SULEYMAN SULEYMANZADE

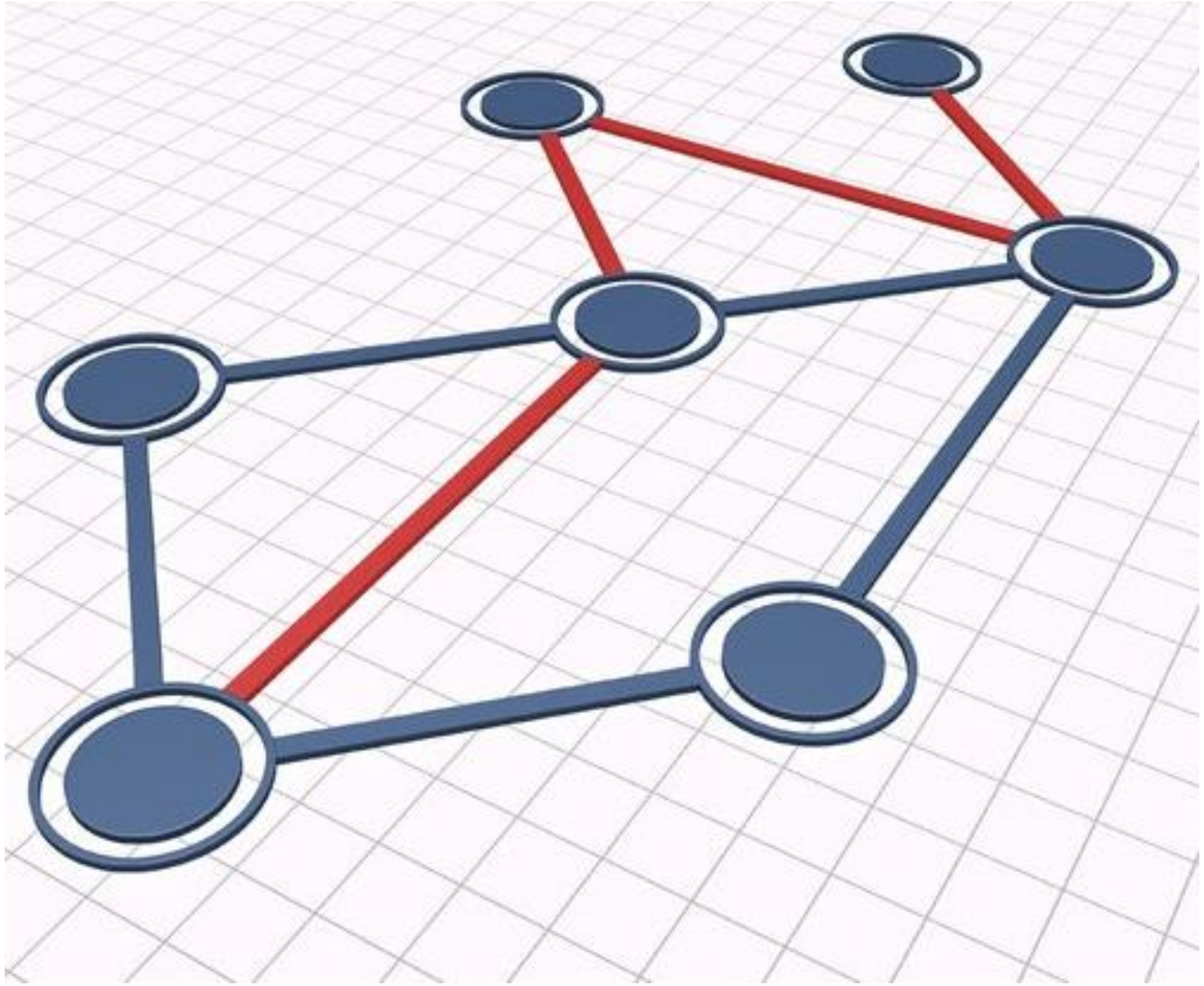
GRAPHS: SHORTEST PATHS: SSSP, ASAP

DATA STRUCTURES AND ALGORITHMS

SSSP AND ASAP

SSSP and ASAP

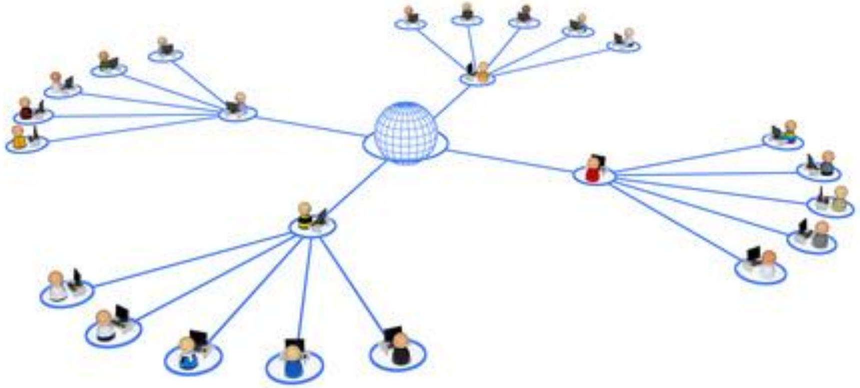
- Overview
- Applications
- Single Source Shortest Path (SSSP)
  - Dijkstra's algorithm
  - Dijkstra's algorithm implementation
  - Bellman Ford's Algorithm
  - Bellman Ford's algorithm implementation
- All Pairs All Paths (ASAP)
  - Floyd Warshall's algorithm
  - Implementation



USAGE OF SHORTEST PATH

Examples

- Navigation Applications
- Game development
- Pipeline automation
- Ballistic applications
- Network developing
  - In routing Bellman's Ford algorithm used



DEFINITIONS

Single-destination shortest-paths problem: Find a shortest path to a given ***destination*** vertex t from each vertex . By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

Single-pair shortest-path problem: Find a shortest path from u to for given vertices u and . If we solve the single-source problem with source vertex u, we solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

All-pairs shortest-paths problem: Find a shortest path from u to for every pair of vertices u and . Although we can solve this problem by running a singlesource algorithm once from each vertex, we usually can solve it faster. Additionally, its structure is interesting in its own right.

DIJKSTRA ALGORITHM

SSSP SINGLE SOURCE SHORTEST PATH

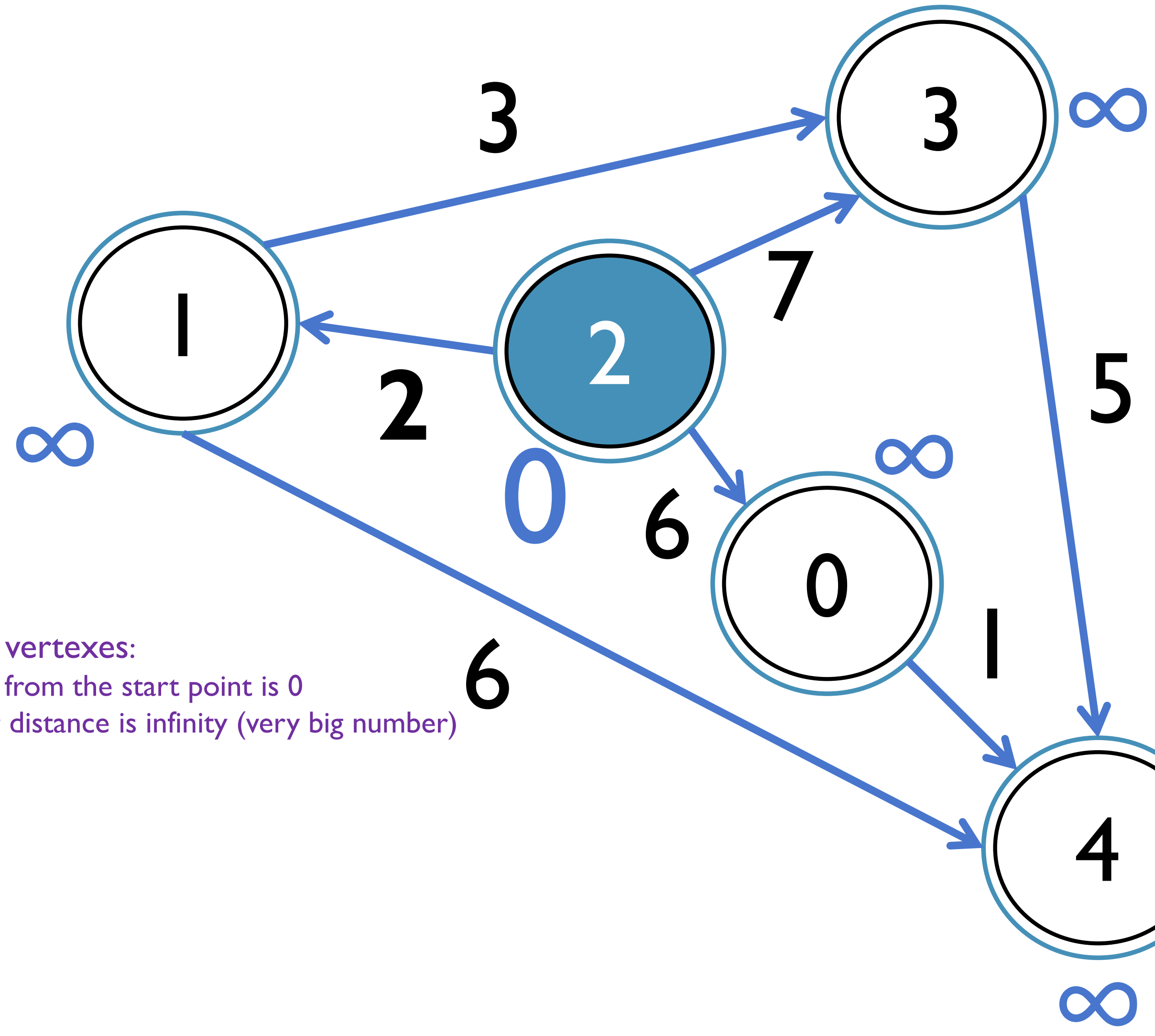
ALGORITHM

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u 2 V S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

Dijkstra's algorithm relaxes edges as shown in Figure Line 1 initializes the d and values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that Q D V S at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V ; since S D ; at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from Q D V S and line 6 adds it to set S, thereby maintaining the invariant. (The first time through this loop, u D s.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in V S. Then, lines 7–8 relax each edge .u; / leaving u, thus updating the estimate :*d* and the predecessor : if we can improve the shortest path to found so far by going through u.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

ALGORITHM



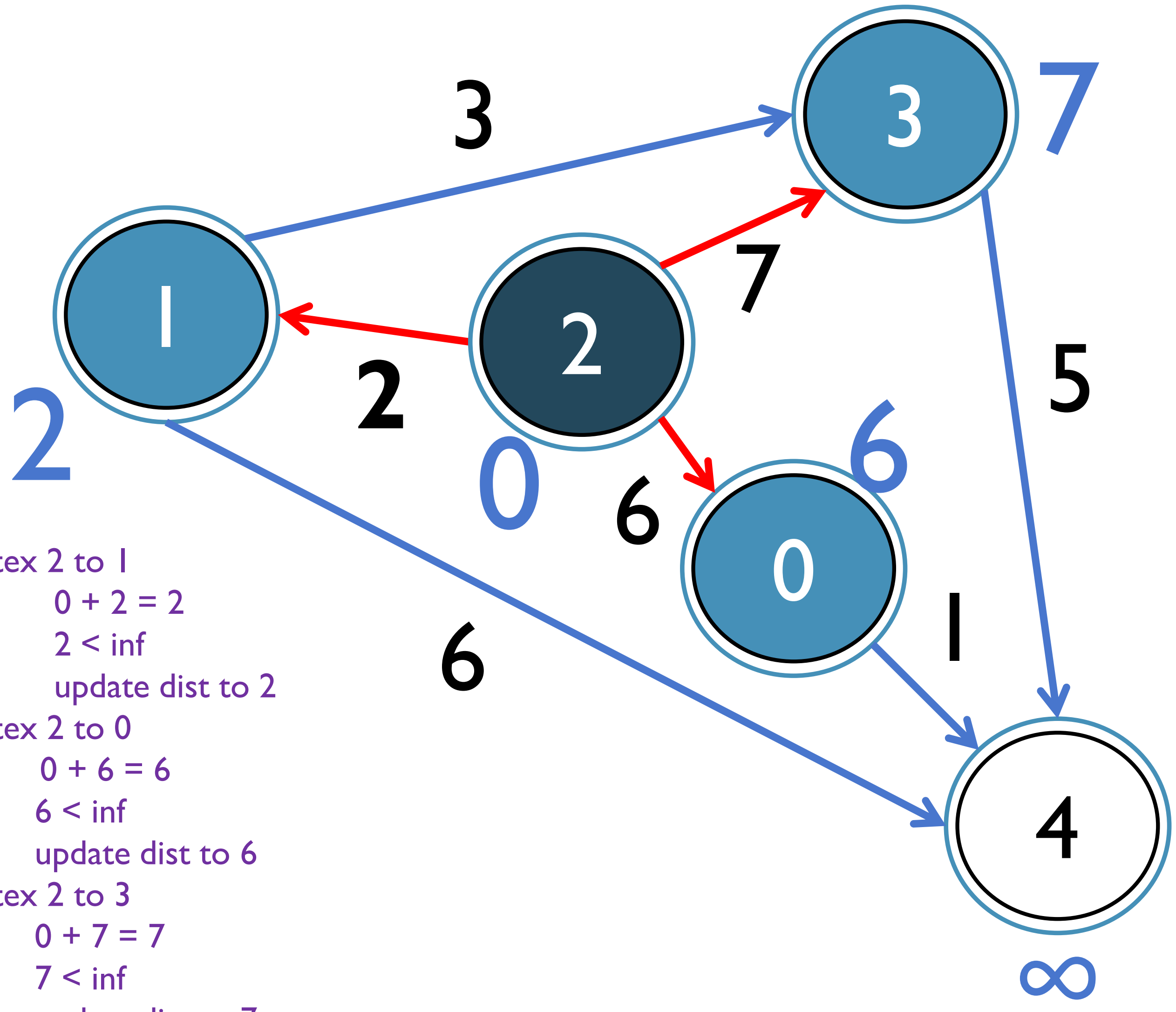
Initializing vertexes:
Distance from the start point is 0
all other distance is infinity (very big number)

Distance table

vertex	distance
0	∞
1	∞
2	0
3	∞
4	∞

Example of SSSP from vertex "2" to all other vertexes

ALGORITHM



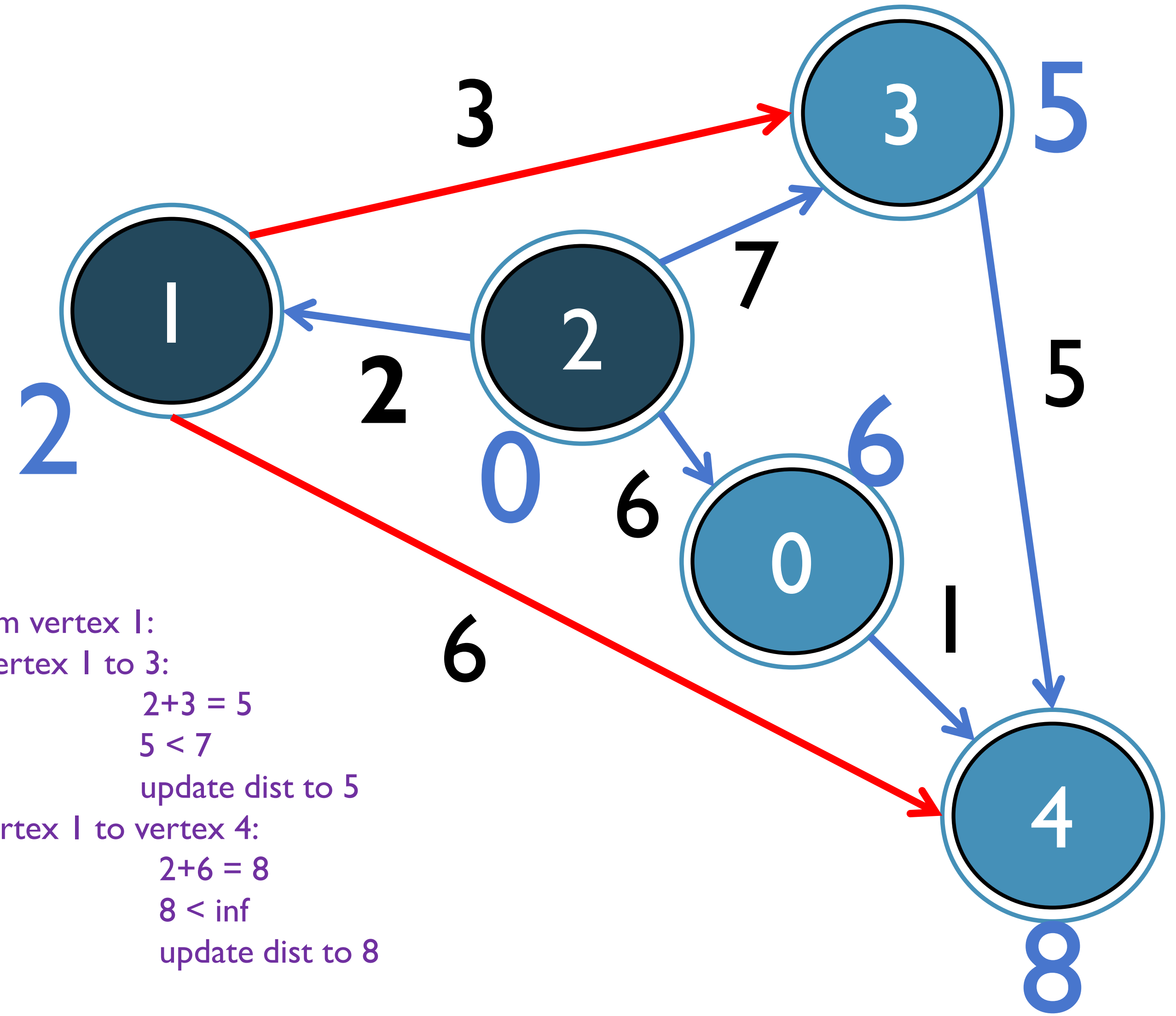
From vertex 2 to 1
0 + 2 = 2
2 < inf
update dist to 2
From vertex 2 to 0
0 + 6 = 6
6 < inf
update dist to 6
From vertex 2 to 3
0 + 7 = 7
7 < inf
update dist to 7

Distance table

vertex	distance
0	6
1	2
2	0
3	7
4	∞

Example of SSSP from vertex “2” to all other vertexes

ALGORITHM



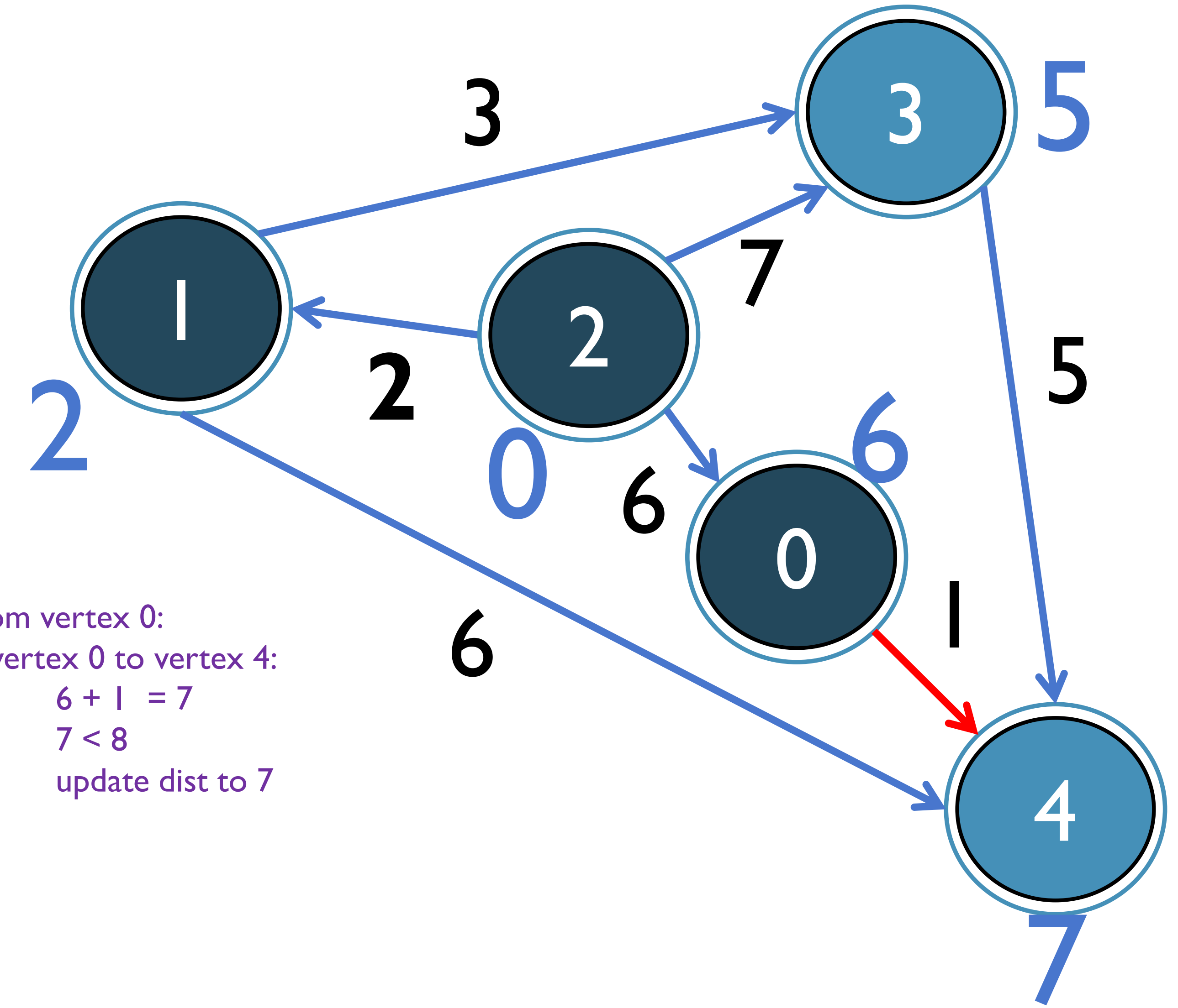
Check from vertex 1:
from vertex 1 to 3:
2+3 = 5
5 < 7
update dist to 5
from vertex 1 to vertex 4:
2+6 = 8
8 < inf
update dist to 8

Distance table

vertex	distance
0	6
1	2
2	0
3	5
4	8

Example of SSSP from vertex "2" to all other vertexes

ALGORITHM



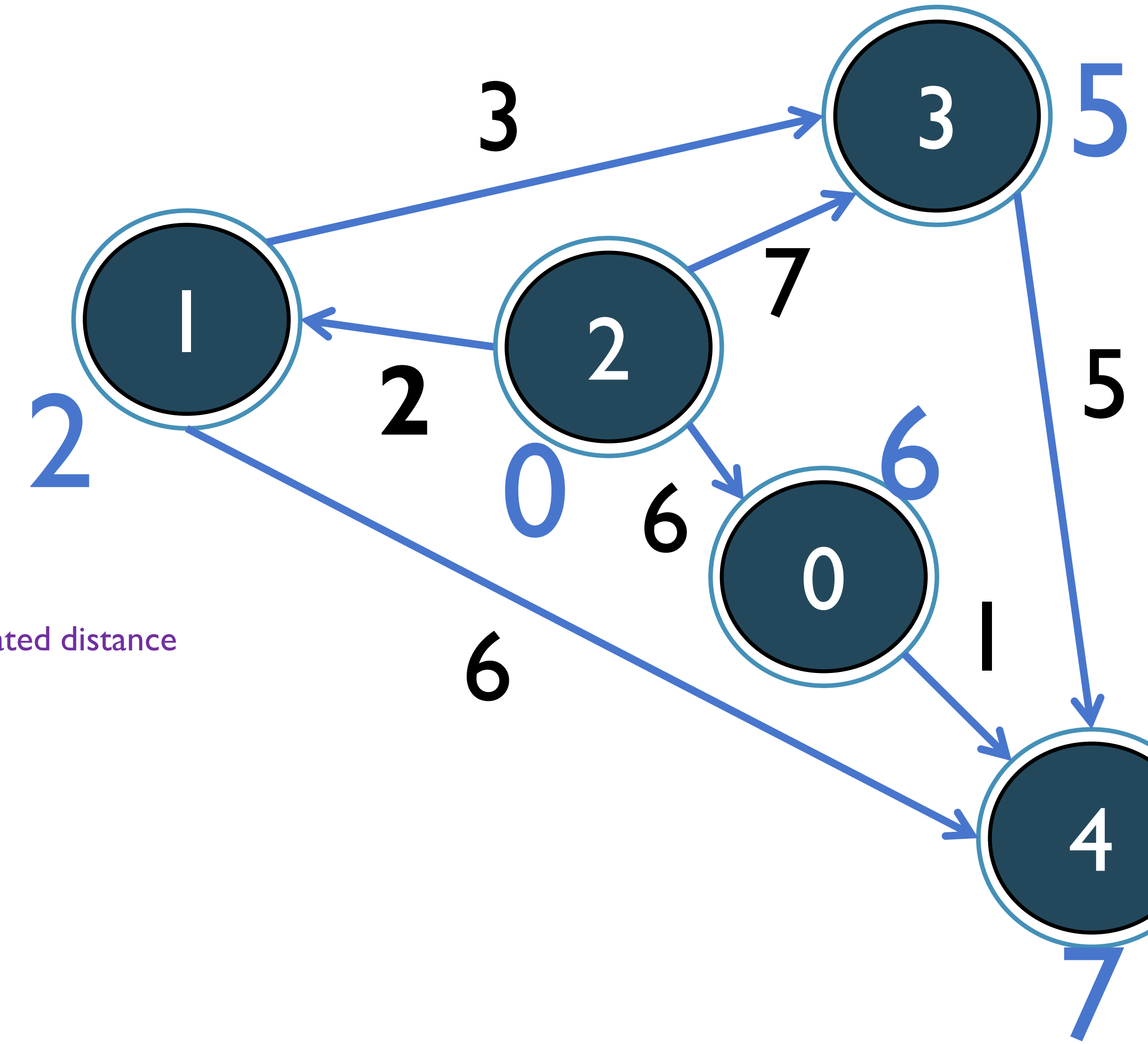
Check from vertex 0:
from vertex 0 to vertex 4:
6 + 1 = 7
7 < 8
update dist to 7

Distance table

vertex	distance
0	6
1	2
2	0
3	5
4	7

Example of SSSP from vertex "2" to all other vertexes

ALGORITHM



Final updated distance

Distance table

vertex	distance
0	6
1	2
2	0
3	5
4	7

Example of SSSP from vertex "2" to all other vertexes

DIJKSTRA IMPLEMENTATION

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<queue>
#include<functional>
using namespace std;

const int INF = 1e9 + 7;

vector<pair<int, int>> graph[100000];
int ans[100000];
int pr[100000];     //prev

int main() {
    //insert graphs

    for (int i = 0; i < n; i++) {
        ans[i] = INF;
        pr[i] = -1;
    }

    ans[start] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>,
    greater<pair<int, int>>> q;

    q.push({0, start});

    while (!q.empty()) {
        pair<int, int> c = q.top();
        q.pop();

        int dst = c.first, v = c.second;

        if (ans[v] < dst) {
            continue;
        }

        for (pair<int, int> e: graph[v]) {
            int u = e.first, len_vu = e.second;

            int n_dst = dst + len_vu;
            if (n_dst < ans[u]) {
                ans[u] = n_dst;
                pr[u] = v;
                q.push({n_dst, u});
            }
        }
    }

    vector<int> path;

    int cur = end;
    path.push_back(cur);

    while (pr[cur] != -1) {
        cur = pr[cur];
        path.push_back(cur);
    }

    reverse(path.begin(), path.end());

    cout << "Shortest path" << start + 1 <<" and
         << end + 1 < " is: "<< endl;

    for (int v: path) {
        cout << v + 1 << ", ";
    }
}
```

BELLMAN FORD ALGORITHM

ALGORITHM, IMPLEMENTATIONS, USAGE

BELMAN FORD ALGORITHM

- The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph G D .V;E/ with source s and weight function w: E→R, the
- Bellman-Ford algorithm returns a boolean value indicating whether or not there is
- a negative-weight cycle that is reachable from the source. If there is such a cycle,
- the algorithm indicates that no solution exists. If there is no such cycle, the
- algorithm produces the shortest paths and their weights.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

THE SIMPLEST BELLMAN'S FORD ALGORITHM

```
struct edge {
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve() {
    vector<int> d (n, INF);
    d[v] = 0;
    for (int i=0; i<n-1; ++i)
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                d[e[j].b] = min (d[e[j].b], d[e[j].a] + e[j].cost);
// print D to the screen
}
```

EXTENDED BELLMAN'S FORD ALGORITHM,

```
void solve() {
    vector<int> d (n, INF);
    d[v] = 0;
    for (;;) {
        bool any = false;
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    any = true;
                }
            if (!any)  break;
    }
    // print d to the screen
}
```

BELLMAN'S FORD ALGORITHMS WITH PATH'S RESTORING

```
void solve() {

  vector<int> d (n, INF);
  d[v] = 0;
  vector<int> p (n, -1);
  for (;;) {
    bool any = false;
    for (int j=0; j<m; ++j)
      if (d[e[j].a] < INF)
        if (d[e[j].b] > d[e[j].a] + e[j].cost) {
            d[e[j].b] = d[e[j].a] + e[j].cost;
            p[e[j].b] = e[j].a;
            any = true;
        }
      if (!any)  break;
   }
```

```
if (d[t] == INF)
  cout << "No path from "<<v<<" to "<<t<<".";
else {
  vector<int> path;
  for (int cur=t; cur!=-1; cur=p[cur])
    path.push_back (cur);
  reverse (path.begin(), path.end());
  cout << "Path from " << v << " to " << t << ": ";
  for (size_t i=0; i<path.size(); ++i)
    cout << path[i] << ' ‘;
  }
}
```

SULEYMAN SULEYMANZADE

FLOYD-WARSHALL ALGORITHM

DATA STRUCTURES AND ALGORITHMS

FLOYD-WARSHALL

FLOYD-WARSHALL(W)

$n = W.rows$

$D^{(0)} = W$

for $k = 1$ **to** n

    let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix

    for $i = 1$ **to** n

        for $j = 1$ **to** n

            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

return $D^{(n)}$

FLOYD-WARSHALL ALGORITHM'S IMPLEMENTATION

```
for (int k=0; k<n; ++k)
    for (int i=0; i<n; ++i)
        for (int j=0; j<n; ++j)
            d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

$O(n^3)$

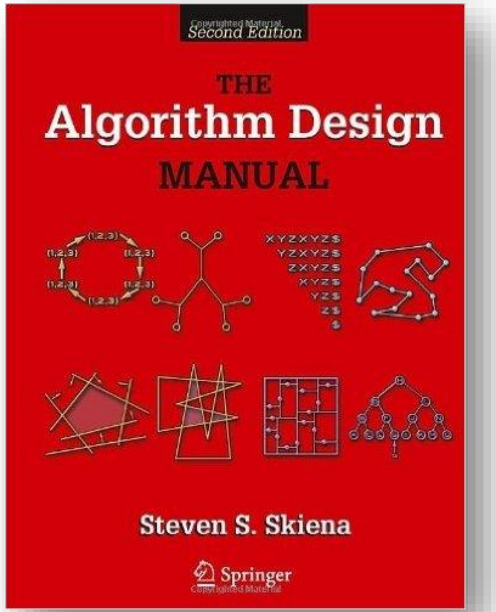
- Important: For any $d[i][j]$ = 0
- If no edges between u and v then $d[u][v] = \infty$ (some very big number)
- If there is negative cycle in adjacency matrix then there could be results like $\infty - 1$ or $\infty - 2$
- For

```
for (int k=0; k<n; ++k)
    for (int i=0; i<n; ++i)
        for (int j=0; j<n; ++j)
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

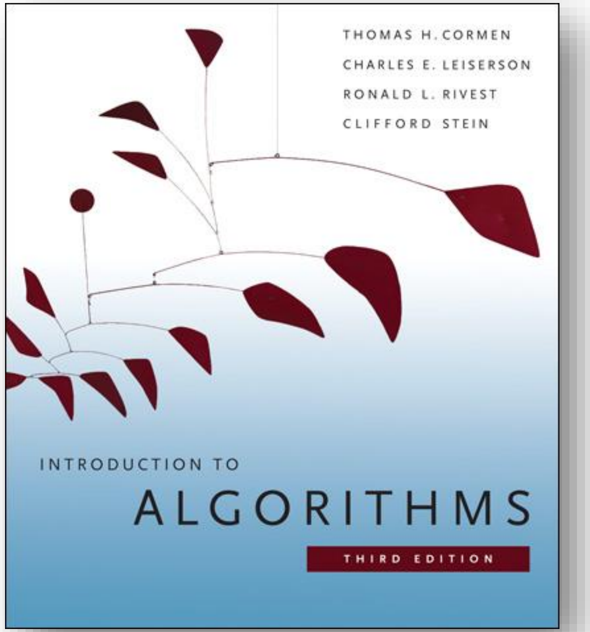
FLOYD-WARSHALL IMPLEMENTATION

```
#include <iostream>
using namespace std;
const int INF = 1e9 + 7;
int dp[1000][1000];
int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dp[i][j] = INF;
        }
    }
    for (int i = 0; i < n; i++) {
        dp[i][i] = 0;
    }
    for (int i = 0; i < m; i++) {
        int u, v, len;
        cin >> u >> v >> len;
        u--, v--;
        dp[u][v] = dp[v][u] = len;
    }
    for (int k = 0; k < n; k++) {         //current node
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
            }
        }
    }
    //dp updated with shortest paths
}
```

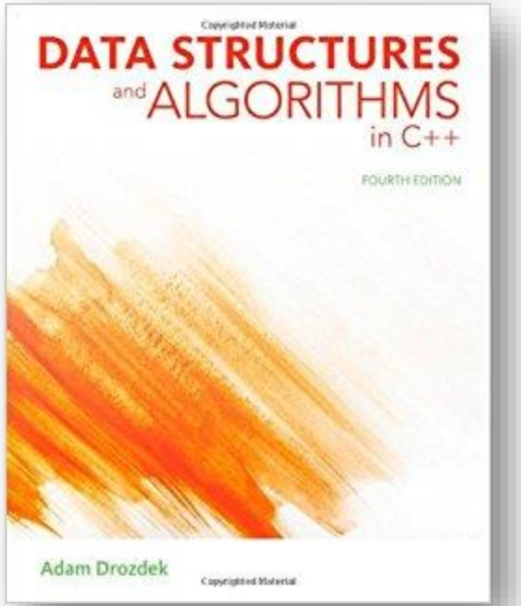
LITERATURE



Stieven Skienna
Algorithms design manual
Chapter 5: Graph Traversal
Page 145



Thomas H. Cormen
Introduction to Algorithms
Chapter VI, 24 Graph Algorithms, Single source shortest path
Page 643.



Adam Drozdek
Data structures and Algorithms in C++
Chapter 8: Graphs
Page 391