POLITECHNIKA WROCŁAWSKA

DATA ZŁOŻENIA PROJEKTU : 27.01.2022r.

Termin zajęć: czwartek, 15:15, TN/TP

Porównanie algorytmów Branch&Bound, TabuSearch, symulowane wyżarzanie, algorytm genetyczny dla problemu komiwojażera w kontekście optymalizacji czasu robienia zakupów w sklepie.

SWDiSK

Prowadzący: dr inż. Wojciech Kmiecik

Grupa:

Daniel Król (241399)

Konrad Prusak (242038)

Bartosz Rodziewicz (226105)

1. Wprowadzenie

Niniejszy projekt zakłada porównanie algorytmów Branch&Bound, TabuSearch, symulowanego wyżarzania i algorytmu genetycznego dla rzeczywistego przykładu problemu komiwojażera zaobserwowanego w życiu codziennym. Problemem tym jest optymalizacja czasu robienia zakupów w sklepie.

Celem projektu jest porównanie algorytmów znajdowania optymalnej ścieżki robienia typowych, codziennych zakupów w supermarkecie. Nikt nie lubi marnowania czasu - zwłaszcza w dzisiejszym świecie, gdy większość ludzi żyje w pędzie. Warto więc zadbać o to, by ograniczyć czas zakupów do niezbędnego minimum.

Do znajdowania optymalnej ścieżki wykorzystano algorytmy rozwiązujące problem komiwojażera. Zmieniano parametry algorytmów w celu znalezienia najlepszej kombinacji. Następnie porównano wyniki i ustalono, który z nich sprawdza się najlepiej.

2. Przegląd literatury

2.1. Problem komiwojażera

W projekcie pochylono się nad problemem komiwojażera [1] (ang. *travelling salesman problem*). Komiwojażer wyrusza z miasta rodzinnego do innych miast. Chce on odwiedzić każde miasto tylko jeden raz, a następnie wrócić do domu. Przy okazji, chce zrobić to jak najszybciej.

Problem ten można zobrazować także posługując się teorią grafów -miasta są wierzchołkami grafu, a trasy pomiędzy nimi to krawędzie z wagami. Waga krawędzi może odpowiadać odległości pomiędzy miastami połączonymi tą krawędzią, czasowi podróży lub kosztom przejazdu - zależy, co chcemy w podróżny komiwojażera zminimalizować - dążymy do tego, by waga trasy (suma wag krawędzi) była minimalna. Trasa komiwojażera jest cyklem przechodzącym przez każdy wierzchołek grafu dokładnie jeden raz - jest to zatem cykl Hamiltona.

Aby uzyskać prawdziwie optymalną trasę trzeba by znaleźć wszystkie możliwe cykle Hamiltona, obliczyć ich wagi i wybrać tę o najmniejszej wadze. Przy rozległych grafach jest to bardzo trudne. Istnieją jednak algorytmy pozwalające na znalezienie trasy przybliżonej do optymalnej.

2.2. Branch&Bound

W rozwiązywaniu problemu komiwojażera można zastosować algorytm Branch&Bound[2] (metoda podziałów i ograniczeń). Algorytm ten polega na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu. W przeciwieństwie do algorytmu Brute Force przeszukiwanie nie odbywa się całkowicie na ślepo. Znajdując się na każdym

wierzchołku drzewa rozwiązań algorytm wylicza najlepszy wynik jaki uzyska sprawdzając poddrzewa każdego ze swoich potomków. Rozgałęzianie (ang. *branching*) to dzielenie zbioru rozwiązań reprezentowanego przez węzeł na rozłączne podzbiory, reprezentowane przez jego podwęzły. Natomiast ograniczanie (ang. bounding) polega na pomijaniu w przeszukiwaniu tych gałęzi drzewa, o których wiadomo, że nie zawierają optymalnego rozwiązania w swoich liściach.

2.3. TabuSearch

Do rozwiązania problemu komiwojażera można zastosować także algorytm TabuSearch[3]. W algorytmie tym decyzja o kolejnym ruchu jest podejmowana na podstawie funkcji wartości ruchu. Nie istnieją ogólne zalecenia co do jej konstrukcji. Algorytm ten, jest tak naprawdę ideą i jego implementacje mogą znacząco się różnić od siebie.

Główny zarys tego algorytmu wygląda następująco:

- Algorytm wyznacza w jakiś sposób początkowe rozwiązanie,
- Algorytm przeszukuje otoczenie tego rozwiązania (otoczeniem jest zbiór innych rozwiązań różniących się od obecnego rozwiązanie o jeden Ruch; Ruch, jak i otoczenie definiowane jest poprzez implementacje),
- Spośród wszystkich rozwiązań w danym otoczeniu algorytm wybiera najlepsze i Ruch dodawany jest do listy zabronionych ruchów na jakiś okres kadencji (czas, bądź liczba iteracji),
- Algorytm kończy pracę, gdy nastąpi spełnienie warunku satysfakcjonującego (czas, bądź liczba iteracji), zwraca wtedy najlepsze dotąd znalezione rozwiązanie.

Algorytm w żaden sposób nie gwarantuje znalezienie najbardziej optymalnego rozwiązania, jednak znacząco skraca czas wyznaczania w miarę optymalnego rozwiązania (użytkownik decyduje ile czasu algorytm pracuje).

Jeśli dla danej instancji problemu nie jest znane rozwiązanie optymalne, nie ma też żadnej możliwości aby oszacować stopnia błędu znalezionego przez algorytm rozwiązania.

Algorytm zakłada istnienie lista tabu, która służy zabezpieczeniu algorytmu przed zablokowaniem się w okolicy jednego lokalnego optimum. Celem zwiększenia szansy znalezienia globalnego optimum w algorytmie stosuje się różne metody dywersyfikacji poszukiwań.

2.4. Symulowane wyżarzanie

Symulowane wyżarzanie[4] to modyfikacja algorytmu zachłannego dodająca do niego losowość. Nie przegląda pełnego sąsiedztwa bieżącego rozwiązania, zamiast tego kolejne rozwiązanie wybierane jest losowo, na bazie prawdopodobieństwa.Prawdopodobieństwo obliczane jest według formuły.

Na początku zadana jest temperatura Tmax, zmniejszająca się o określoną liczbę iteracji, oraz Tmin, której przekroczenie oznacza koniec algorytmu (ewentualnie można ustalić

maksymalną liczbę iteracji). Im wyższa temperatura, tym większe prawdopodobieństwo (wyrażone rozkładem Boltzmana) na przyjęcie gorszego rozwiązania, co pozwala wychodzić z minimów lokalnych.

2.5. Algorytm genetyczny

Algorytm genetyczny[5] to jeden z algorytmów populacyjnych. W odróżnieniu od metody Tabu Search algorytm ten jest o wiele bardziej sprecyzowany. Jego zasada działania została zaczerpnięta z biologii.
Ogólny zarys działania algorytmu prezentuje się następująco:

- Na start losowana jest początkowa populacja,
- Populacja zostaje poddana ocenie jakości i najsłabsze osobniki są odrzucane (tak, aby populacja miała stałą, z góry ustaloną, wielkość),
- Osobniki są poddawane działaniu operatorów ewolucyjnych:
    - Nowe osobniki są tworzone poprzez odpowiednie łączenie genotypów rodziców (krzyżowanie),
    - Osobniki są poddawane działaniu czynnika losowego (mutacja) delikatnie zmieniającego genotyp,
- Nowe osobniki (pokolenie) dodawane jest do ogólnej populacji. Populacja zostaje poddana ocenie i jeśli rozwiązanie spełnia wymaganą jakość algorytm zostaje przerwany, albo kontynuowany jest proces reprodukcji.

Algorytm ten w żaden sposób nie gwarantuje znalezienie najbardziej optymalnego rozwiązania, jednak znacząco skraca czas wyznaczania w miarę optymalnego rozwiązania (użytkownik decyduje ile czasu algorytm pracuje). Jeśli dla danej instancji problemu nie jest znane rozwiązanie optymalne, nie ma też żadnej możliwości aby oszacować stopnia błędu znalezionego przez algorytm rozwiązania.

Algorytm ten, od innych metod heurystycznych, wyróżnia:

- przetwarzanie populacji rozwiązań, prowadzące do równoległego przeszukiwania przestrzeni rozwiązań z różnych punktów,
- w celu ukierunkowania procesu przeszukiwania wystarczającą informacją jest jakość aktualnych rozwiązań,
- celowe wprowadzenie elementów losowych.

Ponieważ algorytm genetyczny jest algorytmem niedeterministycznym, nie można dla niego w całości określić czasowej złożoności obliczeniowej. Można jednak podać złożoność obliczeniową pojedynczego przeglądu całej populacji w celu reprodukcji, która wynosi $O(n!)$, gdzie n oznacza wielkość populacji.

3. Opis problemu badawczego

3.1. Źródło problemu

Problem komiwojażera to zagadnienie optymalizacyjne, którego początki sięgają XIXw. Znany jest więc już od dawna. Jego charakterystyka została opisana w punkcie drugim na podstawie odpowiedniego artykułu. W skrócie - polega on na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Cykl Hamiltona to taki cykl w grafie, w którym każdy wierzchołek grafu odwiedzany jest dokładnie raz (plus powrót do wierzchołka początkowego). Jest to problem NP-trudny.

Główną trudnością problemu jest duża liczba danych do analizy. W przypadku symetrycznego problemu komiwojażera dla *n* miast liczba kombinacji wynosi $(n-1)!/2$. Dla 20 miast będzie to 6×10^{16}. Jak widać, sprawdzenie wszystkich kombinacji po kolei jest praktycznie niemożliwe. Testowane algorytmy pozwolą na znalezienie rozwiązania bliskiego optymalnemu.

3.2. Dane wejściowe

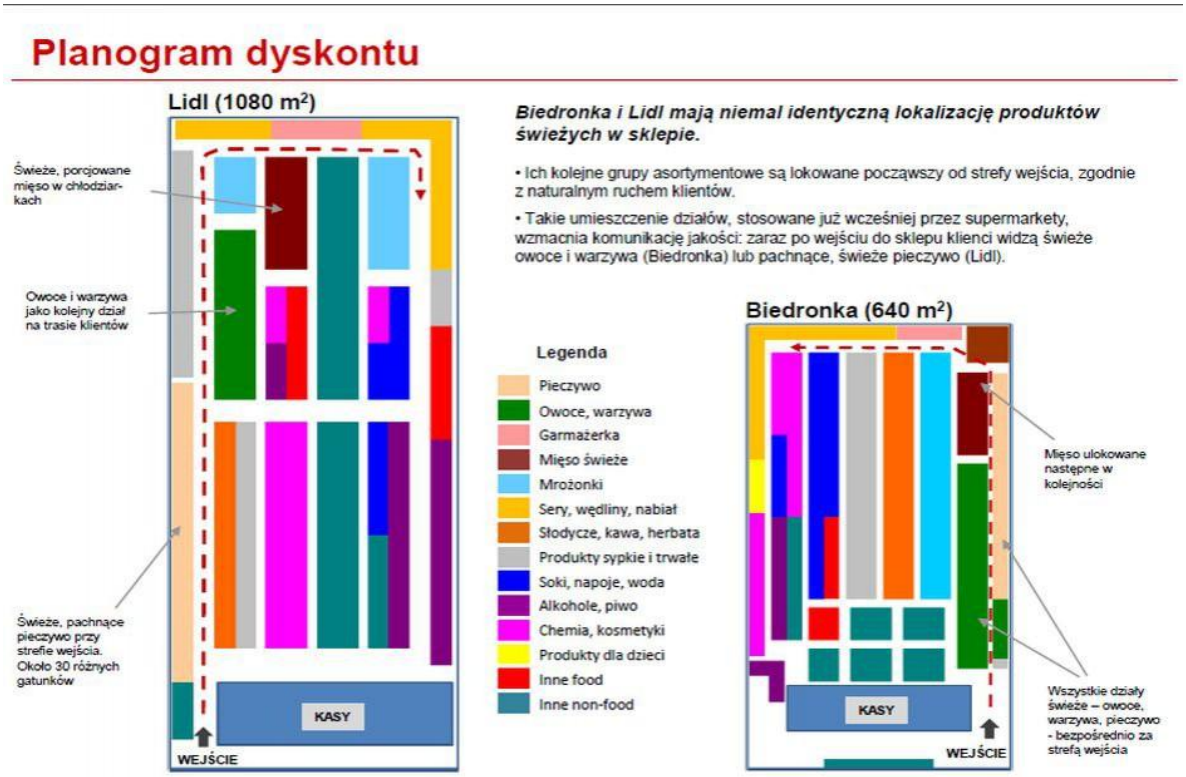
Zestaw danych został stworzony na podstawie kilku niezależnych źródeł. Przykładowe listy zakupów zaczerpnięte z internetu (np. ze strony lisotonic.com).

Listy zakupów były modyfikowane między sobą do stworzenia nowych list zakupów. Produkty zawarte w tych listach należą do różnych kategorii sklepowych. Możliwe umiejscowienie kategorii w budynkach sklepów zostało zaczerpnięte z planów budynków kilku marketów (rys. 1, 2, 3).

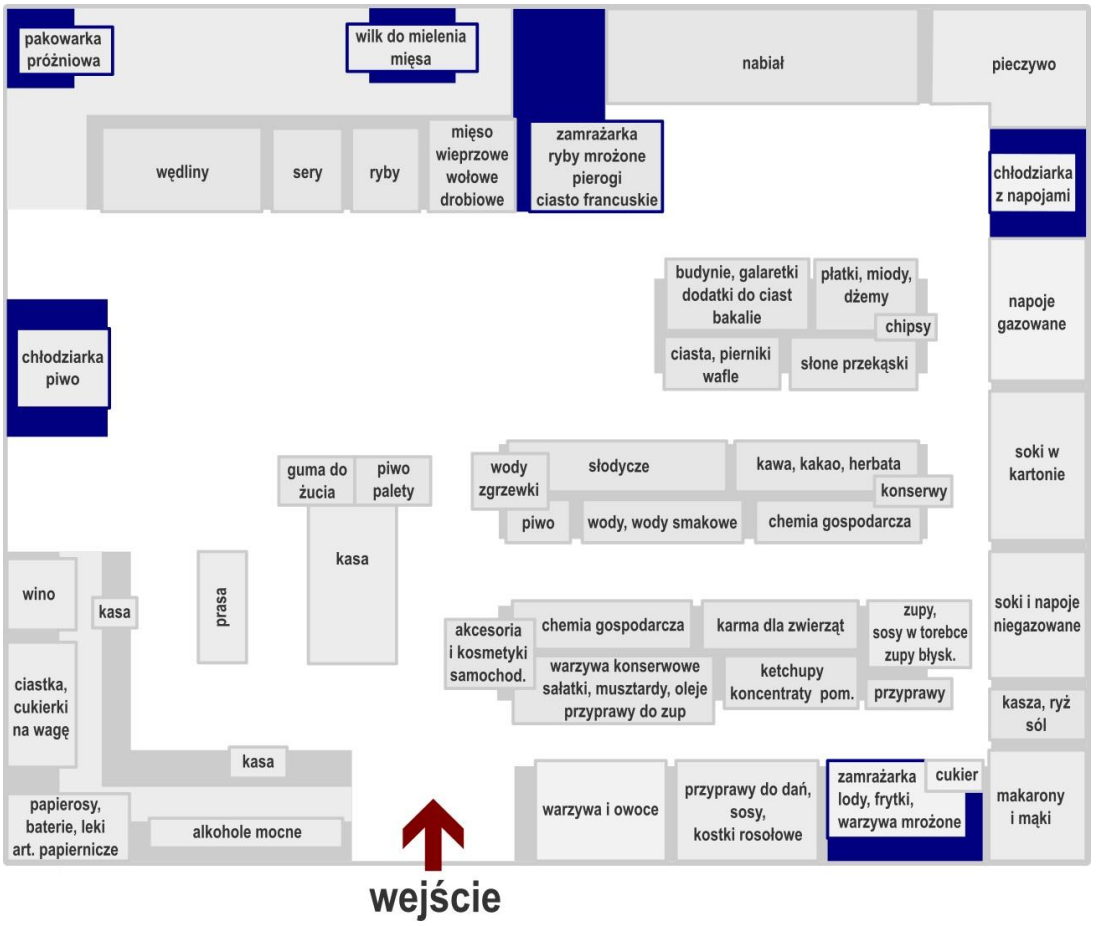
Na podstawie wymienionych danych - list zakupów, a także położenia poszczególnych kategorii produktowych w sklepach - za pomocą programu Excel zostały stworzone instancje problemu odpowiadające realnym przypadkom. Instancje te zostały zapisane w plikach o nazwach skojarzonych z rozmiarem zestawu: 12-1.txt, 12-2.txt, 17.txt, 24.txt, 30.txt oraz 50.txt.

Plik koniecznie musi być zapisany w konkretnym formacie:

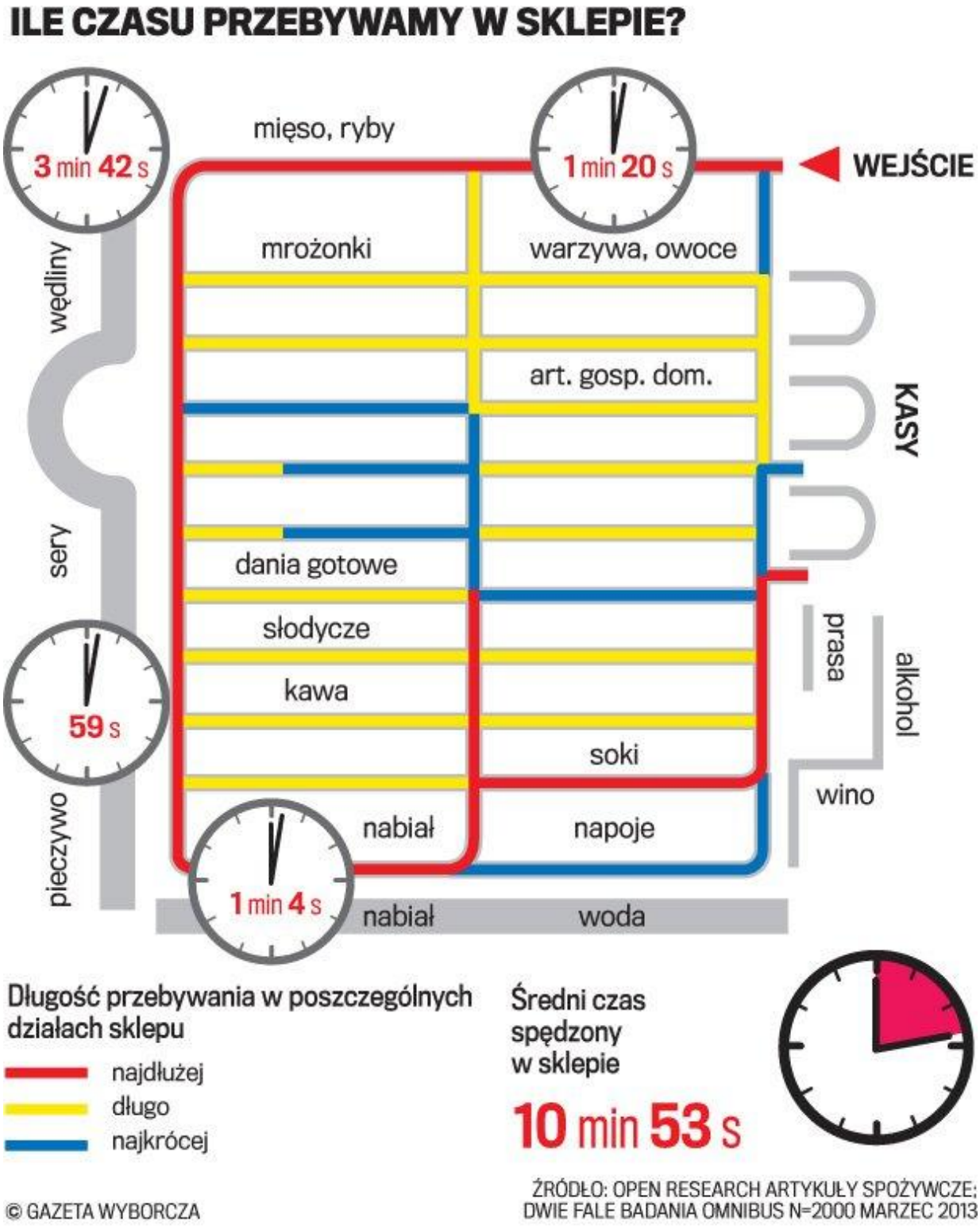
- Pierwsza linia to pojedyncza liczba (całkowita, dodatnia, różna od zera) oznaczająca ilość miast.
- Kolejne linieto macierz X na X (X = liczba z pierwszej linijki), kolejne liczby w linijce oddzielone spacja, zawierające odległości pomiędzy miastami (całkowite, dodatnie, różne od zera). Przekątna macierzy zawiera liczby -1.
- Zawartość przykładowego pliku:
  3
  -1 10 15
  20 -1 5
  25 30 -1



Rys. 1 - Plany sklepów Lidl i Biedronka (link)



Rys. 2 - Sklep sieci Lewiatan o nazwie “Szarek2” (link)



Rys. 3 - Sklep użyty w badaniu Omnibus w 2013r. (link)

3.3. Wskaźniki oceny jakości

Jakość algorytmów została oceniona według następujących wskaźników:

- czas znalezienia rozwiązania,
- jakość (odległość od optimum wyrażona w % lub porównanie do wyników innych algorytmów w przypadku nieznanego optimum).

3.4. Ograniczenia

Ograniczenia napotkane podczas testowania:

- ustalone odgórne ograniczenia czasu (lub liczby iteracji) do wykonywania algorytmu,
- ustalony czas działania zależnego od jakości rozwiązania (np. wiemy że najlepsze znane rozwiązanie wynosi X, więc szukajmy do momentu, gdy znajdziemy rozwiązanie lepsze od X),
- maksymalna zajętość pamięciowa podczas pracy algorytmów.

3.5. Funkcje kryterialne

Funkcje kryterialne do oceny i porównywania rozwiązań opierały się na kosztach rozwiązań. Rozwiązanie, które miało mniejszą wartość liczbową było wybierane jako korzystniejsze. Funkcja celu minimalizowała wartość długości drogi, czyli koszt rozwiązania. Ogólny schemat działania funkcji kryterialnej dla algorytmów:

```
czyPierwszeRozwiazanieKorzystniejsze(int koszt1, int koszt2)
       return koszt1 < koszt2
```

3.6. Dane wyjściowe

Rozwiązaniami algorytmów były dwie wartości:

- pierwsza to wartość liczbowa reprezentująca koszt najlepszego znalezionego rozwiązania.
- druga to kolejność elementów odpowiadająca najlepszemu znalezionemu rozwiązaniu

Dodatkowo dla każdego algorytmu mierzone były czasy działania, które były różnicą czasu pomiędzy rozpoczęciem szukania rozwiązania dla danej instancji problemu a osiągnięciem warunku końcowego.

4. Wybrane algorytmy

Ogólne założenia i zasady działania algorytmów zostały opisane w punkcie drugim zawierającym przegląd literatury. Poniżej scharakteryzowano wyłącznie nasze implementacje wykorzystanych algorytmów.

4.1. Branch&Bound

Znajdując się na każdym wierzchołku drzewa rozwiązań algorytm Branch&Bound wylicza najlepszy wynik jaki uzyska sprawdzając poddrzewa każdego ze swoich potomków. Uwzględniana jest aktualna długość trasy od wierzchołka startowego, wynik funkcji liczącej dolne ograniczenie oraz najlepsze znalezione do teraz rozwiązanie. W całym algorytmie najważniejsza jest funkcja licząca dolną granicę. Zależy ona bezpośrednio od specyfiki problemu i jej dokładność wpływa na czas obliczeń.

W naszym rozwiązaniu funkcja ta jest połączona z minimalnym kosztem odwiedzenia każdego wierzchołka. Możemy zauważyć, że do każdego wierzchołka wchodzi i wychodzi się dokładnie raz, dlatego:

- dla każdej krawędzi połowę jej wagi kojarzymy z wierzchołkiem z którego wychodzi, natomiast drugą połowę z wierzchołkiem do którego wchodzi,
- dlatego dla każdego wierzchołka obliczamy minimalny koszt jego odwiedzenia jako sumę połowy minimalnych kosztów wejścia i wyjścia z niego.

Na początku działania algorytmu liczymy dolne ograniczenie jako sumę minimalnych kosztów odwiedzenia każdego wierzchołka, czyli:

$$(min(wagaKrawedziWchodzacej) + min(wagaKrawedziWychodzacej)) / 2$$

Zaczynamy też liczyć koszt już przebytej trasy. Przy analizie potomków wierzchołka, na którym się znajdujemy do przebytej trasy dodajemy drogę do tego potomka, a z dolnego ograniczenia odejmujemy połowę minimalnego kosztu wyjścia z wierzchołka, w którym jesteśmy i połowę minimalnego kosztu wejścia do wierzchołka będącego potomkiem.

Następnie sumę aktualnie przebytej trasy i dolnego ograniczenia porównujemy z najlepszym znalezionym rozwiązaniem.

Jeśli aktualnie znalezione rozwiązanie jest lepsze od przewidywanego najlepszego jakie można osiągnąć idąc tą gałęzią, gałąź "odcinamy" i bez sprawdzania jej przechodzimy do sprawdzenia kolejnego potomka. takie odcinanie gałęzi powoduje często spore zaoszczędzenie obliczeń w wielu przypadkach. Nigdy nie mamy jednak gwarancji, że uda nam się ominąć dużą liczbę tych poddrzew i w najgorszym przypadku algorytm ten może zdegradować się do przeglądu zupełnego.

W założeniach projektowych chcieliśmy posłużyć się algorytmem Branch&Bound jako benchmarkiem. Ze względu na specyfikę powinien on zawsze znajdować optymalne rozwiązanie.
Parametry sterujące:

- brak

Pseudokod:

```
int wierzcholekStartowy = 0

void sprawdzRozwiazania(int wierzcholek = wierzcholekStartowy) {
    droga.insert(wierzcholek);
    if (droga.size() < iloscMiast) {
        for (miasto in range(0, iloscMiast)) {
            if (not droga.contains(miasto)) {
                if (dystans(droga) + dolneOgraniczenie(miasto) < najlepszeRozwiazanie) {
                    sprawdzRozwiazania(miasto)
                }
            }
        }
    } else {
        droga.insert(wierzcholekStartowy)
        if (dystans(droga) < najlepszeRozwiazanie) {
            najlepszeRozwiazanie = dystans(droga)
        }
    }
}
```

4.2. TabuSearch

Nasza implementacja bazuje na następujących cechach:

- Początkowe rozwiązanie znajdowane jest za pomocą zachłannego algorytmu najbliższego sąsiada (nearest neighbor algorithm). Statystycznie algorytm ten znajduje rozwiązanie o 25% dłuższe niż najlepsze rozwiązanie przy całkowicie losowym rozłożeniu miast.
- Ruchem w naszym algorytmie jest wymiana dwóch miast ze sobą w trasie (algorytm zakłada spójność grafu (asymetryczność jest dozwolona)).
- Otoczeniem rozwiązania (otoczenie typu swap) jest więc każde rozwiązanie różniące się od obecnego pozycją dwóch miast (złożoność obliczeniowa jednej iteracji to O(n^2)).
- Na liście tabu przechowywane są pary wierzchołków zakazując ich ponownej wymiany przez ustaloną przez użytkownika ilość iteracji (domyślnie 0.5 * ilośćWierzchołków).
- Zaimplementowane kryterium aspiracji (opcjonalne, domyślnie włączone) pozwala na wykonanie ruchu znajdujące się na liście tabu jeśli prowadzi ono do poprawienia najlepszego rozwiązania dotychczas znalezionego.
- Strategią dywersyfikacji (opcjonalną, domyślnie włączoną) jest wyznaczenie nowego, całkowicie losowego, rozwiązania po określonej liczbie iteracji (domyślnie 10 tys.) bez poprawy najlepszego znalezionego rozwiązania.
- Kryterium satysfakcji jest czas pracy algorytmu po którym następuje przerwanie pracy algorytmu (domyślnie 10(s)). Czas pracy algorytmu sprawdzany jest po każdym przeszukaniu sąsiedztwa, więc nie jest całkowicie dokładny. Średnio algorytm wykonuje się 10% dłużej niż podany przez użytkownika czas.

Ponieważ algorytm oparty na metodzie Tabu Search jest algorytmem niedeterministycznym, nie można dla niego w całości określić czasowej złożoności obliczeniowej. Można jednak podać złożoność obliczeniową pojedynczego przeglądu całego sąsiedztwa, która dla sąsiedztwa typu swap wynosi O(n^2)

Parametry sterujące:

- Kadencja (np. 0.5 * ilość miast) - parametr określający jak długo dana para miast znajduje się na liście tabu, czyli jak długo zamiana tej pary jest zakazana; uzależnienie długości kadencji od ilości miast, powoduje, że algorytm lepiej sprawować się będzie dla różnych wielkości instancji problemu, bez zmiany parametru
- Aspiracja (tak/nie) - parametr określający czy stosować kryterium aspiracji, czyli czy dopuszczalne jest wykonania ruchu z listy tabu, jeśli taki ruch poprawia najlepsze znane rozwiązanie
- Dywersyfikacja (tak/nie) - parametr określający, czy stosować strategie dywersyfikacji, czyli czy jeśli przez X ruchów nie zostanie znalezione lepsze rozwiązanie, to należy dokonać wyszukania innej drogi, wokół której algorytm zacznie swoją pracę na nowo; strategia dywersyfikacji pomaga opuszczać minima lokalne
- Domyślna ilość iteracji bez poprawy do dywersyfikacji (np. 10 000) - parametr określający jak często stosować strategię dywersyfikacji, czyli co ile ruchów bez poprawy rozwiązania należy wygenerować nową drogę

Pseudokod:

```
while (czasNiePrzekroczony) {
    sprawdzSasiednieRozwiazania()
    wykonajNiezakazanyRuchDajacyNajlepszeSasiednieRozwiazanie()
    if (dystans(droga) < najlepszeRozwiazanie) {
        najlepszeRozwiazanie = dystans(droga)
    }
    dodajRuchDoTabu()
    sprawdzAspiracje()
    sprawdzDywersyfikacje()
}
```

4.3. Symulowane wyżarzanie

Rozwiązanie początkowe to kolejność naturalna (np. 1, 2, 3). Kolejne rozwiązania mogą być wybierane na dwa sposoby:

- przez zamianę - dwa elementy są losowane, a następnie zamieniane miejscami oraz wstawiane w odwrotnej kolejności do pierwotnego rozwiązania
- przez odwrócenie - dwa elementy są losowane, a następnie wszystkie elementy od pierwszego wylosowanego elementu do drugiego wylosowanego elementu są odwracane kolejnością (np. 1, 2, 3, 4, 5, 6, 7 -> wylosowano 2 i 5 -> 1, 5, 4, 3, 2, 6, 7). Tak odwrócony podzbiór rozwiązania jest wstawiany do pierwotnego rozwiązania.

Liczba losowa pomiędzy 0 a 1 jest wybierana na podstawie rozkładu normalnego, aby porównać z nią funkcję oceny rozwiązania zależącą od obecnej temperatury i kosztów wygenerowanego rozwiązania i jego zmodyfikowanego sąsiada.
Parametry sterujące:

- temperatura początkowa (np. 1000) - maksymalna temperatura z jaką algorytm zaczyna oceniać rozwiązania w kontekście wybrania gorszego rozwiązania. W trakcie

postępowania algorytmu maleje. Im jest większa, tym algorytm częściej będzie wybierał gorsze rozwiązania.
- temperatura końcowa (np. 0,01) - temperatura warunkująca moment końca działania algorytmu, gdy temperatura algorytmu osiągnie temperaturę końcową. Im niższa temperatura końcowa, tym częściej algorytm będzie mógł pozostawać w obszarach minimów lokalnych.
- współczynnik stygnięcia alpha (np. 0,999) - współczynnik definiujący szybkość obniżania temperatury poprzez mnożenie obecnej temperatury przez alpha. Im ta liczba jest większa, tym wolniej algorytm będzie obniżał swoją temperaturę i wykona więcej powtórzeń zewnętrznej pętli.
- liczba powtórzeń dla jednej temperatury (np. 25) - liczba określająca ile razy algorytm powinien próbować dokonać modyfikacji obecnego rozwiązania bez zmiany temperatury (więc zachowując to samo prawdopodobieństwo wybrania gorszego rozwiązania).

Pseudokod:

```
najlepszyKosztGlobalny = INT_MAX
najlepszaKolejnosc
x = losowe rozwiązanie (np. porządek naturalny)
while (temperatura > temperaturaKońcowa
        while (numerPowtórzenia < liczbaPowtórzeńDlaJednejTemperatury)
                y = stwórzNoweRozwiązanieNaPodstawieObecnego(x)
                yKoszt = obliczKoszt(y)
                if (yKoszt < najlepszyKosztGlobalny)
                        najlepszyKosztGlobalny = yKoszt
                        najlepszaKolejnosc = y.kolejnosc
                if (yKoszt < obliczKoszt(x) )
                        x = y
                else if (exp ((obliczKoszt (x) - yKoszt) / temperatura)) >losowaLiczbaOd0Do1())
                        x = y
                numerPowtórzenia++
        temperatura = temperatura * wspolczynnikStygnieciaAlpha
return najlepszyKosztGlobalny, najlepszaKolejnosc
```

4.4. Algorytm genetyczny

Nasza implementacja charakteryzuje się następującymi cechami:
- Genotypem osobnika jest trasa komiwojażera,
- Fenotypem jest długość tej trasy,
- Populacja startowa jest generowana całkowicie losowo,
- Wielkość populacji jest parametrem algorytmu, domyślnie 50 osobników,
- Kryterium spełnienia jest długość pracy algorytmu (również parametr; domyślnie 30 sekund),
- Zastosowaliśmy algorytm krzyżowania OX,
- Zaimplementowane są dwa rodzaje mutacji - zamiana wierzchołków albo krawędzi (parametr algorytmu; domyślnie wierzchołków),
- W przypadku krzyżowania, jak i mutacji w algorytmie są dwa parametry:
    - Współczynnik krzyżowania - prawdopodobieństwo, że dwa osobniki zostaną poddane reprodukcji i stworzą dwa nowe (domyślnie 0.8 - 80%),

    - Współczynnik mutacji - prawdopodobieństwo, że na osobniku zajdzie mutacja (domyślnie 0.01 - 1%).

Parametry sterujące:

- Populacja (np. 50) - ilość osobników w populacji, których używamy do krzyżowania, jest to również wielkość startowej populacji, oraz liczba najlepszych osobników, które są brane do kolejnej rundy krzyżowania
- Współczynnik krzyżowania (np. 0.80) - prawdopodobieństwo z jakim z pary dwóch osobników powstanie potomek
- Rodzaj mutacji (wierzchołkowa/krawędziowa) - rodzaj mutacji jakiej potencjalnie mogą być poddani osobniki z nowego pokolenia, zamiana wierzchołków oznacza wymianę dwóch wierzchołków w genotypie, zamiana krawędzi oznacza wymianę dwóch krawędzi (w praktyce dwóch par wierzchołków) w genotypie
- Współczynnik mutacji - prawdopodobieństwo z jakim osobnik nowego pokolenia zostanie poddany mutacji

Pseudokod:

```
populacja = losujStartowaPopulacje()

while (czasNiePrzekroczony) {
    nowaGeneracja = krzyzujPopulacje(prawdopodobienstwoKrzyzowania)
    for (osobnik in nowaGeneracja) {
        mutuj(osobnik, prawdopodobienstwoMutacji)
    }
    populacja.insert(nowaGeneracja)
    odrzucNajslabszeOsobniki(populacja)
}
```

5. Opis opracowanego symulatora

Stworzony przez nas symulator jest programem działającym w terminalu (CLI). Napisany został w języku C++ z wykorzystaniem funkcjonalności języka C++14 oraz elementów biblioteki standardowej.
Program umożliwia:

- wczytanie z pliku grafu reprezentującego instancję problemu, plik koniecznie musi być zapisany w konkretnym formacie:
    - Pierwsza linia to pojedyncza liczba (całkowita, dodatnia, różna od zera) oznaczająca ilość miast.
    - Kolejne linie to macierz X na X (X = liczba z pierwszej linijki), kolejne liczby w linijce oddzielone spacja, zawierające odległości pomiędzy miastami (całkowite, dodatnie, różne od zera). Przekątna macierzy zawiera liczby -1.
    - Zawartość przykładowego pliku:
      3
      -1 10 15

20 -1 5
25 30 -1

- Wygenerowanie losowej instancji problemu o zadanym rozmiarze (ilość miast) oraz maksymalnej drodze pomiędzy dwoma miastami (każda odległość w grafie będzie znajdowała się w przedziale od 0 do podanej wartości maksymalnej)
- Uruchomienie algorytmów na wczytanej/wygenerowanej instancji problemu:
    - Branch and bound
    - TabuSearch
    - genetyczny
    - symulowanego wyżarzania
- Wybranie wartości parametrów dla algorytmów:
    - TabuSearch
    - genetycznego
- Wykonanie zadanych benchmarków algorytmów na różnych instancjach problemu komiwojażera

Wybór wartości parametrów algorytmu symulowanego wyżarzania jest możliwy z poziomu kodu z powodu ograniczonego czasu na wykonanie projektu.
Obsługa programu polega na wybraniu z klawiatury odpowiednich pozycji w menu lub podaniu wartości o które program pyta. Większość pozycji w menu posiada opisy wyjaśniające co dana rzecz robi lub co można w danym miejscu zrobić.

6. Plan badań

Przed przystąpieniem do badań przeprowadzono proces „tuningu" algorytmów heurystycznych. Nie wszystkie parametry zostały wzięte pod uwagę podczas tworzenia scenariuszy testowych. Niektóre z nich zawsze dają lepsze wyniki (np. włączenie parametrów aspiracja oraz dywersyfikacja w TabuSearch), a niektóre z nich mają praktycznie zerowy wpływ na wynik. Parametry tego typu nie zmieniały się podczas testów. Po wybraniu parametrów badawczych zbadano wpływ zmiany jednego z nich na jakość algorytmu.

Parametry badawcze:

- Branch&Bound:
    - brak

- TabuSearch:
    - czas obecności na liście: 0.25n, 0.5n, n, 2n
    - czas pracy: 5s, 10s, 30s

- Algorytm genetyczny:
    - wielkość populacji: 25, 50, 100
    - współczynnik krzyżowania: 0.33, 0.80, 0.99
    - czas pracy: 10s, 30s, 60s,

- Symulowane wyżarzanie:

- Temperatura początkowa: 1000, 10000, 5000
- Temperatura końcowa: 0.1, 0.01, 0.001
- Liczba powtórzeń dla jednej temperatury: 25, 50, 100

Dla algorytmów TabuSearch, Symulowanego Wyżarzania i Algorytmu Genetycznego testy wyglądały w następujący sposób:

1. Wejściowy algorytm posiadał domyślne parametry.
2. Zmieniono jeden z nich na wartość z opisanego powyżej zbioru testów.
3. W takiej konfiguracji test został wywołany dziesięciokrotnie.
4. Uzyskano **10 wyników** dla każdej konfiguracji. Wyniki uśredniono.
    a. Dla TabuSearch i Algorytmu Genetycznego - długość trasy (czas jest parametrem)
    b. Dla Symulowanego Wyżarzania - długość i czas pracy

Dla algorytmy Branch&Bound ze względu na to, że jest algorytmem deterministycznym i nie ma żadnych parametrów testy zostały wykonane jeden raz dla każdego (małego) pliku testowego.

7. Analiza otrzymanych wyników

Na kolejnych stronach przedstawiono wyniki przeprowadzonych testów. Pierwsze cztery tabele (tabele o numerach 1-4) ukazują wyniki (najmniejsze uzyskane koszty przejścia) uzyskane dla każdej z testowanych konfiguracji algorytmów w zależności od zestawu danych wejściowych.

Następnie zaprezentowano zbiorczą dla wszystkich algorytmów tabelę (tabela nr 5) zawierającą zwycięskie (cechujące się najlepszymi wynikami) konfiguracje każdego algorytmu dla poszczególnych zestawów danych. Na podstawie tabeli zostały sporządzone wykresy obrazujące różnice pomiędzy wynikami osiągniętymi przez algorytmy (rysunki nr 4-9).

Czasy pracy algorytmów:

- TabuSearch - 92min.
- Algorytm Genetyczny - 291min.
- Symulowane Wyżarzanie - 6min
- Branch&Bound: 33min.

Tabela nr 1 - wyniki dla Branch&Bound

	Wyniki dla poszczególnych plików z zestawami danych			
	12-1.txt	12-2.txt	17.txt	24.txt
Uzyskany wynik	54	47	62	80

Tabela nr 2 - wyniki dla TabuSearch

Parametry algorytmu	Wyniki dla poszczególnych plików z zestawami danych					
	12-1.txt	12-2.txt	17.txt	24.txt	31.txt	50.txt

Kadencja: 0.25*iloscMiast Czas: 10s Aspiracja: włączona Dywersyfikacja: włączona Iteracje bez poprawy do dywersyfikacji: 10000	54	47	61,1	81	98,14	154
Kadencja: 0.50*iloscMiast Czas: 10s Aspiracja: włączona Dywersyfikacja: włączona Iteracje bez poprawy do dywersyfikacji: 10000	54	47	61	78,88	94	154
Kadencja: 1.00*iloscMiast Czas: 10s Aspiracja: włączona Dywersyfikacja: włączona Iteracje bez poprawy do dywersyfikacji: 10000	54	47	61	79,9	98,7	157
Kadencja: 2.00*iloscMiast Czas: 10s Aspiracja: włączona Dywersyfikacja: włączona Iteracje bez poprawy do dywersyfikacji: 10000	54	47	61	81	98	157
Kadencja: 0.50*iloscMiast **Czas: 5s** Aspiracja: włączona Dywersyfikacja: włączona Iteracje bez poprawy do dywersyfikacji: 10000	54	47	61	79,6	94	154
Kadencja: 0.50*iloscMiast **Czas: 10s** Aspiracja: włączona Dywersyfikacja: włączona Iteracje bez poprawy do dywersyfikacji: 10000	54	47	61	79,4	94	154
Kadencja: 0.50*iloscMiast **Czas: 30s** Aspiracja: włączona Dywersyfikacja: włączona Iteracje bez poprawy do dywersyfikacji: 10000	54	47	61	79	94	153,9

Tabela nr 3 - wyniki dla Algorytmu Genetycznego

Parametry algorytmu	Wyniki dla poszczególnych plików z zestawami danych					
	12-1.txt	12-2.txt	17.txt	24.txt	31.txt	50.txt
Czas: 30s **Populacja: 25** Współczynnik krzyżowania: 0.8 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	63,2	88,8	113	184,7
Czas: 30s **Populacja: 50** Współczynnik krzyżowania: 0.8 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	62,8	87,8	107,3	176
Czas: 30s **Populacja: 100** Współczynnik krzyżowania: 0.8 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	61,8	83,9	104	169,1
Czas: 30s Populacja: 50 **Współczynnik krzyżowania: 0.33** Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	62,7	87,5	109,4	179,8
Czas: 30s Populacja: 50 **Współczynnik krzyżowania: 0.80** Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	62,6	87	108,1	177,4
Czas: 30s Populacja: 50 **Współczynnik krzyżowania: 0.99** Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	62,4	83,8	107,8	174,1
Czas: 10s Populacja: 50 Współczynnik krzyżowania: 0.80 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	63	85,8	107,3	177,7
Czas: 30s Populacja: 50 Współczynnik krzyżowania: 0.80 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	63,7	85,9	109,5	176,8
Czas: 60s Populacja: 50 Współczynnik krzyżowania: 0.80 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	54	47	63	86,1	110	179,6

Tabela nr 4 - wyniki dla Symulowanego Wyżarzania

Parametry algorytmu	Wyniki dla poszczególnych plików z zestawami danych					
	12-1.txt	12-2.txt	17.txt	24.txt	31.txt	50.txt
Temperatura początkowa: 1000 Temperatura końcowa: 0.01 Ilość powtórzeń: 25 Współczynnik alpha: 0.999	54	47	61,1	78,7	94,2	144,1
Temperatura początkowa: 5000 Temperatura końcowa: 0.01 Ilość powtórzeń: 25 Współczynnik alpha: 0.999	54	47	61,1	78,7	93,6	144,3
Temperatura początkowa: 10000 Temperatura końcowa: 0.01 Ilość powtórzeń: 25 Współczynnik alpha: 0.999	54	47	61,2	78,8	93,7	143,2
Temperatura początkowa: 1000 **Temperatura końcowa: 0.1** Ilość powtórzeń: 25 Współczynnik alpha: 0.999	54	47	61	78,7	94,5	144,6
Temperatura początkowa: 1000 **Temperatura końcowa: 0.01** Ilość powtórzeń: 25 Współczynnik alpha: 0.999	54	47	61	78,6	94	143,9
Temperatura początkowa: 1000 **Temperatura końcowa: 0.001** Ilość powtórzeń: 25 Współczynnik alpha: 0.999	54	47	61	79	94	143,8
Temperatura początkowa: 1000 Temperatura końcowa: 0.01 **Ilość powtórzeń: 25** Współczynnik alpha: 0.999	54	47	61,2	79	93,9	144,5
Temperatura początkowa: 1000 Temperatura końcowa: 0.01 **Ilość powtórzeń: 50** Współczynnik alpha: 0.999	54	47	61	78,4	93,8	142,2
Temperatura początkowa: 1000 Temperatura końcowa: 0.01 **Ilość powtórzeń: 100** Współczynnik alpha: 0.999	54	47	61	78,4	93,2	141,7

Analizując tabele ze szczegółowymi wynikami dla algorytmów w zależności od zbioru danych testowych zaobserwowano:

Dla małych zbiorów danych:

1. Zmiana parametrów algorytmów nie miała wpływu na osiągane wyniki.

Dla średnich i/lub dużych zbiorów danych:

Branch&Bound:

1. Dla średnich zbiorów danych algorytm uzyskiwał gorsze wyniki niż reszta algorytmów. Prawdopodobnie spowodowane jest to dosyć niską złożonością sposobu funkcjonowania algorytmu.
2. Algorytm Branch&Bound ze względu na swoją specyfikę miał pełnić rolę benchmarka zawsze znajdując optymalne rozwiązanie. Jak widać nie zawsze tak się działo. Prawdopodobnie w implementacji algorytmu znajduje się błąd, jednak nie udało się nam go zlokalizować.

TabuSearch:

1. Ustawienie kadencji na 0.5*iloscMiast prowadziło do osiągnięcia o średnio 3% lepszych wyników niż dla kadencji 0.25*iloscMiast, 1.00*iloscMiast oraz 2.00*iloscMiast. Im większy zbiór danych tym mniejsza różnica.
2. Im dłuższy czas pracy algorytmu tym lepszy wynik udało się osiągnąć.
3. Włączenie aspiracji i dywersyfikacji wpływa pozytywnie na wyniki osiągane przez algorytm.

Algorytm Genetyczny:

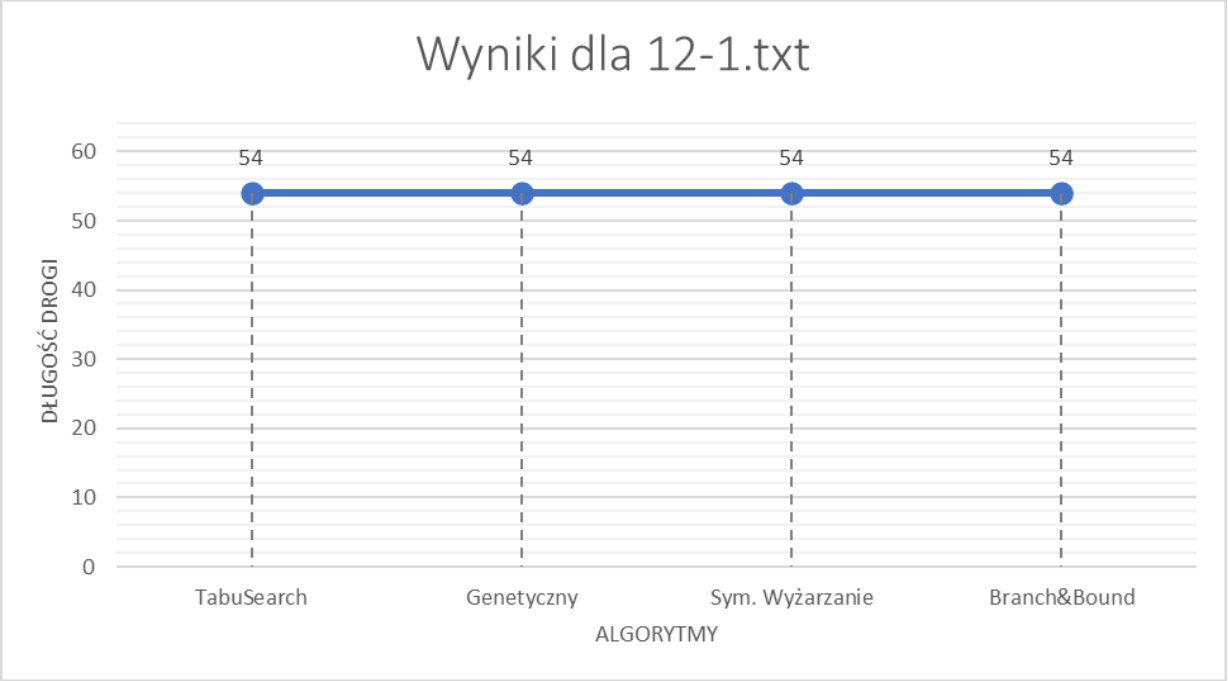
1. Im większa populacja tym lepszy wynik.
2. Im wyższy współczynnik krzyżowania tym lepszy wynik.
3. Dłuższa praca algorytmu prowadziła do nieznacznego pogorszenia wyniku.

Symulowane Wyżarzanie:

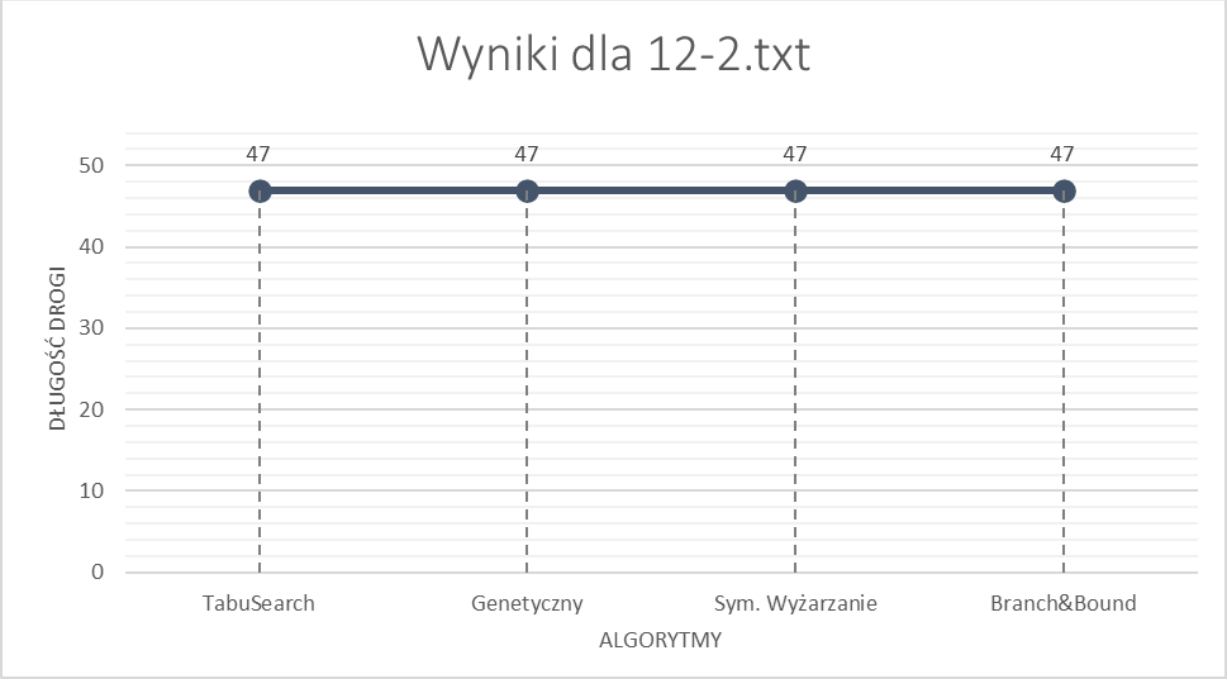
1. Wzrost temperatury początkowej zwykle powodował nieznaczne pogorszenie wyników osiąganych przez algorytm.
2. Spadek temperatury końcowej zwykle powodował nieznaczne polepszenie wyników osiąganych przez algorytm.
3. Zwiększanie ilości powtórzeń pozytywnie wpływało na wyniki osiągane przez algorytm.

Tabela nr 5 - konfiguracje algorytmów, które osiągnęły najlepsze wyniki

Plik (zestaw danych)	Wynik oraz konfiguracja algorytmu			
	TabuSearch	Algorytm Genetyczny	Symulowane Wyżarzanie	Branch&Bound
12-1.txt	Wynik: **54** Konfiguracje osiągnęły ten sam wynik	Wynik: **54** Konfiguracje osiągnęły ten sam wynik	Wynik: **54** Konfiguracje osiągnęły ten sam wynik	Wynik: **54** Brak konfiguracji
12.2.txt	Wynik: **47** Konfiguracje osiągnęły ten sam wynik	Wynik: **47** Konfiguracje osiągnęły ten sam wynik	Wynik: **47** Konfiguracje osiągnęły ten sam wynik	Wynik: **47** Brak konfiguracji
17.txt	Wynik: **61** Kadencja: 0.50*iloscMiast Czas: 10s Aspiracja: tak Dywersyfikacja: tak Iteracje bez poprawy do dywersyfikacji: 10000	Wynik: **61,8** Czas: 30s Populacja: 100 Współczynnik krzyżowania: 0.8 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	Wynik: **61** Temperatura początkowa: 1000 Temperatura końcowa: 0.1 Ilość powtórzeń: 100 Współczynnik alpha: 0.999	Wynik: **62** Brak konfiguracji
24.txt	Wynik: **78,88** Kadencja: 0.50*iloscMiast Czas: 10s Aspiracja: tak Dywersyfikacja: tak Iteracje bez poprawy do dywersyfikacji: 10000	Wynik: **83,8** Czas: 30s Populacja: 50 Współczynnik krzyżowania: 0.99 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	Wynik: **78,4** Temperatura początkowa: 1000 Temperatura końcowa: 0.01 Ilość powtórzeń: 100 Współczynnik alpha: 0.999	Wynik: **80** Brak konfiguracji
31.txt	Wynik: **94** Kadencja: 0.50*iloscMiast Czas: 10s Aspiracja: tak Dywersyfikacja: tak Iteracje bez poprawy do dywersyfikacji: 10000	Wynik: **104** Czas: 30s Populacja: 100 Współczynnik krzyżowania: 0.8 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	Wynik: **93,2** Temperatura początkowa: 1000 Temperatura końcowa: 0.01 Ilość powtórzeń: 100 Współczynnik alpha: 0.999	
50.txt	Wynik: **153,9** Kadencja: 0.50*iloscMiast Czas: 30s Aspiracja: tak Dywersyfikacja: tak Iteracje bez poprawy do dywersyfikacji: 10000	Wynik: **169,1** Czas: 30s Populacja: 100 Współczynnik krzyżowania: 0.8 Współczynnik mutacji 0.01 Mutacja: wierzchołkowa	Wynik: **141,7** Temperatura początkowa: 1000 Temperatura końcowa: 0.01 Ilość powtórzeń: 100 Współczynnik alpha: 0.999	

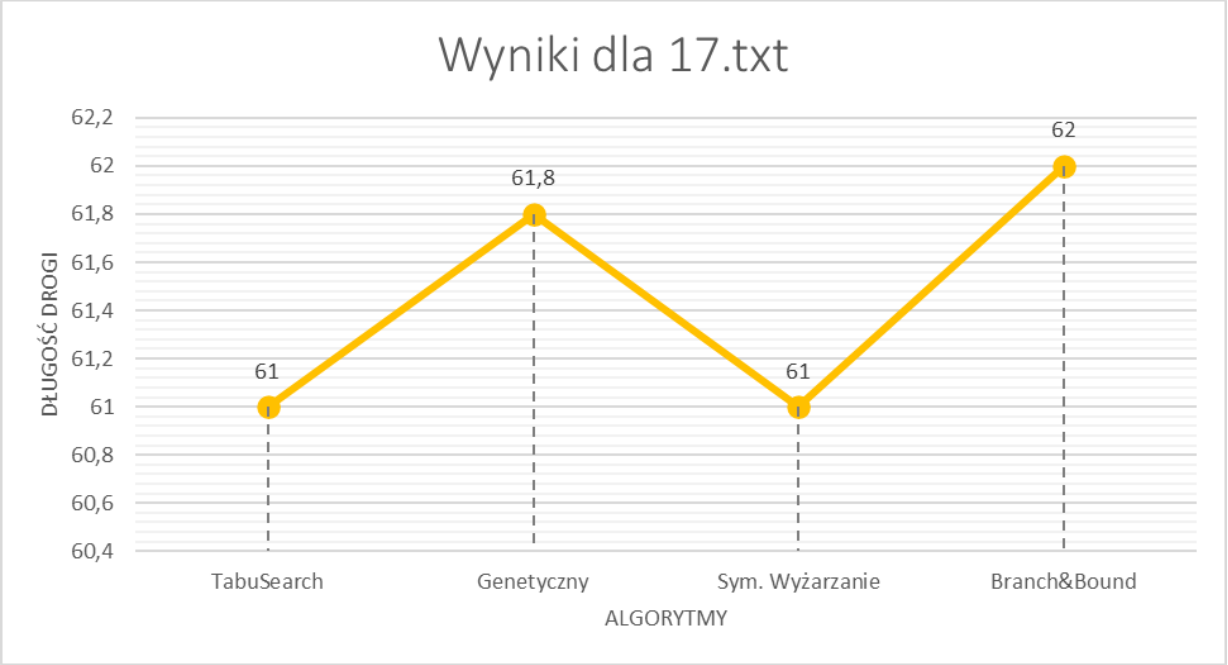


Rys. 4 - Wyniki dla zbioru danych z pliku 12-1.txt

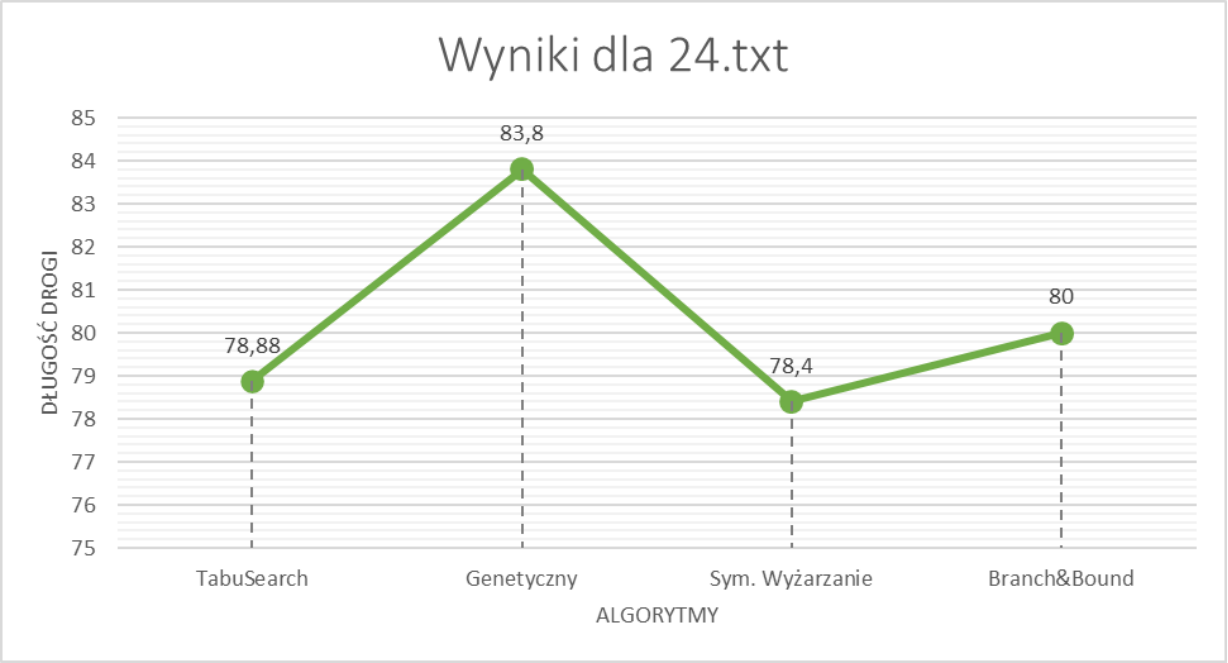


Rys. 5 - Wyniki dla zbioru danych z pliku 12-2.txt

Jak widać na powyższych wykresach dla mniejszych zestawów danych (z plików 12-1.txt oraz 12-2.txt) algorytmy osiągnęły te same wyniki. Warto jednak podkreślić, że nasza implementacja Symulowanego Wyżarzania osiągnęła wynik znacznie szybciej od pozostałych algorytmów.

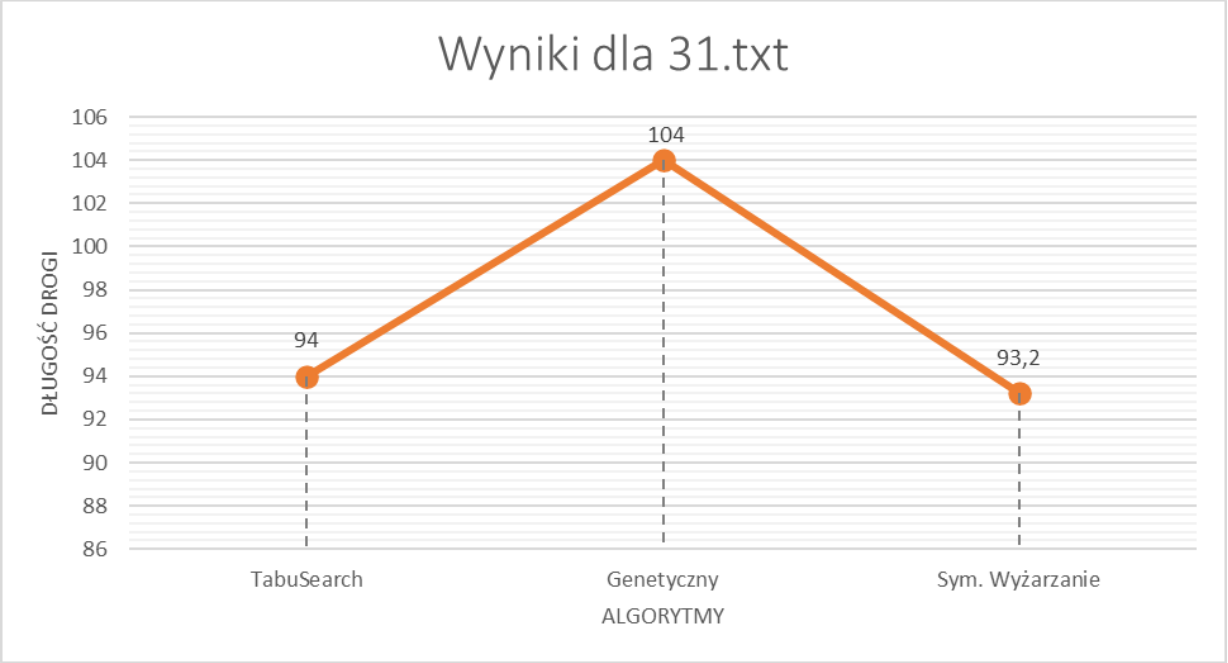


Rys. 6 - Wyniki dla zbioru danych z pliku 17.txt

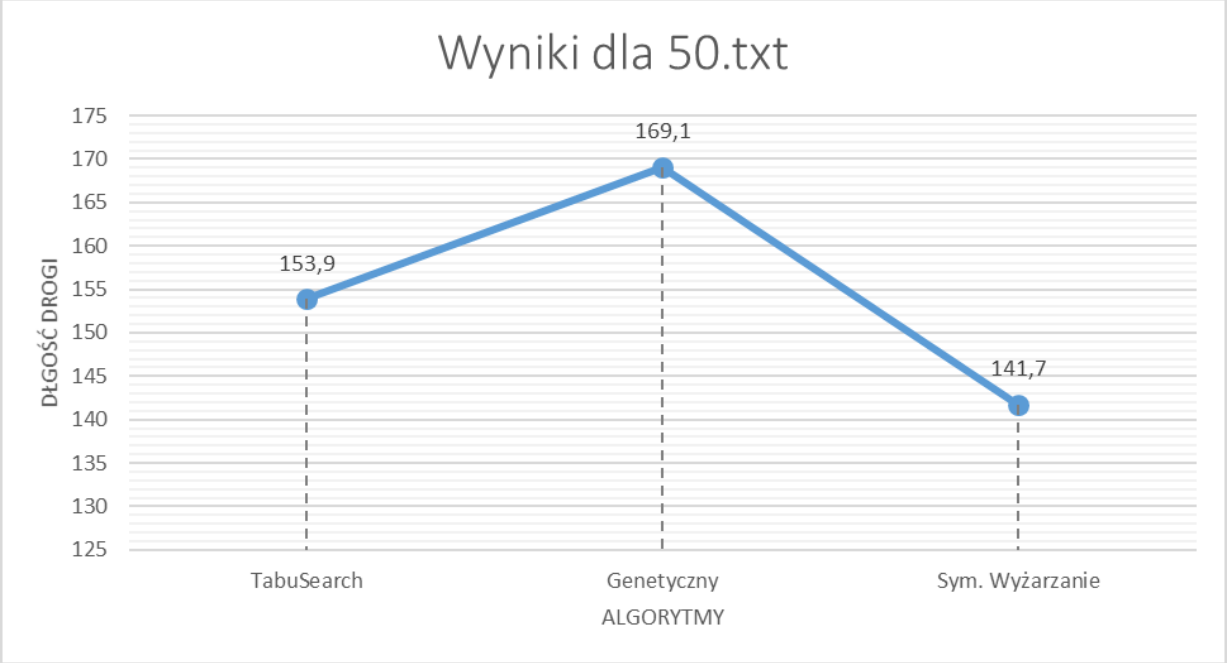


Rys. 7 - Wyniki dla zbioru danych z pliku 24.txt

Jak widać na powyższych wykresach dla średnich zestawów danych (z plików 17.txt i 24.txt) najgorszy okazał się algorytm genetyczny. Pozostałe algorytmy osiągnęły podobne wyniki. Różnica pomiędzy TabuSearch i Symulowanym Wyżarzaniem była szczególnie mała. Najlepszy okazało się jednak Symulowane Wyżarzanie. Dodatkowo, to właśnie ten algorytm znalazł optymalny wynik najszybciej.



Rys. 8 - Wyniki dla zbioru danych z pliku 31.txt



Rys. 9 - Wyniki dla zbioru danych z pliku 50.txt

Dla dużych zbiorów danych zrezygnowano z testów algorytmu Branch&Bound, ponieważ czas działania algorytmu był zbyt długi - co znacząco obniża użyteczność oraz sens wykorzystywania algorytmu. W tym przypadku ponownie najlepszy okazał się algorytm Symulowanego Wyżarzania z małą stratą dla TabuSearch.

Dodatkowo zaobserwowano, że im większy zbiór danych tym różnica pomiędzy wynikami TabuSearch i Symulowanego Wyżarzania jest większa na korzyść drugiego z algorytmów.

8. Podsumowanie

Na podstawie zaprojektowanych badań przeprowadzonych z wykorzystaniem autorskiego symulatora udało się stwierdzić, która z autorskich implementacji algorytmów rozwiązujących problem komiwojażera osiągał najlepsze wyniki. Najlepsze - zarówno pod względem osiągniętego wyniku jak i szybkości działania - okazało się Symulowane Wyżarzanie.

W przyszłości w celu dalszego rozwoju projektu można by udoskonalić implementację algorytmu Branch&Bound. Dzięki temu mógłby on pełnić rolę benchmarka w eksperymencie. Być może dobrym pomysłem byłoby także rozszerzenie testów o dodatkowe algorytmy. Warto byłoby również pomyśleć o jeszcze większym zróżnicowaniu zbiorów danych testowych. Istnieje wtedy możliwość na zbadanie wszystkich algorytmów w szerszym zakresie danych i odkrycie ich cech ujawniających się dopiero na specyficznych zbiorach danych.

9. Bibliografia

1. Hoffman, Karla L., Manfred Padberg, and Giovanni Rinaldi. "Traveling salesman problem." Encyclopedia of operations research and management science 1 (2013): 1573-1578.
2. Balas, Egon, and Paolo Toth, "Branch and bound methods for the traveling salesman problem.", 1983
3. F. Glover, T. Laguna, Tabu search, Kluwer Academic Publishers, 1997.
4. S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi, “Optimization by Simulated Annealing”, Science 220, 671–680, 1983.
5. Mirjalili S., Genetic Algorithm. In: Evolutionary Algorithms and Neural Networks. Studies in Computational Intelligence, vol 780. Springer, Cham (2019).
6. R. Ahuja, O. Ergun, J. Orlin, A. Punnen, A survey of very large-scale neighborhood search techniques, Discrete Applied Mathematics 123 (2002) 75–102.
7. R. Wieczorkowski, Algorytmy stochastyczne w optymalizacji dyskretnej przy zaburzonych wartościach funkcji, Matematyka Stosowana 38 (1995) 119–153.
8. Balas, Egon, and Paolo Toth. "Branch and bound methods for the traveling salesman problem." (1983).
9. Zaawansowane programowanie, Wykład 5: Algorytmy Dokładne, prof. dr hab. inż. Marta Kasprzak, Instytut Informatyki, Politechnika Poznańska
10. Hoffman, Karla L., Manfred Padberg, and Giovanni Rinaldi. "Traveling salesman problem. "Encyclopedia of operations research and management science 1 (2013): 1573-1578.