Projekt nr 16

Porównanie implementacji algorytmu Forda Fulkersona w językach programowania C/C++ i C#

Autorzy:

Petrykowski Daniel

Wojciechowski Grzegorz

2019

Sprawozdanie nr 1

1. Cel projektu

Celem projektu jest zaimplementowanie algorytmu Forda Fulkersona do wyznaczania ścieżki pozwalającej na maksymalny przepływu w sieci przepływowej. Zostanie on zaimplementowany w C/C++ i C#, a następnie zostanie zbadany czas trwania obliczeń dla tych dwóch implementacji przy takich samych parametrach wejściowych.

Sprawozdanie nr 2

2. Opis algorytmu Forda Fulkersona

Algorytm Forda Fulkersona pozwala na wyznaczenie maksymalnego przepływu w sieci przepływowej. Jednak, że przed przedstawieniem zasady działania algorytmu należy wprowadzić potrzebne pojęcia:

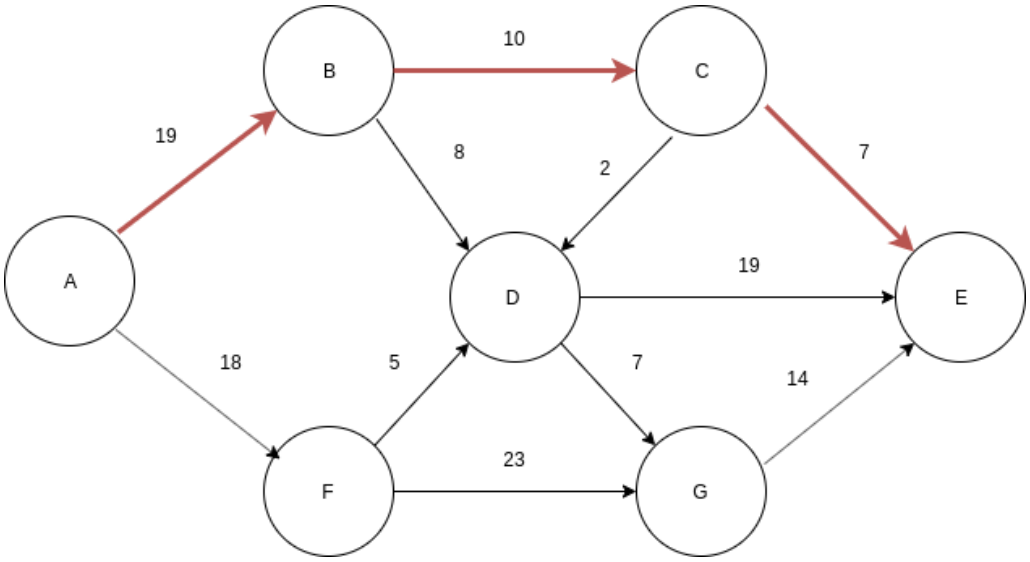
a) sieć przepływowa – jest grafem skierowanym G = (V, E) w którym przepływ odbywa się w kierunku wyznaczonym przez zwrot krawędzi grafu, każda krawędź jest skojarzona z parametrem określającym jej przepustowość.
b) przepustowość - jest definiowana jako funkcja c(u,v), gdzie u i v są wierzchołkami. Jeżeli pomiędzy dwoma wierzchołkami istnieje krawędź to wartość przepustowość będzie wynosić c(u,v) >= 0, jeżeli taka krawędź nie istnieje to wartość przepustowości będzie równa c(u,v) = 0.
c) źródło - wierzchołek s, który jest źródłem z którego zaczyna się przepływ
d) ujście - wierzchołek t będący ujściem przepływu
e) przepływ f(u,v) - określa jaki przepływ odbywa się w kanale. Przy czym przepływ pomiędzy dwoma węzłami sieci nie może być większy niż wartość przepustowości pomiędzy nimi.
f) maksymalna przepustowość sieci pomiędzy źródłem i ujściem - jest ona określona przez funkcje:

$$|f| = \sum_{v \in V} f(s, v)$$

Czyli suma przepływów ze źródła s do wszystkich pozostałych wierzchołków sieci. (1)

g) sieć rezydualna – jest grafem indukowanym z pierwotnej sieci przepływu. Nowo powstała sieć rezydualna posiada tą samą liczbę wierzchołków i krawędzie określające przepustowość rezydualną pomiędzy nimi.
h) przepustowość rezydualna - jest to różnica przepustowości i przepływu w sieci przepływowej. (1)

$$c_f(u, v) = c(u, v) - f(u, v)$$



Rysunek 1 Przepustowość rezydualna

Dla zaznaczonej ścieżki powyższego grafu przepustowość rezydualna jest równa najmniejszej przepustowości rezydualnej jej poszczególnych kanałów czyli 7.

i) ścieżka rozszerzająca – ścieżka w sieci rezydualnej łącząca źródło z ujściem, przy czym wszystkie kanały leżące na ścieżce muszą mieć wartości przepustowości rezydualnej nie zerową.
j) przepustowość rezydualna – jest to wartość równa najmniejszej przepustowości rezydualnej kanałów leżących na ścieżce rozszerzającej

$$c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$$

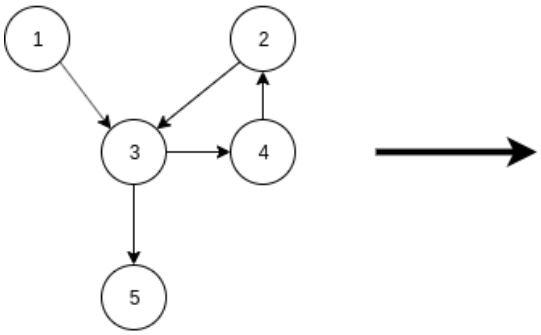
Idea algorytmu opiera się na iteracyjnym zwiększaniu przepływu, zaczynając od zerowego przepływu. Podstawowa wersja brzmi następująco:

1) Wyzeruj wszystkie przepływy w sieci
2) Wyznacz sieć rezydualną dla wejściowej sieci przepływów
3) Dopóki w sieci istnieje ścieżka rozszerzająca p, zwiększ przepływ o przepustowość rezydualną znalezionej ścieżki c_f wzdłuż kanałów zgodnych z kierunkiem ścieżki, a zmniejsz przepływ wzdłuż kanałów przeciwnych (2)

Jak wynika z powyższego opisu algorytm Forda Fulkersona jest bardziej metodą postepowania, niż szczegółowym opisem implementacji. Dlatego też, aby go wykorzystać należy również rozwiązać problem dotyczący sposobu w jaki sieć będzie reprezentowana w pamięci komputera oraz drugi problem jakim jest wyszukiwanie ścieżek rozszerzających w sieci rezydualnej.

Pierwszy problem można rozwiązać na dwa sposoby:

1) tablica listy sąsiedztwa - jest to efektywny pamięciowo sposób reprezentacji grafu, zajmuje pamięć rzędu O(m), gdzie m oznacza liczbę krawędzi grafu. W porównaniu do macierzy sąsiedztwa sprawdzenie czy dana krawędź istnieje będzie wolniejsze, ale pozwoli za to na operowanie na większych sieciach.
2) macierz sąsiedztwa – jest to sposób który pozwala na uproszczenie obliczeń jednakże jego złożoność pamięciowa wynosi $O(m^2)$ z tego też powodu sposób ten staje się mało efektywny przy dużych sieciach. Dla przykładu poniżej znajduje się rysunek przedstawiający macierz sąsiedztwa podanego obok grafu. (3)



	1	2	3	4	5
1	0	0	1	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	0	1	0	0	0
5	0	0	0	0	0

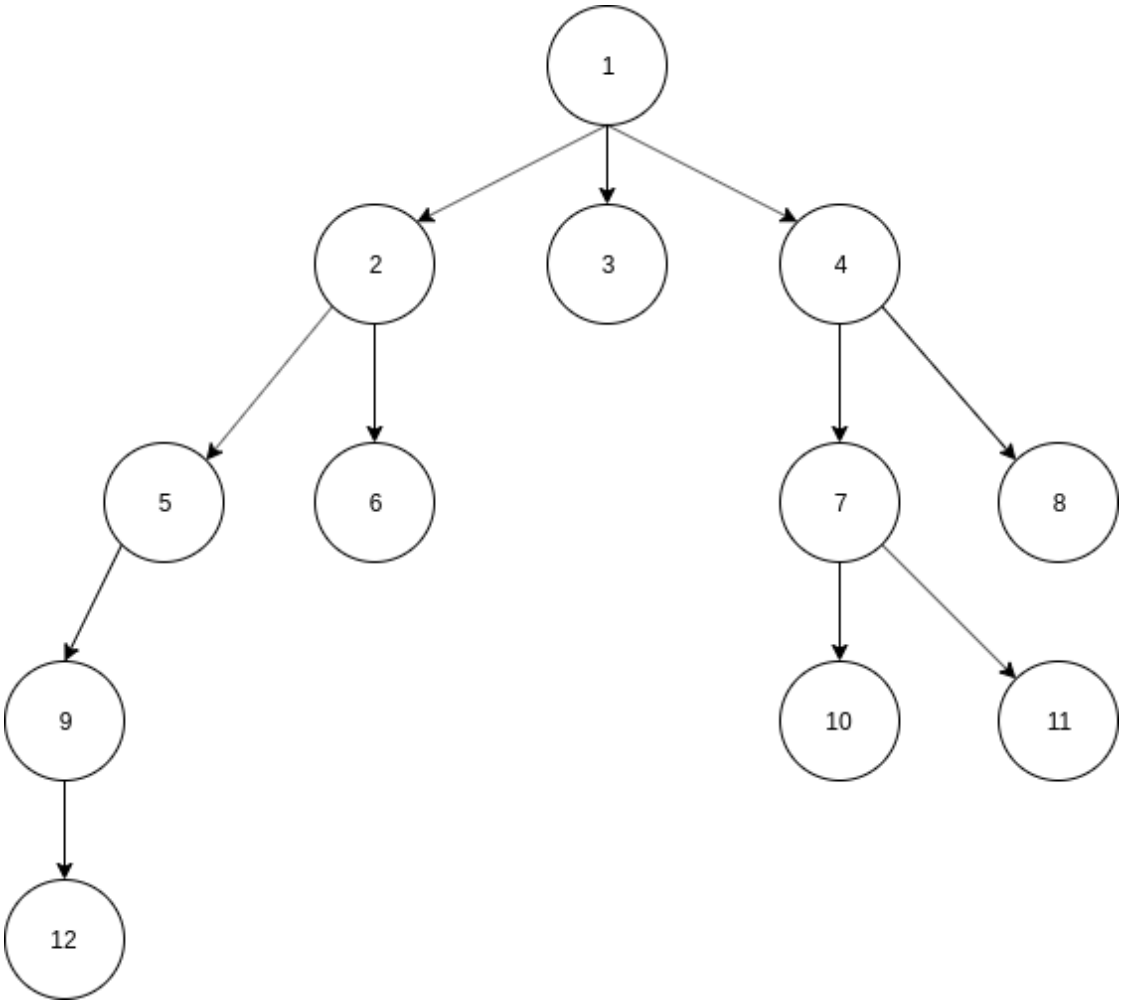
Rysunek 2 Graf skierowany i odpowiadająca mu macierz sąsiedztwa

Natomiast, aby rozwiązać przedstawiony wcześniej problem z wyznaczenia ścieżek rozszerzających algorytmu. Stosuje się najczęściej jeden z dwóch algorytmów:

1) przeszukującego graf w szerz (breadth-first search, BFS) - jest to popularne rozwiązanie pozwalające odnaleźć wszystkie połączone węzły w grafie. Kolejnym plusem jest to, że algorytm ten zawsze będzie wybierał ścieżkę o najmniejszej liczbie krawędzi.

   Algorytm zaczyna się od odwiedzenia wierzchołka startowego. Następnie odwiedza się wszystkich jego sąsiadów, a potem wszystkich nieodwiedzonych sąsiadów sąsiadów i tak iteracyjnie. Aby uniknąć zapętleń w przypadku napotkania cyklu do wierzchołka dodaje się parametr mówiący o tym, czy został on już odwiedzony. (3)

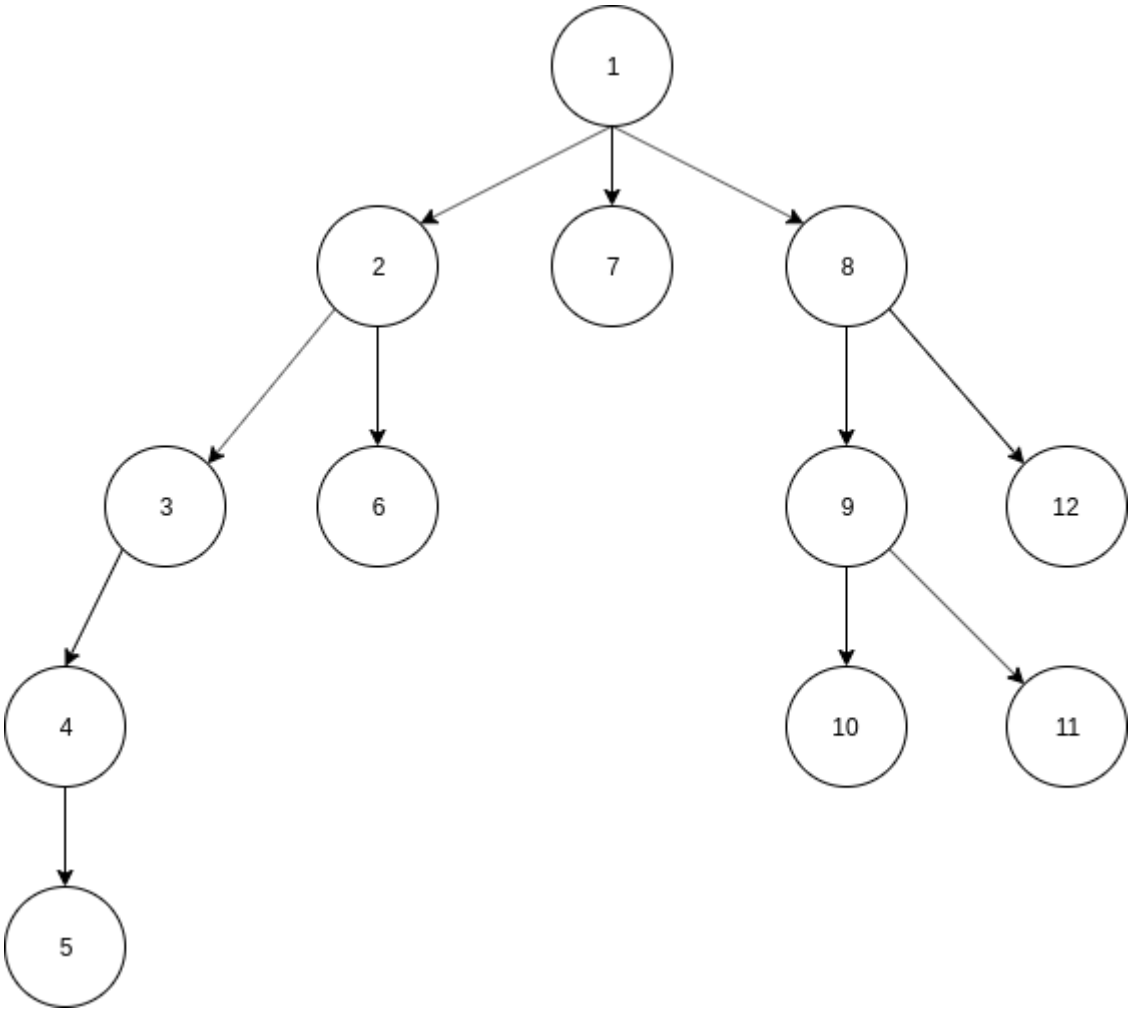
   Kolejność odwiedzania wierzchołków została przedstawiona na poniższym diagramie.



Rysunek 3 Graf przedstawiający kolejność przeszukania grafu przy wykorzystaniu algorytmu BFS

2) przeszukującego graf w głąb (Depth First Search – DFS) – jest to rozwiązanie polegające na przechodzeniu z wierzchołka startowego do jego pierwszego wierzchołka sąsiedniego, a następnie w kolejnym kroku powtarzamy tą akcje i szukamy dla bieżącego wierzchołka sąsiada i do niego przechodzimy. Algorytm kończymy, gdy zostaną odwiedzone wszystkie wierzchołki. Podobnie jak w algorytmie BFS w tym również należy monitorować które z wierzchołków zostały już odwiedzone w celu uniknięcia cykli i pętli. Jednakże, algorytm ten jest bardzo nie efektywny w wyznaczaniu ścieżek rozszerzających ponieważ ma tendencje do wyszukiwania długich ścieżek. (3)

   Kolejność odwiedzania wierzchołków została przedstawiona na poniższym diagramie.



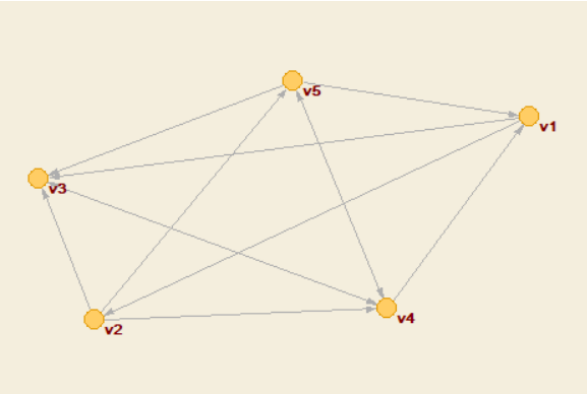
Rysunek 4 Graf przedstawiający kolejność przeszukania grafu przy wykorzystaniu algorytmu DFS

3. Założenia programu

Jak zostało to opisane w punkcie drugim w celu zaimplementowania algorytmu należy podjąć pewne decyzje.

Pierwszą z nich jest wybór sposobu reprezentacji sieci w pamięci komputera. Zdecydowaliśmy się na zastosowanie tablicy sąsiedztwa co pozwoli na operowanie na większych sieciach.

Zostanie stworzona tablica o rozmiarze równym liczbie wierzchołków, zawierająca wskaźniki na listy – kolejne elementy list będą oznaczać kolejnych sąsiadów danego wierzchołka, do którego lista jest przyporządkowana.



Lista wierzchołków	Sąsiednie wierzchołki		
v1	2	3	
v2	3	4	5
v3	4		
v4	1	3	5
v5	1	3	4

Rysunek 5 Graf skierowany wraz z odpowiadającą mu tablicą list sąsiedztwa wierzchołków

Sąsiad będzie opisywany obiektem, który przechowywał będzie informacje o numerze wierzchołka, maksymalnym przepływie i przepływie bieżącym.

Należy równie wybrać algorytm który zostanie zastosowany do wyznaczania ścieżek rozszerzających. W tym przypadku wybór padł na przeszukiwanie wszerz (breadth-first search, BFS), gdyż jak już wcześniej zostało to opisane znacznie lepiej nadaje się on do tego zadania.

4. Koncepcja programu

Aby eksperyment był miarodajny stwierdziliśmy, że obydwie implementacje powinny działać na tych samych danych wejściowych (tzn. ten sam graf). W tym celu oprócz samego algorytmu stworzyliśmy program umożliwiający wygenerowanie dużej liczby grafów o różnych parametrach.

Nasz generator sieci tworzy pozwala na stworzenie sieci o następujących parametrach:

- dowolny rozmiar sieci (możliwość konfiguracji),
- liczba krawędzi wychodzących z wierzchołka definiowana na podstawie zmiennej losowej z rozkładu normalnego o zdefiniowanych wartościach parametrów μ i σ^2
- wartość krawędzi losowana z wcześniej zdefiniowanego przedziału
- tworzony graf nie posiada pętli
- tworzony graf jest siecią przepływową

Oprócz wygenerowania grafu algorytm wybiera ze zbioru wierzchołków losowo źródło i ujście sieci przepływowej.

Tak stworzony przypadek testowy zostaje następnie zapisany w pliku txt w następującym formacie:

- pierwsza liczba = liczba wierzchołków
- druga liczba = źródło
- trzecia liczba = ujście
- zapis sieci w postaci macierzy sąsiedztwa

```
10
7
5
0;0;0;41;0;0;85;0;0;0;
0;0;0;0;0;61;4;0;25;0;
41;0;0;0;0;98;41;0;0;0;
0;83;0;0;0;76;95;57;0;0;
5;27;66;89;0;0;3;0;0;0;
0;0;0;0;45;0;0;0;0;0;
0;0;0;0;0;83;0;0;0;0;
20;0;18;0;55;0;25;0;0;0;
0;0;0;0;0;0;0;25;0;0;
0;0;67;0;0;0;0;0;0;0;
```

Kiedy zostanie już wygenerowana odpowiednia liczba przypadków testowych (na tym etapie szacujemy, że będzie to kilka tysięcy) możliwe jest przejście do właściwej części eksperymentu. Koncepcja działania naszego programu można opisać w kilku krokach:

1) Wczytaj przypadek testowy
2) Uruchom zegar
3) Wyznacz maksymalny przepływ w sieci przy wykorzystaniu algorytmu Forda Fulkersona
4) Zatrzymaj zegar
5) Zapisz czas jaki był potrzebny na wykonanie algorytmu
6) Przejdź do punktu 1 jeżeli nie jest to ostatni przypadek testowy
7) Zwróć wyniki (średni czas wykonywania algorytmu)

5. Projekt testów

Założeniem projektu jest porównanie czasu trwania obliczeń implementacji algorytmu Forda Fulkersona w językach C/C++ i C#. Dlatego też nasze testy będą skupiać się na tym celu.

Aby zapewnić miarodajność testów obydwie implementacje będą wykonywane na tym samym sprzęcie, a na ich wejście zostaną podane te same przypadki testowe.

Sam test będzie polegał na wielokrotnym wykonaniu algorytmu dla różnych grafów podczas którego będzie badany czas jego wykonywania. Zegar będzie uruchamiany przed samym uruchomieniem algorytmu, a zatrzymywany zaraz po otrzymaniu wyniku. Tak zebrane wyniki zostaną ostatecznie poddane uśrednieniu. Ważnym elementem w tym teście jest sama rozdzielczość zegara, która wynosi:

a) dla implementacji w C#

$$\frac{1}{10000000}s = 0{,}0000001s = 100ns$$

W C# pomiar czasu do celów testowych i diagnostycznych wykonuje się poprzez wykorzystanie klasy *Stopwatch* z przestrzeni nazw *System.Diagnostics*. Częstotliwość dla zegara używanego w implementacji tej klasy można odczytać poprzez odczytanie wartości właściwości *Frequency*. (4)

b) dla implementacji w C/C++

$$1ns$$

Wykorzystując chrono API z C++11. Dostępny tam zegar jest zegarem systemowym z najkrótszym możliwym okresem tyknięć. Jest to zegar z największą możliwą precyzją. (5)

std::chrono::high_resolution_clock

Na potrzeby projektu rozdzielczość pomiaru sprowadzimy do tego samego rzędu wielkości.

Sprawozdanie nr 3

6. Dokumentacja kodu
7. Wyniki testów
8. Wnioski

Bibliografia

1. **Zaawansowane algorytmy i struktury danych/Wykład 9 - Wazniak.mimuw. [Online] 2009. [Zacytowano: 14 Listopad 2019.] http://wazniak.mimuw.edu.pl/index.php?title=Zaawansowane_algorytmy_i_struktury_danych/Wyk%C5%82ad_9.**

2. TUM. Ford-Fulkerson Algorithm. *Description of the algorithm.* [Online] [Zacytowano: 08 12 2019.] https://www-m9.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_en.html.

3. mgr Wałaszek Jerzy. Algorytmy Struktury Danych. *Maksymalny przepływ w sieci – algorytmy Forda-Fulkersona i Edmondsa-Karpa.* [Online] [Zacytowano: 08 12 2019.] https://eduinf.waw.pl/inf/alg/001_search/0146.php.

4. Stopwatch Class. *Microsoft.* [Online] Microsoft. [Zacytowano: 21 11 2019.] https://docs.microsoft.com/en-gb/dotnet/api/system.diagnostics.stopwatch?view=netframework-4.8.

5. cplusplus. *<chrono>.* [Online] [Zacytowano: 08 12 2019.] http://www.cplusplus.com/reference/chrono/.

Rysunki