Podstawy Programowania 2

Algorytmy DFS i BFS

Arkadiusz Chrobot

Zakład Informatyki

12 czerwca 2018

Plan

Wstęp

Istnieje wiele algorytmów związanych z grafami, które w skrócie nazywane są algorytmami grafowymi. Większość z nich wywodzi się lub jest modyfikacją dwóch podstawowych algorytmów przechodzenia (przeszukiwania) grafów. Celem takich algorytmów jest uzyskanie ścieżki, która rozpoczyna się w określonym wierzchołku i obejmuje wszystkie wierzchołki w silnie spójnym lub spójnym grafie, albo kończy się w wyznaczonym wierzchołku docelowym. Jednym z tych algorytmów jest algorytm przeszukiwania grafu w głąb, nazywany także algorytmem przeszukiwania zastępującego oraz oznaczany akronimem DFS pochodzącym od jego angielskiej nazwy (ang. *Deep–First Search*). Drugi, to algorytm przeszukiwania wszerz, oznaczany skrótem BFS, również pochodzącym od jego angielskiej nazwy (ang. *Breadth–First Search*). Zostaną one opisane w takiej kolejności, w jakiej zostały wymienione.

Algorytm DFS

Wprowadzenie teoretyczne

Algorytm DFS rozpoczyna przechodzenie grafu od wyznaczonego wierzchołka. Jeśli ten wierzchołek posiada co najmniej jednego sąsiada, to ten sąsiad jest umieszczany na stosie, jako kolejny węzeł[1] do odwiedzenia. Jeśli bieżąco odwiedzany wierzchołek miałby więcej sąsiadów, to tylko jeden z nich jest umieszczany na stosie. Po zakończeniu tej czynności bieżący wierzchołek jest oznaczany jako odwiedzony. Wierzchołki znajdujące się na stosie czasem są nazywane odkrytymi. Jeśli DFS jest użyty do budowy innego algorytmu, to między operacją odłożenia kolejnego wierzchołka na stos, a oznaczeniem bieżącego jako odwiedzonego jest wykonywane przetwarzanie informacji zawartej w bieżącym wierzchołku. Po oznaczeniu bieżącego węzła jako odwiedzonego DFS zdejmuje ze stosu, o ile nie jest on pusty, kolejny wierzchołek i go odwiedza powtarzając wymienione na początku slajdu czynności.

[1] Węzeł to inne określenie na wierzchołek grafu.

Algorytm DFS

Wprowadzenie teoretyczne

Nazwa algorytmu pochodzi od sposobu jego działania - zawsze wybiera jednego z sąsiadów bieżąco odwiedzanego wierzchołka i jego odwiedza jako następnego. Podąża on zatem „w głąb" grafu. Jeśli bieżąco odwiedzany wierzchołek nie ma sąsiadów lub wszyscy jego sąsiedzi byli już wcześniej odwiedzeni przez DFS, to algorytm *powraca* do jego poprzednika i sprawdza, czy są jeszcze jacyś jego sąsiedzi, których nie odwiedził. Jeśli graf, dla którego jest wykonywany algorytm, jest grafem spójnym lub silnie spójnym, to algorytm zakończy się po odwiedzeniu wszystkich jego wierzchołków, jeśli zaś nie jest, to zakończy się po odwiedzeniu wszystkich wierzchołków silniej lub silnie spójnej składowej tego grafu, do której należy wierzchołek początkowy, od którego zaczął wykonanie. Jeśli celem działania tego algorytmu jest odwiedzenie wszystkich wierzchołków w grafie, to należy sprawdzić, czy po jego zakończeniu nie zostały jeszcze nieodwiedzone węzły i ponownie go wywołać, dla jednego z nich, do momentu, aż wszystkie wierzchołki zostaną odwiedzone.

Algorytm DFS

Wprowadzenie teoretyczne

Algorytm DFS może także się zakończyć po napotkaniu wierzchołka, który został wyznaczony jako docelowy (ang. *goal vertex*) lub po napotkaniu węzła, który spełnia warunek końcowy (ang. *goal condition*). Ponieważ DFS korzysta ze stosu, to łatwo jest go zaimplementować w postaci rekurencyjnej. Jeśli ten algorytm zastosujemy do drzew binarnych, to odkryjemy, że działa on tak samo jak algorytm odwiedzania takiego drzewa w porządku preorder. Zatem algorytm DFS jest uogólnieniem algorytmu przechodzenia drzewa binarnego w porządku preorder na wszystkie przypadki grafów. Złożoność czasowa tego algorytmu wynosi $\Theta(V + E)$.

Algorytm DFS

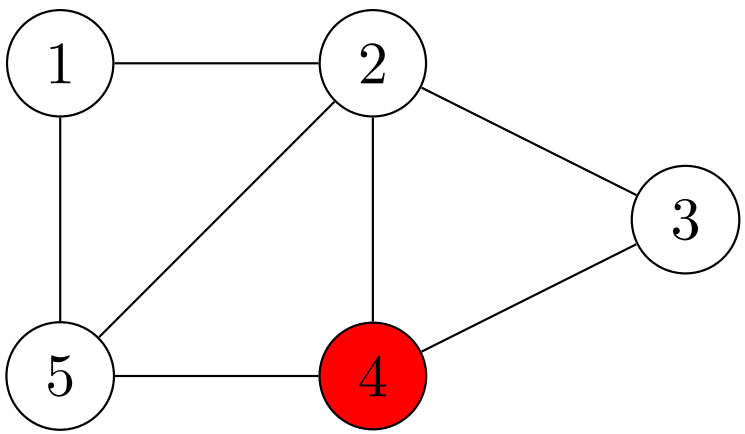
Animacja

Na następnym slajdzie zamieszczona jest animacja przedstawiająca działanie algorytmu DFS dla grafu nieskierowanego, który został zaprezentowany na poprzednim wykładzie. U góry slajdu znajduje się lista wierzchołków, które zostały już odwiedzone i tworzą ścieżkę wygenerowaną przez DFS. Po prawej stronie znajduje się stos, na którym umieszczane są numery wierzchołków, które jako następne będą odwiedzane. Kolejność odwiedzania sąsiadów w tym algorytmie jest dowolna. Kolor czerwony wierzchołka i krawędzi grafu na ilustracji znajdującej się w środku ekranu oznacza, że ten wierzchołek będzie odwiedzany jako kolejny, kolor żółty, że jest przetwarzany, a kolor zielony, że jest już oznaczony jako odwiedzony. Ponieważ graf jest spójny, to algorytm zakończy się po odwiedzeniu wszystkich jego wierzchołków, nie trzeba go ponownie wykonywać dla wierzchołków nieodwiedzonych.

Algorytm DFS

Animacja

ścieżka:



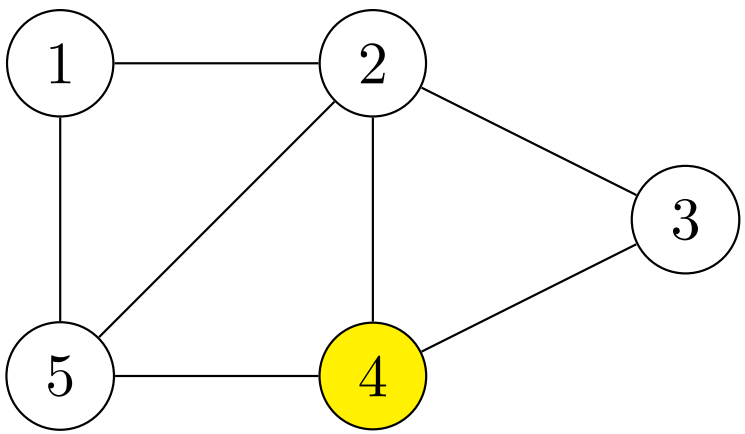
stos:

Przechodzenie grafu nieskierowanego według algorytmu DFS

Algorytm DFS

Animacja

ścieżka: 4



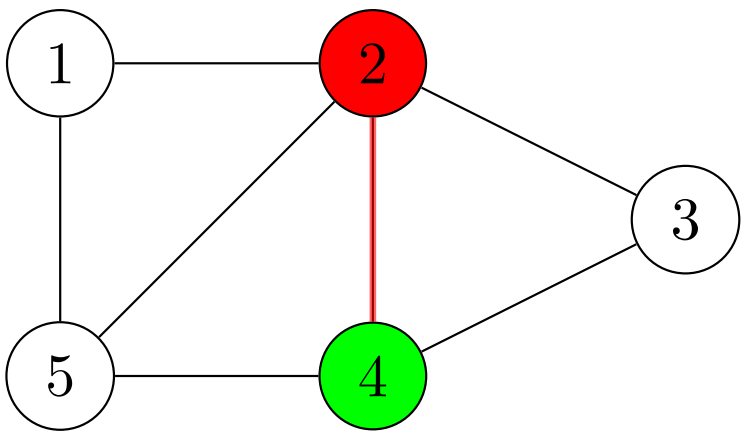
stos: 4

Przechodzenie grafu nieskierowanego według algorytmu DFS

Algorytm DFS

Animacja

ścieżka: 4



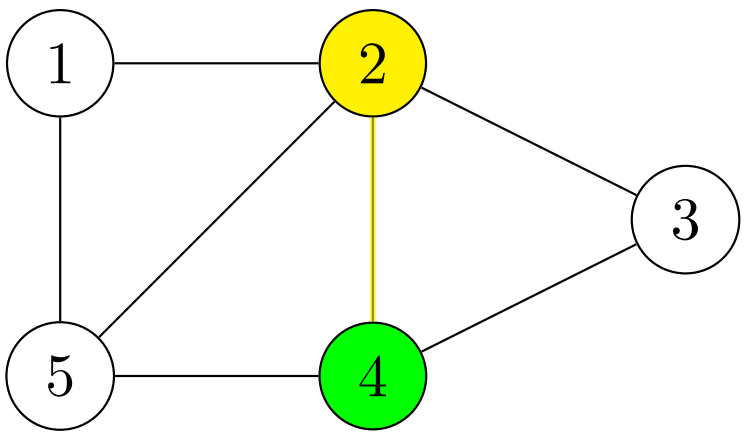
stos: 4

Przechodzenie grafu nieskierowanego według algorytmu DFS

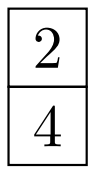
Algorytm DFS

Animacja

ścieżka: 4 2



stos:

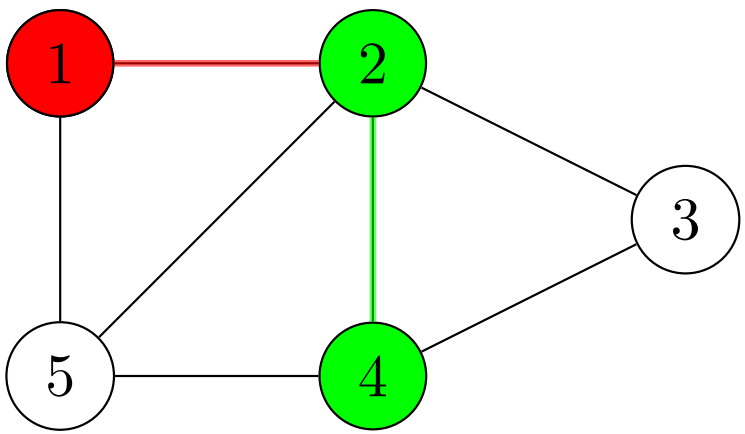


Przechodzenie grafu nieskierowanego według algorytmu DFS

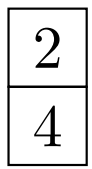
Algorytm DFS

Animacja

ścieżka: 4 2



stos:

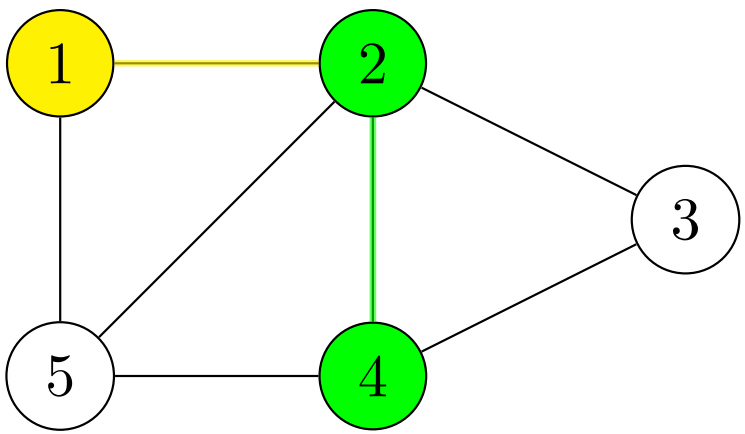


Przechodzenie grafu nieskierowanego według algorytmu DFS

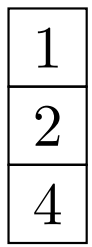
Algorytm DFS

Animacja

ścieżka: 4 2 1



stos:

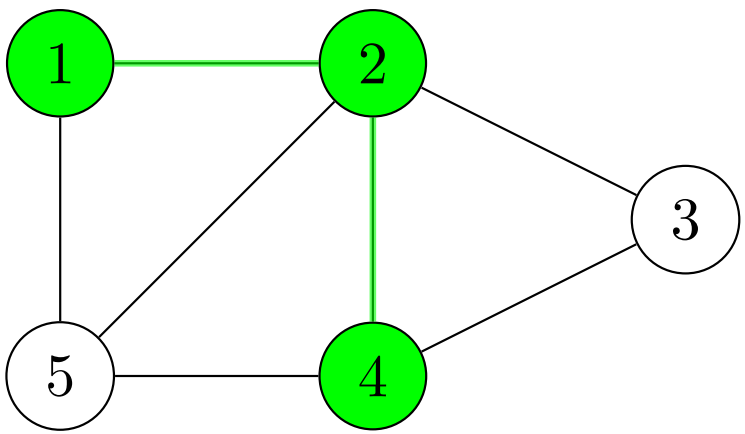


Przechodzenie grafu nieskierowanego według algorytmu DFS

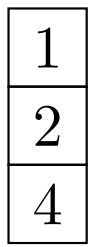
Algorytm DFS

Animacja

ścieżka: 4 2 1



stos:

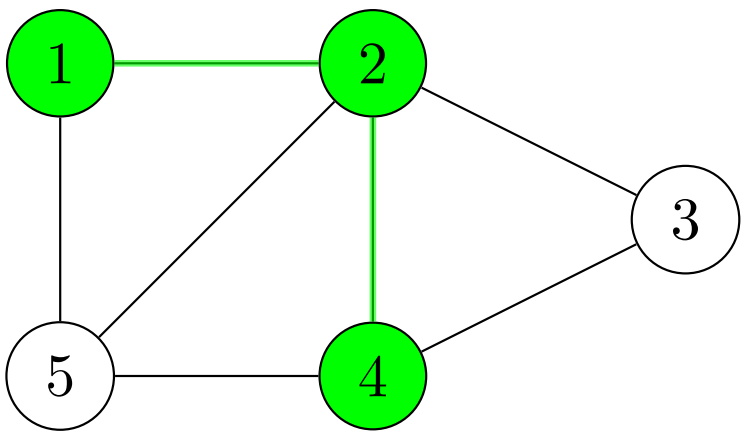


Przechodzenie grafu nieskierowanego według algorytmu DFS

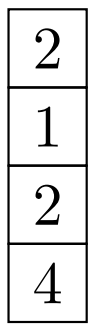
Algorytm DFS

Animacja

ścieżka: 4 2 1



stos:

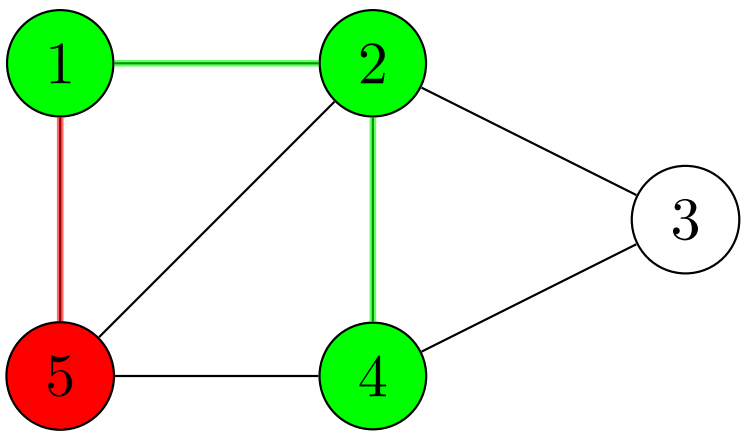


Przechodzenie grafu nieskierowanego według algorytmu DFS

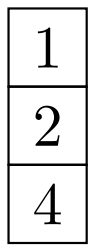
Algorytm DFS

Animacja

ścieżka: 4 2 1



stos:

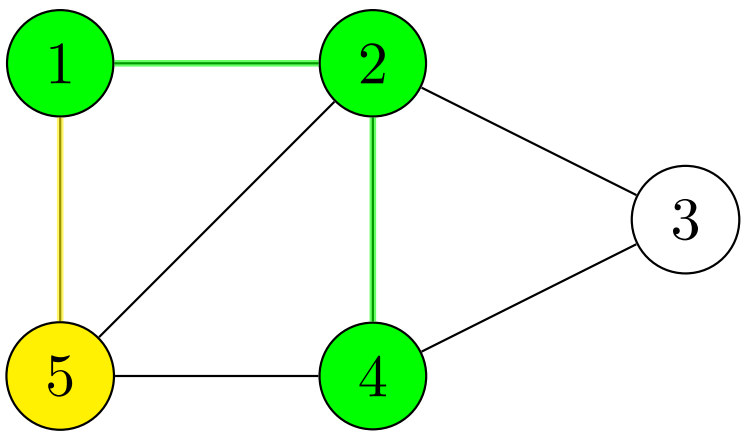


Przechodzenie grafu nieskierowanego według algorytmu DFS

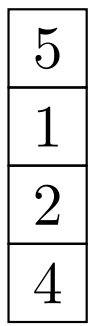
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

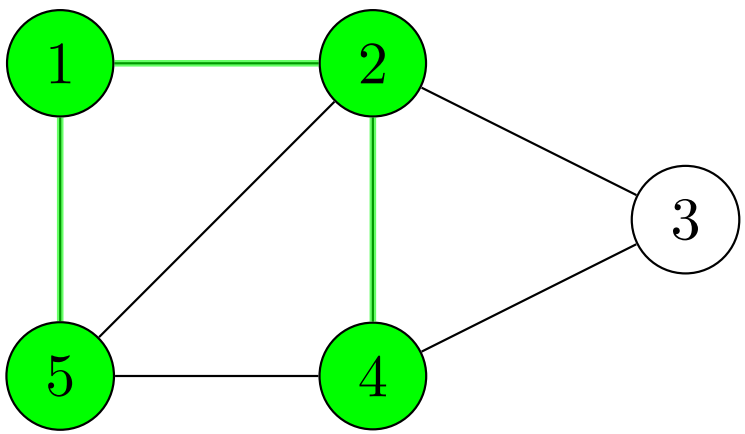


Przechodzenie grafu nieskierowanego według algorytmu DFS

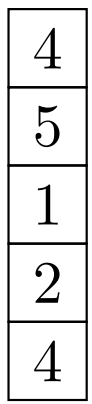
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

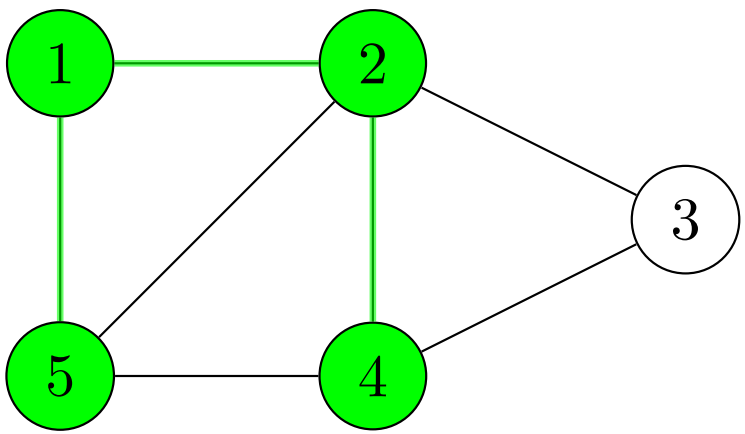


Przechodzenie grafu nieskierowanego według algorytmu DFS

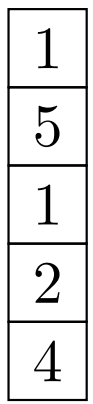
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

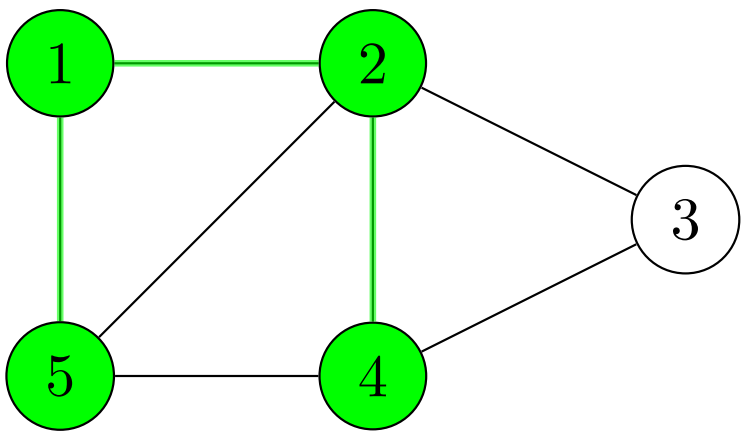


Przechodzenie grafu nieskierowanego według algorytmu DFS

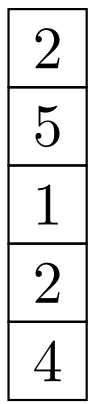
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

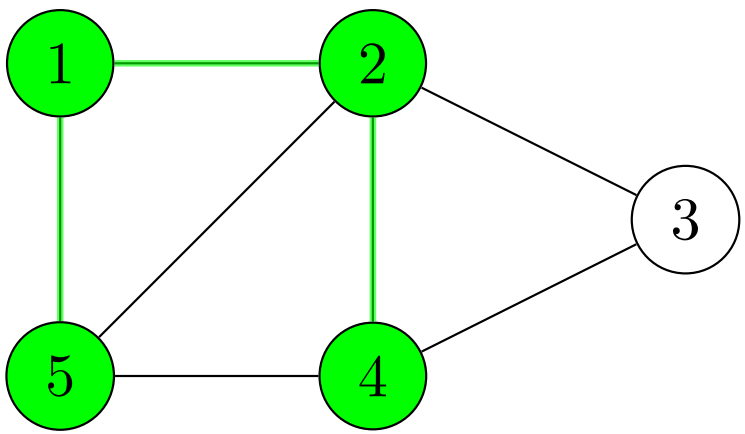


Przechodzenie grafu nieskierowanego według algorytmu DFS

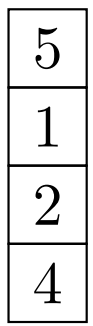
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

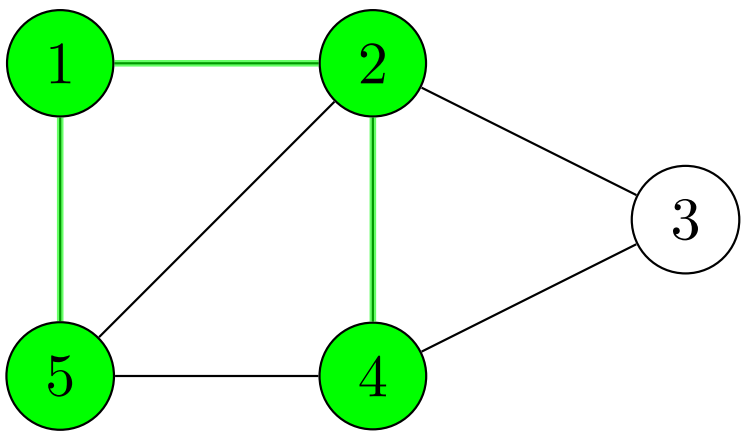


Przechodzenie grafu nieskierowanego według algorytmu DFS

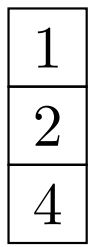
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

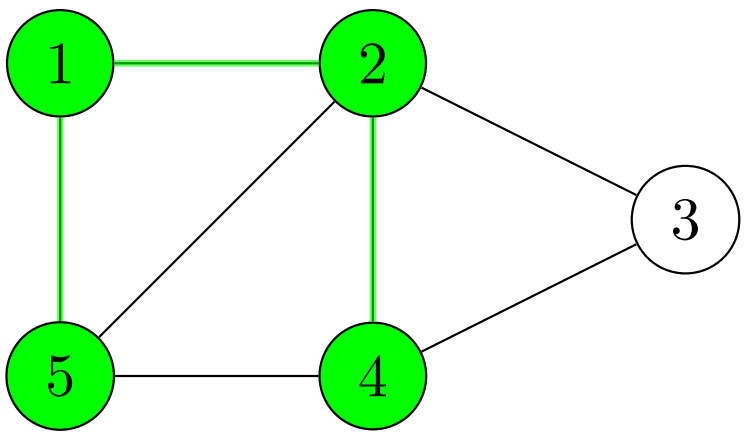


Przechodzenie grafu nieskierowanego według algorytmu DFS

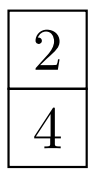
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

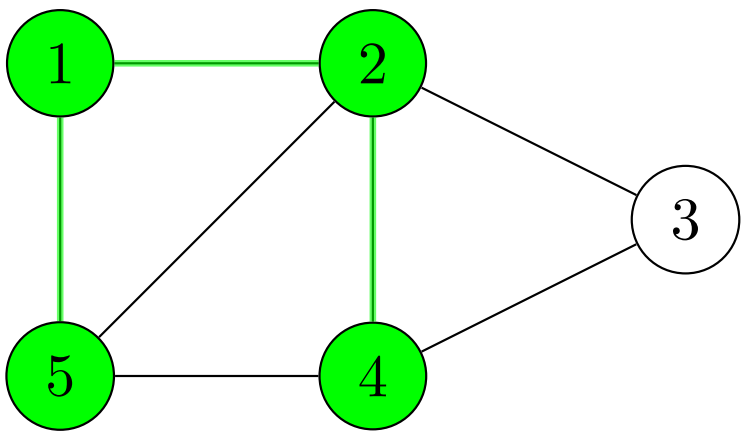


Przechodzenie grafu nieskierowanego według algorytmu DFS

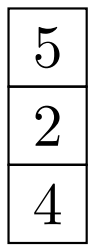
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

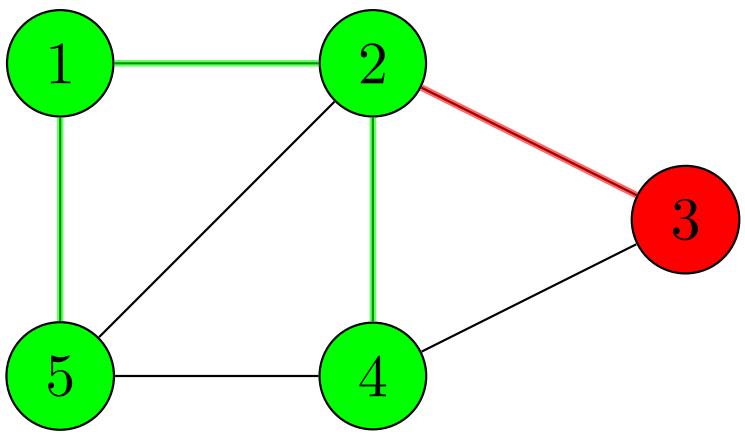


Przechodzenie grafu nieskierowanego według algorytmu DFS

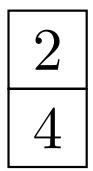
Algorytm DFS

Animacja

ścieżka: 4 2 1 5



stos:

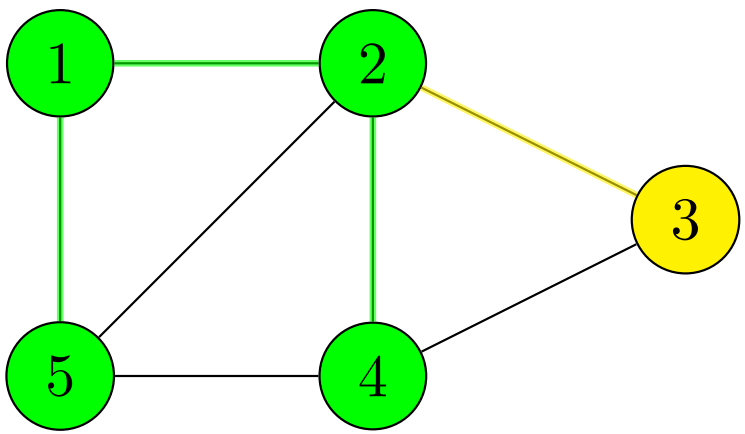


Przechodzenie grafu nieskierowanego według algorytmu DFS

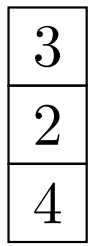
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

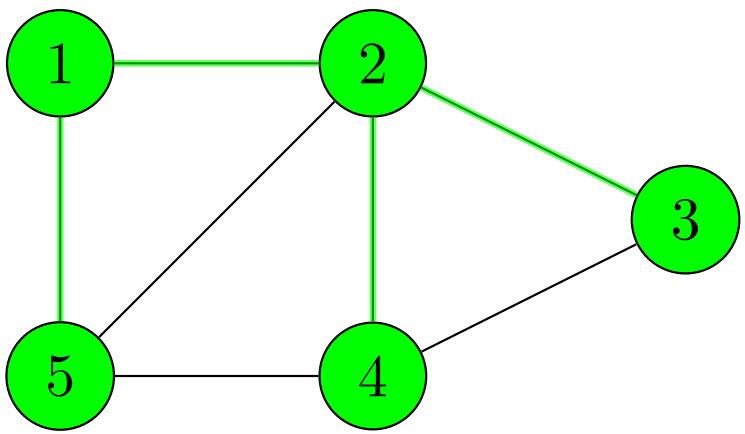


Przechodzenie grafu nieskierowanego według algorytmu DFS

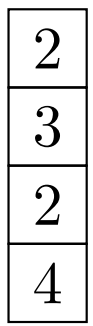
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

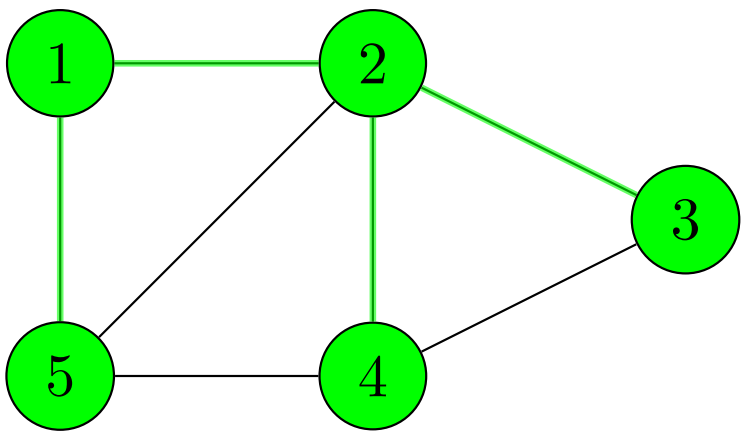


Przechodzenie grafu nieskierowanego według algorytmu DFS

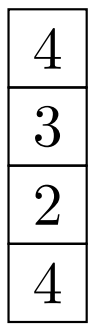
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

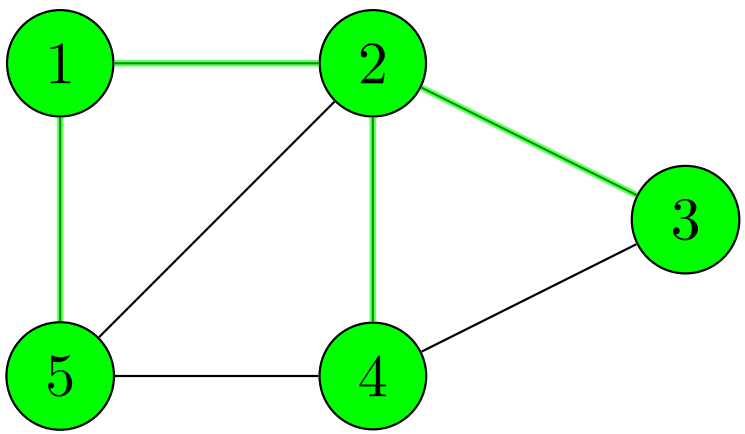


Przechodzenie grafu nieskierowanego według algorytmu DFS

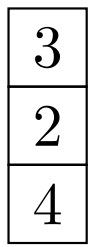
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

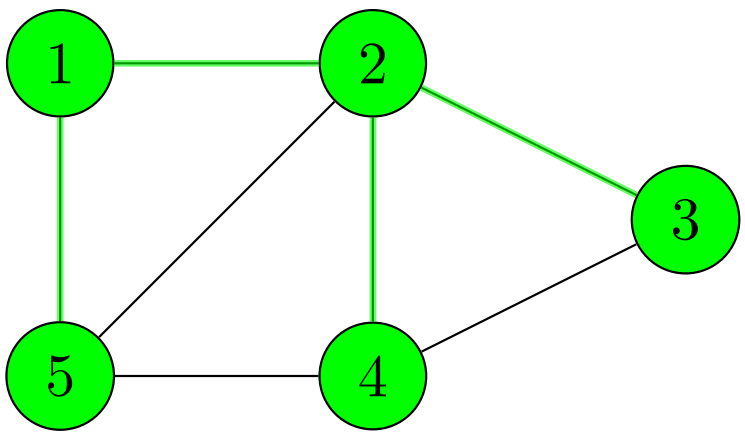


Przechodzenie grafu nieskierowanego według algorytmu DFS

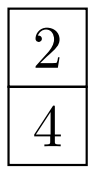
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

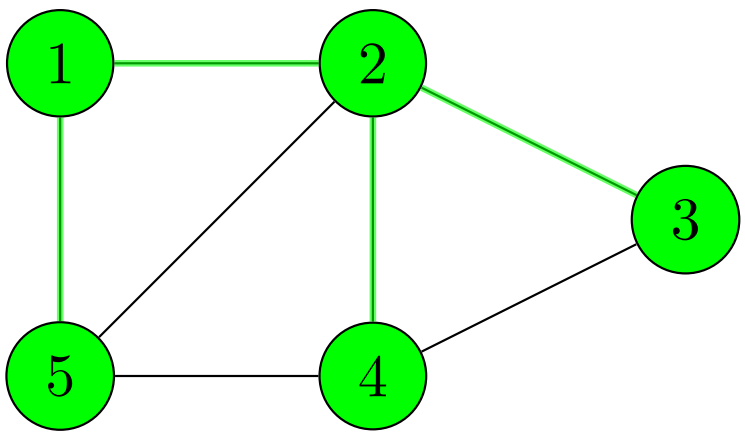


Przechodzenie grafu nieskierowanego według algorytmu DFS

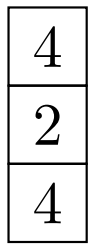
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

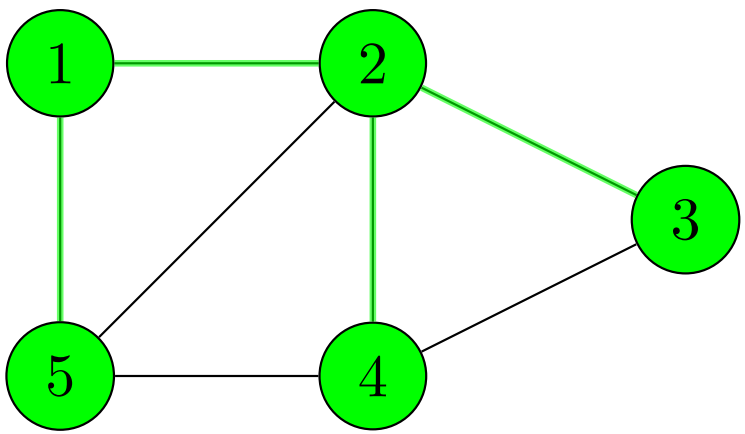


Przechodzenie grafu nieskierowanego według algorytmu DFS

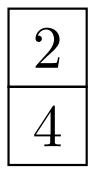
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

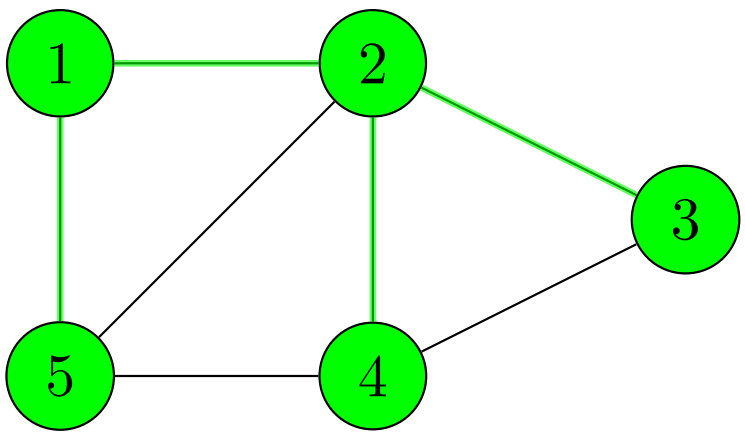


Przechodzenie grafu nieskierowanego według algorytmu DFS

Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



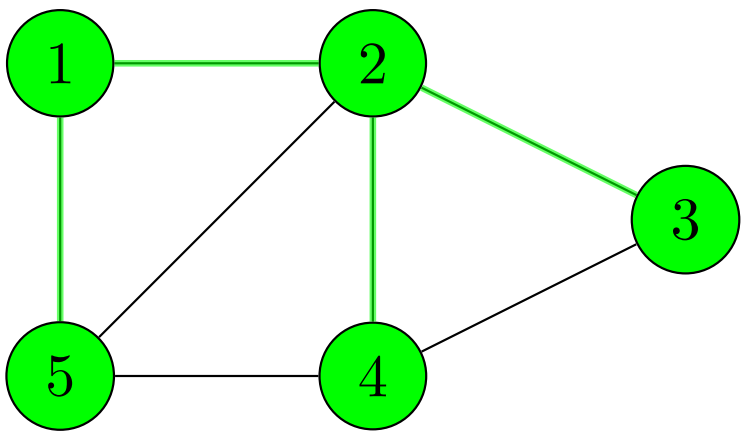
stos: 4

Przechodzenie grafu nieskierowanego według algorytmu DFS

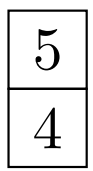
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

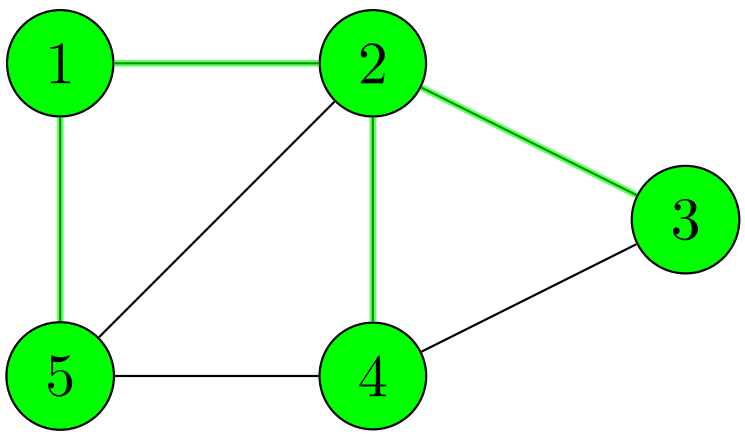


Przechodzenie grafu nieskierowanego według algorytmu DFS

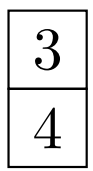
Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

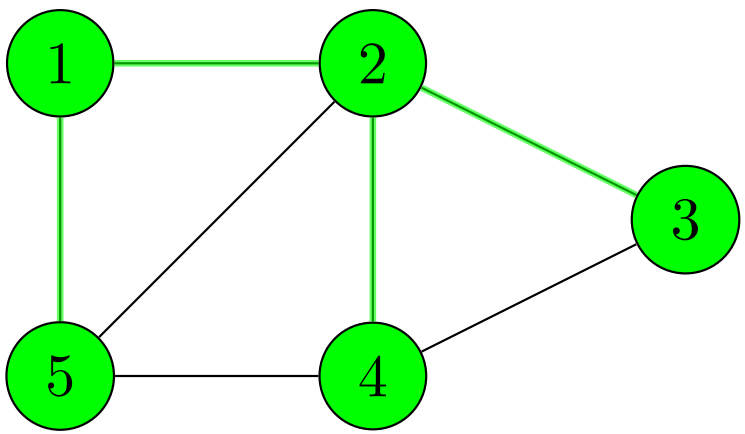


Przechodzenie grafu nieskierowanego według algorytmu DFS

Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



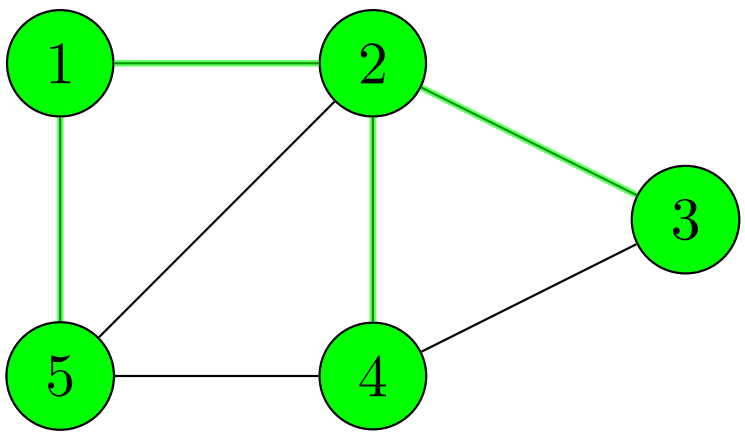
stos: 4

Przechodzenie grafu nieskierowanego według algorytmu DFS

Algorytm DFS

Animacja

ścieżka: 4 2 1 5 3



stos:

Przechodzenie grafu nieskierowanego według algorytmu DFS

Algorytm DFS - implementacja

Następne slajdy przedstawiają zmodyfikowaną wersję programu z poprzedniego wykładu, w której nieskierowany graf, jest przeszukiwany za pomocą algorytmu DFS. Algorytm ten bazuje na liście sąsiedztwa, która otrzymywana jest z macierzy sąsiedztwa tego grafu. Ponieważ przeszukiwany graf jest spójny, to DFS zakończy się po odwiedzeniu wszystkich jego wierzchołków, ale program jest przygotowany na ewentualność odwiedzania grafu niespójnego.

Algorytm DFS - implementacja

Macierz sąsiedztwa i typ bazowy listy sąsiedztwa

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4
5  typedef int matrix[5][5];
6
7  const matrix adjacency_matrix = {{0,1,0,0,1},
8      {1,0,1,1,1,},
9      {0,1,0,1,0,},
10     {0,1,1,0,1,},
11     {1,1,0,1,0,}
12 };
13
14 struct vertex {
15     int vertex_number;
16     bool visited;
17     struct vertex *next, *down;
18 } *start_vertex;
```

Algorytm DFS - implmentacja

Macierz sąsiedztwa i typ bazowy listy sąsiedztwa

Zaprezentowany na poprzednim slajdzie początek kodu źródłowego programu, różni się tylko jednym szczegółem od podobnego fragmentu zaprezentowanego na poprzednim wykładzie. Typ bazowy listy sąsiedztwa posiada dodatkowe pole typu `bool`, które będzie służyło do oznaczania, czy dany wierzchołek został już odwiedzony. Jeśli tak będzie, to jego wartość będzie wynosiła `true`, a w przeciwnym przypadku `false`. Wartość tego pola będzie miała znaczenie jedynie w liście wszystkich wierzchołków grafu (liście „pionowej”).

Algorytm DFS - implementacja

Typ bazowy kolejki i struktura jej wskaźników

```
1  struct fifo_node {
2      int vertex_number;
3      struct fifo_node *next;
4  };
5
6  struct fifo_pointers {
7      struct fifo_node *head, *tail;
8  } path;
```

Algorytm DFS - implementacja

Typ bazowy kolejki i struktura jej wskaźników

Poprzedni slajd zawiera definicje typu bazowego kolejki FIFO i typu struktury przechowującej wskaźniki na początek i koniec tej kolejki. W wierszu nr 8 deklarowana jest również zmienna globalna o nazwie `path` typu `fifo_pointers`. Każdy element kolejki będzie przechowywał numer odwiedzonego wierzchołka, a cała kolejka będzie służyła do zapamiętania ścieżki przedstawiającej kolejność, w jakiej algorytm DFS odwiedził wierzchołki grafu, poczynając od wskazanego wierzchołka początkowego.

Algorytm DFS - implementacja

Funkcja `enqueue()`

```
1  void enqueue(struct fifo_pointers *fifo, int vertex_number)
2  {
3      struct fifo_node *new_node =
4          (struct fifo_node *)malloc(sizeof(struct fifo_node));
5      if(new_node) {
6          new_node->vertex_number = vertex_number;
7          new_node->next = NULL;
8          if(fifo->head==NULL)
9              fifo->head = fifo->tail = new_node;
10         else {
11             fifo->tail->next=new_node;
12             fifo->tail=new_node;
13         }
14     } else
15         fprintf(stderr,"Nowy element nie został utworzony!\n");
16 }
```

Algorytm DFS - implementacja

Funkcja `enqueue()`

Na poprzednim slajdzie pokazano kod źródłowy funkcji `enqueue()`, która służy do tworzenia kolejki FIFO i dodawania do niej nowych elementów. Funkcja ta zaimplementowana jest podobnie do swoich odpowiedniczek przedstawianych na wcześniejszych wykładach, więc nie będzie tu dokładniej opisywana.

Algorytm DFS - implementacja

Funkcja `dequeue()`

```
1  void dequeue(struct fifo_pointers *fifo)
2  {
3      if(fifo->head) {
4          struct fifo_node *tmp = fifo->head->next;
5          free(fifo->head);
6          fifo->head=tmp;
7          if(tmp==NULL)
8              fifo->tail = NULL;
9      }
10 }
```

Algorytm DFS - implementacja

Funkcja `dequeue()`

Funkcja `dequeue()`, zaprezentowana na poprzednim slajdzie, różni się od swojej odpowiedniczki przedstawionej na wykładzie poświęconym kolejkom tym, że nie zwraca żadnej wartości, a jedynie usuwa pierwszy element z kolejki FIFO.

Algorytm DFS - implementacja

Funkcja `remove_queue()`

```
1  void remove_queue(struct fifo_pointers *fifo)
2  {
3      while(fifo->head)
4          dequeue(fifo);
5  }
```

Algorytm DFS - implementacja

Funkcja `remove_queue()`

Funkcja `remove_queue()` wywołuje w pętli `while` funkcję `dequeue()`, celem usunięcia kolejki FIFO. Jako argument wywołania przyjmuje ona adres struktury zawierającej wskaźniki na tę kolejkę. Opisywana funkcja nie zwraca żadnej wartości. Zawarta w jej wnętrzu pętla `while` wykonuje się tak długo, jak długo wskaźnik na czoło kolejki ma wartość różną od `NULL`.

Algorytm DFS - implementacja

Funkcja `print_path()`

```
1  void print_path(struct fifo_pointers fifo)
2  {
3      while(fifo.head) {
4          printf("%d ",fifo.head->vertex_number);
5          fifo.head = fifo.head->next;
6      }
7      puts("");
8  }
```

Algorytm DFS - implementacja

Funkcja `print_path()`

Funkcja `print_path()` jest wariantem funkcji `print_queue()`, przystosowanym do wypisywania informacji z kolejki FIFO, która będzie przechowywała ścieżkę wygenerowaną przez algorytm DFS.

Algorytm DFS - implementacja

Funkcja `create_vertical_list()`

```
1  void create_vertical_list(struct vertex **start_vertex,
2                            const matrix adjacency_matrix)
3  {
4      int i;
5      for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
6          *start_vertex = (struct vertex *)
7                          malloc(sizeof(struct vertex));
8          if(*start_vertex) {
9              (*start_vertex)->vertex_number = i+1;
10             (*start_vertex)->visited = false;
11             (*start_vertex)->down = (*start_vertex)->next = NULL;
12             start_vertex = &(*start_vertex)->down;
13         }
14     }
15 }
```

Algorytm DFS - implementacja

Funkcja `create_vertical_list()`

Funkcja `create_vertical_list()` różni się od swojej odpowiedniczki z poprzedniego wykładu jedynie tym, że inicjuje (wiersz nr 10) pole `visited` każdego wierzchołka na liście wszystkich wierzchołków.

Algorytm DFS - implementacja

Funkcja `convert_matrix_to_list()`

```
1   struct vertex *convert_matrix_to_list(const matrix adjacency_matrix)
2   {
3       struct vertex *start_vertex = NULL;
4       create_vertical_list(&start_vertex,adjacency_matrix);
5       if(start_vertex) {
6           struct vertex *horizontal_pointer = NULL, *vertical_pointer = NULL;
7           horizontal_pointer = vertical_pointer = start_vertex;
8           int i,j;
9           for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
10              for(j=0; j<sizeof(matrix)/sizeof(*adjacency_matrix); j++)
11                  if(adjacency_matrix[i][j]) {
12                      struct vertex *new_vertex = (struct vertex *)malloc(sizeof(struct vertex));
13                      if(new_vertex) {
14                          new_vertex->vertex_number = j+1;
15                          new_vertex->visited = false;
16                          new_vertex->down = new_vertex->next = NULL;
17                          horizontal_pointer->next = new_vertex;
18                          horizontal_pointer = horizontal_pointer->next;
19                      }
20                  }
21              vertical_pointer = vertical_pointer->down;
22              horizontal_pointer = vertical_pointer;
23          }
24      }
25      return start_vertex;
26  }
```

Algorytm DFS - implementacja

Funkcja `convert_matrix_to_list()`

Funkcja konwertująca macierz sąsiedztwa na listę sąsiedztwa również różni się tylko jednym szczegółem od jej odpowiedniczki z poprzedniego wykładu. Tym szczegółem jest również inicjacja pola `visited` nowo utworzonego węzła (wiersz nr 15).

Algorytm DFS - implementacja

Funkcja `print_adjacency_list()`

```
1  void print_adjacency_list(struct vertex *start_vertex)
2  {
3      while(start_vertex) {
4          printf("%3d:",start_vertex->vertex_number);
5          struct vertex *horizontal_pointer = start_vertex->next;
6          while(horizontal_pointer) {
7              printf("%3d",horizontal_pointer->vertex_number);
8              horizontal_pointer = horizontal_pointer->next;
9          }
10         start_vertex = start_vertex->down;
11         puts("");
12     }
13 }
```

Algorytm DFS - implementacja

Funkcja `print_adjacency_list()`

Funkcja `print_adjacency_list()` jest zaimplementowana dokładnie tak samo, jak jej odpowiedniczka z poprzedniego wykładu.

Algorytm DFS - implementacja

Funkcja `remove_adjacency_list()`

```
1  void remove_adjacency_list(struct vertex **start_vertex)
2  {
3      while(*start_vertex) {
4          struct vertex *horizontal_pointer=(*start_vertex)->next;
5          while(horizontal_pointer) {
6              struct vertex *next_horizontal =
7                                          horizontal_pointer->next;
8              free(horizontal_pointer);
9              horizontal_pointer = next_horizontal;
10         }
11         struct vertex *next_vertical = (*start_vertex)->down;
12         free(*start_vertex);
13         *start_vertex= next_vertical;
14     }
15 }
```

Algorytm DFS - implementacja

Funkcja `remove_adjacency_list()`

Również `remove_adjacency_list()` jest zaimplementowana tak samo jak jej odpowiedniczka z poprzedniego wykładu.

Algorytm DFS - implementacja

Funkcja `find_vertex()`

```
1  struct vertex *find_vertex(struct vertex *start_vertex,
2                                                  int vertex_number)
3  {
4      while(start_vertex &&
5                      start_vertex->vertex_number!=vertex_number)
6          start_vertex = start_vertex->down;
7      return start_vertex;
8  }
```

Algorytm DFS - implementacja

Funkcja `find_vertex()`

Funkcja `find_vertex()` pełni rolę pomocniczą dla funkcji implementujących algorytm DFS. Jej zadaniem jest znalezienie na liście wszystkich wierzchołków (liście „pionowej") elementu reprezentującego wierzchołek o określonym numerze. Jako argumenty wywołania przyjmuje ona adres wierzchołka startowego listy sąsiedztwa oraz numer szukanego wierzchołka. W pętli `while` (wiersze 4-6) iteruje ona z użyciem wskaźnika `start_vertex` po liście wszystkich wierzchołków i sprawdza, czy bieżąco wskazywany przez ten wskaźnik element zawiera żądany numer. Jeśli tak nie jest, to przechodzi ona do kolejnego elementu na liście, a jeśli tak jest, to pętla się zatrzymuje i funkcja zwraca adres znalezionego elementu reprezentującego rzeczony wierzchołek. Funkcja jest zabezpieczona na ewentualność przekazania jest wskaźnika o wartości `NULL` lub numeru nieistniejącego wierzchołka, choć jest to mało prawdopodobna sytuacja. Jeśli jednak ona wystąpiłaby, to `find_vertex()` zwróci wartość `NULL`.

Algorytm DFS - implementacja

Funkcja `has_not_been_visited()`

```
1  bool has_not_been_visited(struct vertex *start_vertex,
2                            const struct vertex *vertex)
3  {
4      return !find_vertex(start_vertex,
5                          vertex->vertex_number)->visited;
6  }
```

Algorytm DFS - implementacja

Funkcja `has_not_been_visited()`

Funkcja `has_not_been_visited()` sprawdza, czy wierzchołek o danym numerze nie został już odwiedzony. Otrzymuje ona dwa argumenty wywołania - adres listy sąsiedztwa grafu oraz adres elementu reprezentującego badany wierzchołek na liście sąsiedztwa bieżąco odwiedzanego wierzchołka (jednej z list „poziomych"). Funkcja `has_not_been_visited()` wywołuje funkcję `find_vertex()` celem znalezienia elementu reprezentującego wierzchołek sąsiedni na liście wszystkich wierzchołków (liście „pionowej"). Ponieważ ta ostania funkcja zwraca wskaźnik na poszukiwany element, to w wierszu nr 5 opisywanej funkcji jest on wykorzystywany bezpośrednio do odczytania stanu pola `visited` badanego wierzchołka. Następnie funkcja `has_not_been_visited()` zwraca zanegowaną wartość tego pola.

Algorytm DFS - implementacja

Funkcja `dfs()`

```
1  void dfs(struct vertex *start_vertex, struct vertex *vertex,
2                                        struct fifo_pointers *fifo)
3  {
4      if(start_vertex && vertex) {
5          enqueue(fifo, vertex->vertex_number);
6          vertex->visited = true;
7          while(vertex) {
8              vertex = vertex->next;
9              if(vertex &&
10                     has_not_been_visited(start_vertex,vertex))
11                 dfs(start_vertex,find_vertex(start_vertex,
12                                     vertex->vertex_number),fifo);
13         }
14     }
15 }
```

Algorytm DFS - implementacja

Funkcja `dfs()`

Funkcja `dfs()`, zgodnie ze swoją nazwą, implementuje algorytm DFS. Nie zwraca ona żadnej wartości, ale przyjmuje trzy argumenty wywołania. Pierwszym jest adres wierzchołka startowego listy sąsiedztwa grafu, drugim adres wierzchołka początkowego, od którego funkcja ma rozpocząć przeszukiwanie grafu, a ostatnim struktura wskaźników kolejki FIFO, w której zostanie zapisana wygenerowana przez funkcję ścieżka. W wierszu nr 4 funkcja sprawdza, czy przekazane jej przez parametry adresy wierzchołków grafu mają wartość różną od `NULL`. Jeśli tak jest, to funkcja dodaje do kolejki FIFO element z numerem wierzchołka wskazywanego przez parametr `vertex` (wiersz nr 5) i oznacza go jako odwiedzony zmieniając stan pola `visited` reprezentującego go elementu na `true` (wiersz nr 6). Następnie w pętli `while` funkcja przypisuje wskaźnikowi `vertex` adres pola `next` elementu, na który on wskazuje (wiersz nr 8).

Algorytm DFS - implementacja

Funkcja `dfs()`

Jeśli wartość tego adresu jest różna od `NULL`, to oznacza to, że istnieją sąsiedzi tego bieżąco odwiedzanego wierzchołka i `vertex` wskazuje na element listy sąsiadów reprezentujący pierwszego z nich. Istnienie tego wierzchołka, oraz to, czy nie był on jeszcze odwiedzany jest sprawdzane w wierszach nr 9 i 10 opisywanej funkcji. Jeżeli oba warunki są spełnione, to funkcja wywołuje się rekurencyjnie dla tego sąsiada. Jako drugi argument tego wywołania przekazywany jest wynik działania funkcji `find_vertex()`, która zwraca adres elementu reprezentującego wierzchołek sąsiedni na liście wszystkich wierzchołków (liście „pionowej"). Po powrocie z wywołania rekurencyjnego wykonywana jest kolejna iteracja pętli `while` i jeśli istnieje kolejny wierzchołek sąsiedni, reprezentowany na liście wierzchołków sąsiednich, to ponownie jest dla niego rekurencyjnie wywoływana funkcja `dfs()`.

Algorytm DFS - implementacja

Funkcja `visit_all_vertexes()`

```
1  void visit_all_vertexes(struct vertex *start_vertex)
2  {
3      struct vertex *vertex = start_vertex;
4      while(vertex) {
5          if(!vertex->visited) {
6              struct fifo_pointers path;
7              path.head = path.tail = NULL;
8              dfs(start_vertex,vertex,&path);
9              print_path(path);
10             remove_queue(&path);
11         }
12         vertex = vertex->down;
13     }
14 }
```

Algorytm DFS - implementacja

Funkcja `visit_all_vertexes()`

Funkcja `visit_all_vertexes()` wywoływana jest po pierwszym wywołaniu `dfs()`. Sprawdza ona, czy zostały odwiedzone wszystkie wierzchołki grafu i wywołuje `dfs()` dla tych, które nie zostały jeszcze odwiedzone. Ta funkcja nic nie zwraca, ale przyjmuje jeden argument wywołania w postaci adresu wierzchołka początkowego listy sąsiedztwa grafu. W wierszu nr 2 tej funkcji jest zdefiniowany wskaźnik lokalny `vertex`, któremu przypisywany jest jako wartość początkowa adres przechowywany we parametrze `start_vertex`. Wprawdzie ten parametr stosuje przekazanie przez wartość, więc funkcja mogłaby użyć go do iterowania po liście wierzchołków bez konsekwencji dla innych elementów programu, ale adres który on przechowuje będzie wielokrotnie w tej funkcji wykorzystywany, stąd konieczność zadeklarowania osobnego wskaźnika do iterowania po tej liście.

Algorytm DFS - implementacja

Funkcja `visit_all_vertexes()`

W pętli `while` funkcja iteruje z użyciem wskaźnika `vetex` po liście wszystkich wierzchołków grafu (liście „pionowej”) i sprawdza, czy któryś z nich nie został jeszcze odwiedzony (wiersz nr 4). Jeśli tak jest, to w wierszu nr 7 dla tego nieodwiedzonego wierzchołka wywoływana jest funkcja `dfs()`. Jako ostatni argument wywołania tej funkcji przekazywany jest adres struktury wskaźników kolejki FIFO zadeklarowanej w wierszu nr 5 i zainicjowanej w wierszu nr 6. Innymi słowy, funkcja `visit_all_vertexes()` posługuje się swoją lokalną kolejką FIFO. Po zakończeniu funkcji `dfs()` zawartość tej kolejki jest wyświetlana na ekranie i kolejka jest usuwana, aby w przypadku istnienia kolejnego nieodwiedzonego wierzchołka wskaźniki w strukturze kolejki mogły być ponownie użyte do stworzenia nowej instancji kolejki. Pętla `while` kończy się, gdy zostaną sprawdzone wszystkie elementy na liście wszystkich wierzchołków grafu, co będzie oznaczało, że cały graf został już przeszukany.

Algorytm DFS - implementacja

Funkcja `main()`

```
1   int main(void)
2   {
3       start_vertex = convert_matrix_to_list(adjacency_matrix);
4       if(start_vertex) {
5           print_adjacency_list(start_vertex);
6           puts("Podaj numer wierzchołka startowego:");
7           int vertex_number = 0;
8           scanf("%d",&vertex_number);
9           dfs(start_vertex,find_vertex(start_vertex,vertex_number), &path);
10          puts("Wynik przechodzenia grafu algorytmem DFS:");
11          print_path(path);
12          remove_queue(&path);
13          visit_all_vertexes(start_vertex);
14          remove_adjacency_list(&start_vertex);
15      }
16      return 0;
17  }
```

Algorytm DFS - implementacja

Funkcja `main()`

W stosunku do programu przedstawionego na poprzednim wykładzie funkcja `main()` posiada dodatkowe wiersze od nr 6 do nr 13. W wierszu nr 6 wypisywany jest na ekranie komunikat z prośbą do użytkownika, aby wskazał numer wierzchołka grafu, od którego algorytm DFS zacznie pracę. Ta wartość jest odczytywana z klawiatury (wiersz nr 8) i zapamiętywana w zmiennej lokalnej `vertex_number`. Następnie wywoływana jest funkcja `dfs()`. Drugi argument jej wywołania - adres elementu na liście wszystkich wierzchołków grafu, który reprezentuje wierzchołek początkowy - jest ustalany przy użyciu wywołania funkcji `find_vertex()`. Po zakończeniu działania `dfs()` program wypisuje ścieżkę w grafie, którą wygenerowała ta funkcja wraz z odpowiednim komunikatem (wiersze 10-11) i usuwa kolejkę, w której ta ścieżka była zapisana (wiersz nr 12). W wierszu nr 13 funkcja `main()` wywołuje `visit_all_vertexes()` celem odwiedzenia pozostałych wierzchołków grafu, jeżeli zostały jeszcze jakieś nieodwiedzone.

Algorytm BFS

Wprowadzenie teoretyczne

Algorytm BFS podobnie jak DFS również przeszukuje graf. Zasadnicza różnica między nimi polega na tym, że BFS używa kolejki FIFO zamiast stosu do zapamiętania wierzchołków, które zostały przez niego odkryte, ale jeszcze nie były odwiedzane. Odwiedzając bieżący wierzchołek ten algorytm dodaje wszystkie jego wierzchołki sąsiadujące na koniec wspomnianej kolejki. Po oznaczeniu bieżącego wierzchołka jako odwiedzonego algorytm wybiera pierwszy odkryty wierzchołek z czoła kolejki i jego odwiedza jako kolejny. Ponieważ wszyscy sąsiedzi danego wierzchołka są odwiedzani jako pierwsi, to stąd wywodzi się określenie działania tego algorytmu - przeszukiwanie wszerz. Algorytm ten jest implementowany najczęściej iteracyjnie (z użyciem pętli). Jego złożoność czasowa wynosi $O(V + E)$.

Algorytm BFS

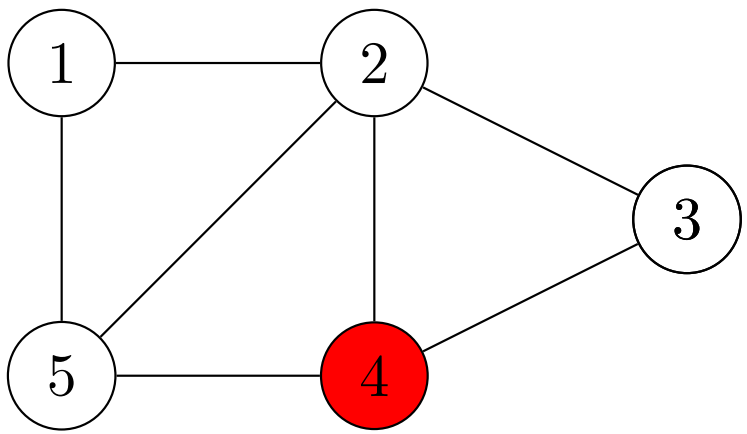
Animacja

Następny slajd przedstawia animację działania algorytmu BFS dla grafu nieskierowanego - tego samego, który posłużył do zilustrowania działania algorytmu DFS. W porównaniu z animacją przedstawiającą działanie tego ostatniego algorytmu, zamiast stosu, na dole rysunku, przedstawiona jest kolejka FIFO. Proszę zwrócić uwagę, że pominięto w animacji umieszczanie już odwiedzonych elementów w rzeczonej kolejce. Wszystkie pozostałe elementy animacji są takie same, jak w przypadku algorytmu DFS.

Algorytm BFS

Animacja

ścieżka:



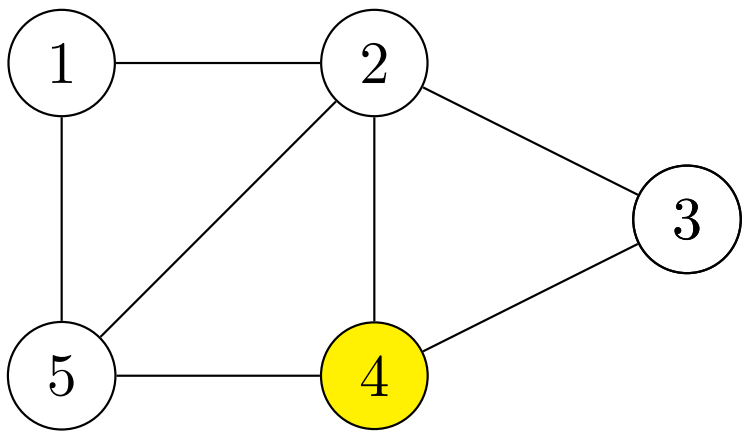
kolejka FIFO:

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka:



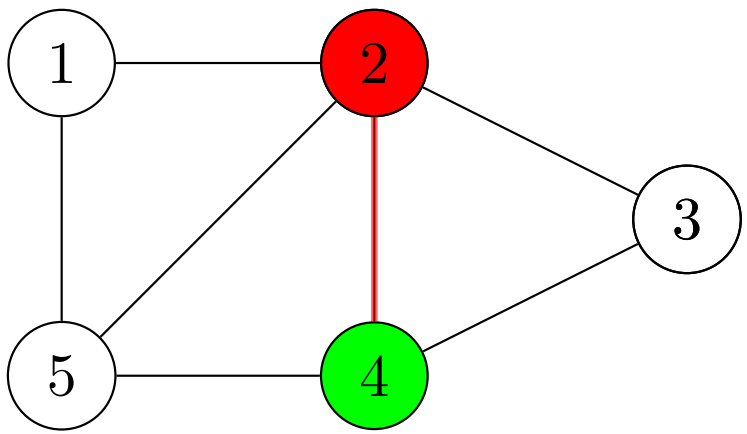
kolejka FIFO: 4

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4



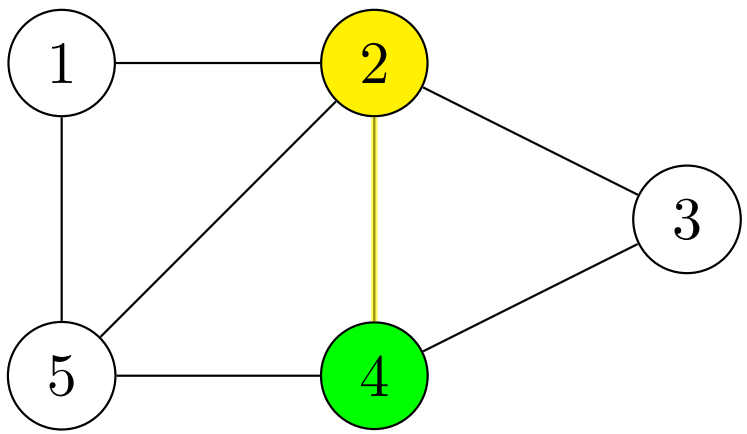
kolejka FIFO:

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4



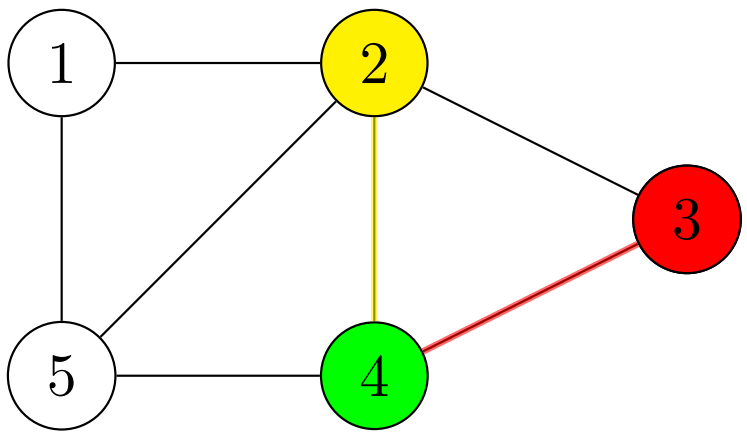
kolejka FIFO: 2

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4



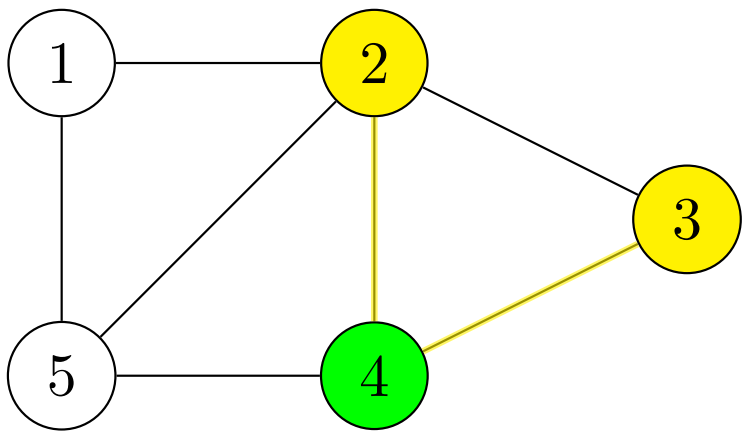
kolejka FIFO: 2

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4



kolejka FIFO:

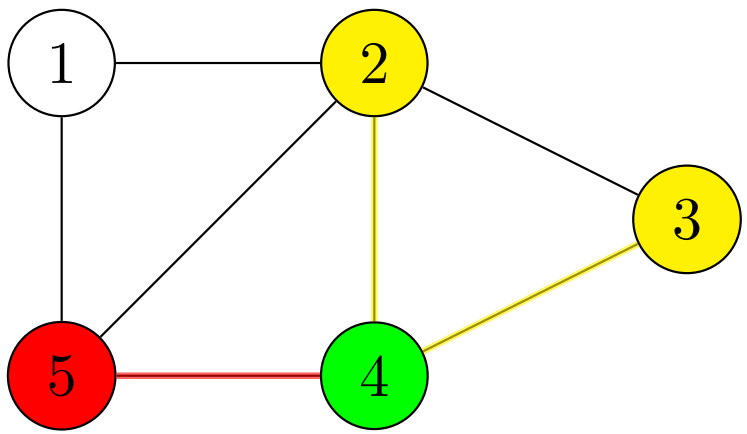
2	3

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4



kolejka FIFO:

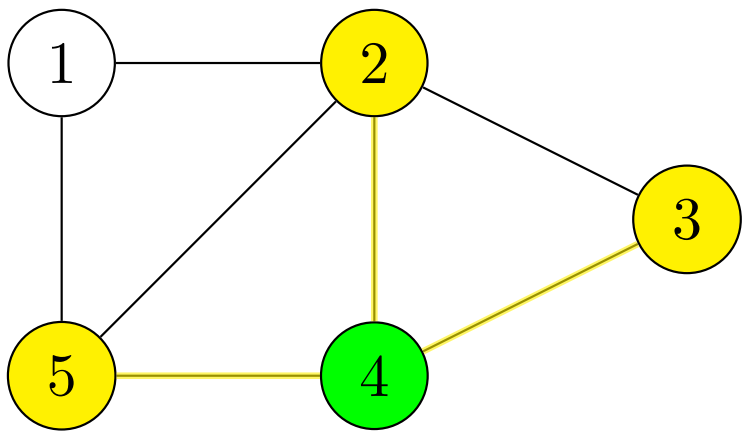
2	3

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4



kolejka FIFO:

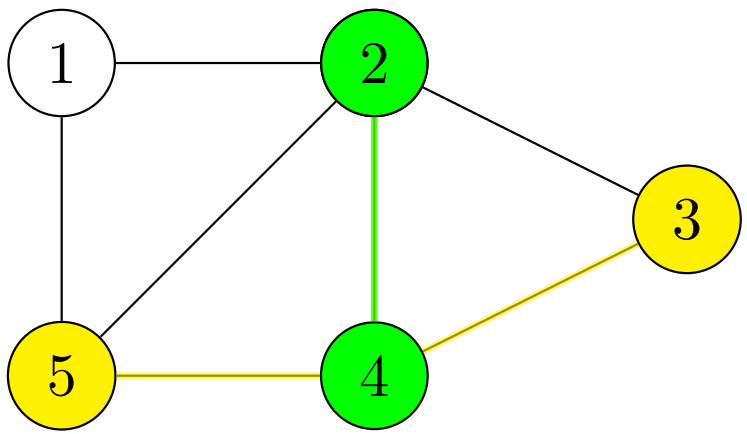
2	3	5

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4 2



kolejka FIFO:

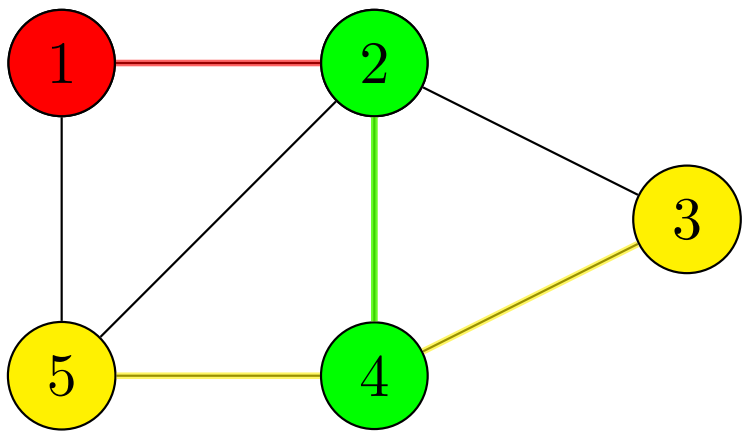
3	5

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4 2



kolejka FIFO:

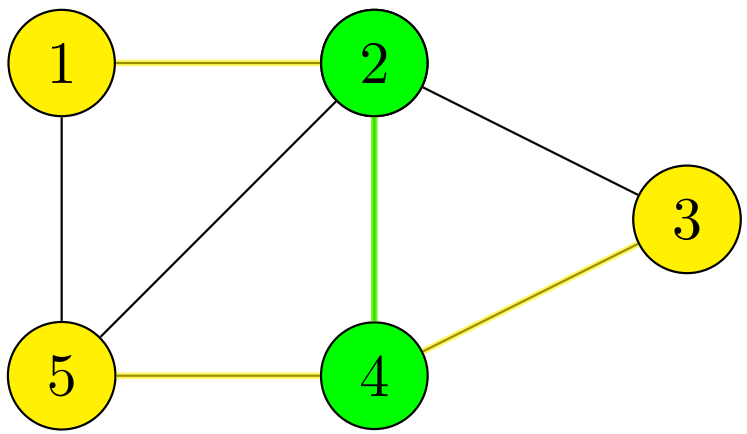
3	5

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4 2



kolejka FIFO:

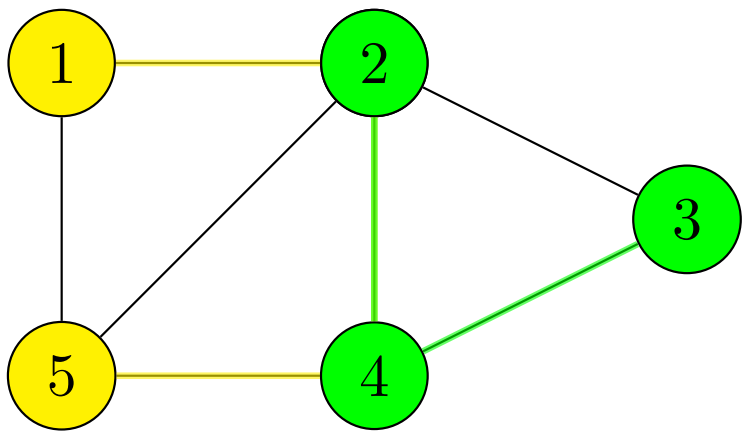
3	5	1

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4 2 3



kolejka FIFO:

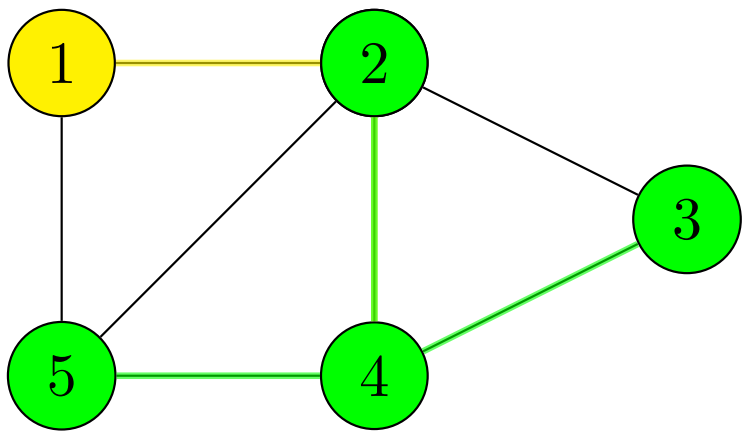
5	1

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4 2 3 5



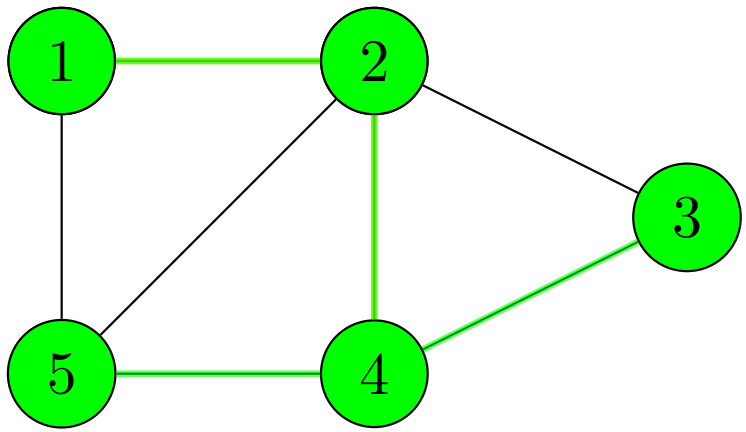
kolejka FIFO: 1

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS

Animacja

ścieżka: 4 2 3 5 1



kolejka FIFO:

Przechodzenie grafu nieskierowanego według algorytmu BFS

Algorytm BFS - implementacja

Kolejne slajdy przedstawiają wersję programu zaprezentowanego wcześniej, w której zamiast algorytmu DFS użyto BFS do przeszukiwania grafu. Ponieważ kod obu programów jest bardzo podobny, to zostaną opisane jedynie te miejsca, w których występują różnice.

Algorytm BFS - implementacja

Macierz sąsiedztwa i typ bazowy listy sąsiedztwa

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4
5  typedef int matrix[5][5];
6
7  const matrix adjacency_matrix = {{0,1,0,0,1},
8                                   {1,0,1,1,1},
9                                   {0,1,0,1,0},
10                                  {0,1,1,0,1},
11                                  {1,1,0,1,0}};
12
13 struct vertex
14 {
15     int vertex_number;
16     bool visited;
17     struct vertex *next, *down;
18 } *start_vertex;
```

Algorytm BFS - implementacja

Typ bazowy kolejki i struktura jej wskaźników

```
1  struct fifo_node
2  {
3      int vertex_number;
4      struct fifo_node *next;
5  };
6
7  struct fifo_pointers
8  {
9      struct fifo_node *head, *tail;
10 } path_fifo, discovered_fifo;
```

Algorytm BFS - implementacja

Typ bazowy kolejki i struktura jej wskaźników

Proszę zwrócić uwagę, że w stosunku do poprzedniego programu została zadeklarowana dodatkowa zmienna o nazwie `discovered_fifo`. Jest to struktura wskaźników na kolejkę wierzchołków, które zostały odkryte i będą odwiedzane w następnej kolejności.

Algorytm BFS - implementacja

Funkcja `enqueue()`

```
1  void enqueue(struct fifo_pointers *fifo, int vertex_number)
2  {
3      struct fifo_node *new_node = (struct fifo_node *)
4                                   malloc(sizeof(struct fifo_node));
5      if(new_node) {
6          new_node->vertex_number = vertex_number;
7          new_node->next = NULL;
8          if(fifo->head==NULL)
9              fifo->head = fifo->tail = new_node;
10         else {
11             fifo->tail->next=new_node;
12             fifo->tail=new_node;
13         }
14     } else
15         fprintf(stderr,"Nowy element nie został utworzony!\n");
16 }
```

Algorytm BFS - implementacja

Funkcja dequeue()

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

Algorytm BFS - implementacja

Funkcja `dequeue()`

Funkcja `dequeue()` została zmieniona w stosunku do swojej odpowiedniczki z poprzedniego programu tak, że zwraca teraz numer wierzchołka reprezentowanego przez usuwany z kolejki element. Jeśli kolejka byłaby pusta, to funkcja zwróci wartość `-1`.

Algorytm BFS - implementacja

Funkcja `remove_queue()`

```
1  void remove_queue(struct fifo_pointers *fifo)
2  {
3      while(fifo->head)
4          dequeue(fifo);
5  }
```

Algorytm BFS - implementacja

Funkcja `print_path()`

```
1  void print_path(struct fifo_pointers fifo)
2  {
3      while(fifo.head) {
4          printf("%d ",fifo.head->vertex_number);
5          fifo.head = fifo.head->next;
6      }
7      puts("");
8  }
```

Algorytm BFS - implementacja

Funkcja `create_vertical_list()`

```
1  void create_vertical_list(struct vertex **start_vertex,
2                            const matrix adjacency_matrix)
3  {
4      int i;
5      for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
6          *start_vertex = (struct vertex *)
7                          malloc(sizeof(struct vertex));
8          if(*start_vertex) {
9              (*start_vertex)->vertex_number = i+1;
10             (*start_vertex)->visited = false;
11             (*start_vertex)->down = (*start_vertex)->next = NULL;
12             start_vertex = &(*start_vertex)->down;
13         }
14     }
15 }
```

Algorytm BFS - implementacja

Funkcja `convert_matrix_to_list()`

```
1   struct vertex *convert_matrix_to_list(const matrix adjacency_matrix)
2   {
3       struct vertex *start_vertex = NULL;
4       create_vertical_list(&start_vertex,adjacency_matrix);
5       if(start_vertex) {
6           struct vertex *horizontal_pointer = NULL, *vertical_pointer = NULL;
7           horizontal_pointer = vertical_pointer = start_vertex;
8           int i,j;
9           for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
10              for(j=0; j<sizeof(matrix)/sizeof(*adjacency_matrix); j++)
11                  if(adjacency_matrix[i][j]) {
12                          struct vertex *new_vertex = (struct vertex *)malloc(sizeof(struct vertex));
13                          if(new_vertex) {
14                              new_vertex->vertex_number = j+1;
15                              new_vertex->visited = false;
16                              new_vertex->down = new_vertex->next = NULL;
17                              horizontal_pointer->next = new_vertex;
18                              horizontal_pointer = horizontal_pointer->next;
19                          }
20                  }
21                  vertical_pointer = vertical_pointer->down;
22                  horizontal_pointer = vertical_pointer;
23          }
24      }
25      return start_vertex;
26  }
```

Algorytm BFS - implementacja

Funkcja `print_adjacency_list()`

```
1  void print_adjacency_list(struct vertex *start_vertex)
2  {
3      while(start_vertex) {
4          printf("%3d:",start_vertex->vertex_number);
5          struct vertex *horizontal_pointer = start_vertex->next;
6          while(horizontal_pointer) {
7              printf("%3d",horizontal_pointer->vertex_number);
8              horizontal_pointer = horizontal_pointer->next;
9          }
10         start_vertex = start_vertex->down;
11         puts("");
12     }
13 }
```

Algorytm BFS - implementacja

Funkcja `remove_adjacency_list()`

```
1  void remove_adjacency_list(struct vertex **start_vertex)
2  {
3      while(*start_vertex) {
4          struct vertex *horizontal_pointer=(*start_vertex)->next;
5          while(horizontal_pointer) {
6              struct vertex *next_horizontal =
7                                          horizontal_pointer->next;
8              free(horizontal_pointer);
9              horizontal_pointer = next_horizontal;
10         }
11         struct vertex *next_vertical = (*start_vertex)->down;
12         free(*start_vertex);
13         *start_vertex= next_vertical;
14     }
15 }
```

Algorytm BFS - implementacja

Funkcja `find_vertex()`

```
1  struct vertex *find_vertex(struct vertex *start_vertex,
2                                                  int vertex_number)
3  {
4      while(start_vertex &&
5                      start_vertex->vertex_number!=vertex_number)
6          start_vertex = start_vertex->down;
7      return start_vertex;
8  }
```

Algorytm BFS - implementacja

Funkcja `has_not_been_visited()`

```
1  bool has_not_been_visited(struct vertex *start_vertex,
2                            struct vertex *vertex)
3  {
4      return !find_vertex(start_vertex,
5                          vertex->vertex_number)->visited;
6  }
```

Algorytm BFS - implementacja

Funkcja `bfs()`

```
1   void bfs(struct vertex *start_vertex, struct vertex *vertex,
2           struct fifo_pointers *path_fifo, struct fifo_pointers *discovered_fifo)
3   {
4       if(start_vertex && vertex) {
5           enqueue(discovered_fifo, vertex->vertex_number);
6           while(discovered_fifo->head) {
7               int vertex_number = dequeue(discovered_fifo);
8               vertex = find_vertex(start_vertex,vertex_number);
9               if(has_not_been_visited(start_vertex,vertex)) {
10                  struct vertex *next_vertex = vertex->next;
11                  while(next_vertex) {
12                      enqueue(discovered_fifo, next_vertex->vertex_number);
13                      next_vertex = next_vertex->next;
14                  }
15                  vertex->visited = true;
16                  enqueue(path_fifo, vertex->vertex_number);
17              }
18          }
19      }
20  }
```

Algorytm BFS - implementacja

Funkcja `bfs()`

Funkcja `bfs()` implementuje przeszukiwanie grafu przy pomocy algorytmu BFS. Funkcja ta nie zwraca żadnej wartości, ale przyjmuje cztery argumenty wywołania - adres wierzchołka startowego na liście sąsiedztwa grafu, adres wierzchołka, od którego algorytm powinien zacząć przeszukiwanie grafu, adres struktury wskaźników kolejki, w które zostanie zapisana wygenerowana przez algorytm ścieżka oraz adres kolejki służącej do zapisywania odkrytych wierzchołków. W wierszu nr 4 funkcja sprawdza, czy przekazane jej wskaźniki na wierzchołki grafu mają wartości różne od `NULL`. Jeśli tak jest, to zapisuje ona w kolejce `discovered_fifo` numer wierzchołka wskazywanego przez wskaźnik `vertex` (wiersz nr 5) i rozpoczyna zewnętrzną pętlę `while`, która wykonywana jest tak długo, jak długo kolejka wierzchołków odkrytych nie jest pusta (wiersz nr 6).

Algorytm BFS - implementacja

Funkcja `bfs()`

W tej pętli usuwany jest pierwszy element z kolejki `discovered_fifo` (wiersz nr 7). Jego numer, zapamiętany w zmiennej `vertex_number` jest następnie używany przez funkcję `find_vertex()` do ustalenia adresu elementu na liście wszystkich wierzchołków grafu, który reprezentuje ten wierzchołek. Ten adres zapamiętywany jest we wskaźniku `vertex`. W wierszu nr 9 funkcja sprawdza, czy wierzchołek nie był już odwiedzony. Jeśli nie był, to w lokalnym wskaźniku `next_vertex` zapamiętywany jest adres z pola `next` elementu reprezentującego ten wierzchołek. Jeśli wskaźnik `next_vertex` ma wartość różną od `NULL`, to oznacza to, że istnieje lista sąsiadów tego węzła. W wewnętrznej pętli `while` (wiersze 11-14) funkcja `bfs()` przegląda tę listę i umieszcza numery wierzchołków sąsiednich w kolejce `discovered_fifo`. Po zakończeniu tej pętli, nadal w pętli zewnętrznej, `bfs()` oznacza bieżący wierzchołek jako odwiedzony i zapisuje jego numer w kolejce `path_fifo`.

Algorytm BFS - implementacja

Funkcja `bfs()`

Funkcja `bfs()` kończy swe działanie, jeśli wszystkie wierzchołki osiągalne z wierzchołka, od którego rozpoczęła działanie zostały odwiedzone. Ścieżka wygenerowana przez tę funkcję jest zapisana w kolejce `path_fifo`.

Algorytm BFS - implementacja

Funkcja `visit_all_vertexes()`

```
1  void visit_all_vertexes(struct vertex *start_vertex)
2  {
3      struct vertex *vertex = start_vertex;
4      while(vertex) {
5          if(!vertex->visited) {
6              struct fifo_pointers path;
7              struct fifo_pointers discovered;
8              path.head = path.tail = discovered.head =
9                                              discovered.tail = NULL;
10             bfs(start_vertex, vertex, &path, &discovered);
11             print_path(path);
12             remove_queue(&path);
13             remove_queue(&discovered);
14         }
15         vertex = vertex->down;
16     }
17 }
```

Algorytm BFS - implementacja

Funkcja `visit_all_vertexes()`

Funkcja `visit_all_vertexes()` różni się od swojej odpowiedniczki z wcześniej zaprezentowanego programu tym, że zamiast funkcji `dfs()` wywołuje funkcję `bfs()` (wiersz nr 9) oraz używa dwóch kolejek FIFO. Struktura wskaźników drugiej kolejki jest zadeklarowana w wierszu nr 7, a zainicjowana w wierszach nr 8 i nr 9. Jest ona przekazywana do wywołania funkcji `bfs()` i służy do zapamiętywania odkrytych wierzchołków. Ta kolejka jest usuwana w wierszu nr 13.

Algorytm BFS - implementacja

Funkcja `main()`

```
1   int main(void)
2   {
3       start_vertex = convert_matrix_to_list(adjacency_matrix);
4       if(start_vertex) {
5           print_adjacency_list(start_vertex);
6           puts("Podaj numer wierzchołka początkowego");
7           int vertex_number = 0;
8           scanf("%d",&vertex_number);
9           bfs(start_vertex,find_vertex(start_vertex,vertex_number),
10                                          &path_fifo, &discovered_fifo);
11          puts("Wynik przechodzenia grafu algorytmem BFS:");
12          print_path(path_fifo);
13          remove_queue(&path_fifo);
14          remove_queue(&discovered_fifo);
15          visit_all_vertexes(start_vertex);
16          remove_adjacency_list(&start_vertex);
17      }
18      return 0;
19  }
```

Algorytm BFS - implementacja

Funkcja `main()`

W stosunku do wcześniej zaprezentowanego programu, funkcja `main()` zawiera dwie zmiany. Zamiast funkcji `dfs()` wywołuje ona funkcję `bfs()`, przekazując jej jako czwarty argument wywołania adres kolejki `discovered_fifo`. Ponadto dodano do funkcji `main()` instrukcję wywołującą funkcję `remove_queue()` dla tej kolejki (wiersz nr 14), celem jej usunięcia.

Podsumowanie

Algorytmy BFS i DFS mogą być zastosowane zarówno dla grafów skierowanych, spójnych i niespójnych, jaki i nieskierowanych, silnie spójnych lub niespójnych. Jak wspomniano na początku wykładu, z tych algorytmów wywodzi się wiele innych związanych z grafami, także inne algorytmy przeszukiwania grafów, takie jak *Best-First Search*, czy algorytm A^*. Początkowo BFS i DFS były używane do rozwiązywania problemów z dziedziny sztucznej inteligencji, ale z czasem znalazły szereg innych zastosowań.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!