DSC 40B

Theoretical Foundations II

Lecture 12 | Part 1

Review: Aggregate Analysis

Exercise

What is the time complexity of the following code in terms of $|V|$ and $|E|$?

$O(E+V)$

```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print("Edge:", u, v)
```

execs: $|E|$ or $2|E|$

Answer

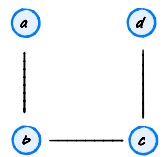
```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print("Edge:", u, v)
```

- Time complexity: $\Theta(V + E)$
- The `print` executes:
  - **once** for each edge, if the graph is **directed**.
  - **twice** for each edge, if the graph is **undirected**.

Another Look...

$\Theta(V+E)$

- Consider the graph's `dict`-of-`sets` representation.




```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print("Edge:", u, v)
```

```
{
    "a": {"b"},
    "b": {"a", "c"},
    "c": {"b", "d"},
    "d": {"c"}
}
```

Time Complexity

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)


def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        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'
```

$\Theta(V)$

#execs:
$|E|$ or $2|E|$ (if undir)
in aggregate

time:
$\Theta(E)$
in aggregate

$\Theta(V+E)$

Exercise

Suppose `bfs` is called on an undirected graph using source node `u`.

If `u` is part of a connected component with nodes V_1 and edges E_1, what is the time complexity of the call to `bfs`?

Answer

- Time complexity: $\Theta(V_1 + E_1)$
- `bfs` explores all nodes and edges in the connected component.

Time Complexity of Full BFS

- `full_bfs` calls `bfs` **once** for each connected component.

- Time complexity:

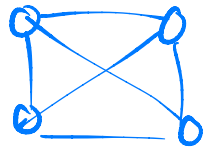
$$\begin{aligned}
&\Theta((V_1 + E_1) + (V_2 + E_2) + \ldots + (V_k + E_k)) \\
&\quad = \Theta((V_1 + V_2 + \ldots + V_k) + (E_1 + E_2 + \ldots + E_k)) \\
&\quad = \Theta(V + E)
\end{aligned}$$

Time Complexity

- Full BFS takes $\Theta(V + E)$

Time Complexity

- Full BFS takes $\Theta(V + E)$
- Why not just $\Theta(E)$?
- $\Theta(V + E)$ works *for all graphs*.
  - If we know more about the number of edges, we might be able to simplify.
  - E.g., if the graph is **complete**, $E = \Theta(V^2)$, so time complexity is $\Theta(V + V^2) = \Theta(V^2)$.



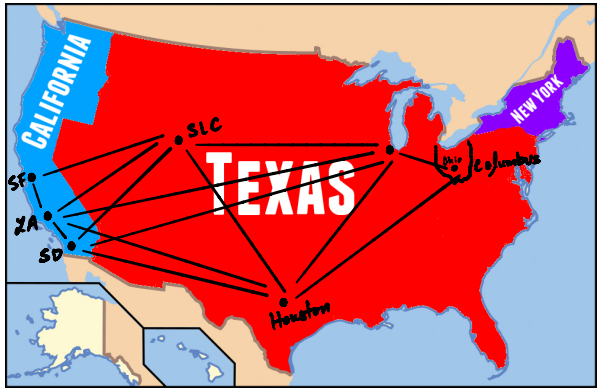
$$|E| = \frac{|V|(|V|-1)}{2} = \Theta(V^2)$$

DSC 40B

Theoretical Foundations II

Lecture 12 | Part 2

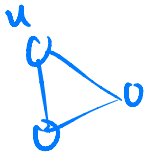
Shortest Paths


CALIFORNIA
NEW YORK
TEXAS
SLC
SF
LA
SD
Houston
Ohio
Columbus

Recall

- The **length** of a path is

  (# of nodes) – 1



Definitions

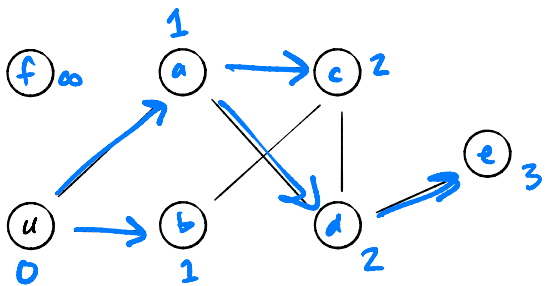
- A **shortest path** between u and v is a path between u and v with smallest possible length.
  - There may be several, or none at all.

- The **shortest path distance** is the length of a shortest path.
  - Convention: ∞ if no path exists.
  - "the distance between u and v" means spd.

Today: Shortest Paths

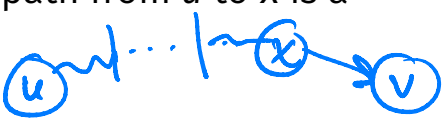
- **Given**: directed/undirected graph G, source u
- **Goal**: find shortest path from u to every other node

Example



Key Property of Shortest Paths

- Suppose you have shortest path from u to v.
- Suppose it goes through the edge (x, v).
  - x is a neighbor of v.
- Then the part of that path from u to x is a shortest path.



Key Property, Restated

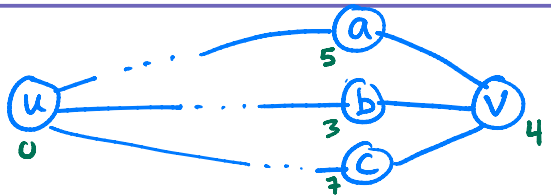
- A shortest path of length k is composed of:
  - A **shortest path** of length $k - 1$.
  - Plus one edge.

Exercise

Node v has three neighbors: a, b, and c. The distance from:

- u to a is 5.
- u to b is 3.
- u to c is 7.

What is the distance from u to v? 4



Algorithm Idea

- Find all nodes distance 1 from source.
- Use these to find all nodes distance 2 from source.
- Use these to find all nodes distance 3 from source.
- ...

It turns out...

...this is exactly what BFS does.

DSC 40B

Theoretical Foundations II

Lecture 12 | Part 3

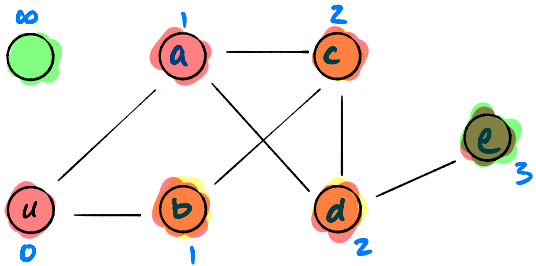
BFS for Shortest Paths

Key Property of BFS

- For any $k \geq 1$ you choose:
- All nodes distance $k - 1$ from source are added to the queue before any node of distance k.
- Therefore, nodes are “processed” (popped from queue) in order of distance from source.

[u a b c d e]
0 1 1 2 2 3

Example



Discovering Shortest Paths

- We “discover” shortest paths when we pop a node from queue and look at its neighbors.
- But the neighbor’s status matters!

Consider This

- We pop a node *s*.
- It has a neighbor *v* whose status is **undiscovered**.
- We've discovered a **shortest path** to *v* through *s*!

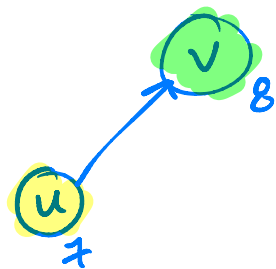
Consider This

- We pop a node *s*.
- It has a neighbor *v* whose status is **pending** or **visited**.
- We already have a shortest path to *v*.

Modifying BFS

- Use BFS “framework”.
- Return dictionary of **search predecessors**.
  - If v is discovered while visiting u, we say that u is the **BFS predecessor** of v.
  - This encodes the shortest paths.
- Also return dictionary of shortest path distances.

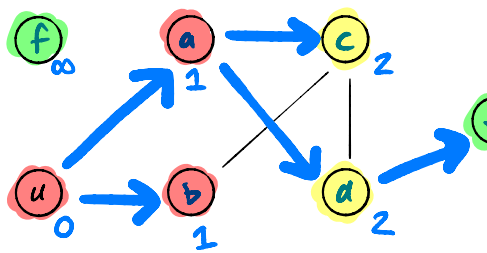
```python
def bfs_shortest_paths(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    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
```



distance = {
'a': 1,
'b': 1
'c': 2
'd': 2
'e': 3
'f': ∞,
'u': 0
}

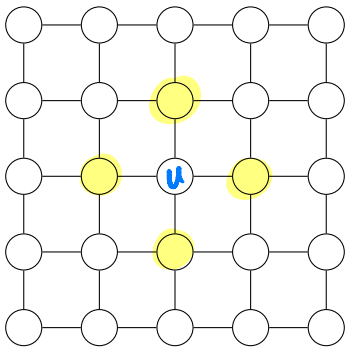
pred = {
'a': 'u',
'b': 'u'
'c': 'a'
'd': 'a'
'e': 'd'
'f': None
'u': None
}

Example



[u, a, b, c, d

Example



DSC 40B

Theoretical Foundations II

Lecture 12 | Part 4

BFS Trees

Result of BFS

- Each node reachable from source has a single BFS predecessor.
  - Except for the source itself.
- The result is a **tree** (or forest).

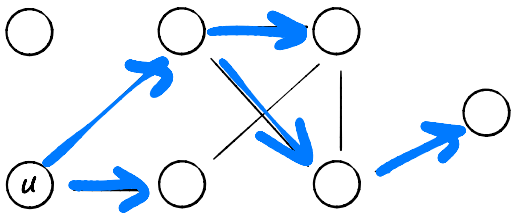
Trees

- A (free) **tree** is an undirected graph $T = (V, E)$ such that T is connected and $|E| = |V| - 1$.
- A **forest** is graph in which each connected component is a tree.

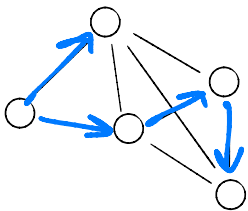
BFS Trees (Forests)

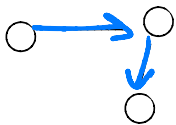
- If the input is connected, BFS produces a **tree**.
- If the input is not connected, BFS produces a **forest**.

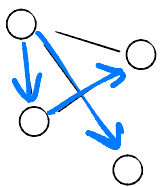
Example



Example







BFS Trees

- BFS trees and forests encode shortest path distances.