Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Replikacja obrazów za pomocą algorytmów genetycznych

Jakub Gogola
Nr indeksu: 236412

Praca inżynierska napisana
pod kierunkiem
dr. inż. Piotra Sygi

Politechnika
Wrocławska

Wrocław 2020

Spis treści

Wstęp

W niniejszej pracy zbadano problem replikacji obrazów za pomocą algorytmów genetycznych. Głównym celem autora, zgodnie z tytułem pracy, było ujęcie problemu replikacji w ramy języka algorytmów genetycznych. W dobie powszechnego wykorzystania technik udostępnianych przez *machine learning*, autor podjął się analizy podejścia stosowanego zanim nastąpił tak dynamiczny rozwój wspomnianej gałęzi informatyki. W pracy dokonano analizy możliwości algorytmów genetycznych oraz, w sposób eksperymentalny, spróbowano wyznaczyć takie ograniczenia zadane tymże, które pozwalają uzyskać najlepsze rezultaty.

Z racji, że praca ta ma charakter inżynierski, oprócz ujęcia teoretycznego, autor przygotował również implementację prezentowanego na jej stronach algorytmu w celu praktycznego zbadania sposobu jego działania oraz generowanych przez niego wyników.

Wartym uwagi jest również fakt, że oprócz przedstawionego w pracy ujęcia problemu replikacji w ramy formalne, autor podjął się opracowania tego tematu również z powodu chęci zbadania jakie efekty daje takie rozwiązanie od strony czysto wizualnej. Skupił się on na replikacji obrazów w ten sposób, aby uzyskać wynik najbardziej zbliżony do oryginalnego, ale należy tutaj zaznaczyć, że opisywane w niniejszej pracy rozwiązanie może być zastosowane chociażby do generowania grafiki typu *pixelart*. Możliwym zastosowaniem jest replikacja fraktali, czyli pewnych struktur z zauważalnymi regularnościami. Problem ten został poruszony w pracy [24], która dotyczy generowania szeroko pojętej sztuki za pomocą właśnie algorytmów genetycznych i technik zbliżonych do tych, które zostały zastosowane w niniejszej pracy.

1.1 Struktura pracy

Praca została podzielona na siedem rozdziałów, które dotyczą zarówno części teoretycznej pracy jak i takie, w których omówiono szczegóły implementacyjne. Przedstawiono w nich również analizę otrzymanych wyników oraz wnioski uzyskane z przeprowadzonych badań nad problemem.

Pierwszy rozdział pracy to niniejszy wstęp, będący krótkim wprowadzeniem w ideę i cel przyświecający jej autorowi.

W rozdziale 2 zawarto dokładną analizę problemu replikacji. Przedstawiono podstawowe pojęcia dotyczące teorii obliczeń i złożoności obliczeniowej, idę działania i zastosowanie algorytmów metaheurystycznych ze szczególnym wyróżnieniem algorytmów genetycznych. Dokonano również analizy problemu replikacji ujętej w ramy problemu optymalizacyjnego. Załączono przegląd alternatywnych rozwiązań.

W rozdziale 3 przedstawiono strukturę projektu w postaci wymagań, które powinien spełniać algorytm oraz jego implementacja - zbiór parametrów wejściowych i wyjściowych oraz wymagania funkcjonalne i niefunkcjonalne.

Rozdział 4 dotyczy implementacji algorytmu w języku `Go`. Omówiono tutaj, w sposób opisowy, poszczególne części implementacji. Przedstawiono również skrótowo sposób reprezentacji obrazów oraz model współbieżności w wybranym przez autora języku programowania.

W rozdziale 5 spisana została instrukcja obsługi programu, uwagi dotyczące jego modyfikacji oraz przykłady kodu, które pozwalają na uruchomienie implementacji. Zawarto tam również spis wymagań technicznych oraz potrzebnych narzędzi i bibliotek.

Rozdział 6 zawiera szczegółową analizę uzyskanych przez autora wyników ze szczególnym naciskiem na przeanalizowanie wpływu parametrów wejściowych programu na jakość otrzymywanych rozwiązań.

Podsumowanie pracy (część 7) zawiera wyciągnięte na podstawie otrzymanych wyników wnioski oraz ogólne podsumowanie pracy.

Do pracy została załączona bibliografia zawierająca spis wykorzystanej literatury oraz opis zawartości płyty CD z kodami źródłowymi dołączonej do pracy, który znajduje się w dodatku A.

Analiza problemu

W niniejszym rozdziale zostanie zdefiniowany problem replikacji obrazów rozważany w pracy. Przedstawiona zostanie również koncepcja i założenia algorytmów stosowanych przez autora do testowania oraz praktycznego zobrazowania analizowanego problemu. Zostaną również omówione podstawowe pojęcie niezbędne do zrozumienia idei i genezy algorytmów metaheurystycznych.

2.1 Replikacja obrazów

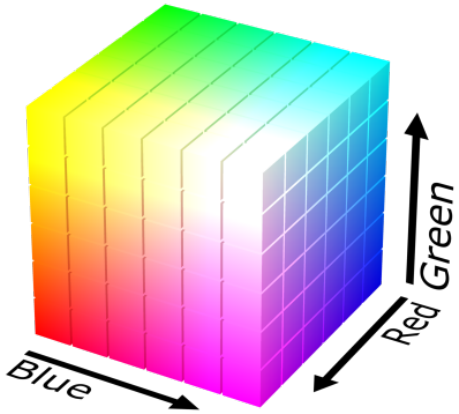
Replikacją obrazów można określić proces, podczas którego oryginalny (wzorcowy) obraz jest reprodukowany za pomocą pewnej techniki. Użyto tutaj nie bez powodu słowa *technika*, ponieważ, pomimo że praca skupia się na pewnym algorytmie i jego implementacji w wysokopoziomowym języku programowania, to sam proces replikacji (reprodukcji) nie jest pojęciem nowym. Stosowany on był już przez wiele stuleci w przeszłości przez artystów (malarzy, rzeźbiarzy), którzy zajmowali się kopiowaniem innych dział malarskich lub rzeźbiarskich. Idea przyświecająca osobom, kopiującym działa malarskie, w dobie obecnej technologii, została przeniesiona na grunt techniki cyfrowej. Obecnie istnieje wiele narzędzi (programów lub bibliotek współpracujących z językami programowania) umożliwiających wykonywanie operacji na cyfrowych formatach obrazów. Narzędzia te stosują pewne wyspecjalizowane algorytmy, gdzie zadaniem każdego z nich jest dokonanie na obrazie pewnych zmian. Wraz z rozwojem techniki cyfrowego przetwarzania obrazów, naturalnym jest, że zdecydowano się również na cyfrową reprodukcję.

W tym miejscu należy również zaznaczyć, że przedstawiany w pracy proces replikacji, do którego autor użył algorytmów genetycznych, nie polega na skopiowaniu pewnego obrazu w formie *jeden do jednego*. Opisywany proces ma na celu jedynie przybliżenie danego obrazu, więc należy też rozpatrywać ten opisywany sposób jako formę procesu twórczego. Nie da się bowiem powiedzieć, że zawsze, po zakończeniu działania algorytmu otrzymane zostanie dokładnie to, co było kopiowane, to jest każdy piksel wyprodukowanego rozwiązania będzie identyczny jak piksel obrazu oryginalnego (rozumie się tutaj obraz przez pewną prostokątną macierz pikseli, gdzie każdemu pikselowi przyporządkowane są pewne wartości liczbowe określające jego barwę). Zamierzonym efektem jest to, że po zakończeniu działania algorytmu jest uzyskiwane jest jedynie pewien obraz podobny [1] do swojego pierwowzoru i nie da się określić przed rozpoczęciem działania programu jaki wynik zostanie uzyskany.

2.1.1 Idea cyfrowej replikacji obrazów

Rozważając cyfrowe podejście do replikacji obrazów należy rozumieć ten proces jako zastosowanie pewnego algorytmu, który mając do dyspozycji obraz oryginalny (reprodukowany) dąży do tego, aby stworzyć jego jak najwierniejszą kopię lub, głównie w przypadku algorytmów uczenia maszynowego (ang. *machine learning*), na podstawie pewnego zbioru obrazów utworzyć obraz mający cechy zbioru, na którym algorytm się uczył i jak najbardziej przypominający obraz stworzony przez człowieka (zdjęcie, rysunek, obraz namalowany przez malarza). Do tego celu wykorzystywane są sieci neuronowe, tzw. GANy (ang. *Generative Adversarial Network*) mające swoje powszechne zastosowanie w *deep learningu* (cf. [8]).

[1] *Podobny* należy rozumieć jako obraz podobny pod względem wizualnym lub względem określonej funkcji mierzącej podobieństwo dwóch obrazów.



Rysunek 2.1: Schematyczne przedstawienie skali RGB. Każdy voxel sześcianu reprezentuje unikalny kolor. Można zatem każdy kolor zapisać za pomocą trójki uporządkowanej (R, G, B), aby wskazać konkretny odcień. (*Źródło:* [9])

2.2 Cyfrowa reprezentacja obrazów

Obraz w postaci cyfrowej jest reprezentowany na poziomie pamięci jako pewien ciąg bitów, jak wszystkie dane cyfrowe. Na poziomie języka programowania, w którym wykonywane są operacje na obrazie, obraz reprezentowany jest w pewnej skali kolorystycznej, np. RGBA, CMYK, HSV, CIELAB lub w skali szarości. Obraz przedstawiany jest jako macierz prostokątna $m \times n$ pikseli, gdzie każdemu pikselowi jest przyporządkowany pewien kolor. W przypadku niniejszej pracy autor będzie używał dwóch skali, za pomocą których obrazy mogą być reprezentowane cyfrowo - RGBA oraz skali szarości.

2.2.1 Skala RGBA

W modelu przestrzeni barw RGBA (cf. [10]) obraz jest reprezentowany jako tablica pikseli o wymiarach $n \times m$. Każdy piksel (reprezentowany za pomocą pewnej struktury lub obiektu) posiada informację o kolorze w tym przypadku zapisywaną za pomocą czterech parametrów zwanych kanałami. Pierwsze trzy kanały (RGB) reprezentują kanałowi czerwonemu (ang. *red*), zielonemu (ang. *green*) oraz niebieskiemu (ang. *blue*). Ostatni kanał A jest nazywany kanałem *alfa* i określa on stopień przeźroczystości danego piksela. Skala ta została schematycznie przedstawiona na rysunku 2.1. Każdy z wymiarów przedstawionego na grafice sześcianu reprezentuje jeden z kanałów RGB. Zakładając, że do reprezentacji używa się dodatnich liczb całkowitych 8-bitowych, wówczas każda ze ścian tego sześcianu ma wymiary 255×255. Każda współrzędna (r, g, b) odpowiada konkretnemu kolorowi ze skali. Jest to jednak jedynie przedstawienie samej skali RGB. W celu uzyskania skali RGBA, na przedstawioną na grafice kostkę należy jeszcze nałożyć kanał *alfa*. Wówczas z trójek uporządkowanych określających współrzędne otrzymuje czwartą wartość α, która definiuje jej przeźroczystość, którą, w celu lepszego uzmysłowienia czytelnikowi czym jest α, można nazwać *widocznością* danego koloru.

Skala RGBA jest stosowana m. in. w formacie `PNG` służącym do zapisu grafiki oraz jest powszechnie stosowana w programach do obróbki graficznej oraz w bibliotekach języków programowania służących do przetwarzania zdjęć.

2.2.2 Skala szarości

W skali szarości, w przeciwieństwie do opisywanej w podrozdziale 2.2.1 skali RGBA, rozważa się tylko jeden kanał. Określa on jedynie intensywność danego piksela. W przypadku, gdy ten kanał reprezentowany

jest przez 8-bitowe dodatnie liczby całkowite, wówczas wartość 0 oznacza najmniejszą intensywność (reprezentowany jako kolor czarny), a wartosc maksymalna 255 oznacza największą intensywność (reprezentowaną jako kolor biały).

Skala szarości jest również podzbiorem skali RGBA - rozważane są w niej takie kolory, w których pierwsze trzy kanały (RGB) maja dokładnie taką samą wartość. Dla skali szarości nie rozważa się wówczas kanału *alfa* i przyjmuje się, że zawsze przyjmuje on wartość maksymalną, czyli 255.

2.2.3 Znaczenie skali barw w technikach cyfrowego przetwarzania obrazów

W cyfrowym przetwarzaniu obrazów można wyróżnić wiele technik i metod pozwalających na dokonywanie różnych przekształceń i operacji obrazów. Celem tej pracy nie jest jednak opisanie algorytmów służących do przetwarzania obrazów, ale skupienie się na jednym, konkretnym problemie, jakim jest ich replikacja. Każdą jednak z techniki i metod, niezależnie od ich zastosowania, łączy jedna wspólna cecha - każda z nich działa na obrazie, który jest reprezentowany w określonej skali barw. Jest to podstawowa struktura w tej dziedzinie, ponieważ pozwala na wygodną reprezentację obrazu w formie cyfrowej i późniejsze jego przekształcenia. W związku z tym, że skala barw pozwala przetłumaczyć widoczne dla ludzkiego oka kolory na zrozumiałe dla komputera liczby, na których można operować różnymi algorytmami.

Po przedstawieniu podstaw cyfrowej reprezentacji obrazu, autor skupi się na prowadzeniu podstawowych pojęć, które są niezbędne do zrozumienia opisywanego w dalszych częściach pracy algorytmu przetwarzania obrazów służącego do ich replikacji.

2.3 Przegląd podstawowych klas złożoności obliczeniowych

Tak jak opisano na początku rozdziału 2, praca dotyczy cyfrowych sposobów replikacji obrazów i, co z tym związane, stosowanych do tego algorytmów i narzędzi. Autor pracy wybrał do przedstawienia tego procesu algorytm genetyczny, należący do rodziny algorytmów metaheurystycznych. Zanim jednak zostaną one opisane, należy przedstawić kilka niezbędnych pojęć ściśle związanych z procesem ich projektowania oraz zastosowaniem. Na początku tej części pracy zostaną przedstawione definicje niezbędne w dalszych rozważaniach, a następnie zostanie przedstawiony związek pomiędzy przedstawianymi pojęciami a badanym problemem.

2.3.1 Problemy decyzyjny, obliczeniowy i optymalizacyjny

Pojęciami niezbędnymi w kolejnym podrozdziale pracy, poświęconym algorytmom metaheurystycznym, są problemy **decyzyjny** oraz **optymalizacyjny**.

Problem decyzyjny

Definicja 2.1 *Problem P określa się jako decyzyjny, jeżeli wymaga on udzielenia odpowiedzi na pytanie w formie "tak" lub "nie".*

W algorytmice istnieje wiele przykładów problemów, które można określić mianem *decyzyjnych*. W teorii złożoności wiele problemów usiłuje się sprowadzić do takiej formy, w której problem wymagający odpowiedzi na bardziej złożone pytanie, wymaga jedynie odpowiedzi *tak* lub *nie*. W przypadku problemów optymalizacyjnych celem jest zredukowanie problemu do takiej formy, w której wymagana jest odpowiedź na pytanie: *Czy możliwym jest osiągnięcie żądanej wartości?*. [22] podaje jako przykład problemu decyzyjnego zagadnienie osiągalności w grafie.

Przykład 2.1 *Niech $G = (V, E)$ będzie grafem, gdzie V jest skończonym zbiorem wierzchołków tego grafu, a E zbiorem jego krawędzi. Problem osiągalności polega na sprawdzeniu czy dla grafu G i jego wierzchołków $1, n \in V$ istnieje ścieżka z 1 do n.*

W powyższym przykładzie łatwo można zauważyć pytanie, na które odpowiedzi wymaga problem: *Czy istnieje taka ścieżka, że... ?*.

Problem optymalizacyjny

Definicja 2.2 *Problem P nazywa się optymalizacyjnym jeżeli postawionym w nim celem jest znalezienie optymalnego rozwiązania wśród wielu innych możliwych rozwiązań. Przez rozwiązanie optymalne rozumie się takie, które zostało najlepiej ocenione przez zdefiniowaną funkcję celu f_c. Dla P będącego problemem minimalizacyjnym rozwiązanie optymalne definiuje się tak, jak opisano poniżej.*

Niech S będzie zbiorem rozwiązań. Za rozwiązanie optymalne s_{opt} w problemie minimalizacyjnym przyjmuje się takie, które spełnia następującą zależność:

$$s_{opt} = \min_{s \in S} f_c(s) \tag{2.1}$$

[22] podaje jako przykład problemu optymalizacyjnego podaje problem maksymalnego przepływu w sieci.

Przykład 2.2 *Niech sieć $N = (V, E, s, t, c)$ będzie grafem (V, E) z wyróżnionymi wierzchołkami s i t zwanymi odpowiednio źródłem i ujściem, gdzie źródło s nie posiada krawędzi doń wchodzących, a ujście t krawędzi zeń wychodzących. Niech c będzie funkcją określającą przepustowość $c : E \to \mathbb{N}$ przyporządkowującą każdej krawędzi wartość $f(i,j) \leqslant c(i,j)$ tak, aby dla każdego wierzchołka v takiego, że $v \neq s \wedge v \neq t$, suma wartości f dla krawędzi wchodzących do wierzchołka, jest taka sama, jak dla krawędzi z wierzchołka wychodzących. Przepływem w grafie jest suma wartości f dla krawędzi wychodzących z wierzchołka s lub, analogicznie, wchodzących do t. Zadaniem postawionym w problemie jest znalezienie przepływu o największej możliwej wartości.*

W powyższym przykładzie daje się zauważyć, że nie jest to problem decyzyjny z uwagi na postawione w nim pytanie wymagające odpowiedzi będącej liczbą lub konkretnym mapowaniem. Daje się on jednak sprowadzić do postaci problemu decyzyjnego. W tym celu należy dodać do opisu problemu *wartość docelową* przepływu, czyli wielkości, którą należy optymalizować, a następnie zadać pytanie: *Czy taka wartość jest osiągalna?*.

W poprzedniej części przedstawiono definicje problemów decyzyjnego oraz optymalizacyjnego wraz z ich przykładami. Jednak, rozważając pewien problem algorytmiczny, w dalszej kolejności należy rozpatrzyć algorytm, który go rozwiązuje ze szczególnym uwzględnieniem jego złożoności czasowej. W celu łatwiejszego i bardziej formalnego zdefiniowania czym jest złożoność czasowa, zostanie wprowadzone pojęcie modelu obliczeń opartego na **maszynie Turinga** oraz języka, który przez tę maszynę może być rozumiany. Obie definicje zostały zaczerpnięte z [12].

2.3.2 Maszyna Turinga

Maszyna Turinga jest modelem obliczeń zaproponowanym przez Alana Turinga będącym jednocześnie modelem bardzo prostego komputera, na którym może zostać zasymulowane wykonywanie dowolnego programu. Składa się ona z następujących elementów:

- nieskończonej taśmy podzielonej na osobne komórki, gdzie w każdej z komórek może być zapisany symbol zrozumiały dla maszyny,
- głowicy mogącej przesuwać się w dowolnym kierunku taśmy, która potrafi zapisywać i odczytywać symbole znajdujące się na taśmie,
- mechanizmu sterującego, który może decydować o kolejnych stanach maszyny.

Formalnie można zdefiniować maszynę Turigna w następujący sposób:

Definicja 2.3 *Maszynę Turinga M definiuje się jako siódemkę $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$, gdzie:*

- *Q jest skończonym zbiorem stanów maszyny,*
- *Σ jest skończonym zbiorem symboli wejściowych zapisanych na taśmie maszyny przed rozpoczęciem obliczeń,*
- *Γ jest skończonym zbiorem symboli taśmowych (oczywistym jest, że $\Sigma \in \Gamma$,*

- *# jest symbolem pustym,*
- *δ jest funkcją przejścia pomiędzy stanami maszyny i definiowana jest jako: $\delta : Q \times \Gamma \to Q \times \Gamma \times K$. Oznacza to, że funkcja będąc w jakimś stanie x i mając na wejściu symbol x może przejść do stanu B jednocześnie zapisując na taśmie symbol y i przesuwając głowicę w lewo lub w prawo, czyli w jeden z kierunków k ze zbioru $K = \{\leftarrow, \rightarrow\}$, co formalnie można zapisać jako $\delta(A, x) = (B, y, k)$.*
- *q_0 jest stanem początkowym, w jakim znajduje się maszyna przed rozpoczęciem wykonywania obliczeń,*
- *F jest zbiorem stanów akceptujących (oczywistym jest, że $F \in Q$). Maszyna kończy obliczenia w momencie osiągnięcia jednego ze stanów z F.*

*Powyższa definicja dotyczy **deterministycznej** maszyny Turinga (DTM). **Niedeterministyczną** maszyną Turinga (NTM)* [2] *nazywamy taką, dla której wynikiem dla każdego argumentu jest pewien zbiór, to jest: $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times K)$. Dla tak zdefiniowanej funkcji przejścia możliwe jest przejście ze stanu A do więcej niż jednego kolejnego stanu.*

Kolejnym ważnym pojęciem, niezbędnym do stworzenia kolejnych definicji, jest język akceptowalny przez daną maszynę M.

Definicja 2.4 *Językiem maszyny M nazywa się taki język $L(M) \subseteq \Sigma^*$ będący zbiorem takich słów wejściowych, dla których maszyna M kończy swoje obliczenia w jednym ze stanów akceptujących. Inaczej mówiąc, maszyna M rozpoznaje język $L(M)$.*

Autor zaznacza, że powyższe definicje mają na celu jedynie wprowadzenie podstawowych pojęć niezbędnych do zrozumienia kolejnych zagadnień. Szczegółowy opis maszyny Turinga można znaleźć w [22, 12].

2.3.3 Złożoność czasowa

W świetle wprowadzonych definicji można wprowadzić formalną definicję złożoności czasowej opartej o model obliczeń, jakim jest maszyna Turinga (definicja została zaczerpnięta z [12]).

Definicja 2.5 *Złożoność czasową danego algorytmu definiuje się jako liczbę kroków potrzebną na wykonanie najdłuższego z możliwych obliczeń na NTM, po której NTM się zatrzyma.*

Próbując ująć mniej formalnie powyższą definicję, można stwierdzić, że za złożoność czasową przyjmuje się *najgorszy przypadek* działania algorytmu (ang. *worst-case scenario*), czyli taki, w którym musi wykonać on najwięcej kroków obliczeniowych.

W przypadku problemu osiągalności w grafie można łatwo wskazać algorytm rozwiązujący dany problem. Można w tym celu wykorzystać *algorytm Dijkstry* (cf. [14]), który służy do znajdowania najkrótszej ścieżki z pojedynczego wierzchołka w grafie, którego krawędzie mają nieujemne wagi. Algorytm, chociaż służy do znajdowania najkrótszej ścieżki, to jednak istnienie takowej implikuje w ogóle istnienie ścieżki pomiędzy dwoma wierzchołkami, a co za tym idzie - zastosowanie tego algorytmu pozwala odpowiedzieć na pytanie postawione w problemie osiągalności.

Złożoność algorytmu Dijsktry dla grafu $G = (V, E)$ może zostać wyrażona w następujący sposób:

- $\mathcal{O}(V^2)$ - implementacja *naiwna*, gdzie do kolejki priorytetowej wykorzystano tablicę,
- $\mathcal{O}(E \log V)$ - implementacja z użyciem kopca,
- $\mathcal{O}(E + V \log V)$ - implementacja z użyciem kopców Fibonacciego.

Widać zatem, że spory wpływ na złożoność algorytmu ma użyta struktura danych - w najgorszym przypadku algorytm działa w czasie wielomianowym, a w najlepszym możliwym jest osiągnięcie czasu logarytmicznego.

Należy tutaj zaznaczyć, że podawane tutaj przykłady algorytmów i ich złożoności nie powinny być do końca utożsamiane ze złożonością problemu. Za złożoność czasową problemu przyjmuje się szybkość działania

[2] Skrótów *DTM* oraz *NTM* autor używa w dalszej części pracy jako odniesień odpowiednio do deterministycznej i niedeterministycznej maszyny Turinga.

najszybszego znanego algorytmu, który go rozwiązuje. Dlatego przyjmuje się, że złożoność problemu jest nie większa, niż złożoność algorytmów go rozwiązujących.

Podobnie dla problemu znajdowania maksymalnego przepływu również można znaleźć algorytm (cf. [22]) rozwiązujący ten problem w czasie wielomianowym. Jednak istnieją pewne algorytmy, których rozwiązanie w czasie co najwyżej wielomianowym względem długości wejścia nie jest znane. Jako przykład można podać **problem komiwojażera** (ang. *traveling salesman problem*) szerzej opisany w [22]. W świetle tych dwóch przykładów zostaną wprowadzone klasy złożoności obliczeniowej, pozwalające na klasyfikację problemów pod względem właśnie tej własności.

2.3.4 Klasy P i NP

Definicja 2.6 *Klasa P (ang. polynomial time) jest zbiorem problemów decyzyjnych, które można rozwiązać na deterministycznej maszynie Turinga w czasie co najwyżej wielomianowym.*

Widać, że do *klasy P* można zaliczyć problem osiągalności w grafie jak również problem maksymalnego przepływu. Znane są bowiem dla nich algorytmy, które znajdują rozwiązanie w czasie co najwyżej wielomianowym.

Definicja 2.7 *Klasa NP (ang. nondeterministic polynomial time) to zbiór problemów decyzyjnych, dla których rozwiązanie można zweryfikować na DTM w czasie wielomianowym. Oznacza to, że istnieje algorytm rozwiązujący dany problem w czasie wielomianowym na niedeterministycznej maszynie Turinga.*

Przykładem problemu należącego do *klasy NP* jest przedstawiony wcześniej problem komiwojażera.

Kolejnym krokiem, który jest ściśle powiązany z prezentowanymi powyżej pojęciami są NP-zupełność oraz NP-trudność. Wymagałoby to jednak dosyć szczegółowej analizy pojęć w celu przedstawienia czytelnikowi pełnego obrazu tych problemów. Bardziej rozbudowany obraz na temat złożoności obliczeniowej można znaleźć w [22].

2.3.5 Podsumowanie

Podsumowując opisywane powyżej algorytmy i klasy złożoności, można stwierdzić, że problemy dające się sklasyfikować jako te należące do klasy P można uznać za *łatwe obliczeniowo* - znane są dla nich algorytmy działające w czasie co najwyżej wielomianowym na DTM i w celu rozwiązania takiego problemu na komputerze konieczne jest stworzenie jedynie odpowiedniej implementacji w wybranym języku programowania. Problemy z klasy NP natomiast można uznać za *trudne obliczeniowo*. Rozwiązanie tych problemów dla dużego zbioru danych nie jest wykonalne w *akceptowalnym* czasie. Przez pojęcie *akceptowalny* należy tutaj rozumieć czas wielomianowy na DTM. Dla problemów tego typu należało więc poszukać innych metod, które pozwalają z wysokim prawdopodobieństwem znaleźć dla nich rozwiązanie w czasie rozumianym jako *akceptowalny*. W przypadku takich metod najczęściej mówi się o rozwiązaniu, które niekoniecznie jest optymalne, ale jest bliskie optimum.

2.4 Replikacja obrazów jako problem optymalizacyjny

Replikacja obrazów jest dobrym przykładem problemu optymalizacyjnego. Głównym celem replikacji jest znalezienie takiego przybliżenia obrazu, które, pod względem pewnej ustalonej funkcji kosztu, jest najbliższe obrazowi oryginalnemu. Jest to dobry przykład problemu minimalizacyjnego.

Definicja 2.8 *Problem optymalizacyjny nazywa się minimalizacyjnym, jeżeli jego celem jest znalezienie rozwiązania, dla którego zadana funkcja celu przyjmuje wartość najmniejszą lub, gdy ta nie istnieje, minimalną (na przykład dla funkcji okresowych). Oznacza to, że jest szukane minimum globalne dla tej funkcji.*

2.4.1 Zagadnienie aproksymacyjne dla problemu replikacji

Problem replikacji obrazów, jak wspomniano na początku tego podrozdziału, również jest przypadkiem problemu optymalizacyjnego. Głównym zadaniem jest taka aproksymacja obrazu replikowanego, aby otrzymać wynik jak najbardziej do niego zbliżony. Aby możliwym było zmierzenie jakości otrzymanej aproksymacji i późniejsze ocenienie tego, jak bardzo jest one oddalone od oczekiwanego. W tym celu koniecznym jet zdefiniowanie odpowiedniej funkcji celu, która pozwoli na statystyczne przyporządkowanie każdemu rozwiązaniu uzyskanemu w danej iteracji pewnej wartości liczbowej, która będzie później traktowana jako ocena każdego z osobników i pozwoli na utworzenie nowej populacji, która zostanie wykorzystana w kolejnej iteracji programu.

Funkcja celu, w przypadku replikacji obrazów, musi mierzyć w pewien określony sposób to, jak bardzo otrzymany obraz różni się od oryginalnego. W przypadku, gdy kolejnymi rozwiązaniami są obrazy, należy uwzględnić cechy charakterystyczne dla tego typu struktur. W tym przypadku najlepszym kryterium porównawczym będzie ocena każdego z osobników na poziomie skali barw, w której jest on zapisany. W przypadku przedstawianego algorytmu może być to skala RGBA lub skala szarości.

Dla każdej skali barw funkcja celu będzie przyjmować inne wartości oraz nieco odmienny jest sposób jej obliczania. W obu jednak przypadkach zostanie ona zdefiniowana w następujący sposób:

$$f_c(o) = \sum_{i=0}^{m} \sum_{j=0}^{n} p_c(o_{ij}) \tag{2.2}$$

gdzie o jest pojedynczym osobnikiem, czyli pewnym obrazem o wymiarach $m \times x$ pikseli (może on być traktowany jako macierz pikseli), a funkcja p oblicza różnicę pomiędzy konkretnym pikselem sprawdzanego osobnika a osobnika wzorcowego (oryginalnego obrazu). Jej definicja będzie się różnić w zależności od zastosowanej skali barw, co zostanie opisane poniżej.

2.4.2 Funkcja celu dla skali szarości

W przypadku zastosowania skali szarości do reprezentacji osobników, każdy piksel pojedynczego przenosi na sobie informację odnośnie swojej intensywności. Przyjmujemy, że jest to 8-bitowa dodatnia liczba całkowita z przedziału $[0, 255]$. Teraz, niech zmienna pix reprezentuje pojedynczy piksel, a funkcja I zwraca różnicę intensywności dla pix i odpowiadającego mu (względem współrzędnych) piksela obrazu oryginalnego. Wówczas funkcja p_c dla pojedynczego piksela przyjmuje postać:

$$p_c(\text{pix}) = |I(\text{pix})| \tag{2.3}$$

Na wynik funkcji I zostaje nałożony moduł w celu uniknięcia występowania wartości ujemnych.

2.4.3 Funkcja celu dla skali RGBA

Funkcja celu w przypadku skali RGBA zakłada obliczenie o ile różnią się dane dwa piksele na każdym z kanałów. Niech funkcje R, G, B, A liczą różnicę pomiędzy zadanym pikselem pix a odpowiadającym mu pikselem obrazu oryginalnego odpowiednio dla kanału czerwonego (ang. *red*), zielonego (ang. *green*), niebieskiego (ang. *blue*) oraz dla kanału *alfa*. Wówczas p_c wyraża się jako:

$$p_c(\text{pix}) = |R(\text{pix})| + |G(\text{pix})| + |B(\text{pix})| + |A(\text{pix})| \tag{2.4}$$

Wartość każdej funkcji (R, G, B, A) zostaje ujęta w moduł w celu uniknięcia wartości ujemnych, podobnie jak ma to miejsce dla funkcji I dla obrazów reprezentowanych w skali szarości.

2.4.4 Funkcja celu a ocena jakości rozwiązań

Po obliczeniu funkcji celu dla każdego osobnika, osobniki zostają posortowane względem wartości f_c dla każdego z nich. Z faktu, że problem replikacji jest problemem minimalizacyjnym, za osobniki najlepsze uznaje się te, dla których funkcja f_c przyjmuje wartości najmniejsze. Oznacza to, że poszczególne kolory dla danych pikseli najmniej odbiegają od tych na obrazie replikowanym.

2.5 Algorytmy metaheurystyczne

Tak jak opisano na początku tego rozdziału, praca dotyczy cyfrowych sposobów replikacji obrazów i, co z tym związane, stosowanych do tego algorytmów i narzędzi. Autor pracy wybrał do przedstawienia tego procesu algorytm genetyczny, należący do rodziny algorytmów metaheurystycznych. W tym podrozdziale zostanie opisana idea algorytmów metaheurystycznych ze szczególnym uwzględnieniem algorytmów genetycznych. Przedstawiony zostanie również algorytm zaimplementowany przy okazji pisania tej pracy.

2.5.1 Idea i działanie

Zanim opisana zostanie idea algorytmów metaheurystycznych, należy wytłumaczyć pochodzenie ich nazwy. Pochodzi ona od słowa *heurystyka*, które wywodzi się z języka greckiego *heuriskō* co tłumaczy się jako *znajduję*. Oznacza ono "umiejętność wykrywania nowych faktów i związków między faktami"(cf. [11]). Przyjmując tę definicję można następnie zdefiniować słowo *metahuerystyka*, które określa ogólny algorytm służący do rozwiązywania pewnych problemów. Dokładniej - określenie to odnosi się do "heurystyki wyższego poziomu". Wynika to z faktu, że tego typu algorytmy nie rozwiązują bezpośrednio żadnego problemu, ale ich celem jest podanie sposobu na utworzenie odpowiedniego algorytmu.

Algorytmy metaheurystyczne używane są najczęściej do rozwiązywania problemów obliczeniowych z dużym naciskiem na problemy optymalizacyjne. Używa się ich w przypadkach, gdy rozwiązanie danego zagadnienia jest bardzo kosztowne. Idealnym przykładem są tutaj problemy należące do omawianej klasy NP.

Posiadając już zbiór definicji i pewną intuicję można przejść do zdefiniowania **algorytmu metaheurystycznego** (cf. [20]).

Definicja 2.9 *Niech dany będzie problem P. Niech $L(P)$ będzie zbiorem możliwych zdań zapisanych w języku problemu P i niech będzie on traktowany jako zbiór wszystkich możliwych rozwiązań P. Rozwiązaniem problemu P jest dowolne $x \in S$, gdzie S jest pewnym podzbiorem zbioru $L(P)$. Znalezienie rozwiązania problemu polega na znalezieniu albo całego S albo, w zależności od tego jak problem jest zdefiniowany, przynajmniej jednego elementu zbioru S. Algorytm metaheurystyczny opisuje w jaki sposób mając dowolny element z $L(P)$ przejść do innego elementu z tego zbioru mając największe prawdopodobieństwo znalezienia rozwiązania P.*

Algorytm metaheurystyczny zajmuje się zatem przeszukiwaniem przestrzeni rozwiązań (niekoniecznie całej, najczęściej tylko jej części). Na podstawie wcześniejszych rozwiązań stara się znaleźć kolejne, które poprawiałoby to poprzednio znalezione. Nie oznacza to, że rozwiązanie lepsze od poprzedniego zostanie znalezione, ponieważ algorytm może utknąć w lokalnym optimum, które może być odległe od poszukiwanego optimum globalnego. Wówczas dążenie do znalezienia jedynie lepszego rozwiązania niekoniecznie może przynieść oczekiwane efekty. W takim przypadku zaakceptowanie rozwiązania gorszego może pomóc na wyjście z lokalnego optimum, na którym skupiły się obliczenia i da szansę na poszukiwanie innego rozwiązania, które, być może, będzie bardziej zbliżone do optimum globalnego.

2.6 Algorytmy genetyczne

Wspomnianym wcześniej podzbiorem algorytmów metaheurystycznych są algorytmy genetyczne i to na nich skupia się niniejsza praca oraz powiązana z nią implementacja. W tym podrozdziale zostanie opisana pokrótce idea i sposób działania algorytmów metaheurystycznych.

2.6.1 Idea i działanie algorytmów genetycznych

Cechą algorytmów metaheurystycznych jest to, że wiele z nich jest inspirowane naturalnymi procesami. W przypadku symulowanego wyżarzania dążono do odwzorowania procesu wyżarzania stosowanego w metalurgii. Algorytmy mrówkowe są inspirowane działaniem kolonii mrówek, a dokładniej - sposobem poszukiwania przez nie pożywienia. Inspiracją do ich tworzenia było zjawisko naturalnej ewolucji biologicznej (cf. [21]).

Algorytmy genetyczne możemy zaliczyć do algorytmów stochastycznych. Przy ich opisie używa się w dużej mierze słownictwa zapożyczonego bezpośrednio z genetyki. Trafnym wstępem do wyjaśnienia działania

tychże algorytmów będzie cytat z [17]: "[...] przenośnia leżąca u podstaw algorytmów genetycznych jest związana z ewolucją w naturze. W trakcie ewolucji każdy gatunek styka się z problemem lepszej adaptacji do skomplikowanego i zmiennego środowiska. "Wiedza", jaką gatunek zyskał, jest wbudowana w układ jego chromosomów". I tak - w przypadku algorytmów genetycznych autor będzie się posługiwał następującymi pojęciami zdefiniowanymi poniżej.

Definicja 2.10 *Osobnikiem lub chromosomem nazywa się jedno z rozwiązań stworzonych przez dany algorytm genetyczny podczas jednej iteracji.*

Definicja 2.11 *Populacją nazywa się zbiór rozwiązań (osobników) w danej iteracji algorytmu.*

Definicja 2.12 *DNA lub materiałem genetycznym nazywa się zbiór cech danego rozwiązania (osobnika).*

Definicja 2.13 *Operatorem genetycznym nazywamy pewna funkcję wprowadzającą określone zmiany w materiale genetycznym osobnika.*

Istotną rzeczą jest w tym miejscu sposób przechowywania informacji genetycznej każdego osobnika, czyli - wartości danego rozwiązania. W przypadku obrazów ciężko brać pod uwagę pojedynczą wartość liczbową. Rozważane tutaj będzie, jak wspomniano na początku tego rozdziału, reprezentowanie obrazu za pomocą skali RGBA lub jej pochodnej - skali szarości. Zatem obraz będzie pewną tablicą pikseli rozmiaru $n \times m$, gdzie każdemu pikselowi będzie przypisany kolor kodowany za pomocą czterech kanałów.

Ogólny schemat działania algorytmu metaheurystycznego można przedstawić w następujący sposób:

1: Stwórz w wybrany sposób populacje początkową o liczebności t.
2: Zadziałaj na osobnikach wybranymi operatorami genetycznymi.
3: Oceń każdego osobnika funkcją oceny.
4: Stwórz populację do kolejnego pokolenia (iteracji) algorytmu.
5: Powtarzaj 2. - 4. aż do warunku zakończenia działania algorytmu.

Wybieranie populacji początkowej

W tym etapie najczęściej tworzone są osobniki z losowo wybranym materiałem genetycznym, czyli, w przypadku reprezentacji komputerowej, losowymi ciągami bitowymi generowanymi za pomocą wybranej funkcji pseudolosowej.

Operatory genetyczne

Jak zdefiniowano powyżej, operatory genetyczne są pewnymi funkcjami, które mają na celu zmodyfikowanie materiału genetycznego danego osobnika. Opisane tutaj zostaną dwa najczęściej wykorzystywane operatory - operator mutacji i krzyżowania. Zostały one użyte w implementacji algorytmu replikacji obrazów powiązanej z niniejszą pracą.

Operator mutacji polega na wprowadzeniu jednej lub więcej losowych zmian w materiale genetycznym danego osobnika. W przypadku reprezentacji bitowej osobników możemy rozumieć mutację jako zamianę (negację) jednego lub więcej bitów w danym ciągu.

Operator krzyżowania ma na celu wymianę materiału genetycznego pomiędzy dwoma osobnikami z populacji i działa bardzo podobnie jak krzyżowanie organizmów w procesie naturalnej ewolucji. Znowu, w przypadku reprezentacji bitowej, operację krzyżowania należy rozumieć jako wybranie podciągów o ustalonej długości z każdego osobnika i zamiany ich miejscami w każdym z ciągów bitów.

Istotnym elementem algorytmu genetycznego jest sposób podejmowania decyzji o zastosowaniu danego operatora. W przypadku operatora mutacji najczęściej definiuje się pewnie prawdopodobieństwo zajścia mutacji w materiale genetycznym danego osobnika. W przypadku operatora krzyżowania ważny jest sposób wyboru osobników do krzyżowania - może być on zupełnie losowy lub mogą być wybrane do niego osobniki najmocniejsze. Odbywa się on na podobnych zasadach jak w przypadku wyboru osobników do każdego następnego pokolenia, co opisano poniżej.

Funkcja oceny

Tak, jak przedstawiono w definicji algorytmu metaheurystycznego, celem tegoż jest znalezienie rozwiązania (lub zbioru rozwiązań) z przestrzeni wszystkich możliwych rozwiązań danego problemu. Jako parametr wejściowy podawane jest jedno ze znanych rozwiązań i na jego podstawie algorytm próbuje przejść do innego rozwiązania, bliższego rozwiązaniu optymalnemu. Bardzo istotną rolę w tym procesie odgrywa funkcja oceny. Pozwala ona określić dopasowanie każdego osobnika do oczekiwanego rozwiązania. Na jej podstawie wybierane są osobniki do kolejnego pokolenia używanego w następnej iteracji algorytmu. Funkcja oceny może też odgrywać też istotną rolę przy sprawdzaniu warunku zakończenia działania algorytmu, co zostanie opisane w dalszej części tego podrozdziału. Może ona być również używana przy wyborze osobników do operacji krzyżowania.

Jako przykład można podać funkcję oceny dla wspomnianego problemu znajdowania wartości dla optimum lokalnego funkcji. Wówczas funkcja oceny będzie sprawdzać różnicę pomiędzy rzeczywistą wartością dla optimum, a wartością danego rozwiązania (osobnika).

Wybór osobników do nowej populacji

Po etapie oceny osobników, kolejnym etapem jest wybór osobników do nowego pokolenia. Może się to odbywać na kilka różnych sposobów i jest to definiowane przez autora danego algorytmu. Istnieje kilka popularnych dróg doboru osobników (należy tutaj doprecyzować, że osobniki najmocniejsze rozumiane są jako te, które zostały ocenione jako najlepsze (funkcja oceny ma dla nich najwyższą wartość), a osobniki najsłabsze jako te, dla których funkcja oceny przyjęła wartość najniższą):

1. **losowy wybór osobników** - w tym podejściu do następnego pokolenia osobniki wybierane są losowo (mogą się one powtarzać). Jest to sposób wyboru najbliższy temu naturalnemu.
2. **wybór osobników najmocniejszy** - do następnego pokolenia wybierane są tylko osobniki najmocniejsze
3. **mieszany** - wybierana jest pewna pula osobników najmocniejszych, pewna pula osobników najsłabszych i pewna pula, którym funkcja oceny przypasowała wartość środkową.

Warunek końca

Warunek zakończenia działania algorytmu jest definiowany w zależności od rozwiązywanego przezeń problemu. Może być on zdefiniowany jako pewna określona liczba N iteracji (pokoleń), po których algorytm powinien zakończyć swoje działanie. Może to być również pewne ograniczenie na dokładność rozwiązania, to jest - algorytm musi znaleźć rozwiązanie, które będzie odległe od optymalnego nie mniej niż pewien ϵ. Ze względu na fakt, że często stosując algorytmy metaheurystyczne nie jest znane optimum funkcji celu, przyjmuje się, że algorytm powinien zakończyć działanie, gdy w ustalonej liczbie iteracji nie znaleziono wyniku lepszego o co najmniej ϵ od poprzedniego najlepszego.

2.6.2 Motywacja do wyboru algorytmów genetycznych

Istnieje wiele podejść metaheurystycznych. Można wymienić wśród nich różne rodzaje takich algorytmów, m. in. mrówkowe, rojowe, genetyczne czy symulowanego wyżarzania. Każde z nich cechuje unikalne podejście do postawionego problemu. Autor niniejszej pracy zdecydował się na wybór algorytmów genetycznych.

Algorytmy genetyczne dają łatwą możliwość ich urównoleglenia. Etapy, na których stosuje się operatory genetyczne mogą przebiegać niezależnie dla poszczególnych osobników lub dla wybranych ich grup. Można znaleźć tutaj odniesienie do naturalnego procesu ewolucji. W przyrodzie wymiana i mutacje w materiale genetycznym są od siebie niezależne. W przypadku próby zastosowania np. algorytmów mrówkowych zastosowanie przetwarzania współbieżnego nie byłoby możliwe. W przypadku tego typu metaheurystyki kolejne rozwiązania są budowane w oparciu o poprzednie, niemożliwym więc byłoby równoległe przeszukiwanie przestrzeni rozwiązań (cf. [20]).

Kolejną ważną przyczyną, dla której autor zdecydował się na wykorzystanie algorytmów genetycznych jest fakt, że dają one zawsze pełne rozwiązania. W przypadku algorytmów mrówkowych do rozwiązania

ostatecznego, z oczywiście zadanymi wcześniej ograniczeniami, dochodzi się tworząc je z kolejnych, mniejszych rozwiązań. Nie można zatem przerwać działa programu w dowolnym momencie, a na rezultat trzeba czekać do osiągnięcia przez algorytm zadanego warunku końca. W przypadku algorytmów genetycznych można przerwać działanie programu w dowolnym momencie i otrzymać wygenerowane przez program rozwiązanie.

O wspomnianych innych rodzajach algorytmów metaheurystycznych, takich jak symulowane **symulowane wyżarzanie**, **algorytmy mrówkowe** czy **przeszukiwanie tabu** można przeczytać w [25, 18, 19].

2.7 Zastosowanie współbieżności w algorytmach genetycznych

Algorytmy genetyczne są dobrym przykładem algorytmów, w których można w nieskomplikowany sposób zastosować model obliczeń równoległych. Wielowątkowość może zostać zastosowana na tych etapach algorytmów, gdzie działa się na materiał genetyczny osobników operatorami genetycznymi. Zazwyczaj jedynym momentem, w którym wymagana jest synchronizacja pomiędzy wątkami jest ten, gdy wybierani są przedstawiciele bieżącej populacji, którzy przejdą do kolejnej iteracji algorytmu. Powodem stosowania przetwarzania równoległego w algorytmach genetycznych jest przede wszystkim skrócenie czasu ich działania. Jest to ich bardzo cenna cecha ze względu na fakt, że nie da się dokładnie określić czasu, w jakim znajdą one rozwiązanie, wyłączając przypadki, gdy jasno jest zdefiniowany limit czasowy lub ograniczenie na liczbę iteracji algorytmu.

Struktura projektu

W niniejszym rozdziale zostaną przedstawione wymagania odnośnie wymagań funkcjonalnych oraz niefunkcjonalnych implementowanej aplikacji

3.1 Wymagania niefunkcjonalne

W niniejszej sekcji opisane zostały wymagania niefunkcjonalne dotyczące badanego algorytmu i zawierające wszystkie ograniczenia oraz zewnętrzne czynniki mające wpływ na uruchamianie lub wydajność działania programu.

1. Szybkość działania programu uzależniona jest od liczby dostępnych procesorów. Jako procesor rozumie się tutaj pojedynczy wątek fizycznego procesora. Większa liczba takowych pozwala na równoległe przetwarzanie większej liczby osobników jednocześnie. Szczegóły implementacyjne zostały przedstawione w rozdziale 4.

2. Aplikacja jest zaprojektowana w taki sposób, aby możliwym było wygodne podmienianie jej poszczególnych modułów, które odpowiadają za jej poszczególne funkcjonalności bez potrzeby lub tylko z minimalną ingerencją w inne fragmenty kodu.

3. Ograniczenia na rozmiar plików wczytywanych przez program oraz przez niego generowanych i zapisywanych wynikają jedynie z ograniczeń języka, w którym algorytm zaimplementowano oraz ograniczeń systemu operacyjnego, na którym program jest uruchamiany.

3.2 Wymagania funkcjonalne

Poniżej zostały opisane wymagania funkcjonalne dla opisywanego algorytmu. Poniższy opis zawiera wszystkie oczekiwane cele, które powinien zrealizować program w trakcie swojego działania.

1. Program wczytuje obraz, który ma zostać zreplikowany lub, w przypadku, gdy zadany plik nie istnieje, zwraca błąd i kończy działania.

2. Program wczytuje parametry określające sposób działania algorytmu, które określają stosowaną skalę barw, liczbę iteracji algorytmu, rozmiar pojedynczego pokolenia, prawdopodobieństwo wystąpienia mutacji, dołączenie operatora krzyżowania oraz ścieżkę do pliku z obrazem, który powinien zostać zreplikowany.

3. Parametr określający ścieżkę dostępu do pliku z obrazem replikowanym jest zawsze wymagany. Dla pozostałych parametrów program powinien przyjmować wartości domyślne, w przypadku, gdy ich wartości nie zostały zdefiniowane przy uruchomieniu programu.

4. Parametry wywołania programu mogą również zostać podane w formie pliku konfiguracyjnego `JSON`. Ścieżka do tego pliku powinna zostać podana jako parametr wywołania programu.

5. Generowana jest populacja początkowa o określonej liczności i w określonej skali barw o rozdzielczości obrazu oryginalnego.

6. Dla każdego osobnika, w każdej iteracji, stosowany jest, z określonym przez parametr wejściowy prawdopodobieństwem, operator mutacji.

7. Każdy osobnik poddawany jest funkcji oceny i jest mu przypisywana wartość określająca jego *odległość* od obrazu oryginalnego. Funkcja powinna być zaprojektowana w taki sposób, aby możliwe było sortowanie osobników względem jej wartości dla każdego z nich.

8. Po zakończeniu każdej iteracji, o ile nie jest to ostatnia iteracja algorytmu, tworzona jest nowa populacja z osobników z bieżącego pokolenia.

9. Jeżeli wyspecyfikowano, jest stosowany parametr krzyżowania pomiędzy dwoma osobnikami polegający na wymianie materiału genetycznego dwóch wybranych osobników z nowopowstałej populacji.

10. Po zakończeniu działanie programu wyniki, w liczbie określonej przez jeden z parametrów wejściowych, zostają zapisane w formie plików `PNG`.

11. Aplikacja, na etapach, które umożliwiają zastosowanie takiego rozwiązania, powinna wykonywać swoje obliczenia równolegle. Powinna zostać zapewniona odpowiednia synchronizacja pomiędzy poszczególnymi wątkami.

3.3 Diagram przepływu

Na grafice 3.1 przedstawiono diagram przepływu dla projektowanej aplikacji. Diagram ten przedstawia poszczególne zadania wykonywane przez program, które zostały zawarte w wymaganiach funkcjonalnych opisanych w podrozdziale 3.2.

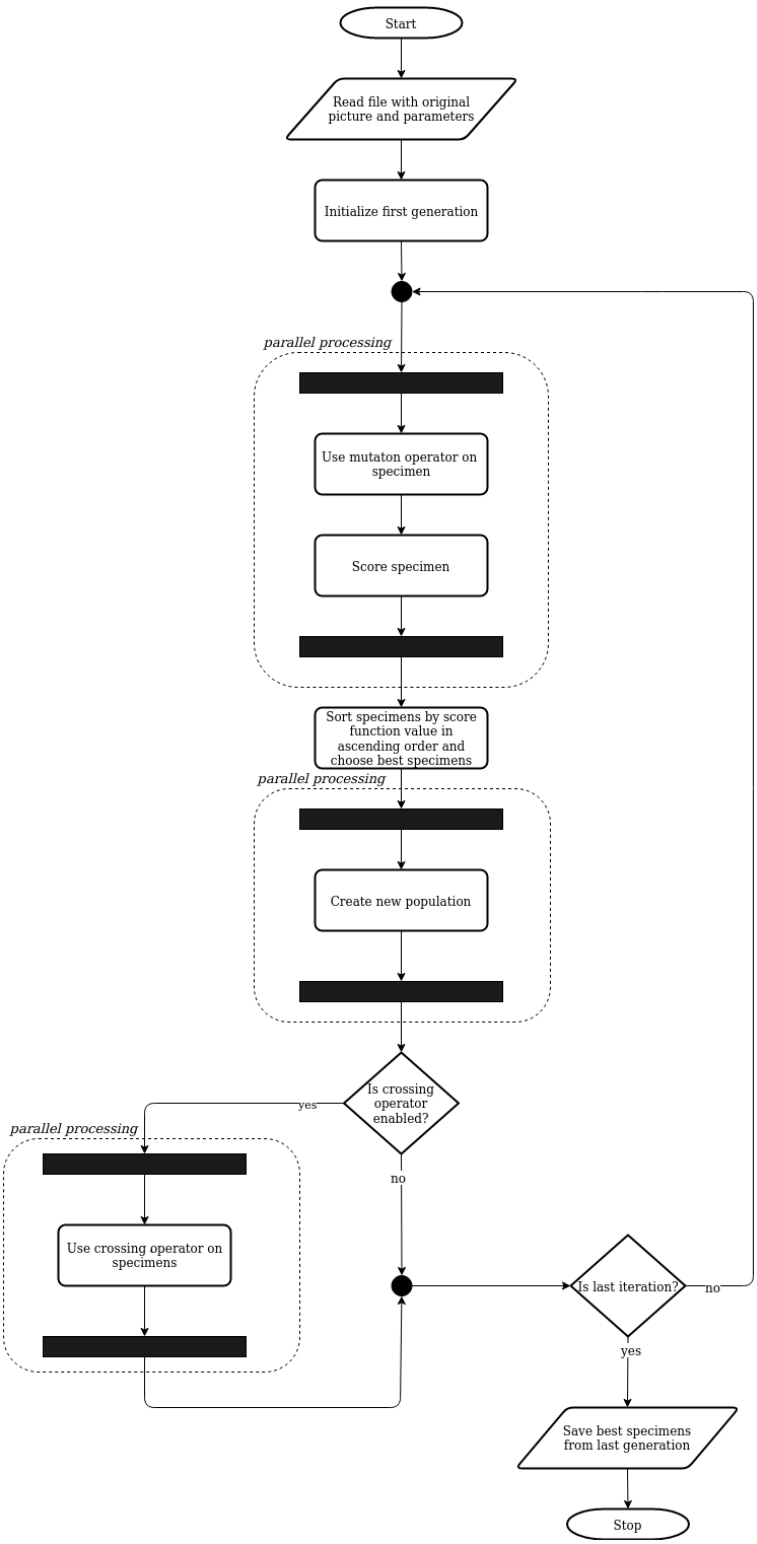
Program rozpoczyna swoje działanie od wczytania wymaganych danych. Następnie zostaje zainicjalizowane pierwsze pokolenie, które będzie przetwarzana w pierwszej iteracji algorytmu. Następnie na każdym osobników zostaje zastosowany operator mutacji, a następnie każdy z nich zostaje poddany funkcji oceny. Ten etap został urównoleglony, co zostało zaznaczone na diagramie. Następnie osobniki są sortowane względem wartości funkcji celu i wybierane są osobniki, które utworzą następne pokolenie - k najlepszych osobników, czyli takich, dla których funkcja celu przyjęła wartości najmniejsze. Parametr k jest ustawiany przez jeden z parametrów wejściowych. Następnie najlepsze osobniki są kopiowane w taki sposób, aby liczność nowego pokolenia była zgodna z wartością podaną przez użytkownika w postaci parametru. Ta część działania programu również jest wykonywana współbieżnie. Po tym etapie, w przypadku gdy operator krzyżowania został dozwolony przez użytkownika, następuje krzyżowanie osobników (ten etap również jest wykonywany równolegle), a w przeciwnym wypadku program przechodzi do kolejnej iteracji. Po czasie, gdy program osiągnie zadaną liczbę iteracji, zostaje wybranych k najlepszych obrazów z ostatniego pokolenia i zostają one zapisane w formie plików `PNG`. Po wykonaniu tej operacji program kończy działanie. Szczegóły dotyczące implementacji zostały przedstawione w rozdziale 4.

3.4 Komponenty aplikacji

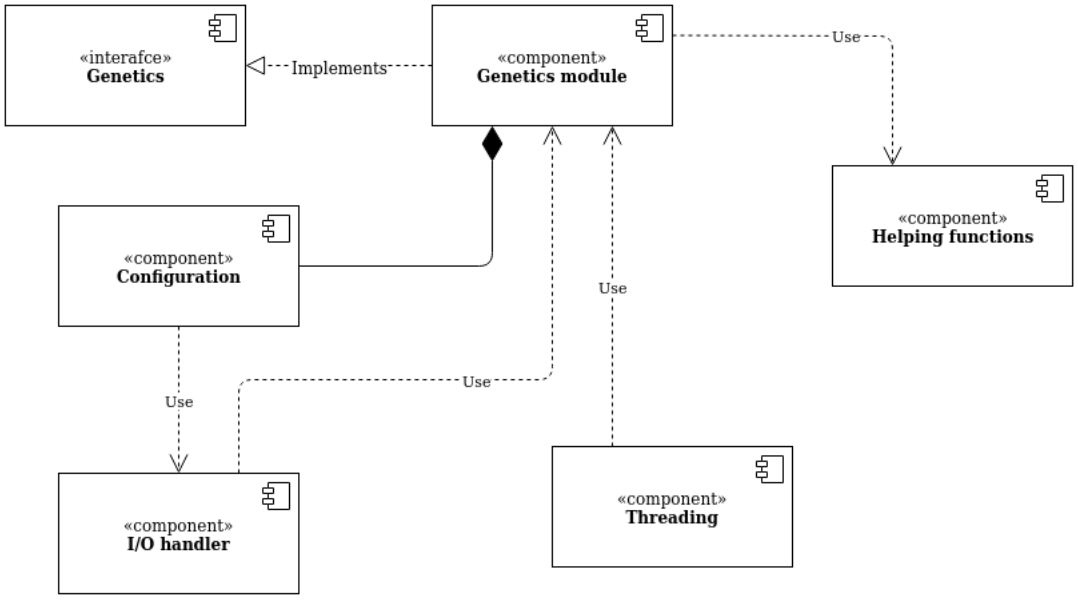
Prezentowana w pracy aplikacja składa się z różnych komponentów, gdzie każdy z nich spełnia określoną funkcjonalność niezbędną dla poprawnego działania aplikacji. W prezentowanej implementacji można wyróżnić następujące komponenty, które zostały przedstawione również na diagramie 3.2.

3.4.1 Moduł konfiguracji

Moduł odpowiedzialny za konfigurację algorytmu wczytuje parametry, które pozwalają na zdefiniowanie sposobu działania programu i przetrzymuje je w pamięci operacyjnej w taki sposób, aby były one dostępne dla pozostałych modułów aplikacji.



Rysunek 3.1: Diagram przepływu dla aplikacji. Program, po rozpoczęciu działania wczytuje odpowiednie pliki i parametry konfiguracyjne oraz inicjalizuje pierwsze pokolenie. Następnie stosowane są operator mutacji oraz funkcja oceny. Dalej tworzona jest nowa populacja i, jeżeli konfiguracja to przewiduje, stosowany jest operator krzyżowania. Program kończy działanie po określonej liczbie iteracji i zapisuje wyniki do odpowiednich plików po czym kończy działanie. Części algorytmu, które zostały urównoleglone zostały odpowiednio opisane na diagramie. Szczegółowy jego opis znajduje się w rozdziale 3.3.



Rysunek 3.2: Diagram komponentów aplikacji. Każdy z widocznych na grafice komponentów spełnia odpowiednią funkcjonalność, gdzie każdy z nich został krótko opisany w odpowiednim podrozdziale: interfejs *Genetics* oraz komponent *Genetics module* odpowiedzialne za logikę genetyki algorytmu opisane są w podrozdziale 3.4.3; *Helping funcions* zawierajacy metody pomocnicze dla modułu genetyki opisany jest w podrozdziale 3.4.4; *Configuration* odpowiedzialny za przechowywanie zdefiniowanej przez użytkownika konfiguracji algorytmu opisany jest w podrozdziale 3.4.1; *Threading* odpowiedzialny za obsługę współbieżności w aplikacji opisany jest w sekcji 3.4.5; moduł *I/O handler* odpowiedzialny za obsługę wczytywanych i zapisywanych plików opisany jest w podrozdziale 3.4.2.

3.4.2 Moduł obsługi wejścia/wyjścia

Moduł jest odpowiedzialny za wczytywanie plików (obraz, który ma zostać zreplikowany) oraz zapisanie rozwiązań wygenerowanych przez program po ostatniej iteracji algorytmu. Moduł obsługuje i wyświetla błędy związane z obsługą plików.

3.4.3 Moduł genetyki

Moduł obsługuje i definiuje działanie operatorów genetycznych (mutacji i krzyżowania) oraz odpowiada za wyliczanie funkcji celu dla każdego osobnika. Moduł ten jest *wymienny* i może zostać przedefiniowany przez użytkownika przy warunku spełniania zadanego przez autora interfejsu.

3.4.4 Moduł funkcji pomocniczych

Komponent zawierający funkcje pomocnicze służące, między innymi, do sortowania osobników względem podanej wartości (w tym przypadku - względem funkcji celu), wyliczania wartości koloru dla pikseli dla operator mutacji, wyliczania wartości pomocniczych dla modułu genetyki. Szczegółowa specyfikacja tego modułu jest zawarta w dokumentacji technicznej kodu.

3.4.5 Moduł obsługi współbieżności

Moduł odpowiedzialny za obsługę zadań, które mogą zostać obsłużone współbieżnie. Zostały one wyszczególnione na diagramie 3.1. Jego zadaniem jest rozdzielenie zadań pomiędzy poszczególne wątki oraz synchronizacja momentu ich zakończenia.

Implementacja

Na podstawie powyższych definicji i opisów dotyczących algorytmów metaheurystycznych można przedstawić algorytm genetyczny użyty w niniejszej pracy do replikacji obrazów. Algorytm został tutaj przedstawiony jedynie w sposób opisowy. Z implementacją opatrzoną szczegółowymi komentarzami można zapoznać się w kodzie dołączonym do pracy.

4.1 Wykorzystywane technologie

W części implementacyjnej pracy inżynierskiej zdecydowano się na oprogramowanie zaprojektowanego algorytmu w języku `Go`. Język ten, od roku 2009, w którym została wydana jego pierwsza wersja, bardzo dynamicznie się rozwija. Został on stworzony przez firmę Google i zaprojektowany z myślą, by zostać współczesnym odpowiednikiem języka `C`. Był on tworzony we współpracy z jednym z twórców `C` - Brianem W. Kernighanem. Jedną z głównych cech języka jest jego zastosowanie do tworzenia oprogramowania współbieżnego, ponieważ posiada on bardzo rozbudowany model współbieżności. Ze względu na fakt, że algorytmy genetyczne dają się w łatwy sposób urównoleglić, autor zdecydował się na implementację pracy w tym właśnie języku. Całość została zaimplementowana jedynie przy użyciu biblioteki standardowej tego języka ze z względu na jej rozbudowane wsparcie dla operacji na obrazach. Model współbieżności w języku `Go` został dokładniej opisany w podrozdziale 4.5.

Ponadto, w pracy użyto narzędzia `Docker` w celu umożliwienia uruchomienia aplikacji na dowolnym systemie operacyjnym. Pozwala ono uniknąć instalowania wszystkich niezbędnych do uruchomienia aplikacji narzędzi lokalnie, na komputerze użytkownika, a uruchamia program w środowisku wirtualnym z wykorzystaniem tzw. *kontenerów* (cf. [7]).

4.2 Parametry algorytmu

Parametry wejściowe

1. **ścieżka do obrazu oryginalnego** - ścieżka do obrazu wzorcowego w formacie `JPG` lub `PNG`.
2. **skala kolorów** - skala kolorów, w której reprezentowane będą poszczególne osobniki (RGBA lub skala szarości) i w której zapisany jest obraz wejściowy.
3. **liczba iteracji** - ograniczenie na liczbę iteracji (pokoleń) algorytmu.
4. **liczba osobników w pokoleniu** - liczba osobników w pojedynczej iteracji (pokoleniu).
5. **liczba najlepszych osobników** - liczba najlepszy osobników określa ile najlepszych rozwiązań (obrazów) zostanie zwrócone po zakończeniu działania algorytmu. Parametr ten wyznacz również ile najlepszych osobników z danej iteracji zostanie wykorzystane do utworzenia kolejnego pokolenia.
6. **operator krzyżowania** - parametr definiujący, czy powinien zostać zastosowany operator krzyżowania. Jeżeli nie jest ustawiony, wówczas stosowany jest jedynie operator mutacji.
7. **prawdopodobieństwo mutacji** - parametr definiujący prawdopodobieństwo wystąpienia mutacji w materiale genetycznym każdego osobnika, który przyjmuje wartość rzeczywistą z zakresu $[0, 1]$.

Parametry wyjściowe

Algorytm zwraca (w postaci plików `PNG`) najlepsze osobniki z ostatniego pokolenia algorytmu. Liczba zwracanych osobników definiowana jest przez jeden z parametrów wejściowych algorytmu.

4.3 Opis algorytmu

W tym rozdziale, w poszczególnych jego sekcjach, opisane zostały poszczególne etapy algorytmu zaprojektowanego algorytmu oraz sposób ich implementacji. 1 przedstawia pseudokod algorytmu.

Algorithm 1 Pseudokod algorytmu replikacji

1: Stwórz populację początkową o liczebności t.
2: Użyj operatora mutacji na każdym z osobników z prawdopodobieństwem p_m.
3: Dokonaj oceny każdego osobnika i posortuj je rosnąco malejąco wartości funkcji oceny.
4: Wybierz do następnego pokolenia
5: Zastosuj operator krzyżowania
6: Powtarzaj kroki 2. - 5. aż do momentu osiągnięcia warunku końca algorytmu.

4.3.1 Wybór populacji początkowej

W przypadku opisywanego algorytmu replikacji stworzenie populacji początkowej polega na wygenerowaniu t czarnych obrazów (każdy z kanałów RGB przyjmuje wówczas wartość minimalną 0) o rozmiarze odpowiadającym rozmiarowi obrazu wzorcowego. Następnie, na każdym z wygenerowanych obrazów umieszczany jest losowy [1] kształt (prostokąt) o losowych wymiarach i losowym położeniu na obrazie. Każdy z kształtów jest półprzeźroczysty co ułatwia późniejsze nakładanie kolejnych kolorów.

Reprezentacja obrazu w skali RGBA w języku Go

W języku `Go` obraz zapisany za pomocą skali RGBA jest reprezentowany przez następującą strukturę przedstawioną w kodzie źródłowym 4.1.

```
type RGBA struct {
    Pix    []uint8
    Stride int
    Rect   Rectangle
}
```

Kod źródłowy 4.1: Struktura reprezentująca obraz w skali RGBA w języku `GO`

Przechowywana jest w niej tablica pikseli obrazu (`Pix`) i dla każdego piksela zachowana jest kolejność kanałów, tj. R, G, B, A. Każdy kanał każdego piksela jest zakodowany za pomocą zmiennej typu `uint8` czyli za pomocą 8-bitowej liczby całkowitej bez znaku. Oznacza to, że każdy kanał może przyjmować wartości od 0 do 255 (256 różnych wartości). Parametr `Stride` oznacza odległość (w bajtach) pomiędzy dwoma sąsiadującymi wertykalnie pikselami, czyli długość jednego poziomego rzędu pikseli. `Rect` to struktura zawierająca wymiary obrazu.

4.3.2 Operator mutacji

Operator mutacji ma na celu wprowadzenie losowej zmiany w materiale genetycznym każdego osobnika. Na początku działania funkcji odpowiedzialnej za dokonanie mutacji na wybranym osobniku wybierana jest rzeczywista liczba losowa z przedziału $[0, 1]$. Jeżeli jest mniejsza od prawdopodobieństwa wystąpienia mutacji, to w materiale genetycznym bieżącego osobnika wprowadzana jest mutacja. Polega ona na dodaniu losowego kształtu na obrazie. Pojawia się tutaj problem w jaki sposób wybrać kolor obszaru, w którym nakładają się

[1] Słowo losowy w odniesieniu do kształtu zawsze będzie oznaczać prostokąt o losowym kolorze, wymiarach i losowym położeniu na obrazie

na siebie dwa lub więcej kształtów o różnych kolorach. Przyjęto zasadę, że wówczas jako kolor danego piksela na opisanym obszarze przyjmuje się średnią z wartości wszystkich kanałów koloru piksela na obrazie i piksela w nakładanym w procesie mutacji kształcie.

4.3.3 Funkcja oceny

Funkcja oceny ma na celu przyznanie każdemu z osobników wartości liczbowej, która, odpowiednio wyliczona, pozwoli określić przystosowanie osobnika (jakość jego genotypu, czyli, w języku genetyki, zbioru genów) i później wyznaczyć, które osobniki mogą zostać uznane za najmocniejsze, które za średnio przystosowane, a które zaliczone zostaną do zbioru osobników słabo przystosowanych.

W prezentowanym w pracy algorytmie funkcja oceny wylicza różnicę pomiędzy wartościami na każdym z kanałów dla każdego piksela obrazu oryginalnego oraz ocenianego osobnika i sumuje wartości bezwzględne wyników. Jest to implementacja wzoru 2.2. Osobnik, dla którego wartość funkcji oceny jest najmniejsza, będzie uznany za osobnik najmocniejszy, ponieważ najmniej *różni się* od obrazu oryginalnego. Analogicznie - osobnik, dla którego wartość funkcji oceny będzie największa zostanie sklasyfikowany jako osobnik najsłabszy, ponieważ najbardziej różni się od obrazu oryginalnego.

4.3.4 Wybór osobników do kolejnej iteracji algorytmu

Do każdej kolejnej iteracji algorytmu wybierane są osobniki najmocniejsze względem wartości funkcji oceny w liczbie określonej przez użytkownika. Może to zapewnić szybsze zbieganie algorytmu do poszukiwanego rozwiązania, jak również powoduje mniejszą różnorodność w materiale genetycznym potomstwa.

4.3.5 Operator krzyżowania

Celem operatora krzyżowania jest wymiana informacji genetycznej pomiędzy dwoma osobnikami. Podobnie, jak w przypadku operatora mutacji, można w łatwy sposób wskazać analogię tej operacji do naturalnego procesu krzyżowania (rozmnażania się organizmów), podczas którego dochodzi do wymiany materiału genetycznego rodziców i stworzenia nowego genotypu potomka.

Operator krzyżowania w przypadku opisywanego algorytmu replikacji wybiera pewien prostokątny obszar na jednym z dwóch osobników przeznaczonych do krzyżowania. Następnie materiał genetyczny jednego osobnika jest wymieniany z drugim, tj. wybrany obszar z pierwszego obrazu jest kopiowany w dokładnie to samo miejsce na obrazie drugim i w ten sam sposób wybrany obszar z drugiego osobnika jest kopiowany w odpowiadające miejsce na osobniku pierwszym. Osobniki początkowe można określić mianem rodziców, a osobniki powstałe w wyniku krzyżowania określa się osobnikami potomnymi.

4.4 Obliczenia równoległe a algorytm replikacji

W implementacji algorytmu replikacji zastosowano model przetwarzania równoległego w celu przyspieszenia obliczeń. Współbieżnie wykonywane są etapy algorytmu, podczas których działa się na osobniki operatorem mutacji i poddaje ocenie oraz proces krzyżowania osobników wybranych do nowej populacji. Synchronizacja pomiędzy wątkami dokonywana jest w momencie tworzenia nowej populacji. Proces odbywa się w ten sam sposób, jaki opisano w rozdziale niniejszej pracy poświęconym analizie problemu, a szczegóły implementacyjne, w raz z odpowiednim komentarze, dostępne są w załączonym do pracy kodzie.

4.4.1 Dekompozycja problemu

W celu przystosowania algorytmu genetycznego do przetwarzania równoległego, należy na początku dokonać dekompozycji problemu (cf. [15]). W przypadku algorytmów genetycznych najlepiej sprawdzającym się podejściem będzie dokonanie dekompozycji względem danych. W przypadku algorytmu genetycznego za zbiór danych przyjmuje się zbiór wszystkich osobników w danym pokoleniu. Każdemu z dostępnych procesorów zostaje wtedy przydzielona pewna pula osobników, na których zostanie zastosowany operator genetyczny.

Oprócz działania operatorami genetycznymi na poszczególne chromosomy, można łatwo zauważyć, że równolegle może być również dokonywana ocena osobników, ponieważ polega ona jedynie na porównaniu danego osobnika z wzorcowym rozwiązaniem.

4.4.2 Synchronizacja

Dla algorytmów genetycznych wymagana jest synchronizacja pomiędzy działającymi wątkami w celu oceny osobników i wybrania nowej populacji. Po odebraniu przez jednostkę zarządzającą wątkami informacji o wykonaniu przez poszczególne procesory wszystkich zadań, to jest: zadziałaniu na osobniki operatorami genetycznymi i ocenie dopasowania poszczególnych osobników, następuje wybór osobników do kolejnej iteracji algorytmu. Ten etap algorytmu genetycznego nie daje się przystosować do przetwarzania równoległego poprzez fakt, iż muszą być rozważone wszystkie osobniki z bieżącej populacji i utworzona musi zostać populacja kolejna. Ze względu na konieczność porównywania ze sobą wartości funkcji oceny dla każdego osobnika urównoleglenie tego kroku algorytmu jest niemożliwe.

4.5 Model współbieżności w języku Go

4.5.1 Gorutyny

Jedną z cech języka `Go` jest bardzo dobrze rozwinięty model współbieżności. Jest to jedno z głównych zastosowań języka `Go`. Główną jednostką używaną w implementacjach wykorzystujących współbieżność w `Go` jest *gorutyna* (ang. *gorutine*) (cf. [3]). Jest to reprezentacja pojedynczego zadania , której główną cechą jest jej niewielki rozmiar w pamięci maszyny, co pozwala na tworzenie znacznej ich ilości z niewielkim zużyciem zasobów. Gorutyna nie powinna być utożsamiana z pojedynczym wątkiem, bowiem w obrębie jednego wątku można utworzyć wiele gorutyn.

W języky `Go` są dwa podejścia do problemu przetwarzania współbieżnego. Jeden z zaimplementowanych modeli oparty jest na komunikację pomiędzy gorutynami za pomocą kanałów (ang. *channel*) (cf. [2]). Odrębnym podejściem jest użycie do synchronizacji dostępu do pamięci klasycznych mutexów (cf. [1]). Kanały są wykorzystywane przede wszystkim do prowadzenia komunikacji pomiędzy poszczególnymi gorutynami. Mutexy służą do synchronizacji dostępu do określonych części kodu przesz poszczególne gorutyny (cf. [6]).

4.5.2 Model oparty o komunikację za pomocą kanałów

W modelu opierającym się na wykorzystaniu kanałów komunikacja pomiędzy poszczególnymi gorutynami odbywa się za struktur, które są pewnym rodzajem strumieni (ang. *pipe*), za pomocą których poszczególne gorutyny mogą przekazywać pomiędzy sobą informacje. Istnieją dwa rodzaje kanałów w języku `Go` - buforowane i niebuforowane. W przypadku tych pierwszych informacja jest wysyłana przez jedną z gorutyn i bezpośrednio odbierana przez drugą, gdy tylko obie są w stanie odbierać lub wysyłać dane. Kanały buforowane, w odróżnieniu od kanałów niebuforowanych, posiadają dodatkowo zaimplementowaną strukturę kolejki o rozmiarze określanym przez użytkownika. Pozwala to na wysłanie do kanału większej liczby danych przez jedną lub więcej gorutyn korzystających z danego kanału i odebranie wysłanych informacji przez odpowiednią gorutynę. W tym przypadku dane te nie muszą być odebrane natychmiast i strona odbierająca nie musi w danym momencie wykazywać gotowości do odebrania informacji w momencie wysłania ich przez inną gorutynę.

4.5.3 Model oparty o mutexy

Innym sposobem pozwalającym na synchronizację pomiędzy poszczególnymi gorutynami jest wykorzystanie mutexów, modelu znanego między innymi systemów typu Unix. Wykorzystuje on struktury, które pozwalają na zablokowanie dostępu do określonej struktury (zmiennej) przez daną gorutynę na czas, gdy jest ona przez nią modyfikowana. Nie służą one zatem do komunikacji pomiędzy gorutynami, ale do synchronizacji dostępu do poszczególnych danych.

Najczęściej, przy tworzeniu oprogramowania współbieżnego w języku `Go`, wykorzystuje się i łączy oba wymienione modele współbieżności. Stosowne przykłady z obszernym wyjaśnieniem można znaleźć w przytaczanej wcześniej dokumentacji języka.

4.5.4 Model współbieżności w języku Go a implementacja algorytmu

W tym rozdziale opisane rozwiązania oferowane przez model przetwarzania równoległego języka `Go` zastosowane w implementacji oraz uzasadniono wybór takiego rozwiązania i omówiono jego wydajność.

Wait groups

Zgodnie z tym, co opisano w podrozdziale 4.4, w implementacji omawianego w pracy algorytmu zdecydowano się na wprowadzenie równoległego modelu obliczeń. Model współbieżności obecny w języku `Go` pozwala na implementowanie bardzo złożonych i wydajnych systemów. Fakt, że gorutyny, są *lekkie* w sensie ilości zużytych zasobów urządzenia, umożliwia tworzenie wielu ich instancji z jednoczesnym niewielkim wzrostem zużycia zasobów.

W problemie urównoleglenia omawianego algorytmu zdecydowano się na wykorzystanie udostępnianej przez `Go` metody nazywanej *wait groups* (cf. [4]). W przypadku zaprojektowanego algorytmu genetycznego, na etapach, które umożliwiają przetwarzanie równoległe, nie jest wymagana żadna komunikacja pomiędzy poszczególnymi wątkami. Każdy wątek dostaje określone zadanie do wykonania. Wymagane jest jedynie zsynchronizowanie momentu zakończenia pracy przez każdą z utworzonych wcześniej gorutyn. Mechanizm wait groups umożliwia utworzenie wielu instancji gorutyn, gdzie każda z nich wykonuje określone zadanie (stosuje na zadanych osobnikach odpowiednie operatory genetyczne i wylicza dla nich wartość funkcji celu) i monitorowanie stanu, w jakim w danym momencie się znajdują, a dokładniej - ile z nich w danym momencie zakończyło już swoją pracę, a ile nadal jest aktywnych i wykonuje jeszcze obliczenia.

```
1  package main
2
3  import (
4      "fmt"
5      "sync"
6      "time"
7  )
8
9  func worker(id int, wg *sync.WaitGroup) {
10     fmt.Printf("Worker %d starting\n", id)
11
12     time.Sleep(time.Second)
13
14     fmt.Printf("Worker %d done\n", id)
15     wg.Done()
16 }
17
18 func main() {
19     var wg sync.WaitGroup
20
21     fmt.Println("Workers starting")
22     for i := 0; i < 5; i++ {
23         wg.Add(1)
24         go worker(i, &wg)
25     }
26
27     wg.Wait()
28     fmt.Println("All workers done")
29 }
```

Kod źródłowy 4.2: Mechanizm wait groups dostępny w bibliotece `sync` języka `Go`. Przedstawiany tutaj kod i jego działanie zostało opisane w sekcji 4.5.4

W przykładzie (kod źródłowy 4.2) zaprezentowano prosty przykład wykorzystywania opisywanego mechanizmu (zaczerpnięty on został z [4]). Zostaje zadeklarowana zmienna `var wg sync.WaitGroup` (struktura `WaitGroup` pochodzi z biblioteki `sync` (dokumentacja: [1]]). Następnie w pętli zostaje utworzonych 5 gorutyn, gdzie każda wykonuje kod zadeklarowanej wcześniej funkcji `worker`. Przy tworzeniu każdej z nich zostaje wywołana metoda `wg.Add(1)`. Zwiększa ona licznik aktwynych gorutyn wewnątrz omawianej struktury (zmienna `wg`). Jednocześnie, do przy każdym wywołaniu funkcji `worker` zmienna `wg` zostaje do niej przekazana jako

jeden z argumentów wywołania poprzez referencję. Po zakończeniu wykonywania pętli `for` zostaje wywołana w głównym wątku metoda `wg.Wait()`, która oczekuje na zakończenie działania przez wszystkie gorutyny. Każda z gorutyn (w tym przypadku - każda z instancji funkcji `worker`) *informuje* o zakończeniu swojego działania poprzez wywołanie metody `wg.Done()`, która zmniejsza licznik aktywnych w danym momencie gorutyn wewnątrz struktury `sync.WaitGroup`. W momencie, gdy licznik osiągnie wartość 0, metoda `wg.Wait()` kończy swoje działanie i w głównym wątku zostaje wypisana informacja o zakończeniu działania przez wszystkie utworzone wcześniej w pętli gorutyny. Poniżej zaprezentowano komunikaty wypisywane przez powyższy program na standardowe wyjście (`?` oznaczono komendy wpisywane przez użytkownika, a `>` komunikaty wypisywane przez program na standardowe wyjście).

```
? go run main.go
> Workers starting
> Worker 4 starting
> Worker 0 starting
> Worker 3 starting
> Worker 1 starting
> Worker 2 starting
> Worker 2 done
> Worker 1 done
> Worker 4 done
> Worker 0 done
> Worker 3 done
> All workers done
?
```

Zastosowane rozwiązanie a wydajność

W 4.5.1 opisano gorutynę jako strukturę *lekką* w kontekście zużywanych zasobów na urządzeniu, na którym został uruchomiony program. Ponadto, w prezentowanej implementacji algorytmu replikacji nie jest wymagana żadna komunikacja pomiędzy pojedynczymi, wykonywanymi współbieżnie zadaniami. Istotny jest jedynie ustalenie momentu, w którym wszystkie uruchomione gorutyny zakończą swoje działanie. W związku z taką specyfikacją przedstawionego algorytmu, autor pracy zdecydował się na wykorzystanie do urównoleglenia obliczeń prezentowanego w poprzedzającej sekcji mechanizmu *wait groups*. Liczba określonych gorutyny odpowiada albo maksymalnej liczbie osobników w populacji (zadanie polegające na zastosowaniu operatora mutacji i wyliczeniu funkcji oceny), albo połowie tej wartości (działanie operatorem krzyżowania). Ze względu na cechy tych struktur możliwym było zrzucenie odpowiedzialności na zarządzanie zasobami na mechanizmy zastosowane w języku `Go`.

Instrukcja użytkownika

5.1 Wymagania systemowe

Program był testowany na systemach operacyjnych `Ubuntu 19.10 Eoan Ermine` oraz `macOS 10.15 Catalina` oraz przy użyciu języka `Go` w wersji `1.13`. Zaleca się uruchamianie programu na systemach *unix-like* (np. `macOs`, `Ubuntu`, `Debian`, `ArchLinux`) oraz przy użyciu kompilatora `Go` co najmniej w wersji `1.11` ze względu na konieczność obsługi `Go modules`. Oprogramowanie nie zostało przetestowane na systemie operacyjnym `Windows` i nie jest gwarantowane jego działanie na tymże.

5.2 Parametry programu

Tak jak opisano w rozdziałach poświęconych strukturze oraz implementacji algorytmu, program przyjmuje na wejściu zbiór argumentów pozwalających na konfigurację zachowania algorytmu. Poniżej zostanie przedstawiony sposób ich definiowania oraz uruchamiania programu.

5.2.1 Wywołanie konsolowe

W przypadku, gdy program jest uruchamiany poprzez konsolę systemową parametry zostają zdefiniowane przez następujące flagi wywołania:

- `-best` - flaga określa liczbę najlepszych osobników, które zostaną wybrane do tworzenia każdej kolejnej populacji oraz zapisane w ostatniej iteracji algorytmu, jako pliki wynikowe. Flaga ta powinna być liczbą typu `uint` (wartość domyślna: `10`).
- `-generation-size` - flaga określa rozmiar pojedynczego pokolenia w jednej iteracji algorytmu. Przyjmuje wartość typu `uint` (wartość domyślna: `200`).
- `-generations` - flaga określająca maksymalną liczbę iteracji algorytmu. Przyjmuje wartość typu `uint` (wartość domyślna: `1000`).
- `-gray-scale` - flaga określająca, czy algorytm powinien działać na obrazach w skali szarości. Flaga przyjmuje wartość typu `bool` (wartość domyślna: `false`).
- `-image-dir` - ścieżka do oryginalnego obrazu, który ma zostać zreplikowany przez algorytm. Flaga przyjmuje wartość typu `string`. Parametr ten jest wymagany, aby uruchomić program.
- `-mutation-chance` - flaga określająca prawdopodobieństwo wystąpienia mutacji. Flaga przyjmuje wartość typu `float` (wartość domyślna: `0.2`).
- `-with-crossing` - flaga określająca, czy powinien zostać zastosowany operator krzyżowania. Flaga przyjmuje wartość typu `bool` (wartość domyślna: `false`).
- `-from-file` - flaga przyjmuje ścieżkę do pliku konfiguracyjnego. Jeżeli została ona ustawiona, program ignoruje pozostałe flagi, jeśli są, i czyta parametry ze wskazanego pliku. Flaga przyjmuje wartość typu `string`.

- `-help` - jeżeli flaga jest ustawiona to zostają wyświetlone informacje na temat dostępnych parametrów wywołania, sposobu ich użycia oraz domyślnych wartości.

Program może zostać skompilowany za pomocą polecenia `go build main.go`, a następnie uruchomiony `./main`, lub bezpośrednio skompilowany i uruchomiony za pomocą polecenia `go run main.go`. Po zakończeniu działania programu można dokonać czyszczenia plików wygenerowanych przez kompilator za pomocą polecenia `go clean`. Wszystkie polecenia należy wywołać w głównym katalogu zawierającym plik `main.go`.

Przykładowe wywołania

```
#!/bin/bash
go build main.go
./main -image-dir ./images/image_in_gray_scale.png \
        -best 5 \
        -generation-size 50 \
        -generations 5000 \
        -gray-scale \
        -mutation-chance 0.6 \
        -with-crossing
go clean

go run main.go -image-dir ./images/image_in_rgba_scale.png
```

5.2.2 Plik konfiguracyjny

Alternatywnym rozwiązaniem jest przekazanie parametrów wywołania przez plik konfiguracyjny. Aby to zrobić, należy użyć flagi `-from-file` i podać ścieżkę do tego pliku. Konfiguracja powinna zostać zapisana w formacie `JSON`. Dostępne są następujące atrybuty, które mogą zostać przekazane poprzez plik konfiguracyjny.

Przykładowy plik konfiguracyjny

```
{
    "NumOfIterations": 2000,
    "SizeOfGeneration": 50,
    "PathToImage": "./images/example_image.jpg",
    "NumOfBest": 4,
    "MutationChance": 0.8,
    "GrayScale": false,
    "WithCrossing": true
}
```

Przykładowe uruchomienie programu z plikiem konfiguracyjnym

```
#!/bin/bash

go run main.go -from-file ./config.json
```

Każdy parametr ma swój odpowiednik w opisanych w poprzedniej części rozdziału fladze wywołania programu.

- `NumOfBest` - parametr określa liczbę najlepszych osobników, które zostaną wybrane do tworzenia każdej kolejnej populacji oraz zapisane w ostatniej iteracji algorytmu, jako pliki wynikowe. Parametr ten powinien być liczbą typu `uint` (wartość domyślna: `10`).

- `SizeOfGeneration` - parametr określa rozmiar pojedynczego pokolenia w jednej iteracji algorytmu. Przyjmuje wartość typu `uint` (wartość domyślna: `200`).

- `NumOfIterations` - parametr określający maksymalną liczbę iteracji algorytmu. Przyjmuje wartość typu `uint` (wartość domyślna: `1000`).
- `GrayScale` - parametr określający, czy algorytm powinien działać na obrazach w skali szarości. Flaga przyjmuje wartość typu `bool` (wartość domyślna: `false`).
- `PathToImage` - ścieżka do oryginalnego obrazu, który ma zostać zreplikowany przez algorytm. Parametr przyjmuje wartość typu `string`.
- `MutationChance` - parametr określający prawdopodobieństwo wystąpienia mutacji. Przyjmuje on wartość typu `float` (wartość domyślna: `0.2`).
- `WithCrossing` - Parametr określający, czy powinien zostać zastosowany operator krzyżowania. Przyjmuje wartość typu `bool` (wartość domyślna: `false`).

W przypadku, gdy któryś z parametrów nie zostanie wyspecyfikowany w pliku, przyjmie on wartość domyślną. Ponownie, jak w przypadku użycia flag wywołania, koniecznym jest zdefiniowanie parametru `PathToImage` (odpowiednik `-image-dir`).

5.3 Docker

Program może zostać uruchomiony z użyciem oprogramowania `Docker`. W tym celu zaleca się instalację pakietu `Docker` w ostatniej dostępnej wersji (testowano na wersji `19.03.5`). W celu uruchomienia programu należy użyć komendy `docker-compose up` w folderze zawierającym kod źródłowy programu. W przypadku uruchamiania programu w ten sposób, konfiguracja programu musi zostać podana poprzez plik konfiguracyjny `config.json` znajdujący się w głównym katalogu z kodem. W przypadku potrzeby modyfikacji ustawień Dockera (w tym - ścieżki dostępowej do pliku konfiguracyjnego), należy dokonać zmian w pliku `docker-compose.yaml` zgodnie z dokumentacją tego oprogramowania [cf. [5]].

5.4 Modyfikacja modułu genetyki algorytmu

Moduł odpowiedzialny za genetykę algorytmu został zaimplementowany jako interfejs w języku `Go` i jest on *wymienny*, to jest możliwe jest wykonanie własnej jego implementacji pod warunkiem spełniania określonego interfejsu przedstawionego w 5.1.

```
1 type Genetics interface {
2     Mutate()
3     Fitness(originalImage image.Image)
4     Cross(spec image.RGBA)
5 }
```

Kod źródłowy 5.1: Interfejs modułu genetyki

W celu własnego zdefiniowania należy zaimplementować metody spełniające interfejs zaprezentowany na 5.1 mając na uwadze, że metoda `Mutate()` definiuje sposób dokonywania mutacji; metoda `Fitness(originalImage image.Image)` definiuje funkcję oceny i jako parametr przyjmuje referencję do obrazu oryginalnego; metoda `Cross(spec image.RGBA)` definiuje sposób krzyżowania osobników i jako parametr przyjmuje referencję do osobnika, z którym dany osobnik ma być krzyżowany. Należy mieć na uwadze, że są to struktury z *dowiązanymi* metodami (ang. *function-like objects*) i mogą się odwoływać do pól struktury, na której operują. Przykład został zaprezentowany takiej struktury został zaprezentowany na 5.2.

```
1 type dummyStruct struct {
2     dummyProperty string
3 }
4
5 func (ds *dummyStruct) dummyMethod() {
6
7 }
```

Kod źródłowy 5.2: Przykład *function-like object*

W celu użycia własnego przygotowanego modułu należy podmienić plik `module.go` znajdujący się w katalogu `/src/genetics`, w którym zaimplementowana jest domyślna genetyka.

Analiza wyników

W ramach pracy dokonano szeregu testów przygotowanej implementacji algorytmu replikacji obrazów. Autor skupił się na tym, aby uzyskać jak najlepsze wyniki, stąd zdecydował się jedynie do metody tworzenia kolejnego pokolenia jedynie z najlepszych osobników z bieżącej iteracji, z jednoczesną możliwością zbadania pewnych parametrów, które pozwolą na stworzenie pewnego schematu działania algorytmów oraz analizę jakości ich działania. Jako osobniki *najlepsze* należy rozumieć te, które są najbardziej *podobne* do obrazu oryginalnego pod względem wartości funkcji celu (funkcja celu jest dla nich jak najmniejsza). Drugorzędnym kryterium oceny była w tym przypadku ocena wizualna. Obranie takiego kryterium wyboru osobników do utworzenia każdego kolejnego pokolenia wynika z faktu, że algorytm do oceny poszczególnych rozwiązań używa regularnej funkcji celu, która posiada lokalne optima z niewielką amplitudą.

W tym rozdziale zostaną przedstawione i omówione przeprowadzone testy. Autor skupił się tutaj na przedstawieniu postawionych hipotez odnośnie wpływu poszczególnych parametrów na jakość uzyskiwanych rozwiązań, analizie i prezentacji uzyskanych wyników oraz wyciągniętych na ich podstawie wniosków odnośnie zależności jakości uzyskiwanych obrazów a danym parametrem.

Większość prezentowanych w tym rozdziale przykładów będzie opierać się na obrazie *Mona Lisa* autorstwa Leonarda da Vinci, a dokładniej - na jego zdjęciu. Autor zdecydował się na wykorzystanie tego dzieła malarskiego ze względu na fakt, że jest to jeden z popularniejszych obrazów, które są wykorzystywane do prezentowania rezultatów w podobnych implementacjach dostępnych w sieci. Ma to na celu umożliwienie porównania czytelnikowi możliwości zaimplementowanego algorytmu z innymi dostępnymi rozwiązaniami wykorzystującymi podobne lub te same techniki.

6.1 Rozmiar populacji

Dla problemu wpływu rozmiaru pojedynczej populacji na jakość otrzymywanych wyników autor pracy postawił następującą hipotezę:

Hipoteza 6.1 *Liczba osobników w każdym pokoleniu wpływa na jakość ostatecznego rozwiązania zwracanego przez algorytm. Wraz ze wzrostem liczby osobników rośnie różnorodność osobników z unikalnym materiałem genetycznym, a przez co wzrasta również prawdopodobieństwo uzyskania osobników lepiej ocenianych przez funkcję celu.*

W celu sprawdzenia powyższej hipotezy wykonano testy pozwalające zbadać wpływ parametru odpowiadającego za rozmiar pojedynczej populacji. Przeprowadzono testy na zdjęciu obrazu *Mona Lisa* autorstwa Leonarda da Vinci zapisany jako plik `PNG` w skali RGBA - rysunek 6.1. Zbadano jak wpływa liczebność pojedynczego pokolenia dla określonej liczby iteracji na jakość uzyskanego rozwiązania.

6.1.1 Obserwacje

Na rysunku 6.2 przedstawiono jakość najlepszych rozwiązań (najlepiej ocenionych przez funkcję celu) po 1000 pokoleń odpowiednio dla 10, 50, 100 oraz 300 osobników w jednym pokoleniu. Zgodnie z tym, jak to opisano na początku niniejszego rozdziału, wybrano metodykę, w której każde kolejne pokolenie tworzone jest jedynie z *pewnej* (określonej przez parametr podany przez użytkownika) liczby osobników, które zostały ocenione najwyżej przez funkcję celu. W tym, konkretnym przypadku dla takich osobników funkcja celu będzie miała wartość najmniejszą, co świadczy o fakcie, że odległość pomiędzy tymi rozwiązaniami a obrazem oryginalnym jest najmniejsza. Widoczna jest jedna znacząca różnica pomiędzy poszczególnymi prezentowanymi

Rysunek 6.1: Replikowany obraz *Mona Lisa* w skali `RGBA` *(Źródło: [16])*

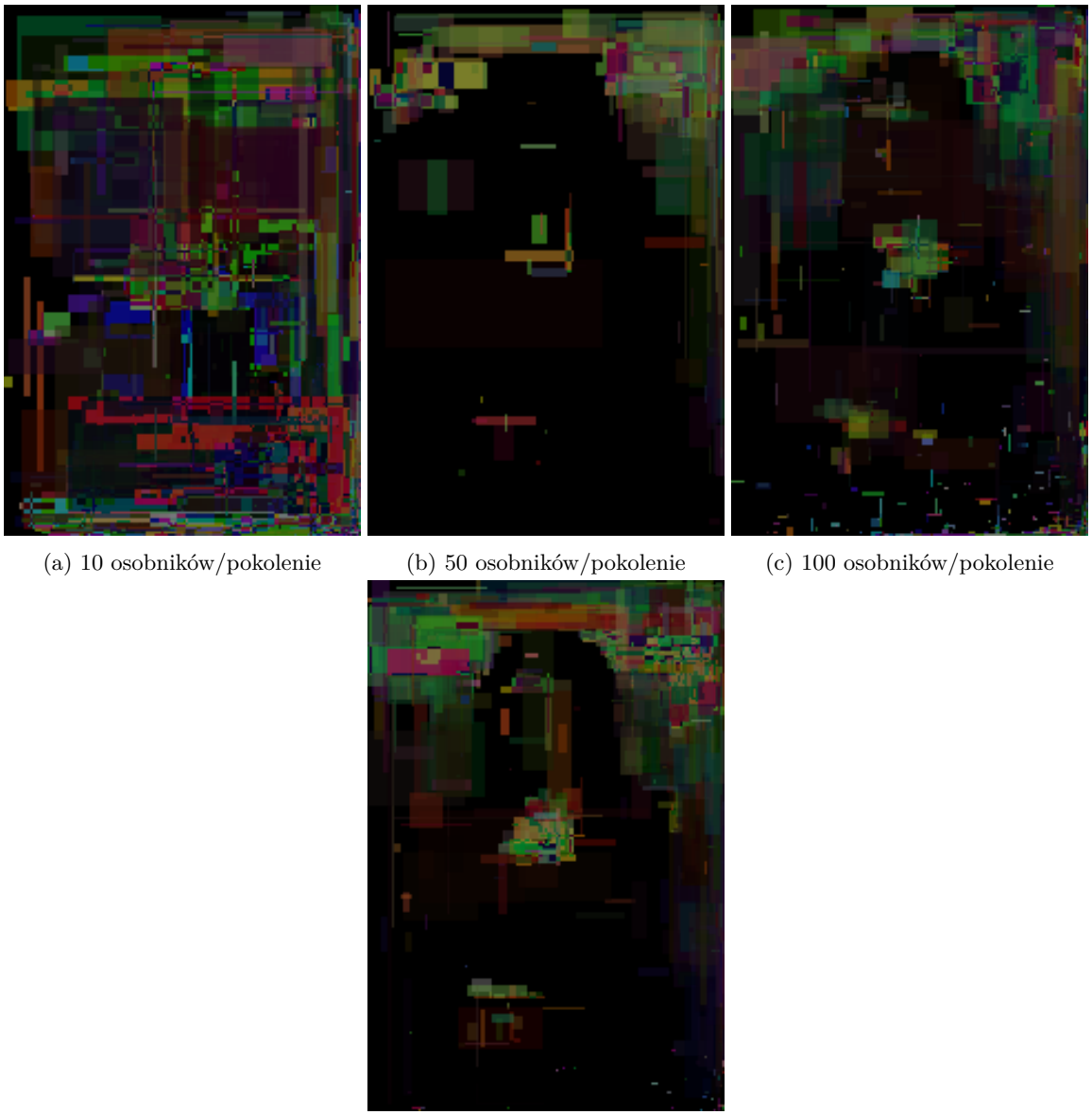
osobnikami pod względem wizualny. Na osobniku, który pochodził z testu, gdzie na pokolenie przypadało 300 osobników, można zauważyć wyraźny zarys postaci oraz elementów tła. Podobnie jest dla osobnika pochodzącego z testu, gdzie na pokolenie przypadało 100 osobników. W przypadku, gdy pokolenia miały liczność odpowiednio 50 i 10 osobników zauważenie konturów tła i postaci jest już praktycznie niemożliwe. Obraz z testu, w którym kolejne populacje liczyły po 10 obrazów, po 1000 iteracji obraz, pod względem wizualnym, w żaden sposób nie przypomina obrazu, do którego replikacji algorytm dąży.

6.1.2 Wnioski

Zauważalna jest wyraźna wizualna różnica pomiędzy poszczególnymi zaprezentowanymi wynikami, dla różnych wartości parametru odpowiadającego za liczebność pokolenia, a obrazem zreplikowanym 6.1. Na podstawie prezentowanych wyników można stwierdzić, że liczebność pojedynczego pokolenia ma zdecydowany wpływ na jakość otrzymywanych rozwiązań - zarówno pod względem wizualnym, jak i względem wartości zwracanych przez funkcję celu. Potwierdza to postawione w hipotezie przypuszczenie, że wzrost liczby osobników powoduje, że różnorodność *materiału genetycznego* jest większa (większa liczba osobników implikuje większą unikalność gwarantowaną przez operator mutacji). W takim przypadku algorytm wybiera osobniki do następnej iteracji z większej puli rozwiązań. Istnieje zatem większe prawdopodobieństwo zajścia mutacji (wzmocnionej potem przez operator krzyżowania) takiej, że osobnik z mutacją znacząco przybliży się, względem wartości funkcji oceny, do obrazu oryginalnego.

6.2 Liczba pokoleń

Hipoteza 6.2 *Liczba pokoleń (iteracji) algorytmu wpływa na jakoś ostatecznego rozwiązania zwracanego przez algorytm. Wraz ze wzrostem liczby iteracji algorytmu zachodzi więcej zmian w materiale genetycznym osobników, przez co rośnie prawdopodobieństwo otrzymania rozwiązania bardziej zbliżonego, względem wartości funkcji celu, do obrazu replikowanego.*



(a) 10 osobników/pokolenie (b) 50 osobników/pokolenie (c) 100 osobników/pokolenie

(d) 300 osobników/pokolenie

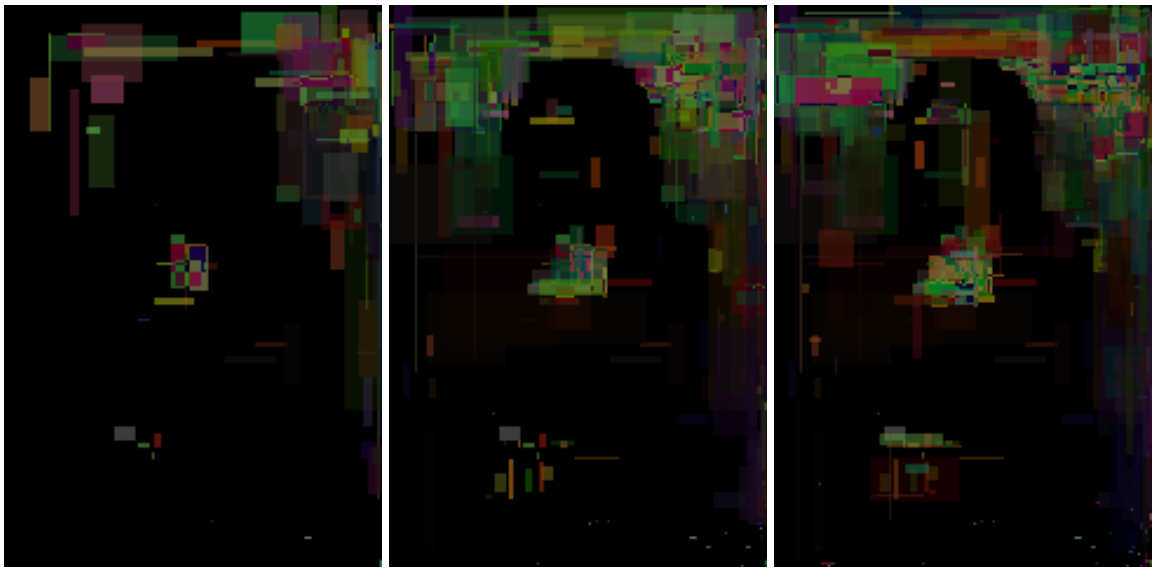
Rysunek 6.2: Wyniki replikacji dla różnych liczności pokolenia dla 1000 iteracji algorytmu

6.2.1 Obserwacje

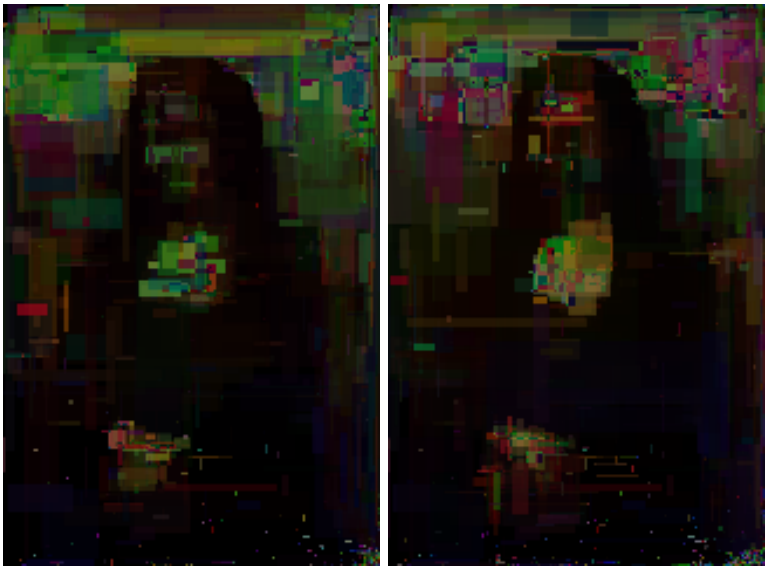
W celu zbadania tego parametru ponownie zreplikowano obraz 6.1. Wyniki zaprezentowano na grafice 6.3. Widoczna jest znacząca różnica pomiędzy rozwiązaniem uzyskanym po 100 iteracjach algorytmu, a rozwiązaniem uzyskanym w ostatniej, 10000., iteracji. Jednocześnie można zauważyć, że pomiędzy obrazem uzyskanym w 5000. iteracji, a osobnikiem z 10000. iteracji nie da się zauważyć, pod względem wizualnym, już wielu różnic.

6.2.2 Wnioski

Na podstawie zaprezentowanych wyników można stwierdzić, że liczba iteracji algorytmu ma znaczący wpływ na jakość zwracanych rozwiązań. Wraz z każdą kolejną iteracją algorytmu widać, że z każdą kolejną iteracją zauważalna jest większa liczba podobieństw między konkretnym rozwiązaniem a obrazem repliko-



(a) 100 iteracji (b) 500 iteracji (c) 1000 iteracji



(d) 5000 iteracji (e) 10000 iteracji

Rysunek 6.3: Wyniki replikacji odpowiednio dla 100, 500, 1000, 5000 oraz 10000 iteracji algorytmu z liczebnością każdego pokolenia - 300 osobników

wanym. Wynika to z faktu, że z każdą kolejną iteracją, genotyp poszczególnych osobników jest wzbogacany przez stosowane operatory genetyczne. Podobnie, jak ma to miejsce w przypadku opisanego w podrozdziale 6.1, w tym przypadku również rośnie prawdopodobieństwo uzyskania obrazu bardziej zbliżonego do oryginału. Czynnikiem decydującym w tym przypadku jest nie ilość osobników w pojedynczym pokoleniu, ale liczba iteracji. Podobnie, jak ma to miejsce w naturalnym procesie ewolucji, z każdym kolejnym pokoleniem w osobnikach zachodzą zmiany genetyczne. Dysponując dużą pulą osobników można uzyskać wiele różnych zmian w genotypie, a powtarzając tę operację określoną liczbę razy, uzyskać wynik bardziej zbliżony do oryginalnego. Należy stwierdzić zatem, że zarówno parametr odpowiadający za liczbę pokoleń w algorytmie, jak i parametr odpowiedzialny za określenie mnogości każdego pokolenia, są ze sobą w ścisły sposób powiązane.

Kolejnym wnioskiem, wynikającym z obserwacji dotyczącej niewielkich zmian pomiędzy 5000. a 10000. pokoleniem, jest to, że prezentowany algorytm replikacji może utknąć w lokalnym optimum. Oznacza to, że

uzyskiwane odchylenia funkcji celu będą się mieścić w pewnym wąskim przedziale i nie będą się już znacząco powiększać. Od strony wizualnej, każde kolejne zmiany będą już praktycznie niezauważalne.

6.3 Skala barw

Hipoteza 6.3 *Zastosowana w obrazie oryginalnym skala barw (rozważane są skala RGBA oraz skala szarości) ma pływ na jakość uzyskanych rozwiązań. Dla skali szarości (obrazy jednokanałowe) algorytm jest w stanie odtworzyć więcej szczegółów obrazu replikowanego, niż w przypadku zastosowania skali RGBA (obrazy czterokanałowe).*

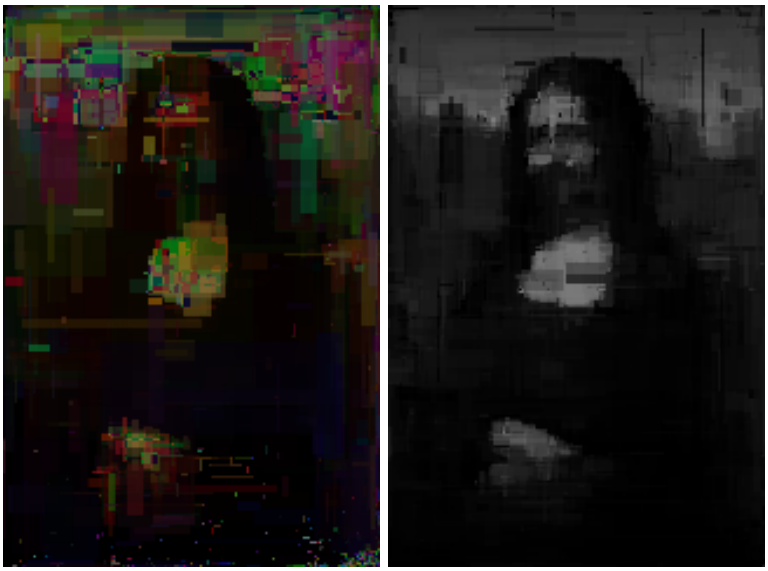
Rysunek 6.4: Replikowany obraz *Mona Lisa* w skali szarości (*Źródło*: [16])

6.3.1 Obserwacje

W celu zaobserwowania różnic dla różnych skali barw dokonano replikacji obrazów 6.1 oraz 6.4 dla takich samych parametrów wejściowych. Wyniki (dla 10000. pokolenia) zaprezentowano na grafice 6.5. Pod względem wizualnym, zauważalnym jest, że dla obrazu, który replikowany był w odcieniach szarości, widoczna jest większa liczba szczegółów, niż dla obrazu prezentowanego w skali RGBA. Porównywanie wartości funkcji celu, ze względu na dwie odmienne skale barw, zostanie tutaj pominięte.

6.3.2 Wnioski

Wyniki replikacji w skali szarości pozwoliły na uzyskanie większej ilości szczegółów względem pierwowzoru, ze względu na fakt, że rozważany jest wówczas tylko jeden kanał i, w konsekwencji, podczas mutacji losowana jest tylko pojedyncza wartość. W przypadku skali RGBA takie wartości muszą być wylosowane cztery co jednocześnie zwiększa liczbę możliwych konfiguracji wartości dla wszystkich kanałów, przez co prawdopodobieństwa *trafienia* w wartość koloru, który byłby jak najmniej oddalony od koloru danego piksela na obrazie oryginalnym maleje.



(a) Skala RGBA (b) Skala szarości

Rysunek 6.5: Porównanie wyników replikacji dla skali szarości i RGBA *(10000 pokoleń, 300 osobników w pokoleniu*

6.4 Prawdopodobieństwo mutacji

Kolejnym badanym parametrem, mającym wpływ na jakość uzyskanych rozwiązań był parametr odpowiadający za prawdopodobieństwo wystąpienia na danym osobniku mutacji. Ponownie zbadano wpływ parametru na jakość uzyskiwanych rozwiązań dla obrazu 6.1.

Hipoteza 6.4 *Większa wartość parametru p_m określającego prawdopodobieństwo wystąpienia mutacji wpływa na szybkość zbiegania algorytmu.*

6.4.1 Obserwacje

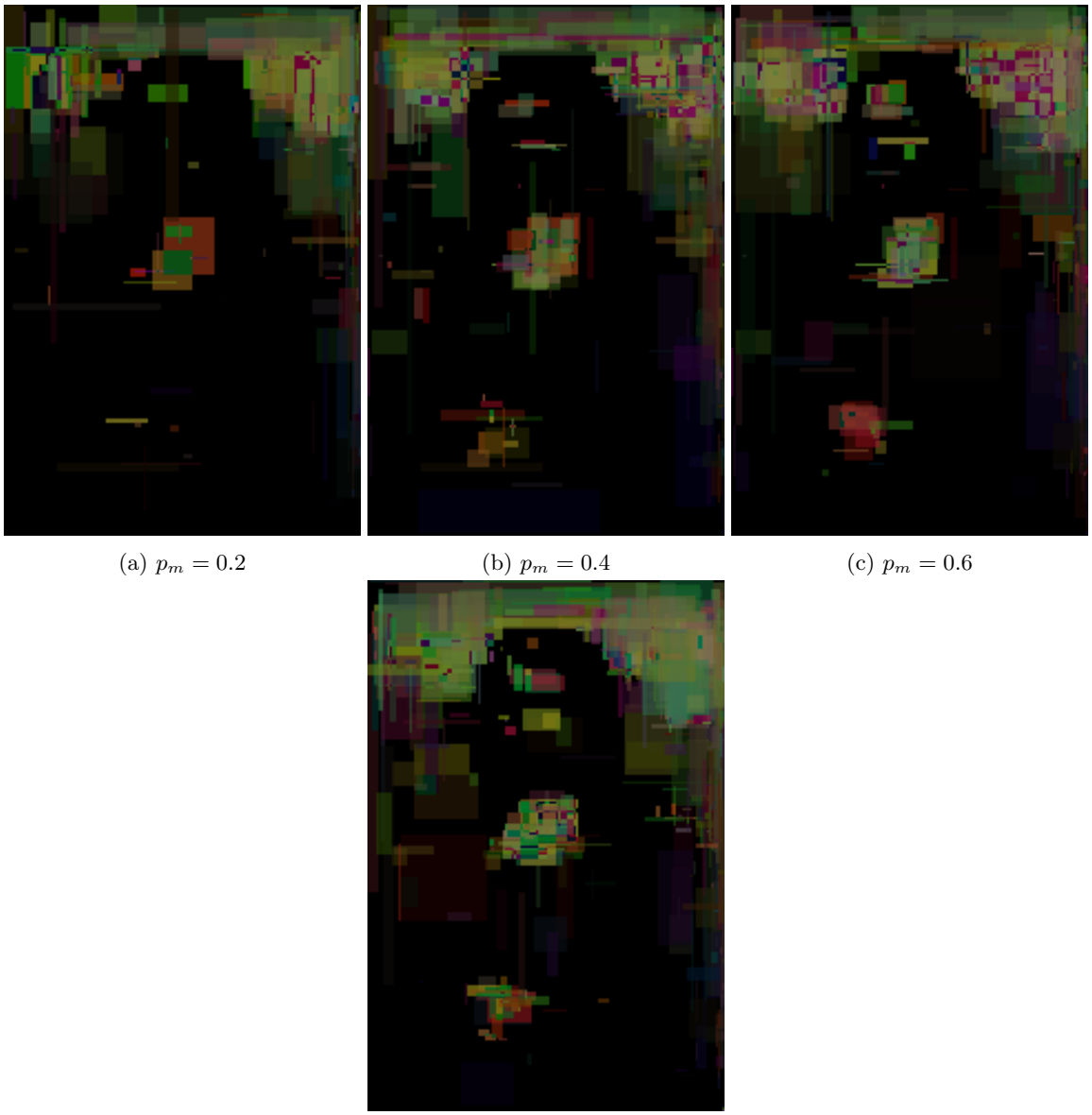
Zbadano jakość uzyskiwanych rozwiązań dla parametru prawdopodobieństwa wystąpienia mutacji $p_m \in \{0.2, 0.4, 0.6, 0.8\}$. Wyniki przeprowadzonych testów zaprezentowano na grafice 6.6. W tabeli 6.1 przedstawiono wartości funkcji celu dla poszczególnych parametrów. Widoczne jest malenie wartości funkcji celu f_c wraz ze wzrostem wartości parametru p_m. Pod względem wizualnym również daje się zauważyć, że na obrazach, gdzie prawdopodobieństwo wystąpienia mutacji było większe, odwzorowanie jest bardziej szczegółowe i na obrazach tych wyraźniej widoczna jest sylwetka postaci z obrazu będącego ich pierwowzorem.

p_m	f_c
0.2	496634177696466.0
0.4	496634145298532.0
0.6	496634099405528.0
0.8	496633985597704.0

Tablica 6.1: Wartości funkcji celu dla różnych wartości parametru p_m określającego prawdopodobieństwo wystąpienia mutacji. Wyniki dla obrazów prezentuje rysunek 6.6.

6.4.2 Wnioski

Na podstawie przedstawionych obserwacji łatwo zauważyć, że większa wartość parametru określającego prawdopodobieństwo wystąpienia mutacji w danym osobniku pozytywnie wpływa na jakość uzyskiwanych



(a) $p_m = 0.2$ (b) $p_m = 0.4$ (c) $p_m = 0.6$

(d) $p_m = 0.8$

Rysunek 6.6: Porównanie wyników replikacji dla różnych wartości parametru p_m określającego prawdopodobieństwo mutacji *1000 iteracji, 300 osobników w pokoleniu*

rozwiązań. Im większe prawdopodobieństwo mutacji, tym większa szansa, że do genotypu danego osobnika zostanie wprowadzona *pozytywna* zmiana. Przez *pozytywną* zmianę rozumie się taką, która sprawie, że odległość pomiędzy wygenerowanym rozwiązaniem, a obrazem oryginalnym zmniejsza się w stosunku do stanu, zanim mutacja została wprowadzona. Należy też w tym miejscu zauważyć, że większa wartość parametru p_m niekoniecznie musi gwarantować uzyskanie lepszych wyników. Może bowiem dojść do takiej sytuacji, że genotyp osobników będzie w taki sposób modyfikowany, że zmiany nie będą korzystne dla oczekiwanych rezultatów, to jest mogą zostać wprowadzone zmiany, które sprawią, że osobniki w kolejnym pokoleniu mogą zostać ocenione gorzej od osobników z pokolenia poprzedniego.

6.5 Liczba osobników wybieranych do utworzenia populacji

W następnej kolejności zbadano wpływ liczby osobników wybieranych jako najlepsze w danym pokoleniu do utworzenia kolejnej populacji na jakość uzyskiwanych rozwiązań.

Hipoteza 6.5 *Wzrost liczby najlepiej przez funkcję celu ocenianych osobników wybieranych do następnego pokolenia zmniejsza prawdopodobieństwo zbiegania algorytmu do optymalnego rozwiązania.*

6.5.1 Obserwacje

W tym przypadku autor ponownie zbadał to zjawisko na podstawie replikacji obrazu 6.1. Wyniki zaprezentowano na grafice 6.7. Bardzo wyraźnie widać, że wraz ze wzrostem liczby osobników wybieranych do tworzenia kolejnego pokolenia zarys sylwetki widocznej na obrazie kobiety zaczyna być co raz mniej dostrzegalny. Dla 200 wybieranych osobników, po 1000 iteracji algorytmu na obrazie widać jedynie losowy szum, który zupełnie nie przypomina pierwowzoru.

6.5.2 Wnioski

Wraz ze wzrostem liczby osobników, z których jest tworzone kolejne pokolenie maleje czytelność uzyskiwanych wyników. Dla wartości parametru 2, 50 i 100 daje się jeszcze na obrazie rozróżnić sylwetkę kobiety z oryginalnego obrazu. W przypadku wartości 100 i 200 na obrazach widoczny jest jedynie losowy szum. Wynika to z faktu, że wraz ze wzrostem liczby wyselekcjonowanych osobników, do następnego pokolenia dostają się również osobniki gorzej ocenione przez funkcję celu. W konsekwencji tego wybór staje się bardziej losowy. Na podstawie obserwacji można stwierdzić, że im mniej osobników zostało wybranych do utworzenia nowej populacji, tym algorytm szybciej zbiega do oczekiwanego rozwiązania.

6.6 Złożoność obrazu

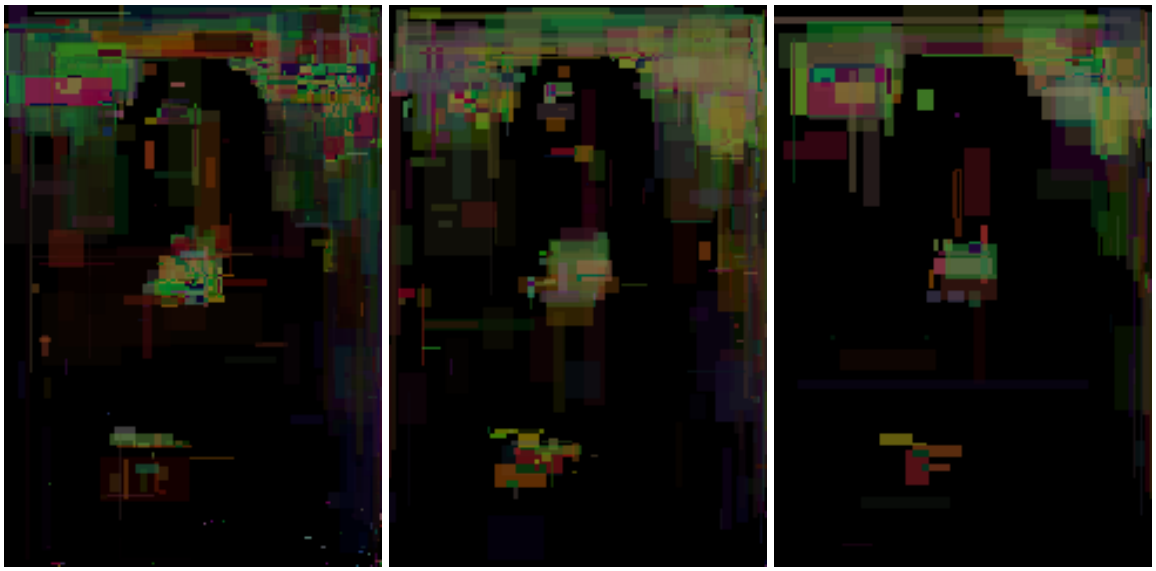
Ostatnim zbadanym czynnikiem, mogącym mieć wpływ na jakość uzyskiwanych wyników, a przez to - na szybkość zbiegania algorytmu, była złożoność replikowanego obrazu.

Hipoteza 6.6 *Wraz ze wzrostem liczby szczegółów na replikowanym obrazie maleje dokładność jego odwzorowania.*

Postawioną powyżej hipotezę sprawdzono na kilku obrazach o różnym stopniu skomplikowania zaprezentowanych na grafice przedstawionych na rysunku 6.8. Obraz **(a)** przedstawia zdjęcie twarzy wygenerowanej przez sztuczną inteligencję. Mimo sporej liczby szczegółów, twarz osoby obecnej na zdjęciu zajmuje sporą część zdjęcia przez to jest elementem zajmującym większość przestrzeni na obrazie. Ponadto obraz ten posiada jednokolorowe tło. Na **(b)** przedstawiona portret postaci w taki sposób, że jest ona widoczna do końca jej tułowia. Obraz zawiera sporo szczegółów w tle jak i na ubraniu, które ma na sobie sportretowana kobieta (zagięcia, cienie). Ostatni obraz **(c)**, zawiera wiele elementów o różnym stopniu jasności i skomplikowania.

6.6.1 Obserwacje

Wyniki replikacji obrazów przedstawia grafika 6.9. Widać, że algorytm poradził sobie najlepiej z obrazem **(a)**, na którym widoczna była sama twarz, a najgorzej z tymi, które wymagały odwzorowania wielu elementów o różnych rozmiarach. W przypadku dwóch pozostałych obrazów widać jedynie zarysy postaci. O ile w przypadku zreplikowanego obrazu **(b)** obserwator jest w stanie domyślać się, że jest to portret kobiety, o tyle w przypadku **(c)** ciężko stwierdzić co dokładnie zostało na obrazie przedstawione.



(a) 2 osobniki (b) 10 osobników (c) 50 osobników

(d) 100 osobników (e) 200 osobników

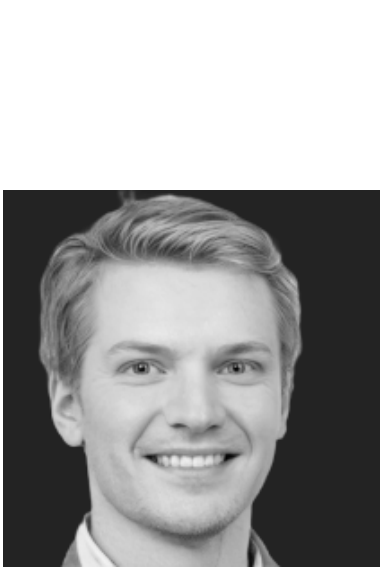
Rysunek 6.7: Porównanie wyników replikacji dla różnej liczby osobników wykorzystanych do tworzenia nowego pokolenia *(1000 pokoleń, 300 osobników w pokoleniu)*

6.6.2 Wnioski

Na podstawie przedstawionych obserwacji można wyciągnąć wnioski, że liczba szczegółów ma znaczący wpływ na szczegółowość odwzorowania obrazu. Przy okazji badania wpływu szczegółowości obrazu na jakość rozwiązania, warto zauważyć, że jest to również poniekąd związane z rozdzielczością obrazu. Im większy obraz, tym więcej szczegółów jest na nim widocznych.

6.7 Czas działania programu

Autor zbadał również zależność czasu działania algorytmu od dwóch najbardziej istotnych parametrów jakimi są liczba iteracji oraz rozmiar pojedynczego pokolenia. Wyniki zaprezentowano w tabelach 6.2 oraz



(a) Zdjęcie twarzy wygenerowane przez AI (*Źródło*: [13])

(b) *Mona Lisa*, Leonardo da Vinci (*Źródło*: [16])

(c) *Guernica*, Pablo Picasso (*Źródło*: [23])

Rysunek 6.8: Obrazy o różnej złożoności pod względem liczby obecnych na nich szczegółów

6.3 (Testy przeprowadzono na systemie `macOS 10.15 Catalina` i sześciordzeniowym procesorze (16 wątków) Intel Core i7. Czas działania programu został zmierzony za pomocą konsolowego programu `time` dostępnego w systemach typu Unix-like).

i	N	time $[s]$
10000	50	294.52
10000	100	540.48
10000	200	1128.97
10000	300	1731.64

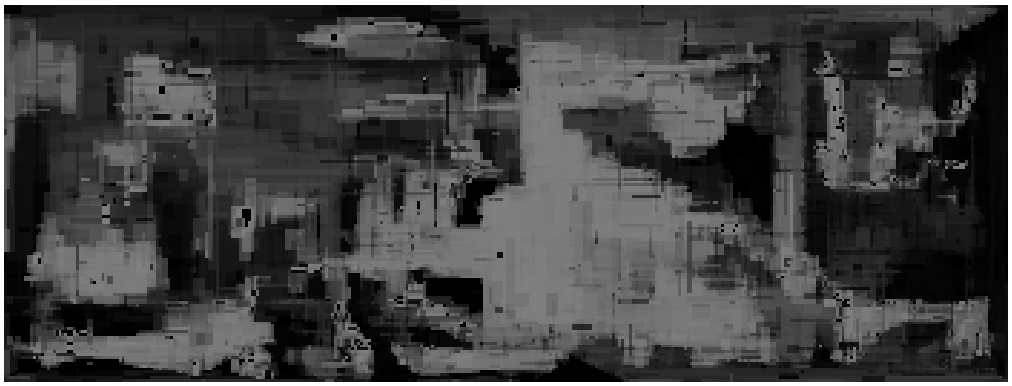
Tablica 6.2: Czasy działania algorytmu (w sekundach) dla stałej liczby iteracji iteracji (i) oraz zmiennego rozmiaru pokolenia (N).

6.7.1 Obserwacje

Na podstawie przeprowadzonych obserwacji można zauważyć, że zarówno, gdy zmienia się rozmiar pokolenia przy stałej liczbie iteracji (tabela 6.2), jak i w sytuacji, gdy zmianie ulega liczba iteracji, liczność

(a) Zdjęcie twarzy wygenerowane przez AI

(b) *Mona Lisa*, Leonardo da Vinci



(c) *Guernica*, Pablo Picasso

Rysunek 6.9: Wyniki replikacji obrazów przedstawionych na 6.8 *(10000 pokoleń, 300 osobników w pokoleniu)*

i	N	time $[s]$
500	200	52.86
1000	200	111.07
5000	200	588.98
10000	200	1731.64

Tablica 6.3: Czasy działania algorytmu (w sekundach) dla zmiennej liczby iteracji iteracji (i) oraz stałego rozmiaru pokolenia (N).

pokolenia jest stała (tabela 6.3), czas działania programu rośnie wprost proporcjonalnie do zmieniającego się parametru.

6.7.2 Wnioski

W obu przypadkach wzrost czasu działania programy wynika z innej przyczyny. W sytuacji, gdy zmienia się jedynie rozmiar pokolenia, program wykonuje się dłużej, ponieważ w każdej iteracji potrzebuje więcej czasu na przetworzenie wszystkich osobników (zastosowanie operatorów mutacji i ocena). W przypadku, gdy stały jest rozmiar pokolenia, a zmienia się liczba iteracji fakt, że program działa dłużej wynika z tego, że

chociaż w każdej iteracji przetworzenie wszystkich osobników w populacji zajmuje tyle samo czasu, ale wzrost jest spowodowany większą liczbą powtórzeń w programie.

6.8 Parametry a ograniczenie czasowe

Ostatnim elementem analizy jest zbadanie zależności pomiędzy liczbą iteracji i licznością pojedynczego pokolenia, w przypadku, gdy na algorytm zostało narzucone ograniczenie czasowe. Celem tego testu jest sprawdzenie, który parametr opłaca się zwiększyć (kosztem zmniejszenia wartości drugiego), aby uzyskać jak najlepsze wyniki. Autor przeprowadził testy na obrazie 6.4 dla różnych wartości liczby pokoleń algorytmu i rozmiaru populacji przy założeniu, że jest to zależność odwrotnie proporcjonalna (czas działania dla każdego był, w przybliżeniu, taki sam). Wyniki prezentuje grafika 6.10. Zbadane konfiguracje parametrów i czas działania programu dla każdej z nich przedstawiono w tabeli 6.4.

i	N	time $[s]$
10000	50	294.52
5000	100	282.98
1000	500	275.97
500	1000	284.22

Tablica 6.4: Czasy działania algorytmu (w sekundach) dla różnych wartości liczby iteracji (i) oraz rozmiaru pojedynczego pokolenia (N). Testy przeprowadzono na systemie `macOS 10.15 Catalina` i sześciordzeniowym procesorze (16 wątków) Intel Core i7. Czas działania programu został zmierzony za pomocą konsolowego programu `time` dostępnego w systemach typu Unix-like.

6.8.1 Obserwacje

Można zauważyć, że czas działania dla każdej z konfiguracji parametrów jest w przybliżeniu taki sam (ponad 4 minuty). Na grafice 6.10 widać wyraźnie, że wraz ze wzrostem liczby pokoleń i jednoczesnym zmniejszeniem liczby osobników w pokoleniu znacznie pogorszyła się, pod względem wizualnym, jakość uzyskanego rozwiązania. Uzyskany rezultat jest bardzo odległy od oczekiwanego - sylwetka postaci przedstawionej na obrazie oryginalnym jest prawie całkowicie niewidoczna. Najlepsze wyniki uzyskano dla przypadku, gdy ograniczono liczbę iteracji algorytmu (do 1000-500) a zwiększono liczbę osobników tworzących dane pokolenie - na tych obrazach wyraźnie można wyróżnić prezentowaną na obrazie postać oraz zarysy elementów tła.

6.8.2 Wnioski

Na podstawie przeprowadzonych obserwacji można stwierdzić, że w przypadku, gdy wymagane jest narzucenie ograniczenia czasowego na program, lepszym wyborem będzie zwiększenie liczby osobników w pokoleniu kosztem zmniejszenia liczby iteracji programu.

6.9 Podsumowanie

W niniejszym rozdziale dotyczącym analizy uzyskanych wyników przedstawiono wyniki testów, które miały zbadać wpływ możliwych do ustawiania przez użytkownika parametrów, które, potencjalnie, mogą mieć największy wpływ na jakość uzyskiwanych rozwiązań. Po analizie przedstawionych obserwacji oraz wniosków można zauważyć, że algorytm lepiej radzi sobie z obrazami, zawierającymi mniej szczegółów oraz w jednolitej skali kolorystycznej - skali szarości. Obserwator może również stwierdzić, że algorytmowi trudniej odwzorować obszary ciemniejsze, a dokładniejsze odwzorowanie pierwowzoru otrzymuje się w przypadku obrazów jaśniejszych. W kontekście możliwych rozszerzeń pracy należałoby przebadać jakość uzyskiwanych rozwiązań dla, na przykład zdjęć zrobionych w dobrych warunkach oświetleniowych, obrazów namalowanych w jasnych kolorach lub zdjęć prześwietlonych.

(a) 10000 pokoleń, 50 osobników w pokoleniu

(b) 5000 pokoleń, 100 osobników w pokoleniu

(c) 1000 pokoleń, 500 osobników w pokoleniu

(d) 500 pokoleń, 1000 osobników w pokoleniu

Rysunek 6.10: Wyniki prezentujące zależność pomiędzy liczbą pokoleń algorytmu a licznością pokolenia

Autor chce zauważyć, że jest to jedynie niewielka część różnych zestawień konfiguracji algorytmu, które mogłyby zostać zbadane. Zdecydował się on na zaprezentowanie jedynie tych, które według jego uznania pokazywały w najlepszy sposób zarówno mocne, jak i słabe strony opisywanego algorytmu. Ze względu małej ilości czasu autorowi nie udało się przeprowadzić wszystkich zamierzonych testów. Sugerowane ścieżki testowania i oceny generowanych rozwiązań to:

- zbadanie subiektywnej oceny osób trzecich w postaci ankiety i późniejsza jej analiza,
- zbadanie innych metryk oceny jakości rozwiązań (funkcja celu),
- zbadanie jak algorytm poradzi sobie z obrazami w dużych rozdzielczościach, na przykład *FullHD*,

- zbadanie dokładnego profilowania pamięci podczas działania programu, na przykład za pomocą narzędzia `Valgrind`.

Podsumowanie

W niniejszym rozdziale przedstawiono wnioski wynikające z przeprowadzonych testów oraz dokonanej analizy problemu. Przedstawiono również możliwości późniejszego rozwoju oraz możliwych do wprowadzenia w aplikacji usprawnień.

7.1 Podsumowanie

W pracy dokonano analizy problemu jakim jest replikacja obrazów. Przedstawiono dostępne techniki oraz przeanalizowano rozwiązanie tego problemu za pomocą algorytmów genetycznych. Następnie został opisany zaprojektowany algorytm służący do replikacji obrazów. Dokonano analizy wymagań dla implementacji tego algorytmu oraz przedstawiono szczegóły techniczne w formie omówienia struktury dołączonej do pracy aplikacji i załączono instrukcję jej użytkowania. Na końcu przedstawiono wyniki testów mających zbadać jakość rozwiązań zwracanych przez zaprojektowany algorytm oraz wpływ różnych konfiguracji algorytmu na te rozwiązania.

7.2 Wnioski

Na podstawie przeprowadzonych testów oraz analizy badanego problemu wyciągnięte zostały przez autora pracy następujące wnioski:

- Algorytmy genetyczne pozawalają na znalezienie w miarę dokładnego przybliżenia rozwiązania dla postawionego problemu - w przypadku pracy, problemu optymalizacyjnego. Przez *w miarę dokładne* należy rozumieć obrazy podobne wizualnie do obrazu replikowanego i wysoko ocenione przez funkcję celu.
- Algorytmy genetyczne dla pewnych zestawów danych zatrzymują się w lokalnym optimum i nie są w stanie wygenerować dokładniejszego rozwiązania, niż to już przez nie osiągnięte.
- Duży wpływ na jakość uzyskiwanych rozwiązań ma, podobnie jak w procesie naturalnej ewolucji, zmienność w genotypach generowanych przez algorytm rozwiązań co daje szansę na uzyskanie rozwiązania bardziej zbliżonego do optymalnego.

7.3 Możliwe rozszerzenia pracy

W czasie pisania pracy, zarówno części teoretycznej, jak i praktycznej, udało się zebrać pewną ilość uwag odnośnie możliwych udoskonaleń pracy oraz jej przyszłego rozwoju. Do najważniejszych zaliczyć należy:

- rozszerzenie algorytmu o dodatkowe operatory genetyczne oraz sposoby selekcji osobników do utworzenia kolejnych pokoleń,
- zbadanie wpływu na jakość rozwiązań innych przestrzeni barw, np. HSV, HSL,
- optymalizacje algorytmu w celu przyśpieszenia jego działania,

- zbadanie innego sposobu oceniania jakości rozwiązań przez algorytm (funkcja celu),
- porównanie opisywanego rozwiązania z innymi dostępnymi technikami replikacji (*machine learning)*,
- stworzenie warstwy GUI dla lepszej prezentacji wyników oraz wygodniejszej konfiguracji algorytmu.

Bibliografia

[1] Dokumentacja języka `Go` - biblioteka `sync`. URL: https://golang.org/pkg/sync/. [Dostęp: 12.11.2019r.].

[2] Dokumentacja języka `Go` - channels. URL: https://gobyexample.com/channels. [Dostęp: 12.11.2019r.].

[3] Dokumentacja języka `Go` - gorutines. URL: https://gobyexample.com/goroutines. [Dostęp: 12.11.2019r.].

[4] Dokumentacja języka `Go` - wait groups. URL: https://gobyexample.com/waitgroups. [Dostęp: 12.11.2019r.].

[5] Dokumentacja oprogramowanie `Docker`. URL: https://docs.docker.com/compose/. [Dostęp: 11.12.2019r.].

[6] Gorutines. URL: https://golangbot.com/goroutines/. [Dostęp: 12.11.2019].

[7] Opis narzędzia `Docker`. URL: https://docs.docker.com/engine/docker-overview/. [Dostęp: 12.12.2019r.].

[8] Przykład generowania obrazów za pomocą ganów. URL:https://towardsdatascience.com/a-new-way-to-look-at-gans-7c6b6e6e9737. [Dostęp: 24.11.2019r.].

[9] Schemat skali rgb. URL: https://upload.wikimedia.org/wikipedia/commons/a/af/RGB˙color˙solid˙cube.png. [Dostęp: 28.12.2019r.].

[10] Specyfikacja przestrzeni barw `RGBA`. URL: https://www.w3.org/TR/PNG-DataRep.html. [Dostęp: 16.11.2019r.].

[11] Słownik języka polskiego. URL: https://sjp.pwn.pl/szukaj/heurystyka.html. [Dostęp: 12.11.2019].

[12] Wstęp do teorii złożoności obliczeniowej. URL: http://wazniak.mimuw.edu.pl/index.php?title=Z%C5%82o%C5%BCono%C5%9B%C4%87˙obliczeniowa/Wyk%C5%82ad˙1:˙Obliczenia˙w˙modelu˙maszyny˙Turinga. [Dostęp: 22.11.2019].

[13] Zdjęcie twarzy wygenerowane przez ai. URL: https://generated.photos/face/joyfull-white-young-adult-male-with-short-blond-hair-and-blue-eyes--5dd09cb4def8b400084dc56f. [Dostęp: 28.12.2019r.].

[14] T. H. Cormen, C. E. Leiserson, et. al. *Wprowadzenie do algorytmów*. Wydawnictwa Naukowo-Techniczne, 2001.

[15] Z. J. Czech. *Wprowadzenie do obliczeń równoległych*. Wydawnictwo Naukowe PWN, 2013.

[16] L. da Vinci. Obraz *Mona Lisa*. URL: https://www.luelue.pl/media/catalog/product/cache/1/image/880x880/9df78eab33525d08d6e5fb8d27136e95/r/e/rep˙vinci˙002˙10x7-wiz1.jpg. [Dostęp: 28.12.2019r.].

[17] L. Davis, M. Steenstrup. *Genetic Algorithms and Simulated Annealing: An Overview*.

[18] M. Dorigo, G. Di Caro. *The Ant Colony Optimization meta-heuristic*.

[19] F. Glover, M. Laguna. *Tabu Search*.

[20] S. Luke. Essentials of metaheuristics, 2015. A Set of Undergraduate Lecture Notes.

[21] Z. Michalewicz. *Algorytmy genetyczne + struktury danych = programy ewolucyjne.* Wydawnictwo Naukowo-Techniczne, 2003.

[22] C. H. Papadimitriou. *Złożoność obliczeniowa.* Wydawnictwo Naukowo-Techniczne, 2002.

[23] P. Picasso. Obraz *Guernica.* URL: https://www.artdependence.com/media/8815/guernica˙all.jpg. [Dostęp: 28.12.2019r.].

[24] J. Rückert. Artificial art: Image generation using evolutionary algorithms. 2015.

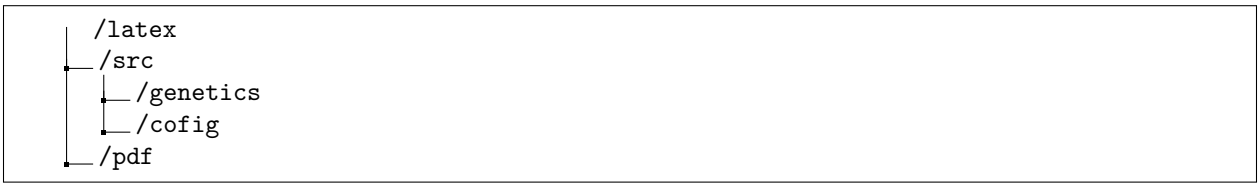
[25] P. J. M. van Laarhoven, E. H. L. Aarts. *Simulated Annealing: Theory and Applications.*

Zawartość płyty CD

Płyta CD dołączona do niniejszej pracy zawiera następujące elementy:

1. Kody źródłowe pisemnej części pracy w języku LaTeXwraz z niezbędnymi grafikami.
2. Skompilowany ze źródeł plik PDF z pracą.
3. Kody źródłowe części implantacyjnej w języku `Go`.

Struktura katalogów na płycie CD została przedstawiona na A.1.



Rysunek A.1: Struktura plików

W katalog `/latex` znajdują się kody źródłowe pisemnej części pracy, w katalogu `/pdf` znajduje się skompilowany plik PDF, a w `/src` pliki źródłowe części implementacyjnej.