哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

数据结构
Data Structures

Chapter 10 Graph

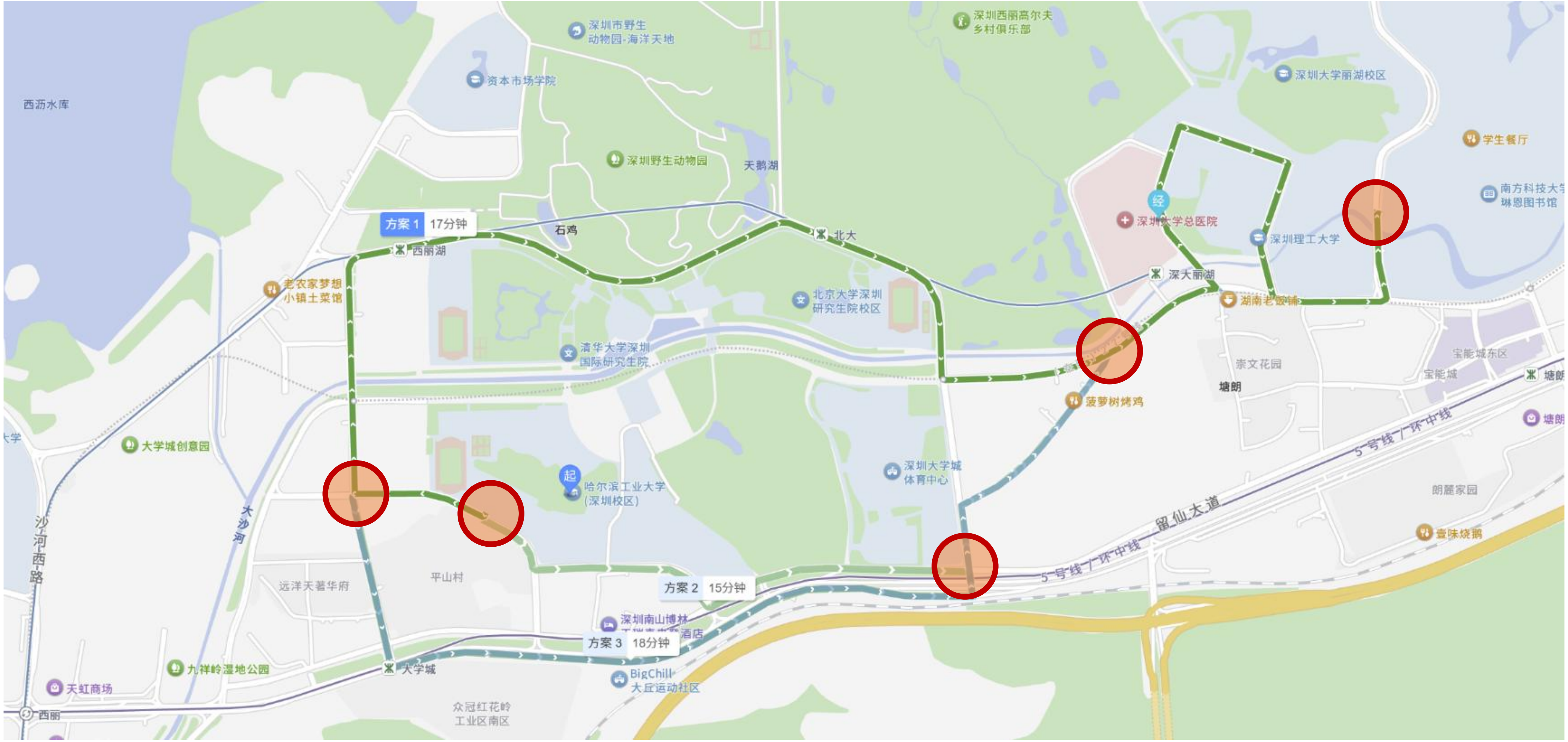
Prof. Yitian Shao
School of Computer Science and Technology

Graph

Course Overview

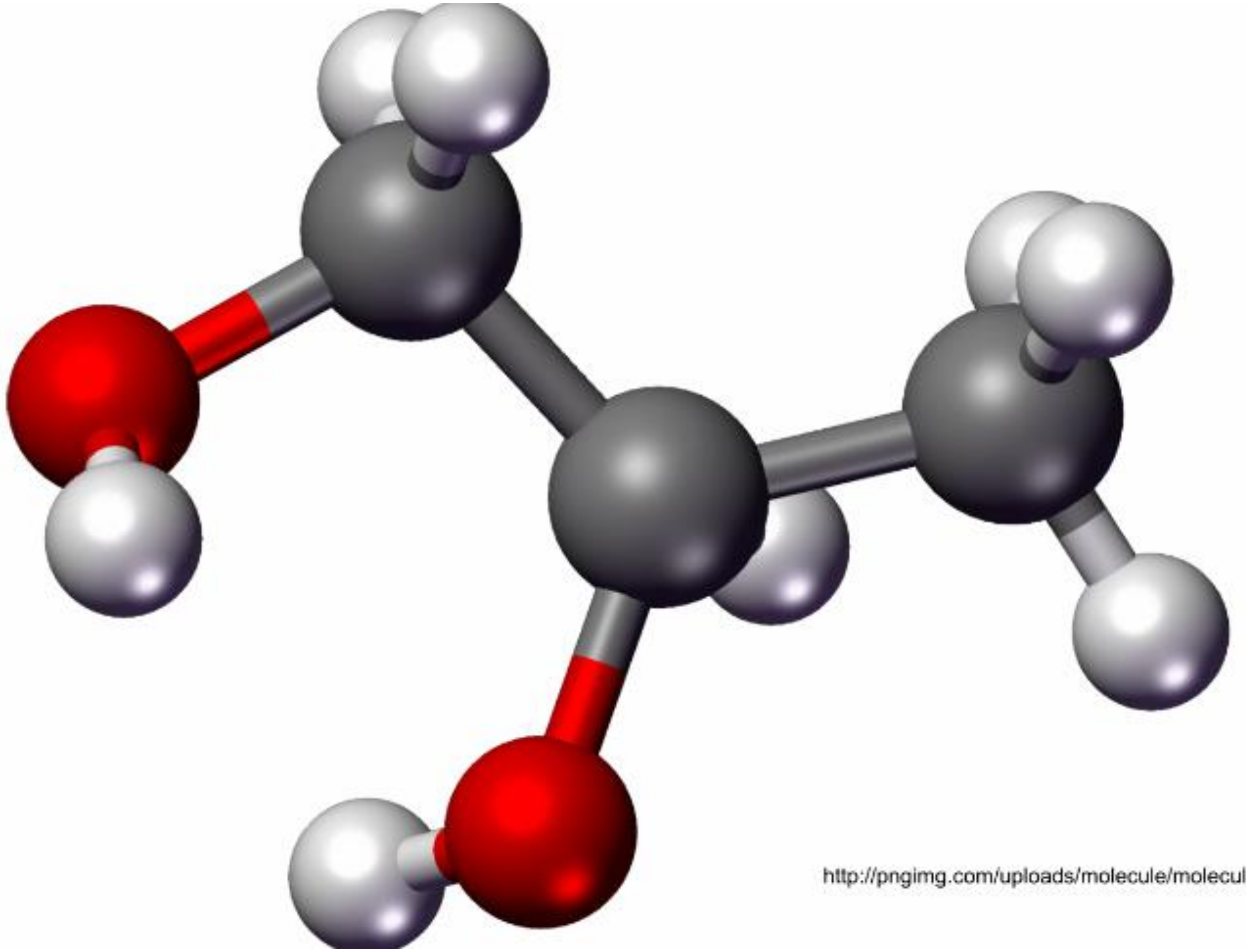
- Introducing Graph
- Graph Types
- Representing Graph
- Path
- Graph Properties
- Find a Path
- Find the Shorted Path
- Find the Least-Cost Path
  - Dijkstra's Algorithm

Maps



深圳市野生
动物园-海洋天地
深圳西丽高尔夫
乡村俱乐部
资本市场学院
西沥水库
深圳大学丽湖校区
学生餐厅
深圳野生动物园
天鹅湖
南方科技大学
琳恩图书馆
方案 1 17分钟
石鸡
北大
深圳大学总医院
深圳理工大学
西丽湖
深大丽湖
老农家梦想
小镇土菜馆
北京大学深圳
研究生院校区
湖南老饭铺
清华大学深圳
国际研究生院
崇文花园
宝能城东区
宝能城
塘朗
菠萝树烤鸡
塘朗
大学城创意园
5号线/环中线
深圳大学城
体育中心
起
哈尔滨工业大学
(深圳校区)
朗麗家园
留仙大道
大沙河
沙河西路
壹味烧鹅
5号线/环中线
平山村
远洋天著华府
方案 2 15分钟
深圳南山博林
酒店
方案 3 18分钟
大学城
九祥岭湿地公园
BigChill
大丘运动社区
天虹商场
西丽
众冠红花岭
工业区南区

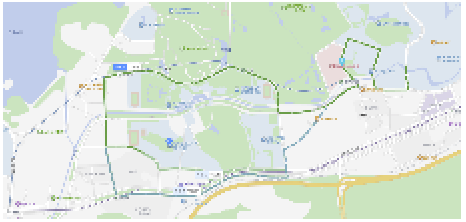
Molecules



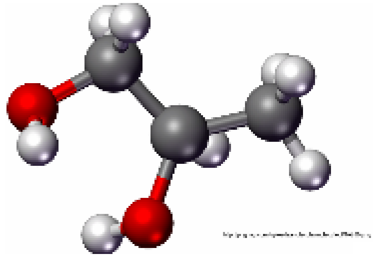
http://pngimg.com/uploads/molecule/molecule_PNG50.png

Introducing the Graph

- A graph is a mathematical structure for representing **relationships**
- Consists of **nodes** (aka **vertices**) and **edges** (aka **arcs**)
  - **Edges** are the relationships, **nodes** are the items
- Examples:
  - Map: cities (nodes) are connected by roads (edges)

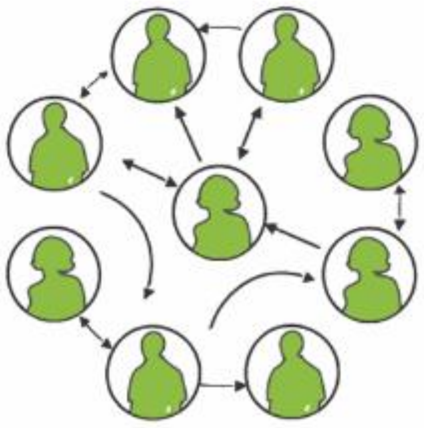


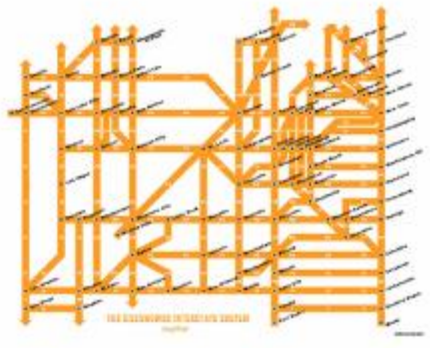
  - Molecules: atoms (nodes) are connected by bonds (edges)



More Graph Examples

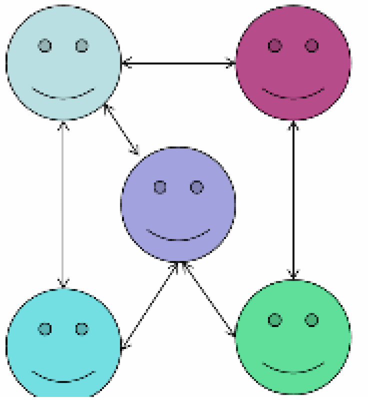
- For each, what are the nodes and what are the edges?
  - Web pages with links
  - Functions in a program that call each other
  - Facebook/WeChat/QQ friends
  - Family trees
  - Paths through a maze



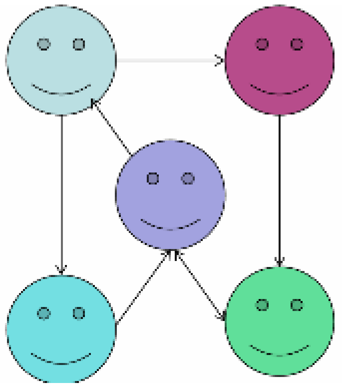


Undirected vs. Directed Graph

- Some relationships are **mutual**
  - Facebook/Wechat/QQ



- Some are **one-way**
  - Twitter/Instagram/Weibo
  - Doesn't mean that all relationships are non-mutual

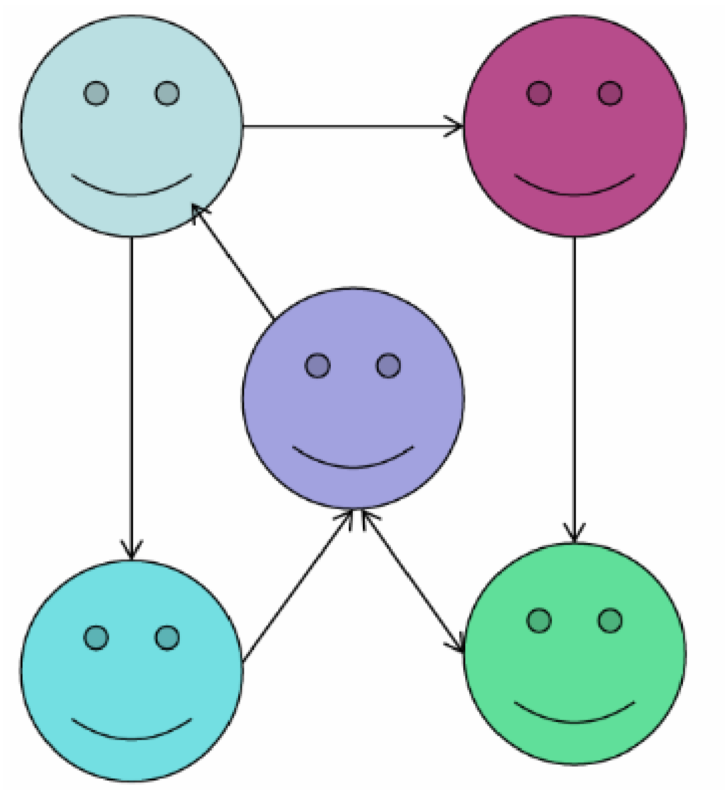


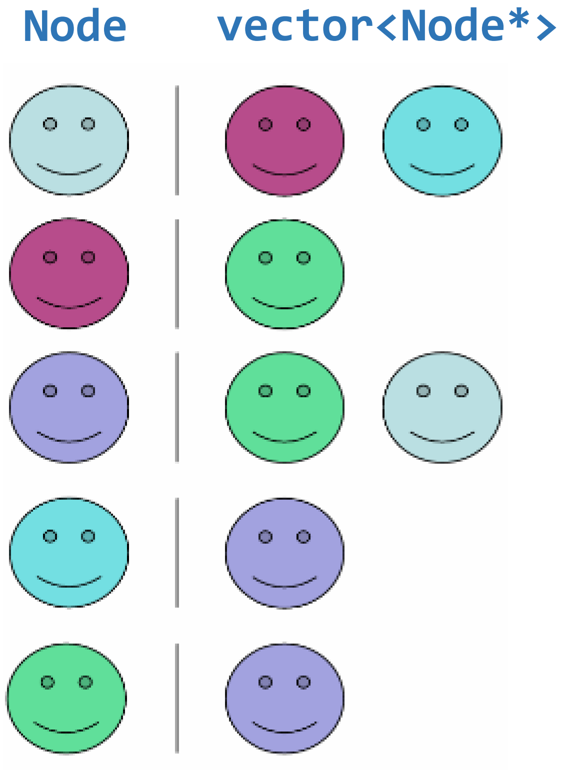
Representing Graphs

- Two main ways:
  - Have **each node store the nodes it's connected to** (adjacency list)
  - Have **a list of all the edges** (edge list)
- The choice depends on the problem you're trying to solve
- You can sometimes represent graphs implicitly instead of explicitly storing the edges and nodes
  - Draw a picture to see the graph more clearly!

Adjacency List

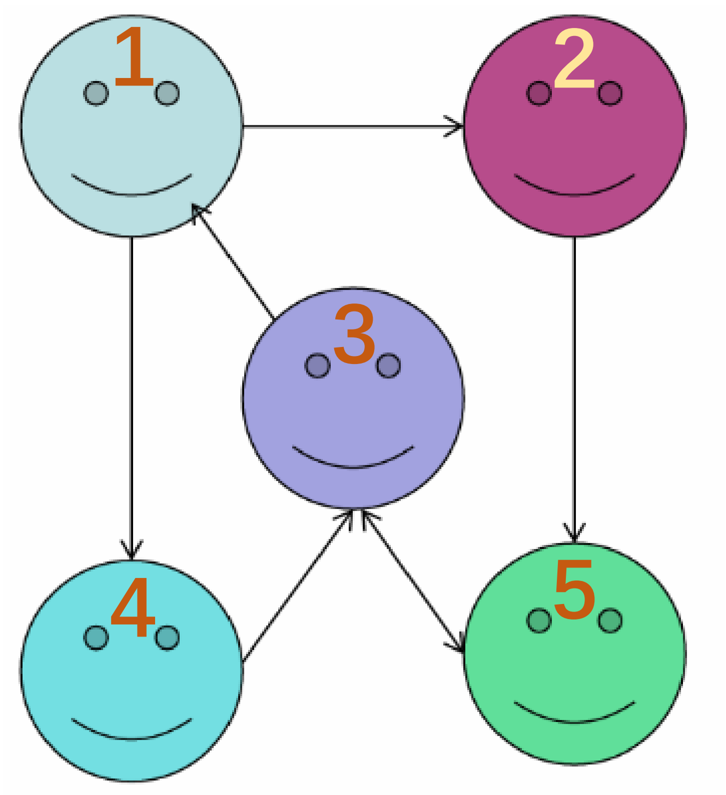
```
struct Node {
	string name;
	vector<Node*> adjacencyList;
};
```

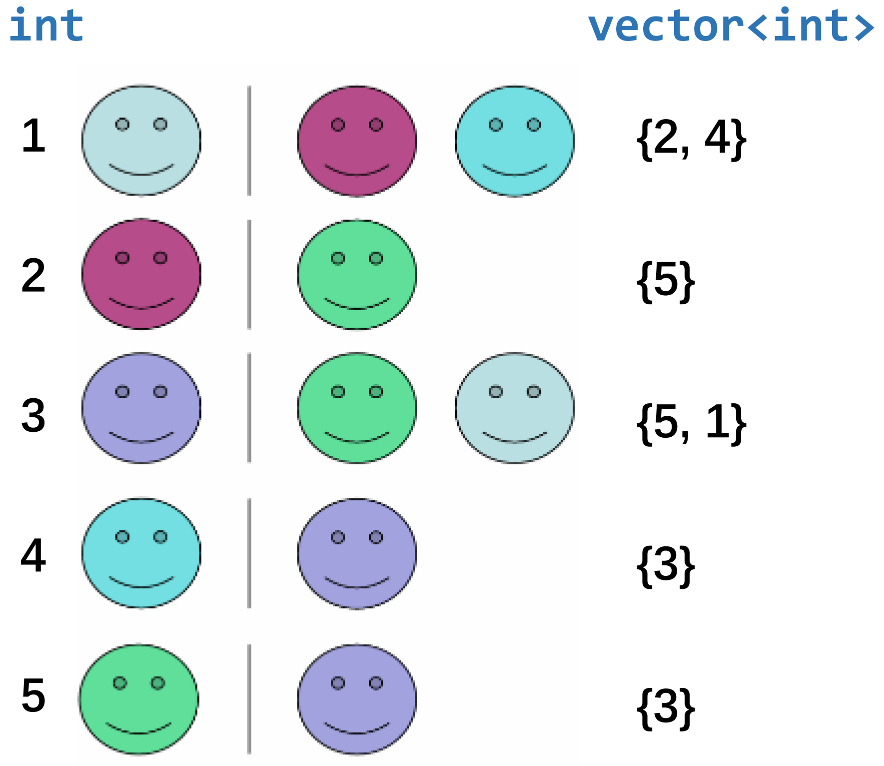




Adjacency List Represented as Unique Integer IDs

```
vector<int> IDs = {1, 2, 3, 4, 5}; // Note that each node must has a unique ID
vector< vector<int> > adjacencyList = {{2,4},{5},{5,1},{3},{3}};
```

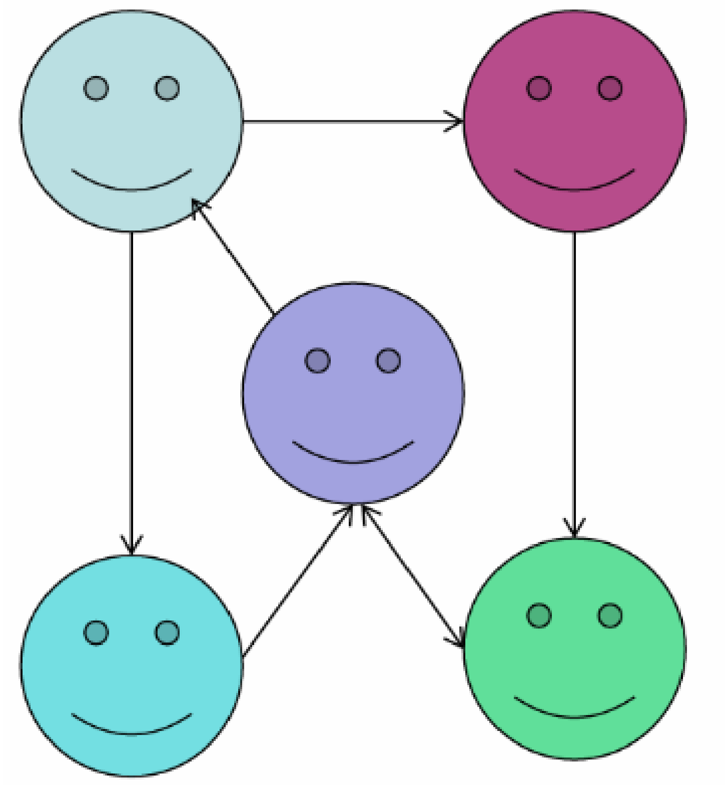


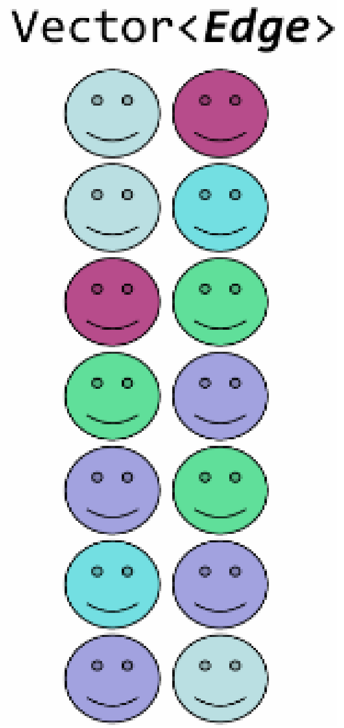


Edge List

- Store a Vector<Edge>
- ***Edge*** struct would have the two node

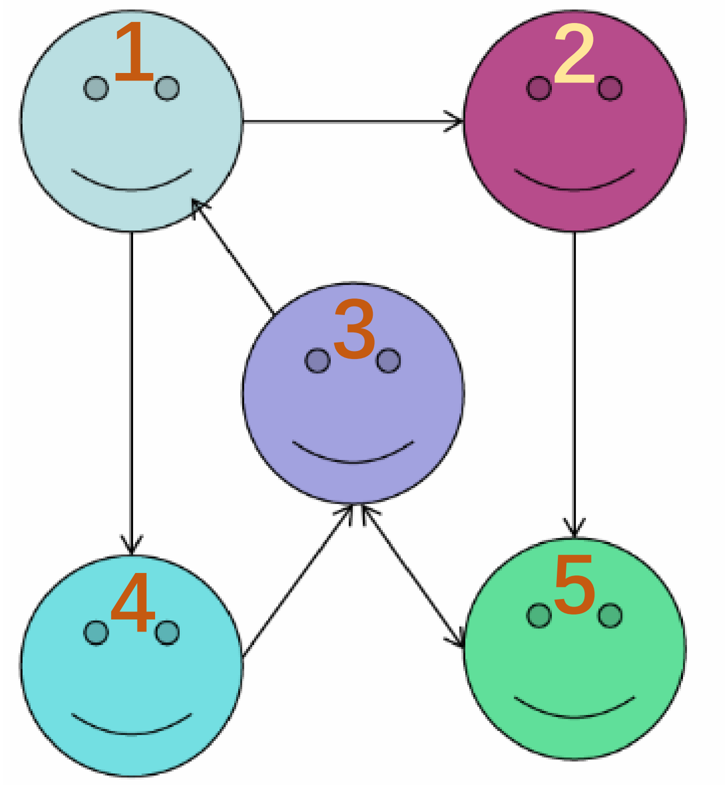
```
struct Edge {
    Node* source, * destination;
};
```



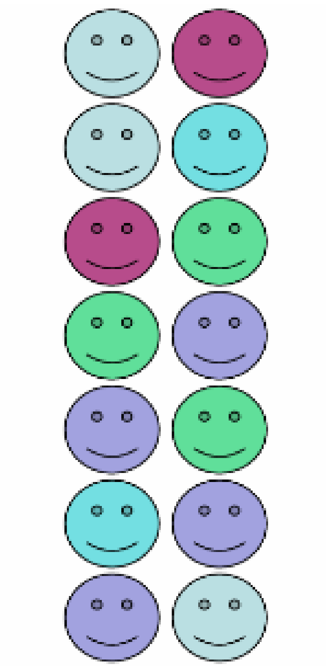


Edge List Represented as Unique Integer IDs

```cpp
// Each edge stores {source, destination}
vector< vector<int> > edges = {{1,2},{1,4},{2,5},{5,3},{3,5},{4,3},{3,1}};
```



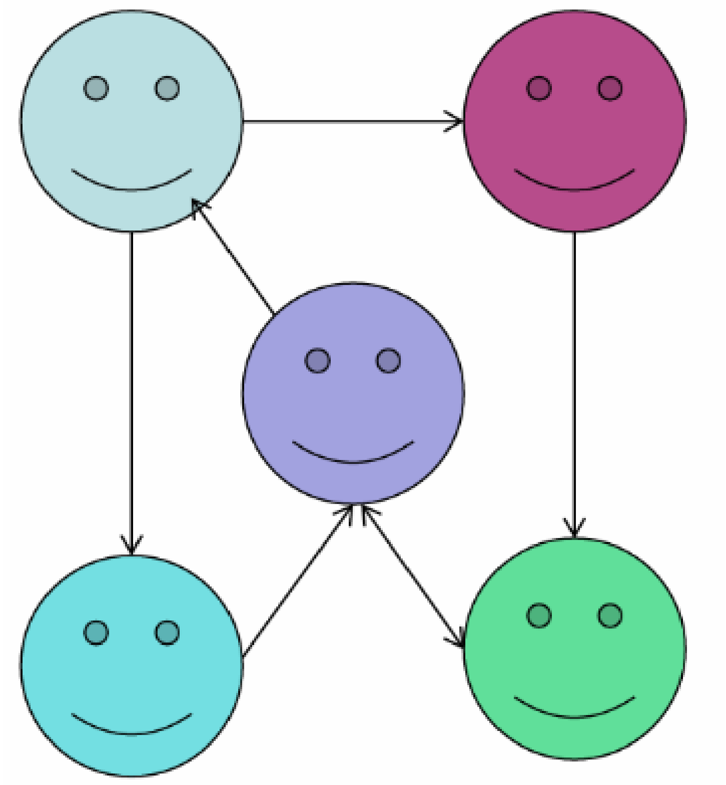
vector<int>

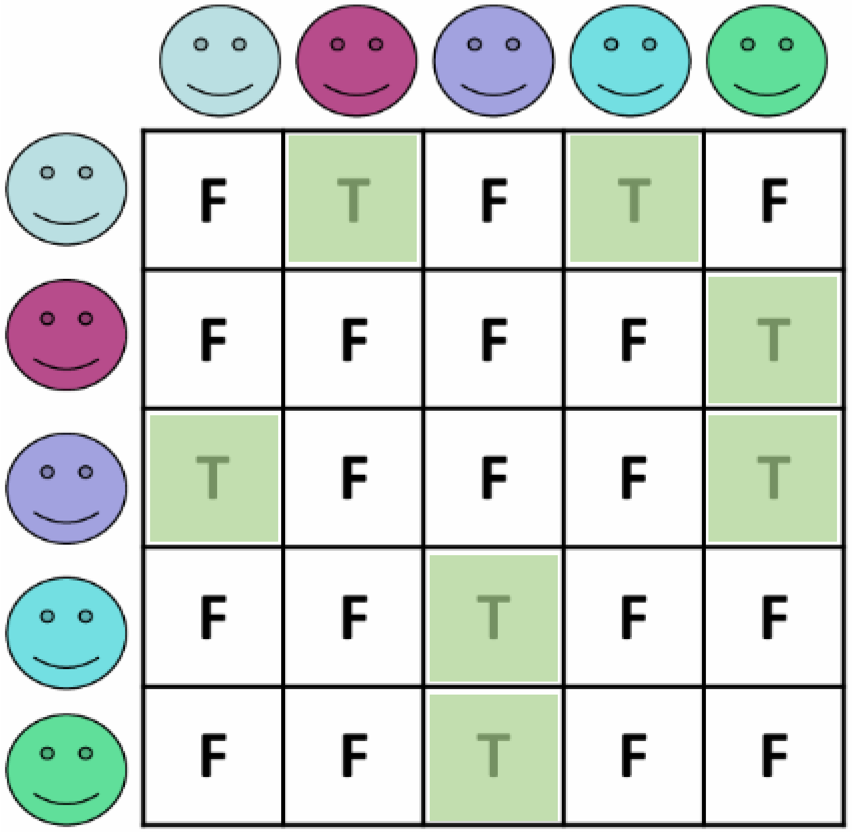


{1, 2}
{1, 4}
{2, 5}
{5, 3}
{3, 5}
{4, 3}
{3, 1}

Adjacency Matrix

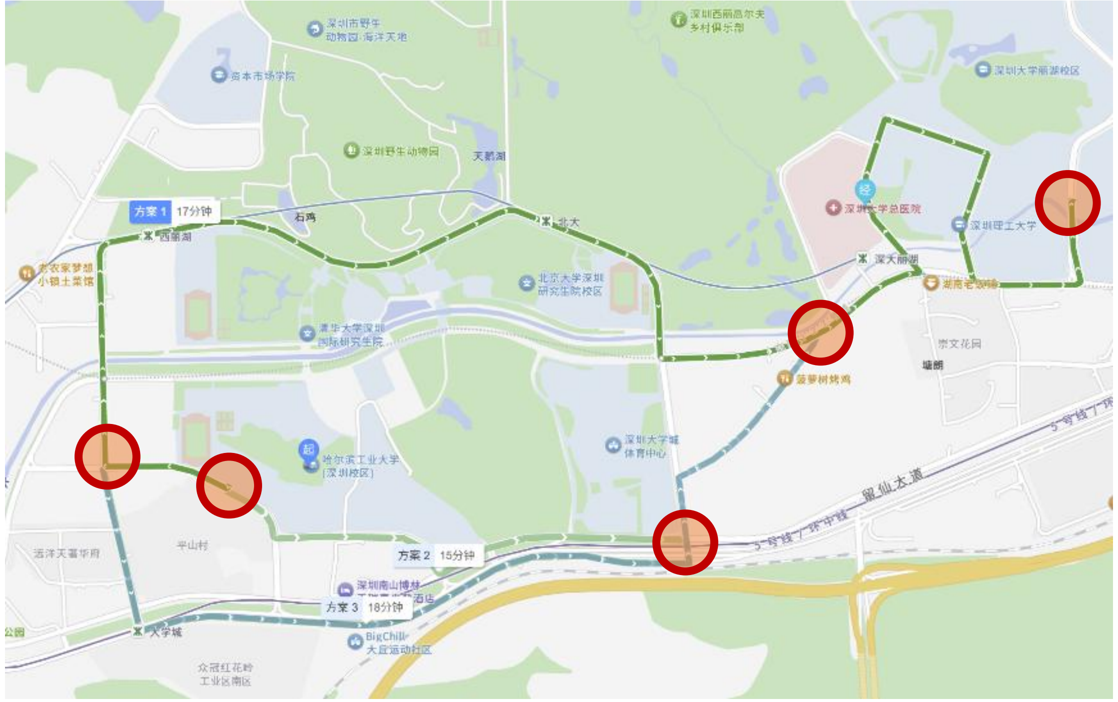
- Store a **boolean** grid, rows/columns correspond to nodes
  - Alternative to Adjacency List





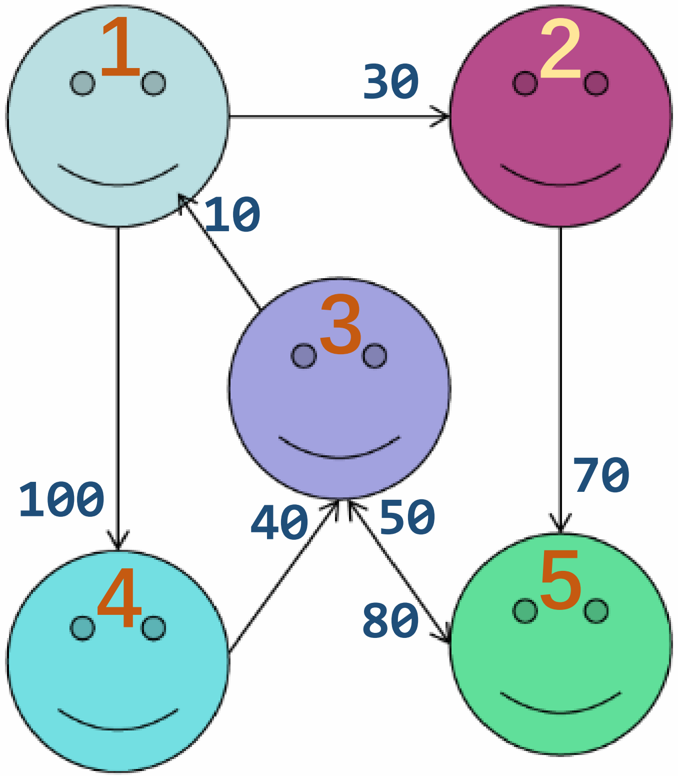
Edge Properties

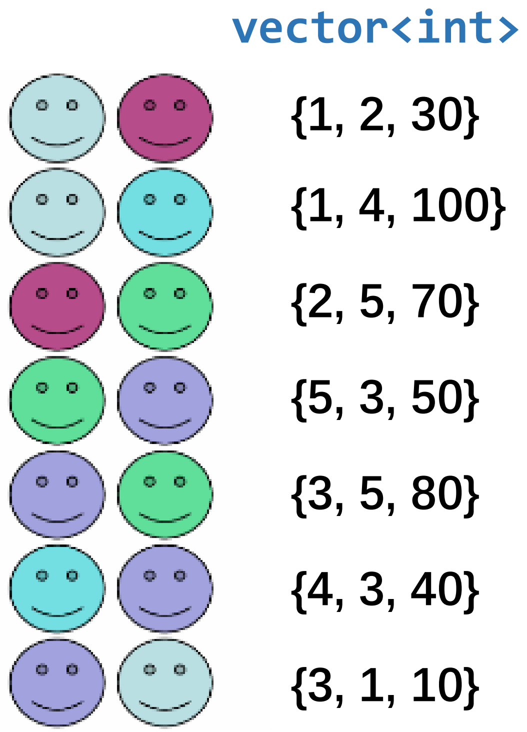
- Not all edges are created equally
  - Some have greater **weight**
- Real life examples:
  - Road toll
  - Miles on a road
  - Time spent on a road
- Store a number with each edge corresponding to its **weight**



Edge List with Weights

```
// Each edge stores {source, destination, weight}
vector< vector<int> > edges =
{{1,2,30},{1,4,100},{2,5,70},{5,3,50},{3,5,80},{4,3,40},{3,1,10}};
```





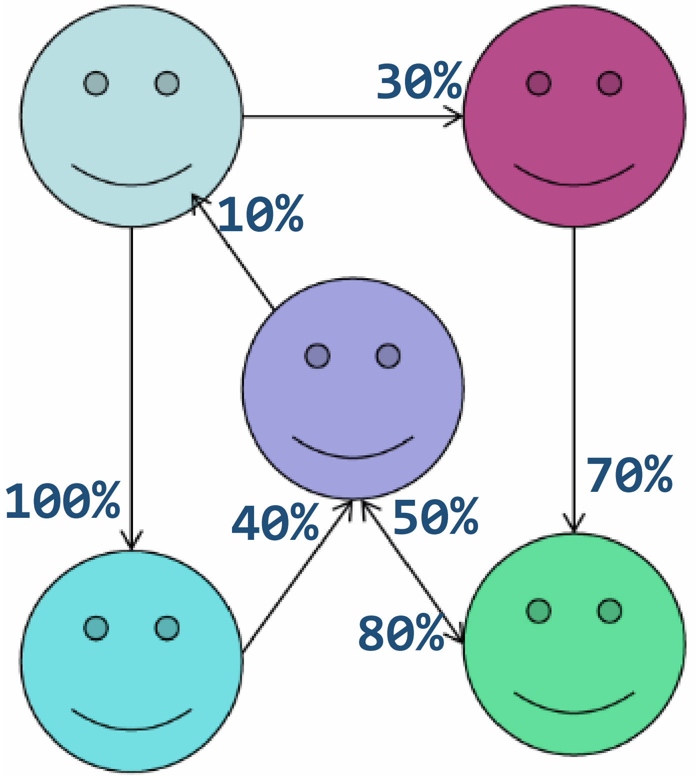
(Here, the weights represent the intimacy scores)

Adjacency Matrix with Weights

- Store an **int**/**float** grid, rows/columns correspond to nodes
- In this example, each float number represents the weight

T ∈ (0.0, 1.0]

F = 0.0

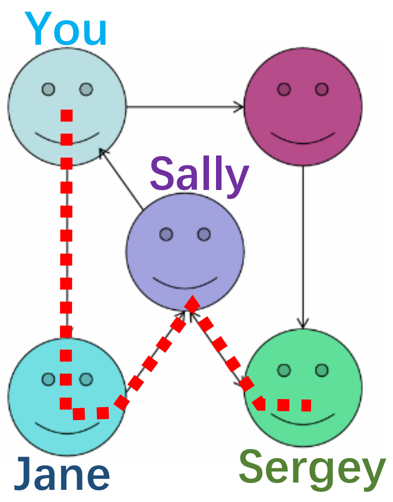


0.0	0.3	0.0	1.0	0.0
0.0	0.0	0.0	0.0	0.7
0.1	0.0	0.0	0.0	0.8
0.0	0.0	0.4	0.0	0.0
0.0	0.0	0.5	0.0	0.0

(Here, the weights represent the intimacy scores in percentage)

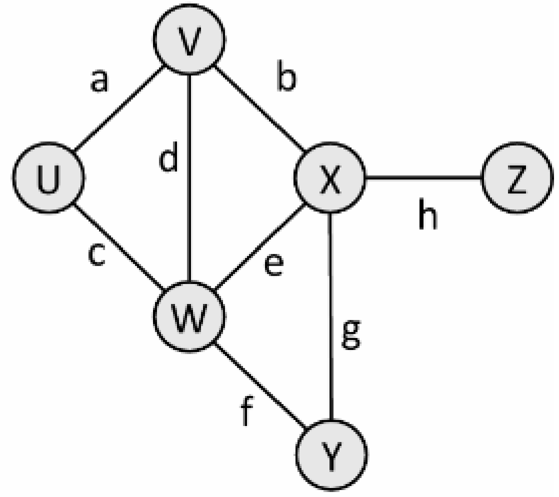
Paths

- I want to find this word on a board made of letters "next to" each other
- I want a job at Google. Do I know anyone who works there? What about someone who knows someone?
- A **path** is a sequence of nodes with edges between them connecting two nodes
- Could **store edges** instead of nodes
- You know Jane. Jane knows Sally. Sally knows knows Sergey Brin, the founder of Google, so the path is: You → Jane → Sally → Sergey

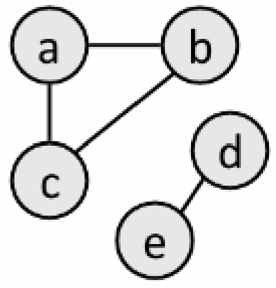


Other graph properties

- **Reachable**: Vertex u is reachable from v if a path exists from u to v.

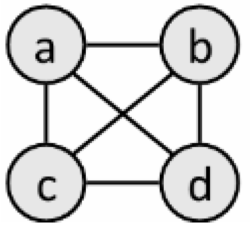


- **Connected**: An **undirected graph** is connected if every vertex is reachable from every other.



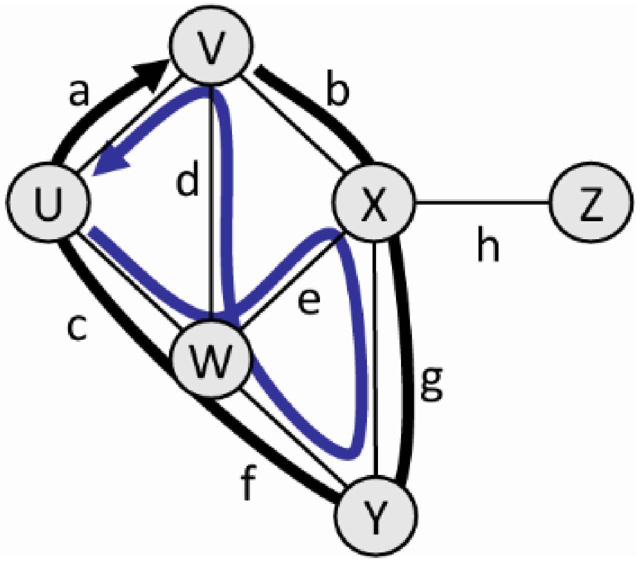
A **directed graph** is said to be **strongly connected** if for every pair of nodes u and v, there is a directed path from u to v, and vice-versa.
A **directed graph** is said to be **weakly connected** (or, more simply, connected) if the corresponding undirected graph is connected

- **Complete**: If every vertex has a direct edge to every other.



Loops and cycles

- **Cycle**: A path that **begins and ends at the same node**.
  - Example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - Example: {c, d, a} or {U, W, V, U}.
  - Acyclic graph: One that does not contain any cycles.
- **Loop**: An edge directly **from a node to itself**.
  - Many graphs don't allow loops.



Types of Graphs

- Boggle game (Letters on board)?

– undirected, unweighted, cyclic, connected

- A molecule?

– undirected, potentially weighted, potentially cyclic, connected

- A map of flights?

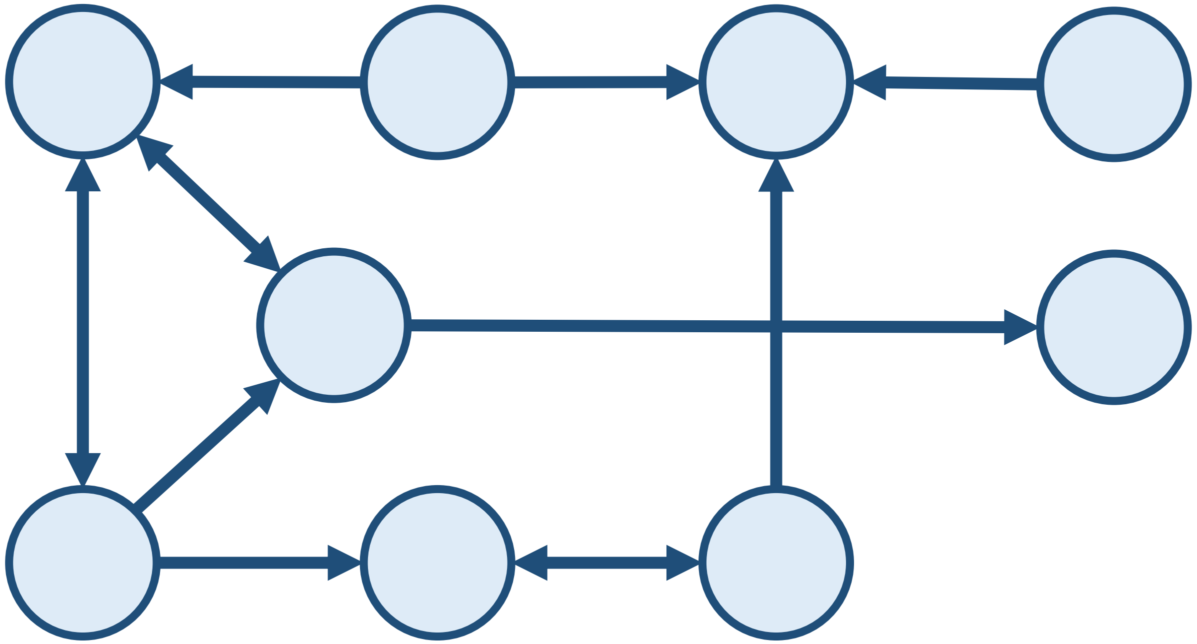
– directed, potentially weighted, potentially cyclic, perhaps not connected

Finding Paths

- Easiest way: **Depth-First Search (DFS)**
  - Recursive backtracking!
- Finds a path between two nodes if it exists
  - Or can find all the nodes reachable from a node
- Where can I travel to, starting in HIT Shenzhen campus?
- If all my friends (and their friends, and so on) share my post, how many will eventually see it?

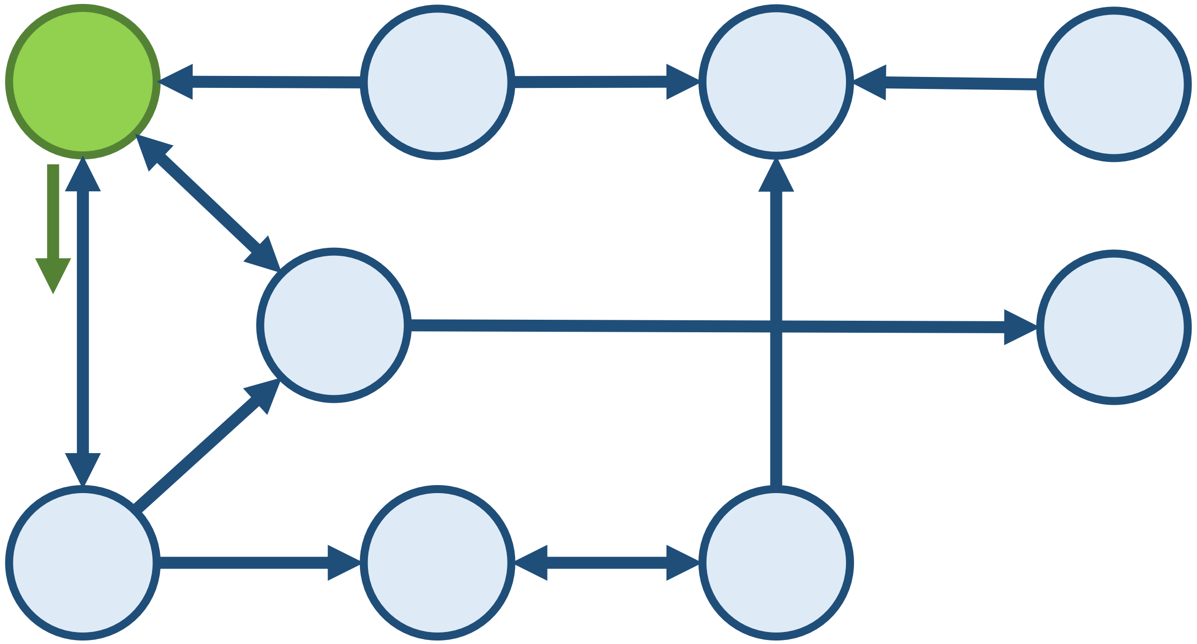
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



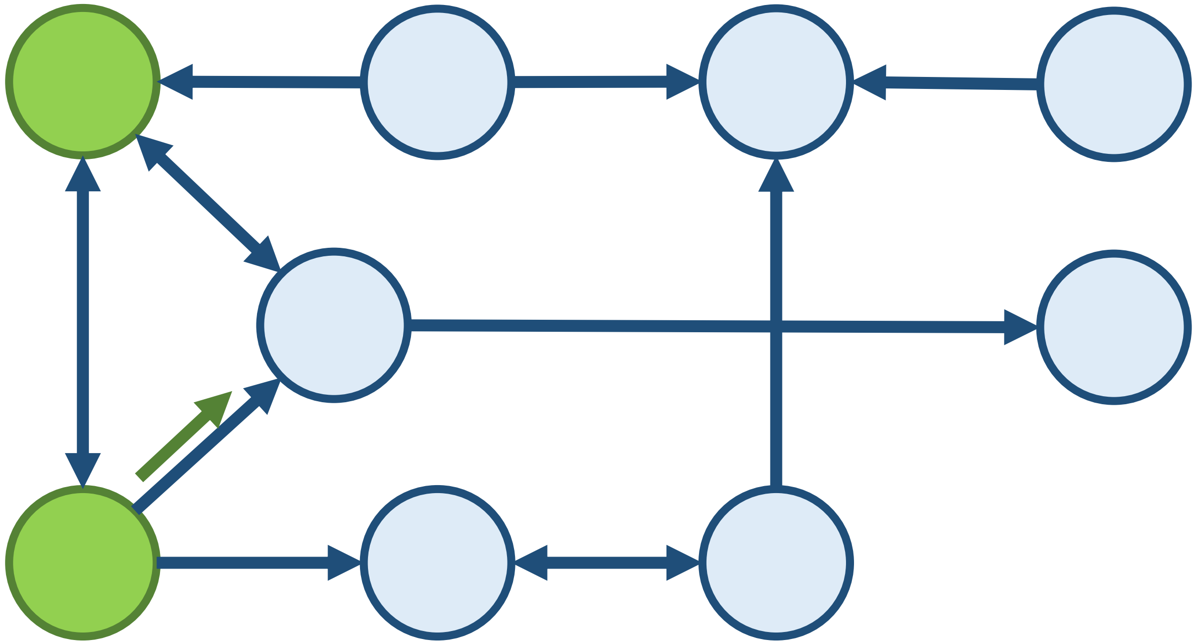
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



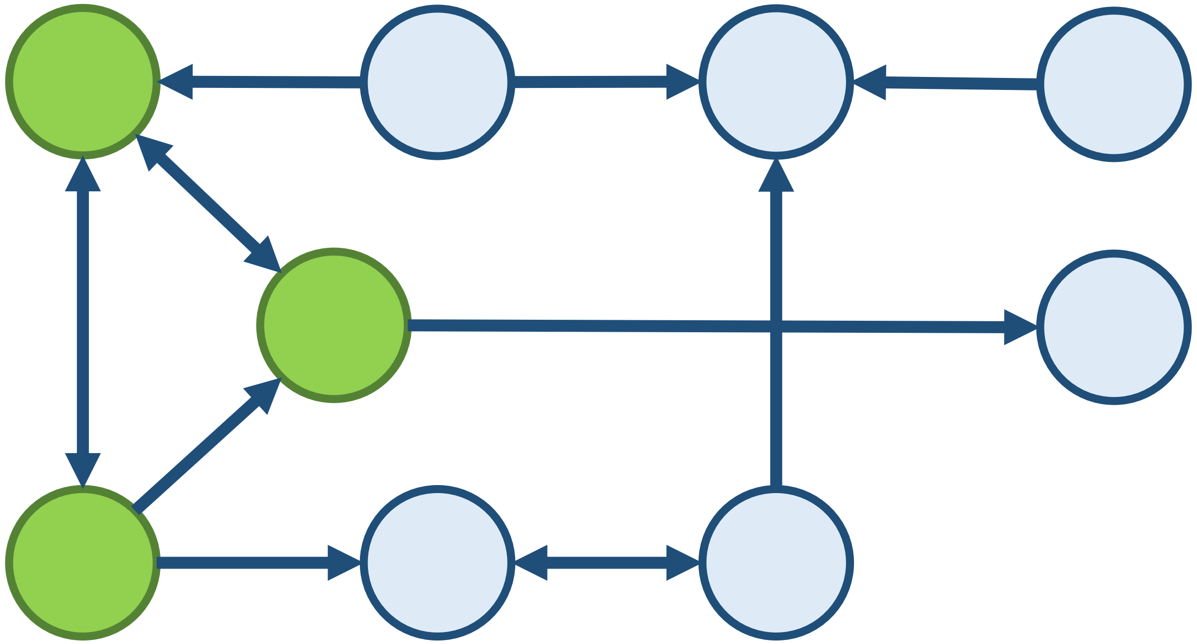
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



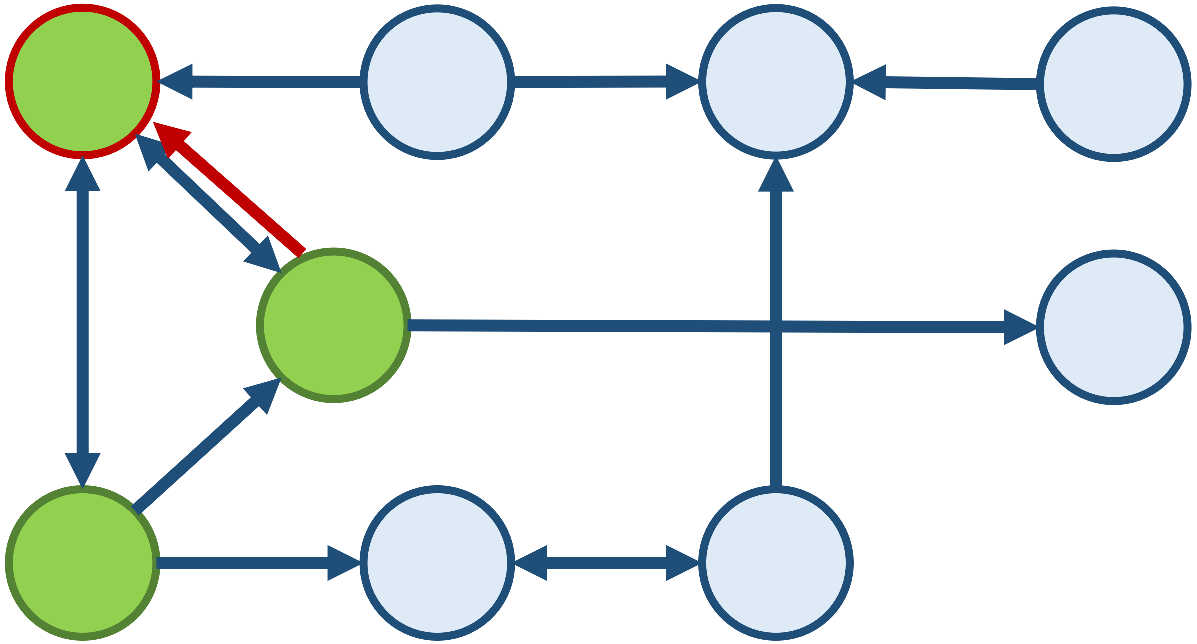
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



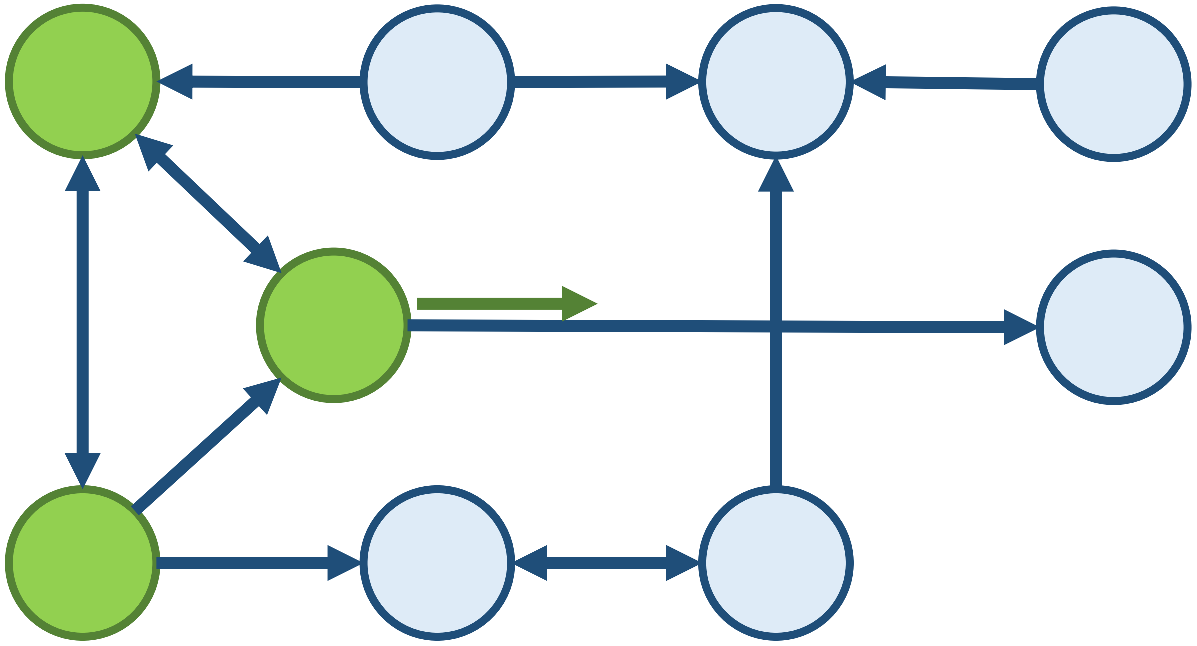
Depth-First Search

- **If we've seen the node before, stop**
- Otherwise, visit all the unvisited nodes from this node



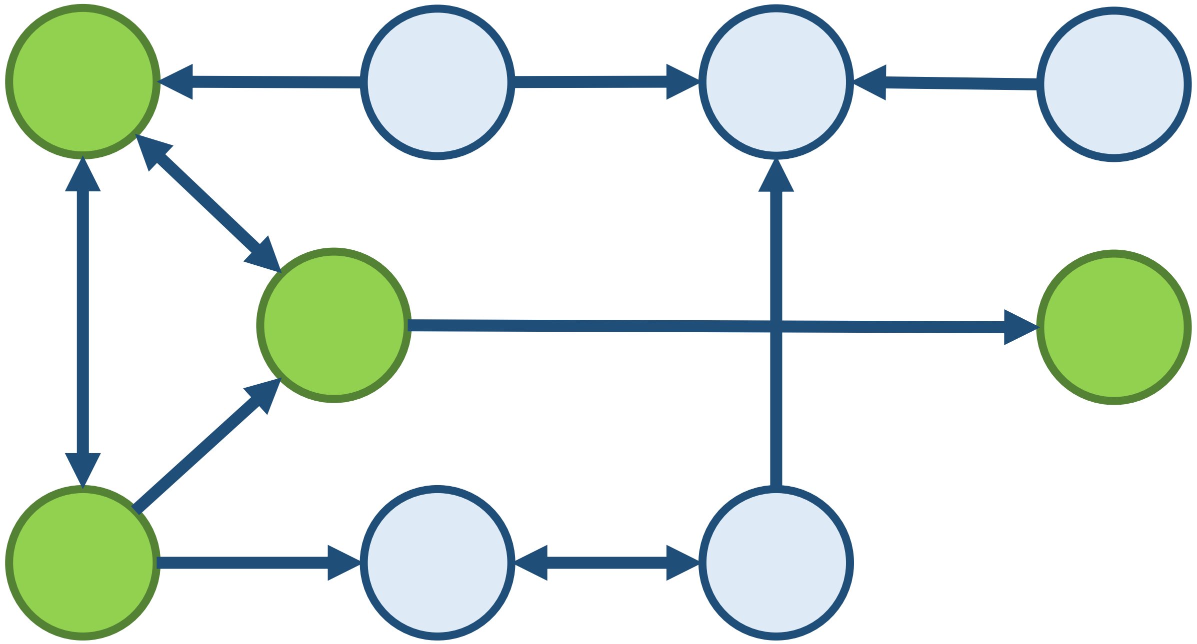
Depth-First Search

- **If we've seen the node before, stop**
- Otherwise, visit all the unvisited nodes from this node



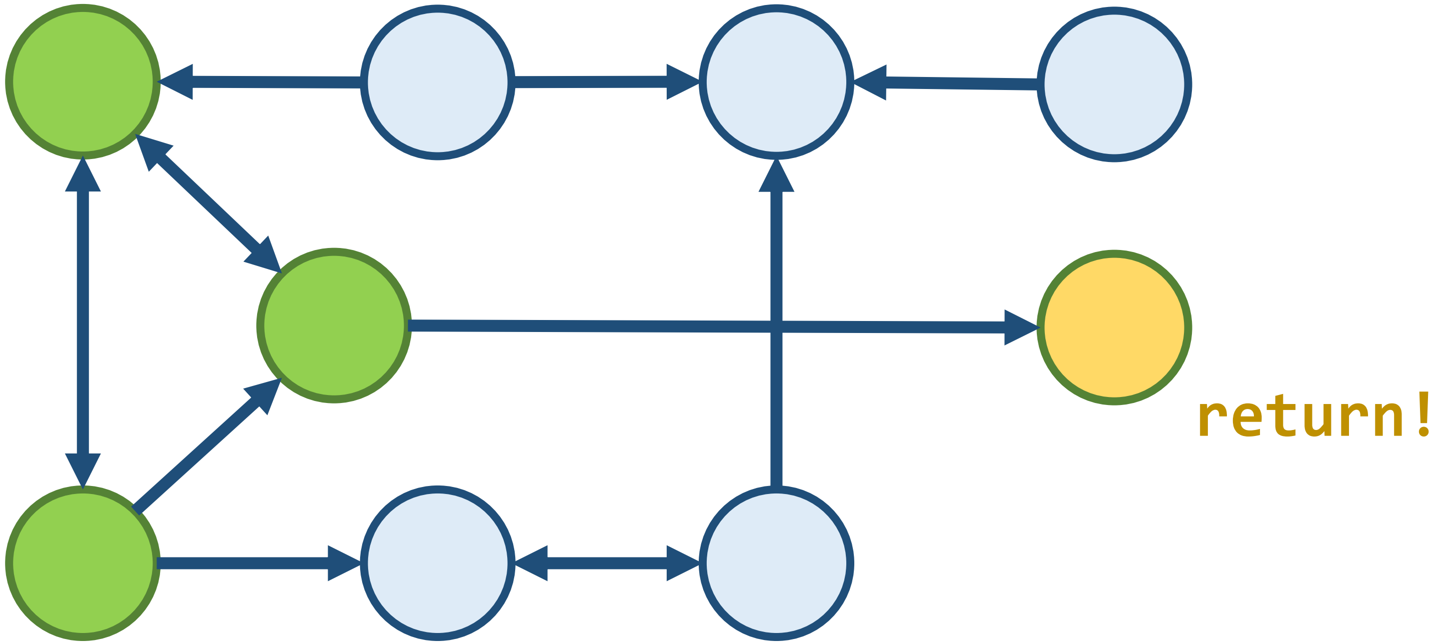
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



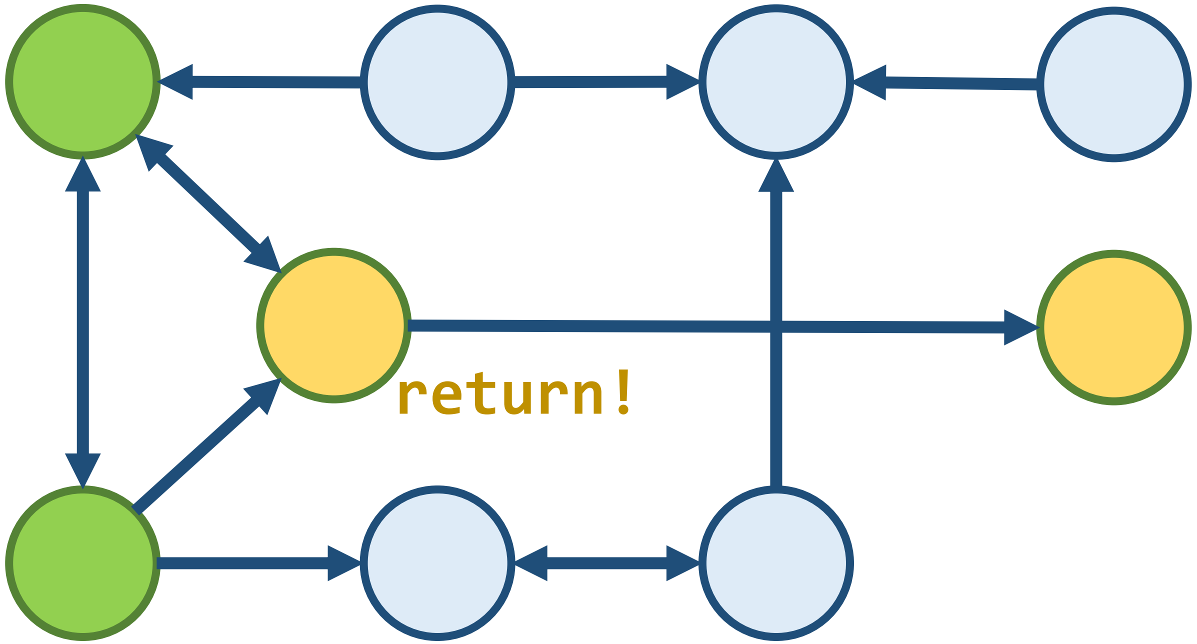
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



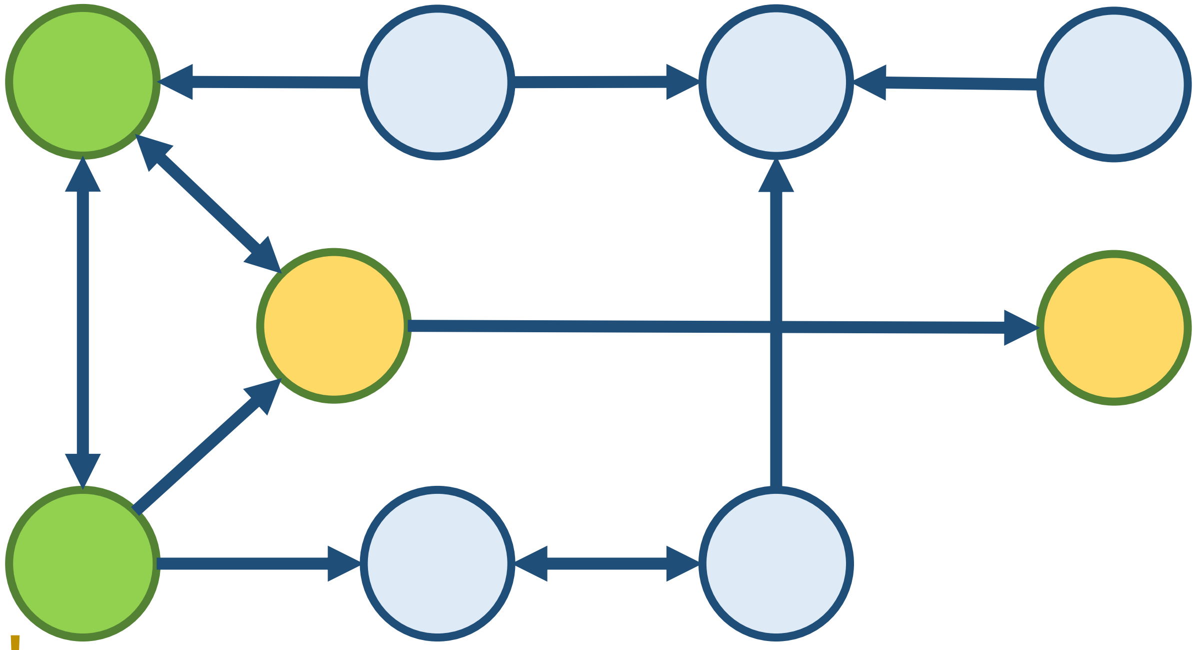
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



Depth-First Search

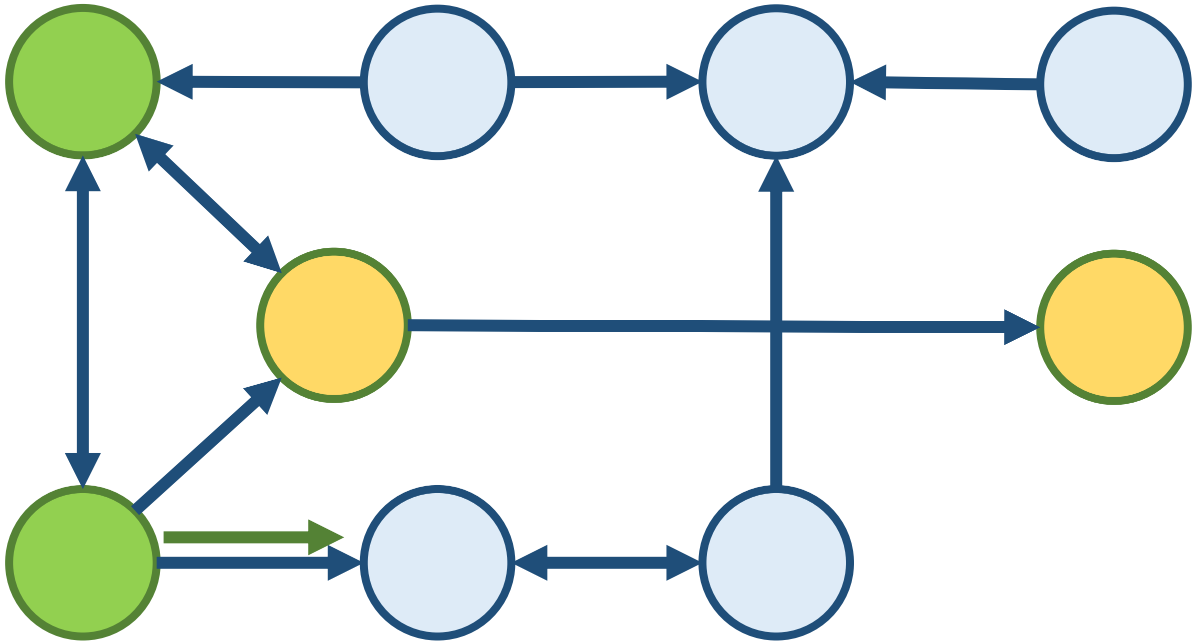
- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



`return!`

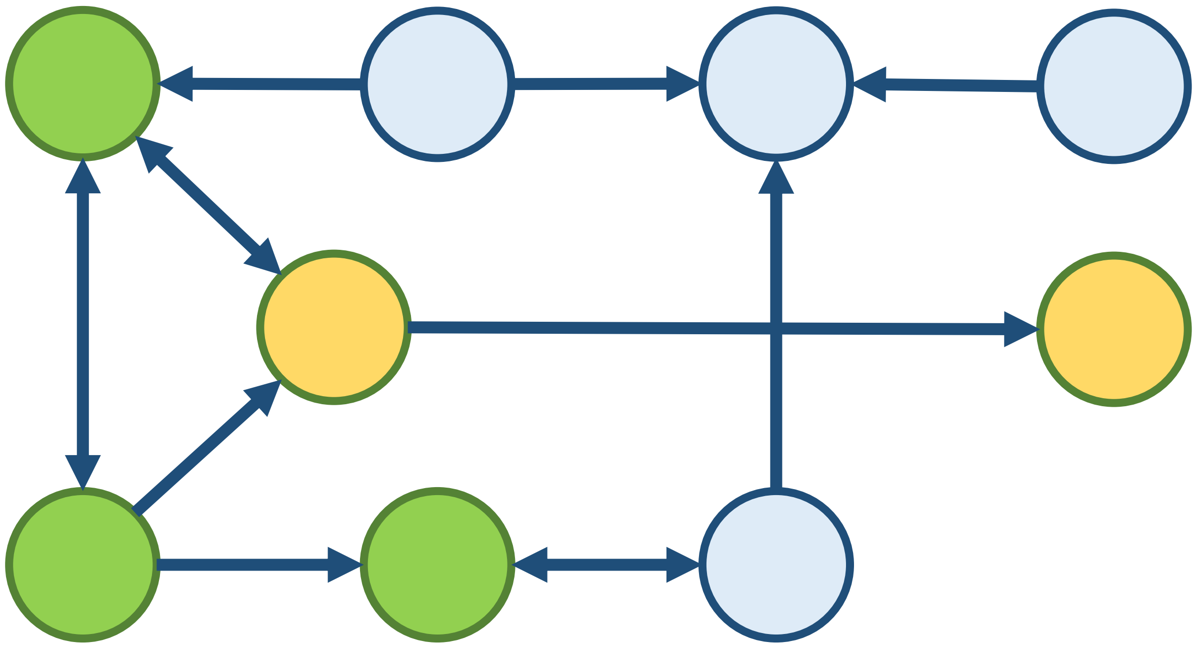
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



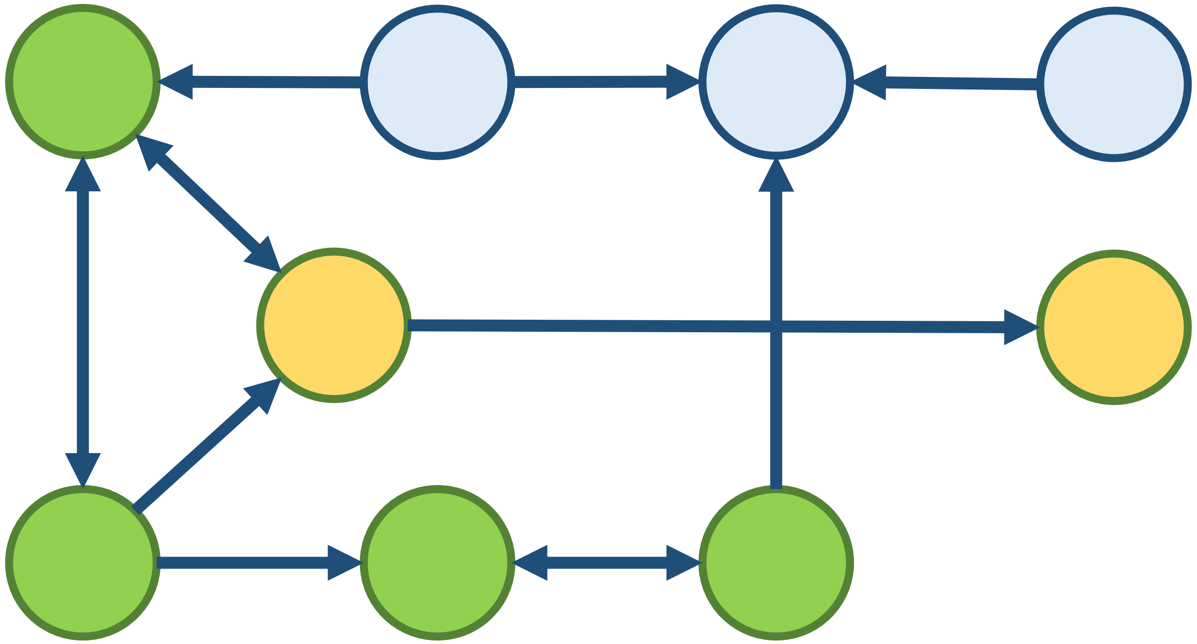
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



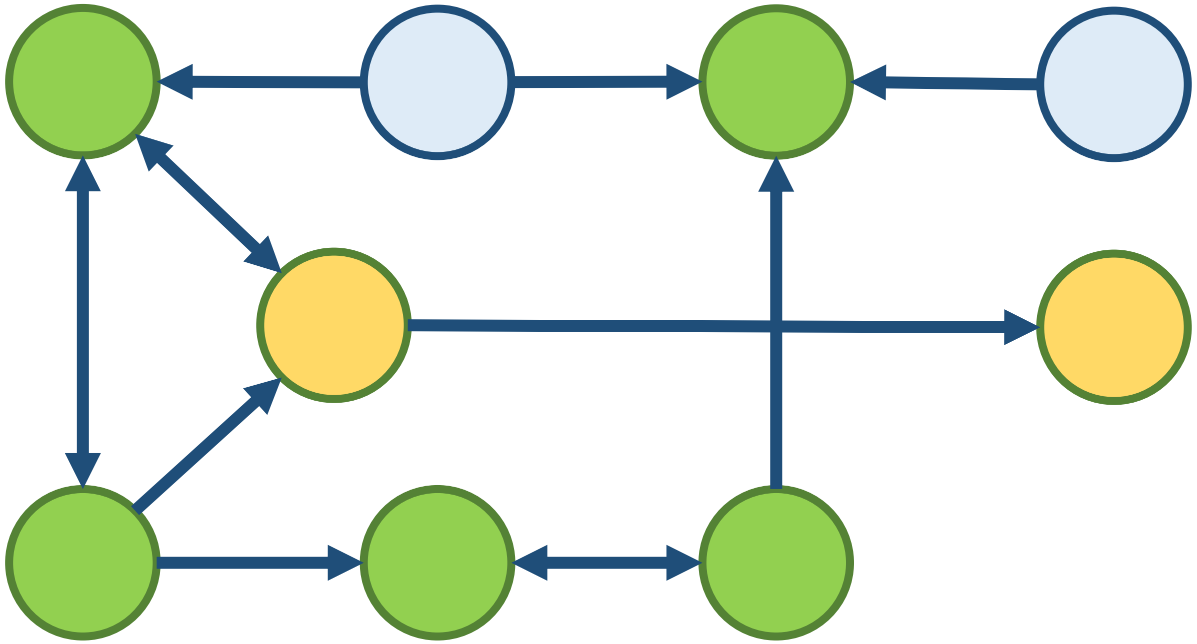
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



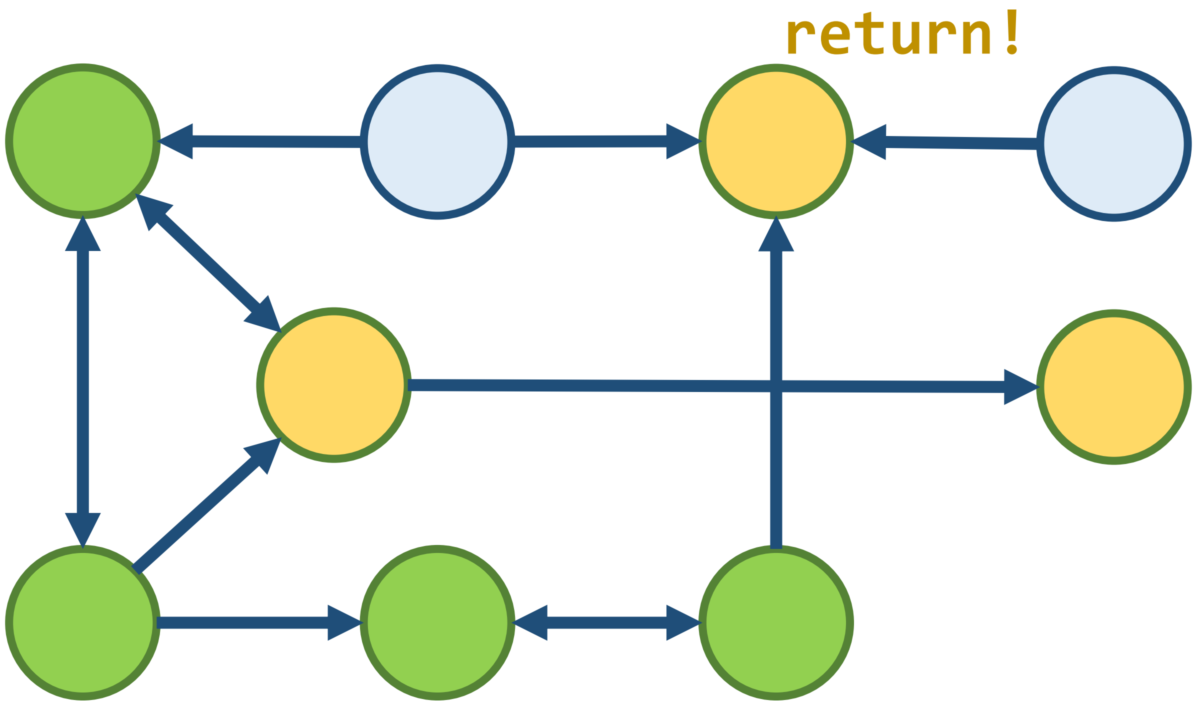
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



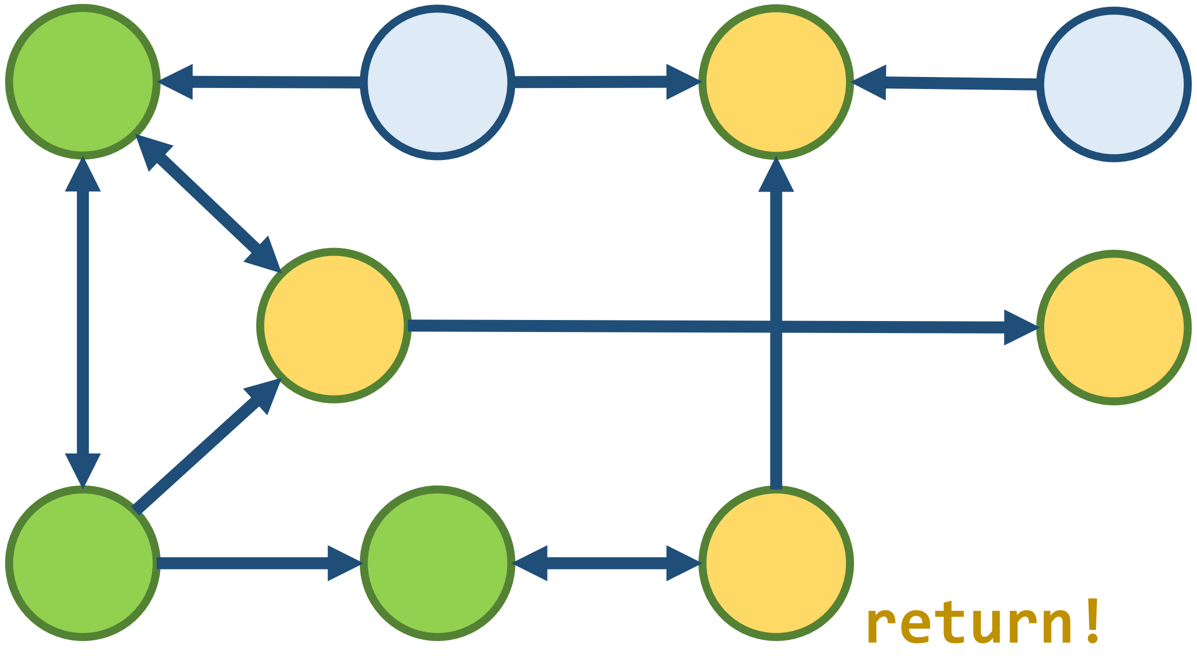
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



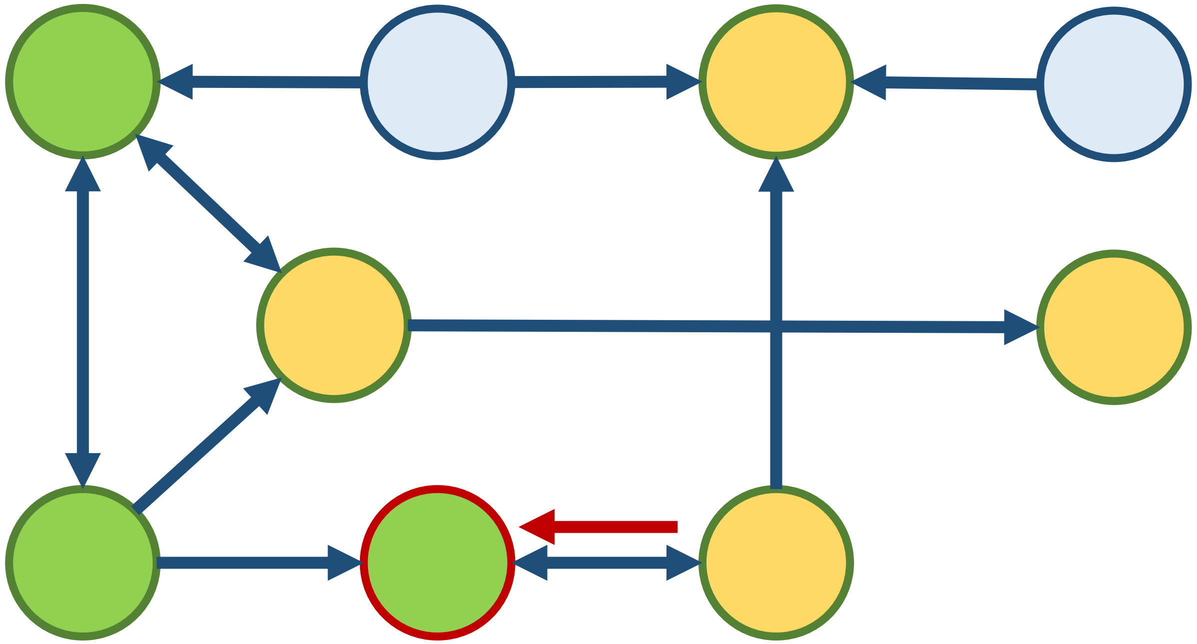
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



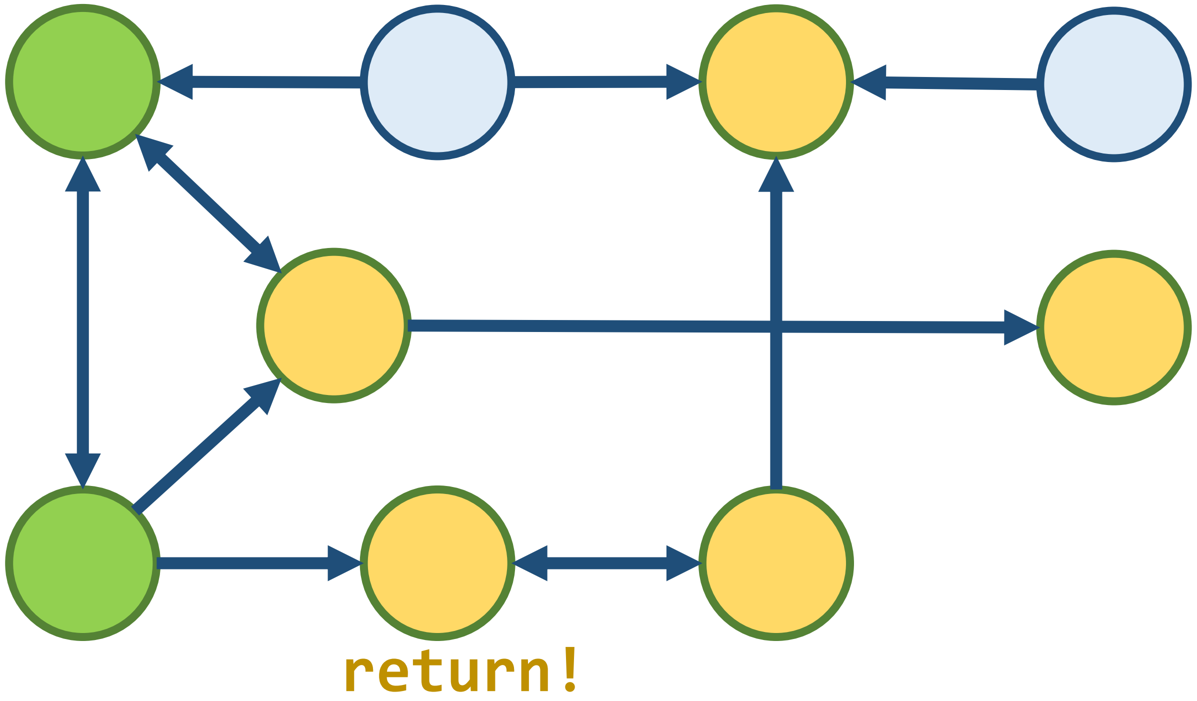
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



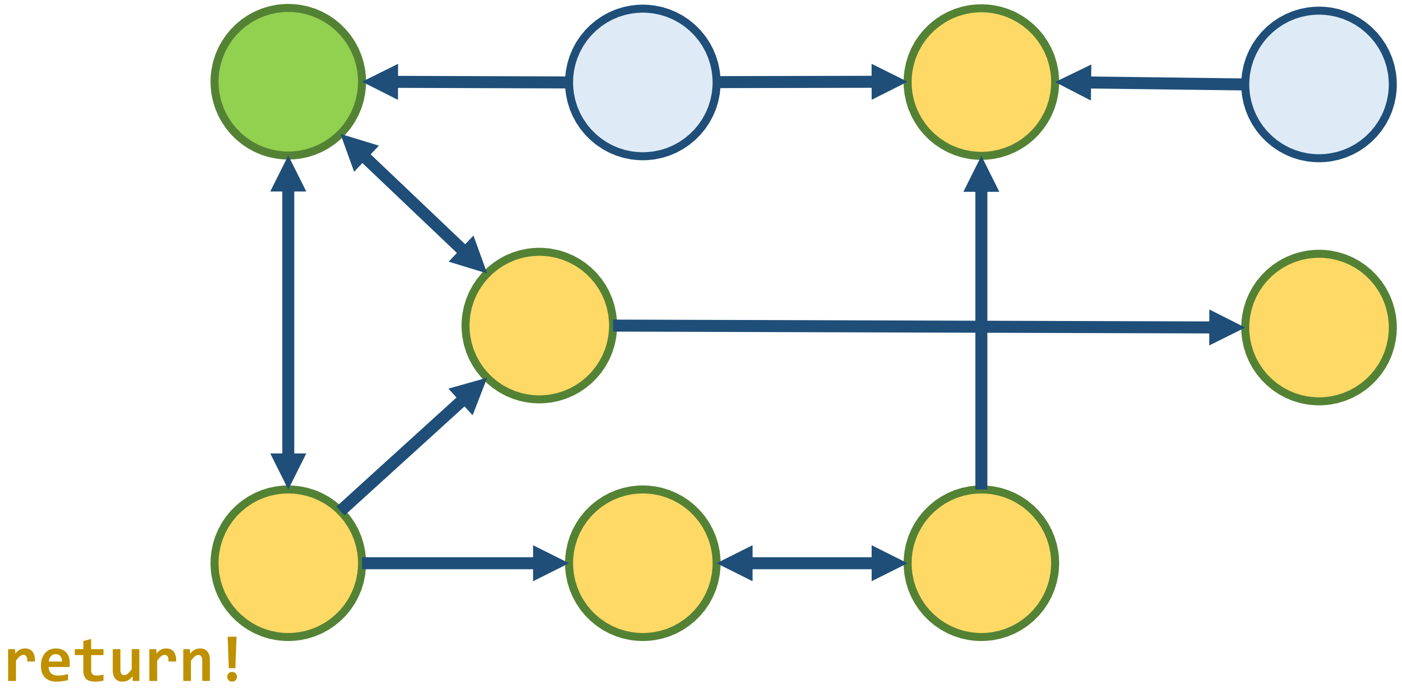
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



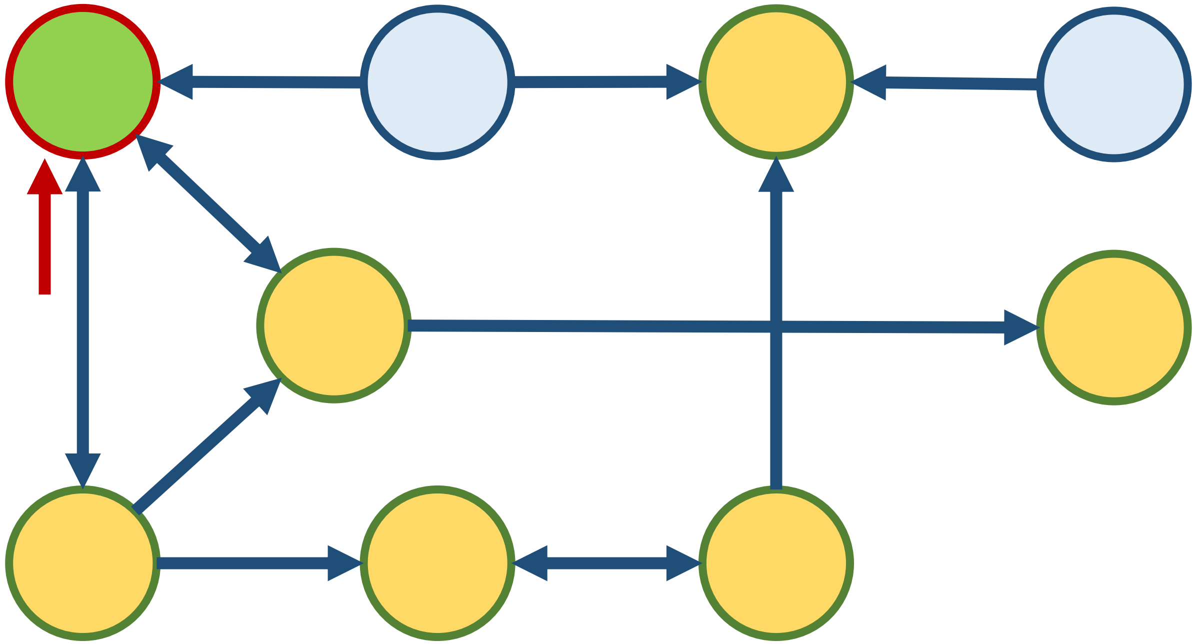
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



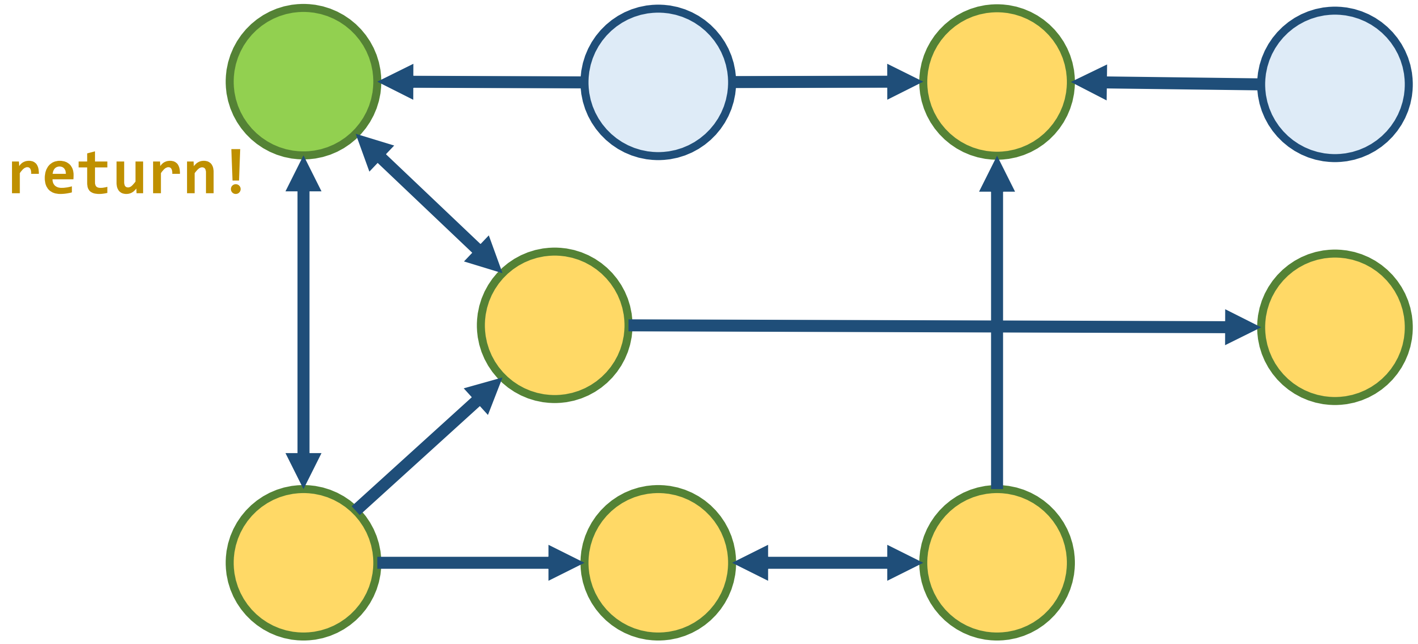
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



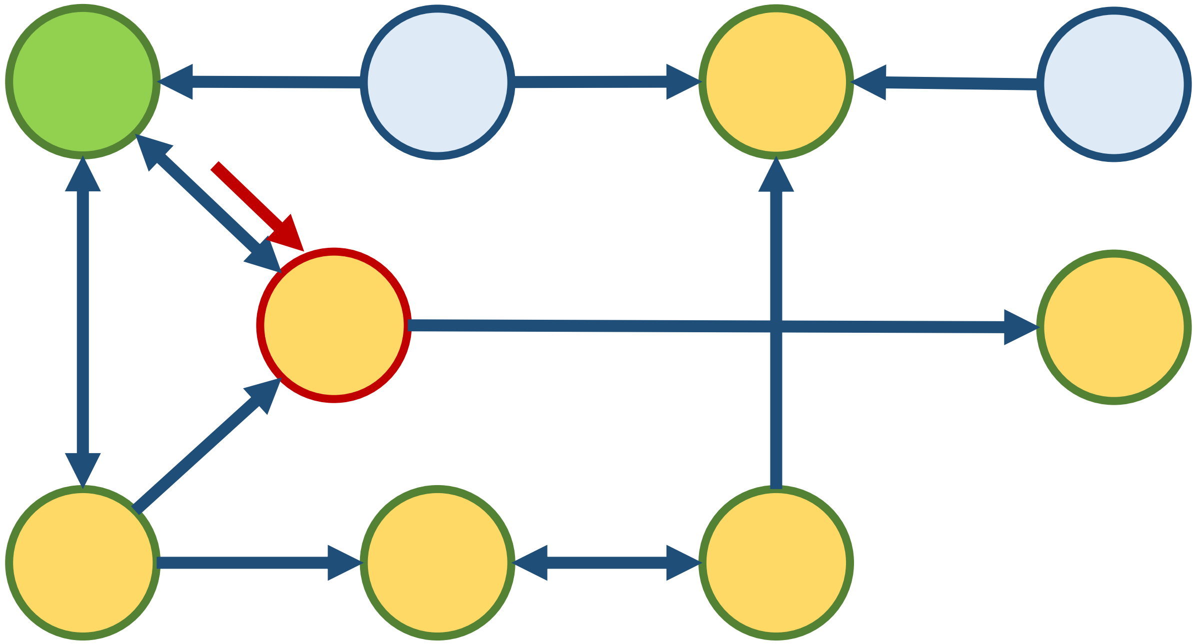
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



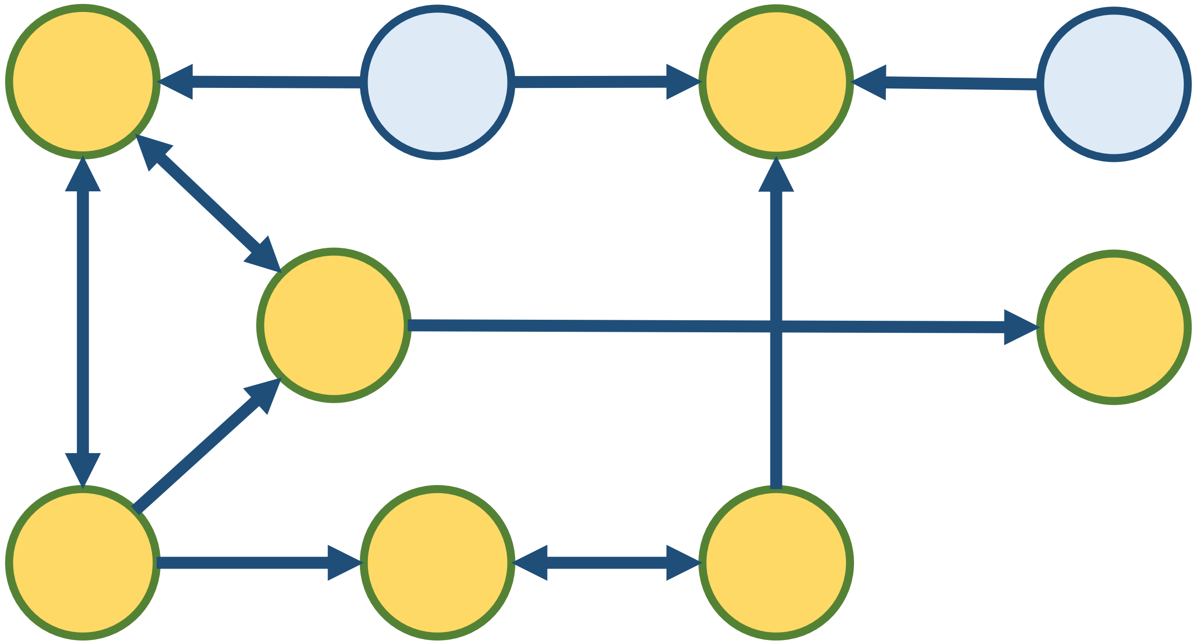
Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



Depth-First Search

- If we've seen the node before, stop
- Otherwise, visit all the unvisited nodes from this node



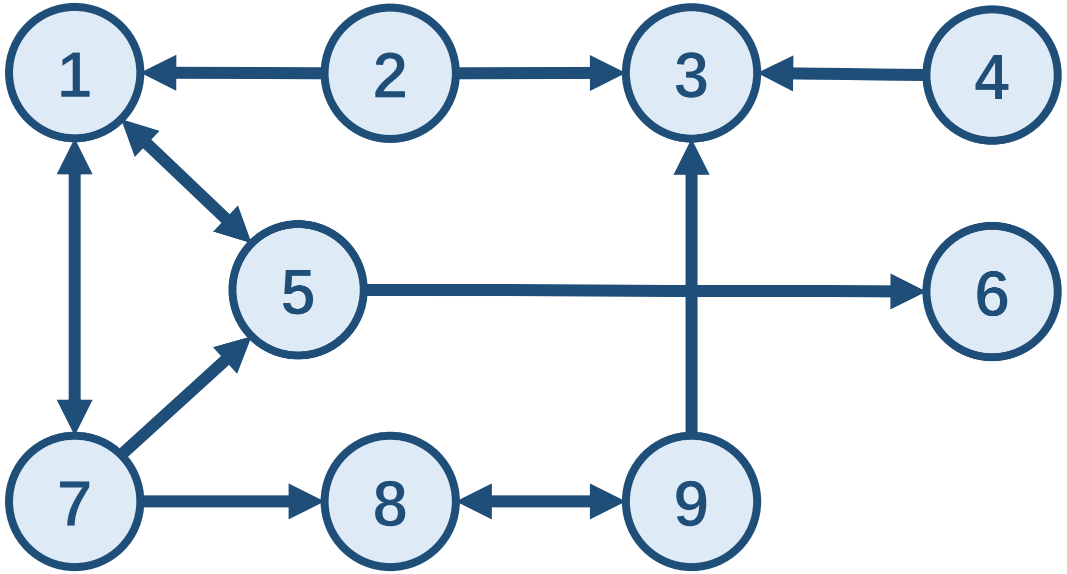
Depth-First Search Implementation

- In an **n-node**, **m-edge** graph, takes O(m + n) time with an adjacency list
  - Visit each edge once, visit each node at most once
- Pseudocode:

```
dfs from v1:
    mark v1 as seen.
    for each of v1's unvisited neighbors n:
        dfs(n)
```

- Recursive Backtracking

Implement DFS in C++



isConnected

				1		1		
1		1						
		1						
1					1			
1				1			1	
								1
		1					1	

```cpp
void dfs(int node, vector<vector<bool>>& isConnected, vector<bool>& visit) {
    visit[node] = true;
    for (int i = 0; i < isConnected.size(); i++) {
        if (isConnected[node][i] && !visit[i]) {
            dfs(i, isConnected, visit);
        }
    }
}
```

visit

In-Class Exercise: City Conquer Game

- There are N cities. Some of them are connected, some are not
- A kingdom can be built by deploy an army to any city, and conquer neighboring cities one after another. Cities not reachable by the army cannot be conquered.
- You are given an N x N matrix isConnected where $\text{isConnected}[\text{i}][\text{j}] = 1$ if the i-th city and the j-th city are directly connected, and $\text{isConnected}[\text{i}][\text{j}] = 0$ otherwise.
- Return the **total number armies needed to conquer all cities** in this map.


```
int conquerAll(vector<vector<int>>& isConnected) {

}
```

	1	2	3	4	5	6
1	1	1				
2	1	1			1	
3			1	1		
4			1	1		
5		1			1	
6						1

City Conquer Game

```
int conquerAll(vector<vector<int>>& isConnected) {
// Implement your algorithm
}

// Testing Code
int main(){vector<vector<int>> map1 = {
        { 1, 1, 0, 0, 0, 0 },
        { 1, 1, 0, 0, 1, 0 },
        { 0, 0, 1, 1, 0, 0 },
        { 0, 0, 1, 1, 0, 0 },
        { 0, 1, 0, 0, 1, 0 },
        { 0, 0, 0, 0, 0, 1 },
    };

    cout << "Map 1 needs " << conquerAll(map1) << "
armies\n";
}
```

```
vector<vector<int>> map2 = {
        { 1, 0, 0, 1, 0, 1, 0, 0},
        { 0, 1, 1, 0, 0, 0, 0, 0 },
        { 0, 1, 1, 0, 0, 0, 0, 0 },
        { 1, 0, 0, 1, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 1, 0, 1, 0 },
        { 1, 0, 0, 0, 0, 1, 0, 0 },
        { 0, 0, 0, 0, 1, 0, 1, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 1 },
    };
```

```
vector<vector<int>> map3 = {
        { 1, 1, 0, 0 },
        { 1, 1, 1, 0 },
        { 0, 1, 1, 1 },
        { 0, 0, 1, 1 },
    };
```

City Conquer Game

```
void dfs(int node, vector<vector<int>>& isConnected, vector<bool>& visit) {
    visit[node] = true;
    for (int i = 0; i < isConnected.size(); i++) {
        if (isConnected[node][i] && !visit[i]) {
            dfs(i, isConnected, visit);
        }
    }
}
```

```
int conquerAll(vector<vector<int>>& isConnected) {
    int n = isConnected.size();
    vector<bool> visit(n);
    int numberOfArmies = 0;
    for (int i = 0; i < n; i++) {
        if (!visit[i]) {
            numberOfArmies++;
            dfs(i, isConnected, visit);
        }
    }
    return numberOfArmies;
}
```

Finding Shortest Paths

- We can find paths between two nodes, but what about the **shortest path**?
    - Fewest number of movements to reach a target city?

- **Breadth-First Search (BFS)** allows us to find the **shortest path**

Breadth-First Search (BFS)

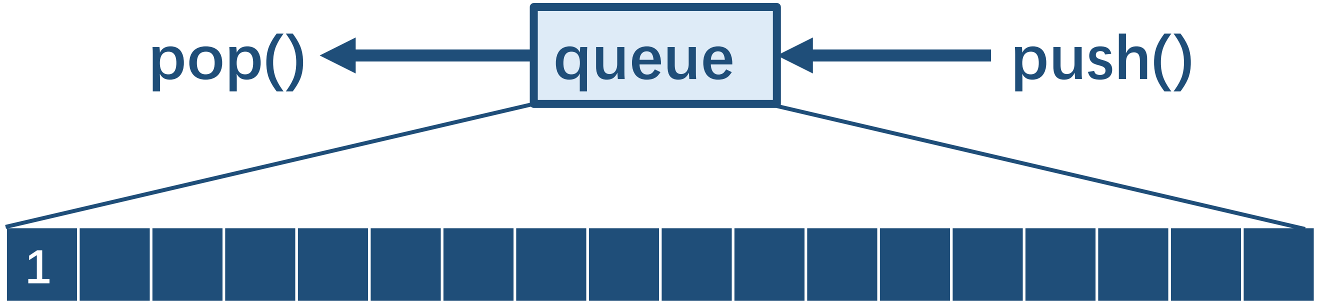
- Idea: processing a node involves visiting all its neighbors (just like DFS)
- Need to keep a TODO list of nodes to process
- Which node from our TODO list should we process first if we want the shortest path?
  - The first one we saw?
  - The last one we saw?
  - A random node?

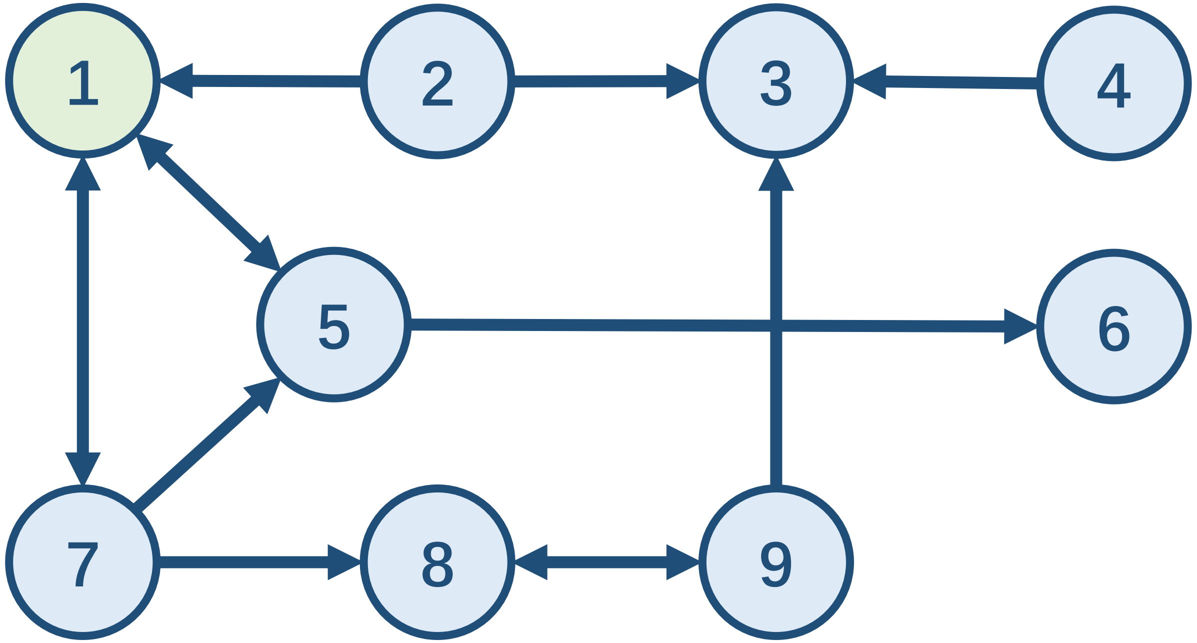
Breadth-First Search

- Keep a **Queue of nodes** as our TODO list
- Idea: **dequeue** a node, **enqueue** all its neighbors
- Still will return the same nodes as reachable, just might have shorter paths

Breadth-First Search

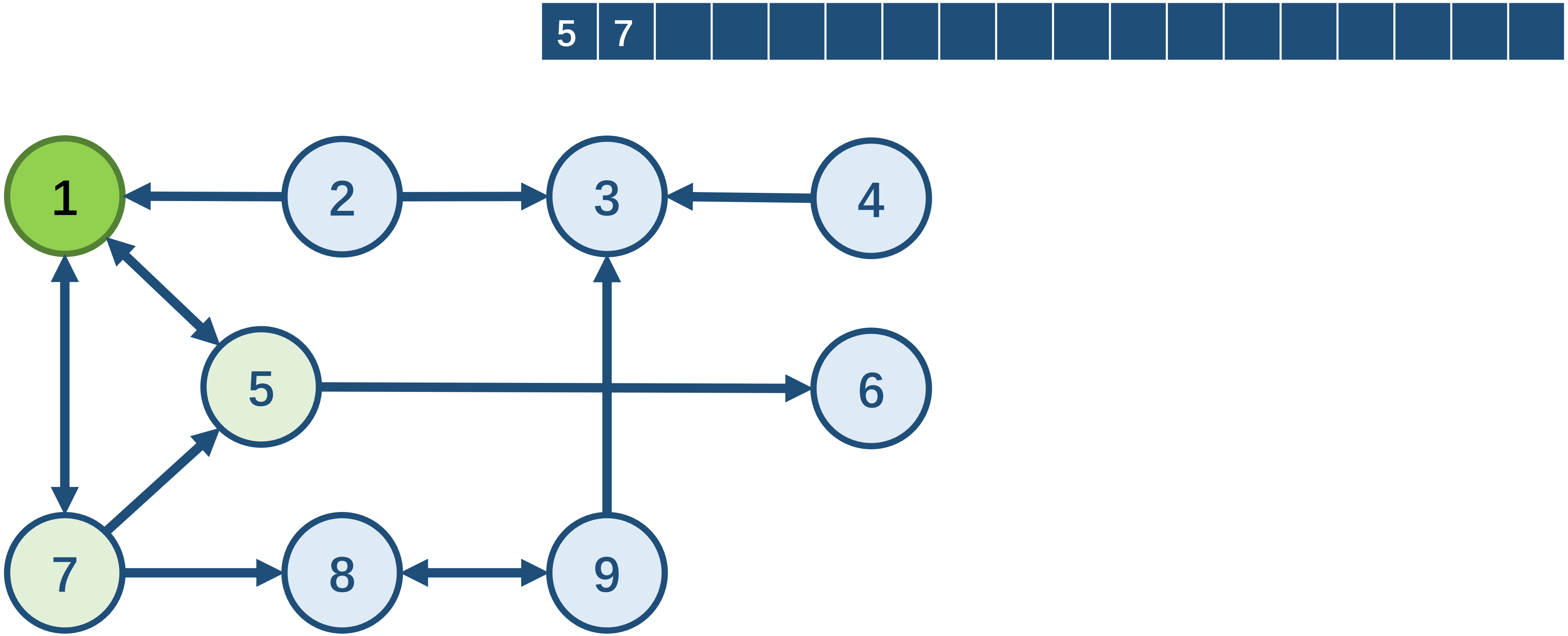
- Start from node 1
- Enqueue node 1 (and mark node 1 as seen)





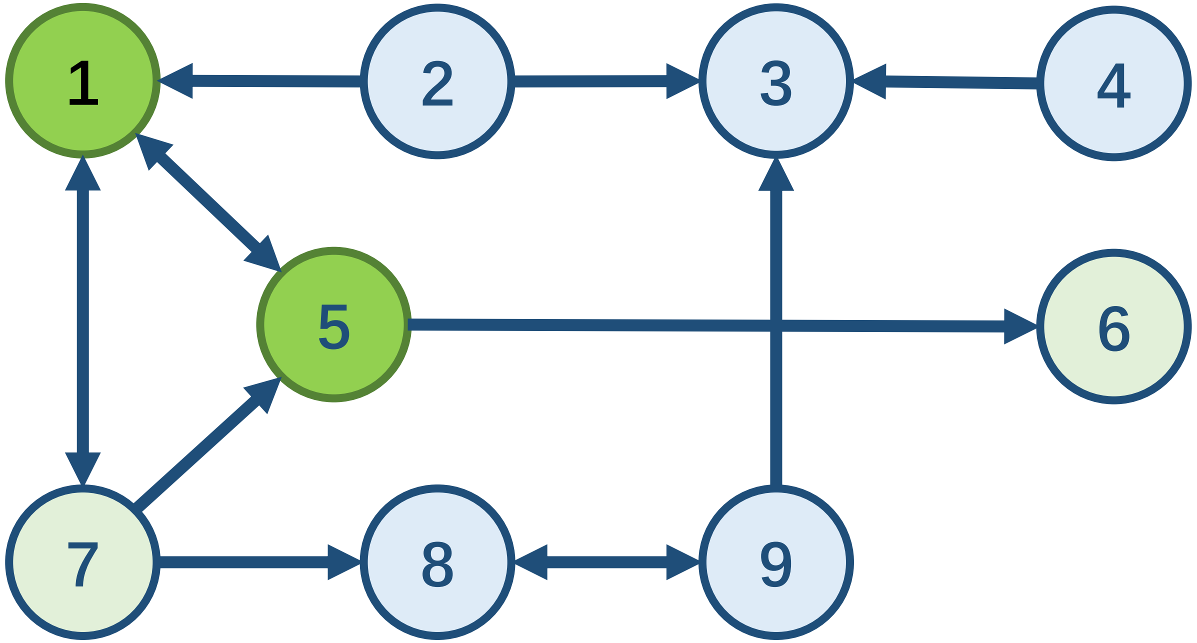
Breadth-First Search

- Dequeue node 1
- Add all its unseen neighbors to the queue and marked as seen (5, 7)



Breadth-First Search

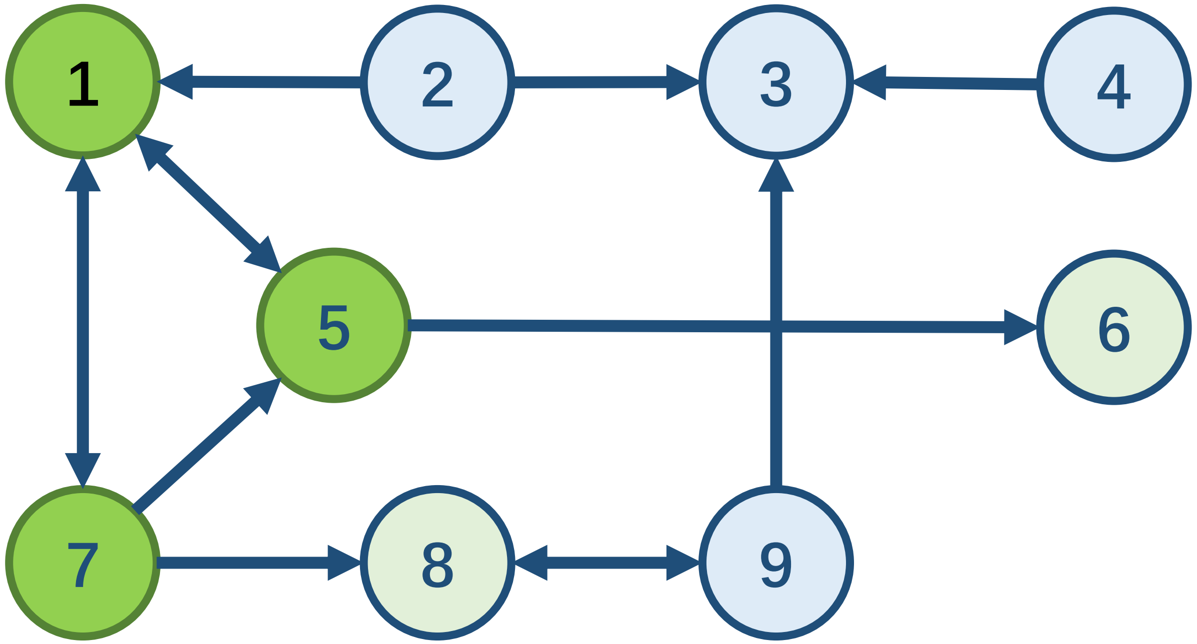
- Dequeue node 5 (and mark node 5 as seen)
- Add all its unseen neighbors to the queue (6)



Breadth-First Search

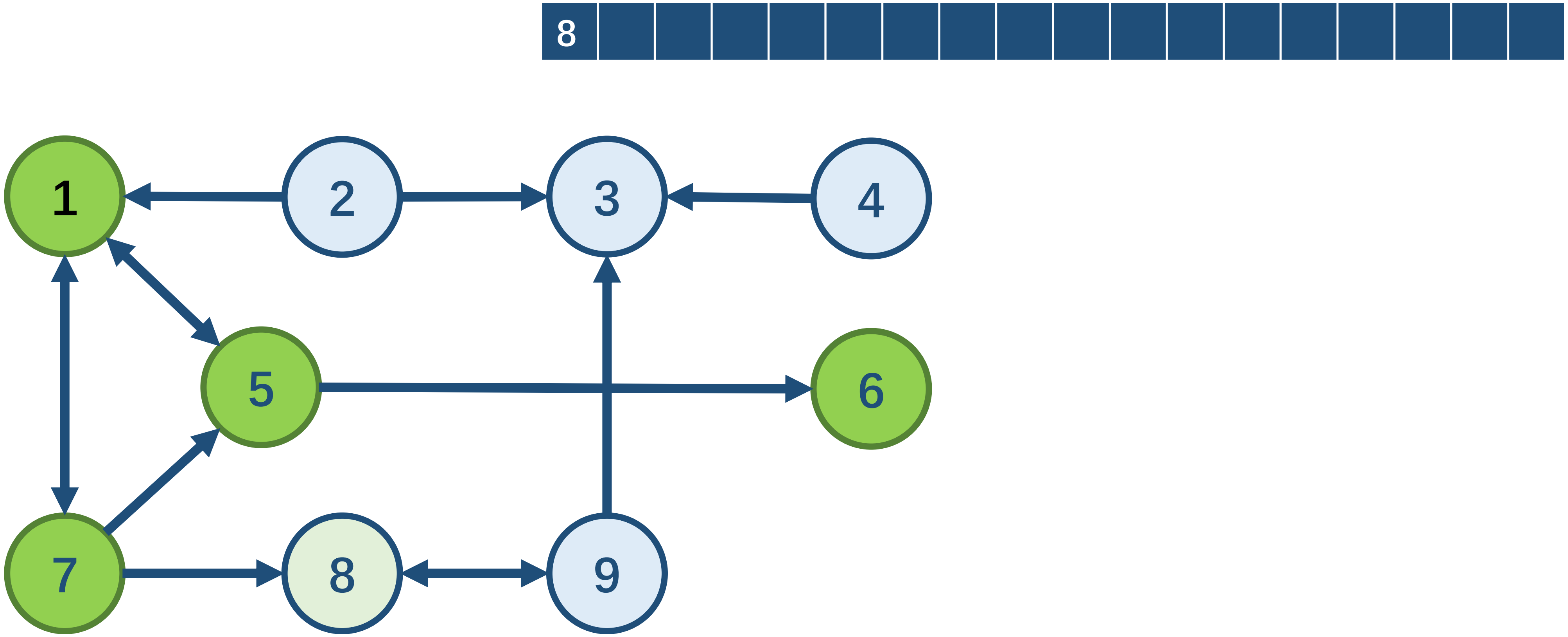
- Dequeue node 7 (and mark node 7 as seen)
- Add all its unseen neighbors to the queue (8)

6	8																



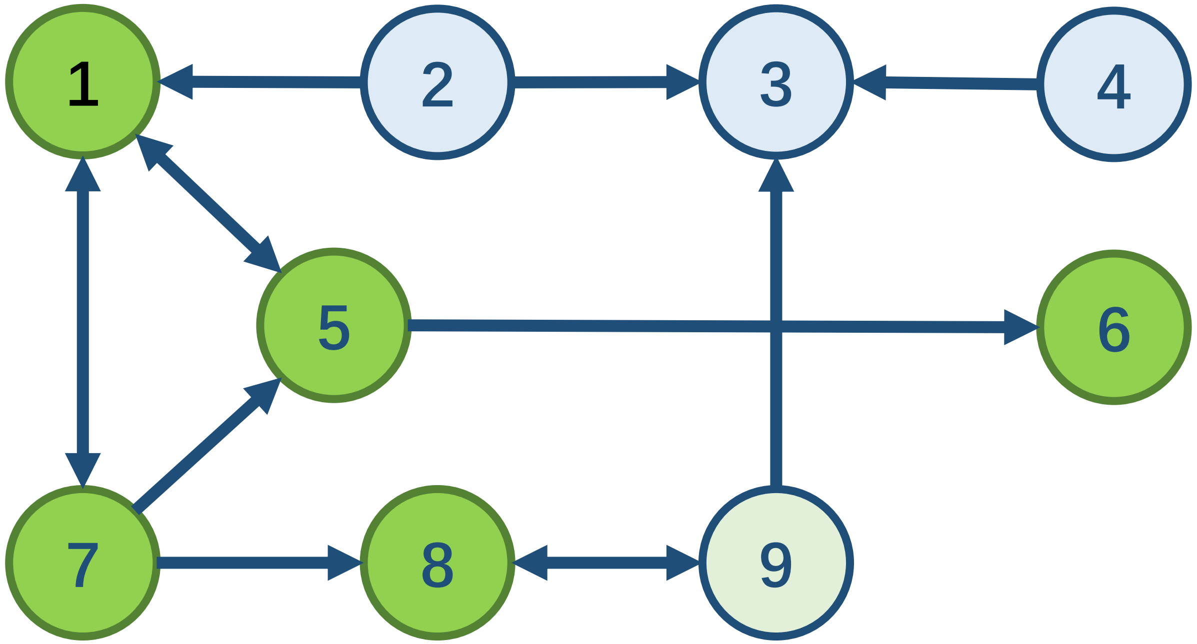
Breadth-First Search

- Dequeue node 7 (and mark node 7 as seen)
- Add all its unseen neighbors to the queue (None)



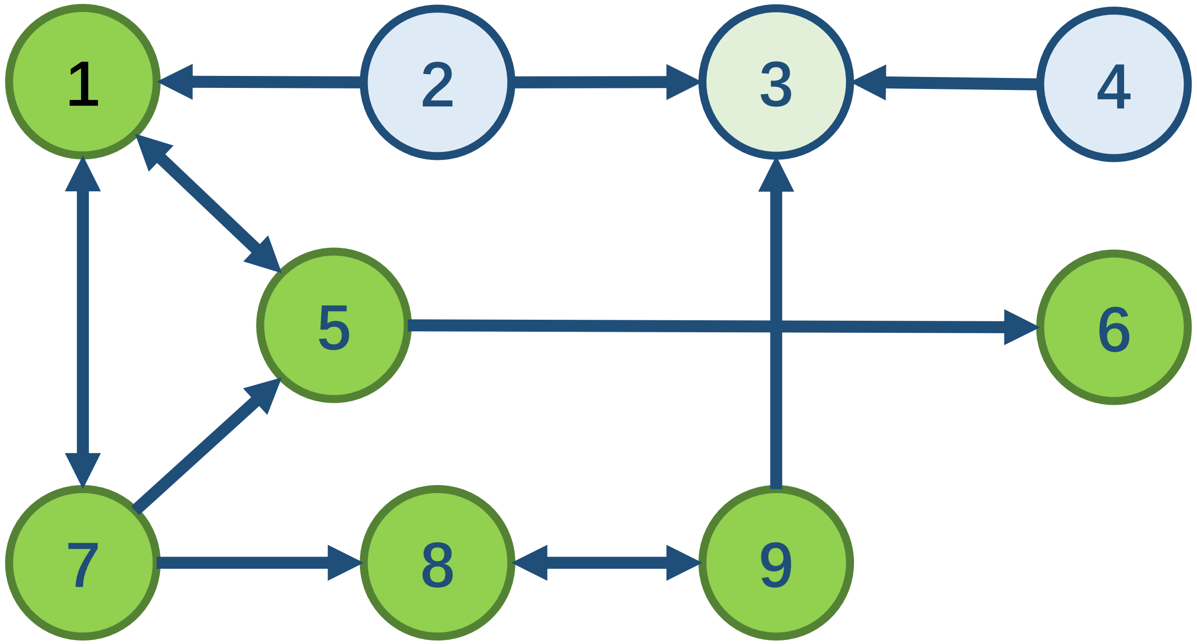
Breadth-First Search

- Dequeue node 8 (and mark node 8 as seen)
- Add all its unseen neighbors to the queue (9)



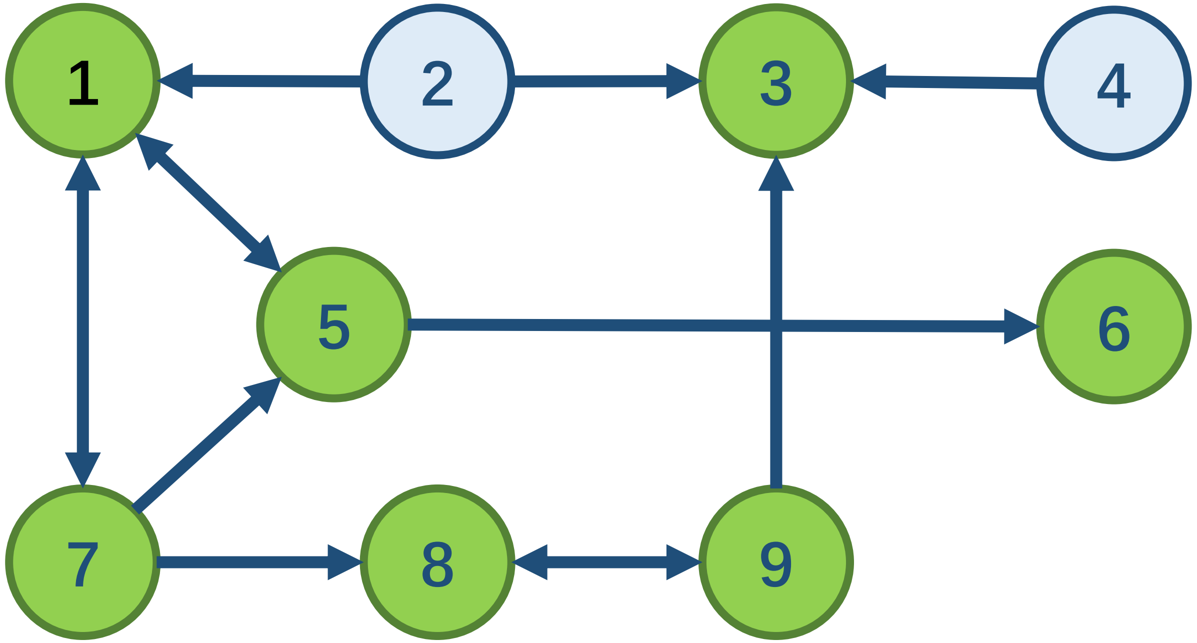
Breadth-First Search

- Dequeue node 9 (and mark node 9 as seen)
- Add all its unseen neighbors to the queue (3)



Breadth-First Search

- Dequeue node 3 (and mark node 3 as seen)
- Add all its unseen neighbors to the queue (None)



Implement BFS in C++

```
void bfs(int node, vector<vector<int>>& isConnected, vector<bool>& visit) {
        queue<int> q;
        q.push(node);
        visit[node] = true;

        while (!q.empty()) {
            node = q.front();
            q.pop();

            for (int i = 0; i < isConnected.size(); i++) {
                if (isConnected[node][i] && !visit[i]) {
                    q.push(i);
                    visit[i] = true;
                }
            }
        }
    }
```

City Conquer Game using BFS

```
int conquerAll(vector<vector<int>>& isConnected) {
    int n = isConnected.size();
    vector<bool> visit(n);
    int numberOfArmies = 0;
    for (int i = 0; i < n; i++) {
        if (!visit[i]) {
            numberOfArmies++;
            bfs(i, isConnected, visit);
        }
    }
    return numberOfArmies;
}
```

Find the Shorted Path using BFS

```
// (dist is initialized with a maximum value, such as INT_MAX)
void bfs(int node, vector<vector<int>>& isConnected, vector<int>& dist) {
    queue<int> q;
    q.push(node);
    dist[node] = 0; // Mark the distance of the source node as 0

    while (!q.empty()) {
        node = q.front();
        q.pop();

        for (int i = 0; i < isConnected.size(); i++) {
            if (isConnected[node][i] && dist[i] == INT_MAX) {
                q.push(i);
                dist[i] = dist[node] + 1; // Increase dist when moving to its neighbor
            }
        }
    }
}
```

Implement DFS in C++ using recursion

- Previously, we implemented **DFS** using recursion, can we use a data structure to **avoid recursion**?

```
void dfs(int node, vector<vector<bool>>& isConnected, vector<bool>& visit) {
    visit[node] = true;
    for (int i = 0; i < isConnected.size(); i++) {
        if (isConnected[node][i] && !visit[i]) {
            dfs(i, isConnected, visit);
        }
    }
}
```

isConnected

				1		1		
1		1						
		1						
1					1			
1				1			1	
								1
		1					1	

visit

Implement DFS using a stack

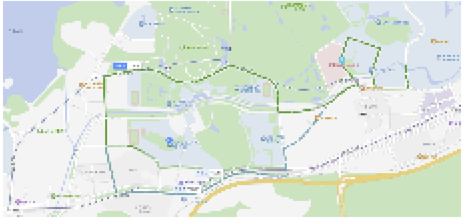
```
void dfs(int node, vector<vector<int>>& isConnected, vector<bool>& visit) {
        stack<int> s;
        s.push(node);
        visit[node] = true;

        while (!s.empty()) {
            node = s.top();
            s.pop();

            for (int i = 0; i < isConnected.size(); i++) {
                if (isConnected[node][i] && !visit[i]) {
                    s.push(i);
                    visit[i] = true;
                }
            }
        }
    }
```

Least-Cost Paths

- BFS allows us to find the **shortest path** (all edges are equally weighted)
- Dijkstra's Algorithm for finding the **least-cost path**
- Real life examples:
  - Road toll
  - Miles on a road
  - Time spent on a road



Shorted Path to Least-Cost Path

```
void bfs(int node, vector<vector<int>>& isConnected, vector<int>& dist) {
    queue<int> q;
    q.push(node);
    dist[node] = 0;

    while (!q.empty()) {
        node = q.front();
        q.pop();

        for (int i = 0; i < isConnected.size(); i++) {
            if (isConnected[node][i] && dist[i] == INT_MAX) {
                q.push(i);
                dist[i] = dist[node] + 1;
            }
        }
    }
}
```

How could we modify this code to **dequeue the least-cost nodes** instead of the closest nodes?

Use a priority queue (min-heap) instead of a queue!

Edsger Dijkstra

(1930-2002)

- Famous Dutch computer scientist and professor at UT Austin
  - Turing Award winner (1972)
- Noteworthy algorithms and software:
  - The multiprogramming system (OS)
    - layers of abstraction
  - Compiler for a language that can do recursion
  - **Dijkstra's algorithm**
  - Dining Philosophers Problem: resource contention, deadlock

Dijkstra Pseudocode

dijkstra(v1, v2):

consider every vertex to have **a cost of infinity**, except **v1 which has a cost of 0**. create a priority queue of vertexes **pq**, ordered by cost, storing only v1 .

while the **pq** is not empty:

    dequeue a vertex v from the **pq**, and mark it as visited.

    for each unvisited neighbor, n, of v, we can reach n with a total cost of (v's cost + the weight of the edge from v to n).

        if this cost is cheaper than n's current cost, we should enqueue the neighbor n to the **pq** with this new cost, and remember v was its previous vertex.

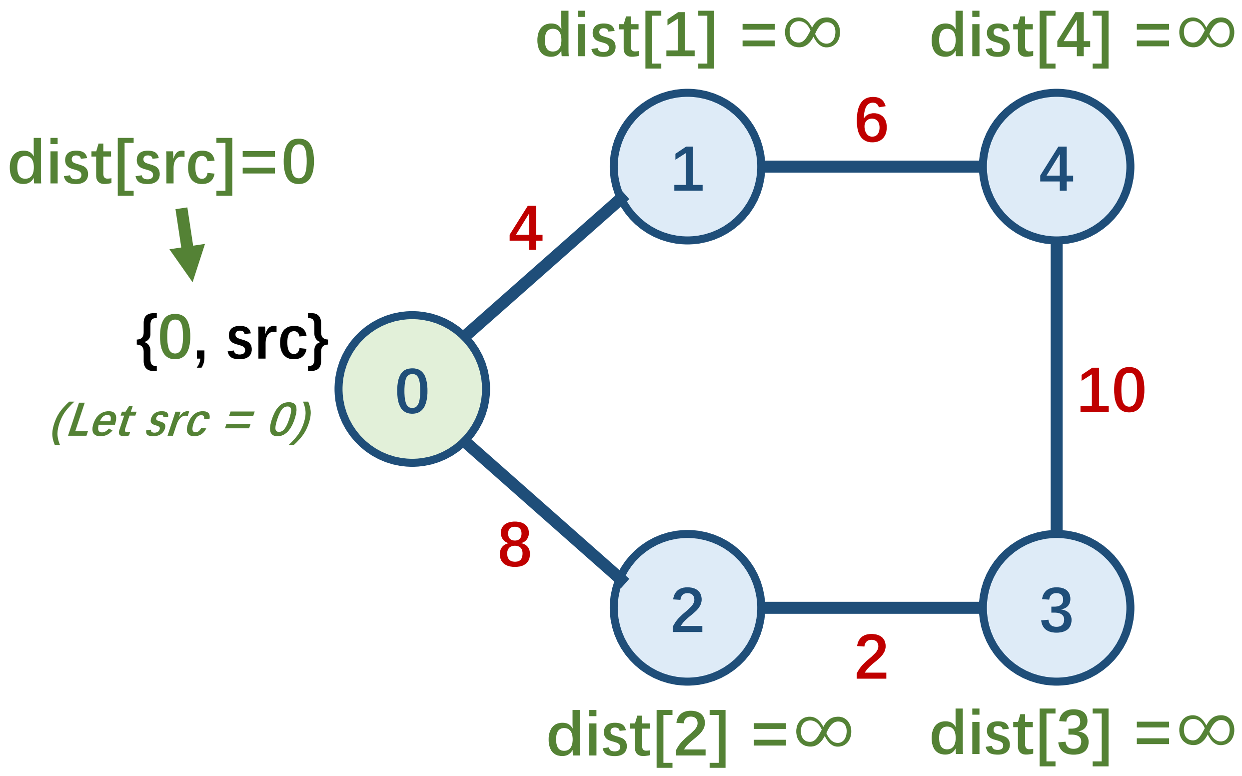
when we are done, we can reconstruct the path from v2 back to v1 by following the path of previous vertices.

Dijkstra's Algorithm C++ Implementation

```
vector<int> dijkstra(vector<vector<vector<int>>>& adj, int src){
    priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
    vector<int> dist(adj.size(), INT_MAX); // Initialize all distances as infinite
    pq.push({0, src}); // Insert source itself in priority queue
    dist[src] = 0; // The source has 0 distance
    while (!pq.empty()){
        int u = pq.top()[1];
        pq.pop();

        for (auto x : adj[u]){
            int v = x[0]; // First element of x stores node index
            int weight = x[1]; // Second element of x stores the weight
            if (dist[v] > dist[u] + weight) // If there is shorter path to v through u
            {
                dist[v] = dist[u] + weight; // Updating distance of v
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

Dijkstra's Algorithm Example



dist

0	∞	∞	∞	∞

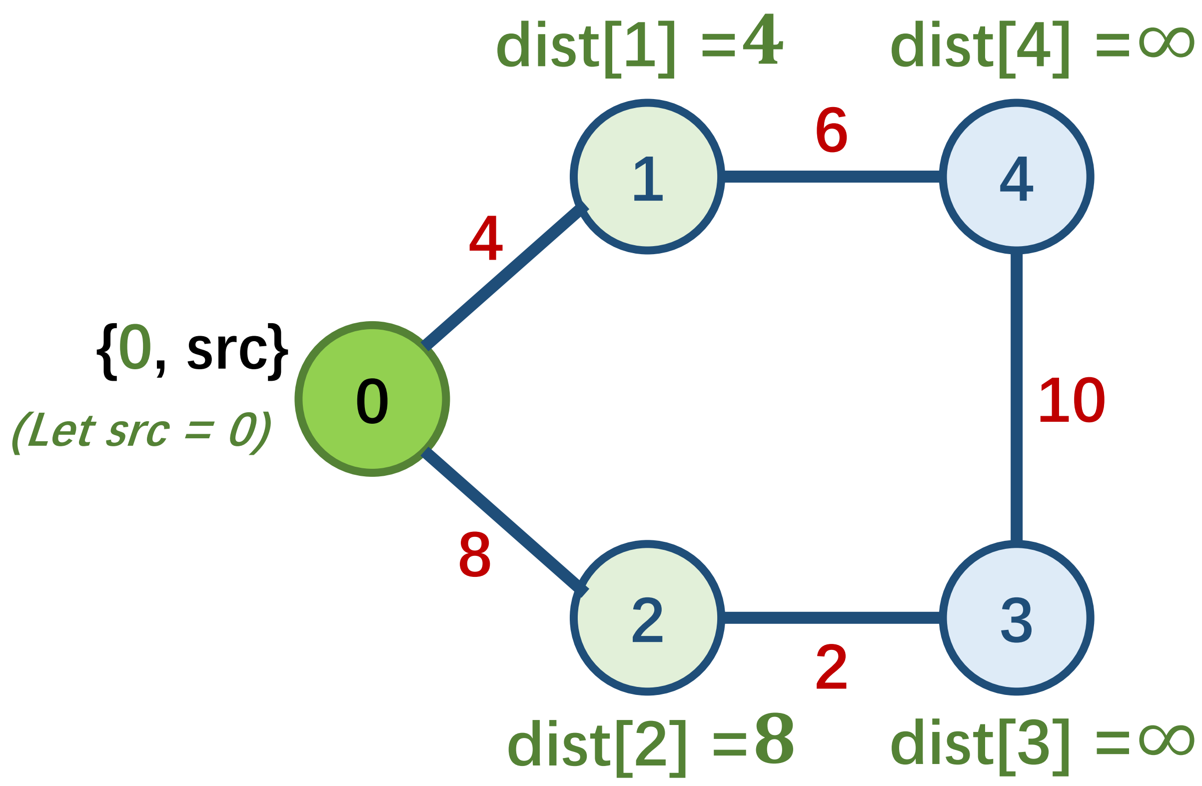
pq

{0, 0}				

push

{0, src}

Dijkstra's Algorithm Example



dist				
0	4	8	∞	∞

pq

pop

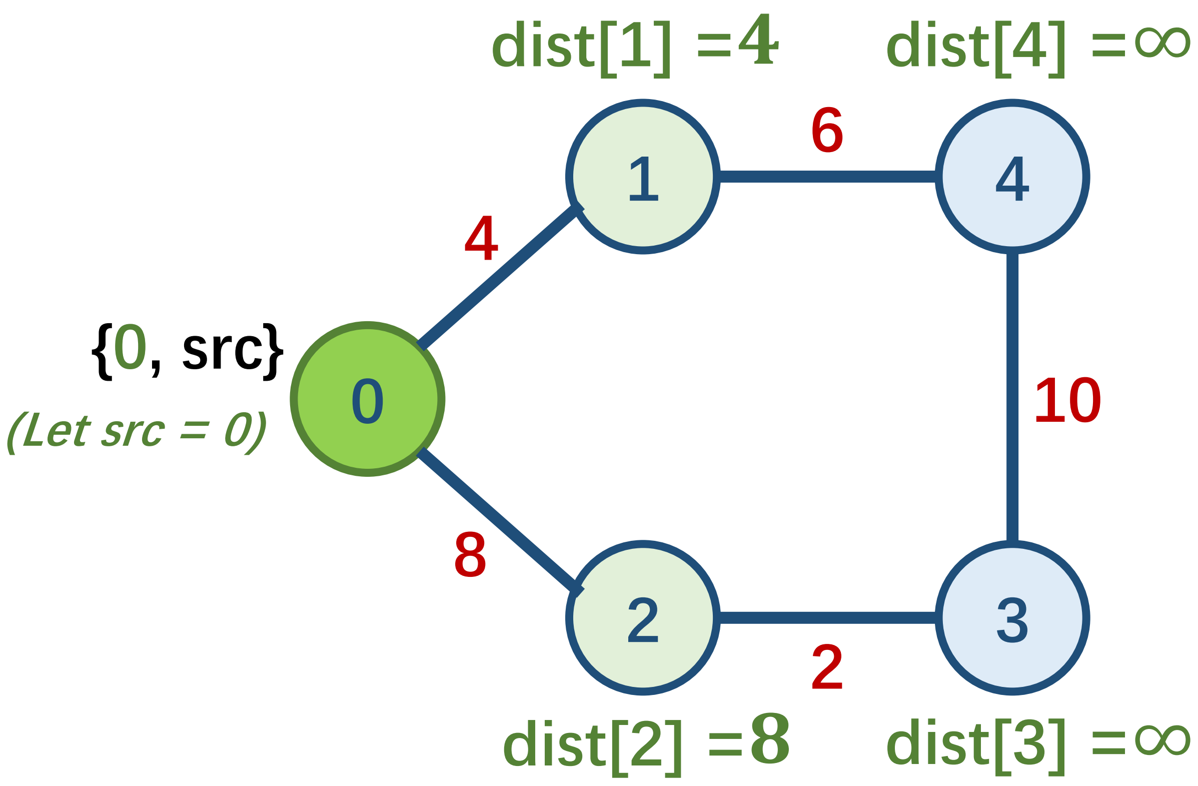
{0, src}

u = src = 0

v = 1, weight = 4
dist[u]+weight = 4

v = 2, weight = 8
dist[u]+weight = 8

Dijkstra's Algorithm Example



dist				
0	4	8	∞	∞

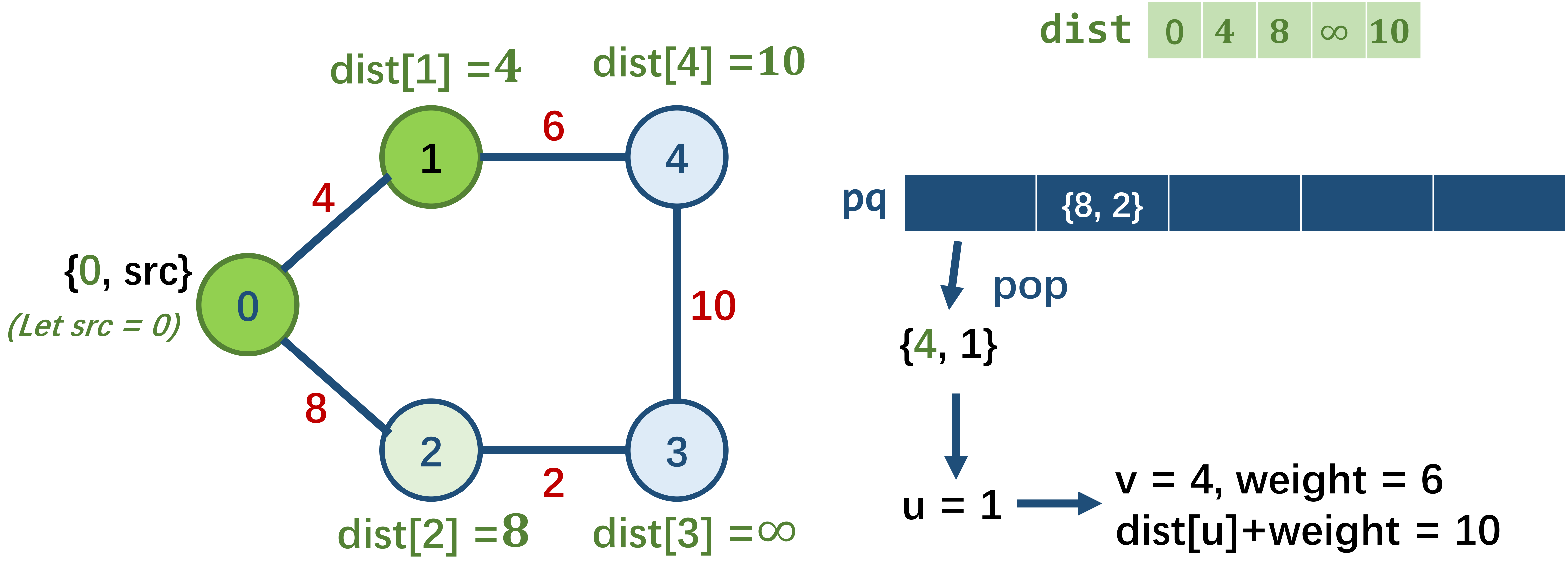
pq					
	{4, 1}	{8, 2}			

push

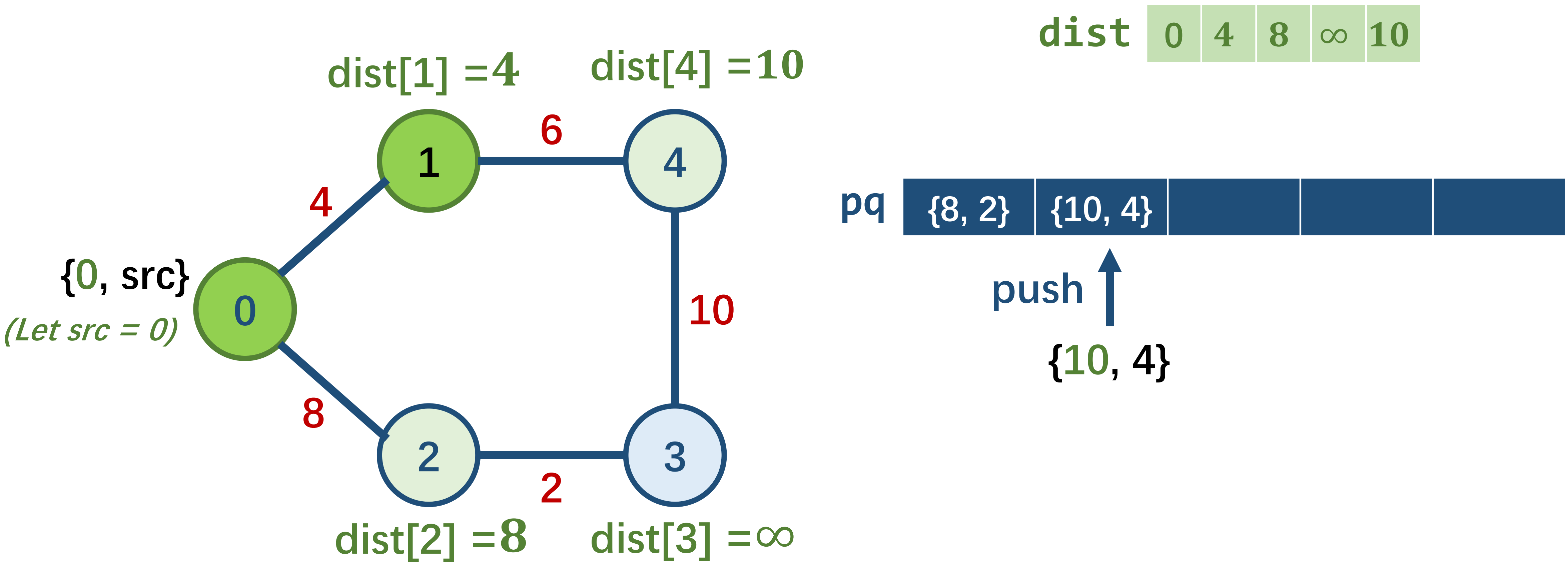
{4, 1} {8, 2}

Note that smaller distance has higher priority

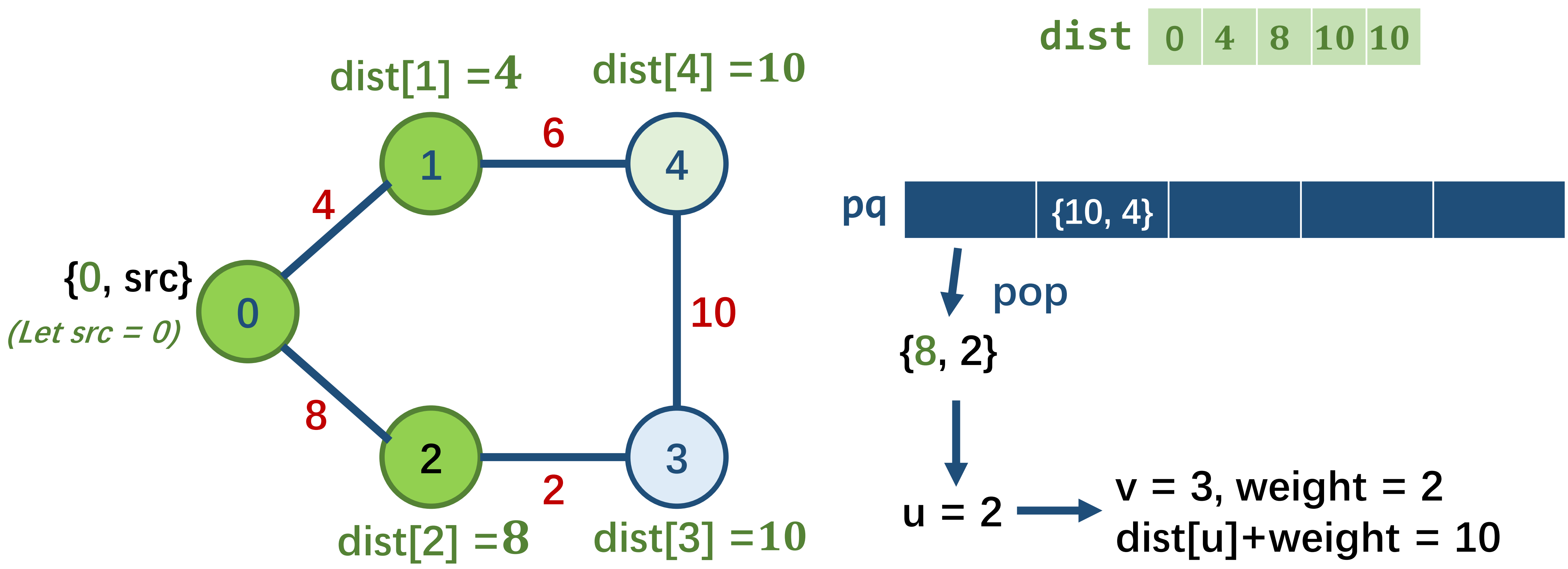
Dijkstra's Algorithm Example



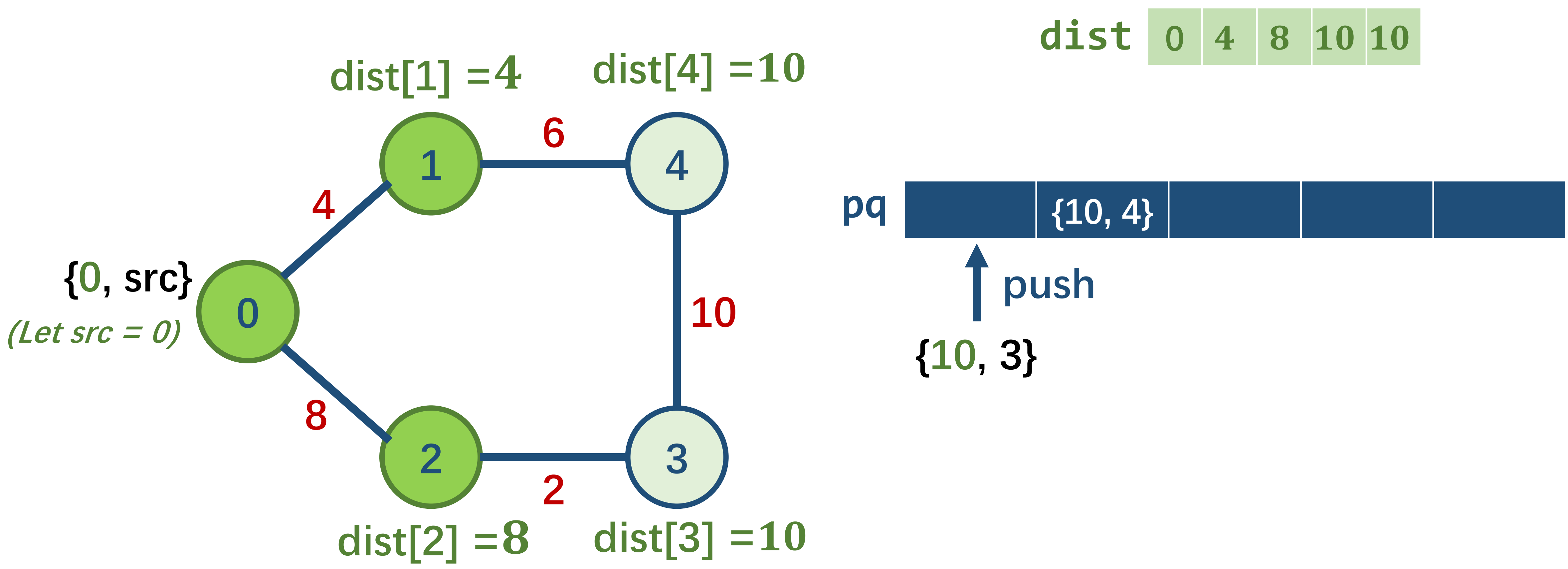
Dijkstra's Algorithm Example



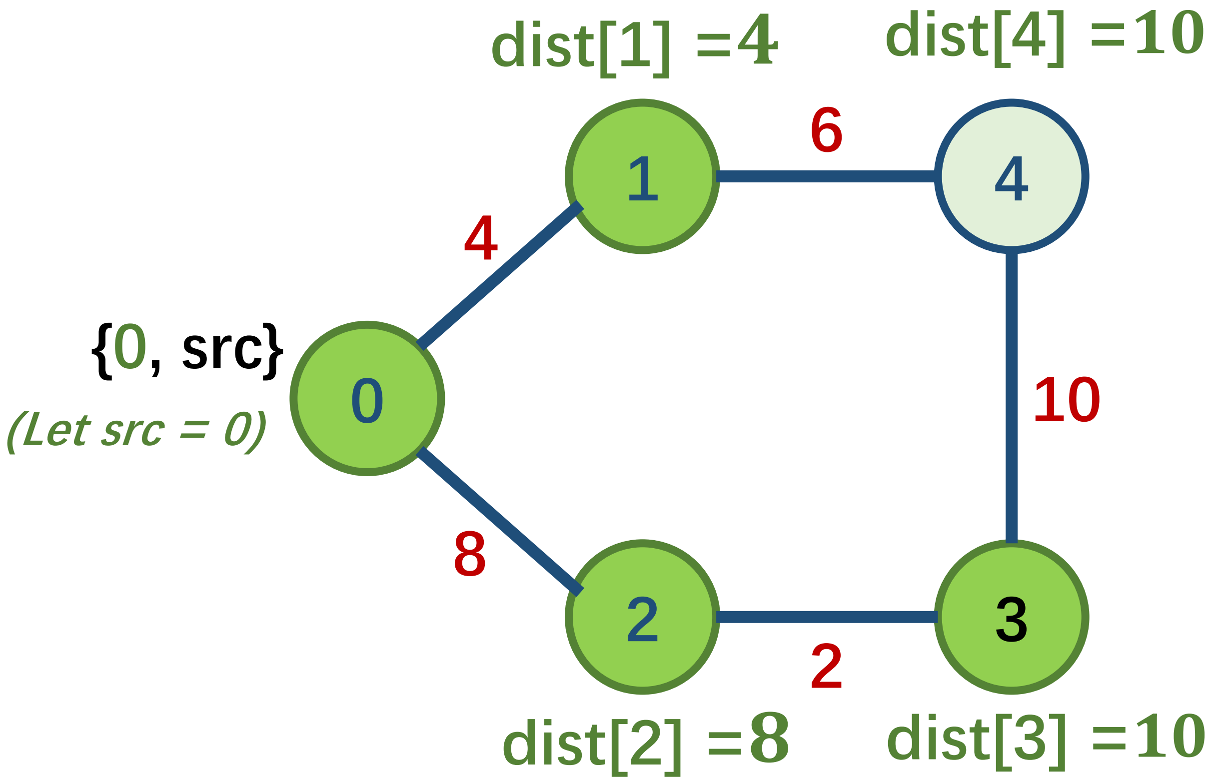
Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



dist	0	4	8	10	10

pq		{10, 4}			

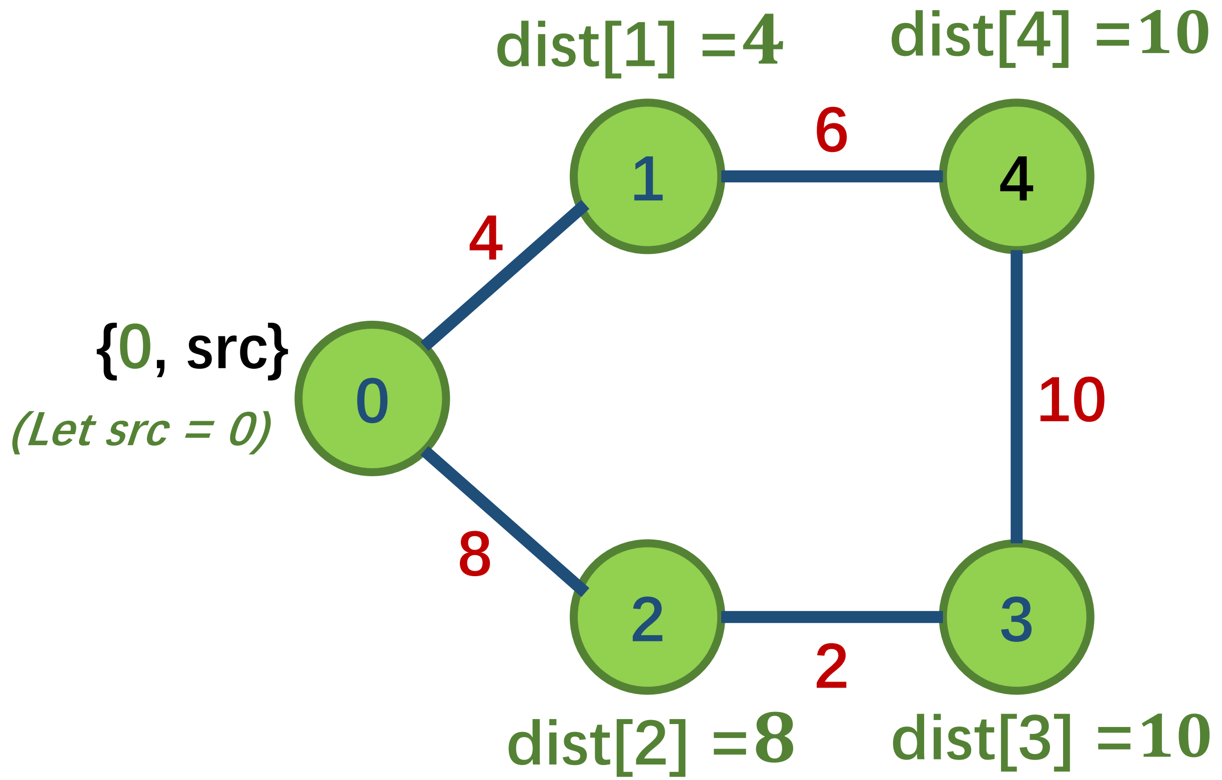
pop

{10, 3}

u = 3 → v = 4, weight = 10
dist[u]+weight = 20

No changes since dist[u]+weight > dist [v]

Dijkstra's Algorithm Example



dist				
0	4	8	10	10

pq

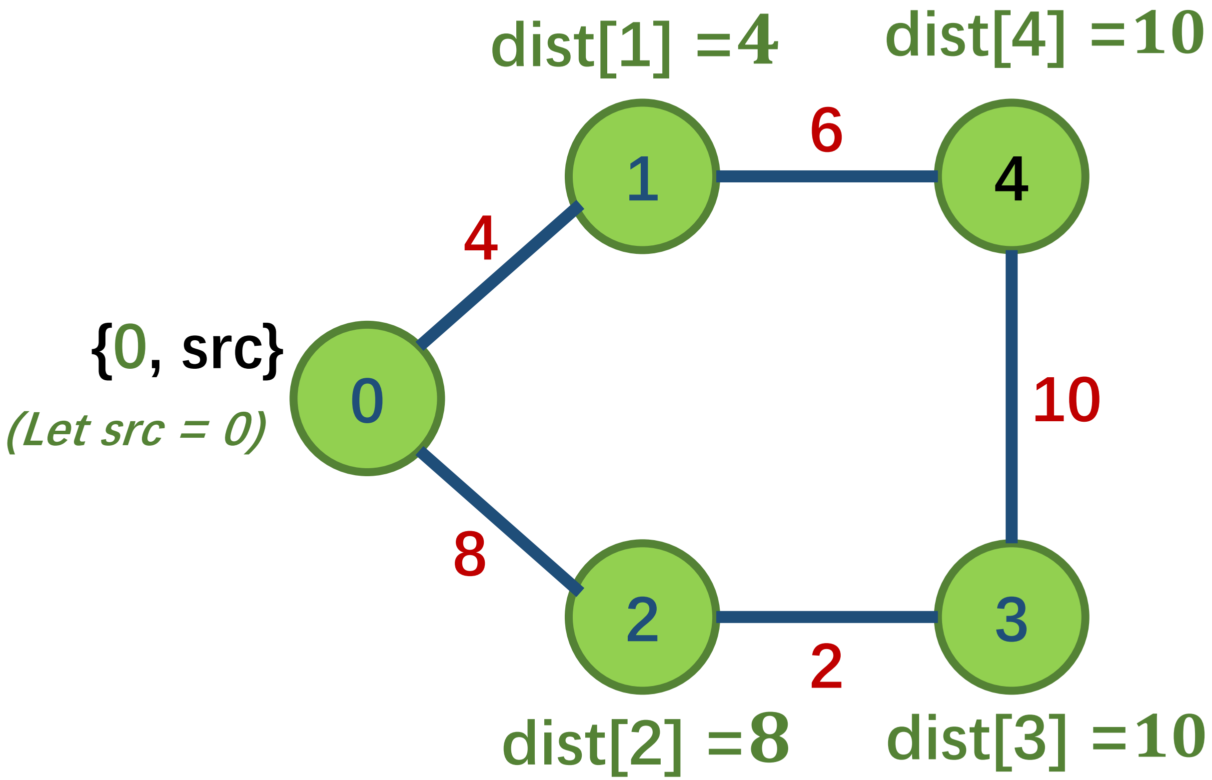
pop

{10, 4}

u = 4 → v = 3, weight = 10
dist[u]+weight = 20

No changes since dist[u]+weight > dist [v]

Dijkstra's Algorithm Example



dist	0	4	8	10	10

Finally, dist *holds the shortest distance from the source to all nodes*

pq					

Exercise 10.1

- Complete LeetCode 1791

1791. Find Center of Star Graph

Easy | Topics | Companies | Hint

There is an undirected **star** graph consisting of `n` nodes labeled from `1` to `n`. A star graph is a graph where there is one **center** node and **exactly** `n - 1` edges that connect the center node with every other node.

You are given a 2D integer array `edges` where each `edges[i] = [`u_i`, `v_i`]` indicates that there is an edge between the nodes u_i and v_i. Return the center of the given star graph.

Exercise 10.2

- Complete [LeetCode 1971]

1971. Find if Path Exists in Graph

Easy | Topics | Companies

There is a **bi-directional** graph with `n` vertices, where each vertex is labeled from `0` to `n - 1` (**inclusive**). The edges in the graph are represented as a 2D integer array `edges`, where each `edges[i] = [u_i, v_i]` denotes a bi-directional edge between vertex `u_i` and vertex `v_i`. Every vertex pair is connected by **at most one** edge, and no vertex has an edge to itself.

You want to determine if there is a **valid path** that exists from vertex `source` to vertex `destination`.

Given `edges` and the integers `n`, `source`, and `destination`, return `true` *if there is a **valid path** from* `source` *to* `destination`*, or* `false` *otherwise.*

Hint: You can build an adjacency list from the given edge list

```
vector<vector<int>>adjList(n);
for(vector<int> temp : edges){
    int u = temp[0];
    int v = temp[1];
    adjList[u].push_back(v);
    adjList[v].push_back(u);
}
```

Exercise 10.3

- Complete LeetCode 547

547. Number of Provinces

Medium | Topics | Companies

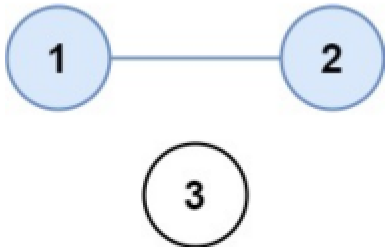
There are `n` cities. Some of them are connected, while some are not. If city `a` is connected directly with city `b`, and city `b` is connected directly with city `c`, then city `a` is connected indirectly with city `c`.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an `n x n` matrix `isConnected` where `isConnected[i][j] = 1` if the i^{th} city and the j^{th} city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return *the total number of* ***provinces***.

Example 1:




```
Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]
Output: 2
```

Exercise 10.4

- Complete [LeetCode 841]

841. Keys and Rooms

Medium | Topics | Companies

There are `n` rooms labeled from `0` to `n - 1` and all the rooms are locked except for room `0`. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of **distinct keys** in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array `rooms` where `rooms[i]` is the set of keys that you can obtain if you visited room `i`, return `true` *if you can visit* ***all*** *the rooms, or* `false` *otherwise*.

Example 2:

```
Input: rooms = [[1,3],[3,0,1],[2],[0]]
Output: false
Explanation: We can not enter room number 2 since the only key that unlocks it is in that room.
```

Exercise 10.5

- You May Try LeetCode 3286 (A Simple Game AI)

3286. Find a Safe Walk Through a Grid

Medium | Topics | Companies | Hint

You are given an `m x n` binary matrix `grid` and an integer `health`.

You start on the upper-left corner `(0, 0)` and would like to get to the lower-right corner `(m - 1, n - 1)`.

You can move up, down, left, or right from one cell to another adjacent cell as long as your health *remains* **positive**.

Cells `(i, j)` with `grid[i][j] = 1` are considered **unsafe** and reduce your health by 1.

Return `true` if you can reach the final cell with a health value of 1 or more, and `false` otherwise.

Example 1:

> **Input:** `grid = [[0,1,0,0,0],[0,1,0,1,0],[0,0,0,1,0]], health = 1`
>
> **Output:** `true`
>
> **Explanation:**
>
> The final cell can be reached safely by walking along the gray cells below.

0	1	0	0	0
0	1	0	1	0
0	0	0	1	0

Hint: When you search through the path, need to check more than just the visiting status of a node