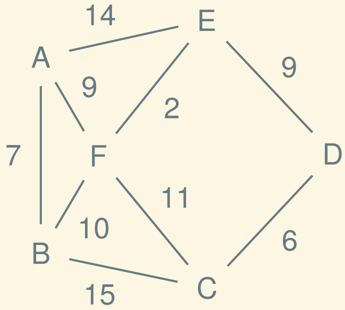
Single-Source Shortest Path

EECS 214, Fall 2017

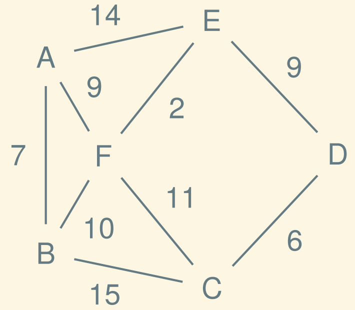
The problem

Find the shortest path from A to D:



The problem

Find the shortest path from A to D:



Typically generalized as single-source shortest paths (SSSP): find the shortest path to everywhere from A.

Relaxation idea

Keep a table with two values for each node:

- the best known distance to it from the start, and
- the predecessor node along that best path

Relaxation idea

Keep a table with two values for each node:

- the best known distance to it from the start, and
- the predecessor node along that best path

To “relax” an edge, we consider whether our knowledge thus far, combined that that edge, can improve our knowledge by finding a shorter path

Relaxation idea

Keep a table with two values for each node:

- the best known distance to it from the start, and
- the predecessor node along that best path

To “relax” an edge, we consider whether our knowledge thus far, combined that that edge, can improve our knowledge by finding a shorter path

For example, suppose that

- the best known distance to node C is 15,
- the best known distance to node D is 4, and
- there’s an edge of weight 5 from D to C.

Relaxation idea

Keep a table with two values for each node:

- the best known distance to it from the start, and
- the predecessor node along that best path

To “relax” an edge, we consider whether our knowledge thus far, combined that that edge, can improve our knowledge by finding a shorter path

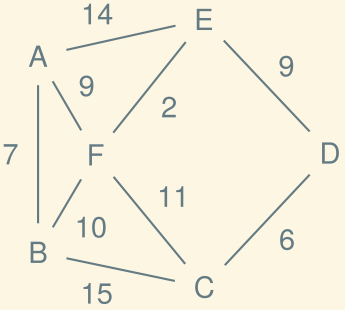
For example, suppose that

- the best known distance to node C is 15,
- the best known distance to node D is 4, and
- there’s an edge of weight 5 from D to C.

Then we update the best known distance to C to be 9, via D.

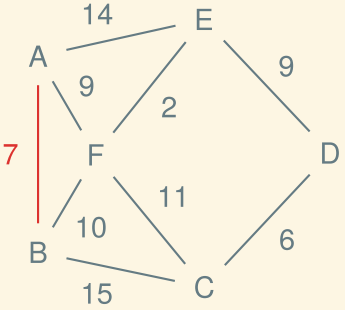
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	



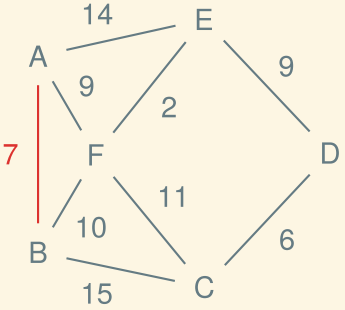
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	



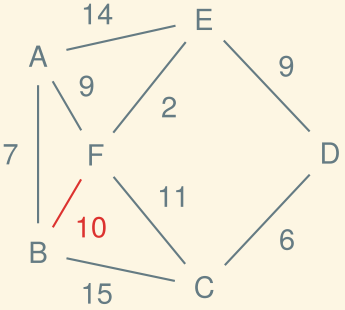
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	∞	
D	∞	
E	∞	
F	∞	



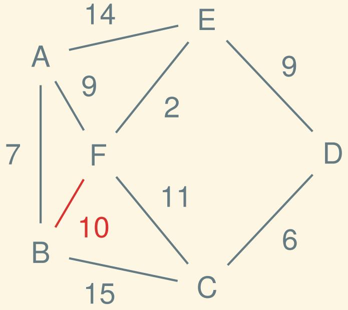
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	∞	
D	∞	
E	∞	
F	∞	



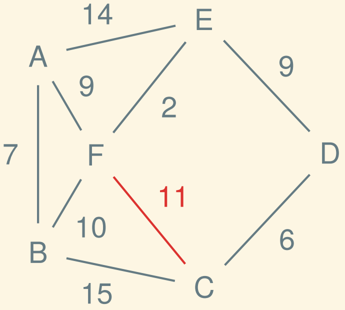
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	∞	
D	∞	
E	∞	
F	17	B



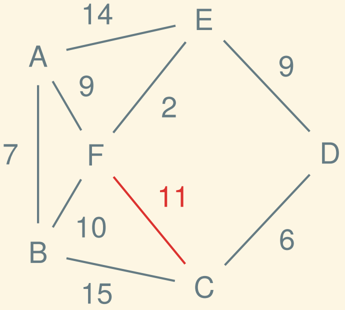
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	∞	
D	∞	
E	∞	
F	17	B



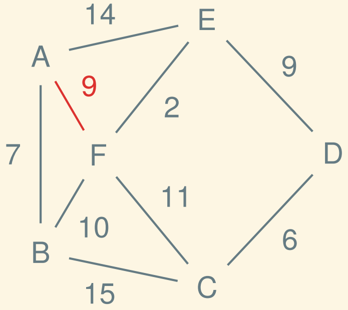
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	28	F
D	∞	
E	∞	
F	17	B



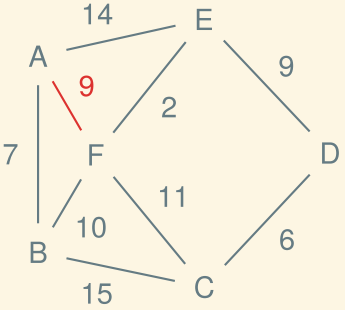
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	28	F
D	∞	
E	∞	
F	17	B



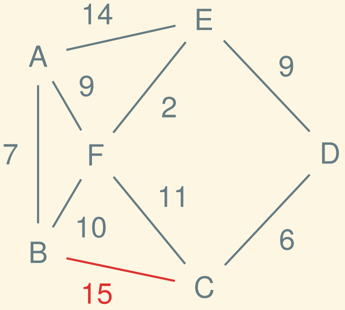
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	28	F
D	∞	
E	∞	
F	9	A



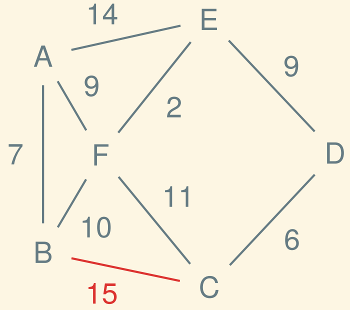
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	28	F
D	∞	
E	∞	
F	9	A



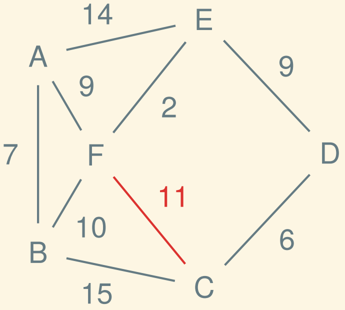
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	22	B
D	∞	
E	∞	
F	9	A



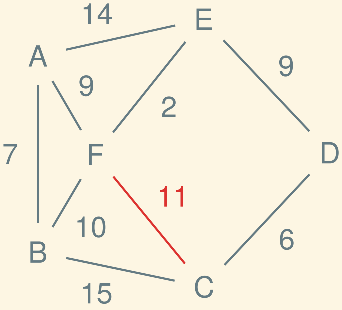
Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	22	B
D	∞	
E	∞	
F	9	A



Relaxation demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	20	F
D	∞	
E	∞	
F	9	A



Bellman–Ford algorithm summary

Solves:	SSSP for graphs with no negative cycles
Main idea:	Relax every edge $\|V\| - 1$ times
Time complexity:	$\mathcal{O}(VE)$

The Bellman–Ford algorithm

Input: A graph *graph* and a starting vertex *start*
Output: Tables of vertex distances *dist* and predecessors *pred*

```
for every vertex v in graph do
    dist[v] ← ∞;
    pred[v] ← −1
end
dist[start] ← 0;

for |Vertices(graph)| − 1 iterations do
    for every edge (v,u) with weight w in graph do
        if dist[v] + w < dist[u] then
            dist[u] ← dist[v] + w;
            pred[u] ← v
        end
    end
end
```

continued...

Bellman–Ford, continued

At this point we have the answer provided there are no negative-weight cycles. We do one more pass to ensure this is the case:

```
for every edge (v,u) with weight w in graph do
    if dist[v] + w < dist[u] then
        graph contains a negative cycle!
    end
end
```

Dijkstra’s algorithm summary

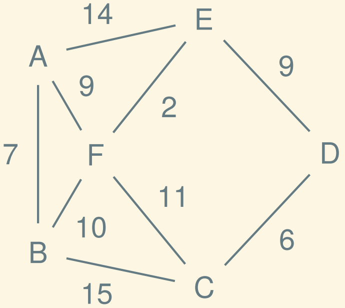
Solves:	SSSP for graphs with no negative *edges*
Main idea:	Relax the edges in a clever order
Time complexity:	depends

Dijkstra's algorithm summary

Solves:	SSSP for graphs with no negative *edges*
Main idea:	Relax the edges in a clever order
Time complexity:	depends

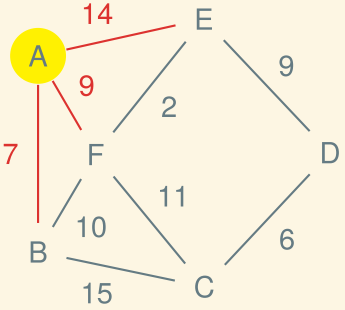
What's the clever order? Relax the edges coming out of the nearest vertex, then repeat

Dijkstra's algorithm demonstration



v	dist[v]	pred[v]
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

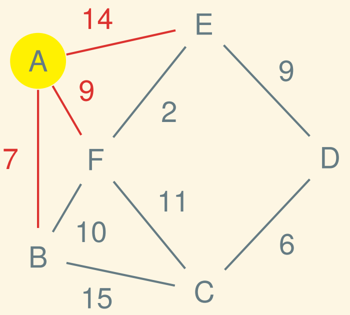
Dijkstra's algorithm demonstration



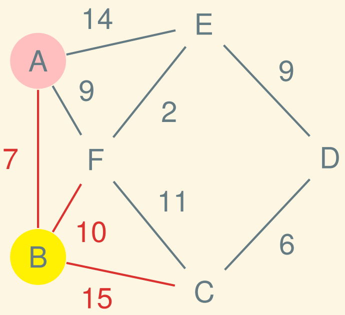
v	dist[v]	pred[v]
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

Dijkstra's algorithm demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	∞	
D	∞	
E	14	A
F	9	A



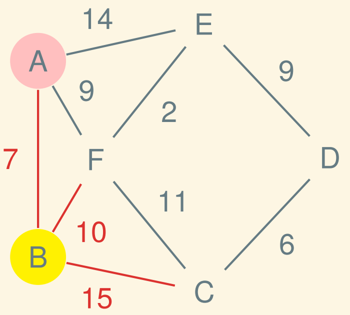
Dijkstra's algorithm demonstration



v	dist[v]	pred[v]
A	0	
B	7	A
C	∞	
D	∞	
E	14	A
F	9	A

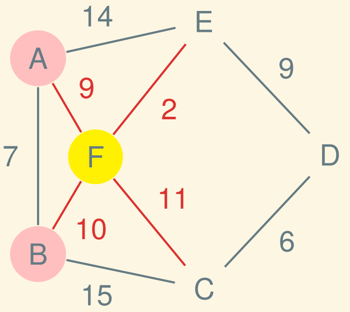
Dijkstra's algorithm demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	22	B
D	∞	
E	14	A
F	9	A



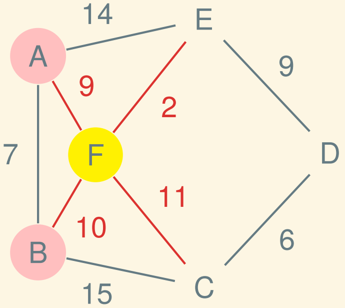
Dijkstra's algorithm demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	22	B
D	∞	
E	14	A
F	9	A



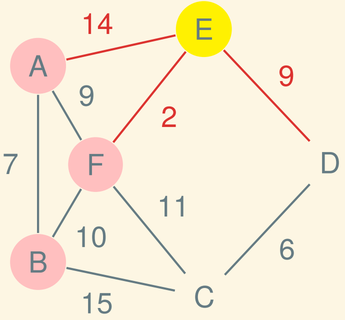
Dijkstra's algorithm demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	20	F
D	∞	
E	11	F
F	9	A

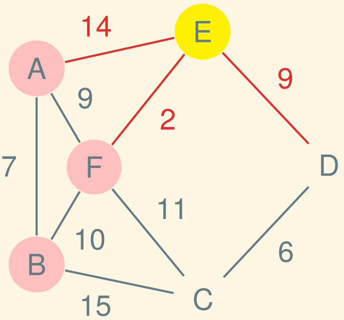


Dijkstra's algorithm demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	20	F
D	∞	
E	11	F
F	9	A

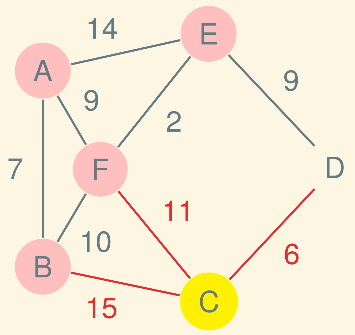


Dijkstra's algorithm demonstration



v	dist[v]	pred[v]
A	0	
B	7	A
C	20	F
D	20	E
E	11	F
F	9	A

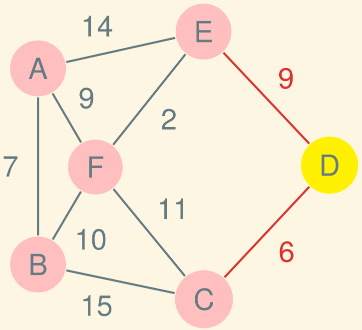
Dijkstra's algorithm demonstration



v	dist[v]	pred[v]
A	0	
B	7	A
C	20	F
D	20	E
E	11	F
F	9	A

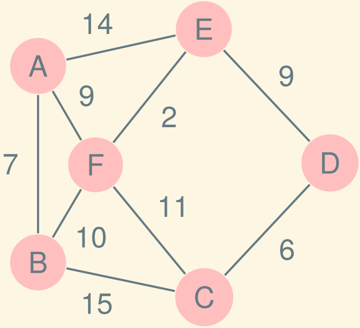
Dijkstra's algorithm demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	20	F
D	20	E
E	11	F
F	9	A



Dijkstra’s algorithm demonstration

v	dist[v]	pred[v]
A	0	
B	7	A
C	20	F
D	20	E
E	11	F
F	9	A



Dijkstra's algorithm (original)

Input: A graph *graph* and a starting vertex *start*
Output: Tables of vertex distances *dist* and predecessors *pred*

```
for every vertex v in graph do
    dist[v] ← ∞; pred[v] ← −1;
end
dist[start] ← 0;

todo ← the set of vertices in graph;
while todo is not empty do
    v ← remove the element of todo with minimal dist[v];
    for every outgoing edge (v,u) with weight w do
        if dist[v] + w < dist[u] then
            dist[u] ← dist[v] + w;
            pred[u] ← v
        end
    end
end
```

Priority Queue ADT

Looks like: ⟨ 2:g 5:i 5:b 17:c 89:g ⟨ (note sorting)

Operations:

- *isEmpty*(PrioQueue): Bool
- *insert*(PrioQueue, Key, Value): Void
- *peekMin*(PrioQueue): (Key, Value)
- *removeMin*(PrioQueue): (Key, Value)

Behavior:

- Keeps key-value pairs sorted by key, so that
- *removeMin* can find and remove the pair with the smallest key

Dijkstra's algorithm with priority queue

Input: A graph *graph* and a starting vertex *start*
Output: Tables of vertex distances *dist* and predecessors *pred*

for every vertex *v* in *graph* do
    $dist[v] \leftarrow \infty$; $pred[v] \leftarrow -1$;
end
$dist[start] \leftarrow 0$;
todo $\leftarrow$ empty priority queue;
`Insert(`*todo, 0, start*`)`;

while *todo* is not empty do
    $(\_, v) \leftarrow$ `RemoveMin(`*todo*`)`;
    for every outgoing edge (*v*,*u*) with weight *w* do
        if $dist[v] + w < dist[u]$ then
            $dist[u] \leftarrow dist[v] + w$;
            $pred[u] \leftarrow v$;
            `Insert(`*todo*, $dist[u]$, *u*`)`
        end
    end
end

Complexity of Dijkstra's algorithm

- Relax every edge once, for $\mathcal{O}(E)$
- For every edge, we (might) do an *insert*, which takes how long?

Complexity of Dijkstra's algorithm

- Relax every edge once, for $\mathcal{O}(E)$
- For every edge, we (might) do an *insert*, which takes how long? Call it T_{in}.
- For every vertex, we do a *removeMin*, which takes how long? Call it T_{rm}.

Complexity of Dijkstra's algorithm

- Relax every edge once, for $\mathcal{O}(E)$
- For every edge, we (might) do an *insert*, which takes how long? Call it T_{in}.
- For every vertex, we do a *removeMin*, which takes how long? Call it T_{rm}.

- Then Dijkstra's algorithm is $\mathcal{O}(ET_{in} + VT_{rm})$.

Next time: making *removeMin* and *insert* fast