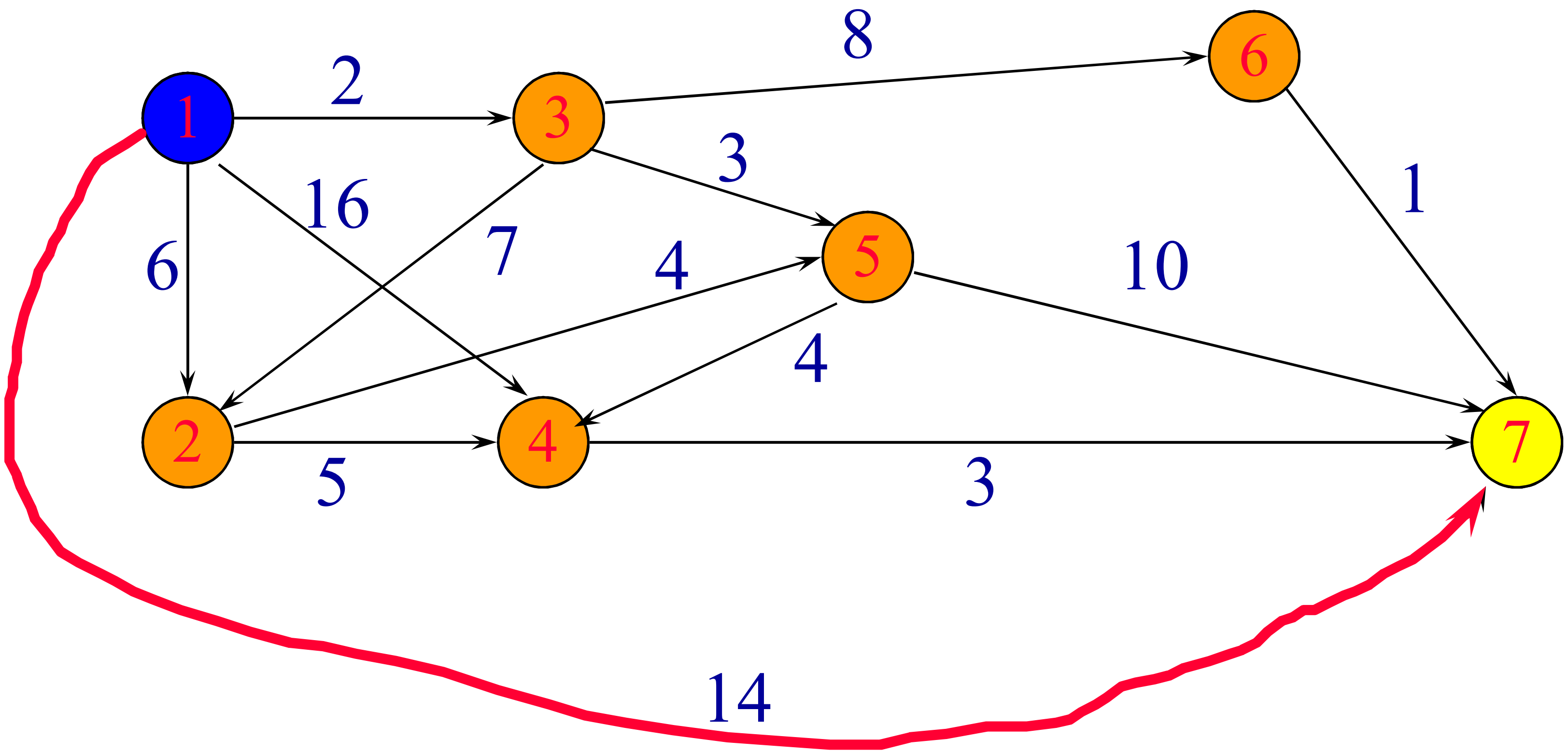
Shortest Paths
Part I

Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

Shortest Path Problems

- Directed weighted graph.
- Path length is the sum of weights of edges on a path.
- The vertex at which the path begins is the source vertex.
- The vertex at which the path ends is the destination vertex.
- If there is more than one path from the source to the destination, which is the shortest path?

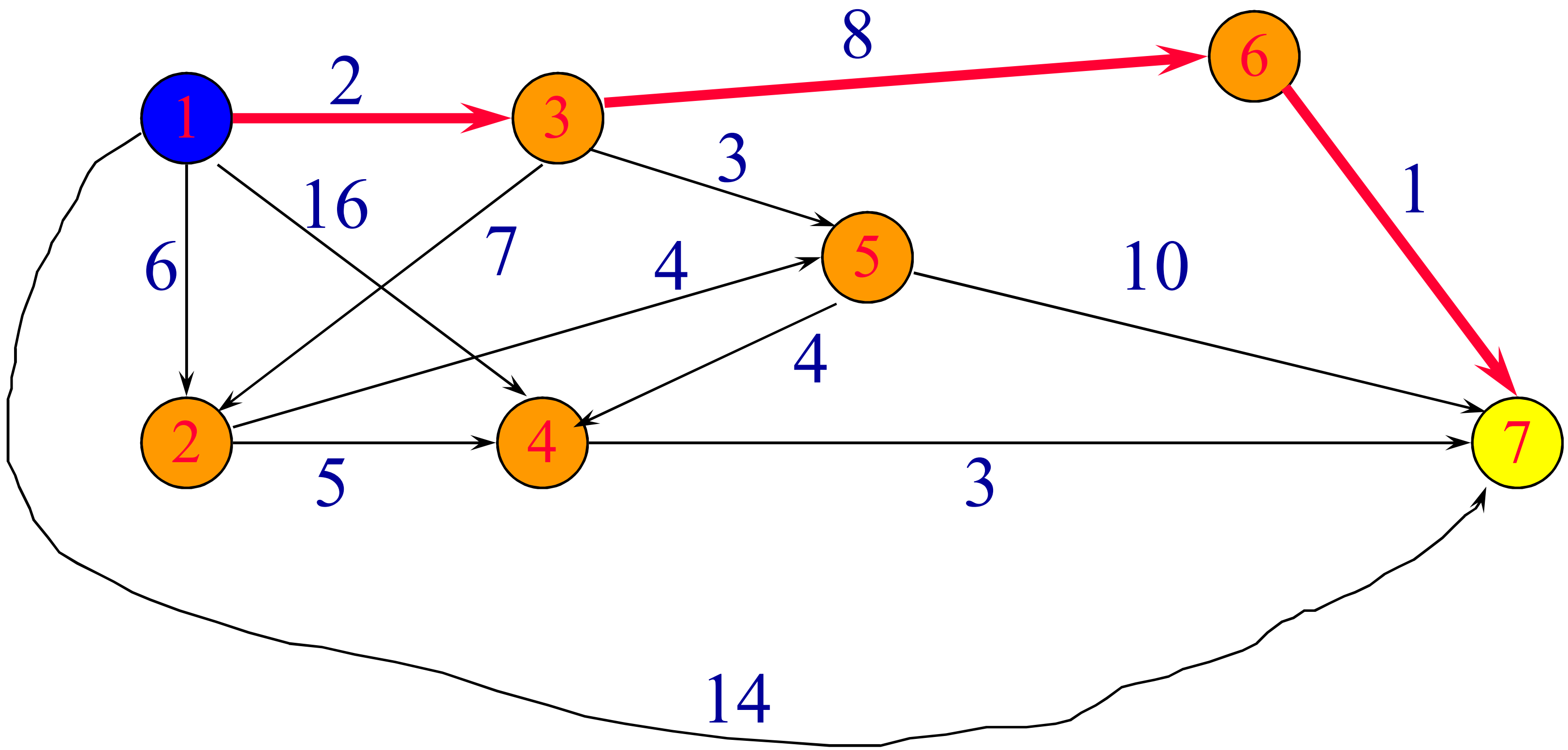
Example



A path from 1 to 7.

Path length is 14.

Example



Another path from 1 to 7.

Path length is 11.

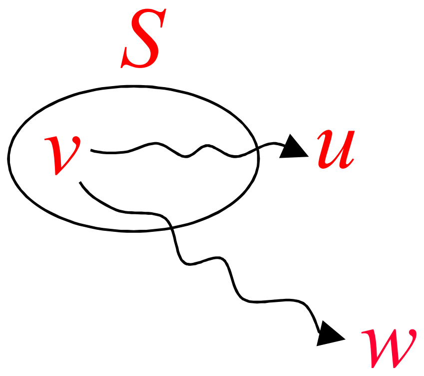
Single Source/All Destinations

- Problem: Determine a shortest path from a source vertex *v* to each of the remaining vertices of *G*
  - We assume non-negative edge weights

Single Source/All Destinations (cont.)

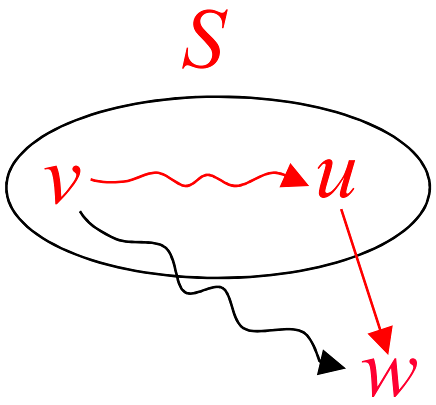
- Solution: Use a greedy algorithm called Dijkstra's algorithm
  - Let S denote the set of vertices, including v, whose shortest paths have been found
  - For w not in S, let $dist[w]$ be the length of the shortest path starting from v, going through only the vertices that are in S, and ending at w
  - Generate shortest paths in non-decreasing order of length

Observations



- If the next shortest path is to *u*, then the path begins at *v*, ends at *u*, and goes through only those vertices in *S*
- *u* is chosen so that it has the minimum distance, *dist*[*u*], among all the vertices not in *S*

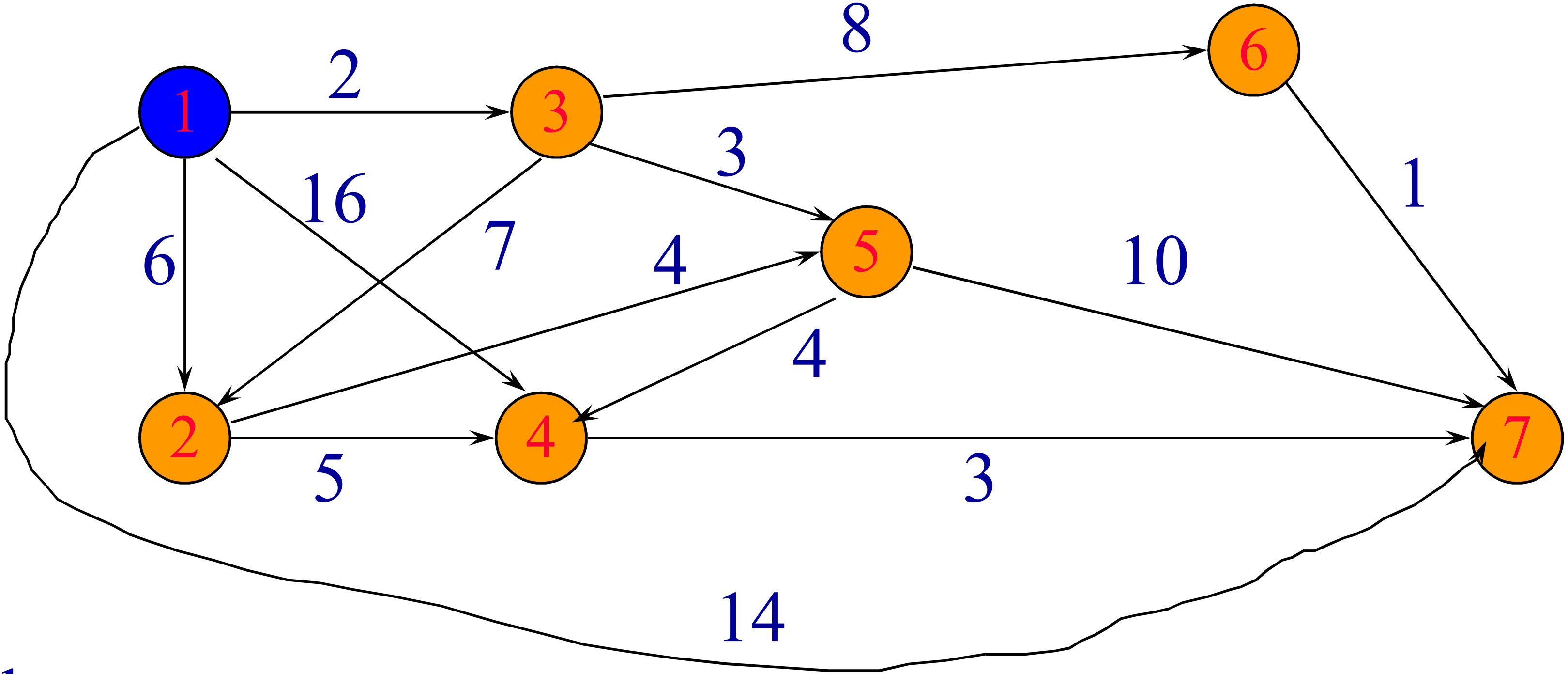
Observations (cont.)



- u becomes a member of S
- This may result in a shorter path from v to $w \notin S$.
- This shorter path goes through u, and the length of this path is $dist[u] + length(<u,w>)$
- For each w that is adjacent from u

$$dist[w] = \min(dist[w],\ dist[u] + length(<u,w>))$$

Single Source/All Destinations



Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

Single Source/All Destinations

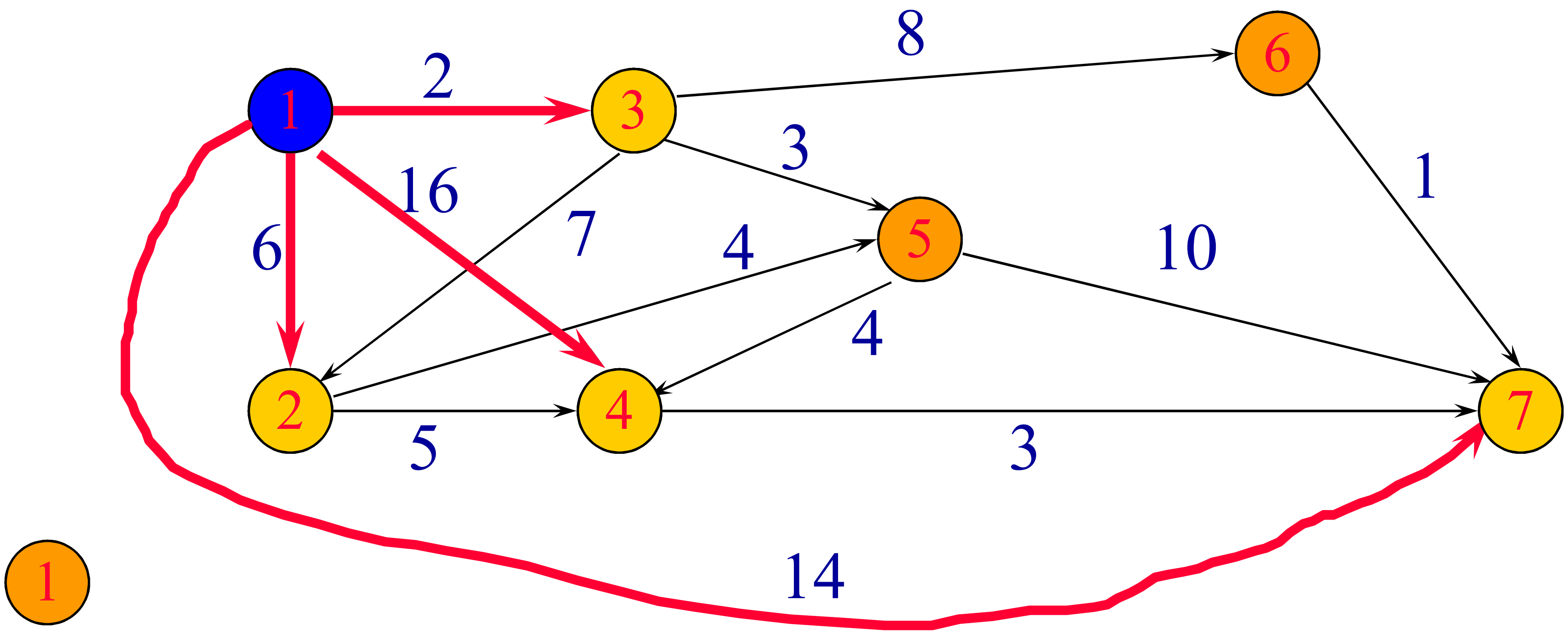
Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

- Each path (other than first) is a one edge extension of a previous path.
- Next shortest path is the shortest one edge extension of an already generated shortest path.

Single Source/All Destinations

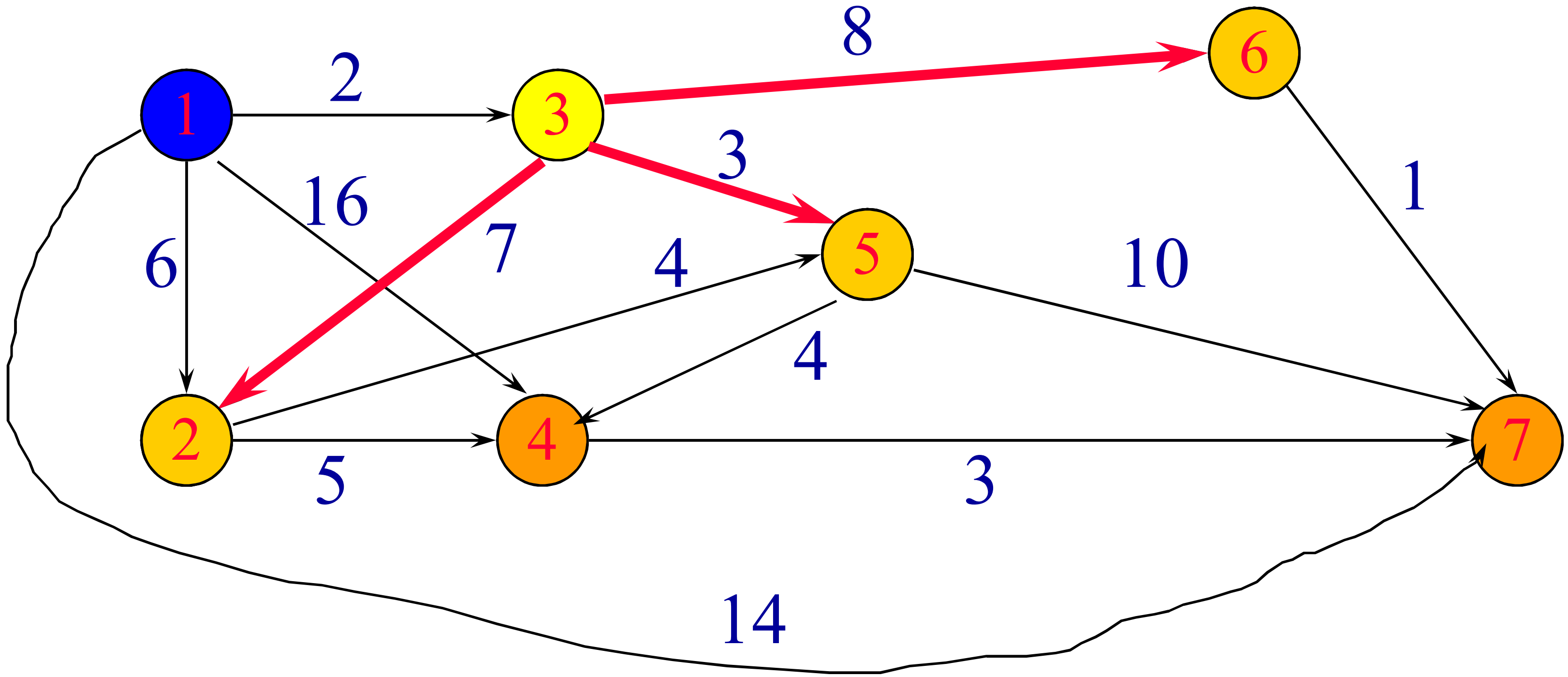
- Let dist[i] be the length of a shortest one edge extension of an already generated shortest path, the one edge extension ends at vertex i.
- The next shortest path is to an as yet unreached vertex for which the dist[] value is least.
- Let prev[i] be the vertex just before vertex i on the shortest one edge extension to i.

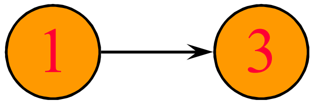
Single Source/All Destinations



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	16	-	-	14
prev	-	1	1	1	-	-	1

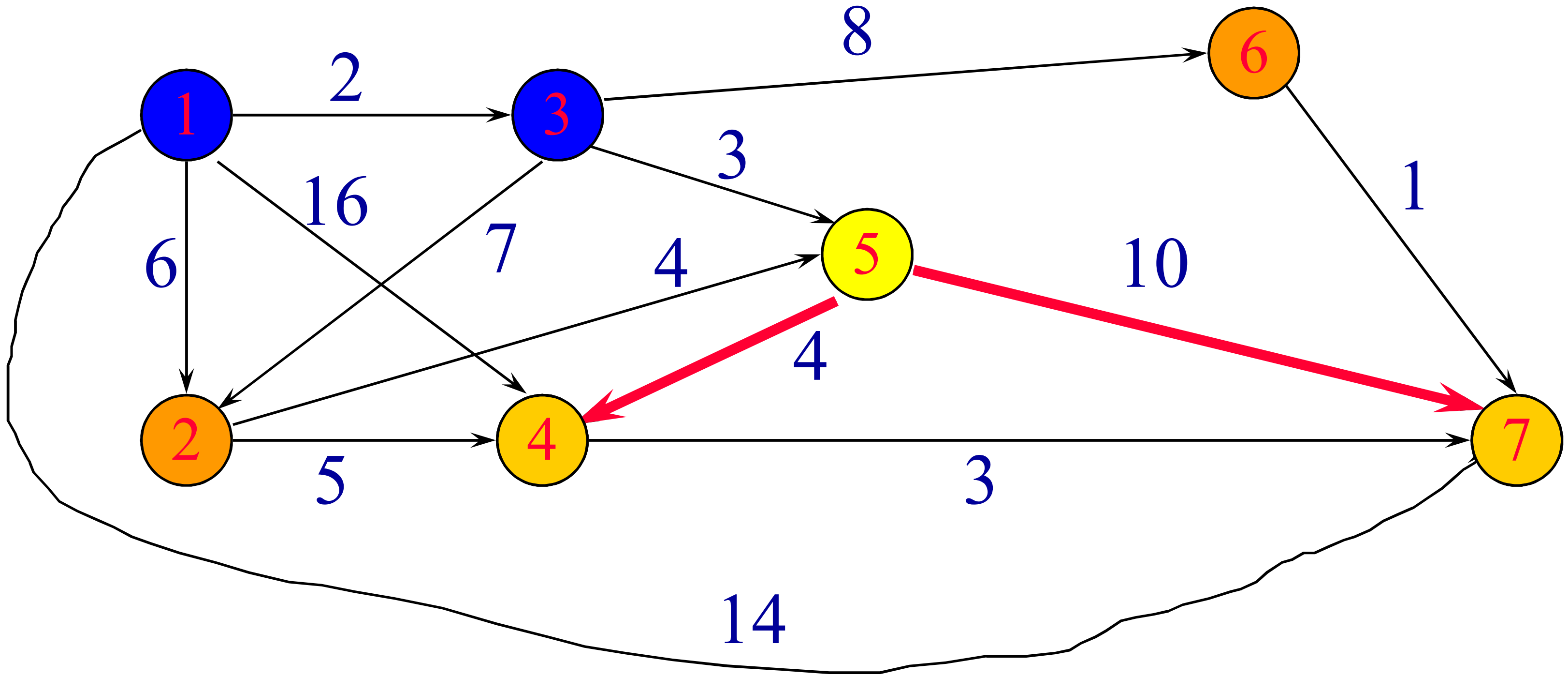
Single Source/All Destinations

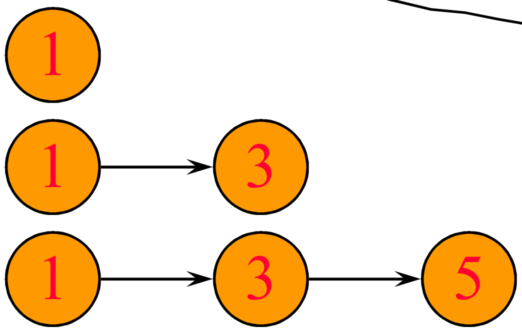




		[1]	[2]	[3]	[4]	[5]	[6]	[7]
after	dist	0	6	2	16	5	10	14
	prev	-	1	1	1	3	3	1
before	dist	0	6	2	16	-	-	14
	prev	-	1	1	1	-	-	1

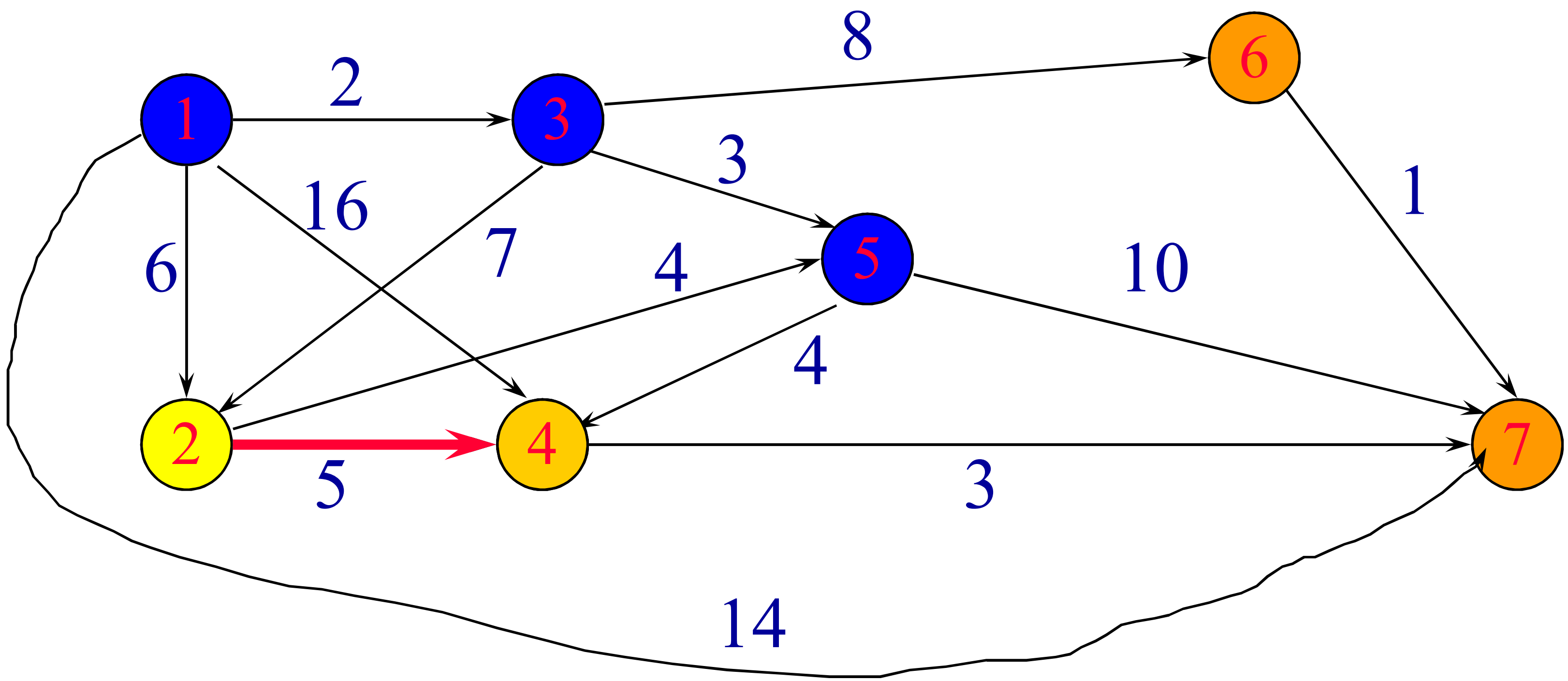
Single Source/All Destinations

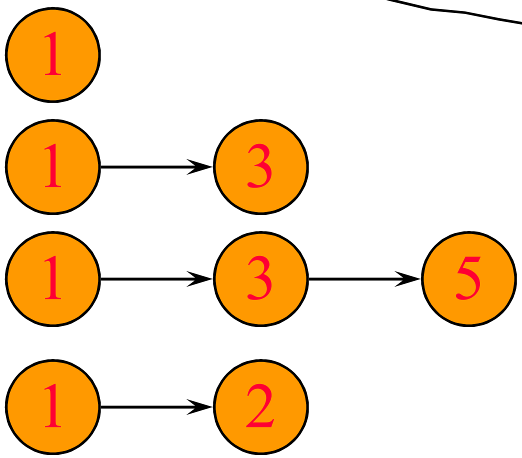




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	14
prev	-	1	1	5	3	3	1
dist	0	6	2	16	5	10	14
prev	-	1	1	1	3	3	1

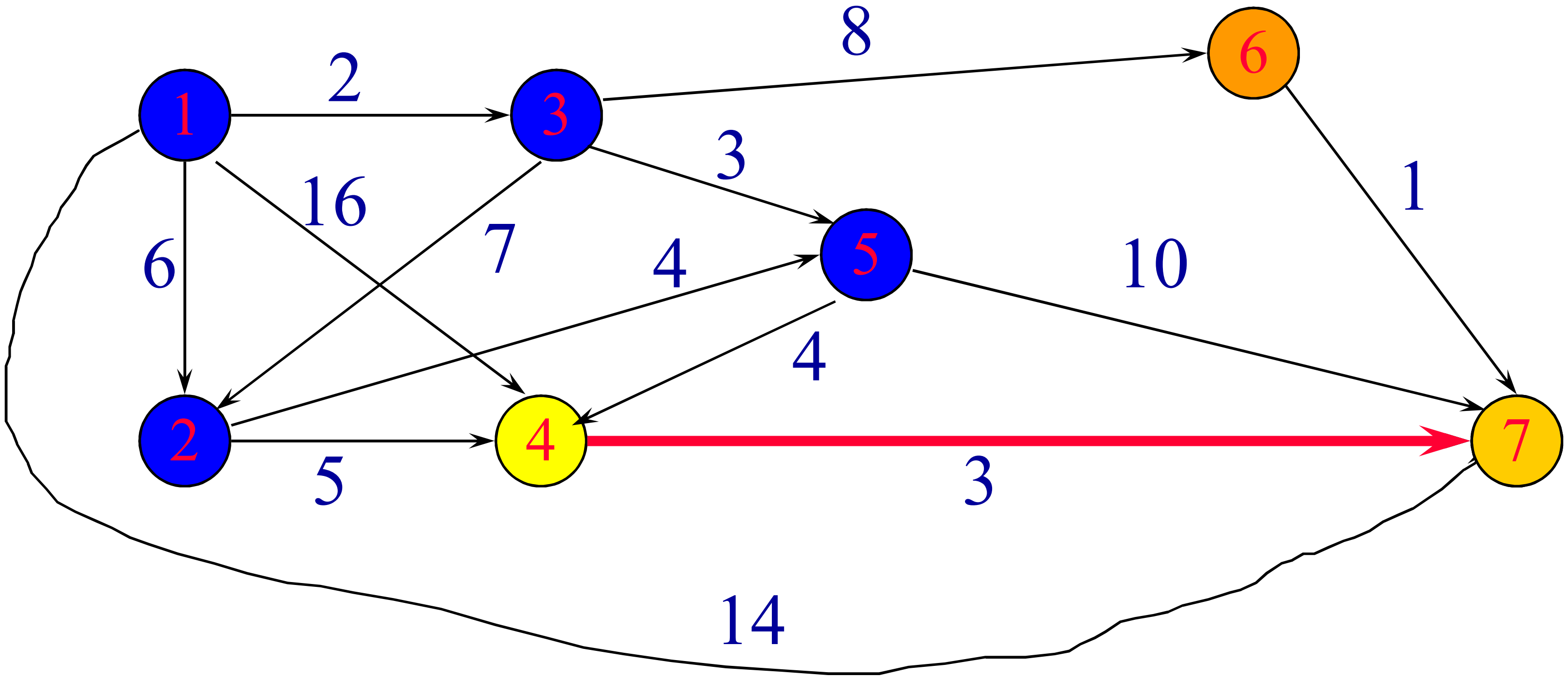
Single Source/All Destinations

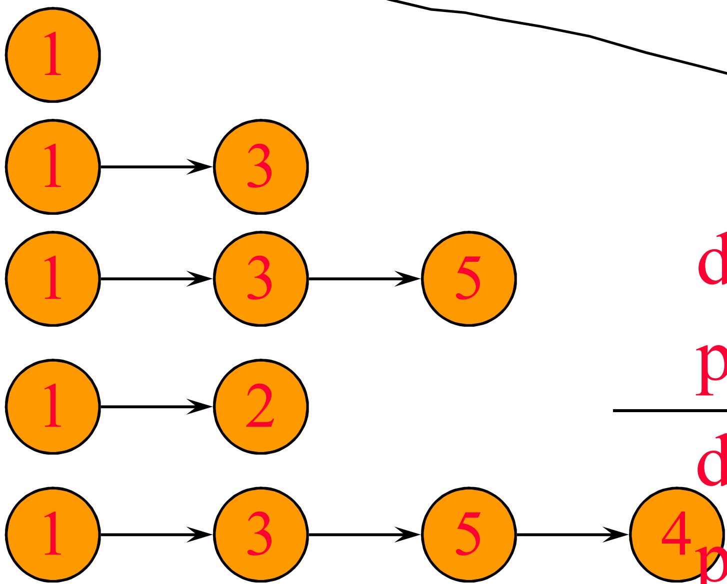




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	14
prev	-	1	1	5	3	3	1
dist	0	6	2	9	5	10	14
prev	-	1	1	1	3	3	1

Single Source/All Destinations

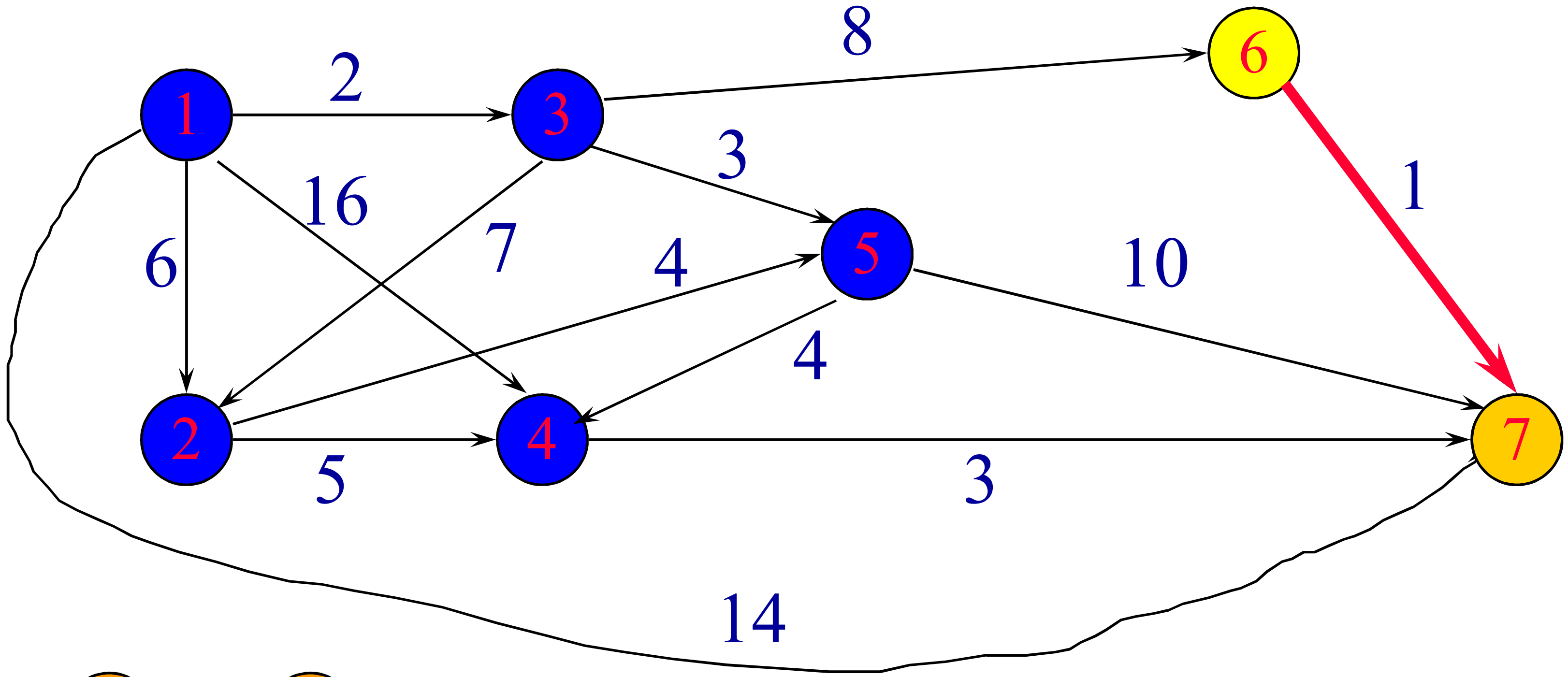


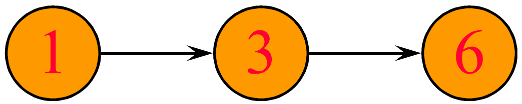


	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	12
prev	-	1	1	5	3	3	4

dist	0	6	2	9	5	10	14
prev	-	1	1	5	3	3	1

Single Source/All Destinations





	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	11
prev	-	1	1	5	3	3	6
dist	0	6	2	9	5	10	12
prev	-	1	1	5	3	3	4

Single Source/All Destinations

Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	6	2	9	5	10	11
-	1	1	5	3	3	6

Single Source/Single Destination

Terminate single source all destinations algorithm as soon as shortest path to desired vertex has been generated.

Data Structures for Dijkstra's Algorithm

- The described single source/all destinations algorithm is known as Dijkstra's algorithm.
- Implement dist[] and prev[] as 1D arrays.
- Keep a linear list L of reachable vertices to which shortest path is yet to be generated.
- Select and remove vertex u in L that has smallest dist[] value.
- Update dist[] and prev[] values of vertices adjacent to u.

length[i][j]

- length[i][j] = the length of the edge <i, j>
- If <i, j> is not an edge of the graph and $i \neq j$, length[i][j] may be set to some large number LARGE
  - LARGE must be larger than any of the values in the length matrix
  - LARGE must be chosen so that the statement dist[u] + length[u][w] does not produce an overflow

```cpp
void MatrixWDigraph::ShortestPath(const int n, const int v)
{// dist[j], 0<= j < n, is set to the length of
 // the shortest path from v to j
 // in a digraph G with n vertices and
 // edge lengths given by length[i][j].
    for (int i = 0; i < n; i++) { // initialize
        s[i] = false;
        dist[i] = length[v][i];
    }
    s[v] = true;
    dist[v] = 0;

    for (i = 0; i < n-2; i++) {
    // determine n-1 paths from vertex v
        int u = Choose(n); // Choose(n) returns a value u such that:
                           // dist[u] = minimum dist[w],
                           // where s[w] = false
        s[u] = true;

        for (int w = 0; w < n; w++)
            // For adjacency lists, do the following
            // only for w that is adjacent from u,
            // i.e., u's adjacency list
            if (s[w] == false)
                dist[w] = min(dist[u] + length[u][w], dist[w]);
    } // end of for (i = 0; ...)
}
```

Complexity

- Select next destination vertex: O(n)
- Update dist[] and prev[] values
  - O(out-degree) when adjacency lists are used.
  - O(n) when adjacency matrix is used.
- Selection and update done once for each vertex to which a shortest path is found.
- Total time is $O(n^2)$.

Complexity

- When a min heap of dist[] values is used in place of the linear list L of reachable vertices, total time is O((n+e) log n)
  - n + e?
  - O(n) remove min operations
  - O(e) decrease key (dist[] value) operations
- When e is $O(n^2)$, using a min heap is worse than using a linear list.
- When a Fibonacci heap is used, the total time is O(n log n + e).

Decrease key in O(log n)

- To avoid O(n) look-up in decrease-key step on a vanilla binary heap, it is necessary to maintain a supplementary index mapping each vertex to the heap's index (and keep it up to date as the priority queue changes), making it take only O(log n) time instead

```cpp
// Program to find Dijkstra's shortest path using
// priority_queue in STL
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

// iPair ==>  Integer Pair
typedef pair<int, int> iPair;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;

public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // prints shortest path from s
    void shortestPath(int s);
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair> [V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}
```

```
/* Looping till priority queue becomes empty (or all
  distances are not finalized) */
while (!pq.empty())
{
    // The first vertex in pair is the minimum distance
    // vertex, extract it from priority queue.
    // vertex label is stored in second of pair (it
    // has to be done this way to keep the vertices
    // sorted distance (distance must be first item
    // in pair)
    int u = pq.top().second;
    pq.pop();

    // 'i' is used to get all adjacent vertices of a vertex
    list< pair<int, int> >::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        // Get vertex label and weight of current adjacent
        // of u.
        int v = (*i).first;
        int weight = (*i).second;

        //  If there is shorted path to v through u.
        if (dist[v] > dist[u] + weight)
        {
            // Updating distance of v
            dist[v] = dist[u] + weight;
            pq.push(make_pair(dist[v], v));
        }
    }
}
```