Projektowanie Efektywnych Algorytmów

Projekt 1: Rozwiązanie problemu komiwojażera przy pomocy programowania dynamicznego i przeglądu zupełnego

Igor Kurek
226004

Spis treści

1 Wstęp

Celem projektu było wykonanie programu, wykorzystującego algorytmy programowania dynamicznego oraz przeglądu zupełnego do rozwiązania asynchronicznego problemu komiwojażera (ang. Asynchronous Travelling Salesman Problem).

2 Specyfikacja techniczna

- Program wykonany został w wieloparadygmatowym języku programowania Rust
- Struktury przechowujące dane alokowane są dynamicznie, zależnie od rozmiaru problemu
- Program posiada możliwość wczytywania danych z pliku, w celu weryfikacji poprawności
- Algorytmy zostały zaimplementowane zgodnie z paradygmatami programowania obiektowego
- Czas wykonania algorytmu mierzony był z dokładnością do nanosekund, przy wykorzystaniu bibliotek systemowych

3 Analiza problemu

Asynchroniczny problem komiwojażera należy do klasy problemów NP-trudnych. Jest to optymalizacyjny problem, rozwiązaniem którego jest znalezienie minimalnego cyklu Hamiltona (ścieżki prowadzącej przez wszystkie wierzchołki grafu, powracając na końcu do wierzchołka początkowego) w grafie pełnym ważonym. W wersji asynchronicznej, odległości pomiędzy wierzchołkami mogą dodatkowo zależeć także od kierunku przejścia pomiędzy nimi. Główną trudnością w rozwiązaniu problemu jest znacząca liczba możliwych kombinacji.

4 Opis algorytmów

4.1 Przegląd zupełny

Algorytm przeglądu zupełnego (ang. brute force) polega na przeanalizowaniu wszystkich możliwych przypadków, oraz wybraniu tego o najlepszej wartości. Zaletą tego algorytmu jest pewność, że otrzymany wynik jest najlepszym rozwiązaniem problemu. Poważną jego wadą jest jednak złożoność czasową wynoszącą $O(n!)$, co w praktyce czyni ten algorytm bezużytecznym dla większych zbiorów danych.

4.2 Programowanie dynamiczne

Programowanie dynamiczne (ang. dynamic programming) jest 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 $O(n^2 * 2^n)$. Jego wykonanie prezentuje się następująco:

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) = $\min_x$ (koszt(S-{p}, x) + d(x, p))
2. 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)
3. 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))
4. 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))
5. 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.

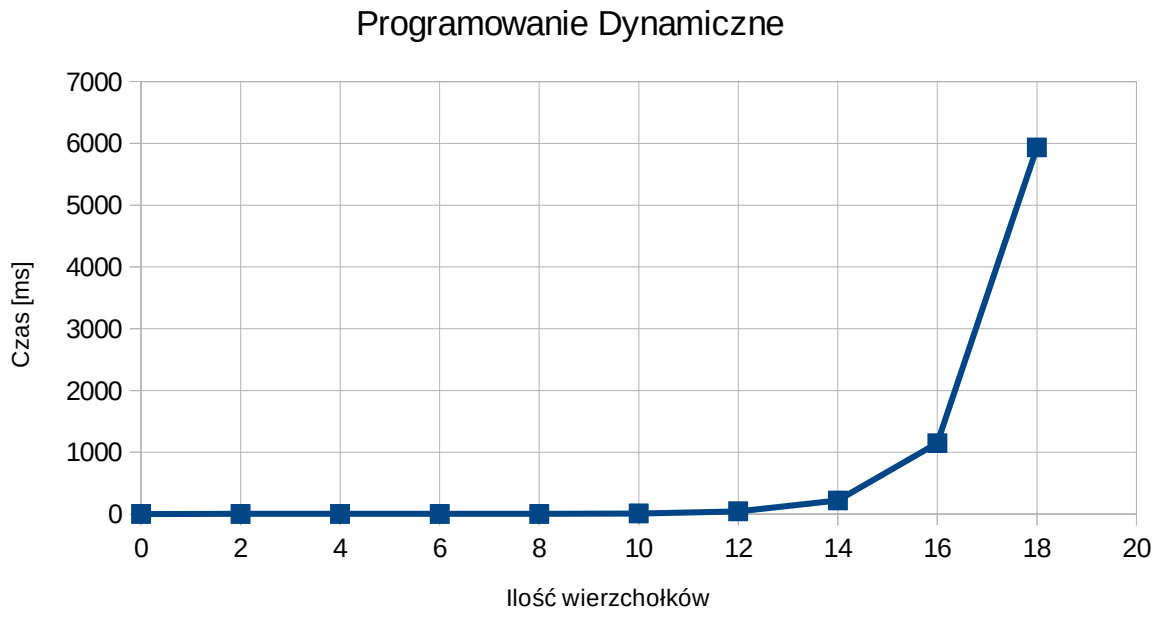
5 Implementacja algorytmów

Algorytmy operują na zmiennych integer 32bit. Pomiar czasu wykonany jest przy pomocy biblioteki języka Rust, mierzącej czas z dokładnością do nanosekund, przy wykorzystaniu pomiaru ilości ticków procesora. Do przechowywania czasów używana jest zmienna typu unsigned integer 64bit, co pozwala zachować dużą dokładność pomiaru. Algorytmy zaimplementowane są w postaci modułów (klas) języka Rust, co zapewnia ich poprawne działanie w każdym środowisku uruchomieniowym i na każdym systemie operacyjnym.

6 Wyniki pomiarów

6.1 Programowanie dynamiczne

Wierzchołki	Czas [ns]	Czas [ms]
0	0	0
2	61883	0,061883
4	105111	0,105111
6	557896	0,557896
8	2052105	2,052105
10	7747087	7,747087
12	41633170	41,63317
14	218679040	218,67904
16	1147388997	1147,388997
18	5934682443	5934,682443

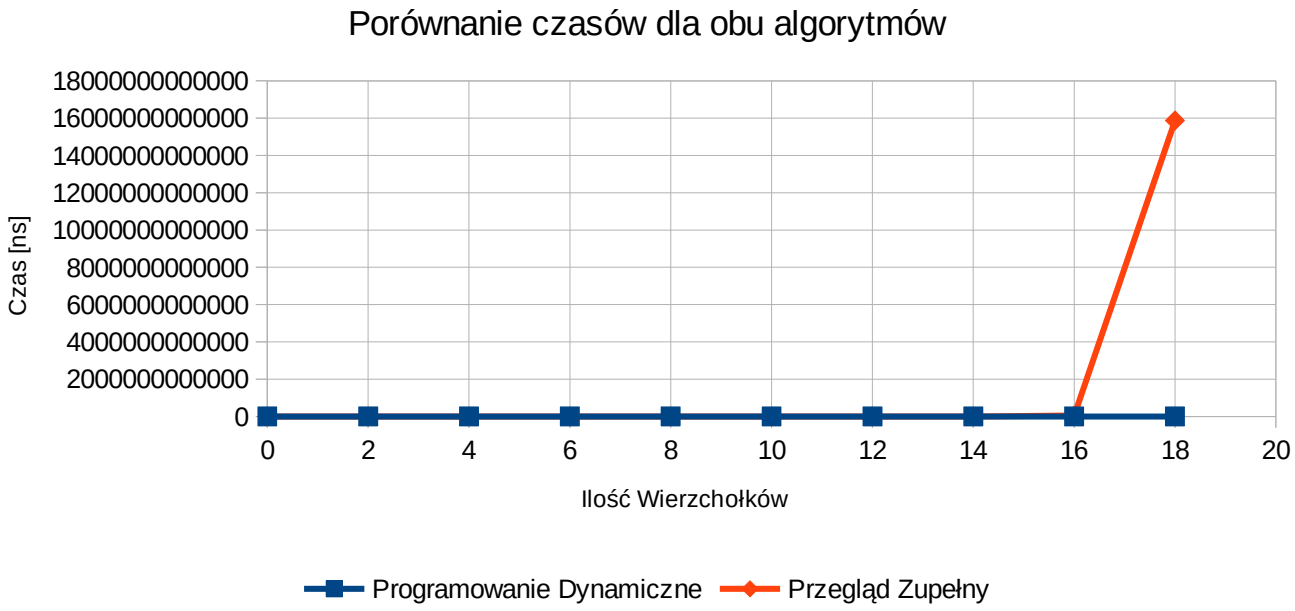


6.2 Przegląd Zupełny

Wierzchołki	Czas [ns]	Czas [ms]
0	0	0
2	39023	0,039023
4	137432	0,137432
6	3237122	3,237122
8	91907169	91,907169
10	8999635017	8999,635017
12	1187951822244	1187951,822244
14	216207231648408	216207231,648408
16	5,18897355956179E+016	51889735595,6179
18	1,58782590922591E+019	15878259092259,1

6.3 Porównanie algorytmów

Wierzchołki	Różnica pomiędzy przeglądem zupełnym a programowaniem dynamicznym [ms]
0	0
2	-0,02286
4	-0,032321
6	2,679226
8	89,855064
10	8991,88793
12	1187910,189074
14	216207012,969368
16	51889734448,2289
18	15878259086324,4



7 Wnioski

Dla dużych zbiorów danych, algorytm przeglądu zupełnego jest całkowicie niewydajny, ze względu na jego złożoność obliczeniową. Algorytm wykorzystujący metodę programowania dynamicznego wykazuje się znacznie szybszym czasem wykonania, rosnącym znacząco wolniej wraz ze wzrostem zbioru danych. Nie zapewnia on jednak stuprocentowej poprawności wyniku, w przeciwieństwie do algorytmów typu brute force.