

DSC40B: Theoretical Foundations of Data Science II

Lecture 12: *More on BFS: shortest
path in (unweighted) graphs*

Instructor: Yusu Wang

- ▶ Previously:

- ▶ Introduced Breadth-first search (BFS) graph search algorithm
- ▶ Can be used to check for connectivity etc

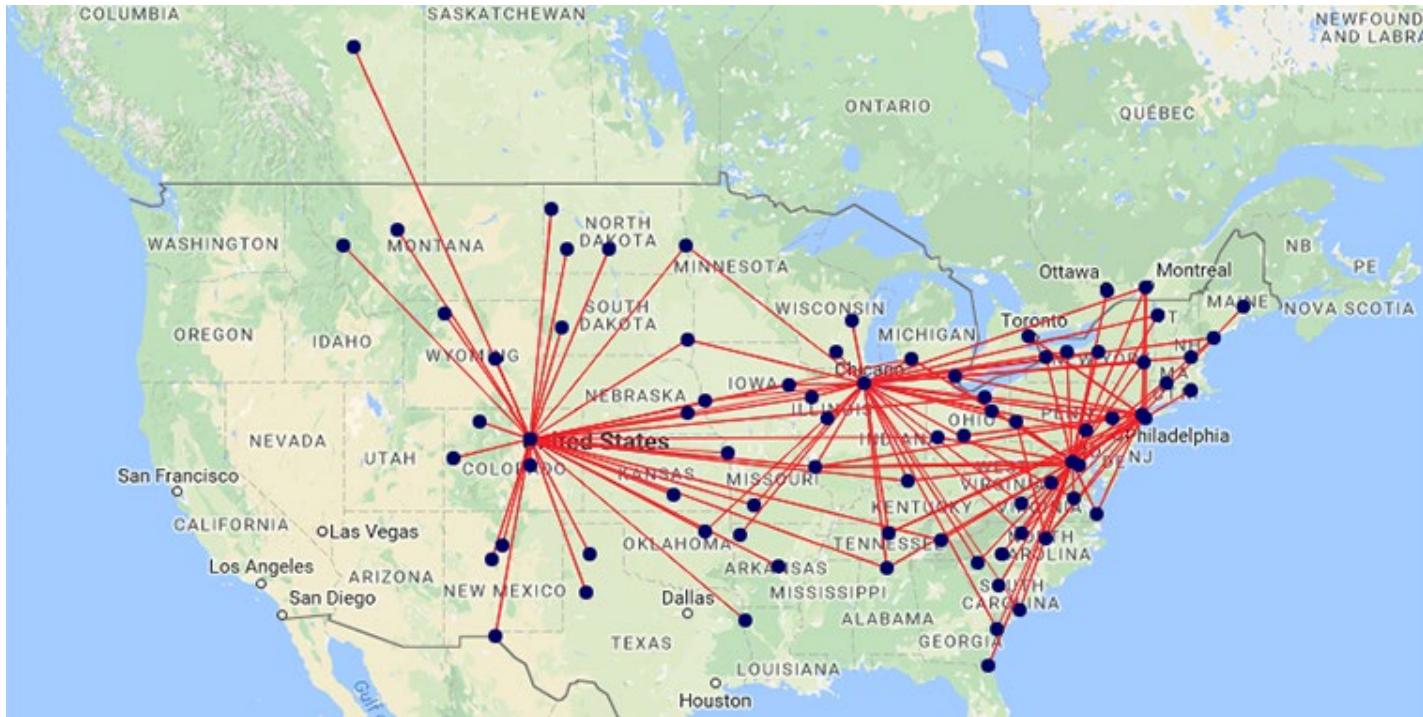
- ▶ Today:

- ▶ Properties of BFS:
 - ▶ Computing shortest path from a source node
 - ▶ BFS tree



Shortest path in (unweighted) graphs



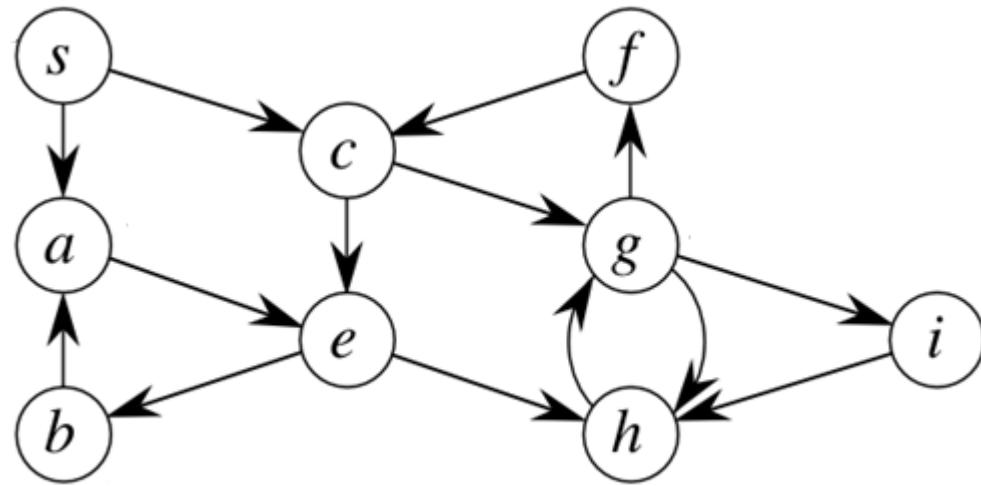


- ▶ How to fly to Denver from Columbus using fewest number of connections

-
- ▶ The **length** of a path is
(#of nodes in this path – 1)
 - ▶ A **shortest path** from u to v
 - ▶ is a path from u to v using with smallest possible length.
 - ▶ Note that there may be multiple shortest paths from u to v , all of which has the same length.
 - ▶ The **shortest path distance** from u to v
 - ▶ is the length of a shortest path from u to v
 - ▶ by convention, the distance is set to be $+\infty$ if there is no path from u to v
-

-
- ▶ **Input:**
 - ▶ A (directed or undirected) graph $G = (V, E)$ and a source node $s \in V$
 - ▶ **Output:**
 - ▶ The shortest path distance from s to all other nodes in V
-

Example



Property of shortest paths (i)

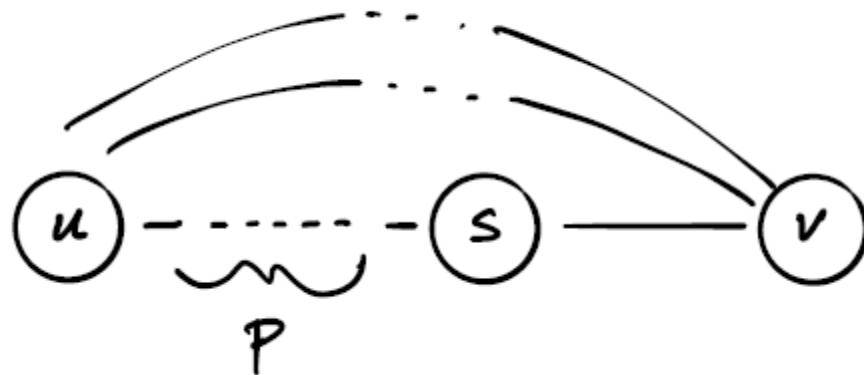
▶ Claim A:

- ▶ Given any $u, v \in V$, if v is reachable from u , a shortest path from u to v has to be **simple**.
- ▶ (Recall, a path is simple if no node in it is visited more than once)



Property of shortest paths (ii)

- ▶ Key structure of shortest paths:
 - ▶ Any sub-path of a shortest path must be a shortest path itself.
 - ▶ This implies that a shortest path of length k consists of a shortest path of length $(k - 1)$ plus one edge.
 - ▶ e.g, given a shortest path $\pi = \langle v_0, v_1, \dots, v_{k-1}, v_k \rangle$ from v_0 to v_k , it consists of a shortest path $\langle v_0, v_1, \dots, v_{k-1} \rangle$ plus edge (v_{k-1}, v_k)



Shortest path and BFS

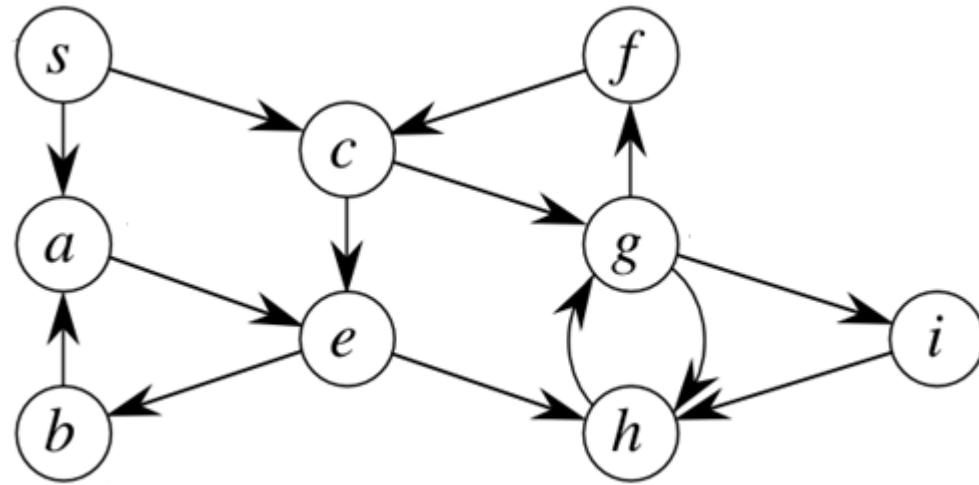


How do we find shortest paths?

- ▶ High level idea:
 - ▶ Starting from the source s
 - ▶ First find all nodes that are at distance 1 from s
 - ▶ Then **use them** to find those nodes at distance 2 from s
 - ▶ Then **use them** to find those nodes at distance 3 from s
 - ▶
 - ▶ Till we find all reachable nodes
- ▶ Note:
 - ▶ to get a node at distance k from source s ,
 - ▶ you have to first reach a node at distance $k - 1$ from s , and extend it from that node via an edge.



Example



-
- ▶ It turns out that this idea is exactly what BFS is doing!
 - ▶ Intuitively,
 - ▶ the first time that we discover a node turns out to encode the fastest way to reach it
 - ▶ that is also when we first change the status of a node from undiscovered to pending
 - ▶ the time this node is discovered relates to the distance to the source nodes
 - ▶ by visiting and exploring the oldest pending nodes (those with smallest distance to the source first), we find fastest way to reach a new node
-

Recall BFS

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```



Property of BFS

- ▶ For any $k \geq 1$,
 - ▶ all nodes distance k from source are added to the queue before any node of distance k
 - ▶ Hence nodes are added to the queue in increasing order of their distances to the source
- ▶ Therefore, nodes are “processed” (popped from the queue) in order of distance from the source,
 - ▶ which further guarantees that the first time to find a undiscovered node, that must be the shortest path to reach this node from the source.



- ▶ Consider a node u that we just popped from the queue
- ▶ Suppose the distance from source s to u is k
- ▶ Our algorithm will then scan through all neighbors of u .
 - ▶ For a neighbor v of u , we have now found a new path π to reach v by the path from s to u , followed by the edge (u, v) . The length is $k + 1$.
 - ▶ If this neighbor v is **undiscovered**
 - ▶ then the new path π must be shortest! Why?
 - ▶ hence the shortest path distance from s to u is $k + 1$
 - ▶ Otherwise, this neighbor v is already discovered (**pending** or **visited**)
 - ▶ then it means that we have already found a path to v before, whose length is at most $k + 1$.
 - ▶ so, the new path π we just found is not useful for shortest path – we already have a shortest path to v



Modified BFS – distance computation

```
def bfs_shortest_paths(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                # append to right
                pending.append(v)
        status[u] = 'visited'

    return distance
```

But we can do more!

- ▶ We can record information to help us recover shortest paths themselves later!
- ▶ Node u is set to be **BFS-predecessor** of v if v is discovered while visiting u .
- ▶ This means that u is the predecessor along a shortest path from the source s to v
 - ▶ In particular, the shortest path from s to u plus edge (u, v) is a shortest path from s to v !
- ▶ If all nodes remember their **BFS-predecessors**,
 - ▶ Then we have enough information to recover shortest paths from the source s to all reachable nodes!



Shortest-Path algorithm

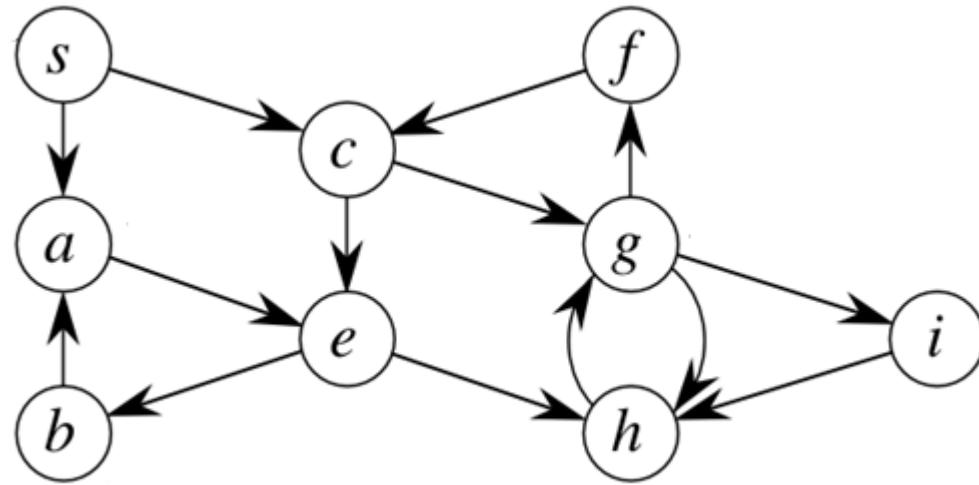
```
def bfs_shortest_paths(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                # append to right
                pending.append(v)
        status[u] = 'visited'

    return predecessor, distance
```

Example



Time complexity

- ▶ Note that this has the same asymptotic time complexity as BFS algorithm
- ▶ $O(|V| + |E|)$



BFS Trees
and
recovering shortest paths



Results of BFS_shortest_path

- ▶ Every node reachable from the source has a single **BFS-predecessor**
 - ▶ except for the source s itself
- ▶ Connecting each node to its BFS-predecessor gives a rooted tree, where the source is the root
 - ▶ in particular, the parent of each node in the tree is its BFS-predecessor
- ▶ This tree is called the **BFS-tree associated to source s**

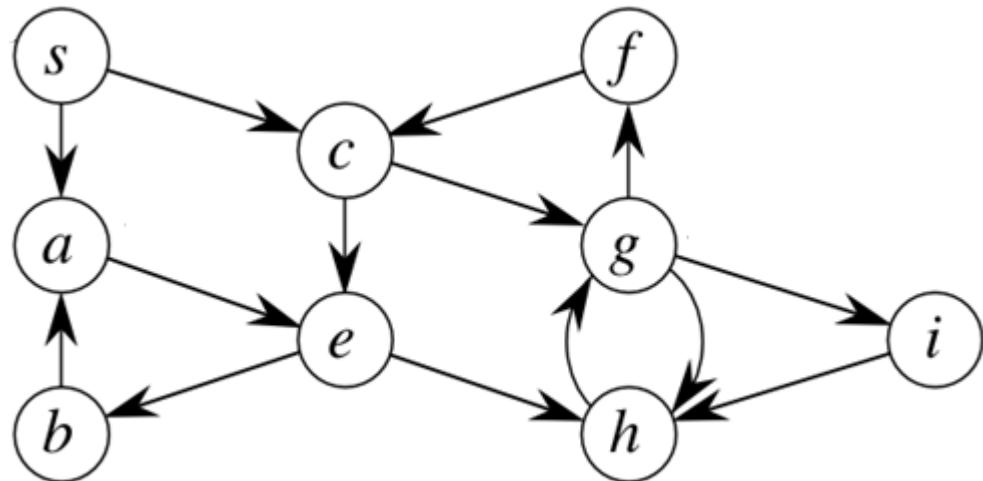


Trees

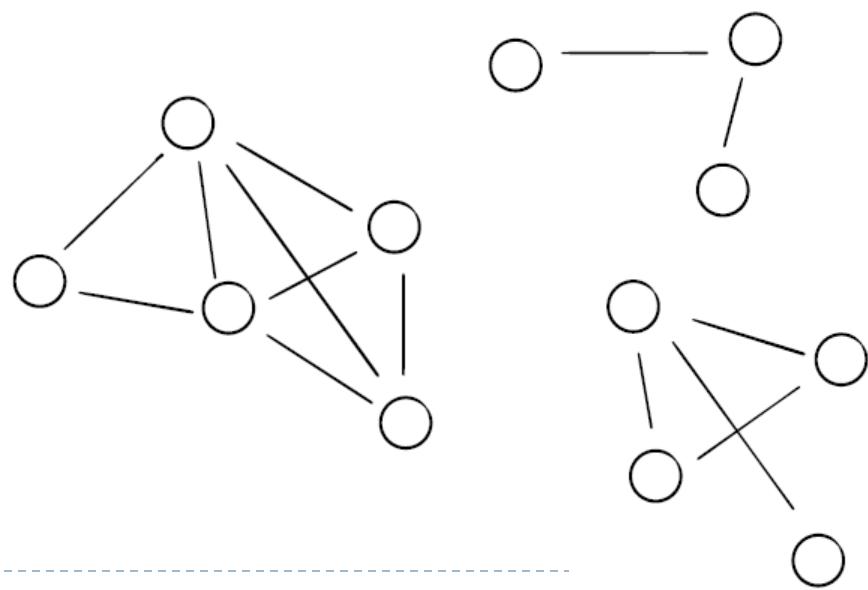
- ▶ A (free) **tree** is a connected graph $T = (V, E)$ where $|E| = |V| - 1$.
- ▶ For any two nodes in a tree, there is only one simple path connecting them.
- ▶ A **root tree** has a root, and each node other than the root has a **parent**.
 - ▶ The parent v is the predecessor of ν along the unique simple path from the root to ν
- ▶ A collection of trees is called a **forest**.



- ▶ BFS-tree associated to source s can be used to recover
 - ▶ both the shortest path and shortest path distance from s to each reachable node.
- ▶ Example:



- ▶ In general, if we run the full-BFS algorithm (with augmentation of computing also predecessors and distances), then we will obtain a collection of BFS-trees, called a BFS-forest.
- ▶ Example:



Summary

- ▶ BFS algorithm:
 - ▶ It explores nodes in order of their first discovery time
 - ▶ In particular, it will explore them in order of their shortest path distance to the source
 - ▶ It will propagate a waveform, first visit all nodes distance 1 to source, then distance 2, then distance 3, and so on
 - ▶ So it explores as broad as possible before moving deeper (meaning further away from the source)
 - ▶ Thus the name: “**breadth-first**” search.
- ▶ It can be used to compute the shortest path distance to a source node
 - ▶ Time complexity is $O(|V| + |E|)$



FIN

