POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI
Instytut Informatyki

Zaawansowane programowanie

ROZWIĄZANIE PROBLEMU CZĘŚCIOWEGO TRAWIENIA (MAX SUBSET PDP) METAHEURYSTYKĄ TABU SEARCH

Adam Łangowski, 147896

POZNAŃ 2024

Spis treści

Rozdział 1

Wstęp

1.1 Wprowadzenie

W dziedzinie bioinformatyki, rozwiązywanie problemów o charakterze kombinatorycznym często okazuje się wysoce czasochłonne przy zastosowaniu tradycyjnych algorytmów, które dążą do znalezienia dokładnych rozwiązań. Stosowanie algorytmów przybliżonych wynika z konieczności skutecznego radzenia sobie ze złożonością obliczeniową, charakterystyczną dla problemów biologicznych o dużej skali, takich jak sekwencjonowanie przez hybrydyzację czy mapowanie restrykcyjne. Algorytmy metaheurystyczne oferują efektywne narzędzie do przeszukiwania przestrzeni rozwiązań danego problemu, umożliwiając jednocześnie skrócenie czasu obliczeń względem algorytmów dokładnych. W niniejszej pracy przedstawiono implementację algorytmu Tabu Search wraz z generatorem instancji dla problemu częściowego trawienia. Celem projektu było ponadto opracowanie aplikacji wizualnej, implementującej wybrany algorytm oraz przeprowadzenie wyczerpujących eksperymentów obliczeniowych, mających na celu zbadanie skuteczności wybranej metaheurystyki w rozwiązaniu analizowanego problemu.

1.2 Zakres pracy

Struktura pracy jest następująca. W rozdziale 2. przedstawiono bliżej podstawy teoretyczne dotyczące mapowania metodą częściowego trawienia oraz bazującą na niej formalną definicję problemu kombinatorycznego *Max Subset PDP*. Przybliżono również zagadnienie złożoności obliczeniowej w danym problemie oraz szczegółowy opis metaheurystyki Tabu Search. Rozdział 3. poświęcony został opisowi powstałej aplikacji. Omówiona została jej architektura, skupiając się na interfejsie użytkownika oraz implementacji niezbędnej wielowątkowości. Następnie przedstawiono stworzony generator instancji dla problemu oraz zastosowane modyfikacje i zasady działania stworzonej metaheurystyki. Rozdział 4. zawiera szczegółowy przegląd przeprowadzonych testów programu oraz zbiorcze wyniki. Wyszczególniono w tym miejscu wszystkie testowane instancje, omówiono czas i jakość uzyskanych rozwiązań oraz najważniejsze aspekty metaheurystyki z perspektywy rezultatów. Rozdział 5. stanowi podsumowanie pracy, gdzie oceniono efektywność metody Tabu Search w rozwiązywaniu problemu *Max Subset PDP* oraz wnioski.

Rozdział 2

Podstawy teoretyczne

2.1 Mapowanie metodą częściowego trawienia

Metoda częściowego trawienia DNA, w jednej ze swoich wersji, opiera się na sekwencyjnym działaniu jednego enzymu restrykcyjnego na kopie badanego fragmentu DNA w zróżnicowanych przedziałach czasowych. Enzymy restrykcyjne rozpoznają specyficzne dla siebie miejsca w dwuniciowym DNA i przecinają łańcuch (trawią) w rozpoznanym miejscu. W tym eksperymencie czas trawienia enzymem, zgodnie z teoretycznym modelem, jest stopniowo dostosowywany w kolejnych reakcjach, tak aby enzym przecinał fragment DNA w kolejnych miejscach restrykcyjnych obecnych w danym fragmencie. Długości uzyskanych tą drogą fragmentów mierzone są za pomocą elektroforezy żelowej. W efekcie uzyskuje się multizbiór A, który reprezentuje długości odcinków uzyskanych we wszystkich przeprowadzonych reakcjach, wraz z pełną długością badanego fragmentu. Warto zauważyć także, że w przeciwieństwie do mapowania przez hybrydyzację, nie posiadamy informacji o nakładaniu się fragmentów względem siebie.

Następnie celem jest odtworzenie mapy miejsc restrykcyjnych w analizowanym fragmencie DNA w oparciu o uzyskany multizbiór. Konieczne jest dokładne określenie rozmieszczenia punktów cięcia, tak aby odległości między wszystkimi parami cięć, jak również końcami fragmentu DNA, były zgodne z uzyskanym multizbiorem A. Ta strategia umożliwia w pełni odzwierciedlenie struktury badanego fragmentu DNA na podstawie eksperymentalnych danych dotyczących częściowego trawienia enzymem restrykcyjnym. Zależność liczby cięć k od liczby elementów w A, zakładając idealnie przeprowadzony proces trawienia, można opisać wzorem: $|A| = \binom{k+2}{2}$.

Problem przy założeniu braku błędów w instancji jest otwarty z punktu widzenia złożoności obliczeniowej. Natomiast zakładając występowanie błędów w danych wejściowych staje się on trudny. Mapowanie restrykcyjne wiąże się najczęściej z błędnym pomiarem długości fragmentów, ponieważ elektroforeza dokonuje pomiaru z pewną określoną dokładnością oraz mogą wystąpić błędy negatywne, czyli braki niektórych elementów w wynikowych multizbiorach wartości. Wprowadzając do instancji błędy takie jak substytucja, delecja czy insercja, wybranych wartości w multizbiorze A, klasyczny problem mapowania metodą częściowego trawienia przyjmuje tym samym postać zadania optymalizacyjnego.

2.2 Definicja problemu - Max Subset Partial Digest Problem

Problem sformułowany jest następująco:
Instancja: Multizbiór liczb całkowitych dodatnich $D = \{d_1, \ldots, d_k\}$.
Rozwiązanie: Zbiór liczb całkowitych nieujemnych $P = \{p_1, \ldots, p_m\}$ taki,
że $\{|p_i - p_j| : 1 \leq i < j \leq m\} \subseteq D$ i m jest maksymalne.

D – multizbiór zawierający długości odcinków
k – liczba elementów w multizbiorze D
P – zbiór stanowiący rozwiązanie
m – liczba elementów w zbiorze P

Celem jest zatem odtworzenie mapy, utworzonej na podstawie multizbioru D, której odcinki znajdą się w zbiorze stanowiącym rozwiązanie P, w taki sposób aby zmaksymalizować ilość odcinków w utworzonym rozwiązaniu.

2.3 Złożoność obliczeniowa omawianego problemu

Rozważany problem jest trudny obliczeniowo ze względu na swoją naturę. W kontekście optymalizacji kombinatorycznej, gdzie przeszukiwanie przestrzeni rozwiązań musi uwzględniać wszystkie możliwe kombinacje, problem staje się NP-trudny. To oznacza, że nie istnieje (na dzisiejszy stan wiedzy) algorytm o wielomianowym czasie działania, który skutecznie rozwiązuje ten problem dla dowolnej instancji.

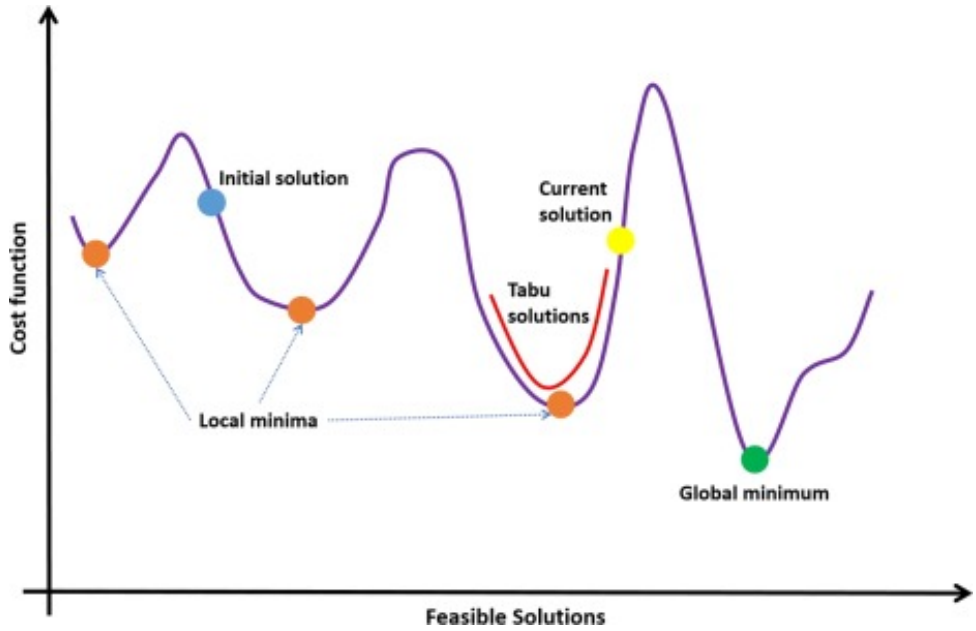
Problem trudności obliczeniowej wynika z konieczności uwzględnienia wszystkich kombinacji długości odcinków z multizbioru D i znalezienia takiego zbioru P, który spełnia warunki maksymalizacji. Złożoność obliczeniowa zależy zatem przede wszystkim od liczby elementów w multizbiorze D. Wraz ze wzrostem rozmiaru instancji problemu, liczba możliwych kombinacji rośnie wykładniczo, co sprawia, że konwencjonalne metody przeszukiwania przestrzeni stanów stają się nieefektywne. Dlatego też problem ten wymaga zastosowania metod optymalizacji, takich jak metaheurystyka tabu, aby skutecznie radzić sobie z trudnościami obliczeniowymi i znaleźć rozwiązanie zbliżone do optymalnego w akceptowalnym czasie.

2.4 Metaheurystyka Tabu Search

Metaheurystyka Tabu Search jest przede wszystkim pewną metodą / ideą rozwiązywania różnych problemów optymalizacyjnych. Jego historia sięga roku 1977 kiedy Fred Glover przedstawił pracę dotyczącą zastosowania w algorytmie pamięci krótkotrwałej i długotrwałej podczas lokalnego przeszukiwania przestrzeni rozwiązań. Jego dzieło było podstawą dla dalszych prac i finalnego powstania heurystyki, w postaci bliskiej do obecnie często stosowanej, w roku 1986. Główną cechą Tabu jest fakt, iż jest to algorytm deterministyczny, w przeciwieństwie chociażby do algorytmu genetycznego. Jak mawiał jego autor: "Zły wybór strategiczny jest lepszy, niż dobry wybór losowy" (bo jest pod kontrolą, więc można ocenić strategię i wyciągnąć odpowiednie wnioski). Wybory deterministyczne są preferowane w tabu search również ze względu na iteracyjnie ulepszane pojedyncze rozwiązanie.

Algorytm Tabu Search opiera się na heurystyce lokalnego przeszukiwania, co oznacza, że zaczyna od pewnego rozwiązania i próbuje je iteracyjnie ulepszać. Jednak, ze względu na charakter heurystyk lokalnych, może utknąć w lokalnych optimum, co obrazuje Rysunek 2.1. Dlatego wprowadza się mechanizmy intensyfikacji, czyli skupiania się na doskonaleniu obecnie badanego obszaru, lecz także dywersyfikacji, czyli poszukiwania nowych obszarów przestrzeni rozwiązań.

Podstawowym pojęciem w Tabu Search jest "sąsiedztwo". Definiuje ono nowe rozwiązanie, które powstaje po zastosowaniu jednego ruchu elementarnego dla aktualnego rozwiązania. Ruchy elementarne mogą obejmować dodawanie elementów, ich usuwanie, substytucję lub zamianę miejscami wybranych elementów. Utworzone sąsiedztwo zawiera w sobie wszystkich rozważanych następnie



Rysunek 2.1: Wizualizacja procesu przeszukiwania przestrzeni rozwiązań metodą Tabu Search [1].

kandydatów, czyli nowe potencjalne rozwiązania. Spośród ocenionych kandydatów wybierany jest ten, który osiągnął najwyższą wartość funkcji celu lub spełnił inne ustalone kryteria. Wybór ten może również uwzględniać zasady listy tabu, bądź kryterium aspiracji. Bardziej wyrafinowane i zazwyczaj skuteczniejsze metody wyboru kandydata opierają się na zaawansowanych algorytmach ewolucyjnych, takich jak na przykład BOA (ang. Bayesian optimization algorithm).

Wprowadzona lista tabu to jedna z kluczowych koncepcji algorytmu. Jest to struktura przechowująca ruchy zabronione na daną liczbę kolejnych iteracji. Implementuje się ją zatem najczęściej w postaci kolejki FIFO. Mechanizm ten wprowadza różnorodność w przeszukiwaniu przestrzeni rozwiązań, pomagając unikać utknięcia w lokalnych optimum. Kryterium aspiracji natomiast pozwala na celowe naruszenie warunków listy tabu. W najprostszej implementacji, jeśli ruch prowadzi do lepszego rozwiązania niż dotychczasowe najlepsze, kryterium aspiracji zezwala na wykonanie takiego ruchu. Inne, znacznie bardziej skomplikowane kryteria aspiracji opierają się na wyspecjalizowanych metodach badających możliwości powstania cyklu po wykonaniu danego ruchu.

Algorytm Tabu Search kończy swoje działanie na podstawie określonego warunku stopu. Może to być na przykład określona liczba iteracji, brak poprawy przez pewną liczbę iteracji, osiągnięcie założonego celu lub maksymalny czas działania.

Efektywność algorytmu zależy od wielu czynników, takich jak właściwy dobór parametrów, strategie intensyfikacji i dywersyfikacji, czy odpowiednia długość listy tabu. Wszystkie z tych parametrów powinny zostać wyznaczone dla konkretnego problemu, na drodze licznych testów, strojąc tym samym tworzoną metaheurystykę. Zazwyczaj duże znaczenie ma także jakość rozwiązania początkowego, stąd popularnym podejściem jest stosowanie innych heurystyk i algorytmów do efektywnego wyznaczenia jak najlepszego punktu startu dla metody Tabu.

Rozdział 3

Opis aplikacji

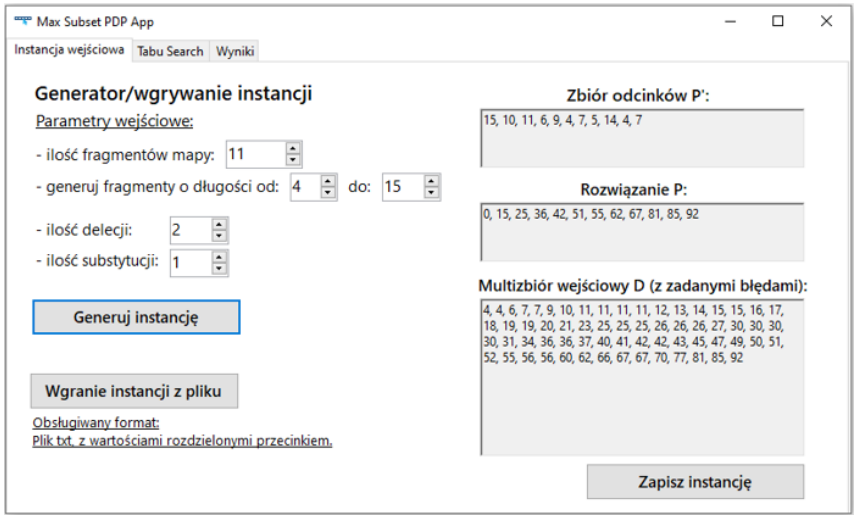
3.1 Architektura aplikacji

Aplikacja została napisana z zastosowaniem Windows Forms oraz .NET w wersji 8.0. Stworzony wielowątkowy program działa w oparciu utworzone klasy `PDPInstanceGenerator` oraz `TabuAlgorithm`, we współpracy z oknem programu - klasą `Form1`. Poniższy rozdział prezentuje działanie każdej składowej omawianej aplikacji.

3.1.1 Interfejs użytkownika

Stworzony interfejs obejmuje 3 zakładki, do których użytkownik ma stały dostęp:

1. Instancja wejściowa



Rysunek 3.1: Widok z pierwszej zakładki.

Rysunek 3.1 przedstawia pełny widok w pierwszej z zakładek, która umożliwia wygenerowanie instancji wejściowej dla problemu lub wczytanie instancji - wejściowego multizbioru - z wybranego pliku o rozszerzeniu .txt. W przypadku generowania instancji, należy ustawić odpowiednie wartości w polach:

a) ilość fragmentów mapy

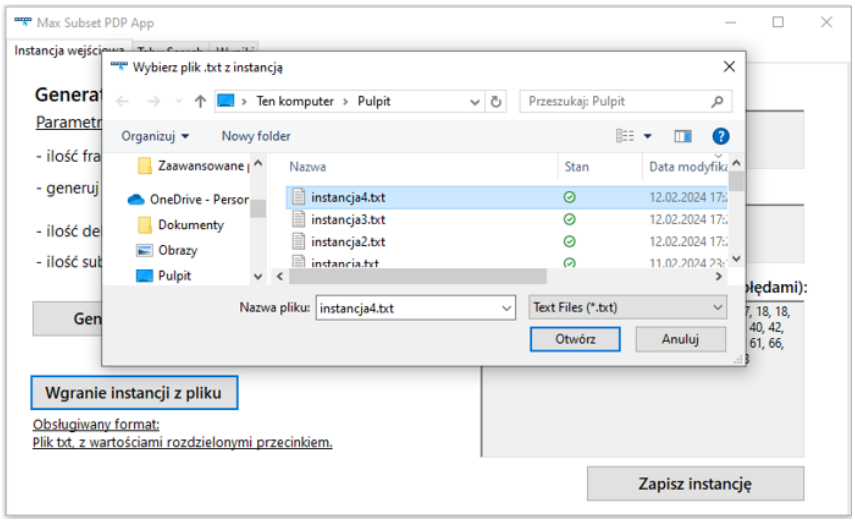
b) generuj fragmenty o długości od / do

c) ilość delecji

d) ilość substytucji

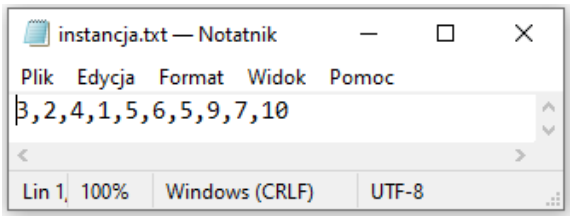
Aplikacja sprawdza poprawność wszystkich wprowadzonych danych. Żadne wartości nie mogą być puste, ujemne ani niecałkowite. Ponadto maksymalna długość elementów mapy musi być nie mniejsza od minimalnej oraz ilość błędów delecji nie może przekraczać liczności utworzonego multizbioru wejściowego. Gdy powyższe warunki będą spełnione użytkownik może kliknąć 'Generuj instancję'. W tym momencie instancja wejściowa problemu jest zapisywana i przekazywana do algorytmu Tabu, którego działanie możemy rozpocząć w drugiej zakładce - 'Tabu Search'. W oknie 'Zbiór odcinków P" wyświetlone zostają wygenerowane losowe odcinki o podanej długości. Poniżej, w oknie 'Rozwiązanie P' widoczne jest rozwiązanie problemu, które wykorzystuje wszystkie odcinki z wcześniejszego zbioru, nie biorąc pod uwagę zadanych w parametrach generatora błędów. Najniżej, w oknie 'Multizbiór wejściowy D (z zadanymi błędami)' uzyskujemy multizbiór wejściowy dla problemu, przesortowany, jego postać jest tożsama z tym, co otrzymuje algorytm Tabu Search na początku działania.

W przypadku wybrania 'Wgranie instancji z pliku' wyświetla się nowe okno, tak jak widać na Rysunku 3.2, z możliwością wyboru pliku z instancją.



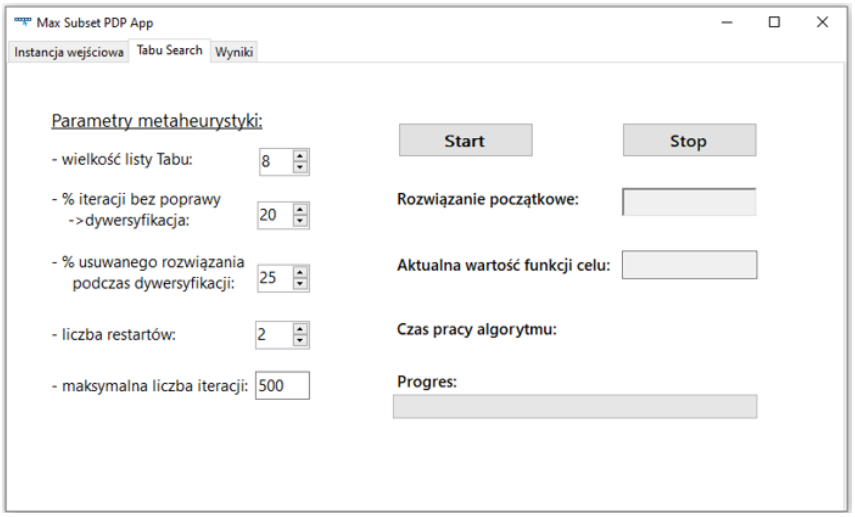
Rysunek 3.2: Wczytywanie instancji z pliku.

Plik powinien mieć odpowiednią strukturę. Przykład odpowiedniej instancji wejściowej widoczny jest na Rysunku 3.3. Ostatnią funkcjonalnością w pierwszej zakładce jest opcja 'Za-



Rysunek 3.3: Przykładowa instancja z pliku.

pisz instancję'. Po jej wybraniu użytkownikowi wyświetla się nowe okno, umożliwiające zapis aktualnego multizbioru wejściowego.



Rysunek 3.4: Widok z drugiej zakładki.

2. Tabu Search

Rysunek 3.4 przedstawia widok z drugiej zakładki aplikacji, w której mamy możliwość uruchomienia algorytmu Tabu, jego zatrzymania i podejrzenia zmieniających się w czasie wybranych statystyk generowanych przez algorytm.
W pierwszej kolejności należy ustawić parametry metaheurystyki:

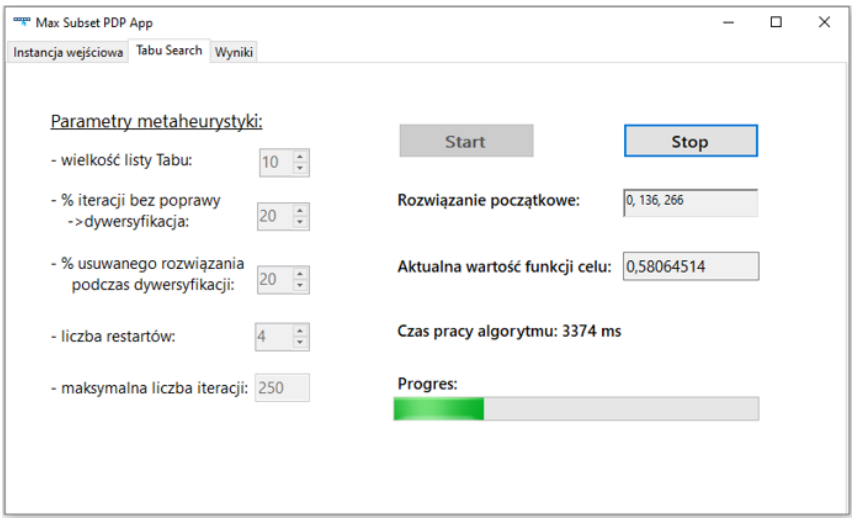
a) wielkość listy Tabu
b) procent iteracji bez poprawy przed dywersyfikacją
c) procent usuwanego rozwiązania podczas dywersyfikacji
d) liczba restartów
e) maksymalna liczba iteracji

Po prawidłowej walidacji wprowadzonych parametrów użytkownik ma możliwość uruchomienia algorytmu Tabu poprzez przycisk "Start". W czasie działania algorytmu zablokowana zostaje możliwość wprowadzenia zmian w parametrach metaheurystyki oraz w instancji wejściowej. Użytkownik ma natomiast możliwość poruszania się po całej aplikacji, z zasługi omówionej, w rozdziale 3.1.2, zaimplementowanej wielowątkowości.

W trakcie działania algorytmu, co ukazuje Rysunek 3.5, użytkownik ma możliwość zobaczyć elementy takie jak:

a) wygenerowane rozwiązanie początkowe
b) aktualna wartość funkcji celu
c) czas pracy algorytmu
d) progres wizualizowany poprzez pasek postępu

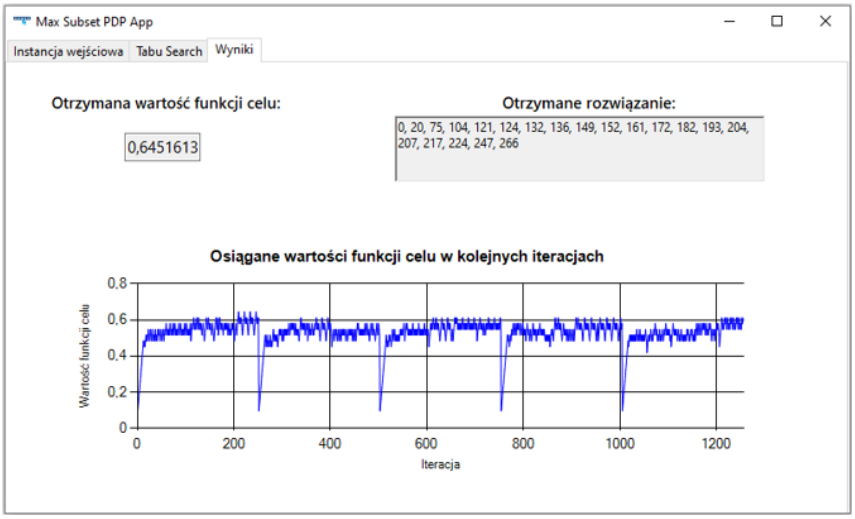
Po zakończeniu pracy algorytmu, co symbolizuje zatrzymany wyświetlany czas oraz wypełniony pasek postępu, wyniki pracy algorytmu są przesyłane do trzeciej zakładki - 'Wyniki'. Użytkownik ma również możliwość zatrzymania w dowolnym momencie pracy algorytmu Tabu poprzez naciśnięcie przycisku 'Stop'. Takie działania skutkuje zapisem aktualnego stanu algorytmu i uzyskanych wyników oraz przesłanie tych informacji do zakładki 'Wyniki'.



Rysunek 3.5: Wizualizacja pracy algorytmu Tabu.

3. Wyniki

   W ostatniej zakładce aplikacji użytkownikowi wyświetlone zostają wyniki działania algorytmu Tabu Search. Rysunek 3.6 pokazuje przykładowy wynik działania programu, obrazując uzyskane rozwiązanie, otrzymaną maksymalną wartość funkcji celu oraz wykres przedstawiający zmieniającą się obliczaną funkcję celu w każdej iteracji metaheurystyki.



Rysunek 3.6: Przykładowy wynik algorytmu Tabu Search.

3.1.2 Wielowątkowość

Klasy `Form1` oraz `TabuAlgorithm`, zaimplementowane w aplikacji, współpracują w celu wykonania algorytmu Tabu Search dla określonego problemu i wizualizacji wyników w czasie rzeczywistym. Wykorzystanie wielowątkowości w tych klasach obejmuje użycie klasy `BackgroundWorker` do obsługi zadań w tle na osobnym wątku w klasie `Form1`. W konstruktorze klasy `Form1` inicjalizowany jest obiekt `BackgroundWorker` o nazwie tabuWorker. Obiekt ten służy do wykonywania operacji na osobnym, dedykowanym wątku. Po uruchomieniu algorytmu Tabu utworzony zostaje nowy obiekt `BackgroundWorker`, a metoda *tabuWorkerDoWork* jest przypisana jako metoda do uruchomienia

algorytmu na osobnym wątku za pomocą *RunWorkerAsync*.

Klasa `TabuAlgorithm` otrzymuje instancję `Form1` w swoim konstruktorze, co pozwala na interakcję z interfejsem użytkownika. Metoda *SearchSolutionSpace*, która wykonuje algorytm Tabu, jest uruchamiana w ramach metody *tabuWorkerDoWork* klasy `Form1`, dzięki czemu działa na osobnym wątku w tle, a użytkownik dalej może sterować aplikacją.

Metoda `UpdateUI` w klasie `Form1` jest używana do aktualizacji elementów interfejsu użytkownika, w celu zapewnienia bezpieczeństwa wątków. Umożliwia ona wyświetlanie wyników i i postępu algorytmu w klasie `TabuAlgorithm`. Metoda `UpdateUI` sprawdza, czy aktualne wywołanie metody pochodzi z innego wątku niż ten, na którym została stworzona kontrolka interfejsu użytkownika. Jeśli tak, używa metody `Invoke` do przełączenia się na wątek, na którym została stworzona kontrolka, a następnie wykonuje dostarczoną akcję (przekazaną jako delegat `Action`). Jeśli aktualne wywołanie pochodzi już z właściwego wątku, akcja jest po prostu wykonana bez konieczności przełączania wątków.

W praktyce, `Invoke` jest używane do zamknięcia akcji (funkcji) w kontekście wątku interfejsu użytkownika, co jest konieczne, aby uniknąć problemów związanych z dostępem do elementów interfejsu z wielu wątków. Wszelkie modyfikacje UI, takie jak zmiana tekstu na etykiecie, aktualizacja kontrolki lub inne operacje na elementach interfejsu, powinny być wykonywane na głównym wątku interfejsu użytkownika, a nie na wątku pracującym w tle. `Invoke` pomaga w tym procesie, umożliwiając bezpieczne wykonanie kodu w odpowiednim kontekście wątku interfejsu.

3.2 Generator instancji

Generator instancji umożliwia stworzenie instancji dla problemu Max Subset PDP.
Parametry:

- ilość fragmentów mapy - liczba miejsc cięć (nie licząc '0') na mapie stanowiącej rozwiązanie problemu
- długość fragmentów minimalna / maksymalna - minimalna oraz maksymalna długość fragmentu mapy, który jest określony przez kolejne 2 miejsca cięć
- ilość delecji - liczba usuwanych elementów z multizbioru
- ilość substytucji - liczba podmienianych elementów w multizbiorze

Po uruchomieniu generatora instancji klasa InstanceGenerator wykonuje następujące operacje:

1. Wygenerowanie podanej ilości fragmentów o długości z określonego zakresu

   Lista przechowująca długości fragmentów jest na początku czyszczona (zabezpieczenie przez nadpisywaniem danych). Następnie dla podanej liczby fragmentów dodawane są iteracyjnie do tej listy losowe wartości z podanego zakresu.

2. Wygenerowanie rozwiązania

   Lista przechowująca rozwiązanie jest czyszczona. Następnie do rozwiązania dodawana jest wartość '0' oraz dla każdego fragmentu, z istniejącej już ich listy, dodajemy sumę aktualnie wybranego elementu do istniejącej już sumy. Przykład:
   Lista fragmentów: [3, 4, 2, 1]
   Rozwiązanie: [0, 3, 7, 9, 10]

3. Wygenerowanie instancji wejściowej (multizbioru odległości pomiędzy cięciami)

   Lista przechowująca wartości multizbioru jest czyszczona. Kolejno, dla każdego fragmentu algorytm iteruje po wszystkich fragmentach, sumując wartości kolejnych fragmentów, tworząc w ten sposób wszystkie kombinacje długości między poszczególnymi miejscami cięć w mapie stanowiącej rozwiązanie

4. Wykonanie zadanych błędów na multizbiorze

   Metoda narzucająca błędy w multizbiorze, usuwa i podmienia zadaną wartość elementów z multizbioru.

5. Przesortowanie i zapisanie multizbioru
   Na końcu multizbiór odległości między cięciami zostaje posortowany i w takiej postaci jest już gotowy do użycia przez algorytm Tabu Search.

3.3 Implementacja Tabu Search

W poniższym podrozdziale omówiono wyróżnione w programie parametry metaheurystyki oraz opisano szczegółowo działanie istotnych elementów algorytmu.

3.3.1 Parametry metaheurystyki

- Wielkość listy tabu - długość listy, która przechowuje rozwiązania problemu.
- Procent iteracji bez poprawy przed dywersyfikacją - procent od maksymalnej liczby iteracji, po którym przeprowadzana jest metoda częściowego restartu (w aplikacji zwana dywersyfikacją).
- Procent usuwanego rozwiązania podczas dywersyfikacji - procent od aktualnie istniejącego rozwiązania, który określa ile elementów ma zostać usuniętych podczas metody częściowego restartu.
- Liczba restartów - liczba restartów, po których algorytm generuje nowe rozwiązanie początkowe i zaczyna od nowa zliczać iteracje.
- Maksymalna liczba iteracji - liczba iteracji po których algorytm kończy działanie (jeśli nie założono restartów).

3.3.2 Objaśnienie kluczowych pojęć dla problemu oraz ich konstrukcja

- Rozwiązanie początkowe
  Rozwiązanie początkowe generowane jest według następującego schematu:
  1. Dodanie 0 do rozwiązania
  2. Iteracja po losowo wybranych elementach ze multizbioru wejściowego D
  3. Dodanie losowego elementu innego niż 0 do rozwiązania
  4. Wybranie losowo trzeciego elementu
  5. Sprawdzenie czy po dodaniu trzeciego elementu do rozwiązania, wartości różnic między kombinacją wszystkich 3 elementów nie występują częściej niż te same wartości w multizbiorze D
  6. Jeśli warunek z punktu 5. jest spełniony zakończ generowanie i zwróć listę trzech elementów, która stanowi rozwiązanie początkowe. Jeśli warunek nie został spełniony, wróć do punktu 4.

- Budowa listy tabu
  Lista tabu została skonstruowana w taki sposób aby przechowywała pełne rozwiązania. Jest to zatem lista, przechowująca listy (gdzie każda lista wewnętrzna to rozwiązanie dopuszczalne).

- Funkcja celu
  Funkcja celu została określona jako liczność listy stanowiącej rozwiązanie podzielona przez maksymalną liczność rozwiązania. Teoretyczna wartość maksymalnej liczności rozwiązania (maksymalne m w problemie) jest wyliczana zgodnie z regułą liczby trójkątnej. Liczba trójkątna T_n, o numerze n dana jest wzorem:

  $$T_n = \frac{n \cdot (n+1)}{2}$$

  Znając T_n, które odpowiada za liczność multizbioru możemy wyliczyć wartość n które odpowiada za liczność zbioru stanowiącego rozwiązanie.

- Generowanie sąsiedztwa
  Generowanie sąsiedztwa, stanowiącego nowe rozwiązanie dopuszczalne odbywa się w następujących krokach:
  1. Iteracja przez elementy multizbioru `D`, który reprezentuje dostępne fragmenty do ułożenia w rozwiązaniu.
  2. Dla każdego elementu, jeśli nie znajduje się on już w bieżącym rozwiązaniu (`Solution`), tworzona jest nowa lista `newSolution`, będąca kopią aktualnego rozwiązania z dodanym tym elementem. 3. Nowa lista jest sortowana, a następnie sprawdzane są warunki wykonalności:

  – Sprawdzane jest, czy różnice między każdą parą elementów w `newSolution` są zawarte w multizbiorze `D`.

  – Równocześnie monitorowane są ilości wystąpień każdej różnicy i sprawdzane, czy nie przekraczają one ilości wystąpień w `D`.

  4. Jeśli warunki wykonalności są spełnione, `newSolution` dodawane jest do zbioru sąsiedztwa.
  5. Jeśli element znajduje się już w bieżącym rozwiązaniu (`Solution`), tworzona jest nowa lista `newSolution`, będąca kopią aktualnego rozwiązania z usuniętym tym elementem.
  6. Lista `newSolution` dodawana jest do zbioru sąsiedztwa.
  Ostatecznie metoda zwraca zbiór sąsiedztwa, który zawiera różne możliwe rozwiązania dopuszczalne, różniące się od bieżącego rozwiązania poprzez dodanie lub usunięcie jednego elementu.

- Metoda częściowego restartu - dywersyfikacja
  W procesie dywersyfikacji, wykonywanym po określonej liczbie iteracji bez poprawy rozwiązania podejmowane są następujące kroki:
  1. Obliczenie liczby elementów do usunięcia w celu dywersyfikacji. Wartość ta jest ustalana na podstawie podanego jako parametr procentowego udziału względem liczby elementów w bieżącym rozwiązaniu.
  2. Iteracja w celu usunięcia określonej liczby elementów
  3. Losowe wybieranie elementów z rozwiązania (z pominięciem elementu startowego '0') i usuwanie ich.
  4. Posortowanie listy pozostałych wartości

5. Zapisanie nowego stanu rozwiązania
Tak scharakteryzowany proces dywersyfikacji, przy odpowiednim ustawieniu parametrów może pomóc w uniknięciu utknięcia w lokalnym minimum i jednoczesnym poszerzeniu przestrzeni poszukiwań.

- Metoda restartu
  Metoda restartu sprowadza się do wytworzenia nowego rozwiązania początkowego, od którego metaheurystyka zaczyna przeszukiwanie przestrzeni rozwiązań od nowa. Liczba restartów jest determinowana przez użytkownika.

- Sposób przeszukiwania przestrzeni rozwiązań
  Jest to kluczowy element metaheurystyki, angażujący wszystkie powyżej wymienione składowe algorytmu. Proces ten możemy opisać w następujących krokach:
  1. Rozpoczęcie działania pętli, której ilość wykonań jest równa zadanej liczbie restartów+1, w której wykonujemy kolejne kroki.
  2. Inicjalizacja potrzebnych obiektów - rozwiązanie aktualne, rozwiązanie najlepsze, wartość funkcji celu aktualnie najlepszego rozwiązania, lista tabu, licznik iteracji.
  3. Wygenerowanie rozwiązania początkowego.
  4. Rozpoczęcie pętli, której ilość wykonań jest równa zadanej liczbie iteracji:
     - Sprawdzenie warunku zatrzymania w przypadku żądania zatrzymania algorytmu,
     - Wygenerowanie sąsiedztwa bieżącego rozwiązania,
     - Znalezienie odpowiedniego kandydata spośród sąsiadów - tutaj przeglądana jest lista potencjalnych rozwiązań (sąsiedztwo). Iterujemy przez wszystkich kandydatów. Gdy dany kandydat nie jest na liście zakazanej lub spełnia kryterium aspiracji (wartość funkcji celu dla kandydata jest większa niż dotychczas najlepsza wartość funkcji celu) oraz jednocześnie ma lepszą wartość funkcji celu niż dotychczasowy analizowani przeanalizowani kandydaci, to ten kandydat (sąsiedztwo) staje się nowym rozwiązaniem.
     - Aktualizacja bieżącego rozwiązania na podstawie wybranego kandydata,
     - Obliczenie wartości funkcji celu dla nowego rozwiązania,
     - Aktualizacja najlepszego rozwiązania, jeśli nowe rozwiązanie jest lepsze,
     - Dodanie bieżącego rozwiązania do listy ruchów zakazanych, usuwanie najstarszego, jeśli lista osiągnie maksymalny rozmiar,
     - Sprawdzenie warunku dywersyfikacji i opcjonalne przeprowadzenie dywersyfikacji w przypadku braku poprawy. 5. Zakończenie pętli iteracji, sprawdzenie czy zadany został kolejny restart - jeśli tak, powrót do punktu 2.

Rozdział 4

Eksperymenty obliczeniowe

Przeprowadzone testy obejmowały różnice we wszystkich zamieszczonych w aplikacji parametrach możliwych do zmiany przez użytkownika. W każdym przypadku zbadane zostały zarówno jakość otrzymanego rozwiązania jak i czas pracy algorytmu Tabu Search. Dla każdego zestawu wartości parametrów wygenerowano 10 losowych instancji, w celu uśrednienia wyników.

4.1 Wyniki przeprowadzonych eksperymentów

Tabela 4.1 przedstawia parametry generatora wybrane jako podstawowe do pierwszej tury testów, która miała uwidocznić zależności potrzebne do dostrojenia metaheurystyki. Natomiast Tabela 4.2 ukazuje domyślnie ustawione parametry algorytmu tabu.

Ilość fragmentów mapy	**Minimalna długość fragmentów**	**Maksymalna długość fragmentów**	**Delecje**	**Substytucje**
20	3	15	2	1

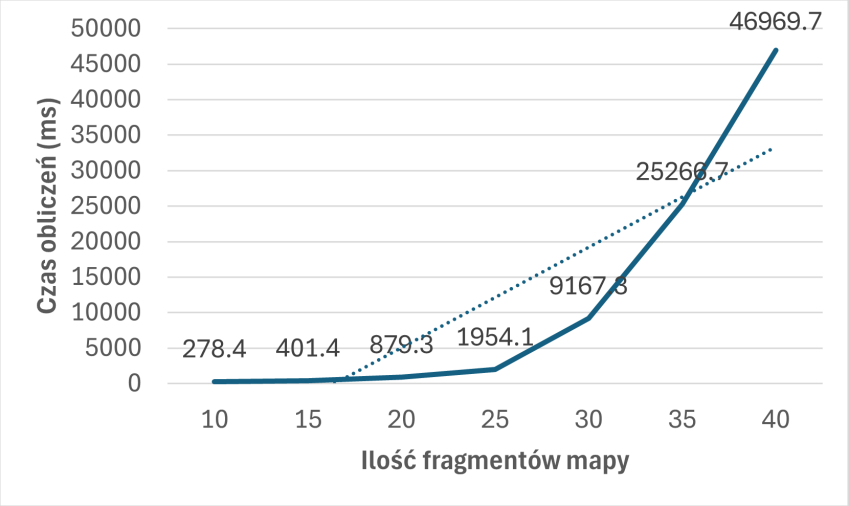
Tabela 4.1: Podstawowe parametry generatora, będące punktem wyjścia do testów

Długość listy tabu	**Procent iteracji przed częściowym restartem**	**Procent usuwanego rozwiązania podczas częściowego restartu**	**Liczba restartów**	**Liczba iteracji**
6	20	20	1	500

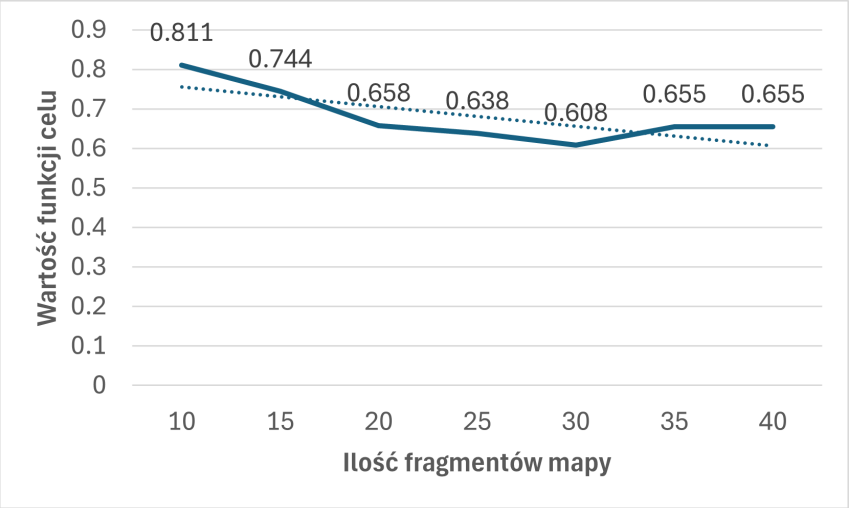
Tabela 4.2: Podstawowe parametry metaheurystyki, będące punktem wyjścia do testów.

Przetestowano wpływ zmian każdego z poniższych parametrów na wartości funkcji celu oraz czasu obliczeń. We wszystkich przeprowadzonych poniżej testach modyfikowano zawsze jeden z parametrów, resztę pozostawiając domyślną (wartości z tabel).

- Ilość fragmentów mapy



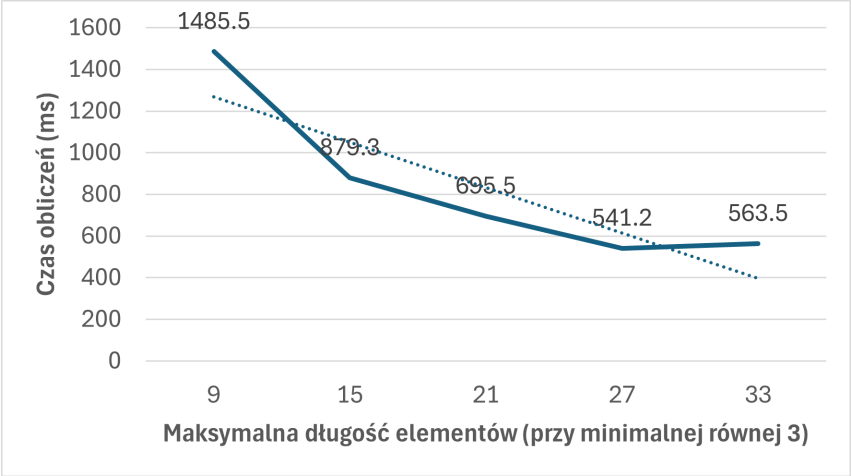
Rysunek 4.1: Zależność wielkości instancji od czasu obliczeń.



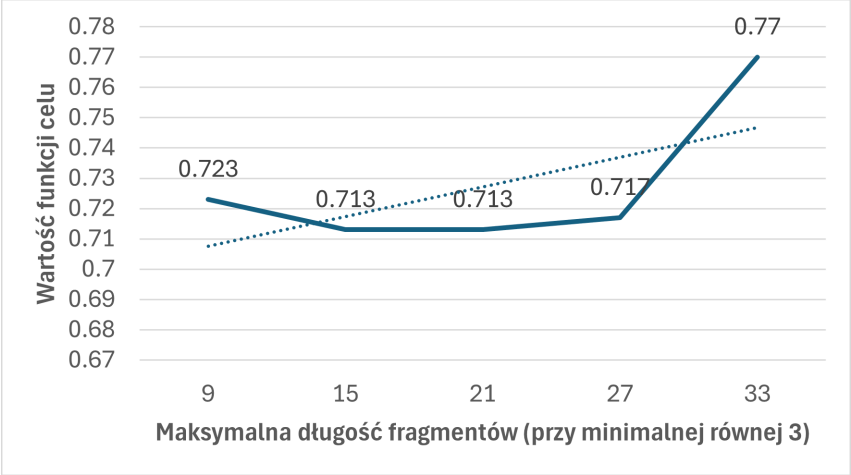
Rysunek 4.2: Zależność wielkości instancji od wartości funkcji celu.

Jak widzimy, na Rysunku 4.1 zwiększenie ilości fragmentów mapy (wielkość instancji) znacznie wpływa na wydłużenie czasu obliczeń. Dla ilości fragmentów równej 45 algorytm potrzebuje już ponad minutę na rozwiązanie problemu. Większa ilość fragmentów przekłada się także na spadnięcie średniej wartości funkcji celu, co widać na Rysunku 4.2. Różnica pomiędzy 10 a 40 elementami przełożyła się na średnią różnicę wartości funkcji celu równą 0.15.

- Długości odcinków w instancji wejściowej



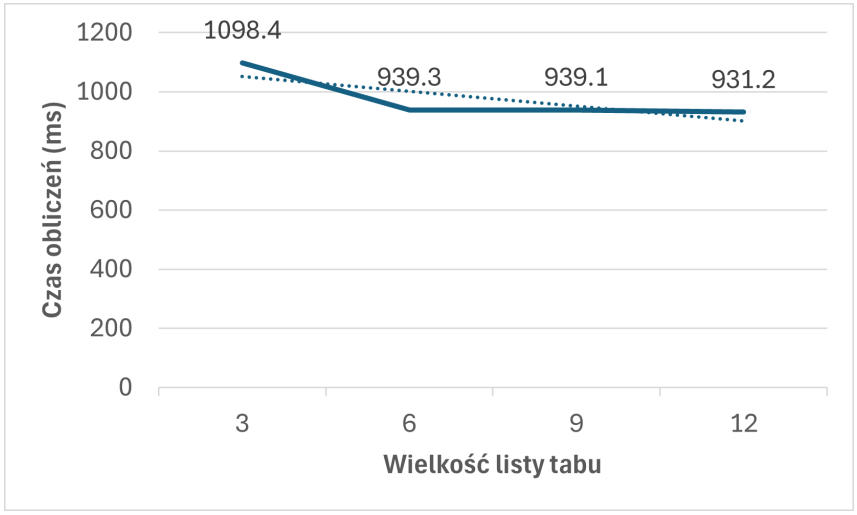
Rysunek 4.3: Zależność oryginalności długości fragmentów od czasu obliczeń.



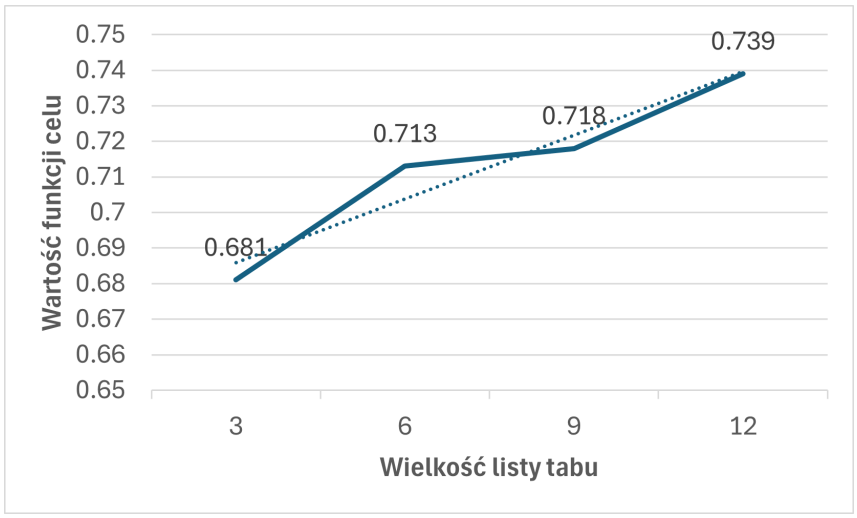
Rysunek 4.4: Zależność oryginalności długości fragmentów od wartości funkcji celu.

Jak pokazuje rysunek 4.3 zwiększenie zakresu, z którego losowane są wartości długości fragmentów mapy pozytywnie przekłada się na skrócenie czasu obliczeń. Można przypuszczać, że bardziej oryginalne długości fragmentów znacznie zmniejszają ilość możliwych kombinacji dopuszczalnych rozwiązań, co ogranicza wielkość sąsiedztwa, którego tworzenie pochłania znaczną część czasu trwania całego algorytmu. Co istotne, większa różnorodność wartości może wpływać pozytywnie także na funkcję celu, co pokazuje linia trendu na Rysunku 4.4. Należy w tym miejscu zaznaczyć, że duża różnorodność wartości (maksymalna długość równa 27 i 33) przekładały się na duże odchylenia od średniego wyniku dla tej próby.

- Długość listy tabu



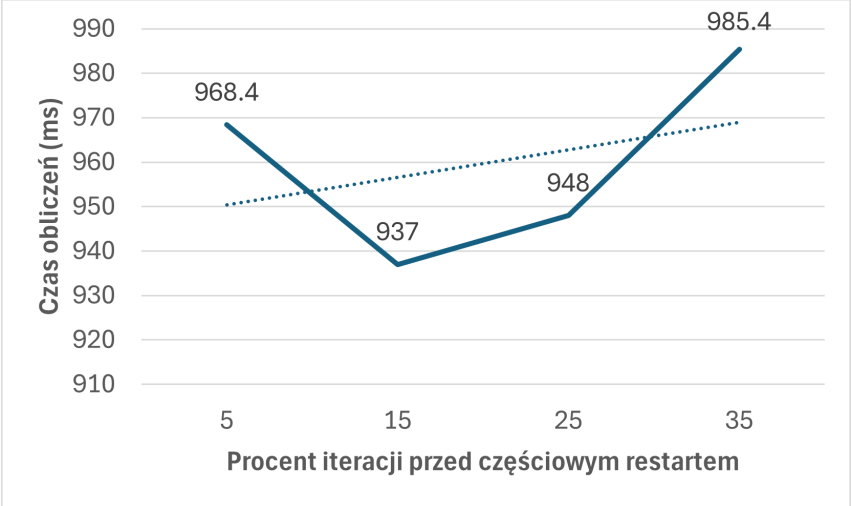
Rysunek 4.5: Zależność długości listy tabu od czasu obliczeń.



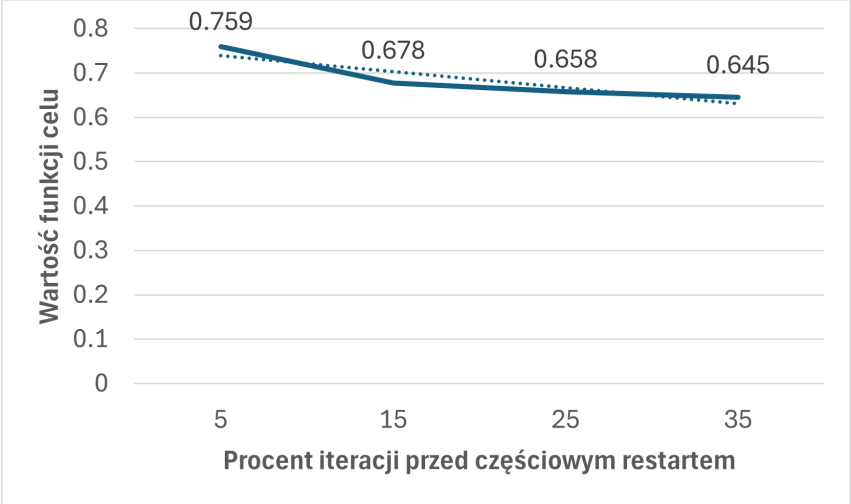
Rysunek 4.6: Zależność długości listy tabu od wartości funkcji celu.

Wielkość listy tabu ma niewielki wpływ na czas obliczeń. Natomiast jeśli chodzi o funkcję celu, zwiększanie listy tabu pozytywnie wpływa na jakość otrzymywanych rozwiązań. Należy zauważyć, że ze względu na budowę listy, która przechowuje całe rozwiązania, a nie pojedyncze elementy, przy testach na większych instancjach powinna ona mieć znacznie większą długość, tak aby spełniała swoje działanie i eliminowała możliwość wpadnięcia w cykl w obrębie lokalnego minimum.

- Procent iteracji bez poprawy przed dywersyfikacją



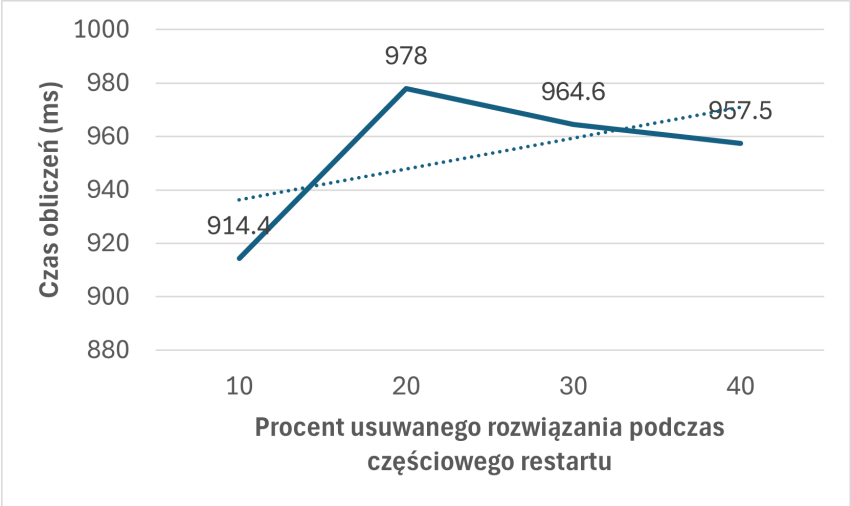
Rysunek 4.7: Zależność procent iteracji bez poprawy przed zastosowaniem metody częściowego restartu od czasu obliczeń.



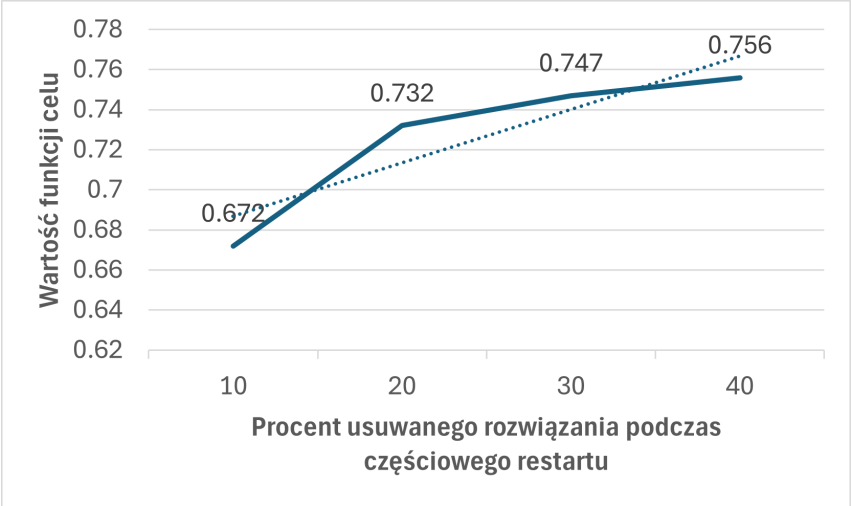
Rysunek 4.8: Zależność procent iteracji bez poprawy przed zastosowaniem metody częściowego restartu od funkcji celu.

Powyższy wykres pokazuje, iż nie ma dużej różnicy względem większej liczby częściowych restartów, a całkowitym czasem obliczeń. Jednak zmniejszenie liczby częściowych restartów zauważalnie przekłada się na spadek średniej wartości funkcji celu. Zwiększając do 30% wartość iteracji bez poprawy przed wykonaniem rozwiązania, wartość funkcji celu obniżyła się średnio o 0.11 względem początkowych 5% iteracji bez poprawy.

- Usuwana ilość rozwiązania przy dywersyfikacji



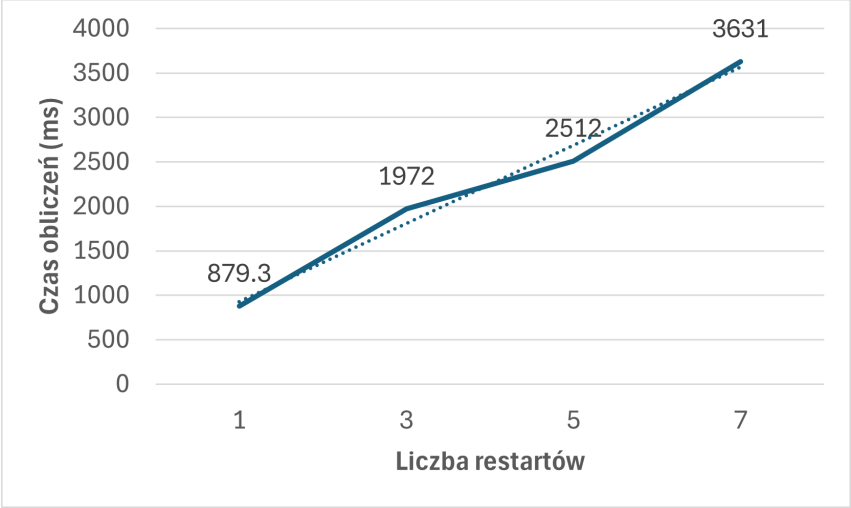
Rysunek 4.9: Zależność usuwanej ilość rozwiązania przy dywersyfikacji od czasu obliczeń.



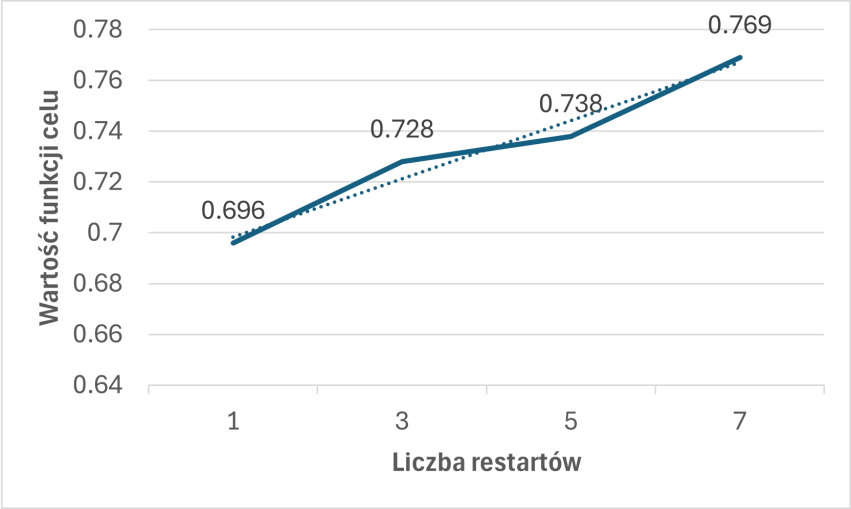
Rysunek 4.10: Zależność usuwanej ilość rozwiązania przy dywersyfikacji od funkcji celu.

Wykres z Rysunku 4.10 obrazuje dużą zależność otrzymywanej średniej wartości funkcji celu od procentu usuwanego rozwiązania podczas metody częściowego restartu. Można przypuszczać, że w zaimplementowanym algorytmie, metoda ta, obok listy tabu, jest drugą najważniejszą sekcją, eliminującą możliwość utknięcia w lokalnym optimum. Usuwając większe części bieżącego rozwiązania, jednocześnie posiadając listę rozwiązań zakazanych, mamy większą szansę przenieść się do wcześniej nieodkrytych obszarów przestrzeni rozwiązań i uciec tym samym z lokalnego minimum.

- Liczba restartów



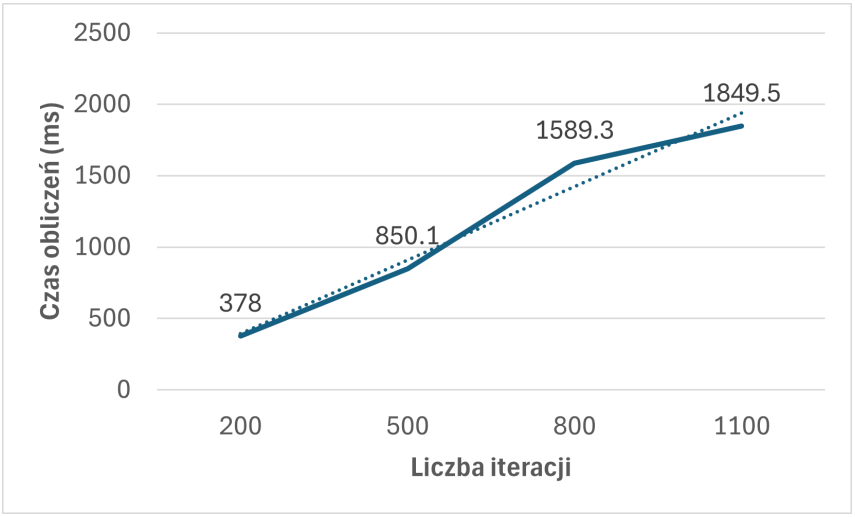
Rysunek 4.11: Zależność parametru liczby restartów od czasu obliczeń.



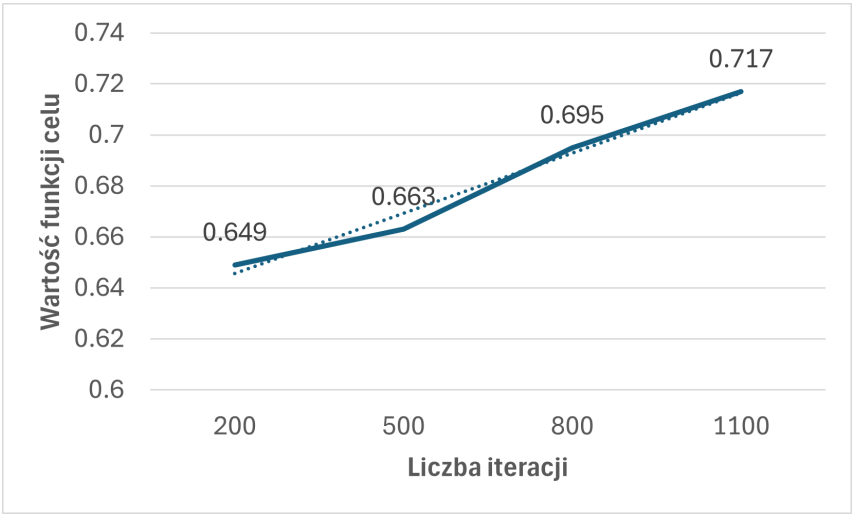
Rysunek 4.12: Zależność parametru liczby restartów od funkcji celu.

Liczba restartów, zgodnie z przewidywaniem, skorelowana jest w dużym stopniu z łącznym czasem obliczeń. Z drugiej strony, zwiększenie liczby restartów może znacznie przyczyniać się do znalezienia lepszego rozwiązania końcowego, poprzez zaczęcie od nowego rozwiązania początkowego. Zachowanie odpowiedniej ilości restartów okazuje się bardzo ważne, gdy celem jest zachowanie balansu między jakością rozwiązania a czasem jego otrzymania. Prawdopodobnie, większy procent usuwanego rozwiązania w trakcie metody częściowego restartu może dać podobne korzyści, nie zwiększając przy tym znacznie czasu obliczeń.

- Liczba iteracji



Rysunek 4.13: Zależność liczby iteracji od czasu obliczeń.



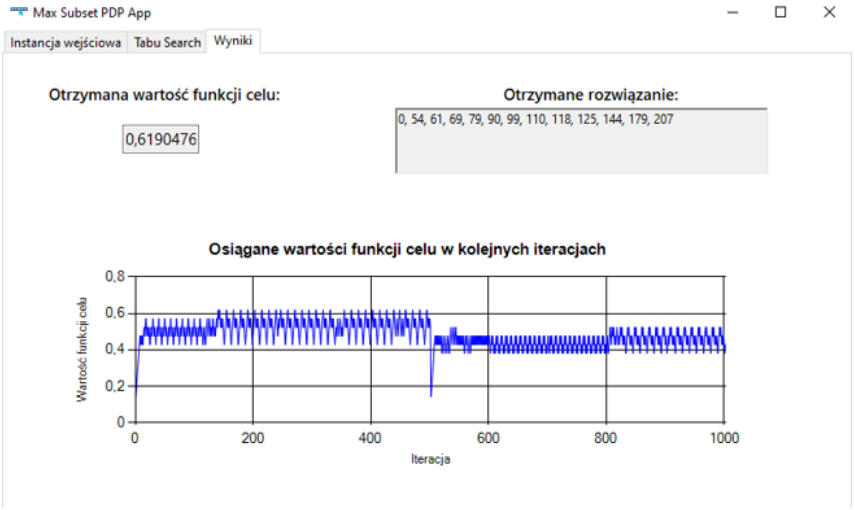
Rysunek 4.14: Zależność liczby iteracji od funkcji celu.

Patrząc na wykres zależności iteracji od czasu obliczeń można stwierdzić, iż jest to, razem z samą wielkością instancji, najważniejszy parametr kontrolujący łączny czas pracy algorytmu. Dla początkowych testów, rosnąca liczba iteracji znacznie zwiększała otrzymywaną średnią wartość funkcji celu, jednak trend ten jest coraz słabszy z dalszym zwiększaniem tej wartości. Sugeruje to, iż dla danej wielkości instancji istnieje pewna optymalna wartość iteracji, której nie warto przekraczać, a lepiej połączyć ją z odpowiednim ustawieniem ilości restartów i częstości częściowych restartów.

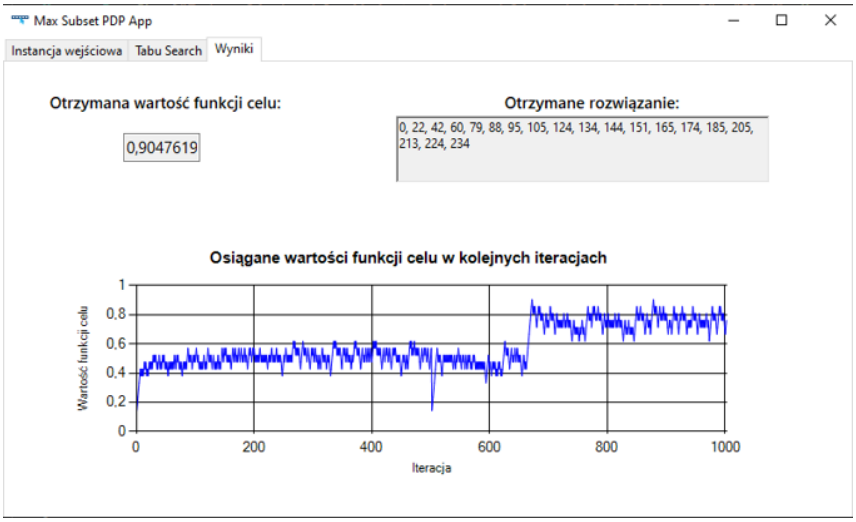
4.2 Dostrojenie metaheurystyki i wyniki testów na większych instancjach

Wyniki początkowo wykonanych testów i obserwacje wykresów funkcji celu w przeprowadzonych próbach pozwoliły ustalić pewne informacje na temat danych parametrów metaheurystyki.

Po pierwsze należy zwrócić uwagę na istotnie większą długość listy tabu, od pierwotnie zakładanej, ze względu przechowywanie przez nią pełnych rozwiązań. Wykresy na Rysunkach 4.15 i 4.16 ukazują różnicę w przeszukanej przestrzeni rozwiązań, porównując próbę na tej samej instancji, ze zmienioną wartością długością listy tabu.



Rysunek 4.15: Wyniki próby dla parametrów domyślnych i długości listy tabu równej 5.

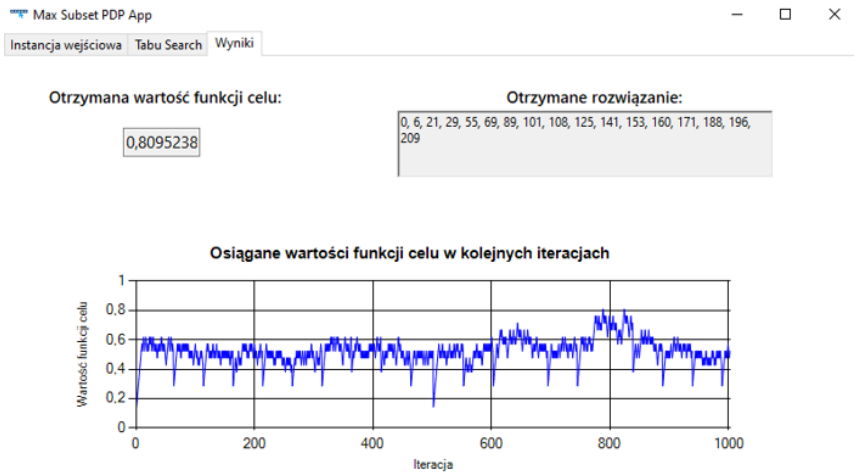


Rysunek 4.16: Wyniki próby dla parametrów domyślnych i długości listy tabu równej 50.

Jak można dostrzec, lista tabu przechowująca 50 rozwiązań znacznie ograniczyła częstość wpadania w cykle tych samych rozwiązań.

Kolejna obserwacja dotyczy istotności podniesienia procentu usuwanego rozwiązania podczas metody częściowego restartu.

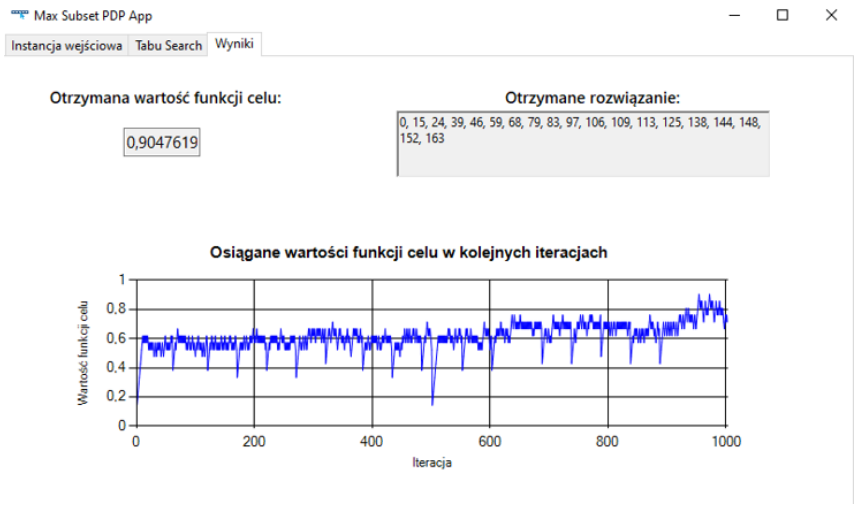
Rysunek 4.17: Wyniki próby dla parametrów domyślnych i procentu usuwanego rozwiązania podczas dywersyfikacji równym 15.



Rysunek 4.18: Wyniki próby dla parametrów domyślnych i procentu usuwanego rozwiązania podczas dywersyfikacji równym 45.

Jak pokazują wykresy na Rysunkach 4.17 i 4.18, podniesienie usuwanej wartości rozwiązania podczas stosowanej dywersyfikacji z 15% do 45% znacząco zmienia sposób przeglądania przestrzeni rozwiązań. Mimo wielokrotnego spadku poniżej satysfakcjonującej wielkości, usuwanie większej ilości bieżącego rozwiązania prowadzi pośrednio do odnajdywanie nowych maksimum lokalnych, a tym samym zwiększa szansę odnalezienia maksimum globalnego, które jest poszukiwanym rozwiązaniem.

Łącząc oba poprzednie wnioski, łącząc powiększoną listę tabu oraz zwiększając procent usuwanego rozwiązania podczas częściowego restartu, dla tej samej instancji zdecydowanie częściej udaje uzyskać się wartość funkcji celu równą 0.9, co wcześniej było bardzo rzadkie. Przeszukanie przestrzeni rozwiązań i wynik prezentuje Rysunek 4.19.



Rysunek 4.19: Wyniki próby dla długości listy tabu równej 50, procentowi usuwanego rozwiązania równym 45 i pozostałych parametrów domyślnych.

Przeprowadzono kolejną turę testów, na większych instancjach, z dostrojonymi na podstawie poprzednich testów parametrami. Tabela 4.3 pokazuje ustawione wartości parametrów generatora, a Tabela 4.4 wartości parametrów tabu. Przeprowadzono osobno testy bez błędów a następnie z wybranymi wartościami delecji i substytucji.

Ilość fragmentów mapy	**Minimalna długość fragmentów**	**Maksymalna długość fragmentów**	**Delecje**	**Substytucje**
60	3	75	0	0

Tabela 4.3: Ustawione parametry generatora.

Długość listy tabu	**Procent iteracji przed częściowym restartem**	**Procent usuwanego rozwiązania podczas częściowego restartu**	**Liczba restartów**	**Liczba iteracji**
150	15	40	2	500

Tabela 4.4: Ustawione parametry metaheurystyki.

1. Testy bez błędów

Wartość funkcji celu	**Czas obliczeń (ms)**
0.96	78672
0.54	56478
0.44	49867
0.52	53931
0.98	75759
0.46	55626
0.44	54108
0.42	48223
0.61	57837
0.46	50646

Tabela 4.5: Wyniki przy braku błędów w instancji. Średni czas obliczeń: 58s.

Metaheurystyka w dwóch próbach z dziesięciu była w stanie znaleźć bardzo zadowalające rozwiązania o wartości funkcji celu powyżej 0.9. Jednak większość wyników cechuje wartość w okolicach 0.5, można zatem rozważać dalszą potrzebę dostosowania parametrów liczby iteracji i metod dywersyfikacji.

2. Testy z błędami delecji

Wartość funkcji celu	**Czas obliczeń (ms)**
0.39	47317
0.33	49370
0.38	55854
0.61	53681
0.33	47369
0.57	56132
0.33	48140
0.86	57956
0.38	55892
0.36	57348

Tabela 4.6: Wyniki przy 3 błędach delecji w instancji. Średni czas obliczeń: 53s.

Po wynikach testu z błędami delecji, można stwierdzić że są one niemałym problemem dla testowanej instancji. Średni wynik wartości funkcji okazał się być niższy o blisko 0.15 w stosunku do instancji tej samej wielkości bez błędów. W jednej z 10 instancji algorytm uzyskał zdecydowanie wyróżniający się wynik na poziomie 0.86.

3. Testy z błędami substytucji

Wartość funkcji celu	**Czas obliczeń (ms)**
0.54	46572
0.36	48509
0.38	51537
0.47	43213
0.33	45667
0.34	48926
0.92	64980
0.46	56709
0.36	51341
0.38	47853

Tabela 4.7: Wyniki przy 3 błędach substytucji w instancji. Średni czas obliczeń: 51s.

Przy 3 błędach substytucji w instancji wejściowej średni wynik funkcji celu jest niemal identyczny jak w przypadku testach z błędami delecji. Tutaj również odnotowany został jeden wyróżniający się wynik równy 0.92.

4. Testy z obydwoma rodzajami błędów

Wartość funkcji celu	**Czas obliczeń (ms)**
0.43	53487
0.34	46781
0.6	64387
0.9	54656
0.33	42758
0.36	53295
0.46	51142
0.39	48612
0.86	68144
0.41	46785

Tabela 4.8: Wyniki przy obu rodzajach błędów w instancji. Średni czas obliczeń: 53s.

Przy nałożeniu 3 błędów delecji i 3 substytucji na instancję wejściową wyniki algorytmu cechuje podobny średni poziom, w okolicach 0.45, podobnie jak przy jednym rodzaju zadanych błędów. W tej próbie dwukrotnie odnotowano odbiegające od średniej wyniki - 0.86 oraz 0.9.

Rozdział 5

Ocena skuteczności Tabu Search i wnioski

Utworzona aplikacja wraz z zaimplementowaną metaheurystyką tabu umożliwiają znajdywanie rozwiązań, które można uznać za akceptowalne, zarówno od strony funkcji celu jak i aspektu czasowego, dla maksymalnie kilkudziesięciu fragmentów, z których jest stworzona oryginalna mapa. Dokładna ilość uzależniona jest mocno od występowania błędów, parametrów tabu, a także różnorodności długości analizowanych fragmentów. Skutecznemu znajdywaniu dobrych rozwiązań sprzyja duża mnogość długości fragmentów względem samej ich ilości.

Szacunkowo, złożoność czasowa całego algorytmu tabu wynosi $O(r*i*k*n^2*log(n))$, gdzie r to liczba restartów, i to liczba iteracji w każdym restarcie, k to rozmiar multizbioru D, a n to rozmiar bieżącego rozwiązania. Generowanie sąsiedztwa wymaga porównania każdej pary elementów w bieżącym rozwiązaniu wraz z wyliczeniem wartości funkcji celu, co daje kwadratową złożoność czasową, przekładając się mocno na całkowitą złożoność implementacji.

Należy przy tym zauważyć także inne możliwości poprawy pracy algorytmu, ulepszając jego niektóre aspekty. Zakładanie z góry określonej liczby iteracji może nie być korzystne, ze względu na fakt, że możemy tym samym zakończyć działanie tabu gdy faktycznie zmierza w kierunku lepszego rozwiązania (co jest możliwe nawet po bardzo dużej ilości wykonanych ruchów). Lepszym podejściem mogłoby być zatrzymanie algorytmu gdy nie znajduje on już lepszych rozwiązań, po uprzednio wykonanych ruchach częściowo losowych oraz restartach, które sprzyjają znajdywaniu nowych i potencjalnie lepszych rozwiązań. Powyższą tezę potwierdzają wyniki, które dowodzą istotności parametrów liczby restartów oraz odpowiedniej kombinacji częstości częściowego restartu i procentu usuwanego przy tym bieżącego rozwiązania.

Przeprowadzone eksperymenty obliczeniowe prowadzą do jeszcze do kilku wniosków i podkreślają istotność strojenia metaheurystyki pod dany problem optymalizacyjny. Z obecnych w algorytmie parametrów metaheurystyki, kluczowe znaczenie wydaje się mieć długość listy tabu. Zbyt krótka lista, jak zostało odnotowane na etapie testów, może spowodować cykliczne powracanie do już wygenerowanych rozwiązań. W stworzonym algorytmie, ze względu na swój charakter, lista tabu powinna być stosunkowo długa. Ostatnim istotnym punktem jest charakter rozwiązania początkowego, które w obecnej postaci stanowi bardzo małą pomoc dla metaheurystyki, szczególnie przy dużych instancjach.

Literatura

[1] Miroslaw Blocho. *Smart Delivery Systems - Solving Complex Vehicle Routing Problems.*

[2] Mark Cieliebak, Stephan Eidenbenz, and Paolo Penna. Partial digest is hard to solve for erroneous input data. *Theoretical Computer Science*, 349(3):361–381, 2005.

[3] Tomasz Głowacki, Adam Kozak, Marcin Borowski, and Piotr Formanowicz. O algorytmach asemblacji długich łancuchów peptydowych, 2010. `https://depot.ceon.pl/bitstream/handle/123456789/6354/kkapd2010.pdf?sequence=1&isAllowed=y`.

[4] Professor Marta Kasprzak Ph.D., Dr. Habil. Algorytmy kombinatoryczne w bioinformatyce. `https://www.cs.put.poznan.pl/mkasprzak/akb/akb.html`, 2023.

[5] Professor Marta Kasprzak Ph.D., Dr. Habil. Zaawansowane programowanie. `https://www.cs.put.poznan.pl/mkasprzak/zp/zp.html`, 2023.

[6] Agnieszka Rybarczyk, Alain Hertz, Marta Kasprzak, and Jacek Blazewicz. Tabu search for the rna partial degradation problem. *International Journal of Applied Mathematics and Computer Science*, 27(2):401–415, 2017.

Instytut Informatyki, Wydział Informatyki i Telekomunikacji
Politechnika Poznańska

Skład przy użyciu systemu LaTeX na platformie Overleaf.