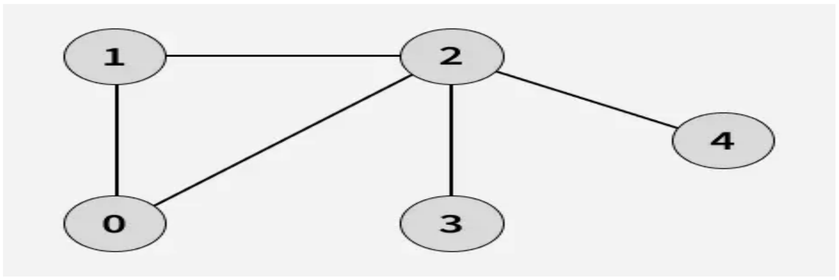
BFS

Breadth First Search (BFS) is a technique used to explore a graph starting from a chosen node. It checks the graph in layers: first, all neighbors of the starting node are visited. After that, it visits the neighbors of those nodes, continuing this process step-by-step. This goes on until there are no more nodes left that can be reached from the starting point.

Example

Graph example adapted from GeeksforGeeks



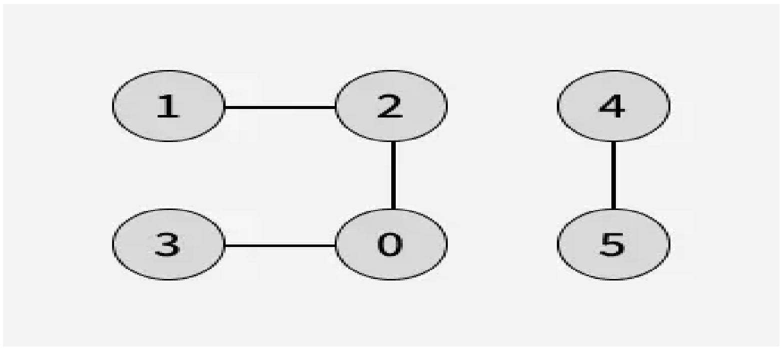
Input: adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]
Output: [0, 1, 2, 3, 4]

Explanation
Starting from node 0, BFS explores nodes in expanding layers:

- Begin at 0 → output 0
- Move to its neighbors: 1 → output 1, then 2 → output 2
- From 2, we discover unvisited nodes: 3 → output 3 and then 4 → output 4

All reachable vertices from the starting point are now visited.

Example 2



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

Explanation
We begin the BFS at node 0:

- Visit 0 → output 0
- Check neighbors of 0: we reach 2 → output 2, then 3 → output 3
- From 2, another unvisited node 1 is found → output 1

Since the earlier BFS only covered part of the graph (it was disconnected),
we launch another BFS with node 4 as the new starting point:

- Visit 4 → output 4
- Go to its neighbor 5 → output 5

Pseudocode — Breadth-First Search

```
BFS(Graph G, start_node):
  for each vertex x in G:
    visited[x] = false
    level[x] = infinity
    predecessor[x] = null

  create a queue Q
  visited[start_node] = true
  level[start_node] = 0
  enqueue start_node into Q

  while Q is not empty:
    current = dequeue Q

    for each adjacent node n of current:
      if visited[n] is false:
        visited[n] = true
        level[n] = level[current] + 1
        predecessor[n] = current
        enqueue n into Q
```

Time Complexity

O(V + E) — This represents linear time.
Thanks to the visited[] mechanism, each vertex and each edge in the graph is examined only once.

Even though BFS may start from different nodes in disconnected components, no element is processed more than a single time.

Auxiliary Space
O(V) — Primarily due to the queue used to store nodes awaiting exploration, along with the visited and other auxiliary arrays.

Applications of BFS in Graphs

BFS is widely used in computer science and graph-based problems, including:

- Finding the shortest path (in an unweighted graph)
- Detecting cycles
- Identifying connected components
- Network and packet routing algorithms

Implementation

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/BFS%20Problems/BFS%20Basic/BFS.cpp

Problem 1 : *Bicoloring*

Problem:

Bicoloring (BFS) — Bicoloring problem in Onlinejudge

Link:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=945

Can a Graph Be Colored Using Only Two Colors?

Many of us have heard about the *Four Color Theorem*, which was proven with computer assistance in 1976. It states that any map can be colored with four colors such that no two neighboring regions share the same color.

Now, let's look at a related — but much simpler — problem. We want to determine whether a graph can be colored using just two colors (for example, white and black), ensuring that no two connected vertices have the same color.
This challenge is known as graph bicoloring.

Bicoloring and Bipartite Graphs

A graph that can be successfully colored with two colors is called a **bipartite graph**.

A graph is bipartite if its vertices can be split into two separate sets (let's call them Group A and Group B) such that:

1. Every edge connects a vertex from Group A to a vertex from Group B.
2. No vertices inside Group A are directly connected.
3. No vertices inside Group B are directly connected.

If we color all vertices in Group A white, and all in Group B black, the graph satisfies the bicoloring requirement.

When Bicoloring Fails

There is a key reason why a graph cannot be 2-colored:

Condition: If the graph contains a cycle of odd length, it cannot be bicolored.

Example:
Nodes (0, 1, 2) forming a triangle: edges are 0-1, 1-2, and 2-0.
This is a cycle of length 3 → which is odd.

- Suppose we color node 0 as white.
- Then node 1 must be black (since it touches 0).
- Next, node 2 must be white (because it touches 1).
- But node 2 is also connected to node 0 — both are white — conflict!

Therefore, graphs containing triangles (3-cycles), 5-cycles, or any odd cycle cannot be bipartite.

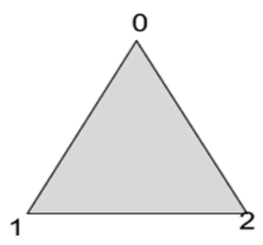
How to Test This — Using BFS

To check if a graph is bicolorable or not, we can use the Breadth-First Search (BFS) strategy.

Steps:

1. **Initialization:**
   All vertices are initially uncolored (marked −1).
   We use an array color[] to track each node's color.
   We may use color 0 for one group and color 1 for the other.
2. **Begin BFS:**
   Start from any vertex (say 0).
   Assign it a color, e.g. color[0] = 0 and put it into the queue.
3. **Process vertices:**
   While the queue has elements:
   - Take out the front node u.
   - Determine the opposite color nextColor = 1 − color[u].
   - Examine each neighbor v of u:
     - If v is uncolored, assign it nextColor and push it to the queue.
     - If v already has a color:
       - If its color is the same as u, we found a conflict → graph is NOT bicolorable.
       - If the color is correct, we continue.
4. **Conclusion:**
   If BFS finishes without any contradiction, the graph can indeed be colored with two colors → it is BICOLORABLE.

Example Input (Triangle)



Input:

3
3
0 1
1 2
2 0

Starting color array:

[-1, -1, -1]

Start BFS from node 0:

color = [0, -1, -1]
queue = [0]

Color neighbors of 0:

color = [0, 1, 1]

Now examine node 1:

- Check neighbor 0 → valid
- Check neighbor 2 → both are color 1 → conflict found

Therefore, the graph cannot be 2-colored.

Solution Code:

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/BFS%20Problems/Bicoloring/Bicoloring.cpp

Problem 2:

Knight Moves Problem: Knight moves (BFS in 2d Grid) — Knight moves problem in Onlinejudge

Link:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1390

Knight's Shortest Path — BFS Approach

Problem Overview

We are examining a classical shortest-move challenge involving a knight on a chessboard:
Given a starting square *a* and a target square *b*, determine the minimum number of knight moves required to reach *b* from *a*.
To model this, we view the chessboard as an unweighted graph:

- **Vertices (Nodes):** all 64 squares
- **Edges:** legal knight moves between squares

Since each valid move costs exactly one unit, **Breadth-First Search (BFS) is the optimal method** for finding the shortest route.

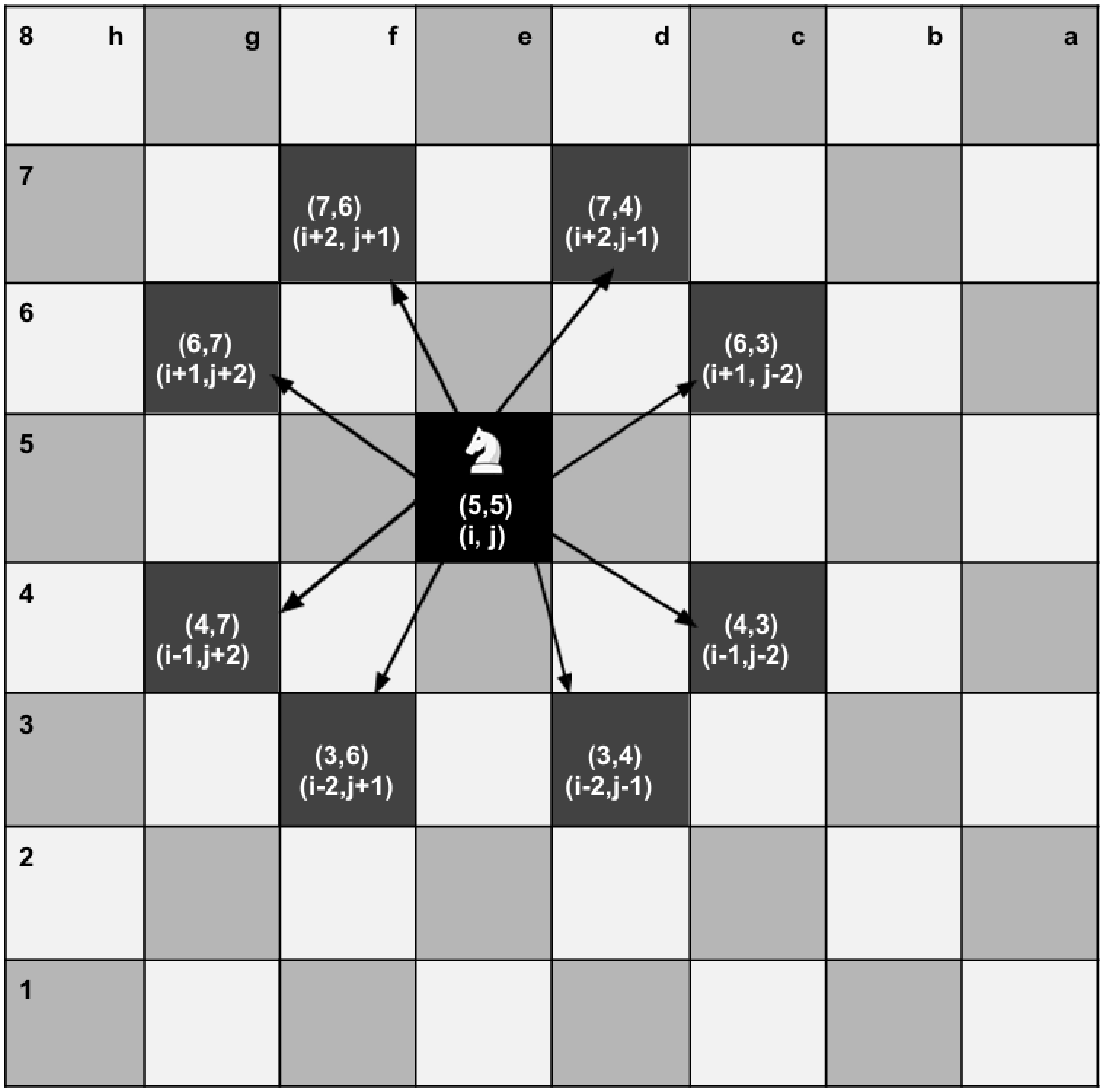
Why BFS Works Here

BFS explores the board step-wise by increasing distance:

- Level 0 → starting cell
- Level 1 → cells reachable in exactly 1 move
- Level 2 → positions reachable in 2 moves
- Level 3 → those reachable in 3 moves
- etc.

Because BFS visits positions in order of increasing distance, the **moment we first reach the destination square**, we can confidently conclude the number of moves is minimal.

Knight Movement on the Board



If the knight is at coordinate (i, j), it may move to the following potential positions:

- (i-2, j+1)
- (i-2, j-1)
- (i-1, j+2)
- (i-1, j-2)
- (i+1, j+2)
- (i+1, j-2)
- (i+2, j+1)
- (i+2, j-1)

To make implementation easier, these are often stored as directional arrays:

```
const int kr[] = {2, 2, -2, -2, 1, 1, -1, -1};
const int kc[] = {1, -1, 1, -1, 2, -2, 2, -2};
```

For each k from 0 to 7:
new_r = r + kr[k]
new_c = c + kc[k]
This eliminates repetitive code and organizes the movement logic cleanly.

Data Structures Used in the BFS

Our BFS solution typically maintains three key components:

1. dist[n][n]
   Stores the shortest known distance from the starting square to every other square.
   Initially every value is set to infinity (INT_MAX).
2. color[n][n]
   Tracks the state of each square during traversal:
    - -1 → unvisited
    - 1 → discovered and currently in queue
    - 2 → fully processed
3. queue<pair<int,int>> q
   Holds the next squares to be processed.

Step-by-Step Example — From a1 to h8
Coordinates:

- a1 → (0, 0)
- h8 → (7, 7)

Step 1 — Initialization

- All distances set to infinity
- All squares marked unvisited (-1)
- Starting point:
- dist[0][0] = 0
- color[0][0] = 1
- Q = [(0, 0)]

Step 2 — Processing distance d = 0

Current node: (0, 0) → distance = 0
Possible reachable squares:

- (2, 1) → b3
- (1, 2) → c2

Update:

dist[2][1] = 1
dist[1][2] = 1

Queue becomes:
Q = [(2,1), (1,2)]
Then mark (0,0) as processed:
color[0][0] = 2

Step 3 — Processing nodes at distance d = 1

Process (2,1) and (1,2).
Each of these squares will produce multiple valid new positions.

New squares discovered at this level get distance:

dist = 2

They are placed into the queue in preparation for the next layer.

Step 4 — Continuing the BFS

This process continues:

- Every square at distance d gets processed
- Their valid knight moves are assigned distance d+1
- Already-visited squares are not re-enqueued

Eventually, one of the computed moves will land on:

(7, 7)

which corresponds to h8.

At that moment:

- color[7][7] will still be −1
- dist[7][7] = dist[r][c] + 1

Thus, the first recorded distance for (7,7) is guaranteed to be minimal.

Final Result

The shortest number of knight moves required to travel from **a1** to **h8** is:
6
Because BFS expands outward by increasing move-count, this is unquestionably the optimal number of moves.

Solution Code

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/BFS%20Problems/Knight%20Moves/Knight%20Moves.cpp

Problem 3: Risk

Problem: Risk (Shortest Path) — Risk problem in Onlinejudge

Link:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&category=0&problem=508&mosmsg=Submission+received+with+ID+30835231

Solving the "Risk" Problem

Imagine you are playing Risk and trying to move armies across territories. Strategy matters — and finding the quickest route across the map is essential. In this task, we have a world of 20 countries with defined borders, and the objective is to determine the minimum number of countries you must pass through to get from one specific country to another.

1. Problem Breakdown

The idea of "minimum countries to conquer" basically translates into a shortest-path problem in graph terminology:

- Each country acts as a node (1–20)
- Each border forms an edge between countries
- Every movement between neighbors has cost = 1

So, our mission is to compute the shortest path between two nodes in an unweighted graph.

2. Why BFS is Ideal

The graph does not contain different weights—every border crossing costs the same effort. Because of that, BFS is the best choice.

BFS explores a graph in breadth:

- Level 0 → starting country
- Level 1 → all bordering countries
- Level 2 → neighbors of those neighbors

As soon as BFS reaches the target country, that path is guaranteed to be the shortest one possible.

3. Graph Input Format

The input provides border data for countries 1 to 19. For each country i, only neighbors j > i are listed.

This avoids duplication.
For example:
If we are told 1 is adjacent to 2, we don't need another line saying 2 is adjacent to 1.

But while storing the graph, we must make it **undirected**, meaning:

if 1 touches 2, then 2 touches 1.

So, we use:

- vector<int> adj[21] for adjacency lists
- Each edge is inserted twice: u→v and v→u

4. BFS Implementation

The central BFS logic used in the solution is shown below:

```
int bfs(int start, int end) {
   vector<int> dist(21, -1);
   queue<int> q;

   q.push(start);
   dist[start] = 0;

   while (!q.empty()) {
      int u = q.front();
      q.pop();

      if (u == end) return dist[u];

      for (int v : adj[u]) {
         if (dist[v] == -1) {
            dist[v] = dist[u] + 1;
            q.push(v);
         }
      }
   }
   return -1;
}
```

Here:

- dist[] keeps steps taken
- -1 indicates unvisited
- Once the destination is found — we return the distance immediately

5. Output Alignment Requirements
The output format requires specific spacing rules:

- Source and destination must be right-aligned in 2-character width fields
- The answer must begin at column 11

This is easily achieved using C-style formatted printing:

```
printf("%2d to %2d: %d\n", start, end, result);
```

- %2d → prints numbers in a 2-space slot
  (so 1 and 20 align correctly)

Example:

```
 1 to 20: 6
```

Everything lines up neatly.

6. Complexity Discussion

- **Time Complexity:** O(V + E) for each BFS call
  Since V=20, the computation is extremely fast.
- **Space Complexity:** O(V) for storing visited distances and the graph itself

Conclusion

This challenge is an example of finding the Shortest Path in an unweighted graph. While algorithms like Dijkstra or Floyd-Warshall also compute shortest paths, BFS is perfectly suited here due to equal edge weights and minimal overhead.

Solution Code

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/BFS%20Problems/Risks/Risk.cpp

Bellman-Ford

Bellman–Ford Algorithm Overview

The Bellman–Ford algorithm is used to determine the shortest path from a single source to all vertices in a weighted graph. Unlike Dijkstra's algorithm, it can handle negative edge weights and can even detect negative cycles, though it cannot correctly compute shortest paths if such cycles exist.

Key Characteristics:

- Works on weighted graphs (including negative weights)
- Can detect negative cycles
- Suitable for graphs that may not be fully connected
- Simpler to implement with a dynamic programming–style approach

When to Use Bellman–Ford:
Use this algorithm if:

- Your graph has edges with negative weights
- You need to check for negative weight cycles
- The graph might be disconnected
- You prefer a straightforward iterative approach

Core Concept: Relaxation

The algorithm relies on the **relaxation principle**:
If the shortest known distance to a vertex v is longer than going through u, update it.
Mathematically:

```
if dist[v] > dist[u] + w(u, v):
   dist[v] = dist[u] + w(u, v)
```

Relaxing an edge improves the distance estimate if a shorter path is found.

How It Works
For a graph with V vertices and E edges:

1. **Initialization:**
    - Set dist[source] = 0
    - Set dist[all other vertices] = ∞
2. **Edge Relaxation (V – 1 times):**
    - Repeat V–1 iterations (since the longest possible shortest path has at most V–1 edges)
    - For each edge (u → v) with weight w, apply relaxation
3. **Negative Cycle Check:**
    - Perform one extra relaxation pass
    - If any distance improves, a **negative cycle** exists

Pseudocode

```
function BellmanFord(graph, source):
    for each vertex v in graph:
        distance[v] = ∞
        previous[v] = NULL
    distance[source] = 0

    for i = 1 to |V| - 1:
        for each edge (u, v) in graph:
            if distance[u] + weight(u, v) < distance[v]:
                distance[v] = distance[u] + weight(u, v)
                previous[v] = u

    for each edge (u, v) in graph:
        if distance[u] + weight(u, v) < distance[v]:
            report "Negative cycle detected"

    return distance[], previous[]
```

Time Complexity

- **O(V × E)** — loops through all edges for every vertex.

Implementation

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/Bellman-Ford%20Problems/Bellman-Ford%20Basic/bellman%20ford%20solve.cpp

Problem 1: Wormholes

Problem: UVA558 — Wormholes problem in Onlinejudge
Link:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=499

Exploring the Cosmos: Are Wormholes Gateways to Time Travel?

Imagine it's the year 2163. Humanity has discovered wormholes—tunnels that connect distant star systems. But these aren't ordinary tunnels; they distort both space and time. Some wormholes might propel you 15 years into the future, while others could throw you 42 years into the past!

Our physicist dreams of witnessing the Big Bang. To travel endlessly into the past, she must locate a specific wormhole loop**.** Traversing this cycle repeatedly could allow her to move backward in time indefinitely. The question arises**:** Does such a loop exist? We can answer this using algorithms and a C++ program.

Converting the Problem into a Graph

We can model the universe as a graph:

- **Nodes (Vertices)** = Star systems. Earth is Node 0**.**
- **Edges** = One-way wormholes connecting these star systems.
- **Weights** = Time differences when using a wormhole:
    - Forward in time (e.g., +1000 years) → Positive weight
    - Backward in time (e.g., -42 years) → Negative weight

Finding a way to travel endlessly into the past is equivalent to finding a negative weight cycle in the graph (a loop where the sum of edge weights is negative).

Bellman-Ford: Detecting the Time Loop

The Bellman-Ford algorithm is perfect for this scenario:

- It works with graphs containing negative weights.
- Most importantly, it can detect negative cycles, which is exactly what our physicist needs.

Why Not Dijkstra?

Dijkstra's algorithm is faster for positive weights but fails when negative edges exist. Bellman-Ford handles negative weights and confirms if a negative cycle is present.

How Bellman-Ford Works

Assume there are N star systems. Bellman-Ford proceeds in two main stages:

1. N-1 Relaxation Passes

- The algorithm iterates over all edges N-1 times.
- In each iteration, it checks if a shorter path to any node exists via the current edge and updates distances if needed.
- After N-1 iterations, all shortest paths stabilize if there is no negative cycle**.**

2. Final N-th Check

- After the N-1 passes, we perform one more pass.
- If any distance still decreases, it proves that a negative cycle exists**.**
- This indicates that the physicist can keep looping through the wormhole cycle to move further into the past.

Step-by-Step Example

Input:

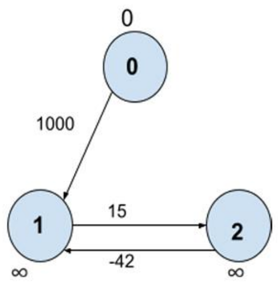
- Star systems: N **= 3** → 0 (Earth), 1, 2
- Wormholes: M **= 3**
    1. E1: 0 → 1, Time +1000
    2. E2: 1 → 2, Time +15
    3. E3: 2 → 1, Time -42

Observation: The path 1 → 2 → 1 forms **a** cycle with total time = 15 + (-42) = -27 years (negative cycle).

Initialization:

dist[0] = 0, dist[1] = INF, dist[2] = INF

- [0, INF, INF]

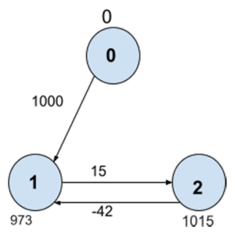


Pass 1

1. E1: 0 → 1, +1000 → 0 + 1000 < INF **→** dist[1] = 1000

2. E2: 1 → 2, +15 → 1000 + 15 < INF → dist[2] = 1015
3. E3: 2 → 1, -42 → 1015 - 42 < 1000 → dist[1] = 973

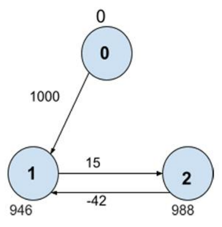
dist after Pass 1: [0, 973, 1015]



Pass 2

1. E1: 0 → 1, +1000 → 0 + 1000 > 973 → No change
2. E2: 1 → 2, +15 → 973 + 15 = 988 < 1015 → dist[2] = 988
3. E3: 2 → 1, -42 → 988 - 42 = 946 < 973 → dist[1] = 946

dist after Pass 2: [0, 946, 988]



Final Check (3rd Pass)

- E1: 0 → 1, +1000 → No change
- E2: 1 → 2, +15 → 946 + 15 = 961 < 988 → RELAXATION OCCURS!

Since a relaxation happened in the N-th pass, a negative cycle exists.

Conclusion

- **Negative cycle found → possible:** The physicist can travel backward indefinitely.
- **No negative cycle → not possible:** No infinite time travel loop exists.

Solution Code

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/Bellman-Ford%20Problems/UVA558_Warmholes/UVA558%20Wormholes.cpp

Problem 2: Traffic

Problem: 10449 — Traffic problem in Onlinejudge

Link:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1390

Dhaka Traffic Unraveled: An Algorithmic Journey

Dhaka is notorious for its traffic jams. To address congestion, the city planners in this problem (UVa 10449) devised an unusual mathematical toll system. Instead of fixed fees, the travel cost between junctions depends on the "busyness" difference cubed.

This simple rule leads to an interesting algorithmic puzzle: what if traveling a road gives you negative cost? What if the city effectively pays you to choose a certain route?

Here's how we tackled this using Graph Theory and the Bellman-Ford algorithm.

1. Representing the City as a Graph

We start by converting the city map into a Directed Graph:

1. Nodes (Junctions): Each junction numbered from 1 to N is treated as a vertex.
2. Edges (Roads): The roads are directed edges connecting these vertices.
3. Weights (Cubic Travel Cost): The twist is in the cost of each road:

Cost = $(\text{Busyness}[v] - \text{Busyness}[u])^3$

Why Cube the Difference

- Going uphill (Bv > Bu): positive difference → positive cost → you pay a toll.
- Going downhill (Bv < Bu): negative difference → negative cost → you "earn" or reduce total cost.

2. Choosing the Right Algorithm: Bellman-Ford

We want the minimum total cost from Junction 1 to all others. Normally, Dijkstra's algorithm would be the first choice, but it fails with negative edge weights because it assumes that once a node's distance is finalized, it cannot improve.

Bellman-Ford works perfectly here because it handles negative weights and repeatedly relaxes edges to find optimal paths.

3. The Danger of Negative Cycles

Unlike the "Wormhole" problem where negative cycles were useful, here a negative cycle is a problem. A loop of roads with negative total cost (e.g., -10) allows a driver to keep cycling endlessly, lowering their total cost infinitely: -10, -20, -30 … → $-\infty$

Detecting & Marking Affected Nodes

1. Relax edges N-1 times to find paths ignoring cycles.
2. Nth relaxation: any edge that can still reduce distance identifies nodes in a negative cycle.
3. BFS propagation: nodes reachable from a "bad" node are also considered part of a negative cycle and marked, so we can ignore them later.

4. Filtering Small Profits

The city ignores paths with total cost less than 3. When answering queries:

- Output ? if the destination is unreachable (distance = Infinity).
- Output ? if the destination is part of a negative cycle.
- Output ? if the cost < 3.

5. Stepwise Solution

Input Example

5 // n (Junctions)
6 7 8 9 10 // Busyness of Junctions 1 to 5
6 // r (Number of Roads)
1 2
2 3
3 4
1 5
5 4
4 5
2 // q (Number of Queries)
4 // Query: destination 4
5 // Query: destination 5

Step 1: Edge Weight Calculation

Weight formula: Weight = $(\text{Busyness}[v] - \text{Busyness}[u])^3$

Edge	**Busyness (u→v)**	**Calculation**	**Weight**
1→2	6→7	$(7-6)^3$	1
2→3	7→8	$(8-7)^3$	1
3→4	8→9	$(9-8)^3$	1
1→5	6→10	$(10-6)^3$	64
5→4	10→9	$(9-10)^3$	-1
4→5	9→10	$(10-9)^3$	1

Step 2: Initialize Distances

Source is Junction 1.

- Initial dist array: [INF, 0, INF, INF, INF, INF] (index 0 unused)
- dist[1] = 0, all others = INF

Step 3: Bellman-Ford Relaxations

Iteration 1

- Edge 1→2: 0+1 = 1 → update dist[2]=1
- Edge 1→5: 0+64 = 64 → update dist[5]=64
- Other edges cannot relax yet
  State: [INF, 0, 1, INF, INF, 64]

Iteration 2

- Edge 2→3: 1+1 = 2 → update dist[3]=2
- Edge 5→4: 64-1=63 → update dist[4]=63
  State: [INF, 0, 1, 2, 63, 64]

Iteration 3

- Edge 3→4: 2+1=3 < 63 → update dist[4]=3
- Edge 4→5: 3+1=4 < 64 → update dist[5]=4
  State: [INF, 0, 1, 2, 3, 4]

Iteration 4

- No updates occur, distances stable
  Final dist array: [INF, 0, 1, 2, 3, 4]

Step 4: Negative Cycle Detection

- Check 4→5→4 loop: weight 1 + (-1) = 0 → not negative
- No negative cycles detected
- inNegativeCycle array remains all 0

Step 5: Answer Queries
Query 4
dist[4]=3, reachable, not negative cycle, >=3 → output: 3
Query 5
dist[5]=4, reachable, not negative cycle, >=3 → output: 4

Solution Code

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/Bellman-Ford%20Problems/LOJ1108_Traffic/LOJ1108_Traffic.cpp

Dijkstra’s Algorithm

Dijkstra’s Shortest Path: A Historical and Practical Overview

The Dijkstra shortest path algorithm was developed in 1956 by Dutch computer scientist Edsger W. Dijkstra. Legend says he came up with it during a twenty-minute coffee break while shopping in Amsterdam with his fiancée. The algorithm was initially created to test a new computer called ARMAC.

How the Algorithm Works

1. Assign initial distances to all vertices: 0 for the source vertex, and infinity for all others.
2. Select the unvisited vertex with the smallest distance from the source as the current vertex. Initially, this is always the source vertex.
3. For each unvisited neighbor of the current vertex, compute the distance from the source. If this new distance is smaller than the previously recorded distance, update it.
4. After evaluating all neighbors, mark the current vertex as visited. A visited vertex will not be checked again.
5. Repeat steps 2–4 until all vertices have been visited.
6. Once finished, the algorithm provides the shortest path from the source vertex to every other vertex in the graph.

Example Input

Source: src = 0
Adjacency list:

```
adj[][] = [
   [[1, 4], [2, 8]],
   [[0, 4], [4, 6], [2, 3]],
   [[0, 8], [3, 2], [1, 3]],
   [[2, 2], [4, 10]],
   [[1, 6], [3, 10]]
]
```

Output: [0, 4, 7, 9, 10]

Explanation of Shortest Paths:

- 0 → 0 = 0: The source itself.
- 0 → 1 = 4: Direct edge from 0 to 1.
- 0 → 2 = 7: Path 0 → 1 → 2 gives 4 + 3 = 7, smaller than the direct edge 8.
- 0 → 3 = 9: Path 0 → 1 → 2 → 3 totals 4 + 3 + 2 = 9.
- 0 → 4 = 10: Path 0 → 1 → 4 totals 4 + 6 = 10.

Pseudocode

```
function dijkstra(G, S)
  for each vertex V in G
    distance[V] <- infinite
    previous[V] <- NULL
    if V != S
      add V to Priority Queue Q
  distance[S] <- 0
  while Q is not empty
    U <- ExtractMin(Q)
    for each unvisited neighbor V of U
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

  return distance[], previous[]
```

Complexity of Dijkstra's Algorithm

- **Time Complexity:** O(E log V), where E is the number of edges and V is the number of vertices.
- **Space Complexity:** O(V)

Applications of Dijkstra's Algorithm

- Calculating the shortest path between locations.
- Social networking analysis.
- Routing in telephone networks.
- Mapping and navigation systems.

Implementation

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/Dijkstra%20Problems/Dijkstra%20Basic/Dijkstra.cpp

Problem 1: Dijkstra?

Problem: Dijkstra? — Dijkstra? Problem from Codeforces
Link:
https://codeforces.com/contest/20/problem/C

Problem Description

The task is to determine the minimum-cost path from vertex 1 to vertex n in a weighted undirected graph. In simple terms, you need to find:

1. The smallest possible sum of edge weights needed to travel from vertex 1 to vertex n.
2. The exact sequence of vertices that forms this shortest path.

Input Format

n m
a1 b1 w1
a2 b2 w2
...
am bm wm

Symbol	**Meaning**
n	Total number of vertices ($2 \le n \le 10^5$)
m	Total number of edges ($0 \le m \le 10^5$)
a, b	Vertices connected by an edge
w	Weight of the edge ($1 \le w \le 10^6$)

Notes:

- Multiple edges between the same vertices and self-loops are allowed.

Output Format

- If there is **no path** from vertex 1 to n, print -1.
- Otherwise, print the **vertices of the shortest path** in order.

Approach

This is a Single Source Shortest Path (SSSP) problem with positive edge weights.

- If all edges had the same weight (e.g., 1), BFS could solve it.
- Since weights vary, Dijkstra's algorithm is the most efficient solution.

Dijkstra's Algorithm Overview

Dijkstra's algorithm is **a** greedy method that grows the shortest path tree from the start vertex (vertex 1).

Steps:

1. **Select:** Pick the unvisited vertex with the smallest known distance from the start.
2. **Relax:** Check all neighbors of the selected vertex and update their distances if a shorter path is found through the current vertex.
3. **Repeat:** Continue until vertex n is reached or all reachable vertices are processed.

Graph Representation

- Use an adjacency list for efficient storage.
- Each vertex stores a list of neighbors along with the edge weights.

Example adjacency list:

1 → (2,2), (4,1)
2 → (1,2), (3,4), (5,5)
3 → (2,4), (4,3), (5,1)
4 → (1,1), (3,3)
5 → (2,5), (3,1)

Key Variables

Variable	**Description**
dist[i]	Minimum distance from vertex 1 to vertex i
orig[i]	Parent of vertex i, to reconstruct the path
pq	Min-heap (priority queue) for selecting the vertex with the smallest dist
INF	A very large number (e.g., 1e18) to represent infinity

Example

Input:

```
5 6
1 2 2
2 5 5
2 3 4
1 4 1
4 3 3
3 5 1
```

Step-by-step execution:

Step	Current Node	Neighbors (v,w)	Relaxation / Updated dist[]	PQ content
0	-	-	dist[1]=0, others=∞	(0,1)
1	1	(2,2),(4,1)	dist[2]=2, dist[4]=1	(1,4),(2,2)
2	4	(1,1),(3,3)	dist[3]=4	(2,2),(4,3)
3	2	(1,2),(5,5),(3,4)	dist[5]=7 (temporary)	(4,3),(7,5)
4	3	(2,4),(4,3),(5,1)	dist[5]=5 (better!)	(5,5)
5	5	-	-	-

Path Reconstruction:
orig[5] = 3, orig[3] = 4, orig[4] = 1
=> Path: 1 → 4 → 3 → 5
Output:
1 4 3 5

Complexity

- **Time:** O((n + m) log n) — Each edge is relaxed once, and priority queue operations take log n.
- **Space:** O(n + m) — For adjacency list and dist/orig arrays.

Solution Code

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/Dijkstra%20Problems/Dijkstra%3F/20C_Dijkstra.cpp

Problem 2: Not The Best

Problem: Not the best — Not the best Problem in Lightoj

Link:

https://lightoj.com/problem/not-the-best

Problem Statement

Robin lives in a village with N intersections and R bidirectional roads. Instead of always taking the shortest route, he enjoys scenic paths and prefers the second shortest path**.**

Task: Find the cost of the second shortest path from intersection 1 to intersection N**.**

- **Second shortest path:**
  - Must be strictly longer than the shortest path.
  - Must be shorter than or equal to any other alternative path.

Note: Roads (edges) may be reused if needed.

Approach

We use a Modified Dijkstra's Algorithm that keeps two distances per node v:

- dist1[v] = shortest path to v
- dist2[v] = second shortest path to v

Relaxing an edge (u, v) with weight w:

1. If new_path < dist1[v]:
   - dist2[v] = dist1[v]
   - dist1[v] = new_path
2. If dist1[v] < new_path < dist2[v]:
   - dist2[v] = new_path

Reasoning: Dijkstra processes nodes by increasing distance. This ensures dist1 is the true shortest path, and dist2 captures the next shortest path.

Algorithm Steps

1. Initialize:
   - dist1[start] = 0, dist1[others] = ∞
   - dist2[all] = ∞
2. While priority queue is not empty:
   - Extract node with smallest current distance
   - Relax all neighbors using the **two-distance update rules**
3. Answer: dist2[N]

Input/Output Format

Input:

```
T
N R
u v w
...
```

- T = number of test cases
- N = nodes (1–5000)
- R = roads (1–100,000)
- u–v = connected intersections, w = road weight (1–5000)

Output:

```
Case X: second_shortest_path_cost
```

Example:

Input:

```
2
3 3
1 2 100
2 3 200
1 3 50
4 4
1 2 100
2 4 200
2 3 250
3 4 100
```

Output:

```
Case 1: 150
Case 2: 450
```

Graph Representation

We represent the graph using an **adjacency list**:

```
vector<vector<Edge>> adj(N + 1);

struct Edge {
   int to;
   int weight;
};
```

Why Use an Adjacency List?

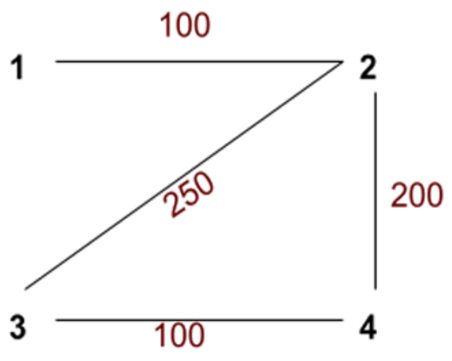
- The number of roads R can be very large (up to 10^5).
- Using an adjacency matrix would require N^2 memory (~25 million for N = 5000), which is inefficient.
- An adjacency list allows faster traversal and saves space**.**

Step-by-Step Visualization: Finding the Second Shortest Path

Example Input:

N = 4, R = 4
1 2 100
2 4 200
2 3 250
3 4 100

Goal: Robin wants to travel from **Intersection 1 → 4** and find the **second shortest path**.



Initial State

Node	**dist1**	**dist2**
1	0	∞
2	∞	∞
3	∞	∞
4	∞	∞

Priority Queue: { (0,1) }

Step 1: Process Node 1

- Neighbor: 1 → 2 = 100
    - Update: dist1[2] = 100

Node	**dist1**	**dist2**
1	0	∞
2	100	∞
3	∞	∞
4	∞	∞

Priority Queue: { (100,2) }

Step 2: Process Node 2

- Neighbor 2 → 4: 100 + 200 = 300 → dist1[4] = 300
- Neighbor 2 → 3: 100 + 250 = 350 → dist1[3] = 350

Node	**dist1**	**dist2**
1	0	∞
2	100	∞
3	350	∞
4	300	∞

Priority Queue: { (300,4), (350,3) }

Step 3: Process Node 4 (dist1 = 300)

- Neighbor 4 → 2: 300 + 200 = 500
    - Greater than dist1[2] = 100 and less than dist2[2] = ∞ → dist2[2] = 500

Node	dist1	dist2
1	0	∞
2	100	500
3	350	∞
4	300	∞

Priority Queue: { (350,3), (500,2) }

Step 4: Process Node 3 (dist1 = 350)

- Neighbor 3 → 2: 350 + 250 = 600 → No update (600 > dist1[2] and ≥ dist2[2])
- Neighbor 3 → 4: 350 + 100 = 450 → Update dist2[4] = 450

Node	dist1	dist2
1	0	∞
2	100	500
3	350	∞
4	300	450

Priority Queue: { (450,4), (500,2) }
Step 5: Process Node 4 (dist2 = 450)
450 is the second shortest path to Node 4.
Even after checking neighbors, no shorter second path exists.

Final Answer

- **Shortest Path:** dist1[4] = 300 → 1 → 2 → 4
- **Second Shortest Path:** dist2[4] = 450 → 1 → 2 → 3 → 4

Output:
Case 1: 450

Solution Code

https://github.com/Dipto-04/Algorithm-Lab-/blob/main/Dijkstra%20Problems/Not%20The%20Best/Not%20The%20Best.cpp