Jakub Superczyński 241381 13.11.2019

Projektowanie efektywnych algorytmów

Implementacja i analiza efektywności algorytmu programowania dynamicznego dla problemu komiwojażera.

Prowadzący zajęcia:

Dr inż. Dariusz Banasiak

1. Wstęp teoretyczny

Problem komiwojażera to zagadnienie optymalizacyjne z teorii grafów polegające na odnalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Dla łatwiejszego zobrazowania możemy przedstawić to w ten sposób: przedstawiciel handlowy ma listę pewnej liczby miast, które musi odwiedzić a następnie powrócić do miasta początkowego. Problem polega na tym, aby tak zaplanował swoją podróż aby przejazdy między miastami kosztowały go jak najmniej (np. aby odległości były jak najkrótsze lub jak najmniej zapłacił za paliwo). Co ważne: zakładamy, że graf może być asymetryczny, tzn. waga tej samej krawędzi (koszt tej samej trasy) może być różna w zależności w którą stronę chcemy przez nią przejść.

Najbardziej oczywistym rozwiązaniem problemu jest tzw. brute-force. Działa on w następujący sposób: generujemy wszystkie możliwe trasy i sprawdzamy ile one kosztują, a zapamiętujemy zawsze minimalny koszt. Metoda ta zawsze da optymalny wynik i jest bardzo prosta w implementacji, ale ma poważną wadę. Generowanie wszystkich możliwych ścieżek to obliczenie wszystkich możliwych permutacji, których jest aż *(n-1)!*. Oznacza to, że złożoność czasowa tego algorytmu wynosi **O(n!)**. Sprawia to, że algorytm staje się praktycznie bezużyteczny już dla kilkunastu wierzchołków.

Innym podejściem jest programowanie dynamiczne z wykorzystaniem masek bitowych. Polega na podziale całego problemu na mniejsze podproblemy. Uruchamiając algorytm dla całego problemu, rekurencyjnie zostaje on wywołany dla coraz to mniejszych zbiorów wierzchołków, dla których są liczone koszty. Następnie wyniki uzyskane dla mniejszych podproblemów są sumowane tak długo aż uzyskamy koszt przejścia całego grafu. Co ważne, algorytm nie liczy wszystkich możliwych rekurencyjnych podzbiorów, a jedynie te które przybliżą nas do rozwiązania, tzn. nie zostały wcześniej obliczone. Złożoność czasowa tego algorytmu wynosi $O(n^2 2^n)$ co sprawia, że jest szybszy niż brute-force.

2. Przykład praktyczny

Rozważmy przykład prostego grafu o 3 wierzchołkach. Maska oznaczająca odwiedzenie wszystkich wierzchołków wygląda następująco: 111
Pierwsze wywołanie funkcji jest dla argumentów (1, 0) co oznacza że ruszamy z pierwszego wierzchołka i odwiedziliśmy pierwszy wierzchołek. Iterujemy po wszystkich wierzchołkach. Pierwszy wierzchołek był odwiedzony więc nic nie robimy. Drugi wierzchołek nie był, więc wywołujemy tę funkcję rekurencyjnie dla argumentów (011, 1) oraz dodajemy wagę krawędzi oraz zapisujemy wynik w tablicy dp[3][1]. Ponownie iterujemy. Pierwszy i drugi był odwiedzony więc nic nie robimy. Trzeci nie był więc znów wywołujemy dla argumentów (111, 2). Maska pokazuje że odwiedziliśmy wszystkie wierzchołki, więc kończymy rekurencję pamiętając o dodaniu wagi krawędzi między ostatnim a pierwszym wierzchołkiem. Zapisujemy wynik w tablicy dp. Wracamy do iteracji w pierwszym wywołaniu funkcji i dalej postępujemy analogicznie.

3. Opis implementacji algorytmu

Graf zapamiętany jest w formie macierzy sąsiedztwa w dwuwymiarowej, alokowanej dynamicznie tablicy *graph*. Wartość oznaczone indeksem *[i][j]* oznacza wagę krawędzi prowadzącej od wierzchołka *i* do *j*, pod warunkiem że są to różne wierzchołki (zakładamy brak pętli). Dodatkowo klasa odpowiedzialna za przechowywanie grafu zawiera również parametr *size* w którym zapamiętana jest wielkość instancji problemu.

Brute-force

W tej metodzie są dodatkowe dwie tablice dynamiczne jednowymiarowe: *path* przechowująca aktualnie liczoną ścieżkę oraz *best_path* zapamiętująca optymalną ścieżkę. Klasa posiada prywatną funkcję *Factorial* służącą liczeniu silni. Algorytm na początku w tablicy *path* zapamiętuje ścieżkę pierwszą jako przejście po kolei przez graf przez wierzchołki *0->1->…->n*, jednocześnie traktując ją jako ścieżkę optymalną. Następnie liczy jej koszt, pamiętając o dodaniu kosztu przejścia z ostatniego wierzchołka do pierwszego. Następnie *(n-1)!* razy tworzy następną permutację korzystając z algorytmu Dijkstry, zapamiętuje ją w tablicy *path*, liczy jej koszt i sprawdza czy jest ona lepsza niż obecnie zapamiętana jako minimum. Jeśli tak: aktualizuje wartość *cost* oraz zawartość tablicy *best_path*

Programowanie dynamiczne

W tej metodzie znów potrzebne są dwie dwuwymiarowe tablice o rozmiarze $[2^n][n]$: *dp* służąca zapamiętywaniu obliczonych już rozwiązań oraz *path* służąca w dalszym ciągu wyłuskiwaniu samej ścieżki poprzez backtracking. Pierwsze wywołanie funkcji jest dla wierzchołka startowego (czyli indeks 0) oraz maski 1 (oznacza to, że odwiedzony został pierwszy wierzchołek: 00…01). Początkowo sprawdzamy czy dany przypadek nie został już obliczony. Następnie iterujemy po wszystkich wierzchołkach i jeśli dany wierzchołek nie został odwiedzony (nałożenie na siebie poprzez AND maski oraz ciągu zer z jedynką na pozycji aktualnie odwiedzanego miasta daje 0), wywołujemy rekurencyjnie funkcje oraz dodajemy do niej wagę krawędzi między wierzchołkiem odwiedzonym w ostatnim wywołaniu oraz w obecnym. Nowymi argumentami wywołania funkcji są: nowa maska utworzona poprzez połączenie poprzez OR starej maski i ciągu zer z jedynką wierzchołka odwiedzonego w obecnym wywołaniu (czyli zaznaczamy w nowej masce, że dany wierzchołek został odwiedzony) oraz indeks odwiedzonego wierzchołka. Rekurencja kończy się, gdy odwiedzimy wszystkie wierzchołki. Jeśli obliczone rozwiązanie danego stopnia jest lepsze od poprzedniego, zastępujemy je.

4. Plan eksperymentu

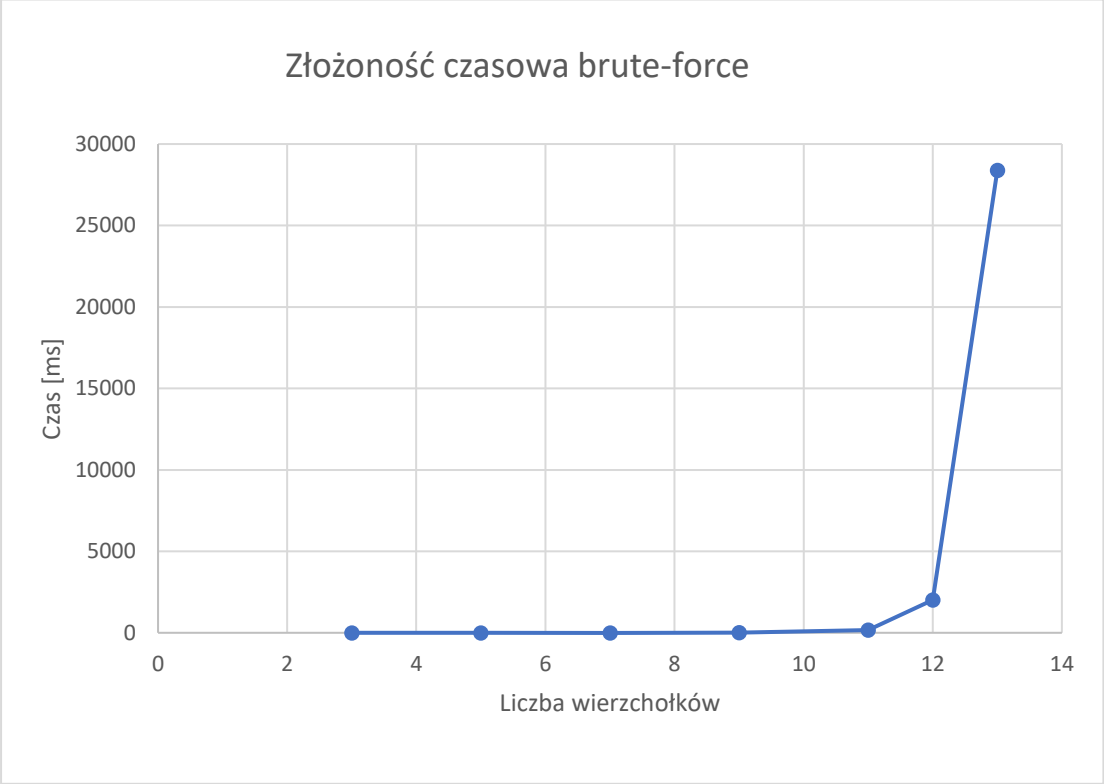
Eksperyment powinien być tak zaplanowany, aby wyniki były miarodajne, a jednocześnie aby nie trwał on zbyt długo oraz aby wystarczała pamięć. Dla każdej metody wybrałem 7 różnych rozmiarów problemu. Dla brute-force były to: 3, 5, 7, 9, 11, 12, 13. Powyżej tej wartości czasy były zbyt duże. Dla programowania dynamicznego były to: 5, 8, 11, 14, 17, 20, 23. Powyżej tej wartości program nie działał z powodu przekroczenia dostępnej pamięci. Dla każdego rozmiaru wykonałem 100 pomiarów, generując 100 różnych

instancji problemu. Do generowania danych wykorzystałem funkcję rand(). Do pomiaru czasu wykorzystałem funkcję QueryPerformanceCounter() oraz QueryPerformanceFrequency(). Poprawność działania algorytmu została przetestowana na różnych instancjach testowych znalezionych w Internecie. Niestety, bilbioteka TSPLIB zawierała niewiele małych instancji, które mogłem wykorzystać, dlatego korzystałem z innych źródeł. Algorytm dawał optymalne wyniki.

5. Wyniki eksperymentów

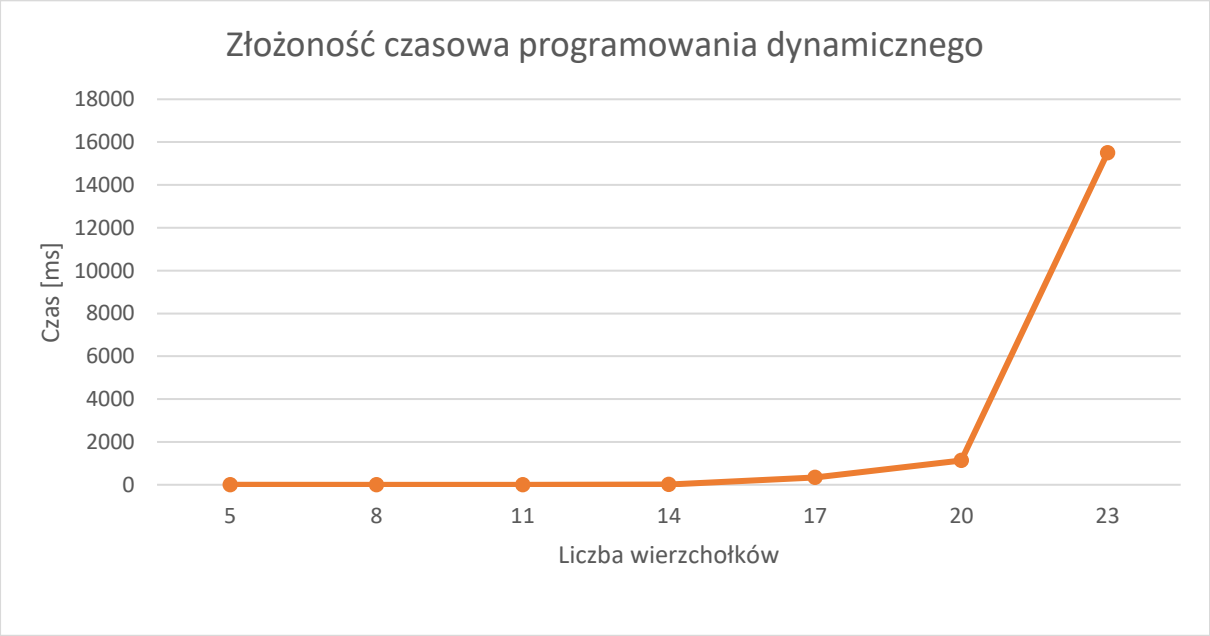
Brute-force

Liczba wierzchołków	Czas [ms]
3	0,000366
5	0,001245
7	0,022722
9	1,526163
11	165,5698
12	2011,64
13	28386,88



Programowanie dynamiczne

Liczba wierzchołków	Czas [ms]
5	0,002006
8	0,033629
11	0,395381
14	5,029135
17	341,1831
20	1136,073
23	15498,76



6. Wnioski

Metoda programowania dynamicznego działa dużo szybciej niż metoda naiwna. Większym problemem dla niej jest złożoność pamięciowa, która sprawia że szybko wyczerpuje się dostępna pamięć. Mimo to, programowanie dynamiczne może być dobrym sposobem na rozwiązanie problemu dla grafów niewielkich, choć większych niż te możliwe do rozwiązania metodą naiwną. Na podstawie wykresów złożoności czasowej możemy stwierdzić, że zgadzają się one ze złożonością teoretyczną.