Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Вятский государственный гуманитарный университет»

Дополнительная подготовка школьников по дисциплине «Информатика и информационные технологии»

Учебный модуль
Графы. Базовые алгоритмы

С. Ю. Иванов

Киров
2011

СОДЕРЖАНИЕ

1. Понятие графа. Ориентированные и неориентированные графы 3
1.1. Основные понятия 3
1.2. Способы описания 3
2. Поиск в глубину 7
2.1. Основная идея 7
2.2. Определение компонент связности 8
2.3. Поиск всех путей 10
2.4. Задача об одностороннем движении 11
3. Поиск в ширину 14
3.1. Основная идея 14
3.2. Поиск кратчайшего пути 15
4. Игры на графах 17
5. Задания для самостоятельного решения 19
6. Заключение 20
Литература 21

1. Понятие графа. Ориентированные и неориентированные графы

1.1. Основные понятия

Граф можно определить как множество вершин V и набор E неупорядоченных и упорядоченных пар вершин. Тогда граф обозначим $G – (V, E)$. Количество элементов в множествах V и E будем обозначать буквами N и M. Неупорядоченная пара вершин называется *ребром*, а упорядоченная пара – *дугой*. Ребро между вершинами i и j означает, что вершина i связана с j, и вершина j связана с i, то есть их связь двусторонняя. Дуга говорит об односторонней связи между вершинами i и j, например, что мы можем попасть в вершину j из вершины i, но в вершину i из вершины j мы перейти не можем.

Граф, содержащий только ребра, называется *неориентированным*; граф, содержащий только дуги, – *ориентированным*, или *орграфом*.

Вершины, соединенные ребром, называются *смежными*. Ребра, имеющие общую вершину, также называются смежными. Ребро и любая из его двух вершин называются *инцидентными*. Говорят, что ребро (i, j) соединяет вершины i и j.

Любой граф можно представить на плоскости множеством точек (вершин), которые соединены линиями (ребрами). На плоскости при представлении графа вполне возможно, что некоторые ребра будут пересекаться, что может затруднять восприятие графа. В трехмерном пространстве любой граф можно представить таким образом, что линии (ребра) не будут пересекаться.

1.2. Способы описания

Для рассмотрения алгоритмов способ описания графа, как правило, не имеет значения, однако для разработки эффективной программы выбор структуры данных для представления графа может иметь принципиальное значение.

При рассмотрении различных способов представления графа будем использовать пример графа на рис. 1.

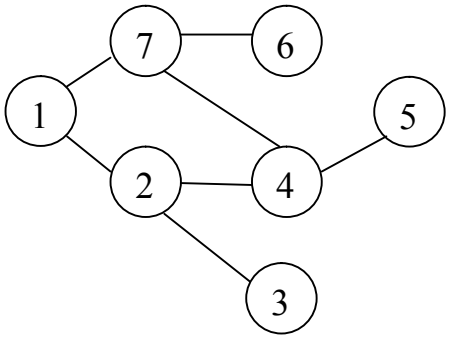

Рис. 1. Неориентированный граф

Чаще всего используют четыре способа описания графа: матрица смежности, матрица инцидентности, списки связи, перечень ребер.

Матрица смежности. Представляет собой двумерный массив размерности $N \times N$, в котором $a[i, j]$ содержит информацию о том, есть ли ребро между вершинами i и j, то есть являются вершины i и j смежными. Например, если вершины i и j соединены ребром, то $a[i, j] = 1$, иначе если ребра между вершинами нет, то $a[i, j] = 0$.

Матрица смежности для графа на рис.1 будет выглядеть следующим образом:

0	1	0	0	0	0	1
1	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	0	1	0	1
0	0	0	1	0	0	0
0	0	0	0	0	0	1
1	0	0	1	0	1	0

Следует обратить внимание, что для неориентированного графа матрица смежности симметрична относительно главной диагонали, то есть $a[i, j] = a[j, i]$. Для ориентированного графа это условие может не выполняться, так если $a[i, j] = 1$ и $a[j, i] = 0$, то это означает, что из вершины i мы можем попасть в j, а из j в i не можем.

В *матрице инцидентности* указываются связи между инцидентными элементами графа – ребром и вершиной. Данные хранятся также в двумерном массиве, столбцы которого – это ребра, а строки – вершины. В каждом столбце находится ровно две единицы, т. к. каждое ребро задается парой вершин. Таким образом, при большом количестве вершин такой способ представления является неэффективным. Кроме того матрицы инцидентности нельзя использовать для графов с петлями (вершина соединена ребром сама с собой).

1	0	0	0	1	0	0
1	1	1	0	0	0	0
0	1	0	0	0	0	0
0	0	1	1	0	1	0
0	0	0	1	0	0	0
0	0	0	0	0	0	1
0	0	0	0	1	1	1

Перечень ребер, так же как матрица инцидентности, задает список ребер, но для этого используется двумерный массив меньшей размерности. Каждое ребро задается парой чисел, обозначающих номера вершин, принадлежащих ребру. Граф на рис.1 может быть представлен в следующем виде:

1	2
2	3
2	4
4	5
1	7
4	7
6	7

Списки связи. Используется одномерный массив, элементами которого являются указатели на списки с номерами вершин, которые связаны с *i*-й вершиной. Рассмотрим, как будет выглядеть граф на рис. 1 в виде списков связи:

1	2	
1	3	4
2		
2	5	7
4		
7		
1	4	6

В дальнейшем мы будем использовать матрицу смежности для представления графа. Приведем описание констант и типов данных, которые нам потребуются.

```
const NMax = 100; {максимальное количество вершин}
type graph = array[1..NMax, 1..NMax] of boolean; {матрица
смежности}
```

```
     mas = array[1..NMax] of boolean; {массив для отметки,
какие вершины были посещены}
var a: graph;      {граф, представленный матрицей смежности}
    visited: mas; {массив, посещенных вершин}
    N: integer;   {количество вершин в графе}
```

2. Поиск в глубину

2.1. Основная идея

Поиск в глубину (англ. *Depth-first search, DFS*) является одним из методов обхода графа. Он также известен под другими названиями, например, бэктрекинг (англ. *backtracking*), поиск с возвратом. Идея данного алгоритма уже знакома нам из раздела «Перебор с возвратом», рассмотрим её сейчас для рекурсивного обхода графа.

Пусть дан граф, приведенный на рис. 2.

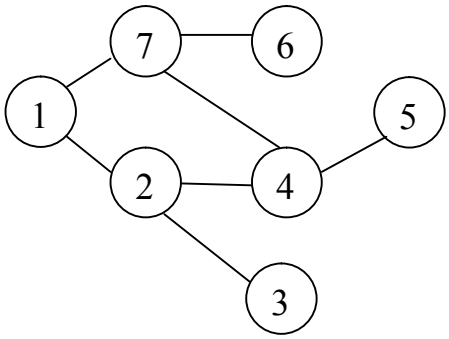

Рис. 2. Поиск в глубину

Суть алгоритма заключается в следующем: на каждом шаге для текущей вершины перебираем все смежные с ней вершины и для каждой из них рекурсивно повторяем все действия, при этом запоминаем в некоторой структуре, какие вершины уже были посещены. То есть из текущей вершины идём в первую смежную с ней, но пока еще посещённую, до тех пор, пока это возможно (идём в глубину графа). Если для текущей вершины нет смежных или они были все просмотрены, то возвращаемся на уровень вверх (к предыдущей вершине) и продолжаем перебор смежных с ней вершин.

Приведем возможную реализацию алгоритма:

```
procedure dfs(var a:graph; var visited: mas; n, v:
integer);
  var i: integer;
begin
  visited[v] := true; {помечаем текущую вершину как
посещенную}
  for i := 1 to n do {перебираем все вершины смежные с v,
которые еще не были посещены}
    if a[v, i] and not visited[i] then
      dfs(a, visited, n, i); {повторяем алгоритм для i-й
вершины}
end;
```

Массив `visited` используем, чтобы пометить *i*-ую вершину как посещенную. Так для текущей вершины *v* ищем не просто смежные вершины, а только те, которые еще не были просмотрены.

Предположим, что мы начали поиск с первой вершины, вызов процедуры будет выглядеть следующим образом:

```
dfs(a, visited, n, 1);
```

Порядок посещения вершин для графа на рис. 2 будет таким:

1 2 3 4 5 7 6.

Если запустить обход графа из вершины 4, то порядок обхода будет следующим:

4 2 1 7 6 3 5.

Поскольку мы использовали матрицу смежности для задания графа, то оценка временной сложности данного алгоритма будет $O(N^2)$: нам нужно посетить N вершин, и для каждой вершины мы перебираем N вершин.

Можно реализовать и нерекурсивный вариант данной процедуры, для этого придется использовать структуру данных *стек* для запоминания порядка обхода вершин. Оставим это задание на самостоятельное выполнение.

2.2. Определение компонент связности

Одна из задач, для решения которых может применяться поиск в глубину – это выявление компонент связности.

Граф называется *связным*, если для любых двух его вершин имеется путь, соединяющий эти вершины. Если граф несвязен, то он состоит из нескольких связных частей, они называются *компонентами связности*. Ориентированный граф называется *сильно связным*, если из каждой вершины в любую другую имеется (ориентированный) путь.

Для нахождения компонент связности нам потребуется либо дополнительный массив, в котором мы будем хранить информацию о том, какой компоненте принадлежит *i*-ая вершина, либо использовать для этих целей массив `visited`, но тогда придется изменить его описание, заменив тип `boolean` на `integer`:

```
mas = array[1..NMax] of integer;
```

Соответствующим образом изменим и процедуру поиска в глубину, в массив `visited` будем помещать номер компоненты связности, которой принадлежит *i*-ая вершина, для этого добавим в процедуру еще один параметр *c* – номер компоненты связности.

```
procedure dfs(var a:graph; var visited: mas;
                  n, c, v: integer);
  var i: integer;
begin
  visited[v] := c; {помечаем текущую вершину как посещенную
и принадлежащую c-й компоненте связности}
  for i := 1 to n do {перебираем все вершины смежные с v,
которые еще не были посещены}
    if (a[v, i]) and (visited[i]=0) then
      dfs(a, visited, n, c, i); {повторяем алгоритм для i-й
вершины}
end;
```

Очевидно, что вызвав процедуру `dfs` один раз, мы найдем лишь вершины, принадлежащие одной компоненты связности. Для нахождения всех компонент связности нам нужно вызвать `dfs` для каждой еще непросмотренной вершины графа. Реализовать это можно следующим образом:

```
procedure connectedParts(var a:graph; var visited: mas;
                                 n: integer);
  var i, c: integer;
begin
  c:=1; {номер очередной компоненты связности}
  for i:=1 to n do
    if visited[i]=0 then begin {если вершина еще не была
посещена, начинаем из нее поиск новой компоненты связности}
      dfs(a, visited, n, c, i); {просматриваем все вершины,
достижимые из i-й}
      Inc(c); {увеличиваем номер компоненты связности}
    end;
end;
```

По завершении процедуры `connectedParts`, массив `visited` будет содержать данные о том, какой компоненте связности принадлежит *i*-ая вершина.

Следует отметить, что для решения данной задачи поиск в глубину не обязателен, можно использовать любой способ обхода графа, например, поиск в ширину.

2.3. Поиск всех путей

Путь в графе – это последовательность вершин $x_1, x_2, \ldots, x_k$, в которой каждая пара (x_i, x_{i+1}) является ребром, причем все эти ребра различны. *Длиной пути* называется число ребер в нем, т.е. $k - 1$. *Цикл* – это путь, у которого $x_1 = x_k$.

Расстоянием между вершинами в графе называется наименьшая длина соединяющего их пути.

Для поиска всех путей между двумя вершинами воспользоваться поиском в ширину уже не получится, здесь можно использовать только поиск в глубину, который позволяет после достижения конечной вершины, вернуться на шаг назад и попробовать найти другой путь.

Рассмотрим граф на рис. 3.

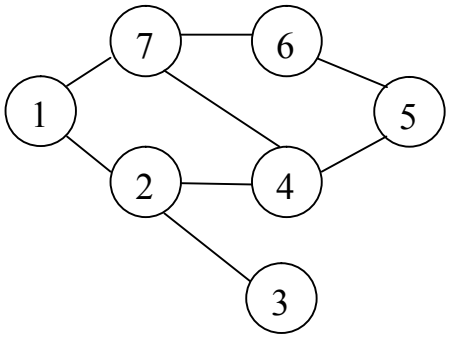

Рис. 3. Поиск путей

Допустим, мы хотим найти все возможные пути из вершины 3 в вершину 6. Если не рассматривать пути, содержащие циклы, то всего существует четыре пути:

- 3 2 1 7 6,
- 3 2 1 7 4 5 6,
- 3 2 4 5 6,
- 3 2 4 7 6.

Рассмотрим применение поиска в глубину для ответа на вопрос: сколько существует различных путей из вершины i в вершину j. Изменим список параметров процедуры `dfs`: x – номер вершины, до которой требуется найти путь, *cnt* – количество найденных путей. Кроме этого в начало процедуры нужно добавить «заглушку», чтобы рекурсивный вызов `dfs` прекращался, как только мы дошли до вершины x.

```
procedure dfs(var a:graph; var visited: mas;
                  n, v, x: integer; var cnt: integer);
  var i: integer;
begin
  if v = x then begin {проверяем достигли ли конечную
вершину}
    Inc(cnt); {увеличиваем количество найденных путей}
    exit; {выходим из процедуры, т.к. путь найден и дальше
просматривать вершину не требуется}
  end;
  visited[v] := true; {помечаем текущую вершину как
посещенную и принадлежащую с-й компоненте связности}
  for i := 1 to n do {перебираем все вершины смежные с v,
которые еще не были посещены}
    if a[v, i] and not visited[i] then
      dfs(a, visited, n, i, x, cnt); {повторяем алгоритм
для i-й вершины}
  visited[v] := false; {отменяем отметку о посещении
вершины, чтобы можно было в нее вернуться другим путем}
end;
```

Аналогичным образом можно находить не количество путей, а сами пути. Для этого потребуется ввести дополнительный массив (или другую структуру данных) для хранения всех найденных путей. Реализацию этой идеи оставим на самостоятельное выполнение.

2.4. Задача об одностороннем движении

Рассмотрим применение поиска в глубину в ориентированном графе на следующем примере: пусть дан связный ориентированный граф, представляющий собой схему улиц. Вершины в таком графе соответствуют перекрёсткам, а дуги графа – это улицы между перекрёстками. Таким образом, если с i-го перекрёстка мы можем напрямую попасть на j-й, то $a[i, j] = true$. Для оптимизации дорожного движения администрация города решила все возможные улицы сделать односторонними таким образом, чтобы сохранить полную достижимость, т. е. чтобы с любого перекрёстка по-прежнему можно было попасть на любой. В терминах теории графа задачу можно сформулировать так: каждому неориентированному ребру графа придать ориентированность так, чтобы граф остался связным. Заметим сразу, что задача не всегда имеет решение, а именно если в графе имеется такое ребро (*перешеек*), при удалении которого граф перестаёт быть связным,

то мы не можем сделать граф ориентированным и сохранить достижимость всех вершин. Проиллюстрируем это на примере (рис. 4).

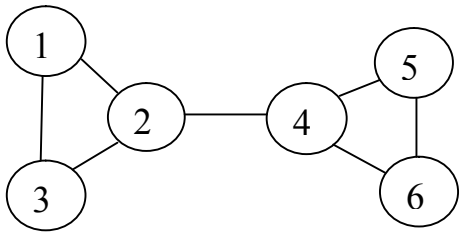

Рис. 4. Задача об одностороннем движении

Удаление ребра между вершинами 2 и 4 приведёт к тому, что граф перестанет быть связным, а значит, не будет выполняться требование задачи – достижимость всех вершин. В случае, когда такого ребра в графе нет, решение задачи существует и может быть получено при помощи поиска в глубину. Для этого при поиске в глубину мы будем придавать одну ориентацию всем ребрам, по которым мы прошли (назовём их *сильными* рёбрами), ребра, которые не были рассмотрены поиском в глубину (назовём их *слабыми*), получат противоположную ориентацию.

Очевидно, что из исходной вершины поиска в глубину можно попасть в любую вершину ориентированным путем, состоящим из сильных ребер. Но также и для любой вершины x, отличной от исходной, существует слабое ребро, ведущее от x или ее потомка к ее предку (в противном случае ребро, соединяющее x с ее непосредственным предком, было бы перешейком). Поэтому из x можно ориентированным путем (сначала по сильным ребрам, потом по слабому) попасть в вершину, являющуюся ее предком. Серия таких «спусков» неизбежно приведет в исходную вершину. Таким образом, из любой вершины достижима начальная вершина поиска. Следовательно, полученный ориентированный граф является связным.

Рассмотрим следующую реализацию данного решения:

```
procedure dfs(var a:graph; var visited: mas;
                  n, v: integer);
  var i: integer;
begin
  visited[v] := true; {помечаем текущую вершину как
посещенную}
  for i := 1 to n do {перебираем все вершины смежные с v,
которые еще не были посещены}
    if a[v, i] then begin
      if not visited[i] then begin
```

```
        a[i, v]:=false; {удаляем обратное ребро из i в v,
оставляем сильное ребро}
        dfs(a, visited, n, i); {повторяем алгоритм для i-
той вершины}
      end
      else if i > v then
        a[v, i]:=false; {удаляем прямое ребро из v в i,
оставляем слабое ребро}
    end;
end;
```

Изменения коснулись только той части, где мы рекурсивно вызываем процедуру `dfs`. В случае если поиск в глубину идёт по какому-то ребру, то в *a*[*i, v*] заносим *false*, при этом в *a*[*v, i*] остаётся *true*. Это и есть придание ориентации ребру, из вершины *v* в *i* мы можем попасть, а из *i* в *v* нет. Для остальных ребёр мы делаем обратную операцию: оставляем ребро *a*[*i, v*], но удаляем *a*[*v, i*], присваивая ему *false*.

3. Поиск в ширину

3.1. Основная идея

Поиск в ширину (англ. *Breadth-first search, BFS*) – ещё один способ обхода графа. Отличие от поиска в глубину заключается в порядке посещения вершин. Для текущей вершины мы находим все смежные и непросмотренные вершины и добавляем их в очередь. Затем извлекаем вершину из очереди и повторяем алгоритм для нее. Можно встретить и другое название алгоритма – *волновой* или *метод волны*, эти названия чаще встречаются, когда говорят о поисках путей на клеточных полях.

Рассмотрим один из возможных способов реализации поиска в ширину:

```
procedure bfs(var a:matrix; startPoint, n:integer);
  var queue,p:mas;
      visited:array[1..NMax] of boolean;
      q_first,q_last,current, i, j:integer;
begin
  for i:=1 to n do visited[i]:=false; {помечаем все вершины
как непосещенные}
  q_first:=0; {указатель для чтения из очереди}
  q_last:=1; {указатель для записи в очередь}
  visited[startPoint]:=true; {посещаем стартовую вершину}
  current:=startPoint; {устанавливаем текущую вершину
равную стартовой}
  {продолжаем поиск пока не все достижимые вершины были
просмотрены}
  while (q_first < q_last) do begin
    for i:=1 to n do begin {просматриваем все не посещенные
вершины, достижимые из вершины current}
      if a[current,i] and not visited[i] then begin
        visited[i]:=true; {помечаем i-ую вершину как
посещенную}
        queue[q_last]:=i; {помещаем вершину в очередь}
        Inc(q_last); {увеличиваем указатель на запись в
очереди}
      end;
    end;
    Inc(q_first); {увеличиваем указатель на чтение в
очереди}
    current:=queue[q_first]; {извлекаем следующий элемент
из очереди}
  end;
end;
```

Для реализации очереди мы использовали обычный массив, что может быть нерациональным с точки зрения расхода памяти, но упрощает написание и чтение программы. Принципиальным отличием от поиска в глубину является то, что для текущей вершины *current* мы сразу берём все смежные с ней вершины, которые ещё не были просмотрены, и помещаем их в очередь для просмотра. В случае поиска в глубину мы перешли бы сразу в первую найденную вершину и продолжили поиск из неё.

3.2. Поиск кратчайшего пути

Запуская поиск в ширину, мы посетим все вершины кратчайшим способом. Таким образом, мы можем найти не только расстояние между двумя заданными вершинами, а кратчайшие расстояния от начальной вершины до всех остальных. Рассмотрим модификацию процедуры `bfs`, которая не только проходит по всем вершинам графа, но и возвращает найденный путь между вершинами `startPoint` и `endPoint`, а также длину этого пути `path_length`.

```
procedure bfs(var a:matrix; startPoint,endpoint,n:integer;
var path:mas; var path_length:integer);
  var queue,p:mas;
      visited:array[1..NMax] of boolean;
      q_first,q_last,current, i, j:integer;
begin
  for i:=1 to n do visited[i]:=false; {помечаем все вершины
как непосещенные}
  q_first:=0; {указатель для чтения из очереди}
  q_last:=1; {указатель для записи в очередь}
  visited[startPoint]:=true; {посещаем стартовую вершину}
  current:=startPoint; {устанавливаем текущую вершину
равную стартовой}
  path_length:=1; {устанавливаем текущую длину пути равную
единице}
  {продолжаем поиск пока не посещена конечная вершина и не
все достижимые вершины были просмотрены}
  while (not visited[endPoint]) and (q_first < q_last) do
begin
    Inc(path_length); {увеличиваем длину пути на единицу}
    for i:=1 to n do begin {просматриваем все не посещенные
вершины, достижимые из вершины current}
      if a[current,i] and visited[i] then begin
        visited[i]:=true; {помечаем i-ую вершину как
посещенную}
        queue[q_last]:=i; {помещаем вершину в очередь}
```

```
        p[i]:=current; {запоминаем из какой вершины
посетили i-ую}
        Inc(q_last); {увеличиваем указатель на запись в
очереди}
      end;
    end;
    Inc(q_first); {увеличиваем указатель на чтение в
очереди}
    current:=queue[q_first]; {извлекаем следующий элемент
из очереди}
  end;

  if (visited[endPoint]) then begin {если конечная вершина
посещена, то восстанавливаем путь}
    path_length:=1; {устанавливаем текущую длину пути
равную единице}
    path[1]:=endPoint; {добавляем последнюю вершину в путь}
    while p[path[path_length]] <> -1 do begin
      Inc(path_length); {увеличиваем длину пути на единицу}
      path[path_length]:=p[path[path_length-1]]; {находим
вершину, из которой пришли в текущую и помещаем в путь}
    end;
  end
  else path_length:=-1; {путь не найден}
end;
```

Мы добавили вспомогательный массив `p`, в котором запоминаем номер вершины, из которой мы пришли в *i*-ую вершину. Этот массив нам поможет восстановить путь после достижения конечной вершины. Проверяем в массиве `visited`, была ли посещена вершина `endPoint`, если была, то путь найден, и надо его восстановить. Иначе пути из вершины `startPoint` в `endPoint` не существует, в этом случае мы просто вернём –1 в переменной `path_length`. Благодаря тому, что в массиве `p` мы сохранили вершины из которых приходили, то сейчас мы легко можем узнать номер вершины из которой пришли в `endPoint`, то есть предпоследнюю вершину – это будет `p[endPoint]`. Аналогично мы можем узнать вершину, из которой мы попали в предпоследнюю – `p[p[endPoint]]`. Передвигаясь таким образом по массиву `p`, мы дойдем до начальной вершины, при этом все вершины будем помещать в массив `path`. Отметим, что после этого массив `path` будет содержать путь, но в обратном порядке, то есть от вершины `endPoint` к `startPoint`. Кроме того, можно заметить, что массив `p` и `visited` изменяются по одному условию, и их можно заменить одним массивом. Устранение этих двух недочётов оставим на самостоятельное выполнение.

4. Игры на графах

Слова «граф» и «игра» нередко оказываются рядом. Чаще всего это происходит из-за того, что придумано много игр на графах. В этих играх игроки передвигаются по графу или что-нибудь делают с ним: удаляют вершины или ребра, красят их в разные цвета и т. д. Некоторые из этих игр возникли как подспорье при изучении проблем самой теории графов, другие моделируют жизненные коллизии или обобщают известные игры. В качестве примера рассмотрим игру *Ним*, проанализированную Бержем [2] и обобщающую известную игру с тем же названием.

В игре Ним два игрока поочередно передвигают одну фишку из вершины в вершину по ребрам ориентированного графа (каждый ход – одно ребро). Проигрывает тот, кто не может сделать очередной ход, так как оказался в тупике – вершине, из которой не выходит ни одно ребро. Начальная позиция выбирается случайным образом.

Если в графе нет тупиков, игра не имеет смысла, так как ни одна партия не может закончиться. Бесконечные партии возможны и в том случае, когда в графе имеется ориентированный цикл. Рассмотрим графы, в которых таких циклов нет – их называют *ациклическими*. Поскольку граф ациклический, то рассмотрим его на примере следующего дерева.

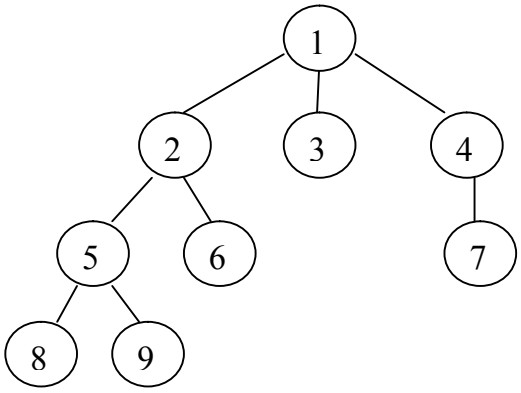

Рис. 5. Ациклический граф

В ациклическом графе обязательно есть хотя бы один тупик. Пусть игра начинается в вершине 1, тогда обозначим A_1 – множество всех тупиков данного графа. В нашем случае в множество A_1 войдут следующие вершины: 3, 6, 7, 8, 9. Игрок, который может сделать ход в какую-нибудь из этих вершин, выигрывает, так как противник оказывается в тупике. Пусть B_1 – множество всех вершин, из которых хотя бы одно ребро ведет в вершину из A_1. Для нашего графа – это 1, 2, 4, 5. Таким образом, поднимаясь снизу вверх, мы можем дать однозначный ответ: выиграет первый или второй игрок.

Это пример ретроспективного анализа и он, разумеется, не всегда так прост. Кроме того, простота и этого примера обманчива, т. к. граф может быть очень большим и необходимые для расчета игры вычисления просто невозможно будет выполнить в разумное время.

Кроме того, встаёт вопрос о том, как найти тупики в графе, множество вершин A_1. Можно использовать различные способы для этого, один из них заключается в том, чтобы представить граф в виде линейного массива, где индекс элемента массива – это номер вершины графа, а его значение – номер вершины-родителя. Приведём пример, как будет выглядеть такой массив для рассмотренного выше графа.

0	1	1	1	2	2	4	5	5

Таким образом, вершины, на которые никто не ссылается в данном представлении, и есть искомое множество A_1. Построить множество B_1 также не представляется сложным. Данная структура позволяет нам легко переходить от нижнего уровня к верхнему, что соответствует чередованию ходов игроков, и на каждом шаге выбирать оптимальное решение.

Многие известные игры могут быть представлены как игры на графах. Например, шахматы. Можно определить граф, вершины которого представляют возможные позиции на доске, а ребра – возможные ходы. Получается игра, похожая на Ним, но это как раз тот случай, когда граф слишком велик. Рассмотрим более простой пример.

В игре «полицейский и вор» два участника поочередно перемещаются из вершины в вершину по ребрам неориентированного графа (одно ребро за ход). Каждый может пропустить ход (остаться на месте). Цель полицейского – оказаться в одной вершине с вором, тогда он выиграл. Цель вора – оказаться в особо отмеченной вершине (выходе), когда там не стоит полицейский, тогда выиграл вор.

Это, конечно, игра на графе, но ее можно превратить в игру с одной передвигаемой фишкой, более похожую на Ним. Каждая ситуация в игре «полицейский и вор» описывается указанием позиций полицейского и вора и того, чей сейчас ход. Можно построить граф, вершинами которого являются тройки (x, y, m), где x и y – номера вершин, где находятся соответственно полицейский и вор, а m указывает очередность хода. Ребра графа соответствуют возможным ходам. Если в исходном графе было n вершин, то в новом их стало $2n^2$. Особо отмечаются вершины, где побеждает кто-нибудь из участников. Теперь позиция в игре – это одна вершина, отмеченная фишкой, которую поочередно передвигают два игрока, каждый из которых стремится попасть в одну из своих выигрышных вершин.

5. Задания для самостоятельного решения

1. Реализовать нерекурсивный вариант поиска в глубину.

2. Модифицировать алгоритм поиска в глубину для нахождения всех путей между двумя вершинами графа.

3. Организовать вывод компонент связности с перечислением, какие вершины принадлежат *i*-й компоненте.

4. Использовать поиск в ширину для нахождения компонент связности.

5. Реализовать поиск в ширину для графа, представленного списками связи.

6. Найти в графе две вершины, максимально удалённые друг от друга. Расстоянием между вершинами считается количество ребер.

7. Найти «центральную» вершину в графе, то есть такую, что расстояние от нее до самой удаленной будет минимальным.

8. Модифицировать поиск в ширину для нахождения расстояния лишь до заданной вершины.

9. Доработать решение задачи об одностороннем движении так, чтобы в случае отсутствия решения программа выводила соответствующий ответ.

10. Изменить нахождение кратчайших путей так, чтобы путь возвращался в прямом порядке, а не обратном, и избавится от использования массива *p*.

6. Заключение

В данном разделе были рассмотрены лишь базовые алгоритмы для работы с графами. Существует большое количество других алгоритмов, решающих самые различные задачи, например:

- генерация деревьев (каркасов) из графа: порождение всех каркасов графа, каркасы минимального веса;
- поиск циклов: эйлеровы циклы, гамильтоновы, фундаментальное множество циклов;
- поиск кратчайших путей: алгоритм Дейкстры, алгоритм Флойда;
- потоки в сетях – определение максимального потока, протекающего между заданной парой вершин;
- генерация паросочетаний – двух множеств вершин таких, что никакие две вершины одного множества не связаны друг с другом, но связаны с вершинами второго множества.

Литература

1. Алексеев В. Е., Таланов В. А. Графы и алгоритмы. Структуры данных. Модели вычислений. – М: Интернет-Университет Информационных Технологий. БИНОМ. Лаборатория знаний, 2006.

2. Берж К. Теория графов и ее применения. М.: ИЛ, 1962.

3. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. – 2-е изд.: Пер. с англ. – М.: Издательский дом «Вильямс», 2005.

4. Окулов С. М. Основы программирования. – 5-е изд., испр. – М.: БИНОМ. Лаборатория знаний, 2010.