DSC 40B

Theoretical Foundations II

Lecture 11 | Part 1

Adjacency Matrices (Recap)

Representations

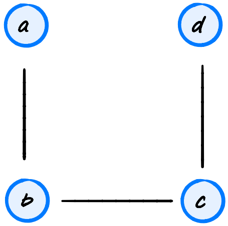
- How do we **store** a graph in a computer's memory?

- Three approaches:
  1. Adjacency matrices.
  2. Adjacency lists.
  3. "Dictionary of sets"

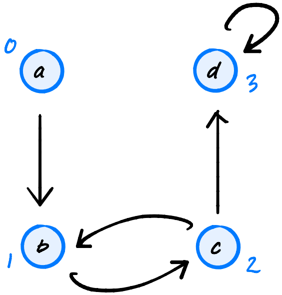
Adjacency Matrices

- Assume nodes are numbered 0, 1, ..., $|V| - 1$
- Allocate a $|V| \times |V|$ (Numpy) array
- Fill array as follows:
  - `arr[i,j] = 1` if $(i,j) \in E$
  - `arr[i,j] = 0` if $(i,j) \notin E$

Example



Example



$$
\begin{array}{c c}
 & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{array}
$$

Observations

- If G is undirected, matrix is symmetric.
- If G is directed, matrix may not be symmetric.

Time Complexity

operation	code	time
edge query	`adj[i,j] == 1`	$\Theta(1)$
degree(i)	`np.sum(adj[i,:])`	$\Theta(\lvert V\rvert)$

Space Requirements

- Uses $|V|^2$ bits, even if there are very few edges.
- But most real-world graphs are **sparse**.
  - They contain many fewer edges than possible.

Example: Facebook

- Facebook has 2 billion users.

$$
\begin{aligned}
(2 \times 10^9)^2 &= 4 \times 10^{18} \text{ bits} \\
&= 500 \text{ petabits} \\
&\approx 6500 \text{ years of video at 1080p} \\
&\approx 60 \text{ copies of the internet as it was in 2000}
\end{aligned}
$$

Adjacency Matrices and Math

- Adjacency matrices are useful mathematically.
- Example: (i, j) entry of A^2 gives number of hops of length 2 between i and j.

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 2

Adjacency Lists

What's Wrong with Adjacency Matrices?

- Requires $\Theta(|V|^2)$ storage.
- Even if the graph has no edges.
- **Idea**: only store the edges that exist.

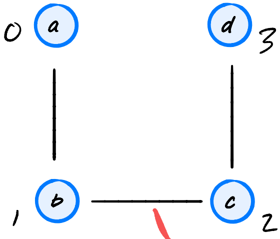
```
[
  [],
  [],
  []
]
```

Adjacency Lists

- Create a list `adj` containing $|V|$ lists.
- `adj[i]` is list containing the neighbors of node i.

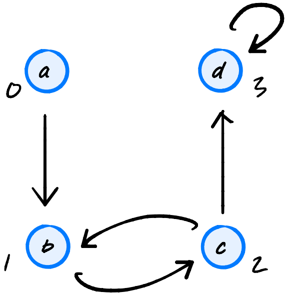
adj[0]

Example



[
0 [1],
1 [0,2],
2 [1,3],
3 [2]
]

Example



[
  [1],
  [2]
  [1,3],
  [3]
]

Observations

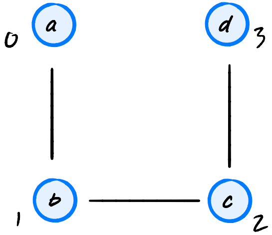
- If G is undirected, each edge appears twice.
- If G is directed, each edge appears once.

Time Complexity

operation	code	time
edge query	`j in adj[i]`	$\Theta(\text{degree}(i))$
degree(i)	`len(adj[i])`	$\Theta(1)$

Exercise

What does the following code do?




```
adj = [
        [1],
        [0, 2],
        [1, 3],
        [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

(0, 1)
(1, 0)
(1, 2)
⋮

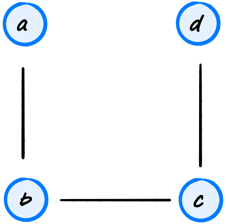
$G = (V, E)$

Exercise

How many times is the `print` statement executed in terms of $|V|$ and $|E|$?

$2|E|$ if undirected

$|E|$ if directed

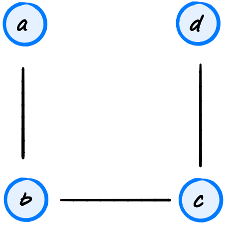



```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

Exercise

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




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

$\Theta(E+V)$

Exercise

What is the time complexity of the following code?

a d

b c

```
adj = [
    [],
    [],
    [],
    []
]
```

|V| iterations

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

Looping Over Edges

- Looping over all edges in this way takes $\Theta(V + E)$ time.

- In aggregate, the `print` statement is executed:
  - $2|E|$ times if graph is undirected.
  - $|E|$ times if graph is directed.

- This is called an **aggregate analysis**.

Space Requirements

- Need $\Theta(|V|)$ space for outer list.
- Plus $\Theta(|E|)$ space for inner lists.
- In total: $\Theta(|V| + |E|)$ space.

$\Theta(V+E)$

Example: Facebook

- Facebook has 2 billion users, 400 billion friendships.

- If each edge requires 32 bits:

$$
\begin{aligned}
&(2 \text{ bits} \times 200 \times (2 \text{ billion}) \\
&= 64 \times 400 \times 10^9 \text{ bits} \\
&= 3.2 \text{ terabytes} \\
&= 0.04 \text{ years of HD video}
\end{aligned}
$$

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 3

Dictionary of Sets

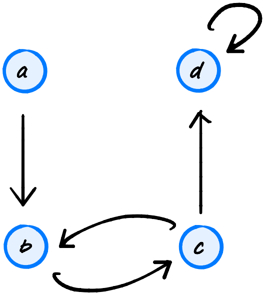
Tradeoffs

- Adjacency matrix: fast edge query, lots of space.
- Adjacency list: slower edge query, space efficient.
- Can we have the best of both?

Idea

- Use **hash tables**.
- Replace inner edge lists by `set`s.
- Replace outer list with `dict`.
  - Doesn't speed things up, but allows nodes to have arbitrary labels.

Example




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

Time Complexity

operation	code	time
edge query	`j in adj[i]`	$\Theta(1)$ average
degree(i)	`len(adj[i])`	$\Theta(1)$ average

Space Requirements

- Requires only $\Theta(V + E)$.
- But there is overhead to using hash tables.

Dict-of-sets implementation

- Install with `pip install dsc40graph`
- Import with `import dsc40graph`

networkx

- Docs: `https://eldridgejm.github.io/dsc40graph/`
- Source code: `https://github.com/eldridgejm/dsc40graph`
- Will be used in HW coding problems.

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 4

Graph Search Strategies

How do we:

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

Search Stategies

- A **search strategy** is a procedure for exploring a graph.
- Different strategies are useful in different situations.

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?

- Idea 1: Visit **newest** pending (**depth-first search**).
- Idea 2: Visit **oldest** pending (**breadth-first search**).

Main Idea

Depth-first search (DFS) and breadth-first search (BFS) each discover different properties of the graph.

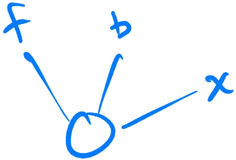
For example, we'll see that BFS is useful for finding shortest paths (DFS in general is not).

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 5

Breadth-First Search



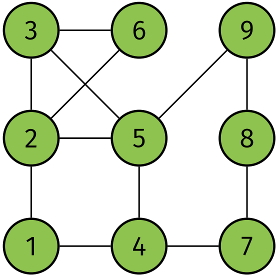
Breadth-First Search

- At every step:
  1. Visit **oldest** **pending** node.
  2. Mark its **undiscovered** neighbors as **pending**.

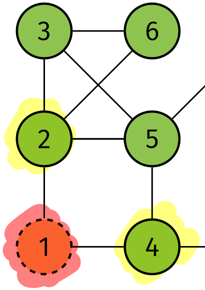
- Convention: in this class, neighbors produced in sorted order.[1]

[1] In general, the order in which a node's neighbors produced is arbitrary.

Example



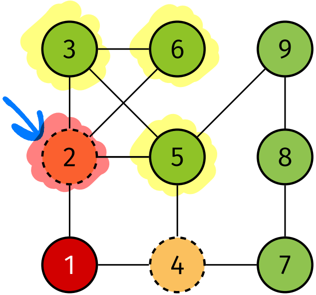
Example



pending = [~~1~~], 2, 4]

Before iterating.

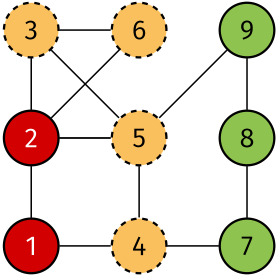
Example



pending = [~~2~~,4,3,5,6]

After 1st iteration.

Example

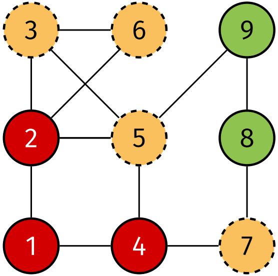


`pending = [4,3,5,6]`

After 2nd iteration.

Exercise: what will the picture look like after each of the next two iterations?

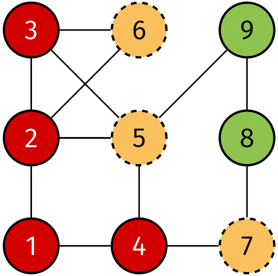
Example




```
pending = [3,5,6,7]
```

After 3rd iteration.

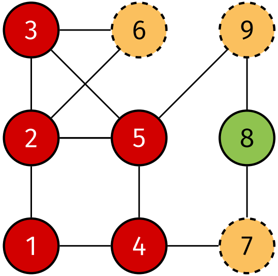
Example



`pending = [5,6,7]`

After 4th iteration.

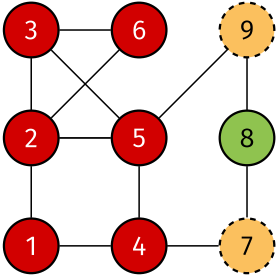
Example




```
pending = [6,7,9]
```

After 5th iteration.

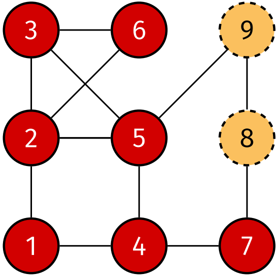
Example



`pending = [7,9]`

After 6th iteration.

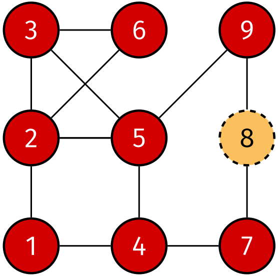
Example



pending = [9,8]

After 7th iteration.

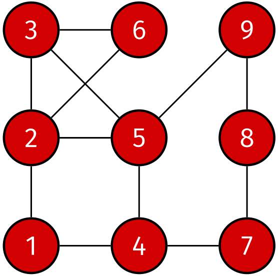
Example



pending = [8]

After 8th iteration.

Example



`pending = []`

After 9th iteration.

[1, 2, 3, 4]

Implementation

- To store pending nodes, use a FIFO **queue**.
  - FIFO = “First in, first out”.

- While queue is not empty:
  - Pop a node, u.
  - Add **undiscovered** neighbors to queue.

Queues in Python

- Want Θ(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.

Exercise

```
for node in graph.nodes:
    status[node] = 'undiscovered'
```

```python
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:
        # EXERCISE: fill this in...
        u = pending.popleft()
        for v in graph.neighbors(u):
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                pending.append(v)
        status[u] = 'visited'
```

BFS

```python
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'
```

Note

- What does this code actually *return*?

Note

- What does this code actually *return*?
- Nothing, yet. It is a *foundation*.

Note

- BFS works just as well for directed graphs.

DSC 40B

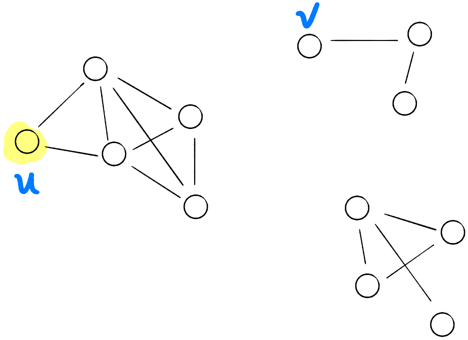
Theoretical Foundations II

Lecture 11 | Part 6

Analysis of BFS

Exercise

What will `bfs` do when run on a disconnected graph?



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?

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:

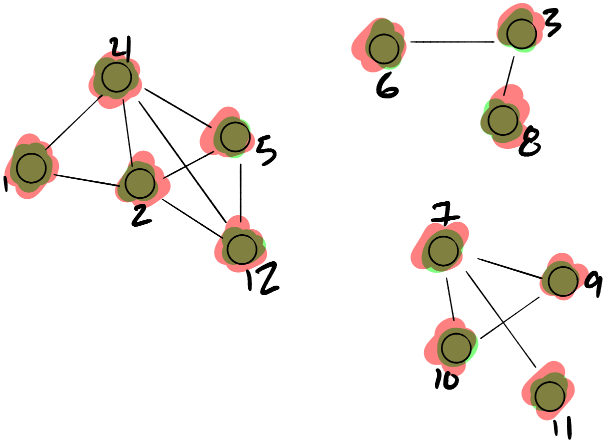
```
def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}
    # ...
```

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



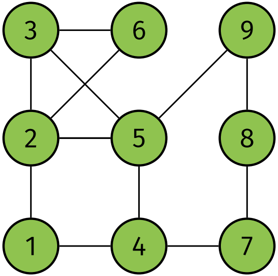
Observation

- 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)
```

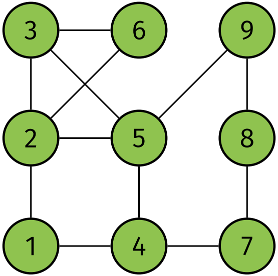
Exercise

How many times is each node added to the queue in a BFS of the graph below? Once.



Exercise

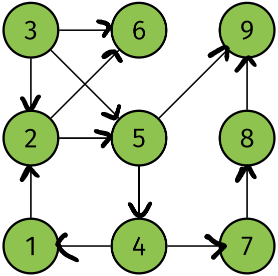
How many times is each edge "explored" in a BFS of the graph below? Twice



(9,8)
(8,9)

Exercise

How many times is each edge "explored" in a BFS of the *directed* graph below? Once



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 an **aggregate analysis**.

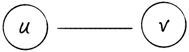
BFS

```python
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'
```

$\Theta(V)$

print

$\Theta(E)$



$\Theta(V+E)$

Time Complexity

```python
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'
```

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**: $\Theta(V)$.

Notational Note

- We’ll often write $\Theta(V + E)$ instead of $\Theta(|V| + |E|)$.
- You can use whichever.

Next Time

- Finding **shortest paths** using BFS.