Лекция 7

Графы, классификация графов, способы представления графов, алгоритмы поиска в графах.

Основные понятия - граф

На практике часто бывает полезно изобразить некоторую ситуацию в виде рисунков, составленных из точек (вершин), представляющих основные ситуации, и линий (ребер), соединяющих определенные пары этих вершин и представляющих связи между ними.

Таким способом удобно представлять структуру системы, в которой вершины – это блоки, а ребра – связи между блоками. Такие рисунки известны под общим названием графов.

Графы встречаются в разных областях: структуры в гражданском строительстве, сети в электротехнике, и системы телекоммуникаций, биология, психология и другие, химия и т.д.

Начало теории графов было положено Эйлером в 1736 г., когда им была написана статья о Кенигсбергских мостах. Однако она была единственной в течение почти ста лет.

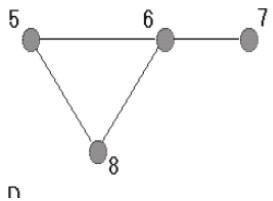
Основные понятия - граф

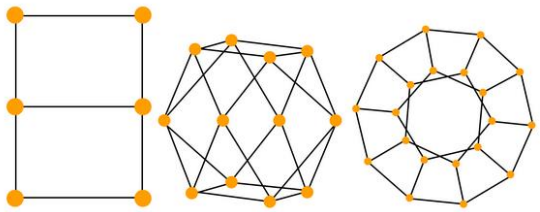
Граф – это абстрактное представление множества объектов и связей между ними.

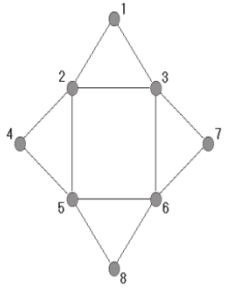
Граф – математический объект, состоящий из двух множеств. Одно из них – любое конечное множество, его элементы называются вершинами графа. Другое множество состоит из пар вершин, эти пары называются ребрами графа.

Графом G называется пара множеств (V,E), где

- V – непустое, конечное множество элементов, называемых вершинами. Графически это множество изображается точками;
- E – конечное множество пар различных элементов из V, называемых ребрами. Графически это множество изображается линией, соединяющей пару точек.

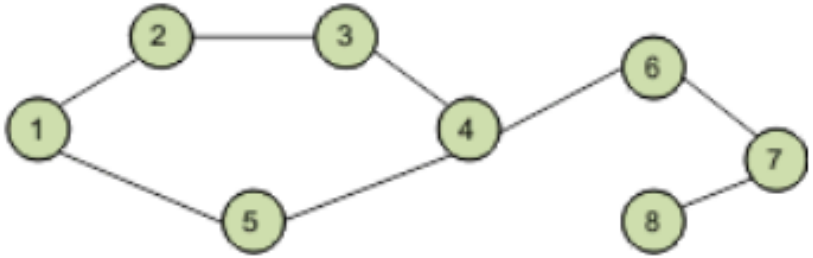




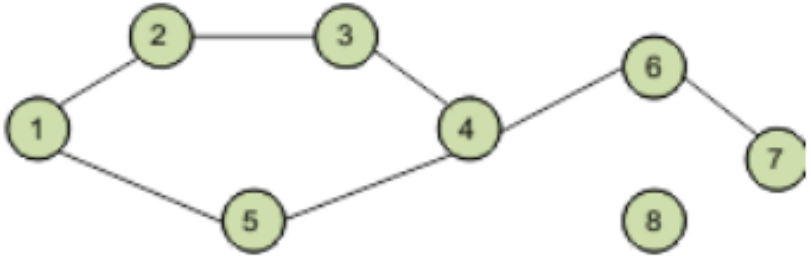


Классификация графов

В ***связном графе*** между любой парой вершин существует как минимум один путь.

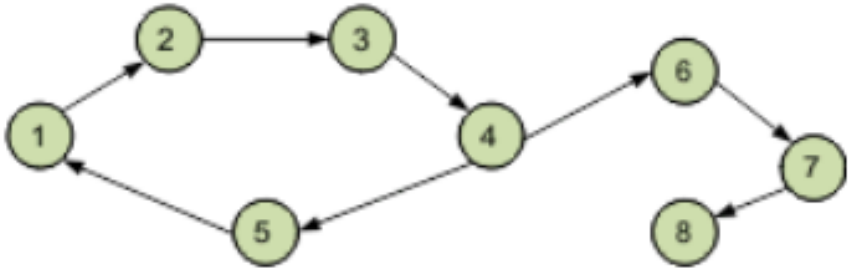


В ***несвязном графе*** существует хотя бы одна вершина, не связанная с другими.

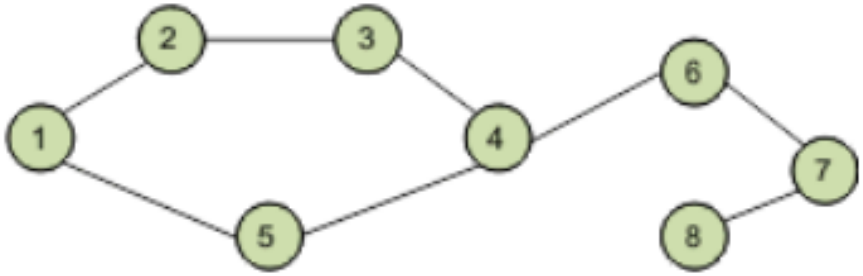


Классификация графов

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.



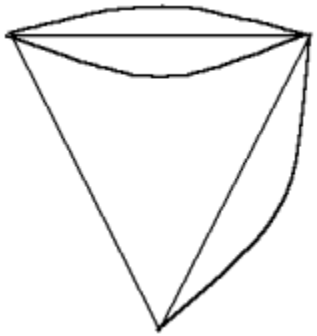
Частный случай двух этих видов – ***смешанный граф***. Он характерен наличием как ориентированных, так и неориентированных ребер.

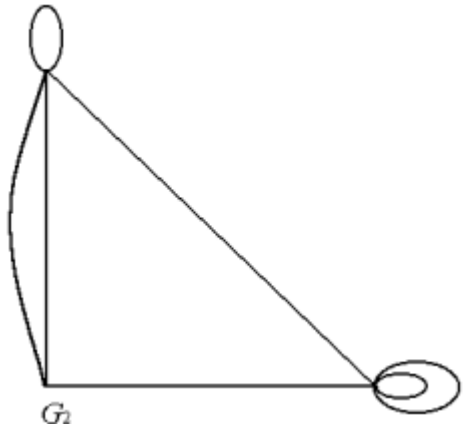
Классификация графов

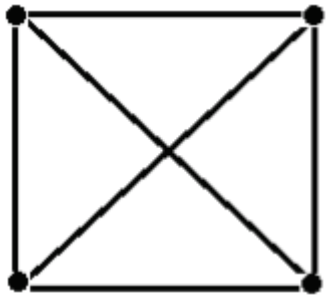
Мультиграфом называется граф, в котором пары вершин могут соединяться более чем одним ребром; эти ребра называются кратными.

Псевдограф – это граф, в котором допускаются петли, то есть ребра, соединяющие вершину саму с собой

Граф называется **полным** (кликой), если любые две его вершины смежны.

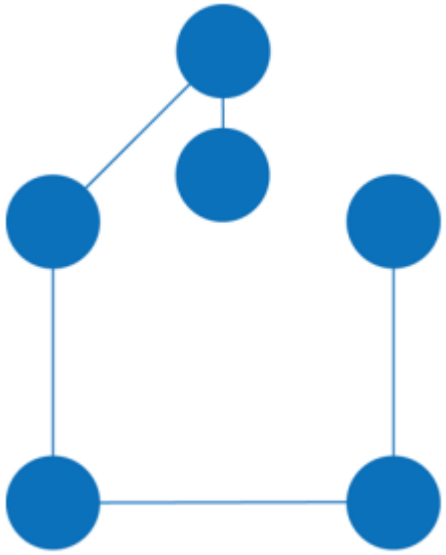




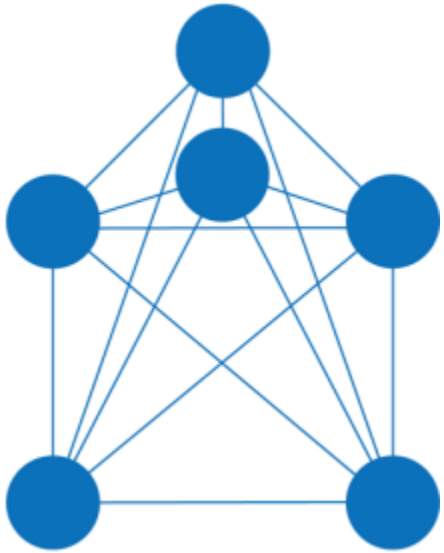


Классификация графов

Графы с большим числом рёбер называют ***плотными***, с малым – ***разреженными***.



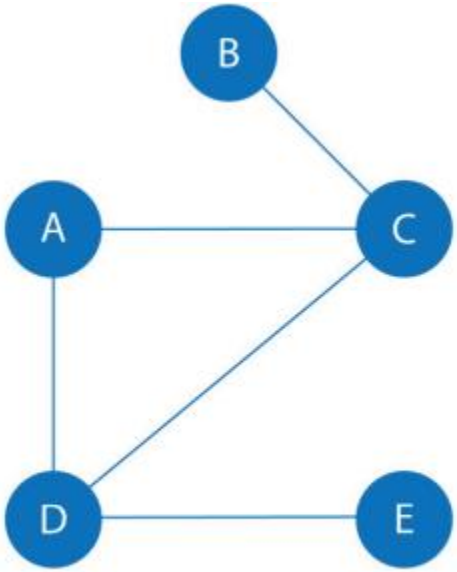
Разреженный



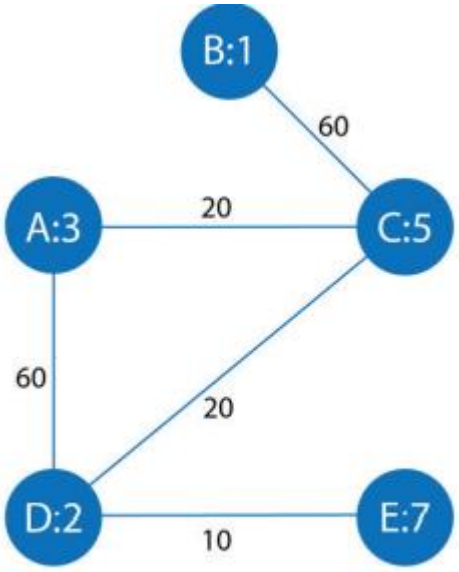
Плотный

Классификация графов

Невзвешенные графы не имеют весовых значений, назначенных их узлам или отношениям. Для ***взвешенных*** графов эти значения могут представлять различные показатели, такие как стоимость, время, расстояние, пропускная способность или даже приоритетность конкретной области.



Невзвешенный



Взвешенный

Основные термины

Путь в графе это конечная последовательность вершин, в которой каждые две вершины идущие подряд соединены ребром. ***Расстояние между двумя вершинами*** - это длина кратчайшего пути, соединяющего эти вершины.

Два ребра называются **смежными**, если у них есть общая вершина.

Ребро называется **петлей**, если его концы совпадают.

Степенью вершины называют количество ребер, для которых она является концевой (при этом петли считают дважды).

Вершина называется **изолированной**, если она не является концом ни для одного ребра.

Вершина называется **висячей**, если из неё выходит ровно одно ребро.

Граф без кратных ребер и петель называется **обыкновенным.**

Способы представления графа

Граф может быть представлен (сохранен) несколькими способами:

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

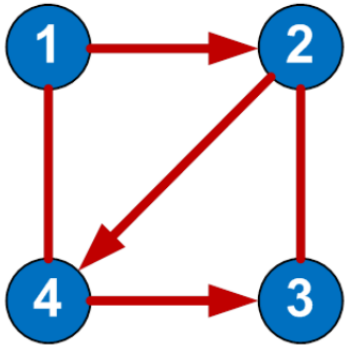
Использование двух первых методов предполагает хранение графа в виде двумерного массива (матрицы). Размер массива зависит от количества вершин и/или ребер в конкретном графе.

Способы представления графа – матрица смежности

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.



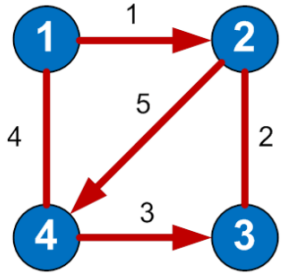
	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0

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.

Способы представления графа - матрица инцидентности

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).

	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



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.

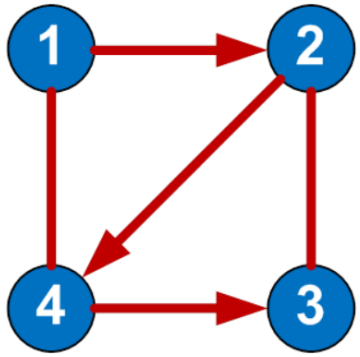
Способы представления графа – список смежности

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.

Список смежности 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



Способы представления графа – список смежности

Преимущества списка смежности:

- рациональное использование памяти;
- позволяет быстро перебирать соседей вершины;
- позволяет проверять наличие ребра и удалять его.

Недостатки списка смежности:

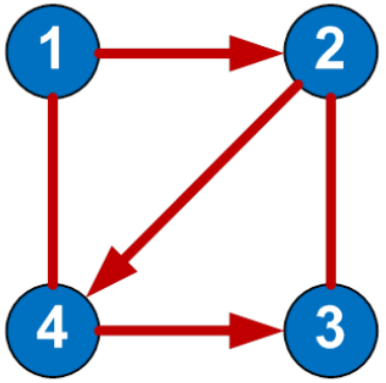
- при работе с насыщенными графами (с большим количеством рёбер) скорости может не хватать.
- нет быстрого способа проверить, существует ли ребро между двумя вершинами;
- количество вершин графа должно быть известно заранее;
- для взвешенных графов приходится хранить список, элементы которого должны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

Способы представления графа – 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	



Способы представления графа – список рёбер

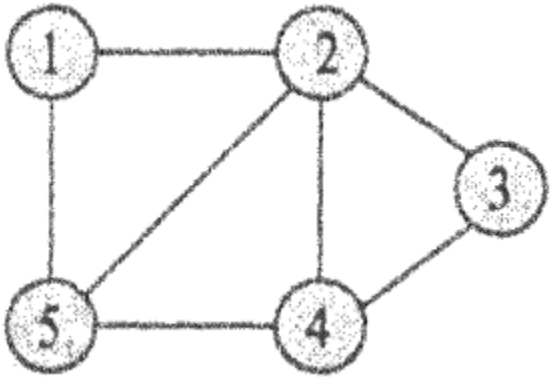
В коде:

```
int             NumTop,    // Число вершин
                NumArc;    // Число ребер

struct A                   // Ребро графа
{
    int         first;     // 1-я вершина ребра
    int         last;      // 2-я вершина ребра
    float       weight;    // Вес ребра
};

// Адрес первого элемента массива структур с информацией о
//   ребрах графа
A               *pArc;

// Структурный тип для графа
struct GRAPH
{
    int         NumTop;    // Число вершин
    int         NumArc;    // Число ребер
    A           *pArc;     // Указатель на начало массива ребер
                           //   в динамической памяти
};
```

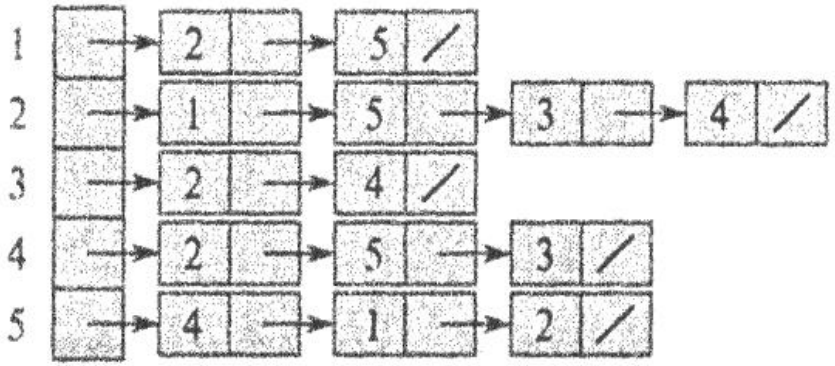
Способы представления графа – примеры

Пример неориентированного графа:

граф:



список смежности:



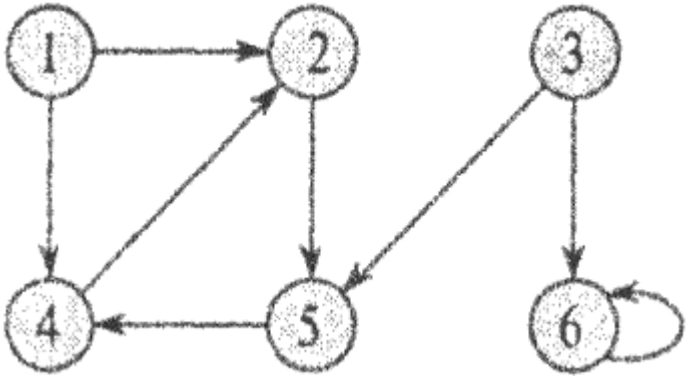
матрица смежности:

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

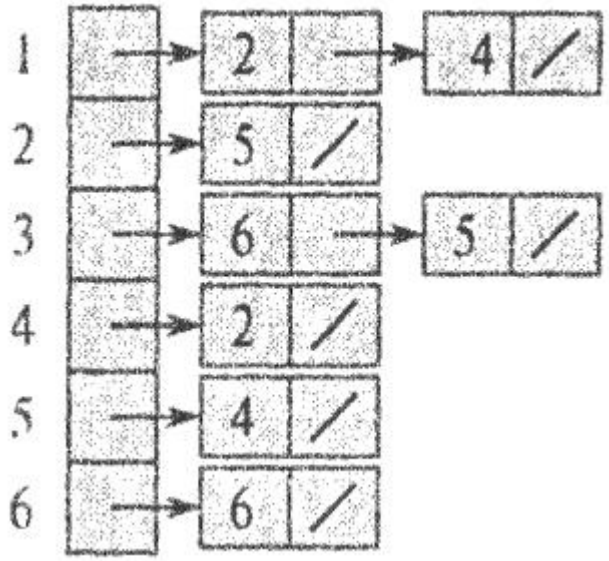
Способы представления графа – примеры

Пример ориентированного графа:

граф:



список смежности:



матрица смежности:

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Способы представления графа

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).

Плотные графы удобнее хранить в виде матрицы смежности, разреженные – в виде списка смежности (графы с большим числом рёбер называют ***плотными***, с малым – ***разреженными***).

Алгоритмы обхода графов

Основными алгоритмами обхода графов являются:

- поиск в ширину;
- поиск в глубину.

Алгоритмы обхода графов - поиск в ширину

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 0 в вершину 4).

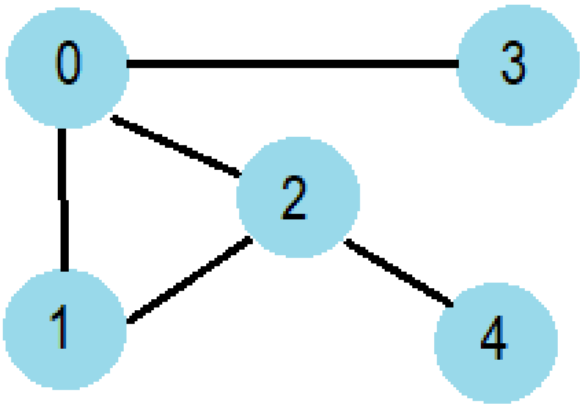
Каждая вершина может находиться в одном из 3 состояний:

- обнаруженная, но не посещенная вершина (открытая и необработанная);
- обработанная вершина (раскрытая);
- не рассмотренные (не обнаруженные вершины).

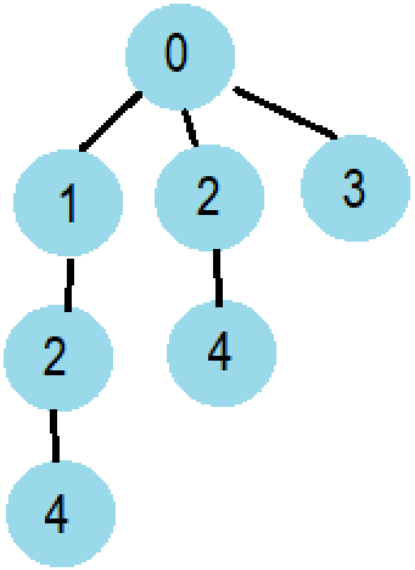
Алгоритмы обхода графов - поиск в ширину

Пример: исходная вершина 0, целевая 4.

Граф:



Дерево вывода:



Путь решения: 0, 1 2 3, 2, 4

Алгоритмы обхода графов - поиск в ширину

Алгоритм:

Пусть OPEN – список открытых вершин (), тогда CLOSED – список закрытых вершин.

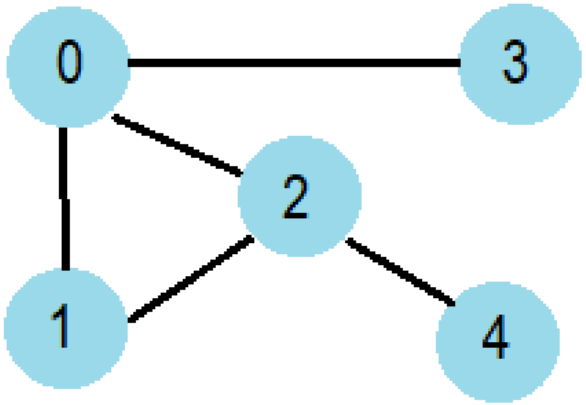
В начале поиска список CLOSED пустой, а OPEN содержит только начальную вершину. Каждый раз из списка OPEN выбирается для раскрытия первая вершина. Раскрытая вершина перемещается в список CLOSED, а ее дочерние вершины помещаются в ***конец*** списка OPEN, т.е. принцип формирования списка OPEN соответствует **очереди**. Согласно этой стратегии, вершины глубиной *k*, раскрываются после того как будут раскрыты все вершины глубиной *k-1*. В этом случае фронт поиска растет в **ширину**. Для построения обратного пути (из целевой вершины в начальную вершину) все дочерние вершины снабжаются указателями на соответствующие родительские вершины. В приведенном ниже алгоритме поиска в ширину функция first(OPEN) выбирает из списка OPEN первую вершину.

Алгоритмы обхода графов - поиск в ширину

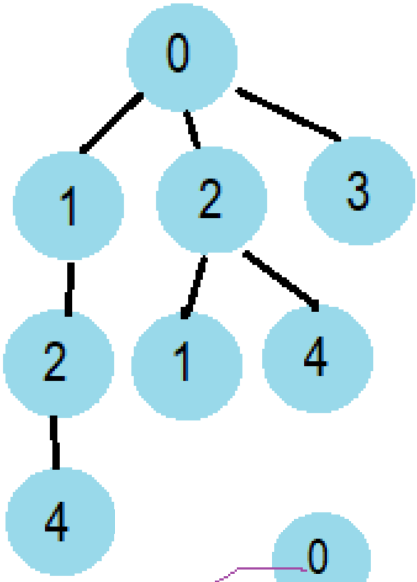
```
bool Breadth_First( граф )
{
 Поместить начальную вершину в список OPEN;
 CLOSED = 'пустой список';
 while (OPEN!= 'пустой список‘)
  {
   n=first(OPEN); // выбор из OPEN первой вершины
   if (n == 'целевая вершина') Выход(УСПЕХ);
    Переместить n из списка OPEN в CLOSED;
    Раскрыть вершину n и поместить все ее дочерние
    вершины, отсутствующие в списках OPEN и CLOSED,
    в конец списка OPEN, связав с каждой дочерней
    вершиной указатель на n;
  }
Выход(НЕУДАЧА);
}
```

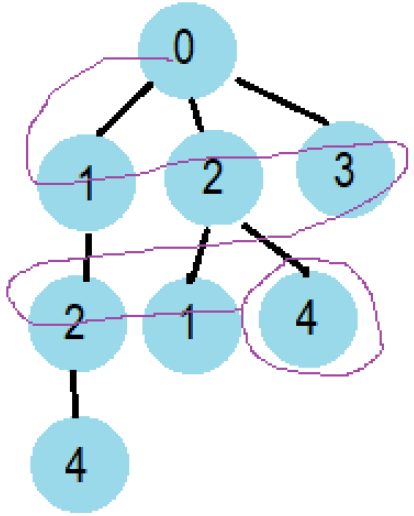
Алгоритмы обхода графов - поиск в ширину

Граф: старт 0, цель 4



Дерево вывода:



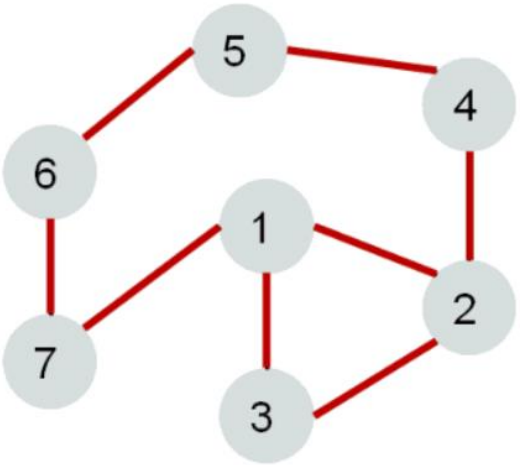


	OPEN	CLOSE
1	0	NULL
2	1 2 3	0
3	2 3 (2)	0 1
4	3 4	0 1 2
5	4	0 1 2 3

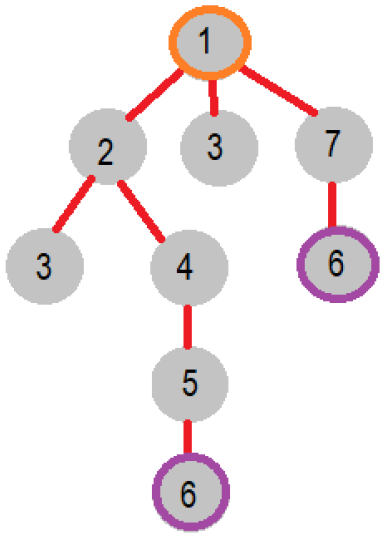
Решение: 0 1 2 3 (2) (1) 4 , в () уже рассмотренные вершины, т.е. решение итого: 0, 1 2 3, 4

Алгоритмы обхода графов - поиск в ширину

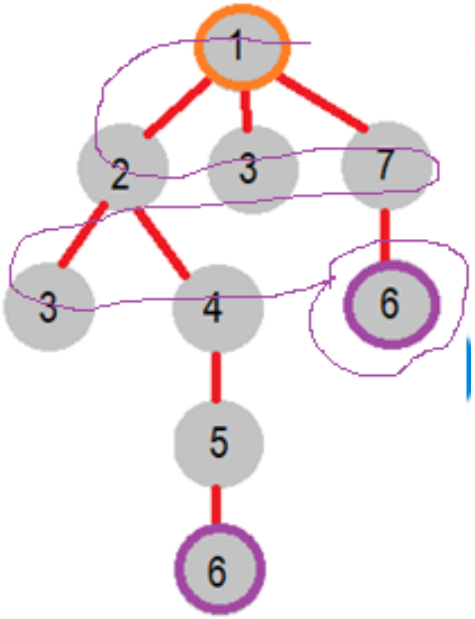
Граф: старт 1, цель 6



Дерево вывода:



	OPEN	CLOSE
1	1	NULL
2	2 3 7	1
3	3 7 (3) 4	1 2
4	7 4	1 2 3
5	4 6	1 2 3 7
6	6 5	1 2 3 7 4



Решение: 1 2 3 7 (3) 4 6

1, 2 3 7, 4 6

Алгоритмы обхода графов – поиск в ширину

Применение алгоритма поиска в ширину:

- поиск кратчайшего пути в невзвешенном графе (ориентированном или неориентированном);
- поиск компонент связности;
- нахождение решения какой-либо задачи (игры) с наименьшим числом ходов;
- поиск всех рёбер, лежащих на каком-либо кратчайшем пути между заданной парой вершин;
- поиск всех вершин, лежащих 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 используют очередь.

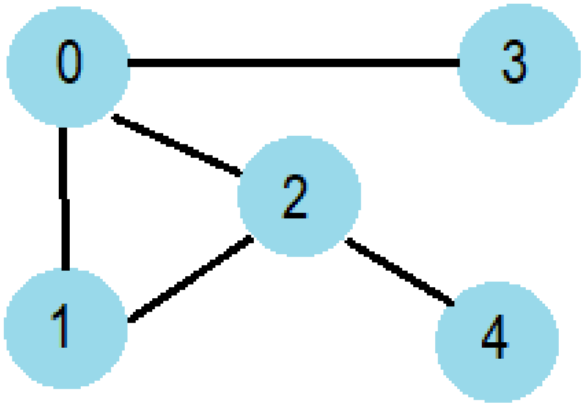
Алгоритмы обхода графов - поиск в глубину

При ***поиске в глубину*** всегда раскрывается самая глубокая вершина в текущем фронте поиска. Процедура поиска в глубину отличается от рассмотренной процедуры поиска в ширину тем, что дочерние вершины, получаемые при раскрытии вершины n, помещаются ***в начало*** списка OPEN т.е. принцип формирования списка OPEN соответствует стеку.

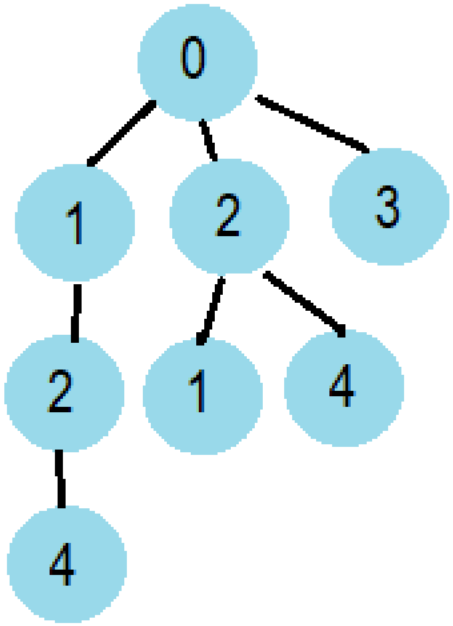
Поиск в глубину требует хранения только единственного пути от корня до листового узла.

Алгоритмы обхода графов - поиск в глубину

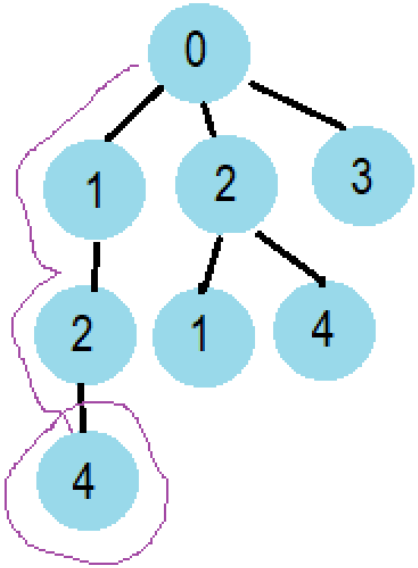
Граф:



Дерево вывода:



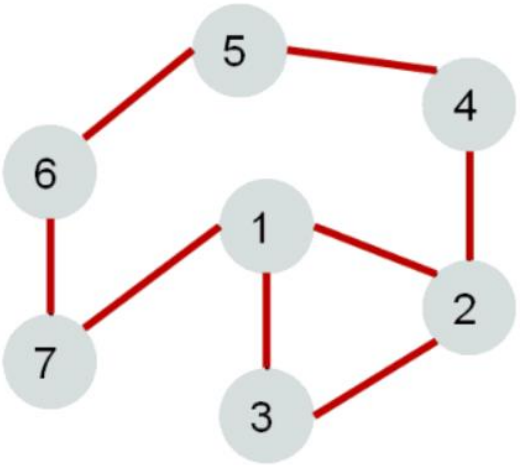
	OPEN	CLOSE
1	0	NULL
2	1 2 3	0
3	(2) 2 3	0 1
4	4 3	0 1 2



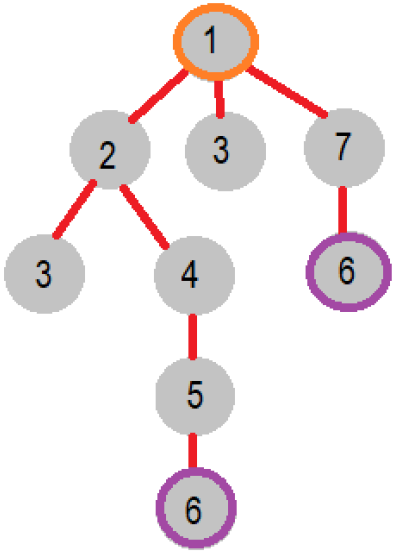
Решение: 0 1 2 4

Алгоритмы обхода графов - поиск в глубину

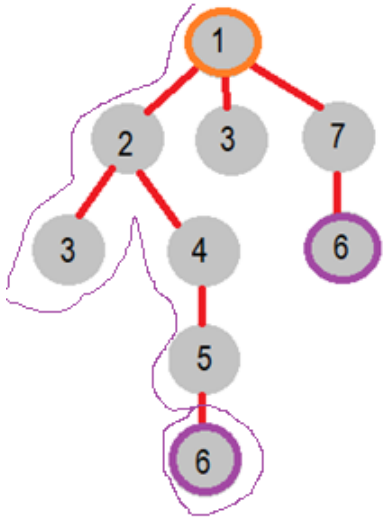
Граф:



Дерево вывода:



	OPEN	CLOSE
1	1	NULL
2	2, 3, 7	1
3	3, 4, 3, 7	1 2
4	4 3 7	1 2 3
5	5 3 7	1 2 3 4
6	6 3 7	1 2 3 4 5



Решение: 1 2 3 4 5 6

Алгоритмы обхода графов - поиск в глубину

Применение алгоритма поиска в глубину:

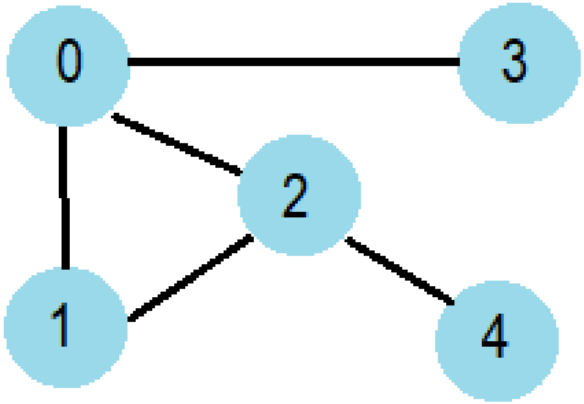
- поиск любого пути в графе;
- поиск лексикографически первого пути в графе;
- проверка, является ли одна вершина дерева предком другой;
- поиск наименьшего общего предка;
- топологическая сортировка;
- поиск компонент связности.

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

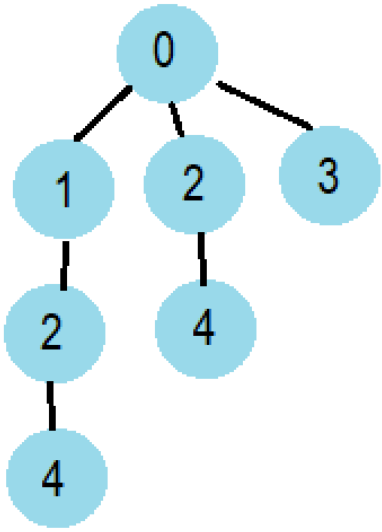
Для реализации алгоритма используют стек или рекурсию.

Сравнение методов поиска

Граф:



Дерево вывода:



поиск в ширину

	OPEN	CLOSE
1	0	NULL
2	1 2 3	0
3	2 3 (2)	0 1
4	3 4	0 1 2
5	4	0 1 2 3

Решение: 0 1 2 3 4

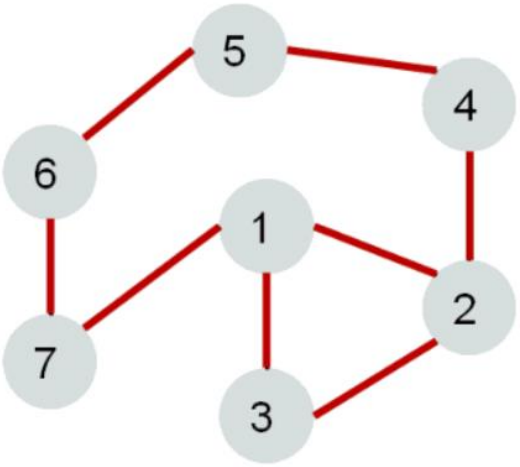
поиск в глубину

	OPEN	CLOSE
1	0	NULL
2	1 2 3	0
3	(2) 2 3	0 1
4	4 3	0 1 2

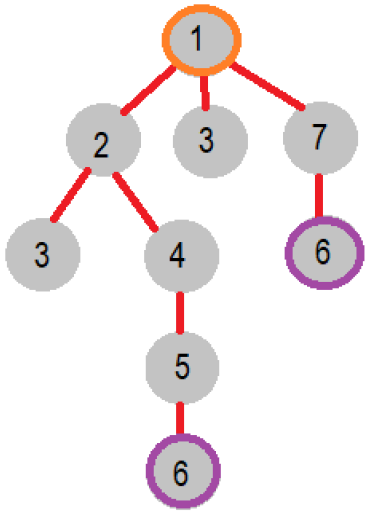
0 1 2 4

Сравнение методов поиска

Граф:



Дерево вывода:



поиск в ширину

	OPEN	CLOSE
1	1	NULL
2	2 3 7	1
3	3 7 (3) 4	1 2
4	7 4	1 2 3
5	4 6	1 2 3 7
6	6 5	1 2 3 7 4

Решение: 1 2 3 7 4 6

поиск в глубину

	OPEN	CLOSE
1	1	NULL
2	2, 3, 7	1
3	3, 4, 3, 7	1 2
4	4 3 7	1 2 3
5	5 3 7	1 2 3 4
6	6 3 7	1 2 3 4 5

1 2 3 4 5 6

Сравнение методов поиска

Вывод: для разных задач, решение разными методами дает разный результат.

Реализация BRS

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue {
    int items[SIZE];  // массив значений
    int front;        // индекс первого
    int rear;         // индекс последнего
};
struct queue* createQueue();  // создание очереди
void enqueue(struct queue* q, int);// добавление в конец очереди
int dequeue(struct queue* q); // взятие следующей нерассм. вершины
void display(struct queue* q);  // просмотр очереди
int isEmpty(struct queue* q);  // проверка на пустоту
void printQueue(struct queue* q);

struct node {   // описание очереди
    int vertex;
    struct node* next;
};
```

Реализация BRS

```
struct node* createNode(int);
struct Graph {
    int numVertices; // кол-во вершин в графе
    struct node** adjLists; // список вершин
    int* visited; // список посещенных вершин(closed)
};
struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void bfs(struct Graph* graph, int startVertex);
int main() {
    struct Graph* graph = createGraph(5);
    addEdge(graph, 0, 3);
    addEdge(graph, 0, 2);
    addEdge(graph, 0, 1);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 4);
    bfs(graph, 0);
    return 0;
}
```

Реализация BRS

```
// -- обход графа в ширину -----------
void bfs(struct Graph* graph, int startVertex) {
      struct queue* q = createQueue();
      graph->visited[startVertex] = 1; // помечаем стартовую вершину как
      enqueue(q, startVertex); // посещенную и добавить в конец очереди
while(!isEmpty(q)) { // пока очередь не пуста
      printQueue(q);     // печатаем содержимое
      int currentVertex = dequeue(q); // берем следующую
                        //нерассмотренную вершину и печатаем ее
      printf("\n Рассмотрим вершину %d\n", currentVertex);
                           // запоминаем текущую вершину
      struct node* temp = graph -> adjLists[currentVertex];
  while(temp) { // пока
      int adjVertex = temp->vertex;
      if(graph->visited[adjVertex] == 0){// если вершина еще не
                                        // посещалась то пометить как посещенную
      graph->visited[adjVertex] = 1;
      enqueue(q, adjVertex); // добавить ее в конец очереди
      } else printf("вершину %d уже рассматривали(не
                                              добавляем)", adjVertex);
    temp = temp->next; // выбор вершины, в которую можно перейти из текущей
   }
  }
}
```

Реализация BRS

```
// ----- создание графа с заданным количеством вершин ----
struct Graph* createGraph(int vertices) {
 struct Graph* graph=(struct Graph*)malloc(sizeof(struct
                                                  Graph));
     graph->numVertices = vertices;
     graph->adjLists=(struct node**)malloc(vertices *
     sizeof(struct node*)); // выделение памяти под max количество

     graph->visited =(int*) malloc(vertices * sizeof(int));
     int i; // инициализация выделенной памяти
     for (i = 0; i < vertices; i++) {
           graph->adjLists[i] = NULL;
           graph->visited[i] = 0;
     }
     return graph;
}
```

Реализация BRS

```
// --- создание вершины  v --------------
struct node* createNode(int v) {
     struct node* newNode = (struct node*)malloc
                                        (sizeof(struct node));
     newNode->vertex = v;
     newNode->next = NULL;
     return newNode;
}

 //---------- добавление ребра --------------------
 void addEdge(struct Graph* graph, int src, int dest) {
     // между вершинами src и dest
     struct node* newNode = createNode(dest);
     newNode->next = graph->adjLists[src];
     graph->adjLists[src] = newNode;

 }
```

Реализация BRS

```
//------- создание очереди (open) --------
struct queue* createQueue() {
    struct queue* q = (struct queue*)malloc
                                (sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// ----- проверка очереди на пустоту ------
int isEmpty(struct queue* q) {
    if(q->rear == -1)
          return 1;
    else
          return 0;
}
```

Реализация BRS

```
// ------ добавление в конец очереди --------
void enqueue(struct queue* q, int value) {
    if(q->rear == SIZE-1)
          printf("\nОчередь заполнена!!");
    else { // заполнение полей
          if(q->front == -1)
                q->front = 0;
          q->rear++;
          q->items[q->rear] = value;
    }
    printf("добавляем в конец очереди %d\n",value);
}
```

Реализация BRS

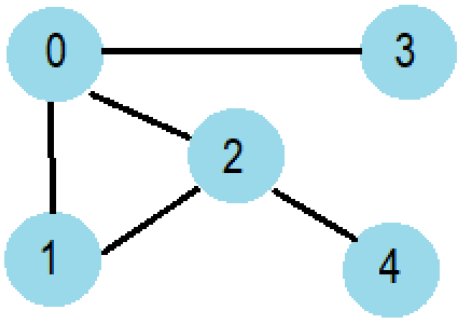
```
//--- получение первого элемента в очереди ---------
int dequeue(struct queue* q) {
     int item;
     if(isEmpty(q)) {
          printf("Очередь пустая");
          item = -1;
     } else {
          item = q->items[q->front];
          q->front++;
          if(q->front > q->rear) {
            printf("Все элементы в очереди рассмотрены");
            q->front = q->rear = -1;
          }
     }
     return item;
}
```

Реализация BRS

```
//-----вывод очереди -------
void printQueue(struct queue *q) {
    int i = q->front;
    if(isEmpty(q)) {
          printf("Очередь пуста");
    } else {
          printf("\nСодержимое очереди: ");
          for(i = q->front; i < q->rear + 1; i++) {
                printf("%d ", q->items[i]);
          }
    }
}
```

Реализация BRS

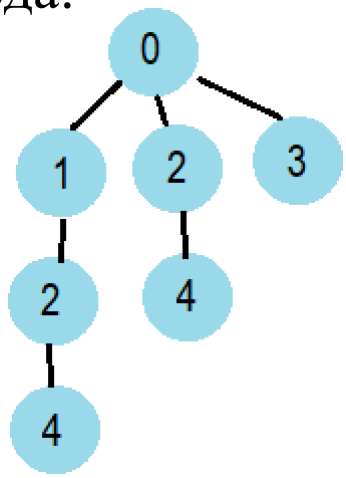
Граф:



Задание в main():

```
addEdge(graph, 0, 3);
addEdge(graph, 0, 2);
addEdge(graph, 0, 1);
addEdge(graph, 1, 2);
addEdge(graph, 2, 4);
```

Дерево вывода:



	OPEN	CLOSE
1	0	NULL
2	1 2 3	0
3	2 3 (2)	0 1
4	3 4	0 1 2
5	4	0 1 2 3

```
добавляем в конец очереди 0

Содержимое очереди: 0 Все элементы в очереди рассмотрены
 Рассмотрим вершину 0
добавляем в конец очереди 1
добавляем в конец очереди 2
добавляем в конец очереди 3

Содержимое очереди: 1 2 3
 Рассмотрим вершину 1
вершину 2 уже рассматривали (не добавляем)
Содержимое очереди: 2 3
 Рассмотрим вершину 2
добавляем в конец очереди 4

Содержимое очереди: 3 4
 Рассмотрим вершину 3

Содержимое очереди: 4 Все элементы в очереди рассмотрены
 Рассмотрим вершину 4


--------------------------------
```

Визуализация графовых моделей

Еще одной из задач решаемых с графами является их визуализация.

Визуализация – это процесс преобразования больших и сложных видов абстрактной информации в интуитивно-понятную визуальную форму. Другими словами, когда мы рисуем то, что нам непонятно – и сразу все встает на свои места.

Графы – и есть помощники в этом деле. Они помогают представить любую информацию, которую можно промоделировать в виде объектов и связей между ними.

Граф можно нарисовать на плоскости или в трехмерном пространстве. Его можно изобразить целиком, частично или иерархически.

Изобразительное соглашение – одно из основных правил, которому должно удовлетворять изображение графа, чтобы быть допустимым. Например, при изображении блок-схемы программы можно использовать соглашение о том, что все вершины должны изображаться прямоугольниками, а дуги – ломаными линиями с вертикальными и горизонтальными звеньями. При этом, конкретный вид соглашения может быть достаточно сложен и включать много деталей.

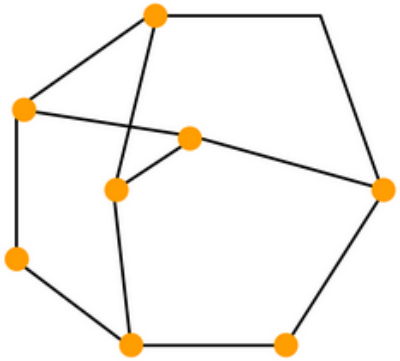
Визуализация графовых моделей

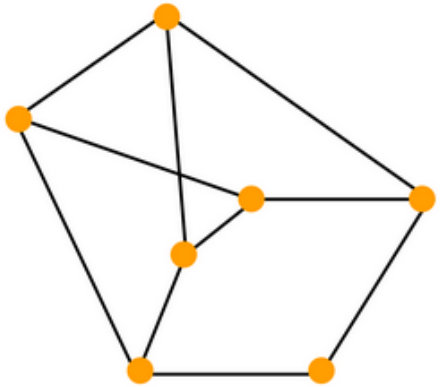
Виды изобразительных соглашений:

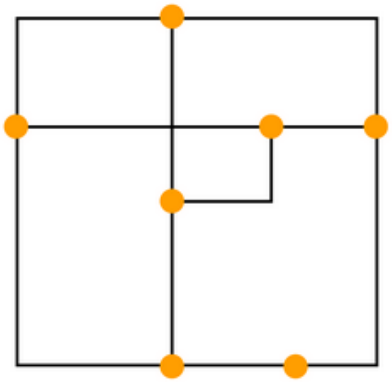
- **полилинейное** изображение – каждое ребро графа рисуют в виде ломаной линии

- **прямолинейное** изображение – каждое ребро представляют с помощью отрезка прямой

- **ортогональное** изображение – каждое ребро графа изображается в виде ломаной линии, состоящей из чередующихся горизонтальных и вертикальных сегментов.

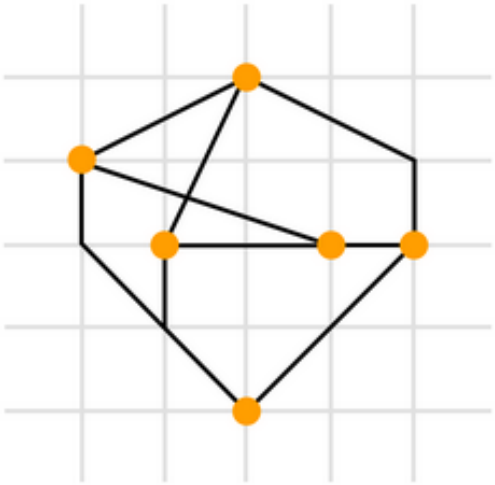






Визуализация графовых моделей

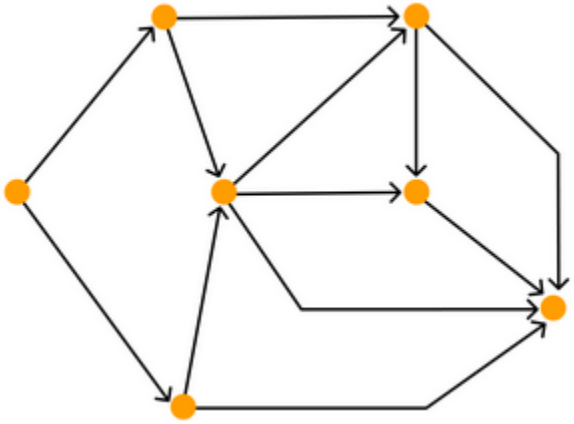
- **сетчатое** изображение – все вершины, а также все точки пересечения и сгибы ребер имеют целочисленные координаты. То есть находятся в узлах координатной сетки, образованной прямыми, параллельными координатным осям и пересекающими их в точках с целочисленными координатами

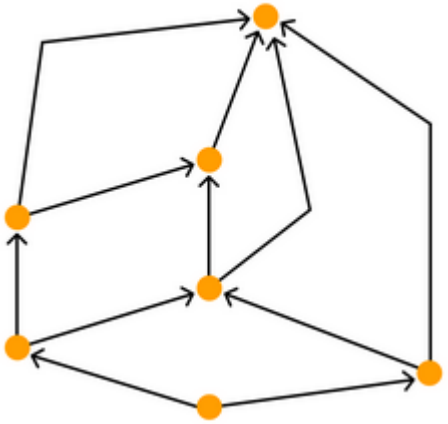


Визуализация графовых моделей

- **плоское** изображение предполагает отсутствие точек пересечения у линий, изображающих ребра.

- **восходящее или нисходящее** изображение имеет смысл по отношению к ациклическому орграфу и предполагает, что каждая дуга изображается ориентированной линией, координаты точек которой монотонно изменяются в направлении снизу вверх или сверху вниз, а также слева направо.





Визуализация графовых моделей

Графы – это мощный инструмент для изучения систем, представленных в виде графа, а также поиска решений задач во многих областях.