Solution Set

CIS 1210—Data Structures and Algorithms—Spring 2025

Dijkstra's—Tuesday, March 25 / Wednesday, March 26

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.

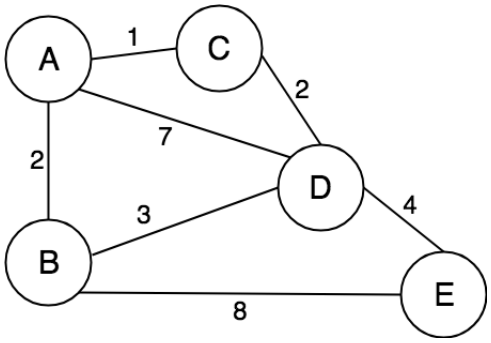
Greedy Algorithms

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:



Solution

Running Dijkstra's algorithm produces the following state:

Node	Distance from A
A	0
B	2
C	1
D	3
E	7

Node	Parent node
A	NULL
B	A
C	A
D	C
E	D

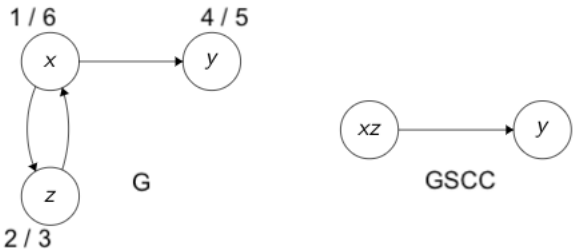
Problem 2: True or False

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?

Solution

1. **False.** Consider the counterexample below. On the left, a DFS traversal of G starting at x and processing the (x, z) edge first yields the starting/finishing times as annotated. On the right in G_{SCC}, we can see that x and z form one SCC and y forms the other with a (xz, y) edge. Observe that the finishing time of z is 3, which is not greater than than finishing time of an SCC reachable from it, since y has a finishing time of 5. However, it is true that at least one vertex must have a larger finishing time than those of the SCCs reachable from s due to the recursive nature of DFS.



2. **False.** This algorithm will not always be correct. Again, consider the counterexample above from part 1. If in the first DFS traversal of G, we start at x and process the (x, z) edge first, then in the second DFS traversal, processing the vertices in order of increasing finishing times in G will start at the source SCC, xz and also discover the sink SCC, y all at once, outputting both in the same DFS

tree — incorrectly declaring the graph as just one SCC.

At a high-level, the intuition behind why this simpler algorithm fails is because visiting vertices in G in order of increasing finishing times in the second DFS does not always correspond to visiting the overall SCCs in order of increasing finishing times — meaning that we **cannot** guarantee we are visiting SCCs of G in reverse topologically sorted order, leading to incorrect outputs. More specifically, this is because the finishing time of a vertex (in the first DFS) being minimum does **not** imply that the finishing time of its SCC will be minimum. In contrast, by definition, the finishing time of a vertex (in the first DFS) being maximum **does** imply that the finishing time of its SCC will be maximum. This is why Kosaraju's works: transposing G and visiting vertices in decreasing order of finishing times (aka starting from the maximum) in the second DFS **guarantees** that we are visiting the SCCs of G^T in reverse topologically sorted order.

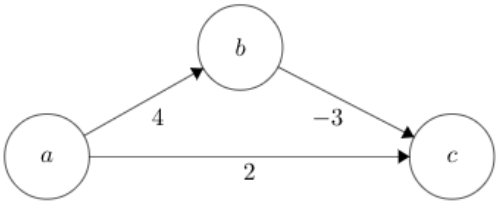
Problem 3: True or False

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.

Solution

1. **False.** Note that the conclusion is clearly false if there are negative weight cycles – the notion of a shortest path doesn't even make much sense in this case. But this is not the only case where Dijkstra's will not work with negative edge weights. Recall that the proof of correctness for Dijkstra's algorithm relies on non-negative edge weights to yield the correct solution because it implies that we can never decrease a path's weight by traversing more edges, allowing us to incrementally make locally optimal decisions to reach a globally optimal solution. However, this assumption breaks when we have negative edge weights.

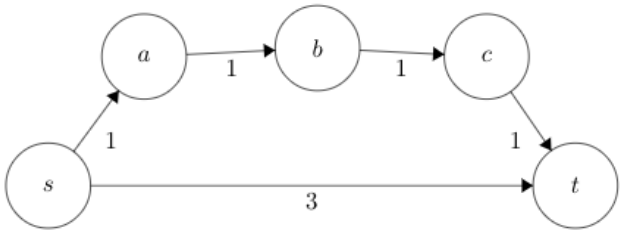
   For instance, consider the counterexample below. If we run Dijkstra's algorithm starting at vertex a, we will update b and c's distances to be 4 and 2, respectively. Then, we will conclude that the shortest path from a to c is $a \to c$, with a weight of 2. However, the actual shortest path from a to c is $a \to b \to c$, with a weight of 1. The assumption in the proof of correctness breaks here because by traversing the $b \to c$ edge which has a negative weight, we are able to decrease the weight of the path from a to c that goes through b. Hence, running Dijkstra's here yields an incorrect solution.



2. **False.** The algorithm will terminate, since in each iteration of our while loop, we still remove the vertex at the root of the heap, and the algorithm terminates when the heap is empty. But as we noted above, when the algorithm terminates, the output could be incorrect.
3. **True.** Any scaling by a positive factor on the weights does not affect the calculation of shortest paths give because we maintain relative path weights. Consider a path with total weight a and a path with

total weight b, where WLOG $a < b$. When we apply this transformation of doubling edge weights, observe that our first path now has total weight $2a$ and our second path now has total weight $2b$. In other words, since $2a < 2b$ still, scaling by a positive factor maintains these relative path weights, so Dijkstra's still produces the same shortest path. Alternatively, this is analogous to unit-conversion. For example, converting edge weights from miles to kilometers will not affect the shortest path.

4. **False.** In contrast to above, squaring the weights of all edges will not always produce the same shortest path because it is not a transformation that preserves relative path weights. As a counterexample, consider the graph below and the shortest path from s to t. In this original graph, the shortest path is just from $s \to t$; however, after squaring the edge weights, the shortest path becomes $s \to a \to b \to c \to t$.



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.

Solution

Algorithm: We run a modified version of Dijkstra's: we initialize the distances to $-\infty$ instead of ∞; we initialize the distance of the source node to be 1 instead of 0; we use a max-heap and call Extract-Max instead of using a min-heap and calling Extract-Min; and in the edge-relaxation step, we switch the inequality to a $<$ instead of a $>$ and check for products instead of sums. After the algorithm terminates, we backtrack from v to the source u via parent pointers to output the path with maximum reliability. The pseudocode is as follows:

```
Maximum-Reliability(G, s)
    for each v ∈ V do
        dist[v] = −∞
        parent[v] = NIL

    dist[s] = 1

    S = ∅
    Q = max-priority queue on all vertices, keyed by dist value

    while Q is not empty do
        u = Extract-Max(Q)
        S = S ∪ {u}
        for each v ∈ Adj[u] do
            if v ∈ Q and dist[v] < dist[u] × w(u, v) then
```

```
dist[v] = dist[u] × w(u, v)
parent[v] = u
```

Proof of Correctness: Since edge weights represent reliabilities, observe that we want to find the path from u to v with the maximum weight, where the weight of a path is now the product (instead of sum) of all weights along the path because reliabilities are independent. At a high-level, our algorithm modifies Dijkstra's by ensuring we keep track of maximum instead of minimum length paths. Formally, we can prove the correctness of our algorithm by modifying the proof of correctness for Dijkstra's. That is, consider the set S at any point in the algorithm's execution. We want to show via induction on $|S|$ that for each $u \in S$, the path P_u is a path $s \rightsquigarrow u$ with maximum reliability.

Base Case: $S = \{s\}$ and $dist[s] = 1$, so $|S| = 1$ holds because the maximum reliability of any path is just 1.

Induction Hypothesis: Assume the claim holds when $|S| = k$ for some $k \geq 1$.

Induction Step: We want to show the claim holds for some S with size $k+1$. Consider the $k+1$-th vertex v. Let (u, v) be the last edge on our $s \rightsquigarrow v$ path P_v. By IH, we know P_u is a $s \rightsquigarrow u$ path with maximum reliability. Now, we want to show that any other $s \rightsquigarrow v$ path P has reliability at most the reliability of P_v. Note that in order to reach v, this path P must have left the set S. Let y be the first node on P that is not in S and let $x \in S$ be the node on P just before y such that we have (x, y).

Observe that P cannot have higher reliability than P_v because it already has at most the reliability of P_v by the time it has left the set S. This is because in iteration $k+1$, our algorithm would have considered adding y to the set S via this (x, y) edge but rejected it in favor of adding v since we use a max-heap. So, the reliability of the subpath of P up until y has at most the reliability of P. Since edge weights are between 0 and 1, we know that the reliability of P can only decrease as we traverse more edges since we are dealing with products. Therefore, we know that P cannot have higher reliability than P_v – so P_v is a $s \rightsquigarrow v$ path with maximum reliability, completing our Induction Step and thus our proof.

Runtime Analysis: Note that none of the modifications affect the runtime of Dijkstra's algorithm, so running our modified version also takes $O((m+n)\log n)$ time. We backtrack no more than $O(n)$ times since the longest path in a graph has $n-1$ edges. Therefore, this algorithm runs in $O((m+n)\log n)$ time.

Alternate Solution

Algorithm: Modify G so that the weight of an edge (u, v) is equal to $-\log(r(u, v))$, or the negative log of its reliability. If r(u, v) = 0, set the weight of edge (u, v) to ∞. Run Dijkstra's now on G. Backtrack from v to the source node u via parent pointers to output the path with the maximum reliability.

Proof of Correctness: Essentially, we want to prove that our algorithm has modified G such that we transformed this problem into a shortest path problem. Since edge weights represent reliabilities, observe that we want to find the path from u to v with the maximum weight, where the weight of a path is now the product (instead of sum) of all weights along the path because reliabilities are independent. Note that $-\log r(u, v) = \log(1/r(u, v))$. By taking the inverse of every $r(u, v)$, we have converted this maximization problem into a minimization problem, and by then taking the log of these inverses, we have converted the problem of maximizing a product into minimizing a sum because of log properties. Therefore, the transformation we applied has converted the problem into a shortest path problem. The newly transformed edge weights are all non-negative, so we know that Dijkstra's algorithm will correctly calculate all shortest paths and thus backtracking will yield the most reliable path.

Runtime Analysis: Modifying G requires iterating through its adjacency list and updating the edge weights, which takes $O(m+n)$ time. Running Dijkstra's takes $O((m+n)\log n)$ time and we backtrack no

more than $O(n)$ times since the longest path in a graph has $n-1$ edges. Therefore, this algorithm runs in $O((m+n)\log n)$ time.