SHORTEST PATHS

Made by: Makhmutov Ali
Kongratbayev Nurzhan
Akhmetshaiykov Nurbakyt
Zhumabek Islam
Otezhan Alua

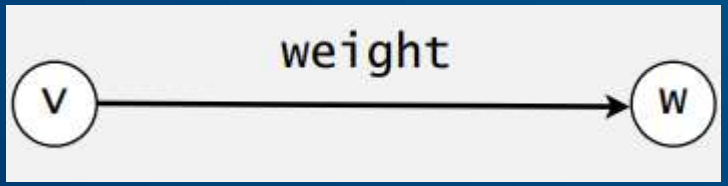
1. Weight of the edge
2. Shortest path
3. Negative-weight cycles
4. Applications
5. Edge relaxation
6. Dijkstra’s algorithm
7. Bellman-Ford algorithm
8. Floyd–Warshall algorithm

Weight of the edge

In a weighted graph, each edge has an associated numerical value, called the weight of the edge

Edge weights may represent distances, costs, etc

int from()	vertex v
int to()	vertex w
double weight()	weight of this edge



Shortest path

A shortest path from vertex s to vertex t in an edge-weighted digraph is a directed path from s to t with the property that no other such path has a lower weigh

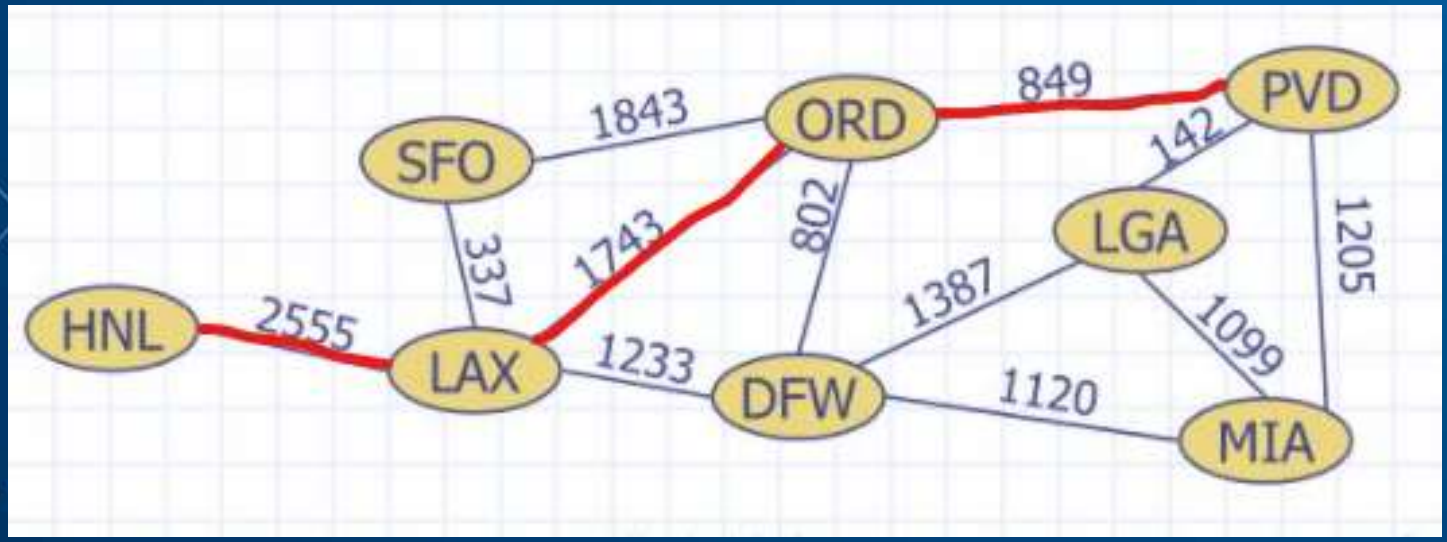
• Model the problem as a graph problem:

– Road map is a weighted graph: vertices = cities

edges = road segments between cities edge weights = road distances

– Goal: find a shortest path between two vertices (cities)

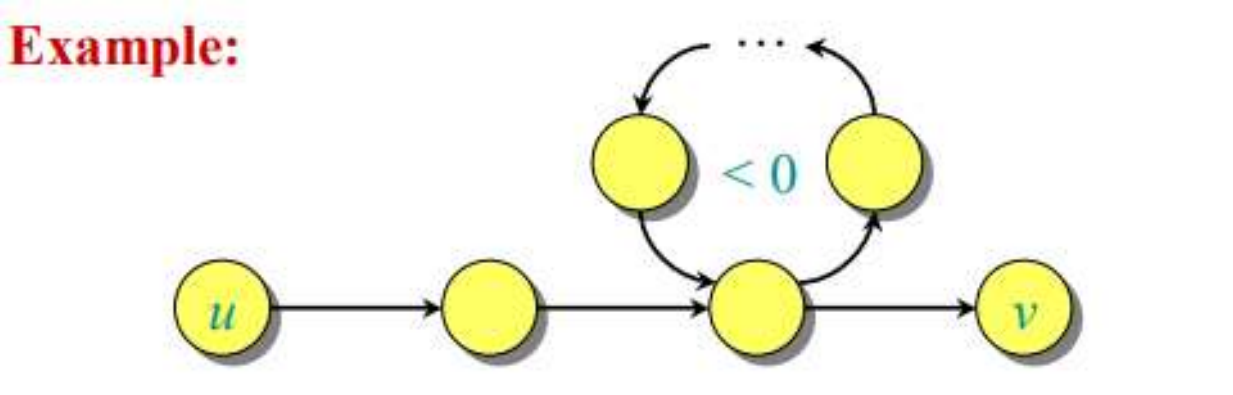
Shortest path between Providence and Honolul



Negative-weight cycles

Recall: If a graph contains a negativeweight cycle, then some shortest paths may not exist.

$w(u, v) = - \infty$



Shortest paths applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Seam carving

Original 1 Seam

Figure 1.1: Original image and single seam

50 Seams Finished

Figure 1.2: 50 seams carved and removed

Robot navigation

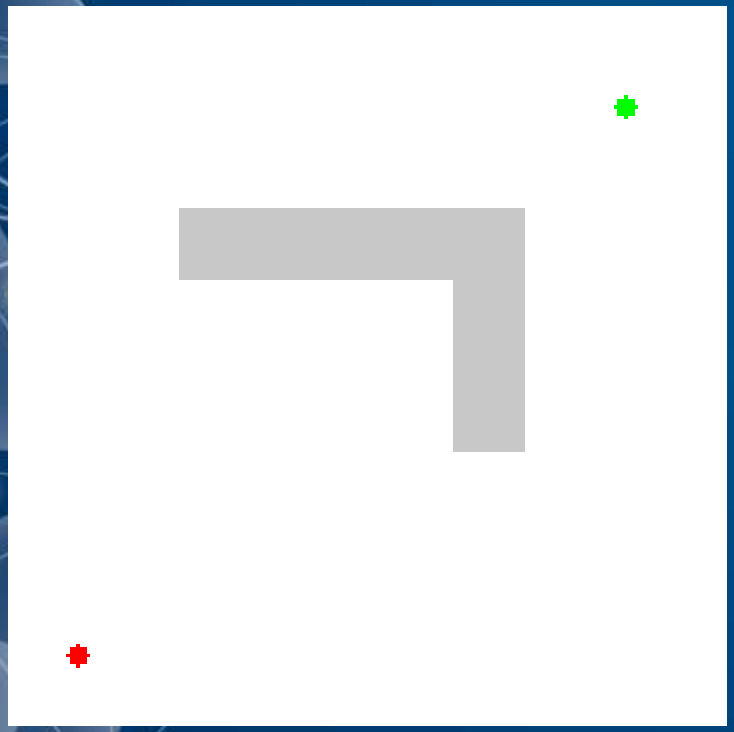
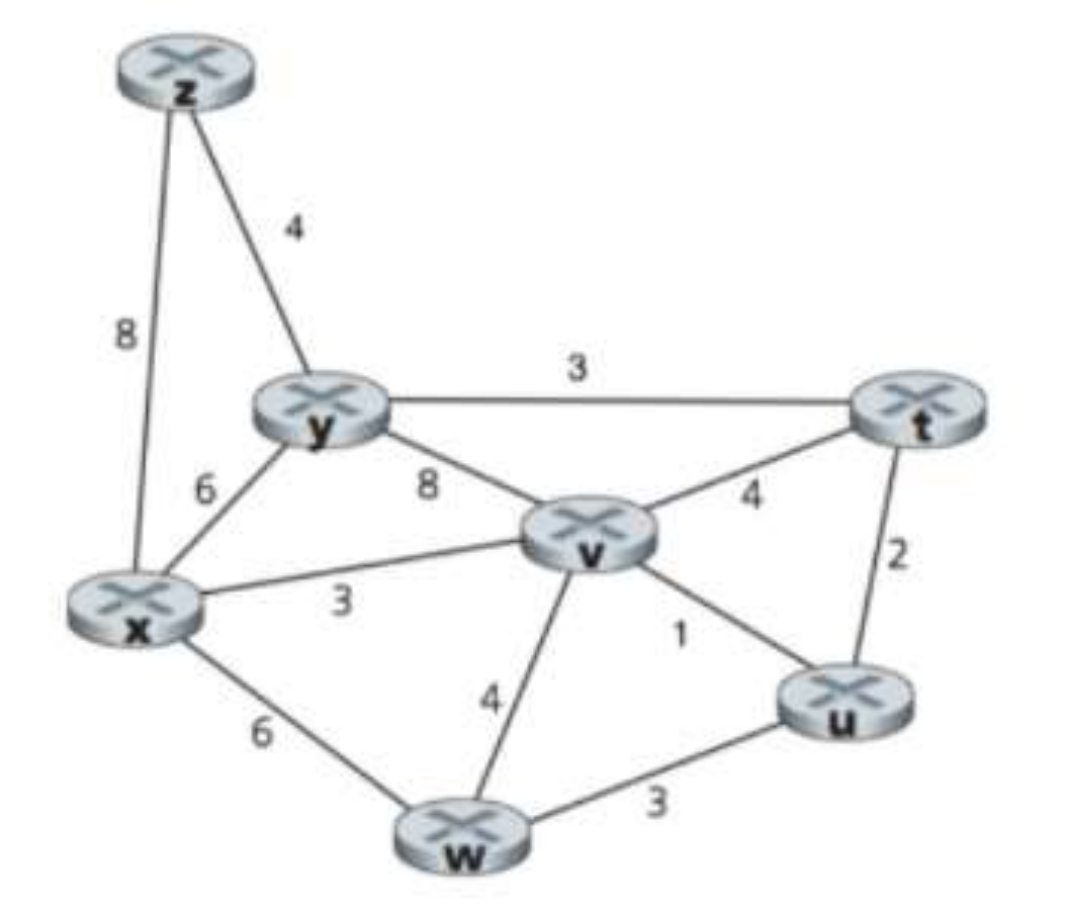


Illustration of Dijkstra's algorithm finding a path from a start node (lower left, red) to a goal node (upper right, green) in a robot motion planning problem. Open nodes represent the "tentative" set (aka set of "unvisited" nodes). Filled nodes are visited ones, with color representing the distance: the greener, the closer. Nodes in all the different directions are explored uniformly, appearing more-or-less as a circular wavefront as Dijkstra's algorithm uses a heuristic identically equal to 0

Map routing


Westbourne Grove
A4206
Notting Hill
Bayswater
A4205
St. George's Fields
A5
marble arch
Oxford St
A402
lancaster gate
Bayswater Rd
B411
queensway
Park Ln
Mt Street Gardens
Hyde Park
Kensington Gardens
The Long Water
2 Mins
Round Pond
The Serpentine
Palace Green
A4202
D
knight's bridge
A4
15 Mins
Kensington Rd
A315
A
high street
A4204
Richmond University (Kensington Campus)
4 Mins
Science Museum
Queen's Gate
Gloucester Rd
Grosvenor Pl
B310
A3216
B319
Knightsbridge
Cromwell Rd
gloucester road
south kensington
C
Brompton Rd
Belgravia
Sloane St
Ebury St
sloane square
Sloane Ave
Draycott Pl
A3214
A3217
B
Barkston Gardens
Earl's Court
A3218
Old Brompton Rd
South Kensington
A309
H
The Royal Marsden
A3220
E

Network routing protocols (OSPF, BGP, RIP)



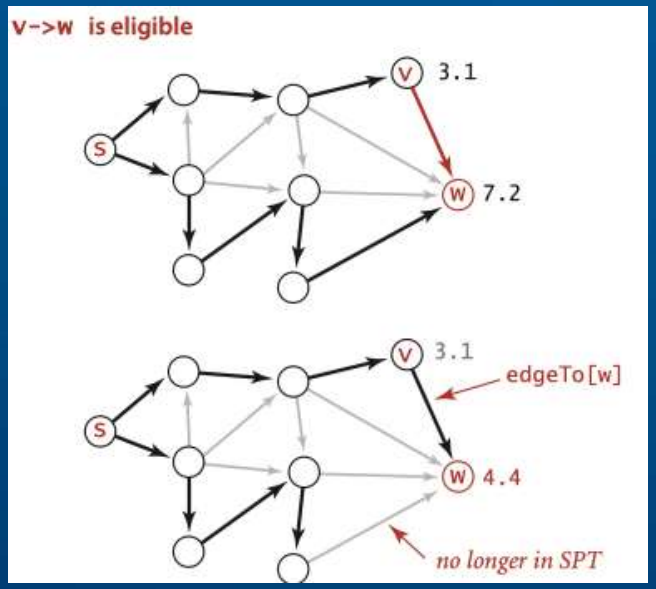
Edge relaxation

To relax an edge v→w means to test whether the best known way from s to w is to go from s to v, then take the edge from v to w, and, if so, update our data structures to indicate that to be the case.

Relax edge e=v→w.

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

```
private void relax(DirectedEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```



Dijkstra's algorithm

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf. ▪ Each edge e = v→w is relaxed exactly once (when v is relaxed), leaving distTo[w] ≤ distTo[v] + e.weight().

▪ Inequality holds until algorithm terminates because:

– distTo[w] cannot increase ← distTo[] values are monotone decreasing. We choose lowest distTo[] value at each step (and edge weights are nonnegative)

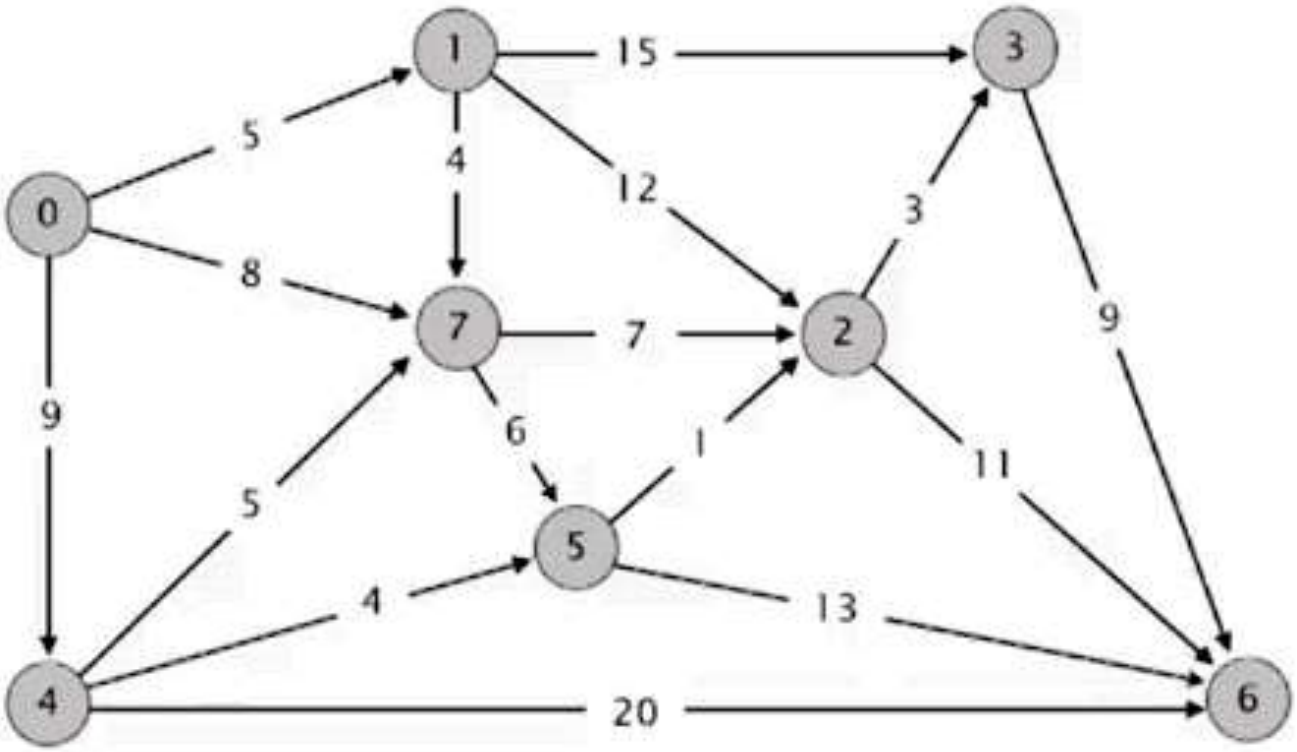
– distTo[v] will not change ←we will change distTo[w]

▪ Thus, upon termination, shortest-paths optimality conditions hold.

Explenation

Assign to every node a tentative weight : set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.

For the current node, consider all of its unvisited neighbours and calculate their weight through the current node. Compare the newly calculated weight to the current assigned value and assign the smaller one. For example, if the current node is marked with a weight of 6, and the edge connecting it with a neighbour has weight 2, then the weight will be 6 + 2 = 8. If it was previously marked with a weight greater than 8 then change it to 8. Otherwise, the current value will be kept.



an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0

Running time

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	log V	log V	log V	E log V
d-way heap (Johnson 1975)	$\log_d V$	d $\log_d V$	$\log_d V$	E $\log_{E/V} V$
Fibonacci heap (Fredman-Tarjan 1984)	1 †	log V †	1 †	E + V log V

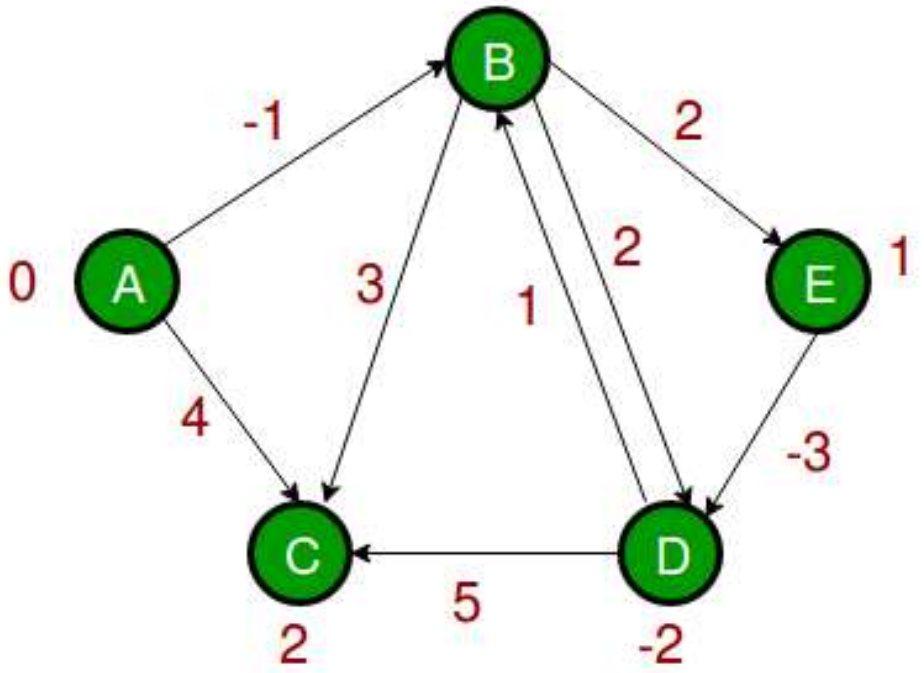
Bellman–Ford algorithm

- The graph may contain negative weight edges. Bellman-Ford works Graphs with negative weight edges .It is slower than Dijkstra's algorithm for the same problem.The algorithm was first proposed by Alfonso Shimbel , but is instead named after Richard Bellman and Lester Ford Jr. , who published it in 1956 and 1958, respectively. Edward F. Moore also published a variation of the algorithm in 1959, and for this reason it is also sometimes called the Bellman–Ford–Moore algorithm.
- Running time: O(|V|*|E|)
- Best case complexity O(|E|)

Explenation

- Like Dijkstra's algorithm, Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path. However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes all the edges, and does this $|V| - 1$ times. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.

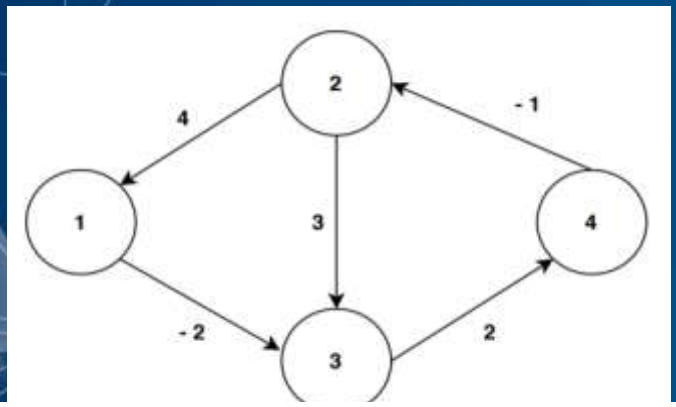
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



Floyd-Warshall algorith

- The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.
- The main idea here is to use a matrix(2D array) that will keep track of the next node to point if the shortest path changes for any pair of nodes. Since there are v vertices, the matrix will have v rows and v columns
- For the Floyd-Warshall algorithm the graph can have negative-weight edges but no negative-weight cycles.

- The time complexity for Floyd Warshall Algorithm is **$O(V^3)$**



1)
$$\begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{bmatrix}$$

$distance[i][j] > distance[i][k] + distance[k][j] \implies distance[1][1] > distance[1][1] + distance[1][1] \implies 0 > 0 + 0 \implies$ FALSE

$distance[i][j] > distance[i][k] + distance[k][j] \implies distance[1][2] > distance[1][1] + distance[1][2] \implies 0 > 0 + \infty \implies$ FALSE

$distance[i][j] > distance[i][k] + distance[k][j] \implies distance[2][3] > distance[2][1] + distance[1][3] \implies 3 > 4 + -2 \implies 3 > 2 \implies$ TRUE

2)
$$\begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{bmatrix}$$

3)
$$\begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ 3 & -1 & \infty & 0 \end{bmatrix}$$

$distance[i][j] > distance[i][k] + distance[k][j] \implies distance[4][1] > distance[4][2] + distance[2][1] \implies \infty > -1 + 4 \implies \infty > 3 \implies$ TRUE

output matrix:
$$\begin{bmatrix} 0 & -1 & -2 & 0 \\ 4 & 0 & 2 & 4 \\ 5 & 1 & 0 & 2 \\ 3 & -1 & 1 & 0 \end{bmatrix}$$