Uniwersytet Śląski

Wydział Nauk Ścisłych i Technicznych
Kierunek Informatyka

Paweł Chmielarski

Nr albumu: 315662

Studia niestacjonarne

Komponowanie utworów muzycznych z zastosowaniem systemów mrowiskowych

Praca magisterska

napisana pod kierunkiem

prof. dr hab. Urszuli Boryczki

Sosnowiec 2022

Słowa kluczowe: .....
(maks. 5)

Oświadczenie autora pracy

Ja, niżej podpisany:

imię (imiona) i nazwisko .....

autor pracy dyplomowej pt. .....

.....

.....

Numer albumu: .....

Student .....Uniwersytetu Śląskiego w Katowicach
(Nazwa wydziału/jednostki dydaktycznej)

kierunku studiów

.....

specjalności*

.....

*(wypełnić, jeśli dotyczy)

Oświadczam, że ww. praca dyplomowa:

- została przygotowana przeze mnie samodzielnie[1],
- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity Dz. U. z 2006 r. Nr 90, poz. 631, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- nie była podstawą nadania dyplomu uczelni wyższej lub tytułu zawodowego ani mnie, ani innej osobie.

Oświadczam również, że treść pracy dyplomowej zamieszczonej przeze mnie w Archiwum Prac Dyplomowych jest identyczna z treścią zawartą w wydrukowanej wersji pracy.

Jestem świadomy/-a odpowiedzialności karnej za złożenie fałszywego oświadczenia.

....

Data Podpis autora pracy

[1] uwzględniając merytoryczny wkład promotora (w ramach prowadzonego seminarium dyplomowego)

Spis treści

1 Wstęp

Człowiek jako jedyny gatunek na ziemi opanował sztukę umiejscawiania dźwięków w czasie, którą znamy jako muzykę. Szacuje się, że około 17 tysięcy lat temu nauczyliśmy się grać na pierwszych instrumentach przypominających łuki myśliwskie. Z czasem nasze zdolności komponowania muzyki [2] rozwijały się, powstawały nowe instrumenty i sposoby tworzenia muzyki. Stawały się one coraz bardziej precyzyjne, wyszukane. Poczynając od prostych instrumentów jednostrunowych, złożonych instrumentów akustycznych, przez te elektroniczne aż po cyfrowe, aktualnie dysponujemy praktycznie nieograniczonymi możliwościami kreowania muzyki. Łącząc umiejętności i techniki muzyczne z osiągnięciami naukowymi ostatnich lat, jakimi są algorytmy sztucznej inteligencji [11], możliwe stało się stworzenie programów komponujących muzykę. Takie rozwiązania powstawały już na przestrzeni ostatnich lat, dając przy tym efekty zbliżone do uzyskiwanych przez ludzkiego kompozytora. Tworzone są również inteligentne narzędzia, które same nie komponują, ale wspierają kreatywność [38] artysty w procesie twórczym. Nieczęsto jednak wykorzystywano algorytmy biomimetyczne [36], czyli inspirowane naturą do realizacji takich systemów.

Jednym z takich algorytmów są algorytmy mrowiskowe [1]. Zostały one stworzone na podstawie obserwacji mrówek żyjących w ramach kolonii i znajdują zastosowanie w rozwiązywaniu problemów kombinatorycznych. Jest to stosunkowo nowy rodzaj algorytmów, który ciągle jest rozwijany i implementowany na różne sposoby [19]. Czy możliwe jest zaadaptowanie takiego algorytmu do zadania komponowania muzyki i uzyskanie zadowalających rezultatów? Niniejsza praca postara się odpowiedzieć na to pytanie.

Komponowanie muzyki jest procesem twórczym, który wymaga czasu, zaangażowania oraz odpowiedniej wiedzy do jego przeprowadzenia. W dzisiejszych czasach istnieje duże zapotrzebowanie na materiał muzyczny do zastosowań w grach, filmach, reklamach oraz innych mediach. Przykładowo skomponowanie muzyki do etapu gry trwającego 30 godzin wymagałoby dużych nakładów finansowych i czasowych. Często do takich zastosowań zamawiane są krótsze utwory, a następnie odtwarzane w pętli, co może zwracać uwagę na powtarzalność muzyki. Z tego punktu widzenia komponowanie muzyki za pomocą sztucznej inteligencji wydaje się niezwykle interesujące ze względu na teoretyczną możliwość generowania nieskończonego i niepowtarzalnego utworu

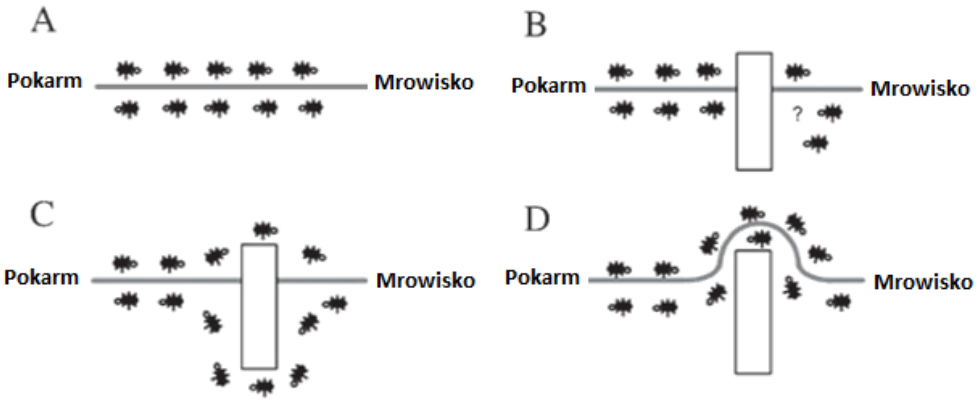
muzycznego przy znikomym koszcie i w krótkim czasie. Istnieje także potencjalne zastosowanie takiego systemu jako narzędzia wspierającego artystę w procesie twórczym, dając mu środowisko pracy, które zwiększa kreatywność. Praca ta będzie jednak traktować o realizacji takiego systemu, a nie jego zastosowaniach. Pomysł ten powstał jako iloczyn trzech zainteresowań autora, jakimi są muzyka, sztuczna inteligencja i przyroda.

Celem pracy jest zaprojektowanie i realizacja systemu komponującego muzykę z zastosowaniem algorytmów mrowiskowych, przeprowadzenie badań, dokonanie pomiaru uzyskiwanych rezultatów oraz wyciągnięcie wniosków. System będzie miał za zadanie generować harmonijnie brzmiące melodie na podstawie wprowadzonych danych wejściowych i ustawionych parametrów systemu. Badania będą polegały na pomiarze jakości uzyskiwanych rezultatów poprzez wykonanie wielu testów z różnymi zestawami parametrów, obliczaniu korelacji między parametrami i analizie wyników, a w szczególności funkcji oceny. Wyniki badań mają zwrócić informacje, jak zaimplementowany system radzi sobie z zadaniem komponowania muzyki oraz jaki wpływ mają poszczególne parametry na uzyskiwane rezultaty. Jako że muzyka jest przez człowieka odbierana subiektywnie, opinia autora oraz osób trzecich na temat uzyskiwanych rezultatach również będzie brana pod uwagę. Na podstawie przeprowadzanych testów, system będzie modyfikowany w celu poprawy uzyskanych rezultatów.

Zakres pracy obejmuje zebranie niezbędnej wiedzy na temat rozważanego problemu w tym opis teoretyczny poszczególnych jego aspektów. Zostaną omówione algorytmy mrowiskowe, teoria muzyki oraz zagadnienia realizacji dźwięku w środowisku cyfrowym. Następnie przestawiony zostanie sposób adaptacji ACO [10] do zadania komponowania muzyki. Opisany zostanie proces implementacji systemu oraz omówiona jego architektura. Następnie przeprowadzone zostaną testy oraz badania. Wyniki badań poddane zostaną analizie, po czym nastąpi podsumowanie całej pracy.

2 Algorytmy mrowiskowe

Algorytmy mrowiskowe [1] to probabilistyczne techniki znajdowania rozwiązań w problemach kombinatorycznych i optymalizacji problemów, które można przedstawić w reprezentacji grafowej [37]. Ten stosunkowo nowy gatunek algorytmów został stworzony przez Marco Dorigo w 1991 roku na podstawie obserwacji zachowań mrówek jako jednostek i działań mrowiska jako całości. Podstawowym problemem mrowiska jest znalezienie źródła pożywienia, a następnie przetransportowanie go do kolonii. Mimo iż pojedyncze jednostki są bardzo ograniczone w percepcji i zachowaniach, to efektywność działania kolonii wynika ze współpracy jednostek. Naturalne mrówki podczas poszukiwania pożywienia początkowo zwiedzają okolicę mrowiska w sposób losowy. Gdy mrówka odnajdzie źródło pożywienia, ocenia jego ilość oraz jakość i na tej podstawie zostawia stosowną ilość feromonu na drodze powrotnej do mrowiska. Jest to chemiczny ślad, który jest wyczuwany przez inne osobniki, a przy tym mrówki mają tendencję do wybierania tych ścieżek, na których ślad jest silniejszy. W ten sposób mrówki są w stanie wyznaczyć najkrótszą drogę do źródła pożywienia, co wykazał Deneubourgh i inni w swoich pracach [6, 7]. Rys. 1 ilustruje sposób, w jaki mrówki znajdują najkrótszą drogę. Na początku mrówki losowo wybierają drogę do celu, jednak średnio w jednostce czasu więcej mrówek przejdzie krótszą trasę. Na trasie dłuższej ślad będzie coraz słabszy dlatego, że feromon stopniowo odparowuje. Ślad umocni się na krótszej ścieżce i będzie ona chętniej wybierana przez kolejne nadchodzące mrówki.



Rysunek 1. Mrówki znajdujące najkrótszą drogę

Na podstawie powyższych obserwacji można wyróżnić uogólnione cechy algorytmów mrowiskowych:

- Optymalizacja przeprowadzana jest iteracyjnie przez stałą, określoną na początku, liczbę agentów.
- Agenty komunikują się poprzez informacje zostawianą w otoczeniu, która reprezentowana jest przez ślad feromonowy. Ilość pozostawianego przez mrówkę feromonu zależy od jakości rozwiązania uzyskanego przez mrówkę.
- Agenty losowo przeszukują przestrzeń, faworyzując rozwiązania wskazywane im przez informację z otoczenia.
- Poszczególne mrówki nie posiadają własnej inteligencji a jedynie pamięć, która umożliwia im zapamiętanie trasy.

```
1  Dane wejściowe: graf G, macierz kosztów C, liczba cykli Nc, liczba iteracji Ni,
2   liczba mrówek Nm
3  Dane wyjściowe: najlepsze rozwiązanie T+
4
5  function ACO(G, C, Nc, Ni, Nm)
6      Inicjuj()
7      for Ic ← 1 to Nc do              (Dla każdego cyklu)
8          UstawMrówki()
9          for I ← 1 to Ni do           (Dla każdej iteracji)
10             for k ← 1 to Nm do       (Dla każdej mrówki)
11                 WyznaczKrawędź()     (Wybór krawędzi zgodnie z regułą przejścia)
12                 DodajKrawędźDoRozwiązania()
13                 AktualizujFeromonLokalnie()      (Krok opcjonalny)
14             end for
15         end for
16         OceńMrówki()                        (Oceń mrówki oraz aktualizuj T+)
17         OdparujFeromon()                    (Krok opcjonalny)
18         AktualizujFeromonGlobalnie()        (Krok opcjonalny)
19     end for
20     return T+                          (Zwróć najlepsze rozwiązanie)
21 end function
```

Rysunek 2. Pseudokod ogólnego algorytmu ACO

Algorytmy mrowiskowe to metaheurystyki [10], czyli metody służące za szkielet do konstrukcji heurystyki rozwiązującej problem grafowy. Heurystyka nie gwarantuje osiągnięcia optymalnego rozwiązania, ale znajduje rozwiązanie przybliżone przy akceptowalnych nakładach obliczeniowych.

Rysunek 2 przedstawia ogólną postać algorytmów optymalizacji mrowiskowej. Do tej pory powstało wiele wariantów i modyfikacji algorytmu ACO [8].
Poniżej zostaną omówione jego podstawowe modyfikacje, poczynając od pierwowzoru.

2.1 System mrówkowy

System mrówkowy [9] to pierwszy algorytm ACO zaproponowany przez Doringo w 1991 roku. Został on opracowany na podstawie modelu matematycznego funkcjonowania kolonii mrówek i zastosowano go do rozwiązania problemu TSP (*ang. Travelling salesman problem).* Problem komiwojażera (TSP) polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym [37]. Oznacza to, że należy znaleźć trasę, która przebiega przez każdy wierzchołek grafu dokładnie raz i wraca do punktu początkowego. Punkt początkowy jest tu dowolnym wierzchołkiem grafu. Inicjuje się połączenia między punktami niewielką ilością feromonu.

Budowanie rozwiązania za pomocą AS odbywa się według określonego schematu [9]. Mrówka przemieszcza się do kolejnego węzła z prawdopodobieństwem p, które jest funkcją odległości do celu oraz wielkości śladu feromonowego pozostawionego na danej krawędzi. Prawdopodobieństwo przejścia $P_{ij}(t)$ w danej chwili opisuje się wzorem:

$$P_{ij}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}(t)]^{\beta}}{\sum_{j \epsilon \Omega} [\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}(t)]^{\beta}}, & dla\ j \epsilon \Omega \\ 0 \end{cases}$$

gdzie:
τ_{ij} – natężenie śladu feromonowego,
η_{ij} – wartość heurystycznie oszacowanej jakości przejścia z węzła i do węzła j,
α – parametr sterujący ważnością intensywności śladu feromonowego τ_{ij},
β – parametr sterujący ważnością wartości η_{ij},
Ω – zbiór decyzji, jakie mrówka może podjąć będąc w węźle i.

Pojedyncza mrówka wyposażona jest w pamięć, reprezentowaną przez tablicę tabu, w której zapisywana jest lista węzłów odwiedzonych w danej iteracji. Może być ona wykorzystana do odtworzenia drogi pokonanej przez mrówkę. Pamięć ta jest czyszczona po każdym cyklu.

Parametry α oraz β sterują przebiegiem algorytmu w taki sposób, że: gdy $\beta = 0$ to tylko ślad feromonowy ma wypływ na uzyskane rozwiązanie, co może prowadzić do osiągnięcia optimum lokalnego. Natomiast gdy $\alpha = 0$, mrówki wybierają krawędzie o lepszej informacji heurystycznej.

Aktualizacja śladu feromonowego następuje, po pełnym cyklu algorytmu, kiedy każda z mrówek zbudowała swoje rozwiązanie. Najpierw następuje odparowanie śladu feromonowego na każdej z krawędzi według wzoru:

$$\tau_{ij}(t+1) = (1 - \rho) \cdot \tau_{ij}(t),$$

gdzie:
ρ – współczynnik odparowania śladu feromonowego, $\rho \in \langle 0,1 \rangle$.

Następnie każda z mrówek nakłada na krawędzie, które odwiedziła feromon według wzoru:

$$\tau_{ij}(t+1) = \tau_{ij}(t) + \Delta\tau_{ij}, \ \vee e_{ij} \epsilon S_k,$$

gdzie:

$\Delta\tau_{ij} = f(C(S_k))$ - ilość nałożonego feromonu, zależna od jakości rozwiązania S_k.

Na przykładzie problemu TSP została wykonana analiza [17], której wynik pokazał, że optymalna liczba mrówek wynosi $m = s_1 \cdot n$, gdzie s_1 jest niewielką stałą rzędu jedności, a n jest liczbą wierzchołków grafu. Wykazano, że czasowa złożoność obliczeniowa algorytmu mrówkowego jest rzędu:

$$T(c,n) = O(c \cdot n^3),$$

gdzie:

c – liczba iteracji algorytmu,
n – liczba wierzchołków grafu.

2.2 System mrowiskowy

System mrowiskowy (ang. Ant Colony System) [18] jest usprawnioną wersją systemu mrowiskowego, która została przedstawiona przez Dorigo i Gambardella w 1996 roku. Zmodyfikowana została reguła przejścia pomiędzy stanami poprzez wprowadzenie sposobów wyboru kolejnego stanu przez eksploatację lub eksplorację. Ponadto zmieniony została metoda odkładania śladu feromonowego.

Zmodyfikowana reguła przejścia zakłada losowanie liczby q, spełniającej warunek $0 \leq q \leq 1$. Dana jest również jako parametr wejściowy algorytmu

liczba q_0. Mrówka, będąc w danej chwili czasu, w określonym węźle grafu podejmuje decyzję przejścia do kolejnego stanu, kierując się eksploatacją lub eksploracją. Jeśli $q \leq q_0$ to deterministycznie wybierana jest najlepsza dostępna decyzja (eksploatacja), w przeciwnym razie mrówka losowo podejmuje decyzję (eksploracja) na podstawie prawdopodobieństw obliczonych ze wzoru:

$$j = \begin{cases} arg \max_{r \epsilon \Omega} \{[\tau_{i,r}(t)] \cdot [\eta_{i,r}(t)]^{\beta}\}, & jeśli \ \ q \leq q_0 \ (eksploatacja) \\ S, \quad w \ przeciwnym \ razie \ (eksploracja), \end{cases}$$

τ_{ij} – natężenie śladu feromonowego jako stopień użyteczności branego pod uwagę przejścia,
η_{ij} – wartość heurystycznie oszacowanej jakości przejścia z węzła i do węzła j,
β – parametr sterujący ważnością wartości η_{ij},
S – następna decyzja wylosowana z prawdopodobieństwem:

$$P_{ij}(t) = \begin{cases} \dfrac{\tau_{ij}(t) \cdot [\eta_{ij}]^{\beta}}{\sum_{r \epsilon \Omega} \tau_{ir}(t) \cdot [\eta_{ir}(t)]^{\beta}}, & dla \ j \epsilon \Omega \\ 0 \end{cases}$$

gdzie Ω jest zbiorem decyzji, jakie mrówka może podjąć będąc w węźle i.
Mrówka nakłada ślad feromonowy po każdym przejściu między węzłami na wybraną przez siebie krawędź. Lokalna aktualizacja śladu feromonowego zakłada również jego częściowe odparowanie, co odbywa się według wzoru:

$$\tau_{ij}(t+1) = (1-\varphi) \cdot \tau_{ij}(t) + \varphi \cdot \tau_0,$$

gdzie:
φ – współczynnik lokalnego odparowania śladu feromonowego, $\varphi \epsilon \langle 0,1 \rangle$,
τ_0 – wartość śladu początkowego.

Równocześnie węzeł, który został wybrany jako kolejny, jest dodawany do tablicy tabu, w której zapisywana jest lista węzłów odwiedzonych w danej iteracji.
Ślad feromonowy jest uaktualniany globalnie na koniec każdej iteracji. Jest on nakładany na krawędzie, które należą do najlepszej znalezionej trasy. Ilość feromonu modyfikowana jest zgodnie ze wzorem:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \rho \cdot \frac{1}{L^+},$$

gdzie przejścia z i do j należą do najlepszego w danej iteracji rozwiązania, ρ reprezentuje szybkość odparowywania feromonu, a L^+ jest długością najlepszej trasy.

2.3 Inne rodzaje algorytmów mrowiskowych

Na przestrzeni lat pojawiło się wiele modyfikacji i interpretacji oryginalnego systemu mrówkowego [19]. Algorytmy te w zależności od potrzeb można dość swobodnie modyfikować oraz stosować na nich różne techniki jak na przykład obliczanie równoległe [25].

Tabela 1. Przegląd wczesnych modyfikacji ACO [19]

Nazwa algorytmu ACO	**Rok powstania**
Ant System	1991
Elitist AS	1992
Ant-Q	1995
Ant Colony System	1996
Max-Min Ant System	1996
Ranked-based AS	1997
ANTS	1998
Best-worst AS	2000
Population based ACO	2002
Beam-ACO	2004
Hyper Cube ACO	2004

Warto wspomnieć o opracowanym przez Tomasa Stützle w 1996 roku algorytmie Max-Min Ant system [1, s. 74]. Podstawową różnicą pomiędzy oryginalnym algorytmem jest sposób aktualizacji śladu feromonowego, który polega na tym, że ślad jest nanoszony jedynie przez mrówkę, która uzyskała najlepszą trasę. Drugą ważną różnicą jest jawne ograniczenie dozwolonych wartości śladu feromonowego do zadanego przedziału $[\tau_{min}, \tau_{max}]$.

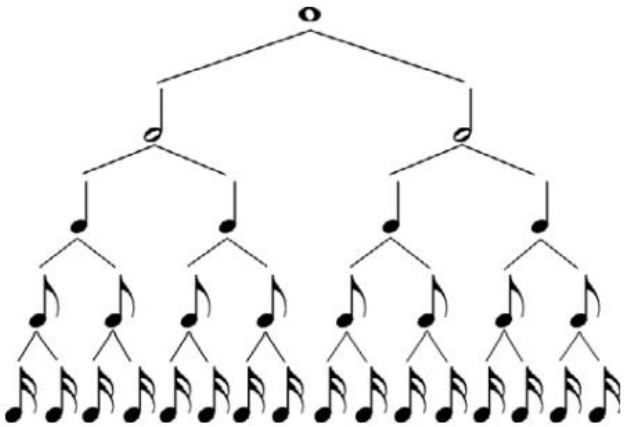
3 Problem badawczy

W niniejszym rozdziale zdefiniowany został problem badawczy pod kątem teorii muzyki oraz cyfrowej realizacji dźwięku. Przedstawione są podstawowe zagadnienia muzyczne, które są konieczne do zrozumienia procesu, jakim jest tworzenie muzyki. Dokonano również przeglądu podobnych istniejących rozwiązań.

3.1 Definiowanie dźwięków

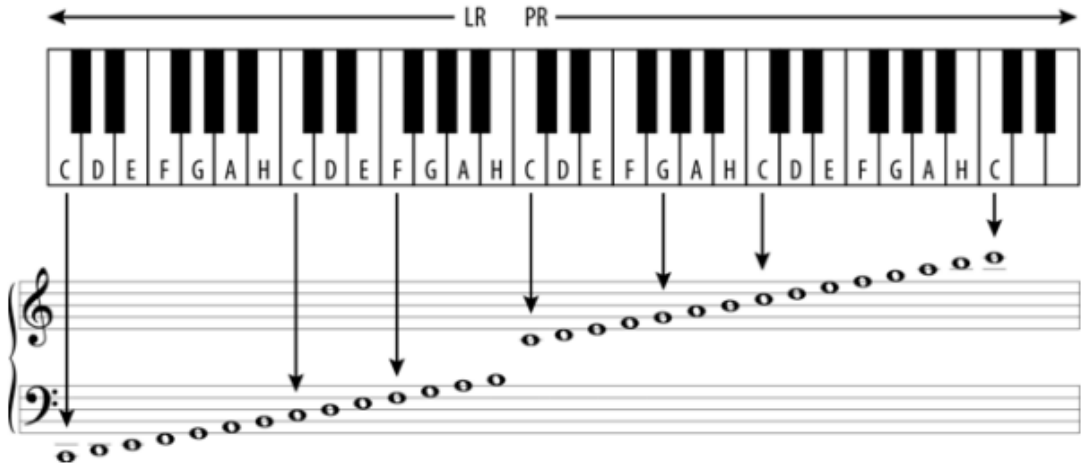
Aby móc definiować dźwięki i zrozumieć zapis nutowy należy na początku wprowadzić pojęcie rytmu oraz bitu [2]. Rytm jest najbardziej podstawowym elementem muzycznym, określający schemat regularnych lub nieregularnych pulsów. Bit z kolei jest pulsacją dzielącą czas na równe odcinki. Dzięki temu możliwe jest wyznaczenie tempa utworu muzycznego, które określa się w jednostce BPM (Beats per minut) oznaczającą liczbę uderzeń w trakcie jednej minuty.

Czas trwania dźwięku określany jest przez rodzaj nuty na pięciolinii. Poniżej przedstawiono dostępne rodzaje nut, a każdy poziom drzewa trwa tyle samo bitów. Poczynając od góry, wyróżniamy: całą nutę, półnutę, ćwierćnutę, ósemkę, szesnastkę.



Rysunek 3. Nuty i ich wartości [2]

Każdy dźwięk posiada określoną częstotliwość oraz czas trwania [2]. Dźwięki standardowo zapisywane są jako nuty na pięciolinii, na której wertykalne położenie określa właśnie wysokość dźwięku. Na poniższym rysunku przedstawiono, jak zapis nutowy odnosi się do dźwięków klawiszy fortepianu.



Rysunek 4. Zapis nutowy i dźwięki w odniesieniu do klawiszy fortepianu [2]

Kolejnym ważnym aspektem muzyki jest jej metrum, gdyż wartość nuty równa jednemu bitowi ulega zmianie w zależności od metrum utworu muzycznego. Określa ono bowiem liczbę bitów w każdym takcie (górna liczba) oraz nutę, która jest równa jednemu bitowi. Najbardziej powszechnym metrum w muzyce popularnej jest $\frac{4}{4}$. Wskazuje ono, że jeden bit ma długość ćwierćnuty, a każdy takt zawiera 4 bity.

Interwałem nazywana jest odległość między dwoma dźwiękami. Wyróżnia się dwa rodzaje interwałów:

- Interwał harmoniczny – oznacza zagranie dwóch dźwięków jednocześnie.
- Interwał melodyczny – występuje, gdy grane są dwa dźwięki jeden po drugim.

Interwały można także sklasyfikować w zależności od tego, czy są one zgodnie, czy niezgodnie brzmiące. Konsonansem [34] nazywa się interwał uważany za zgodnie brzmiący i zachodzi, gdy składowe harmoniczne obu dźwięków się pokrywają. Według Hermanna von Helmholtza do konsonansów należą interwały wyszczególnione w poniższej tabeli, natomiast wszystkie pozostałe interwały oktawy zaliczane są do dysonansów.

Tabela 2. Zestawienie konsonansów

Rodzaj konsonansu	interwał	Różnica półtonów
konsonanse absolutne (brak dudnień)	pryma czysta	0
	oktawa czysta	12
konsonanse doskonałe (dudnienia niesłyszalne)	kwinta czysta	7
	kwarta czysta	5
konsonanse średnie (słabo słyszalne dudnienia)	seksta wielka	9
	tercja wielka	4
konsonanse niedoskonałe (silne dudnienia, jednak niedominujące)	tercja mała	3
	seksta mała	8

Akord to trzy dźwięki lub więcej grane jednocześnie. W muzyce zachodniej większość akordów jest zbudowana z kolejnych interwałów tercji. Oznacza to, że każda nuta w akordzie jest oddalona od poprzedniej o 3 półtony. Wyróżnia się wiele typów akordów, każdy z nich budowany jest według określonej zasady.

Tabela 3. Zasady budowania podstawowych triad [2]

Durowa	Podstawa + 4 półtony + 3 półtony (7 półtonów nad podstawą)
Molowa	Podstawa + 3 półtony + 4 półtony (7 półtonów nad podstawą)
Zwiększona	Podstawa + 4 półtony + 4 półtony (8 półtonów nad podstawą)
Zmniejszona	Podstawa + 3 półtony + 3 półtony (6 półtonów nad podstawą)

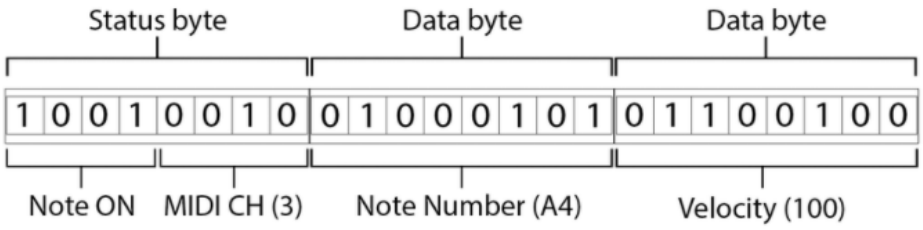
Warto wspomnieć, że muzykę tworzy się w określonej skali [2, s. 81]. W skład danej skali wchodzą tylko określone dźwięki, dlatego, że nie wszystkie dostępne dźwięki pasują do siebie. Przykładowo skala C-dur składa się kolejno z dźwięków C D E F G A H C. Skala durowa ma radosny wydźwięk, natomiast molowa smutny. W muzyce rozrywkowej, począwszy od bluesa, bardzo popularna jest skala pięciostopniowa, czyli pentatonika. Przykładowo pentatonika dla tonacji A-mol lub C-dur składa się z dźwięków A, C, D, E, G. Ze względu na powszechność tej skali i jej elastyczność, implementowany w tej pracy system będzie dostosowany do pracy przede wszystkim z utworami na bazie pentatoniki.

Harmonia jest nauką o łączeniu dźwięków i akordów. Według przyjętych zasad precyzuje, jakie współbrzmienia dźwięków brzmią lepiej lub pomagają osiągnąć pożądany efekt.

Podczas tworzenia melodii harmonijnie brzmiących zazwyczaj wypełniania się brakujące dźwięków akordów wykorzystywanej w utworze progresji. Harmonia [2, s. 165] polega zatem na tworzeniu akordów, których dźwięki pochodzą ze skali, w jakiej została skomponowana melodia.

3.2 Protokół MIDI

Protokół MIDI [5] (ang. Musical Instrument Digital Interface) służy do przekazywania informacji pomiędzy elektronicznymi instrumentami muzycznymi. Dzięki niemu ustandaryzowany został sposób komunikacji pomiędzy urządzeniami. Wykorzystywany jest powszechnie przez instrumenty muzyczne, karty dźwiękowe i w programach do syntezy i obróbki dźwięku. Pliki z komunikatami MIDI są czymś na miarę zapisu nutowego dla urządzeń cyfrowych. Nie zawierają one ścieżek audio, a jedynie zapisaną informację, jak wygenerować dźwięk. Standard ten powstał w roku 1983 [5] z myślą o instrumentach klawiszowych. MIDI zawiera zestaw komend, dzięki którym możliwe jest tworzenie muzyki. Sekwencje takich komend można utworzyć w specjalnym oprogramowaniu, a następnie odtworzyć na dowolnym urządzeniu obsługującym standard. Komunikaty MIDI przesyłają informację m.in. na temat czasu trwania dźwięku, jego natężenia, wysokości czy modulacji. Możliwe jest przesyłanie danych jednocześnie przez 16 równoległych kanałów, a transmisja danych odbywa się w formacie szeregowym z prędkością 31250 bitów na sekundę. Każda komenda składa się jednego do trzech bajtów. Pierwszy to bajt statusu, który określa typ przesyłanej wiadomości i zawsze ma wartość bitu 7 ustawioną na 1. Po bajcie statusu mogą wystąpić 2 bajty danych niosących dodatkowe parametry. Bajt danych rozpoczyna się zawsze od 0 na najstarszej pozycji. Rysunek 5 przedstawia przykładową strukturę słowa w komunikacie MIDI.



Rysunek 5. Struktura słowa w komunikacie MIDI

Do generowania dźwięku na podstawie komunikatu MIDI wykorzystuje się urządzenia lub programy nazywane syntezatorami. Uzyskuje się za ich pomocą możliwość generowania dowolnej barwy dźwięków odczytywanych z komunikatów. Sekwencery służą natomiast do rejestracji, edycji i odtwarzania poleceń MIDI.

Pliki midi zawierają również komunikaty konfiguracyjne dotyczące brzmień instrumentów, metrum utworu, tempa a czasem również tonacji. Ważnym aspektem jest tutaj sposób liczenia czasu trwania nut, gdyż różni się on od tego z tradycyjnej notacji muzycznej. W midi tempo nie jest określane jako BPM, a jako liczba mikrosekund na jeden bit. Występuje również parametr *ticks_per_bit,* określający liczbę tzw. ticków na jeden bit. Każdy komunikat midi zawiera wartość czasu delta (czas względny), który określa, po jakiej liczbie ticków od poprzedniego komunikatu ma być wykonany aktualny komunikat.

3.3 Przegląd istniejących rozwiązań

Historia wykorzystania komputerów w procesie komponowania muzyki sięga lat 50. XX wieku. W 1958 roku Hiller i Isacson [39] stworzyli pionierski projekt na komputerze ILLIAC, którego efektem było skomponowanie utworu dla kwartetu smyczkowego. Proces odbywał się w konwencji „generuj i testuj", dźwięki natomiast były wybierane pseudolosowo za pomocą łańcuchów markowa, a następnie testowane za pomocą heurystycznych reguł kompozycyjnych klasycznej harmonii i kontrapunktu. Doszli oni jednak do wniosku, że metoda łańcuchów markowa nie daje zadowalających efektów w komponowaniu muzyki. Wykorzystana przez Hillera i Isacsona probabilistyczna metoda nie jest metodą sztucznej inteligencji. Pierwsza praca oparta na podejściu regułowym do zadania komponowania muzyki została opublikowana przez Radera w 1974 roku [40]. Wraz z rozwojem algorytmów sztucznej inteligencji, w 1993 powstał system MUSACT, który był jednym z pierwszych systemów wykorzystujących sieci neuronowe do nauki modelu muzycznej harmonii [41]. Od tego czasu powstało wiele systemów komponujących muzykę, a większość z aktualnie dostępnych komercyjnych rozwiązań bazuje na głębokim uczeniu sieci neuronowych.

Najpopularniejszym takim systemem jest AIVA [31] – powstały w 2016 roku system, początkowo był w stanie komponować 2-3 minutowe utwory muzyki klasycznej, a ostatnio jego możliwości zostały poszerzone o inne gatunki muzyczne. Dzięki dostępowi do ogromnej bazy danych utworów (30 000 dla

samej muzyki klasycznej) i zastosowaniu uczenia ze wzmocnieniem sieć jest w stanie generować utwory muzyczne wysokiej jakości. Inne komercyjnie dostępne tego typu systemy to między innymi: Amper Music, Ecrett Music, Humtap, Amadeus Code.

Niewiele dostępnych jest systemów wykorzystujących algorytmy inne niż sieci neuronowe. Pojawiały się jednak projekty badawcze z zastosowaniem algorytmów genetycznych czy systemów mrowiskowych. Pierwszą próbę zastosowania algorytmów mrowiskowych do zadania komponowania muzyki podjęli Christophe Guéret i Nicolas Monmarché w 2004 roku [12], a powstała wtedy publikacja stała się inspiracją do napisania niniejszej pracy magisterskiej. Na uwagę zasługuję projekt AntsOMG [3], komponujący melodie w technice cantus firmus, czy też projekt Geisa i Middendorfa [4], w którym algorytm mrowiskowy zaadaptowany został do komponowania barokowych harmonii.

4 Projekt systemu

W tym rozdziale opisany został sposób na adaptację algorytmu mrowiskowego do realizacji zadania komponowania muzyki. Na podstawie teorii przedstawionej w poprzednich rozdziałach został wykonany teoretyczny projekt systemu, a następnie zostały określone podstawowe wymagania dotyczące aplikacji mającej realizować omawiany algorytm. Algorytm ten został następnie zaimplementowany w aplikacji, która powstała na potrzeby tej pracy. Określone dalej wymagania wynikają ze specyfiki samego algorytmu, możliwości muzycznych, jakie daje format midi oraz wymagań w kontekście badań. W trakcie rozwoju systemu wymagania ulegały modyfikacjom, za sprawą lepszego poznania tematu oraz nowych pomysłów na rozszerzenie funkcjonalności systemu. Przedstawione wymagania określają finalną formę systemu.

4.1 Słownik pojęć

AntsBand – Nazwa aplikacji implementowanej w ramach pracy magisterskiej.
Ścieżka midi – zbiór komunikatów midi, reprezentujących dźwięki grane przez pojedynczy instrument.
Plik wejściowy – Wczytywany do programu, wcześniej utworzony plik midi zawierający jedną lub więcej ścieżek.

Plik wyjściowy (wynikowy) – wygenerowany przez system plik midi, który powstał poprzez przetworzenie algorytmem mrówkowym ścieżek z pliku wejściowego.

4.2 Komponowanie muzyki z zastosowaniem algorytmów mrowiskowych.

Obszerna wiedza z zakresu teorii muzyki daje możliwość komponowania melodii i progresji akordów z niemal matematyczną precyzją. Utwory w gatunkach takie jak jazz czy muzyka klasyczna często komponuje się w oparciu o konkretne założenia formalne, harmoniczne, melodyczne i rytmiczne [2]. Wielu muzyków jednak nie posiada szczegółowej wiedzy z teorii muzyki, a mimo to komponują oni piękne i cenione utwory. Jest tak, gdyż muzykę można również tworzyć na podstawie wcześniej usłyszanych utworów oraz własnych doświadczeń. Autor niniejszej pracy wyznaje te drugie podejście i jako że posiada podstawową wiedzę z teorii muzyki, algorytm również podzieli tę filozofię.

Mrówki nie będą komponować muzyki od podstaw, a raczej improwizować na bazie dostarczonego im zbioru dźwięków. Jako że algorytmy mrowiskowe zostały stworzone do rozwiązywania problemów, które można przedstawić w postaci grafu, dobrze nadają się one do realizacji nadmienionej koncepcji. Opisywany dalej algorytm powstał na bazie inspiracji zaczerpniętych przez autora z publikacji [12]. Jest to jednak odmienne podejście, które zostało tutaj zmodyfikowane i rozszerzone.

W projekcie AntsBand melodie są komponowane na podstawie prawdopodobieństw przejść w grafie. Proces rozpoczyna się od odczytania z pliku wejściowego zdarzeń MIDI. Z każdego zdarzenia, które oznacza odegranie dźwięku, odczytywana jest wartość wysokości tego dźwięku. Dla przykładu z pewnej ścieżki odczytano następującą sekwencję dźwięków: {41, 44, 36, 39, 41, 36}, co oznacza dźwięki {F, G#, C, D#, F, C} zagrane w 3 oktawie. W MIDI wysokości dźwięków określa się w zakresie od 0 do 127, co daje możliwość operowania w 11 niepełnych oktawach i właśnie na tych wartościach pracuje opisywany algorytm.

Kolejnym etapem po odczytaniu dźwięków z pliku wejściowego jest umieszczenie tych dźwięków na wierzchołkach grafu. W trakcie inicjowania grafu na jego krawędziach nakładany jest feromon. Krawędzie łączące dźwięki, które sąsiadowały ze sobą w pliku wejściowym, otrzymują inną wartość feromonu, co

ma na celu danie możliwości nakłonienia mrówek do sugerowania się oryginalną melodią z pliku wejściowego. Inicjowanie feromonu w grafie o n wierzchołkach określa następująca reguła:

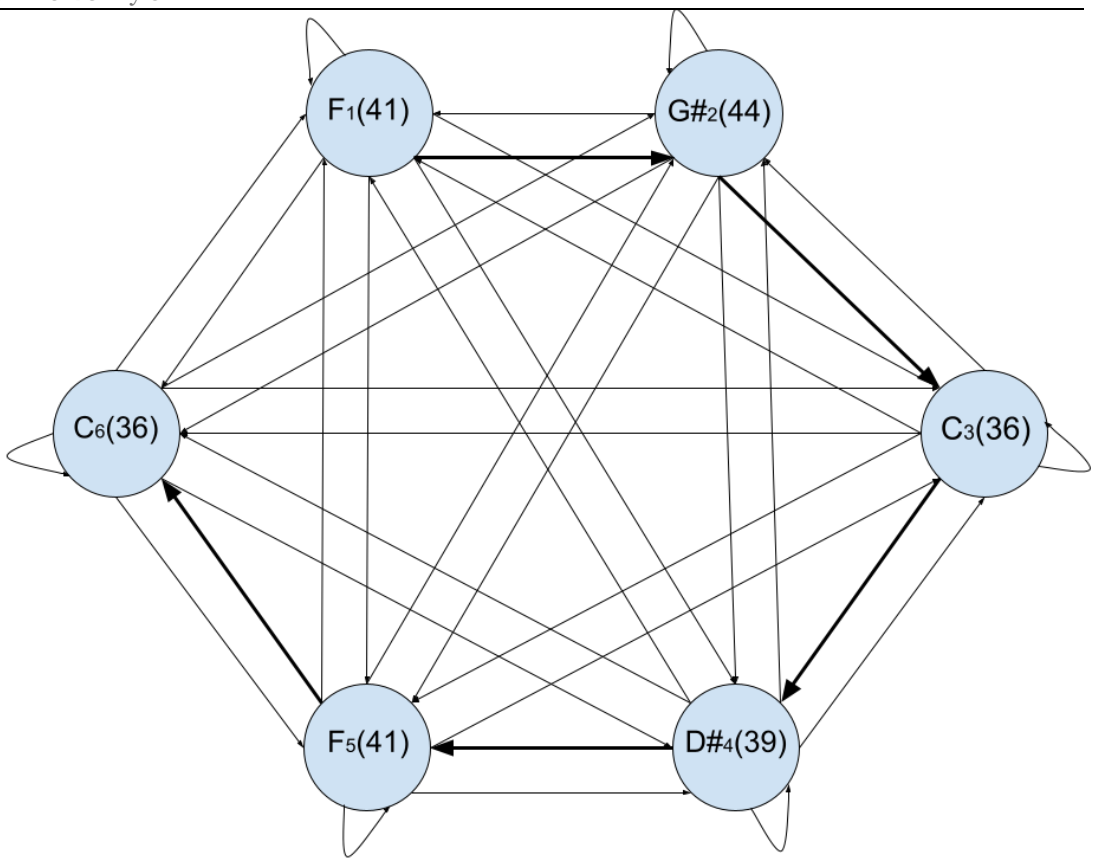
$$\tau_{ij} = \begin{cases} \sigma, & j = i + 1 \\ \frac{1}{n^2}, & j \neq i + 1 \end{cases}$$

Gdzie σ jest stałą wartością inicjowania feromonu dla dźwięków sąsiadujących w sekwencji dźwięków wejściowych. Jednocześnie tworzona jest macierz kosztów przejścia pomiędzy dźwiękami, na podstawie różnicy odległości między dźwiękami. W ten sposób im większa jest różnica w wysokości dźwięków, tym większy koszt przejścia między nimi. Dodatkowo przy tworzeniu macierzy wprowadzono regułę mówiącą, że jeśli dźwięki są tej samej wysokości, to wartość odległości jest losowana spośród wcześniej zdefiniowanego zbioru liczb. Dzięki tej regule można uniknąć wartości zerowego kosztu przejścia pomiędzy węzłami oraz określenie jak bardzo mrówki mają być skłonne do ustawiania tych samych dźwięków obok siebie w melodii wyjściowej. W tabeli 4 przedstawiono przykładową macierz kosztów przejścia pomiędzy węzłami grafu, gdzie zerowe odległości zostały zastąpione wartościami 10 lub 100:

Tabela 4. Przykładowa macierz kosztów przejścia pomiędzy węzłami grafu

	F (41)	G# (44)	C (36)	D# (39)	F (41)	C (44)
F (41)	10	3	5	2	100	5
G# (44)	3	10	8	5	3	8
C (36)	5	8	100	3	5	100
D# (39)	2	5	3	100	2	3
F (41)	10	3	5	2	100	5
C (44)	5	8	10	3	5	10

W grafie dla przykładowej sekwencji dźwięków, przejścia zainicjowane feromonem o wartości σ oznaczono pogrubioną linią:



Rysunek 6. Przykładowy graf dla 6 dźwięków. Węzły zawierają dźwięk, indeks oraz wartość midi w nawiasie.

Gdy graf jest już gotowy, należy znaleźć ścieżkę przejścia pomiędzy jego węzłami, wykorzystując algorytmy AS lub ACS, które zostały opisane w rozdziałach 2.1 oraz 2.2. Każdy z węzłów grafu powinien być odwiedzony dokładnie raz, więc zadanie podobne jest do problemu TSP z tą różnicą, że bez powrotu do węzła początkowego. W systemie AntsBand założono, że długość melodii wyjściowej musi być taka sama jak wejściowej. Dzięki temu uzyskuje się więcej możliwości obróbki gotowego utworu i synchronizacji dźwięków. Algorytm mrówkowy zwraca znalezioną ścieżkę w grafie w postaci kolejnych numerów węzłów. Tak uzyskaną ścieżkę należy następnie przetworzyć na sekwencję odpowiadających węzłom grafu zdarzeń midi. Jako że algorytm mrówkowy operuje jedynie na wartości wysokości dźwięków, pozostałe wartości, takie jak np. czas trwania dźwięku, mogą być pobierane z oryginalnej sekwencji nut z pliku wejściowego lub ustalane na podstawie nowych zdarzeń wynikających z przejścia w grafie. W przypadku pierwszej opcji melodia w utworze wyjściowym będzie się różniła wysokościami dźwięków, a wartości rytmiczne pozostaną bez zmian. W drugim przypadku uzyskuje się zupełnie nową melodię.

Jeśli w pliku wejściowym występuje więcej ścieżek, algorytm jest wykonywany osobno dla każdej z nich lub kilku wybranych. Na końcu w pliku wejściowym ścieżki zostają podmienione na nowo wygenerowane i w ten sposób uzyskuje się gotowy plik wynikowy.

Ze względu na wielomianową złożoność obliczeniową [42] algorytmów mrowiskowych, może się zdarzyć, że przy długich sekwencjach dźwięków, powstanie na tyle duży graf, że czas przetwarzania będzie nieakceptowalnie długi. W tym przypadku w algorytmie AntsBand wprowadza się modyfikację, pozwalającą na rozłożenie danej sekwencji dźwięków na kilka mniejszych części. Przykładowo, jeżeli wejściowa melodia składa się z 128 dźwięków, można ją podzielić na 4 części, uzyskując 32 dźwięki w jednej części. W tym przypadku tworzone są 4 grafy, a dla każdego z nich osobno wykonywany jest algorytm mrowiskowy. Uzyskane w ten sposób sekwencje łączone są na koniec w dowolnej kolejności w celu utworzenia ścieżki wynikowej.

4.3 Wymagania funkcjonalne

Zebrane wymagania funkcjonalne wynikają z wcześniej opisanej teorii działania programu, jak i dodatkowych wymagań, które powstały na drodze tworzenia oprogramowania, a mające potencjalnie interesujący wpływ na uzyskiwane przez program rezultaty.

F01. Wczytanie wejściowego pliku w formacie midi.
F02. Możliwość wyboru przez użytkownika, które ścieżki z wejściowego pliku mają zostać przetworzone, a które pozostawione bez zmian.
F03. Funkcja podziału ścieżek na mniejsze części.
F04. Możliwość wyboru rodzaju algorytmu przetwarzającego oraz wprowadzenie jego parametrów.
F05. Funkcja zmiany długości pliku wyjściowego.
F06. Możliwość zachowania czasów trwania nut z pliku wejściowego lub wygenerowanie nowych na podstawie działania algorytmu.
F07. Odtwarzanie wygenerowanego pliku wyjściowego z możliwością pauzowania oraz zatrzymania odtwarzania.
F08. Wyświetlanie wykresów wysokości kolejnych dźwięków dla wybranej ścieżki z pliku wyjściowego.

F09. Możliwość zapisu wygenerowanego pliku w dowolnej lokalizacji nadając mu wybraną nazwę.

F10. Wyświetlanie obliczonej dla danej ścieżki wartości funkcji ewaluacji oraz stopnia podobieństwa do oryginalnego utworu.

F11. Możliwość odseparowania wybranej ścieżki z pliku wyjściowego. Odseparowaną ścieżkę należy zapisać, a następnie można ją odtworzyć.

4.4 Wymagania niefunkcjonalne

N01. Aplikacja jest przeznaczona na urządzenia z systemem windows.

N02. Aplikacja powinna być odporna na błędy użytkownika. W przypadku wystąpienia błędu (np. nieprawidłowe dane w pliku wejściowym) powinien być wyświetlony stosowny komunikat.

5 Implementacja

Implementacja systemu odbywała się w systemie iteracyjnym. Jako że projekt ma charakter badawczy, planowano rozpocząć od podstawowych elementów systemu, obserwować ich działanie, a następnie je udoskonalać i rozszerzać o kolejne funkcjonalności. Rozdział ten szeroko omawia cały proces rozwoju omawianego systemu wraz z omówieniem jego architektury, poszczególnych komponentów oraz zastosowanych algorytmów.

5.1 Opis aplikacji

Aplikacja AntsBand ma przede wszystkim umożliwić przeprowadzenie badań na potrzeby niniejszej pracy. Drugim zastosowaniem aplikacji ma być umożliwienie zewnętrznemu użytkownikowi wypróbowania możliwości, jakie dają algorytmy mrowiskowe jako narzędzie do komponowania i edycji muzyki. Graficzny interfejs użytkownika powstał głównie dla drugiego zastosowania aplikacji, gdyż badania ze względu na swój charakter będą wykonywane przez dodatkowy skrypt pomijający interfejs graficzny.

Proces korzystania z aplikacji zawsze rozpoczynany jest od wczytania pliku midi zawierającego ścieżki audio. Po załadowaniu pliku wejściowego wyświetlane są znajdujące się na nim ścieżki z możliwością wyboru, które z nich mają być przetwarzane przez algorytm. Następnie użytkownik powinien określić

parametry dla algorytmów mrowiskowych oraz ustawienia dotyczące sposobu przetwarzania ścieżek audio. Po wykonaniu tych czynności można uruchomić proces komponowania, którego rezultat ukaże się w nowym oknie. Z poziomu okna wynikowego, użytkownik może odtworzyć wygenerowaną melodię, przeglądnąć wykresy dla dźwięków w melodii oraz zapoznać się parametrami takimi jak ocena, podobieństwo czy czas wykonania. Okno wynikowe umożliwia także zapis wygenerowanego pliku oraz odseparowanie wybranej ścieżki z możliwością jej zapisu. Okno główne aplikacji umożliwia wielokrotne komponowanie melodii przy różnych ustawianiach parametrów, czego wynikiem będzie otwarcie wielu okien wynikowych, z których można korzystać jednocześnie.

5.2 Zastosowane technologie i narzędzia

Implementacje systemu zdecydowano się wykonać w języku Python [13] w środowisku PyCharm 2021.2.2. Wybór ten uzasadniony jest pokaźnym zestawem bibliotek dostępnych w ramach języka, a zarazem przydatnych z punktu widzenia projektowanego systemu. Jest to język interpretowany, posiadający dynamiczny system typów i automatyczne zarządzanie pamięcią. Wyróżnia się prostą składnią, przejrzystością i zwięzłością. Według autora piszę się w nim szybko, co jest korzystne dla aplikacji badawczej, gdzie potencjalnie często dokonuje się różnych zmian.

Jako że projekt zakłada wykonywanie operacji na plikach midi, o których już wspomniano w części teoretycznej, niezbędna okazała się biblioteka do ich przetwarzania. Mido [15] jest biblioteką umożliwiającą odczytywanie plików, przetwarzanie zdarzeń midi, ich konstruowanie, zapis plików oraz wykonywanie wszelkich możliwych operacji na midi. Komunikaty midi reprezentowane są w formie obiektów Pythona wraz z zestawem funkcji służących do ich przetwarzania.

Do odtwarzania wygenerowanych przez system melodii zastosowano bibliotekę PyGame [16]. Biblioteka ta posiada wiele narzędzi przydatnych do tworzenia gier oraz aplikacji multimedialnych, lecz w niniejszym projekcie jej wykorzystanie ograniczyło się do odtwarzania plików midi.

W części badawczej zasięgnięto po bibliotekę pandas [23], która posiada narzędzia do przetwarzania i analizowania danych. Wykresy występujące w oknach aplikacji oraz w badaniach tworzone są poprzez bibliotekę matplotlib

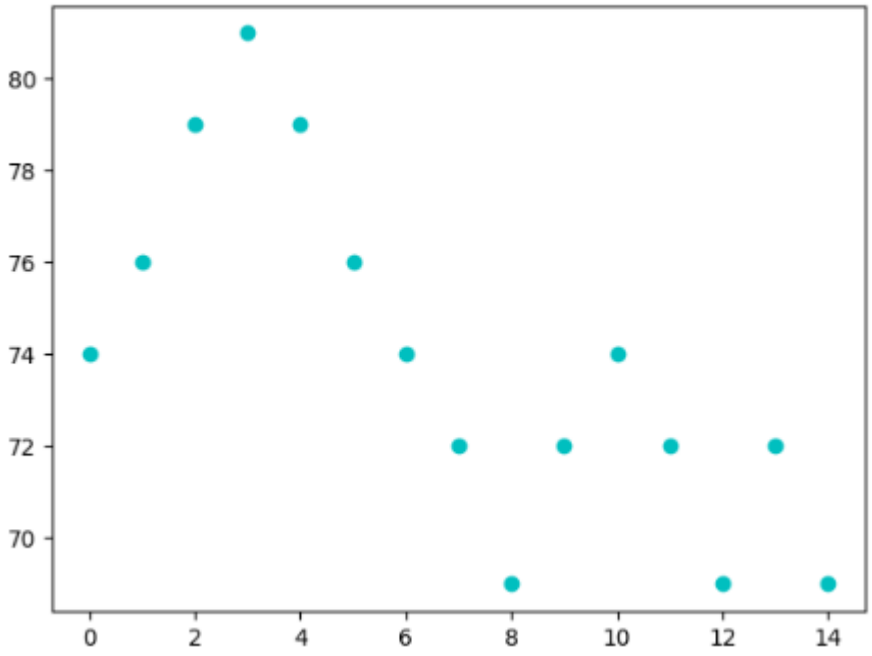
[32]. Graficzny interfejs użytkownika został wykonany z wykorzystaniem biblioteki TKinter, która jest najpopularniejszą tego typu biblioteką dla języka Python. Na koniec prac implementacyjnych, za pomocą narzędzia pyinstaller [35], aplikacja AntsBand została wyeksportowana do pojedynczego wykonywalnego pliku w formacie exe. Pyinstaller pakuje stworzoną aplikacje, interpreter pythona oraz wszystkie niezbędne zależności do jednego pakietu, tak że można go uruchomić na dowolnym komputerze z systemem windows bez konieczności instalowania interpretera pythona.

5.3 Etapy rozwoju aplikacji

W pierwszym etapie implementacji aplikacji założono wykonanie podstawowych i początkowo uproszczonych operacji koniecznych do realizacji zaprojektowanego algorytmu. Początkowo podjęto próbę implementacji systemu z wykorzystaniem biblioteki AcoPy [14], zawierającej gotową implementację ACO. Należało wtedy plik midi przetworzyć do postaci Numpy array, a następnie zastosować bibliotekę NetworkX graph, w celu przetworzenia pliku midi do postaci grafowej. Mnogość i złożoność tych operacji wydawała się autorowi nadmiarowa dla początkowo prostego projektu. Dlatego też zrezygnowano z bibliotek AcoPy oraz NetworkX graph na rzecz własnych implementacji.

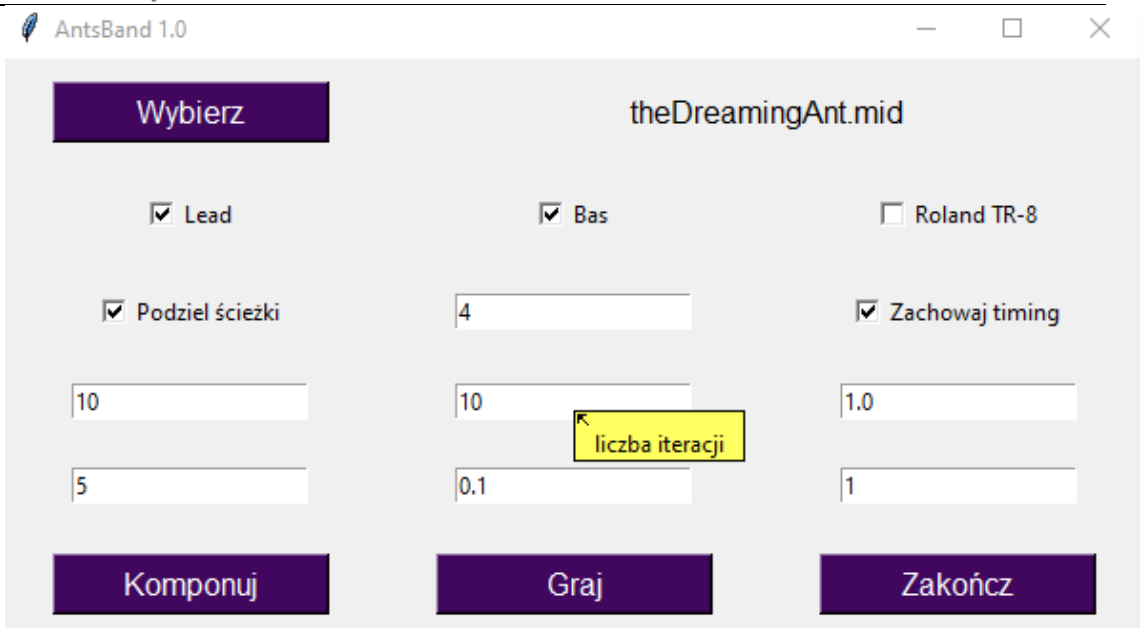
Zaimplementowano bazowy algorytm AS (Ant System), co dało lepszą kontrolę i swobodę jego modyfikacji. Własna implementacja konwersji pliku midi do postaci grafowej również dla pierwszej wersji aplikacji okazała się trafnym pomysłem. Ponadto zaimplementowano odczyt danych z pliku midi, tworzenie pliku midi oraz jego odtwarzanie. Dla danego wejściowego pliku midi wybrać należało jedną ścieżkę melodyczną oraz odczytać jej sekwencję zdarzeń midi. W pierwszej wersji implementacji brany był tylko jeden parametr zdarzenia midi – wysokość dźwięku, gdyż analizowano wtedy proste melodie, w których każdy dźwięk trwał tyle samo. Następnie odczytane zdarzenia należało przekształcić do postaci grafu, tworząc przy tym macierz kosztów przejść pomiędzy poszczególnymi węzłami. W dalszej kolejności AS z ustalonymi parametrami liczby mrówek, iteracje, alfa, beta, ro, generuje nowe przejście pomiędzy wierzchołkami grafu. Uzyskana w ten sposób nowa sekwencja dźwięków musiała zostać przekonwertowana ponownie do formatu midi z pomocą biblioteki mido. Tak spreparowany obiekt midi następnie zapisywany był na dysku oraz odtwarzany na koniec pracy programu. W celu wizualizacji działania algorytmu

wykonano prosty wykres wygenerowanych przez algorytm dźwięków w czasie. Poniżej przedstawiono przykładową sekwencję dźwięków utworzoną przez pierwszą wersję programu. Oś y oznacza wysokości dźwięków a oś x ich kolejność odgrywania.



Rysunek 7. Sekwencja dźwięków utworzona przez pierwszą wersję systemu

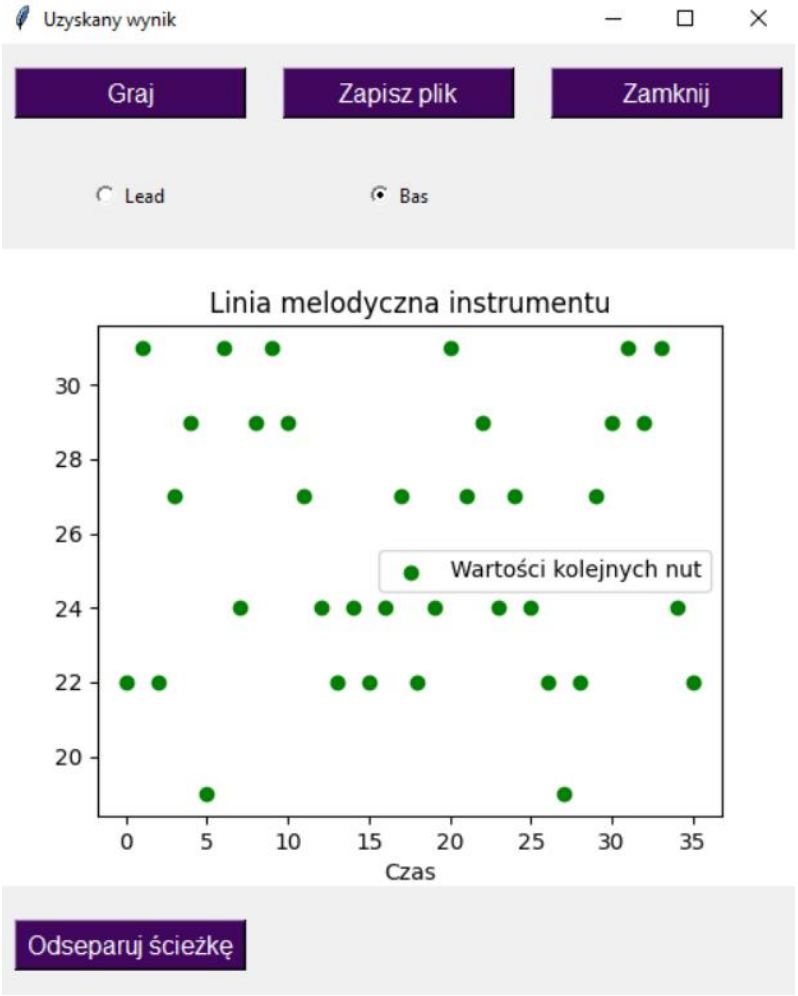
Po zaimplementowaniu podstawowych założeń algorytmu przyszedł czas na udoskonalanie programu, rozszerzenie jego funkcjonalności oraz poprawy błędów. W pierwszej kolejności rozpoczęto pracę nad graficznym interfejsem użytkownika. Nie jest to element niezbędny dla aplikacji badawczej, niemniej jednak pozwoli on na wygodne korzystanie z aplikacji również przez osobę nieposiadającą wiedzy z zakresu programowania oraz estetyczną wizualizację efektów pracy algorytmu. Do graficznego interfejsu wyciągnięto większość konfigurowalnych w algorytmie parametrów oraz możliwych do wykonania akcji.



Rysunek 8. Wstępna wersja głównego okna aplikacji

W międzyczasie zmieniono też wiodący w kodzie paradygmat z imperatywnego [28] na bardziej obiektowy. Pomogło to w lepszym uporządkowaniu kodu oraz zaczęła się powoli kształtować struktura projektu. Na tym etapie zauważono, że przetwarzanie długich plików wejściowych zabiera bardzo dużo czasu, więc rozpoczęto pracę nad mechanizmem podziału ścieżek na frazy, które mogą być przetwarzane osobno. Następnie dodano funkcję zachowania czasów trwania nut, o której wspomniano już w opisie teoretycznym algorytmu. Przy niezachowywaniu oryginalnego „timingu" pojawił się problem w postaci bardzo chaotycznie brzmiącej melodii wyjściowej. Po pewnym czasie udało się poprawić uzyskiwany przy tym ustawieniu rezultat poprzez implementacje mechanizmu kwantyzacji [26], dzięki któremu wartości rytmiczne nut są wyrównywane.

Liczba funkcjonalności w głównym widoku aplikacji zwiększała się, dlatego zdecydowano się na zaimplementowanie okna wynikowego, z poziomu, którego dostępne jest odtwarzanie pliku wynikowego, jego zapis oraz wyświetlanie wykresu dla każdej z wygenerowanych ścieżek melodycznych. Dodano też opcje odseparowania wybranej ścieżki od reszty utworu z możliwością zapisania i odtworzenia w programie. Ma to na celu umożliwienie dokładniejszego wsłuchania się w konkretną ścieżkę. Z poziomu okna głównego aplikacji można otworzyć wiele okien wynikowych i pracować na nich jednocześnie w celu np. porównania rezultatów przy różnych konfiguracjach.



Rysunek 9. Wstępna wersja okna wynikowego

W międzyczasie podchodzono do wielu prób realizacji takich zagadnień, jak branie pod uwagę jednocześnie wysokości dźwięków i wartości rytmicznych, czy znajdowanie cykli w grafie i w ten sposób uzyskiwanie bardziej powtarzalnych melodii. Jednak realizacja tych rozszerzeń okazywała się albo zbyt złożona, albo brakowało pomysłu na ich realizację. Zdecydowano się na pozostanie przy konwencji przetwarzania samych wysokości dźwięków w grafie oraz pozostaniu przy pierwotnych założeniach.

Na dalszym etapie pracy podjęto próbę implementacji funkcjonalności zwielokrotnienia długości utworu wynikowego w stosunku do wejściowego. Opcja ta nie jest bardzo istotna w kontekście badań, jednak daje możliwość uzyskania długiego, ciągle zmieniającego się utworu na bazie krótkiej wejściowej sekwencji dźwięków. Dodanie funkcjonalności wymusiło także modyfikacje funkcji budującej plik wynikowy oraz pojawiły się problemy z odpowiednim

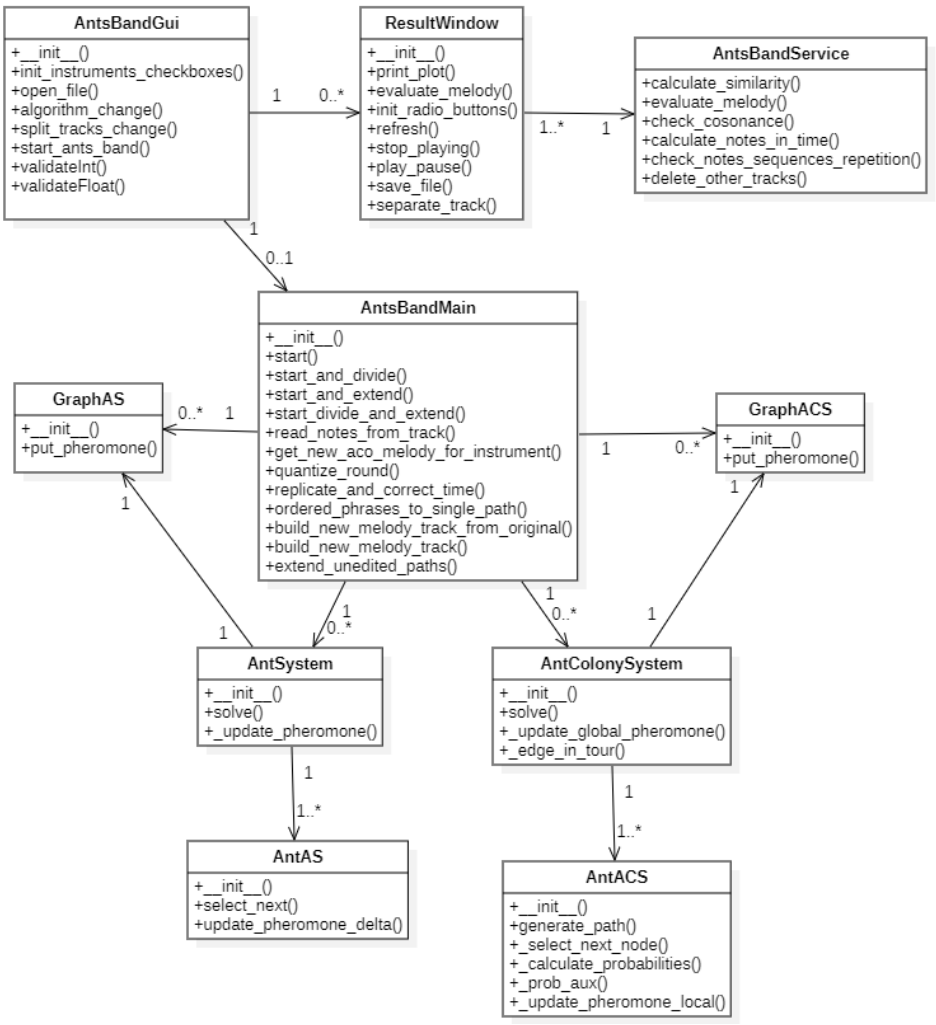
synchronizowaniem w czasie kolejnych części rozszerzanego utworu, co jednak udało się poprawić na dalszym etapie prac.

Gdy większość funkcjonalności była już gotowa, podjęto pracę nad funkcją ewaluacji. Jest to najważniejszy element systemu z punktu widzenia badań, gdyż za jego pomocą możliwe jest uzyskanie informacji zwrotnej w postaci liczbowej na temat jakości uzyskanej melodii. Zastosowano tutaj metodę kryteriów ważonych [20], w której znormalizowane wyniki pochodzące z trzech różnych funkcji oceny są przemnażane przez odpowiednie wagi, a następnie sumowane dając wartość wynikową funkcji ewaluacji. W funkcjach oceny wzięto pod uwagę takie czynniki jak sprawdzanie, czy kolejne nuty mieszczą się we właściwych przedziałach czasowych, szukanie konsonansów w melodii wynikowej oraz sprawdzanie liczby powtarzających się w niej fraz. Następnie zaimplementowano funkcję obliczającą miarę podobieństwa wynikowej melodii do melodii z pliku wejściowego. Sprawdzany jest tutaj stosunek pokrywających się w danym momencie wysokości dźwięków oraz czasów trwania nut. Funkcja ta zwraca zatem dwie wartości w postaci procentowego pokrycia.

Jako ostatnie rozszerzenie do systemu wprowadzono własną implementację algorytmu ACS (Ant colony system), dodając przy tym wybór algorytmu w GUI oraz ustawianie dodatkowych parametrów związanych z tym algorytmem. Na tym etapie okazało się, że jest jeszcze wiele błędów, które należy poprawić lub odpowiednio obsłużyć. Część z nich, takie jak np. brakujący parametr w pliku wejściowym zostało obsłużonych poprzez wyświetlenie stosownego komunikatu w wyskakującym okienku. Pliki midi mogą różnić się od siebie w zależności od tego, w jakim programie zostały utworzone, dlatego niektóre z nich potrafiły generować błędy w przebiegu algorytmu, a inne działały prawidłowo. Poprawki błędów tego typu oraz modyfikacje estetyczne interfejsu użytkownika wykonywano w końcowej fazie implementacji projektu.

5.4 Architektura systemu

Kod aplikacji został podzielony na kilka komponentów realizujących określone zadania i komunikujących się między sobą. Architekturę systemu obrazuje poniżej przedstawiony diagram klas. Połączenia na diagramie oznaczają istnienie instancji klasy wskazywanej przez strzałkę w klasie po drugiej stronie połączenia.



Rysunek 10. Diagram klas aplikacji AntsBand

AntsBandGui – klasa główna interfejsu użytkownika. Definiuje widok umożliwiający wybór pliku oraz wprowadzenie parametrów systemu. W trybie graficznym pracy systemu z jej poziomu kontrolowane jest działanie pozostałych elementów systemu. Zawiera obsługę błędów oraz steruje wykonywaniem odpowiednich funkcji algorytmu generującego (klasa AntsBandMain).

AntsBandMain – klasa ta jest sercem systemu, odpowiadającym za wszystkie operacje przetwarzające ścieżki instrumentów. Następuje tutaj parsowanie plików wejściowych, tworzenie grafów, interakcja z algorytmami mrowiskowymi oraz budowanie plików wynikowych.

ResultWindow – definiuje widok okna wynikowego. Obsługuje odtwarzanie plików, prezentacje wykresów oraz wyników.

AntsBandService – serwis współpracujący z oknem wynikowym i zawierający implementacje funkcji ewaluacji, podobieństwa oraz separowania ścieżki.

AntSystem – klasa zawierająca implementacje algorytmu AS.

AntAS – model pojedynczej mrówki (agenta) działającej w ramach algorytmu AS. Instancje tej klasy tworzone są wewnątrz klasy AntSystem.
AntColonySystem – klasa zwierająca implementacje algorytmu ACS.
AntACS – model pojedynczej mrówki (agenta) działającej w ramach algorytmu ACS. Instancje tej klasy tworzone są wewnątrz klasy AntColonySystem.
GraphAS – model grafu dla algorytmu AS. Jego instancja jest tworzona wewnątrz klasy AntsBandMain na podstawie macierzy kosztów przejścia, a następnie przekazywana do instancji klasy AntAS.
GraphACS – model grafu dla algorytmu ACS. Jego instancja jest tworzona wewnątrz klasy AntsBandMain na podstawie macierzy kosztów przejścia, a następnie przekazywana do instancji klasy AntACS.

5.5 Silnik systemu

Silnik systemu został zaimplementowany w ramach klasy AntsBandMain, która zawiera wszystkie mechanizmy przetwarzające komponowane melodie. W aplikacji AntsBand instancja klasy AntsBandMain jest tworzona w klasie AntsBandGui po zebraniu parametrów i wybraniu akcji komponowania. Aby móc korzystać z metod klasy należy utworzyć jej instancje, podając zestaw parametrów widocznych poniżej.

```
class AntsBand(object):
    def __init__(self, midi_file: MidiFile, tracks_numbers: [int], keep_old_timing: bool, result_track_length: int,
                 algorithm_type: int, ant_count: int, generations: int, alpha: float, beta: float, rho: float,
                 q: int, phi: float, q_zero: float, sigma: float):
```

Rysunek 11. Definicja klasy AntsBand

Klasa AntsBandMain może działać w czterech różnych trybach różniących się sposobem budowania wynikowej sekwencji dźwięków. Dla każdego z trybów zaimplementowano w klasie osobną funkcję:

a) *start* – funkcja realizująca podstawowy przebieg algorytmu. W ramach tej funkcji wejściowe sekwencje dźwięków interpretowane są jako całość, a długość wynikowej melodii jest taka sama jak wejściowej.
b) *start_and_extend* – obsługuje tryb działania, w którym wyjściowa sekwencja dźwięków ma długość n-krotnie dłuższą od wejściowej sekwencji dźwięków, gdzie n jest liczbą całkowitą i $n > 1$. Algorytm mrowiskowy wykonywany jest tu n razy na wejściowej sekwencji

dźwięków, czego skutkiem jest wygenerowanie kilku melodii, które następnie łączone są w całość jako następujące po sobie elementy. W przypadku tego trybu ścieżki, które nie zostały przetworzone przez algorytm mrowiskowy, zostają wydłużone poprzez ich replikowanie.

c) *start_and_divide* – funkcja realizuje przebieg algorytmu, w którym wejściowa sekwencja dźwięków jest dzielona na *k* części, a następnie każda z nich osobno jest przetwarzana przez algorytm mrowiskowy. Następnie w zależności, czy wybrana jest opcja mieszania podziału, sekwencje dźwięków wygenerowane na podstawie części są składane w całość według oryginalnej kolejności lub ich kolejność jest generowana losowo.

d) *start_divide_and_extend* – jest to najbardziej złożona funkcja, która łączy w sobie działanie dwóch poprzednich. Wejściowa sekwencja dźwięków dzielona jest na *k* części, a następnie algorytm mrowiskowy wykonywany jest $k \cdot n$ razy, gdzie *n* to oczekiwana długość utworu.

```
def start(self):
    tracks_data = []
    cost = 0
    for track_number in self.tracks_numbers:
        # odczyt nut i eventów midi ze ścieżek
        line_notes, line_notes_messages = self.read_notes_from_track(self.midi_file.tracks[track_number])
        # ACO dla lini melodycznych
        line_path, cost = self.get_new_aco_melody_for_instrument(line_notes)
        # utworzenie nowej ścieżki dla instrumentu
        line_melody_track = self.build_new_melody_track_from_original(self.midi_file.tracks[track_number],
                                                                       line_path, line_notes_messages)
        # utworzenie pliku wynikowego przez podmianę ścieżek
        self.midi_file.tracks[track_number] = line_melody_track
        tracks_data.append({'track_number': track_number, 'line_path': line_path, 'line_notes': line_notes,
                            'line_melody_track': line_melody_track})
    return [self.midi_file, tracks_data, cost]
```

Rysunek 12. Funkcja *start()* realizująca podstawowy przebieg algorytmu

Na bazie podstawowej wersji algorytmu można wyróżnić kolejne jego etapy:

1. Odczytanie nut z wejściowej ścieżki midi.
2. Wygenerowanie nowej sekwencji nut za pomocą algorytmu mrowiskowego.
3. Utworzenie nowej ścieżki oraz wstawienie jej do pliku wynikowego.

Jeśli ścieżek, które mają zostać przetworzone jest więcej, to powyższe kroki są wykonywane dla każdej z nich osobno.

Operacja odczytu nut z wejściowej sekwencji dźwięków jest taka sama dla wszystkich trybów formowania melodii końcowej. Poniżej zaprezentowana funkcja iteruje po kolejnych zdarzeniach przekazanej w argumencie ścieżce midi.

```
def read_notes_from_track(self, melody_track: MidiTrack):
    notes = []  # zawiera wysokości kolejno zagranych dźwięków w oryginalnej ścieżce
    notes_messages = []  # zawiera kolejne pełne eventy midi z oryginalnej ścieżki
    for i, msg in enumerate(melody_track):  # przeanalizuj każdy event midi z ścieżki
        if msg.type == 'note_on' and msg.velocity != 0:  # interesują nas eventy włączające dźwięk
            notes.append(msg.note)
            j = i+1
            while True:  # do znalezienia eventu wyłączającego dźwięk
                if melody_track[j].type == 'note_off' and melody_track[j].note == msg.note:
                    break  # standardowa forma note_on -> note_off
                if melody_track[j].type == 'note_on' and melody_track[j].note == msg.note \
                        and melody_track[j].velocity == 0:
                    break  # nietypowa forma note_on -> note_on (velocity==0) w fishpolka
                j += 1
            notes_messages.append([msg,melody_track[j]])
    return [notes, notes_messages]
```

Rysunek 13. Odczyt nut ze ścieżki midi

Wyszukiwane są zdarzenia typu *note_on*, które oznaczają odegranie zdefiniowanego przez te zdarzenie dźwięku. Dla każdego zdarzenia włączającego daną nutę, szukane jest zdarzenie odpowiadające za wyłączenie tej nuty. Standardowo za wyłączenie dźwięku odpowiada zdarzenie typu *note_off,* natomiast podczas testów natrafiono na pliki midi, które zawierają tylko zdarzenia typu *note_on*. W takim przypadku dźwięki są wyłączane przez ustawienie w zdarzeniu wartości *velocity* na 0, dlatego funkcja odczytująca została dostosowana do odczytywania również takich plików. Funkcja odczytująca zwraca listę zdarzeń midi składającą się z dwójek włączenie i wyłączenie danego dźwięku, oraz listę kolejno występujących po sobie nut, która to jest wykorzystywana przez algorytm mrowiskowy. Należy tutaj zaznaczyć, że zarówno opisywana funkcja, jak i algorytm mrowiskowy zakładają przetwarzanie pojedynczych dźwięków występujących po sobie. Nie jest zatem możliwe przetwarzanie akordów oraz dwudźwięków. Jeśli w melodii wejściowej występuje akord, to będzie on rozłożony do dźwięków odgrywanych sekwencyjnie. Obsługa akordów jest możliwa i planowana do wprowadzenia w kolejnej wersji aplikacji, jednak wymaga ona bardziej złożonego przetwarzania i rozbudowanej struktury grafu.

Każdy tryb działania programu wykorzystuje do uzyskania nowej melodii funkcję tworzącą graf i generującą nowe przejście pomiędzy dźwiękami za

pomocą algorytmu mrowiskowego. Jeśli ścieżka wejściowa interpretowana jest jako całość, to funkcja ta wywoływana jest tylko raz. Jeśli następuje podział ścieżki na frazy lub wydłużenie utworu to wygenerowanych zostanie kilka sekwencji dźwięków za pomocą poniższej funkcji.

```
def get_new_aco_melody_for_instrument(self, notes: list):
    cost_matrix = []
    number_of_notes = len(notes)
    for i in range(number_of_notes):
        row = []
        for j in range(number_of_notes):
            row.append(self.distance(notes[i], notes[j]))
        cost_matrix.append(row)
    if self.algorithm_type == 0:
        aco = AntSystem(self.ant_count, self.generations, self.alpha, self.beta, self.rho, self.Q)
        graph = GraphAS(cost_matrix, number_of_notes, self.sigma)
        path, cost = aco.solve(graph)
    else:
        acs = ACS(self.ant_count, self.generations, self.alpha, self.beta, self.rho, phi=self.phi,
                  q_zero=self.q_zero)
        graph = GraphACS(cost_matrix, number_of_notes, self.sigma)
        path, cost = acs.solve(graph)
    return [path, cost]
```

Rysunek 14. Uzyskiwanie nowej ścieżki przejścia pomiędzy dźwiękami

Ważnym etapem jest tutaj utworzenie macierzy kosztów przejścia pomiędzy dźwiękami. Dla danej n elementowej listy dźwięków tworzona jest macierz o wymiarze $n \times n$. Koszt przejścia pomiędzy dźwiękami obliczany jest jako wartość bezwzględna różnicy odległości pomiędzy dźwiękami. Przykładowo, jeśli pomiędzy dwoma dźwiękami jest różnica półtonu, to koszt przejścia pomiędzy nimi będzie wynosił 1. W ten sposób mrówki preferują wybór mniejszych interwałów pomiędzy dźwiękami. Jeśli dwa dźwięki mają taką samą wysokość, to koszt przejścia pomiędzy nimi jest losowany spośród liczb: [0.1, 0.5, 1, 5, 10, 100]. Następnie w zależności od wybranego typu algorytmu mrowiskowego tworzona jest jego instancja, odpowiedni graf oraz uzyskiwane rozwiązanie. Zwracane przez algorytm mrowiskowy rozwiązanie jest w postaci listy zawierającej indeksy kolejnych nut.

Po wygenerowaniu nowej sekwencji dźwięków ostatnim etapem jest ich przetworzenie do ścieżki złożonej ze zdarzeń midi odpowiadających uzyskanej sekwencji. Proces ten zaimplementowano na dwa sposoby i są one zamiennie stosowane w zależności od trybu działania algorytmu. Pierwszy sposób polega na podmianie zdarzeń midi w oryginalnej ścieżce wejściowej na nowe zdarzenia wybrane na podstawie uzyskanej ścieżki przejścia. Pętla iteruje po kolejnych

zdarzeniach ścieżki i zamienia je na docelowe. Za pomocą licznika nut dla ścieżki uzyskiwane są kolejne indeksy (zmienna *path*) zwrócone przez algorytm mrowiskowy, na podstawie których pobierane są kolejne zdarzenia midi dla ścieżki docelowej.

```
def build_new_melody_track_from_original(self, melody_track: MidiTrack, path: list, notes_messages: list):
    path_counter = 0
    for i, msg in enumerate(melody_track):
        if msg.type == 'note_on' and (path_counter < len(path)):
            new_on_msg = notes_messages[path[path_counter]][0]
            new_off_msg = notes_messages[path[path_counter]][1]
```

Rysunek 15. Proces budowania wynikowej sekwencji dźwięków

Sposób ten zapewnia odtworzenie oryginalnego czasu trwania nut. Niezależnie od tego, czy w melodii wejściowej były dwudźwięki albo akordy, czasy trwania nut nie powinny się przesunąć. Sposób ten ma jednak ograniczenia, dlatego dla trybu wydłużania utworu lub losowej kolejności komponowanych fraz została utworzona funkcja, która buduje rozwiązanie tylko na podstawie zdarzeń midi uzyskanych w kroku pierwszym algorytmu i ścieżki wygenerowanej przez algorytm mrowiskowy, nie korzystając ze zdarzeń midi dla nut z pliku wejściowego. Jedynie meta zdarzenia, poprzedzające i będące za zdarzeniami dla nut, są kopiowane z pliku wejściowego do wyjściowego. Sposób ten jest elastyczny i daje więcej możliwości w budowaniu pliku wyjściowego, może się jednak zdarzyć, że przy występujących w oryginalnej melodii dwudźwiękach lub akordach, wyjściowa melodia może przesunąć się w czasie, co jest efektem niepożądanym. Dotyczy to opcji zachowywania oryginalnego czasu trwania nut. W przypadku gdy oryginalny czas trwania nut nie jest zachowywany, nowe czasy trwania nut będą brane ze zdarzeń wybranych przez mrówki. W tym przypadku wartości rytmiczne nie są do siebie dopasowane oraz mogą nie mieścić się w odpowiedni sposób w takcie. Z tego względu na potrzeby tego trybu, zaimplementowano mechanizm kwantyzacji [26], który wyrównuje czasy trwania nut i pauz do wartości im najbliższych i zgodnych z obowiązującym w utworze metrum oraz zapisanej w pliku midi liczby tików na bit. Poniżej przedstawiono funkcję przeprowadzającą prostą kwantyzację pojedynczego dźwięku przekazanego w argumencie.

```
def quantize_round(self, value):
    return int(self.ticks_per_semiquaver * round(value / self.ticks_per_semiquaver))
```

Rysunek 16. Funkcja kwantyzująca

Po przeprowadzeniu operacji opisanych powyżej plik wynikowy z nowo wygenerowanymi ścieżkami jest gotowy do odtworzenia.

5.6 Implementacja algorytmów mrowiskowych

Dla języka Python powstało już kilka bibliotek, takich jak na przykład AcoPy [14], które zawierają implementacje systemów mrowiskowych. Biblioteka AcoPy jest dobrze zoptymalizowana i pozwala na łatwe i szybkie jej wykorzystanie do np. rozwiązywania problemu TSP, co autor czynił już w przeszłości w ramach innego projektu. Aplikacja AntsBand jest projektem badawczym, dlatego zdecydowano się na realizację własnych implementacji dwóch algorytmów mrowiskowych. Dzięki temu uzyskano możliwość wprowadzania modyfikacji do algorytmów, lepszej kontroli nad kodem i lepszego dostosowania do potrzeb. Implementacje obejmują najstarszą wersję algorytmu, czyli Ant System oraz jego modyfikację Ant Colony System. Oba algorytmy zostały już opisane w części teoretycznej, natomiast w kolejnych podrozdziałach została opisana ich implementacja w systemie AntsBand.

5.6.1 Ant System

Analizując teorię działania pierwszego, stworzonego przez Doringo algorytmu, można wyróżnić obiekty takie jak graf, mrowisko oraz pojedyncze mrówki. W aplikacji AntsBand zdecydowano się na implementacje systemu mrówkowego w podejściu obiektowym składającego się z trzech wymienionych klas.

Pierwszym krokiem, który trzeba wykonać, zanim zainicjuje się wykonanie algorytmu mrówkowego, jest utworzenie modelu grafu. Graf jest strukturą matematyczną składającą się ze zbioru wierzchołków połączonych krawędziami. W niniejszej implementacji wykorzystywany jest model grafu pełnego, czyli prostego grafu nieskierowanego [37], w którym dla każdej pary węzłów istnieje krawędź je łącząca. Poniżej przedstawiono implementacje grafu dla systemu AntsBand:

```
class GraphAS(object):
    def __init__(self, cost_matrix: list, n: int, sigma: float):
        self.cost_matrix = cost_matrix
        self.N = n
        self.sigma = sigma
        self.pheromone = [[self.put_pheromone(i, j) for j in range(n)] for i in range(n)]

    # zainicjowanie śladu feromonowego zgodnie z oryginalną melodią
    def put_pheromone(self, i: int, j: int):
        if j == i+1:
            return self.sigma
        else:
            return 1 / (self.N * self.N)
```

Rysunek 17. Model grafu dla algorytmu mrówkowego

W systemie AntsBand instancja grafu tworzona jest w silniku systemu przed etapem generowania melodii, co zostało już opisane w rozdziale 5.5. Wierzchołkami grafu są wysokości kolejnych dźwięków w rozpatrywanej sekwencji dźwięków. Sama klasa grafu natomiast nie przechowuje konkretnych wartości węzłów, ale zbiór odległości między węzłami. Klasa grafu zawiera następujące pola, które inicjowane są w konstruktorze:

cost_matrix – macierz kosztów przejścia. Dwuwymiarowa lista zawierająca odległości pomiędzy wszystkimi wierzchołkami grafu.

N – liczba węzłów grafu.

sigma – parametr zawierający wartość inicjalizacji śladu feromonowego dla dźwięków sąsiadujących w oryginalnej melodii.

pheromone – lista zawierająca wartości feromonu na krawędziach grafu.

Pierwsze trzy parametry są ustalane wcześniej w silniku systemu, z kolei tablica feromonu tworzona jest na etapie inicjalizacji grafu. Wszystkie połączenia między wierzchołkami grafu, które nie występują w oryginalnej melodii, inicjowane są feromonem o wartości $\frac{1}{n^2}$. Krawędzie dla dźwięków występujących po sobie w melodii wejściowej są inicjowane feromonu wartością sigma. Mechanizm ten daje możliwość określenia stopnia ważności oryginalnych przejść pomiędzy dźwiękami w procesie komponowania melodii.

Procesem generowania nowego przejścia pomiędzy dźwiękami zajmuje się klasa AntSystem. Po jej zainicjalizowaniu wraz z zestawem parametrów rozwiązanie uzyskiwane jest poprzez przekazanie do funkcji *solve()* odpowiedniego grafu reprezentującego melodię. Poniżej definicję klasy AntSystem wraz z funkcją *solve()*:

```python
class AntSystem(object):
    def __init__(self, ant_count: int, generations: int, alpha: float, beta: float, rho: float, q: int):
        self.ant_count = ant_count
        self.generations = generations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = q

    def solve(self, graph: GraphAS):
        best_cost = float('inf')
        best_solution = []
        for gen in range(self.generations):
            ants = [AntAS(self, graph) for i in range(self.ant_count)]
            for ant in ants:
                for i in range(graph.N - 1):
                    ant.select_next()
                if ant.total_cost < best_cost:
                    best_cost = ant.total_cost
                    best_solution = [] + ant.tabu
                ant.update_pheromone_delta()
            self._update_pheromone(graph, ants)
        return best_solution, best_cost
```

Rysunek 18. Definicja klasy AntSystem i funkcja uzyskania rozwiązania

Klasa inicjuje w swoim konstruktorze standardowy zestaw parametrów dla systemu mrówkowego:
Ant_count – liczba mrówek,
generations – liczba iteracji algorytmu,
alpha – parametr sterujący ważnością feromonu,
beta – parametr sterujący ważnością informacji heurystycznej,
rho – współczynnik odparowania śladu feromonowego,
q – intensywność feromonu – ilość uwalnianego feromonu.

Przebieg algorytmu widoczny jest w ramach funkcji *solve()*, która zwraca najlepsze uzyskane rozwiązanie, w postaci kolejnych indeksów węzłów grafu, oraz koszt uzyskania tego rozwiązania. Funkcja w każdej iteracji tworzy nowy zestaw mrówek (agentów), po czym każda z nich uzyskuje swoje rozwiązanie, przechodząc przez wszystkie węzły grafu. Jeśli nowo uzyskane przez daną mrówkę rozwiązanie jest aktualnie najlepsze, zapamiętywane jest ono w zmiennych *best_cost* oraz *best_solution*. Na koniec każdej iteracji następuje aktualizacja śladu feromonowego w zmiennej *pheromone* grafu, poprzez częściowe odparowanie, a następnie nałożenie feromonów przez każdą z mrówek.

Pozostałe operacje opisywanego algorytmu zawarte zostały w klasie *AntAS*, która reprezentuje pojedynczą mrówkę działającą w ramach mrowiska. Poniżej przedstawiono definicję klasy *AntAS*.

```
class AntAS(object):
    def __init__(self, aco: AntSystem, graph: GraphAS):
        self.colony = aco
        self.graph = graph
        self.total_cost = 0.0
        self.tabu = []
        self.pheromone_delta = []  # lokalny przyrost feromonu
        self.unvisited = [i for i in range(graph.N)]  # lista węzłów jeszcze nie odwiedzonych
        self.eta = [[0 if i == j else 1 / graph.cost_matrix[i][j] for j in range(graph.N)] for i in
                    range(graph.N)]  # informacja heurystyczna
        start = random.randint(0, graph.N - 1)  # rozpoczęcie z losowego węzła
        self.tabu.append(start)
        self.current = start
        self.unvisited.remove(start)

    def select_next(self):
```

Rysunek 19. Definicja klasy AntAS

Klasa *AntAS* posiada następujące pola zdefiniowane w konstruktorze:
colony – instancja klasy *AntSystem,* w ramach której działa mrówka,
graph – instancja klasy grafu, po którym porusza się mrówka,
total_cost – całkowity koszt przejścia dla uzyskiwanego rozwiązania,
tabu – lista węzłów odwiedzonych przez mrówkę w danej iteracji,
unvisited – lista węzłów jeszcze nieodwiedzonych w danej iteracji,
pheromone_delta – lokalny przyrost feromonu na podstawie przejścia w danej iteracji,
eta – Wartość heurystycznie oszacowanej jakości przejścia na podstawie macierzy kosztów przejścia.

```
def select_next(self):
    denominator = 0
    for i in self.unvisited:
        denominator += self.graph.pheromone[self.current][i] ** self.colony.alpha * self.eta[self.current][
                                                                                        i] ** self.colony.beta
    probabilities = [0 for i in range(self.graph.N)]
    for j in range(self.graph.N):
        if j in self.unvisited:
            probabilities[j] = self.graph.pheromone[self.current][j] ** self.colony.alpha * \
                               self.eta[self.current][j] ** self.colony.beta / denominator
    selected = 0
    rand = random.random()  # losowa liczba od 0.0 do 1.0
    for k, probability in enumerate(probabilities):
        rand -= probability
        if rand <= 0:
            selected = k
            break
    self.unvisited.remove(selected)
    self.tabu.append(selected)
    self.total_cost += self.graph.cost_matrix[self.current][selected]
    self.current = selected
```

Rysunek 20. Reguła przejścia do kolejnego węzła w algorytmie AS

Powyższy fragment kodu zawiera implementacje reguły przejścia do kolejnego stanu, opisanej przez wzór w rozdziale 2.1. Na początek obliczany jest mianownik wzoru, a następnie wartości prawdopodobieństw przejść do węzłów jeszcze nieodwiedzonych. Kolejny węzeł wybierany jest przez wylosowanie liczby rzeczywistej z zakresu $\langle 0, 1 \rangle$, a następnie odejmowanie od niej kolejnych prawdopodobieństw. Wybierany jest węzeł, dla którego wartość wylosowanej liczby spadnie poniżej 0. Następnie wylosowany węzeł dodawany jest do listy tabu, usuwany z listy nieodwiedzonych oraz obliczany jest aktualny koszt przejścia. Gdy mrówka wykona w danej iteracji przejście pomiędzy wszystkimi węzłami grafu, nakładany jest na odwiedzone przez nią krawędzie feromon o wartości $\frac{Q}{total\_cost}$.

5.6.2 Ant Colony System

System mrowiskowy jest udoskonaloną wersją systemu mrówkowego. Wprowadzono tu dodatkowy wybór pomiędzy eksploracją a eksploatacją przy wyborze kolejnego węzła oraz zmodyfikowano sposób odkładania śladu feromonowego. Ogólny schemat działania algorytmu jest dość podobny do algorytmu opisywanego we wcześniejszym rozdziale, dlatego opisane zostaną tutaj wszelkie różnice i wprowadzone modyfikacje. Sam kod algorytmu został napisany na podstawie AntSystem, przyjmując taka samą strukturę i implementacje podstawowych mechanizmów.

Podstawowa struktura grafu dla algorytmu ACS jest taka sama jak dla algorytmu AS. Jedyną różnicą jest dodatkowa zmienna, w której zapamiętywane są początkowe wartości śladu feromonowego naniesione w grafie przy jego inicjalizacji. Wynika to z lokalnej reguły aktualizacji śladu feromonowego, według której należy dodać właśnie tę wartość przemnożoną przez współczynnik odparowania lokalnego.

Klasa algorytmu ACS została wykonana na wzór klasy AntSystem. Zawiera zestaw parametrów algorytmu, funkcję uzyskiwania rozwiązania oraz funkcję aktualizującą globalną wartość feromonu. W stosunku do AntSystem zostały tutaj dodane dwa parametry:

phi – współczynnik lokalnego odparowania śladu feromonowego

q_zero – parametr chciwości – określa szanse na wybór kierowania się eksploracją lub eksploatacją w wyborze kolejnego węzła.

```
    self.phi = phi
    self.q_zero = q_zero

def solve(self, graph: GraphACS):
    best_cost = float('inf')
    best_solution = []
    for gen in range(self.generations):
        ants = [AntACS(self, graph) for i in range(self.ant_count)]
        for ant in ants:
            ant.generate_path()
            if ant.total_cost < best_cost:
                best_cost = ant.total_cost
                best_solution = [] + ant.tabu
        self._update_global_pheromone(graph, best_solution, best_cost)
    return best_solution, best_cost

def _update_global_pheromone(self, graph: GraphACS, best: list, lbest):
    for i, row in enumerate(graph.pheromone):
        for j, _ in enumerate(row):
            if self._edge_in_tour(i, j, best):
                graph.pheromone[i][j] = (1 - self.rho) * graph.pheromone[i][j] + self.rho * (1 / lbest)
```

Rysunek 21. Fragment klasy ACS

Ogólny przebieg funkcji *solve(),* jest podobny do implementacji z *AntSystem*. Więcej różnic zawartych jest w implementacji samych mrówek. Funkcja aktualizująca globalnie ślad feromonowy została zmodyfikowana tak, że feromon nakładany jest tylko na krawędzie należące do najlepszej ścieżki.

Klasa *AntACS* reprezentuje mrówkę działającą w ramach mrowiska algorytmu ACS. Całość kodu rozbito tym razem na kilka mniejszych funkcji, a w stosunku do implementacji wcześniejszego algorytmu, zrezygnowano tutaj ze zmiennej *pheromone_delta*. Powodem tego jest zmiana sposobu lokalnej aktualizacji śladu feromonowego, który zakłada, że mrówka nakłada feromon na graf po każdym przejściu do kolejnego węzła.

```
def _update_pheromone_local(self):
    i = self.tabu[-1]
    j = self.tabu[-2]
    self.graph.pheromone[i][j] = (1 - self.aco.phi) * self.graph.pheromone[i][j] + self.aco.phi * \
                                 self.graph.initial_pheromone[i][j]
```

Rysunek 22. Implementacja reguły lokalnej aktualizacji śladu feromonowego w ACS

Kolejną ważną różnicą w stosunku do algorytmu AS jest sposób wyboru następnego węzła w grafie.

```
def _select_next_node(self):
    selected = 0
    q = random.random()
    if q <= self.aco.q_zero:  # eksploatacja
        max_val = 0
        i = self.current
        for g in self.unvisited:
            val = (self.graph.pheromone[i][g] ** self.aco.alpha) * (self.graph.eta[i][g] ** self.aco.beta)
            if val > max_val:
                max_val = val
                selected = g
    else:  # eksploracja
        probabilities = self._calculate_probabilities()
        for i, probability in enumerate(probabilities):
            q -= probability
            if q <= 0:
                selected = i
                break
    return selected
```

Rysunek 23. Reguła wyboru następnego węzła w algorytmie ACS

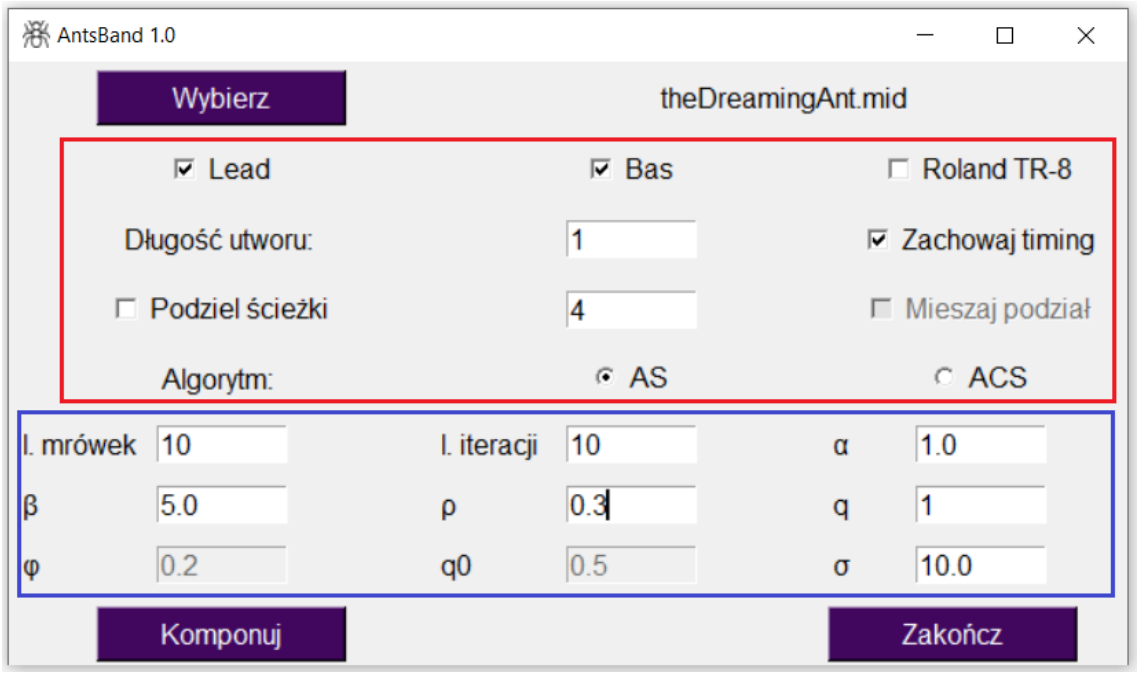
Zgodnie z założeniem algorytmu generowana jest losowa liczba q z zakresu $\langle 0, 1 \rangle$ i na podstawie jej wartości w odniesieniu do parametru wejściowego q_0 wybierany jest tryb eksploatacji lub eksploracji. Eksploatacja polega na wyborze najlepszej z dostępnych decyzji przejścia do następnego węzła według reguły ze wzoru w rozdziale 2.2. Należy tutaj zaznaczyć, że w implementacji nie zrezygnowano z parametru α sterującego ważnością śladu feromonowego w podejmowanej decyzji, mimo że nie ma go w oryginalnym algorytmie ACS. Pozostawienie tego parametru daje więcej możliwości manipulacji algorytmem, a zawsze można nadać mu wartość 1, aby uzyskać oryginalną formułę. Tryb eksploracji przewiduje obliczenie prawdopodobieństw przejść do węzłów jeszcze nieodwiedzonych, po czym wybierany jest następny węzeł metodą ruletki. Eksploracja odpowiada zatem standardowemu modelowi przejścia z algorytmu AntSystem.

5.7 Interfejs graficzny aplikacji

Rozdział ten zawiera dokumentację techniczną komponentów odpowiadających za graficzny interfejs użytkownika oraz ich dokumentację użytkową. W aplikacji AntsBand zaimplementowano dwa komponenty realizujące obsługę graficznego interfejsu użytkownika i wykorzystujące w tym celu bibliotekę tkinter [27].

5.7.1 Widok główny aplikacji

Widok główny jest podstawowym interfejsem obsługi aplikacji w jej trybie graficznym. Z jego poziomu użytkownik może wprowadzić do programu plik wejściowy, ustawić tryb działania algorytmu oraz wprowadzić wartości parametrów dla algorytmu mrowiskowego. Komponent ten zapewnia również komunikacje pomiędzy pozostałymi elementami systemu. Używa silnika systemu do uzyskiwania nowych melodii i przekazuje prezentacje wyników do okien wynikowych. Poniżej przedstawiono interfejs graficzny głównego widoku aplikacji, na którym dodatkowo zaznaczono na czerwono jego sekcje ustawień.



Rysunek 24. Widok główny aplikacji

Przedstawione na rysunku 24 okno umożliwia wykonanie 3 rodzajów akcji, które dostępne są pod fioletowymi przyciskami:

Wybierz – Przycisk otwiera dodatkowe okno wyboru pliku wejściowego. Możliwe jest wybranie jednego pliku w formacie midi, którego nazwa ukaże się, po wybraniu po prawej stronie przycisku „wybierz".

Komponuj – Przycisk staje się aktywny dopiero po wybraniu wejściowego pliku midi. Uruchamia on proces komponowania utworu na podstawie wszystkich parametrów wprowadzonych w oknie głównym. Aby komponowanie było możliwe, należy wybrać przynajmniej jedną ścieżkę z pliku wejściowego.

Zamknij – kończy pracę programu.

Czerwonym prostokątem oznaczono ustawienia dla silnika systemu. Ich wartości domyślne są uzupełniane w polach przy starcie programu. W pierwszym wierszu sekcji znajdują się pola wyboru typu „checkbox" dla ścieżek znajdujących się w pliku wejściowym. Pojawiają się one dynamicznie po wyborze wejściowego pliku, a przez ich zaznaczanie użytkownik może określić, które ścieżki mają być przetwarzane przez algorytm.

Długość utworu – Pole przyjmujące wartości typu całkowitoliczbowego, które określają długość wyjściowego utworu przez jego zwielokrotnienie o podaną tu wartość. Pole obsługuje walidację dla liczb całkowitych, a domyślą wartością jest 1, która oznacza, że długość utworu jest taka sama jak w pliku wejściowym.

Zachowaj timing – Opcja ta jest domyślnie wybrana i oznacza, że w wyjściowych sekwencjach dźwięków czasy trwania nut będą takie same jak w wejściowym pliku. W przypadku odznaczenia opcji czasy trwania nut będą wylosowane przez algorytm mrowiskowy i poddane kwantyzacji.

Podziel ścieżki – Funkcjonalność umożliwia podział wejściowych ścieżek audio na części, których liczbę określa się w polu po prawej stronie. Opcja przydatna jest dla długich plików wejściowych, gdyż znacząco redukuje czas obliczeń poprzez uzyskanie rozwiązania dla każdej części przez osobne mrowisko.

Mieszaj podział – Opcja dotyczy funkcjonalności podziału na ścieżki i określa czy części uzyskane z podziału mają być ustawiane w wyjściowym utworze według oryginalnej kolejności, czy ich kolejność ma być wylosowana. Gdy podział ścieżek nie jest wybrany, opcja ta staje się nieaktywna.

Algorytm – Wybór algorytmu mrowiskowego, który ma zostać użyty do uzyskania rozwiązania. AS – Ant system, ACS – Ant colony system.

Ostatnią sekcją w omawianym widoku aplikacji jest zestaw parametrów algorytmów mrowiskowych. Inicjowane są one z widocznymi na rysunku 24 wartościami domyślnymi. Dostępne są następujące parametry:

Liczba mrówek – liczba agentów biorących udział w uzyskaniu rozwiązania,

Liczba iteracji – liczba iteracji algorytmu mrowiskowego, po których zostanie uzyskane rozwiązanie,

α – ważność feromonu,

β – ważność informacji heurystycznej,

ρ – współczynnik odparowania śladu feromonowego,

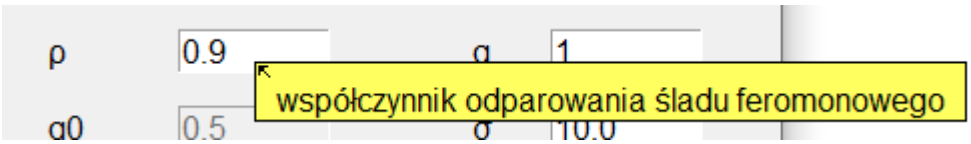
q – intensywność feromonu – ilość uwalnianego feromonu przez mrówkę w algorytmie AS.

φ – Współczynnik odparowania śladu feromonowego (lokalnie, po każdym przejściu). Tylko dla algorytmu ACS.

q0 – współczynnik chciwości. Eksploracja/eksploatacja w algorytmie ACS.

σ – feromon inicjalny. Wartość feromonu jaką są inicjowane w grafie przejścia dla dźwięków sąsiadujących w sekwencji dźwięków wejściowych.

Jeśli chodzi o implementacje opisywanego komponentu, to zastosowano tu rekomendowane dla biblioteki tkinter podejście obiektowe. Przy starcie programu tworzona jest instancja klasy *MainWindow*, która otrzymuje w argumencie instancje klasy tkinter. Następnie w konstruktorze klasy *MainWindow* określane są parametry okna takie jak tytuł, rozmiar czy styl czcionki. Kolejno definiowane są wszystkie widżety [27], czyli elementy takie jak przyciski, pola tekstowe i kontrolki. Elementy te są umieszczane na tak zwanym „gridzie", czyli siatce tworzonej przez obiekt typu *Canvas*, który w przypadku klasy *MainWindow* zawiera 3 kolumny oraz 9 wierszy, w których są umieszczone wcześniej zdefiniowane widżety. Dla lepszego wyjaśnienia znaczenia pól dostępnych w oknie, zdefiniowano zestaw podpowiedzi, które pokazują się po wskazaniu myszką konkretnego elementu. Są to obiekty typu *Baloon,* które przypisywane są do odpowiednich widżetów.



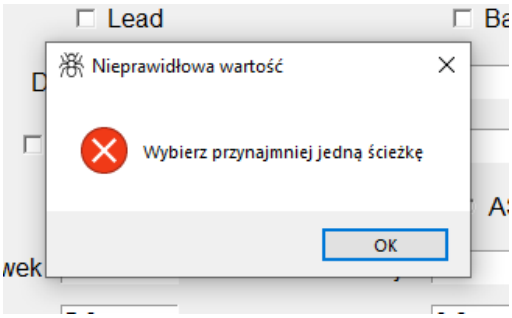
Rysunek 25. Dymek podpowiedzi dla pól tekstowych w głównym oknie aplikacji.

Aby zapewnić możliwość wyboru ścieżek dostępnych w wejściowym pliku midi, konieczne było zaimplementowanie mechanizmu, który będzie dynamicznie wyświetlał w aplikacji opcje po zmianie pliku wejściowego. W tym celu utworzono funkcję, która po zmianie pliku wejściowego najpierw usuwa stare pola wyboru, a następnie przeszukuje plik wejściowy w poszukiwaniu ścieżek audio oraz tworzy dla znalezionych instrumentów nowe pola typu checkbox.

```
def init_instruments_checkboxes(self, mid: MidiFile):
    self.instruments = []
    for label in self.master.grid_slaves():  # usunięcie starych checkboxów
        if int(label.grid_info()["row"]) == 1:
            label.grid_forget()
    for i in range(len(mid.tracks)):  # szukanie instrumentów w pliku
        for msg in mid.tracks[i]:
            if hasattr(msg, 'name'):
                self.instruments.append(
                    {'name': msg.name, 'id': i})  # nazwa instrumentu i index ścieżki
                break
    # print(self.instruments)
    self.start_btn["state"] = "normal"
    for j in range(len(self.instruments)):  # utworzenie nowych checkboxów
        var = IntVar()
        c = Checkbutton(self.master, text=self.instruments[j]['name'], variable=var)
        self.paths_checkbox_dict[self.instruments[j]['id']] = var
        c.grid(row=1, column=j)
```

Rysunek 26. Dynamiczne utworzenie pól wyboru dla instrumentów

W ramach obsługi błędów zaimplementowano komunikaty typu *messagebox* informujące o zaistniałym w programie błędzie. Wyświetlane są błędy nieprawidłowego korzystania z interfejsu, takie jak wybranie akcji komponowania bez określenia ścieżek do przetworzenia. Wychwytywane są również błędy zwracane przez silnik systemu, wynikające z brakujących parametrów w pliku midi czy błędy działania algorytmu.

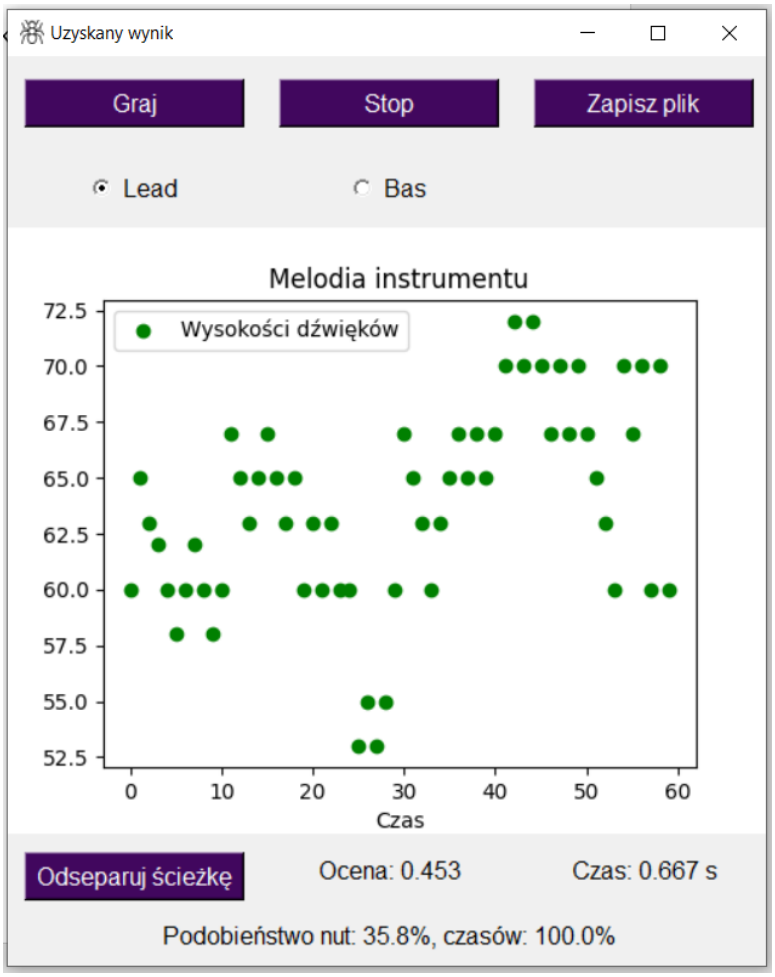


Rysunek 27. Przykład wystąpienia błędu działania aplikacji

Pod przyciskiem Komponuj znajduje się funkcja *start_ants_band(),* która tworzy obiekt klasy AntsBand (silnika systemu) na podstawie parametrów pozyskanych z widżetów, oraz wywołuje odpowiednią do wybranego trybu pracy algorytmu funkcje silnika. Wewnątrz tej funkcji wychwytywane są także ewentualne błędy oraz mierzony jest czas wykonywania algorytmu.

5.7.2 Widok wynikowy

Jeśli generowanie nowego pliku wynikowego przebiegnie pomyślnie, to zostanie utworzona instancja obiektu *ResultWindow,* które wyświetli się jako nowe okno prezentujące wyniki przeprowadzonej operacji komponowania. Podczas działania programu powstać może wiele niezależnych instancji obiektu *ResultWindow,* z których można korzystać jednocześnie.



Rysunek 28. Okno wynikowe aplikacji AntsBand

W oknie znajdują się cztery przyciski, które umożliwiają wykonywanie następujących akcji:

Graj/Pauza – umożliwia odtworzenie lub pauzowanie wygenerowanego pliku. Jeśli plik jest w trakcie odtwarzania to etykieta przycisku zmienia się na „Pauza”.

Stop – Powoduje całkowite zatrzymanie odtwarzania pliku. Ponowne wybranie „Graj” będzie skutkowało odtworzeniem od początku.

Zapisz plik – umożliwia zapisanie wygenerowanego pliku midi w wybranej przez użytkownika lokalizacji, nadając mu dowolną nazwę.

Odseparuj ścieżkę – powoduje wycięcie z pliku wynikowego ścieżki wybranej w drugim wierszu i zapisanie jej pod dowolną nazwą.
W wierszu drugim znajdują się ścieżki, które zostały zmodyfikowane przez algorytm. Dla wybranej ścieżki prezentowany jest odpowiadający jej wykres dźwięków w czasie oraz wartości funkcji oceny i podobieństwa do ścieżki wejściowej. W sekcji dolnej wyświetlany jest również czas generowania przez algorytm całego pliku wynikowego.

W tej klasie została zastosowana ta sama konwencja obiektowego komponentu graficznego tkinter, co w klasie *MainWindow*. Podczas tworzenia instancji obiektu *ResultWindow,* przekazywany jest do jej konstruktora obiekt klasy tkinter, wejściowy plik midi, wygenerowany plik midi, czas wykonania algorytmu oraz obiekt zawierający szczegółowe informacje na temat wygenerowanych ścieżek. Okno wynikowe pracuje na przekazanych mu danych, po pierwsze zapisując wygenerowany plik wyjściowy pod losowo wygenerowaną nazwą, aby zabezpieczyć się przed jego ewentualnym utraceniem w przypadku błędu programu. Kontrolki wyboru ścieżek są dynamicznie tworzone w sposób podobny do tego opisywanego w komponencie głównym aplikacji. Wykres generowany jest na podstawie przekazanych nut i ścieżki przejścia dla wybranego instrumentu. Wykorzystano tutaj obiekt *FigureCanvasTkAgg* dostępnego w ramach biblioteki *matplotlib,* którego implementacje przedstawiono poniżej.

```
def print_plot(self):
    points = self.tracks_data[self.radio_var.get()]['line_notes']
    path = self.tracks_data[self.radio_var.get()]['line_path']
    x = []
    y = []
    for i in range(len(points)):
        x.append(i)
        y.append(points[path[i]])
    figure = plt.Figure(figsize=(5, 4), dpi=100)
    ax = figure.add_subplot(111)
    ax.scatter(x, y, color='g')
    scatter = FigureCanvasTkAgg(figure, self.master)
    scatter.get_tk_widget().grid(row=2, column=0, columnspan=3, sticky='enw')
    ax.legend(['Wysokości dźwięków'])
    ax.set_xlabel('Czas')
    ax.set_title('Melodia instrumentu')
```

Rysunek 29. Implementacja wykresu dźwięków w czasie dla instrumentu

Do odtwarzania plików midi okno wynikowe wykorzystuje obiekt mixer z biblioteki *pygame*. Aby możliwe było odtwarzanie pliku w tym samym wątku,

zaimplementowano funkcję odświeżającą okno i sprawdzającą stan odtwarzania pliku.

```
def refresh(self):
    self.master.update()
    self.master.after(1000, self.refresh)
    if mixer.music.get_busy():
        self.is_playing = True
    else:
        self.play_pause_btn.config(text="Graj")
        self.is_playing = False
```

Rysunek 30. Odświeżanie okna wynikowego i sprawdzanie stanu odtwarzania

Okno wynikowe wykorzystuje serwis *AntsBandService* w celu uzyskiwania wartości wyników funkcji ewaluacji oraz operacji separowania ścieżki. Przy zmianie wybranej ścieżki, wartości uzyskane z serwisu są przypisywane do odpowiednich widżetów, a cała operacja objęta jest blokiem łapiącym wyjątki, który w przypadku ewentualnego wystąpienia zostanie wyświetlony w dodatkowym oknie dialogowym.

5.8 Algorytmy oceny

Podstawą do przeprowadzenia badań jest uzyskiwanie wyników w postaci liczbowej, które następnie można porównywać i analizować. Określenie jakości muzyki na podstawie nawet wielu kryteriów jest zadaniem nietrywialnym, gdyż jej odbiór jest czymś subiektywnym, zależnym od konkretnego słuchacza. Wartość konkretnych cech muzycznych będzie się także różniła w zależności od gatunku muzycznego. Przykładowo powtarzalność pewnych fraz jest bardziej pożądana i oczekiwana w gatunkach takich jak pop czy rock, ale w muzyce jazzowej cecha ta jest traktowana nieco inaczej. Niemniej jednak, aby możliwa była ocena wpływu poszczególnych parametrów algorytmów na końcowy efekt, konieczne było zaimplementowanie funkcji oceny uzyskiwanych rezultatów. W tym celu utworzono funkcję ewaluacji biorącą pod uwagę kilka muzycznych cech oraz funkcję mierzącą stopień podobieństwa do oryginalnego utworu wejściowego.

5.8.1 Funkcja ewaluacji

Funkcję ewaluacji dla systemu AntsBand zdecydowano się zrealizować na podstawie problemu optymalizacji wielokryterialnej [30]. Problem ten polega na

tym, że trzeba jednocześnie uwzględnić wartości kilku funkcji celu przy obliczaniu zadania decyzyjnego. Dla opisywanej tu funkcji ewaluacji utworzono trzy kryteria oceniające generowaną muzykę, na podstawie których obliczane jest kryterium główne metodą kryteriów ważonych [20]. Metoda ta polega na sprowadzeniu zadania wielowymiarowego do zadania jednowymiarowego, co opisuje poniższy wzór:

$$g(x) = \sum_{i=1}^{n} w_i \cdot f_i(x)$$

gdzie:

g(x) – ostateczna funkcja oceny, kryterium zastępcze,
n – liczba kryteriów, $n \in N$,
w_i – waga dla i-tego kryterium,
$f_i(x)$ – pojedyncze i-te kryterium

oraz:

$w_i \in \langle 0, 1 \rangle \quad i \quad \sum_{i=1}^{n} w_i = 1.$

Wartości kolejnych kryteriów są mnożone przez odpowiednie wagi, a następnie sumowane w celu uzyskania wartości ostatecznej funkcji oceny. W aplikacji AntsBand opisywana metoda została zaadaptowana w poniższej funkcji:

```
def evaluate_melody(midi_result: MidiFile, track_data):
    ticks_per_beat = midi_result.ticks_per_beat
    ticks_per_semiquaver = ticks_per_beat / (16 / midi_result.tracks[0][0].denominator)
    numerator = midi_result.tracks[0][0].numerator
    notes = [track_data['line_notes'][track_data['line_path'][i]] for i in range(len(track_data['line_path']))]
    notes_in_time_factor = calculate_notes_in_time(track_data, ticks_per_semiquaver, numerator)
    repeated_sequences_factor = check_notes_sequences_repetition(notes)
    cosonance_factor = check_cosonance(notes)
    evaluation_result = 0.33 * notes_in_time_factor + 0.33 * repeated_sequences_factor + 0.33 * cosonance_factor
    return evaluation_result
```

Rysunek 31. Implementacja funkcji ewaluacji

Funkcja *evaluate_melody* jest wykorzystywana przez okno wynikowe aplikacji i zwraca wartość oceny jako liczbę rzeczywistą w zakresie $\langle 0, 1 \rangle$. Wartość 0 oznacza bardzo niską jakość melodii, a 1 wysoką jakość. Aby obliczyć ostateczną funkcję oceny, najpierw wywoływane są kolejno 3 funkcje kryteriów składowych, które zwracają znormalizowaną wartość z zakresu $\langle 0, 1 \rangle$. Następnie wartość każdego kryterium mnożona jest przez wagę o wartości 0,33 oraz sumowana z pozostałymi. Wartości wag rozdzielono po równo dla każdego kryterium, ale w razie potrzeby, np. gdyby któreś kryterium okazało się ważniejsze, można nimi dowolnie manipulować, tak by ich suma była równa 1.

Pierwszym kryterium składowym jest funkcja sprawdzająca jakość ułożenia nut w czasie w danej sekwencji dźwięków. Funkcja ta iteruje po kolejnych zdarzeniach midi sekwencji dźwięków, zliczając czas, który upłynął od początku utworu. Na podstawie metrum utworu oraz liczby tików zegara MIDI na bit obliczane jest poprzez operacje dzielenia modulo, czy dana nuta trafia w wartość taktu odpowiadającą szesnastce lub ósemce. Każdej nucie możne zostać tutaj przydzielone 0, 1 lub 2 punkty. Na koniec obliczana jest średnia wartość punktów uzyskanych przez pojedynczą nutę. Funkcja ta zwraca wartość $\langle 0, 1 \rangle$, gdzie niskie wartości oznaczają, że dźwięki są nieprawidłowo przesunięte w czasie, a wysokie wartości dają znać, że są precyzyjnie ułożone w rytmie utworu. Dodatkowo wprowadzono tu zabezpieczenie przed dzieleniem przez zero, gdyż niektóre pliki midi mogą mieć nietypowe ułożenie kolejnych zdarzeń.

```
def calculate_notes_in_time(track_data, ticks_per_semiquaver, numerator):
    eval_notes_time = [0] * len(track_data['line_path'])
    time_counter = 0
    notes_counter = 0
    for i, msg in enumerate(track_data['line_melody_track']):
        if hasattr(msg, 'time'):
            time_counter += msg.time
            if msg.type == 'note_on' and msg.velocity != 0:
                if time_counter % ticks_per_semiquaver == 0:  # jeśli mieści się w siatce nut (trafia w szesnastkę)
                    eval_notes_time[notes_counter] += 1
                if time_counter % (ticks_per_semiquaver * 2) == 0:  # ósemka (nuta trafia w którąś z 1/8 taktu)
                    eval_notes_time[notes_counter] += 1
                notes_counter += 1
            if msg.type == 'control_change' and msg.time != 0:  # po ostatnim note off jest control_change wyłączający
                # instrument i posiadający brakujący time - po nim eventów już nie uwzględniamy
                break
    if notes_counter != 0:
        return sum(eval_notes_time) / (notes_counter*2)
    else:
        return 0
```

Rysunek 32. Funkcja sprawdzająca jakość ułożenia dźwięków w czasie

Kolejna funkcja składowa dla kryterium oceny bada powtarzalność sekwencji dźwięków występujących w melodii. W analizowanej melodii wyszukiwane są frazy o długości od 4 nut, a najdłuższa fraza może liczyć połowę długości rozpatrywanej sekwencji dźwięków. Następnie sprawdzane są wszystkie możliwe pozycje wystąpienia fraz w sekwencji dźwięków i zliczane ich powtórzone wystąpienia. Funkcja ta ze względu na 3 zagnieżdżone w sobie pętle ma dość dużą złożoność obliczeniową rzędu $O(n^3)$. Zwracana wartość jest ilorazem liczby znalezionych powtórzeń fraz do maksymalnej liczby powtórzonych fraz, jaka może wystąpić w danej sekwencji dźwięków. Funkcja ta zazwyczaj zwraca niskie wartości, gdyż algorytm nie został zaprojektowany pod kątem powtarzalności

sekwencji. Z tego powodu traktowana jest ona jako mniej ważne kryterium, ale ze względu na zwracanie niskich wartości jej wagę w metodzie kryteriów ważonych pozostawiono taką samą jak dla innych kryteriów.

```
def check_notes_sequences_repetition(notes):  # mierzy powtarzalność sekwencji dźwięków w melodi
    phrase_occurances = 0
    max_phrase_occurances = 0
    minrun = 4   # minimalna długość szukanej frazy
    lendata = len(notes)
    for runlen in range(minrun, lendata // 2):  # iteruje po długościach paternu od 4 po połowy długości melodii
        for i in range(0, lendata - runlen):  # sprawdza wszystkie pozycje dla frazy szukanej długości
            s1 = notes[i:i + runlen]
            j = i+runlen
            while j < lendata - runlen:  # znajduje wszystkie inne frazy za aktualnie szukaną
                s2 = notes[j:j+runlen]
                max_phrase_occurances += 1
                if s1 == s2:
                    phrase_occurances += 1
                j += 1
    return phrase_occurances/max_phrase_occurances
```

Rysunek 33. Funkcja sprawdzająca powtarzalność sekwencji dźwięków w melodii

Trzecim i ostatnim kryterium dla funkcji oceny jest wyszukiwanie konsonansów melodycznych w sekwencji dźwięków. Funkcja ta oblicza wartości interwałów pomiędzy kolejnymi dźwiękami w melodii, a następnie sprawdza, czy wartość ta występuje w tabeli konsonansów. Jeśli tak, to inkrementowany jest licznik konsonansów. Funkcja zwraca wartość będącą stosunkiem liczby znalezionych konsonansów do maksymalnej liczby konsonansów mogących wystąpić w sekwencji dźwięków.

```
# liczy czy pomiędzy dźwiękami występuje kosonans, czy dysonans na podstawie interwałów
def check_cosonance_dissonance(notes):
    cosonance_intervals = [0, 3, 4, 5, 7, 8, 9, 12]  # konsonansy
    cosonances_counter = 0
    for i, prev_note in enumerate(notes, 1):
        interval = abs(prev_note-notes[i])
        if interval in cosonance_intervals:
            cosonances_counter += 1
        if i == len(notes)-1:
            break
    return cosonances_counter/(len(notes)-1)
```

Rysunek 34. Funkcja sprawdzająca wystąpienia konsonansów w melodii

5.8.2 Funkcja podobieństwa

System AntsBand generuje nową muzykę na podstawie już istniejącej, która podawana jest w pliku wejściowym. W zależności od trybu działania programu możliwe jest uzyskanie różnych kompozycji generowanych melodii. Odpowiednio manipulując parametrami, można uzyskać melodię bardzo podobne do oryginalnej lub bardzo się od niej różniącą. Z tego powodu niezbędna jest funkcja obliczająca stopień podobieństwa utworu wyjściowego do oryginalnego, której implementacje przedstawiono poniżej.

```
def calculate_similarity(midi_result: MidiFile, track_data, midi_input: MidiFile):
    track_number = track_data['track_number']
    result_track = midi_result.tracks[track_number]
    all_notes = 0  # notes messages - liczba nut to all_notes/2
    similar_notes = 0
    similar_times = 0
    for i, msg in enumerate(midi_input.tracks[track_number]):  # iteruje po tracku wejściowym
        if msg.type == 'note_on' or msg.type == 'note_off':
            all_notes += 1
            if result_track[i].type == 'note_on' or result_track[i].type == 'note_off':
                if msg.note == result_track[i].note:
                    similar_notes += 1
                if msg.time == result_track[i].time:
                    similar_times += 1
    return [similar_notes/all_notes, similar_times/all_notes]
```

Rysunek 35. Funkcja obliczająca podobieństwo sekwencji dźwięków wyjściowej do wejściowej

Funkcja ta iteruje po kolejnych zdarzeniach midi z pliku wejściowego i sprawdza czy wartości kolejnych nut oraz ich czasy trwania pokrywają się z tymi, które zawarte są w pliku wyjściowym. Dla wartości nut oraz czasów trwania utworzono osobne liczniki, a funkcja zwraca dwie wartości będące ilorazami wartości liczników do liczby wszystkich nut. W przypadku gdy utwór wyjściowy będzie dłuższy od oryginalnego, zostaną porównane tylko te fragmenty które znajdują się w pliku wejściowym.

5.9 Testowanie systemu

Niezbędnym elementem w procesie wytwarzania oprogramowania jest jego testowanie, dzięki czemu możliwe jest znalezienie ewentualnych defektów i zapewnienie stabilnego działania wszystkich elementów systemu. Często w tym celu implementuje się testy jednostkowe [33, s. 141], a w miarę potrzeb również

wydajnościowe lub integracyjne. Jednak w projekcie AntsBand, z uwagi na ograniczone zasoby czasowe i badawczy charakter aplikacji, ograniczono się do testów statycznych kodu, testów manualnych oraz debuggowania kodu. Przyjęto tutaj metodę iteracyjnego sprawdzania wyników po zrealizowaniu kolejnej funkcjonalności aplikacji. Dla każdej funkcjonalności starano się przetestować wszystkie jej możliwe przypadki użycia. Weryfikowano również czy inne ustawienia systemu nie wpłyną na aktualnie testowaną funkcjonalność oraz sprawdzano różne rodzaje plików wejściowych. Niestety wiązało się to z mozolnym debuggowaniem kolejnych zdarzeń midi, zliczaniem czasu trwania nut i statycznej analizie kodu. Bardzo ważnym, a zarazem niecodziennym elementem testowania aplikacji AntsBand było wnikliwe słuchanie wygenerowanych melodii – często był to pierwszy krok do rozpoczęcia dalszych analiz. Najwięcej błędów związanych było z niewłaściwym czasem trwania nut w różnych trybach działania programu, włączając w to kwantyzację. Testy przeprowadzane były na dwóch plikach wejściowych utworzonych przez autora oraz kilku plikach midi ściągniętych z Internetu. Okazywało się, że pliki mają różną strukturę, czasem dodatkowe parametry konfiguracyjne a czasem brak któregoś z wymaganych. Tutaj również napotkano problem z akordami, gdyż algorytm dostosowano do przetwarzania ścieżek zawierających w danej chwili nie więcej niż jeden odgrywany dźwięk. Finalnie udało się dostosować system do odczytu większości plików midi, dodano obsługę wyjątków, natomiast problem obsługi akordów rozwiązano przez ich przetworzenie do melodii.

5.10 Możliwości rozwoju

Kiedy rozpoczynano pracę nad projektem AntsBand planowano realizację podstawowych założeń określonych w celu i zakresie pracy. Wraz z rozwojem aplikacji pojawiały się pomysły, takie jak np. podział ścieżki na części, których cześć udało się zrealizować. Poniżej opisano zakres mechanizmów i funkcjonalności, które można wprowadzić w kolejnej wersji aplikacji.

Aktualna wersja systemu AntsBand wykorzystuje grafy o prostej strukturze, która zakłada przechowywanie w węzłach jedynie informacji o wysokości pojedynczego dźwięku. Strukturę tą można by rozbudować dodatkowo o informację na temat czasu trwania dźwięku oraz większej liczby dźwięków tworzących dwudźwięk lub akord. Dzięki temu możliwe byłoby tworzenie melodii złożonych z akordów oraz nadanie znaczenia czasowi trwania nut jako elementu

współzależnego w tworzeniu przejścia w grafie. Tak rozbudowana struktura grafu wymaga jednak zupełnie innego przetwarzania przez algorytm mrowiskowy. Należałoby znacznie zmodyfikować funkcję obliczania odległości oraz funkcję informacji heurystycznej, która musiałaby obsługiwać akordy, czas trwania nut, a dodatkowo analizować aspekty harmonii.

Generowane melodie w aplikacji AntsBand brzmią harmonijnie, przyjmując założenie, że w utworze wejściowym również są one skomponowane w harmonii, co jest naturalne dla poprawnie skomponowanej muzyki. Niemniej jednak można w ramach rozwoju spróbować wprowadzić reguły, które zapewniają budowanie kolejnej sekwencji dźwięków poprzez analizę dźwięków z wcześniej wygenerowanej melodii dla innego instrumentu. Funkcja taka musiałaby implementować zasady harmonii oraz konieczna byłaby znajomość tonacji utworu, a ta informacja zazwyczaj nie jest zawarta w pliku midi.

Innym usprawnieniem, które należało by wprowadzić w kolejnej wersji aplikacji, jest usprawniona funkcja kwantyzacji. Prostą kwantyzację można by zastąpić bardziej zaawansowaną, która analizuje całkowity czas trwania melodii i dostosowuje na jego podstawie czas trwania pojedynczych dźwięków. Ponadto aktualna wersja aplikacji nie obsługuje instrumentów perkusyjnych, ze względu na to, że takie ścieżki zawierają różne brzmienia perkusji pod określonymi wartościami wysokości dźwięków w zdarzeniach midi. Dla instrumentów perkusyjnych należałoby zaimplementować zupełnie inny algorytm przetwarzania ścieżek, które je zawierają. Ostatnim aspektem, który można by rozwinąć, jest rozbudowa funkcji oceny poprzez dodanie nowych kryteriów.

6 Badania

Niniejszy rozdział prezentuje zrealizowane w ramach pracy badania i eksperymenty. Przedstawiony został cel badań, zakres, który obejmują oraz sposób ich przeprowadzania. Następnie omówiony został przebieg badań, uzyskane wyniki, które zostały dalej przeanalizowane i opracowane w postaci wniosków.

6.1 Cel i plan badań

Podstawowym celem badań jest sprawdzenie jakości uzyskiwanych rezultatów przez zaimplementowany system w zadaniu komponowania muzyki. Wiąże się to z wykonaniem wielu testów zawierających różne zestawy parametrów wejściowych systemu. Następnie zebrane w ten sposób dane będą sprawdzane pod kątem korelacji pomiędzy parametrami wejściowymi a parametrami wyjściowymi takimi jak funkcja oceny, czas wykonania algorytmu czy podobieństwo do utworu wejściowego. Kolejnym celem jest wyłonienie konkretnych zestawów parametrów systemu dających najlepsze rezultaty. System będzie badany w różnych trybach pracy, które posiada aplikacja AntsBand. Dla kilku wybranych trybów, o których mowa w rozdziale 5.5, będą sprawdzane różne zestawy parametrów. Badania obejmą również porównanie dwóch zaimplementowanych algorytmów mrowiskowych oraz porównanie jakości rozwiązań uzyskanych w poszczególnych trybach działania aplikacji.

6.2 Metodyka i sposób realizacji badań

Pierwszym etapem w realizacji badań jest utworzenie zestawu danych wynikowych na podstawie przeprowadzonych wielokrotnie testów algorytmu. W tym celu sporządzono skrypt, napisany w języku Python i należący do projektu AntsBand. Pierwszą operacją realizowaną przez skrypt jest utworzenie zestawów parametrów testowych do przeprowadzenia badań. Dla określonych wartości parametrów wejściowych algorytmu tworzone są wszystkie możliwe ich kombinacje. Bazowe wartości parametrów zostały dobrane na podstawie sugestii zaczerpniętych z literatury [21]. Przedstawiony poniżej przykładowy fragment kodu generuje 7200 zestawów wejściowych danych testowych, co jest dużą liczbą, dającą jednak możliwość dokładnego sprawdzenia wszystkich kombinacji.

```
ant_counts = [1, 5, 10, 30, 50]
generations = [1, 10, 20, 50]
alphas = [0.1, 0.5, 2.0, 5.0]
betas = [0.1, 1.0, 2.0, 5.0, 10.0]
rhos = [0.2, 0.5, 0.9]
qs = [1, 5]
sigmas = [1, 5, 10]
param_sets = []
for a, count in enumerate(ant_counts):
    for b, gen in enumerate(generations):
        for c, alp in enumerate(alphas):
            for d, bet in enumerate(betas):
                for e, rho in enumerate(rhos):
                    for f, q in enumerate(qs):
                        for g, sig in enumerate(sigmas):
                            param_sets.append({'ant_count': count, 'generations': gen,
                                               'alpha': alp, 'beta': bet, 'rho': rho, 'q': q, 'sigma': sig})
Parallel(n_jobs=6, require='sharedmem')(
    delayed(has_shareable_memory)(get_test_mean(params)) for params in param_sets)
```

Rysunek 36. Utworzenie zestawów parametrów testowych do przeprowadzenia badań

Następnie zestawy parametrów przekazywane są do funkcji zbierającej wartości rozwiązań wygenerowanych za pomocą silnika systemu AntsBand. Wykonywanie tych operacji zostało zrównoleglone [25] poprzez wykorzystanie mechanizmu *Parallel* z biblioteki joblib [22], co zaowocowało przyśpieszeniem przetwarzania. Funkcja *get_test_mean()* generuje rozwiązania wykonując algorytm 10 razy dla każdego zestawu parametrów, a następnie obliczając średnią z uzyskanych w ten sposób wyników. Poza wykonaniem algorytmu generowania melodii mierzony jest tutaj czas jego wykonania oraz obliczane są wartości funkcji ewaluacji i podobieństwa. Uśrednione wyniki zapisywane są w formie słownikowej do zbiorczej listy wyników, a po wykonaniu operacji dla wszystkich zestawów parametrów wyniki zapisywane są do pliku csv. Dodatkowo dla każdego testu zapisywany jest jeden plik wynikowy midi w celu umożliwienia późniejszego odsłuchania rezultatów. Poniżej zawarto pełny zestaw danych uzyskiwanych po wykonaniu jednego testu:

Tabela 5. Zestaw danych wynikowych testu

Parametry wejściowe	Parametry wyjściowe
ant_count – liczba mrówek	evaluation_result – wynik funkcji ewaluacji
generations – liczba iteracji	notes_in_time_factor – kryterium ułożenia dźwięków w czasie
alpha – α	repeated_sequences_factor – kryterium powtarzalności sekwencji
beta – β	cosonance_factor – kryterium konsonansów
rho – ρ	similar_notes_factor – podobieństwo nut
q – intensywność feromonu	similar_times_factor – podobieństwo czasów
sigma – σ	execution_time – czas wykonania
phi – φ (tylko ACS)	cost – koszt przejścia w grafie
q_zero (tylko ACS)	midi_filename – nazwa pliku midi

Kolejnym etapem badań jest analiza uzyskanego zbioru danych wynikowych. Na potrzeby analizy sporządzono kolejny skrypt, który pobiera dane wynikowe z wcześniej utworzonego pliku csv, a następnie przetwarza, wykorzystując w tym celu biblioteką *pandas* [23]. Jako pierwsza obliczana jest korelacja [24], która polega na zbadaniu wpływu parametrów wejściowych na wartości parametrów wyjściowych. Zastosowany został współczynnik korelacji rang Spearmana [24], ze względu na to, że nie wiadomo czy zależności pomiędzy zmiennymi są zawsze liniowe oraz badane zmienne mają charakter jakościowy. Omawiana metoda polega na nadaniu wartościom zmiennych rag. Rangi to numery miejsca w szeregu niemalejącym dla wartości danej zmiennej. Współczynnik korelacji rang Spearmana oblicza się ze wzoru:

$$R_{xy} = 1 - \frac{6 \cdot \sum_{j=1}^{n} (X_j - Y_j)^2}{n(n^2 - 1)}$$

Wartość obliczonego współczynnika korelacji jest liczbą z zakresu $\langle -1, 1 \rangle$. Dodatnia wartość współczynnika informuje, że wzrostowi wartości zmiennej X towarzyszy wzrost wartości zmiennej Y. Ujemna wartość natomiast oznacza, że gdy X rośnie, to Y maleje. Im większa wartość bezwzględna współczynnika tym wyższa zależność pomiędzy zmiennymi. Wartość 0 oznacza brak zależności.

Skrypt realizujący analizę po pierwsze wczytuje dane wygenerowane we wcześniejszym etapie, łączy serie danych w jeden zbiór oraz usuwa atrybuty nieistotne z punktu widzenia analizy. Następnie obliczana jest macierz korelacji

metodą spearmana, która dalej jest filtrowana i sortowana. Przetworzone wyniki korelacji są zapisywane w dwóch plikach: jeden zawiera wszystkie wartości korelacji a drugi odfiltrowane bardziej znaczące korelacje z przedziałów $\langle 1, 0{,}4\rangle$ oraz $\langle -0{,}4, -1\rangle$. Skrypt tworzy również wykres dla pełnej macierzy korelacji, wyświetlając go i zapisując w danych wynikowych.

```
correlation = data.corr(method='spearman')
corr_series = correlation[correlation < 1].unstack().transpose()\
    .sort_values(ascending=False)\
    .drop_duplicates()  # przerobienie macierzy korelacji na serie, pofiltrowanie i sortowanie
corr_series.to_csv(root_path + folder_name + '/correlation_' + folder_name + '_all_sorted.csv')
positive_corr = corr_series[corr_series > 0.4]  # odfiltrowanie tylko istotniejszych korelacji
negative_corr = corr_series[corr_series < -0.4]
filtered_corr = pd.concat([positive_corr, negative_corr])
filtered_corr.to_csv(root_path + folder_name + '/correlation_' + folder_name + '_filtered.csv')
```

Rysunek 37. Wyznaczanie macierzy korelacji atrybutów

Na koniec analizowany zbiór danych przeszukiwany jest w celu znalezienia największych i najmniejszych wartości parametrów wynikowych. Tworzony jest kolejny plik wynikowy zawierający nazwę parametru, typ (min/max), wartość parametru oraz identyfikator pełnego testu zawierającego znalezioną wartość. Ponadto pełne wyniki testów zawierające wszystkie parametry systemu, przy których została osiągnięta wartość max/min dla danego parametru wyjściowego są zapisywane w globalnych plikach, zawierających najlepsze wyniki z testów T1 do T5. Pliki te podzielono według nazw parametrów wyjściowych i wartości max/min. Dzięki temu możliwe będzie porównanie parametrów systemu, jakie zawierały poszczególne testy w kontekście osiągnięcia maksymalnego lub minimalnego wyniku danego parametru wyjściowego systemu.

```
for i, column in enumerate(result_columns):
    row_min_id = data[column].idxmin()
    min_val = data[column].min()
    file_min = open(root_path + '/max_min_results/' + file_names[i] + '_min.txt', 'a')
    file_min.write(data.loc[[row_min_id]].to_string() + ' test_name: ' + folder_name + ' \n')
    file_min.close()
    row_max_id = data[column].idxmax()
    max_val = data[column].max()
    file_max = open(root_path + '/max_min_results/' + file_names[i] + '_max.txt', 'a')
    file_max.write(data.loc[[row_max_id]].to_string() + ' test_name: ' + folder_name + '\n')
    file_max.close()
    max_min_data.append({'parameter': column, 'type': 'min', 'value': min_val, 'row_id': row_min_id})
    max_min_data.append({'parameter': column, 'type': 'max', 'value': max_val, 'row_id': row_max_id})
max_min_df = pd.DataFrame(max_min_data)
max_min_df.to_csv(root_path + folder_name + '/max_min_' + folder_name + '.csv', index=False)
```

Rysunek 38. Znajdywanie największych wartości atrybutów i zapis najlepszych wyników

Badanie zostały przeprowadzone na komputerze z systemem Windows 10, posiadającym 4 rdzeniowy, 8 wątkowy procesor intel i7 oraz 16gb pamięci RAM. Skrypty obliczeniowe były uruchamiane równocześnie z poziomu środowiska PyCharm, co pozwoliło na zaangażowanie większej liczby rdzeni procesora i tym samym przyśpieszenie obliczeń.

6.3 Przebieg badań i omówienie uzyskanych wyników

W ramach badań przygotowano 5 testów obejmujących różne konfiguracje systemu AntsBand i generujących wyniki dla wielu zestawów parametrów wyjściowych. W testach zastosowano 3 różne konfiguracje systemu:

- K1 – Sekwencja dźwięków przetwarzana jako całość przez jedno mrowisko. Czas trwania dźwięków jest zachowany z utworu wejściowego. Domyślna długość utworu.
- K2 – Sekwencja dźwięków podzielona na 4 części i przetwarzana przez osobne mrowiska. Kolejność przetworzonych części jest losowa w utworze wyjściowym. Czasy trwania dźwięków są nowe i poddane kwantyzacji. Domyślna długość utworu.
- K3 – Sekwencja dźwięków podzielona na 4 części i przetwarzana przez osobne mrowiska, a domyślna kolejność części jest zachowana w utworze wyjściowym. Czas trwania dźwięków jest zachowany z utworu wejściowego. Domyślna długość utworu.

Kolumna „liczba wykonań” oznacza liczbę badanych w ramach testu różnych zestawów parametrów wejściowych algorytmu mrowiskowego. Dla każdego

zestawu został wygenerowany pojedynczy rezultat. Testowano różne wariacje następujących wartości parametrów:

- Liczba mrówek = {1; 5; 10; 20; 30; 60}
- Liczba iteracji = {1; 10; 20; 30; 50}
- α = {0,1; 0,5; 2; 5}
- β = {0,1; 1; 2; 5; 10}
- ρ = {0,2; 0,5; 0,9}
- q = {1; 5}
- φ = {0,2; 0,5; 0,9}
- q0 = {0,2; 0,5; 0,9}
- σ = {1; 5; 10; 20}

Poniższa tabela przedstawia charakterystykę testów przeprowadzonych za pomocą pierwszego skryptu opisanego w poprzednim podrozdziale.

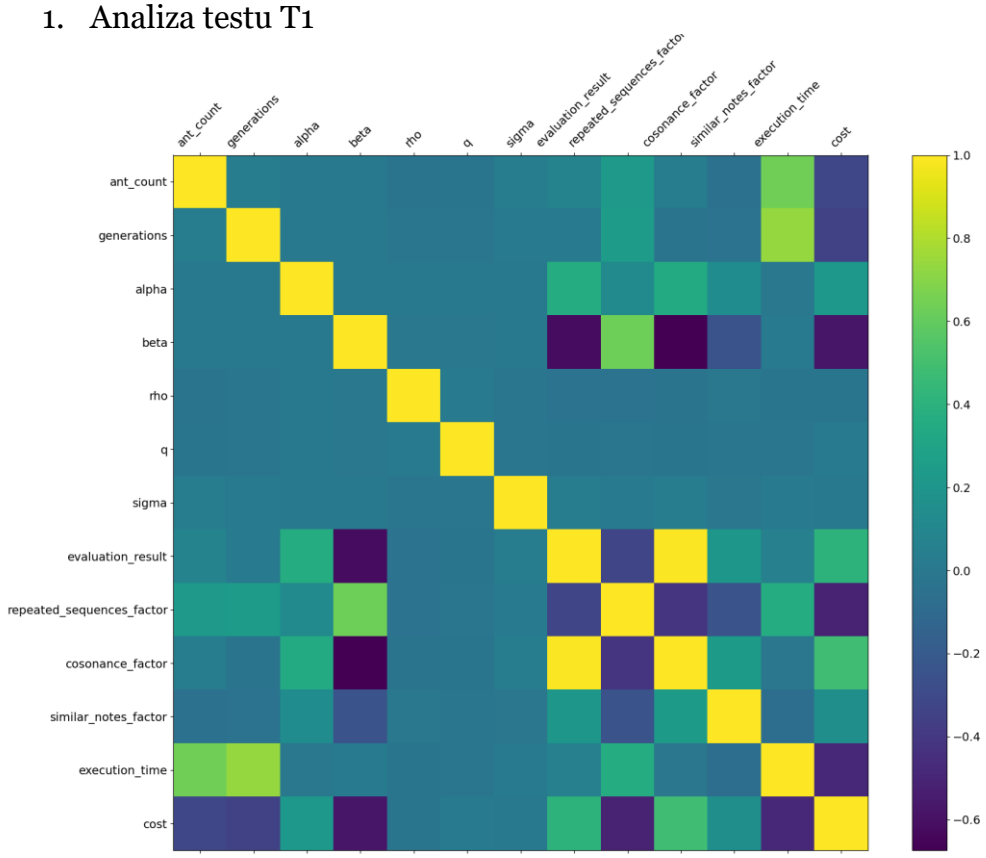
Tabela 6. Charakterystyka testów wykonanych w ramach badań

Oznaczenie testu	**Algorytm**	**Konfiguracja systemu**	**Plik i przetwarzane ścieżki**	**Liczba wykonań**
T1	AS	K1	theRockingAnt.mid, ścieżka nr 3	4374
T2	AS	K2	theRockingAntDrums.mid, ścieżka nr 3	5760
T3	ACS	K1	theRockingAnt.mid, ścieżka nr 3	4212
T4	ACS	K2	theRockingAntDrums.mid, ścieżka nr 3	9072
T5	AS	K3	theDreamingAnt.mid, ścieżki nr 2 i 3	5760

Każdy z testów został podzielony na 2 etapy, zawierające inne zestawy parametrów, co miało na celu przyśpieszenie obliczeń poprzez równoległe wykonanie. Podczas wykonywania testów zauważono, że dla tych samych zestawów testowych testy w konfiguracji k2 wykonywały się prawie 3 razy szybciej niż testy w konfiguracji k1. Oznacza to, że zgodnie z wcześniejszymi założeniami, tryb podziału melodii na części znacząco przyśpiesza wykonywanie algorytmu.

Po otrzymaniu wszystkich wyników testów przeprowadzono analizę uzyskanych danych. Poniżej przedstawiono dla każdego z testów wyniki obliczonej korelacji oraz największe i najmniejsze uzyskane wartości dla poszczególnych parametrów.

1. Analiza testu T1



Rysunek 39. Macierz korelacji dla testu T1

Tabela 7. Tabela wybranych wartości korelacji dla testu T1

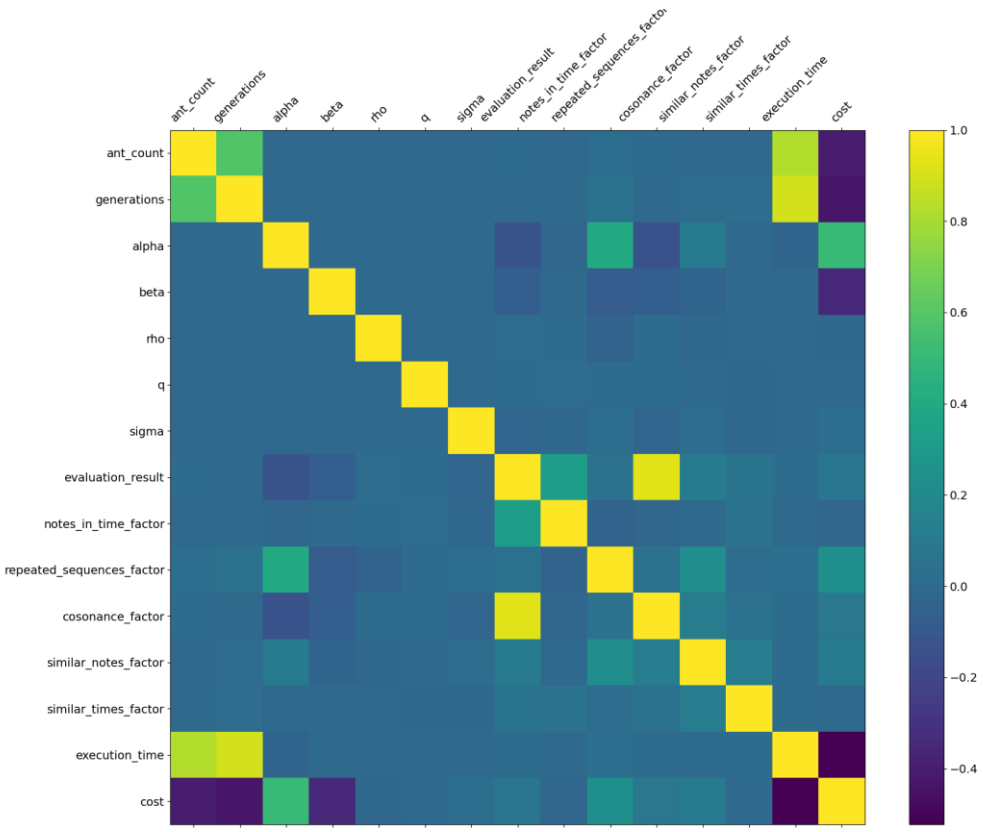
cosonance_factor	evaluation_result	0,987
execution_time	generations	0,737
ant_count	execution_time	0,635
beta	repeated_sequences_factor	0,629
repeated_sequences_factor	cost	-0,517
cost	beta	-0,573
evaluation_result	beta	-0,619
cosonance_factor	beta	-0,674

W teście T1 korelacja nie została policzona dla parametrów *notes_in_time_factor* oraz *similar_times_factor* ze względu na charakter testu, który powoduje, że wartości tych parametrów są zawsze takie same.

Tabela 8. Największe i najmniejsze wartości parametrów w teście T1

Nazwa parametru	**Najmniejsza wartość**	**Id testu min**	**Największa wartość**	**Id testu max**
evaluation_result	0,488	55	0,569	1170
repeated_sequences_factor	0,0001	16	0,056	3115
cosonance_factor	0,558	55	0,813	366
similar_notes_factor	0,131	3087	0,385	278
execution_time	0,005	138	9,714	4371
cost	118	2779	795,6	188

2. Analiza testu T2



Rysunek 40. Macierz korelacji dla testu T2

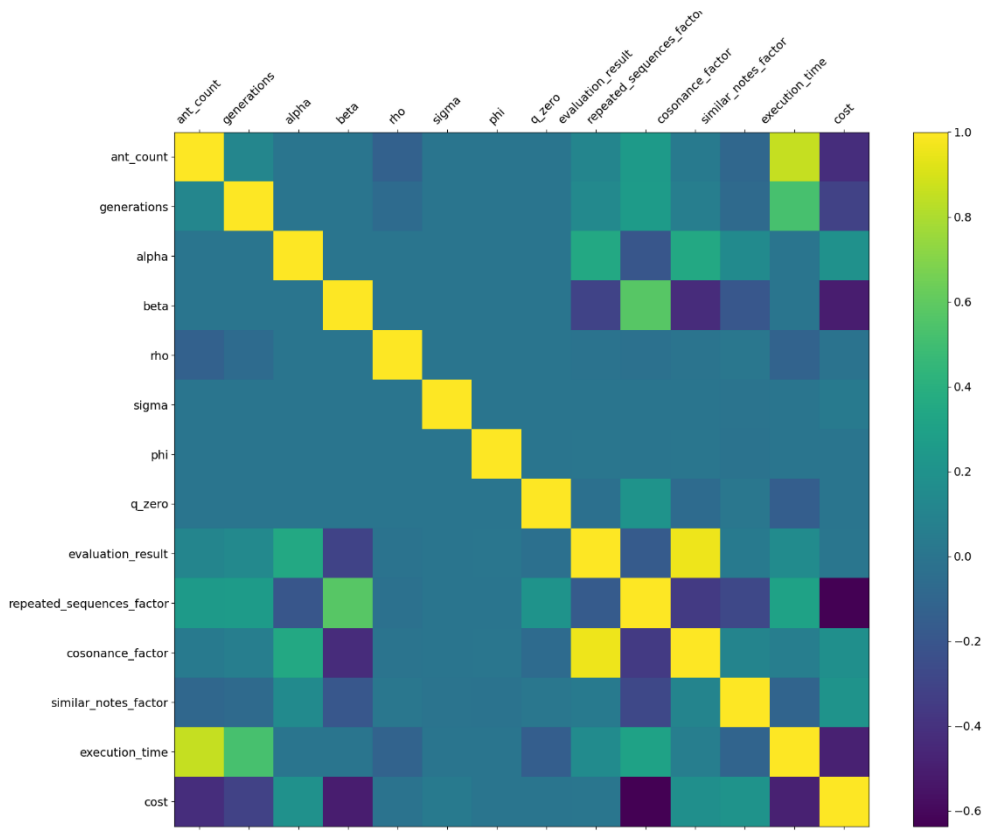
Tabela 9. Tabela wybranych wartości korelacji dla testu T2

cosonance_factor	evaluation_result	0,929
execution_time	generations	0,895
execution_time	ant_count	0,826
ant_count	generations	0,586
cost	alpha	0,508
ant_count	cost	-0,407
cost	generations	-0,429
execution_time	cost	-0,522

Tabela 10. Największe i najmniejsze wartości parametrów w teście T2

Nazwa parametru	**Najmniejsza wartość**	**Id testu min**	**Największa wartość**	**Id testu max**
evaluation_result	0,446	290	0,540	780
notes_in_time_factor	0,701	27	0,809	2791
repeated_sequences_factor	0,0001	1105	0,004	4267
cosonance_factor	0,622	302	0,867	3782
similar_notes_factor	0,178	4723	0,376	928
similar_times_factor	0	1	0,005	257
execution_time	0,002	77	3,403	5551
cost	102,9	4472	759,0	183

3. Analiza testu T3



Rysunek 41. Macierz korelacji dla testu T3

W teście T3 wartość parametru *q* była zawsze stała i równa 1, dlatego w macierzy korelacji brakuje wartości dla tej zmiennej. Zdecydowano się na taki zabieg, aby zmniejszyć liczbę testów do wykonania a wartość *q=1* przyjęto za referencyjną.

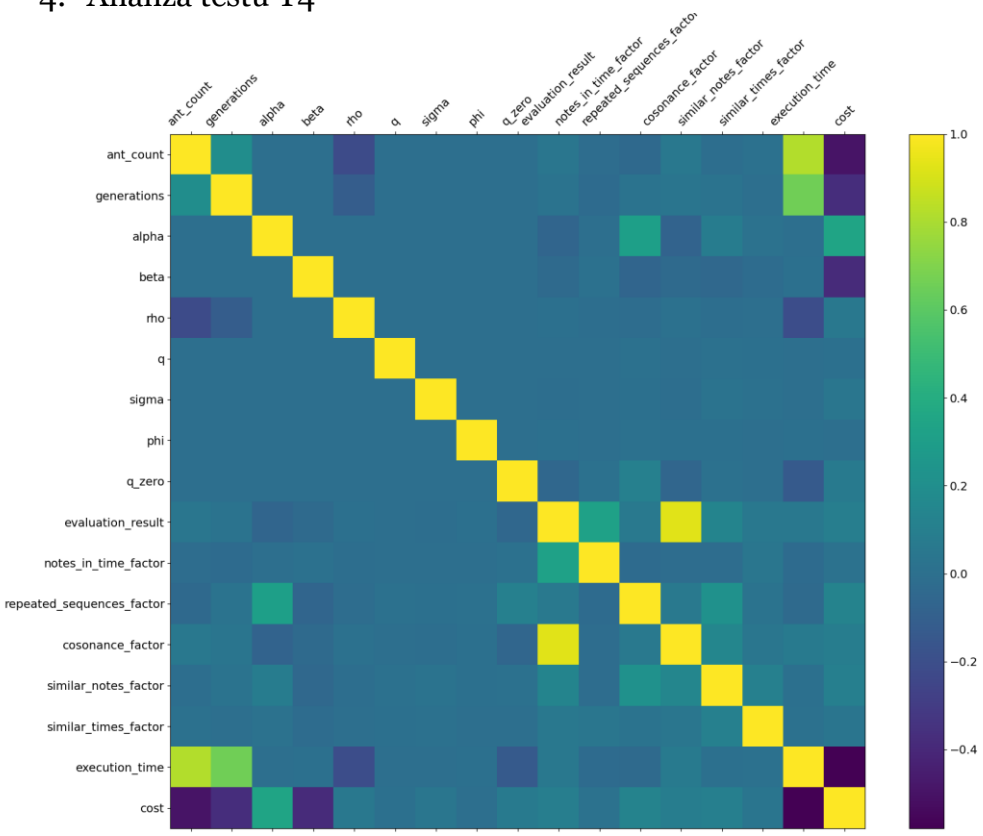
Tabela 11. Tabela wybranych wartości korelacji dla testu T3

cosonance_factor	evaluation_result	0,956
ant_count	execution_time	0,857
repeated_sequences_factor	beta	0,576
generations	execution_time	0,525
cosonance_factor	beta	-0,426
execution_time	cost	-0,488
cost	beta	-0,506
cost	repeated_sequences_factor	-0,637

Tabela 12. Największe i najmniejsze wartości parametrów w teście T3

Nazwa parametru	Najmniejsza wartość	Id testu min	Największa wartość	Id testu max
evaluation_result	0,489	1026	0,560	1638
repeated_sequences_factor	0,0002	12	0,053	3938
cosonance_factor	0,578	68	0,782	1638
similar_notes_factor	0,150	4118	0,374	2874
execution_time	0,008	299	7,717	4185
cost	118	1486	786,5	2

4. Analiza testu T4



Rysunek 42. Macierz korelacji dla testu T4

Tabela 13. Tabela wybranych wartości korelacji dla testu T4

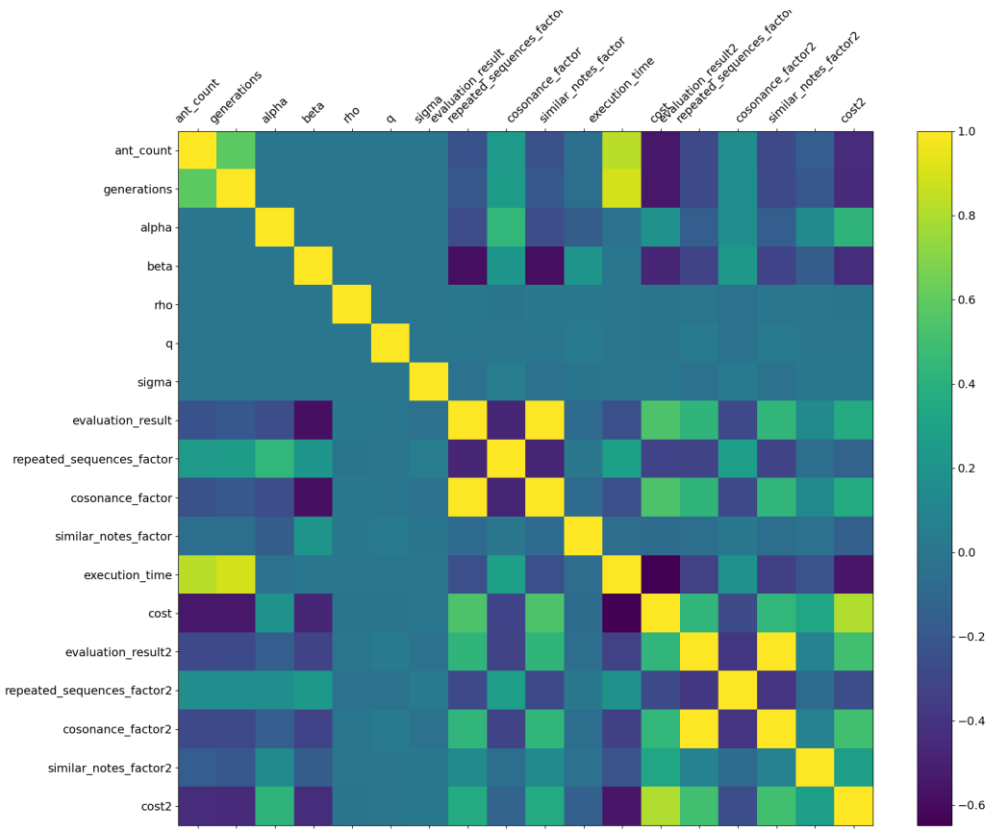
cosonance_factor	evaluation_result	0,928
execution_time	ant_count	0,816
execution_time	generations	0,656
cost	alpha	0,344
cost	ant_count	-0,493
execution_time	cost	-0,579

Tabela 14. Największe i najmniejsze wartości parametrów w teście T4

Nazwa parametru	**Najmniejsza wartość**	**Id testu min**	**Największa wartość**	**Id testu max**
evaluation_result	0,441	8079	0,538	6607
notes_in_time_factor	0,698	3544	0,801	905
repeated_sequences_factor	0	12	0,004	1835
cosonance_factor	0,589	8079	0,871	6607
similar_notes_factor	0,176	703	0,371	281
similar_times_factor	0	5	0,007	924
execution_time	0,002	518	2,771	9054
cost	105,2	7943	787,8	457

5. Analiza testu T5

W tym teście generowane były dwie melodie, dlatego macierz oraz tabele zawierają podwojone parametry wynikowe.



Rysunek 43. Macierz korelacji dla testu T5

Tabela 15. Tabela wybranych wartości korelacji dla testu T5

evaluation_result	cosonance_factor	0,999
evaluation_result2	cosonance_factor2	0,999
execution_time	generations	0,892
ant_count	execution_time	0,831
cost	cost2	0,802
ant_count	generations	0,586
evaluation_result	cost	0,538
cost2	cosonance_factor2	0,501
cost2	alpha	0,424
beta	cost2	-0,435
cost	beta	-0,470
cosonance_factor	beta	-0,583
evaluation_result	Beta	-0,583
cost	execution_time	-0,649

Tabela 16. Największe i najmniejsze wartości parametrów w teście T5

Nazwa parametru	**Najmniejsza wartość**	**Id testu min**	**Największa wartość**	**Id testu max**
evaluation_result	0,453	1689	0,594	96
evaluation_result2	0,476	2874	0,601	5
repeated_sequences_factor	0	8	0,005	2662
repeated_sequences_factor2	0	31	0,010	4823
cosonance_factor	0,369	1685	0,801	96
cosonance_factor2	0,439	2874	0,820	5
similar_notes_factor	0,325	1990	0,454	1883
similar_notes_factor2	0,144	4855	0,322	5322
execution_time	0,004	200	5,841	5441
cost	100,5	4723	512,4	7
cost2	68,9	1947	465,0	279

Ostatnie przedstawione zostaną połączone najlepsze wyniki testów T1 do T5 w kategoriach parametrów wyjściowych. Jako że kategorii tych jest aż 16, zostaną tutaj przedstawione wyniki dla 3, według autora najciekawszych kategorii.

- Evaluation_result_max

Tabela 17. Największe wartości funkcji ewaluacji

	evaluation	**Ants**	**gens**	**α**	**β**	**ρ**	**q**	**φ**	**q0**	**σ**
T1	0,57	5	1	0,5	0,1	0,2	1	-	-	1
T2	0,491	1	20	0,1	5	0,5	1	-	-	1
T3	0,56	5	20	0,1	0,1	0,5	1	0,2	0,2	5
T4	0,538	20	10	0,1	5	0,5	5	0,2	0,5	5
T5	0,594	1	1	0,5	0,1	0,5	1	-	-	1
T5.2	0,601	1	1	0,1	0,1	0,2	5	-	-	10

- Cost_min – najmniejsza suma różnic pomiędzy kolejnymi dźwiękami w melodii.

Tabela 18. najmniejsze wartości kosztu przejścia melodii

	cost	Ants	gens	α	β	ρ	q	φ	q0	σ
T1	118	10	10	2	10	0,5	1	-	-	5
T2	102,9	1	20	0,1	5	0,5	1	-	-	10
T3	118	5	10	1,0	10	0,2	1	0,2	0,5	20
T4	105,2	30	10	0,1	5	0,2	5	0,5	0,9	5
T5	100,5	30	50	0,1	2	0,5	1	-	-	5
T5.2	68,9	5	20	0,5	5	0,2	5	-	-	1

- Execution_time_max – najdłuższy czas wykonania algorytmu

Tabela 19. Przypadki najdłuższego czasu wykonania algorytmu

	execution_time	Ants	gens	α	β	ρ	q	φ	q0	σ
T1	9,715	60	50	5	0,1	0,2	1	-	-	10
T2	3,403	60	50	0,5	5	0,5	1	-	-	5
T3	7,717	60	50	5	0,1	0,2	1	0,2	0,2	10
T4	2,772	60	50	5	10	0,2	5	0,2	0,2	5
T5	5,841	60	50	0,1	2	0,2	5	-	-	10

6.4 Analiza i dyskusja wyników

Analizując wykonane w poprzednim podrozdziale testy, należy zwrócić uwagę na różnicę występujące w macierzach korelacji, które utworzono dla każdego z testów. Nie trzeba wykonywać dogłębnej analizy, by stwierdzić, że macierze testów T2 oraz T4 zawierają znacznie mniej znaczących korelacji pomiędzy parametrami. Wynika to zastosowanej konfiguracji systemowej K2, która zakłada podział sekwencji dźwięków na części, ich losowe ustawienie i nadanie nowych czasów trwania dźwięków. W przypadku tych testów uzyskiwane rezultaty są bardziej nieprzewidywalne i mniej zależne od ustawienia parametrów wejściowych systemu. W przypadku testu T2 można zauważyć niewielką zależność kosztu uzyskanej melodii od parametru α. Oznacza to, że dla wyższych wartości parametru α, który faworyzuje ślad feromonowy, uzyskane melodię posiadają większe interwały między dźwiękami. Może to także oznaczać w

niektórych konfiguracjach wyższy stopień podobieństwa do utworu wejściowego. W przypadku testu T4 zależność kosztu od α jest jeszcze niższa.

Test T1 został wykonany w podstawowej konfiguracji systemu i w tym przypadku widać więcej zależności między parametrami. Znaczący jest tutaj parametr β, sterujący ważnością informacji heurystycznej, który gdy jest zwiększany, powoduje zazwyczaj wzrost liczby powtarzalnych sekwencji dźwięków w wygenerowanej melodii. Zgodnie z oczekiwaniami koszt się zmniejsza, gdy β rośnie, gdyż mrówki wolą wtedy wybierać dźwięki mniej oddalone od siebie. Z kolei wyższe wartości β nie wpływają pozytywnie na zawartość konsonansów w melodii wyjściowej. Opisane zależności parametru β widać również w przypadku testów T3 oraz T5. Pomimo że test T5 wykonywany był w konfiguracji K3, która zakłada podział ścieżek, to parametr β ma tu podobny wpływ na omówione aspekty. Może to wynikać z tego, że w T5 pocięte fragmenty zachowywane są finalnie w oryginalnej kolejności, a czasy trwania nut nie zmieniają się w stosunku do utworu wejściowego.

Jeśli chodzi o czas wykonywania algorytmu to zgodnie z oczekiwaniami, jest on mocno skorelowany z liczbą mrówek i liczbą iteracji. Zależność tę można zauważyć w przypadku wszystkich testów. Czas obliczeń powiązany jest też często ujemnie z kosztem uzyskanego rozwiązania, czyli im dłużej trwają obliczenia, tym większa jest szansa, że interwały między kolejnymi dźwiękami będą mniejsze. Z tabeli 19 wynika, że przy dużej liczbie mrówek i iteracji algorytm AS w konfiguracji K1 jest zdecydowanie najwolniejszy. Podział tego samego problemu na 4 części i przetworzenie ich osobno powoduje zazwyczaj prawie 3-krotne przyśpieszenie obliczeń. Zaobserwowano również, że algorytm ACS jest szybszy od algorytmu AS. Wynika to z tego, że eksploatacja ma mniejszą złożoność obliczeniową, czyli jest szybsza od eksploracji. Występuje pewna niewielka korelacja pomiędzy parametrem q0 a czasem wykonania algorytmu ACS. Najszybciej uzyskiwane były wyniki dla testu T4, który operuje algorytmem ACS w trybie podziału sekwencji dźwięków.

Z tabeli 17 wynika, że nawet przy niewielkiej liczbie mrówek i iteracji można uzyskać wysoką wartość funkcji oceny. Funkcja ta z kolei opiera się na trzech kryteriach, za pomocą których tak naprawdę trudno określić całościową jakość muzyki, gdyż jakość zależy także od czynników, które nie są mierzalne, a sama jakość melodii powinna być także rozpatrywana w kontekście konkretnego gatunku. Jeśli chodzi o wyniki dla funkcji ewaluacji, to należy zaznaczyć, że testy, które nie modyfikują czasu trwania dźwięków (T1, T3, T5), będą miały

maksymalną wartość funkcji oceny *notes_in_time*, natomiast generowanie nowych wartości rytmicznych może spowodować obniżenie wartości tego kryterium oceny. Wynika to ze specyfikacji systemu, który nie został zaprojektowany pod kątem generowania przejścia między dźwiękami na podstawie wartości rytmicznych. Zastosowanie algorytmów mrowiskowych w tej pracy nie sprowadza się do znalezienia najkrótszej ścieżki w grafie, tak jak to jest w problemie TSP. Dlatego nawet jedna wirtualna mrówka w kilku iteracjach może wygenerować przyjemnie brzmiącą melodię, dzięki odpowiedniemu dobraniu pozostałych parametrów. Istotny jest bowiem odpowiedni balans pomiędzy sugerowaniem się melodią wejściową a wybieraniem korzystnych przejść między dźwiękami z heurystycznego punktu widzenia. Pożądany efekt uzyskać można uzyskać, manipulując przede wszystkim parametrami α oraz β. Większe α spowoduje mocniejsze sugerowanie się oryginalną melodią natomiast gdy β jest większa od α, mrówki zaczną mocniej odbiegać od melodii wejściowej i improwizować zgodnie z heurystyką. Pozostałe parametry mają mniejszy wpływ na styl uzyskiwanej melodii. Należy jednak zachować odpowiednie proporcje między przykładowo wartością *q* a σ, by zainicjowany początkowo feromon miał istotne znaczenie. Na podstawie badań i doświadczeń określono podstawowy zestaw parametrów, który daje dobre rezultaty i można go traktować jako punkt wyjścia.

Tabela 20. Referencyjne wartości parametrów dla aplikacji AntsBand

Liczba mrówek	**Liczba iteracji**	**α**	**β**	**ρ**	**q**	**φ**	**q0**	**σ**
10	10	1	2	0.3	1	0,2	0,5	10

7 Podsumowanie i wnioski końcowe

Celem niniejszej pracy było uzyskanie zadowalających efektów dźwiękowych poprzez zaadaptowane do zadania tworzenia muzyki algorytmy mrowiskowe. Powstała w ramach pracy system AntsBand pokazuje, że cel ten udało się osiągnąć dzięki możliwości generowania nowych melodii, granych na różnych instrumentach oraz harmonijnie brzmiących. Aplikacja ta ponadto posiada graficzny interfejs użytkownika, możliwość wczytywania dowolnych plików midi oraz rozbudowany system ustawień, co czyni ją gotowym narzędziem do generowania melodii, przeznaczonym dla zewnętrznego użytkownika. Przy odpowiednio dobranych ustawieniach, system jest w stanie dawać zadowalające rezultaty muzyczne, generując poprawnie i przyjemnie brzmiące melodię.

W ramach badań na zaimplementowanym systemie algorytm wykonano 291780 razy, z wieloma zestawami parametrów wejściowych i w różnych konfiguracjach systemu. Na podstawie uzyskanych w ten sposób danych wynikowych udało się dzięki obliczeniu korelacji poznać znaczenie i zależności pomiędzy poszczególnymi parametrami systemu. Wyłoniono również testy, dla których uzyskane zostały najlepsze lub najgorsze rezultaty w kontekście danego atrybutu. Poprzez analizę uzyskanych wyników udało się ustalić, że zwiększona wartość parametru α dodatnio wpływa na wielkość interwałów w melodii wyjściowej oraz na podobieństwo melodii wyjściowej do wejściowej. Bardzo istotny okazał się parametr β, który powiększony powoduje wzrost liczby powtarzanych sekwencji dźwięków, mniejsze wartości interwałów pomiędzy kolejnymi dźwiękami, ale za to mniejszą zawartość konsonansów w melodii wyjściowej. Badania pokazują również, że tryb podziału melodii na części przyśpiesza wykonanie algorytmu, spada natomiast wartość korelacji pomiędzy parametrami, jeśli części są losowo ustawiane. Wykazano, że algorytm ACS jest szybszy od algorytmu AS oraz, że liczba mrówek i iteracji znacznie spowalnia wykonanie algorytmu. Zważywszy na cel adaptacji systemu mrowiskowego do zadania komponowania muzyki, w którym nie jest istotne uzyskanie najkrótszej ścieżki w grafie, nie ma większego sensu ustawianie liczby mrówek i iteracji na wartości większe niż 10. Najistotniejsze jest uzyskanie odpowiedniego balansu pomiędzy sugerowaniem się melodią wejściową a heurystycznym wyborem przejść, co można uzyskać, odpowiednio ustawiając parametry α oraz β.

Wiedza uzyskana z analizy wyników badań umożliwiła dokonanie ustawień parametrów systemu tak, by uzyskać konkretne, przewidywalne rezultaty.

Uwidoczniły się także wady i aspekty systemu, które można by w przyszłości poprawić. Z pewnością należałoby rozbudować strukturę grafu o obsługę akordów oraz czasów trwania dźwięków, co wiązałoby się ze znaczną modyfikacją funkcji obliczającej jakość informacji heurystycznej. Również koszt przejścia pomiędzy węzłami grafu powinien uwzględniać więcej czynników w tym analizować zasady harmonii. Warto by również wprowadzić interakcje pomiędzy instrumentami na etapie generowania melodii według zasad harmonii. Obsługa instrumentów perkusyjnych i zwiększenie liczby kryteriów oceny to kolejne elementy do potencjalnego rozwoju.

Cały proces pisania pracy był dla autora bardzo cennym doświadczeniem. Udało się zgłębić zasady działania i sposób pracy z algorytmami mrowiskowymi, które są fascynującą gałęzią algorytmiki inspirowanej naturą. Zdecydowanie wartościowe są również doświadczenia z programowania w języku Python oraz operowania komunikatami midi w kodzie. Małą barierą w rozwoju pracy były braki w wiedzy z zakresu teorii muzyki, które częściowo udało się uzupełnić, natomiast wykonanie opisanych wcześniej elementów rozwojowych wymaga dalszego zgłębiania owej wiedzy. Niemniej jednak według autora muzyka jest najważniejszą z wszystkich stworzonych przez człowieka dziedzin sztuk pięknych. Trudnym zadaniem jest opisanie jej za pomocą funkcji matematycznych, a tym bardziej dokonanie matematycznej oceny, co nie oznacza, że jest to zadanie niemożliwe, czego dowodem są osiągnięcia innych w tej dziedzinie (rozdział 3.3) oraz niniejsza praca.

Muzyka w dzisiejszych czasach coraz częściej komponowana jest w środowiskach cyfrowych. Jest to potencjalne zastosowanie systemu AntsBand, który można by rozwinąć i połączyć z cyfrowym syntezatorem midi w celu stworzenia zaawansowanego narzędzia do tworzenia muzyki elektronicznej, wspomagającego przy tym kreatywność artysty. Również interesujące wydaje się zastosowanie systemu do generowania długich, niepowtarzających się utworów na potrzeby np. gier komputerowych. Niniejsza praca dyplomowa jest dla mnie dużą inspiracją do dalszego badania i rozwoju algorytmów mrowiskowych jako narzędzia do komponowania muzyki, a moim marzeniem jest stworzenie syntezatora współpracującego z ulepszonym systemem AntsBand.

8 Bibliografia

1. M. Dorigo, T. Stützle, Ant Colony Optimization. Bradford Company, Scituate. MA, USA, 2004
2. Michael Pilhofer, Holly Day, Teoria muzyki dla bystrzaków. Wydanie II, Septem 2014
3. Chun-Yien Chang, Ying-Ping Chen, AntsOMG: A Framework Aiming to Automate Creativity and Intelligent Behavior with a Showcase on Cantus Firmus Composition and Style Development.
4. Michael Geis, Martin Middendorf, An Ant Colony Optimizer for Melody Creation with Baroque Harmony, 2007
5. MIDI Manufacturers Association, The Complete MIDI 1.0 Detailed Specification, 2014
6. J.-L. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. Journal of insect behavior, 1990.
7. S. Goss, S. Aron, J.-L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. Naturwissenschaften, 1989.
8. Vittorio Maniezzo, Luca Maria Gambardella, Fabio de Luigi, Ant Colony Optimization, 2004.
9. M. Dorigo, V. Maniezzo and A. Colorni, The Ant System: An autocatalytic optimizing process. Technical Report 91-016 Revised, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1991
10. M. Dorigo, T. Stützle, The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances, 2006
11. Maad M. Mijwil, History of Artificial Intelligence, 2015
12. C. Guéret, N. Monmarché, and M. Slimane, Ants can play music, 2004
13. Dokumentacja języka Python, https://docs.python.org/3/
14. Dokumentacja biblioteki acopy, https://acopy.readthedocs.io/en/latest/
15. Dokumentacja biblioteki mido, https://mido.readthedocs.io/en/latest/
16. Dokumentacja biblioteki pygame
17. Marco Dorigo, Vittorio Maniezzo, Alberto Colorni, The Ant System: Optimization by a colony of cooperating agents, 1996
18. Marco Dorigo, Luca Maria Gambardella, Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem, 1997
19. Aleem Akhtar, Evolution of Ant Colony Optimization Algorithm — A Brief Literature Review, SEECS-NUST, ISLAMABAD, 2019

20. Marcin Wściubiak, Ewolucyjna optymalizacja wielokryterialna (2000)
21. Thomas Stützle, Marco Dorigo, ACO Algorithms for the Traveling Salesman Problem, 1999.
22. Dokumentacja biblioteki joblib, https://joblib.readthedocs.io
23. Dokumentacja biblioteki pandas, https://pandas.pydata.org/docs/
24. Jolanta Kurkiewicz, Marcin Stonawski, Podstawy statystyki, Kraków 2005
25. Zbigniew Czech, wprowadzenie do obliczeń równoległych, Warszawa 2010
26. Ali Taylan Cemgil, Peter Desainy, Bert Kappen, Rhythm Quantization for Transcription, University of Nijmegen, 1999
27. Dokumentacja biblioteki tkinter, https://tkdocs.com/index.html
28. S. Haridi, P. Van Roy: Programowanie. Koncepcje, techniki i modele. Helion, 2005.
29. Charles R. Severance, Python dla wszystkich: Odkrywanie danych z Python 3, Andrzej Wójtowicz, 2021
30. H. Eschenauer, J. Koski, and A. Osyczka. Multicriteria design optimization, 1990
31. Witryna system AIVA, https://www.aiva.ai/
32. Dokumentacja biblioteki matplotlib.
33. R.C. Martin, Czysty kod. Podręcznik dobrego programisty, Helion 2014
34. Krzysztof Guczalski, Harmonia nie tkwi w liczbach. O pitagorejczykach, strojach i zgodnych współbrzmieniach, 2015
35. Dokumentacja narzędzia pyinstaller
36. Nathan Lepora, Paul F.M.J. Verschure, Tony J Prescott, The state of the art in biomimetics, Bioinspiration & Biomimetics 2013
37. Robin J. Wilson, Wprowadzenie do teorii grafów, PWN 2007
38. Simon Colton, Ramon López de Mántaras, Oliviero Stock, Computational Creativity: Coming of Age, Ai Magazine, 2009
39. L. Hiller, L. Isaacson, Musical composition with a high-speed digital computer, 1958.
40. G. M. Rader, A method for composing simple traditional music by computer, 1974.
41. J. Bharucha, MUSACT: A connectionist model of musical harmony, 1993
42. Lech Banachowski, Krzysztof Diks, Wojciech Rytter, Algorytmy i struktury danych, Wyd. nowe poprawione PWN. – Warszawa, 2018

9 Spis rysunków

10 Spis tabel