Shortest Path Algorithms

❖ Shortest Path

Path = sequence of edges in graph G

- $p = (v_0,v_1,weight_1), (v_1,v_2,weight_2), \ldots, (v_{m-1},v_m,weight_m)$

cost(path) = sum of edge weights along path

Shortest path between vertices *s* and *t*

- a simple path *p(s,t)* where *s = first(p), t = last(p)*
- no other simple path *q(s,t)* has *cost(q) < cost(p)*

Assumptions: weighted digraph, no negative weights.

Applications: navigation, routing in data networks, ...

❖ ... Shortest Path

Some variations on shortest path (SP) ...

Source-target SP problem

- shortest path from source vertex s to target vertex t

Single-source SP problem

- set of shortest paths from source vertex s to all other vertices

All-pairs SP problems

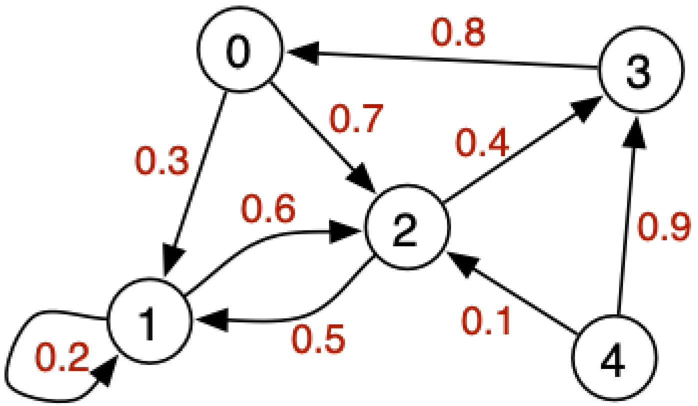
- set of shortest paths between all pairs of vertices s and t

❖ Single-source Shortest Path (SSSP)

Shortest paths from *s* to all other vertices

- **`dist[]`** *V*-indexed array of cost of shortest path from *s*
- **`pred[]`** *V*-indexed array of predecessor in shortest path from *s*

Example:



	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

Shortest paths from s=0

	0	1	2	3	4
dist	1.3	0.6	0.1	0.5	0
pred	3	2	4	2	-

Shortest paths from s=4

❖ Edge Relaxation

Assume: **`dist[]`** and **`pred[]`** as above

- but containing data for shortest paths *discovered so far*

If we have ...

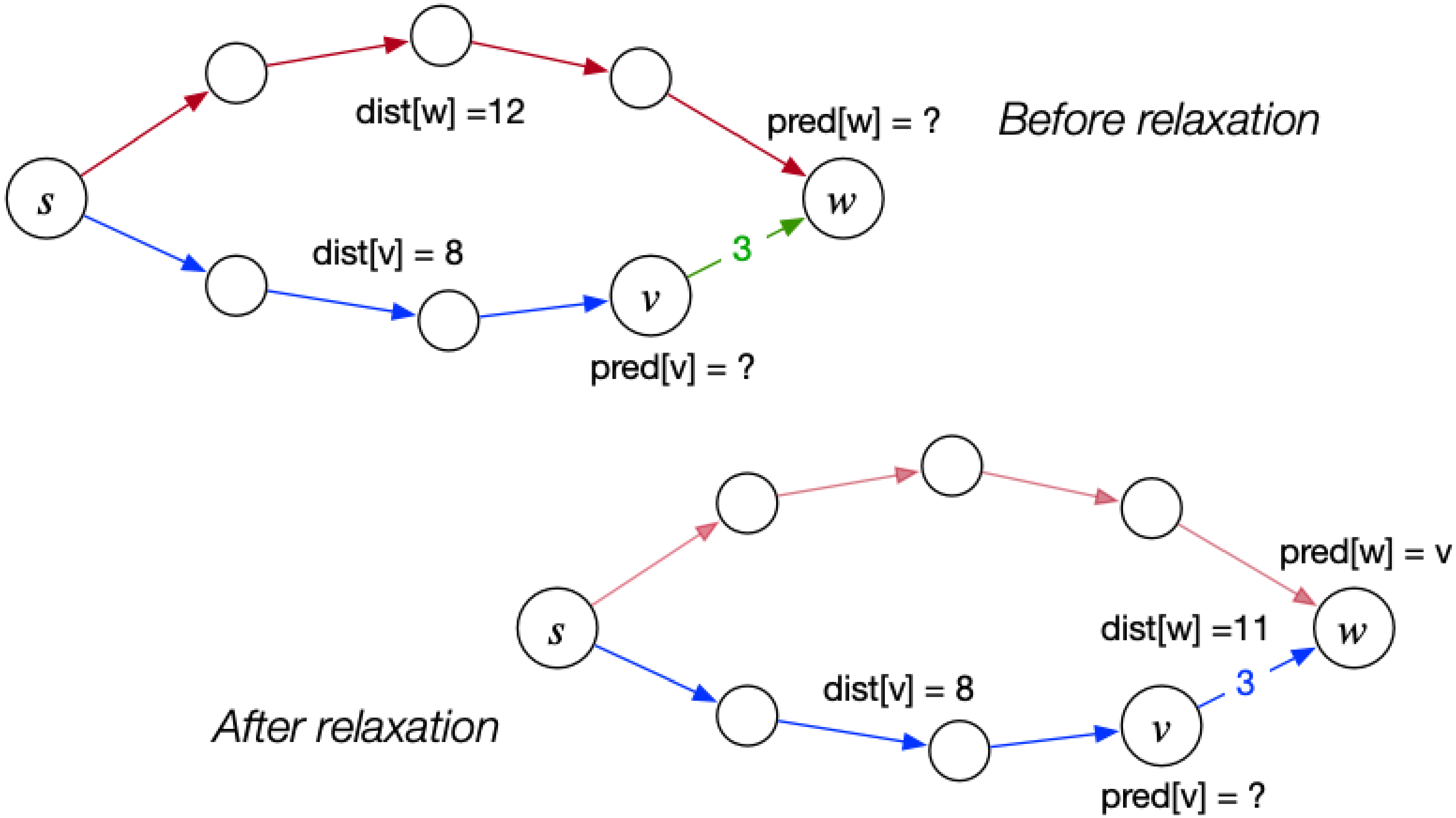
- **`dist[v]`** is length of shortest known path from *s* to *v*
- **`dist[w]`** is length of shortest known path from *s* to *w*
- edge *(v,w,weight)*

Relaxation updates data for *w* if we find a shorter path from *s* to *w*:

- if **`dist[v]+weight < dist[w]`** then
  update **`dist[w]←dist[v]+weight`** and **`pred[w]←v`**

❖ ... Edge Relaxation

Relaxation along edge *e = (v,w,3)* :



❖ Dijkstra's Algorithm

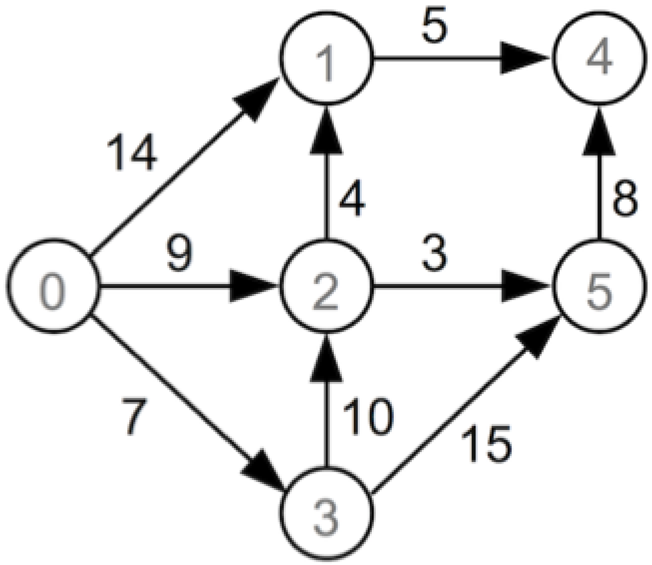
One approach to solving single-source shortest path ...

```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path from s
vSet    // vertices whose shortest path from s is unknown

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise all dist[] to ∞
|  dist[source]=0
|  initialise all pred[] to -1
|  vSet=all vertices of G
|  while vSet ≠ ∅ do
|  |  find v ∈ vSet with minimum dist[v]
|  |  for each (v,w,weight) ∈ edges(G) do
|  |     relax along (v,w,weight)
|  |  end for
|  |  vSet=vSet \ {v}
|  end while
```

❖ Tracing Dijkstra's Algorithm

How Dijkstra's algorithm runs when source = 0:



Initially

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	inf	inf	inf	inf	inf
pred	-	-	-	-	-	-

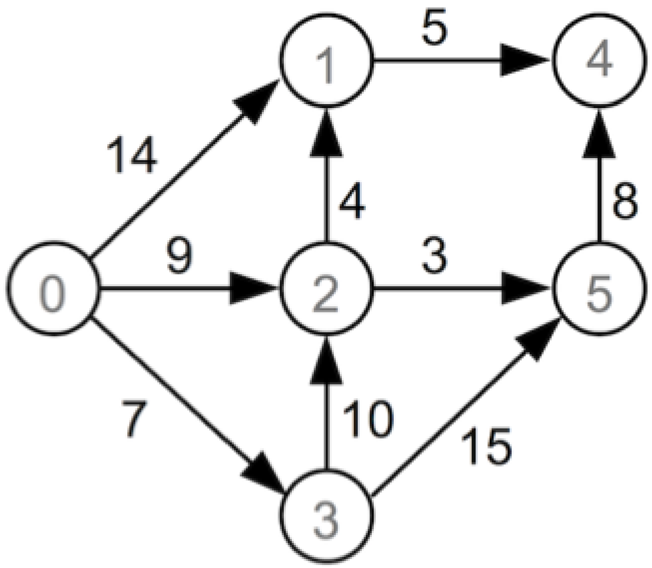
First iteration, v=0

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	7	inf	int
pred	-	0	0	0	-	-

Second Iteration, v=3

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	7	inf	22
pred	-	0	0	0	-	3

```
while vSet not empty do
   find v in vSet
     with min dist[v]
   for each (v,w,weight) in E do
      relax along (v,w,weight)
   end for
   vSet = vSet \ {v}
end while
```




```
while vSet not empty do
  find v in vSet
    with min dist[v]
  for each (v,w,weight) in E do
     relax along (v,w,weight)
  end for
  vSet = vSet \ {v}
end while
```

Third iteration, v=2

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	inf	12
pred	-	2	0	0	-	2

Fourth iteration, v=5

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	20	12
pred	-	2	0	0	5	2

Fifth iteration, v=1

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Sixth iteration, v=4

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Completed, vSet is empty

❖ Analysis of Dijkstra's Algorithm

Why Dijkstra's algorithm is correct ...

Hypothesis:

(a) for visited *s*, dist[*s*] is shortest distance from source

(b) for unvisited *t*, dist[*t*] is shortest distance from source *via visited nodes*

Ultimately, all nodes are visited, so ...

- $\forall\ v$, dist[v] is shortest distance from source

❖ ... Analysis of Dijkstra's Algorithm

Proof:

Base case: no visited nodes, *dist[source]=0*, *dist[s]=∞* for all other nodes

Induction step:

1. If *s* is unvisited node with minimum *dist[s]*, then *dist[s]* is shortest distance from source to *s*:
    - if ∃ shorter path via only visited nodes, then *dist[s]* would have been updated when processing the predecessor of *s* on this path
    - if ∃ shorter path via an unvisited node *u*, then *dist[u]<dist[s]*, which is impossible if *s* has min distance of all unvisited nodes
2. This implies that (a) holds for *s* after processing *s*
3. (b) still holds for all unvisited nodes *t* after processing *s*:
    - if ∃ shorter path via *s* we would have just updated *dist[t]*
    - if ∃ shorter path without *s* we would have found it previously

❖ ... Analysis of Dijkstra's Algorithm

Time complexity analysis ...

Each edge needs to be considered once ⇒ *O(E)*.

Outer loop has *O(V)* iterations.

Implementing "**`find s ∈ vSet with minimum dist[s]`**"

1. try all **`s`**∈**`vSet`** ⇒ cost = *O(V)* ⇒ overall cost = $O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
   - can improve overall cost to $O(E + V \cdot log\ V)$

Produced: 5 Jul 2020