Fundamentals of Programming 2

Graphs, DFS And BFS Algorithms

Arkadiusz Chrobot

Department of Information Systems

May 20, 2024

Outline

Introduction

Graphs are data structures that are used for representing relationships between data items. Although the basic idea behind graphs is quite simple, they are applied for solving many problems in Computer Science. These data structures are based on mathematical concepts originally discovered by the Swiss mathematician Leonhard Euler, and developed by others. Some of these ideas are discussed in the following slides. Unfortunately, the terminology in graph theory is not standardized, so some textbooks may have different definitions of concepts presented here.

Graph Definitions

Graph Definitions

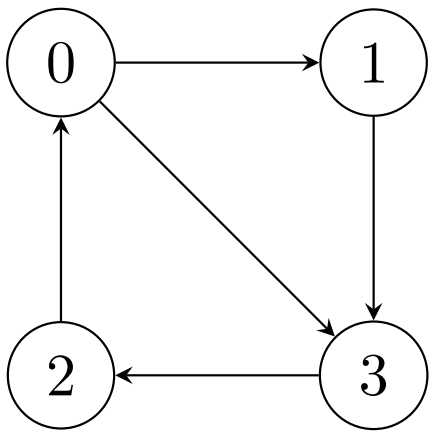
Directed Graph Definition

A **directed graph** or a **digraph** G is defined as a pair (V, E), where V is a finite set, whose elements are called vertices or nodes of the graph G, E is a binary relation on V and $E \subseteq V \times V$. The set V is called the set of vertices. The set E is the set of edges of the graph G. Its elements are called edges.

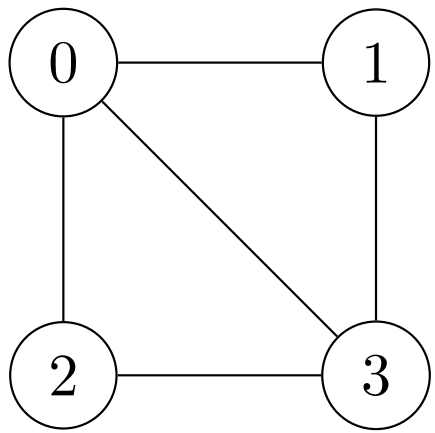
Undirected Graph Definition

An **undirected graph** is a graph whose E set consists of unordered pairs of vertices. It means that an edge is a set $\{u, v\}$ where $u, v \in V$ and $u \neq v$. The edge is denoted as (u, v). The pairs (u, v) and (v, u) specify the same edge. There are no loops (edges that join a vertex to itself) in the undirected graphs.

Graph Examples



(a) A directed graph



(b) An undirected graph

Examples of graphs

Edges And Neighbours

Edge Types

In the directed graph $G = (V, E)$ the edge (u, v) is an **outgoing** edge for the vertex u and an **incoming** edge for the vertex v. In the undirected graph the edge (u, v) is called **incident** to vertices u and v. It **joins** u and v.

Neighbours

A vertex v is an **adjacent vertex** to the vertex u (it is a **neighbour** of the vertex u) in a graph $G = (V, E)$ if these vertices are connected by an edge (v, u). In a directed graph the *adjacency relation* doesn't have to be symmetric.

Vertex Degree

The **degree of a vertex** in an undirected graph is the number of edges incident to the vertex. In a directed graph the **out-degree** of a vertex is the number of its outgoing edges and the **in-degree** of a vertex is the number of its incoming edges. In a directed graph the **degree** of a vertex is a sum of its in-degree and out-degree.

Path And Cycle

Path Definition

A **path (route) of the length k** from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The path length is the number of the edges in the path. The path contains vertices $v_0, v_1, v_2, \ldots, v_k$ and edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. If there is a path from a vertex u to a vertex u', then the u' vertex is reachable from the u vertex via the path p. A path is called a **simple path** if all vertices in the path are different.

Cycle Definition

A path $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ forms a **cycle** if $v_0 = v_k$. A cycle is a **simple cycle** if all of its vertices are different. A **loop** in a directed graph is a cycle of the length 1. A digraph that has no loops or parallel edges (appearing more then once) is called a **simple graph**. A graph that has no cycles is called an **acyclic graph**.

Connectivity

An undirected graph is **connected** if there is a path between any two vertices of the graph. A digraph is **strongly connected** if any two vertices in the graph are reachable from each other.

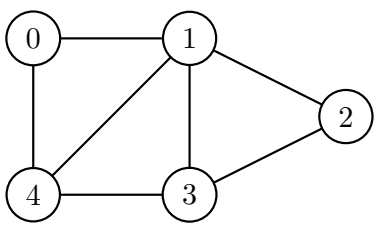
Dense And Sparse Graphs

An *undirected* graph is a **dense graph** if every pair of its vertices is connected by an edge. The number of edges in such a graph is $\binom{n}{2}$, where n is the number of vertices in the graph. A graph that has only a small fraction of the number of edges, compared to a dense graph, is called a **sparse graph**.

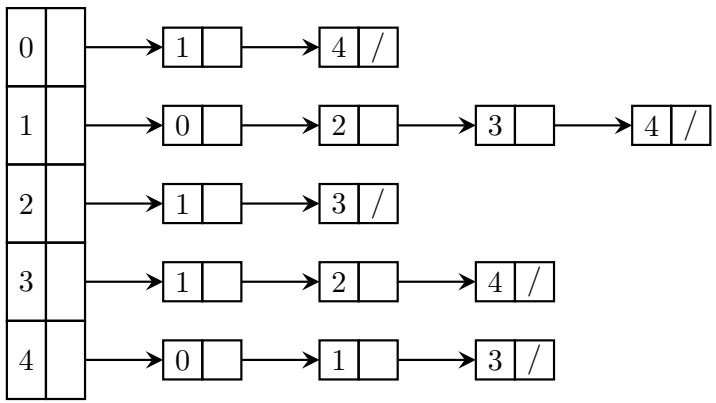
Graphs as Data Structures

There are two basic ways of representing graphs in software: the adjacency matrix and the adjacency list. The adjacency list can be implemented as a list of lists or as an array of pointers to lists (an array of lists, for short). The adjacency matrix is a statically or dynamically allocated two-dimensional array. The rows and columns in such a matrix represent the vertices of a graph. If two vertices are connected by an edge, then the adjacency matrix item located at the intersection of the column and the row associated with those vertices is set to `1`, otherwise it is set to `0`. The next slides present directed and undirected graphs and adjacency matrices and adjacency lists that represent them.

Representations of Undirected Graph



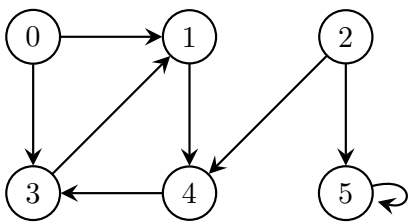
$$
\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{array}{c}
\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\
\left[\begin{array}{ccccc}
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0
\end{array}\right]
\end{array}
$$



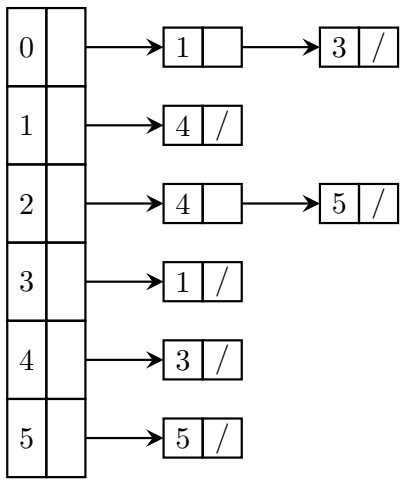
Representations of Undirected Graph

On the left side of the previous slide is an undirected graph. It is followed by its adjacency matrix and then by its adjacency list in a form of an array of pointers to lists. The characters / inside nodes of the adjacency list represents the NULL value. Please observe, that the adjacency matrix is symmetrical along its main diagonal. Thus, $\mathbb{A} = \mathbb{A}^T$, where $\mathbb{A}$ is the adjacency matrix. It means that, some space in the RAM can be saved by storing only items of either the upper or the lower triangular matrix.

Representations of Directed Graph



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



Representations of Directed Graph

Similarly, as in the case of the undirected graph, in the previous slide are shown (respectively, from left to right): a directed graph, its adjacency matrix and its adjacency list. The adjacency matrix is still a square matrix, but it isn't symmetrical[1]. Also, please note, that the graph has an edge that is a loop. In the adjacency matrix the loop is represented by the item located at the intersection of the sixth row and the sixth column, whose value is `1`.

[1] For some directed graphs it may be, but usually it isn't.

Representations of Graphs

Summary

Statistically, the adjacency list is the most frequently used representation of graphs in Computer Science. It's implemented either as a list of lists or as an array of lists. Only the second implementation is discussed here. Each element of such an array represents one of the graph's vertices and points to the list of its neighbours i.e. adjacent vertices. The order of neighbours in the list has no meaning. The number of vertices in all these lists for a directed graph is $|E|$ and for an undirected graph is $2 \cdot |E|$, where $|E|$ is the cardinality of the set of edges. Thus, the space complexity of the adjacency list is $O(|V| + |E|)$, while the space complexity of the adjacency matrix is $\Theta(|V|^2)$. Both representations can be used for describing either weighted or unweighted graphs. In the latter case the space required for storing the matrix can be saved by using a bitwise matrix, which stores the values of its items in single bits. However, the operations on such a matrix are more time-consuming than on a regular matrix.

Representations of Graphs

Summary

The adjacency matrices are more suitable for problems where the existence of an edge had to be verified, or an edge has to be added or deleted in a graph with a fixed number of vertices. On the other hand the adjacency lists are more useful for traversing the graph (most of the graph algorithms perform such an operation) or finding the degree of vertices. Also, they are better than the adjacency matrices in representing small or sparse graphs. Adjacency matrices are a better choice for representing dense graphs. Both representations are interchangeable, i.e. the adjacency matrix can be converted into the adjacency list and the other way.

Depth-First Search Algorithm

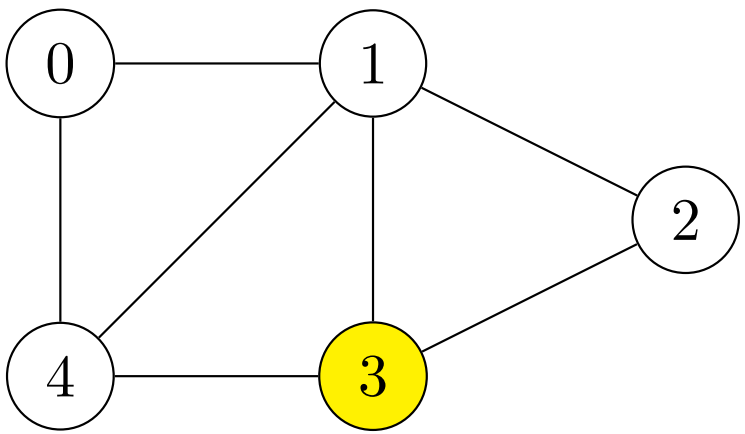
There are two basic graph traversal algorithms, the Depth-First Search (DFS) and Breadth-First Search (BFS). The DFS begins traversing a graph with a given starting vertex. It discovers the vertex's *first unvisited neighbour*, marks the current vertex as *visited* and explores its neighbour. The DFS repeats these steps until it finds a vertex that either has *no neighbours* or *all its neighbours have already been visited*. In that case the DFS *backtracks* to the previous vertex and checks if there are some still unvisited neighbours left. If so, it explores the first of them, otherwise it backtracks further. Eventually, the DFS will visit that way all vertices in the graph. The outcome of this algorithm is a sequence of visited vertices.

The DFS is a generalisation of tree pre-order traversal algorithm for all kind of graphs. Also, it is closely related to the backtracking algorithms. The Depth-First Search uses a stack, thus it can be implemented with the use of recursion. The next slide shows an animation of DFS traversing an example undirected graph.

Depth-First Search Algorithm

Animation

outcome:



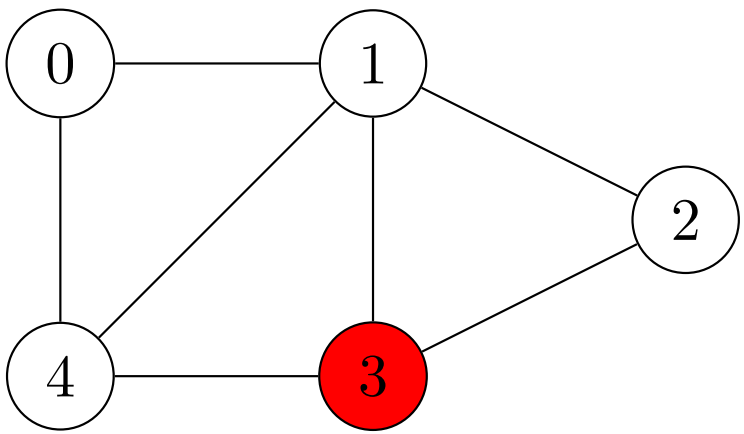
stack:

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

Depth-First Search Algorithm

Animation

outcome: 3



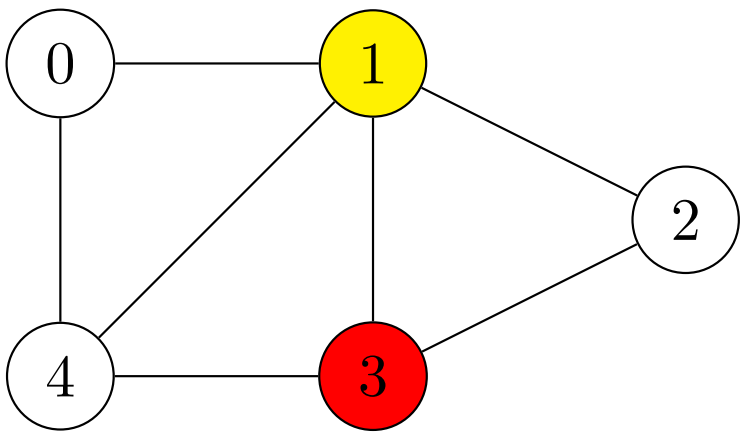
stack: 3

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

Depth-First Search Algorithm

Animation

outcome: 3



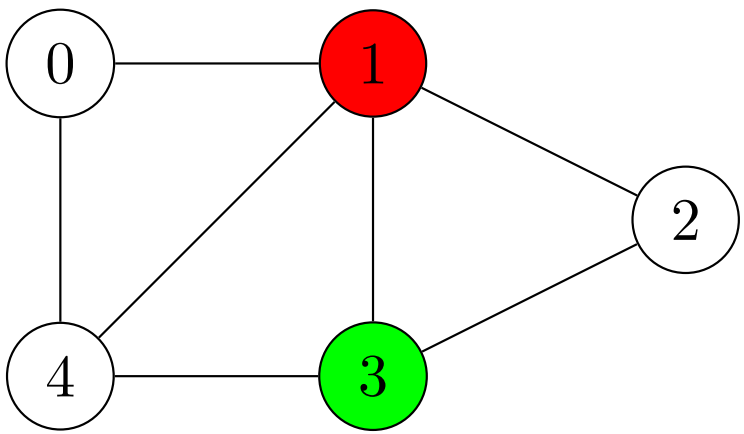
stack: 3

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

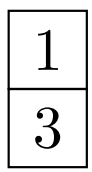
Depth-First Search Algorithm

Animation

outcome: 3 1



stack:

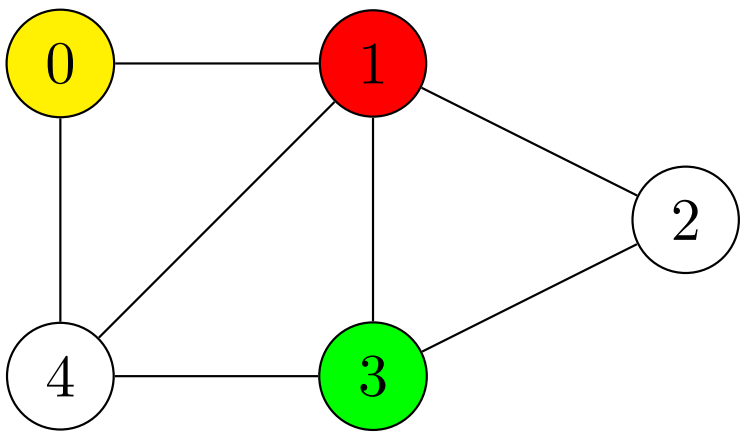


Traversing an undirected graph with the use of the DFS algorithm.

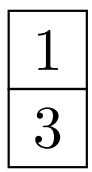
Depth-First Search Algorithm

Animation

outcome: 3 1



stack:

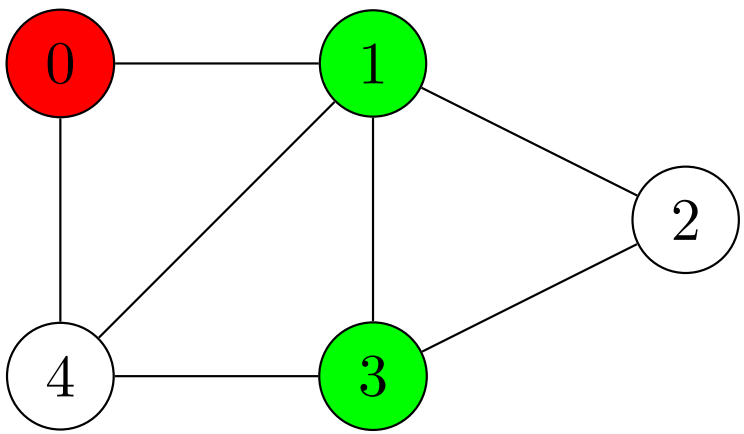


Traversing an undirected graph with the use of the DFS algorithm.

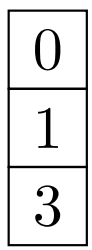
Depth-First Search Algorithm

Animation

outcome: 3 1 0



stack:

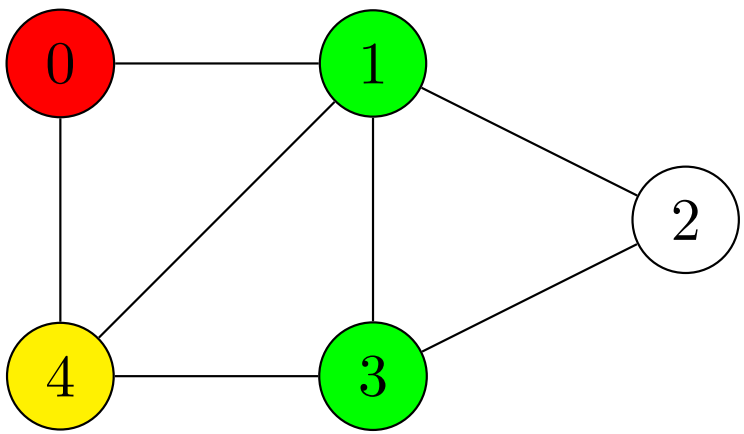


Traversing an undirected graph with the use of the DFS algorithm.

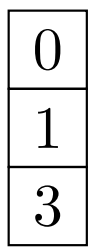
Depth-First Search Algorithm

Animation

outcome: 3 1 0



stack:

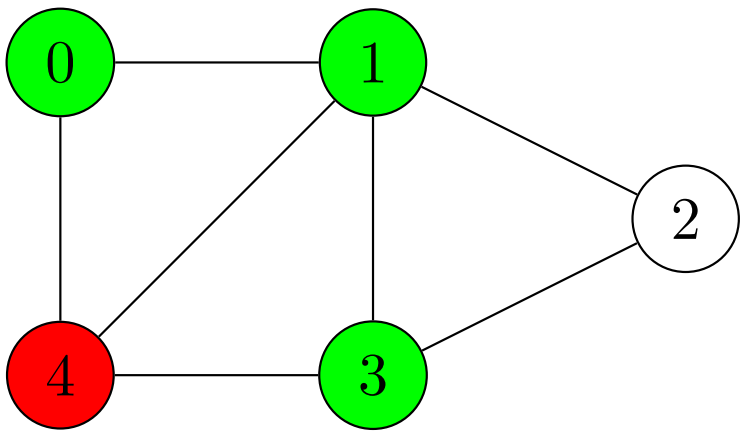


Traversing an undirected graph with the use of the DFS algorithm.

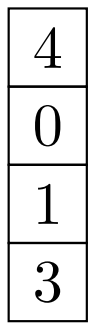
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4



stack:

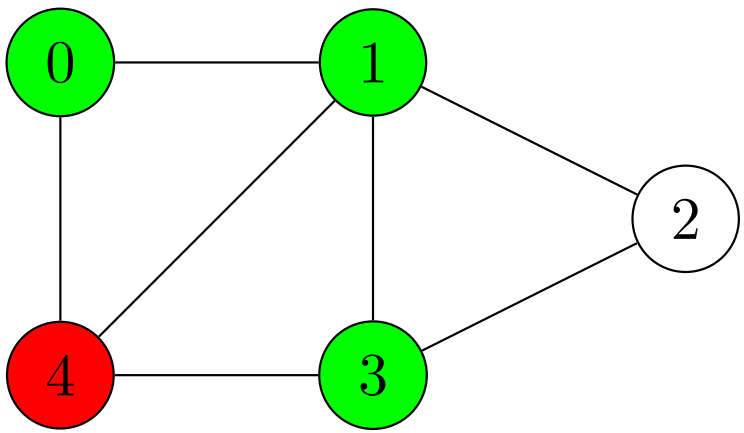


Traversing an undirected graph with the use of the DFS algorithm.

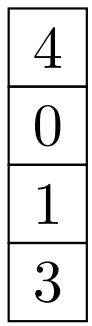
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4



stack:

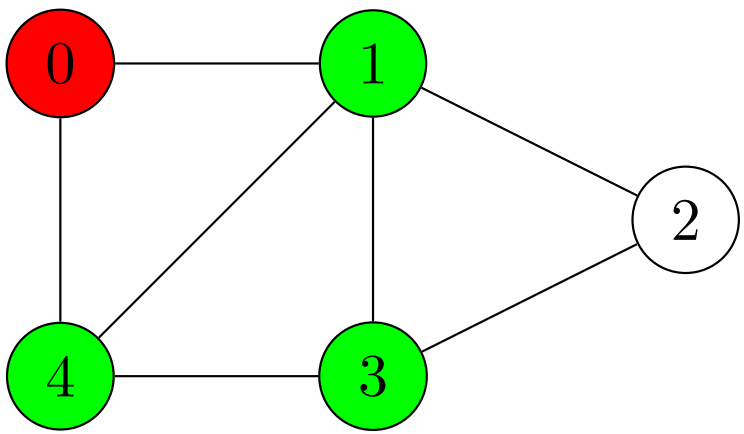


Traversing an undirected graph with the use of the DFS algorithm.

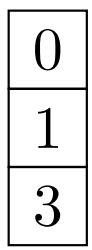
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4



stack:

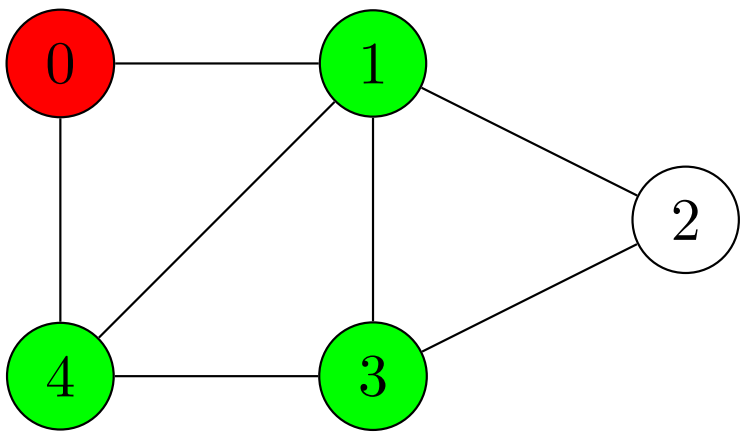


Traversing an undirected graph with the use of the DFS algorithm.

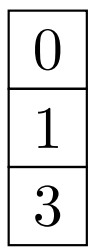
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4



stack:

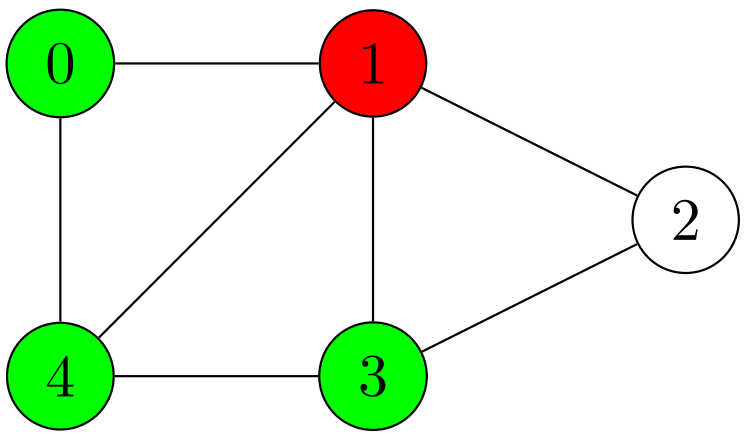


Traversing an undirected graph with the use of the DFS algorithm.

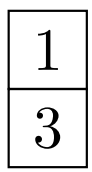
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4



stack:

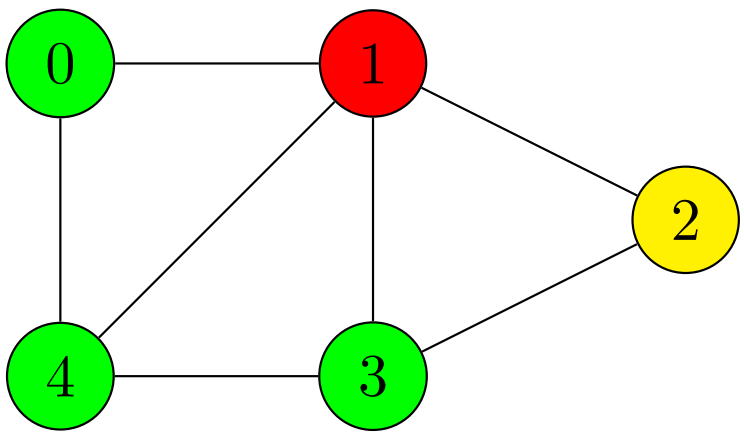


Traversing an undirected graph with the use of the DFS algorithm.

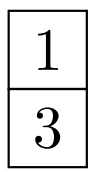
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4



stack:

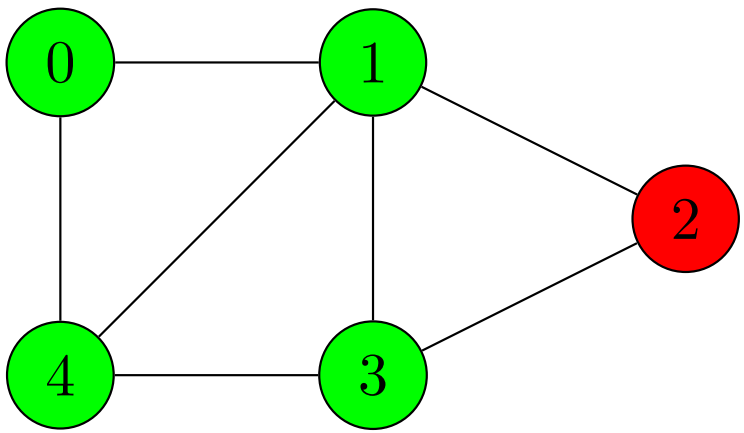


Traversing an undirected graph with the use of the DFS algorithm.

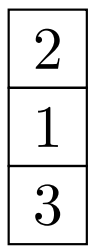
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4 2



stack:

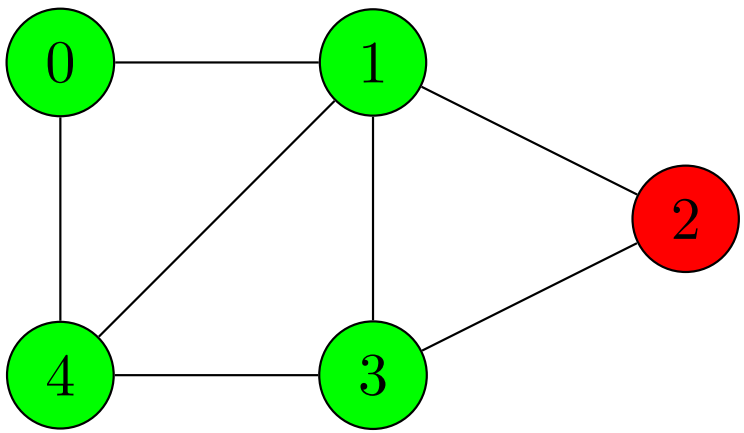


Traversing an undirected graph with the use of the DFS algorithm.

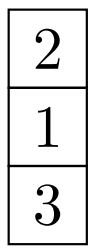
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4 2



stack:

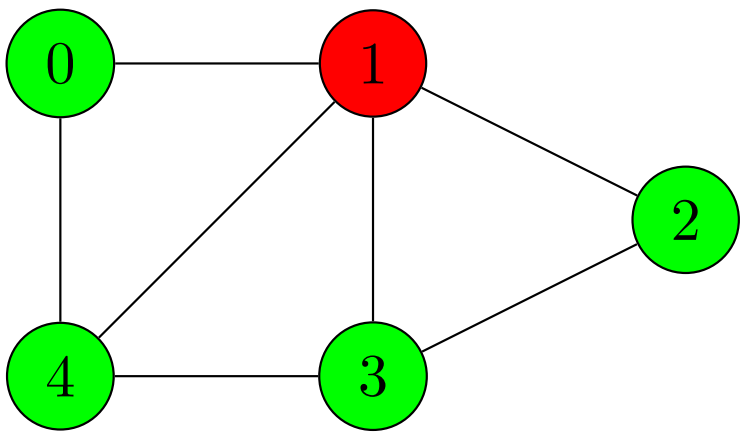


Traversing an undirected graph with the use of the DFS algorithm.

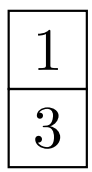
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4 2



stack:

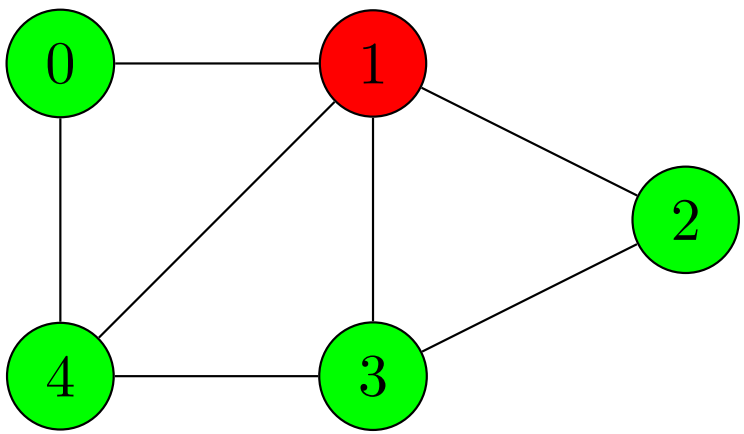


Traversing an undirected graph with the use of the DFS algorithm.

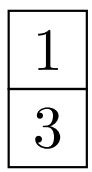
Depth-First Search Algorithm

Animation

outcome: 3 1 0 4 2



stack:

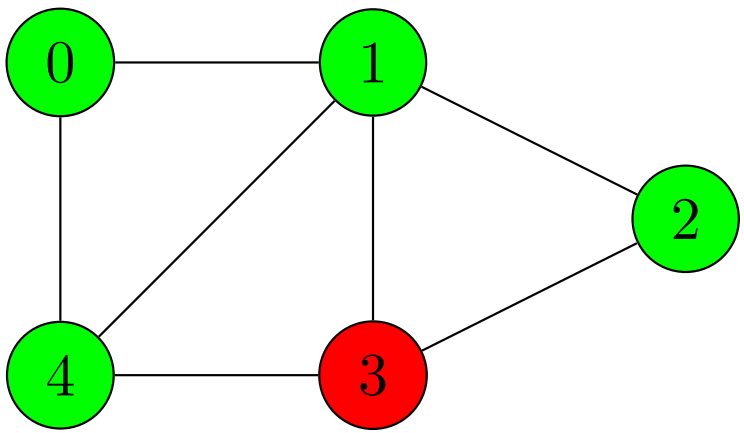


Traversing an undirected graph with the use of the DFS algorithm.

Depth-First Search Algorithm

Animation

outcome: 3 1 0 4 2



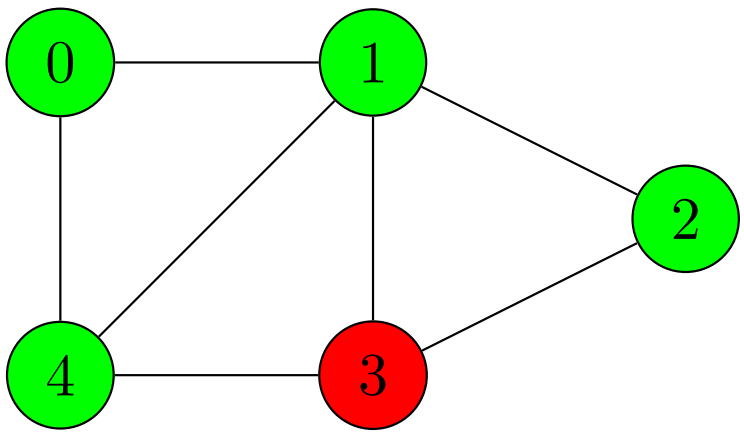
stack: 3

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

Depth-First Search Algorithm

Animation

outcome: 3 1 0 4 2



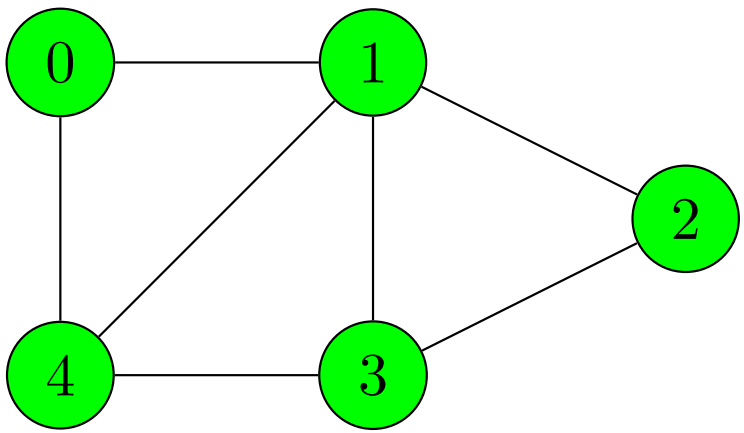
stack: 3

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

Depth-First Search Algorithm

Animation

outcome: 3 1 0 4 2



stack:

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

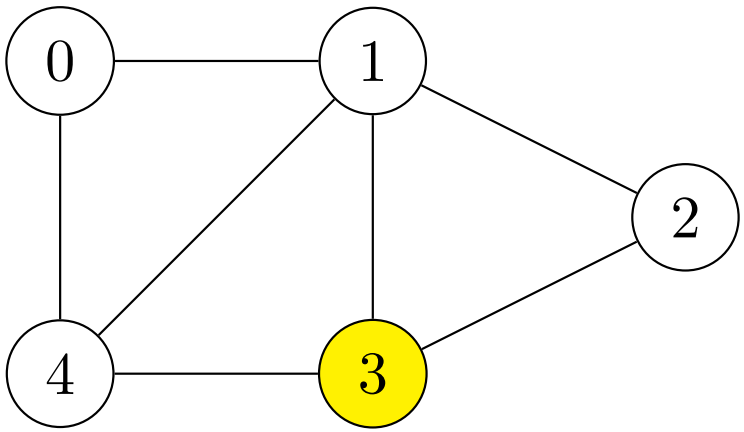
Breadth-First Search Algorithm

The BFS algorithm is similar to the DFS, but when visiting a vertex it discovers *all its unvisited neighbours* and then explores them in sequence, marking them as visited, and discovering their neighbours. It never backtracks. Instead of a stack it uses a FIFO queue. The next slide shows an animation of the BFS traversing the same undirected graph, as the DFS did.

Breadth-First Search Algorithm

Animation

outcome:



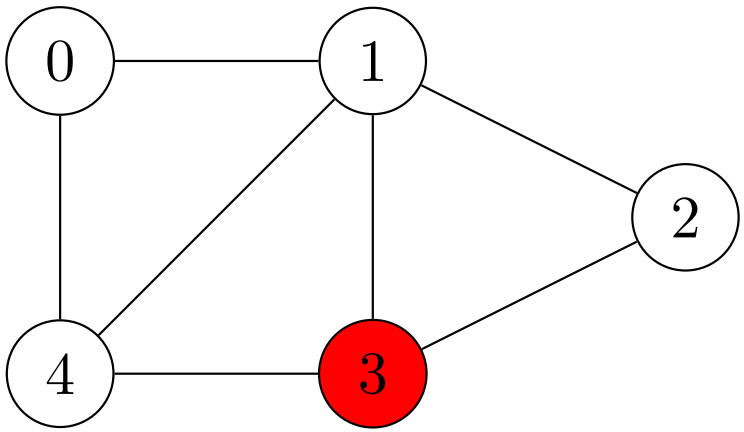
FIFO queue: 3

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

Breadth-First Search Algorithm

Animation

outcome:



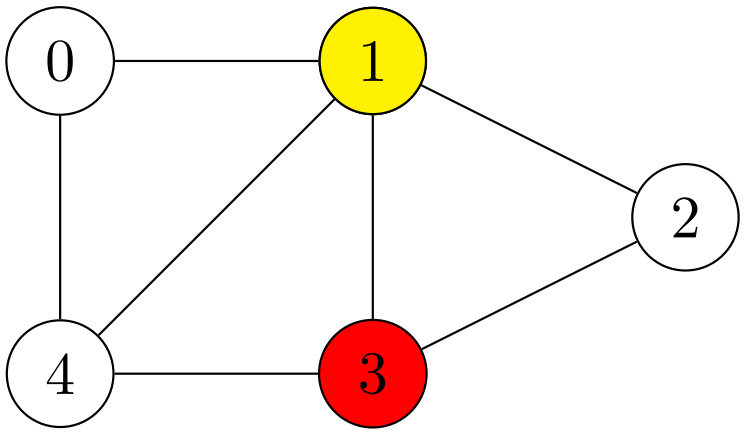
FIFO queue: 3

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

Breadth-First Search Algorithm

Animation

outcome:



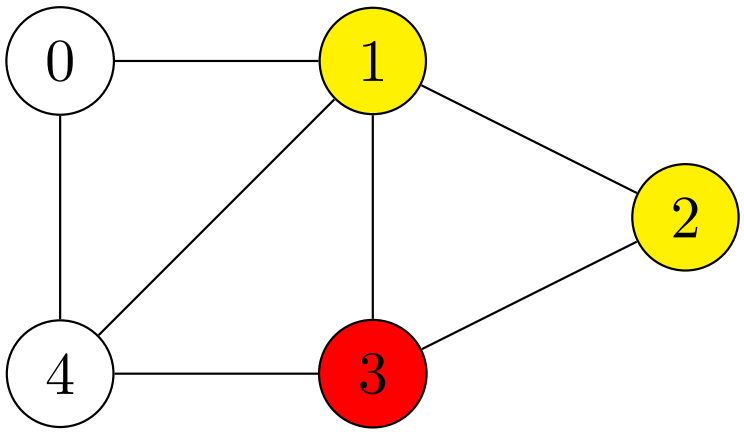
FIFO queue: 1

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

Breadth-First Search Algorithm

Animation

outcome:



FIFO queue:

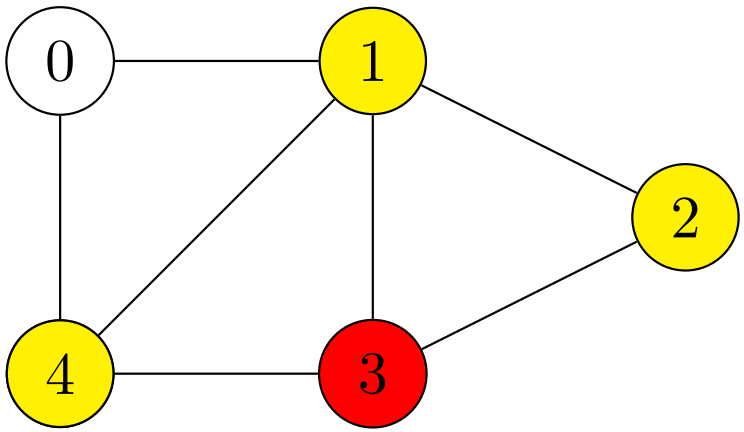
1	2

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

Breadth-First Search Algorithm

Animation

outcome:



FIFO queue:

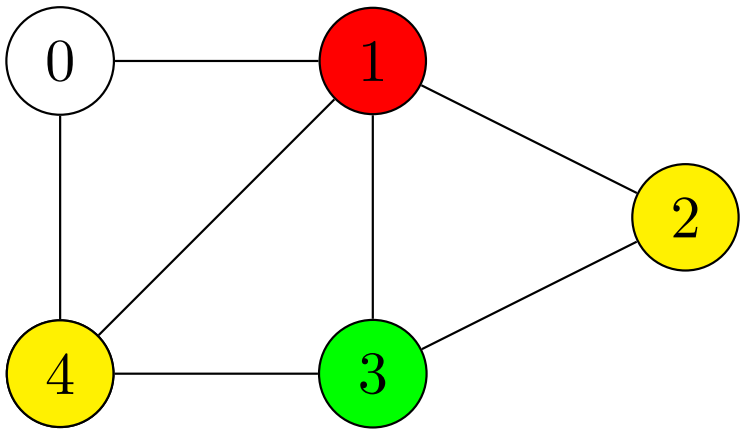
1	2	4

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

Breadth-First Search Algorithm

Animation

outcome: 3



FIFO queue:

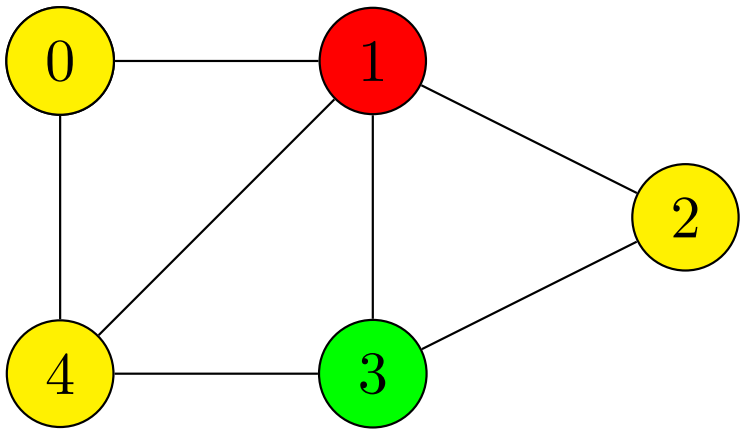
1	2	4

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

Breadth-First Search Algorithm

Animation

outcome: 3



FIFO queue:

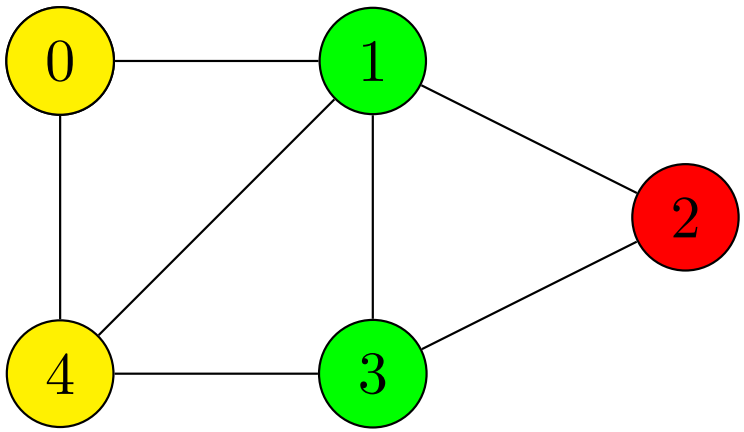
2	4	0

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

Breadth-First Search Algorithm

Animation

outcome: 3 1



FIFO queue:

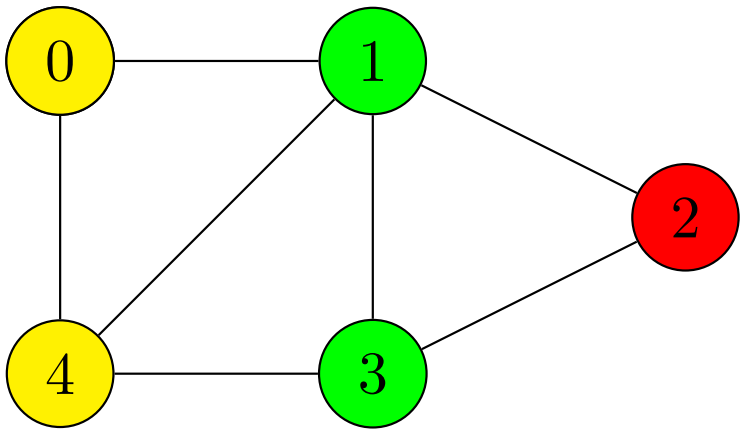
2	4	0

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

Breadth-First Search Algorithm

Animation

outcome: 3 1



FIFO queue:

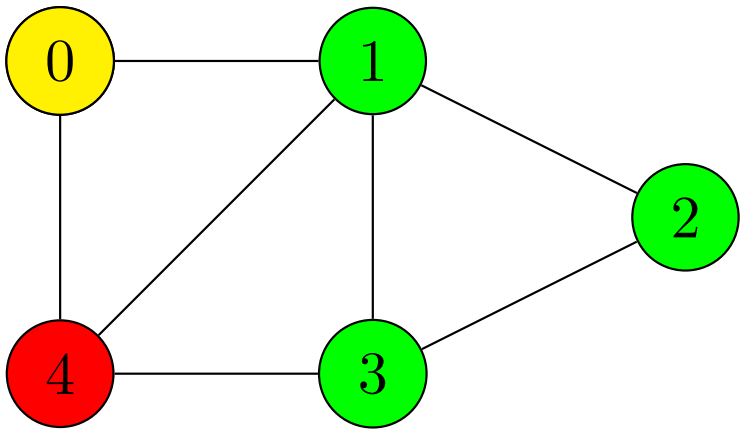
4	0

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

Breadth-First Search Algorithm

Animation

outcome: 3 1 2



FIFO queue:

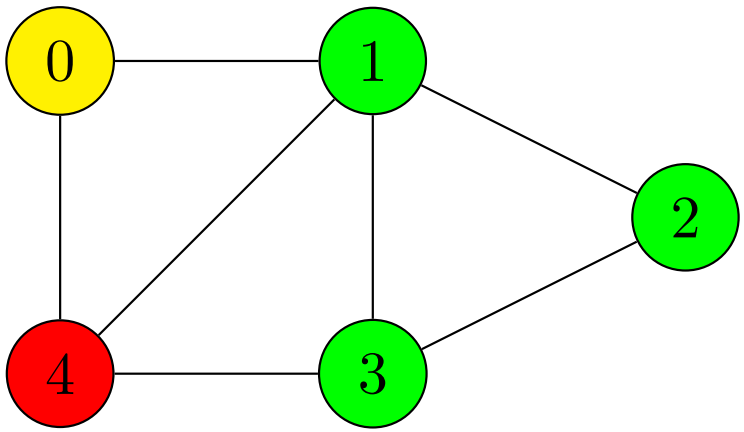
4	0

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

Breadth-First Search Algorithm

Animation

outcome: 3 1 2



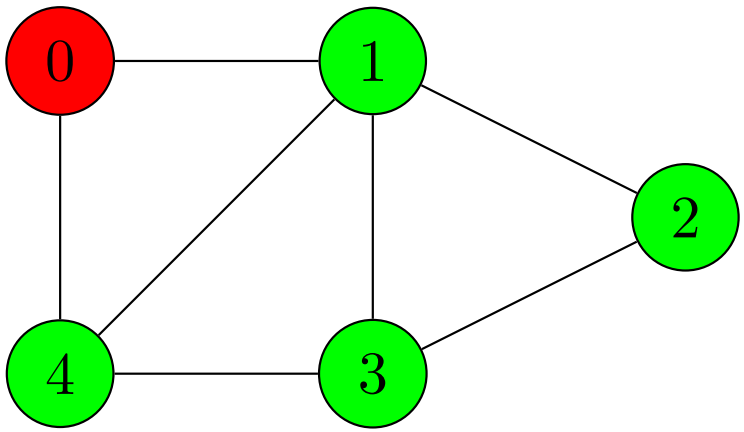
FIFO queue: 0

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

Breadth-First Search Algorithm

Animation

outcome: 3 1 2 4



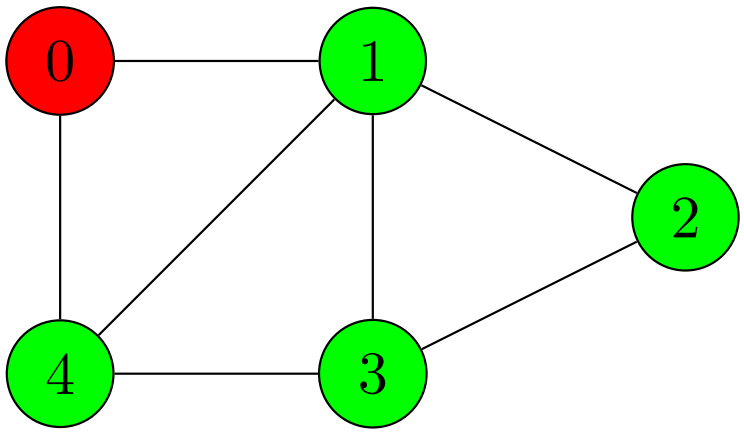
FIFO queue: 0

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

Breadth-First Search Algorithm

Animation

outcome: 3 1 2 4



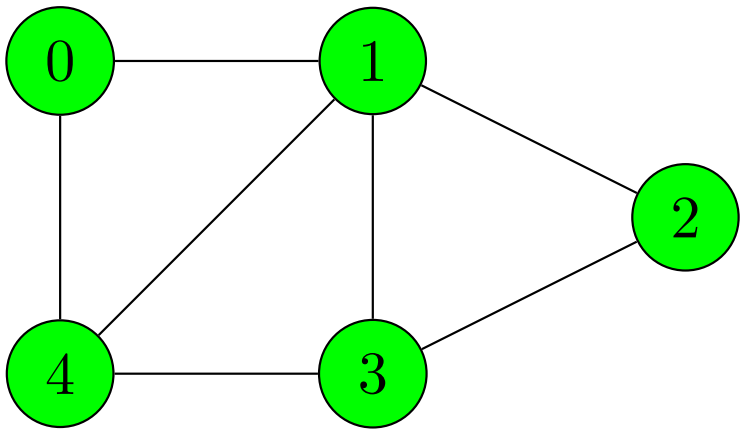
FIFO queue:

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

Breadth-First Search Algorithm

Animation

outcome: 3 1 2 4 0



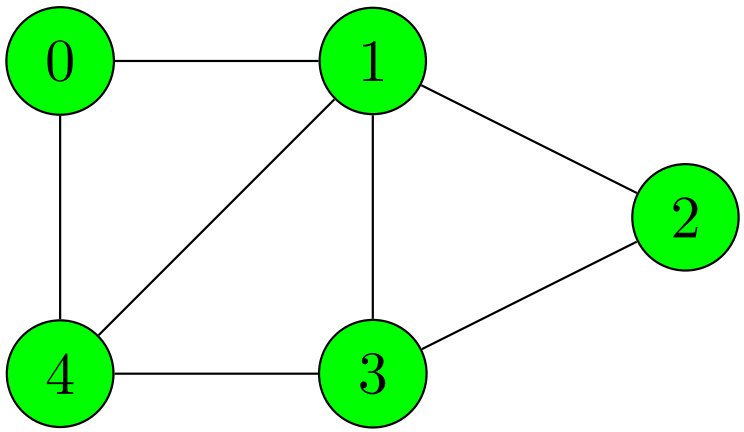
FIFO queue:

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

Breadth-First Search Algorithm

Animation

outcome: 3 1 2 4 0



FIFO queue:

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

Remarks

Please note, that both the DFS and the BFS algorithms can be applied either to the undirected or directed graphs. They will explore all vertices of a graph, if the graph is connected or strongly connected. For a disconnected graph they will visit only these vertices that are reachable from the starting one. In other words, they will explore only the *connected* or *strongly connected component* of this graph. To visit all vertices of the graph the algorithms need to check if any vertex of the graph has been left unvisited after their last run and explore this vertex.

The objective of the DFS and the BFS doesn't have to be visiting all vertices. They may be used to find a path from a starting vertex to a *goal vertex* or to a vertex that satisfies a *goal condition*.

The FIFO Queue Implementation

The FIFO queue, which is required by the BFS algorithm, is implemented in a static library, that consist of a header file (`queue.h`) and a source code file `queue.c`. The content of the former is shown in the next slide.

The queue.h File

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

The `queue.h` File

The header file starts with the `#ifndef` preprocessor directive (line no. 1), which is a part of the *header guard* (lines 1, 2 and 16). When interpreted, the directive causes the preprocessor to check, if the `GRAPH_QUEUE_H` marker has **not** already been defined in the program's source code. If so, then the preprocessor will insert into the program lines 2–15 of the header file. Please notice, that directive in the line no. 2 defines the aforementioned marker. It means that if the header file is included multiple times (with the use of the `#include` directive), then the preprocessor will add its content to the program only once.

In the header file are defined the types of queue node (lines 3–7) and of queue pointers structure (lines 9–12). Each node of the queue stores a vertex number (line no. 5). There are also defined prototypes of the `enqueue()` and `dequeue()` functions. Please observe, that the parameter names don't have to be specified in prototypes.

The enqueue() Function — queue.c File

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

The `enqueue()` Function — `queue.c` File

The `queue.h` header file is included in the `queue.c` file (line no. 1). This allows the compiler to verify that headers of functions defined in the latter file, correspond to prototypes of these functions. The `stdlib.h` header file is also included (line no. 2), because the definitions of `enqueue()` and `dequeue()` invoke functions that allocate and deallocate the heap memory.
The `enqueue()` function is defined similarly as in the third lecture, however it doesn't return anything and takes the number of the graph's vertex as a second argument, passed by the `vertex_number` parameter.

The dequeue() Function — queue.c File

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

The `dequeue()` Function — `queue.c` File

The `dequeue()` function is also defined in a similar way as in the third lecture, but this time it returns the number of the vertex that was stored in the deleted queue node. It has only one parameter, which is used for passing the address of the queue pointers structure (line no. 1). The number of the vertex, that is stored in the queue node to be deleted, is assigned to a local variable named `vertex_number` (line no. 5). The initial value of the variable is `-1` (line no. 3). In case the `dequeue()` was called on an empty FIFO queue, it would return that number. However, if the BFS algorithm is correctly implemented that should never happen.

Adjacency Matrix

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

#define NUMBER_OF_VERTICES 5

typedef int
 ↪  matrix[NUMBER_OF_VERTICES][NUMBER_OF_VERTICES];

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}
};
```

Adjacency Matrix

Four header files are included in the program. The `stdio.h` (line no. 1) contains prototypes of the `scanf()` and `printf()` functions, that the program uses for communicating with the user. In the `stdlib.h` header file (line no. 2) are declared the functions responsible for allocating and deallocating the heap memory. Please notice, that this file is included only once in this program, because it has its own header guard. The program also uses the `bool` type and its values, hence the `stdbool.h` file header is included (line no. 3). Finally, the last `#include` directive includes the `queue.h` header file of the library that provides the implementation of the FIFO queue (line no. 4).

The `NUMBER_OF_VERTICES` constant specifies the number of the vertices in the graph. It is used in the definition of the type of the adjacency matrix (line no. 8). The matrix itself is defined as constant (line no. 10) of the `adjacency_matrix` name and initialized (lines no. 10–16).

Types Definitions

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

Types Definitions

The neighbours list's node type (lines 1–5) is based on a structure that has two members. The first one is used for storing the vertex number (line no. 3), and the second one is a pointer to the next node in the list. The `struct vertices_array` type (lines 7–11) defines the type of the vertices array elements. Each of them is a structure with two members. The first one (line no.9), named `visited`, is used by the implementations of the DFS and the BFS algorithms. Its value indicates if the vertex has already been visited or not. The second one, called `neighbours`, is a pointer to the neighbours list of a given vertex, or more specifically to its first node. It is an empty pointer if the vertex has no adjacent vertices. The global variable `adjacency_list` (line no. 11) is a pointer to the adjacency list, or more specifically to the array of vertices that is a part of that list. Initially, it is an empty pointer, because this array is allocated dynamically. The function pointer type defined in the 13^{th} line is used to declare a parameter in one of the program's function.

The convert() Function

```
1   struct vertices_array *convert(const matrix adjacency_matrix)
2   {
3       struct vertices_array *list = NULL;
4       list = (struct vertices_array *)calloc(NUMBER_OF_VERTICES, sizeof(struct
    ↪   vertices_array));
5       if(list) {
6           for(int i=0; i < NUMBER_OF_VERTICES; i++) {
7               struct vertex **new_vertex = &list[i].neighbours;
8               for (int j = 0; j < NUMBER_OF_VERTICES; j++) {
9                   if (adjacency_matrix[i][j]) {
10                      *new_vertex = (struct vertex *)malloc(sizeof(struct
    ↪   vertex));
11                      if(*new_vertex) {
12                          (*new_vertex)->vertex_number = j;
13                          (*new_vertex)->next = NULL;
14                          new_vertex = &(*new_vertex)->next;
15                      } else fprintf(stderr, "Error creating vertex no.
    ↪   %d.\n",j);
16                  }
17              }
18          }
19      }
20      return list;
21  }
```

The `convert()` Function

The `convert()` function takes as an argument, passed by the constant parameter, the adjacency matrix and returns the address of the adjacency list[2] (line no.1). First, it tries to allocate the memory for the vertices array (line no. 4) and assigns the value returned by the `calloc()` function to the local pointer named `list` (line no. 3). Next, it checks if this variable is not an empty pointer (line no. 5). If the condition is met, then the `convert()` function uses two `for` loops to iterate over the adjacency matrix. The value of the outer loop counter (line no. 6) is used as an index in the vertices array and as the row index in the adjacency matrix. In this loop the `convert()` function takes the vertices array element, specified by the `i` variable, and assigns the address of its `neighbours` pointer field to a local pointer to a pointer, called `new_node` (line no. 7). Then, the inner loop (line no. 8) iterates over all items in the adjacency matrix row, which is also specified by the `i` variable.

[2] More specifically the address of the vertices array, that is a part of this list.

The `convert()` Function

If the value of the item, specified by the `j` variable, is 1 (line no. 9) then the function tries to allocate memory for a new node in the neighbours list (line no. 10). If the allocation is successful (verified in the line no. 11), then `convert()` initializes the new node by assigning to its `vertex_number` member the number of the adjacent vertex, specified by the `j` variable (line no. 12), and by assigning the `NULL` value to its `next` member (line no. 13). Finally, it assigns to the `new_vertex` pointer to a pointer the address of the new node `next` field. However, if the allocation in the line no. 10 failed, then the function would display an appropriate message using the standard error stream (line no. 15). That way the `convert()` function creates neighbours lists for all vertices. When both loops stop, it returns the address of the adjacency list (line no. 20) and exits.

The print_adjacency_list() Function

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

The `print_adjacency_list()` Function

The `print_adjacency_list()` function displays the adjacency list. It doesn't return any value and takes the list's address as an argument. In the `for` loop (line no. 3) the function displays a message informing, that it is going to print on the screen the adjacent vertices of the vertex specified by the loop counter (line no. 4). Then, it assigns to the local pointer, named `neighbour`, the address stored in the `neighbours` field of the vertex array element, specified by the `i` variable (line no. 5). In the `while` loop (lines 6–9) the numbers of adjacent vertices stored in the list are displayed. Note, that if the vertex, currently specified by the `i` variable, has no neighbours, then the `while` loop won't be performed. After this loop stops, the `puts()` function is invoked, to move the cursor to the next line on the screen. The `for` loop stops after it processes the last element in the vertices array.

The dfs() Function

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

The `dfs()` Function

The `dfs()` implements the Depht-First Search graph traversal algorithm. It takes two arguments, the adjacency list (passed by its first parameter) and the number of the starting vertex (passed by its second argument.) The function doesn't returns any value. It first displays the number of the currently visited vertex (line no. 3) and assigns the address stored in the `neighbours` member of the element that represents this vertex in the vertices array, to a local pointer called `neighbour` (line no. 4). Then, the function marks the current vertex as visited (line no. 5). The `while` loop (lines 6–10) traverses the neighbours list of this vertex (if it exists). It checks if the current neighbour is unvisited (line no. 7) and if so, then it calls the `dfs()` function recursively, passing as its argument the adjacency list and the number of this neighbour (line no. 8). The `while` loop stops when there is no neighbours left on the list to visit.

The bfs() Function

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

The `bfs()` Function

The `bfs()` function implements the Breadth-First Search graph traversal algorithm. It takes the same arguments as the `dfs()` function and also returns no value. In the line no. 4 it initializes the FIFO queue pointers. The structure of the queue pointers is declared in the line no. 3. Then, it adds the first node to the queue, that stores the number of the starting vertex (line no. 5). The outer `while` loop (lines 6–17) checks if the queue is not empty (line no. 6). If the condition is satisfied, then it assigns the number of the vertex stored in the queue's first node to the `vertex_number` local variable and removes that node from the queue (line no. 7). Next, it checks whether this vertex has not yet been visited (line no. 8). If so, then the loop assigns the address of its list of neighbours to a local `neighbour` pointer (line no. 9). The inner `while` loop (lines 10–13) traverses this list, on the condition that it is not empty, and adds to the FIFO queue nodes, that store numbers of the vertex neighbours.

The `bfs()` Function

When the inner loop stops, the outer one prints the vertex number (line no. 14) and marks it as visited (line no. 15). The `bfs()` function exits when the outer `while` stops.

The visit_all_vertices() Function

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

The `visit_all_vertices()` Function

The `dfs()` and `bfs()` functions won't visit all vertices if the graph is disconnected. For that reason the `visit_all_vertices()` function has been created. It doesn't return any value but takes three arguments, the first one is the adjacency list, the second one is the starting vertex number and the third one is the address of the function that implements either the `dfs()` or `bfs()` algorithm. The last argument is passed by the `algorithm` function pointer (line no. 1), whose type is defined in the beginning of the program. The function first verifies whether the starting vertex number is valid (line no. 3) and then it invokes the function that implements the graph traversal algorithm, using the function pointer (line no. 4). When this function exits the `visit_all_vertices()` checks in the `for` loop if there are any unvisited vertices left in the graph (line no. 6) and when it finds such a vertex then it invokes the function implementing the graph traversal algorithm for that vertex. After the loop stops there are no unvisited vertices in the graph. If the starting vertex number is invalid, an appropriate message is displayed.

The remove_adjacency_list() Function

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

The `remove_adjacency_list()` Function

The `remove_adjacency_list()` function is responsible for releasing the heap memory allocated for the adjacency list. It takes the address of the list pointer as an argument and returns `NULL`. This value should be assigned to the adjacency list pointer. In the `for` loop (lines 3–10) the function visits all elements of the vertices array and in the `while` loop (lines 5–9) it deletes all nodes of neighbours lists that they point to. Finally, the function deletes the vertices array (line no. 11), returns the `NULL` value (line no. 12) and exits.

The main() Function

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

The `main()` Function

In the `main()` function the program first calls the `convert()` function and assigns the value that it returns to the `adjacency_list` pointer (line no. 3). If this pointer is not an empty pointer (line no. 4) the rest of the operations in the `main()` is performed. The adjacency list is displayed (line no. 5) and the program asks the user to specify the starting vertex for the graph traversal algorithms. The number of that vertex is assigned to the `start_vertex` variable (declared in the line no. 7) by the `scanf()` function (line no. 8). Then, the `main()` function informs the user that it is going to display the result of the DFS algorithm (line no. 9) and invokes the `visit_all_vertices()` function, passing as its last argument the address of the `dfs()` function (line no. 10). Later, the `main()` function invokes `visit_all_vertices()` with the address of the `bfs()` function as its last argument (line no. 15). Before that however, it has to mark all the graph vertices explored by the `dfs()` function as unvisited. Otherwise, the `bfs()` function won't visit any of them.

The `main()` Function

This operation is performed in the `for` loop (lines 12–13). After printing the result of the BFS algorithm the `main()` function deletes the adjacency list (line no. 17), returns zero (line no. 19) and the program ends.

Summary

Graphs are a relatively simple, yet powerful, tool that can be applied for solving many problems. For example, they can be used to model social, computer, land, marine and air transport networks, electronic circuits and algorithms (flowcharts). They are also useful in Artificial Intelligence applications. Moreover, there are a lot of ready-to-use graph algorithms, so there is no need to rediscover them. Usually, finding an answer for an issue with the use of graphs requires only expressing the problem as a graph and choosing a proper algorithm. The DFS and BFS are the primary graph traversal algorithm, that can be used to develop many other useful graph algorithms. They can be applied to any type of graph, including directed, undirected, connected or disconnected.
For more information on these topics please see "Introduction to Algorithms" by T. H. Cormen, Ch. E. Leiserson and R. Rivest, or "The Algorithm Design Manual" by Steven S. Skiena.

Questions

?

The End

Thank You For Your Attention!