Katarzyna Brzezińska 235879

Sprawozdanie z projektu z projektowania efektywnych algorytmów

1. Wstęp teoretyczny.

Celem projektu była implementacja algorytmu pozwalającego rozwiązać problem komiwojażera oraz oszacowanie jego efektywności.

Problem komiwojażera jest problemem NP – zupełnym. Mając pełny graf skierowany G=(V,E;w) ,gdzie |V| =n , a w(e) wyznacza wagę krawędzi e. Problem polega na znalezieniu cyklu Hamiltona o najmniejszej wadze w G.

Jedną z metod rozwiązania tego programu jest użycie algorytmu wykorzystującego przegląd zupełny, czyli innymi słowy sprawdzenie wszystkich możliwych tras i wybranie tej o najmniejszej wadze. Złożoność czasowa takiego algorytmu wynosi O(n!).

Kolejną metodą rozwiązania problemu jest wykorzystanie programowania dynamicznego do implementacji algorytmu. Programowanie dynamiczne jest metodą rozwiązywania zadań optymalizacyjnych, które na mocy pewnych własności, można sformułować jako

poszukiwanie ciągu decyzji.

Ja użyłam algorytmu Helda-Karpa (czasami określany jako algorytm Bellmana-Helda-Karpa) ma on złożoność czasową O(n^2 \*2^n). Jest to co prawda złożoność gorsza od wielomianowej, ale algorytm ten jest znacznie lepszy od algorytmu sprawdzającego wszystkie warianty (złożoność czasowa O(n!)).

1. Przykład praktyczny algorytmu wykorzysującego programowanie dynamiczne.

matrix(0,{a,b})-zaczynamy trasę od 0 i może przejść przez a do b lub przez b do a.

Dodaje wagę drogi powrotnej do wierzchołka początkowego w naszym przypadku jest nim 0

matrix ( 2, {3} ) =  (2,3) + matrix (3, 0 )     4+10=14

matrix ( 3, {2} ) =  (3,2) + matrix (2, 0 )     1+14=15

matrix ( 1, {3} ) =  (1,3) + matrix (3, 0 )     6+10=16

matrix ( 3, {1} ) =  (3,1) + matrix (1, 0 )     28+26 = 54

matrix ( 1, {2} ) =  (1,2) + matrix (2, 0 )     25+14 = 39

matrix ( 2, {1} ) =  (2,1) + matrix (1, 0 )     37+26=63

matrix (1, {2,3} )   = minimum of

=  { (1,2) + matrix (2, {3} )     25+1**4=**39

= { (1,3) + matrix {3, {2} )     6+**15=**21

matrix (2, {1,3} )   = minimum of

=  { (2,1) + matrix (1, {3} )     37+**16**=53

= { (2,3) + v {3, {1} )     4+**54**=58

v (3, {1,2} )   = minimum of

=  { (3,1) + matrix (1, {2} )     28+**39**=67

= { (3,2) + matrix {3, {1} )     1+**54**=55

matrix( 0, {1,2,3} ) = minimum of

= { (0,1) + matrix (1, {2,3} ) 33+21=54

= { 0,2) + matrix (2, {1,3} ) 6+53=59

= { (0,3) + matrix (3, {1,2} ) 15+55=70

Tras:0->1->3->2->0, waga: 54

1. Implementacja algorytmów.

W metodzie *dynamic()* korzystam z tablicy wektorów *matrix,* przechowującej macierz wag oraz tablicy wektorów *dp* ,która w zależności od wierzchołka i głębokości(co określa maska bitowa) drzewa przechowuje wagi danych gałęzi drzewa. Jest to funkcja rekurencyjna która wywołana zostaje w metodzie *d()* z parametrami *pos*=0,*visited*=1 jako, że zaczynamy od wierzchołka 0.

int K:: dynamic(int pos,int visited)

{

if (visited == ((1 << matrix.size()) - 1) //jeśli jesteśmy w liściu drzewa to

return matrix[pos][0]; //zwracamy wagę jego drogi do korzenia

if (dp[pos][visited] != INT\_MAX) //jeśli ta gałęź drzewa była już policzona to

return dp[pos][visited]; // nie liczymy jej znowu

for (int i = 0; i < matrix.size(); ++i)

{

if (i == pos || (visited & (1 << i))//jeśli odwiedziliśmy już ta gałąź

continue; // lub jesteśmy w tym wierzchołku => prawda=>continue

weight = matrix[pos][i] + dynamic(i, visited | (1 << i));

if (weight < dp[pos][visited]){

dp[pos][visited] = weight;

k[pos][visited] = i;//i to następnik

}

}

return dp[pos][visited];

}

W naszym algorytmie wykorzystujemy operacje na bitach np.

*visited & (1 << i),* załóżmy, że *visited*=1,a *i*=2

*1 << i* to inaczej *1\*=*

*visited & (1 << i) ,* gdzie & oznacza operacje AND na bitach, czyli w naszym wypadku jeśli *visited = 1 = 001, a ( 1 << i)= 4 = 100 to*

*001*

*AND 100*

*000*

visited | (1 << i) to 1+=1+4=5

Dla macierzy wag *matrix* z wcześniejszego podpunktu.

Maska bitowa

1

5

9

3

7

11

7

11

13

13

15

15

15

15

15

15

1. Plan eksperymentu.

Eksperyment polegał na pomiarze czasu wykonywania się algorytmów w zależności od rozmiaru struktury(ilości wierzchołków w grafie).

Zmienna *N* przechowuje ilość wierzchołków(miast).Macierzy wag jest przechowywana w tablicy wektorów *matrix*, a znaleziona droga w wektorze *h* ,jest też wektor *v* pomocniczy przechowujący aktualną drogę. Tablica wektorów *dp*, która jest zmienną pomocniczą wykorzystywaną w algorytmie korzystającym z programowania dynamicznego używającym maski bitowej .Do liczenia czasu w Windows używam funkcji:

BOOL QueryPerformanceCounter (

\_\_out LARGE\_INTEGER \*lpPerformanceCount

);

Która jest przechowywana w stworzonej przeze mnie klasie *Timer*.

Jeśli zadziała poprawnie zwrócona wartość będzie nie zerowa. Wyniki będą odpowiednio umieszczane w pliku ,,wyniki.txt”

1. Wyniki eksperymentu.

Wyniki działania algorytmu wykorzystującego programowanie dynamiczne.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar N | 6 | 8 | 10 | 12 | 14 | 16 | 17 |
| Średni czas | 11759,88 | 16416,42 | 23157,4 | 290176,3 | 1,22603 | 6,74786 | 1,66992 |

*Tabela1.*

*Rys1.*

Wyniki działania algorytmu wykorzystującego przegląd zupełny.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar N | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Średni czas | 7571,65 | 2,55838 | 85744,62 | 1,22167 | 5,90704 | 6,22088 | 6,79899 |

*Tabela2.*

*Rys2.*

*Rys3*.

1. Wnioski.

Z uzyskanych wyników możemy zauważyć jak bardzo czas wykonywania algorytmu dla takiej samej liczby wierzchołów rożni się w zależności od użytej metody .Znalezienie rozwiązania dla problemu komiwojażera korzystając z metody *brute force* zajmuje nieporównywalne więcej czasu niż gdybyśmy wykorzystali do tego programowanie dynamiczne, różnica ta zwiększa się wraz z liczba miast przez, które komiwojażer musi przejść. Wyniki eksperyment potwierdziły słuszność podanych wcześniej złożoności czasowych.