4.4 Shortest Paths

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm

https://algs4.cs.princeton.edu
Shortest path in an edge-weighted digraph

Given an edge–weighted digraph, find a shortest path from one vertex to another vertex.

edge-weighted digraph

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

shortest path from 0 to 6
0 → 2 → 7 → 3 → 6

length of path = 1.51
(0.26 + 0.34 + 0.39 + 0.52)
Shortest path applications

- PERT/CPM.
- Map routing.
- **Seam carving.**
- Texture mapping.
- Robot navigation.
- Typesetting in \( \text{TeX} \).
- Currency exchange.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Shortest path variants

Which vertices?
- Source–destination: from one vertex to another vertex.
- Single source: from one vertex to every vertex.
- Single destination: from every vertex to one vertex.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?
- Non–negative weights.
- Euclidean weights.
- Arbitrary weights.

Directed cycles?
- Prohibit.
- Allow.

Simplifying assumption. Each vertex is reachable from \( s \).
Shortest paths: quiz 1

Which variant in car GPS? Hint: drivers make wrong turns occasionally.

A. Source–destination: from one vertex to another vertex.
B. Single source: from one vertex to every vertex.
C. Single destination: from every vertex to one vertex.
D. All pairs: between all pairs of vertices.
4.4 Shortest Paths

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm
Data structures for single-source shortest paths

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

**Observation 1.** There exists a shortest path from $s$ to $v$ that is simple.

**Observation 2.** A shortest-paths tree (SPT) solution exists. Why?

**Consequence.** Can represent a SPT with two vertex-indexed arrays:

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

---

$$\text{parent-link representation}$$

<table>
<thead>
<tr>
<th>$\text{distTo[]}$</th>
<th>$\text{edgeTo[]}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>5-&gt;1 0.32</td>
</tr>
<tr>
<td>2</td>
<td>0-&gt;2 0.26</td>
</tr>
<tr>
<td>3</td>
<td>7-&gt;3 0.37</td>
</tr>
<tr>
<td>4</td>
<td>0-&gt;4 0.38</td>
</tr>
<tr>
<td>5</td>
<td>4-&gt;5 0.35</td>
</tr>
<tr>
<td>6</td>
<td>3-&gt;6 0.52</td>
</tr>
<tr>
<td>7</td>
<td>2-&gt;7 0.34</td>
</tr>
</tbody>
</table>

---

shortest-paths tree from 0
Edge relaxation

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 \( e = v \rightarrow w \) yields shorter path from \( s \) to \( w \), via \( v \), update \( \text{distTo}[w] \) and \( \text{edgeTo}[w] \).
What are the values of $\text{distTo}[v]$ and $\text{distTo}[w]$ after relaxing edge $e = v \rightarrow w$?

A. 10.0 and 15.0
B. 10.0 and 17.0
C. 12.0 and 15.0
D. 12.0 and 17.0
Framework for shortest-paths algorithm

Generic algorithm (to compute a SPT from s)

For each vertex \( v \): \( \text{distTo}[v] = \infty \).
For each vertex \( v \): \( \text{edgeTo}[v] = \text{null} \).
\( \text{distTo}[s] = 0 \).
Repeat until \( \text{distTo}[v] \) values converge:
  - Relax any edge.

Key properties. Throughout the generic algorithm,
• \( \text{distTo}[v] \) is either infinity or the length of a (simple) path from \( s \) to \( v \).
• \( \text{distTo}[v] \) does not increase.
Framework for shortest-paths algorithm

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

For each vertex $v$: $\text{distTo}[v] = \infty$.
For each vertex $v$: $\text{edgeTo}[v] = \text{null}$.
$\text{distTo}[s] = 0$.
Repeat until $\text{distTo}[v]$ values converge:
  - Relax any edge.

Efficient implementations.

- Which edge to relax next?
- How many edge relaxations needed to guarantee convergence?

Ex 1. Bellman–Ford algorithm.
Ex 2. Dijkstra’s algorithm.
Ex 3. Topological sort algorithm.
4.4 Shortest Paths

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm

https://algs4.cs.princeton.edu
Weighted directed edge API

```java
public class DirectedEdge {
    DirectedEdge(int v, int w, double weight) {
        int from()
        int to()
        double weight()
    }

    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;
        }
    }
}
```

Relaxing an edge $e = v \rightarrow w$.

```
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;
    }
}
```
Weighted directed edge: implementation in Java

**API.** Similar to Edge 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; }
}
```

from() and to() replace either() and other()
**Edge-weighted digraph API**

API. Same as `EdgeWeightedGraph` except with `DirectedEdge` objects.

```java
public class EdgeWeightedDigraph

    EdgeWeightedDigraph(int V)  // edge-weighted digraph with V vertices
    void addEdge(DirectedEdge e)  // add weighted directed edge e
    Iterable<DirectedEdge> adj(int v)  // edges incident from v
    int V()  // number of vertices

    ;
    ;
```
Implementation. Almost identical to EdgeWeightedGraph.

```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<>();
    }

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

    public Iterable<DirectedEdge> adj(int v) {
        return adj[v];
    }
}
```
**Single-source shortest paths API**

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

```
public class SP

  SP(EdgeWeightedDigraph G, int s)  // shortest paths from $s$ in digraph $G$

  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 $v$?
```
4.4 Shortest Paths

- properties
- APIs
  - Bellman–Ford algorithm
  - Dijkstra’s algorithm

https://algs4.cs.princeton.edu
Bellman–Ford algorithm

For each vertex v: distTo[v] = ∞.
For each vertex v: edgeTo[v] = null.
distTo[s] = 0.
Repeat V–1 times:
  - Relax each edge.

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;
  }
}

for (int i = 1; i < G.V(); i++)
  for (int v = 0; v < G.V(); v++)
    for (DirectedEdge e : G.adj(v))
      relax(e);

number of calls to relax() in pass i =
  outdegree(0) + outdegree(1) + outdegree(2) + ... = E

Running time. Algorithm takes Θ(E V) time and uses Θ(V) extra space.
Bellman–Ford algorithm demo

Repeat $V - 1$ times: relax all $E$ edges.

An edge-weighted digraph:

- $0 \rightarrow 1$: 5.0
- $0 \rightarrow 4$: 9.0
- $0 \rightarrow 7$: 8.0
- $1 \rightarrow 2$: 12.0
- $1 \rightarrow 3$: 15.0
- $1 \rightarrow 7$: 4.0
- $2 \rightarrow 3$: 3.0
- $2 \rightarrow 6$: 11.0
- $3 \rightarrow 6$: 9.0
- $4 \rightarrow 5$: 4.0
- $4 \rightarrow 6$: 20.0
- $4 \rightarrow 7$: 5.0
- $5 \rightarrow 2$: 1.0
- $5 \rightarrow 6$: 13.0
- $7 \rightarrow 5$: 6.0
- $7 \rightarrow 2$: 7.0
Repeat $V - 1$ times: relax all $E$ edges.

**Bellman–Ford algorithm demo**

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>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>
Bellman–Ford algorithm: correctness proof

**Proposition.** Let \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k = v \) be any path from \( s \) to \( v \) containing \( k \) edges. Then, after pass \( k \), \( \text{distTo}[v_k] \leq \text{weight}(e_1) + \text{weight}(e_2) + \cdots + \text{weight}(e_k) \).

**Pf.** [by induction on number of passes \( i \)]

- **Base case:** initially, \( 0 = \text{distTo}[v_0] \leq 0 \).
- **Inductive hypothesis:** after pass \( i \), \( \text{distTo}[v_i] \leq \text{weight}(e_1) + \text{weight}(e_2) + \cdots + \text{weight}(e_i) \).
- This inequality continues to hold because \( \text{distTo}[v_i] \) cannot increase.
- Immediately after relaxing edge \( e_{i+1} \) in pass \( i+1 \), we have
  
  \[
  \text{distTo}[v_{i+1}] \leq \text{distTo}[v_i] + \text{weight}(e_{i+1}) \leq \text{weight}(e_1) + \text{weight}(e_2) + \cdots + \text{weight}(e_i) + \text{weight}(e_{i+1}).
  \]
- This inequality continues to hold because \( \text{distTo}[v_{i+1}] \) cannot increase. ■
Proposition. Let $s = v_0 \to v_1 \to \ldots \to v_k = v$ be any path from $s$ to $v$ containing $k$ edges. Then, after pass $k$, $\text{distTo}[v_k] \leq \text{weight}(e_1) + \text{weight}(e_2) + \ldots + \text{weight}(e_k)$.

Corollary. Bellman–Ford computes shortest path distances.

**Pf.** [apply Proposition to a shortest path from $s$ to $v$]

- There exists a simple shortest path $P^*$ from $s$ to $v$; it contains $k \leq V - 1$ edges.
- The Proposition implies that, after at most $V - 1$ passes, $\text{distTo}[v] \leq \text{length}(P^*)$.
- Since $\text{distTo}[v]$ is the length of some path from $s$ to $v$, $\text{distTo}[v] = \text{length}(P^*)$. ■
Bellman–Ford algorithm: practical improvement

Observation. If \( \text{distTo}[v] \) does not change during pass \( i \), not necessary to relax any edges incident from \( v \) in pass \( i + 1 \).

Queue–based implementation of Bellman–Ford.

- Perform vertex relaxations. \( \xrightarrow{\text{relax vertex } v} \) relax vertex \( v = \text{relax all edges incident from } v \)
- Maintain queue of vertices whose \( \text{distTo}[] \) values changed since it was last relaxed.

Impact.
- In the worst case, the running time is still \( \Theta(E V) \).
- But much faster in practice on typical inputs.
**Problem.** Given a digraph $G$ with positive edge weights and vertex $s$, find a longest simple path from $s$ to every other vertex.

**Goal.** Design algorithm that takes $\Theta(EV)$ time in the worst case.

longest simple path from 0 to 4: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
Bellman–Ford algorithm: negative weights

**Remark.** The Bellman–Ford algorithm works even if some weights are negative, provided there are no negative cycles.

**Negative cycle.** A directed cycle whose length is negative.

![Diagram of a negative cycle](image)

Length of negative cycle = 1 + 2 + 3 + -8 = -2

**Negative cycles and shortest paths.** Length of path can be made arbitrarily negative by using negative cycle.

0 → 1 → 2 → 3 → 4 → 1 → … → 2 → 3 → 4 → 1 → 2 → 5
4.4 Shortest Paths

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm
"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

-- Edsger Dijkstra
Dijkstra's algorithm

For each vertex v: distTo[v] = ∞.
For each vertex v: edgeTo[v] = null.

T = ∅.
distTo[s] = 0.

Repeat until all vertices are marked:
- Select unmarked vertex v with the smallest distTo[] value.
- Mark v.
- Relax each edge incident from v.

Key difference with Bellman-Ford. Each edge gets relaxed exactly once!
Dijkstra’s algorithm demo

Repeat until all vertices are marked:

- Select unmarked vertex \( v \) with the smallest \( \text{distTo}[] \) value.
- Mark \( v \) and relax all edges incident from \( v \).

an edge-weighted digraph

\[
\begin{align*}
0 &\rightarrow 1 & 5.0 \\
0 &\rightarrow 4 & 9.0 \\
0 &\rightarrow 7 & 8.0 \\
1 &\rightarrow 2 & 12.0 \\
1 &\rightarrow 3 & 15.0 \\
1 &\rightarrow 7 & 4.0 \\
2 &\rightarrow 3 & 3.0 \\
2 &\rightarrow 6 & 11.0 \\
3 &\rightarrow 6 & 9.0 \\
4 &\rightarrow 5 & 4.0 \\
4 &\rightarrow 6 & 20.0 \\
4 &\rightarrow 7 & 5.0 \\
5 &\rightarrow 2 & 1.0 \\
5 &\rightarrow 6 & 13.0 \\
7 &\rightarrow 5 & 6.0 \\
7 &\rightarrow 2 & 7.0 \\
\end{align*}
\]
Dijkstra’s algorithm demo

Repeat until all vertices are marked:

- Select unmarked vertex $v$ with the smallest $\text{distTo}[]$ value.
- Mark $v$ and relax all edges incident from $v$.

<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>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>

shortest-paths tree from vertex $s$
Dijkstra’s algorithm: correctness proof

**Invariant.** For each marked vertex $v$: $\text{distTo}[v] = d^*(v)$.

**Pf.** [by induction on number of marked vertices]

- Let $v$ be next vertex marked.
- Let $P$ be the path from $s$ to $v$ of length $\text{distTo}[v]$.
- Consider any other path $P'$ from $s$ to $v$.
- Let $x \rightarrow y$ be first edge in $P'$ with $x$ marked and $y$ unmarked.
- $P'$ is already as long as $P$ by the time it reaches $y$:

$$\text{length}(P') \leq \text{length}(P)$$

- $P'$ is a path from $s$ to $x$, followed by edge $x \rightarrow y$, followed by non-negative edges

$$\text{length}(P') \leq \text{distTo}[v] \leq \text{distTo}[y] \leq \text{distTo}[x] + \text{weight}(x, y) \leq \text{length}(P')$$

Dijkstra chose $v$ instead of $y$ by construction.
Dijkstra’s algorithm: correctness proof

Invariant. For each marked vertex \( v \): \( \text{distTo}[v] = d^*(v) \).

Corollary 1. Dijkstra’s algorithm computes shortest path distances.

Corollary 2. Dijkstra’s algorithm relaxes vertices in increasing order of distance from \( s \).
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);
            }
    }
}
Dijkstra’s algorithm: Java implementation

When relaxing an edge, also update PQ:

- Found first path from $s$ to $w$: add $w$ to PQ.
- Found better path from $s$ to $w$: decrease key of $w$ in PQ.

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

Q. How to implement DECREASE-KEY operation in a priority queue?
Indexed priority queue (Section 2.4)

Associate an index between 0 and $n - 1$ with each key in a priority queue.

- Insert a key associated with a given index.
- Delete a minimum key and return associated index.
- Decrease the key associated with a given index.

```java
public class IndexMinPQ<Key extends Comparable<Key>>

    IndexMinPQ(int n)  // create PQ with indices 0, 1, ..., n – 1
    void insert(int i, Key key)  // associate key with index i
    int delMin()  // remove min key and return associated index
    void decreaseKey(int i, Key key)  // decrease the key associated with index i
    boolean isEmpty()  // is the priority queue empty?

    :
```

for Dijkstra's algorithm:

$n = V$,
index = vertex,
key = distance from $s$
Goal. Implement **DECREASE–KEY** operation in a binary heap.
Goal. Implement \textsc{Decrease-Key} operation in a binary heap.

Solution.

- Find vertex in heap. How?
- Change priority of vertex and call \texttt{swim()} to restore heap invariant.

Extra data structure. Maintain an inverse array $qp[]$ that maps from the vertex to the binary heap node index.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$v_3$</td>
<td>$v_5$</td>
<td>$v_7$</td>
<td>$v_2$</td>
<td>$v_0$</td>
<td>$v_4$</td>
<td>$v_6$</td>
<td>$v_1$</td>
</tr>
</tbody>
</table>

vertex 2 has priority 3.0 and is at heap index 4
```
Dijkstra’s algorithm: which priority queue?

Number of PQ operations: \( V \text{ INSERT}, V \text{ DELETE-MIN}, \leq E \text{ DECREASE-KEY}. \)

<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_{E/V} 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>

\( ^\dagger \) amortized

**Bottom line.**
- Array implementation optimal for complete digraphs.
- Binary heap much faster for sparse digraphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.
Priority-first search

**Observation.** Prim and Dijkstra are essentially the same algorithm.

- **Prim:** Choose next vertex that is closest to *any vertex in the tree* (via an undirected edge).
- **Dijkstra:** Choose next vertex that is closest to the *source vertex* (via a directed path).

![Prim's algorithm](image1.png)  ![Dijkstra's algorithm](image2.png)
Variations on a theme: vertex relaxations.

- Bellman–Ford: relax all vertices; repeat $V - 1$ times.
- Dijkstra: relax vertices in order of distance from $s$.
- Topological sort: relax vertices in topological order.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case running time</th>
<th>negative weights †</th>
<th>directed cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman–Ford</td>
<td>$E V$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>$E \log V$</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>topological sort</td>
<td>$E$</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

† no negative cycles
Which shortest paths algorithm to use?

Select algorithm based on properties of edge-weighted digraph.

- Negative weights (but no “negative cycles”): Bellman–Ford.
- Non-negative weights: Dijkstra.
- DAG: topological sort.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case running time</th>
<th>negative weights †</th>
<th>directed cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman–Ford</td>
<td>$E V$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>$E \log V$</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>topological sort</td>
<td>$E$</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

† no negative cycles