Dominika Dolik, 235853

Prowadzący: dr inż. Dariusz Banasiak

Termin: środa, 17:25

Projektowanie Efektywnych Algorytmów

Zadanie Projektowe nr 1

Przegląd zupełny i programowanie dynamiczne dla problemu komiwojażera

Wstęp teoretyczny

Problem komiwojażera (ang. travelling salesman problem, TSP) jest zagadnieniem optymalizacyjnym. Nazwa pochodzi od typowej ilustracji problemu, przedstawiającego go z punktu widzenia wędrownego sprzedawcy (komiwojażera): dane jest n miast, które komiwojażer ma odwiedzić oraz odległość między tymi miastami/koszt podróży. Celem jest znalezienie najkrótszej/najtańszej ścieżki łączącej wszystkie miasta, zaczynającej się i kończącej w jednym punkcie.

Problem komiwojażera jest problemem NP-trudnym, co oznacza, że złożoność obliczania poprawnego rozwiązania wzrasta wykładniczo i nie są znane sposoby rozwiązywania go w czasie wielomianowym.

W terminologii grafów rozwiązanie problemu komiwojażera polega na odnalezieniu w nim cyklu Hamiltona – cyklu, w którym każdy wierzchołek jest odwiedzany dokładnie jeden raz. Znalezienie cyklu Hamiltona o minimalnej sumie wag krawędzi jest równoważne rozwiązaniu problemu komiwojażera.

Wykorzystane algorytmy

- ## Przegląd zupełny (Brute Force)

Pierwszym i najprostszym rozwiązaniem jest przegląd zupełny (Brute Force). Znajduje on wszystkie możliwe cykle Hamiltona w grafie, co oznacza, że zawsze zostanie znalezione rozwiązanie optymalne. Jego podstawową wadą jest złożoność czasowa $O(n^n)$ i konieczność sprawdzenia (n-1)! możliwych ścieżek. Z tego względu znalezienie minimalnej ścieżki potrafi zająć naprawdę długi czas.

Implementacja tego sposobu polegała na permutacji zbioru n-elementowego (gdzie n jest liczbą wierzchołków grafu – tutaj: liczbą miast), a następnie na obliczaniu wartości ścieżki dla danego podciągu i porównanie go z ostatnią najmniejszą znalezioną wartością ścieżki. Jeśli ścieżka ma mniejszą wagę – jest ona przypisywana jako najmniejsza znaleziona i jednocześnie jako rozwiązanie optymalne.

- ## Programowanie dynamiczne

Programowanie dynamiczne jest strategią projektowania algorytmów i jednocześnie alternatywą dla zagadnień rozwiązywanych za pomocą algorytmów zachłannych. Opiera się na podziale rozwiązywanego problemu na podproblemy względem kilku parametrów. W tym przypadku rozwiązanie polega na podziale zbioru wierzchołków, przez które chcemy przejść na mniejsze podzbiory, a następnie na obliczaniu wartości cząstkowych dla najmniejszych podproblemów i wykorzystaniu ich dla wyników największych. Algorytm ten zatem działa rekurencyjnie.

Najlepszym algorytmem wykorzystującym metodę programowania dynamicznego jest algorytm Helda-Karpa. Złożoność czasowa tej metody jest znacznie mniejsza od przeglądu zupełnego – wynosi $O(n^2 2^n)$. Opiera się on przede wszystkim na funkcji określającej koszt ścieżki według następującego wzoru:

jeżeli set = 1, to cost(set, p) = wadze krawędzi określonej w macierzy jako d(0, p)

jeżeli set > 1, to cost(set, p) = min_x(cost(set-{v}, x) + d(x, v))

gdzie set jest zbiorem wierzchołków, a v punktem kończącym ścieżkę. Stosując ten algorytm, można rozważyć przykład dla czterech miast:

$cost_{1,2} = cost_{2,1} = 30$

$cost_{1,3} = cost_{3,1} = 36$

$cost_{1,4} = cost_{4,1} = 40$

$cost_{2,3} = cost_{3,2} = 20$

$cost_{2,4} = cost_{4,2} = 50$

$cost_{3,4} = cost_{4,3} = 67$

Startując od wierzchołka 1, w kolejnych iteracjach rozważamy zbiory:

- 1-elementowe:

  cost({2}, 2) = $d_{1,2}$ = 30

  cost({3}, 3) = $d_{1,3}$ = 36

  cost({4}, 4) = $d_{1,4}$ = 40

- 2-elementowe:

  cost ({2, 3}, 2) = min(cost ({3}, 3) + $d_{3,2}$) = min(36 + 20) = min(56) = 56

  cost ({2, 3}, 3) = min(cost ({2}, 2) + $d_{2,3}$) = min(30 + 20) = min(50) = 50

  cost ({2, 4}, 2) = min(cost ({4}, 4) + $d_{4,2}$) = min(40 + 50) = min(90) = 90

  cost ({2, 4}, 4) = min(cost ({2}, 2) + $d_{2,4}$) = min(30 + 50) = min(80) = 80

  cost ({3, 4}, 3) = min(cost ({4}, 4) + $d_{4,3}$) = min(40 + 67) = min(80) = 107

  cost ({3, 4}, 4) = min(cost ({3}, 3) + $d_{3,4}$) = min(36 + 67) = min(80) = 103

- 3-elementowe

  cost ({2, 3, 4}, 2) = min(cost ({3, 4}, 3) + $d_{3,2}$, cost ({3, 4}, 4) + d4,2) = min(107 + 20, 103 + 50) = min(127, 153) = 127

  cost ({2, 3, 4}, 3) = min(cost ({2, 4}, 2) + $d_{2,3,}$ cost ({2, 4}, 4) + d4,3) = min(90 + 20, 80 + 67) = min(110, 147) = 110

  cost ({2, 3, 4}, 4) = min(cost ({2, 3}, 2) + $d_{2,4}$, cost ({2, 3}, 3) + d3,4) = min(56 + 50, 50 + 67) = min(106, 117) = 106

- Zbiory 3-elementowe są w tym przypadku największymi możliwymi zbiorami (ponieważ rozważane są cztery wierzchołki, a jeden z nich jest wierzchołkiem startowym). W ten sposób można wyznaczyć wzór końcowy:

  min(cost ({2, 3, 4}, 2) + $d_{2,1}$, cost ({2, 3, 4}, 3) + $d_{3,1}$, cost ({2, 3, 4}, 4) + $d_{4,1,}$) = min(127 + 30, 110 + 36, 106 + 40) = min(157, 146, 146) = 146

W ten sposób możemy uzyskać wartość liczbową, reprezentującą minimalne wagi krawędzi, które tworzą w grafie cykl Hamiltona. By uzyskać kolejność wierzchołków tworzących najkrótszą ścieżkę, należy przejść po kolejnych iteracjach algorytmu.

Implementacja algorytmów i dane testowe

Projekt został napisany z użyciem języka C++. Pomiar czasu dokonywany był za pomocą biblioteki std::chrono i klasy high_resolution_clock, mierzącej czas z dokładnością do mikrosekund. Dane wczytywane były z plików za pomocą klasy Reader i przechowywane odpowiednio w zmiennej (ilość wierzchołków/miast), tablicy jedno- (kolejne wierzchołki) i dwuwymiarowej (macierz sąsiedztwa z uwzględnieniem odległości miedzy punktami). Na wczytanych danych operowały funkcje bruteForce() i dynamicProgramming(), rezultat zwracany był jako struktura Path, przechowująca koszt i najkrótszą ścieżkę w grafie.

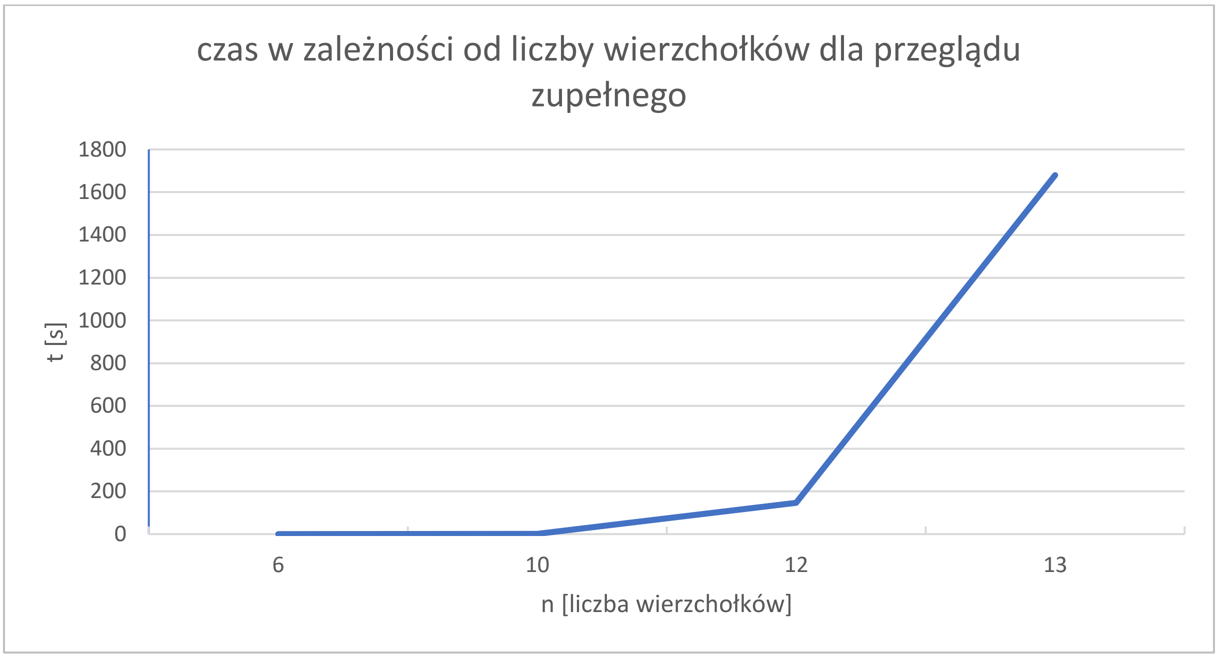
Pliki użyte do testowania algorytmów pochodzą ze strony dr. Jarosława Mierzwy i strony uniwersytetu w Heidelbergu. Każdy z plików posiadał jednakową strukturę: w pierwszej linii liczbę wierzchołków grafu, następnie przedstawiona jest macierz sąsiedztwa, w której i-ty wiersz odpowiada wierzchołkowi początkowemu, a j-ta kolumna wierzchołkowi docelowemu.

Ze względu na czas wykonania, dla algorytmu przeglądu zupełnego wykorzystano instancje 6, 10, 12 i 13 elementowe. Dla programowania dynamicznego dodatkowo testowane były instancje dla n = 14, 15, 17. Z biblioteki TSPLIB wykorzystany był tylko plik dla 17 wierzchołków. Algorytmy dla większych instancji trwały za długo (dla przeglądu zupełnego) lub brakowało pamięci operacyjnej. Dla każdej instancji problemu wykonano 10 pomiarów, a wyniki w tabelach zostały uśrednione.

Wyniki pomiarów

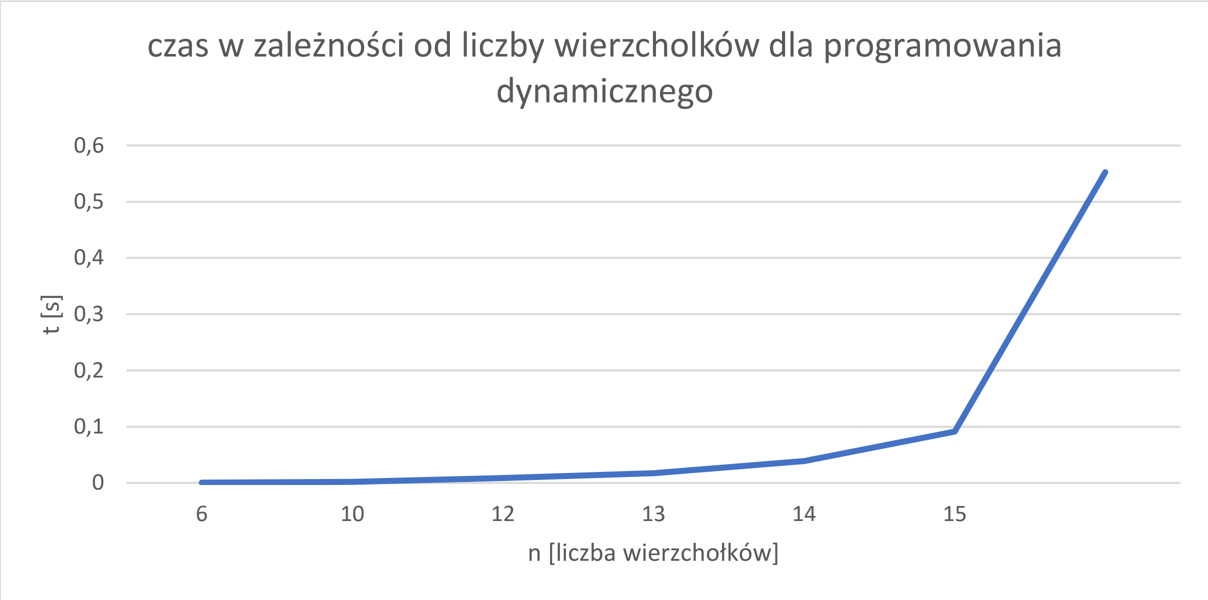
- Przegląd zupełny

instancja	wierzchołki	rozwiązanie	uzyskana wartość	t [µs]	t [s]
tsp_6_2.txt	6	80	80	201	0,000201
tsp_10.txt	10	212	212	1001642	1,001642
tsp_12.txt	12	264	264	146240367	146,240367
tsp_13.txt	13	269	269	1679780099	1679,780099



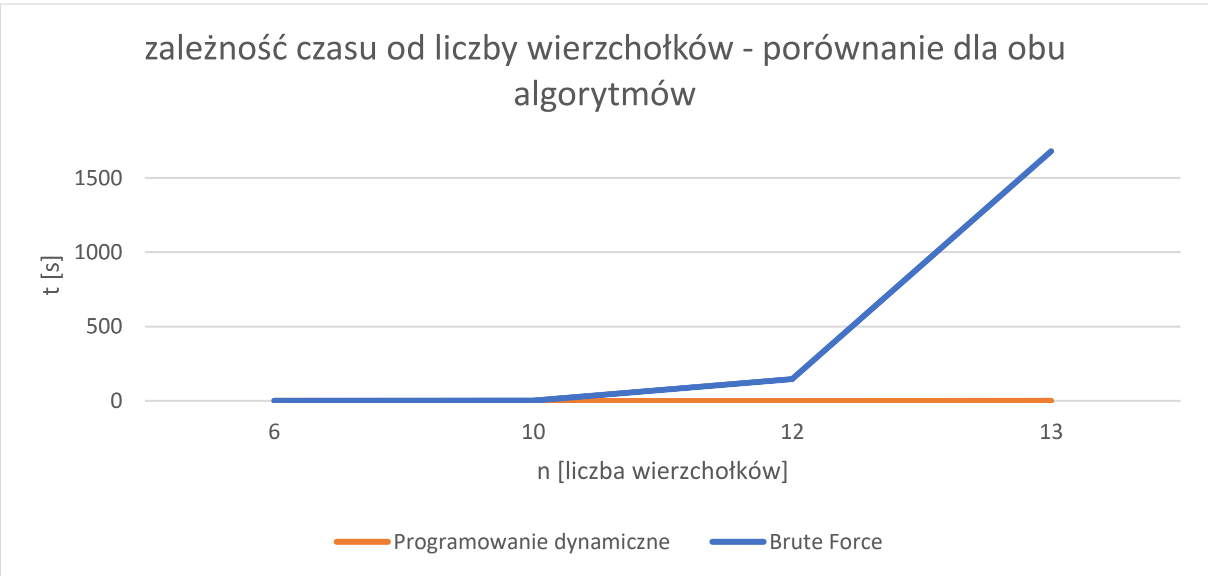
- Programowanie dynamiczne

instancja	wierzchołki	rozwiązanie	uzyskana wartość	t [µs]	t[s]
tsp_6_2.txt	6	80	80	468	0,000468
tsp_10.txt	10	212	212	1753	0,001753
tsp_12.txt	12	264	264	8394	0,008394
tsp_13.txt	13	269	269	17326	0,017326
tsp_14.txt	14	282	282	38709	0,038709
tsp_15.txt	15	291	291	90961	0,090961
gr17.txt	17	2085	2085	552479	0,552479



- Porównanie obu sposobów

wierzchołki	czas wykonania: t[s]		stosunek procentowy między czasami wykonania algorytmów
	przegląd zupełny	programowanie dynamiczne	
6	0,000201	0,000468	42,95%
10	1,001642	0,001753	57138,73%
12	146,240367	0,008394	1742201,18%
13	1679,780099	0,017326	9695140,82%



Wnioski

Dla dużych zbiorów danych przegląd zupełny jest całkowicie niewydajnym algorytmem, ze względu na jego złożoność czasową i obliczeniową. Wykorzystanie programowania dynamicznego wykazuje się krótszym czasem wykonania, rośnie znacznie wolniej w stosunku do ilości wczytanych danych. Z tego powodu też przegląd zupełny nie nadaje się do użytku przy dużych grafach.

Przy implementacji programowania dynamicznego przydaje się zastosowanie masek bitowych – reprezentacji zbioru w postaci ciągu bitów. Na pierwszy rzut oka może się ten sposób wydawać trudny, lecz znacznie ułatwia on operowanie na zbiorach przy mniejszej ilości wierzchołków (dla mojego rozwiązania – maksymalnie 32 wierzchołki).

Źródła

- http://algorytmy.ency.pl/artykul/problem_komiwojazera
- http://algorytmy.ency.pl/artykul/algorytm_helda_karpa
- https://eduinf.waw.pl/inf/alg/001_search/0140.php
- https://www.purepc.pl/technologia/ameby_w_biokomputerze_rozwiazaly_problem_komiwojazera
- http://informatyka.wroc.pl/node/227?page=0,1