4.4 Shortest Paths

- APIs
- shortest-paths properties
- Dijkstra’s algorithm
- edge-weighted DAGs
- negative weights

With added notes and slides by Betty O’Neil for cs310
We will cover Section 4.3 after this one (4.3 and 4.4 are pretty much independent of each other)
Data structures for single-source shortest paths

Goal. Find the shortest path from \( s \) to every other vertex.

Well, if two paths are the same length, we’re happy enough to have one of them. So the real goal: find a shortest path from \( s \) to every other vertex. Those make up the SPT, the shortest-path tree.

Here is an example SPT. See how it reaches out like the nerves in the nervous system to reach all the vertices, from the source vertex. The gray links are the edges outside the SPT.

If there were two paths of the same length to some vertex, we could trim away the last edge of one of them, and do this recursively until there remains only a tree.
Data structures for single-source shortest paths

Goal. Find the shortest path from \( s \) to every other vertex.

Progress on this: there is an SPT (shortest-path tree) for \( s \).

Consequence. Can represent the SPT with two vertex-indexed arrays:
- \( \text{distTo}[v] \) is length of shortest path from \( s \) to \( v \).
- \( \text{edgeTo}[v] \) is last edge on shortest path from \( s \) to \( v \). (not vertex # as before, but the edge object)

Recall how we used \( \text{edgeTo}[] \) to find paths after doing a dfs. Same idea here: work backwards from destination vertex by finding its \( \text{edgeTo} \) entry, then work backwards from that vertex, and so on.

![shortest-paths tree from 0](image)
Recall how we used `edgeTo[]` to find paths after doing a dfs. Same idea here: work backwards from destination vertex by finding its `edgeTo` entry, then work backwards from that vertex, and so on.

For example, find path to 6 (from 0, the source vertex here):
- `edgeTo[6] = 3`, so last edge on path is 3->6
- `edgeTo[3] = 7`, so previous edge is 7->3
- `edgeTo[7] = 2`, so previous edge is 2->7
- `edgeTo[2] = 0` so done: path is 0->2->7->3->6
Goal. Find the shortest path from $s$ to every other vertex.
Observation. A shortest-paths tree (SPT) solution exists.
Consequence. Can represent the SPT with two vertex-indexed arrays:

- $\text{distTo}[v]$ is length of shortest path from $s$ to $v$.
- $\text{edgeTo}[v]$ is last edge on shortest path from $s$ to $v$.

As with paths in DFS, we can use a Stack to store the edges as we find them in reverse order, and we’ll end up with a collection in the right order. (This is a S&W Stack from algs4.jar). The $\text{pathTo}$ caller gets back an $\text{Iterable<DirectedEdge>}$, a simple sequence of DEs.

```java
public double distTo(int v) {
    return distTo[v];
}

public Iterable<DirectedEdge> pathTo(int v) {
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}
```
**Edge relaxation step**

- This is used during an algorithm to find shortest paths
- Here the distTo[] and edgeTo[] values are still in flux
- The dark edges are the explored ones, already in edgeTo[].
- Scenario: edge e is a yet-unexplored edge attached to a vertex that is already in the arrays: does it help with paths to its other vertex?
- Note: its other vertex may or may not be already explored.

Relax edge $e = v \rightarrow w$. See if it can provide a better path to $w$.

- $\text{distTo}[v]$ is length of shortest known path from $s$ to $v$.
- $\text{distTo}[w]$ is length of shortest known path from $s$ to $w$.
- $\text{edgeTo}[w]$ is last edge on shortest known path from $s$ to $w$.

✓ Here, $w$ hasn’t been visited at all yet, so any path to it is welcome: $\text{distTo}[w] = \text{distTo}[v] + \text{weight-of-e}$
Edge relaxation step: more general case where a previous path to \( w \) exists already.

Relax edge \( e = v \rightarrow w \). See if it can reduce \( \text{distTo}[w] \).

- \( \text{distTo}[v] \) is length of shortest known path from \( s \) to \( v \).
- \( \text{distTo}[w] \) is length of shortest known path from \( s \) to \( w \).
- \( \text{edgeTo}[w] \) is last edge on shortest known path from \( s \) to \( w \).
- If \( e = v \rightarrow w \) gives shorter path to \( w \) through \( v \), update both \( \text{distTo}[w] \) and \( \text{edgeTo}[w] \).

Case that \( v \rightarrow w \) successfully reduces \( \text{distTo}[w] \) from 7.2 to 4.4

“Relax” like rubber band finds shortest route when stretched.
Edge relaxation: code

Relax edge \( e = v \rightarrow w \).

- distTo[v] is length of shortest known path from s to v.
- distTo[w] is length of shortest known path from s to w.
- edgeTo[w] is last edge on shortest known path from s to w.
- If \( e = v \rightarrow w \) gives shorter path to w through v, update both distTo[w] and edgeTo[w].

```java
private void relax(DirectedEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```
**Proposition.** Let $G$ be an edge-weighted digraph. Then $\text{distTo[]}$ are the shortest path distances from $s$ iff:

- $\text{distTo}[s] = 0$.
- For each vertex $v$, $\text{distTo}[v]$ is the length of some path from $s$ to $v$.
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight()}$.

- Proof: see text
Generic shortest-paths algorithm

**Generic algorithm (to compute SPT from $s$)**

- Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.
  - Repeat until optimality conditions are satisfied:
    - Relax any edge.

Proof: see text

Note that although this is true, you make no progress if you relax an edge with neither vertex yet explored: it’s just distance infinity vs infinity. In practical algorithms, we always explore in an organized fashion.
Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).
Ex 2. Topological sort algorithm (no directed cycles).
Ex 3. Bellman-Ford algorithm (no negative cycles).

Generic algorithm (to compute SPT from $s$)

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat until optimality conditions are satisfied:

- Relax any edge.
4.4 Shortest Paths

- APIs
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights

http://algs4.cs.princeton.edu
“Do only what only you can do.”

“In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”

“The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.”

“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”

“APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.”
"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

-- Edsger Dijkstra
Dijkstra's algorithm: Also see page 653

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm: Also see slides at Princeton, page 653

• Consider vertices in increasing order of distance from \( s \) (non-tree vertex with the lowest known \text{distTo[]} value).

• Add vertex to tree and relax all edges pointing from that vertex.

First round: find edges out from \( s \), the initial “eyeball”, find lowest-weight one, here 5 going out \( 0 \rightarrow 1 \). Add this to the tree, i.e. start tree.
Dijkstra's algorithm

• Consider vertices in increasing order of distance from s (non-tree vertex with the lowest known distTo[] value).
• Add vertex to tree and relax all edges pointing from that vertex (the “eyeball”, since it is looking around from there)

First round: eyeball starts at s. Find edges out from s, find lowest-weight one, here 5 going out 0 -> 1 Add this to the tree, i.e. start tree. Eyeball moves to new vertex, ready to look around at adjacent edges
Dijkstra's algorithm

- Consider vertices in increasing order of distance from \( s \) (non-tree vertex with the lowest known \( \text{distTo}[] \) value).
- Add vertex to tree and relax all edges pointing from that vertex.

Second round: find edges out from 1, newly added tree vertex and the current eyeball
Two ways to 7: old way 0->7 costs 8, new way 0->1->7 costs 5+4 = 9, higher, so old value stays.
Find lowest-distTo vertex, here 8. Add this to the tree and move eyeball to it.
Dijkstra's algorithm: Also see page 653

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest known distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.

Third round: find edges out from 7, newly added tree vertex and current eyeball
Two ways to 2: old way 0->1->2 costs 17, 0->7->2 costs 15, lower, so new value recorded.
find lowest-distTo one, here 9 going out 0 -> 4  Add this to the tree and move eyeball
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.

Partial shortest-paths tree from vertex s

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1*</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4*</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5*</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7*</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

distTo values are using the tree plus one more edge to destination. For example v = 2 has 14 by ...->5->2 (better than ...->7->2 and ...->1->2). It has lowest non-tree distTo of 14 (out of 14, 20, and 26), so is chosen to be added to the tree, and underlined above.
Dijkstra's algorithm

• Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $\text{distTo}[]$ value).
• Add vertex to tree and relax all edges pointing from that vertex.
  • Here: 2->3 and 2->6

<table>
<thead>
<tr>
<th>$v$</th>
<th>$\text{distTo}[]$</th>
<th>$\text{edgeTo}[]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1*</td>
<td>5.0</td>
<td>0-&gt;1</td>
</tr>
<tr>
<td>2*</td>
<td>14.0</td>
<td>5-&gt;2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2-&gt;3</td>
</tr>
<tr>
<td>4*</td>
<td>9.0</td>
<td>0-&gt;4</td>
</tr>
<tr>
<td>5*</td>
<td>13.0</td>
<td>4-&gt;5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2-&gt;6</td>
</tr>
<tr>
<td>7*</td>
<td>8.0</td>
<td>0-&gt;7</td>
</tr>
</tbody>
</table>

Partial shortest-paths tree from vertex $s$: one step along

14+11 = 25, better than old 26

14+3 = 17, better than old 20
Dijkstra's algorithm completed

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

Final state: shortest-paths tree SPT from vertex s
Dijkstra's algorithm: correctness proof

**Proposition.** Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

**Pf**

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when vertex $v$ is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e\.\text{weight()}$.

Inequality holds until algorithm terminates because:

- $\text{distTo}[w]$ cannot increase $\leftarrow \text{distTo[] values are monotone decreasing}$
- $\text{distTo}[v]$ will not change $\leftarrow$ we choose lowest $\text{distTo[]}$ value at each step (and edge weights are nonnegative)

Thus, upon termination, shortest-paths optimality conditions hold. $\blacksquare$
Data Structure for this?

We see the two arrays are crucial, but if we try to use just these, what does it look like?

Well, need to track SPT, so use mark array for in-SPT. Using Weiss’s terminology, call the working vertex the “eyeball”. It does look around for the best way to progress.

Set all distTo’s to infinity, except the source, to 0
Set up marked array
Let eyeball = s
Loop
  update distTo for all unmarked neighbors of eyeball
  find min distTo among unmarked vertices as new eyeball and mark it

That’s O(V^2), but performance of Dijkstra is supposed to by O(ElogV) (see page 682) which for sparse graphs, is much less.
Data Structure for this?

Set all distTo’s to infinity, except the source, to 0
Set up marked array
Let eyeball = s
Loop
    for update distTo for all unmarked neighbors of eyeball ← not so bad
    find min distTo among unmarked vertices ← bad, if loop through array
        as new eyeball and mark it ← not so bad

How do we hold a collection of elements so it’s easy to get the minimum one out over and over?
Data Structure for this?

• How do we hold a collection of elements so it’s easy to get the minimum one out over and over?
• Answer: Priority Queue, and its deleteMin, much faster (say log(V))

• Here the priority is the distTo value
• One problem: the distTo values change as the algorithm runs
• So it’s actually not a “normal” priority queue that we need, since those assume the priority stays constant for each element in the queue.
• It’s a souped-up priority queue with an additional method changeKey that allows us to change the priority of an element already in the PQ.
IndexMinPQ: a PQ with changeKey

• How can we soup-up a normal priority queue?
• One way: to change a key, delete that element, change its priority, and reinsert it.
• But delete isn’t in the normal API either...
• This approach can be fleshed out with a TreeMap

• This issue is not discussed in the text in this section, but you can see the Dijkstra code on page 655 uses a IndexMinPQ for its PQ.
• The IndexMinPQ is described on pp. 320-321 and partially implemented in the problems on page 333. You can see it’s made out of 3 arrays.
• It’s fully implemented at princeton.edu. So we can use it in Dijkstra.
• OK, given this powerful data structure, let’s see the Dijkstra’s algorithm code...
Dijkstra's algorithm: Java implementation

```java
public class DijkstraSP {
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        pq.insert(s, 0.0);

        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

Special PQ with changeKey
relax vertices in order of distance from s
Dijkstra's algorithm: Java implementation

private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}

decreaseKey is not listed in the API on page 320, just changeKey. But both are implemented in the class. decreaseKey has less code than the more general changeKey, so runs faster.
Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: \( V \) insert, \( V \) delete-min, \( E \) decrease-key.

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>insert</th>
<th>delete-min</th>
<th>decrease-key</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>( V )</td>
<td>1</td>
<td>( V^2 )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( \log V )</td>
<td>( \log V )</td>
<td>( \log V )</td>
<td>( E \log V )</td>
</tr>
<tr>
<td>d-way heap</td>
<td>( \log_d V )</td>
<td>( d \log_d V )</td>
<td>( \log_d V )</td>
<td>( E \log_{EN} V )</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>1 ( \dagger )</td>
<td>( \log V \dagger )</td>
<td>1 ( \dagger )</td>
<td>( E + V \log V )</td>
</tr>
</tbody>
</table>

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs. (I think this must refer to IndexMinPQ, since a simple binary heap can’t do decrease-key in \( \log V \) time)
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

\( \dagger \) amortized
Shortest paths in edge-weighted DAGs

Using topological sorting along with relaxation, we can compute SPT in any edge-weighted DAG in time proportional to $E + V$.

See page 658

Negative edge weights, non-DAG case.

Dijkstra. Doesn’t work with negative edge weights.

Don’t despair, look at Bellman-Ford for this case, as long as there are no negative cycles. See pp 668-678