1. **Problem Statement:**

In this lab, Terry the rat is stuck inside a big underground pipe system. The pipes connect different junctions, and each pipe has a cost (which means how much time or difficulty it takes for Terry to crawl through). Our task is to help Terry move from his starting junction to the target junction where the cheese is hidden.

1. **How I Understood the problem:**

* Each junction is like a **state (node)**.
* Each pipe is like a **connection (edge)** that has some cost.
* Terry’s goal is to find a path to the cheese.
* The start is the first junction, and the goal state is the cheese junction.

So basically, this looks like a **graph problem**.

1. **Plan and Approach to the solution:**

I have considered the whole underground system as a graph with nodes and edges. Then I will use some search algorithms that we have learned to find paths.

**Inputs to be used:**

* Number of junctions and pipes
* Connections between junctions with travel costs
* Coordinates of junctions (for distance calculation)
* Start junction and target junction

**Outputs to get:**

* The path that Terry takes for each search strategy
* Total cost of the path
* How many junctions Terry visited

1. **Algorithms to be used (I have planned to use):**

**1)Depth First Search (DFS):**

* + Terry goes deep into one path first before backtracking.
  + Not always the cheapest path, but simple to implement.

**2)Breadth First Search (BFS):**

* + Terry checks all nearby junctions first.
  + Finds the path with the least number of steps, but doesn’t care about cost.

**3)A\*:**

* A\* is a graph search algorithm that efficiently finds the lowest-cost path from a starting point to a goal by combining actual travel cost with a heuristic (estimate) of the remaining distance.
* It explores paths based on their total estimated cost, making it faster and more optimal

**Reasons for my Approach:**

These algorithms are the basic search techniques which we are studying/ studied. By applying them, I will compare how Terry’s path changes with each method.

* DFS is fast to go deep, but not always efficient.
* BFS ensures shortest steps.
* A\* is efficient and smart because of the heuristic.

1. **Summary:**

I am representing the pipe network as a graph and will try to apply DFS, BFS, UCS, and A\* search. Each of them gives different paths and costs. This way I can see which method is better in different situations.

**#APPENDIX(CODING PART):**

from collections import deque

import heapq

import math

def build\_graph(num\_junctions, pipes):

"""

Build an adjacency list representing the pipe network.

Each junction is mapped to a list of (neighbor, travel\_cost) tuples.

"""

graph = {i: [] for i in range(num\_junctions)}

for node1, node2, cost in pipes:

graph[node1].append((node2, cost))

graph[node2].append((node1, cost)) # Since pipes are undirected

return graph

def bfs(num\_junctions, pipes, start, goal):

"""

Breadth-First Search: finds the shortest path by steps (not by lowest cost).

Returns the path followed and all visited junctions.

"""

graph = build\_graph(num\_junctions, pipes)

queue = deque([(start, [start])])

visited = set([start])

all\_visited = [start]

while queue:

current, path = queue.popleft()

if current == goal:

return path, all\_visited

for neighbor, \_ in graph[current]:

if neighbor not in visited:

visited.add(neighbor)

all\_visited.append(neighbor)

queue.append((neighbor, path + [neighbor]))

return None, all\_visited

def dfs(num\_junctions, pipes, start, goal):

"""

Depth-First Search: explores as deeply as possible. Not always optimal.

Returns the path followed and all visited junctions.

"""

graph = build\_graph(num\_junctions, pipes)

stack = [(start, [start])]

visited = set([start])

all\_visited = [start]

while stack:

current, path = stack.pop()

if current == goal:

return path, all\_visited

for neighbor, \_ in graph[current]:

if neighbor not in visited:

visited.add(neighbor)

all\_visited.append(neighbor)

stack.append((neighbor, path + [neighbor]))

return None, all\_visited

def heuristic(a, b, coords):

"""

Straight-line (Euclidean) distance between two junctions for A\*.

"""

x1, y1 = coords[a]

x2, y2 = coords[b]

return math.sqrt((x1 - x2)\*\*2 + (y1 - y2)\*\*2)

def a\_star(num\_junctions, pipes, coords, start, goal):

"""

A\* Search: finds the lowest-cost path using cost and a heuristic.

Returns the path followed and all visited junctions.

"""

graph = build\_graph(num\_junctions, pipes)

open\_set = []

heapq.heappush(open\_set, (heuristic(start, goal, coords), 0, start, [start]))

best\_cost = {start: 0}

all\_visited = [start]

visited\_node\_set = set([start])

while open\_set:

f, cost\_so\_far, current, path = heapq.heappop(open\_set)

if current == goal:

return path, all\_visited

for neighbor, cost in graph[current]:

new\_cost = cost\_so\_far + cost

if neighbor not in best\_cost or new\_cost < best\_cost[neighbor]:

best\_cost[neighbor] = new\_cost

estimated\_total = new\_cost + heuristic(neighbor, goal, coords)

heapq.heappush(open\_set, (estimated\_total, new\_cost, neighbor, path + [neighbor]))

if neighbor not in visited\_node\_set:

all\_visited.append(neighbor)

visited\_node\_set.add(neighbor)

return None, all\_visited

def total\_cost(path, pipes):

"""

Calculate the total travel cost in the path.

"""

if path is None or len(path) < 2:

return 0

# (u,v) sorted for undirected edge matching

edges = {(min(u, v), max(u, v)): cost for u, v, cost in pipes}

cost = 0

for i in range(len(path) - 1):

u, v = path[i], path[i+1]

key = (min(u, v), max(u, v))

cost += edges.get(key, 0)

return cost

# ------ Example Network (replace with own input data as needed) ------

num\_junctions = 5

pipes = [

(0, 1, 3),

(1, 2, 5),

(2, 3, 2),

(1, 3, 8),

(3, 4, 1)

]

coords = [ # Example coordinates for A\*

(0, 0), # Junction 0

(1, 1), # Junction 1

(2, 2), # Junction 2

(3, 1), # Junction 3

(4, 0) # Junction 4

]

start = 0

goal = 4

# ------------ Run search algorithms and print results ---------------------

bfs\_path, bfs\_visited = bfs(num\_junctions, pipes, start, goal)

dfs\_path, dfs\_visited = dfs(num\_junctions, pipes, start, goal)

a\_star\_path, a\_star\_visited = a\_star(num\_junctions, pipes, coords, start, goal)

print("BFS strategy:")

print(" - Path:", bfs\_path)

print(" - Total cost:", total\_cost(bfs\_path, pipes))

print(" - Junctions visited (in order):", bfs\_visited)

print(" - Number of junctions visited:", len(bfs\_visited))

print()

print("DFS strategy:")

print(" - Path:", dfs\_path)

print(" - Total cost:", total\_cost(dfs\_path, pipes))

print(" - Junctions visited (in order):", dfs\_visited)

print(" - Number of junctions visited:", len(dfs\_visited))

print()

print("A\* strategy:")

print(" - Path:", a\_star\_path)

print(" - Total cost:", total\_cost(a\_star\_path, pipes))

print(" - Junctions visited (in order):", a\_star\_visited)

print(" - Number of junctions visited:", len(a\_star\_visited))

print()