

# Projekt z przedmiotu GIS - sprawozdanie II

Marek Jasiński, Przemysław Piotrowski

4 kwietnia 2013

## 1 Treść zadania

**Temat zadania** Znajdowanie zbioru najkrótszych ścieżek pomiędzy parą wybranych lub wylosowanych wierzchołków grafu przy awaryjnych krawędziach.

**Opis zadania** Muszą być spełnione następujące warunki: a) pierwsza wygenerowana ścieżka: najkrótsza ścieżka pomiędzy wybraną parą wierzchołków b) każda następna wygenerowana ścieżka posiada krawędź różną od co najmniej jednej krawędzi ścieżki z punktu a) i minimalną długość. Krok b) jest powtarzany kolejno aż do wyczerpania wszystkich krawędzi ścieżki z punktu a) Dane do projektu:  $G=(V,E)$  - graf o zadanej strukturze, wylosowana lub wybrana para wierzchołków. Zastosowanie: wyznaczanie ścieżek rezerwowych w sieci w sytuacji awarii pojedynczej krawędzi sieci. Literatura: N.Christofides, Graph Theory - an algorithmic approach, Academic Press 1975, str. 150- 157, 167-170. M.Sysło, N.Deo, J.Kowalik, Algorytmy optymalizacji dyskretnej, PWN 1995. N.Deo, Teoria grafów i jej zastosowania w technice i informatyce. PWN 1980.

## 2 Algorytmy

### 2.1 Wyszukiwanie alternatywnych ścieżek

Graf jest wewnętrznie reprezentowany w postaci listy sąsiedztwa, której implementację dostarcza biblioteka Boost Graph [2]. Pomimo, że biblioteka dostarcza implementację algorytmu znajdującego najkrótszą ścieżkę pomiędzy dwoma wierzchołkami, zaimplementowaliśmy własny algorytm oparty na algorytmie Dijkstry. Ogólny algorytm polega na znalezieniu najkrótszej ścieżki pomiędzy wierzchołkami źródłowym i docelowym oraz wykorzystaniu jej do

dalszego przetwarzania. Kolejne krawędzie z tej ścieżki są tymczasowo usuwane, aby uruchamiać algorytm znajdowania najkrótszej ścieżki, który dostarcza alternatywną ścieżkę dla awaryjnej krawędzi. Warto zwrócić uwagę na to, że mogą istnieć krawędzie, których usunięcie spowoduje, że graf przestanie być spójny. Takie krawędzie oznaczane są kolorem czerwonym, jako krytyczne. Nie ma alternatywnych ścieżek w grafie dla krawędzi krytycznych.

Złożoność algorytmu Dijkstry zależy od liczby wierzchołków  $V$  i krawędzi  $E$ . O rzędzie tej wielkości decyduje sposób implementacji kolejki priorytetowej wykorzystywanej do pobierania wierzchołka o najmniejszym dystansie. W przypadku naiwnej implementacji poprzez tablice złożoność ta wynosi  $O(V^2)$ . W naszym programie zastosowaliśmy implementację kolejki opartą na stercie. W tym przypadku złożoność wynosi  $O(E \log V)$ . Wariant ten jest znacznie lepszy dla grafów rzadkich, którymi są sieci małego świata.

Rozpatrywany problem ma złożoność  $O(P_{avg} * O_{Dijkstra})$ , gdzie  $P_{avg}$  to średnia długość ścieżki w grafie, natomiast  $O_{Dijkstra}$  to złożoność algorytmu Dijkstry. Wynika to z podejścia widocznego w poniższym pseudokodzie.

**Extern** *graph* #struktura grafu

**Extern** *s, t* #wierzchołki startowy i końcowy

```

emergencyPaths ← [ ]
path ← shortestPath(s, t, graph)
for edge e in path do
    graph.removeEdge(e)
    emergencyPath ← shortestPath(s, t, graph)
    emergencyPaths.append( pair(e, emergencyPath) )
    graph.addEdge(e)
end for
return path, emergencyPaths

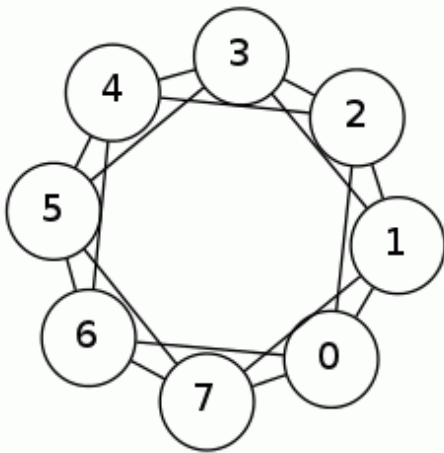
```

Jak widać pętla wykonuje się średnio tyle razy, ile wynosi średnia długość ścieżki w grafie. W pętli najbardziej czasochłonną procedurą jest znalezienie najkrótszej ścieżki (funkcja *shortestPath*). Jej złożoność zależy od złożoności obliczeniowej algorytmu Dijkstry oraz odczytania ścieżki od  $t$  do  $s$ , czyli wynosi  $f = O_{Dijkstra} + P_{avg}$ . Drugi człon ma jednak wolniejszy stopień wzrostu, w związku z czym można przyjąć  $O_{shortestPath} = O(f) = O(O_{Dijkstra} + P_{avg}) = O(O_{Dijkstra}) = O(E \log V)$ . W związku z powyższym złożoność algorytmu to  $O_{alg} = O(P_{avg} E \log V)$ .

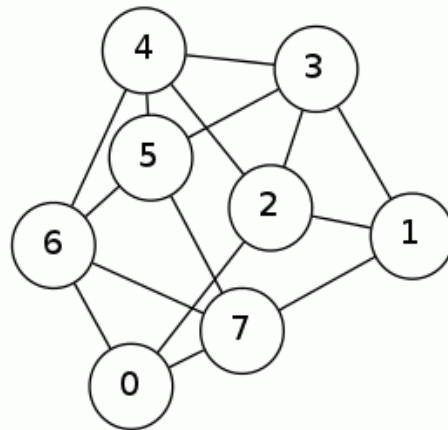
## 2.2 Generowanie grafów

Wygenerowane grafy odwzorowują sieci małego świata[4]. Oznacza to, że większość wierzchołków nie sąsiaduje ze sobą, ale pomiędzy większością par wierzchołków można znaleźć ścieżkę składającą się z małej liczby krawędzi. Sieci takie często obserwowane są np. w sieciach społecznościowych. Krawędź oznacza wtedy, że dwie osoby się znają.

Tworzenie przykładowego grafu małego świata składa się z dwóch kroków. Pierwszym z nich jest stworzenie grafu o regularnej strukturze. Przykład zaprezentowany jest na Rys. 1. Następnie należy dla każdej z krawędzi wylosować, czy powinna ona zostać przepięta, czy też nienaruszona. W efekcie powstaje graf jak na Rys. 2. Dla większej liczby wierzchołków graf widocznie nabiera cech „małego świata”. Widoczne jest to na Rys. 3.



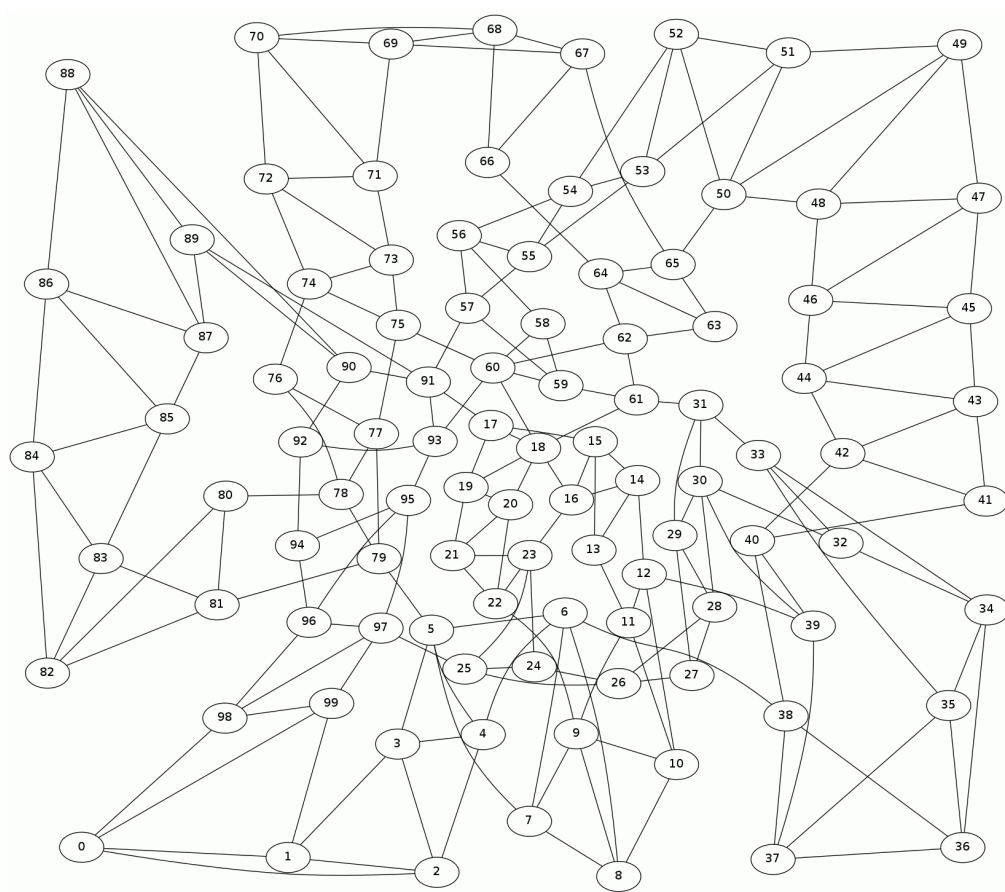
Rysunek 1: *Graf o regularnej strukturze*



Rysunek 2: *Graf po przemieszaniu krawędzi*

## 3 Programy

Na potrzeby projektu zrealizowaliśmy dwa programy. Pierwszy z nich to właściwy program realizujący algorytm wyszukiwania najkrótszej ścieżki w grafie oraz ścieżek alternatywnych. Drugi jest generatorem grafów mających cechy „małego świata”. Nazwaliśmy je odpowiednio *ap* od *Alternate Path* oraz *gg* od *Graph Generator*. Obydwa programy zostały napisane w języku C++ i wykorzystują bibliotekę Boost Graph [2] do wewnętrznej reprezentacji grafu w postaci listy sąsiedztwa. Oprogramowanie tworzyliśmy i uruchamialiśmy na dwóch platformach: Linux oraz Mac OS X. W skład projektu wchodzi



Rysunek 3: *Duży graf wygenerowany przez generator. Posiada cechy małego świata.*

plik *Makefile* służący do budowy projektu. Po wydaniu polecenia *make* w głównym katalogu projektu następuje kompilacja plików źródłowych, której rezultatem są pliki wykonywalne *ap* oraz *gg*.

### 3.1 Właściwy program

Program przyjmuje na wejściu graf oraz dwa wierzchołki: źródłowy i docelowy. Graf musi być grafem nieskierowanym z ważonymi krawędziami (wagi nieujemne) bez krawędzi równoległych. Graf jest zadawany w pliku tekstowym. Format tego pliku jest zgodny z formatem DOT [1]. Parametrami programu są:

- nazwa pliku tekstowego zawierającego opis grafu w formacie DOT,
- nazwa wierzchołka źródłowego (ang. *source*) i nazwa wierzchołka docelowego (ang. *target*),
- nazwa katalogu na pliki wynikowe.

Przykładowe wywołanie programu może wyglądać w ten sposób:

```
$ ./ap graph.dot -s w1 -t w5 -o output
```

Na skutek takiego wywołania program odczytuje graf z pliku *graph.dot*, znajduje najkrótszą ścieżkę pomiędzy wierzchołkami *w1* i *w5* oraz alternatywne ścieżki, a pliki będące wynikiem działania zapisuje w katalogu *output*. Program wspiera również dodatkowe opcje, które były użyteczne podczas testów i eksperymentów:

- **--no-report** – powoduje, że nie jest generowany plik raportu, o którym mowa dalej,
- **--no-html** – powoduje, że nie są generowane pliki HTML wizualizujące rezultat działania programu.

Dostępne opcje programu wraz z krótkim opisem można przejrzeć po uruchomieniu programu z parametrem **--help**:

```
$ ./ap --help
Usage: ap [options]
Options::
  -h [ --help ]          Produce help message
  -f [ --file ] arg      Input graph filename (obligatory)
```

```

-s [ --source ] arg    Source vertex name (obligatory)
-t [ --target ] arg    Target vertex name (obligatory)
-o [ --outdir ] arg    Output directory name (obligatory)
--no-report             If specified, report won't be generated
--no-html               If specified, HTML won't be generated

```

Przykładowy plik z zadaniem grafem wejściowym w formacie DOT może wyglądać w ten sposób:

```

/* graph.dot */
graph G {
w1 -- w2 [weight=1];
w2 -- w3 [weight=1];
w1 -- w4 [weight=1];
w4 -- w2 [weight=1];
w2 -- w5 [weight=1];
w5 -- w3 [weight=1];
}

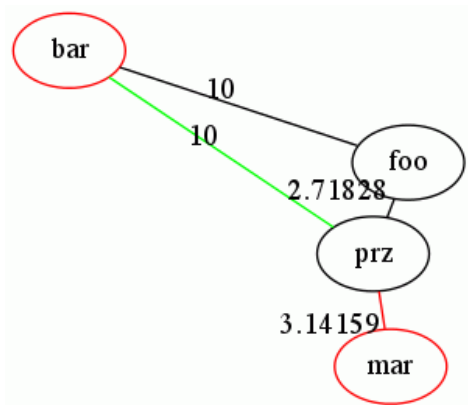
```

Plik opisujący graf może być przygotowany ręcznie, zostać pobrany z Internetu, a także może być wygenerowany przy pomocy naszego drugiego programu, tj. generatora grafów.

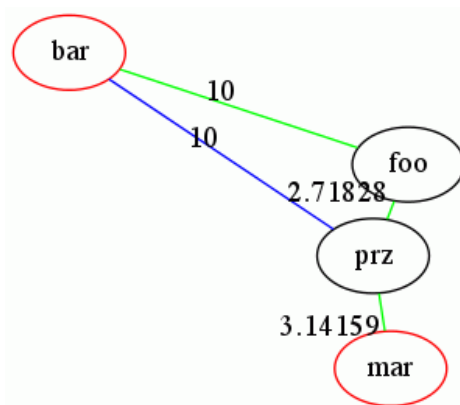
Wizualizację rezultatu działania programu zapewnia program NEATO z pakietu Graphviz [3]. Służy on do rysowania grafów nieskierowanych. Program NEATO jest wywoływany z poziomu naszego programu. Za pomocą programu NEATO generowane są pliki, na podstawie których nasz program generuje strony HTML. Na stronach można zobaczyć, jak wyglądają najkrótsza ścieżka pomiędzy wierzchołkami źródłowym i docelowym oraz najkrótsze ścieżki alternatywne (po kliknięciu na krawędź, którą uznaje się za awaryjną). Przykładowa zawartość stron może wyglądać jak na Rys. 4 i 5.

Na Rys. 4 widzimy najkrótszą ścieżkę pomiędzy wierzchołkami **mar** a **bar**. Została ona oznaczona na kolorowo. Składa się z dwóch krawędzi. Na wypadek awarii krawędzi zielonej istnieje ścieżka alternatywna, która jest przedstawiona na Rys. 5. Czerwona krawędź jest krytyczna - nie ma dla niej alternatywnych ścieżek. Usunięcie tej krawędzi spowodowałoby, że nie byłoby ścieżki pomiędzy wierzchołkami **mar** a **bar**.

Drugim rezultatem działania programu jest plik `report.txt` zawierający informacje o najkrótszej ścieżce oraz ścieżkach alternatywnych. Przy każdej ze ścieżek jest informacja o koszcie (sumie wag krawędzi) i liczbie krawędzi (liczbie przeskoków). Plik raportu dla powyższego przykładu wygląda w ten sposób:



Rysunek 4: Główny obrazek (na stronie *index.html*)



Rysunek 5: Po kliknięciu na zieloną krawędź na głównym obrazku

Shortest path: mar--prz--bar Cost: 13.1416 Hops: 2  
mar--prz | No emergency path  
prz--bar | mar--prz--foo--bar Cost: 15.8599 Hops: 3

## 3.2 Generator grafów

Danymi wyjściowymi generatora są grafy zapisane w formacie DOT. Parametrami generatora są:

- plik wyjściowy,
- liczba wierzchołków,
- liczba krawędzi prowadzonych z każdego z wierzchołków do następnych,
- prawdopodobieństwo przełączenia krawędzi.

Przykładowe wywołanie:

```
$ ./gg graph.dot -n 30 -e 2 -p 20
```

Powoduje wygenerowanie do pliku *graph.dot* grafu o 30 wierzchołkach, w którym z każdego wierzchołka prowadzone są dwie krawędzie do dwóch kolejnych wierzchołków, a krawędzie są losowo przełączone z prawdopodobieństwem  $\frac{20}{100}$ .

Dostępne opcje programu wraz z krótkim opisem można przejrzeć po uruchomieniu programu z parametrem **--help**:

```

$ ./gg --help
Usage: ./gg [options]
Options::
  -h [ --help ]          Produce help message
  -o [ --output ] arg    Result graph filename (obligatory)
  -n [ --num ] arg       Number of vertexes (obligatory)
  -e [ --edges ] arg     Number of edges (obligatory)
  -p [ --prob ] arg      Probability, that edge will be moved (obligatory)

```

## 4 Testy i eksperymenty

Złożoność obliczeniowa programu została zbadana na wygenerowanych grafach pochodzących z generatora grafów. W celu zbadania złożoności algorytmu napisany został skrypt w języku Python, którego nazwa to *analyze\_algorithm.py*. Na początku swojego działania generuje on graf za pomocą generatora używając parametrów grafu z pliku *config.py*. Następnie określoną liczbę razy losuje wierzchołki źródłowy i docelowy, aby zmierzyć czas działania właściwego programu.

Skrypt zapisuje:

- czasy działania programu wraz z długością najkrótszej ścieżki pomiędzy wylosowanymi wierzchołkami,
- 

Przykładowy wynik działania prezentuje Rys. 6.

	A	B	C
1	czas wykonania	długość ścieżki	
2	0.478909	1.0	
3	0.492453	1.0	
4	0.699502	2.0	
5	0.708394	2.0	
6	0.712314	2.0	
7	0.716741	2.0	
8	0.935963	3.0	
9	0.944184	3.0	
10	0.945143	3.0	
11	0.953695	3.0	
12	0.956594	3.0	
13	1.159218	4.0	

Rysunek 6: Zestawienie wyników



Dzięki takiemu zestawieniu można sprawdzić, czy złożoność teoretyczna algorytmu została wyznaczona poprawnie.

## 5 Bibliografia

- [1] „The DOT Language”  
<http://www.graphviz.org/doc/info/lang.html>
- [2] „The Boost Graph Library”  
[http://www.boost.org/doc/libs/1\\_53\\_0/libs/graph/doc/index.html](http://www.boost.org/doc/libs/1_53_0/libs/graph/doc/index.html)
- [3] „Graphviz - Graph Visualization Software”  
<http://www.graphviz.org/>
- [4] „Classes of small-world networks”, Amaral, Luis A Nunes and Scala, Antonio and Barthélemy, Marc and Stanley, H Eugene