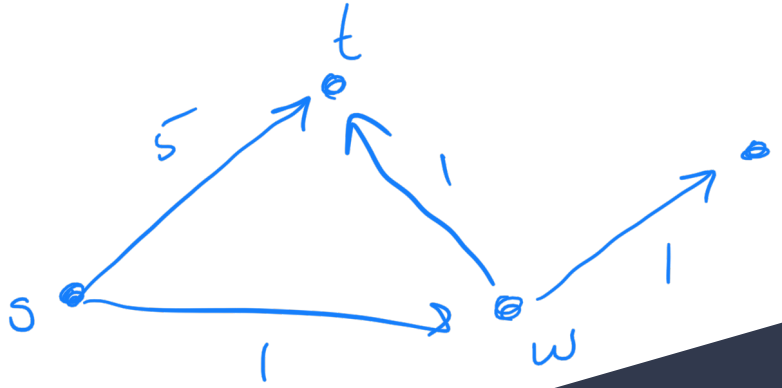
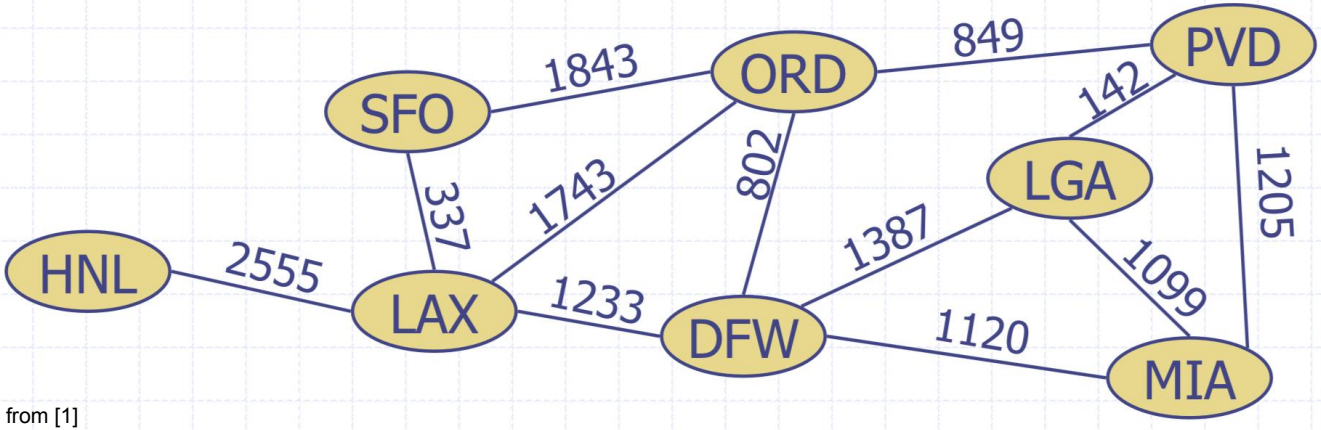
Shortest Path

- Weighted Graphs
- Shortest Path Problem
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- DAG-based Algorithm
- All-Pairs Shortest Path



Weighted Graphs

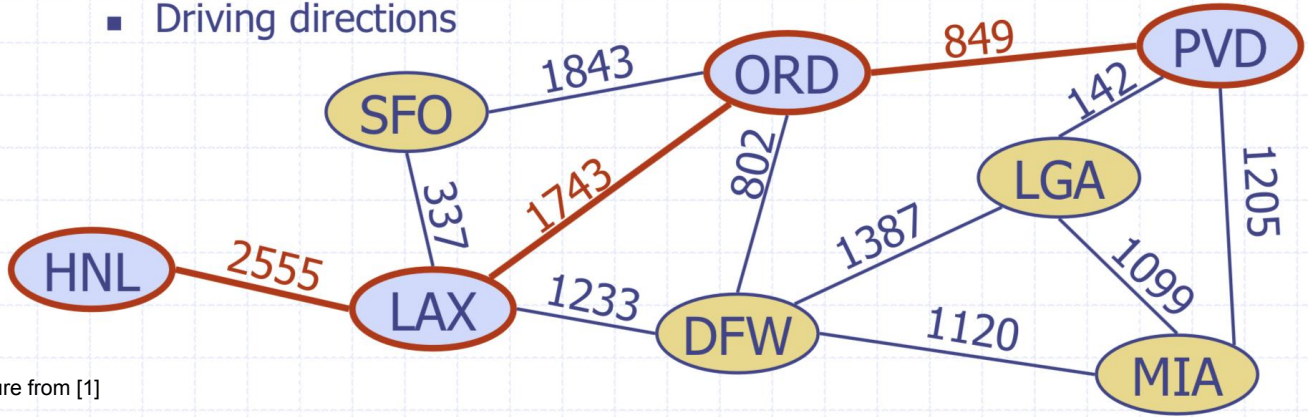
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Picture from [1]

Shortest Path Problem

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Picture from [1]

Shortest Path Properties

Property 1:

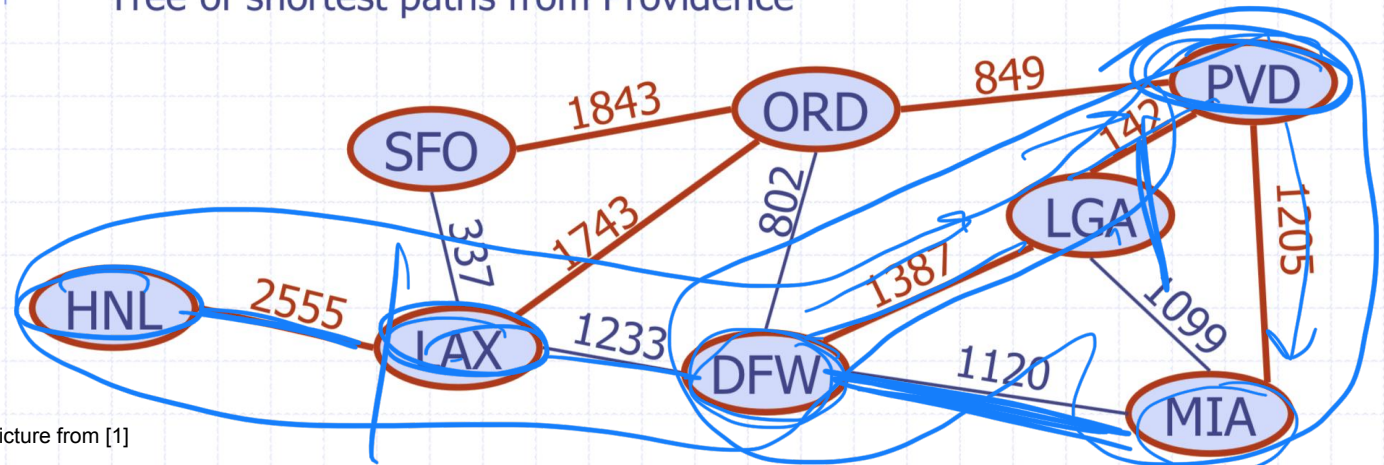
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



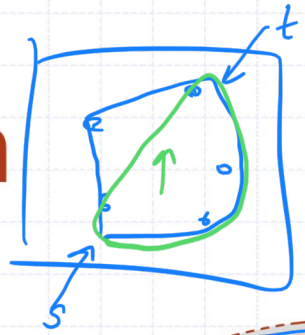
Picture from [1]

Dijkstra's Algorithm: A Greedy Approach

- The *distance* of a vertex **v** from a vertex **s** is the length of the shortest path from **s** to **v**
- Dijkstra's Algorithm computes the distances of all the vertices from the start vertex **s**
- Assumptions:
  - the graph is connected
  - the edge weights are nonnegative

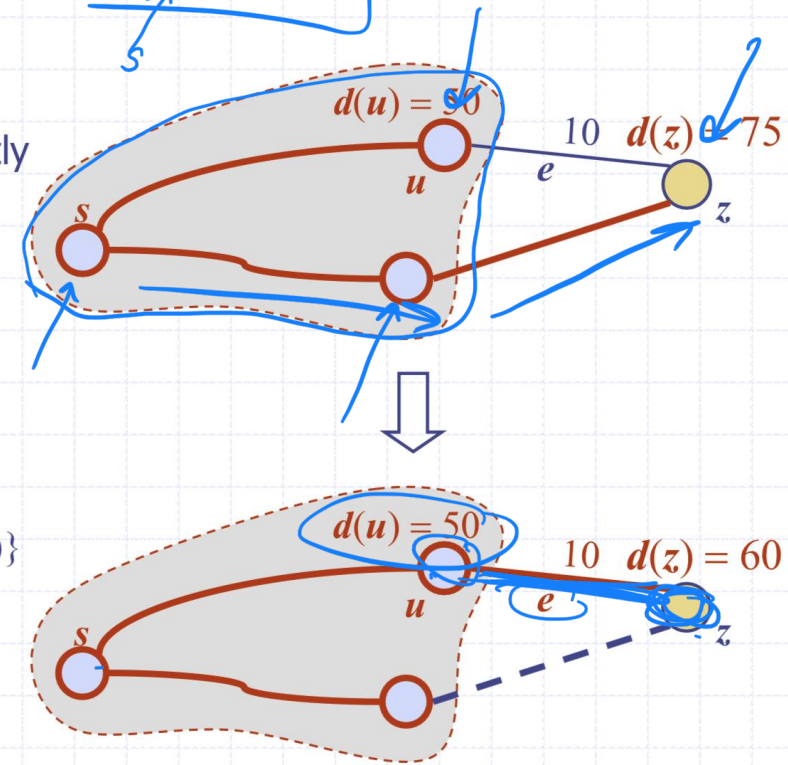
- We grow a "cloud" of vertices beginning with **s** that will eventually cover all vertices.
- We keep track of the current shortest distance from **s** to each vertex in the subgraph made up of the cloud and its adjacent vertices.
- At each step, we consider a new vertex adjacent to the "cloud."

Edge Relaxation



- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Picture from [1]

Algorithm *dijkstrasShortestPath*(G, s) **Input**: a weighted graph $G = (V, E)$ with non-negative edge weights and a source vertex s
Output: the paths with minimum total weight from s to all other vertices in G

```
dist := an array of size |V|
prev := an array of size |V|
Q := a min-heap-based priority queue
dist[s] ← 0
for each vertex v in G:
    if v ≠ s
        dist[v] ← ∞
    prev[v] ← −1
    Q.add(dist[v], v)

while !Q.isEmpty():
    u ← Q.extractMin()
    for each vertex v in G.adj(u) that is still in Q
        alt ← dist[u] + length(u, v)
        if alt < dist[v]
            dist[v] ← alt
            prev[v] ← u
            Q.set(alt, v)

return dist, prev
```

- **dist** array keeps track of the shortest distance from **s** to each vertex
- **prev** array keeps track of the previous vertex on the shortest path to each vertex
- **Q** provides an efficient way to extract the vertex with the current minimum path

Algorithm $dijkstrasShortestPath(G, s)$ **Input**: a weighted graph $G = (V, E)$ with non-negative edge weights and a source vertex s
Output: the paths with minimum total weight from s to all other vertices in G

```
dist := an array of size |V|
prev := an array of size |V|
Q := a min-heap-based priority queue
dist[s] ← 0
for each vertex v in G:
    if v ≠ s
        dist[v] ← ∞
    prev[v] ← −1
    Q.add(dist[v], v)

while !Q.isEmpty():
    u ← Q.extractMin()
    for each vertex v in G.adj(u) that is still in Q
        alt ← dist[u] + length(u, v)
        if alt < dist[v]
            dist[v] ← alt
            prev[v] ← u
            Q.set(alt, v)

return dist, prev
```

- We set the source vertex distance to 0.
- All the other distances are set to INFINITY.
- We set the previous values to -1 (or something else to indicate they are UNDEFINED).
- We add all the vertices into the priority queue with the distances as the keys.

Algorithm $dijkstrasShortestPath(G, s)$ **Input**: a weighted graph $G = (V, E)$ with non-negative edge weights and a source vertex s
Output: the paths with minimum total weight from s to all other vertices in G

```
dist := an array of size |V|
prev := an array of size |V|
Q := a min-heap-based priority queue
dist[s] ← 0
for each vertex v in G:
    if v ≠ s
        dist[v] ← ∞
    prev[v] ← −1
    Q.add(dist[v], v)

while !Q.isEmpty():
    u ← Q.extractMin()
    for each vertex v in G.adj(u) that is still in Q
        alt ← dist[u] + length(u, v)
        if alt < dist[v]
            dist[v] ← alt
            prev[v] ← u
            Q.set(alt, v)

return dist, prev
```

edge relaxation

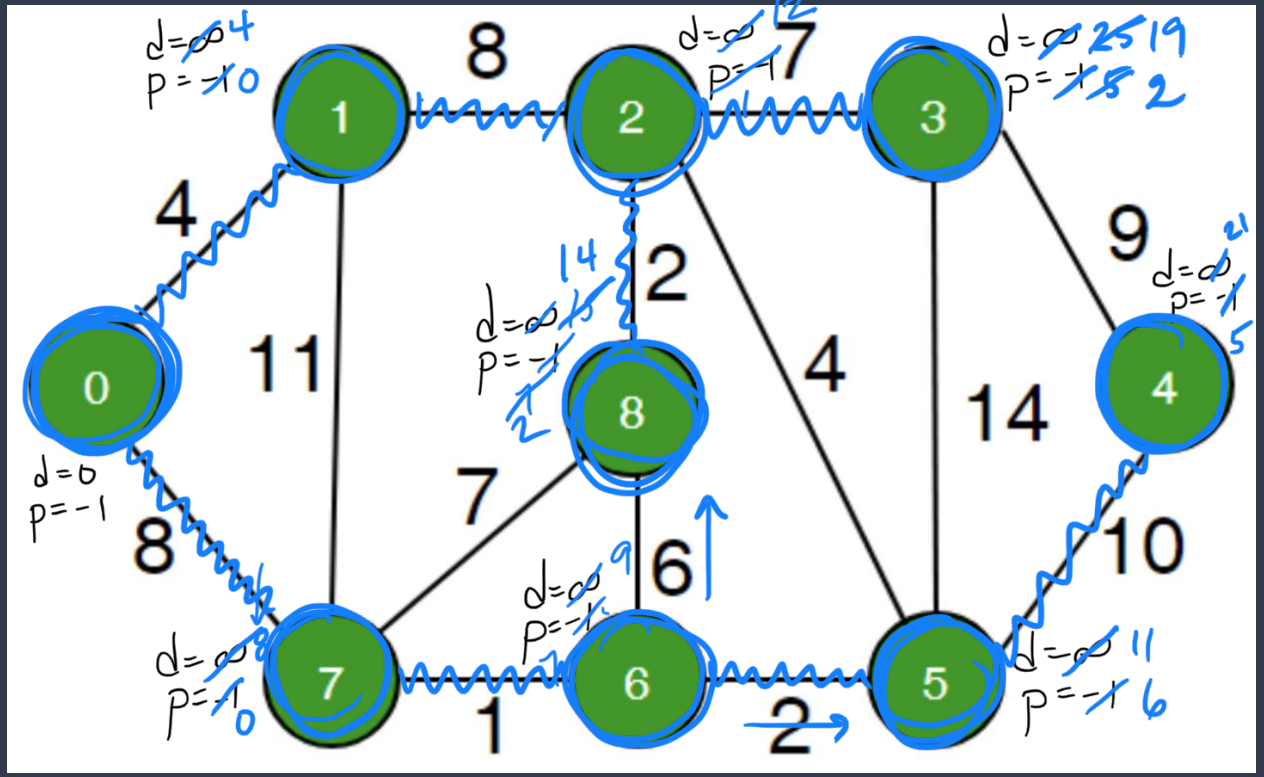
The Main Loop:

Until the priority queue is empty we…

- get the minimum vertex
- check its adjacent vertices to find the “shortest” path to the new vertex
- update **dist**, **prev**, and the queue as necessary

Example. Show Dijkstra's Algorithm on the graph below.

$s = 0$



PQ

~~(0, 0)~~

~~(∞, 1)~~ ~~(4, 1)~~

~~(∞, 2)~~ (12, 2)

(∞, 3)

(∞, 4)

(∞, 5)

~~(∞, 6)~~ (9, 6)

~~(∞, 7)~~ ~~(8, 7)~~

~~(∞, 8)~~ (15, 8)

Analysis: What is the runtime?

Algorithm $dijkstrasShortestPath(G, s)$ **Input**: a weighted graph $G = (V, E)$ with non-negative edge weights and a source vertex s
Output: the paths with minimum total weight from s to all other vertices in G

```
dist := an array of size |V|
prev := an array of size |V|
Q := a min-heap-based priority queue
dist[s] ← 0
for each vertex v in G:
    if v ≠ s
        dist[v] ← ∞
    prev[v] ← −1
    Q.add(dist[v], v)

while !Q.isEmpty():
    u ← Q.extractMin()
    for each vertex v in G.adj(u) that is still in Q
        alt ← dist[u] + length(u, v)
        if alt < dist[v]
            dist[v] ← alt
            prev[v] ← u
            Q.set(alt, v)

return dist, prev
```

- Runtime for setting all the values in the two arrays:
- Runtime for adding all the items to the priority queue:

Algorithm $dijkstrasShortestPath(G, s)$ **Input**: a weighted graph $G = (V, E)$ with non-negative edge weights and a source vertex s
Output: the paths with minimum total weight from s to all other vertices in G

```
dist := an array of size |V|
prev := an array of size |V|
Q := a min-heap-based priority queue
dist[s] ← 0
for each vertex v in G:
    if v ≠ s
        dist[v] ← ∞
    prev[v] ← −1
    Q.add(dist[v], v)

while !Q.isEmpty():
    u ← Q.extractMin()
    for each vertex v in G.adj(u) that is still in Q
        alt ← dist[u] + length(u, v)
        if alt < dist[v]
            dist[v] ← alt
            prev[v] ← u
            Q.set(alt, v)

return dist, prev
```

- Runtime for setting all the values in the two arrays: **O(V)**
- Runtime for adding all the items to the priority queue: **O(VlogV)**

Algorithm $dijkstrasShortestPath(G, s)$ **Input**: a weighted graph $G = (V, E)$ with non-negative edge weights and a source vertex s
Output: the paths with minimum total weight from s to all other vertices in G

```
dist := an array of size |V|
prev := an array of size |V|
Q := a min-heap-based priority queue
dist[s] ← 0
for each vertex v in G:
    if v ≠ s
        dist[v] ← ∞
    prev[v] ← −1
    Q.add(dist[v], v)

while !Q.isEmpty():
    u ← Q.extractMin()
    for each vertex v in G.adj(u) that is still in Q
        alt ← dist[u] + length(u, v)
        if alt < dist[v]
            dist[v] ← alt
            prev[v] ← u
            Q.set(alt, v)

return dist, prev
```

- Runtime for setting all the values in the two arrays: **O(V)**
- Runtime for adding all the items to the priority queue: **O(VlogV)**

- Runtime for resetting values in the two arrays: **O(E)**
- Runtime for resetting values in the queue: **O(ElogV)**

Analysis: What is the runtime?

- In all, the runtime is **O(V) + O(VlogV) + O(E) + O(ElogV) = O((V+E)logV)**
- Since the graph is connected, we can also say **O(ElogV)** since every vertex is connected to at least one edge…

Further explanations:

- At most, the values **dist[v]** and **prev[v]** for any vertex **v** are updated **deg(v)** times, and the sum of all **deg(v)** (for all vertices) is **2E**, which explains the **O(E)** runtime for the updates in the arrays.
- The same idea explains the **O(ElogV)** runtime for the queue updates

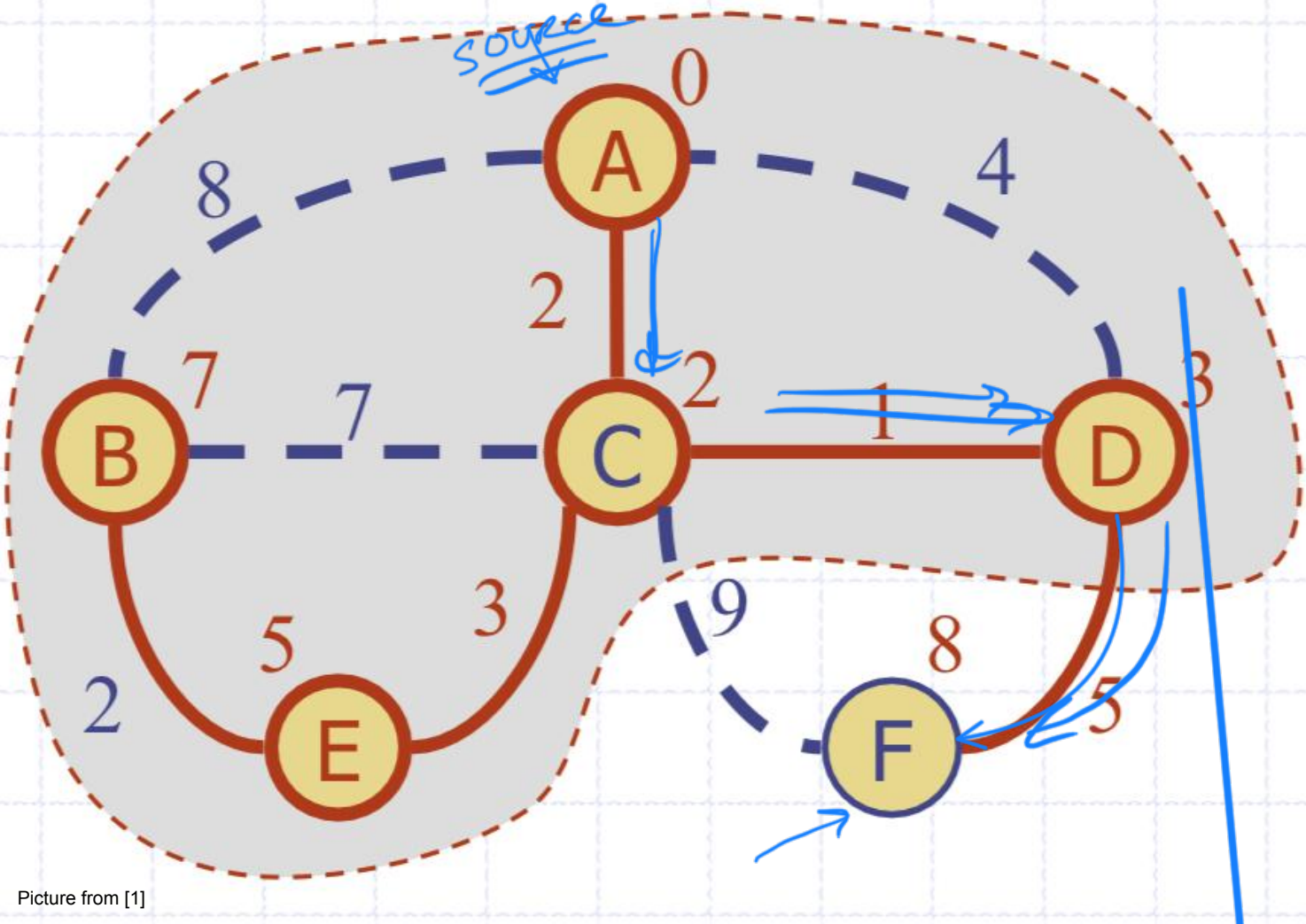
Analysis: Note on Priority Queue Implementation

- The **O((V+E)logV)** runtime depends on the implementation of the algorithm regarding the vertices in the priority queue.
- Recall that a binary heap allows **insert** and **delMin** (for a min-heap) in **O(logV)** time, but **search** in a binary heap is **O(V)** in the worst case.
- This analysis depends upon the method for updating the a key in the priority queue in **O(logV)** time.
- In order to do that, the implementation of the algorithm must maintain a link between vertices and their positions in the queue (e.g. an array of size **V** that gets updated whenever a vertex changes position in the queue).
- This would also allow **O(1)** checking whether a vertex is still in the queue
- When a key gets updated, at most **logV** vertex positions will have to be updated as the heap is rearranged--so updating a key can be done in **O(logV)** time.

Why does Dijkstra's Algorithm work? Proof by Contradiction

Dijkstra's Algorithm is a **greedy** approach to the shortest path problem--it adds nodes by choosing the shortest distances first using the vertices that have been visited.

- Suppose it didn't find all the shortest distances. Suppose the distance it chose from **s** to **F** was NOT the shortest using the previously visited vertices, and **F** is the first vertex chosen incorrectly.
- Let **D** be the vertex previous to **F** on the chosen path. Since **F** was chosen incorrectly, that means that there must be a shorter path to **F** from **s** either using **D** or not using **D**.
  - If that shorter path uses **D**, then the path from **s** to **D** was not the shortest, which is a contradiction because we assumed **F** was the first to be chosen incorrectly.
  - If that shorter path does not use **D**, then the edge from **D** to **F** was not *relaxed*, which is a contradiction because Dijkstra's Algorithm only adds edges once they are relaxed.

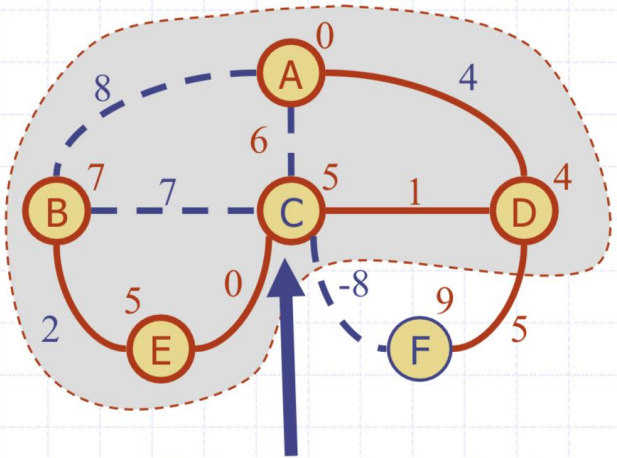


Picture from [1]

Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



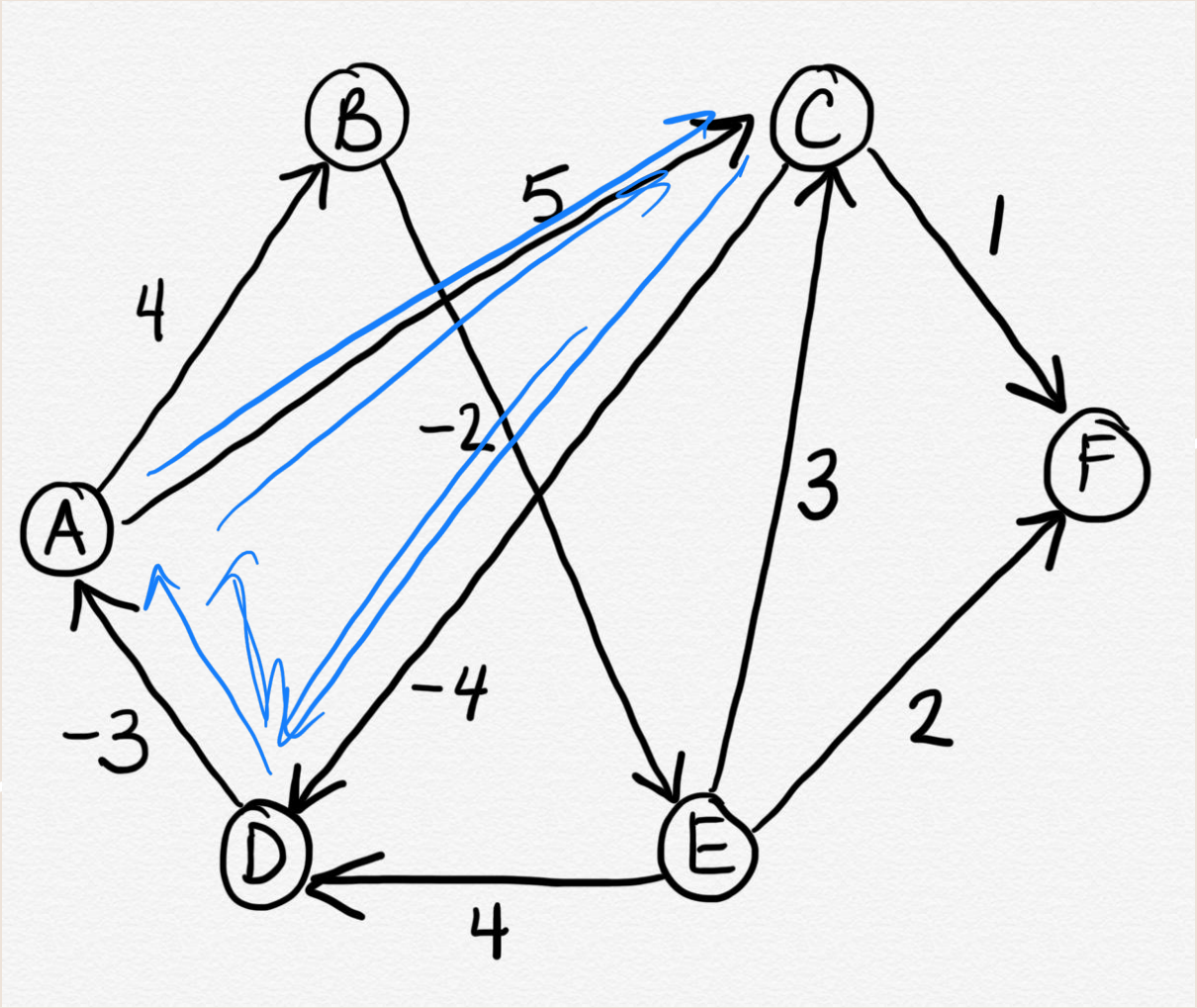
C's true distance is 1, but it is already in the cloud with d(C)=5!

Picture from [1]

Bellman-Ford Shortest Path Algorithm

- Negative Weight Cycles
- Arbitrage Application
- Algorithm Overview and Analysis

Find the shortest path from **A** to **F** in the following graph.


A
B
C
D
E
F
4
5
1
-2
3
-3
-4
2
4

Arbitrage: How to make an easy profit...

In the table, “i buys j.”

Alice has 10,000 USD...

- How many Swiss francs (CHF) can she buy?

- How many Euros (EUR) can she buy?

i/j	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Arbitrage: How to make an easy profit...

In the table, “i buys j.”

Alice has 10,000 USD...

- How many Swiss francs can she buy?

  10,000(1.061) = 10,610 CHF

- How many Euros can she buy?

  10,000(0.741) = 7,410 EUR

i/j	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Arbitrage: How to make an easy profit...

Alice has 10,000 USD...

Alice decides to buy EUR with her USD...

...then she buys CAD with her EUR...

...then she buys USD with her CAD...

How many USD does she have now?

i/j	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Arbitrage: How to make an easy profit...

Alice has 10,000 USD...

Alice decides to buy EUR with her USD...

10,000(0.741) = 7,410 EUR

...then she buys CAD with her EUR...

7,410(1.366) = 10,122.06 CAD

...then she buys USD with her CAD...How many USD does she have now?

10,122.06(0.995) = 10,071.4497 USD

i/j	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Imagine that you have 10,000 USD.

Given the conversion rates on the following slide, determine a cycle of conversions to try to maximize your profit.

i/j	USD	EUR	BOL	BIT	CHY
USD	1	0.89	6.91	0.00026	6.71
EUR	1.12	1	7.76	0.00029	7.55
BOL	0.14	0.13	1	0.000037	0.97
BIT	3874.90	3445.24	26698.71	1	25975.72
CHY	0.15	0.13	1.03	0.000038	1

Reminder: “i buys j”

USD: US Dollars
EUR: Euros
BOL: Bolivian Bolivianos
BIT: Bitcoin
CHY: Chinese Yuan

How would you represent the exchange rates as a graph?

How would you represent the exchange rates as a graph?

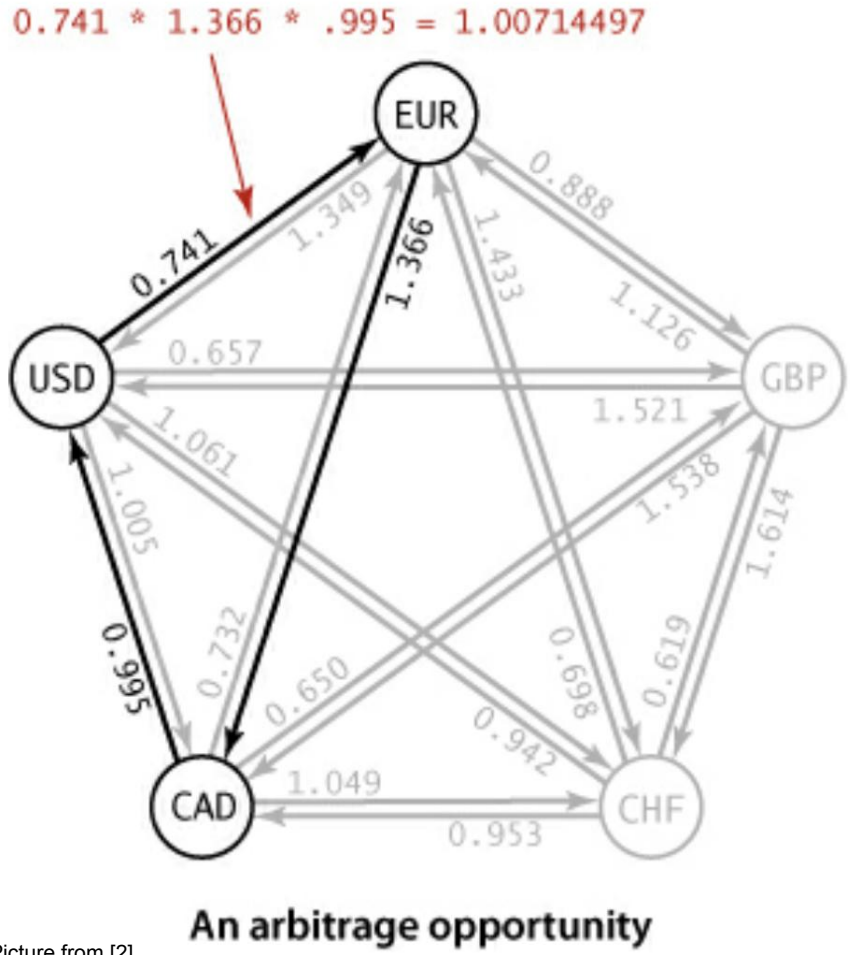
- Use a complete directed graph with the currencies as vertices
- (u, v, w) denotes: “1 u buys w v”
- Example: (USD, BOL, 6.91) is an edge from USD to BOL with weight 6.91 that means “1 USD buys 6.91 BOL”

How can we recognize an arbitrage opportunity with respect to the graph?

How can we recognize an arbitrage opportunity with respect to the graph?

Look for a cycle where the product of the edge weights is greater than 1.

Example from book



An arbitrage opportunity

Picture from [2]

i/j	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

This problem can also be solved as a **negative cycle detection problem**.

But first, let's review some math...

$w \times y > 1$

$\log(w \times y) > \log 1$

$\log w + \log x + \log y > 0$

$-\log w - \log x - \log y < 0$

To convert the arbitrage problem to a negative cycle detection problem:

- Replace each edge weight w with the negative of its logarithm ($-\log w$).
- Now a cycle where $uvw > 1$ becomes a cycle where $-\log u - \log v - \log w < 0$ (a negative-weight cycle).
- Run the Bellman–Ford algorithm to find a negative cycle.

Bellman-Ford Shortest Path

- Works on any weighted digraph.
- Allows negative weight edge values.
- Uses edge relaxation to compute shortest path by considering 1-edge paths, then 2 edge paths,..., then |V|-1-edge paths.
- One final pass through the edges will determine if there are any negative weight cycles.

Bellman-Ford Shortest Path

Algorithm *bellmanFord*(G, s)
Input: Weighted $G = (V, E)$ and source vertex s
Output: The shortest paths from s to all other vertices in G

```
//Initialize array values
dist := an array of size |V|
prev := an array of size |V|
    for all v ∈ V:
        dist[v] = ∞
        prev[v] = −1
    dist[s] = 0

//Relax edges repeatedly
    for i ← 1 to |V| − 1 do
        for each e = (u, v) ∈ E with weight w:
            if dist[u] + w < dist[v]:
                dist[v] := dist[u] + w
                prev[v] := u

//Check for cycles of negative weight
for each e = (u, v) ∈ E with weight w:
    if dist[u] + w < dist[v]:
        error "Graph contains a negative-weight cycle"

return dist[], prev[]
```

- Similar to Dijkstra's, we use two arrays to keep track of the shortest distance to each vertex and the previous vertex on the current path
- Set all distances to INFINITY except the source
- Set all previous values to UNDEFINED (-1 here)

- Like Dijkstra's, B-F uses edge relaxation
- Unlike Dijkstra's, it relaxes all edges, and it repeats the process **O(|V|)** times
- Iteration ***i*** finds all shortest paths that use ***i*** (or more) edges

After the edge relaxation process, if it is still possible to relax an edge, there must be a negative cycle

Analysis

Algorithm *bellmanFord*(G, s)
Input: Weighted $G = (V, E)$ and source vertex s
Output: The shortest paths from s to all other vertices in G

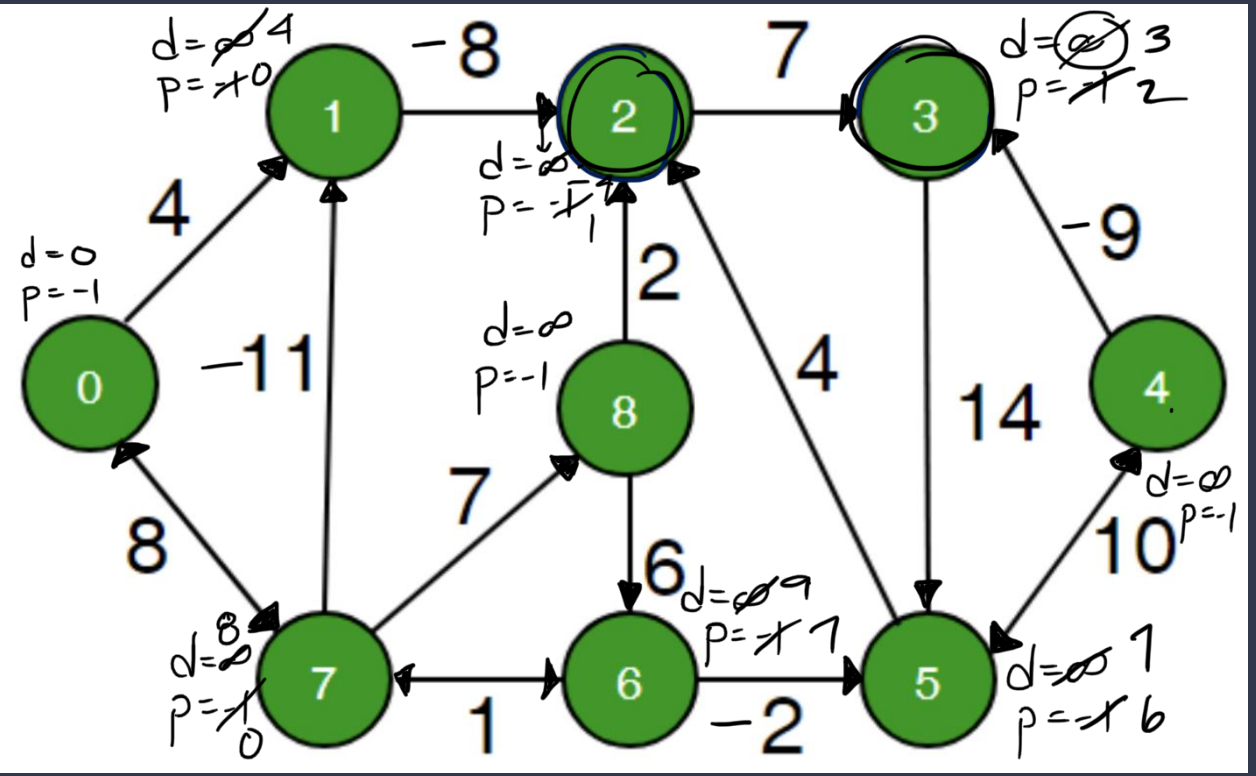
```
//Initialize array values
dist := an array of size |V|
prev := an array of size |V|
    for all v ∈ V:
        dist[v] = ∞
        prev[v] = −1
    dist[s] = 0

//Relax edges repeatedly
    for i ← 1 to |V| − 1 do
        for each e = (u, v) ∈ E with weight w:
            if dist[u] + w < dist[v]:
                dist[v] := dist[u] + w
                prev[v] := u

//Check for cycles of negative weight
for each e = (u, v) ∈ E with weight w:
    if dist[u] + w < dist[v]:
        error "Graph contains a negative-weight cycle"

return dist[], prev[]
```

Setting the values: O(|V|)

Edge Relaxation: Done for all edges |V| - 1 times ⇒ O(|V|*|E|)

After the edge relaxation process, if it is still possible to relax an edge, there must be a negative cycle

Example. Run the Bellman-Ford Algorithm on the following graph.



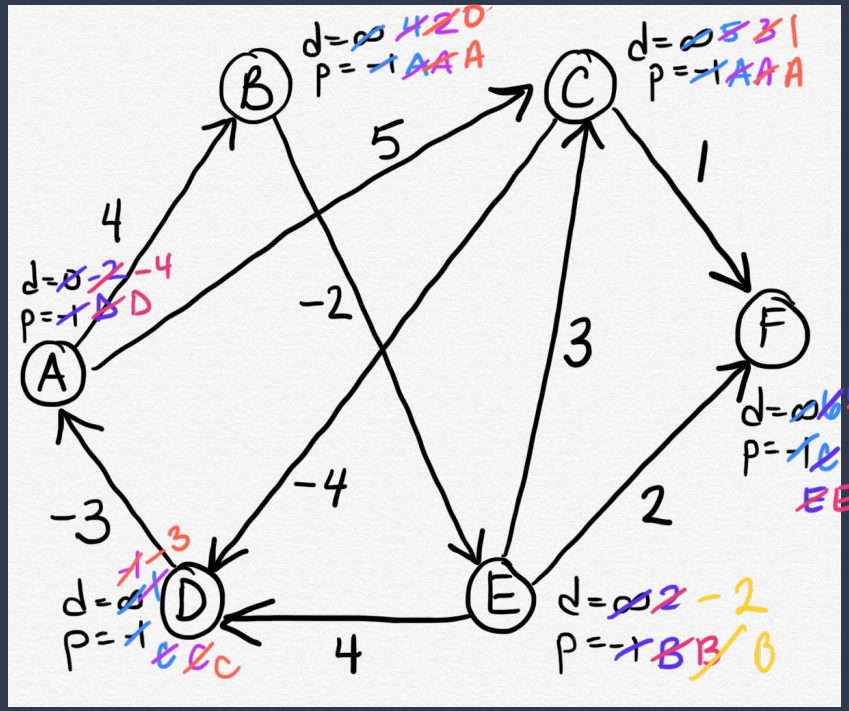
edges

(2, 3) ×✓
(4, 3) ××
(5, 2) ××
(5, 4) ××
(4, 5) ××
(6, 5) ×✓
(6, 7) ×
(7, 8) ×
(7, 1) ×
(7, 0) ×
(0, 1) ✓
(0, 7) ✓
(1, 2) ✓

edges

(3, 5) ×
(7, 6) ✓
(8, 2) ×
(8, 6) ×

Example. Run the Bellman-Ford Algorithm on the following graph.



Edges					
(B, E)	x	✓	x	✓	x
(E, D)	x	x	x	x	x
(E, C)	x	x	x	x	x
(E, F)	x	✓	x	✓	x
(A, B)	✓	x	✓	x	✓
(A, C)	✓	x	✓	x	✓
(D, A)	x	✓	x	✓	x
(C, D)	✓	x	✓	x	✓
(C, F)	✓	x	x	x	x

✓

DAG-based Shortest Path

Given a weighted DAG (directed acyclic graph), come up with an algorithm for finding the shortest path from a given source **s**, and analyze the runtime. (Hint: Think about other DAG-related algorithms)

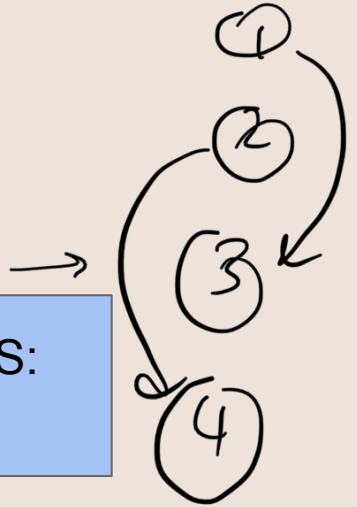
DAG-based Shortest Path

Algorithm *DAGDistances*(G, s)
Input: $G = (V, E)$, a weighted, directed, acyclic graph and s, the source vertex
Output: the paths with minimum total weight from s to all other vertices in G

```
dist[] := an array of size |V|
prev[] := an array of size |V|
for each vertex v ∈ V:
    dist[v] = ∞
    prev[v] = −1
dist[s] = 0

perform a topological sort of the vertices,
assigning values 1 to n = |V|

for u ← 1 to n do: //in topological order
    for each edge e = (u, v) with weight w going out
    of u do:
        //relax edge e
        if dist[u] + w < dist[v]:
            dist[v] := dist[u] + w
            prev[v] := u

return prev[], dist[]
```

Setting the values: **O(|V|)**

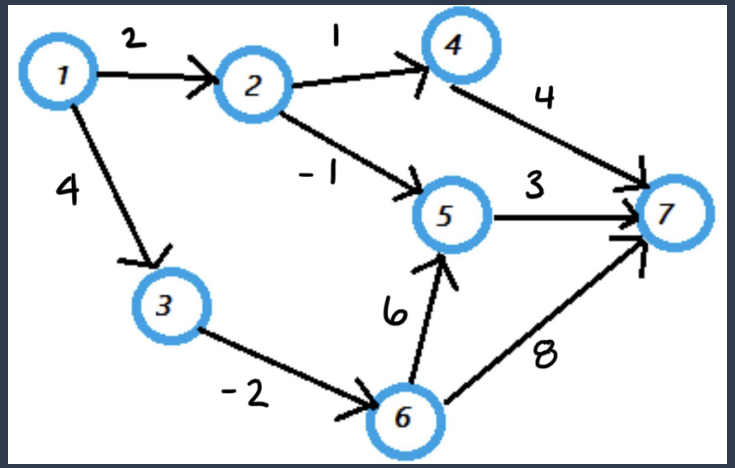
Topological sort with DFS: **O(|V| + |E|)**



Relaxation of edges: **O(|V| + |E|)**

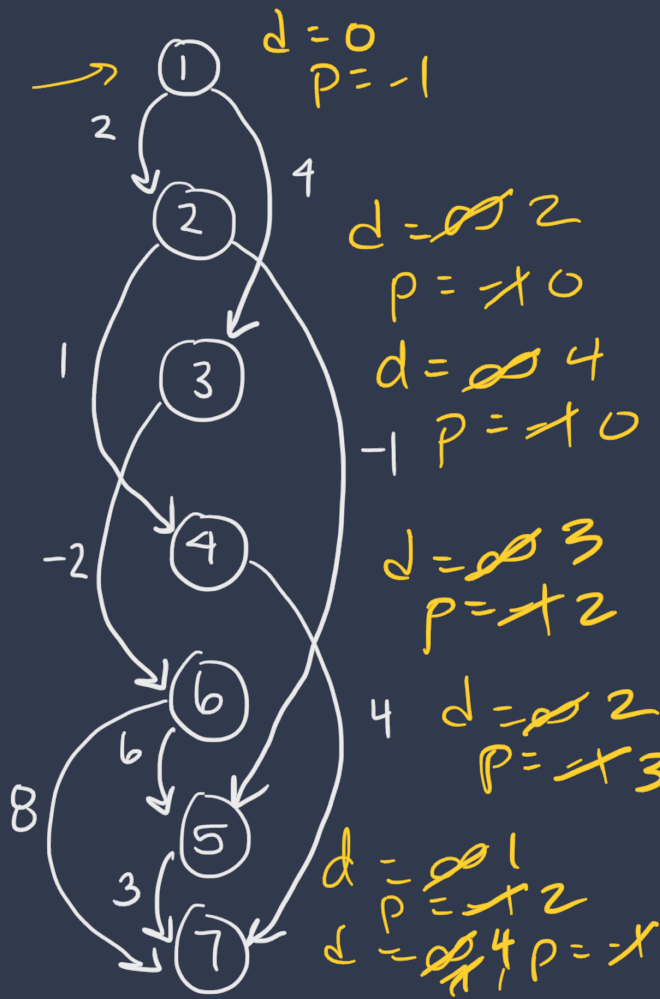
Total: **O(|V| + |E|)**

Example. Run the DAG-based Shortest Path Algorithm on the following graph.



d	
1	0
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

p	
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1



All-Pairs Shortest Path

Design an algorithm for determining the distance (shortest path) between every pair of vertices in a weighted digraph G.

Analyze the runtime. Let **n** be the number of vertices and **m** be the number of edges.

Option 1: Run Dijkstra's algorithm **n** times, each time with a different vertex as the source. Runtime: **O(EVlogV)**

Option 2: Run Bellman-Ford **V** times, each time with a different vertex as the source. Runtime: **O(V²E)**

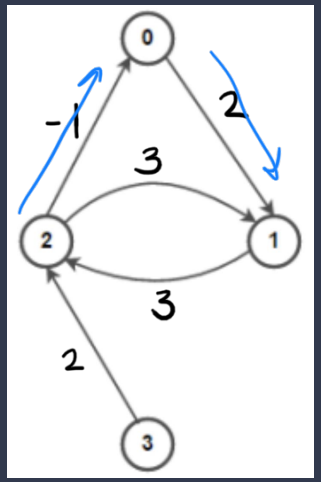
Option 3: Use dynamic programming and design an algorithm similar to Floyd-Warshall's transitive closure algorithm

Runtime: **$O(V^3)$**

Algorithm *AllPair*(G) {assumes vertices $1,\ldots,n$}

```
for all vertex pairs (i,j)
    if i = j
        D_0[i,i] ← 0
    else if (i,j) is an edge in G
        D_0[i,j] ← weight of edge (i,j)
    else
        D_0[i,j] ← +∞
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D_k[i,j] ← min{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]}
return D_n
```

Example. Run the All-Pairs Shortest Path Algorithm on the following graph.



	0	1	2	3
0	0	2	∞	∞
1	∞	0	3	∞
2	-1	3	0	∞
3	∞	∞	2	0

-1+2

	0	1	2	3
0	0	2	∞	∞
1	∞	0	3	∞
2	-1	1	0	∞
3	∞	∞	2	0

	0	1	2	3
0	0	2	5	∞
1	∞	0	3	∞
2	-1	1	0	∞
3	∞	∞	2	0

2+3=

2+

	0	1	2	3
0				
1				
2				
3				

	0	1	2	3
0				
1				
2				
3				

References

1. Goodrich & Tamassia
2. Sedgewick & Wayne

DFS
BFS
transitive closure
Strong conn.
topological ordering
Shortest path