Dijkstra's Algorithm - GeeksforGeeks

Source: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Dijkstra's Algorithm Last Updated : 21 Jan, 2026 Given a weighted undirected graph and a source vertex src . We need to find the shortest path distances from the source vertex to all other vertices in the graph. Note: The given graph does not contain any negative edge. Examples: Input: src = 0, 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: Shortest Paths: 0 -> 0 = 0: Source node itself, so distance is 0. 0 -> 1 = 4: Direct edge from node 0 to 1 gives shortest distance 4. 0 -> 2 = 7: Path $0 \rightarrow 1 \rightarrow 2$ gives total cost 4 + 3 = 7, which is smaller than direct edge 8. 0 -> 3 = 9: Path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ gives total cost 4 + 3 + 2 = 9. 0 -> 4 = 10: Path $0 \rightarrow 1 \rightarrow 4$ gives total cost 4 + 6 = 10. Try it on GfG Practice The idea is to maintain distance using an array dist[] from the given source to all vertices. The distance array is initialized as infinite for all vertexes and 0 for the given source We also maintain two sets, One set contains vertices included in the shortest-path tree, The other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source. Once we pick a vertex, we update the distance of its adjacent if we get a shorter path through it. Use of priority Queue The priority queue always selects the node with the smallest current distance, ensuring that we explore the shortest paths first and avoid unnecessary processing of longer paths. Dijkstra’s algorithm always picks the node with the minimum distance first. By doing so, it ensures that the node has already checked the shortest distance to all its neighbors. If this node appears again in the priority queue later, we don’t need to process it again, because its neighbors have already checked the minimum possible distances Detailed Steps: Create a distance array dist[] of size V and initialize all values to infinity (∞) since no paths are known yet. Set the distance of the source vertex to 0 and insert it into the priority queue. While the priority queue is not empty, remove the vertex with the smallest distance value. Check: if the popped distance is greater than the recorded distance for this vertex(dist[u]), it means this vertex has already been processed with a smaller distance, so skip it and continue to the next iteration. For each neighbor v of u, check if the path through u gives a smaller distance than the current dist[v]. If it does, update dist[v] = dist[u] + edge weight(d) and push (dist[v], v) into the priority queue. Continue this process until the priority queue becomes empty. Once done, the dist[] array will contain the shortest distance from the source to every vertex in the graph.\ C++ //Driver Code Starts #include <iostream> #include <vector> #include <queue> #include <climits> using namespace std ; //Driver Code Ends vector < int > dijkstra (vector < vector < pair < int , int >>>& adj , int src) { int V = adj . size (); // Min-heap (priority queue) storing pairs of (distance, node) priority_queue < pair < int , int > , vector < pair < int , int >> , greater < pair < int , int >>> pq ; vector < int > dist (V , INT_MAX); // Distance from source to itself is 0 dist [src] = 0 ; pq . emplace (0 , src); // Process the queue until all reachable vertices are finalized while (! pq . empty ()) { auto top = pq . top (); pq . pop (); int d = top . first ; int u = top . second ; // If this distance not the latest shortest one, skip it if (d > dist [u]) continue ; // Explore all neighbors of the current vertex for (auto & p : adj [u]) { int v = p . first ; int w = p . second ; // If we found a shorter path to v through u, update it if (dist [u] + w < dist [v]) { dist [v] = dist [u] + w ; pq . emplace (dist [v], v); } } } // Return the final shortest distances from the source return dist ; } //Driver Code Starts int main () { int src = 0 ; vector < vector < pair < int , int >>> adj (5); adj [0] = {{ 1 , 4 }, { 2 , 8 }}; adj [1] = {{ 0 , 4 }, { 4 , 6 }, { 2 , 3 }}; adj [2] = {{ 0 , 8 }, { 3 , 2 }, { 1 , 3 }}; adj [3] = {{ 2 , 2 }, { 4 , 10 }}; adj [4] = {{ 1 , 6 }, { 3 , 10 }}; vector < int > result = dijkstra (adj , src); for (int d : result) cout << d << " " ; cout << " " ; return 0 ; } //Driver Code

Ends Java //Driver Code Starts import java.util.ArrayList ; import java.util.Arrays ; import java.util.PriorityQueue ; class GFG { //Driver Code Ends static ArrayList < Integer > dijkstra (ArrayList < ArrayList < int []>> adj , int src) { int V = adj . size (); // Min-heap (priority queue) storing pairs of (distance, node) PriorityQueue < int []> pq = new PriorityQueue <> ((a , b) -> a [0] - b [0]); // Distance array: stores shortest distance from source int [] dist = new int [V] ; Arrays . fill (dist , Integer . MAX_VALUE); // Distance from source to itself is 0 dist [src] = 0 ; pq . offer (new int [] { 0 , src }); // Process the queue until all reachable vertices are finalized while (! pq . isEmpty ()) { int [] top = pq . poll (); int d = top [0] ; int u = top [1] ; // If this distance is not the latest shortest one, skip it if (d > dist [u]) continue ; // Explore all adjacent vertices for (int [] p : adj . get (u)) { int v = p [0] ; int w = p [1] ; // If we found a shorter path to v through u, update it if (dist [u] + w < dist [v]) { dist [v] = dist [u] + w ; pq . offer (new int [] { dist [v] , v }); } } } ArrayList < Integer > result = new ArrayList <> (); for (int d : dist) result . add (d); // Return the final shortest distances from the source return result ; } //Driver Code Starts static void addEdge (ArrayList < ArrayList < int []>> adj , int u , int v , int w) { adj . get (u). add (new int [] { v , w }); adj . get (v). add (new int [] { u , w }); } public static void main (String [] args) { int V = 5 ; int src = 0 ; ArrayList < ArrayList < int []>> adj = new ArrayList <> (); for (int i = 0 ; i < V ; i ++) { adj . add (new ArrayList <> ()); } addEdge (adj , 0 , 1 , 4); addEdge (adj , 0 , 2 , 8); addEdge (adj , 1 , 4 , 6); addEdge (adj , 1 , 2 , 3); addEdge (adj , 2 , 3 , 2); addEdge (adj , 3 , 4 , 10); ArrayList < Integer > result = dijkstra (adj , src); for (int d : result) System . out . print (d + " "); System . out . println (); } } //Driver Code Ends Python #Driver Code Starts import heapq import sys #Driver Code Ends def dijkstra (adj , src): V = len (adj) # Min-heap (priority queue) storing pairs of (distance, node) pq = [] dist = [sys . maxsize] * V # Distance from source to itself is 0 dist [src] = 0 heapq . heappush (pq , (0 , src)) # Process the queue until all reachable vertices are finalized while pq : d , u = heapq . heappop (pq) # If this distance not the latest shortest one, skip it if d > dist [u]: continue # Explore all neighbors of the current vertex for v , w in adj [u]: # If we found a shorter path to v through u, update it if dist [u] + w < dist [v]: dist [v] = dist [u] + w heapq . heappush (pq , (dist [v], v)) # Return the final shortest distances from the source return dist #Driver Code Starts if __name__ == "__main__" : src = 0 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)]] result = dijkstra (adj , src) print (* result) #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; class MinHeap { private List < int [] > heap ; public MinHeap () { heap = new List < int [] > (); } public int Count => heap . Count ; // Enqueue (node, distance) public void Enqueue (int node , int distance) { heap . Add (new int [] { distance , node }); SiftUp (heap . Count - 1); } // Dequeue the smallest distance pair public bool TryDequeue (out int node , out int distance) { if (heap . Count == 0) { node = - 1 ; distance = int . MaxValue ; return false ; } int [] top = heap [0]; distance = top [0]; node = top [1]; int last = heap . Count - 1 ; heap [0] = heap [last]; heap . RemoveAt (last); if (heap . Count > 0) SiftDown (0); return true ; } private void SiftUp (int i) { while (i > 0) { int parent = (i - 1) / 2 ; if (heap [parent][0] <= heap [i][0]) break ; Swap (i , parent); i = parent ; } } private void SiftDown (int i) { int n = heap . Count ; while (true) { int left = 2 * i + 1 ; int right = 2 * i + 2 ; int smallest = i ; if (left < n && heap [left][0] < heap [smallest][0]) smallest = left ; if (right < n && heap [right][0] < heap [smallest][0]) smallest = right ; if (smallest == i) break ; Swap (i , smallest); i = smallest ; } } private void Swap (int i , int j) { int [] temp = heap [i]; heap [i] = heap [j]; heap [j] = temp ; } } class GFG { //Driver Code Ends static List < int > dijkstra (List < List < int [] >> adj , int src) { int V = adj . Count ; // Min-heap (priority queue) storing pairs of (distance, node) MinHeap pq = new MinHeap (); // Distance array: stores shortest distance from source int [] dist = new int [V]; for (int i = 0 ; i < V ; i ++) dist [i] = int . MaxValue ; // Distance from source to itself is 0 dist [src] = 0 ; pq . Enqueue (src , 0); // Process the queue until all reachable vertices are finalized while (pq . Count > 0) { pq . TryDequeue (out int u , out int d); // If this distance is not the latest shortest one, skip it if (d > dist [u]) continue ; // Explore all adjacent vertices foreach (var p in adj [u]) { int v = p [0]; int w = p [1]; // If we found a shorter path to v through u, update it if (dist [u] + w < dist [v]) { dist [v] = dist [u] + w ; pq . Enqueue (v , dist [v]); } } } // Convert result to List for output List < int > result = new List < int > (); foreach (int d in dist) result . Add (d); // Return the final shortest distances from the source return result ; } //Driver Code Starts static void addEdge (List < List < int [] >> adj , int u , int v , int w) { adj [u]. Add (new int [] { v , w }); adj [v]. Add (new int [] { u , w }); } static void Main () { int V = 5 ; int src = 0 ; List < List < int [] >> adj = new List < List < int [] >> (); for (int i = 0 ; i < V ; i ++) adj . Add (new List < int [] > ()); addEdge (adj , 0 , 1 , 4); addEdge (adj , 0 , 2 , 8); addEdge (adj , 1 , 2 , 3); addEdge (adj , 1 , 4 , 6); addEdge (adj , 2 , 3 , 2); addEdge (adj , 3 , 4 , 10); List < int > result = dijkstra (adj , src); foreach (int d in result) Console . Write (d + " "); Console . WriteLine (); } } //Driver Code Ends JavaScript //Driver Code Starts class MinHeap { constructor () { this . heap = []; } push (

item) { this . heap . push (item); this . _bubbleUp (); } pop () { if (this . heap . length === 1) return this . heap . pop (); const top = this . heap [0]; this . heap [0] = this . heap . pop (); this . _bubbleDown (); return top ; } _bubbleUp () { let i = this . heap . length - 1 ; while (i > 0) { let p = Math . floor ((i - 1) / 2); if (this . heap [p][0] <= this . heap [i][0]) break ; [this . heap [p], this . heap [i]] = [this . heap [i], this . heap [p]]; i = p ; } } _bubbleDown () { let i = 0 ; const n = this . heap . length ; while (true) { let l = 2 * i + 1 , r = 2 * i + 2 , smallest = i ; if (l < n && this . heap [l][0] < this . heap [smallest][0]) smallest = l ; if (r < n && this . heap [r][0] < this . heap [smallest][0]) smallest = r ; if (smallest === i) break ; [this . heap [i], this . heap [smallest]] = [this . heap [smallest], this . heap [i]]; i = smallest ; } } isEmpty () { return this . heap . length === 0 ; } } //Driver Code Ends function dijkstra (adj , src) { let V = adj . length ; // Min-heap (priority queue) storing pairs of (distance, node) let pq = new MinHeap (); let dist = Array (V). fill (Number . MAX_SAFE_INTEGER); // Distance from source to itself is 0 dist [src] = 0 ; pq . push ([0 , src]); // Process the queue until all reachable vertices are finalized while (! pq . isEmpty ()) { let [d , u] = pq . pop (); // If this distance not the latest shortest one, skip it if (d > dist [u]) continue ; // Explore all neighbors of the current vertex for (let [v , w] of adj [u]) { // If we found a shorter path to v through u, update it if (dist [u] + w < dist [v]) { dist [v] = dist [u] + w ; pq . push ([dist [v], v]); } } } // Return the final shortest distances from the source return dist ; } //Driver Code Starts // Driver Code let src = 0 ; let 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]]]; let result = dijkstra (adj , src); console . log (result . join (" ")); //Driver Code Ends Output 0 4 7 9 10 Time Complexity: O((V+E)*logV), Where E is the number of edges and V is the number of vertices. Auxiliary Space: O(V+E) How Does it Work? Once we pop the minimum distance item from the priority queue, we finalize its shortest distance and never consider it again. Assuming that there are no negative weight edges, if there were a shorter path to this node through any other route, that path would have to go through nodes with equal or greater than current distance, and so wouldn’t be shorter Why Does Not Work with Negative Weights: Dijkstra’s algorithm assumes that once a vertex u is picked from the priority queue (meaning it currently has the smallest distance), its shortest distance is finalized - it will never change in the future. This assumption is true only if all edge weights are non-negative. Suppose there exists an edge with a negative weight. After Dijkstra finalizes a vertex u, there might exist another path through some vertex v (processed later) that leads back to u with a smaller total distance because of the negative edge. So, the algorithm’s earlier assumption that dist[u] was final is no longer valid. For common questions, refer to this article: Common Questions on Dijkstra . Problems based on Shortest Paths Shortest Path in Directed Acyclic Graph Minimum Cost Path Print negative weight cycle in a Directed Graph Number of ways to reach at destination in shortest time Snake and Ladder Problem Word Ladder Comment Article Tags: Article Tags: Graph Greedy DSA Amazon Adobe Morgan Stanley Dijkstra Samsung Cisco Accolite Vizury Interactive Solutions Shortest Path + 8 More