Bellman-ford algorithm example in computer networks

Given the graph and vertex src source in the graph, look for the shortest route from the src to all the vertices in the given graph. Graphs may contain negative weight edges. We have discussed Dijkstra's algorithm for this issue. The Dijkstra algorithm is a greedy algorithm and the complexity of time is O(VLogV) (with the use of Fibonacci heaps). Dijkstra does not work for Graf with negative weight edges, Bellman-Ford serves for the graph. Bellman-Ford is also easier than Dijkstra and suites well for theagih system. But the complexity of Bellman-Ford's time is O(VE), which is more so than Dijkstra. The following algorithm is a detailed step. Input: Graph and vertex source src Output: The shortest distance to all verticals of the src. If there is a negative weight cycle, then the shortest distance is not calculated, a negative weight cycle is reported. 1) This step starts the distance from the source to all vertik as infinite and the distance to the source itself as 0. Create multiple chests[] sizes **|** I'm not going to do that with all values as infinite except dist[src] where src is the source vertex. 2) This step calculates the shortest distance. Are there any of the following **|** V|-1 times where **|** I'm not going to do that is the vertic number in the given graph. ..... a) Do the following for every excess u-v .... If dist[v] &gt; dist[u] + uv edge weight, then remote update[v] .... dada[v] = chest[u] + uv edge weight 3) This step reports if there is a negative weight cycle in the graph. Does it follow any advantages of u-v ...... If dist[v] &gt; dist[u] + uv edge weight, then Graf contains a negative weight cycle Idea step 3 is, step 2 guarantees the shortest distance if the graph does not contain a negative weight cycle. If we go through all the edges one more time and get a shorter path for any vertex, then there is a negative weight cycle How does this work? Like other Dynamic Programming Problems, algorithms calculate the shortest path in a lower way. It first calculates the shortest distance it has on an advantage on the route. Then, he thinks the shortest route with at-most 2 edges, and so on. After the outer coil i-th, the shortest route with most i edges is calculated. There can be a maximum **|** I'm not going to do that - 1 edge in any easy road, which is why the outer coil runs |v| – 1 time. The idea is, in the event that there is no negative weight cycle, if we have calculated the shortest path with most i edges, then lesteration over all edges guarantees to provide the shortest path with the most (i+1) edges (Proof is easy, you can refer to this or MIT Video Example Let us understand the algorithm with the following example Images taken from this source. Let the given source vertex be 0. Control all distances as infinity, except the distance to the source itself. The number of vertices in the graph is 5, so all edges must be processed 4 times. Let all edges be processed inside command: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distance when all edges are processed the first time. The first line shows the starting distance. The second row shows the distance when the edges (B, E), (D, B), (B, D) and (A, B) are processed. The third line indicates the distance when (A, C) is processed. The fourth line indicates when (D, C), (B, C) and (E, D) are processed. The first iteration guarantees to provide all the shortest lines that are on most 1 long edges. We get the following distance when all edges are processed the second time (The last line indicates the final value). The second schedule guarantees to provide all the shortest lines that are on most 2 long edges. The algorithm processes all edges 2 more times. The distance is minimized after the second schedule, so the third and fourth iterations do not update the distance. Implementation: Struct Graph* creates graph (int V, int E) struct graph * = New graph; graph-&gt;edge = New edge[E]; invalid printArr (int dist[], int n) printf (Vertex Distance from Source); for (int i = 0; i &lt; n; ++i) printf(%d \t\t%d, i, dist[i]); cancel BellmanFord (graph struct* , int src) for (int i = 0; i &lt; V; i++) to (int i = 1; i &lt;= V - 1; i++) { for (int j = 0; j &lt; E; j ++) { int u = graph-&gt;edge[j.src]; int v = graph-&gt;edge[j].dest; int weight = graph-&gt;edge[j].weight; if (chest[u] != INT_MAX &amp;amp; chest[u] + weight &lt; dist[v]) dist[v] = chest[u] + weight; for (int i = 0; i &lt; E; i++) { int u = graf-&gt;edge[i].src; int v = graph-&gt;edge[i].dest; weight = &gt;edge[i].weight; if (chest[u] != INT_MAX &amp;amp; chest[u] + weight &lt; dist[v]) { printf(Graf contains negative weight cycle); struct Graph* graph = createGraph(V, E); graph-&gt;edge[0].weight = -1; graph-&gt;edge[1].weight = 4; graph-&gt;edge[2].weight = 3; graf-&gt;edge[3].weight = 2; graph-&gt;edge[4].weight = 2; graph-&gt;edge[5].weight = 5; graf-&gt;edge[6].weight = 1; graf-&gt;edge[7].weight = -3; Java to (int i = 0; i &lt; e; ++i) invalid BellmanFord (Graph, int src) int V = graph. V, E = graph. E; for (int i = 0; i &lt; V; ++i) chest[i] = Integer.MAX_VALUE; for (int i = 1; i &lt; V; ++i) { for (int j = 0; j &lt; E; ++j) { int u = graf.edge[j].src; int v = graf.edge[j].dest; int weight = graph.edge[j].weight; if (chest[u] != Integer.MAX_VALUE &amp;amp; chest[u] + weight &lt; chest[v] = chest[u] + weight; for (int j = 0; j &lt; E; ++j) { int u = graph.edge[j].src; int v = graf.edge[j].dest; int weight = graph.edge[j].weight; if (chest[u] != Integer.MAX_VALUE &amp;amp; chest[u] + weight &lt; dist[v]) { System.out.println(Graf contains negative weight cycles); invalid printArr (int dist[], int V) System.out.println (Vertex Distance from Source); for (int i = 0; i &lt; V; ++i) System.out.println(i + \t\t + dist[i]); main invalid layman (String[] args) Graph = New graph (V, E); graf.edge[0].weight = -1; graf.edge[1].weight = 4; graf.edge[2].weight = 3; graf.edge[3].weight = 2; graf.edge[4].weight = 2; graf.edge[5].weight = 5; graf.edge[6].weight = 1; graf.edge[7].weight = -3; Graf. BellmanFord (graph, 0); Python3 __init__ (self, vertical): def addEdge (self, u, v, w): self.graph.append([u, v, w]) def printArr(own, remote): print (Vertex Distance from Source) print ({0}\t\t{1}.format(i, dist[i])) def BellmanFord (itself, src): stay away = [floating(Inf)] * V to _ in range (self. V - 1): for you, v, w in self.graph: if dist[u] != floating (Inf) and chest[u] + w &lt; dist[v]: for you, v, w in self.graph: if dist[u] != float (Inf) and chest[u] + w &lt; dist[v]: print (Graph contains negative weight cycle) C# int awam src, dest, weight; for (int i = 0; i &lt; e; ++i) invalid BellmanFord(Graph, int src) int V = graph. V, E = graph. E; for (int i = 0; i &lt; V; ++i) for (int i = 1; i &lt; V; ++i) { for (int j = 0; j &lt; E; ++j) { int u = graf.edge[j].src; int v = graf.edge[j].dest; int weight = graph.edge[j].weight; if (chest[u] != int. MaxValue &amp;amp; dist[u] + weight &lt; dist[v]) chest[v] = chest[u] + weight; for (int j = 0; j &lt; E; ++j) { int u = graph.edge[j].src; int v = graf.edge[j].dest; int weight = graph.edge[j].weight; if (chest[u] != int. MaxValue &amp;amp; dist[u] + weight &lt; { Console.WriteLine (Graph contains negative weight cycles); invalid printArr (int[] chest, int V) Console.WriteLine (Vertex Distance from Source); to (int i = 0; i &amp; V; ++i) Console.WriteLine(i + \t\t + dist[i]); public static invalid Major() Graph graph = New graph (V, E); graph.edge[0].weight = -1; graph.edge[1].weight = 4; graph.edge[2].weight = 3; graph.edge[3].weight = 2; graph.edge[4].weight = 2; graph.edge[5].weight = 5; graph.edge[6].weight = 1; graph.edge[7].weight = -3; Graph. BellmanFord (graph, 0); Output: Vertex Distance from Source 0 0 0 1 -1 2 2 3 3 -2 4 1 Note 1) Negative weight is found in various graph applications. For example, rather than paying costs for a route, we might get some advantages if we follow the route. 2) Bellman-Ford works better (better than Dijksra' for distributed systems. Unlike Dijkstra where we need to find a minimum value of all verticals, in Bellman-Ford, the edges are considered one by one. Exercise 1) The standard Bellman-Ford algorithm reports the shortest route only if there is no negative weight cycle. Modify it so that it reports a minimum distance even if there is a negative weight cycle. 2) Can we use the Dijkstra algorithm for the shortest route to graph with a negative weight - an idea can, calculate the minimum weight value, add a positive value (equal to the absolute value of minimum weight) to all weights and run the Dijkstra algorithm for modified graphs. Will this algorithm work? Ford Bellman Algorithm Reference (Easy Execution) : E2%80%93Ford_algorithm Please write a comment if you find anything incorrect, or you want to share more information on the topics discussed above. Attention readers! Don't stop learning now. Hold all important DSA concepts with DSA Self-Paced Courses at student-friendly prices and become industry ready. Recommended posts:Improved By : danielwzou, AnkitRai01, thori, gp6, merrcury merrcury merrcury