# BFS

## (Breadth-First Search)

---

### ➢ What Is BFS?

BFS is a way to explore a graph step by step, level by level. You start from one node and visit all the neighbors close to it first, then you move outward.

Think of it like dropping a pebble in water the waves move outward in circles. BFS does the same in a graph.

### ➢ Main Ideas of BFS

**1.Uses a Queue (FIFO):** BFS uses a queue, meaning the first item added is the first to come out.

**2. Keeps Track of Visited Nodes:** We mark nodes as "visited" so we don't process the same node again.

**3. Works Level by Level:** BFS always explores the closest nodes first, then the next layer of nodes.

### ➢ What You Need

To use or write BFS, you need:

- A graph (usually stored as an adjacency list)
- A queue
- A visited list or set
- A starting node

### ➢ How BFS Works

1. You start at one node.
2. You look at all the nodes directly connected to it.
3. After finishing those, you move to the nodes that are one step farther.
4. You continue moving outward, layer by layer.
5. You stop when there are no more nodes left to explore.

## ➢ Where BFS Is Used

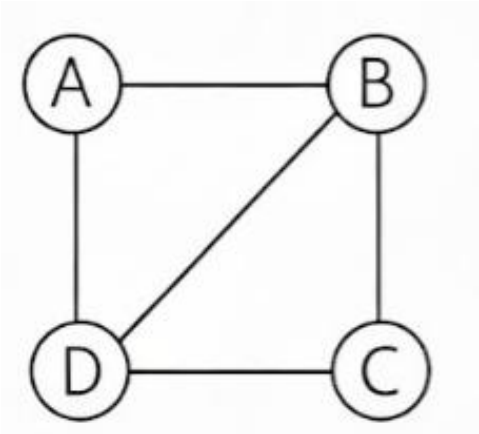
**BFS is helpful in many real-life problems:**

- Finding the shortest path in an unweighted graph
- Maze solving
- Social network friend suggestions
- Web crawling
- Networking and message broadcasting

## ➢ 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
    }
}
```

- **simple Diagrams for Understanding BFS**




BFS from node A:

Step 1: Start at A

Visited: A

Queue: [B, D]

Step 2: Visit B

Visited: A, B

Queue: [D, C]

Step 3: Visit D

Visited: A, B, D

Queue: [C]

Step 4: Visit C

Visited: A, B, D, C

Queue: []

BFS Order: A → B → D → C

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

- **After learning BFS:** you can traverse graphs, trees, and grids, find shortest paths in unweighted graphs, count levels or connected components, check if a graph is bipartite, solve maze or grid-based problems, and implement advanced BFS variations like multi-source BFS or BFS with path reconstruction.
- **Implementation:**
  **https://github.com/angkitasaha/Algorithm/blob/main/BFS/BFS%20Basic/Implementation**

- **Problem1:**
  **Question:** Bicoloring (BFS) — Bicoloring problem in Onlinejudge
  **Link:** https://onlinejudge.org/external/100/p10004.pdf

**Map Coloring, Two Colors, and the Power of Simple Algorithms**

The mathematical world was stunned in 1976 when the famous **Four Color Map Theorem** was finally proven—but only with the assistance of a computer. The theorem states that any map can be colored using a maximum of four colors such that no two adjacent regions share the same hue.

It's a deceptively simple idea, and it leads us to an even simpler, yet fundamentally important, question: can we color a map (or more abstractly, a graph) with **just two colors**?

This is the essence of the **Bicoloring Problem**.

**What Does "Bicolorable" Even Mean?**

In graph theory, a map's regions and boundaries are represented by a **graph**, where regions are **nodes** (or vertices) and shared boundaries are **edges**.

A graph is "bicolorable" (or more formally, **bipartite**) if you can assign one of two colors—let's call them Color 1 and Color 2—to every node such that *every* edge connects a Color 1 node to a Color 2 node. In plain English: **no two adjacent nodes can have the same color.**

Think of it as dividing all the nodes into two perfectly balanced teams: Team Red and Team Blue. If everyone on Team Red is only connected to people on Team Blue, and vice-versa, the graph is bicolorable.

**The Key to Bicoloring: Odd Cycles**

How do we determine if a graph can be bicolored?

The secret lies in identifying **odd-length cycles**. A cycle is a path that starts and ends at the same node.

- **Even Cycles (e.g., a square):** A cycle with 4 nodes can easily be colored 1-2-1-2. Bicolorable!
- **Odd Cycles (e.g., a triangle):** A cycle with 3 nodes forces a contradiction. If Node A is Color 1, Node B must be Color 2. Node C is connected to both A and B. It must be Color 2 (to satisfy A), but it must also be Color 1 (to satisfy B). It's impossible!

**The Rule:** A graph is bicolorable **if and only if it contains no odd-length cycles.**

**The Algorithmic Solution: BFS to the Rescue**

The C++ code we're using solves this problem efficiently using **Breadth-First Search (BFS)**. BFS is perfect for this because it explores the graph layer by layer, which naturally enforces the alternating color pattern.

**How the Code Works Step-by-Step**

1. **Setup:** We use an array (color) to track the color of each node (0 = uncolored, 1 = Color 1, 2 = Color 2). We also use a queue for the BFS traversal.
2. **Start Coloring:** We pick an arbitrary starting node (usually Node 0, since the graph is guaranteed to be connected) and assign it Color 1 (color[start_node] = 1). We then push it onto the queue.
3. **The Traversal Loop:** While the queue isn't empty, we pull a node (u) out.
4. **Coloring Neighbors:** We look at all of u's neighbors (v). We determine the necessary color for them using a trick: next_color = 3 - color[u]. (If u is 1, next_color is 2; if u is 2, next_color is 1.)
    - **Case 1: Uncolored Neighbor:** If v is uncolored (color[v] == 0), we assign it the next_color and add it to the queue to be processed later. This continues the alternating pattern.
    - **Case 2: Conflict:** If v is *already* colored, we must check its color. If color[v] is the **same** as color[u], we have found two adjacent nodes with the same color! This is a contradiction, meaning we found an odd cycle, and the graph is **NOT BICOLORABLE**. The program immediately stops and reports the result.
5. **Success:** If the entire graph is traversed without finding any conflicts, the graph is successfully divided into two color sets and is **BICOLORABLE**.

This elegant use of BFS allows us to prove or disprove the bipartiteness of the graph in a highly efficient manner, even for graphs with up to 200 nodes!

- **Solution Code:**
  **https://github.com/angkitasaha/Algorithm/blob/main/BFS/Bicoloring/Bicoloring.cpp**

- **Problem2:**
  **Question:**Risk (Shortest Path) — Risk problem in Onlinejudge
  **Link:** https://onlinejudge.org/external/5/p567.pdf

**Conquering the World Efficiently: A Risk Strategy Guide**

The game of Risk is about global domination, but often, the most crucial tactical move is getting a large army massed from a starting country to a destination country. But which path is the fastest?

The goal isn't just to reach the destination; it's to do it by conquering the minimum number of intervening countries. This strategic problem, ripped straight from a board game, is a classic example of the **Shortest Path Problem** in computer science.

**The Conquest as a Graph**

To solve this problem efficiently, we first translate the Risk board into the language of graph theory:

1. **Nodes (Vertices):** Each of the 20 countries on the map becomes a **node** in our graph.
2. **Edges (Connections):** Every shared border between two countries becomes an **edge** connecting those two nodes. Since moving armies can go in either direction, this is an *undirected* graph.

3. **The Goal:** We need to find the shortest path between the starting country (Node A) and the destination country (Node B). Since every conquest counts as one "step," and all edges are equal, this is an **unweighted shortest path** problem.
   **The Ideal Algorithm: Breadth-First Search (BFS)**
   For finding the shortest path in an unweighted graph, there is no better tool than **Breadth-First Search (BFS)**.
   BFS works by systematically exploring the graph layer by layer, guaranteeing that the first time it reaches the destination node, it will have found the path with the fewest edges.
   **BFS Logic for Conquest:**

1. **Start:** Begin at the starting country (Source). This is Distance 0.
2. **Layer 1:** Find all immediate neighbors (countries bordering the Source). These are reachable in 1 conquest.
3. **Layer 2:** Find all countries that border the countries in Layer 1, but haven't been visited yet. These are reachable in 2 conquests.
4. **Destination Found:** The process continues until the Destination country is reached. The layer number it belongs to is the minimum number of conquests required.
   **Handling the Input and Output Format**
   The challenge in implementing this solution lies in two parts: parsing the unusual input format and formatting the output strictly.
   **1. Parsing the Adjacency List**
   The input describes the 20 countries' borders concisely, only listing the connection between country $I$ and country $J$ when $I < J$.
   This requires careful loop management. We read the connections for countries 1 through 19. If country $I$ is connected to country $J$, we must ensure our adjacency list registers the border in *both* directions (I to J, and J to I) to properly represent the undirected nature of the board.
   **2. The Final Calculation**
   The BFS algorithm typically returns the shortest distance (the number of edges). In the context of this problem:

- If the start and end countries are neighbors, the distance (edges) is 1. The number of conquests is 1 (the destination).
- If the start and end countries are $k$ edges apart, the number of countries *conquered* is $k$.

   Therefore, the distance returned by BFS directly corresponds to the minimum number of countries that must be conquered.
   By combining the robustness of BFS with careful input parsing and output formatting, we can determine the most efficient conquest route across the Risk world map for any given starting and ending position. Good luck conquering!
   The following C++ code implements this solution, reading multiple test sets until EOF.

   ➢ **Solution Code:**
   **https://github.com/angkitasaha/Algorithm/blob/main/BFS/Risk/Risk.cpp**

- **Problem3:**

  **Question:** Knight moves (BFS in 2d Grid) — Knight moves problem in Onlinejudge

  **Link:** https://onlinejudge.org/external/4/p439.pdf

**The Shortest Knight's Tour: Solving the Mini-TKP**

My friend recently dove into the **Traveling Knight Problem (TKP)**, a famous puzzle where you try to find a minimal closed route that visits every square on a board exactly once. He thought the hardest part was calculating the shortest distance between any two squares. As any graph theorist knows, the hardest part is the *tour* itself, but I gladly took on his "difficult" challenge.

Finding the minimum number of knight moves between two squares is a classic, satisfying problem that introduces the power of one of computer science's most reliable algorithms: **Breadth-First Search (BFS)**.

**Modeling the Chessboard as a Graph**

Just like the countries in Risk, the chessboard can be modeled as an unweighted graph:

1. **Nodes (Vertices):** The 64 squares on the board (a1 through h8).
2. **Edges (Connections):** An edge exists between any two squares if a knight can move directly from one to the other in a single leap. Since all moves are equivalent, this is an **unweighted graph**.
3. **The Goal:** Find the shortest path—the minimum number of edges—between the start node and the destination node.

**Why BFS is Perfect for the Knight's Leap**

Since every move costs exactly "1" (it's unweighted), BFS is the optimal tool. It explores the board in concentric rings of increasing distance, guaranteeing that the first time we discover the destination square, we have found the shortest route.

**The BFS Logic on the Chessboard:**

1. **Start:** Place the starting square into a queue and set its distance to 0.
2. **Explore Layer N:** Dequeue a square and generate all 8 possible *valid* knight moves from that position.
3. **Assign Distance:** For any generated square that has **not yet been visited**, mark its distance as *Distance N + 1* and enqueue it.
4. **Find the Destination:** The moment we generate the destination square, its assigned distance is the minimum number of moves required. The search terminates immediately.

**Converting Coordinates**

The primary practical hurdle is translating the human-friendly algebraic notation (e2, h8) into the numerical coordinates (e.g., array indices 0-7) that an algorithm can process.

- **Column (a-h):** The letter (a to h) maps to the x-coordinate (0 to 7). For example, 'a' is 0, 'e' is 4, etc. This is done by simply subtracting the ASCII value of 'a'.
- **Row (1-8):** The number (1 to 8) maps to the y-coordinate (0 to 7). We often use a slight inversion (e.g., 8 maps to 0 if we assume (0,0) is top-left, or 1 maps to 0 if we assume (0,0) is bottom-left). Consistency is key.

  The following C++ program efficiently handles both the coordinate conversion and the BFS shortest path search, allowing us to quickly solve any jump across the 64 squares.

➢ **Solution Code:**

**https://github.com/angkitasaha/Algorithm/blob/main/BFS/Knight%20moves/Knight%20moves.cpp**

# Dijkstra

---

## ➢ What is Dijkstra?

Dijkstra's Algorithm is a shortest path algorithm used to find the minimum distance from a starting node (source) to all other nodes in a graph **with non-negative edge weights**.

It is widely used in maps, GPS navigation, routing, networking, and many real-world applications.

The main idea is: always expand the **closest unvisited node**, update the cost of its neighbors, and repeat.

## ➢ Why learn Dijkstra?

- It is the foundation of many shortest-path problems.
- Used in real-life applications like GPS route planning.
- Helps understand priority queues and greedy algorithms.
- Essential for solving many competitive programming and computer science problems.
- Works efficiently for large graphs.
- Demonstrates the greedy strategy in practice.

## ➢ What you should know first

- Basics of graphs (nodes, edges, weights).
- Adjacency list or adjacency matrix representation.
- Priority queue / min-heap concept.
- Arrays, loops, functions, and basic algorithmic thinking.

## ➢ Key Idea

- You start at the source with distance **0**.
- All other nodes begin with distance **infinity**.
- Always pick the **closest unvisited node**.
- Update the distances of all its neighbors.
- If a new path is shorter, replace the old distance.

- Once a node is chosen as the closest, its distance becomes **final**.
- Repeat until all nodes are processed or the queue becomes empty.

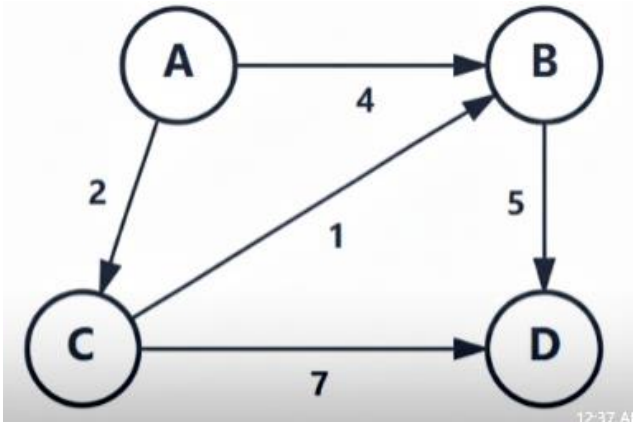
## ➢ How Dijkstra Works

- Start with all distances = infinity, except the source (0).
- Pick the unvisited node with the smallest known distance.
- For each neighbor, update the `distance = current distance + edge weight` if smaller.
- Mark the current node as visited.
- Repeat until all nodes are processed.

## ➢ 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]
} }
```

➢ **Diagram**




Dijkstra's Algorithm from node A:

Initial:

Distance: A=0, B=∞, C=∞, D=∞

Unvisited: {A, B, C, D}

Step 1: Visit A (distance=0)

Update: B=4, C=2

Unvisited: {B, C, D}

Step 2: Visit C (distance=2)

Update: B=min(4, 2+1)=3, D=9

Unvisited: {B, D}

Step 3: Visit B (distance=3)

Update: D=min(9, 3+5)=8

Unvisited: {D}

Step 4: Visit D (distance=8)

Done!

Shortest distances from A:

A→A: 0

A→C: 2

A→B: 3 (via C)

A→D: 8 (via C→B)

Shortest paths:

A→B: A→C→B (total: 3)

A→C: A→C (total: 2)

A→D: A→C→B→D (total: 8)

- **Time & Space Complexity**

Using Priority Queue (recommended)

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

Using simple array (slower)

- **Time:** $O(V^2)$

- **After learning Dijkstra**

Dijkstra's algorithm helps you find the shortest path from one point to all other points in a graph. It works by always picking the closest unvisited point, updating the distances to its neighbors, and then moving on. You keep doing this until all points are checked. It's useful for things like GPS directions or finding the fastest route in a network.

- **Implementation:**
  https://github.com/angkitasaha/Algorithm/blob/main/Dijkstra/Dijkstra%20Basic/Implementation2

- **Problem1:**

**Question:** Dijkstra? — Dijkstra? Problem from Codeforces

Link: https://codeforces.com/contest/20/problem/C

# The Shortest Route: Why Dijkstra's Algorithm is Your Map for Weighted Graphs

We all navigate weighted graphs every day, whether we realize it or not. When you use a GPS app, the roads are the edges, the intersections are the vertices, and the **time or distance** is the weight. Our goal, in programming and in life, is often to find the fastest way from point A (vertex 1) to point B (vertex N).

The problem we just solved—finding the shortest path in a weighted, undirected graph—is a cornerstone of computer science. And the most elegant and efficient way to conquer it is with **Dijkstra's Algorithm**.

## Why Can't We Just Use Simple Search?

In an unweighted graph, you could use a simple Breadth-First Search (BFS) to find the shortest path in terms of the number of steps. But here, the graph is *weighted*.

Imagine two paths from the start (S) to the end (E):

1. **Path 1:** S → A → E (2 steps total)
    a. S to A = 100
    b. A to E = 1
    **c. Total Weight: 101**
2. **Path 2:** S → B → C → E (3 steps total)
    a. S to B = 2
    b. B to C = 2
    c. C to E = 2
    **d. Total Weight: 6**

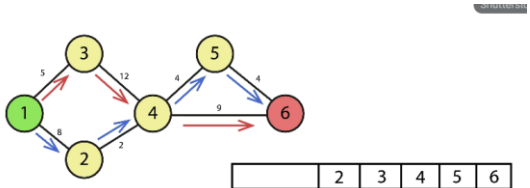
BFS would find Path 1 first because it has fewer steps. But Path 2 is clearly the **shortest** route by weight. This is where Dijkstra's ingenious, greedy approach steps in.

# Dijkstra's Algorithm: The Greedy Navigator

Dijkstra's algorithm works on the principle of local optimality leading to global optimality. It maintains a set of "finalized" vertices (those whose shortest distance from the source is known) and systematically explores their neighbors.

Here's the core idea:

1. **Start at the Source (Vertex 1):** We set its distance to 0 and all others to infinity (`INF`).
2. **Greedy Choice:** We always select the unvisited vertex that currently has the *smallest known distance* from the source.
3. **Relaxation:** For that chosen vertex (let's call it (U), we look at all its neighbors (V). If traveling to V through U results in a shorter path than V current recorded distance, we **update (relax)** V's distance.
4. **Repeat:** We repeat steps 2 and 3 until the destination vertex (N) is included in the finalized set.




| | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| {1} | 8 | 5 | ∞ | ∞ | ∞ |
| {1,3} | 8 | | 17 | ∞ | ∞ |
| {1,3,2} | | | 10 | ∞ | ∞ |
| {1,3,2,4} | | | | 14 | 19 |
| {1,3,2,4,5} | | | | | 18 |

Dijkstra's Algorithm

This process guarantees that when we select a vertex U, we have found the absolute shortest path to it, because any other path would have required passing through a different, lower-distance vertex that would have been selected earlier.

# Deconstructing the C++ Implementation

The efficiency of Dijkstra's algorithm for large graphs (up to 10^5 vertices and edges in our case) comes down to using the right data structures, as seen in the `shortest_path.cpp` file:

1. **`adj` (Adjacency List):** This structure efficiently stores the graph. Instead of a large matrix, we only store the neighbors for each vertex, making iteration quick. We store `pair<long long, int>` (weight, neighbor) for easy access to both pieces of information.
2. **`dist` (Distance Array):** The heart of the algorithm. `dist[i]` holds the shortest known distance from vertex 1 to vertex i. It's initialized to `INF`.
3. **`pq` (Priority Queue - Min-Heap):** This is the speed booster. It allows us to retrieve the unvisited vertex with the minimum distance in O(log V) time. Without the priority queue, we'd have to scan the entire `dist` array in O(V) time in every step, making the algorithm too slow for large inputs.
4. **`parent` (Parent Array):** This array is critical for the output. Every time we relax a distance to a vertex V via vertex U, we set `parent[V] = U`. This maintains a record of the path we took.

## Reconstructing the Shortest Path

Once the main loop finishes, we know the total shortest distance (`dist[n]`). To print the actual path (e.g., `1 4 3 5` from the example), we simply backtrack:

1. Start at the destination vertex, `curr = n`.
2. Look up `parent[curr]`. This is the vertex that came *before* it.
3. Repeat this process until `curr` is the starting vertex (1).
4. Since this generates the path in reverse order (N to dots to 1), we reverse the resulting vector to get the final output.

If `dist[n]` is still `INF`, it means the destination is unreachable, and we correctly output `-1`. Dijkstra's algorithm provides a powerful, reliable, and efficient method for solving the real-world challenge of finding the shortest path, whether you're routing data packets or planning a road trip.

Let me know if you'd like to dive deeper into how the priority queue works, or if you'd like to trace the algorithm with the sample input!

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

- Problem2:

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

# The Scenic Route: How to Find the Second-Best Shortest Path in a Weighted Graph

We've all been there: relying on GPS only to realize the "fastest" route is just a boring highway. Sometimes, like Robin in our problem, you want the next best thing—a path that's still efficient but avoids the absolute minimum distance.

The problem, often called the K-Shortest Path problem (where K=2), challenges us to find a route from a starting intersection (1) to a destination (N) that is **strictly longer** than the shortest path, but shorter than every other possible route.

This requirement—being *strictly longer* than the absolute minimum—means we can't just run the standard shortest path algorithm and call it a day. We need to upgrade our navigation system.

## Why Standard Dijkstra Fails

Dijkstra's algorithm is the gold standard for finding the shortest path (d1). It's a greedy algorithm that works by guaranteeing that the first time it "finalizes" a vertex, it has found the best possible route to it.

However, standard Dijkstra only keeps track of one distance: the current minimum.

Consider a simple graph:

- Path A: Length 10 (Shortest, d1)
- Path B: Length 10 (Also Shortest, equal to d1)
- Path C: Length 12 (The true second shortest, d2)

If we only track d1, we miss Path C. Since the problem demands the second-best path be **strictly longer** than d1, we must introduce a new rule.

# The Modified Dijkstra: Tracking Two Distances

The solution, as implemented in the `second_shortest_path.cpp` file, is to slightly alter Dijkstra's core logic. Instead of a single distance array, we use two:

1. **`d1[i]`:** The length of the absolute shortest path from 1 to intersection i.
2. **`d2[i]`:** The length of the second shortest path from 1 to intersection i (must be d2[i] > d1[i]).

Our priority queue still stores `{distance, vertex}`, but now, when we extract a vertex, that distance might be either a d1 or a d2 value. This is the key: we allow the priority queue to explore paths that are slightly longer.

## The Crucial Relaxation Logic

The real magic happens in the **relaxation step**, where we check every neighbor V of the current vertex U. Let current_dist be the distance to U and new_dist be the path length through U to V (`current_dist + weight`).

We have two scenarios for updating the distances at vertex V:

### *Scenario 1: Found a New Absolute Shortest Path (`new_dist < d1[v]`)*

If the new_dist is better than the current absolute shortest path to V, it means we've found a breakthrough!

- The old `d1[v]` must now become the new `d2[v]`.
- The new_dist becomes the new `d1[v]`.

### *Scenario 2: Found a New Second Shortest Path (`d1[v] < new_dist < d2[v]`)*

If the new_dist is better than d2[v] but *worse* than d1[v], we've found our scenic route!

- The new_dist replaces `d2[v]`.

## Why This Works

By using a priority queue and constantly updating **both** d1 and d2 for every vertex, we ensure that:

1. When we finish the algorithm, d1[N] holds the shortest path.
2. d2[N] holds the shortest path that is *strictly greater* than d1[N].

   This simple but powerful modification allows the algorithm to explore those slightly longer "detours" that might ultimately lead to the desired second-shortest path to the destination. It elegantly solves Robin's dilemma, ensuring he gets the best possible scenic route!

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

# Bellman Ford

---

## ➢ Introduction

The **Bellman-Ford Algorithm** is a graph algorithm used to find the **shortest paths from a single source vertex to all other vertices** in a weighted graph. Unlike Dijkstra's algorithm, it can handle **graphs with negative weight edges**.

## ➢ Key Points

- Works on directed and weighted graphs.
- Can detect negative weight cycles.
- Based on dynamic programming.
- Relaxes all edges V - 1 times.
- If still an update is possible on the V-th iteration → **negative cycle exists**.
- Finds shortest path from one source to all nodes.
- Unreachable nodes remain **infinity (∞)**.
- Works even if the graph contains cycles.

## ➢ Why Learn Bellman-Ford?

- Works with negative weighted edges.
- Detects negative cycles (something Dijkstra cannot do).
- Easy to understand and implement.
- Used in networking (Distance Vector Routing).
- Useful for competitive programming and academic purposes.

## ➢ How Bellman-Ford Works

**1.** Set all distances to **infinity**, except source = **0**.

**2.** Repeat **V - 1 times**:

- Try to *relax* every edge.
- If a shorter path is found, update it.

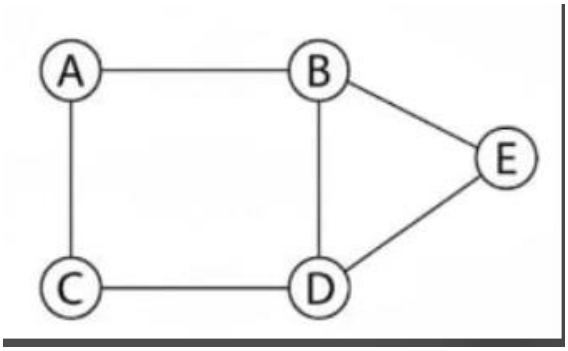
**3.** Run one more iteration:

- If any edge still relaxes → **negative cycle detected**.

## ➢ 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) ∈ G.E do
    if d[v] > d[u] + w(u, v) then
       return FALSE
 return TRUE
```

- **Diagram**




Bellman-Ford Algorithm from node A:
Assuming all edges have weight 1 (since no weights shown):
Initial:

Distance: A=0, B=∞, C=∞, D=∞, E=∞

Iteration 1:

A→B: dist[B] = min(∞, 0+1) = 1
A→C: dist[C] = min(∞, 0+1) = 1
B→E: dist[E] = min(∞, 1+1) = 2
B→D: dist[D] = min(∞, 1+1) = 2
C→D: dist[D] = min(2, 1+1) = 2
D→E: dist[E] = min(2, 2+1) = 2
After Iteration 1: A=0, B=1, C=1, D=2, E=2

Iteration 2:
A→B: dist[B] = min(1, 0+1) = 1
A→C: dist[C] = min(1, 0+1) = 1
B→E: dist[E] = min(2, 1+1) = 2
B→D: dist[D] = min(2, 1+1) = 2
C→D: dist[D] = min(2, 1+1) = 2
D→E: dist[E] = min(2, 2+1) = 2
After Iteration 2: A=0, B=1, C=1, D=2, E=2
Iteration 3 & 4:
No changes (distances stabilized)
Final shortest distances from A:
A→A: 0
A→B: 1
A→C: 1
A→D: 2
A→E: 2

No negative cycles detected!

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

- **After Learning Bellman-Ford**

  Bellman-Ford is a shortest path algorithm that works even with **negative weights**. It relaxes all edges again and again to find the minimum distances. After V−1 rounds, it checks one more time to detect **negative cycles**. It's simple, safe, and detects problems in the graph, but slower than Dijkstra.

- **Implementation:**
  **https://github.com/angkitasaha/Algorithm/blob/main/Bellman-Ford/Bellman-Ford%20Basic/Implementation3**

- **Problem1:**

  **Question:** UVA558 — Warmholes problem in Onlinejudge
  Link: https://onlinejudge.org/external/5/p558.pdf

# Time Travel is Possible (If You Can Find a Negative Cycle)

The year is 2163. The future isn't about flying cars; it's about wormholes that can send you forward or backward in time. For a brilliant physicist on Earth (star system 0), the ultimate goal isn't just to see the future, but to witness the Big Bang itself.

This incredible sci-fi premise is actually a classic computer science problem: **Can we find a negative cycle in a weighted, directed graph?**

## The Time Travel Math

Let's translate the physics into graph theory:

- **Star Systems (**N**):** These are the vertices (nodes) in our graph.
- **Wormholes (**M**):** These are the directed edges. Since they are one-way, the path from X to Y is distinct from Y to X.
- **Time Difference (**t**):** This is the weight of the edge.
    - **Positive weight:** Travel into the future (e.g., +15 years).
    - **Negative weight:** Travel into the past (e.g., -42 years).

To reach the Big Bang, the physicist needs a path that allows for *indefinite* travel into the past. This happens if they can find a loop—a cycle—where the sum of the time differences is negative. Once found, they can traverse this negative cycle over and over, reducing their accumulated time indefinitely.

## Why Your Standard GPS (Dijkstra) Fails

If all the wormholes sent you into the future (all positive weights), the familiar and highly efficient Dijkstra's algorithm would be the perfect tool to find the path that minimizes the total time jump.

But as soon as you introduce a negative weight—a trip into the past—Dijkstra's breaks down. Its core logic is "greedy," assuming that once a vertex is finalized, we've found the shortest path to it. A negative edge can suddenly make an already finalized path shorter, invalidating Dijkstra's fundamental assumption.

We need an algorithm built to handle negativity.

## The Hero: The Bellman-Ford Algorithm

Enter the Bellman-Ford algorithm. While slower than Dijkstra's (running in O(V c dot E) time, where V is vertices and E is edges), it is robust and guarantees correct results even with negative weights.

The Bellman-Ford algorithm works in two distinct phases:

### Phase 1: The N-1 Relaxations

The algorithm iterates through **all** edges N-1 times (where N is the number of star systems). In each iteration, it attempts to "relax" the distance (time) to every vertex.

After N-1 iterations, if there are no negative cycles, we are guaranteed to have found the absolute minimum time required to reach every reachable star system from Earth.

### Phase 2: Detecting the Negative Cycle

This is where the magic happens. We perform one final, N^th iteration over all the edges.

If, during this N^th pass, we find *any* edge that can still be relaxed (meaning `dist[u] + t < dist[v]`), it proves that a negative cycle exists and that this cycle is reachable from the starting system (Earth/0).

Why does a successful relaxation on the N^th iteration prove a negative cycle? Because if no negative cycle were present, the minimum paths would have been stable after the N-1 iterations. The only way a path can continue to get shorter after N-1 steps is if you've found a loop where the time keeps decreasing.

## The Verdict

For the physicist, the final output is simple:

- If the N^th iteration reveals a distance that can be shortened, the output is **'possible'**. The physicist has found her time machine.
- If no further reductions can be made, the wormhole network is safe from paradoxes, and the output is **'not possible'**.

- **Solution Code:** https://github.com/angkitasaha/Algorithm/blob/main/Bellman-Ford/UVA558/UVA558.cpp

- **Problem2:**

  **Question:** LOJ1108 — Traffic problem in Onlinejudge

  Link: https://onlinejudge.org/external/104/p10449.pdf

# Navigating Dhaka's Chaos: Solving the Shortest Path Problem with a Cubed Cost

Dhaka, like many megacities, grapples with perennial traffic congestion. The city authority's response to this chaos isn't just about building new roads; it's about shifting commuter behavior by introducing a unique financial incentive system. This system presents a compelling real-world challenge that requires a powerful, specialized algorithm to solve: the Shortest Path Faster Algorithm (SPFA).

Our task, as appointed by the city authority, is to calculate the minimum total "earning" collected from a traveler moving from a central point (Junction 1) to any other point in the city.

## The Rulebook of the Road: Graph Modeling

At its core, the Dhaka City problem is a classic **shortest path problem** in graph theory.

- **Nodes (Junctions):** Each city junction is a node in the graph, numbered 1 to N (up to 200).
- **Edges (Roads):** The unidirectional roads connecting the junctions are the edges.
- **The Crux: The Busyness Score:** Every junction has a "busyness" score (an integer from 1 to 20).

The cost to traverse a single road from a source junction S to a destination junction D is not fixed, nor is it linear. It is determined by the formula:

Cost = (Busyness_D - Busyness_S^3)

This cost represents the "earning" the city collects from the traveler. Our goal is to find the path that minimizes the sum of these costs.

## Why Standard Algorithms Fail

If all costs in a graph are positive, we would typically use **Dijkstra's Algorithm** for the shortest path. However, the cubed cost function introduces a critical complexity: **negative edge weights**.

Imagine a traveler moving:

1. From a highly busy junction (Busyness=20) to a quiet junction (Busyness=5). The cost is (5 - 20)^3 = (-15)^3 = -3375. This is a massive negative cost.
2. From a quiet junction (Busyness=5) to a highly busy junction (Busyness=20). The cost is (20 - 5)^3 = (15)^3 = 3375. This is a large positive cost.

When a graph contains negative weights, Dijkstra's algorithm breaks down. Since the problem guarantees that the minimum path exists (implying no negative cycles), we must turn to an algorithm designed for this scenario, such as the **Shortest Path Faster Algorithm (SPFA)**, an improvement on the Bellman-Ford algorithm.

## The SPFA Solution

The SPFA algorithm works similarly to BFS but can handle negative weights by allowing nodes to be re-relaxed if a shorter path is found.

1. **Initialization:** We start at Junction 1, setting its minimum earning (`dist`) to 0 and all other nodes to infinity (`INF`).
2. **Queue Management:** We use a queue to store junctions that have been recently updated, as their change in minimum earning might lead to a shorter path for their neighbors.
3. **Relaxation:** When a junction U is dequeued, we check all neighbors V. If the total earning to reach V through U is less than the current recorded minimum earning for V, we update V's minimum earning and re-enqueue V.

This iterative process continues until no further improvements can be found, guaranteeing we have the minimum total earning for every reachable junction.

## The Authority's Unique Constraint

The problem introduces one final, peculiar condition related to the output:

"However, for the queries that gives less than 3 units of minimum total earning, print a ‘?’."

This means that even if a path yields a minimum total earning of 0, 1, or 2, the city authority imposes a failure condition, suggesting these short-haul trips are economically unviable or undesirable.

- **Solution Code:**
  https://github.com/angkitasaha/Algorithm/blob/main/Bellman-Ford/LOJ1108/LOJ1108.cpp