The Bellman-Ford Algorithm

CS/MCS 401: Computer Algorithms I
Lecture 20
Professor: Jonathan Liu

Icebreaker: Do you have a plan to vote today?

Announcements

Big Announcement: Interactive Practice

- Rather than worked examples, I have developed a series of interactive practice problems. They are on the website. These should be easier than the homework, and are intended to give you some practice. The problems and feedback are all written by me. The interface is vibe-coded.

Other things:

- Project inputs released!
- HW 4 Solutions released. Grading will be done tonight.
- HW 5 will be released tomorrow.
- (Civic) Participation: upload a sticker to gradescope (preferably an I Voted one) and get a participation point!

Today's Overview

Today we're going to revisit the classic problem of shortest paths, and see how dynamic programming can be leveraged to tackle a harder version.

This lecture has two purposes:

1. Getting more practice designing dynamic programming problems (e.g. Dynamic Programming unit)
2. Seeing a specific implementation of an important algorithm (e.g. Graph Algorithms unit)

Recap: Dijkstra’s Algorithm

Dijkstra's Algorithm

Dijkstra's(G=(V,E), starting vertex s):

S ← {s} \\ set of visited vertex

d[s] ← 0 \\ distances from s to every vertex

while S ≠ V:

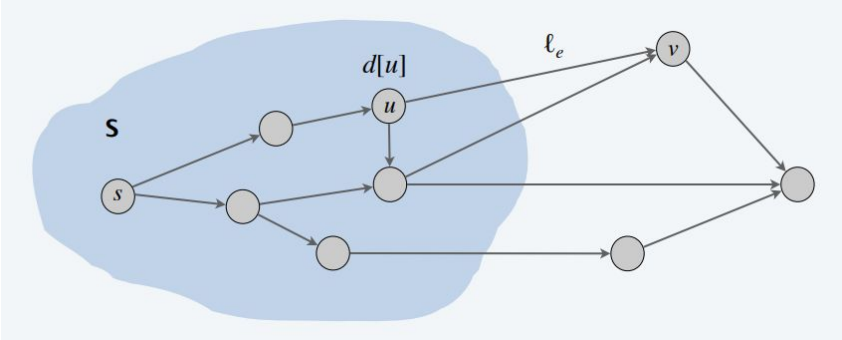
get the unvisited vertex v minimizing $d[u] + l_{(u, v)}$

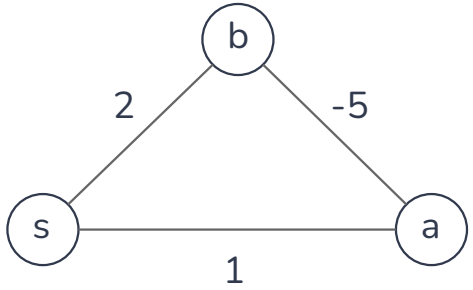
add v to S

$d[v] \leftarrow d[u] + l_{(u, v)}$

Given a weighted graph and starting vertex s, Dijkstra's algorithm returned the distance from s to every vertex in the graph.

Only assuming that edges all had positive weights!





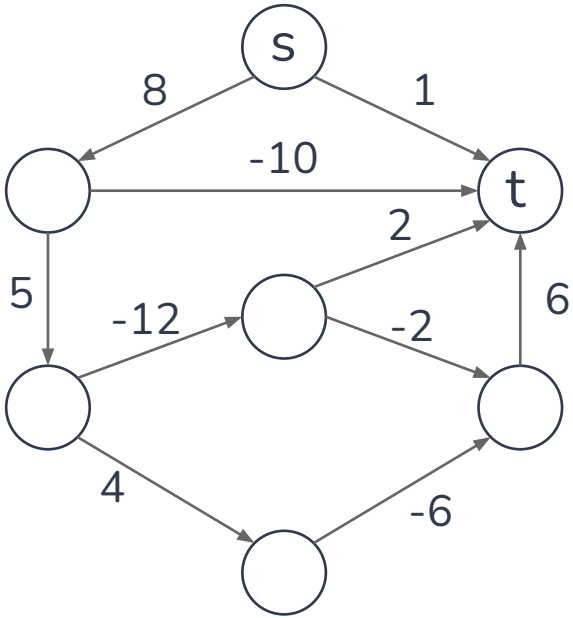
Today: What if the edges can be negative?

What if we tried adding a weight to every edge until they're all non-negative?



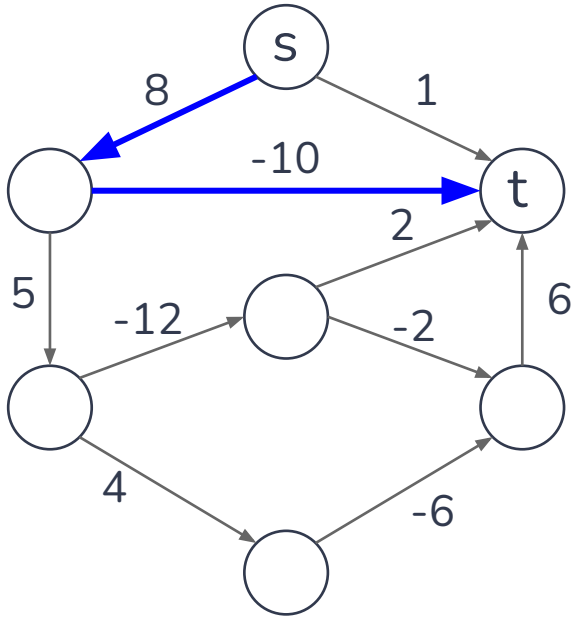
This doesn't preserve shortest paths!

Let’s try a simple version first.



This is a DAG. What is the shortest path from s to t?

Let’s try a simple version first.



This is a DAG. What is the shortest path from s to t?

What made this problem so annoying?

Let’s try dynamic programming!

Shortest Paths on DAGs

Problem: On a weighted DAG, find shortest paths from source vertex s.

Subproblem: ?

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Shortest Paths on DAGs

Problem: On a weighted DAG, find shortest paths from source vertex s.

Subproblem: opt[v] = distance from s to v.

Recurrence: ?

<u>Design Steps for Dynamic Programming</u>

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Shortest Paths on DAGs

Problem: On a weighted DAG, find shortest paths from source vertex s.

Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: ?

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Shortest Paths on DAGs

Problem: On a weighted DAG, find shortest paths from source vertex s.

Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: ?

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Shortest Paths on DAGs

Problem: On a weighted DAG, find shortest paths from source vertex s.

Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: ?

<u>Design Steps for Dynamic Programming</u>

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Shortest Paths on DAGs

Problem: On a weighted DAG, find shortest paths from source vertex s.

Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

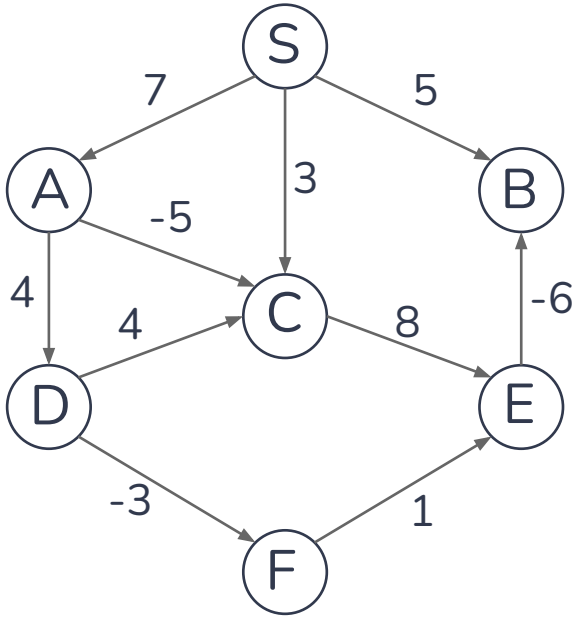
Return Value: return all of opt.

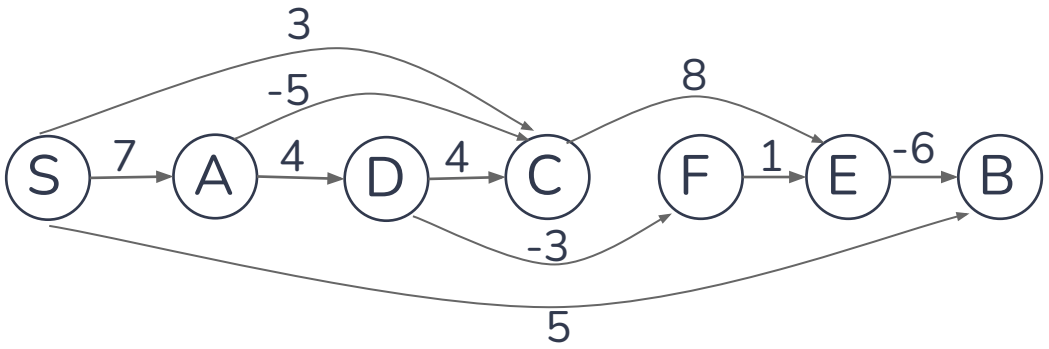
Let's try it!

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Shortest Paths on DAGs: Example





Shortest Paths on DAGs: Example

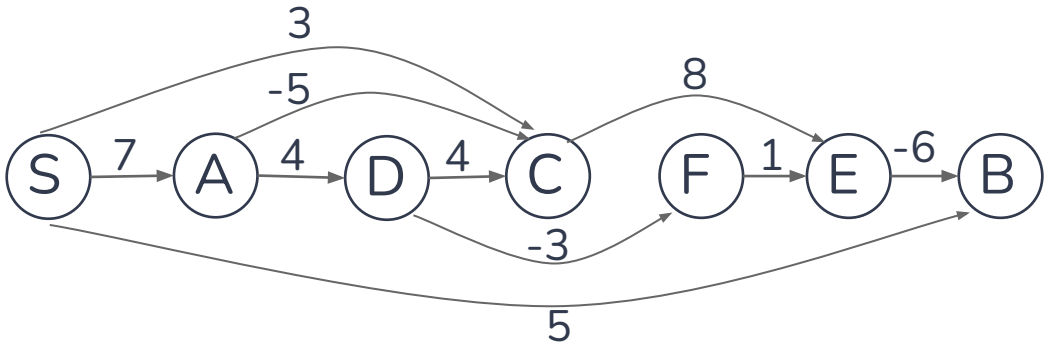
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]							

Shortest Paths on DAGs: Example

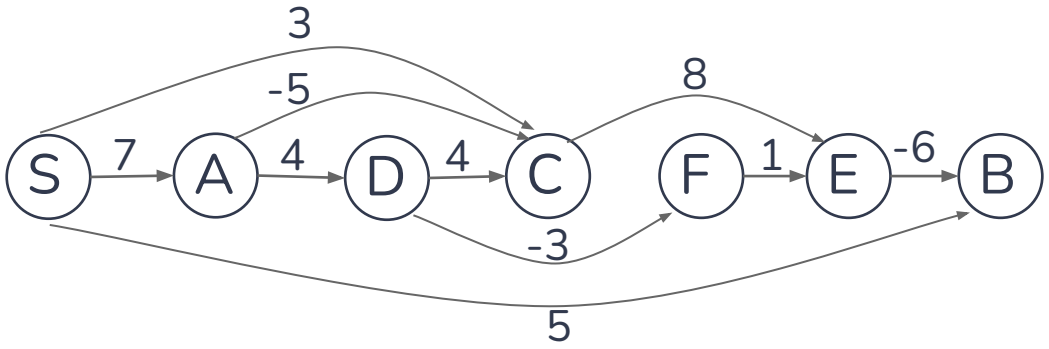
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]	0						

Shortest Paths on DAGs: Example

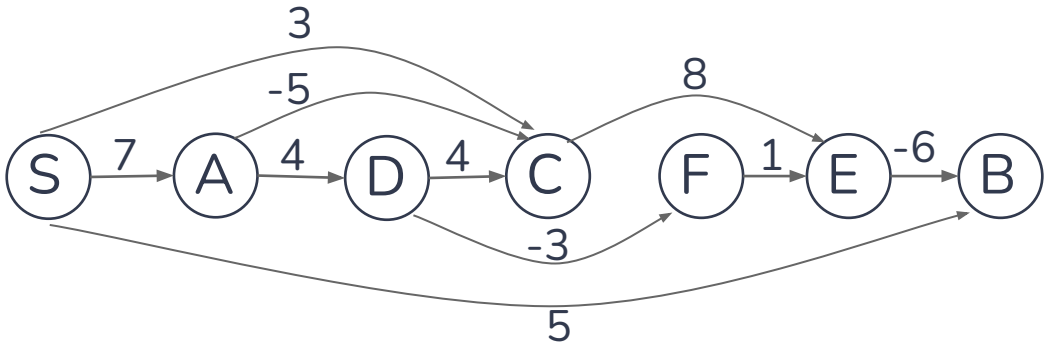
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]	0	7					

Shortest Paths on DAGs: Example

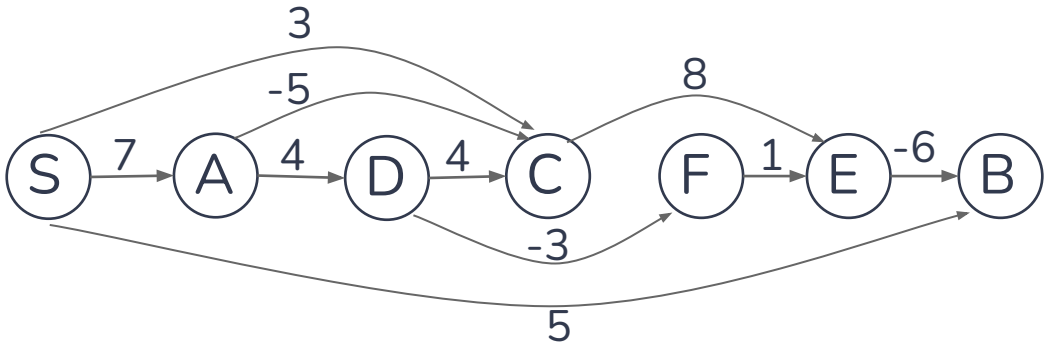
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]	0	7	11				

Compare OPT[S]+3, OPT[A]-5, and OPT[D]+4.

Shortest Paths on DAGs: Example

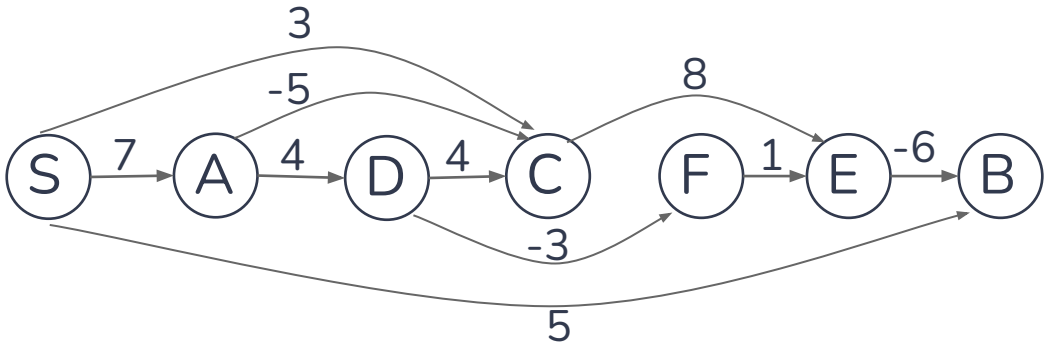
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]	0	7	11	2			

Shortest Paths on DAGs: Example

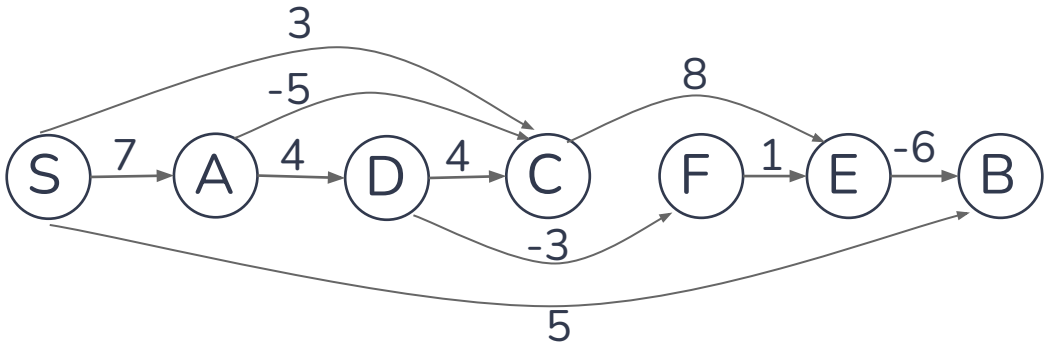
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]	0	7	11	2	8		

Shortest Paths on DAGs: Example

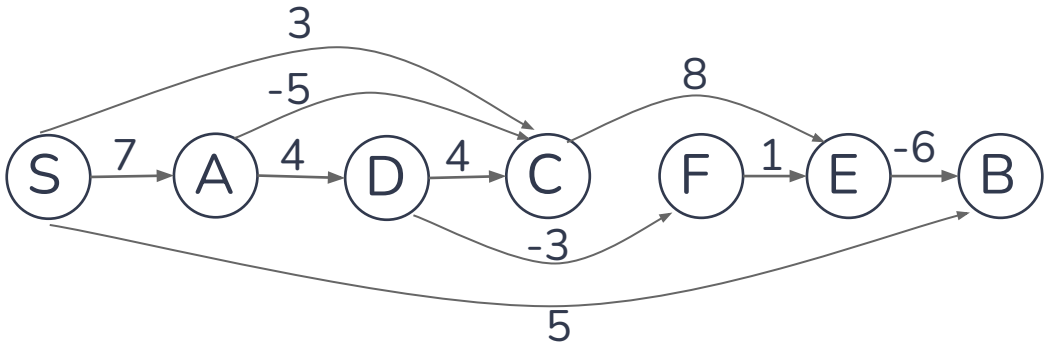
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]	0	7	11	2	8	9	

Shortest Paths on DAGs: Example

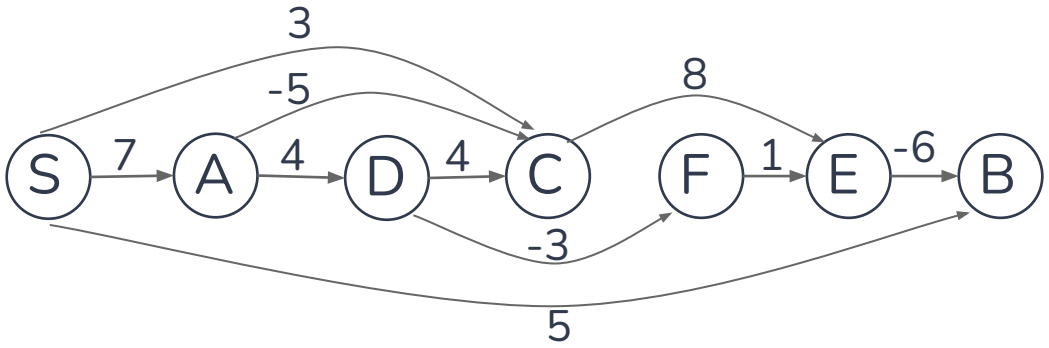
Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: We can start by taking a topological sort of the DAG!

Base Case: opt[s] = 0

Return Value: return all of opt.



v	S	A	D	C	F	E	B
OPT[v]	0	7	11	2	8	9	3

Graphs with Cycles and Negative Weights

Shortest Paths on non-DAGs

If the graph has cycles, then we can't find a topological ordering. What could we try instead?

One thing we could try is just updating every single vertex.

Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: ~~We can start by taking a topological sort of the DAG!~~

Base Case: opt[s] = 0

Return Value: return all of opt.

Shortest Paths on non-DAGs

If the graph has cycles, then we can't find a topological ordering. What could we try instead?

One thing we could try is just updating every single vertex.

```
def shortest_paths(G, s):
    opt[s] = 0, all other opt[v] = ∞
    for all v in V:
        for all (u, v) in E:
            opt[v] = min(opt[v], opt[u] + w(u, v))
    return opt
```

What's wrong with this algorithm?

Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: Just update the distance to every single vertex.

Base Case: opt[s] = 0

Return Value: return all of opt.

Shortest Paths on non-DAGs

If the graph has cycles, then we can't find a topological ordering. What could we try instead?

One thing we could try is just updating every single vertex.

```
def shortest_paths(G, s):
    opt[s] = 0, all other opt[v] = ∞
    for all v in V:
        for all (u, v) in E:
            opt[v] = min(opt[v], opt[u] + w(u, v))
    return opt
```

It only finds distances for neighbors of s!

Subproblem: opt[v] = distance from s to v.

Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

Iteration Order: Just update the distance to every single vertex.

Base Case: opt[s] = 0

Return Value: return all of opt.

Shortest Paths on non-DAGs

Idea: Let's add a dimension to keep track of number of steps.

Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

Old Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

New Recurrence: opt[v][i] =

Shortest Paths on non-DAGs

Idea: Let's add a dimension to keep track of number of steps.

Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

Old Recurrence: opt[v] = min(opt[u] + w(u, v)) for all edges from u to v.

New Recurrence: opt[v][i] = min(
- opt[v][i-1]
- opt[u][i-1] + w(u, v) for all (u, v)

)

We're better off only taking i-1 steps.

Our ith step is (u, v).

Shortest Paths on non-DAGs

Idea: Let's add a dimension to keep track of number of steps.

Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

Recurrence: opt[v][i] = min(
- opt[v][i-1]
- opt[u][i-1] + w(u, v) for all (u, v)

)

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(???):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
    return opt[:][n-1]
```

What does i grow up to?

Shortest Paths on non-DAGs

Idea: Let's add a dimension to keep track of number of steps.

Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

Recurrence: opt[v][i] = min(
- opt[v][i-1]
- opt[u][i-1] + w(u, v) for all (u, v)

)

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
    return opt[:][n-1]
```

A shortest path can't take more than n-1 steps.

Shortest Paths on non-DAGs

Idea: Let's add a dimension to keep track of number of steps.

Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

Recurrence: opt[v][i] = min(
- opt[v][i-1]
- opt[u][i-1] + w(u, v) for all (u, v)

)

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
    return ???
```

What do we return?

Shortest Paths on non-DAGs

Idea: Let's add a dimension to keep track of number of steps.

Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

Recurrence: opt[v][i] = min(
- opt[v][i-1]
- opt[u][i-1] + w(u, v) for all (u, v)

)

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
    return opt[][n-1]
```

The entries at step n-1.

Shortest Paths on non-DAGs

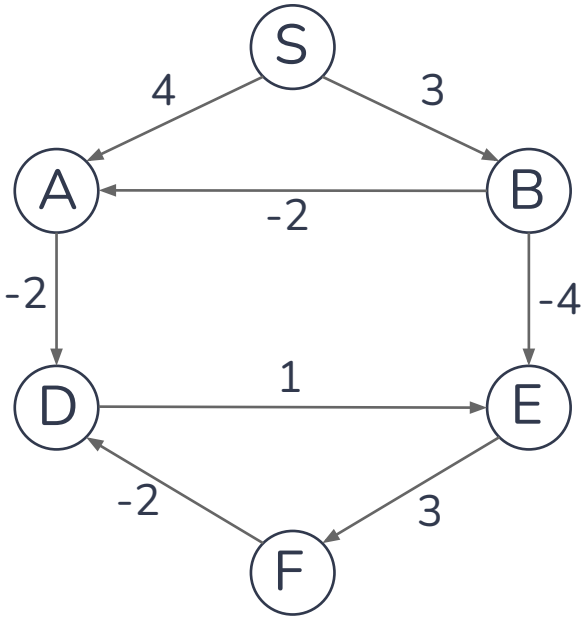
Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
    return opt[:][n-1]
```

Runtime:
The outer loop runs V times.
Inside, it iterates through all edges exactly once.
Thus, the runtime is O(VE).

Try it! When you do, think about if anything seems inefficient.

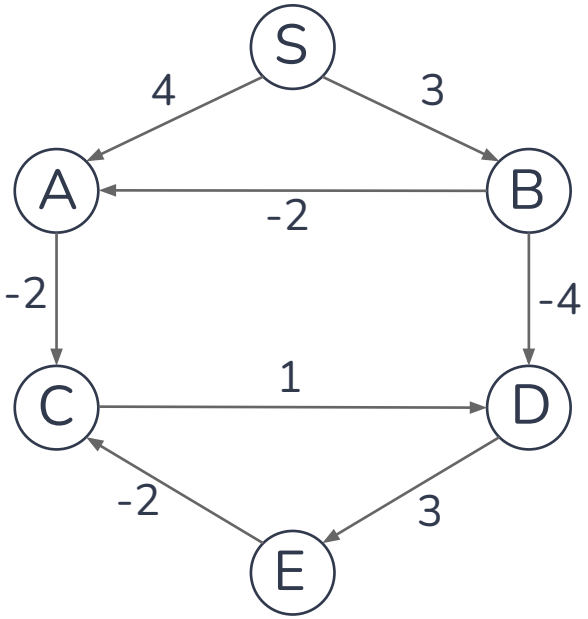
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]						
OPT[v][2]						
OPT[v][3]						
OPT[v][4]						
OPT[v][5]						

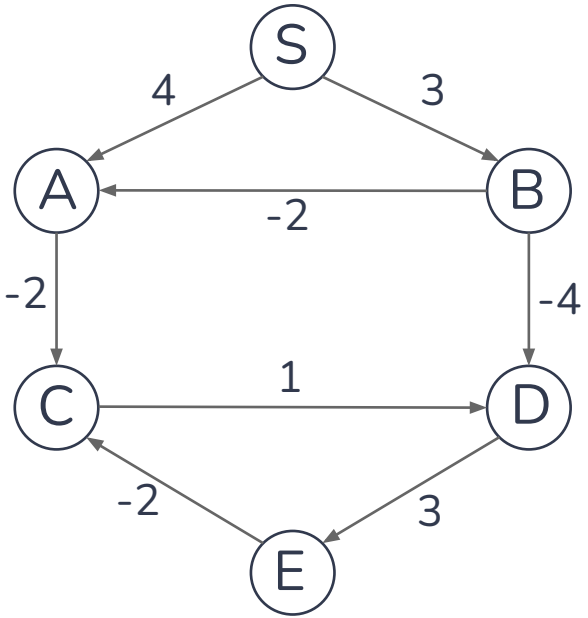
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]						
OPT[v][3]						
OPT[v][4]						
OPT[v][5]						

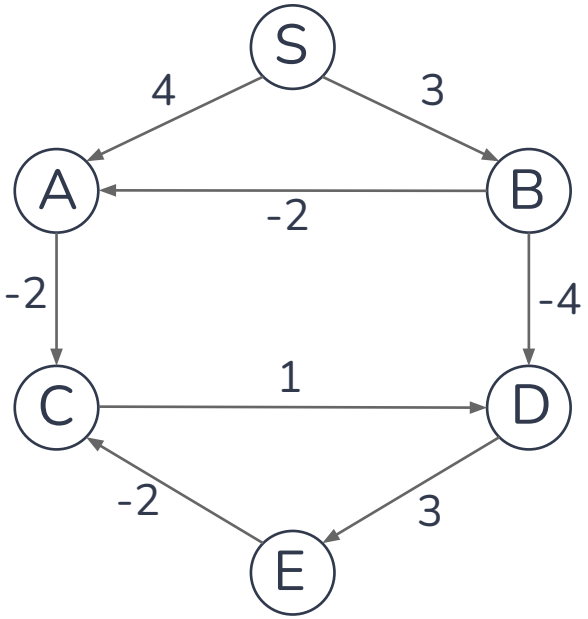
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0					
OPT[v][3]						
OPT[v][4]						
OPT[v][5]						

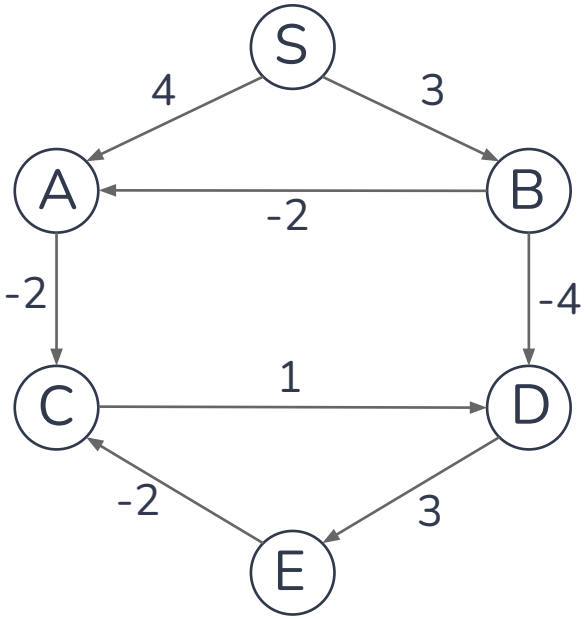
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	**1**				
OPT[v][3]						
OPT[v][4]						
OPT[v][5]						

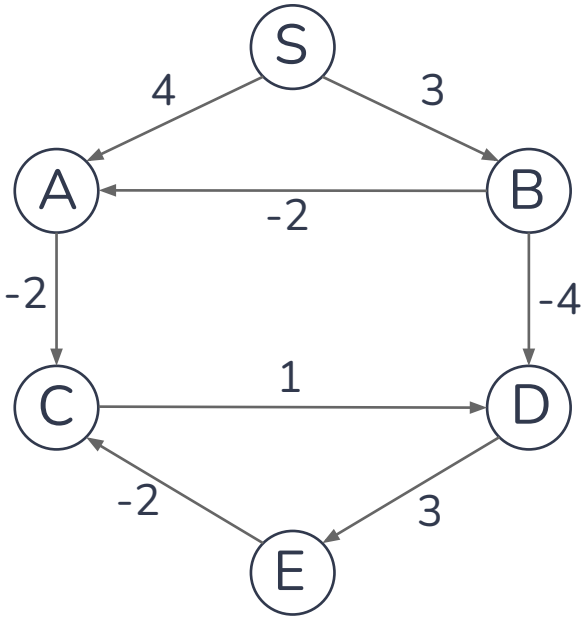
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	1	**3**			
OPT[v][3]						
OPT[v][4]						
OPT[v][5]						

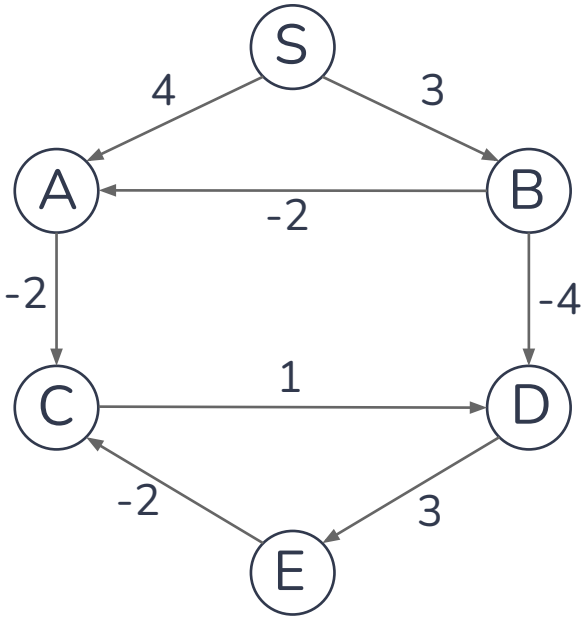
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	1	3	**2**		
OPT[v][3]						
OPT[v][4]						
OPT[v][5]						

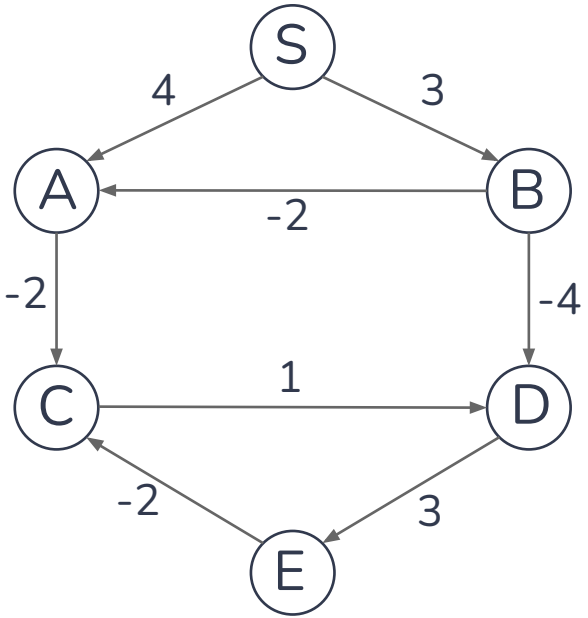
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	1	3	2	-1	∞
OPT[v][3]						
OPT[v][4]						
OPT[v][5]						

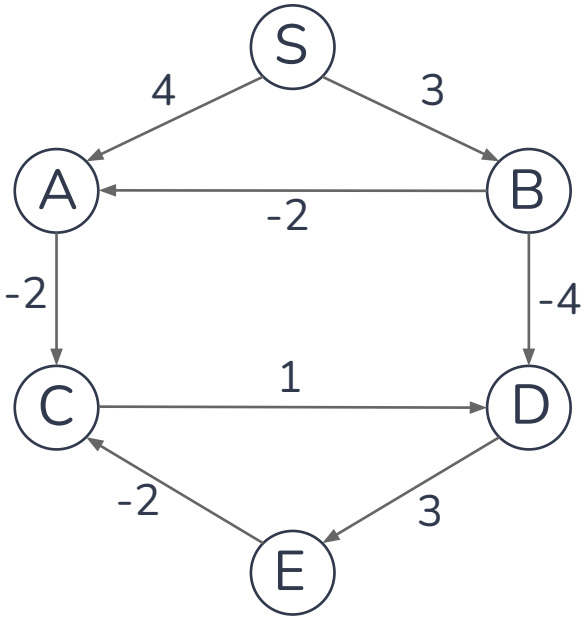
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	1	3	2	-1	∞
OPT[v][3]	0	1	3	-1	-1	2
OPT[v][4]						
OPT[v][5]						

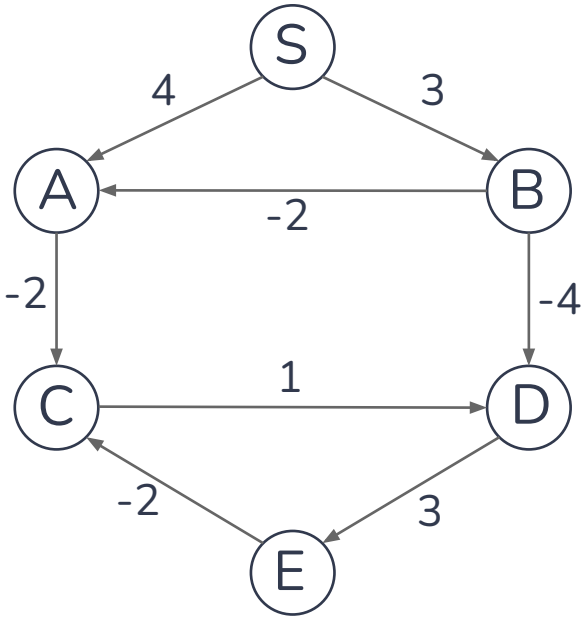
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	1	3	2	-1	∞
OPT[v][3]	0	1	3	-1	-1	2
OPT[v][4]	0	1	3	-1	-1	2
OPT[v][5]						

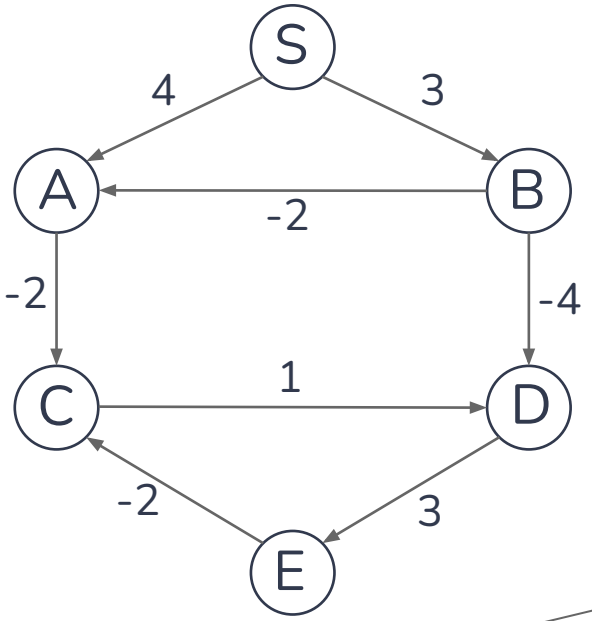
Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	1	3	2	-1	∞
OPT[v][3]	0	1	3	-1	-1	2
OPT[v][4]	0	1	3	-1	-1	2
OPT[v][5]	0	1	3	-1	-1	2

Shortest Paths on non-DAGs



Subproblem: opt[v][i] = shortest path from s to v in at most i steps.

	S	A	B	C	D	E
OPT[v][0]	0	∞	∞	∞	∞	∞
OPT[v][1]	0	4	3	∞	∞	∞
OPT[v][2]	0	1	3	2	-1	∞
OPT[v][3]	0	1	3	-1	-1	2
OPT[v][4]	0	1	3	-1	-1	2
OPT[v][5]	0	1	3	-1	-1	2

We could've stopped early!

Shortest Paths on non-DAGs

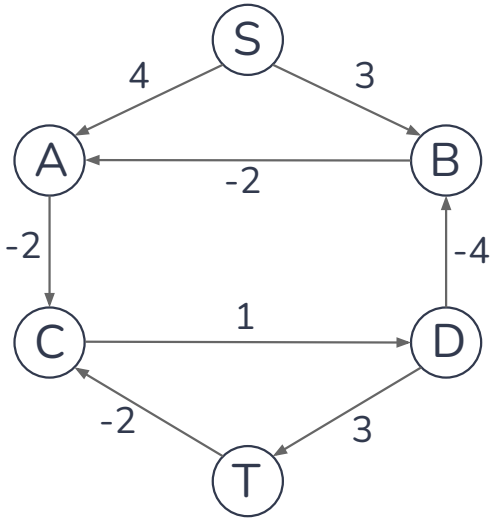
Each opt[v][i] value only depends on the opt[v][i-1] values, so if nothing changes in one loop it won't change in the next either!

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
        if all opt[v][i] = opt[v][i-1], stop the loop
    return opt[:][n-1]
```

A Tricky Case

What's the shortest path from S to T in this graph?
Why does this graph break the algorithm?

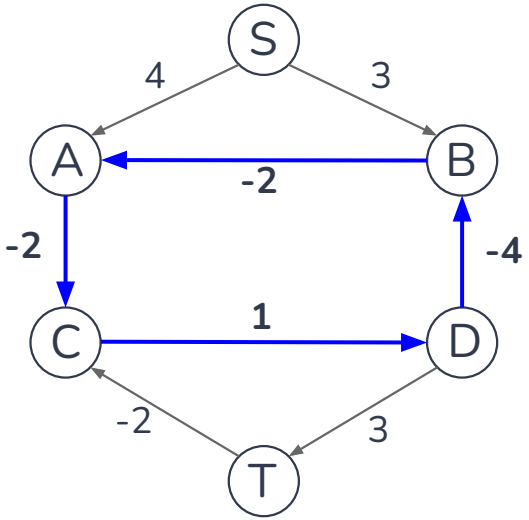
```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
        if all opt[v][i] = opt[v][i-1], stop the loop
    return opt[:][n-1]
```



A Tricky Case

This is a negative cycle! There is **no** shortest path from S to T. **Or, the shortest path has infinite length.**

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
        if all opt[v][i] = opt[v][i-1], stop the loop
    return opt[:][n-1]
```

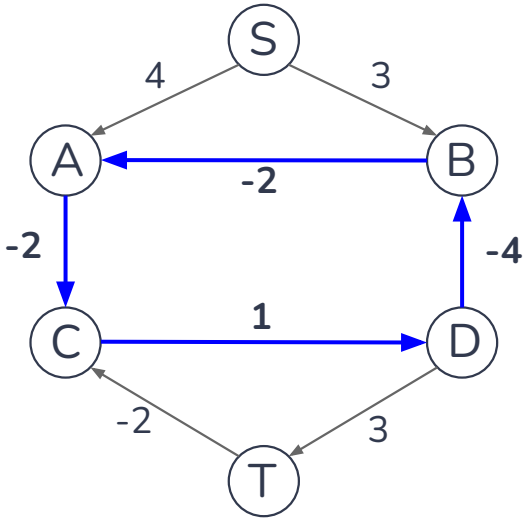


How should we detect whether a graph has a negative cycle?

A Tricky Case

This is a negative cycle! There is **no** shortest path from S to T. **Or, the shortest path has infinite length.**

```
def shortest_paths(G, s):
    opt[s][0] = 0, all other opt[v][i] = ∞
    for i in range(n+1):
        for v in V:
            opt[v][i] = opt[v][i-1]
            for (u, v) in E:
                opt[v][i] = min(opt[v][i], opt[u][i-1] + w(u, v))
        if all opt[v][i] = opt[v][i-1], stop the loop
    if any opt[v][n] ≠ opt[v][n-1], there's a negative cycle.
    return opt[:][n-1]
```



If a graph has a negative cycle, the length of its shortest paths will never stop decreasing.

Lecture Recap

Today we were introduced to the Bellman-Ford algorithm. We saw that it actually serves two different purposes:

- It detects whether a graph has a negative cycle.
- If the graph does not have a negative cycle, it finds the shortest paths from s to any point in the graph.

Though it has a relatively large runtime compared to the other graph algorithms O(EV), it can be stopped early in many cases, and there are many different implementation tricks to make it faster.