

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 przepływ 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 przepustowości będzie wynosić $c(u,v) \geq 0$, 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 funkcję:

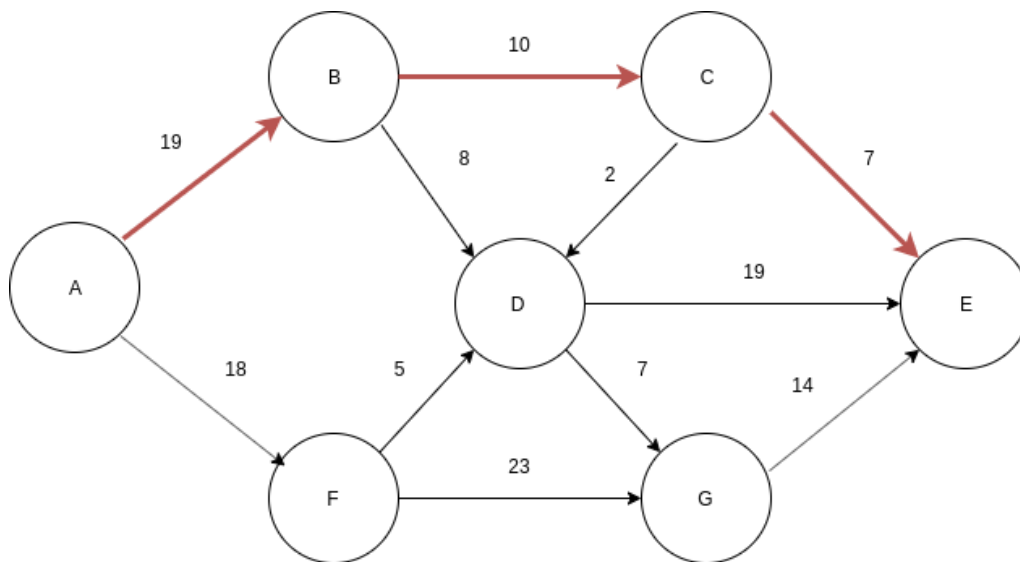
$$|f| = \sum_{v \in V} f(s, v)$$

Czyli suma przepływów ze źródła s do wszystkich pozostałych wierzchołków sieci. [CITATION Zaa09 \l 1045]

- 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. [CITATION Zaa09 \l 1045]

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



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

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

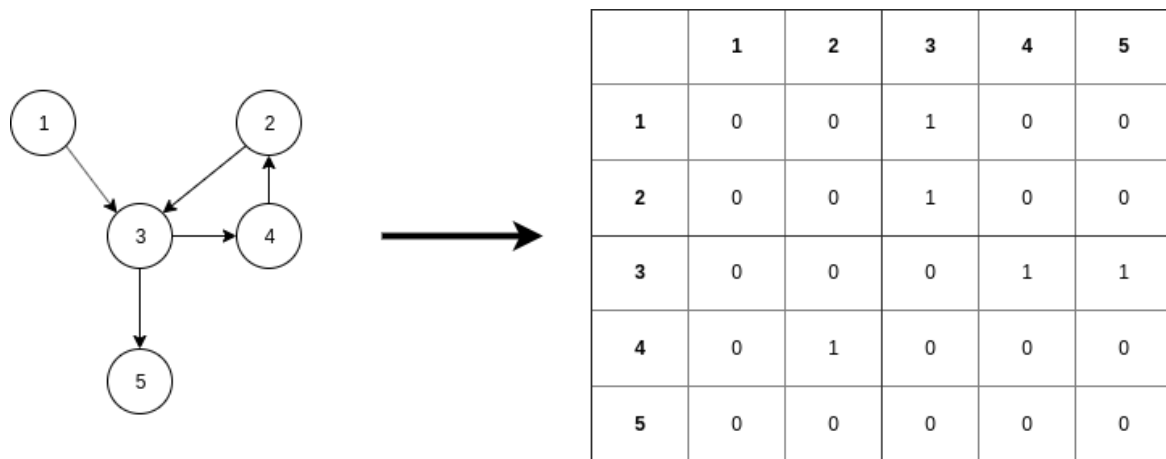
Idea algorytmu opiera się na iteracyjnym zwiększaniu przepływu, zaczynając od zerowego przepływu. Podstawowa wersja brzmi następująco:

- 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 [CITATION For19 \l 1045]

Jak wynika z powyższego opisu algorytm Forda Fulkersona jest bardziej metodą postępowania, 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. [CITATION mgr19 \l 1045]



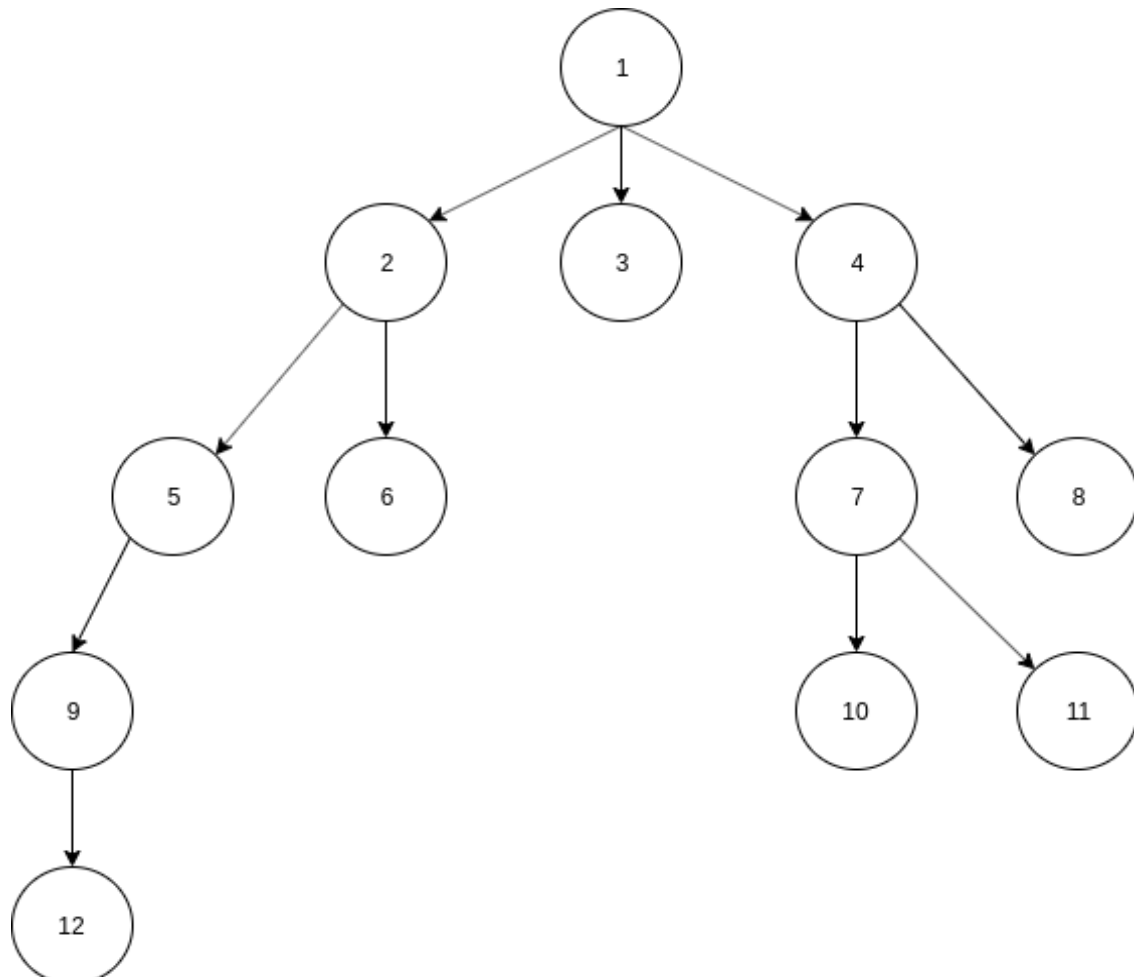
Rysunek 2 Graf skierowany i odpowiadająca mu macierz sąsiedztwa

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 zapętlenia w przypadku napotkania cyklu do wierzchołka dodaje się parametr mówiący o tym, czy został on już odwiedzony. [CITATION mgr19 \l 1045]

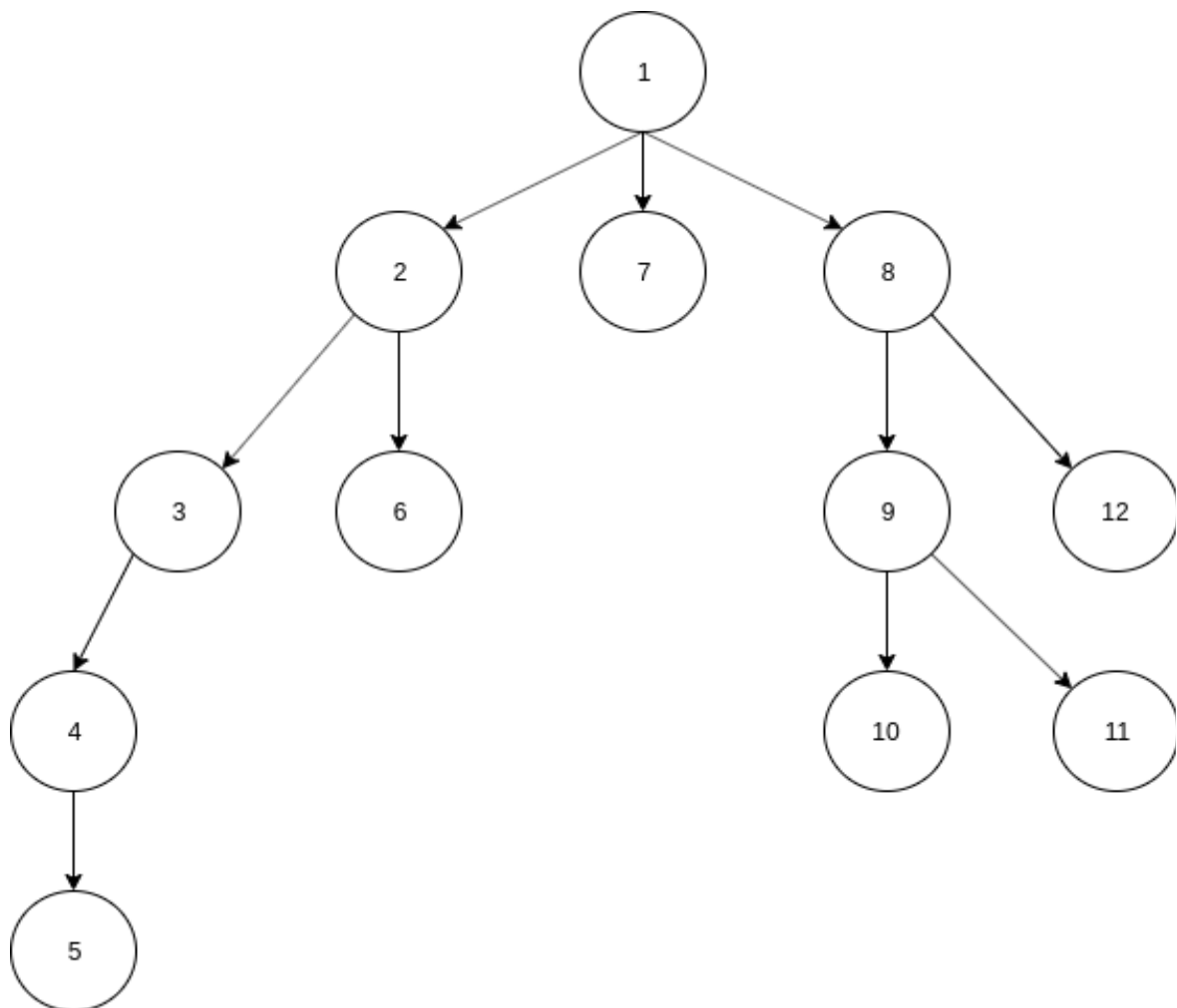
Kolejność odwiedzania wierzchołków została przedstawiona na poniższym diagramie.



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 tę akcję 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. [CITATION mgr19 \l 1045]

Kolejność odwiedzania wierzchołków została przedstawiona na poniższym diagramie.

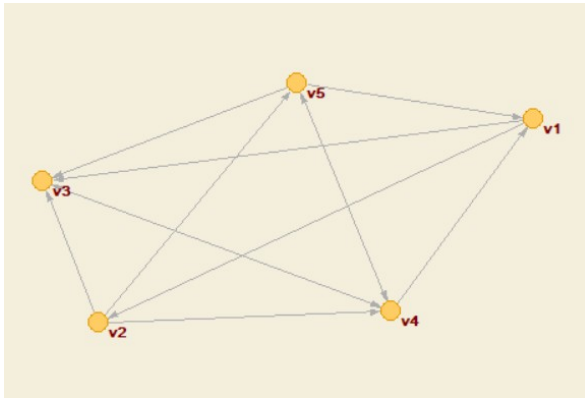


Rysunek 4 Graf przedstawiający kolejność przeszukania grafu przy wykorzystaniu algorytmu DFS

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.



Lista wierzchołków	Sąsiednie wierzchołki		
	2	3	
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

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.

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 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#

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

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*. [CITATION Sto19 \I 1045]

- b) dla implementacji w C/C++

$$1 ns$$

Wykorzystując chrono API z C++11. Dostępny tam zegar jest zegarem systemowym z najkrótszym możliwym okresem tyknięć. Jest to zegar z największą możliwą precyzją. [CITATION cpl19 \I 1045]

std::chrono::high_resolution_clock

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

Sprawozdanie nr 3

6. Dokumentacja kodu

Ogólny algorytm opisujący działanie programu można przedstawić w następujących krokach:

- 1) Wczytaj plik txt opisujący daną sieć przepływową
- 2) Na podstawie wcześniej odczytanych danych z pliku tekstowego stwórz model sieci przepływowej
- 3) Uruchom zegar
- 4) Uruchom algorytm Forda-Fulkersona dla zwirtualizowanej w pamięci komputera sieci przepływowej
- 5) Zatrzymaj zegar
- 6) Podaj wartość maksymalnego przepływu i czas wykonywania algorytmu

Model sieci przepływowej w pamięci komputera - klasa FlowGraph

W celu stworzenia modelu sieci przepływowej do konstruktora klasy FlowGraph przekazywana jest lista stringów zawierające kolejne wiersze pliku tekstowego. Kolejne linie zawierają informacje na temat liczby wierzchołków sieci, lokalizacji źródła oraz ujścia sieci przepływowej oraz przepustowości każdej z krawędzi.

- Parametry
 - o Publiczne
 - `int` Vertics - liczba wierzchołków sieci
 - `int` Source - Źródło sieci przepływowej
 - `int` Sink - Ujście sieci przepływowej
 - o Prywatne
 - `Dictionary<int, List<int>>` AdjacencyListArray - słownik w którym kluczem jest numer wierzchołka, natomiast wartością lista sąsiadujących z nim wierzchołków
 - `Dictionary<Tuple<int, int>, Edge>` Edges - słownik w którym kluczem jest para wierzchołków, gdzie pierwszy numer wierzchołka opisuje początek krawędzi natomiast, kolejny numer wierzchołka jest końcem krawędzi, wartością zaś jest obiekt `Edge` opisujące takie właściwości dla danej krawędzi jak przepustowość i aktualna wartość przepływu
- Metody

- o Publiczne
 - `int` `FordFulkerson()` - uruchamia algorytm Forda-Fulkersona dla sieci, w wyniku zwraca maksymalny przepływ
- o Prywatne
 - `List<int>?` `FindExtendedPath()` - metoda zwracająca listę będącą kolejnymi wierzchołkami znalezionej ścieżki rozszerzającej (w przypadku braku ścieżki rozszerzającej zwraca null)
 - `int` `GetExtendedFlow(List<int> path)` - zwraca wartość przepustowości rezydualnej dla ścieżki rozszerzającej przekazanej do metody w postaci listy kolejnych wierzchołków ścieżki
 - `void` `UpdateFlow(List<int> path, int valueFlow)` - metoda zwiększa przepływ dla danej ścieżki o wartość przekazaną do za pomocą parametru `valueFlow`

Model krawędzi - klasa `Edge`

Każda z krawędzi grafu jest zrytualizowana w pamięci komputera w postaci obiektu `Edge`. Przy czym jest to krawędź skierowana, a zatem jeżeli w krawędzi jest możliwy przepływ zarówno od wierzchołka A do B, jak i od B do A to w pamięci komputera będą to dwie odrębne krawędzie. W celu stworzenia obiektu `Edge`, należy przekazać poprzez konstruktor wierzchołek początkowy i końcowy oraz przepustowość. Początkowy przepływ podczas tworzenia obiektu jest ustawiany na wartość 0.

- Parametry
 - o Prywatne
 - `int` `startVertic` - wierzchołek początkowy
 - `int` `endVertic` - wierzchołek końcowy
- Właściwości
 - o Publiczne
 - `int` `Capacity` - przepustowość
 - `int` `Flow` - aktualny przepływ, przy czym zwiększając wartość właściwości `Flow`, automatycznie zmniejszana jest wartość właściwości `AvailableFlow` według wzoru $\text{AvailableFlow} = \text{Capacity} - \text{Flow}$
 - `int` `AvailableFlow` - wartość o jaką możliwe jest zwiększenie przepływu, przy czym zwiększając wartość właściwości `AvailableFlow`, automatycznie zmniejszana jest wartość właściwości `Flow` według wzoru $\text{Flow} = \text{Capacity} - \text{AvailableFlow}$

Implementacja algorytmu Forda-Fulkersona - metoda FordFulkerson()

Metoda FordFulkerson() działa według następującego algorytmu:

- 1) Zainicjalizuj wartość maksymalnego przepływu w sieci $\text{maxFlow} = 0$
- 2) Znajdź ścieżkę rozszerzającą - metoda FindExtendedPath()
- 3) Jeżeli nie znaleziono ścieżki przejdź do punktu 8, jeżeli znaleziono ścieżkę przejdź do punktu 4
- 4) znajdź wartość przepustowości rezydualnej dla znalezionej ścieżki rozszerzającej - GetExtendedFlow(extendedPath)
- 5) Zwiększ przepływ w sieci - UpdateFlow(extendedPath, valueFlowToUpdate)
- 6) maxFlow zwiększ o wartość o jaką został zwiększony przepływ w sieci
- 7) Wróć do punktu 2
- 8) Zakończ działanie algorytmu i zwróć wartość maksymalnego przepływu w sieci

Metoda FindExtendedPath() wykorzystywana do poszukiwania ścieżki rozszerzającej działa na zasadzie przeszukiwania wierzchołków sieci za pomocą algorytmu przeszukiwania wszerz w celu zlokalizowania ścieżki prowadzącej od źródła do ujścia na której to ścieżce możliwe jest zwiększenie przepływu. Kiedy taka ścieżka zostanie znaleziona metoda GetExtendedFlow(extendedPath) ponownie analizuje każdą z krawędzi ścieżki w celu wyznaczenia wartości przepustowości rezydualnej o jaką możliwe jest zwiększenie przepływu na tej ścieżce. W ostatnim kroku należy zwiększyć przepływ i za to odpowiada metoda UpdateFlow(extendedPath, valueFlowToUpdate), która dla każdej krawędzi wchodzącej w skład ścieżki zwiększa przepływ o daną wartość, natomiast dla krawędzi o zwrocie przeciwnym o taką samą wartość zmniejsza przepływ.

Implementacja w C++ i C# co do zasady działania jest w zasadzie identyczna. Różnice w implementacjach mają związek jedynie z innymi mechanizmami i zasadami obowiązującymi w poszczególnych językach programowania.

Do zarządzania pamięcią w C++ zostały użyte sprytne wskaźniki (std::unique_ptr) oraz struktury danych z biblioteki standardowej (std::map, std::pair, std::vector, std::queue).

7. Wyniki testów

Pierwszym testem jest porównanie szybkości implementacji w grafach o zmiennej liczbie wierzchołków. Do wygenerowania testowych grafów użyliśmy autorskiego generatora. W czasie generowania wszystkie parametry oprócz liczby wierzchołków były stałe. Wygenerowaliśmy kolejno grafy o:

- 1000 wierzchołkach (100 grafów)
- 2000 wierzchołkach (100 grafów)
- 3000 wierzchołkach (100 grafów)
- 4000 wierzchołkach (60 grafów)
- 5000 wierzchołkach (65 grafów)

Te same grafy zostały podane na wejście obu algorytmów.
Dokładne wyniki testów zostały przedstawione w tabeli oraz na wykresie.

Tabela 1. Średni czas (w sekundach) potrzebny na obliczenie maksymalnego przepływu dla jednego grafu przy zadanej liczbie wierzchołków grafu.

Liczba wierzchołków	C++	C#
1000	3.298268160	7.243753825
2000	24.435375440	59.576470779
3000	75.716189280	191.352701836
4000	196.439735830	512.230677821
5000	422.514657600	1143.173310955

Szybkość implementacji przy zmiennej liczbie wierzchołków w grafie



Całkowity czas test dla implementacji w C++: **15h57m32s**

Całkowity czas testu dla implementacji w C#: **1d11h42m51s**

Suma czasu testu obu implementacji: **2d03h40m23s**

Drugim testem jest porównanie szybkości implementacji w grafach o zmiennym średnim stopniu wierzchołka. Wygenerowaliśmy kolejno grafy o 2000 wierzchołkach i średnim stopniu wierzchołka:

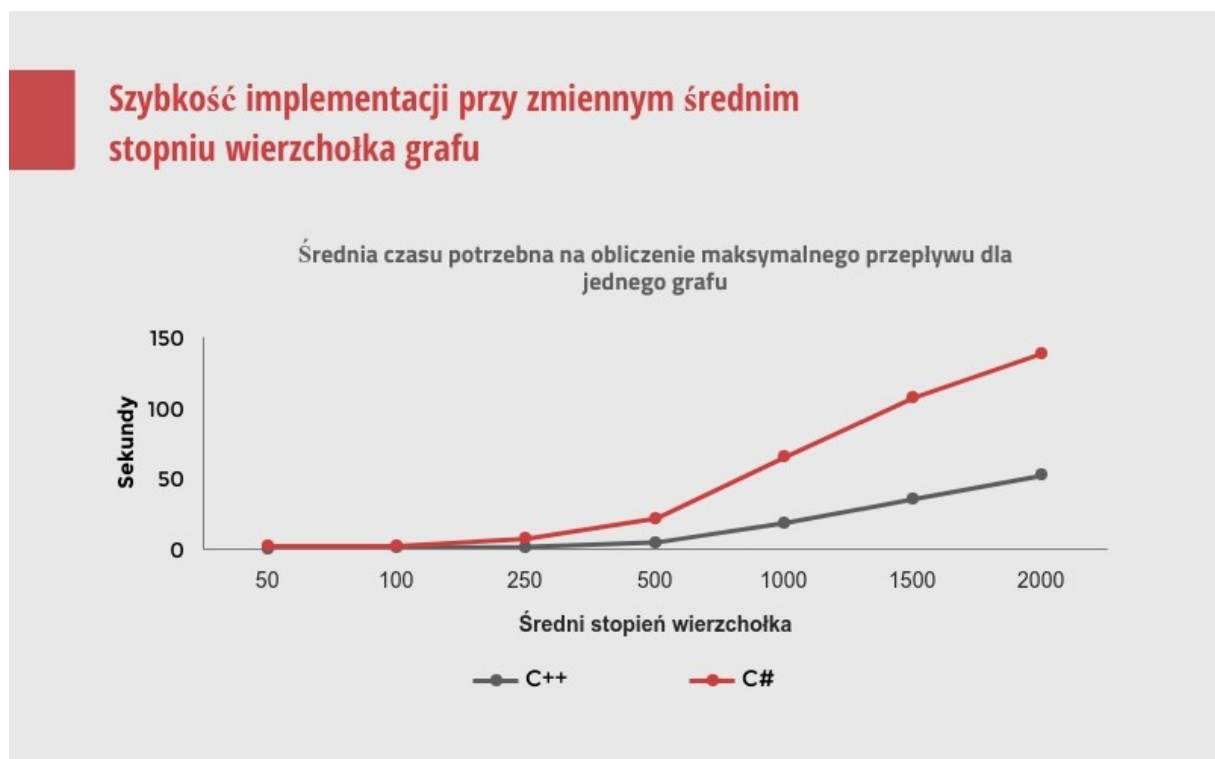
- 50 (50 grafów)
- 100 (50 grafów)
- 250 (50 grafów)
- 500 (50 grafów)
- 1000 (50 grafów)
- 1500 (50 grafów)
- 2000 (50 grafów)

Te same grafy zostały podane na wejście obu algorytmów.

Dokładne wyniki testów zostały przedstawione w tabeli oraz na wykresie.

Tabela 2. Średni czas (w sekundach) potrzebny na obliczenie maksymalnego przepływu dla jednego grafu przy zadanym średnim stopniu wierzchołka.

Średni stopień wierzchołka	C++	C#
50	0.491261278	2.049197284
100	0.711999676	2.128665344
250	1.556825372	7.257152641
500	4.696450637	21.621058230
1000	18.396773524	64.864141279
1500	35.282266709	107.039015298
2000	52.018417921	138.365151997



Całkowity czas test dla implementacji w C++: **01h34m17s**

Całkowity czas testu dla implementacji w C#: **04h46m06s**

Suma czasu testu obu implementacji: **06h20m23s**

Trzecim testem jest porównanie szybkości implementacji w grafach o zmiennym rozrzucie wartości przepływu pomiędzy wierzchołkami. Wygenerowaliśmy kolejno grafy o 2000 wierzchołkach, gdzie rozrzut wartości przepływu wahał się między:

- 0-100 (50 grafów)
- 0-1000 (50 grafów)
- 0-10000 (50 grafów)
- 0-100000 (50 grafów)

Te same grafy zostały podane na wejście obu algorytmów. Dokładne wyniki testów zostały przedstawione w tabeli oraz na wykresie.



Tabela 3. Średni czas (w sekundach) potrzebny na obliczenie maksymalnego przepływu dla jednego grafu przy zadanym zakresie wartości przepływu pomiędzy wierzchołkami

Rozrzut wartości przepływu	C++	C#
0..100	19.875803185	69.710962364
0..1000	18.869949725	62.578765420
0..10000	18.864922421	70.389598843
0..100000	20.179385544	65.531525977

Całkowity czas test dla implementacji w C++: **01h04m49s**

Całkowity czas testu dla implementacji w C#: **03h43m30s**

Suma czasu testu obu implementacji: **04h48m20s**

Łączny czas kompletu testów: 2d14h49m07s

8. Wnioski

Każdy z testów przyniósł spodziewane rezultaty. Implementacja w C++ jest 3-krotnie szybsza, proporcja ta utrzymuje się wraz ze wzrostem wielkości grafów bądź też średnim stopniem wierzchołka.

Wartości przepływu wierzchołków nie mają znaczenia jeśli chodzi o czas działania algorytmu.

W zastosowaniach gdzie liczy się wydajność polecamy implementowanie w C++, jednak należy podkreślić, że sama praca nad kodem może potrwać dłużej z racji tego, że C++ jest niskopoziomowym językiem. C# natomiast jest językiem o wysokim poziomie abstrakcji, posiada mechanizmy takie jak Garbage Collector, które zwalniają z nas obowiązek dbania o pamięć co przyspiesza kodowanie.

Sama platforma nie powinna mieć znaczenia, obie implementacje udało się uruchomić zarówno na Linuxie jak i Windowsie.

Na czas na pewno będzie miał wpływ sprzęt jakiego użyjemy. Test pierwszy (ze zmienną liczbą wierzchołków w grafie) został wykonany na Windowsie, na sprzęcie o specyfikacji:

- Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
- 24 GB RAM DDR3

Dwa pozostałe testy zostały wykonane na Linuxie, na sprzęcie o specyfikacji:

- Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz
- 8 GB RAM DDR3

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

Rysunek 1 Przepustowość rezydualna.....	2
Rysunek 2 Graf skierowany i odpowiadająca mu macierz sąsiedztwa.....	2
Rysunek 3 Graf przedstawiający kolejność przeszukania grafu przy wykorzystaniu algorytmu BFS.....	2
Rysunek 4 Graf przedstawiający kolejność przeszukania grafu przy wykorzystaniu algorytmu DFS.....	2
Rysunek 5 Graf skierowany wraz z odpowiadającą mu tablicą list sąsiedztwa wierzchołków.....	2