**Dijkstra’s shortest path algorithm**

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.  
Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/). Like Prim’s MST, we generate a*SPT (shortest path tree)* with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.  
Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

**Algorithm   
1) in the constructor initialize all the fields like noOfVertices, dist, spt. Keep the distance of all the vertices is Integer.*MAX\_VALUE*;**

**2) now in Dijkstra method….**

**a) Distance of source vertex from itself is always 0…so do like dist[src] = 0;**

**b) find the shortest path for all the vertices..so run a loop**

**1) Pick the minimum distance vertex from the set of vertices not yet processed.**

**2) Mark the picked vertex as processed**

**3) Update dist value of the adjacent vertices of the picked vertex.**

**for** (**int** v = 0; v < noOfVertices; v++)

// Update dist[v] only if

// is not in spt

// there is an edge from u to v

// and total weight of path from src to v through u is smaller than current

// value of dist[v]

**if** (!spt[v] && graph[u][v] != 0 && dist[u] != Integer.***MAX\_VALUE*** && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

Let us understand with the following example: 



The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet*becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively). 



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet*includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



***How to implement the above algorithm?***

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/DijkstrasShortestPath.java>

**Dijkstra’s Algorithm for Adjacency List Representation**

We recommend reading the following two posts as a prerequisite of this post.  
The time complexity for the matrix representation is O(V^2). In this post, O(ELogV) algorithm for adjacency list representation is discussed.  
As discussed in the previous post, in Dijkstra’s algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time using [BFS](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/). The idea is to traverse all vertices of graph using [BFS](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet).  Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is O(LogV) for Min Heap.

**Following are the detailed steps.   
1) Create a priority queue of size V where V is the number of vertices in the given graph**

**2) Initialize priority queue with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).   
3) While Min Heap is not empty, do following   
 a) Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.**

**b) For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of u-v plus distance value of u, then update the distance value of v.**



Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.   
The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap



Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.



Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively). 



Pick the vertex with minimum distance from min heap. Vertex 6 is picked. So min heap now contains all vertices except 0, 1, 7 and 6. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



Above steps are repeated till min heap doesn’t become empty. Finally, we get the following shortest path tree.



**Time Complexity:** The time complexity of the above code/algorithm looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)\*O(LogV) which is O((E+V)\*LogV) = O(ELogV)   
Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) time.  
**Notes:**

1. The code calculates shortest distance, but doesn’t calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim’s implementation](https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/)) and use it show the shortest path from source to different vertices.
2. The code is for undirected graph, same dijekstra function can be used for directed graphs also.
3. The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
4. Dijkstra’s algorithm doesn’t work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](http://en.wikipedia.org/wiki/Bellman-Ford_algorithm) can be used

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/DijkstrasAdjacencyListRepresenation.java>

**Bellman–Ford Algorithm**

Given a graph and a source vertex *src*in graph, find shortest paths from *src*to all vertices in the given graph. The graph may contain negative weight edges.   
We have discussed [Dijkstra’s algorithm](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) for this problem. Dijkstra’s algorithm is a Greedy algorithm and time complexity is O((V+E)LogV) (with the use of Fibonacci heap). *Dijkstra doesn’t work for Graphs with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.* 

**Algorithm**   
Following are the detailed steps.  
***Input:* Graph and a source vertex *src*   
*Output****:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

**1)** This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.  
**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.   
…..**a)** Do following for each edge u-v   
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]   
………………….dist[v] = dist[u] + weight of edge uv  
**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v   
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”   
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all 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, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

**Example**   
Let us understand the algorithm with following example graph. Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

Bellman–Ford Algorithm Example Graph 1

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed. 

Bellman–Ford Algorithm Example Graph 2

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values). 

Bellman–Ford Algorithm Example Graph 3

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

**Notes**   
**1)**Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.  
**2)** Bellman-Ford works better (better than Dijkstra’s) for distributed systems. Unlike Dijkstra’s where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.                                                                    
**3)**Bellman-Ford does not work with undirected graph with negative edges as it will declared as negative cycle.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/BellmanFordAlgorithm.java>

**Floyd Warshall Algorithm**

The [Floyd Warshall Algorithm](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.   
**Example:**

**Input:**

graph[][] = { {0, 5, INF, 10},

{INF, 0, 3, INF},

{INF, INF, 0, 1},

{INF, INF, INF, 0} }

which represents the following graph

10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3

Note that the value of graph[i][j] is 0 if i is equal to j

And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

**Output:**

Shortest distance matrix

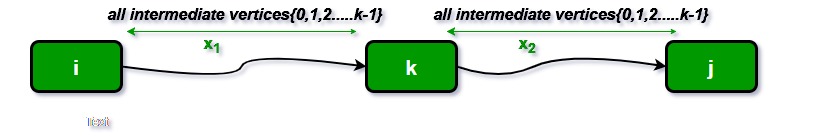
0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**Floyd Warshall Algorithm**   
We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.   
**1)** k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.   
**2)** k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]  
The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



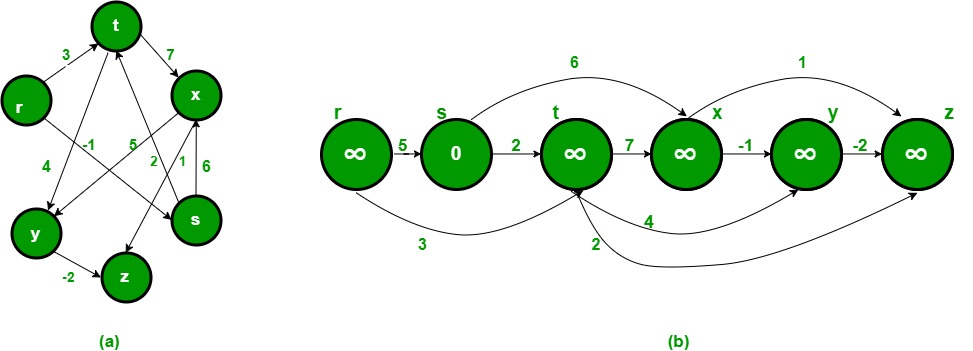
**Time Complexity:**O(V^3)  
The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.   
Also, the value of INF can be taken as INT\_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT\_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

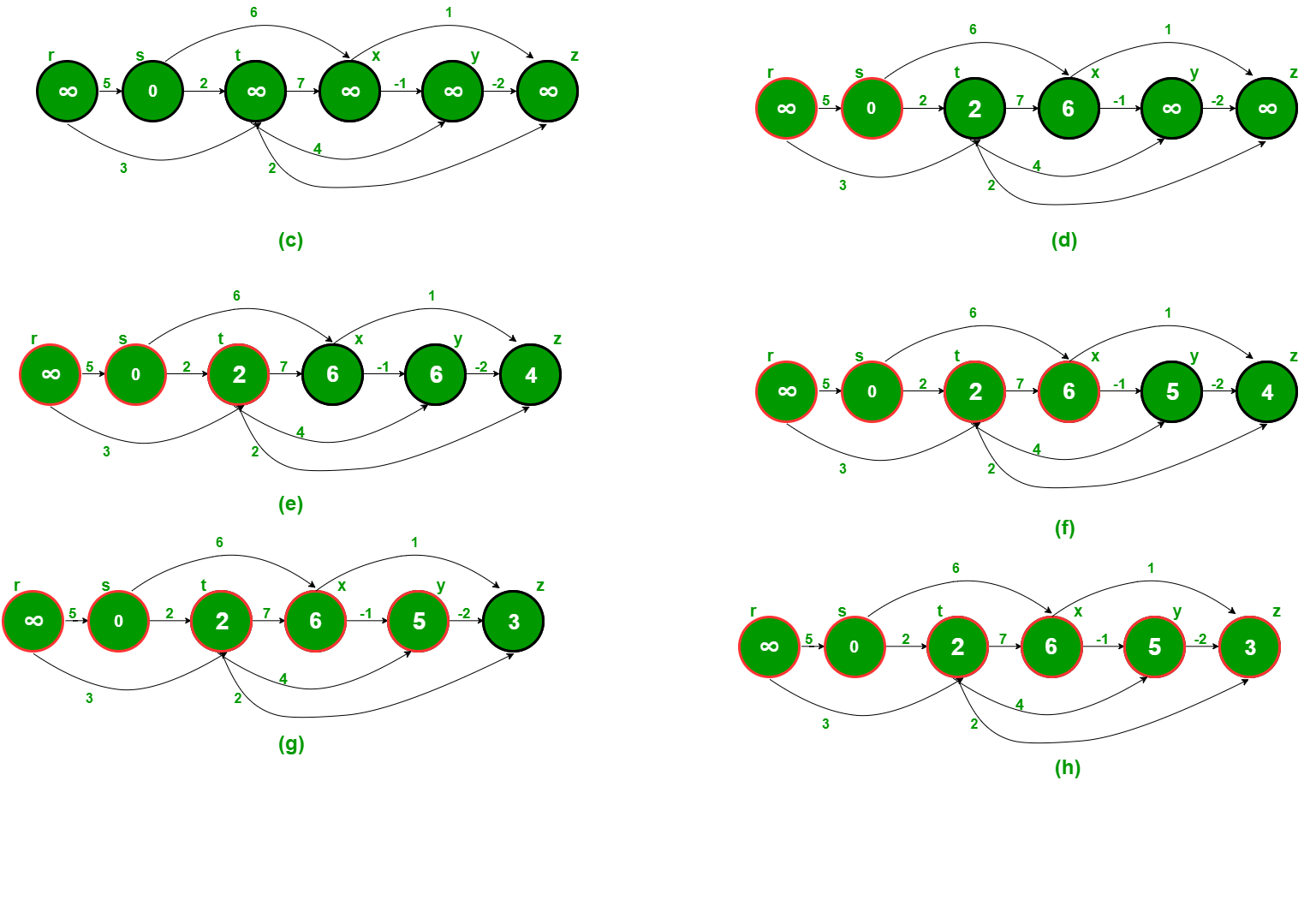
<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/FloyedWarshallAlgorithm.java>

**Shortest Path in Directed Acyclic Graph**

Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.

For a general weighted graph, we can calculate single source shortest distances in O(VE) time using [Bellman–Ford Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/). For a graph with no negative weights, we can do better and calculate single source shortest distances in O(E + VLogV) time using [Dijkstra’s algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/). Can we do even better for Directed Acyclic Graph (DAG)? We can calculate single source shortest distances in O(V+E) time for DAGs. The idea is to use [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/).  
We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/) of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a) ). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.  
Following figure is taken from [this](http://www.utdallas.edu/~sizheng/CS4349.d/l-notes.d/L17.pdf)source. It shows step by step process of finding shortest paths. 





Following is complete algorithm for finding shortest distances.   
**1)** Initialize dist[] = {INF, INF, ….} and dist[s] = 0 where s is the source vertex.   
**2)** Create a topological order of all vertices.   
**3)**Do following for every vertex u in topological order.   
………..Do following for every adjacent vertex v of u   
………………if (dist[v] > dist[u] + weight(u, v))   
………………………dist[v] = dist[u] + weight(u, v)

**Time Complexity:** Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, overall time complexity of this algorithm is O(V+E).

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/ShortestPathInDirectedACyclicGraph.java>

**Printing Paths in Dijkstra’s Shortest Path Algorithm**

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

For example, consider below graph and **source as 0**,

Output should be

Vertex Distance Path

0 -> 1 4 0 1

0 -> 2 12 0 1 2

0 -> 3 19 0 1 2 3

0 -> 4 21 0 7 6 5 4

0 -> 5 11 0 7 6 5

0 -> 6 9 0 7 6

0 -> 7 8 0 7

0 -> 8 14 0 1 2 8

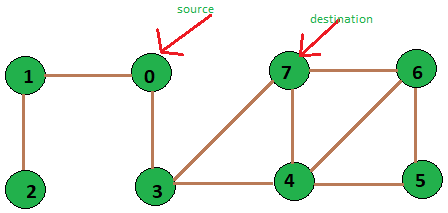
The idea is to create a separate array parent[]. Value of parent[v] for a vertex v stores parent vertex of v in shortest path tree. Parent of root (or source vertex) is -1. Whenever we find shorter path through a vertex u, we make u as parent of current vertex.

Once we have parent array constructed, we can print path using below recursive function.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/PrintAllDijkstrahShortestPath.java>

**Shortest path in an unweighted graph**

Given an unweighted graph, a source, and a destination, we need to find the shortest path from source to destination in the graph in the most optimal way.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/exampleFigure-1.png)

*unweighted graph of 8 vertices*

Input: source vertex = 0 and destination vertex is = 7.

Output: Shortest path length is:2

Path is::

0 3 7

Input: source vertex is = 2 and destination vertex is = 6.

Output: Shortest path length is:5

Path is::

2 1 0 3 4 6

One solution is to solve in O(VE) time using [Bellman–Ford](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/). If there are no negative weight cycles, then we can solve in O(E + VLogV) time using [Dijkstra’s algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/).   
Since the graph is unweighted, we can solve this problem in O(V + E) time. The idea is to use a modified version of [Breadth-first search](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) in which we keep storing the predecessor of a given vertex while doing the breadth-first search.   
We first initialize an array dist[0, 1, …., v-1] such that dist[i] stores the distance of vertex i from the source vertex and array pred[0, 1, ….., v-1] such that pred[i] represents the immediate predecessor of the vertex i in the breadth-first search starting from the source.   
Now we get the length of the path from source to any other vertex in O(1) time from array d, and for printing the path from source to any vertex we can use array p and that will take O(V) time in worst case as V is the size of array P. So most of the time of the algorithm is spent in doing the Breadth-first search from a given source which we know takes O(V+E) time. Thus the time complexity of our algorithm is O(V+E).   
Take the following unweighted graph as an example:

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/ShortestPathInUnweightedGraph.java>

**Find Shortest distance from a guard in a Bank**

Given a matrix that is filled with ‘O’, ‘G’, and ‘W’ where ‘O’ represents open space, ‘G’ represents guards and ‘W’ represents walls in a Bank. Replace all of the O’s in the matrix with their shortest distance from a guard, without being able to go through any walls. Also, replace the guards with 0 and walls with -1 in output matrix.  
Expected **Time complexity** is O(MN) for a M x N matrix.

Examples:

O ==> Open Space

G ==> Guard

W ==> Wall

**Input:**

O O O O G

O W W O O

O O O W O

G W W W O

O O O O G

**Output:**

3 3 2 1 0

2 -1 -1 2 1

1 2 3 -1 2

0 -1 -1 -1 1

1 2 2 1 0

The idea is to do BFS. We first enqueue all cells containing the guards and loop till queue is not empty. For each iteration of the loop, we dequeue the front cell from the queue and for each of its four adjacent cells, if cell is an open area and its distance from guard is not calculated yet, we update its distance and enqueue it. Finally after BFS procedure is over, we print the distance matrix.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/MinDistanceFromaGuardInBank.java>