Introduction to Graphs: Definitions, Traversal, Depth-First Search

54

21

Hello @all,

As I have mentioned on several posts here on forums, I am only now starting to learn graphs in a more educated, methodical and serious way, so, after I have studied DFS for few days and due to me not finding anything useful online, I decided I would write this tutorial so people can benefit from it and hopefully having an easier time in learning about graphs :) **[note: I still don't know much things about graphs, but, I believe that writing about a topic we learnt, it's the best way to strengthen our knowledge on it, so, if any flaws are found, please feel free to correct them :)]**

- **Foreword and my personal view about Graphs and Graph problems in competitive programming**

Graphs are everywhere! From social networks, to route planning and map analysis, passing by party planning, electric circuits analysis and even Galatik football tournaments (:D) there are actually few domains of Science and everyday life where Graphs can't be seen as useful. As such, it can be quite an intimidating topic, especially because all the mathematical theory that lies behind them, seems to be a bit far from this reality.

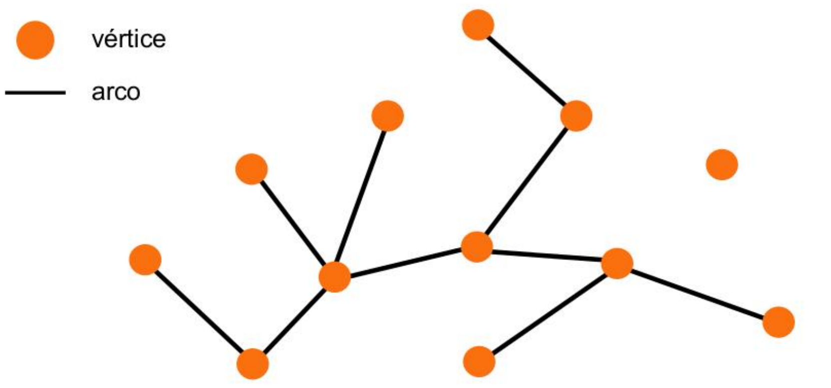
I will try to define a Graph, state some terminology and end with the first of two graph traversal methods, which, as we will see ahead, are both essential in "reading" the graph (in the sense of gathering information about its structure) and, with some additions, allows us to solve some simple problems.

- **Definition of Graph and schemes of internal representation to allow easy computational analysis**

Formally, we define a graph as being composed by two sets, **V** and **E**, respectively denoting the set of **vertices** and the set of **edges**. We say that vertices are connected by edges, so an edge connects two vertices together. If we generically denote a graph as **G**, we usually denote it by its two sets, such that the following notation is very common:

G(V, E) - is the graph G composed by **V** vertices and **E** edges.

Below is a simple example of what a graph looks like (the description is in portuguese, but, it should be easy to understand):



So we can immediately see that the small orange circles represent **vertices** and the black lines represent **edges**.

Please note that while it might seem a bit formal and confusing at the beginning, when we talk about graphs, this is the terminology we are going to use, as it is the only terminology we have which allows us to understand all graph algorithms and to actually talk about graphs, so, for someone who finds this a bit confusing, please twist your mind a bit :)

So, now we know how to define a graph and how it is formally represented. This is very important for the remaining of this post, so, make sure you understand this well :)

However, for sure that graphs aren't only about defining a set of vertices and edges, are they?

Well, it turns out that things can get a little bit more complicated and there are some **special types of graphs** besides the generic one, which was the one we just looked at. So, let us see what more kinds of graphs we can have.

- **Special types of graphs**

As it was mentioned on the foreword, graphs are used to model a lot of different everyday and science problems, so, it was necessary to devise more particular types of graphs to suit such vast problem-set. Let's see some definitions about these "special" graphs and examples of application domains:

Type 1: Directed Graph

Question tags:

dfs ×647

tutorial ×630

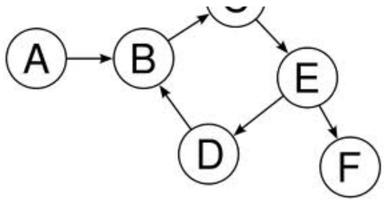
graphs ×334

question asked: **17 Jul '13, 22:24**

question was seen: **57,938 times**

last updated: **19 Apr '17, 19:17**

Related questions
GRTRIP - Editorial
2-EDGE CONNECTED GRAPH
WA in Galactik Football
code for DRHTS works correctly for 50
vertices, but gives WA at codechef
SPOJ question IITKWPCI wrong answer
DFS, BFS using Vectors
Help with INOI graph problem
COZID-Chef and Food Delivery
FAMTREE-editorial
GOSTONES-editorial



On a Directed Graph, the edges can have an associated direction.

This type of graph can be particularly useful if we are modelling for example some given system of roads and/or highways to study traffic, as we can have a defined way on a given road. Making the edges directed embeds this property on the graph.

Type 2: Directed Acyclic Graph

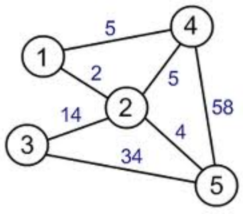
It is alike the graph above, except that on this particular type of graph we can't have "loops", i.e., **we can't** start a given path along the graph **starting on vertex A and ending of vertex A** again.

Type 3: Weighted Graph

In a weighted graph, the main characteristic is that the edges can have an associated "cost".

Mathematically speaking (and also in Operations Research) this "cost", need not to be money. It can model any given quantity we desire and it is always "attached" to the problem we are modelling: if we are analysing a map it can represent the cost of a given road due to a toll that's placed there, or distances between cities.

In Sports, it can represent the time taken by a professional paddling team to travel between two checkpoints, etc, etc... The possibilites are endless.



Besides these types there are also many other types of graphs (a tree, for instance, can be seen as a particular type of graph) that were devised for more advanced applications and that I really don't know much about yet, so these will be all the types of graphs I will talk about on this introductory post.

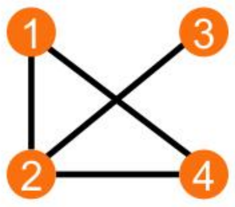
- **Representing a graph so we can manipulate it in a computer problem**

It turns out that one of the hardest things about graphs for me (besides wrapping your head around the enormous amount of applications they have) is that literally **everything** on them is highly dependent on the problem they are modelling, **even the way we decide to represent them is.**

However, there are not that many ways to represent a graph internally in computer memory and even from that small universe we are going to focus solely on **two** representation schemes.

Representation via **adjacency lists** and representation via **adjacency matrix.**

For a better understanding, let us consider the graph below:



We can see now, thanks to our formal definition given earlier on this text, that this graph is composed from two sets, the **set of the vertices** and the **set of the edges**.

We have 4 vertices and 4 edges.

However, only the information concerning the **number** of vertices and edges is not sufficient for us to describe a given graph, because, as you might have figured out by now, the edges could be placed differently (like in a square, for example.) and we wouldn't be able to know it.

This is why there are representations of graphs which allows us to "visualize" the graph in all of its structure, from which the **adjacency matrix** will be the one we will study in the first place.

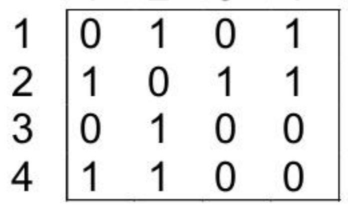
- **Adjacency Matrix**

The adjacency matrix (let's call it A) is very simple to understand and, as the name itself says, it's a representation that is based on a matrix of dimensions V x V, where it's elements are as follows:

A(i,j) -> 1, if there exists an edge between vertex i and vertex j;

A(i,j) -> 0, otherwise;

So, for the above graph, its adjacency matrix representation is as follows:

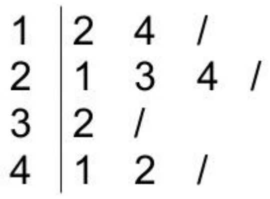


$$\begin{array}{c|cccc} 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 & 0 \end{array}$$

- **Adjacency List**

The representation as an adjacency list is usually more convenient to represent a graph internally as it allows for an easier implementation of graph traversal methods, and it is based, also, as the name states, in having a "list of lists" for each vertex, that states the vertices to which a given vertex is connected.

As a picture is worth a thousand words, here is the adjacency list representation of the above graph:



1	2	4	/	
2	1	3	4	/
3	2	/		
4	1	2	/	

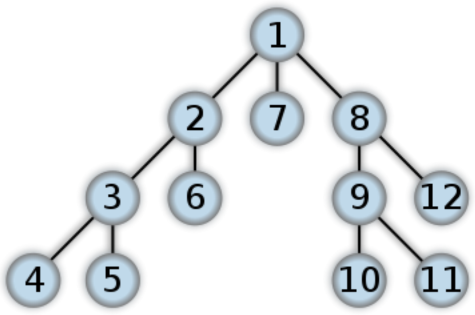
here, the symbol / is used to denote end of list :)

Now that we have looked into two of the most common ways of representing a graph, we are ready to study the first of two traversal algorithms, **Depth-First Search.**

- **Understanding Depth-First Search**

Depth-First search, often known as DFS (it stands for the beginning letters of each word) is a graph traversal algorithm where the vertices of the graph are explored in depth first order, with the search algorithm backtracking whenever it has reached the end of the search space, visiting the remaining vertices as it tracks back.

This idea is illustrated on the figure below where the order by which vertices are visited is described as a number (note that this is a tree, but, also, as discussed earlier, a tree is a particular kind of graph, and it has the advantage of being easy to visualize).



As the vertices are first explored to a certain order and then we must backtrack to explore remaining vertices, the DFS algorithm can be naturally implemented in a recursive fashion. Also please note that we assume here that the visited vertices are visited only once, so we can have, in pseudo-code:

```
dfs(v):
    visited(v)=true
    for i = graph.begin() to graph.end()
        if(visited(i)=false)
            dfs(i)
```

It's now important to mention two things:

The first one is that in Graph Theory, DFS is usually presented in an even more formal fashion, mentioning colors and discovery times and end times for each vertex. Well, from what I understood, in programming contests the above routine is enough and the purpose of having array visited() is precisely to "replace" the color attributes that are related to discovery and end times, which I chose to ommit to keep this post in the ambit of programming competitions.

Finally, to end this explanation, what is more important to mention is that DFS usually is implemented with something more besides only that code.

Usually we "add" to DFS something "useful" that serves the purpose of the problem we are solving and that might or might not be related to two of most well known applications of DFS, which are:

- **DFS applications**

Connected Components;

Topological Sorting;

- **Wrapping it all together: Solving FIRESC problem using what we know**

FIRESC problem is the perfect problem for us to put our skills to test :)

Simply, we can model this is terms of graphs as follows:

Let us define a graph whose vertices are people. An undirected edge connects two people who are friends.

On this graph, two vertices will be of the same color if they share an edge. This represents that the two people should have to go in the same fire escape route. We wish to maximize the number of colors used.

All the vertices in a connected component in this graph will have to be in the same color.

So as you see, we need to count the size of all connected components to obtain part of our answer, and below I leave my commented implementation of this problem, largely based on the work of @anton_lunyov and on the post found on this site (yes, I had to use google translate :p).

```
#include <iostream>
#include <stdio.h>
#include <vector>
#include <iterator>
using namespace std;

vector<bool> visited; //this vector will mark visited components
vector<vector<int> > graph; //this will store the graph represented internally as an
adjacency list
//this is because the adjacency list representation is the most suited to use DFS procedure
on a given graph

int sz_connect_comp = 0; //this will store the size of current connected component
(problem-specific feature)

void dfs(int v)
{
    sz_connect_comp++; //"useful feature" performed on this DFS, this can vary from problem
to problem
    visited[v] = true;

    for(vector<int>::iterator it = graph[v].begin(); it != graph[v].end(); it++)
    {
        if(! visited[*it]) //note that *it represents the adjacent vertex itself
        {
            dfs(*it);
        }
    }
}

int main()
{
    int t;
    cin >> t;
    while(t--)
    {
        int n,m;
        cin >> n >> m;
        graph = vector<vector<int> > (n); //initialization of the graph
        for(int i = 0; i < m; i++)
        {
            int u,v;
            cin >> u >> v;
            u--;
            v--;
            //these are added this way due to the friendship relation being mutual
            graph[u].push_back(v);
            graph[v].push_back(u);
        }
        int res = 0; // the number of fire escape routes
        int ways = 1; // the number of ways to choose drill captains
        visited = vector<bool> (n, 0); // initially mark all vertices as unvisited
        for(int u = 0; u < n; u++)
        {
            //if the vertex was visited we skip it.
            if(visited[u]==true)
                continue;
            // if vertex was not visited it starts a new component
            res++; // so we increase res
            sz_connect_comp = 0; // init sz_connect_comp
            dfs(u); // and calculate it through the dfs, marking visited vertices
            // we multiply ways by sz_connect_comp modulo 1000000007
            ways = (long long)sz_connect_comp * ways % 1000000007;
        }
        printf("%d %d\n", res, ways);
    return 0;
}
```

I hope you have enjoyed this short introduction to graphs and I really hope this can help anyone willing to study graphs to come across a rigorous, yet simple and understandable text :)

Best regards,

Bruno

edited 18 Jul '13, 04:53

graphs tutorial dfs

asked 17 Jul '13, 22:24
2★ kuruma
[17.6k]●72●143●209
accept rate: 8%

1 This is nice. A correction. "Vertices" instead of "Vertexes".

3★ rachitsachdeva (17 Jul '13, 22:35)

Thanks, I will fix this!

2★ kuruma (17 Jul '13, 22:37)

1 @admin , why does it say "asked n days ago by kuruma" shouldn't it rather say "posted n days ago by kuruma" ? btw @Kuruma, really nice and helpful post :)

4★ v_akshay (19 Jul '13, 22:46)

Thank you @v_akshay, I'm glad you liked this :) About the asked vs posted part, as this was a "question", it says asked, but, I guess that is just how the forums work :)

2★ kuruma (22 Jul '13, 23:22)

@Kuruma its really helpful thanks!!

2★ buro (17 Feb '14, 17:02)

16 Answers:

oldest answers newest answers popular answers

1 2 next »

4 @kuruma Real Nice One! Keep it up bro! Your editorials are always superb. :)

I have a little something to add. This is about the array `visited` used in DFS. In CLRS, the authors use another array called `color`. Yes, the same is possible with the array `visited`. Let me add some light on the array `color`. Also, a use of discovery time and finishing time is discussed here.

Please note that this answer is entirely based on the DFS algorithm as given in CLRS text book (Introduction to Algorithms).

`color[v] <-` WHITE, initially for all v, before the start of DFS
`color[v] <-` GRAY, when v is discovered, and the exploring of its children is not complete
`color[v] <-` BLACK, when all the children of v have been completely explored

Now, `visited[v] = false` means `color[v] = WHITE`, and `visited[v] = true` means `color[v]` is either GRAY or BLACK. This is enough for normal applications. However, sometimes, the information whether a vertex is partially or completely explored is useful. In those cases, using this `color` (or `flag` or `status`, as anyone would like to call -- we need three distinct values) information is helpful.

For example, the problem of **labelling / classifying edges of a directed graph as tree-edge / forward-edge / back-edge / cross-edge**. (Go here to know a little bit more on what these labels mean.) Edges used in DFS (i.e., edges from a node v to a child node colored WHITE) are labelled tree-edge. Edges from v to a partially visited node will be back-edge (see the use here) as the 'partial visitedness' of a node u has to mean u is an ancestor of v in the DFS tree. Now, Edges from v to a node u which is colored BLACK can be either a forward-edge (if u is a descendant of v) or a cross-edge (if u is not a descendant of v). The information on whether u is a descendant of v can be found using the discovery times of v and u (u will be discovered after v, if u is a descendant of v).

Following is a pseudocode, solving edge-labelling problem.

```
DFS-Visit(v)
{
    color[v] := GRAY
    time := time + 1
    d[v] := time
    for each vertex u adjacent to v do
        if color[u] == WHITE then
            π[u] := v
            Classify (v, u) as a tree edge
            DFS-Visit(u)
        else if color[u] == GRAY then
            Classify (v, u) as a back edge
        else if color[u] == BLACK then
            if d[v] < d[u] then
                Classify (v, u) as a forward edge
            else
                Classify (v, u) as a cross edge
    color[v] := BLACK
    time := time + 1
```

The solution given here talks about using the discovery time and finishing time to identify and classify edges.

This is one example, where just having the array `visited` will not be sufficient. I do not know whether these arrays in discussion will actually be needed (or even come in handy) for competetive programming or the like. However, it is good to know that these arrays are not there, just for the purpose of completeness. And like @kuruma has said, for most competitive programs, an implementation with just the array `visited` is usually enough!

link

answered 18 Jul '13, 09:42
2★ tijoforyou
[4.2k]●5●23●64
accept rate: 15%

1 Hey @tijoforyou, yes, I was perfectly aware that such arrays are not only there for the sake of completeness and might really be needed at some point (just as you showed :)). I opted for not including them to keep the text as introductory as possible and also because it was what it was mentioned on that russian website link I provided :)

2★ kuruma (18 Jul '13, 14:43)

2

@kuruma hey this is my blog on coding.Just started it. I'll soon be posting about graphs.Do take a look at it. http://cod3rutopia.blogspot.in/

link

answered 18 Jul '13, 12:27
3★ jaskaran_1
[525]●23●35●50
accept rate: 0%

1

@Kuruma ,again one nice,lucid and clear explanation great work :)

link

answered 18 Jul '13, 15:43
1★ faiz
[85]●3
accept rate: 0%

It's nice to be appreciated @faiz :D All I need now is to train hard so I can start applying all these ideas to contest problems :)

2★ kuruma (18 Jul '13, 16:06)

1

Thank you for this post. Keep up the good work. Yes, the best way to learn something is to explain it to others. Thank you for your lovely post and i hope you have a good day you beautiful human being!

link

answered 19 Jul '13, 02:03
3★ tacoder
[517]●4●6●15
accept rate: 18%

Thanks for your positive feedback sir :)

2★ kuruma (19 Jul '13, 14:15)

1

Nice work kuruma....keep it up.It was written in a neat and crisp way.This tutorial was required by many newbies. One can expect people implementing this tutorial's code from the next contests :)

link

answered 23 Jul '13, 13:29
4★ re_hash
[849]●3●8●15
accept rate: 3%

Thank you very much for the positive feedback @re_hash! I hope I can implement all these ideas in next contests myself as well :D

2★ kuruma (23 Jul '13, 19:27)

1

to **ADMIN** staff, pls pin algo tutorials like these on static algo tutorials page. topcoder has that

link

answered 21 Jan '14, 05:34
1★ garakchy
[1.1k]●16●30●48
accept rate: 1%

1 Do we have a "static algo tutorials page" here on Discuss? The closest thing seems to be the tutorials and references posted at Codechef for Schools page, but, if we could pin a "sub-forum" grouping all tutorials or even better all editorials on its own "sub-forums", that would be really useful

2★ kuruma (21 Jan '14, 06:22)

0

@bruno Really nice post. if you want to learn some more things about graphs i would recommend DSA course in NPTEL by Prof. Naveen Garg. here is the link.

link

answered 17 Jul '13, 23:16
3★ kcahdog
[10.0k]●28●54●129
accept rate: 14%

Thank you very much @kcahdog I will look into it asap :D

Also, do you know any more problems which are as straightforward as FIRESC? I wanted to train more :D

2★ kuruma (18 Jul '13, 02:26)

3★ jaskaran_1 (18 Jul '13, 02:47)

Thank you very much for this link @jaskaran_1 and I am glad you liked my post :) It is amazing to see my work recognized and I really feel like I am learning new things now!! :)

2★ kuruma (18 Jul '13, 03:00)

1 @kuruma Even i am new to graphs and GALACTIK is the only graphs problem i have solved so far. Will keep you updated if i find any interesting problems

3★ kcahdog (18 Jul '13, 16:26)

Thank you very much @kcahdog :) I really tried very hard to solve GALACTIK during contest, but, I couldn't :(

2★ kuruma (18 Jul '13, 19:36)

showing 5 of 6 show all

0

@Kuruma Great post !!! Even I learnt about graphs only after the FIRESC problem :)

Also, just wanted to share one of my learnings from the GALACTIK problem (http://www.codechef.com/JULY13/problems/GALACTIK) , which I solved using graphs. I was always confused regarding when to use BFS and when to use DFS. By default I always used DFS. But looks like solving GALACTIK with DFS will give TLE! After many wrong attempts I finally got AC when I shifted to BFS :)

link

answered 18 Jul '13, 10:27
3★ vpyati
[154]●5
accept rate: 0%

I got AC with DFS!!

2★ tijoforyou (18 Jul '13, 10:39)

And because of the ease of programming, me too, usually (or by default :P) use DFS!!!

2★ tijoforyou (18 Jul '13, 10:40)

Hello @vpyati, yes, I also really, really want to solve GALACTIK problem :D I am eagerly waiting for editorials and although I have seen some AC solutions I want to understand everything at its fullest extent :)

2★ kuruma (18 Jul '13, 14:44)

GALACTICK could be solved using Disjoint Sets data structure instead of DFS and BFS. This would have sped up the code by many times :)

2★ r93sachdeva55 (20 Jul '13, 01:18)

I solved the problem first by Disjoint Sets and then by DFS. Learned a lot about DFS from GALACTIK that helped me in FEB14 in solving DRGHTS.

4★ r3gz3n (17 Feb '14, 17:13)

0

Awesome post !! Easy to understand and really good for a beginner . .

I tried my hand on implementing some of the graph algorithms in python and I've uploaded it on github link , if someone wants to go through them , its available . It might be a bit messy though , I've tried my best to make it look good and readable . Hope it helps !!

link

answered 18 Sep '13, 22:19

2★ sayan_paul
[41]●1●3
accept rate: 0%

0

https://theoryofprogramming.wordpress.com/

Have a look at this blog it contains nice stuff on graphs and trees..

link

answered 03 Feb '15, 22:19
2★ v_nishanth
[1]●1
accept rate: 0%

1 2 next »

[hide preview] community wiki:

Preview