Advanced Programming
INFO135

Lecture 7: Graph

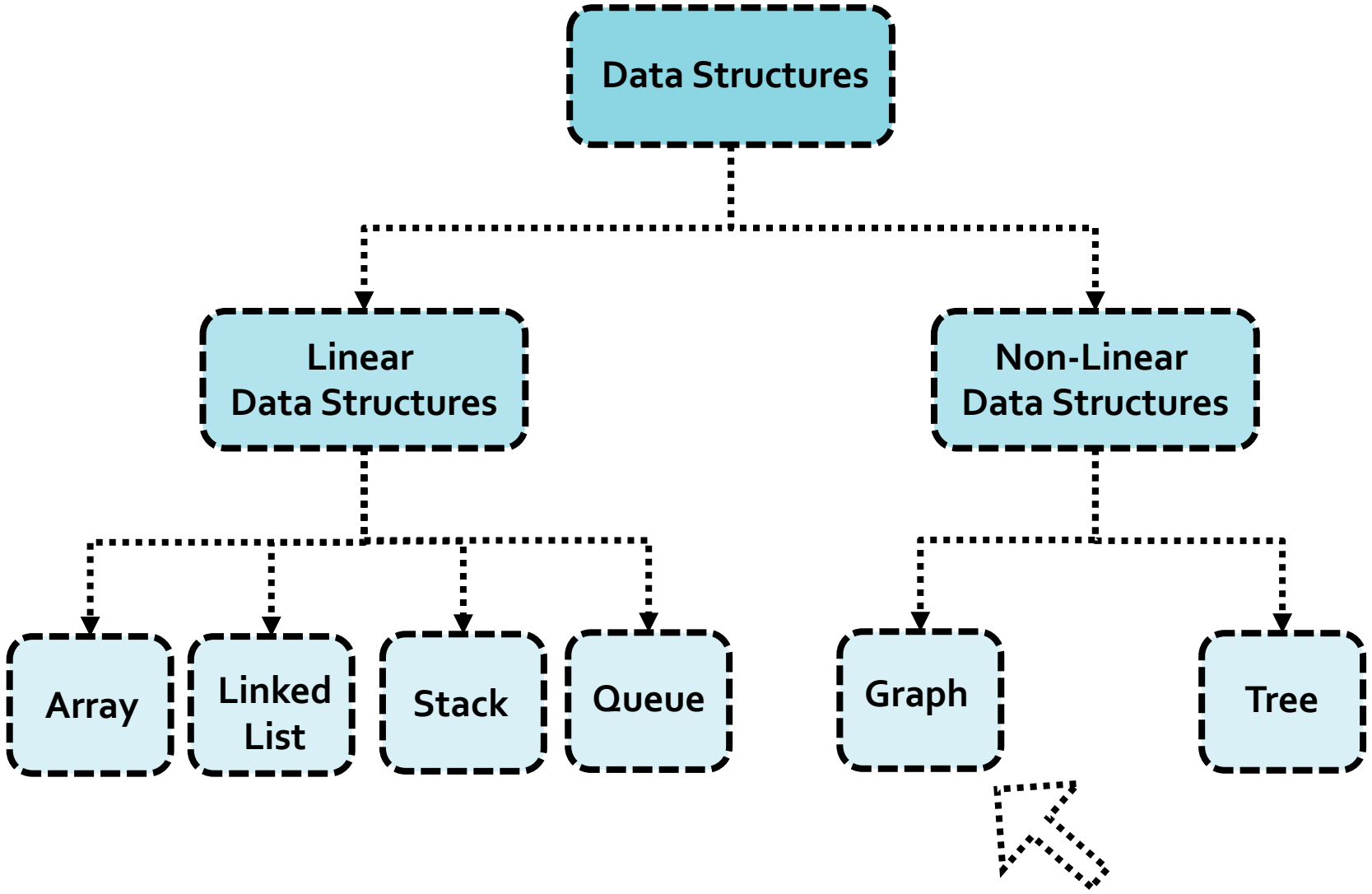
Mehdi Elahi

University of Bergen (UiB)

Data Structures

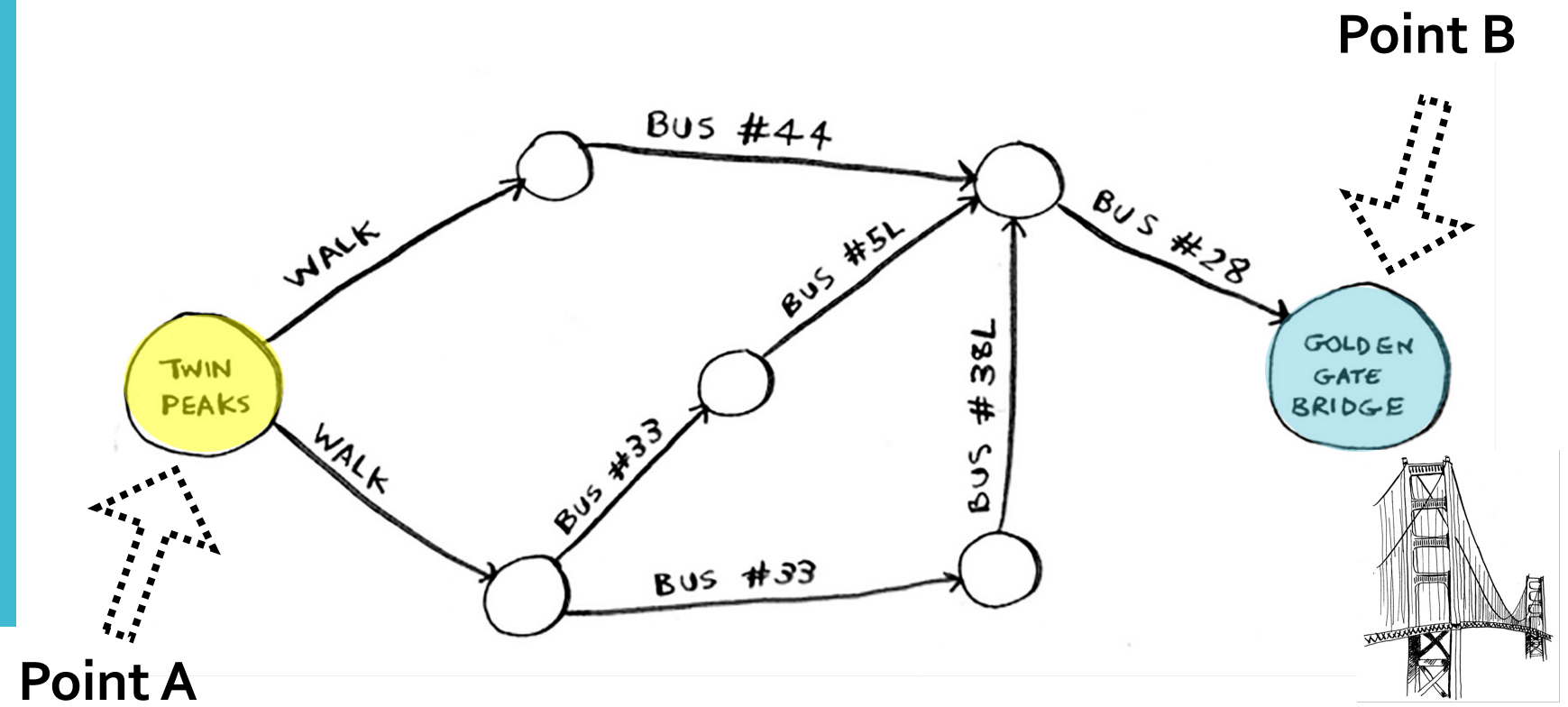
Linear Data Structure	Non-linear Data Structure
Data elements are arranged in a linear order where each element is attached to its **previous and next adjacent**.	Data elements are attached in **hierarchically** manner.
Single level is involved.	**Multiple levels** are involved.
Simpler and easier to implement	More **complex** and difficult to implement.
Data elements **can be traversed** in a **single run**.	Data elements **can't be traversed** in a **single run**.

Data Structures



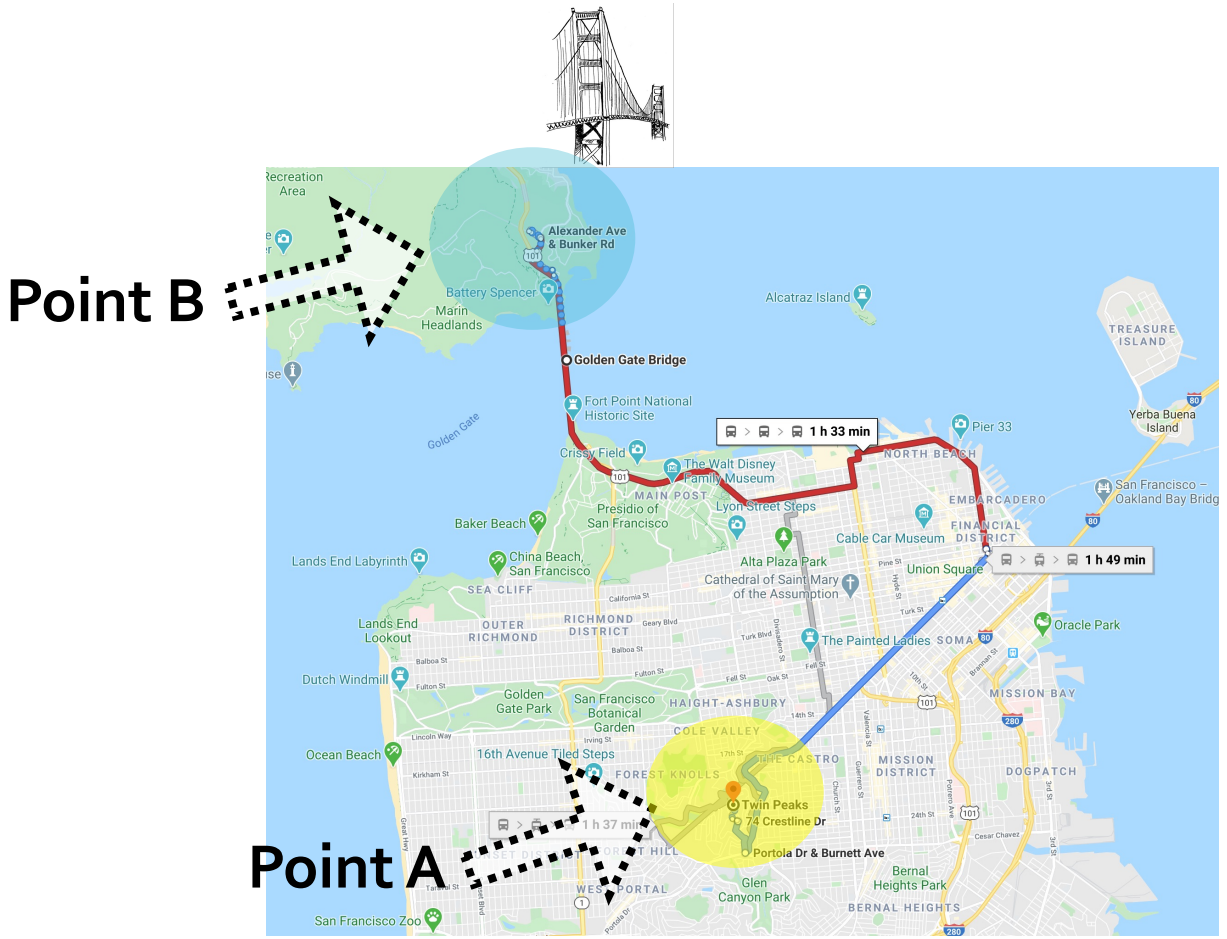
Graphs

- Suppose you're in San Francisco, going from **point A** (Twin Peaks) to **point B** (Golden Gate Bridge).
- You want to get there by bus, with the **fewest steps** (transfers).



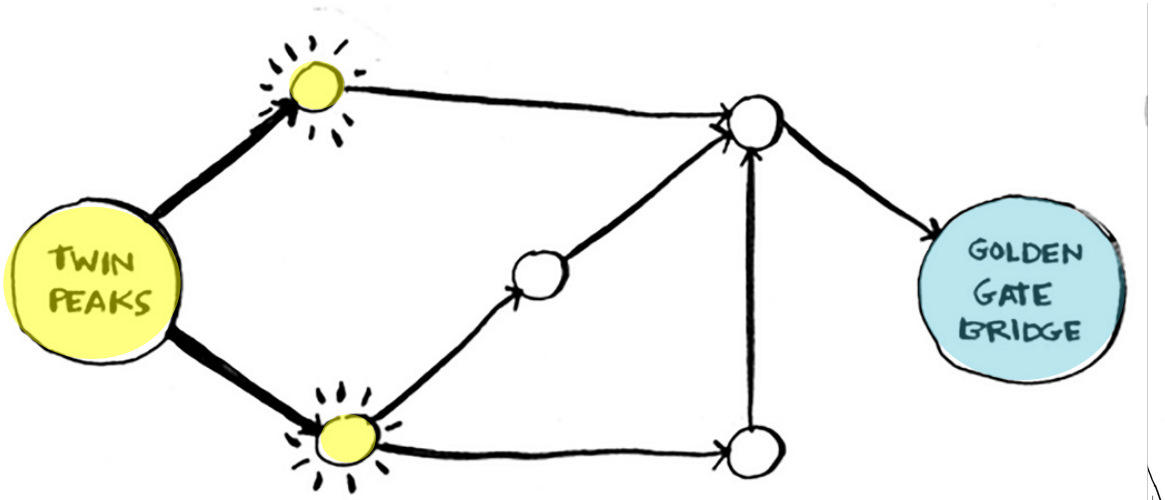
Graphs

- In Google map this will look like the following.



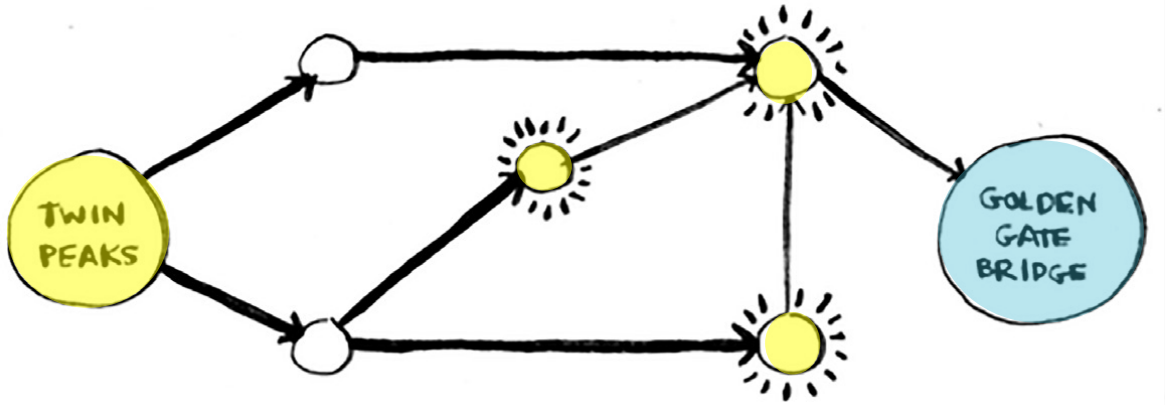
Graphs

- Can you describe an algorithm to find the **path with the fewest steps**?
- Lets check: Start from Twin Peaks with one step.
- You **can't reach to bridge** in one step.



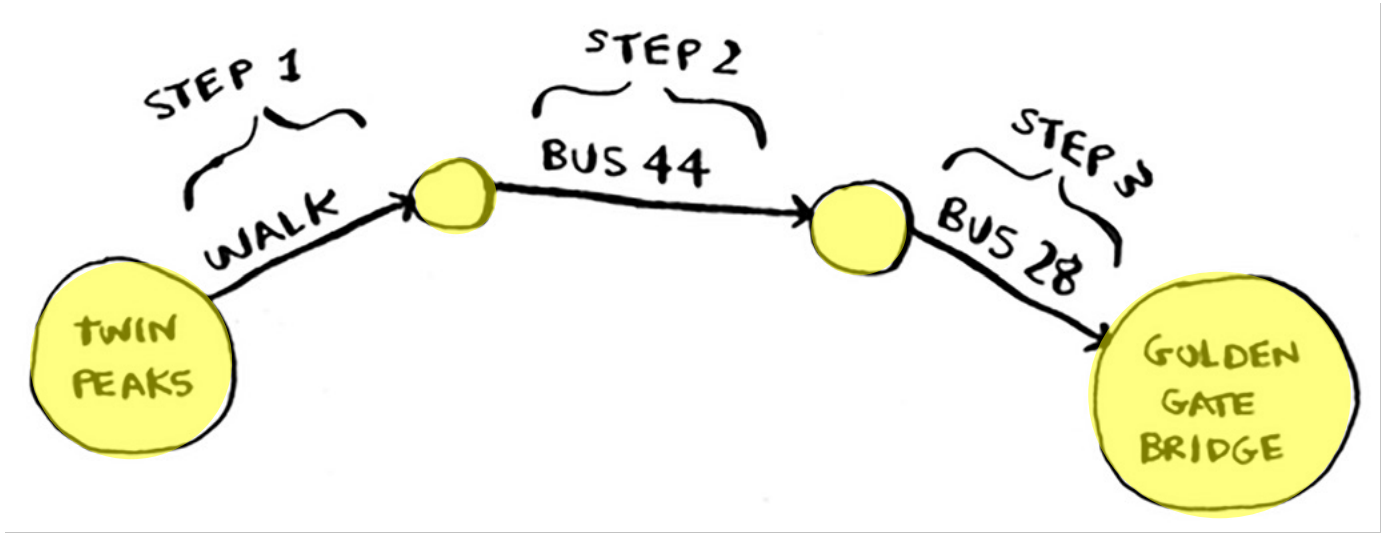
Graphs

- What about **two steps**? **No**, you can't.
- What about **three steps**?



Graphs

- Now the **Golden Gate Bridge** shows up.
- So it takes minimum **three steps** to get from Twin Peaks to the bridge, with this rout.



Graphs

- There are other routes that will get you to the bridge too, but **they're longer** (four steps).
- This problem is known as **Shortest-path Problem**:
  - the shortest route to your friend's house.
  - the smallest number of moves to checkmate in a game of chess.
- Algorithms that can find the path in a Graph:
  - **Breadth-First Search (BFS)**
  - **Depth-First Search (DFS)**

Graphs

- We address the **shortest path problem** by:
  - **modelling** the problem as a **Graph**.
  - **solving** it by using algorithms such as **BFS** or **DFS**.

- Q: But what is a **Graph**?
- A: It is a structure that models a set of points (nodes or vertices) and their connections (edges).

Graphs

Examples of **Graphs.**

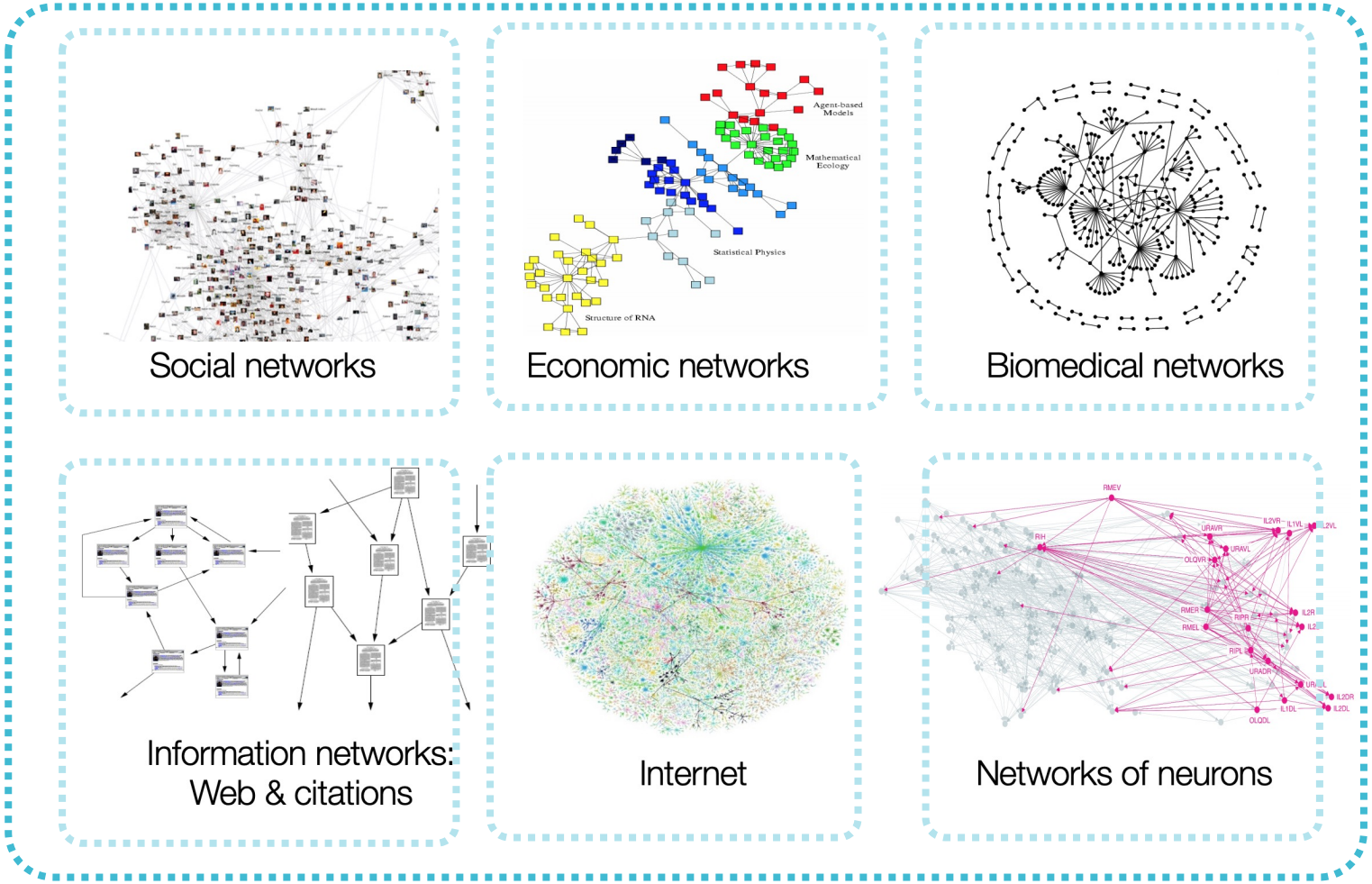


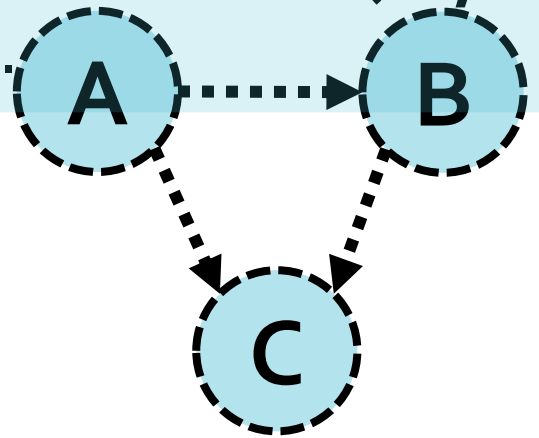
Image is taken from William Hamilton COMP551: Graph Representation Learning

Graphs

- A Graph is built by two sets, i.e.:
  - **Vertices (V) set** also called nodes or points
  - **Edges (E) set** also called links, connections, or arcs

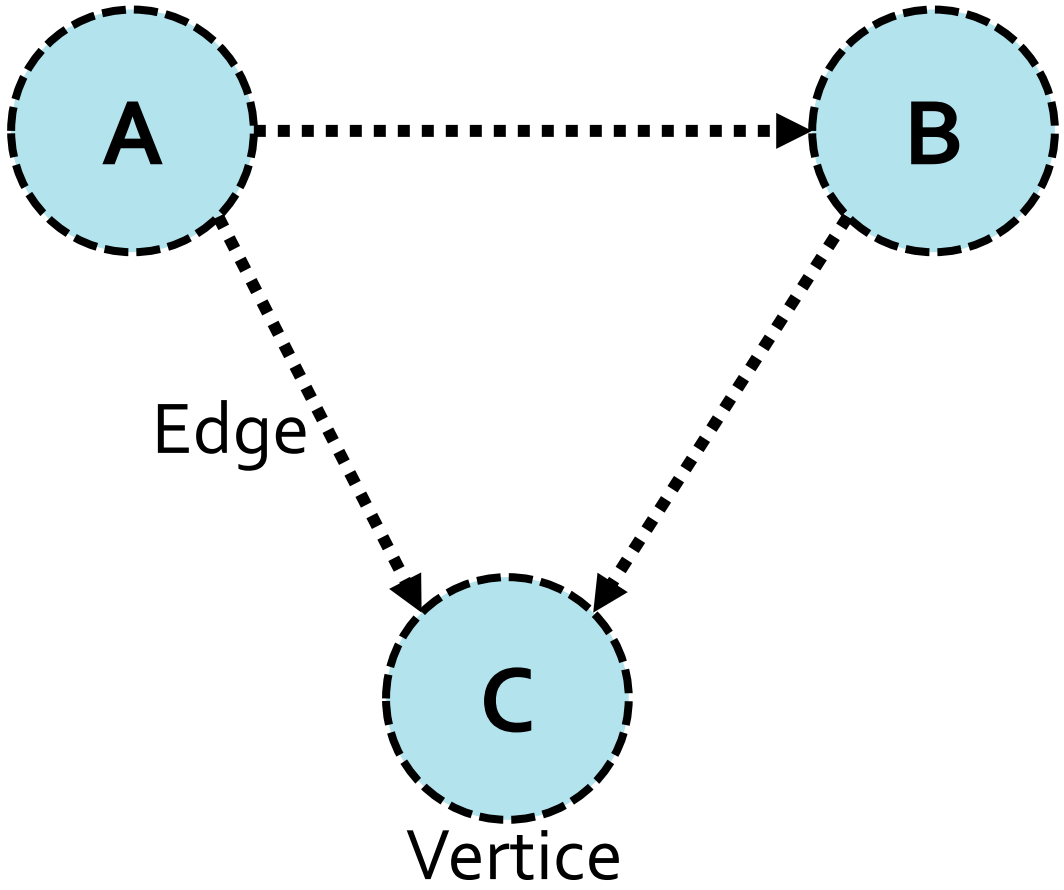
- Graphs are often shown:
  - Vertices set as boxes (or circles)
  - Edge set as lines or arcs between the boxes (or circles).

- There is an edge between v1 and v2 if **(v1,v2) is an element of the Edge set**.



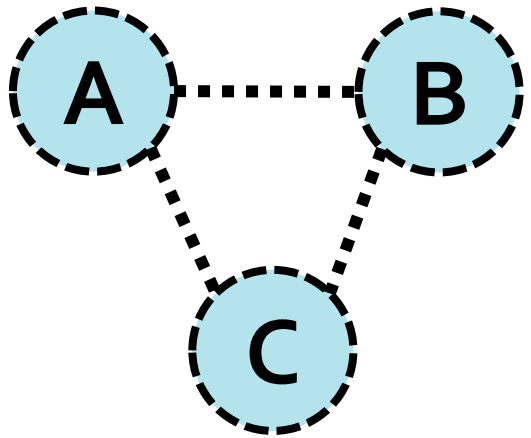
Graphs

- Example:
  - The Vertices (nodes) set = {A,B,C}
  - The Edges (links) set = {(A,B),(B,C),(A,C}



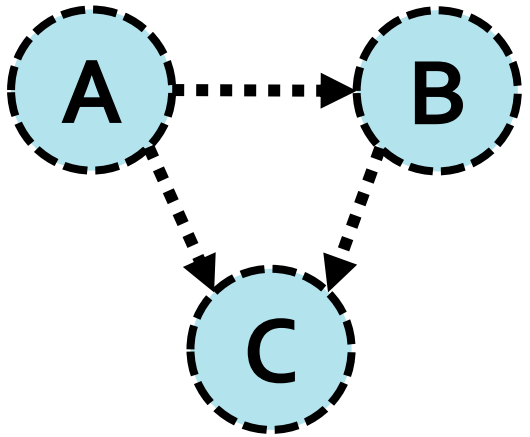
Graphs

- **Undirected Graphs**
  - is a graph in which **edges** does **not have orientations**.
  - **order** of the vertices in the pairs in the edge set **does no matter**.
  - We draw a straight line between the vertices.
  - Adjacency relation is **symmetric** in undirected graph.



Graphs

- **Directed Graphs.**
  - is a graph in which **edges have orientations**.
  - **order** of the vertices in the pairs in the edge set **matters**.
  - We use arrows for the arcs between vertices.
  - Adjacency relation is **asymmetric** in directed graph.



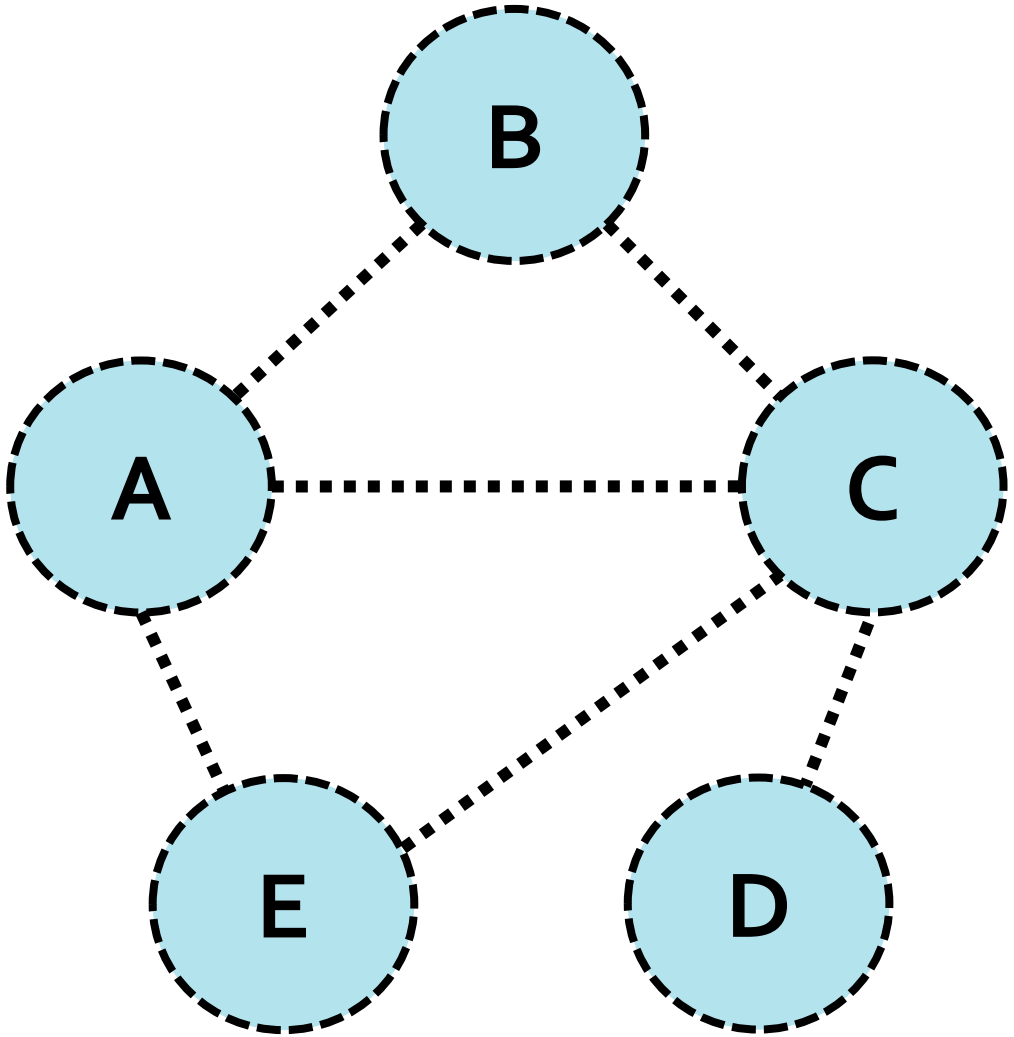
Graphs

- How to represent **Graphs**?

- There can be different methods. Two methods :
  - List (called **Adjacency List**)
  - Matrix (called **Adjacency Matrix**)

- Lets see examples of both.
- First consider **undirected** Graphs.

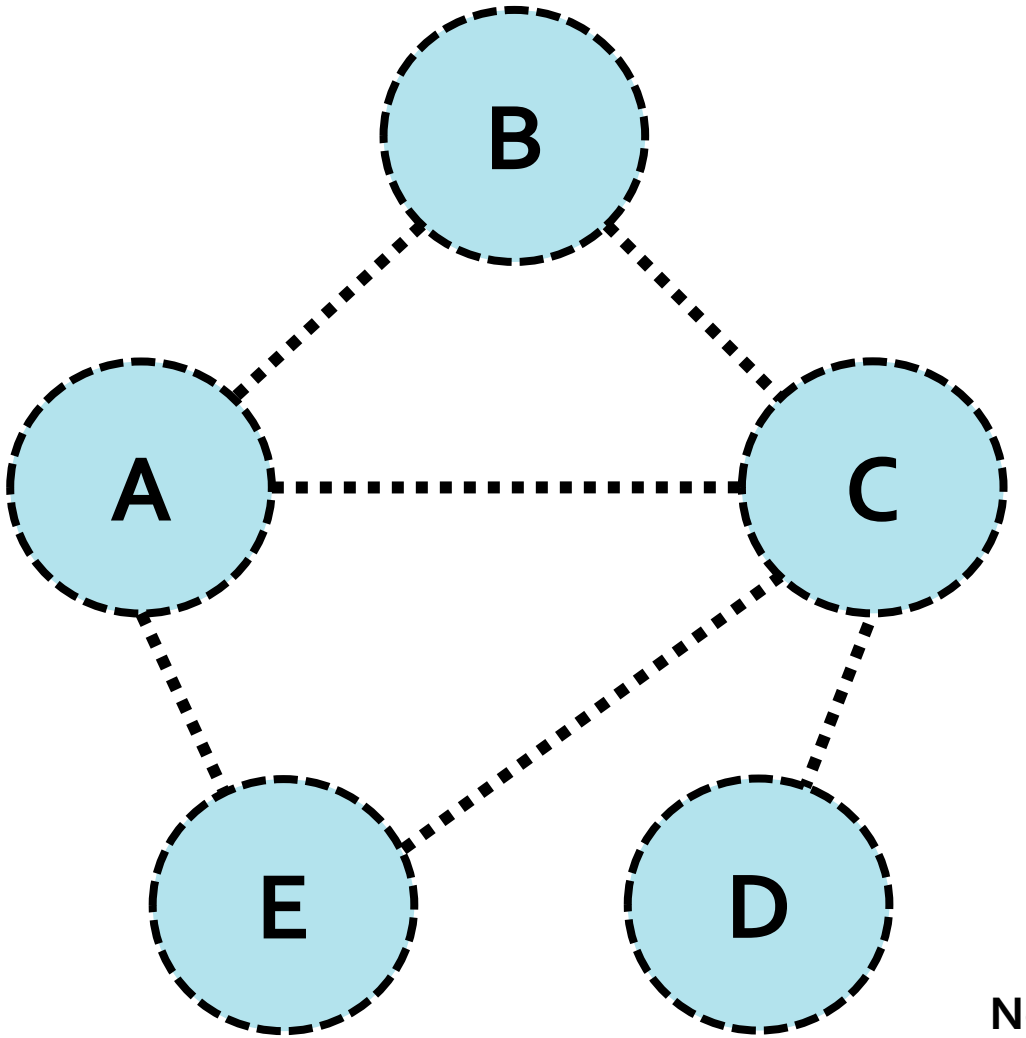
Undirected Graphs



Adjacency **List**

Ref: https://github.com/joeyajames

Undirected Graphs



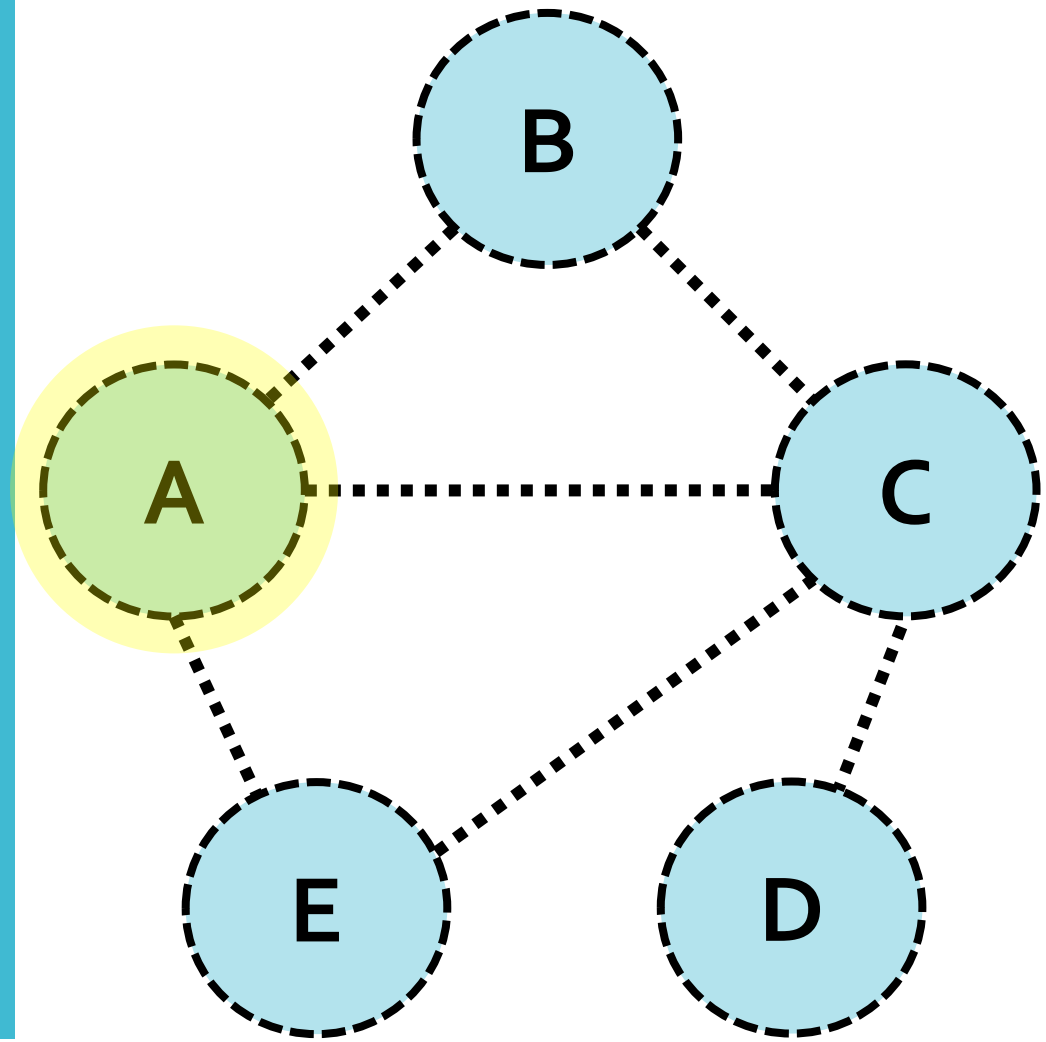
Adjacency **List**

A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C

Note: adjacency list uses O(|E|) space

Ref: https://github.com/joeyajames

Undirected Graphs

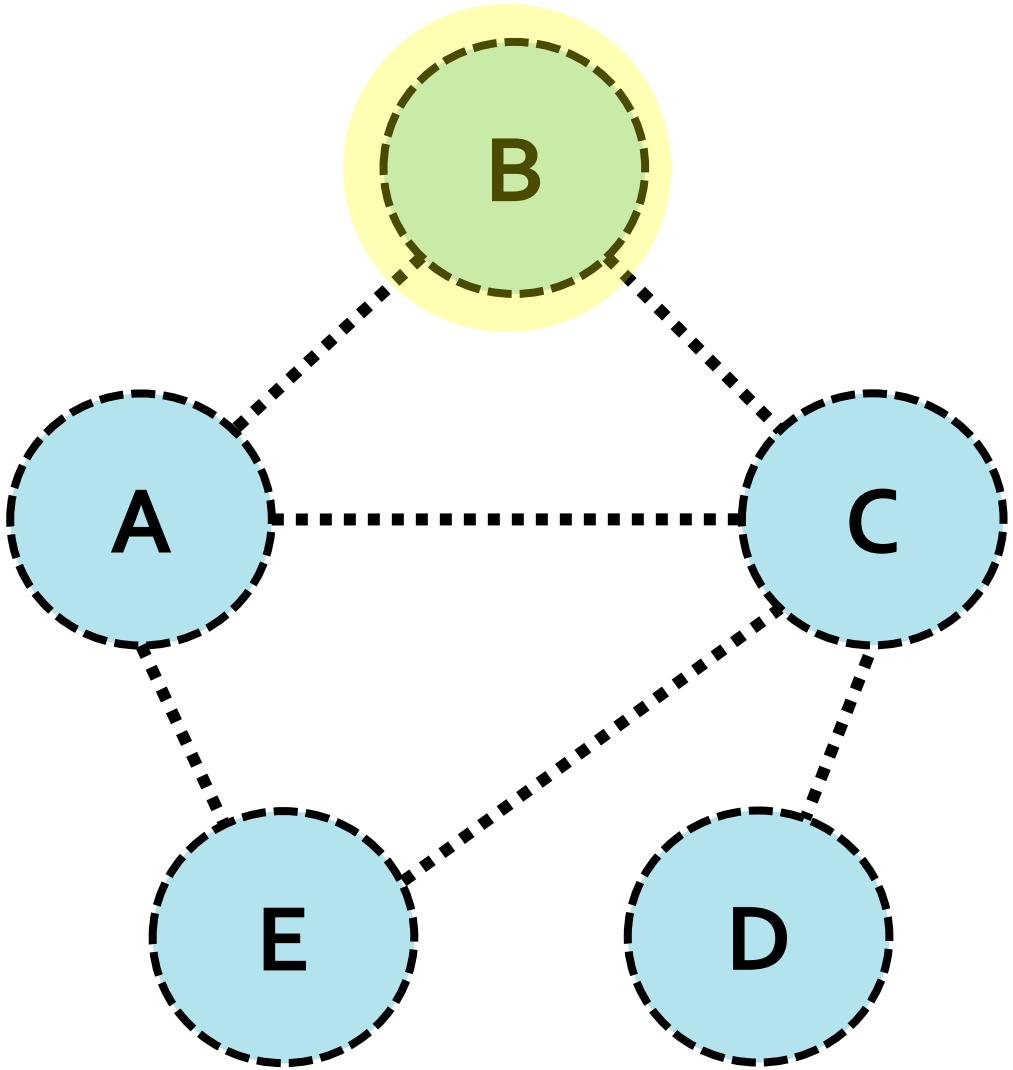


Adjacency **List**

A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C

Ref: https://github.com/joeyajames

Undirected Graphs



Adjacency **List**

A: B, C, E

B: A, C

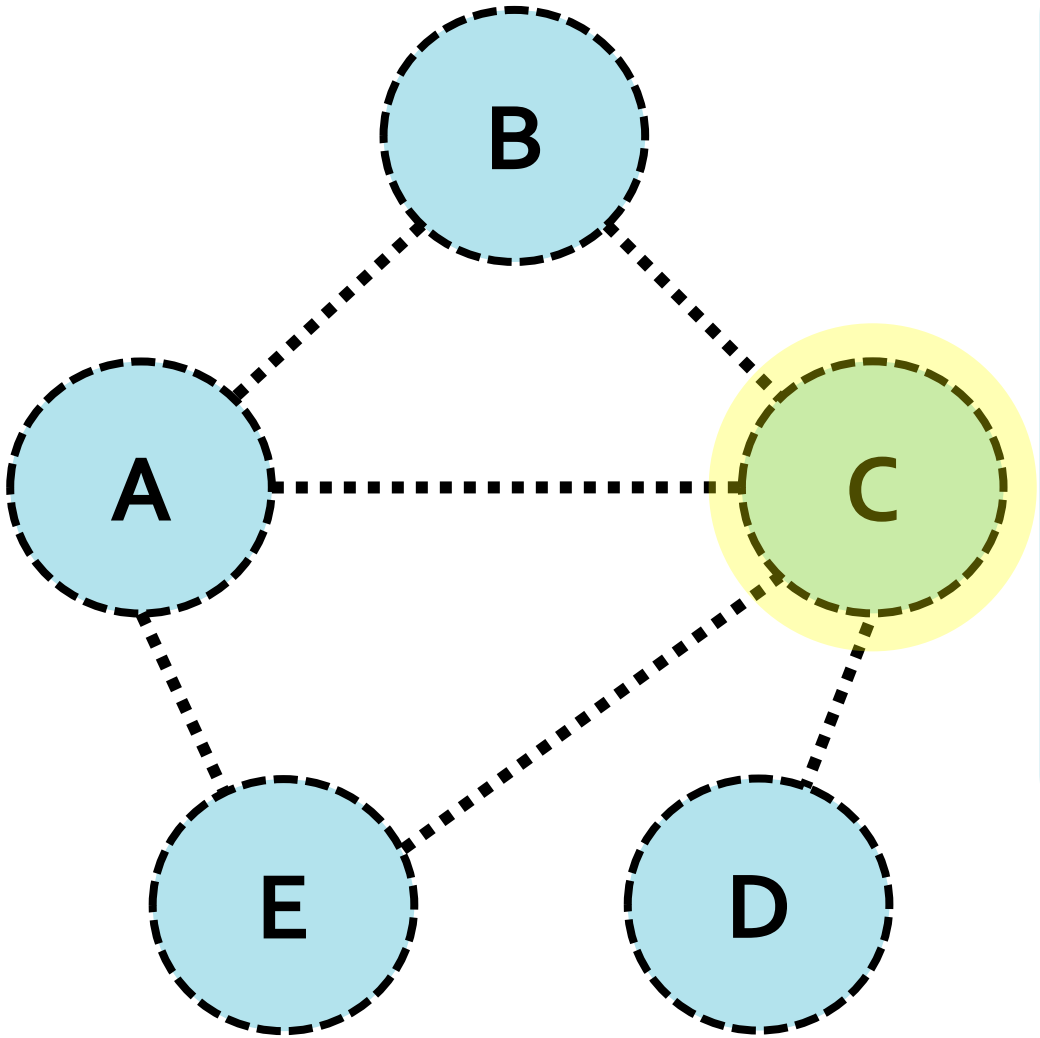
C: A, B, D, E

D: C

E: A, C

Ref: https://github.com/joeyajames

Undirected Graphs

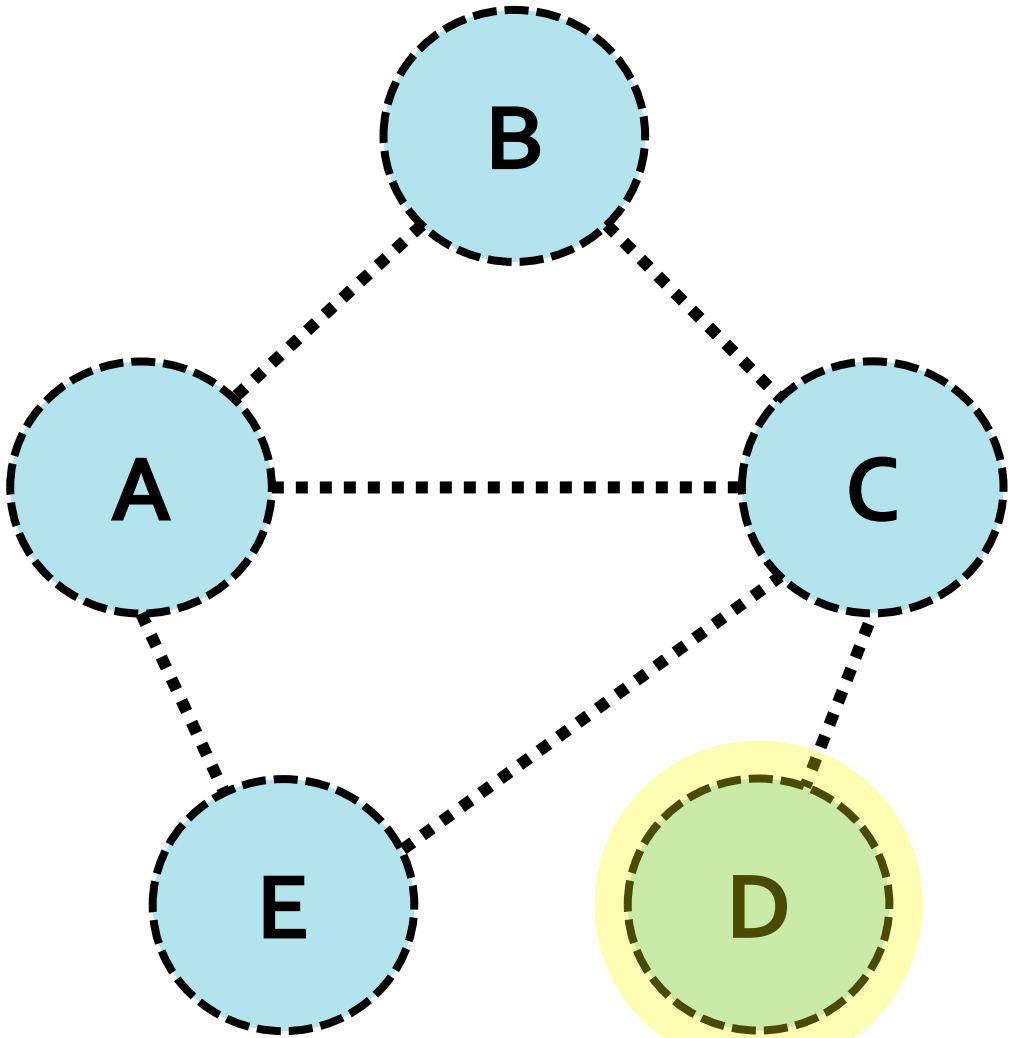


Adjacency **List**

A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C

Ref: https://github.com/joeyajames

Undirected Graphs

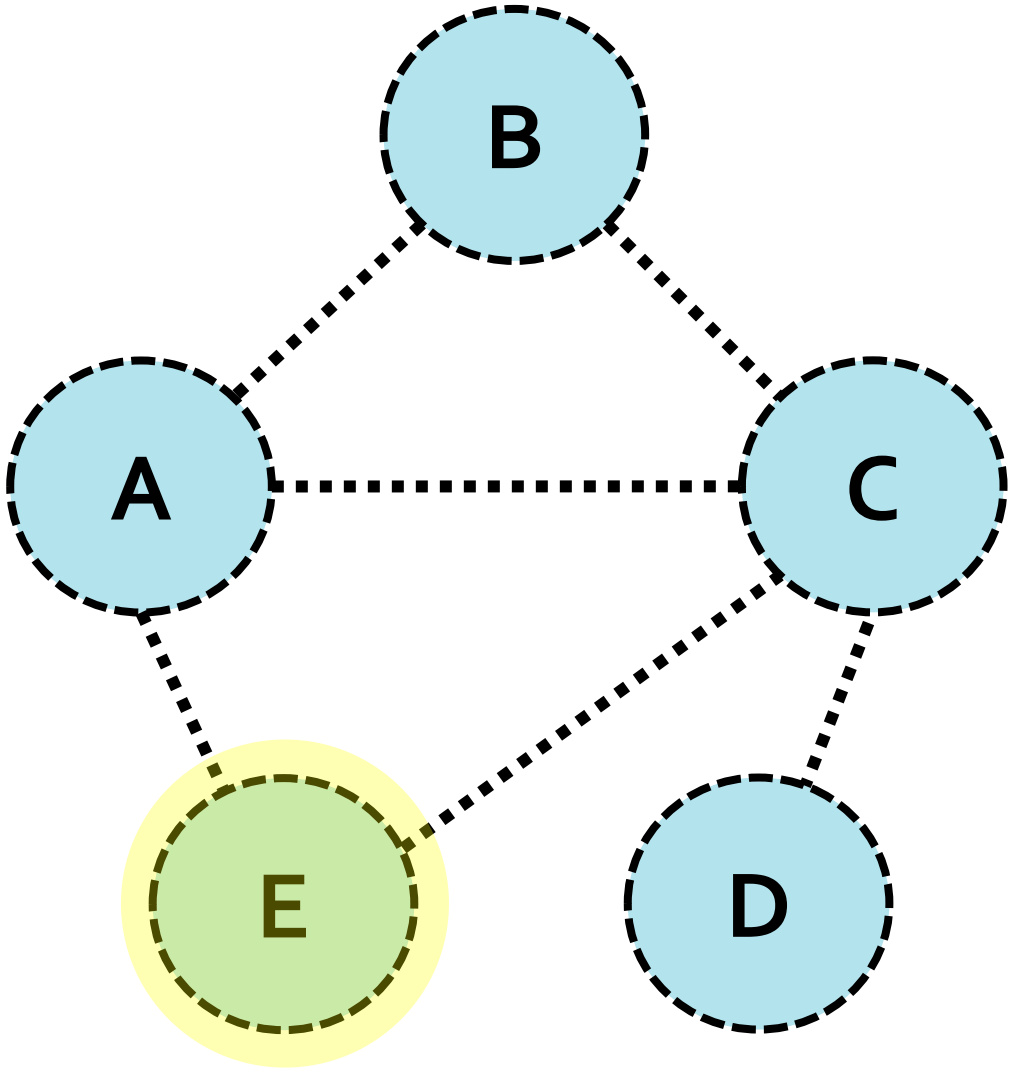


Adjacency **List**

A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C

Ref: https://github.com/joeyajames

Undirected Graphs

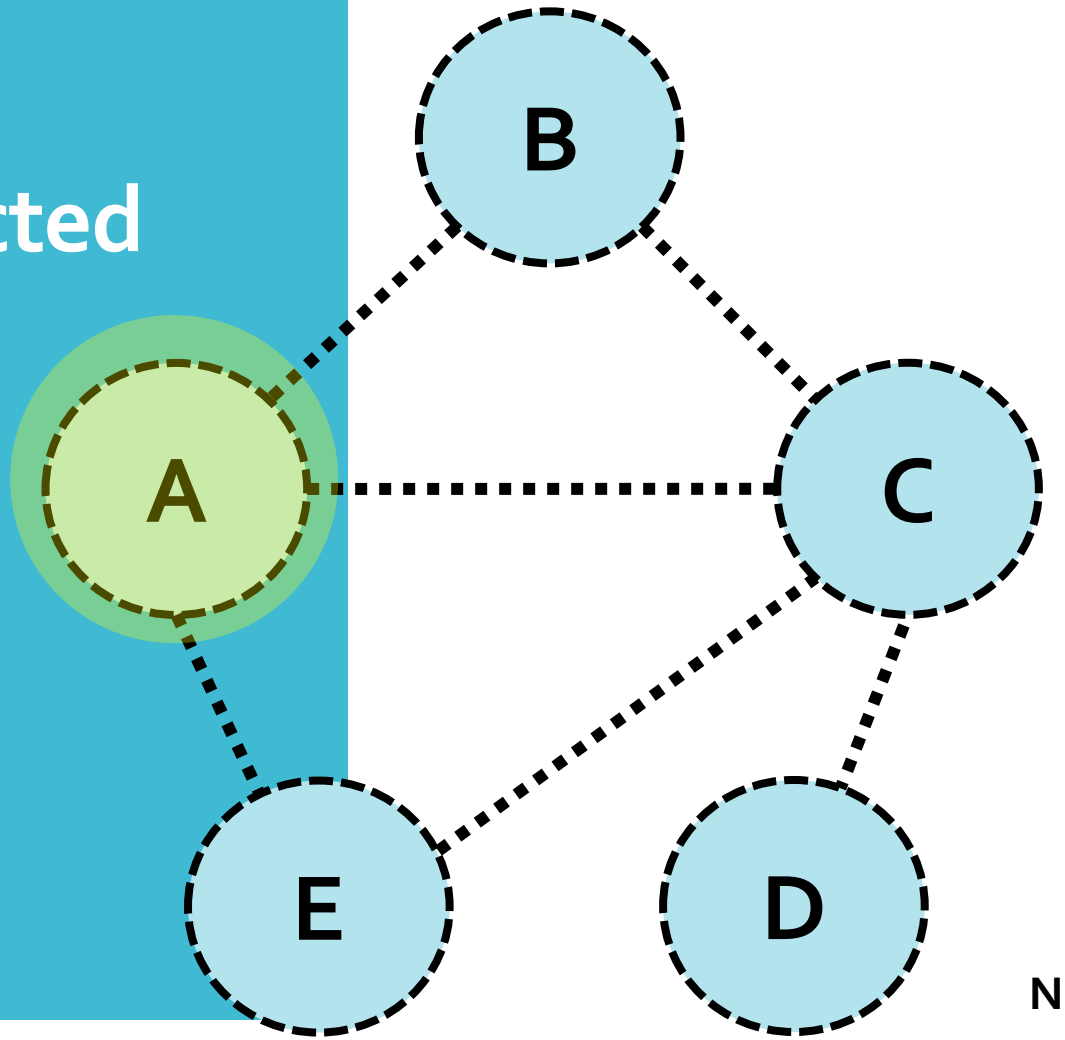


Adjacency **List**

A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C

Ref: https://github.com/joeyajames

Undirected Graphs



Adjacency **Matrix**

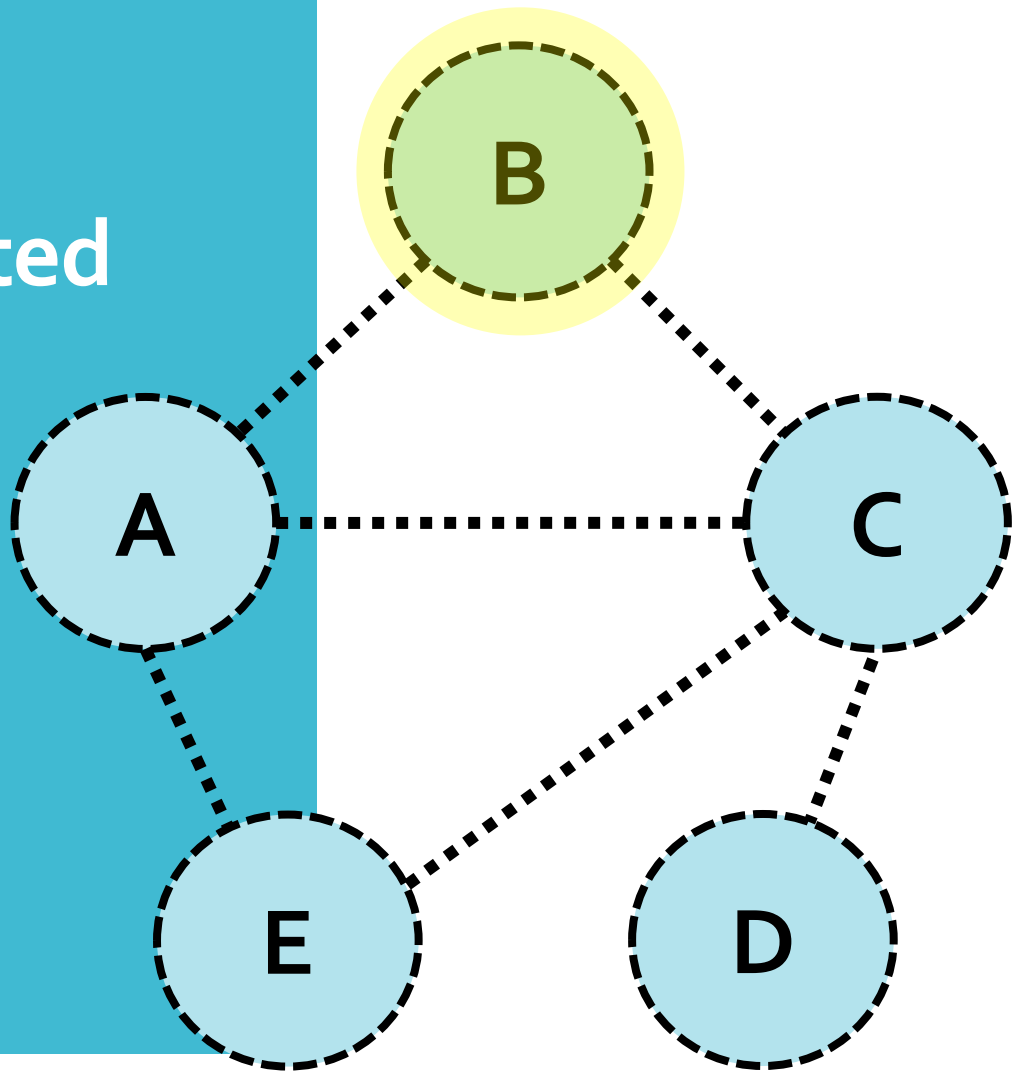
from / **to**

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	0
E	1	0	1	0	0

Note: matrix can be implemented as a 2D Python List with |V| x |V| cells

Ref: https://github.com/joeyajames

Undirected Graphs



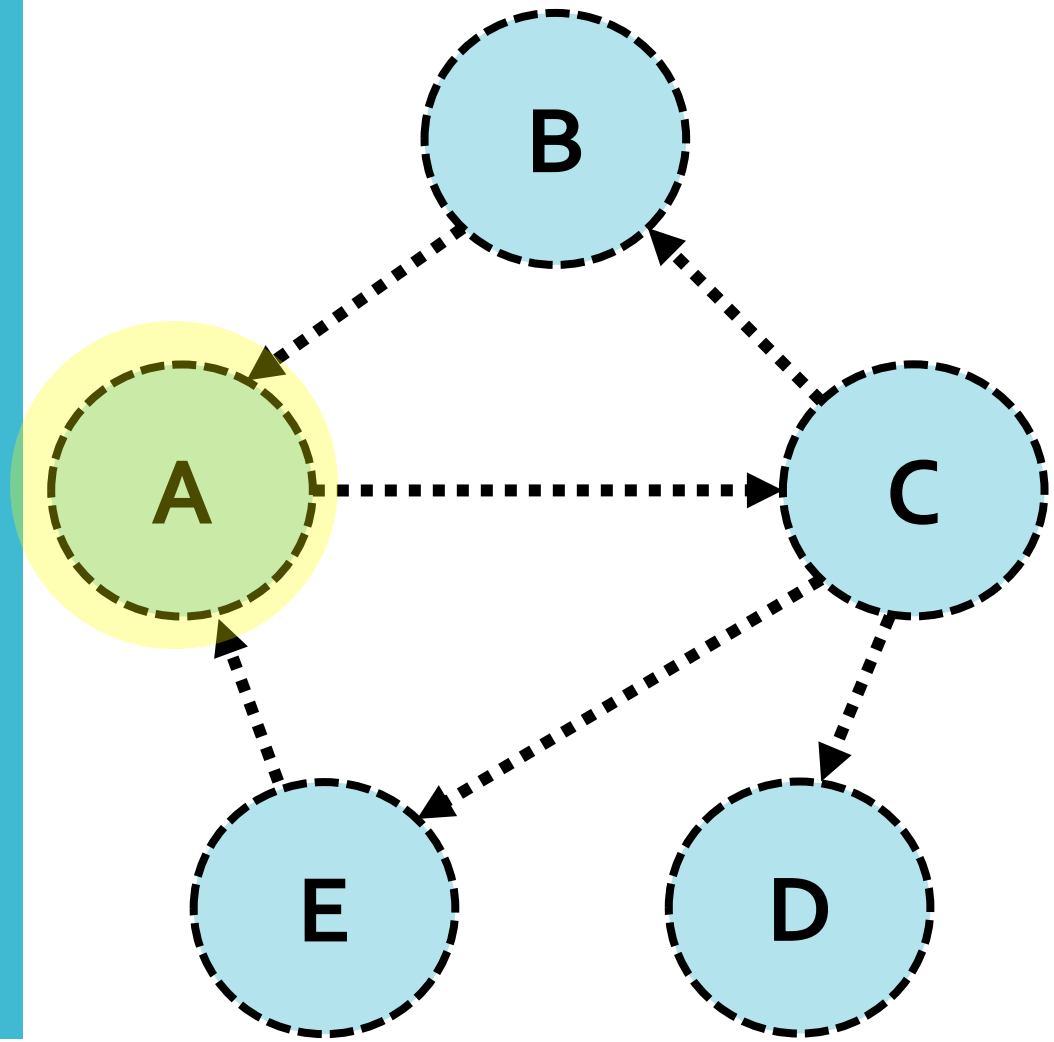
Adjacency **Matrix**

from / **to**

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	0
E	1	0	1	0	0

Ref: https://github.com/joeyajames

Directed Graphs

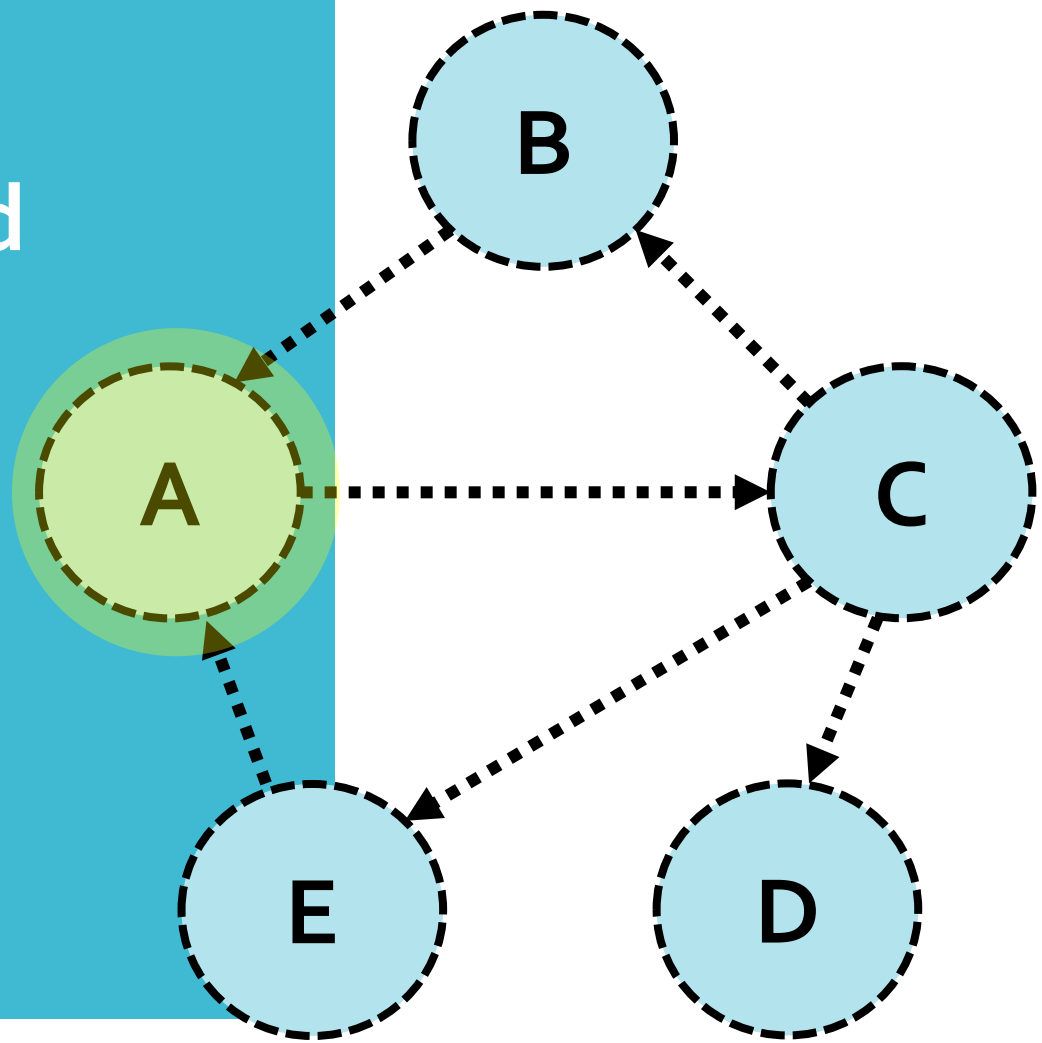


Adjacency **List**

A: C
B: A
C: B, D, E
D:
E: A

Ref: https://github.com/joeyajames

Directed Graphs



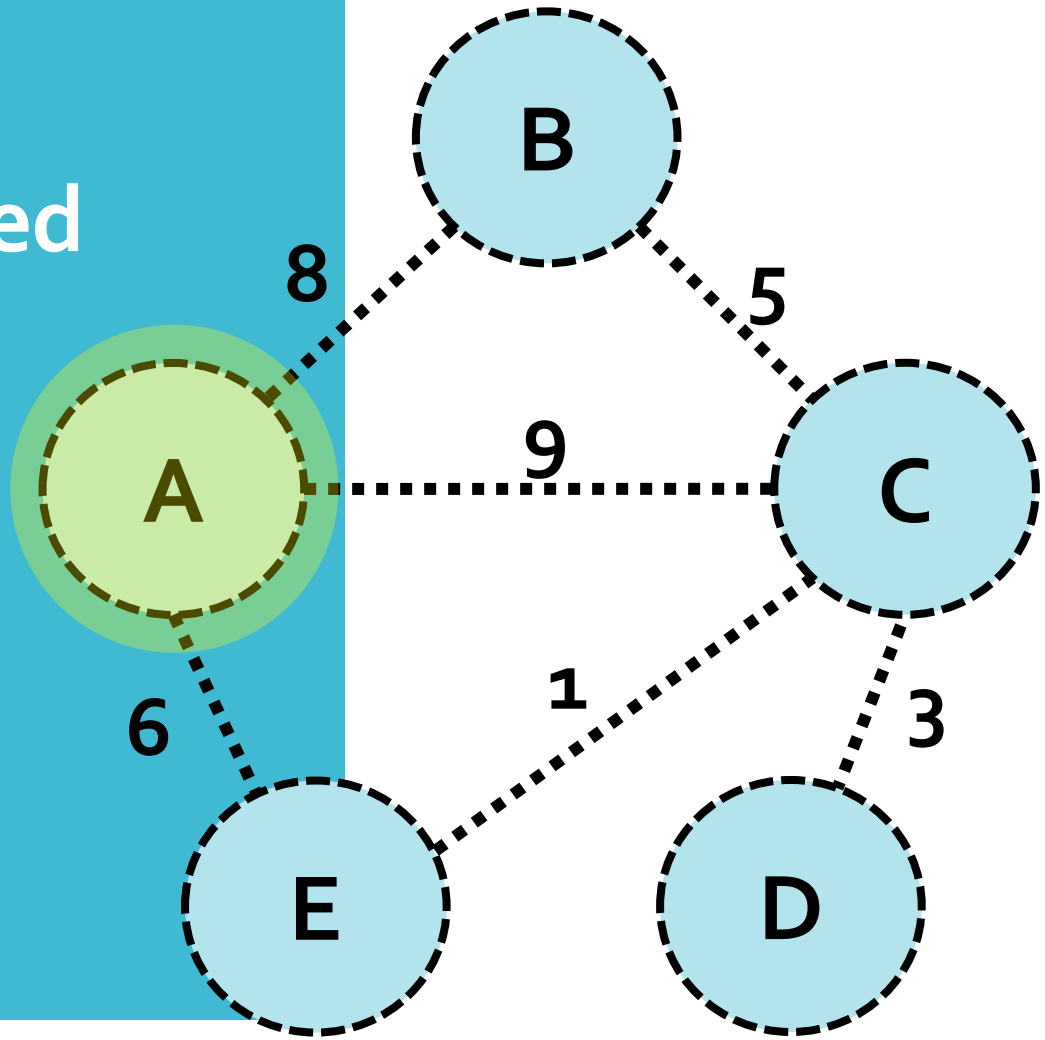
Adjacency **Matrix**

from \ **to**

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	0
C	0	1	0	1	1
D	0	0	0	0	0
E	1	0	0	0	0

Ref: https://github.com/joeyajames

Weighted Graphs



These can be used to represent also **other types** of Graphs

to

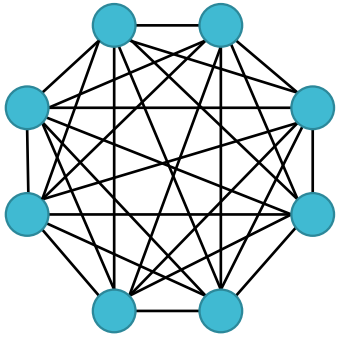
from	A	B	C	D	E
A	0	8	9	0	6
B	8	0	5	0	0
C	9	5	0	3	1
D	0	0	3	0	0
E	6	0	1	0	0

Ref: https://github.com/joeyajames

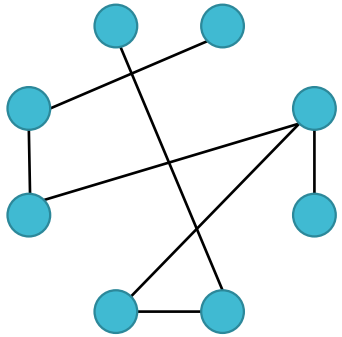
Quiz

❖ Consider the **following graphs**. Which Graph representation better suits to each of them?

- **a) Dense** Graph:
  graph where $|E| = |V|^2$

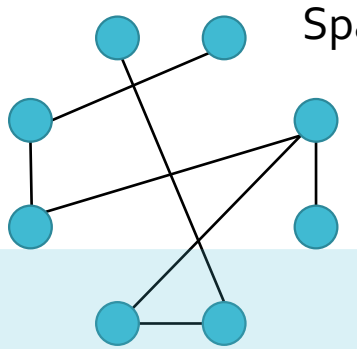


- **b) Sparse** Graph:
  graph where $|E| = |V|$



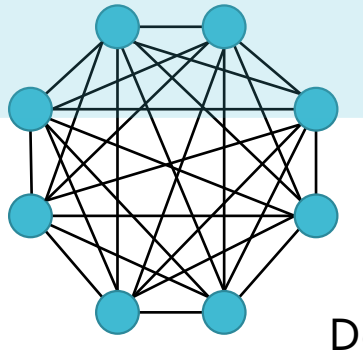
Ref: https://github.com/joeyajames

Answer



- **Adjacency List**
  - **Pro:** Faster (& uses less space) for Sparse graphs.
  - **Con:** Slower for Dense graphs

- **Adjacency Matrix**
  - **Pro:** Faster for Dense graphs.
  - **Con:** Uses more space.

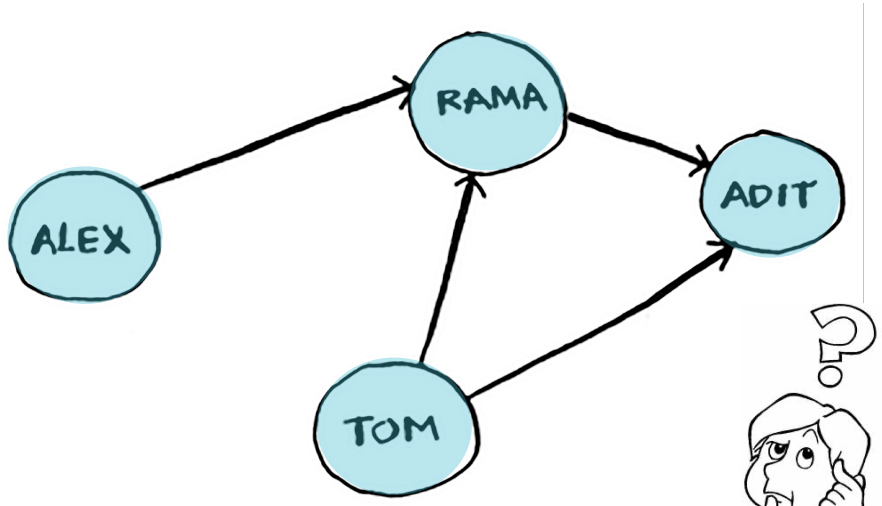


Answer

- **Adjacency Matrix** takes up $|V|^2$ space, regardless of how dense the graph is.
- Matrix for a graph with **10,000 vertices** will take up at least 100,000,000 Bytes!

Quiz

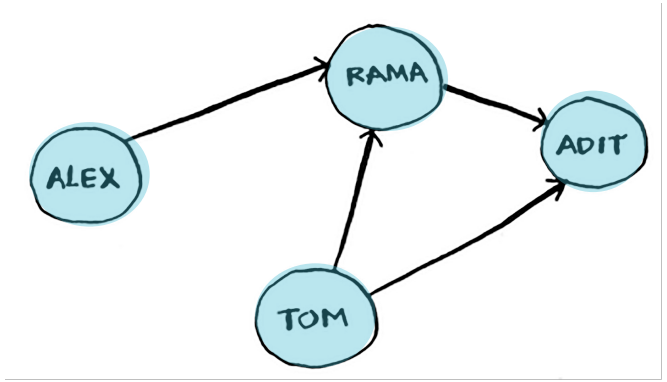
- Suppose the following Graph is given representing **who owes money to whom**.
- Implement a **Graph class** with Python dictionary and create the graph
- The class should have at least the following methods:
  - `add_edge()`
  - `print_graph()`
  - `Print_edges()`



Answer

- This is an implement of a **Graph** class.

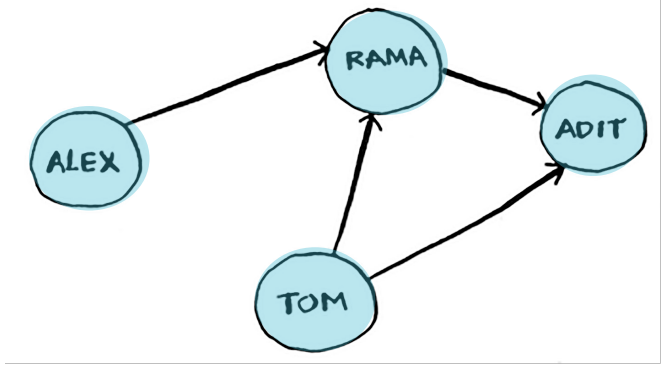
```
class Graph:
    graph = dict()

    def add_edge(self, node, neighbour):
        if node not in self.graph:
            self.graph[node] = [neighbour]
        else:
            self.graph[node].append(neighbour)

    def print_graph(self):
        print(self.graph)
```



Answer

❖ This is an implement of a **Graph** class.

```
my_graph = Graph()

my_graph.add_edge('Alex', 'Rama')
my_graph.add_edge('Rama', 'Adit')
my_graph.add_edge('Tom', 'Rama')
my_graph.add_edge('Tom', 'Adit')

my_graph.print_graph()
```



[Output:]

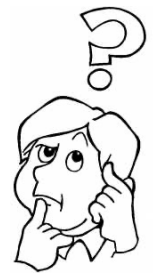
```
{'Alex': ['Rama'], 'Rama': ['Adit'], 'Tom': ['Rama', 'Adit']}
```

Answer

```
def print_edges(self):
    for node in self.graph:
        for neighbour in self.graph[node]:
            print("(", node, ",", neighbour, ")")


my_graph.print_edges()
```

[Output:]

```
( Alex , Rama )
( Rama , Adit )
( Tom , Rama )
( Tom , Adit )
```



Graph Traverse

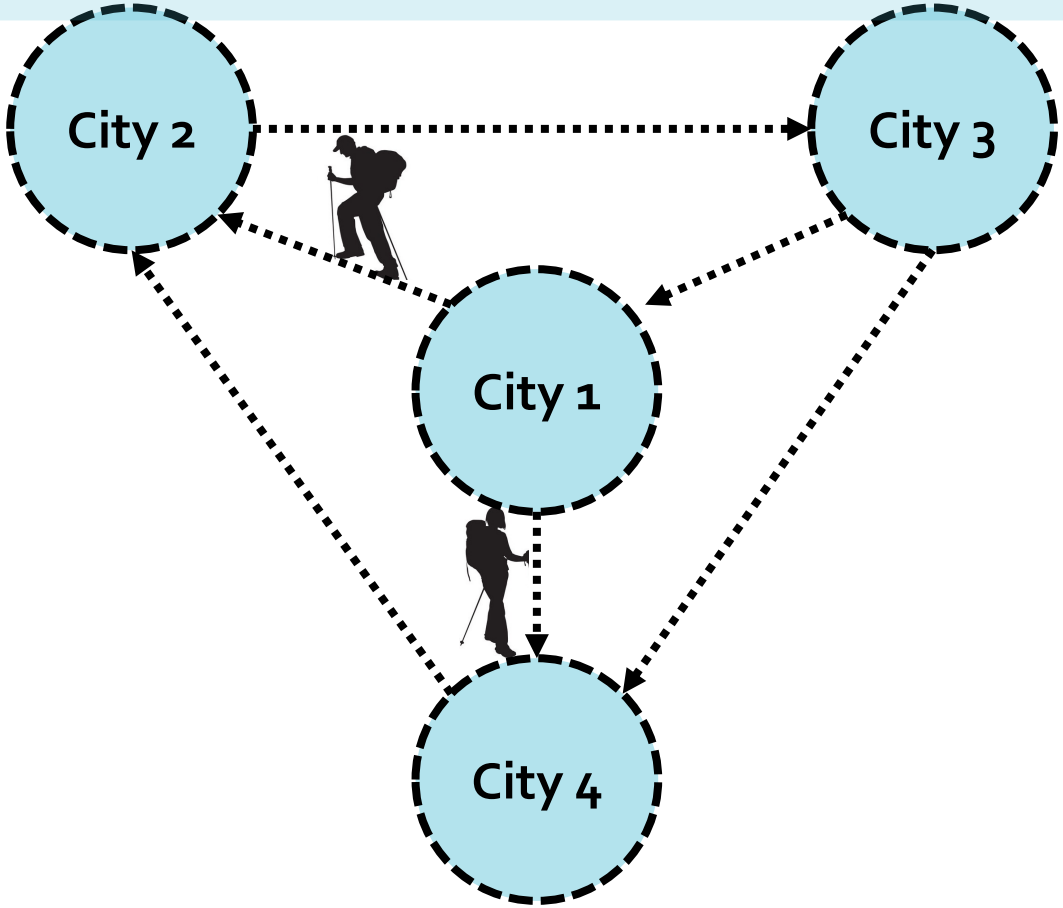
- **Graph Traverse:** is a process for navigating through a graph by examining the vertices (nodes) and edges (links).

- Graph traversal algorithms are key for solving many **fundamental problems** that can be modelled by graphs.

- There are different traverse algorithms. Here are the two popular algorithms:
  - **Breadth-First Search (BFS)**
  - **Depth-First Search (DFS)**

BFS

- **Breadth-First Search (BFS)** is a traverse algorithm that proceeds in rounds and subdivides the vertices into levels.
- **Breadth-First Search (BFS)** looks like and algorithm that sends out many explorers in all directions. They explore the graph in coordinated fashion.



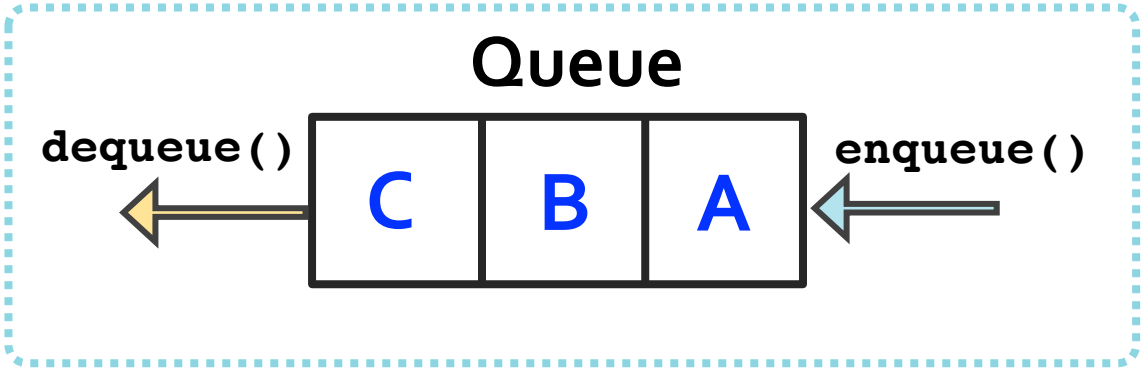
BFS

- **Breadth-First Search (BFS)** algorithm:
  - Starts at a node (vertex): ***Level 0 checked!***
    - In the 1st round, visits the nodes adjacent to the start node: ***Level 1 checked!***
    - In the 2nd round, sends explorers two steps away from the starting node and not previously assigned to a level: ***Level 3 checked!***
  - Continues this process until **no new node** are found. When a node is visited, it is basically **printed out**.

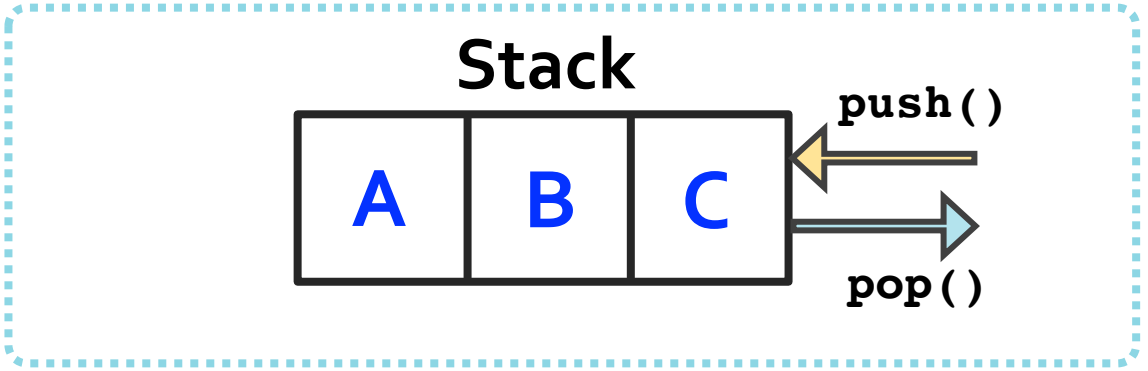
Quiz

- Based on that idea, which **data structure** do you think can be used for implementing BFS?

(i)



(ii)

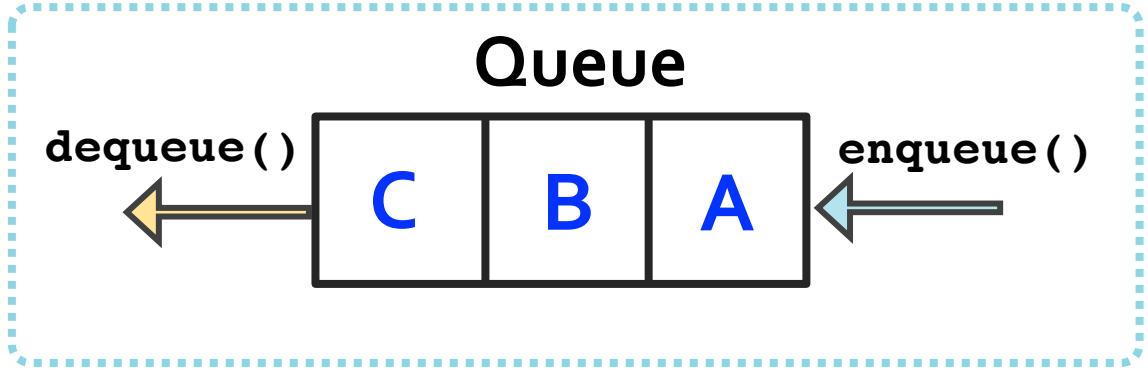


?

Answer

- **Queue** can be used to implment **BFS**.

(i)



BFS

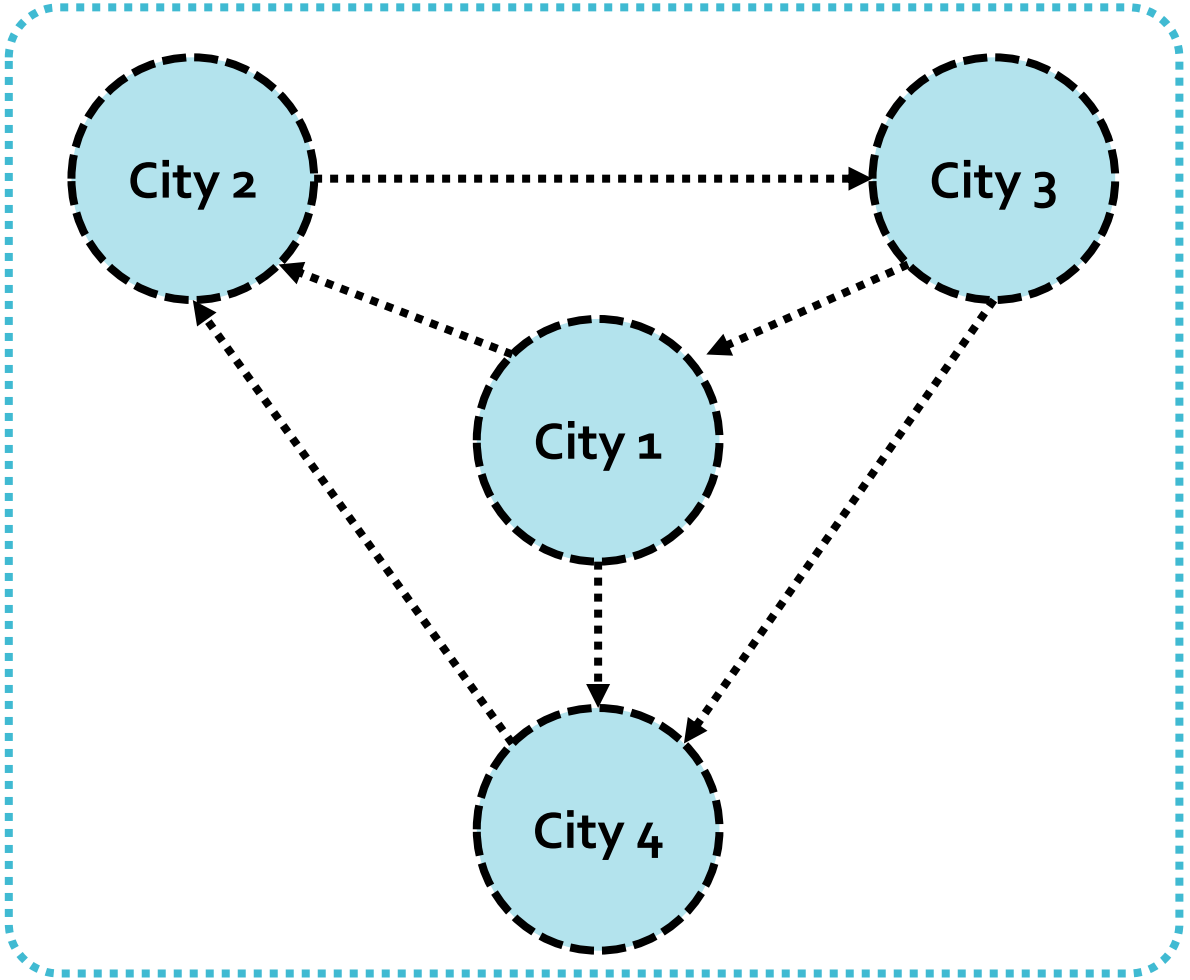
ref : youtu.be/oDqjPvD54Ss

Breadth First Search (BFS)

Searched

City 1	City 2	City 3	City 4
0	0	0	0

[Output:]



starting from **City 1**

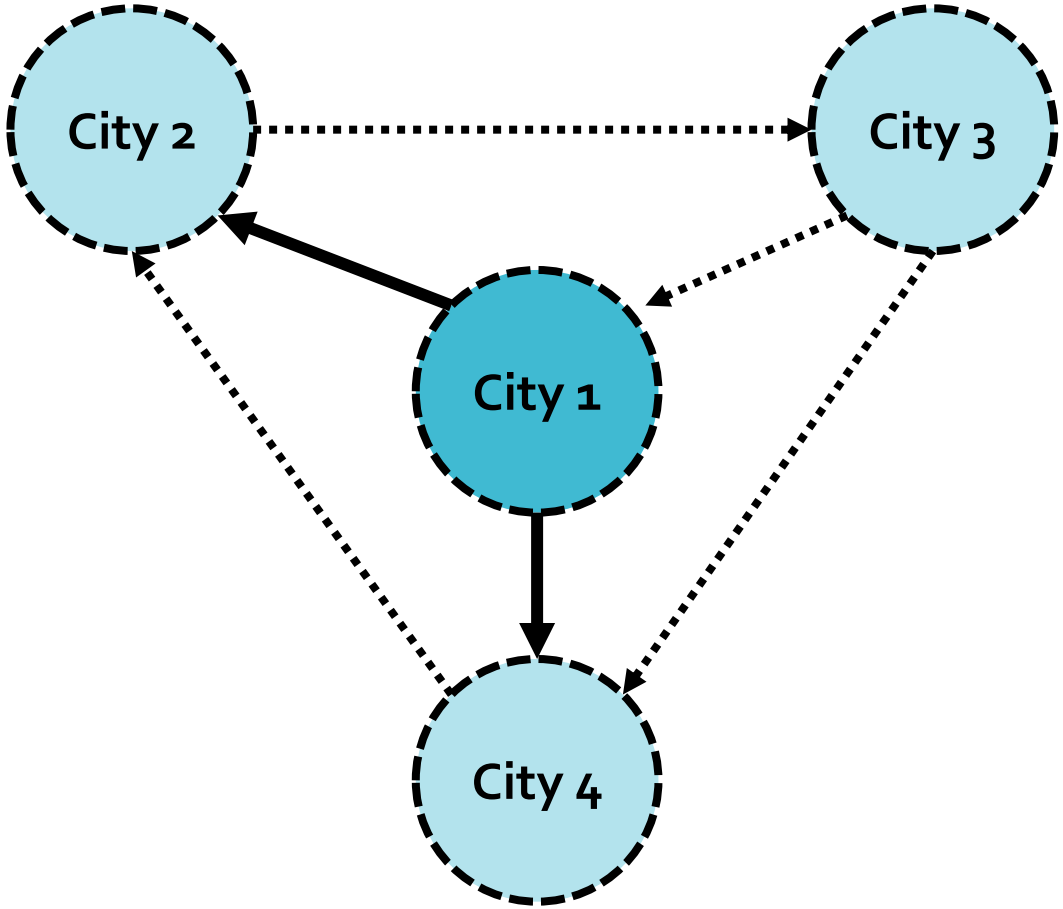
Breadth First Search (BFS)

Searched

City 1	City 2	City 3	City 4
1	0	0	0

[Output:]

[City 1],



starting from **City 1**

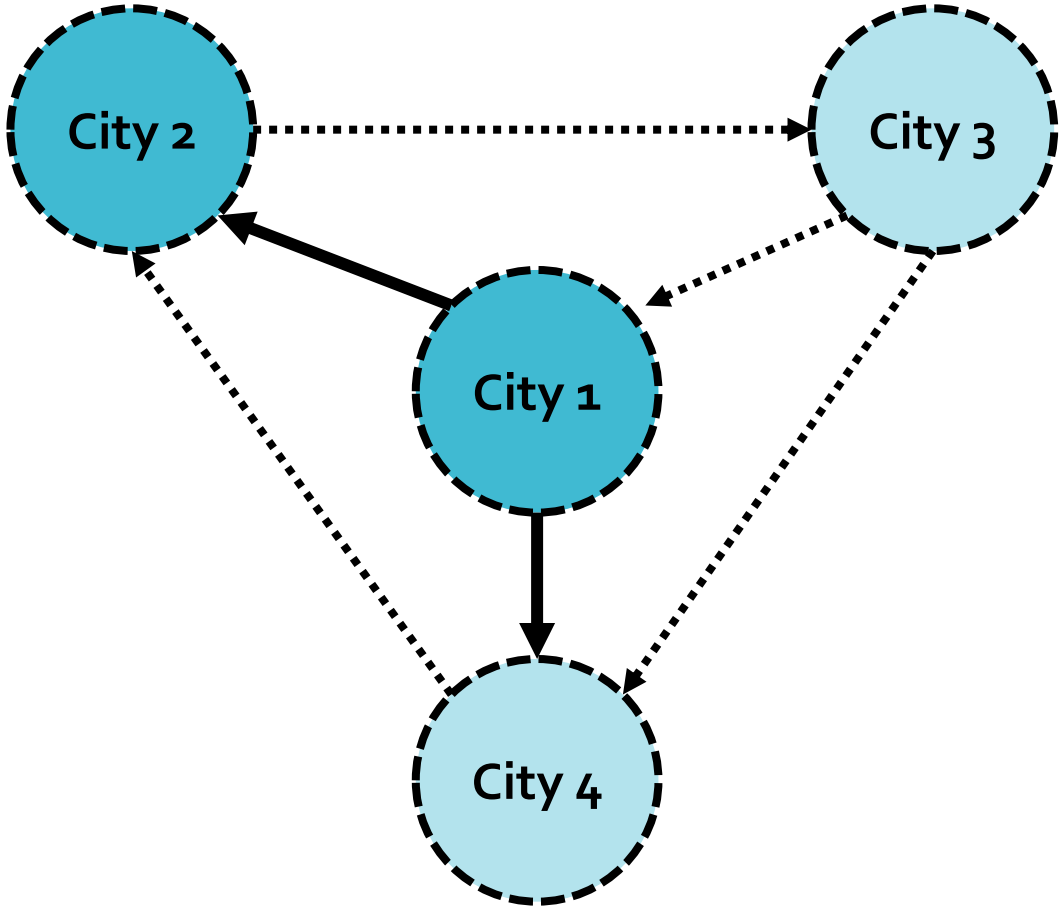
Breadth First Search (BFS)

Searched

City 1	City 2	City 3	City 4
1	1	0	0

[Output:]

[City 1], [City 2],



Starting from **City 1**

Neigbor is **City 2**

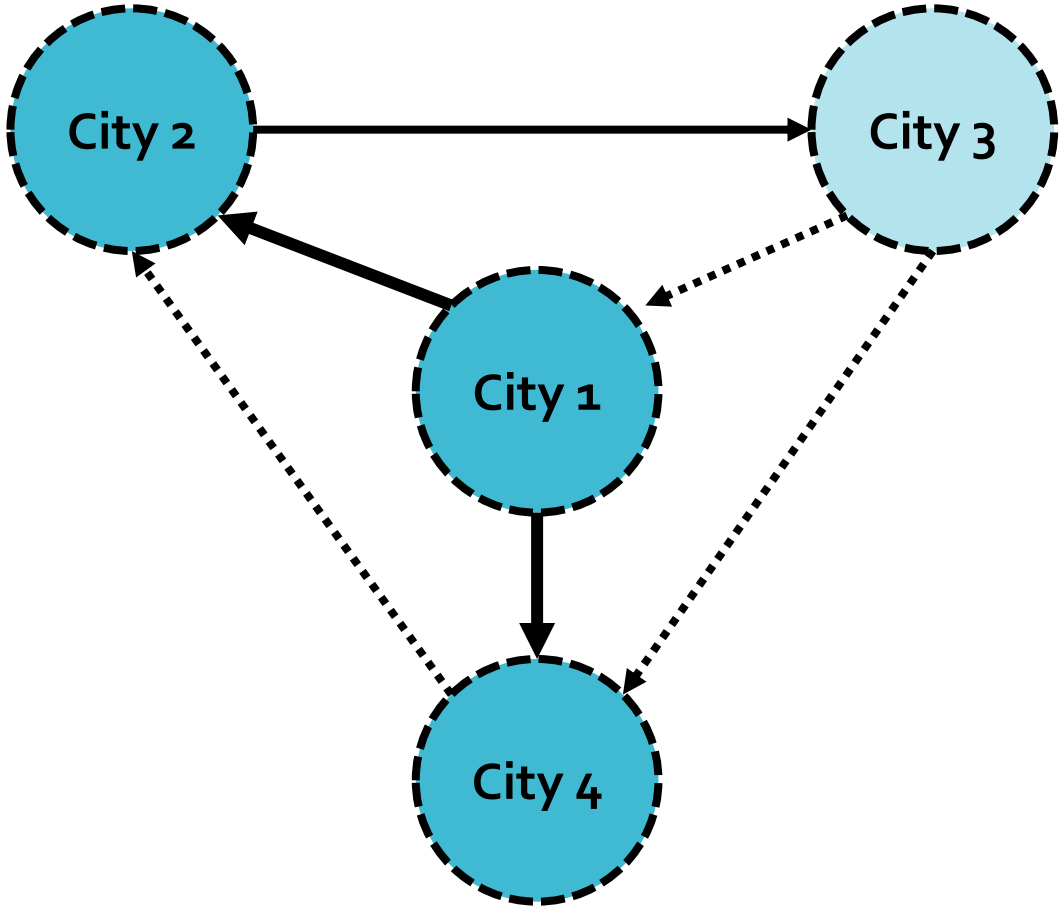
Breadth First Search (BFS)

Searched

City 1	City 2	City 3	City 4
1	1	0	1

[Output:]

[City 1], [City 2], [City 4]



Starting from **City 1**

Neigbor is **City 2**

Neigbor is **City 4**

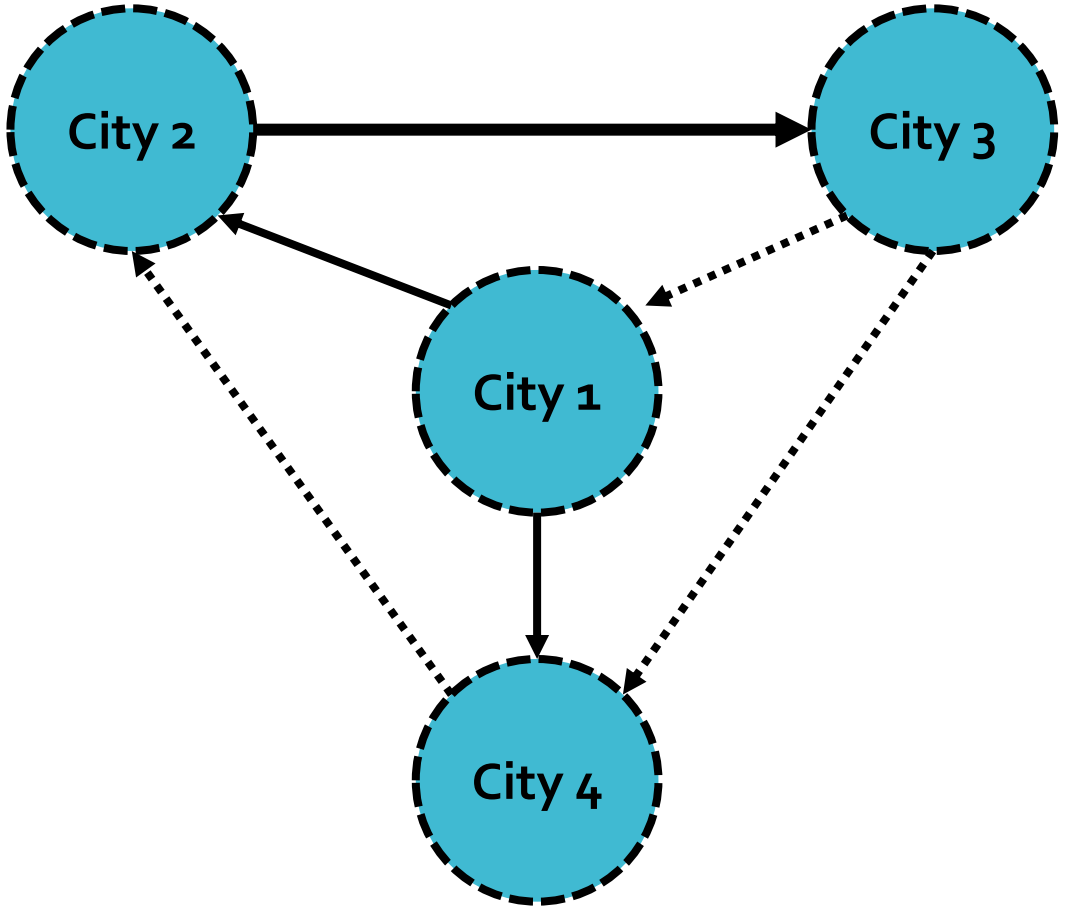
Breadth First Search (BFS)

Searched

City 1	City 2	City 3	City 4
1	1	1	1

[Output:]

[City 1], [City 2], [City 4], [City 3]



Starting from **City 1**

Neigbor is **City 2**

Neigbor is **City 4**

Neigbor is **City 3**

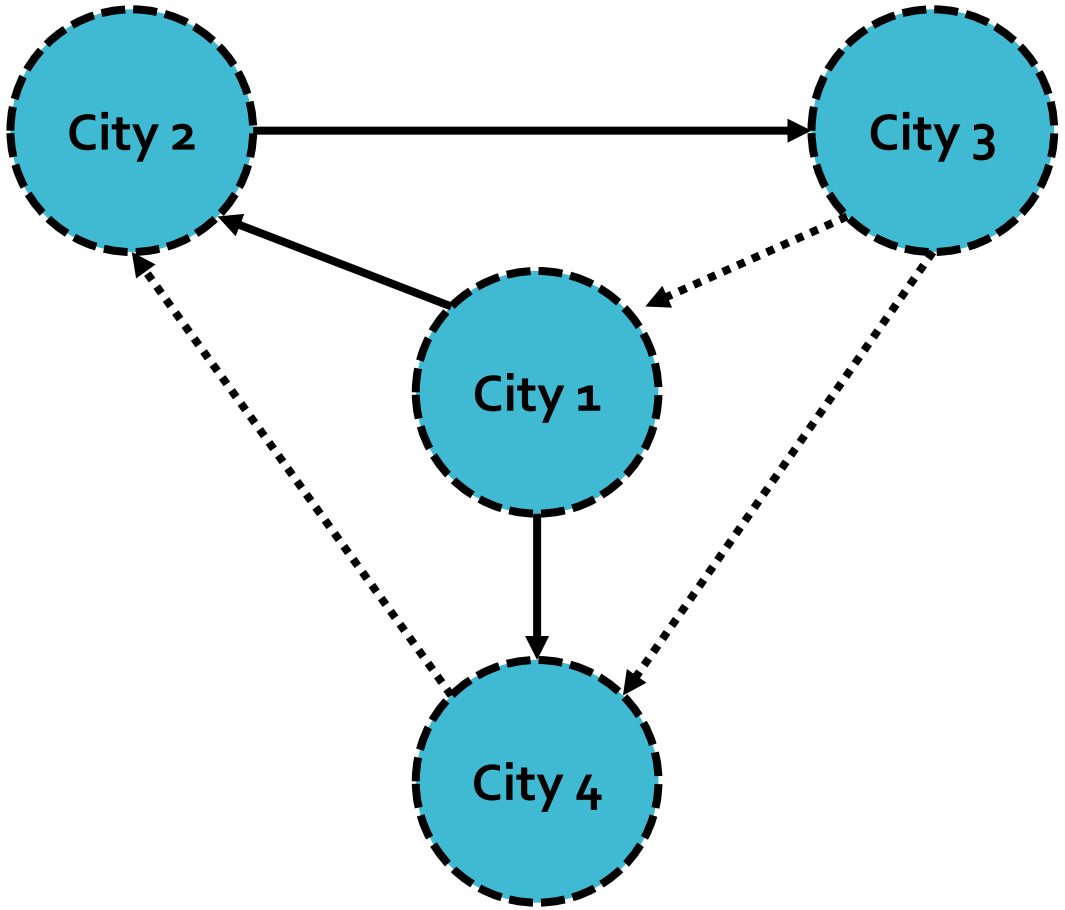
Breadth First Search (BFS)

Searched

City 1	City 2	City 3	City 4
1	1	1	1

[Output:]

[City 1], [City 2], [City 4], [City 3]



Starting from **City 1**

Neigbor is **City 2**

Neigbor is **City 4**

Neigbor is **City 3**

Q: Any new node?

A: No, terminate!

Implementation of **Breadth First Search (BST)**

Graph

```
class Graph:
    graph = dict()

    def breadth_first_search(self, node):

        searched = [node]
        search_queue = [node]

        while search_queue:
            node = search_queue.pop(0)

            print("[", node, end=" ], ")

            if node in self.graph:
                for neighbour in self.graph[node]:
                    if neighbour not in searched:
                        searched.append(neighbour)
                        search_queue.append(neighbour)
```

Implementation of **Breadth First Search (BST)**

Graph

```
class Graph:
    graph = dict()

    def breadth_first_search(self, node):

        searched = [node]
        search_queue = [node]

        while search_queue:
            node = search_queue.pop(0)

            print("[", node, end=" ], ")

            if node in self.graph:
                for neighbour in self.graph[node]:
                    if neighbour not in searched:
                        searched.append(neighbour)
                        search_queue.append(neighbour)
```

Graph

Testing the **Breadth First Search (BST)**

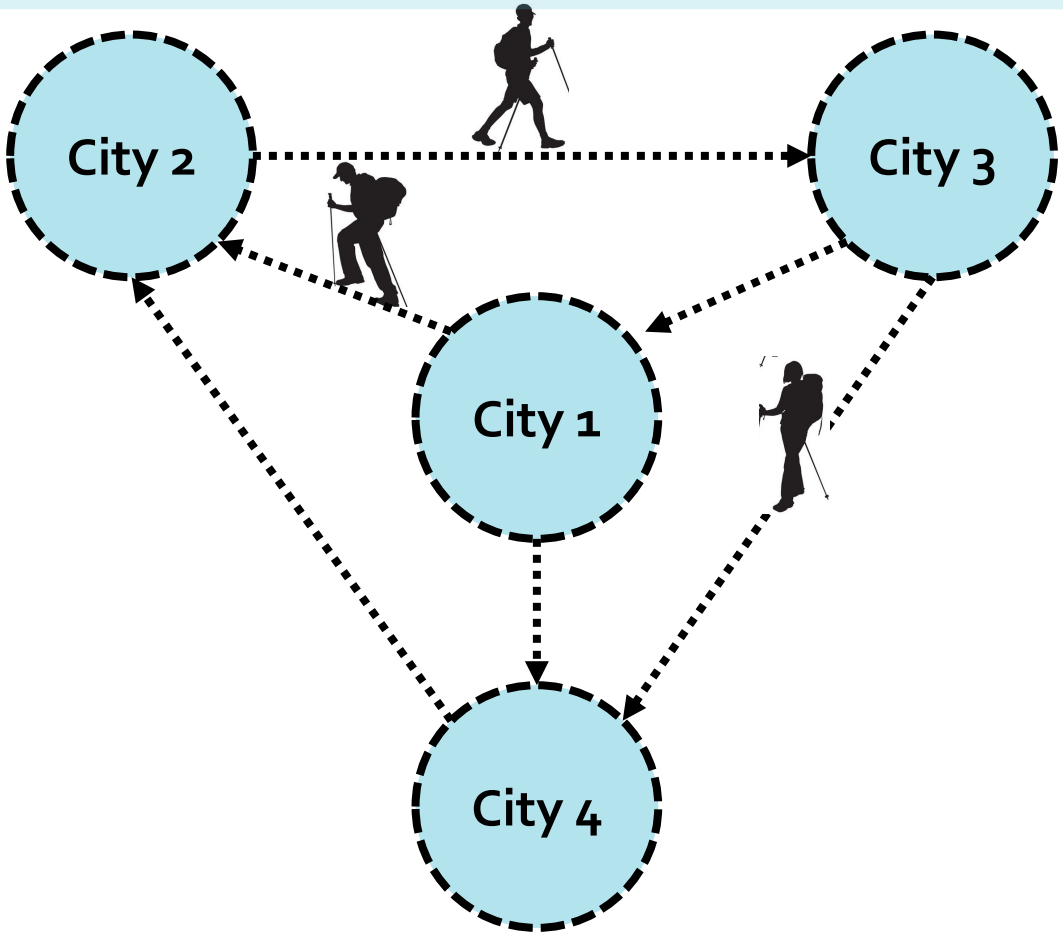
```
my_graph = Graph()

my_graph.add_edge('City 1', 'City 2')
my_graph.add_edge('City 1', 'City 4')
my_graph.add_edge('City 2', 'City 3')
my_graph.add_edge('City 3', 'City 1')
my_graph.add_edge('City 3', 'City 4')
my_graph.add_edge('City 4', 'City 2')

my_graph.breadth_first_search('City 1')
```

[Output:]

```
[ City 1 ], [ City 2 ], [ City 4 ], [ City 3 ]
```

DFS

- **Depth-First Search (DFS)** is another traverse algorithm that looks like an algorithm of backtracking.
- **DFS** sends out an explorer to a route **as deep as possible** in the graph, and then backtracks and sends it to another route.



DFS

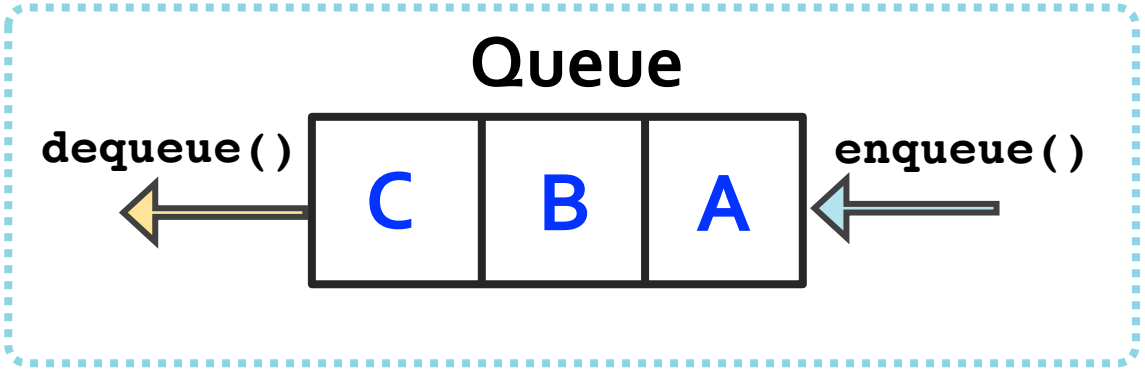
- The **idea behind DFS** is to start from a node and move to the adjacent unvisited node and **continue** until there is no unvisited adjacent node.
- Then **backtrack** and check for other unvisited nodes and traverse them.
- When a node is visited, it is basically **printed out**.

DFS

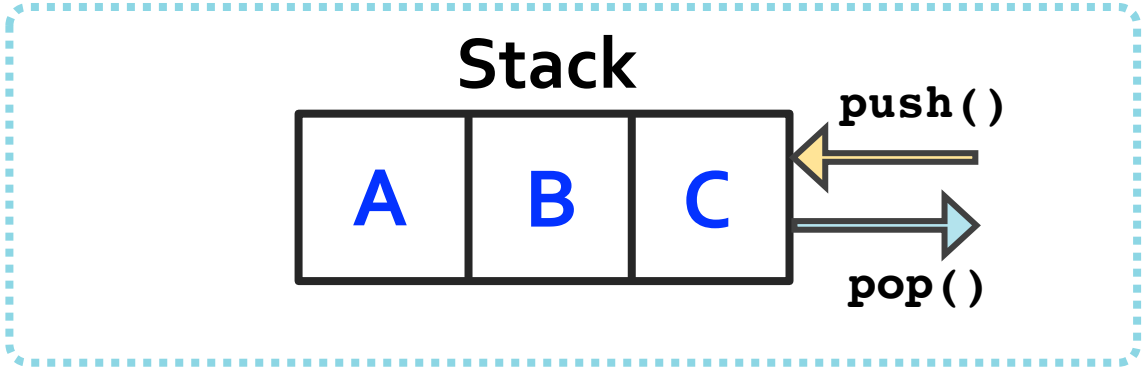
- **Depth-First Search (DFS)** algorithm:
  - Starts at a node (vertex):
    - Check the current node, note it as checked.
    - Traverse the adjacent unmarked node and call the recursive function with each of them.

Quiz

- **Queue** can be used to implment **BFS.**

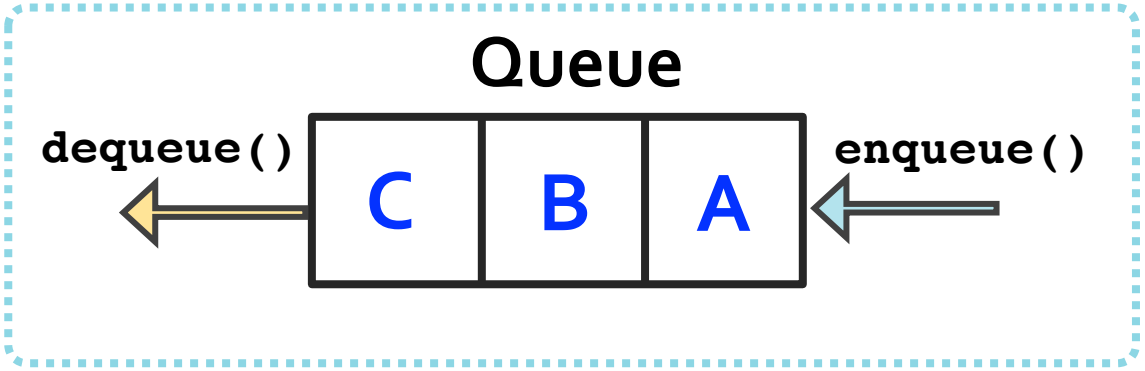


- What about **DFS**?

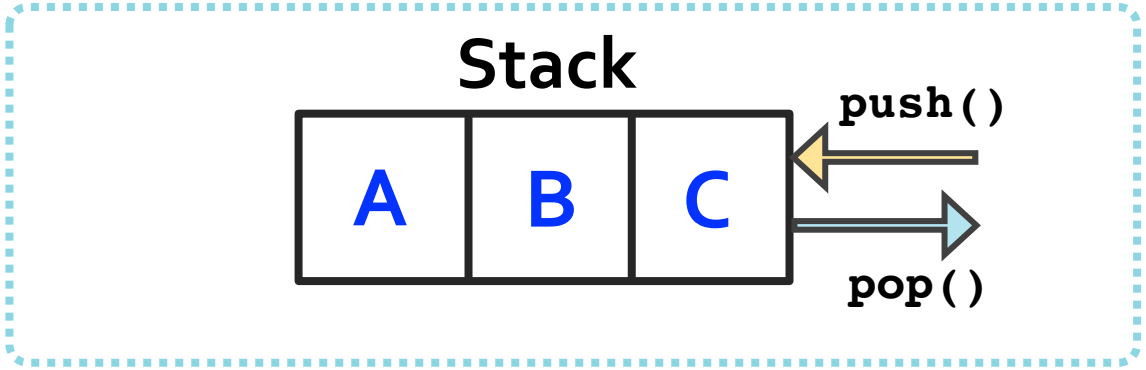


Answer

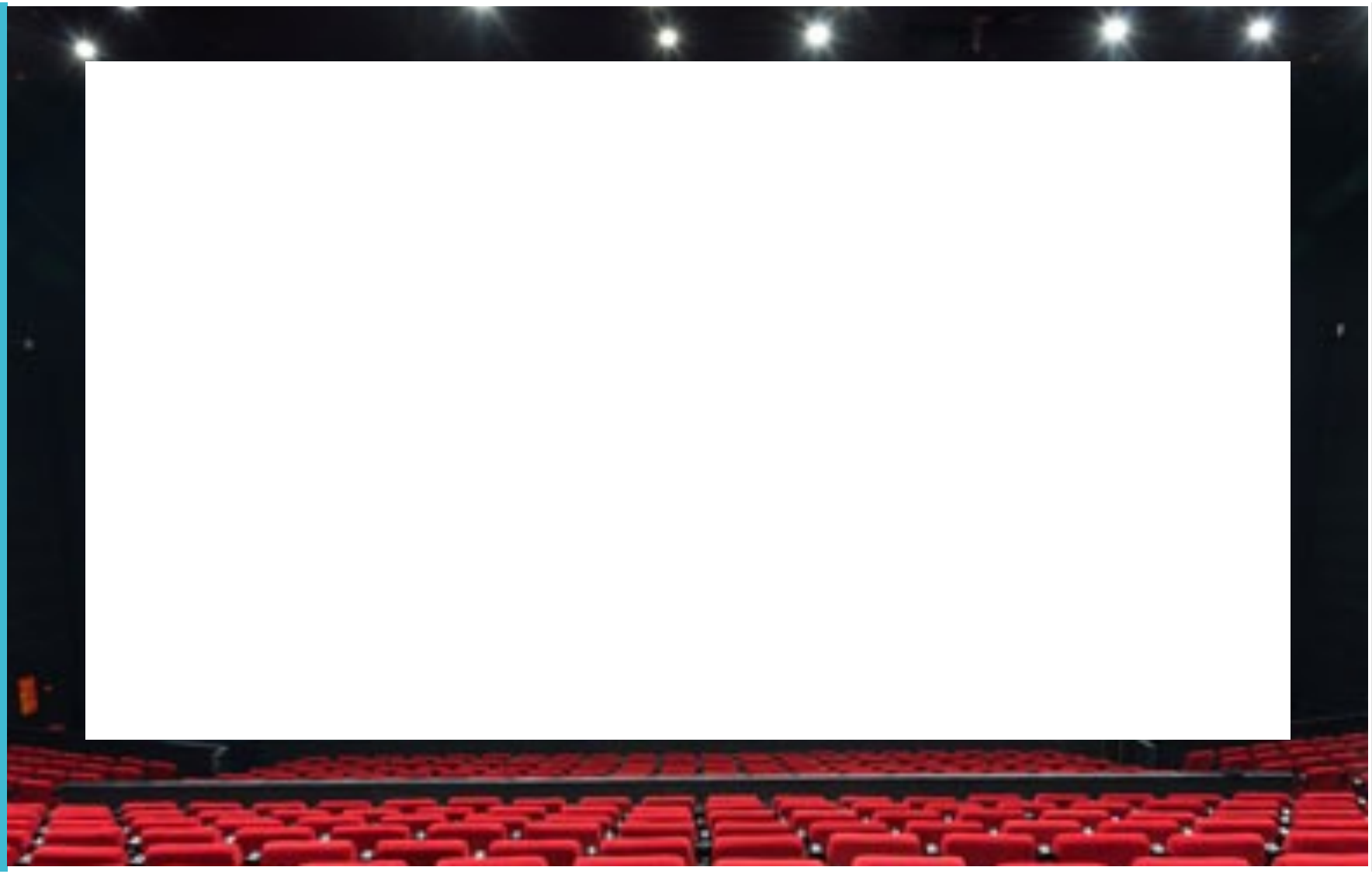
- **Queue** can be used to implment **BFS**.



- **Stacks** can be used to implment **DFS**.



BFS vs DFS



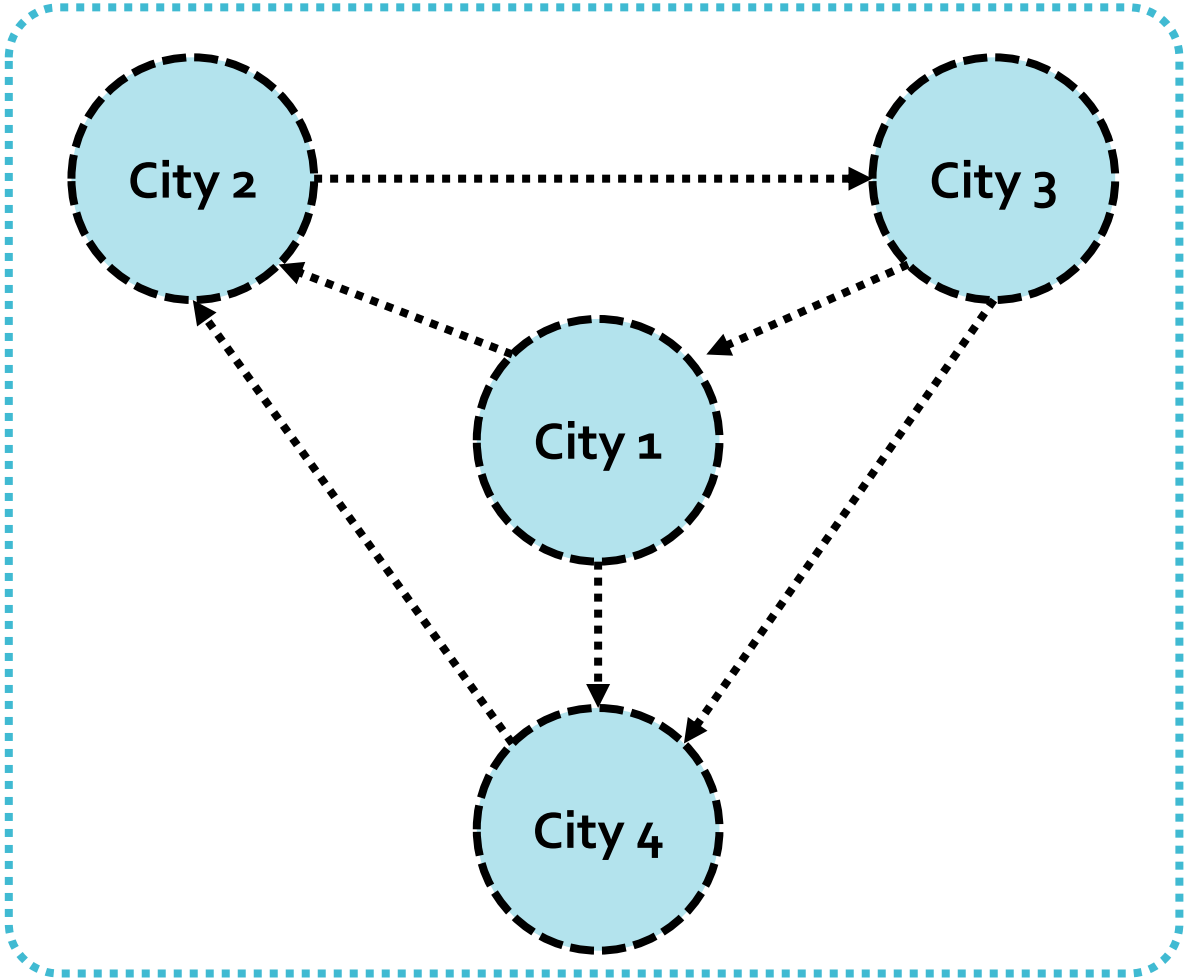
ref : Youtube

Depth First Search (DFS)

Searched

City 1	City 2	City 3	City 4
0	0	0	0

[Output:]



starting from **City 1**

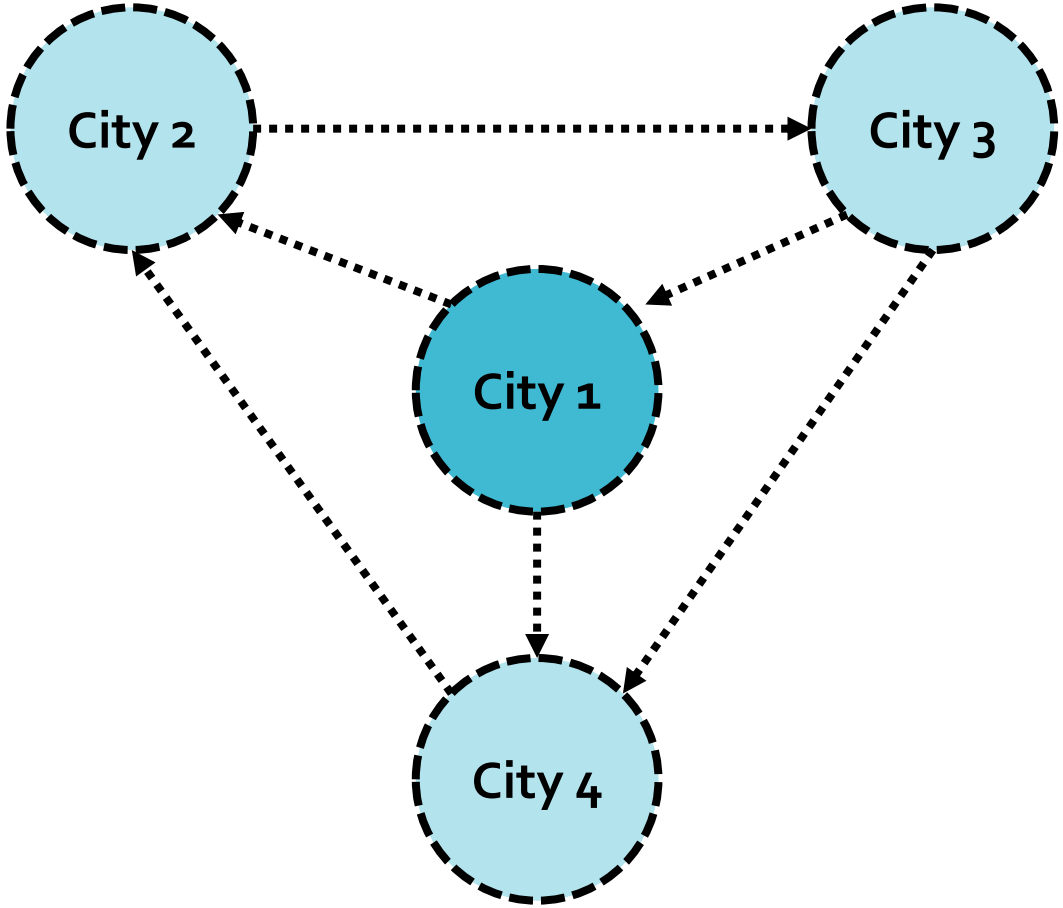
Depth First Search (DFS)

Searched

City 1	City 2	City 3	City 4
1	0	0	0

[Output:]

[City 1],



starting from **City 1**

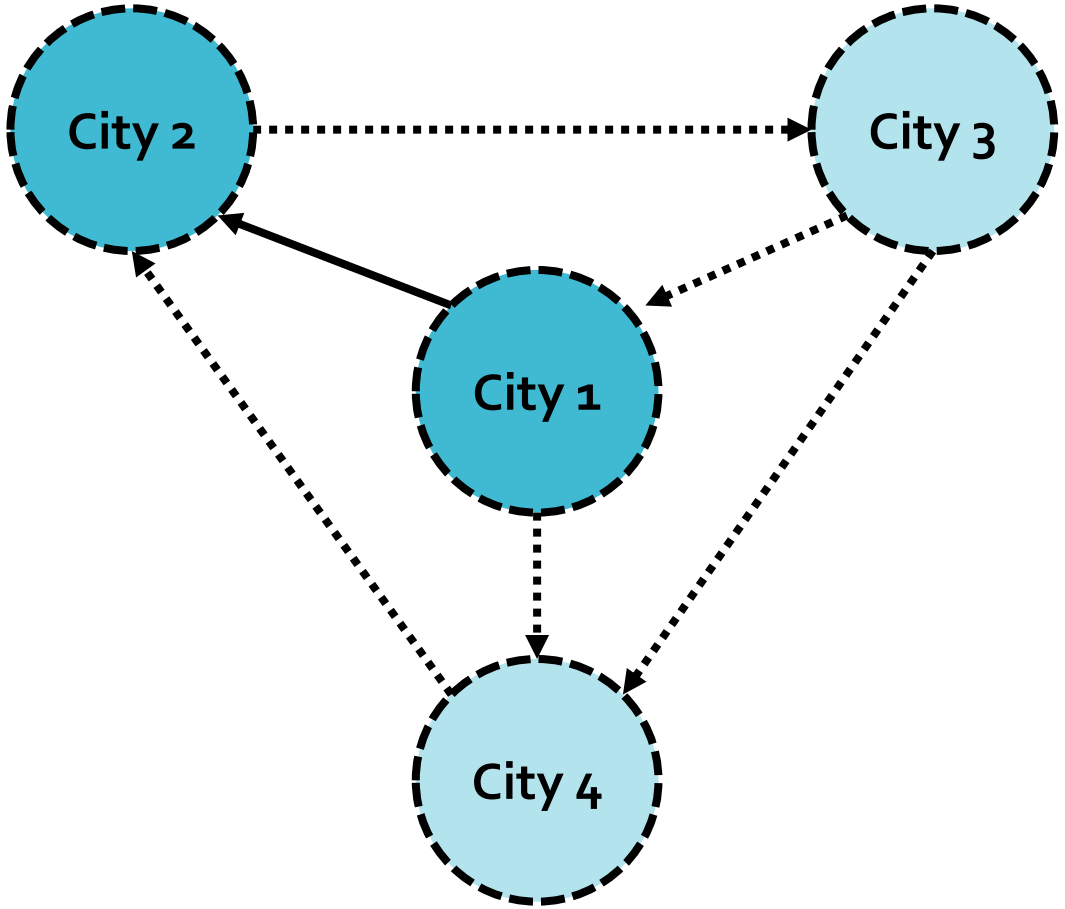
Depth First Search (DFS)

Searched

City 1	City 2	City 3	City 4
1	1	0	0

[Output:]

[City 1], [City 2],



Starting from **City 1**

Neigbor is **City 2**

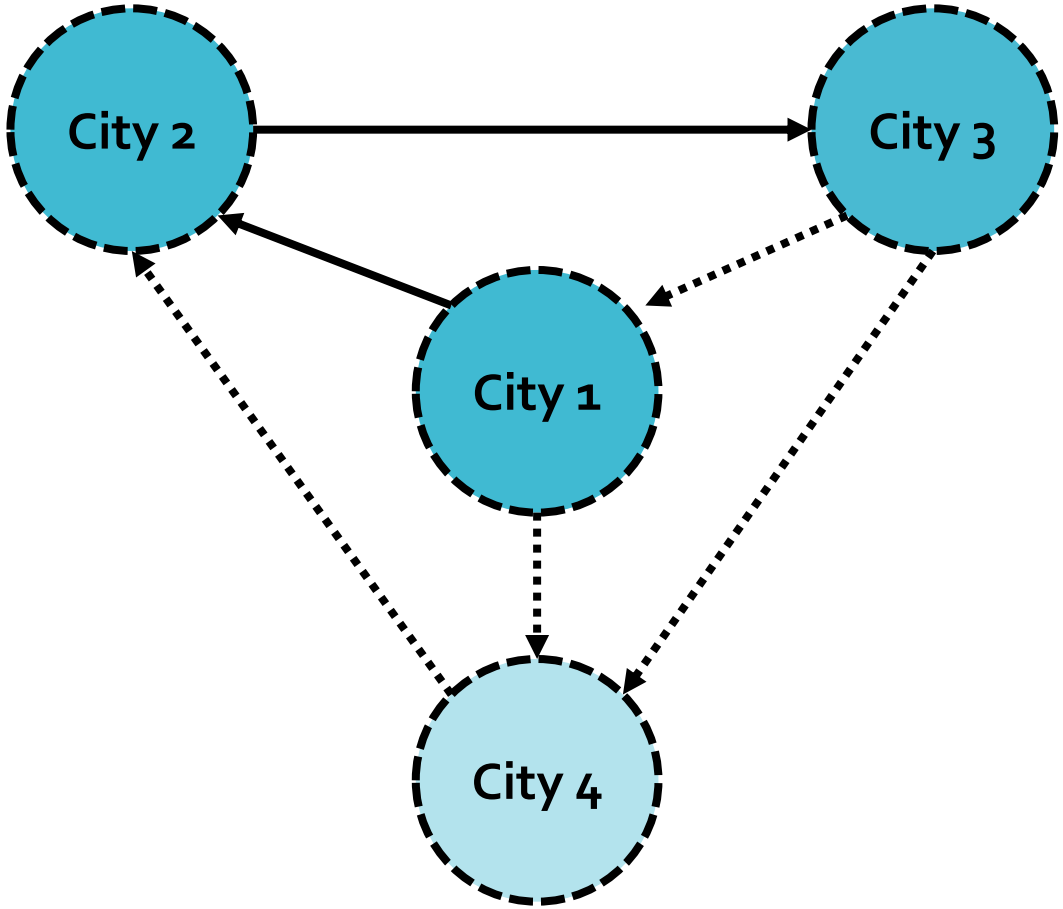
Depth First Search (DFS)

Searched

City 1	City 2	City 3	City 4
1	1	1	0

[Output:]

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



Starting from **City 1**

Neigbor is **City 2**

Neigbor is **City 3**

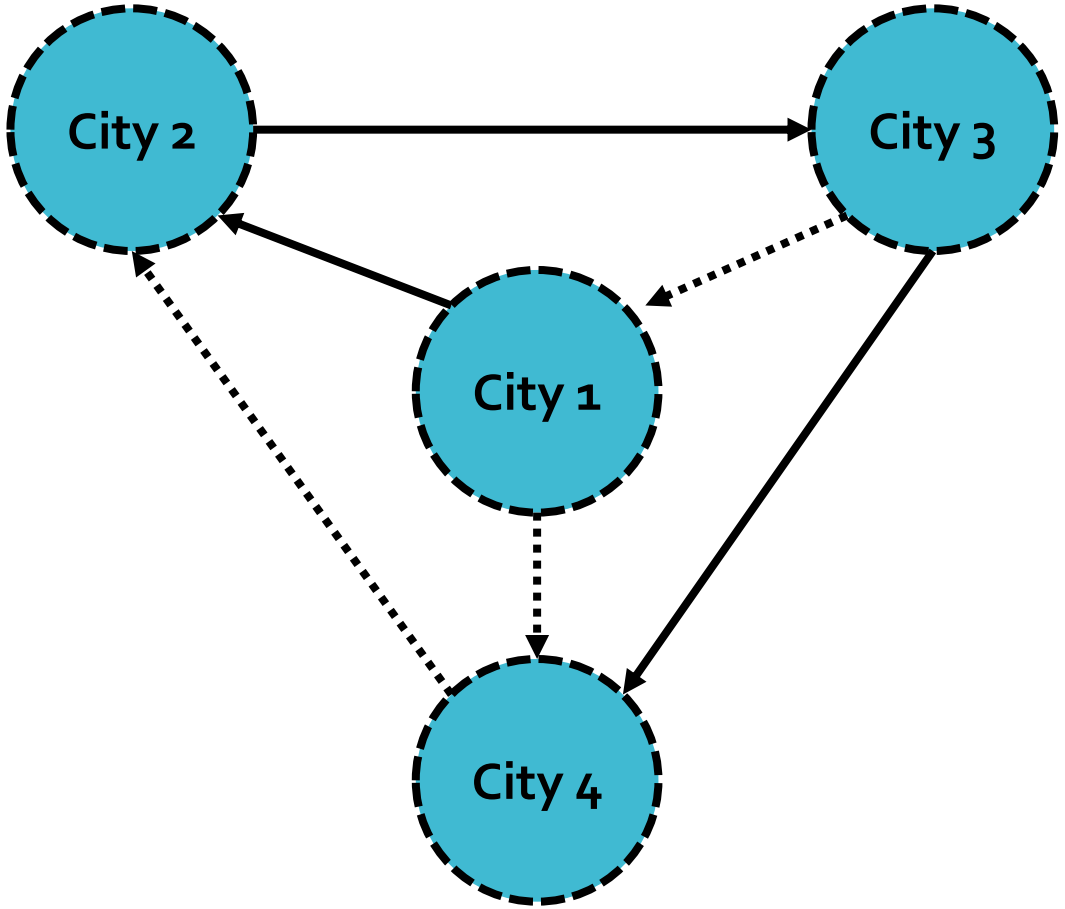
Depth First Search (DFS)

Searched

City 1	City 2	City 3	City 4
1	1	1	1

[Output:]

[City 1], [City 2], [City 3], [City 4]



Starting from **City 1**

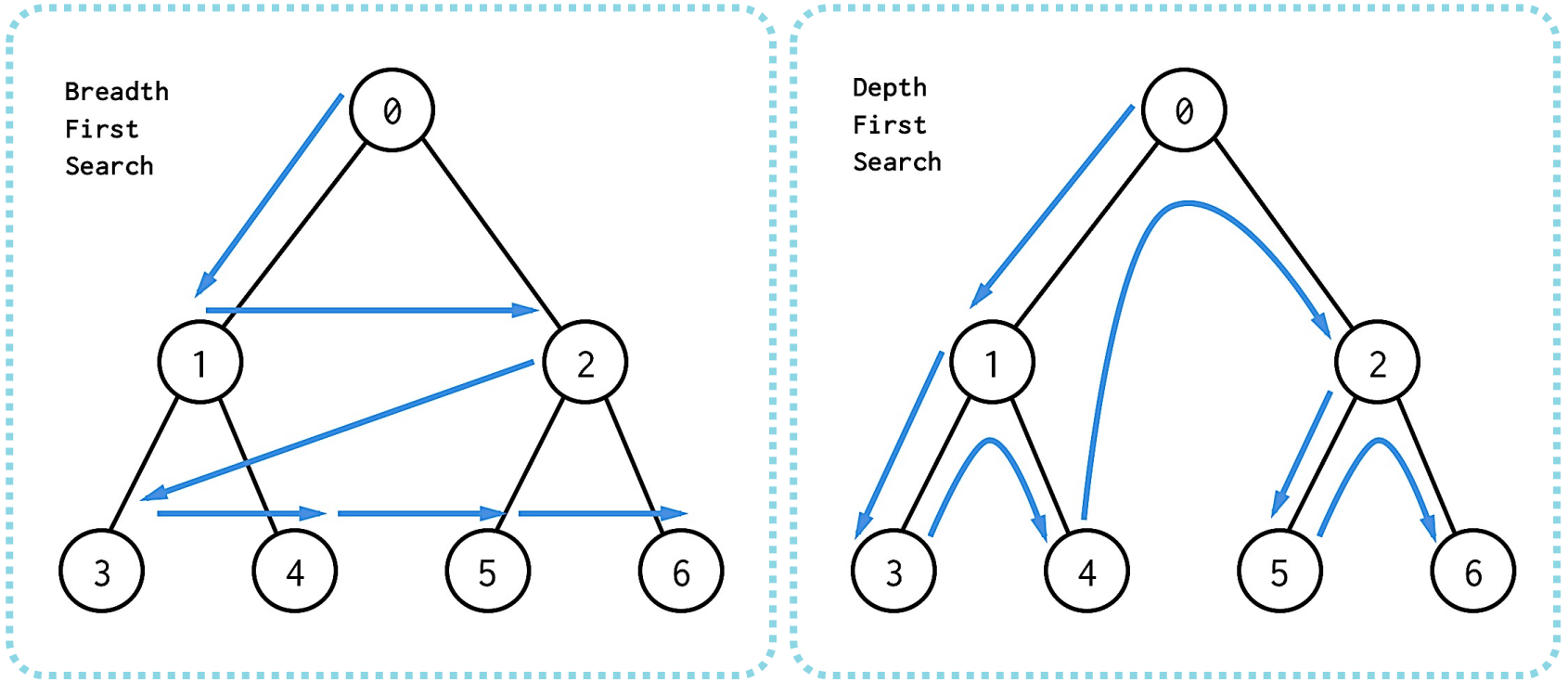
Neigbor is **City 2**

Neigbor is **City 3**

Neigbor is **City 4**

BFS vs DFS

- Breadth First Search (BFS) vs Depth First Search (DFS)



mishadoff.com

Graph

❖ Implementation of **Depth First Search (DFS)**

```
class Graph:
    graph = dict()
    searched = []

    def depth_first_search(self, node):
        if node not in self.searched:

            print("[", node, end=" ],")

            self.searched.append(node)
            if node in self.graph:
                for neighbour in self.graph[node]:
                    self.depth_first_search(neighbour)
```

Graph

❖ Testing the **Depth First Search (DFS)**

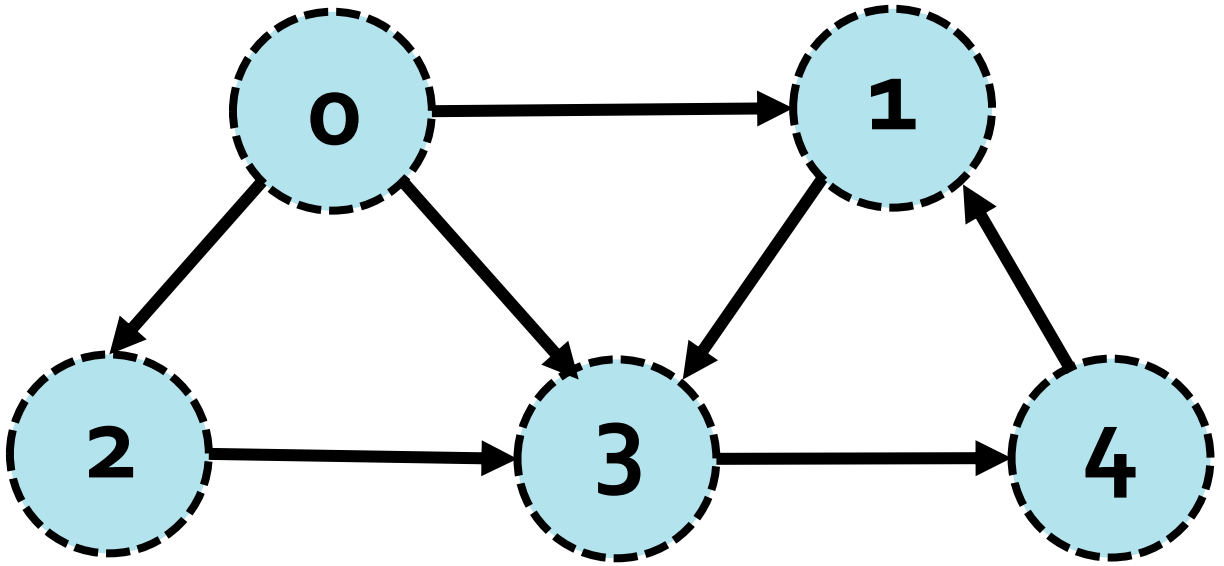
```
my_graph = Graph()

my_graph.add_edge('City 1', 'City 2')
my_graph.add_edge('City 1', 'City 4')
my_graph.add_edge('City 2', 'City 3')
my_graph.add_edge('City 3', 'City 1')
my_graph.add_edge('City 3', 'City 4')
my_graph.add_edge('City 4', 'City 2')

my_graph.depth_first_search('City 1')
```

[Output:]

```
[ City 1 ],[ City 2 ],[ City 3 ],[ City 4 ]
```

Quiz

- Write a function **`build_my_graph()`** that:
  a) creates the following Graph.
  b) runs Breedth First Search (BFS) starting from 0.
  c) runs Depth First Search (DFS) starting from 0.



Answer

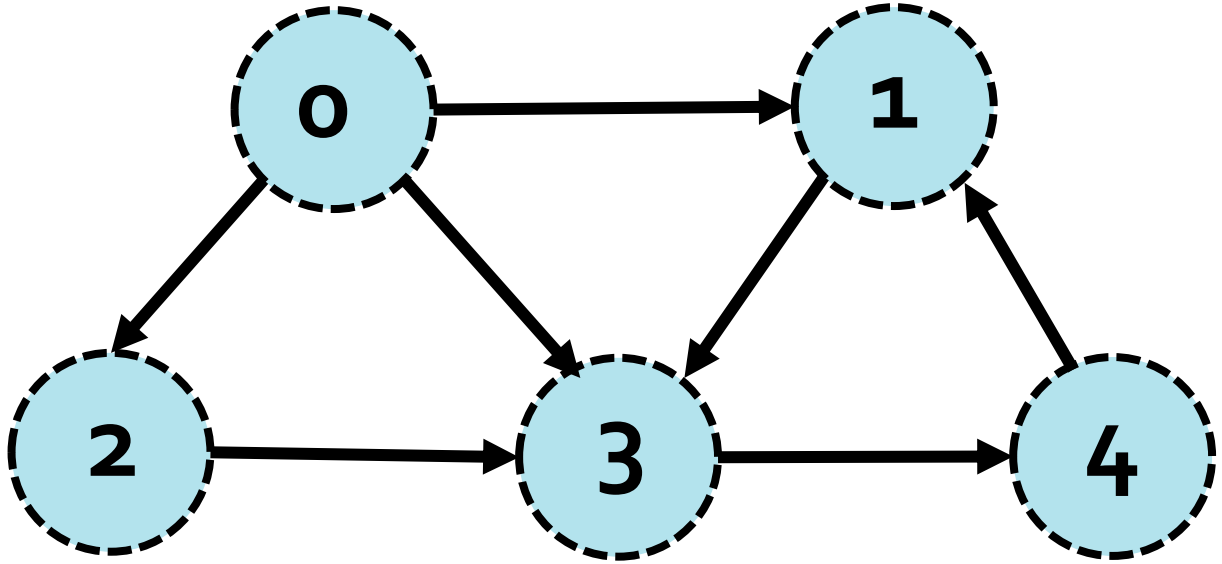
```
def build_my_graph():
    my_graph = Graph()

    my_graph.add_edge(0, 1)
    my_graph.add_edge(0, 2)
    my_graph.add_edge(0, 3)
    my_graph.add_edge(1, 3)
    my_graph.add_edge(2, 3)
    my_graph.add_edge(3, 4)
    my_graph.add_edge(4, 1)

    print('Breadth First Search (BFS):')
    my_graph.breadth_first_search(0)
    print('Depth First Search (DFS):')
    my_graph.depth_first_search(0)
```

Answer

```
build_my_graph()
```

[Output:]

```
Breadth First Search (BFS):
[ 0 ], [ 1 ], [ 2 ], [ 3 ], [ 4 ]
Depth First Search (DFS):
[ 0 ],[ 1 ],[ 3 ],[ 4 ],[ 2 ]
```



Next Lesson

- More Shortest Paths Algorithms (Dijkstra)