Projektowanie Efektywnych Algorytmów

Projekt 2

Katsiaryna Kolyshko 276708

Dr. inż. Marcin Łapuszyński

Wstęp teorytyczny

1. Branch and Bound (B&B)

Branch and Bound to algorytm optymalizacji używany do rozwiązywania problemów kombinatorycznych, takich jak problem komiwojażera, problem plecaka i inne problemy, w których przestrzeń rozwiązań musi być efektywnie eksplorowana. Pomysł B&B polega na systematycznym eksplorowaniu wszystkich możliwych rozwiązań poprzez „rozgałęzianie" się od rozwiązań częściowych. Używa funkcji ograniczającej do przycinania gałęzi, które nie mogą prowadzić do lepszego rozwiązania niż obecne najlepsze rozwiązanie, unikając w ten sposób wyczerpującej eksploracji wszystkich możliwości.

Jak działa algorytm:

Branching: Podział problemu na mniejsze podproblemy.

Bounding: Obliczanie granicy (zwykle dolnej lub górnej) dla każdej gałęzi w celu oszacowania najlepszego możliwego rozwiązania w ramach tej gałęzi. Jeśli granica jest gorsza niż obecne najlepsze rozwiązanie, gałąź ta jest odrzucana.

Purning: Ten proces pomaga uniknąć niepotrzebnych obliczeń poprzez wczesne odrzucanie suboptymalnych ścieżek.

Złożoność czasowa:

Złożoność czasowa B&B zależy w dużym stopniu od funkcji ograniczającej i struktury problemu. W najgorszym przypadku może to zająć czas wykładniczy *(O(2^n) dla problemów takich jak TSP)*, ale przycinanie znacznie zmniejsza liczbę eksplorowanych węzłów w praktyce.

Złożoność pamięci:

B&B wymaga pamięci do przechowywania podproblemów, które w najgorszym przypadku mogą być dość duże (*O(n) lub O(2^n)* w zależności od rozmiaru problemu i współczynnika rozgałęzienia).

Najlepsze scenariusze użycia:

B&B najlepiej sprawdza się w przypadku problemów wymagających znalezienia optymalnego rozwiązania, takich jak problemy TSP, plecakowe i programowania całkowitoliczbowego.

B&B działa dobrze, gdy problem ma dobrą heurystykę lub ograniczenie do skutecznego przycinania gałęzi.

2. Przeszukiwanie wszerz (BFS)

Przeszukiwanie wszerz (BFS) jest podstawowym algorytmem przeszukiwania grafu. Zaczyna się od węzła, a następnie przechodzi przez wszystkie sąsiednie. Po odwiedzeniu wszystkich sąsiednich węzłów, przechodzi się przez ich sąsiednie węzły. Różni się to od DFS tym, że najbliższe wierzchołki są odwiedzane przed innymi. Przeszukujemy głównie wierzchołki poziom po poziomie. Wiele popularnych algorytmów grafowych, takich jak najkrótsza ścieżka Dijkstry , algorytm Kahna i algorytm Prima , opiera się na BFS. Sam BFS może być używany do wykrywania cyklu w grafie skierowanym i nieskierowanym, znajdowania najkrótszej ścieżki w grafie nieważonym i wielu innych problemów.

Jak działa algorytm:

BFS zaczyna od węzła głównego i eksploruje wszystkich sąsiadów przed przejściem do następnego poziomu. Używa kolejki do śledzenia węzłów do eksploracji jako następnych, zapewniając, że węzły są eksplorowane poziom po poziomie.

Złożoność czasowa:

O(V + E), gdzie V to liczba wierzchołków, a E to liczba krawędzi. BFS bada wszystkie wierzchołki i krawędzie w najgorszym przypadku.

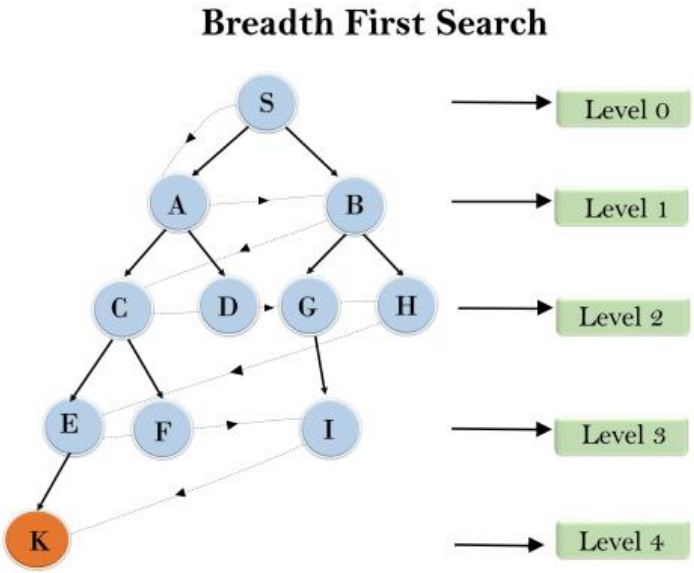
Złożoność pamięci:

O(V), ponieważ musi przechowywać wszystkie węzły na bieżącym poziomie w kolejce.

Najlepsze scenariusze użycia:

BFS jest optymalne, gdy celem jest znalezienie najkrótszej ścieżki w nieważonym grafie, ponieważ eksploruje węzły w coraz większej odległości od węzła początkowego.

BFS jest idealne, gdy trzeba przetwarzać węzły poziom po poziomie, na przykład podczas przechodzenia drzewa w kolejności poziomów.



Rysunek 1: Przykład działania algorytmu BFS

Źródło: www.javatpoint.com

3. Przeszukiwanie w głąb (DFS)

DFS to algorytm używany do eksploracji węzłów i krawędzi grafu lub drzewa. W przeciwieństwie do BFS, który najpierw eksploruje wszystkich sąsiadów węzła, DFS eksploruje potomków węzła przed cofaniem się w celu eksploracji innych gałęzi.

Jak działa algorytm:

DFS zaczyna od węzła głównego i eksploruje tak daleko w dół jednej gałęzi, jak to możliwe, przed cofaniem się.

Używa stosu (jawnie lub za pośrednictwem rekurencji), aby śledzić węzły do eksploracji.

Złożoność czasowa:

O(V + E), gdzie V to liczba wierzchołków, a E to liczba krawędzi. DFS bada również wszystkie wierzchołki i krawędzie w najgorszym przypadku.

Złożoność pamięci:

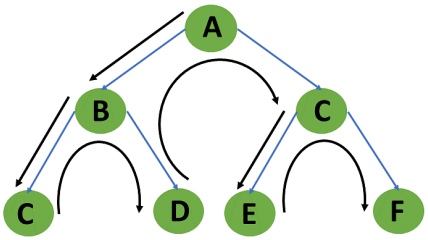
O(V) w najgorszym przypadku, ponieważ DFS musi przechowywać wszystkie węzły w bieżącej ścieżce.

Najlepsze scenariusze wykorzystania:

DFS jest przydatny, gdy przestrzeń rozwiązań jest głęboka lub ma wiele gałęzi, a celem jest eksploracja jak największej ilości wzdłuż jednej gałęzi przed cofnięciem się.

DFS można stosować do sortowania topologicznego w skierowanych grafach acyklicznych (DAG).

DFS może pomóc zidentyfikować wszystkie węzły połączone z danym węzłem.



Rysunek 2: Przykład działania algorytmu DFS

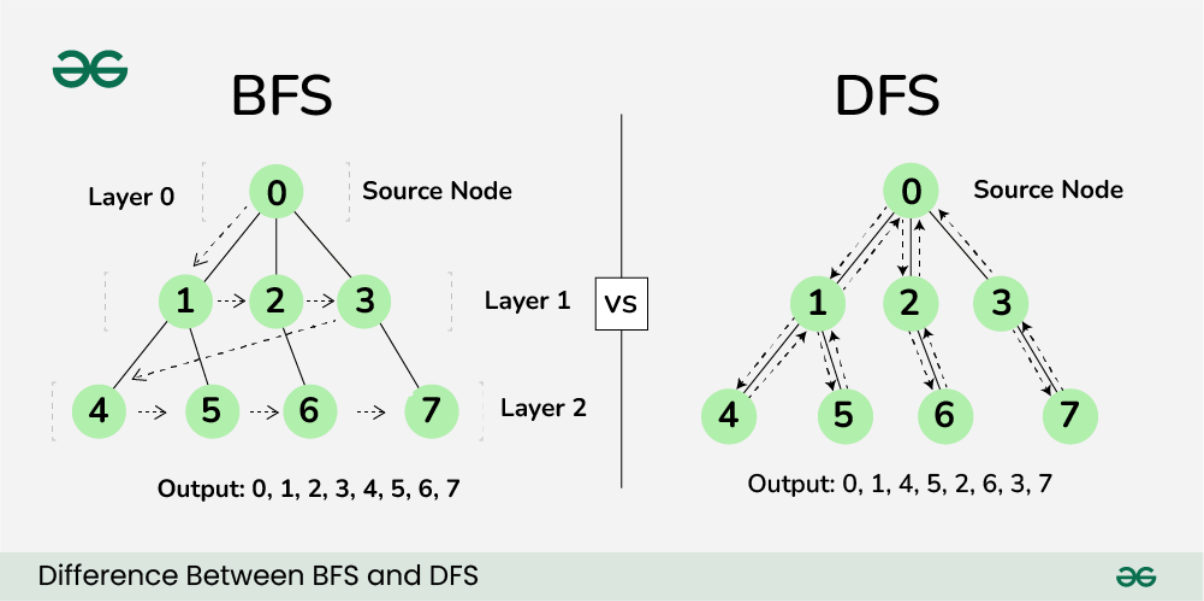
Źródło: www.simplilearn.com

Algorytmy BFS i DFS często są porównane pomiędzy sobą. W czym jest różnica pomiędzy nimi?

BFS i DFS to podstawowe algorytmy przemierzania grafu, ale eksplorują graf na różne sposoby. BFS eksploruje graf poziom po poziomie, zaczynając od węzła głównego i rozszerzając się na wszystkie sąsiednie węzły, zanim przejdzie do następnego poziomu. Używa kolejki do przechowywania węzłów, które należy zbadać, zapewniając, że węzły są eksplorowane w kolejności, w jakiej zostały odkryte. To podejście gwarantuje najkrótszą ścieżkę w nieważonym grafie, ponieważ eksploruje wszystkie węzły w odległości d od początku, zanim przejdzie do odległości (d+1). BFS jest również lepiej przystosowany do problemów, takich jak znalezienie najkrótszej ścieżki lub rozwiązywanie zagadek, w których trzeba rozważyć wszystkie możliwe rozwiązania warstwa po warstwie.

Z drugiej strony DFS eksploruje tak głęboko, jak to możliwe wzdłuż gałęzi, zanim cofnie się, aby zbadać inne gałęzie. Używa stosu (jawnie lub za pośrednictwem rekurencji), aby śledzić węzły, które mają zostać zbadane. DFS nie gwarantuje najkrótszej ścieżki i może utknąć w głębokich, ale nieistotnych gałęziach. Jest bardziej wydajny pod względem pamięci niż BFS w przypadku rzadkich grafów, ponieważ przechowuje tylko węzły wzdłuż bieżącej ścieżki. DFS jest lepiej przystosowany do problemów, w których trzeba zbadać wszystkie węzły w określonej gałęzi przed przejściem dalej, takich jak znajdowanie spójnych komponentów lub wykonywanie sortowania topologicznego w skierowanych grafach acyklicznych. DFS można również używać do rozwiązywania problemów z głęboką rekurencją,

ale może nie działać dobrze w przypadku bardzo dużych lub złożonych grafów z powodu potencjalnych problemów, takich jak utknięcie w nieskończonych pętlach bez prawidłowego wykrywania cyklu.



Rysunek 3: Porównanie działania algorytmów BFS i DFS

Źródło: www.geeksforgeeks.org

4. Algorytm Najniższego Kosztu

Algorytm Najniższego Kosztu to systematyczna strategia wyszukiwania zaprojektowana do eksploracji grafu lub drzewa, koncentrująca się na ścieżkach o najniższym dotychczasowym skumulowanym koszcie. W przeciwieństwie do metod opartych na heurystyce, priorytetowo traktuje ścieżki wyłącznie na podstawie ich aktualnego kosztu, bez polegania na szacunkach przyszłych kosztów.

Jak działa algorytm:

Algorytm Najniższego Kosztu wykorzystuje kolejkę priorytetową (lub min-kopiec), aby zawsze wybierać węzeł z najniższym skumulowanym kosztem. Rozważa wszystkie możliwe ścieżki do następnych węzłów, dodaje koszty przejścia i umieszcza je w kolejce priorytetowej. Stopniowo eksploruje graf w kierunku celu, upewniając się, że rozważa najpierw ścieżki o najniższym koszcie.

Złożoność czasowa:

O(V log V + E), gdzie V to liczba wierzchołków, a E to liczba krawędzi. Dominującą częścią obliczeń są operacje kolejki priorytetowej.

Złożoność pamięci:

O(V), ponieważ algorytm przechowuje wszystkie odwiedzone węzły oraz koszty związane z ich eksploracją.

Najlepsze scenariusze użycia:

Algorytm Najniższego Kosztu jest najbardziej efektywny w sytuacjach, gdy celem jest znalezienie optymalnej ścieżki pod względem kosztu bez potrzeby stosowania heurystyki.

Znajduje zastosowanie w problemach takich jak optymalizacja tras, zarządzanie sieciami komputerowymi, logistyka i wyznaczanie najkrótszej drogi w systemach transportowych.

Jest idealny, gdy istnieje potrzeba gwarancji minimalnego kosztu przejścia, a heurystyka nie jest dostępna lub nie jest konieczna.

5. Porównanie algorytmów:

Stworzymy tabele w którą porównamy wszystkie algorytmy na parametrach: złożność czasowa i pamięciowa, najlepszy przypadek użycia.

Tabela 1: Teoretyczne porównanie algorytmów

Algorytm	Złożoność czasowa	Złożoność pamięci	Najlepszy przypadek użycia
Branch and Bound	Wykładniczy w najgorszym przypadku (w zależności od problemu)	O(n) or O(2^n)	Problemy optymalizacji, w których przycinanie jest skuteczne
Breadth-First Search	O(V + E)	O(V)	Najkrótsza ścieżka w grafach nieważonych, przeglądanie według poziomu
Depth-First Search	O(V + E)	O(V)	Głębokie lub nieograniczone przestrzenie poszukiwań, eksploracja głębokich ścieżek
Lowest Cost	O(V log V + E)	O(V)	Istnieje potrzeba gwarancji minimalnego kosztu przejścia, a heurystyka nie jest dostępna lub nie jest konieczna

Źrodło: opracowanie własne

Kiedy używać każdego algorytmu:

BFS (Breadth-First Search): Używamy BFS, gdy graf jest nieważony i potrzebujemy najkrótszej ścieżki między dwoma węzłami lub musimy przetwarzać węzły poziom po poziomie. BFS jest również skuteczne w problemach, gdzie konieczne jest znalezienie wszystkich możliwych ścieżek na danym poziomie głębokości, takich jak wyszukiwanie w drzewie decyzyjnym.

DFS (Depth-First Search): DFS nadaje się do problemów, które wymagają głębokiej eksploracji, takich jak sortowanie topologiczne, znajdowanie spójnych komponentów lub eksploracja wszystkich możliwych rozwiązań. DFS jest również przydatny, gdy przestrzeń wyszukiwania jest zbyt duża, aby BFS było praktyczne, zwłaszcza gdy chcemy szybko dotrzeć do głębokich ścieżek.

Najniższy Koszt (Lowest Cost Search): Najniższy koszt najlepiej stosować w problemach, gdzie kluczowe jest znalezienie ścieżki o najniższym całkowitym koszcie. Algorytm jest szczególnie użyteczny, gdy:

- Graf jest ważony, a wagi reprezentują rzeczywiste koszty (np. czas, odległość, zasoby).
- Problem wymaga minimalizacji kosztów przejścia, np. w logistyce, sieciach komputerowych lub systemach nawigacji.
- Nie mamy heurystyki do kierowania wyszukiwaniem, ale znamy dokładne koszty każdej krawędzi.

Podsumowanie:

Jeśli zależy na przetwarzaniu poziom po poziomie w grafie nieważonym, najlepszy jest *BFS*. Gdy eksploracja głębokości lub analiza całej przestrzeni rozwiązania jest kluczowa, użyj *DFS*. Jeśli jednak potrzebujemy znaleźć najtańszą ścieżkę w ważonym grafie, a heurystyka nie jest wymagana, *Najniższy Koszt* jest optymalnym wyborem.

Opis Działania Kodu

Funkcji pomiaru czasu

Funkcja StartCounter( ):

```
void StartCounter()
{
    LARGE_INTEGER freq;
    if (!QueryPerformanceFrequency(&freq))
        cout << "QueryPerformanceFrequency failed!\n";

    PCFreq = double(freq.QuadPart) / 1000.0;

    QueryPerformanceCounter(&freq);
    counterStart = freq.QuadPart;
}
```

Rysunek 4: Rzut ekranu z kodem funkcji StartCounter ( )

Źrodło: opracowanie własne

Na rysunku 5 jest przedstawiony kod funkcji StartCounter.

Ta funkcja inicjuje licznik wydajności o wysokiej rozdzielczości, konfiguruje licznik o wysokiej rozdzielczości przy użyciu interfejsu API systemu Windows. Oblicza częstotliwość licznika wydajności i przechowuje początkową wartość licznika dla pomiarów czasu. Drukuje komunikat o błędzie do ‘cout’, jeśli Query Performance Frequency się nie powiedzie.

Ta funkcja nie przyjmuje żadnych parametrów i nie zwraca wartości. Modyfikuje jednak zmienne globalne ‘PCFreq’ i ‘counterStart’.

Następnie mamy funkcje GetCounter( ).

```
double GetCounter()
{
    LARGE_INTEGER freq;
    QueryPerformanceCounter(&freq);
    return double(freq.QuadPart - counterStart) / PCFreq;
}
```

Rysunek 5: Rzut ekranu z kodem funkcji GetCounter ( )

Źrodło: opracowanie własne

Na rysunku 5 jest przedstawiony kod funkcji GetCounter, która oblicza czas, który upłynął od uruchomienia licznika.

Ta funkcja używa licznika o wysokiej wydajności systemu Windows do obliczenia czasu, który upłynął od wywołania funkcji StartCounter( ). Zwraca czas, który upłynął w milisekundach. Zwraca czas, który upłynął w milisekundach od uruchomienia licznika

Aby przejść do opisu kodu wczytywania danych z pliku i konfiguracji, a tak samo wypisywanie wyników w plik formatu cvs, musimy opisać jak konfigurować program i jak program wczytuje dane.

Najpierw, opiszmy plik konfiguracyjny. Przykładowo wygląda następnym sposobem, bardzo podobne do pliku konfiguracyjnego z projektu 1.

```
INPUT_FILE: entryData.txt
ALGORITHM: BFS
SHOW_STEPS: OFF
OUTPUT_FILE: tsp_res.csv
```

Rysunek 6: Rzut ekranu z przykładem działania pliku konfiguracujnego.
Źrodło: opracowanie własne

Jak to działa:

Najpierw musimy napisać nazwę pliku, w naszym przypadku – entryData.txt

Następnie musimy wpisać algorytm który chcemy używać w programie, na przykład, BFS, czyli Breadth-First Search.

Następnym krokiem możemy wybrać, czy chcemy pokazywać jak działa algorytm krok po kroku czy nie. Aby otrzymać wyniki do sprawozdanie, nie jest ta funkcja potrzebna, czyli zastawimy jej wyłączoną.

Ostatnio, jest wypisany plik, w który będą zapisane wyniki. Jest to plik formatu „csv", który można otworzyć w Microsoft Excel. Ale jak będzie wyglądał ten plik? Jest przedstawiony na rysunku 7.

```
Type, Algorithm Used, Best Tour, Minimum Distance, Time Of Execution
TSP, DFS, , 0, 0ms
```

Rysunek 7: Rzut ekranu z przykładem działania pliku końcowego.
Źrodło: opracowanie własne

Na czym polega ten plik: jak by robiliśmy to dla tabelki, zapisujemy czym są elementy pod wynikami. Na przykładzie rysunku 8 opiszemy każdy element.

„Type" – Wyświetla typ problemu który rozwiązany. Na przykład – TSP, czyli „Traveling Salesman Problem".

„Algorithm Used" – Pokazuje algorytm który wybraliśmy do rozwiązania problemu, na przykład – DFS, czyli Depth-First Search.

„Best Tour" – Pokazuje najlepszą ścieżkę którą wylosował algorytm.

„Minimum Distance" – Pokazuje wartość tej minimalnej dystancji/najlepszej ścieżcy.

"Time of Execusion" – Pokazuje ile czasu rozwiązało problem/matryce.

W pliku konfiguracyjnym też jest taki moment, jak „Input_file", czyli wczytujący plik.

Wygląda on następnym sposobem na rysunku 8:

```
TYPE: TSP
DIMENSION: 12
FORMAT: FULL_MATRIX
DATA_SECTION
  0   18  30  15  35  22  25  40  28  20  12  30
 18   0   22  30  18  25  20  35  28  15  40  12
 30  22   0   12  20  15  18  28  40  25  35  30
 15  30  12   0   28  18  35  22  25  20  12  40
 35  18  20  28   0   15  10  30  40  12  25  22
 22  25  15  18  15   0   30  20  28  18  35  40
 25  20  18  35  10  30   0   22  28  12  15  35
 40  35  28  22  30  20  22   0   15  28  18  12
 28  28  40  25  40  28  28  15   0   12  30  18
 20  15  25  20  12  18  12  28  12  0   10  35
 12  40  35  12  25  35  15  18  30  10   0   22
 30  12  30  40  22  40  35  12  18  35  22   0


EOF
```

Rysunek 8: Rzut ekranu z przykładem pliku wczytującego.

Źrodło: opracowanie własne

Na rysunku 9 możemy zobaczyć, że konstrukcja tego pliku przedstawiona następnym sposobem:

„Type” – Typ problemu rozwiązującego

„Dimention” – Rozmiar macierzy wczytującej. Na rysunku 9 jest przedstawiona macierz 12 x 12.

„Data_Section” – Tutaj jest przedstawiona sama macierz i wagi ścieżek. Mogą oni być jak symetryczne, tak i asymetryczne.

„EOF” – Pokazuje i daje znać programu, że to jest koniec plika.

Funkcja saveResults( ):

Omówiliśmy wyżej jak wygląda plik z wynikami. Teraz popatrzmy na funkcje która zapisuje te wyniki w plik. Na rysunku 9 możemy popatrzeć na kod tej funkcji.

```
void saveResults(string outputFileName, string type, string algorithm, vector<int> bestRoute, int minCost, double time)
{
    ofstream outputFile(outputFileName);
    if (outputFile.is_open())
    {
        outputFile << "Type, Algorithm Used, Best Tour, Minimum Distance, Time Of Execution\n";
        outputFile << type << ", ";
        outputFile << algorithm << ", ";
        for (int city : bestRoute)
        {
            outputFile << city << " ";
        }
        outputFile << ", " << minCost;
        outputFile << ", " << time << "ms"
                                      "\n";
        outputFile.close();
        cout << "Results written to tsp_results.csv" << endl;
    }
    else
    {
        cerr << "Unable to open file for writing." << endl;
    }
}
```

Rysunek 9: Rzut ekranu z kodem funkcji saveResults ( )

Źrodło: opracowanie własne

Ta funkcja zapisuje wyniki działania algorytmu do pliku w formacie CSV. Tworzy strumień wyjściowy do wskazanego pliku „outputFileName” i zapisuje dane takie jak typ zadania, używany algorytm, najlepsza trasa, minimalny koszt oraz czas wykonania algorytmu. Jeśli plik zostanie pomyślnie otwarty, dane zostaną zapisane w formie tekstowej.

Funkcja wypisuje komunikat „Results written to tsp_results.csv” do konsoli po poprawnym zapisaniu danych. Jeśli jednak otwarcie pliku się nie powiedzie, wyświetla błąd w cerr: „Unable to open file for writing.”

Nie zwraca żadnej wartości, a zapisuje wyniki do wskazanego pliku.

Funkcja configurate ( ):

Mówiliśmy o tym, że aby korzystać z aplikacji, musimy mieć plik konfiguracyjny. Też musimy zaprogramować go, aby on działał odpowiednio naszym potrzebom. Niżej, na rysunku 11, jest przedstawiony kod działania funkcji configurate ( ).

```
void configurate(const string &filename, string &algorithm, string &inputFile, string &showSteps, string &outputFile)
{
    ifstream file(filename);

    if (!file.is_open())
    {
        cerr << "Unable to open file: " << filename << endl;
        return;
    }

    string line;
    while (getline(file, line))
    {

        line.erase(0, line.find_first_not_of(" \t\n\r"));
        line.erase(line.find_last_not_of(" \t\n\r") + 1);

        if (line.find("INPUT_FILE:") != string::npos)
        {
            inputFile = line.substr(line.find(":") + 1);
            inputFile.erase(0, inputFile.find_first_not_of(" \t"));
        }
        if (line.find("ALGORITHM:") != string::npos)
        {
            algorithm = line.substr(line.find(":") + 1);
            algorithm.erase(0, algorithm.find_first_not_of(" \t"));
        }

        if (line.find("SHOW_STEPS:") != string::npos)
        {
            showSteps = line.substr(line.find(":") + 1);
            showSteps.erase(0, showSteps.find_first_not_of(" \t"));
        }
        if (line.find("OUTPUT_FILE:") != string::npos)
        {
            outputFile = line.substr(line.find(":") + 1);
            outputFile.erase(0, outputFile.find_first_not_of(" \t"));
        }
    }
    file.close();
}
```

Rysunek 10: Rzut ekranu z kodem funkcji configurate ( )

Źrodło: opracowanie własne

Funkcja configurate odczytuje plik konfiguracyjny zawierający ustawienia dla programu i przypisuje ich wartości do odpowiednich zmiennych. Tworzy strumień wejściowy (ifstream) na podstawie nazwy pliku przekazanej w parametrze filename i próbuje otworzyć ten plik. Jeśli plik nie zostanie pomyślnie otwarty, funkcja wypisuje komunikat o błędzie i kończy swoje działanie.

Funkcja działa w pętli, która czyta kolejne linie z pliku. Każda linia jest wstępnie przetwarzana w celu usunięcia zbędnych znaków takich jak spacje, tabulatory, czy znaki nowej linii na początku i końcu. Następnie sprawdzane jest, czy linia zawiera określone klucze, takie jak INPUT_FILE:, ALGORITHM:, SHOW_STEPS: lub OUTPUT_FILE:. Jeśli klucz zostanie znaleziony, funkcja wyodrębnia wartość znajdującą się po dwukropku, usuwa nadmiarowe spacje i przypisuje tę wartość do odpowiedniej zmiennej referencyjnej.

Każda sekcja pliku jest obsługiwana osobno. Dla INPUT_FILE: funkcja przypisuje nazwę pliku wejściowego do zmiennej „inputFile”. Dla ALGORITHM: pobiera wybrany algorytm (np. DFS, BFS) i zapisuje go w zmiennej „algorithm”. Analogicznie, SHOW_STEPS: kontroluje, czy program powinien wyświetlać kroki działania algorytmu, a OUTPUT_FILE: określa nazwę pliku, do którego mają być zapisane wyniki.

Na koniec funkcja zamyka plik przy pomocy metody close() i kończy działanie. Warto zauważyć, że funkcja nie zwraca żadnej wartości, lecz modyfikuje przekazane referencje, dzięki czemu wczytane dane są dostępne w wywołującym kodzie.

Funkcja readTSPFile( ):

Aby rozwiązać problem TSP, musimy wczytywać informacje z pliku z matrycą. Napiszmy funkcje, kod której będzie przedstawiony na rysunku 11.

```
void readTSPFile(const string &filename, string &type, int &dimension, string &format, vector<vector<int>> &unresolvedGraphData)
{
    ifstream file(filename);

    if (!file.is_open())
    {
        cerr << "Unable to open file: " << filename << endl;
        return;
    }

    string line;
    bool inEdgeWeightSection = false;

    while (getline(file, line))
    {

        line.erase(0, line.find_first_not_of(" \t\n\r"));
        line.erase(line.find_last_not_of(" \t\n\r") + 1);

        if (line.find("TYPE:") != string::npos && type.empty())
        {
            type = line.substr(line.find(":") + 1);
            type.erase(0, type.find_first_not_of(" \t"));
        }
        else if (line.find("DIMENSION:") != string::npos && dimension == 0)
        {
            istringstream iss(line.substr(line.find(":") + 1));
            iss >> dimension;
        }
        else if (line.find("FORMAT:") != string::npos && format.empty())
        {
            format = line.substr(line.find(":") + 1);
            format.erase(0, format.find_first_not_of(" \t"));
        }
        else if (line.find("DATA_SECTION") != string::npos)
        {
            inEdgeWeightSection = true;
            continue;
        }

        if (inEdgeWeightSection)
        {
            if (line.find("EOF") != string::npos)
            {
                break;
            }

            vector<int> weightsRow;
            istringstream iss(line);
            int weight;

            while (iss >> weight)
            {
                weightsRow.push_back(weight);
            }

            unresolvedGraphData.push_back(weightsRow);
        }
    }

    file.close();
}
```

Rysunek 11: Rzut ekranu z kodem funkcji readTSPFile ( )

Źrodło: opracowanie własne

Funkcja readTSPFile służy do odczytywania pliku w formacie TSP i wyodrębniania z niego danych takich jak typ problemu, liczba wierzchołków (rozmiar grafu), format danych oraz nieprzetworzone dane grafu. Na podstawie podanej nazwy pliku „filename” tworzy strumień wejściowy i próbuje otworzyć plik. Jeśli operacja otwarcia się nie powiedzie, funkcja wypisuje komunikat o błędzie i kończy działanie.

Wewnątrz pętli „while”, funkcja przetwarza każdą linię z pliku. Nadmiarowe spacje, tabulatory oraz znaki nowej linii na początku i końcu linii są usuwane. W zależności od zawartości linii, funkcja przypisuje wartości do zmiennych referencyjnych.

Po rozpoczęciu sekcji DATA_SECTION, funkcja wczytuje kolejne linie zawierające wagi krawędzi, aż napotka słowo kluczowe EOF, które kończy plik. Dla każdej linii w sekcji danych funkcja tworzy wektor

weightsRow, do którego zapisuje wagi odczytane ze strumienia istringstream. Następnie wektor ten jest dodawany do wektora unresolvedGraphData, który przechowuje wszystkie nieprzetworzone dane grafu w formacie macierzy sąsiedztwa.

Na końcu funkcja zamyka plik, używając metody close(). Dane wczytane z pliku są dostępne poprzez zmienne referencyjne, co pozwala na ich dalsze przetwarzanie w innych częściach programu. Funkcja została zaprojektowana w sposób elastyczny, dzięki czemu obsługuje różne typy i formaty plików TSP.

Przechodzimy do algorytmów i ich działania, co musimy zrobić, aby ich zaimplementować.

Wszystkie funkcje związane z plikami są w „file_reader.cpp" i „file_reader.h"

Zaczniemy od implementacji Branch and Bound.

Go implementacja jest przedstawiona na rysunku 12.

```
pair<int, vector<int>> branch_and_bound_tsp(const vector<vector<int>>& matrix, string search_method, double &time) {
    int n = matrix.size(); // Liczba miast w problemie.
    int best_cost = INT_MAX; // Inicjalizacja najlepszego kosztu jako maksymalnej możliwej wartości.
    vector<int> best_path;   // Wektor przechowujący najlepszą ścieżkę.

    // Inicjalizacja początkowego stanu: startujemy z miasta 0.
    vector<int> path = {0};      // Aktualna ścieżka zawierająca tylko miasto początkowe.
    set<int> visited = {0};     // Zbiór odwiedzonych miast, zawiera na razie tylko miasto początkowe.
    int current_cost = 0;       // Aktualny koszt ścieżki.

    StartCounter(); // Rozpoczęcie pomiaru czasu.

    // Wybór odpowiedniej metody przeszukiwania na podstawie parametru 'search_method'.
    if (search_method == "DFS") {
        // Wywołanie funkcji przeszukiwania w głąb (DFS) dla problemu TSP.
        DFS(matrix, best_path, best_cost, path, visited, current_cost);
    } else if (search_method == "BFS") {
        // Wywołanie funkcji przeszukiwania wszerz (BFS) dla problemu TSP.
        BFS(matrix, best_path, best_cost, path, visited, current_cost);
    } else if (search_method == "LC") {
        // Wywołanie funkcji przeszukiwania metodą najniższego kosztu (Lowest Cost First).
        lowest_cost_first_search(matrix, best_path, best_cost, path, visited, current_cost);
    }

    // Rejestracja czasu wykonania algorytmu.
    time = GetCounter();

    // Zwrócenie wyniku w postaci pary (najlepszy koszt, najlepsza ścieżka).
    return {best_cost, best_path};
}
```

Rysunek 12: Rzut ekranu z kodem funkcji *branch_and_bound_tsp ( )*

Źrodło: opracowanie własne

Funkcja branch_and_bound_tsp implementuje algorytm Branch and Bound dla rozwiązania problemu komiwojażera. Celem tej funkcji jest znalezienie najtańszej ścieżki, która odwiedza wszystkie miasta dokładnie raz, zaczynając i kończąc w tym samym mieście. Funkcja ta stanowi kluczową część programu, umożliwiając rozwiązanie problemu optymalizacyjnego przy użyciu różnych metod przeszukiwania, w tym DF, BFS oraz metody najniższego kosztu.

Dlaczego jest potrzebna ta funkcja:

Algorytm Branch and Bound zapewnia sposób na systematyczne ograniczenie przestrzeni wyszukiwania, aby znaleźć rozwiązanie w rozsądnym czasie. Funkcja umożliwia użytkownikowi wybór spośród trzech metod przeszukiwania (DFS, BFS, LC), które mogą być bardziej odpowiednie w zależności od struktury problemu. Dzięki temu, użytkownik może dostosować algorytm do swoich potrzeb w zależności od dostępnych danych. Funkcja mierzy czas działania algorytmu, co jest istotne w analizie wydajności rozwiązania w przypadku problemów dużej skali.

Opis działania funkcji:

Funkcja zaczyna od inicjalizacji zmiennych, takich jak liczba miast, najlepszy koszt (zainicjowany na maksymalną wartość), najlepsza ścieżka, oraz początkowa ścieżka zawierająca tylko miasto 0. Na podstawie przekazanego parametru „search_method", funkcja wywołuje odpowiednią metodę przeszukiwania (BFS, FDS, LC). Funkcja zwraca parę składającą się z najlepszego kosztu i najlepszej ścieżki, która stanowi rozwiązanie problemu TSP.

Funkcja *calculate_lower_bound( ):*

Na rysunku 13 jest przedstawion kod funkcji *calculate_lower_bound( ).*

```
// Obliczanie dolnej granicy kosztu w algorytmie
Tabnine | Edit | Test | Explain | Document | Ask
int calculate_lower_bound(const vector<vector<int>>& matrix, const vector<int>& path, const set<int>& visited) {
    int n = matrix.size(); // Liczba miast (rozmiar macierzy kosztów).
    int lower_bound = 0;   // Dolna granica kosztu, początkowo ustawiona na 0.

    // Iteracja przez każde miasto, aby znaleźć minimalny koszt krawędzi wychodzących.
    for (int i = 0; i < n; ++i) {
        // Sprawdź, czy miasto 'i' nie zostało jeszcze odwiedzone.
        if (visited.find(i) == visited.end()) {
            int min_edge = INT_MAX; // Minimalny koszt krawędzi dla miasta 'i', początkowo ustawiony na maksymalną wartość.

            // Przejrzyj wszystkie możliwe krawędzie wychodzące z miasta 'i'.
            for (int j = 0; j < n; ++j) {
                // Sprawdź, czy miasto 'j' nie zostało odwiedzone i jest różne od 'i'.
                if (i != j && visited.find(j) == visited.end()) {
                    // Znajdź minimalny koszt krawędzi z 'i' do 'j'.
                    min_edge = min(min_edge, matrix[i][j]);
                }
            }

            // Dodaj minimalny koszt krawędzi z miasta 'i' do dolnej granicy.
            lower_bound += min_edge;
        }
    }

    // Zwróć obliczoną dolną granicę kosztu.
    return lower_bound;
}
```

Rysunek 13: Rzut ekranu z kodem funkcji *calculate_lower_bound( )*

Źrodło: opracowanie własne

Funkcja „calculate_lower_bound" jest kluczowym elementem algorytmu „Branch and Bound", ponieważ umożliwia oszacowanie minimalnego kosztu rozwiązania, które można uzyskać z aktualnej ścieżki. Dolna granica pozwala na odrzucanie potencjalnie nieopłacalnych gałęzi w drzewie wyszukiwania, co znacząco ogranicza liczbę obliczeń i poprawia wydajność algorytmu.

Bez tej funkcji algorytm musiałby eksplorować wszystkie możliwe ścieżki, co jest obliczeniowo nieopłacalne dla większych instancji problemu komiwojażera (TSP).

Dzięki obliczaniu dolnej granicy, algorytm może szybciej porzucać gałęzie, które nie mogą prowadzić do lepszego rozwiązania niż dotychczasowe najlepsze. Funkcja identyfikuje minimalny koszt krawędzi wychodzących z każdego nieodwiedzonego miasta, co pozwala na realistyczne oszacowanie potencjalnego minimalnego kosztu. Funkcja może być używana w algorytmach DFS, BFS oraz metodzie najniższego kosztu, co zwiększa jej uniwersalność.

Funkcja „calculate_lower_bound" współpracuje z funkcjami takich jak DFS, BFS, czy „branch_and_bound_tsp", gdzie odgrywa rolę w ograniczaniu eksploracji przestrzeni wyszukiwania, przyspieszając rozwiązanie problemu TSP.

Jak działa funkcja:

Funkcja iteruje przez wszystkie miasta, sprawdzając, które z nich nie zostały odwiedzone, a następnie dla każdego nieodwiedzonego miasta znajduje minimalny koszt krawędzi wychodzącej do innych nieodwiedzonych miast. Minimalne koszty te są sumowane, aby obliczyć dolną granicę kosztu, która jest następnie zwracana do algorytmu. Na tej podstawie algorytm decyduje, czy eksplorować daną gałąź, porównując sumę obecnego kosztu i dolnej granicy z najlepszym dotychczasowym kosztem.

Przejdziemy do opisu każdego algorytmu.

Zaczniemy od opisu działania algorytmu BFS. Stworzyliśmy do niego osobne „bfs.cpp” i „bfs.h”.

Niżej na rysunku 14 jest przedstawiony kod implementacji algorytmu BFS.

```cpp
void BFS(const vector<vector<int>>& matrix, vector<int>& best_path, int& best_cost, vector<int>& path, set<int>& visited, int current_cost) {
    // Pobieramy rozmiar macierzy, czyli liczbę miast
    int n = matrix.size();

    // Inicjalizujemy kolejkę BFS
    // Każdy element to para składająca się z:
    // 1) Ścieżki (vector<int>) i odwiedzonych miast (set<int>)
    // 2) Aktualnego kosztu (int)

    queue<pair<pair<vector<int>, set<int>>, int>> q;
    // Dodajemy początkowy stan do kolejki
    q.push({{path, visited}, current_cost});

    // Dopóki kolejka nie jest pusta, przetwarzamy jej elementy
    while (!q.empty()) {
        // Pobieramy pierwszy element z kolejki (FIFO)
        auto current = q.front();
        q.pop(); // Usuwamy element z kolejki

        // Rozpakowujemy aktualny stan
        vector<int> current_path = current.first.first;  // Obecna ścieżka
        set<int> current_visited = current.first.second; // Zestaw odwiedzonych miast
        int current_cost = current.second;               // Koszt dotychczasowej ścieżki

        // Jeśli ścieżka zawiera wszystkie miasta (pełny cykl)
        if (current_path.size() == n) {
            // Dodajemy koszt powrotu do miasta początkowego
            current_cost += matrix[current_path.back()][current_path[0]];
            // Jeśli ten koszt jest mniejszy niż najlepszy dotychczasowy koszt
            if (current_cost < best_cost) {
                best_cost = current_cost; // Aktualizujemy najlepszy koszt
                best_path = current_path; // Zapisujemy najlepszą ścieżkę
            }
            continue; // Kończymy przetwarzanie tego stanu
        }

        // Iterujemy przez wszystkie możliwe miasta
        for (int next_city = 0; next_city < n; ++next_city) {
            // Sprawdzamy, czy miasto nie zostało jeszcze odwiedzone
            if (current_visited.find(next_city) == current_visited.end()) {
                // Obliczamy nowy koszt po dodaniu tego miasta do ścieżki
                int new_cost = current_cost + matrix[current_path.back()][next_city];

                // Tworzymy nową ścieżkę, dodając kolejne miasto
                vector<int> new_path = current_path;
                new_path.push_back(next_city);
                // Tworzymy nowy zestaw odwiedzonych miast
                set<int> new_visited = current_visited;
                new_visited.insert(next_city);

                // Obliczamy dolną granicę kosztu dla tej nowej ścieżki
                // Funkcja `calculate_lower_bound` oblicza minimalny koszt pozostałych krawędzi
                int lower_bound = calculate_lower_bound(matrix, new_path, new_visited);

                // Jeśli nowy koszt + dolna granica jest mniejszy niż dotychczasowy najlepszy koszt
                if (new_cost + lower_bound < best_cost) {
                    // Dodajemy nowy stan do kolejki, by go dalej przetwarzać
                    q.push({{new_path, new_visited}, new_cost});
                }
            }
        }
    }
}
```

Rysunek 14: Rzut ekranu z kodem funkcji *BFS( )*

Źrodło: opracowanie własne

Algorytm BFS (Breadth-First Search) w kontekście rozwiązania problemu komiwojażera (TSP) ma na celu znalezienie najtańszej możliwej ścieżki, która odwiedza każde miasto dokładnie raz. Został zaprojektowany z użyciem kolejki, która umożliwia przetwarzanie stanów w porządku szerokości, a nie głębokości, jak ma to miejsce w algorytmie DFS. Dzięki temu algorytm BFS działa w sposób iteracyjny, analizując wszystkie możliwe opcje na danym etapie przed przejściem do bardziej złożonych kombinacji. W odróżnieniu od DFS, który przechodzi głęboko w głąb jednego rozwiązania zanim spróbuje innych możliwości, BFS bada wszystkie dostępne opcje na poziomie początkowym, co zapobiega nadmiernej rekurencji i umożliwia równoczesne przetwarzanie stanów. Ta cecha algorytmu BFS sprawia, że jest on odpowiedni w kontekście problemu TSP, gdzie konieczne jest systematyczne rozważenie różnych ścieżek, aby znaleźć tę, która będzie najtańsza, zachowując jednocześnie optymalność rozwiązania.

Główna pętla algorytmu działa, dopóki kolejka nie jest pusta. Działa to na zasadzie FIFO (First In, First Out). Jeśli ścieżka jeszcze nie zawiera wszystkich miast, algorytm przechodzi do następnego etapu. Dla każdego miasta, które jeszcze nie zostało odwiedzone, obliczany jest nowy koszt po dodaniu tego miasta do ścieżki. Algorytm kończy działanie, kiedy kolejka zostanie opróżniona, czyli wszystkie stany zostały już przetworzone. Na końcu najlepsza ścieżka oraz jej koszt zostają zapisane w zmiennych ‘best_path’ i ‘best_cost’.

Następnie popatrzmy na implementację algorytmu DFS. Stworzone do niego „dfs.cpp” i „dfs.h”.

Na rysunku 15 jest przedstawiony kod z komentarzami do implementacji algorytmu DFS.

```
// Algorytm przeszukiwania w głąb (DFS) dla problemu TSP
Tabnine | Edit | Test | Explain | Document | Ask
void DFS(const vector<vector<int>>& matrix, vector<int>& best_path, int& best_cost, vector<int>& path, set<int>& visited, int current_cost) {
    int n = matrix.size(); // Liczba miast w problemie TSP

    // Sprawdzamy, czy odwiedzono wszystkie miasta (pełny cykl)
    if (path.size() == n) {
        current_cost += matrix[path.back()][path[0]]; // Dodanie kosztu powrotu do miasta początkowego
        if (current_cost < best_cost) { // Aktualizacja najlepszego kosztu, jeśli znaleziono lepszą ścieżkę
            best_cost = current_cost;
            best_path = path; // Zapisanie nowej najlepszej ścieżki
        }
        return; // Zakończenie przetwarzania tej gałęzi
    }

    // Iterujemy przez wszystkie miasta, które mogą być kolejnym przystankiem
    for (int next_city = 0; next_city < n; ++next_city) {
        // Sprawdzamy, czy miasto nie zostało jeszcze odwiedzone
        if (visited.find(next_city) == visited.end()) {
            // Obliczamy nowy koszt po dodaniu tego miasta do ścieżki
            int new_cost = current_cost + matrix[path.back()][next_city];

            // Dodajemy miasto do ścieżki
            path.push_back(next_city);
            visited.insert(next_city); // Oznaczamy miasto jako odwiedzone

            // Obliczamy dolną granicę kosztu dla bieżącej ścieżki
            // Funkcja `calculate_lower_bound` ocenia minimalny koszt kontynuacji ścieżki
            int lower_bound = calculate_lower_bound(matrix, path, visited);

            // Sprawdzamy, czy warto kontynuować eksplorację tej gałęzi
            if (new_cost + lower_bound < best_cost) {
                // Rekurencyjnie wywołujemy DFS dla kolejnych miast
                DFS(matrix, best_path, best_cost, path, visited, new_cost);
            }

            // Cofamy zmiany, by umożliwić eksplorację innych gałęzi
            visited.erase(next_city); // Usuwamy miasto z odwiedzonych
            path.pop_back(); // Usuwamy miasto z bieżącej ścieżki
        }
    }
}
```

Rysunek 15: Rzut ekranu z kodem funkcji *DFS( )*

Źrodło: opracowanie własne

Algorytm DFS (Depth-First Search) dla problemu TSP ma na celu znalezienie najtańszej możliwej ścieżki, która odwiedza każde miasto dokładnie raz, a następnie wraca do miasta początkowego. W tym przypadku algorytm jest zaimplementowany w sposób rekurencyjny, co oznacza, że używa stosu wywołań funkcji do przechowywania bieżących stanów i ścieżek. W przeciwieństwie do algorytmu BFS, który przetwarza stany w porządku szerokości, DFS zagłębia się w jedną gałąź rozwiązania, zanim spróbuje innych możliwości.

Opis działania algorytmu:

Algorytm kończy działanie, gdy wszystkie miasta zostaną odwiedzone, co oznacza, że ścieżka jest pełna. Na tym etapie dodawany jest koszt powrotu do miasta początkowego, a jeśli nowy koszt jest mniejszy niż obecnie najlepszy koszt, aktualizowane są zmienne ‘best_path’ i ‘best_cost’. Algorytm przegląda wszystkie możliwe miasta, które jeszcze nie zostały odwiedzone, i dla każdego z nich oblicza nowy koszt, dodając miasto do ścieżki.

Algorytm wywołuje się rekurencyjnie, aby kontynuować eksplorację dla kolejnych miast, o ile łączny koszt ścieżki z uwzględnieniem dolnej granicy potencjalnego kosztu jest mniejszy niż obecny najlepszy koszt. Też można zaimplementować DFS, używając stos.

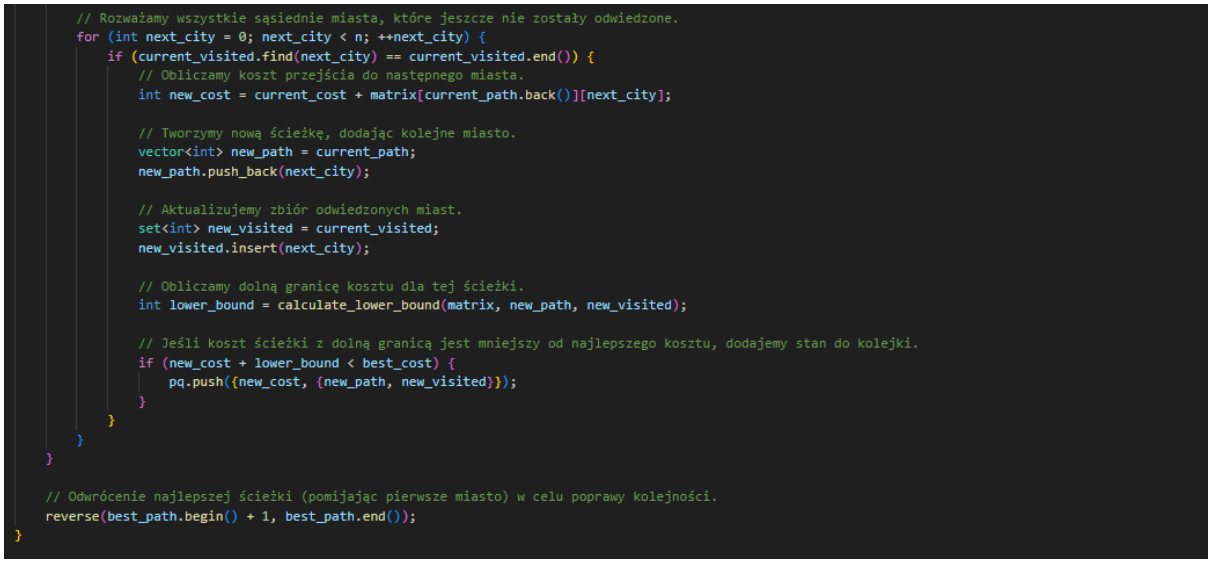
Rekurencja a stos:

W tej implementacji algorytmu DFS używana jest rekurencja, co oznacza, że stos wywołań funkcji jest wykorzystywany do przechowywania stanów. To pozwala na efektywne zarządzanie ścieżkami i powrotem do poprzednich stanów, ale może prowadzić do problemów z wydajnością dla dużych problemów z powodu ograniczonej głębokości stosu wywołań.

Ostatecznie, mamy implementacje algorytmu Lowest Cost.

Na rysunku 16 jest przedstawiony rzut ekranu z implementacją algorytmu Lowest Cost dla problemu TSP.

```
void lowest_cost_first_search(const vector<vector<int>>& matrix, vector<int>& best_path, int& best_cost, vector<int>& path, set<int>& visited, int current_cost) {
    int n = matrix.size(); // Liczba miast w macierzy.

    // Kolejka priorytetowa do przechowywania stanów. Porównanie odbywa się według kosztu (od najmniejszego).
    priority_queue<pair<int, pair<vector<int>, set<int>>>,
            vector<pair<int, pair<vector<int>, set<int>>>>,
            greater<pair<int, pair<vector<int>, set<int>>>>> pq;

    // Dodanie początkowego stanu do kolejki (koszt = 0, ścieżka = startowa, odwiedzone = startowe miasto).
    pq.push({current_cost, {path, visited}});

    // Dopóki kolejka nie jest pusta, przetwarzamy stany.
    while (!pq.empty()) {
        // Pobranie stanu o najniższym koszcie z kolejki.
        auto current = pq.top();
        pq.pop();

        vector<int> current_path = current.second.first;   // Aktualna ścieżka.
        set<int> current_visited = current.second.second; // Zbiór odwiedzonych miast.
        int current_cost = current.first;                 // Aktualny koszt.

        // Jeśli odwiedzono wszystkie miasta:
        if (current_path.size() == n) {
            // Dodaj koszt powrotu do miasta początkowego, aby zamknąć cykl.
            current_cost += matrix[current_path.back()][current_path[0]];

            // Jeśli koszt cyklu jest lepszy niż obecny najlepszy koszt, aktualizujemy najlepszy wynik.
            if (current_cost < best_cost) {
                best_cost = current_cost;
                best_path = current_path;
            }
            continue; // Przetwarzamy kolejny stan.
        }
```



```
        // Rozważamy wszystkie sąsiednie miasta, które jeszcze nie zostały odwiedzone.
        for (int next_city = 0; next_city < n; ++next_city) {
            if (current_visited.find(next_city) == current_visited.end()) {
                // Obliczamy koszt przejścia do następnego miasta.
                int new_cost = current_cost + matrix[current_path.back()][next_city];

                // Tworzymy nową ścieżkę, dodając kolejne miasto.
                vector<int> new_path = current_path;
                new_path.push_back(next_city);

                // Aktualizujemy zbiór odwiedzonych miast.
                set<int> new_visited = current_visited;
                new_visited.insert(next_city);

                // Obliczamy dolną granicę kosztu dla tej ścieżki.
                int lower_bound = calculate_lower_bound(matrix, new_path, new_visited);

                // Jeśli koszt ścieżki z dolną granicą jest mniejszy od najlepszego kosztu, dodajemy stan do kolejki.
                if (new_cost + lower_bound < best_cost) {
                    pq.push({new_cost, {new_path, new_visited}});
                }
            }
        }
    }

    // Odwrócenie najlepszej ścieżki (pomijając pierwsze miasto) w celu poprawy kolejności.
    reverse(best_path.begin() + 1, best_path.end());
}
```

Rysunek 16: Rzut ekranu z kodem funkcji Lowest_Cost_First_Search( )

Źrodło: opracowanie własne

Algorytm "Lowest Cost First Search" (LCFS) jest podejściem optymalizacyjnym, które ma na celu znalezienie najtańszej możliwej ścieżki, która odwiedza wszystkie miasta dokładnie raz, a następnie wraca do miasta początkowego. W odróżnieniu od algorytmów takich jak DFS czy BFS, LCFS korzysta z priorytetowej kolejki, aby dynamicznie przetwarzać stany w kolejności rosnącego kosztu, co sprawia, że algorytm koncentruje się najpierw na badaniu ścieżek o najniższym kosztu.

Jak działa algorytm:

Algorytm rozpoczyna działanie, dodając początkowy stan (świeża ścieżka startowa i zbiór odwiedzonych miast) do priorytetowej kolejki, która jest uporządkowana według kosztu. Dzięki temu stan o najniższym koszcie zawsze jest przetwarzany jako pierwszy. Wyciąga stan o najniższym koszcie z kolejki i sprawdza, czy ścieżka jest kompletna (czy odwiedzono wszystkie miasta). Jeśli tak, dodaje koszt powrotu do miasta początkowego i aktualizuje najlepszy koszt i ścieżkę, jeśli jest lepsza niż dotychczasowa. Dla aktualnego stanu algorytm rozważa przejście do wszystkich sąsiednich miast, które jeszcze nie zostały odwiedzone, obliczając nowy koszt i tworząc nową ścieżkę. Jeśli suma kosztu aktualnej ścieżki i dolnej granicy jest mniejsza niż obecny najlepszy koszt, nowy stan jest dodawany do kolejki. Proces powtarza się, aż kolejka stanie się pusta, co oznacza, że wszystkie możliwe stany zostały przetworzone.

Algorytm używa kolejki priorytetowej (min-heap), aby przechowywać stany i zapewnić, że najpierw przetwarzane są stany o najniższym koszcie. Kolejka priorytetowa jest skonstruowana tak, że elementy są posortowane według kosztu, co pozwala na szybkie wyszukiwanie i aktualizację najmniejszych kosztów.

Algorytm LCFS jest bardziej zaawansowaną wersją podejścia BFS, które uwzględnia dolne ograniczenia kosztów i optymalizuje proces eksploracji poprzez priorytetowe przetwarzanie stanów. Dzięki temu jest bardziej efektywny w rozwiązywaniu problemów TSP niż klasyczne podejścia, takie jak BFS czy DFS, zwłaszcza dla większych problemów. Główne zalety tego algorytmu wynikają z możliwości redukcji liczby badanych ścieżek poprzez ocenę dolnej granicy kosztu, co pozwala na szybsze znalezienie optymalnego rozwiązania.

Opracowanie wyników i wnioski

Aby było wykonywanie projektu i pobieranie danych trochę łatwiejsze, pobierzmy dane z używaniem dystansu Euklidesowego. Zrobimy pomiary na algorytmy BFS, DFS i Lowest Cost dla TPS, które mają rozmiar: 4x4, 5x5, 6x6, 8x8, 9x9, 10x10, 11x11, 12x12. Niektóre z nich są wygenerowane samodzielnie, aby otrzymać więcej wyników i daty dla grafów. Sprawdzimy też wyniki działania do niektórych asymetrycznych macierzy i porównamy wyniki.

Zaczniemy od algorytmu BFS.

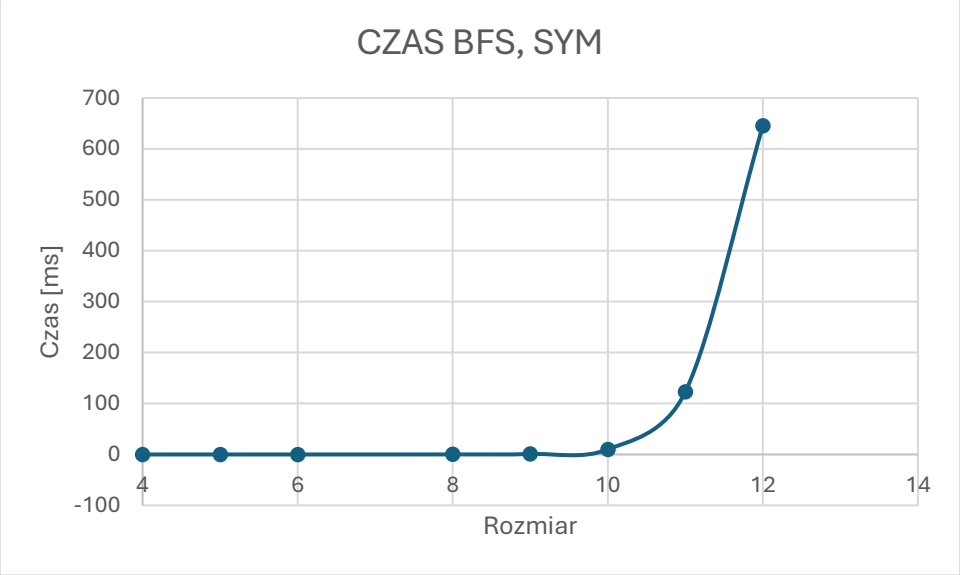
Niżej jest przedstawiona tabela 2, w której są wyniki pomiarów czasu do macierzy symetrycznych i asymetrycznych. Wyniki do symetrycznych i asymetrycznych są podzielone różnymi kolorami. Na rysunkach 17 i 18 są przedstawione grafy, stworzone z wyników, przedstawionych w tabele 2.

Tabela 2: Wyniki działania algorytmu BFS dla problemów symetrycznych i asymetrycznych

Źrodło: opracowanie własne

TYP	ALG	ROZMIAR	TIME [ms]
TSP	BFS	4	0.0001
TSP	BFS	5	0.0004
TSP	BFS	6	0.0022
TSP	BFS	8	0.11
TSP	BFS	9	0.97
TSP	BFS	10	9.97
TSP	BFS	11	122.99
TSP	BFS	12	645.54
ATSP	BFS	4	0.5*10^-9
ATSP	BFS	5	0.0004
ATSP	BFS	6	0.002
ATSP	BFS	8	0.113
ATSP	BFS	9	0.995
ATSP	BFS	10	9.697
ATSP	BFS	11	108.18
ATSP	BFS	12	633.89

Analiza pomiarów dla problemu symetrycznego:



Rysunek 17: Graf opracowania wyników algorytmu *BFS* dla symetrycznego problemu
Źrodło: opracowanie własne

Z tablicy 2 możemy wziąć jakieś parametry, aby udowodnić się, że algorytm działa zgodnie z teorią.

Z danych wynika, że potrzebny czas rośnie niezwykle szybko wraz ze wzrostem rozmiaru problemu (ROZMIAR). Na przykład:

Od rozmiaru 9 do 10 czas działania wzrasta około 10-krotnie (0,97 ms -> 9,97 ms).

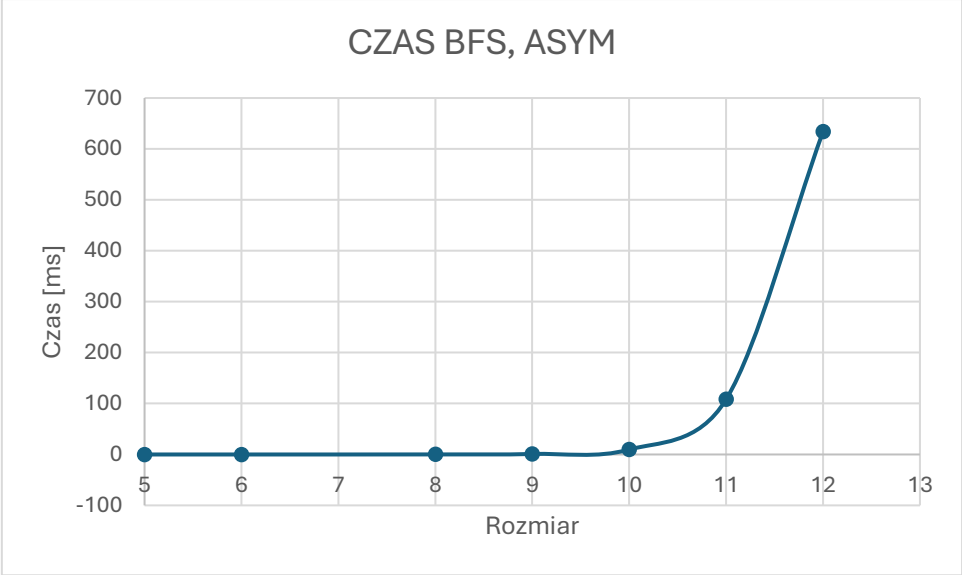
Od 10 do 11 ponownie znacznie skacze (9,97 ms -> 122,99 ms).

Od 11 do 12 rośnie mniej więcej 5x (122,99 ms -> 645,54 ms).

Oznacza to, że środowisko wykonawcze ma wykładniczy wzorzec wzrostu, czego oczekuje się od algorytmu BFS w kontekście TSP.

Podejście BFS do rozwiązania TSP zasadniczo wylicza wszystkie możliwe permutacje miast, aby określić najkrótszą ścieżkę. Liczba permutacji dla n miast wynosi (n-1)! (ze względu na symetrię i ustalony punkt początkowy), co prowadzi do złożoności czasowej czynnikowej, O(n!).

Ta teoretyczna złożoność jest zgodna z obserwowanym szybkim wzrostem czasu wykonania w miarę zwiększania rozmiaru.



Rysunek 18: Graf opracowania wyników algorytmu *BFS* dla asymetrycznego problemu

Źrodło: opracowanie własne

Analiza pomiarów dla problemu asymetrycznego:

Z tablicy wyników dla problemu asymetrycznego możemy zauważyć, że algorytm działa zgodnie z przewidywaniami teoretycznymi.

Z danych wynika, że potrzebny czas działania algorytmu BFS rośnie bardzo szybko wraz ze wzrostem rozmiaru problemu. Oto szczegółowa analiza:

Od 9 do 10: Czas działania wzrasta około 10-krotnie (0,995 ms -> 9,697 ms).

Od 10 do 11: Czas działania zwiększa się znacznie bardziej, prawie 11-krotnie (9,697 ms -> 108,18 ms).

Od 11 do 12: Czas działania wzrasta około 6-krotnie (108,18 ms -> 633,89 ms).

Zachowanie algorytmu BFS dla problemu asymetrycznego potwierdza wzorzec wykładniczego wzrostu, co jest zgodne z teoretycznymi założeniami. Algorytm BFS w kontekście asymetrycznego TSP przeszukuje wszystkie możliwe permutacje miast, jednakże w przeciwieństwie do problemu symetrycznego liczba permutacji wynosi dokładnie n!, a nie (n−1)!, ponieważ w problemie asymetrycznym brak jest ograniczenia wynikającego z symetrii.

Teoretyczna złożoność czasowa O(n!) prowadzi do jeszcze szybszego wzrostu czasu wykonania niż w przypadku symetrycznym, co jest zauważalne w powyższych wynikach.

Wniosek: z powodu tego, że wynik na rozmiar 4x4 był bardzo mały, nie braliśmy go pod uwagę na grafie.

Porównanie wyników dla problemów symetrycznego i asymetrycznego:

Czas wykonania:

Dla małych rozmiarów (ROZMIAR ≤ 6) czasy w obu przypadkach są zbliżone. Dla większych rozmiarów (ROZMIAR ≥ 9) BFS dla problemu asymetrycznego jest minimalnie wolniejszy niż dla symetrycznego.

Przyczyna różnic: W problemie asymetrycznym liczba permutacji wynosi n!, co jest większe niż (n−1)! w problemie symetrycznym. To powoduje nieco szybszy wzrost czasu w asymetrycznym przypadku.

Podsumowanie:

Problemy symetryczny i asymetryczny wykazują podobny wykładniczy wzorzec wzrostu czasu działania BFS, ale asymetryczny jest nieco bardziej czasochłonny ze względu na brak redukcji symetrii.

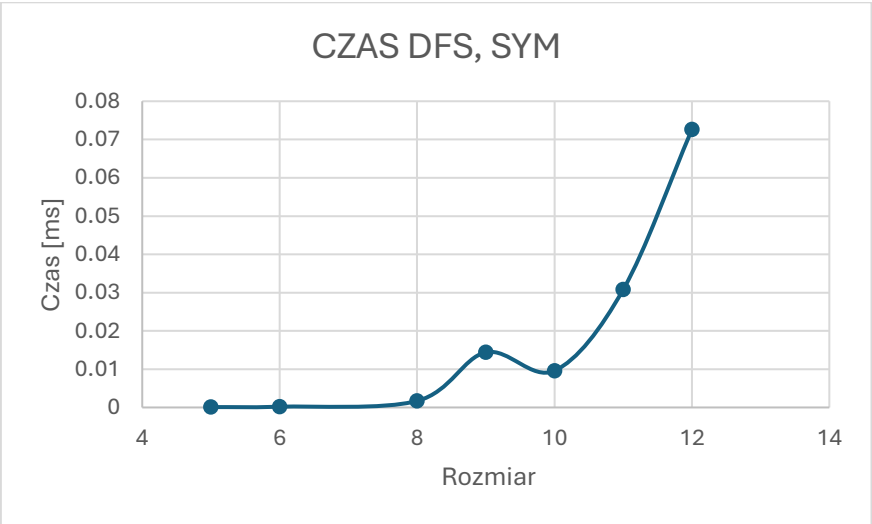
Następnie, zrobimy analizę wyników działania algorytmy DFS.

Niżej jest przedstawiona tabela 3, w której są wyniki pomiarów czasu do macierzy symetrycznych i asymetrycznych. Wyniki do symetrycznych i asymetrycznych są podzielone różnymi kolorami. Na rysunkach 19 i 20 są przedstawione grafy, stworzone z wyników, przedstawionych w tabele 3.

Tabela 3: Wyniki działania algorytmu DFS dla problemów symetrycznych i asymetrycznych

Źrodło: opracowanie własne

TYP	ALG	ROZMIAR	TIME [ms]
TSP	DFS	4	5*10^-9
TSP	DFS	5	0.0001
TSP	DFS	6	0.0002
TSP	DFS	8	0.0017
TSP	DFS	9	0.0144
TSP	DFS	10	0.0096
TSP	DFS	11	0.0308
TSP	DFS	12	0.0726
ATSP	DFS	4	5*10^-9
ATSP	DFS	5	8*10^-9
ATSP	DFS	6	0.0002
ATSP	DFS	8	0.0030
ATSP	DFS	9	0.0052
ATSP	DFS	10	0.0163
ATSP	DFS	11	0.0405
ATSP	DFS	12	0.1145



Rysunek 19: Graf opracowania wyników algorytmu *DFS* dla symetrycznego problemu

Źrodło: opracowanie własne

Analiza pomiarów dla problemu symetrycznego:

Z tablicy wyników możemy zauważyć ogólny wzorzec działania algorytmu DFS. Algorytm DFS pokazuje wzrost czasu wykonania wraz ze wzrostem rozmiaru problemu (ROZMIAR), ale nie jest tak regularny jak w przypadku BFS. Oto kilka szczegółowych obserwacji:

Od 8 do 9 czas wykonania wzrasta ponad 8-krotnie (0,0017 ms -> 0,0144 ms).

Jednak między 9 a 10 czas wykonania nieoczekiwanie spada (0,0144 ms -> 0,0096 ms).

Od rozmiar 10 do 11 czas znowu znacznie rośnie (0,0096 ms -> 0,0308 ms).

Dla rozmiaru 12 czas osiąga wartość 0,0726 ms, co jest zgodne z oczekiwaniami wykładniczego wzrostu, choć nie tak wyraźnego jak w BFS.

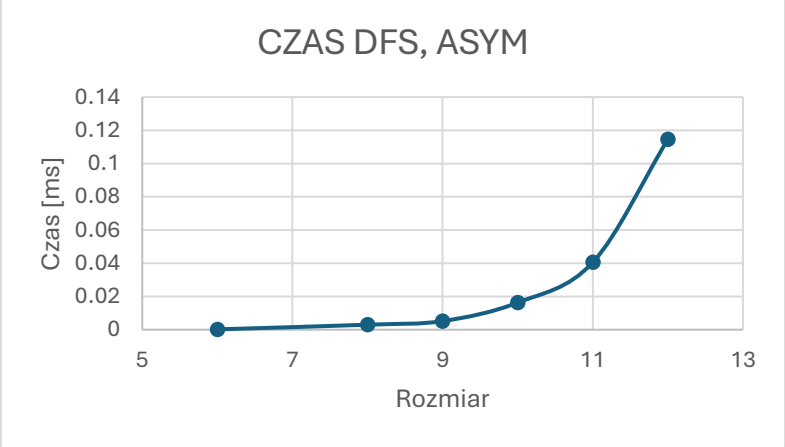
Algorytm DFS dla TSP działa poprzez eksplorację ścieżek do końca i cofanie się w celu sprawdzenia kolejnych możliwości. Jego teoretyczna złożoność czasowa wynosi O(n!), ponieważ również przeszukuje wszystkie możliwe permutacje. Jednak w praktyce, w przypadku DFS, optymalizacje wynikające z heurystyk mogą powodować bardziej złożone wzorce wzrostu czasu działania.

Problemy z wynikami dla rozmiaru 9:

Wyniki dla rozmiaru 9 wydają się anomalią, ponieważ czas działania (0,0144 ms) jest większy niż dla rozmiaru 10 (0,0096 ms). Możliwe przyczyny tego zjawiska to:

- Inne procesy systemowe mogły wpłynąć na pomiary, szczególnie jeśli algorytm działa szybko, a różnice czasów są w milisekundach lub mikrosekundach.
- Kompilator lub sprzętowe mechanizmy cache'owania mogły wpłynąć na wyniki, co czasem powoduje, że większe dane działają szybciej.
- Możliwe, że implementacja DFS w kodzie nie obsługuje dobrze pewnych rozmiarów macierzy. Przykładowo, dla rozmiaru 9, liczba węzłów lub permutacji mogła wywołać nieprzewidziane zachowanie w warunkach bazowych lub iteracjach.
- Ale myślę, że jest to może i z powodu tego, że są bardziej skomplikowane do DFS rozwiązania dane.

Analiza pomiarów dla problemu symetrycznego:



Rysunek 20: Graf opracowania wyników algorytmu *DFS* dla asymetrycznego problemu

Źrodło: opracowanie własne

Wniosek: Z powodu tego, że wynik na rozmiarach 4x4 i 5x5 były bardzo małe, nie braliśmy ich pod uwagę na grafie.

Z tablicy wyników dla macierzy asymetrycznej można zauważyć, że algorytm DFS wykazuje ogólny wzrost czasu wykonania wraz ze wzrostem rozmiaru problemu. Tym razem wzorzec wzrostu jest bardziej regularny niż w przypadku macierzy symetrycznej.

Od 8 do 9 czas wykonania wzrasta około 1,7-krotnie (0,0030 ms -> 0,0052 ms).

Od 9 do 10 wzrost czasu jest ponad 3-krotny (0,0052 ms -> 0,0163 ms).

Dla rozmiaru 11 czas wzrasta prawie 2,5-krotnie w stosunku do 10 (0,0163 ms -> 0,0405 ms).

W przypadku 12 czas osiąga wartość 0,1145 ms, co wskazuje na wyraźną tendencję wzrostu, choć mniej gwałtowną niż w BFS.

DFS w przypadku asymetrycznego TSP musi eksplorować większą liczbę permutacji (pełne n!, bez redukcji symetrii). Teoretyczna złożoność czasowa nadal wynosi O(n!), jednak różnice w czasie wykonania mogą wynikać z lepszej regularności danych asymetrycznych lub implementacyjnych optymalizacji.

Podsumowanie:

Problemy symetryczny i asymetryczny wykazują podobny wzorzec wzrostu czasu działania DFS, zgodny z teoretyczną złożonością O(n!). Jednak problem asymetryczny jest nieco bardziej czasochłonny, ponieważ DFS musi przeszukiwać wszystkie permutacje bez możliwości redukcji wynikającej z symetrii. W przypadku problemu symetrycznego czasami obserwuje się drobne anomalie, które mogą wynikać z implementacji lub środowiska wykonawczego.

Teraz popatrzmy na wyniki algorytmu Lowest Cost.

Niżej jest przedstawiona tabela 4, w której są wyniki pomiarów czasu do macierzy symetrycznych i asymetrycznych. Wyniki do symetrycznych i asymetrycznych są podzielone różnymi kolorami. Na rysunkach 21 i 22 są przedstawione grafy, stworzone z wyników, przedstawionych w tabele 4.

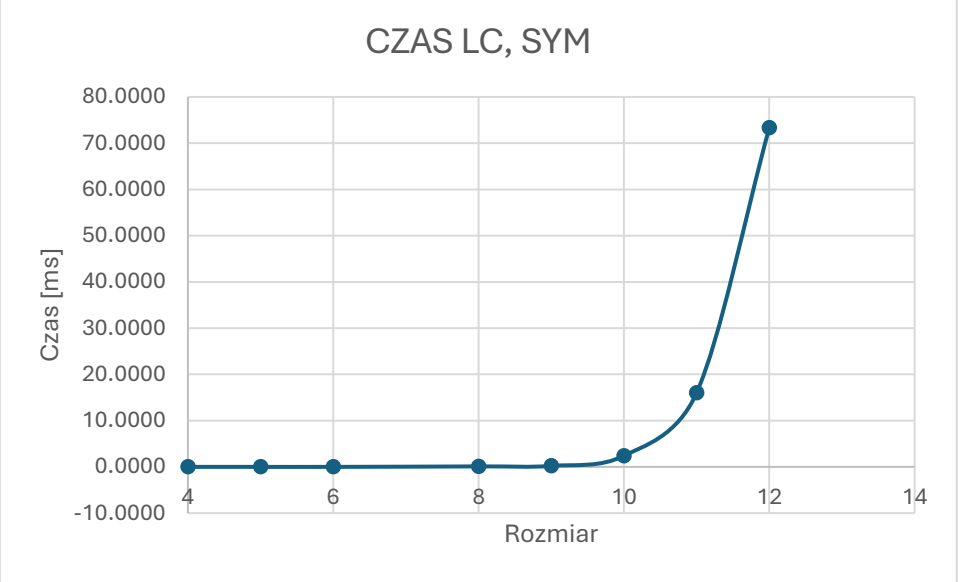
Tabela 4: Wyniki działania algorytmu Lowest Cost dla problemów symetrycznych i asymetrycznych

Źrodło: opracowanie własne

TYP	ALG	ROZMIAR	TIME [ms]
TSP	LC	4	0.0001
TSP	LC	5	0.0005
TSP	LC	6	0.0022
TSP	LC	8	0.1025
TSP	LC	9	0.2146
TSP	LC	10	2.4004
TSP	LC	11	16.0070
TSP	LC	12	73.3270
ATSP	LC	4	0.0001
ATSP	LC	5	0.0004
ATSP	LC	6	0.0020
ATSP	LC	8	0.0822

ATSP	LC	9	0.7188
ATSP	LC	10	2.9024
ATSP	LC	11	28.1476
ATSP	LC	12	94.2588

Analiza pomiarów dla problemu symetrycznego:



Rysunek 21: Graf opracowania wyników algorytmu *Lowest Cost* dla symetrycznego problemu

Źrodło: opracowanie własne

Z wyników przedstawionych w tabeli możemy zauważyć, że czas wykonania algorytmu Lowest Cost (LC) wzrasta wraz ze wzrostem rozmiaru problemu. Wzrost ten jest zgodny z przewidywaniami teoretycznymi, choć jest bardziej umiarkowany w początkowych etapach.

Od rozmiaru 8 do 9 czas wykonania wzrasta około 2-krotnie (0,1025 ms -> 0,2146 ms).

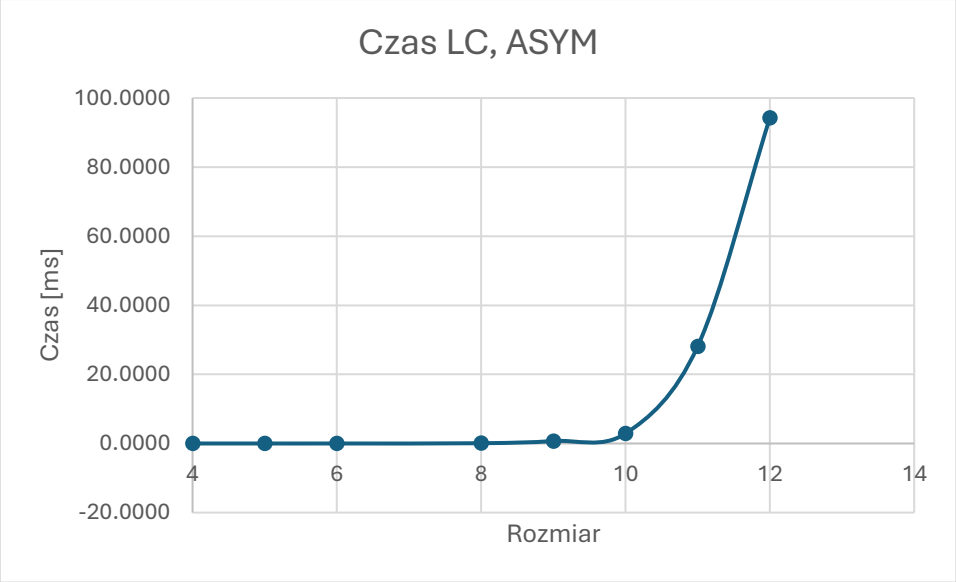
Od 9 do 10 czas wzrasta ponad 10-krotnie (0,2146 ms -> 2,4004 ms).

Od 10 do 11 czas wykonania ponownie rośnie około 7-krotnie (2,4004 ms -> 16,0070 ms).

Od 11 do 12 wzrost wynosi około 4,6-krotnie (16,0070 ms -> 73,3270 ms).

Algorytm Lowest Cost działa na zasadzie przeszukiwania przestrzeni rozwiązań przy jednoczesnym stosowaniu heurystyki ograniczającej eksplorację ścieżek o wysokim koszcie. Chociaż jego złożoność w najgorszym przypadku wynosi O(n!), zastosowanie strategii heurystycznych zmniejsza rzeczywistą liczbę przetwarzanych permutacji, co czyni go bardziej efektywnym w porównaniu do algorytmów pełnego przeszukiwania, takich jak BFS czy DFS. Ale z wyników pomiarów DFS, DFS wychodzi szybszym, chyba że implementacja algorytmu DFS zawiera w sobie błędy.

Analiza pomiarów dla problemu asymetrycznego:



Rysunek 22: Graf opracowania wyników algorytmu *Lowest Cost* dla asymetrycznego problemu

Źrodło: opracowanie własne

Z wyników przedstawionych w tabeli wynika, że czas wykonania algorytmu Lowest Cost (LC) rośnie wraz ze wzrostem rozmiaru problemu. Wzrost ten jest zgodny z teoretyczną złożonością czasową algorytmu i uwzględnia większą złożoność wynikającą z asymetryczności danych wejściowych.

Od rozmiaru 8 do 9 czas wykonania wzrasta prawie 9-krotnie (0,0822 ms -> 0,7188 ms).

Od 9 do 10 wzrost jest około 4-krotny (0,7188 ms -> 2,9024 ms).

Od 10 do 11 czas rośnie blisko 10-krotnie (2,9024 ms -> 28,1476 ms).

Od 11 do 12 wzrost wynosi około 3,35-krotnie (28,1476 ms -> 94,2588 ms).

Algorytm Lowest Cost dla problemu asymetrycznego działa podobnie jak w przypadku problemu symetrycznego, ale ze względu na brak symetrii w danych, nie może stosować optymalizacji redukujących liczbę permutacji do przetworzenia. Teoretyczna złożoność czasowa pozostaje na poziomie O(n!), jednak czas wykonania jest wyższy niż dla problemu symetrycznego, co wynika z większej liczby możliwych rozwiązań do przeanalizowania.

Podsumowanie:

Problemy symetryczny i asymetryczny wykazują podobny wzorzec wzrostu czasu działania algorytmu Lowest Cost, zgodny z teoretyczną złożonością O(n!). Problem asymetryczny jest jednak bardziej czasochłonny, ponieważ brak symetrii uniemożliwia redukcję przestrzeni przeszukiwania, co zwiększa liczbę możliwych permutacji do rozważenia. W przypadku problemu symetrycznego czas wykonania jest niższy dzięki optymalizacjom wynikającym z redukcji symetrii, a różnice te stają się bardziej widoczne przy większych rozmiarach problemu.

Podsumowanie i porównanie trzech algorytmów

Krótkie podsumowanie porównawcze trzech algorytmów:

Wszystkie trzy algorytmy (BFS, DFS i Lowest Cost) mają teoretyczną złożoność czasową O(n!) w kontekście rozwiązania problemu TSP, co oznacza, że ich czas wykonania rośnie wykładniczo wraz ze

wzrostem rozmiaru problemu. Jednakże, ich praktyczne działanie różni się w zależności od implementacji i charakterystyki danych wejściowych.

1. BFS (Breadth-First Search)

Algorytm BFS charakteryzuje się bardzo szybkim wzrostem czasu wykonania przy zwiększaniu rozmiaru problemu, co jest zgodne z jego teoretyczną złożonością O(n!). W praktyce jego działanie jest mało efektywne dla większych rozmiarów problemu, gdyż wymaga przetworzenia ogromnej liczby możliwych permutacji.

2. DFS (Depth-First Search)

DFS również pokazuje wykładniczy wzrost czasu wykonania, ale w praktyce jego czas jest często lepszy od BFS, szczególnie przy mniejszych problemach. W moich pomiarach DFS okazał się najlepszy, ale mogą występować potencjalne błędy lub problemy w implementacji, które mogły wpłynąć na wyniki. Może to obejmować nieoptymalne zarządzanie pamięcią lub efektywność samego algorytmu w danym środowisku.

3. Lowest Cost (LC)

Algorytm Lowest Cost jest teoretycznie najlepszym podejściem w praktyce, ponieważ stosuje heurystyki, które zmniejszają przestrzeń przeszukiwania i mogą znacząco przyspieszyć czas wykonania w porównaniu do algorytmów pełnego przeszukiwania. W moich wynikach LC okazał się być najefektywniejszym algorytmem, co zgadza się z jego teoretyczną przewagą.

Podsumowanie:

W teorii algorytmy BFS, DFS i LC mają złożoność O(n!), ale w praktyce algorytm Lowest Cost jest najskuteczniejszy, dzięki zastosowaniu heurystyk. Algorytm DFS, choć wydaje się lepszy w moich testach, może mieć problemy związane z implementacją lub błędami w kodzie. W rzeczywistości algorytm Lowest Cost powinien być najbardziej efektywnym rozwiązaniem, jeśli jest prawidłowo zaimplementowany i dostosowany do specyfiki problemu.

Jeszcze chciałabym dodać, że dla dużych problemów, nawet już macierzy 12 na 12, lepiej rozwiązać bardziej zaawansowanymi algorytmami, na przykład algorytmem genetycznym, albo mrówkowym.

Bibliografia:

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.geeksforgeeks.org/problems/bfs-traversal-of-graph/1

https://www.youtube.com/watch?v=HZ5YTanv5QE&t

https://www.youtube.com/watch?v=oDqjPvD54Ss

https://www.youtube.com/watch?v=Urx87-NMm6c&t

https://www.youtube.com/watch?v=7fujbpJ0LB4

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

https://en.wikipedia.org/wiki/Depth-first_search

https://www.programiz.com/dsa/graph-dfs

https://www.geeksforgeeks.org/min-cost-path-dp-6/

https://www.youtube.com/watch?v=_lHSawdgXpI

https://www.youtube.com/watch?v=bZkzH5x0SKU&t

https://www.youtube.com/watch?v=smHnz2RHJBY&t

https://www.youtube.com/watch?v=vf-cxgUXcMk&t