Data 04.11.2017r
Prowadzący: dr inż. Jarosław Mierzwa

Sprawozdanie z zadania projektowego nr. 1

„Implementacja i analiza efektywności algorytmu programowania dynamicznego dla asymetrycznego problemu komiwojażera”

Autor: Bartosz Pogoda, 225988

Spis treści

1. Wstęp teoretyczny

1.1. Opis rozpatrywanego problemu

Problem komiwojażera definiowany jest dla grafów pełnych. Wierzchołki grafu nazywane są miastami, a krawędziom przypisane są wartości reprezentujące odległości między nimi. Problem polega na odnalezieniu najkrótszej ścieżki, takiej aby odwiedzić każde miasto dokładnie raz i wrócić do punktu wyjścia.

Wiele problemów, występujących w sektorach takich jak transport oraz logistyka można sprowadzić do problemu komiwojażera. Zastosowanie efektywnych algorytmów rozwiązujących ten problem nie rzadko prowadzi do redukcji kosztów i optymalizacji pewnych rzeczywistych procesów. Przykładową aplikacją mogło by być wyznaczenie optymalnej trasy dla autobusu szkolnego, tak aby odwiedził wszystkie przystanki z dziećmi w jak najkrótszym okresie czasu. Wierzchołkami grafu byłyby poszczególne przystanki, a wartościami na krawędziach szacunkowy czas przemieszczenia się autobusu między przystankami. Warto tutaj zauważyć, że czas przemieszczania się z przystanku A do B, może różnić się od czasu przemieszczenia z B do A. Różnice te są uwzględnione w asymetrycznym problemie komiwojażera, który zostanie poddany analizie w tym sprawozdaniu.

1.2. Opis algorytmu

W celu badań zostały zaimplementowane dwa algorytmy rozwiązujące powyżej sformułowany problem – algorytm przeglądu zupełnego oraz algorytm Held-Karp oparty na metodzie programowania dynamicznego. Rozwiązania dla obydwu algorytmów są zawsze optymalne, ponieważ w ramach algorytmu zostają sprawdzone wszystkie możliwe ścieżki. Podstawową różnica między algorytmami, która zazwyczaj przemawia na korzyść algorytmu dynamicznego jest występująca pamięć. Wyniki częściowe – minimalne ścieżki dla mniejszych pod problemów, czyli o zredukowanej ilości miast są zapamiętywane, dzięki czemu pod problem nie musi być rozwiązywany ponownie przy sprawdzaniu kolejnych możliwych ścieżek. Pod problemem, który jest rozwiązywany, jest określenie minimalnej ścieżki od wierzchołka startowego, kończąc na pewnym wierzchołku, odwiedzając pewien podzbiór wierzchołków. Z rozwiązania takiego pod problemu można skorzystać wiele razy (proporcjonalnie więcej do wielkości instancji). Algorytm zostanie opisany bardziej szczegółowo w dalszej części sprawozdania.

1.3. Złożoności czasowe wynikające z literatury

Tabela 1. Teoretyczne złożoności czasowe algorytmów rozwiązujących asymetryczny problem komiwojażera.

Asymetryczny problem komiwojażera (n – ilość miast)	
Przegląd zupełny	Programowanie dynamiczne Held-Karp
$O(\boldsymbol{n}!)$	$O\ (2^n n^2)$

2. Przykład praktyczny

Przykład rozwiązania asymetrycznego problemu Komiwojażera według algorytmu Held-Karp dla instancji o rozmiarze 4.

Tabela 2. Przykładowa instancja problemu (n = 4). Tablica d[i][j]

i \ j	0	1	2	3
0	∞	5	9	13
1	2	∞	8	6
2	12	6	∞	3
3	21	7	9	∞

1. Przyjmijmy miasto nr 3 jako miasto startowe. Wybór początkowego miasta jest dowolny z punktu widzenia rozwiązania – wyznaczamy minimalny cykl przechodzący przez wszystkie miasta.

2. Oznaczmy $dist(A, dest)$ jako minimalną wartość ścieżki zaczynającej się w mieście startowym, kończącej w mieście *dest* oraz przechodzącej przez wszystkie miasta zawarte w zbiorze *A*.

   Zbiór A będzie zawsze jednym z podzbiorów zbioru miast z wyłączeniem miasta początkowego.

3. Oznaczmy $parent(A, dest)$ jako numer miasta, z podzbioru A, które odwiedzając jako ostatnie otrzymamy minimalną wartość ścieżki zdefiniowanej jako $dist(A, dest)$. Zapisywanie tej informacji jest niezbędne do późniejszego wyznaczenia minimalnej ścieżki.

4. Korzystając z definicji z punktu 2. możemy stwierdzić, że rozwiązanie problemu jest minimalną wartością ścieżki zaczynającej się w mieście startowym, kończącej w mieście startowym oraz przechodzącej przez wszystkie pozostałe miasta, a więc:

   4.1. $minDist = \text{dist}(\{0,1,2\}, 3)$

5. Jest to punkt wyjściowy algorytmu, który zdefiniowany jest w sposób rekurencyjny. Należy zauważyć, iż obliczenie minimalnej wartości ścieżki zaczynającej się w mieście startowym, kończącej w mieście *dest*, a przechodzącej przez miasta zawarte w zbiorze A, można podzielić na mniejsze problemy. Pod problemów jest tyle, ile miast w zbiorze A, ponieważ należy z tego zbioru wybrać miasto, które będzie w ścieżce tuż przed miastem *dest*. Wyznaczenie minimalnej ścieżki dla głównego problemu (koniec w mieście *dest*) sprowadza się wtedy do wyznaczenia minimalnej ze ścieżek dla każdego z miast w podzbiorze A traktowanych jako miasta przed dest – *preDest* uwzględniając również odległość z miasta *preDest* do miasta *dest*.

   5.1. $$dist(\{0,1,2\}, 3) = \min \begin{matrix} \{dist(\{1,2\}, 0) + d(0,3)\,, \\ dist(\{0,2\}, 1) + d(1,3), \\ dist(\{0,1\}, 2) + d(2,3)\} \end{matrix}$$

6. Korzystając z opisanej zależności rekurencyjnej wyznaczmy potrzebne wyrazy.

   6.1. $dist(\{1,2\}, 0) = \min\{dist(\{2\}, 1) + d(1,0), dist(\{1\}, 2) + d(2,0)\}$

   6.2. $dist(\{2\}, 1) = dist(\emptyset, 2) + d(2,1)$

   Należy zauważyć, że $dist(\emptyset, 2)$ jest definiowane jako wartość minimalnej ścieżki rozpoczynającej się w mieście początkowym (3), kończącej w mieście 2 oraz przechodzącej przez miasta zawarte w zbiorze pustym. A więc jest to odległość z miasta 3 do miasta 2, wartość tą odczytujemy z macierzy sąsiedztwa.

   Zapisujemy również $parent(\emptyset, 2)$ = 3, ponieważ jest to miasto, które wystąpi w tym przypadku przed miastem 2.

7. Opisane kroki powtarzamy, aż będziemy w stanie wyliczyć wartość opisaną w wyrażeniach 4.1. oraz 5.1. Szczegółowe obliczenia zostaną przedstawione poniżej.

$dist(\{2\},1) = dist(\emptyset,2) + d(2,1) = d(3,2) + d(2,1) = 9 + 6 = 15$
$parent(\emptyset,2) = 3$
$parent(\{2\},1) = 2$

$dist(\{1\},2) = dist(\emptyset,1) + d(1,2) = d(3,1) + d(1,2) = 7 + 8 = 15$
$parent(\emptyset,1) = 3$
$parent(\{1\},2) = 1$

$dist(\{1,2\},0) = \min\{dist(\{2\},1) + d(1,0), dist(\{1\},2) + d(2,0)\} = \min\{15 + 2,15 + 12\}$
$= \min\{17,27\} = 17$
$parent(\{1,2\},0) = 1$

$dist(\{0,2\},1) = \min\{dist(\{2\},0) + d(0,1), dist(\{0\},2) + d(2,1)\}$

$dist(\{2\},0) = dist(\emptyset,2) + d(2,0) = 3 + 12 = 15$
$parent(\{2\},0) = 2$

$dist(\{0\},2) = dist(\emptyset,0) + d(0,2) = d(3,0) + d(0,2) = 21 + 9 = 30$
$parent(\emptyset,0) = 3$
$parent(\{0\},2) = 0$

$dist(\{0,2\},1) = \min\{dist(\{2\},0) + d(0,1), dist(\{0\},2) + d(2,1)\} = \min\{15 + 5,30 + 6\}$
$= \min\{20,36\} = 20$
$parent(\{0,2\},1) = 0$

$dist(\{0,1\},2) = \min\{dist(\{1\},0) + d(0,2), dist(\{0\},1) + d(1,2)\}$

$dist(\{1\},0) = dist(\emptyset,1) + d(1,0) = 7 + 2 = 9$
$parent(\{1\},0) = 1$

$dist(\{0\},1) = dist(\emptyset,0) + d(0,1) = 21 + 5 = 26$
$parent(\{0\},1) = 0$

$dist(\{0,1\},2) = \min\{dist(\{1\},0) + d(0,2), dist(\{0\},1) + d(1,2)\} = \min\{9 + 9,26 + 8\}$
$= \min\{18,34\} = 18$
$parent(\{0,1\},2) = 0$

Wszystkie pod problemy zostały rozwiązane, teraz można przejść do rozwiązania głównego problemu (5.1):

$$dist(\{0,1,2\},3) = \min \begin{matrix} \{dist(\{1,2\},0) + d(0,3), \\ dist(\{0,2\},1) + d(1,3), \\ dist(\{0,1\},2) + d(2,3)\} \end{matrix} = \min\{17 + 13,20 + 6,18 + 3\}$$
$= \min\{30,26,21\} = \mathbf{21}$
$parent(\{0,1,2\},3) = 2$

8. Na tym etapie można wyznaczyć ścieżkę, korzystając z wartości zapisanych w tablicy *parent*.

   Ostatnim miastem odwiedzonym przed powrotem do miasta początkowego będzie:
   $parent(\{0,1,2\}, 3) = 2$

   Teraz odczytujemy miasto, które zostanie odwiedzone przed miastem nr 2:
   $parent(\{0,1\}, 2) = 0$

   Miasto odwiedzone przed miastem nr 0:
   $parent(\{1\}, 0) = 1$

   Kolejne miasto:
   $parent(\emptyset, 1) = 3$

   Finalna ścieżka ma postać **3 -> 1 -> 0 -> 2 -> 3**, a jej wartość wynosi **21**.

3. Opis implementacji algorytmu

3.1. Reprezentacja podzbioru

W obrębie programu poszczególne podzbiory miast są reprezentowane jako liczby całkowite. Jedynka w zapisie binarnym liczby całkowitej na i-tej pozycji (licząc od LSB) oznacza, że i-te miasto znajduje się w zbiorze. Taka reprezentacja podzbiorów ogranicza wielkość instancji wejściowych do 33, ponieważ podzbiory przechowywane są jako 4 bajtowe – 32 bitowe liczby, a jeden z wierzchołków (traktowany jako początkowy) nie jest uwzględniany w podzbiorach.

Operacje na podzbiorach zostały zdefiniowane w klasie pomocniczej *SubsetUtil*. Jako metody pomocnicze zostały zdefiniowane operacje: obliczenie ilości podzbiorów dla danej ilości miast przejściowych, wygenerowanie podzbioru pełnego dla danej ilości miast przejściowych, usunięcie miasta z podzbioru oraz sprawdzenie czy miasto istnieje w podzbiorze.

3.2. Struktury danych

Instancja wejściowa (graf) została przedstawiona jako dwuwymiarowa macierz sąsiedztwa. Rozwiązania pod problemów zapisywane są w dwóch dwuwymiarowych macierzach.

Macierz *distances* zawiera wartości najkrótszych ścieżek dla poszczególnych pod problemów. Macierz *parents* zawiera numer poprzedniego miasta, dla którego występuje najkrótsza ścieżka w poszczególnym pod problemie.

Obydwie macierze są wielkości $2^{n-1}\ x\ (n-1)$. Pierwszy wymiar macierzy wynika z ilości podzbiorów miast, przez które należy przejść na drodze od miasta początkowego z powrotem do miasta początkowego – jest to ilość podzbiorów zbioru (n-1) elementowego. Drugi wymiar macierzy wynika bezpośrednio z ilości miast będących na drodze do przejścia.

3.3. Szczegóły implementacji algorytmu

W celu uproszczenia indeksowania w obrębie struktur jako miasto startowe uznane zostaje ostatnie miasto. Miasta będące na drodze otrzymują wtedy indeksy 0 do n-1, które pokrywają się z naturalnym indeksowaniem tablic w języku C++.

W pierwszej kolejności następuje inicjalizacja struktur pomocniczych. Struktura zawierająca wyniki poszczególnych pod problemów zostaje wypełniona wartościami -1, co sygnalizuje, że dany pod problem nie został jeszcze rozwiązany i trzeba go rozwiązać.

Następnie wypełniany jest rząd macierzy dla podzbioru pustego. W tym rzędzie znajdują się wartości odległości od miasta początkowego do każdego z pozostałych wierzchołków. Wartości te będą stanowić ograniczenie dla rekurencji.

Kolejnym krokiem jest generacja podzbioru pełnego oraz wystartowanie rekurencji opisanej w punkcie 2. Na początku każdego wywołania funkcji rekurencyjnej następuje sprawdzenie, czy dany pod problem nie został wcześniej rozwiązany – jeśli tak, wcześniej obliczona wartość jest zwracana, jeśli nie – pod problem zostaje rozwiązany. Krok ten jest powtarzany rekurencyjnie, aż do rozwiązania wszystkich pod problemów.

4. Plan eksperymentu

4.1. Rozmiary problemów

Rozmiary problemów zostały dobrane eksperymentalnie, tak aby wykonywały się w rozsądnym czasie na dostępnym sprzęcie. Testy zostały przeprowadzone komputerze z procesorem czterordzeniowym i5-6300HQ 2.30GHz oraz wolną pamięcią operacyjną 4GB.

4.2. Generowanie losowych instancji

Podczas generowania losowej instancji problemu o zadanej wielkości generowana jest osobno każda z całkowitych odległości między miastami w zakresie <1, 10000>. W generowanej macierzy sąsiedztwa pominięte są wartości na przekątnej (pętle), ponieważ nie są one istotne z punktu widzenia algorytmu.

4.3. Pomiar czasu

Do pomiaru czasu zostały użyte funkcje opierające się na działaniu funkcji dostarczonej w bibliotece systemu Windows - QueryPerformanceCounter(). Dokładność procedur pozwoliła na badanie czasów (oraz uzyskanie dokładności) rzędu mikrosekund.

4.4. Uśrednianie wyników

Większość odnotowanych wyników pomiaru jest średnią arytmetyczną wyników uzyskanych dla 100 różnych instancji problemów. Wyjątkiem są tutaj badania granicznych wielkości instancji, dla których zostało wykonanych po 10 iteracji. Wyniki te rozdzielone są w tabelach z wynikami linią pionową, a ich wartość podana została w sekundach.

5. Wyniki pomiarów

Poniżej zestawienie wyników pomiarów czasów działania algorytmów na poszczególnych reprezentacjach grafów w formie tabelarycznej oraz graficznej.

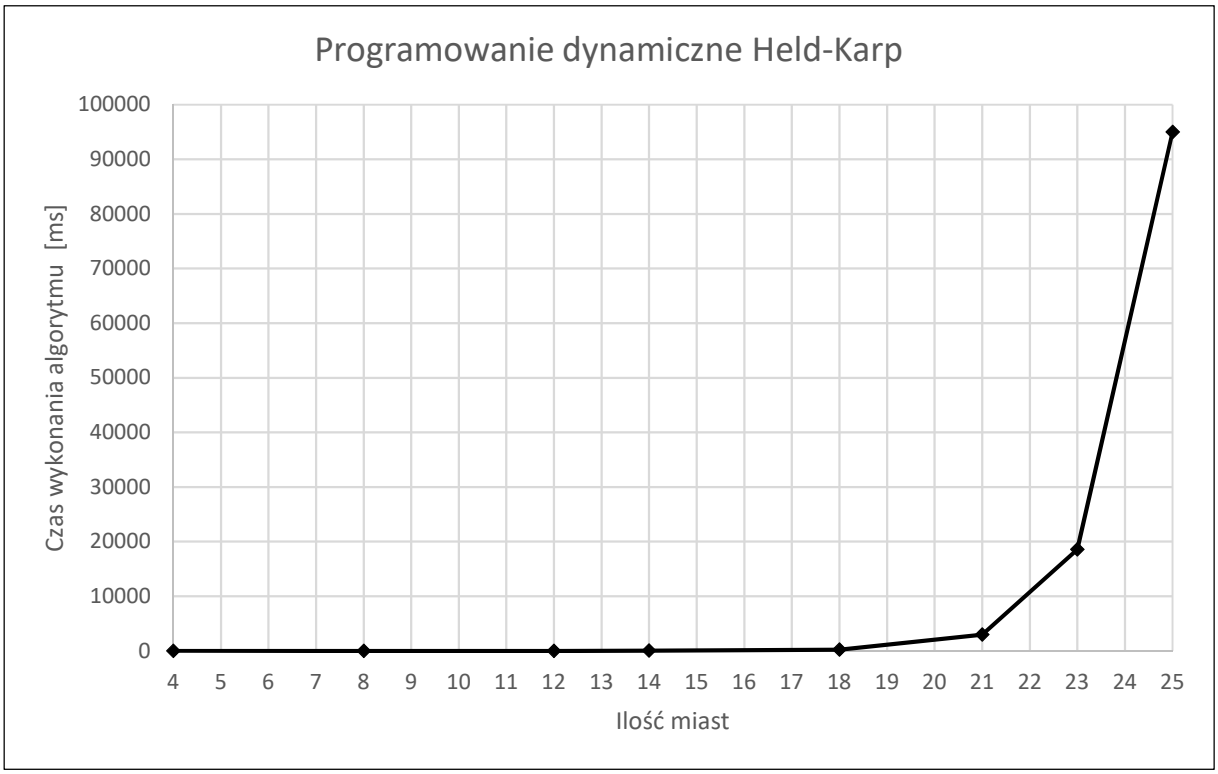
Tabela 3. Wyniki pomiarów czasu wykonania algorytmu programowania dynamicznego Held-Karp

Ilość miast	**4**	**8**	**12**	**14**	**18**	**21**	**23**	**25**
	[ms]					[s]		
Czas	0.002	0.038	1.471	8.092	236.152	2.9	18.6	95.0

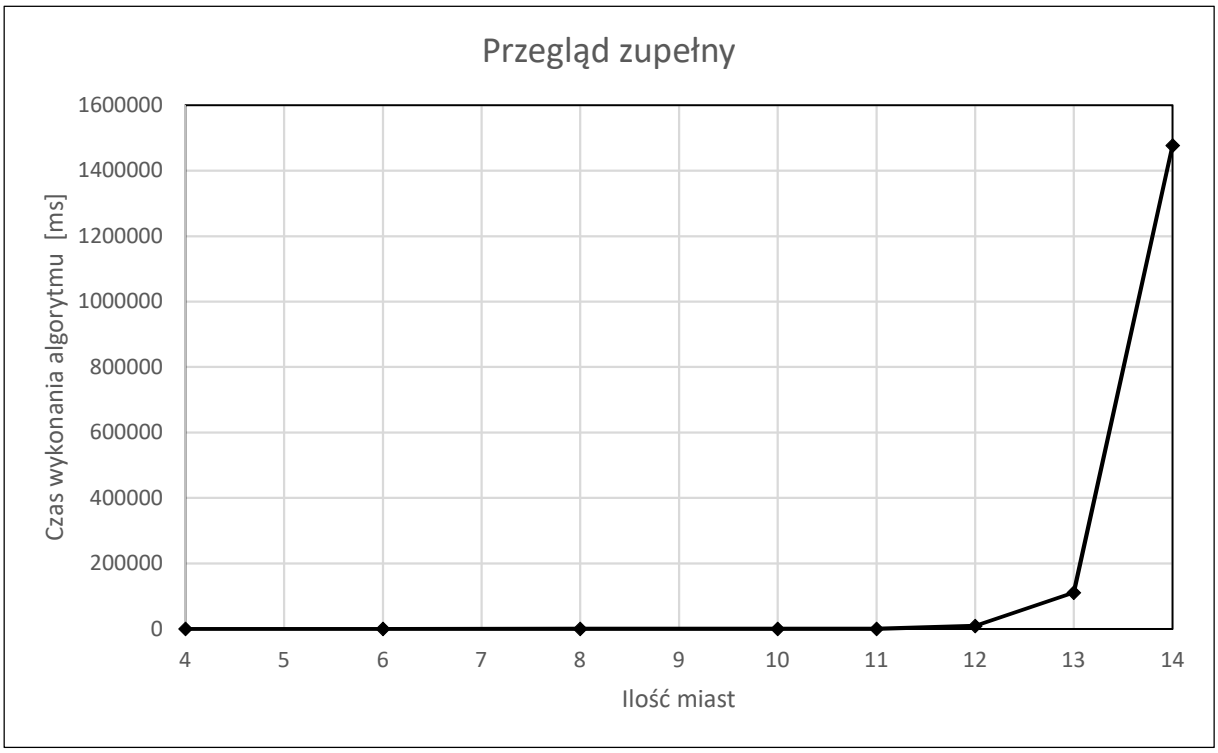
Tabela 4. Wyniki pomiarów czasu wykonania algorytmu przeglądu zupełnego

Ilość miast	**4**	**6**	**8**	**10**	**11**	**12**	**13**	**14**
	[ms]					[s]		
Czas	0.001	0.026	1.092	81.667	850.204	9.1	110.6	1477.3

Wykres 1. Zależność czasu wykonania algorytmu od wielkości instancji dla programowania dynamicznego

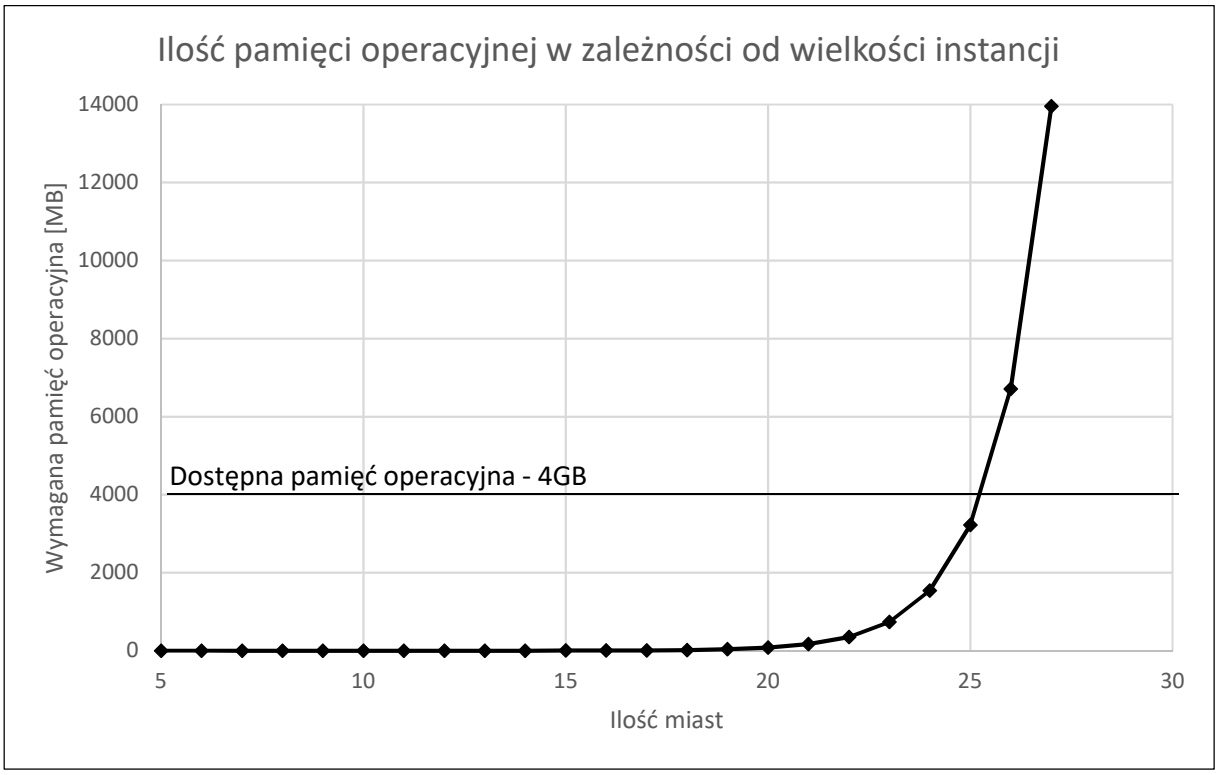


Wykres 2. Zależność czasu wykonania algorytmu od wielkości instancji dla przeglądu zupełnego



Wykres 3. Zestawienie czasów wykonywania obydwu algorytmów

Wykres 4. Zależność wymaganej ilości pamięci RAM od wielkości instancji dla algorytmu Held-Karp



6. Analiza wyników pomiarów i wnioski

6.1. Problem komiwojażera – przegląd zupełny

Algorytm przeglądu zupełnego zawsze znajduje najlepsze rozwiązanie, jednak jest bardzo wolny. Obrazuje to dobrze zestawienie przedstawione na Wykresie 11. Algorytm ma złożoność O(n!) co skutecznie eliminuje jego użycie dla dużych rozmiarów problemów. Wykorzystywany procesor potrzebował aż 15 minut dla wyznaczenie najkrótszej ścieżki dla 14 miast. (Tabela 9). Poniżej obliczenia ile w przybliżeniu mógłby się wykonywać algorytm dla 20 miast.

$$\frac{989.1}{14!}[s] - czas\ sprawdzenia\ jednej\ permutacji$$

$$\frac{989.1}{14!}20!\,[s] \approx 875\ lat - czas\ sprawdzenia\ 20!\ permutacji$$

6.2. Problem komiwojażera – programowanie dynamiczne Held-Karp

Algorytm oparty na programowaniu dynamicznym okazał się dużo szybszy od przeglądu zupełnego. Czynnikiem ograniczającym jego użycie jednak okazała się duża złożoność pamięciowa. Dla zaimplementowanego algorytmu ilość potrzebnej pamięci RAM można wyrazić następującym wzorem.

$$SIZE(n) = 2 * 2^{n-1} * (n-1) * 4\ [B] = 2^{n+2} * (n-1)\ [B]$$

4B reprezentuje wielkość jednego elementu tablicy. W algorytmie występują dwie tablice o wielkości $2^{n-1} * (n-1)$.

Zależność ta została przedstawiona na wykresie 4.

Zapotrzebowanie na pamięć rosnące wykładniczo skutkuje brakiem możliwości lub bardzo drogim zastosowaniem algorytmu dla większych instancji. Możliwe jest również przebudowanie algorytmu tak, aby korzystał z pamięci dyskowej, lecz taka pamięć jest znacznie wolniejsza.

7. Użyte materiały, źródła

1. Sposób pomiaru czasu
   http://staff.iiar.pwr.wroc.pl/antoni.sterna/sdizo/SDiZO_time.pdf

2. Sposób odczytu z plików
   http://staff.iiar.pwr.wroc.pl/antoni.sterna/sdizo/SDiZO_file.pdf

3. Informacje o algorytmach
   http://www.zio.iiar.pwr.wroc.pl/pea.html - wykłady
   https://en.wikipedia.org/

4. Informacje o złożoności obliczeniowej algorytmów
   http://bigocheatsheet.com/
   https://en.wikipedia.org/