

# Teoria grafów - projekt

## Dokumentacja programu

### Algorytm Bellmana-Forda

Program został napisany w języku C++,  
w środowisku Visual Studio Code,  
na kompilatorze MinGW,  
na systemie Windows 10.

Działa poprawnie testowany dodatkowo na  
maszynie wirtualnej,  
na kompilatorze GNU g++,  
na systemie Kubuntu x64.

Program prezentujący algorytm znajduje się w pliku  
[proj/proj.cpp](#), opisanym w sekcji *proj.cpp* poniżej.  
W tej samej sekcji znajduje się instrukcja jego uruchomienia.








Metoda algorytmu Bellmana-Forda jest opisana w sekcji  
*graph.cpp* poniżej.

# Dokumentacja programu

W folderze `/proj` znajdują się:

- plik `proj.cpp`, zawierający właściwy program do skompilowania i uruchomienia,
- plik `graph.cpp`, zawierający autorskie klasy i metody z których korzystam do przechowywania i operacji na grafach (w tym metodę `bellmanFord()`, będącą funkcją z polecenia),
- plik `convert.cpp`, zawierający funkcje służące do czytania i zapisywania grafów do pliku,
- trzy foldery zawierające po jednym grafie - w postaci obrazka `.png`, oraz opisane w pliku `.txt`,
- plik `readme.txt`, zawierający tę dokumentację.

Każdy z tych plików postaram się krótko opisać aby wyjaśnić jaka jest ich rola.

	graf1	22.05.2021 20:34	Folder plików	
	graf2	22.05.2021 15:50	Folder plików	
	graf3	22.05.2021 20:23	Folder plików	
	convert	22.05.2021 20:08	Plik CPP	6 KB
	graph	22.05.2021 20:27	Plik CPP	10 KB
	proj	22.05.2021 19:58	Plik CPP	3 KB
	readme	22.05.2021 20:32	Dokument tekstowy	11 KB

## Foldery

Trzy foldery zawierający trzy przykładowe grafy.

`/graf1` - graf skierowany, opisany rozszerzoną listą sąsiedztw w pliku `graf1/graf1.txt`

`/graf2` - graf skierowany, opisany własnym formatem w pliku `graf2/graf2.txt`

`/graf3` - graf skierowany, opisany listą sąsiedztw w pliku `graf3/graf3.txt`

Nowe dane do przekazania do programu można zapisać w pliku `.txt` w jednym z dostępnym formatów. Następnie należy przekazać jego ścieżkę do odpowiednich funkcji, które zamienią go na string, a potem zbudują graf.

Właściwy program do skompilowania i uruchomienia.

Składa się z plików `graph.cpp`, `convert.cpp`, oraz korzysta z grafów w folderach jako dane wejściowe.

Program należy skompilować i uruchomić tak, aby ścieżka lokalna zaczynała się z folderu w którym znajdują się pozostałe pliki. W przeciwnym wypadku potrzebne będzie zamienienie ścieżek w programie na globalne, łącznie z tymi przy operatorach `#include`.

Uprzedzam na wszelki wypadek, tylko dlatego że VSCode, w którym program był tworzony, zamiast tego wybiera folder kompilatora; możliwe że z innymi środowiskami nie będzie tego problemu. Uruchamianie z poziomu konsoli w folderze `/proj` też działa poprawnie.

Program można oczywiście edytować; ten napisany jest przygotowany tak, aby prezentował działanie algorytmu po jedynie skompilowaniu i uruchomieniu. Jednocześnie służy jako program przykładowy. Sam w sobie nie przyjmuje argumentów, jego działanie można zmienić w funkcji `main()`. Nowe dane można zapisać w pliku `.txt` i przekazać ich ścieżkę jako argumenty odpowiednich funkcji wewnątrz programu. Powinny one być poprawnie zbudowane - program jest odporny na część, ale nie wszystkie możliwe błędne dane wejściowe.

Przygotowany program przykładowy prezentuje działanie algorytmu Bellmana-Forda. Po poprawnym skompilowaniu i uruchomieniu w tym samym folderze wykonuje następujące działania:

- czyta i buduje graf z pliku `graf1/graf1.txt`,
- wykonuje na nim algorytm Bellmana-Forda (metoda `bellmanFord()`),
- wypisuje wynik w konsoli w postaci dwóch list, tak jak ćwiczony był on na zajęciach – listy rodziców oraz listy całkowitego dystansu od źródła.
- zapisuje drzewo minimalnych ścieżek znalezione przez algorytm w pliku `graf1/graf1_tree_str.txt` w moim własnym formacie (opisanym niżej w sekcji `convert.cpp`),
- zapisuje to samo drzewo w pliku `graf1/graf1_tree_list.txt` w postaci listy sąsiedztw (bez wag),
- drukuje to samo drzewo w pliku `graf1/graf1_tree_print.txt` za pomocą metody `print()` w przystępnym do analizy formacie.

Każda czynność jest dokładnie zaznaczona wewnątrz pliku.

# graph.cpp

Własne, autorskie klasy i metody z których korzystam do przechowywania i operacji na grafach. Metody te zacząłem pisać samemu w ramach oswojenia się z obiektowością języka C++ przed pracą na projektem, stąd taki a nie inny sposób przechowywania grafów.

Grafy tworzone są na podstawie stringa o własnym formacie (opisanym w sekcji *convert.cpp*).

Opis klas i ważniejszych metod:

## Node

Klasa węzła - zawiera podstawowe informacje o węźle

<b>index</b>	- numer węzła; $\geq 0$
<b>count</b>	- liczba węzłów sąsiadujących; $\geq 0$
<b>neighbours</b>	- tablica wskaźników na węzły sąsiadujące

## Edge

Klasa krawędzi - zawiera podstawowe informacje o krawędzi

<b>start</b>	- wskaźnik na węzeł początkowy
<b>end</b>	- wskaźnik na węzeł końcowy
<b>weight</b>	- waga krawędzi; liczba całkowita
<b>is_directed</b>	- informacja, czy krawędź jest skierowana; jeśli nie jest, <b>start</b> i <b>end</b> są wymienne

## Graph

Klasa przechowująca węzły i krawędzie grafu, posiadająca metody do pracy z grafem

<b>name</b>	- nazwa grafu; czysto kosmetyczna
<b>is_directed</b>	- informacja, czy graf jest skierowany
<b>node_count</b>	- liczba węzłów grafu
<b>edge_count</b>	- liczba krawędzi grafu
<b>nodes</b>	- tablica wskaźników na węzły
<b>edges</b>	- tablica wskaźników na krawędzie

# graph.cpp

## `print(ostream)`

Wypisuje graf na **ostream** w postaci łatwej do czytania i analizy

## `randomizeWeights(a, b)`

Przypisuje krawędziom losowe wagi całkowite z przedziału  $\langle a, b \rangle$

Gdy **b** jest ominięte, obiera przedział  $\langle 0, |a| \rangle$

Gdy **a** oraz **b** są ominięte, obiera przedział  $\langle 0, 10 \rangle$

Każda krawędź zawsze zawiera wagę; ta funkcja nadpisuje te wartości

Przydatna dla grafów tworzonych z listy sąsiedztw, gdzie każda waga jest równa **1** (opisane w sekcji *convert.cpp*)

## `setWeights(input, size)`

Złożony setter wag każdej krawędzi grafu. Przyjmuje listę liczb całkowitych **input**, i stara się je przypisać jako wagi do każdej krawędzi w kolejności jej występowania w tablicy krawędzi

## `strToGraph(input, inname)`

Tworzy nowy graf ze stringa mojego własnego formatu o nazwie **inname** (format ten jest również opisany w sekcji *convert.cpp*)

## `graphToStr()`

Zwraca string opisujący dany graf moim własnym formatem

## graph.cpp

### `bellmanFord(start, want_str)`

Wykonuje algorytm Bellmana-Forda na grafie. Zwraca string w wybranym formacie.

- `start` - numer węzła startowego dla algorytmu
- `want_str` - informacja, czy algorytm ma zwrócić wynik w postaci dwóch tablic (**true** lub pominięte), czy w postaci grafu drzewa rozpinającego, w formacie własnym (**false**)

Algorytm zakłada, że graf jest spójny - w przeciwnym razie wynik jest nieokreślony.

Algorytm wspiera wagi ujemne.

Algorytm wykrywa cykle ujemne i informuje o ich obecności – wykonuje dodatkową iterację, która powinna być zbędna. W razie gdy okaże się, że podczas tej iteracji zaszła zmiana w zbudowanej liście rodziców i wag, oznacza to że w grafie musi znajdować się cykl ujemny, który sprawia że algorytm nie znajduje właściwego wyniku. Nadal zwraca on znalezione ścieżki, ale jednocześnie informuje na wyjściu błędów o ujemnym cyklu.

Wynik algorytmu jest typu string - można go zwrócić na konsolę lub zapisać w pliku (zademonstrowane w pliku `proj.cpp`), aby przeanalizować jego poprawność.

## convert.cpp

Odrębne funkcje, stworzone specjalnie do czytania i zapisywania grafów do pliku. Obsługują trzy formaty:

- lista sąsiedztw; bez wag, zawsze skierowany
- rozszerzona lista sąsiedztw; z wagami, zawsze skierowany
- własny format; z wagami

### Lista sąsiedztw

Zwykła lista sąsiedztw – nie zawiera informacji o wagach krawędzi, więc każdej jest przypisywana waga 1. Każdy graf opisany listą sąsiedztw jest skierowany, nawet jeśli krawędzie powtarzają się w obu węzłach.

Ostatnia linia opisująca ostatni węzeł **nie może** zawierać znaku końca linii – tworzy on nową pustą linię, rozumianą jako kolejny węzeł bez sąsiadów.

Przykładowy graf opisany tym formatem znajduje się w folderze [/graf3](#).

### Rozszerzona lista sąsiedztw

Zawiera dodatkowo informacje o wagach krawędzi. W pierwszej linii pliku znajduje się tablica wag wszystkich krawędzi w kolejności występowania.

Przykładowy graf opisany tym formatem znajduje się w folderze [/graf1](#).

### Własny format

Opisuje graf - jego liczbę węzłów, liczbę krawędzi, każdą krawędź wraz z jej wagą, oraz to czy jest skierowany.

Każdy graf przedstawiony tym formatem wygląda następująco:

```
string str =  
    "a b c\n"  
    "p>qwr\n"  
    "p>qwr\n"  
    .  
    .  
    .  
    "p>qwr\n"  
    "p>qwr";
```

**a** - informacja czy graf jest skierowany; "0" - nie,  
w przeciwnym razie tak  
**b** - liczba węzłów w grafie;  $\geq 0$   
**c** - liczba krawędzi w grafie;  $\geq 0$   
"p>qwr\n" - opis jednej krawędzi, gdzie:  
**p** - numer węzła startowego;  $\in <0, b)$   
**q** - numer węzła końcowego;  $\in <0, b)$   
**r** - waga krawędzi; liczba całkowita

W kodzie, nazwach funkcji, nazywany jako 'str'.

Przykładowy graf opisany tym formatem znajduje się w folderze [/graf2](#).

## convert.cpp

Każda z funkcji zakłada poprawność formatu pliku/stringa wejściowego - w przeciwnym razie wynik jest nieokreślony.

### readStr(name)

Czyta plik **name** zawierający opis grafu w formacie własnym i zwraca string w formacie własnym

### readList(name)

Czyta plik **name** zawierający opis grafu w formacie listy sąsiedztw i zwraca string w formacie własnym

### readListWeighted(name)

Czyta plik **name** zawierający opis grafu w formacie rozszerzonej listy sąsiedztw i zwraca string w formacie własnym

Jeśli liczba wag w pierwszej linii różni się od liczby opisanych krawędzi, funkcja zatrzymuje działanie w momencie gdy jednej z dwóch danych zabraknie i powiadamia o problemie na wyjściu błędów.

### saveStr(name, input)

Przyjmuje graf w postaci stringa **input** w formacie własnym i zapisuje do pliku **name** w postaci własnego formatu

### saveList(name, input)

Przyjmuje graf w postaci stringa **input** w formacie własnym i zapisuje do pliku **name** w postaci listy sąsiedztw

### saveListWeighted(name, input)

Przyjmuje graf w postaci stringa **input** w formacie własnym i zapisuje do pliku **name** w postaci rozszerzonej listy sąsiedztw



# Podstawowy schemat funkcji

