**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:

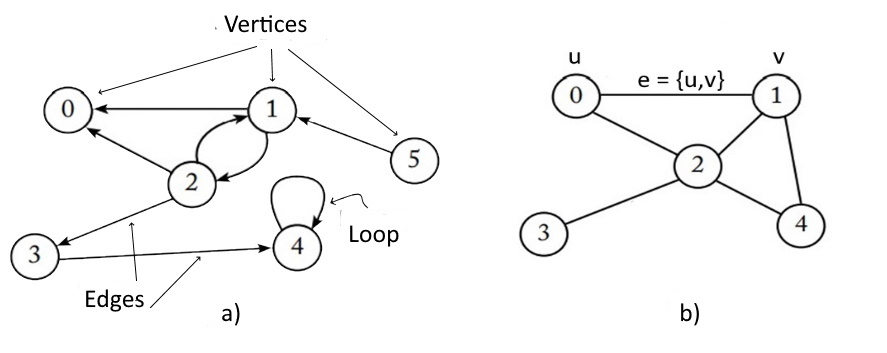


Рис. 9.1 Виды графов а) - неориентированный; б) – направленный

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 ориентированный граф и матрица смежности, которая для ориентированного графа строится таким образом, что (1) обозначает наличие смежной связи между вершинами. Так, для вершины (2) смежными вершинами являются вершины (1) и (3), а для вершины (5) – вершина (2).

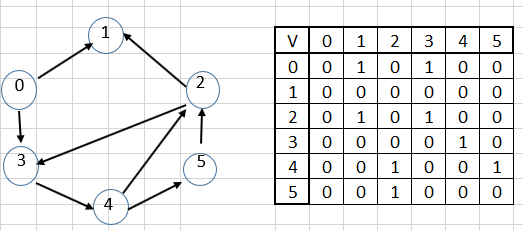


Рис. 9.3. Представление графа в матрице смежности.

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.

Намного более эффективным способом представления графа заключается в использовании списка смежностей. Точнее речь идет о массивах связанных списков (см. Раздел 4), где каждый отдельный список содержит информацию о том, какие вершины являются смежными по отношению к заданной. Так список смежности для графа, изображенного на рис. 9.3. имеет вид:

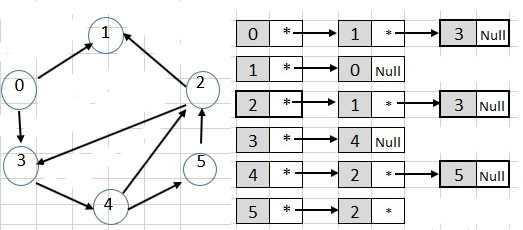


Рис. 9.4. Представление графа в списке смежностеи

Матрица и список смежности для ненаправленного графа отличаются от направленного тем, что метки (1) в матрице проставляются для всех смежных вершин (рис.9.5.). На обеих рисунках можно видеть, что список смежности практически является односвязным списком, состоящим из информационной составляющей (номер вершины) и адреса следующей вершины. Адрес конечного элемента в списке – Null. В то же время следует учитывать то, что в некоторых случаях, например, плотные матрицы, матричное представление оказывается предпочтительней списка смежных вершин.

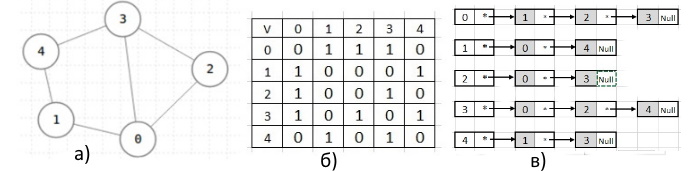


Рис. 9.5. Матрица и список смежности для ненаправленного графа

Как видно, пространственная сложность направленного и ненаправленного графов зависит от его представления: для матричного – О(N\*N, для представления списком – О(N+M), где N – число вершин, M – число ребер. На практике можно использовать оба вида представлений графа, в то же время целесообразно использовать матрицу смежности для представления *плотных графов* и список смежности для представления *разреженных графов*. Плотные графы - это графы, имеющие большое количество ребер, а разреженные графы - те, которые имеют небольшое количество ребер.

**9.3. Алгоритмы графов**

Вследствие широкого распространения графов существует много алгоритмов их обработки [https://kalkicode.com/adjacency-matrix-representation-o]. Задачи, решаемые в рамках теории графов, можно условно поделить на несколько групп:

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

Рассмотрим основные алгоритмы, применяемые в этих задач и реализованные, как и прежде, в рамках гибридного программирования ДРАКОН + Golang.

9.3.1. Представление графа

В перечень основных задач на определение графа входят задачи на построение графа по заданному числу вершин и ребер, построение матрицы смежности и инцидентности, вычисление основных характеристик графа. Для построения графа в языке Golang необходимо создать новые типы, содержание которых определяется конкретной задачей и выбором базовых структур данных, из которых будут создаваться новые типы. Конечно, все новые типы создаются на основе базовой структуры struct, но количество и содержание ее полей зависят от типа графа. В частности, описание нагруженного графа, ребра которого отмечены, например, расстоянием между вершинами, должен включать поле weight (рис. 9.6.):

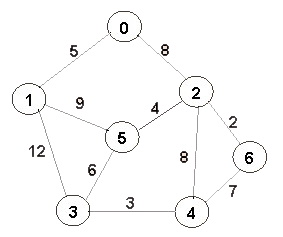


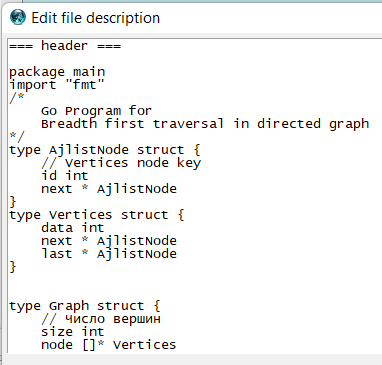
Рис. 9.6. Нагруженный граф

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 (двумерный срезы, карты, срезы, элементами которых являются односвязные списки [Кормен Т. Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д. Вильямс», 2013. ˗ С.626.]. Далее будем использовать типы: для описания вершин, для списка смежных вершин и для описания графа (Табл. 9.1.):

Табл. 9.1. Типы переменных для графовых алгоритмов

|  |  |  |
| --- | --- | --- |
| Описание вершин | Список смежных вершин | Описание графа |
| type Vertices struct {      data int      next \* ListVertAdj      last \* ListVertAdj  } | type ListVertAdj struct {   id int   next \* ListVertAdj  } | type Graph struct {      size int      node []\*Vertices  } |
| *data* – имя вершины;  *next* – адрес ближайшей смежной вершины;  *last* – адрес последней вершины. | *id* - имя вершины;  *next* – адрес смежной вершины (рекурсия) | *size* – число вершин;  node []\* Vertices –  срез адресов вершин |
| type Graph struct {  vertices map[int]map[int]bool  } |  |  |
|  |  |  |

Описания основных сущностей графов формируются в опции File descryption в меню Drakon WEB Editor:



В функции *main ()* создается переменная *graph*, являющаяся указателем на объект типа *Graph,*.которая инициализируется с помощью функции getGraph(6). Эта функция объявляет создание граф с 6 вершинами и возвращает указатель на объект типа *Graph*:

    var graph \*Graph = getGraph(6);

Функции initVert() и getVertices(*data* *int*) инициируют переменную graph, содержащий указатель на структуру объекта Graph, которая включает в себя поле размера 6, а также срез указателей на другие вершины графа (рис. 9.7)..

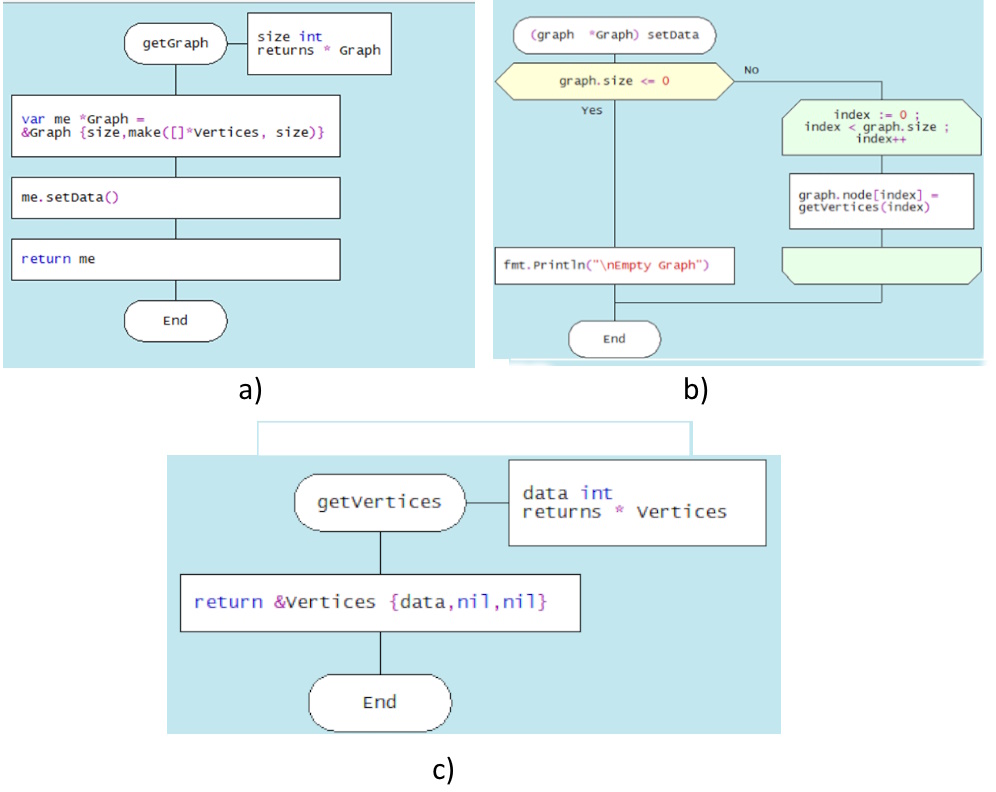


Рис. 9.7. Дракон-диаграммы построения графа: а) getGraph(6); б) initVert(); в) getVertices(*data* *int*)

Новые экземпляры графа выполняется методом addEdge(start, last), который с помощью метода getAjlistNode(*id* *int*) создает и возвращает новый объект типа AjlistNode, имеющий два поля: имя вершины id и нулевым указателем next на следующую вершину. Дракон-диаграмма метода addEdge(start, last) оказана на рис. 9.8.

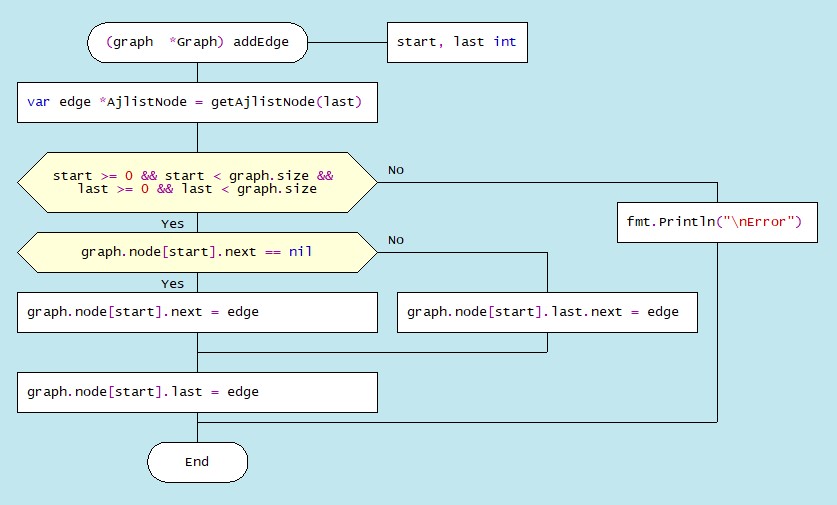


Рис. 9.8. Дракон-диаграмма функции addEdge(start, last)

Cписок cвязности, представляющий граф, формируется методом printGraph(), дракон-диаграмма которого представлена на рис. 9.9.

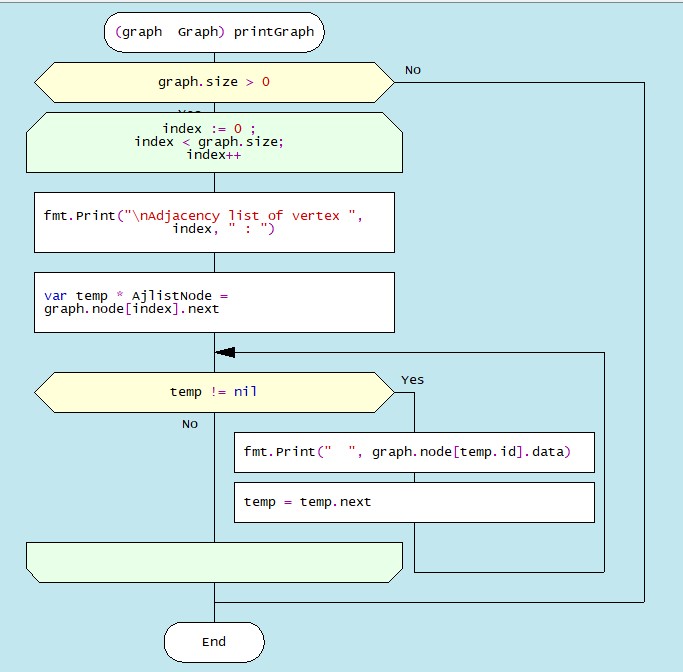


Рис. 9.9. Дракон-диаграмма функции printGraph()

**9.4. Алгоритмы обхода (поиска) графа**

После построения графа и проверки его свойств естественными являются обхода графа по его вершинам. Общая задача формулируется следующим образом: обойти граф, начиная от заданной вершины и перемещаясь по ребрам к другим вершинам, посетить все вершины, соединенные с исходной. Существует два основных способа обхода графов: *обход в глубину и обход в ширину*, обеспечивающие перебор всех соединенных вершин.

Отличие поиска в глубину от поиска в ширину заключается в том, что (в случае неориентированного графа) результатом алгоритма поиска в глубину является некоторый маршрут, следуя которому можно обойти последовательно все вершины графа, доступные из начальной вершины. Этим он принципиально отличается от поиска в ширину, где одновременно обрабатывается множество вершин, в поиске в глубину в каждый момент исполнения алгоритма обрабатывается только одна вершина. С другой стороны, поиск в глубину не находит кратчайших путей, зато он применим в ситуациях, когда граф неизвестен целиком, а исследуется каким-то автоматизированным устройством.

а). Обход графа в глубину

**Алгоритм поиска (или обхода) в глубину** (англ. depth-first search, DFS) позволяет построить обход ориентированного или неориентированного графа, при котором посещаются все вершины, доступные из начальной вершины. Выбор поиска 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. На практике обход в глубину применяется при анализе вариантов выбора дальнейшего поведения в играх.

Обход 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 visited, где visited[i] == True для посещенных вершин, и visited[i] == false для непосещенных. Пометка «о посещении вершины» ставится при заходе в эту вершину.

Поскольку целью обхода в глубину зачастую является построение дерева обхода в глубину, то сразу же будем хранить предшественника для каждой вершины. Алгоритм обхода в глубину оформим в виде рекурсивной функции dfs, где start — номер вершины, из которой запускается обход.

Поиск в глубину начинается с посещения исходной вершины vi, после чего рекурсивно посещаются все смежные вершины. Посещение вершины фиксируется в логическом массиве 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.9.10.

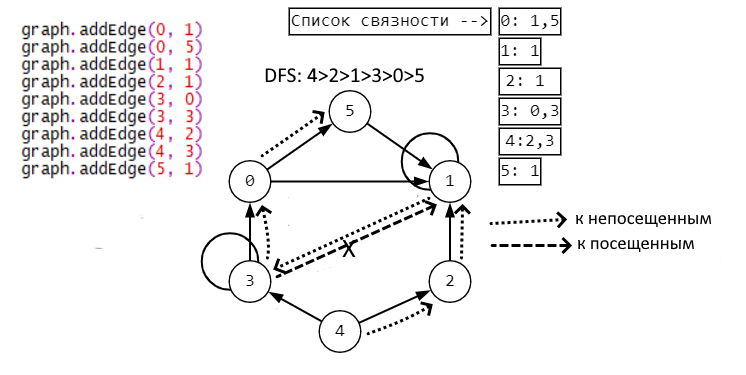


Рис. 9.10. Обход в глубину графа по списку смежности

Дракон-диаграмма алгоритма обхода графа в глубину представлена на рис. 9.11.

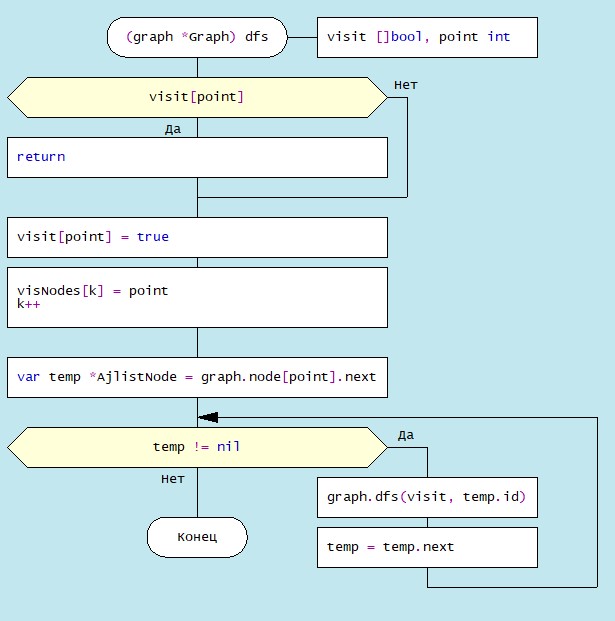


Рис. 9.11. Дракон-диаграмма алгоритма обхода в глубину

б). Обход графа в ширину

 Обход графа в ширину (breadth-first search) осуществляется подобно распространяющейся волне, начиная с посещения исходной вершины, которая выбирается произвольно. Затем посещаются соседние вершины, за ними - соседей соседей, и так далее. Иногда такой обход сравнивают в распространением огня через соседние вершины: сначала поджигаются вершины, являющиеся соседними по отношению к исходной вершине, затем – то же самое по отношению к другим вершинам.

Обход в ширину можно пояснить иначе, более формально. Обход происходит по мере увеличения расстояния от начальной вершины. Здесь расстояние определяется числом ребер между вершинами: в частности, расстояние между соседними вершинами по отношению к исходной равно 1 (рис. 9.12):

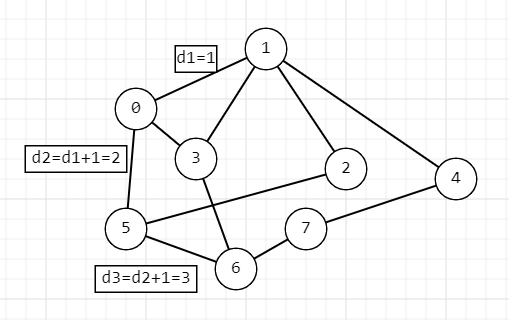


Рис. 9.12. Расстояния от исходной вершины

Иллюстрация обхода графа в ширину на примере гамильтонова графа представлена на рис.9.13:

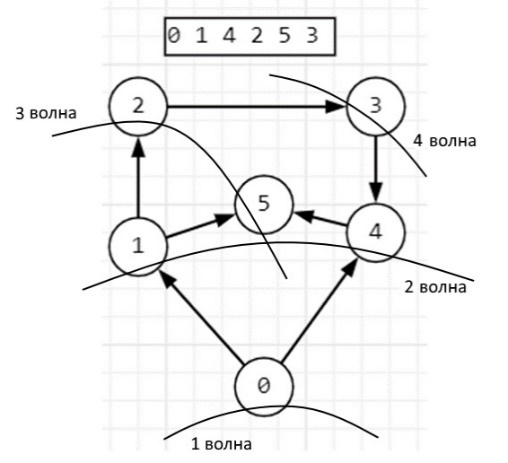
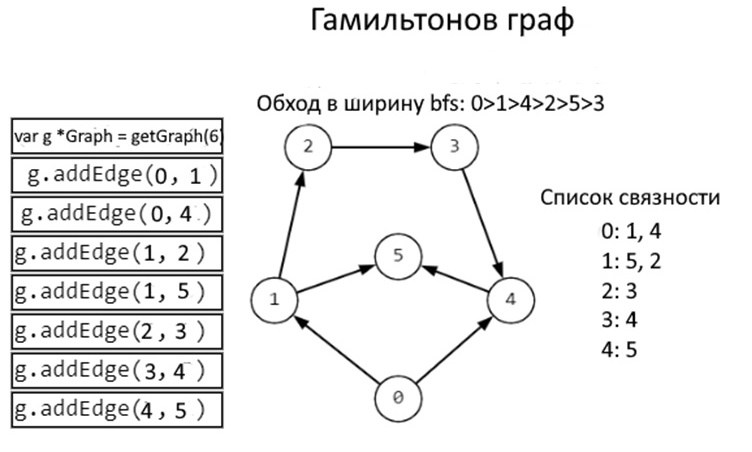


Рис. 9.13. Обход в ширину графа (волновой обход)

Дракон-диаграмма алгоритма обхода графа в глубину представлена на рис.9.14.

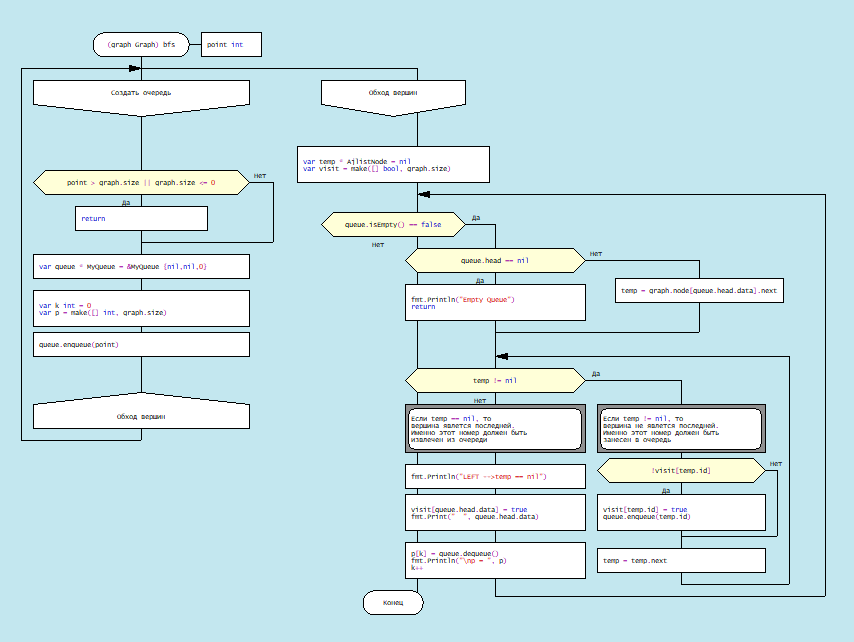
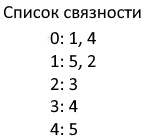


Рис. 9.14. Дракон-диаграмма алгоритма обхода в ширину

Поскольку алгоритм обхода графа в ширину достаточно сложен, рассмотрим его более подробно. Прежде всего, обратим внимание на введенные типы переменных:

а) тип Vertices – для информационной структуры вершины; состоит из трех полей: data int - ключ вершины (в данном случае – ее номер), next – адрес первой смежной вершины графа, last - адрес последней смежной вершины графа:



|  |  |  |
| --- | --- | --- |
| 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>} |

б) тип AjlistNode – для отдельной вершины, фактически структура односвязного списка;

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

в) тип QNode – для элемента очереди, структура которой имеет вид:

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

г) тип MyQueue - для переменной <очередь>, структура которой имеет вид:

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

д) тип Graph – для переменной, представляющая структуру графа: size -количество вершин, node – срез, содержащий адреса смежных вершин:

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

Алгоритм обхода графа в ширину начинается с выбора исходной вершины, пусть эта вершина помечена как «0» (рис.9.10). Эту вершину нужно поместить в очередь queue, имеющую тип \*MyQueue:

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

Структура queue связана с методом постановки в очередь – queue.enqueue(point), где point – ключ вершины (рис. 9.15).

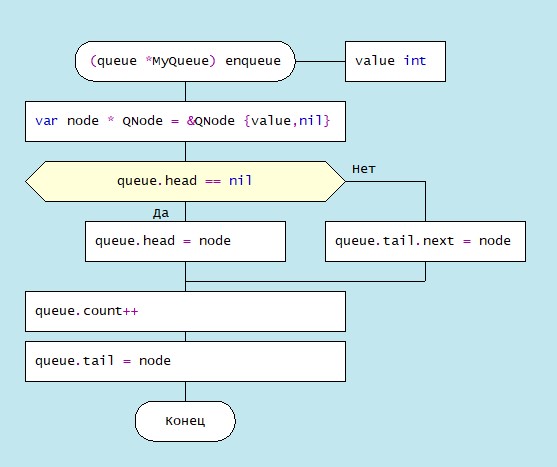


Рис. 9.15. Дракон-диаграмма метода queue.enqueue

В модуле enqueue новый элемент очереди queue как экземпляр структуры node типа QNode представляет собой односвязный список. Далее у головного элемента проверяется адрес следующего узла и если он равен nil, то этому узлу присваивается значение node, формируя таким образом узлы очереди queue. Если же он не равен nil, тогда значение node присваивается последнему элементу очереди: queue.tail.next = node. При этом длина очереди queue.count увеличивается на 1.

Удаление вершины из очереди реализуется методом dequeue, связанным с экземпляром очереди queue, дракон-диаграмма алгоритма которого показана на рис. 9.16:

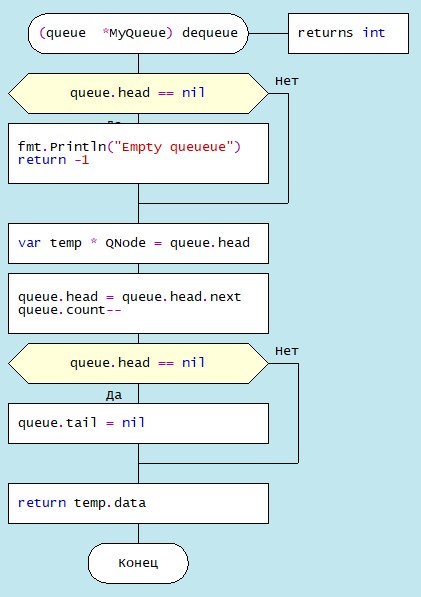


Рис.9.16. Дракон-диаграмма метода queue.dequeue

**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. Хорошо известен алгоритм Дейкстра, определяющий минимальное расстояние между двумя заданными узлами графа. Усложним немного эту задачу: с целью определения оптимального пути найти расстояния всех возможных путей между двумя географическими пунктами, а также суммарная 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 пунктами и ценой проезда за 1 километр (рис. 9.17).

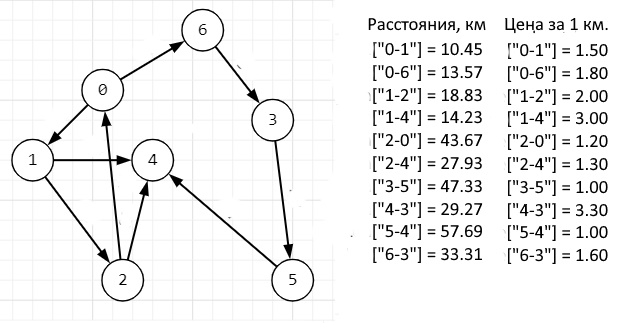


Рис. 9.17. Нагруженный направленный граф с исходными данными

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 модулю findPath которого представлена на рис. 9.18. 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. 9.15). 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.

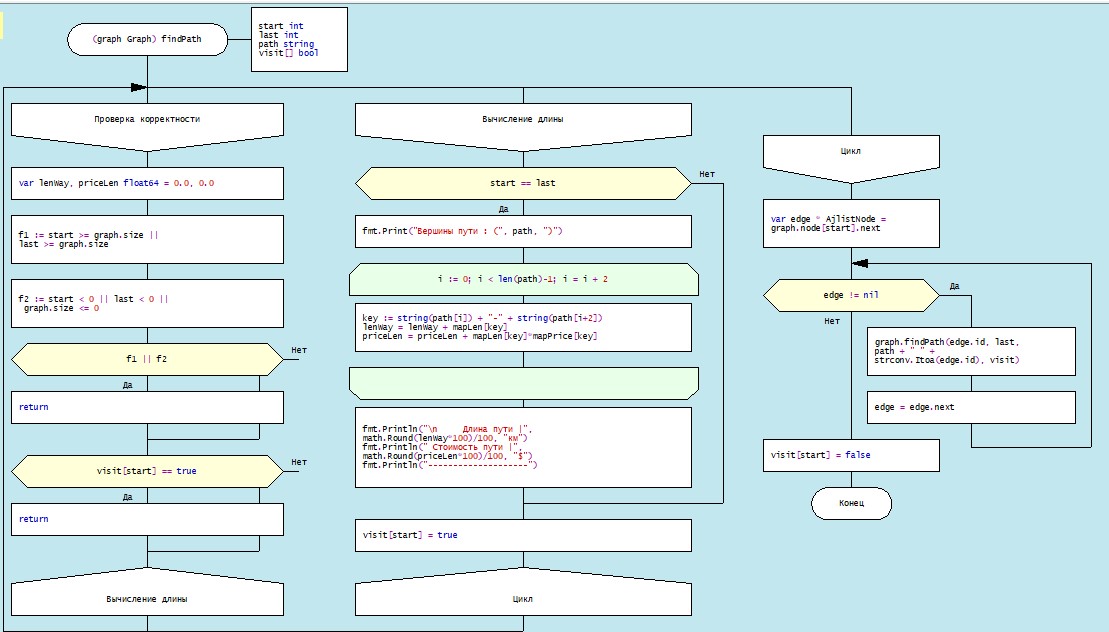


Рис. 9.18. Дракон-диаграмма модуля findPath

Рассмотрим задачу поиска всех возможных путей между двумя вершинами графа, показанного на рис. 9.19:

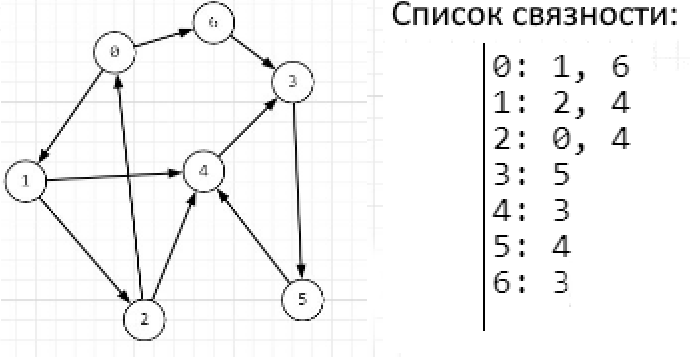
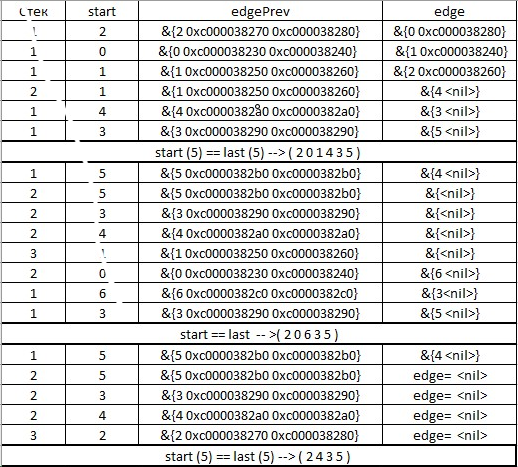
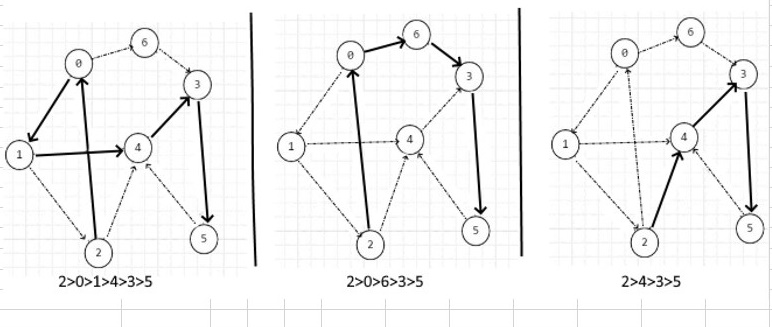


Рис.9.19. Граф и список связности

Процесс реализации алгоритма показан в табл.9.2. В столбце «стек» показано количество рекурсивных обращений к модулю findPath, параметрами которого являются: начальная (start int) и конечная (last int )вершины, путь (path string) и срез логических состояний посещения вершин (visit[] bool). В столбце «start» отображены ключи текущих вершин, в столбце «edgePrev» - односвязный список, который содержит имя текущей вершины и ее адрес, а также адрес следующей записи, в столбце «edge» - имя и адрес следующей записи. Алгоритм, пользуясь списком связанности, проходит, в случае выполнения условия (edge != nil) в цикле по соответствующим вершинам графа, заполняя, в случае выполнения условия (edge == nil) содержимое стека.

Табл.9.2. Процесс реализации модуля findPath 



**9.3. Алгоритм определения минимального количества ребер между двумя узлами**

В предыдущем подразделе был рассмотрен алгоритм поиска всех возможных путей перемещения из одной вершины в другую. Можно видеть, что число ребер в разных путях перемещения варьируется от 3 до 5. Далее рассмотрим алгоритм определения минимального количества ребер между двумя вершинами рис.9.20.:

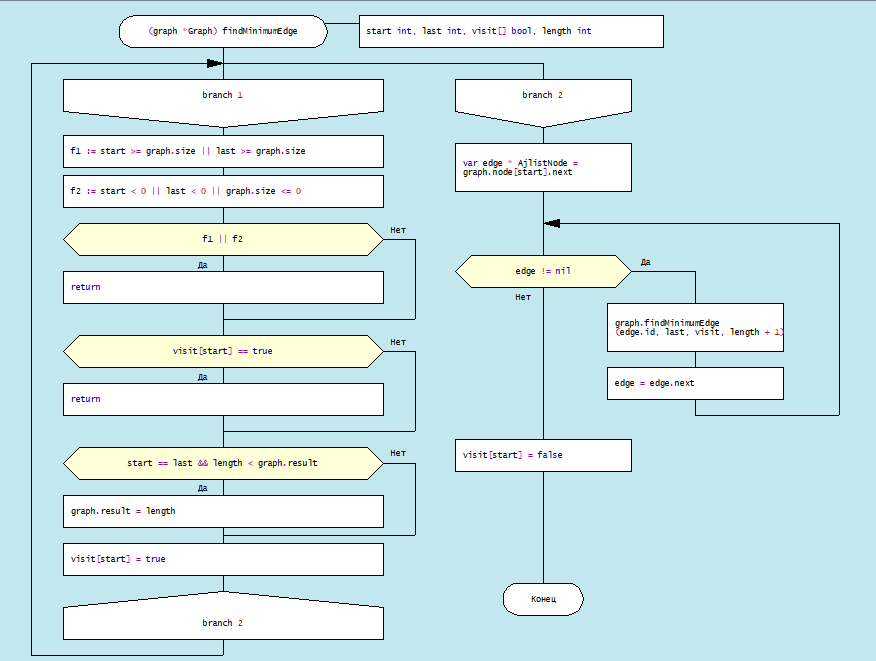
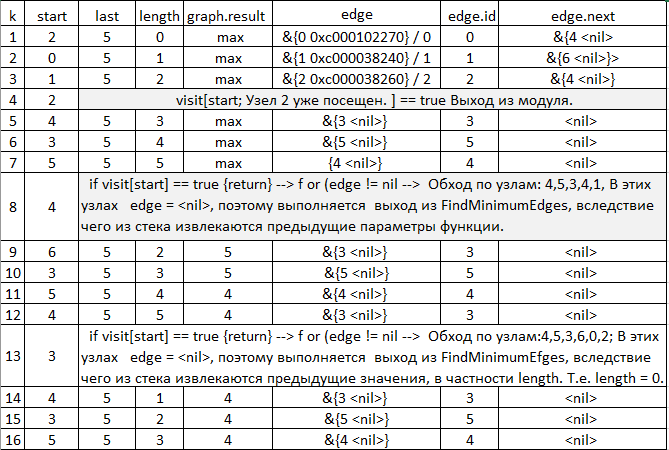


Fig.9.20. Drakon-diagram of the minimum number of edges algorithm

Table. 9.3. Процесс реализации модуля find



Литература по Разделу 9

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 | Programforyou](https://programforyou.ru/#link-about)

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