Графы: основы теории, алгоритмы поиска

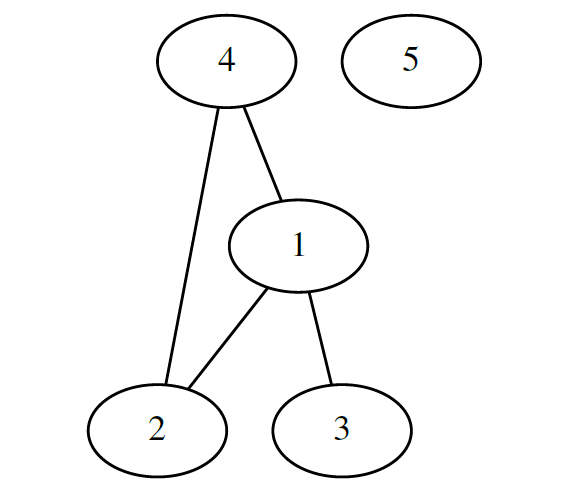
Возможно, вы уже знакомы с понятием спортивного программирования и знаете, что оно помогает развить навыки решения проблем и прокачать технические знания о структурах данных и алгоритмах.

Одной из важнейших составляющих спортивного программирования является изучение алгоритмов. В этой статье мы охватим большое количество алгоритмов, в том числе все алгоритмы на графах, знание которых понадобится вам для успешного решения задач из теории графов на соревнованиях по программированию. Конечно, одних теоретических знаний алгоритмов недостаточно, и вам придётся набить руку в решении практических задач на таких сайтах, как [Codeforces](https://codeforces.com/" \t "_blank). Настоящая же статья представит вам инструменты, необходимые для освоения важных графовых алгоритмов. Итак, приступим.

**Что такое граф?**

Графы, в понимании программистов, — это не те графики, которые мы изучали в школе. Это не столбиковые диаграммы или гистограммы.

С точки зрения компьютерных наук и дискретной математики, графы — это абстрактный способ представления типов отношений, например дорог, соединяющих города, и других видов сетей. Графы состоят из рёбер и вершин. Вершина — это точка на графе, а ребро — это то, что соединяет две точки на графе.



Пример графа

В условиях задач из теории графов на соревнованиях по программированию обычно говорится о таких вещах, как сети и решётки.

Вот список всех терминов, относящихся к теории графов, которые вам нужно знать:

* **путь**— последовательность рёбер, соединяющая разные (неповторяющиеся) вершины;
* **маршруты** — это те же пути, только они не требуют последовательности разных вершин;
* **цикл**— группа вершин, связанных вместе в замкнутую цепь. На рисунке выше вершины [1,2,4] составляют цикл;
* **связный граф**— граф, в котором между любой парой вершин имеется один путь;
* **дерево**— связный граф, не содержащий цикла;
* **неориентированный граф**— граф, в котором рёбра не имеют направления. На рисунке выше показан как раз неориентированный граф. В таком неориентированном графе можно перемещаться вдоль ребра в любом из двух направлений;
* **ориентированный граф**— граф, в котором рёбра имеют направления и обозначаются стрелками. В таком ориентированном графе можно перемещаться вдоль ребра только в указанном направлении.

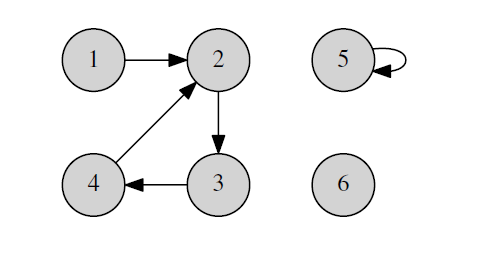
**Представление графов в коде**

Для того, чтобы использовать алгоритмы на графах в коде, сначала нам нужно разобраться, как осуществляется представление графов в коде. Весь следующий код будет на C++, так как для спортивного программирования я предпочитаю именно этот язык за его скорость и встроенные функции, позволяющие упростить написание решений задач.

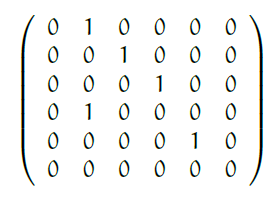
Будут показаны два способа представления графов: матрицы смежности и списки смежности. Больше вам пригодится представление графов в виде списка смежности.

**Матрицы смежности**

Матрица смежности представляет собой граф в виде двумерной матрицы с размерами V x V, где V — количество вершин графа. Матрицы смежности лучше всего применять, когда V² примерно равно E (числу рёбер), то есть когда граф плотный. Запись a\_ij обозначает, сколько рёбер соединяют вершину i и вершину j.



Ориентированный граф, ниже показана его матрица смежности



Матрица 6 x 6, представляющая ориентированный граф выше.

**Код:**

Следующий код используется для создания матрицы смежности *неориентированного* графа. Чтобы код создавал матрицу смежности для *ориентированного* графа, измените функцию add\_edge, удалив вторую строку внутри неё: matrix[v][u] = 1;

#include<bits/stdc++.h>  
using namespace std;  
int matrix[20][20]; //матрица смежности изначально 0  
int count = 0;  
//следующая функция используется для вывода  
void displayMatrix(int v) {  
 int i, j;  
 for(i = 0; i < v; i++) {  
 for(j = 0; j < v; j++) {  
 cout << matrix[i][j] << " ";  
 }  
 cout << endl;  
 }  
}  
void add\_edge(int u, int v){ //функция добавления ребра в матрицу  
 matrix[u][v] = 1;  
 matrix[v][u] = 1;  
}  
main(int argc, char\* argv[]) {  
 int v = 6; //в графе 6 вершин  
 add\_edge(0, 4);  
 add\_edge(0, 3);  
 add\_edge(1, 2);  
 add\_edge(1, 4);  
 add\_edge(1, 5);  
 add\_edge(2, 3);  
 add\_edge(2, 5);  
 add\_edge(5, 3);  
 add\_edge(5, 4);  
 displayMatrix(v);  
}

**Списки смежности**

Другой распространенный способ представления графов в коде — списки смежности. Суть в том, что вы создаёте списки соседей для каждой вершины, а затем помещаете все эти списки в другой список. Их лучше всего применять, когда в графе небольшое количество рёбер, то есть когда граф разрежённый. Если у вас взвешенный граф, т.е. каждое ребро графа имеет какой-то вес, то в списке будут содержаться пары для рёбер (сосед, вес).

**Код:**

#include<bits/stdc++.h>using namespace std;void addEdge(vector<int> adj[], int u, int v){  
 adj[u].push\_back(v);   
 adj[v].push\_back(u);//удаляем эту строку для ориентированных графов  
}int main(){  
 int v = 5; //5 вершин   
 vector<int> adj[v];  
 addEdge(adj, 0, 1);  
 addEdge(adj, 0, 4);  
 addEdge(adj, 1, 2);  
 addEdge(adj, 1, 3);  
 addEdge(adj, 1, 4);  
 addEdge(adj, 2, 3);  
 addEdge(adj, 3, 4);   
}

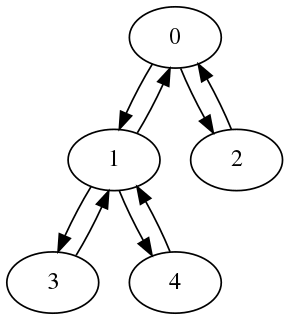
**Поиск в глубину**

Теперь, когда мы научились представлять графы в коде, можем приступить к изучению некоторых алгоритмов на графах! Начнём с поиска в глубину (DFS) и завершим поиском в ширину (BFS).

Поиск в глубину — это один из базовых алгоритмов на графах. Он применяется для поиска расстояния от одной вершины до других вершин в графе. Это **алгоритм обхода.**

Поиск в глубину помечает каждую вершину в графе одной из двух меток: посещённая или не посещённая. Алгоритм помечает каждую вершину как посещённую, если удаётся избежать циклов. Он работает следующим образом:

1. Помещаем любую из вершин графа на стек.
2. Берём элемент со стека и добавляем его в список посещённых.
3. Создаём список соседей этой вершины. Добавляем в стек те, что не находятся в списке посещённых.
4. Повторяем 2 и 3 пункты, пока стек не опустеет.



Визуальное отображение алгоритма поиска в глубину. Поиск начинается в вершине 1.

**Код:**

#include <bits/stdc++.h>using namespace std;const int N = 100000;  
vector<int> adj[N];  
bool visited[N];void dfs(int *s*) {  
 if (visited[s]) return;  
 visited[s] = true;for (auto u: adj[s]) {  
 dfs(u);  
 }  
}

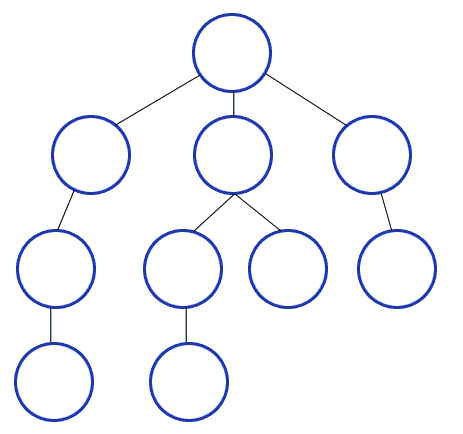
**Поиск в ширину**

Поиск в ширину — ещё один алгоритм обхода графов. Вместе с алгоритмом поиска вглубь он составит большую часть увлекательных соревнований по программированию, по крайней мере, тех из них, что относятся к графам.

Поиск в ширину тоже помещает каждую вершину в графе в одну из двух категорий: посещённых или непосещённых. И цель у обоих алгоритмов одна и та же: помечать каждую вершину в графе как посещённую, если удаётся избежать циклов. Вот как работает алгоритм поиска в ширину:

1. Помещаем любую вершину в графе в конец очереди.
2. Берём элемент в начале очереди и добавляем его в список посещённых.
3. Создаём список соседей этой вершины. Добавляем в конец очереди непосещённые.
4. Повторяем 2 и 3 пункты, пока очередь не опустеет.

Как видите, алгоритм поиска в ширину очень похож на алгоритм поиска в глубину. Однако вместо того, чтобы спускаться вниз по ветви графа или дерева, как это делает алгоритм поиска в глубину, алгоритм поиска в ширину проходит каждый уровень.



Визуальное отображение алгоритма поиска в ширину. Поиск начинается в вершине 1.

**Код:**

*// Быстрая реализация алгоритма поиска в ширину с использованием  
// векторов и очереди*#include <bits/stdc++.h>  
#define pb push\_backusing namespace std;vector<bool> v;  
vector<vector<int> > g;void edge(int *a*, int *b*){  
 g[a].pb(b);  
 *// для неориентированного графа добавляем эту строку  
 // g[b].pb(a);*}void bfs(int *u*){  
 queue<int> q;  
 q.push(u);  
 v[u] = true;  
 while (!q.empty()) {  
 int f = q.front();  
 q.pop();  
 *// Ставим в очередь все соседние F и помечаем их как посещённые* for (auto i = g[f].begin(); i != g[f].end(); i++) {  
 if (!v[\*i]) {  
 q.push(\*i);  
 v[\*i] = true;  
 }  
 }  
 }  
}

**Заключение**

Освоив теоретическую часть, касающуюся двух самых важных алгоритмов обхода на графах, вам остаётся только практиковаться, чтобы использовать эти алгоритмы в соревнованиях по программированию. Я бы порекомендовал для начала [Codeforces](http://codeforces.com/" \t "_blank): решайте задачи, помеченные тегами bfs и dfs с рейтингом до 1400. Когда почувствуете, что справляетесь с ними, увеличьте сложность.