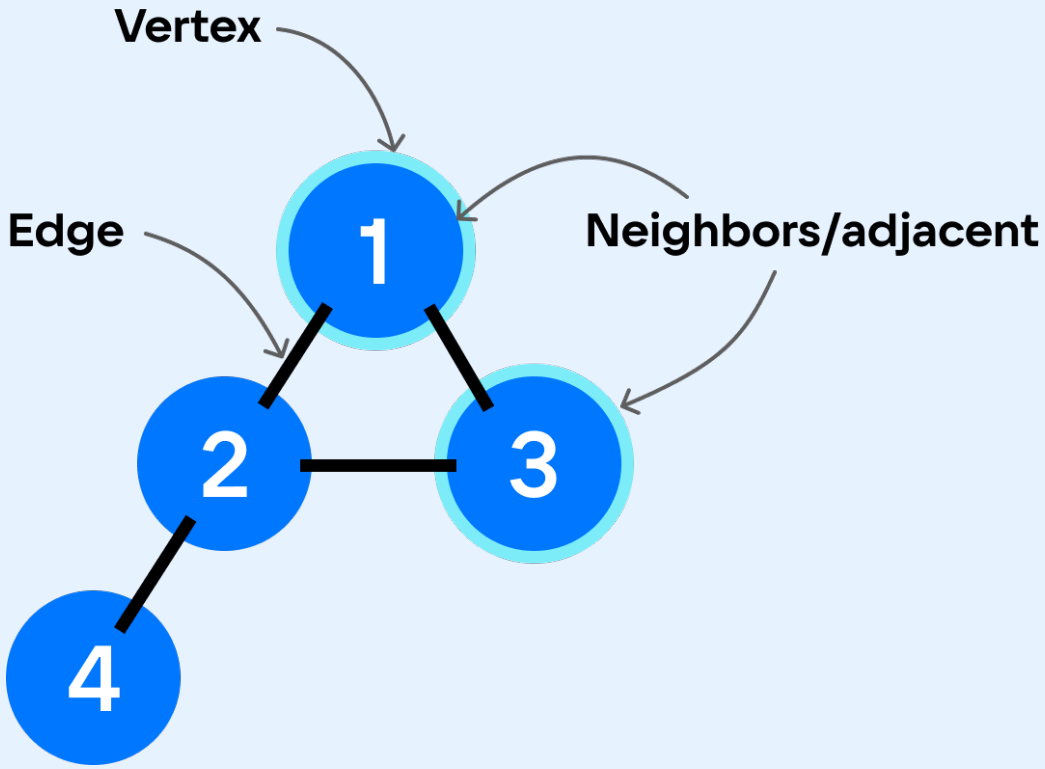
Графы

education

Графы

- Граф, в котором отсутствуют циклы, — ациклический. Ациклический связный граф называется деревом. Множество деревьев — лес (forest).
- Остовное дерево связного графа — это подграф, который содержит все вершины этого графа и представляет собой единое дерево.
- Остовной лес графа – это лес, который содержит все вершины этого графа.

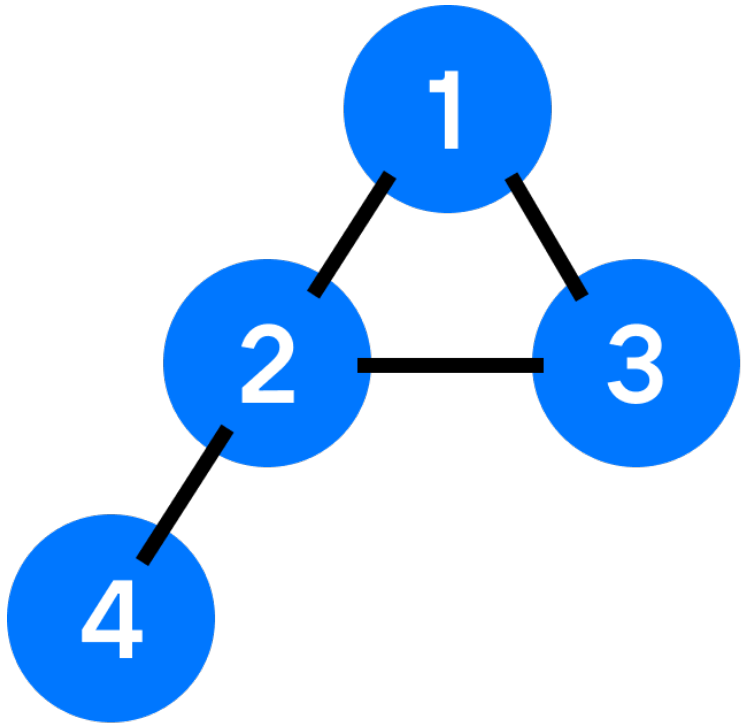


Важно:

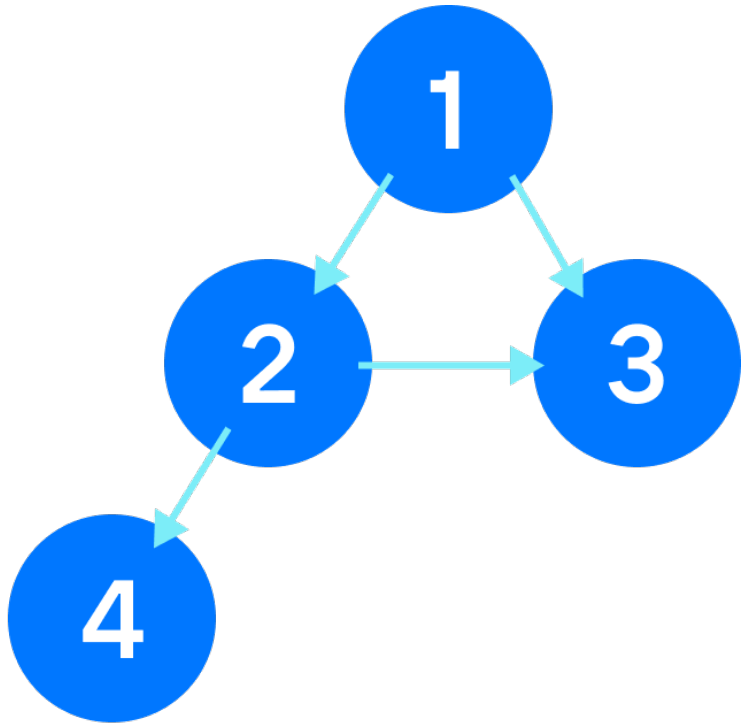
граф, состоящий из V вершин (vertex), содержит не более V (V − 1) / 2 рёбер.

education

Ненаправленные и направленные графы



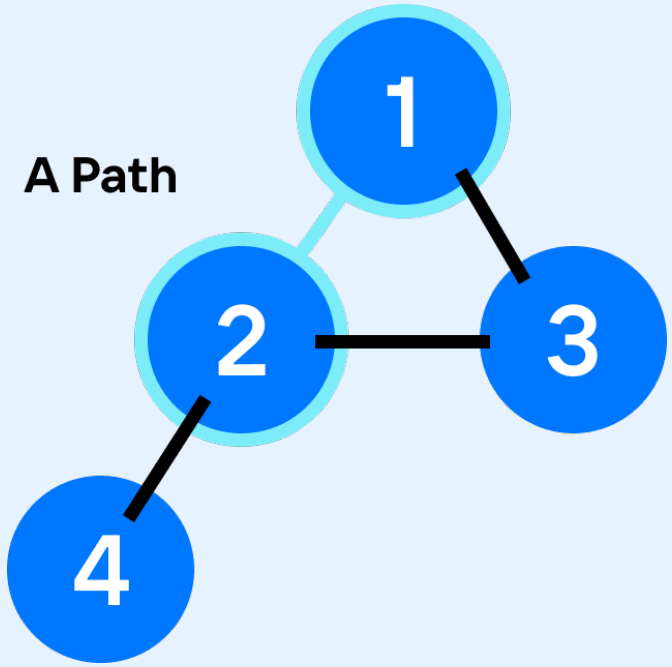
Undirected Graph

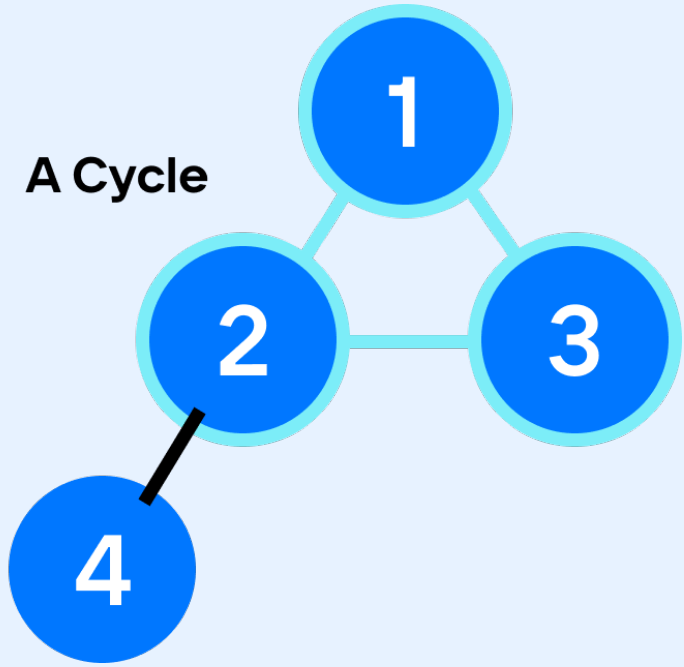


Directed Graph

Графы

- Путь — это последовательность вершин.
- Цикл — это путь, который начинается и заканчивается в одной и той же вершине.

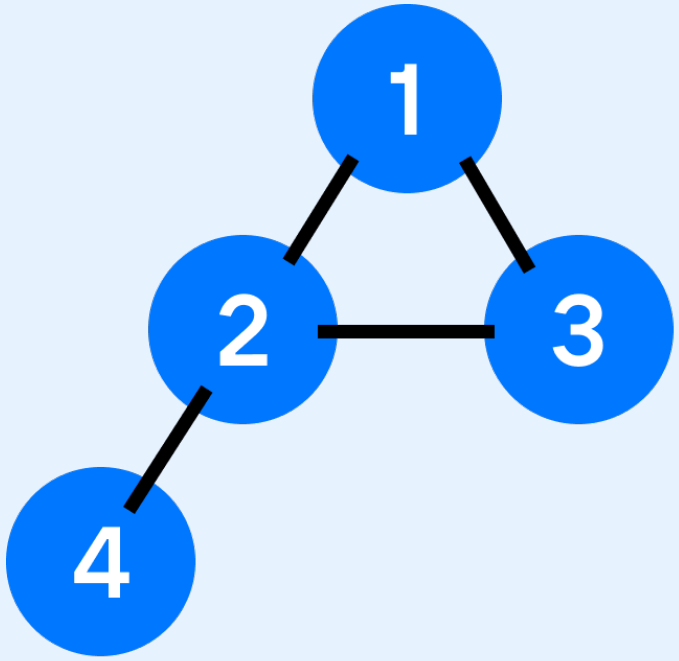




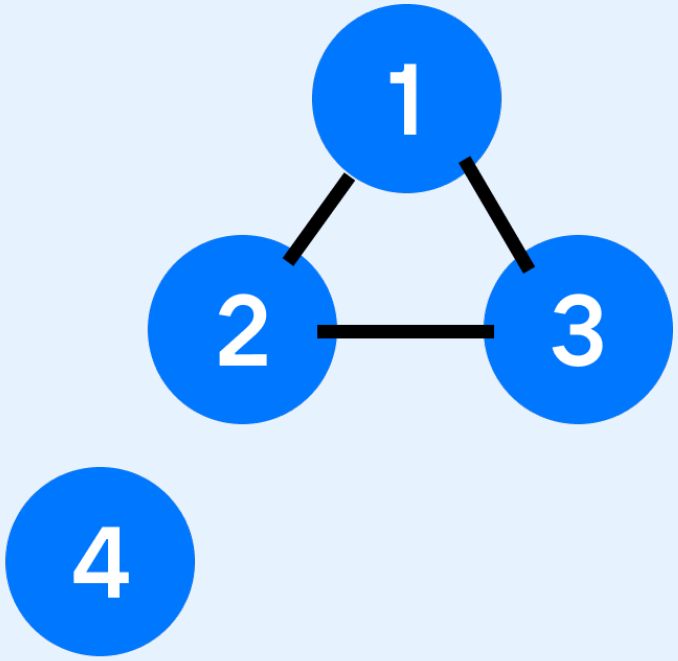
education

Графы

- Неориентированный граф называется связным, если каждая его вершина соединена путём с другой вершиной. В противном случае он отключён.



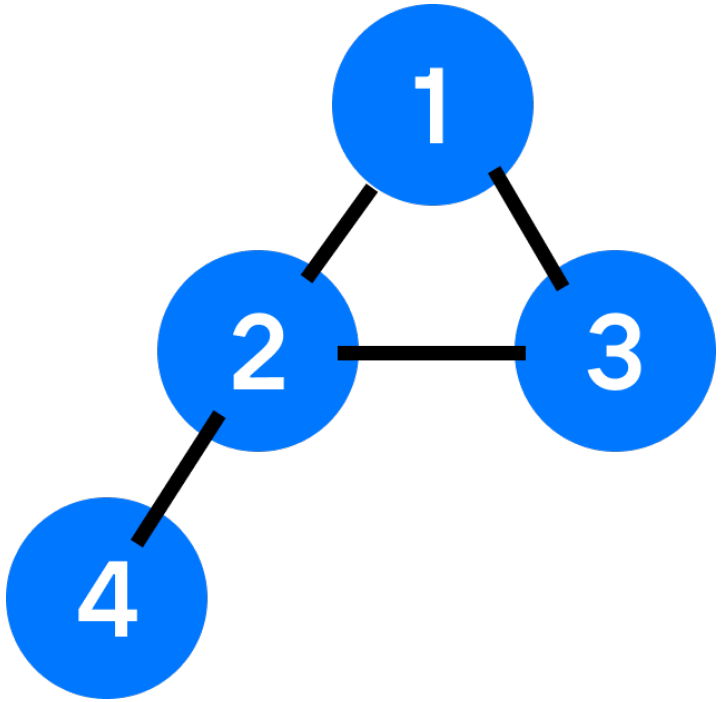
Connected



Disconnected

education

Списки смежности (Adjacency list)



```
{
    1: [2, 3],
    2: [1, 3, 4],
    3: [1, 2],
    4: [2]
}
```

Реализация неориентированного графа

```
#include <map>
#include <vector>
#include <set>
class Graph {
public:
    void addEdge(int v, int w) {
        adjList[v].push_back(w);
        adjList[w].push_back(v);
    }
    const std::vector<int>& getAdjVertices(int v)
const {
        return adjList.at(v);
    }
    int getNumVertices() const {
        return adjList.size();
    }


    int countNodesDFS(int start) {
        std::set<int> visited;
        DFS(start, visited);
        return visited.size();
    }
```

```
private:
    std::map<int, std::vector<int>> adjList;
    void DFS(int v, std::set<int>& visited) {
        visited.insert(v);
        for (int neighbor : getAdjVertices(v)) {
            if (visited.find(neighbor) == visited.end()) {
                DFS(neighbor, visited);
            }
        }
    }
};
```

Реализация неориентированного графа

```
#include <map>
#include <vector>
#include <set>
class Graph {
public:
    void addEdge(int v, int w) {
        adjList[v].push_back(w);
        adjList[w].push_back(v);
    }
    const std::vector<int>& getAdjVertices(int v)
const {
        return adjList.at(v);
    }
    int getNumVertices() const {
        return adjList.size();
    }

    int countNodesDFS(int start) {
        std::set<int> visited;
        DFS(start, visited);
        return visited.size();
    }
```

Реализация неориентированного графа

```
#include <map>
#include <vector>
#include <set>
class Graph {
public:
    void addEdge(int v, int w) {
        adjList[v].push_back(w);
        adjList[w].push_back(v);
    }
    const std::vector<int>& getAdjVertices(int v)
const {
        return adjList.at(v);
    }
    int getNumVertices() const {
        return adjList.size();
    }

    int countNodesDFS(int start) {
        std::set<int> visited;
        DFS(start, visited);
        return visited.size();
    }
```

Реализация неориентированного графа

```
#include <map>
#include <vector>
#include <set>
class Graph {
public:
    void addEdge(int v, int w) {
        adjList[v].push_back(w);
        adjList[w].push_back(v);
    }
    const std::vector<int>& getAdjVertices(int v)
const {
        return adjList.at(v);
    }
    int getNumVertices() const {
        return adjList.size();
    }
```

```
    int countNodesDFS(int start) {
        std::set<int> visited;
        DFS(start, visited);
        return visited.size();
    }
```

Реализация неориентированного графа

```
private:
    std::map<int, std::vector<int>> adjList;
    void DFS(int v, std::set<int>& visited) {
        visited.insert(v);
        for (int neighbor : getAdjVertices(v)) {
            if (visited.find(neighbor) ==
visited.end()) {
                DFS(neighbor, visited);
            }
        }
    }
};
```

education

Реализация неориентированного графа

```
Graph g;

    g.addEdge(1, 2);

    g.addEdge(1, 3);

    g.addEdge(2, 3);

    g.addEdge(2, 4);

    g.addEdge(3, 5);

    g.addEdge(4, 5);


    std::cout << "Number of nodes reachable from node 1: " << g.countNodesDFS(1) << std::endl;
```

Поиск в графе

```
#include <iostream>
#include <map>
#include <iostream>
#include <map>
#include <list>
#include <queue>
#include <string>
class Node
{
public:
    int id;
    std::string info;
    Node(const int id, std::string info) : id(id), info(std::move(info)) {}
};
class Graph {
private:
    std::map<int, std::list<int>> adjList; // adjacency list
    std::map<int, Node*> nodes; // map to store the Node objects
```

Поиск в графе

```
#include <iostream>
#include <map>
#include <iostream>
#include <map>
#include <list>
#include <queue>
#include <string>
class Node
{
public:
    int id;
    std::string info;
    Node(const int id, std::string info) : id(id), info(std::move(info)) {}
};
class Graph {
private:
    std::map<int, std::list<int>> adjList; // adjacency list
    std::map<int, Node*> nodes; // map to store the Node objects
```

Поиск в графе

```
public:
  void AddNode(const int id, std::string info) {
    Node* newNode = new Node(id, std::move(info));
    nodes[id] = newNode;
  }
  void AddEdge(const int id1, const int id2) {
    adjList[id1].push_back(id2);
    adjList[id2].push_back(id1); // for undirected graph
  }
  void DFS(int startId) {
    std::map<int, bool> visited;
    DFSUtil(startId, visited);
  }
  void BFS(int startId) {
    std::map<int, bool> visited;
    std::queue<int> queue;
    visited[startId] = true;
    queue.push(startId);
    while (!queue.empty()) {
      int node = queue.front();
      std::cout << "Visited " << node << std::endl;
      queue.pop();
      for (auto i : adjList[node]) {
        if (!visited[i]) {
          queue.push(i);
          visited[i] = true;
        }
      }
    }
  }
```

Поиск в графе

```
~Graph() {
    for (auto& pair : nodes) {
        delete pair.second;
    }
}


private:
    void DFSUtil(int v, std::map<int, bool>& visited) {
        visited[v] = true;
        std::cout << "Visited " << v << std::endl;
        std::list<int>::iterator i;
        for (i = adjList[v].begin(); i != adjList[v].end(); ++i) {
            if (!visited[*i])
                DFSUtil(*i, visited);
        }
    }
};
```

Поиск в графе

```
/* Graph g;
 g.AddNode(1, "Node 1");
 g.AddNode(2, "Node 2");
 g.AddNode(3, "Node 3");
 g.AddEdge(1, 2);
 g.AddEdge(1, 3);
 std::cout << "DFS traversal starting from node 1:" << std::endl;
 g.DFS(1);
 std::cout << "BFS traversal starting from node 1:" << std::endl;
 g.BFS(1);
 return 0;*/
```

Реализация (простой пример)

DFS traversal starting from node 1:

- Visited 1
- Visited 2
- Visited 3

BFS traversal starting from node 1:

- Visited 1
- Visited 2
- Visited 3

Реализация (сложнее)

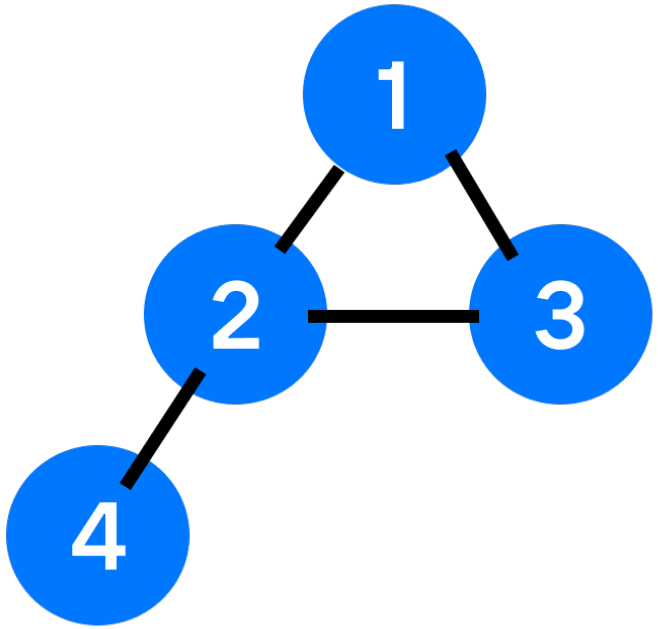
```
Graph g;
  g.AddNode(1, "Node 1");
  g.AddNode(2, "Node 2");
  g.AddNode(3, "Node 3");
  g.AddNode(4, "Node 4");
  g.AddNode(5, "Node 5");
  g.AddEdge(1, 2);
  g.AddEdge(1, 3);
  g.AddEdge(2, 4);
  g.AddEdge(3, 5);
  std::cout << "DFS traversal starting from node 1:" << std::endl;
  g.DFS(1);
  std::cout << "BFS traversal starting from node 1:" << std::endl;
  g.BFS(1);
  return 0;
```

Реализация (сложнее)

- DFS traversal starting from node 1:
- Visited 1 Visited 2 Visited 4 Visited 3 Visited 5
- BFS traversal starting from node 1:
- Visited 1 Visited 2 Visited 3 Visited 4 Visited 5

DFS (визуализация)

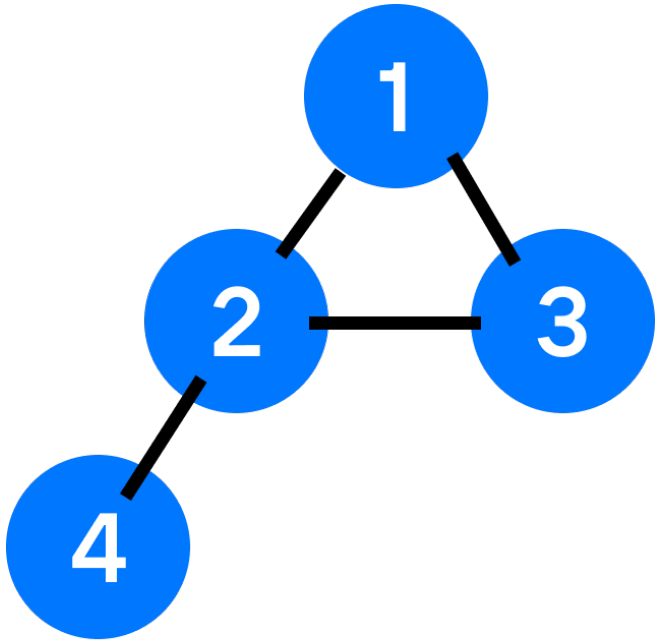
DFS



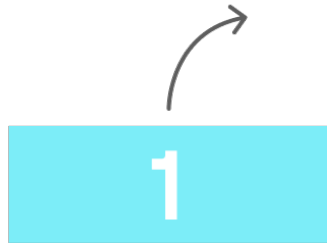
Visited: 1

DFS stack

DFS (визуализация)



Visited: 1



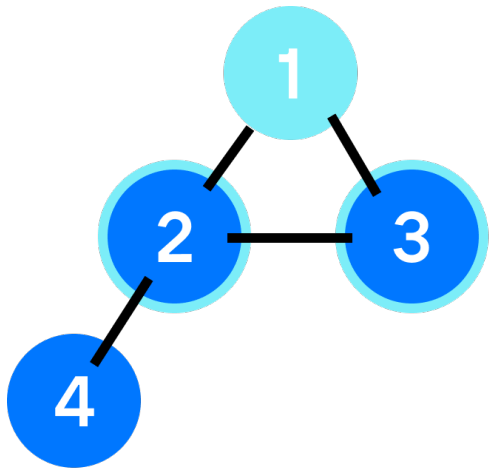
DFS stack

DFS stack

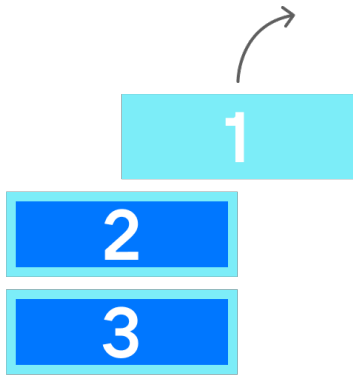
education

Pop first element and add it to visited

DFS (визуализация)

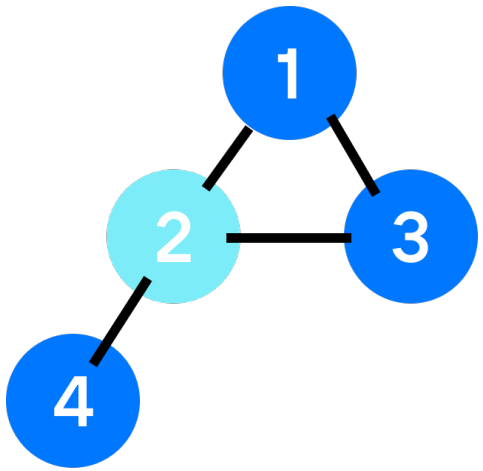


Visited: 1, 2, 3

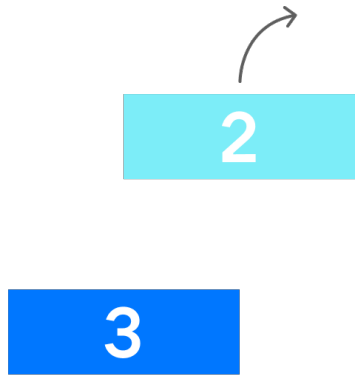


DFS stack

Add 1's neighbors to stack and visited



Visited: 1, 2, 3

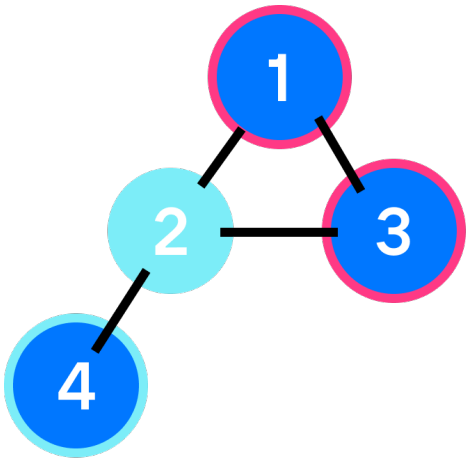


DFS stack

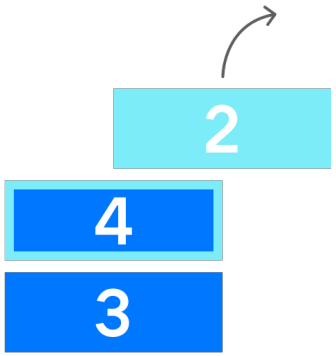
Pop first element

education

DFS (визуализация)

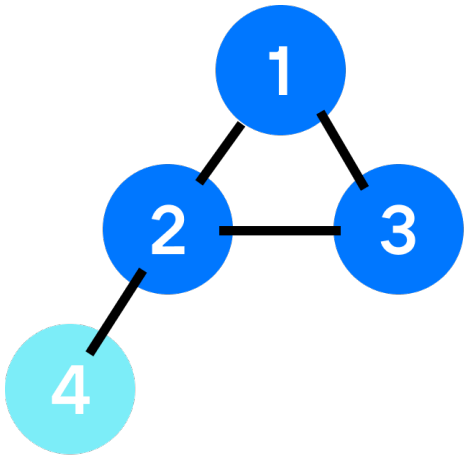


Visited: 1, 2, 3

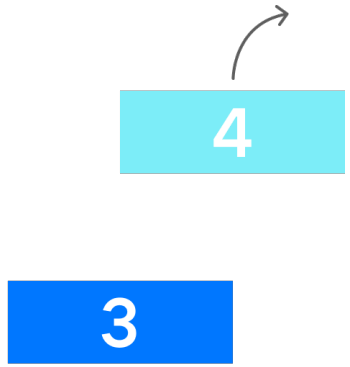


DFS stack

Add 2's neighbors to stack
1 is already visited, skip
3 is already visited, skip
Add 4 to stack and visited



Visited: 1, 2, 3, 4

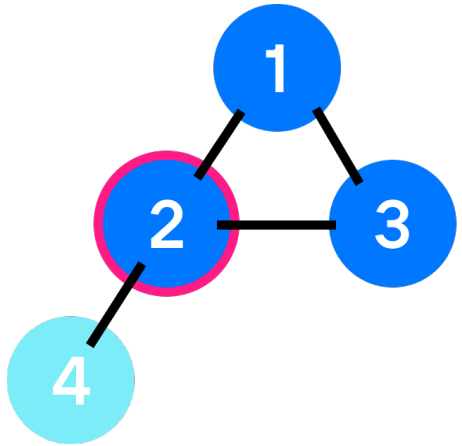


DFS stack

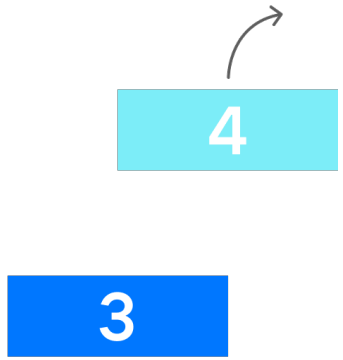
Pop first element, add to visited

education

DFS (визуализация)

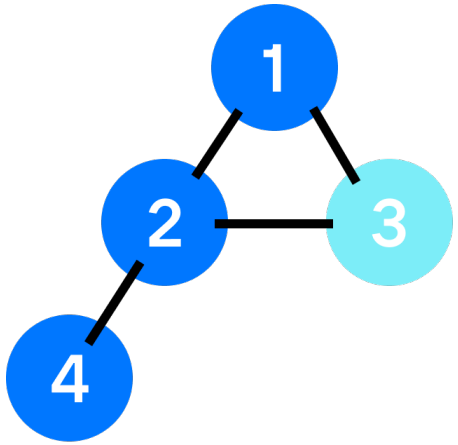


Visited: 1, 2, 3, 4



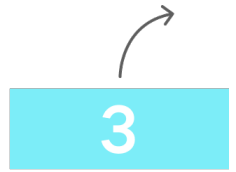
DFS stack

Add 4's neighbors to stack
2 is already visited, skip



Pop first element

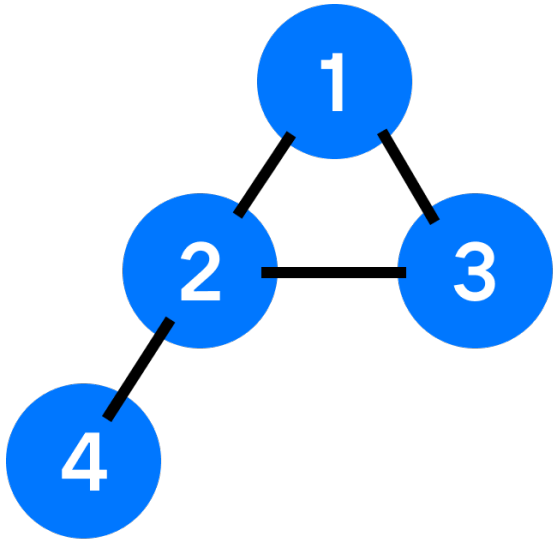
Visited: 1, 2, 3, 4



DFS stack

education

DFS (визуализация)



Visited: 1, 2, 3, 4

DFS stack

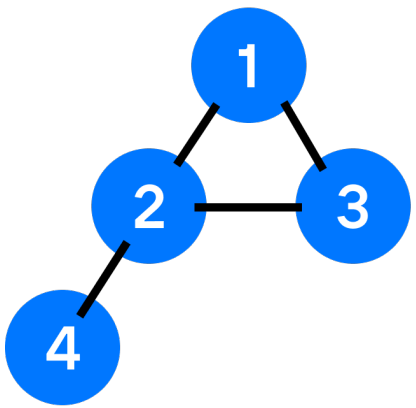
Stack is empty, DFS complete

Реализация (простой пример)

- В некоторых реализациях DFS стек используется явно (т. е. в коде явно присутствует структура данных стека), а в некоторых — нет.
- Например, в рекурсивной версии DFS стек вызовов функций в языке программирования заменяет явно использованный стек.
- Каждый рекурсивный вызов функции добавляет новый уровень на стек вызовов, а возвращение из функции удаляет уровень со стека.

DFS (визуализация)

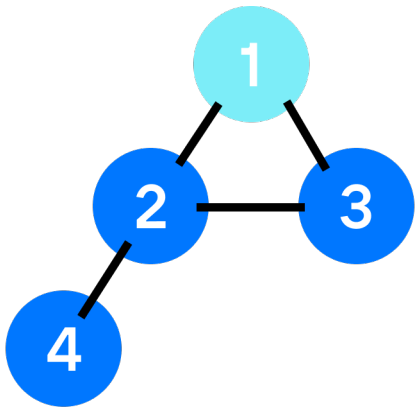
BFS



Visited: 1

Queue

Initially, vertex 1 in queue and visited

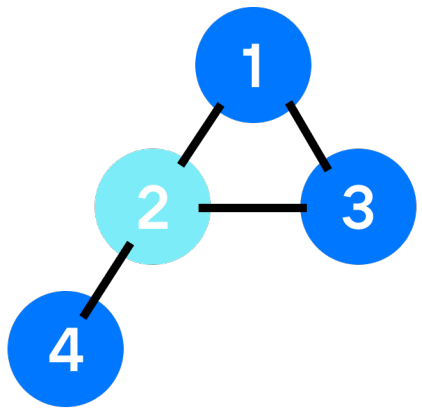


Visited: 1

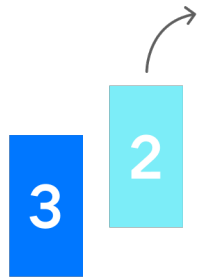
Dequeue first element

education

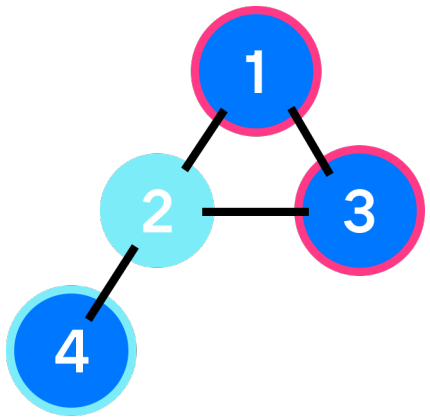
DFS (визуализация)



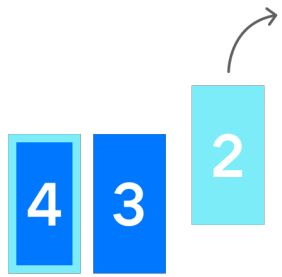
Visited: 1, 2, 3



Dequeue first element



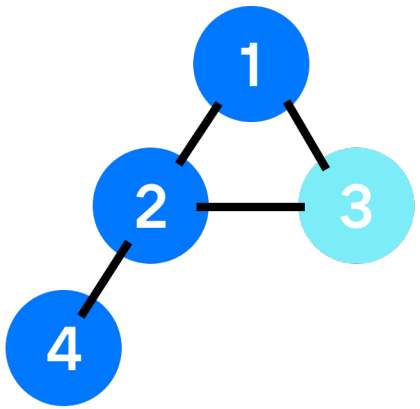
Visited: 1, 2, 3, 4



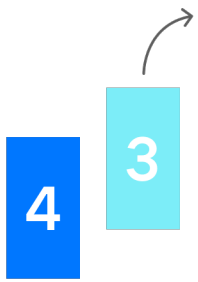
Add 2's neighbors - 3,4
1 is already visited, skip
3 is already visited, skip
4 not in visited, enqueue and add to visited

education

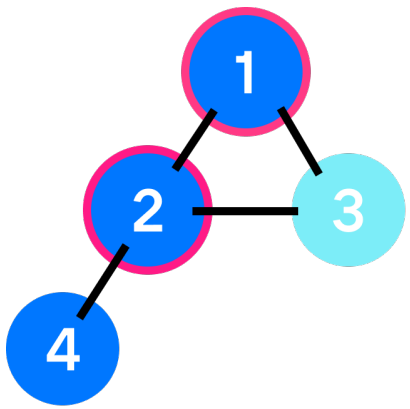
DFS (визуализация)



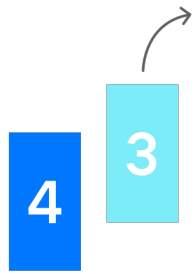
Visited: 1, 2, 3, 4



Dequeue first element



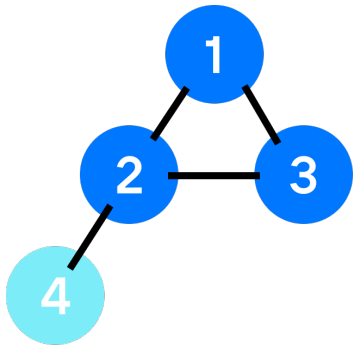
Visited: 1, 2, 3, 4



3's neighbors are all in visited, skip

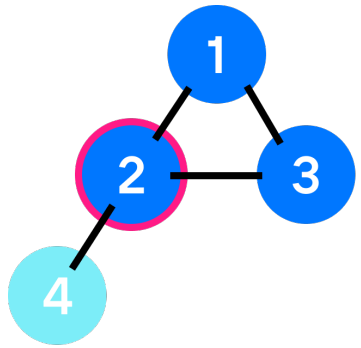
education

DFS (Визуализация)



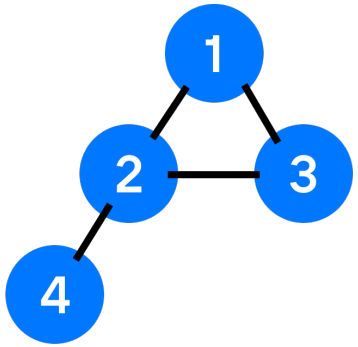
Visited: 1, 2, 3, 4

Dequeue first element



Visited: 1, 2, 3, 4

4's neighbors 2 is in visited, skip



Visited: 1, 2, 3, 4

Queue empty, BFS complete

education

Решение задачи

- Существует двунаправленный граф с n вершинами, где каждая вершина помечена от 0 до n − 1 (включительно).
- Рёбра в графе представлены как рёбра двумерного целочисленного массива, где каждое ребро [i] = [ui, vi] обозначает двунаправленное ребро между вершиной ui и вершиной vi. Каждая пара вершин соединена не более чем одним ребром, и ни одна вершина не имеет рёбра сама по себе.

Решение задачи

- Вы хотите определить, существует ли допустимый путь от источника вершины до места назначения вершины.
- Учитывая ребра и целые числа n, источник и пункт назначения, возвращайте true, если существует действительный путь от источника к месту назначения, или false в противном случае.

```
class Solution {
public:
    bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
        vector<vector<int>> graph(n);
        vector<bool> visited(n, false);
        // build the adjacency list
        for (const auto& edge : edges) {
            int u = edge[0], v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }
        // start the DFS
        return dfs(graph, visited, source, destination);
    }
```

```
class Solution {
public:
    bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
        vector<vector<int>> graph(n);
        vector<bool> visited(n, false);
        // build the adjacency list
        for (const auto& edge : edges) {
            int u = edge[0], v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }
        // start the DFS
        return dfs(graph, visited, source, destination);
    }
```

```
private:
    bool dfs(vector<vector<int>>& graph, vector<bool>& visited, int source, int destination) {
        // if the source is the destination, we have found a valid path
        if (source == destination) {
            return true;
        }
         // mark the current vertex as visited
        visited[source] = true;
        // visit all the vertices adjacent to the current vertex
        for (int next : graph[source]) {
            // if the next vertex is not visited, continue the DFS
            if (!visited[next]) {
                if (dfs(graph, visited, next, destination)) {
                    return true;
                }
            }
        }
        // if no path is found, return false
        return false;
    }
};
```

Будем ВКонтакте!