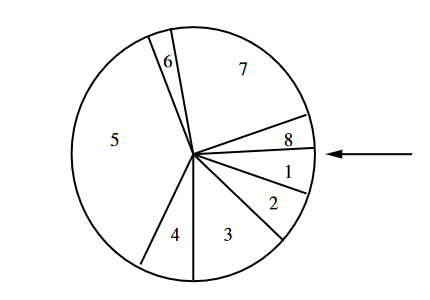
Terminologia:

* ***Chromosom:*** zbiór połączonych ze sobą genów, który przenosi informację genetyczną. Może to być fragment funkcji matematycznej z kilkoma parametrami.
* ***Krzyżowanie:*** wytwarzanie potomstwa poprzez łączenie rodziców. Może to być tworzenie nowej funkcji matematycznej poprzez łączenie zestawów parametrów z innych funkcji. Jest to jedna z głównych operacji wariacyjnych.
* ***Gen:*** nośnik informacji genetycznej dotyczącej określonej cechy, innymi słowy, jednostka dziedziczności. Może to być parametr w funkcji matematycznej, który wpływa na określoną zmienną.
* ***Genom:*** cała kombinacja informacji genetycznej dla danego osobnika. Może to być w całości funkcja matematyczna.
* ***Fitness:*** miara dopasowania osobnika. Może to być sposób w jaki rozwiązanie problemu optymalizacyjnego osiąga dobre wyniki.
* ***Mutacja:*** zmiana w sekwencji genów. Może to być zmiana parametru w funkcji matematycznej prowadząca do zmiany wyników. Jest to również główna operacja wariacyjna.
* ***Populacja:*** zbiór wszystkich osobników branych pod uwagę w badaniu. Może to być zbiór wielu rozwiązań problemu optymalizacyjnego, z których należy wybrać najlepsze.
* ***Kryterium zatrzymania:*** jest to sposób na zatrzymanie algorytmu, jeśli rozwiązanie zostało znalezione, osiągnięto limit dozwolonych generacji lub inne pożądane kryteria. Jest ono również znane jako "warunek zakończenia".
* ***Selekcja:*** metoda wyboru osobników, które zostaną wykorzystane w tworzeniu następnego pokolenia.
* ***Wariacja:*** polega na wprowadzaniu zmian do wybranych osobników, aby zmienić ich genom. Odbywa się to poprzez mutację lub krzyżowanie. [43]

Często stosowanymi operatorami genetycznymi są operatory selekcji, krzyżowania i mutacji. Są one stosowane na populacji chromosomów w celu uzyskania potencjalnie nowego potomstwa. Operatory te zostały opisane poniżej.

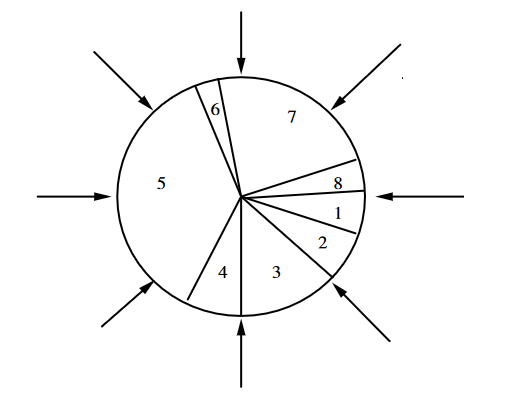
* + 1. **Selekcja**

Proces selekcji/reprodukcji kopiuje indywidualne ciągi (zwane chromosomami rodzicielskimi) do tymczasowej, nowej populacji (zwanej rozrodczą pulą). W celu przeprowadzenia operacji genetycznych. Liczba kopii, które osobnik otrzymuje w następnym pokoleniu jest zwykle przyjmowana jako wprost proporcjonalna do jego wartości, w ten sposób naśladując do pewnego stopnia procedurę selekcji naturalnej. Ten schemat jest powszechnie nazywany schematem selekcji proporcjonalnej. Koło ruletki, stochastyczna selekcja uniwersalna i binarna selekcja turniejowa są jednymi z najczęściej stosowanych procedur selekcyjnych. Selekcja koła ruletki działa jest kołem o takiej liczbie szczelin ile wynosi rozmiar populacji P, gdzie rozmiar szczeliny jest proporcjonalny do względnego dopasowania odpowiadającego mu chromosomu w populacji. Osobnik jest wybierany poprzez kręcenie ruletką i odnotowywanie pozycji markera, gdy ruletka się zatrzyma. Dlatego liczba przypadków, w których dany osobnik zostanie wybrany jest proporcjonalna do jego dopasowania (lub wielkości szczeliny) w populacji.



Rys. 3.3.1 Selekcja poprzez koło ruletki. [39]

W stochastycznej selekcji uniwersalnej, P jednakowo odległych markerów jest umieszczonych na kole. Wszystkie P osobników jest wybieranych przez kręcenie kołem przy czym liczba kopii, które otrzymuje osobnik jest równa liczbie markerów, które zatrzymają się w konkretnej szczelinie. Rysunek poniżej demonstruje tę procedurę selekcji. [39]

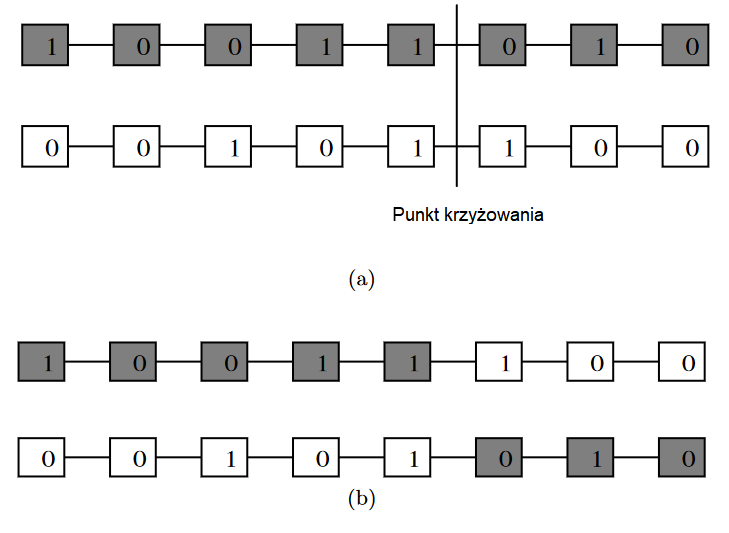


Rys. 3.3.2 Stochastyczna selekcja uniwersalna. [39]

* + 1. **Krzyżowanie**

Krzyżowanie jest operatorem, który pozwala na połączenie materiału genetycznego dwóch lub więcej rozwiązań. W naturze większość gatunków ma dwoje rodziców. W Algorytmach Genetycznych możemy nawet rozszerzyć operatory krzyżowania na więcej niż dwóch rodziców. Pierwszym krokiem w naturze jest wybór potencjalnego partnera. Wiele gatunków przeznacza wiele zasobów na procesy selekcji, ale także na wybór potencjalnego partnera i strategie przyciągania partnerów. W szczególności samce przeznaczają wiele zasobów na to, by zrobić wrażenie na samicach. Po wyborze partnera następnym naturalnym krokiem jest dobranie się w pary. Z biologicznego punktu widzenia dwóch partnerów tego samego gatunku łączy swój materiał genetyczny i przekazuje go potomstwu. [37]

Głównym celem krzyżowania jest wymiana informacji pomiędzy losowo wybranymi chromosomami rodzicielskimi poprzez rekombinację części ich materiału genetycznego. Operacja ta wykonywana probabilistycznie łączy części dwóch par chromosomów w celu wytworzenia potomstwa dla następnego pokolenia. Jednopunktowe krzyżowanie (Single-point crossover) jest jednym z najczęściej stosowanych schematów. W tym przypadku, w pierwszej kolejności z wybranych ciągów z puli rozrodczej są losowo łączone w pary. Następnie, aby wykonać krzyżowanie na parze wybiera się równomiernie liczbę całkowitą k (zwaną punktem krzyżowania), jest wybierana losowo z przedziału od 1 do l - 1, gdzie l jest długością łańcucha. Dwa nowe ciągi są tworzone przez zamianę wszystkich znaków od pozycji (k + 1) do l. Na przykładzie dwóch łańcuchów macierzystych jak pokazano na rys. 3.4.3(a), w wyniku operacji krzyżowania powstają potomstwa pokazane na rys. 3.4.3(b). Inne popularne techniki krzyżowania to krzyżowanie dwupunktowe, krzyżowanie wielopunktowe, tasowanie-wymiana oraz jednolite krzyżowanie. [39]

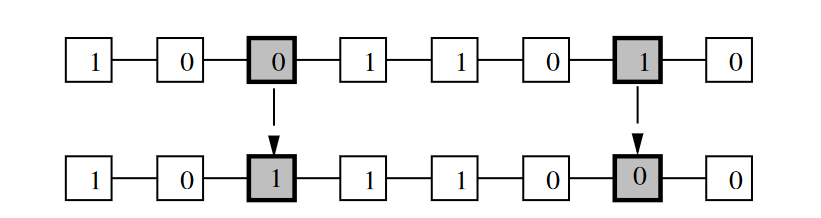


Rys. 3.3.3 Przykład działania jednopunktowego krzyżowania: (a) przed krzyżowaniem; (b) po krzyżowaniu. [39]

Sukces GA zależy w znacznym stopniu od techniki kodowania zastosowanej do reprezentacji zmiennych problemu. Hipoteza bloków konstrukcyjnych wykazuje, że GA działają poprzez identyfikację dobrych bloków konstrukcyjnych, a następnie przez równomierne łączenie ich w celu uzyskania większych bloków. O ile dobre bloki konstrukcyjne nie są ściśle zakodowane, operacja krzyżowania nie może ich połączyć. Zatem interakcja kodowanie - krzyżowanie jest ważna dla pomyślnego działania GA. Problem ciasnego lub luźnego kodowania zmiennych jest powszechnie znany jako problem powiązania. [39]

* + 1. **Mutacja**

Mutacja to proces, w którym dochodzi do przypadkowej zmiany w strukturze genetycznej chromosomu. Jej głównym celem jest wprowadzenie różnorodności genetycznej do populacji. Może się zdarzyć, że optymalne rozwiązanie znajduje się w części przestrzeni poszukiwań, która nie jest reprezentowana w strukturze genetycznej populacji. Proces nie będzie więc w stanie osiągnąć optimum globalnego. W takiej sytuacji jedynie mutacja może ewentualnie skierować populację do optymalną część przestrzeni poszukiwań poprzez losową zmianę informacji w chromosomie. Mutacja genu binarnego polega na prostej negacji bitu, podczas gdy mutacje dla genów zakodowanych w rzeczywistości są definiowane na wiele sposobów. Tutaj omówię binarną mutację bit po bicie, gdzie każdy bit w chromosomie podlega mutacji z (zazwyczaj niskim) prawdopodobieństwem. Przykład binarnej mutacji bit po bicie jest pokazany na rys. 3.3.4. Pozycje 3 i 7 chromosomu pokazanego powyżej zostały poddane mutacji. [39]



Rys. 3.3.4 Przykład mutacji bit po bicie. [18]

Chociaż wszystkie algorytmy genetyczne oparte są na tej samej koncepcji, ich konkretne implementacje mogą się dość znacznie od siebie różnić. Jednym ze sposobów, w jaki konkretne implementacje mogą się różnić, są ich parametry. Podstawowy algorytm genetyczny będzie miał co najmniej kilka parametrów, które muszą być brane pod uwagę podczas implementacji.

* + 1. **Wskaźnik mutacji**

Współczynnik mutacji to prawdopodobieństwo, w którym określony gen w chromosomie danego rozwiązania zostanie zmutowany. Technicznie nie ma poprawnej wartości dla współczynnika mutacji algorytmu genetycznego, ale niektóre współczynniki mutacji zapewniają znacznie lepsze wyniki niż pozostałe. Wyższy współczynnik mutacji pozwala na większą różnorodność genetyczną w populacji i może również pomóc algorytmowi uniknąć lokalnego optimum. Jednakże, zbyt wysoki współczynnik mutacji może spowodować zbyt dużą zmienność genetyczną między każdym pokoleniem powodując utratę dobrych rozwiązań znalezionych w poprzedniej populacji. Jeśli współczynnik mutacji jest zbyt niski algorytm może potrzebować nieracjonalnie dużo czasu na poruszanie się po przestrzeni wyszukiwania, utrudniając znalezienie satysfakcjonującego rozwiązania. Zbyt wysoki współczynnik mutacji może również wydłużyć czas potrzebny na znalezienie akceptowalnego rozwiązania. Chociaż wysoki współczynnik mutacji może pomóc algorytmowi genetycznemu uniknąć utknięcia w lokalnym optimum, to gdy jest on ustawiony na zbyt wysokim poziomie, może mieć negatywny wpływ na wyszukiwanie. Jest to spowodowane tym, że rozwiązania w każdym pokoleniu są mutowane w tak dużym stopniu, że są praktycznie losowe po zastosowaniu mutacji.

Aby zrozumieć dlaczego dobrze skonfigurowany współczynnik mutacji jest ważny, należy rozważyć dwa binarnie zakodowane potencjalne rozwiązania, "100" i "101". Bez mutacji nowe rozwiązania mogą pochodzić tylko z krzyżowania. Jednakże, kiedy krzyżujemy nasze rozwiązania istnieją tylko dwa możliwe wyniki dostępne dla potomstwa, "100" lub "101". Dzieje się tak dlatego, że jedyna różnica w genomach rodziców znajduje się w ich ostatnich bitach. Jeśli potomek otrzyma swój ostatni bit od pierwszego rodzica, będzie to "1", w przeciwnym razie, będzie to "0". Gdyby algorytm potrzebował znaleźć alternatywne rozwiązanie, musiałby zmutować istniejące rozwiązanie dając mu nową informację genetyczną, która nie jest dostępna w innym miejscu puli genów.

Współczynnik mutacji powinien być ustawiony na taką wartość, która pozwala na wystarczającą różnorodność, aby zapobiec ustabilizowaniu się algorytmu, ale nie na tyle, aby spowodować utratę przez algorytm cennej informacji genetycznej z poprzedniej populacji. Równowaga ta będzie zależała od natury rozwiązywanego problemu. [38]

* + 1. **Współczynnik krzyżowania**

Częstotliwość, z jaką stosowane jest krzyżowanie, również ma wpływ na ogólną wydajność algorytmu genetycznego. Zmiana współczynnika krzyżowania pozwala na dostosowanie szansy, w jakiej rozwiązania w populacji będą miały zastosowany operator krzyżowania. Wysoki współczynnik pozwala na znalezienie wielu nowych, potencjalnie lepszych rozwiązań podczas fazy krzyżowania. Niższy współczynnik pomoże zachować informację genetyczną z dopasowanych osobników nienaruszoną dla następnego pokolenia. Współczynnik krzyżowania powinien być zazwyczaj ustawiony na rozsądnie wysoki współczynnik promujący poszukiwanie nowych rozwiązań, pozwalając jednocześnie na zachowanie niewielkiego procentu populacji nienaruszonej dla następnego pokolenia. [38]

* + 1. **Rozmiar populacji**

Wielkość populacji to po prostu liczba osobników w populacji algorytmu genetycznego w danym pokoleniu. Im większy rozmiar populacji, tym większą część przestrzeni wyszukiwania algorytm może próbkować. Pomoże to w uzyskaniu dokładniejszych i globalnie optymalnych rozwiązań. Mały rozmiar populacji często powoduje, że algorytm znajduje mniej pożądane rozwiązania w lokalnie optymalnych obszarach przestrzeni poszukiwań, jednak wymagają one mniej zasobów obliczeniowych na generację. Również tutaj, podobnie jak w przypadku współczynnika mutacji, należy znaleźć równowagę dla optymalnej wydajności algorytmu genetycznego. Wymagany rozmiar populacji będzie się zmieniał w zależności od charakteru rozwiązywanego problemu. Duże, pagórkowate przestrzenie poszukiwań zwykle wymagają większej liczebności populacji, aby znaleźć najlepsze rozwiązania. Co ciekawe, podczas wybierania wielkości populacji istnieje punkt, w którym zwiększenie jej rozmiaru przestanie zapewniać algorytmowi znaczną poprawę dokładności znajdowanych rozwiązań. Zamiast tego spowolni wykonywanie algorytmu z powodu dodatkowego zapotrzebowania obliczeniowego potrzebnego do przetworzenia dodatkowych osobników. Wielkość populacji w okolicach tego przejścia zwykle zapewnia najlepszą równowagę między zasobami a wynikami. [38]

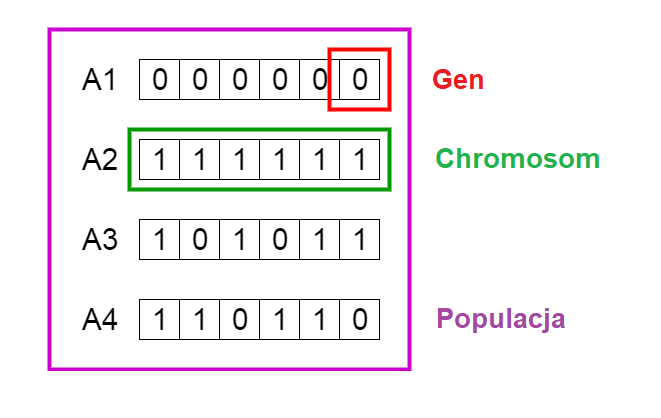
* + 1. **Funkcja fitness**

Funkcja fitness (fitness function) jest ważnym zagadnieniem w rozwiązywaniu problemów optymalizacyjnych z wykorzystaniem algorytmu genetycznego. Często konieczne jest odwzorowanie naturalnej funkcji celu na funkcję dopasowania poprzez jedno lub więcej odwzorowań. Pierwsze odwzorowanie jest dokonywane w celu przekształcenia funkcji celu w problem maksymalizacji, a nie minimalizacji, aby dopasować koncepcje wyboru najbardziej dopasowanego chromosomu, który ma najwyższą funkcję celu. Drugim ważnym odwzorowaniem jest skalowanie wartości funkcji dopasowania. Skalowanie jest ważnym krokiem podczas procedur przeszukiwania GA. Ma to na celu utrzymanie odpowiedniego poziomów konkurencji w całej symulacji. Bez skalowania na początku istnieje tendencja do zdominowania procesu selekcji przez kilka super jednostek. Później, gdy populacja w dużej mierze się zbiegła konkurencja wśród członków populacji jest mniej silna i symulacja ma tendencję do błądzenia. Skalowanie jest więc użytecznym procesem zapobiegającym zarówno przedwczesnej zbieżności algorytmu, jak i przypadkowej poprawie, która może wystąpić w późnych iteracjach algorytmu. Istnieje wiele metod skalowania, takich jak skalowanie liniowe, skrócenie sigma i prawo siły. [38]

Każdy problem ma swoją własną funkcję fitness, która powinna być użyta zależy od danego problemu. Wymyślenie funkcji fitness dla danego problemu jest najtrudniejszą częścią, jeśli chodzi o formułowanie problemu z wykorzystaniem algorytmów genetycznych. Nie ma prostej i skutecznej reguły, że dana funkcja powinna być użyta w konkretnym problemie. Jednakże pewne funkcje zostały przyjęte przez naukowców zajmujących się danymi w odniesieniu do pewnych typów problemów. Zazwyczaj dla zadań klasyfikacji, w których wykorzystywane jest uczenie nadzorowane, miary błędu, takie jak odległość euklidesowa i odległość Manhattanu są powszechnie stosowane jako funkcje fitness.. Poniższe wymagania powinny być spełnione przez każdą funkcję fitness:

* Funkcja fitness powinna być jasno zdefiniowana. Czytelnik powinien być w stanie jasno zrozumieć, jak wynik funkcji jest obliczany.
* Funkcja fitness powinna być zaimplementowana efektywnie. Jeśli funkcja fitness stanie się wąskim gardłem algorytmu, wówczas ogólna wydajność algorytmu genetycznego zostanie zmniejszona.
* Funkcja fitness powinna mierzyć ilościowo, jak bardzo dane rozwiązanie jest dopasowane do rozwiązania problemu.
* Funkcja fitness powinna generować intuicyjne wyniki. Najlepsi/najgorsi kandydaci powinni mieć najlepsze/najgorsze wartości punktowe. [43]
  + 1. **Opis działania algorytmu genetycznego**

Algorytm genetyczny pracuje na populacji składającej się z pewnych rozwiązań, gdzie rozmiar populacji jest liczbą rozwiązań. Każde rozwiązanie nazywane jest osobnikiem. Każde rozwiązanie indywidualne posiada chromosom. Chromosom jest reprezentowany jako zbiór parametrów (cech), które definiują dany osobnik. Każdy chromosom posiada zestaw genów. Każdy gen jest reprezentowany w jakiś sposób, np. jest reprezentowany jako ciąg 0 i 1, jak na poniższym diagramie.

****

Rys. 3.3.5 Populacja, chromosomy i geny. [49]

Każdy osobnik ma również swoją wartość fitness. Aby wybrać najlepsze osobniki, stosuje się funkcję fitness. Wynikiem działania funkcji fitness jest wartością fitness reprezentująca jakość rozwiązania. Im wyższa wartość fitness, tym wyższa jakość rozwiązania. Selekcja najlepszych osobników na podstawie ich jakości jest stosowana do generowania tzw. puli rozrodczej, gdzie osobnik o wyższej wartości fitness ma większe prawdopodobieństwo bycia wybranym do puli rozrodczej. Osobniki w puli rozrodczej nazywane są rodzicami. Każda dwójka rodziców wybranych z puli rozrodczej wygeneruje dwoje potomstwa (dzieci). Poprzez kojarzenie tylko osobników z wysoką wartością fitness, oczekuje się, że potomstwo będzie miało lepszą jakość rozwiązania niż jego rodzice. To ograniczy wpływ słabszych osobników na generowanie potomstwa. Poprzez ciągłe wybieranie i kojarzenie osobników wysokiej jakości, będą większe szanse na zachowanie tylko dobrych cech osobników i pominięcie złych. W końcu skończy się to pożądanym optymalnym lub akceptowalnym rozwiązaniem. Aczkolwiek potomstwo obecnie generowane przy użyciu wybranych rodziców po prostu mają tylko cechy swoich rodziców. Potomstwo nie posiada żadnych nowych, unikatowych cech, a zatem te same wady rodziców będą dziedziczone przez potomstwo. Aby przezwyciężyć ten problem, do każdego z potomstwa zostaną wprowadzone pewne zmiany, aby stworzyć bardziej unikatowe osobniki. Zbiór wszystkich nowo wygenerowanych osobników będzie nową populacją, która zastąpi poprzednio używaną starą populację. Każda utworzona populacja nazywana jest pokoleniem. Proces zastępowania starej populacji przez nową nazywany jest wymianą. Poniższy pseudokod i diagram podsumowują kroki działania algorytmu. [42]

begin

count = 0

zainicjuj populację

oblicz wartość fitness populacji

while not warunek zakończenia do

begin

count = count + 1

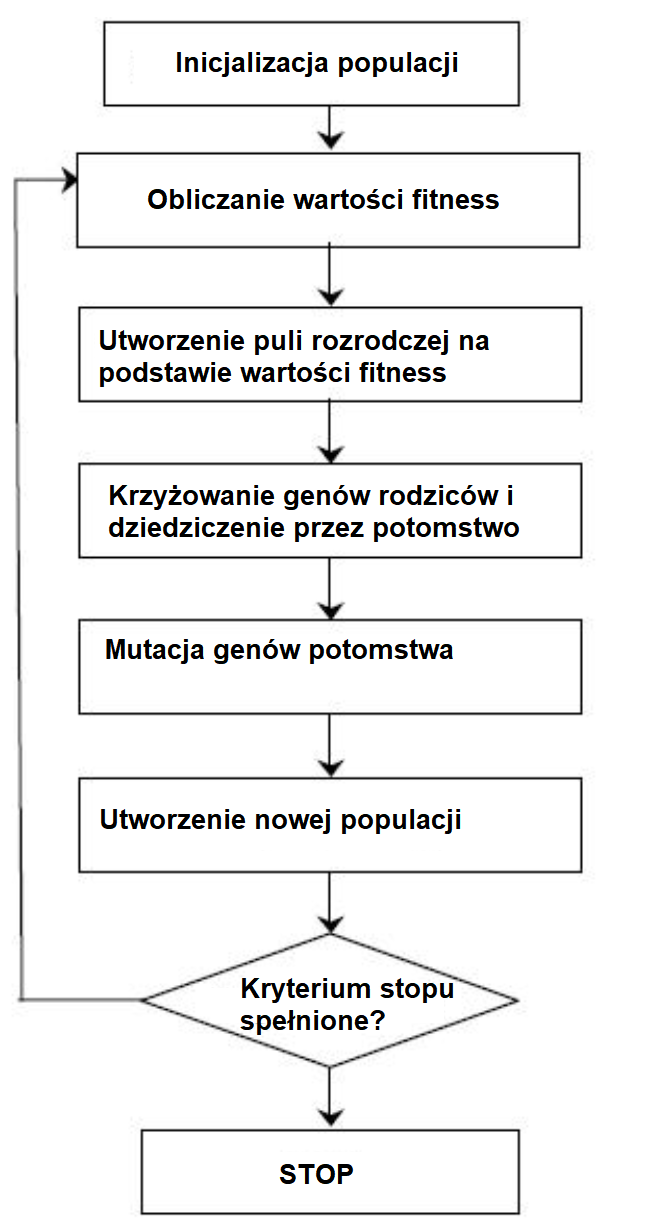
wybierz osobniki do reprodukcji

zastosuj operatory zmienności

oblicz wartość fitness potomstwa

end

end



Rys. 3.3.6 Schemat blokowy działania algorytmu mrówkowego. [51]

1. **Implementacja algorytmu**
   1. **Język programowania i środowisko programistyczne**

**Python** – [język programowania](https://pl.wikipedia.org/wiki/J%C4%99zyk_programowania) [wysokiego poziomu](https://pl.wikipedia.org/wiki/J%C4%99zyk_wysokiego_poziomu) ogólnego przeznaczenia, o rozbudowanym pakiecie [bibliotek standardowych](https://pl.wikipedia.org/wiki/Biblioteka_standardowa), którego ideą przewodnią jest czytelność i klarowność [kodu źródłowego](https://pl.wikipedia.org/wiki/Kod_%C5%BAr%C3%B3d%C5%82owy). Jego składnia cechuje się przejrzystością i zwięzłością. Python wspiera różne [paradygmaty programowania](https://pl.wikipedia.org/wiki/Paradygmat_programowania): [obiektowy](https://pl.wikipedia.org/wiki/Programowanie_obiektowe), [imperatywny](https://pl.wikipedia.org/wiki/Programowanie_imperatywne) oraz w mniejszym stopniu [funkcyjny](https://pl.wikipedia.org/wiki/Programowanie_funkcyjne). Posiada w pełni [dynamiczny](https://pl.wikipedia.org/wiki/Typowanie_dynamiczne) [system typów](https://pl.wikipedia.org/wiki/System_typ%C3%B3w) i automatyczne [zarządzanie pamięcią](https://pl.wikipedia.org/wiki/Od%C5%9Bmiecanie_pami%C4%99ci), będąc w tym podobnym do języków [Perl](https://pl.wikipedia.org/wiki/Perl), [Ruby](https://pl.wikipedia.org/wiki/Ruby_(j%C4%99zyk_programowania)), [Scheme](https://pl.wikipedia.org/wiki/Scheme) czy [Tcl](https://pl.wikipedia.org/wiki/Tcl_(j%C4%99zyk_programowania)). Podobnie jak inne [języki dynamiczne](https://pl.wikipedia.org/wiki/Dynamiczny_j%C4%99zyk_programowania) jest często używany jako [język skryptowy](https://pl.wikipedia.org/wiki/J%C4%99zyk_skryptowy). [Interpretery](https://pl.wikipedia.org/wiki/Interpreter_(program_komputerowy)) Pythona są dostępne na wiele [systemów operacyjnych](https://pl.wikipedia.org/wiki/System_operacyjny). Python rozwijany jest jako projekt [Open Source](https://pl.wikipedia.org/wiki/Otwarte_oprogramowanie) zarządzany przez [Python Software Foundation](https://pl.wikipedia.org/wiki/Python_Software_Foundation), która jest [organizacją non-profit](https://pl.wikipedia.org/wiki/Organizacja_non-profit). Standardową implementacją języka jest [CPython](https://pl.wikipedia.org/wiki/CPython) (napisany w [C](https://pl.wikipedia.org/wiki/C_(j%C4%99zyk_programowania))), ale istnieją też inne, np. Jython (napisany w [Javie](https://pl.wikipedia.org/wiki/Java)).

Pythona stworzył we wczesnych latach 90. [Guido van Rossum](https://pl.wikipedia.org/wiki/Guido_van_Rossum) – jako następcę [języka ABC](https://pl.wikipedia.org/wiki/ABC_(j%C4%99zyk_programowania)), stworzonego w [Centrum voor Wiskunde en Informatica](https://pl.wikipedia.org/w/index.php?title=Centrum_voor_Wiskunde_en_Informatica&action=edit&redlink=1) (CWI – Centrum Matematyki i Informatyki w [Amsterdamie](https://pl.wikipedia.org/wiki/Amsterdam)). Van Rossum jest głównym twórcą Pythona, choć spory wkład w jego rozwój pochodzi od innych osób. Nazwa języka nie pochodzi od zwierzęcia lecz od serialu komediowego emitowanego w latach siedemdziesiątych przez [BBC](https://pl.wikipedia.org/wiki/BBC) – „Monty Python’s Flying Circus” ([Latający cyrk Monty Pythona](https://pl.wikipedia.org/wiki/Lataj%C4%85cy_cyrk_Monty_Pythona)). Projektant, będąc fanem serialu i poszukując nazwy krótkiej, unikalnej i nieco tajemniczej uznał tą za świetną.

Python realizuje jednocześnie kilka paradygmatów. Podobnie do [C++](https://pl.wikipedia.org/wiki/C%2B%2B), a w przeciwieństwie do [Smalltalka](https://pl.wikipedia.org/wiki/Smalltalk) nie wymusza jednego stylu programowania. W Pythonie możliwe jest [programowanie obiektowe](https://pl.wikipedia.org/wiki/Programowanie_obiektowe), [programowanie strukturalne](https://pl.wikipedia.org/wiki/Programowanie_strukturalne) i [programowanie funkcyjne](https://pl.wikipedia.org/wiki/Programowanie_funkcyjne). Typy sprawdzane są dynamicznie, a do zarządzania pamięcią stosuje się [garbage collection](https://pl.wikipedia.org/wiki/Od%C5%9Bmiecanie_pami%C4%99ci). Choć w jego popularyzacji kładzie się nacisk na różnice w stosunku do [Perla](https://pl.wikipedia.org/wiki/Perl), Python jest pod wieloma względami do niego podobny. Jednakże projektanci Pythona odrzucili złożoną składnię Perla na rzecz bardziej oszczędnej i – ich zdaniem – bardziej czytelnej. Mimo że podobnie do Perla, Python jest czasem klasyfikowany jako język skryptowy, wykorzystuje się go do tworzenia dużych projektów jak [serwer aplikacji](https://pl.wikipedia.org/wiki/Serwer_aplikacji) [Zope](https://pl.wikipedia.org/wiki/Zope), system wymiany plików [Mojo Nation](https://pl.wikipedia.org/w/index.php?title=Mojo_Nation&action=edit&redlink=1) czy nawet oprogramowanie klasy ERP – [Odoo](https://pl.wikipedia.org/wiki/Odoo).

W Pythonie wartości, a nie zmienne, posiadają typ – tak więc Python jest językiem z typami dynamicznymi, podobnie jak [Lisp](https://pl.wikipedia.org/wiki/Lisp), a w przeciwieństwie do [Javy](https://pl.wikipedia.org/wiki/Java). W przeciwieństwie do wielu języków, wartości nie są przekazywane ani przez wartość, ani przez referencję, ale przez przypisanie. Reguły składniowe Pythona umożliwiają wyrażanie pojęć bez pisania dodatkowego kodu. Dla typów numerycznych zdefiniowana jest automatyczna konwersja, tak więc możliwe jest np. mnożenie liczby zespolonej przez liczbę całkowitą typu long bez rzutowania. Jednak w przeciwieństwie do Perla nie ma np. automatycznej konwersji pomiędzy napisami i liczbami; liczba nie jest prawidłowym argumentem dla operacji napisowej. Python oferuje szeroki zakres podstawowych typów danych – w tym typy liczbowe (całkowite, zmiennoprzecinkowe, [zespolone](https://pl.wikipedia.org/wiki/Liczby_zespolone)) oraz kolekcje. [28]

**PyCharm** – [zintegrowane środowisko programistyczne](https://pl.wikipedia.org/wiki/Zintegrowane_%C5%9Brodowisko_programistyczne) (IDE) dla [języka programowania](https://pl.wikipedia.org/wiki/J%C4%99zyk_programowania) [Python](https://pl.wikipedia.org/wiki/Python) firmy [JetBrains](https://pl.wikipedia.org/wiki/JetBrains). Zapewnia m.in.: edycję i analizę kodu źródłowego, graficzny [debugger](https://pl.wikipedia.org/wiki/Debugger), uruchamianie [testów jednostkowych](https://pl.wikipedia.org/wiki/Test_jednostkowy), integrację z [systemem kontroli wersji](https://pl.wikipedia.org/wiki/System_kontroli_wersji). Wspiera także programowanie i tworzenie aplikacji internetowych w [Django](https://pl.wikipedia.org/wiki/Django_(framework)). Jest [oprogramowaniem wieloplatformowym](https://pl.wikipedia.org/wiki/Wieloplatformowo%C5%9B%C4%87) pracującym na platformach systemowych: [Microsoft Windows](https://pl.wikipedia.org/wiki/Microsoft_Windows), [GNU/Linux](https://pl.wikipedia.org/wiki/GNU/Linux) oraz [macOS](https://pl.wikipedia.org/wiki/MacOS).

* 1. **Biblioteki i technologie**

**NumPy**, co jest skrótem od Numerical Python, jest biblioteką składającą się z obiektów wielowymiarowych tablic oraz zbioru procedur do przetwarzania tych tablic. Za pomocą NumPy można wykonywać matematyczne i logiczne operacje na tablicach. NumPy jest pakietem Pythona. Jest to skrót od "Numerical Python". Jest to biblioteka składająca się z wielowymiarowych obiektów tablicowych oraz kolekcji procedur do przetwarzania tablic. Numeric, przodek NumPy, został opracowany przez Jima Hugunina. Powstał także inny pakiet Numarray, posiadający kilka dodatkowych funkcjonalności. W 2005 roku Travis Oliphant stworzył pakiet NumPy poprzez włączenie funkcji pakietu Numarray do pakietu Numeric. Jest wielu współtwórców tego projektu open-source. [30]

**Matplotlib.pyplot** jest zbiorem funkcji, które sprawiają, że matplotlib działa jak MATLAB. Każda funkcja pyplot dokonuje jakiejś zmiany w figurze: np. tworzy figurę, tworzy obszar plotowania w figurze, rysuje jakieś linie w obszarze plotowania, dekoruje wykres etykietami, itp. W matplotlib.pyplot różne stany są zachowywane przez wywołania funkcji, tak że śledzi on takie rzeczy jak bieżąca figura i obszar plotowania, a funkcje plotowania są kierowane na bieżące osie ("osie" odnoszą się do części osiowej figury, a nie ścisłego matematycznego terminu dla więcej niż jednej osi). [31]

**Random** jest wbudowanym modułem Pythona, który jest używany do generowania liczb losowych. Są to liczby pseudolosowe, co oznacza, że nie są one prawdziwie losowe. Moduł ten może być używany do wykonywania losowych działań, takich jak generowanie liczb losowych, drukowanie losowych wartości dla listy lub łańcucha, itp. [32]

Moduł **math** zapewnia dostęp do funkcji matematycznych zdefiniowanych przez standard C. Funkcje te nie mogą być używane z liczbami złożonymi; jeśli potrzebujesz obsługi liczb złożonych, użyj funkcji o tej samej nazwie z modułu cmath. Rozróżnienie pomiędzy funkcjami, które obsługują liczby złożone, a tymi, które ich nie obsługują, zostało wprowadzone, ponieważ większość użytkowników nie chce uczyć się tyle matematyki, ile jest wymagane do zrozumienia liczb złożonych. Otrzymanie wyjątku zamiast wyniku złożonego pozwala na wcześniejsze wykrycie nieoczekiwanej liczby złożonej użytej jako parametr, dzięki czemu programista może ustalić, jak i dlaczego została ona wygenerowana w pierwszej kolejności. Poniższe funkcje są dostarczane przez ten moduł. O ile wyraźnie nie zaznaczono inaczej, wszystkie zwracane wartości są zmiennoprzecinkowe. [33]

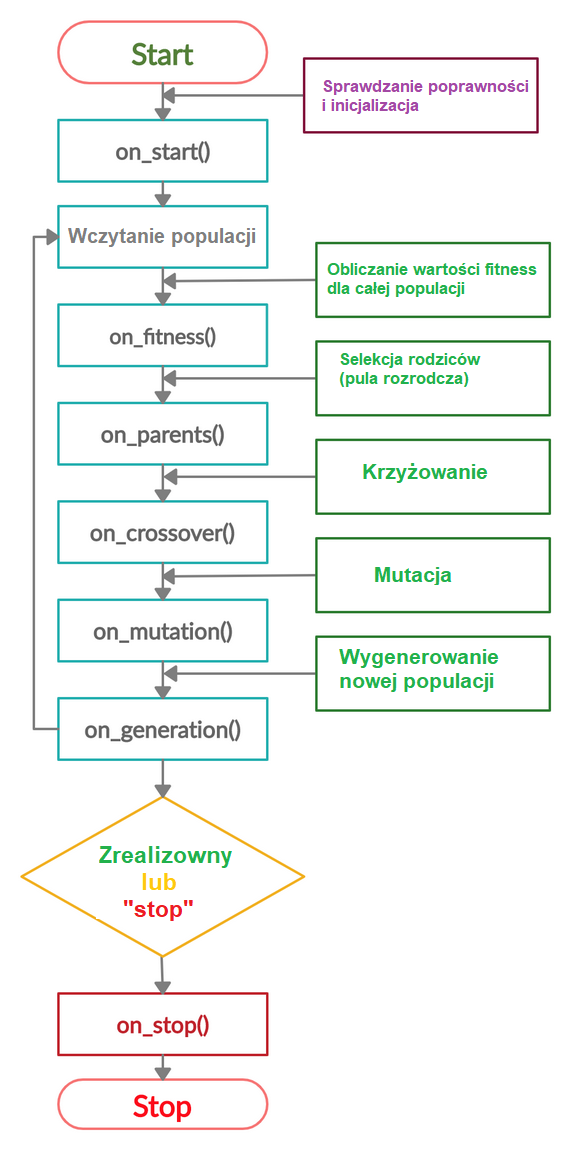
Moduł **time** dostarcza wielu funkcji, które zajmują się datami i czasem w ciągu dnia. Jest to cienka warstwa na wierzchu biblioteki runtime języka C. Dana data i czas mogą być reprezentowane jako wartość zmiennoprzecinkowa (liczba sekund od daty odniesienia, zwykle 1 stycznia 1970) lub jako krotka czasu. [34]

**Keras** to wysokopoziomowe API (Interfejs programowania aplikacji) do głębokiego uczenia, opracowane przez Google w celu implementacji sieci neuronowych. Jest napisany w Pythonie i służy do łatwej implementacji sieci neuronowych. Obsługuje również wiele obliczeń sieci neuronowych backend. Jest stosunkowo łatwy do nauki i pracy z nim, ponieważ zapewnia frontend Pythona z wysokim poziomem abstrakcji, mając jednocześnie opcję wielu back-endów do celów obliczeniowych. To sprawia, że Keras jest wolniejszy niż inne frameworki głębokiego uczenia, ale niezwykle przyjazny dla początkujących. [46]

**PyTorch** to zoptymalizowana biblioteka tensorowa używana głównie w aplikacjach głębokiego uczenia z wykorzystaniem procesorów graficznych i CPU. Jest to open-source'owa biblioteka uczenia maszynowego dla Pythona, stworzona głównie przez zespół Facebook AI Research. Jest to jedna z powszechnie używanych bibliotek uczenia maszynowego, inne to TensorFlow i Keras. Google Search Trends, które pokazuje, że popularność biblioteki PyTorch jest stosunkowo wyższa w porównaniu do TensorFlow i Keras. PyTorch jest zbudowany w oparciu o pythona i bibliotekę torch, która wspiera obliczenia tensorów na jednostkach przetwarzania graficznego. Obecnie jest to najbardziej popularna biblioteka dla społeczności badawczej zajmującej się głębokim uczeniem i sztuczną inteligencją. [47]

**TensorFlow** jest darmową i open-source'ową biblioteką oprogramowania do uczenia maszynowego i sztucznej inteligencji. Może być wykorzystywana w wielu zadaniach, ale szczególnie skupia się na szkoleniu i inferencji głębokich sieci neuronowych. Została opracowana przez zespół Google Brain do wewnętrznego użytku Google’a w badaniach i produkcji. Początkowa wersja została wydana na licencji Apache License 2.0 w 2015 roku. Google wydało zaktualizowaną wersję TensorFlow, nazwaną TensorFlow 2.0, we wrześniu 2019 roku. Może być używany w wielu różnych językach programowania, przede wszystkim Python, a także Javascript, C++ i Java. Ta elastyczność nadaje się do szeregu zastosowań w wielu różnych sektorach. [48]

**PyGAD** jest open-source'ową, łatwą w użyciu biblioteką Pythona 3 do budowania algorytmów genetycznych i optymalizacji algorytmów uczenia maszynowego. Obsługuje Keras i PyTorch. PyGAD wspiera różne typy krzyżowania, mutacji i wyboru rodzica. Pozwala na optymalizację różnych typów problemów przy użyciu algorytmu genetycznego poprzez dostosowanie funkcji fitness. Biblioteka jest aktywnie rozwijana i kolejne funkcje są dodawane regularnie. Biblioteka jest rozwijany w Pythonie 3.7.3 i zależy od NumPy do tworzenia i manipulowania tablicami oraz Matplotlib do tworzenia wykresów. Dokładna wersje używane w bibliotece to NumPy 1.16.4 i Matplotlib 3.1.0. Następny rysunek przedstawia różne etapy cyklu życia instancji klasy pygad.GA. Warto mieć na uwadze, że PyGAD zatrzymuje się, gdy wszystkie generacje zostaną zakończone, albo gdy funkcja przekazana do parametru on\_generation() zwróci komendę stop. [44]

****

Rys. 4.2.1 Schemat blokowy przedstawiający cykl życia programu. [44]

* + 1. **Klasa pygad.GA**

Pierwszy moduł dostępny w PyGAD nosi nazwę pygad i zawiera klasę o nazwie GA służącą do budowy algorytmu genetycznego. Konstruktor, metody, funkcje i atrybuty wewnątrz klasy są omówione poniżej. [45]

## **\_\_init\_\_()**

W celu utworzenia instancji klasy pygad.GA, konstruktor przyjmuje kilka parametrów, które pozwalają użytkownikowi na dostosowanie algorytmu genetycznego do różnych typów aplikacji. Konstruktor klasy pygad.GA obsługuje następujące parametry:

* num\_generations: Liczba pokoleń.
* pop\_size: Rozmiar populacji.
* best\_solutions\_fitness: Lista zawierająca wartości fitness najlepszych rozwiązań dla wszystkich pokoleń.
* best\_solution\_generation: Numer pokolenia, w którym osiągnięta została najlepsza wartość fitness. Numer pokolenia jest przypisywany dopiero po zakończeniu działania metody run(). W przeciwnym razie jego wartość wynosi -1.
* best\_solutions: Tablica NumPy przechowująca najlepsze rozwiązanie dla każdego pokolenia. Istnieje tylko wtedy, gdy parametr save\_best\_solutions w konstruktorze klasy pygad.GA jest ustawiony na True.
* last\_generation\_fitness: Wartości fitness rozwiązań w ostatnim pokoleniu.
* num\_parents\_mating: Liczba rozwiązań, które mają być wybrane jako rodzice.
* fitness\_func: Akceptuje funkcję, która musi przyjąć 2 parametry (pojedyncze rozwiązanie i jego indeks w populacji) i zwrócić wartość fitness rozwiązania.
* initial\_population: Populacja początkowa zdefiniowana przez użytkownika. Jest to przydatne, gdy użytkownik chce rozpocząć generacje z niestandardową populacją początkową. Domyślnie ustawienie to None, co oznacza, że użytkownik nie podał żadnej populacji początkowej. W tym przypadku, PyGAD tworzy populację początkową używając parametrów sol\_per\_pop i num\_genes.
* sol\_per\_pop: Liczba rozwiązań (chromosomów) w populacji. Ten parametr nie działa, jeśli istnieje parametr initial\_population.
* num\_genes: Liczba genów w rozwiązaniu/chromosomie. Ten parametr nie jest potrzebny, jeśli użytkownik poda populację początkową w parametrze initial\_population.
* gene\_type=float: Określa typ genu. Może być przypisany do pojedynczego typu danych, który jest stosowany do wszystkich genów lub może określać typ danych każdego genu z osobna. Domyślnie jest to float (liczby zmiennoprze cinkowe), co oznacza, że wszystkie geny mają typ danych float.
* init\_range\_low=-4: Dolna wartość losowego zakresu, z którego wybierane są wartości genów w populacji początkowej, domyślnie przyjmuje wartość -4.
* init\_range\_high=4: Górna wartość losowego zakresu, z którego wybierane są wartości genów w populacji początkowej, domyślnie przyjmuje wartość 4.
* parent\_selection\_type=”sss”: Typ selekcji rodziców, obsługiwane typy to: „sss” (steady-state selection, wybór ustalony), „rws” (ang. roulette wheel selection, wybór koła ruletki), „sus” (ang. stochastic universal selection, stochastyczny wybór uniwersalny), „rank” (ang. rank selection, wybór rankingowy), „random” (ang. random selection, wybór losowy) i „tournament” (ang. tournament selection, wybór turniejowy).
* keep\_parents=-1: Liczba rodziców do zachowania w bieżącej populacji. -1 (domyślnie) oznacza zachowanie wszystkich rodziców w następnej populacji. 0 oznacza nie zachowywanie żadnych rodziców w następnej populacji. Wartość większa niż 0 oznacza zachowanie określonej liczby rodziców w następnej populacji. Należy pamiętać, że wartość przypisana do keep\_parents nie może być mniejsza od - 1 lub większa od liczby rozwiązań w populacji sol\_per\_pop.
* K\_tournament=3: W przypadku, gdy typem wyboru rodziców jest selekcja turniejowa, K\_tournament określa liczbę rodziców biorących udział w wyborze turniejowym. Domyślnie jest to 3.
* crossover\_type=”single\_point”: Typ operacji krzyżowania, obsługiwane typy to „single\_point” (ang. single-point crossover, krzyżowanie jednopunktowe), „two\_points” (ang. two points crossover, krzyżowanie dwupunktowe), „uniform” (ang. uniform crossover, krzyżowanie równomierne) oraz „scattered” (ang. scattered crossover, krzyżowanie rozproszone). Domyślnie jest to „single\_point”.
* crossover\_probability=None: Prawdopodobieństwo wyboru rodzica do zastosowania operacji krzyżowania. Jego wartość musi zawierać się w przedziale od 0.0 do 1.0 włącznie. Dla każdego rodzica generowana jest losowa wartość z przedziału od 0.0 do 1.0. Jeśli ta losowa wartość jest mniejsza lub równa wartości przypisanej do parametru crossover\_probability, to rodzic ten jest wybierany.
* mutation\_type=”random”: Typ operacji mutacji, obsługiwane typy to „random” (ang. random mutation, mutacja losowa), „swap” (ang. swap mutation, mutacja zamiany miejsc), „inversion” (ang. inversion mutation, mutacja inwersyjna), „scramble” (ang. scramble mutation, mutacja mieszania) oraz „adaptive” (ang. adaptive mutation, mutacja adaptacyjna). Domyślnie jest to „random”.
* mutation\_probability=None: Prawdopodobieństwo wyboru genu do zastosowania operacji mutacji. Jego wartość musi zawierać się w przedziale od 0.0 do 1.0 włącznie. Dla każdego genu w rozwiązaniu generowana jest losowa wartość z przedziału od 0.0 do 1.0. Jeśli ta losowa wartość jest mniejsza lub równa wartości przypisanej do parametru mutation\_probability, to gen jest wybierany. Jeśli ten parametr istnieje, to nie ma potrzeby stosowania 2 parametrów: mutation\_percent\_genes i mutation\_num\_genes.
* mutation\_by\_replacement=False: Opcjonalny parametr logiczny (boolean). Działa tylko wtedy, gdy wybrany typ mutacji jest losowy (mutation\_type="random"). W tym przypadku, mutation\_by\_replacement=True oznacza zastąpienie genu przez losowo wygenerowaną wartość. Jeśli wartość parametru wynosi False, to nie ma to żadnego efektu i mutacja losowa działa poprzez dodanie losowej wartości do genu.
* mutation\_percent\_genes=”default”: Procent genów do zmutowania. Domyślnie jest to "default", który później jest tłumaczony na liczbę całkowitą 10, co oznacza, że 10% genów zostanie zmutowanych. Musi być z przedziału większego od 0 i mniejszego, bądź równego 100. Z tego procentu wyliczana jest liczba genów do zmutowania, która jest przypisywana do parametru mutation\_num\_genes. Parametr mutation\_percent\_genes nie ma żadnego działania, jeśli istnieją parametry mutation\_probability lub mutation\_num\_genes.
* mutation\_num\_genes=None: Liczba genów do zmutowania, która domyślnie jest równa None, co oznacza, że nie podano żadnej liczby. Parametr mutation\_num\_genes nie ma działania, jeśli istnieje parametr mutation\_probability.
* random\_mutation\_min\_val=-1.0: Dla mutacji losowej, parametr random\_mutation\_min\_val określa wartość dolną zakresu, z którego wybierana jest wartość losowa dodawana do genu. Domyślnie jest to -1.
* random\_mutation\_max\_val=1.0: Dla mutacji losowej, parametr random\_mutation\_max\_val określa wartość górną zakresu, z którego wybierana jest wartość losowa dodawana do genu. Domyślnie jest to 1.
* gene\_space=None: Służy do określenia możliwych wartości dla każdego genu w przypadku, gdy użytkownik chce ograniczyć wartości genów. Jest to przydatne, jeśli przestrzeń genów jest ograniczona do pewnego zakresu lub do wartości dyskretnych. Akceptuje listę, krotkę lub zakres. Gdy wszystkie geny mają tę samą globalną przestrzeń, należy określić ich wartości jako list/tuple/range. Na przykład, gene\_space = [0.3, 5.2, -4, 8] ogranicza wartości genów do 4 podanych wartości. Jeśli każdy gen ma swoją własną przestrzeń, to parametr gene\_space może być zagnieżdżony jak [[0.4, -5], [0.5, -3.2, 8.2, -9], ...] gdzie pierwsza podlista określa wartości dla pierwszego genu, druga podlista dla drugiego genu itd. Jeśli zagnieżdżona lista lub krotka ma wartość None, to wartość początkowa genu jest wybierana losowo z zakresu określonego przez parametry init\_range\_low i init\_range\_high, a wartość mutacji jest wybierana losowo z zakresu określonego przez parametry random\_mutation\_min\_val i random\_mutation\_max\_val.
* on\_start=None: Akceptuje funkcję, która ma być wywołana tylko raz, zanim algorytm genetyczny rozpocznie obliczenia. Funkcja ta musi przyjmować jeden parametr reprezentujący instancję algorytmu genetycznego.
* on\_fitness=None: Akceptuje funkcję, która ma zostać wywołana po obliczeniu wartości fitness wszystkich rozwiązań w populacji. Funkcja ta musi przyjmować 2 parametry: pierwszy reprezentuje instancję algorytmu genetycznego, a drugi jest listą wartości fitness wszystkich rozwiązań.
* on\_parents=None: Akceptuje funkcję, która ma być wywołana po wybraniu rodziców. Funkcja ta musi przyjmować 2 parametry: pierwszy reprezentuje instancję algorytmu genetycznego, a drugi wybranych rodziców.
* on\_crossover=None: Akceptuje funkcję, która ma być wywoływana za każdym razem, gdy zastosowana zostanie operacja krzyżowania. Funkcja ta musi przyjmować 2 parametry: pierwszy z nich reprezentuje instancję algorytmu genetycznego, a drugi reprezentuje potomstwo wygenerowane za pomocą krzyżowania.
* on\_mutation=None: Akceptuje funkcję, która ma być wywoływana za każdym razem, gdy zastosowana zostanie operacja mutacji. Funkcja ta musi przyjmować 2 parametry: pierwszy z nich reprezentuje instancję algorytmu genetycznego, a drugi reprezentuje potomstwo po zastosowaniu mutacji.
* on\_generation=None: Akceptuje funkcję, która ma być wywoływana po każdym pokoleniu. Funkcja ta musi przyjąć jeden parametr reprezentujący instancję algorytmu genetycznego. Jeśli funkcja zwróciła stop, to metoda run() zatrzymuje się bez ukończenia pozostałych generacji.
* on\_stop=None: Akceptuje funkcję, która ma być wywołana tylko raz, dokładnie przed zatrzymaniem algorytmu genetycznego lub po zakończeniu wszystkich pokoleń. Funkcja ta musi przyjmować 2 parametry: pierwszy reprezentuje instancję algorytmu genetycznego, a drugi jest listą wartości fitness rozwiązań ostatniej populacji.
* delay\_after\_gen=0.0: Przyjmuje nieujemną liczbę określającą czas w sekundach, jaki należy odczekać po zakończeniu generacji przed przejściem do następnej generacji. Domyślnie przyjmuje wartość 0.0, co oznacza brak opóźnienia po zakończeniu generacji.
* save\_best\_solutions=False: Jeżeli parametr wynosi True, wtedy najlepsze rozwiązanie po każdym pokoleniu jest zapisywane do atrybutu best\_solutions. Jeśli False (domyślnie), to żadne rozwiązania nie są zapisywane i atrybut best\_solutions będzie pusty.
* save\_solutions=False: Jeśli True, to wszystkie rozwiązania w każdym pokoleniu są dołączane do atrybutu solutions, który jest tablicą NumPy.
* suppress\_warnings=False: Parametr logiczny (boolean), który kontroluje czy komunikaty ostrzegawcze są drukowane czy nie. Domyślną wartością jest False.
* allow\_duplicate\_genes=True: Jeśli True, to rozwiązanie/chromosom może mieć zduplikowane wartości genów. Jeśli False, to każdy gen będzie miał unikalną wartość w swoim rozwiązaniu.
* stop\_criteria=None: Pewne kryteria zatrzymania ewolucji. Każde kryterium jest przekazywane jako wartość tekstowa (string), który wywołuje komendę stop. Obecnie 2 obsługiwane słowa to reach i saturate. reach zatrzymuje metodę run() jeśli wartość fitness jest równa lub większa od podanej wartości fitness. Przykładem jest "reach\_40", które zatrzymuje ewolucję jeśli fitness jest >= 40. saturate oznacza zatrzymanie ewolucji jeśli fitness ustabilizuje się na danej wartości fitness przez daną liczbę kolejnych pokoleń. Przykładem saturate jest "saturate\_7", co oznacza zatrzymanie metody run(), jeśli fitness nie zmieni się przez 7 kolejnych pokoleń.
  + 1. **Metody generowania wykresów w klasie pygad.GA**
* plot\_fitness(): Pokazuje, jak wartość fitness ewoluuje z pokolenia na pokolenie.
* plot\_genes(): Pokazuje, jak zmienia się wartość genu dla każdego pokolenia.
* plot\_new\_solution\_rate(): Pokazuje liczbę nowych rozwiązań odkrytych w każdym elemencie.
  + 1. **Główne metody w klasie pygad.GA**

**initialize\_population()**

Tworzy początkową populację losowo, jako tablicę NumPy. Tablica jest zapisywana w atrybucie instancji o nazwie population. Przyjmuje następujące parametry:

* low: Dolna wartość losowego zakresu, z którego wybierane są wartości genów w początkowej populacji. Domyślnie przyjmuje wartość -4.
* high: Górna wartość zakresu losowego, z którego wybierane są wartości genów w populacji początkowej. Domyślnie 4.

Metoda ta przyporządkowuje wartości następujących trzech atrybutów instancji:

* pop\_size
* population
* initial\_population

**cal\_pop\_fitness()**

Oblicza wartości fitness wszystkich rozwiązań w bieżącej populacji. Działa poprzez iterację po rozwiązaniach i wywołanie funkcji przypisanej do parametru fitness\_func w konstruktorze klasy pygad.GA dla każdego rozwiązania. Zwraca tablicę wartości fitness wszystkich rozwiązań.

**run()**

Uruchamia algorytm genetyczny. Jest to główna metoda, w której algorytm genetyczny jest ewoluowany przez kilka pokoleń. Nie przyjmuje żadnych parametrów, ponieważ wykorzystuje instancję do uzyskania dostępu do wszystkich swoich wymogów. Wartości fitness wszystkich rozwiązań w populacji są obliczane zgodnie z metodą cal\_pop\_fitness(), która wewnętrznie wywołuje funkcję przypisaną do parametru fitness\_func w konstruktorze klasy pygad.GA dla każdego rozwiązania. Na podstawie wartości fitness wszystkich rozwiązań wybierani są rodzice za pomocą metody select\_parents(). Zachowanie tej metody jest określane na podstawie typu wyboru rodziców w parametrze parent\_selection\_type w konstruktorze klasy pygad.GA. Na podstawie wybranych rodziców generowane jest potomstwo poprzez zastosowanie operacji krzyżowania i mutacji przy użyciu metod crossover() i mutation(). Zachowanie tych dwóch metod jest definiowane zgodnie z parametrami crossover\_type i mutation\_type w konstruktorze klasy pygad.GA. Po zakończeniu generowania zachodzi następujący proces:

* Atrybut population jest aktualizowany o nową populację.
* Atrybut generations\_completed jest przypisywany przez numer ostatniego zakończonego pokolenia.
* Jeśli istnieje funkcja wywołania zwrotnego przypisana do atrybutu callback\_generation, to zostanie ona wywołana.

Po zakończeniu działania metody run() zachodzi następująca sytuacja:

* best\_solution\_generation przypisany jest numer generacji, w której osiągnięto najlepszą wartość fitness.
* Atrybut run\_completed jest ustawiony na True.

**4.3 Programowanie zorientowane obiektowo**

Cały program został napisany zgodnie z paradygmatem programowania zorientowanego obiektowo (object-oriented programming, w skrócie „OOP”) w celu uniwersalności i możliwości powtórnego odtwarzania kodu wraz z modyfikowalnymi zmiennymi wejściowymi, zapewni to możliwość rozwiązywania dowolnie złożonych problemów opartych na liczbach. Programowanie zorientowane obiektowo (OOP) to model programowania komputerowego, który organizuje projektowanie oprogramowania wokół danych lub obiektów, a nie funkcji i logiki. Obiekt może być zdefiniowany jako pole danych, które ma unikalne atrybuty i zachowanie. OOP koncentruje się na obiektach, którymi programiści chcą manipulować, a nie na logice wymaganej do manipulowania nimi. To podejście do programowania jest dobrze przystosowane do programów, które są duże, złożone i aktywnie aktualizowane lub utrzymywane. Obejmuje to programy do produkcji i projektowania, a także aplikacje mobilne; na przykład, OOP może być używany do oprogramowania symulacyjnego systemu produkcyjnego. Organizacja programu zorientowanego obiektowo sprawia, że metoda ta jest również korzystna dla rozwoju zespołowego, w którym projekty są podzielone na grupy. Dodatkowe korzyści płynące z OOP to możliwość ponownego użycia kodu, skalowalność i wydajność. Pierwszym krokiem w OOP jest zebranie wszystkich obiektów, którymi programista chce manipulować i określenie jak się one ze sobą wiążą - jest to ćwiczenie znane jako modelowanie danych. Kiedy obiekt jest już znany, jest on oznaczany jako klasa obiektów, która definiuje rodzaj danych jakie zawiera oraz wszelkie sekwencje logiczne, które mogą nim manipulować. Każda odrębna sekwencja logiczna jest znana jako metoda. Obiekty mogą komunikować się za pomocą dobrze zdefiniowanych interfejsów zwanych komunikatami. Strukturę lub bloki konstrukcyjne programowania zorientowanego obiektowo obejmują następujące elementy:

* **Klasy** są zdefiniowanymi przez użytkownika typami danych, które działają jako schemat dla poszczególnych obiektów, atrybutów i metod.
* **Obiekty** są instancjami klasy utworzonymi z konkretnie zdefiniowanych danych. Obiekty mogą odpowiadać obiektom świata rzeczywistego lub abstrakcyjnym bytom. Kiedy klasa jest definiowana na początku, opis jest jedynym obiektem, który jest zdefiniowany.
* **Metody** są funkcjami zdefiniowanymi wewnątrz klasy, które opisują zachowanie obiektu. Każda metoda zawarta w definicjach klas zaczyna się od referencji do obiektu instancji. Dodatkowo, podprogramy zawarte w obiekcie nazywane są metodami instancji. Programiści używają metod do ponownego użycia lub utrzymania funkcjonalności zamkniętej wewnątrz jednego obiektu na raz.
* **Atrybuty** są zdefiniowane w szablonie klasy i reprezentują stan obiektu. Obiekty będą posiadały dane przechowywane w polu atrybutów. Atrybuty klasy należą do samej klasy.

Programowanie zorientowane obiektowo opiera się na następujących zasadach:

* **Enkapsulacja**. Zasada ta mówi, że wszystkie ważne informacje są zawarte wewnątrz obiektu i tylko wybrane informacje są ujawniane. Implementacja i stan każdego obiektu są prywatnie przechowywane wewnątrz zdefiniowanej klasy. Inne obiekty nie mają dostępu do tej klasy ani uprawnień do wprowadzania zmian. Mogą one jedynie wywoływać listę publicznych funkcji lub metod. Ta cecha ukrywania danych zapewnia większe bezpieczeństwo programu i pozwala uniknąć niezamierzonego uszkodzenia danych.
* **Abstrakcja**. Obiekty ujawniają tylko te wewnętrzne mechanizmy, które są istotne dla użycia innych obiektów, ukrywając zbędny kod implementacyjny. Klasa pochodna może mieć rozszerzoną funkcjonalność. Ta koncepcja może pomóc programistom łatwiej wprowadzać dodatkowe zmiany lub uzupełnienia w czasie.
* **Dziedziczenie**. Klasy mogą ponownie wykorzystać kod z innych klas. Relacje i podklasy pomiędzy obiektami mogą być przypisane, co pozwala programistom na ponowne wykorzystanie wspólnej logiki przy jednoczesnym zachowaniu unikalnej hierarchii. Ta właściwość OOP wymusza dokładniejszą analizę danych, skraca czas rozwoju i zapewnia wyższy poziom dokładności.
* **Polimorfizm**. Obiekty są zaprojektowane do współdzielenia zachowań i mogą przybierać więcej niż jedną formę. Program określi, które znaczenie lub zastosowanie jest niezbędne dla każdego wykonania tego obiektu z klasy nadrzędnej redukując potrzebę duplikowania kodu. Następnie tworzona jest klasa potomna, która rozszerza funkcjonalność klasy nadrzędnej. Polimorfizm pozwala różnym typom obiektów na przejście przez ten sam interfejs.

Korzyści z OOP obejmują:

* **Modularność**. Enkapsulacja umożliwia obiektom bycie samowystarczalnymi, co ułatwia rozwiązywanie problemów i rozwój współpracy.
* **Możliwość ponownego użycia**. Kod może być ponownie użyty poprzez dziedziczenie, co oznacza, że zespół nie musi pisać tego samego kodu wiele razy.
* **Produktywność**. Programiści mogą szybciej konstruować nowe programy dzięki wykorzystaniu wielu bibliotek i kodu wielokrotnego użytku.
* **Łatwość rozbudowy i skalowalność**. Programiści mogą implementować funkcjonalności systemu niezależnie od siebie.
* **Opisy interfejsów**. Opisy systemów zewnętrznych są proste, dzięki technikom przekazywania komunikatów, które są używane do komunikacji między obiektami.
* **Bezpieczeństwo**. Dzięki zastosowaniu enkapsulacji i abstrakcji, złożony kod jest ukryty, utrzymanie oprogramowania jest łatwiejsze, a protokoły internetowe są chronione.
* **Elastyczność**. Polimorfizm pozwala pojedynczej funkcji dostosować się do klasy, w której jest umieszczona. Różne obiekty mogą również przechodzić przez ten sam interfejs.

Model programowania zorientowanego obiektowo był krytykowany przez programistów z wielu powodów. Największym problemem jest to, że OOP nadmiernie podkreśla komponent danych w rozwoju oprogramowania i nie skupia się wystarczająco na obliczeniach lub algorytmach. Dodatkowo, kod OOP może być bardziej skomplikowany do napisania i dłużej się kompiluje. Metody alternatywne do OOP obejmują:

* Programowanie funkcyjne. Obejmuje ono języki takie jak Erlang i Scala, które są używane w telekomunikacji i systemach odpornych na błędy.
* Programowanie strukturalne lub modułowe. Obejmuje ono języki takie jak PHP i C#.
* Programowanie imperatywne. Ta alternatywa dla OOP skupia się raczej na funkcjach niż na modelach i obejmuje języki C++ i Java.
* Programowanie deklaratywne. Ta metoda programowania zawiera deklaracje dotyczące zadania lub pożądanego wyniku, ale nie sposobu jego osiągnięcia. Do języków tych należą Prolog i Lisp.
* Programowanie logiczne. Ta metoda, która opiera się głównie na logice formalnej i wykorzystuje języki takie jak Prolog zawiera zestaw zdań, które wyrażają fakty lub reguły dotyczące dziedziny problemu. Skupia się ona na zadaniach, które mogą skorzystać z zapytań logicznych opartych na regułach. [36]
  1. **Przykładowa implementacja algorytmu**

Aby użyć modułu pygad, poniżej znajduje się podsumowanie wymaganych kroków:

* Przygotowanie modułu fitness\_func.
* Podanie pozostałych parametrów.
* Import biblioteki pygad.
* Utworzyć instancję klasy pygad.GA.
* Uruchomienie algorytmu genetycznego.
* Wyświetlenie wyników.
* Informacja o najlepszym rozwiązaniu.
* Zapisanie wyników.

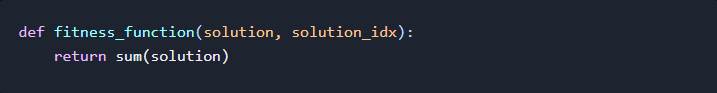
Konstruktor klasy pygad.GA ma 19 parametrów, z których 16 jest opcjonalnych. Trzy obowiązkowe parametry to:

* num\_generations
* num\_parents\_mating
* fitness\_func

Parametr fitness\_func pozwala na dostosowanie algorytmu genetycznego do różnych problemów. Parametr ten akceptuje zdefiniowaną przez użytkownika funkcję, która oblicza wartość fitness dla pojedynczego rozwiązania. Pobiera ona dwa dodatkowe parametry: rozwiązanie oraz jego indeks w populacji. Załóżmy, że istnieje populacja z trzema rozwiązaniami, jak poniżej.

Rys. 4.4.1 Listing programu przedstawiający wygenerowaną populację algorytmu.

Funkcja przypisana do parametru fitness\_func musi zwracać pojedynczą liczbę reprezentującą wartość fitness każdego rozwiązania. Poniżej znajduje się przykład, który zwraca sumę rozwiązań.

Rys. 4.4.2 Listing programu przedstawiający metodę zwracająca wartość fitness dla wylosowanych parametrów.

Wartości fitness dla trzech rozwiązań wynoszą:

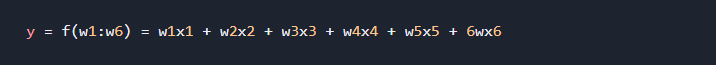
1. 776
2. 949
3. 263

Rodzice są wybierani na podstawie tych wartości fitness. Im wyższa wartość fitness, tym lepsze rozwiązanie. Po utworzeniu instancji klasy pygad.GA, kolejnym krokiem jest wywołanie metody run(), która przechodzi przez generacje ewoluujące rozwiązania.

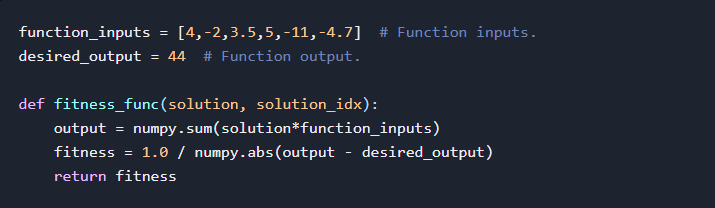
Rys. 4.4.3 Listing programu przedstawiający zaimportowanie modułu, utworzenie instancji klasy i wywołanie metody run().

To są podstawowe kroki do użycia PyGAD. Oczywiście są też dodatkowe kroki, które można wykonać, ale te są niezbędnym minimum, aby zadziałał algorytm genetyczny.

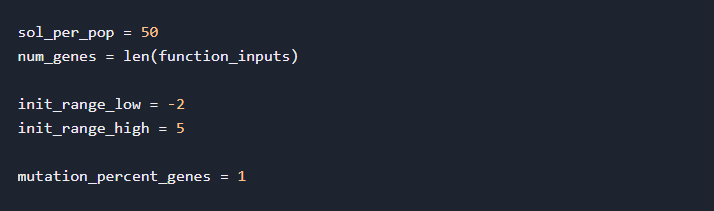
Załóżmy, że istnieje równanie z sześcioma danymi wejściowymi, jednym wyjściowym i sześcioma parametrami, jak poniżej:

Rys. 4.4.4 Listing programu przedstawiający równanie z poszukiwanymi sześcioma parametrami.

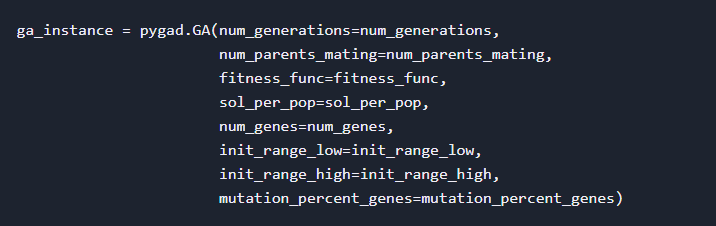
Załóżmy, że dane wejściowe to (4,-2,3.5,5,-11,-4.7), a dane wyjściowe to 44. Jakie są wartości 6 parametrów, aby spełnić to równanie? Do znalezienia odpowiedzi można wykorzystać algorytm genetyczny. Pierwszą rzeczą, którą należy zrobić jest przygotowanie funkcji fitness jak na rysunku poniżej. Oblicza ona sumę iloczynów pomiędzy każdym wejściem, a odpowiadającym mu parametrem. Ponieważ funkcja fitness musi być funkcją maksymalizującą, zwracana wartość fitness jest ułamkiem, gdzie licznik przyjmuje wartość 1, a mianownik bezwzględną wartość różnicy między pożądanym wyjściem, a sumą iloczynów. Rozwiązania z najwyższymi wartościami fitness są wybierane na rodziców.

Rys. 4.4.5 Listing programu przedstawiający zdefiniowanie początkowych parametrów i funkcji fitness.

Teraz, gdy mamy już przygotowaną funkcję fitness należy zdefiniować pozostałe parametry. Przyjąłem wielkość populacji na poziomie 50 osobników, gdzie każdy chromosom (osobnik) będzie posiadał ilość genów odpowiadającą długości listy z danymi wejściowymi. Dolny zakres przedziału liczb do generowania wartości genów wynosi -2, a górny 5. Procent genów, które następnie zostaną poddane metodzie mutowania wynosi 1.

Rys. 4.4.6 Listing programu przedstawiający zdefiniowanie wartości pozostałych parametrów algorytmu.

Należy określić pożądane parametry obowiązkowe według własnego uznania. Po przygotowaniu niezbędnych parametrów klasa pygad.GA jest inicjalizowana.

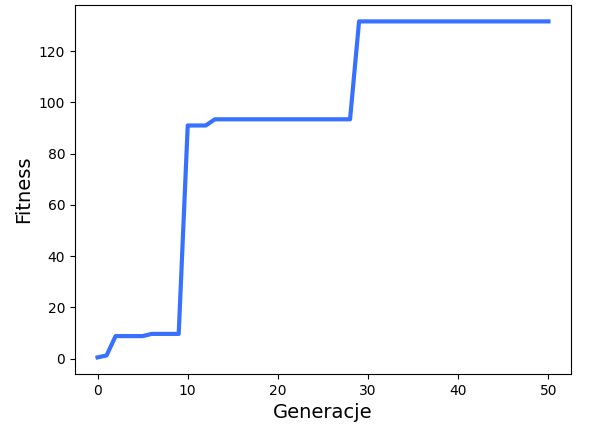
Rys. 4.4.7 Listing programu przedstawiający utworzoną instancję klasy.

Kolejnym krokiem jest wywołanie metody run(), która uruchamia generacje.

Rys. 4.4.8 Listing programu przedstawiający wywołanie metody run() utworzonej instancji.

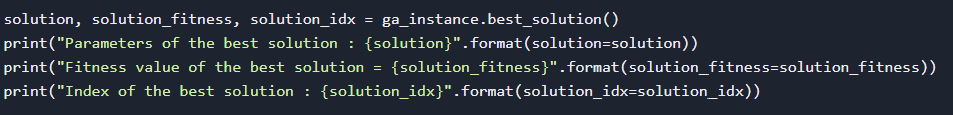
Po zakończeniu działania metody run(), metoda plot\_result() może być użyta do wyświetlenia wartości fitness na przestrzeni pokoleń.

Rys. 4.4.9 Listing programu przedstawiający wywołanie metody plot\_results().



Rys. 4.4.10 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje.

Używając metody best\_solution() możemy również dowiedzieć się jakie było najlepsze rozwiązanie, jego fitness oraz indeks w populacji.

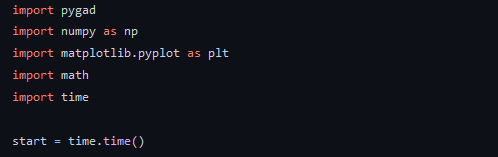
Rys. 4.4.11 Listing programu przedstawiający wyświetlenie w konsoli parametrów, wartości fitness I indeksu w populacji dla najlepszego rozwiązania.

**4.5 Algorytm genetyczny dla funkcji wielomianowej**

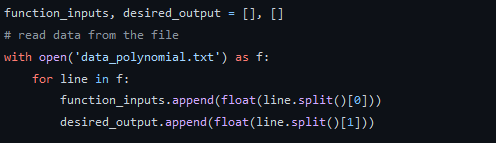
 W tym podrozdziale przedstawię bardziej dokładny opis działania programu, tym razem obiektem badań będzie funkcja wielomianowa przedstawiona na rysunku poniżej. Parametry jakie chcę otrzymać to: w1 = 4, w2 = -5 i w3 = 3.

Rys. 4.5.1 Listing programu przedstawiający poszukiwane równanie funkcji wielomianowej o trzech parametrach.

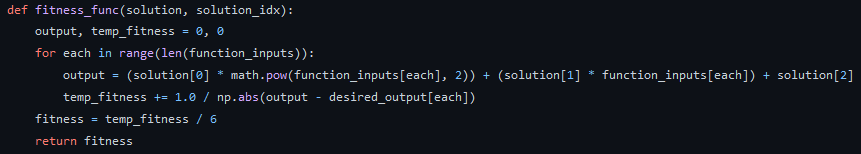
W pierwszym kroku należy zaimportować biblioteki i moduły, które zostały opisane w poprzednich rozdziałach. Dodatkowo została wykorzystana metoda time() z modułu time, który służy zmierzeniu czasu działania programu.

Rys. 4.5.2 Listing programu przedstawiający zaimportowanie bibliotek i modułów, oraz przypisanie do zmiennej czasu startu działania programu.

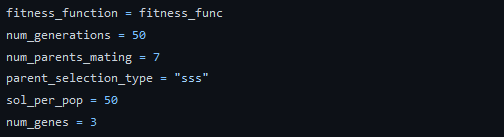
Program pobiera dane wejściowe (x) i wyjściowe (y) z pliku tekstowego i wczytuje je za pomocą metody open(), iteruje po wierszach dodając dane wejściowe przekształcone na typ zmiennoprzecinkowy za pomocą metody float() do listy function\_inputs, a dane wyjściowe do listy desired\_output. Danych jest 50 par x i y.

Rys. 4.5.3 Listing programu przedstawiający skrypt wczytujący dane z pliku tekstowego.

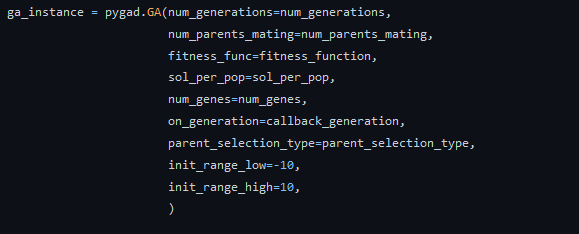
W kolejnym kroku jest liczona wartość fitness. Metoda fitness\_func(), zawiera dwa parametry: solution, czyli bieżący chromosom, którego geny są potencjalnymi rozwiązaniami i solution\_idx, czyli indeks określający miejsce pod którym znajduje się chromosom w populacji. Wewnątrz funkcji pętla for liczy wynik output dla poszczególnych danych wejściowych z wczytanych z pliku tekstowego. temp\_fitness jest zmienną do której jest dodawana wartość fitness dla poszczególnych wartości osi odciętej. W ostatnim kroku w pętli for liczona jest zmienna tymczasowa wartości fitness, jako ułamek w liczniku 1.0, a w mianowniku różnica bezwzględna wartości osi rzędnych do wartości wprowadzonych jako dane wyjściowe. Po zakończeniu pętli dla 50 par obliczamy już ostateczną wartość fitness dla pojedynczego osobnika w populacji podzieloną przez 6, co wynika z faktu, że przy dużych populacjach i generacjach zbliżających się do 1000 wartość fitness byłą w stanie urosnąć do tak dużych rozmiarów, że Python zwracał nieskończoność jako inf. Należy jeszcze zauważyć, że opisana operacji musi zostać wykonana tyle razy ile wynosi iloczyn parametrów num\_generation i sol\_per\_pop.

Rys. 4.5.4 Listing programu definiujący metodę zwracającą wartość fitness.

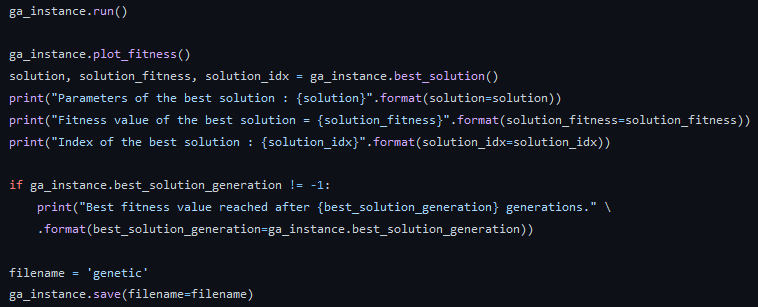
Teraz jak już została zdefiniowana metoda fitness\_func() należy ją wywołać i przypisać do zmiennej fitness\_function. Następnie trzeba przyjąć założenia co do parametrów klasy. Liczbę generacji wynosi 50, liczba rozwiązań wybranych na rodziców do puli rozrodczej została przyjęta jako 7, typ wyboru rodziców jest domyślny, czyli selekcja ustalona. Populacja wynosi 50 osobników. Liczba genów wynosi tyle ile jest poszukiwanych parametrów, czyli 3.

Rys. 4.5.5 Listing programu przedstawiający zadeklaraowane wartości parametrów klasy.

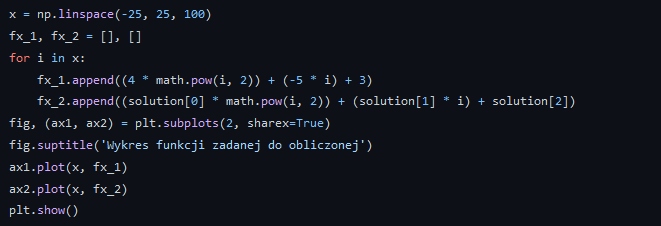
Należy przypisać zainicjowane parametry w poprzednim kroku i dodać parametry opcjonalne, nie są uznawane jako wymagane, ponieważ mają przypisaną wartość domyślną, zakres (ang. scope). Z dodatkowych parametrów został wprowadzony przedział generowania potencjalnych rozwiązań z dolną granicą -10 i górną 10. Po przygotowaniu niezbędnych parametrów klasa pygad.GA jest inicjalizowana.

Rys. 4.5.6 Listing programu przedstawiający utworzenie instancji klasy pygad dla określonuch parametrów poszukującej wartości parametrów dla najlepszego rozwiązania.

Po uruchomieniu instancji klasy pygad.GA za pomocą metody run()możemy użyć metody plot\_fitness() do wygenerowania wykresu przedstawiającego wartość fitness na przestrzeni generacji. Dodatkowo warto wyświetlić najlepsze rozwiązanie, wartość fitness dla tego rozwiązania, numer indeksu pod którym się znajduje w populacji i po której generacji została osiągnięta największa wartość fitness. Następnie opcjonalnie plik z wynikami obliczeń jest zapisywany pod nazwą „genetic” w katalogu roboczym, w którym została zdefiniowana zmienna środowiskowa.

Rys. 4.5.7 Listing programu wywołujący metodę klasy pygad run(), wczytanie i wyświetlenie najlepszego rozwiązania.

W ostatnim kroku mając już wyniki warto zwizualizować wykres funkcji podanej przez użytkownika na tle funkcji z parametrami dobranymi przez program. Funkcja do której staramy się dopasować jest koloru niebieskiego, a funkcja wygenerowana przez program koloru pomarańczowego.

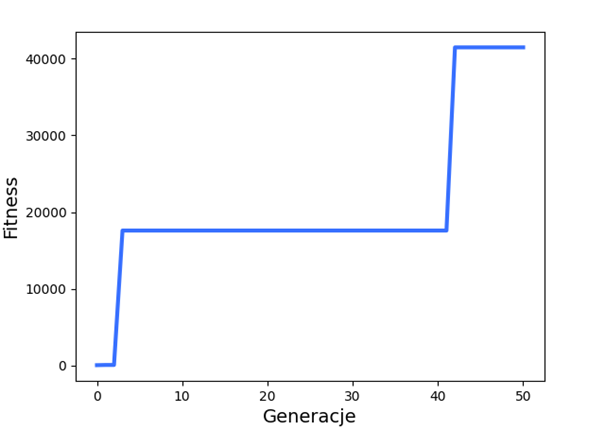
Rys. 4.5.8 Listing programu wyświetlajacy wykres przedstawiający funkcję zadaną do obliczonej przez algorytm.

Uruchomienie programu tylko raz może mieć zbyt duży wydźwięk świadczący o czystej losowości, co sprawi, że przedstawiony wynik nie będzie wiarygodny. Dlatego postanowiłem uruchomić program 10 krotnie i wyświetlić wykres funkcji fitness na tle generacji i wykres porównujący funkcję zadaną do obliczonej dla dwóch wyników, najlepszego i najgorszego. Zabieg ten ma na celu zaprezentowanie zbieżności wyników przy 10 próbach. Zależy na tym, aby zbieżność była jak największa. Wyniki zostały zaprezentowane w tabeli, w kolumnach uwzględniłem rozmiar populacji, liczbę generacji, maksymalną wartość funkcji fitness, czas działania programu, numer generacji w której została znaleziona wartość fitness dla najlepszego rozwiązania i parametry rozwiązania prezentujące się jako lista [w1, w2, w3].

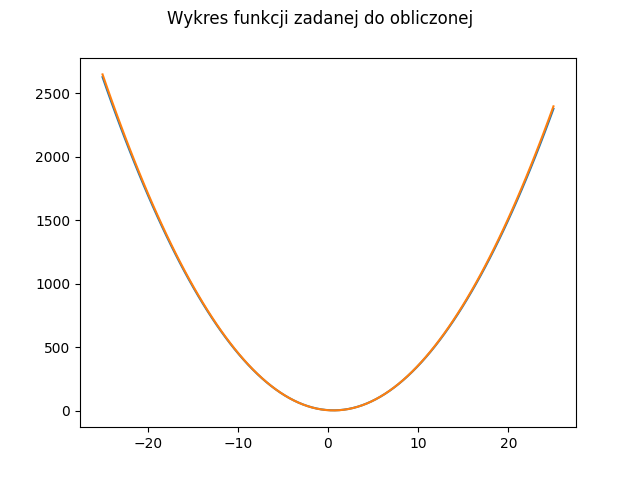
Tak prezentują się dane wyjściowe programu:

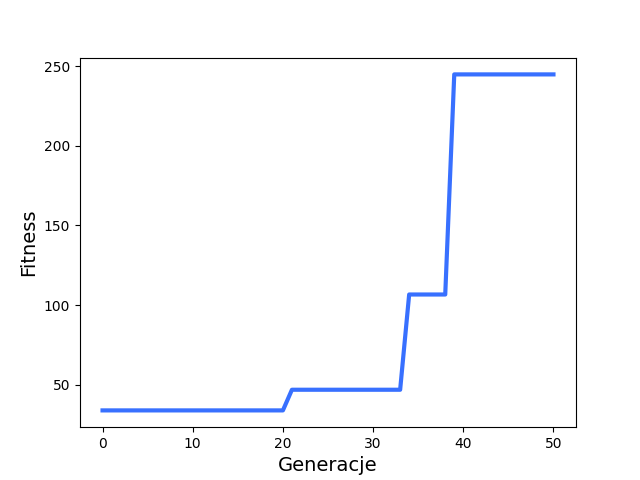
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Epoka | rozmiar populacji | liczba generacji | wartość fitness | czas [s] | nr. generacji z max fitness | rozwiązanie | wartość błędu funkcji fitness [%] |
| 1 | 50 | 50 | 41472.03 | 2.55 | 42 | [ 4.03, -5.04, 2.33] | 0.87 |
| 2 | 50 | 50 | 409.08 | 2.77 | 13 | [ 2.97, -0.88, 2.99] | 22.73 |
| 3 | 50 | 50 | 4960.52 | 2.67 | 44 | [ 2.38, -1.77, 3.00] | 39.09 |
| 4 | 50 | 50 | 4311.69 | 2.49 | 49 | [ 4.15, -5.31, 3.01] | 3.50 |
| 5 | 50 | 50 | 1430.51 | 2.91 | 36 | [ 4.09, -4.92, 2.99] | 2.27 |
| 6 | 50 | 50 | 2032.22 | 2.58 | 44 | [1.30, 0.41, 3.00] | 65.37 |
| 7 | 50 | 50 | 299.47 | 2.87 | 11 | [-1.48, 0.45, 3.03] | 61.74 |
| 8 | 50 | 50 | 365.51 | 2.81 | 44 | [4.23, -4.76, 2.52] | 5.99 |
| 9 | 50 | 50 | 200.39 | 2.51 | 46 | [2.06, -1.13, 2.99] | 46.84 |
| 10 | 50 | 50 | 244.82 | 2.62 | 39 | [-0.97, -0.02, 2.99] | 74.67 |
|  |  |  | **średnia** | 2.68 | 37 | **mediana** | 30.91 |

Tab. 4.5.1 Wyniki programu dla parametrów funkcji wielomianowej: rozmiar populacji i liczba generacji przyjęto 50, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców jest domyślny, czyli selekcja ustalona.

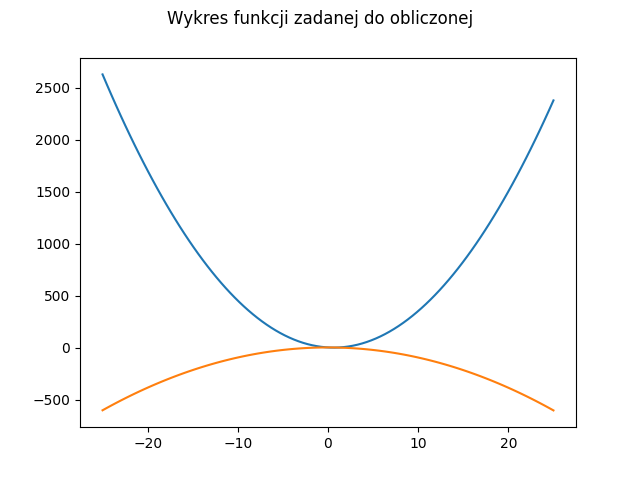


Rys. 4.5.9 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 0.87%.

Rys. 4.5.10 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 0.87%.



Rys. 4.5.11 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie 74.67%

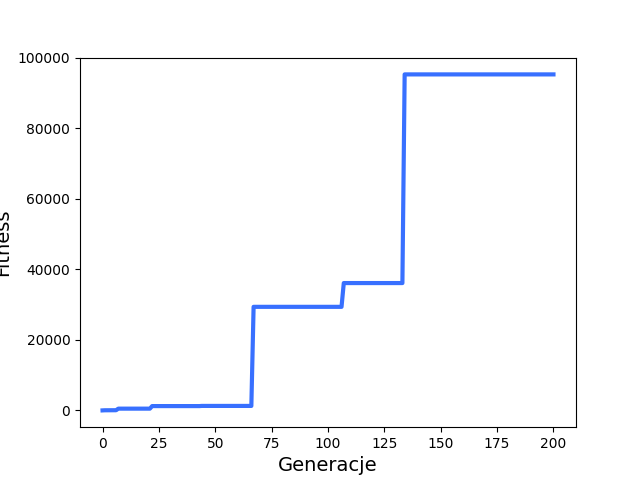
Rys. 4.5.12 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 74.67%.

Tak, jak widać w wynikach zamieszczonych w postaci tabeli (tab. 4.5.1) i wykresów (rys. 4.5.9 - 12) rozwiązanie jest dalekie od satysfakcjonującego. W dziesięciu kolejnych uruchomieniach programu otrzymana mediana wartości błędu funkcji fitness wynosi aż 30.91 %, z kolei rozrzut wyników też jest bardzo duży, najlepszy osiągnięty wynik był wręcz idealny, wyniósł 0.87 %, aczkolwiek jest to wartość stricte losowa, w tym przypadku świetnym określeniem tego zjawiska jest anomalia. Najgorsze rozwiązanie miało 74,67 %, co jak widać na wykresie (rys. 4.5.23) jest zupełnie rozbieżnym rozwiązaniem posiadającym jedynie wspólny punkt w 0. Czas działania programu osiągnął zaledwie 2.68 s, co wynika z faktu względnie małych wartości parametrów jak liczba generacji i wielkość populacji, które w tym przypadku miały przypisaną wartość 50. Warto też zwrócić uwagę na bardzo ważną wskazówkę, którą warto brać pod uwagę przy definiowaniu liczby generacji, w której generacji zostało znalezione najlepsze rozwiązanie, obserwacja tego parametru pomoże oszacować optymalną liczbę generacji, co znacząco przyczyni się do zmniejszenia czasu obliczeniowego algorytmu. Jak widać w tym przypadku nie widzimy jasnej wskazówki, ze względu ( jak możemy przypuszczać po wielkości błędu funkcji fitness ) małej liczby generacji. Jak uważniej przyjrzymy się wynikom w tabeli, można zauważyć zastanawiające wartości fitness dla najlepszych rozwiązań w poszczególnych epokach. Dla przykładu w epoce 5 wartość fitness wynosi 1430.51 przy wartości błędu funkcji fitness wynoszącym 2.27 % co w zestawieniu z rozwiązaniem w epoce 6, gdzie wartość fitness i wartość błędu funkcji fitness wynoszą odpowiednio 2032.22 I 65.37 %, nie napawa optymizmem. Aczkolwiek na tą chwilę możemy przymknąć oko na tą niezgodność, ponieważ wartość fitness jest wartością względną, stosujemy ją jako punkt odniesienie do znalezienie najlepszego rozwiązania w danej epoce, a nie do porównywania epok między sobą, tą rolę pełni z kolei wartość błędu funkcji fitness. Przyjmujemy domyślną metodę liczenia funkcji fitness w wykorzystywanym przez nas module. W tym podrozdziale będziemy się starali zmniejszać wartość błędu modyfikując proste parametry.

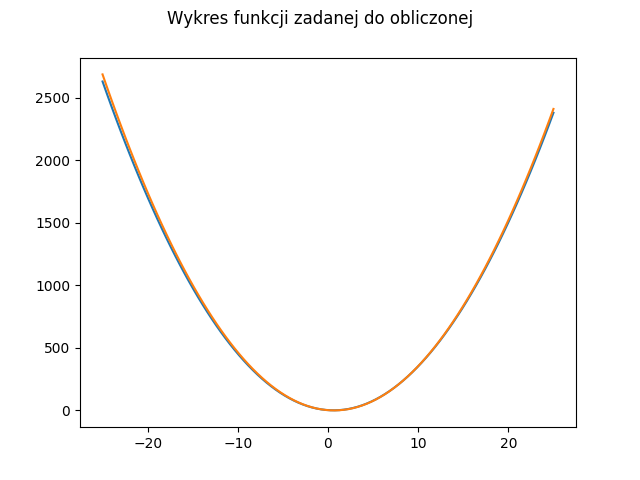
Teraz należy poczynić pewne usprawnienia w działaniu programu, aby zredukować medianę błędu towarzyszącą w naszych testach algorytmu genetycznego. W pierwszej kolejności warto spróbować zwiększyć liczbę generacji i rozmiar populacji, co na pewno przyczyni się do wydłużenia czasu pracy programu, ale również do znalezienia lepszych rozwiązań, poprzez zminimalizowanie wartości błędu funkcji fitness, również przy zwiększonej liczbie generacji będzie bardziej widoczna optymalna wartość tej zmiennej. Tak więc, nowe parametry programu i wyniki przedstawiają się następująco:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Epoka | rozmiar populacji | liczba generacji | wartość fitness | czas [s] | nr. generacji z max fitness | rozwiązanie | wartość błędu funkcji fitness [%] |
| 1 | 75 | 200 | 9573.12 | 15.45 | 31 | [-2.87, 1.86, 3.01] | 26.70 |
| 2 | 75 | 200 | 4528.75 | 16.07 | 116 | [-0.81, -0.19, 3.00] | 78.71 |
| 3 | 75 | 200 | 95259.2 | 15.21 | 134 | [4.07, -5.52, 2.63] | 1.27 |
| 4 | 75 | 200 | 4311.69 | 15.93 | 131 | [ 4.15, -5.31, 3.01] | 3.50 |
| 5 | 75 | 200 | 58094.34 | 16.67 | 98 | [4.49, -4.51, 3.00] | 11.80 |
| 6 | 75 | 200 | 2032.22 | 15.21 | 61 | [1.76, -0.51, 2.99] | 54.26 |
| 7 | 75 | 200 | 867.39 | 15.73 | 2 | [3.52, 0.10, -1.57] | 7.59 |
| 8 | 75 | 200 | 11500.9 | 15.31 | 28 | [2.90, -3.90, 3.00] | 27.78 |
| 9 | 75 | 200 | 8356.31 | 16.1 | 75 | [4.09, -4.40, 2.95] | 2.83 |
| 10 | 75 | 200 | 7991.8 | 15.91 | 23 | [0.86, -1.86, 3.00] | 79.15 |
|  |  |  | **średnia** | 15.76 | 70 | **mediana** | 19.25 |

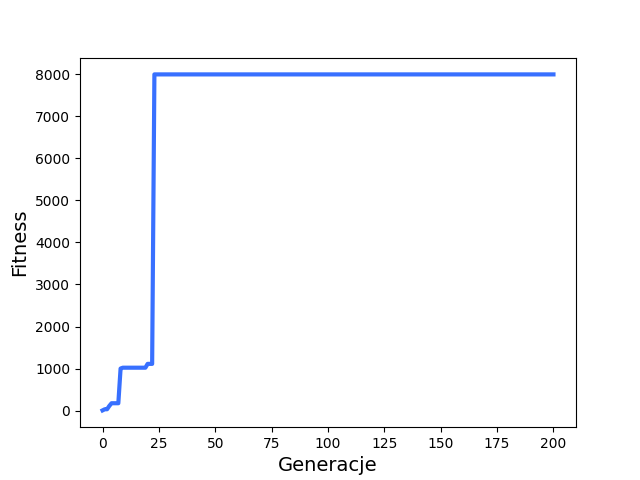
Tab. 4.5.2 Wyniki programu dla parametrów funkcji wielomianowej: rozmiar populacji wynoszący 75, liczba generacji przyjęto 200, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców jest domyślny, czyli selekcja ustalona.



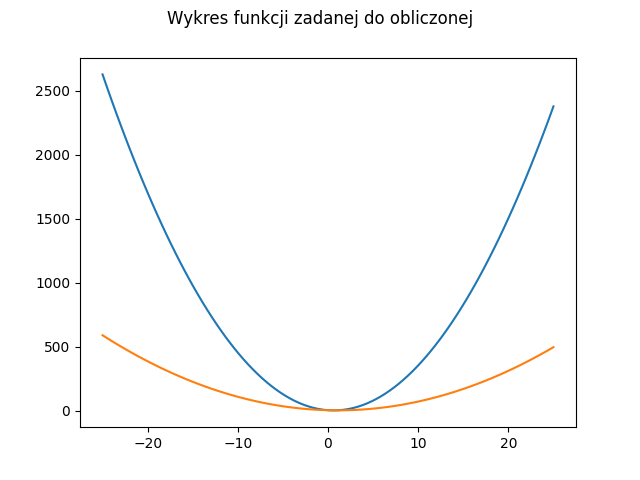
Rys. 4.5.13 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 1.27%.



Rys. 4.5.14 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 1.27%.



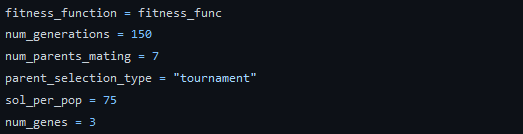
Rys. 4.5.15 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie 79.15%



Rys. 4.5.16 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 79.15%.

Po wykonaniu 10 epok otrzymane wyniki znajdują się w tabeli ( tab. 4.5.2), a wykresy prezentujące wartości fitness na tle generacji i funkcje zadane do obliczonych, znajdują się na rys. 4.5.13 – 16. Widać pewne efekty poczynionych usprawnień, między innymi w postaci redukcji błędu wartości fitness, gdzie mediana zmniejszyła się o 62.28 %, do 19.25 %. Najmniejsza wartość błędu wystąpiła w epoce 3 i wyniosłą 1.27 %, a największa w epoce 10 i wyniosłą odpowiednio 79.15 %. Jest do dosyć satysfakcjonujący wynik, aczkolwiek kosztem znaczącego wzrostu czasu działania programu, z 2.68 na 15.76, co jest wzrostem o niemal 588.10 %. Jest to bardzo duży wzrost, ale na tym etapie priorytetem jest osiągnięcie dokładności otrzymanych wyników, nawet kosztem wzrostu wydajności. Jak widać zwiększenie liczby generacji do 200 i wielkości populacji do 75, znacząco przyczyniło się do zredukowania wartości błędu funkcji fitness. Niemniej nadal mediana błędu jest wysoka i program sprawia wrażenie sporej losowości w przeprowadzonych 10 epokach. W związku z tym trzeba poczynić kolejne usprawnienia.

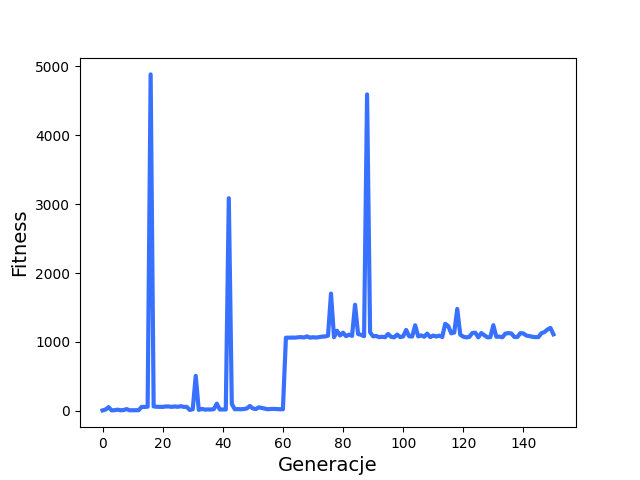
W następnej symulacji na pewno warto rozważyć zmniejszenie liczby generacji, po pierwsze wpłynie to korzystniej na czas wykonywania programu ( nie należy tego traktować jako nadrzędnego argumentu ), po drugie jak możemy w końcu zauważyć, algorytm zaczął dawać pewne znaki co do liczby generacji, najpóźniej zostało znalezione rozwiązanie w epoce 3, o numerze generacji 134, a średnia dla wszystkich epok wynosi 70. Co pozwala rozważyć zmniejszenie liczby generacji o pewną liczbę. Najsensowniejsze będzie przyjęcie liczby generacji na poziomie 150, jest to ponad dwukrotnie większa wartość w stosunku do liczby średniej numerów generacji, które osiągnęły jako pierwsze maksymalną wartość fitness z epok, jak i również dodatkowy margines bezpieczeństwa w stosunku do najpóźniejszej generacji z epoki 3. W kolejnym kroku można się przyjrzeć rodzajowi selekcji rodzica. Do tej pory wszystkie symulacje były przeprowadzane dla zmiennej przydzielonej z zakresu ( domyślnej ), czyli dla typu „sss” ( wybór ustalony ). W algorytmie genetycznym w stanie ustalonym wymienia się tylko kilku osobników na raz, w programie domyśla wartość to 7, którą pozostawimy bez zmian. Wybór ustalony zastąpimy na wybór turniejowy. Selekcja turniejowa jest metodą wyboru osobnika z populacji osobników w algorytmie genetycznym. Selekcja turniejowa polega na przeprowadzeniu kilku "turniejów" wśród kilku osobników ( chromosomów ) wybranych losowo z populacji. Zwycięzca każdego turnieju (ten z największą wartością fitness) jest wybierany do krzyżowania. Poniżej zmiany jakie wystąpiły w zdefiniowanych parametrach klasy:

Rys. 4.5.17 Listing programu przedstawiający zadeklaraowane wartości parametrów klasy.

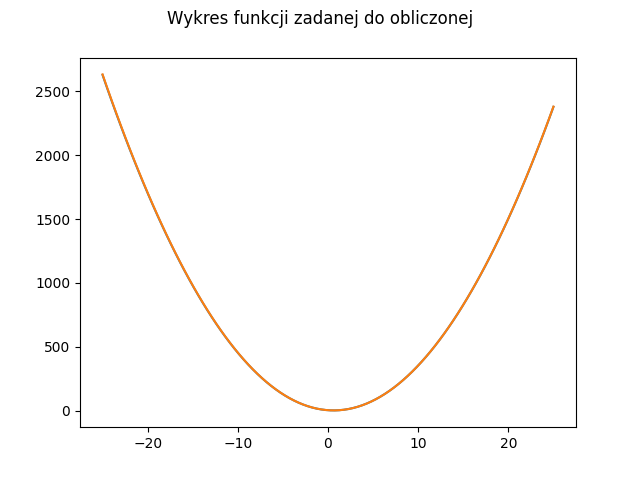
Po wykonaniu pozostałych kroków analogicznie jak w poprzednich symulacji, program zwrócił następujące wyniki:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Epoka | rozmiar populacji | liczba generacji | wartość fitness | czas [s] | nr. generacji z max fitness | rozwiązanie | wartość błędu funkcji fitness [%] |
| 1 | 75 | 150 | 2831.69 | 12.97 | 120 | [4.02, -5.20, 3.00] | 0.23 |
| 2 | 75 | 150 | 73398.33 | 14.29 | 132 | [3.79, -5.42, 3.00] | 6.00 |
| 3 | 75 | 150 | 4884.21 | 13.20 | 16 | [4.00, -5.04, 3.00] | 0.09 |
| 4 | 75 | 150 | 7324.94 | 12.82 | 122 | [ 2.90, -3.90, 3.00] | 27.79 |
| 5 | 75 | 150 | 2232.45 | 11.59 | 106 | [3.94, -5.54, 3.00] | 2.15 |
| 6 | 75 | 150 | 5231.27 | 11.14 | 85 | [4.07, -5.51, 3.00] | 1.37 |
| 7 | 75 | 150 | 28945.82 | 12.27 | 55 | [3.83, -4.93, 3.00] | 4.36 |
| 8 | 75 | 150 | 2803.43 | 11.28 | 117 | [4.09, -5.19, 3.00] | 2.24 |
| 9 | 75 | 150 | 44453.86 | 11.23 | 88 | [3.78, -5.45, 3.00] | 6.26 |
| 10 | 75 | 150 | 3010.32 | 12.37 | 68 | [4.08, -4.76, 3.00] | 2.33 |
|  |  |  | **średnia** | 12.32 | 91 | **mediana** | 2.29 |

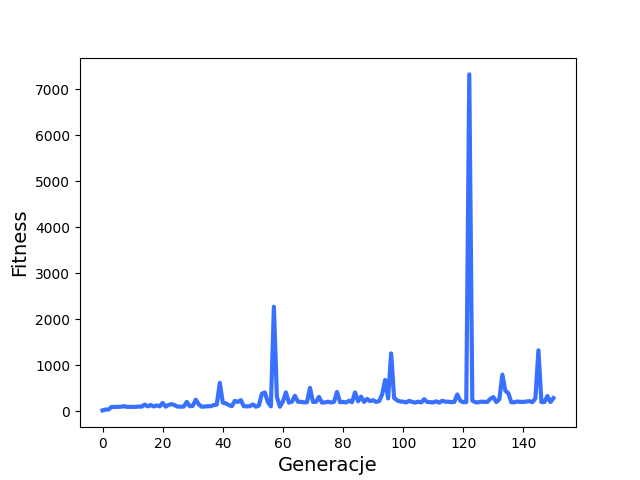
Tab. 4.5.3 Wyniki programu dla parametrów funkcji wielomianowej: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy.



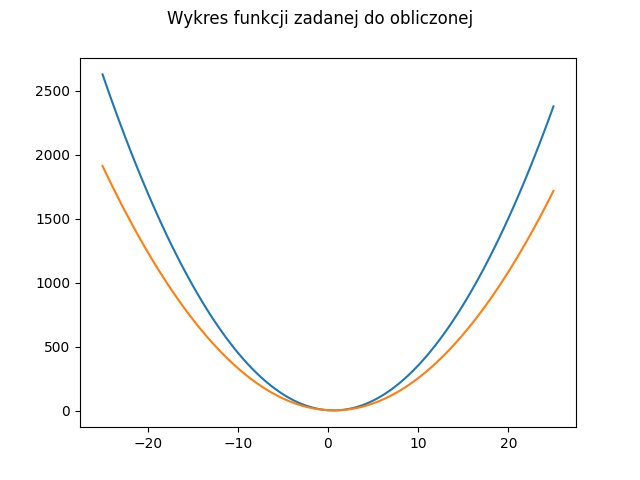
Rys. 4.5.18 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 0.09%.



Rys. 4.5.19 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 0.09%.



Rys. 4.5.20 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie 27.79%.



Rys. 4.5.21 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 27.79%.

Po zmianie typu selekcji rodziców do puli rozrodczej, z selekcji ustalonej na selekcję turniejową, udało się uzyskać świetny wynik błędu, którego mediana dla 10 epok wynosi 2.29 %. Najniższa wartość błędu wystąpiła w 3 epoce i wynosi 0.09 %, a najwyższa w epoce 4 i wynosi 27.79 %. Mediana błędu w porównaniu z pierwszą symulacją ( z wszystkimi ustawieniami algorytmu jako domyślne ) zmniejszyła się, aż o 92.59 %, z 30.91 % do 2.29 %, do czego jak najbardziej w poszczególnych krokach dążyłem. Biorąc pod uwagę, że wszystkie pozostałe wyniki nie wykraczają powyżej 6.26 %, to wartość dla 4 epoki można uznać za zwykły przypadek potwierdzający poprawność działania algorytmu. Jak też można było przypuszczać, zredukowanie liczby iteracji, nie wpłynęło na pogorszenie działania algorytmu, a dodatkowo było w stanie przyspieszyć czas wykonywania skryptu o średnio 21.83 % w stosunku do poprzedniej symulacji, której wyniki są przestawione w tabeli ( tab. 4..2).

**4.6 Algorytm genetyczny dla funkcji sinus**

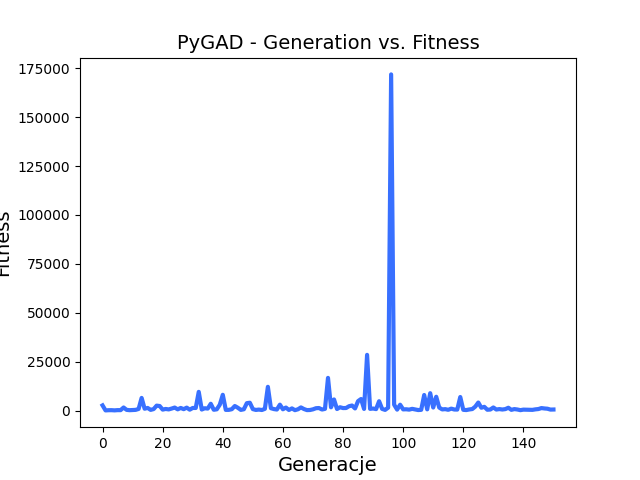
Na ten moment możemy założyć, że w kilku prostych krokach i po przeprowadzeniu 3 symulacji składających się z 10 epok udało się znaleźć „idealnie” dostrojony algorytm genetyczny. Pytanie, czy algorytm jest już na tyle udoskonalony aby można go z pełnym zaufaniem używać do innych typów funkcji? Najprawdopodobniej nie, aczkolwiek warto podejść do tego empirycznie i sprawdzić na podstawie kolejnej symulacji z ustawieniami algorytmu identycznymi do ostatniej przedstawionej symulacji. W tym przypadku przeanalizujemy funkcję sinusoidalną, zawierająca dwa parametry podawane przez użytkownika.

Rys. 4.6.1 Listing programu przedstawiający poszukiwane równanie funkcji sinus o dwóch parametrach.

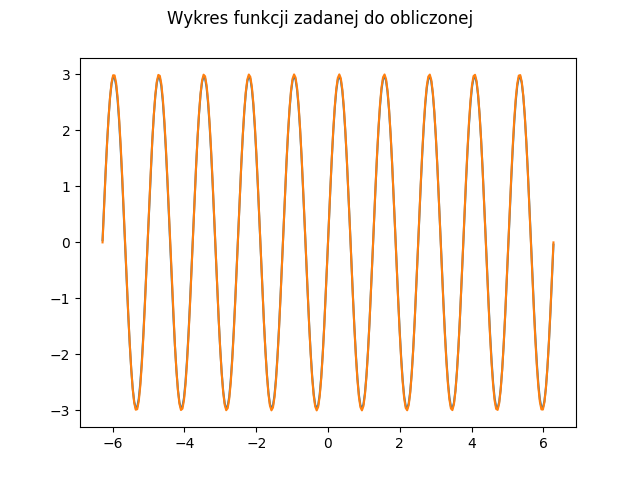
Wartości parametrów podane przez użytkownika wynoszą w1 = 3 i w2 = 5. Na podstawie tych parametrów zostały obliczone wartości x, y wczytane do programu. Wyniki symulacji są zaprezentowane w tabeli ( tab. 4.6.1 ):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Epoka | rozmiar populacji | liczba generacji | wartość fitness | czas [s] | nr. generacji z max fitness | rozwiązanie | wartość błędu funkcji fitness [%] |
| 1 | 75 | 150 | 112068.35 | 12.26 | 40 | [2.31, 4.72] | 37.65 |
| 2 | 75 | 150 | 72930.71 | 11.53 | 30 | [-3.15, -4.97] | 3.81 |
| 3 | 75 | 150 | 286011.22 | 11.65 | 114 | [-2.81, -5.35] | 96.27 |
| 4 | 75 | 150 | 64723.46 | 11.20 | 108 | [3.14, 4.50] | 6.76 |
| 5 | 75 | 150 | 73297.98 | 11.43 | 86 | [3.17, 4.54] | 4.69 |
| 6 | 75 | 150 | 171837.23 | 11.19 | 96 | [2.97, 5.00] | 1.18 |
| 7 | 75 | 150 | 591583.44 | 11.28 | 102 | [-2.71, -5.59] | 84.57 |
| 8 | 75 | 150 | 154179.24 | 11.31 | 17 | [-2.96, -5.07] | 42.48 |
| 9 | 75 | 150 | 334633.85 | 12.00 | 86 | [-2.65, -5.59] | 94.57 |
| 10 | 75 | 150 | 176467.82 | 11.41 | 35 | [2.70, 4.44] | 69.56 |
|  |  |  | **średnia** | 11.53 | 71 | **mediana** | 40.07 |

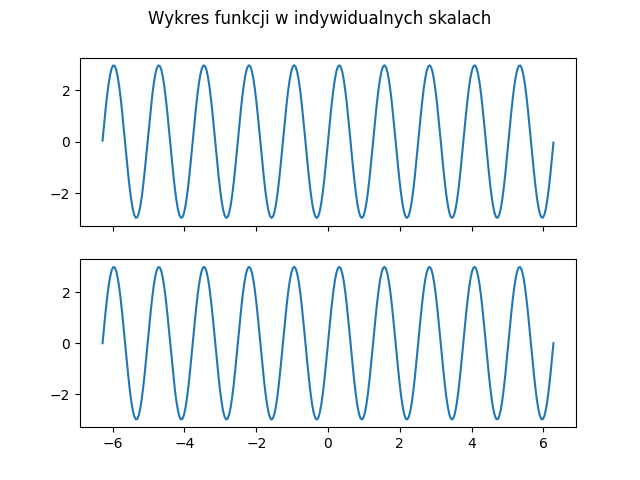
Tab. 4.6.1 Wyniki programu dla parametrów funkcji sinus: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy.



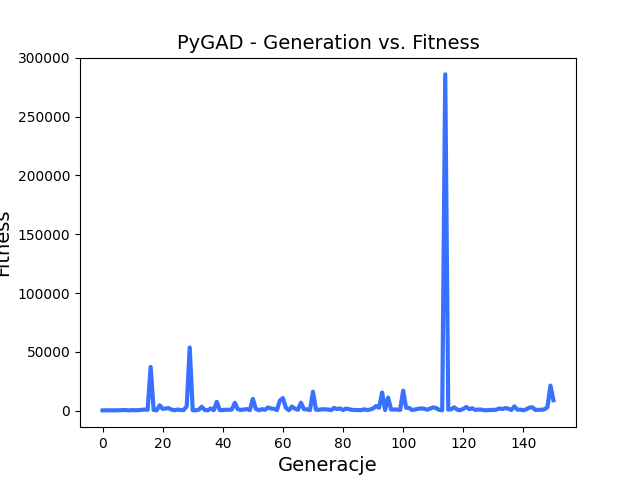
Rys. 4.6.2 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 1.18%.



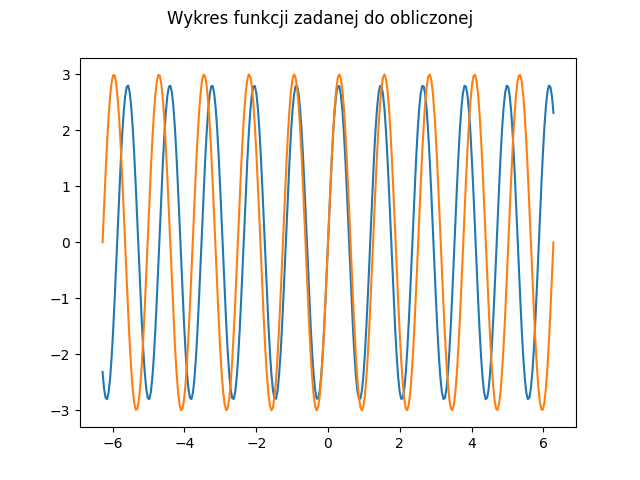
Rys. 4.6.3 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 1.18%.



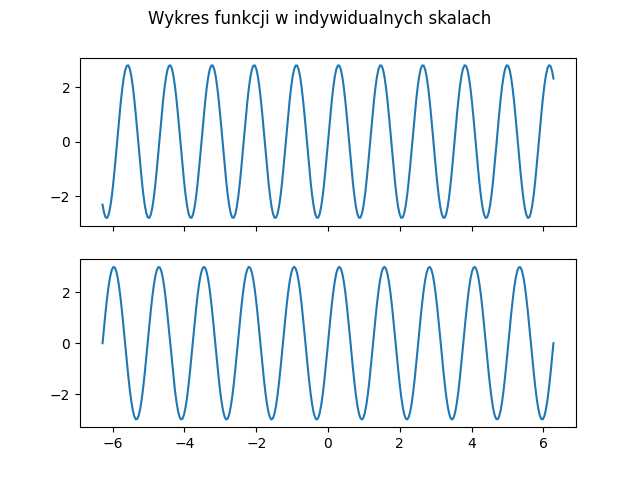
Rys. 4.6.4 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 1.18%



Rys. 4.6.5 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie wynoszącym 96.27%.



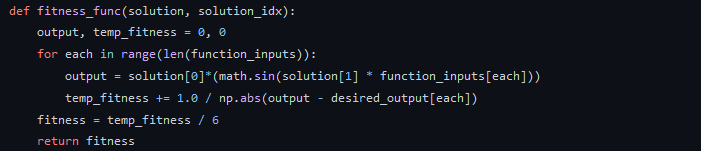
Rys. 4.6.6 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 96.27%.



Rys. 4.6.7 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 96.27%.

Po wynikach przedstawionych w tabeli (tab. 4.6.1) widać, że jeszcze jest zdecydowanie zbyt wcześnie nazywać program „uniwersalnym”. Mediana błędu jest największa jaką dotychczas udało się osiągnąć, wynosi 40.07 %. Z czego można zaobserwować duży rozrzut, największa wartość błędu wynosi, aż 96.27 %, z kolei najmniejsza 1.18 %. Średni czas wyniósł 11.53 s, co jest wartością spodziewaną, ponieważ jest bardzo zbliżona do wyniku z ostatniej symulacji programu dla funkcji wielomianowej. Po skalibrowaniu algorytmu w ten sam sposób nie można było oczekiwać jakiegokolwiek zaskoczenia w tym aspekcie. Po numerze generacji osiągającej jako pierwsza maksymalną wartość fitness można stwierdzić, że liczba 150 generacji jest jak najbardziej wystarczająca w przypadku prostych funkcji.

Warto teraz przeanalizować odwlekany problem związany z funkcją fitness, a mianowicie osiąganie wyższych wartości fitness przy większym błędzie funkcji fitness. Dlatego w kolejnym kroku należy zmodyfikować funkcję fitness, ale najpierw trzeba znaleźć przyczynę wywołujący dany problem. Na rysunku z listingiem programu zawierającym funkcje fitness jest zaprezentowany skrypt obliczający wartość fitness dla poszczególnego osobnika:

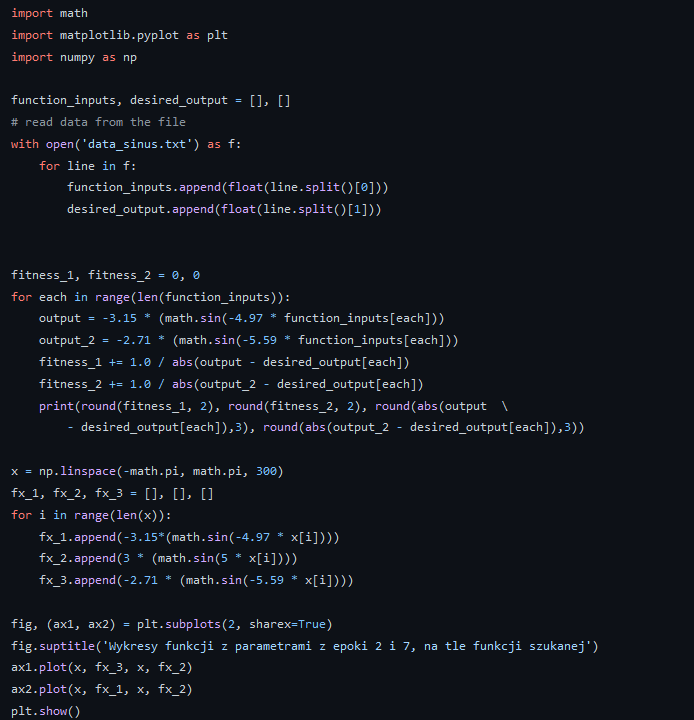
Rys. 4.6.8 Listing programu przedstawiający funkcję fitness algorytmu genetycznego w domyślnej wersji modułu PyGad.

W pierwszym kroku są zdefiniowane dwie zmienne, output i temp\_fitness, z przypisaną wartością 0. Zmienna output pełni rolę przenośnika pamięci wartość funkcji dla poszczególnych wartości osi x wczytanych przez program z pliku tekstowego podczas wykonywanej pętli. Analogiczną funkcję pełni zmienna temp\_fitness, lecz zawiera tymczasową sumę wartości fitness, liczoną jako ułamek zawierający w mianowniku wartość bezwzględną różnicy zmiennej output i poszukiwanej wartości funkcji dla poszczególnych wartości na osi x, a w liczniku 1. Jest to wartość tymczasowa, ponieważ w przedostatnim kroku jest przypisana finalna wartość fitness, która zostanie zwrócona przez program. Zwracana wartość fitness jest pomniejszona o 6, co było spowodowane szybkim uciekaniem wyniku do nieskończoności i python wyświetlał „inv”, zamiast liczby. Po przeanalizowaniu budowy samej funkcji fitness należy przyjrzeć się bliżej samemu działaniu, a w szczególności zwracanym wartościom w kolejnych generacjach. Do tego wykorzystam otrzymane rozwiązania z tabeli z poprzedniej symulacji (tab. 4.6.1), które zamieściłem w tabeli poniżej.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Epoka | rozmiar populacji | liczba generacji | wartość fitness | czas [s] | nr. generacji z max fitness | rozwiązanie | wartość błędu funkcji fitness [%] |
| 2 | 75 | 150 | 1244.23 | 11.53 | 30 | [-3.15, -4.97] | 3.81 |
| 7 | 75 | 150 | 4145.78 | 11.28 | 102 | [-2.71, -5.59] | 84.57 |

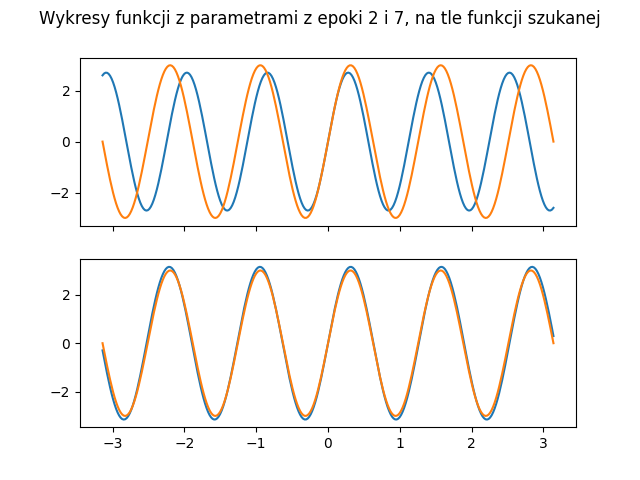
Tab. 4.6.2 Wyniki programu dla parametrów funkcji sinus: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy. Z wybranymi dwoma najbardziej skrajnymi rozwiązaniami i wartość fitness jest liczona w metodyce selekcji ustalonej, zamiast turniejowej, ze względu na trudność optymalnego przedstawienia różnic w obliczeniach funkcji fitness.

Tak jak widać w tabeli ( tab. 4.6.2 ), wybrałem dwa najbardziej skrajne wyniki pod względem wartości błędu funkcji fitness, jak i wartości fitness. Widać, że pomimo znacznie mniejszej wartości błędu w epoce 2, wartość fitness jest dużo mniejsza od rozwiązania fitness. W celu zbadania tej niepoprawności przeanalizuję poszczególne wartości funkcji dla parametrów w1, w2 z tabeli powyżej, a wartości danych x i y są wczytane z pliku tekstowego data\_sinus.txt ( łącznie 59 par danych ). Program wyświetla w pierwszej kolumnie numer iteracji, w drugiej wartość fitness dla rozwiązania z epoki 2, w trzeciej kolumnie dla epoki 7. W kolumnie czwartej jest wartość bezwzględna różnicy między rozwiązaniem dla epoki 2, a rozwiązaniem szukanym, im mniejszy wynik tym większa będzie wartość fitness. W ostatniej kolumnie znajdują się wyniki uzyskane w analogiczny sposób jak w kolumnie czwartej, czyli wartość bezwzględna różnicy między rozwiązaniem dla epoki 7, a rozwiązaniem szukanym. Pełny listing programu wykonujący opisaną wyżej operację został dodany poniżej, jak i wyniki z wykresami.

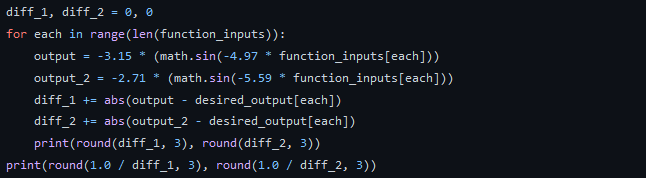
Rys. 4.6.9 Listing programu przedstawiający cały skrypt programu do analizy działania funkcji fitness.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Numer iteracji | wartość fitness dla rozwiązania z epoki 2 | wartość fitness dla rozwiązania z epoki 7 | wartość bezwzględna różnicy wartości funkcji z epoki 2 i poszukiwanej | wartość bezwzględna różnicy wartości funkcji z epoki 7 i poszukiwanej |
| 1 | 47.090 | 233.760 | 0.021 | 0.004 |
| 2 | 70.860 | 425.030 | 0.042 | 0.005 |
| 3 | 86.960 | 3944.700 | 0.062 | 0.000 |
| 4 | 99.310 | 4011.300 | 0.081 | 0.015 |
| 5 | 109.480 | 4035.590 | 0.098 | 0.041 |
| 6 | 118.270 | 4048.050 | 0.114 | 0.080 |
| 7 | 126.140 | 4055.570 | 0.127 | 0.133 |
| 8 | 133.390 | 4060.580 | 0.138 | 0.199 |
| 9 | 140.240 | 4064.180 | 0.146 | 0.278 |
| 10 | 146.850 | 4066.900 | 0.151 | 0.367 |
| 11 | 153.360 | 4069.060 | 0.154 | 0.464 |
| 12 | 159.910 | 4070.830 | 0.153 | 0.565 |
| 13 | 166.620 | 4072.340 | 0.149 | 0.665 |
| 14 | 173.670 | 4073.650 | 0.142 | 0.760 |
| 15 | 181.240 | 4074.830 | 0.132 | 0.846 |
| 16 | 189.630 | 4075.920 | 0.119 | 0.916 |
| 17 | 199.260 | 4076.960 | 0.104 | 0.966 |
| 18 | 210.870 | 4077.970 | 0.086 | 0.992 |
| 19 | 225.940 | 4078.980 | 0.066 | 0.990 |
| 20 | 248.210 | 4080.020 | 0.045 | 0.955 |
| 21 | 293.280 | 4081.150 | 0.022 | 0.888 |
| 22 | 1027.140 | 4082.420 | 0.001 | 0.786 |
| 23 | 1066.710 | 4083.960 | 0.025 | 0.649 |
| 24 | 1087.090 | 4086.050 | 0.049 | 0.480 |
| 25 | 1100.940 | 4089.590 | 0.072 | 0.282 |
| 26 | 1111.550 | 4106.450 | 0.094 | 0.059 |
| 27 | 1120.280 | 4111.910 | 0.115 | 0.183 |
| 28 | 1127.800 | 4114.190 | 0.133 | 0.438 |
| 29 | 1134.520 | 4115.630 | 0.149 | 0.698 |
| 30 | 1135.380 | 4116.030 | 1.160 | 2.452 |
| 31 | 1136.250 | 4116.500 | 1.159 | 2.135 |
| 32 | 1137.130 | 4117.080 | 1.128 | 1.744 |
| 33 | 1138.070 | 4117.850 | 1.067 | 1.290 |
| 34 | 1139.090 | 4119.130 | 0.978 | 0.784 |
| 35 | 1140.250 | 4123.290 | 0.863 | 0.240 |
| 36 | 1141.630 | 4126.360 | 0.725 | 0.326 |
| 37 | 1143.390 | 4127.480 | 0.567 | 0.898 |
| 38 | 1145.930 | 4128.160 | 0.394 | 1.457 |
| 39 | 1150.710 | 4128.660 | 0.209 | 1.988 |
| 40 | 1205.640 | 4129.070 | 0.018 | 2.473 |
| 41 | 1211.380 | 4129.410 | 0.174 | 2.897 |
| 42 | 1214.140 | 4129.720 | 0.363 | 3.244 |
| 43 | 1215.980 | 4130.010 | 0.543 | 3.503 |
| 44 | 1217.390 | 4130.280 | 0.710 | 3.663 |
| 45 | 1218.550 | 4130.550 | 0.859 | 3.718 |
| 46 | 1219.570 | 4130.820 | 0.986 | 3.663 |
| 47 | 1220.490 | 4131.110 | 1.089 | 3.498 |
| 48 | 1221.350 | 4131.420 | 1.163 | 3.225 |
| 49 | 1222.170 | 4131.770 | 1.207 | 2.850 |
| 50 | 1222.990 | 4132.190 | 1.220 | 2.382 |
| 51 | 1223.830 | 4132.730 | 1.200 | 1.834 |
| 52 | 1224.700 | 4133.550 | 1.150 | 1.220 |
| 53 | 1225.630 | 4135.350 | 1.069 | 0.558 |
| 54 | 1226.670 | 4142.810 | 0.960 | 0.134 |
| 55 | 1227.890 | 4144.010 | 0.825 | 0.835 |
| 56 | 1229.390 | 4144.660 | 0.667 | 1.524 |
| 57 | 1231.420 | 4145.120 | 0.492 | 2.180 |
| 58 | 1234.720 | 4145.480 | 0.303 | 2.783 |
| 59 | 1244.230 | 4145.780 | 0.105 | 3.314 |

Tab. 4.6.3 Wyniki symulacji porównującej wartości fitness rozwiązań z epoki 2 i 7.

Rys. 4.6.10 Wykresy przedstawiające funkcje ze znalezionymi wartościami parametrów na tle funkcji do której algorytm się dopasowuje. Na wykresie u góry, na pomarańczowo zaznaczona jest funkcja poszukiwana, a na niebiesko funkcja znaleziona dla epoki 2. Na wykresie drugim analogicznie dla epoki 7.

Analizując tabelę z wynikami analizy działania funkcji fitness (tab. 4.6.3) widać, że dla rozwiązania z epoki 7 wartość fitness nagle dynamicznie zwielokrotniała w iteracji numer 3. Jest to spowodowane tym, że punkty wykresu funkcji z rozwiązaniami z epoki 7 i funkcji poszukiwanej w tym zadaniu wypadły w tym samym punkcie, co jest oczywiście zjawiskiem pożądanym, lecz nie jednorazowo tak jak w tym przypadku. W związku z tym, że oba punkty się pokrywały ich różnica wyniosła wartość ~0 (w tabeli jest przedstawiona wartość 0 ze względu na przyjęte rozszerzenie do trzech miejsc po przecinku, co też pokazuję jak różnica jest mała), następnie ta różnica jest mianownikiem w ułamku obliczającym wartość fitness i jak widać wartość osiągnięta dla rozwiązania z teoretycznie bardziej odległym rozwiązaniem od pożądanego dominuje już na samym starcie wszystkie iteracje. Do kolejnych generacji jest brane to gorsze rozwiązaniem, które ma większe prawdopodobieństwo być wybranym jako rodzic do utworzenia nowej populacji. Dodatkowo widać, że rozwiązanie posiadające dużo mniejszy błąd w każdej kolejnej iteracji osiąga bardzo ładne wyniki, rzadko kiedy przekraczające różnicę dopasowania do funkcji szukanej większą niż 1. Gdzie w rozwiązaniu o wyższym błędzie zdarza się to znacznie częściej. Poprzez jeden szczęśliwy „traf” funkcja z rozwiązaniem z epoki 2 nie jest w stanie osiągnąć większej wartości fitness w trakcie pozostałych 56 iteracji w generacji. W związku z tym należy zastanowić się jak można ograniczyć ten negatywny przypadek. Na pierwszy rzut oka funkcja fitness jest liczona zgodnie z naszymi założeniami, cała warstwa logiczna jest prawidłowa, aczkolwiek funkcja fitness narastająca co kolejną iterację w generacji nie jest zbyt efektywna do wyznaczenia najlepszego rozwiązania. Dlatego warto zmodyfikować funkcję fitness, aby liczyła wartość fitness ( ułamek, w którego skład wchodzi w liczniku 1, a w mianowniku różnica między dopasowaniem się rozwiązania do funkcji poszukiwanej ) po zakończeniu pętli, a sumować tylko same różnice dopasowania funkcji zamiast wartości fitness. To usprawnienie będzie miało następującą postać:

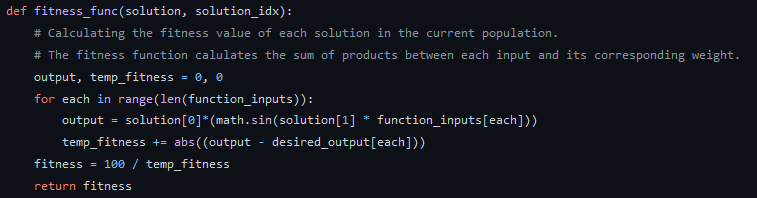
Rys. 4.6.11 Listing programu przedstawiający fragment skrypt z udoskonaloną wersją liczenie funkcji fitness

A wyniki wygenerowane po wykonaniu skryptu znajdują się w tabeli poniżej:

|  |  |  |
| --- | --- | --- |
| Numer iteracji | wartość bezwzględna różnicy wartości funkcji z epoki 2 do poszukiwanej | wartość bezwzględna różnicy wartości funkcji z epoki 7 do poszukiwanej |
| 1 | 0.021 | 0.004 |
| 2 | 0.063 | 0.010 |
| 3 | 0.125 | 0.010 |
| 4 | 0.206 | 0.025 |
| 5 | 0.305 | 0.066 |
| 6 | 0.418 | 0.146 |
| 7 | 0.546 | 0.279 |
| 8 | 0.683 | 0.479 |
| 9 | 0.829 | 0.757 |
| 10 | 0.981 | 1.124 |
| 11 | 1.134 | 1.588 |
| 12 | 1.287 | 2.152 |
| 13 | 1.436 | 2.817 |
| 14 | 1.578 | 3.577 |
| 15 | 1.710 | 4.423 |
| 16 | 1.829 | 5.339 |
| 17 | 1.933 | 6.305 |
| 18 | 2.019 | 7.298 |
| 19 | 2.085 | 8.287 |
| 20 | 2.130 | 9.242 |
| 21 | 2.153 | 10.130 |
| 22 | 2.154 | 10.916 |
| 23 | 2.179 | 11.565 |
| 24 | 2.228 | 12.045 |
| 25 | 2.300 | 12.328 |
| 26 | 2.395 | 12.387 |
| 27 | 2.509 | 12.570 |
| 28 | 2.642 | 13.008 |
| 29 | 2.791 | 13.707 |
| 30 | 3.951 | 16.159 |
| 31 | 5.109 | 18.294 |
| 32 | 6.237 | 20.038 |
| 33 | 7.304 | 21.328 |
| 34 | 8.282 | 22.111 |
| 35 | 9.145 | 22.351 |
| 36 | 9.870 | 22.677 |
| 37 | 10.437 | 23.575 |
| 38 | 10.831 | 25.032 |
| 39 | 11.040 | 27.021 |
| 40 | 11.059 | 29.494 |
| 41 | 11.233 | 32.391 |
| 42 | 11.596 | 35.635 |
| 43 | 12.139 | 39.137 |
| 44 | 12.849 | 42.800 |
| 45 | 13.708 | 46.518 |
| 46 | 14.694 | 50.181 |
| 47 | 15.783 | 53.678 |
| 48 | 16.946 | 56.903 |
| 49 | 18.152 | 59.753 |
| 50 | 19.372 | 62.135 |
| 51 | 20.572 | 63.969 |
| 52 | 21.722 | 65.188 |
| 53 | 22.791 | 65.746 |
| 54 | 23.751 | 65.880 |
| 55 | 24.575 | 66.715 |
| 56 | 25.243 | 68.239 |
| 57 | 25.735 | 70.419 |
| 58 | 26.038 | 73.202 |
| 59 | 26.143 | 76.516 |
| **Wartość fitness** | 0.038 | 0.013 |

Tab. 4.6.4 Wyniki symulacji porównującej wartości fitness rozwiązań z epoki 2 i 7 dla zmodyfikowanej metody liczenia wartości fitness.

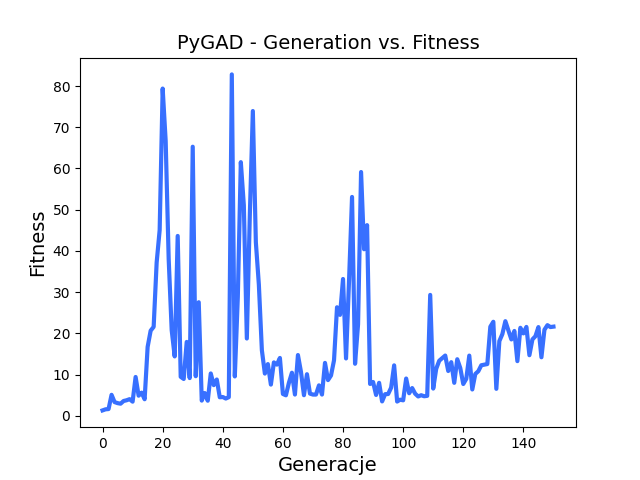
Po przeprowadzeniu symulacji na tych samych danych dla zmienionej metodyki liczenie wartości fitness z sumowania wartości fitness dla każdej iteracji na jednorazowe wyliczenie na koniec pętli. Otrzymane wyniki ( tab. 4.6.4 ) wyglądają już zacznie lepiej niż w poprzedniej symulacji. Zostały wyeliminowane skokowe wartości co iterację, w zamian za zamianę dzielenia na różnicę osiągane są liniowe przyrosty różnicy dopasowania funkcji. Dzięki temu finalnie otrzymujemy wynik fitness większy dla funkcji z epoki 2 o znacznie mniejszym błędzie. Ten przypadek świetnie obrazuje jak niewielka, wręcz niezauważalna różnica jest w stanie przysłonić ryzyko związane z niepożądanym działaniem programu. Ten przykład również obrazuje jak ważna jest funkcja licząca wartość fitness, ponieważ w oparciu o tą wartość są realizowane wszystkie procesy mające na celu wyłonienie najlepszego ( najbardziej zbliżonego ) dopasowania. Dlatego należy przykładać bardzo dużą wagę do funkcji fitness i w razie niezadowalających wyników warto szczegółowo przeanalizować każdy poszczególny krok, iterację. Na koniec jeszcze warto wrócić do dopasowania funkcji sinus z poprzedniego przykładu, tym razem symulacje zostały przeprowadzone ze zmodyfikowaną funkcją liczącą wartość fitness. Skrypt programu w stosunku do wcześniejszej wersji programu wykorzystanej do obliczenie dopasowania funkcji sinus różni się tylko fragmentem zawierającym funkcję fitness. Ten fragment został umieszczony w listingu poniżej:

Rys. 4.6.12 Listing programu definiujący zmodyfikowaną metodę zwracającą wartość fitness.

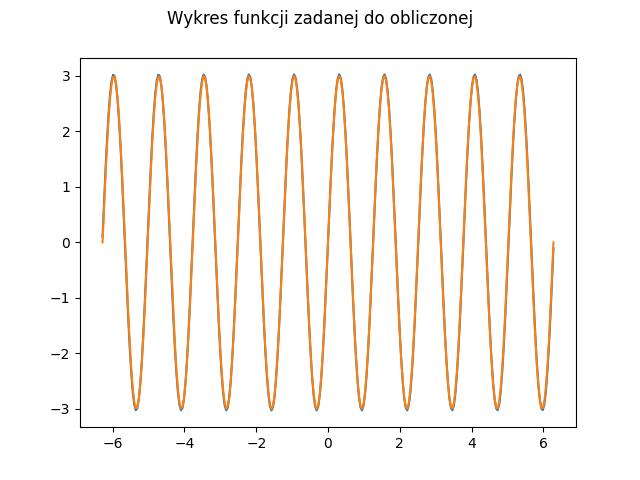
Wyniki symulacji są umieszczone w tabeli 4.6.5:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Epoka | rozmiar populacji | liczba generacji | wartość fitness | czas [s] | nr. generacji z max fitness | rozwiązanie | wartość błędu funkcji fitness [%] |
| 1 | 75 | 150 | 188.38 | 6.34 | 66 | [3.05, 4.99] | 1.40 |
| 2 | 75 | 150 | 163.89 | 6.39 | 138 | [-3.10, -5.02] | 1.01 |
| 3 | 75 | 150 | 1356.82 | 6.29 | 124 | [2.97, 5.00] | 1.03 |
| 4 | 75 | 150 | 82.82 | 6.72 | 43 | [-3.03, -4.99] | 0.58 |
| 5 | 75 | 150 | 262.72 | 6.77 | 146 | [-2.94, -5.00] | 2.04 |
| 6 | 75 | 150 | 51.39 | 6.39 | 84 | [2.96, 5.01] | 1.95 |
| 7 | 75 | 150 | 408.64 | 6.49 | 54 | [-3.02, -5.03] | 0.93 |
| 8 | 75 | 150 | 36.34 | 6.43 | 150 | [2.98, 5.01] | 0.78 |
| 9 | 75 | 150 | 284.01 | 6.46 | 64 | [3.46, 4.50] | 3.99 |
| 10 | 75 | 150 | 37.09 | 6.45 | 46 | [2.83, 4.99] | 6.57 |
|  |  |  | **średnia** | 6.47 | 92 | **mediana** | 1.22 |

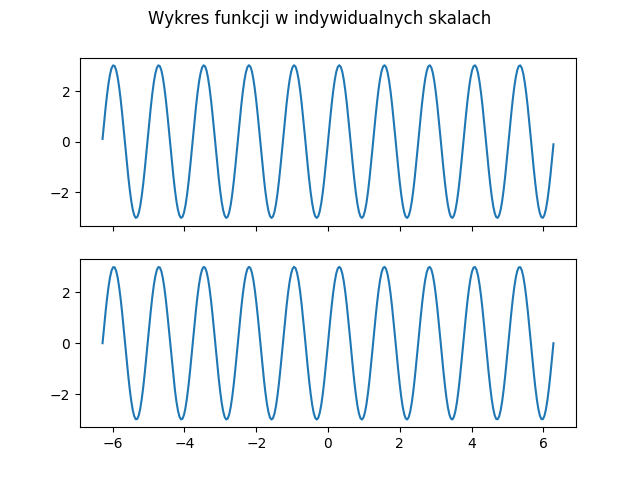
Tab. 4.6.5 Wyniki programu dla parametrów funkcji sinus: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy. Dodatkowo została zmodyfikowana funkcja fitness.



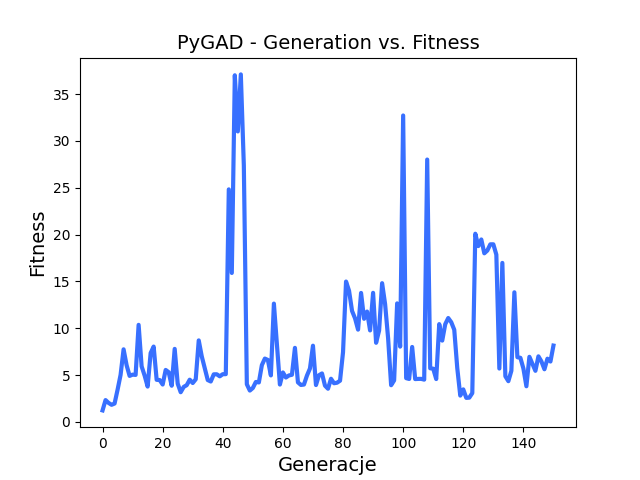
Rys. 4.6.13 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 0.58%.



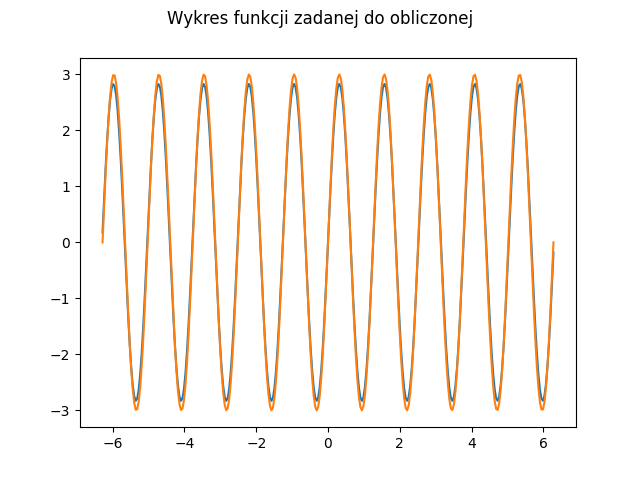
Rys. 4.6.14 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 0.58%.

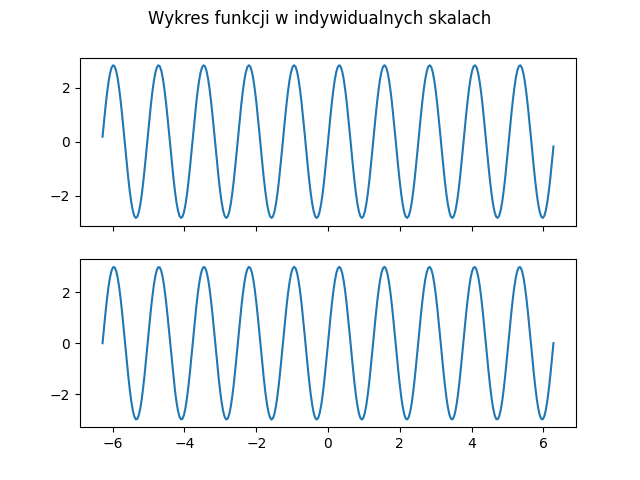


Rys. 4.6.15 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 0.58%



Rys. 4.6.16 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie wynoszącym 6.57%.

Rys. 4.6.17 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 6.57%.



Rys. 4.6.18 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 6.57%.

Po przeprowadzonej symulacji wyniki ( tab.4.6.5 ) znacząco się poprawiły, co pokazuje jak ważna jest poprawnie zdefiniowana funkcja fitness. Dzięki zmodyfikowaniu funkcji fitness mediana błędu spadła z 40.07% do 1.22%, co daję zredukowanie wartości błędu funkcji fitness o 96.95%. Znaczącej poprawie uległa rozbieżność wartości błędu w poszczególnych epokach, największa wartość błędu wynosi 6.57%, a najmniejsza 0.58%. Również poprawienie funkcji fitness wpłynęło znacząco na czas działania programu ze względu na zoptymalizowanie kodu. Sumowanie różnic dla kolejnych wartości osi x zamiast wyliczania co każdą wartość x wartości funkcji fitness znacząco poprawiło szybkość działania programu. We wcześniejszej symulacji z domyślną funkcją fitness czas działania programu wyniósł średnio 11.53 s, a w symulacji z poprawioną funkcją fitness było to już 6.47%, co jest poprawą o 43.89%. Warto również przyjrzeć się kolumnie z numerem generacji, która osiąga maksymalną wartość fitness jako pierwsza. Średnia wartość dla poprzedniej symulacji wyniosła 71, a w obecnej 92. Co daje informację, że algorytm z każdą kolejną generacją poszukuje coraz lepszych rozwiązań od tych osiągniętych obecnie, poprzez m.in. krzyżowanie, mutację i selekcję rodziców. W poprzedniej symulacji wykres przebiegu maksymalnej funkcji fitness w poszczególnych generacjach (rys. 4.6.2), jest dosyć „spłaszczony” z pojedynczym zdominowanym rozwiązaniem. Jest to całkowicie odmienna sytuacji do tej z obecnej (rys. 4.6.16), w której znacznie częściej pojawiają się lokalne maksyma zbliżone do globalnych. Warto również wspomnieć o przypadku w epoce 8, gdy najlepsze rozwiązanie zostało znalezione w ostatniej 150 generacji. To pokazuje, że algorytm w końcu działa tak jak tego można było oczekiwać i daje to też uzasadnienie do zwiększenia liczby generacji w celu znalezienie jeszcze lepszych wyników. W poprzedniej wersji funkcji fitness byłoby to zbędne. Cały ten ciąg logiczny nie wykorzystuje całego spectrum możliwości optymalizowania algorytmu genetycznego.

**5. Graficzny interfejs użytkownika**

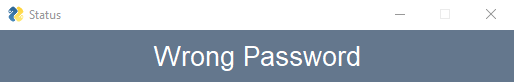
W niniejszym rozdziale zostanie zaprezentowany i szczegółowo omówiony program z graficznym interfejsem użytkownika.Program został stworzony w oparciu o podzielenie aplikacji na dwie części: „backend’ową”, zaprezentowaną w poprzednim rozdziale, jak i „frontend’ową”, która zostanie omówiona poniżej.

Przy części frontend’owej została wykorzystana biblioteka PySimpleGUI. Która jest pakietem Pythona umożliwiającym programistom na każdym poziomie zaawansowania tworzenie graficznych interfejsów użytkownika. Aplikacja ma na celu zintegrowanie wszystkich komponentów wymaganych do zainicjowania algorytmu genetycznego, tj. załadowanie parametrów, wybór pliku tekstowego z danymi i go załadowanie, uruchomienie obliczeń, wyświetlenie wyników na wykresach i możliwość debugowania programu przy użyciu informacji wyświetlanych przez program w czasie rzeczywistym, tzw. logów.

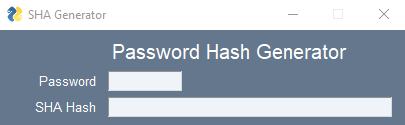
Aby uruchomić program należy uruchomić plik z rozszerzeniem .py w dowolnym kompilatorze kodu. Następnie pojawi się okienko z informacją o wpisanie hasła (rys. 5.1.1). Hasła są szyfrowane przez SHA-1 (Secure Hash Algorithm), program przechowuje szyfr hasła, a następnie wpisane hasło zaszyfrowuje i sprawdza zgodność z zapisanym. Jeżeli hasło jest nieprawidłowe to pojawi się okienko z informacją o błędnym haśle (rys. 5.1.2). Dodatkową funkcjonalnością jest możliwość zaszyfrowania tekstu po wpisaniu „gui” (rys. 5.1.3). Jeżeli walidacja okaże się prawidłowa, wyświetli się pasek postępu ładowania aplikacji (rys. 5.1.4).



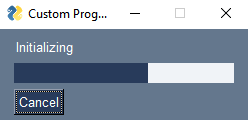
Rys. 5.1.1. Ekran startowy programu służący do walidacji hasła.



Rys.5.1.2 Komunikat informujący o błędnym haśle.



Rys.5.1.3 Po wpisaniu „gui” pojawi się pole do wpisania hasła tekstowego, a w drugim okienku wyświetli się zaszyfrowany kod przy użyciu algorytmu szyfrującego SHA-1.



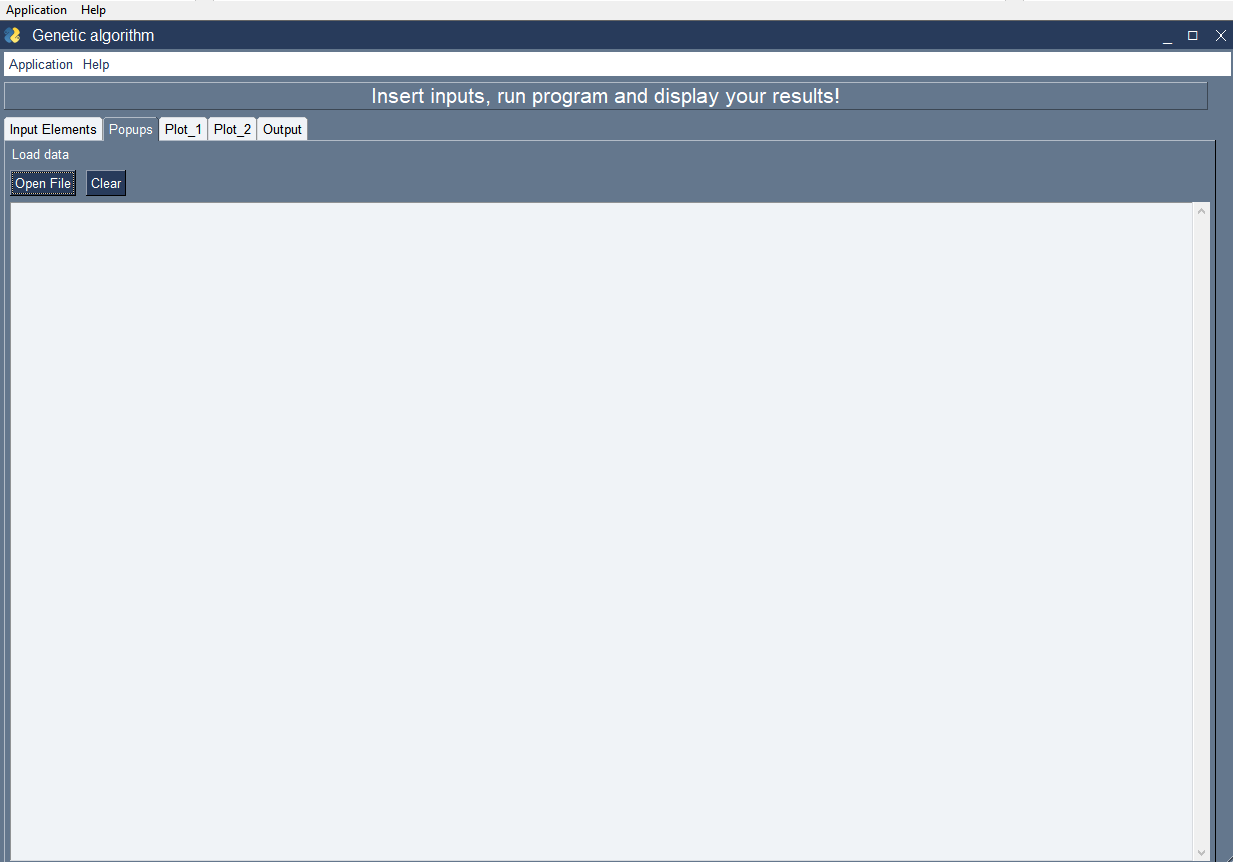
Rys.5.1.4 Pasek informujący o staniu załadowania się programu.

W następnym kroku wyświetli się okno zainicjowanego programu, składającego się z pięciu zakładek. Pierwsza z nich „Input Elements” służy do wprowadzenia niezbędnych danych potrzebnych do uruchomienia algorytmu genetycznego, o których była mowa w poprzednim rozdziale. Na samym początku należy wybrać dla jakiego typu funkcji chcemy wykonywać obliczenia, funkcja sinusoidalna (GA\_sinus) lub funkcja wielomianowa (GA\_polynomial).

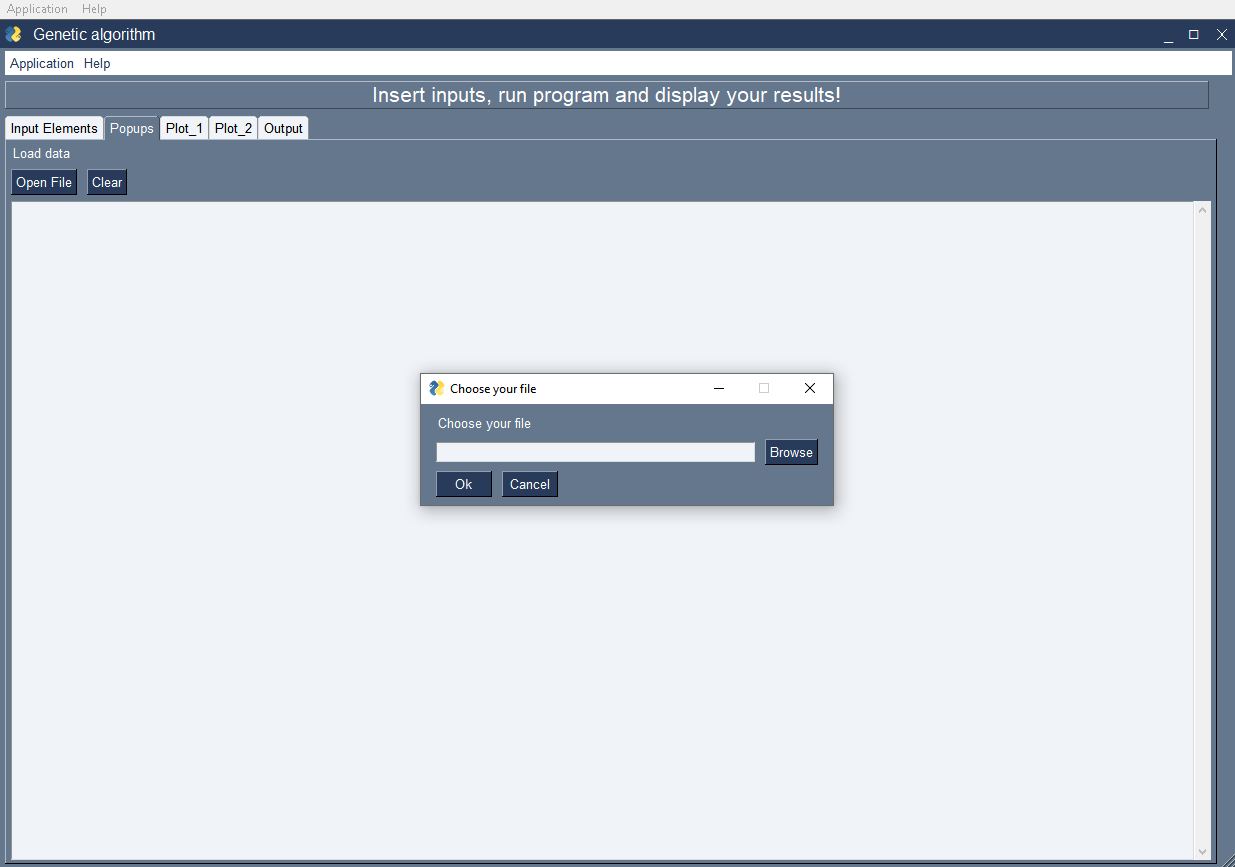


Rys.5.1.5 Okno z wyświetloną zakładką służącą do wczytania parametrów algorytmu genetycznego.

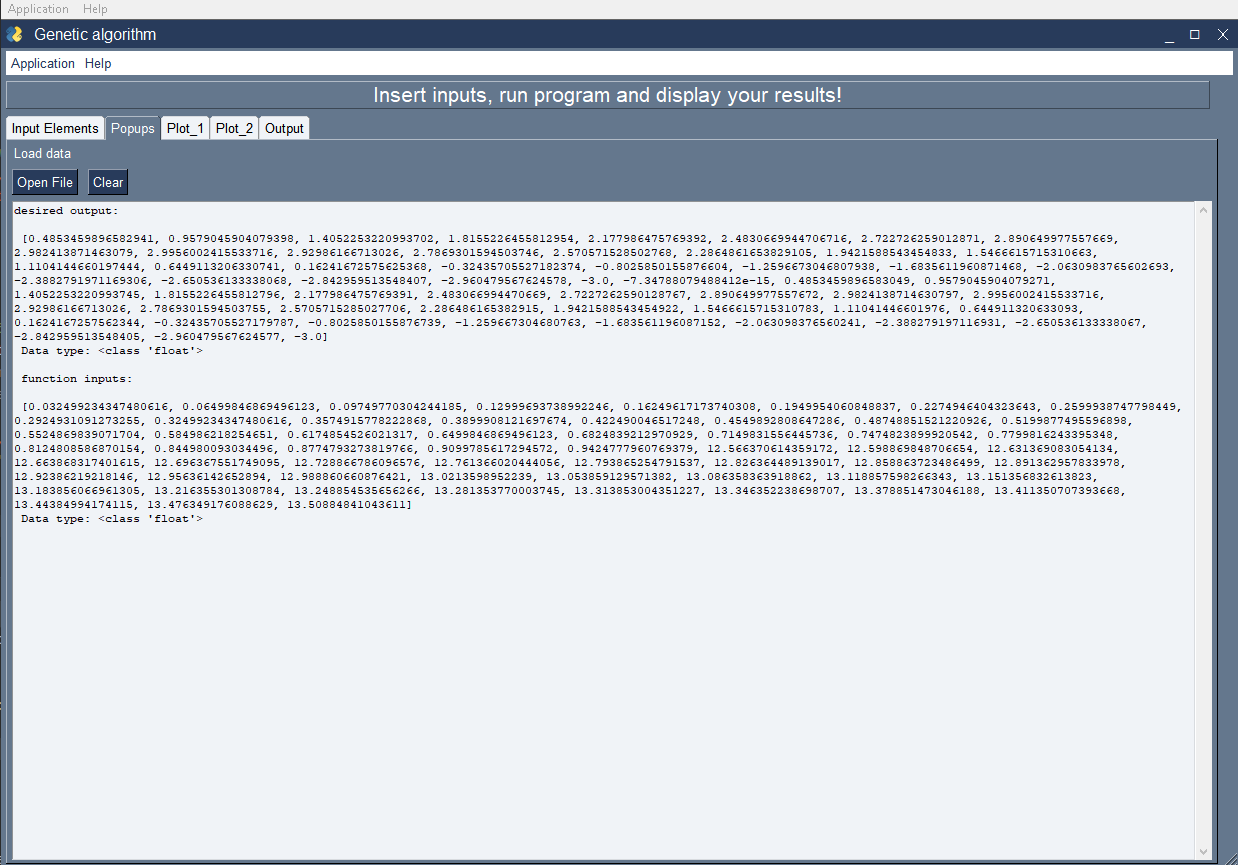
Zakładka „Popups” służy do wprowadzenia danych eksperymentu, które są wykorzystywane przez model do porównywania otrzymanych wyników i na tej podstawie wylicza funkcje kosztu algorytmu. Przycisk „Open File” służy do zlokalizowania konkretnego pliku i wczytania jego zawartości. (Rys. 5.1.6) Po wybraniu pliku wyświetli się jego zawartość w oknie zakładki.



Rys. 5.1.6 Okno zakładki „Popups” służącej do wcyztania danych eksperymentalnych.

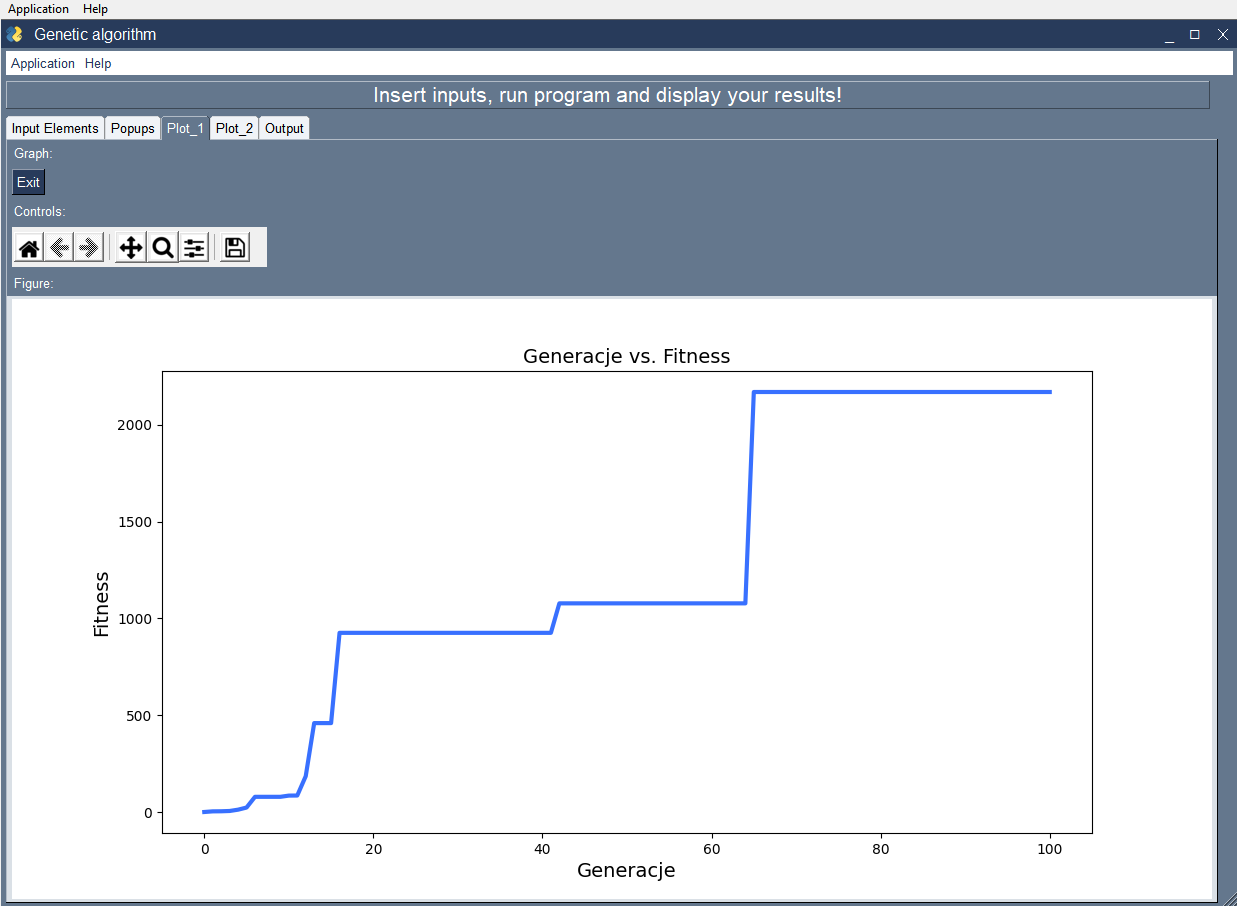
****

Rys. 5.1.7 Okienko pojawiające się po wciśnięciu przycisku „Open File”

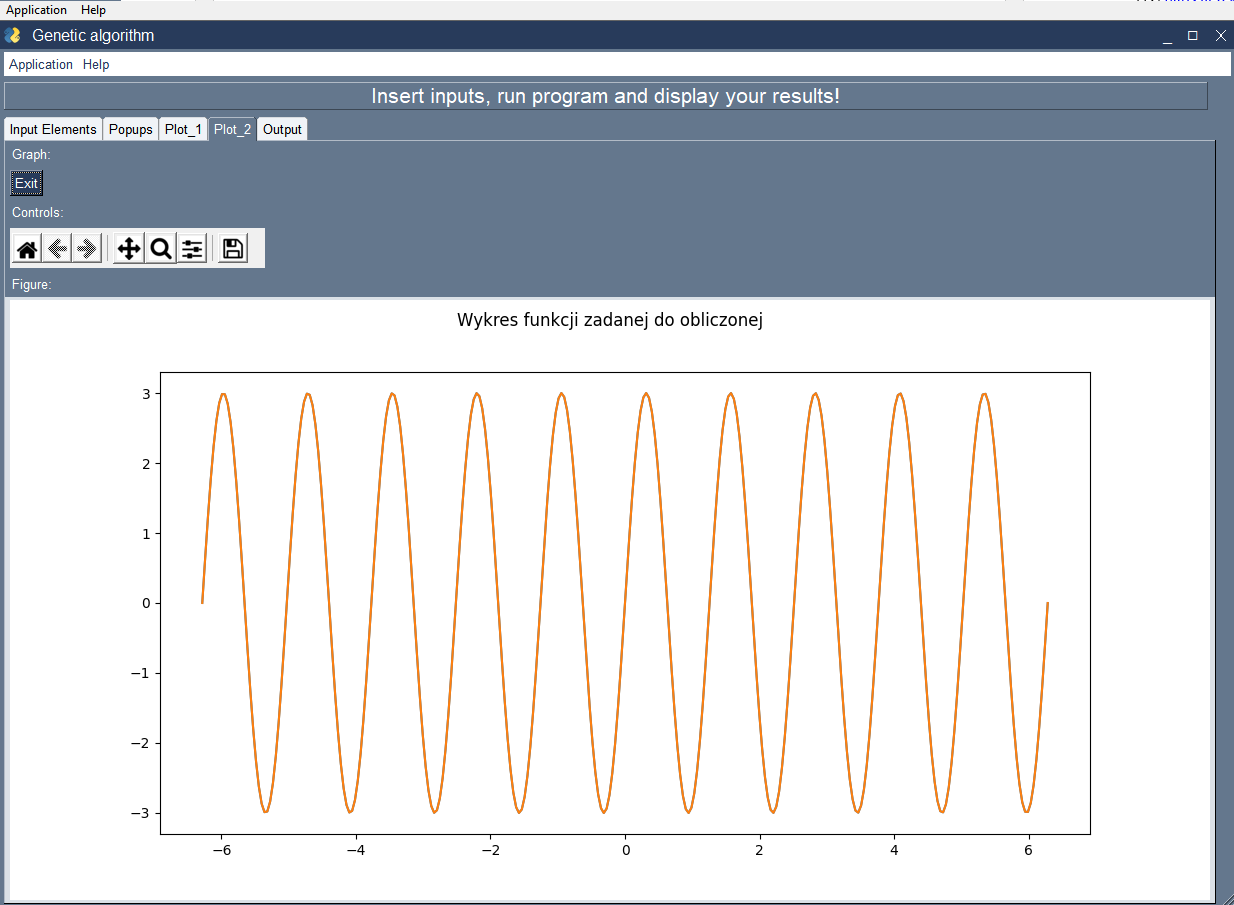
****

Rys. 5.1.8 Wczytanie przykładowych danych z pliku csv.

Po wprowadzeniu parametrów algorytmu (Rys. 5.1.5) i danych eksperymentalnych (Rys. 5.1.6) można już uruchomić program klikając w przycisk „RUN” znajdujący się w zakładce „Input Elements” (Rys. 5.1.5). Jeżeli któreś z pól parametrów, bądź dane eksperymentalne nie zostały wprowadzone program się nie uruchomi. Jeżeli wszystkie dane zostały wprowadzone, to w program zacznie liczyć współczynniki dla wybranej funkcji. Następnie zostaną wyświetlone wykresy „Plot\_1” i „Plot\_2”. Pierwszy z nich przedstawia wartość funkcji fitness na tle poszczególnych generacji (Rys. 5.1.9), drugi wyświetla wynik funkcji eksperymentu (kolor niebieski) i funkcji modelu (kolor pomarańczowy) na jednym wykresie (Rys. 5.1.10). W obu zakładkach jest możliwość wyczyszczenia okien wyświetlających wykresy lub dane, za pomocą przycisku „Clear”.

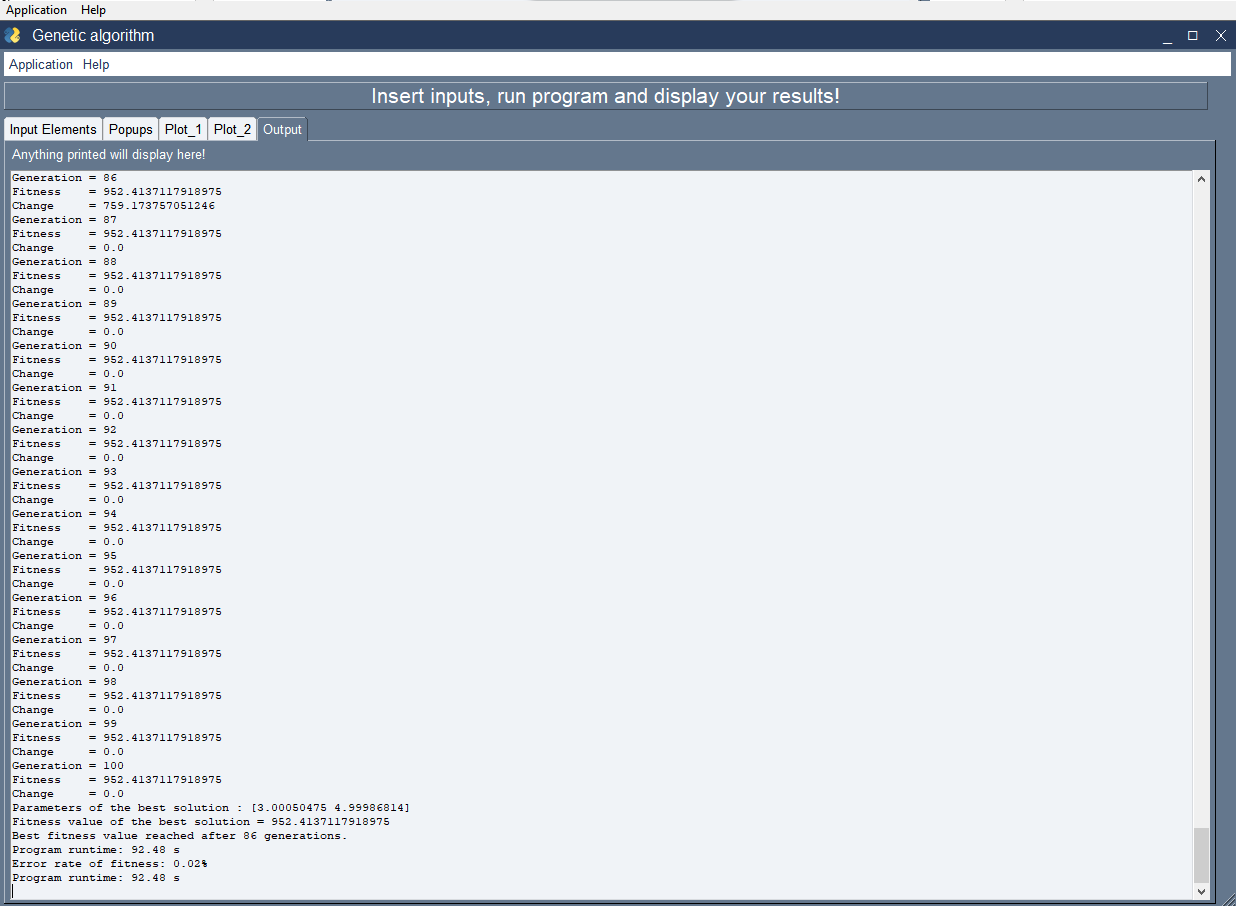


Rys. 5.1.9 Wykres w zakładce „Plot\_1” przedstawiający wartość funkcji fitness na tle generacji.

****

Rys. 5.1.10 Wykres w zakładce „Plot\_2” przedstawiający wynik funkcji eksperymentu i modelu.

W ostatniej zakładce o nazwie „Output” (Rys. 5.1.11) są wyświetlane informacje dla użytkownika o stanie programu. Jeżeli aplikacja napotka błąd to wynik tego błędu zostanie wyświetlony w oknie zakładki, jest to rozwiązanie służące ułatwieniu rozwiązywania problemów. Oprócz samych błędów, są wyświetlane informacje opisujące każde poszczególne zdarzenia wynikające z kliknięcia w dowolny przycisk. Również bardzo pomocnym elementem jest wyświetlenie wszystkich wyników dla poszczególnej generacji, a finalnie informacja o znalezionych parametrach funkcji, największej wartości fitness, w której generacji wystąpiła, czas trwania obliczeń algorytmu i wartość błędu.



Rys. 5.1.11 Zakładka „Output” pomagająca w monitorowaniu działania programu.

**6. Podsumowanie**

Cel niniejszej pracy, którym jest zaimplementowanie biblioteki z algorytmem genetycznym PyGAD do znajdowania współczynników szukanej funkcji jako graficzny interfejs użytkownika w języku python, udało się zrealizować.Finalnym efektem pracy jest działająca aplikacja z pełnym funkcjonalnym interfejsem graficznym, również napisanym w pythonie przy wykorzystaniu biblioteki PySimpleGUI.

Wykonana w rozdziale 4 analiza i przykładowa kalibracja modelu algorytmu genetycznego przeprowadziła przez ścieżkę przyczynowo - skutkową do osiągnięcia zadowalających rezultatów, jak i lepszego zrozumienia natury problemu i poznania możliwości jednego z chętniej wykorzystywanych algorytmów w problemach kombinatorycznych. Algorytm uzyskał wysoką skuteczność przy modyfikacji parametrów tj. rodzaj selekcji rodziców, liczbie populacji, liczbie generacji i przede wszystkim sposobie liczenia funkcji fitness, która jest głównym elementem decyzyjnym przy wyborze współczynników funkcji. Warto nadmienić, że zabiegi związane z modyfikacją parametrów przyczyniały się do sukcesywnego usprawniania wyniku obliczeniowego, ale również wpływały na zwiększanie czasu obliczeń. Przytaczając wstęp pracy, mówiący o poszukiwaniu rozwiązanie optymalnego, dążącego do znalezienia parametrów o stosunkowo małym błędzie w dosyć krótkim czasie. I to właśnie udał się w tej pracy zrealizować.

Następnie w rozdziale 5 rdzeń programu został opakowany w graficzny interfejs użytkownika, w celu ułatwienia użytkownikowi uruchomienia wszystkich niezbędnych kroków do otrzymania wyników. Aplikacja zawiera zakładki służące do wprowadzenia parametrów, wczytania danych, wyświetlenia wykresów i monitowania danych wyjściowych. Dużym ułatwieniem jest również możliwość wyczyszczenia danych wejściowych i załadowania nowych parametrów w celu ponownego uruchomienia algorytmu ze zmienionymi parametrami.

Oprócz niewątpliwie wielu zalet algorytmów genetycznych o których się przekonałem w trakcie pisania pracy, to również jest kilka mankamentów. Jednym z nich jest relatywnie niska uniwersalność, co ma związek z modyfikowaniem funkcji kosztu pod konkretny problem z jakim w tym momencie ma się do czynienia. Nie jest możliwe stworzenie aplikacji do rozwiązywania problemu z dopasowaniem każdego rodzaju funkcji matematycznej, bez ingerencji do kodu źródłowego. Ewentualnie, gdyby rozważać ten problem jako znacznie większy projekt, to oczywiście byłoby to wykonalne. Gdyby rozbudować logikę programu o metody liczące każdy rodzaj funkcji wprowadzonej przez użytkownika, aczkolwiek byłby to program bardzo dużych rozmiarów. Z kolei drugi problem jaki mi się nasunął jest związany ze słabym wykrywaniem potencjalnych przestrzeni rozwiązań. Ten krok odbywa się czysto losowo z zakresu liczb jakie podamy algorytmowi. Dodatkowo też można było zauważyć problem z utknięciem na lokalnych minimach, z czym można sobie poradzić np. poprzez zwiększenie prawdopodobieństwa krzyżowania kosztem większej liczby iteracji.

Dalszym krokiem rozwoju aplikacji może być próba sprawdzenia skuteczności algorytmu dla bardziej złożonych równań nieliniowych. Ale przedtem warto stworzyć głęboką sieć neuronową, dzięki której dobierzemy wszystkie wartości parametrów algorytmu, ponieważ zmieniając kilka z nich i otrzymując zadowalający efekt, nie jesteśmy w stanie odpowiedzieć na pytanie czy przy innej modyfikacji owych parametrów nie uzyskalibyśmy lepszego rezultatu. Innym kierunkiem rozwoju problemu przedstawionego w pracy, byłoby zbudowanie hybrydowej metaheurystyki składającej się na przykład z algorytmu genetycznego i algorytmu mrówkowego. W tej sytuacji algorytm mrówkowy świetnie się sprawdzi w obszarach, których algorytm genetyczny radzi sobie słabiej.

**7. Bibliografia**

*[1] M.Dorigo, Thomas Stϋtzle. 2004. ANT COLONY OPTIMIZATION. Massachusetts Institute of Technology.*

*[2] K.Sorensen, F.Glover. 2010. METAHEURISTICS. University of Antwerp, University of Colorado and OptTek Systems. Str. 2*

*[3] A.Kaveh, T.Bakhshpoori. 2019. METAHEURISTICS: OUTLINES, MATLAB CODES AND EXAMPLES. Springer International Publishing AG. Str. 1 - 5*

*[4] T. El-Ghazali. 2009. METAHEURISTICS: FROM DESIGN TO IMPLEMENTATION. John Wiley & Sons, Inc. Str. 2-3, 199-200*

*[5] I J.* [*Lepagnot,*](https://primo-48tuw.hosted.exlibrisgroup.com/primo-explore/search?query=creator%2Cexact%2C%20Lepagnot%2C%20Julien%20&tab=default_tab&search_scope=primo_all_scope&vid=48TUW_VIEW&lang=pl_PL&offset=0) *P.Siarry.* *A SURVEY ON OPTIMIZATION METAHEURISTICS. Information sciences, 2013-07-10, Vol.237, Str.82-11*

*[6]* [*http://metaheuristics.eu*](http://metaheuristics.eu) *(01.05.2021)*

# *[7] C.Blum, J.Puchinger, G.R.Raidl, A.Roli. 2011.HYBRID METAHEURISTICS IN COMBINATORIAL OPTIMIZATION: A SUREY, Applied soft computing, 2011, Vol.11 (6), p.4135-4151. Elsevier B.V*

*[8] R. Chibante, A. Araújo, A.Carvalho.*

*[9] Glover F., Laguna M. (1997). TABU SEARCH. Springer, Boston, MA. Str. 3-4*

*[10] T.Ying. SWARM INTELLIGENCE - VOLUME 3: APPLICATIONS. 2019. Institution of Engineering and Technology.*

*[11] T.Ying. SWARM INTELLIGENCE - VOLUME 1: PRINCIPLES, CURRENT ALGORITHMS AND METHODS. 2018. Institution of Engineering and Technology. Str. 3*

*[12] https://en.wikipedia.org/wiki/No\_free\_lunch\_in\_search\_and\_optimization* (06.05.2021)

*[13] K.Lakshmi. GENCOS WIND–THERMAL SCHEDULING PROBLEM USING ARTIFICIAL IMMUNE SYSTEM ALGORITHM. 2014. International journal of electrical power & energy systems. tom:54 s.:112 -122*

*[14] M. Ünal, A. Ak, V.Topuz, H.Erdal. 2013. OPTIMIZATION OF PID CONTROLLERS USING ANT COLONY AND GENETIC ALGORITHMS. Springer. Str. 31,32*

*[15] J. S. Angelo, H. J.C. Barbosa. 2011. ON ANT COLONY OPTIMIZATION ALGORITHMS FOR MULTIOBJECTIVE PROBLEMS. Laborat´orio Nacional de Computa¸c˜ao Cient´ıfica LNCC/MCT. Str. 58,59*

*[16] V. Maniezzo, L.M. Gambardella, F. de Luigi. 2004. ANT COLONY OPTIMIZATION. Springer-Verlag Berlin Heidelberg. Str. 101-102*

*[17] M.R. Bonyadi, M.R Azghadi, H. Shah-Hossein. 2008. POPULATION-BASED OPTIMIZATION ALGORITHMS FOR SOLVING THE TRAVELLING SALESMAN PROBLEM. Department of Electrical and Computer Engineering, Shahid Beheshti University. Str. 1,2*

*[18]* *Y. Feng, G. Wang, S. Deb, M. Lu, X. ZHAO. SOLVING 0–1 KNAPSACK PROBLEM BY A NOVEL BINARY MONARCH BUTTERFLY OPTIMIZATION. Neural computing & applications, 2019-09, Vol.31 (9), p.5477-5495. London: Springer London*

*[19]* [*https://pl.wikipedia.org/wiki/Problem\_komiwoja%C5%BCera*](https://pl.wikipedia.org/wiki/Problem_komiwoja%C5%BCera) *(10.05.2021)*

*[20]* [*https://www.baeldung.com/cs/p-np-np-complete-np-hard*](https://www.baeldung.com/cs/p-np-np-complete-np-hard) *(11.05.2021)*

*[21] A REVIEW OF ANT ALGORITHMS. Expert systems with applications [0957-4174] Mullen, R J rok:2009 tom:36 zesz.:6 s.:9608 -9617*

*[22] A MAX–MIN ANT SYSTEM FOR THE SPLIT DELIVERY WEIGHTED VEHICLE ROUTING PROBLEM. Expert systems with applications [0957-4174] Tang, Jiafu rok:2013 tom:40 zesz.:18 s.:7468 -7477*

*[23] DYNAMIC CONSTRUCTION SITE LAYOUT PLANNING USING MAX-MIN ANT SYSTEM. Automation in construction [0926-5805] Ning, Xin rok:2010 tom:19 zesz.:1 s.:55 -65*

*[24] E. Chen, X. Liu. MULTI-COLONY ANT ALGORITHM. 2010. School of Business Administration, Shandong Institute of Commerce and Technology. Str. 4 – 5*

*[25] ANT-INSPIRED ALGORITHMS FOR DECISION TREE INDUCTION.   
Information Technology in Bio- and Medical Informatics [0302-9743] Bursa, Miroslav rok:2015 s.:95 -106*

*[26] K. Trojanowski. METAHEURYSTYKI PRAKTYCZNIE. 2008. Warszawa : Wyższa Szkoła Informatyki Stosowanej i Zarządzania.*

*[27] C. Eyckelhof, M. Snoek. ANT SYSTEMS FOR A DYNAMIC TSP. 2002. Berlin, Heidelberg: Springer Berlin*

*[28]* [*https://pl.wikipedia.org/wiki/Python*](https://pl.wikipedia.org/wiki/Python) *(07.11.2021)*

*[29]* *https://pl.wikipedia.org/wiki/PyCharm* *(07.11.2021)*

*[30]* [*https://www.mygreatlearning.com/blog/python-numpy-tutorial/*](https://www.mygreatlearning.com/blog/python-numpy-tutorial/) *(08.11.2021)*

*[31]* *https://matplotlib.org/stable/tutorials/introductory/pyplot.html (08.11.2021)*

*[32]* *https://www.geeksforgeeks.org/random-setstate-in-python/?ref=lbp (08.11.2021)*

*[33]* *https://docs.python.org/3/library/math.html (08.11.2021)*

*[34]* *https://www.oreilly.com/library/view/python-standard-library/0596000960/ch01s15.html (08.11.2021)*

*[35]* [*https://www.mygreatlearning.com/blog/python-numpy-tutorial/*](https://www.mygreatlearning.com/blog/python-numpy-tutorial/) *(08.11.2021)*

*[36]* *https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP (08.11.2021)*

*[37] S.N. Sivanandam, S.N. Deepa. GENETIC ALGORITHMS. Springer-Verlag Berlin Heidelberg 2008*

# *[38] Lee Jacobson; Burak Kanber. GENETIC ALGORITHMS IN JAVA BASICS. Apress 2015*

# *[39] Sanghamitra Bandyopadhyay, Sankar K. Pal. GENETIC ALGORITHMS. Springer-Verlag Berlin Heidelberg 2007*

# *[40] Abdel-aal H. Mantawy. GENETIC ALGORITHMS APPLICATION TO ELECTRIC POWER SYSTEMS. 2011. Ain Shams University, Faculty of Engineering, Egypt.*

# *[41]* [*https://medium.com/the-andela-way/on-genetic-algorithms-and-their-application-in-solving-regression-problems-4e37ac1115d5*](https://medium.com/the-andela-way/on-genetic-algorithms-and-their-application-in-solving-regression-problems-4e37ac1115d5) *(17.12.2021)*

# *[42] https://www.linkedin.com/pulse/introduction-optimization-genetic-algorithm-ahmed-gad (18.12.2021)*

# *[43]* [*https://medium.com/the-andela-way/on-genetic-algorithms-and-their-application-in-solving-regression-problems-4e37ac1115d5*](https://medium.com/the-andela-way/on-genetic-algorithms-and-their-application-in-solving-regression-problems-4e37ac1115d5) *(17.12.2021)*

# *[43] https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4 (17.12.2021)*

# *[44] https://github.com/ahmedfgad/GeneticAlgorithmPython (19.12.2021)*

# *[45] https://pygad.readthedocs.io/en/latest/README\_pygad\_ReadTheDocs.html#random-mutation (19.12.2021)*

# *[46] https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-keras (19.12.2021)*

# *[47] https://www.analyticsvidhya.com/blog/2021/04/a-gentle-introduction-to-pytorch-library/ (19.12.2021)*

# *[48] https://en.wikipedia.org/wiki/TensorFlow (19.12.2021)*

# *[49] https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 (18.12.2021)*

# *[50]* [*https://sjp.pwn.pl/szukaj/tabu.html*](https://sjp.pwn.pl/szukaj/tabu.html) *(20.12.2021)*

# *[51] https://www.researchgate.net/figure/the-canonical-GA-algorithm\_fig1\_224178234* *(18.12.2021)*

# *[52] https://en.wikipedia.org/wiki/No\_free\_lunch\_in\_search\_and\_optimization* *(26.01.2021)*

1. **Spis rysunków:**

*Rys. 2.1.1 Mapa odwiedzanych miast z najkrótszą ścieżką (A, B, C, D, E) 5*

*Rys. 3.2.1 Wykres przedstawia generowanie pełnego cyklu jednego pokolenia w algorytmach ewolucyjnych 11*

*Rys. 3.2.2 Rysunek przedstawia algorytm symulowanego wyżarzania w postaci schematu blokowego 13*

*Rys.3.2.3 Rysunek przedstawiający cztery wymiary wyszukiwania tabu 15*

*Rys. 3.2.4 Schemat blokowy proponowanego algorytmu AIS 16*

*Rys. 3.2.5 Tor do przeprowadzenia eksperymentu podwójnego mostu. (a) Mosty są równej długości. (b) Jeden z mostów jest dwukrotnie większy od drugiego 19*

*Rys. 3.2.6 Wykresy przedstawiające wyniki przeprowadzonych eksperymentów. (a) Wynik dla przypadku obu mostów tej samej długości, na wykresie widać, że mrówki wybierały jedną z dwóch ścieżek na zmianę. (b) Wynik dla przypadku, gdy jeden z mostów jest dokładnie dwukrotnie dłuższy od drugiego, wykres obrazuje dominację krótszej ścieżki 19*

*Rys.3.2.7 W tym eksperymencie tylko dłuższa ścieżka była dostępna od początku. Po 30 minutach, kiedy uformował się feromon na jedynej dostępnej ścieżce, została dodana krótsza ścieżka. (a) Początkowa faza przeprowadzania badania i zaktualizowana po 30 minutach, poprzez dodanie krótszej ścieżki. (b) W większości przypadków, po dodaniu krótszej ścieżki, większość mrówek wciąż chętniej wybierała dłuższą trasę. 20*

*Rys. 3.3.1 Selekcja poprzez koło ruletki 22*

*Rys. 3.3.2 Stochastyczna selekcja uniwersalna 23*

*Rys. 3.3.3 Przykład działania jednopunktowego krzyżowania: (a) przed krzyżowaniem; (b) po krzyżowaniu 24*

*Rys. 3.3.4 Przykład mutacji bit po bicie 25*

*Rys. 3.3.5 Populacja, chromosomy i geny 27*

*Rys. 3.3.6 Schemat blokowy działania algorytmu mrówkowego 29*

*Rys. 4.2.1 Schemat blokowy przedstawiający cykl życia programu 33*

*Rys. 4.4.1 Listing programu przedstawiający wygenerowaną populację algorytmu 41*

*Rys. 4.4.2 Listing programu przedstawiający metodę zwracająca wartość fitness dla wylosowanych parametrów 41*

*Rys. 4.4.3 Listing programu przedstawiający zaimportowanie modułu, utworzenie instancji klasy i wywołanie metody run() 42*

*Rys. 4.4.4 Listing programu przedstawiający równanie z poszukiwanymi sześcioma parametrami 42*

*Rys. 4.4.5 Listing programu przedstawiający zdefiniowanie początkowych parametrów i funkcji fitness 42*

*Rys. 4.4.6 Listing programu przedstawiający zdefiniowanie wartości pozostałych parametrów algorytmu 43*

*Rys. 4.4.7 Listing programu przedstawiający utworzoną instancję klasy 43*

*Rys. 4.4.8 Listing programu przedstawiający wywołanie metody run() utworzonej instancji 43*

*Rys. 4.4.9 Listing programu przedstawiający wywołanie metody plot\_results() 43*

*Rys. 4.4.10 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje 44*

*Rys. 4.4.11 Listing programu przedstawiający wyświetlenie w konsoli parametrów, wartości fitness I indeksu w populacji dla najlepszego rozwiązania 44*

*Rys. 4.5.1 Listing programu przedstawiający poszukiwane równanie funkcji wielomianowej o trzech parametrach 45*

*Rys. 4.5.2 Listing programu przedstawiający zaimportowanie bibliotek i modułów, oraz przypisanie do zmiennej czasu startu działania programu 45*

*Rys. 4.5.3 Listing programu przedstawiający skrypt wczytujący dane z pliku tekstowego 45*

*Rys. 4.5.4 Listing programu definiujący metodę zwracającą wartość fitness 46*

*Rys. 4.5.5 Listing programu przedstawiający zadeklaraowane wartości parametrów klasy 46*

*Rys. 4.5.6 Listing programu przedstawiający utworzenie instancji klasy pygad dla określonuch parametrów poszukującej wartości parametrów dla najlepszego rozwiązania 46*

*Rys. 4.5.7 Listing programu wywołujący metodę klasy pygad run(), wczytanie i wyświetlenie najlepszego rozwiązania 47*

*Rys. 4.5.8 Listing programu wyświetlajacy wykres przedstawiający funkcję zadaną do obliczonej przez algorytm 47*

*Rys. 4.5.9 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 0.87% 48*

*Rys. 4.5.10 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 0.87% 49*

*Rys. 4.5.11 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie 74.67% 49*

*Rys. 4.5.12 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 74.67% 50*

*Rys. 4.5.13 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 1.27% 51*

*Rys. 4.5.14 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 1.27% 52*

*Rys. 4.5.15 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie 79.15% 52*

*Rys. 4.5.16 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 79.15% 53*

*Rys. 4.5.17 Listing programu przedstawiający zadeklaraowane wartości parametrów klasy 54*

*Rys. 4.5.18 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 0.09% 55*

*Rys. 4.5.19 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 0.09% 55*

*Rys. 4.5.20 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie 27.79% 56*

*Rys. 4.5.21 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 27.79% 56*

*Rys. 4.6.1 Listing programu przedstawiający poszukiwane równanie funkcji sinus o dwóch parametrach 57*

*Rys. 4.6.2 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 1.18% 58*

*Rys. 4.6.3 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 1.18% 58*

*Rys. 4.6.4 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 1.18% 59*

*Rys. 4.6.5 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie wynoszącym 96.27% 59*

*Rys. 4.6.6 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 96.27% 60*

*Rys. 4.6.7 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 96.27% 60*

*Rys. 4.6.8 Listing programu przedstawiający funkcję fitness algorytmu genetycznego w domyślnej wersji modułu PyGad 61*

*Rys. 4.6.9 Listing programu przedstawiający cały skrypt programu do analizy działania funkcji fitness 62*

*Rys. 4.6.10 Wykresy przedstawiające funkcje ze znalezionymi wartościami parametrów na tle funkcji do której algorytm się dopasowuje. Na wykresie u góry, na pomarańczowo zaznaczona jest funkcja poszukiwana, a na niebiesko funkcja znaleziona dla epoki 2. Na wykresie drugim analogicznie dla epoki 7 64*

*Rys. 4.6.11 Listing programu przedstawiający fragment skrypt z udoskonaloną wersją liczenie funkcji fitness 65*

*Rys. 4.6.12 Listing programu definiujący zmodyfikowaną metodę zwracającą wartość fitness 67*

*Rys. 4.6.13 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o najmniejszym błędzie wynoszącym 0.58% 68*

*Rys. 4.6.14 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 0.58% 69*

*Rys. 4.6.15 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 0.58% 69*

*Rys. 4.6.16 Wykres przedstawiający maksymalną wartość fitness przypadająca na poszczególne generacje dla rozwiązania o największym błędzie wynoszącym 6.57% 70*

*Rys. 4.6.17 Wykres (x, y) przedstawiający zbieżność dla otrzymanego rozwiązania przy błędzie funkcji fitness wynoszącym 6.57% 70*

*Rys. 4.6.18 Wykresy obu funkcji przedstawione oddzielnie, na różnych skalach, otrzymanych dla rozwiązania o błędzie wynoszącym 6.57% 71*

*Rys. 5.1.1 Ekran startowy programu służący do walidacji hasła 72*

*Rys. 5.1.2 Komunikat informujący o błędnym haśle. 72*

*Rys. 5.1.3 Po wpisaniu „gui” pojawi się pole do wpisania hasła tekstowego, a w drugim okienku wyświetli się zaszyfrowany kod przy użyciu algorytmu szyfrującego SHA-1 72*

*Rys. 5.1.4 Pasek informujący o staniu załadowania się programu. 73*

*Rys. 5.1.5 Okno z wyświetloną zakładką służącą do wczytania parametrów algorytmu genetycznego. 73*

*Rys. 5.1.6 Okno zakładki „Popups” służącej do wczytania danych eksperymentalnych*

*74*

*Rys. 5.1.7 Okienko pojawiające się po wciśnięciu przycisku „Open File” 74*

*Rys. 5.1.8 Wczytanie przykładowych danych z pliku csv. 75*

*Rys. 5.1.9 Wykres w zakładce „Plot\_1” przedstawiający wartość funkcji fitness na tle generacji. 76*

*Rys. 5.1.10 Wykres w zakładce „Plot\_2” przedstawiający wynik funkcji eksperymentu i modelu. 76*

*Rys. 5.1.11 Zakładka „Output” pomagająca w monitorowaniu działania programu. 77*

1. **Spis tabel:**

*Tab. 4.5.1 Wyniki programu dla parametrów funkcji wielomianowej: rozmiar populacji i liczba generacji przyjęto 50, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców jest domyślny, czyli selekcja ustalona 48*

*Tab. 4.5.2 Wyniki programu dla parametrów funkcji wielomianowej: rozmiar populacji wynoszący 75, liczba generacji przyjęto 200, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców jest domyślny, czyli selekcja ustalona 51*

*Tab. 4.5.3 Wyniki programu dla parametrów funkcji wielomianowej: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy 54*

*Tab. 4.6.1 Wyniki programu dla parametrów funkcji sinus: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy 57*

*Tab. 4.6.2 Wyniki programu dla parametrów funkcji sinus: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy. Z wybranymi dwoma najbardziej skrajnymi rozwiązaniami i wartość fitness jest liczona w metodyce selekcji ustalonej, zamiast turniejowej, ze względu na trudność optymalnego przedstawienia różnic w obliczeniach funkcji fitness 61*

*Tab. 4.6.3 Wyniki symulacji porównującej wartości fitness rozwiązań z epoki 2 i 7*

*63-64*

*Tab. 4.6.4 Wyniki symulacji porównującej wartości fitness rozwiązań z epoki 2 i 7 dla zmodyfikowanej metody liczenia wartości fitness 65-67*

*Tab. 4.6.5 Wyniki programu dla parametrów funkcji sinus: rozmiar populacji wynoszący 75, liczba generacji przyjęto 150, liczba rozwiązań wybranych na rodziców do puli rozrodczej wynosi 7, typ wyboru rodziców to wybór turniejowy. Dodatkowo została zmodyfikowana funkcja fitness 68*