**WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI**

**POLITECHNIKA WROCŁAWSKA**

KIERUNEK: INFORMATYKA TECHNICZNA

**Projektowanie efektywnych algorytmów**

**PROJEKT** 1

**Autor**: Maksymilian Łukaszewski 272975

**Prowadzący**: dr inż. Dariusz Banasiak

Czwartek 7:30

1. **Cel Projektu**

Celem projektu było zaprojektowanie i zaimplementowanie 2 algorytmów dla asymetrycznego problemu komiwojażera. Dodatkowo należało zbadać i przeanalizować czas potrzebny na rozwiązanie problemu w zależności od rozmiaru instancji oraz zastosowanego algorytmu. Następujące algorytmy zostały zbadane w ramach projektu:

Algorytm dynamicznego programowania

Algorytm podziału i ograniczeń

Przegląd zupełny (wykorzystany jako kontrast do efektywności pozostałych algorytmów)

1. **Przyjęte założenia**

Podczas projektowania i implementacji przyjęto następujące założenia:

* zadanie zrealizowane jest w ramach jednego programu konsolowego
* wszystkie struktury danych alokowane są dynamicznie
* odległości pomiędzy miastami są liczbami całkowitymi
* program umożliwia wczytanie danych wejściowych z pliku
* program jest stworzony zgodnie z zasadami programowania obiektowego

**Język Programowania i środowisko**

Program, który stworzono w ramach projektu został napisany w języku C++. Badania przeprowadzono na komputerze lenovo Thinkpad T480 z systemem operacyjnym Windows10. Badania przeprowadzono po kompilacji i buildowaniu kodu w trybie *debug.* Podczas przeprowadzania testów wyłączono wszystkie procesy w tle, w celu uzyskania jak najbardziej precyzyjnych wyników.

1. **Wstęp teoretyczny**

Problemy optymalizacyjne o wysokiej złożoności obliczeniowej, takie jak problem komiwojażera (ang. Travelling Salesman Problem, TSP), stanowią wyzwanie dla tradycyjnych metod rozwiązywania problemów. W takich przypadkach stosuje się różne podejścia, w tym algorytmy dokładne, takie jak branch and bound i programowanie dynamiczne, które umożliwiają systematyczne przeszukiwanie przestrzeni rozwiązań, dążąc do znalezienia optymalnych wyników, choć ich czas obliczeniowy może być znaczny w przypadku problemów o dużej skali.

Algorytm podziałów i ograniczeń jest techniką rozwiązywania problemów optymalizacyjnych, która polega na systematycznym podziale przestrzeni rozwiązań na podproblemy (gałęzie) oraz eliminowaniu nieoptymalnych rozwiązań na podstawie określonych ograniczeń (wiązania). Metoda ta umożliwia skuteczne wykluczanie części przestrzeni poszukiwań, co prowadzi do redukcji liczby sprawdzanych rozwiązań i przyspiesza proces znalezienia optymalnego rozwiązania. Wymaga jednak precyzyjnego dobrania funkcji wiązania i strategii podziału, aby jej efektywność była jak największa.

Programowanie dynamiczne, z kolei, jest techniką, która polega na rozwiązywaniu problemów optymalizacyjnych przez dzielenie ich na mniejsze, zachodzące na siebie podproblemy, które są rozwiązywane niezależnie i przechowywane w celu uniknięcia ich wielokrotnego rozwiązywania. Zastosowanie programowania dynamicznego polega na optymalizacji rozwiązań w sposób iteracyjny, zaczynając od najprostszych przypadków i stopniowo rozwiązując bardziej złożone podproblemy, co pozwala na osiągnięcie optymalności globalnej w czasie, który jest zależny od struktury problemu.

Obie te techniki, metoda podziałów i ograniczeń oraz programowanie dynamiczne, różnią się od algorytmów metaheurystycznych, ponieważ zapewniają dokładne rozwiązania, choć w niektórych przypadkach, zwłaszcza przy dużej złożoności problemu, czas ich wykonania może być nieakceptowalnie długi. W przeciwieństwie do heurystyk i metaheurystyk, które poszukują rozwiązań bliskich optymalnym w akceptowalnym czasie, algorytmy te starają się znaleźć rozwiązanie o gwarantowanej optymalności, kosztem większego zużycia zasobów obliczeniowych.

1. **Programowanie dynamiczne**

Programowanie dynamiczne (ang. dynamic programming, DP) jest techniką optymalizacji, która polega na rozwiązywaniu problemów poprzez dzielenie ich na mniejsze, zachodzące na siebie podproblemy, które są rozwiązywane niezależnie i przechowywane w celu uniknięcia wielokrotnego obliczania tych samych wyników. W każdym kroku algorytm podejmuje decyzję na podstawie już rozwiązanych podproblemów, co pozwala na stopniowe budowanie optymalnego rozwiązania globalnego.

Metoda ta jest szczególnie efektywna w problemach, w których rozwiązanie problemu można wyrazić jako funkcję zależną od rozwiązań jego podproblemów. Programowanie dynamiczne jest wykorzystywane w problemach takich jak najkrótsza ścieżka w grafie (algorytm Floyda-Warshalla), problem plecakowy w wersji całkowitej czy problem komiwojażera, gdy przestrzeń rozwiązań jest rozkładająca się w sposób, który pozwala na zapamiętywanie wyników pośrednich.

Podstawowe etapy algorytmu programowania dynamicznego można przedstawić następująco:

* Zainicjuj rozwiązania dla najprostszych podproblemów (np. dla najmniejszych podzbiorów danych).
* Stopniowo buduj rozwiązanie dla większych podproblemów, wykorzystując wyniki wcześniejszych obliczeń.
* Przechowuj rozwiązania podproblemów w tabeli (np. w tablicy lub macierzy), aby uniknąć wielokrotnego ich rozwiązywania.
* Na koniec, zrekonstruuj optymalne rozwiązanie na podstawie wyników przechowanych w tabeli.

Programowanie dynamiczne charakteryzuje się tym, że zapewnia optymalność rozwiązania, jednak w przypadku dużych przestrzeni rozwiązań może prowadzić do dużych wymagań pamięciowych oraz czasowych. Mimo to, w przypadku problemów, gdzie istnieje naturalna struktura dzieląca problem na podproblemy, programowanie dynamiczne jest niezwykle skuteczną metodą.

1. **Metoda podziałów i ograniczeń**  
   Metoda podziałów i ograniczeń (ang. branch and bound) jest techniką stosowaną w rozwiązywaniu problemów optymalizacyjnych, która łączy eksplorację przestrzeni rozwiązań z inteligentnym ograniczaniem zbioru możliwych rozwiązań. Polega ona na dzieleniu przestrzeni rozwiązań na mniejsze podproblemy (tzw. gałęzie), które są następnie analizowane, a te, które z powodu naruszenia określonych ograniczeń są mniej obiecujące, są eliminowane.

Algorytm *branch and bound* opiera się na dwóch kluczowych mechanizmach: podziale (ang. branching), który dzieli przestrzeń rozwiązań na mniejsze podproblemy, oraz ograniczeniu (ang. bounding), które pozwala na odrzucenie części rozwiązań, które nie mogą prowadzić do optymalnego rozwiązania. W każdym kroku algorytm oblicza tzw. wiązanie, które stanowi górną lub dolną granicę dla najlepszych możliwych rozwiązań w danym podproblemy, i na tej podstawie podejmuje decyzję, które gałęzie należy kontynuować, a które odrzucić.

Podstawowe etapy algorytmu podziałów i ograniczeń na przykładzie problemu komiwojażera (TSP) mogą wyglądać następująco:

* Zainicjuj rozwiązanie początkowe (np. zaczynając od konkretnego wierzchołka w grafie).
* Dziel przestrzeń rozwiązań na mniejsze podproblemy (np. rozważając różne kolejne miasta do odwiedzenia).
* Oblicz wiązanie dla każdego podproblemu, które pozwala ocenić, czy warto kontynuować analizę tej gałęzi.
* Odrzuć te podproblemy, które nie prowadzą do lepszego rozwiązania niż już znalezione.
* Powtarzaj kroki 2-4, aż znajdziesz optymalne rozwiązanie.

Metoda *branch and bound* jest szczególnie efektywna w rozwiązywaniu problemów, gdzie przestrzeń rozwiązań jest złożona i istnieje możliwość wykluczenia dużej liczby nieoptymalnych rozwiązań na wczesnym etapie analizy. Choć metoda ta zapewnia optymalność rozwiązania, jej czas obliczeniowy może być znaczny, zwłaszcza w przypadku dużych problemów. Jednak dzięki zastosowaniu inteligentnych mechanizmów ograniczeń, *branch and bound* jest w stanie rozwiązywać problemy, które są trudne do uchwycenia innymi algorytmami dokładnymi.

1. **Klasy Pomocnicze**

**7.1. Klasa Matrix**

**Konstruktor i Destruktor**

* **Matrix(int size)**: Inicjalizuje macierz o zadanym rozmiarze.
* **~Matrix()**: Zwalnia pamięć zaalokowaną dla macierzy.

**Operacje na macierzy**

* **freeMatrix()**: Zwalnia pamięć zaalokowaną dla macierzy.
* **loadFromFile(const string& filename)**: Ładuje macierz z pliku. Odczytuje rozmiar macierzy i jej elementy.
* **generateRandom(int newSize, int min, int max)**: Generuje losową macierz o określonym rozmiarze, z wartościami w podanym zakresie (min, max).
* **print()**: Wypisuje macierz na ekran z sformatowanym wyjściem.
* **getSize()**: Zwraca rozmiar macierzy.
* **getMatrix()**: Zwraca wskaźnik na surową tablicę macierzy.
* **getValue(int row, int col)**: Zwraca wartość z macierzy na podanych (wiersz, kolumna).

**7.2. Klasa Timer**

**Operacje na timerze**

* **Timer()**: Konstruktor inicjalizuje elapsedTime na 0.
* **start()**: Rozpoczyna pomiar czasu, zapisując czas początkowy.
* **stop()**: Zatrzymuje pomiar czasu, zapisując czas końcowy i obliczając czas wykonania w
* milisekundach.
* **getElapsed()**: Zwraca czas wykonania.
* \**printElapsed(const char* label)\*\*: Wypisuje czas wykonania z etykietą.

1. **Algorytm Podziałów i Ograniczeń (Branch and Bound)**

**8.1 Opis teoretyczny**

Algorytm Podziałów i Ograniczeń (Branch and Bound, BnB) jest metodą optymalizacji wykorzystywaną do rozwiązywania problemów kombinatorycznych, w tym problemu komiwojażera (TSP). Główna zasada tego algorytmu polega na systematycznym eksplorowaniu przestrzeni rozwiązań przez jej podział na mniejsze podproblemy, przy jednoczesnym ograniczeniu (bound) przestrzeni, które są rozważane, na podstawie już poznanych wyników.

Kluczowe elementy algorytmu BnB:

* **Podział (Branching)**: Polega na dzieleniu przestrzeni rozwiązań na mniejsze podproblemy (części przestrzeni rozwiązań), które są rozwiązywane rekurencyjnie. W przypadku problemu komiwojażera, polega to na rozważeniu różnych ścieżek do odwiedzenia kolejnych miast.
* **Ograniczenie (Bounding)**: W ramach każdego podproblemu obliczany jest dolny limit (lower bound) najlepszego możliwego wyniku dla danego podproblemu. Jeśli ten dolny limit jest gorszy niż najlepsze znane rozwiązanie (np. koszt przejazdu w przypadku TSP), to dalsze eksplorowanie tego podproblemu jest pomijane, ponieważ wiadomo, że nie da on lepszego wyniku.
* **Zasada przycinania (Pruning)**: Jeśli dolny limit podproblemu przekracza wartość najlepszej znanej ścieżki (kosztu), podproblem jest odrzucany (przycinany). Dzięki temu algorytm nie bada nieoptymalnych dróg, oszczędzając czas obliczeniowy.

**8.2 Złożoność obliczeniowa**

Złożoność algorytmu BnB zależy od kilku czynników: liczby miast (n), liczby generowanych podproblemów oraz obliczeń związanych z oceną ograniczeń. W algorytmie BnB złożoność obliczeniowa może być trudna do określenia w sposób dokładny, ponieważ zależy od jakości przycinania (branching) oraz skuteczności obliczania ograniczeń. Niemniej jednak, w najgorszym przypadku, algorytm może eksplorować przestrzeń rozwiązań w sposób wykładniczy, co skutkuje złożonością O(n!).

Optymalizacja algorytmu poprzez skuteczne przycinanie, np. przez dobre oszacowanie dolnych granic (lower bounds), może znacząco zmniejszyć rzeczywistą złożoność obliczeniową.

**8.3 Opis implementacji**

Algorytm BnB operuje na kilku kluczowych elementach:

* **Obliczanie dolnej granicy (Lower Bound)**: Algorytm w każdym etapie oblicza dolną granicę kosztu rozwiązania dla danego podproblemu. W przypadku TSP jest to suma najtańszych dostępnych krawędzi (dróg) z danego wierzchołka do pozostałych, które jeszcze nie zostały odwiedzone.
* **Rekurencyjne dzielenie przestrzeni rozwiązań (Branching)**: Proces rozwiązywania problemu jest rekurencyjny. Na każdym etapie algorytm rozważa, które miasto należy odwiedzić następnie, a potem wywołuje rekurencyjnie algorytm dla tego podproblemu. Jeśli ograniczenie dla tego podproblemu nie pozwala na poprawienie wyniku, algorytm go odrzuca.
* **Sprawdzanie ścieżki**: Gdy algorytm dotrze do pełnej ścieżki (wszystkie miasta zostały odwiedzone), oblicza całkowity koszt podróży. Jeśli koszt jest mniejszy niż najlepszy dotychczasowy, aktualizuje wynik i zapisuje ścieżkę.
* **Zasady przycinania**: Jeśli dolna granica dla danego podproblemu jest większa od najlepszego znanego rozwiązania, podproblem jest odrzucany i nie jest dalej eksplorowany.

**8.4 Struktura algorytmu BnB**

* **Inicjalizacja**: Na początku algorytm ustawia początkowe rozwiązanie i wylicza koszt dla tego rozwiązania. Dla każdego wierzchołka obliczany jest minimalny koszt krawędzi, a przestrzeń rozwiązań jest dzielona.
* **Pętla główna**: W tej pętli algorytm rozważa kolejne podproblemy. Dla każdego rozwiązywanego podproblemu, sprawdzane są jego dolne granice. Jeśli granica jest obiecująca, algorytm kontynuuje eksplorację, przechodząc do kolejnego etapu.
* **Odrzucanie nieoptymalnych dróg**: Podproblemy, które są nieoptymalne, są natychmiast odrzucane.
* **Zakończenie**: Algorytm kończy działanie, gdy nie ma już podproblemów do zbadania, a najlepsza znaleziona ścieżka jest ostatecznym rozwiązaniem.

**8.5 Zastosowanie w rozwiązaniu problemu komiwojażera (TSP)**

Dzięki zastosowaniu metody BnB, problem komiwojażera (TSP) jest rozwiązywany przez systematyczne poszukiwanie najkrótszej drogi, której koszt jest minimalizowany poprzez eliminowanie dróg, które prowadzą do rozwiązania gorszego niż najlepsze dotychczasowe. Algorytm jest szczególnie przydatny w problemach, w których przestrzeń rozwiązań jest bardzo duża, ale istnieje możliwość przycinania nieoptymalnych gałęzi, co pozwala na znalezienie rozwiązania w krótszym czasie.

1. **Dynamiczne Programowanie**

**9.1 Opis teoretyczny**

Dynamiczne programowanie (DP) to metoda rozwiązywania problemów optymalizacyjnych poprzez dzielenie dużego problemu na mniejsze podproblemy i zapamiętywanie wyników tych podproblemów, aby uniknąć ich ponownego rozwiązywania. Jest to technika bardzo efektywna w przypadkach, gdy problem można rozbić na mniejsze, powtarzające się podproblemy.

W kontekście problemu komiwojażera (TSP), celem jest znalezienie najkrótszej możliwej trasy, która odwiedza każde miasto dokładnie raz i wraca do punktu początkowego. Dynamiczne programowanie pozwala rozwiązać ten problem poprzez iterację po różnych podzbiorach miast, co znacząco zmniejsza złożoność rozwiązania w porównaniu do klasycznych algorytmów przeszukiwania.

**9.2 Podejście z Maskami Bitowymi**

W algorytmie dynamicznego programowania dla problemu komiwojażera, wykorzystuje się tzw. maski bitowe, które reprezentują różne podzbiory miast. Każdy bit w masce oznacza, czy dane miasto zostało odwiedzone (bit ustawiony na 1) czy nie (bit ustawiony na 0). To pozwala na efektywne śledzenie, które miasta zostały odwiedzone i jaki jest koszt przejścia między nimi.

Algorytm ten przechowuje informacje o minimalnych kosztach przejść dla różnych podzbiorów miast, co pozwala na iteracyjne obliczenie minimalnego kosztu całej podróży. W każdej iteracji sprawdzane są możliwe przejścia między miastami w danym podzbiorze, a koszty te są zapisywane, aby uniknąć ich ponownego obliczania.

**9.3 Złożoność Obliczeniowa**

Złożoność obliczeniowa algorytmu dynamicznego programowania z maskami bitowymi wynosi O(n² \* 2ⁿ), gdzie n to liczba miast. W szczególności:

* 2ⁿ: liczba możliwych podzbiorów miast (maski bitowe),
* n²: liczba operacji, które trzeba wykonać dla każdej maski bitowej w celu sprawdzenia przejść między miastami.

Z tego powodu algorytm jest efektywny tylko dla małych instancji problemu (np. dla n mniejszych niż 20), ponieważ złożoność rośnie wykładniczo wraz ze wzrostem liczby miast.

**9.4 Kroki Algorytmu**

**Inicjalizacja:**

Na początku dla każdego podzbioru miast (maski bitowej) oraz dla każdego miasta, ustawiane są wartości początkowe dla kosztów przejść. Zwykle początkowy koszt jest ustawiany na "nieskończoność", a dla początkowego miasta jest to wartość 0 (jeśli zaczynamy w tym mieście).

**Iteracja po maskach bitowych:**

Algorytm iteruje przez wszystkie możliwe podzbiory miast, reprezentowane jako maski bitowe. Dla każdej maski, sprawdzane są możliwe przejścia pomiędzy miastami, które jeszcze nie zostały odwiedzone w danym podzbiorze.

**Obliczenie minimalnego kosztu powrotu:**

Po obliczeniu minimalnych kosztów dla wszystkich podzbiorów miast, algorytm znajduje najtańszą trasę, która powraca do miasta początkowego, bazując na obliczonych wcześniej kosztach.

**Rekonstrukcja ścieżki:**

Na koniec, po obliczeniu minimalnego kosztu, algorytm rekonstruuje optymalną trasę, prześledzając zapamiętane informacje o poprzednich miastach w tabeli, co pozwala uzyskać pełną ścieżkę odwiedzin.

**9.5 Rekonstrukcja Ścieżki**

Po obliczeniu minimalnego kosztu przejścia przez wszystkie miasta, ważnym krokiem jest rekonstrukcja ścieżki, czyli ustalenie kolejności odwiedzin miast. W tym celu algorytm śledzi, z którego miasta przybyliśmy do aktualnego miasta, korzystając z zapisanych informacji o poprzednich miastach (np. w tablicy pomocniczej). Proces ten jest powtarzany, aż do powrotu do miasta początkowego.

**9.6 Przykład Zastosowania**

Algorytm dynamicznego programowania z maskami bitowymi jest szczególnie skuteczny w rozwiązywaniu problemu komiwojażera, gdzie:

* Należy znaleźć najkrótszą trasę przez n miast,
* Trasa musi zaczynać się i kończyć w tym samym mieście,
* Każde miasto musi być odwiedzone dokładnie raz.

Dzięki zastosowaniu dynamicznego programowania, możliwe jest znaczne zmniejszenie złożoności obliczeniowej w porównaniu do podejść prostszych, takich jak brute force.

**9.7 Praktyczne Zastosowanie**

Przykładem zastosowania tego algorytmu może być rozwiązanie problemu w aplikacji planowania tras w firmach logistycznych, gdzie celem jest zoptymalizowanie trasy transportu pomiędzy wieloma punktami (miastami). Dzięki dynamicznemu programowaniu, firma może szybko znaleźć najtańszą trasę, nawet w przypadku większej liczby punktów, przy zachowaniu dobrej efektywności obliczeniowej.

1. **WYNIKI**

A white sheet with black text and numbers

Description automatically generated

Na czerwono zaznaczono wyliczone hipotetyczne wyniki w przypadku metody przeglądu zupełnego, ze względu na niemożliwość faktycznych pomiarów. Dla n=14, bruteforce działał koło 600000 milisekund, czyli ~10 minut. A graph with numbers and lines

Description automatically generatedA graph with colored lines and numbers

Description automatically generatedA graph with colored lines and numbers

Description automatically generated

Na rysunkach powyżej 3 wykresy z kolejno odejmowanymi kolejnymi punktami dla pokazania skali „prędkości” rośnięcia czasu wykonywania danych algorytmów.

**10.WNIOSKI**

Przegląd zupełny naturalnie jest najmniej efektywny. Przy złożoności O(n!), jego czas działania dla n=20 oszacowano na około 50 lat, korzystając z dostępnego środowiska badań. Programowanie dynamiczne zaczyna być bardziej efektywne od przeglądu zupełnego dopiero dla n>5, przy złożoności O(n^2⋅2^n)O. Algorytm podziału i ograniczeń powinien znajdować się pomiędzy O(2^n) a O(n!), zależnie od przypadku, co odzwierciedlają wyniki.

**Dodatkowe wnioski:**

**10.1.Przegląd zupełny**:  
Jego praktyczne zastosowanie ogranicza się jedynie do bardzo małych wartości nnn (do około n=7n = 7n=7). Dla większych wartości liczba operacji staje się astronomiczna, co czyni tę metodę niepraktyczną. Jest to jednak najprostsza metoda do zrozumienia i implementacji.

**10.2.Programowanie dynamiczne**:  
Okazało się bardziej efektywne w przypadku większych wartości nnn, co wynika z jego znacznie lepszej złożoności obliczeniowej w porównaniu do przeglądu zupełnego. Mimo to, rosnąca wykładniczo liczba operacji dla n>20n > 20n>20 ogranicza jego zastosowanie dla bardzo dużych problemów. Dynamiczne programowanie jest szczególnie użyteczne w środowiskach, gdzie pamięć jest ograniczona, ale czas działania jest akceptowalny.

**10.3.Podział i ograniczenia (Branch and Bound)**:  
Algorytm ten wyróżnia się swoją uniwersalnością i zdolnością do wykorzystania ograniczeń w celu odrzucania nieopłacalnych gałęzi, co czyni go szczególnie skutecznym w wielu przypadkach. Jak wynika z badań, dla n=20n = 20n=20 działa on znacznie szybciej niż przegląd zupełny (różnica kilku rzędów wielkości), ale jego efektywność w dużej mierze zależy od struktury problemu i skuteczności ograniczeń. Algorytm ten jest dobrym kompromisem między czasem działania a łatwością implementacji.

**10.4.Porównanie metod**:  
Każda z analizowanych metod ma swoje zastosowanie w zależności od wielkości problemu oraz zasobów sprzętowych. Dla małych wartości nnn przegląd zupełny może być wystarczający, dla średnich wartości nnn dynamiczne programowanie staje się najlepszym wyborem, a dla większych wartości nnn warto rozważyć podział i ograniczenia.

**10.5.Zalecenia**:  
W praktycznych zastosowaniach algorytm podziału i ograniczeń wydaje się najbardziej uniwersalnym rozwiązaniem, zwłaszcza w środowiskach, gdzie liczba miast (lub punktów w problemie) może być zmienna. Jednak dla problemów, gdzie gwarancja optymalności jest kluczowa, programowanie dynamiczne może być lepszym wyborem, jeśli zasoby na to pozwalają. Przegląd zupełny powinien być wykorzystywany jedynie jako narzędzie do weryfikacji wyników w małych przypadkach testowych.