Fundamentals of Programming 2

Graphs And Their Reperesentations

Arkadiusz Chrobot

Department of Computer Science

June 4, 2019

Outline

Introduction

Graphs is Computer Science are data structures that have many applications. Before graphs have been applied in algorithms they were used in mathematics, where they had been introduced by Swiss mathematician Leonard Euler. It happened while he was solving the problem of the Seven Bridges of Königsberg at the same time defining a new branch of mathematics called topology. In contemporary mathematics graphs are an object of study for graph theory, set theory and overall for discrete mathematics.

Before the applications and ways of representing graphs as data structures is presented, some of the mathematical definitions associated with graphs is introduced in the lecture. Unfortunately, there is no common terminology in graph theory, so some of the definitions may be formulated a little differently in other learning materials.

Graph Theory

Directed Graph

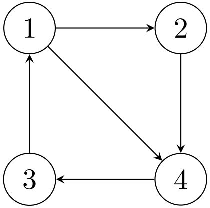
A **directed graph** or a **digraph** G is defined as a pair (V, E), where V is a finite set, which elements are vertices of the graph G, and E is a binary relation in V and $E \subseteq V \times V$. The set V is called a set of vertices for short and the E set is a set of edges of the graph G. Its elements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

Graph Theory

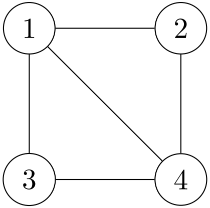
Undirected Graph

An **undirected graph** is a graph which E set is unordered. 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 with itself) in the undirected graphs.

Graph Theory



(a) A directed graph



(b) An undirected graph

Examples of graphs

Graph Theory

Types of the Edge

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

Graph Theory

Neighbourhood

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

Graph Theory
Vertex Degree

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

Graph Theory

Path

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

Graph Theory

Cycles

A path $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a **cycle** if $v_0 = v_k$. A cycle is a **simple cycle** if additionally all of its vertexes are different. A **loop** is a cycle of the length 1. The 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**.

Graph Theory

Connectedness

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

Graph Theory

Isomorphism

Two graphs $G = (E, V)$ and $G' = (V', E')$ are **isomorphic** if there exists a bijective mapping $f : v \to v'$, so that if the edge $(u, v) \in E$, then $(f(u), f(v)) \in E'$. The property of the graphs means that every undirected graph can be changed in its directed version by changing every undirected edge into two directed ones. The directed graph can be changed in its undirected version by changing every directed edge into undirected one and removing loops.

Graph Theory

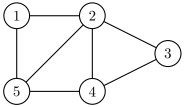
Dense and Sparse Graphs

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 equal to $\binom{n}{2}$, where n is the number of edges in the graph. A graph that has only a small fraction of the number of vertices in a dense graph is called a **sparse graph**.

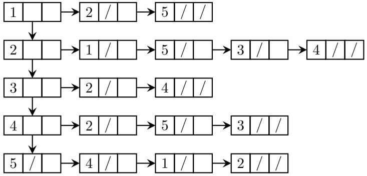
Graphs as Data Structures

There are two basic ways of representing graphs in computer memory: the adjacency matrix and the adjacency list. The adjacency list can implemented as a list of lists or as an array of pointers to lists. The adjacency matrix is a statically or dynamically allocated two-dimensional array. The rows and columns in such a matrix represents the vertices of a graph. If two vertices are connected by an edge, then in the element of the adjacency matrix located at the intersection of the column and the row associated with vertices is stored a number `1`, otherwise there is stored the `0` number. The next slides show a directed and undirected graphs and adjacency matrices and lists that represents them.

Representations of Undirected Graph



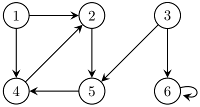
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$



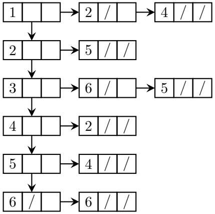
Representations of Undirected Graph

On the left side in the previous slide is an diagram of an undirected graph. In the middle of the slide is its adjacency matrix and on the right side is located the adjacency list in a from of a list of lists. The characters `/` inside elements of the adjacency list denote pointer fields that store a value of `NULL`. 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. Since the adjacency matrix is equal to its transposed self, then a memory space can be saved by storing the values of the elements of only the upper or lower triangular matrix.

Representations of Directed Graph



$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$



Representations of Directed Graph

Similarly as in the case of the undirected graph, in the previous slide are shown respectively (from left to right): the diagram of a directed graph, its adjacency matrix and its adjacency list. The adjacency matrix is still a square matrix, but its not symmetrical. Please also note, that the graph has a single edge that is a loop. In the adjacency matrix the loop is represented by the one number in the element located at the intersection of the sixth row and sixth column.

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 an array of lists. Each element of such an array or a list (the vertical list in drawings from previous slides) corresponds to one of the graph's vertices and points to the list of its neighbours i.e. adjacent vertices. The order of the vertices on the latter list has no meaning. The sum of all neighbours lists for a directed graph is $|E|$ and for an undirected graph is $2 \cdot |E|$, where $|E|$ means 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 applied to express 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 elements 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 the problem of checking the existence of an edge between two vertices or for adding or removing an edge in a graph with fixed number of vertices. On the other hand the adjacency lists are more useful for traversing the graph (the majority of graph algorithms performs 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 adjacency list and the other way. Next slides contain the source code of a program, that converts the adjacency matrix of the undirected graph, presented on previous slides, into an adjacency list.

Graphs as Data Structures

Adjacency Matrix and Base Type for the Adjacency List

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

typedef int matrix[5][5];

const matrix adjacency_matrix = {{0,1,0,0,1},
                                 {1,0,1,1,1},
                                 {0,1,0,1,0},
                                 {0,1,1,0,1},
                                 {1,1,0,1,0}};

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

Graphs as Data Structures

Adjacency Matrix and Base Type for Adjacency List

In the program are used functions defined in `stdio.h` and `stdlib.h` header files. A data type for the adjacency matrix (a two-dimensional square array of 25 elements) is defined in the 4th line. An adjacency matrix for an undirected and unweighted graph is created in lines 6–10. The base data type for the adjacency list (the list of lists) is defined in the lines 12–16. The `down` pointer field is used for linking the elements in a list of all vertices (the "vertical list") and the `next` pointer field is used for building the list of neighbours of a vertex (the "horizontal lists"). Additionally, in the 16th line is declared the `start_vertex` pointer that points an element of the adjacency list, that represents the starting vertex[1]. The pointer is a global variable, so its initial value is `NULL`.

[1]It is the top left element in the pictures from the previous slides.

Graphs as Data Structures

The `create_vertical_list()` Function

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

Graphs as Data Structures

The `create_vertical_list()` Function

The `create_vertical_list()` function creates the “vertical” list, i.e. the list of all vertices in the graph. It doesn’t return any value. By the first parameter of the function is passed the address of the `start_vertex` pointer. The parameter is also used in the function’s body for different purposes. By the second parameter is passed the adjacency matrix. The passing by constant is applied in the case, because the content of the matrix is not changed inside the function. The “vertical” list is created inside the `for` loop. The number of iterations of this loop is defined by the number of graph’s vertices given by the number of rows in the adjacency matrix. The latter number is calculated by dividing the size of the matrix data type by the size of a single row (in the case of the two-dimensional array, dereferencing the pointer to the array gives access to a single row of the matrix).

Graphs as Data Structures

The `create_vertical_list()` Function

In the `for` loop of the function the memory for subsequent elements of the "vertical" list is allocated. If the allocation is successful, then in the field `vertex_number` of the new element is stored the number of a vertex (the value of the loop counter is incremented by one, because the vertices are numerated starting from one, and the rows of matrix are indexed starting from zero), both pointer fields are initialised (10th line) and finally the address of the element's `down` pointer field is assigned to the `start_vertex` pointer (11th line). It makes it possible to avoid writing a separated code for handling the case in which the first element of the list is created. After the loop terminates the pointer points the `down` field of the last element.

Graphs as Data Structures

The convert_matrix_to_list() Function

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

Graphs as Data Structures

The `convert_matrix_to_list()` Function

The `convert_matrix_to_list()` function converts the adjacency matrix to the adjacency list. As a result it returns the address of the element of the list that represents the starting vertex and as an argument it takes the adjacency matrix. The matrix is passed by constant. The function has a local pointer variable named `start_vertex`, which is initialised with the `NULL` value. The function creates the "vertical" list by calling the `create_vertical_list()` function (4th line). If the list is successfully created, which is tested in the 5th line, then the function starts iterating over all elements of the matrix with the use of the two `for` loops. However, before it happens, two local pointers (`horizontal_pointers` and `vertical_pointer`) are declared and initialized (lines no. 6 and 7). The former is used for traversing the "horizontal" lists and the latter for traversing the "vertical" list.

Graphs as Data Structures

The `convert_matrix_to_list()` Function

The outer `for` loop iterates over the rows of adjacency list and the internal one over the columns. The value of the column's index incremented by one is the number of a graph vertex, which is potentially adjacent to the vertex specified by the row index. In the internal `for` loop the function checks if the value of the current element of matrix in different than zero (11th line). If so, then a new element of the list of list is created which represents the adjacent vertex (12th line). If the node is created successfully then a number of the vertex is stored in it (14th line) and its pointer fields are initialized (15th line). Finally, the node is added to the list of neighbours (the "horizontal" list) of current vertex (lines no. 16 and 17). In the last operation, the `horizontal_pointer` variable is used, which points to the last (initially also the first) element of the list of the adjacent vertices.

Graphs as Data Structures

The `convert_matrix_to_list()` Function

After the internal `for` loop terminates, the address of the next node on the "vertical" list (the list of all vertices) is assigned to the `vertical_pointer` variable (20th line). The same address is also stored in the `horizontal_pointer` variable. After both loops terminate the function returns the address of the starting vertex and also terminates.

Graphs as Data Structures

The print_adjacency_list() Function

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

Graphs as Data Structures

The `print_adjacency_list()` Function

The function from the previous slide displays the content of the adjacency list on the screen in a form, that is presented in the figures illustrating this data structure. It doesn't return any value but takes as an argument the address of the node in the adjacency list that represents the starting node. There are two `while` loops in the body of the function. The outer one iterates over the list of all vertices (the "vertical" list) and the inner one iterates over the lists of adjacent vertices (provided they are not empty). The outer loop is performed under the condition (3rd line) that the value of the `start_vertex` pointer is not `NULL`. If the condition is fulfilled, then the number of the vertex stored in the first node of the list of all vertices is displayed (4th line) and then the pointer to the list of adjacent vertices, declared in the 5th line is initialised. If its value is also different than `NULL`, then the internal `while` loop is performed (6th line).

Graphs as Data Structures

The `print_adjacency_list()` Function

In the internal loop the numbers of vertices from the adjacent vertices list are printed (7th line). The `horizontal_pointer` variable is used for traversing the list. The addressed of the subsequent nodes of the list are assigned to the pointer in the subsequent iterations of the loop (8th line). After the internal loop terminates the address of the next node of the “vertical” list is assigned to the `start_vertex` pointer in the outer loop (10th line) and the cursor is moved to the next line on the screen (11th line).

Graphs as Data Structures

The `remove_adjacency_list()` Function

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

Graphs as Data Structures

The `remove_adjacency_list()` Function

The `remove_adjacency_list()`, which removes the adjacency list from the computer memory is written similarly to the function described in the previous slide. Just like the `print_adjacency_list()` function it doesn't return any value, but it has a parameter which is a pointer to a pointer to the adjacency list. Also two `while` loops are performed in the function body. In the outer one, if the adjacency list is not empty (3rd line) the declared in the 4th line `horizontal_pointer` variable is initialised. If its value is different than `NULL` then the inner `while` loop is performed. In the loop the list of the vertices adjacent to the vertex represented by the node currently pointed by the `start_vertex` pointer is deleted by freeing the memory allocated to its nodes.

Graphs as Data Structures

The `remove_adjacency_list()` Function

Deleting the nodes is performed according to the algorithm applied in the singly linked list for the same task, i.e. first, the address of the next node of the adjacent vertices list is assigned to the `next_horizontal` pointer (6th and 7th lines). Next, the node pointed by `horizontal_pointer` variable is deleted (8th line) and the address stored in the `next_horizontal` pointer is assigned to the former pointer (9th line). After the whole list of adjacent vertices is removed, the node from the list of all vertices (the “vertical” list), that represents the vertex with which the vertices in the “horizontal” list were adjacent, is removed in the outer loop. The algorithm of deleting the node is the same as for the nodes of the adjacent vertices list. First, the address of the next vertex in the list is assigned to the `next_vertical` pointer (11th line). Then memory allocated to the node pointed by the `start_vertex` pointer is freed (12th line) and the address stored in the `next_vertical` is assigned to the former pointer (13th line).

Graphs as Data Structures

The `remove_adjacency_list()` Function

After both `while` loops terminate the removing of the adjacency list from the computer memory is completed and the value of the pointer to the list (the `start_vertex` variable) is `NULL`.

Graphs as Data Structures

The `main()` Function

```c
int main(void)
{
    start_vertex = convert_matrix_to_list(adjacency_matrix);
    if(start_vertex) {
        print_adjacency_list(start_vertex);
        remove_adjacency_list(&start_vertex);
    }
    return 0;
}
```

Graphs as Data Structures

The `main()` Function

In `main()` function the `convert_matrix_to_list()` function is used for converting the adjacency matrix to the adjacency list (3rd line). If the list is not empty, what is checked in the 4th line, then its content is printed with the use of the `print_adjacency_list()` function and then it is removed from the computer memory by the `remove_adjacency_list()` function. After that the `main()` function returns `0` and terminates, which means that also the program terminates.

Applications of Graphs

Graphs are a simple formalism that can be applied to many problems. Usually the issues involve the need of expressing some kind of relations. An example of such an issue is analysis of social networks. Beside that the graphs are used for modeling electric circuits, VLSI electronic integrated circuits, land, water and air routes systems and telecommunication networks. The vital advantage of using the graphs in Computer Science is that there are many ready-to-use and effective algorithms associated with those data structures. More information about the issue can be found in the “Introduction to Algorithms” book by T. H. Cormen, Ch. E. Leiserson and R. Rivest or in the “The Algorithm Design Manual” by Steven S. Skiena.

Questions

?

The End

Thank You For Your Attention!