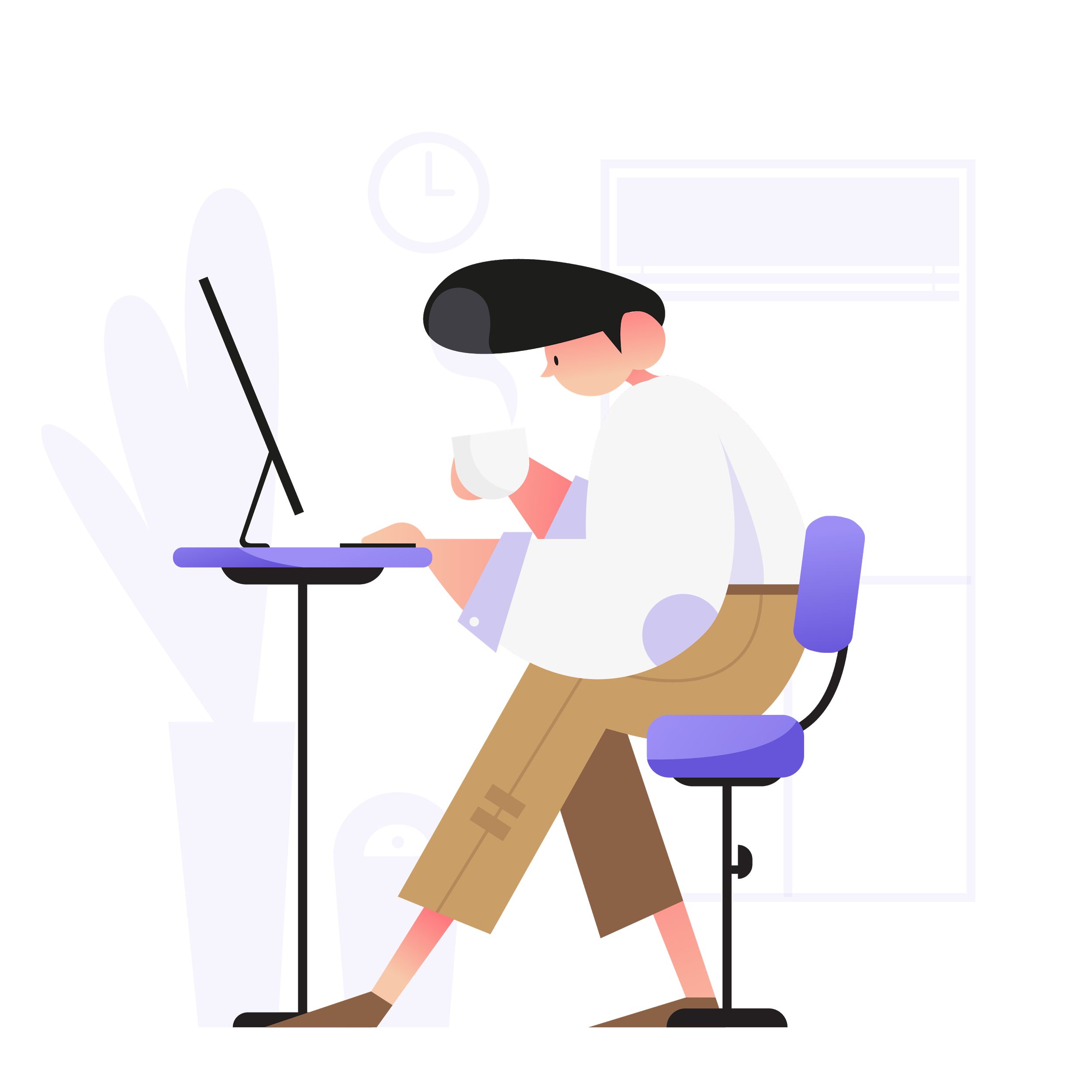
Алгоритмы и структуры данных на С#

Графы

Netcore 3.1



**На этом уроке**

1. Узнаем, что такое граф.
2. Освоим методы описания графа в коде, обход графа в ширину, обход графа в глубину.
3. Выясним, что такое волновой алгоритм и жадные алгоритмы.
4. Научимся находить кратчайшие маршруты.
5. Изучим алгоритм Флойда — Уоршелла.

**Оглавление**

[Введение](#_upzs4fkk051i)

[Графы](#_4wwy4y6m9g9f)

[Методы записи графа](#_lxpvp7e5flp4)

[Обход графа в ширину](#_uqt0ksv83tm2)

[Обход графа в глубину](#_naghchgo6znn)

[Волновой алгоритм](#_ivcnq5mmfdzq)

[Реализация](#_tyjcwt)

[Жадные алгоритмы](#_2eswv94go5tx)

[Задача 1](#_7ir9kph4a6vd)

[Задача 2](#_imvto52ilkpe)

[Кратчайшие маршруты](#_5y8we9xp1rp4)

[Алгоритм Флойда — Уоршелла](#_f64lyryawsk1)

[Заключение](#_o565zp6oq5pa)

[Практическое задание](#_j3tppzoujtfw)

[Дополнительные материалы](#_2jxsxqh)

[Используемые источники](#_7c9jrnh1eqkx)

# 

# Введение

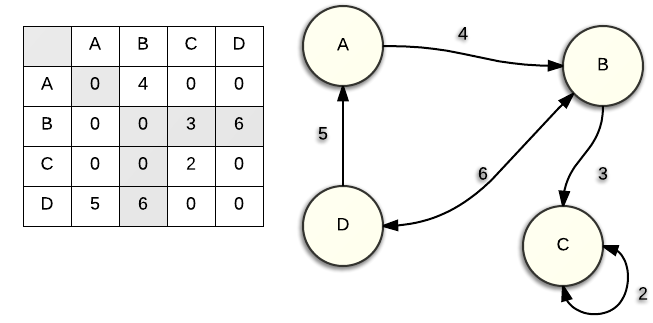
Граф — структура данных, которая присутствует во множестве мест: от социальных сетей до построения маршрутов по карте. Для программиста важно уметь оперировать этой структурой и находить кратчайшие пути в ней.

# Графы

Граф — это набор узлов (**вершин**) и связей между ними (**рёбер**). Информацию об узлах и связях графа обычно хранят в виде специальной таблицы — матрицы смежности.

Единица на пересечении строк A и B обозначает связь. Ноль указывает, что связи нет. Единица на главной диагонали обозначает петлю — ребро, которое начинается и заканчивается в одной и той же вершине. Строго говоря, граф — это математический объект, а не рисунок. Конечно, его можно нарисовать на плоскости, но матрица смежности не даёт никакой информации о том, как выглядит граф.

В примере все узлы связаны, то есть между любой парой узлов существует путь — последовательность рёбер, по которым можно перейти из одного узла в другой. Такой граф называется **связным**. Дерево — это частный случай связного графа, в котором нет замкнутых путей циклов.



Если для каждого ребра указано направление, граф называют **ориентированным (орграфом)**. Рёбра орграфа называют дугами. Его матрица смежности не всегда симметрична. Число, отличное от нуля, стоящее на пересечении строки A и столбца B, говорит о том, что существует путь из вершины A в вершину B.

## 

## Методы записи графа

Граф можно описать с помощью матрицы смежности. Это двумерный массив, в котором стоимость перехода из узла N в узел M имеет вес w[N,M], где w — матрица смежности (двумерный массив). На рисунке выше вы можете увидеть пример матрицы.

Ещё граф можно описать так же, как дерево, — с помощью ссылок и классов:

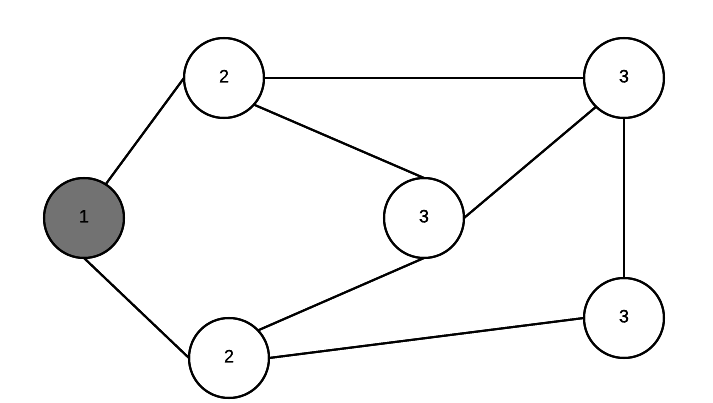
|  |
| --- |
| public class Node *//Вершина* {  public int Value { get; set; }  public List<Edge> Edges { get; set; } *//исходящие связи* }  public class Edge *//Ребро* {  public int Weight { get; set; } *//вес связи*  public Node Node { get; set; } *// узел, на который связь ссылается* } |

Описание в виде матрицы удобно, когда в графе очень много связей. В остальных случаях лучше использовать описание через классы.

## 

## Обход графа в ширину

Обход такого типа похож на движение волны. В примере ниже волна начинает движение от вершины, залитой серым цветом. У этой вершины есть две смежные. Волна заливает их на следующем шаге, после чего образуется новый фронт волны, состоящий уже из двух вершин. На втором шаге волна заливает все остальные вершины.



Заметим: существуют вершины, в которые можно попасть на некотором шаге из двух и более вершин, но при обходе в ширину в неё заходят единожды из какой-нибудь одной. Какой из нескольких путей в вершину выбрать — важный вопрос, но общего метода здесь нет. В каждом случае находят своё решение, исходя из требований конкретной задачи.

Ключевое понятие здесь — **фронт волны**. До завершения итерации множество вершин графа можно разделить на три подмножества:

* подмножество вершин, через которые волна уже прошла;
* подмножество вершин, достигнутых на последнем шаге: волна пришла в вершину, но ещё не ушла из неё;
* подмножество вершин, ещё не достигнутых волной.

Для построения пути волны нужно множество фронта. Это множество может быть представлено обычным массивом, который перестраивается по следующим правилам:

1. Вершина, в которую пришла волна, включается в массив фронта.
2. Вершина, из которой волна ушла, исключается из массива.

Итерационный шаг заключается в следующем: для каждой очередной вершины массива фронта определяются вершины, смежные с ней и ещё не достигнутые волной. Найденные смежные вершины включаются в массив фронта, после чего очередная вершина из массива убирается.

Такой массив удобно организовать как двустороннюю очередь (дек). Тогда, например, вершину, находящуюся в одном конце очереди, можно считать очередной для об­работки, а добавлять вершины в этом случае стоит в другой конец очереди. Как выше уже было замечено, все вершины графа делятся на три подмножества. Введём обозначения: 1 — вершина не достигнута волной; 2 — вершина находится во фронте волны; 3 — волна ушла из вершины. Очередную итерацию можно записать на псевдокоде следующим образом:

Обход всех вершин графа

Если статус текущей вершины = 2, то

Для всех вершин, смежных с текущей

Если статус смежной = 1, то статус смежной = 2

Статус текущей = 3

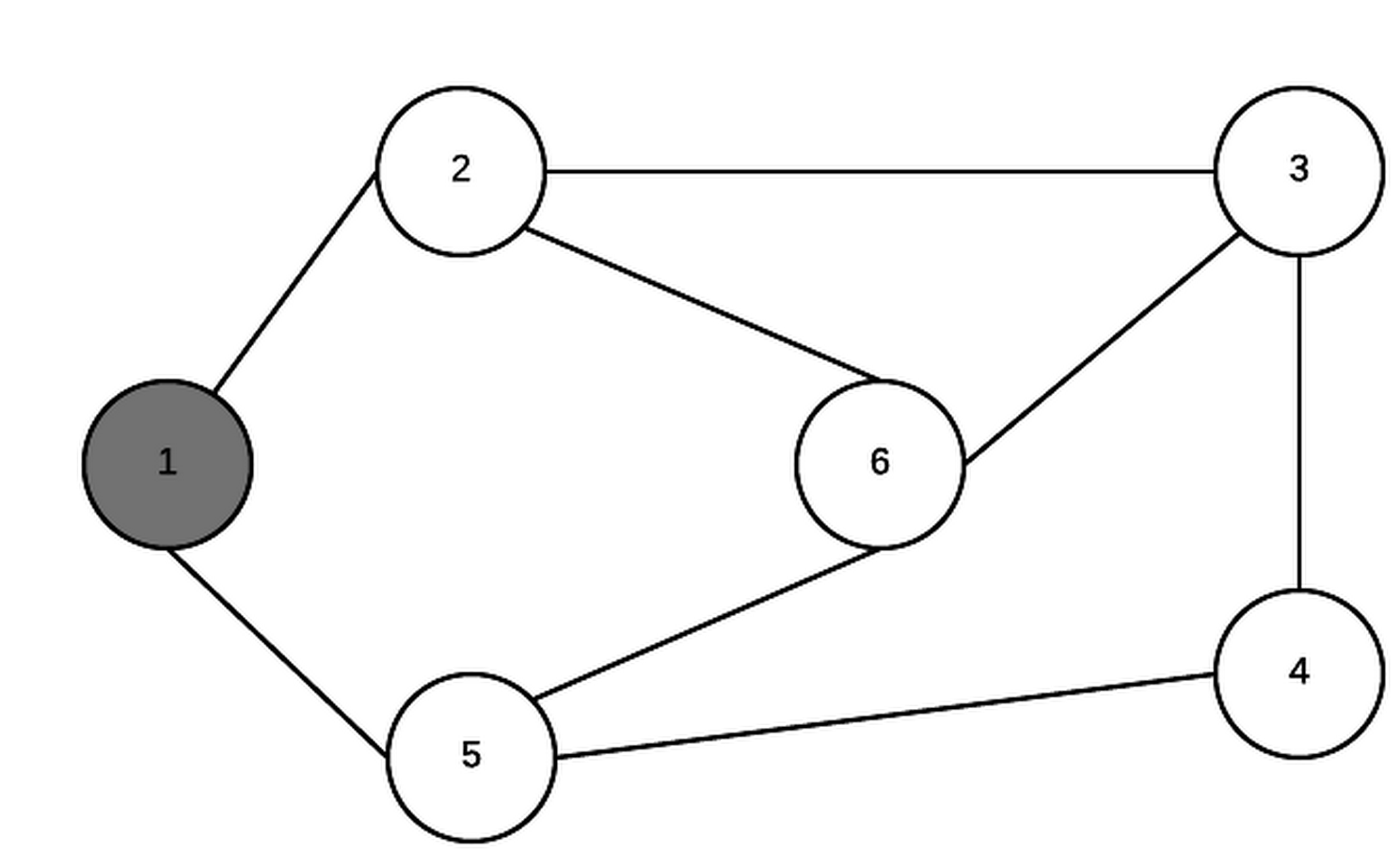
Если есть фрагмент, описывающий очередную итерацию, то весь процесс мож­но построить как цикл, условием завершения которого будет отсутствие вершин со статусом ниже 3. Процесс начинается с вершины-источника, которой первой присваивается статус 2.

## 

## Обход графа в глубину

**Обход графа в глубину** — это попытка уйти по графу как можно дальше, насколько это возможно. Если на каком-то шаге дальнейший путь оказывается невозможным (тупик), то выполняется возврат до тех пор, пока не будет обнаружено ещё не пройденное ребро.

**Тупик** — это вершина, из которой либо не выходит ни одно ребро, либо выходит, но все эти рёбра уже пройдены. Процесс обхода в глубину можно организовать рекурсивной процедурой. Попав в очередную вершину, обходчик графа обнаруживает перед собой ту же картину, которую он видел в предыдущей вершине, — некоторое количество ещё не пройденных рёбер. Таким образом, переход от вершины к новой вершине — это переход от задачи прохода графа к задаче прохода графа, но от ново­го источника, а это рекурсивная постановка. Ниже приведён пример:



Здесь не показан весь процесс. Обход вглубь начинается от вершины 1 и идёт до тупика, то есть до вершины 1, в которую обходчик попадает из пятой вершины. Обнаружив тупик, обходчик возвращается в пятую вершину и продолжает свой путь через шестую вершину до второй, на которой он опять встречает тупик. На этом шаге граф можно считать пройденным, так как обходчик побывал во всех вершинах. Но для обхода в глубину может быть поставлена задача не просто обойти все вершины или рёбра, а полностью перебрать все пути по графу. А в этом случае полный путь представляет собой дерево.

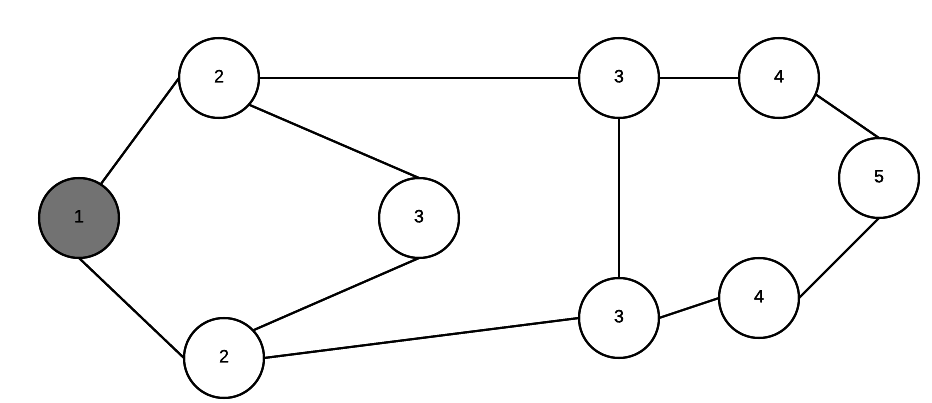
## 

## Волновой алгоритм

Формулировка задачи: дан непустой граф, необходимо найти путь между двумя вершинами, содержащий наименьшее количество вершин (рёбер).

Идея алгоритма: перед началом работы алгоритма некоторой начальной вершине присваивается число, называемое волновой меткой и имеющее минимальное значение. В алгоритме это минимальное значение равно единице. Далее волна начинает движение по графу, увеличивая свою высоту и расставляя волновые метки в тех вершинах, в которые она приходит. Метки расставляются так: если волна в вершину В пришла из вершины А, при этом в вершине А значение волновой метки равно Н, то в вершине В значение волновой метки определяется как Н+1.

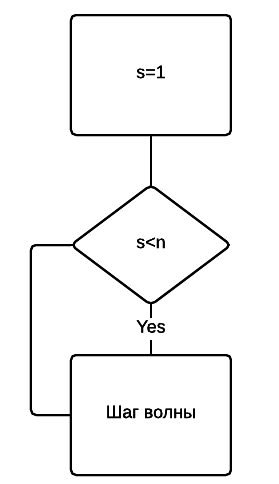
Таким образом, значения волновых меток зависят от длины пути, пройденного волной. Следовательно, можно утверждать, что чем больше значение волновой метки, тем больший путь был пройден волной до этой вершины. Заметим, что волновой алгоритм использует стратегию обхода графа в ширину. На рисунке показан пример распространения волны по графу.



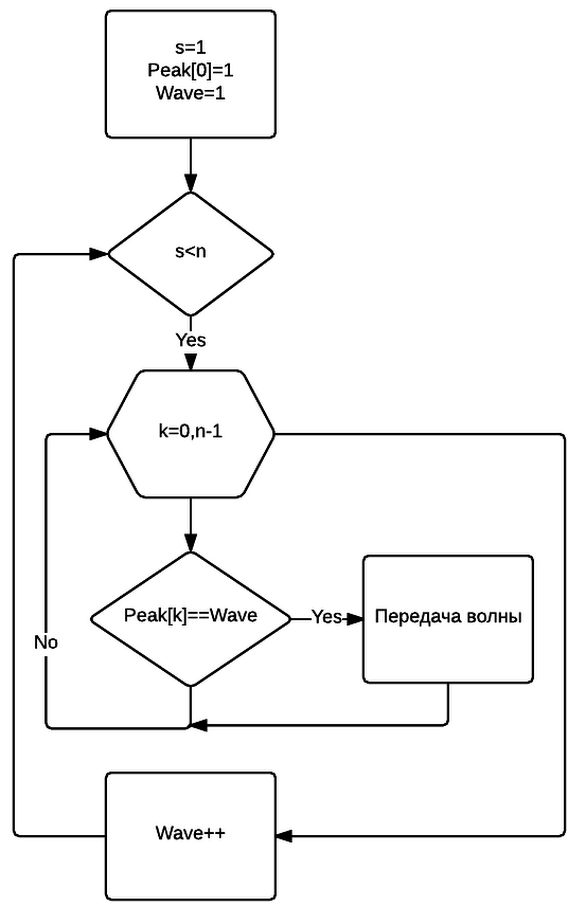
### Реализация

Представим граф матрицей смежности. После прохождения волны граф превращается во взвешенный. При этом веса присваиваются вершинам графа, а информация о вершинах никак не отображается в матрице смежности. Значит, нужна дополнительная структура данных, в которой будет сохраняться информация о проходящей волне, — одномерный массив вполне подойдёт. Для формирования массива вершины графа достаточно пронумеровать произвольным образом, тогда индекс элемента массива есть номер вершины графа. Договоримся также, чтобы избежать путаницы с нумерацией, что номер вершины соответствует номеру строки в матрице смежности.

Очевидно, что процесс прохождения волны можно смоделировать циклом, работающим до тех пор, пока волна не заполнит весь граф. Если создать счётчик, считающий количество «передач волны», то условие завершения цикла будет таким:

Условие предполагает, что вершины нумеруются с нуля, и n — их количество. В некоторые из вершин графа волна пришла на предыдущем шаге, и они становятся источником дальнейшего движения. Назовем их вершинами типа A. Некоторые из вершин уже были источниками — это тип B, и некоторые ещё ждут волны — тип C. 

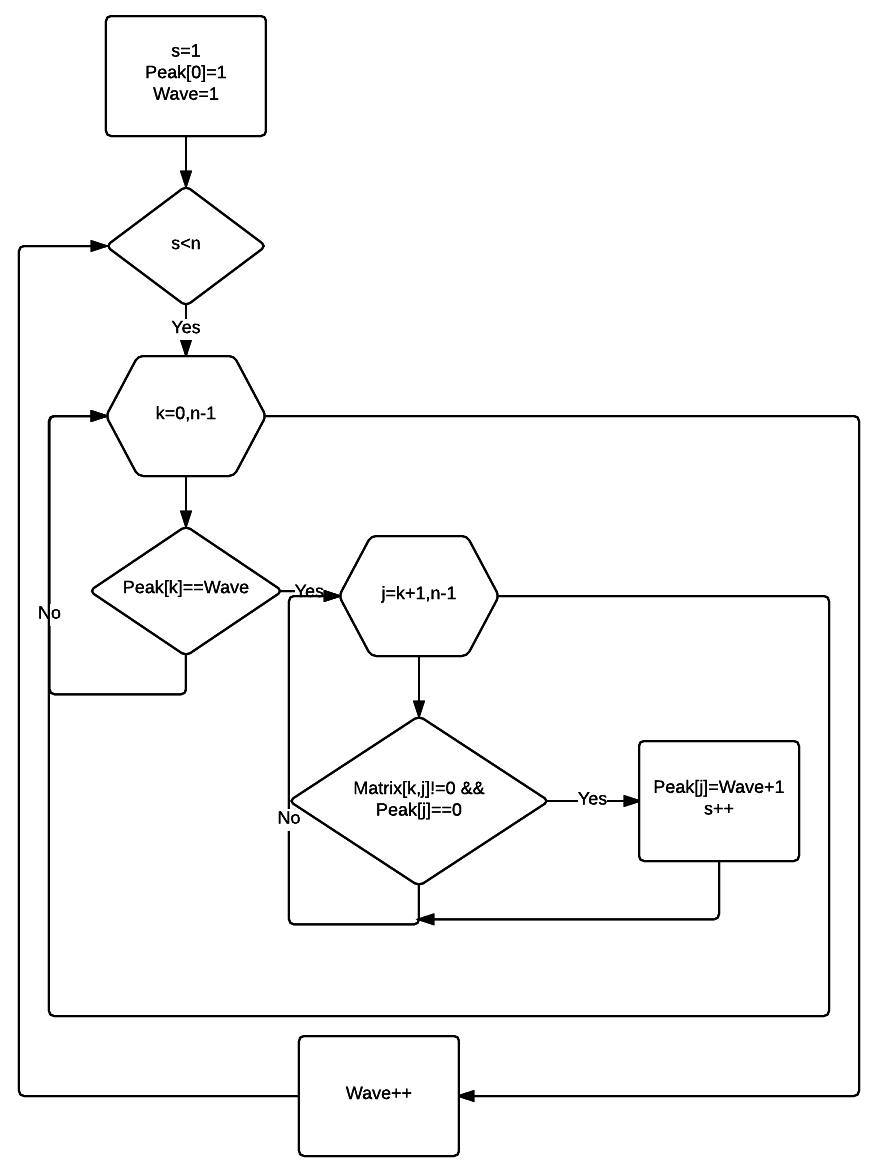
Вершины типа C содержат NULL. Речь идёт о массиве вершин, а не о матрице смежности. Вершины типа А и типа В содержат значения, отличные от нуля, а различие между ними в том, что значение вершин типа A равно текущему значению волны. Следовательно, на каждом шаге главного цикла необходимо просмотреть весь массив вершин и найти вершины, значение которых равно текущему значению волны. Цикл немного усложнится:



В этом фрагменте мы учли ещё несколько важных вещей:

1. В начале процесса нулевая вершина содержит единичное значение высоты. Будем для упрощения считать, что процесс распространения начинается с нулевой вершины.
2. Высота волны начинается с единицы и после пересчёта, на следующей итерации, увеличивается на единицу.

Найдя очередную вершину-источник, мы должны найти все смежные ей вершины и передать им значение высоты волны на единицу больше текущего. Это единственный момент, когда становится нужна матрица смежности. По номеру вершины определяем строку, соответствующую ей, и проходим эту строку: каждый её элемент, не равный нулю, означает вершину, смежную данной. Единственное — из рассмотрения необходимо исключить вершины, до которых волна уже дошла: это вершины, уже имеющие ненулевое значение. Алгоритм приобретает следующий вид:



Это окончательное решение, но это не совсем то, что требуется. Изначально стояла задача получить кратчайший путь. Поэтому сейчас рассмотрим способ восстановления пути по полученному распределению высот и матрице смежности.

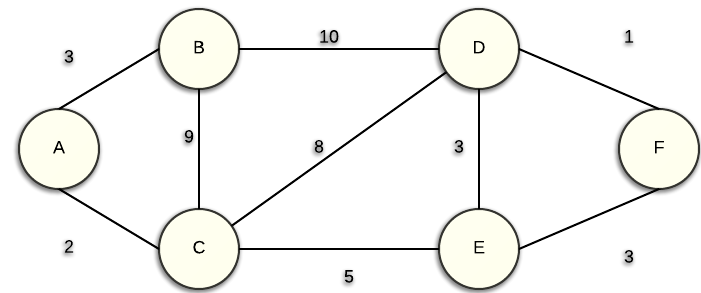
Предположим, что до некоторой вершины (назовем её *текущей*) путь уже построен, и необходимо найти вершину — *продолжение*. Пусть значение волновой метки в *текущей* вершине будет *Волна*. Тогда очевидно, что волна пришла в *текущую* вершину из вершины со значением высоты *Волна-1*. Всё, что теперь нужно, — это в соответствующей строке матрицы смежности найти все ненулевые элементы (они соответствуют смежным вершинам) и одну вершину, содержащую волновую метку со значением *Волна-1*. Это и будет вершина-*продолжение*.

Заметим, однако, что таких *продолжений* может быть несколько. Это естественно, так как кратчайший путь не обязан быть единственным. Задача поиска всех кратчайших путей несколько сложнее, но идея та же.

## Жадные алгоритмы

### Задача 1

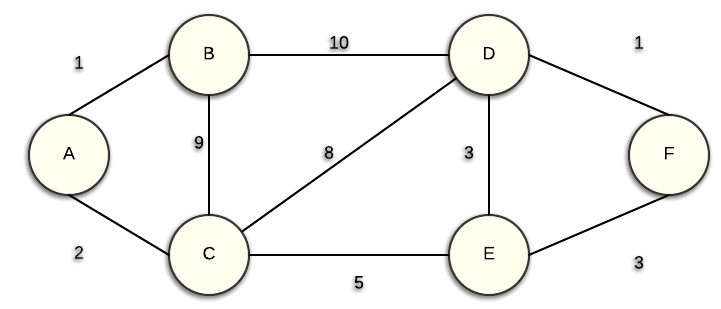
Известна схема дорог между несколькими городами. Числа на схеме обозначают расстояние. Нужно найти кратчайший маршрут из города A в город F.



Первая мысль, которая приходит в голову: на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы ещё не были. Для заданной схемы на первом этапе едем в город C, далее в E, затем в D и, наконец, в F. Общая длина маршрута равна 10.

Алгоритм, который мы применили, называется **жадным**. Его суть в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы жадный алгоритм даёт оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой «жадный» алгоритм даст маршрут A – B – C – E – F длиной 18, хотя существует более короткий маршрут A – C – E – F длиной 10.



Есть задачи, в которых жадный алгоритм всегда приводит к правильному решению. Одна из них — задача Прима – Краскала. Названа в честь Роберта Прима и Джозефа Краскала, которые независимо предложили её в середине XX века. Формулируется она так.

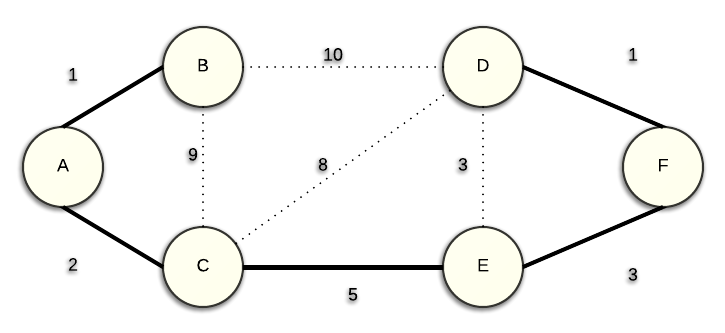
### Задача 2

В стране Лимонии есть N городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?

В теории графов эта задача называется задачей построения **минимального остовного дерева**, то есть дерева, связывающего все вершины. Остовное дерево для связного графа с N вершинами имеет N – 1 ребро. Рассмотрим жадный алгоритм решения этой задачи, предложенный Краскалом:

1. Начальное дерево пустое.
2. На каждом шаге к дереву добавляется ребро минимального веса, которое ещё не входит в дерево и не приводит к появлению цикла в дереве.

На рисунке показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии):



Здесь возможна такая последовательность добавления рёбер: AB, DF, AC, EF, CE. Обратите внимание, что после ребра EF следующим претендентом на добавление было ребро DE, но оно образовало цикл с рёбрами DF и EF, поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро ещё не включено в дерево и не образует цикла в нём. Есть очень красивое решение этой проблемы, основанное на раскраске вершин.

Сначала все вершины раскрашиваются в разные цвета, то есть им присваиваются разные числовые коды:

|  |
| --- |
| for(int i=0;i<N;i++) a[i]=i; |

Здесь N — количество вершин, a — целочисленный массив с индексами от 1 до N.

Затем в цикле N – 1 раз (именно столько рёбер нужно включить в дерево) выполняем следующие операции:

1. Ищем ребро минимальной длины среди всех рёбер, концы которых окрашены в разные цвета.
2. Найденное ребро (iMin,jMin) добавляется в список выбранных, и все вершины, имеющие цвет a[jMin], перекрашиваются в цвет a[iMin].

Фрагмент программы:

|  |
| --- |
| static void Main(string[] args)  {  int[,] W = new int[N,N]; *// Весовая матрица*  Load(W);    int[,] ostov = new int[N, 2]; *// Состояния вершин (просмотрена или не просмотрена)*  int[] a = new int[N];  int iMin = 0;  int jMin = 0;  for(int k = 0; k < N - 1; k++)  { *//Поиск ребра с минимальным весом*  int min = Int32.MaxValue;  for (int i = 0; i < N; i++)  {  for (int j = 0; j < N; j++)  {  if (a[i] != a[j] && W[i, j] < min)  {  iMin = i;  jMin = j;  min = W[i, j];  }  }  }  *// Добавление ребра в список выбранных*  ostov[k, 0] = iMin;  ostov[k, 1] = jMin; *// Перекрашивание вершин*  int jM = a[jMin], iM = a[iMin];  for (int i = 0; i < N; i++)  {  if (a[i] == jM)  a[i] = iM;  }  } *// Вывод результата*  for(int i = 0; i < N - 1; i++)   Console.WriteLine($"({ostov[i, 0]}, {ostov[i, 1]}");  } |

Здесь W — целочисленная матрица размера N × N; ostov — целочисленный массив из N – 1 строк и двух столбцов для хранения выбранных рёбер. Для каждого ребра хранятся номера двух вершин, которые оно соединяет.

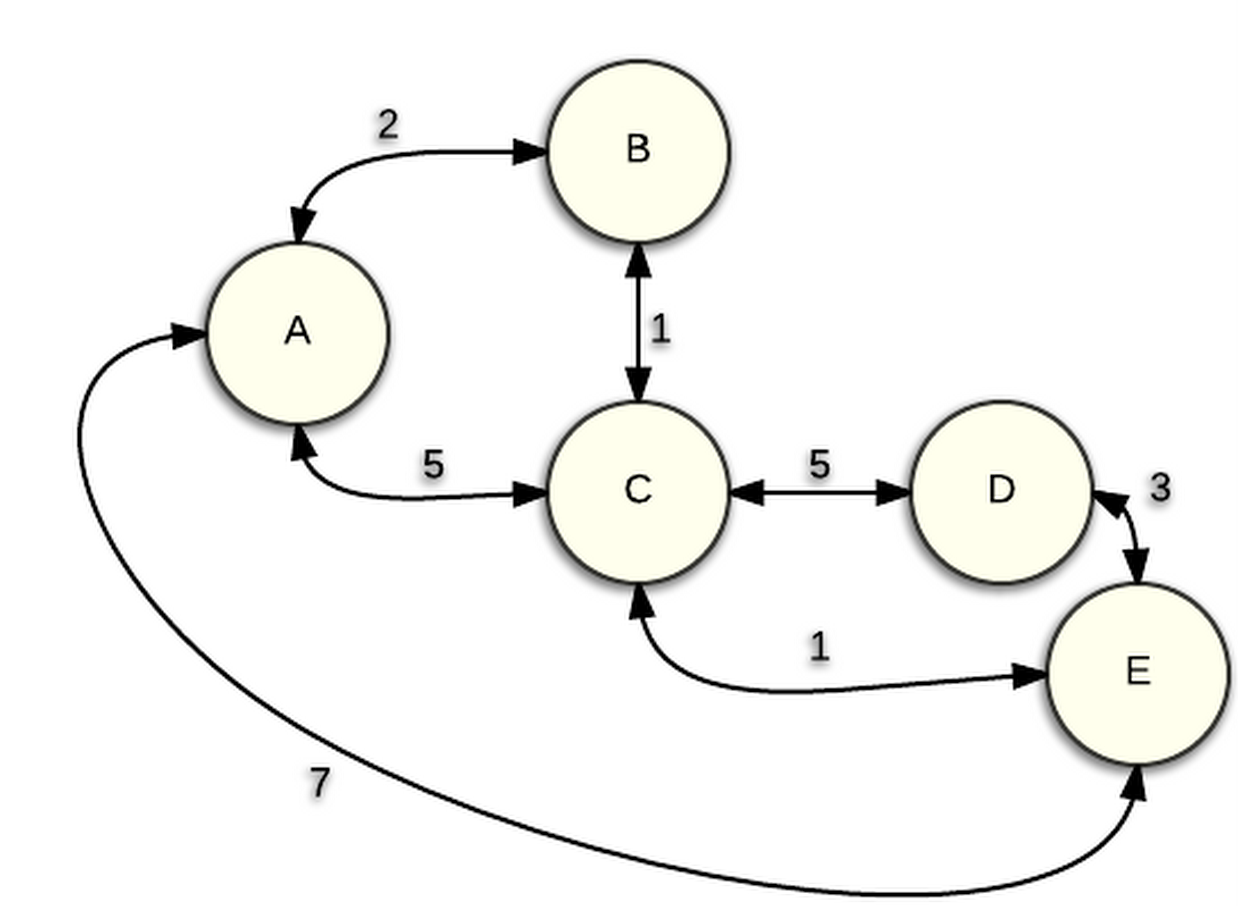
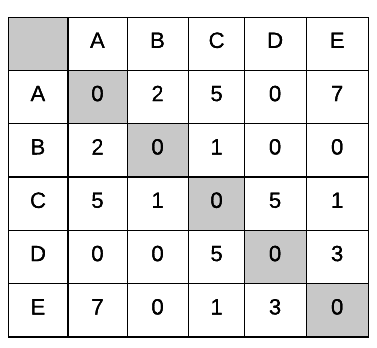
После окончания цикла остаётся вывести результат — рёбра из массива ostov.

## 

## Кратчайшие маршруты

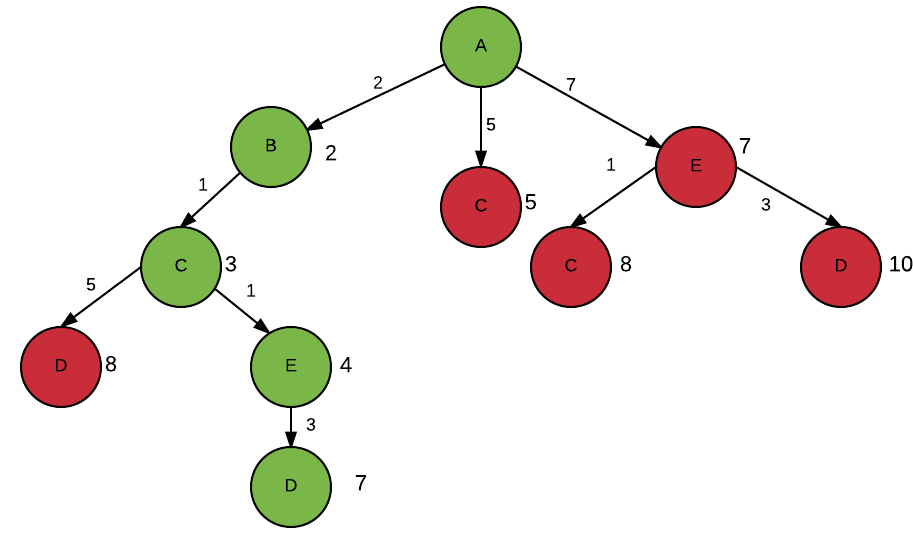
В 1960 году Эдсгер Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех рёбер (расстояния между вершинами) положительные.

Решим задачу для начала теоретически. Пусть у нас задана матрица смежности и требуется найти кратчайший путь из пункта A в E.



Для этого изобразим дерево возможных маршрутов. Из А мы можем попасть в пункты B(2), E(7) и C(5). Из пункта B мы можем попасть в пункт C(3), а из E в С(8). Так как у нас есть возможность попасть в C более коротким путём из B, то другие варианты мы больше не рассматриваем (закрашены красным цветом). Из E(7) мы можем попасть в D(10), а из C(3) — в D(8) и E(4). Так как мы видим, что есть более короткий путь в E(4) из C(3), то E(7) более не рассматриваем. Отсюда самый короткий путь — A – B – C – E.

Стоить отметить, что алгоритм Дейкстры не ищет самый короткий путь из точки A в точку E, а находит самый короткий путь по связанному графу, то есть находит все возможные пути. Поэтому на рисунке ниже также изображён путь из точки E в точку D.



Алгоритм Дейкстры использует дополнительные массивы: в одном из них (назовём его R) хранятся кратчайшие на данный момент расстояния от исходной вершины до каждой из вершин графа, а во втором (массив P) — вершина, из которой нужно «приехать» в эту вершину.

Сначала записываем в массив R расстояния от исходной вершины A до всех вершин, а в соответствующие элементы массива P — вершины A.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A(0)** | **B(1)** | **C(2)** | **D(3)** | **E(4)** |
| **R** | 20000000 | 2 | 5 | 20000000 | 7 |
| **P** | 0 | 0 | 0 | 0 | 0 |

2 000 000 означает, что прямого пути из вершины A(0) в эту вершину нет. Начинаем просмотр с P[0], записывая туда 0.

Из оставшихся вершин находим вершину с минимальным значением в массиве R: это вершина B(1). Теперь проверяем пути, проходящие через эту вершину: нет ли возможности добраться до других вершин через неё более коротким путём.

|  |
| --- |
| *// Проверка маршрута через вершину kMin* for(j = 0; j < N; j++) *// Просматриваем расстояние получившегося маршрута*   if (R[kMin] + W[kMin , j] < R[j] && W[kMin, j] != MaxInt)  {  R[j] = R[kMin] + W[kMin][j];  P[j] = kMin;  } |

Действительно, через B до C расстояние составляет всего 3. Поэтому мы в R под номером 3 помещаем 3, а в P под номером 3 помещаем 2.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A(0)** | **B(1)** | **C(2)** | **D(3)** | **E(4)** |
| R | 2000000 | 2 | 3 | 2000000 | 7 |
| P | 0 | 0 | 1 | 1 | 0 |

Проверяем наш граф: путь из A в C через B короче, поэтому записываем его в C(2).

Алгоритм Дейкстры можно рассматривать как своеобразный жадный алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается такая вершина X, что длина пути от A до X минимальна, если ехать только через уже выбранные вершины. Однако можно доказать, что это расстояние — действительно минимальная длина пути от A до X.

Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом X — ближайшая невыбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в X, проходящие через ещё не выбранные вершины, будут длиннее, поскольку все рёбра имеют положительную длину. Таким образом, найденная длина пути из A в X — минимальная.

После завершения алгоритма, когда все вершины выбраны, в массиве R находятся длины кратчайших маршрутов.

В программе объявим константу и переменные:

|  |
| --- |
| int N = 5; int[,] W = new int[N,N]; *// Весовая матрица* Load(W); int[] active = new int[N]; *// Состояния вершин (просмотрена или не просмотрена)* int[] route = new int[N]; int[] peak = new int[N]; int i; int j; int min; int kMin = 0; |

В первой части программы присваиваем начальные значения. Сразу помечаем, что вершина 0 просмотрена (не активна), с неё начинается маршрут:

|  |
| --- |
| for (i = 0; i < N; i++)  {  active[i] = 1;  route[i] = W[0,i];  peak[i] = 0;  }  *// Сразу помечаем, что вершина A(0) просмотрена,* *// с неё начинается маршрут*  active[0] = 0; |

В основном цикле, который выполняется N – 1 раз, так что все вершины были просмотрены, среди активных вершин ищем вершину с минимальным соответствующим значением в массиве R и проверяем, не лучше ли ехать через неё:

|  |
| --- |
| for (i = 0; i < N - 1; i++)  { *// Среди активных вершин* *// ищем вершину с минимальным соответствующим значением в массиве R* *// и проверяем, не лучше ли ехать через неё:*  min = int.MaxValue;  for (j = 0; j < N; j++)  if (active[j] == 1 && route[j] < min)  {  min = route[j]; *// Минимальный маршрут*  kMin = j; *// Номер вершины с минимальным маршрутом*  }   active[kMin] = 0; *// Просмотрели эту точку* *// Проверка маршрута через вершину kMin* *// Есть ли более короткий путь*  for (j = 0; j < N; j++) *// Если текущий путь в вершину J (R[j]* *// больше, чем путь из найденной вершины (R[kMin] +* *// путь из этой вершины W[kMin][j], то*  if (route[j] >route[kMin] + W[j,kMin] &&  W[j,kMin] != int.MaxValue && active[j] == 1)  { *// мы запоминаем новое расстояние*  route[j] = route[kMin] + W[j,kMin]; *// и запоминаем, что можем добраться туда более* *// коротким путём в массиве P*  peak[j] = kMin;  }  } |

В конце программы выводим оптимальный маршрут в обратном порядке:

|  |
| --- |
| while (i != 0)  {  Console.Write($"{i} ");  i = peak[i];  } |

Примечание: для решения задачи можно также использовать стек, поместив туда найденные вершины. Тогда распечатка стека даст нам возможность вывести кратчайший путь.

## 

## Алгоритм Флойда — Уоршелла

Алгоритм Дейкстры находит кратчайшие пути из одной заданной вершины во все остальные. Найти все кратчайшие пути из любой вершины в любую другую можно с помощью алгоритма Флойда — Уоршелла. Он основан на той же самой идее сокращения маршрута: иногда путь бывает короче через промежуточные вершины, чем напрямую.

|  |
| --- |
| static void FloydWarshal(int[,] W)  {  for (int k = 0; k < N; k++)  for (int i = 0; i < N; i++)  for (int j = 0; j < N; j++)  if (W[i,k] + W[k,j] < W[i,j])  W[i,j] = W[i,k] + W[k,j];  } |

Мы N раз просматриваем матрицу, отыскивая самые короткие пути из одной точки в другую.

В результате исходная весовая матрица графа W размером N × N превращается в матрицу, хранящую длины оптимальных маршрутов. Чтобы найти сами маршруты, нужно использовать ещё одну дополнительную матрицу, которая выполняет ту же роль, что и массив Peak в алгоритме Дейкстры.

Программа целиком:

|  |
| --- |
| using System;  namespace Graph1 {  class Program  {  const int N = 5;  static void Load(int[,] W)  {  *// Тут ваша логика заполнения массива*  }    static void FloydWarshal(int[,] W)  {  for (int k = 0; k < N; k++)  for (int i = 0; i < N; i++)  for (int j = 0; j < N; j++)  if (W[i,k] + W[k,j] < W[i,j])  W[i,j] = W[i,k] + W[k,j];  }   static void Main(string[] args)  {  int[,] W = new int[N,N]; *// Весовая матрица*  Load(W);  int[] active = new int[N]; *// Состояния вершин (просмотрена или не просмотрена)*  int[] route = new int[N];  int[] peak = new int[N];  int i;  int j;  int min;  int kMin = 0; *// В начале программы присваиваем начальные значения* *// Если = 1, то вершина ещё не просмотрена*   for (i = 0; i < N; i++)  {  active[i] = 1;  route[i] = W[0,i];  peak[i] = 0;  }  *// Сразу помечаем, что вершина A(0) просмотрена,* *// с неё начинается маршрут*  active[0] = 0;  for (i = 0; i < N - 1; i++)  { *// Среди активных вершин* *// ищем вершину с минимальным соответствующим значением в массиве R* *// и проверяем, не лучше ли ехать через неё:*  min = int.MaxValue;  for (j = 0; j < N; j++)  if (active[j] == 1 && route[j] < min)  {  min = route[j]; *// Минимальный маршрут*  kMin = j; *// Номер вершины с минимальным маршрутом*  }   active[kMin] = 0; *// Просмотрели эту точку* *// Проверка маршрута через вершину kMin* *// Есть ли более короткий путь*  for (j = 0; j < N; j++) *// Если текущий путь в вершину J (R[j]* *// больше чем путь из найденной вершины (R[kMin] +* *// путь из этой вершины W[kMin][j], то*  if (route[j] >route[kMin] + W[j,kMin] &&  W[j,kMin] != int.MaxValue && active[j] == 1)  { *// мы запоминаем новое расстояние*  route[j] = route[kMin] + W[j,kMin]; *// и запоминаем, что можем добраться туда более* *// коротким путём в массиве P*  peak[j] = kMin;  }  }   i = N - 1;  while (i != 0)  {  Console.Write($"{i} ");  i = peak[i];  }  FloydWarshal(W);  }  } } |

# Заключение

Мы узнали, что такое граф. Выяснили, что граф можно записать с помощью массива или класса. Чтобы найти все кратчайшие пути из одной точки во все остальные, нужно использовать алгоритм Дейкстры. Но если вам захочется найти кратчайшие пути из всех точек во все остальные, тут вам поможет алгоритм Флойда — Уоршелла.

# 

# Практическое задание

Модифицируйте DFS и BFS из предыдущего урока, но уже для графа, также с выводом каждого шага

# Дополнительные материалы

[Graphonline.ru](http://graphonline.ru) — сервис для отображения графов, заданных матрицей смежности.

# Используемые источники

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. Д. Рихтер CLR via C# М.: Питер, 2019.