Projektowanie Efektywnych Algorytmów Laboratorium

ZADANIE NR 1

Temat: Implementacja i analiza efektywności metody programowania dynamicznego dla wybranego problemu optymalizacji

Autor:	Karolina Herlender
Nr indeksu:	199294
Rok i kierunek studiów:	3 rok, Informatyka
Termin projektu:	Pn 11:15
Prowadzący:	Dr inż. Łukasz Jeleń

1. Wstęp teoretyczny

1.1Opis problemu komiwojażera

Problem Komiwojażera(ang. *TSP –Travelling Salesman Problem*) polega na odwiedzeniu N miast tylko jeden raz i wrócić do miasta początkowego i wybrać przy tym trasę o najmniejszym koście. Drogi pomiędzy miastami mają przypisane wagi kosztów przebycia, tzw. czas podróży dla komiwojażera.

Problem Komiwojażera często przedstawia się w postaci grafu pełnego hamiltona. W problemie symetrycznym koszt przebycia drogi do miasta i z miasta jest jednakowy. W takcie laboratorium rozpatrywano przypadek niejednakowego rozkładu kosztów.

Podsumowując, rozwiązanie problemu komiwojażera sprowadza się do znalezienia takiej drogi, aby każdy z N wierzchołków występował na niej tylko raz sprowadza się do znalezienia w grafie tzw. cyklu Hamiltona. Problem Komiwojażera jest problemem NP-trudnym.

1.2 Rozwiązanie Problemu Komiwojażera

1.2.1 Przegląd Zupełny

Najprostszą metodą rozwiązania problemu komiwojażera jest zbadanie wszystkich istniejących cyklów hamiltona i dokonanie wyboru ścieżki o najmniejszej sumie wag.

W praktyce działanie algorytmu można przedstawić następująco:

Dany jest graf o N=9 wierzchołkach i należy policzyć ile znajduje się w nim cykli hamiltona.

Z pierwszego wierzchołka można wybrać 8 krawędzi, z drugiego 7, z trzeciego 6, …, z dziewiątego –1 krawędź. W efekcie liczba cykli Hamiltona jest równa:

H = 8 * 7 * 6 * 5 * … * 1 = 8! = 40 320

Zatem dla N wierzchołków liczba cykli wyniesie H = (N-1)!. Wynik ten należy do wykładniczej klasy złożoności obliczeniowej O(n!).

1.2.2 Algorytm Dynamiczny

Podstawowe podejście do programowania dynamicznego opiera się na powszechnym przekonaniu, że rozbicie problemu na grupę podproblemów ułatwia znalezienie rozwiązania. Jeżeli liczba podproblemów jest wielomianowa można obliczyć wartość każdego od najmniejszych do największych. W przypadku implementacji programista zapisuje wartości dla obliczonych rozwiązań

podproblemów w tablicy, aby uniknąć ponownego obliczania rozwiązań w trakcie rekurencyjnego wywołania i obliczać dany podproblem tylko raz. Korzystając z tablicy wartości podproblemów można łatwiej obliczyć rozwiązanie problemu głównego.

Sposób podziału algorytmu może zależeć od różnie zdefiniowanych parametrów: liczby elementów w problemie, określonej wartości liczbowej zmieniającej się w zakresie od 0 do największej stałej występującej w problemie, lub inne zdefiniowane parametry.

Rozwiązanie zadania dynamicznym programowaniem sprowadza się do znalezieniu równania rekurencyjnego opisującego optymalną wartość funkcji celu dla danego problemu jako funkcji optymalnych wartości funkcji celu dla podproblemów o mniejszych rozmiarach.

Algorytm programowania dynamicznego pozwala na rozwiązanie problemów NP-Trudnych, najczęściej jest określany jako algorytm o pseudowielomianowej złożoności obliczeniowej.

Przy wykorzystaniu programowania dynamicznego dla rozwiązania problemu komiwojażera zawierającego n jako liczbę miast złożoność szacuje się na $O(n^2 2^n)$. Dla przykładu, gdy algorytm zupełny sprawdza 10! kombinacji równe 3 628 800, podczas gdy przy programowaniu dynamicznym złożoność ogranicza się do: $10^2 2^{10} = 102\ 400$

2. Opis implementacji

Implementację algorytmu wykonano w języku programowania Python. Skorzystano z modułów Mapa.py, Dynamiczny.py, Tworz.py, Timer.py, gdzie najważniejszy skrypt to Dynamiczny.py zawierający realizację algorytmu, pozostałe moduły są pomocnicze dla uzyskania możliwości rzetelnego pomiaru czasu, zasymulowania w strukturach danych grafu, po którym przemieszcza się komiwojażer, tworzenie pliku z danymi wejściowymi.

Jako dane wejściowe program przyjmuj asymetryczną macierz kwadratową z kosztami przejścia do poszczególnych miast (z 100 jako przekątną). Przykładowy plik wygląda następująco:

```
100 5 20 4 7 58
14 100 8 5 9 52
16 3 100 14 19 80
7 20 17 100 6 73
18 8 13 13 100 6
9 10 11 19 10 100
```

Listing 1 - Plik testowy z danymi wejściowymi programu

Główny kod algorytmu opiera się na pętli, w której stopniowo dzieli się problem na coraz mniejsze wszystko w pętli while:

```
        while miasto:
            l, miasto = self.rekurencja(dokad, miasta)
            najlepsza_sciezka.append(miasto)
            self.najlepsza_droga.append(miasto)
            dokad = miasto
            miasta = miasta - set([miasto])
```

Listing 2 - Kod implementacji cz 1

Gdzie funkcja rekurencji jest zdefiniowana następująco:

```
  def rekurencja(self, dokad, tmp_miasta):
        """Przejscie po wszyskich miastach w zbiorze tmp_miasta i zebranie funkcji
optymalizuujacej"""
        if tmp_miasta:
            """funkcja min zwraca wartosc minimalna w tym przypadku iterujac po
aktualym zbiorze"""
            return min((self.mapa.miasta[miasto][dokad] + self.rekurencja(miasto,
tmp_miasta - set([miasto]))[0], miasto)
                       for miasto in tmp_miasta)
        else:
            return (self.mapa.miasta[0][dokad], 0)
```

Listing 3 - Kod implementacji cz. 2

Funkcja rekurencji zawiera konstrukcję min(x1,x2) wyznaczającą argument z mniejszą wartością. Wykonuje się rekurencyjnie, na każdym kroku wykonując badanie typowe dla algorytmów programowania dynamicznego: wyciąga mniejszą z wartości. Dopóki zbiór miast jest niepusty dla wszystkich miast zwracana jest mniejsza wartość. Funkcja zwraca krotkę (pythonowa konstrukcja, rodzaj listy, w tym przypadku para zwracanych wartości). Po warunku else nastąpi przerwanie wykonywania nadrzędnej pętli while po podaniu drugiego elementu krotki jako 0.

W kodzie reprezentacja grafu jest przetrzymywana w liście list. Dodatkowo ze struktur danych użyto zbiorów liczbowych - pythonowych set(), na których można wykonywać typowe dla zbiorów operacje - np. znak odejmowania spowoduje wykluczenie tej wartości liczbowej ze zbioru, w przypadku kodu zbiór miast pomniejszy się o wskazane miasto. W ten sposób unikniemy wchodzenia dwa razy do tego samego miasta.

W pliku wykonywalnym programu istnieje możliwość podglądnięcia, w jaki sposób algorytm dzielił miasta na zbiory i wyliczał najlepsze rozwiązanie.

```
cel: pozostale miasta:
0 {1, 2, 3}
```

```
cel: pozostale miasta:
3 {1, 2}
cel: pozostale miasta:
2 {1}
cel: pozostale miasta:
1 set()
A oto efekt algorytmu:
[1, 2, 3, 0]
```

3. Wyniki eksperymentu

Wyniki generowano dla najmniejszej instancji równej 2, i zwiększając o 2 miasta mierzono czas maksymalnie do 14 miast. Ze względów technicznych program nie był w stanie policzyć dla większej instancji.

Wykres 1 - Wykres zależności czasu wykonywania algorytmu w zależności od ilości miast

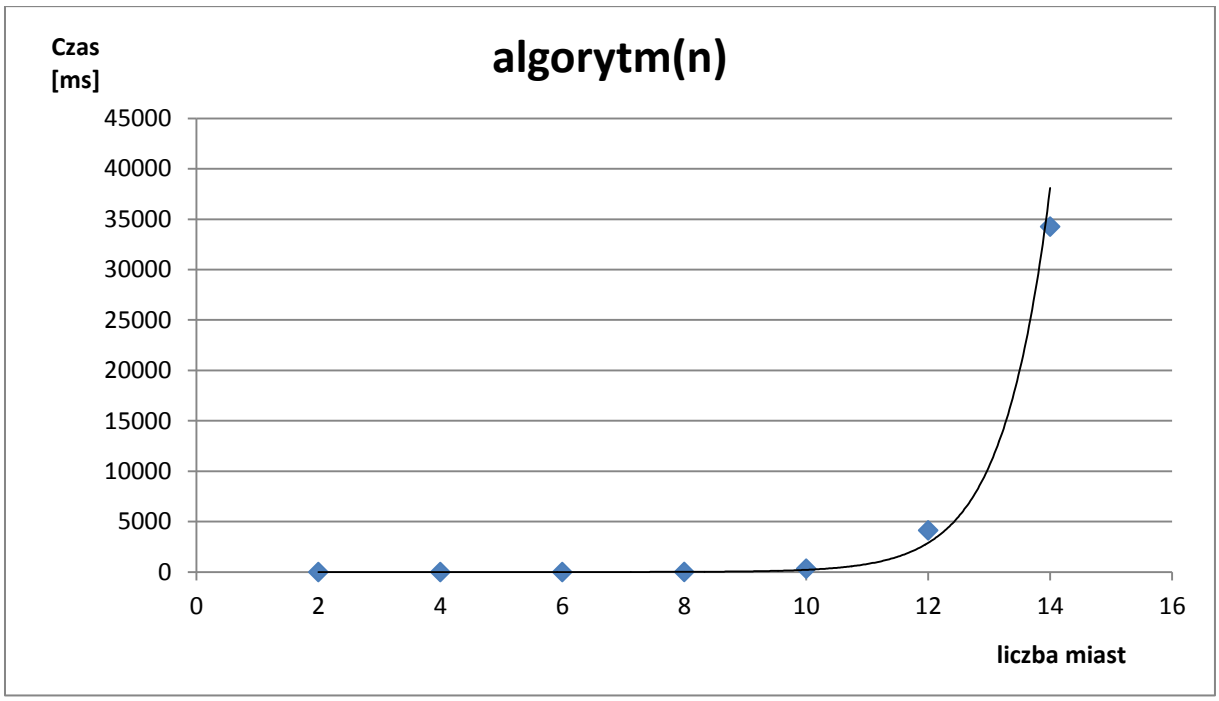

Tabela 1 - Wyniki pomiarów czasowych algorytmu programowania dynamicznego

Ilość miast	Czas [ms]
2	0,01
4	0,15
6	0,46
8	11,17
10	332,12
12	4135,24
14	34266,86

Wykonano również pomiary pod kątem porównania efektywności algorytmu programowania dynamicznego na tle już poznanych algorytmów rozwiązujących problem komiwojażera: zupełnego i Branch&Bound.

Efekt wyglądał następująco:

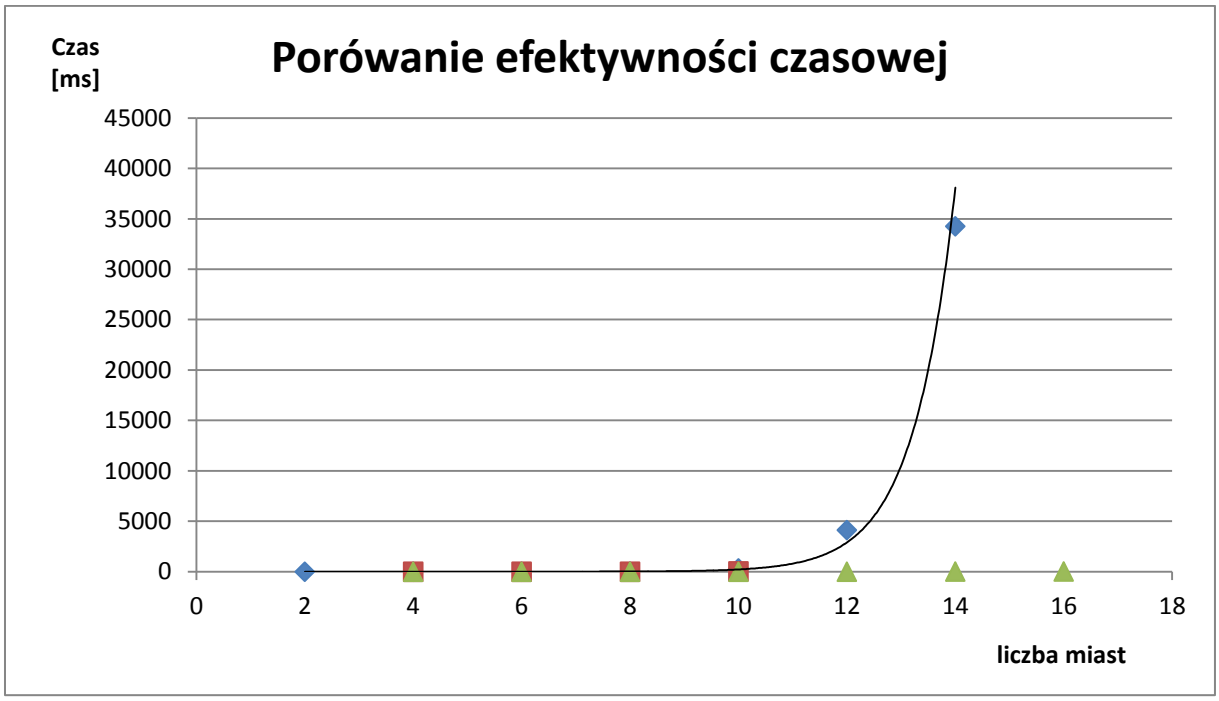

Tabela 2 - Wyniki pomiarów czasowych wszystkich 3 algorytmów

Ilość miast	Zupełny	B&B	Dynamiczny
4	0,001	0,0002	0,01
6	0,018	0,0012	0,15
8	0,566	0,0185	0,46
10	35,099	0,09	11,17
12		7,409	332,12
14		21,38	4135,24
16		29,925	34266,86

Dla większej liczby instancji miast w przypadku algorytmu zupełnego nie udało się obliczyć rozwiązania ze względu na charakterystykę algorytmu zupełnego.

Z wykresu wynika, że wciąż najefektywniejszym algorytmem pozostaje branch&bound.

4. Wnioski

Programowanie dynamiczne pozwala na zaplanowanie algorytmu, który znajdzie zastosowania do rozwiązywania zagadnień optymalizacyjnych. Rozwiązaniem optymalnym jest rozwiązanie posiadające maksimum lub minimum.

Z eksperymentu wynika, że algorytm programowania dynamicznego pozwala na znalezienie rozwiązania w znacznie krótszym okresie czasu. W jego możliwości jest również znalezienie rozwiązania dla instancji o większych rozmiarach. Szczególną cechą programowania dynamicznego jest fakt, że działa szybciej od algorytmów siłowych, a zarazem uzyskujemy funkcję celu z bardziej dokładną wartością od algorytmów zachłannych.

5. Bibliografia

[1] T. Cormen, C.E. Leiserson, R.L. Rivest, *Wprowadzenie do algorytmów,* WNT ,2003.

[2] R. Bellman. *On the Theory of Dynamic Programming*, Proceeding of the National Academy of Sciences (USA), 1952.

[3] K.Mroczek, *Problem komiwojażera - konkurs prac uczniowskich,* Instytut Matematyki UW, Warszawa

[4] http://pl.wikipedia.org/wiki/Programowanie_dynamiczne