Projektowanie Efektywnych Algorytmów

Zadanie projektowe nr 1

Implementacja i analiza efektywności algorytmu przeszukania zupełnego oraz programowania dynamicznego.

Piątek, 11:15

Bartosz Szymczak
252 734

prowadzący
Dr inż. Jarosław Mierzwa

Informatyka Techniczna
Wydział Informatyki i Telekomunikacji

19 listopada 2021

Spis treści

1 Wstęp teoretyczny

Pierwsze zadanie projektowe polegało na zaimplementowaniu oraz dokonaniu analizy efektywności algorytmu przeglądu zupełnego oraz programowania dynamicznego dla problemu komiwojażera.

1.1 Podstawowe założenia

- Używane struktury powinny być alokowane dynamicznie,
- Implementacje algorytmów należy dokonać zgodnie z obiektowym paradygmatem programowania,
- Program napisano w języku C++, który kompiluje się do kodu natywnego,
- Skorzystano ze środowiska Microsoft Visual Studio oraz biblioteki STL,

1.2 Opis problemu

Rozpatrywany problem opiera się na idei komiwojażera, który musi odwiedzić pewną ilość miast. Podróż rozpoczyna od miasta początkowego, po czym powinien przejść przez każde miasto dokładnie jeden raz, po czym wraca do początkowego miasta. Należy wybrać ścieżkę, która spełnia powyższe założenia oraz jest najkrótsza ze wszystkich możliwych.

Przekładając powyższy problem na teorię grafów, odpowiednio miasta to wierzchołki, drogi pomiędzy danymi miastami to krawędzie. Waga krawędzi może odpowiadać odległości pomiędzy danymi miastami, czasu przebycia drogi lub kosztowi podróży.

1.3 Opis algorytmów

1.3.1 Metoda przeglądu zupełnego

Metoda przeglądu zupełnego polega na wyznaczeniu wszystkich możliwych rozwiązań danego problemu. Dalej wyznaczane są dla nich wartości funkcji celu oraz wybierane jest rozwiązanie o ekstremalnej wartości funkcji celu, odpowiednio najniższej lub najwyższej. Zaletą metody jest stosunkowa łatwość implementacji, wadą jest wysoka złożoność obliczeniowa.

W kontekście problemu komiwojażera, użycie tej metody sprowadza się do sprawdzenia wszystkich możliwych dróg, zgodnych z założeniami oraz wyznaczenia tej, której sumaryczna wartość ścieżek jest najmniejsza.

1.3.2 Metoda programowania dynamicznego

Metoda programowania dynamicznego polega na podziale rozwiązywanego problemu na podproblemy względem danych parametrów.

W kontekście problemu komiwojażera, danymi wejściowymi będzie podzbiór wymaganych miast do odwiedzenia, odległość między miastami oraz miasto początkowe. Początkowo wszystkie miasta są nieodwiedzone oraz rozpoczynamy od miasta początkowego. Dalej oblicza się wartość najkrótszej ścieżki komiwojażera rekurencyjnie, zgodnie z założeniami problemu. Algorytm zapisuje odpowiednie trasy pomiędzy miastami i ich wartości, dzięki czemu nie ma potrzeby wyznaczania ich za każdym razem. Ostatecznie algorytm zwraca najkrótszą ścieżkę oraz jej wartość.

1.4 Złożoność obliczeniowa

Złożoność obliczeniowa to ilość zasobów komputerowych, potrzebnych do jego wykonania. Skupimy się na złożoności czasowej, czyli na ilości czasu potrzebnego do wykonania zadania, która zostanie wyrażona jako funkcja ilości danych.

1.4.1 Złożoność przeglądu zupełnego

Algorytm przeglądu zupełnego ma dokładnie n! permutacji, które należy wyznaczyć. Z tego powodu złożoność czasowa wynosi O(n!).

1.4.2 Złożoność programowania dynamicznego

Algorytm ma złożoność, oszacowaną na około $O(n^2 2^n)$.

2 Implementacja poszczególnych metod

Zgodnie z założeniami projektu, opis krok po kroku zostanie sformułowany dla jednej z metod. Implementacja każdej z metod zostanie dodatkowo opisana.

2.1 Przegląd zupełny

Za metodę przeglądu zupełnego odpowiedzialna jest klasa BruteForce. Jej istotnymi zmiennymi są:

- int cost - koszt najkrótszej ścieżki,
- int size - ilość miast,
- vector<int> path - wektor przechowujący indeksy miast, wchodzących w skład najkrótszej ścieżki,
- vector < vector <int> > temp - macierz przechowująca tymczasowo przekazaną macierz połączeń między miastami,

Klasa składa się z przeciążonego konstruktora oraz 3 dodatkowych metod:

- Konstruktor definiuje zmienne, przekazywany argument to macierz miast i tras pomiędzy nimi, nad którą będą przeprowadzane obliczenia. Wartość kosztu ścieżki początkowo ustawiona na maksymalny int w celu późniejszych porównań. Tymczasowa macierz jest równa przekazanej w argumencie. Zmienna size przyjmuje wartość ilości miast zgodnie z otrzymaną macierzą. Zmienna path przechowuje najkrótszą ścieżkę, początkowo jest wektorem o ilości elementów równej ilości miast. Wartości wektora to początkowo zera.
- Metoda doBruteForce() służy do wykonania algorytmu przeszukania zupełnego na danych przekazanych w konstruktorze. Korzystając z biblioteki STL oraz funkcji next_permutation() wytwarzane są wszystkie możliwe ścieżki oraz ich koszty. Ostatecznie metoda przekazuje do zmiennych cost i path rozwiązanie problemu.
- Metoda pomocnicza countCost(vector<int> path) oblicza i zwraca koszt ścieżki przekazanej w argumencie. Polega na obliczeniu sumy złożonej z kolejnych wartości macierzy, o indeksach zawartych w przekazanej ścieżce path. Na koniec dodawana jest ścieżka z ostatniego miasta do miasta początkowego.
- Metoda showPath() służy do wyświetlenia wyznaczonej ścieżki,

2.2 Programowanie dynamiczne

Za metodę programowania dynamicznego odpowiedzialna jest klasa DynamicProgramming. Jej istotnymi zmiennymi są:

- int size - ilość miast,
- int cost - koszt najkrótszej ścieżki,
- int bits - bity odpowiadające ilości miast,
- vector<int> path - wektor przechowujący indeksy miast, wchodzących w skład najkrótszej ścieżki,
- vector < vector <int> > temp - macierz przechowująca tymczasowo przekazaną macierz połączeń między miastami,
- vector < vector <int> > subProblems - macierz przechowująca długości fragmentów tras, przykładowo subProblems[0][6] oznacza, że w danej komórce znajduje się najniższy koszt drogi z miasta 0 przez miasta 1 i 2 (ponieważ $6 = 110_{(2)}$) i z powrotem do 0.
- vector < vector <int> > track - macierz przechowująca ścieżki, jakimi należy poruszać się przez fragmenty tras, przykładowo track[0][6] oznacza, że w danej komórce znajduje się indeks miasta, do którego należy się udać, aby podążać najmniejszym kosztem z miasta 0 przez miasta 1 i 2.

Klasa składa się z przeciążonego konstruktora oraz 4 dodatkowych metod:

- Konstruktor przypisuje wartości do zmiennych, przekazywany argument to macierz miast i tras pomiędzy nimi, nad którymi będą przeprowadzane obliczenia. Tymczasowa macierz jest równa przekazanej w argumencie. Zmienna size przyjmuje wartość ilości miast zgodnie z otrzymaną macierzą a zmienna bits przechowuje bity odpowiadające ilości miast. Macierze subProblems i track mają rozmiar size na bits i są wypełniane początkowo wartościami -1.
- Metoda dpAlghoritm(int firstTownIndex, int townsBits) zwraca wartość rekurencyjnie obliczonej najkrótszej ścieżki od firstTownIndex przez townsBits. Początkowo sprawdzane jest czy dany podproblem został już rozwiązany, co ewentualnie pozwala na pobranie jego wartości. Dalej poprzez maskowanie wyznaczane są kolejne miasta zadane przez argument townsBits. Potem obliczana jest rekurencyjnie wartość danej ścieżki oraz przypisywane są ewentualne rozwiązania podproblemów.
- Metoda countPath(int firstTown, int townsBits) służy do określenia najkrótszej ścieżki od firstTownIndex, przez townsBits. Do wektora ścieżki dodawane są kolejne miasta, korzystając z macierzy track oraz z maskowania następnych miast.
- Metoda doDynamicProgramming() służy do wykonania algorytmu programowania dynamicznego. Początkowo zmienna bits ustawiana jest w celu otrzymania "1" na miejscach odpowiadającym indeksom miast. Następnie podawane są drogi z każdego miasta do miasta początkowego. Dalej wyznaczany jest koszt najkrótszej ścieżki używając funkcji dpAlghoritm() oraz ścieżka używając metody countPath().
- Metoda showPath() służy do wyświetlenia wyników algorytmu.

Przykład praktyczny:

	0	1	2
0	-1	2	5
1	3	-1	4
2	1	7	-1

Tabela 1: Przykładowa macierz możliwych tras

bits = $1000_{(2)}$,
ustawienie -1 na zmiennych pomocniczych,
bits = $0111_{(2)}$,
podanie do subProblems dróg z każdego miasta do początkowego,
subProblems[0][0] = -1, subProblems[1][0] = 3, subProblems[2][0] = 1,
Wywołanie głównej funkcji, w celu wyznaczenia najkrótszej możliwej ścieżki dla problemu komiwojażera, z miasta 0 przez wierzchoki 1 i 2,
cost = dpAlghoritm (0, $0110_{(2)}$) ,
firstTownIndex = 0, townsBits = $0110_{(2)}$,
result = -1,
subProblems[0][$0110_{(2)}$] == -1, więc podproblem nie został jeszcze rozwiązany,
i = 0,
mask = $0110_{(2)}$,
masked = $0110_{(2)}$ & $0110_{(2)}$ == $0110_{(2)}$,
masked == townsBits, zatem iteraca skończona,
i = 1,
mask = $0101_{(2)}$,
masked = $0110_{(2)}$ & $0101_{(2)}$ = $0100_{(2)}$,
masked != townsBits, zatem wyznaczono właściwe miasto,
value = temp[0][1] + dpAlghroitm(1,$0100_{(2)}$) = 2 + 5 = 7,
Powyższa rekurencja:
subProblems[0][$0100_{(2)}$] == -1, więc podproblem nie został rozwiązany,
dla i = 0 i 1 nie otrzymamy żadnego wyniku,
i = 2,
mask = $0011_{(2)}$,
masked = $0100_{(2)}$ & $0011_{(2)}$ = $0000_{(2)}$,
masked != townsBits,
value = temp [1][2] + dpAlghoritm(2,0) = 4 + 1 = 5,
Wykonanie powyższej rekurencji:
subProbelms[2][0] = 1,
return 1,
result == -1,
result = value = 5,
track[1][$0100_{(2)}$]=2,
koniec pętli for,
subProblems[1][$0100_{(2)}$] = result = 5,
return 5,
result == -1, więc:
result = 7,
track[0][$0110_{(2)}$] = i = 1,

i=2,
mask = $0011_{(2)}$,
masked = $0110_{(2)}$ & $0011_{(2)}$ = $0010_{(2)}$
masked != townsBits, dlatego:
value = temp[0][2] + dpAlghoritm(2,$0010_{(2)}$) = 5 + 10 = 15,
Powyższa rekurencja:
subProblems[2][$0010_{(2)}$] == -1, zatem podproblem nie został rozwiązany,
wartość i = 1 dostarcza rozwiązanie,
mask = 0,
masked = $0010_{(2)}$ & $0000_{(2)}$ = 0
masked != townsBits,
value = temp[2][1] + dpAlghoritm(1,0) = 7+3 = 10
subProblems[1][$0000_{(2)}$] = 3,
return 3
result == -1, więc:
result = value = 10,
track[2][$0010_{(2)}$] = 2,
i = 2 nie przynosi rzowiązania
subProblems[2][0010] = result = 10,
return result = 10,
result == 7 != -1 && value == 15 ¿ result, zatem nie następuje zamiana,
subProblems[2][0010] = result = 7,
return 7,
Główna część algorytmu zakończona,
Najmniejszy koszt jest równy subProblems[0][0110] = 7,
track[0][$0110_{(2)}$] = 1,
track[1][$0100_{(2)}$] = 2,
Najkrótsza ścieżka to 0 -> 1 -> 2 -> 0.

	$0000_{(2)}$	$0001_{(2)}$	$0010_{(2)}$	$0011_{(2)}$	$0100_{(2)}$	$0101_{(2)}$	$0110_{(2)}$	$0111_{(2)}$
0	0	-1	-1	-1	-1	-1	7	-1
1	3	-1	-1	-1	5	-1	-1	-1
2	1	-1	10	-1	-1	-1	-1	-1

Tabela 2: Stan zmiennej subProblems pod koniec algorytmu

	$0000_{(2)}$	$0001_{(2)}$	$0010_{(2)}$	$0011_{(2)}$	$0100_{(2)}$	$0101_{(2)}$	$0110_{(2)}$	$0111_{(2)}$
0	-1	-1	-1	-1	-1	-1	1	-1
1	-1	-1	-1	-1	2	-1	-1	-1
2	-1	-1	2	-1	-1	-1	-1	-1

Tabela 3: Stan zmiennej track pod koniec algorytmu

3 Eksperymenty

3.1 Plan eksperymentu

Testom zostały poddane powyższe implementacje algorytmów. Do tego celu stworzona została klasa Tests. Do uruchomienia testów służyło menu główne.

Główne założenia:

- Pomiary wykonywane były dla 6, 9, 12, 15, 18, 21 i 24 miast w przypadku algorytmu programowania dynamicznego. Dla metody przeglądu zupełnego były to 2, 4, 6, 8, 10, 12 i 14 miast.
- W trakcie testów koszty połączeń były losowane z zakresu od 0 do 99, za pomocą funkcji rand().
- Pomiary czasów wykonano przy użyciu funkcji QueryPerformanceCounter(),
- Testy odbywały się używając trybu RELEASE środowiska Microsoft Visual Studio.
- W celu uśrednienia wyników, każdy pomiar powtórzono 100 razy, a następnie uśredniono wyniki.

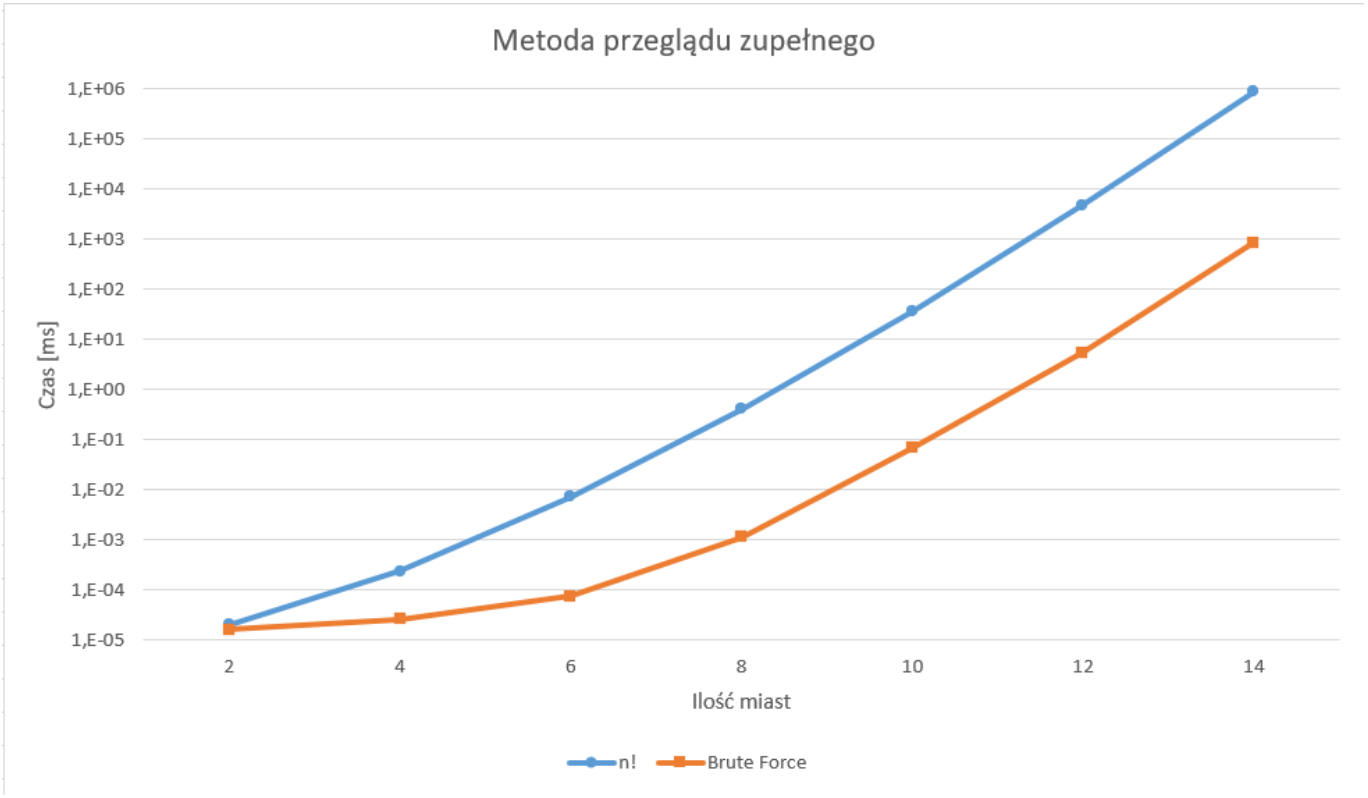
3.2 Wyniki

Na wykresach zastosowano skalę logarytmiczną w celu zwiększenia czytelności.

3.2.1 Przegląd zupełny

n	t [ms]	σ [ms]
2	0,002	0
4	0,003	0
6	0,007	0,001
8	0,115	0,006
10	6,883	1,107
12	541,832	38,658
14	84054,268	5504,611

Tabela 4: Dane dla algorytmu przeglądu zupełnego

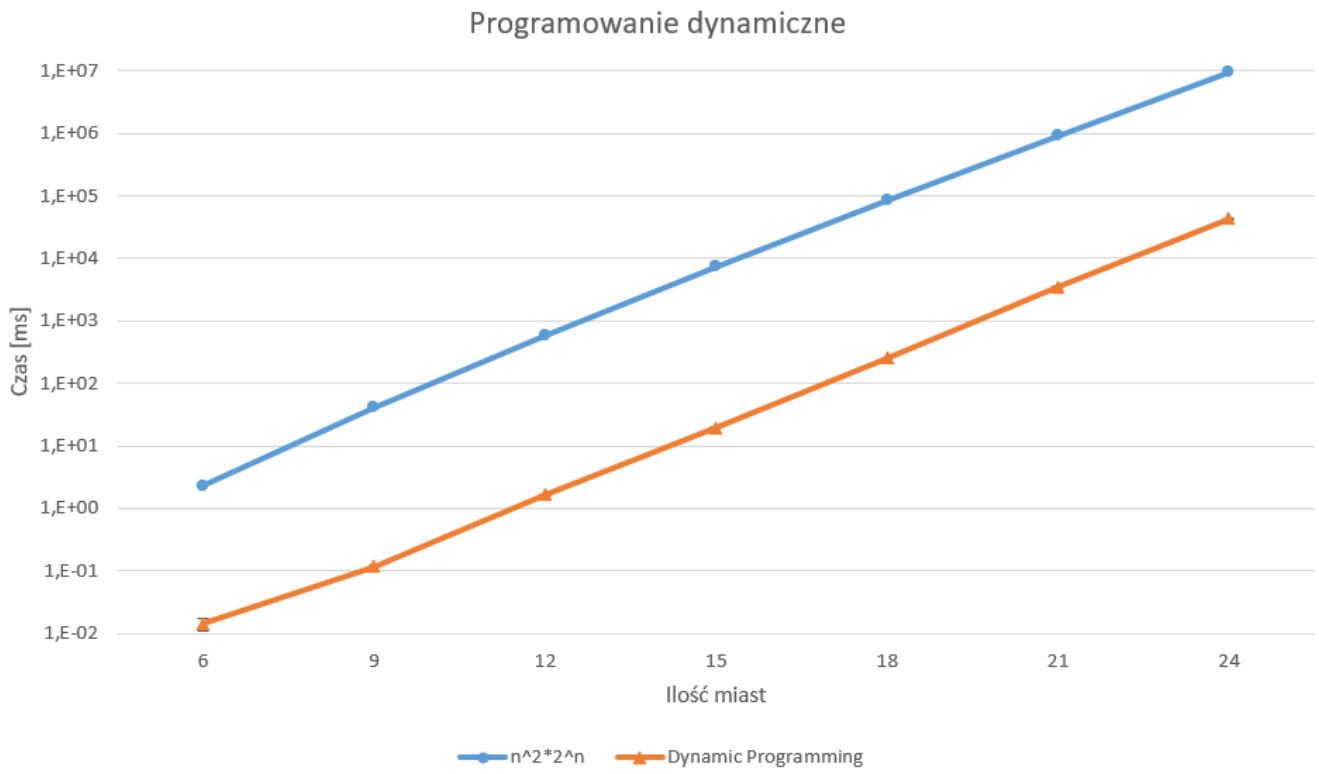


Rysunek 1: Wykres dla algorytmu Brute Force wraz z uwzględnieniem n!

3.2.2 Programowanie dynamiczne

n	t [ms]	σ [ms]
6	0,014	0,003
9	0,115	0,012
12	1,635	0,023
15	19,458	0,256
18	253,808	9,501
21	3513,47	25,569
24	43352,889	406,114

Tabela 5: Dane dla algorytmu programowania dynamicznego



Rysunek 2: Wykres dla algorytmu programowania dynamicznego wraz z uwzględnieniem $n^2 2^n$

4 Wnioski

Otrzymane podczas testów złożoności czasowe algorytmów pokrywają się z założeniami projektowymi. Zastosowana implementacja algorytmu przeglądu zupełnego pozwoliła na uzyskanie złożoności mniejszej od n!. Odchylenie standardowe jest małe, ponieważ za każdym razem wykonywana jest podobna ilość obliczeń. Szybszy okazał się algorytm programowania dynamicznego. Uwzględniając ilość zapisywanych danych, można dojść do wniosku że krótszy czas rozwiązania uzyskiwany jest kosztem wysokiego zużycia pamięci. Przegląd zupełny jest najprostszy do zaimplementowania, lecz najmniej skuteczny w działaniu dla dużej ilości danych. W zdecydowanej większości bardziej opłacalne jest wykorzystanie algorytmu programowania dynamicznego.

Literatura

[1] https://home.agh.edu.pl/~horzyk/lectures/pi/ahdydpiwykl8.html

[2] https://eduinf.waw.pl/inf/alg/001_search/0140.php

[3] https://scholarworks.calstate.edu/downloads/xg94hr81q?locale=it

[4] https://iq.opengenus.org/travelling-salesman-branch-bound/

[5] https://home.agh.edu.pl/~grzesik/Informatyka/informatyka_W7_bez_RSA.pdf

[6] https://www.youtube.com/watch?v=XaXsJJh-Q5Y