Shortest Paths: Dijkstra’s Algorithm
CMPSC 465 – References: Epp 10.7, CLRS Chapter 22, KT 4.4

I. Background on the Shortest Path Problem

Warm-Up Problem:
Consider the following directed graph:

![Graph Image]

a. You’ll notice one path from $s$ to $t$ is highlighted. Sum the weights of the edges on this path.

b. List at least 3 other paths from $s$ to $t$. Sum the weights of the edges in each path.

An important problem is to find a shortest path in a graph. Here are some specifics on the problem:

- One input is a weighted graph, $G$.
- We’ll have a weight function $w: E(G) \rightarrow \mathbb{R}$, where $w(e)$ is the weight of edge $e$.
- We also need a source vertex $s$ and a destination $t$.
- The problem is to find the shortest directed path from $s$ to $t$.
- The weight of path $p = (v_0, v_1, \ldots, v_k)$ is given by

$$\sum_{i=0}^{k} w(v_{i-1}, v_i)$$

In the graph we looked at above, the highlighted edge corresponds to the shortest path from $s$ to $t$. Our challenge will be to find a shortest path from $s$ to $t$ in any graph.

Pay close attention to language once again: a shortest path is not necessarily the only shortest path.
II. Dijkstra’s Algorithm

Dijkstra’s Algorithm solves the shortest paths problem when all edge weights are ____________________________.

Here how it goes:

- We maintain a set of **explored nodes** __________ for which we have determined…

- In order to kick off the algorithm, we initialize
  - $S =$
  - $d(s) =$

- At each step, we…
  - Consider the function that sums the weight of the shortest path to some vertex $u$ in the explored part, followed by a single edge $(u, v)$. We want to choose the unexplored node $v$ which minimizes this function. More formally, this function is
    - Add $v$ to $S$.
    - Set $d(v)$ to __________.

Here is a visualization:

Here is pseudocode for the algorithm for a connected weighted graph $G$, source $s$, and destination $t$:

```pseudo
Dijkstra(G, s, t)
{
  Define set $S$ // set of explored nodes
  Define auxiliary $d(u)$ for elements of $S$ // minimum distance from $s$ to each node $u$
  $S = \{s\}$ // start with source $s$ as explored and
  $d(s) = 0$ // no distance to get to it

  while $S \neq V(G)$
  {
    find a node $v \not\in S$ with at least one edge from $S$ for which $d'(v) = \min_{(u,v)\in E} d(u) + w(u,v)$
    $S = S \cup \{v\}$
    $d(v) = d'(v)$
  }

  return $d(t)$
}
```
Notice that Dijkstra’s algorithm really finds the minimum distance from $s$ to all nodes in $V(G)$. We could easily adapt the loop test to stop whenever we have calculated $d(t)$, though.

Also notice the similarities between Dijkstra and other algorithms:

- Like Prim, we explore from a source vertex. But, instead we minimize distance from the source.
- Like traversals and Prim, we have a set of unexplored vertices and a set of explored vertices.

### III. Examples

**Example:**
Execute Dijkstra’s algorithm on our PSU graph. Find the shortest path from Old Main to all other buildings.

**Problem** (with some CSE dept. history):
Suppose it’s winter break of 2003-2004 when the CSE department is moving from Pond to IST. Suppose edge weights in the graph above represent times it takes in walk each path. Find the shortest path between these buildings, so that you can carry something between them while spending the least amount of time outside.
Example: Find all shortest paths from $s$ in the following directed graph:

![Directed Graph]

IV. Dijkstra’s Algorithm Implementation

For each *unexplored* node, we can explicitly maintain the value of …

In this model, the next node to explore is ________________.

We can maintain the unexplored nodes via….

Every time we explore a vertex $v$, we update…
V. Performance Analysis of Dijkstra’s Algorithm

First, let’s consider what would happen if we did not use a priority queue:

- The while loop runs \( \text{times} \).
- Finding an edge which adds a new vertex to \( S \) is a matter of finding a minimum among remaining edges. This requires scanning \( \text{edges} \).
- So, such an implementation would require \( \text{time} \).

The running time of Dijkstra depends on the implementation of the priority queue. Let’s assume we’re using a binary heap, like we studied in CLRS 6.5.

Then…

- Each selection of a node is an extraction of a min, which takes \( \text{time} \).
- Just calling the PQ operation to change the priority of a node takes \( \text{time} \).
- The call to the priority change can happen at most once per \( \text{over the whole graph} \).
- So, the total running time is…

VI. Correctness Proof of Dijkstra’s Algorithm

Let’s prove Dijkstra’s algorithm correct. More precisely, we’ll prove that when a connected, simple graph with positive edge weights is input to Dijkstra’s algorithm with source \( a \) and destination \( z \), the output is a shortest path from \( a \) to \( z \). We’ll do this like using an invariant, but closer to traditional induction on the size of \( S \). We define the following property:

Proof:
We proceed via induction.

In the proof, we’ll use the notation \( \text{LSP}(a, y) \) to be the length of a shortest path from \( a \) to \( y \) and use other notation from the algorithm.

Base:
Inductive Step:
VII. Optimization Problems and Algorithmic Strategies

Both the shortest paths problem and minimum spanning tree problems were **optimization problems**, in that they look for the best possible solution given some constraints.

All three of Kruskal’s, Prim’s, and Dijkstra’s algorithms fall into a class of algorithms called **greedy algorithms**. Kleinberg/Tardos perhaps defines this best: “It is hard, if not impossible, to define precisely what is meant by a greedy algorithm. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step that myopically to optimize some underlying criterion.”

This fits in that…

- Dijkstra selects one vertex to add at each step that is the best choice of distances for one new vertex, growing \( S \), not worrying about vertices farther from the source.
- Kruskal picks a new edge of least weight that doesn’t yield a cycle, not thinking about whether there might be a better edge choice that would yield a more optimal tree when more edges are added.
- Prim picks a new vertex to add, looking at just minimum edge choices at one step, again not thinking about what that does for picking vertices farther from the growing MST.

The greedy strategy works in these cases. In all of these cases, we fit the motto “the greedy algorithm stays ahead.” But, the greedy strategy doesn’t always work. We’ll find that we’ll need to use a technique called dynamic programming to solve some optimization problems we can’t handle via greedy algorithms. A dynamic programming algorithm called Bellman-Ford also solves the shortest paths problem, and handles the case with negative edge weights.

**Homework:** CLRS Exercises 24.3-1, 24.3-3