CSE 331:
Algorithms & Complexity
“More BFS and DFS”

Prof. Charlie Anne Carlson (She/Her)

Lecture 12

Wednesday September 24th, 2025

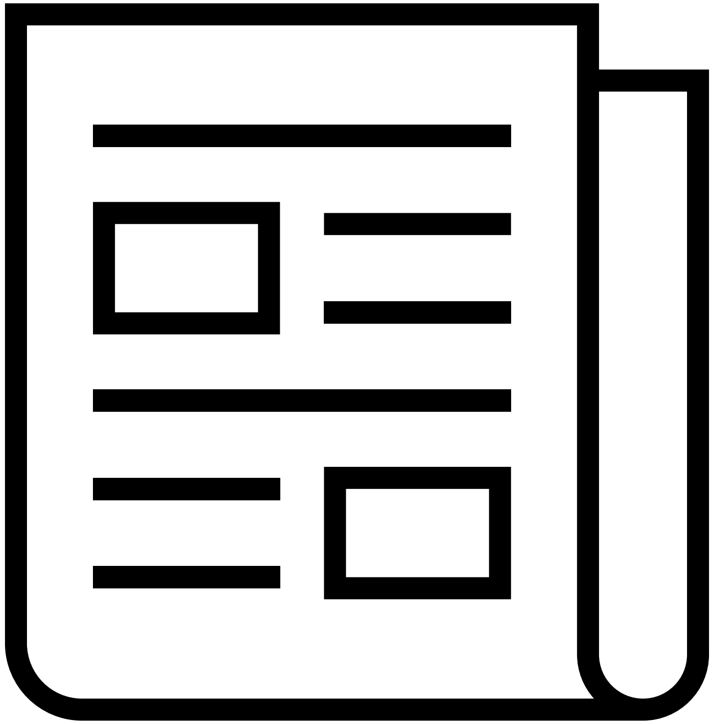
University at Buffalo

Schedule

1. Course Updates
2. Graph Traversal
    1. BFS
    2. DFS
    3. Stacks & Queues
3. Coloring

Course Updates

- HW 1 Grading Out
- HW 2 Solutions Out Soon
- HW 3 Out
- Group Project
  - Team Emails Soon
  - No Autolab Registration
- First Quiz NEXT Monday!
- Sample Midterms



Q: How do we show this solves Connectivity?

- Input: $G = (V, E)$ and $s \in V$
- Output: CC(s)
- Let $\mathrm{R} = \{s\}$
- While there exists $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$ such that $\mathrm{u} \in \mathrm{R}$ and $v \notin R$:
  - Add v to R
- Return R

Q: How do we show this solves Connectivity?

- Input: $G = (V, E)$ and $s \in V$
- Output: CC(s)
- Let $\mathrm{R} = \{\mathrm{s}\}$
- While there exists $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$ such that $\mathrm{u} \in \mathrm{R}$ and $v \notin R$:
  - Add v to R
- Return R

- **Argue that R = CC(s)!**
  - **Show $\boldsymbol{R \subseteq CC(s)}$**
  - **Show $\boldsymbol{CC(s) \subseteq R}$**

Breadth First Search (Properties)

Claim: $\mathrm{R} \subseteq \mathrm{CC}(s)$

Proof Idea:

- This wants us to show that everything reached by Explore is in the connected component of s.
- Let's do induction on iteration of the algorithm.
  - Do you believe the first iteration.
  - Given any iteration is true, how do you feel about the next iteration?

Q: Does this always terminate?

- Input: $G = (V, E)$ and $s \in V$
- Output: CC(s)
- Let $\mathrm{R} = \{s\}$
- While there exists $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$ such that $\mathrm{u} \in \mathrm{R}$ and $v \notin R$:
  - Add v to R
- Return R

Q: Does this always terminate?

- Input: $G = (V, E)$ and $s \in V$
- Output: CC(s)
- Let $\mathrm{R} = \{s\}$
- While there exists $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$ such that $\mathrm{u} \in \mathrm{R}$ and $v \notin R$:
  - Add v to R
- Return R

- Yes, in each round we either add a vertex, or we exit. There are only |V| vertices and show provided the input is finite, the algorithm must terminate.

Explore Proofs

Claim: $\mathrm{CC}(s) \subseteq \mathrm{R}$

Proof:

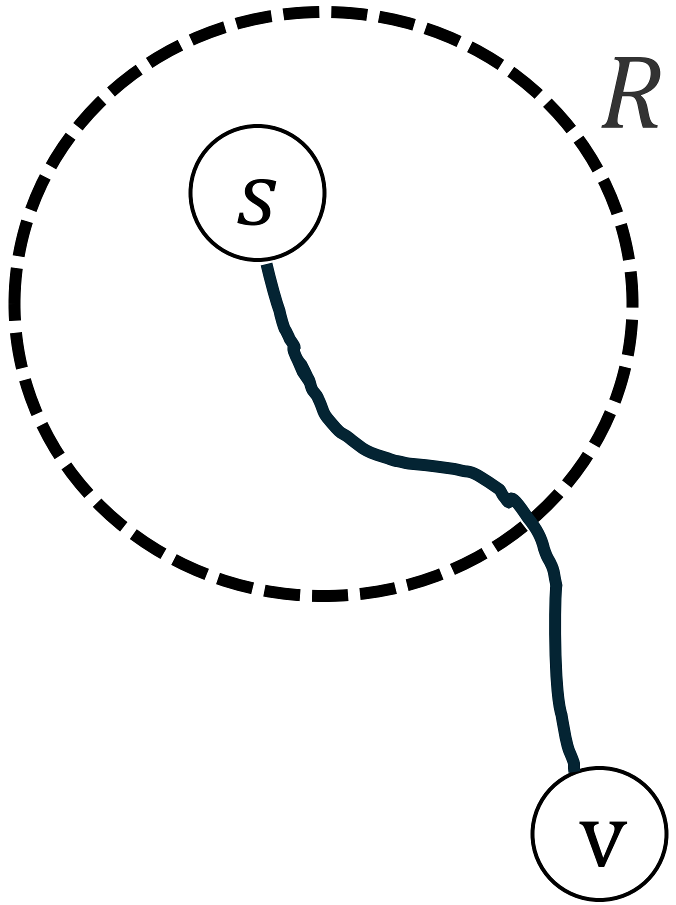
- This says that every vertex in the connected component is added to R by Explore.
- That is, for every vertex v such that there is a path from s to v, v is added to R by Explore.

Explore Proofs

Claim: $\mathrm{CC}(s) \subseteq \mathrm{R}$

Proof:

- Suppose to the contrary that there exists $v \in \mathrm{CC}(s)$ such that $v \notin \mathrm{R}$.
  - Then there must exist a path that starts at s (inside R) and ends at v (outside R).

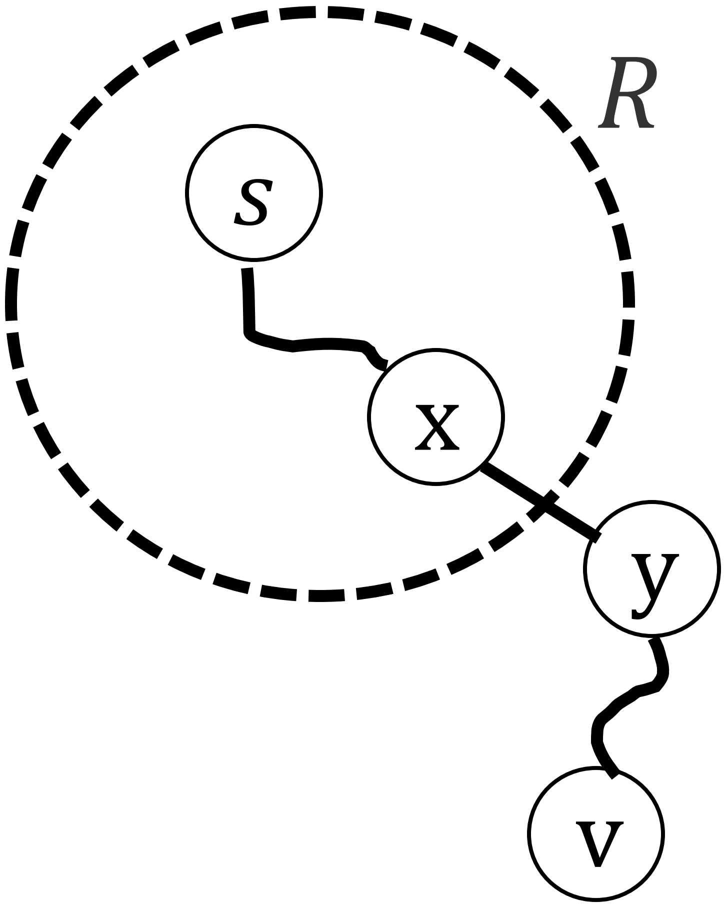


Explore Proofs

Claim: $\mathrm{CC(s)} \subseteq \mathrm{R}$

Proof:

- There then must exist $\{\mathrm{x}, \mathrm{y}\} \in \mathrm{E}$ such that $\mathrm{x} \in \mathrm{R}$ and $\mathrm{y} \notin \mathrm{R}$.



Q: What is wrong with such an $\{x, y\}$ existing?

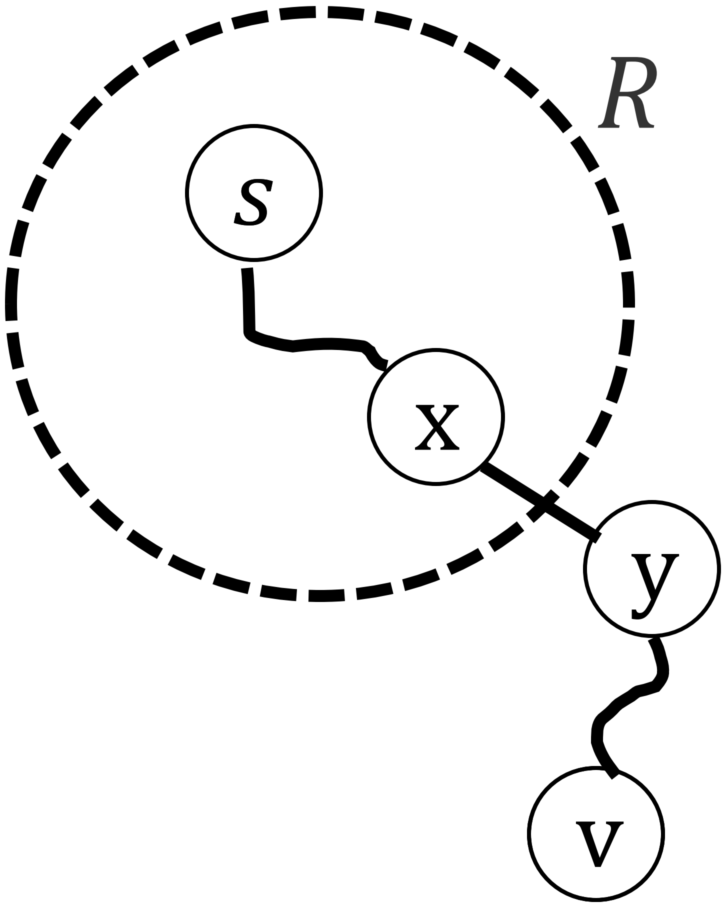
- Input: $G = (V, E)$ and $s \in V$
- Output: CC(s)
- Let $R = \{s\}$
- While there exists $\{u, v\} \in E$ such that $u \in R$ and $v \notin R$:
  - Add v to R
- Return R

Explore Proofs

Claim: $\mathrm{CC}(s) \subseteq \mathrm{R}$

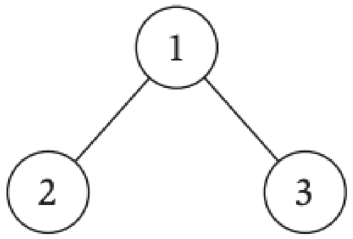
Proof:

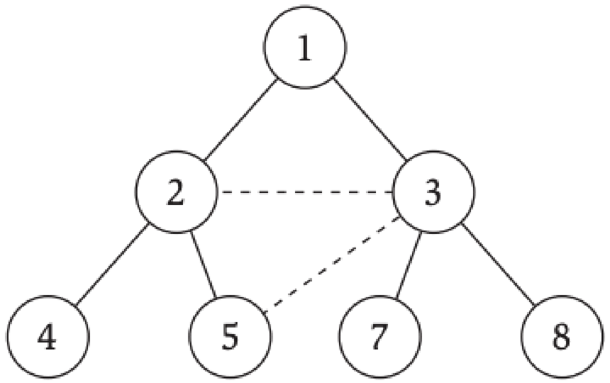
- There then must exist $\{x, y\} \in E$ such that $x \in R$ and $y \notin R$.
- If this the case, then the algorithm wouldn't have terminated and would have instead added y. =><=

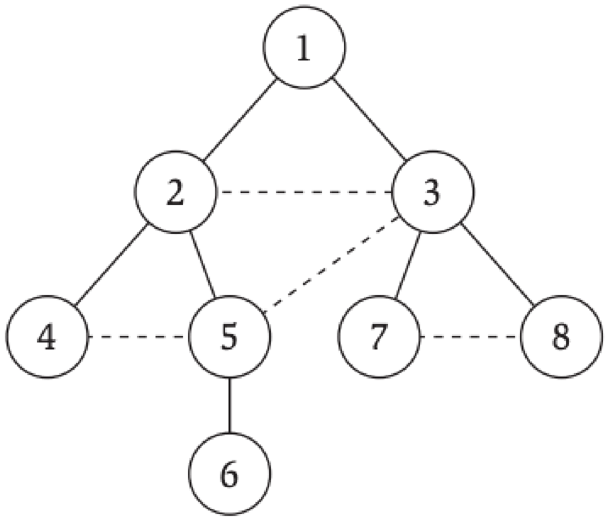


Q: How would you describe BFS?

- $L_0 = s$
- $L_1 =$ neighbors of L_0.
- $L_2 =$ neighbors of L_1 that are not in L_0.
- $L_\mathrm{i} =$ neighbors of $L_{\mathrm{i}-1}$ that are not in previous layer.







Depth First Search

- **Input**: The current vertex $\mathrm{u} \in V$
- **Global**: An array of exploration $A \in \{0,1\}^V$
- Mark current vertex as explored ($\mathrm{A}[\mathrm{u}] = 1$).
- For each $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$:
  - If v is not explored ($\mathrm{A}[\mathrm{v}] == 0$):
    - DFS(v)

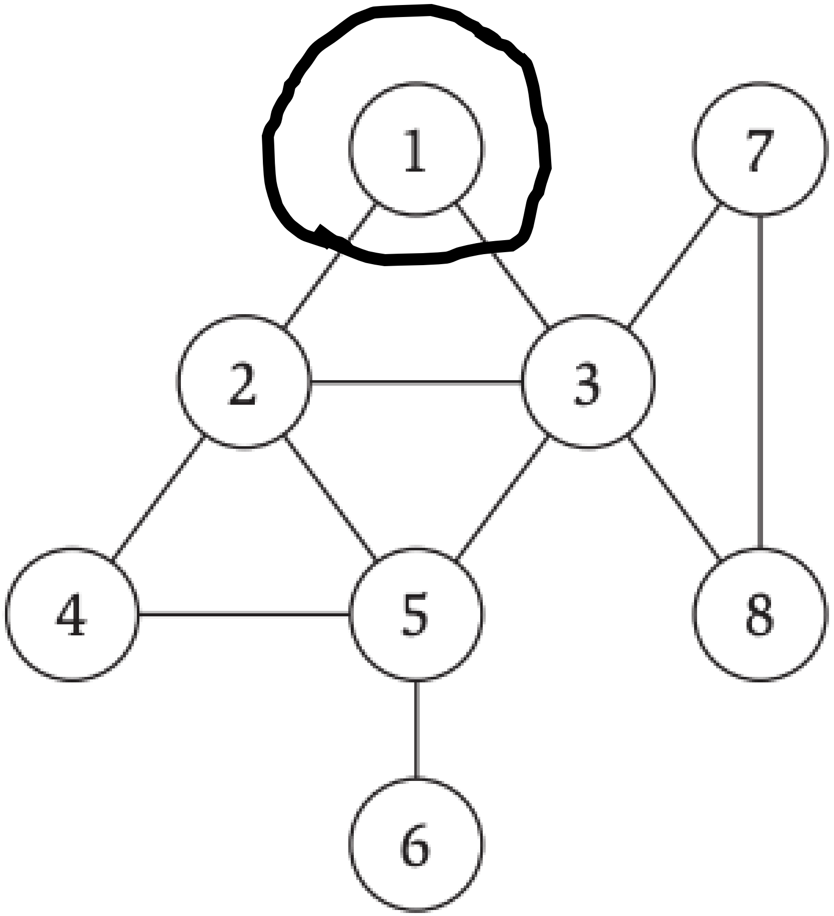
Depth First Search

- **Input**: The current vertex $\mathrm{u} \in \mathrm{V}$
- **Global**: An array of exploration $A \in \{0,1\}^V$
- Mark current vertex as explored ($\mathrm{A[u]} = 1$).
- For each $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$:
  - If v is not explored ($\mathrm{A[v]} == 0$):
    - DFS(v)

- **Idea**: You are recursing or "drilling down". If you get stuck, you go up a step and try the next choice.

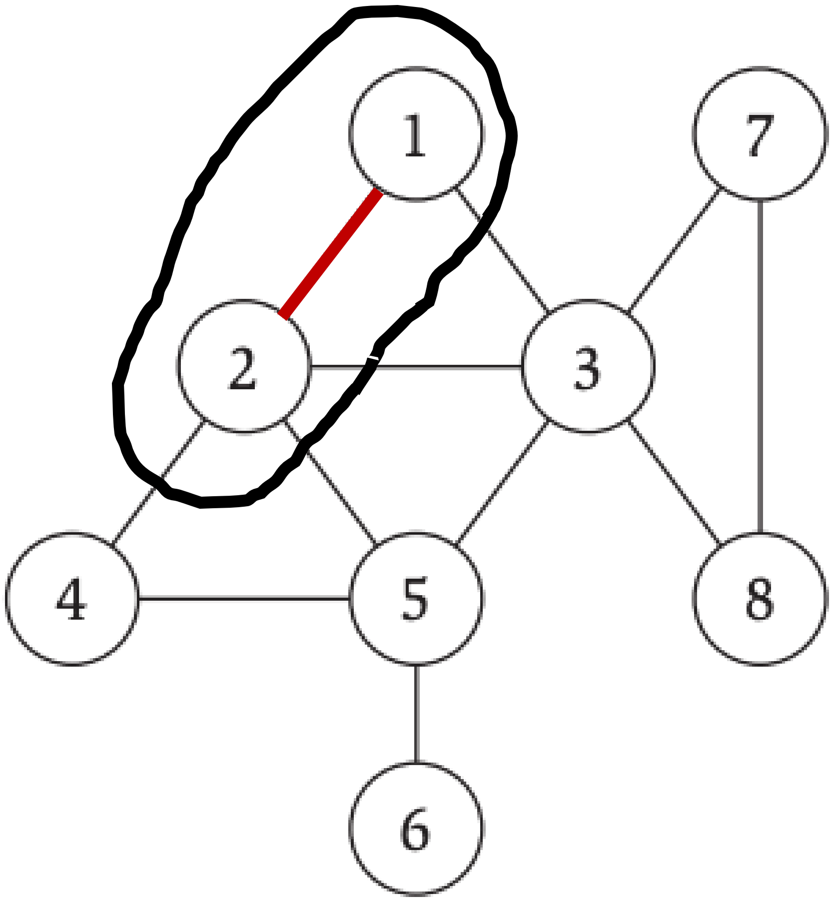
Depth First Search

A = {1}



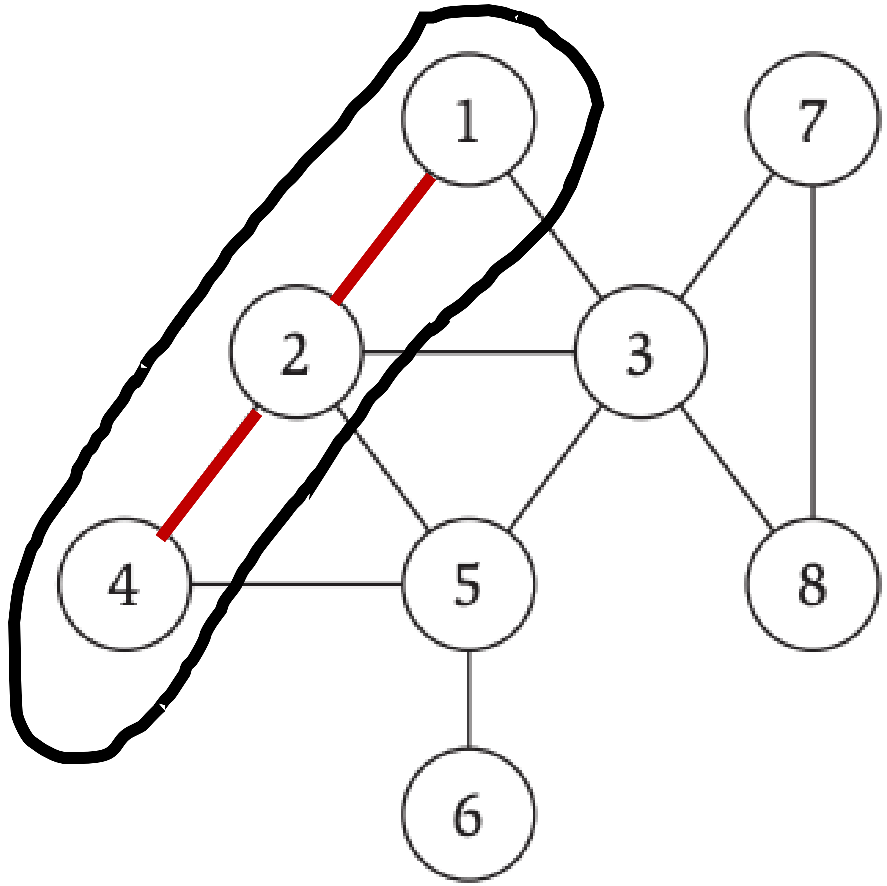
Depth First Search

A = {1,2}



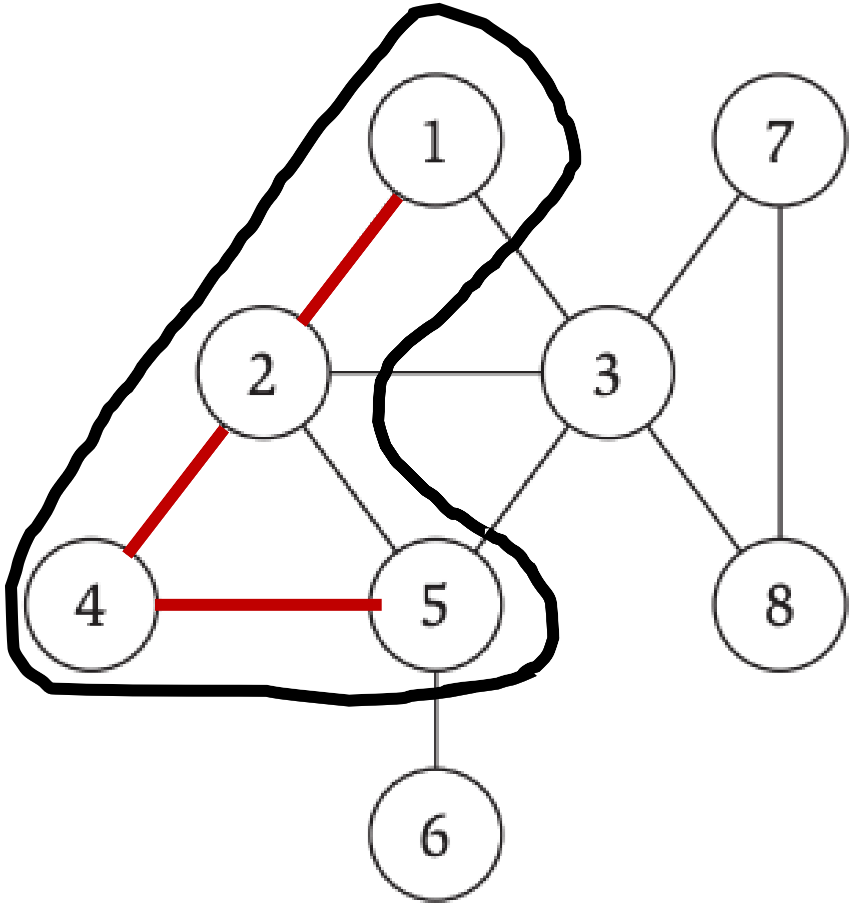
Depth First Search

A = {1,2,4}



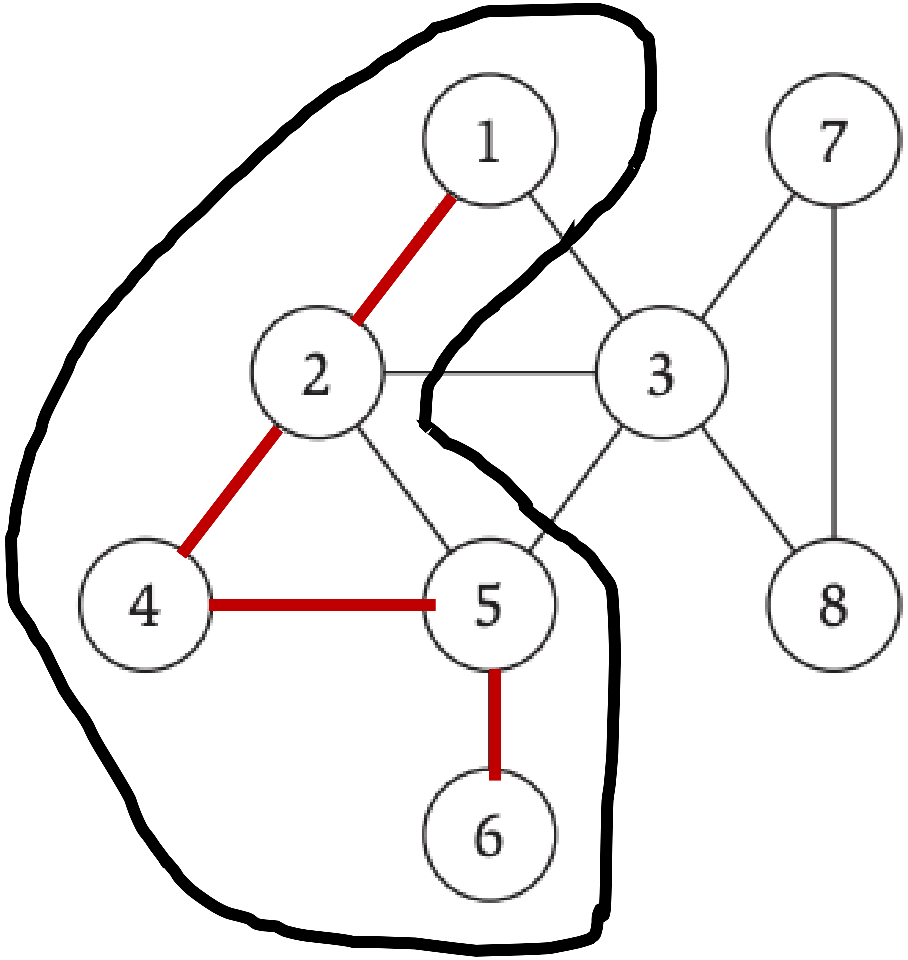
Depth First Search

A = {1,2,4,5}



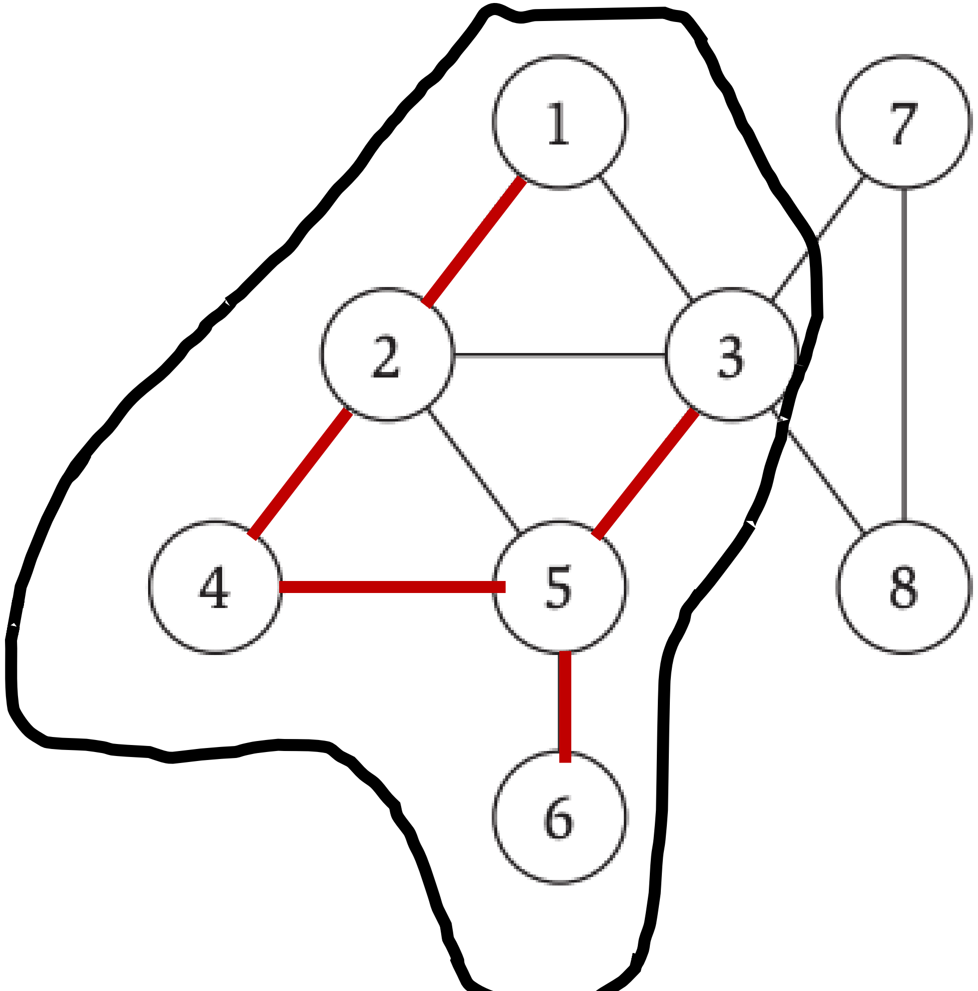
Depth First Search

A = {1,2,4,5,6}



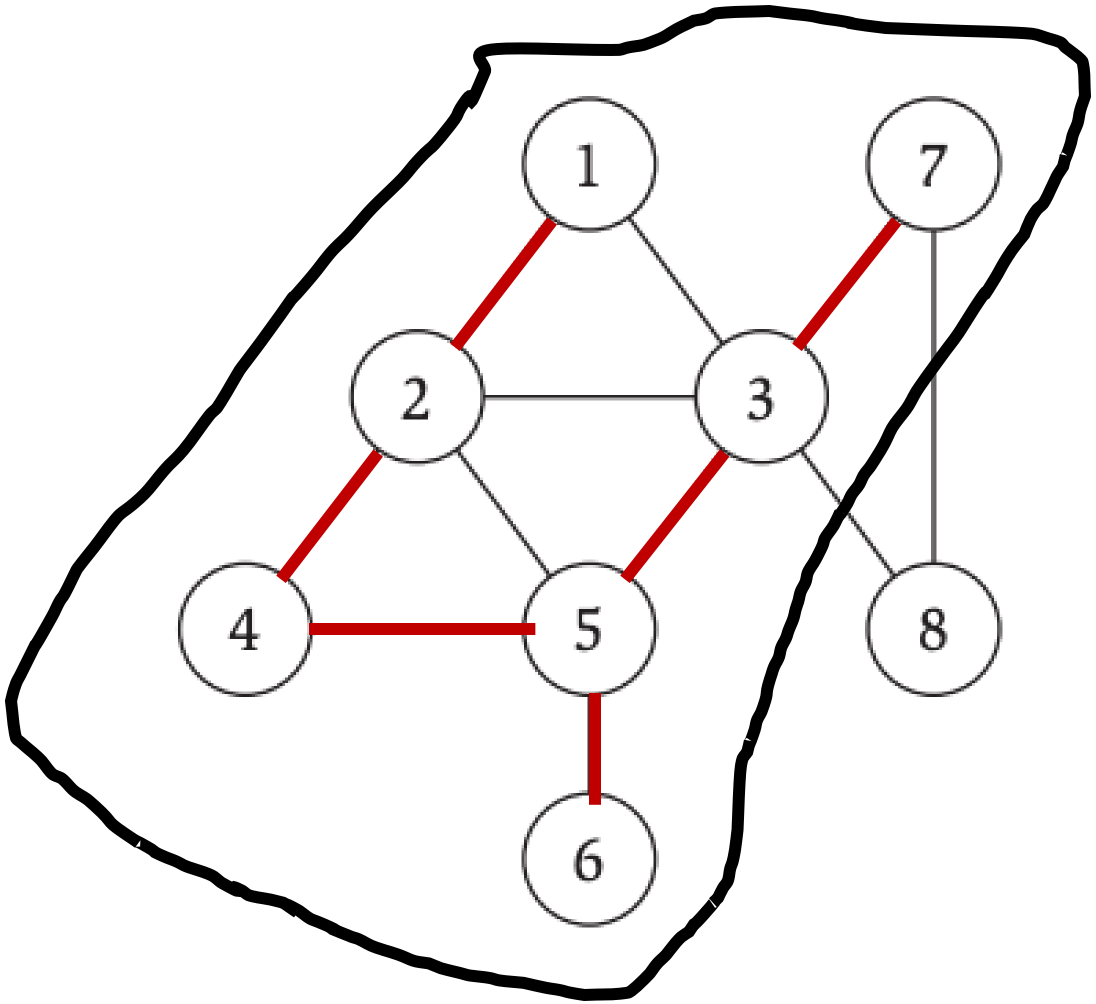
Depth First Search

A = {1,2,4,5,6,3}



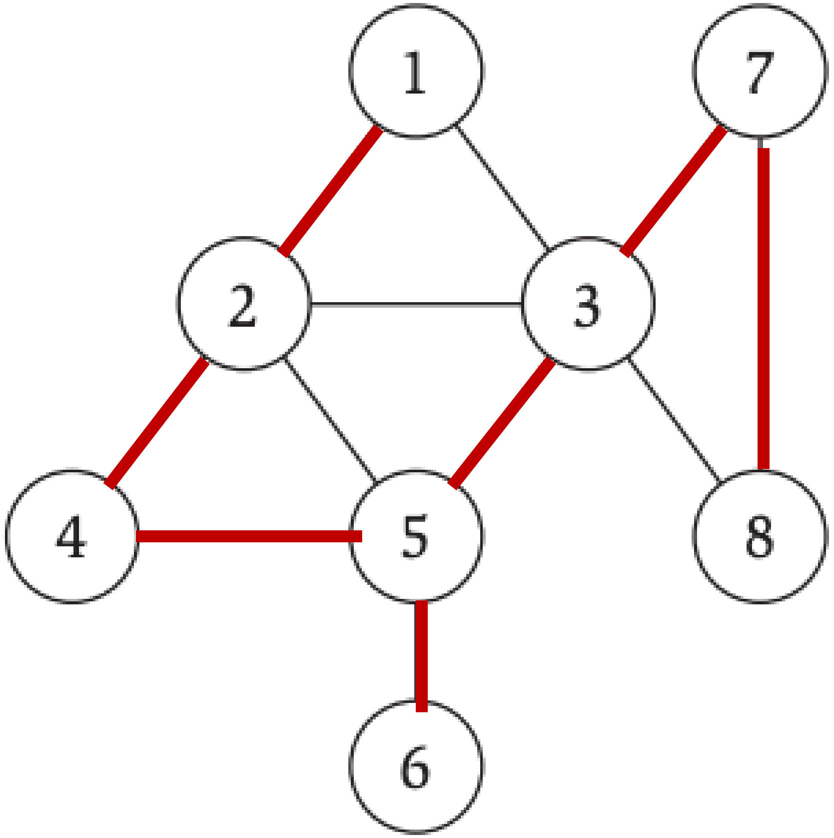
Depth First Search

A = {1,2,4,5,6,3,7}

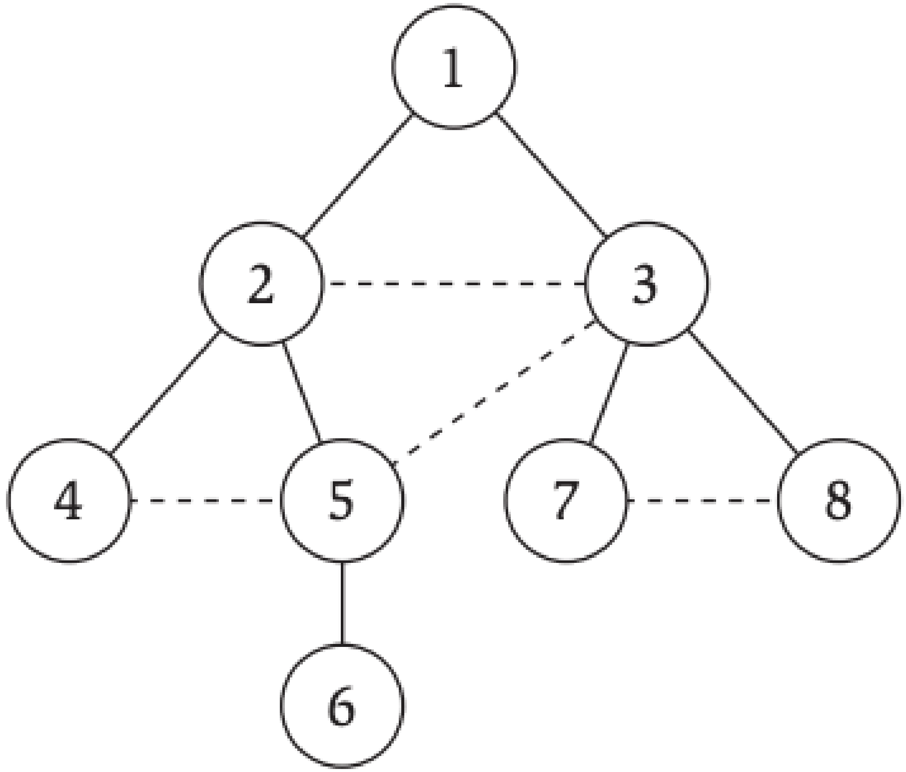


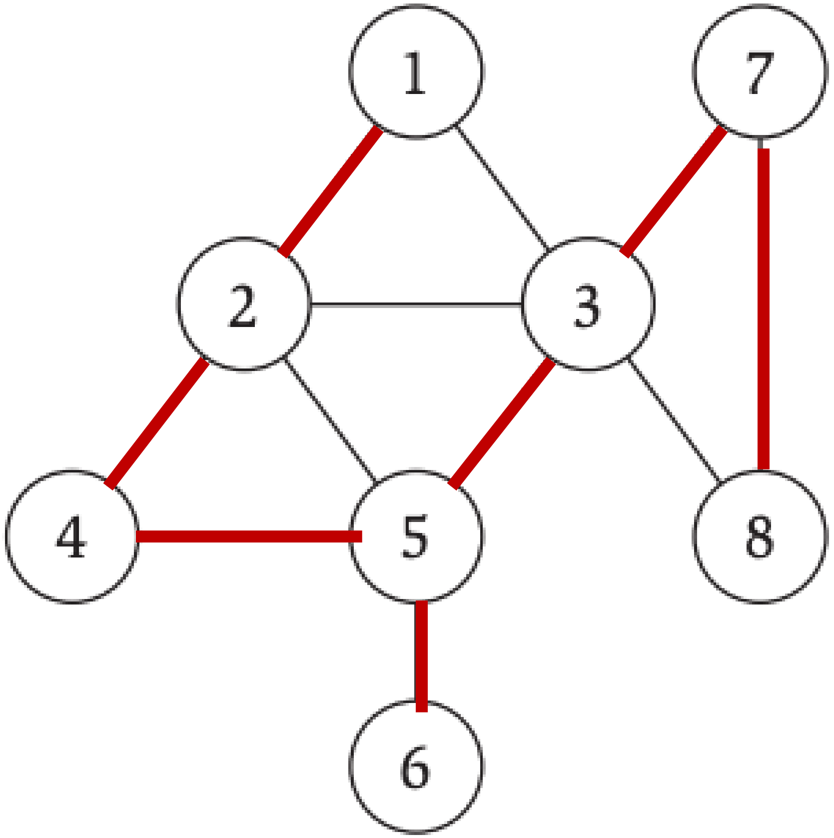
Depth First Search

A = {1,2,4,5,6,3,7,8}

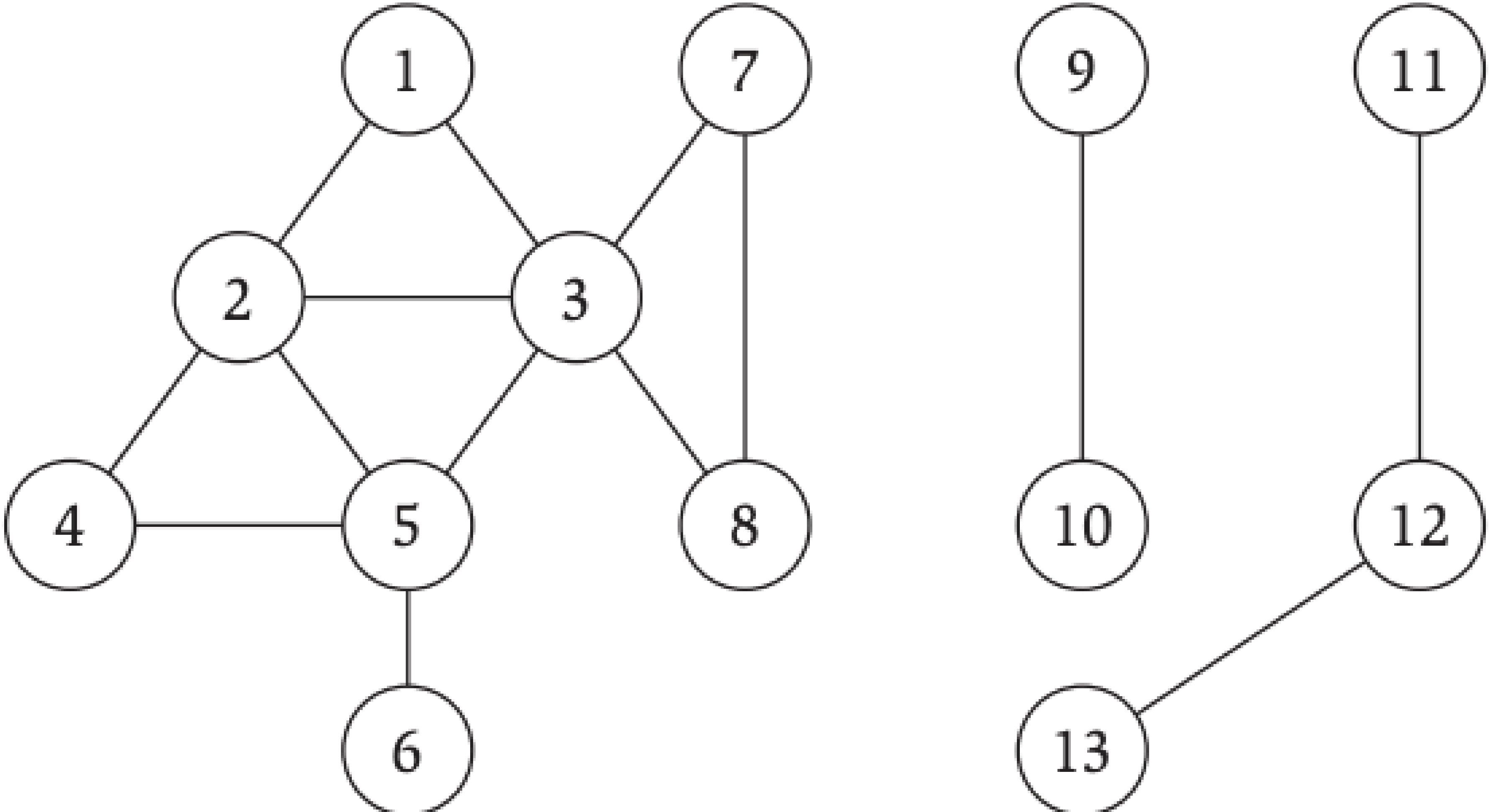


DFS Trees vs BFS Trees

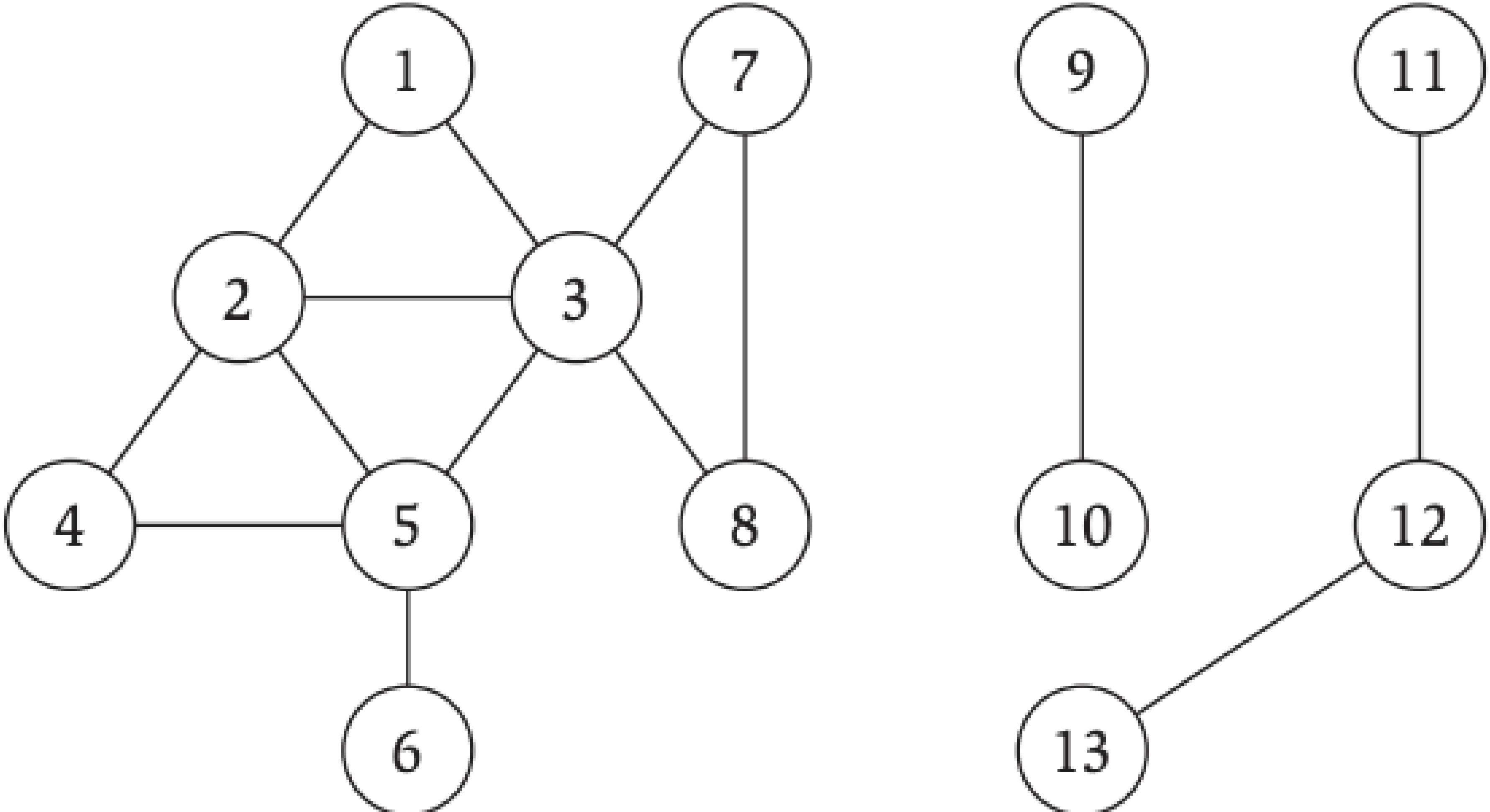




Q: How can you compute all the connected components of a graph?



A: Run Explore on a vertex. Keep track of all visited vertices. Run Explore on unvisited vertex. Stop when everyone is visited.



Q: What is the difference? (DFS vs Explore)

- **Input:** $G = (V, E)$ and $s \in V$
- **Output:** CC(s)
- Let $\mathrm{R} = \{s\}$
- While there exists $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$ such that $\mathrm{u} \in \mathrm{R}$ and $v \notin R$:
  - Add v to R
- Return R

- **Input**: The current vertex $\mathrm{u} \in \mathrm{V}$
- **Global**: An array $A \in \{0,1\}^V$
- Mark current vertex ($\mathrm{A[u]} = 1$).
- For each $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$:
  - If ($\mathrm{A[v]} == 0$):
    - DFS(v)

Q: What is the difference? (DFS vs Explore)

- **Input:** $G = (V, E)$ and $s \in V$
- **Output:** CC(s)
- Let $\mathrm{R} = \{\mathrm{s}\}$
- While there exists $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$ such that $\mathrm{u} \in \mathrm{R}$ and $v \notin R$:
  - Add v to R
- Return R

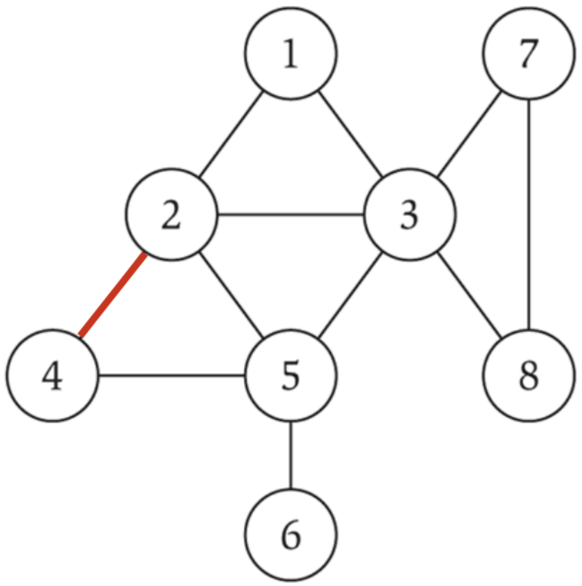
DFS is Explore but Explore isn't necessarily DFS!

- **Input**: The current vertex $\mathrm{u} \in \mathrm{V}$
- **Global**: An array $A \in \{0,1\}^V$
- Mark current vertex ($\mathrm{A[u]} = 1$).
- For each $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$:
  - If ($\mathrm{A[v]} == 0$):
    - DFS(v)

Adjacency List

Fix a graph $G = (V, E)$. The **adjacency list** of G is a vertex indexed array L of linked lists such that for each $v \in V$, $N[v]$ is a linked list that contains all neighbors of v exactly once.

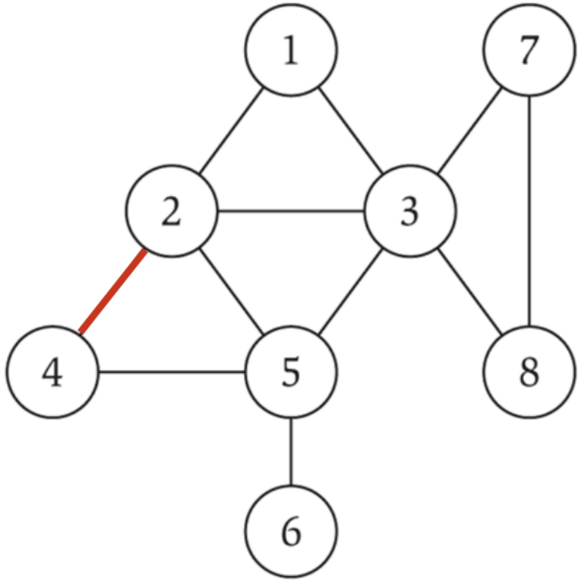
N[1] : [2] -> [3]
N[2] : [1] -> [3] -> [5] -> [4]
N[3] : [1] -> [2] -> [5] -> [7] -> [8]
N[4] : [2] -> [5]
N[5] : [2] -> [3] -> [4] -> [6]
N[6] : [5]
N[7] : [3] -> [8]
N[8] : [3] -> [7]



Adjacency List vs Adjacency Matrix

Space:
Lookup:
List Neighbors:

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0



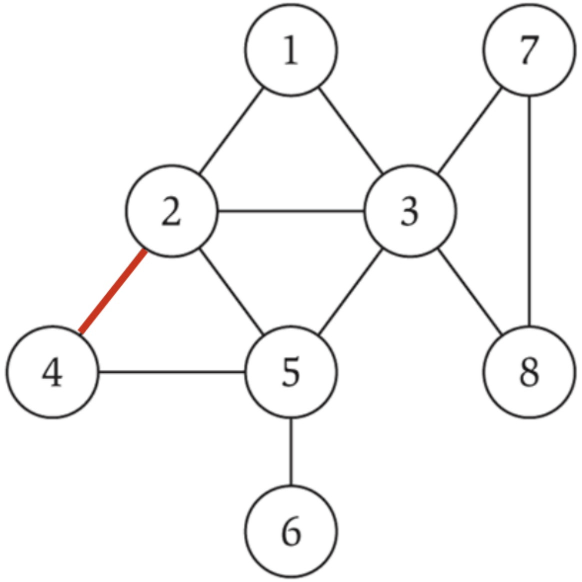
Space:
Lookup:
List Neighbors:

N[1] : [2] -> [3]
N[2] : [1] -> [3] -> [5] -> [4]
N[3] : [1] -> [2] -> [5] -> [7] -> [8]
N[4] : [2] -> [5]
N[5] : [2] -> [3] -> [4] -> [6]
N[6] : [5]
N[7] : [3] -> [8]
N[8] : [3] -> [7]

Adjacency List vs Adjacency Matrix

Space: O(n^2)
Lookup: O(1)
List Neighbors: O(n)

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0



Space: O(n + m)
Lookup: O(d_u)
List Neighbors: O(d_u)

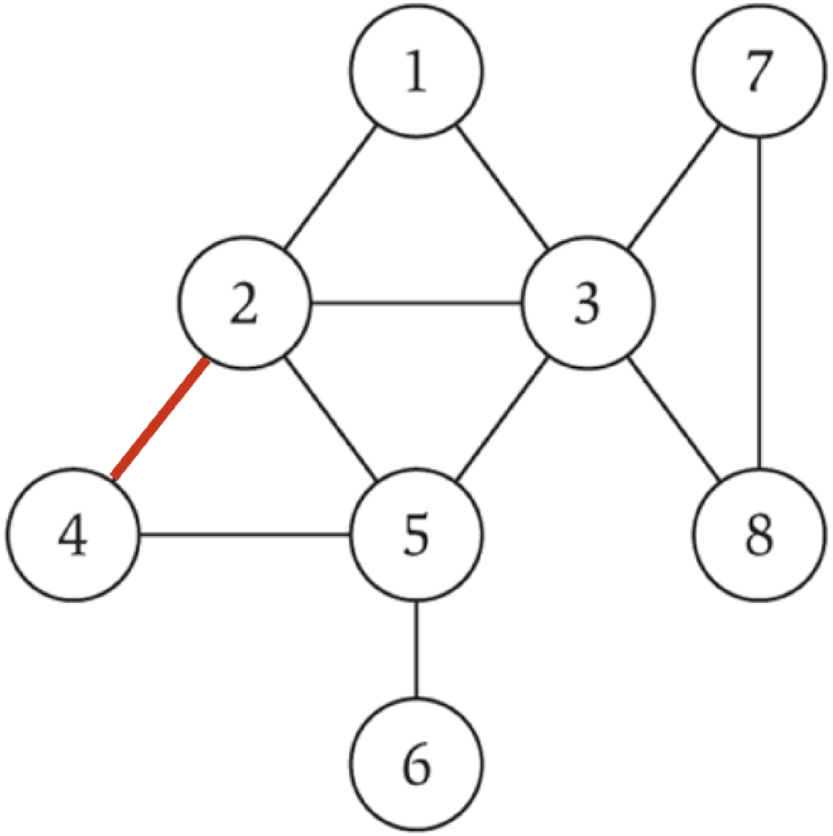
N[1] : [2] -> [3]
N[2] : [1] -> [3] -> [5] -> [4]
N[3] : [1] -> [2] -> [5] -> [7] -> [8]
N[4] : [2] -> [5]
N[5] : [2] -> [3] -> [4] -> [6]
N[6] : [5]
N[7] : [3] -> [8]
N[8] : [3] -> [7]

Degree Sum Formula or “Handshaking Lemma”

Fix a graph $G = (V, E)$. Say that every vertex $v \in V$ represents a person and two people share an edge if they have shaken hands.

Q1: How many handshakes happened?

Q2: How many times did someone prepare to shake a hand?

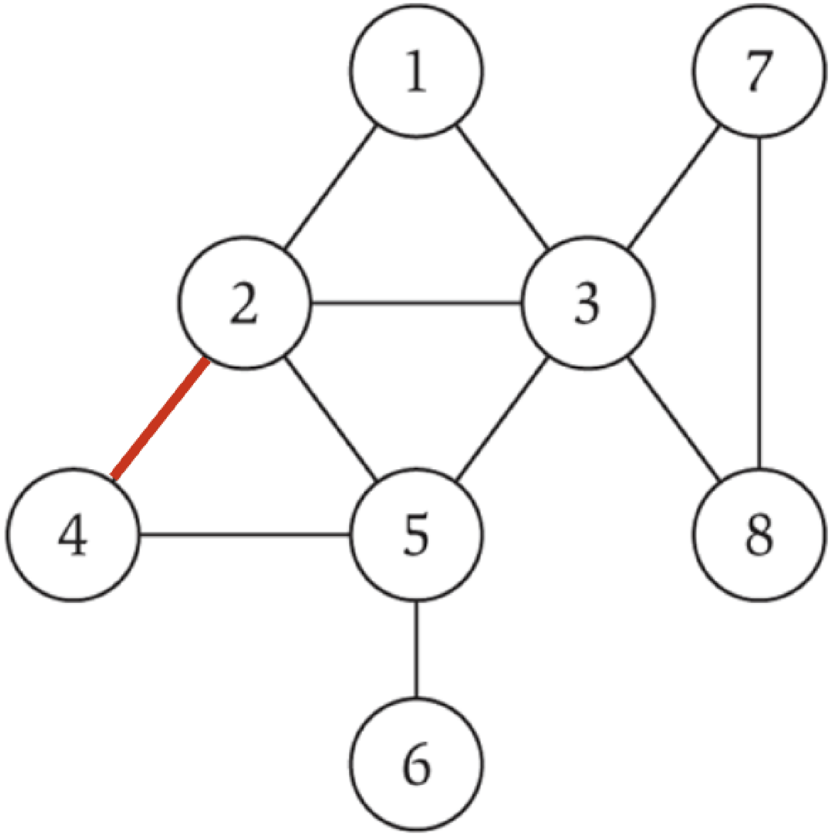


Degree Sum Formula or "Handshaking Lemma"

Fix a graph $G = (V, E)$. Say that every vertex $v \in V$ represents a person and two people share an edge if they have shaken hands.

A1: $|E|$

A2: $\sum_u d_u = 2|E|$



Q: What is a stack?

- A data structure for maintaining a set of elements.
- We can add and remove elements from the stack in constant time.
- When we remove an element, we get the last element that was added.
  - “last-in, first-out” or LIFO

Q: What is Queue?

- A data structure for maintaining a set of elements.
- We can add and remove elements from the stack in constant time.
- When we remove an element, we get the first element that was added (and is still in the set).
    - “first-in, first-out” or FIFO

Stack vs Queue

- Both can be implemented with a (doubly) linked list.
- Let's assume that both implement the remove function to take the first element of the linked list.
- Q: How do we implement the add function for Stack vs Queue?

Stack vs Queue

- Both can be implemented with a (doubly) linked list.
- Let's assume that both implement the remove function to take the first element of the linked list.
- A: For a Stack we insert at the front and for a Queue we insert at the end.

Breadth First Search

- **Input:** $G = (V, E)$ and $s \in V$
- **Output:** BFS Tree
- Let $L_0 = \{s\}$
- Assume $L_0, \ldots, L_i$ have been constructed:
  - Let $L_{\mathrm{i}+1}$ be nodes do not appear in $L_0, \ldots, L_i$ and have an edge to L_i.
  - If $L_{\mathrm{i}+1}$ is empty, stop.
- Return all layers.

A more specific BFS

```
BFS(s):
   Initialize Discovered to be a node index array of false
   Set Discovered[s] = true
   Initialize L[0] to be a linked list with one element s
   Initialize i to be 0
   While L[i] is not empty:
      Initialize L[i+1] to be empty linked list
      For u in L[i]:
            For each edge (u, v) incident to u:
                  If Discovered[v] = false:
                        Set Discovered[u] = true
                        Add v to L[i+1]
      ++i
```

Q: What is the runtime?

```
BFS(s):
   Initialize Discovered to be a node index array of false
   Set Discovered[s] = true
   Initialize L[0] to be a linked list with one element s
   Initialize i to be 0
   While L[i] is not empty:
      Initialize L[i+1] to be empty linked list
      For u in L[i]:
            For each edge (u, v) incident to u:
                  If Discovered[v] = false:
                        Set Discovered[u] = true
                        Add v to L[i+1]
      ++i
```

Total Time:

Q: What is the runtime?

```
BFS(s):
    Initialize Discovered to be a node index array of false
    Set Discovered[s] = true
    Initialize L[0] to be a linked list with one element s
    Initialize i to be 0
    While L[i] is not empty:
        Initialize L[i+1] to be empty linked list
        For u in L[i]:
            For each edge (u, v) incident to u:
                If Discovered[v] = false:
                    Set Discovered[u] = true
                    Add v to L[i+1]
        ++i
```

O(n) — Initialize Discovered to be a node index array of false

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
      Initialize L[0] to be a linked list with one element s
      Initialize i to be 0
      While L[i] is not empty:
         Initialize L[i+1] to be empty linked list
         For u in L[i]:
               For each edge (u, v) incident to u:
                     If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
         ++i
```

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
O(1)  Initialize L[0] to be a linked list with one element s
      Initialize i to be 0
      While L[i] is not empty:
         Initialize L[i+1] to be empty linked list
         For u in L[i]:
               For each edge (u, v) incident to u:
                     If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
         ++i
```

Total Time:

Q: What is the runtime?

```
     BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
O(1)  Initialize L[0] to be a linked list with one element s
O(1)  Initialize i to be 0
      While L[i] is not empty:
         Initialize L[i+1] to be empty linked list
         For u in L[i]:
               For each edge (u, v) incident to u:
                     If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
         ++i
```

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n) Initialize Discovered to be a node index array of false
O(1) Set Discovered[s] = true
O(1) Initialize L[0] to be a linked list with one element s
O(1) Initialize i to be 0
     While L[i] is not empty:
         Initialize L[i+1] to be empty linked list
         For u in L[i]:
               For each edge (u, v) incident to u:
                     If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
         ++i
```

Q: How many times does this loop run?

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
O(1)  Initialize L[0] to be a linked list with one element s
O(1)  Initialize i to be 0
      While L[i] is not empty:
         Initialize L[i+1] to be empty linked list
         For u in L[i]:
               For each edge (u, v) incident to u:
                     If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
         ++i
```

A: $|\bigcup_i L_i| \leq n$

Q: What is the runtime?

Total Time:

```
BFS(s):
O(n) Initialize Discovered to be a node index array of false
O(1) Set Discovered[s] = true
O(1) Initialize L[0] to be a linked list with one element s
O(1) Initialize i to be 0
O(n) While L[i] is not empty:
         Initialize L[i+1] to be empty linked list
         For u in L[i]:
               For each edge (u, v) incident to u:
                     If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
         ++i
```

Q: How many layers max?

Q: What is the runtime?

Total Time:

```
     BFS(s):
O(n)   Initialize Discovered to be a node index array of false
O(1)   Set Discovered[s] = true
O(1)   Initialize L[0] to be a linked list with one element s
O(1)   Initialize i to be 0
O(n)   While L[i] is not empty:
O(n)       Initialize L[i+1] to be empty linked list
           For u in L[i]:
                 For each edge (u, v) incident to u:
                       If Discovered[v] = false:
                             Set Discovered[u] = true
                             Add v to L[i+1]
           ++i
```

A: One for each vertex.

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n) Initialize Discovered to be a node index array of false
O(1) Set Discovered[s] = true
O(1) Initialize L[0] to be a linked list with one element s
O(1) Initialize i to be 0
O(n) While L[i] is not empty:
O(n)     Initialize L[i+1] to be empty linked list
         For u in L[i]:
               For each edge (u, v) incident to u:
                     If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
     ++i
```

Q: How many times does this loop run?

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
O(1)  Initialize L[0] to be a linked list with one element s
O(1)  Initialize i to be 0
O(n)  While L[i] is not empty:
O(n)      Initialize L[i+1] to be empty linked list
O(n)      For u in L[i]:
              For each edge (u, v) incident to u:
                  If Discovered[v] = false:
                      Set Discovered[u] = true
                      Add v to L[i+1]
          ++i
```

A: $|\bigcup_i L_i| \leq n$

Total Time:

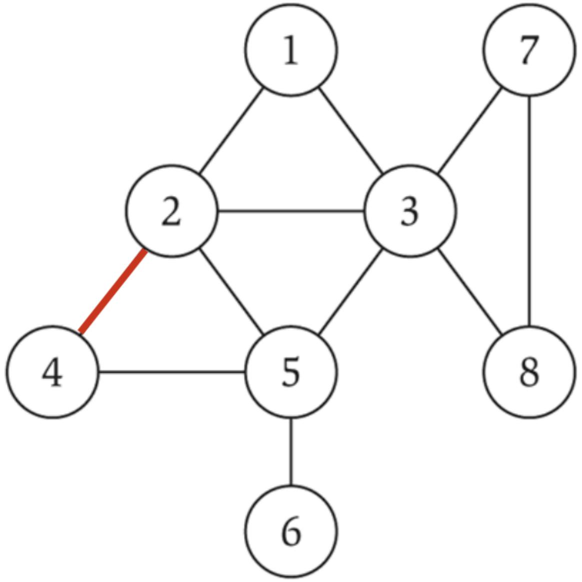
Q: What is the runtime?

```
      BFS(s):
O(n)    Initialize Discovered to be a node index array of false
O(1)    Set Discovered[s] = true
O(1)    Initialize L[0] to be a linked list with one element s
O(1)    Initialize i to be 0
O(n)    While L[i] is not empty:
O(n)        Initialize L[i+1] to be empty linked list
O(n)        For u in L[i]:
                  For each edge (u, v) incident to u:   Q: How do we do this?
                        If Discovered[v] = false:
                              Set Discovered[u] = true
                              Add v to L[i+1]
            ++i
```

Q: Which should we use?

Space: O(n^2)
Lookup: O(1)
List Neighbors: O(n)

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0



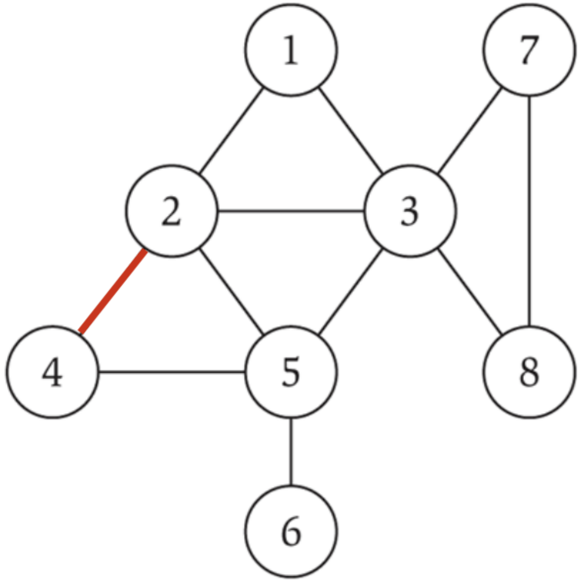
Space: O(n + m)
Lookup: O(d_u)
List Neighbors: O(d_u)

N[1] : [2] -> [3]
N[2] : [1] -> [3] -> [5] -> [4]
N[3] : [1] -> [2] -> [5] -> [7] -> [8]
N[4] : [2] -> [5]
N[5] : [2] -> [3] -> [4] -> [6]
N[6] : [5]
N[7] : [3] -> [8]
N[8] : [3] -> [7]

Adjacency List vs Adjacency Matrix

Space: O(n^2)
Lookup: O(1)
List Neighbors: O(n)

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0



Space: O(n + m)
Lookup: O(d_u)
List Neighbors: O(d_u)

N[1] : [2] -> [3]
N[2] : [1] -> [3] -> [5] -> [4]
N[3] : [1] -> [2] -> [5] -> [7] -> [8]
N[4] : [2] -> [5]
N[5] : [2] -> [3] -> [4] -> [6]
N[6] : [5]
N[7] : [3] -> [8]
N[8] : [3] -> [7]

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n) Initialize Discovered to be a node index array of false
O(1) Set Discovered[s] = true
O(1) Initialize L[0] to be a linked list with one element s
O(1) Initialize i to be 0
O(n) While L[i] is not empty:
O(n)     Initialize L[i+1] to be empty linked list
O(n)     For u in L[i]:
             For each edge (u, v) incident to u:   A: Linked List
                 If Discovered[v] = false:
                     Set Discovered[u] = true
                     Add v to L[i+1]
     ++i
```

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
O(1)  Initialize L[0] to be a linked list with one element s
O(1)  Initialize i to be 0
O(n)  While L[i] is not empty:
O(n)      Initialize L[i+1] to be empty linked list
O(n)      For u in L[i]:
O(d_u)        For each edge (u, v) incident to u:
                  If Discovered[v] = false:
                      Set Discovered[u] = true
                      Add v to L[i+1]
          ++i
```

Q: How many times does this loop run for u?

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n)     Initialize Discovered to be a node index array of false
O(1)     Set Discovered[s] = true
O(1)     Initialize L[0] to be a linked list with one element s
O(1)     Initialize i to be 0
O(n)     While L[i] is not empty:
O(n)         Initialize L[i+1] to be empty linked list
O(n)         For u in L[i]:
O(∑d_u)          For each edge (u, v) incident to u:
                      If Discovered[v] = false:
                           Set Discovered[u] = true
                           Add v to L[i+1]
             ++i
```

O($\sum d_u$)

A: One time for each u!

Q: What is the runtime?

Total Time:

```
BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
O(1)  Initialize L[0] to be a linked list with one element s
O(1)  Initialize i to be 0
O(n)  While L[i] is not empty:
O(n)      Initialize L[i+1] to be empty linked list
O(n)      For u in L[i]:
O(m)          For each edge (u, v) incident to u:
                  If Discovered[v] = false:
                      Set Discovered[u] = true
                      Add v to L[i+1]
          ++i
```

Total Time:

Q: What is the runtime?

```
BFS(s):
O(n)  Initialize Discovered to be a node index array of false
O(1)  Set Discovered[s] = true
O(1)  Initialize L[0] to be a linked list with one element s
O(1)  Initialize i to be 0
O(n)  While L[i] is not empty:
O(n)      Initialize L[i+1] to be empty linked list
O(n)      For u in L[i]:
O(m)              For each edge (u, v) incident to u:
              O(1)If Discovered[v] = false:
                  O(1) Set Discovered[u] = true
                  O(1) Add v to L[i+1]
    O(1)++i
```

Q: What is the runtime?

Total Time:

```
       BFS(s):
O(n)     Initialize Discovered to be a node index array of false
O(1)     Set Discovered[s] = true
O(1)     Initialize L[0] to be a linked list with one element s
O(1)     Initialize i to be 0
O(n)     While L[i] is not empty:
O(n)         Initialize L[i+1] to be empty linked list
O(n)         For u in L[i]:
O(m)             For each edge (u, v) incident to u:
O(m)                 If Discovered[v] = false:
O(m)                     Set Discovered[u] = true
O(m)                     Add v to L[i+1]
O(n)         ++i
```

Total Time:

Q: What is the runtime? O(n + m)

```
BFS(s):
O(n) Initialize Discovered to be a node index array of false
O(1) Set Discovered[s] = true
O(1) Initialize L[0] to be a linked list with one element s
O(1) Initialize i to be 0
O(n) While L[i] is not empty:
O(n)     Initialize L[i+1] to be empty linked list
O(n)     For u in L[i]:
O(m)         For each edge (u, v) incident to u:
O(m)             If Discovered[v] = false:
O(m)                 Set Discovered[u] = true
O(m)                 Add v to L[i+1]
O(n)     ++i
```

Q: What if I use a matrix?

```
BFS(s):
   Initialize Discovered to be a node index array of false
   Set Discovered[s] = true
   Initialize L[0] to be a linked list with one element s
   Initialize i to be 0
   While L[i] is not empty:
      Initialize L[i+1] to be empty linked list
      For u in L[i]:
            For each edge (u, v) incident to u:
                  If Discovered[v] = false:
                        Set Discovered[u] = true
                        Add v to L[i+1]
      ++i
```

Q: What if I use a matrix? O(n^2)

```
BFS(s):
    Initialize Discovered to be a node index array of false
    Set Discovered[s] = true
    Initialize L[0] to be a linked list with one element s
    Initialize i to be 0
    While L[i] is not empty:
        Initialize L[i+1] to be empty linked list
        For u in L[i]:
O(n^2)          For each edge (u, v) incident to u:
O(n^2)              If Discovered[v] = false:
O(n^2)                  Set Discovered[u] = true
O(n^2)                  Add v to L[i+1]
        ++i
```

A New BFS

```
BFS(s):
   Initialize Discovered to be a node index array of false
   Set Discovered[s] = true
   Initialize Q to be a Queue with one element s
   While Q is not empty:
      u = Q.dequeue()
      For each edge (u, v) incident to u:
            If Discovered[v] = false:
                  Set Discovered[u] = true
                  Add v to Q
      ++i
```

Q: What is the runtime? (Assume Linked List)

```
BFS(s):
   Initialize Discovered to be a node index array of false
   Set Discovered[s] = true
   Initialize Q to be a Queue with one element s
   While Q is not empty:
      u = Q.dequeue()
      For each edge (u, v) incident to u:
            If Discovered[v] = false:
                  Set Discovered[u] = true
                  Add v to Q
      ++i
```

Q: What is the runtime? O(m+n)

```
BFS(s):
   Initialize Discovered to be a node index array of false
   Set Discovered[s] = true
   Initialize Q to be a Queue with one element s
   While Q is not empty:
      u = Q.dequeue()
      For each edge (u, v) incident to u:
            If Discovered[v] = false:
                  Set Discovered[u] = true
                  Add v to Q
      ++i
```

Q: What if we change the queue to a stack?

```
???(s):
   Initialize Explored to be a node index array of false
   Set Explored[s] = true
   Initialize Q to be a Stack with one element s
   While Q is not empty:
      u = Q.remove()
      If Explored[u] = false:
            For each edge (u, v) incident to u:
                  Add v to Q
      ++i
```

A: We DFS

```
DFSs):
   Initialize Explored to be a node index array of false
   Set Explored[s] = true
   Initialize Q to be a Stack with one element s
   While Q is not empty:
      u = Q.remove()
      If Explored[u] = false:
            For each edge (u, v) incident to u:
                  Add v to Q
      ++i
```

Q: What is the runtime of DFS?

```
DFSs):
   Initialize Explored to be a node index array of false
   Set Explored[s] = true
   Initialize Q to be a Stack with one element s
   While Q is not empty:
      u = Q.remove()
      If Explored[u] = false:
            For each edge (u, v) incident to u:
                  Add v to Q
      ++i
```

Q: What is the runtime of DFS? O(m+n)

```
DFSs):
   Initialize Explored to be a node index array of false
   Set Explored[s] = true
   Initialize Q to be a Stack with one element s
   While Q is not empty:
      u = Q.remove()
      If Explored[u] = false:
            For each edge (u, v) incident to u:
                  Add v to Q
      ++i
```

Q: How many times does this loop run for u?

Q: What is the runtime of DFS? O(m+n)

```
DFSs):
   Initialize Explored to be a node index array of false
   Set Explored[s] = true
   Initialize Q to be a Stack with one element s
   While Q is not empty:
      u = Q.remove()
      If Explored[u] = false:
            For each edge (u, v) incident to u:
                  Add v to Q
      ++i
```

A: At most once for each edge

Problem: Bipartite Graph

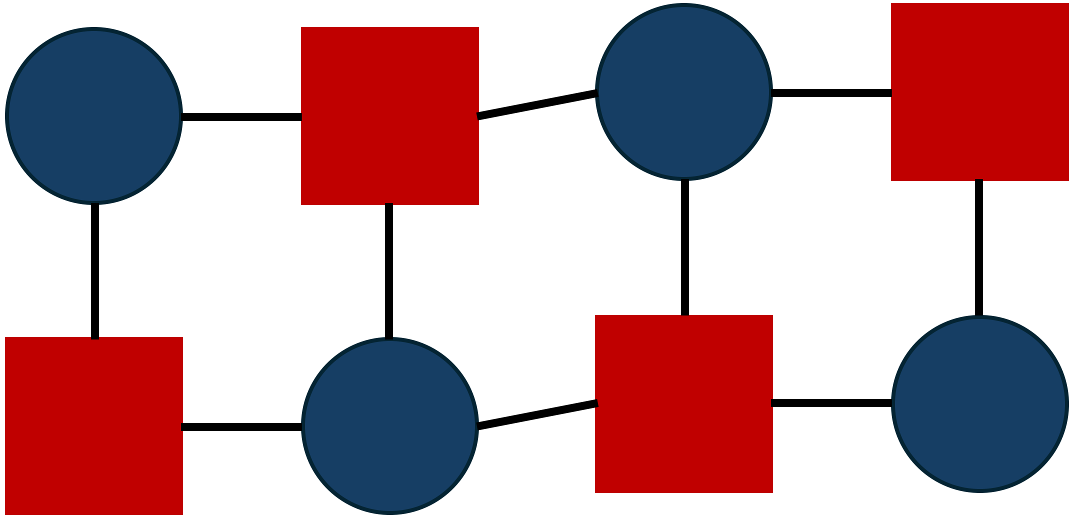
Input: A graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$

Output: True if G is bipartite and False otherwise.

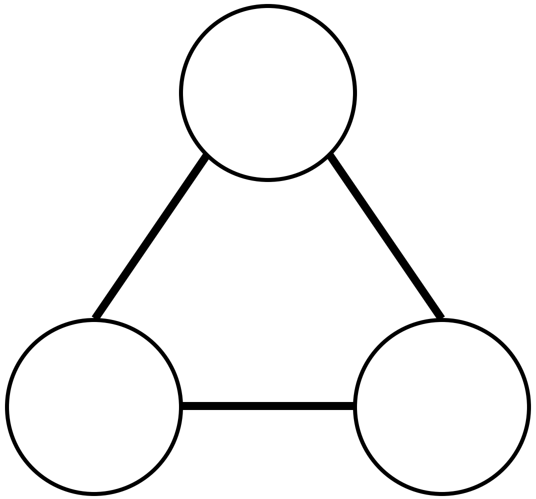
Def: We say that a graph is bipartite if we can partition the vertices V into two groups L and R such that each edge has one endpoint in L and the other endpoint in R.

Problem: Bipartite Graph

Def: We say that a graph is bipartite if we can partition the vertices V into two groups L and R such that each edge has one endpoint in L and the other endpoint in R.



Bipartite



Not Bipartite

Problem: Bipartite Graph

Input: A graph $G = (V, E)$

Output: True if G is bipartite and False otherwise.

Algorithm Ideas:

Problem: Bipartite Graph

Input: A graph $G = (V, E)$

Output: True if G is bipartite and False otherwise.

Algorithm Ideas: We will find the layering produced by BFS and color odd levels Red and even layers Blue.

Q: When will this fail?

Problem: Bipartite Graph

Input: A graph $G = (V, E)$

Output: True if G is bipartite and False otherwise.

Algorithm Ideas: We will find the layering produced by BFS and color odd levels Red and even layers Blue. This will fail if there is a cross edge in one of the layers.

Q: What does this imply?

Problem: Bipartite Graph

Input: A graph $G = (V, E)$

Output: True if G is bipartite and False otherwise.

Algorithm Ideas: We will find the layering produced by BFS and color odd levels Red and even layers Blue. This will fail if there is a cross edge in one of the layers. This implies there is an odd cycle in the graph.

Q: Is that a problem?

Problem: Bipartite Graph

Input: A graph $G = (V, E)$

Output: True if G is bipartite and False otherwise.

Algorithm Ideas: We will find the layering produced by BFS and color odd levels Red and even layers Blue. This will fail if there is a cross edge in one of the layers. This implies there is an odd cycle in the graph. We can show that a graph is not bipartite if it contains an odd cycle.