Rat Escape: Finding the Shortest Path for the Rat

1. Problem Statement

A rat is trapped inside a system of pipes connected through multiple junctions. The rat needs to reach the cheese placed at another junction using the shortest possible path. Each pipe has a specific distance (or cost). The task is to determine the most efficient route from the starting junction to the goal (cheese location).

2. Understanding the Problem

This problem is a graph traversal and pathfinding challenge:

- **Junctions** → nodes of a graph.
- **Pipes** → edges connecting nodes with weights (distance).
- **Start** → rat's position.
- **Goal** → cheese location.
- **Objective** → find the path with the minimum total cost.

3. Approach

1. **Graph Representation**: Nodes = junctions, Edges = pipes with costs.
2. **Input Required**: number of junctions and pipes, connections with costs, start and goal junction, optional coordinates for heuristic in A*.
3. **Algorithms Implemented**:
4. DFS (Depth-First Search)
5. BFS (Breadth-First Search)
6. UCS (Uniform Cost Search)
7. A* (with heuristic)

4. Python Implementation

```python
import heapq
import math
from collections import deque

# DFS
def dfs(graph, start, goal, path=None, visited=None):
    if path is None: path = []
    if visited is None: visited = set()
    path.append(start)
    visited.add(start)
    if start == goal: return path
    for neighbor in graph.get(start, {}):
        if neighbor not in visited:
            result = dfs(graph, neighbor, goal, path.copy(), visited.copy())
            if result: return result
    return None

# BFS
def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()
    while queue:
        node, path = queue.popleft()
        if node == goal: return path
        if node in visited: continue
        visited.add(node)
        for neighbor in graph.get(node, {}):
            if neighbor not in visited: queue.append((neighbor, path +
[neighbor]))
    return None

# UCS
def ucs(graph, start, goal):
    queue = [(0, start, [])]
    visited = set()
    cost_so_far = {start: 0}
    while queue:
        queue.sort()
        cost, node, path = queue.pop(0)
        if node == goal: return path + [node], cost
        if node in visited: continue
        visited.add(node)
        for neighbor, edge_cost in graph.get(node, {}).items():
            new_cost = cost + edge_cost
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                queue.append((new_cost, neighbor, path + [node]))
    return None, None

# A*
def a_star(graph, coords, start, goal):
    if start == goal: return [start], 0
    def heuristic(node):
        if coords is None: return 0.0
        x1, y1 = coords.get(node, (0.0,0.0))
        x2, y2 = coords.get(goal, (0.0,0.0))
        return math.hypot(x1-x2, y1-y2)
    queue = [(0.0, start, [])]
    cost_so_far = {start: 0.0}
    visited = set()
    while queue:
        priority, node, path = heapq.heappop(queue)
        if node == goal: return path + [node], cost_so_far[node]
        if node in visited: continue
        visited.add(node)
        for neighbor, cost in graph.get(node, {}).items():
            new_cost = cost_so_far[node] + float(cost)
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                heapq.heappush(queue, (new_cost + heuristic(neighbor), neighbor,
path + [node]))
    return None, None

# Main Program
def main():
    num_junctions = int(input("Enter the number of pipe junctions: ").strip())
    num_pipes = int(input("Enter the number of pipes: ").strip())
    coords = {}
    if input("Provide coordinates for junctions? (y/n): ").strip().lower() ==
'y':
        for _ in range(num_junctions):
            node, x, y = input("node x y: ").split()
            coords[node] = (float(x), float(y))
    else: coords = None
    graph = {}
    print("Enter pipe connections (junction1 junction2 cost):")
    for _ in range(num_pipes):
        u, v, cost = input().split()
        cost = float(cost)
        graph.setdefault(u, {})[v] = cost
        graph.setdefault(v, {})[u] = cost
    start = input("Enter starting junction: ").strip()
    goal = input("Enter cheese junction: ").strip()

    print("\n--- DFS ---")
    path_dfs = dfs(graph, start, goal)
    if path_dfs: print("Path:", " -> ".join(path_dfs), "\nVisited junctions:",
len(path_dfs))
    else: print("No path found")

    print("\n--- BFS ---")
    path_bfs = bfs(graph, start, goal)
    if path_bfs:
        cost_bfs = sum(graph[path_bfs[i]][path_bfs[i+1]] for i in
range(len(path_bfs)-1))
        print("Path:", " -> ".join(path_bfs), "\nTotal distance:", cost_bfs,
"\nVisited junctions:", len(path_bfs))
    else: print("No path found")
    print("\n--- UCS ---")
    path_ucs, cost_ucs = ucs(graph, start, goal)
    if path_ucs: print("Path:", " -> ".join(path_ucs), "\nTotal distance:",
cost_ucs, "\nVisited junctions:", len(path_ucs))
    else: print("No path found")

    print("\n--- A* ---")
    path_astar, cost_astar = a_star(graph, coords, start, goal)
    if path_astar: print("Path:", " -> ".join(path_astar), "\nTotal distance:",
cost_astar, "\nVisited junctions:", len(path_astar))
    else: print("No path found")

if __name__ == "__main__":
    main()
```

5. Sample Input & Output

Input:

```
Enter the number of pipe junctions: 4
Enter the number of pipes: 4
Provide coordinates? n
Enter pipe connections:
A B 1
B C 2
A C 4
C D 1
Enter starting junction: A
Enter cheese junction: D
```

Output:

```
--- DFS ---
Path: A -> B -> C -> D
Visited junctions: 4

--- BFS ---
Path: A -> B -> C -> D
Total distance: 4.0
Visited junctions: 4

--- UCS ---
Path: A -> B -> C -> D
Total distance: 4.0
```

```
Visited junctions: 4

--- A* ---
Path: A -> B -> C -> D
Total distance: 4.0
Visited junctions: 4
```

6. Conclusion

- DFS explores deeply but may not find optimal paths in weighted graphs.
- BFS guarantees shortest path only for equal-cost edges.
- UCS always finds optimal path in weighted graphs.
- A* efficiently finds optimal path using heuristic information.

This simulation demonstrates intelligent navigation for real-world pathfinding through pipe networks.