CSE100: Design and Analysis of Algorithms
Lecture 20 – Weighted Graphs (wrap up) and Dynamic Programming

Apr 7th 2022

Dijkstra, Bellman-Ford and Floyd-Warshall

Dijkstra's algorithm (review)

Dijkstra(G,s):

- Set all vertices to **not-sure**
- d[v] = ∞ for all v in V
- d[s] = 0
- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node u with the smallest estimate **d[u].**
  - **For** v in u.neighbors:
    - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**.
- Now d(s, v) = d[v]

Lots of implementation details left un-explained.
We'll get to that!

As usual

- Does it work?
  - Yes.

- Is it fast?
  - Depends on how you implement it.

Running time?

Dijkstra(G,s):

- Set all vertices to **not-sure**
- d[v] = ∞ for all v in V
- d[s] = 0
- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node u with the smallest estimate **d[u].**
  - **For** v in u.neighbors:
    - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**.
- Now dist(s, v) = d[v]

- n iterations (one per vertex)
- How long does one iteration take?

Depends on how we implement it…

We need a data structure that:

- Stores unsure vertices v
- Keeps track of d[v]
- Can find u with minimum d[u]
    - `findMin()`
- Can remove that u
    - `removeMin(u)`
- Can update (decrease) d[v]
    - `updateKey(v,d)`

Just the inner loop:

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
    - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.

Total running time is big-oh of:

$$\sum_{u \in V} \left(T(\text{findMin}) + \left(\sum_{v \in u.neighbors} T(\text{updateKey}) \right) + T(\text{removeMin}) \right)$$

$= n(\ T(\texttt{findMin}) + T(\texttt{removeMin})\ ) + m\ T(\texttt{updateKey})$

If we use an array

- T(findMin) = O(n)
- T(removeMin) = O(n)
- T(updateKey) = O(1)

- Running time of Dijkstra

$$= O(n(\, T(\texttt{findMin}) + T(\texttt{removeMin})\,) + m\, T(\texttt{updateKey}))$$

$$= O(n^2) + O(m)$$

$$= O(n^2)$$

If we use a red-black tree

- $T(\text{findMin}) = O(\log(n))$
- $T(\text{removeMin}) = O(\log(n))$
- $T(\text{updateKey}) = O(\log(n))$

- Running time of Dijkstra

$$= O(n(\,T(\texttt{findMin}) + T(\texttt{removeMin})\,) + m\,T(\texttt{updateKey}))$$

$$= O(n\log(n)) + O(m\log(n))$$

$$= O((n + m)\log(n))$$

Better than an array if the graph is sparse!

aka if m is much smaller than n^2

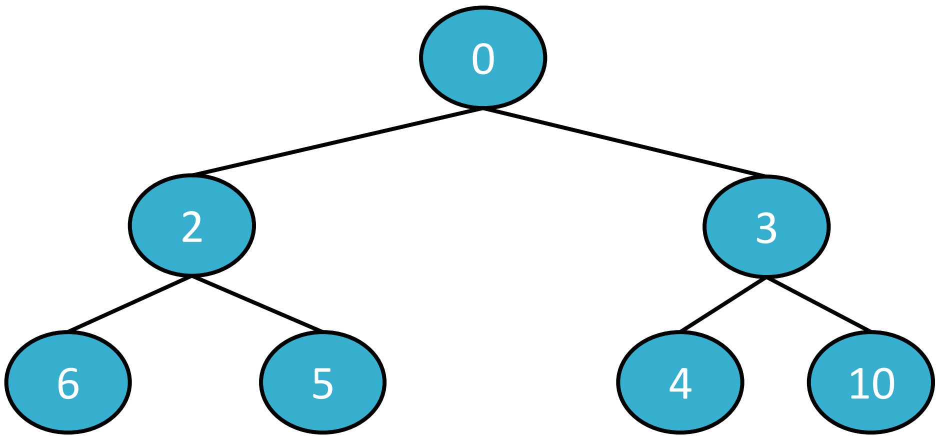
O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))

Is a hash table a good idea here?

- **Not really**:
  - `Search`(v) is fast (in expectation)
  - But `findMin`() will still take time O(n) without more structure.

Heaps support these operations

- T(findMin)
- T(removeMin)
- T(updateKey)



- A **heap** is a tree-based data structure that has the property that every node has a smaller key than its children.
- Review previous lecture that we covered heaps!
- We will use them.

Many heap implementations

Nice chart on Wikipedia:

Operation	Binary[7]	Leftist	Binomial[7]	Fibonacci[7][8]	Pairing[9]	Brodal[10][b]	Rank-pairing[12]	Strict Fibonacci[13]
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$
insert	$O(\log n)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$[c]	$o(\log n)$[c][d]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
merge	$\Theta(n)$	$\Theta(\log n)$	$O(\log n)$[e]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Say we use a **Fibonacci Heap**

- T(findMin) = O(1) (amortized time*)
- T(removeMin) = O(log(n)) (amortized time*)
- T(updateKey) = O(1) (amortized time*)
- See CLRS for more!

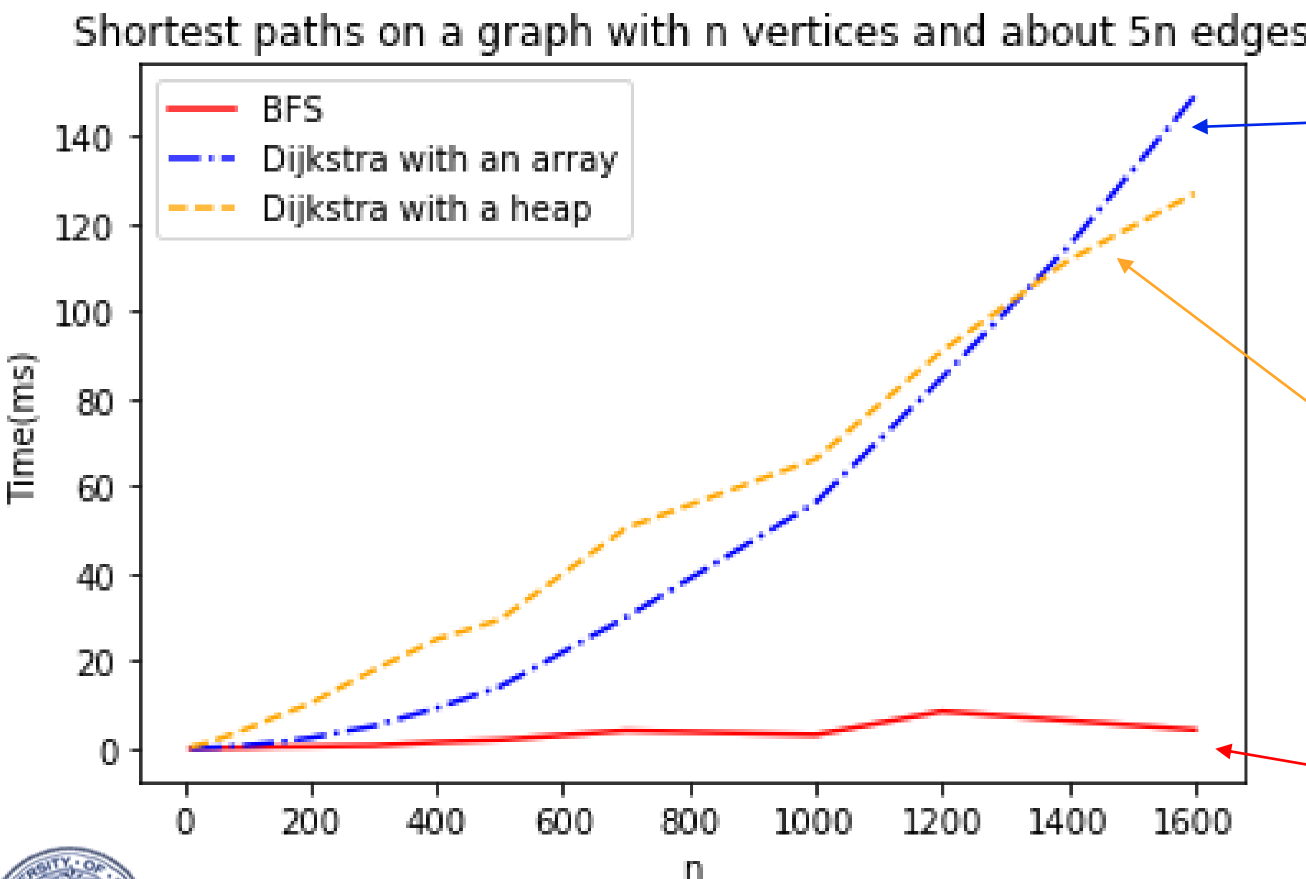
- Running time of Dijkstra

  = O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))

  = O(nlog(n) + m) (amortized time)

*This means that any sequence of d `removeMin` calls takes time at most O(dlog(n)). But a few of the d may take longer than O(log(n)) and some may take less time..

In practice



Dijkstra using a Python list to keep track of vertices has quadratic runtime.

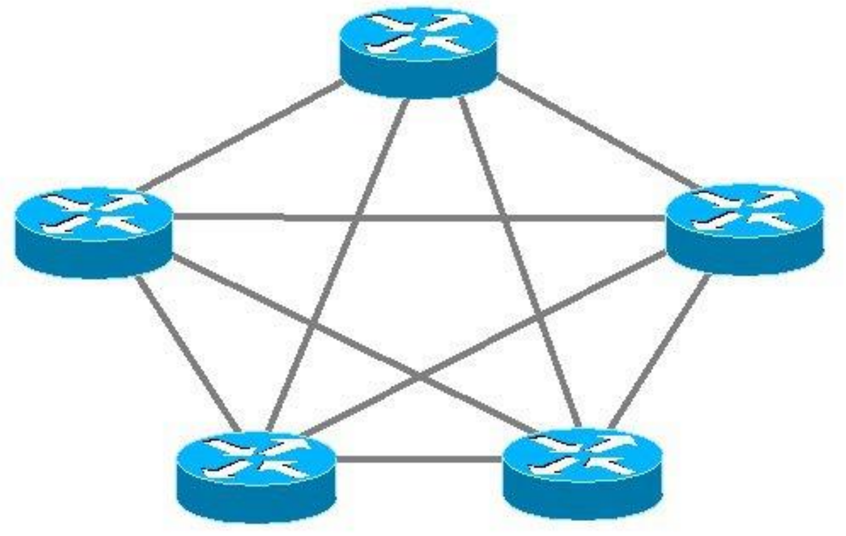
Dijkstra using a heap looks a bit more linear (actually nlog(n))

BFS is really fast by comparison! But it doesn't work on weighted graphs.

Dijkstra is used in practice

- eg, OSPF (Open Shortest Path First), a routing protocol for IP networks, uses Dijkstra.

But there are some things it's not so good at.



Dijkstra Drawbacks

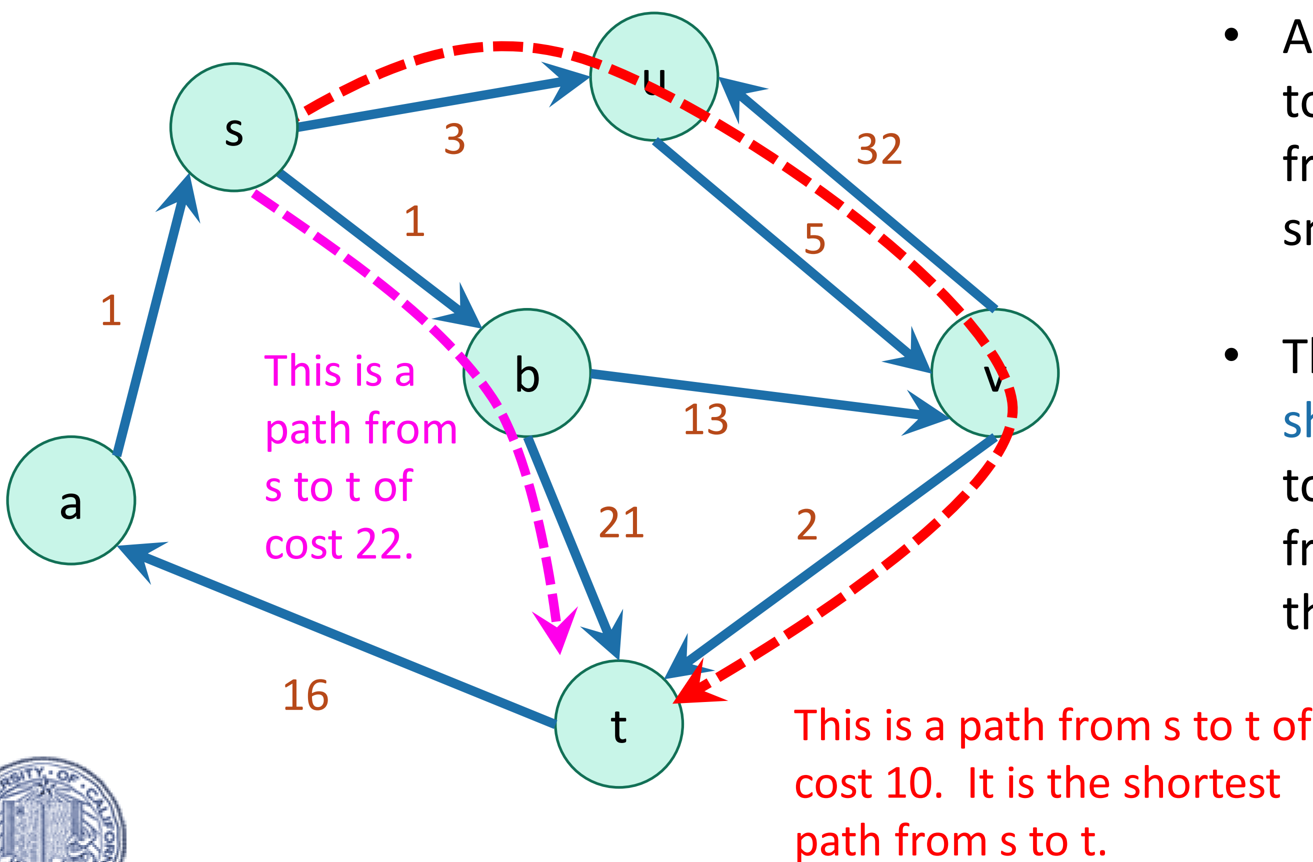
- Needs non-negative edge weights.
- If the weights change, we need to re-run the whole thing.
    - in OSPF, a vertex broadcasts any changes to the network, and then every vertex re-runs Dijkstra's algorithm from scratch.

Rest of Today

- Bellman-Ford
- Bellman-Ford is a special case of ***Dynamic Programming!***
- What is dynamic programming?
  - Warm-up example: Fibonacci numbers
- Another example:
  - Floyd-Warshall Algorithm

Recall

- A weighted directed graph:



- Weights on edges represent costs.
- The cost of a path is the sum of the weights along that path.
- A shortest path from s to t is a directed path from s to t with the smallest cost.
- The single-source shortest path problem is to find the shortest path from s to v for all v in the graph.

Bellman-Ford algorithm

- (-) Slower than Dijkstra's algorithm
- (+) Can handle negative edge weights.
  - Can be useful if you want to say that some edges are actively good to take, rather than costly.
  - Can be useful as a building block in other algorithms.
- (+) Allows for some flexibility if the weights change.
  - We'll see what this means later

- Basic idea:
  - Instead of picking the u with the smallest d[u] to update, just update all of the u's simultaneously

Bellman-Ford algorithm

Bellman-Ford(G,s):

- $d[v] = \infty$ for all v in V
- $d[s] = 0$
- **For** i=0,…,n-1:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d[v] \leftarrow \min(d[v] , d[u] + \text{edgeWeight}(u,v))$

Instead of picking u cleverly, just update for all of the u's.

Compare to Dijkstra:

- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node u with the smallest estimate **d[u].**
  - **For** v in u.neighbors:
    - $d[v] \leftarrow \min(d[v] , d[u] + \text{edgeWeight}(u,v))$
  - Mark u as **sure**.

For pedagogical reasons
which we will see later

- We are actually going to change this to be less smart.
- Keep n arrays: $d^{(0)}, d^{(1)}, \ldots, d^{(n-1)}$

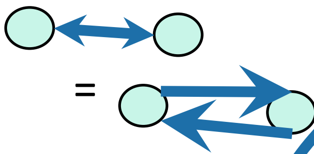
Bellman-Ford*(G,s):

Slightly different than the original Bellman-Ford algorithm, but the analysis is basically the same.

- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
- Then $\text{dist}(s,v) = d^{(n-1)}[v]$

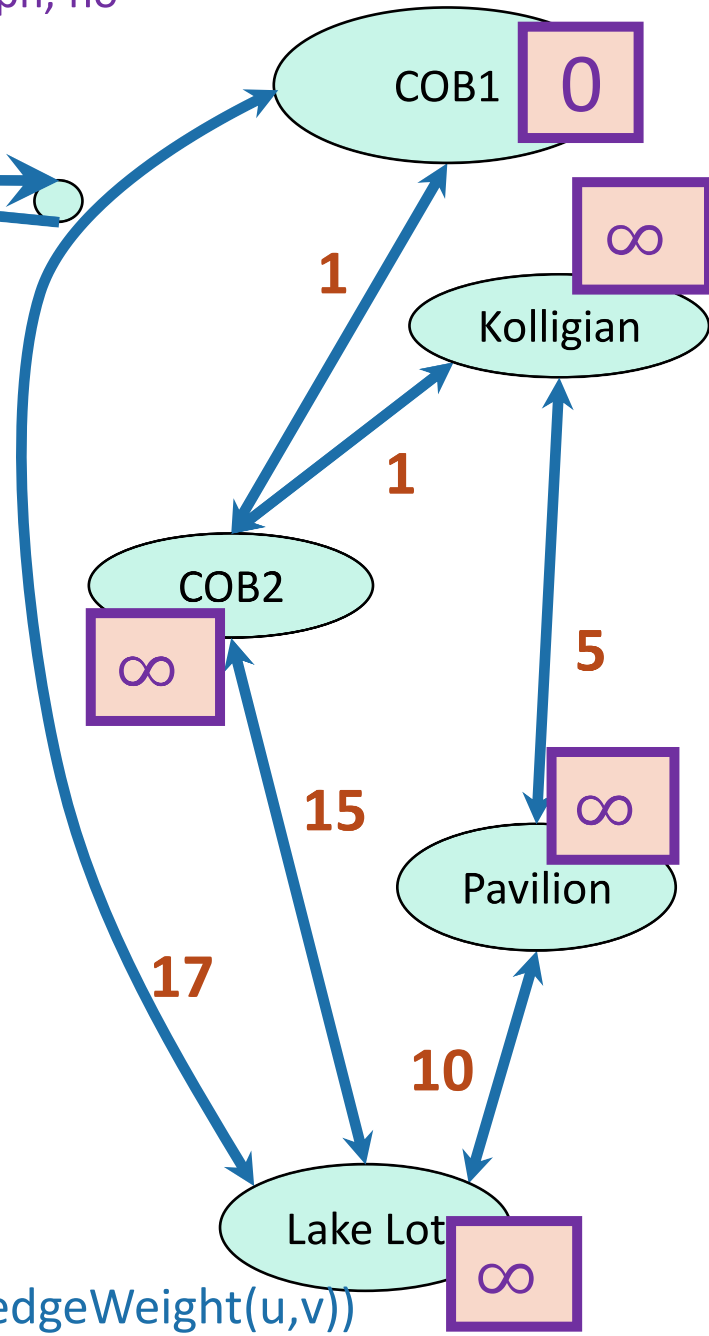
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from COB1?

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	∞	∞	∞	∞	∞
$d^{(2)}$	∞	∞	∞	∞	∞
$d^{(3)}$	∞	∞	∞	∞	∞
$d^{(4)}$	∞	∞	∞	∞	∞

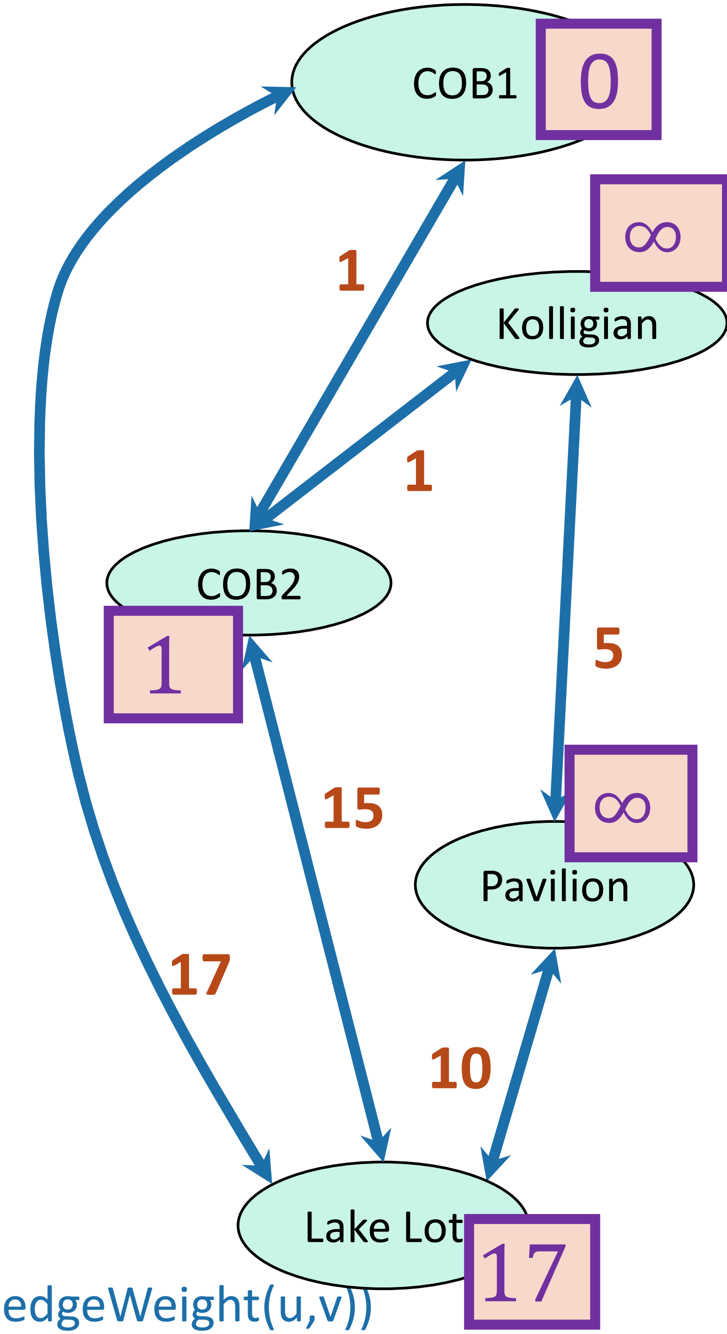


- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

How far is a node from COB1?

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**17**
$d^{(2)}$	∞	∞	∞	∞	∞
$d^{(3)}$	∞	∞	∞	∞	∞
$d^{(4)}$	∞	∞	∞	∞	∞

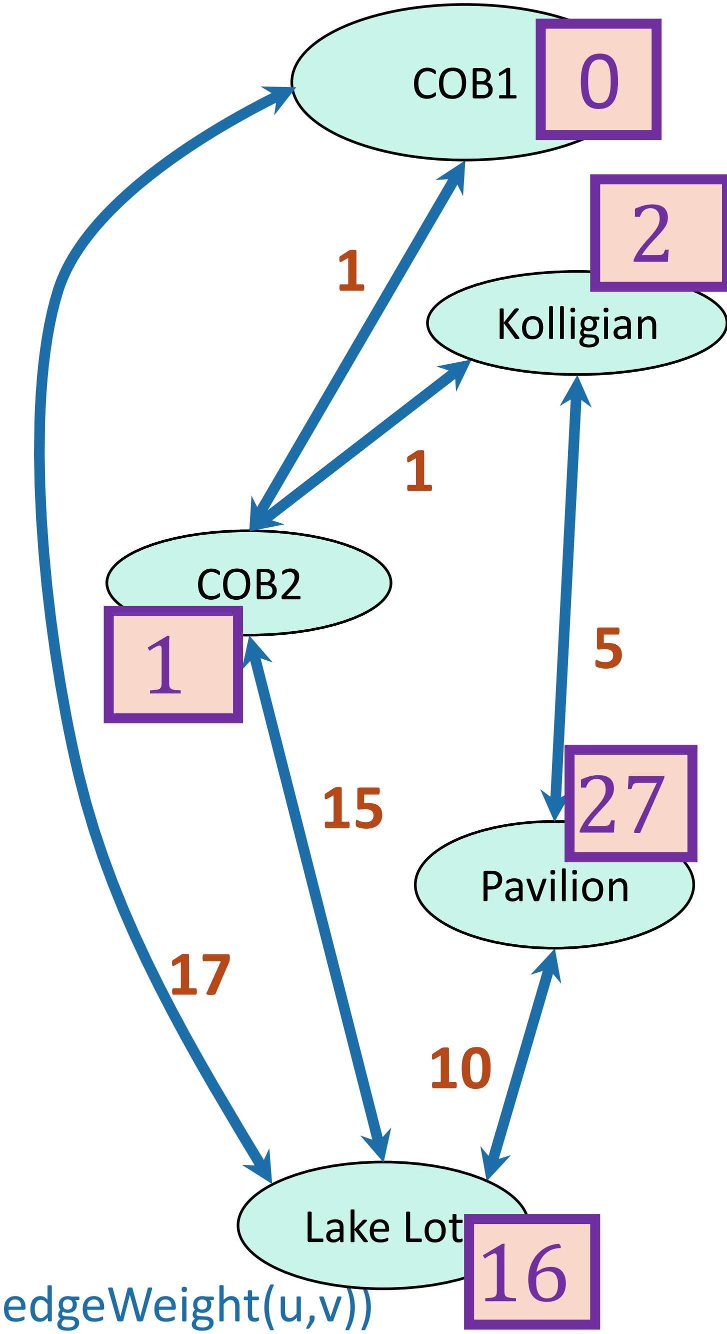


- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

How far is a node from COB1?

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**17**
$d^{(2)}$	0	1	**2**	**27**	**16**
$d^{(3)}$	∞	∞	∞	∞	∞
$d^{(4)}$	∞	∞	∞	∞	∞

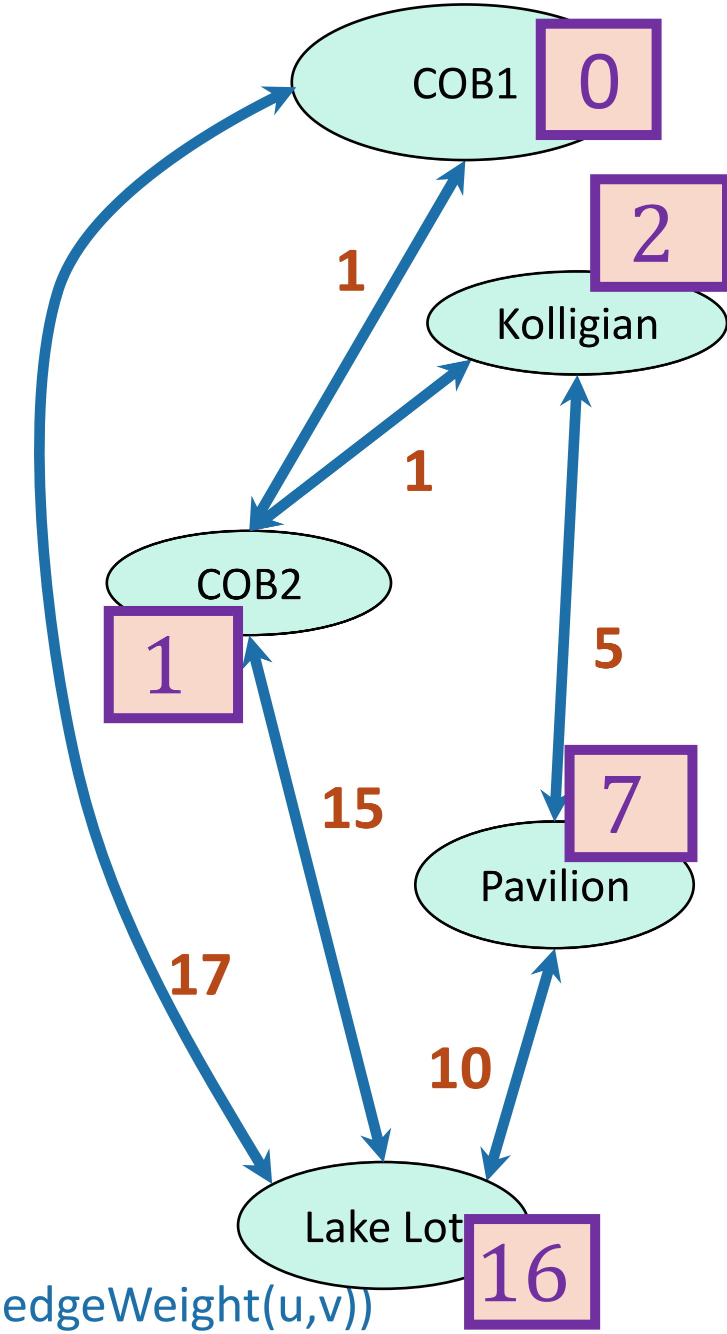


- **For** i=0,…,n-2:
    - **For** u in V:
        - **For** v in u.neighbors:
            - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

How far is a node from COB1?

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**17**
$d^{(2)}$	0	1	**2**	**27**	**16**
$d^{(3)}$	0	1	2	**7**	16
$d^{(4)}$	∞	∞	∞	∞	∞



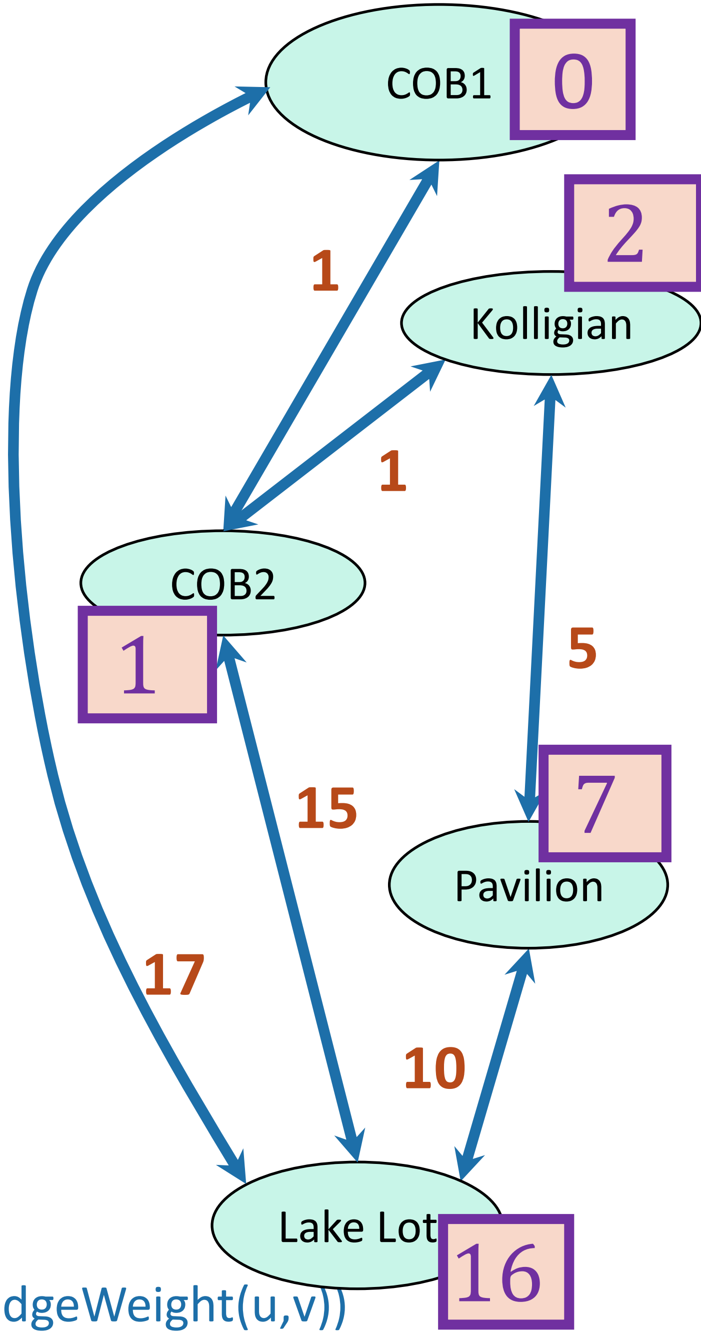
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

How far is a node from COB1?

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**17**
$d^{(2)}$	0	1	**2**	**27**	**16**
$d^{(3)}$	0	1	2	**7**	16
$d^{(4)}$	0	1	2	7	16

These are the final distances!

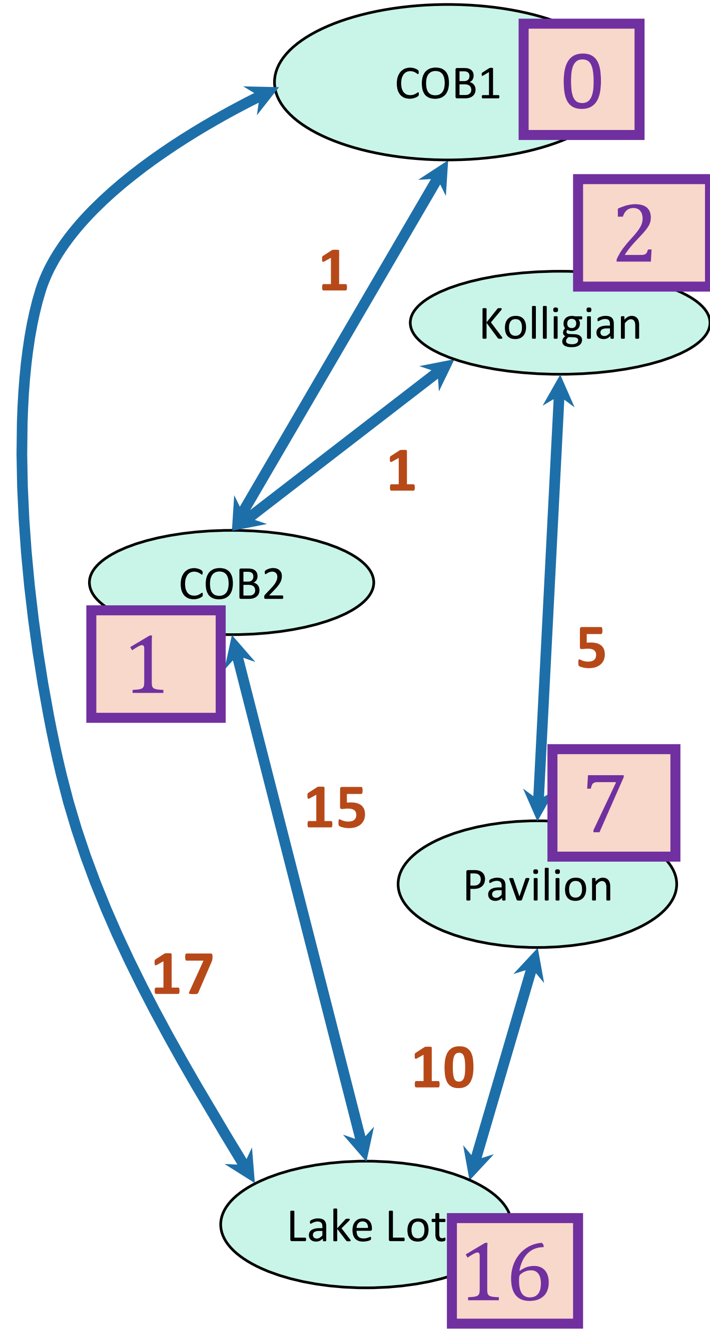


- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$

Interpretation of $d^{(i)}$

$d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**17**
$d^{(2)}$	0	1	**2**	**27**	**16**
$d^{(3)}$	0	1	2	**7**	16
$d^{(4)}$	0	1	2	7	16



Why does Bellman-Ford work?

- Inductive hypothesis:
  - $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.
- Conclusion:
  - $d^{(n-1)}[v]$ is equal to the cost of the shortest path between s and v **with at most n-1 edges**.

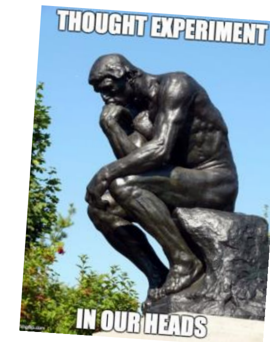
Inductive step

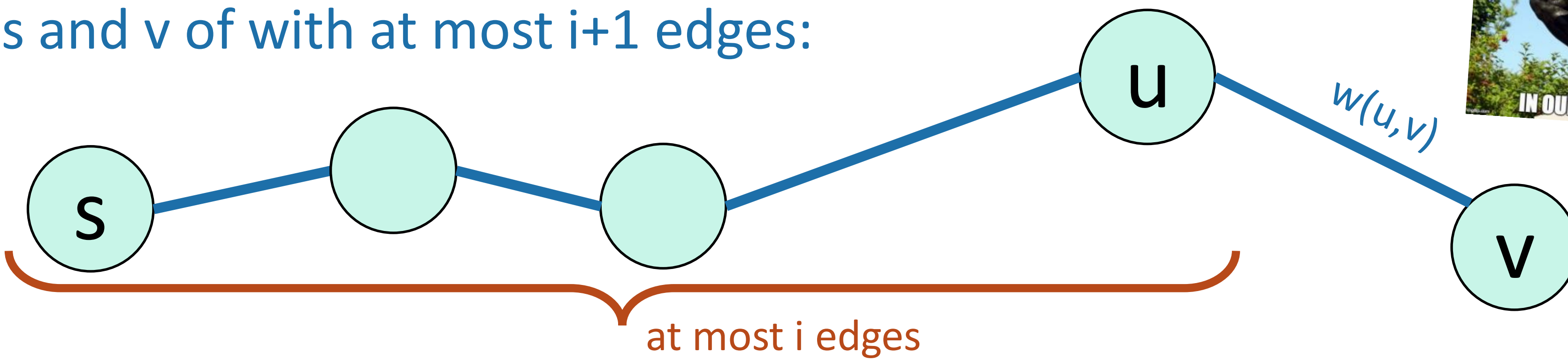
Hypothesis: After iteration i, for each v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v with at most i edges.

- Suppose the inductive hypothesis holds for i.
- We want to establish it for i+1.

Say this is the shortest path between s and v of with at most i+1 edges:

Let u be the vertex right before v in this path.





- By induction, $d^{(i)}[u]$ is the cost of a shortest path between s and u of i edges.
- By setup, $d^{(i)}[u] + w(u,v)$ is the cost of a shortest path between s and v of i+1 edges.
- In the i+1'st iteration, we ensure **$d^{(i+1)}[v] <= d^{(i)}[u] + w(u,v)$.**
- So $d^{(i+1)}[v]$ <= cost of shortest path between s and v with i+1 edges.
- But $d^{(i+1)}[v]$ = cost of a particular path of at most i+1 edges >= cost of shortest path.
- So $d^{(i+1)}[v]$ = cost of shortest path with at most i+1 edges.

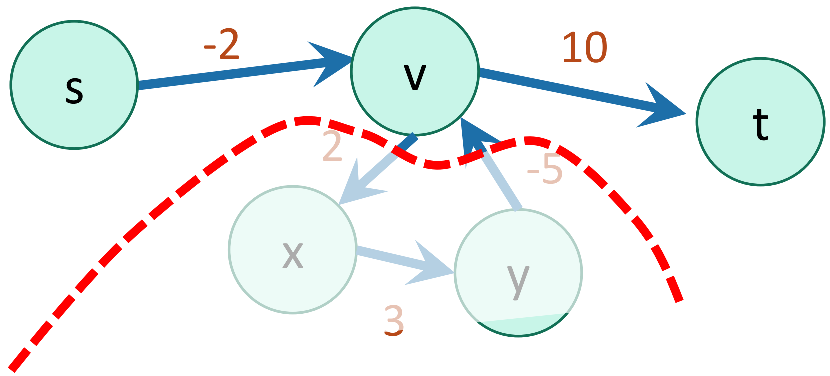
Proof by induction

- **Inductive Hypothesis:**
  - After iteration i, for each v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v of length at most i edges.
- **Base case:**
  - After iteration 0…
- **Inductive step:**
- **Conclusion:**
  - After iteration n-1, for each v, d[v] is equal to the cost of the shortest path between s and v of length at most n-1 edges.
  - **Aka, d[v] = d(s,v) for all v** as long as there are no cycles!

Aside: simple paths

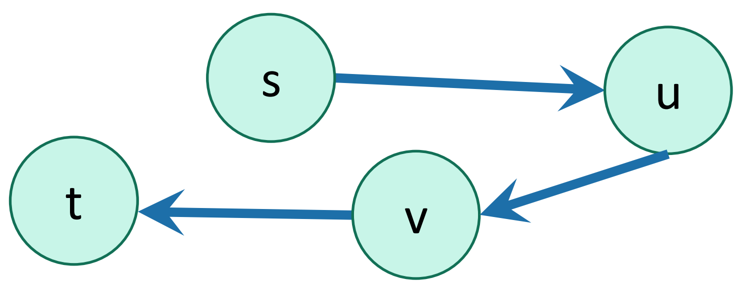
Assume there is no negative cycle.

- Then there is a shortest path from s to t, and moreover there is a simple shortest path.



This cycle isn't helping.
Just get rid of it.

- A simple path in a graph with n vertices has at most n-1 edges in it.



Can't add another edge without making a cycle!

"Simple" means that the path has no cycles in it.

- So there is a shortest path with at most n-1 edges

Why does it work?

- Inductive hypothesis:
  - $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.
- Conclusion:
  - $d^{(n-1)}[v]$ is equal to the cost of the shortest path between s and v **with at most n-1 edges**.
  - **If there are no negative cycles**, $d^{(n-1)}[v]$ is equal to the cost of the shortest path.

Notice that negative edge weights are fine.
Just not negative cycles.

G = (V,E) is a graph with n vertices and m edges.

Bellman-Ford* algorithm

Bellman-Ford*(G,s):

- Initialize arrays $d^{(0)},\ldots,d^{(n-1)}$ of length n to be all ∞
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.outNeighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i+1)}[v] , d^{(i)}[u] + w(u,v))$
- Now, dist(s,v) = $d^{(n-1)}[v]$ for all v in V.
  - (Assuming G has no negative cycles)

Here, Dijkstra picked a special vertex u – Bellman-Ford will just look at all the vertices u.

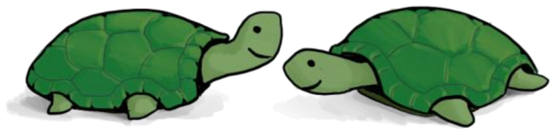
*Slightly different than some versions of Bellman-Ford…but this way is pedagogically convenient for today's lecture.

We can simplify the pseudocode a bit

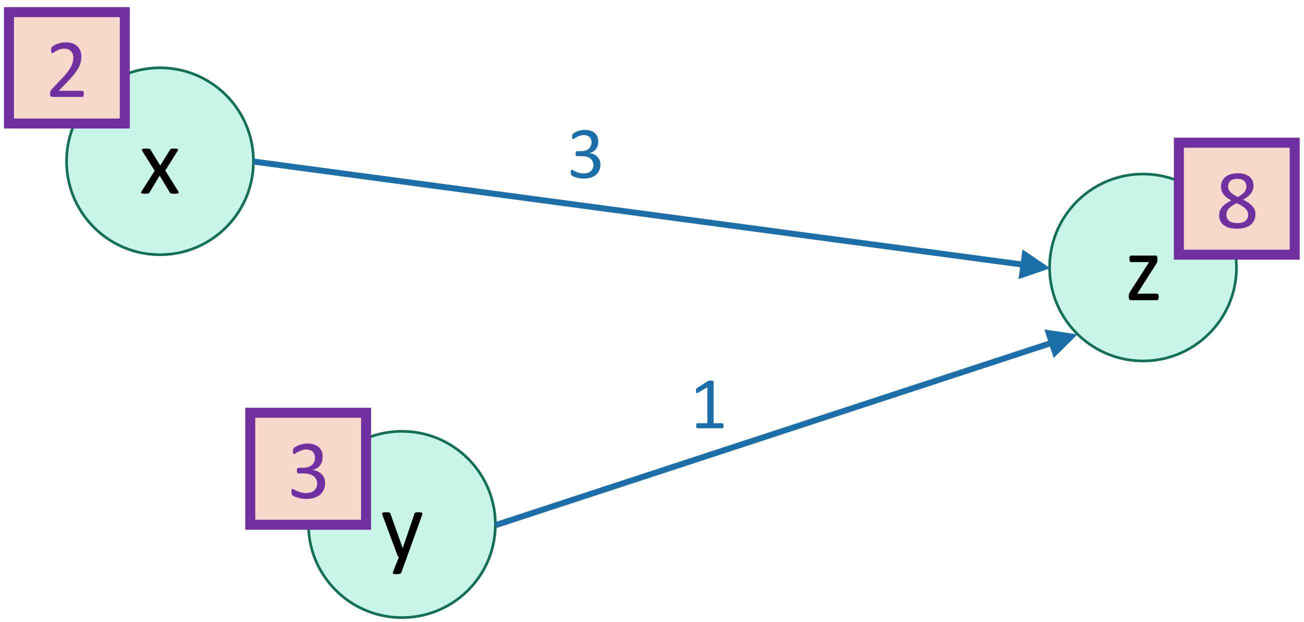
- This will be useful later…

One step of Bellman-Ford

What will happen to z if we run these for-loops?

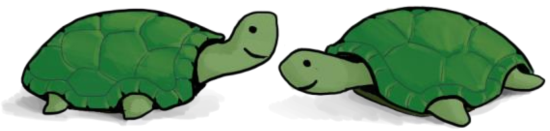


- **For** u in V:
  - **For** v in u.outNeighbors:
    - $d^{(i+1)}[v] \leftarrow \min(\, d^{(i)}[v]\, ,\, d^{(i+1)}[v]\, ,\, d^{(i)}[u] + w(u,v))$

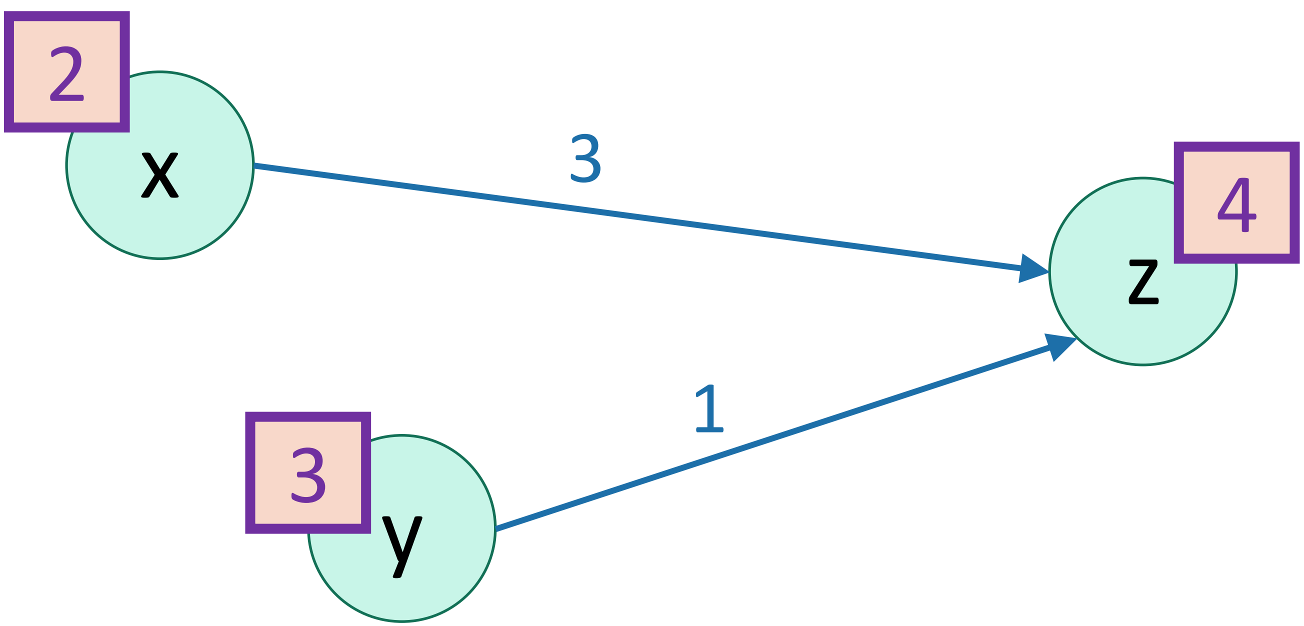


One step of Bellman-Ford

What will happen to z if we run these for loops?



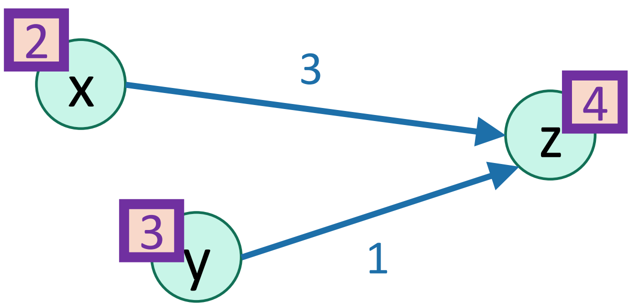
- **For** u in V:
  - **For** v in u.outNeighbors:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i+1)}[v] , d^{(i)}[u] + w(u,v))$



One step of Bellman-Ford

- **For** u in V:
  - **For** v in u.outNeighbors:
    - $d^{(i+1)}[v] \leftarrow \min(\ d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + w(u,v))$

- Each vertex z finds the in-neighbor u so that $d^{(i)}[u] + w(u,z)$ is smallest and goes with that.
- (Unless z chooses not to update).
- So we can equivalently write:



- **For** z in V:
  - $d^{(i+1)}[z] \leftarrow \min(\ d^{(i)}[z]\ ,\ \min_{u \text{ in } z.\text{inNbrs}}\{d^{(i)}[u] + w(u,z)\}\ )$

G = (V,E) is a graph with n vertices and m edges.

Bellman-Ford* algorithm

Bellman-Ford*(G,s):

- Initialize arrays $d^{(0)},\ldots,d^{(n-1)}$ of length n
- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\})$
- Now, dist(s,v) = $d^{(n-1)}[v]$ for all v in V.

*Slightly different than some versions of Bellman-Ford…but this way is pedagogically convenient for today's lecture.

Note on implementation

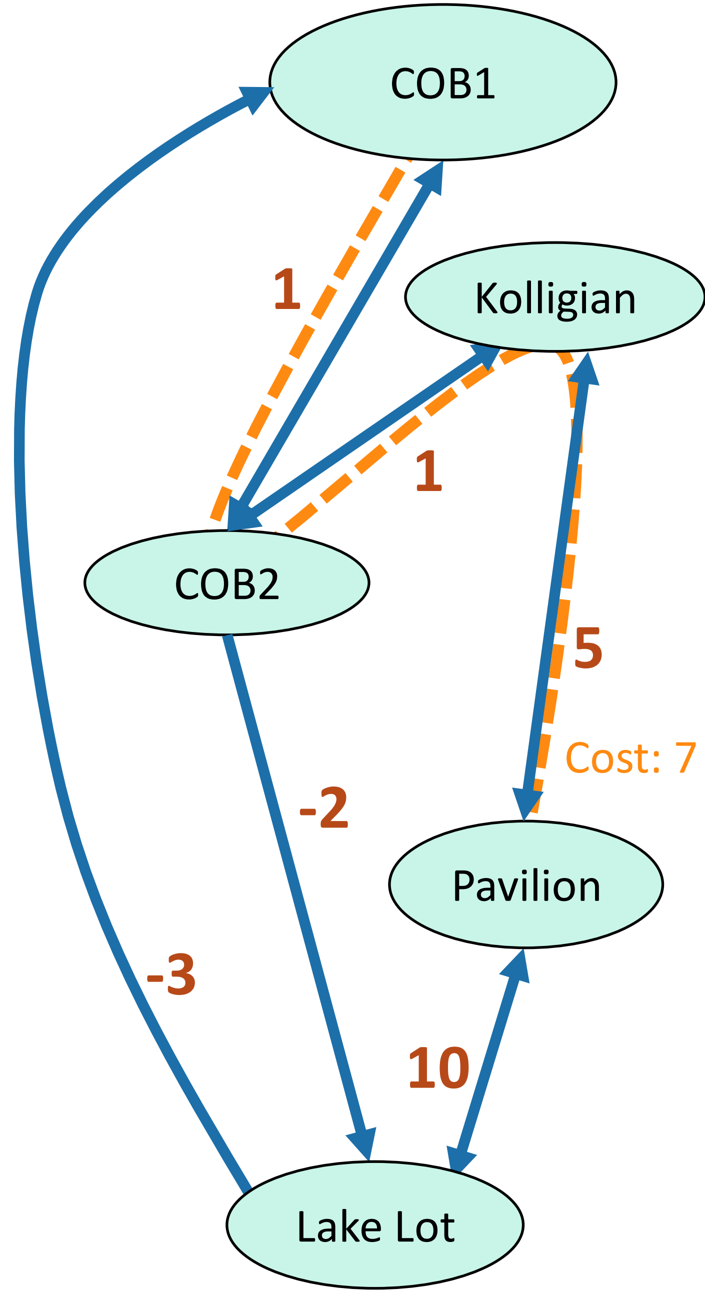
- Don't actually keep all n arrays around.
- Just keep two at a time: "last round" and "this round"

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**3**
$d^{(2)}$	0	**0**	**2**	**13**	3
$d^{(3)}$	0	0	**1**	**7**	3
$d^{(4)}$	0	0	1	**6**	3

Only need these two in order to compute $d^{(4)}$

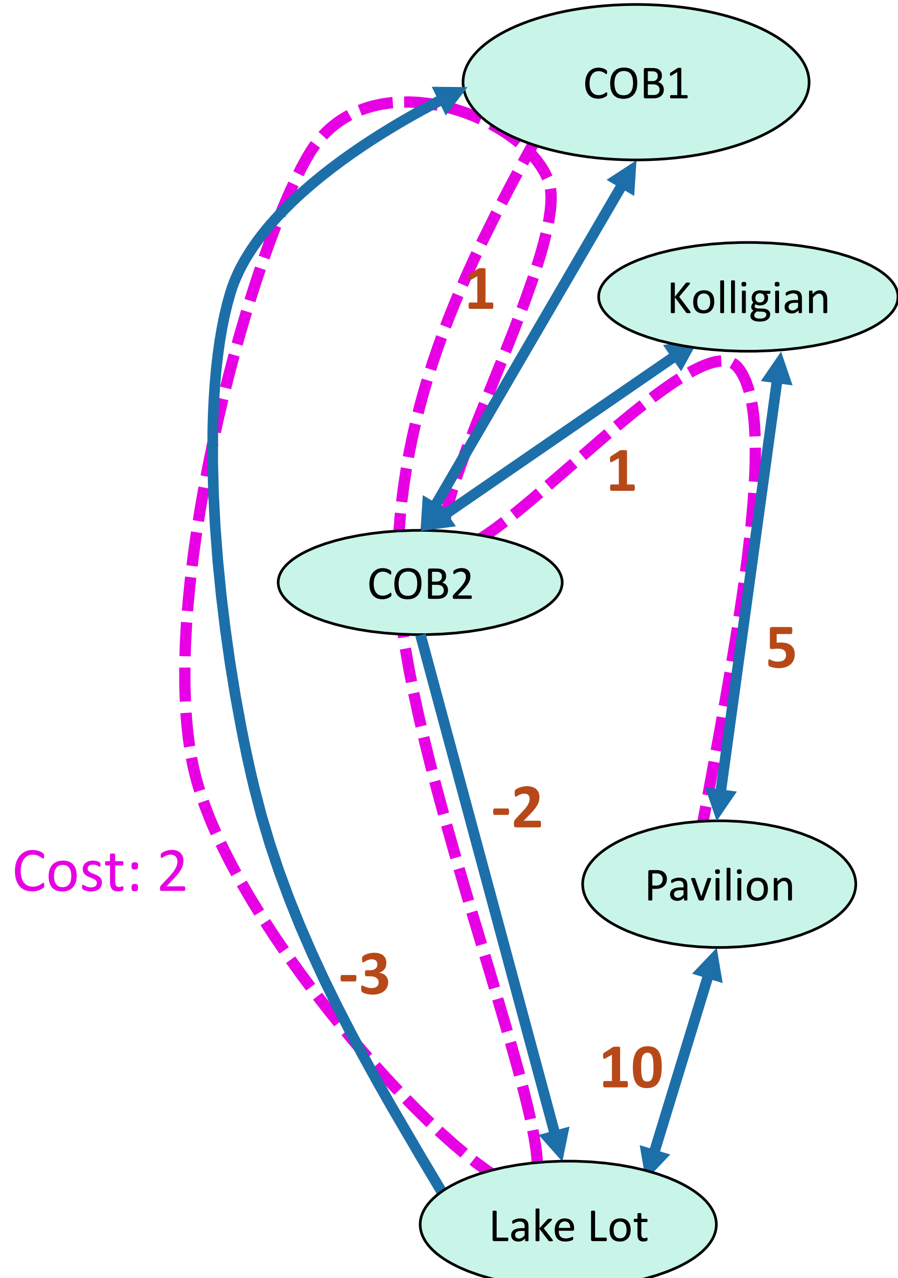
Wait a second…

- What is the shortest path from COB1 to the Pavilion?



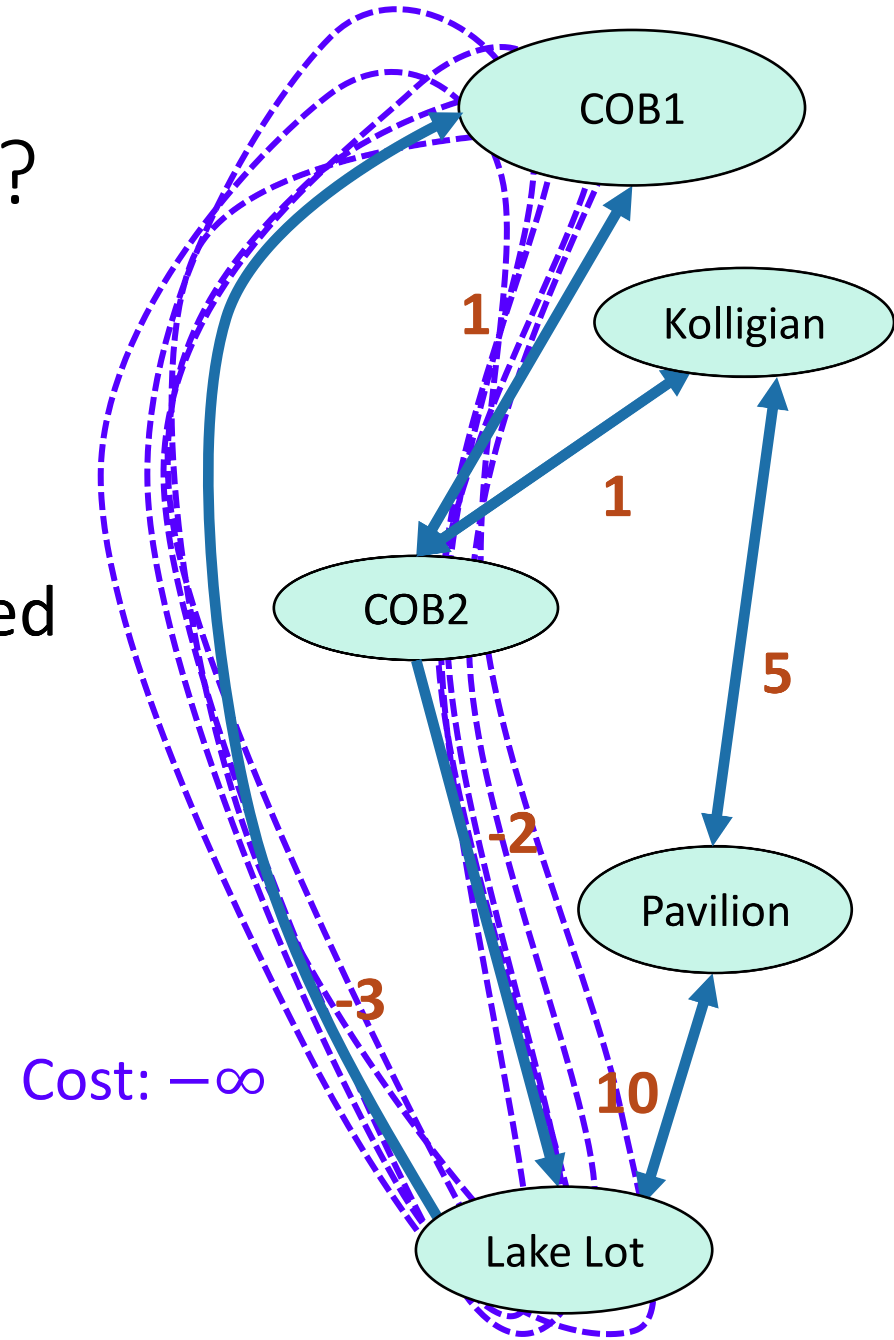
Wait a second…

- What is the shortest path from COB1 to the Pavilion?



Negative edge weights?

- What is the shortest path from COB1 to the Pavilion?
- Shortest paths aren't defined if there are negative cycles!



Bellman-Ford and negative edge weights

- B-F works with negative edge weights…as long as there are not negative cycles.
  - A negative cycle is a path with the same start and end vertex whose cost is negative.
- However, B-F can detect negative cycles.

Back to the correctness

- Does it work?
  - Yes
  - Idea to the right.

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**17**
$d^{(2)}$	0	1	**2**	**27**	**16**
$d^{(3)}$	0	1	2	**7**	16
$d^{(4)}$	0	1	2	7	16

Idea: proof by induction.
Inductive Hypothesis:
$d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.
Conclusion:
$d^{(n-1)}[v]$ is equal to the cost of the shortest simple path between s and v. **(Since all simple paths have at most n-1 edges).**

If there are negative cycles, then non-simple paths matter!
So the proof breaks for negative cycles.