

Continuation of Lecture 4: BFS and Preorder/Postorder

Claim 1. If $(u, v) \in E$ then $\text{postorder}(u) < \text{postorder}(v) \iff (u, v)$ is a back edge.

Proof. If $(u, v) \in E$, then before u is popped off of the stack, we *could* have pushed v onto the stack via (u, v) . (\Leftarrow) But if (u, v) is a backedge and not a tree edge, it must already be on the stack underneath u , and thus will pop after u . (\Rightarrow) Or, if $\text{postorder}(u) < \text{postorder}(v)$, (u, v) cannot be a tree edge or we'd push and pop v after pushing u , and thus we'd have $\text{postorder}(v) < \text{postorder}(u)$, a contradiction. Then (u, v) must be a backedge. \square

Claim 2. $G = (V, E)$ has a cycle \iff the DFS tree of G yields a back edge.

Proof. If (u, v) is a back edge, then (u, v) together with the path from v to u in the DFS forest form a cycle.

Conversely, for any cycle in $G = (V, E)$, consider the vertex assigned the smallest postorder number. Then the edge leaving this vertex in the cycle must be a back edge by Claim 1, since it goes from a lower postorder number to a higher postorder number. \square

Application: Topological Sort

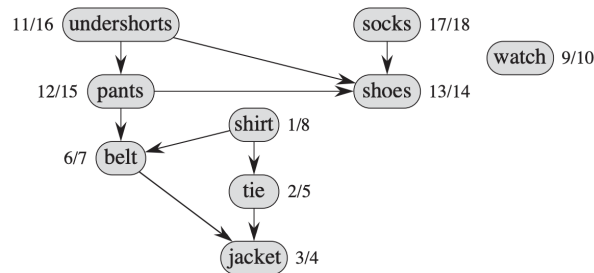


Figure 1: Top sort example graph from CLRS.

Theorem 1. G has a topological order $\iff G$ is a DAG.

Proof. (\Rightarrow) Topological ordering means only edges $(i, j) \in E$ where $i < j$. Consider the smallest i in the cycle. There exists an edge (j, i) in the cycle for $j > i$. Contradiction.

(\Leftarrow) If there's a DAG, this implies that there exists a node with no incoming edges. Otherwise, one could backtrack, and after n steps, would find a cycle. \square

Topological Sort Algorithm:

- Naive algorithm: Recursively remove a node with no incoming edges. $T(n) = O(n^2)$.
- Or, run DFS to assign postorder times, and then sort the DFS forest by decreasing postorder. $T(n) = O(n + m)$.

Theorem 2. *If the tasks are scheduled by decreasing postorder number, then all precedence constraints are satisfied.*

Proof. If G is acyclic then the DFS tree of G produces no back edges by Claim 2. Therefore by Claim 1, $(u, v) \in G$ implies $\text{postorder}(u) > \text{postorder}(v)$. So, if we process the tasks in decreasing order by postorder number, when task v is processed, all tasks with precedence constraints into v (and therefore higher postorder numbers) must already have been processed. \square

Introduction to Greedy: Shortest Path

Say we want to find the shortest path from home to any other point in the city. How might we do that? One way would be to use BFS.

But, what if there's congestion on the roads? That is, what if the graph is weighted?

Definition 1. Let w_e (or w_{uv}) denote the *weight* of edge e (or (u, v)).

We can think of this as the length of the edge, or the time (or cost) to traverse it.

Question: Given a graph G , how can we find the shortest (least-weight) path from s to any other vertex v ?

Algorithm 1 Dijkstra's Algorithm(G, w)

Input: Graph $G = (V, E)$ and weights w .

Let S be the set of explored nodes

for each $u \in S$, store a distance $d(u)$

Initially $S = \{s\}$ and $d(s) = 0$

while $S \neq V$ **do**

 let $d'(v) = \min_{(u,v) \in E, u \in S} d(u) + w_{uv}$

 select $v \in \text{argmin}_{v \notin S} d'(v)$

 add v to S and set $d(v) = d'(v)$

end while

Let P_v denote the shortest path to v from s —that is, $P_v = P_u \cup (u, v)$ when $v \in \text{argmin}_{v \notin S} d'(v)$ and $(u, v) \in \text{argmin}_{(u,v) \in E, u \in S} d(u) + w_{u,v}$.

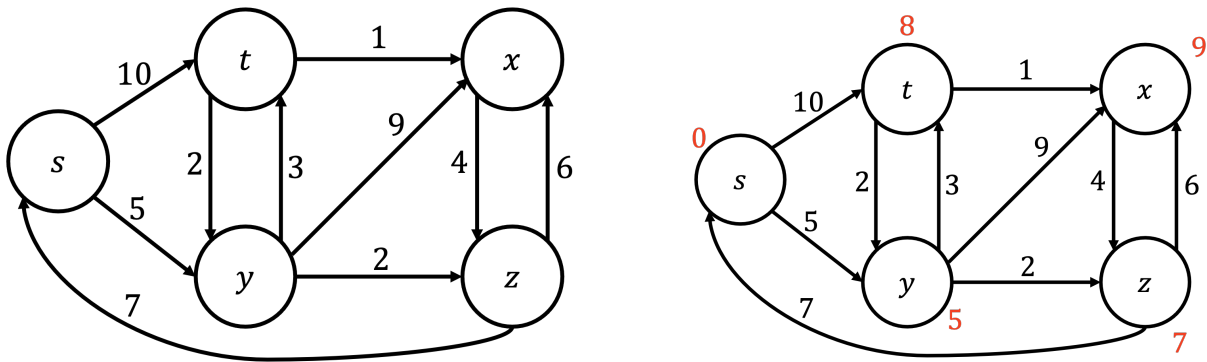


Figure 2: A weighted graph G (left) and the shortest path from s calculated for each vertex (right).

Greedy Stays Ahead

There are four main steps for a greedy stays ahead proof.

Step 1: Define your solutions. Describe the form your greedy solution takes, and what form some other solution takes (possibly the optimal solution). For example, let A be the solution constructed by the greedy algorithm, and let O be a (possibly optimal) solution.

Step 2: Find a measure. Find a *measure* by which greedy stays ahead of the other solution you chose to compare with. Let a_1, \dots, a_k be the first k measures of the greedy algorithm, and let o_1, \dots, o_m be the first m measures of the other solution ($m = k$ sometimes).

Step 3: Prove greedy stays ahead. Show that the partial solutions constructed by greedy are always just as good as the initial segments of your other solution, based on the measure you selected.

- For all indices $r \leq \min(k, m)$, prove (often by induction) that $a_r \geq o_r$ or that $a_r \leq o_r$, whichever the case may be. Don't forget to use your algorithm to help you argue the inductive step.

Step 4: Prove optimality. Prove that since greedy stays ahead of the other solution with respect to the measure you selected, then it is optimal.

Proof of Dijkstra

Then we prove the following via a “greedy stays ahead”-style induction.

Lemma 1. Consider the set S at any point in the algorithm's execution. For each $u \in S$, P_u is a shortest $s - u$ path.

(Proof of Correctness: When the algorithm terminates, S contains all nodes, thus by Lemma 1, we'll have found shortest paths from s to all nodes.)

Proof. By induction on $|S|$.

Base case ($|S| = 1$): $|S| = 1$, so $S = \{s\}$ and $d(s) = 0$.

Inductive Hypothesis: Suppose the claim holds for some $k \geq 1$.

Inductive Step ($|S| = k + 1$): The algorithm grows to $|S| = k + 1$ by adding some node v . Let (u, v) be the final edge on P_v . By the IH, P_u is the shortest $s - u$ path for all $u \in S$.

Consider any other $s - v$ path P . It must leave S somewhere—call (x, y) the edge that first crosses. But P cannot be shorter than P_v , as it contains P_x and (x, y) , and Dijkstra could have just added y to S with this P_y if it were shortest, but instead it chose $P_u + (u, v)$ as shorter. \square

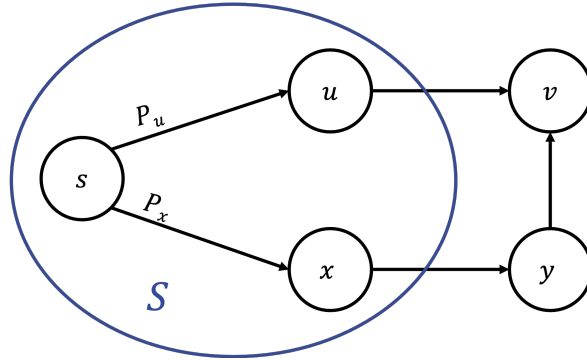


Figure 3: Illustration of the argument in the inductive step.

Dijkstra is really continuous/waterfilling BFS.