4.4 Shortest Paths

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

Plan for covering Sections 4.3 and 4.4

- Note we are skipping Sec. 4.3 for now
- It’s on Minimum Spanning Trees, an important problem but not related to our graph project
- Whereas Sec. 4.4 is on Shortest Paths, very relevant.
- We’ll come back to Sec. 4.3 to finish up.

Shortest paths in an edge-weighted digraph

Given an edge-weighted digraph, find the shortest path from \( s \) to \( t \).

shortest path from 0 to 6

edge-weighted digraph

Google maps

Shortest path applications

- PERT Charts. Program Evaluation Review Technique
- Map routing.
- Texture mapping.
- Robot navigation.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Restrictions on edge weights?
- Nonnegative weights.
- Euclidean weights.
- Arbitrary weights.

Cycles?
- No directed cycles.
- No "negative cycles."

Simplifying assumption. Shortest paths from \( s \) to each vertex \( v \) exist.

http://algs4.cs.princeton.edu

4.4 **Shortest Paths**

**APIs**

- weighted directed edge
  
  ```java
  public class DirectedEdge
  {
    private final int v, w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight)
    {
      this.v = v;
      this.w = w;
      this.weight = weight;
    }

    public int from()
    {
      return v;
    }

    public int to()
    {
      return w;
    }

    public double weight()
    {
      return weight;
    }

    public String toString()
    {
      return String.valueOf(v) + "→" + String.valueOf(w) + " w = " + String.valueOf(weight);
    }
  }
  ```

**Idiom for processing an edge e**: `int v = e.from(), w = e.to();`

**Central part of MBTA track map**. Shows that the Green line tracks cross above the Red line tracks at Park St., and also above the Blue line tracks at Government Center. They were there first, apparently. The colored rectangles represent the platforms. So we see that the transfers between lines involve vertical movement, usually using stairs.

**3-D model of Tokyo’s subway system**: makes ours look simple!

**How we want to use this in pa4**

With this setup, the cost of going two stops on the same line (top to bottom or vice versa, or left to right or vice versa) is 2, but the cost of going from the left to the bottom or top is 9, because of the estimated 7 minute wait of the transfer.

**3-D model of Tokyo’s subway system**: makes ours look simple!

History: Boston’s first electrified rapid transit line, now the Orange Line (but rebuilt), was opened in 1901, after the London Underground, the Chicago L, and the Paris Metro, but before New York (1904), the system now with the highest number of stations.

**How does this look in 3-D?**

**Weighted directed edge**: implementation in Java

Similar to Edge (of Sec. 4.3, not yet covered) for undirected graphs, but a bit simpler.

```java
public class DirectedEdge
{
  private final int v, w;
  private final double weight;

  public DirectedEdge(int v, int w, double weight)
  {
    this.v = v;
    this.w = w;
    this.weight = weight;
  }

  public int from()
  {
    return v;
  }

  public int to()
  {
    return w;
  }

  public double weight()
  {
    return weight;
  }

  public String toString()
  {
    return String.valueOf(v) + "→" + String.valueOf(w) + " w = " + String.valueOf(weight);
  }
}
```
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. These are directed edges.

![Image of SPT](image)

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.

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[0]` = 1, so last edge on path is 1->6
- `edgeTo[1]` = 7, so previous edge is 7->3
- `edgeTo[7]` = 2, so previous edge is 2->7
- `edgeTo[2]` = 0 so done: path is 0->7->3->6

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.

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:

- \(distTo[v]\) is length of shortest path from \(s\) to \(v\).
- \(edgeTo[v]\) is last edge on shortest path from \(s\) to \(v\). (not vertex \(x\) as before, but the whole edge object)

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 `pathTo` caller gets back an `Iterable<DirectedEdge>`, a simple sequence of DEs.

```java
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.

  ![Image of edge relaxation](image)

  **Relax edge** \(e = s \rightarrow w\). See if it can provide a better path to \(w\):

  - \(distTo[w]\) 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 \(w\).

  ✓ In the above picture, \(w\) hasn’t been visited at all yet, so any path to it is welcome: `distTo[w] = distTo[v] + weight of e`
**Edge relaxation: code**

```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;
    }
}
```

**Shortest-paths optimality conditions**

**Proposition.** Let G be an edge-weighted digraph. Then \( \text{distTo}[] \) are the shortest path distances from s iff:

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

**Proof:** see text

**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).

**Edsger W. Dijkstra: select quotes**

- “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.”
Dijkstra's algorithm: Also see page 653

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

First round: eyeball starts at \( s \). Find edges out from \( s \), find lowest-weight one, here 5 going out 0 \( \rightarrow \) 5. Add this to the tree, i.e. start tree with edge 0 \( \rightarrow \) 5 and move eyeball to that lowest-weight vertex.

Second round: find edges out from 1, newly added tree vertex and the current eyeball

Dijkstra's algorithm

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

First round: eyeball starts at \( s \). Find edges out from \( s \), find lowest-weight one, here 5 going out 0 \( \rightarrow \) 5. Add this to the tree, i.e. start tree with edge 0 \( \rightarrow \) 5 and move eyeball to that lowest-weight 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 \( \text{distTo}[\cdot] \) value).
- Add vertex to tree and relax all edges pointing from that vertex.

First round: eyeball starts at \( s \). Find edges out from \( s \), find lowest-weight one, here 5 going out 0 \( \rightarrow \) 5. Add this to the tree, i.e. start tree with edge 0 \( \rightarrow \) 5 and move eyeball to that lowest-weight vertex.

Dijkstra on Java (summary, not a quote)

It’s just as bad as C, C++, etc., all the imperative languages. Students should learn a functional language in college to shock them into thinking.

See Dijkstra on Haskell and Java, a letter of 2001. Haskell is a functional language.

Edsger W. Dijkstra: select quotes

Dijkstra on Haskell and Java

Shocked them into thinking.

Learn a functional language in college to shock them into thinking.
Consider vertices in increasing order of distance from \( s \) (non-tree vertex with the lowest known \( \text{distTo}[v] \) value.

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

Thus round: find edges out from 7, newly added tree vertex and current eyeball.

Pf: Dijkstra’s algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

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

Inequality holds until algorithm terminates because:

- \( \text{distTo}(v) \) cannot increase (\( \text{distTo} \) values are monotonically decreasing).
- \( \text{distTo}(v) \) will not change (we choose lowest \( \text{distTo} \) value at each step and edge weights are nonnegative).

Thus, upon termination, shortest-paths optimality conditions hold.

This is a smart greedy algorithm: look around on each pass, find the lowest-cost way to add an edge.

\[ \text{distTo}[s] = 0; \text{distTo}[v] = \text{e.weight}(); \text{distTo}[w] = \text{distTo}(v) + \text{e.weight}(); \]

\[ \text{distTo}(v) \leq \text{distTo}[w] \]

\[ \text{distTo}(v) = \text{distTo}[w] \]

\[ \text{distTo}(v) \geq \text{distTo}[w] \]

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 a mark array for in-SPT. Using Weiss’s terminology, call the working vertex the ‘eyeball’. It does define where look around from, for the best way to progress. S&W doesn’t use ‘eyeball’.

Set all \( \text{distTo} \)’s to infinity, except the source, to 0

Set up marked array

Let eyeball = \( s \)

Loop

- update \( \text{distTo} \) for all unmarked neighbors of eyeball
- find min \( \text{distTo} \) among unmarked vertices for new eyeball and mark it

That’s O(V^2), but performance of Dijkstra is supposed to by O((E + \( V \log V \))).
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?

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.
- 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);
        }
    }

    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]);
        }
    }
}
```

Dijkstra's algorithm: which priority queue?

- 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
- 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.

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>V</td>
<td>V</td>
<td>V</td>
<td>V^2</td>
</tr>
<tr>
<td>IndexMinPQ</td>
<td>log V</td>
<td>log V</td>
<td>V</td>
<td>V log V</td>
</tr>
<tr>
<td>d-way heap</td>
<td>log V</td>
<td>log V</td>
<td>log V</td>
<td>V log V</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>1</td>
<td>1</td>
<td>log V</td>
<td>V</td>
</tr>
</tbody>
</table>

Bottom line:
- Array implementation optimal for dense graphs.
- 4-way heap worth the trouble in performance-critical situations.
- K-ary heap article at GeeksForGeeks
- K=2 is the familiar binary heap, with added decrease-key method
- Fibonacci heap best in theory, but not worth implementing.
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.

![Diagram](image)

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