Лабораторная работа №4

Тема: *Программирование алгоритмов на графах.*

Цель: *ознакомиться с понятием «графы», изучить основные алгоритмы графов, научиться применять полученные знания на практике.*

1 Краткая теория

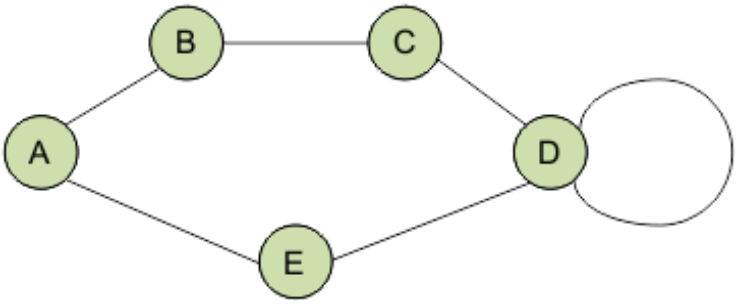
1.1 ГРАФ.

Граф – совокупность точек, соединенных линиями. Точки называются ***вершинами***, или ***узлами***, а линии – ***ребрами***, или ***дугами***.

Степень входа вершины – количество входящих в нее ребер, ***степень выхода*** – количество исходящих ребер.

Граф, содержащий ребра между всеми парами вершин, является ***полным***. Встречаются такие графы, ребрам которых поставлено в соответствие конкретное числовое значение, они называются ***взвешенными графами***, а это значение – ***весом ребра***.

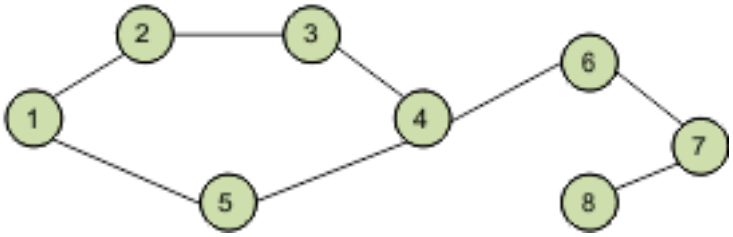
Когда у ребра оба конца совпадают, т.е. оно выходит из вершины и входит в нее, то такое ребро называется ***петлей***.



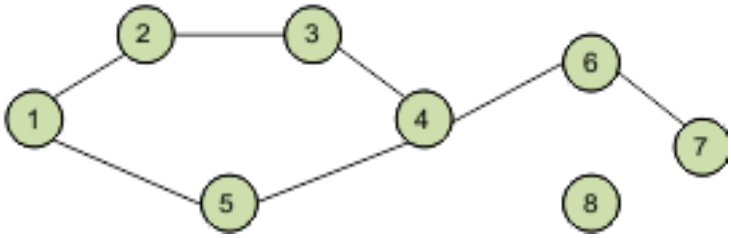
Классификация графов

Графы делятся на

- Связные



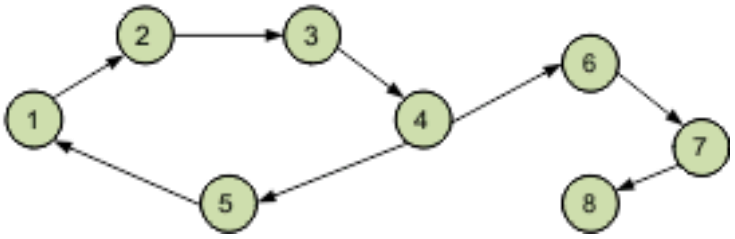
- Несвязные



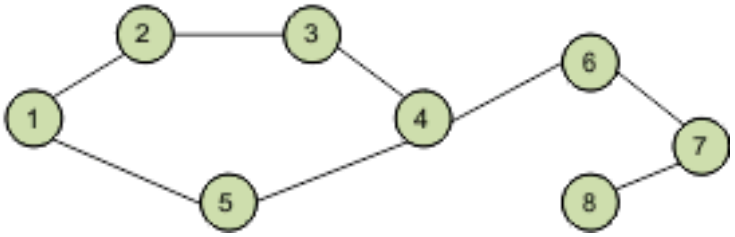
В связном графе между любой парой вершин существует как минимум один путь. В несвязном графе существует хотя бы одна вершина, не связанная с другими.

Графы также подразделяются на

- Ориентированные



- Неориентированные



- смешанные.

В ориентированном графе ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами. В неориентированном графе по каждому из ребер можно осуществлять переход в обоих направлениях.

Частный случай двух этих видов – смешанный граф. Он характерен наличием как ориентированных, так и неориентированных ребер.

Способы представления графа

Граф может быть представлен (сохранен) несколькими способами:

- матрица смежности;
- матрица инцидентности;
- список смежности (инцидентности);
- список ребер.

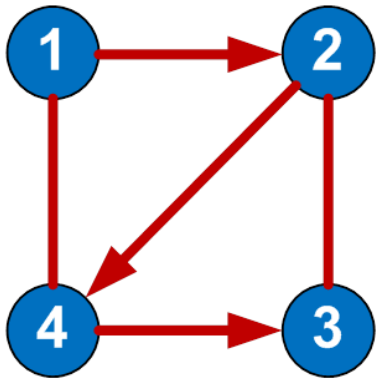
Использование двух первых методов предполагает хранение графа в виде двумерного массива (матрицы). Размер массива зависит от количества вершин и/или ребер в конкретном графе.

Матрица смежности графа — это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1.

Число строк матрицы смежности равно числу столбцов и соответствует количеству вершин графа.

- 0 – соответствует отсутствию ребра,
- 1 – соответствует наличию ребра.

	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0



Когда из одной вершины в другую проход свободен (имеется ребро), в ячейку заносится 1, иначе – 0. Все элементы на главной диагонали равны 0 если граф не имеет петель.

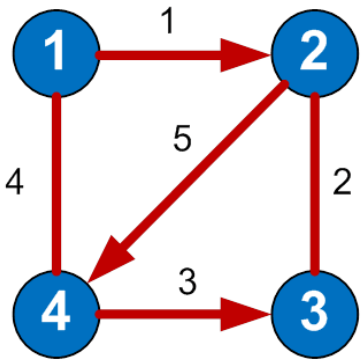
Матрица инцидентности (инциденции) графа — это матрица, количество строк в которой соответствует числу вершин, а количество столбцов – числу рёбер. В ней указываются связи между инцидентными элементами графа (ребро(дуга) и вершина).

В неориентированном графе если вершина инцидентна ребру то соответствующий элемент равен 1, в противном случае элемент равен 0.

В ориентированном графе если ребро выходит из вершины, то соответствующий элемент равен 1, если ребро входит в вершину, то соответствующий элемент равен -1, если ребро отсутствует, то элемент равен 0.

Матрица инцидентности для своего представления требует нумерации рёбер, что не всегда удобно.

	1	2	3	4	5
1	1	0	0	1	0
2	-1	1	0	0	1
3	0	1	-1	0	0
4	0	0	1	1	-1

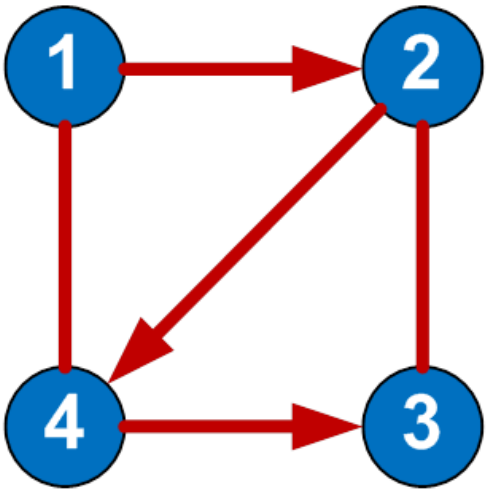


Список смежности (инцидентности)

Если количество ребер графа по сравнению с количеством вершин невелико, то значения большинства элементов матрицы смежности будут равны 0. При этом использование данного метода нецелесообразно. Для подобных графов имеются более оптимальные способы их представления.

По отношению к памяти списки смежности менее требовательны, чем матрицы смежности. Такой список можно представить в виде таблицы, столбцов в которой – 2, а строк — не больше, чем вершин в графе. В каждой строке в первом столбце указана вершина выхода, а во втором столбце – список вершин, в которые входят ребра из текущей вершины.

1	2, 4
2	3, 4
3	2
4	1, 3



Преимущества списка смежности:

- Рациональное использование памяти.
- Позволяет быстро перебирать соседей вершины.
- Позволяет проверять наличие ребра и удалять его.

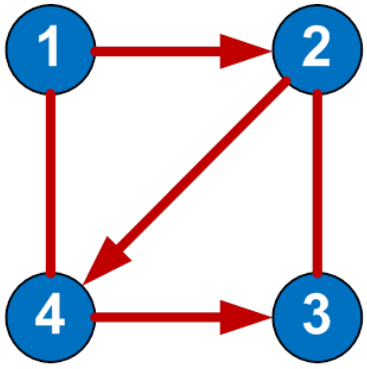
Недостатки списка смежности:

- При работе с насыщенными графами (с большим количеством рёбер) скорости может не хватать.
- Нет быстрого способа проверить, существует ли ребро между двумя вершинами.
- Количество вершин графа должно быть известно заранее.
- Для взвешенных графов приходится хранить список, элементы которого должны содержать два значащих поля, что усложняет код:
  - номер вершины, с которой соединяется текущая;
  - вес ребра.

Список рёбер

В списке рёбер в каждой строке записываются две смежные вершины и вес соединяющего их ребра (для взвешенного графа). Количество строк в списке ребер всегда должно быть равно величине, получающейся в результате сложения ориентированных рёбер с удвоенным количеством неориентированных рёбер.

	Начало	Конец	Вес
1	1	2	
2	1	4	
3	2	3	
4	2	4	
5	3	2	
6	4	1	
7	4	3	



Какой способ представления графа лучше? Ответ зависит от отношения между числом вершин и числом рёбер. Число ребер может быть довольно малым (такого же порядка, как и количество вершин) или довольно большим (если граф является полным). Графы с большим числом рёбер называют ***плотными***, с малым — ***разреженными***. Плотные графы удобнее хранить в виде матрицы смежности, разреженные — в виде списка смежности.

Алгоритмы обхода графов

Основными алгоритмами обхода графов являются

- Поиск в ширину
- Поиск в глубину

Поиск в ширину подразумевает поуровневое исследование графа:

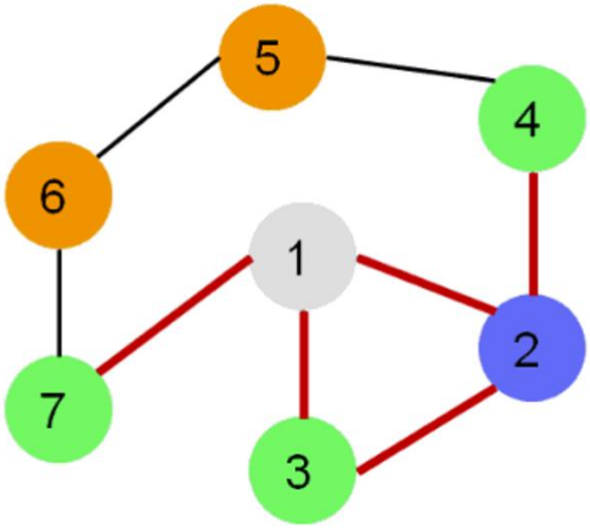
- вначале посещается корень – произвольно выбранный узел,
- затем – все потомки данного узла,
- после этого посещаются потомки потомков и т.д.

Вершины просматриваются в порядке возрастания их расстояния от корня. Алгоритм прекращает свою работу после обхода всех вершин графа, либо в случае выполнения требуемого условия (например, найти кратчайший путь из вершины 1 в вершину 6).

Каждая вершина может находиться в одном из 3 состояний:

- 0 — оранжевый – необнаруженная вершина;
- 1 — зеленый – обнаруженная, но не посещенная вершина;
- 2 — серый – обработанная вершина.

Фиолетовый – рассматриваемая вершина.



Применения алгоритма поиска в ширину

- Поиск кратчайшего пути в невзвешенном графе (ориентированном или неориентированном).
- Поиск компонент связности.
- Нахождения решения какой-либо задачи (игры) с наименьшим числом ходов.
- Найти все рёбра, лежащие на каком-либо кратчайшем пути между заданной парой вершин.
- Найти все вершины, лежащие на каком-либо кратчайшем пути между заданной парой вершин.

Алгоритм поиска в ширину работает как на ориентированных, так и на неориентированных графах. Для реализации алгоритма удобно использовать очередь.

Реализация на C++ (с использованием очереди STL)

```
#include <iostream>
#include <queue> // очередь
using namespace std;
int main()
{
  queue<int> Queue;
  int mas[7][7] = { { 0, 1, 1, 0, 0, 0, 1 }, // матрица смежности
  { 1, 0, 1, 1, 0, 0, 0 },
  { 1, 1, 0, 0, 0, 0, 0 },
  { 0, 1, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 1, 0, 1, 0 },
  { 0, 0, 0, 0, 1, 0, 1 },
  { 1, 0, 0, 0, 0, 1, 0 } };
  int nodes[7]; // вершины графа
  for (int i = 0; i < 7; i++)
    nodes[i] = 0; // исходно все вершины равны 0
  Queue.push(0); // помещаем в очередь первую вершину
  while (!Queue.empty())
  { // пока очередь не пуста
```

```
    int node = Queue.front(); // извлекаем вершину
    Queue.pop();
    nodes[node] = 2; // отмечаем ее как посещенную
    for (int j = 0; j < 7; j++)
    { // проверяем для нее все смежные вершины
      if (mas[node][j] == 1 && nodes[j] == 0)
      { // если вершина смежная и не обнаружена
        Queue.push(j); // добавляем ее в очередь
        nodes[j] = 1; // отмечаем вершину как обнаруженную
      }
    }
    cout << node + 1 << endl; // выводим номер вершины
  }
  cin.get();
  return 0;
}
```

Результат выполнения

Поиск в глубину – это алгоритм обхода вершин графа.

Поиск в ширину производится симметрично (вершины графа просматривались по уровням). Поиск в глубину предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность продвижения означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина).

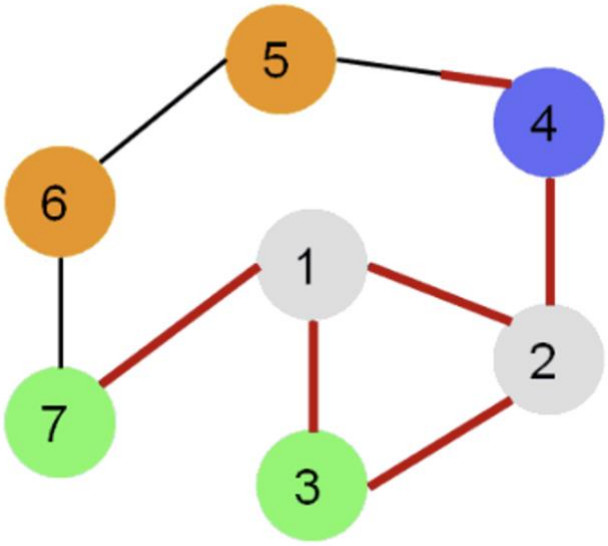
Отсутствие последнего свидетельствует об одной из двух возможных ситуаций:

- все вершины графа уже просмотрены,
- просмотрены вершины доступные из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают последний вариант).

Каждая вершина может находиться в одном из 3 состояний:

- 0 - оранжевый – необнаруженная вершина;
- 1 - зеленый – обнаруженная, но не посещенная вершина;
- 2 - серый – обработанная вершина;

Фиолетовый – рассматриваемая вершина.



Применения алгоритма поиска в глубину

- Поиск любого пути в графе.
- Поиск лексикографически первого пути в графе.
- Проверка, является ли одна вершина дерева предком другой.
- Поиск наименьшего общего предка.
- Топологическая сортировка.
- Поиск компонент связности.

Алгоритм поиска в глубину работает как на ориентированных, так и на неориентированных графах. Применимость алгоритма зависит от конкретной задачи.

Для реализации алгоритма удобно использовать стек или рекурсию.

Реализация на C++ (с использованием стека STL)

```
#include <iostream>
#include <stack> // стек
using namespace std;
int main()
{
  stack<int> Stack;
  int mas[7][7] = { { 0, 1, 1, 0, 0, 0, 1 }, // матрица смежности
  { 1, 0, 1, 1, 0, 0, 0 },
  { 1, 1, 0, 0, 0, 0, 0 },
  { 0, 1, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 1, 0, 1, 0 },
  { 0, 0, 0, 0, 1, 0, 1 },
  { 1, 0, 0, 0, 0, 1, 0 } };
  int nodes[7]; // вершины графа
  for (int i = 0; i < 7; i++) // исходно все вершины равны 0
    nodes[i] = 0;
  Stack.push(0); // помещаем в очередь первую вершину
  while (!Stack.empty())
  { // пока стек не пуст
    int node = Stack.top(); // извлекаем вершину
    Stack.pop();
    if (nodes[node] == 2) continue;
```

```
    nodes[node] = 2; // отмечаем ее как посещенную
    for (int j = 6; j >= 0; j--)
    { // проверяем для нее все смежные вершины
      if (mas[node][j] == 1 && nodes[j] != 2)
      { // если вершина смежная и не обнаружена
        Stack.push(j); // добавляем ее в стек
        nodes[j] = 1; // отмечаем вершину как обнаруженную
      }
    }
    cout << node + 1 << endl; // выводим номер вершины
  }
  cin.get();
  return 0;
}
```

! Дополнительный материал !

1. Графы: основы теории, алгоритмы поиска: *https://medium.com/nuances-of-programming/графы-основы-теории-алгоритмы-поиска-b93672f59747*
2. Как использовать алгоритмы на графах: *https://bestprogrammer.ru/programmirovanie-i-razrabotka/algoritmy-101-kak-ispolzovat-algoritmy-na-grafah*

2 Задания

Обязательное 1*: Написать программу поиска (в глубину) лексикографически первого пути на графе*

! Контрольные вопросы !

1. Определение понятия граф.
2. Классификация графов.
3. Какой граф называют связным?
4. Отличие ориентированного графа от неориентированного.
5. Способы представления графа.
6. Алгоритмы обхода графов.
7. Принцип алгоритма поиска в ширину.
8. Принцип алгоритма поиска в глубину.

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Ответы на контрольные вопросы.
6. Выводы.