Podstawy programowania 2

Grafy, algorytmy DFS i BFS

Arkadiusz Chrobot

Katedra Systemów Informatycznych

29 maja 2024

Plan

Wprowadzenie

Grafy są strukturami danych używanymi do reprezentowania relacji między danymi. Choć są stosunkowo prostą koncepcją, to mają wiele zastosowań w informatyce. Struktury te bazują na pojęciach matematycznych, które zostały pierwotnie opracowane przez szwajcarskiego matematyka Leonharda Eulera i rozwinięte przez innych matematyków. Definicje niektórych z nich są przedstawione na kolejnych slajdach. Niestety, nie ma standardowej terminologii w teorii grafów, zatem niektóre z przytoczonych definicji mogą być w niektórych podręcznikach inne.

Definicja grafu

Definicja grafu

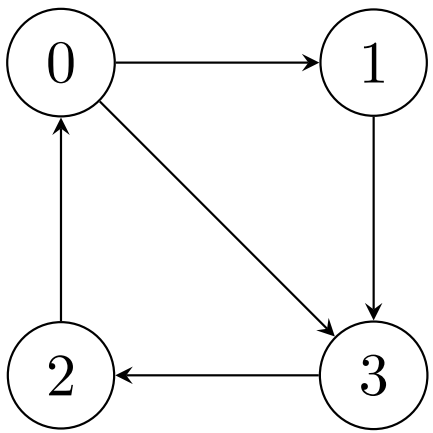
Definicja grafu skierowanego

Graf skierowany G jest opisany parą (V, E), gdzie V jest zbiorem skończonym, którego elementy są wierzchołkami grafu G, a E jest relacją binarną w V i $E \subseteq V \times V$. Zbiór V jest zbiorem wierzchołków, natomiast zbiór E jest zbiorem krawędzi.

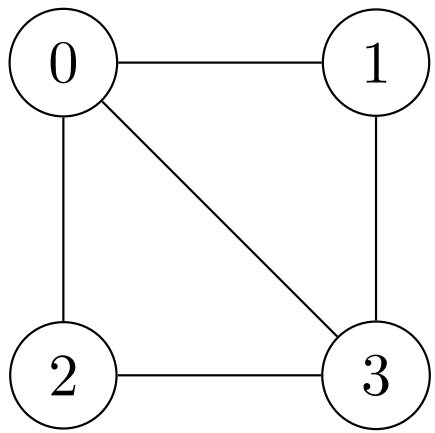
Definicja grafu nieskierowanego

Graf nieskierowany jest grafem, którego zbiór E jest nieuporządkowany. Oznacza to, że krawędź jest zbiorem $\{u, v\}$, gdzie $u, v \in V$ i $u \neq v$. Krawędź oznaczamy używając zapisu (u, v). Zapisy (u, v) i (v, u) oznaczają tę samą krawędź. W grafie nieskierowanym nie mogą występować pętle, czyli krawędzie prowadzące do tego samego wierzchołka, z którego się zaczynają.

Przykłady grafów



(a) Graf skierowany



(b) Graf nieskierowany

Przykłady grafów

Krawędzie i sąsiedztwo

Rodzaje krawędzi

W grafie skierowanym $G = (V, E)$ krawędź (u, v) jest krawędzią **wychodzącą** z wierzchołka u i **wchodzącą** do wierzchołka v. W grafie nieskierowanym taka krawędź (u, v) jest określana jako **incydentna** z wierzchołkami u i v.

Sąsiedztwo

Wierzchołek v jest **sąsiedni** do wierzchołka u w grafie $G = (V, E)$ jeśli łączy te wierzchołki krawędź (v, u). W grafie skierowanym *relacja sąsiedztwa* nie musi być symetryczna.

Stopień wierzchołka

Stopniem wierzchołka w grafie nieskierowanym jest liczba incydentnych z nim krawędzi. W grafie skierowanym **stopniem wejściowym** wierzchołka nazywamy liczbę krawędzi wchodzących do tego wierzchołka, a **stopniem wyjściowym** liczbę krawędzi z niego wychodzących. W grafie skierowanym **stopniem** wierzchołka jest suma stopnia wejściowego i wyjściowego.

Ścieżka i cykl

Definicja ścieżki

Ścieżka (droga) długości k z wierzchołka u do wierzchołka u' w grafie $G = (V, E)$ jest ciągiem wierzchołków $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ takich, że $u = v_0$, $u' = v_k$ i $(v_{i-1}, v_i) \in E$ dla $i = 1, 2, \ldots, k$. Długość ścieżki jest liczbą jej krawędzi. Ścieżka zawiera wierzchołki $v_0, v_1, v_2, \ldots, v_k$ i krawędzie $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Jeśli istnieje ścieżka z u do u', to mówimy, że u' jest osiągalny z u po ścieżce p. Ścieżka jest nazywana **ścieżką prostą** jeśli wszystkie jej wierzchołki są różne.

Definicja cyklu

Ścieżka $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ tworzy **cykl** jeśli $v_0 = v_k$. Cykl nazywamy cyklem **prostym** jeśli dodatkowo wszystkie jego wierzchołki są różne. **Pętla** jest cyklem o długości 1. Graf skierowany nieposiadający pętli i krawędzi wielokrotnych (występujących więcej niż raz) nazywamy grafem **prostym**. Graf, który nie zawiera cykli nazywamy grafem **acyklicznym**.

Spójność

Graf nieskierowany jest **spójny** jeśli każda para jego wierzchołków jest połączona ścieżką. Graf skierowany **silnie spójny** to taki, w którym każde dwa wierzchołki są osiągalne jeden z drugiego.

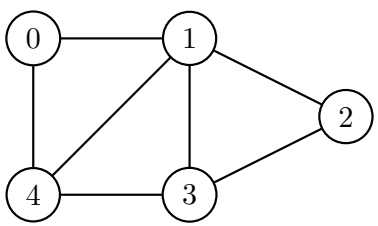
Grafy pełne i rzadkie

Graf *nieskierowany* nazywamy **grafem pełnym** jeśli każda para jego wierzchołków jest połączona krawędzią. Liczba krawędzi w takim grafie jest równa $\binom{n}{2}$, gdzie n jest liczbą wierzchołków grafu. Graf zawierający małą liczbę krawędzi w stosunku do liczby wierzchołków nazywamy **grafem rzadkim**.

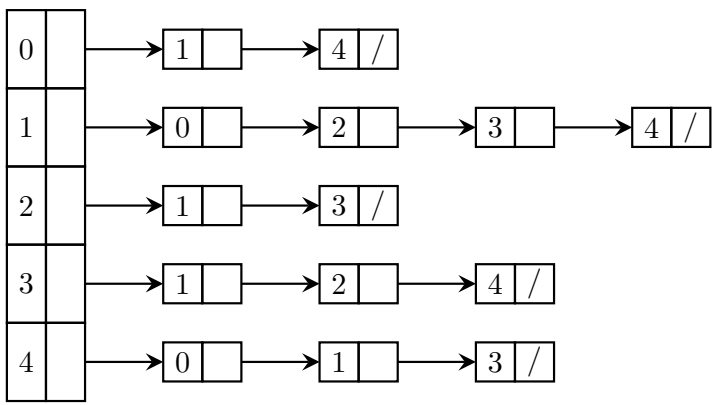
Grafy jako struktury danych

Istnieją dwa podstawowe sposoby reprezentowania grafów w pamięci komputera: za pomocą macierzy sąsiedztwa lub za pomocą listy sąsiedztwa. Lista sąsiedztwa może być zaimplementowana jako lista list lub jako tablica wskaźników na listy. Macierze sąsiedztwa to dwuwymiarowe tablice, na które pamięć jest przydzielana statycznie lub dynamiczne. Wiersze i kolumny w takiej macierzy reprezentują wierzchołki grafu. Jeśli dwa wierzchołki w grafie łączy krawędź, to w elemencie macierzy sąsiedztwa znajdującym się na przecięciu wiersza i kolumny odpowiadającym tym wierzchołkom zapisana jest liczba 1, w przeciwnym razie umieszczona tam jest wartość 0. Następne slajdy przedstawiają graf skierowany oraz nieskierowany i ich reprezentacje za pomocą listy sąsiedztwa oraz macierzy sąsiedztwa.

Reprezentacje grafu nieskierowanego



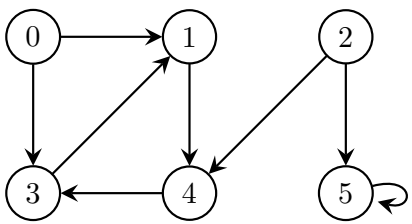
$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$



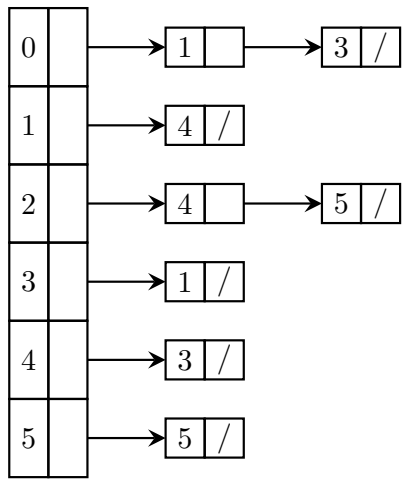
Reprezentacje grafu nieskierowanego

Po lewej stronie poprzedniego slajdu znajduje się ilustracja grafu nieskierowanego. W środku znajduje się macierz sąsiedztwa, a po prawej stronie lista sąsiedztwa zrealizowana jako tablica list. Znaki / wewnątrz elementów list oznaczają pola wskaźnikowe o wartości NULL. Proszę zwrócić uwagę, że macierz sąsiedztwa jest symetryczna względem głównej przekątnej, a więc zachodzi równość $\mathbb{A} = \mathbb{A}^T$, gdzie $\mathbb{A}$ to macierz sąsiedztwa. Skoro ta macierz jest równa swojej macierzy transponowanej, to można zaoszczędzić miejsce w pamięci operacyjnej przechowując tylko jej elementy z macierzy trójkątnej górnej (lub trójkątnej dolnej).

Reprezentacje grafu skierowanego



$$
\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\left[\begin{array}{cccccc}
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]
\end{array}
$$



Reprezentacje grafu skierowanego

Podobnie jak w przypadku grafu nieskierowanego na poprzednim slajdzie przedstawiono kolejno (od lewej do prawej): ilustrację grafu, macierz sąsiedztwa i listę sąsiedztwa. Macierz sąsiedztwa jest nadal macierzą kwadratową, ale nie jest już symetryczna. Proszę także zwrócić uwagę, że graf posiada jedną krawędź, która jest pętlą. Jest ona w macierzy reprezentowana przez jedynkę znajdującą się na przecięciu szóstej kolumny i szóstego wiersza.

Reprezentacje grafów

Podsumowanie

Statystycznie rzecz ujmując częściej stosowaną reprezentacją grafów w informatyce jest lista sąsiedztwa. Jest ona implementowana jako lista list lub tablica list. W tym wykładzie jest opisana tylko ta druga opcja. Każdy element takiej tablicy reprezentuje pojedynczy wierzchołek grafu i wskazuje na listę jego sąsiadów. Kolejność wierzchołków na tej liście nie ma znaczenia. Suma długości wszystkich list sąsiedztwa wynosi w przypadku grafu skierowanego $|E|$, a w przypadku grafu nieskierowanego $2 \cdot |E|$, gdzie zapis $|E|$ oznacza liczebność zbioru krawędzi grafu. Reprezentacja za pomocą listy sąsiedztwa wymaga zatem $O(|V| + |E|)$ pamięci, natomiast macierz sąsiedztwa wymaga $\Theta(|V|^2)$. Obie reprezentacje mogą być używane do reprezentowania zarówno grafów z wagami, jak i grafów bez wag. W tym ostatnim przypadku można zaoszczędzić pamięć potrzebną na macierz sąsiedztwa zapisując wartość każdego jej elementu na pojedynczym bicie. Jest to oszczędność pamięci kosztem czasu wykonania.

Reprezentacje grafów

Podsumowanie

Macierze sąsiedztwa lepiej się sprawdzają od list w zagadnieniach polegających na ustalaniu, czy istnieje krawędź między wierzchołkami grafu lub (pod warunkiem, że liczba wierzchołków jest niezmienna) na dodawaniu nowych krawędzi do grafu albo usuwaniu istniejących. Z kolei listy sąsiedztwa są bardziej przydatne w zagadnieniach związanych z przeszukiwaniem grafu (większość algorytmów grafowych wykonuje tę czynność) lub znajdywaniem stopnia wierzchołków. Również lepiej nadają się one do reprezentowania grafów małych lub rzadkich. Macierze sąsiedztwa są nieco lepszym rozwiązaniem, jeśli chcemy reprezentować w pamięci komputera gęste grafy.

Obie reprezentacje są wymienne, tzn. macierz sąsiedztwa może zostać zastąpiona listą sąsiedztwa i odwrotnie. Na kolejnych slajdach przedstawiony jest program, który konwertuje zaprezentowaną wcześniej macierz sąsiedztwa grafu nieskierowanego na listę sąsiedztwa.

Algorytm przeszukiwania w głąb

Są dwa podstawowe algorytmy przeszukiwania grafu, algorytm przeszukiwania w głąb (ang. *Depth-First Search Algorithm*, DFS) i algorytm przeszukiwania wszerz (ang. *Breadth-First Search*, BFS). Algorytm DFS rozpoczyna przeszukiwanie grafu od wskazanego *wierzchołka początkowego*, odkrywa jego *pierwszego nieodwiedzonego sąsiada*, oznacza bieżący wierzchołek jako *odwiedzony* i przechodzi do wspomnianego odkrytego sąsiada. Algorytm DFS powtarza te kroki aż napotka wierzchołek, który albo *nie ma są sąsiadów* albo *wszyscy jego sąsiedzi zostali już odwiedzeni*. W takim przypadku DFS *zawraca* (ang. *backtracks*) od poprzedniego wierzchołka i sprawdza, czy ten wierzchołek ma jeszcze innych nieodwiedzonych sąsiadów. Jeśli tak, to przechodzi do pierwszego z nich, w przeciwnym przypadku cofa się jeszcze dalej. Ostatecznie, DFS odwiedzi wszystkie wierzchołki w grafie. Wynikiem jest sekwencja odwiedzonych przez niego wierzchołków.

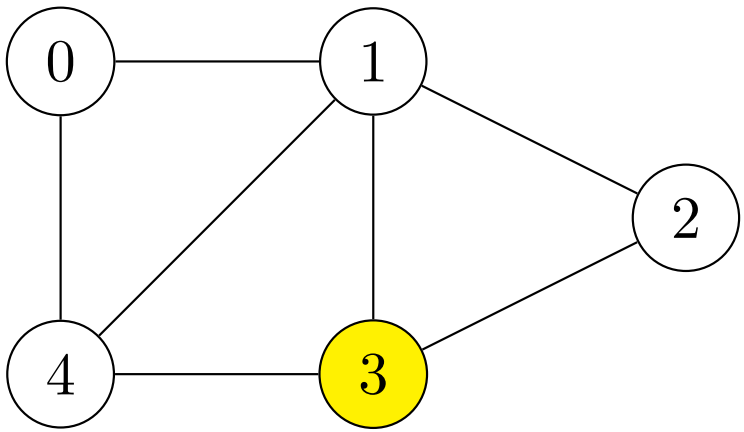
Algorytm przeszukiwania w głąb

Algorytm DFS jest uogólnieniem algorytmu przechodzenia drzewa w porządku preorder. Jest on również powiązany z algorytmami z nawrotami (ang. *backtracking algorithms*). Algorytm przeszukiwania w głąb używa stosu, zatem można go zaimplementować przy pomocy rekurencji. Animacja na następnym slajdzie obrazuje przeszukiwanie przykładowego grafu nieskierowanego przy pomocy DFS.

Algorytm przeszukiwania w głąb

Animacja

wynik:



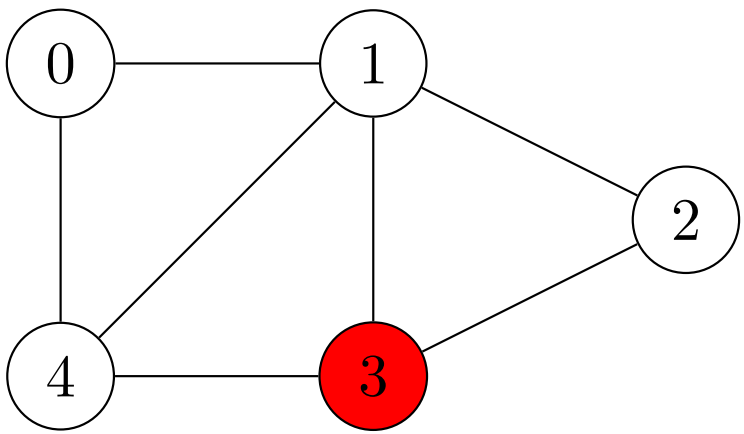
stos:

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

Algorytm przeszukiwania w głąb

Animacja

wynik: 3



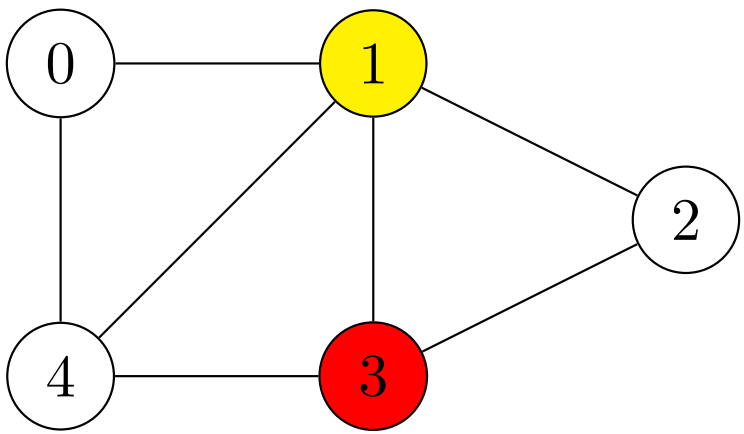
stos: 3

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

Algorytm przeszukiwania w głąb

Animacja

wynik: 3



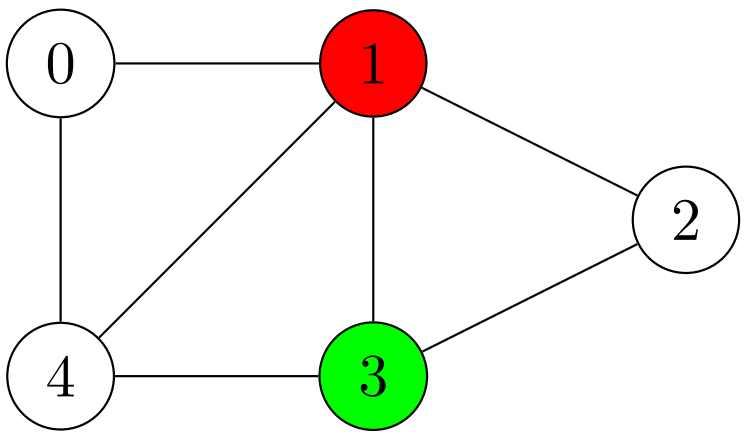
stos: 3

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

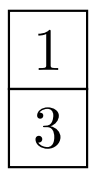
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1



stos:

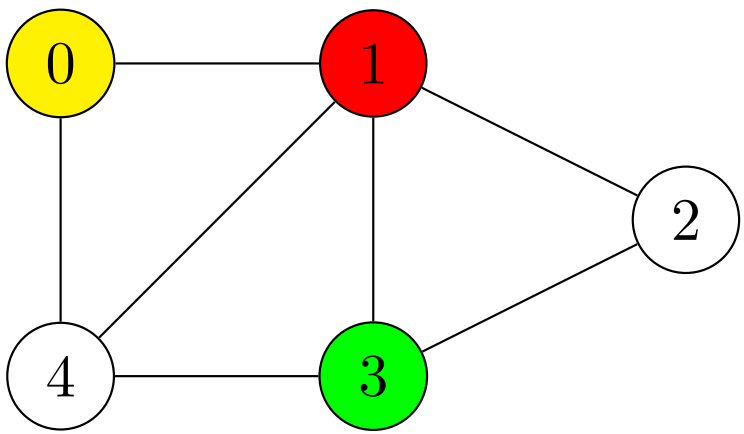


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

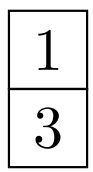
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1



stos:

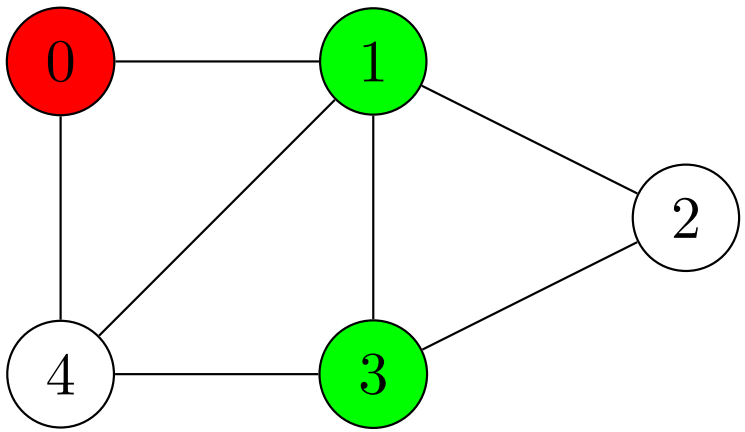


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

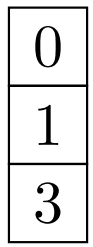
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0



stos:

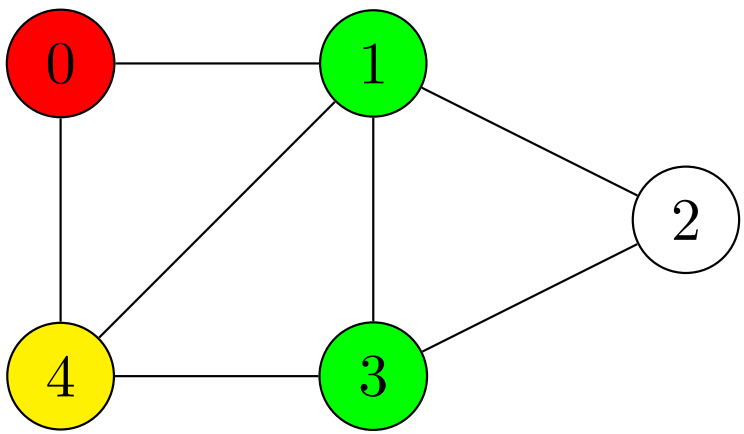


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

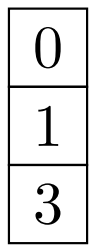
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0



stos:

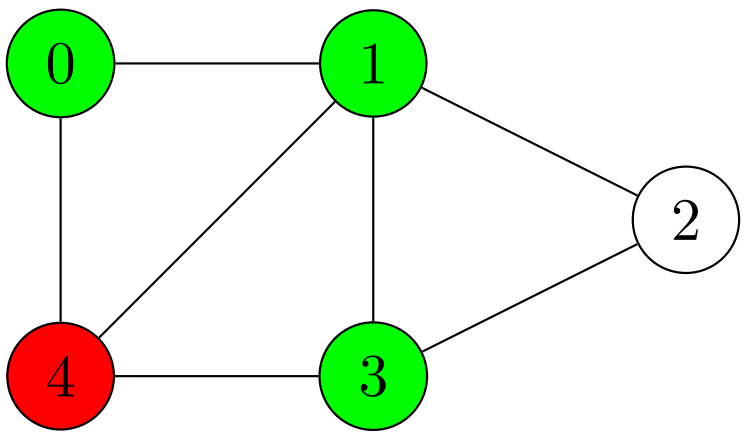


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

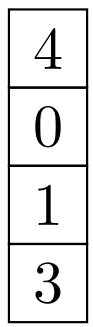
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4



stos:

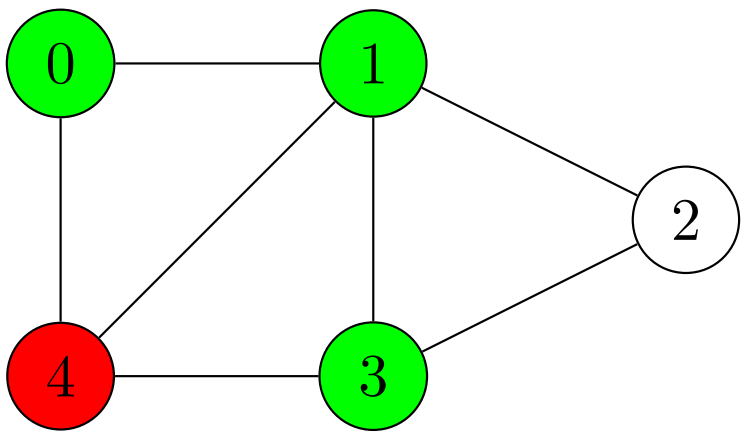


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

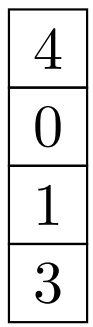
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4



stos:

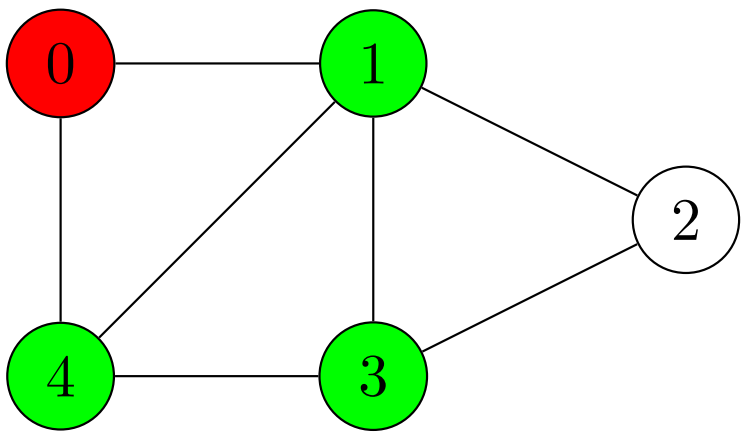


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

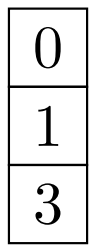
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4



stos:

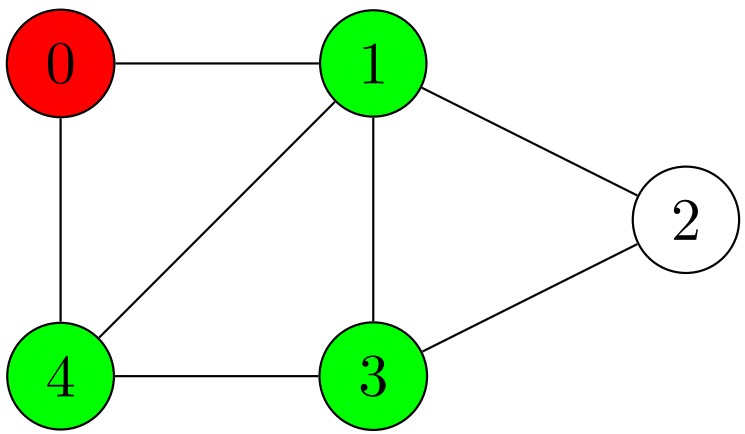


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

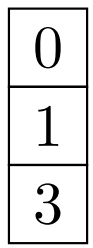
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4



stos:

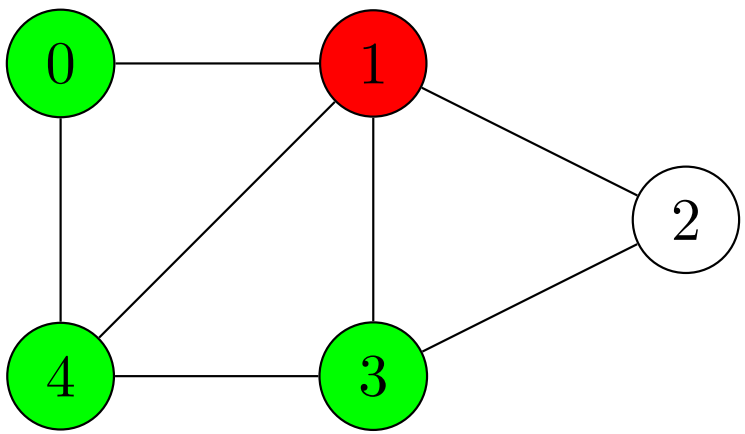


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

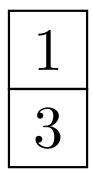
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4



stos:

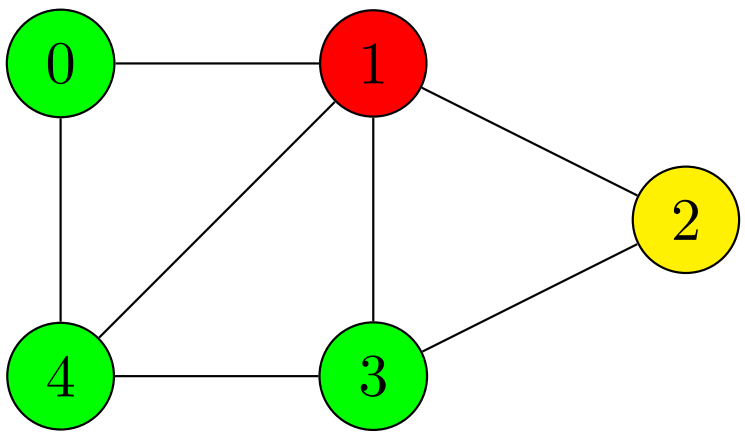


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

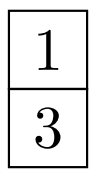
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4



stos:

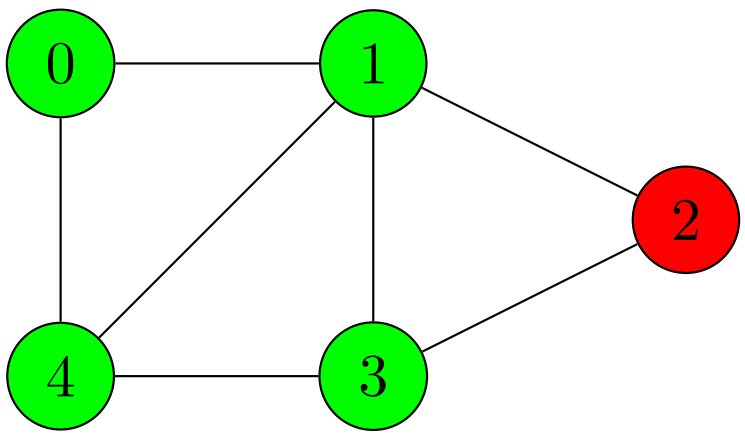


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

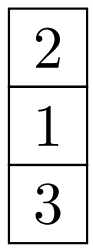
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4 2



stos:

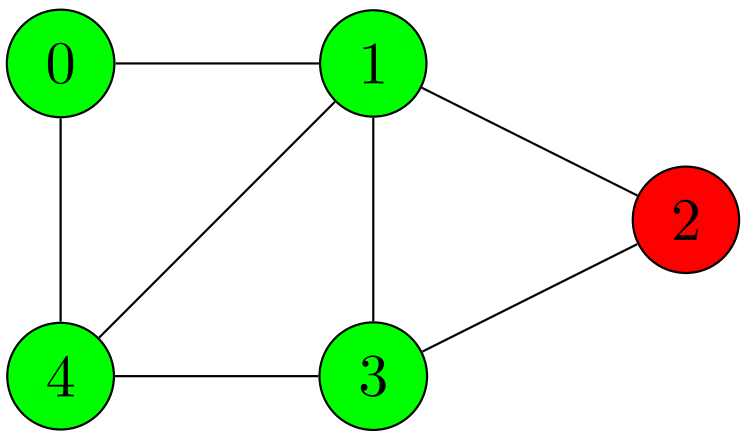


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

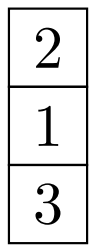
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4 2



stos:

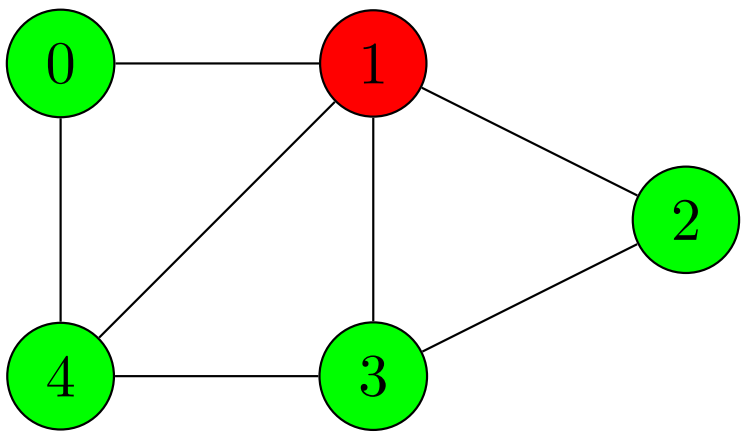


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

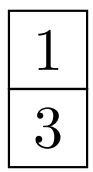
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4 2



stos:

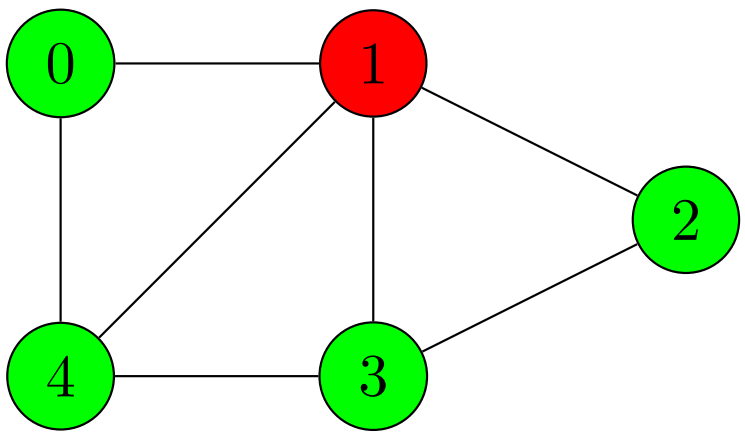


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

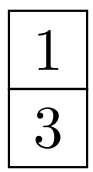
Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4 2



stos:

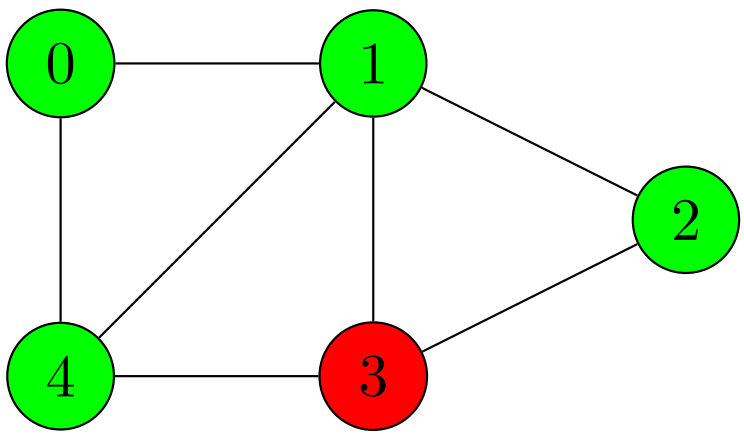


Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4 2



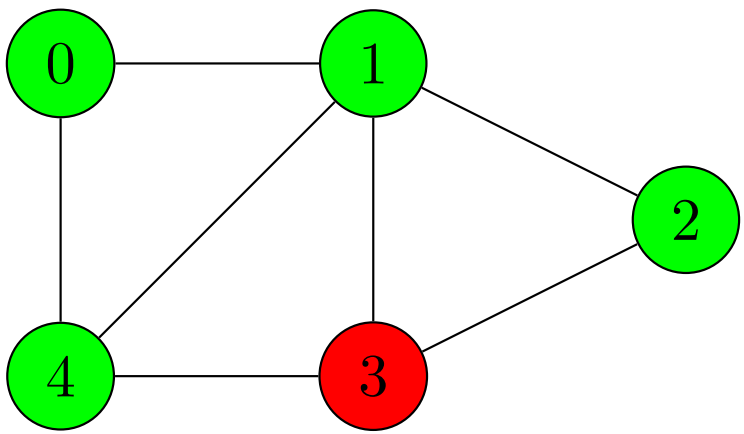
stos: 3

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4 2



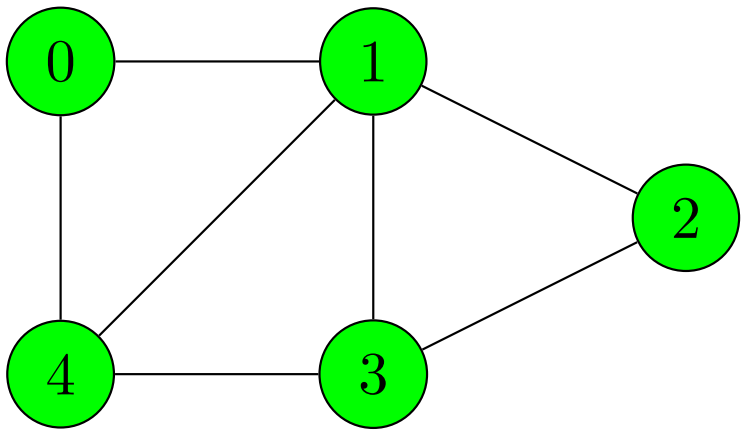
stos: 3

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

Algorytm przeszukiwania w głąb

Animacja

wynik: 3 1 0 4 2



stos:

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu DFS.

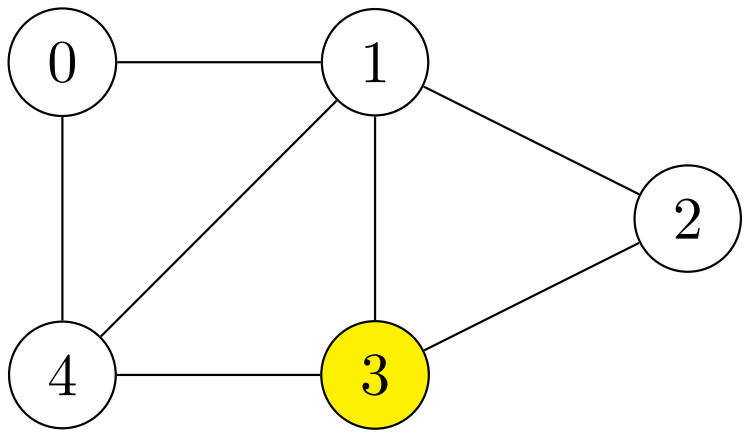
Przeszukiwanie grafu wszerz

Algorytm BFS jest podobny do DFS, ale odwiedzając dany wierzchołek, odkrywa *wszystkich jego nieodwiedzonych* sąsiadów, a potem po kolei przechodzi przez nich, zaznaczając ich jako odwiedzonych i odkrywając ich sąsiadów. Nie stosuje zawracania. Zamiast stosu używa kolejki FIFO. Animacja na następnym slajdzie pokazuje przeszukiwanie tego samego nieskierowanego grafu, co w przypadku algorytmu DFS, ale tym razem w oparciu o algorytm BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik:



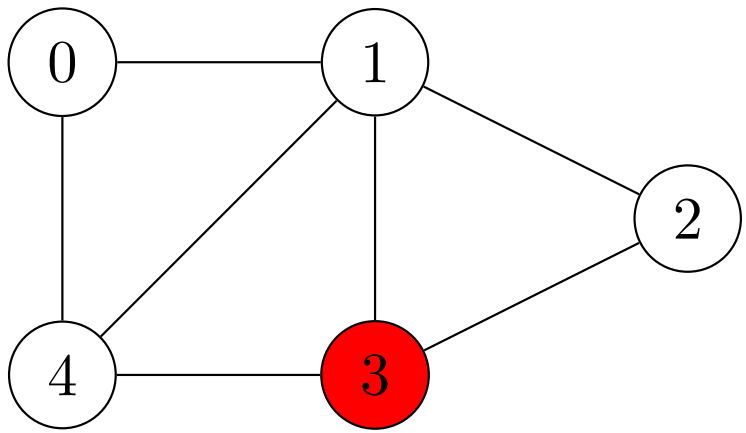
kolejka FIFO: 3

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik:



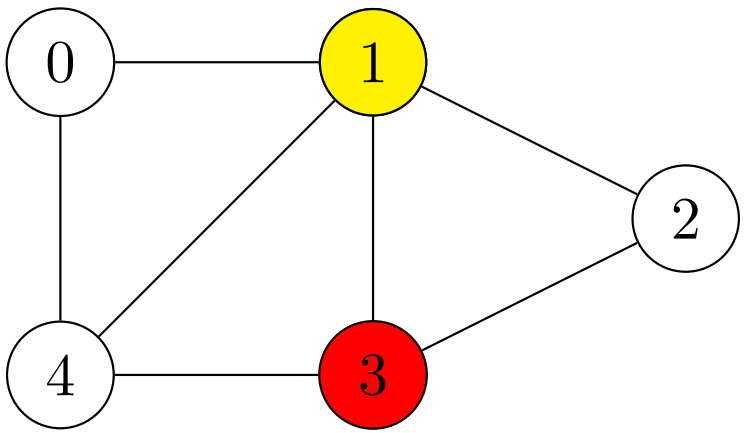
kolejka FIFO: 3

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik:



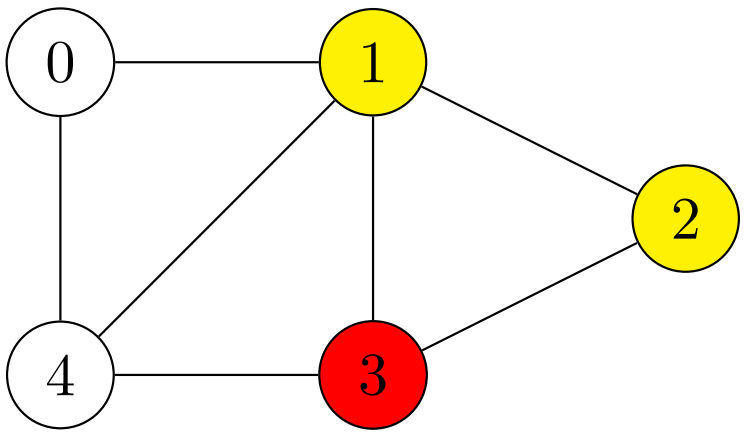
kolejka FIFO: 1

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik:



kolejka FIFO:

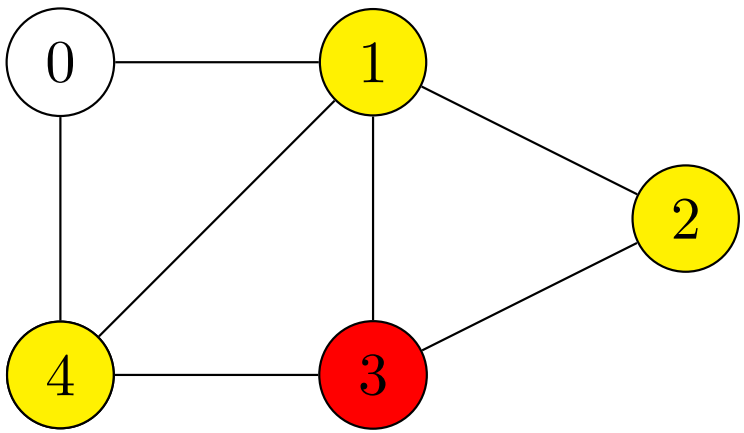
1	2

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik:



kolejka FIFO:

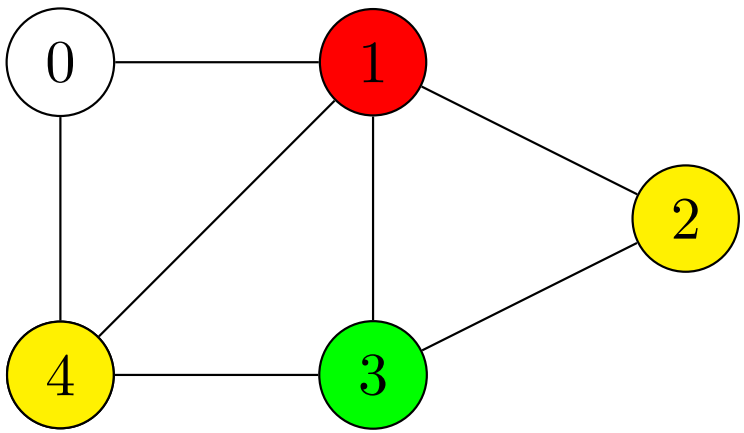
1	2	4

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3



kolejka FIFO:

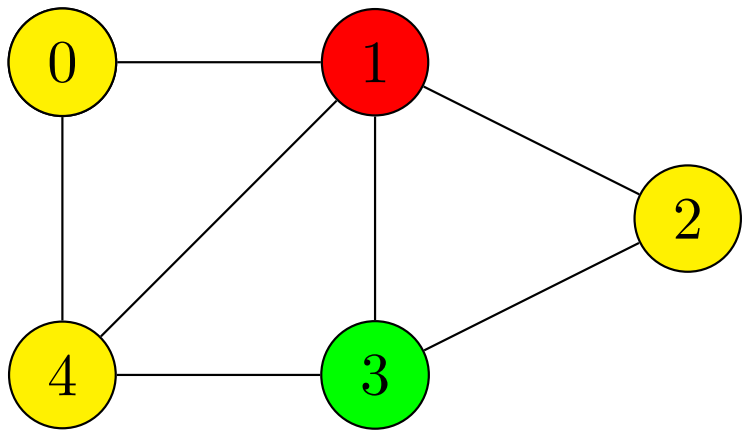
1	2	4

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3



kolejka FIFO:

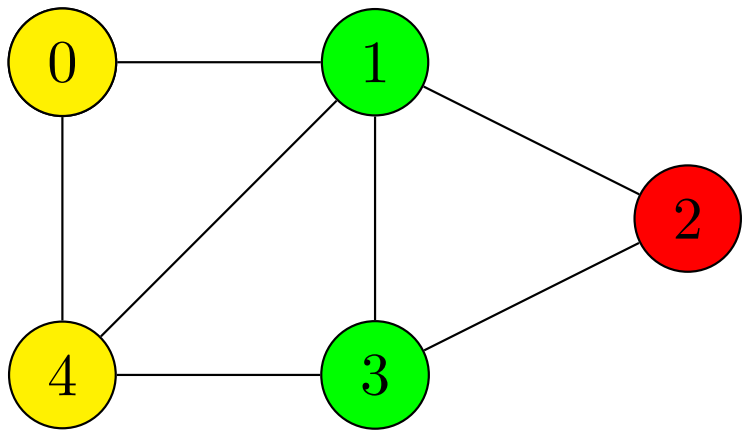
2	4	0

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1



kolejka FIFO:

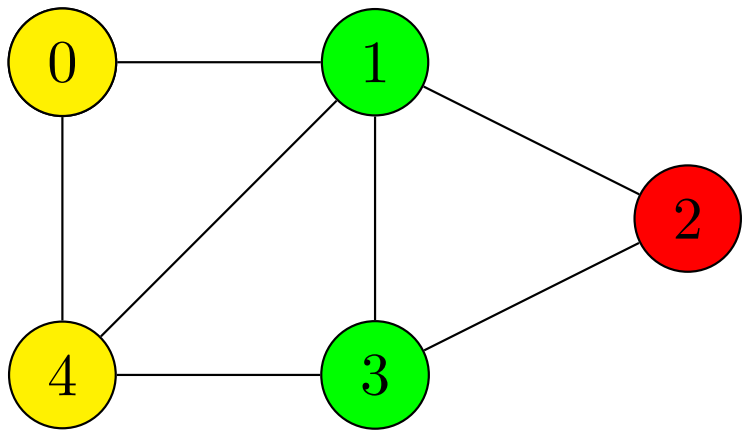
2	4	0

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1



kolejka FIFO:

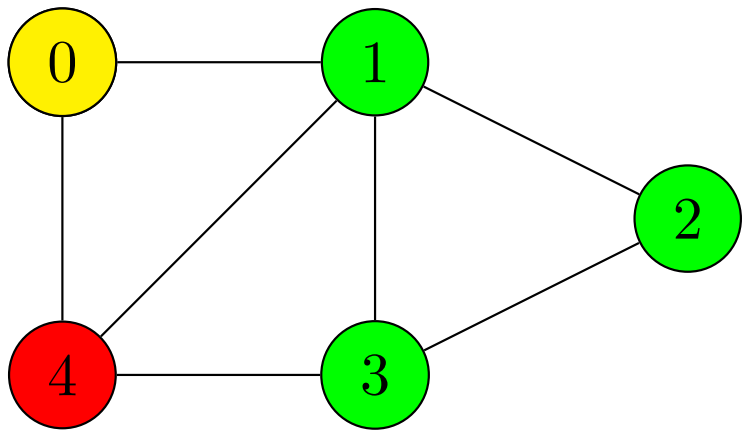
4	0

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1 2



kolejka FIFO:

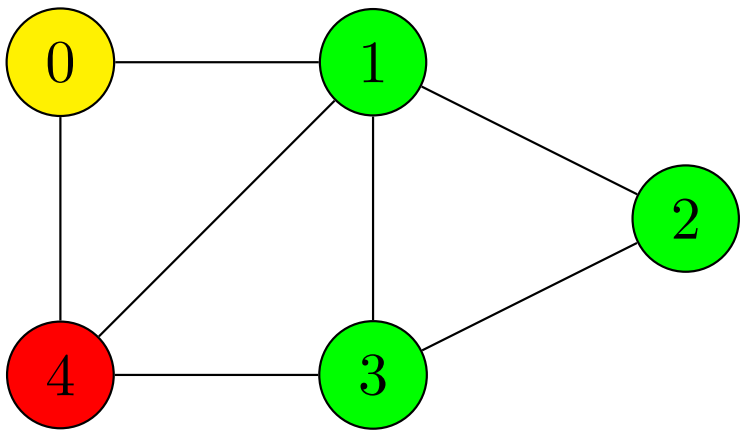
4	0

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1 2



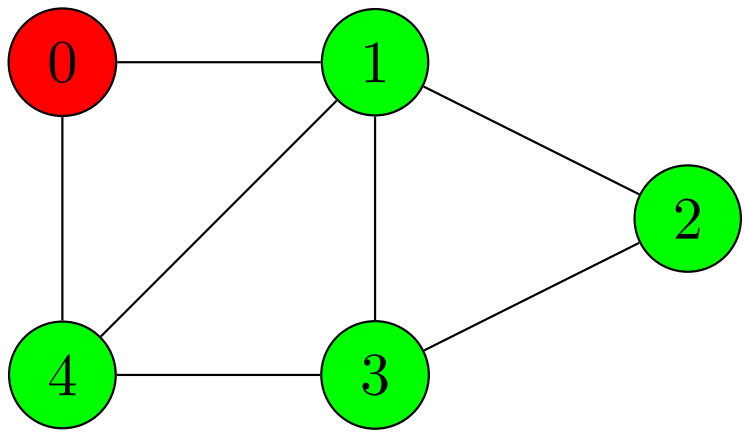
kolejka FIFO: 0

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1 2 4



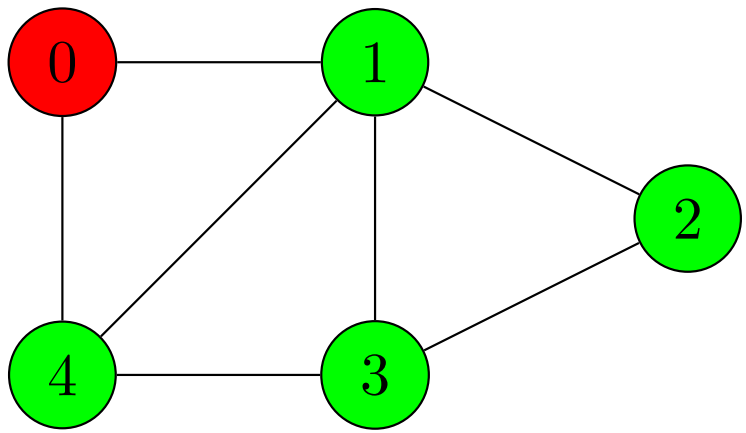
kolejka FIFO: 0

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1 2 4



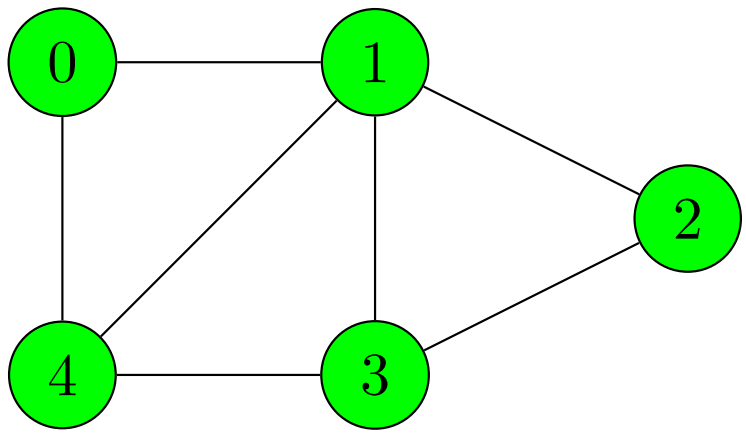
kolejka FIFO:

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1 2 4 0



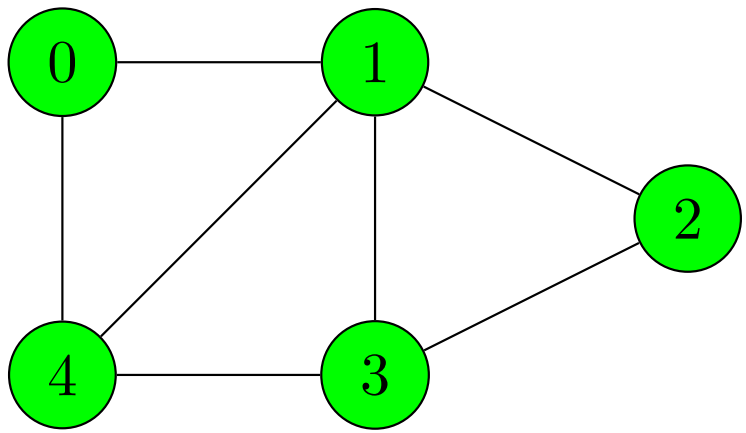
kolejka FIFO:

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Przeszukiwanie grafu wszerz

Animacja

wynik: 3 1 2 4 0



kolejka FIFO:

Przeszukiwanie nieskierowanego grafu z użyciem algorytmu BFS.

Uwagi

Oba algorytmy mogą być zastosowane zarówno do grafów skierowanych, jak i nieskierowanych. Jeśli zostaną wykonane dla grafu spójnego lub silnie spójnego, to odwiedzą wszystkie jego wierzchołki. W przypadku grafów niespójnych odwiedzą tylko te wierzchołki, które są osiągalne z wierzchołka startowego. Innymi słowy, przeszukają tylko *spójną* lub *silnie spójną składową* tego grafu. Każdy z tych algorytmów, aby odwiedzić wszystkie wierzchołki , powinien po zakończeniu przeszukiwania grafu sprawdzić, czy jest jeszcze jakiś wierzchołek nieodwiedzony i ponownie przeszukać graf, zaczynając od niego.

Celem DFS i BFS nie musi być koniecznie odwiedzenie wszystkich wierzchołków. Mogą one zostać użyte do znalezienia ścieżki prowadzącej od wierzchołka startowego, do określonego *wierzchołka docelowego* (ang. *goal vertex*) lub do wierzchołka, który spełnia *warunek celu* (ang. *goal vertex*).

Implementacja kolejki FIFO

Kolejka FIFO, która jest wymagana do realizacji algorytmu BFS, została zaimplementowana w postaci biblioteki dołączanej statycznie, która składa się z pliku nagłówkowego `queue.h` i pliku źródłowego `queue.c`. Zawartość pierwszego z nich przedstawiono na następnym slajdzie.

Plik queue.h

```
1  #ifndef GRAPHS_QUEUE_H
2  #define GRAPHS_QUEUE_H
3  struct fifo_node
4  {
5      int vertex_number;
6      struct fifo_node *next;
7  };
8
9  struct fifo_pointers
10 {
11     struct fifo_node *head, *tail;
12 };
13
14 void enqueue(struct fifo_pointers *, int);
15 int dequeue(struct fifo_pointers*);
16 #endif
```

Plik queue.h

Na początku pliku nagłówkowego znajduje się dyrektywa `#ifndef` (wiersz nr 1), która jest częścią *wartownika nagłówka* (ang. *header guard*) (wiersze 1, 2 i 16). Kiedy jest ona interpretowana przez preprocesor, to nakazuje mu sprawdzić, czy w kodzie programu **nie** został już zdefiniowany znacznik `GRAPH_QUEUE_H`. Jeśli nie został, to preprocesor skopiuje wiersze 2–15 z pliku nagłówkowego do programu. Proszę zauważyć, że dyrektywa z wiersza nr 2 definiuje wspomniany znacznik. Oznacza to, że nawet jeśli plik nagłówkowy zostanie wielokrotnie włączony (za pomocą dyrektywy `#include`), to preprocesor tylko raz doda jego zawartość w programie.
W pliku nagłówkowym są zdefiniowane typy węzła kolejki (wiersze 3–7) i struktury jej wskaźników (wiersze 9–12). Każdy węzeł w tej kolejce będzie przechowywał numer wierzchołka (wiersz nr 5). Plik zawiera również definicje prototypów funkcji `enqueue()` i `dequeue()`. Proszę zwrócić uwagę, że w prototypach nazwy parametrów funkcji nie muszą być określone.

Funkcja enqueue() — plik queue.c

```
1  #include"queue.h"
2  #include<stdlib.h>
3
4  void enqueue(struct fifo_pointers *queue, int vertex_number)
5  {
6      struct fifo_node *new_node = (struct fifo_node
   ↪  *)malloc(sizeof(struct fifo_node));
7      if(new_node) {
8          new_node->vertex_number = vertex_number;
9          new_node->next = NULL;
10         if(queue->head==NULL && queue->tail==NULL)
11             queue->head = queue->tail = new_node;
12         else {
13             queue->tail->next = new_node;
14             queue->tail = new_node;
15         }
16     }
17 }
```

Funkcja `enqueue()` — plik `queue.c`

Plik nagłówkowy `queue.h` włączany jest do pliku `queue.c` (wiersz nr 1). To pozwala kompilatorowi sprawdzić, czy nagłówki funkcji, zdefiniowanych w drugim z wymienionych plików, są zgodne z ich prototypami. Plik nagłówkowy `stdlib.h` również jest dołączany (wiersz nr 2), ponieważ `enqueue()` i `dequeue()` wywołują funkcje odpowiedzialne za alokację i zwalnianie pamięci na stercie.
Funkcja `enqueue()` jest zdefiniowana podobnie jak jej odpowiedniczka z trzeciego wykładu, ale ona nie zwraca żadnej wartości, za to pobiera jako drugi argument numer wierzchołka, które jest przekazywany do niej przez parameter `vertex_number`.

Funkcja dequeue() — plik queue.c

```
1  int dequeue(struct fifo_pointers *queue)
2  {
3      int vertex_number = -1;
4      if(queue->head) {
5          vertex_number = queue->head->vertex_number;
6          struct fifo_node *temporary = queue->head->next;
7          free(queue->head);
8          queue->head = temporary;
9          if(temporary==NULL)
10             queue->tail = NULL;
11     }
12     return vertex_number;
13 }
```

Funkcja `dequeue()` — plik `queue.c`

Funkcja `dequeue()` również jest zdefiniowana podobnie jak jej odpowiedniczka z trzeciego wykładu, ale zwraca numer wierzchołka, który był przechowywany w usuniętym węźle. Ma ona tylko jeden parameter (wiersz nr 1), przez który przekazywany jest adres struktury wskaźników kolejki. Numer wierzchołka przechowywany w węźle do usunięcia jest przypisywany do lokalnej zmiennej o nazwie `vertex_number` (wiersz nr 5). Jej początkową wartością jest `-1` (wiersz nr 3). Gdyby `dequeue()` została wywołana dla pustej kolejki FIFO, to zwróciłaby właśnie tę liczbę. Jednakże, jeżeli algorytm BFS jest zaimplementowany prawidłowo, to taka sytuacja nie powinna się nigdy zdarzyć.

Macierz sąsiedztwa

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4  #include"queue.h"
5
6  #define NUMBER_OF_VERTICES 5
7
8  typedef int
    ↪  matrix[NUMBER_OF_VERTICES][NUMBER_OF_VERTICES];
9
10 const matrix adjacency_matrix = {
11     {0,1,0,0,1},
12     {1,0,1,1,1},
13     {0,1,0,1,0},
14     {0,1,1,0,1},
15     {1,1,0,1,0}
16 };
```

Macierz sąsiedztwa

Do programu są dołączane cztery pliki nagłówkowe. Plik `stdio.h` (wiersz nr 1) zawiera prototypy funkcji `scanf()` i `printf()`, których program używa do komunikacji z użytkownikiem. W pliku nagłówkowym `stdlib.h` (wiersz nr 2) są zadeklarowane funkcje odpowiedzialne za alokowanie i zwalnianie pamięci na stercie. Proszę zauważyć, że ten plik jest włączany tylko raz do programu, mimo że w pliku `queue.c` również występuje dyrektywa go dodająca. Dzieje się tak, ponieważ `stdlib.h` ma własnego wartownika nagłówka. Program korzysta również z typu `bool` i jego wartości, stąd dołączany jest do niego plik `stdbool.h` (wiersz nr 3). W wierszu nr 4 włączany jest plik nagłówkowy `queue.h` biblioteki dostarczającej implementacji kolejki FIFO.

Stała `NUMBER_OF_VERTICES` określa liczbę wierzchołków w grafie. Jest ona używana w definicji typu macierzy sąsiedztwa (wiersz nr 8). Stała o nazwie `adjacency_matrix`, która reprezentuje macierz sąsiedztwa jest zdefiniowana i zainicjowana w wierszach 10–16.

Definicje typów

```
1  struct vertex
2  {
3      int vertex_number;
4      struct vertex *next;
5  };
6
7  struct vertices_array
8  {
9      bool visited;
10     struct vertex *neighbours;
11 } *adjacency_list;
12
13 typedef void (*algorithm_pointer)(struct vertices_array
    ↪  *, int);
```

Definicje typów

Typ węzła listy sąsiedztwa (wiersze 1–5) bazuje na strukturze, która posiada dwie składowe. Pierwsza jest używana do przechowywania numeru wierzchołka (wiersz nr 3), a druga jest wskaźnikiem na następny węzeł listy. Typ `struct vertices_array` (wiersze 7–11) jest typem elementów tablicy wierzchołków. Każdy z nich jest strukturą z dwiema składowymi. Pierwsza (wiersz nr 9), o nazwie `visited`, jest używana przez funkcje implementujące algorytmy DFS i BFS. Jej wartość określa, czy wierzchołek został już odwiedzony. Druga, nazwana `neighbours`, jest wskaźnikiem na listę sąsiadów danego wierzchołka, a dokładniej na pierwszy węzeł tej listy. Jeśli wierzchołek nie ma sąsiadów, to ta składowa jest pustym wskaźnikiem. Zmienna globalna `adjacency_list` (wiersz nr 11) jest wskaźnikiem na listę sąsiedztwa, a dokładniej na tablicę wierzchołków, która jest częścią tej listy. Początkowo wspomniany wskaźnik jest pusty, ponieważ lista jest tworzona dynamicznie. Typ wskaźnika na funkcję zdefiniowany w wierszu nr 13 jest używany w deklaracji parametru jednej z funkcji zdefiniowanych w programie.

Funkcja convert()

```
1  struct vertices_array *convert(const matrix adjacency_matrix)
2  {
3      struct vertices_array *list = NULL;
4      list = (struct vertices_array *)calloc(NUMBER_OF_VERTICES, sizeof(struct
   ↪  vertices_array));
5      if(list) {
6          for(int i=0; i < NUMBER_OF_VERTICES; i++) {
7              struct vertex **new_vertex = &list[i].neighbours;
8              for (int j = 0; j < NUMBER_OF_VERTICES; j++) {
9                  if (adjacency_matrix[i][j]) {
10                     *new_vertex = (struct vertex *)malloc(sizeof(struct
   ↪  vertex));
11                     if(*new_vertex) {
12                         (*new_vertex)->vertex_number = j;
13                         (*new_vertex)->next = NULL;
14                         new_vertex = &(*new_vertex)->next;
15                     } else fprintf(stderr, "Wyjątek przy tworzeniu
   ↪  wierzchołka %d.\n",j);
16                 }
17             }
18         }
19     }
20     return list;
21 }
```

Funkcja `convert()`

Funkcja `convert()` jako argument, przekazywany przez stałą, przyjmuje macierz sąsiedztwa, a zwraca adres listy sąsiedztwa[1] (wiersz nr 1). Najpierw próbuje ona przydzielić pamięć na tablicę wierzchołków (wiersz nr 4) i przypisuje wartość zwracaną przez funkcję `calloc()` do wskaźnika lokalnego `list` (wiersz nr 3). Następnie sprawdza, czy przydział się powiódł (wiersz nr 5). Jeśli tak, to wykonywane są dwie pętle `for`, które iterują po macierzy sąsiedztwa. Licznik zewnętrznej pętli (wiersz nr 6) jest używany jako indeks w tablicy wierzchołków i jako indeks wiersza w macierzy sąsiedztwa. W tej pętli funkcja `convert()` z elementu tablicy wierzchołków, określanego przez zmienną `i` pobiera adres pola `neighbours` i przypisuje go do lokalnego wskaźnika na wskaźnik, o nazwie `new_node` (wiersz nr 7). W wewnętrznej pętli (wiersz nr 8) iteruje ona po wszystkich elementach wiersza macierzy sąsiedztwa, który również jest określany przez zmienną `i`.

[1] Bardziej precyzyjnie, to zwraca adres tablicy wierzchołków, będącej częścią listy sąsiedztwa.

Funkcja `convert()`

Jeśli wartość elementu tego wiersza, określanego przez zmienną `j`, wynosi 1 (wiersz nr 9), to funkcja próbuje przydzielić pamięć na nowy węzeł listy sąsiadów wierzchołka o numerze `i` (wiersz nr 10). Jeśli ten przydział się powiedzie, co jest sprawdzane w wierszu nr 11, to `convert()` zainicjuje nowy węzeł poprzez przypisanie do jego składowej `vertex_number` numeru sąsiedniego wierzchołka, który znajduje się w zmiennej `j` (wiersz nr 12) i poprzez nadanie składowej `next` wartości `NULL` (wiersz nr 13). Na koniec przypisuje do wskaźnika na wskaźnik `new_node` adres pola `next` nowego węzła. Jeśli przydział pamięci w wierszu nr 10 by się nie powiódł, to funkcja wyświetli odpowiedni komunikat używając standardowego wyjścia diagnostycznego (wiersz nr 15). W opisany sposób `convert()` tworzy listy sąsiadów dla każdego z wierzchołków grafu. Kiedy obie pętle się zakończą ta funkcja zwróci adres listy sąsiedztwa (wiersz nr 20) i również zakończy działanie.

Funkcja print_adjacency_list()

```
1  void print_adjacency_list(struct vertices_array
   ↪  *adjacency_list)
2  {
3      for(int i=0; i < NUMBER_OF_VERTICES; i++) {
4          printf("Wierzchołek %d ma sąsiadów: ",i);
5          struct vertex *neighbour =
   ↪  adjacency_list[i].neighbours;
6          while(neighbour) {
7              printf("%3d", neighbour->vertex_number);
8              neighbour = neighbour->next;
9          }
10         puts("");
11     }
12 }
```

Funkcja `print_adjacency_list()`

Funkcja `print_adjacency_list()` wyświetla listę sąsiedztwa grafu. Nie zwraca ona żadnej wartości, a jako argument pobiera adres listy. W pętli `for` funkcja wyświetla wiadomość informującą, że będzie wypisywała na ekranie sąsiadów wierzchołka o numerze określanym licznikiem pętli (wiersz nr 4). Następnie, zapisuje adres przechowywany w polu `neighbours` elementu tablic wierzchołków, określanym przez zmienną `i`, w lokalnym wskaźniku `neighbour` (wiersz nr 5). W pętli `while` (wiersze 6–9) przechowywane w liście numery sąsiednich wierzchołków są wypisywane na ekranie. Warto zwrócić uwagę, że jeśli bieżący wierzchołek, określony zmienną `i`, nie ma sąsiadów, to pętla `while` nie będzie dla niego wykonana. Po zakończeniu tej pętli wywoływana jest funkcja `puts()`, celem przemieszczenia kursora do następnego wiersza ekranu. Funkcja `for` kończy działanie, po przetworzeniu ostatniego elementu w tablicy wierzchołków.

Funkcja dfs()

```
1  void dfs(struct vertices_array *adjacency_list, int
   ↪  vertex_number)
2  {
3      printf("%3d", vertex_number);
4      struct vertex *neighbour =
   ↪  adjacency_list[vertex_number].neighbours;
5      adjacency_list[vertex_number].visited = true;
6      while(neighbour) {
7          if(!adjacency_list[neighbour->vertex_number].visited)
8              dfs(adjacency_list, neighbour->vertex_number);
9          neighbour = neighbour->next;
10     }
11 }
```

Funkcja `dfs()`

Funkcja `dfs()` implementuje algorytm przeszukiwania grafu w głąb. Pobiera ona dwa argumenty, listę sąsiedztwa (przekazywaną przez pierwszy parametr) i numer wierzchołka początkowego (przekazywany przez drugi parametr). Funkcja nie zwraca żadnej wartości. Najpierw wyświetla ona numer bieżąco odwiedzanego wierzchołka (wiersz nr 3) i przypisuje do lokalnego wskaźnika `neighbour` adres przechowywany w składowej `neighbours` elementu reprezentującego ten wierzchołek w tablicy wierzchołków (wiersz nr 4). Następnie funkcja oznacza bieżący wierzchołek jako odwiedzony (wiersz nr 5). Pętla `while` (wiersze nr 6–10) iteruje po liście sąsiadów tego wierzchołka (jeśli oni istnieją). Sprawdza ona, czy bieżący sąsiad **nie** został jeszcze odwiedzony (wiersz nr 7). Jeśli ten warunek jest spełniony, to wywołuje rekurencyjnie funkcję `dfs()`, przekazując jej jako argumenty listę sąsiedztwa i numer tego sąsiada (wiersz nr 8). Pętla `while` kończy działanie, kiedy na liście nie będzie żadnego sąsiada, który byłby nieodwiedzony.

Funkcja bfs()

```
1  void bfs(struct vertices_array *adjacency_list, int vertex_number)
2  {
3      struct fifo_pointers queue;
4      queue.head = queue.tail = NULL;
5      enqueue(&queue,vertex_number);
6      while(queue.head) {
7          vertex_number = dequeue(&queue);
8          if(!adjacency_list[vertex_number].visited) {
9              struct vertex *neighbour =
   ↪   adjacency_list[vertex_number].neighbours;
10             while(neighbour) {
11                 enqueue(&queue,neighbour->vertex_number);
12                 neighbour = neighbour->next;
13             }
14             printf("%3d", vertex_number );
15             adjacency_list[vertex_number].visited = true;
16         }
17     }
18 }
```

Funkcja `bfs()`

Funkcja `bfs()` implementuje algorytm przeszukiwania grafu wszerz. Przyjmuje ona te same argumenty jak `dfs()` i również nie zwraca żadnej wartości. W wierszu nr 4 funkcja inicjuje wskaźniki kolejki FIFO. Struktura tych wskaźników jest zadeklarowana w wierszu nr 3. Następnie, funkcja dodaje do kolejki pierwszy węzeł, który przechowuje numer wierzchołka startowego (wiersz nr 5). Zewnętrzna pętla `while` (wiersze 6–17) sprawdza, czy kolejka **nie** jest pusta (wiersz nr 6). Jeśli ten warunek jest spełniony, to `bfs()` przypisuje zmiennej lokalnej `vertex_number` numer wierzchołka przechowywany w pierwszym węźle kolejki i usuwa ten węzeł (wiersz nr 7). Potem, sprawdza, czy ten wierzchołek **nie** był już odwiedzony (wiersz nr 8). Jeśli nie był, to pętla przypisuje adres jego listy sąsiadów do lokalnego wskaźnika `neighbour` (wiersz nr 9). Wewnętrzna pętla `while` (wiersze 10–13) iteruje po tej liście, jeśli nie jest ona pusta, i dodaje do kolejki FIFO węzły, które przechowują numery sąsiadów bieżącego wierzchołka.

Funkcja `bfs()`

Kiedy wewnętrzna pętla się kończy, to zewnętrzna wyświetla numer bieżącego wierzchołka (wiersz nr 14) i oznacza go jako odwiedzony (wiersz nr 15). Funkcja `bfs()` kończy działanie, kiedy zatrzymuje się zewnętrzna pętla `while`.

Funkcja visit_all_vertices()

```
1  void visit_all_vertices(struct vertices_array
    ↪  *adjacency_list, int start_vertex,
    ↪  algorithm_pointer algorithm)
2  {
3      if(start_vertex>=0 && start_vertex <
    ↪  NUMBER_OF_VERTICES) {
4          algorithm(adjacency_list, start_vertex);
5          for (int i = 0; i < NUMBER_OF_VERTICES; i++)
6              if (!adjacency_list[i].visited)
7                  algorithm(adjacency_list, i);
8      } else
9          puts("Zły numer wierzchołka początkowego.");
10 }
```

Funkcja `visit_all_vertices()`

Funkcje `dfs()` i `bfs()` nie odwiedzą wszystkich wierzchołków grafu, jeśli nie jest on spójny. Z tego powodu została zdefiniowana funkcja `visit_all_vertices()`. Nie zwraca ona żadnej wartości, ale przyjmuje trzy argumenty. Pierwszym jest lista sąsiedztwa, drugim numer wierzchołka początkowego, a trzecim adres funkcji, która implementuje algorytm DFS lub BFS. Ostatni argument jest przekazywany przez parametr `algorithm`, będący wskaźnikiem na funkcję (wiersz nr 1). Jego typ jest zdefiniowany na początku programu. Funkcja najpierw sprawdza, czy numer wierzchołka początkowego jest prawidłowy (wiersz nr 3), a potem wywołuje funkcję implementującą algorytm przeszukiwania grafu, używając do tego wskaźnika na funkcję (wiersz nr 4). Kiedy ta funkcja zakończy działanie, `visit_all_vertices()` sprawdza w pętli `for`, czy został jeszcze jakiś nieodwiedzony wierzchołek grafu i jeśli tak, to wywołuje dla niego funkcję implementującą przeszukiwanie grafu. Po zakończeniu tej pętli w grafie nie będzie nieodwiedzonych wierzchołków.

Funkcja `visit_all_vertices()`

Jeśli numer wierzchołka początkowego jest nieprawidłowy, to funkcja `visit_all_vertices` wyświetla odpowiedni komunikat.

Funkcja remove_adjacency_list()

```
1  struct vertices_array *remove_adjacency_list(struct
    ↪  vertices_array *adjacency_list)
2  {
3      for (int i = 0; i < NUMBER_OF_VERTICES; i++) {
4          struct vertex *neighbour =
    ↪  adjacency_list[i].neighbours;
5          while(neighbour) {
6              struct vertex *temporary = neighbour->next;
7              free(neighbour);
8              neighbour = temporary;
9          }
10     }
11     free(adjacency_list);
12     return NULL;
13 }
```

Funkcja `remove_adjacency_list()`

Funkcja `remove_adjacency_list()` odpowiada za zwolnienie pamięci na stercie przydzielonej dla listy sąsiedztwa. Przyjmuje jako argument adres wskaźnika listy i zwraca `NULL`. Ta wartość jest zapisywana do wskaźnika listy sąsiedztwa. W pętli `for` (wiersze 3–10) funkcja odwiedza poszczególne elementy tablicy wierzchołków i w pętli `while` (wiersze 5–9) usuwa wszystkie węzły ze wskazywanych przez nie list sąsiadów. Na koniec `remove_adjacency_list()` usuwa tablicę wierzchołków (wiersz nr 11), zwraca wartość `NULL` (wiersz nr 12) i kończy działanie.

Funkcja main()

```
1  int main(void)
2  {
3      adjacency_list = convert(adjacency_matrix);
4      if(adjacency_list) {
5          print_adjacency_list(adjacency_list);
6          puts("Proszę wskazać wierzchołek początkowy:");
7          int start_vertex;
8          scanf("%d",&start_vertex);
9          printf("Wynik algorytmu DFS: ");
10         visit_all_vertices(adjacency_list,start_vertex,dfs);
11         puts("");
12         for(int i=0; i < NUMBER_OF_VERTICES; i++)
13             adjacency_list[i].visited = false;
14         printf("Wynik algorytmu BFS: ");
15         visit_all_vertices(adjacency_list,start_vertex,bfs);
16         puts("");
17         adjacency_list = remove_adjacency_list(adjacency_list);
18     }
19     return 0;
20 }
```

Funkcja `main()`

Funkcja `main()` najpierw wywołuje `convert()` i przypisuje wartość przez nią zwróconą do wskaźnika `adjacency_list` (wiersz nr 3). Jeśli po tym ten wskaźnik nie jest pusty (wiersz nr 4), to wykonywana jest reszta operacji. Wyświetlana jest lista sąsiedztwa (wiersz nr 5) i program prosi użytkownika o określenie wierzchołka startowego dla algorytmów przeszukiwania grafu. Numer tego wierzchołka jest zapisywany przez funkcję `scanf()` (wiersz nr 8) w zmiennej `start_vertex`, zadeklarowanej w wierszu nr 7. Następnie, funkcja `main()` informuje użytkownika, że wyświetli wynik algorytmu DFS (wiersz nr 9) i wywołuje funkcję `visit_all_vertices()`, przekazując jako jej ostatni argument adres funkcji `dfs()` (wiersz nr 10). Potem, `main()` wywołuje ponownie `visit_all_vertices()`, tym razem przekazując adres funkcji `bfs()` jako jej ostatni argument (wiersz nr 15). Zanim to jednak nastąpi, musi ona oznaczyć wszystkie wierzchołki odwiedzone przez funkcję `dfs()` jako nieodwiedzone. Inaczej funkcja `bfs()` nie odwiedzi żadnego z nich.

Funkcja `main()`

Ta operacja wykonywana jest w pętli `for` (wiersze 12–13). Po wyświetleniu wyniku algorytmu BFS funkcja `main()` usuwa listę sąsiedztwa (wiersz nr 17) i zwraca zero (wiersz nr 19) kończąc tym samym działanie programu.

Podsumowanie

Grafy są stosunkowo prostym, ale bardzo przydatnym narzędziem, które może być zastosowane do rozwiązania wielu problemów. Przykładowo, mogą one modelować sieci społeczne i komputerowe, drogi lądowe, morskie i powietrzne, obwody elektroniczne i algorytmy (schematy blokowe). Są one także stosowane w zagadnieniach z dziedziny sztucznej inteligencji. Co więcej, istnieje wiele gotowych algorytmów grafowych, których nie trzeba ponownie odkrywać. Zazwyczaj znalezienie rozwiązania problemu przy pomocy grafów sprowadza się do wyrażenia tego zagadnienia jako grafu i wyboru jednego z tych algorytmów. Algorytmy DFS i BFS są podstawowymi algorytmami przeszukiwania grafu, na bazie których można opracować inne algorytmy grafowe. Można je zastosować do dowolnego typu grafów, włącznie ze skierowanymi, nieskierowanymi oraz spójnymi i niespójnymi.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!