Projektowanie efektywnych algorytmów
Projekt – zadanie 1

Algorytm przeglądu zupełnego, programowania dynamicznego, oraz przeglądu i ograniczeń

Problem komiwojażera

Autor: Filip Przygoński, 248892
Prowadzący: mgr inż. Antoni Sterna
Grupa zajęciowa: śr. 13:15

Wstęp teoretyczny

Problem komiwojażera (Travelling Salesman Problem - TSP)

W problemie komiwojażera dane jest n miast. Komiwojażer zaczyna w pierwszym mieście, odwiedza każde inne miasto jeden raz i wraca do pierwszego miasta. Rozwiązaniem problemu jest najkrótsza droga spełniająca powyższy warunek.

Patrząc informatycznie, problem można zobrazować jako ważony, skierowany (jeśli 'koszt' przejazdu z miasta 1 do miasta 2 jest różny od przejazdu z miasta 2 do miasta 1), graf pełny, gdzie wierzchołkami są miasta.

Przegląd zupełny (Brute Force)

Przegląd zupełny, jak nazwa wskazuje, polega na sprawdzeniu wszystkich możliwych permutacji i wybranie z nich tej, gdzie suma wag krawędzi jest najmniejsza. Ponieważ ta metoda sprawdza wszystkie możliwości, mamy gwarancję znalezienia poprawnego rozwiązania. Oczywistą wadą jest złożoność obliczeniowa $(n-1)!$, która sprawia że dla dużych wartości n rozwiązanie problemu tą metodą jest praktycznie niemożliwe przez niewyobrażalnie długi czas potrzebny do sprawdzenia każdej permutacji.

Programowanie dynamiczne (Dynamic Programming)

Koncepcja programowania dynamicznego opiera się na dzieleniu skomplikowanego problemu na mniejsze podproblemy, aż do momentu gdy podproblemy stają się trywialne. Każdy rozwiązany podproblem jest zapamiętywany, dzięki czemu nie trzeba rozwiązywać go wielokrotnie, wystarczy wziąć wcześniej otrzymany wynik. Prostym przykładem jest obliczenie n-tej liczby ciągu Fibonacciego. Wzór rekurencyjny:

$$fib(n) = fib(n-1) + fib(n-2)$$

Licząc np. $fib(5)$, musimy obliczyć $fib(4)$ i $fib(3)$, żeby mieć $fib(4)$, musimy obliczyć $fib(3)$ i $fib(2)$ etc. Już tutaj widać, że liczenie np. $fib(3)$ odbędzie się więcej niż jeden raz, oraz łatwo zauważyć że dla bardzo dużego n bardzo dużo działań będzie się powtarzać. Za pomocą programowania dynamicznego, możemy zapamiętać każde obliczone $fib(n-i)$, i zamiast obliczać funkcję ponownie, wziąć wcześniej obliczoną wartość.

Aby rozwiązać problem komiwojażera tym sposobem, musimy znaleźć sposób dzielenia problemu na podproblemy. Dla przykładu, mając 4 miasta, komiwojażer musi zacząć w mieście 1, przejść przez miasta 2, 3, 4 i wrócić do 1 najkrótszą ścieżką. Teraz mamy podproblem, musimy znaleźć minimum z trzech opcji:

droga 1->2 + najkrótsza ścieżka od 2, prowadząca przez 3, 4, kończąca na 1

droga 1->3 + najkrótsza ścieżka od 3, prowadząca przez 2, 4, kończąca na 1

droga 1->4 + najkrótsza ścieżka od 4, prowadząca przez 2, 3, kończąca na 1

Rozwijając np. 1->2, musimy znaleźć kolejne minimum z opcji:

1->2->3 + najkrótsza ścieżka od 3, prowadząca przez 4, kończąca na 1

1->2->4 + najkrótsza ścieżka od 4, prowadząca przez 3, kończąca na 1

Rozwijając 1->2->3, mamy:

1->2->3->4 + najkrótsza ścieżka od 4, prowadząca przez Ø, kończąca na 1

Ścieżka 4->1 jest nam znana, więc jest to problem trywialny. Wracając do podproblemu wyżej (1->2->3 + minimum), mamy 3->4->1, 4->1 znamy z problemu niżej, dodajemy do drogi 3->4 i mamy kolejny rozwiązany podproblem. Musimy rozwiązać również problem 1->2->4 + minimum, wybrać z nich obu minimum, i to „przekazać" do problemu wyżej, etc.

Zatem ogólny wzór można określić tak:

$$f(i,S) = \min\left(d_{i\to k} + f(k, S-k)\right)$$

gdzie i to miasto startowe, S to zbiór miast, przez które trzeba przejść, k to dowolne miasto ze zbioru S, a $d_{i\to k}$ to droga pomiędzy miastami i oraz k.

Programowanie dynamiczne wykorzystujemy do zapamiętywania obliczonych już rozwiązań, za pomocą tablicy dwuwymiarowej o rozmiarach $2^n - 1$ (-1, ponieważ pierwsze miasto jest już ustalone) na n. Zbiory będą reprezentowane jako maski bitowe, tzn. kolumna np. o numerze 10 reprezentuje zbiór zawierający miasta 3 i 1 (licząc od 0), ponieważ $10_{10} = 1010_2 = 2^3 + 2^1$. Wiersze reprezentują początkowe wierzchołki, tzn. wiersz 2 i kolumna 10 trzyma w sobie wagę trasy zaczynającej się od miasta 2, przechodzącej przez zbiór {1, 3} i kończącej na mieście 0.

Metoda podziału i ograniczeń (Branch and Bound)

Branch and Bound polega na przeglądaniu drzewa reprezentującego wszystkie ścieżki przez jakie może przejść algorytm. Algorytm zaczyna się w korzeniu i tworząc potomków konstruuje ich rozwiązania, oraz oblicza ich granicę, która pozwoli określić, czy potomek jest obiecujący czy nie. W dalszej fazie algorytm bada tylko tych potomków, którzy są obiecujący, i tworzy potomków tych potomków etc. Pozwala to zmniejszyć liczbę sprawdzanych podgałęzi wcześniej wspomnianego drzewa.

W programie zostały zaimplementowane dwie strategie przeszukiwań.

Przeszukiwanie wszerz (Breadth Search), pseudokod:

```
queue = new PriorityQueue
v = korzeń drzewa
queue.Enqueue(v)
best = value(v)
bestRoute = items(v)
while(queue.HasItems())
     v = queue.Dequeue()
     foreach (child of v)
          if(value(child) < best)
               best = value(child)
               bestRoute = items(child)
          if(bound(child) < best)
               queue.Enqueue(child)
```

Przeszukiwanie najpierw najlepszy (Best First), pseudokod:

```
queue = new PriorityQueue
v = korzeń drzewa
queue.Enqueue(v)
best = value(v)
bestRoute = items(v)
while(queue.HasItems())
     v = queue.Dequeue()
     if (bound(v) < best)
          foreach (child of v)
               if(value(child) < best)
                    best = value(child)
                    bestRoute = items(child)
               if(bound(child) < best)
                    queue.Enqueue(child)
```

Dla problemu komiwojażera, funkcje BnB działają następująco:

- *value*(v) – jeśli trasa przebiega przez wszystkie wierzchołki, zwraca koszt całej trasy, w przeciwnym wypadku zwraca maksymalną wartość;
- *items*(v) – każdy element drzewa zawiera listę wierzchołków, przez które dotychczas przeszedł, funkcja zwraca tę listę;
- *bound*(v) – suma wagi aktualnej (niepełnej) trasy + suma minimalnych wag krawędzi wychodzących z wierzchołków, z których „jeszcze można wyjść" do wierzchołków, których jeszcze nie uwzględniono w trasie.

Przykład

0	1	2	3	4
1	0	5	6	7
2	5	0	8	9
3	6	8	0	10
4	7	9	10	0

Macierz sąsiedztwa

Wartość *bound* dla korzenia (tylko wierzchołek 0) wynosiłaby:

$$0 + \min(1,2,3,4) + \min(5,6,7) + \min(5,8,9) + \min(6,8,10) + \min(7,9,10) = 24$$

Dla elementu drzewa z trasą np. 0->1 z kolei:

$$bound = 1 + \min(5,6,7) + \min(8,9) + \min(8,10) + \min(9,10) = 31$$

Dla 0->1->2:

$$bound = 1 + 5 + \min(8,9) + \min(10) + \min(10) = 34$$

Etc.

Plan eksperymentu

Program został napisany w języku C#, w .NET Framework, w środowisku Visual Studio 2019.

Ponieważ .NET w swoich kolekcjach nie posiada kolejki priorytetowej, w programie użyto kolejki priorytetowej z pakietu *MedallionPriorityQueue*: https://www.nuget.org/packages/MedallionPriorityQueue/1.1.0

Badane rozmiary problemu to 4, 5, 6, 7, 8, 9, 10, 11, 12 dla wszystkich algorytmów, oraz większe rozmiary, dopóki średni czas pojedynczego wykonania algorytmu był „rozsądny".

Wagi krawędzi w grafie były losowane z przedziału [1, 100].

Do mierzenia czasu wykorzystano klasę *Stopwatch* z przestrzeni nazw *System.Diagnostics,* mierzącą czas z dokładnością do nanosekund: https://docs.microsoft.com/en-gb/dotnet/api/system.diagnostics.stopwatch

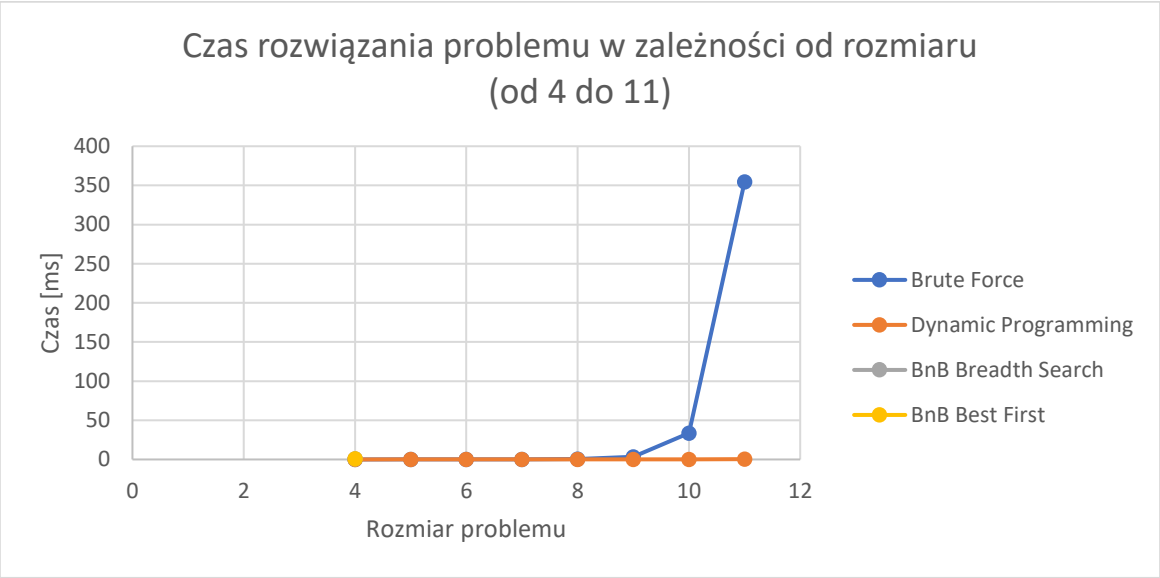
Test dla każdego algorytmu i każdego rozmiaru problemu został wykonany 101 razy, ponieważ wynik pierwszego testu może zaburzyć średnią (dokumentacja Microsoft, link powyżej).

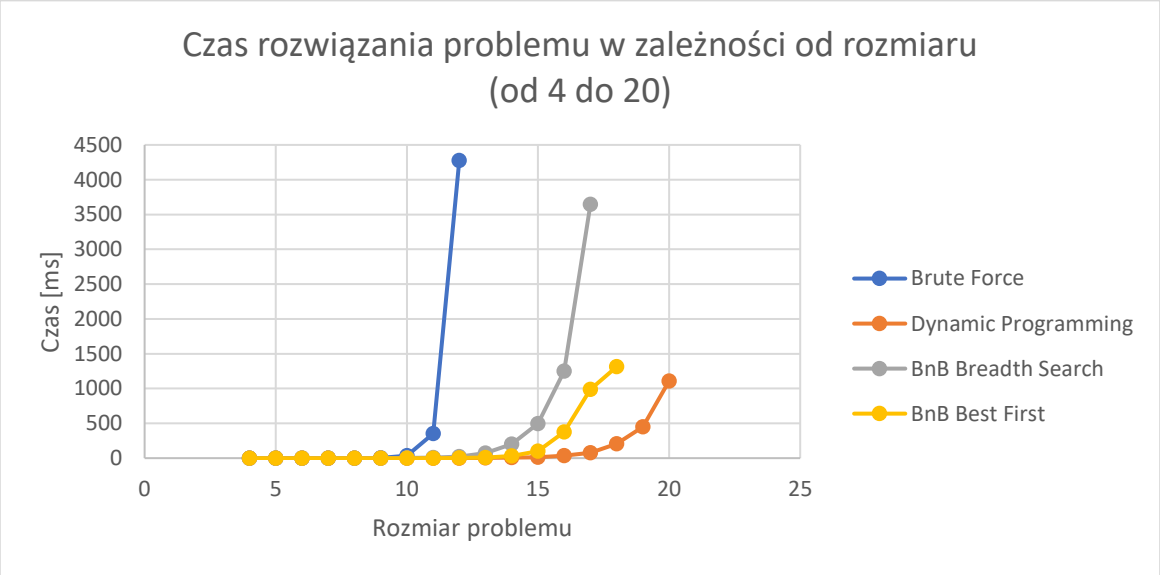
Do każdego testu generowano nowy, losowy graf.

Wyniki eksperymentu

Czasy podane w milisekundach.

Rozmiar problemu	Brute Force	Dynamic Programming	BnB Breadth Search	BnB Best First
4	0,000614	0,000871	0,004025	0,002273
5	0,001265	0,001806	0,01164	0,006349
6	0,007042	0,002984	0,05925	0,01507
7	0,05096	0,00817	0,1591	0,1069
8	0,3862	0,03148	0,2837	0,1206
9	3,546	0,07731	1,050	0,2772
10	33,63	0,1721	2,309	0,8991
11	354,4	0,4255	5,999	0,8813
12	4277	0,9831	19,71	2,813
13	-	2,624	74,24	7,597
14	-	5,904	199,5	30,12
15	-	13,71	494,6	98,98
16	-	34,27	1250	373,7
17	-	76,28	3645	988,3
18	-	204,6	-	1312
19	-	450,7	-	-
20	-	1107	-	-





Wnioski

Przegląd zupełny dla najmniejszych badanych rozmiarów problemu okazywał się najszybszym algorytmem. Dla takich rozmiarów możliwych permutacji jest wystarczająco mało, że opłacało się po prostu sprawdzić wszystkie. Jednak już dla rozmiaru 8, Brute Force był wolniejszy od pozostałych algorytmów. A dla rozmiaru 13, pojedyncze wykonanie problemu trwało zbyt długo, aby móc przeprowadzić test dla takiego rozmiaru 101 razy w rozsądnym czasie.

Z dwóch porównywanych metod podziału i ograniczeń, szybszą okazała się metoda przeszukiwania najpierw najlepszy, co zgadza się z teorią.

Najszybszym algorytmem okazało się programowanie dynamiczne. Jednakże trzeba pamiętać, że odbywa się to kosztem pamięci - złożoność pamięciowa rośnie wykładniczo.

Źródła

http://antoni.sterna.staff.iiar.pwr.wroc.pl/pea/PEA_wprowadzenie.pdf

https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf

https://cs.pwr.edu.pl/zielinski/lectures/om/mow10.pdf