DTS203TC
Design and Analysis of Algorithms

Lecture 16: Graph algorithms

Dr. Qi Chen and Pascal Lefevre
School of AI and Advanced Computing

Outline 1/2

1. Graph representations
    1. Basic representation
    2. Adjacency list
    3. Adjacency matrix
2. Graph traversal
    1. BFS
        1. Definition
        2. Word Ladder (Leetcode 127, Hard)
    2. DFS
        1. Definition
        2. Clone Graph (Leetcode 133, Medium)

Outline 2/2

3. Minimum Spanning Tree
    1. Spanning trees
    2. Prim's algorithm
    3. Min Cost to Connect all Points (Leetcode 1584, Medium)
4. Shortest Path Problem
    1. Paths
    2. Dijkstra's algorithm
    3. Network Delay Time (Leetcode 743, Medium)

1. Graph representations

Basic representation

- Let G = (V, E) a graph
  - V = {v_1, v_2, …, v_n} the set of vertices
  - E = {e_1, …, e_m} the set of edges

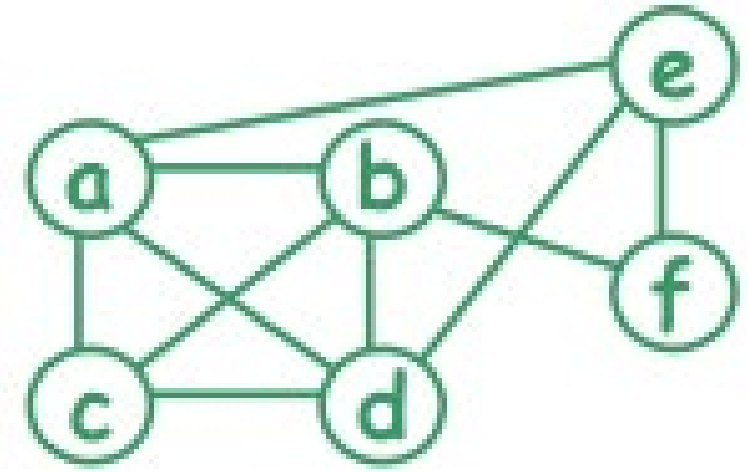
- Notes
  - e = (a, b) is an edge from a to b with a, b in V
  - (b, a) is not necessarily in V, if G is directed

Basic representation

- Let G = (V, E) a graph
  - V = {v_1, v_2, …, v_n} the set of vertices
  - E = {e_1, …, e_m} the set of edges

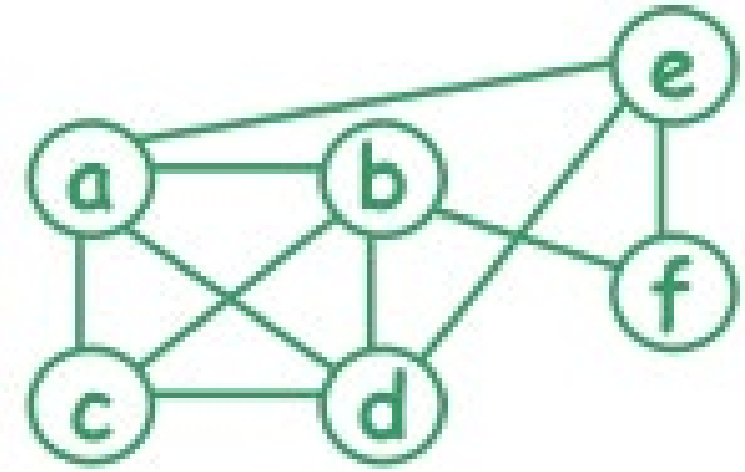
- Example
  - V = {a, b, c, d, e, f}
  - E = {(a, e), (a, b), (b, f),
    (c, b), (a, d), (c, d),
    (a, c), (b, d), (e, f)}

  (undirected notation)



Basic representation

- $G = (V, E)$
  - $V = \{a, b, c, d, e, f\}$
  - $E = \{(a, e), (a, b), (b, f), (c, b), (a, d), (c, d), (a, c), (b, d), (e, f)\}$

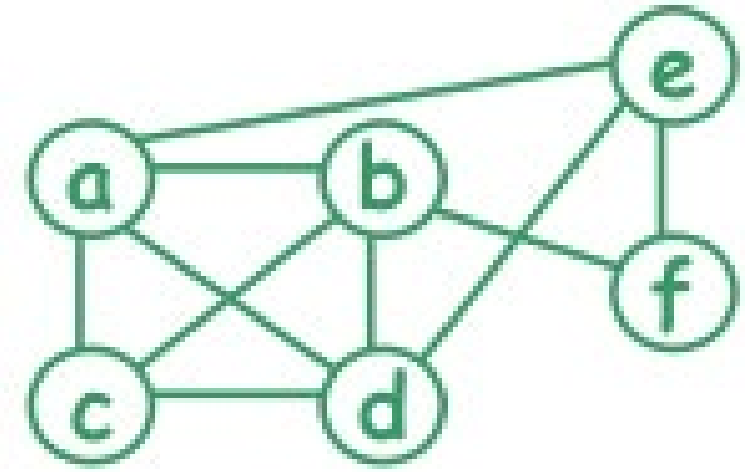


In [4]:
```
# Graph representation in Python

V = ['a', 'b', 'c', 'd', 'e', 'f']
E = [('a', 'e'), ('a', 'b'), ('b', 'f'), ('c', 'b'), ('a', 'd'), ('c', 'd'), ('a', 'c'), ('b', 'd'), ('e', 'f')]
G = (V, E)

print(V)
print()
print(E)
print()
print(G)
```

```
['a', 'b', 'c', 'd', 'e', 'f']

[('a', 'e'), ('a', 'b'), ('b', 'f'), ('c', 'b'), ('a', 'd'), ('c', 'd'), ('a', 'c'), ('b', 'd'), ('e', 'f')]

(['a', 'b', 'c', 'd', 'e', 'f'], [('a', 'e'), ('a', 'b'), ('b', 'f'), ('c', 'b'), ('a', 'd'), ('c', 'd'), ('a', 'c'), ('b',
'd'), ('e', 'f')])
```

Adjacency list

- G = {(a, (bcd)), (b, (acdf)), (c, (abd)), (d, (acbe)), (e, (adf)), (f, (be))}
- Eg: if a points to b, c and d, then we write (a, (bcd))

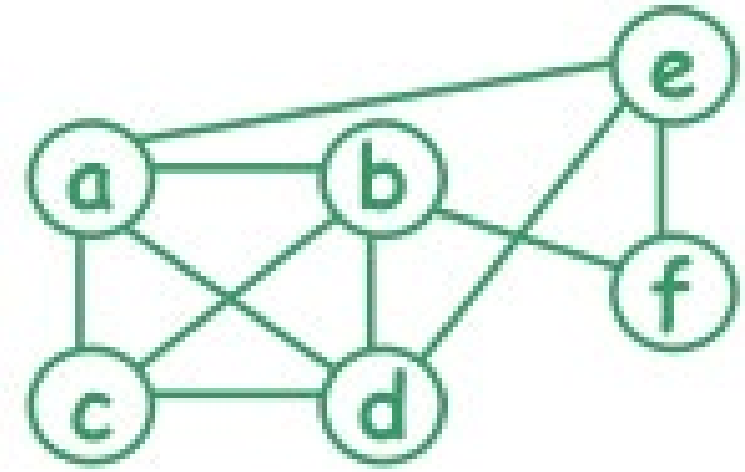



```
In [6]: # Adjacency graph representation
        # we use a Python dictionary to represent the graph with an adjacency list
        G = {
            'a': ['b', 'c', 'd'],
            'b': ['a', 'c', 'd', 'f'],
            'c': ['a', 'b', 'd'],
            'd': ['a', 'c', 'b', 'e'],
            'e': ['a', 'd', 'f'],
            'f': ['b', 'e'],
        }

        print(G)
```

```
{'a': ['b', 'c', 'd'], 'b': ['a', 'c', 'd', 'f'], 'c': ['a', 'b', 'd'], 'd': ['a', 'c', 'b', 'e'], 'e': ['a', 'd', 'f'], 'f':
['b', 'e']}
```

Adjacency matrix

- Adjacency matrix M of a (directed) graph G
- M(i, j) = 1 if i points to j (edge)
- M(i, j) = 0 otherwise
- In our case, G is undirected, so M(i, j) = M(j, i)



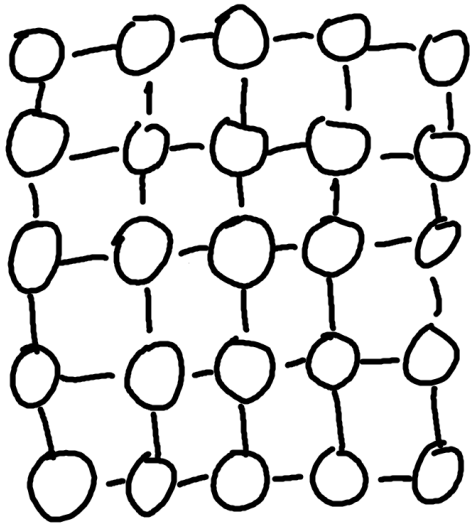
In [7]:
```
# Adjacency matrix graph representation
# we use a Python double list to represent the matrix M

# 1 <= i <= 6, 1 <= j <= 6

#    1, 2, 3, 4, 5, 6
#    a, b, c, d, e, f
M = [
    [0, 1, 1, 1, 1, 0],  # a 1
    [1, 0, 1, 1, 0, 1],  # b 2
    [1, 1, 0, 1, 0, 0],  # c 3
    [1, 1, 1, 0, 1, 0],  # d 4
    [0, 1, 0, 0, 1, 0],  # e 5
    [0, 1, 0, 0, 1, 0]   # f 6
]

print(M)
```

```
[[0, 1, 1, 1, 1, 0], [1, 0, 1, 1, 0, 1], [1, 1, 0, 1, 0, 0], [1, 1, 1, 0, 1, 0], [0, 1, 0, 0, 1, 0], [0, 1, 0, 0, 1, 0]]
```

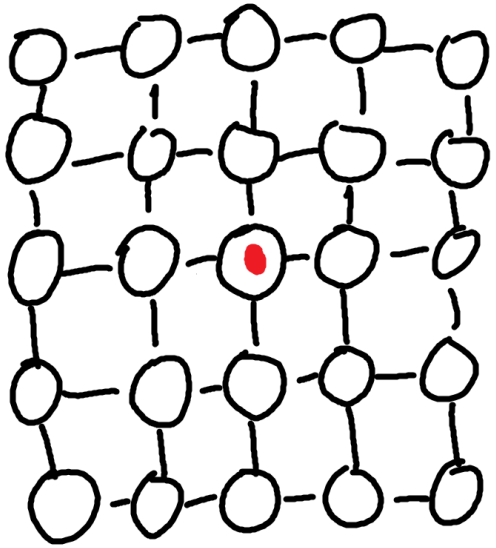
2. Graph Traversal – BFS 1/2

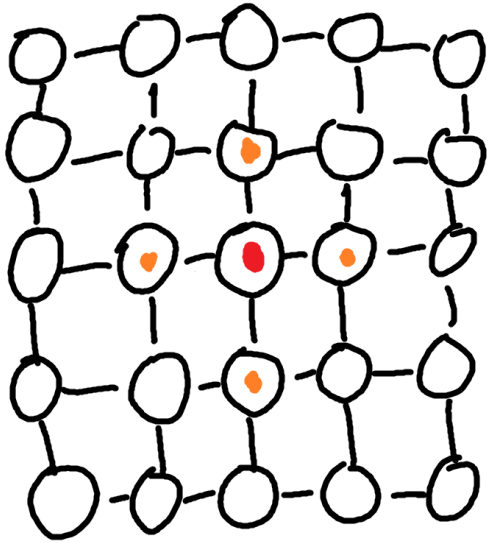
Breadth First Search (BFS)

- BFS is a traversal algorithm
- Vertex exploration by neighborhood
- Select a starting node and explore layer by layer

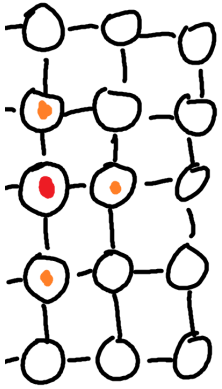


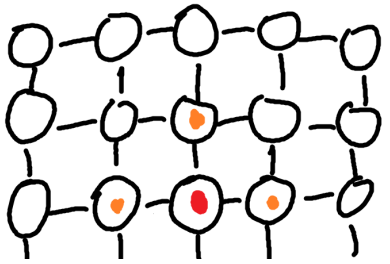
Grid graph example





First layer exploration

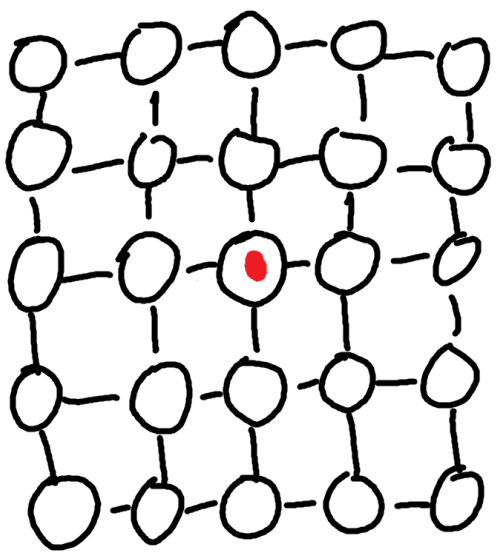




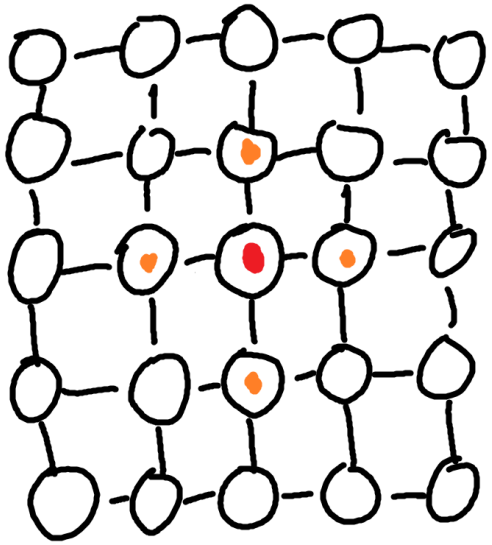
Other graph configurations

Breadth First Search (BFS)

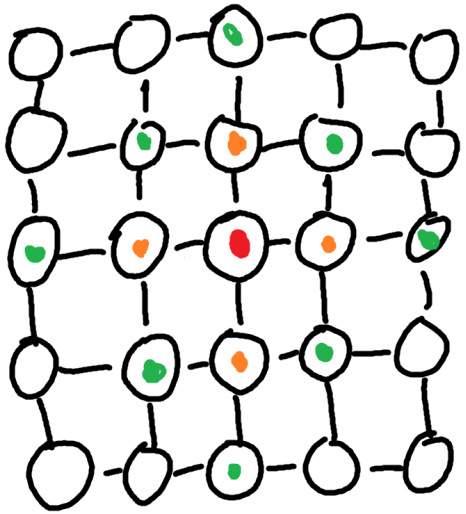
- Vertex exploration by neighborhood
- First layer discovery, second, third, until all the node are visited



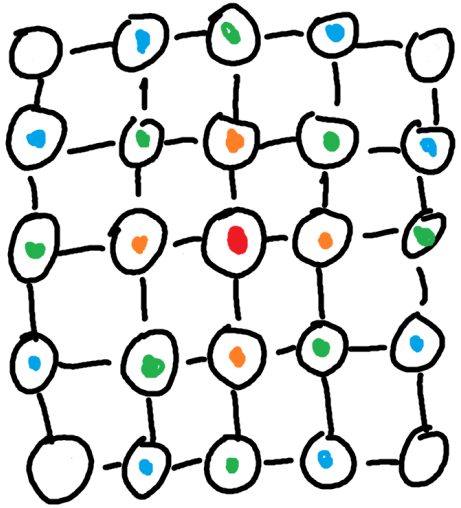
Starting vertex



First layer



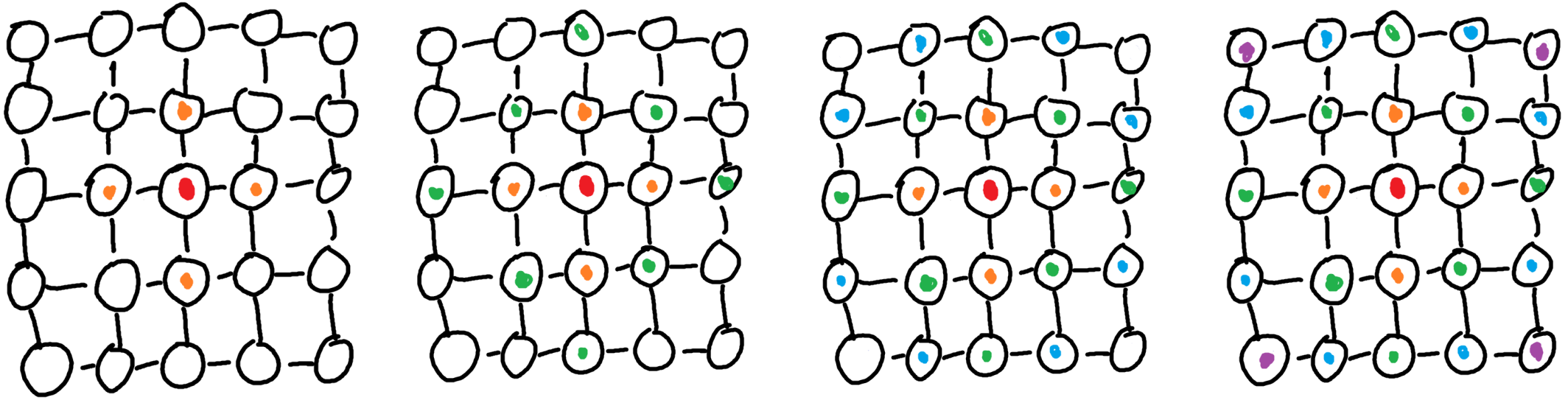
Second layer



Third layer

Breadth First Search (BFS)

- Vertex exploration by neighborhood
- 4 layers from the starting vertex



Breadth First Search (BFS)

- Algorithm

BFS – Pseudo code

unmark all vertices
choose some starting vertex s
mark s and insert s into tail of list L
while L is nonempty do
begin
    remove a vertex v from front of L
    visit v
    for each unmarked neighbor w of v do
        mark w and insert w into tail of list L
end

```
visited = [] # List of visited nodes
queue = []   # empty queue

def bfs(visited, graph, starting_node):

    visited.append(starting_node)
    queue.append(starting_node)

    while queue:                # loop to visit each node
        m = queue.pop(0)        # remove the first element to be displayed
        print (m, end = " ")  # display the visited node

        for neighbour in graph[m]:  # loop through all the neighbours
            if neighbour not in visited:
                visited.append(neighbour)  # mark neighbour as visited
                queue.append(neighbour)
```

Breadth First Search (BFS)

```
# BFS in Python

graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List of visited nodes
queue = []   # empty queue

def bfs(visited, graph, starting_node):

    visited.append(starting_node)
    queue.append(starting_node)

    while queue:              # loop to visit each node
        m = queue.pop(0)      # remove the first element to be displayed
        print (m, end = " ")  # display the visited node

        for neighbour in graph[m]:  # loop through all the neighbours in the adjacency list
            if neighbour not in visited:
                visited.append(neighbour)  # mark neighbour as visited
                queue.append(neighbour)

# test code with starting node '5'
bfs(visited, graph, '5')    # function call
```

```
5 3 7 2 4 8
```

Breadth First Search (BFS)

127. Word Ladder

Hard | Topics | Companies

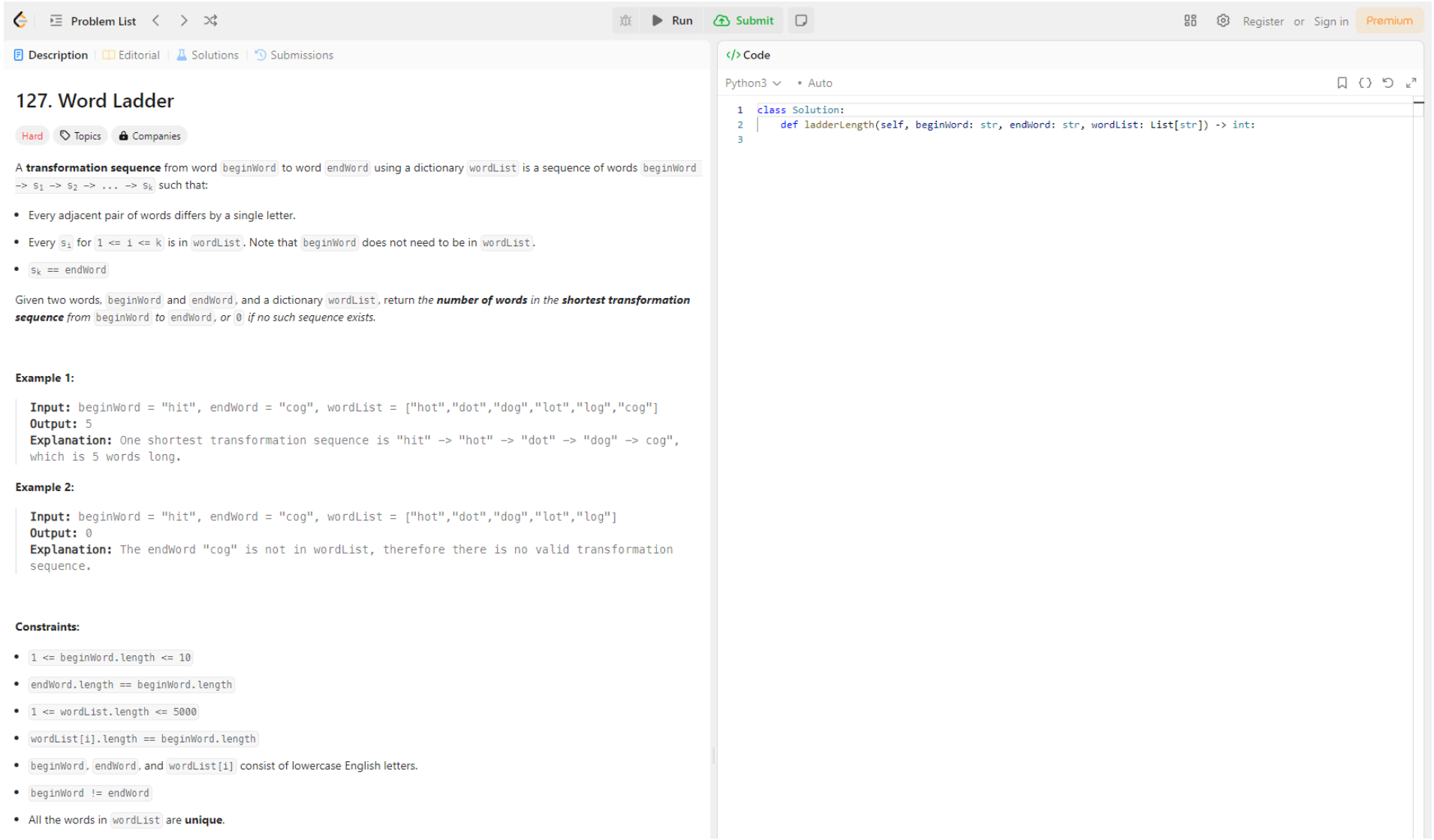
A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* `beginWord` *to* `endWord`, *or* `0` *if no such sequence exists.*

- https://leetcode.com/problems/word-ladder

Breadth First Search (BFS)



Problem List | Run | Submit | Register or Sign in | Premium

Description | Editorial | Solutions | Submissions

127. Word Ladder

Hard | Topics | Companies

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* `beginWord` *to* `endWord`, *or* `0` *if no such sequence exists.*

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog",
which is 5 words long.
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation
sequence.
```

Constraints:

- `1 <= beginWord.length <= 10`
- `endWord.length == beginWord.length`
- `1 <= wordList.length <= 5000`
- `wordList[i].length == beginWord.length`
- `beginWord`, `endWord`, and `wordList[i]` consist of lowercase English letters.
- `beginWord != endWord`
- All the words in `wordList` are **unique**.

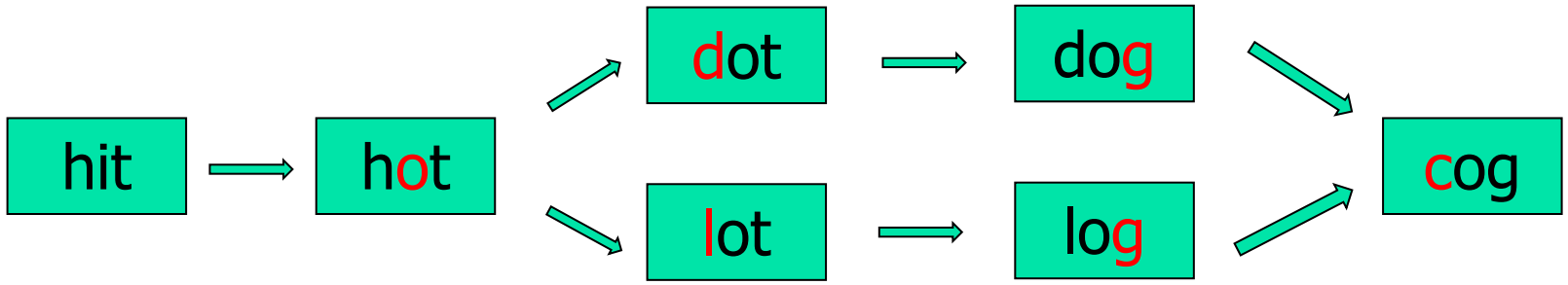
Code

Python3 • Auto

```
1  class Solution:
2      def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
3
```

Breadth First Search (BFS)

Example 1: beginWord = hit, endWord = cog
wordList = [hot, dot, dog, lot, log, cog]
The shortest transformation is
- Hit -> hot -> dot -> dog -> cog (5 words)



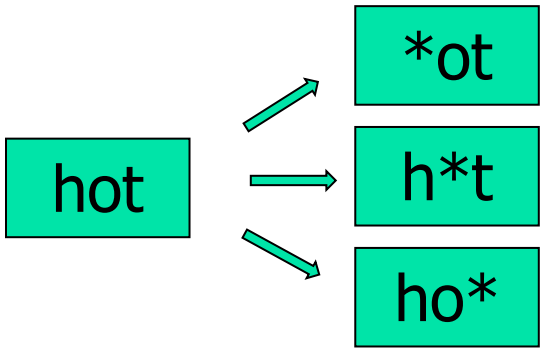
From s_k to s_{k+1}: only 1 character change allowed

Breadth First Search (BFS)

wordList = [“hot”, “dot”, “dog”, “lot”, “log”, “cog”]

Solution

1. Build the adjacency list of the graph



Neighbors of hot by finding the pattern matched words
{*ot: [hot, dot, lot]}
{h*t: [hit]}
{ho*: []}
Neighbors: {hot: [dot, lot]}

2. Perform BFS to find the shortest path

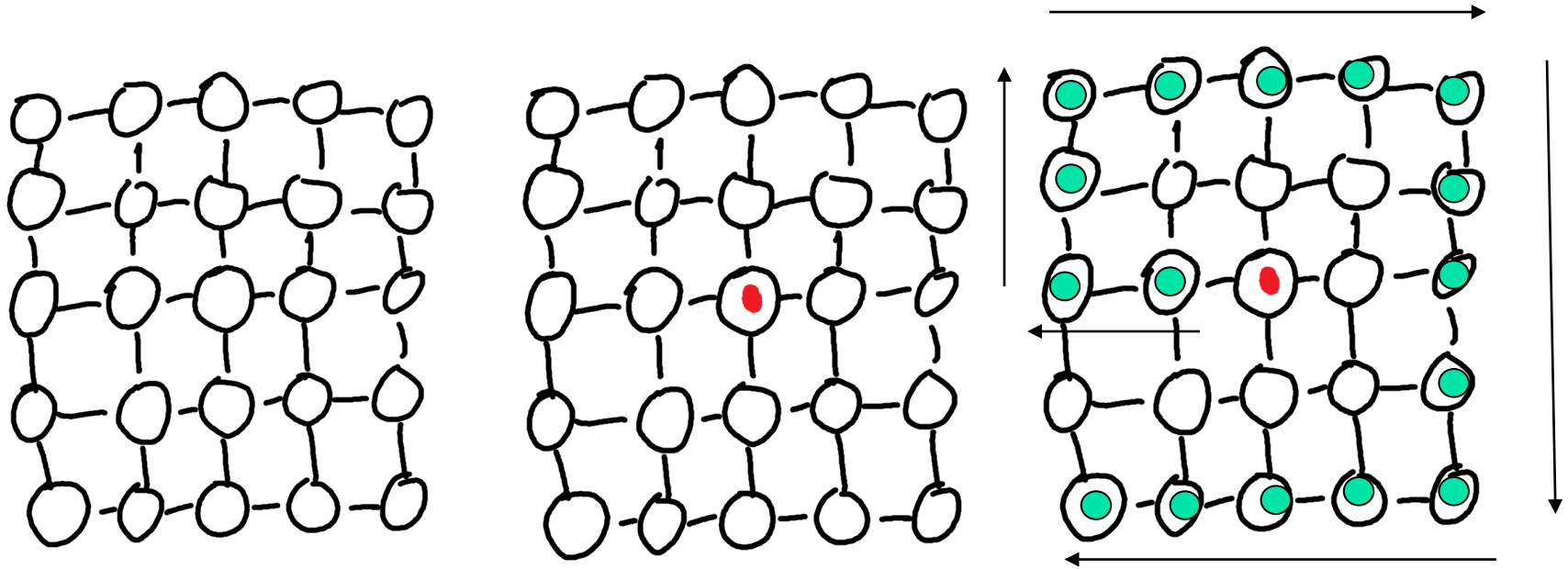
Breadth First Search (BFS)

```python
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        if endWord not in wordList:
            return 0

        nei = collections.defaultdict(list)
        wordList.append(beginWord)
        for word in wordList:
            for j in range(len(word)):
                # add the pattern *
                pattern = word[:j] + '*' + word[j + 1:]
                nei[pattern].append(word)

        visit = set([beginWord])
        q = deque([beginWord])
        res = 1
        # BFS loop, with BFS we are guaranteed to find the shortest path
        while q:
            for i in range(len(q)):
                word = q.popleft()
                if word == endWord:
                    return res
                for j in range(len(word)):  # move the * inside the word
                    pattern = word[:j] + '*' + word[j + 1:]
                    for neiWord in nei[pattern]:
                        if neiWord not in visit:
                            visit.add(neiWord)
                            q.append(neiWord)
            res += 1
        return 0
```

2. Graph Traversal – DFS 2/2

Depth First Search (DFS)

- DFS is a traversal algorithm
- Vertex exploration by depth
- Select a starting node and explore until all the nodes are visited

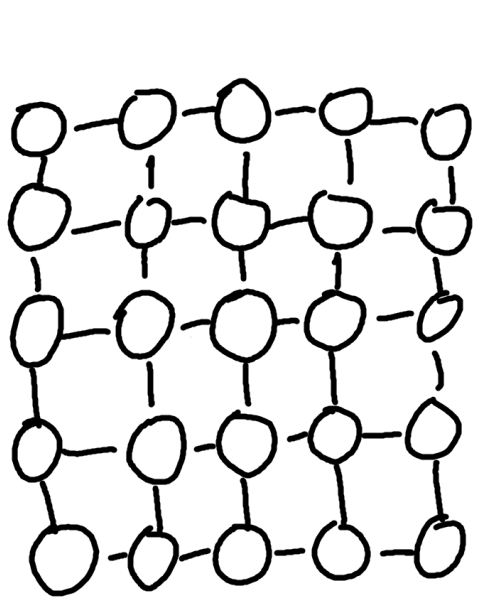


Grid graph example

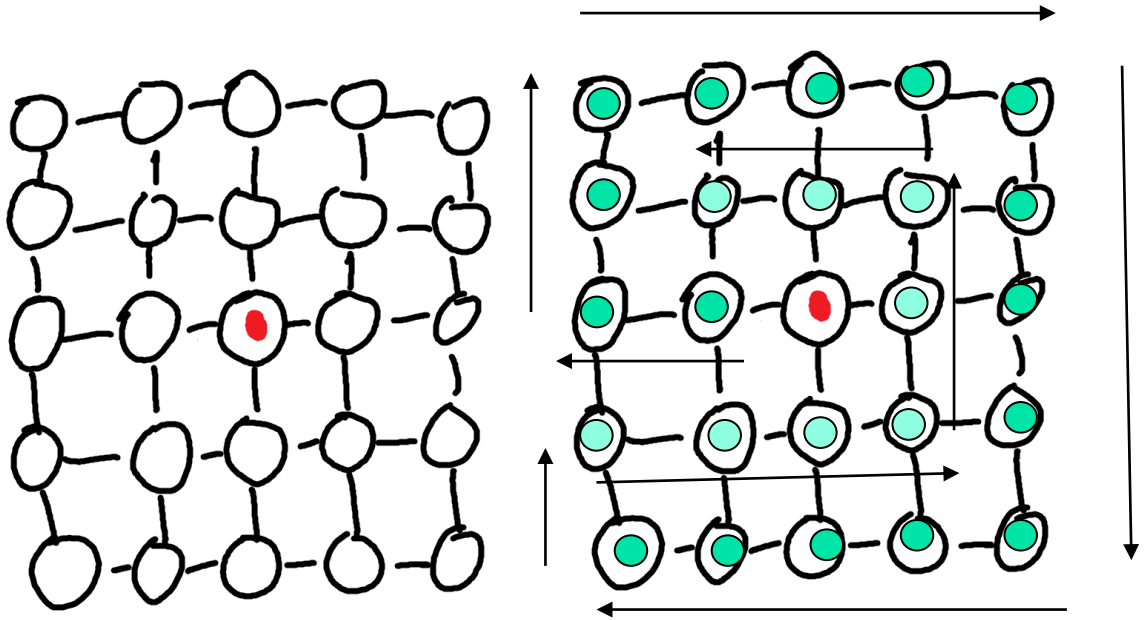
First part of the first path

Depth First Search (DFS)

- DFS is a traversal algorithm
- Vertex exploration by depth
- Select a starting node and explore until all the nodes are visited



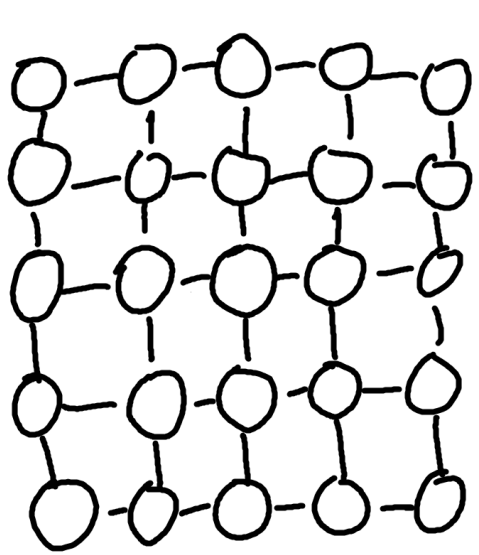
Grid graph example



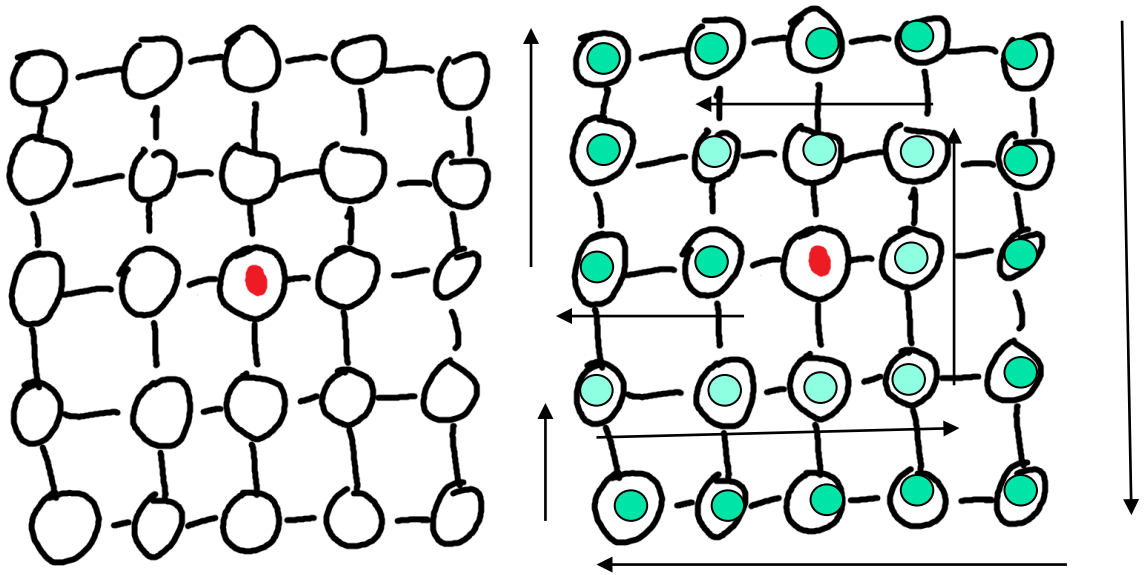
Second part of the first path

Depth First Search (DFS)

- DFS is a traversal algorithm
- Vertex exploration by depth
- Select a starting node and explore until all the nodes are visited
- In this example, we visit all the nodes in one path (depends on the adjacency list)



Grid graph example



Second part of the first path

Depth First Search (DFS)

- Algorithm

DFS – Pseudo code (recursive)

```
Algorithm DFS(vertex v)
    visit v
    for each unvisited neighbor w of v do
    begin
        DFS(w)
    end
```

```python
visited = set() # visited nodes

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)  # recursive call
```

Depth First Search (DFS)

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # visited nodes

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)  # recursive call

# test code with node '5'
dfs(visited, graph, '5')
```

```
5
3
2
4
8
7
```

Depth First Search (DFS)

133. Clone Graph

Medium | Topics | Companies

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

- https://leetcode.com/problems/clone-graph

Depth First Search (DFS)

Problem List | Run | Submit | Premium

Description | Editorial | Solutions | Submissions

133. Clone Graph

Medium | Topics | Companies

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.

9.3K | 116

Code

Python3 | Auto

```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []
"""

from typing import Optional
class Solution:
    def cloneGraph(self, node: Optional['Node']) -> Optional['Node']:

```

Saved to local | Ln 1, Col 1

Testcase | Test Result

Depth First Search (DFS)

```
"""
# Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []
"""

from typing import Optional
class Solution:
    def cloneGraph(self, node: Optional['Node']) -> Optional['Node']:
        oldToNew = {}  # hashmap or dictionary old node -> new node

        def dfs(node):
            if node in oldToNew:  # we can finish dfs directly if the new node already exists
                return oldToNew[node]

            copy = Node(node.val)  # otherwise we make a copy
            oldToNew[node] = copy  # and store the copy inside the hashmap
            for nei in node.neighbors:  # visit all old neighbors
                copy.neighbors.append(dfs(nei))  # store the new neighbor copies and add them to the new node's neighbors
            return copy  # returns a new node as copy with its neighbors (also copies)

        return dfs(node) if node else None  # return by a simple dfs call or None

# Note 1: there is also a BFS version of Clone graph problem
# Note 2: the dfs function is specialized (or adapted) for this problem
```

Depth First Search (DFS)

Problem List | Run | Submit | Premium

Description | Editorial | Solutions | Submissions

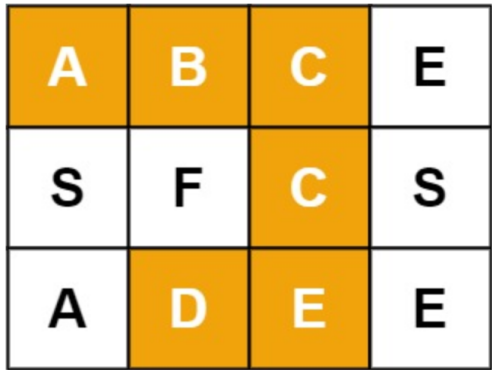
79. Word Search

Medium | Topics | Companies

Given an `m x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid.*

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:




```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],
word = "ABCCED"
Output: true
```

15K | 128

Code

Python3 | Auto

```
1  class Solution:
2      def exist(self, board: List[List[str]], word: str) -> bool:
3
```

Saved to local | Ln 1, Col 1

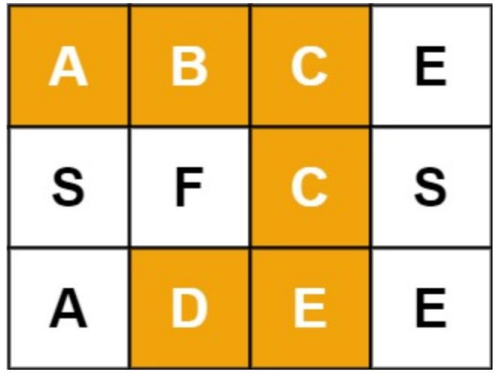
Testcase | Test Result

Depth First Search (DFS)

How to solve this problem?

- No simple or easy way
- Backtracking, brute force: DFS

Example 1:



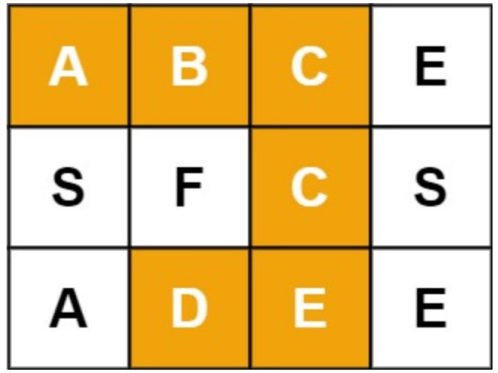

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],
word = "ABCCED"
Output: true
```

Depth First Search (DFS)

How to solve this problem?

- No simple or easy way
- Backtracking, brute force: DFS

Example 1:



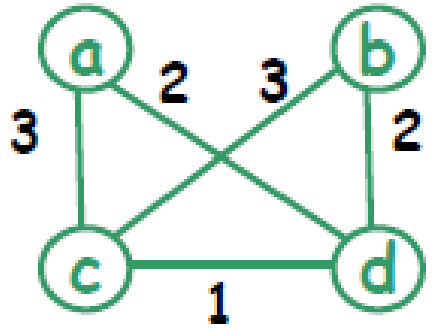
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],
word = "ABCCED"
Output: true

```python
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        ROWS, COLS = len(board), len(board[0])
        path = set()

        def dfs(r, c, i):
            if i == len(word):
                return True

            # checks if (r, c) is outside the board
            if (r < 0 or c < 0 or          # checks if (r, c) is outside the board
                r >= ROWS or c >= COLS or  # checks if (r, c) is outside the board
                word[i] != board[r][c] or  # checks if the character is matching the word
                (r, c) in path):           # checks if (r, c) is already visited
                return False

            # adds the path, run dfs in the 4 directions and removes the path
            path.add((r, c))
            res = (dfs(r + 1, c, i + 1) or
                   dfs(r - 1, c, i + 1) or
                   dfs(r, c + 1, i + 1) or
                   dfs(r, c - 1, i + 1))
            path.remove((r, c))
            return res

        for r in range(ROWS):
            for c in range(COLS):
                if dfs(r, c, 0):
                    return True
        return False
```

3. Minimum Spanning Tree

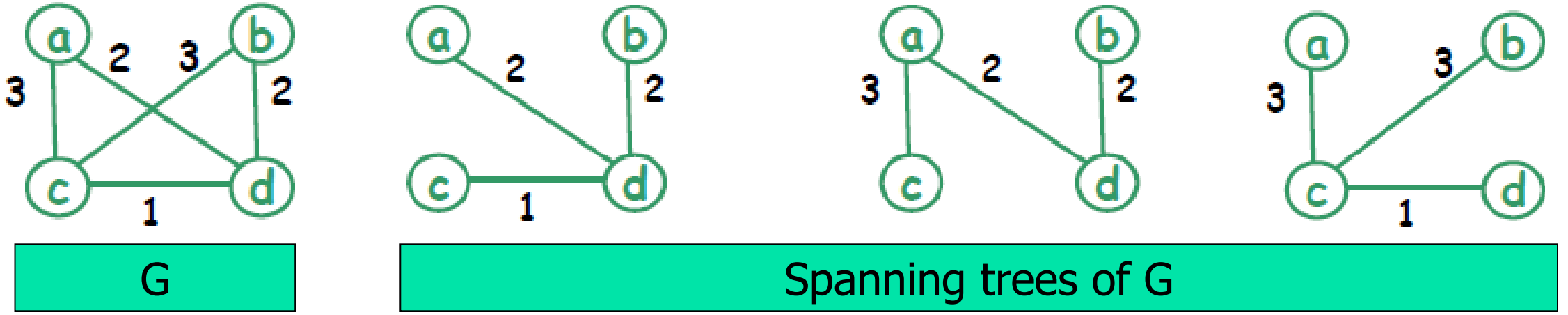
Spanning Tree

- A spanning tree T is a subgraph of a weighted graph G = (V, E)
- T contains all the vertices of V
- T does not necessarily contain all the edges in E
- Elements of E have weights w
- Example: V = {a, b, c, d}, E = {(ac, 3), (ad, 2), (bc, 3), (bd, 2), (cd, 1)}



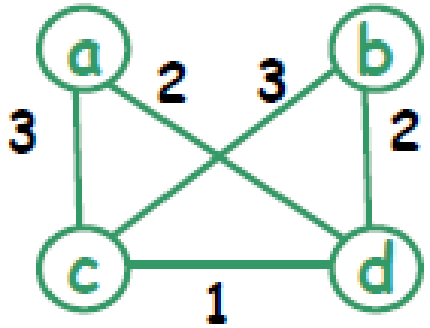
Spanning Tree

- A spanning tree T is a subgraph of a weighted graph G = (V, E)
- T contains all the vertices of V
- T does not necessarily contain all the edges in E
- Elements of E have weights w
- Example: V = {a, b, c, d}, E = {(ac, 3), (ad, 2), (bc, 3), (bd, 2), (cd, 1)}

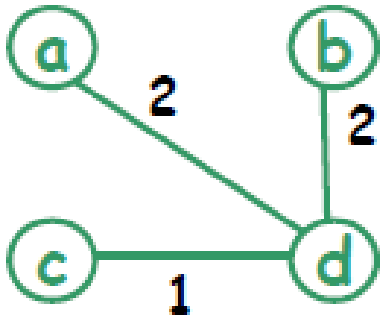


Spanning Tree

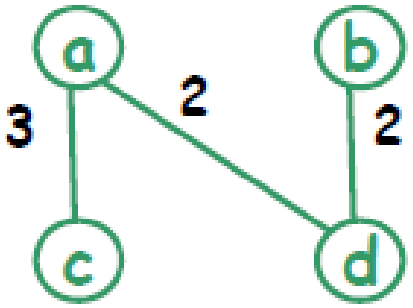
- A spanning tree T is a subgraph of a weighted graph G = (V, E)
- T contains all the vertices of V
- T does not necessarily contain all the edges in E
- Elements of E have weights w
- Example: V = {a, b, c, d}, E = {(ac, 3), (ad, 2), (bc, 3), (bd, 2), (cd, 1)}



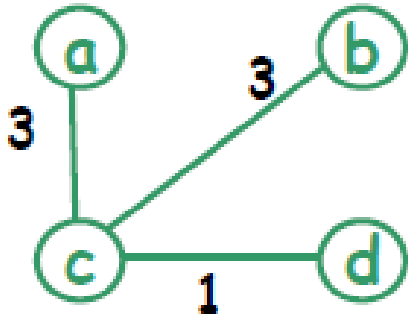
G



Cost: 2+2+1=5



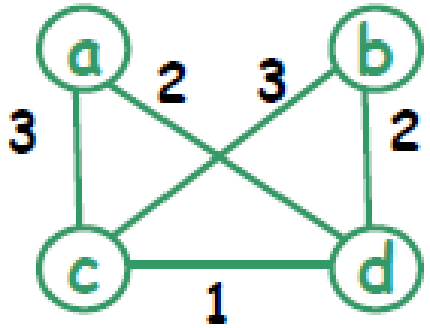
Cost: 3+2+2=7



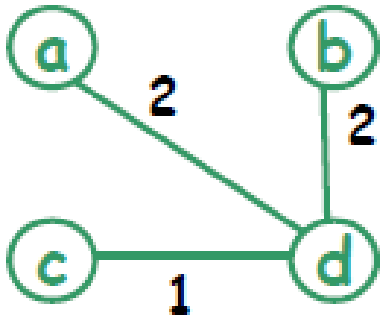
Cost:3+3+1=7

Spanning Tree

- A spanning tree T is a subgraph of a weighted graph G = (V, E)
- T contains all the vertices of V
- T does not necessarily contain all the edges in E
- Elements of E have weights w
- Example: V = {a, b, c, d}, E = {(ac, 3), (ad, 2), (bc, 3), (bd, 2), (cd, 1)}

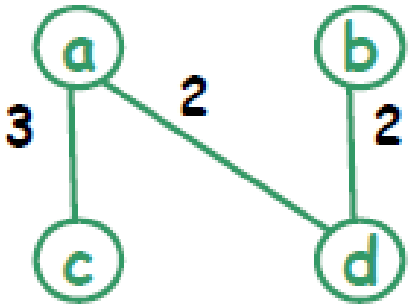


G

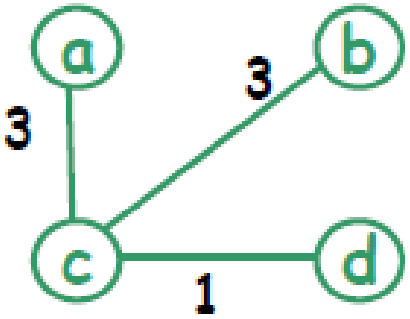


Cost: 5

Minimum Cost Spanning Tree



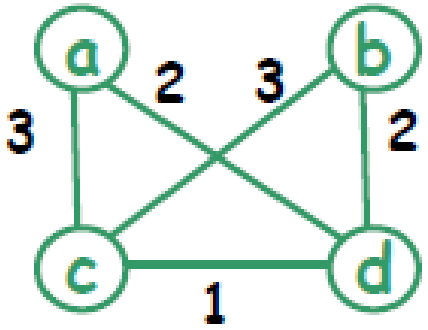
Cost: 7



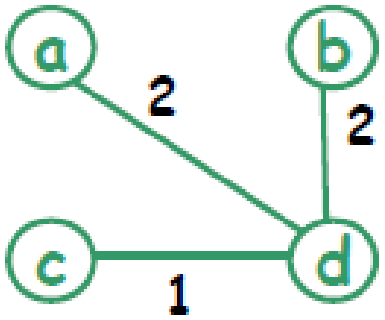
Cost: 7

Minimum Cost Spanning Tree

- Among all the spanning trees (ST) of *G*, find the ST with minimum cost



G



Cost: 5

Minimum Cost Spanning Tree

Prim's algorithm

- Algorithm to find the Minimum Cost Spanning Tree

Prim's algorithm

- Algorithm to find the Minimum Cost Spanning Tree
- Idea:
  1. Initialize with an arbitrary vertex
  2. Grow the tree by one edge from the leaves of the current tree to new vertices, find the minimum weight edge and add it to the tree
  3. Repeat until all the vertices are in the tree

Prim's algorithm

```
Prim(V, E, s)  // not an optimized version
        // V vertices, E edges
        // starting node s
        MST = {}
        visited = {s}
        while |visited| < |V|
                find lowest weighted edge (x, v)
                // x in visited, v not in visited
                add(x, v) to MST
                add v to visited
        return MST
```

```python
selected_node = [0, 0, 0, 0, 0]

no_edge = 0

selected_node[0] = True

# printing for edge and weight
print("Edge : Weight\n")
while (no_edge < N - 1):

    minimum = INF
    a = 0
    b = 0
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
    selected_node[b] = True
    no_edge += 1
```

Naïve implementation

Prim's algorithm

Naïve implementation

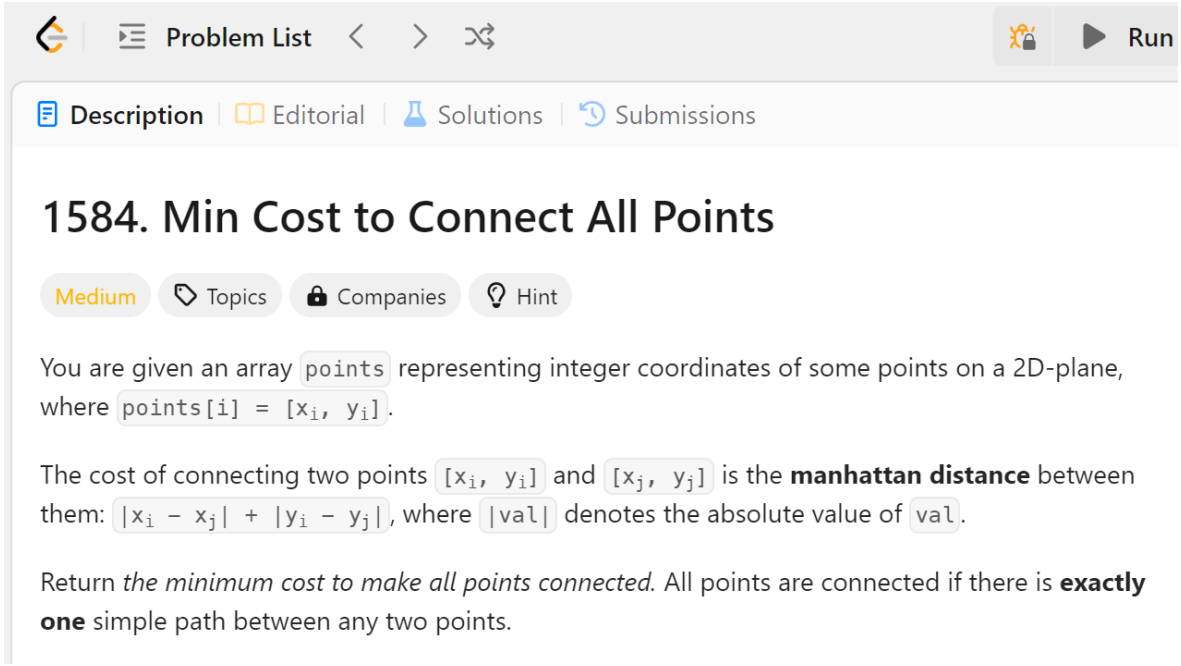
```
# Prim's Algorithm in Python

INF = 9999999
# number of vertices in graph
N = 5
#creating graph by adjacency matrix method
G = [[0, 19, 5, 0, 0],
     [19, 0, 5, 9, 2],
     [5, 5, 0, 1, 6],
     [0, 9, 1, 0, 1],
     [0, 2, 6, 1, 0]]
```

Output example

```
Edge : Weight

0-2:5
2-3:1
3-4:1
4-1:2
```

```
selected_node = [0, 0, 0, 0, 0]

no_edge = 0

selected_node[0] = True

# printing for edge and weight
print("Edge : Weight\n")
while (no_edge < N - 1):

    minimum = INF
    a = 0
    b = 0
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
    selected_node[b] = True
    no_edge += 1
```

Minimum Cost Spanning Tree



https://leetcode.com/problems/min-cost-to-connect-all-points

Minimum Cost Spanning Tree

Problem List | Run | Submit | Premium

Description | Editorial | Solutions | Submissions

1584. Min Cost to Connect All Points

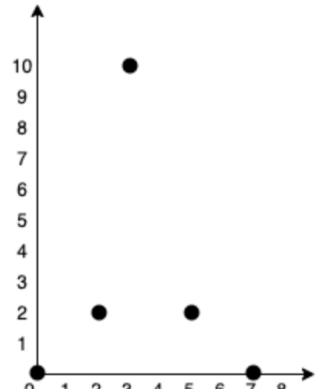
Medium | Topics | Companies | Hint

You are given an array `points` representing integer coordinates of some points on a 2D-plane, where `points[i] = [x_i, y_i]`.

The cost of connecting two points `[x_i, y_i]` and `[x_j, y_j]` is the **manhattan distance** between them: `|x_i - x_j| + |y_i - y_j|`, where `|val|` denotes the absolute value of `val`.

Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.

Example 1:



4.9K | 50

Code

Python3 | Auto

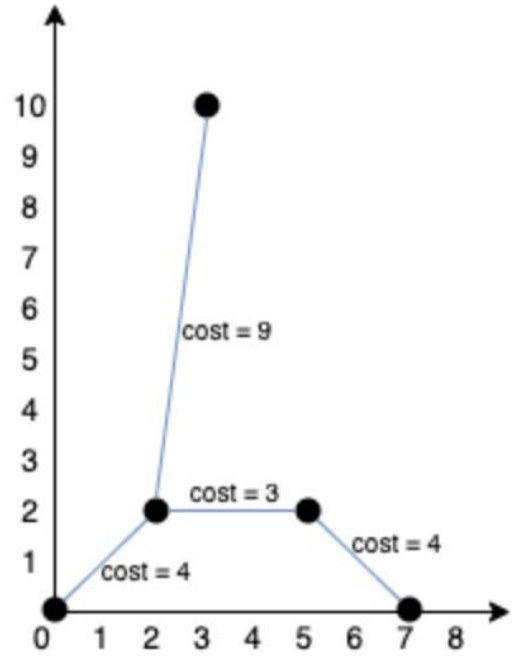
```
1  class Solution:
2      def minCostConnectPoints(self, points: List[List[int]]) -> int:
3
```

Saved to local | Ln 1, Col 1

Testcase | Test Result

Minimum Cost Spanning Tree

Input: points = [[0,0],[2,2],[3,10],[5,2],[7,0]]
Output: 20
Explanation:



We can connect the points as shown above to get the minimum cost of 20.
Notice that there is a unique path between every pair of points.

Minimum Cost Spanning Tree

The solution involves two data structures

- Set of visited nodes: to keep track and avoid cycles
- Min Heap: to pop out the frontier nodes of the tree

Minimum Cost Spanning Tree

```python
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        N = len(points)

        # Part 1: build the adjacency list for Prim's algorithm with distances
        # all the points are inter-connected and their cost is the Manhattan distance, pairwise
        adj = {i:[] for i in range(N)}  # i: list of [cost, node]
        for i in range(N):
            x1, y1 = points[i]
            for j in range(i + 1, N):
                x2, y2 = points[j]
                dist = abs(x1 - x2) + abs(y1 - y2)
                adj[i].append([dist, j])
                adj[j].append([dist, i])

        # Part 2: Prim's algorithm
        res = 0
        visit = set()
        minH = [[0, 0]]  # [[cost, point]]

        while len(visit) < N:
            cost, i = heapq.heappop(minH)  # we always make sure to pop a leaf of the current tree using the min heap
            if i in visit:
                continue
            res += cost   # we accumulate the cost in res
            visit.add(i)  # we add the node with minimal distance
            for neiCost, nei in adj[i]:
                if nei not in visit:
                    heapq.heappush(minH, [neiCost, nei])  # push the unvisited neighbors in the min heap for the next iterations
        return res
```

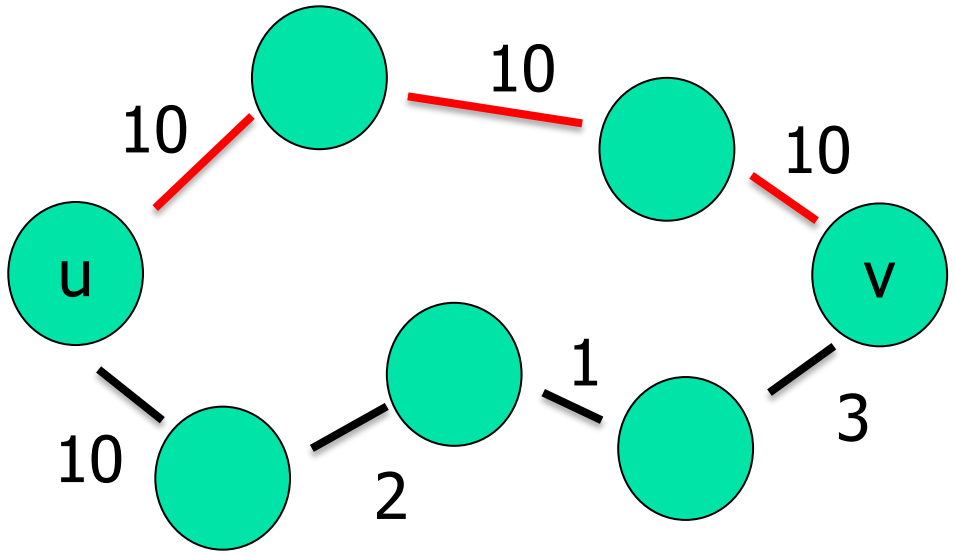
4. Shortest Path Problem

Shortest Path Problem

- Given a weighted graph G, and 2 nodes u and v
- What is the shortest path between u and v?
  - The shortest path is the one with the lowest cost
  - The cost can be seen as a distance between 2 nodes

Shortest Path Problem

- Given a weighted graph G, and 2 nodes u and v
- What is the shortest path between u and v?
  - The shortest path is the one with the lowest cost
  - The cost can be seen as a distance between 2 nodes



Top path: (4 nodes)
d1 = d(u, v) = 30

Bottom path: (5 nodes)
d2 = d(u, v) = 16

-> Shortest distance: d2

Dijkstra's algorithm

```
def dijkstra(graph, root):
    n = len(graph)
    # initialize distance list as all infinities
    dist = [Inf for _ in range(n)]
    # set the distance for the root to be 0
    dist[root] = 0
    # initialize list of visited nodes
    visited = [False for _ in range(n)]
    # loop through all the nodes
    for _ in range(n):
        # "start" our node as -1 (so we don't have a start node yet)
        u = -1
        # loop through all the nodes to check for visitation status
        for i in range(n):
            # if the node 'i' hasn't been visited and
            # we haven't processed it or the distance we have for it is less
            # than the distance we have to the "start" node
            if not visited[i] and (u == -1 or dist[i] < dist[u]):
                u = i
        # all the nodes have been visited or we can't reach this node
        if dist[u] == Inf:
            break
        # set the node as visited
        visited[u] = True
        # compare the distance to each node from the "start" node
        # to the distance we currently have on file for it
        for v, l in graph[u]:
            if dist[u] + l < dist[v]:
                dist[v] = dist[u] + l
    return dist
```

Dijkstra's algorithm

```python
def dijkstra(graph, root):
    n = len(graph)
    # initialize distance list as all infinities
    dist = [Inf for _ in range(n)]
    # set the distance for the root to be 0
    dist[root] = 0
    # initialize list of visited nodes
    visited = [False for _ in range(n)]
    # loop through all the nodes
    for _ in range(n):
        # "start" our node as -1 (so we don't have a start node yet)
        u = -1
        # loop through all the nodes to check for visitation status
        for i in range(n):
            # if the node 'i' hasn't been visited and
            # we haven't processed it or the distance we have for it is less
            # than the distance we have to the "start" node
            if not visited[i] and (u == -1 or dist[i] < dist[u]):
                u = i
        # all the nodes have been visited or we can't reach this node
        if dist[u] == Inf:
            break
        # set the node as visited
        visited[u] = True
        # compare the distance to each node from the "start" node
        # to the distance we currently have on file for it
        for v, l in graph[u]:
            if dist[u] + l < dist[v]:
                dist[v] = dist[u] + l
    return dist
```

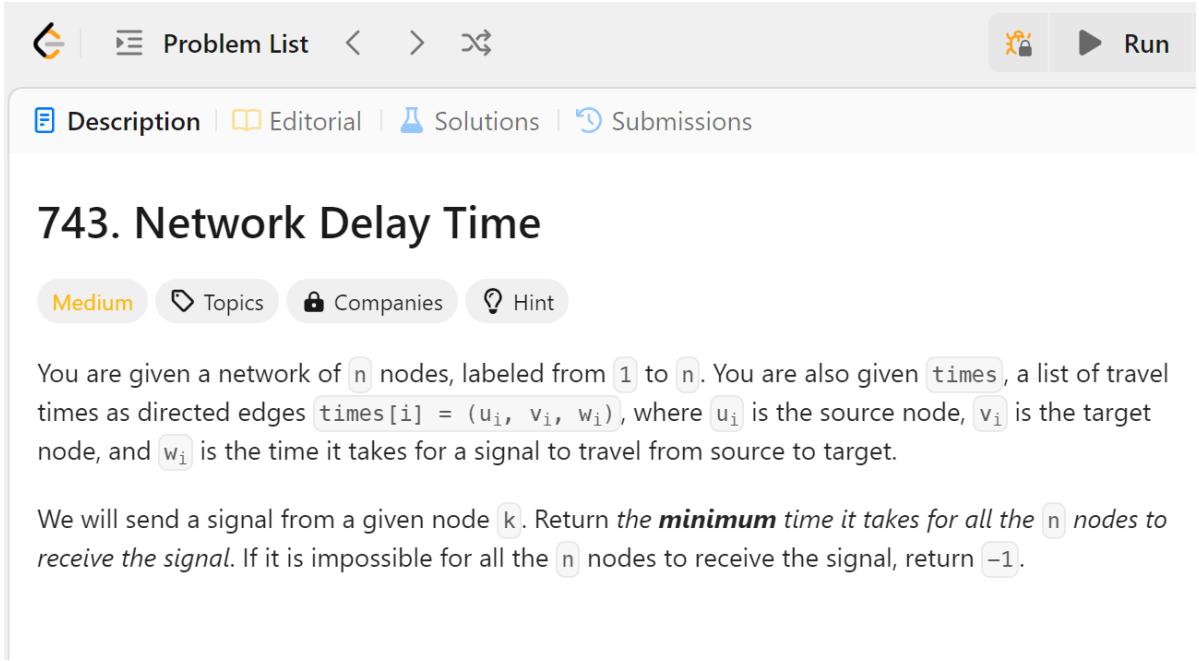
```python
from numpy import Inf

# adjacency list
# format: (node_name, distance)
graph = {
    0: [(1, 1)],
    1: [(0, 1), (2, 2), (3, 3)],
    2: [(1, 2), (3, 1), (4, 5)],
    3: [(1, 3), (2, 1), (4, 1)],
    4: [(2, 5), (3, 1)]
}
```

Output

```python
print(dijkstra(graph,1))
```

```
[1, 0, 2, 3, 4]
```

Shortest Path Problem



https://leetcode.com/problems/network-delay-time

Shortest Path Problem

Problem List | Run | Submit | Premium

Description | Editorial | Solutions | Submissions

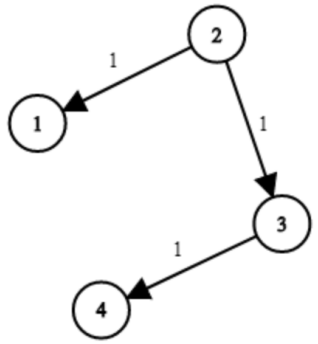
743. Network Delay Time

Medium | Topics | Companies | Hint

You are given a network of `n` nodes, labeled from `1` to `n`. You are also given `times`, a list of travel times as directed edges `times[i] = (u_i, v_i, w_i)`, where `u_i` is the source node, `v_i` is the target node, and `w_i` is the time it takes for a signal to travel from source to target.

We will send a signal from a given node `k`. Return *the* ***minimum*** *time it takes for all the* `n` *nodes to receive the signal.* If it is impossible for all the `n` nodes to receive the signal, return `-1`.

Example 1:




```
Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
Output: 2
```

7.2K | 37

Code

Python3 | Auto

```
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:

```

Saved to local | Ln 1, Col 1

Testcase | Test Result

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Shortest Path Problem

```python
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        edges = collections.defaultdict(list)

        # create the adjacency list
        for u, v, w in times:
            edges[u].append((v, w))

        minHeap = [(0, k)]
        visit = set()
        t = 0

        # Dijkstra: we keep poping the min heap until its empty
        # BFS
        # the min heap ensures we pop out the shortest path at every iteration
        while minHeap:
            w1, n1 = heapq.heappop(minHeap) # n1 is the node with shortest path
            if n1 in visit:
                continue  # continue if already visited
            visit.add(n1) # otherwise add the node
            t = max(t, w1)

            for n2, w2 in edges[n1]:
                if n2 not in visit:
                    heapq.heappush(minHeap, (w1 + w2, n2))  # add the total path w1+ w2 that it takes from n1 to n2

        return t if len(visit) == n else -1
```