Урок 7

Графы

Алгоритмы на графах

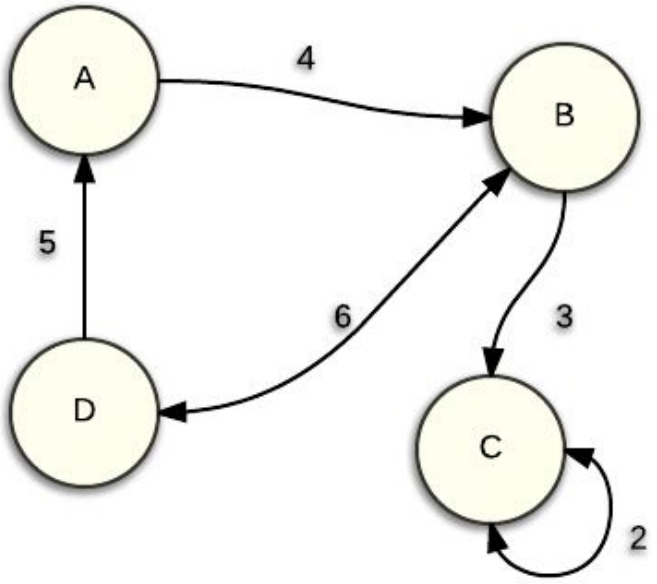
Графы

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

Единица на пересечении строки A и B обозначает связь. Ноль указывает на то, 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 дает никакой информации о том, как выглядит граф.

В примере все узлы связаны, т.е. между любой парой узлов существует путь — последовательность ребер, 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.

	A	B	C	D
A	0	4	0	0
B	0	0	3	6
C	0	0	2	0
D	5	6	0	0



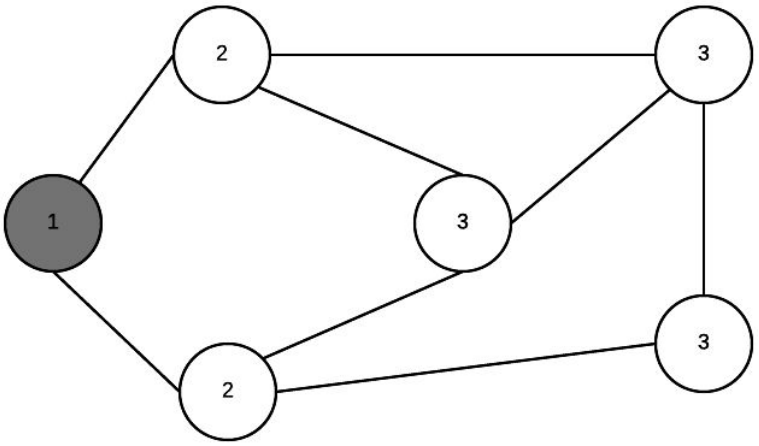
Если для каждого ребра указано направление, граф называют **ориентированным (орграфом)**. Ребра орграфа называют дугами. Его матрица смежности не всегда симметричная. Число, отличное от нуля, стоящее на пересечении строки A и столбца B, говорит о том, что существует путь из вершины A в вершину B.

Задание: напишите функцию, которая считывает граф из текстового файла в двумерный массив. Первым числом идет размер графа, далее сам граф:

```
4
0 4 0 0
0 0 3 6
0 0 2 0
5 6 0 0
```

Обход графа в ширину

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



Заметим одну особенность. Существуют вершины, в которые можно попасть на некотором шаге из двух и более вершин, но обход в ширину делает это единожды из какой-нибудь одной. Важный вопрос о выборе пути в вершину, достижимую из нескольких вершин фронта, не есть вопрос общего метода. Это содержательная проблема, решается она, исходя из потребности конкретной задачи.

Ключевым моментом является фронт волны. До завершения итерации множество вершин графа можно разделить на три подмножества: подмножество, через которое волна уже прошла, подмножество вершин, достигнутых на последнем шаге (волна пришла в вершину, но еще не ушла из неё), и подмножество вершин, еще не достигнутых волной. Для построения пути волны нужно именно множество фронта. Это множество может быть представлено обычным массивом, перестраиваемым 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. Введем 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:

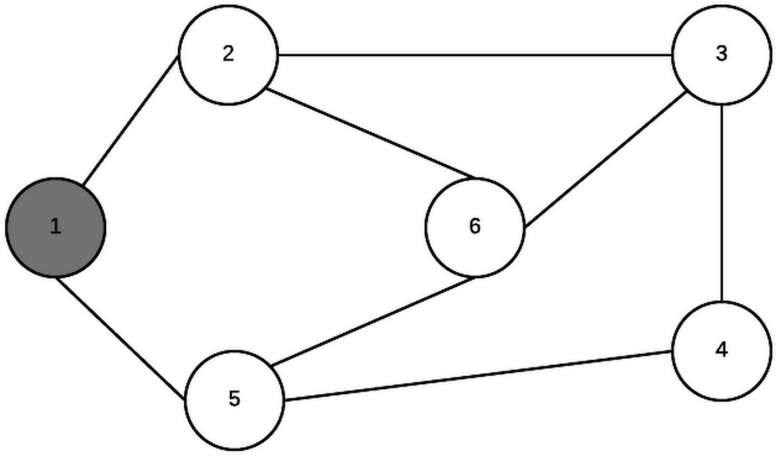
```
Обход всех вершин графа
        Если статус Текущей Вершины = 2, то
                Для всех вершин смежных Текущей
                        Если статус смежной=1, то Статус смежной = 2
        Статус Текущей = 3
```

Если есть фрагмент, описывающий очередную итерацию, то весь процесс можно построить как цикл, условием завершения которого будет отсутствие вершин со статусом ниже 3, а начинается процесс с некоторой 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, и 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 все они 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 приведен 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 обхода всех вершин или рёбер, а полный перебор всех путей по графу, в этом случае полный путь есть некоторое дерево.

Задание для самостоятельной работы

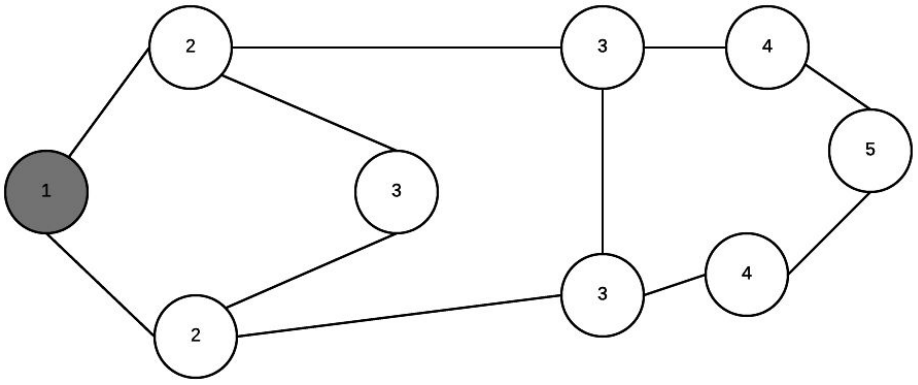
Постройте дерево перебора всех путей в глубину для графа.

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 B пришла из вершины A, при этом в вершине A значение волновой метки равно H, то в вершине B значение волновой метки определяется как H+1.

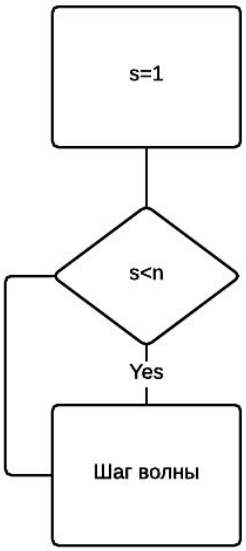
Таким образом, значения волновых меток зависят от длины пути, пройденного волной. Следовательно, можно утверждать, что чем больше значение волновой метки, тем больший путь был пройден волной до данной вершины. Заметим, что волновой алгоритм использует стратегию обхода графа в ширину. На рисунке показан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



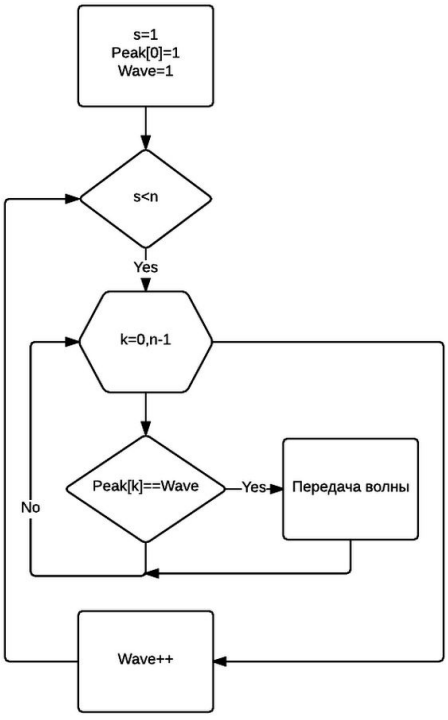
Реализация

Представим граф матрицей смежности. После прохождения волны граф превращается во взвешенный. При этом веса присваиваются вершинам графа, а какая-либо информация о вершинах никаким образом в матрице смежности не отображается. Это наводит на мысль о необходимости дополнительной структуры данных, сохраняющей информацию о проходящей волне. В качестве такой структуры вполне подойдет одномерный массив. 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, процесс прохождения волны можно смоделировать циклом, работающим до тех пор, пока волна не заполнит весь граф. Если создать счетчик, считающий количество «передач волны», то, видимо, условие завершения цикла будет таким:



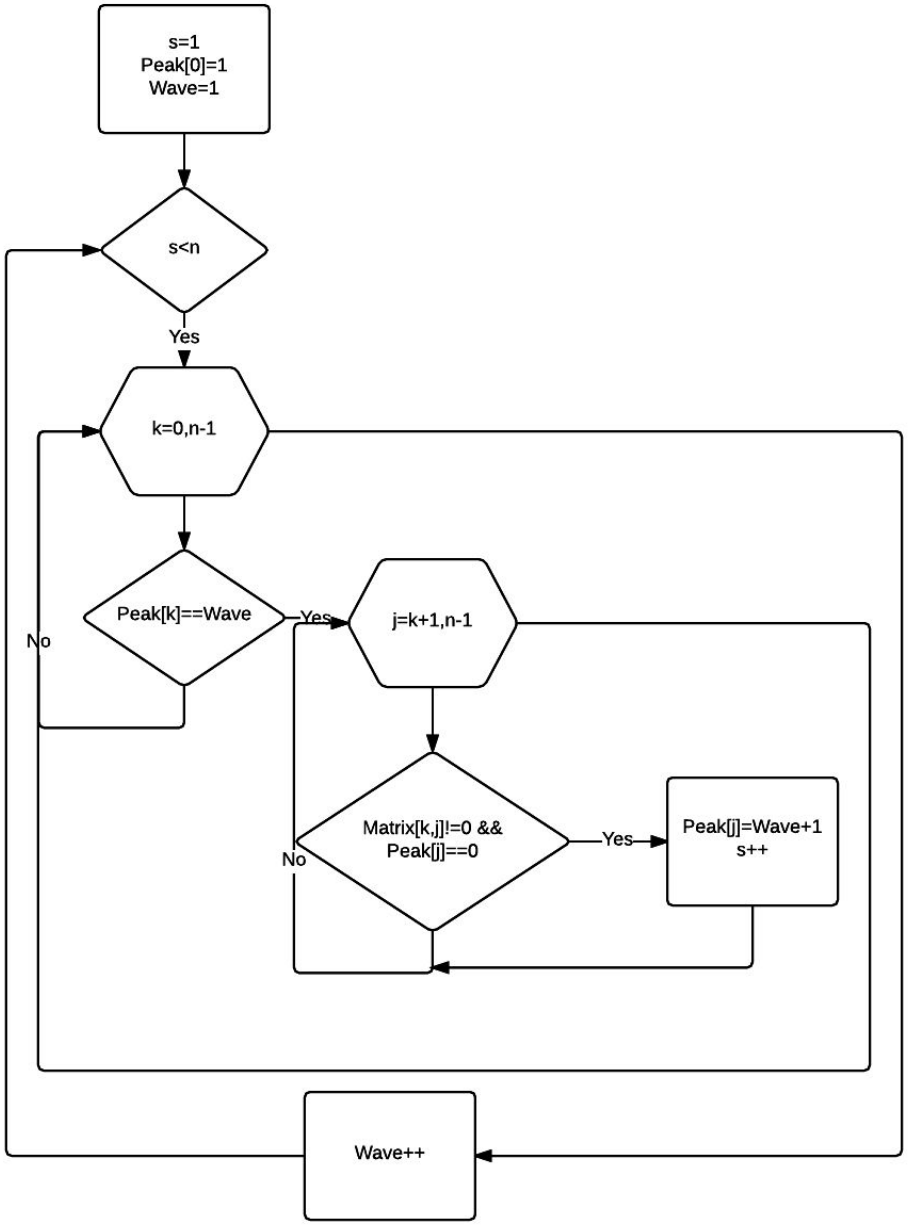
Условие предполагает, что вершины нумеруются с нуля, и n — их количество. В некоторые из вершин графа волна пришла на предыдущем шаге, и они становятся источником для дальнейшего движения. Назовем их вершинами типа A. Некоторые из вершин уже были источниками — это тип B, и некоторые еще ждут волны — тип C. Вершины типа C содержат NULL (речь идет о массиве вершин, а не о матрице смежности). Вершины типа A и типа B 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 еще некоторые важные вещи. А именно:

1. В начале процесса нулевая (будем для упрощения считать, что процесс распространения начинается с нулевой вершины) вершина содержит единичное значение высоты.
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, ей соответствующую, 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 не вполне 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*.

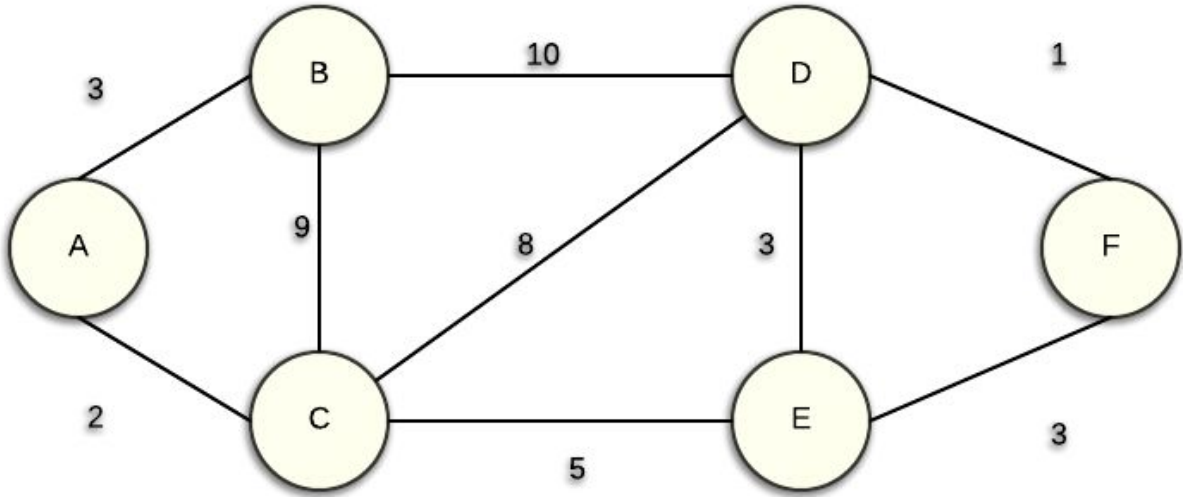
Заметим, однако, что таких *продолжений* может быть несколько. Это естественно, так как кратчайший путь не обязан быть единственным. Задача поиска всех кратчайших путей несколько сложнее, но идея та же.

Задание: реализуйте алгоритм в качестве программы на языке C.

«Жадные» алгоритмы

Задача 1

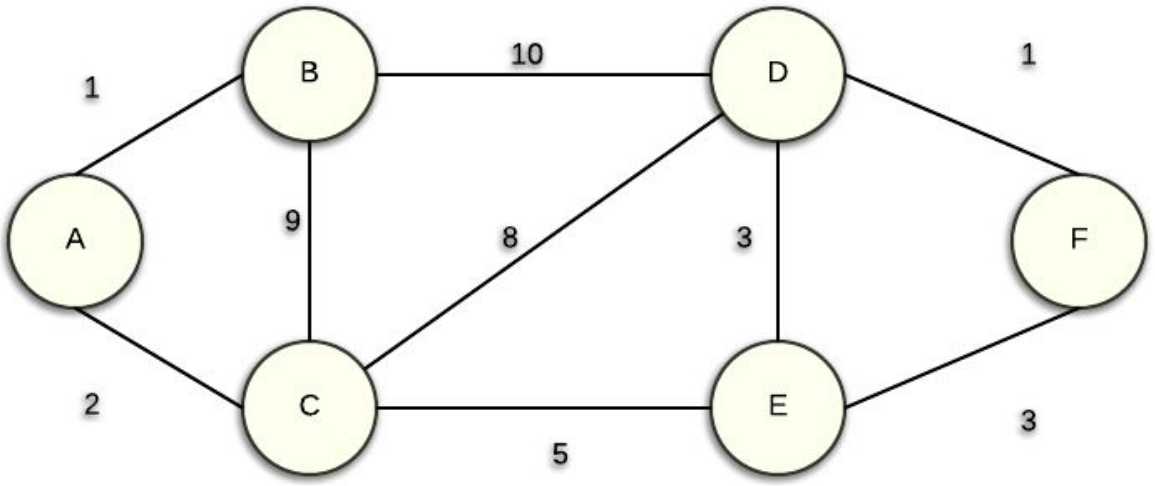
Известна схема дорог между несколькими городами. Числа на схеме обозначают расстояние. Нужно найти кратчайший маршрут из города A в город F.



Первая мысль, которая приходит в голову: на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы еще 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 дает 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.



Однако есть задачи, в которых «жадный» алгоритм всегда приводит к правильному решению. Одна из них — задача Прима-Крускаля (названа в честь Роберта Прима и Джозефа Крускаля, которые независимо предложили её в середине XX века) формулируется так:

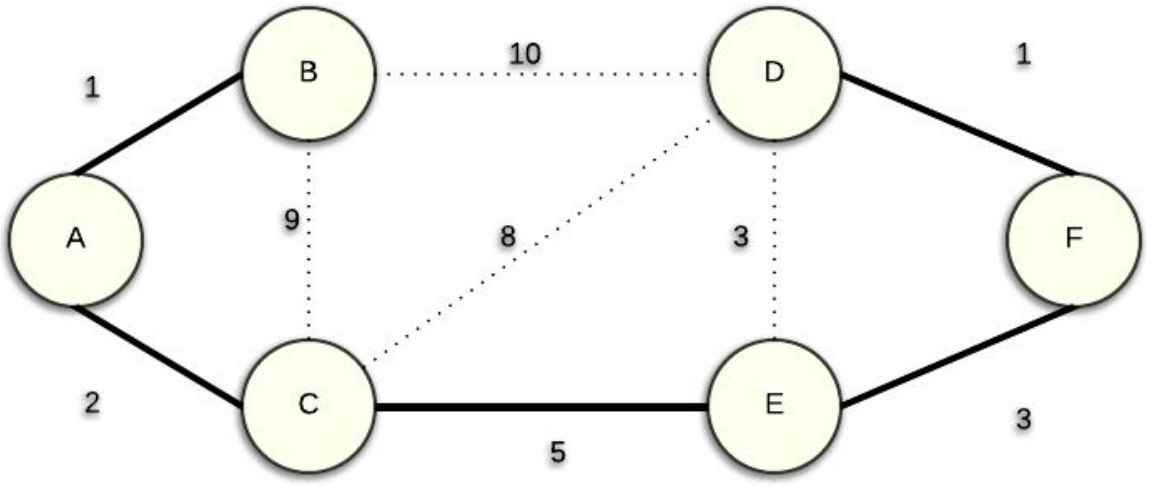
Задача 2

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 Крускалом:

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 ребер: AB, DF, AC, EF, CE. Обратите внимание, что после ребра EF следующим претендентом на добавление являлось ребро DE, но оно образовало цикл с ребрами DF и EF, поэтому не было включено в дерево.
При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро еще не включено в дерево и не образует цикла в нём. Существует 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:

```
#include <stdio.h>
#define MAXINT 2147483647

int main(int argc, char *argv[])
{
     for(int k = 0; k < N - 1; k++)
   {
   //поиск ребра с минимальным весом
      int min = MAXINT;
      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;
    }
    // Выводим результат
    for(int i = 0; i < N - 1; i++)
        printf("(%d,%d)", ostov[i, 0], ostov[i, 1]);
    return 0;
}
```

Здесь W — целочисленная матрица размера NxN; 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 остается вывести результат — рёбра из массива ostov.

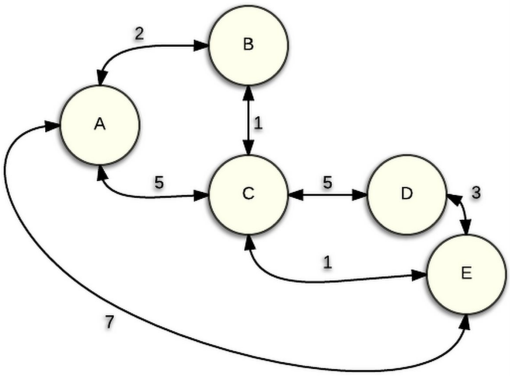
Кратчайшие маршруты

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 для начала задачу теоретически.

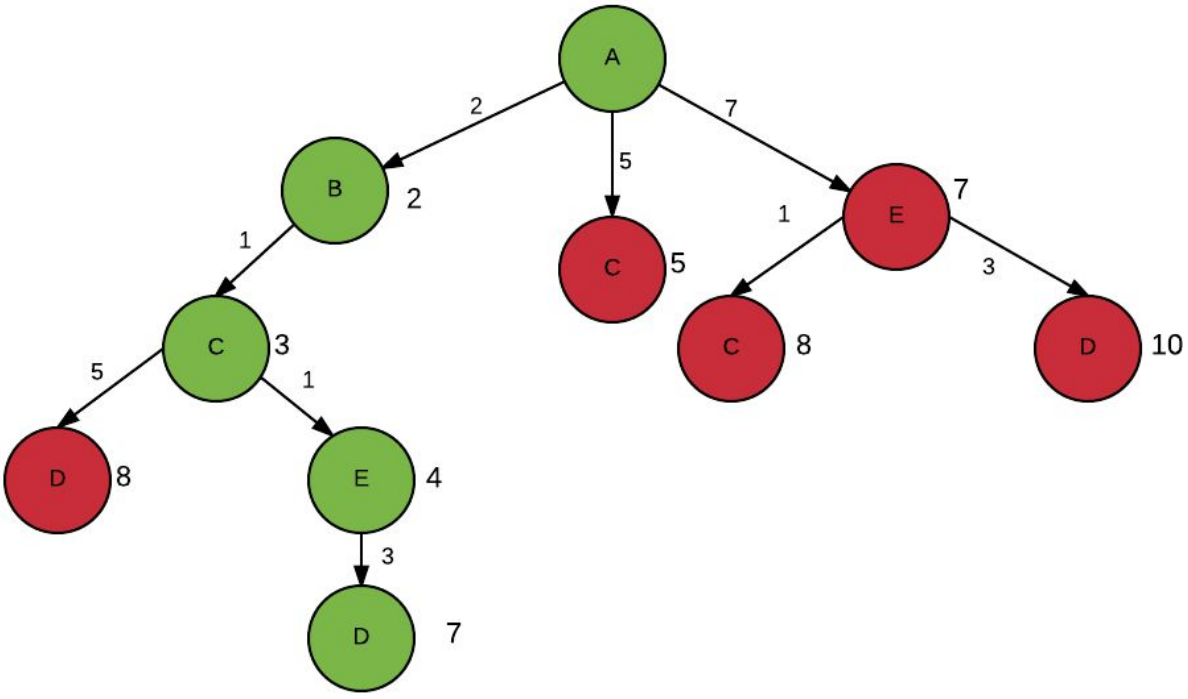
Пусть у нас задана матрица смежности, и требуется найти кратчайший путь из пункта A в E.

	A	B	C	D	E
A	0	2	5	0	7
B	2	0	1	0	0
C	5	1	0	5	1
D	0	0	5	0	3
E	7	0	1	3	0



Для этого изобразим дерево возможных маршрутов. Из A 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 C(8). Так как у нас есть возможность попасть в C более коротким путем из B, то другие варианты мы более 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 это A-B-C-E. Стоить отметить, что алгоритм Дейкстра 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 изображен путь из точки E в точку D.



Алгоритм Дейкстра использует дополнительные массивы: в одном из них (назовем 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 данную 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

2000000 означает, что прямого пути из вершины A(0) в данную 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 полу
   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: ехать из A в C через B, короче, поэтому записываем этот путь в C(2).

Алгоритм Дейкстра 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 минимальная, 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:

```
const int N = 6;
int W[N, N];          // Весовая матрица
int active[N] = {0};// состояния вершин (просмотрена или нет)
                      // Если = 0 то вершина еще не просмотрена
int R[N], P[N];
int i, j, min, kMin;
```

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] = -1;
}
// сразу помечаем, что вершина A(0) просмотрена,
// с нее 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 нее:
   system("cls");
   printMatrix(W);
   printInfo(Peak, Route);
   min = MaxInt;
   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 маршрут через вершину kMin
   // Есть ли путь, более короткий
   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] != MaxInt && active[j] == 1)
      {
   // мы запоминаем новое расстояние
         Route[j] = Route[kMin] + W[j][kMin];
   // и запоминаем, что можем добраться туда более
   // коротким путем в массиве P
         Peak[j] = kMin;
         printMatrix(W);
         printInfo(Peak,Route);
      }
}
```

В конце программы выводим оптимальный маршрут в обратном порядке:

```
i=N-1;
while(i!=-1)
{
    printf("%c ",i+65);
    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 Дейкстра 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, основанного на той же самой идее сокращения маршрута (иногда бывает короче ехать через промежуточные вершины, чем напрямую):

```
void FloydWarshal(int W[N][N])
{
   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 x N превращается в матрицу, хранящую длины оптимальных маршрутов. Для того чтобы найти сами маршруты, нужно использовать еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив Peak в алгоритме Дейкстра.

Программа целиком:

```
#include <stdio.h>
#include <stdlib.h>

const int N = 5;
const char* filename = "D:\\TEMP\\data.txt";
const int MaxInt = 20000000;

int load(int W[N][N])
{
   FILE* file = fopen(filename, "r");
   if (file == NULL)
   {
      printf("Can't open file");
      exit(1);
   }
   for(int i = 0; i < N; i++)
      for(int j = 0; j < N; j++)
```

```
        {
            int a;
            fscanf(file, "%d", &a);
            if (a == 0)
                W[i][j] = MaxInt;
            else
                W[i][j] = a;
        }
    fclose(file);
    return 0;
}

void printMatrix(int W[N][N])
{
    printf("%s", "      ");
    for(int i = 0; i < N; i++)
        printf("%c(%d) ", 65 + i, i);
    printf("\n");
    for(int i = 0; i < N; i++)
    {
        printf("%c(%d)", 65 + i, i);
        for(int j = 0; j < N; j++)
            printf("%5d", (W[i][j] == MaxInt) ? 0 : W[i][j]);
        printf("\n");
    }
}

void printInfo(int P[N], int R[N])
{
    printf("P:\n");
    for(int i = 0; i < N; i++)
        printf("%c(%d) %c(%d)\n", P[i] + 65, P[i], i + 65, i);
    printf("R:\n");
    for(int i = 0; i < N; i++)
        printf("%c%10d\n", i + 65, R[i]);
}

void FloydWarshal(int W[N][N])
{
    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];
}

int main(int argc, char *argv[])
{
    int W[N][N];                    // Весовая матрица
    load(W);
    int active[N];                  // состояния вершин (просмотрена или не просмотрена)
    int Route[N], Peak[N];
    int i, j, min, kMin;
    // В начале программы присваиваем начальные значения,
    // Если =1 то вершина еще 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 нее 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 нее:
        system("cls");
        printMatrix(W);
        printInfo(Peak, Route);
       // system("pause");
        min = MaxInt;
        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 маршрут через вершину kMin
        // Есть ли путь, более короткий
        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] != MaxInt && active[j] == 1)
            {
        // мы запоминаем новое расстояние
                Route[j] = Route[kMin] + W[j][kMin];
        // и запоминаем, что можем добраться туда более
        // коротким путем в массиве P
                Peak[j] = kMin;
                printMatrix(W);
                printInfo(Peak, Route);
            }
    }
    i = N - 1;
    while(i != 0)
    {
        printf("%c ", i + 65);
        i = Peak[i];
    }
    printf("A");
    FloydWarshal(W);
    printf("\n");
    printMatrix(W);
    return 0;
}
```

Домашняя работа

1. Написать функции, которые считывают матрицу смежности из файла и выводят ее на экран
2. Написать рекурсивную функцию обхода графа в глубину.
3. Написать функцию обхода графа в ширину.
4. *Создать библиотеку функций для работы с графами.

Дополнительные материалы

1. http://graphonline.ru/ - сервис для отображения графов заданных матрицей смежности

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Род Стивенс, Алгоритмы. Теория и практическое применение. Издательство «Э», 2016.
2. Никлаус Вирт, Алгоритмы и структуры данных. Новая версия для Оберона, ДМК, Москва, 2010.
3. Потопахин В.В., Искусство алгоритмизации, ДМК, 2011.
4. Учебник «Информатика-11 класс. Часть 2. Углубленный уровень», К.Ю. Поляков, Е.А. Еремин, Бином, 2013.