CIS 1210—Data Structures and Algorithms—Fall 2024

Dijkstra's—Tuesday, October 29 / Wednesday, October 30

Readings

- Lecture Notes Chapter 20: Dijkstra's Algorithm

Review: Dijkstra's Algorithm

In Recitation 7, we proved that BFS solves the "Single Source Shortest Path" problem for unweighted graphs. However, for weighted graphs, we need something more robust. Dijkstra's algorithm finds the shortest path between two given vertices in a weighted graph, assuming that the graph's edge weights are **non-negative**. The running time of the algorithm is $O(|E|\log|V| + |V|\log|V|)$ when the graph is implemented using adjacency lists. The pseudocode for the algorithm can be found here.

Runtime Analysis

The running time of Dijkstra's algorithm has two terms: $|E|\log|V|$ and $|V|\log|V|$. We first consider the $|V|\log|V|$ term: each EXTRACT-MIN operation takes $O(\log|V|)$ time, and this operation is called $|V|$ times because there are $|V|$ vertices.

The $|E|\log|V|$ term has to do with the **relaxation** step of Dijkstra's algorithm. Each edge examined may result in a relaxation of the neighboring node in the heap — a DECREASE-KEY operation that takes $O(\log|V|)$ time. The number of vertices examined in the for loop is bounded by the total degree of all vertices, as each vertex is added and popped exactly once from the min-heap. This value is $2|E|$ by the Handshaking Lemma, so in the worst-case we cannot have more than $2|E|$ decrease-key operations, for a total of $O(|E|\log|V|)$.

This analysis works for easily proving our runtime, but we can actually do a better analysis. **Each edge** (u, v) **can only cause one relaxation**, not 2 as the Handshaking Lemma suggests. This is because (u, v) is explored only when node u is popped from the min-heap. This means that when (u, v) is explored from node v, we know node u has already been removed, so its key cannot be decreased. Hence, the $O(|E|\log|V|)$ term comes from the $O(\log|V|)$ cost of a DECREASE-KEY operation, which is called at most $|E|$ times overall.

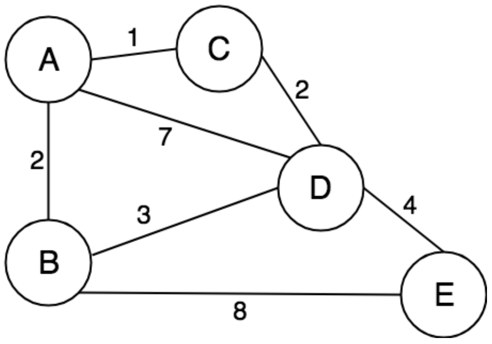
Greedy Algorithms

We call an algorithm "greedy" if it makes the **locally optimal** choice — the best available choice at that moment—in order to find a **globally optimal** solution. Greedy algorithms do not always yield optimal solutions, but for many problems they do. Note that Dijkstra's algorithm solves the "Single Source Shortest Path" problem by following this paradigm — it uses a priority queue structure that always yields the node with the shortest distance from the source node when polled. Consider the set S of vertices in Dijkstra's whose final shortest-path weights from the source have already been determined. At each step in the algorithm, since we ultimately want to find the shortest path from our source, we use our priority queue to make the locally optimal choice by adding the node with the current shortest distance from the source node to the set S.

Problems

Problem 1

Find the shortest paths from A:



Problem 2: True or False

1. The finishing times of all vertices in a SCC s must be greater than the finishing times of other SCCs reachable from s during the first DFS.
2. (Adapted from CLRS 22.5) Consider a "simpler" version of Kosaraju's algorithm, where we use the original (instead of the transposed) graph in the second DFS traversal but process vertices in order of increasing finishing times. Does this algorithm always return the correct result?

Problem 3: True or False

1. Provided there are no negative weight cycles, Dijkstra's algorithm will correctly return shortest path to all vertices.
2. Dijkstra's algorithm will not *terminate* when run on a graph with negative edge weights.
3. If we double the weights of all edges, then Dijkstra's algorithm produces the same shortest path.
4. If we square the weights of all edges, then Dijkstra's algorithm produces the same shortest path.

Problem 4

(From CLRS 24.3-6) We are given a directed graph $G = (V, E)$ where each edge $(u, v) \in E$ has an associated value $r(u, v) \in [0, 1]$ that represents the *reliability* of a communication channel from u to v. We thus interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Design an efficient algorithm to find the most reliable path (lowest failure probability) between a vertex s and any other vertex in the graph.