

Grafy i Sieci

Sprawozdanie nr 3.

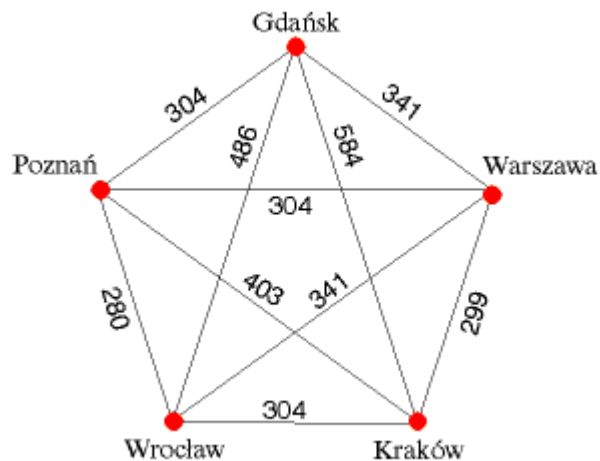
Problem komiwojażera

Marcin Nazimek
Piotr Jastrzębski

Warszawa, 14.01.2013

1 Szczegółowy opis zadania

Problem komiwojażera (TSP - ang. travelling salesman problem) jest zagadnieniem optymalizacyjnym, polegającym na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Nazwa związana jest z typowym problemem. Tytułowy komiwojażer podczas podróży chce odwiedzić określony zbiór miast ponosząc przy tym minimalny koszt, który rozumiany jest jako suma kosztów, które należy ponieść na przemieszczenie się pomiędzy sąsiednimi w cyklu miastami. Jako koszt rozumiany jest najczęściej czas lub odległość pomiędzy miastami. W cyklu Hamiltona każdy z wierzchołków uwzględniany jest tylko raz. Zatem komiwojażer nie przejeżdża przez żadne miasto więcej niż jeden raz, z wyjątkiem miasta, które stanowi początek cyklu - jest ono jednocześnie końcem drogi.



2 Ograniczenia

Zagadnienie jest problemem NP-zupełnym. Jedynym rozwiązaniem dającym gwarancję znalezienia najkrótszego cyklu jest porównanie ze sobą wszystkich możliwych cykli. Złożoność takiej metody wynosi $n!$ w związku z czym jej zastosowanie ogranicza się do zbiorów zawierających nie więcej niż kilkanaście wierzchołków. Ponieważ nie jest znany algorytm działający w czasie wielomianowym dający optymalne rozwiązanie, należy ograniczyć się do implementacji metod dających rozwiązanie zbliżone do optymalnego w rozsądnym czasie.

3 Dane wejściowe

W pierwszym wersie pliku podawana jest liczba wierzchołków. Następnie: kolejne wiersze macierzy sąsiedztwa. Np.:

```
5
0 2 4 3 6
1 0 5 5 6
2 4 0 2 3
4 6 4 0 2
3 6 1 5 0
```

4 Dane wyjściowe

Przestawiony zostanie znaleziony cykl, jego długość oraz czas operacji[ms]. Np.:

```
cycle: 4 0 2 1 3
distance: 23 time: 0.02
```

5 Testy

W programie będzie istniała możliwość wczytania własnego pliku z grafem testowym oraz generacja losowego grafu o zadanej liczbie wierzchołków.

6 Algorytmy

Zostały porównane dwa podejścia.

- Przeszukiwanie rozwiązań w poszukiwaniu minimum
- Strategia zachłanna

6.1 Przeszukiwanie rozwiązań w poszukiwaniu minimum

Podójście polega na wylosowaniu dowolnego cyklu Hamiltona i analizie jego sąsiedztwa. Jeśli okaże się, że któryś z sąsiadów jest rozwiązaniem lepszym - oznaczamy je jako aktualnie najlepsze i przeszukujemy jego sąsiedztwo. Operację powtarzamy tak długo, dopóki spośród sąsiadów aktualnie oznaczonego jako najlepsze rozwiązanie nie jesteśmy w stanie wskazać lepszego. Oznacza to, że znajdujemy się w minimum. Wadą tego podejścia jest możliwość trafienia do minimum lokalnego. Dlatego wykonane zostanie kilka przebiegów algorytmu. Liczba przebiegów jest parametrem p , którego zwiększenie daje szansę na uzyskanie lepszego rozwiązania, ale jednocześnie wydłuża czas potrzebny na uzyskanie rozwiązania.

Do przeszukiwania sąsiednich rozwiązań zostanie użyty algorytm 2-opt [1]:

$X = \{a, b, c, d, \dots, z, a\}$ - bieżący cykl

$|X|$ = suma długości krawędzi cyklu

$Y = \{\{ab\}\{bc\}\{cd\}\dots\{za\}\}$ - zbiór wszystkich kolejnych krawędzi w cyklu

Dopóki (!warunekPrzerwania)

 Dla każdej pary krawędzi ze zbioru Y : $\{ij\} \{kl\}$

$X' = X$

 Usuń ze zbioru krawędzi X' krawędzie $\{ij\} \{kl\}$

 Dodaj do zbioru krawędzi X' krawędzie $\{ij\} \{kl\}$

 Jeżeli $(|X'| < |X|)$

$X_{opt} = X'$

 Jeżeli $(X_{opt} == X)$

 warunekPrzerwania = true

Koniec

Złożoność obliczeniowa:

$$\Theta = p * s * n * (n - 1) * n * 1/2 \quad (1)$$

p - liczba przebiegów

n - liczba wierzchołków

s - średnia odległość od rozwiązania inicjacyjnego do minimum lokalnego

6.2 Strategia zachłanna

Strategia zachłanna polega na zbudowaniu minimalnego drzewa rozpinającego. Następnie następuje przejście od korzenia do kolejnych węzłów, przy czym jako kolejny wybierany jest najbliższy nieodwiedzony wierzchołek. W przypadku gdy dla konkretnego wierzchołka w drzewie nie istnieją sąsiedni nie odwiedzony, następuje przejście do innego wierzchołka w drzewie. Strategia polega na budowaniu rozwiązania poprzez wybór minimów w kolejnych

iteracjach. Nie daje to jednak pewności, że otrzymane rozwiązanie będzie najlepsze z możliwych.

Do budowy minimalnego drzewa rozpinającego zostanie wykorzystany algorytm Kruskala [2].

Utwórz las L z wierzchołków oryginalnego grafu

Utwórz zbiór S zawierający wszystkie krawędzie oryginalnego grafu.

Dopóki S nie jest pusty:

Wybierz i usuń z S krawędź o minimalnej wadze.

Jeżeli krawędź ta łączyła dwa różne drzewa

dodaj ją do lasu L, tak aby połączyła dwa odpowiadające drzewa w jedno

W przeciwnym wypadku

odrzuć ją.

Po zakończeniu algorytmu L jest minimalnym drzewem rozpinającym.

Złożoność obliczeniowa: W pierwszym etapie decydującym o złożoności czynnikiem jest posortowanie wszystkich krawędzi. Złożoność tej operacji przy wykorzystaniu algorytmu *quicksort* wyniesie $n \cdot \lg(n)$. Później należy w każdej iteracji sprawdzać czy bieżąca gałąź łączy różne poddrzewa. Wymaga to sprawdzenia za sobą każdej pary krawędzi z osobnych podrzew.

$$\Theta = n * n \quad (2)$$

7 Implementacja

Program został napisany w języku Java. Do jego poprawnego działania wymagana jest wcześniejsza instalacja *Java Virtual Machine* w wersji 1.7 (co nie wyklucza, że program będzie poprawnie funkcjonował również na starszych wersjach oprogramowania).

Projekt składa się z 3 komponentów:

- Program wykorzystujący przeszukiwanie rozwiązań w poszukiwaniu minimum - algorytm 2-op
- Program wykorzystujący strategię zachłanną - Algorytm Kruskala
- Generator losowych grafów testowych

7.1 Poszukiwanie minimum

Aby uruchomić program realizujący algorytm należy w katalogu `.../GIS/src/` uruchomić polecenie:

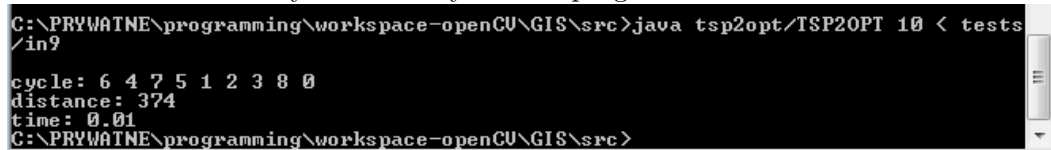
```
java tsp2opt/TSP2OPT N <testfile
```

N - liczba iteracji algorytmu (startów z nowego losowego rozwiązania). Jeśli

nie zostanie podany przyjmuje wartość 5.

testfile - plik z danymi grafu testowego. Jeśli nie zostanie podany należy wpisać dane za pośrednictwem konsoli.

Rysunek 1: Wywołanie programu.



```
C:\PRYWATNE\programming\workspace-openCU\GIS\src>java tsp2opt/TSP2OPT 10 < tests/in9
cycle: 6 4 7 5 1 2 3 8 0
distance: 374
time: 0.01
C:\PRYWATNE\programming\workspace-openCU\GIS\src>
```

7.2 Strategia zachłanna

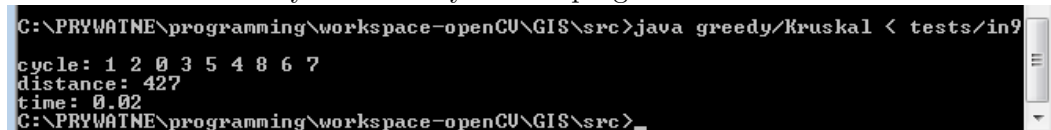
Aby uruchomić program realizujący algorytm należy w katalogu `.../GIS/src/` uruchomić polecenie:

java greedy/Kruskal <testfile

testfile - plik z danymi grafu testowego. Jeśli nie zostanie podany należy wpisać dane za pośrednictwem konsoli.

W katalogu `.../GIS/src/tests` znajduje się kilka plików, na których można przetestować działanie programu.

Rysunek 2: Wywołanie programu.



```
C:\PRYWATNE\programming\workspace-openCU\GIS\src>java greedy/Kruskal < tests/in9
cycle: 1 2 0 3 5 4 8 6 7
distance: 427
time: 0.02
C:\PRYWATNE\programming\workspace-openCU\GIS\src>
```

7.3 Generator grafów testowych

Aby uruchomić program umożliwiający generację losowych grafów o zadanym rozmiarze należy w katalogu `.../GIS/src/` uruchomić polecenie:

java generator/Generator i podać liczbę wierzchołków

Pliki zapisane zostają w katalogu `.../GIS/src/tests`

Rysunek 3: Wywołanie programu.



```
C:\PRYWATNE\programming\workspace-openCU\GIS\src>java generator/Generator
Podaj liczbe wierzcholkow: 8
Wygenerowano plik: C:\PRYWATNE\programming\workspace-openCU\GIS\src\tests\in8
C:\PRYWATNE\programming\workspace-openCU\GIS\src>
```

8 Wyniki testów

Programy były uruchamiane na sprzęcie o następujących parametrach:

- Procesor Intel Core i5-2520M 2.5GHz, liczba rdzeni: 4
- Dostępna pamięć RAM: 3,90 GB

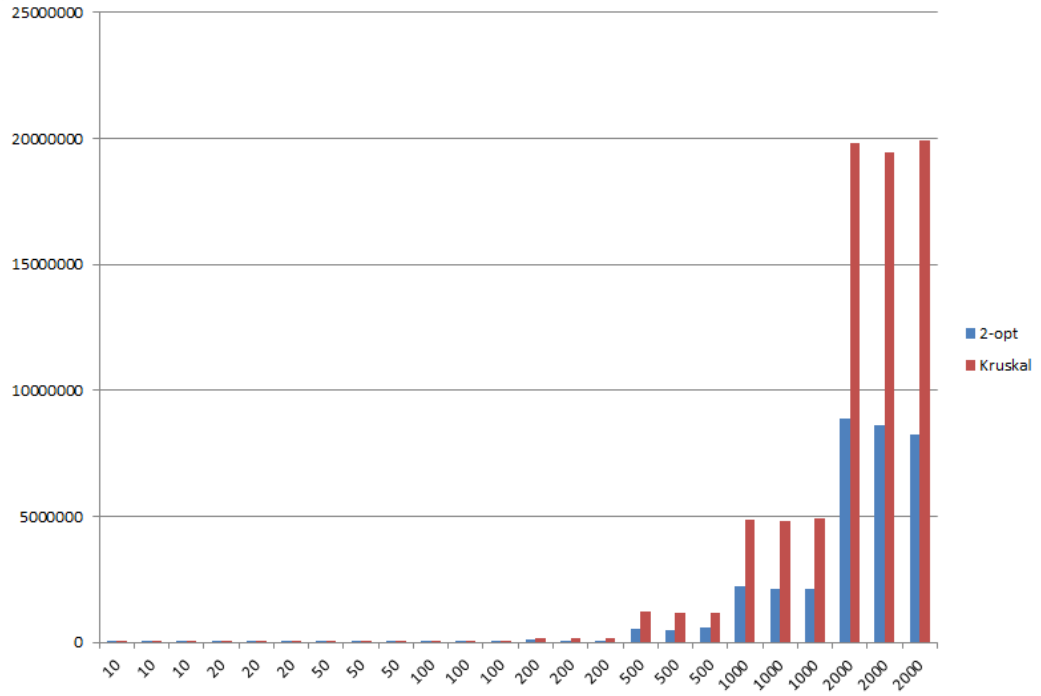
Testy zostały wykonane na losowych grafach o rozmiarach: 10, 20, 50, 100, 200, 500, 1000, 2000 wierzchołków. Znajdują się one w katalogu:

.../src/tests/documentation. Poniżej znajdują się wyniki czasowe oraz długość cykli dla poszczególnych testów programów. **Algorytm 2-opt wyko-**
nano dla liczby startów = 5

Rozmiar grafu	Algorytm Kruskala		Algorytm 2-opt	
	długość	czas[s]	długość	czas[s]
10	281	0.003	338	0.001
10	295	0.004	238	0.001
10	246	0.004	465	0.002
20	919	0.008	1563	0.001
20	933	0.004	1983	0.001
20	1089	0.005	2049	0.001
50	6780	0.19	11910	0.01
50	5781	0.18	11039	0.02
50	5130	0.19	10784	0.01
100	20407	0.057	44899	0.002
100	20819	0.063	44453	0.002
100	21309	0.053	45624	0.001
200	96362	0.045	189465	0.003
200	86690	0.046	193112	0.003
200	87837	0.048	187759	0.002
500	531609	0.17	1248646	0.05
500	488124	0.191	1196571	0.06
500	574521	0.177	1172881	0.06
1000	2217361	0.357	4891008	0.012
1000	2110133	0.338	4821965	0.012
1000	2147762	0.347	4935702	0.012
2000	8903871	1.168	19797518	0.028
2000	8608136	1.140	19456883	0.030
2000	8249490	1.312	19897376	0.033

Poniższy wykres przedstawia wyniki pomiarów umieszczone w tabeli. Warto zauważyć, że algorytm Kruskala daje zawsze około 2-krotnie lepsze wyniki niż algorytm 2-opt. W obliczu przyjętego parametru dla liczby losowań początkowych, rozwiązań nie można jednak obiektywnie porównać.

Rysunek 4: Wykres



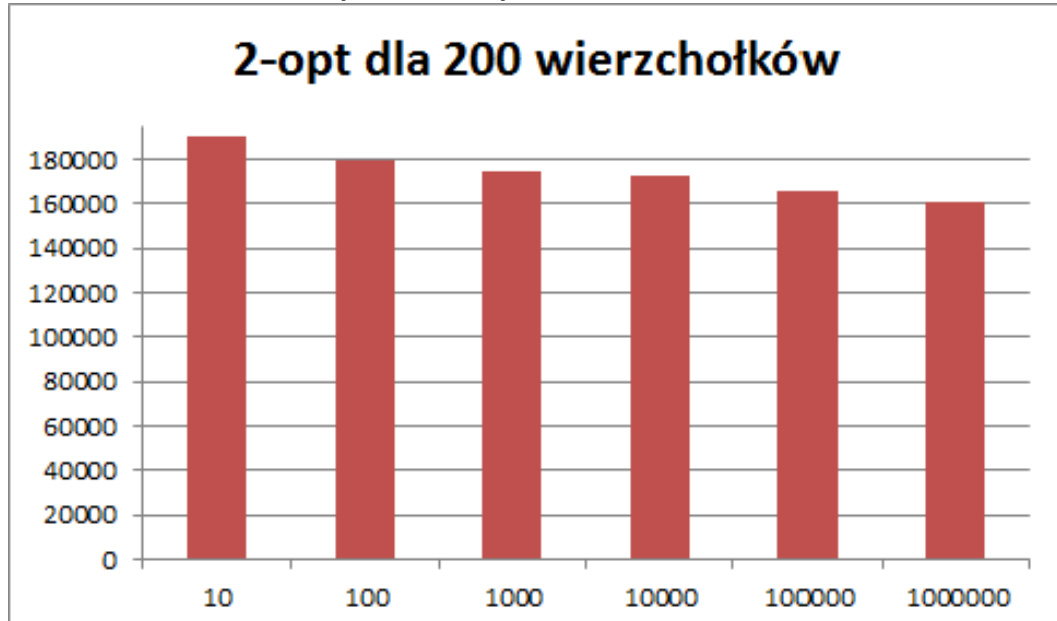
Istotne jest jednak, że dla kilku zadanych losowych grafów o określonej liczbie wierzchołków każdy algorytm daje względnie przewidywalny rezultat. Zarówno długość uzyskanego cyklu jak i czas osiągnięcia rozwiązania różnią się pomiędzy poszczególnymi próbkami o maksymalnie około 15

8.1 Wpływ liczby losowań na wynik algorytmu 2-opt

Ponieważ algorytm 2-opt daje znacznie gorsze rezultaty niż algorytm Kruskala można się spodziewać, że dla małej liczby powtórzeń algorytm często kończy działanie w minimum lokalnym. Należy więc poddać analizie wpływ liczby losowań na wynik działania programu. Poniższa tabela ilustruje przebieg eksperymentu do grafu o **200** wierzchołkach.

Liczba startów	Algorytm 2-opt	
	długość	czas[s]
10	190043	0.004
100	179011	0.031
1000	174580	0.065
10000	172365	0.218
100000	165905	1.995
1000000	161066	19.99

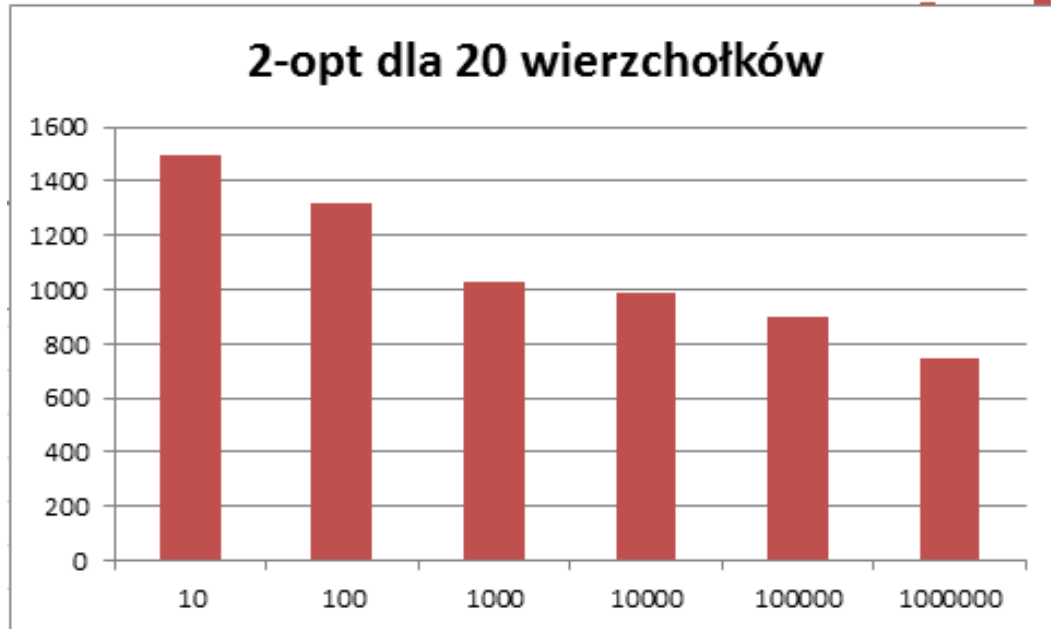
Rysunek 5: Wyniki testów



Zwiększenie Liczby losowań przynosi co prawda poprawę rezultatu, ale nie jest to zysk znaczący biorąc pod uwagę znaczne wydłużenie czasu obliczeń. Dużo lepszą poprawę możemy natomiast zaobserwować dla grafu złożonego z **20** wierzchołków.

Liczba startów	Algorytm 2-opt	
	długość	czas[s]
10	1496	0.001
100	1317	0.007
1000	1031	0.026
10000	991	0.053
100000	901	0.232
1000000	746	1.797
10000000	548	17.954

Rysunek 6: Wyniki testów



Można przeprowadzić wiele testów dla różnych zestawów danych jednak dotychczasowa Liczba pomiarów jest wystarczająca do sformułowania wniosków z eksperymentu.

9 Wnioski

Obserwacje wyników oraz analiza algorytmów doprowadziły nas do poniższych wniosków:

- Nie istnieje możliwość porównania algorytmów celem stwierdzenia, który z nich jest obiektywnie lepszy. Algorytm k-opt jest metodą heurystyczną, w związku z czym nigdy nie mamy gwarancji, że otrzymane rozwiązanie jest najlepsze. Jego wyniki poprawiają się w zależności od czasu jakim dysponujemy. Inaczej sytuacja wygląda w przypadku strategii zachłannej. Wykorzystanie minimalnego drzewa rozpinającego za pomocą algorytmu Kruskala zwraca wynik po pewnym czasie. Ponieważ algorytm nie zawiera elementów losowych, jest deterministyczny i kolejne iteracje nie są w stanie poprawić wyniku. W tym przypadku również nie mamy gwarancji uzyskania najlepszego rozwiązania.
- W sytuacji gdyby zlecono nam znalezienie jak najkrótszego cyklu Hamiltona w zadanym grafie, przed wyborem zapoznaliśmy się z liczbą wierzchołków i czasem jaki możemy poświęcić na wykonanie programu. Przykładowo mając za zadanie obliczeniu najkrótszego cyklu

dla grafu 20-wierzchołkowego wzięlibyśmy pod uwagę poniższe dane:

Algorytm	długość	czas[s]
2-opt, 100 iteracji	1496	0.001
Kruskal	919	0.008
2-opt, 1000000 iteracji	746	1.797

Dysponując dłuższym czasem lepiej skorzystać z algorytmu 2-opt dla 1000000 iteracji. W przypadku kiedy chcemy wynik otrzymać szybciej należy skorzystać z algorytmu Kruskala, który daje w tej sytuacji wynik dwa razy gorszy, ale w czasie ponad 1000 razy krótszym.

- Dużą wartość dla powyższych rozważań mogłaby mieć analiza wpływu czynnika k w strategii przeszukiwania rozwiązań. W porównaniu do obecnej implementacji wymagało by to możliwości parametryzacji liczby krawędzi w algorytmie k-opt zamiast przyjęcia stałej (w zaimplementowanym rozwiązaniu: 2). Z pewnością wydłużyłoby to czas działania pojedynczego przebiegu, ale mogłoby dawać dla niego lepsze wyniki wobec czego do znalezienia odpowiednio krótkiego cyklu mogłaby być potrzebna mniejsza Liczba iteracji.

Literatura

- [1] *Politechnika Warszawska, Wydział Matematyki i Nauk Informacyjnych*
<http://www.mini.pw.edu.pl/~januszwa/zad7.pdf>
- [2] *Politechnika Warszawska, Wydział Mechatroniki, Instytut Automatyki i Robotyki*
http://iair.mchtr.pw.edu.pl/~bputz/aisd_cpp/lekcja7/segment5/main.htm