Sprawozdanie

Zaawansowane Programowanie

Bioinformatyka II stopnia, 1 semestr

Algorytm Tabu Search do wyszukiwania miejsc cięć enzymatycznych z instancji.

Jakub Sudoł, 147906

Poznań 2024

Spis Treści

1. Wstęp

W ostatnich latach, rozwój technologii informatycznych znacząco wpłynął na postęp w dziedzinie bioinformatyki, oferując nowe narzędzia i metody analizy danych biologicznych. Jednym z kluczowych zadań w tej dziedzinie jest identyfikacja i analiza punktów cięć enzymatycznych, co ma fundamentalne znaczenie w rozwijaniu metod umożliwiających analizę sekwencji biologicznych.

Stworzony program, oparty na algorytmie Tabu Search, reprezentuje nowoczesne podejście do problemów optymalizacyjnych, charakteryzujących się złożonymi przestrzeniami przeszukiwań. Algorytm Tabu Search, wykorzystany w programie, jest techniką heurystyczną zaprojektowaną do znajdowania rozwiązań optymalnych lub suboptymalnych w rozległych przestrzeniach wyszukiwania, gdzie tradycyjne metody często zawodzą. Jego adaptacja do identyfikacji punktów cięć enzymatycznych pozwala na efektywne przeszukiwanie multizbiorów odległości, maksymalizując liczbę identyfikowanych cięć, co jest jednym z kluczowych czynników w analizie między innymi sekwencji DNA i RNA.

Celem tego projektu było nie tylko zastosowanie algorytmu Tabu Search w specyficznym kontekście bioinformatycznym, ale także demonstracja jego skuteczności w rozwiązywaniu problemów optymalizacyjnych w bioinformatyce. Program, opracowany w środowisku C# Windows Forms, charakteryzuje się intuicyjnym interfejsem użytkownika, który ułatwia wprowadzanie danych, konfigurację parametrów algorytmu oraz wizualizację wyników.

W niniejszym sprawozdaniu przedstawiono wstęp teorytyczny do tematu, szczegółowy opis projektu, w tym architekturę programu oraz algorytm Tabu Search. Omówiono także metodologię testowania programu oraz analizę wyników, demonstrując skuteczność i efektywność zaproponowanego rozwiązania.

2. Teoria

2.1 Metaheurystyka

Metaheurystyka to zaawansowany rodzaj algorytmu optymalizacyjnego, zaprojektowany do rozwiązywania trudnych problemów optymalizacyjnych, które są zbyt złożone dla tradycyjnych metod optymalizacyjnych. Charakteryzują się one zdolnością do efektywnego przeszukiwania bardzo dużych przestrzeni rozwiązań w poszukiwaniu optymalnych lub suboptymalnych rozwiązań, często w sytuacjach, gdzie dokładne metody optymalizacyjne są niepraktyczne z powodu wysokich wymagań obliczeniowych. Metaheurystyki opierają się na różnych strategiach, takich jak symulowane wyżarzanie, algorytmy genetyczne, optymalizacja rojem cząstek, algorytm mrówkowy oraz Tabu Search, Ich kluczową cechą jest zdolność do unikania lokalnych optimum i kontynuowania poszukiwań w celu znalezienia lepszych rozwiązań, co jest realizowane poprzez stosowanie różnorodnych strategii eksploracji i eksploatacji.

Metaheurystyki są stosowane w szerokim zakresie dziedzin i problemów, od bioinformatyki i inżynierii przez logistykę i problemy harmonogramowania. Ich elastyczność i skuteczność czynią je atrakcyjnym wyborem w sytuacjach, gdzie metody dokładne nie są w stanie uzyskać wyniku w rozsądnym czasie lub rozwiązanie dokładne nie istnieje. W kontekście bioinformatyki, metaheurystyki umożliwiają efektywne rozwiązywanie problemów związanych z analizą i interpretacją dużych zbiorów danych genetycznych i molekularnych. Przykładem zastosowania metaheurystyk jest identyfikacja punktów cięć enzymatycznych w sekwencjach DNA, gdzie algorytm Tabu Search może być używany do maksymalizacji liczby identyfikowanych cięć, przeszukując złożone przestrzenie rozwiązań w poszukiwaniu optymalnej sekwencji cięć.

2.2 Opis Problemu

Wybrany problem stanowi adaptację klasycznego Problemu Partial Digest (PDP), polegającą na analizie losowo wygenerowanej mapy długości i na podstawie multizbioru stworzonego w oparciu o tę mapę należy odnaleźć miejsca cięć restrykcyjnych maksymalizując liczbę elementów *m* w sekwencji punktów cięć. Problem charakteryzuje się możliwością wystąpienia błędów w wczytywanej instancji. Jest to ciekawe urozmaicenie problemu.

W ramach tego problemu generujemy losową mapę długości, która symuluje hipotetyczną sekwencję biologiczną, a następnie definiujemy multizbiór odległości między wszystkimi parami punktów cięć. Odległości te są analogiczne do tych, które można by uzyskać w rzeczywistym eksperymencie z użyciem enzymów restrykcyjnych. Głównym celem jest odtworzenie sekwencji punktów cięć w jak najlepszy sposób. W odróżnieniu od tradycyjnego PDP, gdzie głównym celem jest jedynie odtworzenie możliwej sekwencji punktów cięć, w tym przypadku dodatkowym wyzwaniem jest znalezienie takiego rozwiązania, które maksymalizuje liczbę punktów cięć. Oznacza to, że oprócz identyfikacji poprawnej sekwencji punktów cięć, wysiłek koncentruje się na zwiększeniu kompleksowości mapy enzymatycznej. Dodatkowo zakładamy w problemie, że instancja wejściowa może zawierać błędy które mogą wpłynąć

na końcowe rozwiązanie i jego wierność do oryginalnej sekwencji.

Wybrany problem Partial Digest Problem można opisać w następujący sposób:

Instancja: $\boldsymbol{D = \{d_1, \ldots, d_k\}\ z\ k = (\frac{m}{2})}$

Rozwiązanie: $\boldsymbol{P = \{p_m, \ldots, p_m\}\ taki, że\ \{|p_i - p_j|: 1 \leq i < j \leq m\} = D'}$

i liczba pokrywających się elementów w D i D' jest maksymalna.

2.3 Opis Algorytmu

Algorytm Tabu Search (TS) jest zaawansowaną metaheurystyką optymalizacyjną, opracowaną przez Freda Glovera w latach 80-tych XX wieku, służącą do rozwiązywania problemów optymalizacyjnych. Cechuje się ona unikalnym podejściem do przeszukiwania przestrzeni rozwiązań, które pozwala na efektywne omijanie lokalnych optimum i dążenie do znalezienia rozwiązania globalnie optymalnego lub suboptymalnego. Algorytm Tabu Search jest szeroko stosowany w różnorodnych dziedzinach, takich jak bioinformatyka, logistyka, finanse i wiele innych, dzięki swojej elastyczności i zdolności do radzenia sobie z złożonymi problemami optymalizacyjnymi.

Algorytm Tabu dzięki swojej elastyczności może być zaimplementowany na wiele różnych sposobów. Rodzaje zmiennych warunkujących działanie algorytmu zależą od implementacji jednak jest kilka elementów które są stałe dla niemal wszystkich zastosowań. Centralnym elementem algorytmu TS jest lista tabu, która jest rodzajem pamięci krótkoterminowej przechowującej informacje o niedawno wykonanych ruchach lub rozwiązaniach, które są tymczasowo zakazane. Celem listy tabu jest zapobieganie powracaniu do niedawno odwiedzonych rozwiązań, co pomaga algorytmowi unikać cykli i zachęca do eksploracji nowych obszarów przestrzeni rozwiązań. List tabu w obrębie jednego programu może być wiele. Na przykład jedna zabrania usuwania a druga dodawania nowych elementów. Drugim kluczowym elementem algorytmu Tabu jest Metoda oceny (celu). Każde potencjalne rozwiązanie jest oceniane za pomocą funkcji oceny, która odzwierciedla, jak dobrze dane rozwiązanie spełnia cel optymalizacyjny. Metoda ta jest kluczowa dla kierowania procesem przeszukiwania w kierunku optymalnych lub suboptymalnych rozwiązań. W zależności od rodzaju rozwiązywanego problemu metoda celu może przybierać różne formy natomiast jej

wystąpienie w algorytmie jest kluczowe. Ostatnim kluczowym elementem algorytmu Tabu jest sąsiedztwo. Definiowanie sąsiedztwa jest niezbędne do eksploracji przestrzeni rozwiązań. Algorytm TS w każdym kroku generuje "sąsiadujące" rozwiązania poprzez wprowadzenie niewielkich modyfikacji do aktualnego rozwiązania. Następnie, spośród tych sąsiadujących rozwiązań, wybierane jest to, które najlepiej spełnia funkcję oceny, przy uwzględnieniu listy tabu i kryteriów aspiracji.

Schemat działania algorytmu jest całkiem prosty i uniwersalny dla większości zastosowań: Algorytm zaczyna od utworzenia rozwiązania początkowego. Można je uzyskać za pomocą losowości lub innej metaheurystyki jak algorytm zachłanny. Następnie:

1. Algorytm generuje sąsiedztwo na podstawie aktualnego rozwiązania
2. Ocenia każde sąsiedztwo i sprawdza czy nie występuje ono na listach tabu
3. Wybiera najlepsze rozwiązanie spośród tych, które nie są zabronione
4. Aktualizuje listę tabu i, jeśli to konieczne, najlepsze rozwiązanie

Ten proces jest powtarzany przez określoną liczbę iteracji lub do momentu osiągnięcia innych kryteriów zakończenia, takich jak brak poprawy w określonej liczbie iteracji. Pomiędzy wyżej opisywanymi krokami znajdują się także metody wychodzenia z optimum lokalnego które są wykorzystywane w momencie gdy nie jest możliwe utworzenie żadnego nowego sąsiedztwa lepszego niż dotychczasowe rozwiązanie.

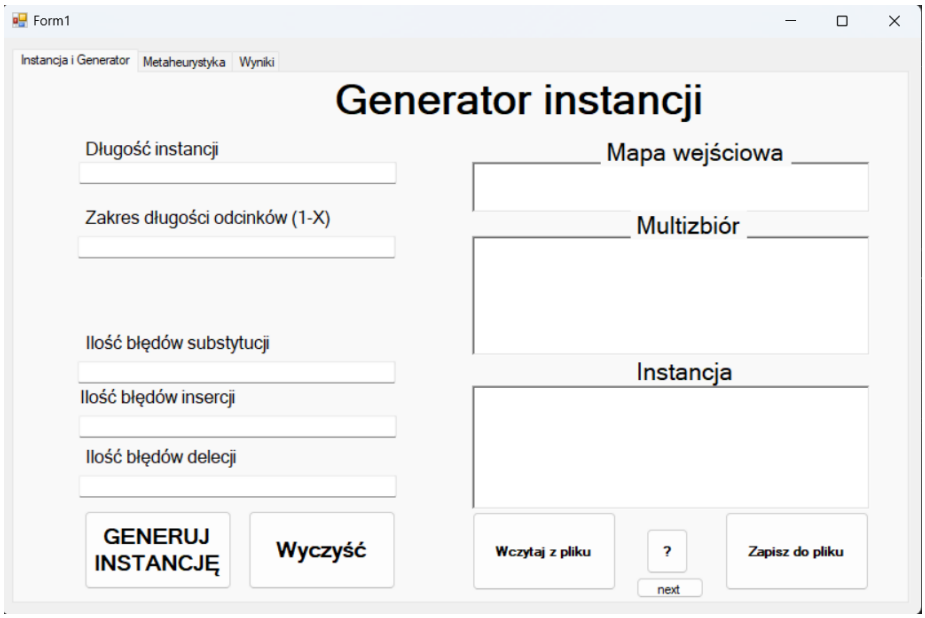
3. Opis Programu

Program został napisany w języku C# i wykorzystuje Windows Forms jako graficzny interfejs. Program został podzielony na dwa pliki. Jeden obsługuje interfejs graficzny, generator instancji, funkcje zapisu i odczytu pliku oraz analizę wyników działania algorytmu. Drugi plik jest całkowicie poświęcony algorytmowi. Łącznie obejmuje około 800 linijek kodu.

3.1 GUI

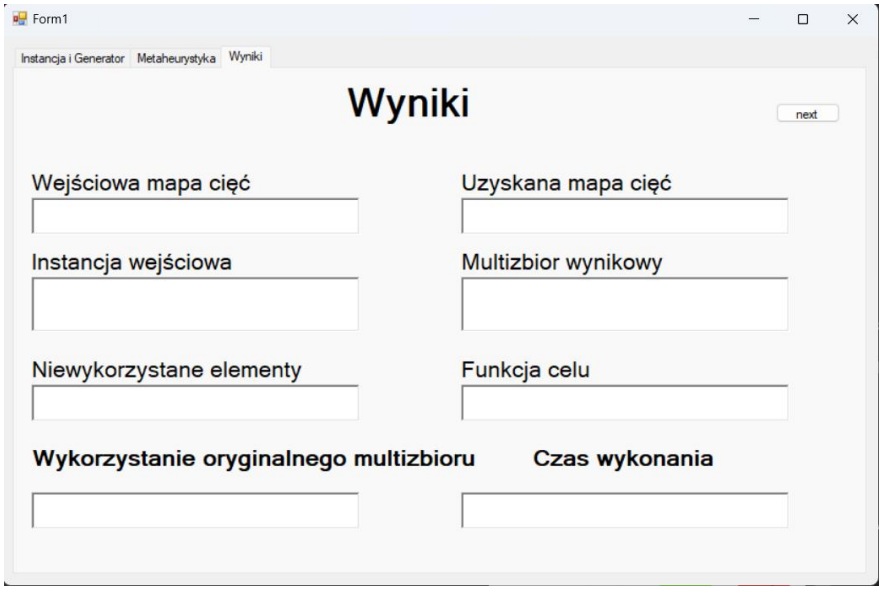
Program wykorzystuje interfejs Windows Forms. Aby rozdzielić trzy różne etapy korzystania z programu, wykorzystano zakładki. Pozwala to na uporządkowanie programu i zaprojektowaniu bardziej czytelnego interfejsu. Pierwsza zakładka obejmuje generator instancji. Znajduje się tam 5 parametrów, za pomocą których określa się właściwości generowanej instancji.

Pierwszym z parametrów jest wielkość instancji. Określa ona ilość elementów w mapie początkowej która zawiera odległości między punktami cięć. Pole przyjmuje tylko jedną liczbę. Drugim parametrem to zakres określający jak duże odległości ma zawierać mapa. Tak samo jak wcześniej, kontrolka *textBox* przyjmuje tylko jedną wartość liczbową. Trzy parametry na samym dole to ilość błędów jakie wprowadza generator. Generator obejmuje błędy insercji, delecji i substytucji. Jeśli nie zostaną wprowadzone żadne wartości, program automatycznie uzupełni pola tekstowe wartością *0*. Poza polami tekstowymi zawierającymi parametry, znajdują się trzy *richTextBox*. Pierwszy zawiera mapę, drugi multizbiór stworzony na podstawie mapy a trzeci instancję wejściową która jest tworzona w oparciu o parametry błędów. Poza wspomnianymi polami *textBox*, zakładka generatora instancji obejmuje 5 przycisków. Pierwszy z nich generuje instancję, drugi czyści wszystkie pola. Następnie są przyciski sterujące zapisywaniem i odczytywaniem plików. Ostatnim przyciskiem jest pomoc, która podpowiada jaki format musi mieć plik wejściowy.



Kolejna zakładka obejmuje metaheurystykę. Przede wszystkim obejmuje pola tekstowe z parametrami do metaheurystyki. Pierwszym parametrem jest wielkość list tabu. Jest to maksymalna liczba, powyżej której najstarszy element jest usuwany. Przyjmuje jedną wartość liczbową. Kolejnym polem tekstowym jest liczba progu pogarszania. Określa ona po którym usunięciu z rozwiązania algorytm porzuca to rozwiązanie by wygenerować nowe. Następnie znajduje się pole tekstowe w którym użytkownik określa jak wiele iteracji bez poprawy najlepszego rozwiązania może wykonać algorytm zanim skończy działanie. Ostatnim parametrem jest maksymalny czas działania algorytmu, po którym przerywane są obliczenia i zwracane jest najlepsze rozwiązanie. Oprócz wymienionych pól tekstowych jest jedno pole, które wyświetla aktualne najlepsze rozwiązanie i jest regularnie odświeżane podczas działania algorytmu. Na tej zakładce znajdują się także dwa przyciski. Jeden odpowiada za włączenie algorytmu a drugi za jego przedwczesne zatrzymanie. Po wcześniejszym zatrzymaniu zwracany jest dotychczasowy najlepszy wynik. Na ekranie znajduje się także pasek postępu, który pokazuje upływ czasu w stosunku do maksymalnej jego wartości.

Ostatni ekran obejmuje wyniki działania algorytmu i podstawowe porównanie otrzymanych danych z danymi wejściowymi. Znajdują się tutaj pola tekstowe w które wpisywane są wartości. Pola po lewej stronie obejmują dane wejściowe jak mapa początkowa czy multizbiór wejściowy. Po prawej stronie znajdują się dane otrzymane w wyniku działania programu. Cztery dolne pola tekstowe zawierają porównania takie jak elementy którymi różnią się multizbiory, procent wykorzystania multizbioru wejściowego, różnice między mapami oraz czas działania programu.



3.2 Generator instancji

Generator przyjmuje 5 parametrów: ilość elementów w mapie, jak duże elementy są w mapie oraz ilość trzech rodzajów błędów. Po wciśnięciu przycisku do generowania instancji program sprawdza które pola są zapełnione. Jeśli wszystkie pola są puste to generator tworzy instancję od początku. Jeśli podana jest mapa, ale brak multizbioru, generator tworzy multizbiór na podstawie podanej mapy. Tak samo jest z podaniem multizbioru, jeśli podany jest multizbiór to program wprowadzi do niego błędy zgodnie z ustawieniami a następnie zwróci gotową instancję wejściową.

Generowanie mapy odbywa się przy użyciu *random.Next().* Generowane jest tyle odległości ile podana liczba w polu tekstowym. Generator liczb losowych ma zakres od 1 do liczby podanej przez użytkownika. Wygenerowane elementy są dodawane do listy ‘mapa’.

Następnie generowany jest multizbiór. Odczytywane są elementy mapy. Najpierw dodawane są wszystkie elementy mapy a później dodawana jest suma wszystkich elementów do nowej listy. Następnie od sumy elementów odejmowany jest pierwszy element i wynik tego działania jest dodawany do listy multizbioru. Następnie za pomocą zagnieżdżonych pętli obliczane są pozostałe sumy i po kolei dodawane do multizbioru. Po zakończeniu działania pętli lista jest sortowana by nie podpowiadać algorytmowi poprawnego rozwiązania.

Na końcu działania generatora instancji inicjowana jest metoda dodająca błędy do instancji. Ilość błędów jest określana przez użytkownika. Na początku jest delecja, ponieważ w ten sposób unikana jest sytuacja, w której usunięte zostaną inne rodzaje dodanych błędów. Za pomocą pętli jest wybierana odpowiednia ilość elementów do usunięcia. Za pomocą *random.Next()* losowo wybierany jest element do usunięcia a następnie jest usuwany. Następne są błędy insercji. Błędy insercji są wstawiane jako drugie, ponieważ umożliwia to większy wybór dla błędów substytucji. Zasada działania jest podobna do błędów delecji. Losowo wybierana jest liczba z przedziału od 1 do największej liczby w multizbiorze. Ostatnim rodzajem błędu są błędy substytucji. Zasada działania jest podobna, losowany jest indeks liczby, która będzie podmieniana. Następnie losowana jest wartość na jaką zmieniona będzie

wybrana liczba. Jest ona ograniczona największym elementem z multizbioru. Po wybraniu obu wartości, liczba jest podmieniana. Lista z multizbiorem jest sortowana i wyświetlana w richTextBox. Jest to koniec działania generatora i można przejść do karty algorytmu.

3.3 Algorytm

Dla przejrzystości kodu, część algorytmu została umieszczona w osobnym pliku. Dane przekazywane do metaheurystyki są w postaci krotki. Najpierw tworzona jest krotka 'parameters' która zawiera po kolei: instancję, wielkość listy tabu, liczba progów pogarszania, maksymalna liczba iteracji bez poprawy i maksymalny czas działania programu. Po utworzeniu krotki włączany jest program w osobnym wątku za pomocą *algorytm.StartTask(parameters)*. Główna metoda *DoWork* zaczyna od odczytania przekazanych parametrów i przypisuje je do zmiennych wewnątrz głównej pętli. Tworzone są także pozostałe zmienne. Obliczanie zaczyna się od utworzenia rozwiązania początkowego. Rozpoczynana jest metoda, do której przekazywany jest multizbiór. Tworzony jest słownik, który grupuje elementy multizbioru według ich wartości. Kluczem jest wartość odległości a wartością jest liczba wystąpień tej odległości. Żeby wprowadzić generowanie losowe, klucze są losowo przemieszane. Następnie za pomocą zagnieżdżonych pętli *foreach* poszukiwane są takie dwie wartości, które spełniają warunki utworzenia rozwiązania początkowego, czyli pierwszy element ma wartość 0 a pozostałe dwie muszą tworzyć odległości, które występują w multizbiorze. Po znalezieniu takiego rozwiązania sprawdzane jest to czy takie rozwiązanie początkowe było już wykorzystywane wcześniej. Wywoływana jest osobna metoda, do której przekazywane jest potencjalne rozwiązanie początkowe. Jeśli nie znajduje się na zmiennej typu *HashSet* która przechowuje wszystkie dotychczas użyte rozwiązania początkowe to jest zwracana wartość *true*, wybrane rozwiązanie dodawane jest do pamięci i zwracane do dalszego rozwiązywania.

Rozwiązanie początkowe jest przypisywane do zmiennej aktualnego rozwiązania i do zmiennej najlepszego rozwiązania. Rozpoczyna się mierzenie czasu a następnie rozpoczyna się główna pętla algorytmu. Jest to pętla *while* która ma trzy warunki które muszą być spełnione by nadal działała: zmienna *iteracjeBezPoprawy* jest mniejsze niż maksymalna wartość określona przez

użytkownika, czas wykonywania musi być mniejszy niż górny limit określony przez użytkownika oraz kontrolka *e.Cancel* która działa w momencie przedwczesnego przerwania działania programu. Zaraz po inicjacji pętli głównej generowany jest kandydat. Wywoływana jest metoda *SelectBestCandidate* do której przekazywane jest aktualne rozwiązanie, instancja i maksymalna wielkość listy tabu. W funkcji *SelectBestCandidate* wywoływana jest na samym początku inna metoda *GenerateNeighbours* która generuje sąsiedztwo i przypisuje do zagnieżdżonej listy. Metoda *GenerateNeighbours* działa w taki sposób, że pętlą *for* generuje po kolei wszystkie wartości od 1 do największego elementu z instancji. Wygenerowany element sprawdzany jest pod kątem tego czy już występuje w rozwiązaniu, czy znajduje się na liście tabu ostatnio usuniętych elementów a jeśli spełni te warunki to wywoływana jest inna metoda sprawdzająca wszystkie odległości w wygenerowanym sąsiedztwie. Jeśli odległości się zgadzają czyli sąsiedztwo jest zgodne z zasadami dodawania elementów, sąsiedztwo jest dodawane do listy z wszystkimi pozostałymi sąsiedztwami. Lista jest zwracana z powrotem do funkcji *SelectBestCandidate*.

SelectBestCandidate wybiera które sąsiedztwo będzie ostatecznym kandydatem na podstawie przewidywania potencjału na dalszy rozwój tego rozwiązania. Generowane są potencjalne następne miejsca cięć dla danego sąsiedztwa a po tym generowane są wszystkie odległości. Wygrywa sąsiedztwo, które będzie miało największe wykorzystanie odległości czyli, jest szansa by dodać więcej elementów niż u pozostałych sąsiedztw. Liczba tych odległości jest liczona. Jeśli wartość jest większa niż dotychczas najlepszy kandydat to wartości są podmieniane. Następnie sprawdza czy w ogóle jakiekolwiek sąsiedztwo zostało znalezione. Jeśli zostało znalezione to aktualizowana jest lista tabu elementów dodanych. Wywoływana jest metoda która porównuje rozwiązanie dotychczasowe i nowego kandydata. Element którym się różnią jest dodawany do listy tabu. Tutaj też sprawdzana jest wielkość tabu. Jeśli wielkość będzie większa niż limit to najstarszy element jest usuwany. Ta sama metoda odpowiada za tablicę tabu elementów ostatnio dodanych i usuniętych i w obydwu przypadkach działa prawie tak samo. Po zaktualizowaniu listy tabu najlepszy kandydat zwracany jest do głównej pętli.

W głównej pętli sprawdzane jest to czy kandydat ma realną wartość lub czy nie jest pusty. Jeśli kandydat jest pusty wywoływana jest metoda mająca na celu wyjście z optimum. Wywoływana jest z zmiennymi takimi jak aktualne rozwianie, instancja, liczba progów pogarszania i referencja na liczbę usuniętych elementów. Jeśli rozwiązanie ma więcej elementów niż 1 to

wybierana jest ostatnia od końca wartość z rozwiązania która nie znajduje się na liście tabu zakazującej usunięcia i nie wynosi 0. Jeśli warunki zostaną spełnione to element jest usuwany, dodawany jest nowy element do listy tabu i inkrementowana jest wartość zmiennej liczącej usunięte elementy. Jeśli jednak liczba usuniętych elementów jest większa niż próg pogarszania to generowane jest nowe rozwiązanie początkowe które wykorzystuje tę samą funkcję co algorytm tabu dla pierwszego rozwiązania początkowego. Zerowana jest też zmienna licząca liczbę usuniętych elementów ponieważ zaczynamy od nowa. Rozwiązanie nieważne czy po usunięciu elementu czy nowo wygenerowane, zwracane jest to głównej pętli. W głównej pętli wykonywane jest polecenie *continue* i szukany jest nowy kandydat. Jeśli kandydat zostanie znaleziony i zwrócony do głównej pętli algorytmu. Sprawdzana jest liczba elementów aktualnego rozwiązania. Jeśli jest większa to sprawdzana jest łączna długość rozwiązania. Jeśli jest dłuższe to aktualne rozwiązanie staje się najlepszym rozwiązaniem. Jeśli jednak uzyskane rozwiązanie nie zawiera więcej elementów niż dotychczas najlepsze rozwiązanie, inkrementowana jest zmienna licząca iteracje bez poprawy. Następnie pętla wykonuje się po raz kolejny. Po całkowitym zakończeniu działania pętli czas jest zatrzymywany i razem z najlepszym rozwiązaniem przypisywany jest do *e.Result*. Wartości te są przekazywane do głównego pliku programu za pomocą delegata.

W głównym pliku programu wartości są odbierane, przetwarzane i wypisywane w *textBox*. Porównywane są ze sobą mapy wejściowe i wyjściowe pod względem ilości elementów. Obliczana jest funkcja celu i podawana jest w formie procentowej. Dzielona jest ilość elementów mapy wejściowej przez ilość elementów mapy wyjściowej. W taki sposób uzyskujemy funkcję celu i przejrzystą informację o ile rozwiązanie jest lepsze lub gorsze od przewidywanego. Jeśli mapa wejściowa miała 6 elementów a uzyskana posiada ich 3 to wartość funkcji celu wynosi 50%. Bardzo podobnie porównywane są także multizbiory. Sprawdzane jest użycie elementów w uzyskanym rozwiązaniu w porównaniu do oryginalnego multizbioru. Obliczane jest wykorzystanie elementów oraz są wypisane te wartości które nie zostały wykorzystane. Na koniec podawany jest czas działania algorytmu.

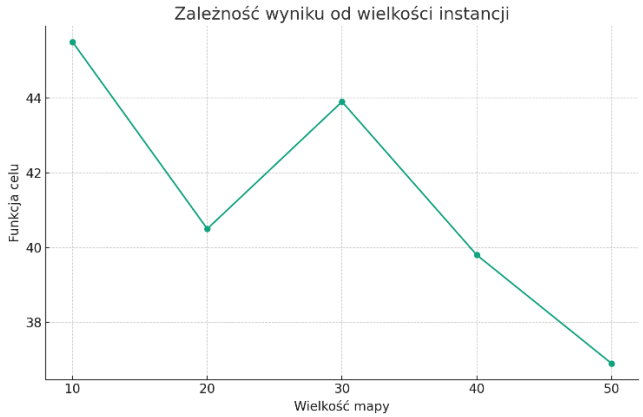
4. Testowanie

By uzyskać dane z testowe, przeprowadzono 1680 powtórzeń dla różnych danych i zmiennych. Najważniejsze były parametry generatora instancji. Przeprowadzono testy dla instancji o wielkości mapy od 10 do 50 z inkrementacją co 10. Większe instancje wykazały słabe wyniki i zostały porzucone. Dla parametru określającego wielkość elementów w zbiorze ustawiono jedną stałą wartość będącą tą samą wartością co wielkość mapy. W ten sposób zniwelowano parametr do testowania który nie ma realnie dużego wpływu na czas czy jakość rozwiązania. Dla parametrów określających błędy, dla każdego rodzaju błędu przetestowano 3 wartości: 0 i aby zunifikować sposób tworzenia danych, 1/5 i 1/2 wielkości mapy. Nigdy nie były testowane dwa typy błędów na raz ze względu na trudniej zauważalne wnioski. Następnie były dane do algorytmu tabu. Dla dwóch najważniejszych parametrów czyli wielkość listy tabu i próg pogarszania zostały zastosowane 2 różne wartości dla każdego parametru. Wielkość list tabu jest wielkością mapy pomnożoną razy 0,4 lub 0,7. Dzięki temu dostajemy ciekawy przedział by pokazać różnice. Rozmiar progu pogarszania jest obliczany podobnie do wielkości listy tabu jednak wartość elementów mapy jest pomnożona razy 0.2 i 0.5. Maksymalna ilość iteracji czy czas to zmienne które mają znaczenie w większych instancjach i wtedy gdy mamy ograniczony czas. W naszych testach były to stałe wartości. Maks iteracji bez poprawy był obliczany za pomocą pomnożenia razy 2 wielkości mapy natomiast czas był liczony razy 3. Są to bezpieczne wartości dla pozostałych parametrów i nie ograniczają zbyt mocno. Wszystkie wartości uzyskane w testach są zapisane w postaci pliku Excel. Najważniejszym wynikiem jaki otrzymywano z pojedynczych testów to wartość funkcji celu. Była ona obliczana jak podobne ilością elementów są rozwiązania oryginalne i otrzymane. Wynik podany jest za pomocą procentów. Znak ‘%’ musiał być usunięty ze względu na obliczenia średnich oraz tworzenie wykresów.

Dla każdych ustawień programu wykonano 10 testów by wyciągnąć wnioski z danych uśrednionych.

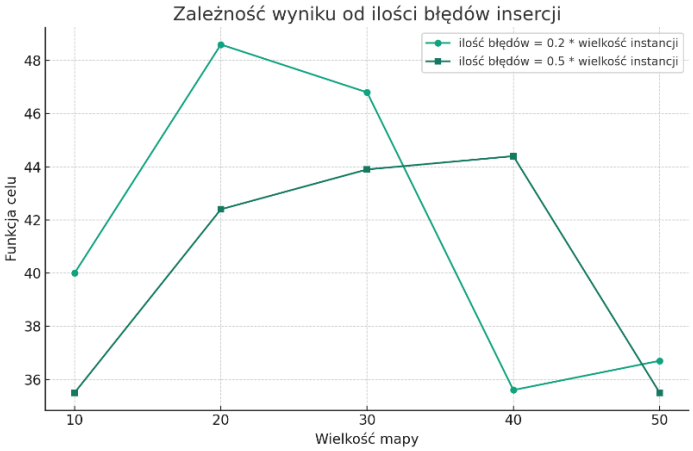
Rozmiar mapy

Jednym z najważniejszych czynników wpływających na jakość rozwiązania jest wielkość mapy. Dane zostały wybrane z najlepszymi parametrami generatora i instancji by zminimalizować wpływ innych czynników na zmianę wielkości instancji



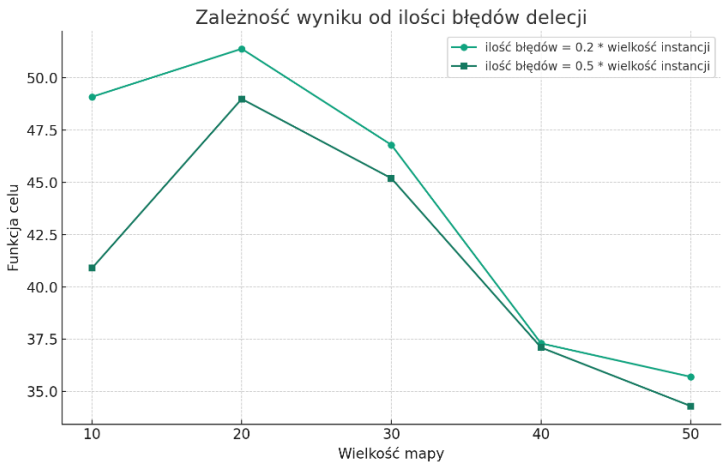
Oś x reprezentuje wielkość mapy natomiast oś y funkcję celu podana w procentach. Wykres pomimo niecodziennego wzrostu dla instancji o mapie wielkości 30 wykazuje spodziewane efekty. Im więcej elementów tym słabszy wynik uzyskanego rozwiązania a był on obliczany porównując liczbę elementów w mapie wejściowej do liczby elementów w mapie wyjściowej. Spadek dla wartości 30 może być spowodowany wygenerowaniem trudniejszych instancji.

Błędy insercji



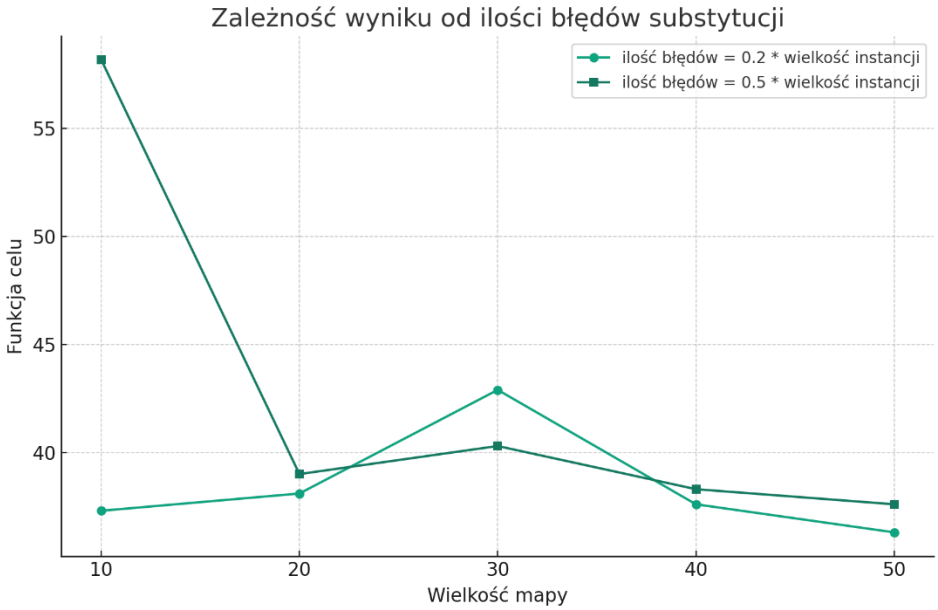
Oś x reprezentuje wielkość mapy a oś y funkcję celu przedstawioną za pomocą wartości procentowych. Błędy insercji porównywane są przez dwie linie wykresu, każda odpowiada innej wielkości błędów. Wnioski jakie można z tego wykresu wyciągnąć są następujące: ilość błędów jaka była wstawiona do małej instancji rzędu 10 jest zbyt duża. Algorytm musiał mieć zbyt dużą ilość błędnych odległości przez co spowodowało spadek jakości rozwiązania. Dalej wyniki są głównej mierze logiczne, im więcej elementów tym spada jakość rozwiązania. Instancje na poziomie 20-30 poradziły sobie najlepiej.

Błędy delecji



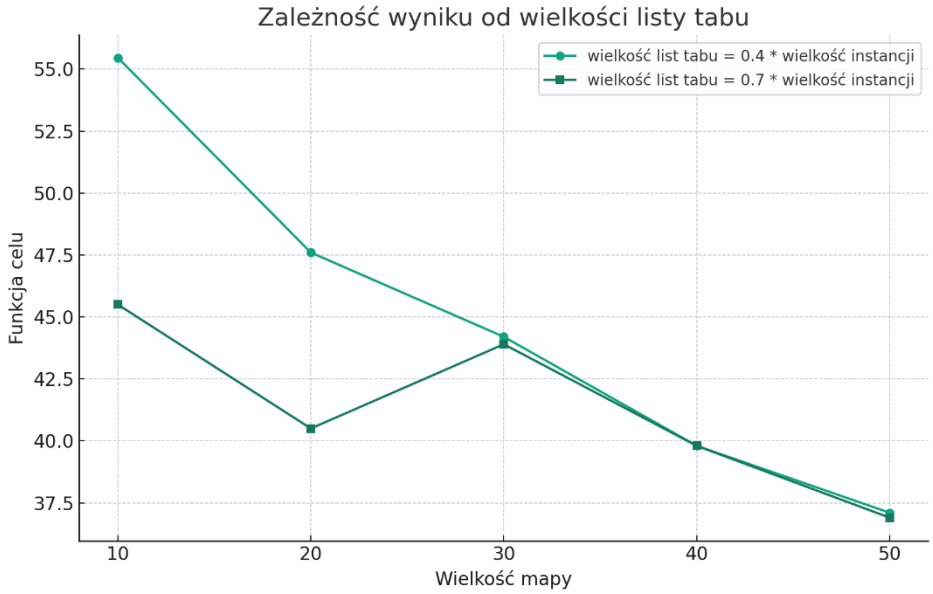
Sytuacja z instancją wielkości 10 się powtarza tak jak na poprzednich wykresach. Zastosowano zbyt dużą ilość błędów jak na taką wielkość instancji. Następne wielkości poradziły sobie znacznie lepiej i występuje przewidywany spadek jakości rozwiązania wraz z zwiększaniem się wielkości instancji.

Błędy substytucji



Na tym wykresie można zaobserwować dwie bardzo ciekawe sytuacje. Po pierwsze, podczas generowania instancji wejściowej dla mapy wielkości 10, błędy substytucji były w pewnym stopniu korzystne dla testów. Instancja z mapą wielkości 10 i z ilością błędów wynoszącą 5 uzyskała najlepszy wynik. Drugim ciekawym zjawiskiem jest mocne spłaszczenie wykresu dla pozostałych wielkości. Brak znaczącego spadku wraz z wzrostem wielkości mapy .

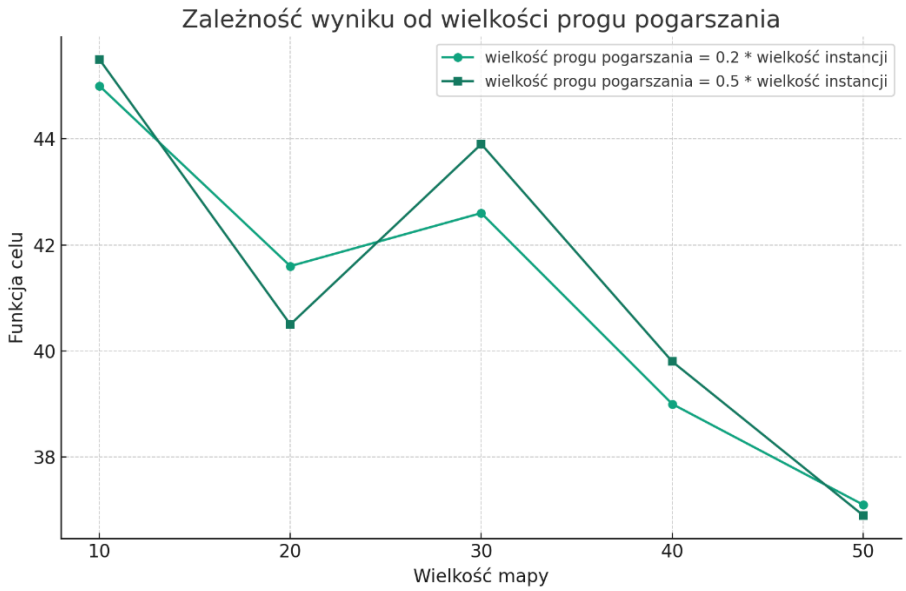
Wielkość listy tabu



Ten wykres prezentuje wyniki przefiltrowane pod kątem wielkości list tabu. Przede wszystkim rzuca się w oczy fakt, że duża lista tabu czyli silne ograniczenie działania algorytmu nie zawsze

działa na korzyść działania programu. Mocniejsze ograniczenie często zmusza program do przeszukiwania nowego rozwiązania początkowego jednak może to oznaczać, że algorytm porzuci rozwiązanie które jest poprawne tylko ze względu na zbyt dużą listę ruchów zabronionych.

Rozmiar Progu pogarszania



Jest to wykres zależny od rozmiaru progu pogarszania. Jest to zmienna która ogranicza jak wiele razy algorytm może usunąć wartość z rozwiązania zanim nie zostanie wywołana metoda generowania nowego rozwiązania. Porównując do pozostałych wykresów widać duże podobieństwo do wykresu pierwszego reprezentującego wielkość instancji w stosunku do jakości rozwiązania. Może to sugerować, że albo wielkość parametru została źle ustawiona lub ten parametr nie ma większego wpływu na wydajność algorytmu lub jakość rozwiązania.

5. Wnioski

W ramach projektu zaimplementowano program w środowisku C# Windows Forms, którego zadaniem było rozwiązanie problemu PDP (Partial Digest Problem) z wykorzystaniem algorytmu tabu. Zanim rozpoczęto etap programowania, konieczne było dogłębne zrozumienie zarówno samego problemu PDP, jak i mechanizmów stojących za algorytmem tabu. Znacząca część czasu poświęcona została na analizę teoretyczną, co umożliwiło późniejsze efektywne zaprojektowanie i implementację rozwiązania. Programowanie rozpoczęto od stworzenia interfejsu użytkownika, który ułatwia interakcję z algorytmem.

Kluczowym etapem projektu było przeprowadzenie testów, które były czasochłonne i wymagały szczegółowego planowania oraz analizy. Wykonano ponad 1600 testów, które miały na celu nie tylko weryfikację poprawności działania programu, ale także ocenę wydajności algorytmu tabu w różnych scenariuszach. Wyniki otrzymane z testów nie są całkowicie zadowalające. Przez dużą liczbę testów nie zauważyłem, że niektóre zmienne które uznałem za najlepsze wcale nie zwracały idealnych wyników. Przykładem jest lista tabu gdzie przeceniłem jak duża taka lista powinna być. Sam program też nie działa idealnie. Należy przejrzeć sposób szukania nowego kandydata i dodać ograniczenia które pomogą uzyskiwać lepsze wyniki. Pomimo świadomości błędów w danych testowych oraz faktu, że algorytmowi brakuje większej dokładności, widzę pole do poprawy a na ten moment jakość zwracanych wyników przy odpowiednich parametrach oceniam na zadowalający.

6. Bibliografia

Marta Kasprzak's website: https://www.cs.put.poznan.pl/mkasprzak/zp/zp.html

Wikipedia, Metaheuristics: https://en.wikipedia.org/wiki/Metaheuristic