Algorytmy ewolucyjne

Zasada działania, obszary zastosowań, biblioteki programistyczne, zastosowanie do wybranego problemu optymalizacyjnego oraz eksperymenty

Paweł Malec, Miłosz Darecky

WSZiB, 2020

1. Zasada działania i zastosowania

Ewolucja, to jedno z wielkich odkryć ludzkości. Przetrwanie najsilniejszych, eliminacja najsłabszych, teoria ewolucji Darwina pokazała nam, jak doszliśmy do aktualnego stanu ludzkości. Teraz wyobraźmy sobie, że tak samo mogą ewoluować nasze programy. Na YouTube można znaleźć wiele ciekawych filmów o Algorytmie ewolucji(nazywanym w skrócie EA). Większość takich eksperymentów pokazuje programy, które poprzez metodę prób i błędów są w stanie przejść jak najszybciej jakąś prostą grę(jak mario). Algorytmy ewolucyjny dzielą się na: algorytmy genetyczne, programowanie genetyczne oraz strategie ewolucyjne. W algorytmie ewolucyjnym problem, który mamy rozwiązać staje się środowiskiem, w którym „żyje" populacja osobników podobnie jak w projekcie na naszych zajęciach sztucznego życia Framstick. Każdy z takich osobników reprezentuje możliwe rozwiązanie problemu. Taką analogię możemy zauważyć w procesie biologicznym, algorytm ewolucyjny tworzy stopniowo coraz to lepsze rozwiązania. Wynika z tego, że możemy to wykorzystać do rozwiązywania problemów optymalizacyjnych czyli takich gdzie rozwiązanie polega na znalezieniu największej bądź najmniejszej wartości pewnego parametru problemu, która spełnia daną własność. Schemat AE można przedstawić następująco:

1.Wygenerowanie populacji startowej (np. poprzez losowanie geotypów).

2.Powtarzanie do wystąpienia określonego warunku stop (np. otrzymanie odpowiedniej jakości rozwiązań, osiągnięcie odpowiedniej ilości powtórzeń).

a) Wybranie najlepszych osobników (selekcja genotypów).

b) Dokonaj krzyżowania pośród poprzednio wybranych osobników (np. z określonym prawdopodobieństwem).

c) Dokonaj mutacji genotypów (np. z określonym prawdopodobieństwem).

d) Odrzuć najsłabsze osobniki, zastępując je ich potomstwem.

Główne zastosowanie algorytmów ewolucyjnych w rzeczywistym świecie:

- Rozproszone topologie sieci komputerowych
- Filtrowanie i przetwarzanie sygnałów
- Tworzenie harmonogramów
- Planowanie komunikacji satelitarnej
- Optymalizacja struktury molekularnej (Chemia)
- Problem z podróżującymi sprzedawcami (Problem komiwojażera)
- Alokacja plików w systemach rozproszonych
- Projektowanie układów elektronicznych

- Budowanie twarzy podejrzanych przez naocznych świadków w kryminalistyce
- Tworzenie zestawów reguł
- Nauka zachowania robotów

2. Biblioteki programistyczne

Jenetics to biblioteka napisana dla języka Java. Jest ona zaprojektowana z wyraźnym rozdzieleniem kilku koncepcji algorytmu, np. Gen, Chromosom, Genotyp, Fenotyp, Populacja i funkcja fitness. Jenetics pozwala na zminimalizowanie i zmaksymalizowanie danej funkcji fitness bez jej udoskonalania. W przeciwieństwie do innych implementacji Google Analytics(w skrócie GA), biblioteka ta wykorzystuje koncepcję strumienia ewolucji(EvolutionStream) do realizacji kroków ewolucji. Ponieważ EvolutionStream implementuje interfejs Java Stream, działa on płynnie z pozostałą częścią API Java Stream. Tą bibliotekę możemy zainstalować w systemie Linux za pomocą komendy git clone https://github.com/Jenetics/jenetics.git <builddir> pod warunkiem posiadania już posiadania Java JRE11 oraz JDK11.

DEAP jest frameworkiem języka programowania Python do wdrażania algorytmów ewolucyjnych. Służy do szybkiego prototypowania i testowania pomysłów. Dąży do tego, aby algorytmy były jednoznaczne, a struktury danych przejrzyste. Pracuje w doskonałej harmonii z mechanizmami równoległymi, takimi jak multiprocessing czy SCOOP. Najprostszą metodą instalacji w systemach linux jest użycie komendy pip install deap, po wcześniejszej konfiguracji środowiska Python.

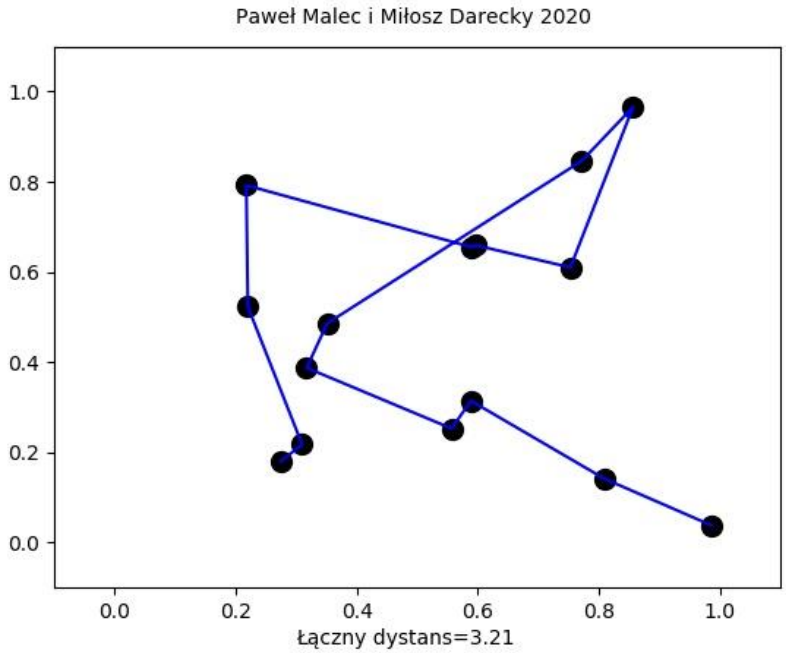
NumPy to biblioteka języka oprogramowania Python, dodaje ona obsługę dużych, wielowymiarowych tablic i macierzy, wraz z dużym zbiorem wysokopoziomowych funkcji matematycznych do obsługi tych tablic. Uzywanie NumPy w Pythonie daje funkcjonalność porównywalną z MATLAB. Najprostszą metodą instalacji tej biblioteki w systemie Linux jest komenda apt install python-numpy pod warunkiem wcześniej zainstalowanego środowiska Python.

Matplotlib to biblioteka do tworzenia wykresów dla języka oprogramowania Python i jego rozszerzenia numerycznego NumPy. Zawiera ona API pylab zaprojektowane tak aby było jak najbardziej podobne do środowiska MATLAB. Dzięki temu jest prosty do opanowania. Biblioteka ta jest dostępna na licencji przypominającej BSD. Najprostszą metodą instalacji tej biblioteki w systemie Linux jest komenda python -m pip install -U pip, python -m pip install -U matplotlib pod warunkiem wcześniej zainstalowanego środowiska Python.

openGA to otwarta biblioteka algorytmów genetycznych dla języka programowania C++. Użytkownik tej biblioteki posiada pełną kontrolę nad operacjami genetycznymi, takimi jak tworzenie rozwiązań, mutacja czy krzyżowanie. Standardowe biblioteki w języku C++ są wystarczające do obsługi tej biblioteki i nie ma potrzeby instalowania dodatkowej biblioteki do kompilacji. Bibliotekę można pobrać z linka https://github.com/Arash-codedev/openGA/archive/v1.0.3.zip

3. Zaimplementowany algorytm rozwiązujący wybrany problem optymalizacyjny

W problemie komiwojażera (Travelling salesman problem), kupiec musi przejeżdżać przez pewne miasta, ale miasta nie są takie same jak inne. Problem planowania trasy staje się bardziej skomplikowany gdy zwiększamy ilość miast oraz ich położenie pomiędzy sobą. Jeśli obliczana jest każda możliwa ścieżka, wymaga ona dużych zasobów obliczeniowych. Stosując algorytm ewolucyjny, problem komiwojażera można uznać za bardzo typowy problem.



Rysunek 1 Przykładowe wykonanie programu i prezentacja wykresu z matplotlib

Fitness i DNA: Możemy utworzyć identyfikator dla każdego miasta dzięki czemu sortowanie miast będzie przechodziło według nadanym im identyfikatorów. Dla przykładu mając pięć miast (Kraków, Warszawa, Łódź, Poznań, Wrocław) kupiec przejeżdżając trzy z miast posiada ma możliwe kombinacje tras:

Kraków Warszawa Łódź
Kraków Warszawa Wrocław
Kraków Warszawa Poznań
Kraków Łódź Wrocław
Kraków Łódź Poznań

Kraków Wrocław Poznań

Warszawa Łódź Wrocław

Warszawa Łódź Poznań

Warszawa Wrocław Poznań

Łódź Wrocław Poznań

Pojawia się nam 10 możliwości. Każde z tych możliwości może być traktowana jako sekwencja DNA, która może być wygenerowana za pomocą biblioteki numpy w języku programowania Python.

Sekwencja w naszym programie staje się prosta.

```
np.random.permutation(3)
# array([1, 2, 0])
```

Aby obliczyć fitness, wystarczy umieścić te miasta w DNA i podliczyć sumę. W zależności od długości ustalamy regułę, że im krótsza całkowita przebyta ścieżka, tym lepszy jest fitness. Do obliczania kondycji wykorzystujemy fittness0. Do obliczania celu czyli krótszej ścieżki wykorzystujemy fitness1.

```
fitness0 = 1/total_distance
fitness1 = np.exp(1/total_distance)
```

Ewolucja: Algorytm GA zastępujemy klasą GA, która posiada główne cechy jak w przykładzie poniżej.

Ewolucja następuje w trzech krokach: Przetrwanie najsprawniejszego(selection), krzyżowanie

```
class GA:
    def select(self, fitness):
    def crossover(self, parent, pop):
    def mutate(self, child):
    def evolve(self):
```

DNA(crossover), Zamiana DNA(mutation), używamy trzech funkcji Python aby je przedstawić.

Należy zwrócić uwagę na fakt, że istnieje niewielka różnica pomiędzy crossover a mutation, gdyż nie możemy po prostu zmienić punktów trasy. Jeżeli użyjemy normalnego crossover otrzymamy np.

P1=[0,1,2,3] (Tata)

P2=[3,2,1,0] (Mama)

CP=[M,T,M,T] (Krzyżowanie, M:Mama, T:Tata)

C1=[3,1,1,3] (Dziecko)

Więc C1 przechodzi przez miasto 3 dwa razy, miasto 1 dwa razy, lecz nie przechodzi przez punkty 2,0. Musimy więc zastosować mutację i krzyżowanie w inny sposób. Jeden z możliwych jest następujący:

P1=[0,1,2,3] (Tata)

CP=[_,T,_,T) (Wybierz punkt od Taty)

C1=[1,3_,_] (Najpierw wypełnij od Taty)

W tym momencie, oprócz 1 i 3 od taty pozostają nam punkty 0,2 miasta, którego rzędy będą odziedziczone od mamy. Miasta 0,2 w P2=[3,2,1,0] (Mama) znajdują się na dwóch pierwszych pozycjach więc dodamy je do DNA dziecka w tej właśnie kolejności otrzymując taki wynik jak poniżej.

C1=[1,3,2,0] (Dziecko)

W ten sposób z powodzeniem unikniemy wymienionych wcześniej problemów unikania pewnych miast (jak w przykładzie punkty 2 i 0).

```
if np.random.rand() < self.cross_rate:
        i_ = np.random.randint(0, self.pop_size, size=1)                        # Wybierz inną osobę z populacji
        cross_points = np.random.randint(0, 2, self.DNA_size).astype(np.bool)   # Wybierz punkt
crossover
        keep_city = parent[~cross_points]                                       # Znajdź numer miasta
        swap_city = pop[i_, np.isin(pop[i_].ravel(), keep_city, invert=True)]
        parent[:] = np.concatenate((keep_city, swap_city))
```

W mutacji DNA odnajdujemy dwa różne DNA a następnie zamieniamy je:

```
for point in range(self.DNA_size):
    if np.random.rand() < self.mutate_rate:
        swap_point = np.random.randint(0, self.DNA_size)
        swapA, swapB = child[point], child[swap_point]
        child[point], child[swap_point] = swapB, swapA
```

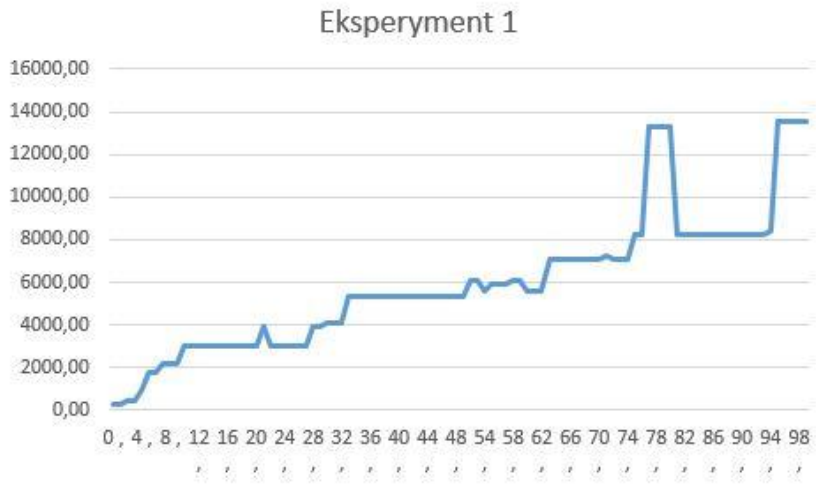
Po krótkim omówieniu najważniejszych miejsc w programie przejdźmy do wartości przyjętych przez nas do eksperymentu.

```
N_CITIES = 15        # Wielkość DNA
CROSS_RATE = 0.5     # Wskaźnik krzyżowy
MUTATE_RATE = 0.05   # Współczynnik mutacji
POP_SIZE = 100       # Ilość populacji
N_GENERATIONS = 100  # Liczba generacji
```

Wartości te pozwalały otrzymywać nam najlepsze wyniki w jak najmniejszym czasie wykonywania programu. Ilość populacji oraz generacji powinny posiadać jednakowe wartości. Zbyt mała ilość miast w parametrze N_CITIES powoduje, że program bardzo szybko odnajduje najkrótszą trasę i trudno zauważyć jego prawidłowe działanie. Wartości zostały dobrane tak aby udało nam się szybko wykonać kilka prób a jednocześnie odnalezienie trasy nie było dla algorytmu zbyt proste i możliwa była obserwacja zmian w czasie powstawania kolejnych generacji. Mutate_rate oraz Pop_size o wartościach kolejno: 0,5 oraz 0,05 pozwalały na prawidłowe odnajdywanie bez zbędnych nieprawidłowych mutacji w wyniku zbyt dużej wartości parametru.

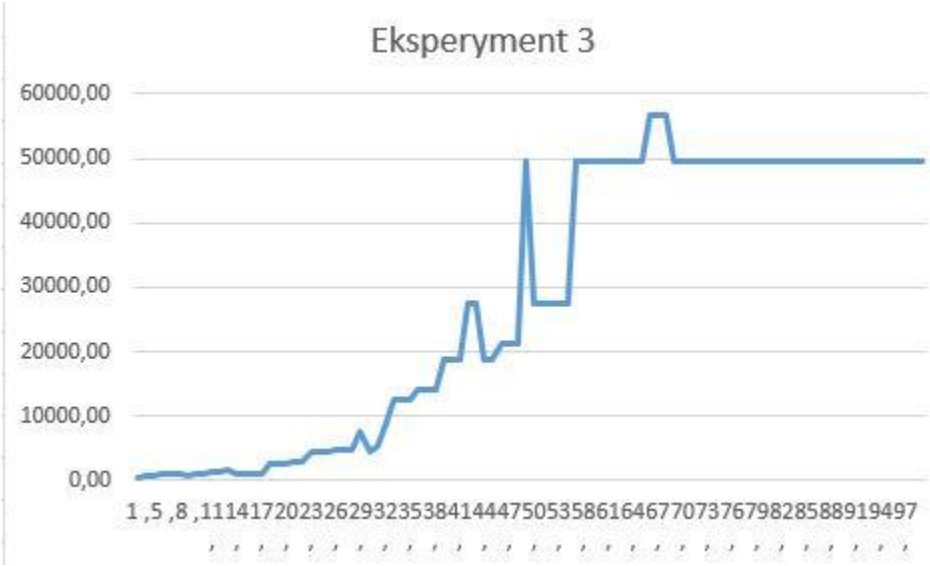
4. Wyniki eksperymentów

[Wyniki eksperymentów: skomentowane wykresy pokazujące średnie wyniki z odchyleniami standardowymi z co najmniej 10 uruchomień algorytmu (wykresy (podobnie jak w przypadku platformy Framsticks): najlepszy, najgorszy i średni fitness w populacji w trakcie trwania eksperymentu), omówienie najlepszego znalezionego rozwiązania]



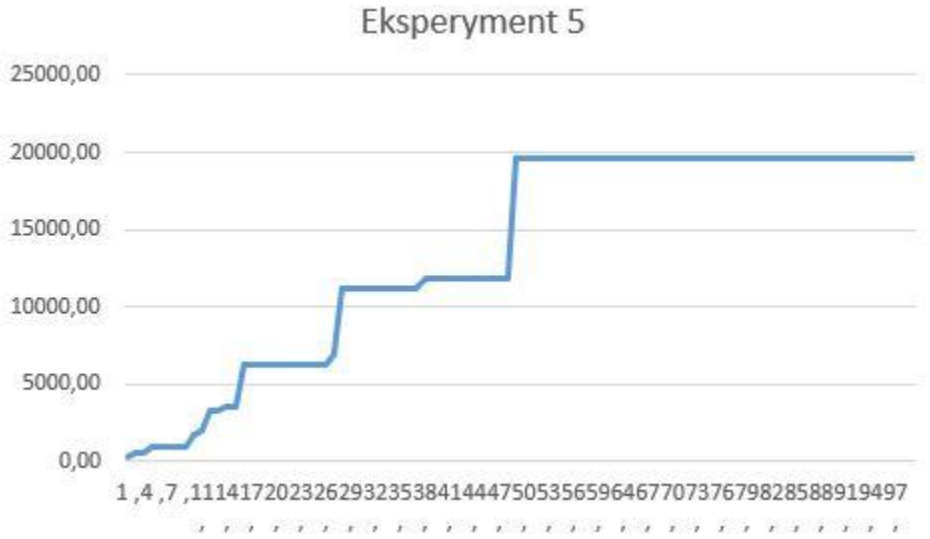
Pierwsze uruchomienie eksperymentu na naszym algorytmie pokazało, że stopniowo dążymy do jak najlepszego wyniku fitness (od 0 do 98) a problemy pojawiają się dopiero od punktu 78 gdzie następuje załamanie, jednak algorytm wrócił do wartości wyższej już przy 94 punkcie.

Eksperyment nr 2 przeszedł dużo łagodniej stopniowo trafiając w jak najlepszy fitness, problem pojawia się dopiero na końcu gdzie delikatnie trafność opadła.

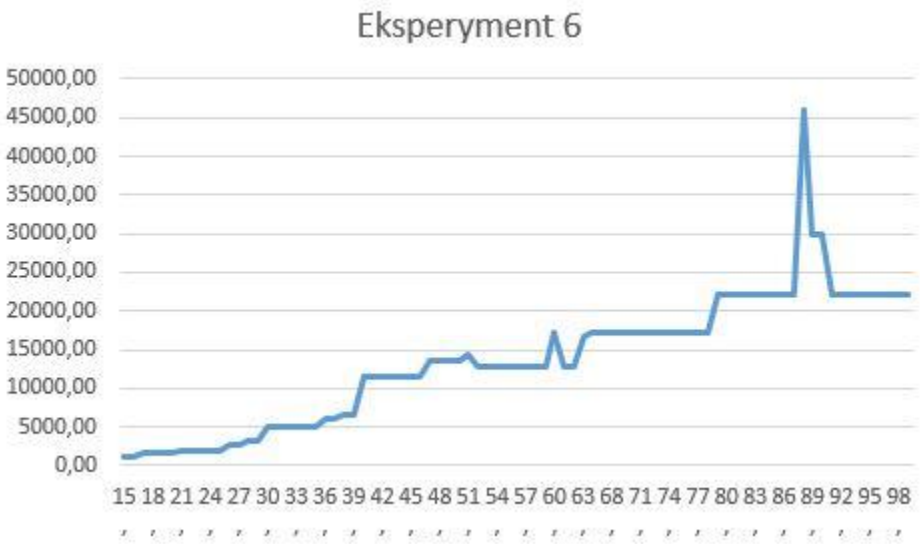


Eksperyment numer 3 nie wykazał się niczym specjalnym, dąży on prawidłowo do osiągnięcia wyniku aż do punktu 50 mając dalej chwilowe załamanie z powodu dobranego parametru mutacji.

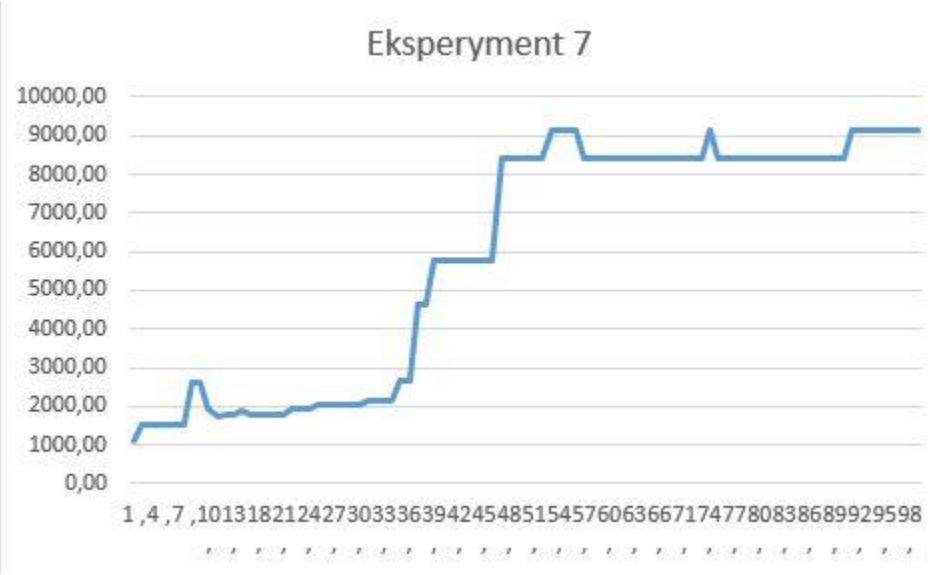
Eksperyment numer cztery okazał się trudnym dla algorytmu, przez większość przebiegów powoli trafia on na co raz lepsze wyniki fitness.



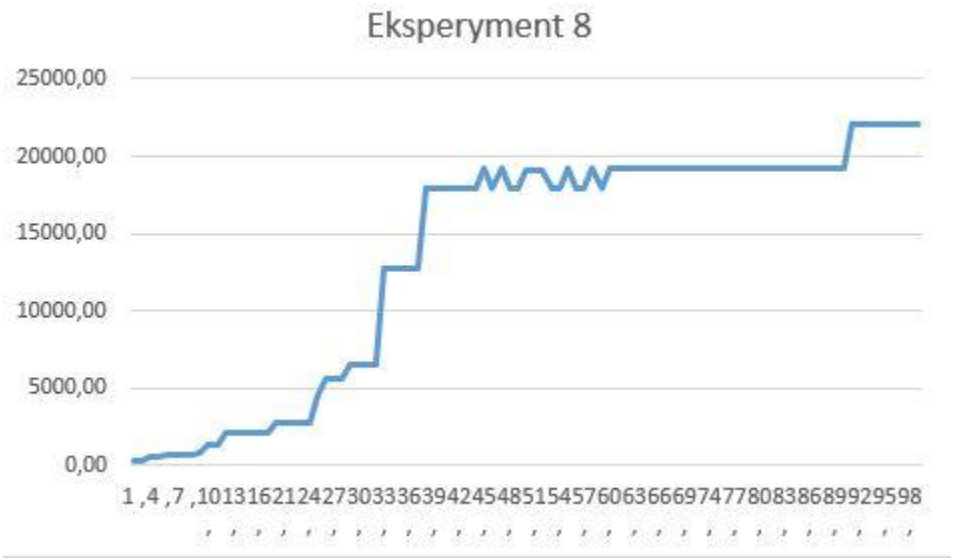
Najlepszy wynik w czasie eksperymentów, algorytm trafia już dobrze na poziomie 50 nie łamiąc się w następnych aż do końca wykonania programu z zaproponowanymi przez nas parametrami.



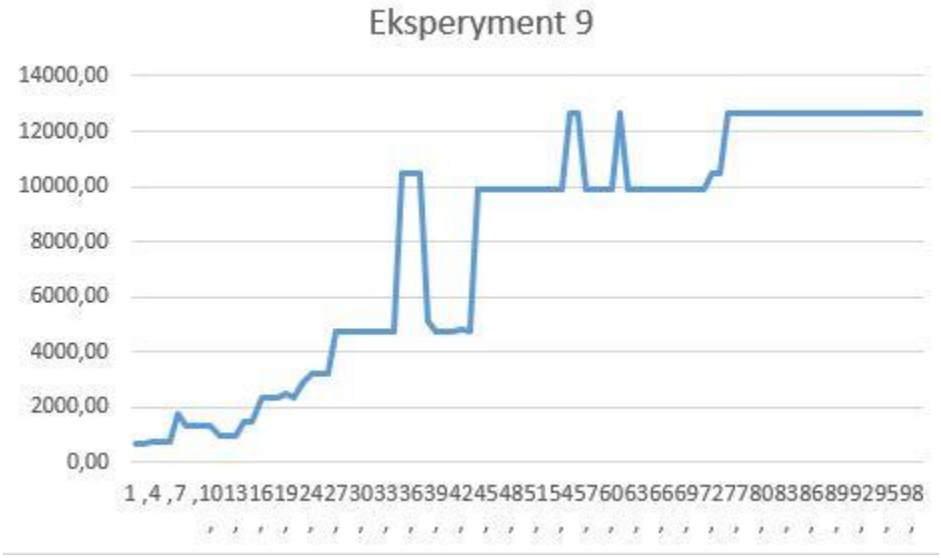
Problem załamania pod koniec wykonywania programu można wyeliminować zwiększając ilość generacji i zmniejszając poziom mutacji, jednak wydłużyłoby to czas trwania eksperymentów oraz dopasowywania parametrów.



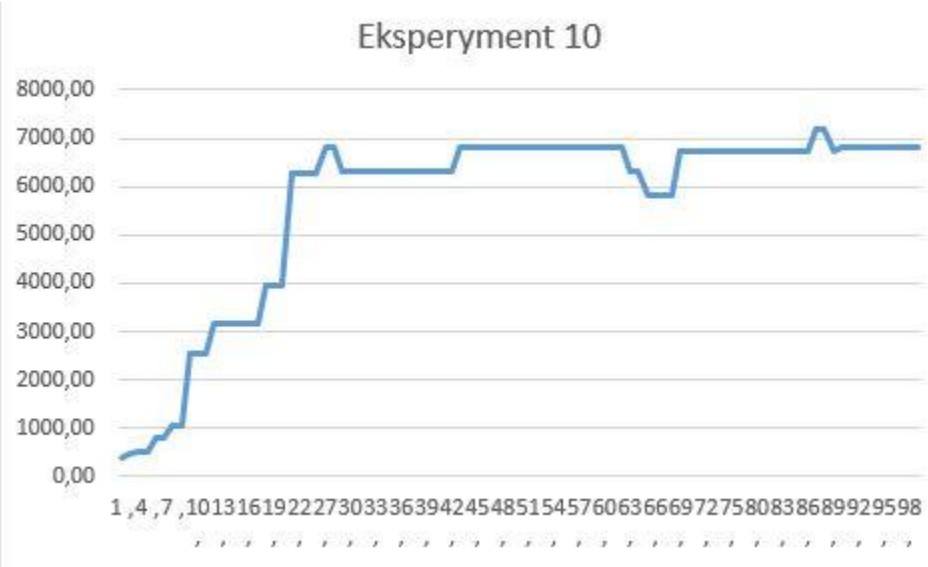
Numer 7 także prezentuje bardzo dobre działanie naszego algorytmu, ponieważ nie widzimy tutaj błędnych załamań i dążymy do jak najlepszego wyniku ustawionego wcześniej w kodzie programu.



Jeden z najlepszych wyników, ponieważ stopniowo zwiększa się ilość trafień chwilowo łamiąc się delikatnie dopiero przy punktach od 42 do 60 wychodząc z lepszym wynikiem w dalszym pomiarze.



Algorytm trafił w punkcie 39 na pewne problemy jednak wyszedł z nich już po 45 przebiegu dalej trafiając w lepsze wyznaczenie trasy pomiędzy miastami.



Algorytm trafił prawdopodobnie na prostą ścieżkę i poradził sobie z nią bardzo szybko później lawirując delikatnie pomiędzy nieznacznymi wartościami.

5. Podsumowanie

Poprzednie projekty pozwoliły nam zaprzyjaźnić się z tworzeniem sztucznego życia (w Framsticks) a następnie tworzenia systemów eksperckich przyswajając podstawowe koncepcje z nimi związane (W BayEx oraz potem w Clips).

W tym projekcie poszerzyliśmy znacznie naszą wiedzę odnośnie tworzenia własnych algorytmów ewolucyjnych. Pojawiły się także problemy znacznie bardziej złożone jak np. w naszym przypadku wzięcie pod uwagę każdego miasta przy krzyżowaniu DNA. Po chwili namysłu jednak udało się go wyeliminować. Nie mały problem pojawił się także w doborze odpowiednich parametrów mutacji oraz ilości generacji, które zostały poprawione w czasie kilku dni testowania działającego już algorytmu.

Język programowania Python okazał się bardzo przyjemny w użytkowaniu w naszym przypadku i bardzo prosty w instalacji na systemie Linux. Gotowy program można uruchomić prostą komendą w terminalu: python3 travel.py. Natomiast tworzenie programu można wykonać w naszym ulubionym edytorze bądź IDLE(Środowisko programistyczne dla Python). W naszym przypadku był to sprawdzony już notepad++. Możliwe jest nawet używanie nano w terminalu.

Program po dopracowaniu pokazuje przede wszystkim graficznie wykonywane operacje co ułatwia zrozumienie jak działa i z jakim problemem się boryka w trakcie wykonywania. Oczywiście możliwa jest także obserwacja w terminalu powstawania generacji i zmian w dopasowaniu fitness. Nie działają jednak one dobrze na wyobraźnię użytkownika. Matplotlib okazał się przydatny i prosty w konfiguracji, pokazuje także aktualne najlepsze dopasowanie poza rozmieszczeniem miast oraz aktualnie ułożoną trasę przez punkty. Jednak najbardziej potrzebnym w naszym projekcie był numpy, który wykonuje wszystkie złożone obliczenia i dzięki niemu możliwe jest sprawne ustalanie trasy.

Algorytm mógłby zostać także udoskonalony w przyszłości o dodatkowe GUI(powłokę graficzną) aby łatwo zmieniać w nim parametry i obserwować wyniki tych zmian. Bardzo ułatwiłoby to możliwość usprawnienia algorytmu w określonym przez nas celu i nie wymagałoby zmian bezpośrednio w notepad++ oraz kolejnym uruchamianiu programu w teminalu. Nie jest to jednak rzecz wymagana ale warto o niej wspomnieć a sam Python jest bardzo przyjazny w tworzeniu takich rzeczy i poradzą sobie z tym nawet nowicjusze chcący poszerzyć swoją wiedzę.

Bibliografia

1. http://zeszyty-naukowe.wwsi.edu.pl/zeszyty/zeszyt1/Algorytmy_Ewolucyjne_I_Ich_Zastosowania.pdf

2. http://edu.pjwstk.edu.pl/wyklady/nai/scb/wyklad10/w10.htm

3. https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_travelling_salesman_problem.htm

4. https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6

5. https://matplotlib.org/tutorials/introductory/pyplot.html

6. https://numpy.org/devdocs/user/quickstart.html

7. https://towardsdatascience.com/an-extensible-evolutionary-algorithm-example-in-python-7372c56a557b