**РАЗДЕЛ 9. ГРАФЫ**

**9.1. Общие сведения. Базовая терминология**

В этом разделе рассматриваются такие структуры данных как *графы*. Граф в математике представляет собой структуру, состоящую из вершин (узлов) и рёбер (связей), которые соединяют эти вершины. Вообще говоря, графы имеют много общего с *деревьями*, - можно сказать, что деревья являются частным случаем графа. Однако их практическая ценность при решении многих реальных задач намного выше. Многие задачи сводятся к рассмотрению совокупности объектов, свойства которых описываются связями между ними. В числе таких объектов – электрические и электронные схемы, печатные платы, карты дорог, авиационные маршруты, описание конструкций, игры, а в числе задач - поиск кратчайшего пути из одной вершины до другой, решение задачи максимальной пропускной способности трубопровода или дорожной сети или компьютерной сети, распределение N работников для выполнения M различных типов работ, выбор наиболее эффективного метода решения задачи и т.д. [[Задачи, которые можно решить с помощью графов | Вики справка Graph Online](https://graphonline.ru/wiki/Статьи/ЗадачиКоторыеМожноРешитьСПомощьюГрафов)]

Более строго, граф G задан множеством вершин {V} и множеством рёбер {E}, соединяющих все или часть этих вершин. Таким образом, граф G полностью определяется как {V, E}. Если рёбра ориентированы, то они называются дугами, а граф с такими рёбрами называется ориентированным графом (рисунок 9.1 a). Если рёбра не имеют ориентации, то граф называется неориентированным:

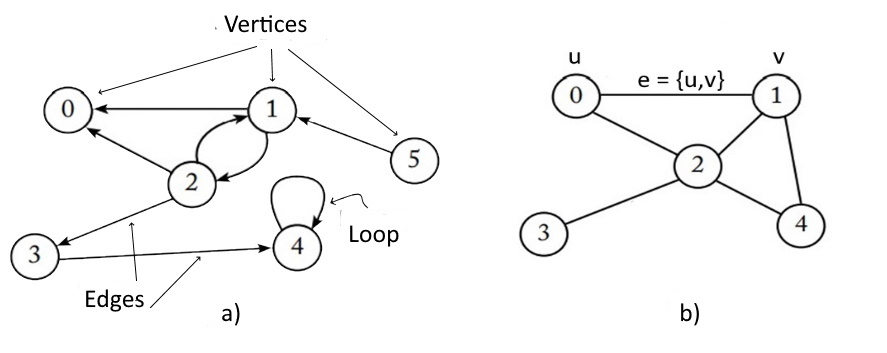


Рис. 9.1 Виды графов а) - неориентированный; б) – направленный

Вершины и рёбра называются элементами графа, число вершин в графе является порядком, число рёбер - размером графа. Вершины (u,v) называются конечными точками e = {u, v}, а две конечные точки одного ребра называются смежными. Рёбра называются смежными, если они имеют общую конечную точку. Рёбра называются кратными, если наборы конечных точек одинаковы. Ребро называется петлей, если его концы совпадают, то есть e = {u, u}. Если вершина является началом или концом ребра, то они (вершина и ребро) инцидентны. Число рёбер, инцидентных вершине, называется степенью вершины (рис. 9.2).



Рис. 9.2. Основные параметры графа

**9.2. Методы представления графов**

Решение задач, связанных с обработкой совокупности данных, организованных в виде графов, требует их моделирования в компьютерных программах. В принципе совокупность вершин можно хранить в массиве и обращаться к ним по индексу. Хранение вершин также можно организовать с помощью односвязного списка или другой структури данных. На практике для моделирования структуры графа обычно применяются две структуры: *матрица смежности* и *список смежности*. Смежными в том смысле, что такие вершины соединены одним ребром.

Матрица смежности представляет собой двумерный массив, элементы которого обозначают наличие связи между двумя вершинами. Если граф содержит V вершин, то матрица смежности представляет собой массив V × V. На рис. 9.3. показан неориентированный и ориентированный графы и соответствующие матрицы смежности. В частности, для ориентированного графа матрица смежности строится таким образом, что 1 обозначает наличие связи между вершинами, 0 – ее отсутствие. Так, для вершины (2) смежными вершинами являются вершины (1) и (3), а для вершины (5) – вершина (2).

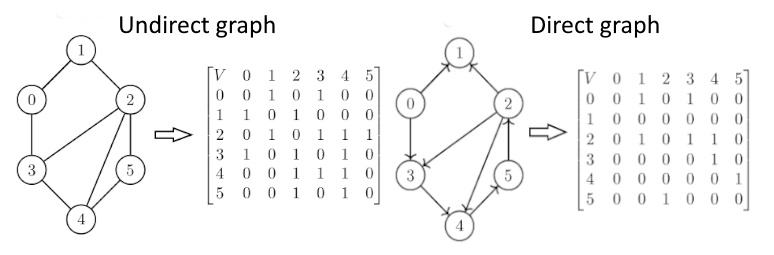


Рис. 9.3. Представление графа в матрице смежности

Представление графа в матрице смежностей связано с определенными проблемами. Прежде всего при построении матрицы нужно заранее знать количество вершин в графе, что приводит к необходимости каждый раз строить новую матрицу при внесении новых вершин. Кроме того, матрица смежностей состоит в основном из нулей, что приводит к неэффективному использованию памяти, если граф содержит *V* вершин, то должна быть отведена память для V2 элементов.

Во многих случаях более эффективным способом представления графа является использование списка смежностей. Неориентированный и ориентированный графы и их списки смежности изображены на рисунке 9.4.

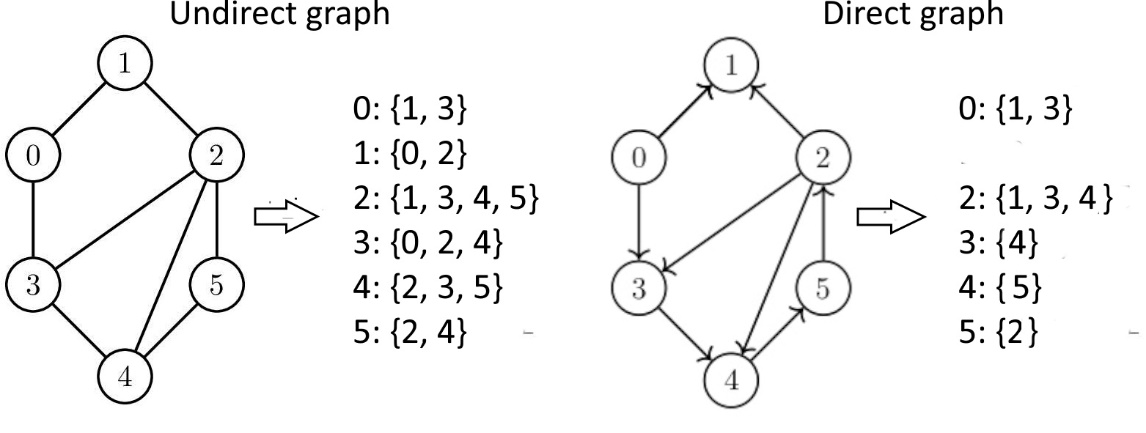


Рис. 9.4. Представление графа в списке смежностей

Матрица и список смежности для ненаправленного графа отличаются от направленного тем, что метки ‘1’ в матрице проставляются для всех смежных вершин (рис.9.4.).

В большинстве практических случаев задачи обработки графов содержат значения, которые связаны с либо с вершинами графа, либо с его ребрами. Например, в задаче поиска оптимального пути между двумя пунктами, ребра графа нужно нагрузить такими данными как расстояние между этими вершинами и стоимостью проезда за 1 км (Рис. 9.5):

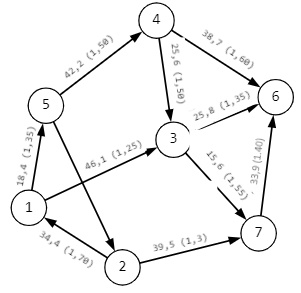


Рис. 9.5. Нагруженный граф

**9.3. Интерфейсы графов**

Интерфейс графа как структуры данных представляет собой набор методов, которые определяют базовые операции, доступные для работы с графом. Этот интерфейс включает в себя методы для получения количества вершин и рёбер в графе, доступа к конкретному ребру между вершинами, добавления и удаления вершин и рёбер, а также методы для работы с итераторами, позволяющие получить информацию о входящих и исходящих рёбрах для конкретной вершины. Интерфейс графа позволяет работать с графовыми структурами данных и реализовывать их в соответствии с требованиями конкретной задачи.

В языке Golang интерфейс графа может быть реализован с использованием структур и методов, содержание которых определяется конкретной задачей и выбором базовых структур данных, из которых будут создаваться новые типы. В простейшем случае для работы с графами требуется включение в структуру графа полей для отображения вершин и/или ребер. Для создания типа Graph с поддержкой представления графа с помощью списка смежностей используют различные базовые структуры языка Golang (срезы, элементами которых являются односвязные списки (Linked List), карты (map), двумерный срез [] [].

Покажем примеры построения ориентированного графа, приведенного на рис. 9.6., с помощью указанных выше базовых структур (Табл. 9.1.).

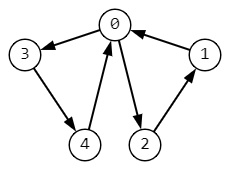


Рис. 9.6. Ориентированный граф для иллюстрации интерфейса

При выборе способа реализации интерфейса графа средствами языка Golang следует руководствоваться следующими критериями:

1. Linked List:

- Используется, если необходимо хранить связи между вершинами графа в виде списка смежности.

- Подходит для реализации графов с большим количеством ребер и небольшим количеством вершин.

- Позволяет эффективно добавлять и удалять ребра между вершинами.

2. Map:

- Используется, если необходимо хранить связи между вершинами графа в виде словаря, где ключами являются вершины, а значениями - их соседи.

- Подходит для реализации графов с произвольным количеством вершин и ребер.

- Позволяет быстро получать список соседей для заданной вершины.

3. Slice:

- Используется, если необходимо хранить связи между вершинами графа в виде срезов, где индексы срезов соответствуют вершинам, а значения - их соседи.

- Подходит для реализации графов с фиксированным количеством вершин.

- Позволяет эффективно получать список соседей для заданной вершины.

Выбор конкретного способа реализации зависит от конкретных требований к графу, таких как количество вершин и ребер, операции, которые чаще всего будут выполняться (добавление вершин, добавление ребер, поиск соседей и т.д.), а также от особенностей самой задачи, в которой будет использоваться граф.

|  |  |  |
| --- | --- | --- |
| Табл. 9.1. Базовые структуры создания интерфейсов графа | | |
| 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.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(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. Базовые** а**лгоритмы графов**

Вследствие широкого распространения графов существует большое количество алгоритмов их обработки. Среди задач, решаемых в рамках теории графов, таковыми являются:

* Определение графа и его свойства;
* Действия с графами;
* Маршруты, цепи и циклы, контуры**;**
* Вычисление характеристик графа;

Рассмотрим основные алгоритмы, используемые в этих задачах и реализованные, как и прежде, в рамках гибридного программирования DRAKON + Golang. После того, как граф построен и его свойства проверены, естественно может возникнуть задача обхода графа по его вершинам. Общая задача формулируется следующим образом: обойти граф, начиная с заданной вершины и двигаясь по ребрам к другим вершинам, посетить все вершины.

Существует два основных способа обхода графов: обход по глубине и обход по ширине, которые обеспечивают перебор всех связных вершин. Разница между поиском в глубину и поиском в ширину заключается в том, что результатом алгоритма поиска в глубину является некоторый маршрут, по которому можно последовательно обойти все вершины графа, доступные из начальной вершины. Это принципиально отличается от поиска в ширину, где вначале обходятся все вершины одного уровня, а затем последовательно обходятся вершины следующих уровней.

**9.4.1. Обход в глубину (In depth)**

9.4.1.1 Формирование графа

Напомним, что алгоритм поиска в глубину заключается в последовательном обходе вершин графа, доступных из начальной вершины. Для реализации этого алгоритма граф будем представлять с помощью связных списков (Табл. 9.1). Описание типов переменных, используемых в программе обработки графа представлены на рис. 9.7.

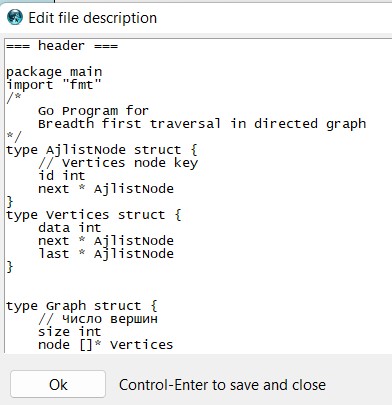


Рис. 9.7. Описание типов переменных программы обхода в глубину

В этом фрагменте описываются следующие структуры:

1. AjlistNode - структура, которая представляет узел в связанном списке для представления смежности графа. Она содержит следующие поля:

- id int - целочисленное значение, которое представляет идентификатор вершины графа.

- next \*AjlistNode - указатель на следующий узел в связанном списке.

2. Vertices - структура данных, которая представляет вершину графа в программе обработки графа. Она содержит следующие поля:

- data int - это целочисленное значение, которое представляет данные, связанные с вершиной графа.

- next \*AjlistNode - это указатель на первый узел в связанном списке смежных вершин.

- last \*AjlistNode - это указатель на последний узел в связанном списке смежных вершин.

3. Graph – структура описывает граф, состоящий из узлов типа Verticesи имеющий два поля:

- size представляет количество узлов в графе (размер графа);

- node представляет собой срез указателей на эти узлы.

На рис.9.8. представлена полная Дракон-диаграмма алгоритма обхода графа в глубину:

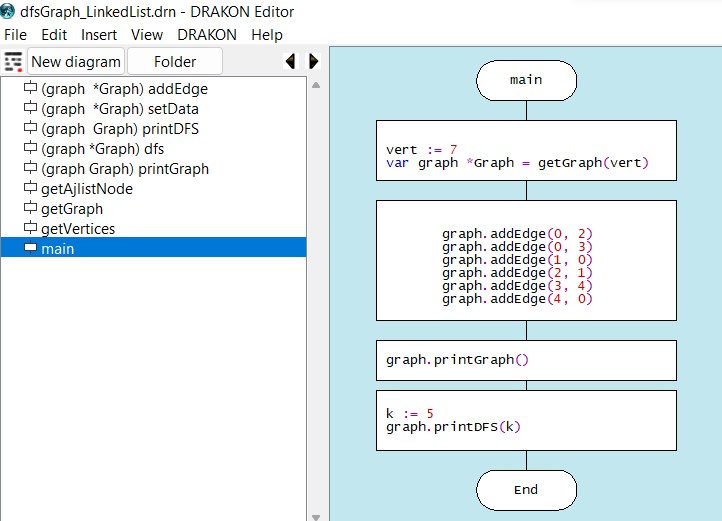
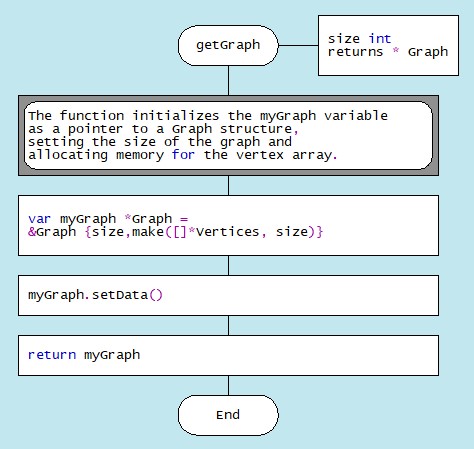
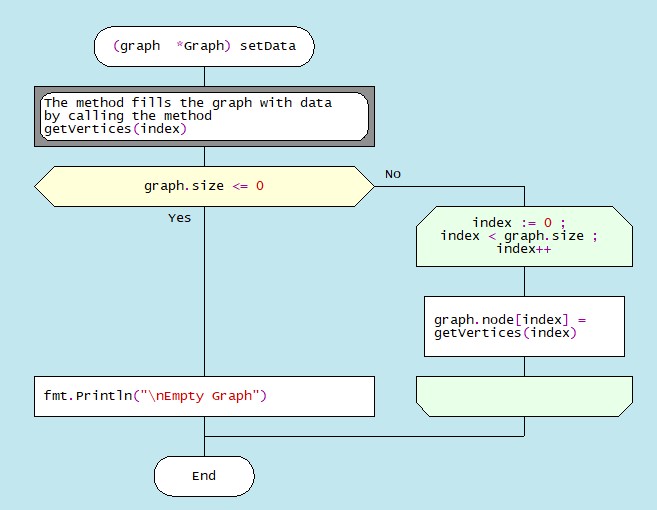


Рис. 9.8. Дракон-диаграмма функции main()

В функции *main ()* создается переменная graph типа указателя \*Graph, описывающая граф, состоящий из структур данных типа Vertices, где size представляет количество узлов в графе, а node представляет собой срез указателей на эти узлы (Рис. 9.9.).

После создания нового экземпляра графа выполняется, метод addEdge(start, last), который добавляет ребро между узлами start и last. Для создания узла для списка смежности в методе используется переменная var edge \*AjlistNode, инициализирующая уникальный идентификатор (id) и указатель на следующий узел (next).

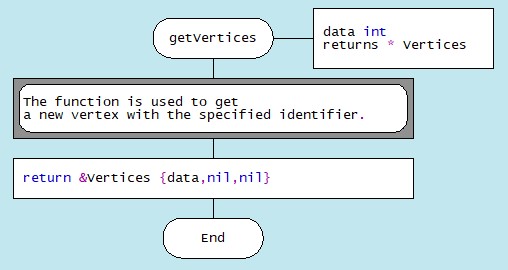
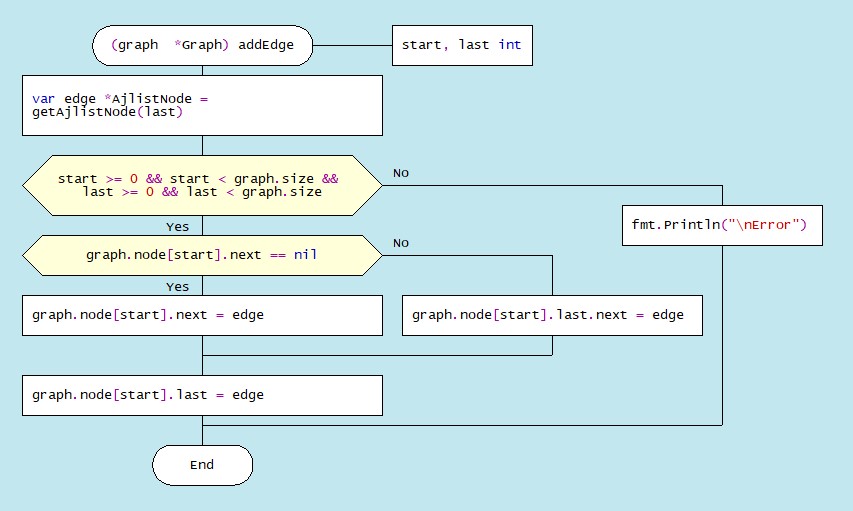
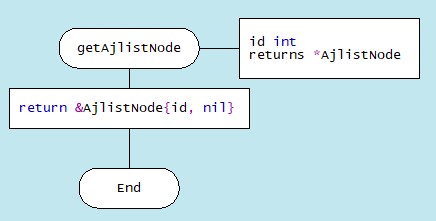


Рис. 9.9. Дракон-диаграммы методов формирования графа



1. метод addEdge((start, last int)



б) метод getAjlistNode (id int)

Рис. 9.10. Дракон-диаграммы заполнения графа данными

Метод printGraph()позволяет вывести список смежности графа, где каждый узел связан со своими соседями (рис. 9.11).

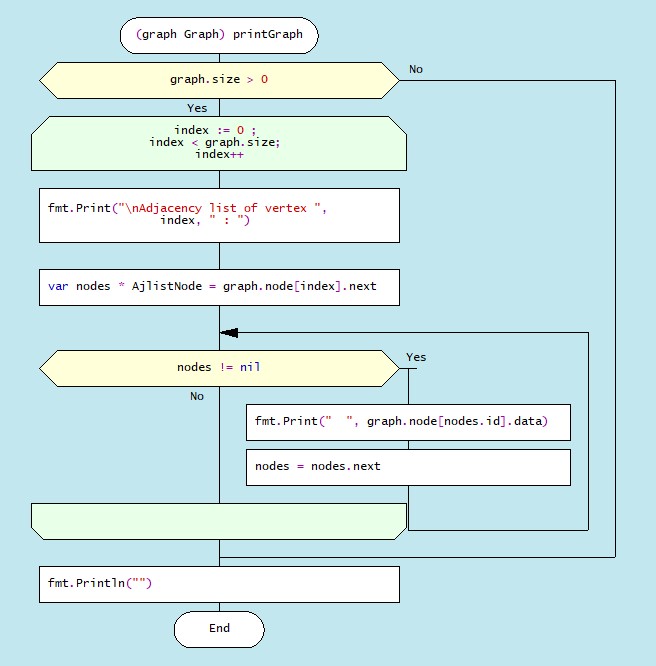


Рис. 9.11. Дракон-диаграмма метода printGraph

9.4.1.2. Алгоритм обхода в глубину (DFS)

Обход графа в глубину является оправданным в ситуациях, когда необходимо исследовать неизвестную структуру реального объекта, представленного в виде графа. Если же граф ориентированный, то поиск в глубину строит дерево путей из начальной вершины во все доступные из нее верщины. Алгоритма обхода графа в глубину можно представить себе следующим образом. Пусть наблюдателю, находящемуся в одной из вершин графа, поставлена задача обойти все его вершины. Находясь в этой вершине, наблюдатель видит ребра, исходящие из этой вершины. В случае достаточно сложной структуры графа наблюдатель рискует проходить некоторые вершины по нескольку раз и, в конце концов, зациклиться. Для избежания такой ситуации наблюдатель должен отмечать все посещенные вершины и не должен идти в ту вершину, которую он уже посещал. Тогда алгоритм может выглядеть следующим образом:

* Выбрать вершину, с которой начинается обход графа;
* Перейти в любую смежную вершину, не посещенную ранее;
* Запустить из этой вершины алгоритм обхода в глубину;
* Вернуться в начальную вершину;
* Повторить процесс для всех не посещенных ранее смежных вершин.

Таким образом для реализации алгоритма понадобится отмечать, в каких вершинах был исследователь, а в каких — нет. Пометку будем делать в списке visit, где visit[i] == True для посещенных вершин, и visit[i] == false для непосещенных. Пометка «о посещении вершины» ставится при заходе в эту вершину.

Поиск в глубину начинается с посещения исходной вершины start, после чего рекурсивно посещаются все смежные вершины. Посещение вершины фиксируется в логическом срезе visit. Алгоритм обхода в глубину основан на применении рекурсивной функции, которая извлекает из стека вершину, проверяя ее на посещаемость. Если вершина уже посещалась, обход продолжает поиск по списку смежности до достижения тупиковой вершины. Иллюстрация обхода графа в глубину представлена на рис.9.12.

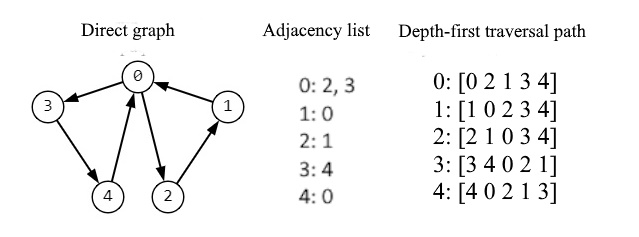


Рис. 9.12. Иллюстрация обхода ориентированного графа в глубину

Алгоритм обхода в глубину в ненаправленных и направленных графах в целом аналогичен, но его выполнение может меняться из-за различий в направленности рёбер и стрелок в графе. Это различие состоит в том, что в ненаправленных графах возможна связь между вершинами в обоих направлениях без явного учёта направления. В направленных графах каждое ребро учитывается только один раз в направлении от начальной вершины к конечной. Дракон-диаграмма алгоритма обхода графа в глубину представлена на рис. 9.13.

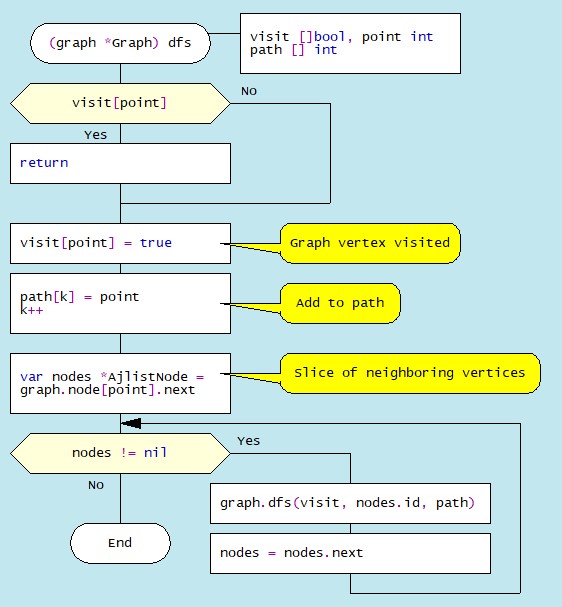


Рис. 9.13. Дракон-диаграмма алгоритма обхода в глубину dfs

**9.4.2. Обход графа в ширину**

Обход графа по ширине широко применяется в различных областях, таких как компьютерные сети, поиск веб-страниц, обработка изображений и других областях, где требуется анализ структуры данных, представляемой в виде графа. При обходе в ширину сначала посещаются все вершины, расположенные на одном уровне, а затем происходит переход к следующему уровню и так далее. Напомним, что уровень графа – это группа вершин, находящихся на одинаковом расстоянии от начальной вершины. В качестве примера рассмотрим алгоритм обхода графа в ширину неориентированного графа (рис. 9.14),

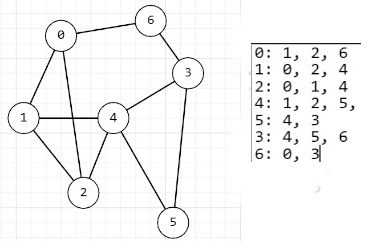


Рис. 9.14. Неориентированный граф и его список смежности

В данном случае, в отличие от предыдущего алгоритма обхода в глубину, для обхода графа в ширину создадим пустую карту map [int] bool для отслеживания посещенных вершин и пустой срез queue [] для добавления посещенных вершин (рис. 9.15).

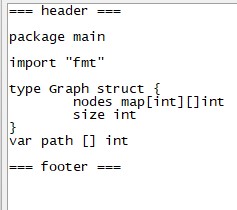
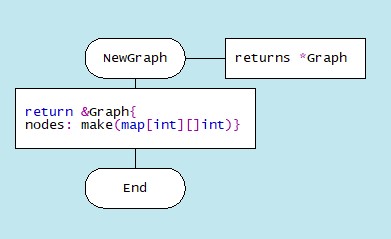
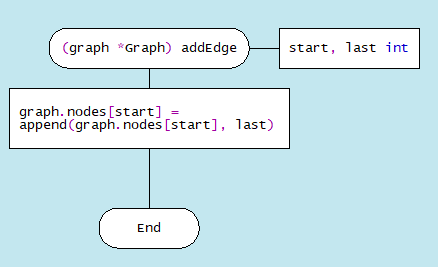


Рис. 9.15. Описание типов переменных

Формирование графа осуществляется методами NewGraph и addEdge(start, last) (рис. 9.16)



1. Метод newGraph



1. Метод addEdge(start, last int)

Рис. 9.16. Алгоритмы формирования графа

На первом шаге алгоритма начальная вершина (**start**) добавляется в очередь queue. На каждом шаге обхода графа по ширине извлекается вершина из начала очереди. Если эта вершина уже посещалась (находится в состоянии visited), программа переходит к следующей итерации цикла, прерывая текущую итерацию. Если вершина еще не посещалась, то она выводится и помечается как посещенная. Затем, каждый сосед (neighbor) этой вершины, который еще не посещен, добавляется в очередь Затем обход продолжается до тех пор, пока очередь не станет пустой. Если очередь пуста, алгоритм завершается. Дракон-диаграмма алгоритма обхода графа по ширине представлена на рис. 9.17.

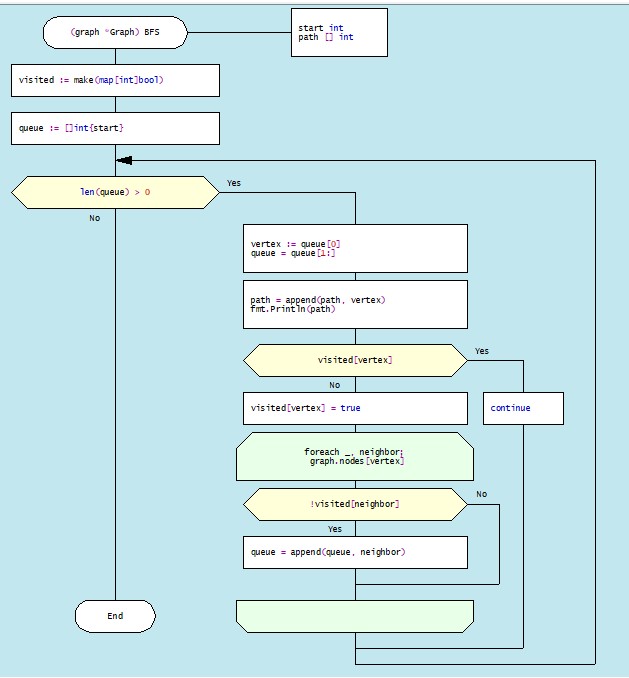


Рис. 9.17. Дракон-диаграмма алгоритма обхода в ширину BFS

Результаты обхода направленного графа в глубину и ширину представлены на рис. 9.18.

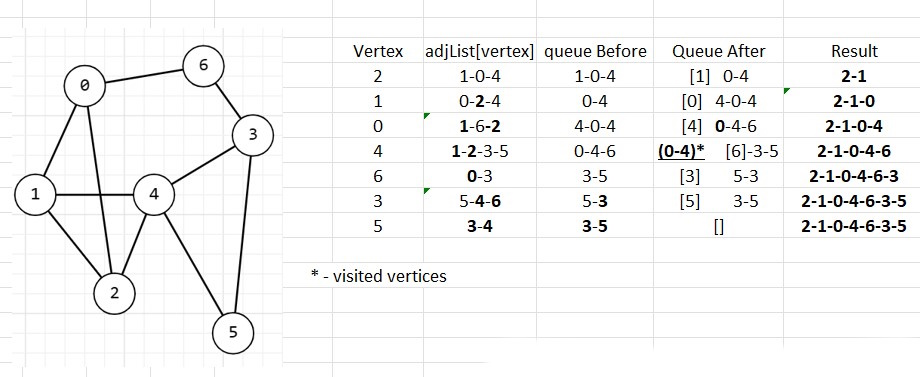


Рис. 9.18. Процесс формирования пути обхода графа по ширине

**9.4.3. Удаление вершины из графа**

Удаление вершины из графа происходит путем копирования всех вершин до удаляемой вершины, затем удаляемая вершина пропускается, после чего копирование оставшихся вершин продолжается. Дракон-диаграмма алгоритма удаления вершины показана на рис. 9.19..

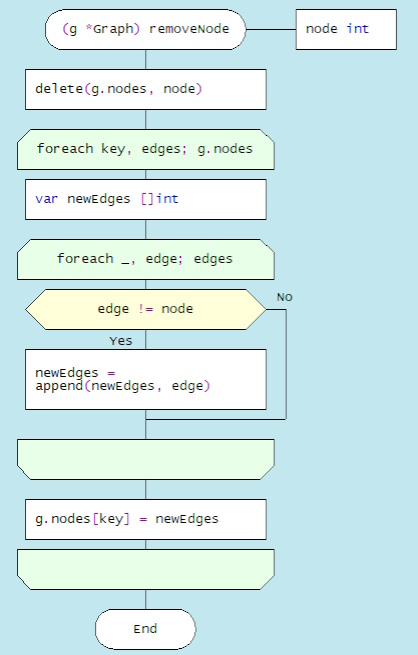


Рис. 9.19. Дракон-диаграмма алгоритма удаления вершины из графа

**9.5. Выбор пути между вершинами в направленном графе**

Выбор пути между двумя вершинами в направленном графе, имеющий практическое значение в смысле поиска оптимального пути между двумя населенными пунктами, соединенными дорогами с односторонним движением, может осуществляться по различным критериям. Поставим задачу следующим образом: с целью определения оптимального пути найти расстояния всех возможных путей между двумя географическими пунктами, а также суммарную стоимость проезда с учетом различной стоимости за 1 км на различных участка дороги.

Для решения этой задачи необходимо создать нагруженный граф, конкретнее, нагрузить ребра между всеми вершинами значениями расстояний между соседними пунктами и ценой проезда за 1 километр (рис. 9.20).

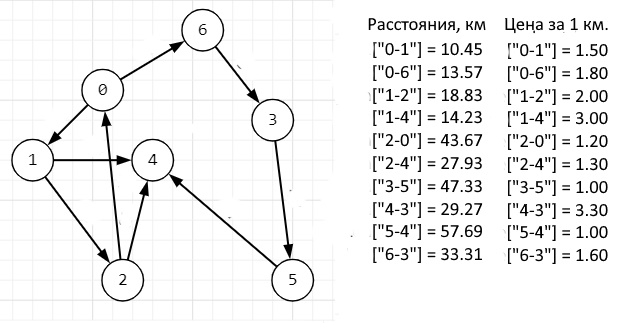


Рис. 9.20. Нагруженный направленный граф с исходными данными

Создание нового экземпляра графа (метод newGraph, добавление информации о ребрах (метод addEdge) и вызов метода findPaths (метод allPaths) представлены на рис. 9.21.

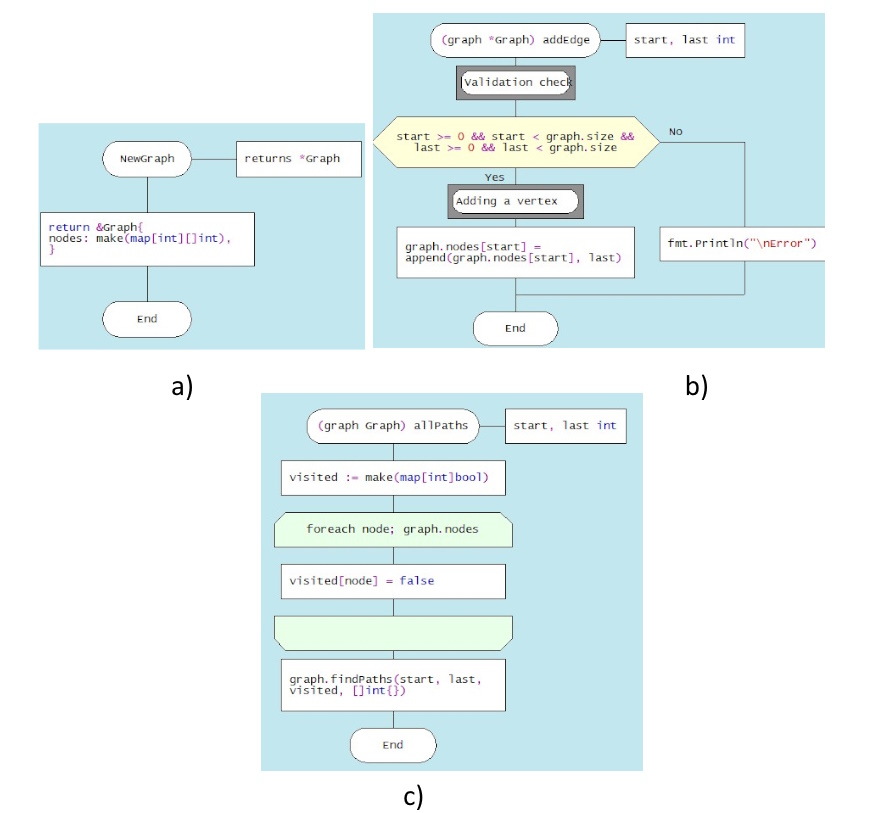


Рис. 9.21. Дракон-диаграммы методов создания графа

1. newGraph; b)addEdge; c) allPaths

Алгоритм поиска всех возможных путей между двумя вершинами основан на последовательном анализе списка связностей вершин и рекурсивном обращении к модулю findPath, реализующим обход узлов с фиксацией посещения каждой вершины. Дракон-диаграмма метода findPath представлена на рис. 9.22. Алгоритм состоит из трех частей: в первой части устанавливается корректность параметров модуля (имена начальной и конечной вершин), проверка факта посещения текущей вершины. Во второй части осуществляется проход по еще не посещенным узлам графа, формируя строку, состоящую из посещенных имен вершин и разделителя между ними, например, «2-0-1-4-3-5» (рис. 9.21). В третьей части при достижении конечной вершины, то-есть при выполнении условия «start == last», определяется минимальный путь между этими вершинами. Кроме того, строка возможного пути, например, «2-0-1-4-3-5», разбивается на участки типа «2-0» с целью формирования ключей для карт, содержащих расстояния между соответствующими вершинами и стоимости проезда 1 км на этих участках.

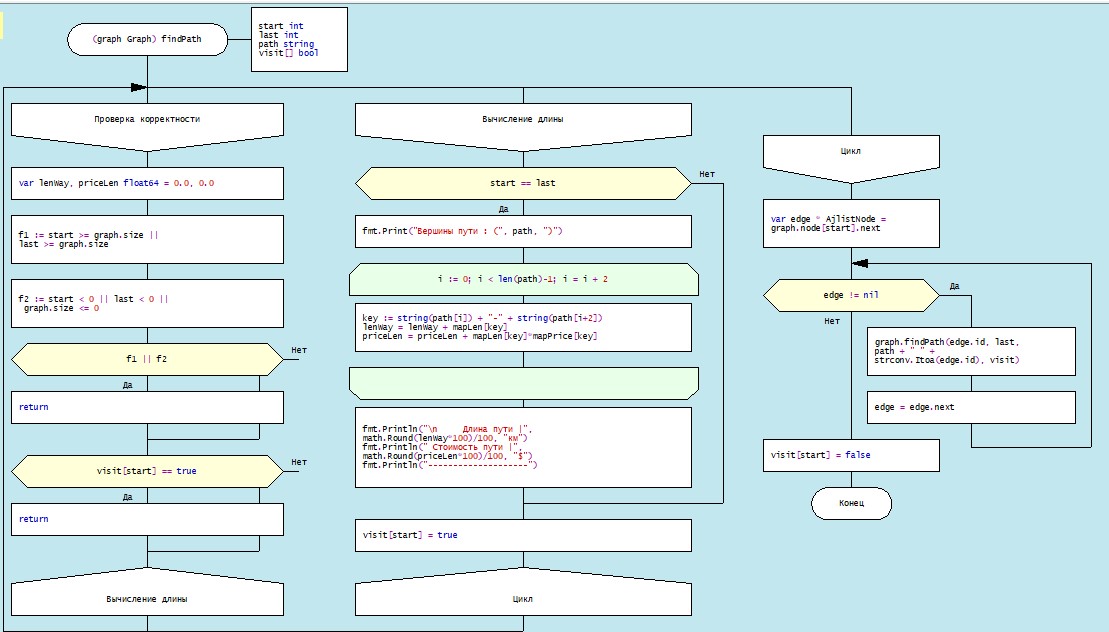


Рис. 9.22. Дракон-диаграмма модуля findPath

Рассмотрим задачу поиска всех возможных путей между двумя вершинами графа (2) и (5),, показанного на рис. 9.23:

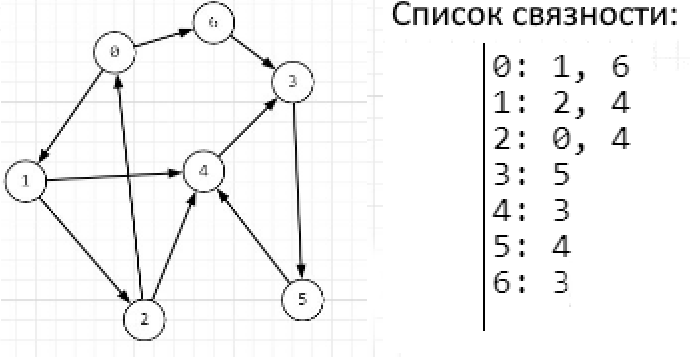


Рис.9.23. Граф и список связности

Результат первого обхода графа между вершинами (2) и (5), включая процесс рекурсии, показан в табл.9.2.

Табл.9.2. Процесс реализации модуля findPath

|  |  |  |  |
| --- | --- | --- | --- |
| 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** |

Все пути обхода ориентированного графа между вершинами (2) и (5) показаны на рис. 9.24.

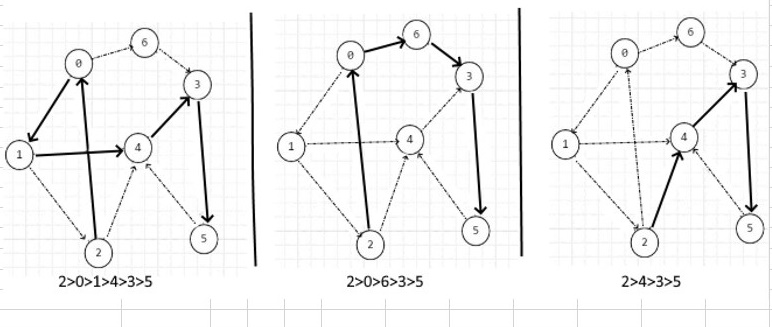


Рис. 9.24. Возможные пути обхода вершин ориентированного графа

Результаты сравнения стоимостей проезда от вершины (2) и (5) показаны в таблице 9.3.

Таблица 9.3. Расчет затрат на проезд

|  |  |  |
| --- | --- | --- |
| Travel | Length of travel, km | Cost of travel, $ |
|  |
| 2-0-1-4-3-5 | 144.95 | 254.69 |  |
| 2-0-6-3-5 | 137.88 | 177.46 |  |
| 2-4-3-5 | 104.53 | 180.23 |  |

Из таблицы 9.3. следует, что с точки зрения затрат второй маршрут (2-0-6-3-5) является наиболее выгодным.

**9.6. Алгоритм Дейкстры – метод нахождения кратчайшего пути**

Алгоритм Дейкстры (англ. *Dijkstra’s algorithm*) разработан с целью нахождения кратчайших путей от заданной вершины  до всех остальных вершин при условии неотрицательных весов ребер рафа. Алгоритм был назван по имени голландского ученого Э. В. Дейкстры в 1956 году. Данный алгоритм получил широкое распространение в различных областях науки, техники и социальной жизни. В GPS-системах алгоритм используется для нахождения наиболее быстрого пути до пункта назначения. В телекоммуниционных сетях он используется для определения оптимального пути передачи данных от источника к получателю. В робототехнике алгоритм может быть использован для планирования пути робота для достижения цели наиболее эффективным образом. В компьютерных играх – для определения пути персонажей или объектов. В сетевом планировании алгоритм может быть применен для нахождения оптимального пути доставки товаров.

Для реализации алгоритма Дейкстры создаются структуры vertex типа Vertex для описания вершин и Graph для описания графа (рис.24)

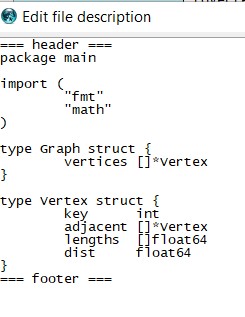


Рис. 9.25 Описание структур вершин и графа

Здесь key – ключ вершины графа; adjacent – список смежности для вершины; lengths - набор длин ребер до вершин из списка смежности; dist –суммарная длина пути.

Далее, в программе main создается экземпляр графа и в цикле создаются экземпляры вершин (рис. 9.26):

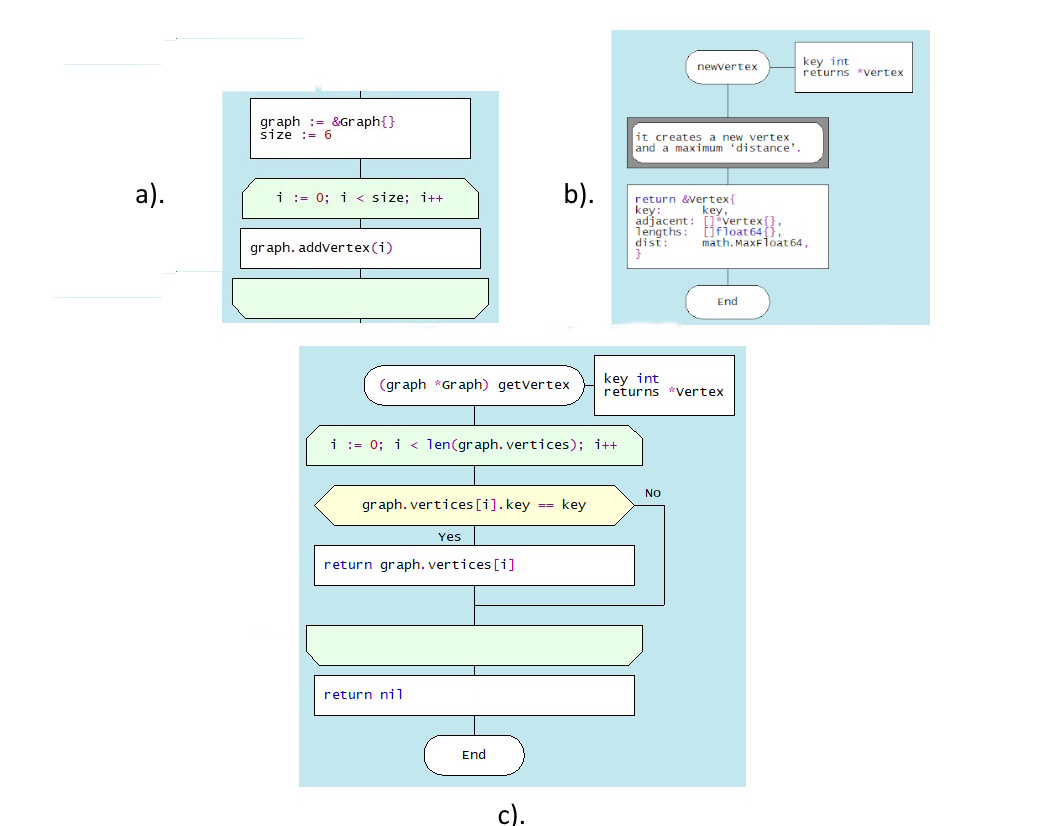


Рис. 9.26. Дракон-диаграммы методов формирования вершин графа

а). создание экземпляра графа и шаблонов вершин

b). создание новой вершины

c). наполнение вершины данными

После создания вершин графа и заполнения их данными (ключи, соседние вершины и расстояния), формируются ребра с помощью метода addEdge (Рис. 9.27):

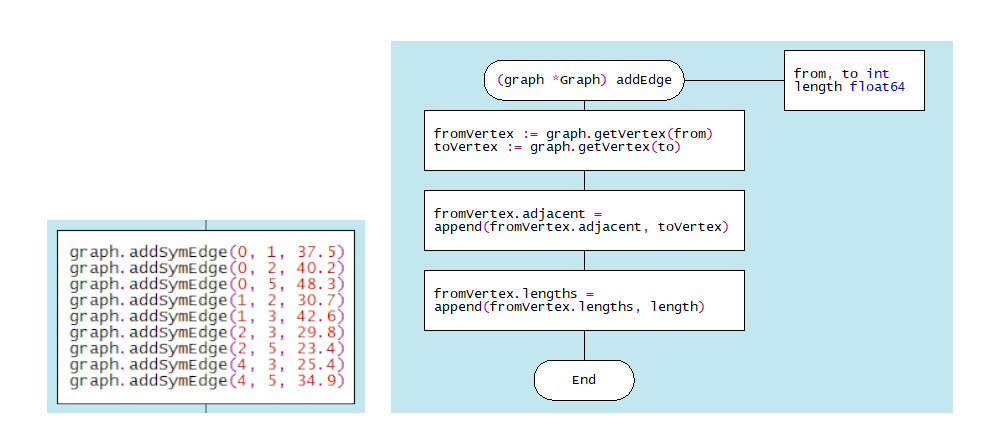


Рис. 9.27. Дракон-диаграммы метода формирования ребер графа addEdge

Алгоритм метода addEdge(from, to, length) реализуется следующим образом. Сначала метод получает две вершины, между которыми нужно добавить ребро (graph.getVertex(from), graph.getVertex(to)), где from и to - ключи вершин. Затем метод добавляет вершину toVertex в список смежных вершин для вершины fromVertex (append(fromVertex.adjacent, toVertex). В результате формируется ребро между этими двумя вершинами fromVertex и toVertex. На последнем шаге метод добавляет длину ребра в список длин ребер для вершины fromVertex (append(fromVertex.lengths, length)). Это означает, что длина ребра между вершинами fromVertex и toVertex теперь известна.

Заметим, что в программе main используется метод addSymEdge (u,v,L) который создает симметричные данные по каждому ребру в ненаправленном графе (рис.9.28):

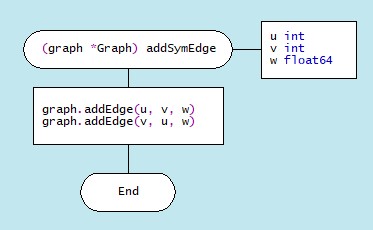


Рис.9.28. Дракон-диаграмма метода симметризации ребра addSymEdge

Далее происходит обращение к методу Dijkstra (startKey int), где startKey int – исходная вершина, и вывод конечных результатов (Рис.9.29):

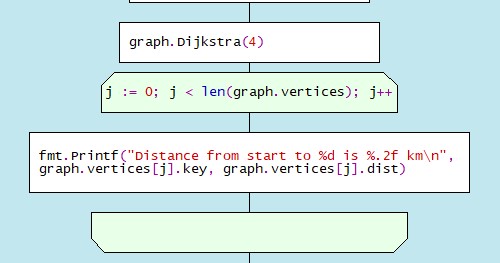
. 

Рис.9. 29. Дракон-диаграмма обращения к методу Deikstra и вывод результатов

Дракон-диаграмма метода Dijkstra(startkey) представлена на рис. 9.30.

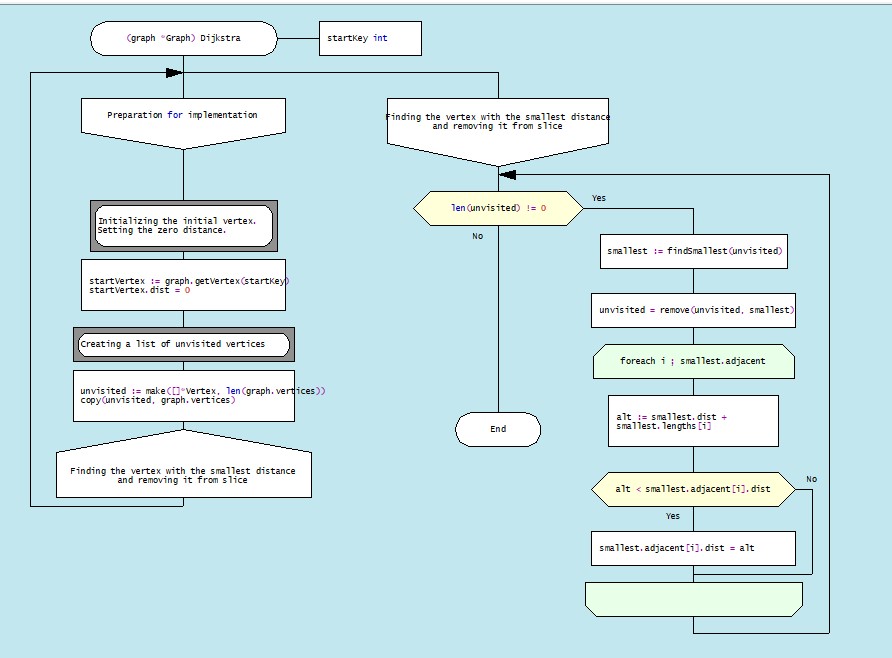


Рис. 9.30. Дракон-диаграмма метода Dijkstra(startkey)

Рассмотрим основные шаги реализация этого алгоритма. Вначале происходит и**нициализация** (startVertex := graph.getVertex(startKey); startVertex.dist = 0), результатом которой является получение начальной вершины по ключу и установление ее расстояния, равное 0. На втором шаге создается список **непосещенных вершин**,которыйзатем копируетсякомандойcopy с целью обеспечения возможности манипулировать этим списком в процессе обработки данных вершин, не меняя сам граф.

Далее выполняется цикл for len(unvisited) != 0 обхода всех непосещенных вершин с целью поиска вершины с наименьшим расстоянием с помощью обращения к методу smallest := findSmallest(unvisited) (Рис. 9. 31.)

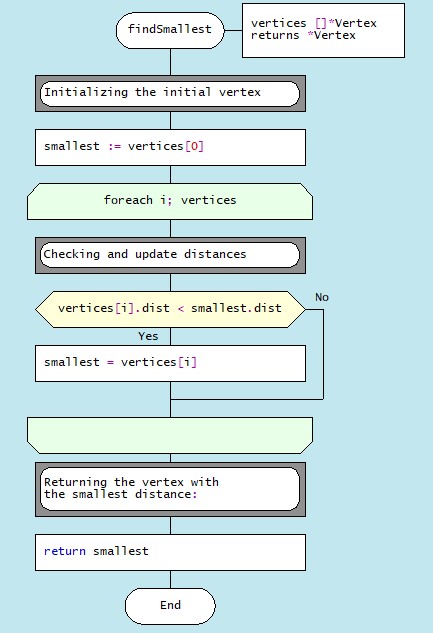


Рис. 9.31. Дракон диаграмма метода findSmallest(unvisited)

Затем **из списка непосещенных вершин удаляется текущая вершина (**unvisited = remove(unvisited, smallest)). И, наконец, в цикле по всем вершинам происходит обновление расстояний до смежных вершин в результате обнаружения более короткого пути.

В качестве примера рассмотрим применение алгоритма Дейкстры к нахождению кратчайших путей от заданной вершины для неориентированного графа, изображенного на рис. 9.32.

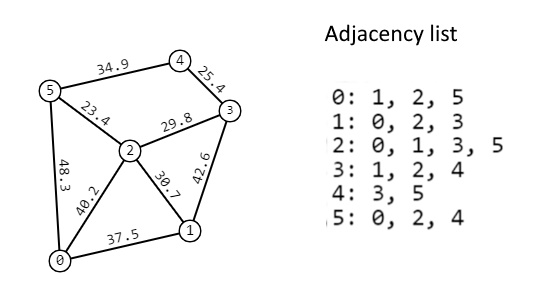


Рис.9.32. Неориентированный нагруженный граф

Результаты расчетов кратчайших путей от вершины 0 до остальных вершин графа (рис. 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

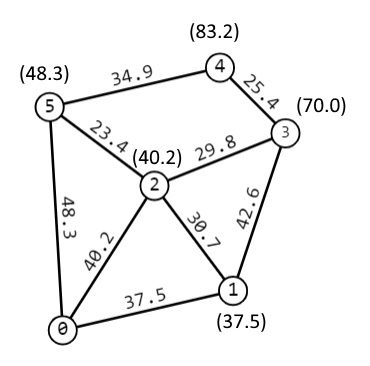


Рис. 9.33. Кратчайшие расстояния от вершины 0 указаны в скобках

И, в завершение, об оценке сложности алгоритма Дейкстры. Эта оценка может варьироваться в зависимости от структуры данных, используемой для представления графа. В случае применения связанного списка (Linked List) временная сложность алгоритма Дейкстры оценивается как  O(V^2), где V - количество вершин в графе. Это связано с тем, что приходится просматривать все вершины при каждом извлечении минимального элемента из очереди приоритетов.

При использовании структуры данных map временная сложность алгоритма Дейкстры будет O((V+E) log V), где E - количество ребер в графе. Это связано с тем, что извлечение минимального элемента из очереди приоритетов, реализованной с помощью map, занимает время **O(log V)**.

В обоих случаях пространственная сложность будет O(V + E), так как нужно хранить все вершины и ребра графа. Однако, стоит отметить, что map обычно занимает больше места, чем связанный список, из-за дополнительной информации, которую он хранит (ключи и значения).