**SECTION 9. GRAPHES**

**9.1. General informationы. Basic terminology**

This section covers data structures such as graphs. In mathematics, a graph is a structure consisting of vertices (nodes) and edges (connections) that connect these vertices. In general, graphs share many similarities with trees, and one can consider trees as a special case of graphs. However, the practical value of graphs in solving real-world problems is much greater. Many tasks can be reduced to considering a set of objects whose properties are described by the relationships between them. Such objects include electrical and electronic circuits, circuit boards, road maps, aviation routes, descriptions of constructions, and games. Among the tasks that can be solved using graphs are finding the shortest path between two vertices, solving the problem of maximum capacity of pipelines, road networks, or computer networks, distributing N workers to perform M different types of work, and choosing the most efficient method of problem solving, among others.

A graph is a collection of vertices and edges that are connected by nodes. The graph G is completely defined by the set of vertices {V} and the set of edges {E} that connect all or part of these vertices. If edges are oriented, which is usually shown by an arrow, then they are called arcs, and the graph with such edges is called a directed graph (Figure 9.1 a). If edges have no orientation, then the graph is called undirected.

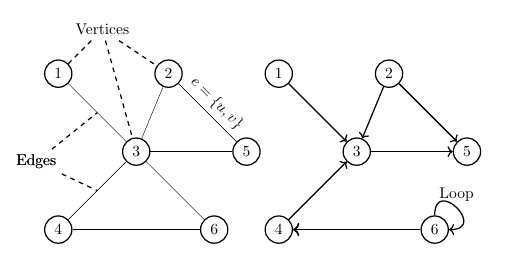


Figure 9.1. The view of a) -undirected; b) - directed

The elements of a graph are called vertices and edges. The number of vertices in the graph is called the order, and the number of edges is called the size of the graph. The endpoints of an edge e = {u,v} are called vertices (u,v), and two endpoints of the same edge are called adjacent. Two edges are called adjacent if they have a common endpoint. Two edges are called multiples if the sets of their endpoints are the same. An edge is called a loop if its ends coincide, that is, e = {u,u}. If the vertex is the beginning or end of the edge, then they (vertex and edge) are incident. The number of edges incident to a vertex is called the vertex degree (Figure 9.2).

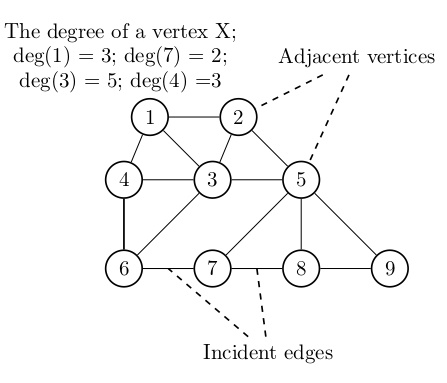


Figure 9.2. Basic graph parameters

**9.2. Graph representation methods**

Solving problems related to processing a data set organized in the form of graphs requires modeling them in computer programs. In principle, a set of vertices can be stored in an array and accessed by index. You can also store vertices using a simple-linked list or other data structure. In practice, two structures are typically used to model the structure of a graph: the adjacency matrix and the adjacency list. Adjacent in the sense that such vertices are connected by one edge.

An adjacency matrix is a two-dimensional array that indicates the existence of a relationship between two vertices. If the graph has V vertices, then the adjacency matrix is a V × V array. Fig. Table 9.3. Undirected and directed graphs and the corresponding adjacency matrices are shown, In particular, for a directed graph, the adjacency matrix is constructed in such a way that 1 indicates the presence of a connection between the vertices, 0 - its absence. So, for vertex (2) the adjacent vertices are vertices (1) and (3), and for vertex (5) – vertex (2).

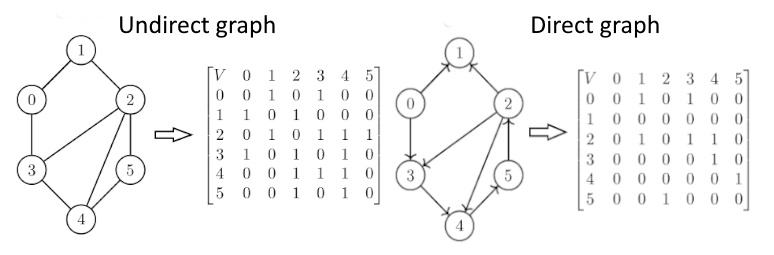


Figure 9.3. Graph representations in the adjacency matrix

The representation of a graph in the adjacency matrix is problematic. Foremost, when constructing a matrix, it is necessary to know in advance the number of vertices in the graph, which leads to the need to build a new matrix each time new vertices are inserted. In addition, the adjacency matrix consists mainly of zeros, resulting in inefficient memory usage, and if the graph contains V vertices, then memory for V2 elements must be allocated.

In some cases, a more efficient way to represent a graph is to use a adjacency list. More precisely, we are talking about arrays of related lists (see Section 4), where each individual list contains information about which vertices are adjacent to a given one. Undirected and directed graphs and their adjacency lists are depicted on Figure 9.4.

The representation of a graph in the adjacency matrix is problematic. Foremost, when constructing a matrix, it is necessary to know in advance the number of vertices in the graph, which leads to the need to build a new matrix each time new vertices are inserted. In addition, the contiguity matrix consists mostly of zeros, which leads to inefficient memory usage, if the graph contains V vertices, then memory must be allocated for V2 elements.

In some cases, a more efficient way to represent a graph is to use an adjacency list. [Undirected and directed graphs and their adjacency lists are depicted in Figure 9.4](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/).

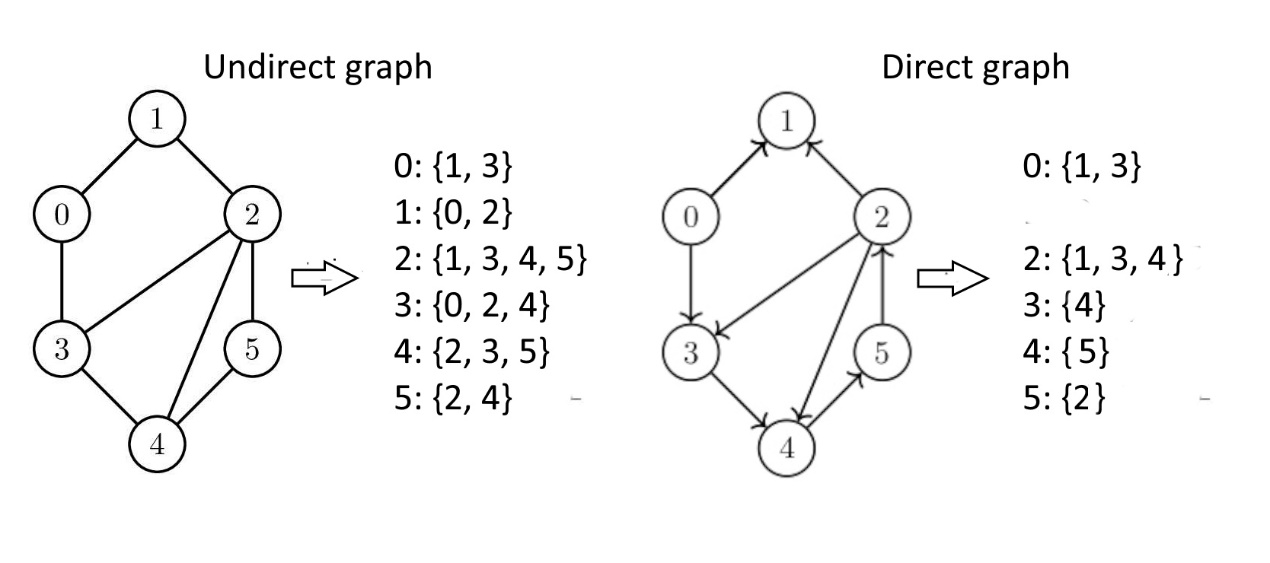


Figure 9.4. Graph representation in the adjacent list

In Golang, you can represent graphs using various data structures such as linked lists, maps, or slices of slices. With **Linked Lists** realization each node can be represented as a struct, with a value and a pointer to the next node. This is useful for representing edges in the graph. With using m**ap-**realization you can use a map where each key-value pair represents a node and its adjacent nodes. [This provides constant time performance for basic operations](https://pkg.go.dev/github.com/yourbasic/graph)[1](https://pkg.go.dev/github.com/yourbasic/graph). The realization with using **Slices of Slices** the graph is represented as a 2D array (or slice of slices in Go. The choice of graph representation often depends on the specific requirements of concrete problem, such as the need for efficient traversal, space complexity, and the type of operations you need to perform on the graph.

As you can see, in the matrix and list, the adjacency for an undirected label ‘1’ is applied to all adjacent vertices. However, for a directed graph, label ‘1’ is applied only to the vertices pointed to by the arc of the adjacent vertex. At the same time, it should be taken into account that in some cases, such as dense matrices, a matrix representation is preferable to a list of adjacent vertices.

As can be seen, the spatial complexity of directed and undirected graphs depends on its representation: for a matrix graph – O(N\*N), for a list representation – O(N+M), where N is the number of vertices, M is the number of edges. In practice, it is possible to use both kinds of graph representations, but it is useful to use an adjacency matrix to represent dense graphs  *and an adjacency list to represent* sparse graphs. Dense graphs are graphs that have a large number of edges, while sparse graphs are those that have a small number of edges.

To construct a graph in the Golang language, you need to create new types, the content of which is determined by a specific task and the choice of basic data structures from which new types will be created. Of course, all new types are created based on structure, but the number and content of its fields depend on the type of graph. In particular, a description of a loaded graph whose edges are marked, for example, by the distance between vertices, should include a weight field (Fig. 9.5.):

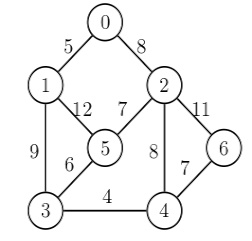


Figure 9.5. Weighted graph

In the simplest case, work with graphs requires the inclusion of fields for vertices and/or edges in the structure. To create a Graph type with support for graph representation using the list of adjacencies, use different basic structures of the language Golang (two-dimensional slices, maps, cuts, elements of which are simply connected lists [Korman T. X. etc. Algorithms: construction and analysis, 3rd d. - M.: LLC "J.D. Williams", 2013. P.626. ]. Next we will use types: to describe the vertices, to list the adjacent vertices, and to describe the graph (Table. 9.1.):

Table. 9.1. Several types of graph algorithms

|  |  |  |
| --- | --- | --- |
| Vertex description | List of adjacent vertices | Graph description |
| type Vertices struct {      data int      next \* ListVertAdj      last \* ListVertAdj  } | type ListVertAdj struct {   id int   next \* ListVertAdj  } | type Graph struct {      size int      node []\*Vertices  } |
| *data* – vertex name;  *next* – address of the nearest adjacent vertex;  *last* – last vertex address. | *id* - vertex name;  *next* – adjacent vertex address (recursion) | *size* – number of vertices;  node []\* Vertices –  vertex address collection |

**9.3. Graph algorithms**

Due to the wide distribution of graphs, there are many algorithms for their processing [https://kalkicode.com/adjacency-matrixrepresentation-o]. Problems solved within the framework of graph theory can be conventionally divided into several groups:

• Definition of a graph and its properties;

• Actions with graphs;

• Routes, circuits and cycles, contours;

• Calculation of graph characteristics;

• Tasks on graphs.

Consider the main algorithms used in these problems and implemented, as before, within the framework of hybrid programming DRAKON- + Golang.  
After a graph is constructed and its properties tested, it is natural to traverse its vertices. The general problem is formulated as follows: to circumvent the graph, starting at a given vertex and moving along the edges to other vertices, visit all vertices connected to the original. There are two main ways to circumvent graphs: depth traversal and width traversal, which provide a brute-force of all connected vertices.

The difference between depth search and width search is that (in the case of an undirected graph), the result of a depth-first search algorithm is some route by which it is possible to circumvent sequentially all vertices of the graph available from the initial vertex. This is fundamentally different from a breadth search, where multiple vertices are simultaneously processed, and only one vertex is processed in depth at each time. On the other hand, depth-first searches do not find the shortest paths, but they do apply to situations where the graph is unknown in its entirety and is investigated by some automated device.

9.3.1. The depth-first search algorithm

Depth-first search algorithm (DFS) allows constructing a traversal of a directed or undirected graph that visits all vertices accessible from the initial vertex. The choice of depth-first search is justified in situations where it is necessary to investigate the unknown structure of a real object represented as a graph. If the graph is directed, then depth-first search builds a tree of paths from the initial vertex to all accessible from it. In practice, depth-first crawling is used in the analysis of options for choosing further behaviour in games.

Depth-first search can be thought of in this way. Let an observer located at one of the vertices of the graph be given the task of visiting all its vertices. Being at this vertex, the observer sees the edges emanating from this vertex. In the case of a sufficiently complex structure, the graph observes the risk of developing some vertices several times and, in the end, looping. To avoid this situation, the observer should mark all visited vertices and should not go to the vertex that he has already visited. The algorithm may look like this:

* Select the vertex from which the graph traversal begins;
* Go to the next neighboring vertex not visited before;
* Run from this vertex the traversal bypass algorithm;
* Return to the starting vertex;
* Repeat the process for all previously unvisited adjacent vertices.

Thus, to implement the algorithm, you will need to note which vertices the researcher was at and which vertices he was not at. The annotation will be made in the visited list, where visited[i] == True for visited vertices, and visited[i] == false for unvisited ones. The mark "about visiting vertices" is placed when entering this vertex. Since the purpose of depth traversal is often to construct a depth traversal tree, the preceding vertices are retained for each active vertex.   
 The depth-first search algorithm takes the form of a recursive function (DFS), where start is the number of the vertex from which the traverse is started. Depth-first search starts by visiting the original vertex vi, after which all adjacent vertices are visited recursively. A visit to a vertex is recorded in the logical array visit. The depth-first search algorithm is based on the use of a recursive function that pops a vertex from the stack, checking it for attendance. If the vertex has already been visited, the traversal continues the search through the adjacency list until a dead vertex is reached. An illustration of the graph bypass in depth is shown in Figure 9.7.

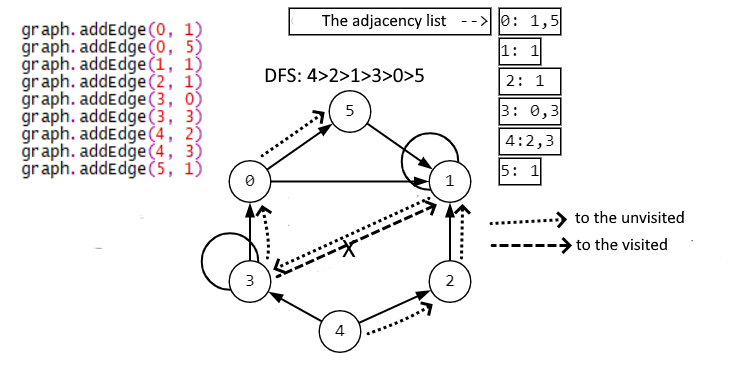


Figure 9.7. Depth-first search of the graph by adjacency list

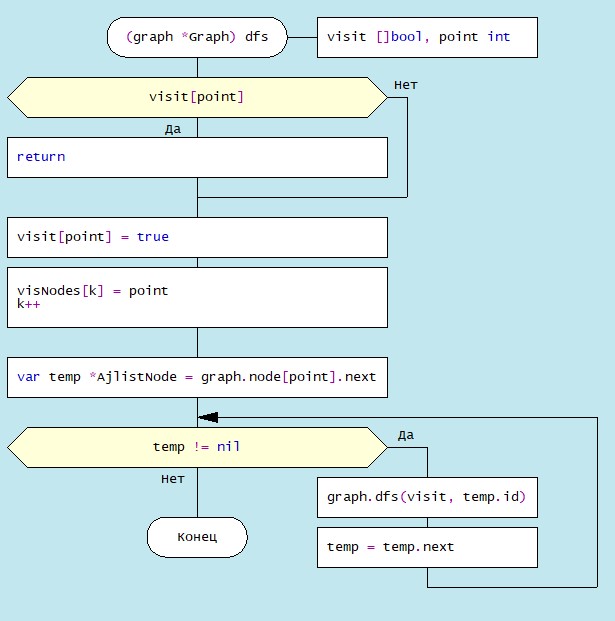
The DRAKON-diagram of depth-first search algorithm is represented in Figure 9.9.  
 

Figure 9.9. The DRAKON-diagram of depth-first search algorithm

9.3.2. The breadth-first search algorithm

 Breadth first search is carried out like a propagating wave, starting from visiting the original vertex, which is chosen arbitrarily. Then you visit the neighbouring peaks, and then you visit the neighbours, and so on. Sometimes this traversal is compared to the spread of fire through adjacent vertices: first, the vertices that are adjacent to the original vertex are ignited, then the same is the same with respect to the other vertices.

Breadth-first search can be explained differently, more formally. Traversal occurs as the distance from the initial vertex increases. Here, the distance is determined by the number of edges between the vertices: in particular, the distance between adjacent vertices relative to the reference is 1 (Figure 9.9):

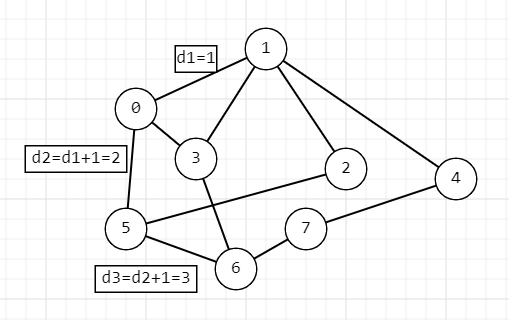


Figure 9.9. Distance from the original vertex

An illustration of breadth-first search using the example of the Hamiltonian graph is given in Figure 9.10:

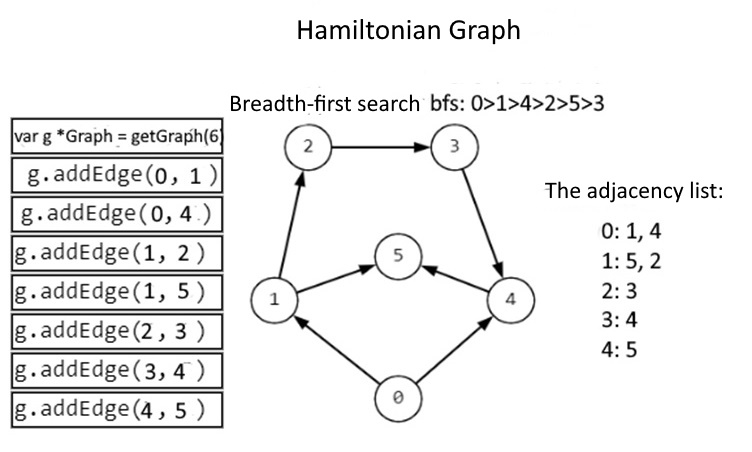


Figure 9.10. Breadth-first search of graph (wave traversal)

The DRAKON-diagram of the graph traversal algorithm is represented in Figure 9.11.

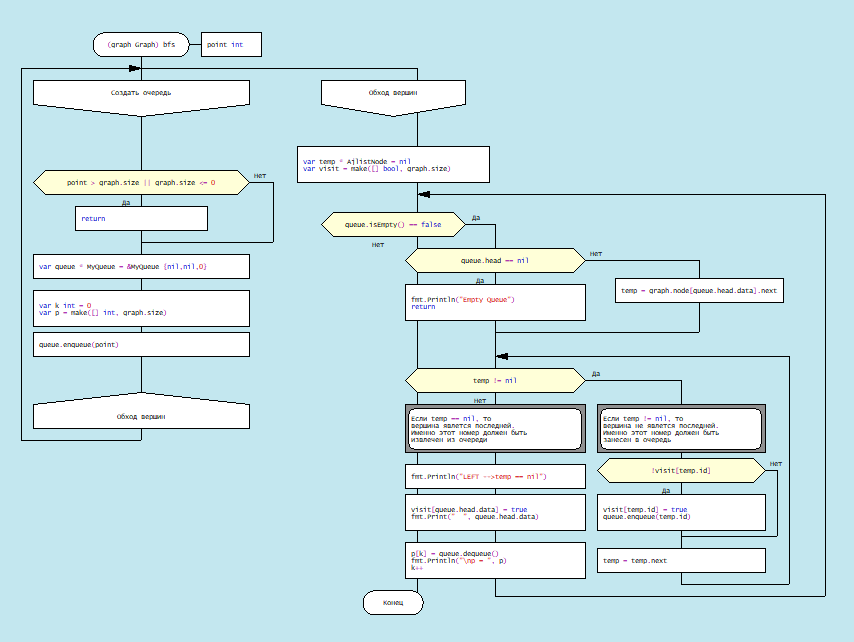
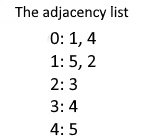


Рис. 9.11. DRAKON-diagram of Breadth-first traversal algorithm

Since the algorithm of breadth-first traversal is quite complicated, consider it in more detail. Firstly, let’s pay attention to the entered types of variables:   
 а) type Vertices - for the information structure of a vertex; consists of three fields: data int - vertex key (in this case its number), next - address of the first adjacent vertex of the graph, last - address of the last adjacent vertex of the graph:

|  |  |  |
| --- | --- | --- |
| Vertices { data int, next \*AjlistNod*,*  last \*AjlistNod } | | |
| data | next | last |
| 0 | {1 0xc000038260} | {4 <nil>} |
| 1 | {2 0xc0000382c0} | {5 <nil>} |
| 2 | {3 <nil>} | {3 <nil>} |
| 3 | { 4 <nil>} | {4 <nil>} |
| 4 | {5 <nil>} | {5 <nil>} |



b) type AjlistNode – for a single vertex, the actual structure of the single linked list;

|  |  |
| --- | --- |
| AjlistNode {id int, next \* AjlistNod} | |
| id | next |
| 0 | {1 0xc000038260} |

c) type QNode – for a queue element whose structure looks like:

|  |  |
| --- | --- |
| QNode {data int, next \* QNode} | |
| data | next |
| 1 | 0xc00014a028 |

d) type MyQueue - for <queue> variable whose structure looks like:

|  |  |  |
| --- | --- | --- |
| *MyQueue* {head \**QNode, tail* \**QNode, count int*} | | |
| *head* | *tail* | *count* |
| 0xc0000383f0 | *0xc000038410* | *2* |

e) type Graph – for variable, representing graph structure: size - number of vertices, node – slice, containing addresses of adjacent vertices:

|  |  |
| --- | --- |
| *Graph* {size int*, node [] \** Vertices} | |
| 6 | *[0xc000096060 0xc000096090 0xc0000960c0 0xc0000960f0 0xc000096120 0xc000096150]}* |

The algorithm for breadth-first traversal begins with the selection of the original vertex, let this vertex be marked as "0" (Figure 9.10). This vertex should be placed in a queue queue of type \*MyQueue:

var queue \* MyQueue = &MyQueue {nil,nil,0}

The queue structure is associated with the enqueuing method, queue.enqueue(point), where point is the vertex key (Figure 9.12).

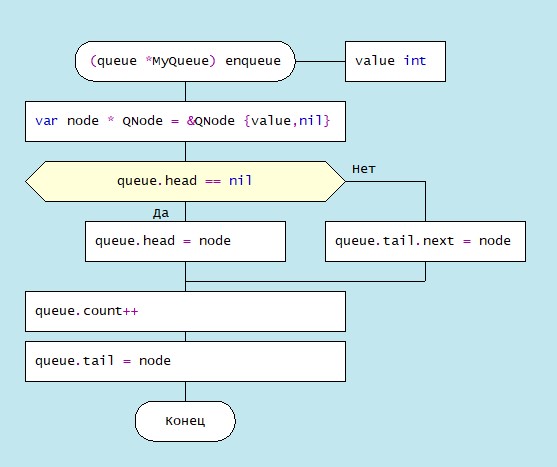


Figure 9.12. DRAKON-diagram of the queue.enqueue method

In the enqueue module, the new element of queue as an instance of a node structure of type QNode is a singly linked list. Next, the address of the next node is checked for the head element, and if it is equal to nil, then this node is assigned the value node, thus forming the nodes of the queue queue. If it is not equal to nil, then the value of node is assigned to the last element of the queue: queue.tail.next = node. In this case, the queue.count queue length is increased by 1.

Deleting a vertex from the queue is implemented by dequeue, associated with the queue instance whose algorithm DRAKON-diagram is shown in Figure 9.13:

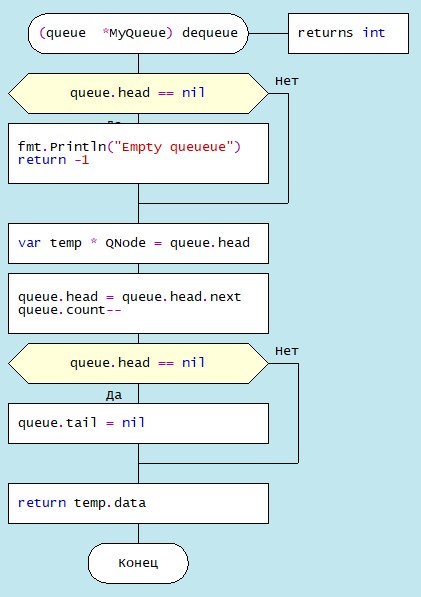


Figure 9.13. DRAKON-diagram of method queue.dequeue

**9.4. Choosing a path between vertices in a directed graph**

The choice of a path between two vertices in a directed graph, which is of practical importance in the sense of finding the optimal path between two settlements connected by one-way roads, can be carried out according to various criteria. Dijkstra's algorithm is well known, which determines the minimum distance between two given nodes of a graph. Let's complicate this task a little: in order to determine the optimal path, find the distances of all possible paths between two geographical points, as well as the total fare, taking into account the different cost per 1 km on different sections of the road.

To solve this problem, it is necessary to create a weighted graph, more specifically, to load the edges between all vertices with the values of the distances between the points and the fare per 1 kilometre (Figure 9.14).

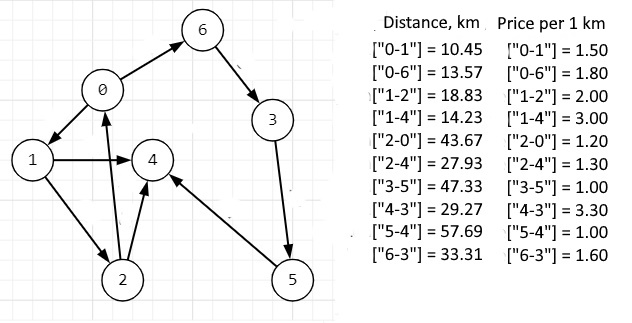


Figure 9.14. Weighted directed graph with source data

The algorithm for finding all possible paths between two vertices is based on the sequential analysis of the list of vertex connections and recursive access to the findPath module, which implements a traversal of nodes with fixing a visit to each vertex. The DRAKON-diagram for the findPath module is shown in Figure 9.13. The algorithm consists of three parts: in the first part, the correctness of the module parameters (the names of the initial and final vertices) is set, and the fact of visiting the current vertex is checked. The second part traverses through the nodes of the graph that have not yet been visited, forming a string consisting of visited vertex names and a separator between them, for example, “2-0-1-4-3-5” (Figure 9.15). In the third part, when the final vertex is reached, that is, when the condition “start == last” is met, the minimum path between these vertices is determined. In addition, a possible path string, for example, "2-0-1-4-3-5", is divided into sections of the type "2-0" in order to form keys for maps containing the distances between the corresponding vertices and the cost of travelling 1 km, these areas.

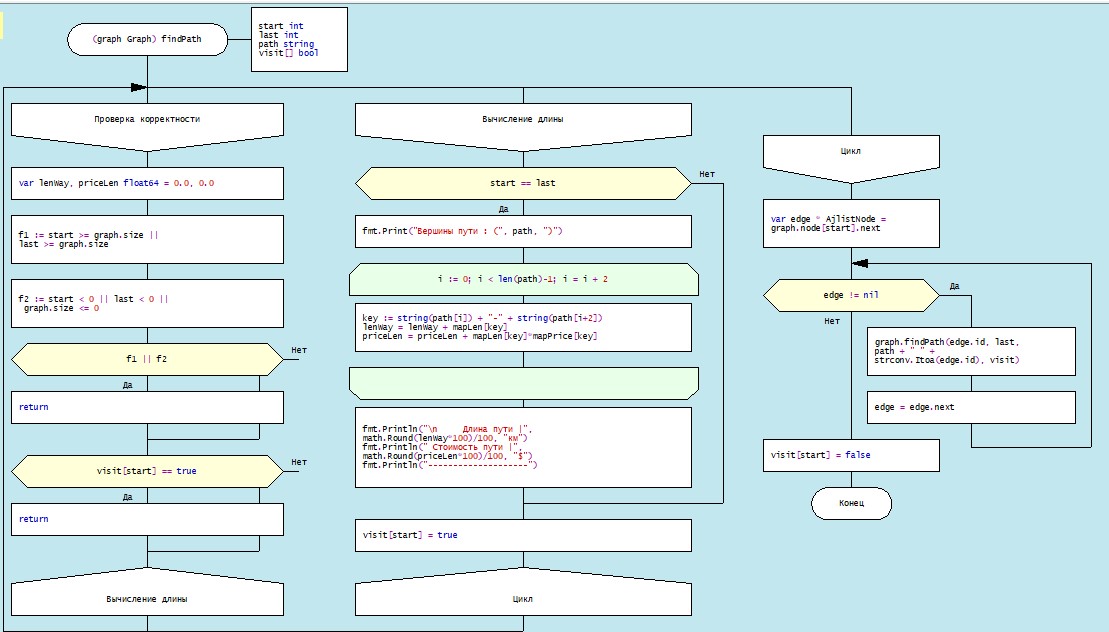


Figure 9.15. DRAKON-diagram of method findPath

Consider the problem of finding all possible paths between two vertices of the graph shown in Figure 9.16:

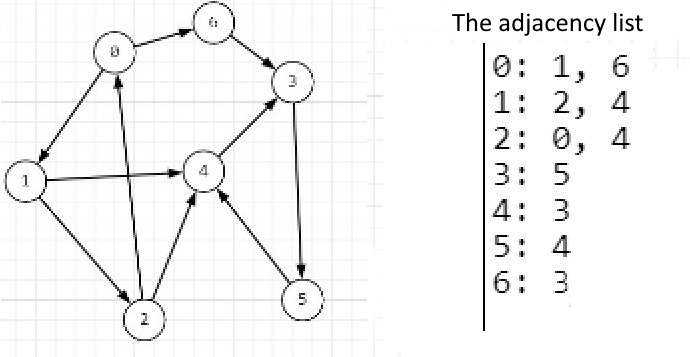
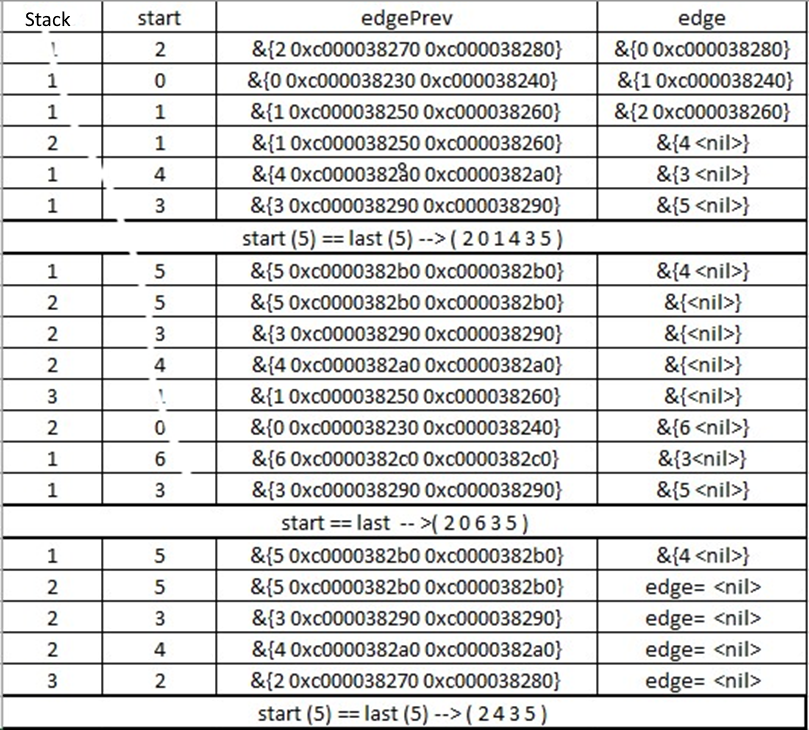


Figure 9.16. Graph and the adjacency list

The algorithm implementation process is shown in Table 9.2. The "stack" column shows the number of recursive calls to the findPath module, whose parameters are: initial (start int) and final (last int) vertices, path (path string) and slice of logical states of visiting vertices (visit[] bool). The "start" column displays the keys of the current vertices, the "edgePrev" column displays a singly linked list that contains the name of the current vertex and its address, as well as the address of the next entry, and the "edge" column displays the name and address of the next entry. The algorithm, using the adjacency list, passes, if the condition (edge != nil) is met, in a loop through the corresponding vertices of the graph, filling, if the condition (edge == nil) is met, the contents of the stack.

Table.9.2. Method of implementation process findPath 

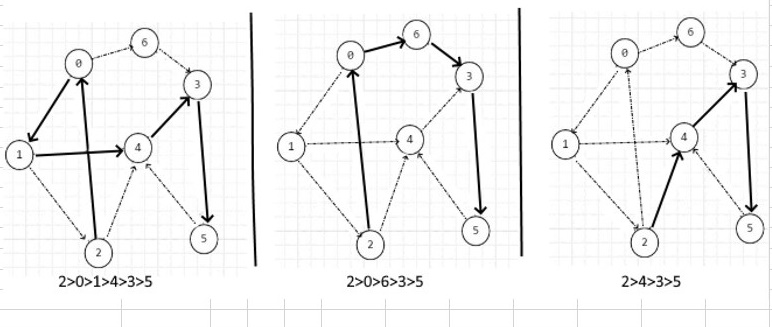


Figure 9.16. Output of implementation method findPath

**9.5. Algorithm for determining the minimum number of edges between two nodes**

In the previous subsection, the algorithm of finding all possible ways of moving from one vertex to another was considered. It can be seen that the number of edges in the different pathways varies from 3 to 5. Next, consider the algorithm for determining the minimum number of edges between two vertices (Figure 9.17):

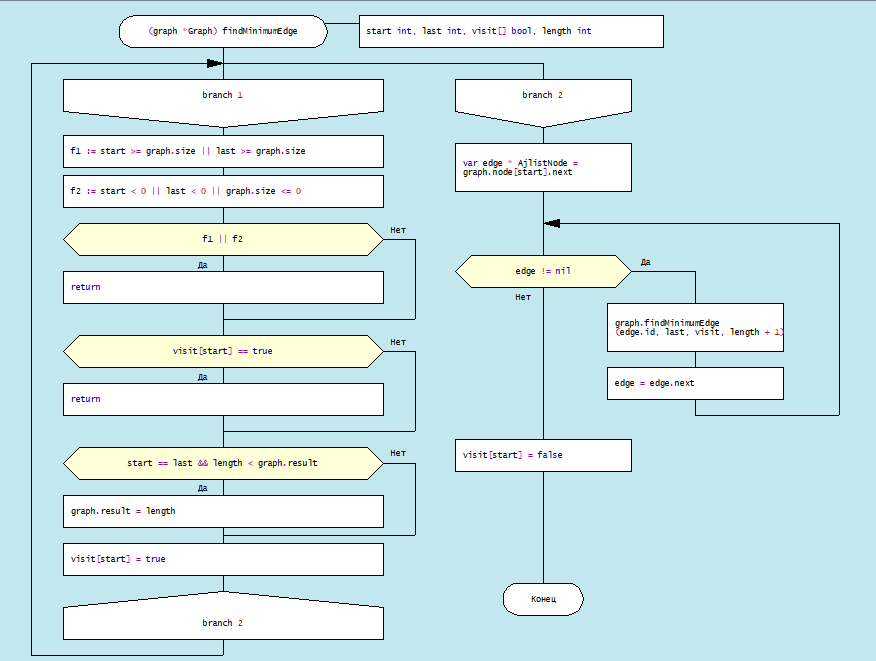


Рис.9.17. DRAKON-diagram of the minimum number of edges algorithm

Table. 9.3. Process of module implementation find

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| k | start | last | length | graph.result | edge | edge.id | edge.next |
| 1 | 2 | 5 | 0 | max | &{0 0xc000102270} / 0 | 0 | &{4 <nil> |
| 2 | 0 | 5 | 1 | max | &{1 0xc000038240} / 1 | 1 | &{6 <nil>}> |
| 3 | 1 | 5 | 2 | max | &{2 0xc000038260} / 2 | 2 | &{4 <nil>} |
| 4 | 2 | visit[start; The node 2 is already visited . ] == true Exiting a module. | | | | | |
| 5 | 4 | 5 | 3 | max | &{3 <nil>} | 3 | <nil> |
| 6 | 3 | 5 | 4 | max | &{5 <nil>} | 5 | <nil> |
| 7 | 5 | 5 | 5 | max | {4 <nil>} | 4 | <nil> |
| 8 | 4 | if visit[start] == true {return} --> f or (edge != nil --> Node traversal: 4,5,3,4,1,These nodes have edge = <nil>, so FindMinimumEdges exits, popping the previous function parameters from the stack. | | | | | |
| 9 | 6 | 5 | 2 | 5 | &{3 <nil>} | 3 | <nil> |
| 10 | 3 | 5 | 3 | 5 | &{5 <nil>} | 5 | <nil> |
| 11 | 5 | 5 | 4 | 4 | &{4 <nil>} | 4 | <nil> |
| 12 | 4 | 5 | 5 | 4 | &{3 <nil>} | 3 | <nil> |
| 13 | 3 | if visit[start] == true {return} --> for (edge != nil --> Traverse through nodes:4,5,3,6,0,2; In these nodes, edge = <nil>, so the FindMinimumEfges is exited, as a result of which the previous values are popped from the stack, in particular length, i.e. length = 0. | | | | | |
| 14 | 4 | 5 | 1 | 4 | &{3 <nil>} | 3 | <nil> |
| 15 | 3 | 5 | 2 | 4 | &{5 <nil>} | 5 | <nil> |
| 16 | 5 | 5 | 3 | 4 | &{4 <nil>} | 4 | <nil> |
| 17 | 4 | if visit[start] == true {return} --> for (edge != nil --> Node traversal: 5,2. In these nodes, edge = <nil>, so FindMinimumEfges exits, resulting in the stack retrieves the previous previous parameters of the function | | | | | |

Литература по Разделу 8

Кормен Т. Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д. Вильямс», 2013. ˗ С.626

<https://kalkicode.com/adjacency-matrix-representation-o>

[Помощь с программированием | Статьи, калькуляторы, сервисы |](https://programforyou.ru/#link-about) Programforyou

[Пример адресов Golang - itcodet](https://www.itcodet.com/golang/golang-addresses-class-examples.html)