Week 08: Connectivity

FanFly

April 26, 2020

Data Structures

Let us review some data structures we have learned.

Data Structures

Let us review some data structures we have learned.

List

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.
- Update: $O(1)$ time.

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.
- Update: $O(1)$ time.
- Delete: $O(n)$ time.

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.
- Update: $O(1)$ time.
- Delete: $O(n)$ time.

Dictionary / Set

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.
- Update: $O(1)$ time.
- Delete: $O(n)$ time.

Dictionary / Set

- Create: $O(1)$ time (average-case).

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.
- Update: $O(1)$ time.
- Delete: $O(n)$ time.

Dictionary / Set

- Create: $O(1)$ time (average-case).
- Search: $O(1)$ time (average-case).

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.
- Update: $O(1)$ time.
- Delete: $O(n)$ time.

Dictionary / Set

- Create: $O(1)$ time (average-case).
- Search: $O(1)$ time (average-case).
- Update: $O(1)$ time.

Data Structures

Let us review some data structures we have learned.

List

- Create: $O(1)$ time (amortized).
- Search: $O(n)$ time.
- Update: $O(1)$ time.
- Delete: $O(n)$ time.

Dictionary / Set

- Create: $O(1)$ time (average-case).
- Search: $O(1)$ time (average-case).
- Update: $O(1)$ time.
- Delete: $O(1)$ time.

Data Structures for Graphs

We learned adjacency lists and adjacency matrix last week.

Adjacency List

Data Structures for Graphs

We learned adjacency lists and adjacency matrix last week.

Adjacency List

- Preprocess: $O(n + m)$ time.

Data Structures for Graphs

We learned adjacency lists and adjacency matrix last week.

Adjacency List

- Preprocess: $O(n + m)$ time.
- Check if vertices u and v are adjacent: $O(d(u))$ time.

Data Structures for Graphs

We learned adjacency lists and adjacency matrix last week.

Adjacency List

- Preprocess: $O(n+m)$ time.
- Check if vertices u and v are adjacent: $O(d(u))$ time.

Adjacency Matrix

Data Structures for Graphs

We learned adjacency lists and adjacency matrix last week.

Adjacency List

- Preprocess: $O(n + m)$ time.
- Check if vertices u and v are adjacent: $O(d(u))$ time.

Adjacency Matrix

- Preprocess: $O(n^2)$ time.

Data Structures for Graphs

We learned adjacency lists and adjacency matrix last week.

Adjacency List

- Preprocess: $O(n + m)$ time.
- Check if vertices u and v are adjacent: $O(d(u))$ time.

Adjacency Matrix

- Preprocess: $O(n^2)$ time.
- Check if vertices u and v are adjacent: $O(1)$ time.

Connectivity

In a graph $G = (V, E)$, vertices u and v are said to be **connected** if there is a path from u to v.

Connectivity

In a graph $G = (V, E)$, vertices u and v are said to be **connected** if there is a path from u to v.

- Also, we say that a graph is **connected** if any two vertices in the graph are connected.

Connectivity

In a graph $G = (V, E)$, vertices u and v are said to be **connected** if there is a path from u to v.

- Also, we say that a graph is **connected** if any two vertices in the graph are connected.

Today, we want to build a data structure satisfying the following properties.

Connectivity

In a graph $G = (V, E)$, vertices u and v are said to be **connected** if there is a path from u to v.

- Also, we say that a graph is **connected** if any two vertices in the graph are connected.

Today, we want to build a data structure satisfying the following properties.

A Mysterious Data Structure

Connectivity

In a graph $G = (V, E)$, vertices u and v are said to be **connected** if there is a path from u to v.

- Also, we say that a graph is **connected** if any two vertices in the graph are connected.

Today, we want to build a data structure satisfying the following properties.

A Mysterious Data Structure

- Preprocess: $O(n + m)$ time.

Connectivity

In a graph $G = (V, E)$, vertices u and v are said to be **connected** if there is a path from u to v.

- Also, we say that a graph is **connected** if any two vertices in the graph are connected.

Today, we want to build a data structure satisfying the following properties.

A Mysterious Data Structure

- Preprocess: $O(n + m)$ time.
- Check if vertices u and v are connected: $O(1)$ time.

Connected Components

We want to label the vertices with the **connected components** they belong to.

Connected Components

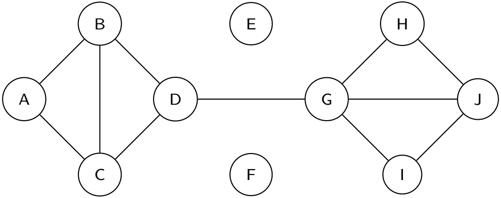
We want to label the vertices with the **connected components** they belong to.

- A connected component of a graph is a maximal set of vertices such that each pair of vertices are connected.

Connected Components

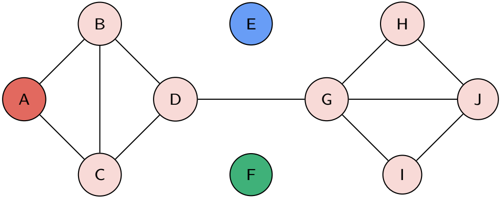
We want to label the vertices with the **connected components** they belong to.

- A connected component of a graph is a maximal set of vertices such that each pair of vertices are connected.



Connected Components (cont.)

For each connected components, we choose a representative to identify it.



The Graph Class

We are going to introduce a very useful “algorithm”, which is called **depth-first search**.

The Graph Class

We are going to introduce a very useful "algorithm", which is called **depth-first search**.

First we construct a class in Python called `Graph`.

The Graph Class

We are going to introduce a very useful "algorithm", which is called **depth-first search**.

First we construct a class in Python called `Graph`.

```
class Graph:
```

The Graph Class

We are going to introduce a very useful “algorithm”, which is called **depth-first search**.

First we construct a class in Python called `Graph`.

```
class Graph:
    def __init__(self, vertices):
        self.vertices = list(vertices)
        self.adj = {u: [] for u in vertices}
```

The Graph Class

We are going to introduce a very useful “algorithm”, which is called **depth-first search**.

First we construct a class in Python called `Graph`.

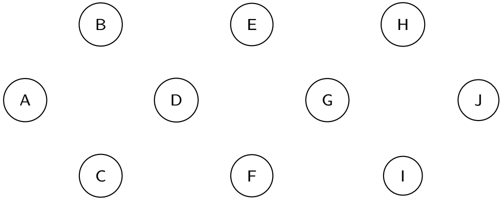
```python
class Graph:
    def __init__(self, vertices):
        self.vertices = list(vertices)
        self.adj = {u: [] for u in vertices}

    def add_edge(self, u, v):
        self.adj[u].append(v)
        self.adj[v].append(u)
```

The Graph Class (cont.)

For example, we can represent the graph as follows.

The Graph Class (cont.)

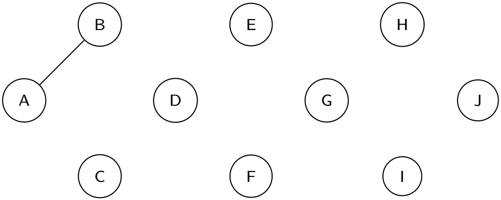
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
```

The Graph Class (cont.)

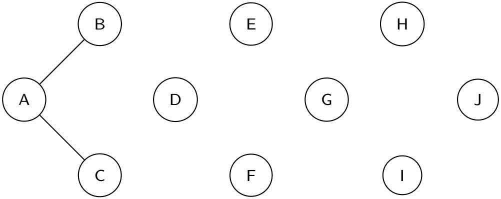
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
```

The Graph Class (cont.)

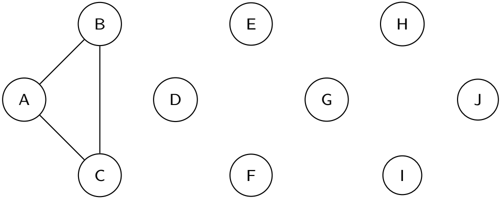
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
```

The Graph Class (cont.)

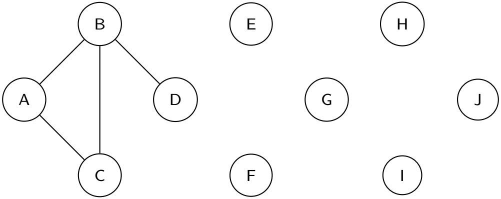
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
```

The Graph Class (cont.)

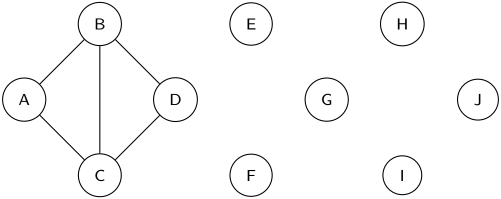
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
```

The Graph Class (cont.)

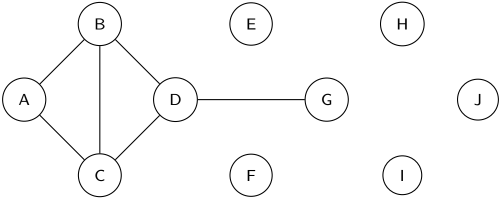
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
>>> G.add_edge("C", "D")
```

The Graph Class (cont.)

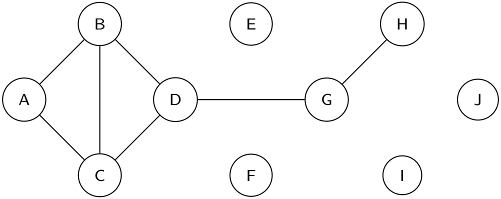
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
>>> G.add_edge("C", "D")
>>> G.add_edge("D", "G")
```

The Graph Class (cont.)

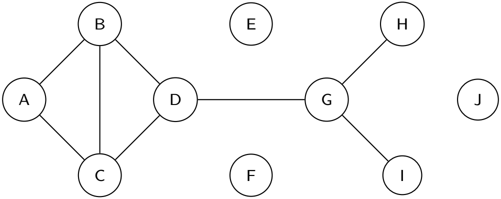
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
>>> G.add_edge("C", "D")
>>> G.add_edge("D", "G")
>>> G.add_edge("G", "H")
```

The Graph Class (cont.)

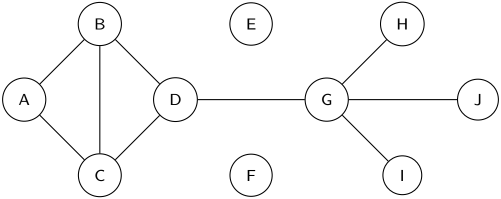
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
>>> G.add_edge("C", "D")
>>> G.add_edge("D", "G")
>>> G.add_edge("G", "H")
>>> G.add_edge("G", "I")
```

The Graph Class (cont.)

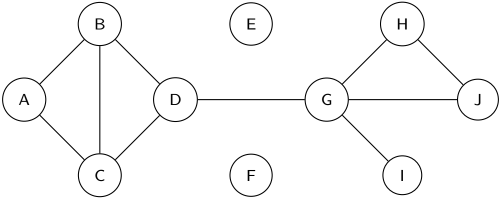
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
>>> G.add_edge("C", "D")
>>> G.add_edge("D", "G")
>>> G.add_edge("G", "H")
>>> G.add_edge("G", "I")
>>> G.add_edge("G", "J")
```

The Graph Class (cont.)

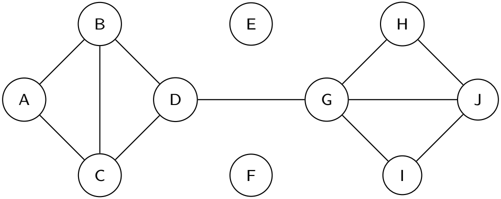
For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
>>> G.add_edge("C", "D")
>>> G.add_edge("D", "G")
>>> G.add_edge("G", "H")
>>> G.add_edge("G", "I")
>>> G.add_edge("G", "J")
>>> G.add_edge("H", "J")
```

The Graph Class (cont.)

For example, we can represent the graph as follows.




```
>>> G = Graph(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])
>>> G.add_edge("A", "B")
>>> G.add_edge("A", "C")
>>> G.add_edge("B", "C")
>>> G.add_edge("B", "D")
>>> G.add_edge("C", "D")
>>> G.add_edge("D", "G")
>>> G.add_edge("G", "H")
>>> G.add_edge("G", "I")
>>> G.add_edge("G", "J")
>>> G.add_edge("H", "J")
>>> G.add_edge("I", "J")
```

Depth-First Search

Now we introduce the depth-first search algorithm.

Depth-First Search

Now we introduce the depth-first search algorithm.

```
class Graph:
    ...
    def dfs(self):
        visited = set()
```

Depth-First Search

Now we introduce the depth-first search algorithm.

```python
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A recursive function used by depth first search
        def dfs_visit(u):
            visited.add(u)
            for v in self.adj[u]:
                if v not in visited:
                    dfs_visit(v)
```

Depth-First Search

Now we introduce the depth-first search algorithm.

```
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A recursive function used by depth first search
        def dfs_visit(u):
            visited.add(u)
            for v in self.adj[u]:
                if v not in visited:
                    dfs_visit(v)
        # Start depth first search
        for u in self.vertices:
            if u not in visited:
                dfs_visit(u)
```

Depth-First Search

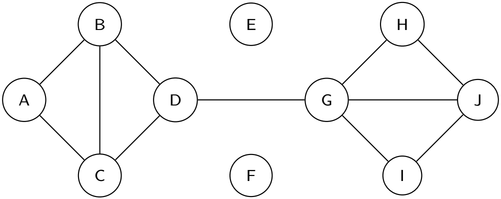
Now we introduce the depth-first search algorithm.

```python
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A recursive function used by depth first search
        def dfs_visit(u):
            visited.add(u)
            for v in self.adj[u]:
                if v not in visited:
                    dfs_visit(v)
        # Start depth first search
        for u in self.vertices:
            if u not in visited:
                dfs_visit(u)
```

In fact, the method `dfs()` does not do anything, but it traverses all the vertices exactly once in a specific order.

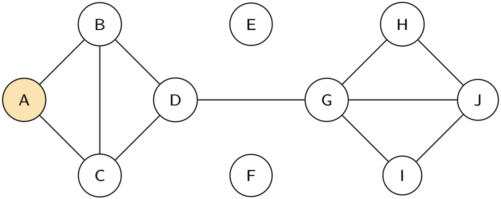
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



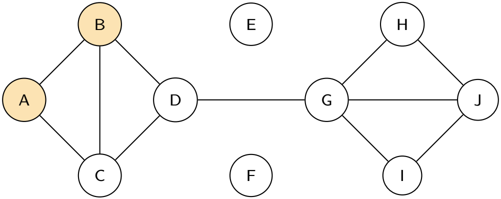
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



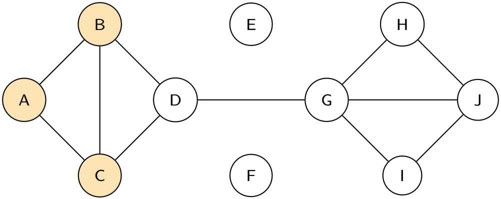
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



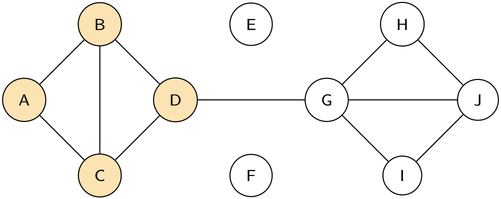
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



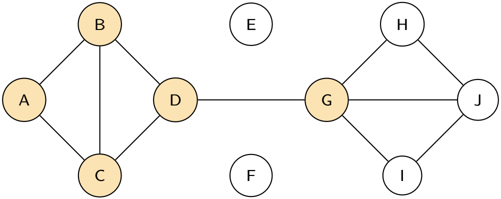
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



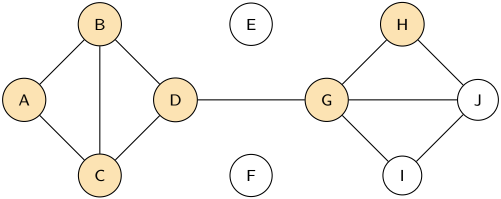
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



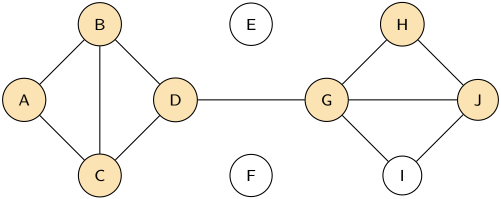
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



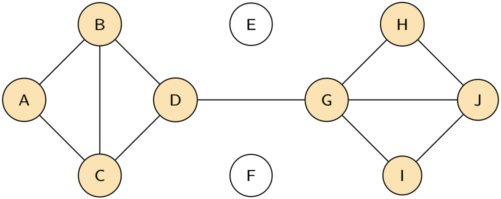
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



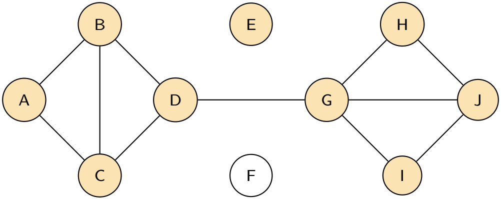
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



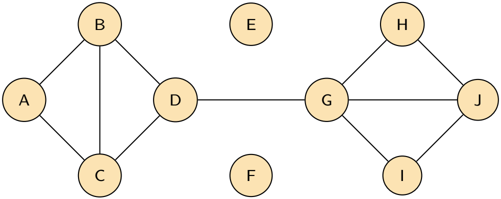
Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



Depth-First Search (cont.)

```
>>> for u in G.vertices:
...     print(u, G.adj[u])
...
A ["B", "C"]
B ["A", "C", "D"]
C ["A", "B", "D"]
D ["B", "C", "G"]
E []
F []
G ["D", "H", "I", "J"]
H ["G", "J"]
I ["G", "J"]
J ["G", "H", "I"]
```



Time Complexity of Depth-First Search

```python
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A recursive function used by depth first search
        def dfs_visit(u):
            visited.add(u)
            for v in self.adj[u]:
                if v not in visited:
                    dfs_visit(v)
        # Start depth first search
        for u in self.vertices:
            if u not in visited:
                dfs_visit(u)
```

What is the time complexity of depth-first search?

Time Complexity of Depth-First Search

```
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A recursive function used by depth first search
        def dfs_visit(u):
            visited.add(u)
            for v in self.adj[u]:
                if v not in visited:
                    dfs_visit(v)
        # Start depth first search
        for u in self.vertices:
            if u not in visited:
                dfs_visit(u)
```

What is the time complexity of depth-first search?

- The function `dfs_visit()` is called exactly once for each vertex.

Time Complexity of Depth-First Search

```python
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A recursive function used by depth first search
        def dfs_visit(u):
            visited.add(u)
            for v in self.adj[u]:
                if v not in visited:
                    dfs_visit(v)
        # Start depth first search
        for u in self.vertices:
            if u not in visited:
                dfs_visit(u)
```

What is the time complexity of depth-first search?

- The function `dfs_visit()` is called exactly once for each vertex.
- During the execution of `dfs_visit()` for vertex u, the `for` loop runs in $\Theta(d(u))$ time.

Time Complexity of Depth-First Search

```python
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A recursive function used by depth first search
        def dfs_visit(u):
            visited.add(u)
            for v in self.adj[u]:
                if v not in visited:
                    dfs_visit(v)
        # Start depth first search
        for u in self.vertices:
            if u not in visited:
                dfs_visit(u)
```

What is the time complexity of depth-first search?

- The function `dfs_visit()` is called exactly once for each vertex.
- During the execution of `dfs_visit()` for vertex u, the `for` loop runs in $\Theta(d(u))$ time.
- Thus, the overall running time is $\Theta(n + m)$.

Finding Connected Components

Depth-first search can be used to find connected components.

Finding Connected Components

Depth-first search can be used to find connected components.

- When `dfs()` calls `dfs_visit()`, a connected component is found.

Finding Connected Components

Depth-first search can be used to find connected components.

- When `dfs()` calls `dfs_visit()`, a connected component is found.
- Thus, with little revision we can use `dfs()` to label the vertices with the connected component they belong to.

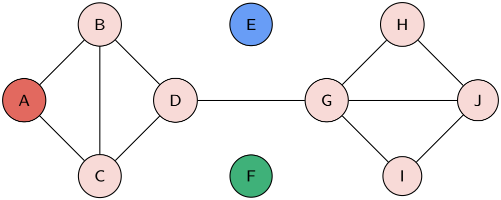
Finding Connected Components (cont.)

We use a dictionary `label` to store the representative of the connected component that a vertex belongs to.

```
class Graph:
    ...
    def dfs(self):
      visited = set()
      label = {}
      # A recursive function used by depth first search
      def dfs_visit(u, r):
          label[u] = r
          visited.add(u)
          for v in self.adj[u]:
              if v not in visited:
                  dfs_visit(v, r)
      # Start depth first search
      for u in self.vertices:
          if u not in visited:
              dfs_visit(u, u)
      return label
```

Finding Connected Components (cont.)

```
>>> label = G.dfs()
>>> for u in G.vertices:
...     print(u, label[u])
...
A A
B A
C A
D A
E E
F F
G A
H A
I A
J A
```



Conclusion

With depth-first search we can know the connected component a vertex belongs to, and thus we have the following data structure.

Conclusion

With depth-first search we can know the connected component a vertex belongs to, and thus we have the following data structure.

Adjacency List with Label

- Preprocess: $O(n + m)$ time.
- Check if u and v are connected: $O(1)$ time.

Iterative Method

In fact, we can use a **stack** to eliminate recursion!

Iterative Method

In fact, we can use a **stack** to eliminate recursion!

```
class Graph:
    ...
    def dfs(self):
        visited = set()
        # A non-recursive function used by depth first search
        def dfs_visit(u):
            stack = [u]
            while stack:
                u = stack.pop()
                if u not in visited:
                    visited.add(u)
                    for v in self.adj[u]:
                        stack.append(v)
        # Start depth first search
        for u in self.vertices:
            if u not in visited:
                dfs_visit(u)
```

Exercise #1

Consider the following problem.

Exercise #1

Consider the following problem.

Connected Component Problem

- Input: A graph G with n vertices and m edges.
- Output: The number of connected components of G.

Exercise #1

Consider the following problem.

Connected Component Problem

- Input: A graph G with n vertices and m edges.
- Output: The number of connected components of G.

What is the time complexity of the connected component problem?

- $\Theta(n+m)$
- $\Theta(n^2)$
- $\Theta((n+m)\log n)$
- $\Theta(2^n)$

Exercise #1

Consider the following problem.

Connected Component Problem

- Input: A graph G with n vertices and m edges.
- Output: The number of connected components of G.

What is the time complexity of the connected component problem?

- $\Theta(n+m)$
- $\Theta(n^2)$
- $\Theta((n+m)\log n)$
- $\Theta(2^n)$

Solution

An $O(n+m)$-time depth-first search can solve the connected component problem. Thus, its time complexity is $\Theta(n+m)$ since the input size is $\Omega(n+m)$.

Exercise #2

Let $G = (V, E)$ with n vertices and m edges. Let u, v, w be vertices in G.

Exercise #2

Let $G = (V, E)$ with n vertices and m edges. Let u, v, w be vertices in G. Which of the following statements is true?

Exercise #2

Let $G = (V, E)$ with n vertices and m edges. Let u, v, w be vertices in G. Which of the following statements is true?

- A vertex can have at most n neighbors.

Exercise #2

Let $G = (V, E)$ with n vertices and m edges. Let u, v, w be vertices in G. Which of the following statements is true?

- A vertex can have at most n neighbors.
- m should not be less than n.

Exercise #2

Let $G = (V, E)$ with n vertices and m edges. Let u, v, w be vertices in G. Which of the following statements is true?

- A vertex can have at most n neighbors.
- m should not be less than n.
- If u and v are adjacent and v and w are adjacent, then u and w are adjacent.

Exercise #2

Let $G = (V, E)$ with n vertices and m edges. Let u, v, w be vertices in G. Which of the following statements is true?

- A vertex can have at most n neighbors.
- m should not be less than n.
- If u and v are adjacent and v and w are adjacent, then u and w are adjacent.
- If u and v are connected and v and w are connected, then u and w are connected.

Exercise #2

Let $G = (V, E)$ with n vertices and m edges. Let u, v, w be vertices in G. Which of the following statements is true?

- A vertex can have at most n neighbors.
- m should not be less than n.
- If u and v are adjacent and v and w are adjacent, then u and w are adjacent.
- If u and v are connected and v and w are connected, then u and w are connected.

Solution

Only the last statement is correct.

Exercise #3

Let $G = (V, E)$ with $|V| = n$ and $|E| = m$.

Exercise #3

Let $G = (V, E)$ with $|V| = n$ and $|E| = m$. Furthermore, let A and C be $n \times n$ matrices such that

$$A_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are adjacent} \\ 0, & \text{otherwise} \end{cases}$$

and

$$C_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are connected} \\ 0, & \text{otherwise.} \end{cases}$$

Exercise #3

Let $G = (V, E)$ with $|V| = n$ and $|E| = m$. Furthermore, let A and C be $n \times n$ matrices such that

$$A_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are adjacent} \\ 0, & \text{otherwise} \end{cases}$$

and

$$C_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are connected} \\ 0, & \text{otherwise.} \end{cases}$$

Let $B = I + A + A^2 + \cdots + A^{n-2} + A^{n-1}$.
What is the relation between B and C?

Exercise #3

Let $G = (V, E)$ with $|V| = n$ and $|E| = m$. Furthermore, let A and C be $n \times n$ matrices such that

$$A_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are adjacent} \\ 0, & \text{otherwise} \end{cases}$$

and

$$C_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are connected} \\ 0, & \text{otherwise.} \end{cases}$$

Let $B = I + A + A^2 + \cdots + A^{n-2} + A^{n-1}$.
What is the relation between B and C?

Solution

$C_{ij} = 1$ if and only if $B_{ij} \geq 1$.