**SECTION 9. GRAPHES**

**9.1. General information and 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.

More strictly, a graph G is given by a set of vertices {V} and a set of edges {E} connecting all or part of these vertices. Thus, a graph G is completely defined as {V, E}. If edges are oriented, they are called arcs, and a graph with such edges is called an oriented graph (Figure 9.1 a). If the edges have no orientation, the graph is called an undirected graph:

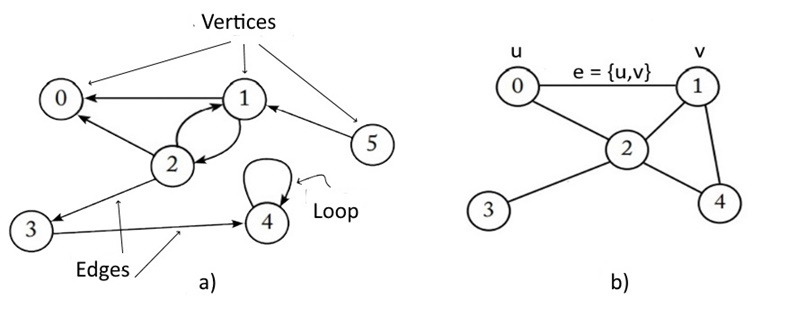


Figure 9.1. The view of a) -undirected; b) - directed graph

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).

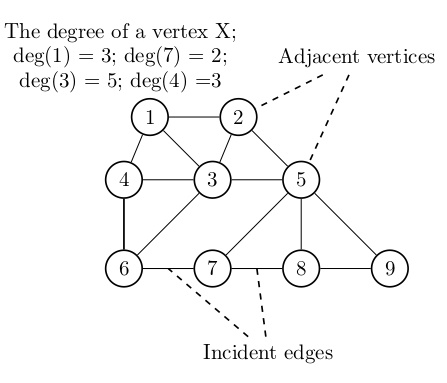


Figure 9.2. Basic graph parameters

**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).

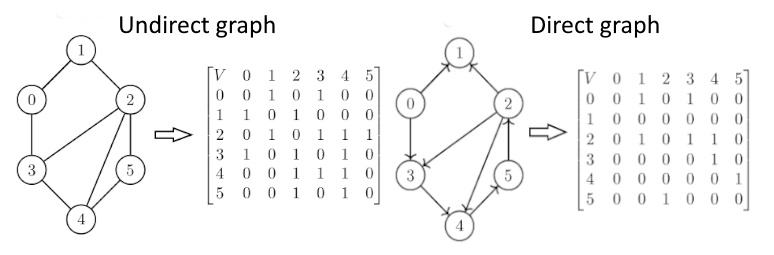


Figure 9.3. Graph representations in the adjacency matrix

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. Undirected and directed graphs and their adjacency lists are depicted on Figure 9.4.

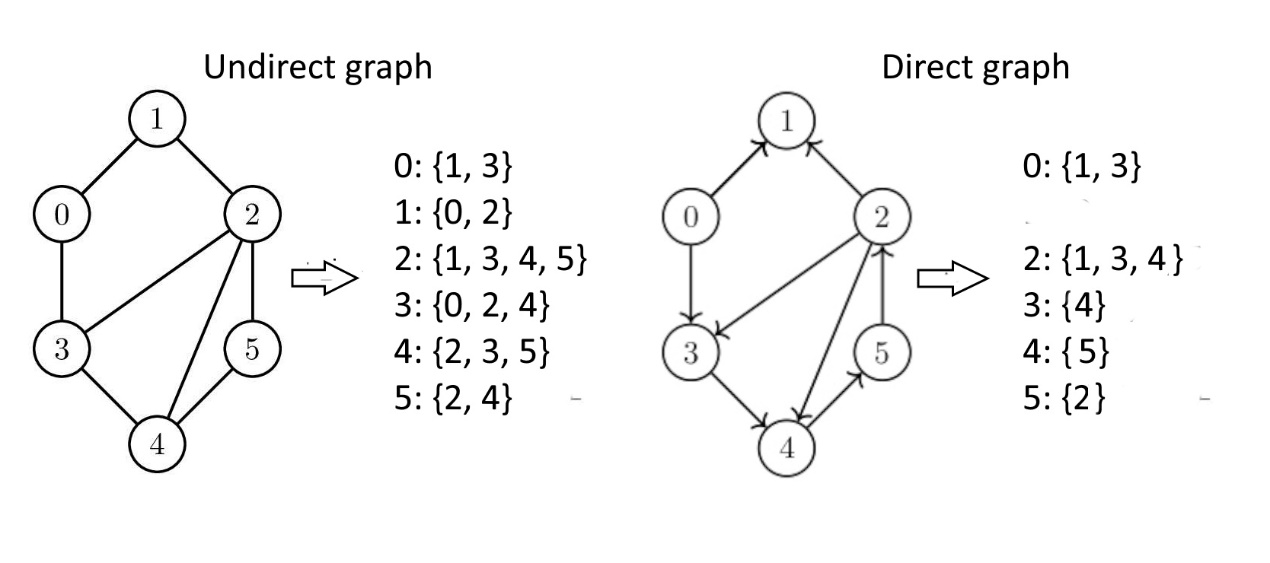


Figure 9.4. Graph representation in the adjacent list

The matrix and adjacency list for an undirected graph differ from a directed graph in that the '1' labels in the matrix are applied to all adjacent vertices (Fig. 9.4.). In most practical cases, graph processing problems contain values that are associated with either the vertices of the graph or its edges. For example, in the problem of finding the optimal path between two points, the edges of the graph need to be loaded with such data as the distance between these vertices and the cost of travel per 1 km (Fig. 9.5):

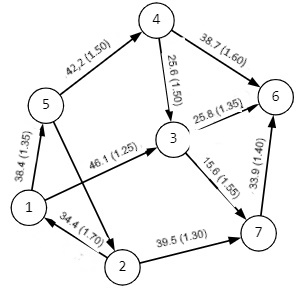


Fig.9.5. Loaded graph

**9.3. Graph interface and algorithms**

The interface of a graph as a data structure is a set of methods that define the basic operations available for working with a graph. This interface includes methods for getting the number of vertices and edges in a graph, accessing a specific edge between vertices, adding and removing vertices and edges, and methods for working with iterators to get information about incoming and outgoing edges for a particular vertex. The graph interface allows you to work with graph data structures and implement them in accordance with the requirements of a specific task.

In the Golang programming language, a graph interface can be implemented using structures and methods, the content of which is determined by a specific task and the choice of basic data structures from which new types will be created. In the simplest case, working with graphs requires the inclusion of fields in the graph structure to display vertices and/or edges. To create a Graph type With support for representing a graph using a list of contiguities, various basic structures of the Golang language are used (slices whose elements are Linked Lists, maps, two-dimensional slices [] []).

Let's look at some examples of the construction of a directed graph as shown in Fig. 9.6. using the basic structures mentioned above (Table 9.1.).

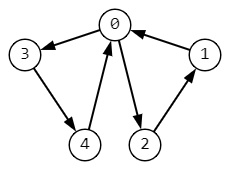


Figure. 9.6. Directed graph for interface illustration

When choosing how to implement the graph interface using the Golang language, you should be guided by the following criteria:

1. Linked List:

- it is used when you want to store relationships between vertices in a graph as an adjacency list;

- suitable for implementing graphs with a large number of edges and a small number of vertices;

- allows you to efficiently add and remove edges between vertices.

2. Map:

- it is used when you want to store relationships between vertices of a graph in the form of a dictionary, where the keys are vertices and the values are their neighbors;

- suitable for implementing graphs with an arbitrary number of vertices and edges;

- allows you to quickly get a list of neighbors for a given vertex.

3. Slice:

- it is used when you want to store relationships between vertices in a graph as slices, where the slice indexes correspond to vertices and the values are their neighbors;

- suitable for implementing graphs with a fixed number of vertices;

- allows you to efficiently get a list of neighbors for a given vertex.

The specific implementation method depends on the specific requirements for the graph, such as the number of vertices and edges, the operations that will be performed most often (adding vertices, adding edges, finding neighbors, and so on), and the specifics of the problem in which the graph will be used.

Table. 9.1. Basic structures for creating graph interfaces

|  |  |  |
| --- | --- | --- |
| Linked List | Map | Slice |
| package main  import "fmt"  type Node struct {  value int  next \*Node  }  type Graph struct {  vertices []\*Node  }  func (g \*Graph) addVertex(value int) {  newNode := &Node{value: value}  g.vertices = append(g.vertices, newNode)  }  func (g \*Graph) addEdge(v1, v2 int) {  node1 := g.vertices[v1]  node2 := g.vertices[v2]  newNode := &Node{value: v2}  newNode.next = node1.next  node1.next = newNode  newNode = &Node{value: v1}  newNode.next = node2.nextnode2.next = newNode  func (g \*Graph) getNeighbors(vertex int) {  node := g.vertices[vertex]  for node != nil {  fmt.Println(node.value)  node = node.next  }  }  func main() {  graph := Graph{}  graph.AddVertex(0)  graph.AddVertex(1)  graph.AddVertex(2)  graph.AddVertex(3)  graph.AddVertex(4)  graph.AddEdge(0, 1)  graph.AddEdge(0, 2)  graph.AddEdge(0, 3)  graph.AddEdge(1, 4)  graph.AddEdge(2, 4)  graph.getNeighbors(4)  } | package main  import "fmt"  type Graph struct {  vertices map[int][]int  }  func (g \*Graph) addVertex(vertex int) {  g.vertices[vertex] = []int{}  }  func (g \*Graph) addEdge(v1, v2 int) {  g.vertices[v1] = append(g.vertices[v1], v2)  g.vertices[v2] = append(g.vertices[v2], v1)  }  func (g \*Graph) getNeighbors(vertex int) {  fmt.Println(g.vertices[vertex])  }  func main() {  graph := NewGraph()  graph.adVertex(0)  graph.addVertex(1)  graph.addVertex(2)  graph.addVertex(3)  graph.addVertex(4)  graph.addEdge(0, 1)  graph.addEdge(0, 2)  graph.addEdge(0, 3)  graph.addEdge(1, 4)  graph.addEdge(2, 4)  graph.getNeighbors(0)  } | package main  import "fmt"  type Graph struct {  vertices [][]int  }  func (g \*Graph) addVertex() {  g.vertices = append(g.vertices, []int{})  }  func (g \*Graph) addEdge(v1, v2 int) {  g.vertices[v1] = append(g.vertices[v1], v2)  g.vertices[v2] = append(g.vertices[v2], v1)  }  func (g \*Graph) getNeighbors(vertex int) {  fmt.Println(g.vertices[vertex])  }  func main() {  graph := Graph{}  graph.addVertex()  graph.addVertex()  graph.addVertex()  graph.addEdge(0, 1)  graph.addEdge(1, 2)  graph.getNeighbors(0)  }  func main() {  graph := Graph{}  graph.addVertex()  graph.addVertex()  graph.addVertex()  graph.addVertex()  graph.addVertex()  graph.addEdge(0, 1)  graph.addEdge(0, 2)  graph.addEdge(0, 3)  graph.addEdge(1, 4)  graph.addEdge(2, 4)  graph.getNeighbors(0)  } |

**9.4. Basic graph algorithms**

Due to the widespread use of graphs, there are a large number of algorithms for processing them. Among the problems solved within the framework of graph theory are:

* Definition of the graph and its properties;
* Actions with graphs;
* Routes, chains and cycles, contours**;**
* Calculation of graph characteristics;

Let's have a look at the main algorithms used for these tasks, implemented, as before, within the framework of DRAKON + Golang hybrid programming. Once the graph has been constructed and its properties have been checked, the problem of traversing the graph through its vertices can naturally arise. The general task is formulated as follows: to traverse the graph, starting from a given vertex and moving along the edges to other vertices, in order to visit all vertices.

There are two main ways of traversing graphs: depth-first traversal (dfs) and breadth traversal (bfs), which ensure that all connected vertices are traversed. The difference between depth-first and breadth-first search is that the result of a depth-first search algorithm is a route that can sequentially traverse all the vertices of the graph that are accessible from the initial vertex. This is fundamentally different from breadth-first search, which traverses all the vertices of one level first, and then traverses the vertices of the next levels sequentially.

**9.4.1. Depth-first traversal (DFS)**

9.4.1.1. Graph Formation

Recall that the depth-first search algorithm consists of sequentially traversing the vertices of the graph accessible from the initial vertex. To implement this algorithm, the graph will be represented using an adjacency list (Table 9.1). A description of the types of variables used in the graph processing program is shown in Fig. 9.7.

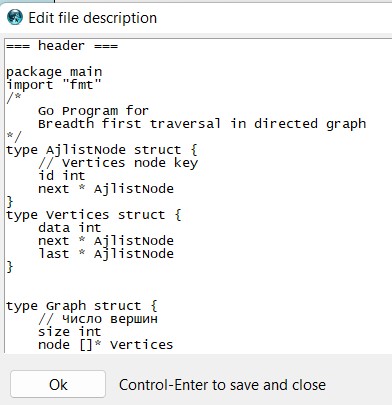


Fig. 9.7. Description of depth-in variable types

The following structures are used in this description:

1. AjlistNode **is**  a structure that represents a node in a linked list to represent the adjacency of a graph. It contains the following fields:

- id intis an integer value that represents the vertex identifier of the graph.

- next \*AjlistNode - pointer to the next node in the linked list.

2. Verticesis a data structure that represents the vertex of a graph in a graph processing program. It contains the following fields:

- data intis an integer value that represents the data associated with the vertex of the graph.

- next \*AjlistNodeis a pointer to the first node in a linked list of adjacent vertices.

- last \*AjlistNode is a pointer to the last node in a linked list of adjacent vertices.

3. Graph – the structure describes a graph consisting of nodes of the **Vertices** type and having two fields**:**

- sizerepresents the number of nodes in the graph (graph size);

- node is a slice of pointers to these nodes.

The full Drakon-diagram of the graph depth-first traversal algorithm is presented in Fig.9.8.

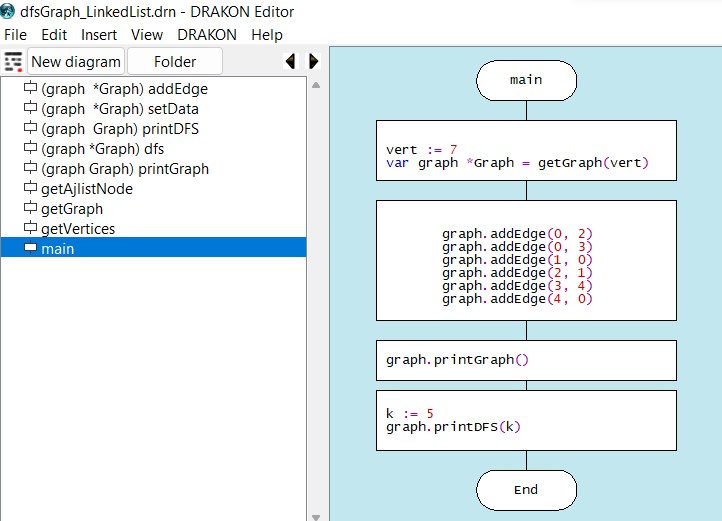
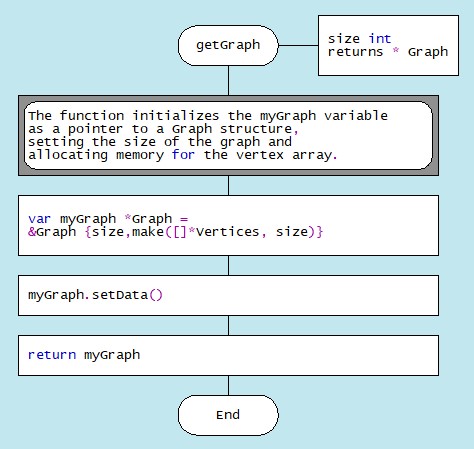
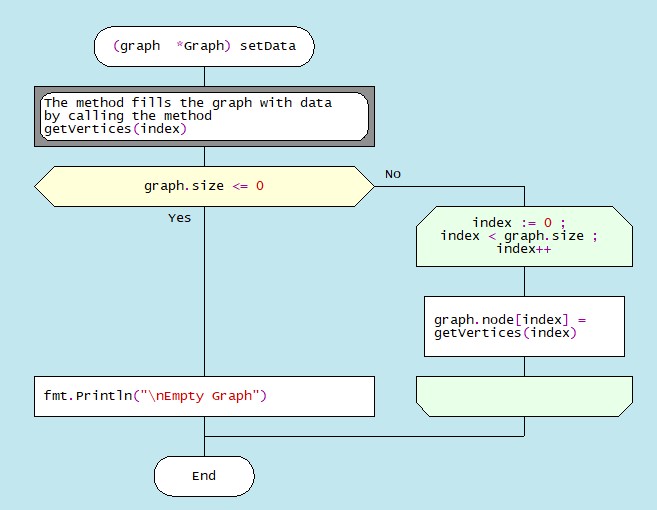


Figure 9.8. The full Drakon-diagram of algorithm in depth

In the main() function, a graph variable of the \*Graph pointer type is created, which describes a graph consisting of data structures of type Vertices, where size represents the number of nodes in the graph, and node is a slice of pointers to these nodes (Figure 9.9.).

After creating a new instance of the graph, the addEdge(start, last) method is executed, which adds an edge between the start and last nodes. To create a node for an adjacency list, the method uses the var edge \*AjlistNode variable, which initializes a unique identifier (id) and a pointer to the next node (Figure 9.10.)..

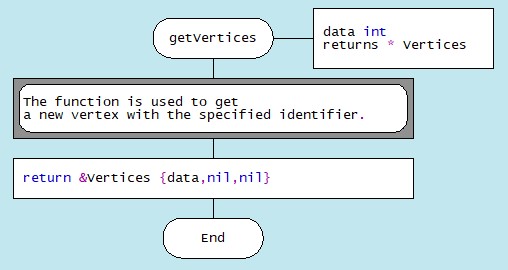
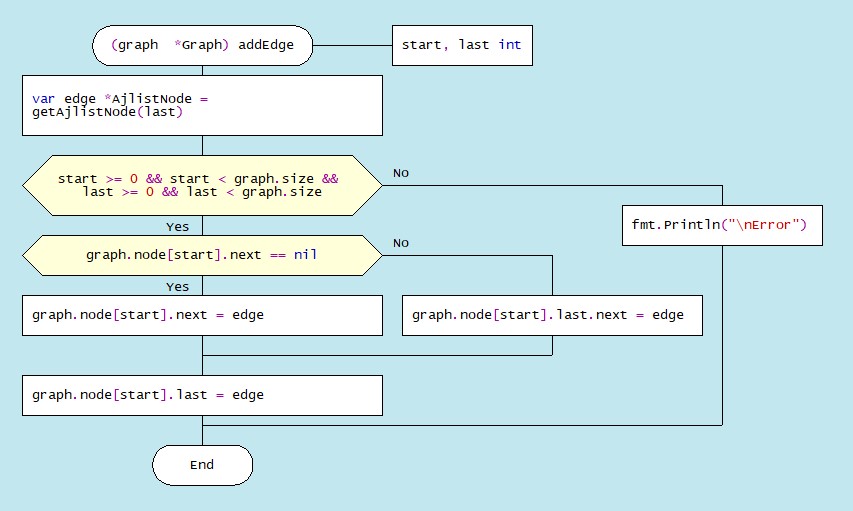
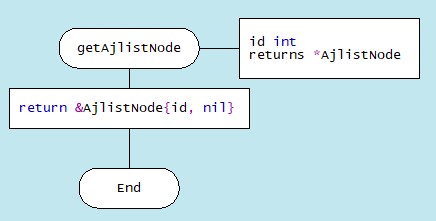


Figure 9.9. Drakon-diagrams of graph formation methods



1. Method addEdge((start, last int)



1. Method getAjlistNode (id int)

Figure 9.10. Drakon-diagrams of filling a graph with data

The printGraph() method allows you to the adjacency list of the graph, where each node is connected to its neighbors (Figure 9.11).

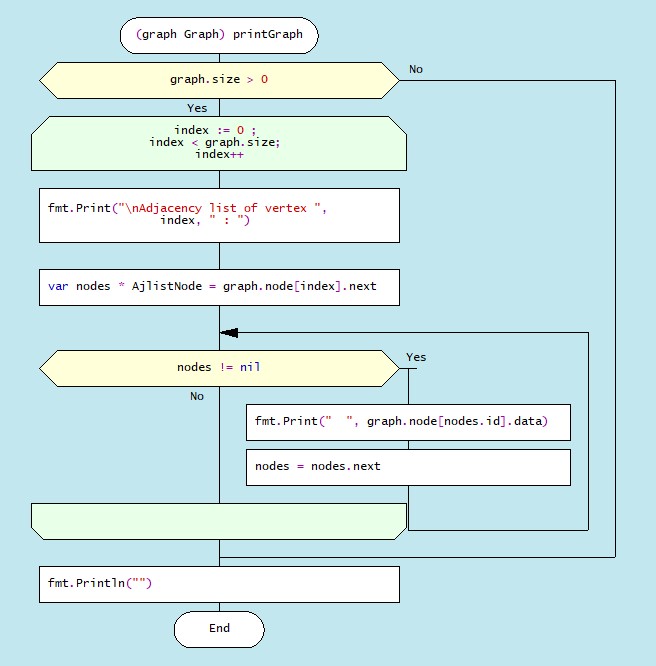


Figure 9.11. Drakon- diagram of method printGraph

9.4.1.2. Depth-first traversal algorithm

The depth-first traversal of the graph 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 oriented, the depth search builds a tree of paths from the starting vertex to all the tops accessible from it. The algorithm for traversing the graph in depth can be represented as follows. Suppose an observer in one of the vertices of a graph is given the task of traversing all its vertices. While at this vertex, the observer sees the edges emanating from that vertex. In the case of a sufficiently complex graph structure, the observer runs the risk of passing through some vertices several times and, in the end, getting stuck in a loop. To avoid such a situation, the observer should mark all the visited vertices and should not go to the peak that he has already visited. Then the algorithm might look like this:

* Select the vertex from which the graph traversal begins;
* Go to any adjacent vertex that has not been visited before;
* Run the depth-first traversal algorithm from this vertex;
* Return to the starting vertex; Repeat the process for all previously unvisited adjacent vertices.

Thus, in order to implement the algorithm, you will need to note which vertices the researcher was in and which were not. The mark will be made in the slice visit, where visit[i] == True for visited vertices, and visit[i] == false for unvisited vertices. The mark "about visiting a vertice" is put when entering this vertex.

A depth-first search starts by visiting the original vertex start, and then recursively visits all adjacent vertices. The visit to the vertex is recorded in the logical slice visit. The depth-first traversal algorithm is based on a recursive function that retrieves a vertex from the stack and checks it for attendance. If the vertex has already been visited, the traversal continues to search the adjacency list until it reaches the dead-end vertex. An illustration of a graph traversal in depth is presented Fig.9.12.

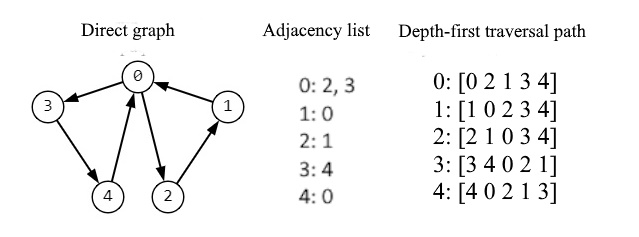


Figure 9.12. Illustration of depth-first traversal of a directed graph

The depth-traversal algorithm for undirected and directed graphs is broadly similar, but its execution may vary due to differences in the directionality of the edges and arrows in the graph. This difference lies in the fact that in undirected graphs it is possible to relate vertices in both directions without explicitly taking into account the direction. In directed graphs, each edge is counted only once in the direction from the start vertex to the end vertex. The dragon diagram of the algorithm for traversing the graph in depth is shown in Fig. 9.13.

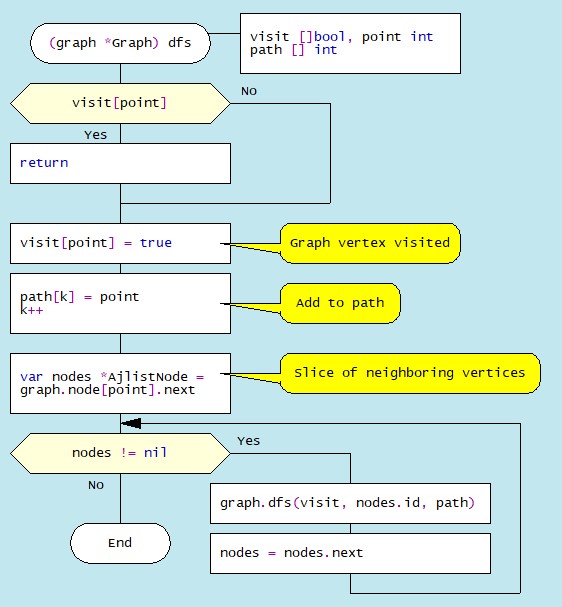


Figure 9.13. Depth-first traversal algorithm Drakon-diagram **dfs**

9.4.2. The breadth-first search algorithm

Graph breadth-first traversal is widely used in a variety of fields, such as computer networks, web search, image processing, and other areas that require analysis of the data structure represented as a graph. A breadth-first traversal visits all vertices that are on the same level, and then moves on to the next level, and so on. Recall that a graph level is a group of vertices that are at the same distance from the initial vertex. As an example, consider the algorithm for traversing a graph in the width of an undirected graph (Fig. 9.14),

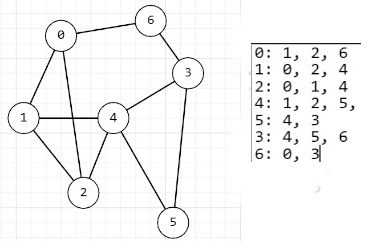


Figure 9.14. Undirected graph and Its adjacency list

In this case, unlike the previous depth-first traversal algorithm, to traverse the graph in width, we create an empty map [int] bool to track the visited vertices and an empty queue [] slice to add the visited vertices (Figure 9.15).

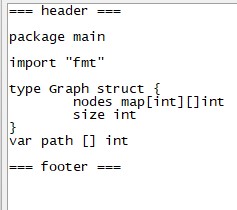
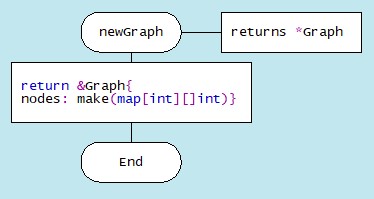
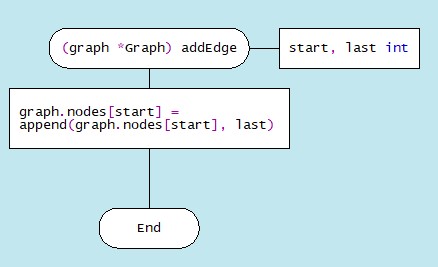


Figure 9.15. Description of variable types

The graph is generated using the NewGraph and addEdge(start, last) methods (Figure. 9.16).



1. Method newGraph



1. Method addEdge(start, last int)

Figure 9.16. Graph generation algorithms

In the first step of the algorithm, the start vertex is added to the queue. At each width traversal step, a vertex is retrieved from the top of the queue. If this vertex has already been visited (is in the visited state), the program moves on to the next iteration of the loop, interrupting the current iteration. If the summit has not yet been visited, it is displayed and marked as visited. Then, each neighbor of that vertex that has not yet been visited is added to the queue Then the traversal continues until the queue is empty. If the queue is empty, the algorithm ends. The Drakon-diagram of the graph traversal algorithm is shown in Fig. 9.17.

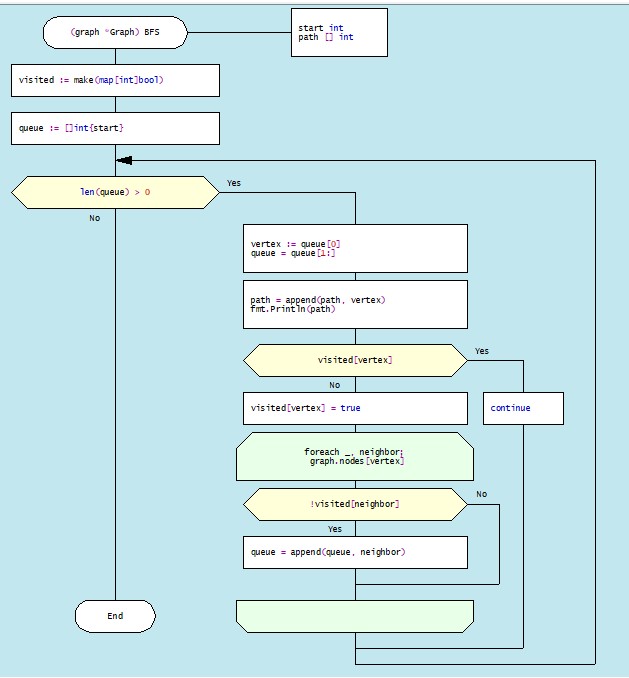


Figure 9.17. DRAKON-diagram of Breadth-first traversal algorithm

The results of traversal of the directed graph in depth and width are shown in Figure 9.18.

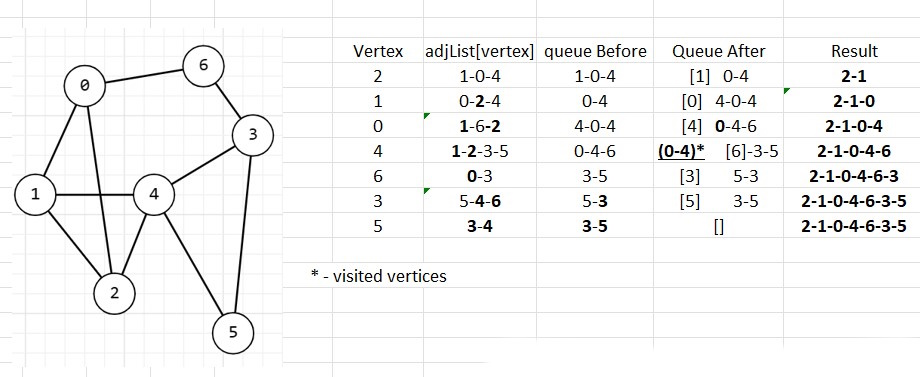


Figure 9.18. The process of forming a graph traversal path

**9.4.3. Removing a vertex from a graph**

A vertex is removed from the graph by copying all vertices up to the vertex to be deleted, then the vertex to be deleted is skipped, and then the remaining vertices are copied. The dragon diagram of the vertex removal algorithm is shown in Fig. 9.19.

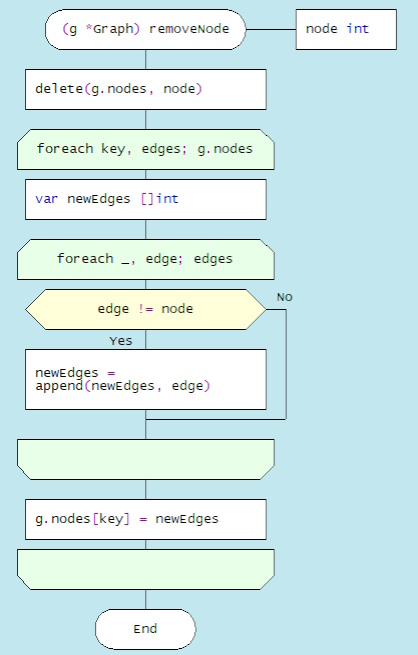
****

Figure 9.19. Drakon-diagram of the algorithm for removing a vertex from a graph

**9.5. 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. Let's set the task as follows: in order to determine the optimal path, find the distances of all possible paths between two geographical points, as well as the total cost of travel, taking into account the different costs 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 9.20).

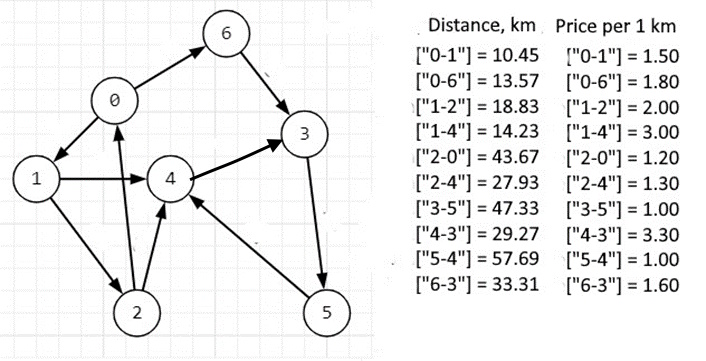


Figure 9.20. Weighted directed graph with source data

Creating a new instance of the graph ( the newGraph method), adding edge information (the addEdge method), and calling the findPaths method (the allPaths method) is shown in Figure 9.21.

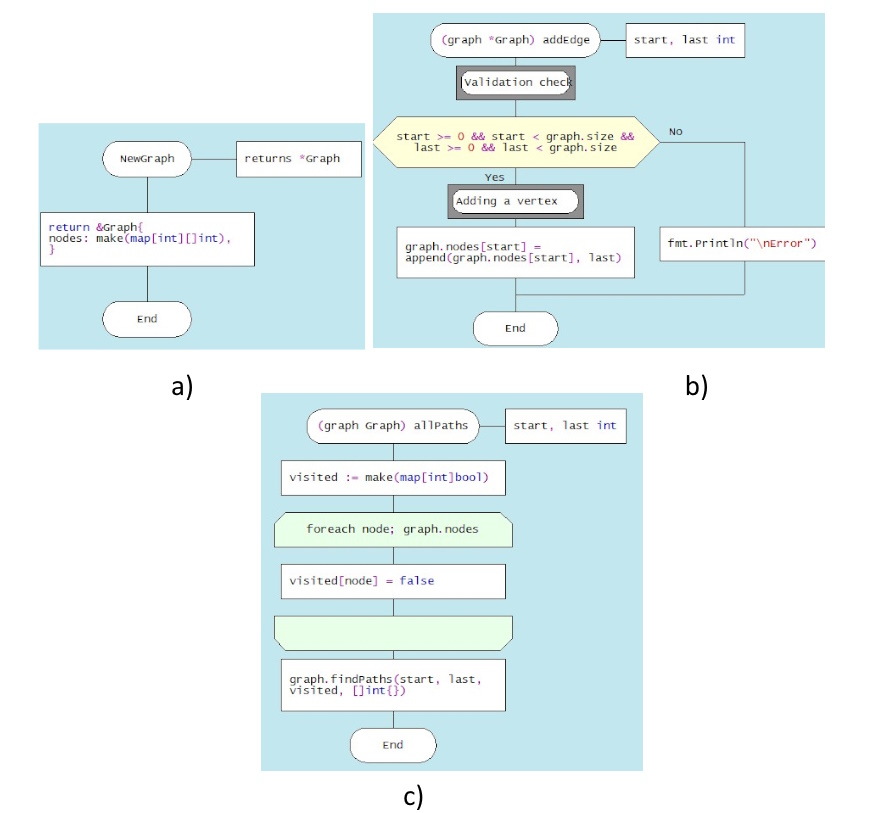


Figure 9.21. Drakon-diagrams of methods a). newGaph, b). addEdge, c).allPaths

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 9.21. 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” (Fig. 9.22). 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.

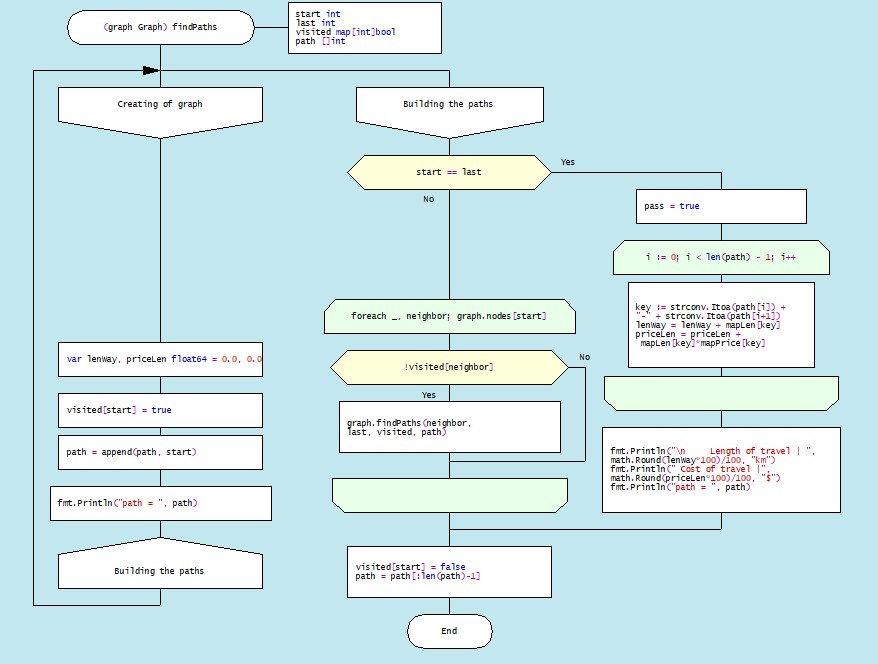


Figure 9.22. DRAKON-diagram of method findPath

Consider the problem of finding all possible paths between two vertices of the graph (2) and (5) shown in Figure 9.23.:

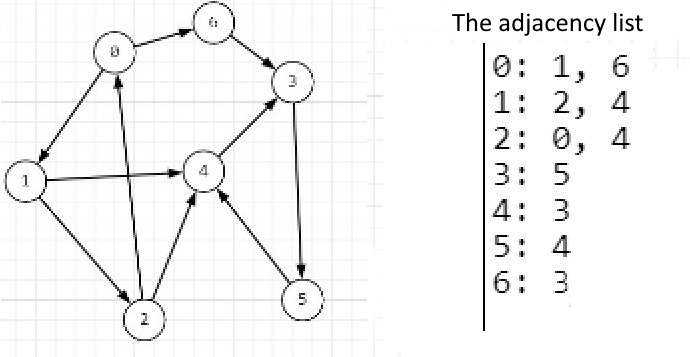


Figure 9.23. Graph and the adjacency list

All the paths of the directed graph between vertices (2) and (5) are shown in Fig. 9.24.

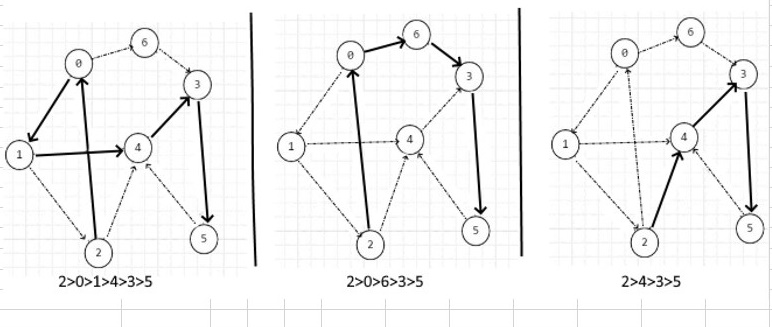


Figure 9.24 Possible paths to traversal of vertices of a directed graph

The results of passing through the vertices in algorithm implementation process is shown in Table 9.2.

Table.9.2. Implementation process findPath (first traversal)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Vertices | | Neighbors | Path | Visited vertices |
| start ≠ last | | | | |
| 2 | | (2, 4) | 2 | TRUE |
| 0 | | (1, 6) | 2 0 | TRUE |
| 1 | | (2,4) | 2 0 1 | TRUE |
| 4 | | (3) | 2 0 1 4 | TRUE |
| 3 | | (5) | 2 0 1 4 3 | TRUE |
| 5 | | (4) | 2 0 1 4 3 5 | TRUE |
| start = last à Recursion à (path = path[:len(path) – 1] | | | | | | | |
| Vertices | Path | | Visited vertices (true); Unvisited vertices (false) | | | | |
| 5 | 2 0 1 4 3 5 | | 0:true 1:true 2:true 3:true 4:true 5:true **6:false** | | | | |
| 3 | 2 0 1 4 3 | | 0:true 1:true 2:true 3:true 4:true **5:false 6:false** | | | | |
| 4 | 2 0 1 4 | | 0:true 1:true 2:true **3:false** 4:true **5:false 6:false** | | | | |
| 1 | 2 0 1 | | 0:true 1:true 2:true **3:false 4:false 5:false 6:false** | | | | |
| 0 | 2 0 | | 0:true **1:false** 2:true **3:false 4:false 5:false 6:false** | | | | |

The results of the comparison of fares from the top (2) and (5) are shown in Table 9.3.

Table 9.3. Distance and fare

|  |  |  |
| --- | --- | --- |
| Travel | Length of travel, km | Cost of travel, $ |
|  |
| 2-0-1-4-3-5 | 134.5 | 239.02 |  |
| 2-0-6-3-5 | 124.31 | 153.03 |  |
| 2-4-3-5 | 104.53 | 180.23 |  |

Table 9.3 shows that the second trip (2-0-6-3-5) is the most cost-effective.

**9.6. Dijkstra's algorithm – a method for finding the shortest path**

The Dijkstra algorithm is designed to find the shortest paths from a given vertex to all other vertices, provided that the edges of the graph are not negative. The algorithm was named after the Dutch scientist E. W. Dijkstra in 1956. This algorithm is widely used in various fields of science, technology and social life. In GPS systems, an algorithm is used to find the fastest route to a destination. In telecommunications networks, it is used to determine the optimal path for data to be transmitted from source to receiver. In robotics, an algorithm can be used to plan a robot's path to reach a goal in the most efficient way. In computer games, to determine the path of characters or objects. In network planning, an algorithm can be used to find the optimal delivery path for goods.

To implement Dijkstra's algorithm, vertex structures of type Vertex are created to describe vertices and Graph to describe the graph (Fig. 9.25)

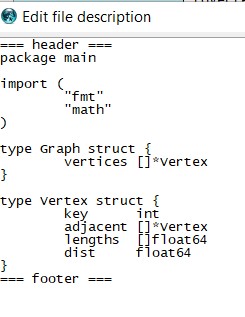
****

Figure. 9.25 Description of vertex and graph structures

Here, key is the key of the vertex of the graph; adjacent – adjacency list for the vertex; **lengths** - a set of edge lengths to vertices from the adjacency list; **dist** is the total length of the path.

Next, an instance of the graph is created in the main program and vertices are instantiated in the loop (Fig. 9.26):

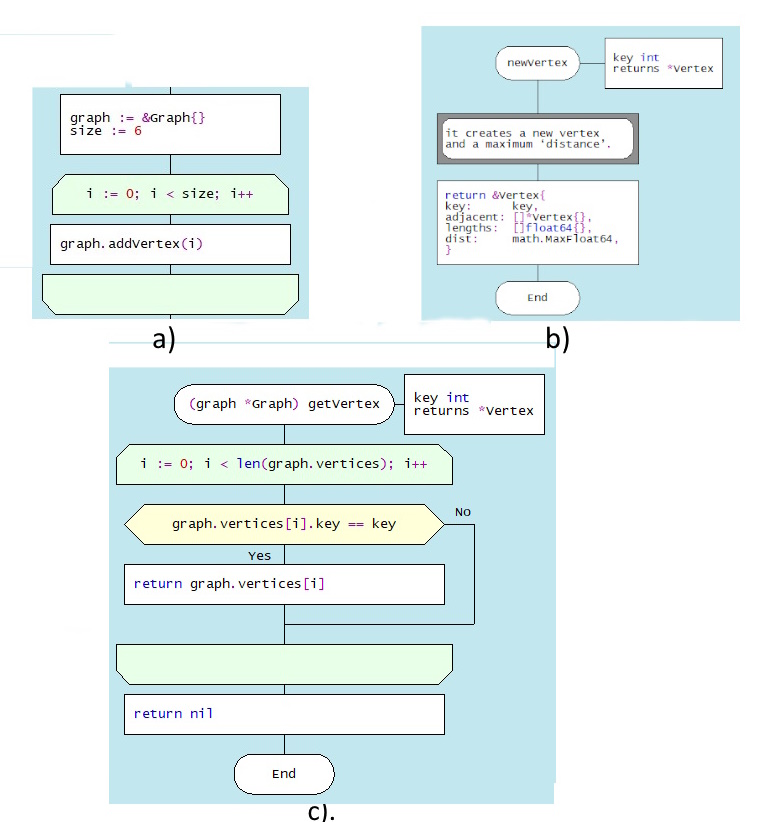


Figure 9.2**6**. Drakon-diagrams of graph vertex formation methods

a). instantiating the graph and vertex templates

b). creating a New Vertex

c). populating the vertex with data

After creating the vertices of the graph and populating them with data (keys, neighboring vertices, and distances), edges are formed using the addEdge method (Figure 9.27):

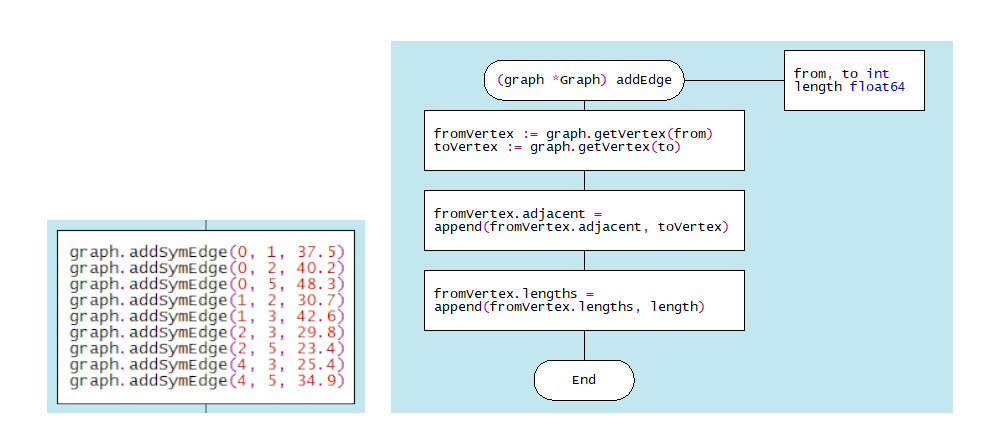


Figure 9.27. Дракон-диаграммы метода формирования ребер графа addEdge

The algorithm of the addEdge(from, to, length) method is implemented as follows. First, the method gets two vertices, between which you need to add an edge (graph.getVertex(from), graph.getVertex(to)), where from and to are vertex keys. The method then adds the toVertex vertex to the list of adjacent vertices for the vertex fromVertex (append(fromVertex.adjacent, toVertex)). As a result, an edge is formed between these two vertices fromVertex and toVertex. In the last step, the method adds the edge length to the list of edge lengths for the vertex fromVertex (append(fromVertex.lengths, length)). This means that the length of the edge between the vertices fromVertex and toVertex is now known.

Note that the main program uses the addSymEdge (u,v,L) method, which creates symmetric data for each edge in the undirected graph (Figure 9.28):

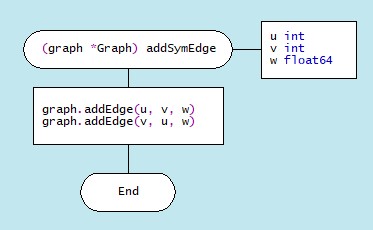


Figure 9.28. Drakon-diagram of the addSymEdge edge symmetrization method

Next, we call the Dijkstra method (startKey int), where startKey int is the initial vertex, and output the final results (Fig. 9.29):

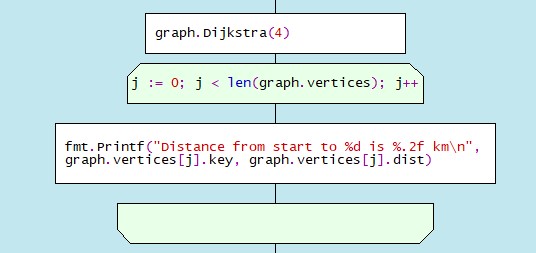


Figure Fig.9.29. Drakon-diagram of the Deikstra method and output of the results

The dragon diagram of the Dijkstra(startkey) method is shown in Fig. 9.30.

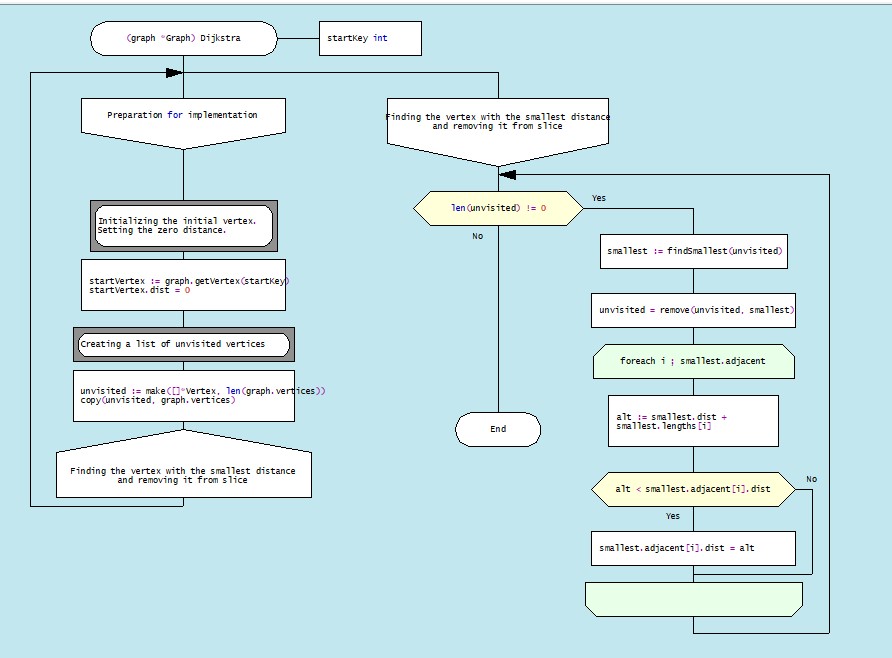


Figure 9.30 Drakon-diagram of method Dijkstra

Let's take a look at the basic steps of implementing this algorithm. First, initialization (startVertex := graph.getVertex(startKey); startVertex.dist = 0) takes place, which results in getting the initial vertex from the key and setting its distance to 0. In the second step, a list of unvisited vertices is created, which is then copied with the copy command in order to be able to manipulate this list during the processing of vertex data without changing the graph itself.

Next, a loop for len(unvisited) != 0 is performed to traverse all unvisited vertices in order to find the vertex with the shortest distance using the method smallest := findSmallest(unvisited) (Fig. 9. 31.)

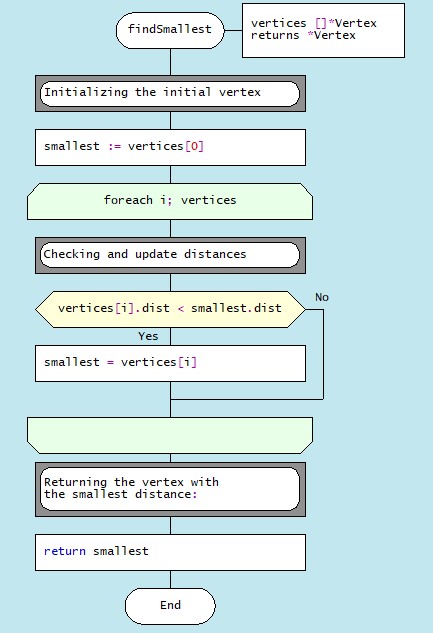


Figure. 9.31. Drakon-diagram of mrthod findSmallest(unvisited)

The **current vertex is then removed from the list of unvisited vertices (**unvisited = remove(unvisited, smallest)). Finally, in a loop across all vertices, the distances to adjacent vertices are updated as a result of the shorter path being detected.

As an example, consider the application of Dijkstra's algorithm to finding the shortest paths from a given vertex for the undirected graph shown in Fig. 9.32.

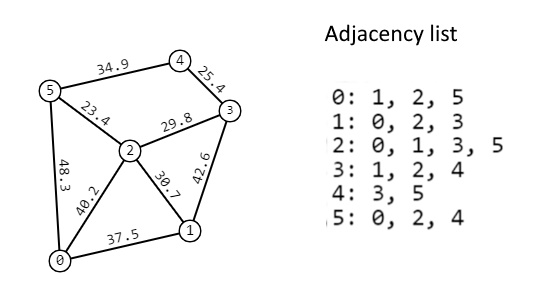


Figure .9.32. Undirected loaded graph

The results of calculations of the shortest paths from vertex 0 to the rest of the vertices of the graph (Fig. 9.33.):

Distance from start to 0 is 0.00 km

Distance from start to 1 is 37.50 km

Distance from start to 2 is 40.20 km

Distance from start to 3 is 70.00 km

Distance from start to 4 is 83.20 km

Distance from start to 5 is 48.30 km

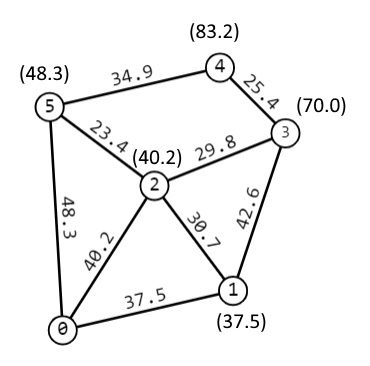


Figure 9.33. Shortest distances from vertex 0 are given in parentheses

And, finally, about estimating the complexity of Dijkstra's algorithm. This estimate can vary depending on the data structure used to represent the graph. In the case of a Linked List, the time complexity of Dijkstra's algorithm is estimated to be O(V^2), where V is the number of vertices in the graph. This is because you have to look at all vertices each time you retrieve a minimal item from the priority queue.

When using the map data structure, the time complexity of Dijkstra's algorithm is O((V+E) log V), where E is the number of edges in the graph. This is because it takes O(log V) time to retrieve the minimum element from the priority queue implemented with map.

In both cases, the spatial complexity will be O(V + E) because you need to store all the vertices and edges of the graph. However, it's worth noting that a map usually takes up more space than a linked list because of the additional information it stores (keys and values).