

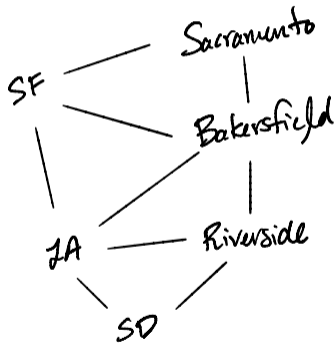
DSC 40B

Theoretical Foundations II

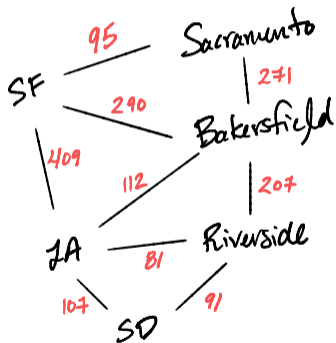
Lecture 14 | Part 1

Shortest Paths in Weighted Graphs

Google Maps



Google Maps



Weighted Graphs

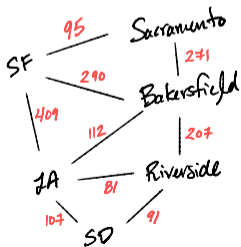
An **edge weighted graph** $G = (V, E, \omega)$ is a triple where (V, E) is a graph and $\omega : E \rightarrow \mathbb{R}$ maps each edge to a **weight**.

- ▶ Can be directed or undirected.
- ▶ In general, weights can be positive, negative, zero.
- ▶ Many uses, such as representing **metric spaces**.

Path Lengths

The **length** of a path in a **weighted graph** (usually) refers to the total weight of all edges in the path.

Example: (SD, Riverside, Bakersfield, SF)

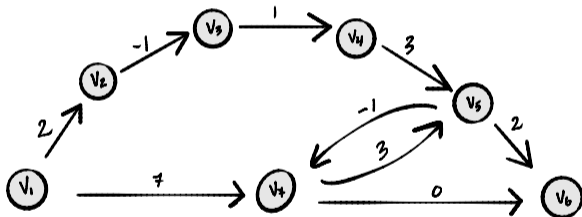


Shortest Paths

- ▶ A **shortest path** between u and v is a path between u and v with minimum length.
 - ▶ In other words, minimum total weight.

Example

What is the shortest path from v_1 to v_6 ?



Path:

Length:

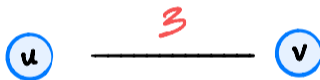
Today (and next time)

How do we find shortest paths in weighted graphs?

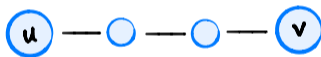
Idea #0

- ▶ Does BFS work?
 - ▶ **No**, not really. Only if all weights are the same.
- ▶ Can we “convert” a weighted graph to an unweighted one?

Idea #0



Idea #0

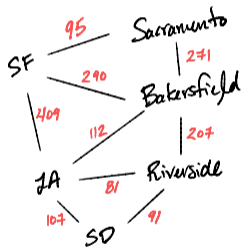


Idea #0

- ▶ Step 1: “Convert” weighted graph to unweighted one with dummy nodes.
- ▶ Step 2: Call BFS on this new graph.

Idea #0

- ▶ **Very inefficient** for large weights.



- ▶ What if edge weights are floats, or negative?

Ideas #1 and #2

- ▶ We'll look at two algorithms: **Bellman-Ford** and **Dijkstra's**.

INPUT: weighted graph, source vertex s .

OUTPUT: shortest paths from s to every other node.

- ▶ Both work by:
 - ▶ keeping track of shortest known path (**estimates**).
 - ▶ iteratively **updating** these until they're correct.

Shortest Path Estimates

- ▶ B-F and Dijkstra's keep track of the shortest paths found so far; we call these the **estimated shortest paths**.
- ▶ For each node u , remember u 's:
 - ▶ predecessor in estimated shortest path;
 - ▶ distance from source s in estimated shortest path.
- ▶ **Key:** estimated distance will always be \geq actual distance.

Updates

- ▶ Both algorithms work by iteratively updating their estimates.
- ▶ On each iteration, consider a new edge (u, v) . Ask: is the best known shortest path from

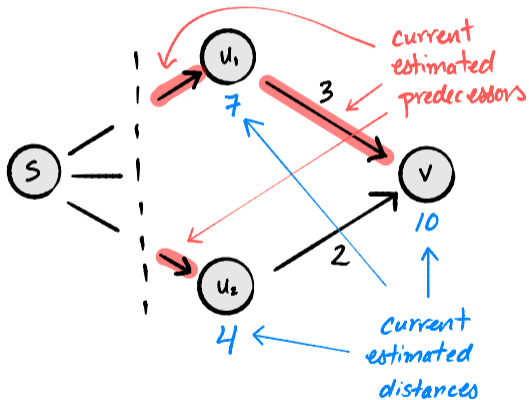
$\text{source} \rightarrow \dots \rightarrow u \rightarrow v$

shorter than the best known shortest path from

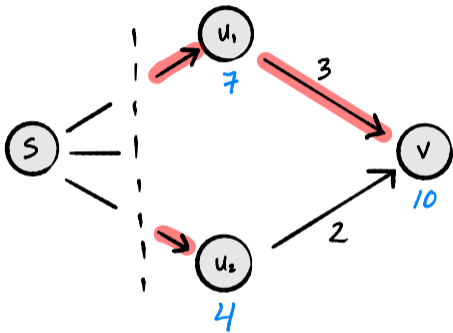
$\text{source} \rightarrow \dots \rightarrow \text{predecessor}[v] \rightarrow v?$

- ▶ If it is, we have discovered a shorter path to v .

Example: Updating (u_2, v) :

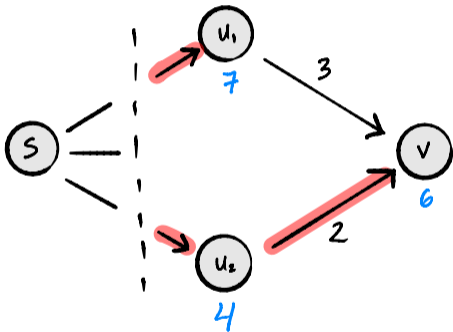


Example: Updating (u_2, v) :



$$\begin{aligned} & \text{estimated length of } s \rightarrow u_2 \rightarrow v \\ &= (\text{estimated length of } s \rightarrow u_2) + w(u_2, v) \\ &= 4 + 2 = 6 < 10 \end{aligned}$$

Example: After Updating (u_2, v) :



A shorter path has been found.

Updating, in Code

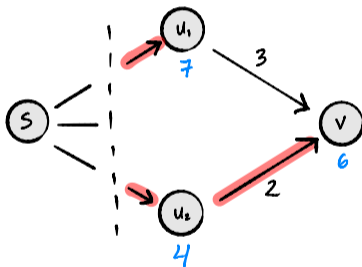
- ▶ Let:
 - ▶ `est` be a dictionary of estimated shortest path distances.
 - ▶ `predecessor` be a dictionary of estimated shortest path predecessors.
 - ▶ `weights` be a function which returns edge weights.

Updating, in Code

```
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
```

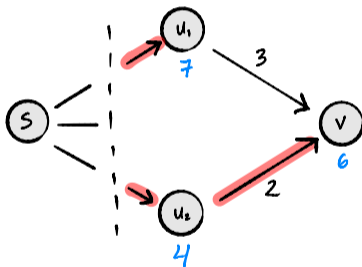
- ▶ Time complexity: _____

When does an update discover a shortest path?



- ▶ Suppose updating (u_2, v) finds a shorter path to v .
- ▶ True or False: the actual shortest path must go through u_2 .

When does an update discover a shortest path?



- ▶ Suppose updating (u_2, v) finds a shorter path to v .
- ▶ True or False: the actual shortest path must go through u_2 .
- ▶ **False**: we might later discover a better path to u_1 .

When does an update discover the shortest path?

- ▶ Let (u, v) be an edge.
- ▶ Suppose:
 - ▶ the actual shortest path to u has been found;
 - ▶ the actual shortest path to v goes through (u, v) .
- ▶ Then after updating (u, v) , the estimated shortest path to v is correct.

DSC 40B

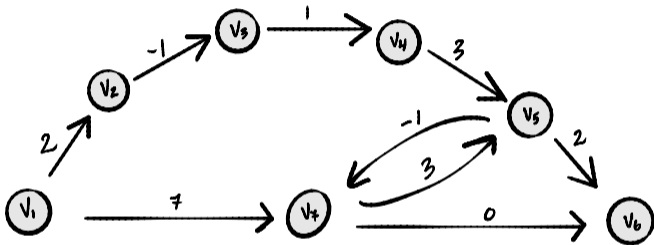
Theoretical Foundations II

Lecture 14 | Part 2

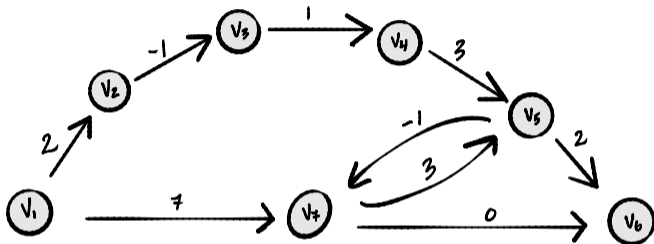
Bellman-Ford

Intuition

- ▶ Shortest paths that have many edges are “harder” to discover.
 - ▶ May require many updates.
- ▶ Shortest paths that have few edges are “easier” to discover.
- ▶ Once we’ve discovered all of the shortest paths with few edges, it makes it easier to discover the shortest paths with more edges.

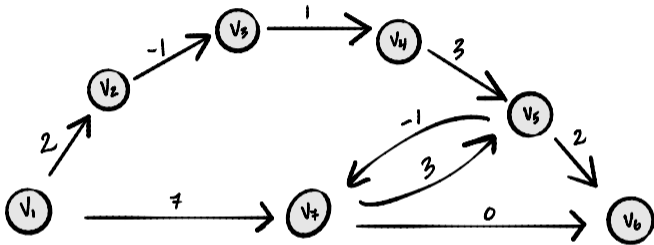


Updating All Edges



- ▶ Suppose we update all of the edges, one by one.
- ▶ Then all nodes whose shortest path from s has only **one** edge are **guaranteed** to be estimated correctly.

Updating All Edges



- ▶ Suppose we update all of the edges again.
- ▶ Then all nodes whose shortest path from s has at most **two** edges are **guaranteed** to be estimated correctly.

Loop Invariant

- ▶ One iteration: update all edges in arbitrary order.
- ▶ Loop invariant: After α iterations, all nodes whose shortest path from s has $\leq \alpha$ edges are guaranteed to be estimated correctly.

The Bellman-Ford Algorithm

```
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(?):  
        for (u, v) in graph.edges:  
            update(u, v, weights, est, predecessor)  
  
    return est, predecessor
```

Bellman-Ford

- ▶ Claim: each node must have a shortest path which is simple¹.
- ▶ The most edges a simple path can have is $|V| - 1$
- ▶ Idea of Bellman-Ford: iteratively update all edges, repeat $|V| - 1$ times.

¹Edge case: cycles of weight zero.

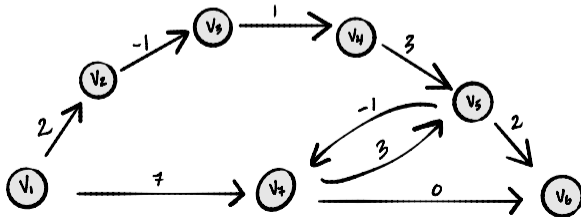
The Bellman-Ford Algorithm

```
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
```

Example

Suppose graph .edges returns edges in following order:

$(v_3, v_4), (v_1, v_2), (v_2, v_3), (v_7, v_6), (v_5, v_7),$
 $(v_7, v_5), (v_4, v_5), (v_5, v_6), (v_1, v_7)$



Time Complexity

```
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: _____ time
- ▶ Each update takes _____ time
- ▶ There are exactly _____ updates
- ▶ Total time complexity: _____

DSC 40B

Theoretical Foundations II

Lecture 14 | Part 3

Early Stopping and Negative Cycles

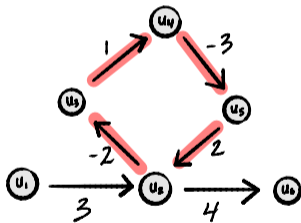
Early Stopping

- ▶ B-F may not need to run for $|V| - 1$ iterations.
- ▶ If no predecessors change, we can break:

```
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

- ▶ A **negative cycle** is a cycle whose total edge weight is negative:



- ▶ If a graph has a negative cycle, (some) shortest paths are **not well defined**.

Detecting Negative Cycles

- ▶ If graph **does not have** negative cycles, estimated distances eventually stop changing (after at most $|V| - 1$ iterations).
- ▶ If graph **has** negative cycles, estimated distances always decrease.
- ▶ To detect them: run a $|V|$ th iteration; if distances change, a negative cycle exists.

Detecting Negative Cycles

```
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
```