Algorithms. Lecture Notes 10

Graph Traversals

Graph traversals are techniques to visit all nodes in a graph in a fast and systematic way. The 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).

Perhaps the simplest traversal strategy is **Breadth-First-Search (BFS)**. (Don't forget the "d" in "breadth" ...) It starts in one node s which we put in a queue and mark as discovered. 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. 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, $j \leq i + 1$. It follows that L_i contains exactly the nodes with (directed) distance i from s, in order words, the nodes reachable from s on a directed path of i (but not fewer than i) edges. Hence BFS as such includes an algorithm for the Shortest Paths problem, provided that all edges have unit length.

BFS also gives rise to a directed tree which contains all discovered nodes and a certain subset of the edges from E: Whenever a node v is discovered the first time, via the edge (u, v), we insert this edge in the tree. This yields a tree rooted at s, since every node except s has exactly one predecessor. We refer to it as the **BFS tree**. All edges in the BFS tree go from a layer to the next layer.

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. 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 nodes adjacent to the current one 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)): Mark u as explored, and call DFS(v) for all unmarked v with $(u, v) \in E$. 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 but 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**. Indeeed, they form a tree, since v becomes the input parameter of a recursive call only once, and then v gets marked. Differences to BFS 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**. Furthermore, there exist no **cross edges**, that is, edges joining nodes from different paths of the DFS tree. 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 a descendant node, **back edges** going from a node to an 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 might be misjudged as a very simple 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:

BFS starting in node s in a graph G reaches exactly those nodes reachable from s on directed paths. The same is true for DFS. If the search reaches some u, then all v with $(u, v) \in E$ will be reached, too. From this fact, the statements follow by induction.

In particular, if G is undirected, 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 either 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 is easy to check in $O(m)$ time: Run BFS (or DFS) with an arbitrary 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). Both searches must reach all nodes. This condition is sufficient, since one can get from every node to every node via s. If the graph is not strongly connected, this 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: 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. It is possible to compute all strongly connected components in $O(m)$ time by some sophisticated use of DFS, but we have to skip this theme.

A nice use of the DFS properties in undirected graphs is the recognition of **articulation points**. An articulation point is a node whose deletion disconnects the graph. We can state: A node v is an articulation point if and only if the DFS tree of DFS(v) has more than one child. This follow from the absence of cross edges in undirected graphs.

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 a 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. BFS merely serves as a framework to organize the coloring efficiently. Now in detail: We compute the

BFS tree and the layers. Then, all nodes in layers L_i, i even, get one color, and all nodes in 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 an edge of the BFS tree, essentially only one valid 2-coloring can exist in each connected component. (We can only swap the two colors.)

This algorithm does not work for $k > 2$ 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 Vertex Cover.

Problem: Detecting Directed Cycles

A **directed cycle** in a directed graph is a cycle that can be traversed respecting the orientation of the edges: $v_1, v_2, v_3, \ldots, v_n, v_1$, where every (v_i, v_{i+1}) and (v_n, v_1) is a directed edge. A *directed acyclic graph* (DAG) is a directed graph without directed cycles. DAGs should not be confused with trees which are connected graphs without *any* cycles (which are in general undirected).

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

Goal: Find a directed cycle in G, or report that G is a DAG.

Motivations:

Directed cycles are undesirable in plans of tasks where directed edges (u, v) model pairwise precedence relations (task u must be done before task v). These tasks can be calculations in a program or logic circuit, jobs in a project, steps in a manufacturing process, etc. In such models, directed cycles indicate errors in the design.

Some systems in Artificial Intelligence, so-called partial order planners, automatically create plans to achieve some goal, given a formal description of the goal and of available actions. Part of the construction algorithms are tests for directed cycles. If such cycles are detected in a plan, some actions must be removed, and the corresponding partial goals must be realized in a different way, avoiding new cycles.

Problem: Topological Order

A **topological order** of a directed graph $G = (V, E)$ is an order of all nodes of V so that all directed edges go to the right. In other words, for every directed edge (u, v), node u appears earlier than node v in the order.

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

Goal: Construct a topological order of G, or report that G does not admit a topological order.

Motivations:
The nodes are jobs with pairwise dependency constraints, as above. Any topological order is a possible order of executing these jobs without violating the precedence constraints.

Directed Acyclic Graphs (DAGs) and Topological Ordering

We continue with fast algorithms that detect directed cycles or construct a topological order (if existing) in a directed graph G.

Looking at the specifications of these problems, perhaps it is not hard to guess that G is a DAG if and only if G allows a topological order. The "if" direction is obvious: If all edges go in the same direction, one can never close a directed cycle. The "only if" direction is more interesting, and the **proof is constructive** in the sense that it also shows how to obtain a topological order, provided that G is a DAG. The proof is done by induction on the number of nodes.

Observe that the first node v in a topological ordering must not have incoming edges (u, v). Conversely, any node v without incoming edges can be put at the first position of a topological order. Here comes the inductive argument: Remove v and all incident edges from G. The remaining graph is still a DAG. (No new directed cycles can be created by removing parts of the graph.) Hence, G without v has a topological order, and by setting v in front of this topological order, we get one for the entire G.

The resulting algorithm has a very simple structure: Put some node v without incoming edges at the next position of the topological order, remove v and all incident edges, and so on.

This algorithm is obviously correct if it goes through. But how do we know that there always exists such a node v to continue? Assume by way of contradiction that every node has an incoming edge. Then we can traverse a path of such edges in opposite direction, but since G is finite, we must sometimes meet a node again. But G has no directed cycle by assumption.

A remarkable detail is that we can always take an *arbitrary* node v without incoming edges. We can never miss a topological order by an unlucky choice of v in some step. The proof ensures this. In a sense, we can consider this algorithm a greedy algorithm, even though there is nothing to optimize.