DSC 40B

Theoretical Foundations II

Lecture 12 | Part 1

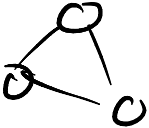
Warmup: Aggregate Analysis

$|V|$ $|E|$

Time Complexity

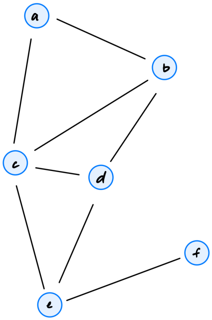
```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)


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'
```

print(u)



Exercise

What is printed if we run a BFS starting at a?




```python
...
while pending:
    u = pending.popleft()
    print(f'Popped {u}')
    for v in graph.neighbors(u):
        print(f'Exploring edge ({u}, {v})')
        # explore edge (u,v)
        ...
```

Answer

```
Popping a
Exploring edge (a, b)
Exploring edge (a, c)
Popping b
Exploring edge (b, a)
Exploring edge (b, c)
Exploring edge (b, d)
Popping c
Exploring edge (c, a)
Exploring edge (c, b)
Exploring edge (c, d)
Exploring edge (c, e)
```

```
Popping d
Exploring edge (d, b)
Exploring edge (d, c)
Exploring edge (d, e)
Popping e
Exploring edge (e, c)
Exploring edge (e, d)
Exploring edge (e, f)
Popping f
Exploring edge (f, e)
```

Aggregate Analysis

- During any one call to `bfs`:
  - Number of printed nodes: ?
  - Number of printed edges: ?

- In **aggregate** (over all calls):
  - Number of printed nodes: *exactly* $|V|$
  - Number of printed edges: *exactly* $2|E|$

Time Complexity

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

Time Complexity

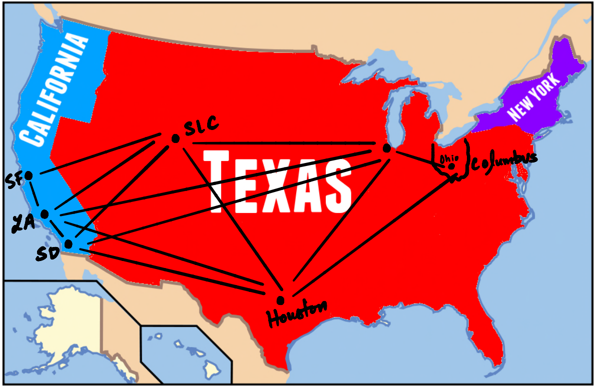
- Full BFS takes $\Theta(V + E)$
- Why not just $\Theta(E)$?
- $\Theta(V + E)$ works *for all graphs.*
  - If we know more about the number of edges, we might be able to simplify.
  - E.g., if the graph is **complete**, $E = \Theta(V^2)$, so time complexity is $\Theta(V + V^2) = \Theta(V^2)$.

DSC 40B

Theoretical Foundations II

Lecture 12 | Part 2

Shortest Paths



CALIFORNIA
NEW YORK
TEXAS
SLC
SF
LA
SD
Ohio
Columbus
Houston

Recall

- The **length** of a path is

  (# of nodes) – 1

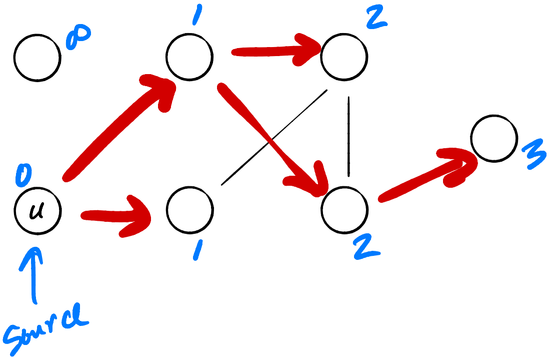
Definitions

- 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

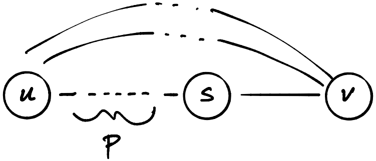
- **Given**: directed/undirected graph G, source u
- **Goal**: find shortest path from u to every other node

Example



Key Property

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



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

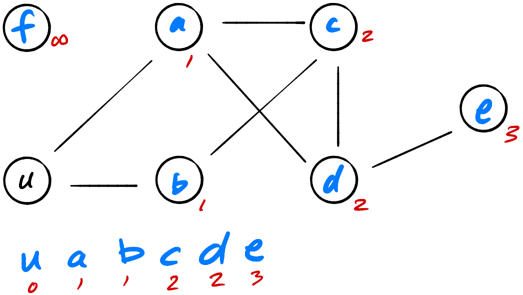
Lecture 12 | Part 3

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.

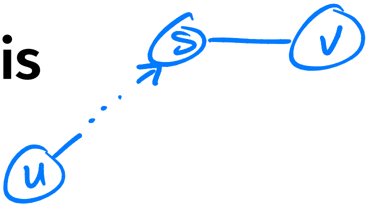
Example



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.

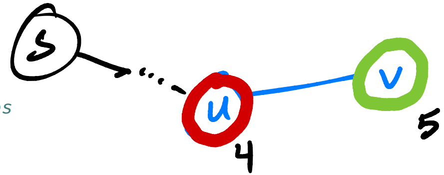
{
'a': 'b',
'b': 'd',

Modifying BFS

$u \rightarrow v$

- 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
```

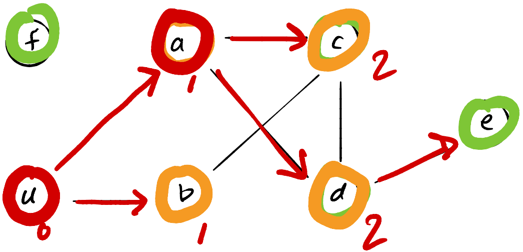


Example

pred = {
'u': None,
'f': None,
'a': 'u',
'c': 'a',
⋮
}

[b, c, d

dist = {
'u': 0,
'a': 1,
'b': 1,
⋮
}



e ← d ← a ← u

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).

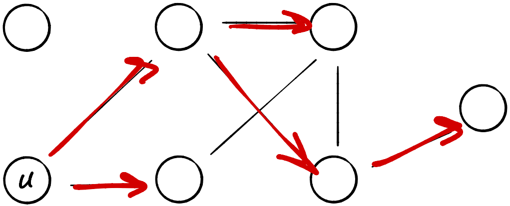
Trees

- 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.

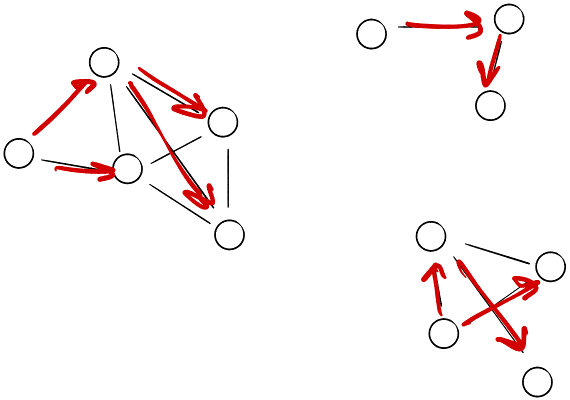
BFS Trees (Forests)

- 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.

BFS Trees

- BFS trees and forests encode shortest path distances.