Lecture 17 - Bellman-Ford and Arbitrage

Eric A. Autry

Last time: Dijkstra's Algorithm

This time: Bellman-Ford and Arbitrage

Next time: Floyd-Warshall (Shortest Path meets DP Tables)

HW 6 due this Thursday, the 8th.

Project 2 due Friday the 16th.

Dijkstra's Algorithm Implementation

```
def dijkstra(graph, start):
    # Set initial dist and prev.
    for vertex in graph:
        vertex.dist = ∞
        vertex.prev = null
    start.dist = 0

    # Make the queue out of the distances.
    queue = makequeue(dists)

    # Loop while there are vertices left to visit.
    while not queue.isEmpty():
        # Find the next vertex to visit.
        u = deletemin(queue)

        # Check each neighbor of u.
        # Update predictions and previous vertex.
        for neighbor of u:
            if neighbor.dist > u.dist + length(u, neighbor):
                neighbor.dist = u.dist + length(u, neighbor)
                neighbor.prev = u
                decreasekey(queue, neighbor)
```

Dijkstra's Algorithm

Main Idea:

- Make distance predictions.
- Iteratively find the next closest vertex.
- Then use that vertex to update the predictions.

Major Assumption:

- Every vertex on the shortest path from the start to vertex v is closer to the start than vertex v.

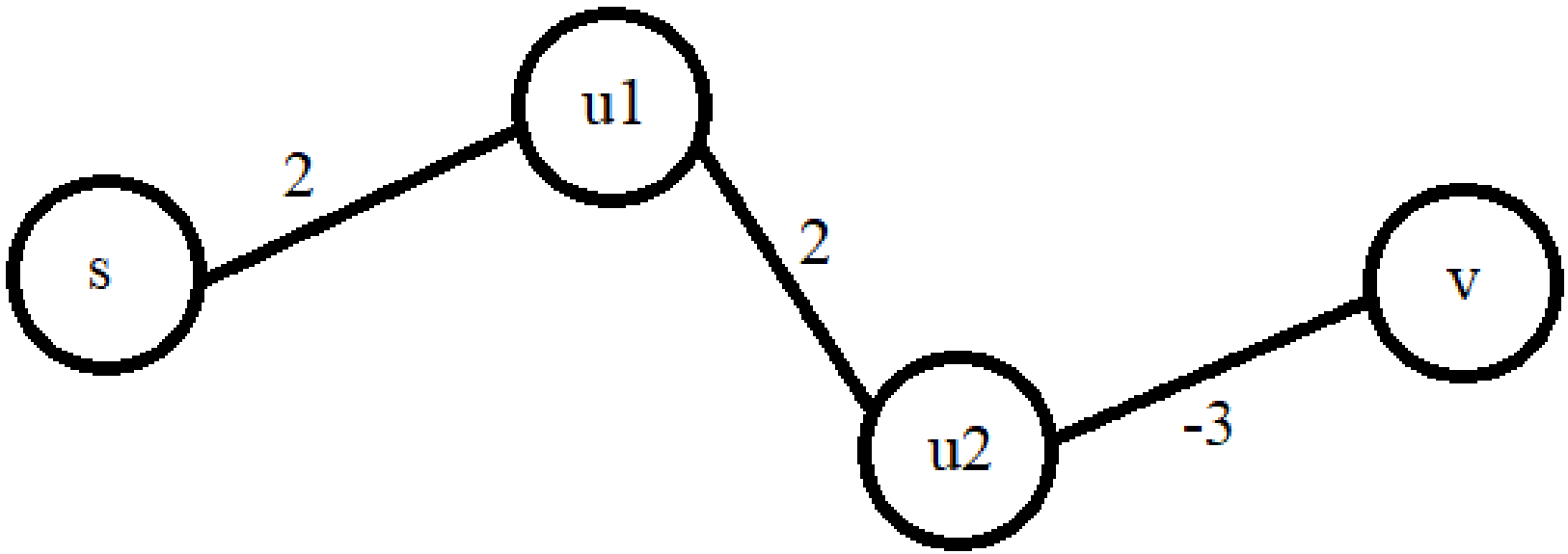
Problem:

- What happens if we have negative cost edges?

Negative Cost Edges

Problem:

- What happens if we have negative cost edges?



Our major assumption is wrong! Vertex v is closer than the vertices on the path...

Safe Updates

What can we do? We would still like to use the major idea of Dijkstra's algorithm if possible.

Important Note: in the algorithm, the `dist` values are either overestimated or are exactly correct, they are never underestimated.

They start at infinity and are only ever updated along an edge from u to v following the rule:

```
v.dist = min( v.dist, u.dist+length(u,v) )
```

This is because the distance to v cannot possibly be more than the distance to u plus the length of the edge from u to v.

Safe Updates

Distances are updated following the rule:

```
v.dist = min( v.dist, u.dist+length(u,v) )
```

This rule has two important properties:

1. It gives the correct distance to v if we are in the case where u was the previous vertex on the shortest path to v.

2. It will never make `v.dist` too small because the shortest path can never be larger than the prediction.

    - This is important because it means we are making safe predictions.

    - Repetitively updating the predictions can never hurt us as long as we follow this rule.

Sequences of Safe Updates

This update step is very useful: it is harmless, and if used carefully it can give us the correct distances.

In fact, we can view Dijkstra's Algorithm as a sequence of update operations performed in a specific order.

This sequence doesn't get us correct distances when there are negative edges, but maybe another sequence of updates can.

Sequences of Safe Updates

Can we find a sequence of updates that will give us the correct answer?

Consider the shortest path from vertex s to vertex t that goes through vertices u_1, u_2, u_3, ..., u_k.

What order do the predictions need to be made in to give a correct distance for vertex t?

$$s \to u_1, \quad u_1 \to u_2, \quad u_2 \to u_3, \quad \ldots, \quad u_k \to t$$

Note: other updates can occur because updates are safe. Once each of these updates occurs, it will fix any overestimated distances.

So as long as these updates are all performed in this order (with potentially other updates in between), we will arrive at the correct distances.

Sequences of Safe Updates

But we don't know the shortest path beforehand, how can we be sure we make those updates in that specific order?

Note: how many total updates could be required? $|V| - 1$

Idea: update along **all** of the edges in the graph $|V| - 1$ times.

- All of the necessary updates occurs **each** iteration.
- Since we iterate $|V| - 1$ times, we know that all of the necessary updates were eventually done in order.
- The first iteration *correctly* did update $s \to u_1$, the second iteration *correctly* did update $u_1 \to u_2$, etc.

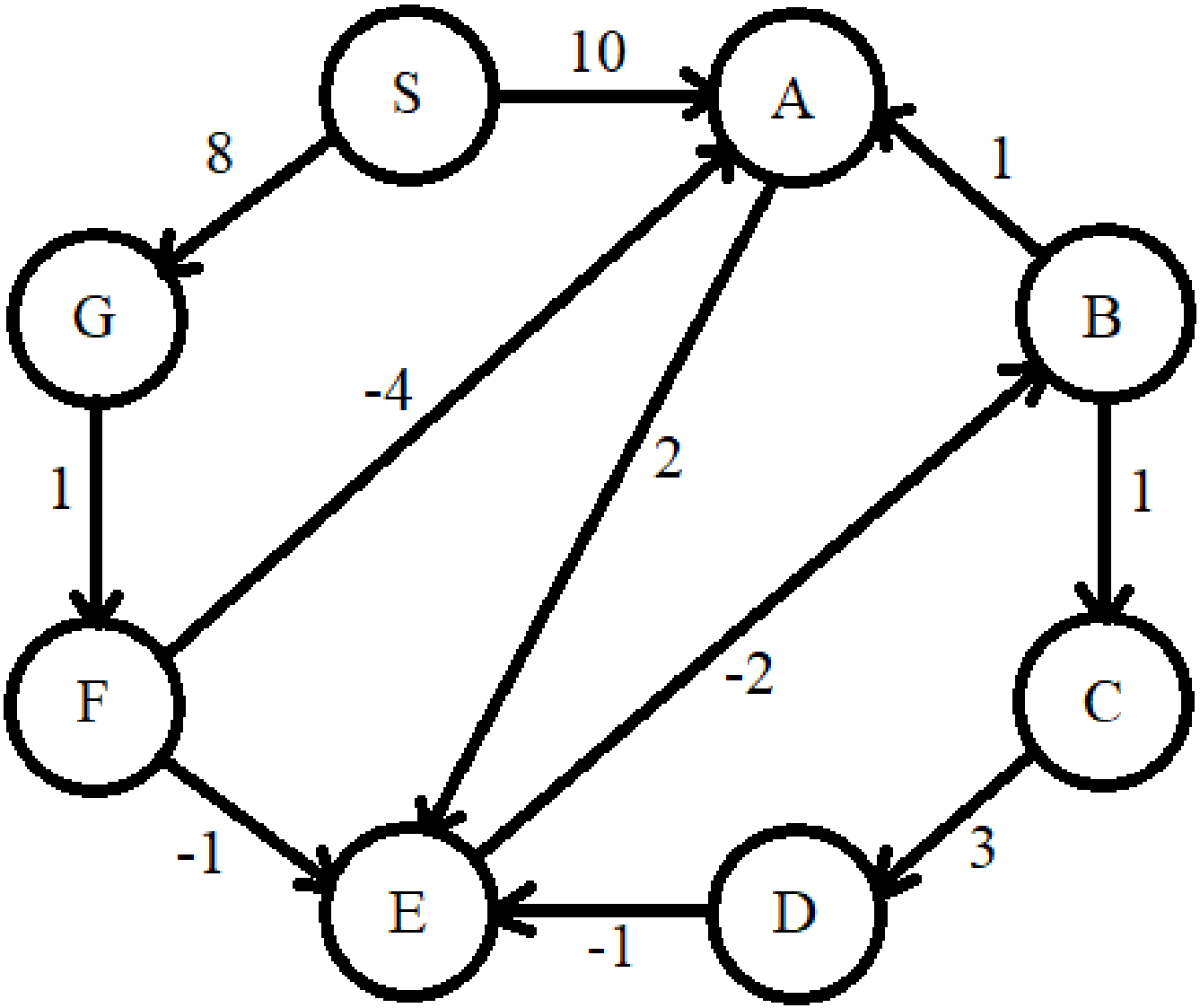
Repeating an update, or having other unnecessary updates, is fine because updates are safe

Once the correct update gets done, the value of the distance will be correct.

Bellman-Ford Implementation

```
def bellmanFord(graph, start):

    # Set initial dist and prev.
    for vertex in graph:
        vertex.dist = ∞
        vertex.prev = null
    start.dist = 0

    # Iterate |V| − 1 times.
    for iter in range(0, |V| − 1):

        # Look at each vertex.
        for u in graph:

            # Check each neighbor of u.
            # Update predictions and previous vertex.
            for neigh of u:
                # Only update if the new value is better!
                if neigh.dist > u.dist + length(u, neigh):
                    neigh.dist = u.dist + length(u, neigh)
                    neigh.prev = u
```

Bellman-Ford Example



Bellman-Ford Runtime

What is the runtime of Bellman-Ford?

Each edge is updated each iteration, with $|V| - 1$ iterations.

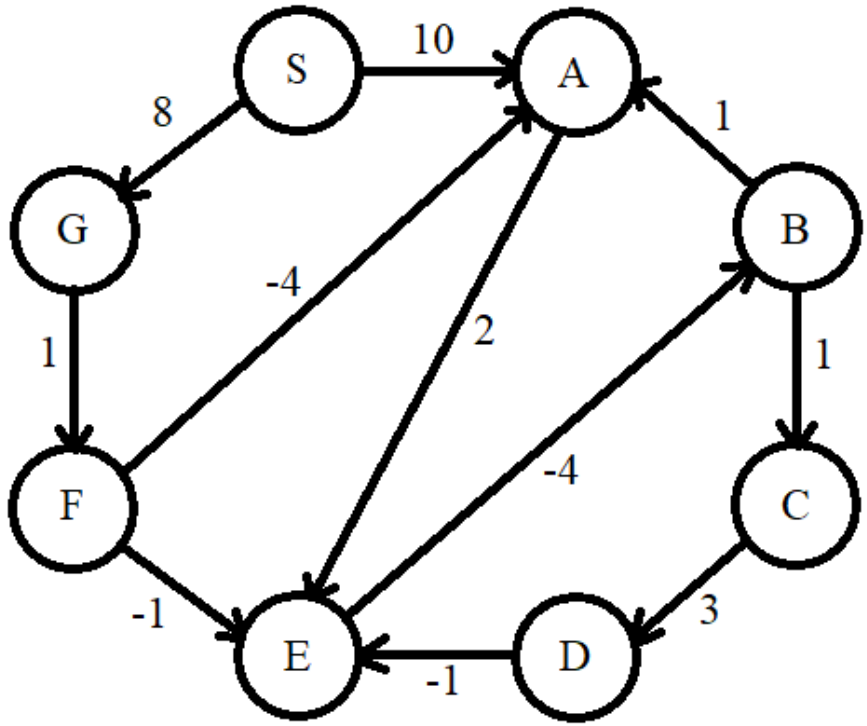
$O(|V| \cdot |E|)$

Recall that in a dense graph with n vertices, $|E| \in O(n^2)$, so worst case is $O(n^3)$.

Worse runtime than Dijkstra's, but can handle negative cost edges.

Negative Cost Cycles

What happens if we change the weight of the edge between E and B to -4?



The cost of going through the cycle: $E \to B \to C \to D \to E$ is -1.

What is the shortest path to E?

$$S \to G \to F \to A \to E \quad = \quad 7$$

But going around that cycle subtracts 1 **each time**!

Negative Cost Cycles

Shortest path algorithms are invalid when negative cost cycles exist because there cannot be shortest paths!

Bellman-Ford can actually detect these negative cost cycles:

- Run for 1 extra iteration, if any values change, there is a negative cost cycle.
- Why?
  - At the end of Bellman-Ford, all vertices have the correct shortest path distances.
  - After $|V| - 1$ iterations, even the longest possible path has been accounted for.
  - At this point, a distance could only change on a new update if a cycle is involved (i.e., the path had more than $|V|$ vertices).

Negative Cost Cycles

Application: finance.

Let's consider currency exchange rates.

Say that:

- 1 USD buys 0.82 Euro,
- 1 Euro buys 129.7 Yen,
- 1 Yen buys 12 Turkish Lira,
- 1 Turkish Lira buys 0.0008 USD

Then 1 USD buys:

$$0.82 \cdot 129.7 \cdot 12 \cdot 0.0008 \approx 1.02 USD$$

We made a 2% profit!

This is called arbitrage.

Negative Cost Cycles

Let's encode these exchange rates as a graph.

Say that we have exchange rates where one unit of currency c_i buys $R(c_i, c_j)$ units of currency c_j.

Our goal is to find a cycle of currencies $c_1, c_2, c_3, \ldots, c_k$ where:

$$R(c_1, c_2) \cdot R(c_2, c_3) \cdot \cdots \cdot R(c_k, c_1) > 1.$$

Note that this is equivalent to:

$$\frac{1}{R(c_1, c_2)} \cdot \frac{1}{R(c_2, c_3)} \cdot \cdots \cdot \frac{1}{R(c_k, c_1)} < 1.$$

Clever trick: take the log of both sides.

$$\log \frac{1}{R(c_1, c_2)} + \log \frac{1}{R(c_2, c_3)} + \cdots + \log \frac{1}{R(c_k, c_1)} < 0.$$

Negative Cost Cycles

We are looking for a cycle of currencies such that

$$\log \frac{1}{R(c_1, c_2)} + \log \frac{1}{R(c_2, c_3)} + \cdots + \log \frac{1}{R(c_k, c_1)} < 0.$$

So we will use these logarithmic values as our edge weights:

$$\texttt{length(u,v)} = \log \frac{1}{R(u, v)} = -\log R(u, v).$$

Once we have created our currency graph...

- If Bellman-Ford detects a negative cost cycle, there is an arbitrage opportunity.
- Consider the vertices that changed values in that final iteration and follow their `prev` values to find the cycle.

This will be project 3.

Shortest Paths between All Vertices

What do I do if I want to find the shortest paths between all pairs of vertices?

I could run Bellman-Ford $|V|$ times, taking each vertex as the starting vertex.

This would have a runtime of $O(|V|^2 \cdot |E|)$, which could be $O(n^4)$ worst case.

It turns out we can do better.

When we re-run Bellman-Ford, we are throwing away a lot of information. It seems like there should be a way to store that info...

Idea: consider a use-it-or-lose-it approach and use dynamic programming!

This will give us Floyd-Warshall (next time).

Course Feedback

I would like you all to take a course feedback survey at:

`http://duke.is/xdRkXT`

I would like this feedback to give me a better sense of where you all stand with the class, how you feel about the pacing and difficulty, and what your backgrounds are.

I will use this feedback to potentially adjust the pace of the lectures going forward.

This survey will be **completely anonymous**, so please feel free to share negative feedback along with any positive feedback you have.

Please complete the survey by **TODAY**. It should only take a couple minutes.