Chapter Nine

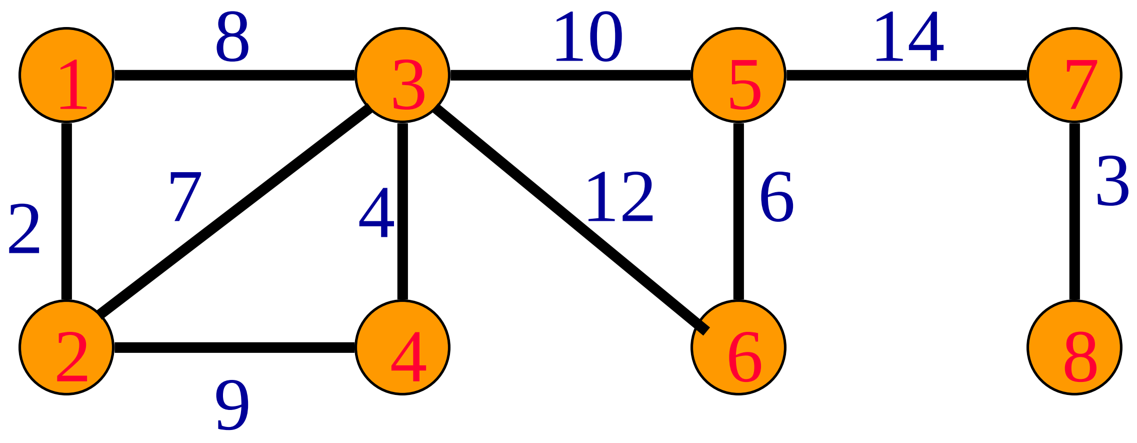
Weighted Trees

9.1 Minimum Spanning Tree
9.2 Shortest Past Problems
9.3 Dijkstra’s Algorithm
(*9.4 Floyd’s Algorithm*)

9.1 Minimum-Cost Spanning Tree

- weighted connected undirected graph
- spanning tree
- cost of spanning tree is sum of edge costs
- find spanning tree that has minimum cost

Example



- Network has 10 edges.
- Spanning tree has only n - 1 = 7 edges.
- Need to either select 7 edges or discard 3.

Edge Selection Strategies

- Start with an n-vertex 0-edge forest. Consider edges in ascending order of cost. Select edge if it does not form a cycle together with already selected edges.
  - Kruskal’s method.
- Start with a 1-vertex tree and grow it into an n-vertex tree by repeatedly adding a vertex and an edge. When there is a choice, add a least cost edge.
  - Prim’s method.

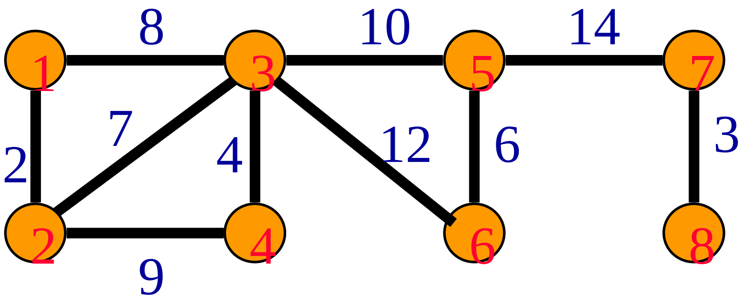
Edge Selection Strategies

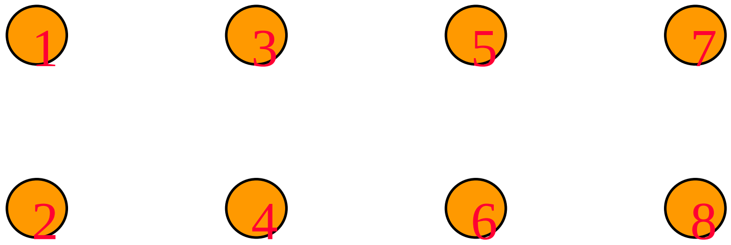
- Start with an n-vertex forest. Each component/tree selects a least cost edge to connect to another component/tree. Eliminate duplicate selections and possible cycles. Repeat until only 1 component/tree is left.
    - Sollin’s method.

Edge Rejection Strategies

- Start with the connected graph. Repeatedly find a cycle and eliminate the highest cost edge on this cycle. Stop when no cycles remain.
- Consider edges in descending order of cost. Eliminate an edge provided this leaves behind a connected graph.

Kruskal’s Method



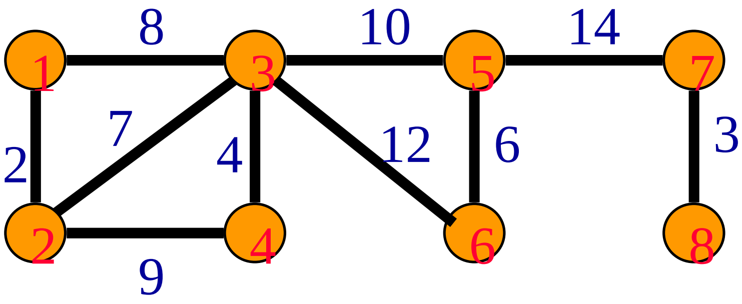


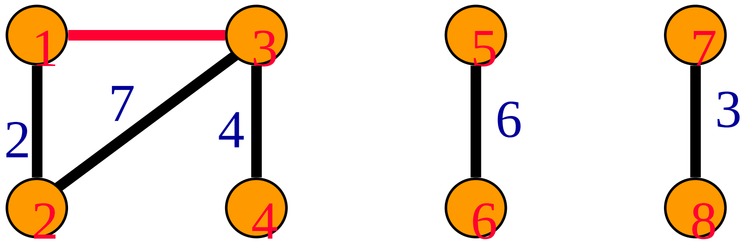
- Start with a forest that has no edges.

- Consider edges in ascending order of cost.

- Edge (1,2) is considered first and added to the forest.

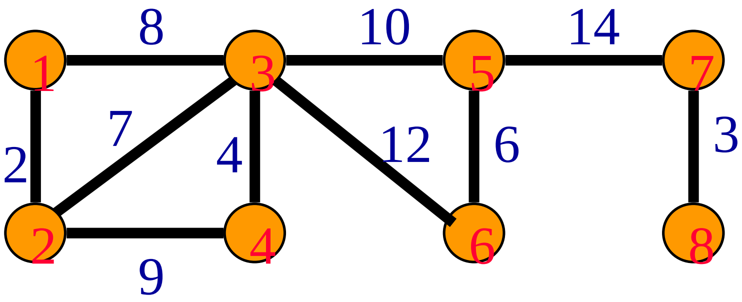
Kruskal’s Method

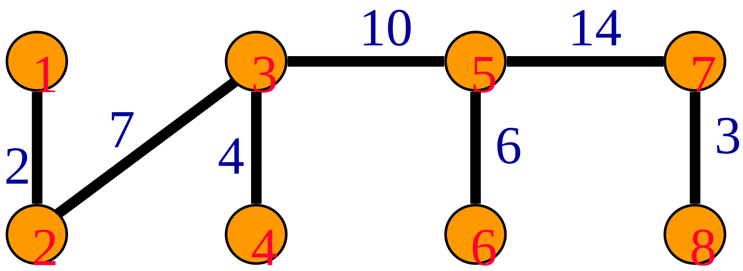




- Edge (7,8) is considered next and added.
- Edge (3,4) is considered next and added.
- Edge (5,6) is considered next and added.
- Edge (2,3) is considered next and added.
- Edge (1,3) is considered next and rejected because it creates a cycle.

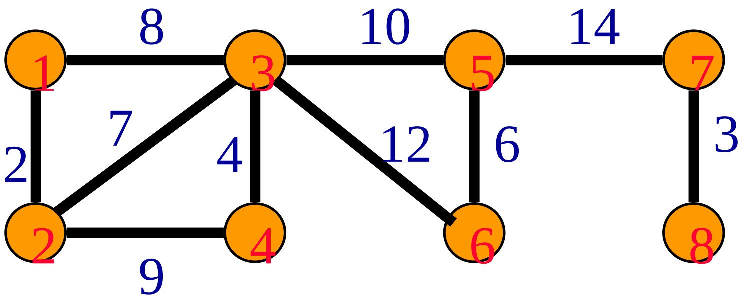
Kruskal’s Method

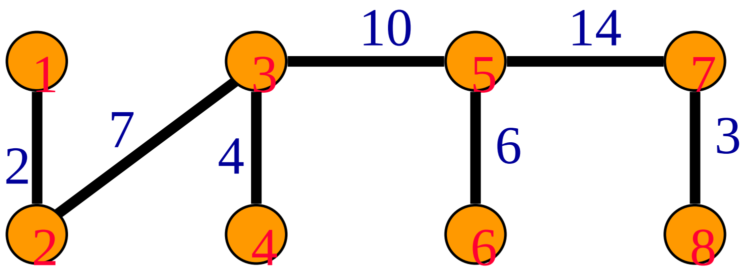




- Edge (2,4) is considered next and rejected because it creates a cycle.
- Edge (3,5) is considered next and added.
- Edge (3,6) is considered next and rejected.
- Edge (5,7) is considered next and added.

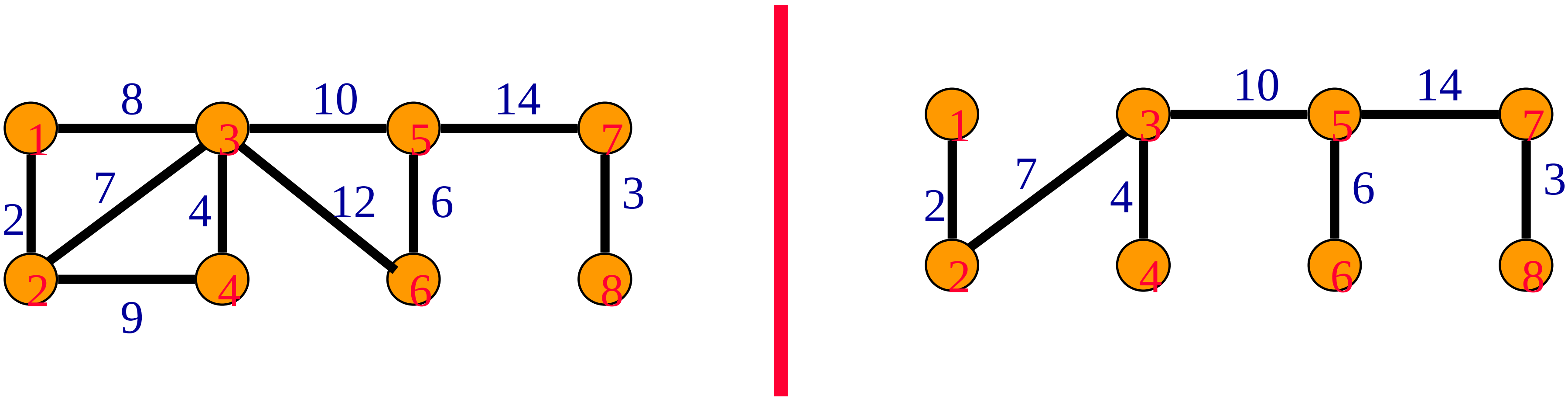
Kruskal’s Method





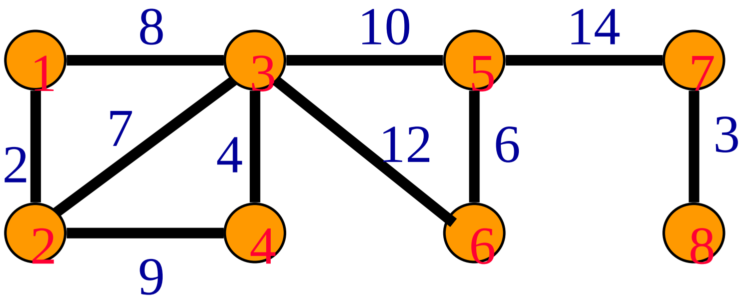
- n - 1 edges have been selected and no cycle formed.
- So we must have a spanning tree.
- Cost is 46.
- Min-cost spanning tree is unique when all edge costs are different.

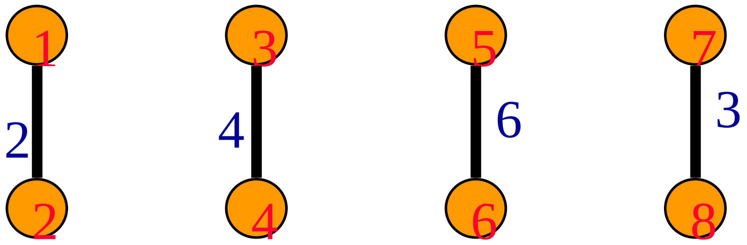
(***Prim’s Method



- Start with any single vertex tree.
- Get a 2-vertex tree by adding a cheapest edge.
- Get a 3-vertex tree by adding a cheapest edge.
- Grow the tree one edge at a time until the tree has n - 1 edges (and hence has all n vertices).

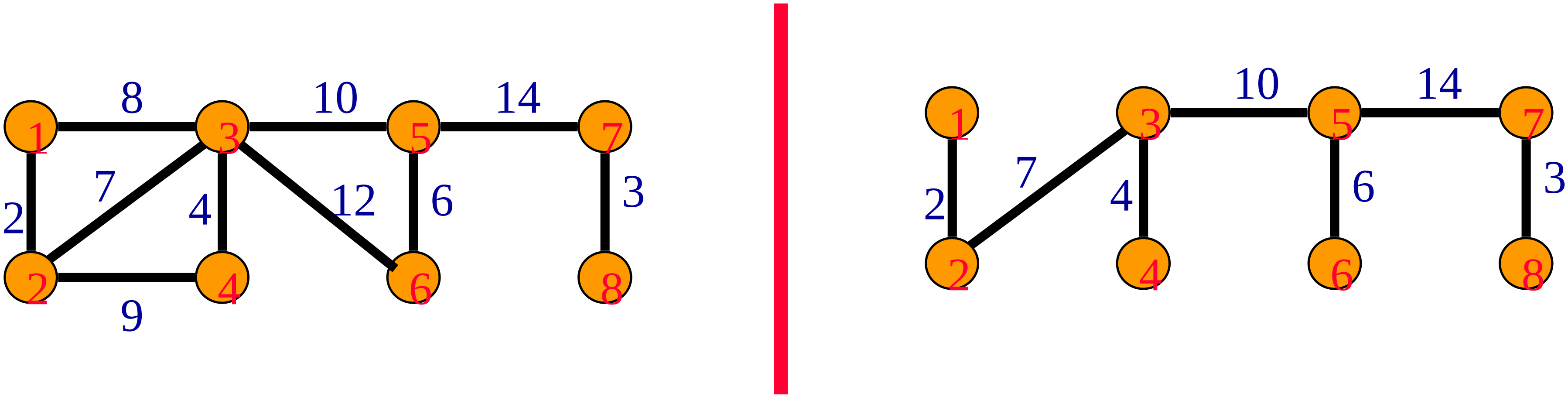
Sollin’s Method





- Start with a forest that has no edges.
- Each component selects a least cost edge with which to connect to another component.
- Duplicate selections are eliminated.
- Cycles are possible when the graph has some edges that have the same cost.

Sollin’s Method



- Each component that remains selects a least cost edge with which to connect to another component.
- Beware of duplicate selections and cycles.

Minimum-Cost Spanning Tree Methods

- Can prove that all stated edge selection/rejection result in a minimum-cost spanning tree.
- Prim's method is fastest.
  - $O(n^2)$ using an implementation similar to that of Dijkstra's shortest-path algorithm.
  - $O(e + n \log n)$ using a Fibonacci heap.
- Kruskal's uses union-find trees to run in $O(n + e \log e)$ time.

Pseudocode For Kruskal’s Method

```
Start with an empty set T of edges.
while (E is not empty && |T| != n-1)
{
    Let (u,v) be a least-cost edge in E.
    E = E - {(u,v)}. // delete edge from E
    if ((u,v) does not create a cycle in T)
       Add edge (u,v) to T.
}
if (| T | == n-1) T is a min-cost spanning tree.
else Network has no spanning tree.
```

Data Structures For Kruskal’s Method

Edge set E.

Operations are:

- Is E empty?
- Select and remove a least-cost edge.

Use a min heap of edges.

- Initialize. O(e) time.
- Remove and return least-cost edge. O(log e) time.

Data Structures For Kruskal’s Method

Set of selected edges T.

Operations are:

- Does T have n - 1 edges?
- Does the addition of an edge (u, v) to T result in a cycle?
- Add an edge to T.

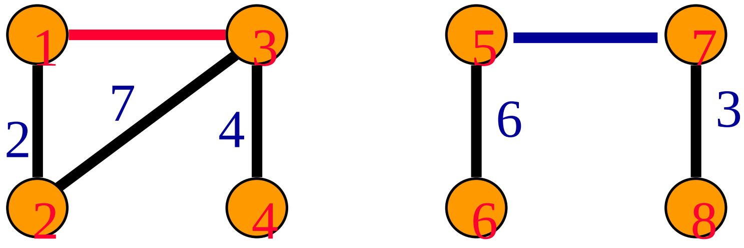
Data Structures For Kruskal's Method

Use an array for the edges of T.

- Does T have n - 1 edges?
  - Check number of edges in array. O(1) time.
- Does the addition of an edge (u, v) to T result in a cycle?
  - Not easy.
- Add an edge to T.
  - Add at right end of edges in array. O(1) time.

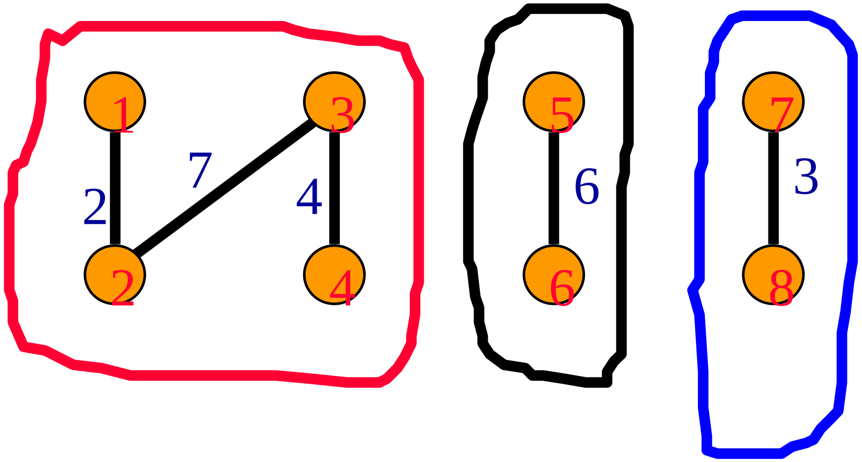
Data Structures For Kruskal's Method

Does the addition of an edge (u, v) to T result in a cycle?



- Each component of T is a tree.
- When u and v are in the same component, the addition of the edge (u,v) creates a cycle.
- When u and v are in the different components, the addition of the edge (u,v) does not create a cycle.

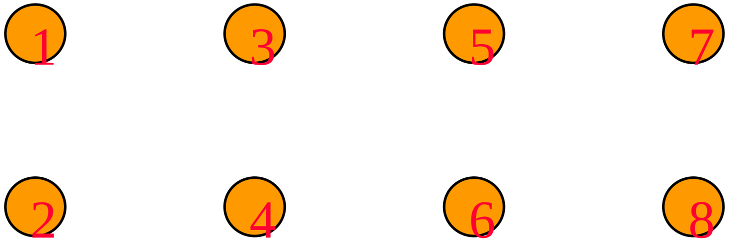
Data Structures For Kruskal's Method



- Each component of T is defined by the vertices in the component.
- Represent each component as a set of vertices.
  - {1, 2, 3, 4}, {5, 6}, {7, 8}
- Two vertices are in the same component iff they are in the same set of vertices.

Data Structures For Kruskal's Method

- Initially, T is empty.



- Initial sets are:
  - {1} {2} {3} {4} {5} {6} {7} {8}
- Does the addition of an edge (u, v) to T result in a cycle? If not, add edge to T.

```
s1 = Find(u); s2 = Find(v);

if (s1 != s2) Union(s1, s2);
```

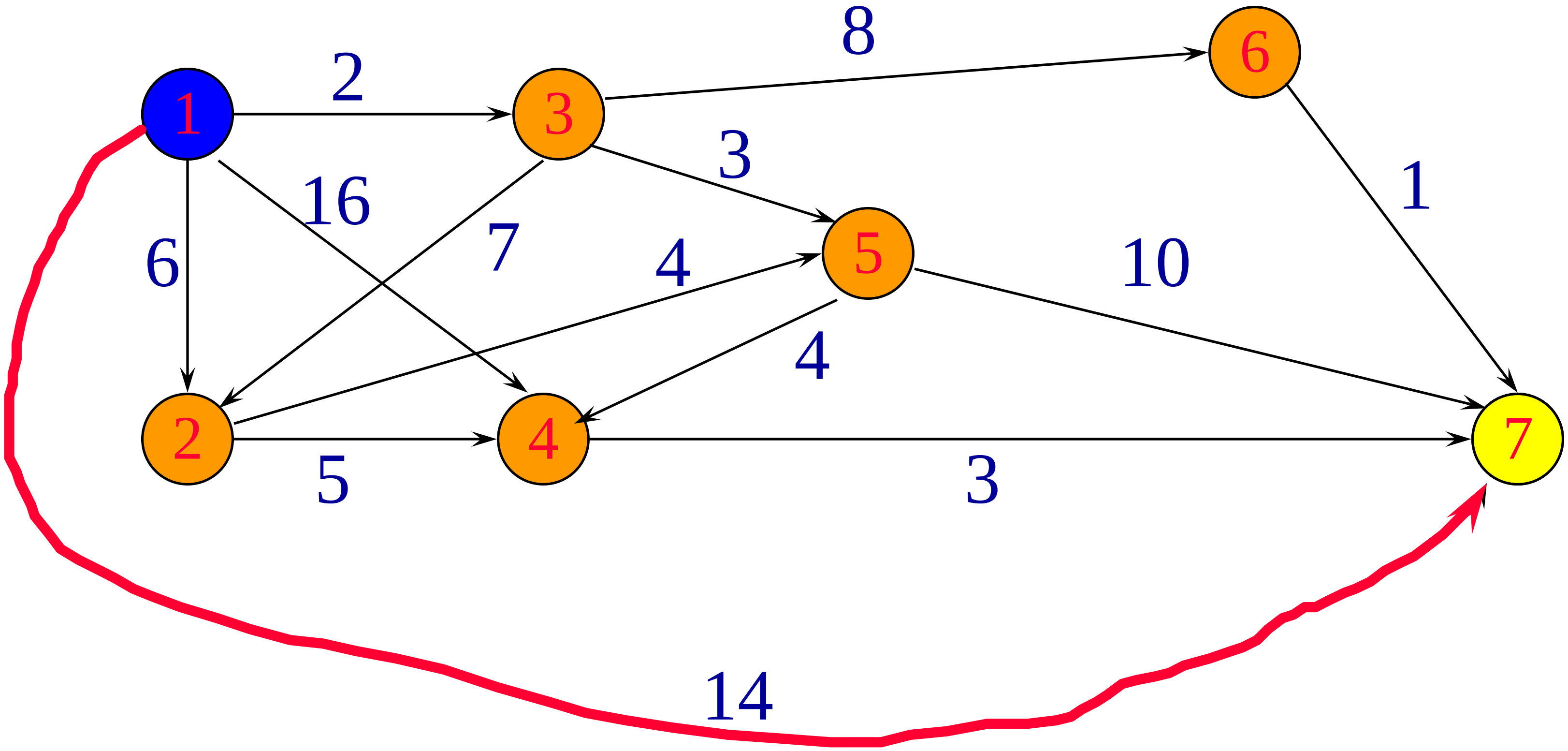
Data Structures For Kruskal’s Method

- Use fast solution for disjoint sets.
- Initialize.
  - O(n) time.
- At most 2e finds and n-1 unions.
  - Very close to O(n + e).
- Min heap operations to get edges in increasing order of cost take O(e log e).
- Overall complexity of Kruskal’s method is O(n + e log e).

9.2 Shortest Path Problems

- Directed weighted graph.
- Path length is sum of weights of edges on path.
- The vertex at which the path begins is the source vertex.
- The vertex at which the path ends is the destination vertex.

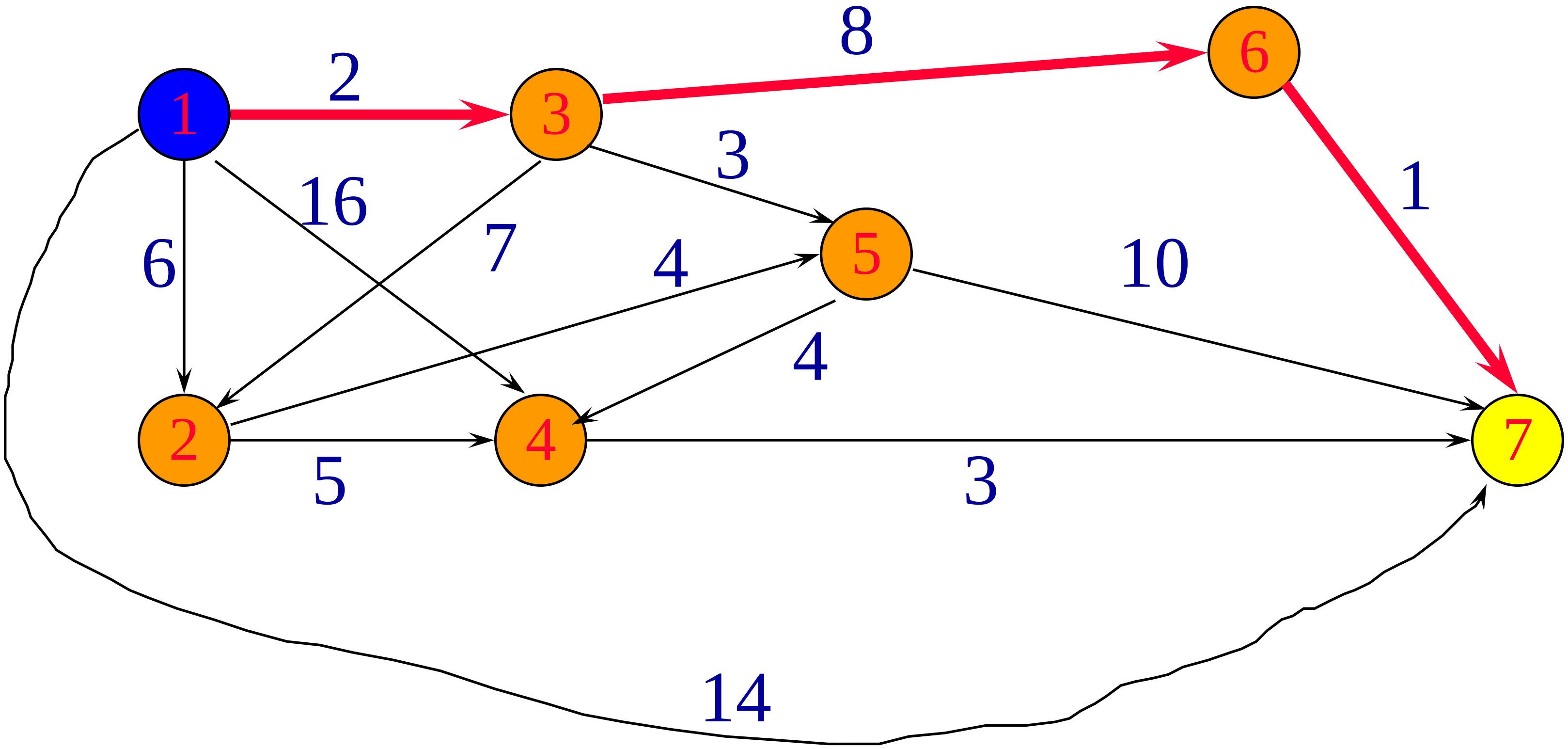
Example



A path from 1 to 7.

Path length is 14.

Example



Another path from 1 to 7.

Path length is 11.

Shortest Path Problems

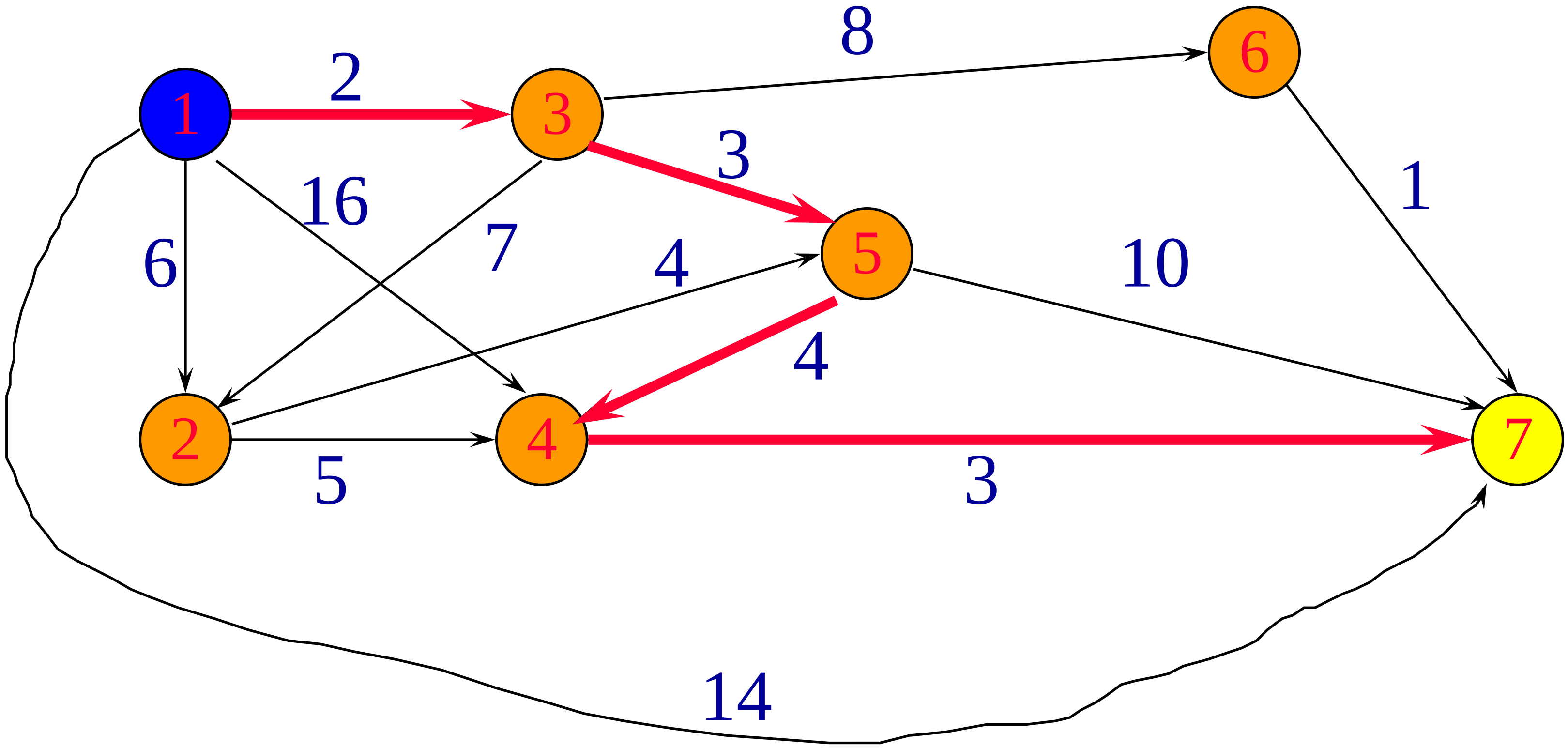
- Single source single destination.
- Single source all destinations.
- All pairs (every vertex is a source and destination).

Single Source Single Destination

Possible algorithm: (*****)

- Leave source vertex using cheapest/shortest edge.
- Leave new vertex using cheapest edge subject to the constraint that a new vertex is reached.
- Continue until destination is reached.

Constructed 1 To 7 Path



Path length is 12.

Not shortest path. Algorithm doesn’t work!

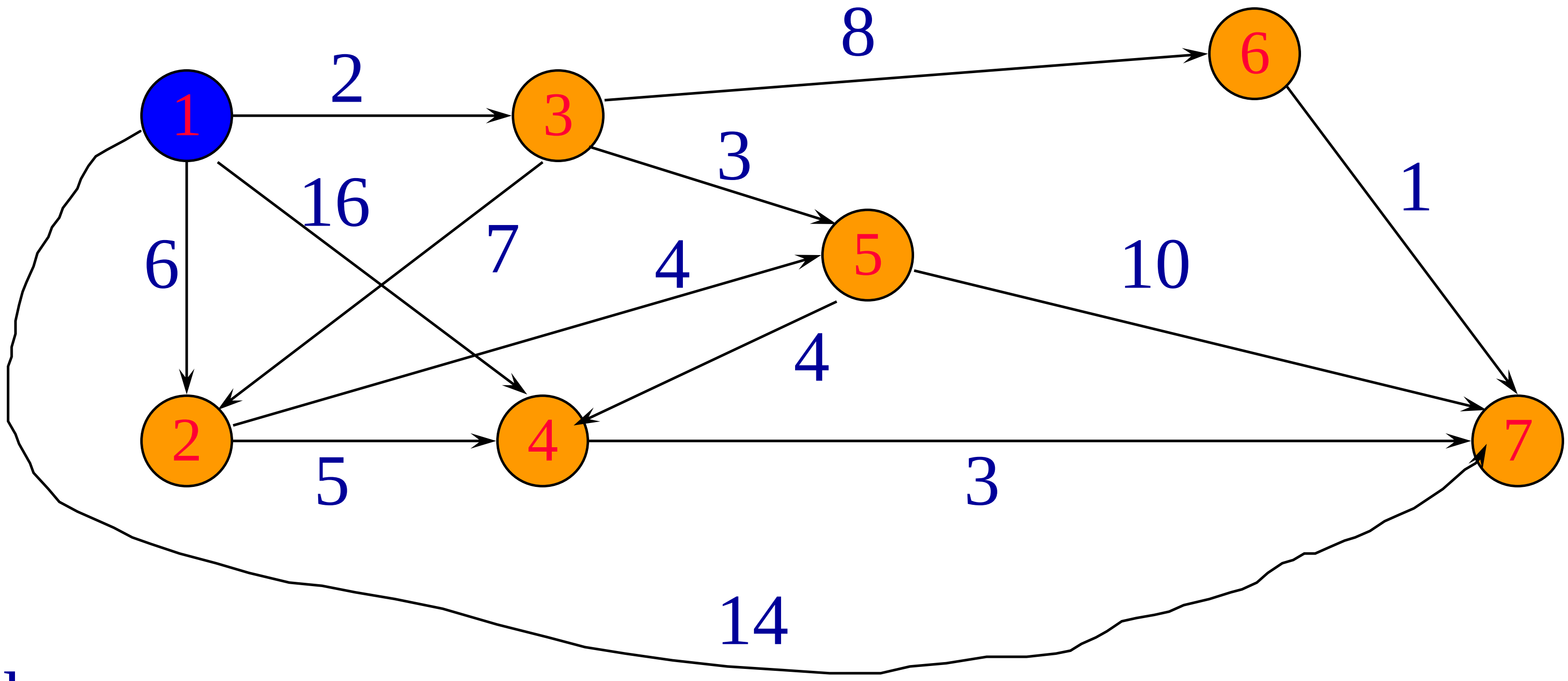
Single Source All Destinations

Need to generate up to n (n is number of vertices) paths (including path from source to itself).

Dijkstra’s method:

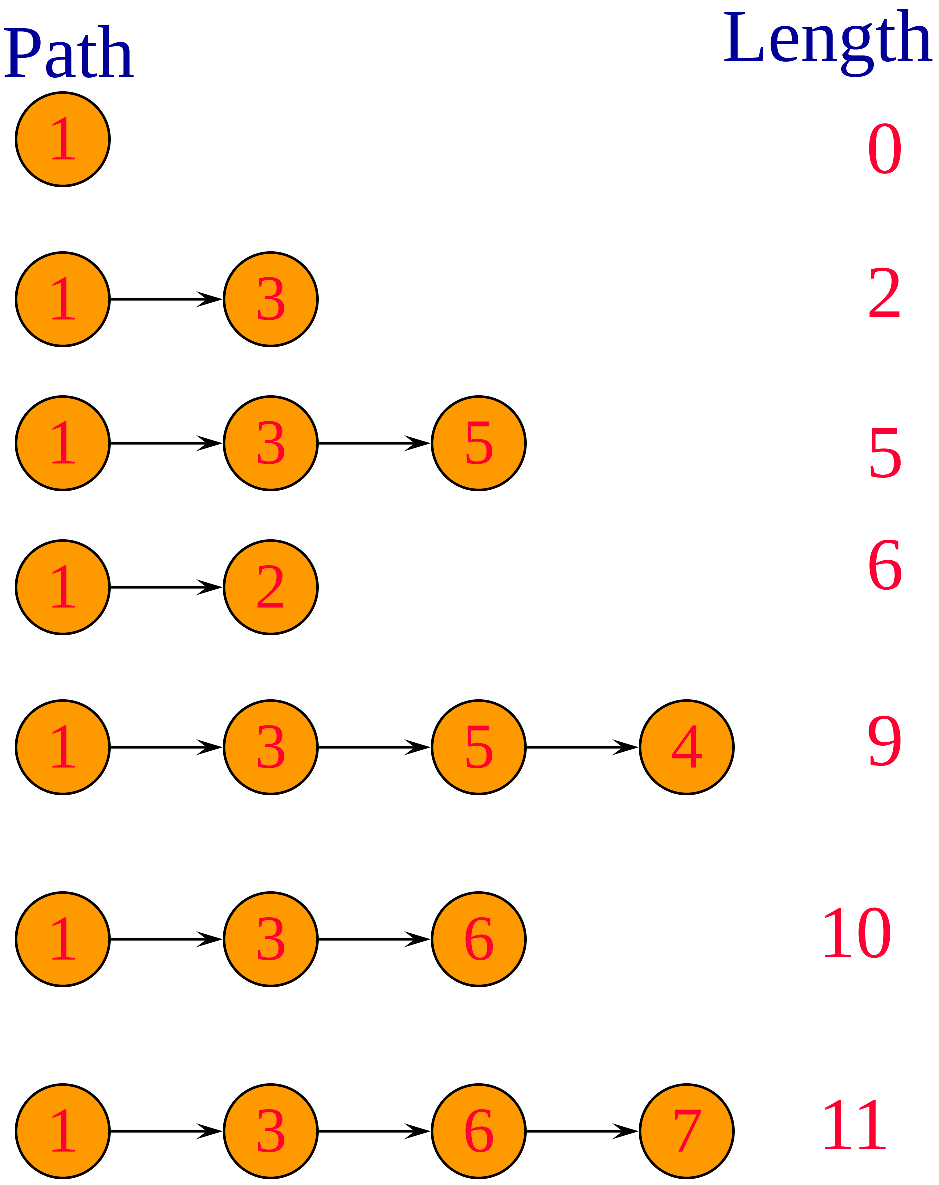
- Construct these up to n paths in order of increasing length.
- Assume edge costs (lengths) are >= 0.
- So, no path has length < 0.
- First shortest path is from the source vertex to itself. The length of this path is 0.

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



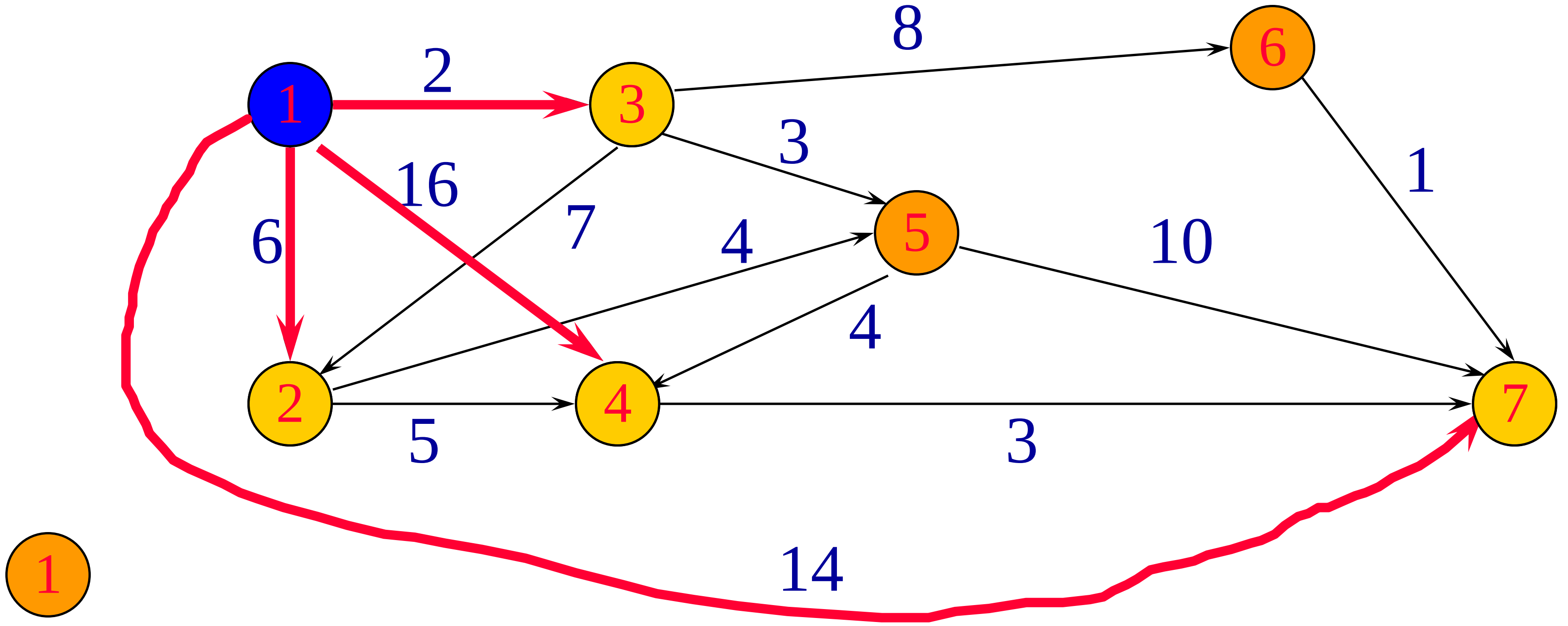
- Each path (other than first) is a one edge extension of a previous path.

- Next shortest path is the shortest one edge extension of an already generated shortest path.

Single Source All Destinations

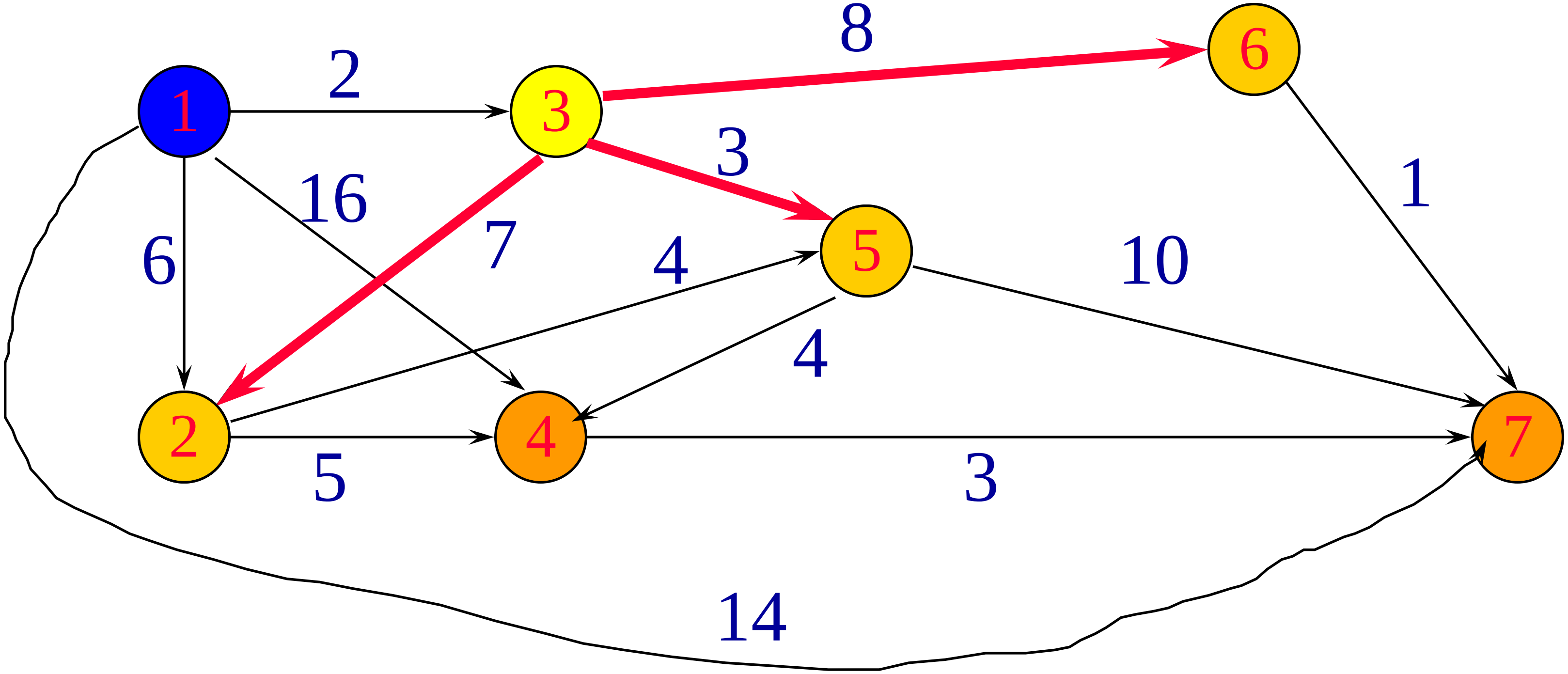
- Let d[i] be the length of a shortest one edge extension of an already generated shortest path, the one edge extension ends at vertex i.
- The next shortest path is to an as yet unreached vertex for which the d[] value is least.
- Let p[i] be the vertex just before vertex i on the shortest one edge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

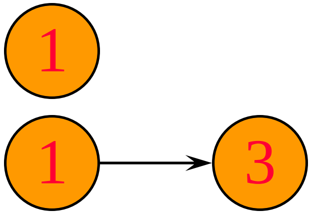
Single Source All Destinations



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	16	-	-	14
p	-	1	1	1	-	-	1

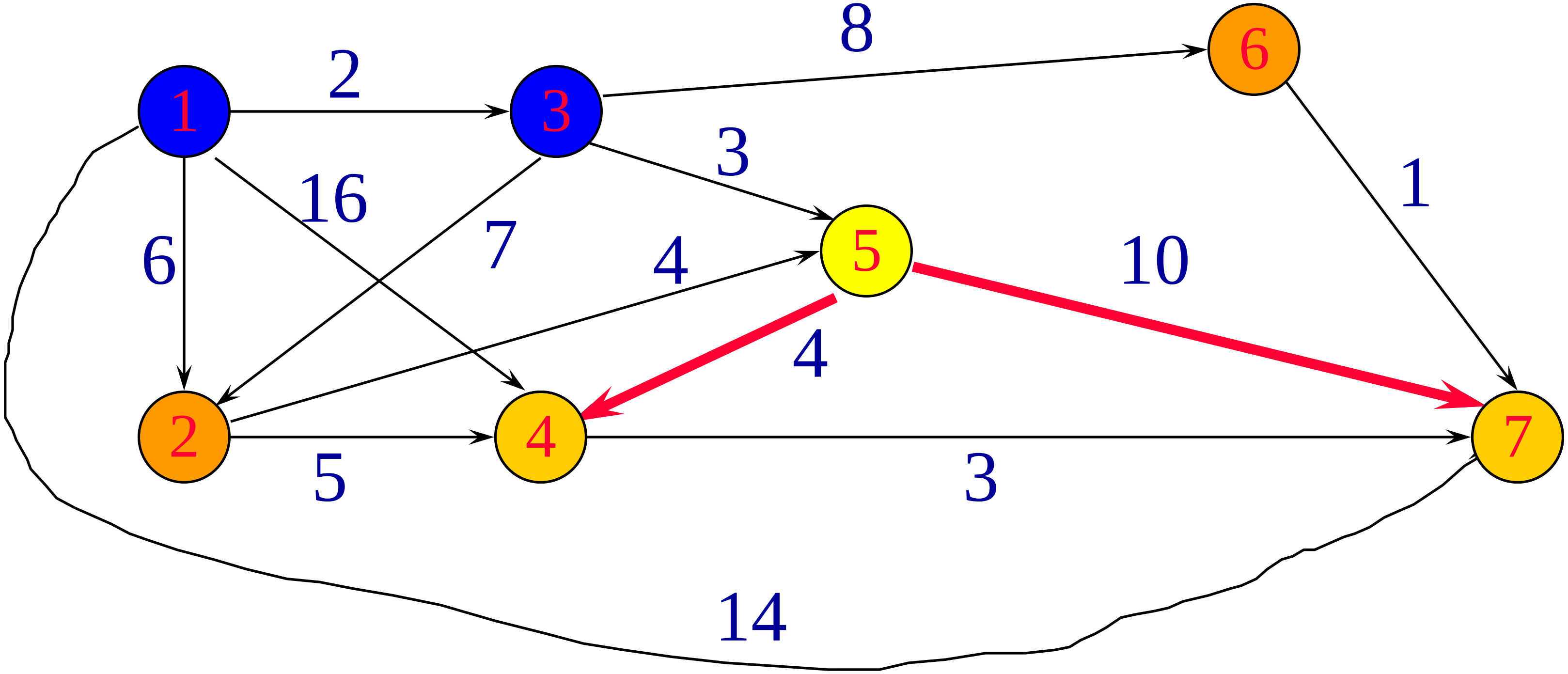
Single Source All Destinations

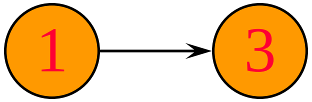


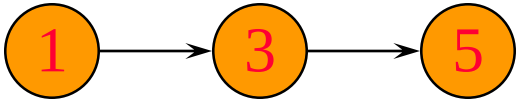


	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	16	5	10	14
p	-	1	1	1	3	3	1

Single Source All Destinations

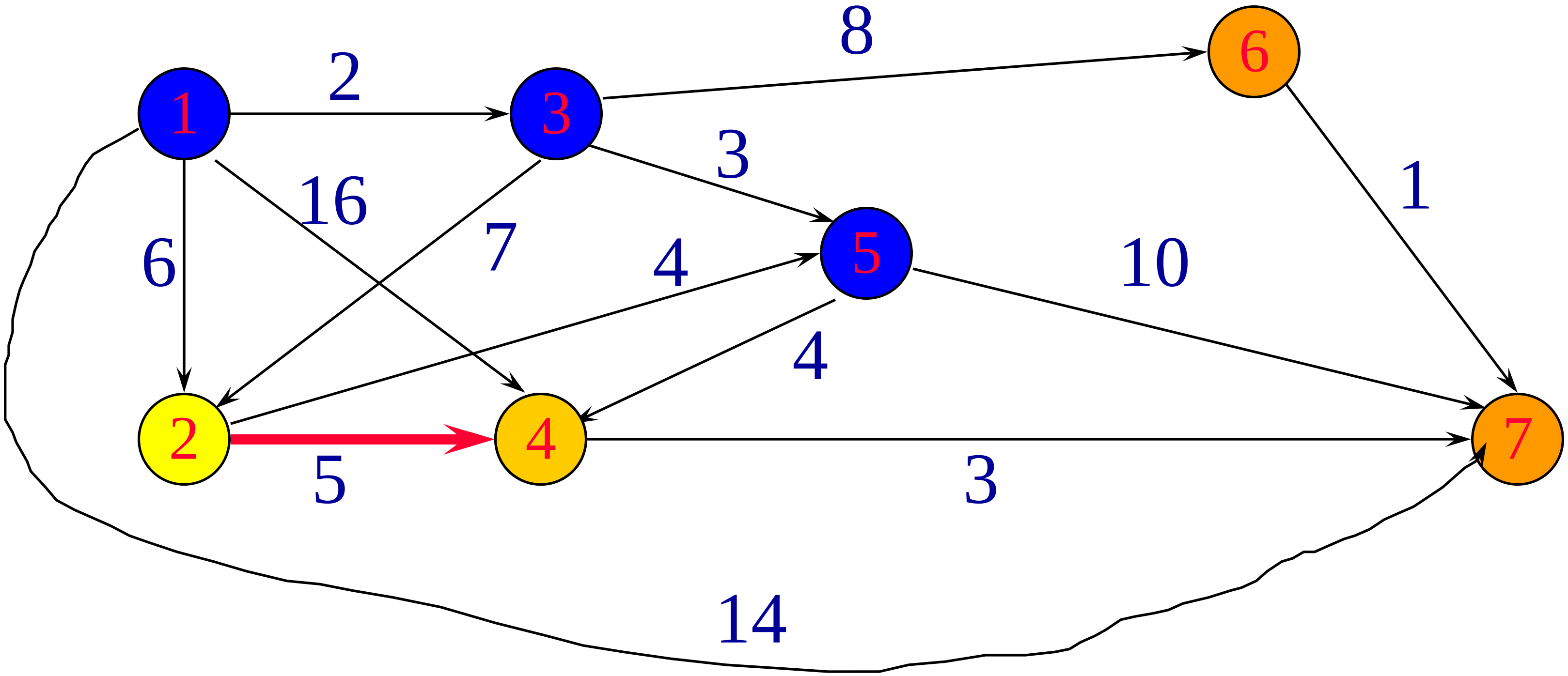


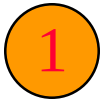


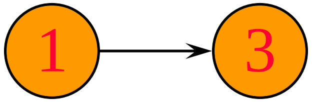


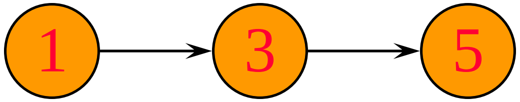
	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

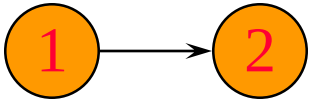
Single Source All Destinations





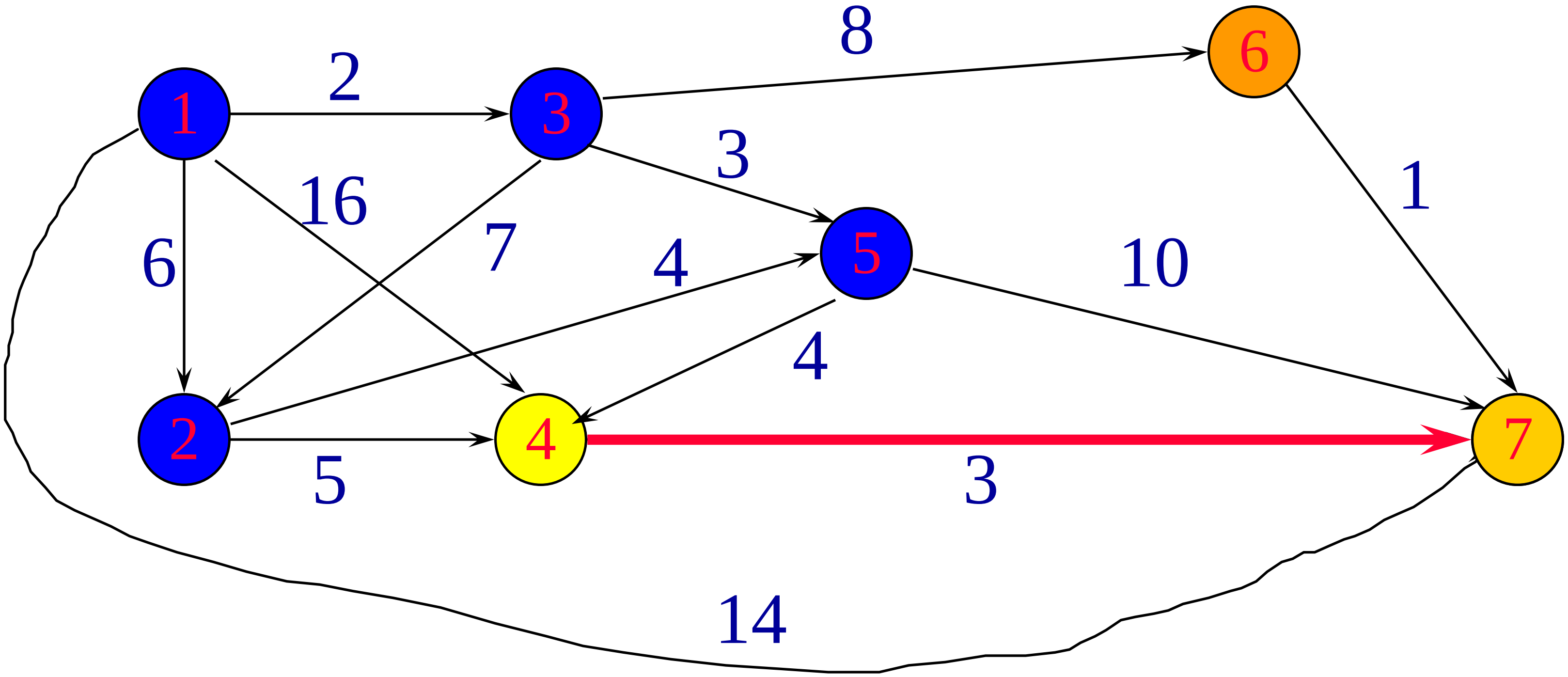


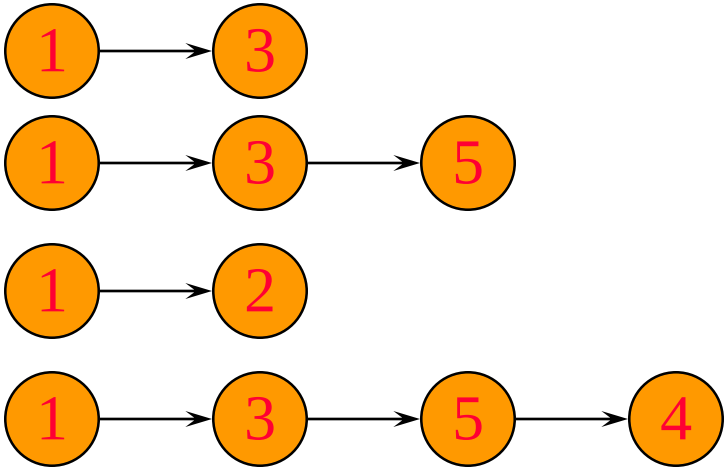




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

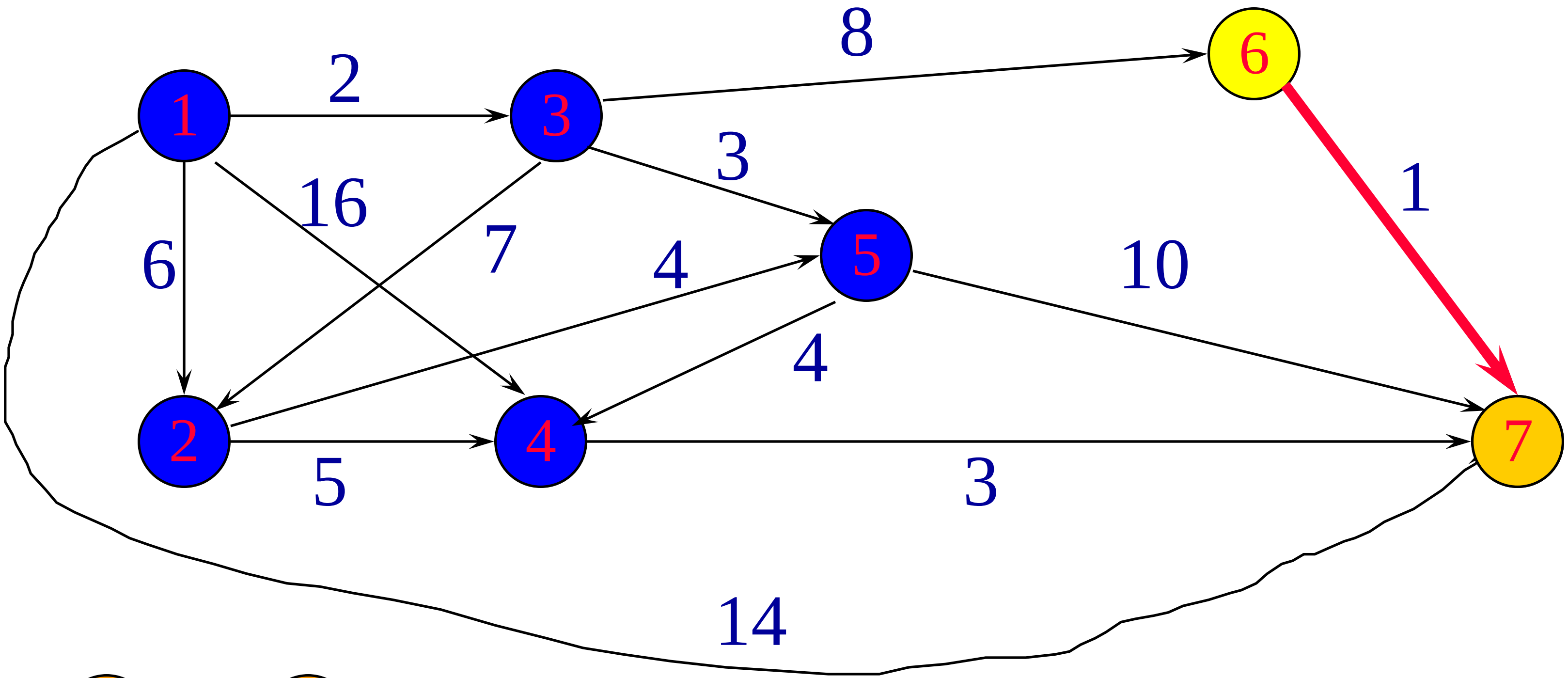
Single Source All Destinations

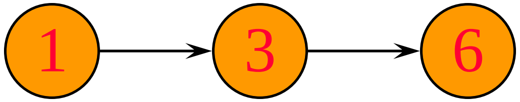




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	12
p	-	1	1	5	3	3	4

Single Source All Destinations

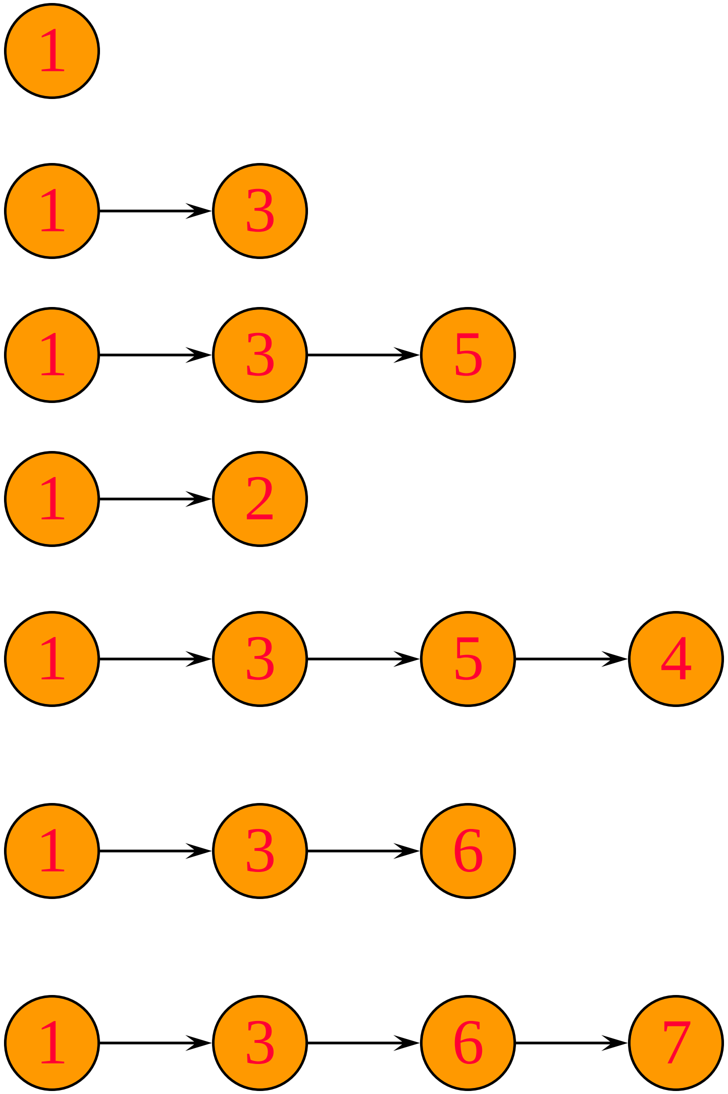




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	11
p	-	1	1	5	3	3	6

Single Source All Destinations

Path Length



0

2

5

6

9

10

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.

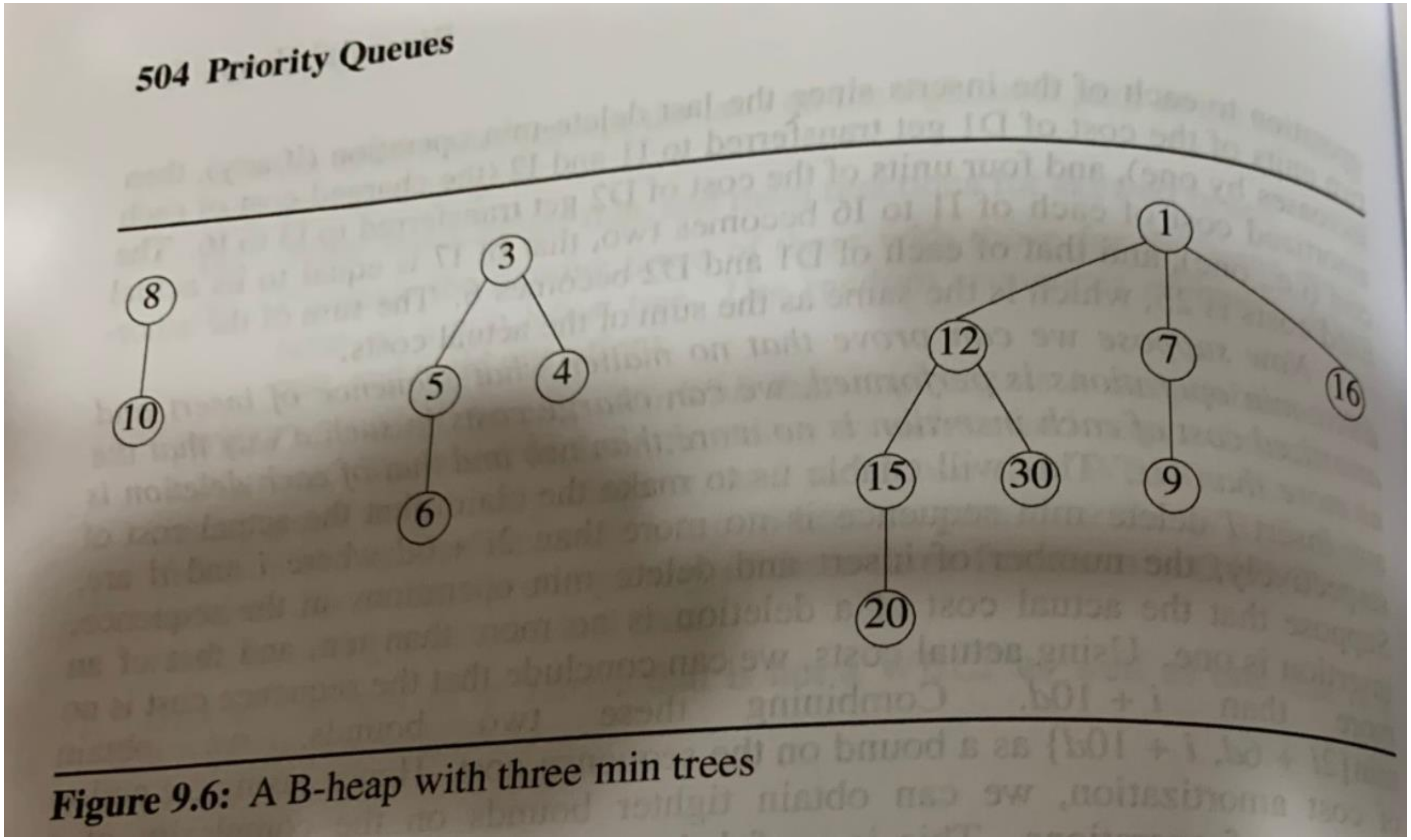
Fibonacci Heaps

- Def: There are two varietiese of Fibonacci heaps: min and max. A min Fibonacci heap is a collection of min trees; a max Fibonacci heap is a collection of max trees.
- Here considered is min Fibonacci heap, and named as F_heaps.
- An F-heap is a data structure that support the seven operations: getMin, Insert, DeleteMin, Meld, Delete, and DecreaseKey, where the last two are defined as follows:
  - Delete: Delete the element in a specified node. We refer too this delete operation as arbitrary delete.
  - DecreaseKey: Decrease the key/priority of a specified node by a given positive amount.

Fibonacci Heaps(c.)

- When an F-heap, is used, the Delete operation takes O(log n) amortized time and the DecreaseKey takes O(1) anotized time.
- During implementation, there are at least five pointers inside each node: parent, child, childCut, leftlink and rightlink, one datum representing the number of its children, and one datum associated with a key. The children nodes of a node can be connected by a double linked list according to liftlink and rightlink correspondingly.

Fibonacci Heaps(c.)



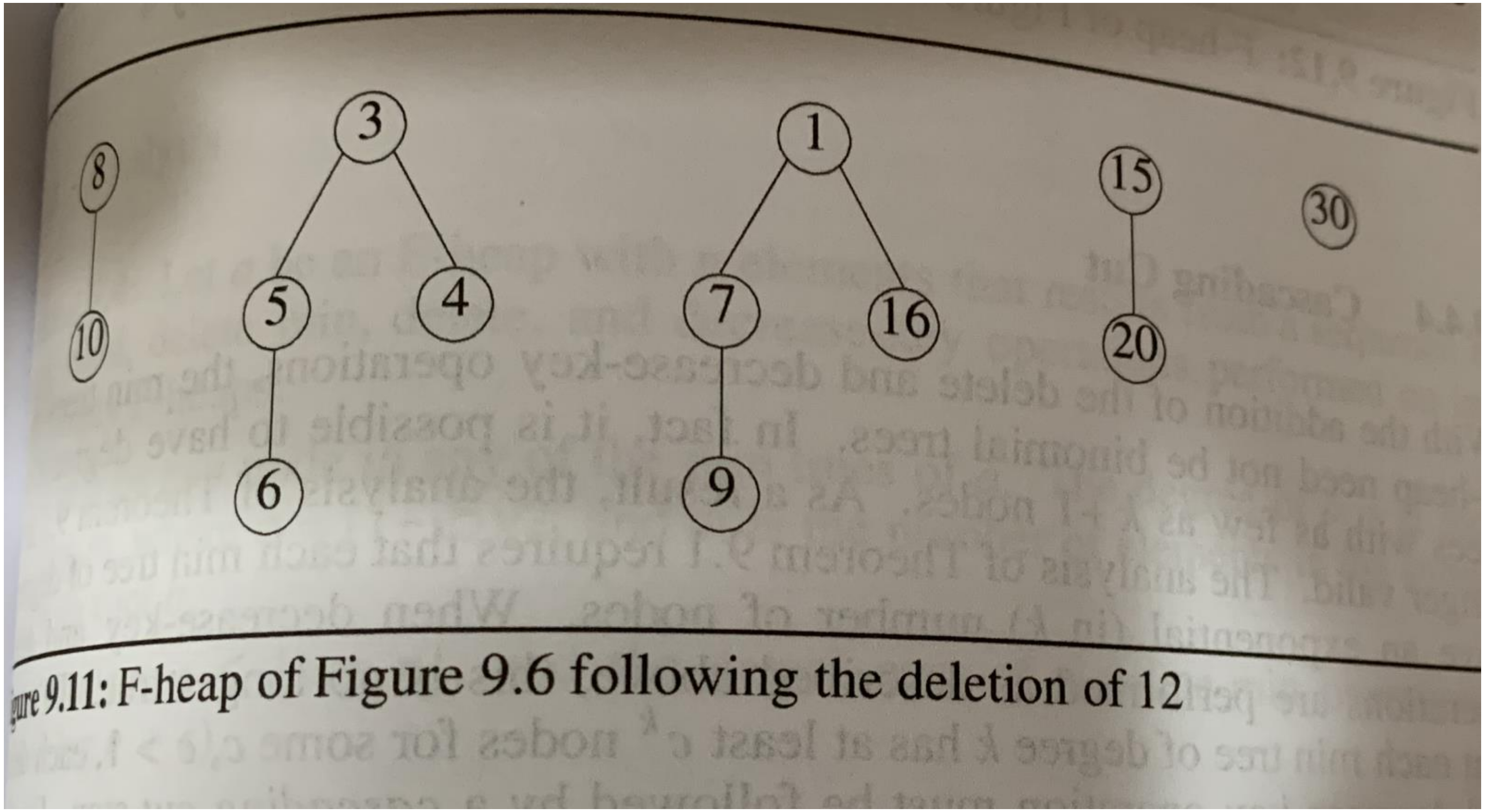
504 Priority Queues

Figure 9.6: A B-heap with three min trees

Fibonacci Heaps(c.)

- Deletion from an F-Heap
- To delete an arbitrary node b from a F-heap, we do the following:
  1. If min=b, then do a delete-min; otherwise do Steps 2, 3, and 4 below.
  2. Delete b from its doubly linked list.
  3. Combine the doubly linked list of b's children with the doubly linked list pointed at by min into a single double linked list. Trees of equal degree are not jointed as a delete-min operation.
  4. Dispose of node b.

Fibonacci Heaps(c.)



gure 9.11: F-heap of Figure 9.6 following the deletion of 12

Fibonacci Heaps(c.)

- Decrease Key
- To decrease the key in node *b* we do the following:
  1. Reduce the key in *b*.
  2. If *b* is not a min tree root and its key is smaller than that in its parent, then delete *b* from its doubly linked list and insert it into the doubly linked list of min tree nodes
  3. Change min to point to be if the key in *b* is smaller than that in min.

Fibonacci Heaps(c.)

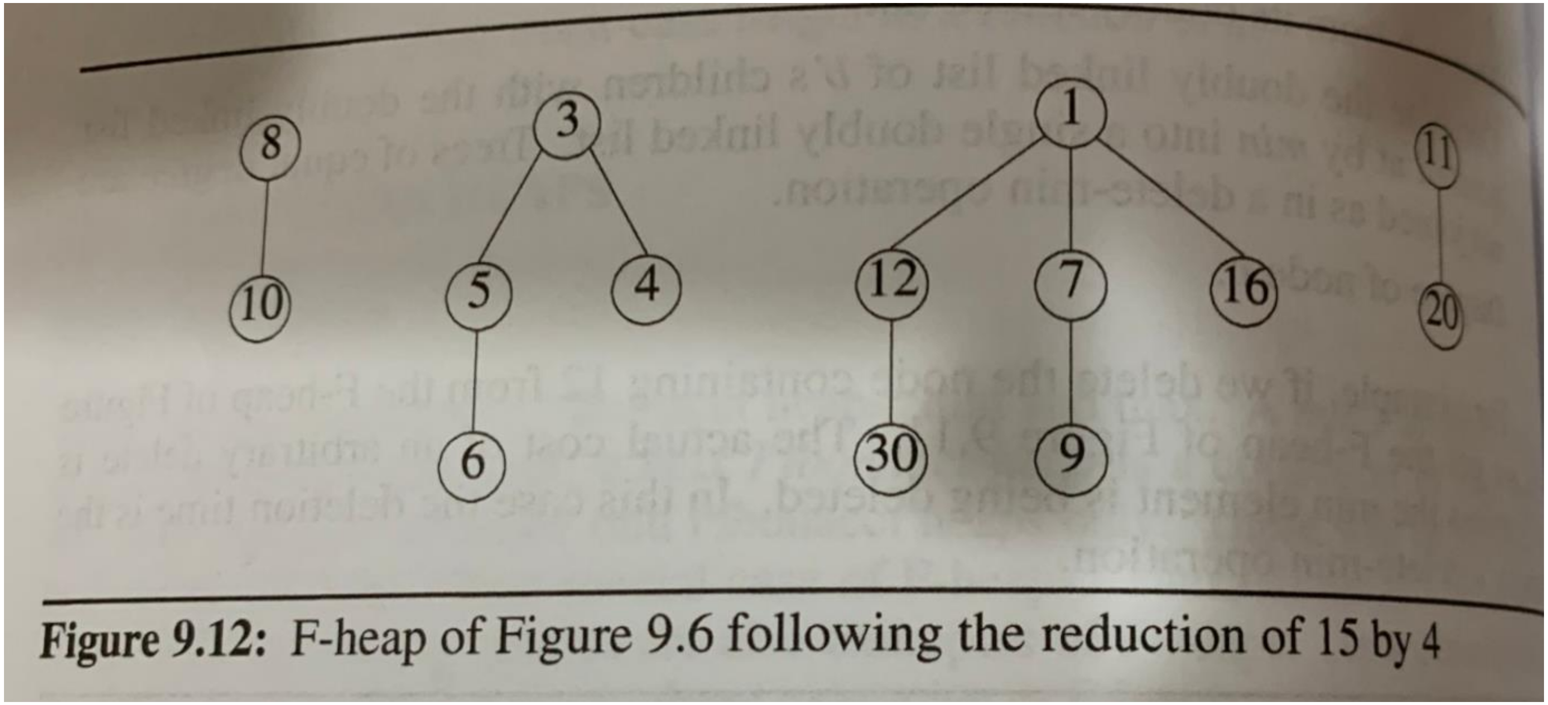


Figure 9.12: F-heap of Figure 9.6 following the reduction of 15 by 4

Fibonacci Heaps(c.)

- Cascading Cut
- To decrease the key in node b we do the following:
    1. Reduce the key in b.
    2. If b is not a min tree root and its key is smaller than that in its parent, then delete b from its doubly linked list and insert it into the doubly linked list of min tree nodes
    3. Change min to point to be if the key in b is smaller than that in min.

Fibonacci Heaps(c.)

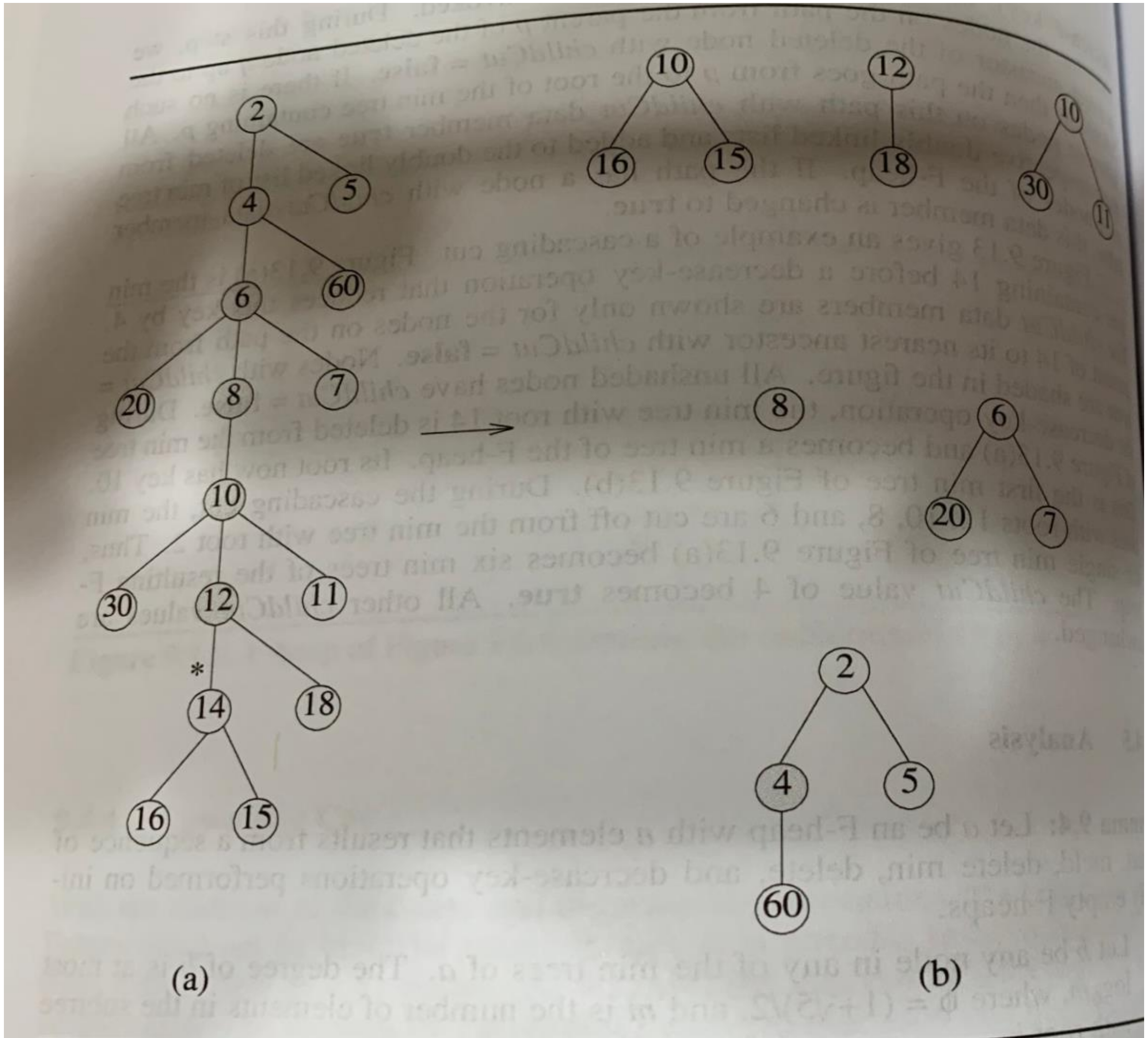


Figure 9.13: A cascading cut following a decrease of key 14 by 4

Fibonacci Heaps(c.)

Analysis：

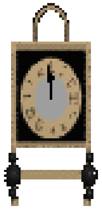
- Lemma 1: Let *a* be an F-heap with *n* elements that results from *a* sequence of insert, meld, delete min, delete, and decrease-key operations performed on initial empty F-heaps.
    1. Let *b* be any node in any of the min trees of *a*. The degree of *b* is at most $\log_\phi m$, where $\phi = (1+\sqrt{5})/2$, and *m* is the number of elements in the subtree with root *b*.
    2. maxDegree $\leq \lfloor \log_\phi n \rfloor$, and the actual cost of a delete-min operation is $O(\log n + s)$
- Theorem 2: If a sequence of *n* insert, meld, delete min, delete, and decrease-key operations is performed on an initially empty F-heap, then we can amortize costs such that the amortized time complexity of each insert, meld, and decrease-key operation is O(1) and that of each delete min and delete operation is O(log *n*). The total time complexity of the entire sequence is the sum of the amortized complexities of the individual operations in the sequence.

9.3 Dijkstra’s Algorithm

Data Structures for Dijkstra’s Algorithm

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

Complexity



- O(n) to select next destination vertex.
- O(out-degree) to update d[] and p[] values when adjacency lists are used.
- O(n) to update d[] and p[] values when adjacency matrix is used.
- Selection and update done once for each vertex to which a shortest path is found.
- Total time is $O(n^2 + e) = O(n^2)$.

Complexity

- When a min heap of d[] values is used in place of the linear list L of reachable vertices, total time is $O((n+e) \log n)$, because $O(n)$ remove min operations and $O(e)$ change key (d[] value) operations are done.
- 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 $O(n \log n + e)$.

Single-Source All-Destinations Shortest Paths With General Weights

- Directed weighted graph.
- Edges may have negative cost.
- No cycle whose cost is < 0.
- Find a shortest path from a given source vertex s to each of the n vertices of the digraph.

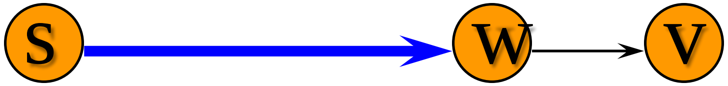
Single-Source All-Destinations Shortest Paths With General Weights

- Dijkstra’s $O(n^2)$ single-source greedy algorithm doesn’t work when there are negative-cost edges.

Bellman-Ford Algorithm

- Single-source all-destinations shortest paths in digraphs with negative-cost edges.
- Uses dynamic programming.
- Runs in $O(n^3)$ time when adjacency matrices are used.
- Runs in O(ne) time when adjacency lists are used.

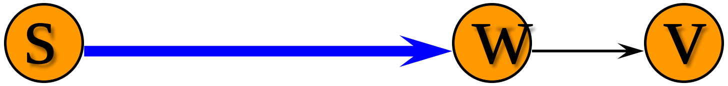
Strategy



- To construct a shortest path from the source to vertex v, decide on the max number of edges on the path and on the vertex that comes just before v.

- Since the digraph has no cycle whose length is < 0, we may limit ourselves to the discovery of cycle-free (acyclic) shortest paths.
- A path that has no cycle has at most n-1 edges.

Cost Function d



- Let d(v,k) (dist^k[v]) be the length of a shortest path from the source vertex to vertex v under the constraint that the path has at most k edges.
- d(v,n-1) is the length of a shortest unconstrained path from the source vertex to vertex v.
- We want to determine d(v,n-1) for every vertex v.

Value Of d(*,0)

- d(v,0) is the length of a shortest path from the source vertex to vertex v under the constraint that the path has at most 0 edges.

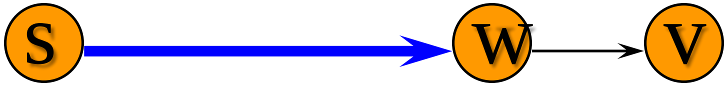
- d(s,0) = 0.
- d(v,0) = infinity for v != s.

Recurrence For d(*,k), k > 0

- d(v,k) is the length of a shortest path from the source vertex to vertex v under the constraint that the path has at most k edges.
- If this constrained shortest path goes through no more than k-1 edges, then d(v,k) = d(v,k-1).

Recurrence For d(*,k), k > 0

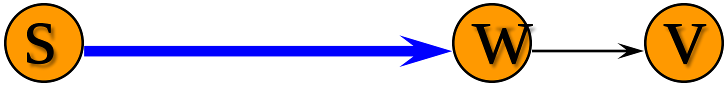
- If this constrained shortest path goes through k edges, then let w be the vertex just before v on this shortest path (note that w may be s).



- We see that the path from the source to w must be a shortest path from the source vertex to vertex w under the constraint that this path has at most k-1 edges.
- d(v,k) = d(w,k-1) + length of edge (w,v).

Recurrence For d(*,k), k > 0

- d(v,k) = d(w,k-1) + length of edge (w,v).



- We do not know what w is.
- We can assert
  - d(v,k) = min{d(w,k-1) + length of edge (w,v)}, where the min is taken over all w such that (w,v) is an edge of the digraph.
- Combining the two cases considered yields:
  - d(v,k) = min{d(v,k-1),
    min{d(w,k-1) + length of edge (w,v)}}

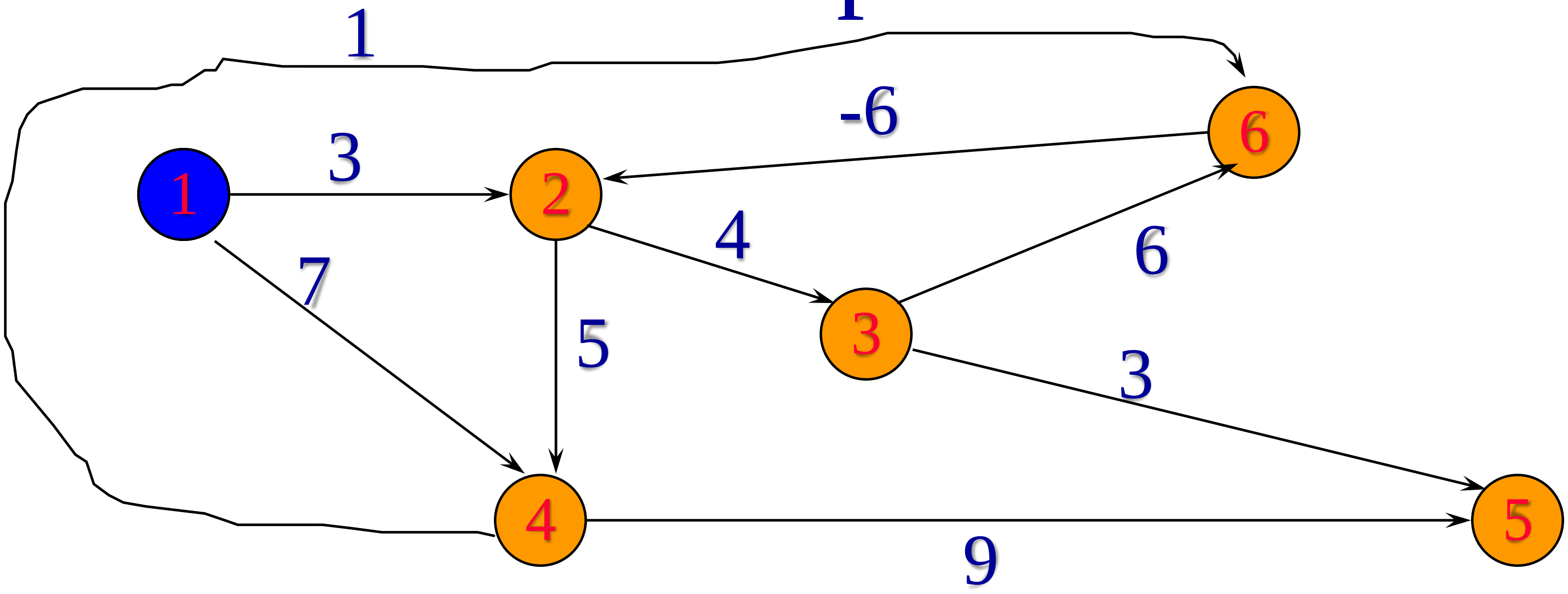
Pseudocode To Compute d(*,*)

```
// initialize d(*,0)
d(s,0) = 0;
d(v,0) = infinity, v != s;
// compute d(*,k), 0 < k < n
for (int k = 1; k < n; k++)
{
    d(v,k) = d(v,k-1), 1 <= v <= n;
    for (each edge (u,v))
        d(v,k) = min{d(v,k), d(u,k-1) + cost(u,v)}
}
```

p(*,*)

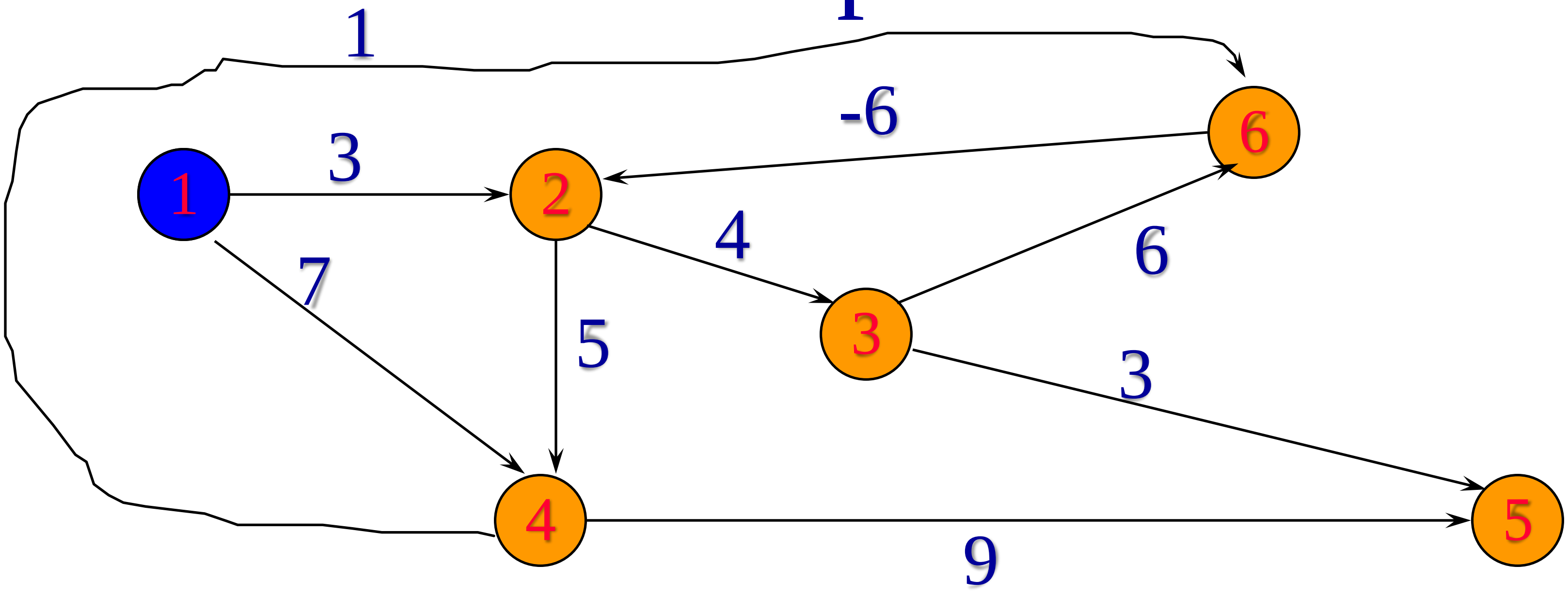
- Let p(v,k) be the vertex just before vertex v on the shortest path for d(v,k).
- p(v,0) is undefined.
- Used to construct shortest paths.

Example



Source vertex is 1.

Example



	1	2	3	4	5	6
0	0	-	-	-	-	-
1	0	3	-	7	-	-
2	0	3	7	7	16	8
3	0	2	7	7	10	8
4	0	2	6	7	10	8

d(v,k)

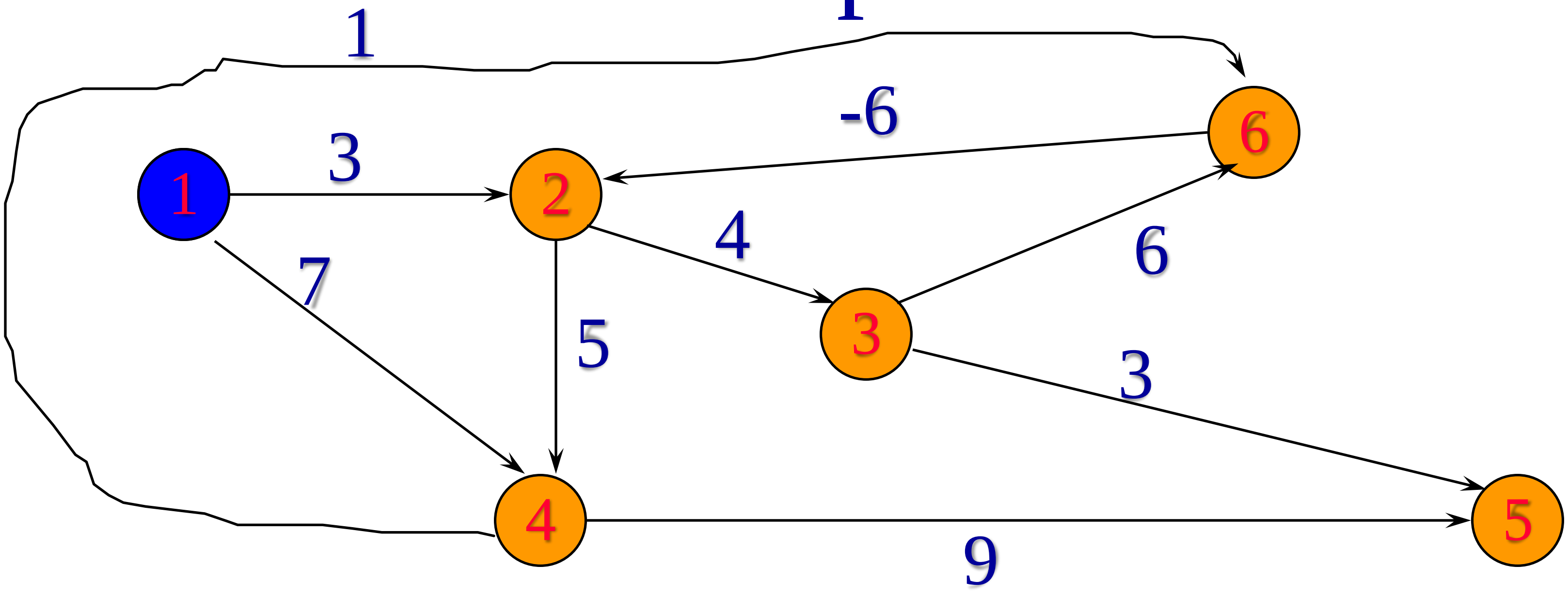
v

k

-	-	-	-	-	-
-	1	-	1	-	-
-	1	2	1	4	4
-	6	2	1	3	4
-	6	2	1	3	4

p(v.k)

Example



	1	2	3	4	5	6
4	0	2	6	7	10	8
5	0	2	6	7	9	8

d(v,k)

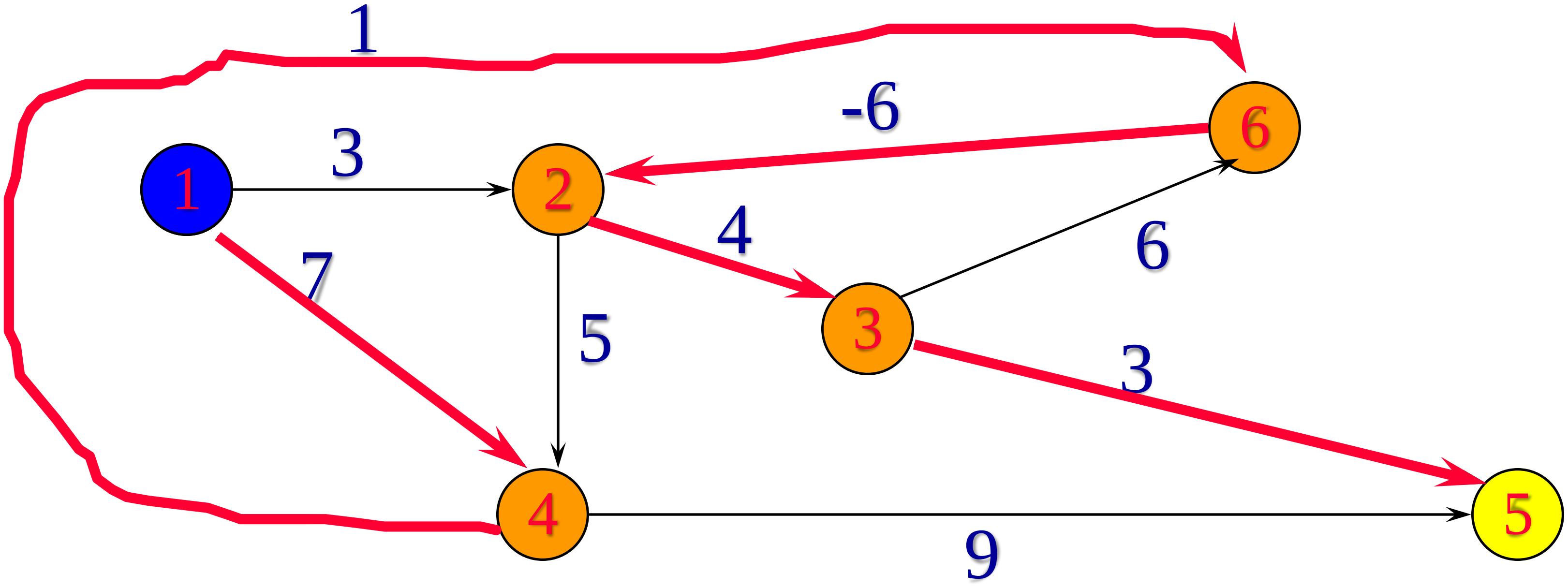
v →

-	6	2	1	3	4
-	6	2	1	3	4

k ↓

p(v.k)

Shortest Path From 1 To 5



	1	2	3	4	5	6
5	0	2	6	7	9	8

d(v,5)

1	2	3	4	5	6
-	6	2	1	3	4

p(v,5)

Observations

- $d(v,k) = \min\{d(v,k-1),$
  $\min\{d(w,k-1) + \text{length of edge } (w,v)\}\}$
- $d(s,k) = 0$ for all k.
- If $d(v,k) = d(v,k-1)$ for all v, then $d(v,j) = d(v,k-1)$, for all $j >= k-1$ and all v.
- If we stop computing as soon as we have a $d(*,k)$ that is identical to $d(*,k-1)$ the run time becomes
  - $O(n^3)$ when adjacency matrix is used.
  - $O(ne)$ when adjacency lists are used.

Observations

- The computation may be done in-place.

  $d(v) = \min\{d(v), \min\{d(w) + \text{length of edge } (w,v)\}\}$

  instead of

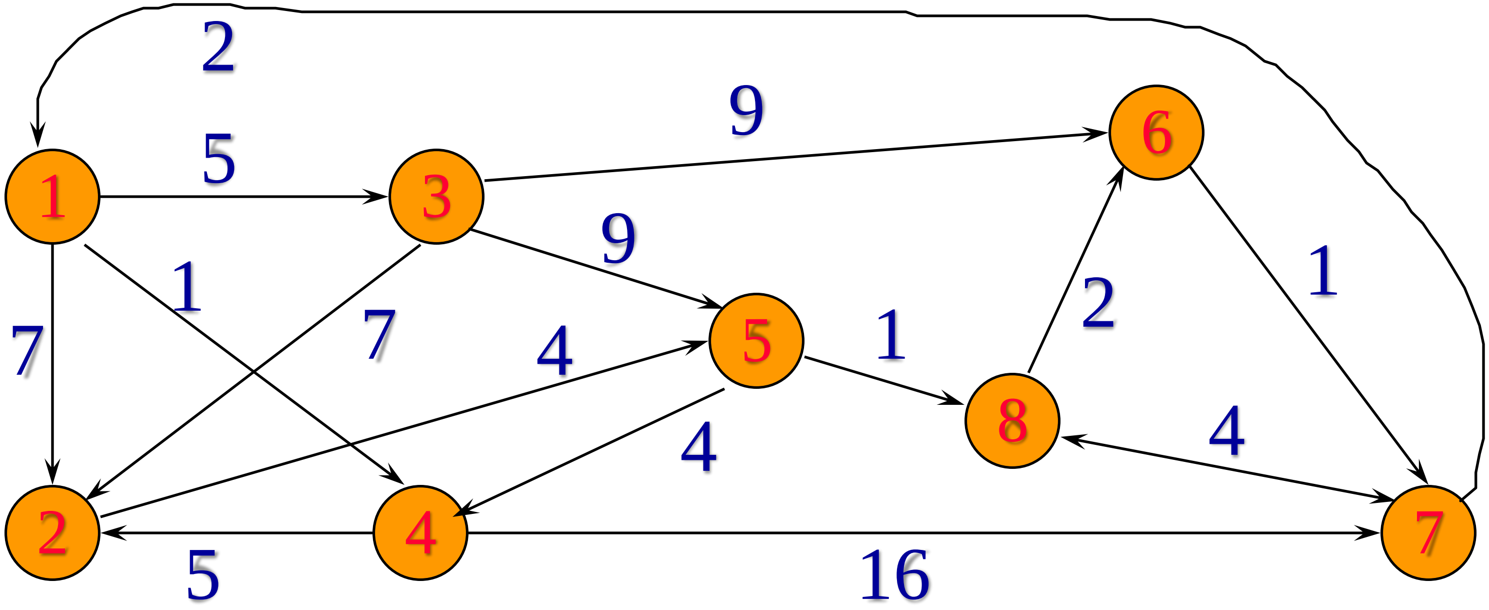
  $d(v,k) = \min\{d(v,k-1),$

  $\min\{d(w,k-1) + \text{length of edge } (w,v)\}\}$

- Following iteration k, $d(v,k+1) <= d(v) <= d(v,k)$
- On termination $d(v) = d(v,n-1)$.
- Space requirement becomes $O(n)$ for $d(*)$ and $p(*)$.

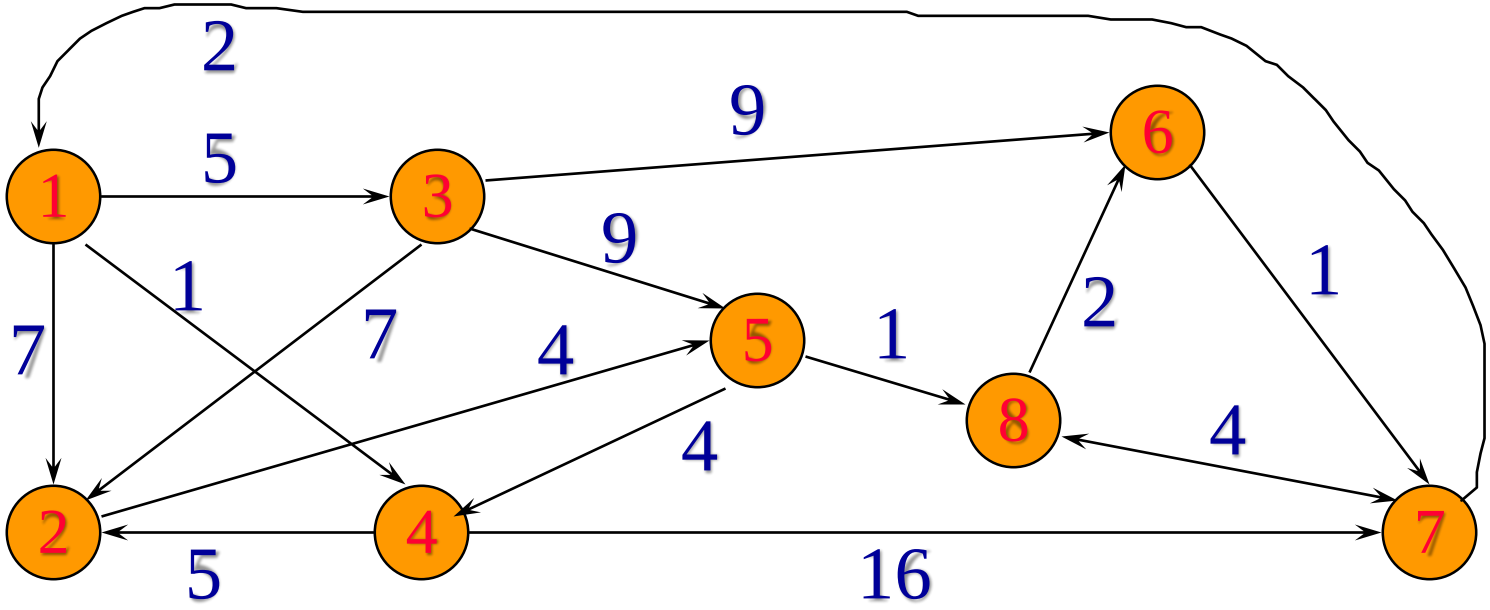
All-Pairs Shortest Paths

- Given an n-vertex directed weighted graph, find a shortest path from vertex i to vertex j for each of the n^2 vertex pairs (i,j).

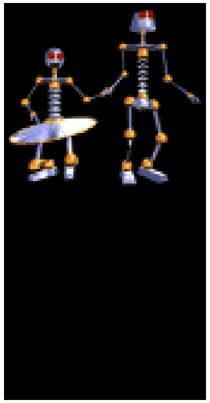


Dijkstra's Single Source Algorithm

- Use Dijkstra's algorithm n times, once with each of the n vertices as the source vertex.



Performance

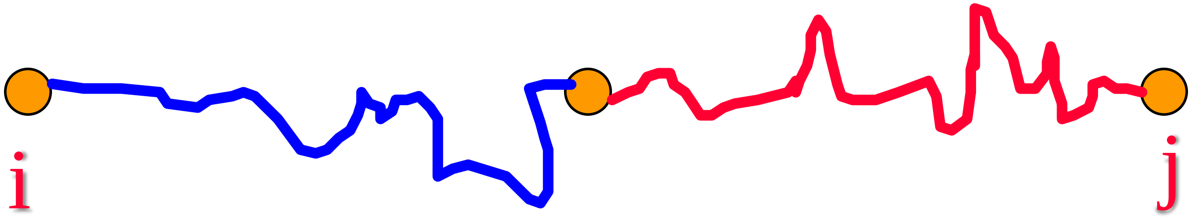


- Time complexity is $O(n^3)$ time.
- Works only when no edge has a cost < 0.

(** 9.4 Floyd's Algorithm **)

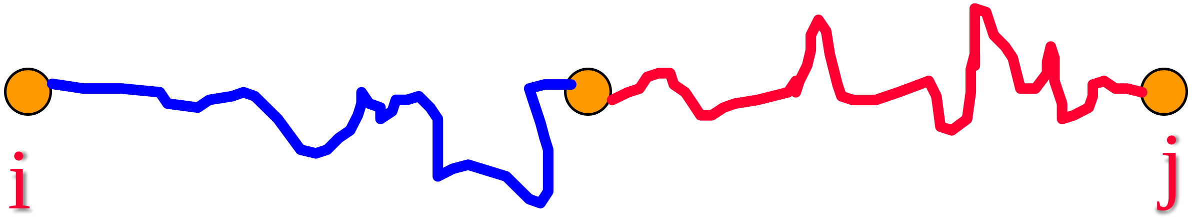
- Time complexity is $\Theta(n^3)$ time.
- Works so long as there is no cycle whose length is < 0.
- When there is a cycle whose length is < 0, some shortest paths aren't finite.
  - If vertex 1 is on a cycle whose length is -2, each time you go around this cycle once you get a 1 to 1 path that is 2 units shorter than the previous one.
- Simpler to code, smaller overheads.

Decision Sequence



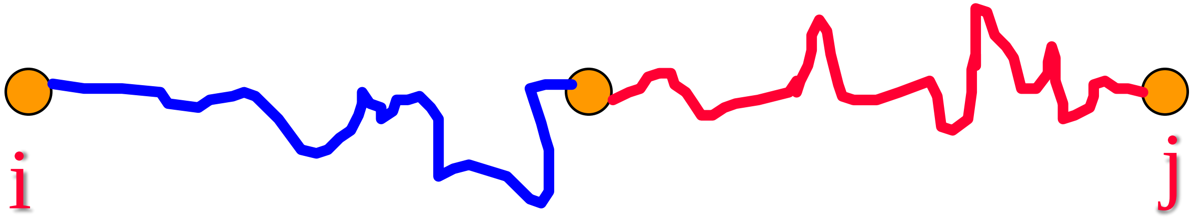
- First decide the highest intermediate vertex (i.e., largest vertex number) on the shortest path from i to j.
- If the shortest path is i, 2, 6, 3, 8, 5, 7, j the first decision is that vertex 8 is an intermediate vertex on the shortest path and no intermediate vertex is larger than 8.
- Then decide the highest intermediate vertex on the path from i to 8, and so on.

A Triple



- (i,j,k) denotes the problem of finding the shortest path from vertex i to vertex j that has no intermediate vertex larger than k.
- (i,j,n) denotes the problem of finding the shortest path from vertex i to vertex j (with no restrictions on intermediate vertices).

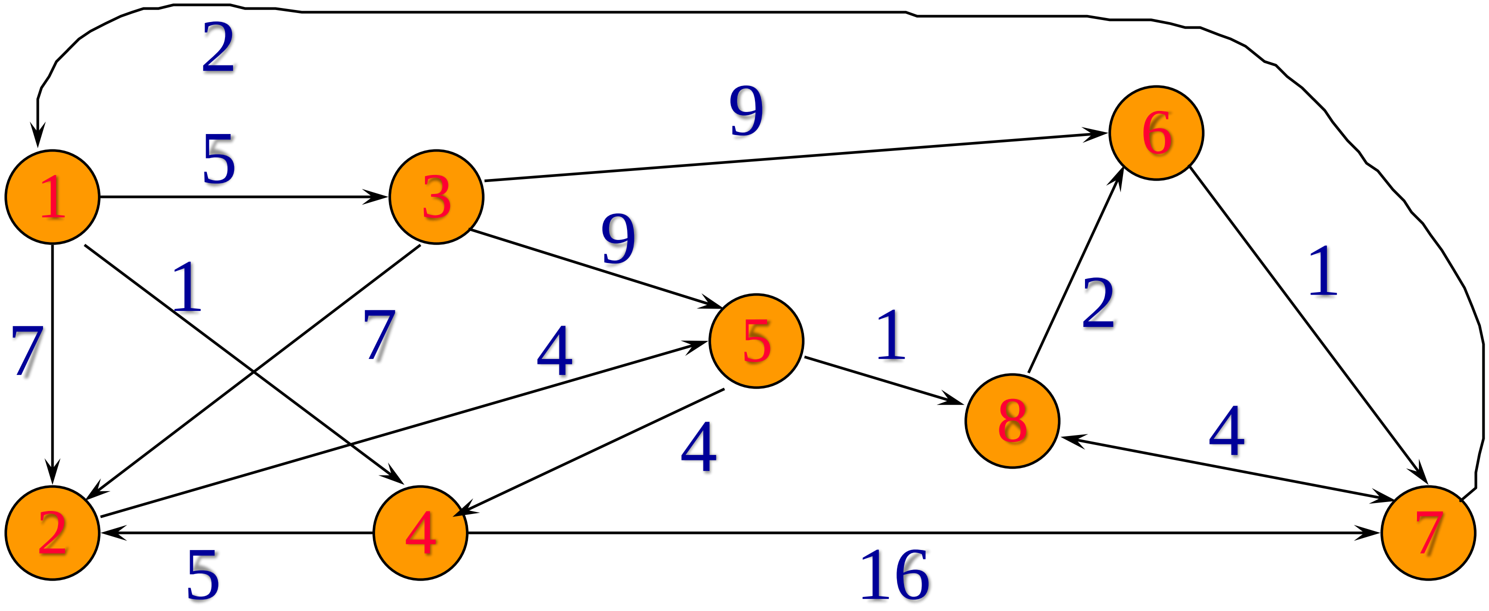
Cost Function



- Let c(i,j,k) be the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than k.

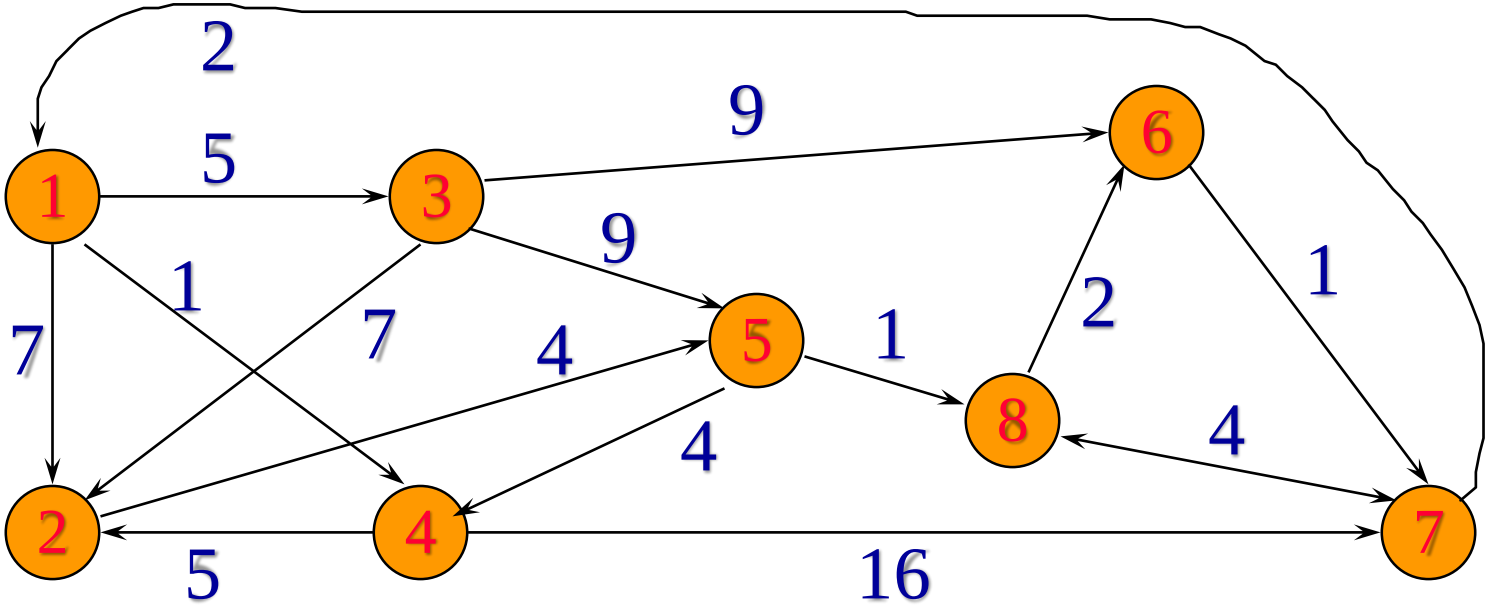
c(i,j,n)

- c(i,j,n) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than n.
- No vertex is larger than n.
- Therefore, c(i,j,n) is the length of a shortest path from vertex i to vertex j.



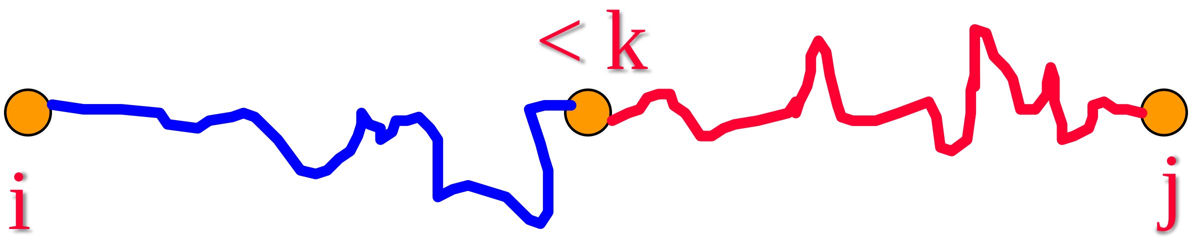
c(i,j,0)

- c(i,j,0) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than 0.
  - Every vertex is larger than 0.
  - Therefore, c(i,j,0) is the length of a single-edge path from vertex i to vertex j.



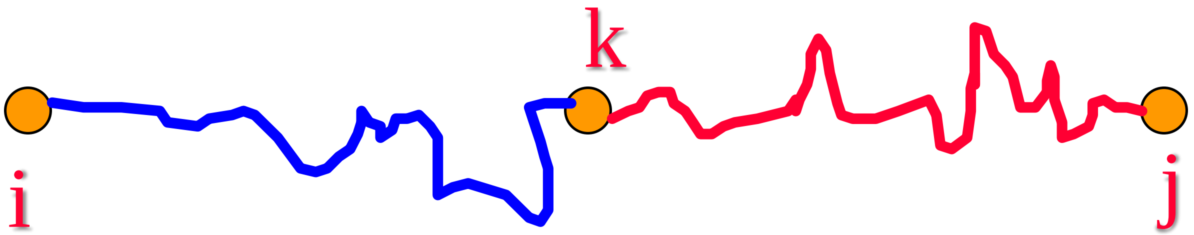
Recurrence For c(i,j,k), k > 0

- The shortest path from vertex i to vertex j that has no intermediate vertex larger than k may or may not go through vertex k.
- If this shortest path does not go through vertex k, the largest permissible intermediate vertex is k-1. So the path length is c(i,j,k-1).



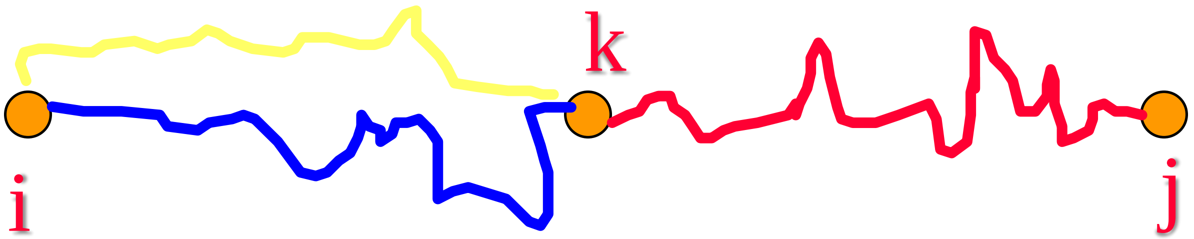
Recurrence For c(i,j,k)), k > 0

- Shortest path goes through vertex k.



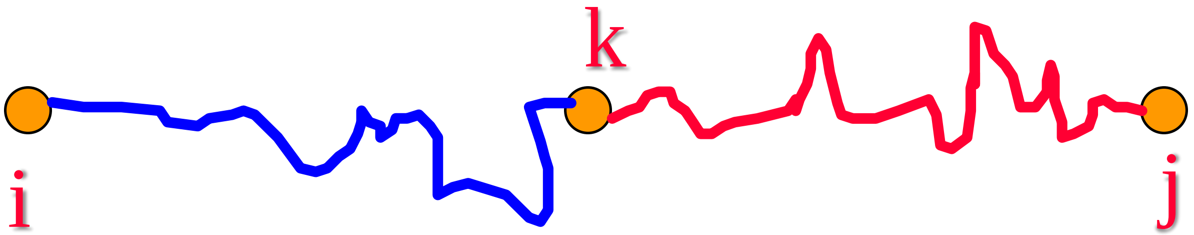
- We may assume that vertex k is not repeated because no cycle has negative length.
- Largest permissible intermediate vertex on i to k and k to j paths is k-1.

Recurrence For c(i,j,k)), k > 0



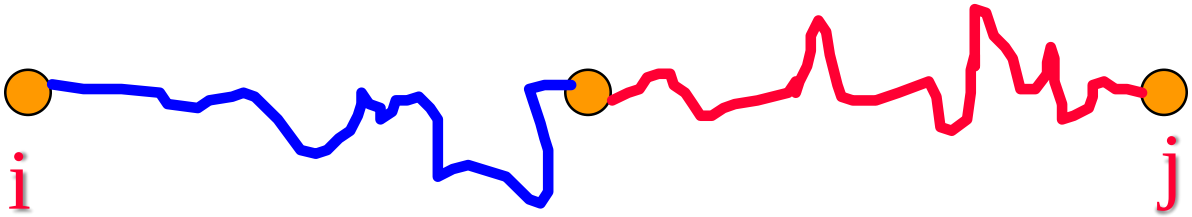
- i to k path must be a shortest i to k path that goes through no vertex larger than k-1.
- If not, replace current i to k path with a shorter i to k path to get an even shorter i to j path.

Recurrence For c(i,j,k)), k > 0



- Similarly, k to j path must be a shortest k to j path that goes through no vertex larger than k-1.
- Therefore, length of i to k path is c(i,k,k-1), and length of k to j path is c(k,j,k-1).
- So, c(i,j,k) = c(i,k,k-1) + c(k,j,k-1).

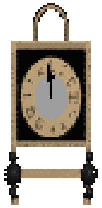
Recurrence For c(i,j,k)), k > 0



- Combining the two equations for c(i,j,k), we get c(i,j,k) = min{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)}.
- We may compute the c(i,j,k)s in the order k = 1, 2, 3, …, n.

Floyd's Shortest Paths Algorithm

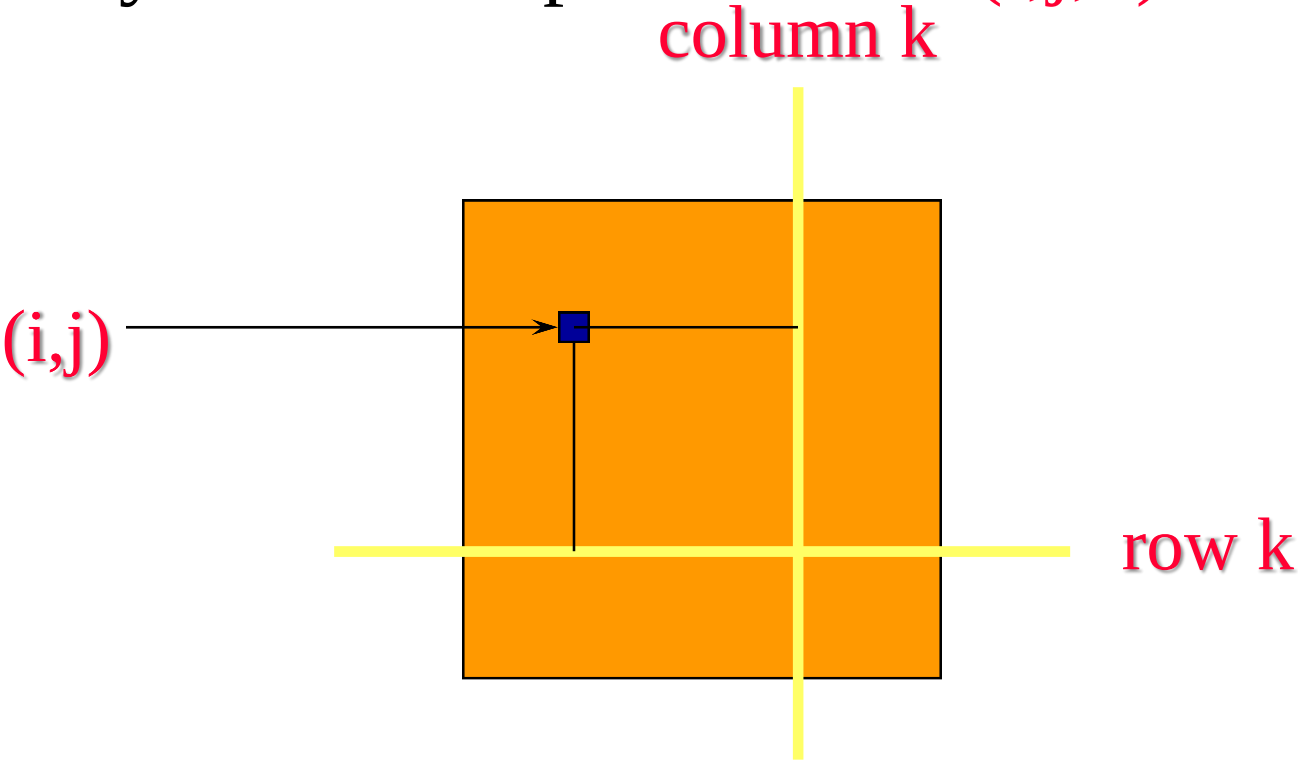
```
for (int k = 1; k <= n; k++)
   for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
         c(i,j,k) = min{c(i,j,k-1),
                           c(i,k,k-1) + c(k,j,k-1)};
```



- Time complexity is $O(n^3)$.
- More precisely $\Theta(n^3)$.
- $\Theta(n^3)$ space is needed for c(*,*,*).

Space Reduction

- c(i,j,k) = min{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)}
- When neither i nor j equals k, c(i,j,k-1) is used only in the computation of c(i,j,k).



- So c(i,j,k) can overwrite c(i,j,k-1).

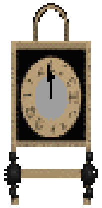
Space Reduction

- $c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\}$
- When i equals k, $c(i,j,k-1)$ equals $c(i,j,k)$.
  - $c(k,j,k) = \min\{c(k,j,k-1), c(k,k,k-1) + c(k,j,k-1)\}$
    $= \min\{c(k,j,k-1), 0 + c(k,j,k-1)\}$
    $= c(k,j,k-1)$
- So, when i equals k, $c(i,j,k)$ can overwrite $c(i,j,k-1)$.
- Similarly when j equals k, $c(i,j,k)$ can overwrite $c(i,j,k-1)$.
- So, in all cases $c(i,j,k)$ can overwrite $c(i,j,k-1)$.

Floyd’s Shortest Paths Algorithm

```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      c(i,j) = min{c(i,j), c(i,k) + c(k,j)};
```

- Initially, $c(i,j) = c(i,j,0)$.
- Upon termination, $c(i,j) = c(i,j,n)$.
- Time complexity is $\Theta(n^3)$.
- $\Theta(n^2)$ space is needed for $c(*,*)$.

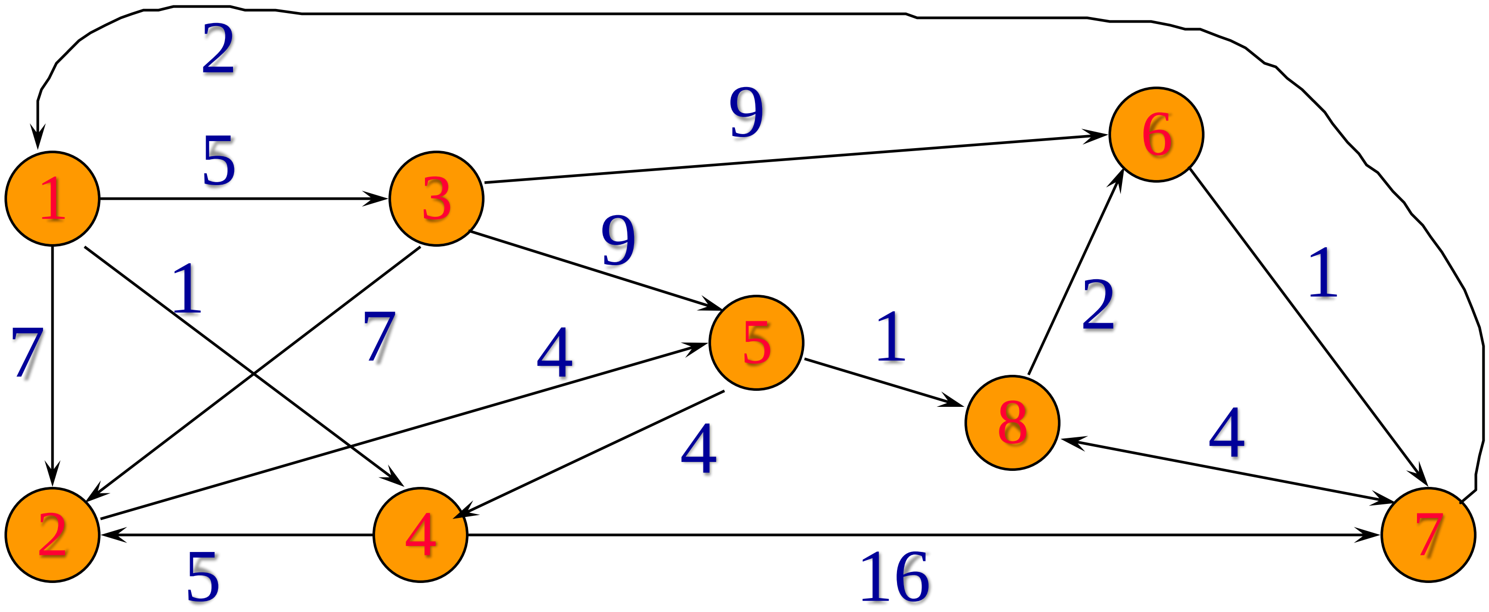


Building The Shortest Paths

- Let kay(i,j) be the largest vertex on the shortest path from i to j.
- Initially, kay(i,j) = 0 (shortest path has no intermediate vertex).

```
for (int k = 1; k <= n; k++)
   for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
         if (c(i,j) > c(i,k) + c(k,j))
            {kay(i,j) = k; c(i,j) = c(i,k) + c(k,j);}
```

Example



-	7	5	1	-	-	-	-
-	-	-	-	4	-	-	-
-	7	-	-	9	9	-	-
-	5	-	-	-	-	16	-
-	-	-	4	-	-	-	1
-	-	-	-	-	-	1	-
2	-	-	-	-	-	-	4
-	-	-	-	-	2	4	-

Initial Cost Matrix

$c(*,*) = c(*,*,0)$

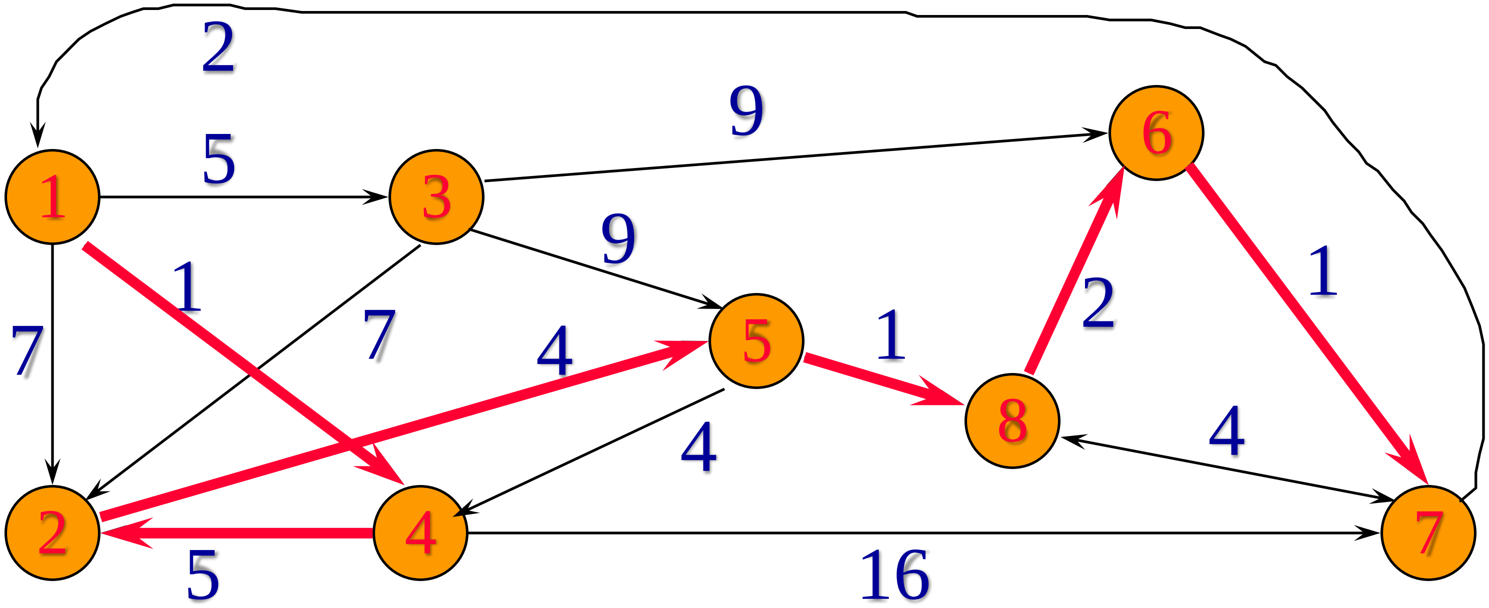
Final Cost Matrix c(*,*) = c(*,*,n)

$$\begin{matrix}
0 & 6 & 5 & 1 & 10 & 13 & 14 & 11 \\
10 & 0 & 15 & 8 & 4 & 7 & 8 & 5 \\
12 & 7 & 0 & 13 & 9 & 9 & 10 & 10 \\
15 & 5 & 20 & 0 & 9 & 12 & 13 & 10 \\
6 & 9 & 11 & 4 & 0 & 3 & 4 & 1 \\
3 & 9 & 8 & 4 & 13 & 0 & 1 & 5 \\
2 & 8 & 7 & 3 & 12 & 6 & 0 & 4 \\
5 & 11 & 10 & 6 & 15 & 2 & 3 & 0
\end{matrix}$$

kay Matrix

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

Shortest Path



Shortest path from 1 to 7.

Path length is 14.

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.
- kay(1,7) = 8

$1 \longrightarrow 8 \longrightarrow 7$

- kay(1,8) = 5

$1 \longrightarrow 5 \longrightarrow 8 \longrightarrow 7$

- kay(1,5) = 4

$1 \longrightarrow 4 \longrightarrow 5 \longrightarrow 8 \longrightarrow 7$

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.

1 → 4 → 5 → 8 → 7

- kay(1,4) = 0

1 4 → 5 → 8 → 7

- kay(4,5) = 2

1 4 → 2 → 5 → 8 → 7

- kay(4,2) = 0

1 4 2 → 5 → 8 → 7

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.

1 4 2 → 5 → 8 → 7

- kay(2,5) = 0

1 4 2 5 → 8 → 7

- kay(5,8) = 0

1 4 2 5 8 → 7

- kay(8,7) = 6

1 4 2 5 8 → 6 → 7

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.

  1 4 2 5 8 → 6 → 7

- kay(8,6) = 0

  1 4 2 5 8 6 → 7

- kay(6,7) = 0

  1 4 2 5 8 6 7

Output A Shortest Path

```
void outputPath(int i, int j)
{// does not output first vertex (i) on path
  if (i == j) return;
  if (kay[i][j] == 0)  // no intermediate vertices on path
    cout << j << " ";
  else {// kay[i][j] is an intermediate vertex on the path
          outputPath(i, kay[i][j]);
          outputPath(kay[i][j], j);
       }
}
```

Time Complexity Of outputPath

O(number of vertices on shortest path)