Algorithms. Lecture Notes 10

Graph Traversals

From now on, $G = (V, E)$ denotes a graph with $n = |V|$ nodes and $m = |E|$ edges.

Graph traversals are techniques to visit all nodes in a graph $G = (V, E)$ in a fast and systematic way. They provide a basis for several efficient graph algorithms. We consider directed graphs $G = (V, E)$ and denote a directed edge from u to v by (u, v). Note that undirected graphs may be considered as special directed graph where both directed edges (u, v) and (v, u) exist, for every pair of adjacent nodes u and v.

Perhaps the simplest traversal strategy is **Breadth-First-Search (BFS)**. (Don't forget the "d" in "breadth" ...) It starts in one node s which is put in a queue and marked. In every step, BFS takes the next node u from queue and visits *all* unmarked nodes v such that $(u, v) \in E$. Every such v is put in the queue and marked. BFS stops as soon as the queue is empty.

We study some properties of BFS. First of all, BFS partitions the set of nodes into layers L_i, $i \geq 0$, inductively defined as follows. L_0 contains only the start node s, and L_{i+1} contains all nodes v such that: an edge $(u, v) \in E$ for some $u \in L_i$ exists, and v is not already in an earlier layer. It is easy to see that BFS, implemented with a queue, processes the nodes exactly layer by layer. More importantly, the layers provide some useful structure: Edges (u, v), with $u \in L_i$, $v \in L_j$ go at most to the next layer, that is, $j \leq i + 1$. (But j can be arbitrarily smaller than i.) It follows that L_i contains exactly the nodes with (directed) distance i from s, in other words, the nodes reachable from s on a directed path with i (but not fewer than i) edges. Hence BFS as such yields an algorithm for the Shortest Paths problem, provided that all edges have unit length.

Take some time to think about the proof of the last assertion. One must verify two things for every node $t \in L_i$: (a) There *exists* some directed path of length i from s to t. (b) There is *no shorter* directed path from s to t.

BFS also gives rise to a directed tree which contains all marked nodes and a certain subset of the edges from E: Whenever a node v is found for the first time, via the edge (u, v), we insert this edge in the tree. In fact, this yields a tree rooted at s, since every node except s has exactly one predecessor. We refer to it as the **BFS tree**. Note that all edges in the BFS tree go from a layer to the next layer (but in general not all edges to the next layer are in the BFS tree).

To analyze the time for BFS, note that every edge is considered only once. The crucial step is to determine the nodes v with $(u, v) \in E$, for a given u. The time for this operation depends on the way the graph is represented. When adjacency lists are used, we simply need to traverse the list for u, thus we spend only constant time on every edge. We conclude that BFS needs $O(m)$ time. (This simple argument in the time analysis is very common, for a number of graph algorithms.) If an adjacency matrix is used, we need $O(n^2)$ time, which is in general worse. Namely, for the node u considered in each step we have to check all matrix entries in u's row, even in the case that almost all of them are 0.

The other standard graph traversal strategy is **Depth-First-Search (DFS)**. It starts in a node s and follows a directed path of yet unexplored nodes, as long as possible. When it reaches a dead end (where all successor nodes of the current node are already explored), it goes one step back on the path, looks for another unexplored successor node, and so on.

The most compact formulation is a recursive procedure DFS(u) with start node u as input parameter (the main program is to call DFS(s)): As long as unmarked nodes v with $(u, v) \in E$ exist, choose one such v, mark v, and call DFS(v). – Since each recursive call is done only after termination of the previous call, this gives the desired depth-first behaviour. DFS can also be written as an iterative program, but then the stack must be implemented explicitly.

DFS exhibits some similarities to BFS. The time for DFS is $O(m)$ when adjacency lists are used to collect all successors of a node. A **DFS tree** can be defined as follows: Edge (u, v) belongs to the DFS tree if DFS(u) calls DFS(v). Such edges (u, v) are said to be **tree edges**. Indeed, they form a tree, since v becomes the input parameter of a recursive call only once, and then v gets marked.

But there are also major differences to BFS. They concern the positions of edges from E which are *not* in the DFS tree:

In undirected graphs, such edges can only go from a node to an ancestor

node in the DFS tree. This follows easily from the rules of DFS. We call them **back edges**. There exist no **cross edges**, that is, edges joining nodes from different paths of the DFS tree. (Why not? To understand the reason, assume for contradiction that a cross edge exists, and see how DFS would have produced it ...)

In directed graphs this issue is somewhat more complicated. Directed edges which are not in the DFS tree can be divided into three types: **forward edges** going from a node to some descendant node, **back edges** going from a node to some ancestor node, and **cross edges** going from a node to another node on an "earlier" directed path of the DFS tree. – These structural properties are useful in several graph algorithms based on DFS.

Problem: Undirected Graph Connectivity

An undirected graph is **connected** if there exists a path between any two nodes. The **connected components** are the maximal connected subgraphs.

Given: an undirected graph $G = (V, E)$.

Goal: Decide whether G is connected. If not, compute the connected components.

Problem: Strong Connectivity in Directed Graphs

A directed graph is **strongly connected** if there exists a *directed* path from every node to every node. The **strongly connected components** are the maximal strongly connected subgraphs.

Given: a directed graph $G = (V, E)$.

Goal: Decide whether G is strongly connected. If not, compute the strongly connected components.

Motivations:

If the graph models states of a system and possible transitions between them, strong connectivity means it is always possible to recover every state, i.e., the system has no irreversible moves. The street map of a city with one-way streets should be strongly connected as well, or the traffic planners made a mistake.

Some Applications of BFS and DFS: Connectivity

Testing connectivity of a graph can easily be misjudged as a trivial problem, but without some systematic strategy we would aimlessly walk around in the labyrinth of the graph and use much more time than necessary. Graph traversal solves several connectivity problems efficiently, as we will see now.

BFS starting in node s in a graph G reaches exactly those nodes being reachable from s on directed paths. The same is true for DFS. In particular, if G is undirected, then the traversal explores exactly the connected component of G which contains s. This gives an $O(m)$ algorithm to test whether an undirected graph G is connected: Run BFS or DFS, with an arbitrary start node. G is connected if and only if all nodes are reached. We can also determine the connected components of G in $O(m+n)$ time: If the search has aborted without finding all nodes, restart the search in a yet unmarked node, and so on.

Connectivity is more intricate in directed graphs. Still, strong connectivity can be checked in $O(m)$ time. But first we give a naive algorithm: Run BFS (or DFS) twice for evry start node s: once on the given directed graph and once on the reversed graph where all edges (u, v) are replaced with (v, u). Thus we find all nodes t being reachable from s, and we find all nodes t from which s is reachable. The graph is strongly connected if and only if the result is positive for all s and t. The algorithm needs $O(nm)$ time for n times BFS.

But a little thinking and problem analysis yields a much better algorithm: Run BFS twice (as above), but with only one arbitrary but fixed start node s. The graph is strongly connected if and only if both BFS runs reach all nodes. This is correct because, in a strongly connected graph, one can get from every node to every node also via the fixed node s. (It is recommended to write down, for yourself, the complete proof in logical steps, after this hint.) This algorithm needs only $O(m)$ time.

If the graph is *not* strongly connected, then his simple algorithm determines the strongly connected component which contains s: It is the set of nodes reached in both the given graph and the reversed graph. One can obviously extend this algorithm, in order to partition the graph into its strongly connected components. Hovewer, we may need $O(nm)$ time again: In the worst case, the graph may have many small strongly connected components, but we may need $O(m)$ time to determine each one in this way. Actually, it is possible to compute even all strongly connected components in $O(m)$ time by some sophisticated use of DFS, but we have to skip this theme.

Problem: Graph Coloring

Given a set of k colors, a **k-coloring** of a graph assigns a color to each vertex, so that adjacent vertices get different colors. A graph is **k-colorable** if it admits a k-coloring. The 2-colorable graphs are exactly the bipartite graphs.

Given: an undirected graph $G = (V, E)$ and an integer k.

Goal: Construct some k-coloring of G, or report that G is not k-colorable.

Motivations:

Imagine that a person who is not exactly an expert in botany gets a set of plants, and he is told that they belong to two different species. He does not always see whether two plants belong to the same species or not, however, *some* pairs of plants are obviously different. Is it possible for him to divide the set correctly and efficiently? This can be translated into the 2-coloring problem: Every species (class, category, etc.) is represented by a "color". The plants (or whatever objects) are nodes of a graph $G = (V, E)$, where any two nodes that are *known* to belong to different classes are joined by an edge. The 2-colorable graphs are also called bipartite graphs.

Various problems dealing with packing, frequency assignment, job assignment, scheduling, partitioning, etc., can be considered as Graph Coloring, where the graph models pairwise conflicts. Note that Interval Partitioning problem is a special case of Graph Coloring, with the goal to minimize the number of colors: Intervals are nodes, two nodes are adjacent if the corresponding intervals overlap, and the "colors" are copies of the resource.

One Graph and Two Colors

We conclude with another simple application of BFS: The 2-coloring problem is solvable in $O(m)$ time. The key observation is: If a node gets one color, then all adjacent nodes *must* get the other color, and so on. This is, a bit implicitly, already the correctness proof of the following algorithm. BFS merely serves as a framework to organize the enforced coloring efficiently.

Now the algorithm in detail: We compute the BFS tree and the layers. Then, all nodes in the layers L_i, i even, get one color, and all nodes in the layers L_i, i odd, get the other color. Since each node in L_{i+1} is joined to some node in L_i via some edge of the BFS tree, essentially only one valid 2-coloring can exist in each connected component. The only degree of freedom

is that we can swap the two colors. (Alternatively one may also use DFS, but then the details are, of course, different.)

The idea cannot be extended to $k > 2$ colors, because the color of a node does no longer determine the color of all neigbored nodes. We have the choice between different colors, and it is not clear how we could safely avoid later coloring conflicts.

Actually, k-coloring is $\mathcal{NP}$-complete for every $k \geq 3$. This can be shown by a reduction from 3SAT being somewhat similar to the reduction from 3SAT to Independent Set.

Problem: Minimum Spanning Tree

A **spanning tree (MST)** in an undirected graph $G = (V, E)$ is a tree that contains all nodes of V (it "spans" the graph) and a subset of edges taken from E.

Given: a connected undirected graph $G = (V, E)$ where every edge has some positive cost.

Goal: Construct a spanning tree T in G with minimum total cost (sum of costs of all edges in T).

Motivations:

This is a basic network design problem. It appears when certain sites have to be connected in the cheapest way by streets, cables, virtual links, or whatever. Edge costs may represent lengths, costs of material, or other costs of the links. Note that a minimum-cost connected spanning subgraph of G is always a tree, since if there were a cycle, we could remove some edge without destroying connectivity.

Further Instructions and Examples for Self-Study

- Make sure that you understand why BFS solves the Shortest Path problem (with unit edge lengths), whereas DFS has nothing to do with shortest paths. (Actually, the latter is a frequent misconception.)
- In Lecture Notes 1 we stressed that "An algorithm is a precise and unambiguous description of the calculations" etc., but now, the results of DFS and BFS can depend on the exact ordering of nodes presented. (See also the little example below.) Isn't this a contradiction – what do you think?
- As for the time bounds of linear-time graph algorithms, we sometimes write $O(n + m)$, and sometimes $O(m)$. Do you see the subtle difference, and how this "discrepancy" is justified? Note especially that a connected graph with n nodes has $m \geq n - 1$ edges.

BFS/DFS Example

Some minimalistic example illustrates the differences of the two main traversal techniques. Consider an undirected graph with node set $V = \{a, b, c, d\}$ and edge set $E = \{ab, ac, bc, bd\}$. (Here we use a common "lazy" notation for undirected edges; for instance, ab means the edge between a and b.) Recall that an undirected edge equals a pair of opposite directed edges.

BFS with start node a yields $L_0 = \{a\}$, simply by definition, $L_0 = \{b, c\}$, as these are all neighbors of a, and $L_2 = \{d\}$, as this node has a neighbor in L_1, namely b. The layers are uniquely determined. In this example, the BFS tree is also uniquely determined (containing all edges except bc). In larger examples, however, the BFS tree may depend on the ordering of nodes within the layers, because a node can in general have several neighbors in the previous layer.

What happens in DFS with start node a? This heavily depends on the order in which we consider the adjacent nodes. If we continue with c, the we will follow the path $a - c - b - d$. Then this path is also the DFS tree, and the edge ab is a back edge. If we, alternatively, visit b immediately after a, we get a DFS tree with two overlapping paths: $a - b - c$ and $a - b - d$. The edge ac becomes a back edge on the first path. Note that there are (of course) no cross edges.