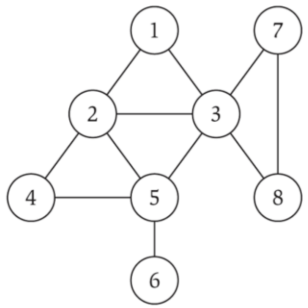
Graph Algorithms 1

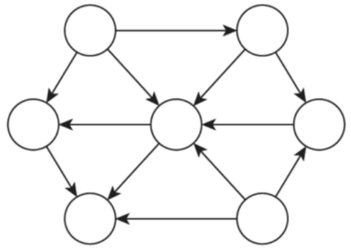
Outline

- DFS
- BFS
- Dijkstra's algorithm
- Bellman-Ford algorithm
- Floyd-Warshall algorithm
- Johnson's algorithm

Graph: Notions



Undirected Graph
V = { 1, 2, 3, 4, 5, 6, 7, 8 }
E = { 1–2, 1–3, 2–3, 2–4, 2–5, 3–5, 3–7, 3–8, 4–5, 5–6, 7–8 }
m = 11, n = 8

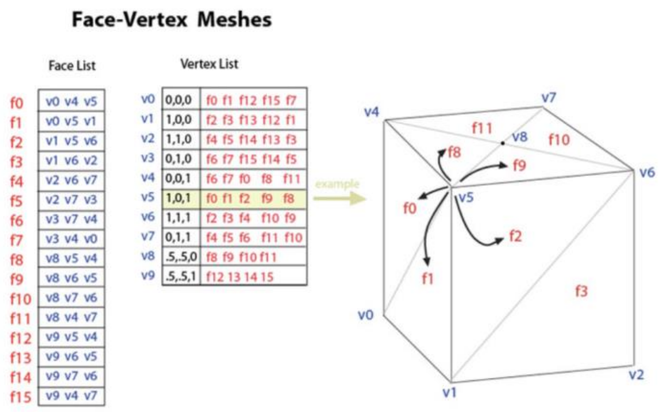


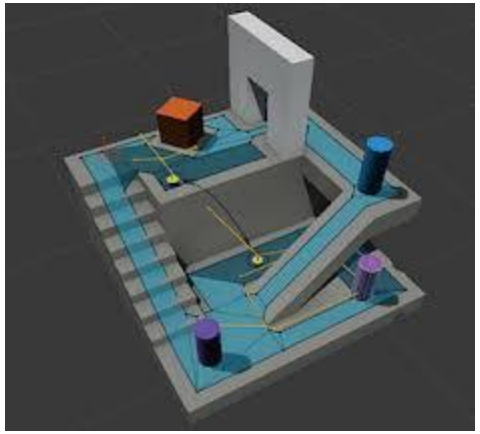
Directed Graph

- **Undirected** Graph: G = (V, E)
  - V = nodes (or vertices)
  - E = edges (or arcs) between pairs of nodes
  - Graph size parameters: n = | V |, m = | E |
- **Directed** Graph: G = (V, E), similar definition
  - Edge (u, v) or u->v leaves node u (head) and enters node v (tail)
- u-v or u->v: u a **neighbor** of v and vice versa, u and v are **adjacent**
- **Degree** of a node is its number of neighbors
  - For directed graph, u: a predecessor of v, v: a successor of u
    - in-degree of a vertex: its number of predecessor
    - out-degree of a vertex: its of successors
- Undirected graph: 0<=m<=$\binom{n}{2}$, directed graph: 0<=m<=n(n-1)
- Graph without loops and parallel edges: **simple** graph
- Non-simple graph: **multigraph**
- **Dense** graph: m is close to n^2, **sparse** graph: m is much less than n^2

Some graph applications

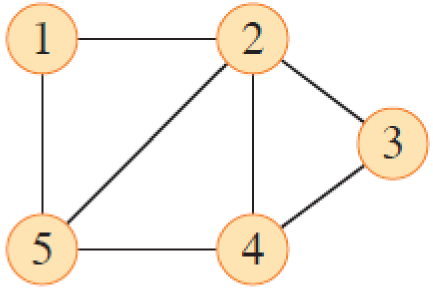
graph	node	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

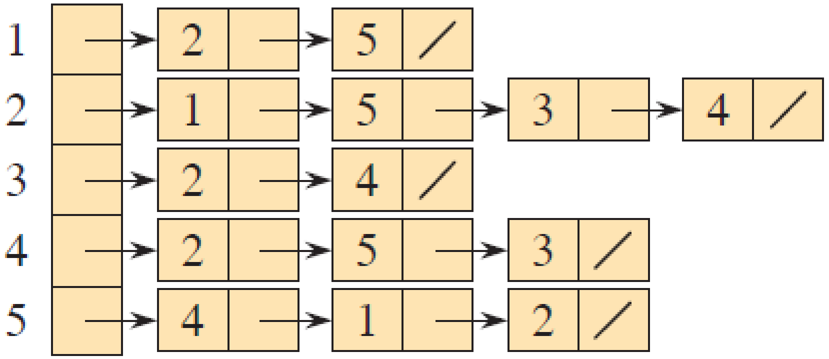


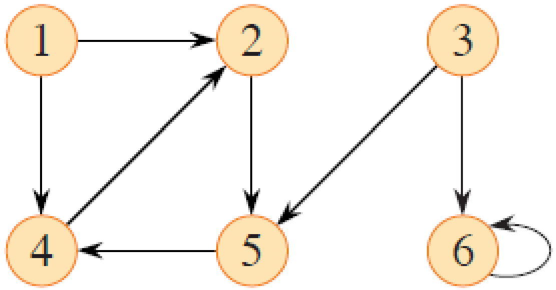


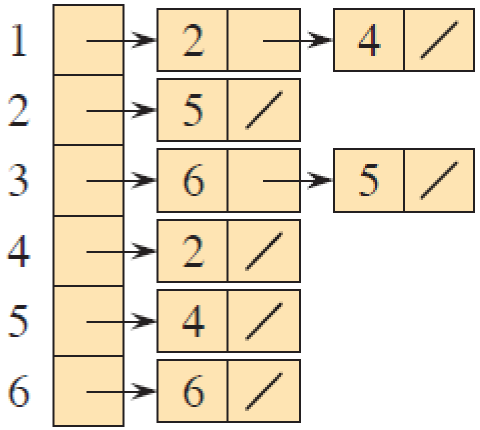
Graph Representations

Adjacency List









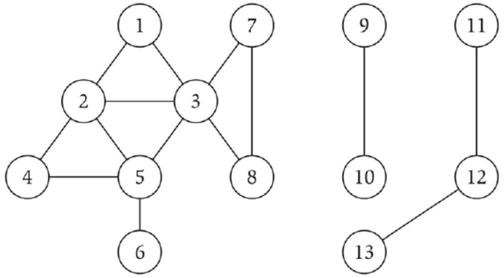
Adjacent Matrix (small and dense)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Paths and connectivity

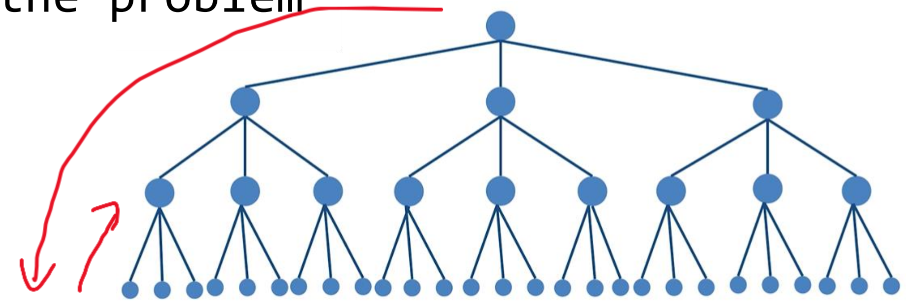
- A path in an undirected graph G = (V, E) is a sequence of nodes $v_1, v_2, v_3, \ldots, v_k$ with the property that each consecutive pair v_{i-1}, v_{i-2} is joined by a (different) edge in E (path is vague, sometimes it allows repetitions)
- A path is simple if all nodes are distinct
- An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v

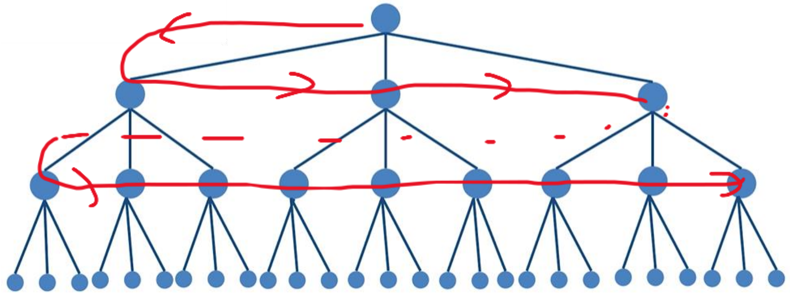


- Common problems:
  - **Connectivity:** Given 2 nodes s and t, is there a path between s and t ?
  - **Shortest path** problem: Given two nodes s and t, what is the length of a shortest path between s and t ?
  - **Reachability/Connected component**: Find all nodes reachable from s.
  - Search, Matching, Max flow, …

DFS and BFS

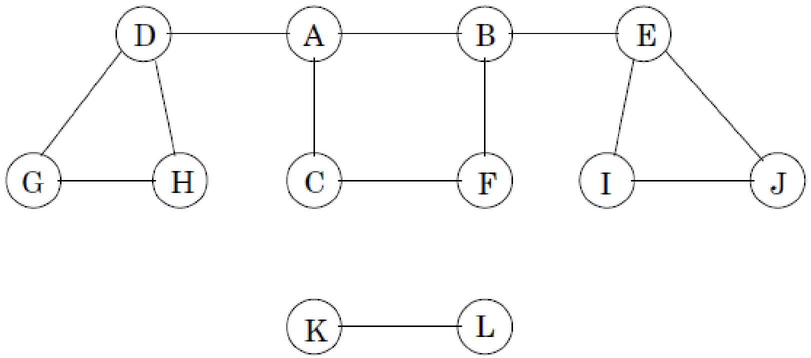
- Enumerate (traversal) + check
- Depth-first search (DFS) and Breath-first search (BFS)
  - DFS: search tree is shallow, less and wide level
  - BFS: search tree is deep, more and narrow level, **min step**
- Similar to DP
  - State: the parameters and values related to the problem
  - State extent: transit the state
- Solution
  - Meet the search goal
  - Solution != Search finished
- Resolve repetitions
  - Usually hash table
  - Depends on the problem
    - Discard
    - Or min/max, counter

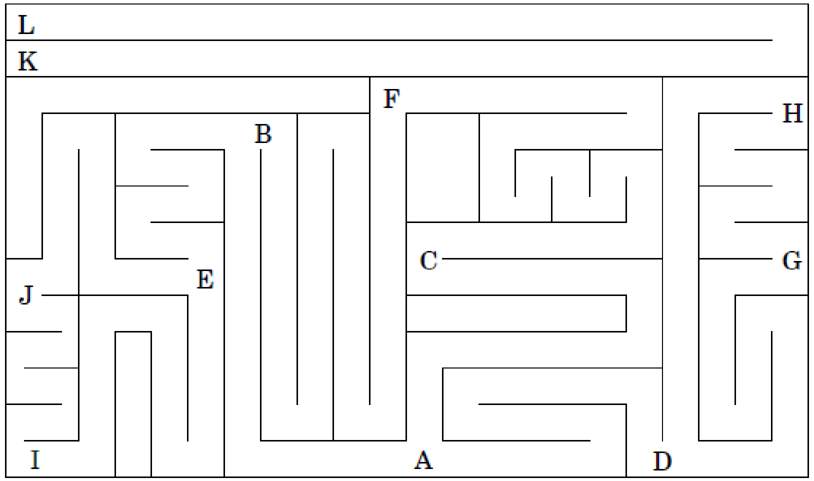




DFS

- Exploring a graph like navigating a maze
- Just one basic operation: getting the neighbors of a vertex





- Method:
  - Explore as far as possible along one branch
  - Reach dead end, backtrack

DFS: Property

- Property
  - Exhaustive search, time complexity: O(n+m) with O(1) to traverse one edge, space complexity: O(h), h is the height
  - All possible permutations == complete graph (every pair of distinct vertices is connected by a unique edge)
  - **Stack** (implicit or explicit): traversal order and current branch from the root/source ~ backtracked/visited nodes can be discarded
  - **Hash Tables:** (marked as) **visited** (to resolve repetition, can store T/F, optimal value, counter, …), or/and **options** (if the current option will affect future options)
  - Pitfalls: infinite loop, collect solution/dead end return, resolve repetitions

```
RecursiveDFS(v):
    if v is unmarked
        mark v
        for each edge vw
            RecursiveDFS(w)
```

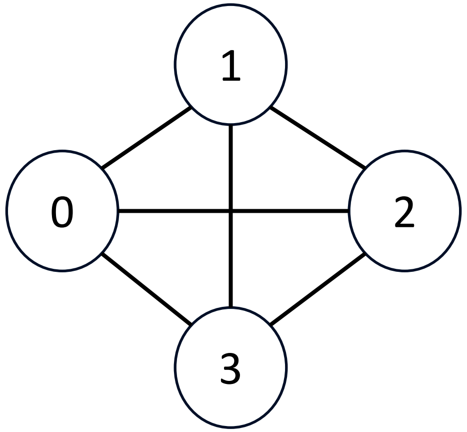
```
IterativeDFS(s):
    Push(s)
    while the stack is not empty
        v ← Pop
        if v is unmarked
            mark v
            for each edge vw
                Push(w)
```

DFS: Pseudocode

```
dfs(v){// may add path as another parameter
    if(v is solution) update solution
    // prevent infinite loop
    if(v is dead end) return;
    // iterate all valid options, prevent infinite loop
    for each edge vw in options {
        get next node
        // e.g. boundary in a maze
        if(w is illegal) continue;
        // prevent infinite loop
        check if w is visited and update visited hash table
        // if w will affect future options, e.g., no repeated digits
        update options hash table
        // the core
        dfs(w)
        // w is backtracked thus revert the selection, but don't revert the visited hash table
        revert options hash table
    }
}
```

DFS: Permutations

Given an array nums of distinct integers, return all the possible permutations



	options
0	
1	
2	
3	

stack

DFS: Permutations

```
vector<vector<int>> permute(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> result;
        vector<int> path(n), options(n);
        dfs(0, nums, path, result, options);
        return result;
    }

    void dfs (int i, vector<int>& nums, vector<int> &path, vector<vector<int>> &result, vector<int> &options) {
        int n = nums.size();
        if (i == n) {
            result.push_back(path); // append
            return;
        }
        for (int j = 0; j < n; j++) {
            if (!options[j]) {
                path[i] = nums[j]; // select from the options that is unselected
                options[j] = true; // select
                dfs(i + 1, nums, path, result, options);
                options[j] = false; // revert the select
                // simply overwrite path
            }
        }
    }
```

DFS: Number of Islands

- Problem:
  - Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.
  - An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water
  - Input: grid = [
    ["1","1","0","0","0"],
    ["1","1","0","0","0"],
    ["0","0","1","0","0"],
    ["0","0","0","1","1"]
    ]
  - Output: 3

DFS: Number of Islands

```
int numIslands(vector<vector<char>>& grid) {
    int m = grid.size(), n = grid[0].size();
    int result = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == '1') { // find an unvisited island
                // flood this island as 'v', so next '1' will be an unvisited island
                dfs(i, j, grid);
                result++;
            }
        }
    }
    return result;
}
```

```
void dfs (int i, int j, vector<vector<char>>& grid){
    int m = grid.size(), n = grid[0].size();
    // boundry or not islands
    if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] != '1') {
        return;
    }
    grid[i][j] = 'v'; // visited

    // 4 options
    dfs(i, j - 1, grid); // left
    dfs(i, j + 1, grid); // right
    dfs(i - 1, j, grid); // up
    dfs(i + 1, j, grid); // down
};
```

BFS: Property

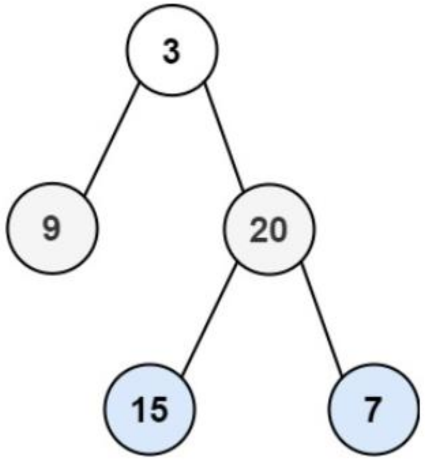
- Property
  - Exhaustive search, time complexity: O(n+m) with O(1) to traverse one edge, space complexity: O(k^h), h is the height, and k is the branching number (# of options)
  - **Queue:** traversal order, **layer by layer**: finish current layer before travers to the next
  - Layer distance/step distance to the source is increasing
  - Usually suitable for shallow solution or min step
  - Others are similar to DFS
- Applications
  - Finding the shortest path from the starting point to all other points on an **unweighted** graph.
  - Finding all connected components.
  - Minimum steps to win a game.
  - Finding the smallest cycle in a directed **unweighted** graph.
  - Finding edges that must lie on the shortest path between two nodes
  - Finding a shortest path of even length.
  - Finding the shortest path on a graph with edge weights of 0 or 1. (https://cp-algorithms.com/graph/01_bfs.html)

```
bfs(s) {
  q = new queue()
  q.push(s), visited[s] = true
  while (!q.empty()) {
    u = q.pop()
    for each edge(u, v) {
      if (!visited[v]) {
        q.push(v)
        visited[v] = true
      }
    }
  }
}
```

BFS: Level Order Traversal

SINGAPORE INSTITUTE OF TECHNOLOGY

- Problem
  - Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).
  - Input: root = [3,9,20,null,null,15,7]
  - Output: [[3],[9,20],[15,7]]




```
vector<vector<int>> levelOrder(TreeNode *root) {
    if (root == nullptr) return {};
    vector<vector<int>> result;
    queue<TreeNode *> q;
    q.push(root);
    while (!q.empty()) {
        vector<int> vals;
        for (int n = q.size(); n--;) {
            auto node = q.front();
            q.pop();
            vals.push_back(node->val);
            if (node->left)  q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.emplace_back(vals);
    }
    return result;
}
```

BFS Improvement: Bidirectional BFS

- Problem: the tree could be very wide
- Main idea
  - Start searching simultaneously/alternatively from both directions. Once the same value is found, it means a shortest path connecting the start and end points has been identified
  - Only for traverse can be reverted, e.g., two way not one way street
- Steps
  - Create **two queues** for bidirectional BFS in both directions
  - Create **two hash tables** to avoid repeated searches on the same nodes and to **record the number of transformations**
  - If a node that has been visited by the opposite search is encountered during the search, it indicates that the shortest path has been found

Bidirectional BFS: Example

```
// Returns true if target is reachable from src
// by meeting in the middle using Bidirectional BFS
bool BidirectionalBFS(Graph, src, target)
    if src == target
        return true

    // Initialize two queues for each search direction
    queue_src := empty queue
    queue_target := empty queue
    queue_src.enqueue(src)
    queue_target.enqueue(target)

    // Initialize visited sets for each direction
    visited_src := {src}
    visited_target := {target}

    // Perform BFS from both directions until a meeting point is found
    while not queue_src.is_empty() and not queue_target.is_empty()
        // Expand from the source side
        if BFSExpand(Graph, queue_src, visited_src, visited_target)
            return true

        // Expand from the target side
        if BFSExpand(Graph, queue_target, visited_target, visited_src)
            return true

    return false   // No path found
return false
```

```
// Expands the BFS frontier from one direction and checks for intersection
bool BFSExpand(Graph, queue, visited_current, visited_opposite)
    node := queue.dequeue()

    // Explore each neighbor of the current node
    for each neighbor in Graph[node]
        // Check if neighbor has been visited from the opposite direction
        if neighbor in visited_opposite
            return true   // Meeting point found

        // Continue expanding if neighbor has not been visited in current direction
        if neighbor not in visited_current
            visited_current.add(neighbor)
            queue.enqueue(neighbor)
```

DFS Improvement: Iterative Deepening DFS

- Problem
  - DFS: if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges)
  - BFS: level by level, but requires more space
- Main Idea
  - IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. DFS in a BFS fashion

```
// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
    for limit from 0 to max_depth
       if DLS(src, target, limit) == true
           return true
    return false

bool DLS(src, target, limit)
    if (src == target)
        return true;

    // If reached the maximum depth,
    // stop recursing.
    if (limit <= 0)
        return false;

    foreach adjacent i of src
        if DLS(i, target, limit-1)
            return true

    return false
```

Generic Shortest Path Algorithm

- BFS: unweighted/step shortest path
- Efficient implementations: How to choose which edge to relax? Priority Queue
  - Dijkstra's algorithm (nonnegative weights)
  - Bellman-Ford algorithm (no negative cycles)

Generic algorithm (to compute SPT from s)

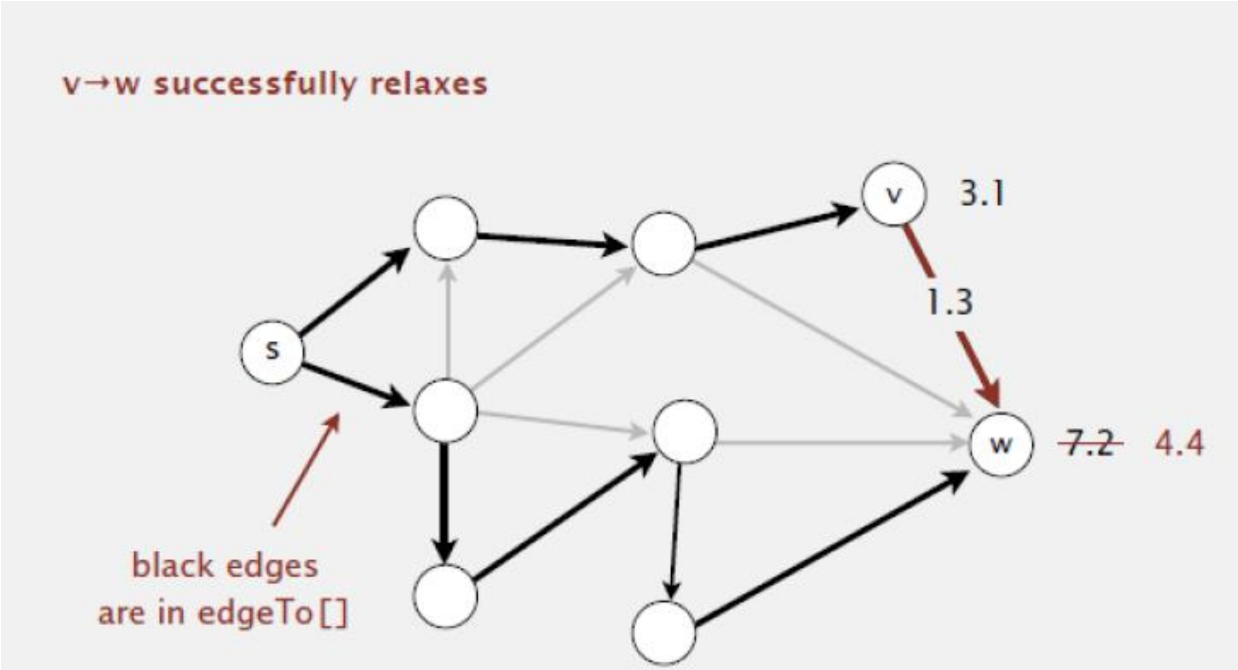
Initialize dist[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

– Relax any edge.

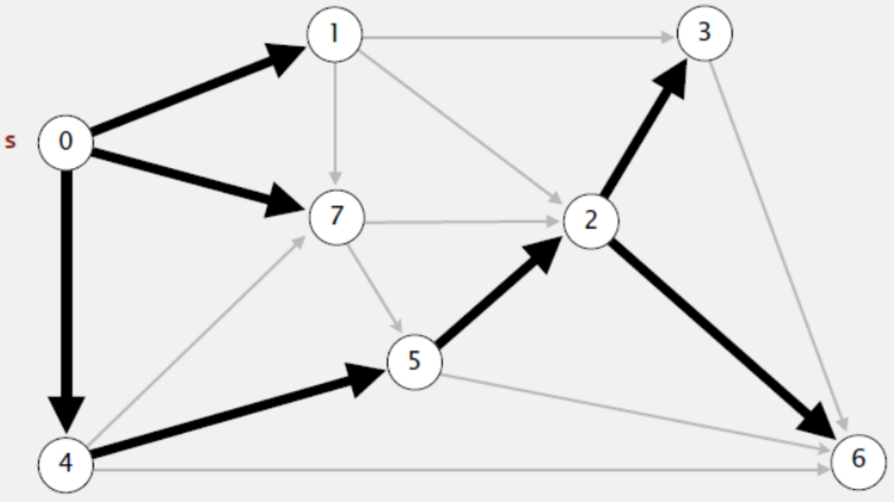
Edge Relaxation

- Relax edge e = v -> w
- dist[v] is the length of shortest known path from s to v
- dist[w] is the length of shortest known path from s to w
- edgeTo[w] is the last edge on shortest known path from s to w
- If e = v -> w gives shorter path to w through v, update both dist[w] and edge[w]



Dijkstra Algorithm

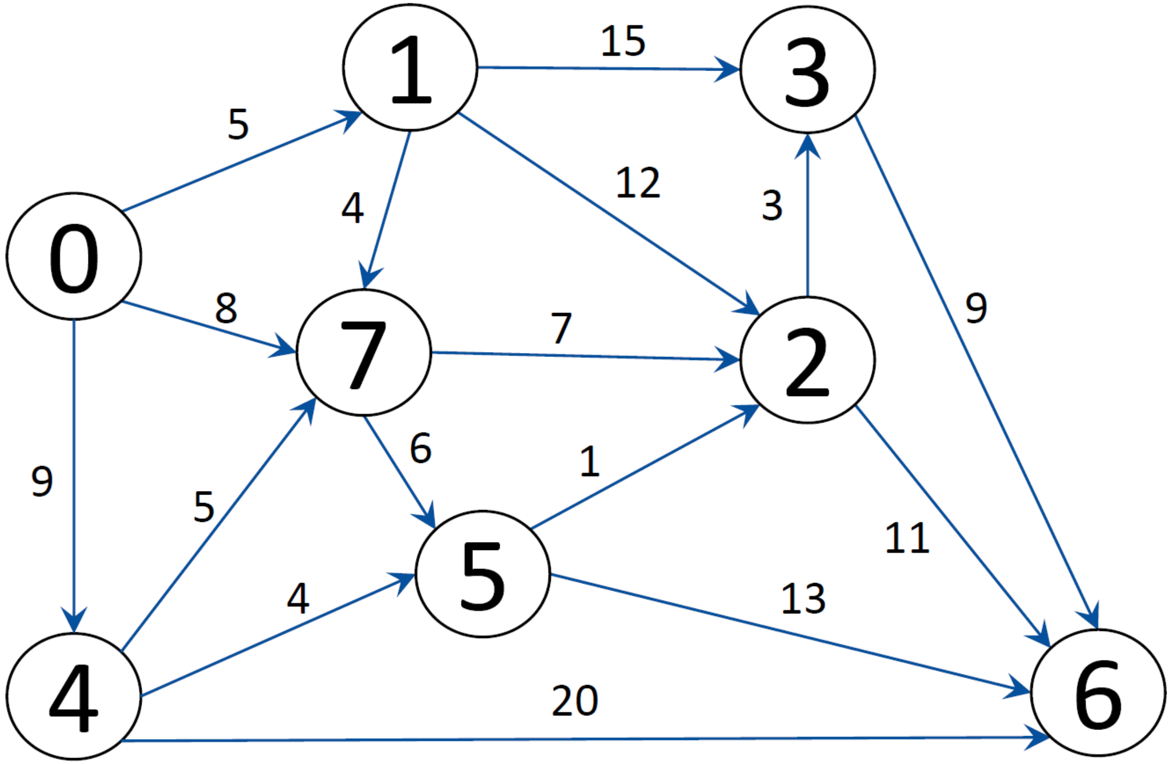
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest dist[] value
- Add vertex to tree and **relax** all edges **pointing from that vertex**



v	dist[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

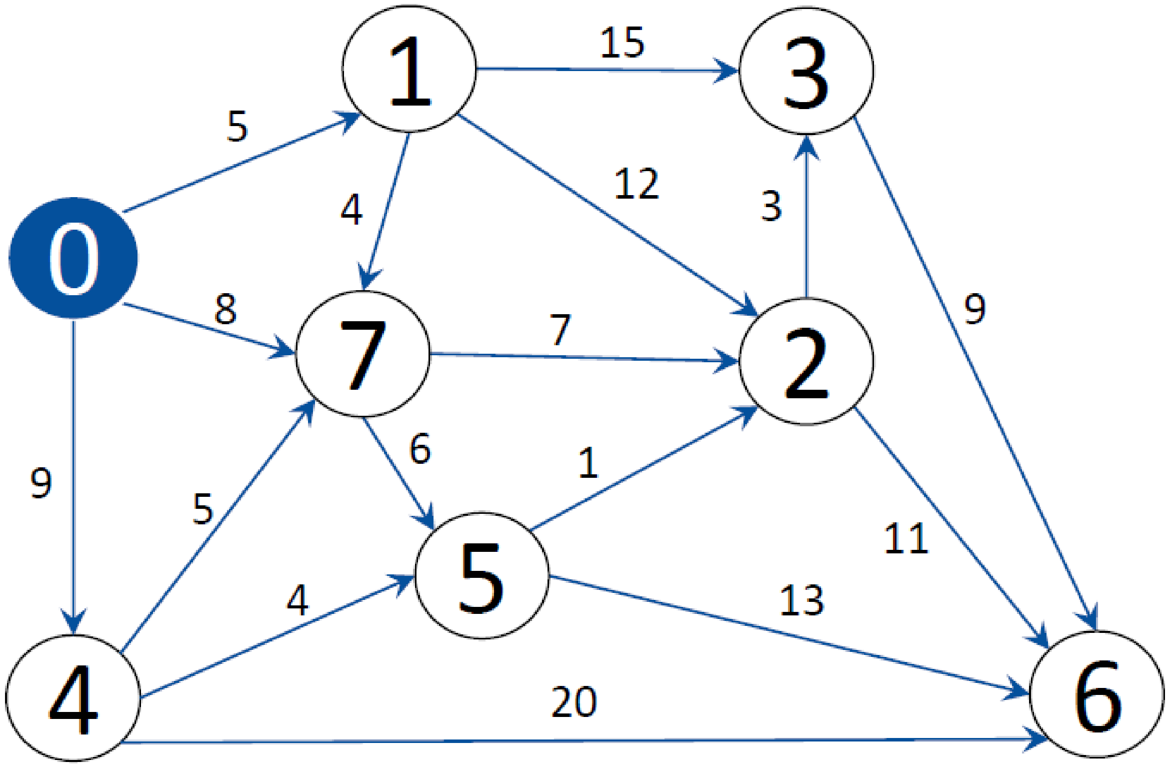
Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	∞	-
2	∞	-
3	∞	-
4	∞	-
5	∞	-
6	∞	-
7	∞	-

Dijkstra Algorithm

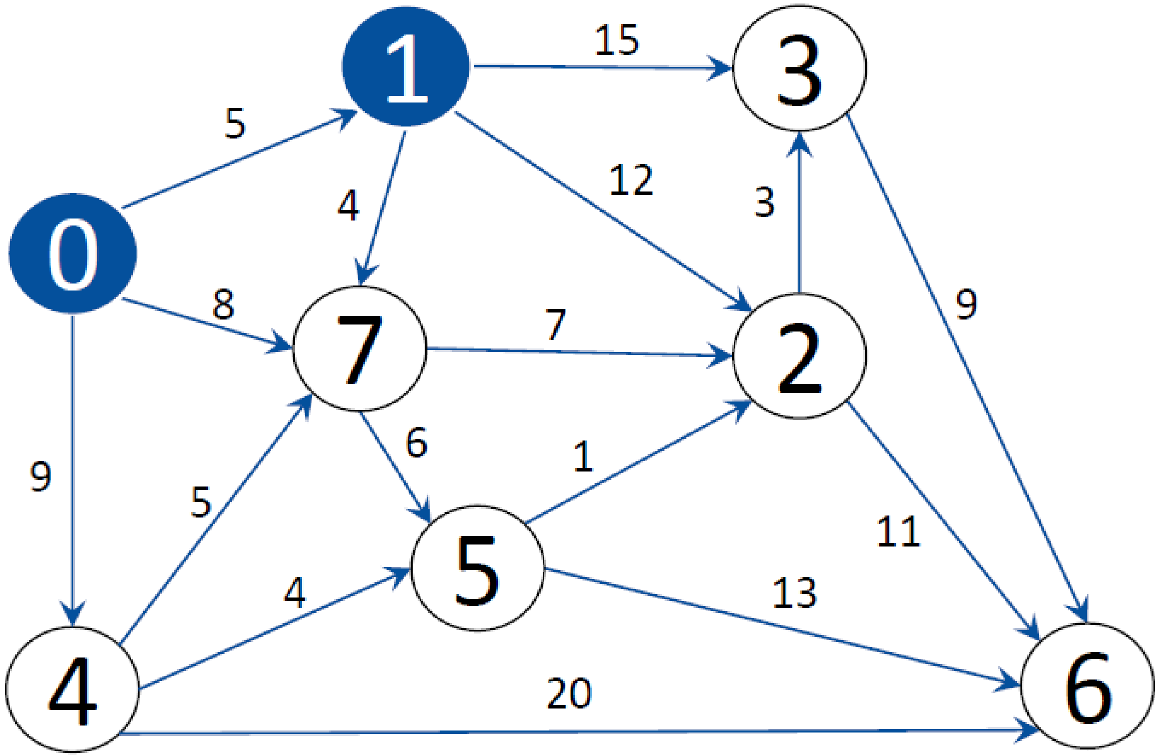
SINGAPORE INSTITUTE OF TECHNOLOGY



v	dist	edgeTo
0	0	-
1	5	0
2	∞	-
3	∞	-
4	9	0
5	∞	-
6	∞	-
7	8	0

Comment: Mark node 0, relax neighbors of node 0, update edgeTo for nodes 1, 7, 4 as node 0

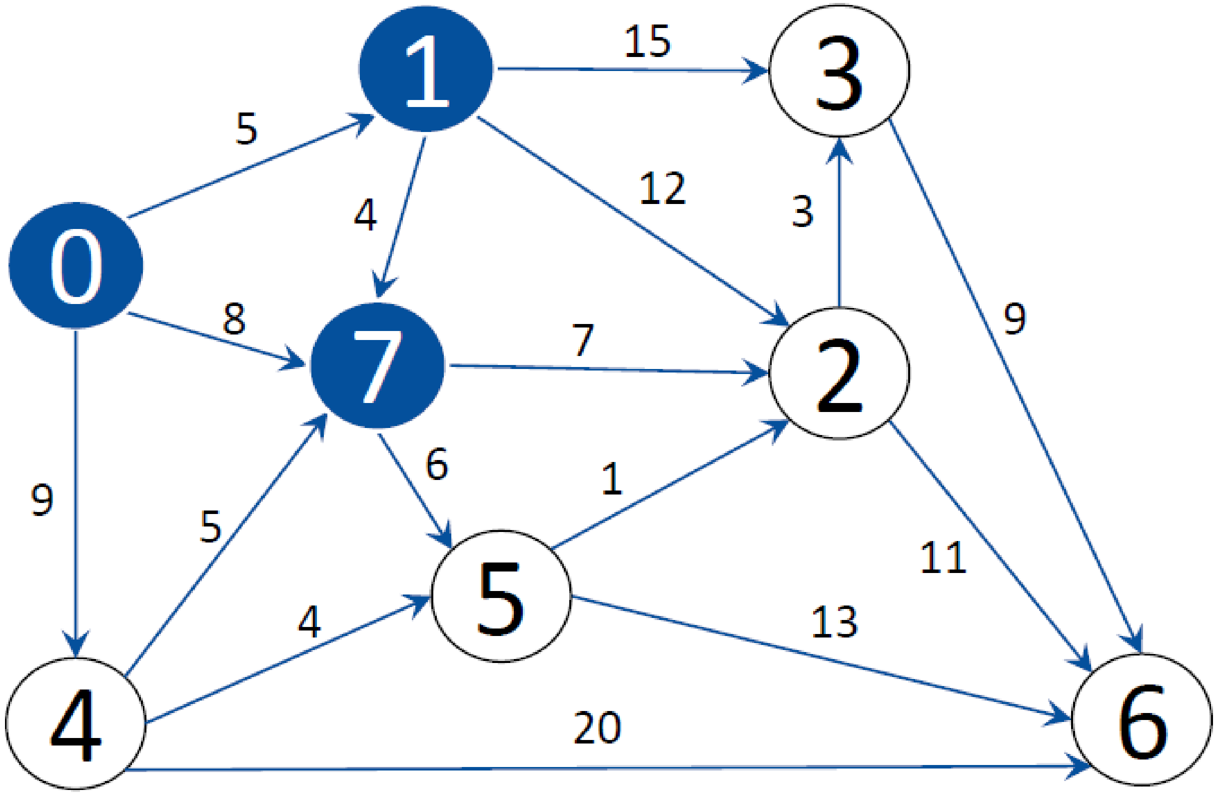
Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	5	0
2	17	1
3	20	1
4	9	0
5	∞	-
6	∞	-
7	8	0

Comment: Mark node 1 (distance is the minimum), relax neighbor nodes 2 & 3, update edgeTo for nodes 2, 3 as node 1. Note that node 7 remains unchanged after the relaxation as dist(1)+weight(1-7)=5+4>dist(7)=8

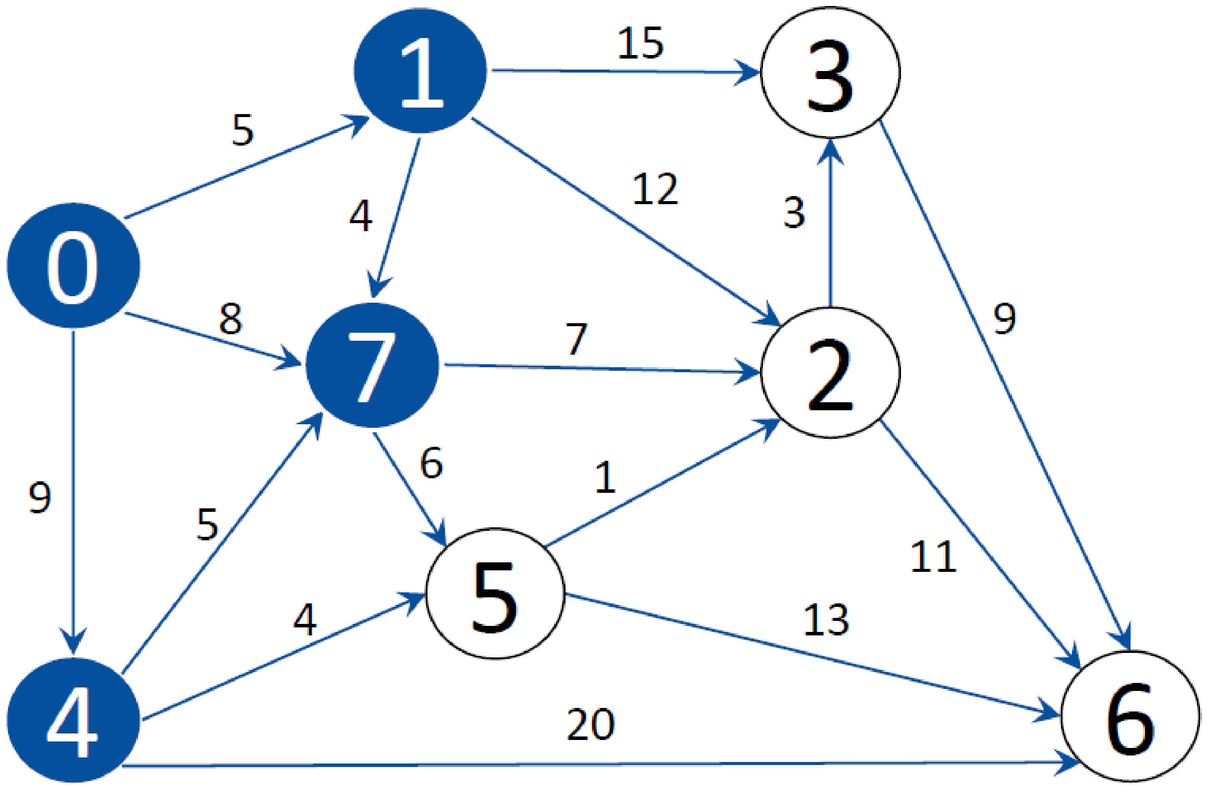
Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	5	0
2	15	7
3	20	1
4	9	0
5	14	7
6	∞	-
7	8	0

Comment: Mark node 7 (distance is the minimum), relax neighbor nodes 2 & 5, update edgeTo for nodes 2, 5 as node 7.

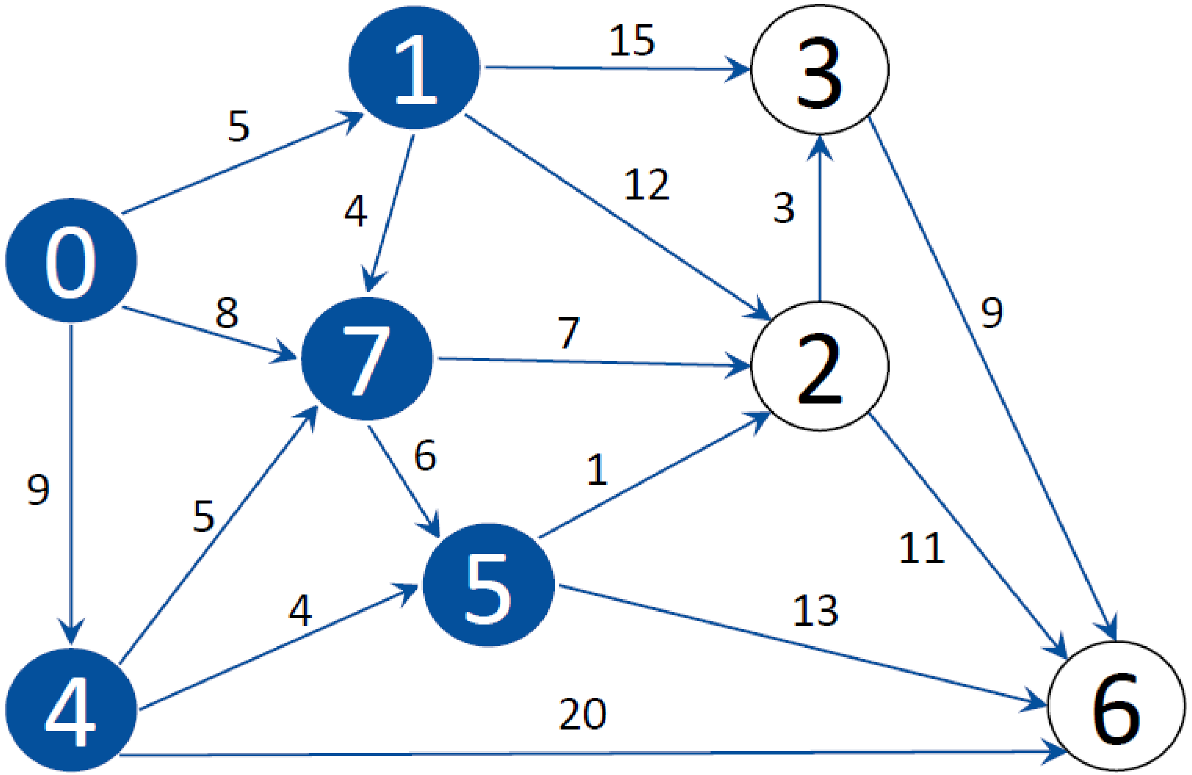
Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	5	0
2	15	7
3	20	1
[illegible]	9	0
5	13	4
6	29	4
7	8	0

Comment: Mark node 4 (distance is the minimum), relax neighbor nodes 5 & 6, update edgeTo for nodes 5 & 6 as node 4.

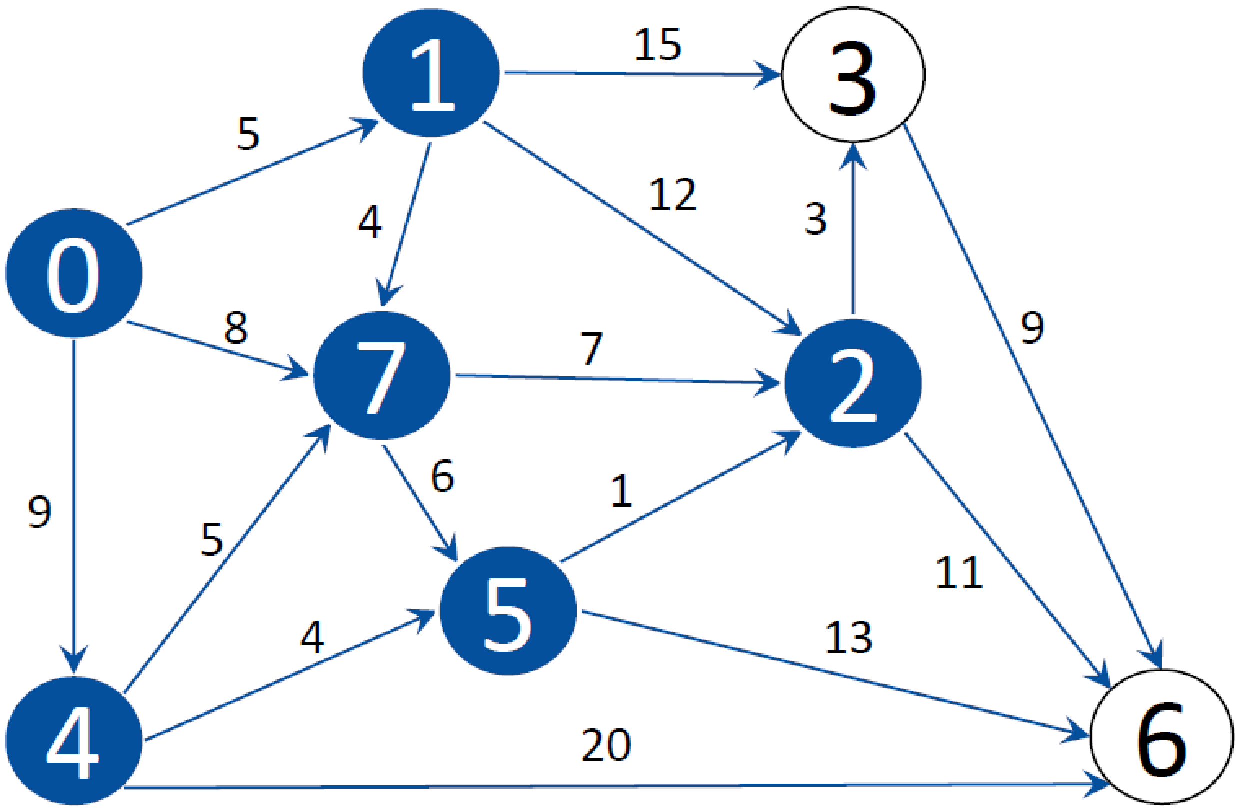
Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	5	0
2	14	5
3	20	1
4	9	0
5	13	4
6	26	5
7	8	0

Comment: Mark node 5 (distance is the minimum), relax neighbor nodes 2 & 6, update edgeTo for nodes 2 & 6 as node 5.

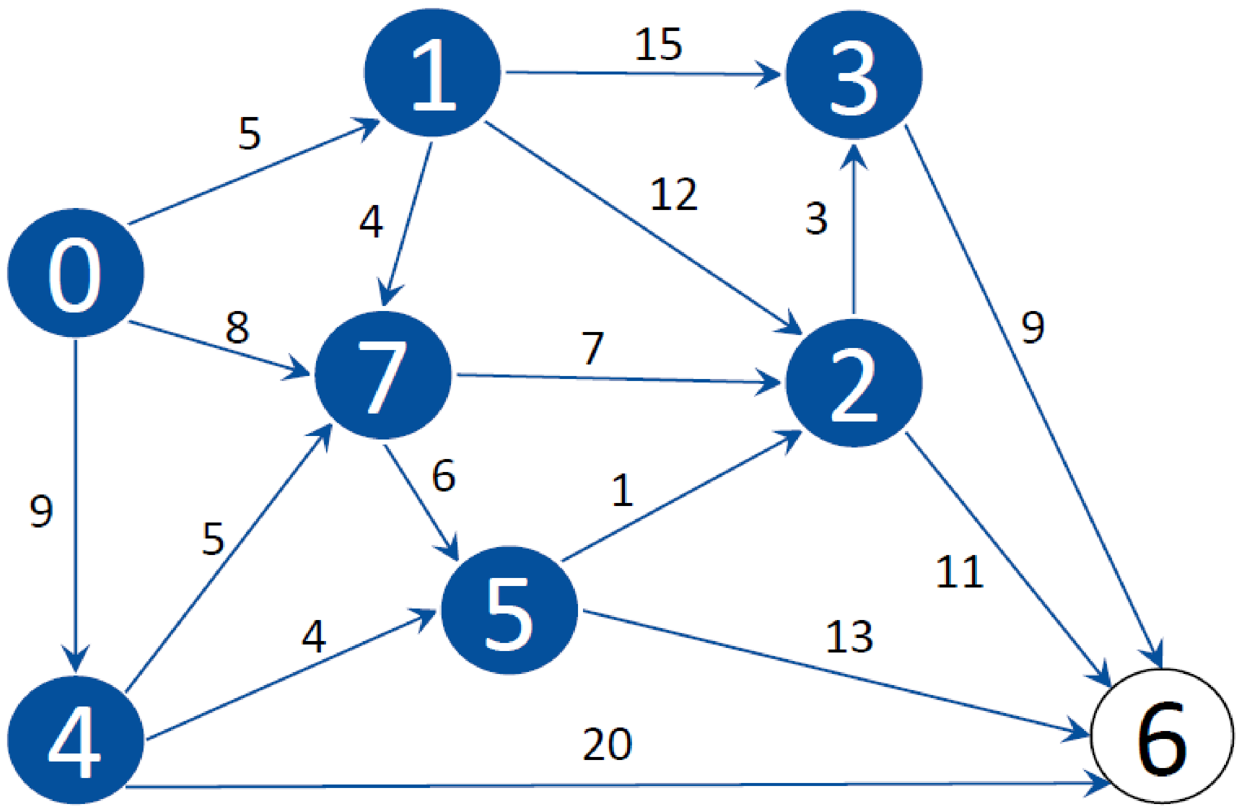
Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	5	0
2	14	5
3	17	2
4	9	0
5	13	4
6	25	2
7	8	0

Comment: Mark node 2 (distance is the minimum), relax neighbor nodes 3 & 6, update edgeTo for nodes 2 & 6 as node 5.

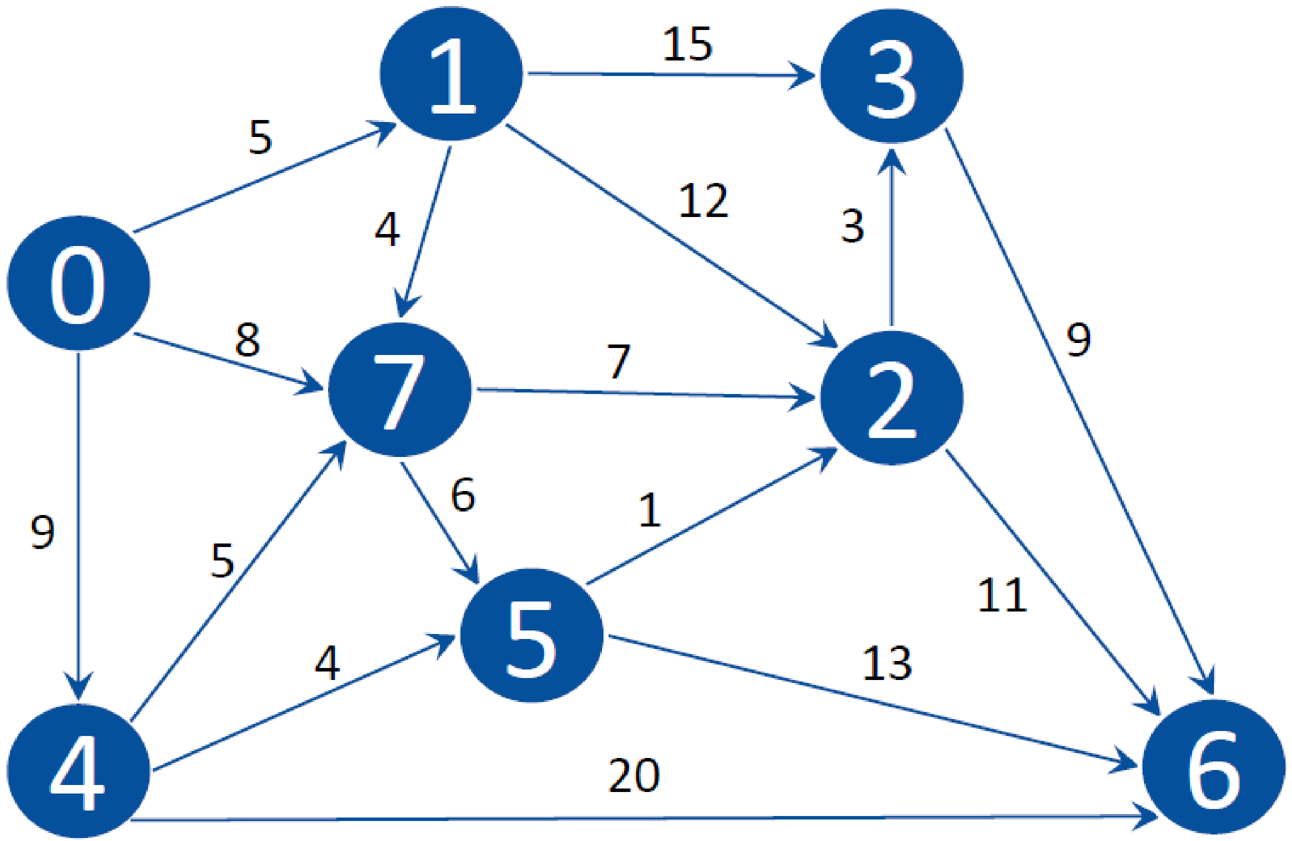
Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	5	0
2	14	5
3	17	2
4	9	0
5	13	4
6	25	2
7	8	0

Comment: Mark node 3 (distance is the minimum). Note that node 6 remains unchanged after the relaxation as dist(3)+weight(3-6)=17+9>dist(6)=25

Dijkstra Algorithm



v	dist	edgeTo
0	0	-
1	5	0
2	14	5
3	17	2
4	9	0
5	13	4
6	25	2
7	8	0

Comment: Mark node 6 (distance is the minimum). All nodes are marked.

Dijkstra Algorithm: Pseudocode

```
function Dijkstra(Graph, s):
  for each vertex v in Graph.Vertices:
      distTo[v] ← INFINITY
      edgeTo[v] ← UNDEFINED
      add v to Q
  distTo[source] ← 0
  while Q is not empty:
      u ← vertex in Q with minimum distTo[u] <<<
      remove u from Q
      for each neighbor v of u still in Q:
          alt ← distTo[u] + Graph.Edges(u, v)
          if alt < distTo[v]:
             distTo[v] ← alt
             edgeTo[v] ← u
  return distTo[], edgeTo[]
```

```
Can terminate at <<<
if u==target
     S ← empty sequence
     u ← target
     if prev[u] is defined or u = source:// Only if the vertex is reachable
         while u is defined:// Construct the shortest path with a stack S
             insert u at the beginning of S // Push the vertex onto the stack
             u ← prev[u] // Traverse from target to source
```

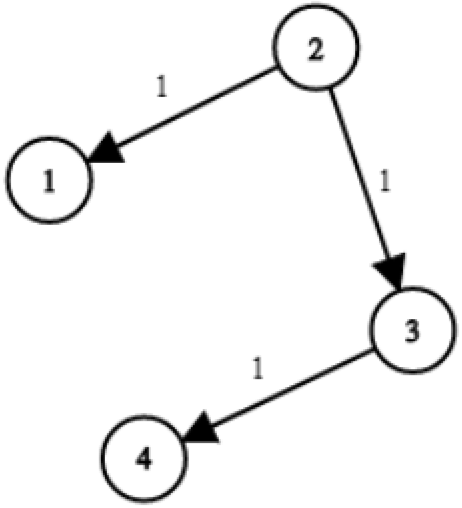
```
function Dijkstra(Graph, s):
   create vertex priority queue Q

   distTo[source] ← 0
   Q.add_with_priority(s, 0)

   for each vertex v in Graph.Vertices:
       if v ≠ s
           prev[v] ← UNDEFINED
           dist[v] ← INFINITY
           Q.add_with_priority(v, INFINITY)

   while Q is not empty:
       u ← Q.extract_min()
       for each neighbor v of u:
           alt ← distTo[u] + Graph.Edges(u, v)
           if alt < distTo[v]:
               edgeTo[v] ← u
               distTo[v] ← alt
               Q.decrease_priority(v, alt)

   return distTo, edgeTo
```

Dijkstra Algorithm: Shortest Path for All Paths

- Define
  - g[i][j]: edge weight between nodes i and j. No edge from i to j: g[i][j]=∞
  - dist[i]: shortest path length from the start node s to node i.
  - Initially, dist[s]=0 and all other dist[i]=∞ indicating they have not yet been computed.
- Our goal is to calculate the final dist array.
- First, update the shortest path from the start node s to each of its neighbors y, setting dist[y]=g[s][y]. Then, find the minimum dist[i] value among nodes other than s; assume this minimum corresponds to node u. At this point, we can assert that dist[u] is indeed the shortest path length from s to u; no other path from s to u could be shorter.
- Proof by contradiction: suppose a shorter path exists. In that case, we would start from s and pass through a node w with dist[w]<dist[u], then reach u via some additional edges, yielding a smaller dist[u]. However, if w is unvisited, since dist[u] is already the minimum and the graph has no negative edge weights, such a w does not exist, leading to a contradiction. If w is visited and suppose it is the last to u on shortest path, thus (through w is shorter) dist[w]+g[w][u]<dist[u], however, w has been used to relax u, dist[w]+g[w][u] >= dist[u], leading to a contradiction. Thus, the original statement is true, and we now have the final value for dist[u].
- Next, update/relax dis[y] using the edge weight g[u][y]: if dist[u]+g[u][y]<dist[y], then update dist[y] to dist[u]+ g[u][y], otherwise, do not update it. Then, find the minimum dist[i] value among nodes other than s and u, and repeat the process above.
- By mathematical induction, we can confirm that this approach calculates the shortest path for each node. The algorithm ends once the shortest paths for all nodes have been determined.

Dijkstra Algorithm: Network Delay Time

- Problem:
  - You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.
  - We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.
  - Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
  - Output: 2



Dijkstra Algorithm: Network Delay Time

```
// min Heap
class mycomparison {
public:
    bool operator()(const pair<int, int>& lhs, const pair<int, int>& rhs){
        return lhs.second > rhs.second;
    }
};
struct Edge {
    int to;
    int val; // weight
    Edge(int t, int w): to(t), val(w) {}
};
int networkDelayTime(vector<vector<int>>& times, int n, int k) {

    std::vector<std::list<Edge>> Graph(n + 1);
    for(int i = 0; i < times.size(); i++){
        int p1 = times[i][0];
        int p2 = times[i][1];
        Graph[p1].push_back(Edge(p2, times[i][2]));
    }

    std::vector<int> dist(n + 1, INT_MAX/2);

    std::vector<bool> visited(n + 1, false);

    priority_queue<pair<int, int>, vector<pair<int, int>>, mycomparison> pq;

    pq.push(pair<int, int>(k, 0));
    dist[k] = 0;

    while (!pq.empty()) {
            // find the unvisited and closest to s an
            pair<int, int> cur = pq.top(); pq.pop();

             // cur is finished, cannot find shorter
            if (visited[cur.first]) continue;
            //if (cur.second>dist[cur.first]) continue;

            //  mark, cur is done
            visited[cur.first] = true;

            for (Edge edge : Graph[cur.first]) { // iteratate cur's neighbors
                // cur points to edge.to with weight edge.val
                if (!visited[edge.to] && dist[cur.first] + edge.val < dist[edge.to]){
                    // relax cur's neighbors
                    dist[edge.to] = dist[cur.first] + edge.val;
                    pq.push(pair<int, int>(edge.to, dist[edge.to]));
                }
            }

        }

        int result = 0;
        for (int i = 1; i <= n; i++) {
            if (dist[i] == INT_MAX/2) return -1;// no path
            result = max(dist[i], result);
        }
        return result;

}
```

Dijkstra Algorithm: Time Complexity

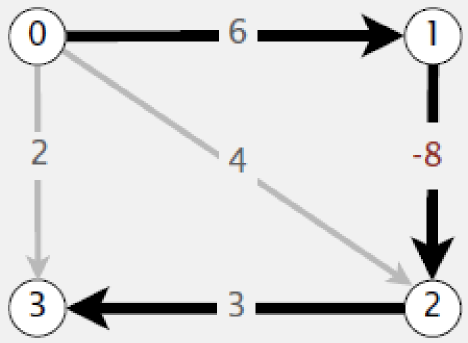
- Using Priority Queue as data structure:

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

- Array implementation optimal for dense graphs
- Binary heap much faster for sparse graphs

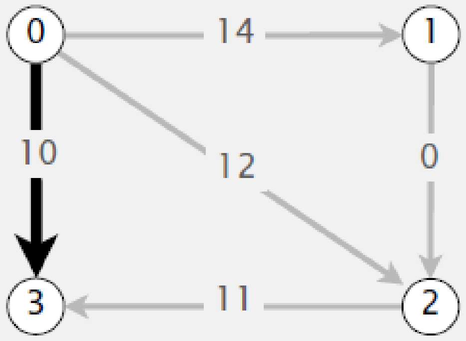
Shortest-Path with Negative Edge Weights

- Dijkstra: Doesn't work with negative edge weights



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is 0→1→2→3.

- Re-weighting: Add a constant to every edge weight doesn't work



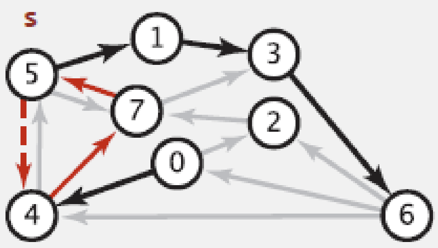
Adding 8 to each edge weight changes the shortest path from 0→1→2→3 to 0→3.

Shortest-Path with Negative Cycles

- A negative cycle is a directed cycle whose sum of edge weights is negative

digraph

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

shortest path from 0 to 6

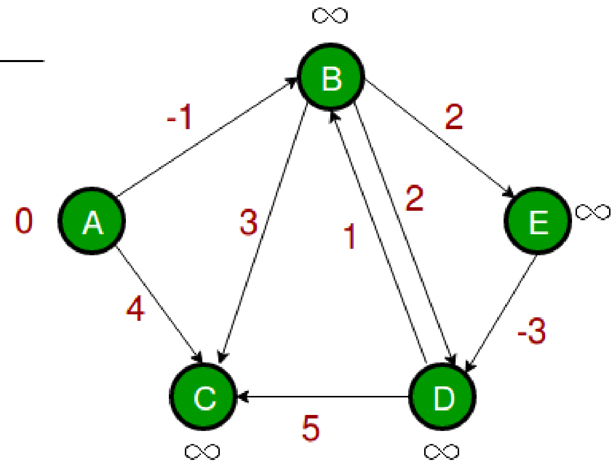
0->4->7->5->4->7->5...->1->3->6

- A Shortest-Path exists iff no negative cycles (assuming all vertices reachable from s)

Bellman-ford: Introduction

- Problem: Compute the **shortest path** from a single **source** vertex **to all** vertices in a weighted directed graph with positive/negative weights.
- General idea: try all edges and get the best.
  - Sounds like brute-force. Bad?
  - Lucky: optimal substructure holds.
- Bound (one factor):
  - Suppose n vertices on a path.
  - Number of edges: n-1.
- More specific idea:
  - We have an edge, say (u, v).
  - Edge distance is e. v is destination.
  - The latest shortest path from source to v is ev.
  - The latest shortest path from source to u is eu.
  - We consider something like a triangle.
  - ev > eu + e? If so, there is a shorter path, so update/record ev := eu + e.
  - Do this comparison and update again and again -> eventually get optimal.

A	B	C	D	E
0	∞	∞	∞	∞



https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Bellman-ford: Pseudocode

```
function BellmanFord(Graph, source):
    // Step 1: Initialize distances from source to all vertices as infinite
    // and distance to the source itself as 0
    for each vertex v in Graph:
        distance[v] := ∞
        predecessor[v] := null
    distance[source] := 0

    // Step 2: Relax edges repeatedly
    for i from 1 to n - 1:   // n is |V|, the number of vertices
        for each edge (u, v) in Graph:
            if distance[u] + weight(u, v) < distance[v]:
                distance[v] := distance[u] + weight(u, v)
                predecessor[v] := u

    // Step 3: Check for negative-weight cycles, after n-1 still relax
    for each edge (u, v) in Graph:
        if distance[u] + weight(u, v) < distance[v]:
            print("Graph contains a negative-weight cycle")
            return

    return distance, predecessor
```

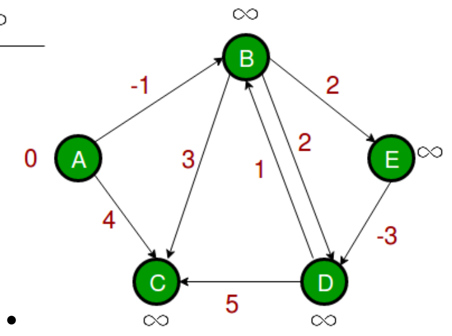
Bellman-ford: Discussions

- **Insight on the Shortest Path Discovery**
  - **One iteration** through all the edges in the **Bellman-Ford** algorithm helps identify at least one edge that contributes to the shortest path for each vertex if the shortest path contains that edge.
  - Each iteration helps propagate the shortest distance information further along the graph, effectively expanding the search until all shortest paths are found.
  - **1st iteration:** the Bellman-Ford algorithm will correctly determine the shortest path for all vertices that can be reached from the source with **just one edge**.
  - **2nd iteration**: it extends to **paths involving two edges**.
  - **n-1** iterations: it will have found the shortest paths that use up to **n-1 edges**.
- **Why n-1 Iterations?**
  - In the worst case, a shortest path could involve all vertices (i.e., a path of **n-1** edges in a graph with **n** vertices).

Bellman-ford: Example

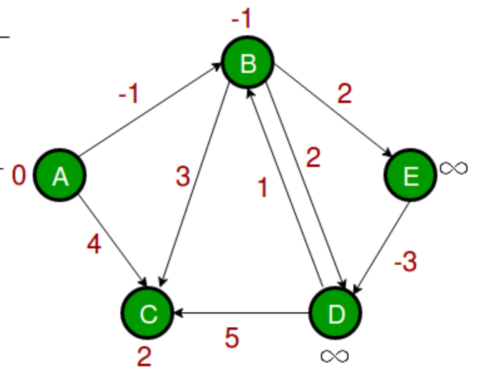
- Initialization: A table, with the latest distances.
  - Distance: 0 from source A to source A.
  - Distance: infinite from A to other vertices.

A	B	C	D	E
0	∞	∞	∞	∞



- 1^{st} iteration:
  - Consider all the edges:
    - (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D).
    - Any order in the beginning. Once fixed, use it till the end.
  - (B,E): E won’t change (inf -> inf).
  - (D,B): B won’t change (inf -> inf).
  - (B,D): D won’t change (inf -> inf). Not 1?
  - (A,B): B inf -> -1 (shorter path).
  - (A,C): C inf -> 4 (shorter path).
  - (D,C): C won’t change (inf > 4, infeasible).
  - (B,C): C 4 -> 2 (shorter path, via B, not A).
  - (E,D): D won’t change (inf -> inf)

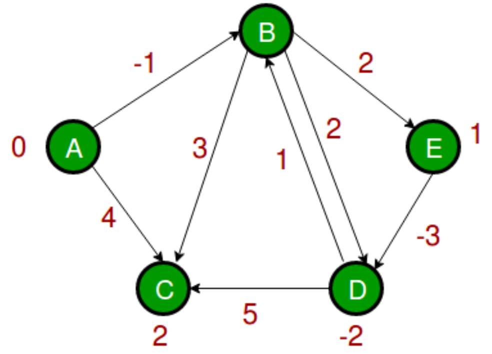
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



Bellman-ford: Example

- 2nd iteration:
  - Same edge order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D).
  - (B,E): E inf -> 1 (shorter path, via B).
  - (D,B): B won't change (inf > -1, infeasible).
  - (B,D): D inf -> 1 (shorter path, via B).
  - (A,B): B won't change (source unchanged).
  - (A,C): C won't change (4 > 2 , infeasible).
  - (D,C): C won't change (6 > 2, infeasible).
  - (B,C): C won't change (B unchanged).
  - (E,D): D 1 -> -2 (shorter path, via E).
- 3rd iteration:
  - (B,E), E won't change; (D,B): B won't change; …
  - No more update. Done.
- Time complexity: O(nm). Check proofs if interested.
  - n and m are the # of vertices and edges, respectively.
  - Better complexity/implementation available.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Bellman-ford Improvement

- Main idea
  - Reduce redundant relaxations by **only processing vertices that might lead to further distance reductions**
  - Using a **queue** to track only the vertices whose distances were recently updated, as these are the only vertices that may lead to further reductions in distances to other vertices.
  - Only processing vertices from the queue if they potentially offer a shorter path to adjacent vertices.
- Helps avoid redundant updates, especially on sparse graphs
  - Worst case: O(nm), Average case: O(m)

Queue Optimized Bellman-Ford

```
function Bellman-Ford (Graph, source):
    // Step 1: Initialize distances from source to
    //all vertices as infinity
    // and distance to the source itself as 0
    for each vertex v in Graph:
        distance[v] := ∞
        inQueue[v] := false
        edgeTo[v] := null
        dist[source] := 0

    // Step 2: Initialize the queue
    //and add the source vertex
    queue := empty queue
    enqueue(queue, source)
    inQueue[source] := true

// Step 3: Process vertices in the queue
while queue is not empty:
    u := dequeue(queue)
    inQueue[u] := false

    // Relax all edges from vertex u
    for each edge (u, v) in Graph:
        if dist[u] + weight(u, v) < dist[v]:
            dist[v] := dist[u] + weight(u, v)
            edgeTo[v] := u

        // If v is not already in the queue, add it
            if not inQueue[v]:
                enqueue(queue, v)
                inQueue[v] := true

return dist, edgeTo
```

Bellman-ford: Negative Cycles

- Bellman-ford can be used to **detect negative cycles**.
  - An important application for many graph algorithms.
  - Think about a triangle with -1 weight for all edges.
  - Without negative cycles, the algorithm can find the shortest paths within certain (n-1) number of iterations.
  - With negative cycles, every iteration there will be some changes.
  - Method: after the "certain number of iterations", check one more iteration.
    - Any change, there is negative cycles.
    - Otherwise, it is good.
- Optimal substructure. Satisfied?
  - Suppose the path = s -> v1 -> v2 -> … -> vi -> t.
  - dist(path) = dist(s -> v1 -> v2 -> … -> vi) + dist(vi -> t).
  - Can decompose.
  - Can solve smaller problem dist(s -> v1 -> v2 -> … -> vi) first.
  - Details available in our reference books.

https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Floyd-Warshall

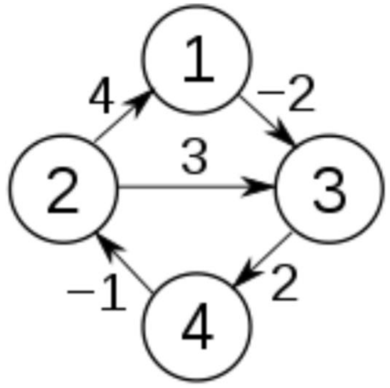
- Problem: find **shortest paths** between **all pairs** of vertices in a weighted graph with positive/negative edge weights (no negative cycles allowed).
  - Dijkstra algorithm: one vertex to another or all the other vertex.
  - Bellman-ford: one vertex to all the other vertices.
  - Floyd-Warshall: all vertices to all vertices.
- Algorithm (O(n^3)):

```
let dist be a n × n array of minimum distances initialized to infinity
for each edge (u, v)
    do dist[u][v] := w(u, v)  // The weight of the edge (u, v)
for each vertex v
    do dist[v][v] := 0
for k from 1 to n <- k has to be the outer loop
    for i from 1 to n
        for j from 1 to n
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] := dist[i][k] + dist[k][j]
```

https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Floyd-Warshall

```
for each edge …
for each vertex …
for k from 1 to n
    for i from 1 to n
        for j from 1 to n
            dist update
```



k0	j1	j2	j3	j4
i1	0	∞	-2	∞
i2	4	0	3	∞
i3	∞	∞	0	2
i4	∞	-1	∞	0

2 -> 1 -> 3 better

k1	j1	j2	j3	j4
i1	0	∞	-2	∞
i2	4	0	2	∞
i3	∞	∞	0	2
i4	∞	-1	∞	0

4 -> 2 -> 1 -> 3 better
4 -> 2 -> 1 better

k2	j1	j2	j3	j4
i1	0	∞	-2	∞
i2	4	0	2	∞
i3	∞	∞	0	2
i4	3	-1	1	0

1 -> 3 -> 4 better
2 -> 1 -> 3 -> 4 better

k3	j1	j2	j3	j4
i1	0	∞	-2	0
i2	4	0	2	4
i3	∞	∞	0	2
i4	3	-1	1	0

k4	j1	j2	j3	j4
i1	0	-1	-2	0
i2	4	0	2	4
i3	5	1	0	2
i4	3	-1	1	0

1->3->4->2; 3 -> 4 -> 2 -> 1; 3 -> 4 -> 2 better

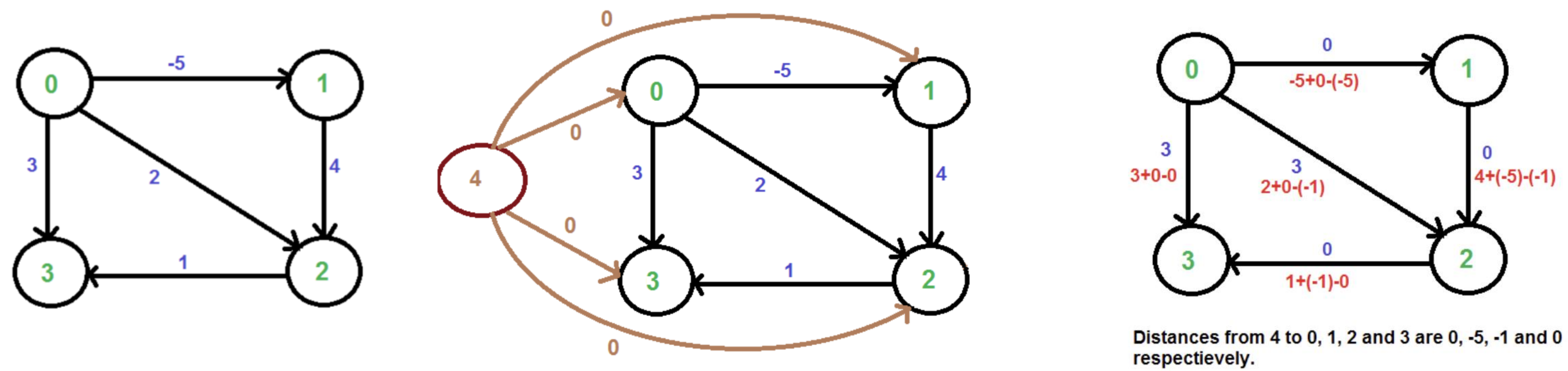
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Johnson's Algorithm

- Find the shortest paths between **all pairs** of vertices in a weighted directed graph
- Useful when there are **negative edge weights** (but no negative cycles)
- Main idea: combines **Dijkstra** and **Bellman-Ford** algorithms
  - Re-weight the edges in the graph so that all edge weights become non-negative.
  - Once re-weighted, Dijkstra Algorithm (efficient with non-negative weights) can be applied to each vertex to determine the shortest path from that vertex to all others.
  - Re-weighting ensures that the shortest path structure of the graph is preserved.
- Time Complexity: the main steps in the algorithm are Bellman-Ford Algorithm called once and Dijkstra called n times. Time complexity of Bellman Ford is O(mn) and time complexity of Dijkstra (if with Fibonacci heaps) is O(m+nlogn). So overall time complexity is O(n*nlog n + mn).

Johnson’s Algorithm: Steps

- Let the given graph be g. Add a new vertex s to the graph, add edges from the new vertex to all vertices of g. Let the modified graph be g’.
- Run the Bellman-Ford algorithm on g’ with s as the source. Let the distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.
- Reweight the edges of the original graph. For each edge (u, v), assign the new weight as “ w[u][v] (original weight) + h[u] – h[v]”.
- Remove the added vertex s and run Dijkstra algorithm for every vertex.



Distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectievely.

Johnson’s Algorithm: Implementation

```
void JohnsonAlgorithm(const vector<vector<int>>& graph, const vector<vector<int>>& edges) {
    int n = graph.size();  // Number of vertices
    // Get the modified weights from Bellman-Ford algorithm
    vector<int> altered_weights = BellmanFord_Algorithm(edges, n);
    vector<vector<int>> altered_graph(n, vector<int>(n, 0));
    // Modify the weights of the edges to remove negative weights
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (graph[i][j] != 0) {
                //w[u][v] + h[u] – h[v]
               altered_graph[i][j] = graph[i][j] + altered_weights[i] - altered_weights[j];
            }
        }
    }
    // Run Dijkstra's algorithm for every vertex as the source
    for (int source = 0; source < n; ++source) {
        Dijkstra_Algorithm(graph, altered_graph, source);
    }
}
```

Summary

Shortest Path methods	Dijkstra	Bellman–Ford	Floyd	Johnson
Type	Single source	Single source	All pairs	All pairs
Graph type	Non-negative weighted	Any	Any	Any
Detect Negative Cycles	No	Yes	Yes	Yes
Time Complexity	O(mlogn)	O(nm)	O(n^3)	O(nmlogn)

- Dijkstra using binary heap