Projektowanie Efektywnych Algorytmów

Zadanie projektowe nr 1

Implementacja i analiza efektywności algorytmu podziału i ograniczeń, przeglądu zupełnego i programowania dynamicznego na podstawie problemu komiwojażera

|  |  |
| --- | --- |
| *Autor:* | *Prowadzący* |
| Damian Owczarczyk | Mgr inż. Antoni Sterna |
| *Grupa projektowa* | *Data oddania* |
| Czwartek 17:05 | 15.11.2018 |

Spis treści

[1 Cel projektu. 3](#_Toc530060266)

[2 Wstęp teoretyczny 3](#_Toc530060267)

[3 Przykłady praktyczne 4](#_Toc530060268)

[3.1 Programowanie dynamiczne 4](#_Toc530060269)

[3.2 Podziały i ograniczenia 5](#_Toc530060270)

[4 Opis implementacji 6](#_Toc530060271)

[5 Wyniki pomiarów 6](#_Toc530060272)

[5.1 Algorytm podziałów i ograniczeń 6](#_Toc530060273)

[5.2 Algorytm przeglądu zupełnego 7](#_Toc530060274)

[5.3 Algorytm programowania dynamicznego 8](#_Toc530060275)

[6 Wnioski 9](#_Toc530060276)

# Cel projektu.

Celem projektu jest zaimplementowanie oraz dokonanie analizy efektywności algorytmów podziału i ograniczeń, przeglądu zupełnego i programowania dynamicznego dla asymetrycznego problemu komiwojażera.

# Wstęp teoretyczny

Problem komiwojażera jest problemem optymalizacyjnym polegającym na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie zupełnym. Należy on do zbioru problemów NP-trudnych. W zadaniu, które mieliśmy wykonać rozważamy Asymetryczny Problem Komiwojażera, który różni się od symetrycznego tym, że odległość np. z miasta A do miasta B nie musi być taka sama jak odległość z miasta B do miasta A. Innymi słowy, rozpatrujemy problem, którym pewien handlarz musi odwiedzić wszystkie możliwe miasta, a następnie powrócić do tego, z którego wyruszył. Co więcej, droga ta musi być odbyta możliwie najmniejszym kosztem. Chociaż zadanie wydaje się banalne, to na większą skalę, przy np. 10 miastach, stanowi już niemały problem. Z tego też powodu zaczęły powstawać algorytmy, które umożliwiają obliczenie najkorzystniejszej drogi szybciej niż poprzez sprawdzenie każdej możliwej i wybór tej najlepszej.

W moim programie zaimplementowałem trzy algorytmy. Jednym z nich jest metoda Brute Force, czyli rozpatrywanie każdej możliwej ścieżki i zapamiętanie tej najlepszej. Niestety, ale taki algorytm ma bardzo dużą złożoność obliczeniową, która wynosi 𝑂 = (𝑛 − 1)!

Druga zaimplementowana technika to programowanie dynamiczne, jest ono metodą rozwiązywania złożonych problemów, poprzez rozbicie ich na zbiór podproblemów o mniejszej złożoności, przy założeniu, że każdy podproblem rozważany jest jedynie raz, a wynik jego analizy przechowywany jest do wykorzystania w późniejszych obliczeniach. Dla problemu komiwojażera, najlepszym algorytmem wykorzystującym tę metodę, jest algorytm Helda-Karpa, posiadający złożoność czasową *O(n2 \* 2n)*.

Algorytm wykorzystuje tablice 2(*n−*1) elementowa indeksowana od zera. Tablica jest wypełniana według algorytmu Helda-Karpa jak i również znajdujemy minimalny koszt przejścia instancji. Indeks elementu jest również maska, która mówi o miastach, które się odwiedziło. Przykładowo dla 5 miast ostatnim indeksem jest 15, czyli 11112 - ta maska mówi o odwiedzeniu wszystkich miast poza ostatnim. Mimo 5 punktów potrzebne są nam tylko 4 bity, gdyż ostatnie miasto jest już wybrane i znajdujemy wierzchołek, który zapewni nam najmniejszy koszt podróży od ostatniego wierzchołka. W tym elemencie znajduje się lista elementów mówiących o koszcie przejścia tych wszystkich miast wraz ze wskazaniem na to który punkt był ostatnim. Wybierając najmniejszy element, na przykład 2 przechodzimy do maski 10112, gdzie powtarzamy algorytm, aż do całkowitego odtworzenia drogi (na samym początku należy pamiętać o dodaniu kosztu powrotu do ostatniego wierzchołka, by można było wybrać poprawny element).

Trzecim z algorytmów jest algorytm Little’a, który jest algorytmem podziału i ograniczeń (B&B). W tym algorytmie poszukuje się najmniejszych wartości w każdym z wierszy i następnie odejmuje się wartość najmniejszą od każdego z wierszy. Następnie wykonuje się analogiczną czynność z kolumnami. Suma wszystkich najmniejszych elementów jest początkowym dolnym ograniczeniem. Następnie zostaje wykonany wybór skreślonej kolumny i wiersza na podstawie otrzymanego dolnego ograniczenia po skreśleniu. Po skreśleniu rozpatrywana jest macierz wielkości o jeden mniejsza od poprzedniej. Powyższe kroki są powtarzane aż do osiągnięcia macierzy o rozmiarze 2x2. Na końcu dodaje się początkowe miasto i tak powstaje droga wyświetlona w programie. Minusem tego algorytmu jest duża złożoność pamięciowa, gdyż dla każdego elementu tworzymy uaktualniona dla danego elementu kopie macierzy kosztów przejścia pomiędzy wierzchołkami. W najgorszym przypadku odwiedzimy każdy wierzchołek, tak jak przy przeglądzie zupełnym, co daje nam pesymistyczną złożoność obliczeniową na równą 𝑂 = (𝑛 − 1)! Jednakże jest to bardzo rzadko spotykane. W związku z tym średnia złożoność obliczeniowa będzie mniejsza od złożoności obliczeniowej algorytmu przeglądu zupełnego.

# Przykłady praktyczne

## Programowanie dynamiczne

1. Określamy funkcję koszt(S, p), gdzie S jest zbiorem wierzchołków, a p punktem kończącym ścieżkę, w następujący sposób:

* Jeżeli S = 1, to koszt(S, p) = wadze krawędzi określonej w macierzy jako d(0, p)
* Jeżeli S > 1, to koszt(S, p) = minx (koszt(S-{p}, x) + d(x, p))

1. Stosując ten algorytm, dla przykładowego zbioru czterech miast, w pierwszej iteracji otrzymamy:

* koszt({1}, 1) = d(0, 1)
* koszt({2}, 2) = d(0, 2)
* koszt({3}, 3) = d(0, 3)

1. Następnie, należy zacząć rozważać zbiory 2-elementowe, stosując ten sam wzór:

* koszt({1, 2}, 1) = min(koszt({2}, 2) + d(2, 1))
* koszt({1, 2}, 2) = min(koszt({1}, 1) + d(1, 2))
* koszt({1, 3}, 1) = min(koszt({3}, 3) + d(3, 1))
* koszt({1, 3}, 3) = min(koszt({1}, 1) + d(1, 3))
* koszt({2, 3}, 2) = min(koszt({3}, 3) + d(3, 2))
* koszt({2, 3}, 3) = min(koszt({2}, 2) + d(2, 3))

1. Kolejno, dla zbiorów 3-elementowych:

* koszt({1, 2, 3}, 1) = min(koszt({2, 3}, 2) + d(2, 1), koszt({2, 3}, 3) + d(3, 1))
* koszt({1, 2, 3}, 2) = min(koszt({1, 3}, 1) + d(1, 2), koszt({1, 3}, 3) + d(3, 2))
* koszt({1, 2, 3}, 3) = min(koszt({1, 2}, 1) + d(1, 3), koszt({1, 2}, 2) + d(2, 3))

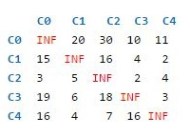
1. Zbiory 3-elementowe to maksymalne zbiory, ponieważ rozważamy 4 wierzchołki, z których jeden jest wierzchołkiem startowym. Możemy więc wyznaczyć wzór końcowy:

min(koszt({1, 2, 3}, 1) + d(1, 0), koszt({1, 2, 3}, 2) + d(2, 0), koszt({1, 2, 3}, 3) + d(3, 0))

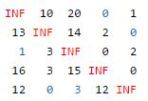
Obliczenie wartości otrzymanego wzoru pozwoli nam uzyskać wartość liczbową, reprezentującą minimalną wagę krawędzi, jakie tworzą cykl Hamiltona w zadanym grafie. Następnie, należy przejść po kolejnych iteracjach algorytmu, w celu uzyskania kolejności wierzchołków wyznaczających najkrótszą ścieżkę.

## Podziały i ograniczenia

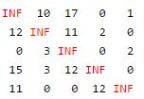
1. Posiadamy poniższą macierz kosztów.



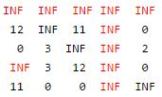
1. Wyznaczamy najmniejsze wartości w danym wierszu macierzy. Otrzymujemy tablicę najmniejszych wartości w wierszu [10 2 2 3 4].
2. Odejmujemy najmniejszą wartość w wierszu od każdej wartości w wierszu. Powstaje zredukowana macierz kosztów.



1. Analogicznie postępujemy z kolumnami. Otrzymujemy tablicę [1 0 3 0 0] i zredukowaną macierz



1. Teraz możemy obliczyć dolne ograniczenie poprzez dodanie wszystkich najmniejszych wartości wyznaczonych w poprzednich krokach (dolne ograniczenie=[10 2 2 3 4] + [1 0 3 0 0]=25
2. Następnie wybieramy miasto do którego podróżujemy na podstawie całkowitego kosztu podróży ze wzoru (dolne ograniczenie miasta 0+koszt wierzchołka(0,x) + dolne ograniczenie ścieżki startującej w mieście x, gdzie x oznacza dowolne miasto, do którego można podróżować z miasta 0). Wybieramy podróż o najmniejszym koszcie, więc podróżujemy z miasta 0 do miasta 3. (koszt=25+0+0=25)
3. Zmieniamy wszystkie wartości z wiersza 0 i kolumny 3 i komórkę o indeksie (3,0) w macierzy kosztów na INF, co daje nam poniższą macierz. Wykonywane jest to w celu zamknięcia drogi powrotnej z miasta 3 do miasta 0 i uzyskania zredukowanej macierzy kosztów, która będzie wykorzystywana dalej w algorytmie.



1. Miasto 3 zostaje dodane do drzewa, które przechowuje najlepszą ścieżkę. Następnie wykonywane są powyższe kroki, aż drzewo się nie zapełni ilością elementów równą liczbie miast. Na końcu dodawane jest miasto początkowe (miasto 0) i wyświetlany jest wynik oraz koszt.

# Opis implementacji

Algorytmy operują na dwuwymiarowych dynamicznych tablicach. Do pomiaru czasu wykorzystana została biblioteka <time.h>, zawarta w zbiorze standardowych bibliotek języka C++. Biblioteka ta operuje na liczbie cykli procesora, co pozwala na bardzo dokładny pomiar czasu. Program pobiera czas przed i po wykonaniu funkcji, a następnie oblicza różnicę czasu bazując na ilości okresów przebiegu procesora.

Przebieg pomiaru:

Będziemy mierzyć czas wykonywania operacji dla liczb miast: 5, 7, 9, 11, 13, 15, 17. Jako że wynik zależy od rozkładu danych (generacja liczb pseudolosowych) pomiar zostanie powtórzony stokrotnie, a następnie zostanie uśredniony.

# Wyniki pomiarów

## Algorytm podziałów i ograniczeń

|  |  |
| --- | --- |
| Algorytm Little’a | |
| Wierzchołki | Czas[ms] |
| 5 | 0,04 |
| 7 | 0,18 |
| 9 | 0,72 |
| 11 | 4,78 |
| 13 | 41,65 |
| 15 | 697,04 |
| 17 | 3335,1 |

*Tabela 1. Wyniki pomiarów dla algorytmu Little’a*

*Rysunek 1. Wyniki pomiarów dla algorytmu Little’a*

## Algorytm przeglądu zupełnego

|  |  |
| --- | --- |
| Przegląd zupełny | |
| Wierzchołki | Czas[ms] |
| 5 | 0,01 |
| 7 | 0,12 |
| 9 | 5,78 |
| 11 | 939,74 |
| 13 | 163742 |

*Tabela 2. Wyniki pomiarów dla przeglądu zupełnego*

*Rysunek 2. Wyniki pomiarów dla przeglądu zupełnego*

## Algorytm programowania dynamicznego

|  |  |
| --- | --- |
| Algorytm Helda-Karpa | |
| Wierzchołki | Czas[ms] |
| 5 | 0,0006 |
| 7 | 0,002 |
| 9 | 0,11 |
| 11 | 0,62 |
| 13 | 3,77 |
| 15 | 20,14 |
| 17 | 98,06 |

*Tabela 3. Wyniki pomiarów dla algorytmu Helda-Karpa*

*Rysunek 3. Wyniki pomiarów dla algorytmu Helda-Karpa*

# Wnioski

* Algorytm przeglądu zupełnego jest najmniej wydajnym algorytmem ze względu na jego złożoność obliczeniową. Przy 11 wierzchołkach traci on efektywność. Jednakże jest jedynym algorytmem z powyżej testowanych, który gwarantuje odnalezienie najlepszego rozwiązania.
* Algorytm podziałów i ograniczeń staje się nieefektywny przy ilości wierzchołków większej niż 13. Średnio jest on zdecydowanie szybszy od przeglądu zupełnego, jednakże posiada dużą złożoność pamięciową i nie gwarantuje odnalezienia najlepszego rozwiązania.
* Algorytm programowania dynamicznego jest najszybszy ze wszystkich testowanych algorytmów. Nawet dla 17 miast zachowuje on swoją efektywność. Posiada także znacznie mniejszą złożoność pamięciową od algorytmu podziałów i ograniczeń. Jednakże minusem jest brak gwarancji odnalezienia najlepszego rozwiązania.
* W przypadku poszukiwania dokładnego rozwiązania problemu komiwojażera powinniśmy wybrać algorytm przeglądu zupełnego. Niestety przy dużej ilości wierzchołków (miast) otrzymanie rezultatu może zajmować dużo czasu.
* W przypadku poszukiwania oszacowanego najlepszego rozwiązania problemu komiwojażera należy wybrać algorytm programowania dynamicznego. Jest on zdecydowanie szybszy od algorytmów podziału i ograniczeń oraz przeglądu zupełnego i ma zdecydowanie mniejszą złożoność pamięciową.
* Możliwe jest, że istnieje kilka ścieżek optymalnych dla danej instancji problemu. W związku z tym różne algorytmy mogą podać różne ścieżki końcowe i te ścieżki mogą okazać się optymalnymi.
* Dla algorytmu przeglądu zupełnego zostały pominięte pomiary dla 15 i 17 wierzchołków (miast) z powodu ogromnych czasów rozwiązań problemu związanych ze złożonością obliczeniową.