Lecture 16: BFS and DFS

Graph is one of the most important concepts in algorithms. We will spend several weeks on algorithms for graphs. In this lecture, we study two basic algorithms for graph search: breadth-first search (BFS) and depth-first search (DFS). Many of you have known these two algorithms before. But I suggest you to refresh your memory and also think a little deeper about these two algorithms.

First, BFS can find the distances of nodes to a source node. Here, distance just means the smallest number of edges connecting the two nodes. See the textbook for the BFS algorithm. In class, I presented a simpler version of the algorithm: no coloring is used since we just use the distance to check whether a node is visited before. The key question on BFS is why the distances set by the BFS algorithm are correct. The correctness is based on three properties that holds for all \( d = 0, 1, 2, \ldots \) at some point of time (note it is some time, not always): (1) each node with distance (from source \( s \)) \( \leq d \) is set properly, (2) all other nodes are not discovered (i.e. distance is set to \( \infty \)), (3) the queue contains precisely the nodes with distance \( d \). We can see why these hold by using induction. At \( d = 0 \), this is almost trivial: we only have \( s \) in stack (whose distance is 0, and \( s \) is the only such node). Then assume at some point of time, these three properties hold. Then, we can show that there exists some moment where the three properties hold for \( d + 1 \). Let us focus on one part: the distance set by the first node \( u \) of the queue is correct. Note in this case, a newly discovered node \( v \) (distance currently set to \( \infty \)) will be set to distance \( d + 1 \). Why is this action correct: can \( \text{dist}(v) \geq d + 2 \) or \( \text{dist}(u) \leq d' \)? First, \( \text{dist}(v) \) must be at most \( d + 1 \) because \( v \) is reachable in one hop from a distance \( d \) node. Second, \( \text{dist}(v) \) can not be \( d \) or smaller because by the induction assumption, all such nodes (with distance \( d \) or smaller) have already been set (and thus \( v \)'s distance can not be \( \infty \). We are not quite done yet, but I hope you have seen where I am leading to. You can finish the details yourself.

Then we come to DFS, which is like exploring the maze. Just like exploring the maze, we must try to avoid going in cycles. The tools for this is (a) chalk for marking visited places, and (b) a roll of strings to backtrack. These are easily done by having a flag for each graph node (which is marked when it is visited), and by recursion. In class, I described a slightly simpler algorithm than the one in textbook. There, we just use a flag \( \text{visited}(u) \) to indicate whether \( u \) is visited or not (instead of colors in the textbook). The rest of algorithm is the same as the textbook (read the section if you did not write down all the steps in class). A key utility of DFS is timestamp, which tells when a node is initially discovered and finished. This time interval \([u.d, u.f]\) is often quite useful for some problems. We will continue on this next week.

The timestamp can be very useful. The time interval (discovery time, finish time) has a useful property: time interval for any two nodes traversed by DFS must be either disjoint or one contain another. See the textbook for proof and more explanations.

DFS also helps to classify the graph edges into four types. (a) Tree edge: along which a node is first visited. Why these edges form trees? Any node at most one incoming edge. A node can be discovered just once. (b) Forward edge: lead to an already visited descendant. (c) Back edge: lead to an ancestor (and form a cycle). (d) Cross edge: lead to a node has been explored before but not descendants or ancestors. Forward edge must go from an ancestor (in the order of node discovery) to a descendant. For example, we have nodes \( a, b, c \) and \( d \), and DFS starts from \( a \) and visit \( b \) from \( a \), \( c \) from \( b \), and then return to \( c \) and visit \( d \). In this case, if there is an edge from \( d \) to \( c \), it is not a forward edge because \( d \) is not the ancestor of \( c \) in DFS tree (that is, we did not visit \( c \) during the traversal of \( d \)). Thus, edge from \( d \) to \( c \) is a cross edge. This means that just like tree edge, the time interval of forward edge is just like tree edge: starting node's time interval contains that of ending node. We also discussed how to determine the types of edges based on the timestamp (time interval). That is, how can we determine the edge type by just looking at the time intervals? Refer to your notes or textbook if you forgot.
Lecture 17: DFS and Applications

Now the first application of topological sort. First, a directed graph G is acyclic iff G has no back edges in DFS. We proved it in class (review it and it is in textbook). If G is a DAG, we can order its nodes along a line (called linearization), so that edges go one direction. A useful property is that each DAG allows a topological order. Here an observation helps: for each edge \((u, v)\) in a DAG, we have \(u.f > v.f\). Why? Suppose \(u.f < v.f\). Then, can \((u, v)\) be cross edge? No: cross-edge means u’s finish time is later. Can be forward/tree? No: \(u.f > v.f\) in those cases. So must be back edge. But we know NO BACK edges, since G has no cycles. So what should be the starting? The one with largest \(u.f\) (called source). Review your notes to see how we did this and how to analyze the running time.

Now we move to the topic of finding the strongly connected components. Strongly connected component: subset of nodes in the graph, such that each pair of nodes in the subset can go to each other in both directions. Suppose we shrink each component into a node, this created a condensed graph (called component graph). This graph must be DAG, otherwise we can merge the components along the cycle form a larger strongly connected component.

We continue our last lecture on finding strongly connected components. Suppose you start from some node the sink component (no outgoing edges to any other components in component graph) and perform DFS, you traverse inside this component only. But this only works for sink component. Second, the graph with highest finish time must be in the source component (i.e. no other component connects to it). Why? Consider two components \((C_1, C_2)\) (i.e. there is an edge from \(C_1\) to \(C_2\) in component graph). Note we can not go from a node in \(C_2\) and reach nodes in \(C_1\) (component graph \(G_c\) has no cycles). In class, we verified that either \(C_1\) or \(C_2\) is visited first, we all get contradictions to the assumption that a node of \(C_1\) has the largest finish time. You should verify this yourself and this helps you to gain more understanding of the directed graph.

But this is not what we want: we want a node in the sink not the source component. The trick is to create the complement of G (called G’) by reversing the edges of G. G’ has same components as G, but the component graph is also reversed. Now this gives the algorithm: (1) first do DFS; (2) Construct G’ (the complement graph); (3) do DFS on G’ by visiting the nodes according to decreasing order of finish time of the first DFS; (4) Output nodes found inside one DFS-visit (i.e. the DFS trees) as strongly connected components. The key to make this work is that in the complemented graph, the second DFS from the node with highest (original) DFS finish time will not visit other components; this also holds when we visit the other nodes in the decreasingly order because the components that are reachable from a node has already been visited by the second DFS and we will not revisit them. Read the textbook again to ensure you understand what I am writing here.