Fundamentals of Programming 2

Graphs And Their Reperesentations

Arkadiusz Chrobot

Department of Computer Science

May 18, 2020

Outline

Introduction

Graphs are data structures that have many applications in Computer Science. Generally, they are used for expressing relations between data items. Those data structures are based on the mathematical concept of graphs discovered by a Swiss mathematician Leonhard Euler, while working on the problem of the Seven Bridges of Königsberg. At the same time he defined a new branch of mathematics called topology. In contemporary mathematics graphs are an object of study for such branches of discrete mathematics as the graph theory and the set theory.

Before the applications and representations of graphs as data structures are presented, some of the mathematical definitions associated with them will be introduced in this lecture. Unfortunately, there is no standardized terminology in the graph theory, so the definitions may differ a bit from the ones presented in other learning materials.

Graph Theory

Directed Graph

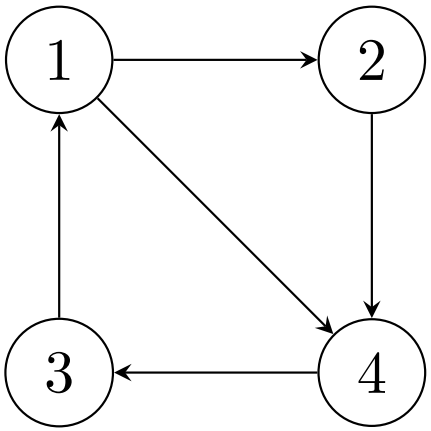
A **directed graph** or a **digraph** G is defined as a pair (V, E), where V is a finite set, which elements are called vertices or nodes of the graph G, E is a binary relation on V and $E \subseteq V \times V$. The set V is called the set of vertices. The set E is the set of edges of the graph G. Its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

Graph Theory

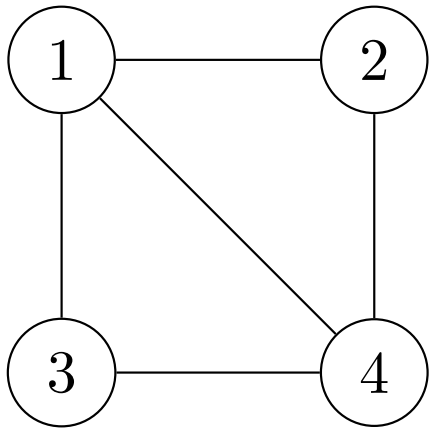
Undirected Graph

An **undirected graph** is a graph which 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 Theory



(a) A directed graph



(b) An undirected graph

Examples of graphs

Graph Theory

Edge Types

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. It **joins** u and v.

Graph Theory

Neighbourhood

A vertex v is an **adjacent vertex** to the vertex u and it is a **neighbour** of the vertex u in a graph $G = (V, E)$ if those vertices are connected by an edge (v, u). 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 a directed graph the **degree** of a vertex is a sum of its in-degrees and out-degrees.

Graph Theory

Path

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.

Graph Theory

Cycles

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**.

Graph Theory

Connectedness

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.

Graph Theory

Isomorphism

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

Graph Theory

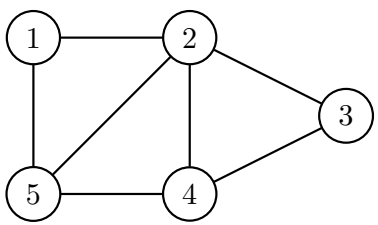
Dense and Sparse Graphs

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 $\binom{n}{2}$, 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**.

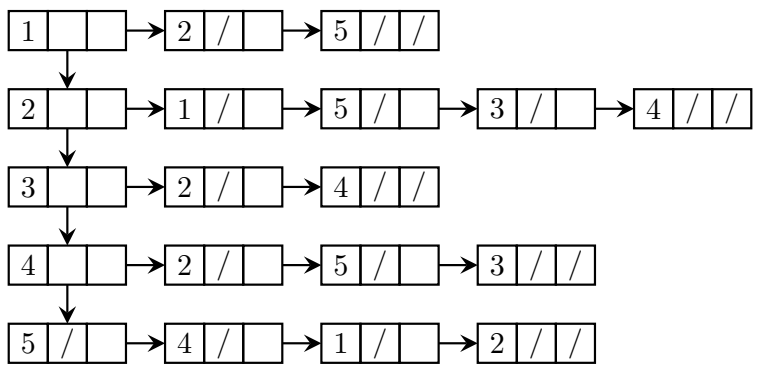
Graphs as Data Structures

There are two basic ways of representing graphs in programming languages: 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. 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 those vertices is stored `1`, otherwise there is stored `0`. The next slides present directed and undirected graphs and adjacency matrices and lists that represents them.

Representations of Undirected Graph



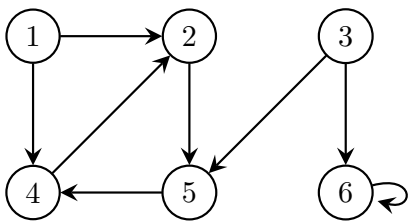
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$



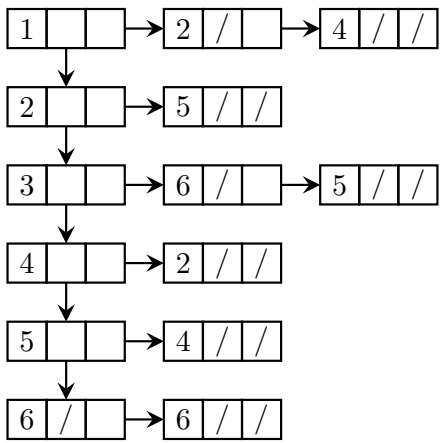
Representations of Undirected Graph

On the left side of the previous slide is a diagram of an undirected graph. In the middle of the slide is the adjacency matrix of that graph and on the right side is its adjacency list in a from of a list of lists. The characters / inside elements of the adjacency list denote pointer fields that store 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. Because the adjacency matrix equals its transposed self, then a memory space can be saved by storing only the values of the elements of either the upper or the 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. Also please note, that the graph has a single edge that is a loop. In the adjacency matrix the loop is represented by the element located at the intersection of the sixth row and sixth column, which value is `1`.

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 of vertices (the vertical list in figures 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 neighbour list vertices has no meaning. The number of vertices in all neighbour 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 expressing 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 problems of checking the existence of an edge between two vertices or for adding or removing an edge 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 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 adjacency list and the other way. Next slides show 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

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  typedef int matrix[5][5];
5
6  const matrix adjacency_matrix = {{0,1,0,0,1},
7                                   {1,0,1,1,1},
8                                   {0,1,0,1,0},
9                                   {0,1,1,0,1},
10                                  {1,1,0,1,0}};
11
12 struct vertex
13 {
14     int vertex_number;
15     struct vertex *next, *down;
16 } *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. The 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 of a list of all vertices (the "vertical list") and the `next` pointer field is used for building the lists of vertex neighbours (the "horizontal lists"). Additionally, in the 16th line is declared the `start_vertex` pointer that points to 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 figures from the previous slides.

Graphs as Data Structures

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)->down = (*start_vertex)->next = NULL;
11             start_vertex = &(*start_vertex)->down;
12         }
13     }
14 }
```

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. In this case passing by constant is used because the content of the matrix is not modified inside the function. The "vertical" list is created inside the `for` loop. The number of iterations of this loop equals the number of graph vertices defined by the number of columns 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 a two-dimensional array a single row of the matrix can be accessed by dereferencing a pointer to the array).

Graphs as Data Structures

The `create_vertical_list()` Function

In the `for` loop the memory for elements of the "vertical" list is allocated. If the allocation is successful, then the vertex number[2] is stored in the `vertex_number` field of the new element, both pointer fields are initialised (10th line) and the address of the element `down` pointer field is assigned to the `start_vertex` pointer (11th line). This makes it possible to avoid writing a separated code for handling the case when the first element of the list is created. After the loop stops the pointer points the `down` field of the last element.

[2]The value of the loop counter incremented by one — because vertices are numbered starting from one and the row of the matrix is represented by a zero-based array.

Graphs as Data Structures

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->down = new_vertex->next = NULL;
16                              horizontal_pointer->next = new_vertex;
17                              horizontal_pointer = horizontal_pointer->next;
18                          }
19                  }
20              vertical_pointer = vertical_pointer->down;
21              horizontal_pointer = vertical_pointer;
22          }
23      }
24      return start_vertex;
25  }
```

Graphs as Data Structures

The `convert_matrix_to_list()` Function

The `convert_matrix_to_list()` function converts the adjacency matrix to the adjacency list. 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 verified in the 5th line, then the function starts iterating over all elements of the matrix with the use of two `for` loops. Before that happens, two local pointers (the `horizontal_pointer` and the `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 inner one over the columns. The value of the column 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 inner `for` loop the function checks if the value of the current element of matrix is not zero (11th line). If so, then a new element of the list of lists is created which represents the adjacent vertex (12th line). If the node is created successfully then the vertex number 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 neighbours.

Graphs as Data Structures

The `convert_matrix_to_list()` Function

After the inner `for` loop stops, the address of the next element of 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 stop the `convert_matrix_to_list()` function returns the address of the starting vertex and exits.

Graphs as Data Structures

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 }
```

Graphs as Data Structures

The `print_adjacency_list()` Function

The function in the previous slide displays the content of the adjacency list on the screen in a form close to what 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 vertex. There are two `while` loops in the function. The outer loop iterates over the list of all vertices (the "vertical" list) and the inner loop iterates over the lists of adjacent vertices (provided they are not empty). The outer loop is performed if the value of the `start_vertex` pointer is not `NULL` (3rd line). If the condition is met, then the vertex number 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 other than `NULL`, then the inner `while` loop is performed (6th line).

Graphs as Data Structures

The `print_adjacency_list()` Function

In the inner loop the vertex numbers from the neighbours 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 inner loop stops 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

```
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 }
```

Graphs as Data Structures

The `remove_adjacency_list()` Function

The `remove_adjacency_list()` function which deletes the adjacency list from the computer memory is written in a similar way as the function described in the previous slide. Just like the previous 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 used in the function. 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 other than `NULL` then the inner `while` loop is performed. In this loop the list of neighbours of 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

The operation of deleting nodes is performed in a similar way as in the singly linked list. First, the address of the next node of the neighbours list is assigned to the `next_horizontal` pointer (6th and 7th lines). Next, the node pointed by the `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 adjacent to vertices in the now-deleted "horizontal" list, is removed in the outer loop. The algorithm of deleting the node is the same as for the nodes of the neighbours 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

When both `while` loops stop the adjacency list is removed from the computer memory and the value of the pointer to the list (the `start_vertex` variable) is `NULL`.

Graphs as Data Structures

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          remove_adjacency_list(&start_vertex);
7      }
8      return 0;
9  }
```

Graphs as Data Structures

The `main()` Function

In the `main()` function the `convert_matrix_to_list()` function is used for converting the adjacency matrix into the adjacency list (3rd line). Next, the `main()` function verifies if the list is not empty (4th line). If so, it prints the list content with the help of the `print_adjacency_list()` function and then the list is deleted by the `remove_adjacency_list()` function. After that the `main()` function returns `0` and exits.

Applications of Graphs

Graphs are a relatively simple tool that can be applied to many problems that involve expressing relations between some kind of entities. An example of such issues is analysis of social networks. Aside from that the graphs are used for modeling electric circuits, integrated circuits, overland routes, sea routes, air routes, telecommunication networks and many other concepts. The vital advantage of using graphs in Computer Science is that there are many ready-to-use and effective algorithms for those data structures. More information about this topic 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!