

DSC 40B

Theoretical Foundations II

Graph Search Strategies

How do we:

- ▶ determine if there is a path between two nodes?
- ▶ check if graph is connected?
- ▶ count connected components?

Search Strategies

- ▶ A **search strategy** is a procedure for exploring a graph.
- ▶ Different strategies have different properties.

Node Statuses

At any point during a search, a node is in exactly one of three states:

- ▶ **visited**
- ▶ **pending** (discovered, but not yet visited)
- ▶ **undiscovered**

Rules

- ▶ At every step, next visited node chosen from among **pending** nodes.
- ▶ When a node is marked as **visited**, all of its neighbors have been marked as **pending**.

Choosing the next Node

How to choose among pending nodes?

- ▶ Visit **newest** pending (**depth-first** search).
- ▶ Visit **oldest** pending (**breadth-first** search).

Breadth-First Search

At every step:

1. Visit oldest pending node.
 2. Mark its undiscovered neighbors as pending.
- To store pending nodes, use a FIFO **queue**.

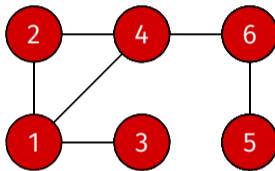
Queues in Python

- ▶ Want $\Theta(1)$ time pops/appends on either side.
- ▶ `from collections import deque` (“deck”).
 - ▶ `.popleft()` and `.pop()`
 - ▶ `list` doesn't have right time complexity!
 - ▶ `import queue` isn't what you want!
- ▶ Keep track of node status attribute using dictionary.

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'
```

Example



pending = [~~1~~, 3, 4]

Before iterating. After 1st iteration. After 2nd iteration. After 3rd iteration. After 4th iteration. After 5th iteration. After 6th iteration.

Note

- ▶ BFS works just as well for directed graphs.

Claim

- ▶ bfs with source u will visit all nodes reachable from u (and only those nodes).
- ▶ Useful!
 - ▶ Is there a path between u and v ?
 - ▶ Is graph connected?

DSC 40B

Theoretical Foundations II

Analysis of BFS

Exploring with BFS

- ▶ BFS will visit all nodes reachable from source.
- ▶ If **disconnected**, BFS will not visit all nodes.
- ▶ We can do so with a **full BFS**.
 - ▶ Idea: “re-start” BFS on undiscovered node.
 - ▶ Must pass statuses between calls.

Making Full BFS

Modify bfs to accept statuses:

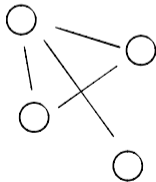
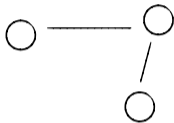
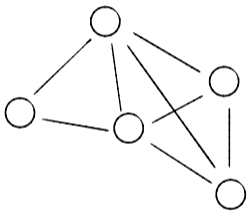
```
def bfs(graph, source, status=None):  
    """Start a BFS at `source`."""  
    if status is None:  
        status = {node: 'undiscovered' for node in graph.nodes}  
    # ...
```

Making Full BFS

Call bfs multiple times:

```
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)
```

Example



Observation

- If there are k connected components, bfs in line 5 is called exactly k times.

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

Key Properties of full_bfs

- ▶ Each node added to queue **exactly once**.
- ▶ Each edge is explored **exactly**:
 - ▶ **once** if graph is **directed**.
 - ▶ **twice** if graph is **undirected**.

Time Complexity of full_bfs

- ▶ Analyzing full_bfs is easier than analyzing bfs.
 - ▶ full_bfs visits all nodes, no matter the graph.
- ▶ Result will be **upper bound** on time complexity of bfs.
- ▶ We'll use a **aggregate analysis**.

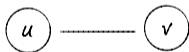
BFS

```
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'
```



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)

    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'
```

Time Complexity of Full BFS

- ▶ $\Theta(V + E)$
- ▶ If $|V| > |E|$: $\Theta(V)$
- ▶ If $|V| < |E|$: $\Theta(E)$
- ▶ Namely, if graph is **complete**: $\Theta(V^2)$.
- ▶ Namely, if graph is **very sparse**: $\Theta(V)$.

Next Time

- ▶ Finding **shortest paths** using BFS.