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

Shortest path variants

Which vertices?

**Single source:** from one vertex $s$ to every other vertex.

**Single sink:** from every vertex to one vertex $t$.

**Source-sink:** from one vertex $s$ to another $t$.

**All pairs:** between all pairs of vertices.

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.
4.4 Shortest Paths

- APIs
  - shortest-paths.properties
  - Dijkstra's algorithm
  - edge-weighted DAGs
  - negative weights
**Weighted directed edge API**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectedEdge(int v, int w, double weight)</td>
<td>weighted edge v→w</td>
</tr>
<tr>
<td>int from()</td>
<td>vertex v</td>
</tr>
<tr>
<td>int to()</td>
<td>vertex w</td>
</tr>
<tr>
<td>double weight()</td>
<td>weight of this edge</td>
</tr>
<tr>
<td>String toString()</td>
<td>string representation</td>
</tr>
</tbody>
</table>

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

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.
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 int weight() {
        return weight;
    }
}
```

Vs. undirected Graph: from() and to() replace either() and other() (used in sec. 4.3)
Conventions. Allow self-loops and parallel edges.
Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

V

8
15
4 5 0.35
5 4 0.35
4 7 0.37
5 7 0.28
7 5 0.28
5 1 0.32
0 4 0.38
0 2 0.26
7 3 0.39
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93

adj

0

1

2

3

4

5

6

7

Bag objects

reference to a DirectedEdge object

0 2 0.26 0 4 0.38
1 3 0.29
2 7 0.34
3 6 0.52
4 7 0.37 4 5 0.35
5 1 0.32 5 7 0.28 5 4 0.35
6 4 0.93 6 0 0.58 6 2 0.40
7 3 0.39 7 5 0.28
Same as EdgeWeightedGraph (of Sec. 4.3) except replace GraphWith Digraph.

```java
public class EdgeWeightedDigraph {
    private final int V;
    private final Bag<DirectedEdge>[] adj;
    public EdgeWeightedDigraph(int V) {
        this.V = V;
        adj = (Bag<DirectedEdge>[])(new Bag[V]);
        for (int v = 0; v < V; ++v)
            adj[v] = new Bag<DirectedEdge>();
    }

    public void addEdge(DirectedEdge e) {
        int v = e.from();
        adj[v].add(e);
    }

    public Iterable<DirectedEdge> adj(int v) {
        return adj[v];
    }
}
```

add edge $e = v \rightarrow w$ to only v's adjacency list
**Goal.** Find the shortest path from $s$ to every other vertex.

Here SP stands for DijkstraSP, pg. 655, or AcyclicSP, pg. 660, or BellmanFordSP, pg. 674

```java
public class SP

SP(EdgeWeightedDigraph G, int s)  // shortest paths from s in graph G
    double distTo(int v)  // length of shortest path from s to v
    Iterable<DirectedEdge> pathTo(int v)  // shortest path from s to v
    boolean hasPathTo(int v)  // is there a path from s to v?
```

```java
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
    StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
    for (DirectedEdge e : sp.pathTo(v))
        StdOut.print(e + " ");
    StdOut.println();
}
```

(output on next slide)
Single-source shortest paths API: S&W pseudo-interface+constructor

**Goal.** Find the shortest path from $s$ to every other vertex.

Here SP stands for DijkstraSP, pg. 655, or AcyclicSP, pg. 660, or BellmanFordSP, pg. 674.

```java
public class SP

SP(EdgeWeightedDigraph G, int s)  // shortest paths from s in graph G
    double distTo(int v)  // length of shortest path from s to v
    Iterable <DirectedEdge> pathTo(int v)  // shortest path from s to v
    Boolean hasPathTo(int v)  // is there a path from s to v?
```

```bash
% java SP tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38 4->5 0.35 5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26 2->7 0.34 7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38 4->5 0.35
0 to 6 (1.51): 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52
0 to 7 (0.60): 0->2 0.26 2->7 0.34
```
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. These are directed edges.

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 whole 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.
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:
• distTo[v] is length of shortest path from $s$ to $v$.
• 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 pathTo caller gets back an 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.

✓ In the above picture, 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 distTo[w].

- distTo[v] is length of shortest known path from s to v. (here 3.1)
- distTo[w] is length of shortest known path from s to w. (here 7.2)
- 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].

Case that $v \rightarrow w$ successfully reduces distTo[w] from 7.2 to 4.4 “Relax” like a rubber band that finds the shortest route when stretched.
Relax edge $e = v \rightarrow w$.

- $\text{distTo}[v]$ 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 $\text{edgeTo}[w]$.

```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 distTo[s] = 0 and distTo[v] = ∞ 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.
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.

**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).
4.4 Shortest Paths

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

\[
\begin{align*}
0 &\to 1 \quad 5.0 \\
0 &\to 4 \quad 9.0 \\
0 &\to 7 \quad 8.0 \\
1 &\to 2 \quad 12.0 \\
1 &\to 3 \quad 15.0 \\
1 &\to 7 \quad 4.0 \\
2 &\to 3 \quad 3.0 \\
2 &\to 6 \quad 11.0 \\
3 &\to 6 \quad 9.0 \\
4 &\to 5 \quad 4.0 \\
4 &\to 6 \quad 20.0 \\
4 &\to 7 \quad 5.0 \\
5 &\to 2 \quad 1.0 \\
5 &\to 6 \quad 13.0 \\
7 &\to 5 \quad 6.0 \\
7 &\to 2 \quad 7.0
\end{align*}
\]

an edge-weighted digraph
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 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 -> 1
Add this to the tree, i.e. start tree with edge 0->5 and move eyeball to that lowest-distTo vertex.
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 vertex 1 since it has lowest distTo, 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 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 \texttt{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 \( \text{distTo}[] \) 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>( \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→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>

\( \text{distTo} \) values are using the tree plus one more edge to destination. For example \( v = 2 \) has 14 by \( ...\rightarrow 5 \rightarrow 2 \) (better than \( ...\rightarrow 7 \rightarrow 2 \) and \( ...\rightarrow 1 \rightarrow 2 \)). It has lowest non-tree \( \text{distTo} \) 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 \( \text{distTo}[\cdot] \) value).
- Add vertex to tree and relax all edges pointing from that vertex.
  - Here: 2->3 and 2->6

Partial shortest-paths tree from vertex \( s \): one step along

\[ 14+3 = 17, \text{ better than old 20} \]

\[ 14+11 = 25, \text{ better than old 26} \]
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.

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.

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

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

```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;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

decreaseKey is not listed in the API for IndexMinPQ 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>IndexMinPQ</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.
- 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 decreaseKey method
- 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