

# IT5001 Software Development Fundamentals

## 16. Miscellaneous

# Agenda

- Decorators for Memoization
- Graph Problems
  - Route Problems
  - Shortest Distance

# Decorators

- Decorator is a closure
  - Additionally, outer function accepts a function as input argument
- Modify input function's behaviour with an inner function without explicitly changing input function's code
- Example

```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper
```

```
def f():  
    pass
```

```
f = deco(f)
```

```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper
```

```
@deco  
def f():  
    pass
```

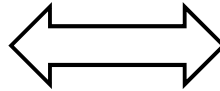
# Decorators - Example

```
def my_decorator(func):  
    def wrapper():  
        print('Hello')  
        func()  
        print('Welcome')  
    return wrapper
```

```
def func():  
    print('IT5001')
```

```
decorated_func = my_decorator(func)
```

```
decorated_func()
```



```
def my_decorator(func):  
    def wrapper():  
        print('Hello')  
        func()  
        print('Welcome')  
    return wrapper
```

```
@my_decorator  
def func():  
    print('IT5001')
```

```
func()
```

# Decorators: Example

- Decorating functions that has arguments

```
def my_decorator(func):  
    def wrapper(*args, **kwargs):  
        #statements  
        func(*args, **kwargs)  
    return wrapper
```

```
@my_decorator  
def f():  
    pass
```

# Decorators: Example

- Decorating functions that has arguments

```
def my_decorator(func):  
    def wrapper(*args, **kwargs):  
        print('executing decorated function')  
        return func(*args, **kwargs)  
    return wrapper
```

```
@my_decorator  
def g(x, y = None):  
    return x**2+y**2
```

# Decorators: Applications

- Profiling
  - For timing functions
- Logging
  - For debugging
- Caching
  - For Memoization

# Decorators for Caching

```
def cache(func) :  
    memo = {}  
    def wrap (*args) :  
        if args not in memo:  
            memo[args] = func(*args)  
        return memo[args]  
    return wrap
```



# Fibonacci using decorators

```
def fibm(n):  
    if n in fibans.keys():  
        return fibans[n]  
  
    if (n == 0):  
        ans = 0  
    elif (n == 1):  
        ans = 1  
    else:  
        ans = fibm(n-1)+fibm(n-2)  
    return ans  
print(fibm(10))
```

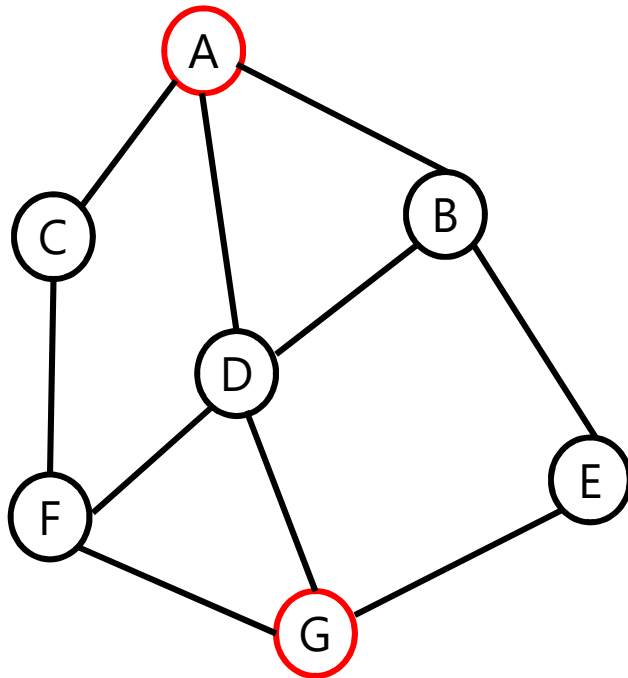


```
@cache  
def fib(n):  
    if n == 0:  
        return 0  
    if n <= 2:  
        return 1  
    return fib(n-1) + fib(n-2)  
  
print(fib(50))
```

# Graphs

Path Between Vertices

# Breadth-First Search

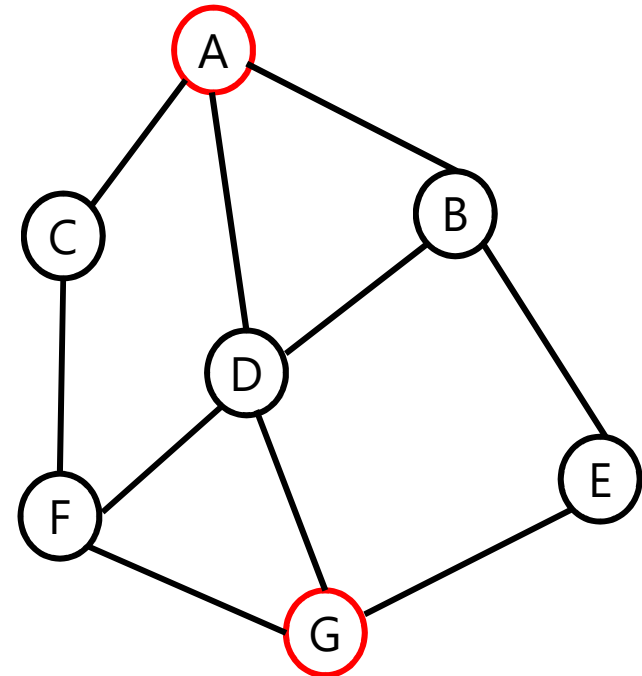


**Objective:**

Checking if a path exists from vertex A to vertex G

# Graph Representation

- Consists of two components
  - Vertices
    - A,B,C,D, E, ....
  - Edges
    - (A,B), (A,C), (B,D)....
- How to represent it?
  - Edge List
    - Contains list of all edges
  - Adjacency List/Dictionary
    - List of vertices that are adjacent to a given vertex
    - Can use dictionary
      - Provides mapping between each vertex and its neighbors
- Assume we are given list of edges, i.e., edge list



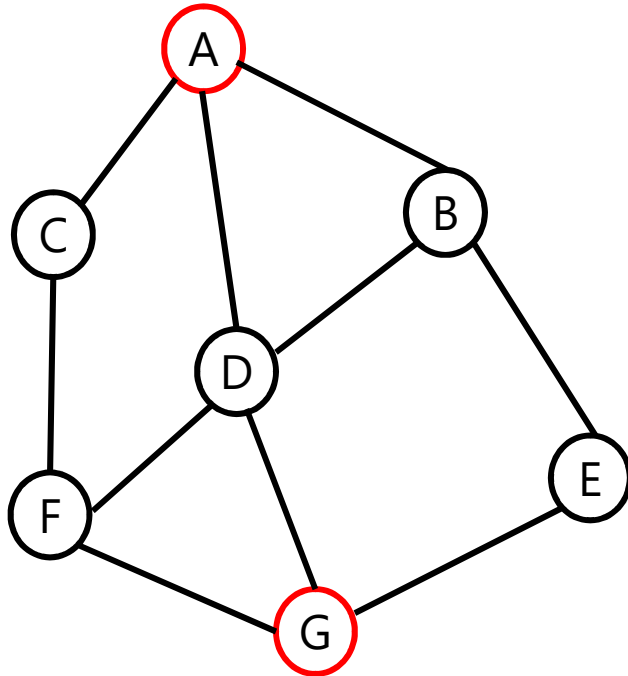
## Edge List to Adjacency List

```
def edgeList_to_adjList(edgeList):  
    adjacencyList = {}  
    for a, b in edgeList:  
        if a not in adjacencyList:  
            adjacencyList[a] = []  
        if b not in adjacencyList:  
            adjacencyList[b] = []  
        adjacencyList[a].append(b)  
        adjacencyList[b].append(a)  
    return adjacencyList
```

# Breadth-First Search

```
def can_travel_bfs(edgeList, source, destination):  
    adjacencyList = edgeList_to_adjList(edgeList)  
    visited = set()  
    frontier = [source]  
    while frontier:  
        current = frontier.pop(0)  
        if current == destination:  
            return True  
        if current not in adjacencyList or current in visited:  
            continue  
        visited.add(current)  
        frontier.extend(adjacencyList[current])  
    return False  
  
print(can_travel_bfs(edge_list, 'A', 'C'))
```

# Breadth-First Search

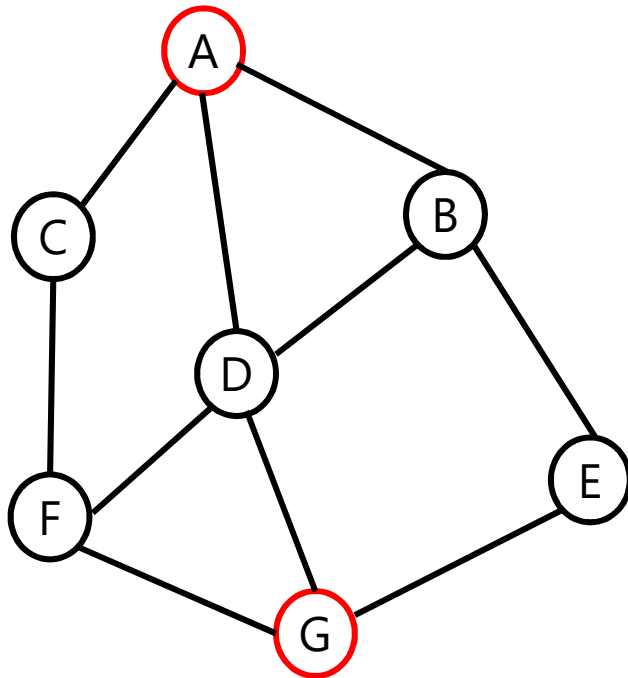


current  $\neq$  destination  
current *not in* visited

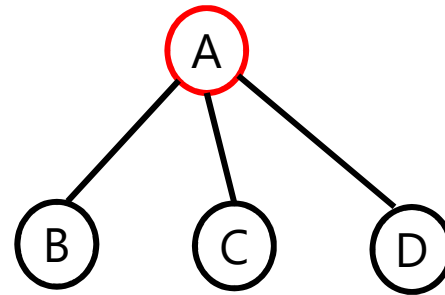
visited = {}  
frontier = ['A']

```
print(can_travel_bfs(edge_list, 'A', 'C'))
```

# Breadth-First Search



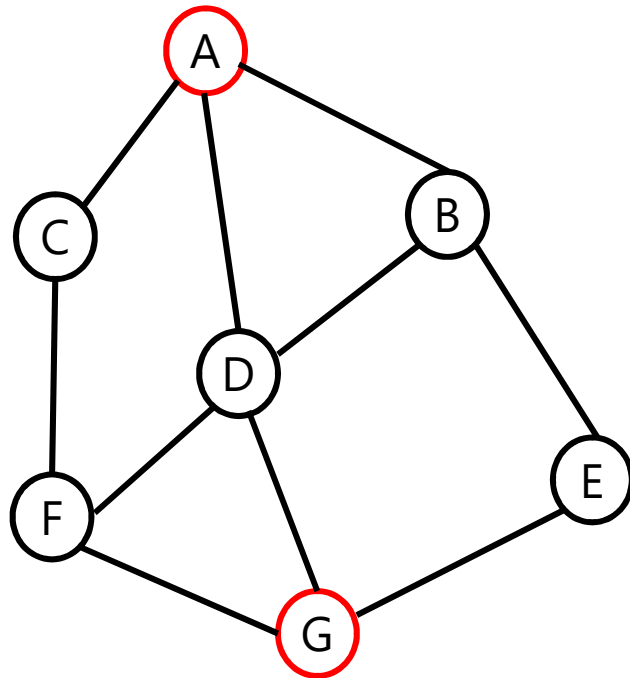
visited = {'A'}  
frontier = ['B', 'C', 'D']



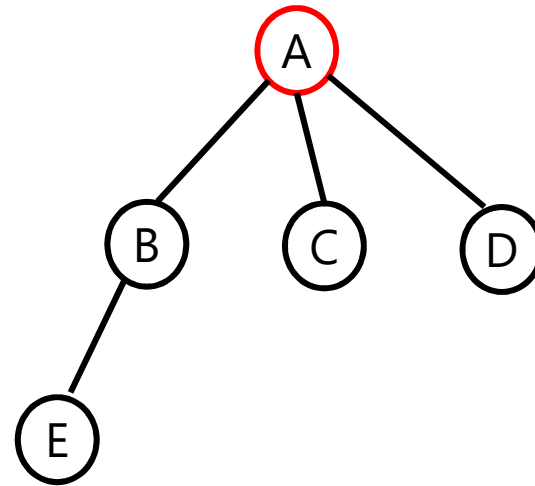
current  $\neq$  destination  
current *not in* visited



# Breadth-First Search

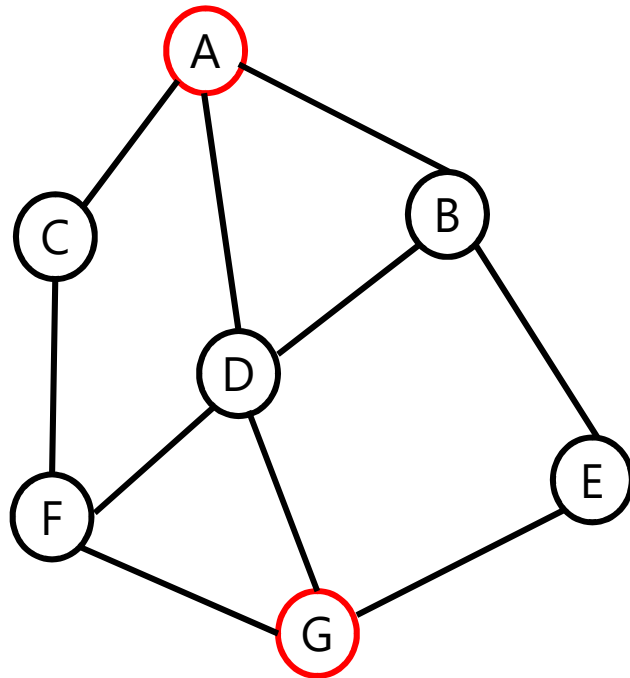


visited = {'A', 'B'}  
frontier = ['C', 'D', 'E']

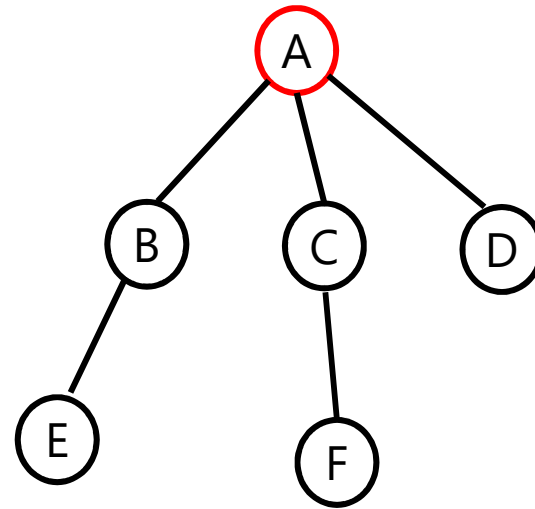


current  $\neq$  destination  
current *not in* visited

# Breadth-First Search

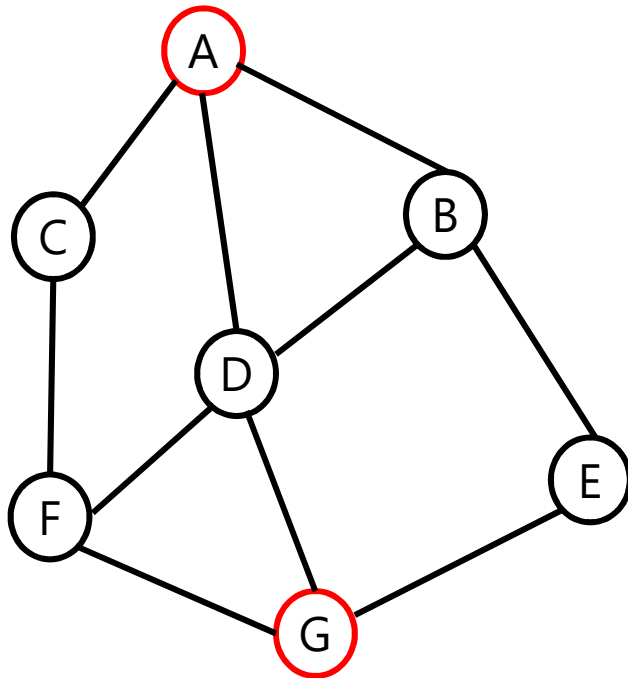


visited = {'A', 'B', 'C'}  
frontier = ['D', 'E', 'F']

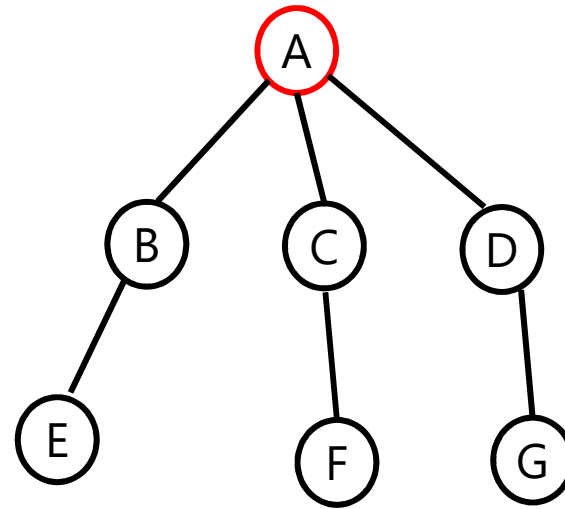


current  $\neq$  destination  
current *not in* visited

# Breadth-First Search



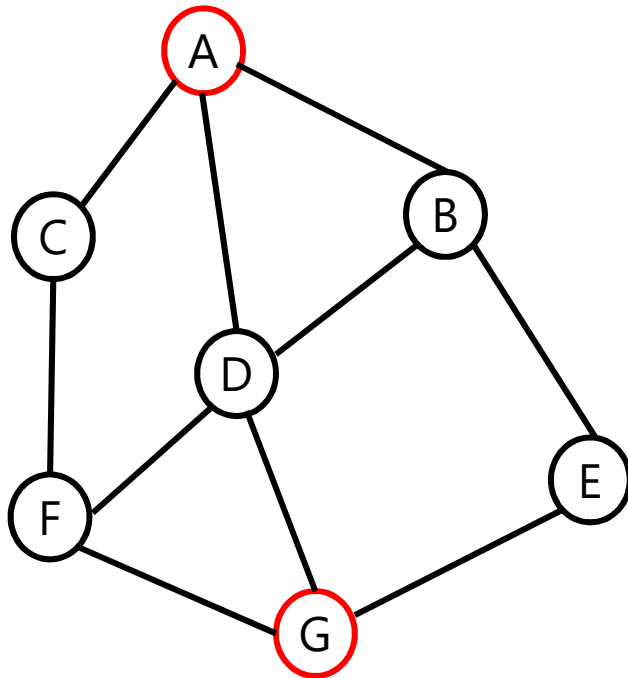
visited = {'A', 'B', 'C', 'D'}  
frontier = ['E', 'F', 'G']



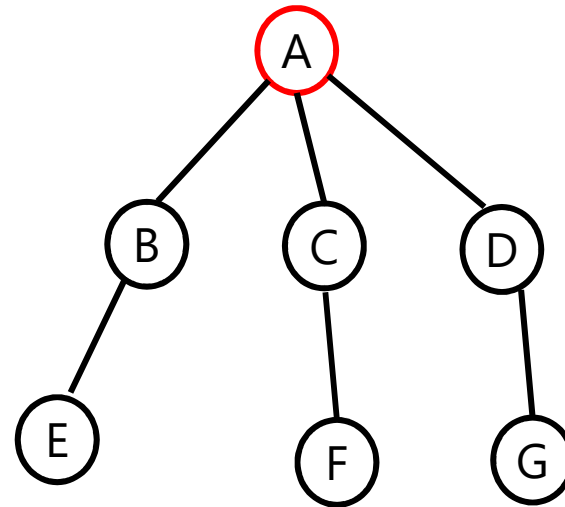
current *not in* visited

current  $\neq$  destination

# Breadth-First Search

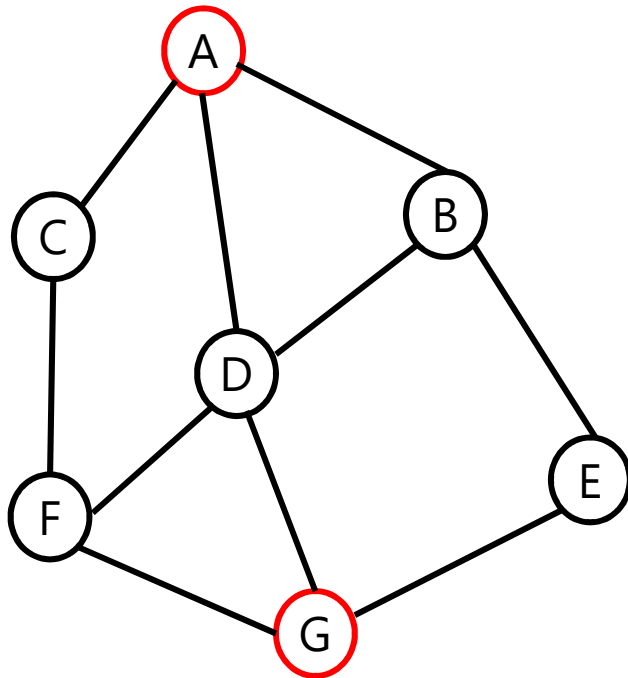


visited = {'A', 'B', 'C', 'D', 'E'}  
frontier = [ 'F', 'G' ]

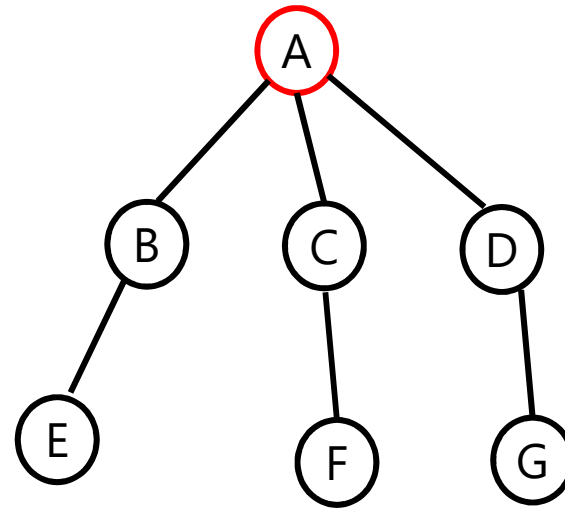


current  $\neq$  destination  
current *not in* visited

# Breadth-First Search

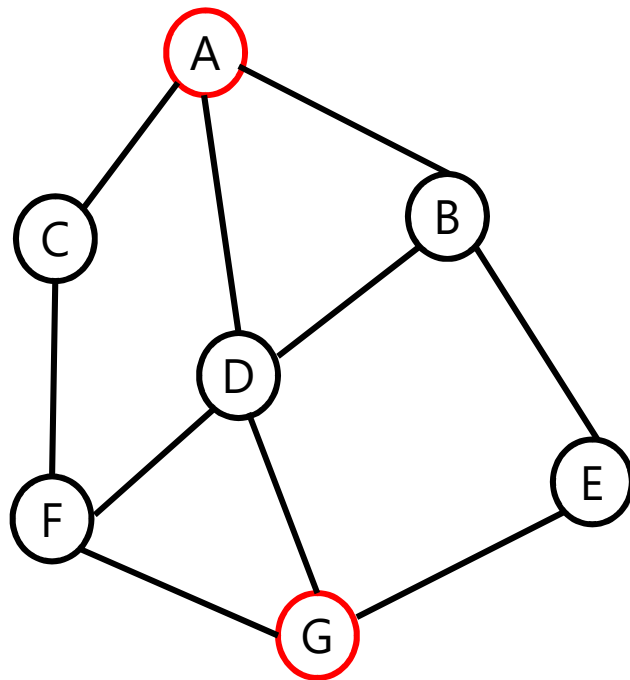


visited = {'A', 'B', 'C', 'D', 'E'}  
frontier = ['G']

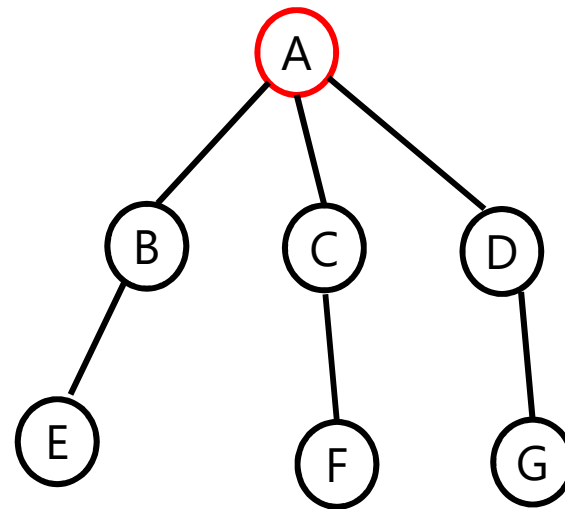


current  $\neq$  destination  
current *not in* visited

# Breadth-First Search



visited = {'A', 'B', 'C', 'D', 'E'}  
frontier = [ ]



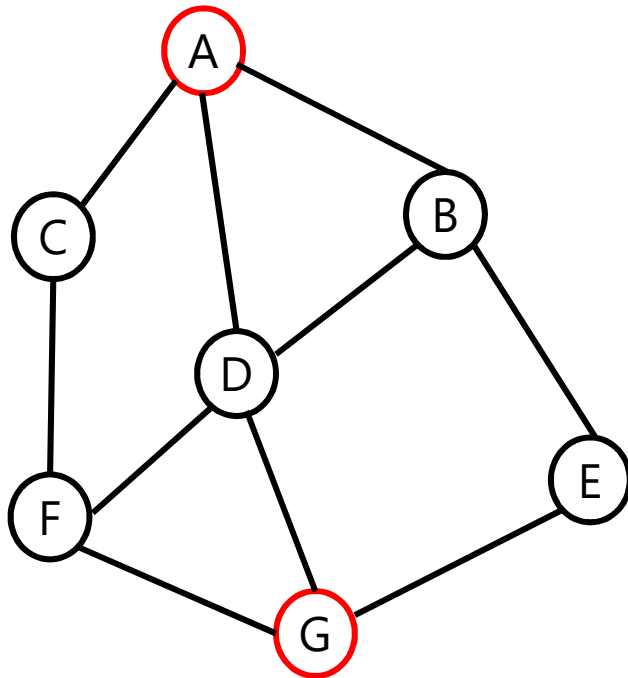
current == destination

Returns *True*

# Depth-First Search

```
def can_travel_dfs(edgeList, source, destination):  
    adjacencyList = edgeList_to_adjList(edgeList)  
    visited = set()  
    frontier = [source]  
    while frontier:  
        current = frontier.pop()  
        if current == destination:  
            return True  
        if current not in adjacencyList or current in visited:  
            continue  
        visited.add(current)  
        frontier.extend(adjacencyList[current])  
    return False  
  
print(can_travel_dfs(edge_list, 'A', 'C'))
```

# Depth-First Search



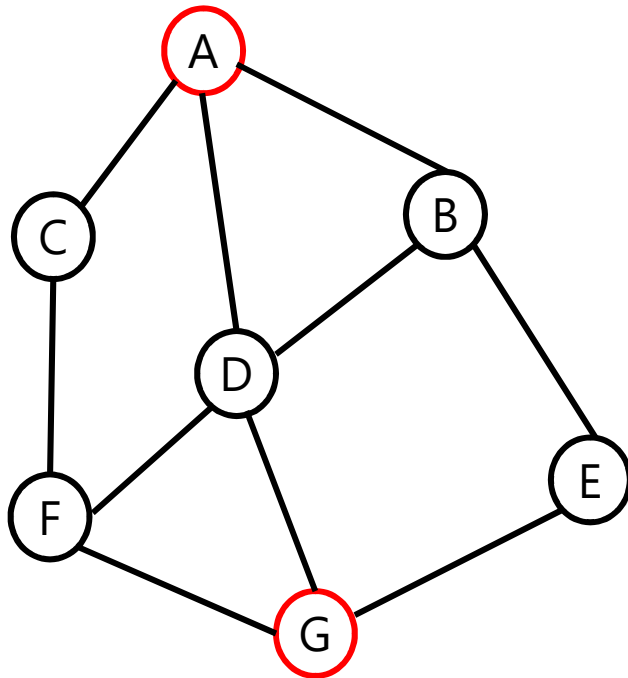
visited = {}  
frontier = ['A']

A

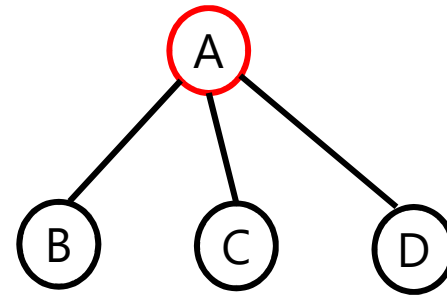
current  $\neq$  destination  
current *not in* visited



# Depth-First Search

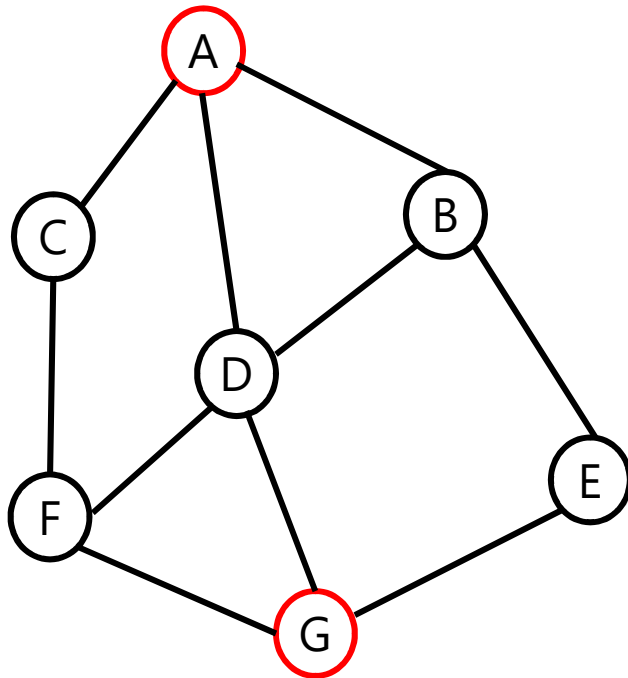


visited = {'A'}  
frontier = ['B', 'C', 'D']

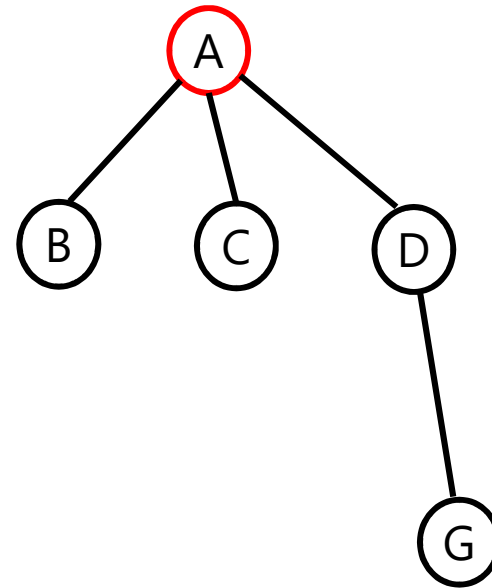


current  $\neq$  destination  
current *not in* visited

# Depth-First Search

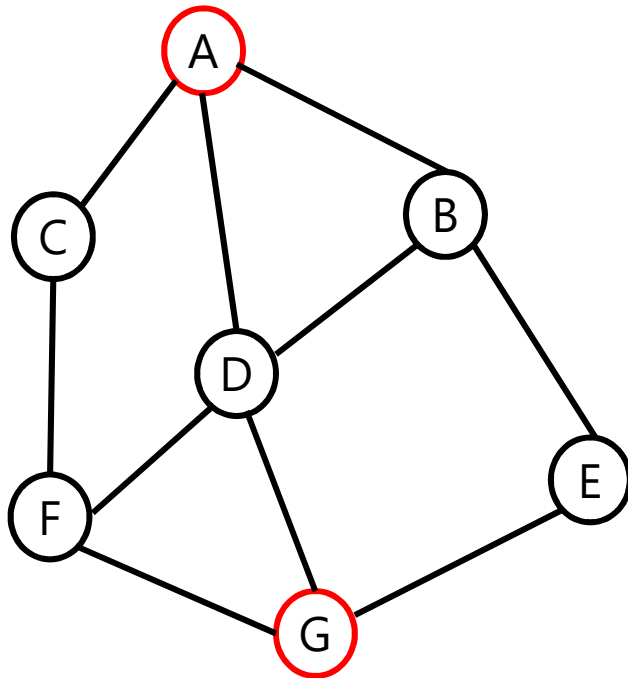


visited = {'A','D'}  
frontier = ['B', 'C', 'G']

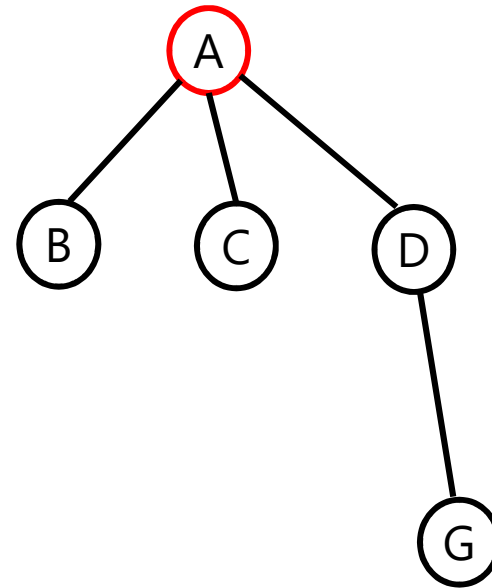


current  $\neq$  destination  
current *not in* visited

# Depth-First Search



visited = {'A', 'D'}  
frontier = ['B', 'C', 'G']

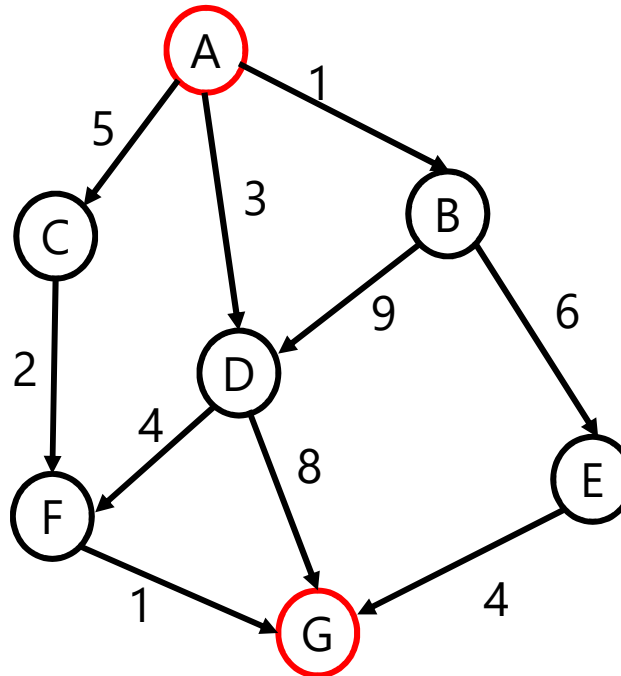


current == destination

Returns *True*

# Weighted Graphs

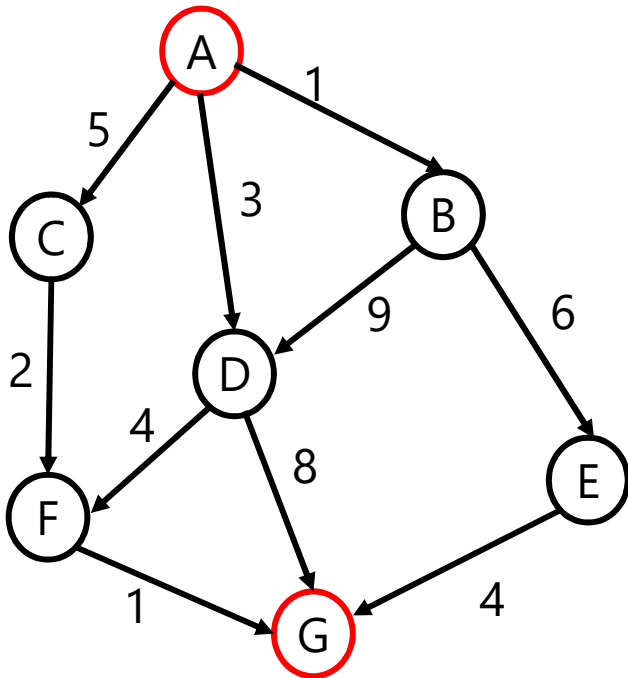
- Objective:
  - Finding distance of shortest path between a source vertex and goal vertex



Directed Acyclic Graph

# How to represent the Weighted graph?

- Nested Adjacency Dictionary

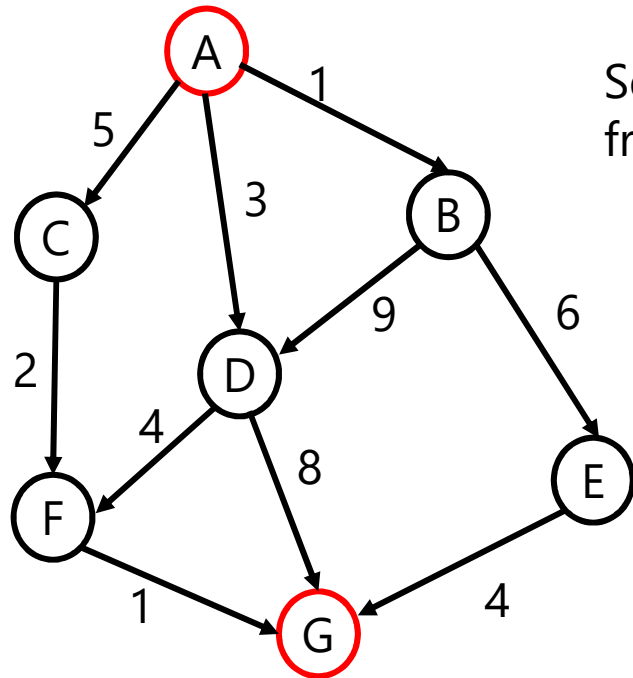


```
adjacencyGraph = { 'A': { 'B':1, 'C':5, 'D':3},  
                  'B' : { 'D':9, 'E':6},  
                  'C' : { 'F':2},  
                  'D' : { 'F':4, 'G':8},  
                  'E' : { 'G':4},  
                  'F' : { 'G': 1}}
```

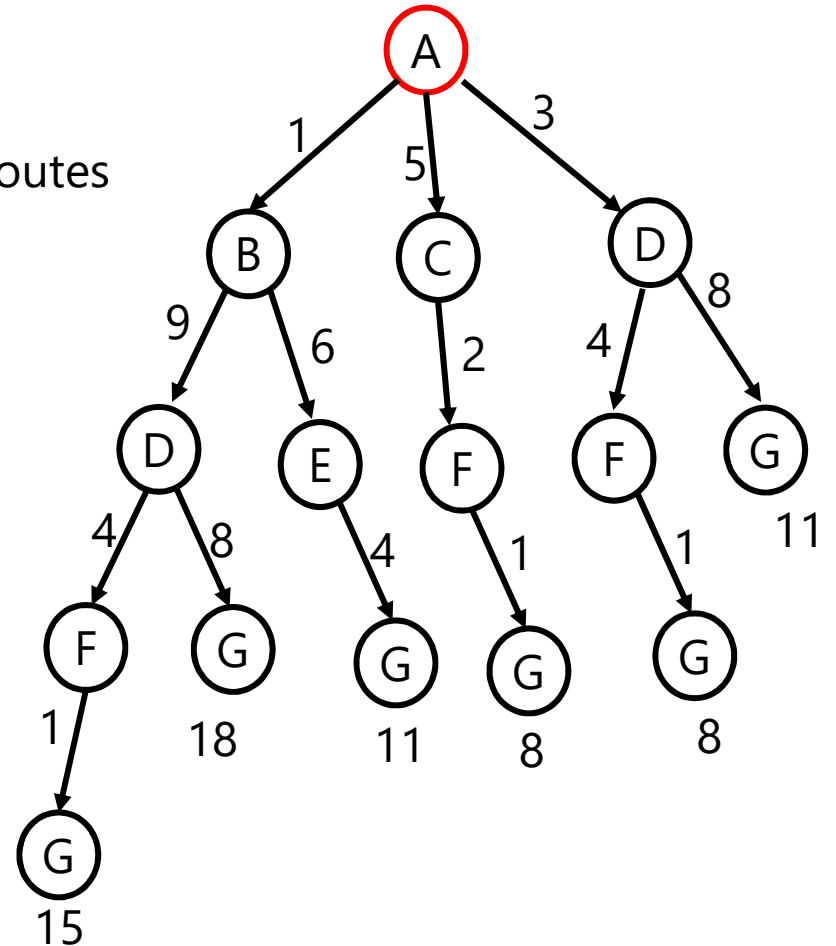
# Algorithms

- Exhaustive Search
- Dynamic Programming, etc.
- Dijkstra's Algorithm

# Exhaustive Search




Search all possible routes  
from A to G




# Recursive Algorithm

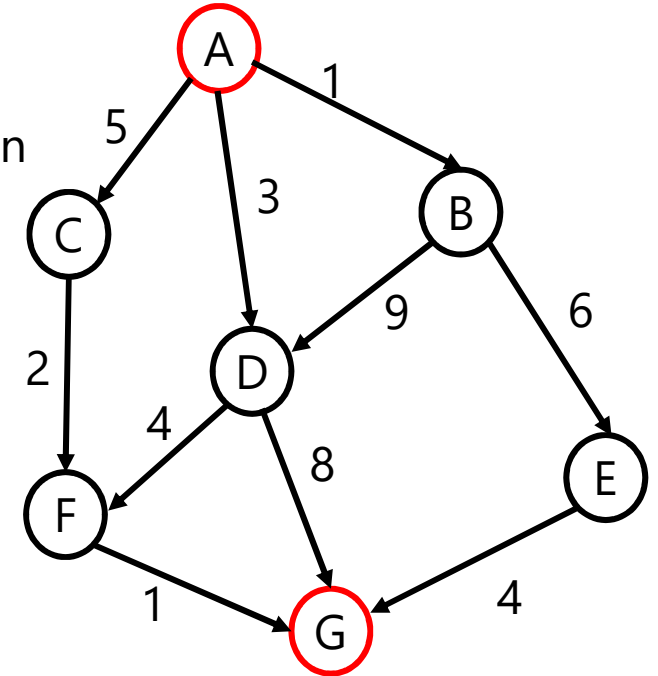
- Assumption
  - Shortest distance from neighbours of  $A$  to  $G$  is known
- Shortest distance from  $A$  to  $G$  is
  - $\min\{d(A, v) + d(v, G)\}, v \in \{B, C, D\}$



Distance from  $A$   
to its neighbour  $v$



Distance from  
neighbour  $v$  to  $G$
- But how do you find distance from neighbour  $v$  to  $G$ ?
  - Repeat the above process, look at its neighbours and select shortest distance to  $G$
  - Till  $G$  is found





# Shortest Distance between Two Nodes

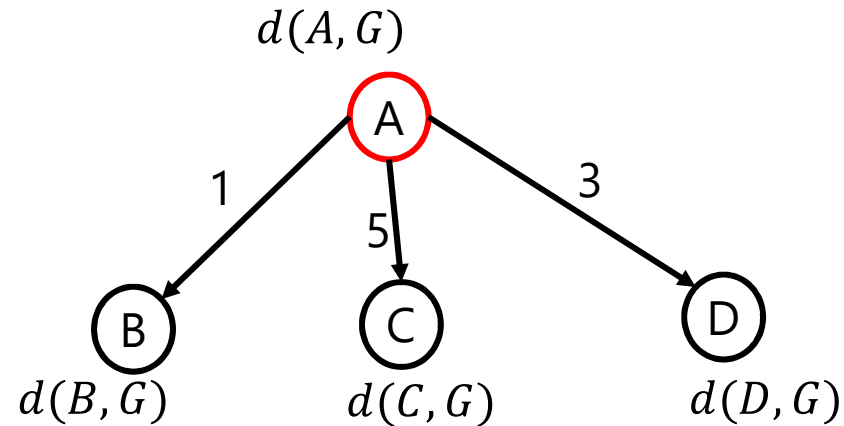
```
import math
def least_cost(adjDict, source, target):
    def d(vertex):
        if vertex == target:
            return 0
        try:
            return min(adjDict[vertex][i]+d(i) for i in adjDict[vertex])
        except:
            return math.inf
    return d(source)
```

neighbours of vertex

$d(A, v)$   
obtained from problem

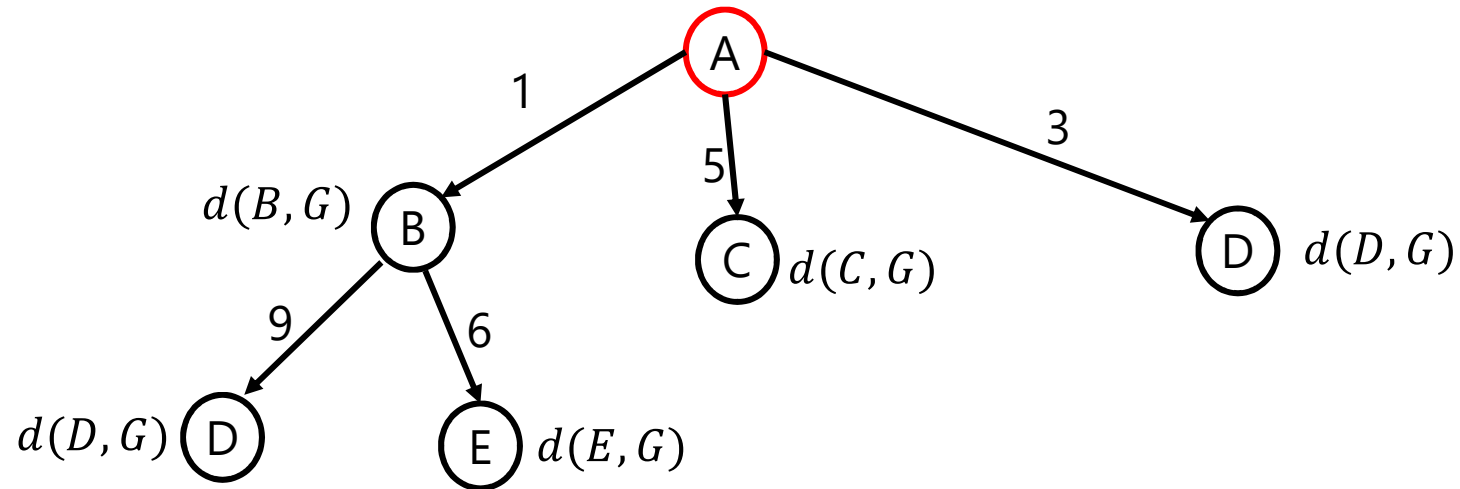
recursive function call  $d(v, G)$

# Shortest Distance: Recursive Method



$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

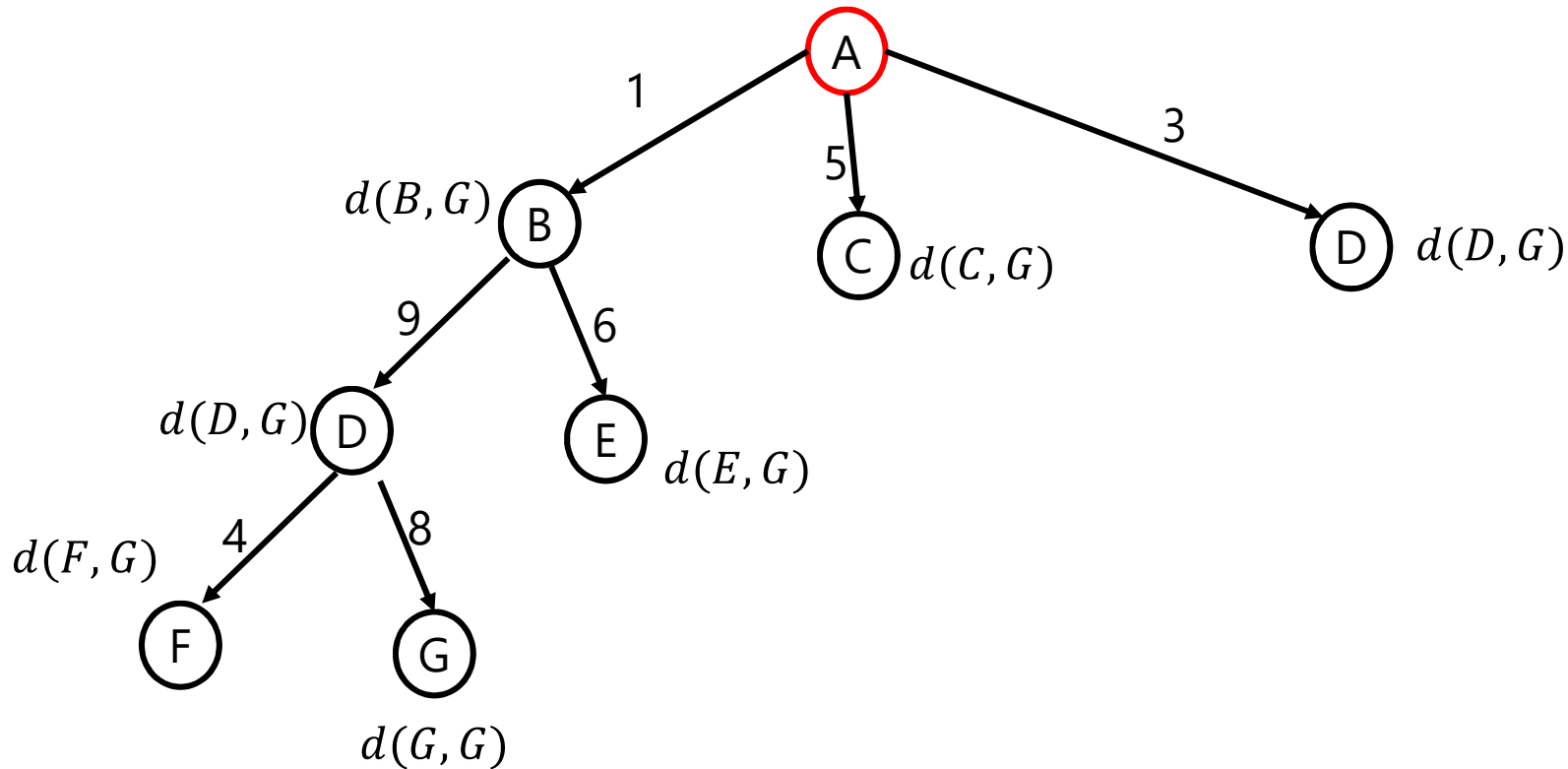
# Shortest Distance: Recursive Method



$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(B, G) = \min\{9 + d(D, G), 6 + d(E, G)\}$$

# Shortest Distance: Recursive Method

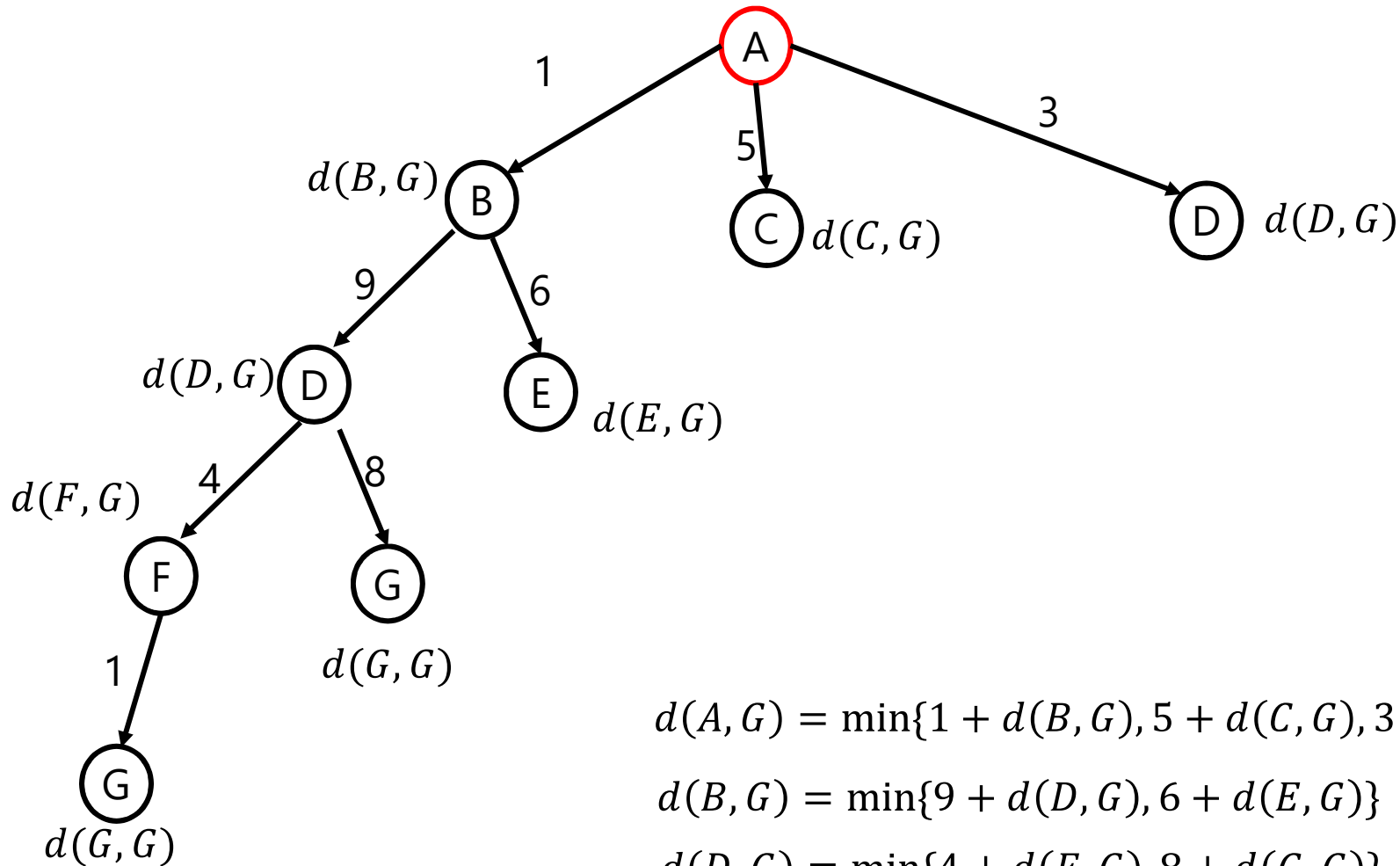


$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(B, G) = \min\{9 + d(D, G), 6 + d(E, G)\}$$

$$d(D, G) = \min\{4 + d(F, G), 8 + d(G, G)\}$$

# Shortest Distance: Recursive Method



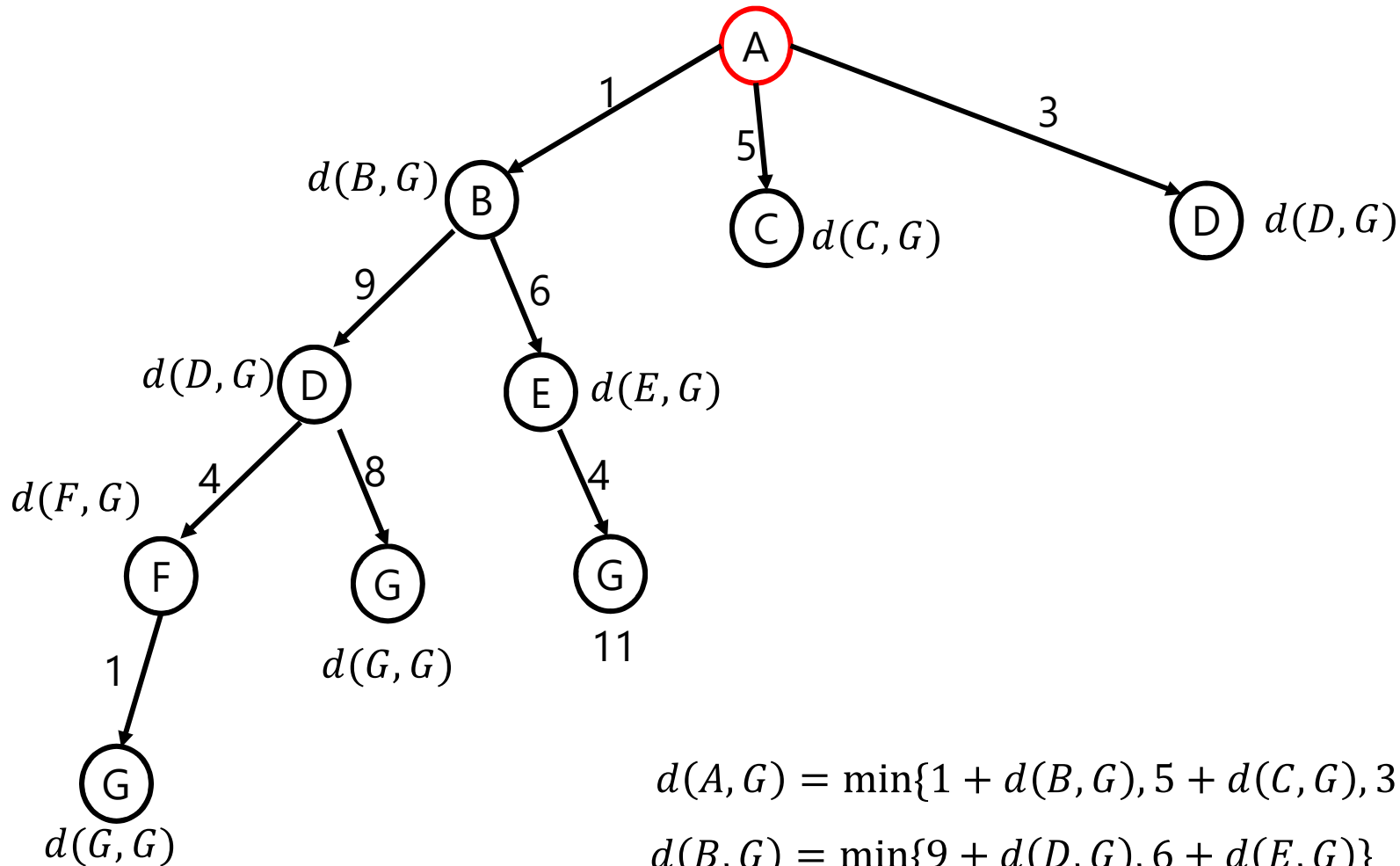
$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(B, G) = \min\{9 + d(D, G), 6 + d(E, G)\}$$

$$d(D, G) = \min\{4 + d(F, G), 8 + d(G, G)\}$$

$$d(F, G) = 1 + d(G, G)$$

# Shortest Distance: Recursive Method

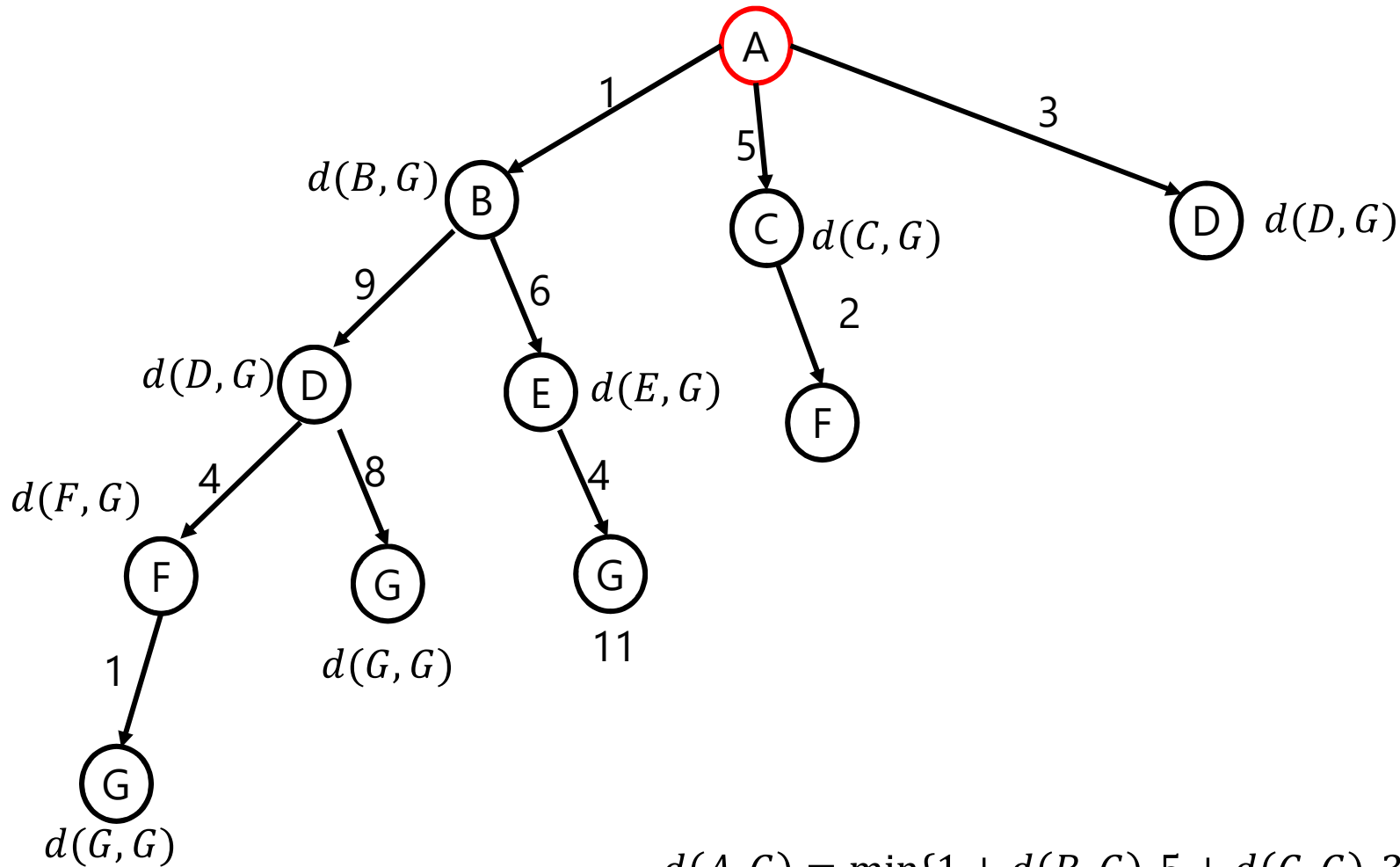


$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(B, G) = \min\{9 + d(D, G), 6 + d(E, G)\}$$

$$d(E, G) = 4 + d(G, G)$$

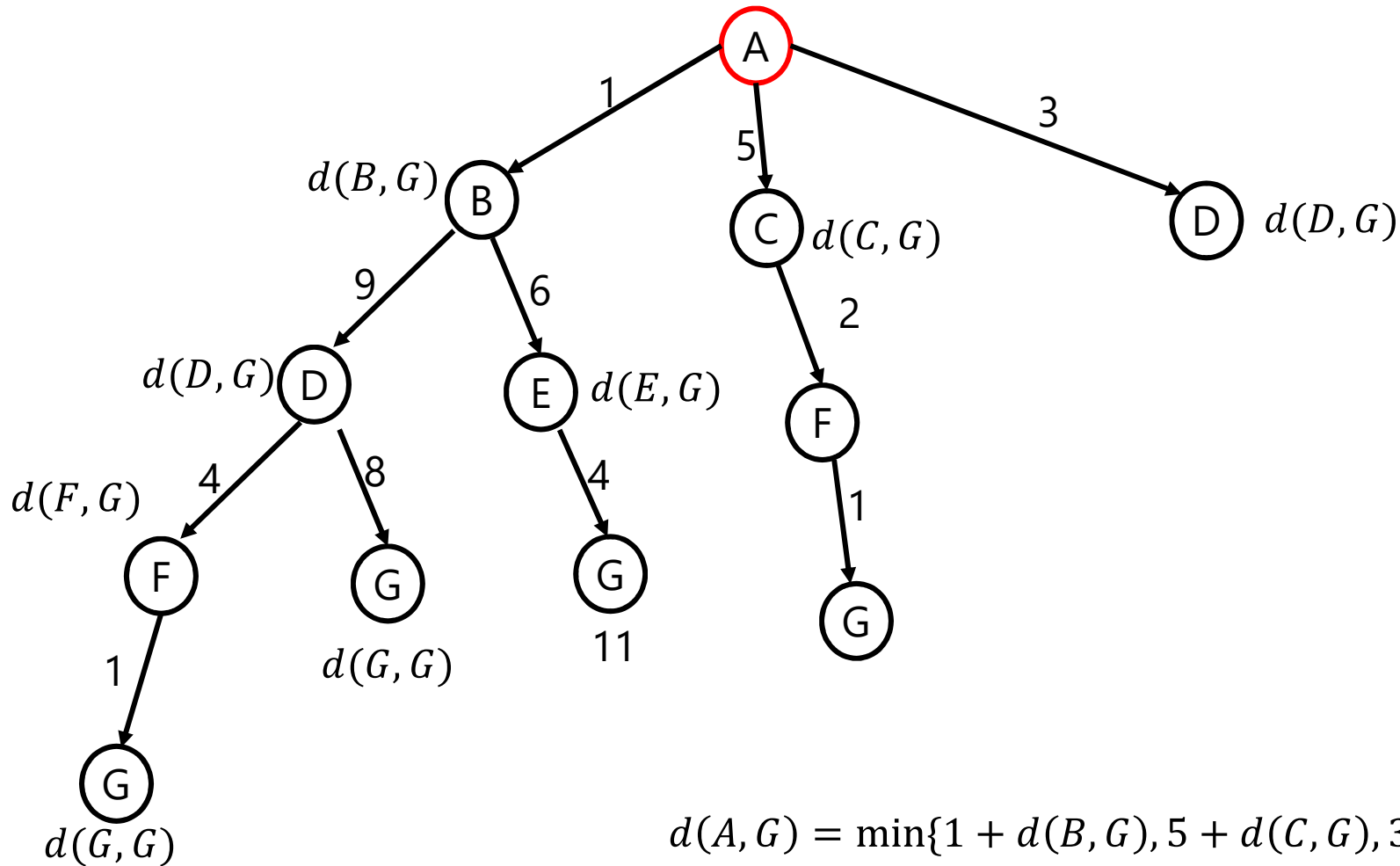
# Shortest Distance: Recursive Method



$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(C, G) = 2 + d(F, G)$$

# Shortest Distance: Recursive Method



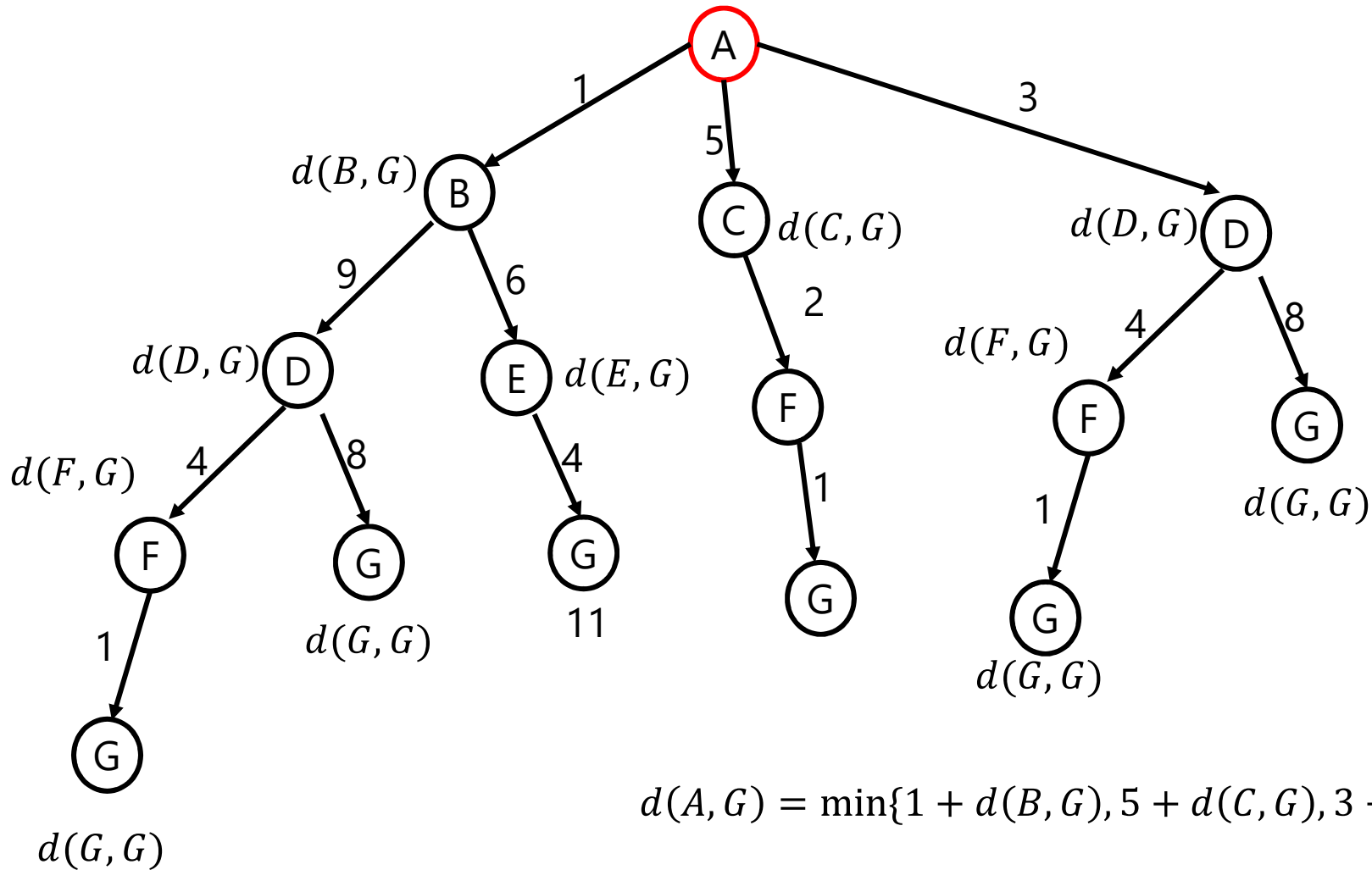
$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(C, G) = 2 + d(F, G)$$

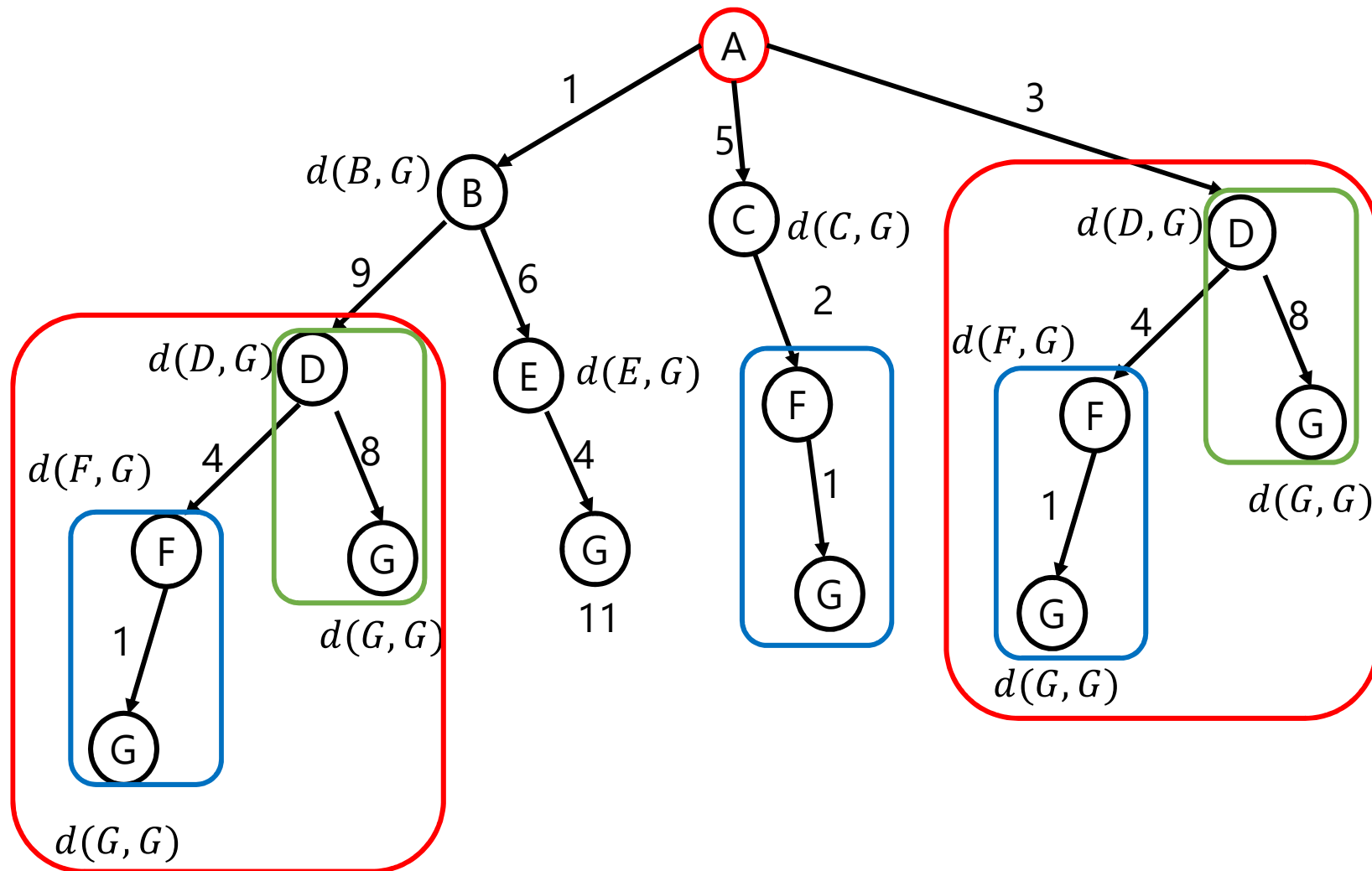
$$d(F, G) = 1 + d(G, G)$$



# Shortest Distance: Recursive Method



# Shortest Distance: Recursive Method



# Shortest Distance: Memoized Recursive Version

```
import math
def least_cost(adjDict, source, target):
    @cache
    def d(vertex):
        if vertex == target:
            return 0
        try:
            return min(adjDict[vertex][i]+d(i) for i in adjDict[vertex])
        except:
            return math.inf

    return d(source)

print(least_cost(adjDict, 'A', 'G'))
```

# Summary

- Dynamic Programming
  - General Problem Solving
    - Divide-and-conquer with redundant or overlapping subproblems
  - Optimization
    - Solving problems that have overlapping subproblems with optimal solutions
    - Subproblem dependency should be acyclic (i.e., only for DAGs)
  - Python's decorators simplifies memoization in DP