BFS

(Breadth First Search)

➢ What is BFS?

BFS is a technique used to systematically explore all the nodes in a tree or graph. It begins at a chosen starting node and examines every node at the same depth before moving on to the next layer. The process relies on a queue to maintain the order of exploration. Since it reaches the closest nodes first, BFS is particularly effective for finding the shortest path in unweighted graphs.

➢ Key Points:

- Works on **graphs and trees**.
- Explores nodes **level by level**.
- Uses a **queue** to keep track of the next nodes to visit.
- Gives the **shortest path** in unweighted graphs.
- Very useful for solving problems like shortest path, finding connected components, and level-order traversal.

➢ Why Do We Use BFS?

A few reasons make BFS extremely useful:

1. **Shortest Path (Unweighted Graph)**
   BFS guarantees the shortest number of edges from start to end
2. **Level-Based Problems**
   BFS naturally processes nodes level by level.
3. **Checking Connectivity**
   You can find which nodes are reachable.
4. **Building Trees Layer by Layer**
5. **Real-World Uses**
   - Social network friend suggestion
   - GPS navigation (when roads are considered equal)
   - Web crawlers

➢ How BFS Works:

Imagine a person standing at a starting node in a graph.
Instead of moving deep into one long chain of nodes, the person:

1. First visits all the nodes directly connected to the starting node.
2. Then moves to the next set of nodes that are connected to those nodes.
3. Continues this process layer by layer, expanding outward through the graph.

➢ Concepts Needed for BFS

To understand BFS clearly, these concepts help build the foundation needed to follow how the algorithm works.

- ✓ **Graph Basics**
  - Nodes (vertices): The individual points or positions in a graph.
  - Edges: The links or connections between nodes.
  - Directed vs Undirected Graphs: Whether edges have a specific direction or not.
- ✓ **Data Structures**
  - Queue: The primary structure used in BFS (FIFO — First In, First Out).
  - Adjacency List / Adjacency Matrix: Common ways to represent a graph in code.
  - Visited Array: Keeps track of which nodes have already been explored.
- ✓ **Important Terms**
  - Source Node: The node from where the BFS begins.
  - Neighbors: Nodes directly connected to the current node.
  - Levels: The distance from the source measured in steps or layers.

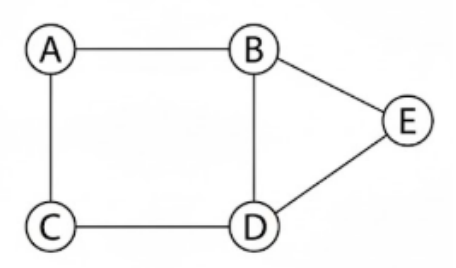
➢ BFS Algorithm — Step-by-Step

1. Pick a starting node.
2. Mark it as visited.
3. Push it into a queue.
4. While the queue is not empty:
   - Remove the front node.
   - Visit all its unvisited neighbors.
   - Mark neighbors as visited.
   - Push neighbors into the queue.

➢ BFS Pseudocode:

```
BFS(G, s) {
    for each u in V {
        color[u] = white
        d[u] = infinity
        pred[u] = null
    }
    color[s] = gray
    d[s] = 0
    Q = {s}
    while (Q is nonempty) {
        u = Q.Dequeue()
        for each v in Adj[u] {
            if (color[v] == white) {
                color[v] = gray
                d[v] = d[u]+1
                pred[v] = u
                Q.Enqueue(v)
            }
        }
        color[u] = black
    }
}
```

➢ Example Graph:



BFS from source node A:

- ✓ **Step 1:** Start from A
  Visited: A
  Queue: [B, C]
- ✓ **Step 2:** Visit B
  Visited: A, B
  Queue: [C, D, E]
- ✓ **Step 3:** Visit C
  Visited: A, B, C
  Queue: [D, E]
- ✓ **Step 4:** Visit D
  Visited: A, B, C, D
  Queue: [E]
- ✓ **Step 5:** Visit E
  Visited: A, B, C, D, E
  Queue: []

BFS Order: A → B → C → D → E

➢ Time & Space Complexity

- Time Complexity: O(V + E)
- Space Complexity: O(V)

➢ When Not to Use BFS?

Although BFS is great, not every problem fits it. Use BFS only when:

- Graph is unweighted or all weights are equal.
- You want the shortest path in terms of number of edges.

Use Dijkstra if weights differ. Use DFS for deep exploration or recursion-based tasks.

➢ Summary:

After learning BFS, one can traverse graphs, trees, and grids, find shortest paths in unweighted graphs, count levels or connected components, check whether a graph is bipartite, solve maze or grid-based problems, and implement advanced BFS variations such as multi-source BFS or BFS with path reconstruction.

➢ Implementation:

https://github.com/nafisabindu/Algorithm-/blob/main/BFS/BFS%20basic/Implementation.cpp

Problem-1

- **Problem:** Bicoloring (BFS) — Bicoloring problem in Onlinejudge

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

- **Solution Approach:**

Can You Two-Color This Graph?
Exploring the Bipartite World!

Ever tried to color a map? You know, making sure no two neighboring countries share the same shade? It's a classic puzzle! The famous "Four Color Map Theorem" even states that any map can be colored with just four colors, a fact proven with computer assistance back in 1976. Mind-blowing, right?

But what if I told you we're going to tackle an even simpler, yet equally fascinating, version of that problem? Today, we're diving into the world of **bicolorable graphs**. Forget four colors; we're trying to see if we can get away with just **two**!

The Challenge: Bicoloring a Graph

Imagine a network of friends, where lines connect people who know each other. Our challenge is to assign each person one of two "team colors" (let's say red or blue) such that no two friends (connected by a line) end up on the same team. If we can do this, our friend network (or graph, in math speak) is "bicolorable."
Sounds simple, right? Well, sometimes it is, and sometimes it's a tricky little beast!

What Makes a Graph Bicolorable?

At its heart, a bicolorable graph is also known as a **bipartite graph**. Think of it like this: you can perfectly divide all the nodes (our friends) into two groups. Every single connection (friendship) then goes *between* a person in one group and a person in the other, never *within* the same group.

The secret sauce to identifying a bipartite graph lies in its cycles. If you can trace a path from a node back to itself that involves an **odd number of edges**, then congratulations, you've found an "odd cycle"! And if a graph has *any* odd-length cycles, it's impossible to two-color it. Why? Because as you alternate colors along an odd cycle, you'll inevitably end up needing to assign the same color to two adjacent nodes.

Conversely, if a graph has *no* odd-length cycles, it's always bicolorable! Every single time.

How Do We Figure It Out? Meet BFS!

So, how do we actually check this? We can't just stare at every graph and instantly spot odd cycles (unless it's tiny!). This is where algorithms come to the rescue, specifically **Breadth-First Search (BFS)**.

Here's a simplified rundown of our strategy:

1. **Pick a Starting Point:** Since our graphs are guaranteed to be "strongly connected" (meaning you can get from any node to any other), we just pick any node to start, say Node 0.
2. **Color It!** We give Node 0 our first color (let's say, red).
3. **Spread the Word (and Color):** Now, we look at all of Node 0's direct neighbors. They *must* be the opposite color (blue). We mark them blue and add them to our "to visit next" list.
4. **Keep Alternating:** We continue this process. When we visit a blue node, all its uncolored neighbors must be red. When we visit a red node, its uncolored neighbors must be blue.
5. **Spotting Trouble:** What happens if we try to color a neighbor, but it's *already* colored, and it has the *same* color as the node we're currently processing? BINGO! We've found a conflict. This means there's an odd-length cycle, and the graph is **NOT BICOLORABLE**.
6. **Success!** If we go through the entire graph, coloring every node without running into any conflicts, then pat yourself on the back! The graph **IS BICOLORABLE**.

Here's a quick visual of BFS in action on a bicolorable graph:

Why This Matters (Beyond Puzzles!)

This problem isn't just a fun brain teaser for computer science students. Bipartite graphs have real-world applications in many fields:

- **Scheduling:** Imagine assigning students to projects. If each student can only be assigned to one project, and each project needs a student, this can often be modeled as a bipartite matching problem.
- **Networking:** Optimizing network routing or resource allocation.
- **Chemistry:** Understanding molecular structures.
- **Data Science:** Analyzing relationships in complex datasets.

➢ **Solution Code:**

https://github.com/nafisabindu/Algorithm-/blob/main/BFS/BICOLORING/Bicoloring.cpp

Problem-2

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

Link:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=508

➢ **Solution Approach:**

The World Conquest Puzzle: Finding the Shortest Path in Risk

If you've ever played **Risk**, the classic game of global domination, you know the feeling: you've built up a massive army in one corner of the map, and now you need to shuttle them across several territories to strike a vital target on the other side. This isn't just a physical move; it's a series of calculated conquests.

The goal is simple: **get your armies from point A to point B using the fewest conquered countries possible**. It's a real-world strategic problem hidden inside a cardboard box, and it turns out, we can solve it with a fundamental concept from computer science: the **Shortest Path Algorithm.**

The Risk Board as a Graph

To a computer scientist, the Risk board isn't a map of countries; it's a graph.

- Countries are the nodes (or vertices).
- Shared Borders are the edges (or connections) that link the nodes.

In the game of Risk, an army can only attack a country if it shares a border with the attacking country. In graph terms, this means you can only move from your current node to an adjacent node.

The problem we want to solve—finding the minimum number of countries to conquer—is exactly equivalent to finding the path with the fewest number of steps (edges) between the starting country and the destination country.

The Tool for the Job: Breadth-First Search (BFS)

When every "step" or "conquest" costs the same amount (one country), the fastest way to find the shortest path isn't a complex algorithm like Dijkstra's, but a simpler one called **Breadth-First Search (BFS).**

How BFS Works: The 'Rippling Wave' Approach

Imagine dropping a stone in a pond. BFS works like the ripples spreading out:

1. Start: Begin at the starting country. Mark its distance from the start as 0.
2. **Level 1:** Find all countries that border the starting country (neighbors). These are the countries you can conquer in 1 move. You add them to a list (a queue) to visit next.
3. **Level 2:** Now, look at all the countries you found in Level 1. Find their unvisited neighbors. These are the countries you can reach in 2 moves (two conquests).

4. **Repeat:** The process continues, level by level, ensuring that you always explore all countries reachable in k moves before moving on to countries reachable in k+1 moves.

Because BFS explores the graph in this widening, concentric manner, the moment you hit the destination country, you are guaranteed to have found the path with the absolute fewest number of border crossings.

The Conquest Calculation

The key to the final answer is remembering what the question asks for: the minimum number of countries that must be conquered, including the destination.

- If the shortest path is 3 borders (edges) long, that means you go: **Start**→Country 1 →Country 2 →Destination.
- The countries you had to conquer are: Country 1, Country 2, and the Destination.
- Number of Conquests = (Number of Edges in Path) + 1

If your starting country and destination are neighbors, the path has 1 edge, and the number of conquests is 1 + 1 = 2. Wait!

Correction based on the problem statement:

The problem states: "For example, if starting and destination countries are neighbors, then your program should return one."

This means the "conquest count" is just the number of countries we pass through to reach the destination, starting with the first one conquered.

- Start→Destination (1 border, 1 conquest)
- Start→Country 1 →Destination (2 borders, 2 conquests)
- The minimum number of conquests is equivalent to the Shortest Path Length in Edges (Borders).

So, the BFS algorithm finds the minimum number of borders that must be crossed, and that is our answer!

Implementation Insight (C++)

To implement this solution, we use a C++ program that:

1. **Builds the Adjacency List:** It reads the 19 lines of input to create an efficient representation of the map. An `std::vector<std::vector<int>>` is perfect for this, where the index represents a country and the inner list holds all its bordering countries.

2. Runs BFS: For each query (Start → Destination), it uses a queue and a distance array (initialized to -1) to manage the exploration, stopping the moment the destination is dequeued.

3. Formats Output: The final step involves carefully using C++ stream manipulators like `std::setw` and `std::right` to match the required column-based format for the output.

It's a wonderful example of how game strategy and real-world logistics often boil down to pure, elegant graph theory. Next time you're plotting a move in Risk, you'll know you're just mentally running a **Breadth-First Search**!

- **Solution Code:**
  https://github.com/nafisabindu/Algorithm-/blob/main/BFS/Risk/Risk.cpp

Problem-3

- **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=380

- **Solution Approach**

The Knight's Journey:
Why Finding the Shortest Hop is Fast, But the Full Tour is a Nightmare

We've all been there: staring at a giant problem and trying to figure out which part is the *real* obstacle. A friend recently wrestling with the **Traveling Knight Problem (TKP)** had this exact dilemma. He figured the toughest challenge was simply calculating the minimum number of knight moves between any two arbitrary squares, say, from 'a1' to 'f6'.

He was wrong. And his mistake brilliantly highlights a fundamental difference between two key concepts in computer science: **Shortest Path** and **Hamiltonian Cycle**.

The Illusion of Difficulty: A to B

The problem we successfully solved for him—"What is the shortest number of knight moves from square X to square Y?"—*feels* difficult because you have to think several moves ahead. But computationally, it's a walk in the park.

On a chessboard, every square is a node, and every legal knight move is a connection (or an *edge*). Since all moves count as '1' (it costs one move to get from E2 to G3), we are dealing with an **unweighted graph**.

The perfect tool for finding the minimum distance in an unweighted graph is **Breadth-First Search (BFS)**.

The Power of BFS

Imagine you start a rumor at E2.

1. **Time 0:** Only E2 knows the rumor.
2. **Time 1:** The rumor spreads to every square reachable in one hop (G3, F4, D4, C3...).
3. **Time 2:** From those Time 1 squares, the rumor spreads to the next layer (F6, H4, E6...).

BFS is designed to explore the entire graph *in layers of increasing distance*. The moment the "rumor" (our search) reaches the target square (say, E4), we know two things:

1. We have found the target.
2. Because we explored layer by layer, this must be the path with the fewest steps.

Our C++ program simply formalizes this process. It converts the algebraic notation (like 'a1') into simple coordinates (0, 0), uses a queue to track which squares to visit next, and uses an 8x8 array to prevent revisiting squares and to record the distance. It is highly efficient, guaranteeing the shortest path every time.

The True Nightmare: The Traveling Knight Tour (TKP)

The moment we change the goal from "get from A to B" to the **Traveling Knight Problem**, the computational cost skyrockets.

The TKP asks: **Can you find a single, continuous, closed path (a tour) that visits every square on the entire 8x8 chessboard exactly once?**

This is not a shortest path problem; it is a **Hamiltonian Cycle Problem**, and solving it moves us from the realm of "easy and fast" algorithms (polynomial time, or P) to the terrifying landscape of **NP-hard** problems.

The Tyranny of the Global Choice

In the shortest path problem, a local choice (the next move) only affects the short-term distance. In the TKP, a local choice has **global, irreversible consequences.**

For example, if the knight starts at A1 and moves to C2, that move might seem fine. But what if C2 is the only square that can connect to a corner square like H8 later in the game? By taking that path early, you might cut off a crucial future connection, making the entire tour impossible—and you won't know it until you're 60 moves deep!

- **Shortest Path:** We only need to find *one* path out of billions.
- **Hamiltonian Cycle:** We need to ensure the *entire* sequence of 64 moves forms a continuous, unbroken, non-repeating loop.

How Mathematicians Tackle the TKP

Since we can't efficiently use BFS or similar quick algorithms, mathematicians and programmers resort to strategies that deal with this global constraint:

1. **Backtracking:** The most basic method. Try a path, and if it hits a dead end, retreat (backtrack) and try the last choice again. This is extremely slow because the number of possible paths is astronomical.
2. **Warnsdorff's Rule (Heuristic):** A clever rule that dictates: always move to the square from which the knight has the fewest subsequent legal moves. This is a greedy strategy; it tries to leave the hardest-to-reach squares for later, increasing the probability of closing the loop. It often works but is not guaranteed to find a solution.
3. **Advanced Optimization:** For truly difficult instances of the TSP (which the TKP is related to), researchers use advanced methods like genetic algorithms or constraint programming.

➢ **Solution Code:**
https://github.com/nafisabindu/Algorithm-/blob/main/BFS/KNIGHT%20MOVES/Knight%20moves.cpp

Dijkstra

- **What is Dijkstra?**

  Dijkstra's Algorithm is one of the most famous and widely used algorithms for finding the **shortest path** from a source node to all other nodes in a graph—as long as all edge weights are non-negative.

- **Why Do We Use Dijkstra?**

  Choose Dijkstra when:

  - The graph has non-negative weights
  - You need fast shortest-path computation
  - Performance matters (Dijkstra is much faster than Bellman–Ford)

- **Key Points:**

  - ✓ **Weighted Graph**

    Each edge has a positive cost/weight.
  - ✓ **Priority Queue (Min-Heap)**

    Dijkstra uses a **priority queue** to always pick the next closest node.
  - ✓ **Relaxation**

    Just like Bellman–Ford, Dijkstra updates distances when it finds a shorter path.
  - ✓ **Distance Array**

    Stores the shortest known distance to each node.

- **How Dijkstra Works**

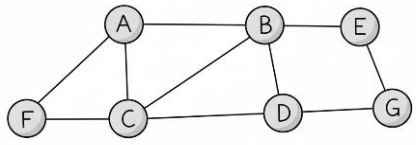
  1. Set all distances to infinity, except the source which is 0.
  2. Push the source into a min-priority queue.
  3. While the queue is not empty:
     - Extract the node with the smallest distance.
     - For each neighbor:
       - Check if the path through this node is shorter.
       - If yes, update the distance and push the neighbor into the queue.

  Because weights are non-negative, once a node is processed, its shortest distance is final.

- **Pseudocode:**

```
Dijkstra(G, source) {
  for each vertex u in G {
      dist[u] = infinity
     parent[u] = null
}
   dist[source] = 0
  PQ = priority_queue()
  PQ.push( (0, source) )
  while (PQ is not empty) {
(du, u) = PQ.pop()
      for each v in Adj[u] {
        w = weight(u, v)
       if (dist[u] + w < dist[v]) {
       dist[v] = dist[u] + w
      parent[v] = u
     PQ.push( (dist[v], v) )   }
  }
}
  for each vertex u in G {
  print dist[u]
}
}
```

- **Example Graph**



Dijkstra's Algorithm from node A:

Assuming all edges have weight 1

- ✓ **Initial:**
  Distance: A=0, others=∞
  Unvisited: {A, B, C, D, E, F, G}

- ✓ **Step 1:** Visit A (distance=0)
  Update: B=1, C=1, F=1
  Unvisited: {B, C, D, E, F, G}

- ✓ **Step 2:** Visit B (distance=1)
  Update: D=2, E=2
  Unvisited: {C, D, E, F, G}

- ✓ **Step 3:** Visit C (distance=1)
  Update: D=2 (already), F=2 (already)
  Unvisited: {D, E, F, G}

- ✓ **Step 4:** Visit F (distance=1)
  No new updates
  Unvisited: {D, E, G}

- ✓ **Step 5:** Visit D (distance=2)
  Update: G=3
  Unvisited: {E, G}

- ✓ **Step 6:** Visit E (distance=2)
  Update: G=3 (already)
  Unvisited: {G}

- ✓ **Step 7:** Visit G (distance=3)

Shortest distances from A:

A→A: 0
A→B: 1
A→C: 1
A→F: 1
A→D: 2
A→E: 2
A→G: 3

➢ Time & Space Complexity

Using a Min-Heap

- Time complexity: O((V + E) log V)
- Space complexity: O(V)

This makes Dijkstra one of the fastest shortest-path algorithms.

➢ What You Need to Implement Dijkstra

- A graph representation:
  - Adjacency list (recommended)
- A priority queue (min-heap)
- Distance array
- Source vertex

➢ When NOT to Use Dijkstra

Do not use Dijkstra when:

- The graph has negative weights → use Bellman–Ford
- You need all-pairs shortest paths → use Floyd–Warshall

➢ Final Thoughts

Dijkstra's Algorithm is fast, useful, and essential for learning pathfinding and graph algorithms. After mastering it, you can explore:

- **A*** Search — faster pathfinding with heuristics
- **Floyd–Warshall** — all-pairs shortest paths
- **Johnson's Algorithm** — efficient all-pairs for sparse graphs

➢ Implementation:

https://github.com/nafisabindu/Algorithm-/blob/main/Dijkstra/Dijkstra(basic)/implementation%202.cpp

Problem-1

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

- **Solution Approach:**

The Road Not Taken:
Solving the Shortest Path Problem with Dijkstra's Algorithm

Every time you open Google Maps, send a data packet across the internet, or try to navigate a complex warehouse, you are relying on the shortest path problem being solved instantly. It's one of the most fundamental challenges in computer science.
The problem, in its simplest form, is this: Given a network of locations (vertices) connected by roads (edges), each with a cost (weight), what is the quickest way to get from point A to point B? In our specific challenge, we were tasked with finding the shortest route from vertex 1 to vertex N in a weighted, undirected graph.
With up to 10^5 vertices and edges, a brute-force approach would be impossible. We need a sophisticated, efficient solution. The answer lies in a legendary algorithm developed by Dutch computer scientist Edsger W. Dijkstra: **Dijkstra's Algorithm.**

Why Dijkstra's Wins: The Greedy Approach

Dijkstra's Algorithm is a greedy algorithm. This means it always makes the locally optimal choice at each step, confident that this will lead to the global optimum. Because all edge weights in our graph are positive (non-negative), this greedy strategy works perfectly.
The core idea is simple:

1. **Start Small:** Begin at the starting vertex (vertex 1) and assign it a distance of 0. All other vertices are assigned a theoretical "infinite" distance.
2. **Explore and Relax:** Repeatedly select the unvisited vertex with the smallest current distance.
3. **Update Neighbors:** For that chosen vertex, look at all its neighbors. If travelling to a neighbor *through* the current vertex results in a shorter total path than the neighbor currently has, we update the neighbor's distance. This is called relaxation.

Under the Hood: The C++ Implementation

To handle the large constraints (N, M <= 10^5), the efficiency of the implementation is crucial. We must use a priority queue to quickly find the vertex with the smallest distance in step 2.

In the provided C++ solution (shortest_path.cpp), here's how the key components work:

1. **Adjacency List (adj):** The graph is stored as an adjacency list. For an undirected graph, every edge (u, v) with weight w is added to both u's list and v's list.

2. **Distance and Parent Arrays:**
   - dist[]: Stores the minimum distance from the start (vertex 1) to every other vertex. It's initialized to INF (1e18).

- parent[]: This is the secret to path reconstruction. When we relax an edge and update dist[v] through u, we set parent[v] = u. This means we record where we came *from* to get to v most efficiently.

3. **The Priority Queue (pq):** This is where the magic happens. We use (std::priority_queue<PII), (vector<PII>), (greater<PII>>) to create a min-heap. It stores pairs of {distance, vertex}. The (greater<PII>) template argument ensures that the vertex with the smallest distance is always at the top, ready to be processed next.

The time complexity is roughly O((N+M) log N), making it incredibly fast for large graphs.

Finding the Path

Dijkstra's algorithm only calculates the shortest *distance*; to find the actual sequence of vertices, we rely entirely on the parent array.

Once the algorithm finishes, if dist[N] is still INF, no path exists, and we output -1. Otherwise, we perform backtracking:

1. Start at the target vertex N.
2. Add N to the path list.
3. Move to its parent (curr = parent[curr]).
4. Repeat until the parent is 0 (which signifies the starting vertex, 1, or an unvisited vertex).
5. Finally, we reverse the list, turning the N →⋯→ 1 sequence into the required 1→⋯→ N shortest path.

This elegant combination of a greedy choice mechanism and efficient data structures allows us to solve massive navigation problems with speed and precision.

- **Solution Code:**
  https://github.com/nafisabindu/Algorithm-/blob/main/Dijkstra/Dijkstra%3F/Dijkstra.cpp

Problem-2

- **Problem:** Not the best — Not the best Problem in Lightoj
  Link:
  https://lightoj.com/problem/not-the-best

- **Solution Approach:**.

The Scenery Seeker's Dilemma: Finding the Second-Shortest Path

My friend Robin recently moved to a lovely, sleepy village. He often visits his best mate in town, but being a true admirer of the countryside, he refuses to take the quickest route. Instead, he opts for the second-best path—the one that's scenic but still practical.

This simple preference presents us with a fascinating challenge: How do we computationally find the second-shortest path in a complex network of roads?

This problem, often encountered in algorithms and graph theory, requires a clever adaptation of our go-to shortest path solver: Dijkstra's Algorithm.

Why the Standard Shortest Path Won't Cut It

When faced with a weighted graph (a map with distances, like Robin's roads), our first instinct is usually to run Dijkstra's Algorithm. This algorithm efficiently finds the single shortest path from a starting point (Robin's house, Node 1) to all other points (his friend's house, Node N).

But here's the kicker:

- The shortest path might be unique, or there might be dozens of them with the exact same minimum length.
- Robin wants the path that is strictly longer than the absolute shortest one, but still the shortest among all the remaining options.
- The path can also be complex: it can use the same road multiple times (backtracking) or pass through the same intersection more than once.

We need a way to track the two best possibilities for every single intersection we visit.

The Elegant Solution: Dijkstra's with a Twist

The trick to solving this is to modify the state we track for each intersection. Instead of recording just one shortest distance, we need to track two:

1. d1[i]: The length of the shortest path found so far to intersection i.
2. d2[i]: The length of the second-shortest path found so far to intersection i.

We initialize all d1 and d2 values to ∞, except for the start node, where d1[1] = 0.

The Relaxation Logic

The core of Dijkstra's algorithm is the relaxation step, where we update the distances to a neighbor. When we extract a path of length d to a node u from the priority queue and consider a neighbor v connected by a road of weight w, we calculate the potential new path length: new d = d + w.

We then apply a cascading update rule to v:

1. **Found a Better Shortest Path** (new d < d1[v])
   If the new path is shorter than the currently best path to v:

- The old shortest path length, d1[v], is demoted and becomes the new second-shortest path length, d2[v].
- d1[v] is updated to new d.

2. Found a Better Second-Shortest Path (d1[v] < new d < d2[v])

If the new path is strictly longer than the shortest path but shorter than the second-shortest path:

- d2[v] is updated to new d.
- d1[v] remains unchanged.

We push all updated distances back into the priority queue to continue the search.

The Final Result

Once the modified Dijkstra's algorithm finishes (when the priority queue is empty), the answer to Robin's dilemma is simply the value stored in d2[N]—the second-shortest path length to his friend's house (the destination node N).

This method ensures we explore all relevant paths efficiently, pruning only the ones that are definitely longer than our two current best options at any given point. It turns a seemingly complex pathfinding puzzle into an elegant, manageable adaptation of a foundational algorithm.

- **Solution Code:**
  https://github.com/nafisabindu/Algorithm-/blob/main/Dijkstra/Not%20the%20best/Not%20the%20best.cpp

Bellman Ford

➢ What is Bellman Ford?

Bellman–Ford is a shortest path algorithm that finds the minimum distance from a single source node to all other nodes in a weighted graph.
It works even when:

- Edges have negative weights
- The graph contains cycles

However, it cannot handle graphs with negative weight cycles that are reachable from the source — but it can detect them.

➢ When Should You Use Bellman–Ford?

You choose Bellman–Ford over other algorithms when:

- The graph has negative edge weights
- You need to detect negative cycles
- You don't need extremely fast performance (Bellman–Ford is slower than Dijkstra)

Common applications:

- Currency arbitrage detection (negative cycles represent profit opportunities)
- Routing protocols (e.g., RIP)
- Optimizing paths in networks where weights may decrease

➢ Key Concepts You Need to Know

1. **Weighted Graph**

   Bellman–Ford works on a weighted graph where each edge has a cost (positive or negative).

2. **Relaxation**

   This is the heart of the algorithm. To "relax" an edge (u → v with weight w) means:

   - If the shortest known path to v can be improved by going through u, update it.

3. **Negative Weight Cycle**

   A cycle whose edges sum to a negative value.

Bellman–Ford can detect this by trying one more relaxation step after finishing the main loop.

➢ How Bellman–Ford Works

1. Initialize all distances to infinity, except the source which is set to 0.
2. Repeat the relaxation process (V − 1) times (where V = number of vertices):
   - For every edge, try to update the shortest path.
3. Perform one more relaxation loop:
   - If any distance updates again, it means there is a negative cycle.

- **Pseudocode:**

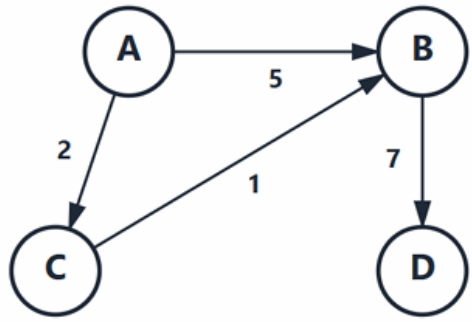
```
BELLMAN-FORD (G,s)

    INITIALIZE SINGLE – SOURCE (G,s)

     for i ← 1 to |V|-1 do
        for each edge (u,v) ∈ G.E do
          RELAX (U, V)
     for each edge (u,v) ∈ E do
           if d[V] > d[U] + W(U, V) then

return FALSE
return TRUE
```

- **Example Graph:**



Bellman-Ford Algorithm from node A:

- ✓ **Initial:**
  Distance: A=0, B=∞, C=∞, D=∞
- ✓ **Iteration 1:**
  A→B: dist[B] = min(∞, 0+5) = 5
  A→C: dist[C] = min(∞, 0+2) = 2
  C→B: dist[B] = min(5, 2+1) = 3
  B→D: dist[D] = min(∞, 3+7) = 10

After Iteration 1: A=0, B=3, C=2, D=10

- ✓ **Iteration 2:**
  A→B: dist[B] = min(3, 0+5) = 3
  A→C: dist[C] = min(2, 0+2) = 2
  C→B: dist[B] = min(3, 2+1) = 3
  B→D: dist[D] = min(10, 3+7) = 10

After Iteration 2: A=0, B=3, C=2, D=10

- ✓ **Iteration 3:**
  No changes (distances stabilized)

Final shortest distances from A:

A→A: 0
A→C: 2
A→B: 3 (via C)
A→D: 10 (via C→B)

No negative cycles detected!

- **Time & Space Complexity**
    - Time complexity: O(V × E)
    - Space complexity: O(V)

This makes Bellman–Ford slower than Dijkstra, but far more flexible.

- **What Need to Implement Bellman–Ford?**
    - A list of edges (edge list is ideal)
    - A distance array
    - The number of vertices V

Graph representations:

- Edge list (best for Bellman–Ford)
- Adjacency list (usable, but requires converting to edges)

- **Final Thoughts**

Bellman–Ford may not be the fastest, but it is one of the most important graph algorithms due to its ability to handle negative weights and detect negative cycles.
Once you understand Bellman–Ford, you're ready to explore:

- Dijkstra's Algorithm
- Floyd–Warshall (all-pairs shortest path)
- Johnson's Algorithm

- **Implementation:**

https://github.com/nafisabindu/Algorithm-/blob/main/BellmanFord/BellmanFord(basic)/implementation%203.cpp

Problem-1:

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

- **Solution approach:**

Can We See the Big Bang?
The Bellman-Ford Algorithm and Time Travel

Imagine a future where warp drives are still science fiction, but wormholes—subspace tunnels connecting distant star systems—are a reality. Our brilliant physicist wants to travel back to the dawn of time, the Big Bang, by exploiting a strange property of these cosmic shortcuts: they can shift you forward or *backward* in time.
The core problem, as tackled by the C++ code you selected from the Canvas, isn't actually about physics; it's a profound question of computer science: **Does a negative cycle exist in the network of wormholes?**

The Cosmic Map is a Graph

To a computer scientist, the universe in the year 2163 is not filled with stars, but with nodes and edges.

- Star Systems (Earth, Alpha Centauri, etc.) become Nodes (or Vertices).
- Wormholes become Edges (or paths), connecting two nodes. Since wormholes are one-way, this is a Directed Graph.
- The Time Difference (t) becomes the Weight of the edge.

If a wormhole sends you 15 years into the future, that's a positive weight (+15). If a wormhole sends you 42 years into the past, that's a negative weight (-42).

The Loophole to the Past

The scientist's goal—reaching the Big Bang—requires an infinite trip backward in time. In graph theory, this corresponds exactly to finding a Negative Weight Cycle.

A cycle is simply a closed loop where you can travel from one star system back to itself. If the *sum* of the time differences along this cycle is negative (e.g., 10 + 15 - 42 = -17), then every time you complete the loop, you jump 17 years further into the past. By repeating this loop infinitely, you can reach any point in the past, including the moment of creation.

Enter the Bellman-Ford Algorithm

When we look for the shortest path between nodes in a graph, we usually turn to Dijkstra's algorithm. However, Dijkstra's fails spectacularly when faced with negative weights—it simply can't handle the possibility of a path getting shorter indefinitely.

This is where the **Bellman-Ford algorithm**, the engine behind the C++ solution, shines. It is specifically designed to handle negative edge weights and, crucially, to detect negative cycles.

How Bellman-Ford Works :

The algorithm operates by repeatedly trying to relax (or shorten) the time taken to reach every star system in the network. If there are N star systems, here's the process:

1. **The N-1 Passes:** The algorithm iterates over all wormholes N-1 times. Since the longest *simple* path (a path with no cycles) in a graph has N-1 edges, by the end of this phase, we are guaranteed to have found the shortest time path to every reachable node, provided there are no negative cycles.
2. **The N-th Check:** This is the decisive step. The algorithm runs one final, N-th pass over all the wormholes.
    - **If no further path times decrease:** It means the N-1 passes were enough. No negative cycle exists. Time travel to the Big Bang is not possible.
    - **If any path time does decrease:** This can only happen if a negative cycle has just been traversed, allowing the path time to continuously drop. A negative cycle is reachable, meaning infinite time travel is possible.

In the Canvas C++ code, the *hasNegativeCycle* function perfectly implements this logic, starting with system 0 (our solar system) and checking if any wormhole can still "relax" the time difference after N-1 initial passes. This is a brilliant application of a fundamental computer science concept to solve a theoretical physics puzzle.

- **Solution Code:**

  https://github.com/nafisabindu/Algorithm-/blob/main/BellmanFord/UVA558/UVA558.cpp

Problem-2:

- **Problem:** LOJ1108 — Traffic problem in Onlinejudge
  Link:
  https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1390

- **Solution Approach:**

The Shortest Path Trap:
Why Dhaka's Traffic Problem is a Computer Science Puzzle

Dhaka, Bangladesh. Just mentioning the name conjures images of incredible energy, vibrant culture, and—let's be honest—some of the world's most epic traffic jams. We've all been there: staring at our navigation app, trusting it to give us the shortest route, only to realize we're now trapped on the most notorious, congested road in the city.

What if there was a way to make the journey not just short, but *smarter*?

I recently came across a classic competitive programming problem that offers a brilliant, if slightly ruthless, solution to this urban headache. It turns the simple act of choosing a route into a fascinating challenge in graph theory.

The Anti-Shortest Path Plan

Imagine the city authorities decide to actively discourage the use of perpetually blocked roads. They don't do this by adding tolls; they do it by creating a monetary incentive (or penalty!) tied to the busyness of each area.

The Setup

1. **Junction Busyness**: Every junction (an intersection) in the city is marked with a "busyness" value, a small positive integer (1 to 20). High number = High busyness.
2. **The Earning Rule:** When you travel from one junction (Source,u) to the next (Destination,v), the city "earns" an amount calculated by:

   $$\text{Earning} = (\text{Destination} - \text{Source})^3$$

This formula is the heart of the puzzle.

- It means a road connecting two equally busy areas gives an earning of $(x - x)^3 = 0$. But if you go from a low-busyness area (e.g., a quiet neighborhood, value 6) to a high-busyness area (e.g., the central business district, value 10), the earning is

  $$(10-6)^3 = 4^3 = 64$$

  The authority *earns* a lot.
- Crucially, if you go from high-busyness (10) to low-busyness (6), the earning is $(6-10)^3 = (-4)^3 = -64$.

The authority effectively *loses* (or refunds) money!

The Goal: Minimizing the Authority's Total Earning

The city's challenge is to find the path that results in the minimum total cumulative earning for the authority.

Why the Shortest Path Algorithm Isn't Enough

At its core, this is a Shortest Path Problem—finding the path with the smallest sum of "costs" (the earnings) from a starting point (Junction 1) to all other junctions.
If all the earnings were positive, we could simply use the famous Dijkstra's Algorithm.
But look at that formula again: a path segment can have a negative earning (cost). This is a game-changer for shortest path algorithms.

The Negative Cycle Problem

A negative cost is fine, but if you create a loop (a cycle) where the sum of earnings around that loop is negative, you have a negative cycle.
If you find a negative cycle, you can traverse it infinitely, driving the total cumulative earning down towards negative infinity! This means the true "minimum" cost is unbounded.
For real-world problem-solving, we need an algorithm that can handle these negative costs and, more importantly, detect those negative cycles.

The Solution: Enter Bellman-Ford

The perfect tool for this job is the Bellman-Ford Algorithm.

1. **Handling Negatives:** Unlike Dijkstra's, Bellman-Ford doesn't break down when it encounters negative edge weights.
2. **Detection:** It performs N-1 passes over all edges (where N is the number of junctions) to find the shortest path. If, after those N-1 passes, it can still find a path that can be "relaxed" (made shorter) in the N-th pass, it has definitively detected a negative cycle.

In the context of our Dhaka problem, if the minimum total earning to a destination is still being reduced in the final pass, it means that destination is reachable from an endless, loss-making loop, and the city authority's earning is effectively $-\infty$

Final Output Rule

The problem adds one last layer of complexity:

- If the minimum total earning is less than 3, the city authority finds the earning too small to bother reporting, and we must print a '?'.
- This '?' also covers nodes that are unreachable (earning is "infinity") or those caught in a negative cycle (earning is "negative infinity").

This puzzle beautifully illustrates how urban planning can be modeled as a complex optimization problem. By creating a smart cost function that rewards travelers for using lower-busyness areas and penalizes them for overloading central zones, the city can nudge traffic distribution toward a more balanced, efficient outcome.

➢ **Solution Code:**

https://github.com/nafisabindu/Algorithm-/blob/main/BellmanFord/LOJ1108/LOJ1108.cpp