**Алгоритм поиска в глубину**

**Алгоритм поиска (или обхода) в глубину** (англ. depth-first search, DFS) позволяет построить обход ориентированного или неориентированного графа, при котором посещаются все вершины, доступные из начальной вершины.

Отличие поиска в глубину от поиска в ширину заключается в том, что (в случае неориентированного графа) результатом алгоритма поиска в глубину является некоторый маршрут, следуя которому можно обойти последовательно все вершины графа, доступные из начальной вершины. Этим он принципиально отличается от поиска в ширину, где одновременно обрабатывается множество вершин, в поиске в глубину в каждый момент исполнения алгоритма обрабатывается только одна вершина. С другой стороны, поиск в глубину не находит кратчайших путей, зато он применим в ситуациях, когда граф неизвестен целиком, а исследуется каким-то автоматизированным устройством.

Если же граф ориентированный, то поиск в глубину строит дерево путей из начальной вершины во все доступные из нее.

Обход в глубину можно представить себе следующим образом. Пусть исследователь находится в некотором лабиринте (графе) и он хочет обойти весь лабиринт (посетить все доступные вершины в графе). Исследователь находится в некоторой вершине и видит ребра, исходящие из этой вершины. Очевидная последовательность действий исследователя такая:

1. Пойти в какую-нибудь смежную вершину.
2. Обойти все, что доступно из этой вершины.
3. Вернуться в начальную вершину.
4. Повторить алгоритм для всех остальных вершин, смежных из начальной.

Видим, что алгоритм является рекурсивным — для обхода всего графа нужно переместиться в соседнюю вершину, после чего повторить для этой вершины алгоритм обхода. Но возникает проблема зацикливания — если из вершины A можно перейти в вершину B, то из вершины B можно перейти в вершину A и рекурсия будет бесконечной. Для борьбы с рекурсией нужно применить очень простую идею — исследователь не должен идти в ту вершину, в которой он уже был раньше, то есть которая не представляет для него интерес (считаем, что интерес для исследователя представляют только вершины, в которых он не был ранее). Итак, уточненный алгоритм может выглядеть следующим образом:

1. Пойти в какую-нибудь смежную вершину, не посещенную ранее.
2. Запустить из этой вершины алгоритм обхода в глубину
3. Вернуться в начальную вершину.
4. Повторить пункты 1-3 для всех не посещенных ранее смежных вершин.

Для реализации алгоритма понадобится отмечать, в каких вершинах был исследователь, а в каких — нет. Пометку будем делать в списке visited, где visited[i] == True для посещенных вершин, и visited[i] == false для непосещенных. Пометка «о посещении вершиных» ставится при заходе в эту вершину.

Поскольку целью обхода в глубину зачастую является построение дерева обхода в глубину, то сразу же будем хранить предшественника для каждой вершины.

Алгоритм обхода в глубину оформим в виде рекурсивной функции dfs, где start — номер вершины, из которой запускается обход.

**Пример реализации на языке C++**

**void dfs(int start, vector<bool> & visited, vector <int> & prev,**

**const vector <vector <int> > g)**

**{**

**visited[start] = true;**

**for (auto u : g[start])**

**if (!visited[u]) {**

**prev[u] = start;**

**dfs(u, visited, prev, g);**

**}**

**}**

**int main()**

**{**

**…**

**vector <bool> visited(n + 1);**

**vector <int> prev(n + 1, -1);**

**dfs(start, visited, prev, g);**

**Пример реализации на языке Python**

**visited = [False] \* (n + 1)**

**prev = [None] \* (n + 1)**

**def dfs(start, visited, prev, g):**

**visited[start] = True**

**for u in g[start]:**

**if not visited[u]:**

**prev[u] = start**

**dfs(u)**

**dfs(start, visited, prev, g)**

В этом алгоритме n – число вершин в графе, вершины нумеруются числами от 1 до n, а v[u] хранит множество вершин смежных с u. Для запуска алгоритма, например, для вершины с номером start необходимо вызвать dfs. После этого вызова все вершины, доступные из start, будут отмечены в списке visited, а при помощи списка prev можно построить пути из вершины start до всех доступных вершин. Если не требуется строить дерево обхода в глубину, то можно убрать заполнение списка start, в этом случае алгоритм dfs становится чрезвычайно простым.

**Выделение компонент связности**

Алгоритм обхода в глубину позволяет решать множество различных задач. Например, реализуем при помощи алгоритма обхода в глубину подсчет числа компонент связности в неориентированном графе.

Для этого будем обходить все вершины графа и проверять, была ли очередная вершина посещена ранее. Если не была – то это означает, что найдена новая компонента связности, для выделения всей компоненты связности необходимо запустить DFS от этой вершины.

**Пример реализации на языке C++**

**void dfs(int start, vector<bool> & visited, const vector <vector <int> > g)**

**{**

**visited[start] = true;**

**for (auto u : g[start])**

**if (!visited[u])**

**dfs(u, visited, g);**

**}**

**int main()**

**{**

**…**

**vector <bool> visited(n + 1);**

**int ncomp = 0;**

**for (i = 1; i <= n; ++i)**

**if (!visited[i]) {**

**++ncomp;**

**dfs(start, visited, g);**

**}**

**Пример реализации на языке Python**

**Visited = [False] \* (n + 1)**

**def DFS(start):**

**Visited[start] = True**

**for v in V[start]:**

**if not Visited[v]:**

**DFS(v)**

**ncomp = 0**

**for i in range(1, n + 1):**

**if not Visited(i):**

**ncomp += 1**

**DFS(i)**

**Проверка графа на двудольность**

Граф называется двудольным, если его вершины можно разбить на два множества так, что концы каждого ребра принадлежат разным множествам. Иными словами, можно покрасить вершины графа в два цвета так, что концы каждого ребра покрашены в разный цвет.

Модифицируем алгоритм DFS так, что он будет проверять граф на двудольность и строить покраску графа в два цвета (если он двудольный). Для этого заменим список Visited на список Color, в котором будем хранить значение 0 для непосещенных вершин, а для посещенных вершин будем гранить значение 1 или 2 – ее цвет.

Алгоритм DFS для каждого ребра будет проверять цвет конечной вершины этого ребра. Если вершина не была посещена, то она красится в цвет, неравный цвету текущей вершины. Если же вершина была посещена, то ребро либо пропускается, если его концы – разноцветные, а если его концы одного цвета, то делается пометка, что граф не является двудольным (переменной IsBipartite присваивается значение False, по ее значению можно судить о том, является ли граф двудольный).

**Пример реализации на языке C++**

**bool is\_bipartite = true;**

**void dfs(int start, vector<int> & color, const vector <vector <int> > g)**

**{**

**for (auto u : g[start])**

**if (color[u] == 0) {**

**color[u] = 3 - color[start]**

**dfs(u, color, g);**

**}**

**else if (color[u] == color[start])**

**is\_bipartite = false;**

**}**

**int main()**

**{**

**…**

**vector <int> color(n + 1);**

**for (i = 1; i <= n; ++i)**

**if (color[i] == 0) {**

**color[i] = 1;**

**dfs(i, color, g);**

**}**

**Пример реализации на языке Python**

**Color = [0] \* (n + 1)**

**IsBipartite = True**

**def DFS(start):**

**for u in V[start]:**

**if Color[u] == 0:**

**Color[u] = 3 - Color[start]**

**DFS(u)**

**else if Color[u] == Color[start]:**

**IsBipartite = False**

**for i in range(1, n + 1):**

**if Color[i] == 0:**

**Color[i] = 1**

**DFS(i)**

Основная программа проходит по всем ребрам графа и при обнаружении ранее не обнаруженной вершины красит ее в цвет 1 и запускает DFS из этой вершины.

**Поиск цикла в ориентированном графе**

Цикл в ориентированном графе можно обнаружить по наличию ребра, ведущего из текущей вершины в вершину, которая в настоящий момент находится в стадии обработки, то есть алгоритм DFS зашел в такую вершину, но еще не вышел из нее. В таком алгоритме DFS будем красить вершины в три цвета. Цветом 0 («белый») будем обозначать еще непосещенные вершины. Цветом 1 («серый») будем обозначать вершины в процессе обработки, а цветом 2 («черный») будем обозначать уже обработанные вершины. Вершина красится в цвет 1 при заходе в эту вершину и в цвет 2 – при выходе. Цикл в графе существует, если алгоритм DFS обнаруживает ребро, конец которого покрашен в цвет 1.

**Пример реализации на языке C++**

**bool cycle\_found = false;**

**void dfs(int start, vector<int> & color, const vector <vector <int> > g)**

**{**

**color[start] = 1;**

**for (auto u : g[start])**

**if (color[u] == 0)**

**dfs(u, color, g);**

**else if (color[start] == 1)**

**cycle\_found = true;**

**color[start] = 2;**

**}**

**int main()**

**{**

**…**

**vector <int> color(n + 1);**

**for (i = 1; i <= n; ++i)**

**if (color[i] == 0)**

**dfs(start, color, g);**

**Пример реализации на языке Python**

**Color = [0] \* (n + 1)**

**CycleFound = False**

**def DFS(start):**

**Color[start] = 1**

**for u in V[start]:**

**if Color[u] == 0:**

**DFS(u)**

**elif Color[start] == 1:**

**CycleFound = True**

**Color[start] = 2**

**for i in range(1, n + 1):**

**if Color[i] == 0:**

**DFS(i)**

**Топологическая сортировка**

Наконец, еще одно важное применение поиска в глубину – топологическая сортировка. Пусть дан ориентированный граф не содержащий циклов. Тогда вершины этого графа можно упорядочить так, что все ребра будут идти от вершин с меньшим номером к вершинам с большим номером.

Для топологической сортировки графа достаточно запустить алгоритм DFS, при выходе из вершины добавляя вершину в конец списка с ответом. После окончания алгоритма список с ответом развернуть в противоположном порядке.

**Пример реализации на языке C++**

**void dfs(int start, vector <bool> & visited, vector <bool> & ans, const vector <vector <int> > g)**

**{**

**visited[start] = true;**

**for (auto u : g[start])**

**if (!visited[u])**

**dfs(u, visited, ans, g);**

**ans.push\_back(start);**

**}**

**int main()**

**{**

**…**

**vector <bool> visited(n + 1);**

**vector <int> ans;**

**for (i = 1; i <= n; ++i)**

**if (!visited[i])**

**dfs(start, visited, ans, g);**

**reverse(ans.begin(), ans.end());**

**Пример реализации на языке Python**

**Visited = [False] \* (n + 1)**

**Ans = []**

**def DFS(start):**

**Visited[start] = True**

**for u in V[start]:**

**if not Visited[u]:**

**DFS(u)**

**Ans.append(start)**

**for i in range(1, n + 1):**

**if not Visited(i):**

**DFS(i)**

**Ans = Ans[::-1]**

**Поиск мостов**

Мостом называется ребро, при удалении которого граф распадается на две компоненты связности.

Алгоритм поиска в глубину позволяет найти все мосты в связном графе за один DFS, то есть за сложность O(n).

Подвесим граф за какую-то вершину, запустим из этой вершины DFS. DFS построит дерево обхода графа, при этом будут найдены *обратные рёбра* - рёбра, которые идут из текущей вершины в вершину, которая находится в настоящий момент в стадии обработки. Каждой вершине u сопоставим значение h(u) — её глубина в дереве обхода.

Кроме этого, каждой вершине сопоставим значение функции f(u), где f(u) - это минимальное значение h(v) для всех вершин v, которые достижимы из вершины u в дереве обхода, а также достижимы при помощи прохода по одному обратному ребру из любого потомка u в дереве обхода.

Тогда ребро uv будет мостом, если f(v)>h(u).

Значения h(u) и f(u) можно считать одним DFS.

**Пример реализации на языке C++:**

**void dfs(int u, int parent, int curr\_h, vector <vector<int> > & g, vector <bool> & visited, vector<int> & h, vector<int> & f)**

**{**

**h[u] = curr\_h++;**

**f[u] = h[u];**

**visited[u] = true;**

**for (auto v : g[u])**

**{**

**if (v == parent)**

**continue;**

**if (!visited[v])**

**{**

**dfs(v, u, curr\_h, g, visited, h, f);**

**f[u] = min(f[u], f[v]);**

**if (f[v] > h[u])**

**{ // Найден мост u-v**

**}**

**}**

**else**

**f[u] = min(f[u], h[v]);**

**}**

**}**

Параметры, передаваемые в функцию:

**u**

 - текущая вершина

**parent**

 - родитель, чтобы не проходить по ребру в обратном направлении (эта реализация не работает на графе с кратными ребрами).

**curr\_h**

 - текущая глубина

**g**

 - списки смежности графа

**h**

 - массив значений глубины для вершин

**f**

 - массив значения целевой функции для вершин