Fundamentals of Programming 2

The DFS and BFS Algorithms

Arkadiusz Chrobot

Department of Computer Science

June 13, 2019

Outline

1. Introduction
2. The DFS Algorithm
3. The BFS Algorithm
4. Summary

Introduction

There are many algorithms for graphs that are known simply as the graph algorithms. Most of them is based on or is a modification of two fundamental graph traversing algorithms. The result of such algorithms is a path that starts in a given vertex and covers all vertices of a connected or strongly connected graph or ends with a specified goal vertex. The first of those two algorithms is the Deep-First Search algorithm called DFS for short. The second one is the Breadth-First Search algorithm also known as BFS. They are described in the lecture in the same order as they are mentioned in this slide.

The DFS Algorithm

Theoretical Introduction

The DFS algorithm starts traversing the graph from a specified vertex and visits all vertices that are reachable from this node. If a vertex belonging to those vertices has at least one neighbour (adjacent vertex), which has not been yet visited, then the neighbour is added to a stack, as the one to be visited next. In case the vertex has more than one unvisited neighbour, only one is selected in this step of the algorithm. The vertices that are on the stack are described as *discovered*. After marking the current vertex as a *visited* the DFS algorithm removes a single vertex from the stack (provided the data structure is not empty) and visits it repeating the steps described in this slide. If the DFS is used as a basis for another algorithm then before the current vertex is marked as visited the data stored in it are processed.

The DFS Algorithm

Theoretical Introduction

The name of the algorithm comes from how it works — 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 current vertex has no unvisited neighbours (or no neighbours at all) then the DFS *backtracks* ("goes back") to the previous vertex and checks if it has any neighbours that it should visit. If the graph traversed by the DFS is a connected or strongly connected one, then the algorithm will terminate after visiting all vertices. If not, the DFS will finish after visiting a component (a maximal connected subgraph) to which the initial vertex belongs. If the objective of using the DFS algorithm is to visit all vertices then after applying it once for a given graph it should be checked if there are still any unvisited vertices. If so, the algorithm should be repeatedly applied for those vertices until none of them is left unvisited.

The DFS Algorithm

Theoretical Introduction

The DFS algorithm can also be terminated after visiting a specified *goal vertex* or a vertex that satisfies the *goal condition*. Since the DFS uses a stack then it is easy to implement it in a recursive form. If the algorithm is applied to a binary tree it gives the same results as the pre-order traversing algorithm. Thus, the DFS is a generalised for all kinds of graphs version of the pre-order traversing algorithm. The time-complexity of the DFS algorithm is $\Theta(V + E)$.

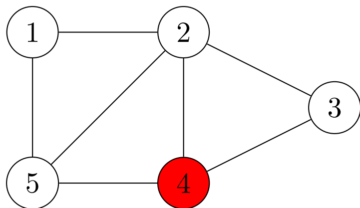
The DFS Algorithm

Animation

In the next slide is an animation that shows the behaviour of the DFS algorithm for the undirected graph that has been presented in the previous lecture. At the top of the slide is a list of vertices which are visited by the DFS and form a path that is the result of the algorithm. On the right is illustrated a stack, where the numbers of discovered vertices are stored. The order of visiting the neighbours is arbitrary. The red marked vertex of the graph in the middle of the slide, together with an associated edge is the vertex to be visited next. The vertex marked in yellow colour is being processed and the vertex marked in green colour is already visited. Since the graph is connected, the algorithm terminates after visiting all its vertices. There is no need to repeat it for unvisited vertices.

The DFS Algorithm

Animation

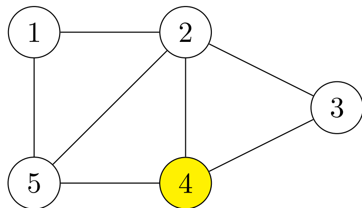


stack:

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation



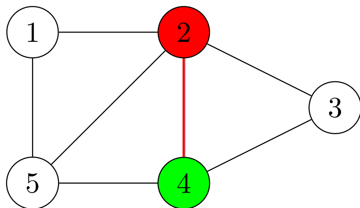
stack: 4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4



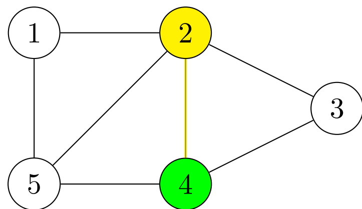
stack: 4

Traversing an undirected graph by the DFS algorithm

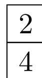
The DFS Algorithm

Animation

path: 4 2



stack:

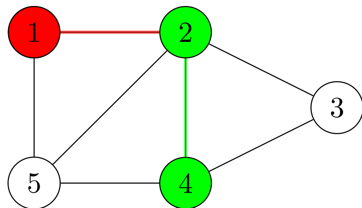


Traversing an undirected graph by the DFS algorithm

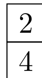
The DFS Algorithm

Animation

path: 4 2



stack:

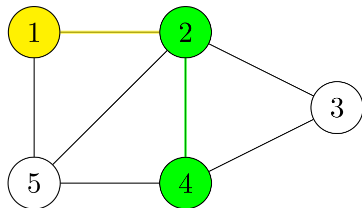


Traversing an undirected graph by the DFS algorithm

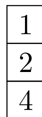
The DFS Algorithm

Animation

path: 4 2 1



stack:

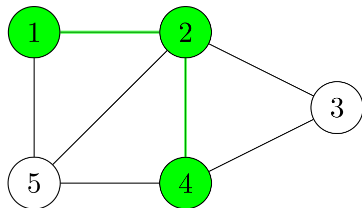


Traversing an undirected graph by the DFS algorithm

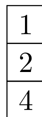
The DFS Algorithm

Animation

path: 4 2 1



stack:

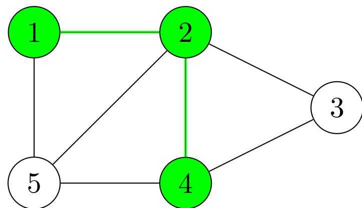


Traversing an undirected graph by the DFS algorithm

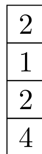
The DFS Algorithm

Animation

path: 4 2 1



stack:

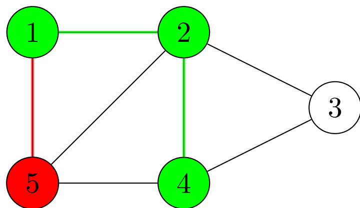


Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1



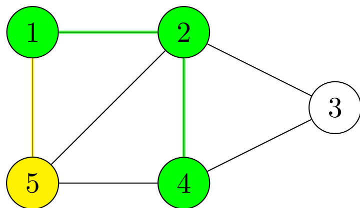
stack: 1 2 4

Traversing an undirected graph by the DFS algorithm

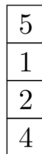
The DFS Algorithm

Animation

path: 4 2 1 5



stack:

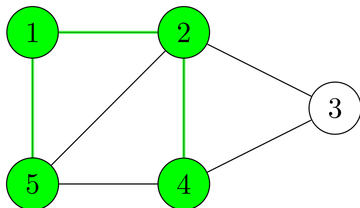


Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5



stack:

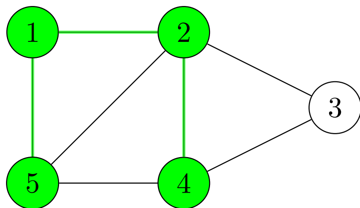
4
5
1
2
4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5



stack:

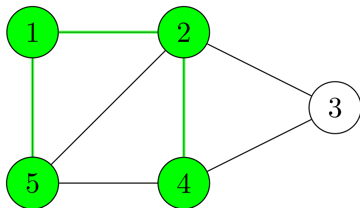
1
5
1
2
4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5



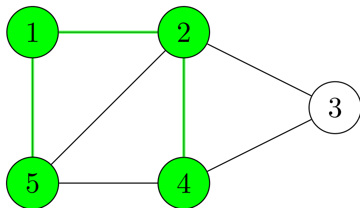
stack:
2
5
1
2
4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5



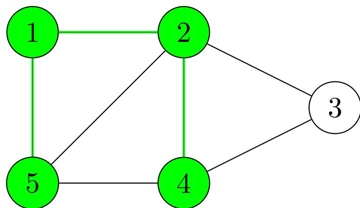
stack: 5 1 2 4

Traversing an undirected graph by the DFS algorithm

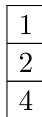
The DFS Algorithm

Animation

path: 4 2 1 5



stack:

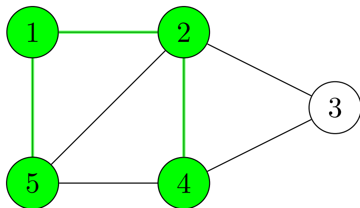


Traversing an undirected graph by the DFS algorithm

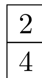
The DFS Algorithm

Animation

path: 4 2 1 5



stack:

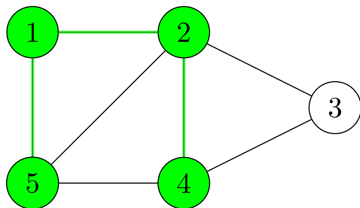


Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5



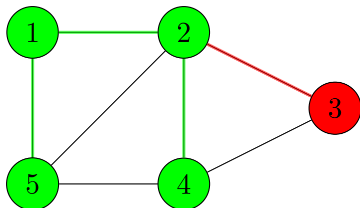
stack:

5
2
4

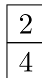
Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation



stack:

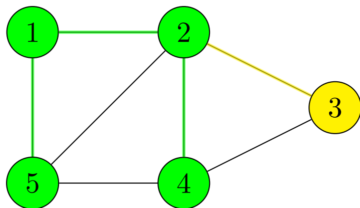


Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5 3



stack:

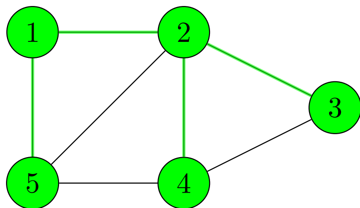
3
2
4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5 3



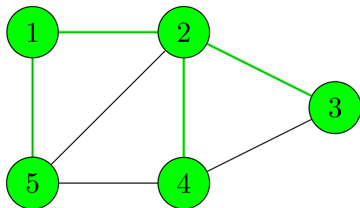
stack:

2
3
2
4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

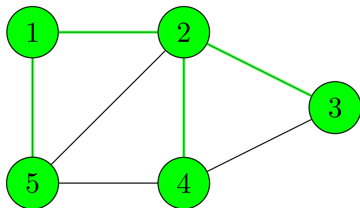
Animation



Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation



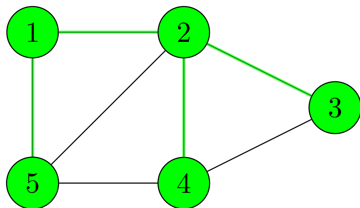
stack: 3 2 4

Traversing an undirected graph by the DFS algorithm

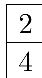
The DFS Algorithm

Animation

path: 4 2 1 5 3



stack:

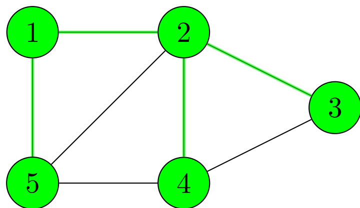


Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5 3



stack:

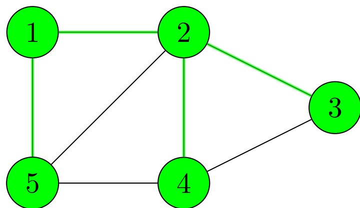
4
2
4

Traversing an undirected graph by the DFS algorithm

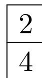
The DFS Algorithm

Animation

path: 4 2 1 5 3



stack:

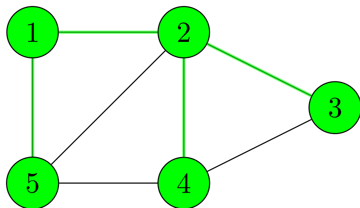


Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5 3



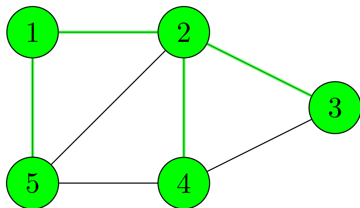
stack: 4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5 3



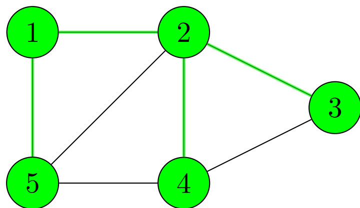
stack:

Traversing an undirected graph by the DFS algorithm

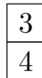
The DFS Algorithm

Animation

path: 4 2 1 5 3



stack:

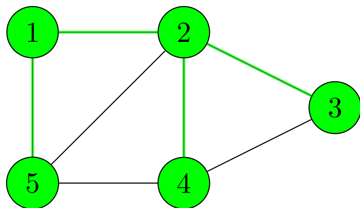


Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation

path: 4 2 1 5 3

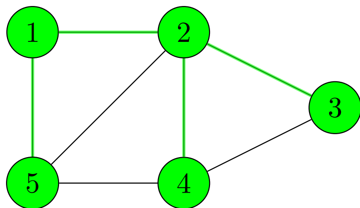


stack: 4

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm

Animation



stack:

Traversing an undirected graph by the DFS algorithm

The DFS Algorithm — An Implementation

Next slides show a modified version of the program presented on the previous lecture, which traverses an undirected graph using the DFS algorithm. The algorithm operates on an adjacency list which is the result of converting the adjacency matrix of the graph. Since the graph is connected, the DFS terminates after visiting all its vertices, but the program is ready to be used also for graphs which are not connected.

The DFS Algorithm — An Implementation

Adjacency Matrix and Base Type of Adjacency List

```
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;
```

The DFS Algorithm — An Implementation

Adjacency Matrix and Base Type of Adjacency List

The beginning of the program presented in the previous slide differs only by one detail from the version presented on the previous lecture. The base type of the adjacency list has one additional field of `bool` type which is used for marking the vertex as visited. In such a case the field's value is `true`, otherwise `false`. The value of the field has a meaning only in the list of all vertices (the "vertical" list, which is a part of the adjacency list).

The DFS Algorithm — An Implementation

The Queue Base Type and the Queue Pointers Structure

```
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;
```

The DFS Algorithm — An Implementation

The Queue Base Type and the Queue Pointers Structure

The previous slide contains definitions of the base type of a FIFO queue and a structure that stores the head and tail pointers of the queue. A global variable of the `fifo_pointers` type, named `path`, is also declared in the 8th line. Each element of the queue stores the number of a visited vertex. The queue as the whole serves for storing a path which is the result of the DFS and which shows the order in which the algorithm traversed the vertices of the graph, starting from an initial vertex.

The DFS Algorithm — An Implementation

The enqueue() Function

```
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,"No new element was created!\n");
16 }
```

The DFS Algorithm — An Implementation

The `enqueue()` Function

In the previous slide is shown a definition of the `enqueue()` function, which creates a FIFO queue and adds new elements to it. Since the function is defined almost in the same way as on previous lectures, it is not further discussed in details.

The DFS Algorithm — An Implementation

The dequeue() Function

```
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 }
```

The DFS Algorithm — An Implementation

The `dequeue()` Function

The `dequeue()` function presented in the previous slide differs from its counterpart, presented on the lecture about queues, in that it returns no value, but only removes a single element from the head of the FIFO queue.

The DFS Algorithm — An Implementation

The `remove_queue()` Function

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

The DFS Algorithm — An Implementation

The `remove_queue()` Function

The `remove_queue()` function calls in the `while` loop the `dequeue()` function to remove the FIFO queue. As an argument the former function takes an address of the queue pointers structure. It doesn't return any value. The `while` loop is repeated as long as the head pointer has a value different than `NULL`.

The DFS Algorithm — An Implementation

The print_path() Function

```
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  }
```

The DFS Algorithm — An Implementation

The `print_path()` Function

The `print_path()` function is a version of the `print_queue()` function modified to print the content of the FIFO queue which stores the path created by the DFS algorithm.

The DFS Algorithm — An Implementation

The create_vertical_list() Function

```
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 }
```

The DFS Algorithm — An Implementation

The `create_vertical_list()` Function

The `create_vertical_list()` function differs from its counterpart 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 — An Implementation

The convert_matrix_to_list() Function

```
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  }
```

The DFS Algorithm — An Implementation

The `convert_matrix_to_list()` Function

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

The DFS Algorithm — An Implementation

The print_adjacency_list() Function

```
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 }
```

The DFS Algorithm — An Implementation

The `print_adjacency_list()` Function

The `print_adjacency_list()` function is implemented in exactly the same way as its counterpart presented on the previous lecture.

The DFS Algorithm — An Implementation

The `remove_adjacency_list()` Function

```
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 }
```

The DFS Algorithm — An Implementation

The `remove_adjacency_list()` Function

Also the `remove_adjacency_list()` function is implemented in the same way as its counterpart from the previous lecture.

The DFS Algorithm — An Implementation

The `find_vertex()` Function

```
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  }
```

The DFS Algorithm — An Implementation

The `find_vertex()` Function

The `find_vertex()` is a helper function for the subroutine 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 starting node of the adjacency list and number of the vertex to find. The function iterates over the list of all vertices using the `while` loop and the `start_vertex` pointer (lines no. 4–6) and tests if the node currently pointed by the parameter stores the searched number. If not, it goes to the next element in the list, otherwise the loop terminates and the function returns the address of the found node that represents the searched vertex. The function is prepared for receiving by the parameters a `NULL` value or a number of a nonexistent vertex, although such a situation is very unlikely in the program. Should it happen, the `find_vertex()` function will return the `NULL` value.

The DFS Algorithm — An Implementation

The `has_not_been_visited()` Function

```
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 }
```

The DFS Algorithm — An Implementation

The `has_not_been_visited()` Function

The `has_not_been_visited()` function check if the vertex of a given number is unvisited. It takes two arguments — the address of the starting node of the adjacency list and the address of the node that represents the examined vertex in the list of adjacent vertices (one of the "horizontal" lists) of the currently visited vertex. The `has_not_been_visited()` function calls the `find_vertex()` function to locate the node representing the adjacent vertex in the list of all vertices (the "vertical" list). The latter function returns the address of the searched element, which is directly used in the 5th line of the described function for reading the value of the `visited` field of the examined vertex. The `has_not_been_visited()` function returns a negated value of the field.

The DFS Algorithm — An Implementation

The dfs() Function

```
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 }
```

The DFS Algorithm — An 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 starting node of the graph's adjacency list. The second one is an address of the initial vertex, from which the function begins traversing the graph. The last argument is an address of the FIFO queue pointers structure. In the queue the path generated by the function is stored. In the 4th line the function checks if the addresses of vertices passed by its parameters are different than `NULL`. If so, then the function adds to the FIFO queue a new element that stores the number of the vertex represented by a node pointed by the `vertex` parameter (5th line). Next, it marks the vertex as visited by assigning the `true` value to the `visited` field of the node (6th line). Then, inside the `while` loop, the function assigns to the `vertex` pointer an address stored in the `next` field of the node currently pointed by that pointer (8th line).

The DFS Algorithm — An Implementation

The `dfs()` Function

If the address is different than `NULL`, then it means that vertices adjacent to the currently visited vertex exist and the `vertex` pointer points to a node in the adjacent vertices list that represents the first of them. The function checks if the vertex exists and hasn't been yet visited in the lines no. 9 and 10. If both conditions are fulfilled then the function invokes itself recursively for the adjacent vertex. As the second argument it takes this time the result of the `find_vertex()` function, which returns the address of the node representing the adjacent 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 adjacent vertex exists, which is represented in the list of neighbours of the current vertex, then the `dfs()` function is again called recursively for the adjacent vertex.

The DFS Algorithm — An Implementation

The visit_all_vertexes() Function

```
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 }
```

The DFS Algorithm — An Implementation

The `visit_all_vertexes()` Function

The `visit_all_vertexes()` function is called after the `dfs()` function terminates. It checks if all the vertices of the graph have been visited and it calls the `dfs()` function for those of them that haven't. The function returns no value, but takes one argument which is the address of the starting node of the adjacency list. A local pointer named `vertex` is declared in the 2nd line of the function, which is initialised with the address stored in the `start_vertex` parameter. Although the parameter passes by the value and the function could use it for iterating the list of all vertices without consequences for the argument passed by the parameter, it is necessary to use another pointer for that purpose, because the address the `start_vertex` parameter stores is used many times in the function's body.

The DFS Algorithm — An Implementation

The `visit_all_vertexes()` Function

In the `while` loop the function iterates the list of all vertices of the graph (the "vertical" list) using the `vertex` pointer and checks if any of them is not yet visited (line no. 4). If so, then in the 7th line the `dfs()` function is invoked for the unvisited vertex. The address of the FIFO queue pointers structure, which is declared in the 5th line and initialised in the 6th line, is passed as the last argument of the `dfs()` function recursive call. In other words the `visit_all_vertexes()` function uses its own local FIFO queue. After the `dfs()` function terminates the content of the queue is displayed on the screen and the queue is removed, 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 terminates after each node in the list of all vertices of the graph has been checked, which means that the whole graph is traversed.

The DFS Algorithm — An Implementation

The main() Function

```
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("Please enter the number of the initial vertex:");
7          int vertex_number = 0;
8          scanf("%d",&vertex_number);
9          dfs(start_vertex,find_vertex(start_vertex,vertex_number), &path);
10         puts("The DFS traversing result:");
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 }
```

The DFS Algorithm — An Implementation

The `main()` Function

Comparing to the program presented in the previous lecture, the `main()` function has additional lines starting from no. 6 until no. 13. In the 6th line a message is displayed to the user asking her or him to enter the number of a graph's vertex from which the DFS algorithm should start traversing the graph. The number is read from the keyboard (8th line) and stored in a local variable named `vertex_number`. Next the `dfs()` function is called. The second argument of the function — the address of the node in the list of all vertices that represents the initial vertex — is returned by the `find_vertex()` function. The path generated by the `dfs()` function is displayed on the screen with an adequate message (lines no. 10 and 11), and then the program removes the queue where the path is stored (12th line). In the 13th line the `main()` function calls the `visit_all_vertexes()` function to visit all the not yet visited vertices of the graph.

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 current vertex the algorithm adds *all unvisited vertices adjacent to this vertex* to the queue. After marking the current vertex as visited the BFS algorithm removes the first discovered vertex from the head of the queue and visits it as the next. All adjacent vertices of a given vertex are visited first, hence the name of the algorithm: Breadth-First Search. It is usually implemented in an iterative form (using a loop). Its time-complexity is $O(V + E)$.

The BFS Algorithm

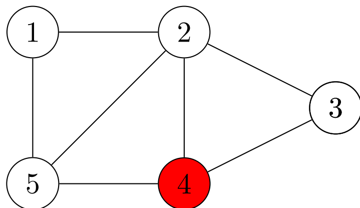
An Animation

The next slide shows an animation of the working of the BFS algorithm for an undirected graph — the same graph that has been used for illustrating the working of the DFS algorithm. In relation to the previous animation in the next one instead of the stack a FIFO queue is used, which is shown at the bottom of the slide. All other elements of the animation are the same as for the DFS animation.

The BFS Algorithm

Animation

path:

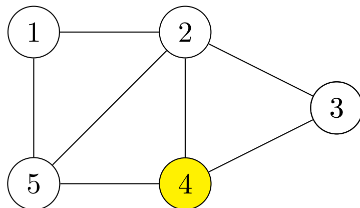


FIFO queue:

Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

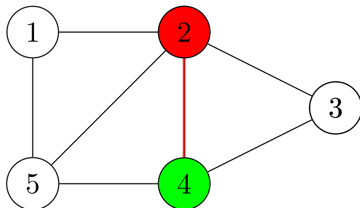
Animation



Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

Animation



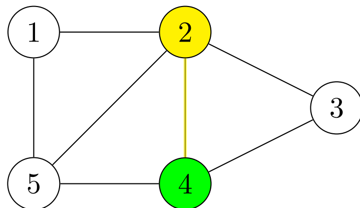
FIFO queue:

Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

Animation

path: 4

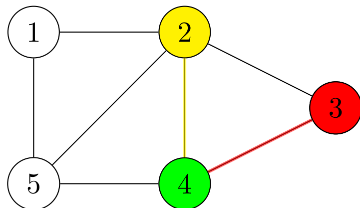


FIFO queue: 2

Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

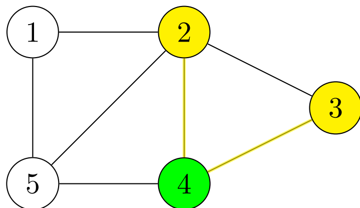
Animation



Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

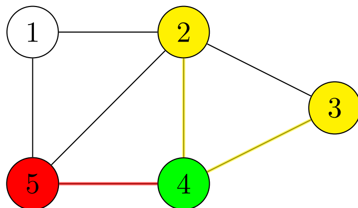
Animation



Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

Animation

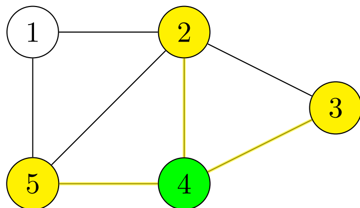


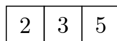
Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

Animation

path: 4



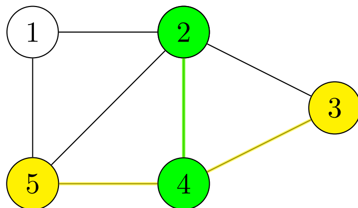
FIFO queue: 

Traversing an undirected graph by the BFS algorithm

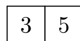
The BFS Algorithm

Animation

path: 4 2



FIFO queue:

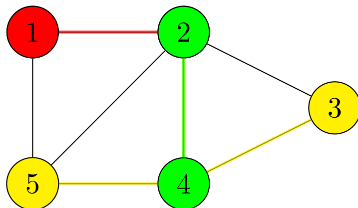


Traversing an undirected graph by the BFS algorithm

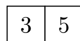
The BFS Algorithm

Animation

path: 4 2



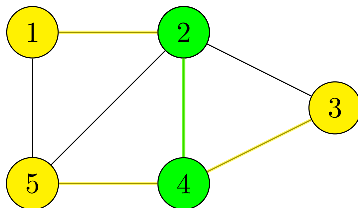
FIFO queue:



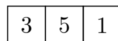
Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

Animation



FIFO queue:

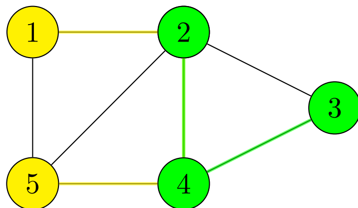


Traversing an undirected graph by the BFS algorithm

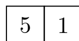
The BFS Algorithm

Animation

path: 4 2 3



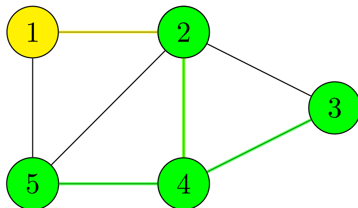
FIFO queue:



Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

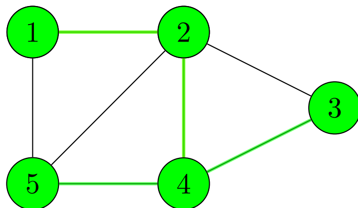
Animation



Traversing an undirected graph by the BFS algorithm

The BFS Algorithm

Animation



FIFO queue:

Traversing an undirected graph by the BFS algorithm

The BFS Algorithm — An Implementation

The next slides present a version of the program demonstrated earlier, which uses the BFS algorithm instead of DFS for traversing a graph. Since the source code of both programs is very similar, only those elements that have changed are described.

The BFS Algorithm — An Implementation

Adjacency Matrix and Base Type of Adjacency List

```
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;
```

The BFS Algorithm — An Implementation

The Queue Base Type and the Queue Pointers Structure

```
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;
```

The BFS Algorithm — An Implementation

The Queue Base Type and the Queue Pointers Structure

Please note, that in relation to the previous program, an additional variable of the name `discovered_fifo` is created. It is a structure of pointers for the queue of vertices that are discovered and visited in the next iteration of the algorithm.

The BFS Algorithm — An Implementation

The enqueue() Function

```
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,"No new element was created!\n");
16 }
```

The BFS Algorithm — An Implementation

The `dequeue()` Function

```
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 }
```

The BFS Algorithm — An Implementation

The `dequeue()` Function

The `dequeue()` function, unlike its counterpart 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 — An Implementation

The `remove_queue()` Function

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

The BFS Algorithm — An Implementation

The print_path() Function

```
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  }
```

The BFS Algorithm — An Implementation

The create_vertical_list() Function

```
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 }
```

The BFS Algorithm — An Implementation

The convert_matrix_to_list() Function

```
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 }
```

The BFS Algorithm — An Implementation

The `print_adjacency_list()` Function

```
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 }
```

The BFS Algorithm — An Implementation

The remove_adjacency_list() Function

```
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 }
```

The BFS Algorithm — An Implementation

The find_vertex() Function

```
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  }
```

The BFS Algorithm — An Implementation

The `has_not_been_visited()` Function

```
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  }
```

The BFS Algorithm — An Implementation

The `bfs()` Function

```
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   }
```

The BFS Algorithm — An Implementation

The `bfs()` Function

The `bfs()` function implements the traversing of the graph with the use of the BFS algorithm. It returns no value but takes four arguments — an address of the starting node of the adjacency list, an address of the initial vertex from which it starts traversing the graph, an address of the pointers structure of the queue where the path generated by the algorithm is stored and an address of a pointers structure of the queue for storing the discovered vertices. The function checks in the 4th line if the addresses passed to it by the parameters are different than `NULL`. If so, that it stores in the `discovered_fifo` queue the number of the vertex stored in the node pointed by the `vertex` parameter (5th line) and starts the outer `while` loop, which is repeated as long as the queue of discovered vertices is not empty (6th line).

The BFS Algorithm — An Implementation

The `bfs()` Function

Inside this loop the first element of the `discovered_fifo` queue is removed (7th line). The number stored in it and assigned to the `vertex_number` variable is then used by the `find_vertex()` function to find an address of the node in the list of all vertices of the graph, that represents this vertex. The address is assigned to the `vertex` pointer. In the 9th line the function checks if the vertex is still unvisited. If so, then to the local pointer named `next_vertex` is assigned the address stored in the `next` field of the node that represents the vertex. If the `next_vertex` pointer has a value different than `NULL` then it means, that there is an nonempty list of vertices adjacent to the vertex. In the inner `while` loop (lines no. 11–14) the `bfs()` function iterates over the list and stores the numbers of the adjacent vertices in the `discovered_fifo` queue. After the inner loop terminates, but still in the outer loop, the `bfs()` function marks the current vertex as visited and stores its number in the `path_fifo` queue.

The BFS Algorithm — An Implementation

The `bfs()` Function

The `bfs()` function terminates when all vertices reachable from the initial vertex are visited. The path generated by the function is stored in the `path_fifo` queue.

The BFS Algorithm — An Implementation

The visit_all_vertexes() Function

```
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 }
```

The BFS Algorithm — An Implementation

The `visit_all_vertexes()` Function

The `visit_all_vertexes()` function differs from its counterpart from the previous program in that it invokes the `bfs()` function instead of the `dfs()` function (9th line) and it uses two FIFO queues. The pointers structure of the second queue is declared in the 7th line and initialised in the 8th and 9th lines. The queue is passed to the `bfs()` function and used for storing the discovered vertices. Then it is removed in the 13th line.

The BFS Algorithm — An Implementation

The main() Function

```
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("Please enter the number of the initial vertex:");
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("The BFS traversing result:");
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  }
```

The BFS Algorithm — An Implementation

The `main()` Function

The `main()` function underwent two modifications in relation to its counterpart from the earlier program. Instead of the `dfs()` function it calls the `bfs()` function and passed to it, as the forth argument, the address of the `discovered_fifo` queue. Moreover, an invocation of the `remove_queue()` function for this queue is added to the `main()` function source code, to remove it from the computer memory (14th line).

Summary

The BFS and DFS algorithms can be applied either to the connected or unconnected undirected graphs or strongly connected or unconnected directed graphs. As it has been mentioned at the beginning of the lecture, the algorithms are basis for many other graph algorithms, also other graph traversing algorithms, like the *Best–First Search* or A^* algorithm. Initially the BFS and DFS algorithms were applied for the Artificial Intelligence problems, but with time they become used in many other applications.

Questions

?

The End

Thank You For Your Attention!