Obraz zawierający tekst

Opis wygenerowany automatycznie

**Dariusz Strojny**

Algorytmy i Struktury Danych

Projekt zaliczeniowy nr 3

Rzeszów, 2023

# Spis treści

[1 Spis treści 3](#_Toc125446976)

[2 Spis ilustracji 5](#_Toc125446977)

[3 Temat 6](#_Toc125446978)

[4 Analiza, projektowanie 7](#_Toc125446979)

[4.1 Wstęp teoretyczny 7](#_Toc125446980)

[4.1.1 Graf 7](#_Toc125446981)

[4.1.2 Graf skierowany 7](#_Toc125446982)

[4.1.3 Wierzchołki grafu 7](#_Toc125446983)

[4.1.4 Krawędzie w grafie 8](#_Toc125446984)

[4.1.5 Sąsiedzi w grafie 8](#_Toc125446985)

[4.1.6 Stopień wierzchołka 8](#_Toc125446986)

[4.1.7 Pętle 8](#_Toc125446987)

[4.2 Zasada działania programu 9](#_Toc125446988)

[4.3 Struktury danych 9](#_Toc125446989)

[4.4 Metodyka 10](#_Toc125446990)

[4.4.1 Funkcje użyte w programie 10](#_Toc125446991)

[5 Dane wejściowe 13](#_Toc125446992)

[6 Opisy i działanie funkcji programu 14](#_Toc125446993)

[6.1 Wypisywanie wszystkich sąsiadów dla każdego wierzchołka grafu 14](#_Toc125446994)

[6.1.1 Wynik działania programu 14](#_Toc125446995)

[6.1.2 Pseudokod 14](#_Toc125446996)

[6.1.3 Schemat blokowy 15](#_Toc125446997)

[6.2 Wypisywanie wszystkich wierzchołków, które są sąsiadami każdego wierzchołka 16](#_Toc125446998)

[6.2.1 Wynik działania programu 16](#_Toc125446999)

[6.2.2 Pseudokod 16](#_Toc125447000)

[6.2.3 Schemat blokowy 17](#_Toc125447001)

[6.3 Wypisywanie stopni wychodzących wszystkich wierzchołków 18](#_Toc125447002)

[6.3.1 Wynik działania programu 18](#_Toc125447003)

[6.3.2 Pseudokod 18](#_Toc125447004)

[6.3.3 Schemat blokowy 19](#_Toc125447005)

[6.4 Wypisywanie stopni wchodzących wszystkich wierzchołków 20](#_Toc125447006)

[6.4.1 Wynik działania programu 20](#_Toc125447007)

[6.4.2 Pseudokod 20](#_Toc125447008)

[6.4.3 Schemat blokowy 21](#_Toc125447009)

[6.5 Wypisywanie wszystkich wierzchołków izolowanych 22](#_Toc125447010)

[6.5.1 Wynik działania programu 22](#_Toc125447011)

[6.5.2 Pseudokod 22](#_Toc125447012)

[6.5.3 Schemat blokowy 23](#_Toc125447013)

[6.6 Wypisywanie wszystkich pętli 24](#_Toc125447014)

[6.6.1 Wynik działania programu 24](#_Toc125447015)

[6.6.2 Pseudokod 24](#_Toc125447016)

[6.6.3 Schemat blokowy 25](#_Toc125447017)

[6.7 Wypisywanie wszystkich krawędzi dwukierunkowych 26](#_Toc125447018)

[6.7.1 Wynik działania programu 26](#_Toc125447019)

[6.7.2 Pseudokod 26](#_Toc125447020)

[6.7.3 Schemat blokowy 27](#_Toc125447021)

[7 Kod programu 28](#_Toc125447022)

[7.1 Definicja struktur 28](#_Toc125447023)

[7.2 Deklaracje funkcji 28](#_Toc125447024)

[7.3 Funkcja main 29](#_Toc125447025)

[7.4 Funkcje pomocnicze 30](#_Toc125447026)

[7.5 Funkcje z treści zadania 31](#_Toc125447027)

# Spis ilustracji

[Rysunek 1 - Przykładowy graf skierowany 7](#_Toc125447028)

[Rysunek 2 - Wejściowy graf skierowany 13](#_Toc125447029)

[Rysunek 3 - Wynik działania funkcji 1 14](#_Toc125447030)

[Rysunek 4 - Pseudokod funkcji 1 14](#_Toc125447031)

[Rysunek 5 - Schemat blokowy funkcji 1 15](#_Toc125447032)

[Rysunek 6 - Wynik działania funkcji 2 16](#_Toc125447033)

[Rysunek 7 - Pseudokod funkcji 2 16](#_Toc125447034)

[Rysunek 8 - Schemat blokowy funkcji 2 17](#_Toc125447035)

[Rysunek 9 - Wynik działania funkcji 3 18](#_Toc125447036)

[Rysunek 10 - Pseudokod funkcji 3 18](#_Toc125447037)

[Rysunek 11 - Schemat blokowy funkcji 3 19](#_Toc125447038)

[Rysunek 12 - Wynik działania funkcji 4 20](#_Toc125447039)

[Rysunek 13 - Pseudokod funkcji 4 20](#_Toc125447040)

[Rysunek 14 - Schemat blokowy funkcji 4 21](#_Toc125447041)

[Rysunek 15 - Wynik działania funkcji 5 22](#_Toc125447042)

[Rysunek 16 - Pseudokod funkcji 5 22](#_Toc125447043)

[Rysunek 17 - Schemat blokowy funkcji 5 23](#_Toc125447044)

[Rysunek 18 - Wynik działania funkcji 6 24](#_Toc125447045)

[Rysunek 19 - Pseudokod funkcji 6 24](#_Toc125447046)

[Rysunek 20 - Schemat blokowy funkcji 6 25](#_Toc125447047)

[Rysunek 21 - Wynik działania funkcji 7 26](#_Toc125447048)

[Rysunek 22 - Pseudokod funkcji 7 26](#_Toc125447049)

[Rysunek 23 - Schemat blokowy funkcji 7 27](#_Toc125447050)

# Temat

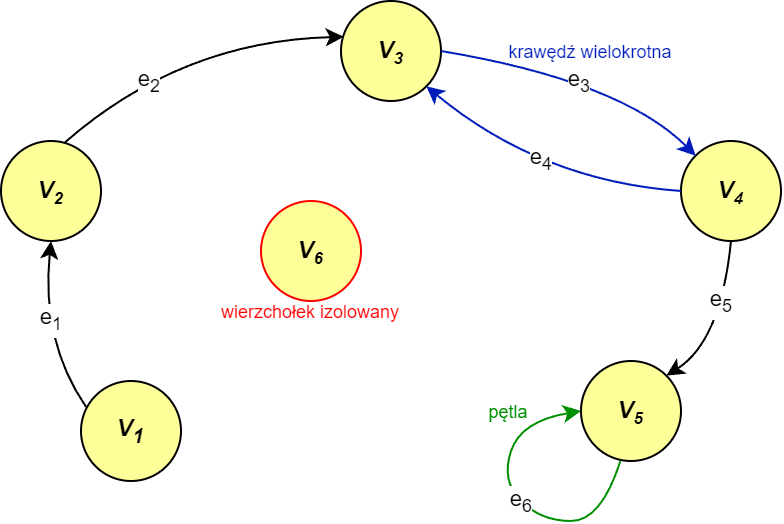
**Zadanie 9.** Napisz program, który dla zadanego grafu skierowanego reprezentowanego przy pomocy macierzy sąsiedztwa wyznaczy i wypisze następujące informacje:

1. Wszystkich sąsiadów dla każdego wierzchołka grafu (sąsiad wierzchołka to ten wierzchołek, do którego prowadzi krawędź z
2. Wszystkie wierzchołki, które są sąsiadami każdego wierzchołka
3. Stopnie wychodzące wszystkich wierzchołków
4. Stopnie wchodzące wszystkich wierzchołków
5. Wszystkie wierzchołki izolowane
6. Wszystkie pętle
7. Wszystkie krawędzie dwukierunkowe

Każdy z powyższych podpunktów powinien być realizowany jako oddzielna funkcja. W funkcji main() należy przedstawić działanie napisanej przez siebie biblioteki na reprezentatywnym przykładzie. Kod powinien być opatrzony stosownymi komentarzami.

# Analiza, projektowanie

## Wstęp teoretyczny



Rysunek 1 - Przykładowy graf skierowany

### Graf

Graf to matematyczny model reprezentujący zbiór obiektów (wierzchołków lub nodów) oraz relacji między nimi (krawędzi lub połączeń).

### Graf skierowany

Graf skierowany jest to rodzaj grafu, w którym krawędzie posiadają kierunek. Oznacza to, że krawędzie nie są symetryczne i nie można przejść od wierzchołka A do wierzchołka B tą samą krawędzią, co z powrotem. Grafy skierowane są często używane do modelowania różnych procesów, w których istnieją jasno określone kierunki przepływu, np. w sieciach komunikacyjnych lub w procesach biznesowych.

### Wierzchołki grafu

Wierzchołki (ang. vertices) to elementy grafu, które reprezentują obiekty lub relacje między obiektami. Mogą one reprezentować różne rzeczy, w zależności od kontekstu, w jakim jest używany graf. Na przykład, w grafie opisującym sieć drogową, wierzchołki mogą reprezentować miasta, a krawędzie drogi między nimi.

### Krawędzie w grafie

Krawędzie (ang. edges) to połączenia między wierzchołkami, które określają relacje między nimi. Krawędzie mogą być oznaczone różnymi atrybutami, takimi jak waga lub przepustowość, w zależności od kontekstu. W przypadku grafu opisującego sieć drogową, krawędzie mogą mieć atrybut "długość" lub "przepustowość"

### Sąsiedzi w grafie

Sąsiedzi (ang. neighbors) to wierzchołki, z którymi dany wierzchołek jest połączony krawędzią. W grafie skierowanym, sąsiedzi to tylko te wierzchołki, do których krawędź jest skierowana.

### Stopień wierzchołka

Stopień wierzchołka (ang. vertex degree) to liczba krawędzi, które do niego prowadzą lub od niego wychodzą. W grafie skierowanym rozróżniamy stopie

### Pętle

Pętle (ang. loops) to krawędzie, które łączą dany wierzchołek z samym sobą. Są one szczególnym przypadkiem krawędzi skierowanych, ponieważ kończą się one i zaczynają w tym samym wierzchołku. Pętle są często używane w różnych modelach matematycznych, w których jest wymagane przejście przez dany wierzchołek wiele razy.

W grafach skierowanych, stopień wierzchołka oznacza liczbę krawędzi, które do niego prowadzą lub od niego wychodzą. Pętle są wliczane do stopnia wierzchołka, co oznacza, że jeśli wierzchołek posiada pętlę, jego stopień jest o jeden większy niż liczba krawędzi, które do niego prowadzą lub od niego wychodzą.

## Zasada działania programu

Program ten ma za zadanie wykonanie zbioru określonych akcji na grafie skierowanym reprezentowanym przy pomocy macierzy sąsiedztwa. Macierz ta ma zostać wypełniona wartościami określającymi istnienie krawędzi pomiędzy poszczególnymi wierzchołkami tego grafu.

## Struktury danych

Najważniejszą strukturą danych w programie jest macierz sąsiedztwa reprezentująca wprowadzany do niej graf skierowany. Macierz sąsiedztwa posiada wymiary n x n, gdzie n jest ilością wierzchołków w danym grafie. Każde pole w macierzy sąsiedztwa reprezentuje informację o istnieniu pewnej krawędzi poprzez wartość 1 lub jej brak przez wartość 0. Pola znajdujące się na przekątnej reprezentują istnienie krawędzi tworzących pętle na poszczególnych wierzchołkach. Pozostałe pola definiują krawędzie między wierzchołkami, których numery są odpowiednio zależne od numeru wiersza i kolumny, na których to te pola się znajdują.

Na przykład: Pole o numerze wiersza 3 i kolumny 2 w macierzy sąsiedztwa, reprezentuje istnienie krawędzi skierowanej z wierzchołka 3 do wierzchołka 2, natomiast pole o numerze wiersza 1 i kolumny 1 definiuje istnienie pętli na tym wierzchołku.

Dodatkowo utworzoną przeze mnie strukturą danych jest struktura grafu, przechowująca w kodzie tablicę będącą macierzą sąsiedztwa oraz ilość wierzchołków w grafie. Jej definicja wygląda następująco:

// struktura grafu przechowująca macierz sąsiedztwa oraz ilość wierzchołków grafu

struct {

// tablica dwuwymiarowa przechowująca informacje o krawędziach grafu

int\*\* array;

// ilość wierzchołków grafu

int size;

} typedef graph;

## Metodyka

### Funkcje użyte w programie

• graph **new\_graph**(int size)

Tworzy nową strukturę typu graph i inicjuje w niej pamięć dla macierzy sąsiedztwa

Parametry

*size* – ilość wierzchołków w tworzonym grafie

Zwraca

Zainicjowany obiekt typu graph

• void **free\_graph**(graph &g)

Zwalnia pamięć struktury typu graph

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_add\_edge**(graph& g, int a, int b)

Ustawia w macierzy sąsiedztwa grafu skierowanego informację o obecności krawędzi od wierzchołka a do wierzchołka b

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

*a* – numer wierzchołka, z którego wychodzi krawędź

*b* – numer wierzchołka, do którego wchodzi krawędź

• void **graph\_add\_edge**(graph& g, const std::vector<std::pair<int, int>>& edges)

Ustawia w macierzy sąsiedztwa grafu skierowanego informację o obecności krawędzi pomiędzy każdą z par przekazanych w tablicy

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

*edges* – tablica zawierająca pary numerów wierzchołków, między którymi mają powstać krawędzie

• void **print\_graph**(const graph& g)

Wypisuje zawartość macierzy sąsiedztwa grafu

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_print\_neighbors\_of\_each\_vertex**(const graph& g)

Wypisuje wszystkich sąsiadów dla każdego wierzchołka grafu

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_print\_each\_vertex\_that\_is\_neighbor**(const graph& g)

Wypisuje wszystkie wierzchołki, które są sąsiadami każdego wierzchołkaParametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_print\_vertex\_outdegrees**(const graph& g)

Wypisuje stopnie wychodzące wszystkich wierzchołków

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_print\_vertex\_indegrees**(const graph& g)

Wypisuje stopnie wchodzące wszystkich wierzchołków

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_print\_isolated\_vertexes**(const graph& g);

Wypisuje wszystkie wierzchołki izolowane

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_print\_loops**(const graph& g)

Wypisuje wszystkie pętle

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

• void **graph\_print\_multi\_edges**(const graph& g)

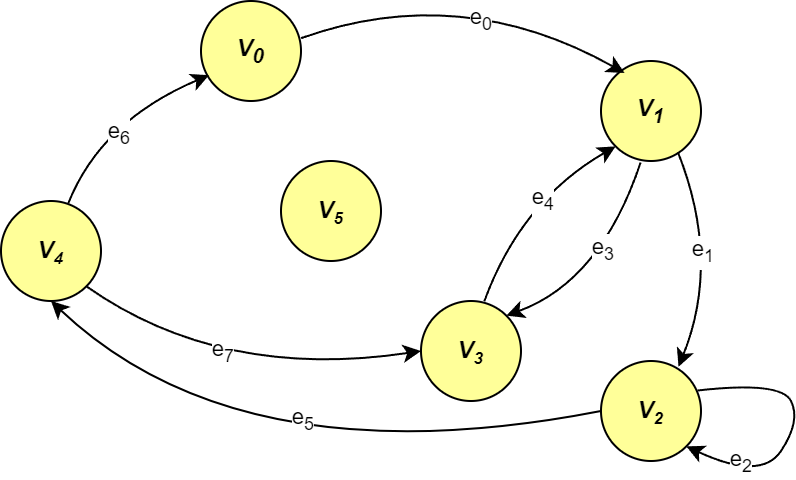
Wypisuje wszystkie krawędzie dwukierunkowe

Parametry

*g* – obiekt typu graph przechowujący macierz sąsiedztwa i ilość wierzchołków grafu skierowanego

# Dane wejściowe

Do przedstawienia działania napisanych funkcji do programu wprowadzony zostanie poniższy graf skierowany:



Rysunek 2 - Wejściowy graf skierowany

Jak możemy odczytać z grafu, posiada on:

Jeden wierzchołek izolowany ,

Jedną pętlę przy wierzchołku ,

Jedną krawędź dwukierunkową pomiędzy wierzchołkami

# Opisy i działanie funkcji programu

## Wypisywanie wszystkich sąsiadów dla każdego wierzchołka grafu

### Wynik działania programu

Wywołanie programu dla przykładowego grafu zwróciło następujące wyniki:

Sasiedzi dla kazdego wierzcholka grafu:

0: 1

1: 2, 3

2: 2, 4

3: 1

4: 0, 3

5: brak

Rysunek 3 - Wynik działania funkcji 1

### Pseudokod

wejście:

n - ilość wierzchołków grafu

g[n][n] - macierz sąsiedztwa grafu

dane:

i - iterator wierszy macierzy sąsiedztwa

j - iterator kolumn macierzy sąsiedztwa

algorytm:

i <- 0

dopóki i < n:

wypisz i

j <- 0

dopóki j < n:

jeżeli g[i][j] == 1:

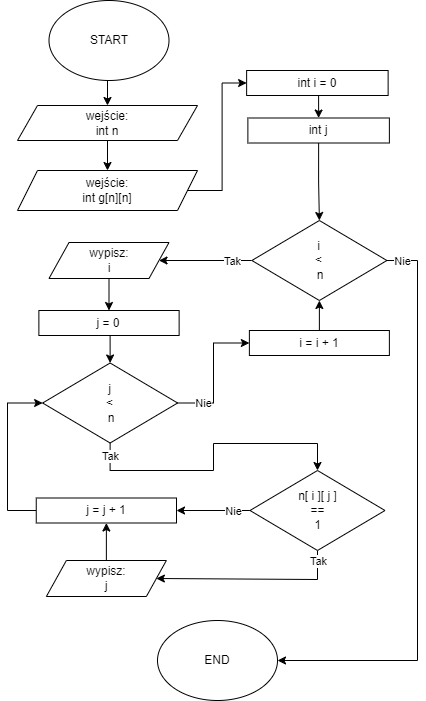
wypisz j

j <- j + 1

i <- i + 1

Rysunek 4 - Pseudokod funkcji 1

### Schemat blokowy



Rysunek 5 - Schemat blokowy funkcji 1

## Wypisywanie wszystkich wierzchołków, które są sąsiadami każdego wierzchołka

### Wynik działania programu

Wywołanie programu dla przykładowego grafu zwróciło następujące wyniki:

Wierzcholki, ktore sa sasiadami kazdego wierzcholka:

0: 4

1: 0, 3

2: 1, 2

3: 1, 4

4: 2

5: brak

Rysunek 6 - Wynik działania funkcji 2

### Pseudokod

wejście:

n - ilość wierzchołków grafu

g[n][n] - macierz sąsiedztwa grafu

dane:

i - iterator wierszy macierzy sąsiedztwa

j - iterator kolumn macierzy sąsiedztwa

algorytm:

i <- 0

dopóki i < n:

wypisz i

j = 0

dopóki j < n:

jeżeli g[j][i] == 1:

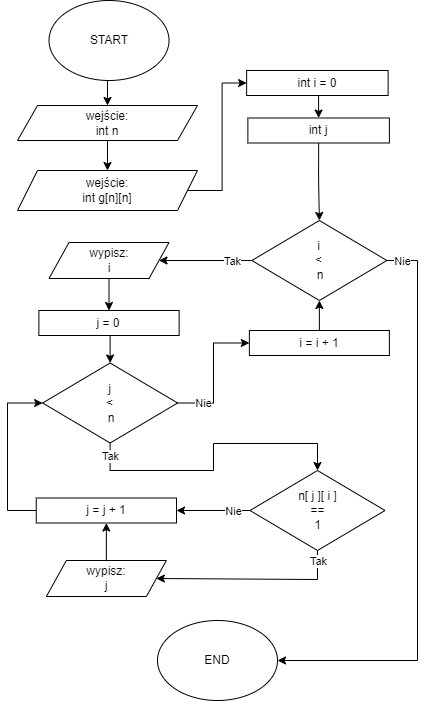
wypisz j

j <- j + 1

i <- i + 1

Rysunek 7 - Pseudokod funkcji 2

### Schemat blokowy



Rysunek 8 - Schemat blokowy funkcji 2

## Wypisywanie stopni wychodzących wszystkich wierzchołków

### Wynik działania programu

Wywołanie programu dla przykładowego grafu zwróciło następujące wyniki:

Stopnie wychodzace:

0: 1

1: 2

2: 2

3: 1

4: 2

5: 0

Rysunek 9 - Wynik działania funkcji 3

### Pseudokod

wejście:

n - ilość wierzchołków grafu

g[n][n] - macierz sąsiedztwa grafu

dane:

i - iterator wierszy macierzy sąsiedztwa

j - iterator kolumn macierzy sąsiedztwa

count - licznik wierzchołków, których dany wierzchołek jest sąsiadem

algorytm:

i <- 0

dopóki i < n:

wypisz i

count <- 0

j <- 0

dopóki j < n:

jeżeli g[i][j] == 1:

count <- count + 1

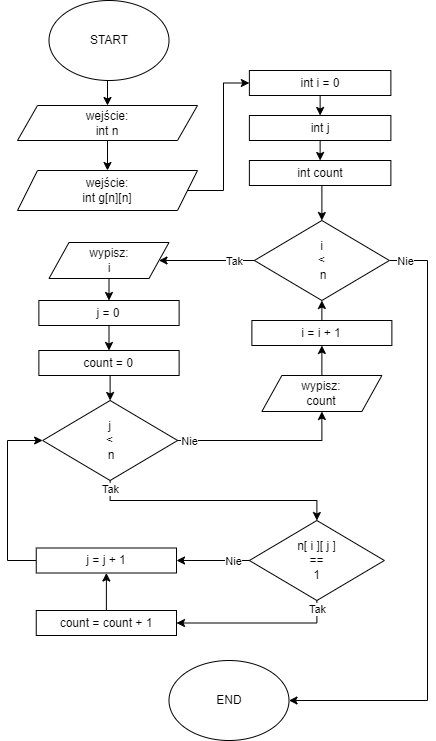
j <- j + 1

wypisz count

i <- i + 1

Rysunek 10 - Pseudokod funkcji 3

### Schemat blokowy



Rysunek 11 - Schemat blokowy funkcji 3

## Wypisywanie stopni wchodzących wszystkich wierzchołków

### Wynik działania programu

Wywołanie programu dla przykładowego grafu zwróciło następujące wyniki:

Stopnie wchodzace:

0: 1

1: 2

2: 2

3: 2

4: 1

5: 0

Rysunek 12 - Wynik działania funkcji 4

### Pseudokod

wejście:

n - ilość wierzchołków grafu

g[n][n] - macierz sąsiedztwa grafu

dane:

i - iterator wierszy macierzy sąsiedztwa

j - iterator kolumn macierzy sąsiedztwa

count - licznik wierzchołków będących sąsiadami danego wierzchołka

algorytm:

i <- 0

dopóki i < n:

wypisz i

count <- 0

j <- 0

dopóki j < n:

jeżeli g[j][i] == 1:

count <- count + 1

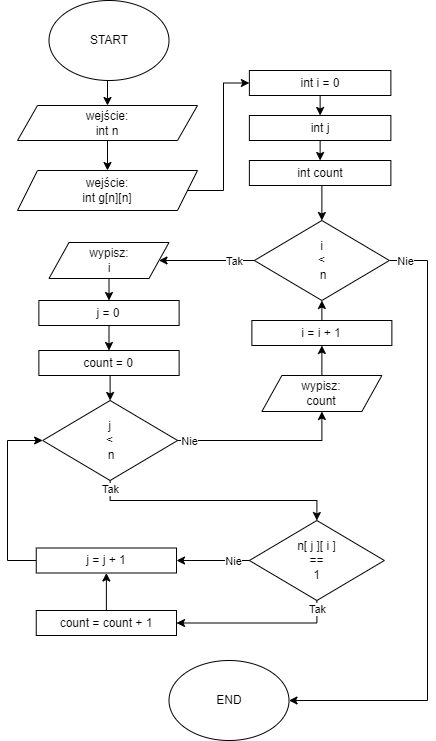
j <- j + 1

wypisz count

i <- i + 1

Rysunek 13 - Pseudokod funkcji 4

### Schemat blokowy



Rysunek 14 - Schemat blokowy funkcji 4

## Wypisywanie wszystkich wierzchołków izolowanych

### Wynik działania programu

Wywołanie programu dla przykładowego grafu zwróciło następujące wyniki:

Wierzcholki izolowane:

5

Rysunek 15 - Wynik działania funkcji 5

### Pseudokod

wejście:

n - ilość wierzchołków grafu

g[n][n] - macierz sąsiedztwa grafu

dane:

i - iterator wierszy macierzy sąsiedztwa

j - iterator kolumn macierzy sąsiedztwa

count - licznik krawędzi wchodzących bądź wychodzących z danego wierzchołka

algorytm:

i <- 0

dopóki i < n:

count <- 0

j <- 0

dopóki j < n:

jeżeli g[j][i] == 1 lub g[i][j] == 1:

count <- count + 1

j <- j + 1

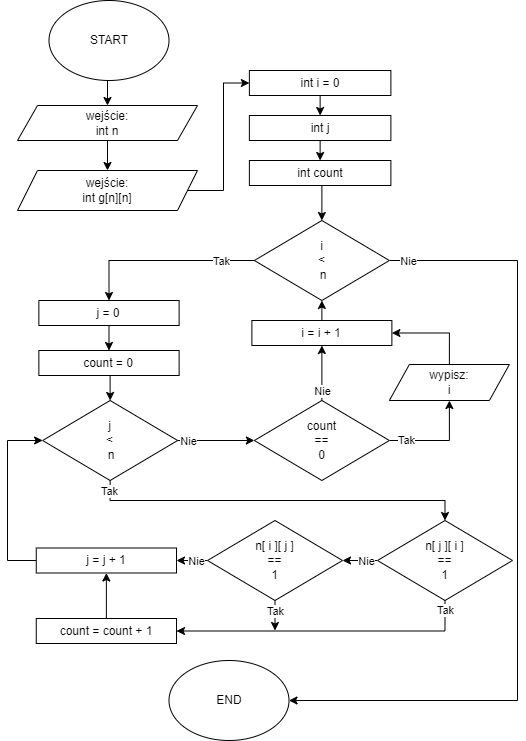
jeżeli count > 0:

wypisz i

i <- i + 1

Rysunek 16 - Pseudokod funkcji 5

### Schemat blokowy



Rysunek 17 - Schemat blokowy funkcji 5

## Wypisywanie wszystkich pętli

### Wynik działania programu

Wywołanie programu dla przykładowego grafu zwróciło następujące wyniki:

Petle:

2

Rysunek 18 - Wynik działania funkcji 6

### Pseudokod

wejście:

n - ilość wierzchołków macierzy sąsiedztwa

g[n][n] - macierz sąsiedztwa macierzy sąsiedztwa

dane:

i - iterator numerów wierzchołków

algorytm:

i <- 0

dopóki i < n:

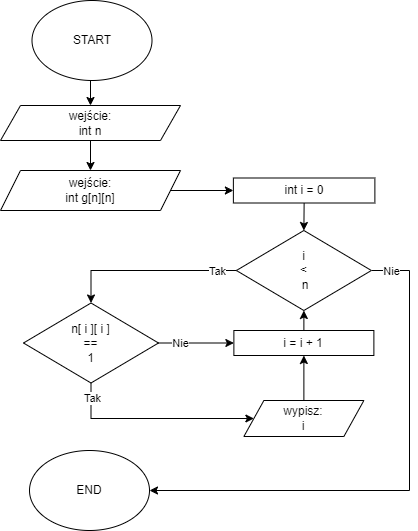
jeżeli g[i][i] == 1:

wypisz i

i <- i + 1

Rysunek 19 - Pseudokod funkcji 6

### Schemat blokowy



Rysunek 20 - Schemat blokowy funkcji 6

## Wypisywanie wszystkich krawędzi dwukierunkowych

### Wynik działania programu

Wywołanie programu dla przykładowego grafu zwróciło następujące wyniki:

Krawedzie dwukierunkowe:

1 <-> 3

Rysunek 21 - Wynik działania funkcji 7

### Pseudokod

wejście:

n - ilość wierzchołków macierzy sąsiedztwa

g[n][n] - macierz sąsiedztwa macierzy sąsiedztwa

dane:

i - iterator wierszy macierzy sąsiedztwa

j - iterator kolumn macierzy sąsiedztwa

algorytm:

i <- 0

dopóki i < n:

j <- i + 1

dopóki j < n:

jeżeli g[i][j] == 1 oraz g[j][i] == 1:

wypisz i

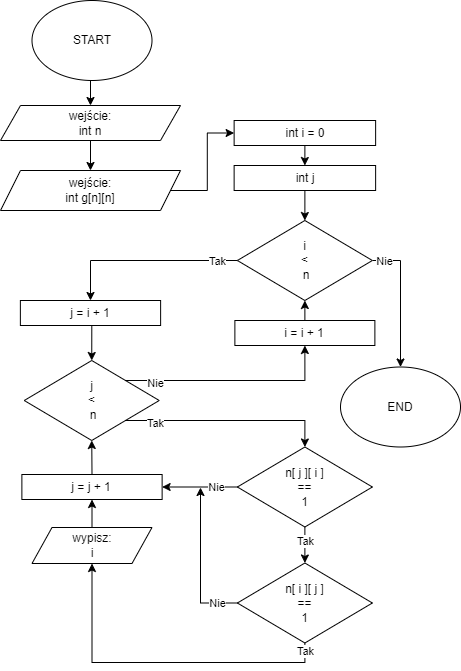
wypisz j

j <- j + 1

i <- i + 1

Rysunek 22 - Pseudokod funkcji 7

### Schemat blokowy



Rysunek 23 - Schemat blokowy funkcji 7

# Kod programu

## Definicja struktur

// struktura grafu przechowująca macierz sąsiedztwa oraz ilość wierzchołków grafu

struct {

// tablica dwuwymiarowa przechowująca informacje o krawędziach grafu

int\*\* array;

// ilość wierzchołków grafu

int size;

} typedef graph;

## Deklaracje funkcji

graph new\_graph(int size);

void free\_graph(graph& g);

void graph\_add\_edge(graph& g, int a, int b);

void graph\_add\_edge(graph& g, const std::vector<std::pair<int, int>>& edges);

void print\_graph(const graph& g);

void graph\_print\_neighbors\_of\_each\_vertex(const graph& g);

void graph\_print\_each\_vertex\_that\_is\_neighbor(const graph& g);

void graph\_print\_vertex\_outdegrees(const graph& g);

void graph\_print\_vertex\_indegrees(const graph& g);

void graph\_print\_isolated\_vertexes(const graph& g);

void graph\_print\_loops(const graph& g);

void graph\_print\_multi\_edges(const graph& g);

## Funkcja main

int main()

{

// utworzenie grafu zawierającego 6 wierzchołków

graph g = new\_graph(6);

// wpisanie krawędzi do grafu

graph\_add\_edge(g, {

{0, 1},

{1, 2},

{2, 2},

{1, 3},

{3, 1},

{2, 4},

{4, 0},

{4, 3}

});

// wypisanie macierzy sąsiedztwa znajdującej się w grafie

print\_graph(g);

// 1) wypisanie wszystkich sąsiadów dla każdego wierzchołka grafu.

std::cout << "Sasiedzi dla kazdego wierzcholka grafu: \n";

graph\_print\_neighbors\_of\_each\_vertex(g);

// 2) wypisanie wszyskich wierzchołków, które są sąsiadami każdego wierzchołka

std::cout << "Wierzcholki, ktore sa sasiadami kazdego wierzcholka: \n";

graph\_print\_each\_vertex\_that\_is\_neighbor(g);

// 3) wypisanie stopni wychodzących wszystkich wierzchołków

std::cout << "Stopnie wychodzace: \n";

graph\_print\_vertex\_outdegrees(g);

// 4) wypisanie stopni wchodzących wszystkich wierzchołków

std::cout << "Stopnie wchodzace:\n";

graph\_print\_vertex\_indegrees(g);

// wypisanie wszystkich wierzchołków izolowanych

std::cout << "Wierzcholki izolowane: \n";

graph\_print\_isolated\_vertexes(g);

// wypisanie wysztkich pętli

std::cout << "Petle: \n";

graph\_print\_loops(g);

// wypisanie wszystkich krawędzi dwukierunkowych

std::cout << "Krawedzie dwukierunkowe: \n";

graph\_print\_multi\_edges(g);

// zwolnienie pamięci struktury grafu

free\_graph(g);

return 0;

}

## Funkcje pomocnicze

// inicjalizacja struktury grafu tablicą wypełnioną zerami

graph new\_graph(int size)

{

if (size < 0) throw "Rozmiar grafu nie moze byc ujemny!\n";

// stworzenie nowej struktury typu graf

graph g;

// przypisanie wymiarów grafu do struktury

g.size = size;

// zainicjowanie tablicy wskaźników

g.array = new int\* [size];

int i, j;

// dla każdego elementu tablicy

for (int i = 0; i < size; i++) {

// zainicjowanie pamięci dla wiersza

g.array[i] = new int[size];

// zapełnienie zainicjowanej pamięci zerami

memset(g.array[i], 0, size \* sizeof(int));

}

// zwrócenie utworzonej struktury

return g;

}

// zwolnienie pamięci zajętej przez strukturę grafu

void free\_graph(graph& g)

{

// dla każdego elementu w tablicy

for (int i = 0; i < g.size; i++)

{

// zwolnienie pamięci dla wiersza

delete[] g.array[i];

}

// zwolnienie pamięci tablicy

delete[] g.array;

// wyzerowanie elementów w strukturze

g.array = NULL;

g.size = 0;

}

// dodanie krawędzi między wierzchołkami a i b do grafu

void graph\_add\_edge(graph& g, int a, int b)

{

// jeżeli podane wierzchołki znajduja się w zakresie grafu

if (a < 0 || a >= g.size || b < 0 || b >= g.size) throw "Podane wierzcholki nie znajduja sie w grafie!";

// wpisanie istnienia krawędzi do macierzy sąsiedztwa

g.array[a][b] = 1;

}

// przeciążenie funkcji graph\_add\_edge przyjmujące tablicę par numerów wierzchołków tworzących krawędzie

void graph\_add\_edge(graph& g, const std::vector<std::pair<int, int>>& edges)

{

// dla każdej pary wierzchołków

for (const std::pair<int, int>& edge : edges)

{

// utworzenie krawędzi między nimi

graph\_add\_edge(g, edge.first, edge.second);

}

}

// wypisanie zawartości tablicy relacji między wierzchołkami grafu

void print\_graph(const graph& g)

{

int i, j;

// wypisanie numerów kolumn grafu

printf(" ");

for (i = 0; i < g.size; i++) printf("%3d ", i);

printf("\n\n");

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

// wypisanie numeru wiersza

printf("%2d ", i);

// wypisanie zawartości wiersza

for (j = 0; j < g.size; j++)

printf("%3d ", g.array[i][j]);

printf("\n");

}

}

## Funkcje z treści zadania

// wypisanie sąsiadów każdego wierzchołka

void graph\_print\_neighbors\_of\_each\_vertex(const graph& g)

{

int i, j, print\_count;

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

print\_count = 0;

// wypisanie numeru wierzchołka

printf("%2d: ", i);

// dla każdej kolumny w tablicy

for (j = 0; j < g.size; j++)

{

// jeżeli istnieje krawędź w grafie od wierzchołka i do j

if (g.array[i][j] == 1)

{

// wypisanie numeru wierzchołka j

printf("%s%2d", print\_count++ ? ", " : "", j);

}

}

// wypisanie informacji o braku elementów jeżeli

licznik wypisań jest równy 0

if (print\_count == 0) printf(" brak");

printf("\n");

}

}

// wypisanie każdego wierzchołka, który jest sąsiadem każdego wierzchołka

void graph\_print\_each\_vertex\_that\_is\_neighbor(const graph& g)

{

int i, j, print\_count;

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

print\_count = 0;

// wypisanie numeru wierzchołka

printf("%2d: ", i);

// dla każdej kolumny w tablicy

for (j = 0; j < g.size; j++)

{

// jeżeli istnieje krawędź w grafie od wierzchołka j do i

if (g.array[j][i] == 1)

{

// wypisanie numeru wierzchołka j

printf("%s%2d", print\_count++ ? ", " : "", j);

}

}

// wypisanie informacji o braku elementów jeżeli

licznik wypisań jest równy 0

if (print\_count == 0) printf(" brak");

printf("\n");

}

}

// wypisanie stopni wychodzących każdego z wierzchołków

void graph\_print\_vertex\_outdegrees(const graph& g)

{

int i, j, count;

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

// wypisanie numeru wierzchołka

printf("%2d: ", i);

// zerowanie licznika

count = 0;

// dla każdej kolumny w tablicy

for (j = 0; j < g.size; j++)

{

// jeżeli istnieje krawędź w grafie od wierzchołka i do j

if (g.array[i][j] == 1)

// zwiększenie licznika krawędzi wychodzących z wierzchołka i

count++;

}

// wypisanie licznika

printf("%d\n", count);

}

}

// wypisanie stopni wchodzących każdego z wierzchołków

void graph\_print\_vertex\_indegrees(const graph& g)

{

int i, j, count;

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

// wypisanie numeru wierzchołka

printf("%2d: ", i);

// zerowanie licznika

count = 0;

// dla każdej kolumny w tablicy

for (j = 0; j < g.size; j++)

{

// jeżeli istnieje krawędź w grafie od wierzchołka j do i

if (g.array[j][i] == 1)

// zwiększenie licznika krawędzi wchodzących do wierzchołka i

count++;

}

// wypisanie licznika

printf("%d\n", count);

}

}

// wypisanie wszystkich izolowanych wierzchołków

void graph\_print\_isolated\_vertexes(const graph& g)

{

int i, j, count, print\_count = 0;

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

// zerowanie licznika

count = 0;

// dla każdej kolumny w tablicy

for (j = 0; j < g.size; j++)

{

// jeżeli istnieje krawędź w grafie między wierzchołkami i oraz j

if (g.array[j][i] == 1 || g.array[i][j] == 1)

// zwiększenie licznika krawędzi wchodzących lub wychodzących z wierzchołka i

count++;

}

// jeżeli licznik jest równy 0 to dany wierzchołek nie posiada żadnych krawędzi i jest izolowany

if (count == 0) printf("%s%2d", print\_count++ ? ", " : "", i);

}

printf("\n");

}

// wypisanie wszystkich pętli w grafie

void graph\_print\_loops(const graph& g)

{

int i, print\_count = 0;

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

// jeżeli wierzchołek i łączy się z krawędzią z samym sobą to jest to pętla

if (g.array[i][i] == 1) printf("%s%2d", print\_count++ ? ", " : "", i);

}

if (print\_count == 0) printf(" brak");

printf("\n");

}

// wypisanie wszystkich krawędzi dwukierunkowych w grafie

void graph\_print\_multi\_edges(const graph& g)

{

int i, j, print\_count = 0;

// dla każdego elementu w tablicy

for (i = 0; i < g.size; i++)

{

// dla każdej kolumny w tablicy pomijając elementy już sprawdzone przez poprzednie iteracje oraz wierzchołek i

for (j = i + 1; j < g.size; j++)

{

// jeżeli istnieją krawędzie w obu kierunkach między wierzchołkami i oraz j

if (g.array[i][j] == 1 && g.array[j][i] == 1)

// wypisanie tej pary wierzchołków

printf("%s%d <-> %d", print\_count++ ? ", " : "", i, j);

}

}

if (print\_count == 0) printf(" brak");

printf("\n");

}