<https://codeforces.com/blog/entry/22276>

[Link To PDF version (Latex Formatted)](https://www.dropbox.com/s/sgzwlmsmx5rrrug/0-1%20BFS.pdf?dl=0)

**Topic :** 0-1 BFS

**Pre Requisites :** Basics of Graph Theory , BFS , Shortest Path

**Problem :**

You have a **graph G** with **V vertices** and **E edges**. The graph is a weighted graph but the weights have a contraint that they can only be 0 or 1. Write an efficient code to calculate shortest path from a given source.

**Solution :**

**Naive Solution — Dijkstra's Algorithm.**

This has a complexity of O(E + VlogV) in its best implementation. You might try heuristics , but the worst case remains the same. At this point you maybe thinking about how you could optimise Dijkstra or why do I write such an efficient algorithm as the naive solution? Ok , so firstly the efficient solution isn't an optimisation of Dijkstra. Secondly , this is provided as the naive solution because almost everyone would code this up the first time they see such a question , assuming they know Dijkstra's algorithm.

Supposing Dijkstra's algorithm is your best code forward , I would like to present to you a very simple yet elegant trick to solve a question on this type of graph using Breadth First Search (BFS).

Before we dive into the algorithm, a lemma is required to get things crystal clear later on.

**Lemma :** "During the execution of BFS, the queue holding the vertices only contains elements from at max two successive levels of the BFS tree."

**Explanation :** The BFS tree is the tree built during the execution of BFS on any graph. This lemma is true since at every point in the execution of BFS , we only traverse to the adjacent vertices of a vertex and thus every vertex in the queue is at max one level away from all other vertices in the queue.

So let's get started with 0-1 BFS.

**0-1 BFS :**

This is so named , since it works on graphs with edge weights 0 and 1. Let's take a point of execution of BFS when you are at an arbitrary vertex "u" having edges with weight 0 and 1. Similar to Dijkstra , we only put a vertex in the queue if it has been relaxed by a previous vertex (distance is reduced by travelling on this edge) and we also keep the queue sorted by distance from source at every point of time.

Now , when we are at "u" , we know one thing for sure : Travelling an edge (u,v) would make sure that v is either in the same level as u or at the next successive level. This is because the edge weights are 0 and 1. An edge weight of 0 would mean that they lie on the same level , whereas an edge weight of 1 means they lie on the level below. We also know that during BFS our queue holds vertices of two successive levels at max. So, when we are at vertex "u" , our queue contains elements of level L[u] or L[u] + 1. And we also know that for an edge (u,v) , L[v] is either L[u] or L[u] + 1. Thus , if the vertex "v" is relaxed and has the same level , we can push it to the front of our queue and if it has the very next level , we can push it to the end of the queue. This helps us keep the queue sorted by level for the BFS to work properly.

But, using a normal queue data structure , we cannot insert and keep it sorted in O(1). Using priority queue cost us O(logN) to keep it sorted. The problem with the normal queue is the absence of methods which helps us to perform all of these functions :

Remove Top Element (To get vertex for BFS)

Insert At the beginning (To push a vertex with same level)

Insert At the end (To push a vertex on next level)

Fortunately, all of these operations are supported by a double ended queue (or deque in C++ STL). Let's have a look at pseudocode for this trick :

for all v in vertices:

dist[v] = inf

dist[source] = 0;

deque d

d.push\_front(source)

while d.empty() == false:

vertex = get front element and pop as in BFS.

for all edges e of form (vertex, u):

if travelling e relaxes distance to u:

relax dist[u]

if e.weight = 1:

d.push\_back(u)

else:

d.push\_front(u)

As you can see , this is quite similar to BFS + Dijkstra. But the time complexity of this code is O(E + V) , which is linear and more efficient than Dijkstra. The analysis and proof of correctness is also same as that of BFS.

Before moving into solving problems from online judges , try these exercises to make sure you completely understand why and how 0-1 BFS works :

Can we apply the same trick if our edge weights can only be 0 and x (x >= 0) ?

Can we apply the same trick if our edge weights are x and x+1 (x >= 0) ?

Can we apply the same trick if our edge weights are x and y (x,y >= 0) ?

This trick is actually quite a simple trick, but not many people know this. Here are some problems you can try this hack at :

<http://www.spoj.com/problems/KATHTHI/> — [My implementation](http://ideone.com/SttVq6)

<https://community.topcoder.com/stat?c=problem_statement&pm=10337>

Problem J of Gym

Div1 — 500 on topcoder are tough to crack. So congrats on being able to solve one of them using such a simple trick :). I will add more problems as I find.

**Relation between Dijkstra and BFS ?**

**Refer to chatGPT**

Dijkstra's algorithm and BFS (Breadth-First Search) are both graph traversal algorithms but are used for different purposes and have distinct characteristics. Here's a detailed comparison and explanation of their relationship:

**Key Characteristics**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Dijkstra's Algorithm** | **BFS** |
| **Purpose** | Finds the shortest path in a **weighted** graph. | Explores all nodes at the same depth level in an **unweighted** graph. |
| **Graph Type** | Works on weighted graphs (non-negative weights). | Works on unweighted graphs or with uniform edge weights. |
| **Edge Costs** | Considers edge weights to calculate shortest paths. | Assumes all edges have equal cost (1 or uniform). |
| **Data Structure** | Uses a priority queue (min-heap) for selecting the next node with the smallest tentative distance. | Uses a simple queue for level-wise traversal. |
| **Time Complexity** | O((V+E)log⁡V) with a binary heap. | O(V+E), where V is vertices and  E is edges. |
| **Output** | Returns the shortest path distances or paths. | Returns the set of reachable nodes in level order. |

**Relationship Between Dijkstra's Algorithm and BFS**

Special Case of Dijkstra:

BFS can be seen as a special case of Dijkstra's algorithm where all edge weights are equal (e.g., 1).

In BFS, the shortest path in terms of the number of edges from the source to all reachable nodes is computed, which is equivalent to Dijkstra with uniform weights.

Priority Mechanism:

Dijkstra uses a **priority queue** to always process the node with the smallest tentative distance (lowest cost).

BFS uses a **FIFO queue**, which effectively works as a priority queue with equal priorities.

Edge Weights:

BFS doesn't account for weights, as it assumes all edges contribute equally to the path length.

Dijkstra accommodates varying edge weights and selects paths based on cumulative cost.

**Analogies**

Unweighted Graphs:

If you run Dijkstra on an unweighted graph, it behaves exactly like BFS since all edges have the same weight, and the shortest path in terms of edges equals the shortest path in terms of cost.

0-1 BFS:

In graphs where edge weights are restricted to 0 and 1, **0-1 BFS** can be used. It combines the principles of BFS and Dijkstra by using a deque (double-ended queue) to prioritize 0-cost edges over 1-cost edges. This is conceptually a hybrid of the two algorithms.

**Example: Weighted vs. Unweighted Graph**

**Unweighted Graph (BFS):**

Graph: 1 -- 2 -- 3 -- 4

Edge Weights: All edges = 1

Source: 1

BFS traverses: 1→2→3→4

Shortest path from 1→4: 3 edges.

**Weighted Graph (Dijkstra):**

Graph: 1 --(1)--> 2 --(3)--> 3 --(1)--> 4

Edge Weights: As shown above

Source: 1

Dijkstra considers edge weights: 1+1+3.

Shortest path from 1→4: 4 units.

**Summary of Relation**

BFS is a simplified version of Dijkstra that applies when edge weights are uniform (equal).

Dijkstra generalizes BFS to handle non-uniform weights by using a priority queue.

Both are fundamental algorithms, but Dijkstra's algorithm is more versatile due to its ability to handle weighted graphs.

**Relation between Dijkstra and 0 - 1 BFS ?**

**Refer to chatGPT**

Dijkstra's algorithm and **0-1 BFS** are related approaches used for finding the shortest path in a graph, but they are optimized for different scenarios and use distinct mechanisms. Here's a detailed comparison and explanation of the relationship between them:

**Dijkstra’s Algorithm**

General Usage: Used to find the shortest path in a weighted graph where edge weights can be any non-negative values.

Key Mechanism:

It uses a **priority queue (min-heap)** to process nodes in the order of their shortest known distance.

Each node is processed at most once at its shortest distance due to the priority queue ordering.

Edge Weight Handling:

Works for arbitrary non-negative weights (e.g., weights 1, 3, 10, etc.).

Time Complexity:

O((V+E)log⁡V), where V is the number of vertices and E is the number of edges.

This complexity comes from the priority queue operations and edge relaxation.

Data Structure: Priority queue is crucial for determining the next node to process.

**0-1 BFS**

Specific Usage: Optimized for graphs where edge weights are only **0 or 1**.

Key Mechanism:

Uses a **deque (double-ended queue)** instead of a priority queue.

Edges with weight 0 are processed by pushing nodes to the **front** of the deque.

Edges with weight 1 are processed by pushing nodes to the **back** of the deque.

Edge Weight Handling:

Specialized for edge weights of 0 and 1. This allows the algorithm to avoid the overhead of a priority queue while maintaining the correct order of processing.

Time Complexity:

O(V+E), because each edge is relaxed once and deque operations are O(1).

Data Structure: Deque is used to handle edges of weight 0 and 1 efficiently.

**Relationship Between Dijkstra and 0-1 BFS**

Both Solve Single-Source Shortest Path Problems:

Both algorithms aim to find the shortest path from a source node to all other nodes in the graph.

**0-1 BFS is a Special Case of Dijkstra:**

**When all edge weights are 0 or 1, Dijkstra's priority queue can be replaced with a deque.**

**The deque mimics the behavior of a priority queue by processing 0-weight edges (higher priority) before 1-weight edges (lower priority).**

This reduces the complexity from O((V+E) log⁡V).

Priority Queue vs. Deque:

In Dijkstra, the priority queue is needed to order nodes by distance dynamically for arbitrary weights.

In 0-1 BFS, the deque achieves this ordering implicitly since weights are binary.

Efficiency:

0-1 BFS is more efficient than Dijkstra when the graph has only 0 and 1 weights because it avoids the overhead of maintaining a priority queue.

**Example: Compare Dijkstra and 0-1 BFS**

**Graph Example**

Graph edges:

0 → 1 (weight 0)

1 → 2 (weight 1)

0 → 2 (weight 1)

**0-1 BFS Process**

Start at node 0. Initialize dist[0] = 0 and dist[i] = ∞ for other nodes.

Add 0 to the deque.

Pop 0: Explore edges:

Edge 0 → 1 (weight 0): Push 1 to the **front** → dist[1] = 0.

Edge 0 → 2 (weight 1): Push 2 to the **back** → dist[2] = 1.

Pop 1: Explore edges:

Edge 1 → 2 (weight 1): No update since dist[2] = 1 is already minimal.

Pop 2: No further edges to explore.

Result: dist = [0, 0, 1].

**Dijkstra’s Process**

Start at node 0. Initialize dist[0] = 0 and dist[i] = ∞ for other nodes. Use a priority queue:

Pop (0, 0) (node 0, distance 0): Explore edges:

Edge 0 → 1 (weight 0): Update dist[1] = 0, push (1, 0) to the queue.

Edge 0 → 2 (weight 1): Update dist[2] = 1, push (2, 1) to the queue.

Pop (1, 0) (node 1, distance 0): Explore edges:

Edge 1 → 2 (weight 1): No update since dist[2] = 1 is already minimal.

Pop (2, 1) (node 2, distance 1): No further edges to explore.

Result: dist = [0, 0, 1].

**Key Takeaway**

For graphs with only 0 and 1 weights, **0-1 BFS** is essentially a more efficient version of Dijkstra because it leverages the restricted edge weights to simplify processing while maintaining correctness. For general edge weights, Dijkstra remains the go-to algorithm.