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. 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 gives actually 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 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 neighbored 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 neighbors 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). This gives actually a tree, since v becomes 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 some graph algorithms based on DFS.

Some Applications of BFS and DFS

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 nontrivial use of DFS, but we have to skip this point.

Instead we will discuss the use of DFS for another connectivity problem in *undirected* graphs: finding all articulation points.

Finding all Articulation Points

We run DFS in an arbitrary start node s. The root s of the DFS tree is an articulation point if and only if s has more than one child in the DFS tree.

This criterion follows from the absence of cross edges. We could run DFS n times, once from every start node, which costs $O(nm)$ time.

Amazingly, it is possible to solve the problem in $O(m)$ time, using only one DFS tree and a variant of dynamic programming. We skip this more sophisticated algorithm.

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: 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 the edges 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 one edge without destroying connectivity.

Problem: Clustering with Maximum Spacing

A **clustering** of a set of (data) points is simply a partitioning into disjoint subsets of points, called **clusters**. Some distance function is defined between the points. The distance of two point sets A and B is the minimum distance of two points $a \in A$ and $b \in B$. The **spacing** of a clustering is the minimum distance of two clusters (or equivalently, the minimum distance of any two points from different clusters).

Given: a set of n points in some geometric space, and an integer $k < n$. The pairwise distances of points are known, or they can be easily computed from their coordinates.

Goal: Construct a clustering with k clusters and maximum spacing.

Motivations:
Clustering in general has many applications in data reduction, pattern recognition, classification, data mining, and related fields. Coordinates of points are often numerical features of objects. Every cluster shall consist of "similar" objects, whereas objects in different clusters shall be "dissimilar". However, we have to make these intuitive notions precise. There exist myriads of meaningful quality measures for clusterings, and each one gives rise to an algorithmic problem: to find a clustering that optimizes this quality measure.

Many clustering problems can be formulated as graph problems, where the data objects are nodes. For instance, Graph Coloring can be seen as a clustering problem: The desired number k of clusters is given, and every cluster must fulfill some "internal" criterion, namely, not to contain any pair of dissimilar nodes. Spacing is an "external" quality measure. It demands that any two clusters be far away from each other, while nothing is explicitly said about the inner structure of clusters.

Problem: Longest Paths

Given: an undirected or directed graph $G = (V, E)$, the lengths $l(u, v)$ of all edges $(u, v) \in E$, and a start ("source") node $s \in V$.

Goal: For all nodes $x \in V$, compute a (directed) path from s to x with maximum length, but such that no node appears repeatedly on the path.

Motivations:

Finding longest paths is not a silly problem. In particular, it makes much sense on DAGs. For example, if the DAG is the plan of a project with parallelizable tasks modelled by the edges, and the edge lengths are execution times, then the longest path in the graph gives the necessary execution time (makespan) for the whole project. It is sometimes called the critical path.