Fundamentals of Programming 2

The DFS and BFS Algorithms

Arkadiusz Chrobot

Department of Computer Science

May 25, 2020

Outline

Introduction

There are many algorithms for processing data expressed as graphs that are generally known as graph algorithms. Most of them is based on one of two fundamental graph traversing algorithms. Results of such algorithms are paths that start in a specified vertex and cover all vertices of a connected or strongly connected graph or end in a goal vertex. The first of those two algorithms is the Deep-First Search algorithm called the DFS for short. The second one is the Breadth-First Search algorithm also known as the BFS. They are described in this lecture in the same order as they are introduced in this slide.

The DFS Algorithm

Theoretical Introduction

The DFS algorithm traverses a graph starting from a specified initial vertex and visiting all vertices that are reachable from this node. If a visited vertex has at least one neighbour (adjacent vertex), which has not been yet visited, then this neighbour is added to a stack, as the one to be visited next. In case the visited vertex has two or more unvisited neighbours, only one of them is selected. Nodes that are stored on the stack are called *discovered vertices*. The algorithm marks the current vertex as *visited*, takes a single vertex from the stack (provided the stack is not empty) and visits it repeating the steps described in this slide. If another algorithm is based on the DFS then it may process data stored in the currently visited vertex before marking it as visited.

The DFS Algorithm

Theoretical Introduction

The name of the algorithm comes from the way it visits graph nodes — it always choses one of the unvisited neighbours of the current vertex and visits it as next. In other words it “goes deeper” in the graph. If the visited vertex has no unvisited neighbours (or no neighbours at all) then the DFS *backtracks* (“goes back”) to the previous vertex and checks if there are any unvisited neighbours left. If the graph processed by the DFS is connected or strongly connected, then the algorithm will visit all its vertices. Otherwise, the DFS will visit a component (a maximal connected subgraph) that contains the initial vertex. In this case, if the objective of using the DFS is to visit all vertices of such a graph then the algorithm should be repeated for the unvisited vertices until all of them are visited.

The DFS Algorithm

Theoretical Introduction

The DFS algorithm can also be used for finding a path from an initial vertex to a specified *goal vertex* or a vertex that satisfies a *goal condition*. Because the DFS is a stack-based algorithm then it is easy to implement in a form of a recursive function. When this algorithm is applied to a binary tree it gives the same results as the pre-order traversal algorithm. It means that the DFS is a generalised pre-order traversal algorithm that can be applied to any graph. The time-complexity of the DFS algorithm is $\Theta(V + E)$.

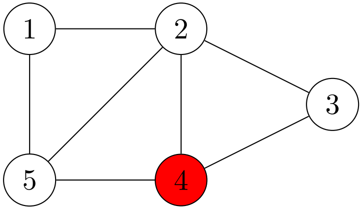
The DFS Algorithm
Animation

In the next slide is an animation that shows how the DFS algorithm works when applied to the undirected graph that has been presented in the previous lecture. At the top of the slide is a list of vertices which have been visited by the DFS and form a path that is the result of the algorithm. On the right is a stack, where the discovered vertices are stored. The order of visiting the neighbours is arbitrary. The vertex and the edge to be visited next are marked in red in the graph diagram. The processed vertices are marked in yellow and the vertices that have been already visited are marked in green. Because the graph is connected, the algorithm visits all its vertices in one go. There is no need to repeat it for unvisited vertices.

The DFS Algorithm

Animation

path:

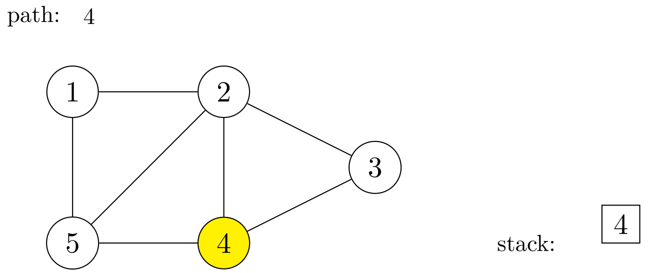


stack:

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

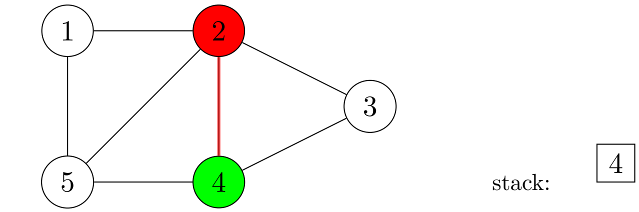
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm
Animation

path: 4

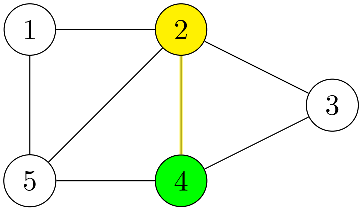


Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2



stack:

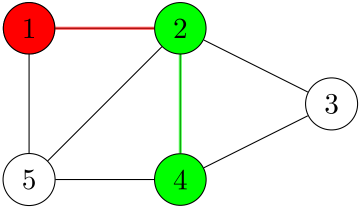
2
4

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2

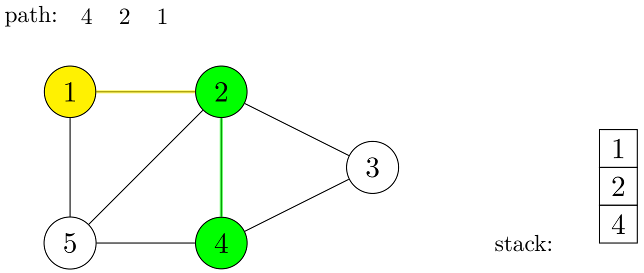


stack:

2
4

Traversing an undirected graph with the use of the DFS algorithm.

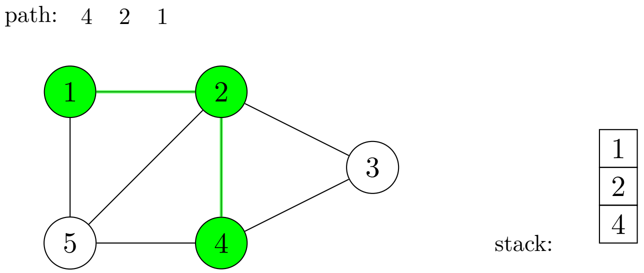
The DFS Algorithm
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

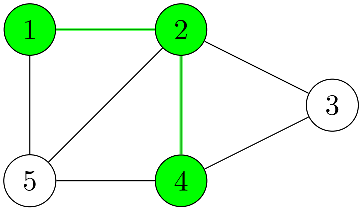


Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2 1



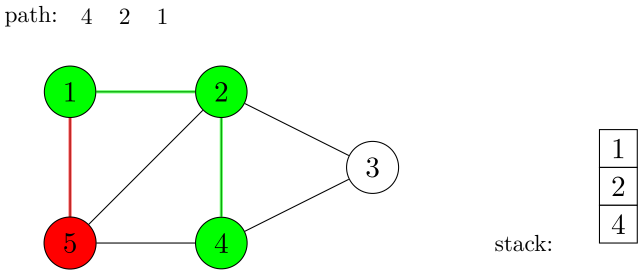
stack:

2
1
2
4

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

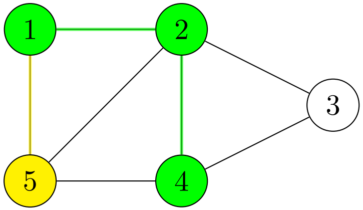


Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2 1 5



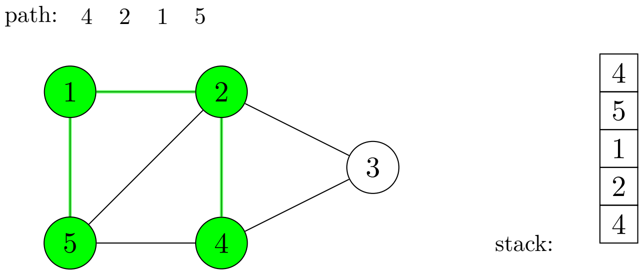
stack:

5
1
2
4

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

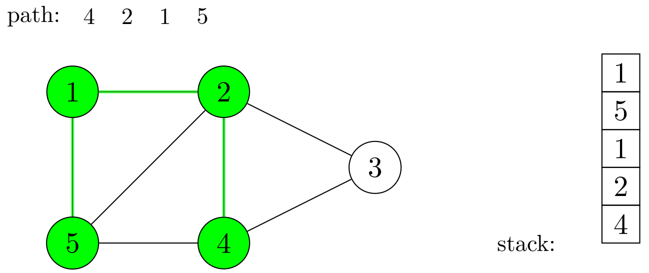
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

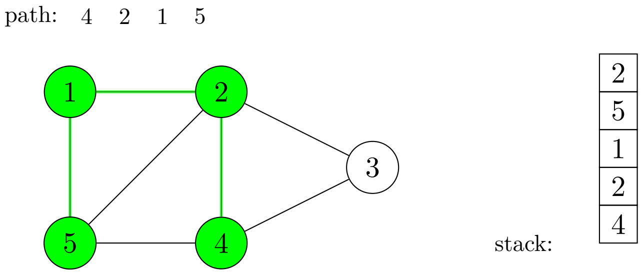
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

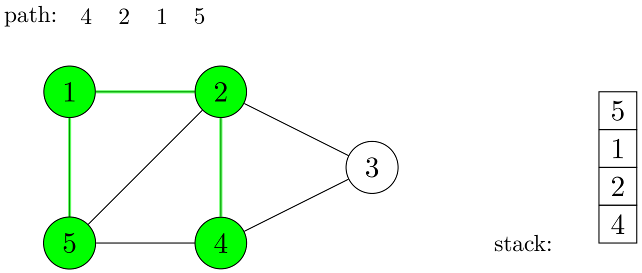
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

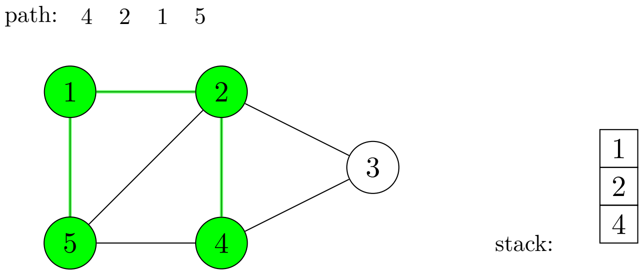
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

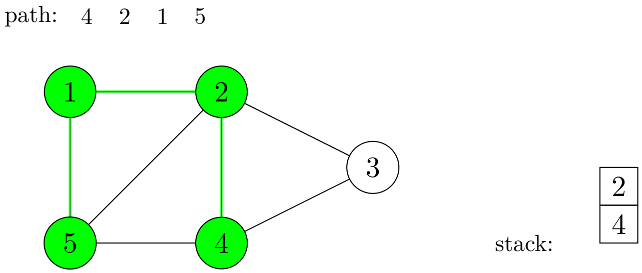
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

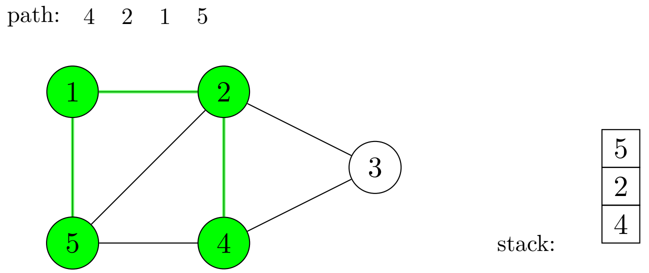
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

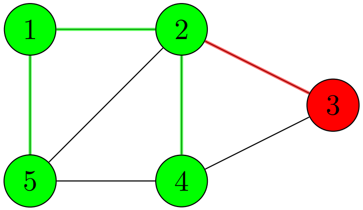


Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2 1 5



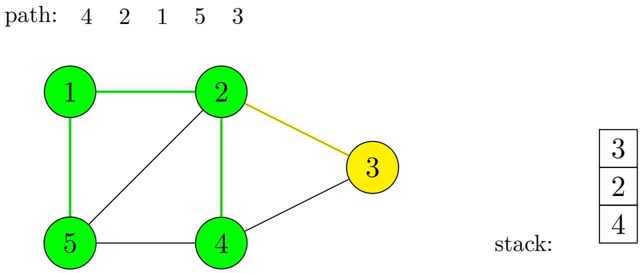
stack:

2
4

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

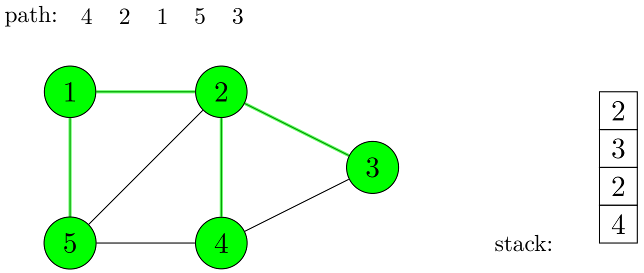
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

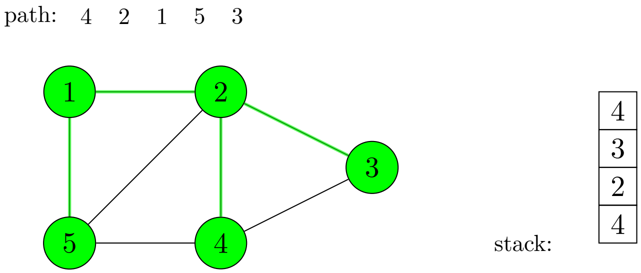
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

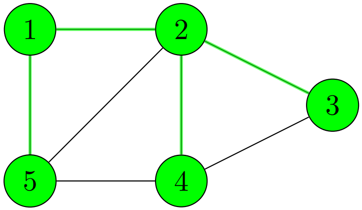


Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2 1 5 3

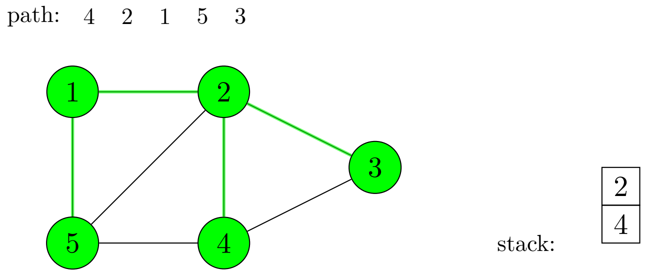


stack:

3
2
4

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm
Animation

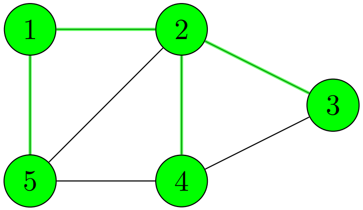


Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2 1 5 3



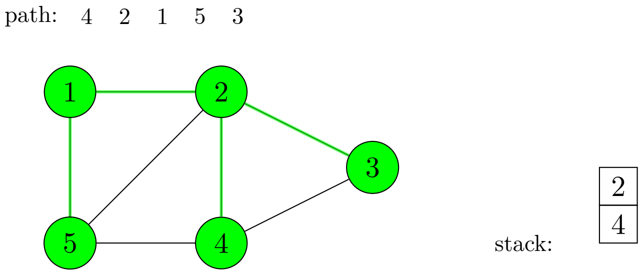
stack:

4
2
4

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

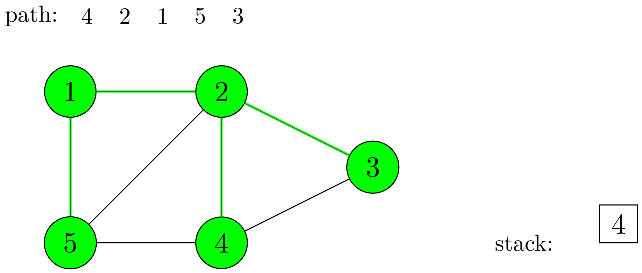
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

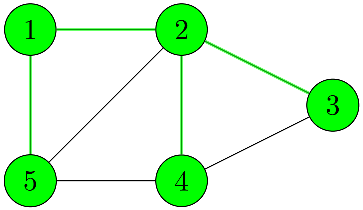


Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2 1 5 3



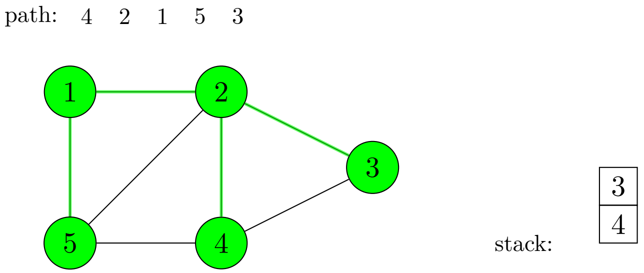
stack:

5
4

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

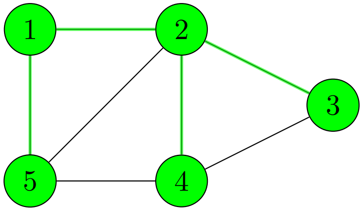
Animation



Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm
Animation

path: 4 2 1 5 3



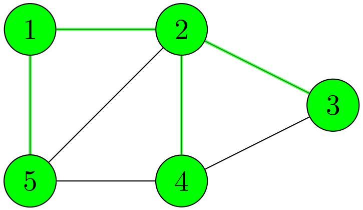
stack: $\boxed{4}$

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm

Animation

path: 4 2 1 5 3



stack:

Traversing an undirected graph with the use of the DFS algorithm.

The DFS Algorithm — Implementation

Next slides show a modified version of the program presented in the previous lecture which traverses an undirected graph using the DFS algorithm. The input for this algorithm is the graph adjacency list which is the result of converting the adjacency matrix. Because the graph is connected, the DFS visits all its vertices in one go, but the program is also prepared for processing graphs which are not connected.

The DFS Algorithm — Implementation

Adjacency Matrix and Base Type of Adjacency List

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

typedef int matrix[5][5];

const matrix adjacency_matrix = {{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,}};

struct vertex {
    int vertex_number;
    bool visited;
    struct vertex *next, *down;
} *start_vertex;
```

The DFS Algorithm — Implementation

Adjacency Matrix and Base Type of Adjacency List

Two new elements have been added to the beginning of the program. The first one is the preprocessor directive that includes the `stdbool.h` header file. The second one is a field in the adjacency list base type, called `visited`. The data type of this field is `bool`. The value of the field has a meaning only in the list of all graph vertices — the “vertical” list, which is a part of the adjacency list. If the vertex represented by the element of this list has been already visited then the value of its `visited` field is `true`, otherwise it is `false`.

The DFS Algorithm — Implementation

The Queue Base Type and the Queue Pointers Structure

```c
struct fifo_node {
    int vertex_number;
    struct fifo_node *next;
};

struct fifo_pointers {
    struct fifo_node *head, *tail;
} path;
```

The DFS Algorithm — Implementation

The Queue Base Type and the Queue Pointers Structure

In the previous slide are presented definitions of a FIFO queue element base type and a structure that stores the head and tail pointers of that queue. A global variable of the `fifo_pointers` type, named `path`, is also declared in this part of the program (8th line). The queue is used for storing the path that is the result of the DFS, hence each of its elements stores the number of a visited vertex.

The DFS Algorithm — Implementation

The enqueue() Function

```c
void enqueue(struct fifo_pointers *fifo, int vertex_number)
{
    struct fifo_node *new_node =
        (struct fifo_node *)malloc(sizeof(struct fifo_node));
    if(new_node) {
        new_node->vertex_number = vertex_number;
        new_node->next = NULL;
        if(fifo->head==NULL)
            fifo->head = fifo->tail = new_node;
        else {
            fifo->tail->next=new_node;
            fifo->tail=new_node;
        }
    } else
        fprintf(stderr,"No new element was created!\n");
}
```

The DFS Algorithm — Implementation

The `enqueue()` Function

In the previous slide is shown a definition of the `enqueue()` function, which adds an element to a FIFO queue. Please refer to the lecture on queues for more detailed description of this function.

The DFS Algorithm — Implementation

The `dequeue()` Function

```
void dequeue(struct fifo_pointers *fifo)
{
    if(fifo->head) {
        struct fifo_node *tmp = fifo->head->next;
        free(fifo->head);
        fifo->head=tmp;
        if(tmp==NULL)
            fifo->tail = NULL;
    }
}
```

The DFS Algorithm — Implementation

The `dequeue()` Function

The main difference between the `dequeue()` function presented in the previous slide and its equivalent presented in the lecture on queues is that the former returns no value, but only removes a single element from the head of the FIFO queue.

The DFS Algorithm — Implementation

The remove_queue() Function

```
void remove_queue(struct fifo_pointers *fifo)
{
    while(fifo->head)
        dequeue(fifo);
}
```

The DFS Algorithm — Implementation

The `remove_queue()` Function

The `remove_queue()` function calls in the `while` loop the `dequeue()` function to delete the FIFO queue. As an argument it takes the address of the queue pointers structure. It doesn't return any value. The `while` loop stops when the head pointer has a value of `NULL`.

The DFS Algorithm — Implementation

The `print_path()` Function

```c
void print_path(struct fifo_pointers fifo)
{
    while(fifo.head) {
        printf("%d ",fifo.head->vertex_number);
        fifo.head = fifo.head->next;
    }
    puts("");
}
```

The DFS Algorithm — Implementation

The `print_path()` Function

The `print_path()` function is basically the `print_queue()` function modified to print the content of the FIFO queue storing the path created by the DFS algorithm.

The DFS Algorithm — Implementation

The `create_vertical_list()` Function

```
void create_vertical_list(struct vertex **start_vertex,
                                                const matrix adjacency_matrix)
{
    int i;
    for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
        *start_vertex = (struct vertex *)
                                                malloc(sizeof(struct vertex));
        if(*start_vertex) {
            (*start_vertex)->vertex_number = i+1;
            (*start_vertex)->visited = false;
            (*start_vertex)->down = (*start_vertex)->next = NULL;
            start_vertex = &(*start_vertex)->down;
        }
    }
}
```

The DFS Algorithm — Implementation

The `create_vertical_list()` Function

The `create_vertical_list()` function differs from its equivalent from the previous lecture only in that it initializes (10th line) the `visited` field of each node representing a vertex in the list of all vertices.

The DFS Algorithm — Implementation

The `convert_matrix_to_list()` Function

```c
struct vertex *convert_matrix_to_list(const matrix adjacency_matrix)
{
    struct vertex *start_vertex = NULL;
    create_vertical_list(&start_vertex,adjacency_matrix);
    if(start_vertex) {
        struct vertex *horizontal_pointer = NULL, *vertical_pointer = NULL;
        horizontal_pointer = vertical_pointer = start_vertex;
        int i,j;
        for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
            for(j=0; j<sizeof(matrix)/sizeof(*adjacency_matrix); j++)
                if(adjacency_matrix[i][j]) {
                    struct vertex *new_vertex = (struct vertex *)malloc(sizeof(struct vertex));
                    if(new_vertex) {
                        new_vertex->vertex_number = j+1;
                        new_vertex->visited = false;
                        new_vertex->down = new_vertex->next = NULL;
                        horizontal_pointer->next = new_vertex;
                        horizontal_pointer = horizontal_pointer->next;
                    }
                }
            vertical_pointer = vertical_pointer->down;
            horizontal_pointer = vertical_pointer;
        }
    }
    return start_vertex;
}
```

The DFS Algorithm — Implementation

The `convert_matrix_to_list()` Function

The function that converts an adjacency matrix into an adjacency list also differs by only one detail from its equivalent from the previous lecture. This detail is the initialisation of the `visited` field (15th line) of each newly created node of the neighbours lists ("vertical" lists in the adjacency list).

The DFS Algorithm — Implementation

The `print_adjacency_list()` Function

```c
void print_adjacency_list(struct vertex *start_vertex)
{
    while(start_vertex) {
        printf("%3d:",start_vertex->vertex_number);
        struct vertex *horizontal_pointer = start_vertex->next;
        while(horizontal_pointer) {
            printf("%3d",horizontal_pointer->vertex_number);
            horizontal_pointer = horizontal_pointer->next;
        }
        start_vertex = start_vertex->down;
        puts("");
    }
}
```

The DFS Algorithm — Implementation

The `print_adjacency_list()` Function

The `print_adjacency_list()` function is implemented in exactly the same way as its equivalent from the previous lecture.

The DFS Algorithm — Implementation

The remove_adjacency_list() Function

```
void remove_adjacency_list(struct vertex **start_vertex)
{
    while(*start_vertex) {
        struct vertex *horizontal_pointer=(*start_vertex)->next;
        while(horizontal_pointer) {
            struct vertex *next_horizontal =
                                           horizontal_pointer->next;
            free(horizontal_pointer);
            horizontal_pointer = next_horizontal;
        }
        struct vertex *next_vertical = (*start_vertex)->down;
        free(*start_vertex);
        *start_vertex= next_vertical;
    }
}
```

The DFS Algorithm — Implementation

The `remove_adjacency_list()` Function

The `remove_adjacency_list()` function is also implemented in the same way as its equivalent from the previous lecture.

The DFS Algorithm — Implementation

The `find_vertex()` Function

```c
struct vertex *find_vertex(struct vertex *start_vertex,
                                              int vertex_number)
{
    while(start_vertex &&
                     start_vertex->vertex_number!=vertex_number)
        start_vertex = start_vertex->down;
    return start_vertex;
}
```

The DFS Algorithm — Implementation

The `find_vertex()` Function

The `find_vertex()` function is a helper subroutine for the function that implements the DFS algorithm. Its task is to locate a node in the list of all vertices (the "vertical" list) that represents a vertex of a specified number. It takes as arguments the address of the adjacency list starting node and the number of the sought vertex. The function traverses the list of all vertices using the `while` loop and the `start_vertex` parameter (lines no. 4–6) and it checks if the node currently pointed by this parameter stores the sought number. If not, it advances to the next element of the list, otherwise the loop stops and the function returns the address of the located node that represents the sought vertex. The function returns the `NULL` value when its first argument is an empty pointer or the second argument is a number of a nonexistent vertex.

The DFS Algorithm — Implementation

The has_not_been_visited() Function

```c
bool has_not_been_visited(struct vertex *start_vertex,
                                         const struct vertex *vertex)
{
    return !find_vertex(start_vertex,
                                     vertex->vertex_number)->visited;
}
```

The DFS Algorithm — Implementation

The `has_not_been_visited()` Function

The `has_not_been_visited()` function checks if a specified vertex is unvisited. It takes two arguments: the address of the adjacency list starting node and the address of the node representing the verified vertex in the list of neighbours (one of the "horizontal" lists) of the currently visited vertex. The `has_not_been_visited()` function invokes the `find_vertex()` function to locate the node representing the verified vertex in the list of all vertices (the "vertical" list). The latter function returns a pointer to the sought element, which is immediately dereferenced[1] (5th line) to get the value of the `visited` field of the verified vertex. The `has_not_been_visited()` function returns a negated value of that field.

[1]It is not the best idea — the `find_vertex()` function may return the `NULL` value.

The DFS Algorithm — Implementation

The **dfs()** Function

```c
void dfs(struct vertex *start_vertex, struct vertex *vertex,
                                        struct fifo_pointers *fifo)
{
    if(start_vertex && vertex) {
        enqueue(fifo, vertex->vertex_number);
        vertex->visited = true;
        while(vertex) {
            vertex = vertex->next;
            if(vertex &&
                      has_not_been_visited(start_vertex,vertex))
                dfs(start_vertex,find_vertex(start_vertex,
                                   vertex->vertex_number),fifo);
        }
    }
}
```

The DFS Algorithm — Implementation

The `dfs()` Function

The `dfs()` function, as its name suggests, implements the DFS algorithm. It doesn't return any value and takes three arguments. The first one is the address of the adjacency list starting node. The second one is the address of the initial vertex for the DFS algorithm. The last argument is the address of the FIFO queue pointers structure. The queue is used for storing the path created by the function. In the 4th line the function verifies if the addresses of vertices passed by its parameters are not `NULL`. If so, then it adds to the FIFO queue a new element that stores the number of the vertex that is represented by a node pointed by the `vertex` parameter (5th line). Next, the function marks the vertex as visited by storing the `true` value in the `visited` field (6th line). Then it performs the `while` loop. Inside that loop the function takes the address stored in the `next` field of a node pointed by the `vertex` pointer and assigns it to that pointer (8th line).

The DFS Algorithm — Implementation

The `dfs()` Function

If the address is not `NULL`, then it means that the currently visited vertex has at least one neighbour and now the `vertex` pointer points to a node in the neighbours list that represents the first of them. In lines no. 9 and no. 10 the `dfs()` function verifies the existence of that vertex and if it is unvisited. If both conditions are met then the function invokes itself recursively for the neighbour vertex. This time as the second argument it takes the result of the `find_vertex()` function which returns the address of the node representing the neighbour vertex in the list of all vertices of the graph (the “vertical” list). After the `dfs()` function returns from the recursive call a next iteration of the `while` loop is performed. If another unvisited neighbour of the current vertex exists then the `dfs()` function calls itself recursively again, but this time for that neighbour vertex.

The DFS Algorithm — Implementation

The visit_all_vertexes() Function

```c
void visit_all_vertexes(struct vertex *start_vertex)
{
    struct vertex *vertex = start_vertex;
    while(vertex) {
        if(!vertex->visited) {
            struct fifo_pointers path;
            path.head = path.tail = NULL;
            dfs(start_vertex,vertex,&path);
            print_path(path);
            remove_queue(&path);
        }
        vertex = vertex->down;
    }
}
```

The DFS Algorithm — Implementation

The `visit_all_vertexes()` Function

The `visit_all_vertexes()` function is invoked after the `dfs()` function exits. It verifies if all the vertices of the graph have been visited and it calls the `dfs()` function for those of them that haven't been. The function returns no value, but takes one argument which is the address of the adjacency list starting node. In the 3rd line a local pointer named `vertex` is declared and initialized with the address stored in the `start_vertex` parameter. Although the function could used the `start_vertex` parameter for traversing the list of all vertices, without any consequences for the rest of the program, since its argument is passed by value, it is necessary to use another pointer for that purpose, because the address of the adjacency list starting node is used in another part of the function.

The DFS Algorithm — Implementation

The `visit_all_vertexes()` Function

In the `while` loop the function traverses the list of all vertices of the graph (the “vertical” list) using the `vertex` pointer and checks if any of them has not been yet visited (line no. 5). If so, then in the 8th line the `dfs()` function is invoked for that unvisited vertex. The address of the FIFO queue pointers structure declared in the 6th line and initialised in the 7th line is passed as the last argument of the `dfs()` function. In other words the `visit_all_vertexes()` function uses its own local FIFO queue. After the `dfs()` function exits the content of the queue is displayed on the screen and the queue is deleted, allowing the queue pointers structure to be used again for creating another instance of the queue in case some unvisited vertices are still left in the graph. The `while` loop stops after each node in the list of all vertices has been verified, which assures that the whole graph is traversed.

The DFS Algorithm — Implementation

The `main()` Function

```c
int main(void)
{
    start_vertex = convert_matrix_to_list(adjacency_matrix);
    if(start_vertex) {
        print_adjacency_list(start_vertex);
        puts("Please enter the number of the initial vertex:");
        int vertex_number = 0;
        scanf("%d",&vertex_number);
        dfs(start_vertex,find_vertex(start_vertex,vertex_number), &path);
        puts("The DFS algorithm result:");
        print_path(path);
        remove_queue(&path);
        visit_all_vertexes(start_vertex);
        remove_adjacency_list(&start_vertex);
    }
    return 0;
}
```

The DFS Algorithm — Implementation

The main() Function

In the 6th line the `main()` function displays a message asking the user to enter the number of the initial vertex for the DFS algorithm. The number is stored in a local variable named `vertex_number` (8th line). Next, the `dfs()` function is called. The second argument of that function — the address of the node representing the initial vertex in the list of all vertices — is returned by the `find_vertex()` function. The path returned by the `dfs()` function is displayed on the screen (lines no. 10 and 11) and the program deletes the queue that stores the path (12th line). In the 13th line the `main()` function calls the `visit_all_vertexes()` function to visit all the unvisited vertices of the graph. The rest of the `main()` function is the same as in the program presented in the previous lecture.

The BFS Algorithm

Theoretical Introduction

The BFS algorithm, just like the DFS algorithm, traverses a graph. The main difference between those two algorithms is that the BFS uses a FIFO queue instead of a stack to store the discovered vertices. When visiting a vertex the algorithm adds *all its unvisited neighbours* to this queue. After marking the current vertex as visited the BFS algorithm removes and visits the first discovered vertex from the head of the queue. All neighbours of every vertex are visited before any other vertices, hence the name of this algorithm: Breadth-First Search. The algorithm is usually implemented in a form of an iterative function. Its time-complexity is $O(V + E)$.

The BFS Algorithm

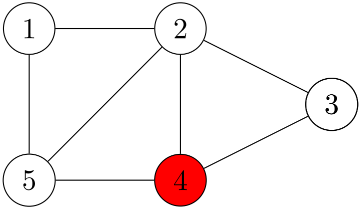
Animation

In the next slide is an animation that shows how the BFS algorithm works when it is applied for an undirected graph — it is the same graph as in the case of the DFS algorithm. In the animation the FIFO queue is used instead of the stack. It is shown at the bottom of the slide. All other elements of the animation are the same as for the DFS algorithm.

The BFS Algorithm

Animation

path:



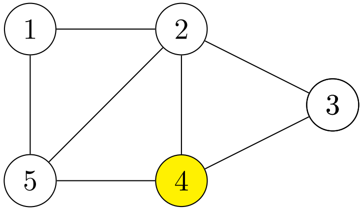
FIFO queue:

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path:



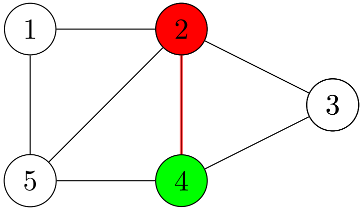
FIFO queue: $\boxed{4}$

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4

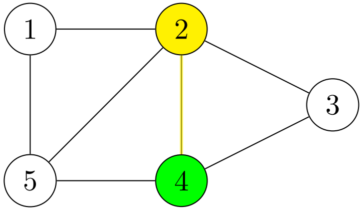


FIFO queue:

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm
Animation

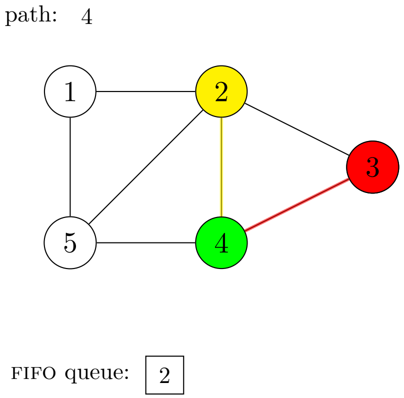
path: 4



FIFO queue: $\boxed{2}$

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm
Animation

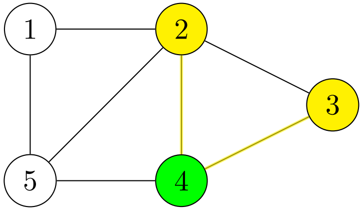


Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4



FIFO queue:

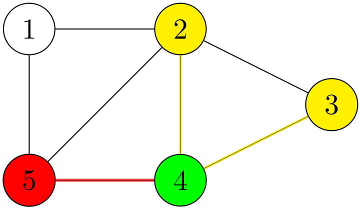
2	3

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4



FIFO queue:

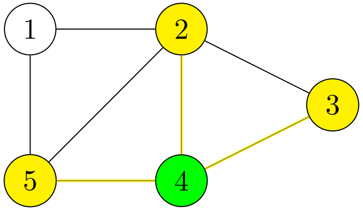
2	3

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4



FIFO queue:

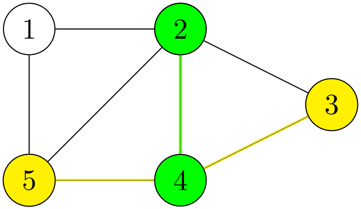
2	3	5

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4 2



FIFO queue:

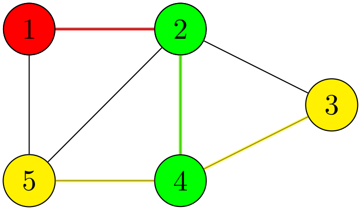
3	5

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4 2



FIFO queue:

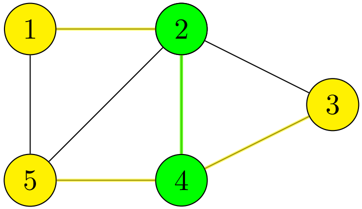
3	5

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4 2



FIFO queue:

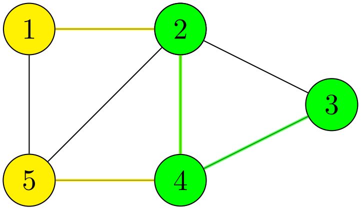
3	5	1

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4 2 3



FIFO queue:

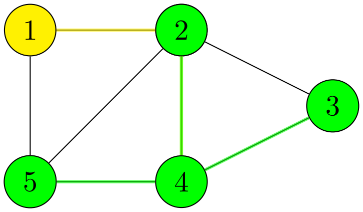
5	1

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4 2 3 5



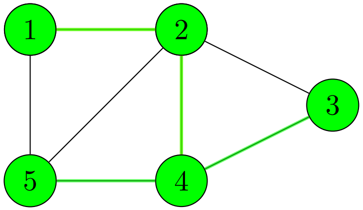
FIFO queue: $\boxed{1}$

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm

Animation

path: 4 2 3 5 1



FIFO queue:

Traversing an undirected graph with the use of the BFS algorithm.

The BFS Algorithm — Implementation

Next slides present the program demonstrated earlier, but this time it uses the BFS algorithm instead of the DFS for traversing a graph. Only those elements that have been modified are described.

The BFS Algorithm — Implementation

Adjacency Matrix and Base Type of Adjacency List

```c
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

typedef int matrix[5][5];

const matrix adjacency_matrix = {{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}};

struct vertex
{
    int vertex_number;
    bool visited;
    struct vertex *next, *down;
} *start_vertex;
```

The BFS Algorithm — Implementation

The Queue Base Type and the Queue Pointers Structure

```
struct fifo_node
{
    int vertex_number;
    struct fifo_node *next;
};

struct fifo_pointers
{
    struct fifo_node *head, *tail;
} path_fifo, discovered_fifo;
```

The BFS Algorithm — Implementation

The Queue Base Type and the Queue Pointers Structure

Please notice, that an additional variable, named `discovered_fifo`, is declared in this part of code. It is a structure of pointers for the queue of discovered vertices.

The BFS Algorithm — Implementation

The enqueue() Function

```c
void enqueue(struct fifo_pointers *fifo, int vertex_number)
{
    struct fifo_node *new_node = (struct fifo_node *)
                                malloc(sizeof(struct fifo_node));
    if(new_node) {
        new_node->vertex_number = vertex_number;
        new_node->next = NULL;
        if(fifo->head==NULL)
            fifo->head = fifo->tail = new_node;
        else {
            fifo->tail->next=new_node;
            fifo->tail=new_node;
        }
    } else
        fprintf(stderr,"No new element was created!\n");
}
```

The BFS Algorithm — Implementation

The **dequeue()** Function

```c
int dequeue(struct fifo_pointers *fifo)
{
    int vertex_number = -1;
    if(fifo->head) {
        struct fifo_node *tmp = fifo->head->next;
        vertex_number = fifo->head->vertex_number;
        free(fifo->head);
        fifo->head=tmp;
        if(tmp==NULL)
            fifo->tail = NULL;
    }
    return vertex_number;
}
```

The BFS Algorithm — Implementation

The `dequeue()` Function

The `dequeue()` function, unlike its equivalent from the earlier program, returns the number of the vertex represented by the element removed from the queue. If the queue is empty the function returns `-1`.

The BFS Algorithm — Implementation

The `remove_queue()` Function

```c
void remove_queue(struct fifo_pointers *fifo)
{
    while(fifo->head)
        dequeue(fifo);
}
```

The BFS Algorithm — Implementation

The `print_path()` Function

```c
void print_path(struct fifo_pointers fifo)
{
    while(fifo.head) {
        printf("%d ",fifo.head->vertex_number);
        fifo.head = fifo.head->next;
    }
    puts("");
}
```

The BFS Algorithm — Implementation

The `create_vertical_list()` Function

```c
void create_vertical_list(struct vertex **start_vertex,
                                     const matrix adjacency_matrix)
{
    int i;
    for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
        *start_vertex = (struct vertex *)
                                     malloc(sizeof(struct vertex));
        if(*start_vertex) {
            (*start_vertex)->vertex_number = i+1;
            (*start_vertex)->visited = false;
            (*start_vertex)->down = (*start_vertex)->next = NULL;
            start_vertex = &(*start_vertex)->down;
        }
    }
}
```

The BFS Algorithm — Implementation

The convert_matrix_to_list() Function

```c
struct vertex *convert_matrix_to_list(const matrix adjacency_matrix)
{
    struct vertex *start_vertex = NULL;
    create_vertical_list(&start_vertex,adjacency_matrix);
    if(start_vertex) {
        struct vertex *horizontal_pointer = NULL, *vertical_pointer = NULL;
        horizontal_pointer = vertical_pointer = start_vertex;
        int i,j;
        for(i=0; i<sizeof(matrix)/sizeof(*adjacency_matrix); i++) {
            for(j=0; j<sizeof(matrix)/sizeof(*adjacency_matrix); j++)
                if(adjacency_matrix[i][j]) {
                        struct vertex *new_vertex = (struct vertex *)malloc(sizeof(struct vertex));
                        if(new_vertex) {
                            new_vertex->vertex_number = j+1;
                            new_vertex->visited = false;
                            new_vertex->down = new_vertex->next = NULL;
                            horizontal_pointer->next = new_vertex;
                            horizontal_pointer = horizontal_pointer->next;
                        }
                }
            vertical_pointer = vertical_pointer->down;
            horizontal_pointer = vertical_pointer;
        }
    }
    return start_vertex;
}
```

The BFS Algorithm — Implementation

The print_adjacency_list() Function

```c
void print_adjacency_list(struct vertex *start_vertex)
{
    while(start_vertex) {
        printf("%3d:",start_vertex->vertex_number);
        struct vertex *horizontal_pointer = start_vertex->next;
        while(horizontal_pointer) {
            printf("%3d",horizontal_pointer->vertex_number);
            horizontal_pointer = horizontal_pointer->next;
        }
        start_vertex = start_vertex->down;
        puts("");
    }
}
```

The BFS Algorithm — Implementation

The remove_adjacency_list() Function

```c
void remove_adjacency_list(struct vertex **start_vertex)
{
    while(*start_vertex) {
        struct vertex *horizontal_pointer=(*start_vertex)->next;
        while(horizontal_pointer) {
            struct vertex *next_horizontal =
                                           horizontal_pointer->next;
            free(horizontal_pointer);
            horizontal_pointer = next_horizontal;
        }
        struct vertex *next_vertical = (*start_vertex)->down;
        free(*start_vertex);
        *start_vertex= next_vertical;
    }
}
```

The BFS Algorithm — Implementation

The `find_vertex()` Function

```c
struct vertex *find_vertex(struct vertex *start_vertex,
                                              int vertex_number)
{
    while(start_vertex &&
                     start_vertex->vertex_number!=vertex_number)
        start_vertex = start_vertex->down;
    return start_vertex;
}
```

The BFS Algorithm — Implementation

The `has_not_been_visited()` Function

```c
bool has_not_been_visited(struct vertex *start_vertex,
                                                struct vertex *vertex)
{
    return !find_vertex(start_vertex,
                                        vertex->vertex_number)->visited;
}
```

The BFS Algorithm — Implementation

The **bfs()** Function

```
void bfs(struct vertex *start_vertex, struct vertex *vertex,
        struct fifo_pointers *path_fifo, struct fifo_pointers *discovered_fifo)
{
    if(start_vertex && vertex) {
        enqueue(discovered_fifo, vertex->vertex_number);
        while(discovered_fifo->head) {
            int vertex_number = dequeue(discovered_fifo);
            vertex = find_vertex(start_vertex,vertex_number);
            if(has_not_been_visited(start_vertex,vertex)) {
                struct vertex *next_vertex = vertex->next;
                while(next_vertex) {
                    enqueue(discovered_fifo, next_vertex->vertex_number);
                    next_vertex = next_vertex->next;
                }
                vertex->visited = true;
                enqueue(path_fifo, vertex->vertex_number);
            }
        }
    }
}
```

The BFS Algorithm — Implementation

The `bfs()` Function

The `bfs()` function implements the BFS graph traversal algorithm. It returns no value but takes four arguments — an address of the adjacency list starting node, an address of the initial vertex for the BFS algorithm, an address of the pointers structure for the queue where the path created by the algorithm will be stored and an address for another pointers structure, but this time for the queue which will store the discovered vertices. In the 4th line the function verifies if the first two addresses passed by its parameters are not `NULL`. If so, it adds to the `discovered_fifo` queue a new element storing the number of the vertex also stored in the node pointed by the `vertex` parameter (5th line) and starts the outer `while` loop, which is repeated until the queue of discovered vertices is empty (6th line).

The BFS Algorithm — Implementation

The `bfs()` Function

Inside this loop the `bfs()` function removes the first element from the `discovered_fifo` queue (7th line) and assigns the vertex number stored in that element to the `vertex_number` variable, which is then used by the `find_vertex()` function to find the address of the node representing that vertex in the list of all vertices. This address is assigned to the `vertex` pointer. In the 9th line the function checks if the vertex is unvisited. If so, then it assigns the address stored in the `next` field of the node pointed by the `vertex` variable to the `next_vertex` pointer. If after the assignment the value of this pointer is not `NULL` then it means that the vertex has a nonempty list of its neighbours and that the `next_vertex` pointer points to the first element of this list. Inside the inner `while` loop (lines no. 11–14) the `bfs()` function traverses that list, gets the vertex numbers stored in its elements and adds them to the `discovered_fifo` queue.

The BFS Algorithm — Implementation

The `bfs()` Function

When the inner loop stops the `bfs()` function marks the current vertex (pointed by the `vertex` pointer) as visited and stores its number in the `path_fifo` queue. The function exits when all vertices reachable from the initial vertex are visited. The result of this function is the path stored in the `path_fifo` queue.

The BFS Algorithm — Implementation

The visit_all_vertexes() Function

```c
void visit_all_vertexes(struct vertex *start_vertex)
{
    struct vertex *vertex = start_vertex;
    while(vertex) {
        if(!vertex->visited) {
            struct fifo_pointers path;
            struct fifo_pointers discovered;
            path.head = path.tail = discovered.head =
                                          discovered.tail = NULL;
            bfs(start_vertex, vertex, &path, &discovered);
            print_path(path);
            remove_queue(&path);
            remove_queue(&discovered);
        }
        vertex = vertex->down;
    }
}
```

The BFS Algorithm — Implementation

The `visit_all_vertexes()` Function

The `visit_all_vertexes()` function differs from its equivalent from the previous program in that it invokes the `bfs()` function instead of the `dfs()` function (10th line) and it uses two FIFO queues instead of one. The pointers structure of the second queue is declared in the 7th line and initialised in lines no. 8 and no. 9. This queue is used by the `bfs()` function for storing the discovered vertices and then it is deleted in the 13th line.

The BFS Algorithm — Implementation

The `main()` Function

```c
int main(void)
{
    start_vertex = convert_matrix_to_list(adjacency_matrix);
    if(start_vertex) {
        print_adjacency_list(start_vertex);
        puts("Please enter the number of the initial vertex:");
        int vertex_number = 0;
        scanf("%d",&vertex_number);
        bfs(start_vertex,find_vertex(start_vertex,vertex_number),
                                            &path_fifo, &discovered_fifo);
        puts("The BFS algorithm result:");
        print_path(path_fifo);
        remove_queue(&path_fifo);
        remove_queue(&discovered_fifo);
        visit_all_vertexes(start_vertex);
        remove_adjacency_list(&start_vertex);
    }
    return 0;
}
```

The BFS Algorithm — Implementation

The `main()` Function

There are two changes in the `main()` function. First of all it calls the `bfs()` function instead of the `dfs()` function. It also declares and initializes the `discovered_fifo` queue pointers structure, which is then used by the `bfs()` function. The function that deletes the discovered vertices queue is called in the 14th line, although if the `bfs()` function completes its job successfully then this queue should already be empty.

Summary

The BFS and DFS algorithms can be applied either to undirected or directed graphs and it doesn't matter if those graphs are connected, strongly connected or disconnected. The two algorithms are the basis of many other graph algorithms. Among them is a family of heuristic graph traversal algorithms known as *Best-First Search* algorithms. One of them is the A^* algorithm. Initially the BFS and DFS algorithms were used in the Artificial Intelligence, but nowadays they are applied in many other branches of Computer Science.

Questions

?

The End

Thank You For Your Attention!