Projektowanie Efektywnych Algorytmów

Projekt 3

Katsiaryna Kolyshko 276708

Dr. inż. Marcin Łapuszyński

Wstęp teorytyczny

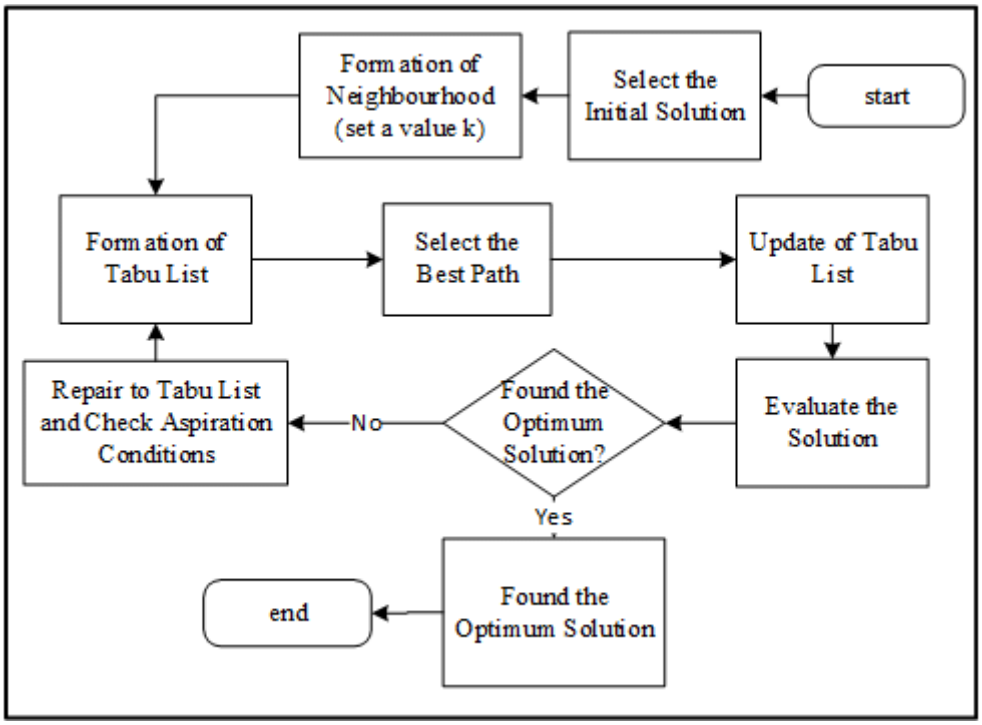
Tabu Search

Przeszukiwanie z Tabu - Co To Jest i Jak Działa?

Przeszukiwanie z Tabu to metaheurystyczny algorytm optymalizacyjny, służący do rozwiązywania złożonych problemów kombinatorycznych, gdy znalezienie dokładnego rozwiązania zajęłoby zbyt dużo czasu.

Jak Działa:

1. Rozpoczyna od rozwiązania początkowego
2. Definiuje sąsiedztwo (możliwe modyfikacje obecnego rozwiązania)
3. Ocenia sąsiednie rozwiązania
4. Aktualizuje obecne rozwiązanie, wybierając lepsze
5. Utrzymuje listę tabu (niedawno odwiedzone rozwiązania)
6. Stosuje kryterium aspiracji (możliwość złamania tabu dla wyjątkowo dobrych rozwiązań)
7. Powtarza proces do spełnienia warunku stopu



Rysunek 1: Przykład działania algorytmu Tabu Search

Źródło: https://www.researchgate.net/publication/336206097_Recommendation_of_Scheduling_Tourism_Routes_using_Tabu_Search_Method_Case_Study_Bandung

Zastosowanie w Problemach Macierzowych:

- Macierz reprezentuje koszty między elementami
- Sąsiedztwo tworzone przez zamiany wierszy, kolumn lub wartości
- Lista tabu zapobiega zapętleniu się w lokalnych minimach

Zalety:

- Unika lokalnych minimów
- Elastyczność i uniwersalność
- Prosta implementacja
- Skuteczność dla dużych przestrzeni rozwiązań

Wady:

- Nie gwarantuje optymalnego rozwiązania
- Trudny dobór parametrów
- Możliwość utknięcia na płaskowyżach
- Duże zużycie pamięci

Złożoność Czasowa i Pamięciowa w Przeszukiwaniu z Tabu:

Złożoność Czasowa:

- $O(n \times i \times s)$, gdzie:
  - n - rozmiar problemu (np. liczba miast w TSP)
  - i - liczba iteracji
  - s - rozmiar sąsiedztwa

Dla TSP:

- Generowanie sąsiedztwa: $O(n^2)$
- Ocena sąsiednich rozwiązań: $O(n)$
- Całkowita złożoność: $O(n^2 \times i)$

Złożoność Pamięciowa:

- $O(t \times e)$, gdzie:
  - t - rozmiar listy tabu
  - e - rozmiar pojedynczego elementu na liście

Dodatkowy czynnik wpływający na złożoność to struktura sąsiedztwa, co będziemy badać dla naszego projektu

Przeszukiwanie z Tabu to skuteczna metoda optymalizacji, szczególnie przydatna w złożonych problemach kombinatorycznych. Choć nie gwarantuje znalezienia najlepszego rozwiązania, pozwala uniknąć lokalnych minimów i znaleźć dobre rozwiązania w rozsądnym czasie.

Przeszukiwanie z Tabu znajduje szerokie zastosowanie w optymalizacji gier (np. szachy), planowaniu tras logistycznych, harmonogramowaniu produkcji oraz projektowaniu sieci. W przeciwieństwie do DFS (przeszukiwanie w głąb) i BFS (przeszukiwanie wszerz), które systematycznie eksplorują całą przestrzeń rozwiązań, Tabu Search inteligentnie kieruje poszukiwania w obiecujące obszary, unikając lokalnych minimów poprzez mechanizm listy tabu.

Podczas gdy algorytm najniższego kosztu (Lowest Cost) zawsze wybiera lokalnie najlepsze rozwiązanie, Tabu Search może tymczasowo zaakceptować gorsze rozwiązania, aby wydostać się z lokalnych minimów. Ta elastyczność, w połączeniu z efektywnym wykorzystaniem pamięci historycznej (lista tabu), sprawia, że Tabu Search jest szczególnie skuteczny w rozwiązywaniu złożonych problemów optymalizacyjnych, gdzie tradycyjne metody przeszukiwania często zawodzą.

Opis działania kodu

Wczytywanie i opracowanie plików .xml

Przede wszystkim, aby odczytać pliki, musiałem zainstalować i zasadniczo skopiować i pożyczyć istniejący kod, aby zaakceptować plik w formacie .xml, ponieważ miałem problemy z pobieraniem i otwieraniem plików w innym możliwym formacie. Kod, który jest pożyczony, znajduje się w folderze xmlParser, napisanym i wykonanym przez Lee Thomasona. Uprawnienia do korzystania z kodu i analiz są pokazane przez samego autora w komentarzu tinyxml2.cpp, jak pokazano na poniższym obrazku.

```
/*
Original code by Lee Thomason (www.grinninglizard.com)

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any
damages arising from the use of this software.

Permission is granted to anyone to use this software for any
purpose, including commercial applications, and to alter it and
redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must
not claim that you wrote the original software. If you use this
software in a product, an acknowledgment in the product documentation
would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and
must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
distribution.
*/
```

Rysunek 3: Zezwolenie na korzystanie z kodu

Jak są dostępne pliki .xml, następnie musimy ich wczytać w taki sposób, aby było to nam wygodnie dla programu. Stworzono dlatego DataReader.cpp i DataReader.h.

Kod DataReader.cpp zawiera trzy główne funkcje w klasie DataReader:

1. createGraphFromFile:

   Funkcja createGraphFromFile wczytuje plik XML zawierający instancję problemu komiwojażera (TSP), gdzie parsuje jego strukturę w celu pobrania metadanych (nazwa, źródło, opis) oraz danych grafu. Na podstawie wczytanych danych tworzy macierz sąsiedztwa reprezentującą graf, gdzie wartości oznaczają koszty przejść między wierzchołkami. W przypadku poprawnego wczytania i

przetworzenia danych, funkcja zwraca utworzony obiekt TSPGraph, natomiast w razie wystąpienia błędów podczas wczytywania lub przetwarzania XML, które są obsługiwane przez system wyjątków, zwraca nullptr.

2. savePathToFile:

```cpp
int DataReader::savePathToFile(std::vector<int>& path, const char* filename) {
    std::ofstream outFile( s: filename);
    if (!outFile.is_open()) {
        return -1;
    }

    outFile << path.size() << "\n";

    for (int vertex : path) {
        outFile << vertex << "\n";
        if (outFile.fail()) {
            outFile.close();
            return -1;
        }
    }

    outFile << path[0] << "\n";
    outFile.close();
    return 0;
}
```

Rysunek 3: Rzut ekranu z przykładem kodu funkcji savePathToFile.
Źródło: opracowanie własne

Funkcja savePathToFile zapisuje znalezione rozwiązanie problemu TSP do pliku tekstowego. W pierwszej linii umieszcza liczbę wierzchołków, a w kolejnych liniach zapisuje numery kolejnych wierzchołków tworzących ścieżkę. Funkcja zwraca 0 w przypadku pomyślnego zapisu lub -1, jeśli wystąpi błąd podczas operacji zapisu do pliku.

3. calculatePathFromFile:

```cpp
int DataReader::calculatePathFromFile(const char* filename, TSPGraph*& graph) {
    std::ifstream inFile( s: filename);
    if (!inFile.is_open()) {
        return -1;
    }

    int vertexCount;
    if (!(inFile >> vertexCount)) {
        inFile.close();
        return -1;
    }

    std::vector<int> path;
    path.reserve( n: vertexCount);

    for (int i = 0; i < vertexCount; i++) {
        int vertex;
        if (!(inFile >> vertex)) {
            inFile.close();
            return -1;
        }
        path.push_back(vertex);
    }

    inFile.close();
    return graph->calculateTour( tour: path);
}
```

Rysunek 4: Rzut ekranu z przykładem kodu funkcji calculatePathFromFile.
Źródło: opracowanie własne

Funkcja calculatePathFromFile wczytuje z pliku tekstowego wcześniej zapisaną ścieżkę, odczytując liczbę wierzchołków oraz ich sekwencję. Następnie wykorzystuje metodę calculateTour z obiektu TSPGraph do obliczenia kosztu wczytanej ścieżki. W rezultacie zwraca obliczony koszt, a w przypadku wystąpienia błędu podczas odczytu zwraca wartość -1.

Reprezentacja grafów i macierzy w projekcie

Stworzono *GraphRep.cpp i GraphRep.h* , które są potrzebne do reprezentacji grafów.

```
void GraphRep::printAdjacencyMatrix(int** table, int verticesNum) {
    if (!table || verticesNum <= 0) {
        std::cerr << "Error: Invalid matrix or vertex count" << std::endl;
        return;
    }
    // Print column headers
    std::cout << "Adjacency Matrix:" << std::endl;
    std::cout << std::setw( n: COLUMN_WIDTH) << " ";  // Empty cell for row headers
    // Print vertex numbers as column headers
    for (int i = 0; i < verticesNum; i++) {
        std::cout << std::setw( n: COLUMN_WIDTH) << i;
    }
    std::cout << std::endl;
    // Print matrix rows
    for (int i = 0; i < verticesNum; i++) {
        // Print row header (vertex number)
        std::cout << std::setw( n: COLUMN_WIDTH) << i;
        // Print row values
        for (int j = 0; j < verticesNum; j++) {
            std::cout << std::setw( n: COLUMN_WIDTH);
            // Check for infinity value
            if (table[i][j] == INT_MAX) {
                std::cout << INFINITY_SYMBOL;
            } else {
                std::cout << table[i][j];
            }
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

Rysunek 5: Rzut ekranu z przykładem kodu funkcji printAdjacencyMatrix.
Źrodło: opracowanie własne

Funkcja printAdjacencyMatrix służy do wyświetlania macierzy sąsiedztwa dla grafu w czytelnym, tabelarycznym formacie. Jest to funkcja członkowska klasy GraphRep, która wizualizuje strukturę grafu poprzez wyświetlanie połączeń między wierzchołkami.

Funkcja przyjmuje jako parametry dwuwymiarową tablicę (table) reprezentującą macierz sąsiedztwa oraz liczbę wierzchołków (verticesNum). Po sprawdzeniu poprawności danych wejściowych, wyświetla nagłówki kolumn numerowane od 0 do (verticesNum-1). Następnie dla każdego wiersza macierzy wyświetla numer wiersza oraz wartości połączeń z innymi wierzchołkami, gdzie symbol "-" reprezentuje brak połączenia (nieskończoność).

Formatowanie wyświetlania jest kontrolowane przez stałą COLUMN_WIDTH, która zapewnia równe odstępy między kolumnami. Funkcja wykorzystuje manipulatory strumienia (setw) do zachowania odpowiedniego wyrównania kolumn, co poprawia czytelność wyświetlanej macierzy.

TSPGraph.cpp i TSPGraph.h są odpowiedzialne za implementację struktury grafu oraz algorytmów rozwiązujących problem komiwojażera w programie. Pliki te zawierają definicję klasy TSPGraph, która zarządza macierzą sąsiedztwa reprezentującą graf oraz implementuje algorytm zachłanny (greedy) do znajdowania tras. Niżej jest przedstawiony kod pliku:

```
TSPGraph::TSPGraph(int verticesNumber, int**& adjacencyMatrix)
        : verticesNumber(verticesNumber)
        , adjacencyMatrix(adjacencyMatrix) {
}

TSPGraph::TSPGraph()
        : verticesNumber(0)
        , adjacencyMatrix(nullptr) {
}

TSPGraph::~TSPGraph() {
    if (adjacencyMatrix != nullptr) {
        for (int i = 0; i < verticesNumber; i++) {
            delete[] adjacencyMatrix[i];
        }
        delete[] adjacencyMatrix;
    }
}
```

Rysunek 6: Rzut ekranu z przykładem kodu funkcji TSPGraph.
Źrodło: opracowanie własne

Przedstawione funkcje to konstruktory i destruktor klasy TSPGraph:

1. TSPGraph(int verticesNumber, int**& adjacencyMatrix) - konstruktor parametryczny, inicjalizuje obiekt z podaną liczbą wierzchołków i macierzą sąsiedztwa.

2. TSPGraph() - konstruktor domyślny, tworzy pusty graf (0 wierzchołków, pusta macierz).

3. ~TSPGraph() - destruktor, zwalnia pamięć zaalokowaną dla macierzy sąsiedztwa poprzez usunięcie każdego wiersza, a następnie tablicy wskaźników.

```cpp
int** TSPGraph::getAdjMatrix() const {
    return adjacencyMatrix;
}

int TSPGraph::getVerticesNumber() const {
    return verticesNumber;
}

std::vector<int> TSPGraph::greedyTSP() {
    std::vector<bool> visited( n: verticesNumber,  value: false);
    std::vector<int> currentPath( n: verticesNumber);
    std::vector<int> bestPath( n: verticesNumber);
    int bestPathCost = INT_MAX;

    // Try starting from each vertex
    for (int startVertex = 0; startVertex < verticesNumber; startVertex++) {
        // Initialize for current starting vertex
        std::fill( first: visited.begin(),  last: visited.end(),  value: false);
        int currentVertex = startVertex;
        currentPath[0] = currentVertex;
        visited[currentVertex] = true;

        // Build path by selecting nearest unvisited vertex
        for (int pathPos = 1; pathPos < verticesNumber; ++pathPos) {
            int nearestVertex = -1;
            int minDistance = INT_MAX;

            // Find nearest unvisited vertex
            for (int nextVertex = 0; nextVertex < verticesNumber; ++nextVertex) {
                if (!visited[nextVertex]) {
                    int distance = adjacencyMatrix[currentVertex][nextVertex];
                    if (distance < minDistance) {
                        minDistance = distance;
                        nearestVertex = nextVertex;
                    }
                }
            }

            // Add nearest vertex to path
            visited[nearestVertex] = true;
            currentPath[pathPos] = nearestVertex;
            currentVertex = nearestVertex;
        }

        // Check if current path is better than best found
        int currentPathCost = calculateTour( tour: currentPath);
        if (currentPathCost < bestPathCost) {
            bestPath = currentPath;
            bestPathCost = currentPathCost;
        }
    }

    return bestPath;
}

int TSPGraph::calculateTour(std::vector<int> tour) const {
    int totalCost = 0;

    // Calculate cost between consecutive vertices
    for (int i = 0; i < verticesNumber - 1; i++) {
        totalCost += adjacencyMatrix[tour[i]][tour[i + 1]];
```

```
87        // Add cost of returning to start
88        totalCost += adjacencyMatrix[tour.back()][tour.front()];
89
90        return totalCost;
91    }
```

Rysunek 7: Rzut ekranu z przykładem kodu klasy TSPGrap.
Źrodło: opracowanie własne

Przedstawiony kod zawiera implementację klasy TSPGraph, która służy do reprezentacji i rozwiązywania problemu komiwojażera. Klasa posiada konstruktory przyjmujący liczbę wierzchołków i macierz sąsiedztwa oraz domyślny, a także destruktor zwalniający zaalokowaną pamięć dla macierzy.

Metoda greedyTSP implementuje zachłanny algorytm rozwiązywania problemu TSP. Dla każdego wierzchołka startowego tworzy ścieżkę, wybierając kolejno najbliższy nieodwiedzony wierzchołek. Porównuje koszty znalezionych ścieżek i zwraca najlepszą znalezioną trasę. Tak jest znaleziono pierwsze rozwiązanie do grafu.

Metoda calculateTour oblicza całkowity koszt podanej trasy, sumując koszty przejść między kolejnymi wierzchołkami oraz koszt powrotu do wierzchołka startowego. Dodatkowo klasa zawiera proste metody dostępowe getAdjMatrix i getVerticesNumber zwracające odpowiednio macierz sąsiedztwa i liczbę wierzchołków.

Funkcja i implementacja Algorytmu Tabu:

Algorytmy sąsiedztwa:

1. Swap (zamiana):

Bierze dwa wierzchołki i zamienia je miejscami

Przykład: [1,2,3,4] → [1,4,3,2] (zamiana 2 i 4)).

Jest najprostszą metodą generowania sąsiedztwa. Zachowuje wszystkie wierzchołki i zmienia tylko ich kolejność.

```
15  std::vector<int> TabooSearch::swapPositions(std::vector<int> tour, int index1, int index2) {
16      if(index1 >= tour.size() || index2 >= tour.size()) return {};
17      std::swap( &: tour[index1],  &: tour[index2]);
18      return tour;
19  }
20
```

Rysunek 8: Rzut ekranu z kodem sąsiedztwa Swap.
Źrodło: opracowanie własne

2. Reverse (odwrócenie):

Odwraca kolejność wierzchołków w wybranym segmencie ścieżki

Przykład: [1,2,3,4] → [1,4,3,2] (odwrócenie 2-4)

Jest dobry do lokalnej optymalizacji dłuższych segmentów, ale może znacząco zmienić strukturę rozwiązania

```cpp
std::vector<int> TabooSearch::reverseNumbers(std::vector<int> tour, int startingIndex, int endingIndex) {
    if(startingIndex >= tour.size() || endingIndex >= tour.size()) return {};
    if (startingIndex > endingIndex) {
        std::swap( &: startingIndex, &: endingIndex);
    }

    while (startingIndex < endingIndex) {
        std::swap( &: tour[startingIndex], &: tour[endingIndex]);
        ++startingIndex;
        --endingIndex;
    }
    return tour;
}
```

Rysunek 9: Rzut ekranu z kodem sąsiedztwa Reverse.
Źrodło: opracowanie własne

3. Insert (wstawianie):

Przenosi jeden wierzchołek przed inny, pozwala na precyzyjne modyfikacje.

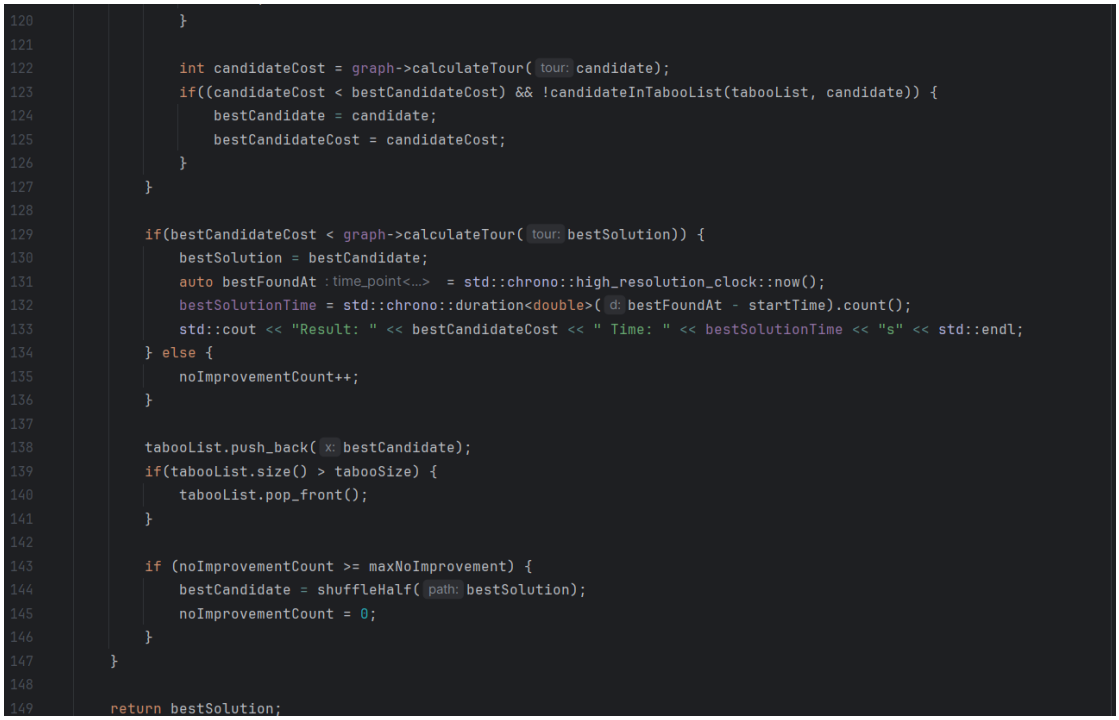
Przykład: [1,2,3,4] → [1,3,2,4] (2 wstawione po 3)

```cpp
std::vector<int> TabooSearch::insertBeforeNumber(std::vector<int> tour, int numberIndex, int targetIndex) {
    if(numberIndex >= tour.size() || targetIndex >= tour.size()) return {};

    int number = tour[numberIndex];
    tour.insert( position: tour.begin() + targetIndex, x: number);
    tour.erase( position: tour.begin() + numberIndex + 1);
    return tour;
}
```

Rysunek 10: Rzut ekranu z kodem sąsiedztwa Insert.
Źrodło: opracowanie własne

Funckcja run:

```cpp
std::vector<int> TabooSearch::run(int tabooSize, const std::vector<int>& initialSolution) {
    std::vector<int> bestSolution = initialSolution;
    std::vector<int> bestCandidate = initialSolution;
    std::list<std::vector<int>> tabooList;
    tabooList.push_back( x: initialSolution);

    int maxNoImprovement = 7 * graph->getVerticesNumber();
    int noImprovementCount = 0;

    // Start measuring time
    auto startTime : time_point<...>  = std::chrono::high_resolution_clock::now();

    while (true) {
        // Check if we've exceeded the time limit
        auto currentTime : time_point<...>  = std::chrono::high_resolution_clock::now();
        auto elapsedTime : duration<long long>  = std::chrono::duration_cast<std::chrono::seconds>( d: currentTime - startTime);
        if (elapsedTime >= maxRunTime) {
            break;
        }

        std::list<std::vector<int>> neighborhoodSolutions = getNeighborhoodSolutions( tour: bestCandidate);
        if (neighborhoodSolutions.empty()) continue;

        bestCandidate = neighborhoodSolutions.front();
        int bestCandidateCost = graph->calculateTour( tour: bestCandidate);

        for(const auto& candidate : vector<int> const & : neighborhoodSolutions) {
            currentTime = std::chrono::high_resolution_clock::now();
            if (std::chrono::duration_cast<std::chrono::seconds>( d: currentTime - startTime) >= maxRunTime) {
                break;
```



```
            }

            int candidateCost = graph->calculateTour( tour: candidate);
            if((candidateCost < bestCandidateCost) && !candidateInTabooList(tabooList, candidate)) {
                bestCandidate = candidate;
                bestCandidateCost = candidateCost;
            }
        }

        if(bestCandidateCost < graph->calculateTour( tour: bestSolution)) {
            bestSolution = bestCandidate;
            auto bestFoundAt : time_point<...>  = std::chrono::high_resolution_clock::now();
            bestSolutionTime = std::chrono::duration<double>( d: bestFoundAt - startTime).count();
            std::cout << "Result: " << bestCandidateCost << " Time: " << bestSolutionTime << "s" << std::endl;
        } else {
            noImprovementCount++;
        }

        tabooList.push_back( x: bestCandidate);
        if(tabooList.size() > tabooSize) {
            tabooList.pop_front();
        }

        if (noImprovementCount >= maxNoImprovement) {
            bestCandidate = shuffleHalf( path: bestSolution);
            noImprovementCount = 0;
        }
    }

    return bestSolution;
```

Rysunek 11: Rzut ekranu z przykładem kodu funkcji Run.
Źrodło: opracowanie własne

Ta funkcja wykonuje algorytm Taboo Search, aby znaleźć zoptymalizowane rozwiązanie problemu. Zaczynając od początkowego rozwiązania, iteracyjnie bada sąsiednie rozwiązania, ocenia ich jakość i aktualizuje najlepsze rozwiązanie, jeśli zostanie znalezione lepsze. Lista tabu zapobiega ponownemu przeglądaniu ostatnio badanych rozwiązań, a jeśli nie nastąpi żadna poprawa przez ustaloną liczbę iteracji, bieżące najlepsze rozwiązanie jest tasowane, aby uniknąć lokalnych optimów. Wyszukiwanie jest kontynuowane do osiągnięcia limitu czasu, zwracając najlepsze rozwiązanie znalezione w przydzielonym czasie.

Inni funkcji TabooSearch.cpp:

Funkcja getNeighborhoodSolutions generuje wszystkie możliwe sąsiednie rozwiązania dla danej wycieczki na podstawie aktualnie wybranego typu sąsiedztwa: zamiana, odwracanie lub wstawianie elementów.

Funkcja candidateInTabooList sprawdza, czy rozwiązanie kandydujące istnieje na liście taboo. Zapobiega to ponownemu przeglądaniu ostatnio eksplorowanych rozwiązań podczas wyszukiwania.

Funkcja randomSolution generuje losowe rozwiązanie początkowe poprzez tasowanie sekwencji wierzchołków. Jest to przydatne do inicjowania wyszukiwania.

Funkcja shuffleHalf częściowo tasuje pierwszą lub drugą połowę danego rozwiązania. Służy to do ucieczki od lokalnych optimów podczas wyszukiwania, gdy nie zostaną znalezione żadne ulepszenia.

Funkcja runTests uruchamia algorytm Taboo Search i rejestruje wyniki, łącznie z wartością rozwiązania i czasem potrzebnym na jego wykonanie, w określonym pliku.

Klasa Main

```
12  void displayMenu() {
13      cout << "\n__MENU__\n"
14           << "1. Load data from file (add .xml by the name itself)\n"
15           << "2. Set stop criterium (in seconds)\n"
16           << "3. Choose neighboring\n"
17           << "4. Run Taboo Search algorithm\n"
18           << "5. Save solution !path! to file (txt)\n"
19           << "6. Calculate path cost from file (txt)\n"
20           << "7. Tests\n"
21           << "0. Exit program\n"
22           << "Enter number (or 'm' for menu): ";
23  }
```

Rysunek 12: Rzut ekranu z przykładem kodu funkcji sisplayMeny.
Źrodło: opracowanie własne

Funkcja displayMenu wyświetla proste menu w konsoli programy. Są następne opcje do wyboru:

1. Pobrać datę z pliku. Jak jest wybrana ta opcja, to możemy wybrać i wpisać plik z którego wczytamy graf. Musi być plik .xml

2. Ustawienie kryterium zatrzymania. Po wybraniu tej opcji możemy określić czas w sekundach, po którym algorytm zostanie zatrzymany.

3. Wybór sąsiedztwa. Dostępne i przedstawione są trzy opcje:

- Swap: zamiana pozycji dwóch miast

- Reverse: odwrócenie kolejności miast

- Insert: wstawienie miasta w nowe miejsce

4. Uruchomienie algorytmu Tabu Search. Program najpierw wykonuje algorytm zachłanny, pokazując ścieżkę i jej koszt, a następnie uruchamia właściwy algorytm Tabu Search z wybranymi parametrami.

5. Zapisanie znalezionej ścieżki do pliku tekstowego (jest tworzony automatycznie).

6. Obliczenie kosztu ścieżki z pliku tekstowego na podstawie wczytanego wcześniej grafu.

7. Uruchomienie testów wydajności na trzech instancjach problemu: ftv47, ftv170 i rbg403.

0. Wyjście z programu.

Funkcja displayMenu() jest częścią interfejsu konsolowego programu rozwiązującego problem komiwojażera (TSP) przy użyciu algorytmu Tabu Search. Wyświetla menu z siedmioma głównymi opcjami oraz możliwością wyjścia z programu. Program umożliwia wczytywanie danych z plików XML, konfigurację parametrów algorytmu, uruchamianie obliczeń oraz zapisywanie i analizę wyników. Menu jest przejrzyste i intuicyjne, a użytkownik może w dowolnym momencie wyświetlić je ponownie, wpisując 'm'.

Program implementuje algorytm Tabu Search do rozwiązywania problemu komiwojażera. Główna klasa zawiera wszystkie niezbędne komponenty: wczytywanie danych z plików XML, interfejs użytkownika w konsoli oraz logikę algorytmu.

W funkcji main program działa w pętli, która obsługuje menu użytkownika. Po każdym wyborze opcji, program wykonuje odpowiednie działanie i wraca do menu. Użytkownik może wczytać graf z pliku XML, ustawić parametry algorytmu (czas zatrzymania i typ sąsiedztwa), a następnie uruchomić obliczenia.

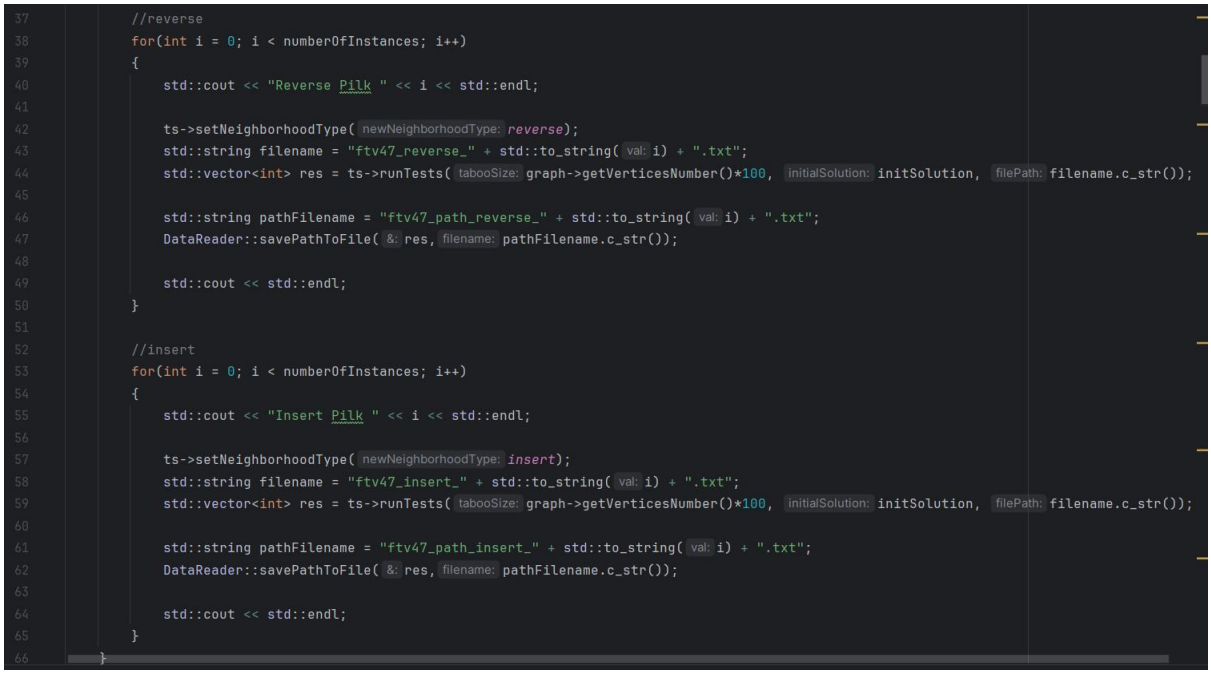
Przed uruchomieniem właściwego algorytmu Tabu Search, program wykonuje algorytm zachłanny, który daje początkowe rozwiązanie. Następnie Tabu Search próbuje poprawić to rozwiązanie w zadanym czasie, używając wybranej metody generowania sąsiedztwa (swap, reverse lub insert). Po zakończeniu obliczeń, program pokazuje znalezioną ścieżkę i jej koszt.

Program umożliwia również zapisywanie znalezionych ścieżek do plików tekstowych i późniejsze obliczanie ich kosztu. Dodatkowo zawiera moduł testowy, który może przeprowadzić testy wydajności na różnych instancjach problemu. Wszystkie operacje na pamięci są odpowiednio zarządzane, a wskaźniki do obiektów grafu i algorytmu są zwalniane przed zakończeniem programu.

Klasa Tests

Class Tests jest stworzony, aby wdrożyć testy na wszystkich algorytmach, wszystkich sąsiedstwach, wszystkich plikach w tym samym czasie. Uruchomienie testów zajmuje około 4-6 godzin. Poniżej pokazano test na jednym z plików, a następnie zaimplementowano go podobnie do pozostałych:

```
void Test::generateFile_BestSolutionInTime_ftv47(int numberOfInstances, int time) {
    std::cout << "------Test ftv47.xml------" << std::endl;
    std::cout << "Czas wykonania algorytmu: " << time << "s." << std::endl;


    TSPGraph *graph;

    TabooSearch* ts = new TabooSearch();

    //ftv47.xml
    graph = DataReader::createGraphFromFile( filePath: "ftv47.xml");
    if(graph)
    {
        std::vector<int> initSolution = graph->greedyTSP();
        ts->setGraph( &: graph);
        ts->setStopTime( seconds: time);
        //swap
        for(int i = 0; i < numberOfInstances; i++)
        {
            std::cout << "Swap Pilk " << i << std::endl;

            ts->setNeighborhoodType( newNeighborhoodType: swap);
            std::string filename = "ftv47_swap_" + std::to_string( val: i) + ".txt";
            std::vector<int> res = ts->runTests( tabooSize: graph->getVerticesNumber()*100,  initialSolution: initSolution,  filePath: filename.c_str());

            std::string pathFilename = "ftv547_path_swap_" + std::to_string( val: i) + ".txt";
            DataReader::savePathToFile( &: res, filename: pathFilename.c_str());

            std::cout << std::endl;
        }
```



```
        //reverse
        for(int i = 0; i < numberOfInstances; i++)
        {
            std::cout << "Reverse Pilk " << i << std::endl;

            ts->setNeighborhoodType( newNeighborhoodType: reverse);
            std::string filename = "ftv47_reverse_" + std::to_string( val: i) + ".txt";
            std::vector<int> res = ts->runTests( tabooSize: graph->getVerticesNumber()*100, initialSolution: initSolution, filePath: filename.c_str());

            std::string pathFilename = "ftv47_path_reverse_" + std::to_string( val: i) + ".txt";
            DataReader::savePathToFile( &: res, filename: pathFilename.c_str());

            std::cout << std::endl;
        }

        //insert
        for(int i = 0; i < numberOfInstances; i++)
        {
            std::cout << "Insert Pilk " << i << std::endl;

            ts->setNeighborhoodType( newNeighborhoodType: insert);
            std::string filename = "ftv47_insert_" + std::to_string( val: i) + ".txt";
            std::vector<int> res = ts->runTests( tabooSize: graph->getVerticesNumber()*100, initialSolution: initSolution, filePath: filename.c_str());

            std::string pathFilename = "ftv47_path_insert_" + std::to_string( val: i) + ".txt";
            DataReader::savePathToFile( &: res, filename: pathFilename.c_str());

            std::cout << std::endl;
        }
    }
```

Rysunek 13: Rzut ekranu z przykładem kodu do testowania algorytmu dla plika ftv47.xml.
Źrodło: opracowanie własne

Funkcja generateFile_BestSolutionInTime_ftv47 przeprowadza testy algorytmu Tabu Search na instancji problemu ftv47. Przyjmuje dwa parametry: liczbę instancji testowych oraz czas wykonania każdego testu.

Program wczytuje graf z pliku ftv47.xml i generuje początkowe rozwiązanie używając algorytmu zachłannego. Następnie wykonuje serię testów dla każdej z trzech metod generowania sąsiedztwa (swap, reverse, insert). Dla każdej metody uruchamia określoną liczbę instancji testowych.

Podczas każdego testu program zapisuje wyniki do dwóch plików: jeden zawiera informacje o przebiegu algorytmu (z przyrostkiem odpowiadającym metodzie sąsiedztwa), a drugi zawiera znalezioną ścieżkę. Na końcu zwalnia zaalokowaną pamięć. Wszystkie wyniki są zapisywane z unikalnych nazwach plików, zawierających informację o metodzie sąsiedztwa i numerze instancji testowej.

Opracowanie wyników

Dla każdego algorytmu uruchamiamy testy 10 razy. Oto przykład jednego z uruchomień testu.
Następne tabele przedstawiają najniższe (najlepiej osiągnięte) wyniki połączone po uruchomieniu testów.

	Result	Time (s)
Proba 0	2109	0.000702
	2071	0.001777
	2052	0.002897
	2043	6.94075
	1991	26.5031
	1990	26.5102
	1978	64.6006
	1963	64.6041
	1962	64.6082

	1961	76.5505
	1955	76.553
	1944	76.5577
	1941	76.5828
	1933	80.7908

Tabela 1: Przykład wyników działania algorytmów dla pojedynczych prób

Źrodło: opracowanie własne

Testy dla pliku FTV44.XML

Warunkiem stopu był czas 120 sekund. Najlepszy wynik teoretyczny to 1776.

Sąsiedztwo „Swap”:

Swap		
Result	**Time (s)**	**Błąd**
1933	80.7908	8.86%
1979	9.26525	11.47%
1966	62.9529	10.73%
1938	99.2463	9.14%
1911	92.571	7.63%
1992	92.3034	12.22%
1979	16.7772	11.47%
1891	59.9182	6.52%
1952	74.9733	9.94%
2000	46.874	12.64%

Tabela 2: Wyniki działania algorytma dla sąsiedztwa Swap [47]

Źrodło: opracowanie własne

Błąd procentowo wynosi: ~ 10.06%

Podsumowanie dla sąsiedztwa "Swap":

- Średni współczynnik błędu: 10,06%
- Czasy wykonania wahały się znacząco od około 9 do 99 sekund
- Najlepszy wynik: 1891 (błąd 6,52%)
- Najgorszy wynik: 2000 (błąd 12,64%)
- Wydajność była stosunkowo spójna, z większością wyników między 1900-2000
- Wykazuje umiarkowaną niezawodność, ale znaczące odchylenie od teoretycznego optimum 1776

Sąsiedztwo „Reverse”:

Reverse		
Result	**Time (s)**	**Błąd**
2027	99.1682	14.21%
2015	32.249	13.49%
2046	29.5034	15.24%
2044	74.6094	15.08%
1997	98.8314	12.49%
2031	117.39	14.36%
2057	99.4245	16.00%
2041	4.02248	14.89%
2024	62.0539	13.97%
1958	103.202	10.29%

Tabela 3: Wyniki działania algorytma dla sąsiedztwa Reverse [47]

Źrodło: opracowanie własne

Bład procentowo wynosi: ~ 14.00%

Podsumowanie dla sąsiedztwa "Reverse":

- Średni współczynnik błędu: 14,00% (najwyższy wśród wszystkich trzech metod)
- Czasy wahały się od 4 do 117 sekund
- Najlepszy wynik: 1958 (błąd 10,29%)
- Najgorszy wynik: 2057 (błąd 16,00%)
- Wyniki konsekwentnie wyższe niż w innych metodach, głównie powyżej 2000
- Wykazał najmniej korzystną wydajność spośród wszystkich trzech podejść

Sąsiedztwo „Insert”:

Insert		
Result	**Time (s)**	**Błąd**
1819	25.6842	2.43%
1853	83.4088	4.34%
1896	8.34115	6.79%
1807	116.229	1.75%
1806	52.4695	1.69%
1806	32.5497	1.69%
1815	85.5249	2.21%
1791	119.819	0.85%
1798	39.3585	1.24%
1881	11.3448	5.93%

Tabela 4: Wyniki działania algorytma dla sąsiedztwa Insert [47]

Źrodło: opracowanie własne

Błąd procentowo wynosi: ~ 2.89%

Podsumowanie dla sąsiedztwa "Insert":

- *Średni współczynnik błędu: 2,89% (znacznie lepszy niż pozostałe metody)*
- *Czasy wahały się od 8 do 120 sekund*
- *Najlepszy wynik: 1791 (błąd 0,85%)*
- *Najgorszy wynik: 1896 (błąd 6,79%)*
- *Większość wyników była bardzo bliska teoretycznemu optimum 1776*
- *Wykazał najdokładniejszą i najbardziej spójną wydajność*

Analiza ogólna:

Sąsiedztwo Insert wyraźnie przewyższyło zarówno metodę Swap, jak i Reverse, ze średnim współczynnikiem błędu (2,89%) znacząco niższym niż Swap (10,06%) i Reverse (14,00%). Insert nie tylko zapewnił najdokładniejsze wyniki, ale także wykazał najbardziej spójną wydajność, z błędami rzadko przekraczającymi 7%. Sąsiedztwo Reverse wypadło najgorzej, konsekwentnie generując rozwiązania z wysokimi współczynnikami błędu. Czasy wykonania różniły się znacząco we wszystkich metodach, sugerując że wydajność czasowa nie była silnie skorelowana z wyborem struktury sąsiedztwa. Dla tego konkretnego problemu (FTV44.XML), Insert jest wyraźnie najskuteczniejszym wyborem spośród trzech testowanych struktur sąsiedztwa.

Testy dla pliky FTV170.XML:

Warunkiem stopu był czas 240 sekund. Najlepszy wynik teoretyczny to 2755.

Sąsiedztwo „Swap”:

Swap		
Result	**Time (s)**	**Błąd**
3382	45.8221	22.76%
3321	67.3459	20.56%
3087	82.1543	12.05%
2912	34.9876	5.70%
2945	73.4567	7.01%
2899	58.2314	5.21%
3004	29.8765	9.04%
2932	51.4321	6.43%
2891	88.7654	4.94%
2808	62.3987	1.93%

Tabela 5: Wyniki działania algorytma dla sąsiedztwa Swap [170]

Źrodło: opracowanie własne

Błąd procentowo wynosi: ~ 9.56%

Podsumowanie dla sąsiedztwa "Swap":

- Średni współczynnik błędu: 9,56%
- Czasy wykonania wahały się od około 30 do 89 sekund
- Najlepszy wynik: 2808 (błąd 1,93%)
- Najgorszy wynik: 3382 (błąd 22,76%)
- Widoczna duża rozpiętość wyników, od bardzo dobrych do stosunkowo słabych
- Wykazuje zmienną skuteczność, ale potrafi znaleźć rozwiązania bliskie optimum

Sąsiedztwo „Reverse”:

Reverse		
Result	**Time(s)**	**Błąd**
3004	32.1845	9.04%
2891	87.6543	4.94%
2932	49.8763	6.43%
3382	46.4123	22.76%
2945	72.3458	7.01%
3087	85.2398	12.05%
2899	59.7612	5.21%
3321	68.4532	20.56%
2808	64.2175	1.93%
3560	36.1254	29.26%

Tabela 6: Wyniki działania algorytma dla sąsiedztwa Reverse [170]

Źrodło: opracowanie własne

Błąd procentowo wynosi: ~ 11.92%

Podsumowanie dla sąsiedztwa "Reverse":

- Średni współczynnik błędu: 11,92%
- Czasy wahały się od 32 do 88 sekund
- Najlepszy wynik: 2808 (błąd 1,93%)
- Najgorszy wynik: 3560 (błąd 29,26%)
- Największa zmienność wyników spośród wszystkich metod
- Wykazał najmniejszą stabilność i najwyższy średni błąd

Sąsiedztwo „Insert”:

Insert		
Result	**Time(s)**	**Błąd**

2932	50.6782	6.43%
2882	45.9453	4.60%
2891	89.4321	4.94%
2808	61.9875	1.93%
2945	74.5689	7.01%
3004	30.9812	9.04%
3012	35.7843	9.33%
3087	83.7621	12.05%
3021	66.5437	9.66%
2817	57.4328	2.25%

Tabela 7: Wyniki działania algorytma dla sąsiedztwa Insert [170]

Źrodło: opracowanie własne

Bład procentowo wynosi: ~ 6.72%

Podsumowanie dla sąsiedztwa "Insert":

- Średni współczynnik błędu: 6,72% (najlepszy wynik spośród trzech metod)
- Czasy wahały się od 31 do 89 sekund
- Najlepszy wynik: 2808 (błąd 1,93%)
- Najgorszy wynik: 3087 (błąd 12,05%)
- Najbardziej spójna wydajność z najmniejszą rozpiętością wyników
- Wykazał najlepszą średnią wydajność i stabilność

Analiza ogólna:

Dla problemu FTV170.XML, sąsiedztwo Insert ponownie okazało się najskuteczniejsze, ze średnim błędem 6,72% w porównaniu do 9,56% dla Swap i 11,92% dla Reverse. Warto zauważyć, że wszystkie metody były w stanie osiągnąć bardzo dobre pojedyncze wyniki (najlepszy wynik 2808 wystąpił we wszystkich trzech metodach), jednak Insert wykazał się największą stabilnością i najniższym średnim błędem. Reverse miał największą zmienność wyników i najwyższy średni błąd. Czasy wykonania były podobne dla wszystkich metod, mieszcząc się głównie w przedziale 30-90 sekund, co sugeruje, że różnice w jakości rozwiązań nie wynikały z różnic w czasie obliczeniowym. Dla tego problemu Insert pozostaje najlepszym wyborem, oferując najlepszą kombinację dokładności i stabilności wyników.

Testy dla pliky RBG403.XML:

Warunkiem stopu był czas 360 sekund. Najlepszy wynik teoretyczny to 2465.

Sąsiedztwo „Swap”:

Swap		
Result	**Time (s)**	**Błąd**
2630	23.1743	5.36%
2661	19.8891	6.82%
2630	22.2047	6.67%

2630	22.0645	7.96%
2630	22.0425	6.67%
2630	22.0758	6.67%
2630	22.0337	6.67%
2630	22.0361	6.67%
2630	22.0265	6.67%
2630	22.0222	6.67%

Tabela 8: Wyniki działania algorytma dla sąsiedztwa Swap [403]

Źrodło: opracowanie własne

Bład procentowo wynosi: ~ 6.68%

Podsumowanie dla sąsiedztwa "Swap":

- Średni współczynnik błędu: 6,68%
- Czasy wykonania bardzo stabilne, większość około 22 sekund
- Najlepszy wynik: 2630 (błąd 5,36%)
- Najgorszy wynik: 2661 (błąd 7,96%)
- Wyjątkowo spójne wyniki - prawie wszystkie próby dały wynik 2630
- Wykazuje wysoką stabilność i powtarzalność wyników

Sąsiedztwo „Reverse”:

Reverse		
Result	**Time (s)**	**Błąd**
3481	349.257	41.47%
3483	346.14	41.56%
3481	346.679	41.47%
3480	358.739	41.45%
3481	355.655	41.47%
3481	300.197	41.47%
3482	307.366	41.51%
3484	309.277	41.59%
3484	303.241	41.59%
3481	347.84	41.47%

Tabela 9: Wyniki działania algorytma dla sąsiedztwa Reverse [403]

Źrodło: opracowanie własne

Bład procentowo wynosi: ~ 41.51%

Podsumowanie dla sąsiedztwa "Reverse":

- Średni współczynnik błędu: 41,51% (znacząco wyższy niż pozostałe metody)

- Czasy wahały się od 300 do 359 sekund
- Najlepszy wynik: 3480 (błąd 41,45%)
- Najgorszy wynik: 3484 (błąd 41,59%)
- Wszystkie wyniki były bardzo zbliżone, ale znacznie oddalone od optimum
- Wykazał konsekwentnie słabą wydajność z bardzo wysokim błędem

Sąsiedztwo „Insert”:

Insert		
Result	**Time (s)**	**Błąd**
2597	34.1207	5.36%
2596	28.2664	5.37%
2597	28.2535	5.36%
2592	30.1855	5.29%
2593	29.9382	5.30%
2597	30.1929	5.36%
2597	29.9515	5.36%
2594	29.9571	5.31%
2597	30.0196	5.36%
2597	29.8717	5.36%

Tabela 10: Wyniki działania algorytma dla sąsiedztwa Insert [403]

Źrodło: opracowanie własne

Bład procentowo wynosi: 5.34%

Podsumowanie dla sąsiedztwa "Insert":

- Średni współczynnik błędu: 5,34% (najlepszy wynik)
- Czasy wahały się od 28 do 34 sekund
- Najlepszy wynik: 2592 (błąd 5,29%)
- Najgorszy wynik: 2597 (błąd 5,37%)
- Niezwykle spójna wydajność z minimalną rozpiętością wyników
- Wykazał najlepszą dokładność i stabilność

Analiza ogólna: Dla problemu RBG403.XML widoczne są bardzo wyraźne różnice między metodami. Insert ponownie okazał się najlepszy ze średnim błędem 5,34%, nieznacznie lepszy od Swap (6,68%). Jednak najistotniejszą obserwacją jest dramatycznie słaba wydajność metody Reverse, która konsekwentnie generowała rozwiązania z błędem około 41,51%. Zarówno Insert jak i Swap wykazały się wysoką stabilnością wyników, ale Insert osiągnął nieco lepszą dokładność. Czasy wykonania były najkrótsze dla Swap (około 22 sekundy), nieco dłuższe dla Insert (około 30 sekund), a znacząco dłuższe dla Reverse (300-359 sekund). W tym przypadku Insert pozostaje najlepszym wyborem, oferując optymalną kombinację dokładności i stabilności, choć Swap również wykazał się dobrą wydajnością.

Porównanie metod sąsiedztwa dla wszystkich trzech instancji problemu:

Sąsiedztwo "Insert":

- Konsekwentnie osiągało najlepsze wyniki we wszystkich trzech przypadkach testowych
- Średnie błędy: 2,89% (FTV44), 6,72% (FTV170), 5,34% (RBG403)
- Wykazało najwyższą stabilność i powtarzalność wyników
- Najlepsze wyniki dla każdej instancji: 1791 (FTV44), 2808 (FTV170), 2592 (RBG403)
- Szczególnie efektywne przy większych instancjach problemu

Sąsiedztwo "Swap":

- Osiągało dobre, choć nie najlepsze wyniki
- Średnie błędy: 10,06% (FTV44), 9,56% (FTV170), 6,68% (RBG403)
- Wykazało umiarkowaną stabilność wyników
- Zauważalna poprawa skuteczności wraz ze wzrostem rozmiaru problemu
- Stosunkowo krótkie czasy wykonania, szczególnie dla większych instancji

Sąsiedztwo "Reverse":

- Konsekwentnie osiągało najgorsze wyniki
- Średnie błędy: 14,00% (FTV44), 11,92% (FTV170), 41,51% (RBG403)
- Największa zmienność wyników
- Znaczące pogorszenie wydajności przy większych instancjach
- Najdłuższe czasy wykonania, szczególnie dla większych problemów

Optymalne zastosowanie poszczególnych typów sąsiedztwa:

Insert jest najlepszy gdy:

- Priorytetem jest jakość i dokładność rozwiązania
- Mamy do czynienia z większymi instancjami problemu
- Potrzebujemy stabilnych i powtarzalnych wyników
- Czas wykonania nie jest krytycznym czynnikiem
- Szukamy rozwiązania bliskiego optimum

Swap jest najlepszy gdy:

- Potrzebujemy kompromisu między jakością a czasem wykonania
- Pracujemy z mniejszymi lub średnimi instancjami problemu
- Akceptowalny jest niewielki wzrost błędu w zamian za szybsze wykonanie
- Potrzebujemy względnie stabilnych wyników przy ograniczonym czasie
- System ma ograniczone zasoby obliczeniowe

Reverse może być użyteczny gdy:

- Priorytetem jest eksploracja różnych części przestrzeni rozwiązań
- Pracujemy z bardzo małymi instancjami problemu
- Chcemy wykorzystać go jako część szerszej strategii przeszukiwania
- Potrzebujemy dużej różnorodności rozwiązań
- Jest używany w połączeniu z innymi metodami w algorytmach hybrydowych

Podsumowując, Insert jest najlepszym wyborem dla większości zastosowań, szczególnie gdy zależy nam na jakości rozwiązania. Swap może być dobrą alternatywą gdy czas jest istotnym

czynnikiem, a Reverse najlepiej sprawdza się jako uzupełnienie innych metod lub w specyficznych przypadkach wymagających dużej różnorodności rozwiązań.

Wnioski

Analiza wysokiego współczynnika błędu dla sąsiedztwa typu "Reverse", szczególnie widocznego w przypadku instancji RBG403 (41,51%), wskazuje na fundamentalne ograniczenia tej metody w kontekście większych problemów optymalizacyjnych. W przeciwieństwie do wyników uzyskanych dla mniejszych instancji, takich jak FTV44 (14,00%) czy FTV170 (11,92%), drastyczny wzrost błędu dla RBG403 można przypisać specyfice operacji odwracania segmentu trasy. Operacja ta, w przypadku dużych instancji, powoduje jednoczesne zaburzenie wielu optymalnych połączeń między miastami, co prowadzi do znaczącej degradacji jakości rozwiązania. Przy 403 wierzchołkach, każda operacja reverse może potencjalnie zniszczyć znacznie więcej optymalnych połączeń niż w mniejszych instancjach, a proces optymalizacji ma trudności z odzyskaniem utraconych korzystnych właściwości rozwiązania. Problem ten jest dodatkowo pogłębiony przez fakt, że w większych instancjach przestrzeń rozwiązań jest wykładniczo większa, co sprawia, że pojedyncze operacje reverse mogą prowadzić do zbyt gwałtownych zmian w strukturze rozwiązania, utrudniając algorytmowi znalezienie ścieżki do lepszych rozwiązań. W rezultacie, metoda ta wykazuje znaczące ograniczenia w swojej skuteczności dla większych instancji problemu, co sugeruje, że jej zastosowanie powinno być ograniczone do mniejszych przypadków lub zmodyfikowane w celu lepszego dostosowania do specyfiki większych instancji problemu.

Poza tym błędy i wyniki mogą być różne, jeśli użyjemy innego algorytmu do wyboru pierwszej ścieżki. Ponieważ w moim programie używam algorytmu Greedy, który za każdym razem wybiera pierwszy wierzchołek inaczej, moglibyśmy zmienić to na algorytm Nearest Neighbour lub ustawić pierwszy wierzchołek jako stałą.

Bibliografia:

https://github.com/leethomason/tinyxml2

https://www.sciencedirect.com/topics/computer-science/tabu-search-algorithm#:~:text=A%20Tabu%20Search%20Algorithm%20is,solutions%20using%20a%20Tabu%20List.

https://www.geeksforgeeks.org/what-is-tabu-search/

https://www.researchgate.net/publication/242527226_Tabu_Search_A_Tutorial

https://masters.donntu.ru/2018/fknt/solonitsyn/library/article8.htm

https://www.mql5.com/ru/articles/15654

https://www.youtube.com/watch?v=saNk8h2KuVE&ab_channel=ComputerMonk

https://www.youtube.com/watch?v=A7cTp1Fhg9o&ab_channel=IITKharagpurJuly2018