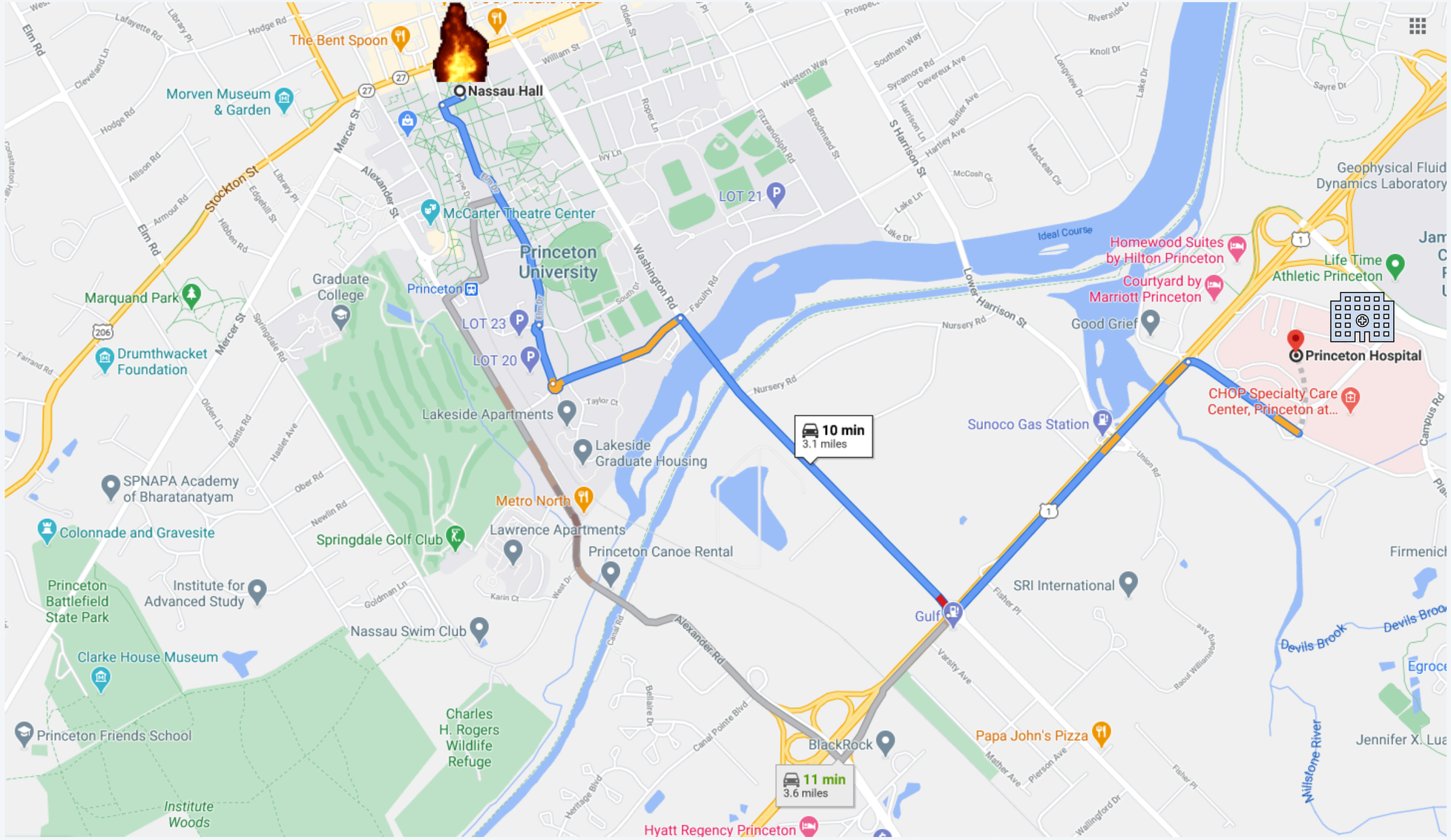
https://algs4.cs.princeton.edu

4.4 Shortest Paths

- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

A motivating example: route planning in a road network



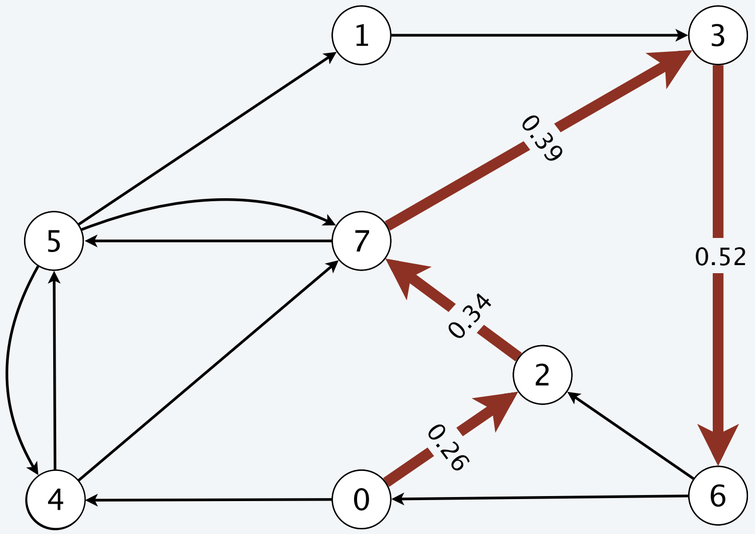
Shortest path in an edge-weighted digraph

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

length of path = sum of edge weights

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 variants

Which vertices?

- Source–destination: from one vertex s to another vertex t.
- Single source: from one vertex s to every other vertex.
- Single destination: from every vertex to one vertex t.
- All pairs: from every vertex to every other vertex.

Restrictions on edge weights?

- Non-negative weights. ← *we assume this in today's lecture (except as noted)*
- Euclidean weights.
- Arbitrary weights.

Directed cycles?

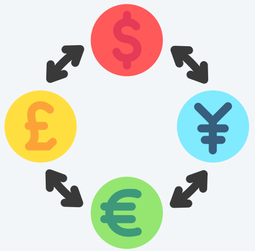
- Prohibit. ← *can derive faster algorithms in DAGs (see next lecture)*
- Allow.

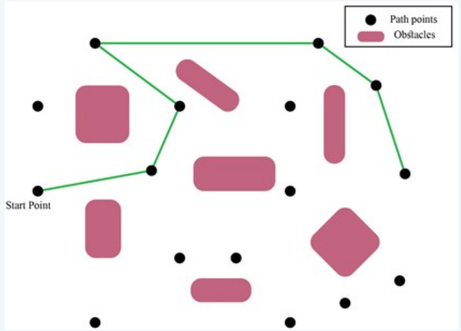
Simplifying assumption.

Each vertex is reachable from s. ← *ensures that shortest path from s to v exists (and that $E \geq V - 1$)*

Shortest path applications

- Seam carving. ⟵ *see Assignment 6*
- Texture mapping.
- Typesetting in TEX.
- Currency exchange.
- Urban traffic planning.
- Robot motion planinng.
- Social network analysis.
- Telecommunication routing.
- Project scheduling (PERT/CPM).
- Optimal pipelining of VLSI chip.
- Packet routing (OSPF, BGP, RIP, IS-IS).
- Route planning (Google maps, car navigation, ...).





Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest paths: poll 1

Which shortest path variant for route planning by a car navigation system?

Hint: drivers make wrong turns occasionally.

A. Source–destination: from one vertex s to another vertex t.

B. Single source: from one vertex s to every vertex.

C. Single destination: from every vertex to one vertex t.

D. All pairs: from every vertex to every other vertex.

https://algs4.cs.princeton.edu

4.4 Shortest Paths

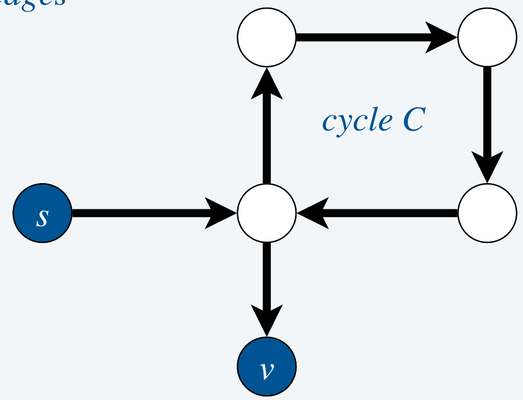
- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Shortest path properties

Simple paths. There exists a shortest $s \rightsquigarrow v$ path that has no repeated vertices. $\Longrightarrow$ $\leq V - 1$ *edges*

Pf. [by contradiction]

- Let P^* be a shortest $s \rightsquigarrow v$ path with the fewest edges, and assume it has a repeated vertex.
- Remove the cycle C from P^*, yielding a shortest path with fewer edges.

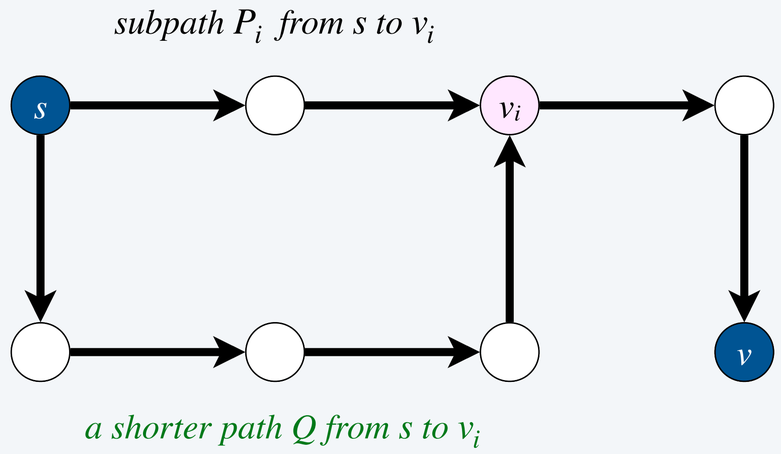


Optimal substructure. Let P^* be a shortest $s \rightsquigarrow v$ path.

Then, any subpath P_i of P^* from s to v_i is a shortest $s \rightsquigarrow v_i$ path.

Pf. [by contradiction]

- Suppose subpath P_i were not a shortest $s \rightsquigarrow v_i$ path.
- Let Q be a shortest $s \rightsquigarrow v_i$ path.
- Replace subpath P_i with Q in P^*, yielding a shorter $s \rightsquigarrow v$ path.



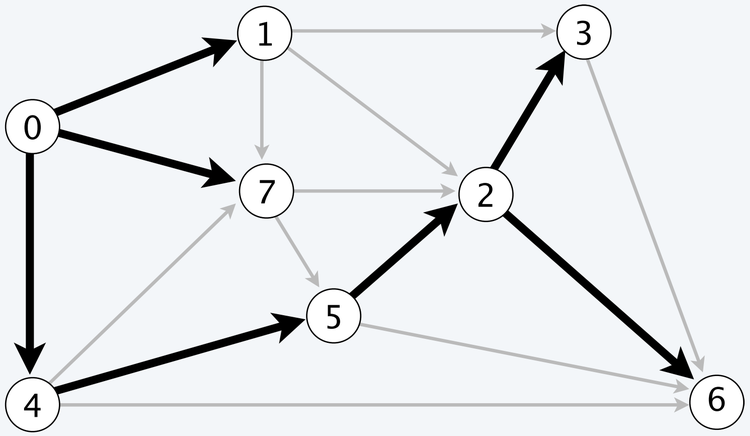
Data structures for single-source shortest paths

Goal. Find a shortest path from s to every other vertex.

Observation 2. A shortest-paths tree solution exists. Why?

Consequence. Can represent shortest paths with two parallel arrays:

- `distTo[v]`: length of a shortest $s \rightsquigarrow v$ path.
- `edgeTo[v]`: last edge on a shortest $s \rightsquigarrow v$ path.



shortest-paths tree from 0

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

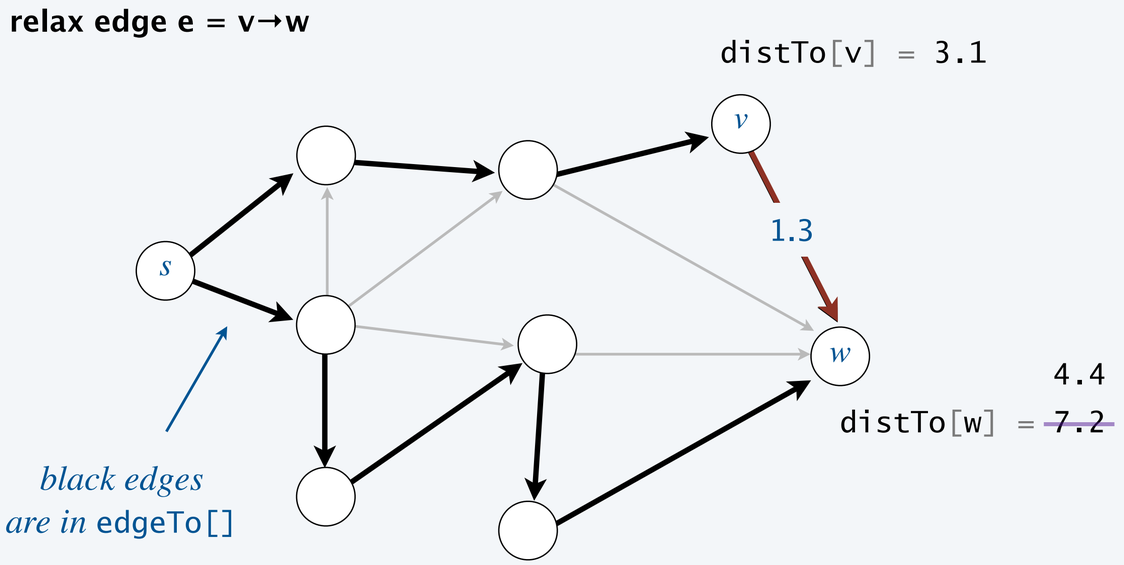
parent-link representation

Edge relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]`: length of shortest known $s \rightsquigarrow v$ path.
- `distTo[w]`: length of shortest known $s \rightsquigarrow w$ path.
- `edgeTo[w]`: last edge on shortest known $s \rightsquigarrow w$ path.
- If edge e yields a shorter $s \rightsquigarrow w$ path via v, then update:
  - `distTo[w] = distTo[v] + e.weight()`
  - `edgeTo[w] = e`

context: shortest-path algorithms maintain champion distances and paths (updated through edge relaxations)



Shortest paths: poll 2

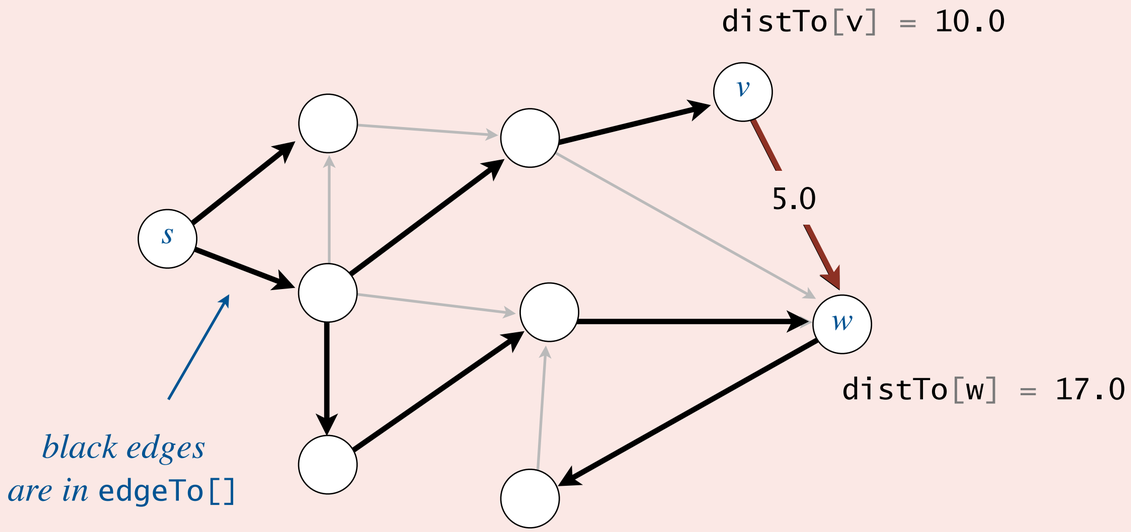
What are the values of `distTo[v]` and `distTo[w]` immediately after relaxing edge $e = v \rightarrow w$?

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: distTo[v] = ∞.

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

distTo[s] = 0.

Repeat until distTo[v] values converge:

- **Relax any edge.**

Key properties. Throughout the generic algorithm:

- `distTo[v]` is either ∞ or the length of a (simple) $s \rightsquigarrow v$ path.
- `distTo[v]` never increases; it decreases when a shorter $s \rightsquigarrow v$ path is found.
- `edgeTo[]` defines a shortest-paths tree rooted at s.

Framework for shortest-paths algorithm

Generic algorithm (to compute a SPT from s)

For each vertex v: distTo[v] = ∞.

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

distTo[s] = 0.

Repeat until 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. ⟵ *next lecture*

https://algs4.cs.princeton.edu

4.4 Shortest Paths

- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Weighted directed edge API

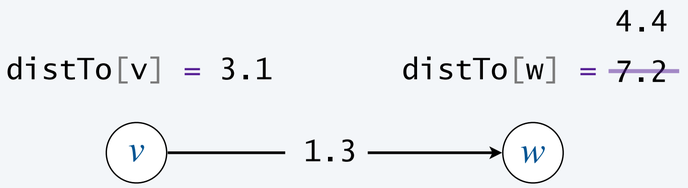
API.

`public class DirectedEdge`

	`DirectedEdge(int v, int w, double weight)`	*create weighted edge v→w*
`int`	`from()`	*tail vertex v*
`int`	`to()`	*head vertex w*
`double`	`weight()`	*weight of edge v→w*
	⋮	⋮

Ex. Relax 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: Java implementation

```
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.

`public class EdgeWeightedDigraph`		
	`EdgeWeightedDigraph(int V)`	*edge-weighted digraph with V vertices (and no edges)*
`void`	`addEdge(DirectedEdge e)`	*add edge e*
`Iterable<DirectedEdge>`	`adj(int v)`	*edges incident from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
	⋮	⋮

Edge-weighted digraph: adjacency-lists implementation in Java

Implementation. Almost identical to `EdgeWeightedGraph`.

```
public class EdgeWeightedDigraph {
   private final int V;
   private final Queue<DirectedEdge>[] adj;

   public EdgeWeightedDigraph(int V) {
     this.V = V;
     adj = (Queue<Edge>[]) new Queue[V];
     for (int v = 0; v < V; v++)
        adj[v] = new Queue<>();
   }

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

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

add edge e = v→w to only v's adjacency list

Single-source shortest paths API

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

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

https://algs4.cs.princeton.edu

4.4 Shortest Paths

- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Bellman–Ford algorithm

Bellman–Ford algorithm

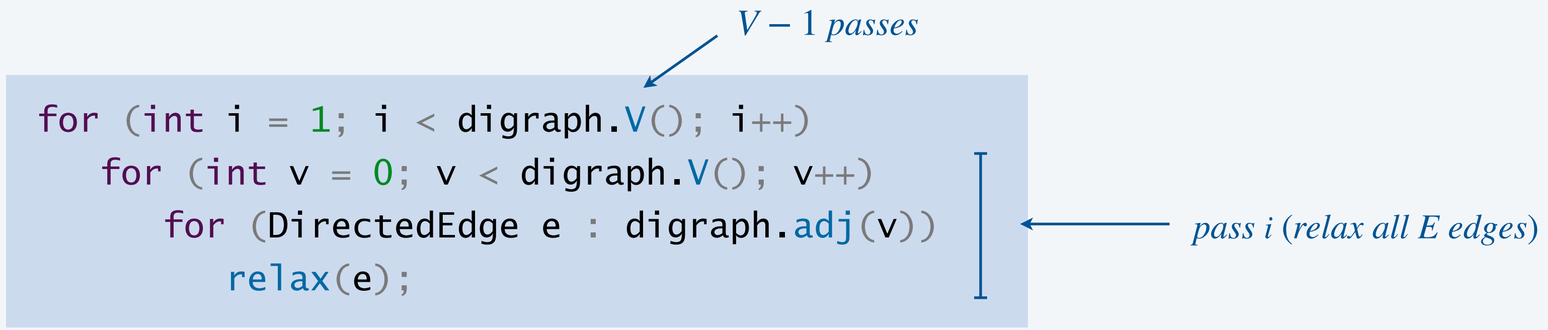
For each vertex v: distTo[v] = ∞.

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

distTo[s] = 0.

Repeat V–1 times:

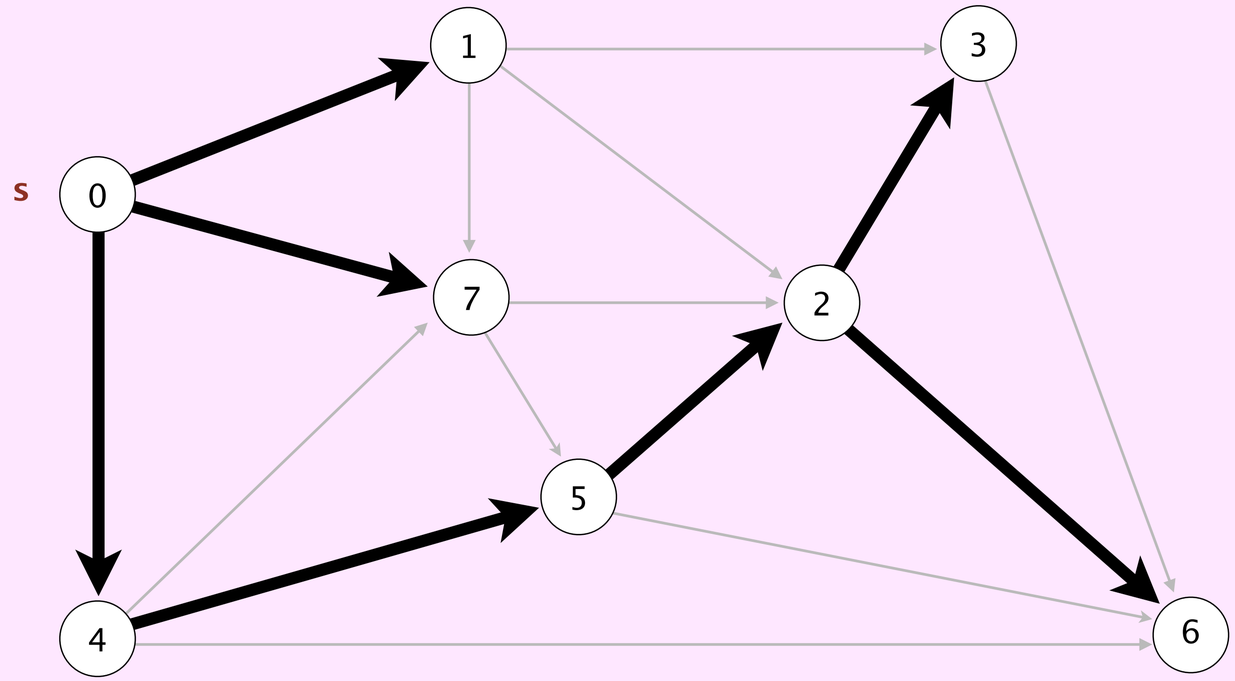
– Relax all E edges.



Performance. Algorithm takes $\Theta(E\,V)$ time and uses $\Theta(V)$ extra space.

Bellman–Ford algorithm demo

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

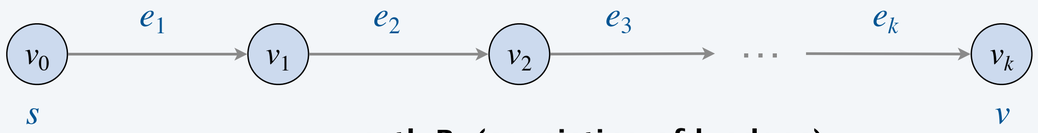


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Bellman–Ford algorithm: proof of correctness

Notation. Let $P^* = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$ be a shortest path of k edges from $s = v_0$ to $v = v_k$, and let $P_i = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i$ be the subpath consisting of the first i edges in P^*.



Proposition. After pass i of Bellman–Ford, $\texttt{distTo}[v_i] = length(P_i)$. ← *after pass i, have already found a shortest $s \rightsquigarrow v_i$ path*

Pf. [by induction on number of passes i]

- Base case: initially, $\texttt{distTo}[v_0] = 0 = length(P_0)$.
- Inductive hypothesis: after pass i, $\texttt{distTo}[v_i] = length(P_i)$.
- Immediately after relaxing edge e_{i+1} in pass $i+1$, we have

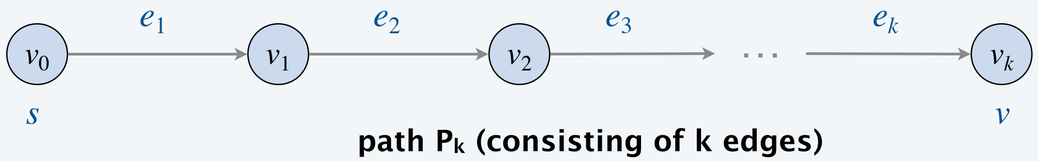
$$\begin{aligned} \texttt{distTo}[v_{i+1}] &\leq \texttt{distTo}[v_i] + weight(e_{i+1}) && \leftarrow \textit{edge relaxation} \\ &= length(P_i) + weight(e_{i+1}) && \leftarrow \textit{inductive hypothesis} \\ &= length(P_{i+1}) \end{aligned}$$

and remains that way for the rest of algorithm ↓

- Since $\texttt{distTo}[v_{i+1}]$ is the length of some $s \rightsquigarrow v_{i+1}$ path, we must have $\texttt{distTo}[v_{i+1}] = length(P_{i+1})$ ▪

Bellman–Ford algorithm: proof of correctness

Notation. Let $P^* = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$ be a shortest path of k edges from $s = v_0$ to $v = v_k$, and let $P_i = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i$ be the subpath consisting of the first i edges in P^*.



Proposition. After pass i of Bellman–Ford, `distTo[`v_i`]` $= length(P_i)$. ⟵ *after pass i, have already found a shortest $s \rightsquigarrow v_i$ path*

Corollary. For each vertex v, Bellman–Ford computes the length of a shortest $s \rightsquigarrow v$ path.

Pf.

- There exists a shortest $s \rightsquigarrow v$ path P^* that is simple. So, P^* contains $k \leq V - 1$ edges.
- From the Proposition, after k passes, `distTo[`v_k`]` $= length(P_k)$. ▪

$v = v_k$ $\qquad$ $P^* = P_k$

Bellman–Ford algorithm: practical improvement

Observation. If `distTo[v]` does not change during pass i,
not necessary to relax any edges incident from v in pass $i + 1$.

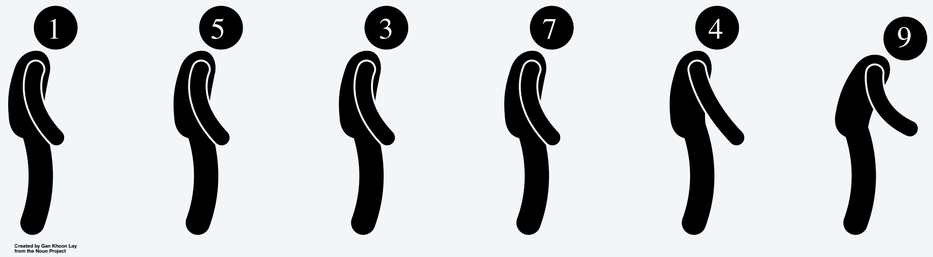
```
private void relax(int v) {
   for (DirectedEdge e : digraph.adj(v))
      relax(e);
}
```

vertex relaxation

Queue-based implementation of Bellman–Ford.

- Perform vertex relaxations. ⟵ *relax vertex v = relax all edges incident from v*
- Maintain queue of vertices whose `distTo[]` values changed since it was last relaxed.

must ensure each vertex is on queue at most once
(or exponential blowup!)



relax in pass $i + 1$

relax in pass i

relax vertex v

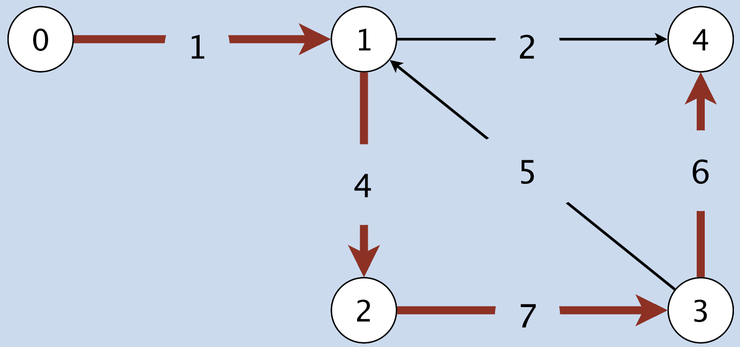
Impact.

- In the worst case, the running time is still $\Theta(E V)$.
- But much faster in practice on typical inputs.

Longest path

Problem. Given a digraph G with positive edge weights and source vertex s, find a longest simple path from s to every other vertex.

Goal. Design an algorithm that takes $O(EV)$ time in the worst case.



longest simple path from 0 to 4

0 → 1 → 2 → 3 → 4

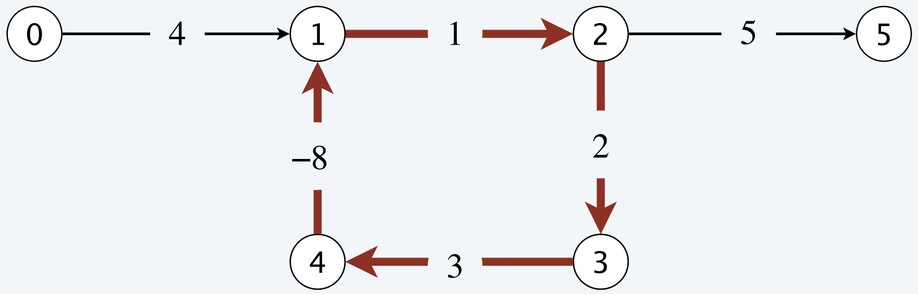
length of path = 18

(1 + 4 + 7 + 6)

Bellman–Ford algorithm: negative weights

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

Def. A negative cycle is a directed cycle whose total length is negative.



negative cycle
(length = 1 + 2 +3 + −8 = −2 < 0)

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

$$0 \to 1 \to \underline{2 \to 3 \to 4 \to 1} \to \quad \cdots \quad \to \underline{2 \to 3 \to 4 \to 1} \to 2 \to 5$$

https://algs4.cs.princeton.edu

4.4 Shortest Paths

- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Edsger Dijkstra: select quote

Dijkstra's algorithm

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:

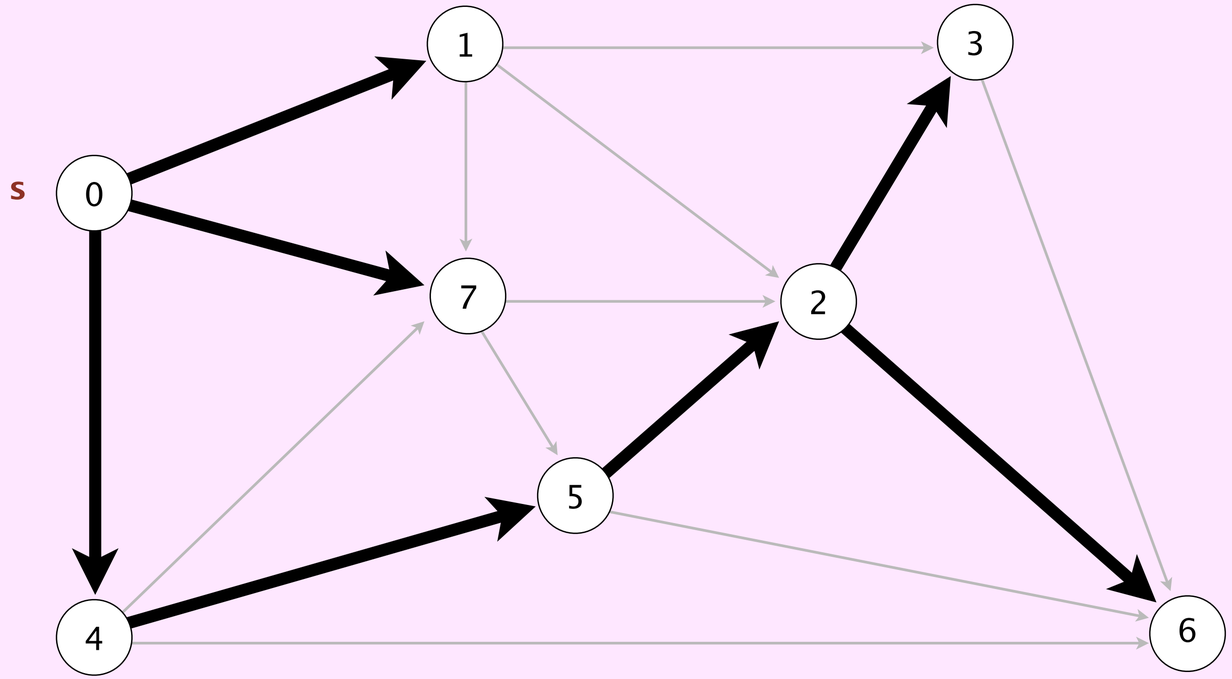
- **Select unmarked vertex v with the smallest distTo[] value.**
- **Mark v.**
- **Relax each edge incident from v.**

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 `distTo[]` value.
- Mark v and relax all edges incident from v.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

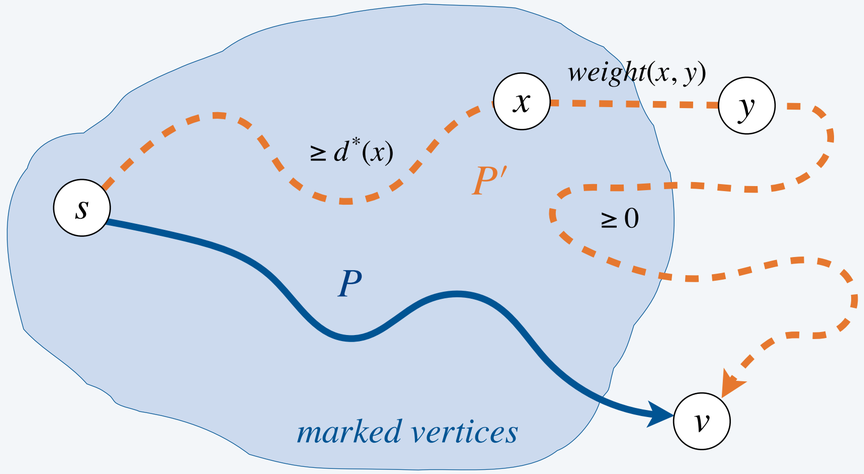
Dijkstra's algorithm: proof of correctness

Invariant. For each marked vertex v : `distTo[v]` $= d^*(v)$.

notation for length of shortest $s \rightsquigarrow v$ path

Pf. [by induction on number of marked vertices]

- Let v be next vertex marked.
- Let P be the $s \rightsquigarrow v$ path of length `distTo[v]`.
- Consider any other $s \rightsquigarrow v$ path P'.
- 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 :



by definition

$$length(P) = \texttt{distTo[v]}$$

Dijkstra chose v instead of y $\longrightarrow$ $\leq$ `distTo[y]`

vertex x is marked (so it was relaxed) $\longrightarrow$ $\leq$ `distTo[x]` $+ weight(x, y)$

induction $\longrightarrow$ $= d^*(x) + weight(x, y)$

P′ is a path from s to x, followed by edge x→y, followed by non-negative edges $\longrightarrow$ $\leq length(P')$ ▪

Dijkstra's algorithm: proof of correctness

Invariant. For each marked vertex v : `distTo[v]` $= d^*(v)$.

notation for length of shortest $s \rightsquigarrow v$ path

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

Corollary 2. Dijkstra's algorithm relaxes vertices in increasing order of distance from s.

generalizes both
level-order traversal in a tree
and breadth-first search in a graph

Dijkstra's algorithm: Java implementation

```
public class DijkstraSP {
   private DirectedEdge[] edgeTo;
   private double[] distTo;

   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph digraph, int s) {
      edgeTo = new DirectedEdge[digraph.V()];
      distTo = new double[digraph.V()];

      pq = new IndexMinPQ<Double>(digraph.V());

      for (int v = 0; v < digraph.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 : digraph.adj(v))
            relax(e);
      }
   }
}
```

PQ that supports decreasing the key (stay tuned)

PQ contains the unmarked vertices with finite `distTo[]` *values*

relax vertices in increasing order of distance from s

Dijkstra's algorithm: Java implementation

When relaxing an edge $e = v \rightarrow w$, also update PQ:

- Found first $s \rightsquigarrow w$ path : add vertex w to PQ.
- Found better $s \rightsquigarrow w$ path: decrease priority associated with vertex w in PQ.

```
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]);
   }
}
```

update PQ

index *priority*

Q. How to efficiently 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.

for Dijkstra's algorithm:
$n = V$,
index = vertex,
key = distance from s

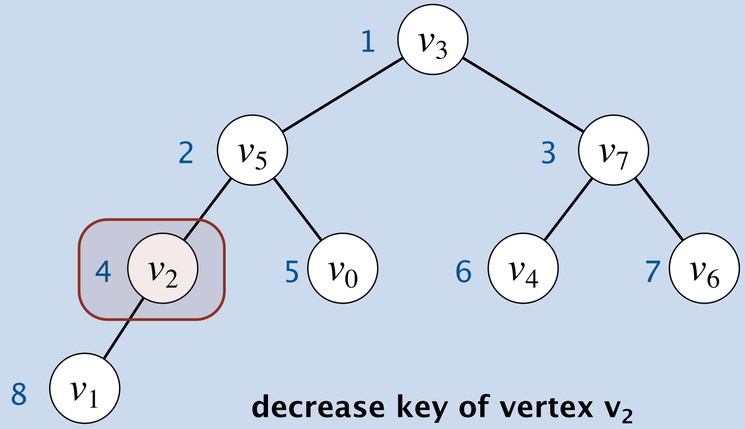
`public class IndexMinPQ<Key extends Comparable<Key>>`

	`IndexMinPQ(int n)`	*create PQ with indices $0, 1, \ldots, n - 1$*
`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`	`contains(int i)`	*does the priority queue contain index i ?*
`boolean`	`isEmpty()`	*is the priority queue empty ?*
	⋮	⋮

Decreasing the key in a binary heap

Goal. Implement DECREASE-KEY operation in a min-oriented binary heap.

	0	1	2	3	4	5	6	7	8
pq[]	–	v_3	v_5	v_7	v_2	v_0	v_4	v_6	v_1



decrease key of vertex v_2

Decreasing the key in a binary heap

Goal. Implement DECREASE-KEY operation in a min-oriented binary heap.

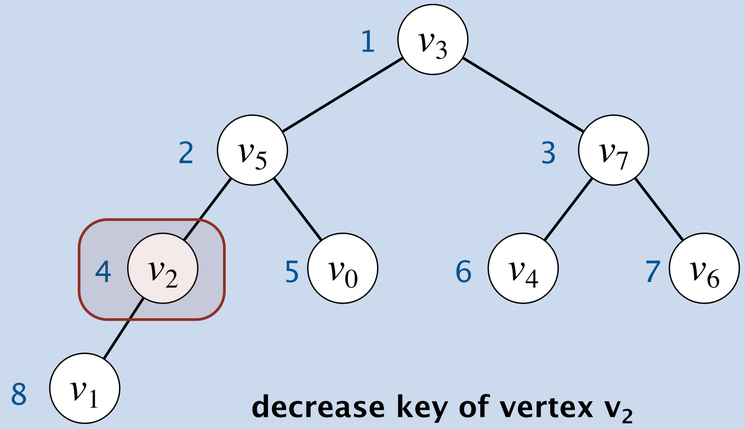
Solution.

- Find vertex in binary heap. How?
- Change priority of vertex and call `swim()` to restore heap invariant.

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

	0	1	2	3	4	5	6	7	8
pq[]	–	v_3	v_5	v_7	v_2	v_0	v_4	v_6	v_1
qp[]	5	8	4	1	6	2	4	3	–
keys[]	1.0	2.0	3.0	0.0	6.0	8.0	4.0	2.0	–

vertex 2 has priority 3.0 and is at heap index 4



decrease key of vertex v_2

Dijkstra's algorithm: which priority queue?

Number of PQ operations: V INSERT, V DELETE-MIN, $\leq E$ DECREASE-KEY.

PQ implementation	INSERT	DELETE-MIN	DECREASE-KEY	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1 †	$\log V$ †	1 †	$E + V \log V$

† *amortized*

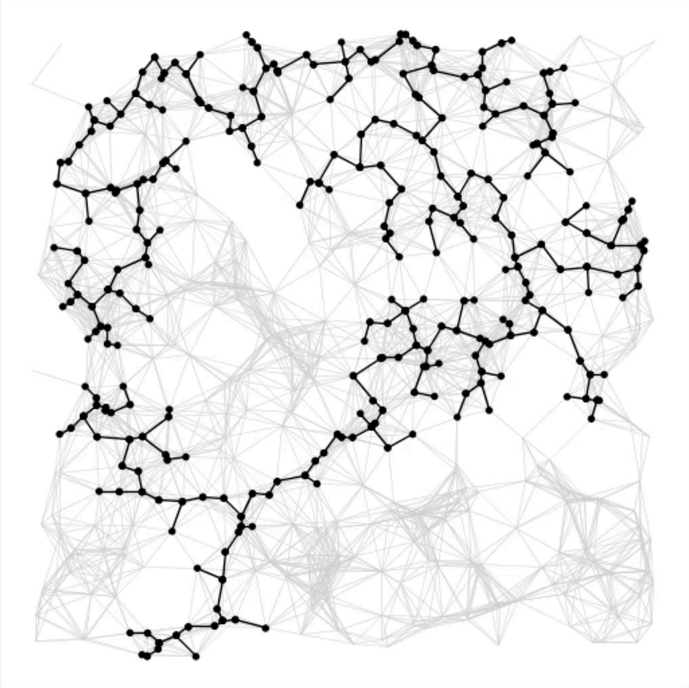
Bottom line.

- 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 probably not worth implementing.

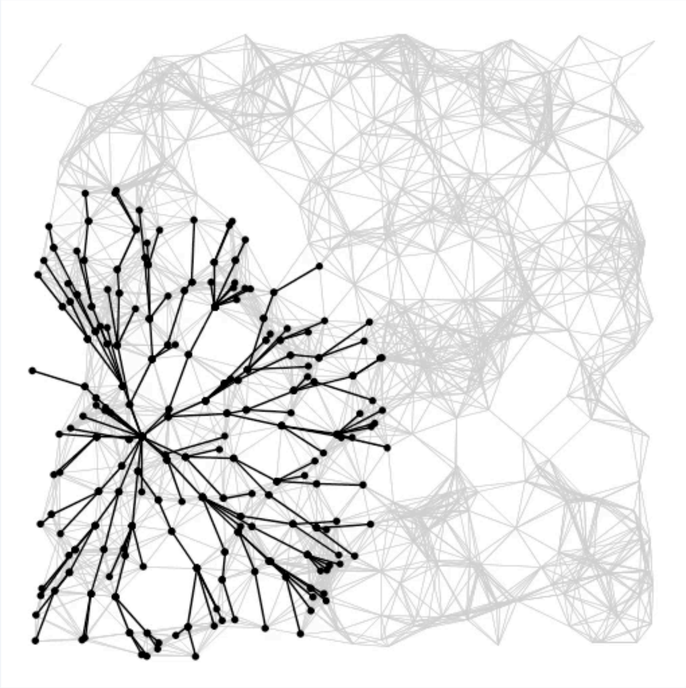
Priority-first search

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



Dijkstra's algorithm

Algorithms for shortest paths

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. ⟵ *see Section 4.4 and next lecture*

algorithm	worst-case running time	negative weights †	directed cycles
Bellman–Ford	$\Theta(E\ V)$	✔	✔
Dijkstra	$\Theta(E \log V)$		✔
topological sort	E	✔	

† *no negative cycles*

Which shortest paths algorithm to use?

Select algorithm based on properties of edge-weighted digraph.

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

algorithm	worst-case running time	negative weights †	directed cycles
Bellman-Ford	$\Theta(E\ V)$	✔	✔
Dijkstra	$\Theta(E \log V)$		✔
topological sort	E	✔	

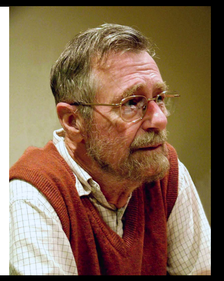
† *no negative cycles*

Credits

image	source	license
Map of Princeton, N.J.	Google Maps	
Broadway Tower	Wikimedia	CC BY 2.5
Car GPS	Adobe Stock	education license
Queue for Registration	Noun Project	CC BY 3.0
Dijkstra T-shirt	Zazzle	
Edsger Dijkstra	Wikimedia	CC BY-SA 3.0

A final thought

> *Do only what only you can do.*
>
> — Edsger W. Dijkstra



A final thought