1) **Warshall’s & Floyd’s Algorithm**

def floyd\_warshall(n, edges, distanceThreshold):

dist = [[float('inf')] \* n for \_ in range(n)]

for i in range(n):

dist[i][i] = 0

for u, v, w in edges:

dist[u][v] = min(dist[u][v], w)

dist[v][u] = min(dist[v][u], w)

for k in range(n):

for i in range(n):

for j in range(n):

if dist[i][j] > dist[i][k] + dist[k][j]:

dist[i][j] = dist[i][k] + dist[k][j]

result\_city = -1

min\_reachable = float('inf')

for i in range(n):

reachable\_count = sum(1 for j in range(n) if dist[i][j] <= distanceThreshold)

if reachable\_count < min\_reachable or (reachable\_count == min\_reachable and i > result\_city):

min\_reachable = reachable\_count

result\_city = i

return result\_city

print(floyd\_warshall(4, [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], 4))

print(floyd\_warshall(5, [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], 2))

Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4

Output: 3

Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distance Threshold = 2

Output: 0

2) **Warshall’s & Floyd’s Algorithm**

def networkDelayTime(times, n, k):

dist = [[float('inf')] \* (n + 1) for \_ in range(n + 1)]

for i in range(1, n + 1):

dist[i][i] = 0

for u, v, w in times:

dist[u][v] = min(dist[u][v], w)

for mid in range(1, n + 1):

for start in range(1, n + 1):

for end in range(1, n + 1):

if dist[start][mid] + dist[mid][end] < dist[start][end]:

dist[start][end] = dist[start][mid] + dist[mid][end]

max\_time = 0

for i in range(1, n + 1):

if dist[k][i] == float('inf'):

return -1

max\_time = max(max\_time, dist[k][i])

return max\_time

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

Output: 2

Input: times = [[1,2,1]], n = 2, k = 1

Output: 1

Input: times = [[1,2,1]], n = 2, k = 2

Output: -1

3)**Bellman-Ford Algorithm**

def catMouseGame(graph):

from collections import defaultdict

n = len(graph)

memo = {}

def dfs(mouse, cat, turn):

if mouse == 0: return 1

if cat == mouse: return 2

if (mouse, cat, turn) in memo:

return memo[(mouse, cat, turn)]

if turn == 0:

for next\_mouse in graph[mouse]:

if dfs(next\_mouse, cat, 1) == 1:

memo[(mouse, cat, turn)] = 1

return 1

memo[(mouse, cat, turn)] = 0

return 0

else:

for next\_cat in graph[cat]:

if next\_cat != 0 and dfs(mouse, next\_cat, 0) == 2:

memo[(mouse, cat, turn)] = 2

return 2

memo[(mouse, cat, turn)] = 0

return 0

return dfs(1, 2, 0)

print(catMouseGame([[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]))

print(catMouseGame([[1,3],[0],[3],[0,2]]))

Input: graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

Output: 0

Input: graph = [[1,3],[0],[3],[0,2]]

Output: 1

4) **Bellman-Ford Algorithm**

from collections import defaultdict

import heapq

def maxProbability(n, edges, succProb, start, end):

graph = defaultdict(list)

for (a, b), prob in zip(edges, succProb):

graph[a].append((b, prob))

graph[b].append((a, prob))

max\_heap = [(-1.0, start)]

visited = set()

while max\_heap:

prob, node = heapq.heappop(max\_heap)

prob = -prob

if node in visited:

continue

visited.add(node)

if node == end:

return prob

for neighbor, edge\_prob in graph[node]:

if neighbor not in visited:

heapq.heappush(max\_heap, (-(prob \* edge\_prob), neighbor))

return 0.0

print(maxProbability(3, [[0,1],[1,2],[0,2]], [0.5,0.5,0.2], 0, 2))

Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2

Output: 0.25000

Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start = 0, end = 2

Output: 0.30000

Input: n = 3, edges = [[0,1]], succProb = [0.5], start = 0, end = 2

Output: 0.00000

5)**Sudoku Solver**

def solve\_sudoku(board):

def is\_valid(num, row, col):

for i in range(9):

if board[row][i] == num or board[i][col] == num:

return False

start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)

for i in range(start\_row, start\_row + 3):

for j in range(start\_col, start\_col + 3):

if board[i][j] == num:

return False

return True

def backtrack():

for i in range(9):

for j in range(9):

if board[i][j] == '.':

for num in map(str, range(1, 10)):

if is\_valid(num, i, j):

board[i][j] = num

if backtrack():

return True

board[i][j] = '.'

return False

return True

backtrack()

return board

board = [["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]]

solved\_board = solve\_sudoku(board)

print(solved\_board)

Input: board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]