DSC40B: Theoretical Foundations of Data Science II

Lecture 14: *Shortest Path in Weighted Graphs – part I*

Instructor: Yusu Wang

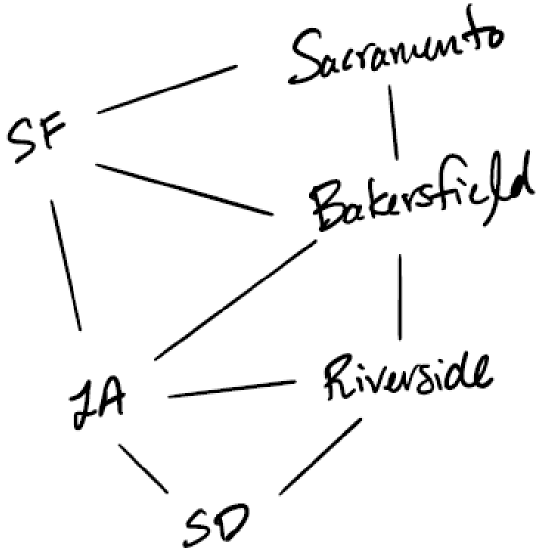
Prelude

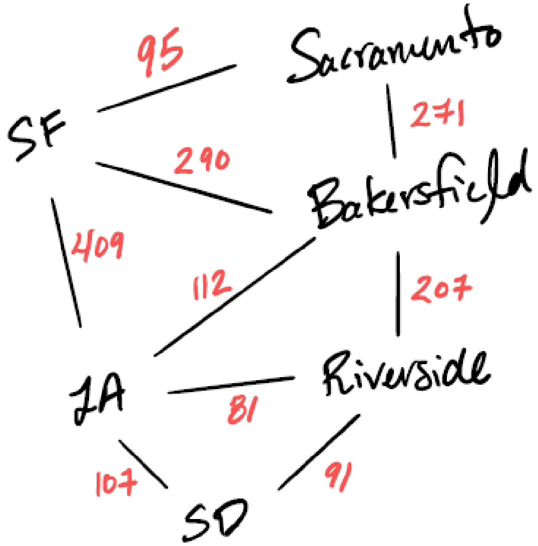
- Previously
  - Basics of graphs, representations, graph search strategies
  - BFS: also leads to shortest path distance in input graph
    - Note: graph is unweighted!

- Today:
  - Weighted graphs
    - where each edge has an edge weight
  - Properties of shortest paths in weighted graphs
  - Bellman-Ford algorithm for computing single-source shortest path for any weighted graphs

Weighted graphs, and shortest paths in them

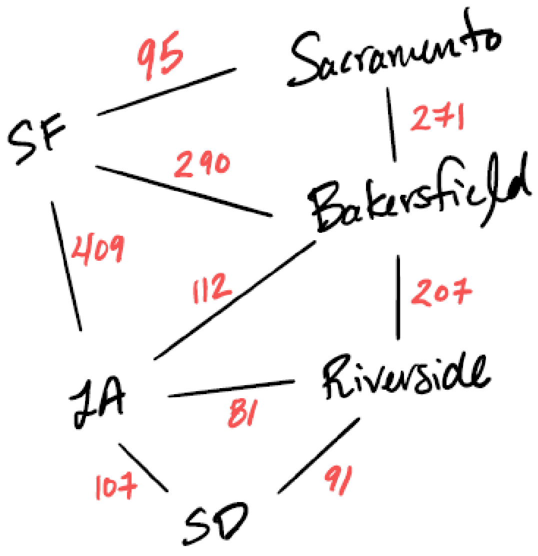
- Unweighted graph:
  - $G = (V, E)$
- Nodes and edges can carry meaning, and thus can carry weights as well.
- Example: transportation network





Weighted Graph

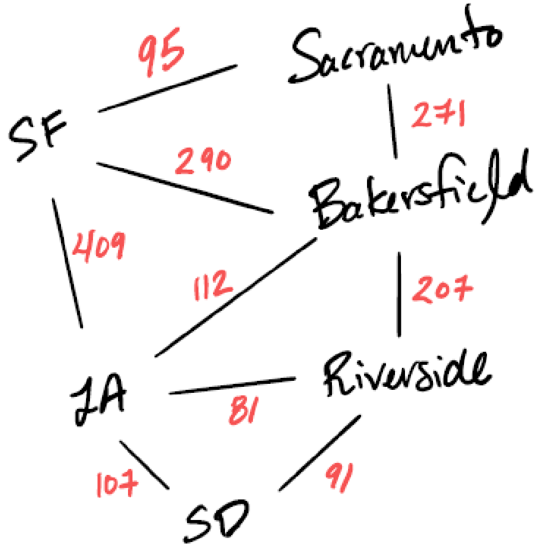
- (Edge) weighted graph $G = (V, E; \omega)$
  - is a graph $G = (V, E)$ together with an edge weight assignment map: $\omega: E \to R$
  - i.e., a graph where each edge e has a weight (real value) $\omega(e)$



- can be directed / undirected
  - weights can be positive/negative
- useful in many applications
  - strength of connection in a social network
  - distance in a transportation network
  - probability that two nodes interact in a protein-protein interaction network

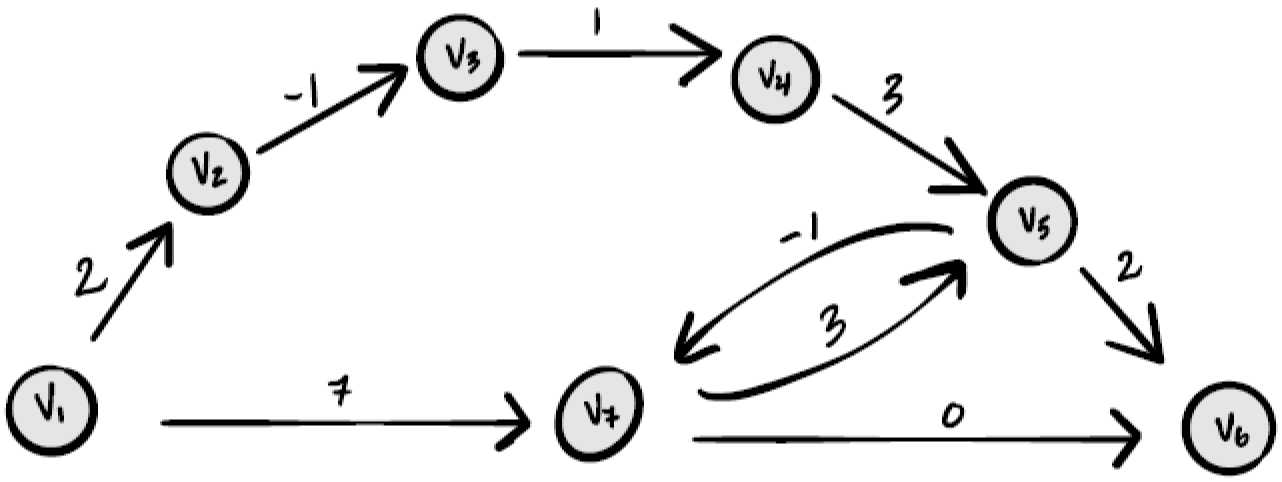
Path lengths

- Given a path in a weighted graph, its length is the total weights of all edges in the path.
- Examples:
  - SF, LA, Bakersfield, Riverside
    - length = 728
  - SF, LA, Riverside
    - length = 490
  - LA, SD, Riverside, LA, SF
    - length = 688
  - LA, SF
    - length = 409
  - LA, Bakerfield, SF
    - length = 402



Shortest Paths

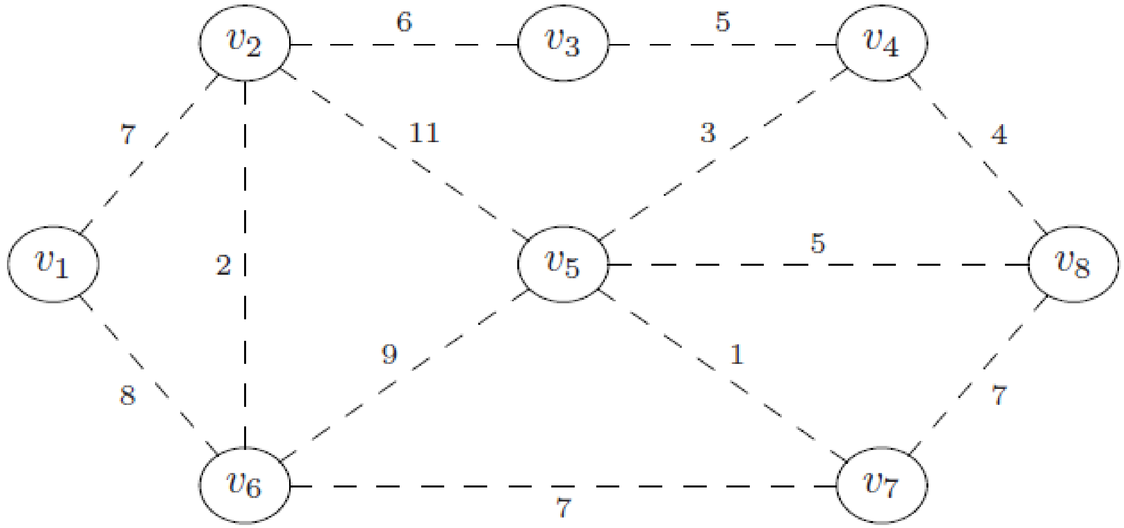
- A shortest path from u to v is a path from u to v with minimum length.



- Shortest path from v_1 to v_6?

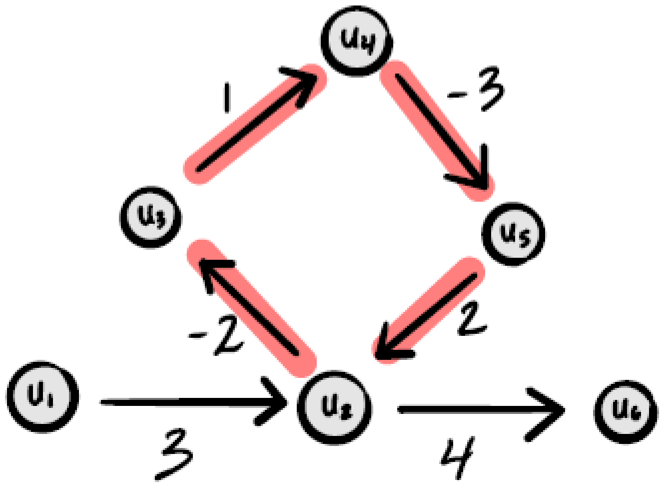
Shortest Paths

- A shortest path from u to v is a path from u to v with minimum length.
- The (shortest-path) distance from u to v is the length of the shortest path from u to v
- A shortest path from u to v may not be unique
  - but all shortest paths from u to v have same length



Properties of shortest paths

- Shortest path is not well-defined if the input graph has "negative cycles"
  - A negative cycle is a cycle whose length is negative

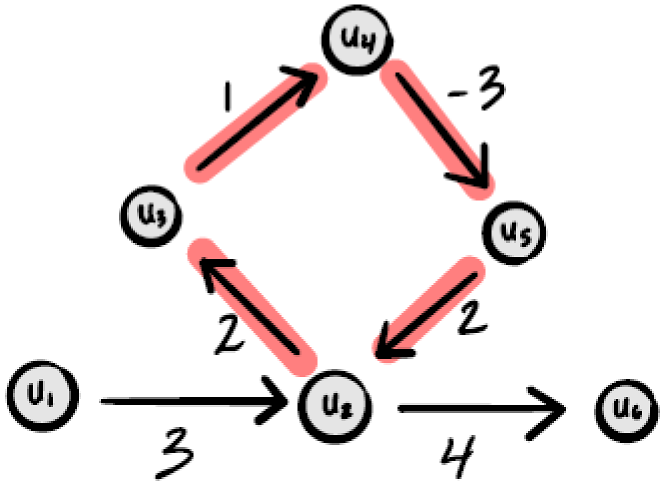


Negative weights are usually okay. Negative cycles make shortest path not well-defined.

Properties of shortest paths

- Shortest path is not well-defined if the input graph has "negative cycles"
  - A negative cycle is a cycle whose length is negative

- Assume we have a graph with no negative cycle
  - Then for any pair $u, v \in V$, there is always a shortest path that is simple.
    - If a path is not simple, there is a cycle inside, then removing this cycle can only make the path length shorter



Properties of shortest paths

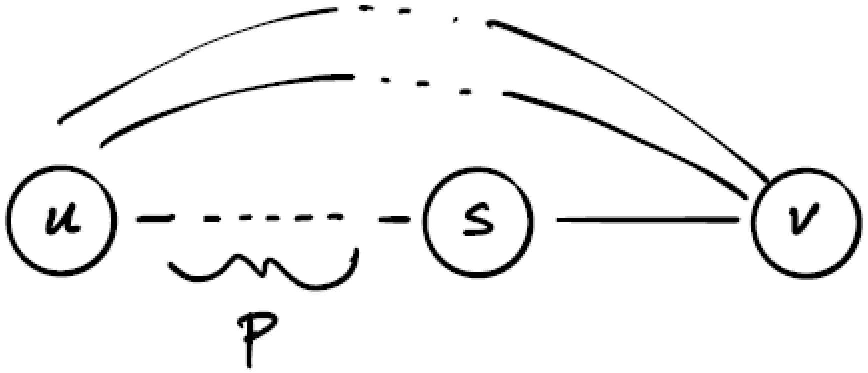
- Shortest path is not well-defined if the input graph has "negative cycles"
  - A negative cycle is a cycle whose length is negative

- Assume we have a graph with no negative cycle
  - Then for any pair $u, v \in V$, there is always a shortest path that is simple.
    - If a path is not simple, there is a cycle inside, then removing this cycle can only make the path length shorter

- Hence from now on, when we can assume that shortest paths are simple.

Properties of shortest path

Optimal Substructure Property (Theorem):
If $(u_1, u_2, \ldots, u_m)$ is a shortest path from u_1 to u_m, then any sub-path $(u_i, \ldots, u_j)$ is also a shortest path.



Property of shortest paths

- From now on, let $\delta(u, v)$ denote the (shortest path) distance from u to v

- Triangle inequality:
  - Suppose (z, v) is an edge. Then

    $$\delta(s, v) \leq \delta(s, z) + \omega(z, v)$$
  - One can view $\delta(s, z) + \omega(z, v)$ as the shortest path from s to v using edge (z, v) as last edge

  - If $\delta(s, v) = \delta(s, z) + \omega(z, v)$, then z is the predecessor of v along a shortest path from s to v
    - that is, (z, v) is the last edge along a shortest path from s to v
    - that is, z is the predecessor of v along the shortest path from s to v

Single-source shortest paths (SSSP) and Does BFS work for weighted graphs?

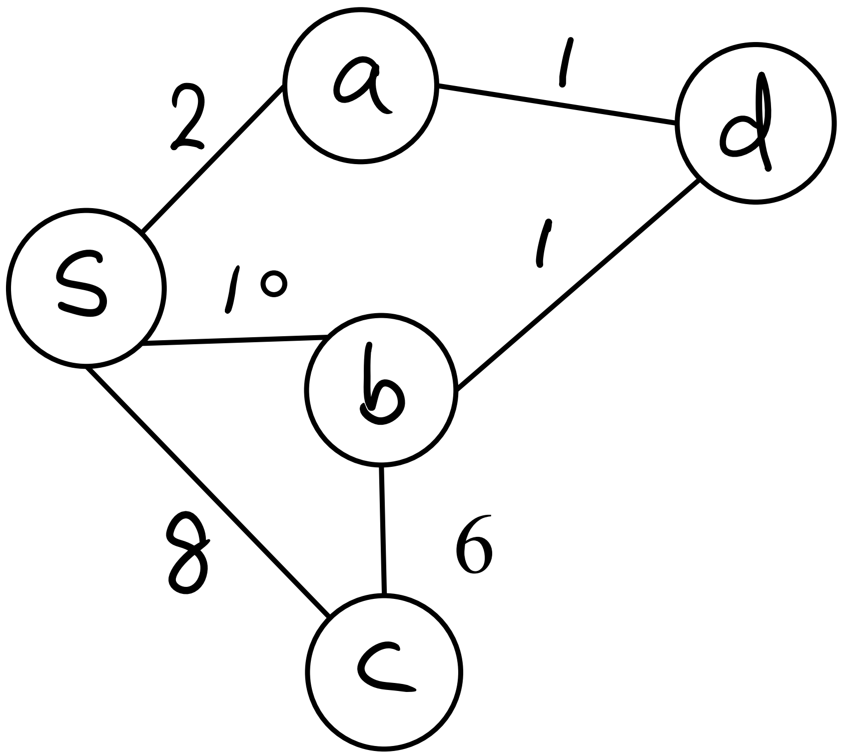
- Single-source shortest path (SSSP) problem:
  - Given a weighted graph $G = (V, E; \omega)$, and a source node s, compute the shortest path distance from s to all other nodes in V
    - i.e, compute $\delta(s, u)$ for all $u \in V$
- Once we have an algorithm for SSSP, we can use it to solve for all-pairs shortest path problem (where we compute shortest path distance among all pairs of nodes in V)
  - by simply running SSSP once using each node as source

- An unweighted graph $G = (V, E)$
  - can be thought of as a weighted graph where all edges have the same unit weight, i.e, $\omega(e) = 1$ for any edge $e \in E$

- Hence,
  - BFS can compute the shortest path lengths (to the source) for an unweighted graph, or equivalently, a graph where all edges have the same positive edge weights!

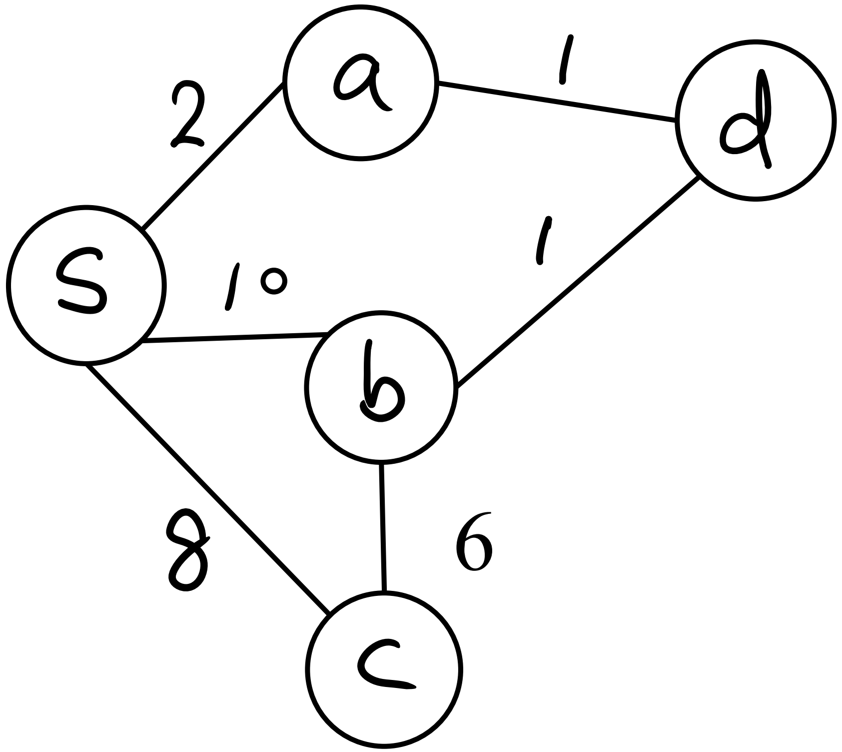
Can BFS idea work in general?

- Not really
- Example:



Can BFS idea work?

- Not really
- Example:



- Intuitively, what went wrong?
  - Recall BFS is a greedy algorithm and keeps exploring nodes in increasing distance to the source.
  - In BFS, at the time we explore a node u, we have found its correct distance to the source s already.
  - This however fails when edges have non-equal weights.

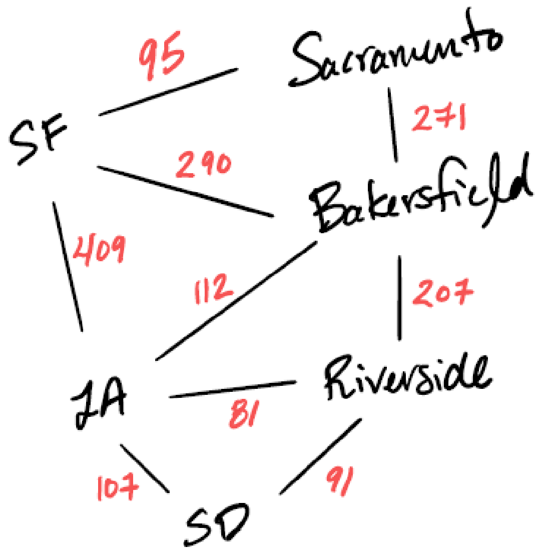
Can we use BFS to solve SSSP?

- Suppose edge weights are all positive integers
- Then here is an idea to compute SSSP:
- Input:
  - $G = (V, E; \omega)$ where $\omega: E \to \mathbb{Z}$ gives integer weights, and a source node $s \in V$
- Output:
  - The shortest path distance from s to all nodes in V

- Approach #0:
  - Step 1: For each edge with weight k, replace it by a path with k edges, each of which has weight 1. Call the new graph $\hat{G} = (\hat{V}, \hat{E})$.
  - Step 2: Use BFS on the new graph $\hat{G}$

Problem with Approach #0

- Only works when edges have positive integer weights
- Even for integer-weighted graphs, it can be highly inefficient.



- We need better algorithms for SSSP.

Key operation for SSSP: Edge Update

Two algorithms for SSSP

- Bellman-Ford algorithm
  - Works for any weighted graph $G = (V, E)$
  - Has time complexity $\Theta(V \cdot E)$
- Dijkstra algorithm
  - Works for graphs with positive edge weights
  - More efficient! Has time complexity $\Theta((V + E) \lg V)$ (which we will discuss in class), and can be made to run in $\Theta(V \lg V + E)$ time.
- Both algorithms
  - use an update( ) operation to keep track of shortest path estimates
  - perform it repeatedly till all shortest path distances to source are round

From now on, for simplicity, we use V and E to denote $|V|$ and $|E|$ in time complexity.

Estimated shortest path

- Fix the source node to be s
- Both algorithms keep track of the shortest path found so far,
  - we call these estimated shortest paths
  - set u.est = the length of estimated shortest path source s to u

- At the beginning, u.est = ∞ for all nodes other than the source s
- And $s.\mathrm{est} = 0$
- Then the algorithm will iteratively update shortest path estimates u.est when it finds better (shorter) path to reach it.

Estimated shortest path so far

- Fix the source node to be s
- Both algorithms keep track of the shortest path found so far,
  - we call these estimated shortest paths
  - set u.est = the length of estimated shortest path source s to u

- Key: during the update process, at any moment,
  - the estimated shortest path can only improve,
  - is at least as long as the true shortest path (i.e, u.est $\geq \delta(s,u)$),
  - and once it finds shortest distances, it will stay that way.

- For each node u, we will remember u's
  - predecessor along the estimated shortest path from s to u
  - u.est, the current estimated distance from s to u

Updating edges

- The way we update the estimates is via repeatedly performing "update(u, v)" operation over an edge $(u, v) \in E$
  - v has current predecessor
  - is u a better predecessor for v?
  - if yes, then we should update v.est and v's predecessor!

- In particular:

  Is the current shortest path from

  source $s \rightsquigarrow u \rightarrow v$

  shorter than the current shortest path from

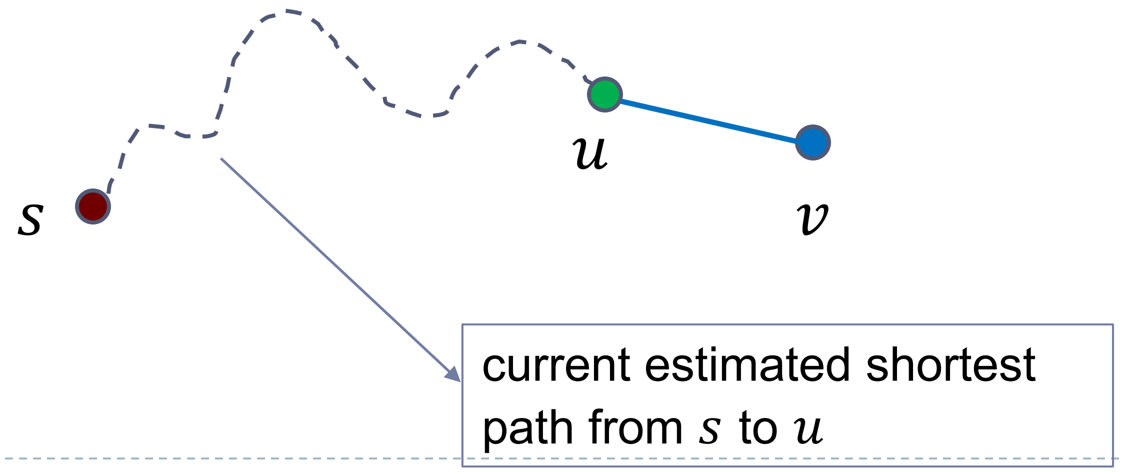
  source $s \rightsquigarrow v$'s current predecessor $\rightarrow v$?

  If yes, then we have discovered a shorter path to v, and we change v's predecessor and estimated distance.

Updating edges

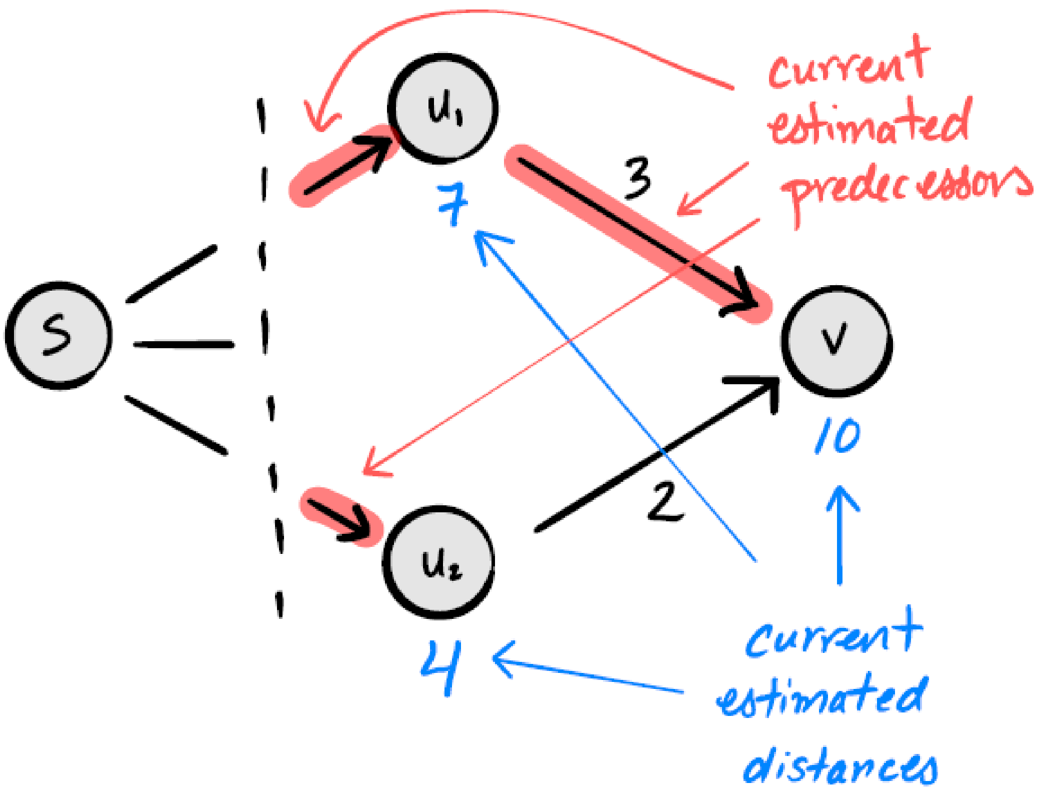
update(u, v) *// where $(u, v) \in E$ is an edge in graph*

- If $v.\text{est} > u.\text{est} + \omega(u, v)$
  - Then we found a better path from s to v
    - by first going from s to u, and then go to v through edge (u, v)
  - So we update $v.\text{est} = u.\text{est} + \omega(u, v)$ and set u to be v's predecessor
- Otherwise, we do nothing.



Example

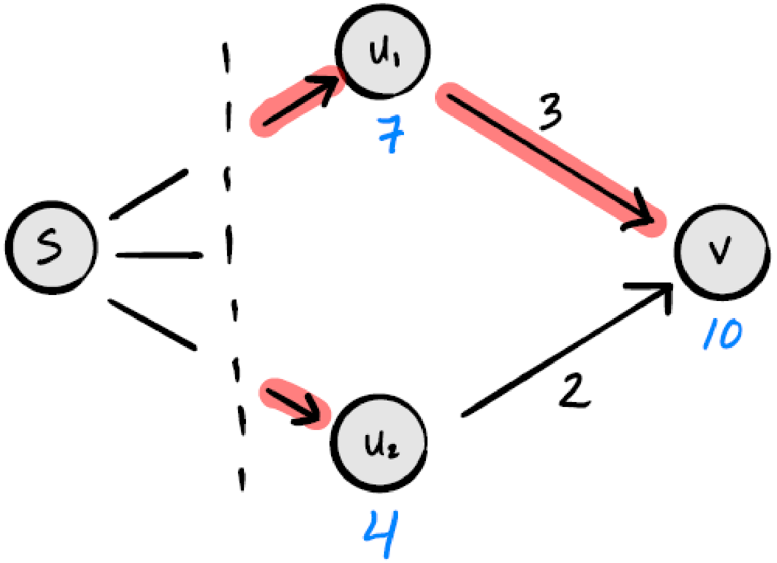
- Before update(u_2, v)



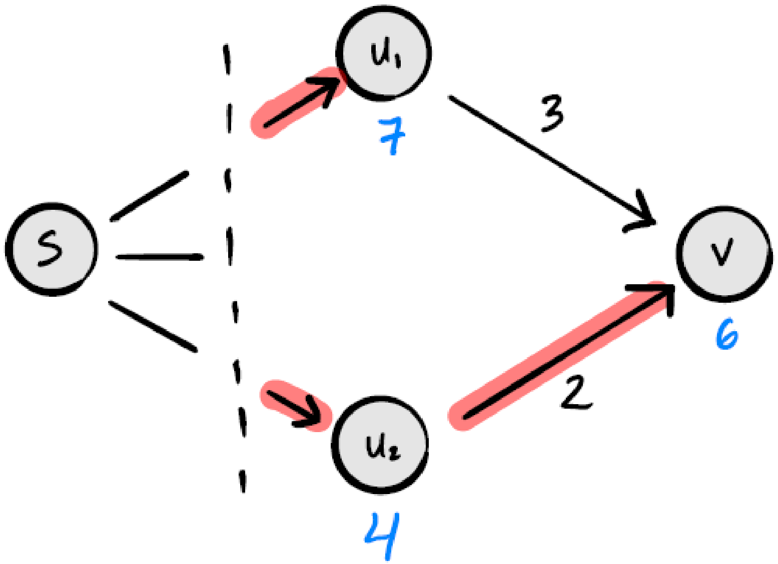
Example

- update(u_2, v)

Before update



After update



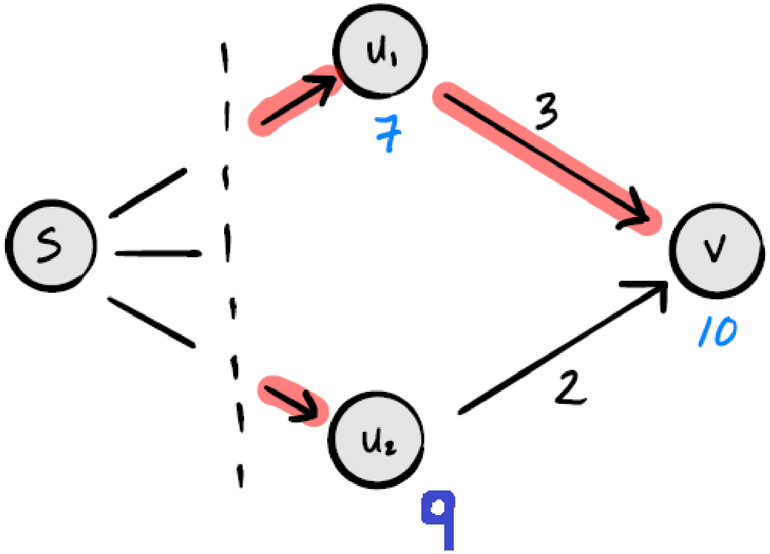
v.est = 10, u_2.est = 4 , $\omega(u_2, v) = 2$
$\Rightarrow$ v.est $>$ u_2.est $+\omega(u_2, v)$
$\Rightarrow$ find a shorter path from s to v !

So we update
v.est $=$ u_2.est $+\omega(u_2, v) = 6$
and set v's predecessor to u_2

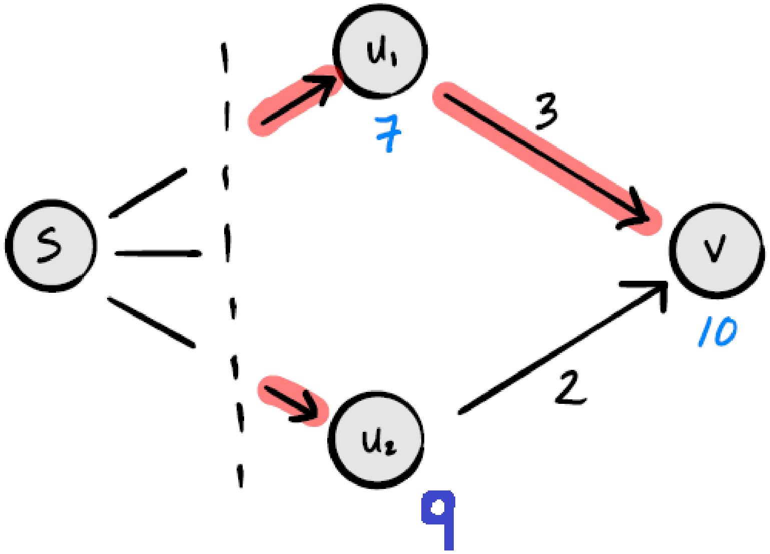
Another Example

- update(u_2, v)

Before update



After update



v.est $= 10,$ u_2.est = 9 , $\omega(u_2, v) = 2$
⇒ v.est $<$ u_2.est $+\omega(u_2, v)$
⇒ The path from s to u_2 then to v is not shorter than what already discovered for v!

Do Nothing!

Implementing update() in python

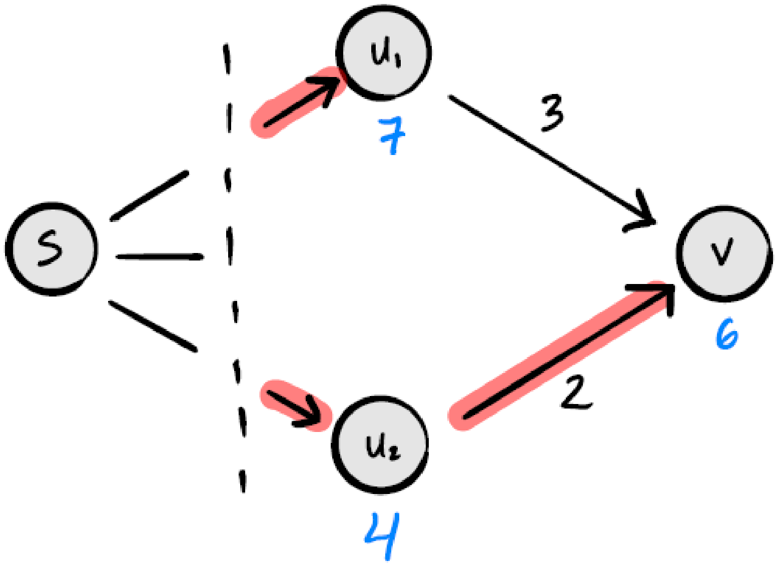
- To implement update(u,v) in python, let
  - est be a dictionary of estimated shortest path distances
  - predecessor be a dictionary of estimated shortest path predecessors
  - weights be a function which returns edge weights

Implementing update() in python

```python
def update(u, v, weights, est, predecessor):
    """Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```

Time complexity: $\Theta(1)$

When does an update discover a shortest path?



- So update(u_2, v) discovered a new estimated shortest path from s to v
- Is this the shortest path?
- Not necessarily: We might discover a shorter path, say, reaching v through u_1

When does an update discover a shortest path?

[Theorem Update]

Let u and v be graph nodes

Suppose:

- (a) current shortest path distance estimate u.est is correct
  - i.e, $u.\text{est} = \delta(s, u)$
- (b) there is a shortest path from s to v with u being v's predecessor

Then, after update(u, v), the estimated shortest path distance to v is correct

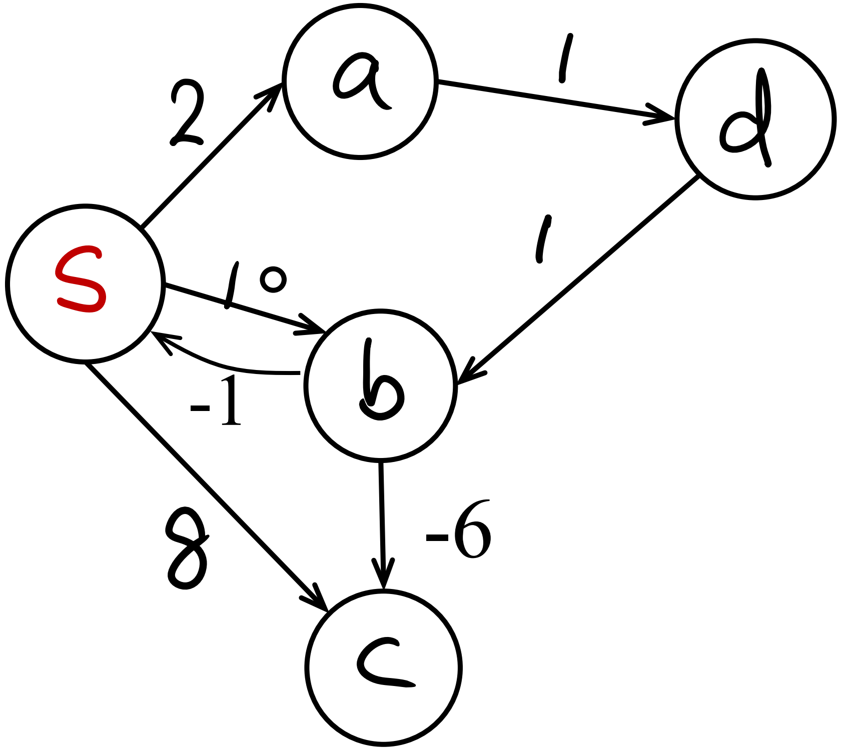
- i.e., after update, $v.\text{est} = \delta(s, v)$

Bellman-Ford shortest path algorithm

- [Theorem Update] implies that if we have already computed the shortest path distance from source s for those nodes whose shortest paths froms s s have k hops, then we can compute those shortest paths with $k+1$ hops via the update( ) operations.
  - In particular, let u be the predecessor of v along some shortest path from s to v, and assume u.est is already correct (i.e, u.est = $\delta(s, u)$)
  - then performing update(u,v) will render v.est = $\delta(s, v)$

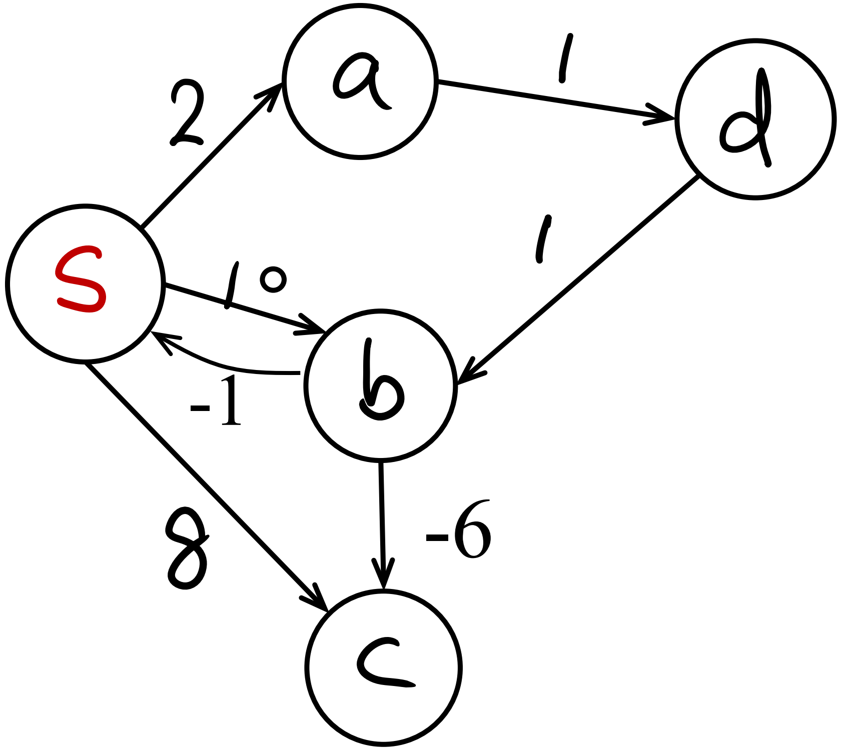
- [Observation]:
  - Any any moment, if v.est is already correct, then performing further update() on any edge will not change v.est.

More specifically



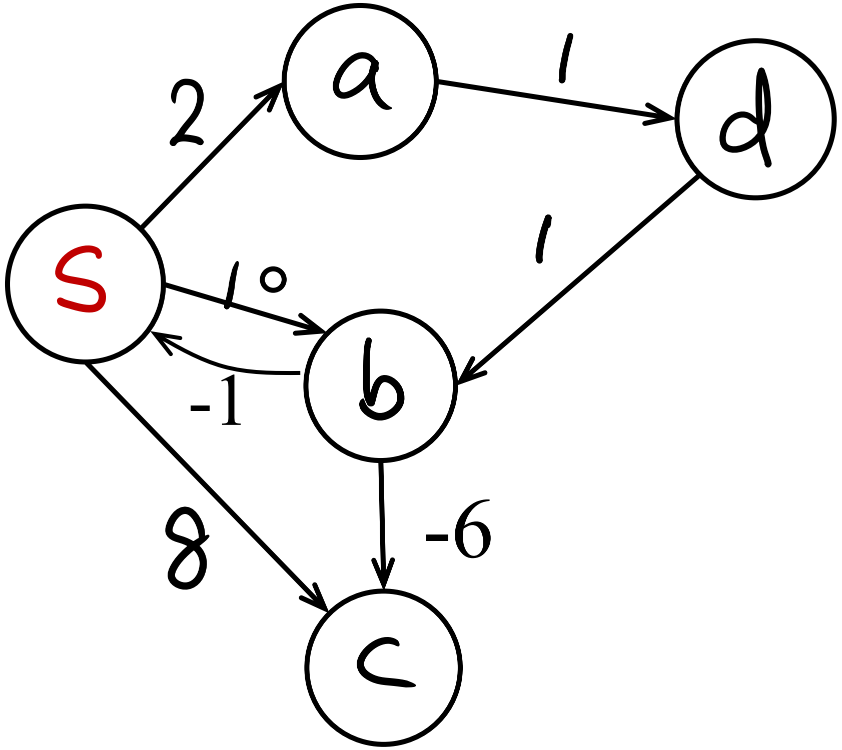
- At the beginning, we only know that $s.\text{est} = 0$
- For all other nodes $v \in V$, we set $v.\text{est} = \infty$

More specifically



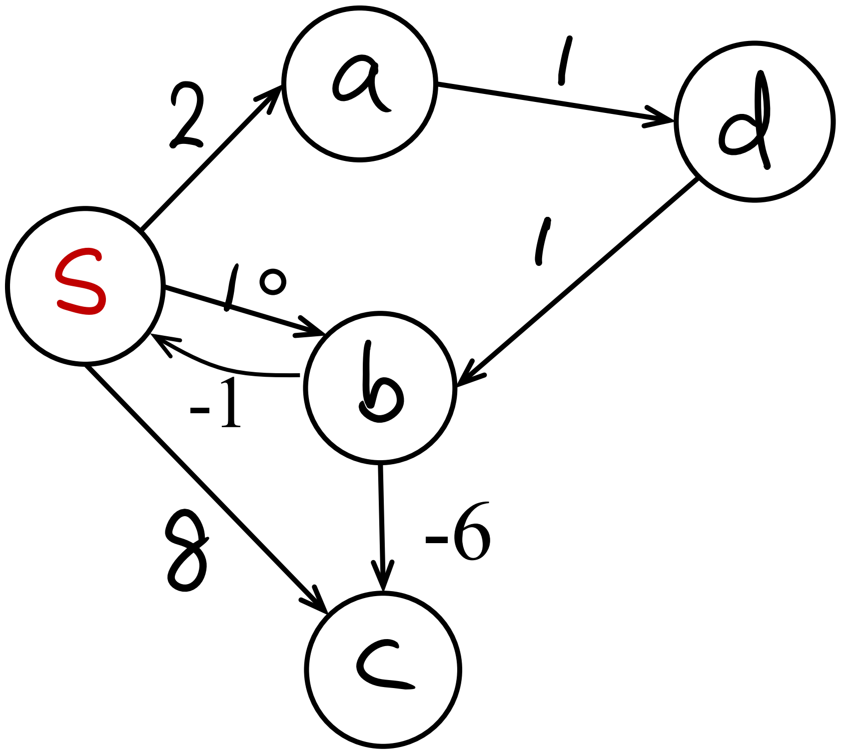
- Now perform update for **all edges** in E
- Afterwards, all nodes whose shortest path from s has only one edge are now guaranteed to be estimated correctly!

More specifically



- Now perform update for **all edges** in E again
- Afterwards, all nodes whose shortest path from s has at most two edges are now guaranteed to be estimated correctly!

More specifically



- Now perform update for **all edges** in E repeatedly …
- Afterwards, all nodes whose shortest path from s has at most more and more edges will be now guaranteed to be estimated correctly

Loop Invariant

- Suppose we perform "update all edges" k times
- Loop invariant:
  - Then all nodes whose shortest path from source s has $\leq k$ edges are guaranteed to be estimated correctly.

- Note that it is possible that some nodes whose shortest path has $> k$ edges are also estimated correctly.

- How many times should we perform "update all edges"
  - in order to guarantee that we find shortest path for all nodes?

- Note that shortest paths are simple
  - Hence a shortest path has at most $V - 1$ edges in it

- Hence after $r = V - 1$ rounds of "update all edges",
  - we can guarantee that the shortest path distances to all nodes in V are estimated correctly.

This is the idea behind Bellman-Ford Algorithm !

Bellman-Ford Algorithm in Python

```
def bellman_ford(graph, weights, source):
    """Assume graph is directed."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes) - 1):
        for (u, v) in graph.edges:
            update(u, v, weights, est, predecessor)

    return est, predecessor
```

- Setup takes ______________ time
- Each update takes ______________ time
- There are ____________ numbers of updates
- Total time complexity is _________________.

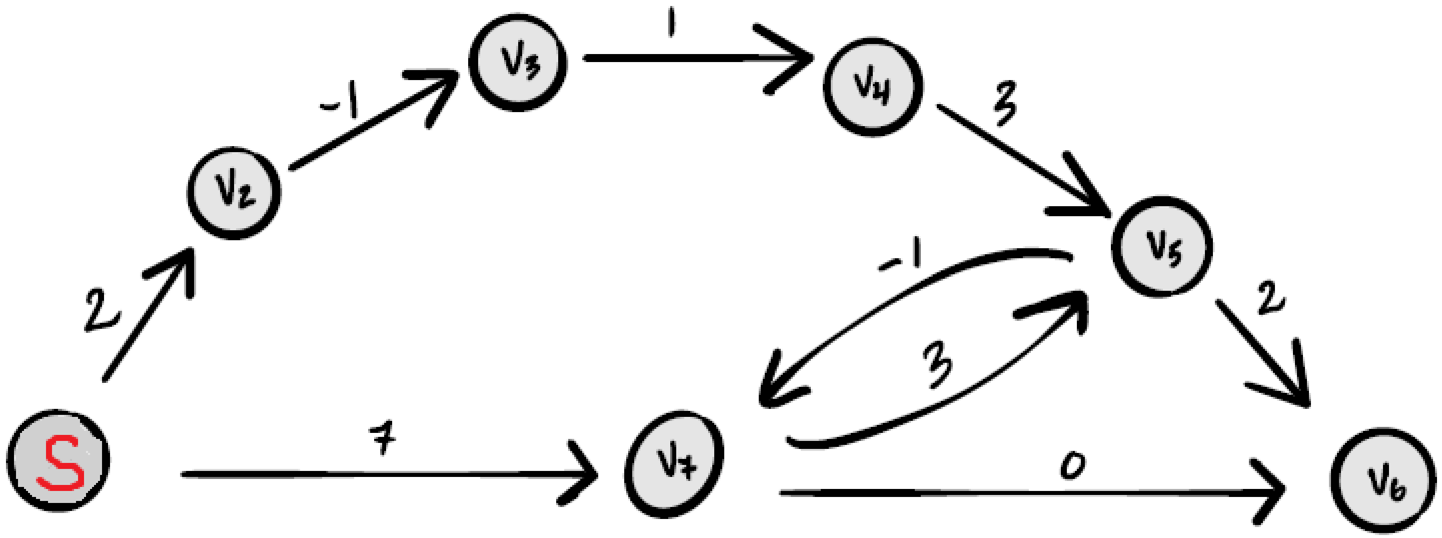
Bellman-Ford Algorithm in Python

```
def bellman_ford(graph, weights, source):
    """Assume graph is directed."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes) - 1):
        for (u, v) in graph.edges:
            update(u, v, weights, est, predecessor)

    return est, predecessor
```

- Setup takes ___ $\Theta(V)$ ___ time
- Each update takes ___ $\Theta(1)$ ___ time
- There are ___ $E \cdot (V-1)$ ___ numbers of updates
- Total time complexity is ___ $\Theta(V \cdot E)$ ___.

Example

- Suppose graph.edges returns edges in the following order:
  - $(v_3, v_4), (s, v_2), (v_2, v_3), (v_7, v_6), (v_5, v_7), (v_7, v_5), (v_4, v_5), (v_5, v_6), (v_5, v_4), (s, v_7)$



Early-stopping and negative cycles

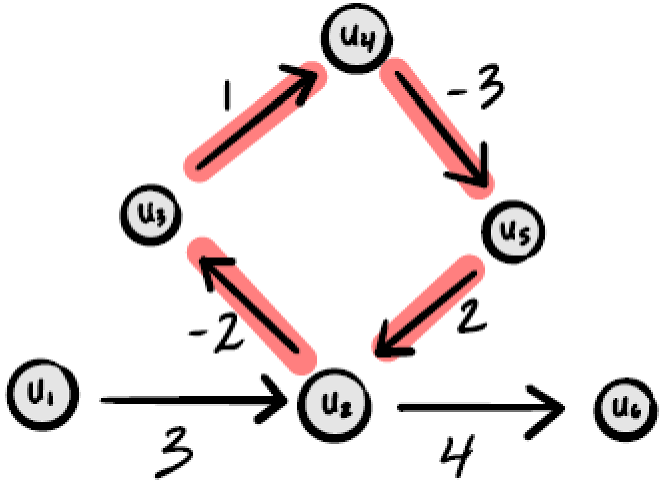
Early-stopping

- Bellman-Ford may not need to run $V - 1$ iterations
- If there is no distance change after a round, we can stop (called early-stopping)

```
def bellman_ford(graph, weights, source):
    """Early stopping version."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes) - 1):
        any_changes = False
        for (u, v) in graph.edges:
            changed = update(u, v, weights, est, predecessor)
            any_changes = changed or any_changes
        if not any_changes:
            break
    return est, predecessor
```

Negative Cycles

- Recall: if a graph has negative cycle(s), then the shortest paths are not well-defined

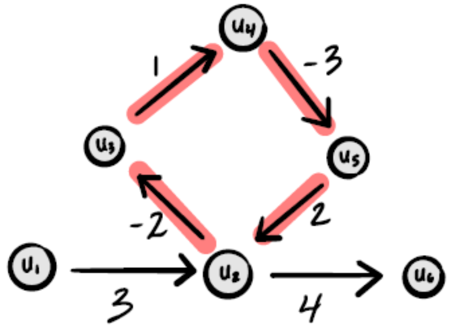


Negative Cycles

- Recall: if a graph has negative cycle(s), then the shortest paths are not well-defined

- If a graph does not have any negative cycle, then the estimated distances stop changing after $V - 1$ iterations
  - Why?

- But if a graph has negative cycle(s), then some estimated distances continue to decrease even after V iterations



Negative Cycles

- If a graph does not have any negative cycle, then the estimated distances stop changing after $V - 1$ iterations
- But if a graph has negative cycle(s), then some estimated distances continue to decrease even after V iterations

- Hence Bellman-Ford can be modified to also detect negative cycles
  - Run a V iteration of "update all edges"
  - If any estimated distance is still decreasing, a negative cycle exists

Modified Bellman-Ford with early stopping and negative cycle detection

```
def bellman_ford(graph, weights, source):
    """Early stopping version, detects negative cycles."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes)):
        any_changes = False
        for (u, v) in graph.edges:
            changed = update(u, v, weights, est, predecessor)
            any_changes = changed or any_changes
        if not any_changes:
            break
    # this will be True if negative cycles exist
    contains_negative_cycles = any_changes
    return est, predecessor, contains_negative_cycles
```

FIN