Lec14: Dynamic Programming I

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee
School of Computer Science and Engineering
Kyungpook National University **(KNU)**

Last time

- Breadth-First search (BFS)
- Plus, applications!
  - Dijkstra's Algorithm for solving the single-source shortest path problem in weighted graphs.

Several useful properties

Please read CLRS 24.5

- **Corollary 1**. If there is no path from s to v, then we have:

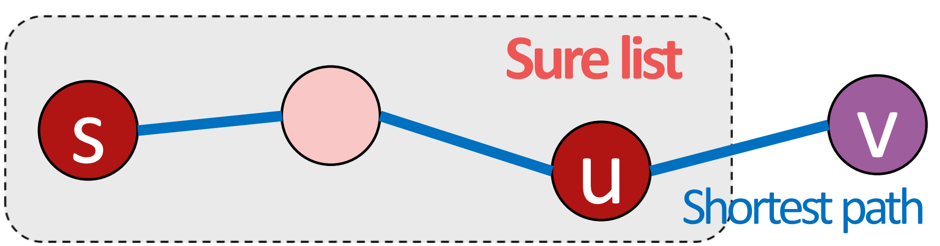
$$d[v] = d(s, v) = \infty$$

- **Lemma 1.** We always have $d[v] \geq d(s, v)$ for all v.

$$d[v] \leftarrow \min(\ \underbrace{d[v]}_{\text{Whatever path we had in mind before}},\ \underbrace{d[u] + \text{edgeWeight}(u,v)}_{\text{The shortest path to u, and then the edge from u to v.}}\ )$$

$d[v]$ = length of the path we have in mind $\geq$ length of shortest path = $d(s,v)$

- **Lemma 2.** If s -> u -> v is a shortest path in G for some u, v in V, and if $d[u] = d(s, u)$ at any time prior to update $w(u, v)$,



Then,
$d[v] = d(s, u) + w(u, v) = d(s, v)$
at all time afterward.

Why does this work?

- **Theorem:**
  - Suppose we run Dijkstra on G = (V,E), starting from s.
  - At the end of the algorithm, the estimate d[v] is equal to the shortest-path weight d(s, v) for all v.

- **Proof.** We use the following loop invariant:

At the start of each iteration of the **while** loop, d[v] = d(s, v) for all v in **the sure list.**

  - **Initialization:** Initially, there is no v in **the sure list**, so the invariant is trivially true.

Proof cont'd – Maintenance

1) Suppose that we are about to add u to **the sure list.**
   - That is, we picked u in the first line here:

Recall:

- 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**
- Repeat

   - Thus, d[u] is the smallest in the not-sure list.

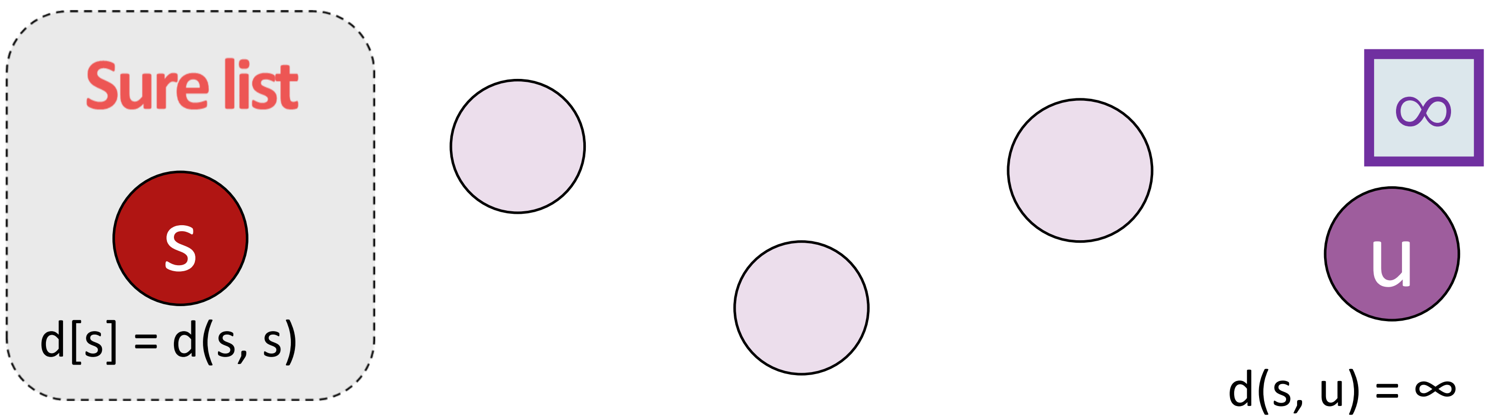
2) Also suppose u is the first vertex that marked sure with d[u] != d(s, u).
   - (This is the way of contradiction.)

Proof cont'd – Maintenance

1) d[u] is the smallest in the not-sure list
2) d[u] != d(s, u)

- s is the first vertex that is marked as sure,
  - At this time, d[s] = d(s, s) = 0. Thus, s ≠ u.

- If there is no path,
  - then, d[u] = d(s, u) = ∞, (by corollary 1) <-- violate the assumption!

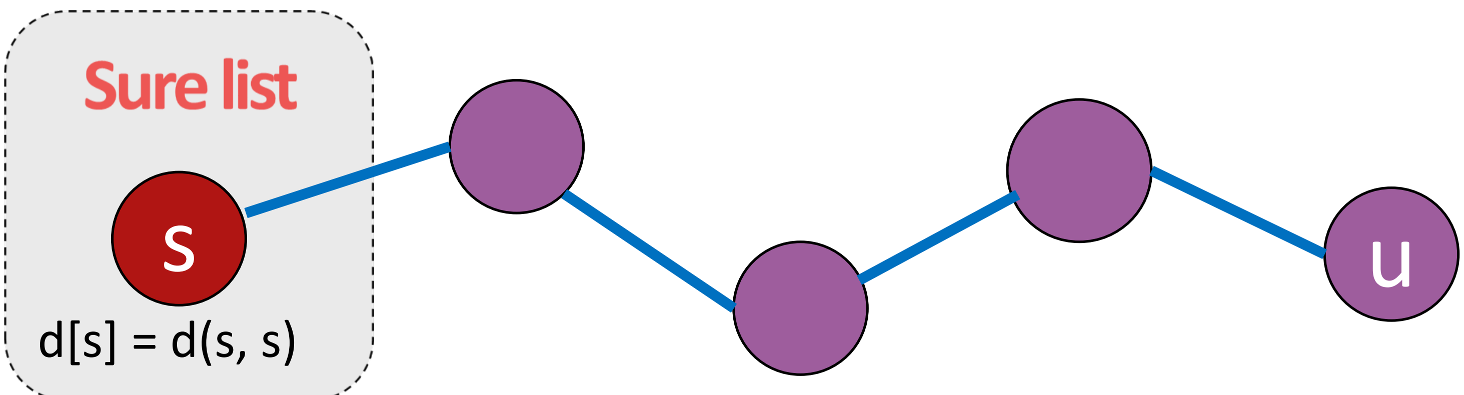


Proof cont'd – Maintenance

1) d[u] is the smallest in the not-sure list
2) **d[u] != d(s, u)**

- s is the first vertex that is marked as sure,
  - At this time, d[s] = d(s, s) = 0. Thus, s ≠ u.

- If there is no path,
  - then, d[u] = d(s, u) = ∞, (by corollary 1) <-- violate the assumption!
  - **Therefore, there must exist one path and the shortest path too.**



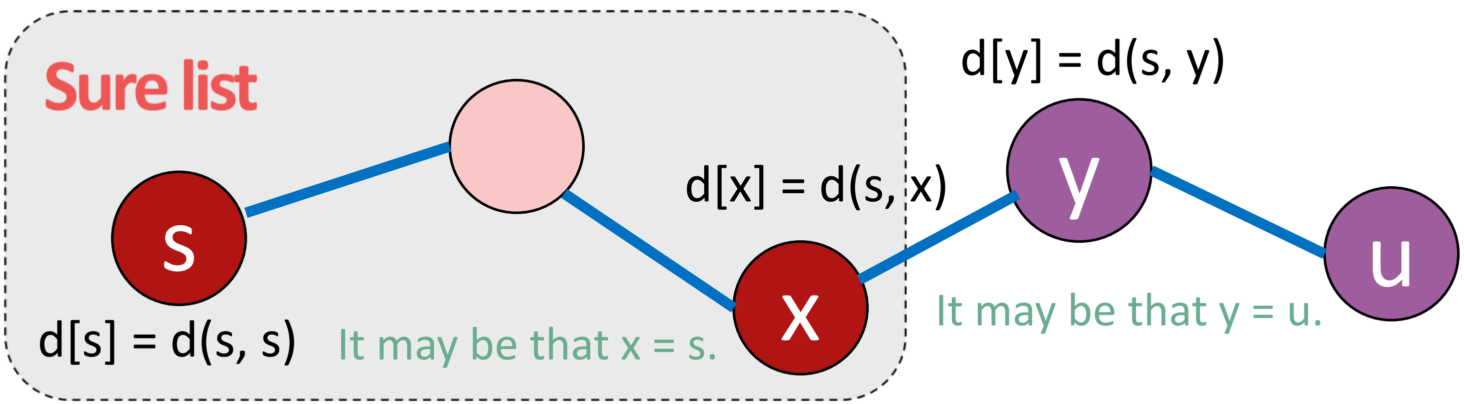
Consider this the shortest path *p*.

Proof cont'd – Maintenance

1) d[u] is the smallest in the not-sure list
2) **d[u] != d(s, u)**

- let x, and y:
  - y be the first vertex along *p* such that y is in **the not-sure list**.
  - x be y's predecessor along *p*.
- Then,
  - d[x] = d(s, x). (By the hypothesis)
  - Also, d[y] = d(s, y). (By the Lemma 2)

u is the first vertex that marked sure with d[u] ≠ d(s, u))

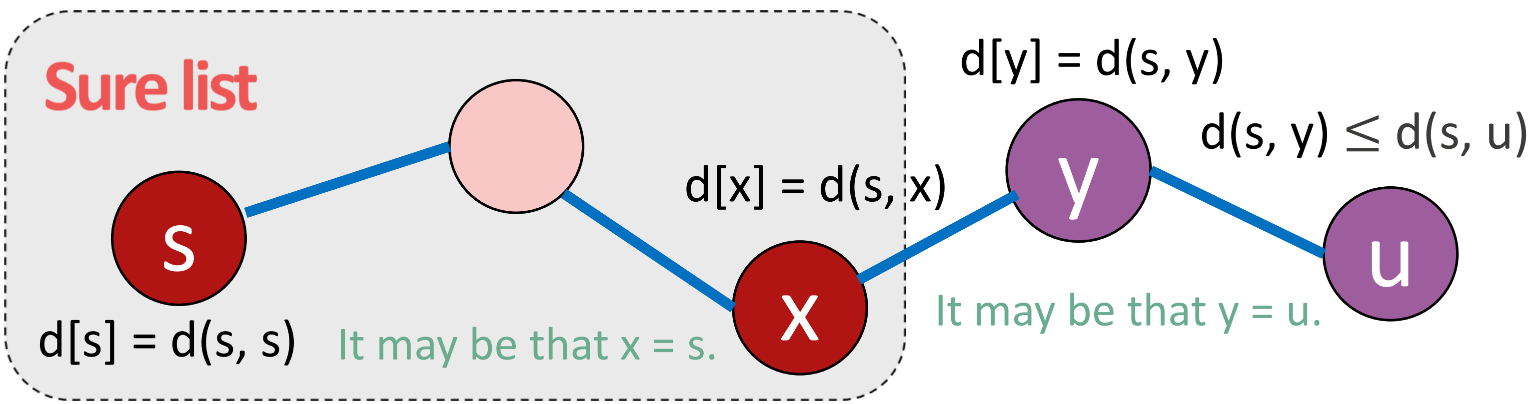


Consider this the shortest path *p*.

Proof cont'd – Maintenance

1) **d[u] is the smallest in the not-sure list**
2) d[u] != d(s, u)

- Then, d[y] = d(s, y) $\leq$ d(s, u) $\leq$ d[u]
  (by Lemma 1)
  - But because both y, and u were in **the not-sure list**, when u was chosen, we have d[u] $\leq$ d[y].
  - Consequently, everything is equal. Thus d[u] = d(s, u). **CONTRADICTION!!**
- We conclude that d[u] = d(s, u) when u is added to **the sure list.**



Consider this the shortest path *p*.

Proof cont'd – Termination

- **Termination:**
  - At termination, **the not-sure list** = Ø which, along with our earlier invariant implies that **the sure list** is equal to V.
  - Thus, d[u] = d(s, u) for all u in V.

Recap: shortest paths

- BFS:
  - (+) O(n+m)
  - (–) only unweighted graphs
- Dijkstra's algorithm:
  - (+) weighted graphs
  - (+) O(nlog(n) + m) if you implement it right.
  - (–) no negative edge weights

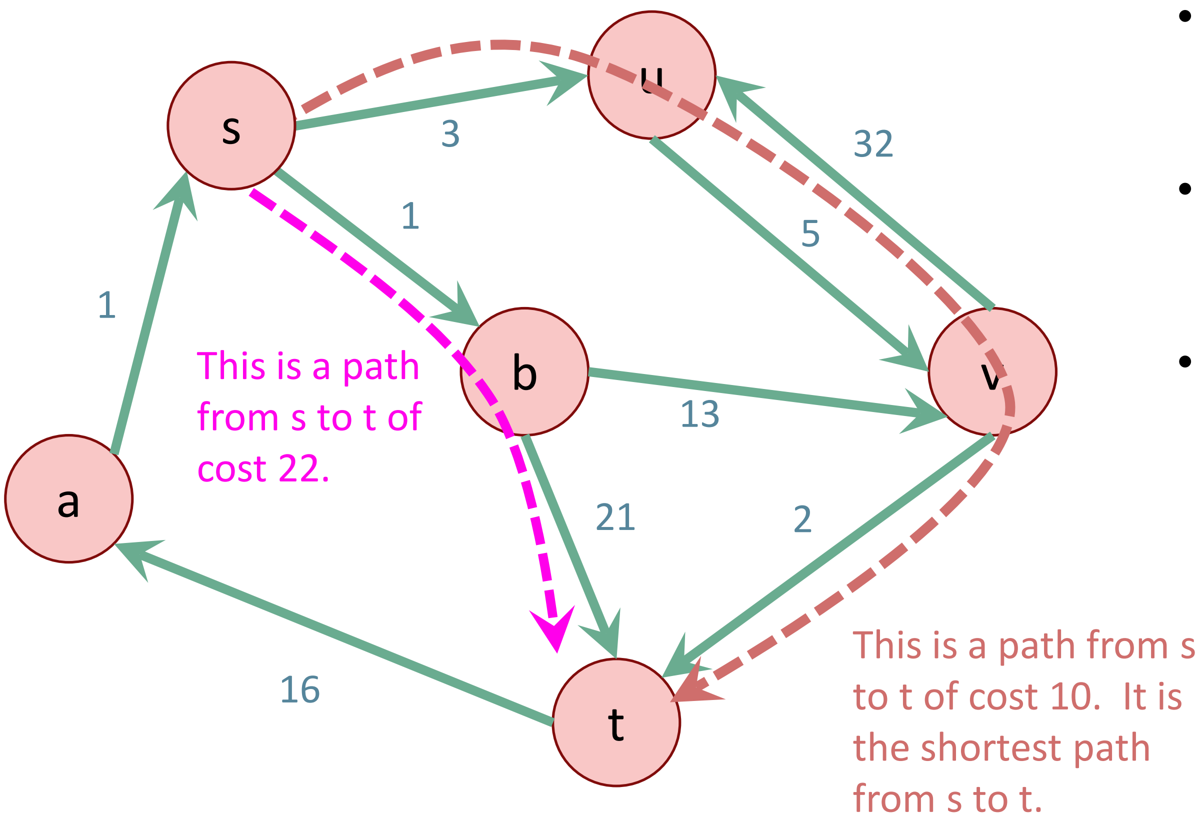
Outline

1. Bellman-Ford Algorithm
2. Dynamic programming
   - Warm-up example: Fibonacci numbers

- *Reading: CLRS 24.1, 15.3*

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

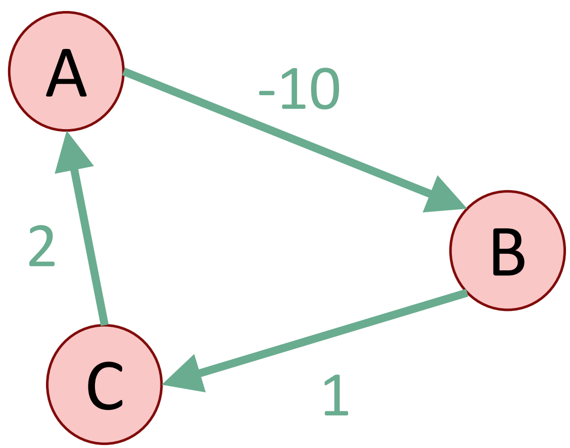
- (–) Slower than Dijkstra's algorithm

- (+) 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.

Aside: Negative Cycles

- A **negative cycle** is a cycle whose edge weights sum to a negative number.
- Shortest paths aren't defined when there are negative cycles!



The shortest path from A to B has cost…negative infinity?

Bellman-Ford vs. Dijkstra

- **Dijkstra:**
  - Find the u with the smallest d[u]
  - Update u's neighbors: d[v] = min(d[v], d[u] + w(u,v))

- **Bellman-Ford:**
  - Don't bother finding the u with the smallest d[u]
  - Everyone updates!

Bellman-Ford Algorithm

- **Bellman-Ford(G,s):**

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

- 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.
  - (Assuming no negative cycles)

Here, Dijkstra picked a special vertex u and updated u's neighbors – Bellman-Ford will update all the vertices.

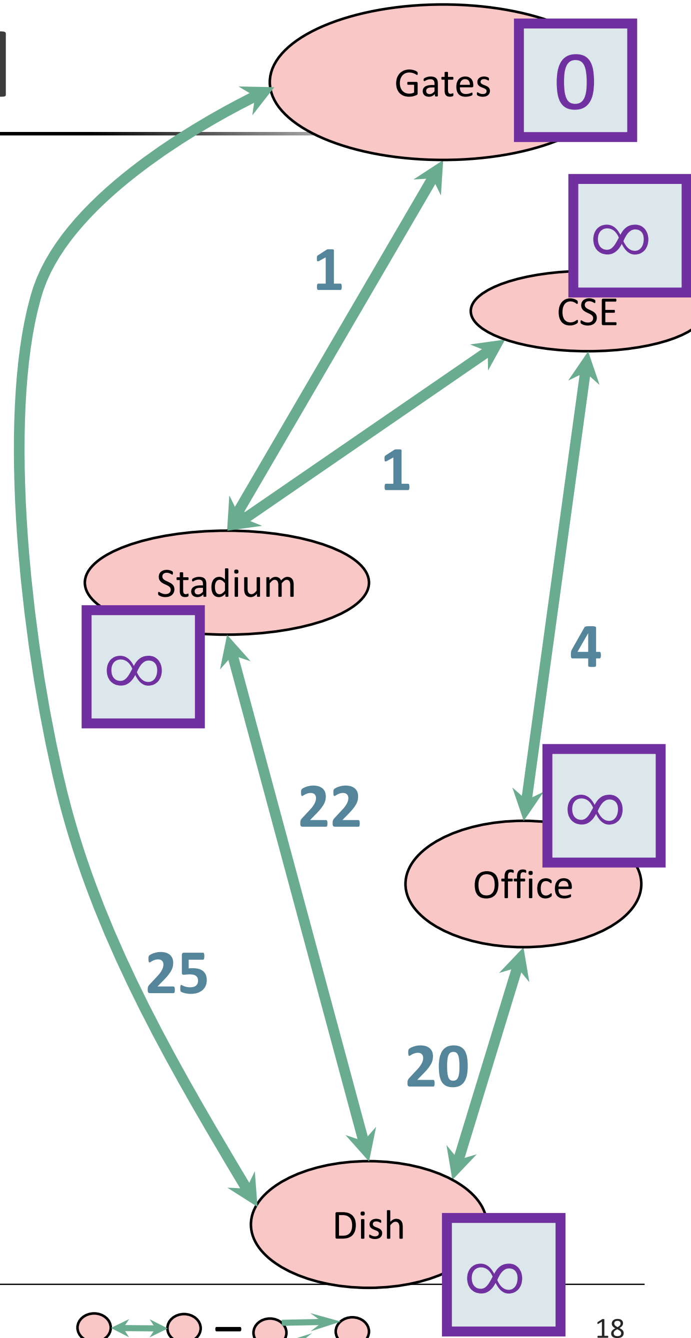
- Running time: O(nm)

Bellman-Ford

- **How far is a node from Gates?**

	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$					
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					

For i=0,…,n-2:
  For v in V:
    $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + w(u,v))$
    where we are also taking the min over all
    u in v.inNeighbors

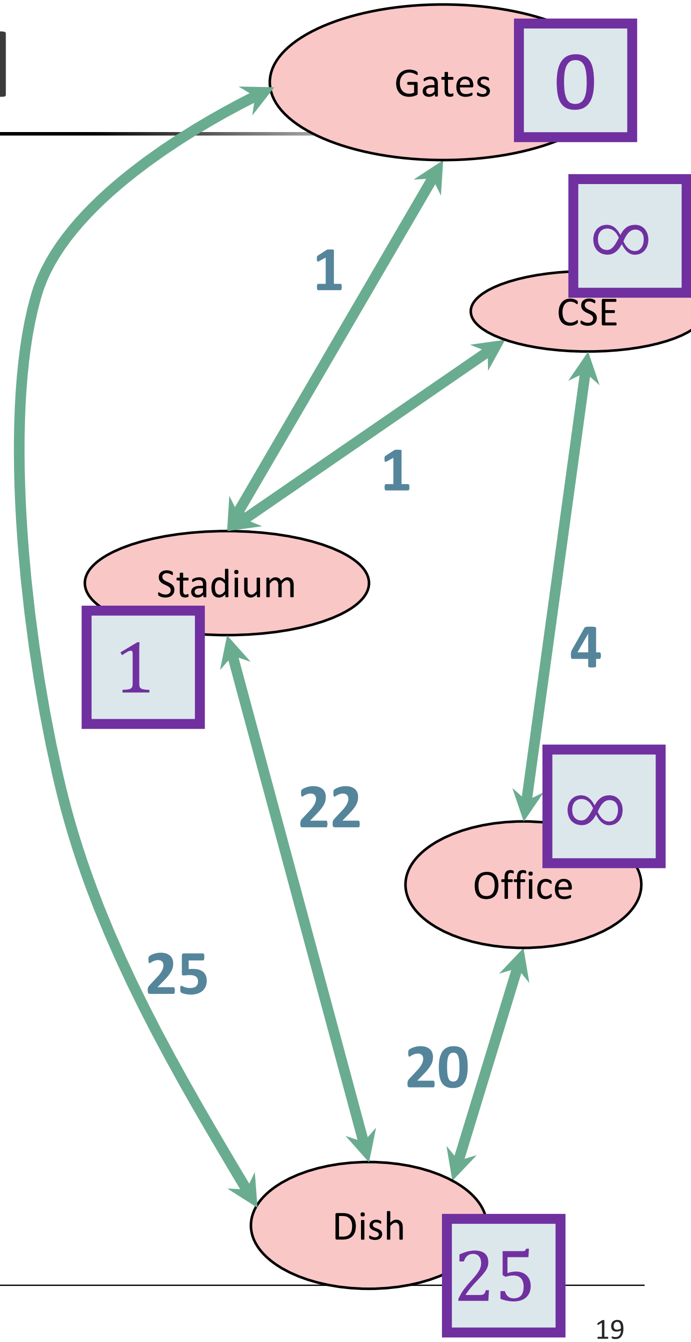


Bellman-Ford

- **How far is a node from Gates?**

	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					

For i=0,…,n-2:
    For v in V:
        $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i)}[u] + w(u,v))$
        where we are also taking the min over all
        u in v.inNeighbors

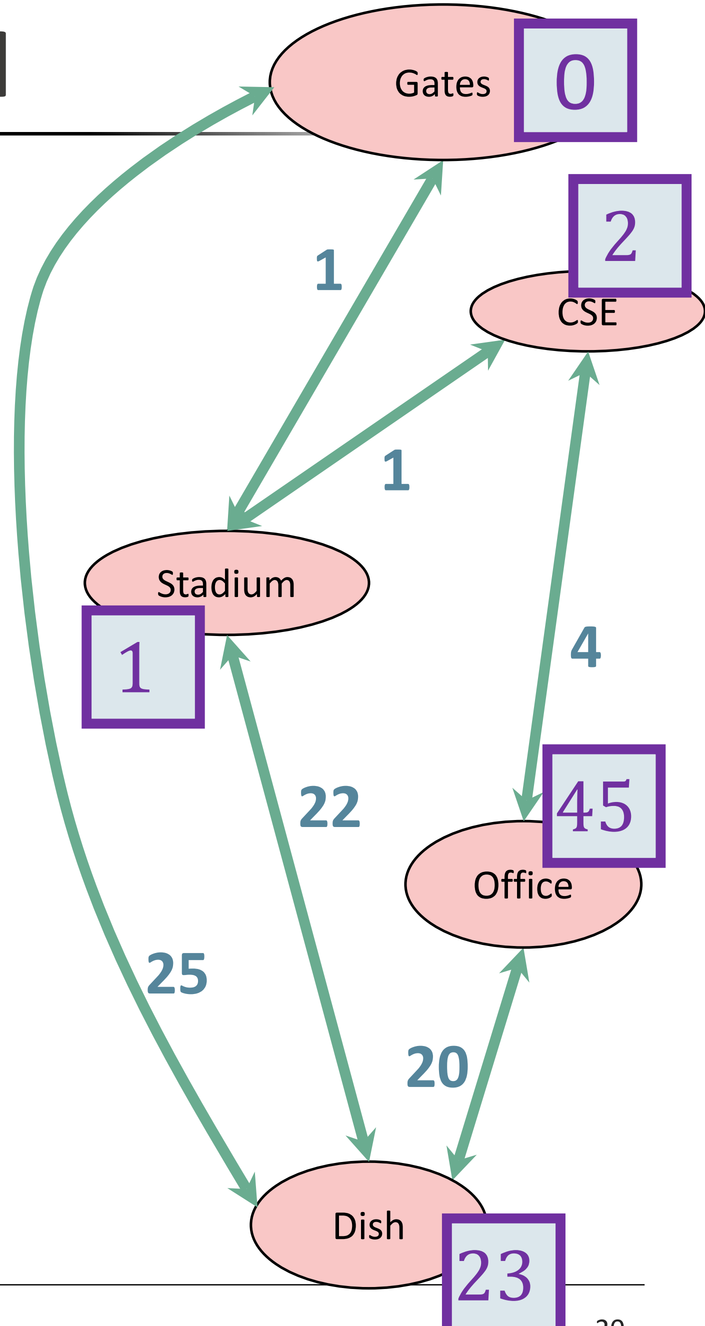


Bellman-Ford

- **How far is a node from Gates?**

	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$					
$d^{(4)}$					

For i=0,…,n-2:
  For v in V:
    $d^{(i+1)}[v] \leftarrow \min(\ d^{(i)}[v]\ ,\ d^{(i)}[u] + w(u,v)\ )$
    where we are also taking the min over all
    u in v.inNeighbors

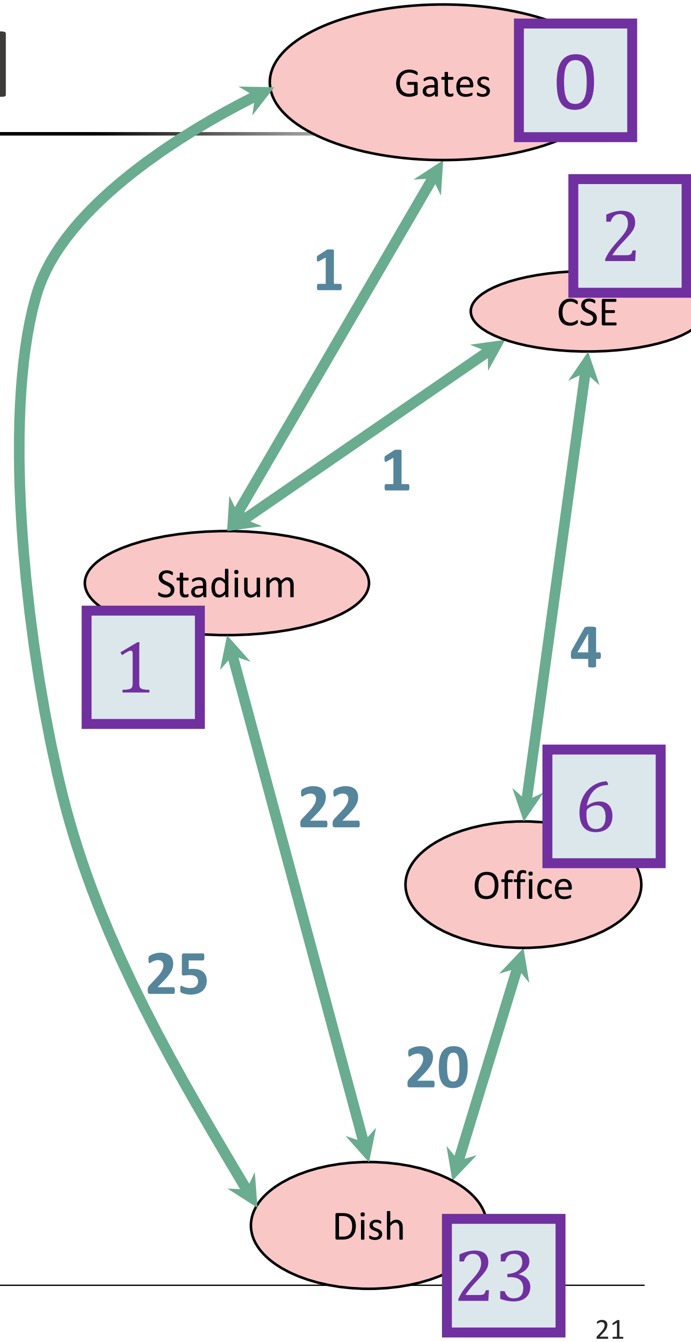


Bellman-Ford

- **How far is a node from Gates?**

	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$					

For i=0,…,n-2:
    For v in V:
        $d^{(i+1)}[v] \leftarrow \min(\ d^{(i)}[v]\ ,\ d^{(i)}[u] + w(u,v)\ )$
        where we are also taking the min over all
        u in v.inNeighbors



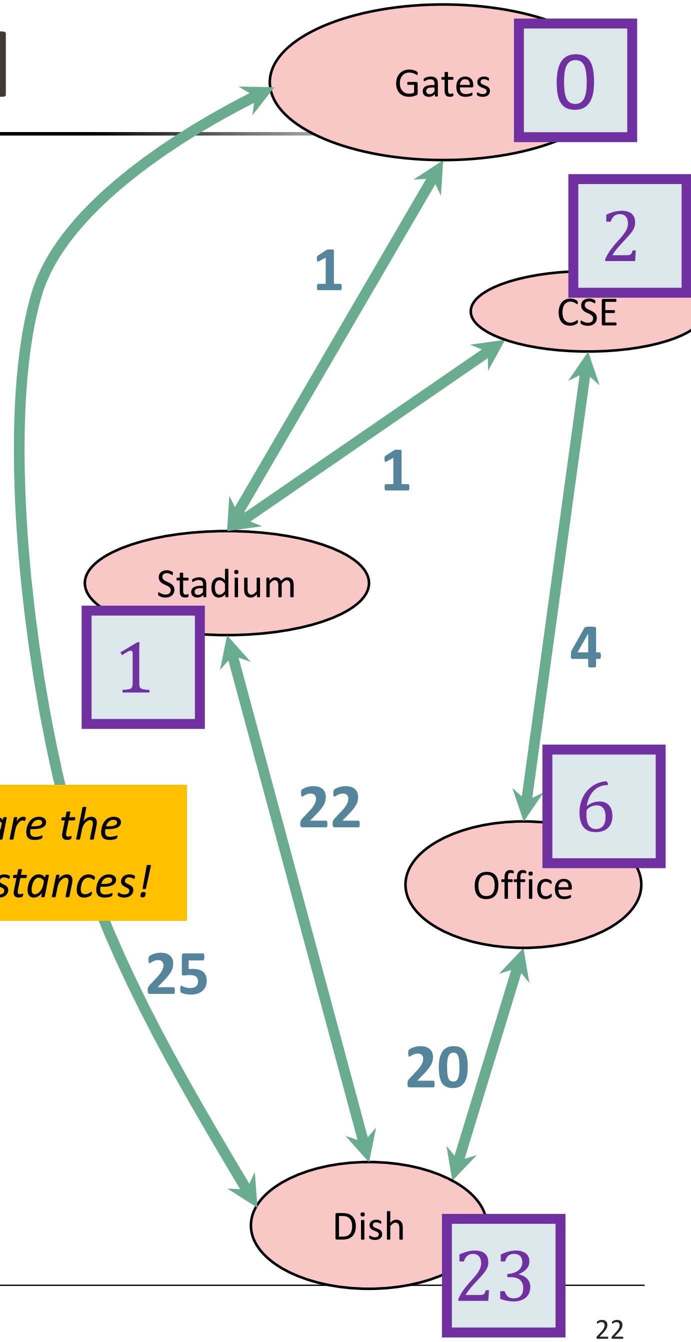
Bellman-Ford

- **How far is a node from Gates?**

	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$	0	1	2	6	23

These are the final distances!

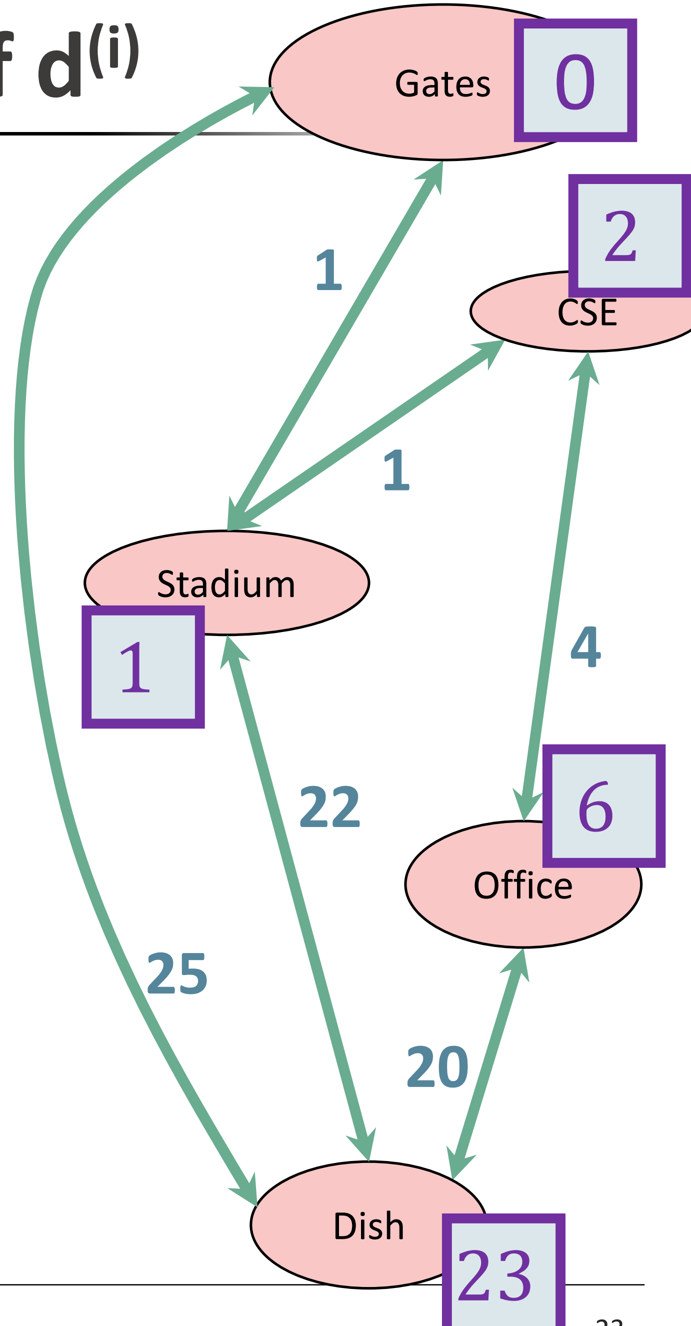
For i=0,…,n-2:
    For v in V:
        $d^{(i+1)}[v] \leftarrow \min(\ d^{(i)}[v]\ ,\ d^{(i)}[u] + w(u,v)\ )$
        where we are also taking the min over all u in v.inNeighbors



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.**

	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	1	∞	∞	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$	0	1	2	6	23
$d^{(4)}$	0	1	2	6	23

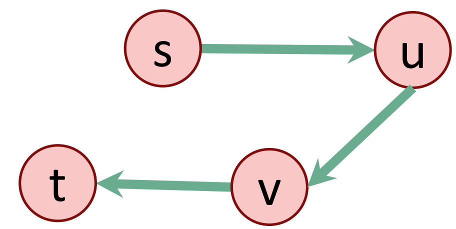


Why does Bellman-Ford work?

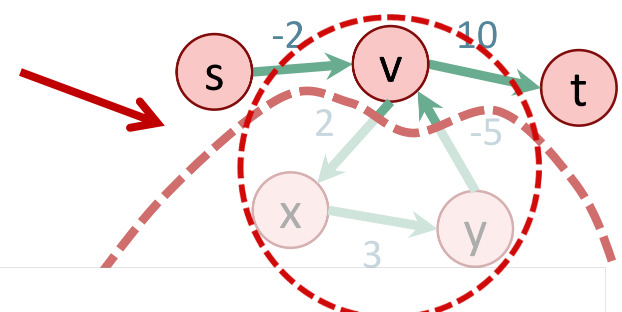
- Idea: proof by induction.
- **Inductive Hypothesis:**
  - $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v with at most i edges.
- **Conclusion:**
  - $d^{(n-1)}[v]$ is equal to the cost of the shortest path between s and v with at most n-1 edges.
  - Aka, the shortest path with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

Aside: the shortest simple path

- **The shortest path with at most n-1 edges is the shortest path in this graph.**
  - Let's say there is a path that uses more than n-1 edges. Since the total number of vertices in the graph is n, there must be at least one cycle in this path.
  - In the definition of the shortest path problem, a negative cycle cannot exist, so this cycle is not negative.
  - Excluding all cycles from that path, the path is shorter (positive cycle) or the same (zero-length cycle). Therefore, among the shortest paths using n-1 or fewer edges, there must exist the actual shortest path.



Can't add another edge without making a cycle!



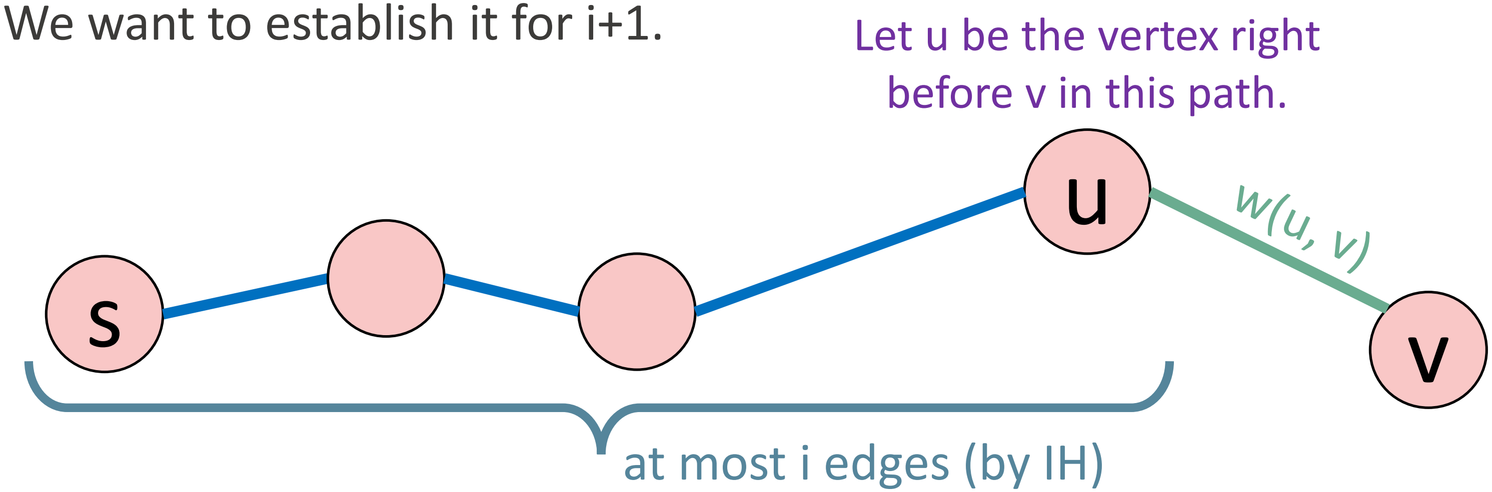
This cycle isn't helping. Just get rid of it.

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 with at most i edges.
- **Base case:**
  - After iteration 0…
- **Inductive step:**

Inductive step

- **Inductive 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.



  - Then, d[v] is decided by min{d[v] , d[u] + w(u, v)}, where d[u] is the shortest path with at most i edges.
  - Therefore, d[v] becomes the shortest path with at most i+1 edges in iteration i+1.

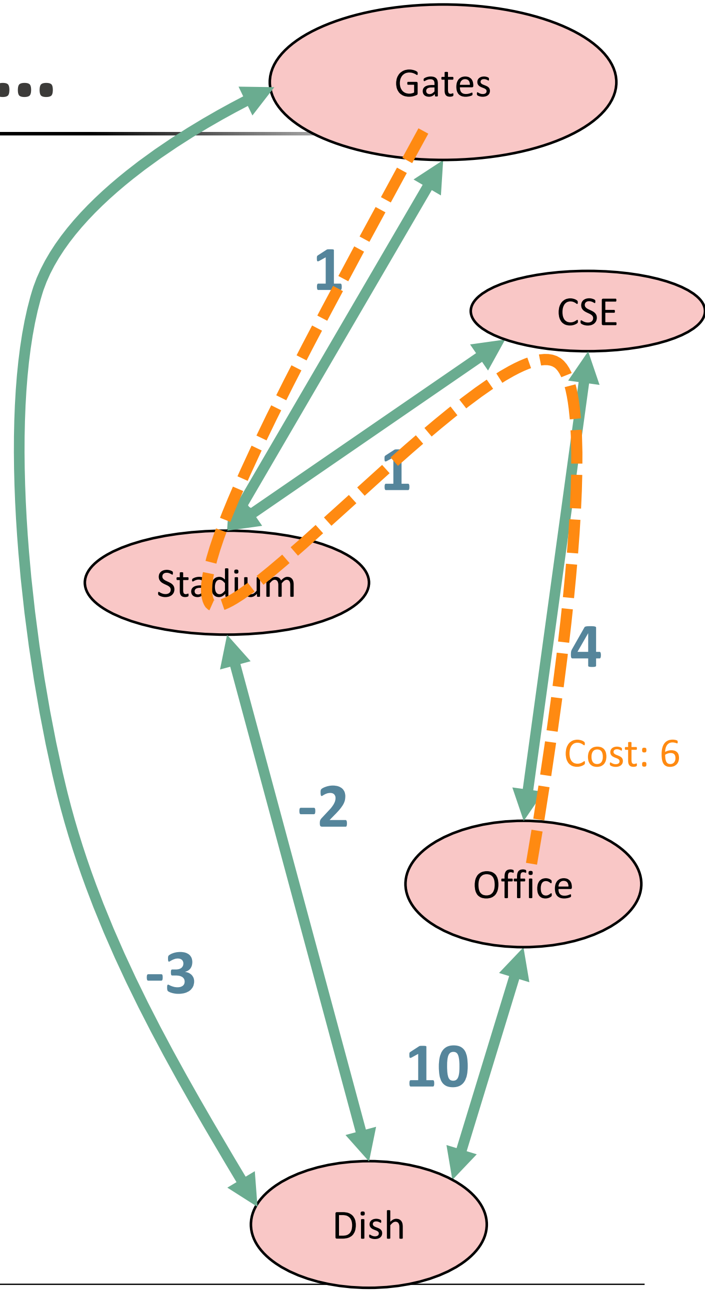
Conclusion

- **Inductive 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.
- **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.
  - And 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.
  ✓

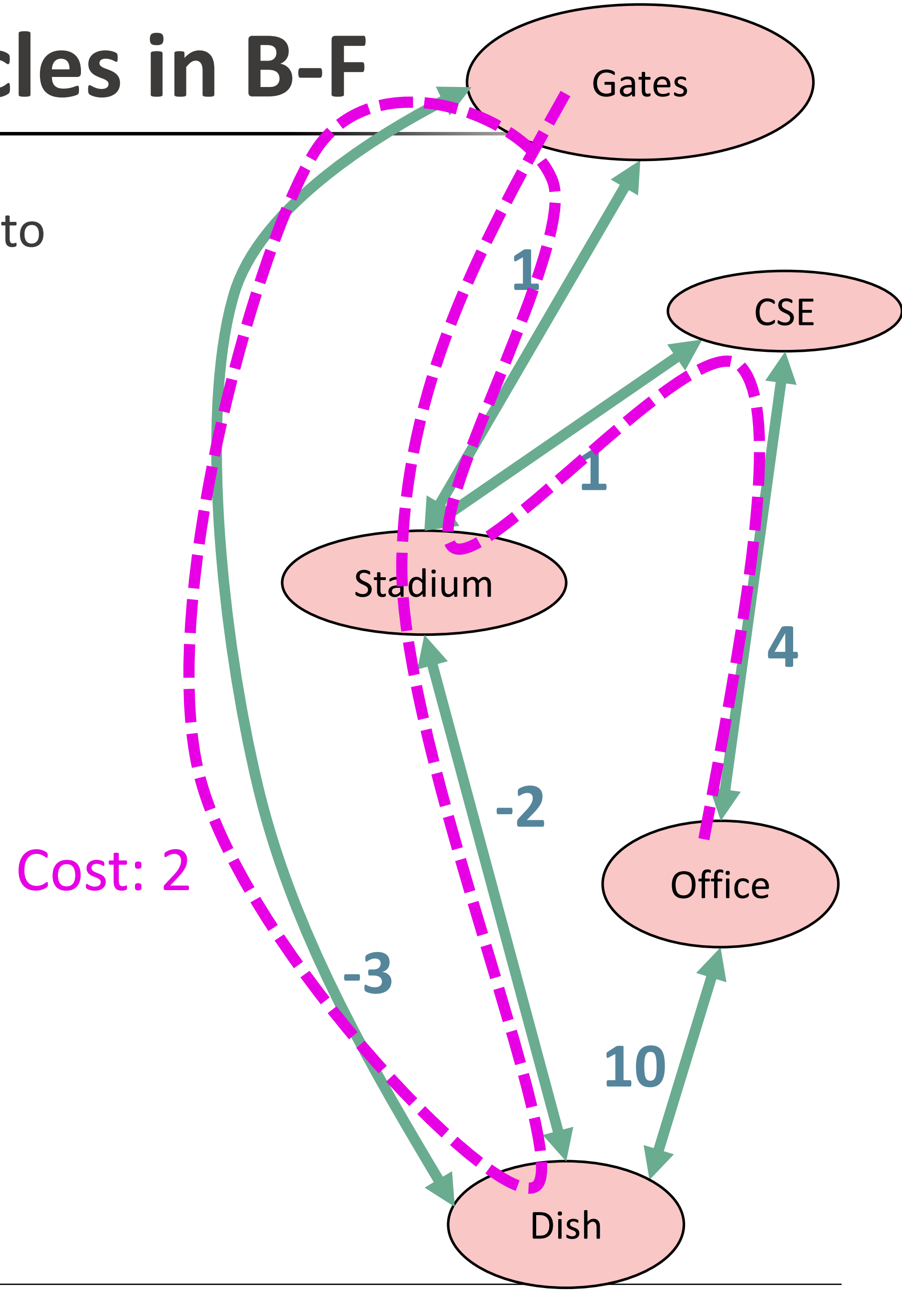
Wait a second…

- What is the shortest path from Gates to the Office?



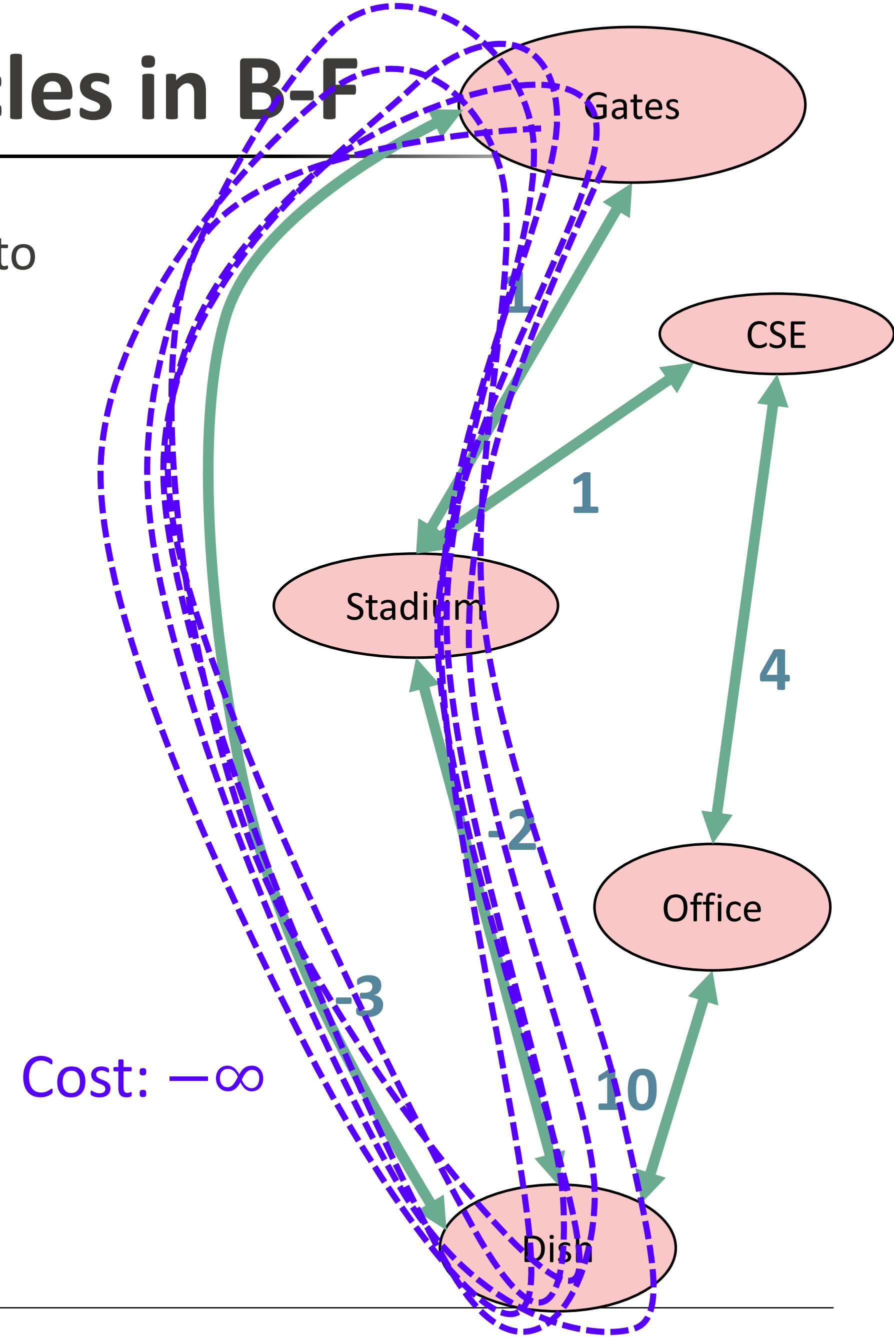
Negative cycles in B-F

- What is the shortest path from Gates to the Office?



Negative cycles in B-F

- What is the shortest path from Gates to the Office?



Negative cycles in Bellman-Ford

- B-F works with negative edge weights… as long as there are no negative cycles.
- However, B-F can detect negative cycles.

Back to the correctness

- 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 path between s and v with at most n-1 edges.
  - Aka, the shortest path with at most n-1 edges is the shortest simple path and equals to the shortest path.

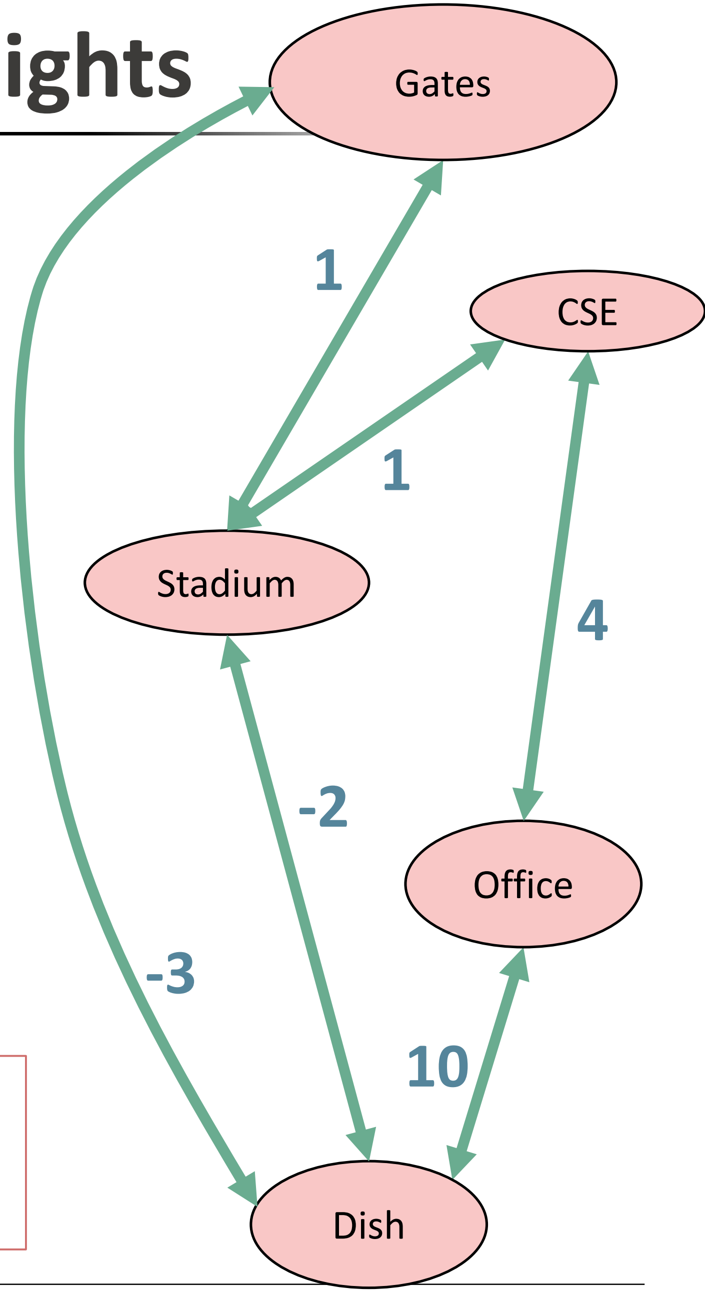
If there are negative cycles,
then non-simple paths matter!

Negative edge weights

	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**-3**
$d^{(2)}$	0	**-3**	**2**	**7**	**-3**
$d^{(3)}$	**-4**	-5	**-4**	**6**	**-3**

This is not looking good!

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



Negative edge weights

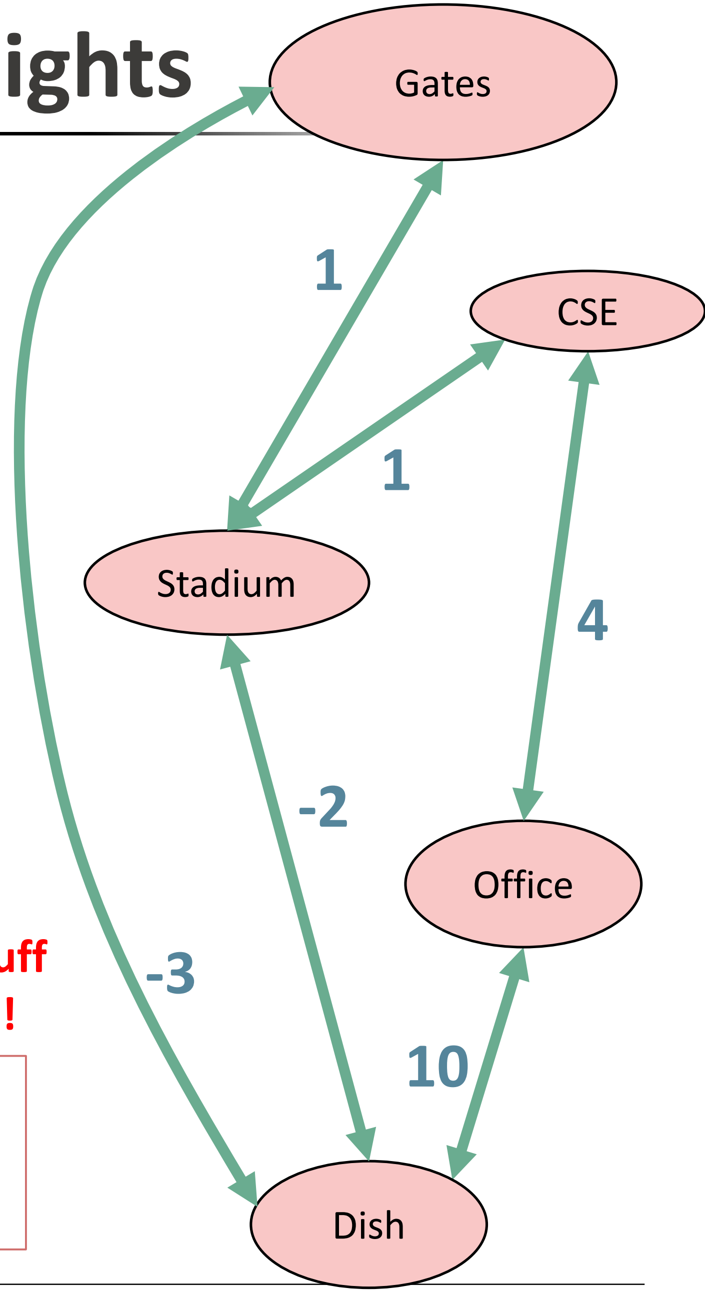
	Gates	Stadium	CSE	Office	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**-3**
$d^{(2)}$	0	**-3**	**2**	**7**	**-3**
$d^{(3)}$	**-4**	-5	**-4**	**6**	**-3**
$d^{(4)}$	-4	-5	-4	6	**-7**

We can tell that it's not looking good:

	Gates	Stadium	CSE	Office	Dish
$d^{(5)}$	-4	**-9**	-4	**3**	-7

Some stuff changed!

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



How B-F deals with negative cycles

- **If there are no negative cycles:**
  - Everything works as it should.
  - The algorithm stabilizes after n-1 rounds.
- **If there are negative cycles:**
  - Not everything works as it should…
    - It couldn’t possibly work, since shortest paths aren’t well-defined if there are negative cycles.
  - The d[v] values will keep changing.
- **Solution:**
  - Go one round more and see if things change.
  - If so, return NEGATIVE CYCLE ☹

Bellman-Ford Algorithm

Bellman-Ford(G,s): *G = (V,E) is a graph with n vertices and m edges, s is a start vertex*

- 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-1:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\})$
- If $d^{(n-1)} \mathrel{!}= d^{(n)}$:
  - Return NEGATIVE CYCLE ☹
- Otherwise, $\text{dist}(s,v) = d^{(n-1)}[v]$

Bellman-Ford take-aways

- Running time is O(mn).
  - For each of n rounds, update m edges.
- Works fine with negative edges.
- Does not work with negative cycles.
  - No algorithm can – shortest paths aren't defined if there are negative cycles.
- B-F can detect negative cycles.
- Bellman-Ford is also used in practice.
  - e.g., Routing Information Protocol (RIP) uses something like Bellman-Ford. (Older protocol, not used as much anymore.)

And B-F is also an example of…

Dynamic Programming!

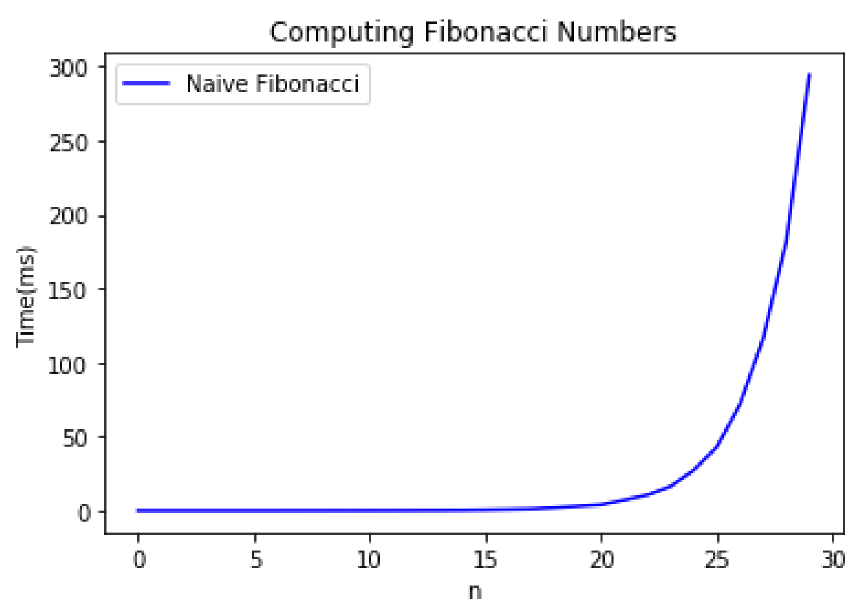
Let's start with a simple example

- How to compute Fibonacci Numbers
- Definition:
  - $F(n) = F(n-1) + F(n-2)$, with $F(1) = F(2) = 1$.
  - The first several are:
    - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,…
- Question:
  - Given n, what is F(n)?

Candidate algorithm

```
•def Fibonacci(n):
  •if n == 0, return 0
  •if n == 1, return 1
  •return Fibonacci(n-1) + Fibonacci(n-2)
```

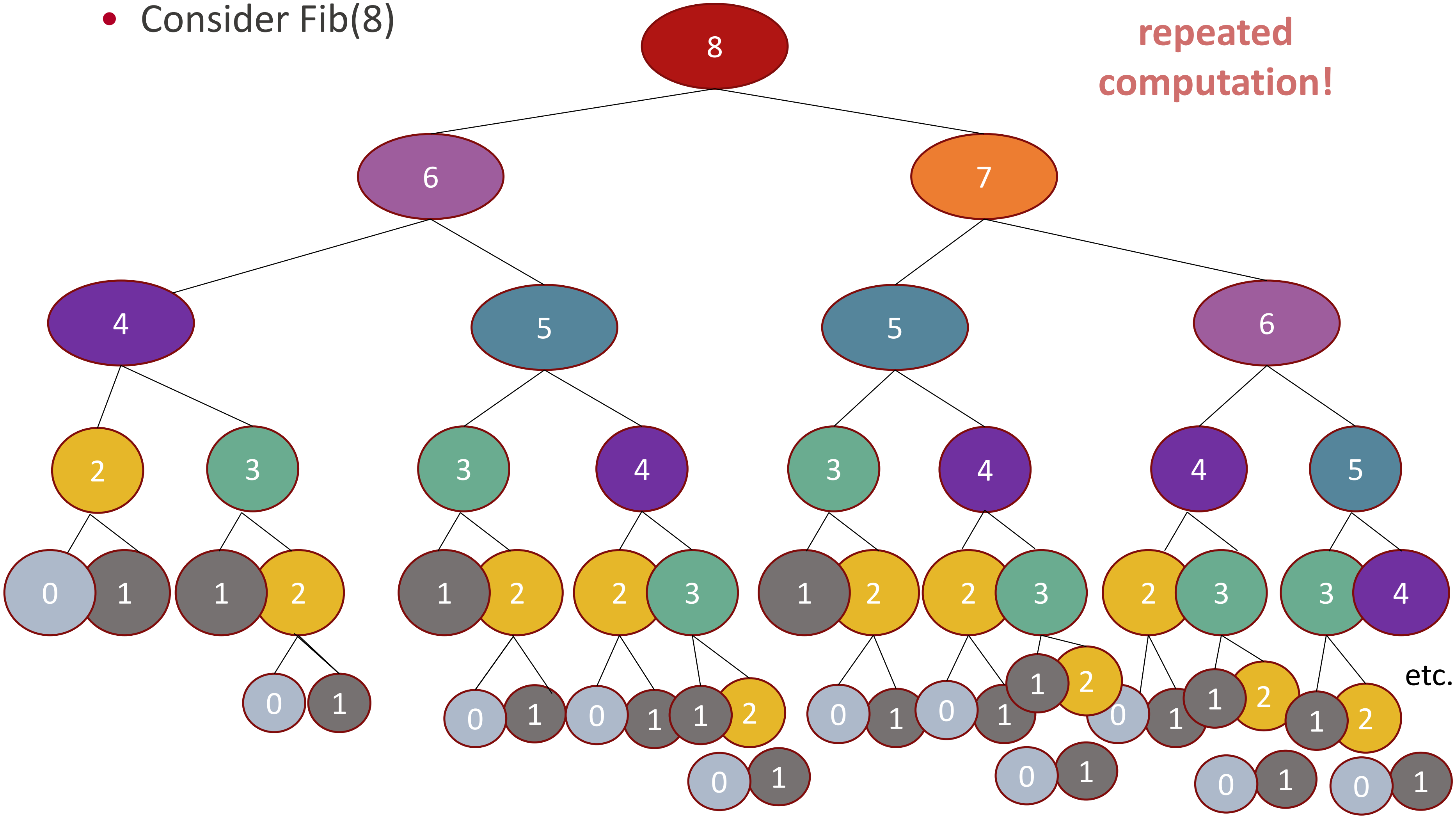
- Running time?
- T(n) ≥ T(n-1) + T(n-2) for n ≥2
- This is **EXPONENTIALLY QUICKLY!**
  - $T(n) \geq 2T(n-2)$ implies $T(n) \geq \Omega(2^{n/2})$.



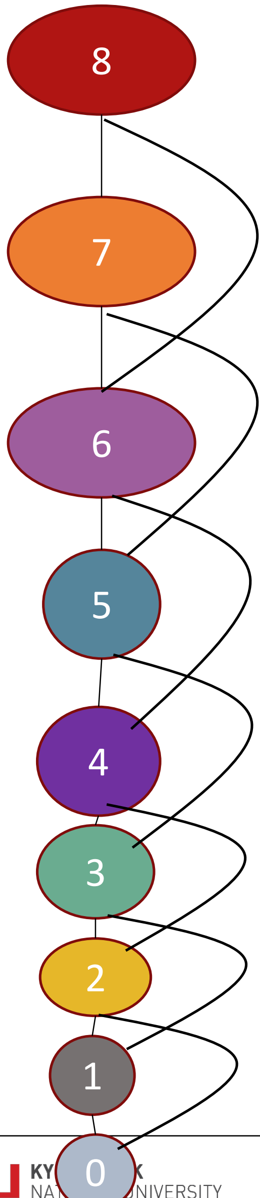
What's going on?

- Consider Fib(8)

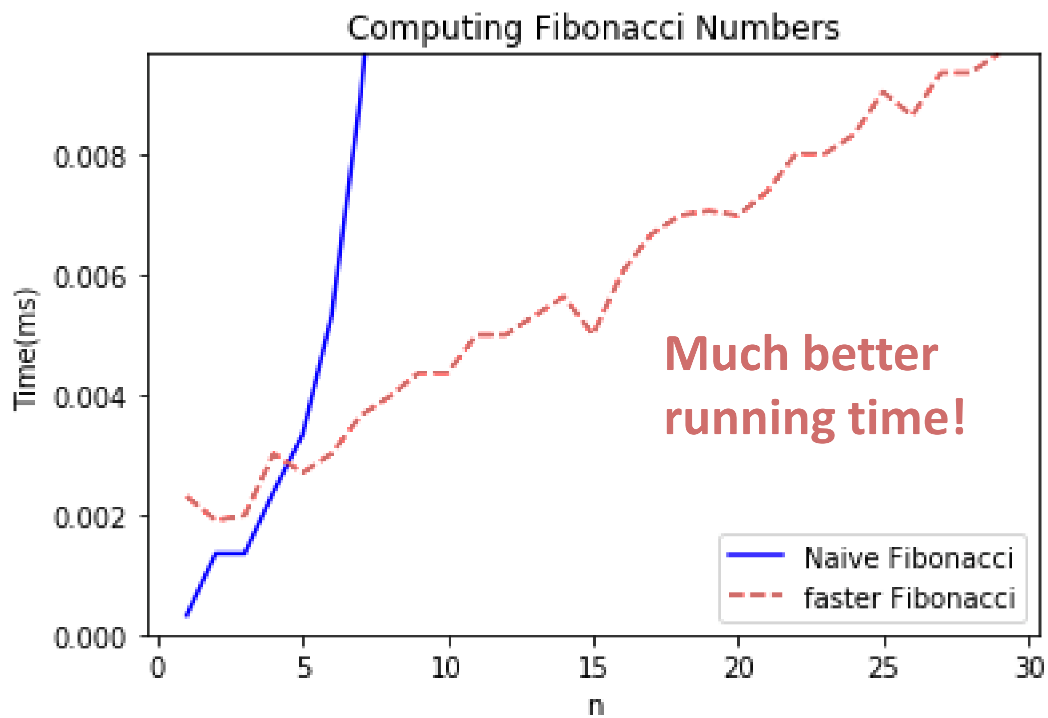
That's a lot of repeated computation!



Maybe this would be better:




```
def fasterFibonacci(n):
    • F = [0, 1, None, None, …, None ]
      // F has length n + 1
    • for i = 2, …, n:
        • F[i] = F[i-1] + F[i-2]
    • return F[n]
```



What is dynamic programming?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually, it is for solving **optimization problems**
  - If the answer to a large problem includes the answer to a smaller problem, it is said to have an optimal structure.
  - E.g., ***shortest*** path

Elements of dynamic programming

1. Optimal sub-structure:

- Big problems break up into sub-problems.
  - Fibonacci: F(i) for $i \leq n$
  - Bellman-Ford: Shortest paths with at most i edges for $i \leq n$
- The solution to a problem can be expressed in terms of solutions to smaller sub-problems.
  - Fibonacci: $F(i+1) = F(i) + F(i-1)$
  - Bellman-Ford: $d^{(i+1)}[v] \leftarrow \min\{ d^{(i)}[v], d^{(i)}[u] + weight(u,v) \}$

Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap.
  - Fibonacci:
    - Both F[i+1] and F[i+2] directly use F[i].
    - And lots of different F[i+x] indirectly use F[i].
  - Bellman-Ford:
    - Many different entries of $d^{(i+1)}$ will directly use $d^{(i)}[v]$.
    - And lots of different entries of $d^{(i+x)}$ will indirectly use $d^{(i)}[v]$.

- → *This means that we can save time by solving a sub-problem just once and storing the answer.*

Elements of dynamic programming

1. **Optimal substructure.**
   - Optimal solutions to sub-problems can be used to find the optimal solution of the original problem.
2. **Overlapping subproblems.**
   - The subproblems show up again and again.

- Using these properties, we can design a ***dynamic programming*** algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - At the end we can use information we collected along the way to find the solution to the whole thing.

DP algorithms

- Two ways to think about and/or implement
  - Top down
  - Bottom up

Bottom up approach

- What we just saw.

- **For Fibonacci:**
  - Solve the small problems first
    - fill in F[0], F[1]
  - Then bigger problems
    - fill in F[2]

      ...
  - Then bigger problems
    - fill in F[n-1]
  - Then finally solve the real problem.
    - fill in F[n]

Bottom up approach

- **For Bellman-Ford:**
  - Solve the small problems first
    - fill in $d^{(0)}$
  - Then bigger problems
    - fill in $d^{(1)}$

      …
  - Then bigger problems
    - fill in $d^{(n-2)}$
  - Then finally solve the real problem.
    - fill in $d^{(n-1)}$

Top down approach

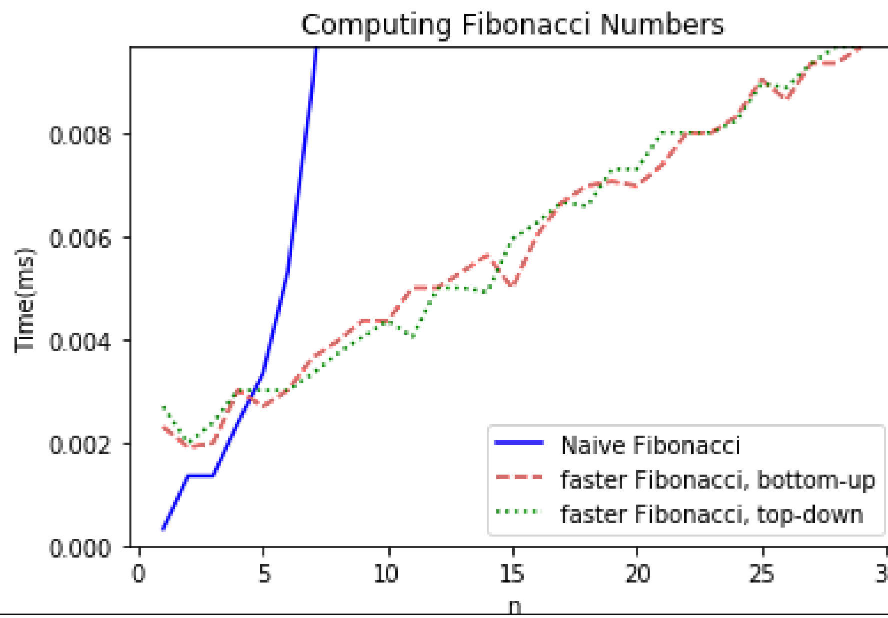
- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
    - etc..

- The difference from divide and conquer:
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
  - Aka, "memo-ization"

Example of top-down Fibonacci

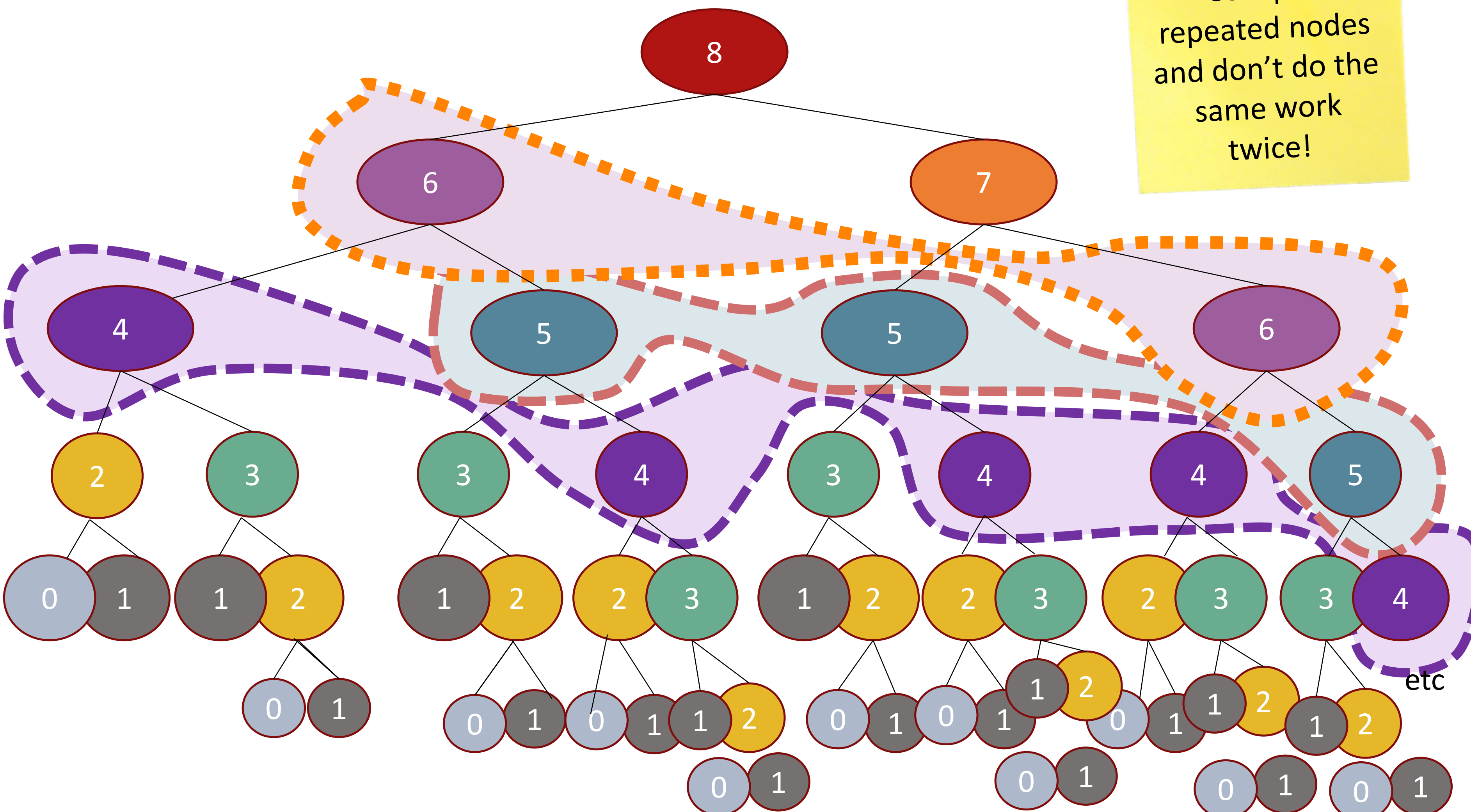
- `define a global list F = [0,1,None, None, …, None]`
- **`def`** `Fibonacci(n):`
  - **`if`** `F[n] != None:`
    - **`return`** `F[n]`
  - **`else`:**
    - `F[n] = Fibonacci(n-1) + Fibonacci(n-2)`
  - **`return`** `F[n]`

Memo-ization:
Keeps track (in F) of the stuff you've already done.



Memo-ization Visualization

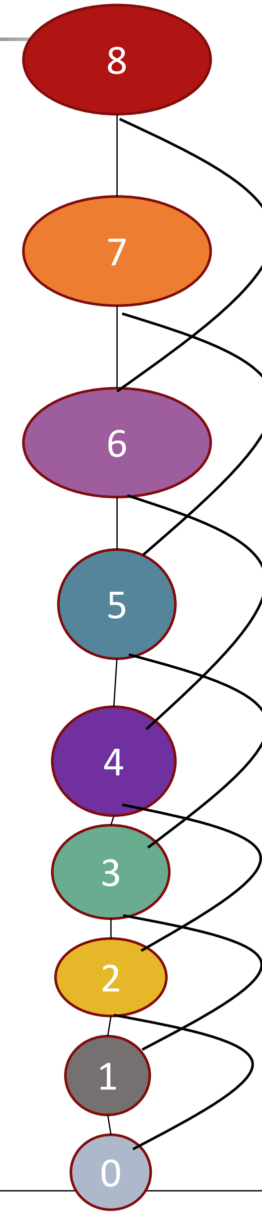
Collapse repeated nodes and don't do the same work twice!



Memo-ization Visualization

Collapse repeated nodes and don't do the same work twice!

But otherwise treat it like the same old recursive algorithm.




```
• define a global list F = [0,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```

What have we learned?

- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Uses optimal substructure
  - Uses overlapping subproblems
  - Can be implemented bottom-up or top-down.


Any Question?