COMP_SCI 214: Data Structures and Algorithms

Single-Source Shortest Path

PROF. SRUTI BHAGAVATULA

Announcements

- Homework 5 due today
- Homework 4 self-eval due today

- Project to be released later today
  - Much larger than a homework
    - and hence much more time and more submissions
  - START EARLY!!!
  - Details forthcoming on Piazza and in the handout --- Read fully!

Another class of graph algorithms

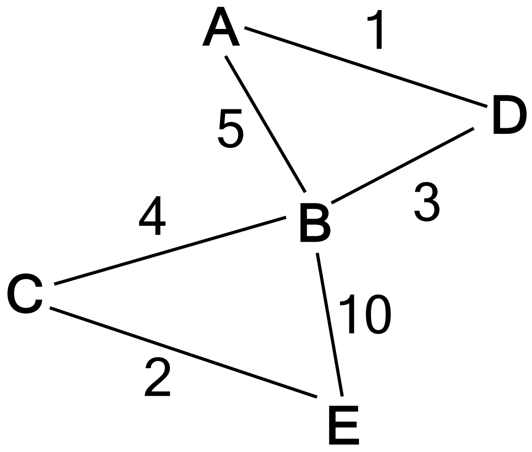
Problem of the day

- Given a graph containing locations (nodes) connected via roads (edges) of different lengths (weights), how can we find the shortest path between two nodes?

- Basically: how does Google Maps do this?

The problem: Finding shortest paths

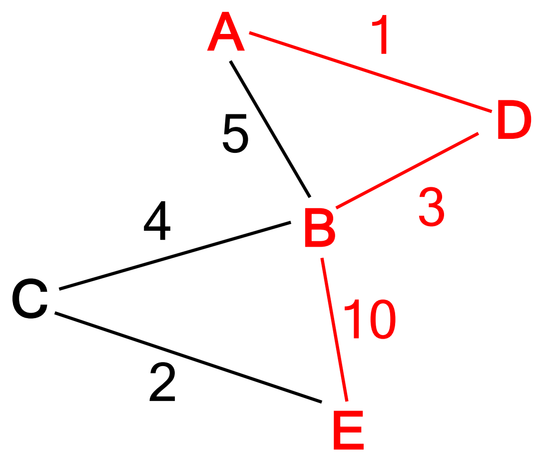
- On this weighted graph, what is the shortest path from A to E?



The problem: Finding shortest paths

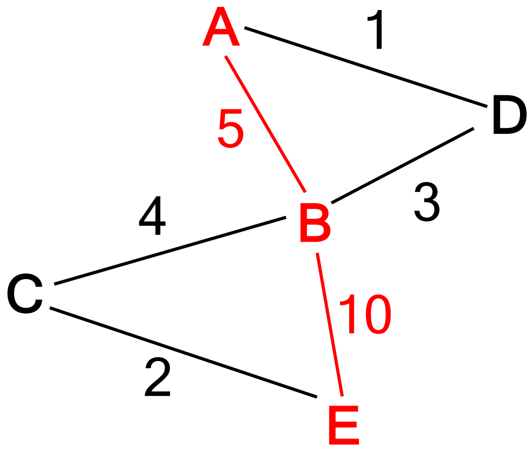
- On this weighted graph, what is the shortest path from A to E?

- A-D-B-E: 14



The problem: Finding shortest paths

- On this weighted graph, what is the shortest path from A to E?

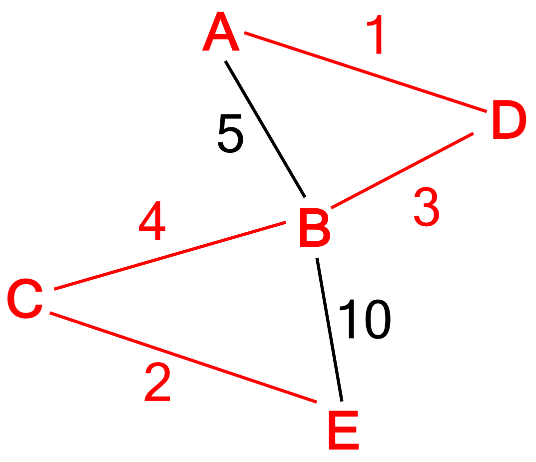


- A-D-B-E: 14
- A-B-E: 15. Worse!

The problem: Finding shortest paths

- On this weighted graph, what is the shortest path from A to E?

- A-D-B-E: 14
- A-B-E: 15. Worse!
- A-D-B-C-E: 10
  - We have a winner!



The problem, generalized

- Find the shortest path to everywhere from a given vertex
  - **Single-source shortest path (SSSP)** problem
- Why generalize?
  - To find SP to one destination, we may end up finding the shortest path to other destinations anyway
  - So might as well
- Applies to directed and undirected graphs alike

Why not BFS?

- BFS can do exactly this
  - If #edges in a path is the path length
  - Because you are always minimizing number of edges in path by checking paths of a specific length before moving onto next level of neighbors
- But what about when we have weights?
  - Path length = sum of weights in path
  - BFS doesn't consider this
  - Need to modify BFS or use a different approach

Problem setup

- For each vertex in the graph, we want to find:
  - The minimum "cost" of getting from starting vertex to vertex in question
  - The actual path that gives us this minimal cost
- Think of "cost" as the sum of weights along a path
- Let's maintain the cost of reaching every vertex and the predecessor of the vertex in this minimal path

The solution: Just relax

- Relaxation:
  - Set cost for reaching each vertex from starting vertex to its maximum: ∞
  - Bring cost of reaching a vertex down as we learn more information about the rest of the graph
  - Once you’ve relaxed enough, you have the real cost of reaching a node
- General approach to algorithms

SSSP setup

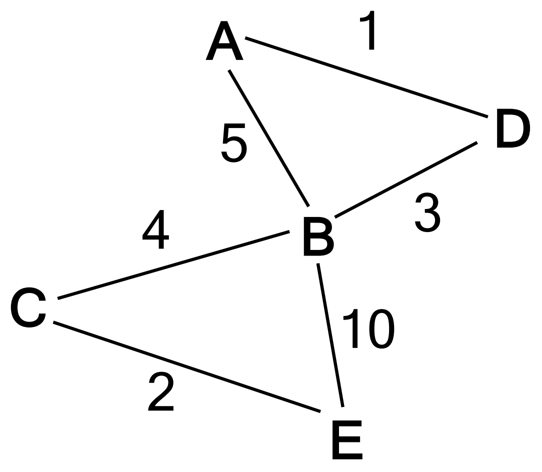
- Let's maintain two pieces of information:

1. The cost of reaching every vertex from the starting vertex (`dist[v]`)
   - We'll keep updating this as we learn about the graph
2. The predecessor of the vertex in this minimal path (`pred[v]`)
   - At the end, we can trace this back to get the shortest path

Relaxation for SSSP

- SSSP from A
- To start, most pessimistic costs possible: everything ∞
- As we look at edges, we learn more information about the cost to get to vertices

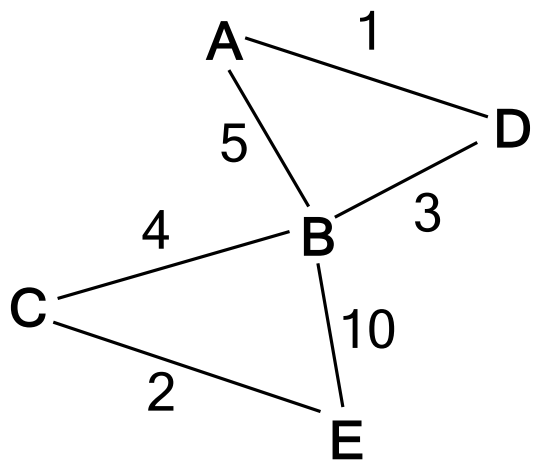
`v`	`dist[v]`	`pred[v]`
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	



Relaxation for SSSP

- Visit A first
- Since node is itself, path length is 0

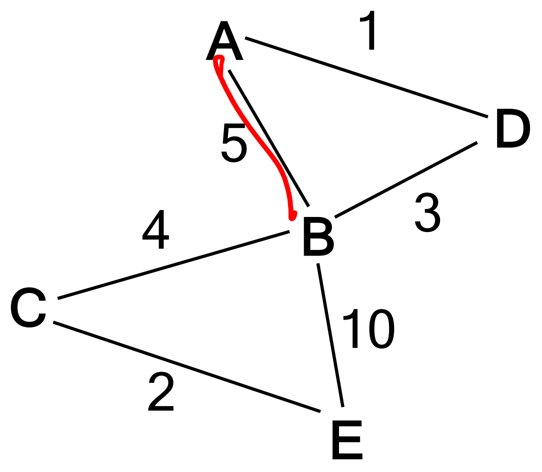
v	dist[v]	pred[v]
A	0	
B	∞	
C	∞	
D	∞	
E	∞	



Relaxation for SSSP

- Next visit B via edge A-B (A's neighbor)
- Distance from A-B is 5
  - Update `dist[B]` and `pred[B]` in table

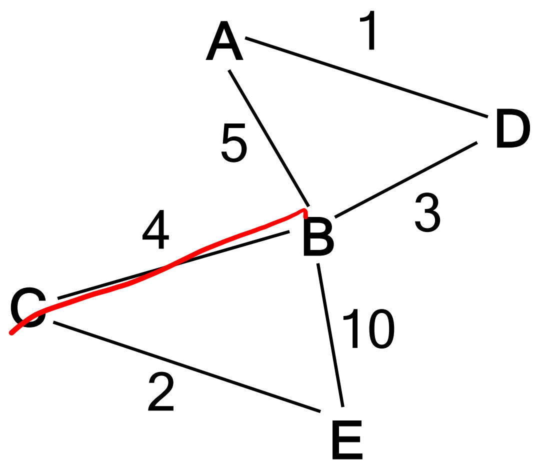
v	dist[v]	pred[v]
A	0	
B	5	A
C	∞	
D	∞	
E	∞	



Relaxation for SSSP

- Let's visit C via the B-C edge now (B's neighbor)
- Distance from A-C = A-B distance + B-C distance = 5 + 4 = 9
  - Update `dist[C]` and `pred[C]` in table

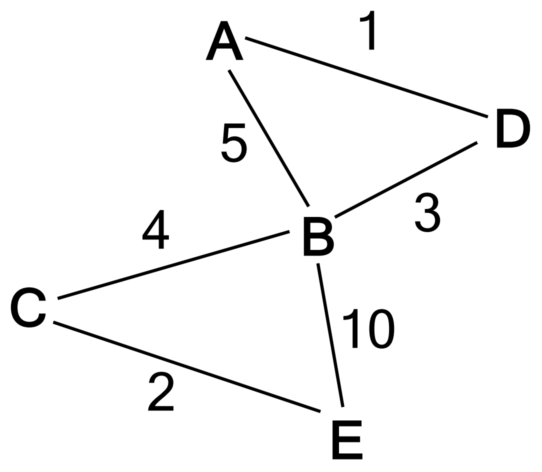
v	**dist[v]**	**pred[v]**
A	0	
B	5	A
C	9	B
D	∞	
E	∞	



Relaxation for SSSP

- Let's visit E via the C-E edge now (C's neighbor)
- Distance from A-E = A-C distance + C-E distance = 9 + 2 = 11
  - Update `dist[E]` and `pred[E]` in table

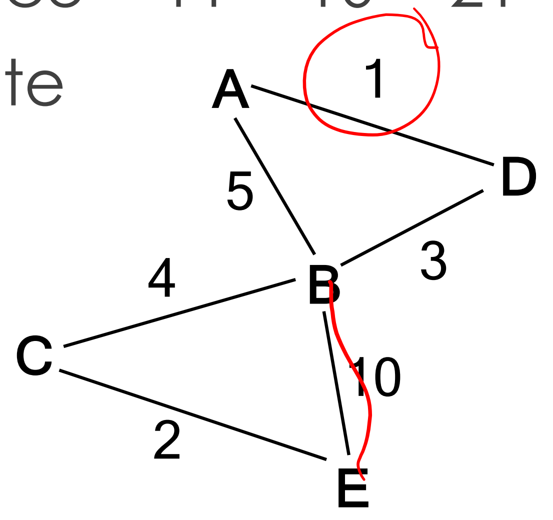
v	dist[v]	pred[v]
A	0	
B	5	A
C	9	B
D	∞	
E	11	C



Relaxation for SSSP

- Let's visit B via the E-B edge now (E's neighbor)
- Distance from A-B = A-E distance + E-B distance = 11 + 10 = 21
  - Not as good as `dist[B] = 5`; don't update

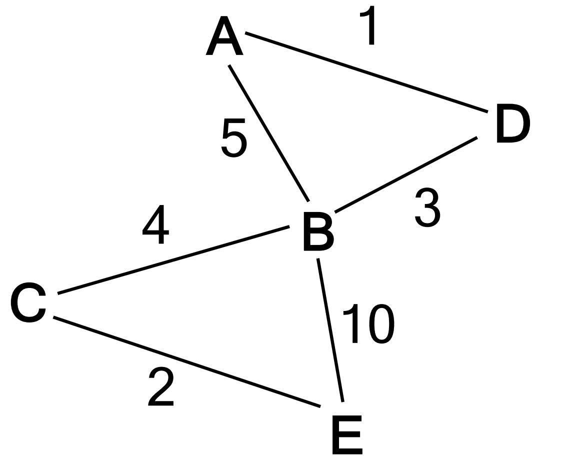
v	dist[v]	pred[v]
A	0	
B	5	A
C	9	B
D	∞	
E	11	C



Relaxation for SSSP

- Let's now look at D via the A-D edge (A's neighbor from earlier)
- Distance from A-D = A-A distance + A-D distance = 0 +1 = 1
  - Update `dist[D]` and `pred[D]`

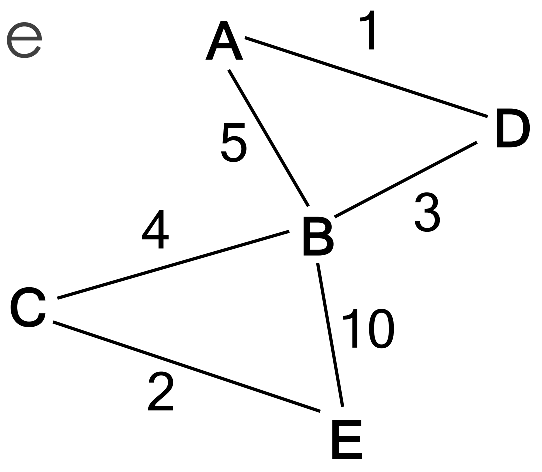
v	dist[v]	pred[v]
A	0	
B	5	A
C	9	B
D	1	A
E	11	C



Relaxation for SSSP

- Let's now look at B via the D-B edge (D's neighbor)
- Distance from A-B = A-D distance + D-B distance = 1 + 3 = 4
  - Better than old `dist[B]` so update in table

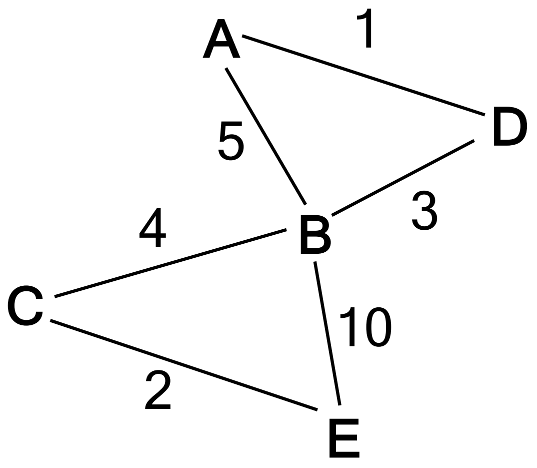
v	**dist[v]**	**pred[v]**
A	0	
B	~~5~~ 4	~~A~~ D
C	9	B
D	1	A
E	11	C



Relaxation for SSSP

- Oh, but our graph is undirected! D-B = B-D!
- Did we find a shorter path to D via B too? No. **5+3 > 1** in the graph
  - Each edge direction needs to be treated separately even if undirected

v	dist[v]	pred[v]
A	0	
B	4	D
C	9	B
D	1	A
E	11	C



Relaxing an edge

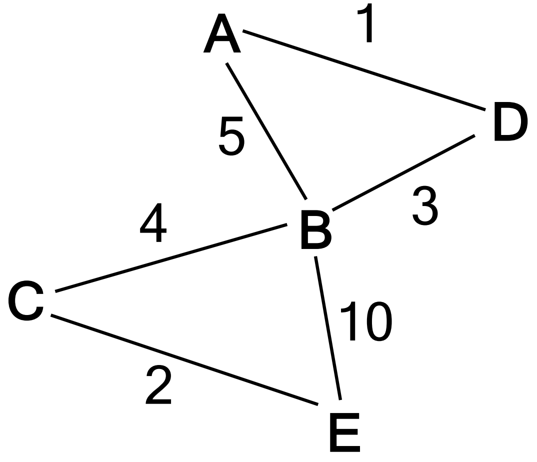
- Given the starting vertex `s`
- Relaxing an edge `(u, v)` means checking if a path from `s` to `v` which passes through `(u, v)` is shorter than other paths to `v` seen before
  - To start, since all paths to vertices start at ∞, any path will be shorter the first time we relax an edge
- If `(u, v)` makes the path shorter, we reduce the cost to reach `v`

```
if dist[u] + w(u, v) < dist[v]:
    dist[v] ← dist[u] + w(u, v)
```

Relaxation for SSSP

- Are we done? Maybe! But we'll stop here.
- Different algorithms have different stopping conditions
  - All guaranteed to find shortest path to each node

v	**dist[v]**	**pred[v]**
A	0	
B	4	D
C	9	B
D	1	A
E	11	C



What do we need from an algorithm?

- Systematic way to relax edges further after a round of relaxing
- A clear stopping condition

Dynamic programming

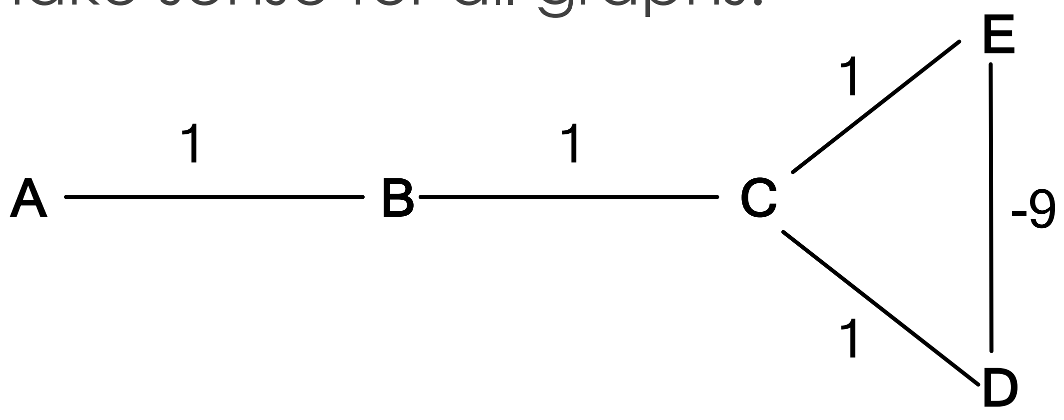
- To determine shortest path between `a` and `b`, we use an intermediate shortest path on the way (e.g., `a` and `q`) and build on that

- These algorithms that solve sub-problems and determine the final solutions using the results of sub-solutions use **dynamic programming**
  - You should see more of this in later classes!

Bellman-Ford Algorithm

A note on negative path lengths

- SSSP doesn't make sense for all graphs.



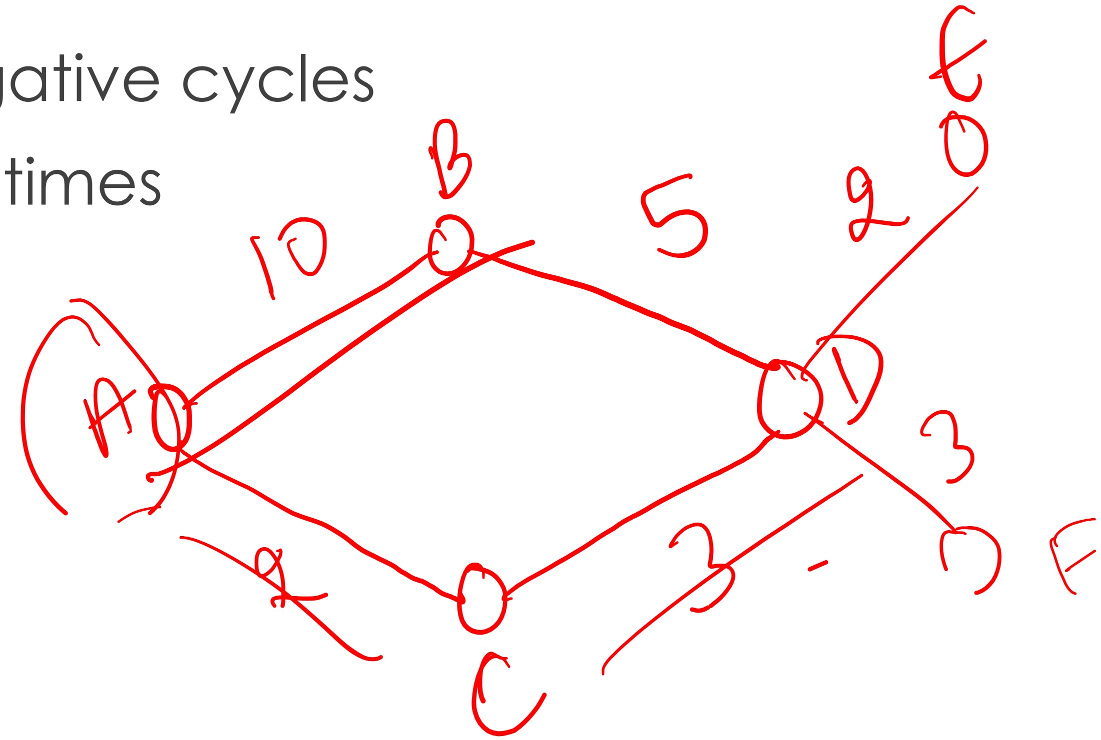
- A-B-C-E has length 3.
- A-B-C-D-E has length -6. Shorter!
- A-B-C-D-E-C-D-E has length -13. Even shorter!

Negative cycles

- Negative cycle = path length with negative value
- Negative cycles means no short*est* path

Bellman-Ford algorithm

- **Solves:** SSSP for graphs with no negative cycles
- **Main idea:** Relax every edge $v - 1$ times
- **Time complexity:** ?
- Steps:
  - Repeat steps 1-2 $v - 1$ times
  1. Pick edges in some order
  2. Relax the edges



A	0
B	10
C	2
D	~~15~~ 5

Initialization for Bellman-Ford algorithm

Input: A graph `graph` and starting vertex `start`
Output: Table of vertex distances `dist` and predecessors `pred`

```
for every vertex v in graph do
   dist[v] ← ∞; pred[v] ← None
end

dist[start] ← 0
```

v	dist[v]	pred[v]
0	∞	
1	∞	
2	∞	
3	∞	
…		

The Bellman-Ford algorithm

```
for |Vertices(graph)|-1 iterations do
    for every edge (v, u) with weight w in graph do
        # This conditional relaxes edge v-u.
        if dist[v] + w < dist[u] then
            # (if undirected, we'd also relax edge u-v)
            dist[u] ← dist[v] + w;
            pred[u] ← v;
        end
    end
end
```

Why does $v - 1$ iterations work?

- Iteration 1: Nodes whose shortest path is 1 edge away now have their correct shortest paths
  - No guarantees about the other nodes because of arbitrary order
- Iteration 2: Nodes whose shortest path is 2 edges away now have their correct shortest paths
- Iteration 3: Nodes whose shortest path is 3 edges away now have their correct shortest paths
- …
- A shortest path between nodes can have a max of $v - 1$ edges, therefore we'll have the largest shortest path after $v - 1$ iterations

Bellman-Ford with negative cycles

- If there are no negative cycles, we're done with Bellman-Ford
- But we should check for negative cycles:
  - After we're done with Bellman-Ford, try to relax all the edges one more time
  - If a cost reduces further, we have a negative cycle
    - Shouldn't be possible to improve further after $v - 1$ iterations otherwise
    - There will never be a minimum then as cost can keep reducing

- Are we satisfied to go to use Bellman-Ford for our map problem now?

Revisit: problem of the day

- Given a graph containing locations (nodes) connected via roads (edges) of different lengths (weights), how can we find the shortest path between two nodes?

- This graph may have LOTS of edges in $O(v^2)$ in the worst case if it's a dense graph

The issue with Bellman-Ford

- Bellman-Ford is $O(e \ * v)$
- …which, in the worst case, becomes $O(v^3)$ if edges in $O(v^2)$
  - Cubic, yikes!

Wastage in Bellman-Ford

- We may not need all $v - 1$ iterations of relaxations
  - Could stop when a round of relaxing all edges doesn't improve anything

- Relaxing edges in an arbitrary order may not be optimal
  - A more clever order could get us there faster
  - Let's see another algorithm that does this; relaxes each edge *once*!

Dijkstra’s Algorithm

Dijkstra’s algorithm

- Named after Edsger W. Dijkstra (pronounced: “dike-strah”)
  - Early CS pioneer, 1972 Turing award
  - A name you’ll see often in CS

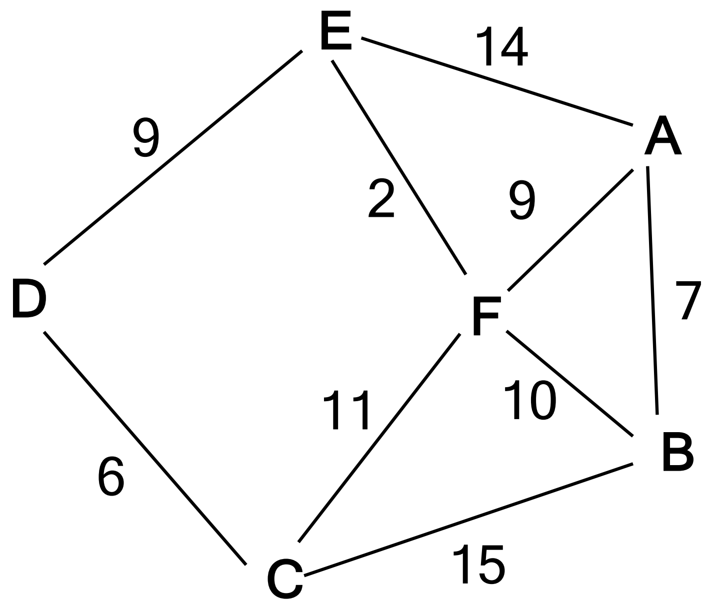
Dijkstra's algorithm

- **Solves:** SSSP for graphs with <some restrictions>
- **Main idea:** Relax the edges in a clever order (like BFS)
- **Time complexity:** ?

- Main approach:
  - Relax the edges coming out of the *nearest* vertex so far
  - Then repeat with next nearest, etc.
  - Keep track of edges already relaxed; relax each edge only once!

Dijkstra's algorithm in action: SSSP from A

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

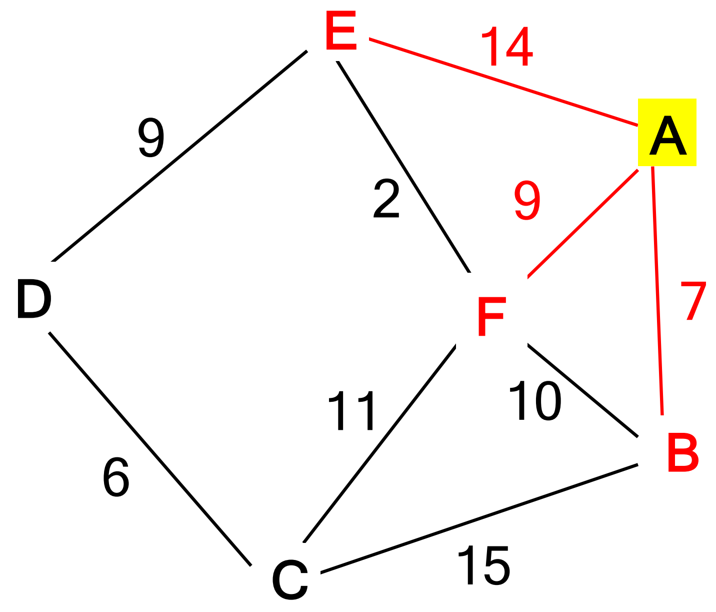


Left: {A, B, C, D, E, F}

Dijkstra's algorithm in action

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

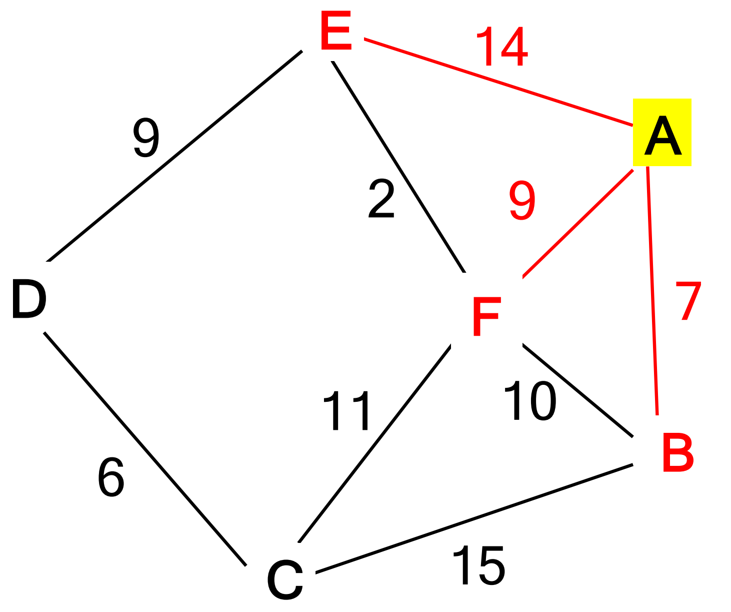
Step:
`A` has lowest `dist` in table so far, visit it



Left: {A, B, C, D, E, F}

Dijkstra's algorithm in action

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

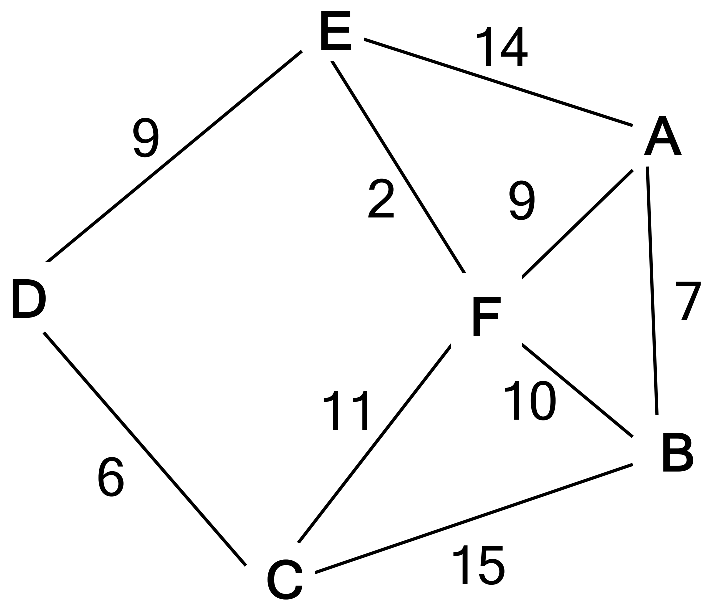


Step:
For each of A's neighbors `n`, relax (A, n)

Left: {~~A~~, B, C, D, E, F}

Dijkstra's algorithm in action

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

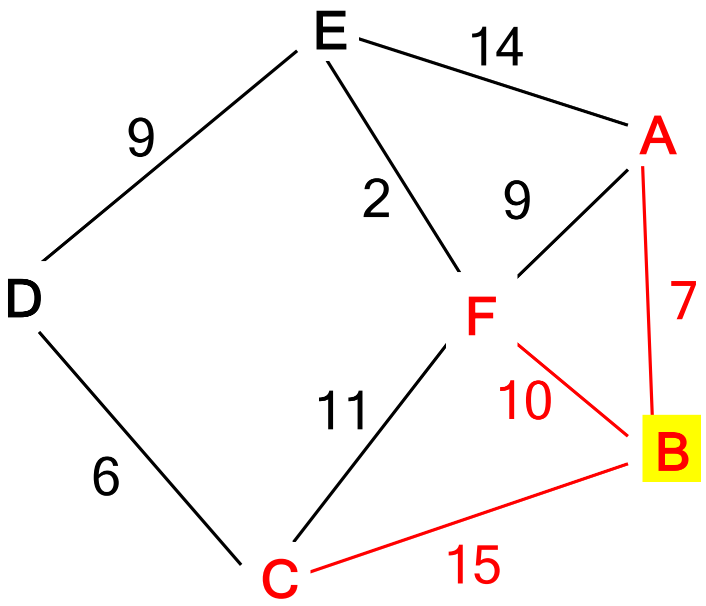


Step:
Next, look at A's nearest unvisited vertex: B

Left: {B, C, D, E, F}

Dijkstra's algorithm in action

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



Now that we're at B, we **know** we found the shortest path to it

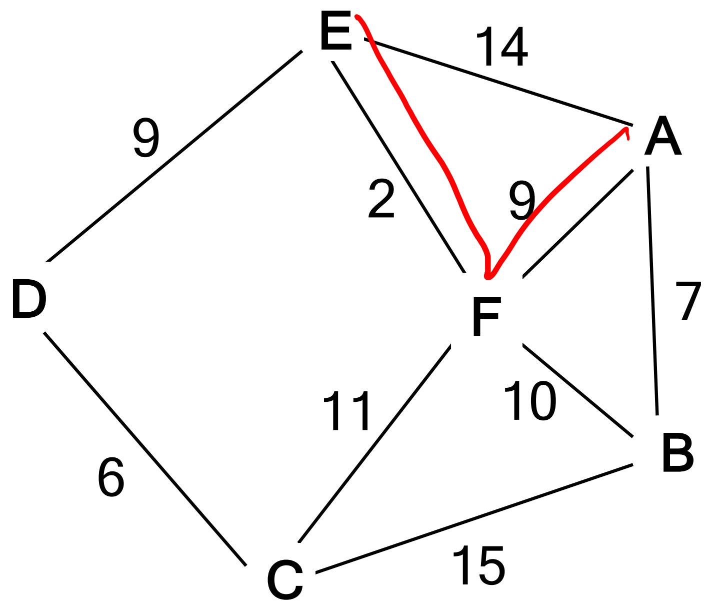
Step:

For each of B's neighbors `n`, relax (B, n)

Left: {~~B~~, C, D, E, F}

Dijkstra's algorithm in action

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



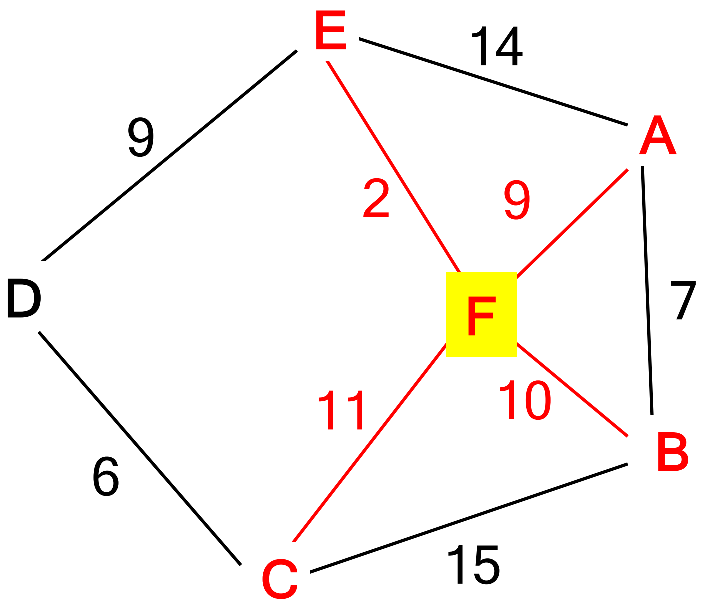
Step:

Next, look at A's nearest unvisited vertex: F

Left: {C, D, E, F}

Dijkstra's algorithm in action

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



Now that we're at F, we **know** we found the shortest path to it

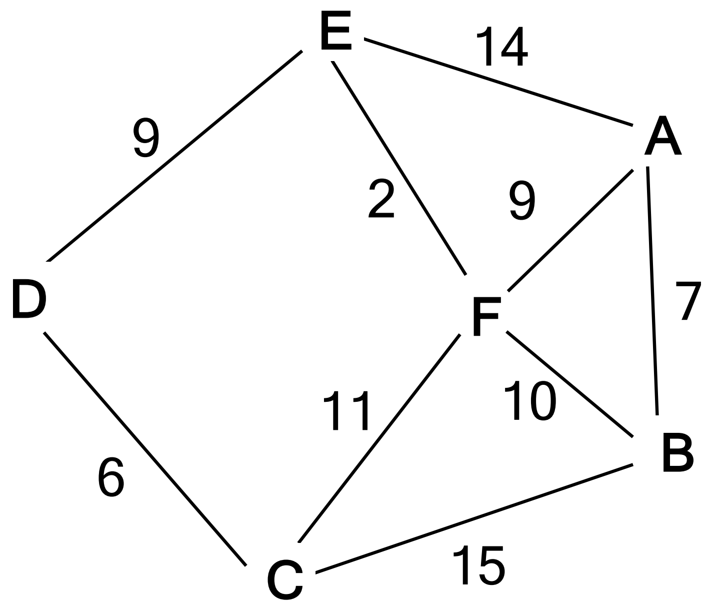
Step:

For each of F's neighbors `n`, relax (F, n)

Left: {C, D, E, ~~F~~}

Dijkstra's algorithm in action

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

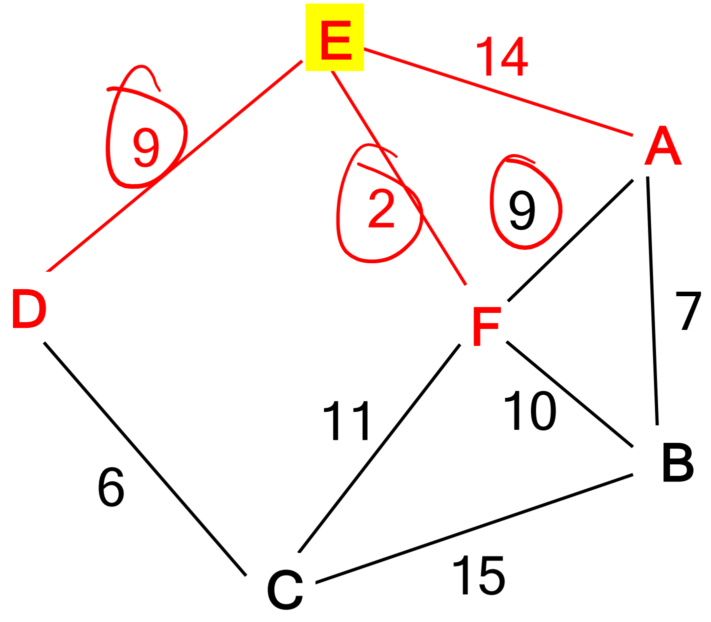


Step:
Next, look at A's nearest unvisited vertex: E

Left: {C, D, E}

Dijkstra's algorithm in action

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



Now that we're at E, we **know** we found the shortest path to it

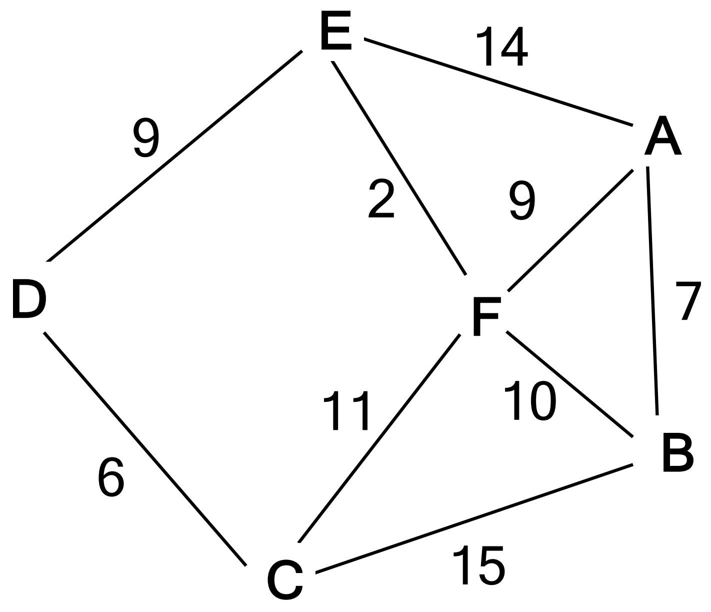
Step:

For each of E's neighbors `n`, relax (E, n)

Left: {C, D, ~~E~~}

Dijkstra’s algorithm in action

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

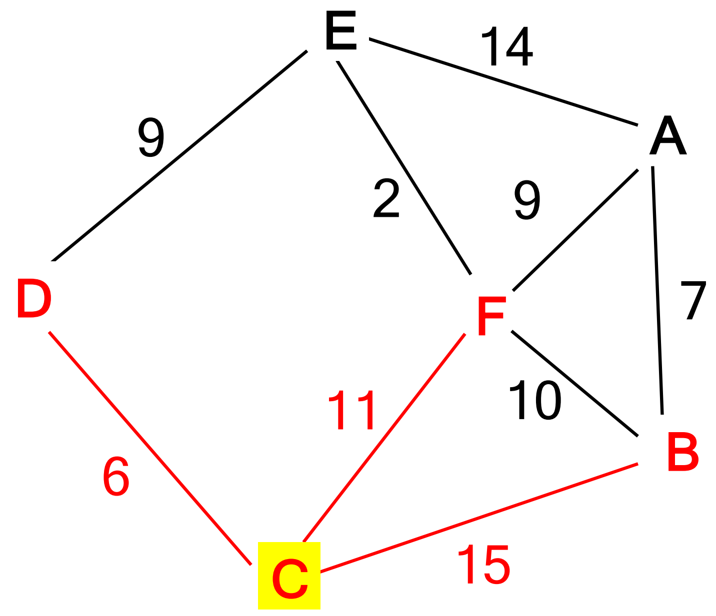


Step:
Next, look at A’s nearest unvisited vertex: C

Left: {C, D}

Dijkstra's algorithm in action

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



Now that we're at C, we **know** we found the shortest path to it

Step:

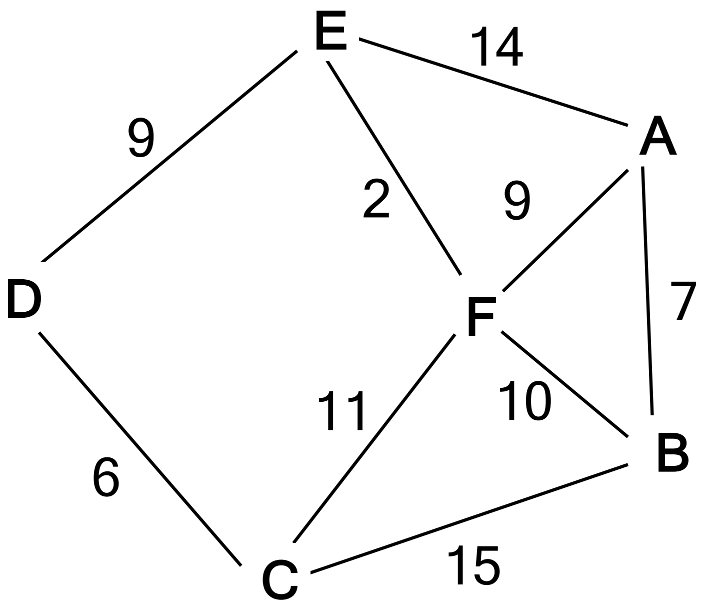
For each of C's neighbors `n`, relax (C, n)

Left: {~~C~~, D}

Dijkstra’s algorithm in action

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

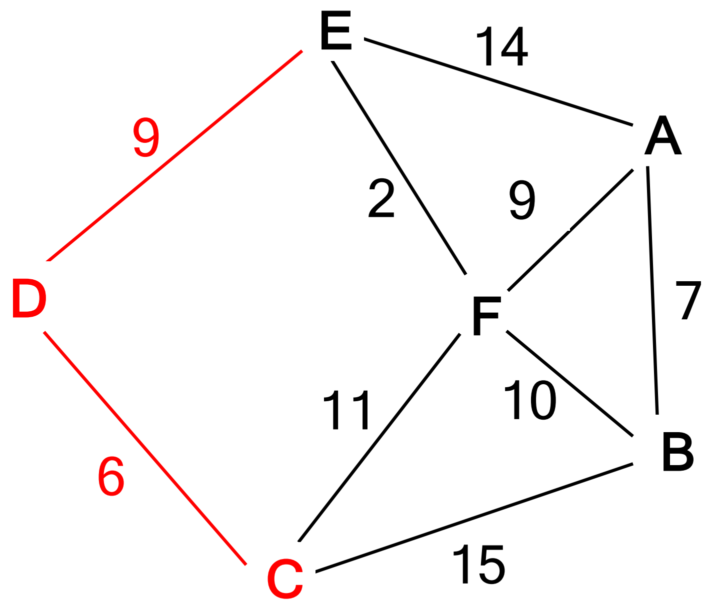
Step:
Next, look for A’s nearest unvisited vertex: D



Left: {D}

Dijkstra's algorithm in action

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



Now that we're at D, we **know** we found the shortest path to it

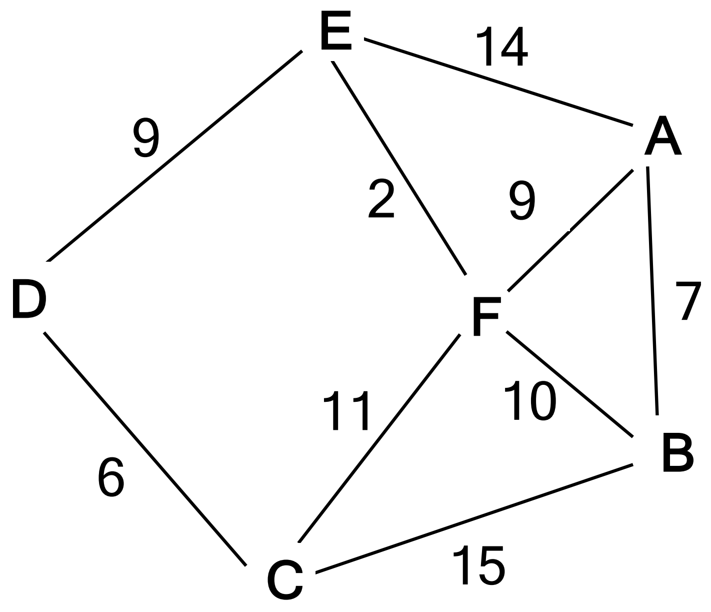
Step:

For each of D's neighbors `n`, relax (D, n)

Left: {~~D~~}

Dijkstra's algorithm in action

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



No vertices left to visit, we found all the shortest paths!

Left: {}

Recovering the shortest path

- Start from destination and build path backwards using pred
  - Just like in DFS/BFS!

- **Quiz:** Shortest path from A to D?
  1. A-E-D
  2. A-F-E-D
  3. A-B-C-D

A - F - E - D

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

Initialization for Dijkstra’s algorithm v1

Input: A graph `graph` and starting vertex `start`
Output: Table of vertex distances `dist` and predecessors `pred`

```
for every vertex v in graph do
   dist[v] ← ∞; pred[v] ← None
end

dist[start] ← 0
left ← set of vertices in graph
```

v	dist	pred
0	∞	
1	∞	
2	∞	
3	∞	
…		

Dijkstra’s algorithm v1

```
left ← set of vertices in graph

while left is not empty do
   v ← remove the element of left 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
```

Dijkstra's algorithm v1

```
left ← set of vertices in graph

while left is not empty do
    v ← remove the element of left 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
```

Cool algorithm.

But how do you find the minimal element in `left`?

We need a worklist

- BFS used a queue, but we don't want FIFO
- We want the minimum element in a worklist
  - Or element of highest priority?
- We can use a priority queue!
  - Values are vertices to look at next (like in BFS)
  - Priorities are `dist[v]`

Recall: Priority queue ADT

- Abstract values look like (note the sorting) →
  - Priority-value pairs

```
interface PRIORITY_QUEUE[T]:

  def empty?(self) -> bool?
  def insert(self, priority: num?, value: T) -> NoneC
  def remove_min(self) -> T
```

Highest priority

2	Brain damage
5	Heart attack
17	Fever
89	Cold
…	…

Initialization for Dijkstra's algorithm v2 w/ Priority Queue

Input: A graph `graph` and starting vertex `start`
Output: Table of vertex distances `dist` and predecessors `pred`

```
for every vertex v in graph do
   dist[v] ← ∞; pred[v] ← None
end

dist[start] ← 0

todo ← empty priority queue;
done ← empty vertex set e.g., mark array

todo.insert(0, start);
```

Dijkstra’s algorithm v2 w/ Priority Queue

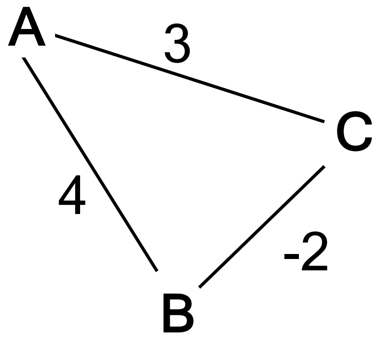
```
while todo is not empty do
   v ← todo.remove_min(); #pick the nearest vertex
   if v ∉ done then
      done ← done ∪ {v};
      for every outgoing edge (v,u) with weight w do
         #relax outgoing edges
         if dist[v] + w < dist[u] then
            dist[u] ← dist[v] + w;
            pred[u] ← v;

            todo.insert(dist[u], u);
         end
      end
   end
```

In-class exercise (6 minutes)

1. Run Dijkstra's algorithm from A and write the results of the table in the format on the right
2. Dijkstra's algorithm cannot find shortest path in this graph from A-C. Why might that be based on your table above?

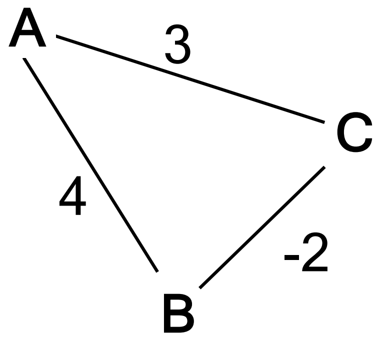
A, d[v], p[v]
B, d[v], p[v]
C, d[v], p[v]



v	dist[v]	pred[v]
A	0	
B		
C		

In-class exercise solution

1. Actual shortest path: A-B-C with path length of 4-2 = 2
2. Dijkstra's algorithm would not be able to find this path because it would prioritize first going to C from A and then would relax the C-B edge overwriting `dist[B]`
   - Dijkstra's wouldn't be able to find a path from A-C as following the above approach it would give us C-B-C-B-C… by following predecessors from C
   - No guarantees on an optimal path or a path at all
- Hence…



v	dist[v]	pred[v]
A	0	
B	1	C
C	-1	B

Dijkstra’s algorithm

- **Solves:** SSSP for graphs with **no negative edge weights**
- **Main idea:** Relax the edges in a clever order (like BFS)
- **Time complexity:** ?

- More restrictive but a more optimal algorithm

Time complexity of Dijkstra's algorithm

```
while todo is not empty do
    v ← todo.remove_min(); #pick the nearest vertex
    if v ∉ done then
        done ← done ∪ {v};
        for every outgoing edge (v,u) with weight w do
            #relax outgoing edges
            if dist[v] + w < dist[u] then
                dist[u] ← dist[v] + w;
                pred[u] ← v;

                todo.insert(dist[u], u);
            end
        end
    end
```

Time complexity of Dijkstra's algorithm

- Relax every edge once: $O(e)$
- For every edge we relax, we do an `insert`
  - Which takes $O(\log e)$
- For every edge we relax, we do a `remove_min`
  - Which takes $O(\log e)$

- So Dijkstra's algorithm is $O(e \log e)$
  - Which is bounded by $O(v^2 \log v^2) = O(v^2 \log v)$ in the worst case
  - Better than Bellman-Ford's $O(v^3)$