JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 11 (2019) 137-144
Wysłane: 2019-02-16
Przyjęte: 2019-02-23

Porównanie wydajności języka skryptowego oraz kompilowanego na podstawie działania algorytmu genetycznego

Filip Dzikowski*
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Celem artykułu jest porównanie wydajności wybranych języków programowania (Python, C) - poprzez pomiar czasu działania oraz wykorzystania zasobów komputera - algorytmu genetycznego dla zadanych parametrów, a następnie ocena czy język skryptowy może być porównywalny pod względem szybkości z językiem kompilowanym. Do tego celu zaimplementowany został algorytm genetyczny w każdym z wymienionych języków, a następnie przeprowadzone zostały testy, których wyniki stanowiły podstawę ostatecznej oceny ich wydajności oraz dowód, że język skryptowy może osiągać czasy działania porównywalne z językiem kompilowanym.

Słowa kluczowe: Python; C; wydajność

*Autor do korespondencji.

Adres e-mail: filip.dzikowski5@gmail.com

Comparison of the performance of scripting and compiled languages based on the operation of the genetic algorithm

Filip Dzikowski*
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The aim of this work was to compare the performance of selected programming languages (Python, C) by measuring the time of operation and use of computer resources of the genetic algorithm for given parameters, and then assessing whether the scripting language can be comparable in terms of speed with the compiled language. For this purpose, a genetic algorithm has been implemented in each of these languages and test scenarios were developed. The results form the basis for the final evaluation of the performance of the presented languages and proof that the scripting language can achieve operating times comparable to the compiled language.

Keywords: Python; C; efficiency

*Corresponding author.

E-mail address: filip.dzikowski5@gmail.com

1. Wstęp

Początek dwudziestego pierwszego wieku to okres intensywnego rozwoju technologii, a tym samym systemów informatycznych, których rozmiary często przekraczają kilka milionów linii kodu. Utrzymanie takiej ilości kodu oraz dostosowanie jej do przyszłego rozwoju przy zachowaniu wydajności zapewniającej swobodę użytkowania stanowi duże wyzwanie. Podczas projektowania nowego systemu należy zdecydować jakie priorytety powinny zostać przyjęte w systemie – rozszerzalność i utrzymanie, szybkość działania czy może minimalne zużycie dostępnych zasobów. W zależności od ich ustawienia w dużej mierze zależeć będzie wybór technologii, w tym języka programowania w którym system zostanie napisany. Jeżeli priorytetem będzie szybkość działania pod uwagę powinny być brane języki niskiego poziomu, w pełni kompilowane o statycznej definicji typów jak na przykład język C, natomiast jeżeli na pierwszym miejscu znajdzie się rozszerzalność i utrzymanie, języki o prostej i intuicyjnej składni zyskuję przewagę, np. Python. Czasami jednak wybranie głównego priorytetu jest bardzo trudne i wszystkie wspomniane cechy systemu powinny zostać wzięte po uwagę. Czy istnieje zatem sposób na połączenie intuicyjności i możliwości języka skryptowego z szybkością języka kompilowanego?

Niniejszy artykuł skupia się na porównaniu przedstawicieli obu wspomnianych języków (Python, C) pod kątem szybkości wykonywania obliczeń oraz zużycia zasobów komputera, a także pokazuje metody dzięki którym możliwe jest znalezienie odpowiedzi na pytanie czy język skryptowy może osiągać czasy działania porównywalne z językiem kompilowanym. Przeprowadzone testy wraz z wynikami mogą stanowić silną podstawę do dyskusji podczas rozważań nad językiem programowania w którym projekt/system/program powinien zostać zaimplementowany.

2. Przegląd literatury

2.1. Język C

Język C został zaprojektowany w 1972, przez Dennisa Ritchiego i od tamtej pory jest stale rozwijany. Został znormalizowany przez American National Standards Institute (ANSI) w 1989 roku. Najnowsza stabilna wersja o oznaczeniu C18, została wydana w czerwcu 2018 roku. C jest językiem ogólnego przeznaczenia, wspierającym

programowanie strukturalne (proceduralne), leksykalny zapis zmiennych oraz rekurencję. Ponieważ jest typowany statycznie już w procesie kompilacji zapobiega błędom np. przy przekazywaniu argumentów do funkcji. Z założenia, C dostarcza konstrukcje, które efektywnie odwzorowują typowe instrukcje maszynowe, a zatem znalazł trwałe zastosowanie w aplikacjach, które wcześniej były kodowane w języku maszynowym (assembler), w tym w systemach operacyjnych, a także w różnych aplikacjach dla komputerów, od superkomputerów po systemy wbudowane [1].

Został zaprojektowany do kompilacji za pomocą stosunkowo prostego kompilatora, aby zapewnić niskopoziomowy dostęp do pamięci, liczne konstrukcje językowe, które skutecznie mapują instrukcje maszynowe oraz zapewnić minimalną potrzebę wsparcia w czasie pracy. Język został zaprojektowany, aby zachęcić do programowania na wielu platformach. Zgodny z normami program C, napisany z myślą o przenośności może być skompilowany dla bardzo różnorodnych platform komputerowych i systemów operacyjnych z niewielkimi zmianami w kodzie źródłowym [2].

2.2. Język Python

Zgodnie z badaniami przeprowadzonymi przez Instytut Inżynierów Elektryków i Elektroników (IEEE) w latach 2017 oraz 2018 i opublikowanych w czasopiśmie naukowym IEEE Spectrum, język Python uplasował się na pierwszym miejscu najczęściej używanych języków programowania [1, 3].

Python został stworzony we wczesnych latach dziewięćdziesiątych przez Guido van Rossuma, w Stichting Mathematisch Centrum (CWI) w Holandii jako następca języka zwanego ABC. Jest on jego głównym autorem, jednak od wersji 1.6.1. jest dystrybuowany na licencji GNU (GPL). W związku z powyższym wszystkie źródła języka Python dostępne są dla każdego na stronie projektu na serwisie GitHub [15]. Aktualnie dostępną stabilną wersją języka jest wersja 3.7.2. Ponieważ Python jest koncepcją języka, doczekał się wielu implementacji. Najpopularniejszą z nich i omawianą w niniejszej pracy jest implementacja w języku C – CPython, jednak znaleźć można również implementację w języku Java (Jython), która z powodzeniem tłumaczy kod Pythona do kodu bajtowego wirtualnej maszyny JVM oraz implementację przygotowaną z myślą o programistach .NET – IronPython [4].

2.3. Porównanie wydajności języków C oraz Python w literaturze

Porównanie zostanie rozpoczęte pracą opracowaną przez czterech autorów: Muhammada Ateeq, Hina Habib, Adnana Umer, Muzammila Ul Rehman, pt. *C++ or Python? Which One to Begin With: A Learners Perspective* [5]. W tytule powyższego artykułu znajduje się „C++", jednak wgląd do pracy pozwala stwierdzić, że w porównaniu nie są wykorzystywane specyficzne dla wspomnianego języka mechanizmy, jak klasy czy szablony. Mając to na uwadze wraz z faktem, że zachowanie zgodności na poziomie kodu źródłowego z językiem C, jest podstawowym wymogiem dla kolejnych standardów C++ oraz, że biblioteka standardowa pokrywa się w znacznej części z biblioteką języka C, przykład ten może być bez przeszkód wykorzystany do przedstawienia, który z porównywanych w niniejszej pracy języków jest lepszym wyborem dla początkującego programisty. Autorzy postanowili przebadać studentów po dwóch semestrach nauczania w których poznawali oni wymienione języki (I semestr – Python, II semestr – C++). W przeprowadzonej ankiety, która skupiała się na intuicyjności i prostocie pisania kodu, biorąc po uwagę takie aspekty jak prostota, elastyczność, narzędzia do debuggowania czy dostępność modułów, badacze zauważyli, że studenci byli bardziej usatysfakcjonowani Pythonem, a jego funkcje, jak instrukcje warunkowe, czy pętle, były prostsze w porównaniu do C++.

Następnym przykładem będzie praca autorstwa L. Dobrescu: *Replacing ANSI C with other modern programming languages* [6], również rozprawiająca o zaletach oraz wadach zastąpienia ustandaryzowanego i wydajnego języka C, językiem zorientowanym obiektowo jak Python lub Java. W artykule autor punktuje obiektywne uwagi, z których pierwszą informacją na temat Pythona jest jego prostota, składania, interpreter oraz ogromna społeczność, które zachęcają do programowania. Użytkownika nie czeka tutaj żmudny proces kompilacji i linkowania. Użycie interpretera pozwala otrzymać wynik działania programu od razu. Autor zaznacza, że jest to bardzo dobry język programowania do wyboru jako pierwszy oferujący bardzo duże możliwości wraz z bardzo dokładnymi komunikatami błędów, które dla początkującego programisty są ważną informacją.

W kolejnej pracy pt. *An empirical comparison of seven programming languages* [7], autor Lutz Prechelt dokonał porównania siedmiu języków programowania: C, C++, Java, Perl, Python, Rexx, i Tcl. Języki te mogą być podzielone na dwie główne grupy:

1) języki kompilowane – C, C++, Java,
2) języki skryptowe – Perl, Python, Rexx, Tcl,

Z każdej z powyższych grup wybrani zostali przedstawiciele, na których skupia się temat niniejszej pracy - C oraz Python. Do ich porównania Autor publikacji wykorzystał problem nazywany *Phonecode*. Zasada działania algorytmu była bardzo prosta. Program wczytywał słownik 73,113 słów do pamięci, a następnie odczytywał z drugiego pliku numery telefonów i tłumaczył je na słowa wykorzystując poprzednio wczytany słownik. Rolę drugiego pliku odgrywały dwa różne zbiory danych – zawierający 1,000 numerów telefonów oraz drugi pusty plik, który służył jedynie do zmierzenia czasu wczytywania słownika. Na podstawie przeprowadzonych badań stwierdzono, że wszystkie algorytmy działają z porównywalną niezawodnością, natomiast języki skryptowe, zużywają o około połowę więcej pamięci operacyjnej w porównaniu do C/C++ (Java, która jest językiem typowanym statycznie, jest kompilowana do kodu bajtowego JVM, powodując tym samym, że zużywa ona prawie czterokrotnie więcej pamięci niż C/C++). Nie jest również zaskoczeniem, że czas wczytania słownika zawierającego ponad 70 tysięcy

rekordów, zajął językom C/C++ dziesięciokrotnie mniej czasu niż językom skryptowym. Braki w wydajności w prezentowanych przykładach, języki skryptowe nadrabiają natomiast szybkością oraz prostotą implementacji [19]. Napisanie algorytmów w językach skryptowych wymagało o średnio prawie o połowę mniej linii kodu w porównaniu z językami kompilowanymi.

Praca Li Jun oraz Li Ling, pt. *Comparative research on Python speed optimization strategies* [8], rzuca pierwsze światło na sposoby optymalizacji szybkości działania Pythona. Metody przyspieszenia wykonywania kodu programu zostały przez Autorów przedstawione w klarowny sposób oraz skategoryzowane w dwóch grupach – *optymalizacji, które w znikomym stopniu wpłynęły na czas wykonywania programu* oraz *optymalizacji, które znacząco wpłynęły na czas wykonywania się programu* (których czasy działania w porównaniu do takiego samego kodu w języku C są bardzo obiecujące). W pracy autorstwa pary Jun oraz Ling zostało również zawarte bardzo ciekawe porównanie szybkości wykonywania obliczeń z wykorzystaniem różnych wersji języka Python, które zostały zestawione z czasem wykonania tych samych operacji z wykorzystaniem języka C. Wyniki uzyskane podczas przeprowadzania tego testu jasno pokazują jak ważne jest posiadanie najbardziej aktualnej wersji interpretera języka Python oraz jak słabo wypada (niezależnie od wersji) w porównaniu do języka kompilowanego. Warto zaznaczyć, że czasy uzyskane podczas wykonywania optymalizacji z grupy znacząco poprawiających wydajność, testowane były z wykorzystaniem najwolniejszego interpretera języka Python ze wszystkich porównywanych. Wydaje się być logicznym, że czasy te mogły by być jeszcze lepsze przy wykorzystaniu najnowszego (bądź najlepszego z testowanych) interpretera.

Kolejną pracą, zestawiającą ze sobą bezpośrednio języki Python oraz C, jest *Program performance test based on different computing environment* [9], autorstwa Hailong Zang oraz Jun Nie. Porównanie dokonywane jest na podstawie czasu działania algorytmu obliczającego odległość pomiędzy punktami na sferze. Podobnie do poprzedniego artykułu przedstawionego w poprzednim akapicie, Autorzy wykorzystują implementację CPython, która pozwala na dodawanie kodu napisanego w języku C. W tym przypadku jednak, porównanie zostaje dokonane również po optymalizacji obliczeń napisanych w C, z wykorzystaniem biblioteki OpenMP (Open Multi-Processing). Praca skupia się niestety tylko na jednej metodzie optymalizacji (po jednej dla każdego języka), jednak wyniki uzyskane przez badaczy pozwalają zobaczyć, jak wybrana metoda przyspieszenia działania języka Python wpływa na jego wydajność. W krótkim punkcie na początku artykułu można przeczytać dlaczego do porównania zostały wybrane akurat te języki – prostota i elegancja (Python) kontra szybkość (C).

2.4. Algorytm genetyczny

Od czasu przedstawienia koncepcji przełożenia zachowań panujących w naturze, na język komputerowy, nastąpił wykładniczy wzrost prac badawczych w tej dziedzinie i na dzień dzisiejszy można stwierdzić, że rozwój algorytmów genetycznych osiągnął swego rodzaju dojrzałość. Było to możliwe zarówno dzięki wspomnianym badaniom oraz znacznie malejącym kosztom szybkich i tanich komputerów. Problemy, które kiedyś rozważane były w kategoriach niemożliwych do wyznaczenia, w dzisiejszych czasach, z punktu widzenia złożoności obliczeń nie stanowią już wyzwania [16]. Dzięki temu, złożone problemy, które wymagają jednoczesnego rozwiązywania mogą być wyznaczone z użyciem algorytmu genetycznego. Warto również zaznaczyć, że w przypadku wspomnianej rodziny algorytmów uzyskana optymalizacja jest przekształcana z pokolenia na pokolenie bez ścisłego formułowania matematycznego, które występuje w tradycyjnych gradientowych sposobach optymalizacji [10].

Algorytm genetyczny inspirowany jest naturalną selekcją, tj. biologicznym procesem w którym silniejszy osobnik jest faworytem w niesprzyjającym otoczeniu. Algorytm genetyczny używa bezpośredniej analogii do procesu ewolucji. Zakłada, że potencjalnym rozwiązaniem problemu może być jednostka, która jest reprezentowana przez zbiór cech. Cechy takie nazywane są genotypem osobnika (chromosomu), który najczęściej przedstawiany jest w formie ciągu binarnego. Bardzo ważną cechą z punktu widzenia algorytmu genetycznego jest funkcja przystosowania osobnika, która jest miarą jakości rozwiązania.

Najbardziej podstawowy zapis algorytmu genetycznego sprowadza się do następującej procedury:

1) Wybierz początkową populację osobników – chromosomów,
2) Genotypy wybranych osobników poddawane są operatorom ewolucyjnym (mutacja, krzyżowanie),
3) Oblicz funkcję przystosowania dla każdego osobnika,
4) Dokonaj selekcji osobników na podstawie wybranej metody selekcji,
5) Zastąp osobniki o najmniejszej wartości funkcji przystosowania nowymi osobnikami.

Podczas projektowania działania algorytmu genetycznego niezbędne jest ustalenie następujących rzeczy:

- genotypu jako reprezentanta wyniku,
- funkcji przystosowania,
- sposobu selekcji osobników,
- definicji operatorów ewolucyjnych.

Kodowanie

Fundamentalną rzeczą w algorytmie genetycznym jest sposób kodowania wartości, które reprezentują rozwiązanie optymalizowanego problemu. Mechanizm ten polega na naturze wykorzystywanych zmiennych, na przykład przy próbie wyznaczenia rozwiązania dla optymalnych przepływów w transporcie cieczy, zmienne (przepływy w różnych kanałach) przyjmują wartości ciągłe, natomiast dla problemu komiwojażera zmiennymi są wielkości binarne reprezentujące uwzględnienie lub wykluczenie krawędzi w obwodzie hamiltonowskim. W każdym wypadku mechanizm kodowania powinien zapewniać jednoznaczność dekodowania tzn. każdemu genotypowi (np. każdej

kombinacji bitów) musi odpowiadać punkt z przestrzeni stanów, czyli rozwiązanie zadania [11].

Funkcja przystosowania

Funkcja przystosowania, nazywana również funkcją celu lub dopasowania, używana jest w algorytmach genetycznych do obliczenia wartości rozwiązania dla danego osobnika w każdej iteracji. Jest ona podstawowym źródłem informującym o jakości każdego chromosomu w przestrzeni rozwiązań [11].

Krzyżowanie

Operator ten polega na łączeniu w pary losowych chromosomów z danej populacji w celu przeprowadzenia na nich krzyżowania, tj. wybór losowego miejsca (bitu) w chromosomie jednego z rodziców, a następnie wymiana kodu genetycznego pomiędzy wybranymi osobnikami zaczynając od wylosowanego punktu. Czy dojdzie do skrzyżowania zależy od ustalonego w drodze losowania prawdopodobieństwa [17].

Mutacja

Kolejnym operatorem genetycznym jest mutacja. Mutacja bitu polega na jego zamianie z 0 na 1 oraz w drugą stronę w przypadku chromosomów kodowanych binarnie. Jeżeli genotyp danego osobnika zakodowany jest w postaci liczb całkowitych, stosuje się permutacje, natomiast w przypadku liczb rzeczywistych wprowadza się do losowo wybranych genów niedeterministyczne zmiany o wybranym rozkładzie. Tak jak w przypadku krzyżowania, szansa, że akurat dany bit będzie poddany mutacji określana jest losowo wygenerowaną wartością. Każdy z bitów w ciągu binarnym mutuje z niezależnym prawdopodobieństwem, tj. prawdopodobieństwo mutacji danego bitu, nie wpływa na prawdopodobieństwo mutacji kolejnych bitów [11].

3. Oprogramowanie testowe

Do porównania wydajności obydwu języków i ustalenia czy języki skryptowe mogą być porównywalnie szybkie jak języki kompilowane wykorzystany zostanie algorytm genetyczny, który przy pomocy figur geometrycznych (trójkątów), będzie się starał odtworzyć podany na wejście znormalizowany obraz. Na podstawie czasu działania algorytmu (w różnych wersjach - optymalizacja) dla wybranej liczby osobników oraz sprecyzowanej liczby pokoleń, a także poprzez pomiar zużycia zasobów (pamięci) dokonana zostanie analiza, która ostatecznie pozwoli ocenić, czy wydajność skryptu może osiągnąć wartości programu napisanego w języku C.

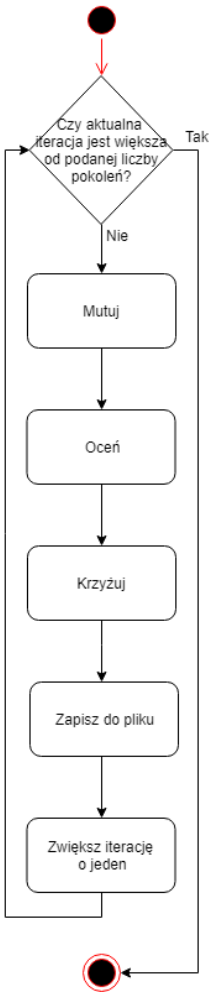
3.1. Zasada działania

Zasada działania algorytmu w pierwszym kroku polega na wczytaniu do pamięci operacyjnej – tablicy o rozmiarze iloczynu wysokości oraz szerokości pliku – obrazu w postaci surowych bajtów (plik w formacie .raw/.data), który posłuży jako wzorzec dla algorytmu genetycznego. Następnie dla każdego osobnika, których liczba może być dowolnie zdefiniowana zostanie utworzona tablica o identycznym rozmiarze zainicjalizowana zerami. Ponieważ algorytm jako metodę selekcji wykorzystuje metodę rankingową na samym początku zostaną zainicjalizowane dwie kolejne tablice – pierwsza przechowująca najlepsze chromosomy, wytypowane w drodze rankingu, druga przechowująca osobniki wygenerowane na drodze krzyżowania. Ponieważ obraz podany na wejście jest w skali szarości (każdy bajt ma wartość z przedziału [0,255]), funkcją przystosowania użytą w celu selekcji najlepszych osobników będzie suma kwadratów różnic, a więc im mniejsza wartość sumy tym lepszy dany osobnik. Dąży się więc do minimalizacji funkcji celu. Mutacja w przypadku tego konkretnego algorytmu będzie się odbywała poprzez uśrednianie wartości danego bajtu z losowo wygenerowaną wartością, również z wcześniej wspomnianego przedziału – będzie to zatem proste przyciemnianie lub rozjaśnianie danego piksela. Krzyżowanie wykorzystywane podczas generowania nowego pokolenia jest prostą, jednak bardzo skuteczną funkcją. Skupia się ona na kopiowaniu najlepszych osobników aż do czasu zapełnienia następnej populacji. Przy wykorzystaniu tej metody blokowana jest możliwość reprodukcji dla osobników najgorzej przystosowanych, jednak na potrzeby badań wykonywanych podczas niniejszej pracy jest to wystarczające rozwiązanie. Całość jest powtarzana do czasu osiągnięcia podanej liczby pokoleń, zapisując przy okazji na dysk najlepiej przystosowanego osobnika z danego pokolenia w celu umożliwienia podejrzenia jak radzi sobie algorytm. Działanie algorytmu można przedstawić przy pomocy diagramu (Rys. 1.).

Całość oprogramowania zawiera się w dwóch plikach źródłowych. Jeden napisany w języku Python o rozszerzeniu *.py*, drugi napisany w języku C z rozszerzeniem *.c*. Ponieważ są to języki skrajnie różne, podczas implementacji algorytmu w języku kompilowanym w celu emulacji obiektowości, wykorzystana została technika *C-structure subtyping*, które pozwala na taki zabieg w języku C, w celu jak najbardziej dokładnego odwzorowania algorytmu w dwóch plikach. Jako obraz wejściowy wykorzystany został znormalizowany obraz autorstwa Leonarda da Vinci pt. *Portret damy z gronostajem,* potocznie noszący nazwę *Damy z łasiczką.*

Każdy z algorytmów składa się z kilku głównych części (ponieważ różnica pomiędzy klasą, a strukturą polega tylko na różnych prawach dostępu do zmiennych, w dalszej części pracy, terminy *struktura* oraz *klasa* będą używane wymiennie w odniesieniu do obydwu języków):

- *struktura reprezentująca pokolenie* – jest to główna część programu, zawierająca w sobie tablice przechowujące dane każdego chromosomu, najlepsze osobniki, a także metody wykorzystywane przy mutacji, krzyżowaniu oraz ocenianiu,
- *mutacja, krzyżowanie, ocena* – jak wspomniano w poprzednim punkcie każdy algorytm składa się z tych samych wyrażeń wykorzystanych przy implementacji wspomnianych funkcji, przetłumaczonych na semantykę danego języka,



Rys. 1. Schemat działania wykorzystywanego algorytmu genetycznego

- *struktura reprezentująca osobnika* – w programach została również zaimplementowana struktura definiująca dany chromosom, przechowująca wartość jego funkcji dopasowania, a także jej definicję,
- *funkcja dopasowania* – zdefiniowana jako metoda w klasie reprezentującej osobnika,
- *struktura opisująca trójkąt* – służy ona do generowania nowego osobnika podczas procesu mutacji, który w prezentowanym algorytmie przyjmuje postać figury geometrycznej - trójkąta.

3.2. Metody optymalizacji języka Python

Ze względu na wygodę używania języka Python oraz ogromnej społeczności, stworzone zostało bardzo wiele metod oraz sposobów optymalizacji. W niniejszym podpunkcie przedstawione zostaną techniki optymalizacji, które zostaną wykorzystane podczas testów oprogramowania.

1) Technologia Just-In-Time

Metoda ta jest bardzo dobrze znana wszystkim użytkownikom języka JAVA. Polega ona na kompilowaniu kodu do kodu maszynowego podczas działania programu, bezpośrednio przed jego wykonaniem w przeciwieństwie do standardowej ścieżki wykonania, gdzie kod najpierw jest tłumaczony do kodu bajtowego maszyny danego języka. Python również posiada swoją implementację kompilatora JIT. Do roku 2012, jedynym dostępnym rozwiązaniem był kompilator *Psyco* [12], który przestał być rozwijany. Na całe szczęście w jego miejsce pojawił się w odczuciu autora niniejszej pracy dużo lepszy kompilator *Just-In-Time*, który w przeciwieństwie do *Psyco* (który był importowany w postaci modułu), jest całkowicie oddzielną implementacją Pythona, w pełni kompatybilną z wykorzystywanym w niniejszej pracy CPythonem – *PyPy* [13]. Jest to bardzo dobre rozwiązanie, które nie wymaga żadnego przekształcania kodu, który został napisany dla danej wersji standardowej implementacji Pythona, będąc dodatkowo w pełni przenośnym rozwiązaniem.

2) Rozszerzenia w języku C

W każdym algorytmie znajdują się wąskie gardła, które skutecznie spowalniają pracę programu. Dzięki możliwości rozszerzania języka Python, z wykorzystaniem języka C [14], mogą one zostać zaimplementowane w języku kompilowanym, korzystając z udostępnionego API implementacji języka Python, a następnie dołączone do projektu jak zwykły moduł. Niestety to rozwiązanie jest bardzo pracochłonne i podatne na błędy. Programista musi być zaznajomiony z wewnętrznymi mechanizmami języka, jak na przykład zliczanie referencji, które błędne zaimplementowane może prowadzić do wycieków pamięci i niezdefiniowanego zachowania programu, które może skutkować jego awaryjnym zakończeniem.

3.3. Metody optymalizacji języka C

Głównym sposobem na zoptymalizowanie kodu napisanego w języku C jest jego ponowne napisanie używając coraz bardziej wyrafinowanych technik, jak wstawki kodu asemblera [19]. By uniknąć niepotrzebnego zaciemniana kodu, do testów zostaną wykorzystane optymalizacje dostępne w kompilatorze. Do kompilacji kodu źródłowego napisanego w języku C, przedstawiany algorytm genetyczny wykorzystuje kompilator *GCC 8.2.0.*. Ponieważ kompilacja przeprowadzana jest na systemie Windows, wykorzystane zostanie środowisko MinGW.

GCC udostępnia cztery podstawowe rodzaje optymalizacji:

- *O1* – niezależna od architektury wysoko poziomowa optymalizacja obliczeń,
- *O2* – zawiera optymalizacje wykonywane w *O1*, a także specyficzne optymalizacje dla danej architektury procesora,
- *O3* – tak zwana *agresywna optymalizacja,* nacisk zostaje postawiony na całkowitą maksymalizację szybkości obliczeń.

4. Testy

Testowanie opracowanego oprogramowania zostało przeprowadzone na maszynie o procesorze Intel® Core™ i7-3517U, taktowanym zegarem o częstotliwości 1.9 GHz w trybie normalnym oraz możliwości taktowania na poziomie 2.4 GHz w trybie boost, posiadającej 12 GB pamięci RAM, działającej pod kontrolą systemu operacyjnego Windows 10 (64-bit).

Wnioski wyciągnięte na podstawie wyników otrzymanych z przeprowadzonym testów pozwoliły ocenić stopień wydajności poszczególnych języków oraz zdecydować czy język skryptowy może być porównywalny pod względem szybkości działania z językiem kompilowanym.

4.1. Badanie wydajności

Badanie wydajności zostało przeprowadzone na podstawie wcześniej zdefiniowanych scenariuszy. Dla poszczególnego języka obejmowały one przetestowanie czasu działania algorytmów oraz wykorzystywanych zasobów pamięci bez zastosowania optymalizacji oraz z jej zastosowaniem, dla różnych wartości osobników w populacji oraz różnych wartości najlepszych osobników dla stałej liczby pokoleń, która pozwoli wyraźnie zauważyć działanie algorytmu i wynosić będzie 1000.

1) *Scenariusz 1* - pierwszy scenariusz zakłada przetestowanie działania algorytmu dla 300 osobników w populacji oraz dla liczby najlepszych osobników wynoszącej 2,
2) *Scenariusz 2* – w tym przypadku liczba osobników w populacji wynosi 500, reszta zgodnie z poprzednim scenariuszem,
3) *Scenariusz 3* – liczba osobników zostanie podniesiona do 1000, wartość liczby najlepszych osobników również nie ulega zmianie,
4) *Scenariusz 4* – scenariusz czwarty jako pierwszy zmienia liczbę najlepszych osobników do 10, zmniejszając również liczbę chromosomów w pokoleniu do 300,
5) *Scenariusz 5* – liczba najbardziej przystosowanych osobników, którzy zostaną poddani krzyżowaniu wzrasta do 50, przy zachowaniu stałej liczby populacji,
6) *Scenariusz 6* – liczba najlepszych osobników: 100, osobników w populacji: 300.

Powyższe scenariusze zostały wykonane ośmiokrotnie: cztery razy w przypadku języka C – bez optymalizacji i cztery z optymalizacjami oraz analogicznie dla języka Python.

4.2. Język Python – wyniki testów

Wyniki uzyskane podczas testów dla języka Python zostały przedstawione w dwóch kolejnych podpunktach i przeprowadzone zostały przy wykorzystaniu najnowszego dostępnego interpretera języka w wersji 3.7.2. – w przypadku testów przeprowadzanych bez optymalizacji oraz z wykorzystaniem *PyPy* w wersji 6.0.0. w przypadku optymalizacji wykorzystującej JIT.

Bez wykorzystania optymalizacji

Wyniki zostały przedstawione w tabeli 1.

Tabela 1. Wyniki otrzymane dla skryptu pythonowego, bez wykorzystania optymalizacji

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	7 godzin 56 minut	196 MB
2.	13 godzin 8 minut	317 MB
3.	25 godzin 36 minut*	619 MB
4.	7 godzin 38 minut	196 MB
5.	7 godzin 42 minuty	196 MB
6.	7 godzin 22 minuty	196 MB

*jest to w tym przypadku aproksymacja, ponieważ ostatni test został przerwany na 612 iteracji po upływie 15,5 godziny.

Z wykorzystaniem optymalizacji

Testowanie oprogramowania z wykorzystaniem optymalizacji zostało podzielone na 3 części:

- Z wykorzystaniem możliwości rozszerzania języka Python językiem C,
- Z wykorzystaniem kompilatora *Just-In-Time,*
- Z połączeniem dwóch powyższych punktów.

1) Rozszerzenia w języku C

Tabela 2. Wyniki dla optymalizacji z wykorzystaniem rozszerzeń Pythona w języku C

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	28 minut 52 sekund	225 MB
2.	51 minut 13 minut	317 MB
3.	1 godzina 52 minuty	620 MB
4.	32 minuty 33 sekundy	196 MB
5.	26 minut 54 sekundy	196 MB
6.	27 minut 38 sekund	196 MB

2) Kompilator JIT

Tabela 3. Wyniki dla optymalizacji z wykorzystaniem kompilatora JIT

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	3 minuty 14 sekund	400 MB
2.	4 minuty 38 sekund	479 MB
3.	8 minut 17 sekund	899 MB
4.	2 minuty 50 sekund	337 MB
5.	2 minuty 53 sekundy	361 MB
6.	2 minuty 44 sekundy	341 MB

3) Kompilator JIT + rozszerzenia C

Tabela 4. Wyniki dla optymalizacji z wykorzystaniem kompilatora JIT i rozszerzeń napisanych w języku C

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	1 godzina 36 minut	1816 MB
2.	2 godziny 59 minut	2502 MB
3.	11 godzin 43 minuty	3765 MB
4.	1 godzina 37 minut	1330 MB
5.	1 godzina 35 minut	1441 MB
6.	1 godzina 36 minut	1358 MB

4.3. Język C

Testy dla języka C przeprowadzane były z wykorzystaniem scenariuszy opisanych w punkcie 4.1.. Kod kompilowany był przy użyciu kompilatora *GCC 8.2.0* rewizja 3, udostępnionego przez środowisko MinGW.

Bez wykorzystania optymalizacji

Tabela 5. Wyniki uzyskane dla algorytmu napisanego w języku C, bez optymalizacji

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	30 minut 53 sekundy	46 MB
2.	51 minut 17 sekund	75 MB
3.	1 godzina 40 minut	144 MB
4.	31 minut 39 sekund	44 MB
5.	31 minut 2 sekundy	44 MB
6.	31 minut 3 sekundy	44 MB

Z wykorzystaniem optymalizacji

1) Optymalizacja O1

Tabela 6. Wyniki otrzymane dla algorytmu (C), z wykorzystaniem optymalizacji O1

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	2 minuty 28 sekund	46 MB
2.	3 minuty 59 sekund	75 MB
3.	7 minut 52 sekundy	147 MB
4.	2 minuty 22 sekundy	46 MB
5.	2 minuty 22 sekundy	46 MB
6.	2 minuty 22 sekundy	46 MB

2) Optymalizacja O2

Tabela 7. Wyniki otrzymane dla algorytmu (C), z wykorzystaniem optymalizacji O2

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	2 minuty 21 sekund	46 MB
2.	3 minuty 53 sekundy	75 MB
3.	7 minut 50 sekund	147 MB
4.	2 minuty 20 sekund	46 MB
5.	2 minuty 20 sekund	46 MB
6.	2 minuty 20 sekund	46 MB

3) Optymalizacja O3

Tabela 8. Wyniki otrzymane dla algorytmu (C), z wykorzystaniem optymalizacji O3

Numer scenariusza	Czas działania	Wykorzystana pamięć
1.	2 minuty 25 sekund	46 MB
2.	3 minuty 10 sekund	75 MB
3.	7 minut 49 sekund	147 MB
4.	2 minuty 26 sekund	46 MB
5.	2 minuty 26 sekund	46 MB
6.	2 minuty 26 sekund	46 MB

5. Wnioski

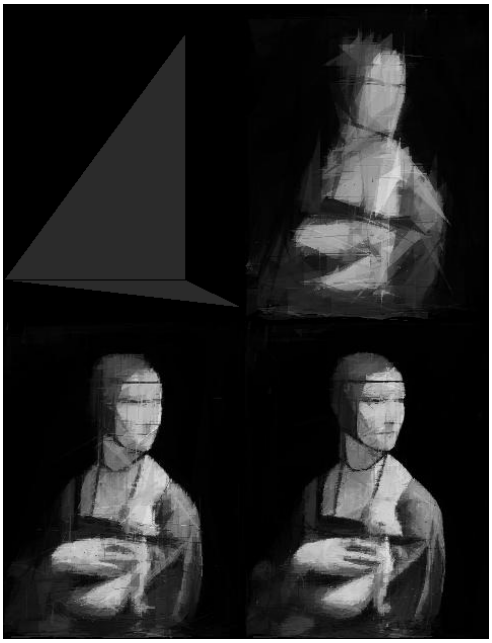
Analizując otrzymane wyniki, stanowczo można stwierdzić, że język Python w swojej podstawowej formie jest językiem bardzo wolnym i mało wydajnym. Wyniki otrzymane dla algorytmu napisanego w tym właśnie języku, bez żadnych optymalizacji są niedopuszczalne, jeżeli chodzi o codzienną pracę i testy. Program napisany w języku C, również pozbawiony metod optymalizacyjnych wykonywał się średnio 15 razy szybciej, zużywając przy tym około 4 razy mniej pamięci. Wynik testów pozbawionych optymalizacji nie jest zaskakujący – język skryptowy był wolniejszy, zaskoczeniem jest natomiast różnica w czasach wykonywania poszczególnych wersji algorytmu. Pamiętając jednak o bardzo dużych możliwościach poprawienia efektywności języka Python, wyniki uzyskane z wykorzystaniem rozszerzeń C API języka Python pozwoliły uzyskać czasy minimalnie lepsze od podstawowej wersji algorytmu napisanej w języku C, nie zwiększając przy tym kosztów związanych z wykorzystaniem pamięci. W opozycji do pierwszej z metod optymalizacyjnych języka Python, wykorzystana została flaga kompilatora *GCC* – O1, która pozwoliła poprawić czasy uzyskane przez podstawową wersję algorytmu (C) dwunastokrotnie, nie zmieniając przy tym zużycia pamięci.

Wykorzystanie *PyPy,* a więc implementacji Pythona, która wykorzystuje kompilator *Just-In-Time* okazał się najlepszą metodą optymalizacyjną w rozpatrywanych przypadkach – biorąc pod uwagę najpierw czas wykonywania algorytmu. Pozwolił osiągnąć czasy gorsze średnio o 32 sekundy od wyników uzyskanych dzięki kolejnej optymalizacji kodu generowanego przez kompilator języka – O2 - które pozwoliło poprawić uzyskane wcześniej (przy zastosowaniu optymalizacji O1) czasy średnio o dodatkowe 6%. Agresywna optymalizacja kodu napisanego w języku C spowodowała nieznaczne pogorszenie wyników, przybliżając się tym samym do wyników otrzymanych przy wykorzystaniu *PyPy.* Ostatnia metoda optymalizacji wykorzystana w przypadku algorytmu w języku Python okazała się zupełnie nietrafiona. Wyniki otrzymane przy jej zastosowaniu były bardzo niezadowalające.

Biorąc pod uwagę powyższe rozważania język C jest językiem bardziej wydajnym od języka Python zarówno pod względem czasu wykonywania obliczeń jak również zużycia pamięci. Wydajność języka Python może być jednak znacząco podwyższona przy wykorzystaniu ogólnodostępnych narzędzi, dorównując wydajności języka C (bez optymalizacji). Narzut w postaci większej ilości zużywanej pamięci można uznać (biorąc pod uwagę wielkości pamięci jakimi operują dzisiejsze komputery) za nieistotny – nawet przy komputerze korzystającym z 4GB pamięci RAM, algorytm nie zakłócałby działania systemu. Przedostatnie zdanie jest bardzo ważne, ponieważ na poczet wydajności danego języka można również zaliczyć szybkość pisania kodu, jego utrzymanie, przejrzystość i zrozumienie, w których język Python jest niekwestionowanym liderem, co w połączeniu z możliwością uzyskiwania czasów działania porównywalnych z językiem C czyni z niego prawdziwe

narzędzie do tworzenia kodu szybkiego i przejrzystego, który będzie łatwy w utrzymaniu i rozbudowie.

W całej pracy porównywane były czasy działania algorytmów, a nie jakość generowanych rozwiązań, ponieważ w rozważanym temacie sprawa ta jest drugorzędna. W opinii autora warto jednak przedstawić wynik zaimplementowanego algorytmu genetycznego. Rysunek 2. przedstawia wyniki działania algorytmu dla 1, 1000, 20000, oraz 100000 pokoleń przy założeniu trzystu osobników w populacji oraz osobników zdolnych do rozmnażania w liczbie 2.



Rys. 2. Wynik działania algorytmu genetycznego dla (od lewej): 1, 1000, 20000, 100000 pokoleń

Literatura

[1] https://www.businessinsider.com/the-10-most-popular-programming-languages-according-to-github-2018-10?IR=T#10-ruby-1 [05.01.2019]

[2] Andersen L. O.: Program Analysis and Specialization for the C Programming Language. Dania, Maj 1994.

[3] https://en.wikipedia.org/wiki/C_(programming_language) [05.01.2019]

[4] Lutz M.: Learning Python. O'Reilly Media, 2013.

[5] Ateeq M., Habib H., Umer A., Rehman M. U.: C++ or Python? Which One to Begin with: A Learner's Perspective. [W]: 2014 International Conference on Teaching and Learning in Computing and Engineering, IEEE, 11-13 April 2014.

[6] Dobrescu L.: Replacing ANSI C with other modern programming languages. [W]: 2014 International Symposium on Fundamentals of Electrical Engineering (ISFEE), IEEE, 28-29 Nov. 2014.

[7] Prechelt L.: An empirical comparison of seven programming languages. Computer, Volume: 33, Issue: 10, Oct 2000.

[8] Jun L., Ling L.: Comparative research on Python speed optimization strategies. [W]: 2010 International Conference on Intelligent Computing and Integrated Systems, IEEE, 22-24 Oct. 2010.

[9] Zhang H., Nie J.: Program performance test based on different computing environment. [W]: 2016 IEEE International Conference of Online Analysis and Computing Science (ICOACS), IEEE, 28-29 May 2016.

[10] Man K. F., Tang K. S., Kwong S.: Genetic algorithms: concepts and applications [in engineering design]. IEEE Transactions on Industrial Electronics, 1996, Volume: 43, Issue: 5, Oct 1996, p.: 519 - 534.

[11] Srinivas M., Patnaik L. M.: Genetic algorithms: a survey. Computer, 1994, Volume: 27, Issue: 6, June 1994, p.: 17 - 26.

[12] http://psyco.sourceforge.net/ [02.02.2019]

[13] http://pypy.org/index.html [02.02.2019]

[14] https://docs.python.org/3.6/extending/extending.html [02.02.2019]

[15] https://github.com/python/cpython [06.01.2019]

[16] Merelo-Guervós J. J., Blancas-Álvarez I., Castillo P. A., Romero G., Rivas V. M., García-Valdez M., Hernández-Águila A., Román M.: A comparison of implementations of basic evolutionary algorithm operations in different languages. [W]: 2016 IEEE Congress on Evolutionary Computation (CEC), IEEE, 24-29 July 2016.

[17] Suman S., Giri V. K.: Genetic Algorithms: Basic Concepts and Real World Applications. Gorakhpur, Uttar Pradesh, India, May 2016.

[18] Myalapalli V. K., Myalapalli J. K., Savarapu P. R.: High performance C programming. [W]: 2015 International Conference on Pervasive Computing (ICPC), IEEE, 8-10 Jan. 2015.

[19] Gerardo de la Fraga L., Tlelo-Cuautle E., Azucena A. D. P.: On the Execution Time of a Computational Intensive Application in Scripting Languages. [W]: 2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT), IEEE, 25-27 Oct. 2017.