CS 228: Introduction to Data Structures
Lecture 39

[redacted]

Shortest Paths Trees

Throughout this lecture, G denotes a weighted graph — directed or undirected—, s denotes the source node in G, and for each edge (u,v), w(u,v) is the weight (length) of e. For each node v in G, dist(v) denotes the length of a shortest path from s to v in G. Note that there may be multiple shortest paths from s to v, but they all have the same total length dist(v). Also, since edge lengths are positive, dist(s) = 0.

A ***shortest-path tree*** is a tree T made up nodes and edges of G and rooted at s, such that the path from s to any other vertex v in T is a shortest path from s to v in G. For instance, a BFS tree is a shortest-path tree for the case where all edges have the same length.

To see that a shortest-path tree always exists, suppose that we know dist(v) for every node v in G reachable from s. Let P be any shortest paths from s to v. Let pred(v) be the last node in P before v. Let P' be the subpath of P that goes from s to pred(v). Then, P' must be a shortest path from s to pred(v) — if it were not, P would not be a

shortest path to v, violating our assumption. This means that

$$\text{dist}(v) = \text{dist}(\text{pred}(v)) + w(\text{pred}(v),v).$$

Since each node other than s has exactly one predecessor, the set of edges (pred(v),v), $v \neq s$, form a tree rooted at s. By construction, if the tree contains a path from s to v, this path must be a shortest path from s to v in G.

The length of the shortest path from s to any other node v is unique; the path itself, however, may not be. That is, there may be multiple shortest paths (all of the same length) from s to v. Draw some examples to verify this!

Dijkstra's algorithm

Dijkstra's algorithm starts from the source node s, and in each iteration adds another node to the shortest-path tree. This node is the point closest to the root that is still outside the tree. The process resembles breadth-first search using a priority queue instead of a FIFO queue. Note, however, that it is *not* breadth-first search: we do not care about the number of edges on the tree path, only the sum of their weights.

Throughout its execution, Dijkstra's algorithm maintains the length d(v) of the best path from s to v found so far,

for each node v in G. At all times, d(v) ≥ dist(v) (where, as before, dist(v) is the length of the shortest path from s to v). Initially

- d(s) = 0, and
- d(v) = ∞ for every node v different from s.

Dijkstra's algorithm also keeps track of v's predecessor pred(v) in the current best path from s to v. Initially, pred(v) is null for every node v.

Dijkstra's algorithm maintains a set C of nodes, which we will call the ***cloud***. The algorithm repeats the following two steps until all nodes are in the cloud.

Step 1. Pick a node u such that d(u) is minimum among the nodes not in C and add u to C.

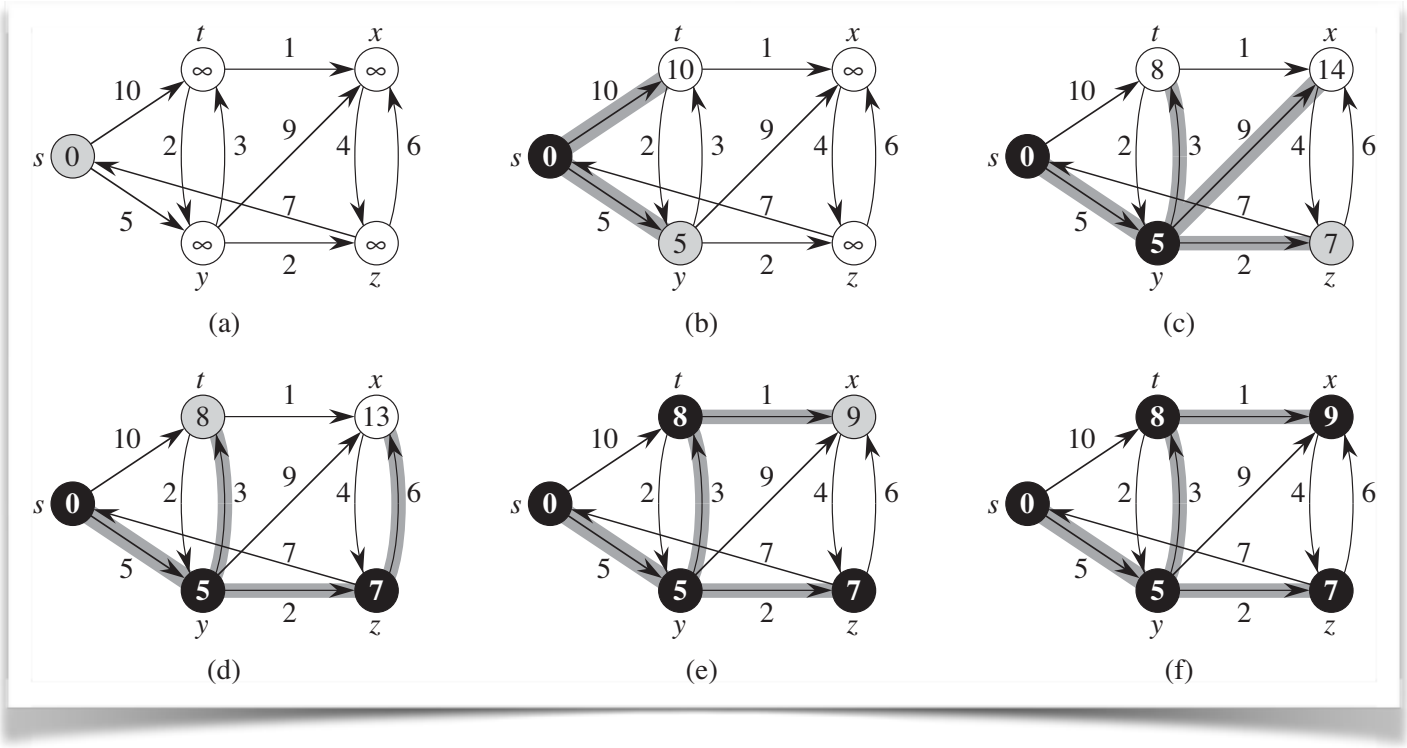
Step 2. For each neighbor v of u, ***relax*** edge (u,v). That is, if

$$d(v) > d(u) + w(u,v), \qquad (\dagger)$$

make pred(v) = u and d(v) = d(u) + w(u,v).

The intuitive justification of Step 2 is that encountering condition (†) implies that we have just found a path of length less than d(v) to v via u, so we should update d(v).

Example[1]. In the sample execution of Dijkstra's algorithm shown below black nodes are in the cloud. The node u with the minimum d-value at each step is shaded. Shaded edges indicate predecessor values.



Correctness

The correctness of Dijkstra's algorithm is a consequence of the following lemma. As before, dist(v) denotes the length of the shortest path from s to v.

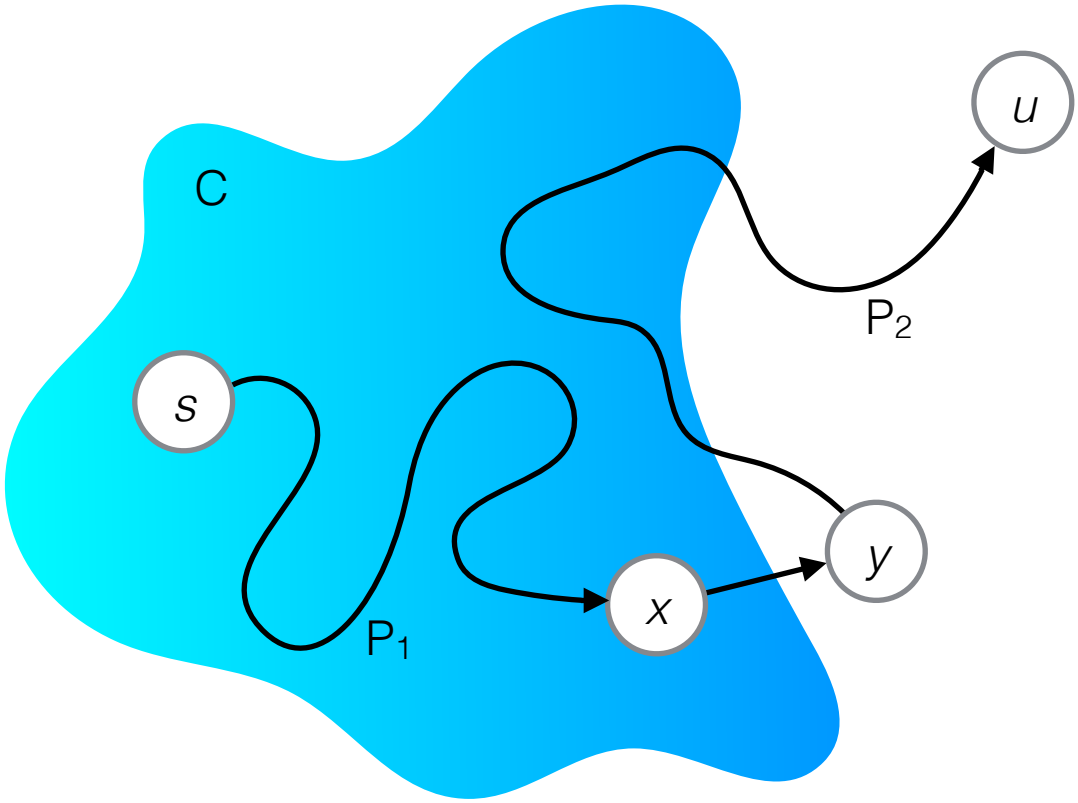
[1] This example is taken from CLRS. Slides for this example and another example, which uses an undirected graph, are posted on Blackboard.

Lemma. *At the start of each iteration of the algorithm,* d(v) = dist(v) *for every node* v *in C.*

Proof Sketch. The lemma is trivially true at the start, because C is empty. The first iteration adds s to C. Since dist(s) = d(s) = 0, the lemma remains true. (You can also check that it's true after the second iteration.) Let's assume that a number of iterations have taken place and that, so far, d(v) = dist(v) for every node v in C. Now, suppose that in the next iteration, the algorithm makes a mistake, and adds to C a node u such that dist(u) ≠ d(u). We'll sketch why this mistake cannot happen.

Suppose P is a shortest path from s to u. Let y be the first node in P that is not in C and let x be the node immediately preceding y in P. Then, as the figure on the next page illustrates, we can decompose P into three parts: (1) a path P_1 from s to x, (2) the edge (x,y), and (3) a path P_2 from y to u. The key observation is that we would then have d(y) < d(u) and so node u would not have been chosen by Dijkstra's algorithm, a contradiction.

We emphasize that the preceding argument is just a sketch of a proof. For a formal proof, consult CLRS.



Implementation

The pseudocode for Dijkstra's algorithm shown on the next page uses a priority queue Q, implemented as a binary heap, to store the nodes of G that are not in the cloud. Nodes in Q are prioritized by d-values, so the `removeMin()` operation executed at the beginning of each iteration always extracts the node in the cloud with the minimum d-value.

Relaxing edge (u,v) may necessitate decreasing d(v), so Q requires a `decreaseKey()` method (specifically, we need this in `(*)`). This method has two steps:

1. Reduce the value of d(v) in Q.

2. Use `percolateUp()` to readjust Q.

The time for this is dominated by `percolateUp()`, which is logarithmic in the number of nodes in the heap; i.e., it is O(log V). We leave the details as an exercise.

```
DIJKSTRA(G,s):
    for each node v in G except s
        d(v) = ∞
        pred(v) = null
    d(s) = 0
    let Q be a priority queue containing
       the nodes of G prioritized by d-value
    let C be an empty set
    while Q is not empty
        u = removeMin(Q)
        C = C u {u}
        foreach neighbor v of u
            // Relax (u, v)
            if d(v) > d(u) + w(u,v)
                d(v) = d(u) + w(u,v)        (*)
                pred(v) = u
    return pred
```

Time complexity. Assume that we use a binary heap to implement the priority queue Q. Initializing Q and the d- and pred-values takes O(V) time. The **while** loop does the

bulk of the work. There are V iterations. Removing the minimum element at the beginning of each iteration takes O(log V) time, for a total of O(V log V) time over all V iterations. During these iterations, the number of times we may have to update pred(v) and reduce the value of d(v) in Q is at most equal to the in-degree of v (assume that the graph is directed — a similar argument applies to undirected graphs). The total number of such updates over all iterations is at most the sum of the in-degrees; this sum equals E. Each update takes O(log V) time (dominated by the time to update Q), so the total time spent on these steps is O(E log V). This is the dominant term in the run time, so the time complexity of Dijkstra's algorithm is O(E log V).

Single-Source Shortest Paths in DAGs

In several important applications of the single-source shortest path problem, the graph G is a weighted directed acyclic graph (DAG). It turns out that for DAGs there is a faster and simpler alternative to Dijkstra's algorithm, which uses edge relaxation (like Dijkstra's algorithm) and topological sorting. This algorithm achieves a running time of O(V + E), without using any major data structures — in particular, no heaps. Before describing the DAG algorithm, we need to review a few things about shortest paths that apply to arbitrary graphs, not just acyclic ones.

Recall that by ***relaxing*** an edge (u,v) in a graph G we mean (1) checking whether there is a path to v going via u that is better than the current best path to v and (2) if the answer is "yes", updating d(v) and pred(v) accordingly.

```
RELAX(u,v):
|   if d(v) > d(u) + w(u,v)
|   |   d(v) = d(u) + w(u,v)
|   |   pred(v) = u
```

Suppose G is a weighted graph (not necessarily a DAG), where edge weights may be negative weights, but which has no negative cycles (i.e., cycles where the sum of the edge weights is negative). Now, suppose we put d(s) = 0 and d(v) = ∞ for all other nodes v. Then, the following property can be proved.

Path-relaxation property. If $p = v_0, v_1, \ldots, v_k$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0,v_1), (v_1,v_2), \ldots, (v_{k-1}, v_k)$, then $d(v_k) = dist(v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Now suppose G is a DAG, and that we have sorted G topologically. The shortest path algorithm shown next begins by initializing the d- and pred-values as usual. It then makes a single pass over the nodes in the

topologically order. As it process each node, it relaxes each edge that leaves the node.

```
DagShortestPaths(G,s):
|
|   topologically sort the nodes of G
|
|   // Initialization
|   foreach node v in G
|   |   d(v) = ∞
|   |   pred(v) = null
|   d(s) = 0
|
|   foreach node u in G in topological order
|   |   foreach neighbor v of u
|   |   |   // Relax (u, v)
|   |   |   if d(v) > d(u) + w(u,v)
|   |   |   |   d(v) = d(u) + w(u,v)
|   |   |   |   pred(v) = u
|
|   return pred
```

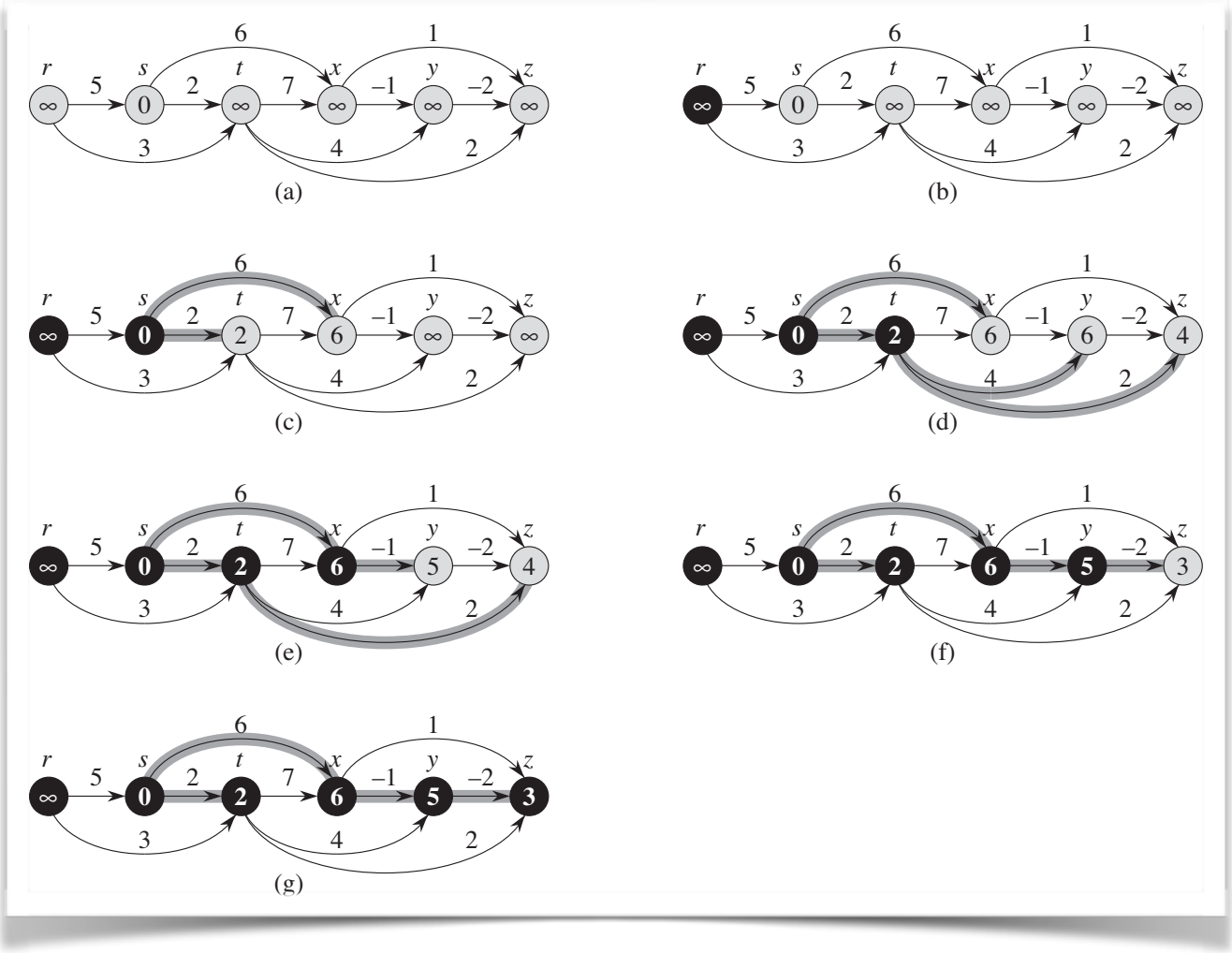
We claim that at termination, for every node v, d(v) is the length of the shortest path from s to v. To verify this, observe that if G contains a path from node u to node v, then u precedes in the topological sort. Thus, for every node v, the edges of a shortest path from s to v are relaxed in order, from first to last. By the path-relaxation

property, this means that, at termination, d(v) is indeed the length of a shortest path from s to v.

Observe that DAGSHORTESTPATHS is correct even for DAGs with negative-weight edges. The reason is that shortest paths are always well-defined in a DAG, since even if there are negative-weight edges, no negative-weight cycles can exist. Further, the path relaxation property holds as long as there are no negative cycles.

Example[2]**.** The figure on the next page shows a sample execution of DAGSHORTESTPATHS. The nodes are topologically sorted from left to right. The source node is s. The d values appear within the nodes, and shaded edges indicate the pred-values. The newly blackened node in each iteration was used as u in that iteration.

[2] This example is from CLRS and slides for it are posted on Blackboard.



Time complexity. As we have seen, topological sort takes O(V + E) time. The initialization of d- and pred-values takes O(V) time. The outer **foreach** loop makes one iteration per vertex. Altogether, the inner **foreach** loop relaxes each edge exactly once. Because each iteration of the inner **foreach** loop takes O(1) time, the total running time is O(V + E), which is linear in the size of an adjacency-list representation of the graph.

Caution: Shortest paths are *not* well-defined when negative cycles are present. In fact, if G has a negative cycle C, you can make the length of a path between two nodes in C arbitrarily small by going around C multiple times.