GITAM
(DEEMED TO BE UNIVERSITY)
(Estd. u/s 3 of the UGC Act, 1956)
VISAKHAPATNAM • HYDERABAD • BENGALURU
Accredited by NAAC with A+ Grade

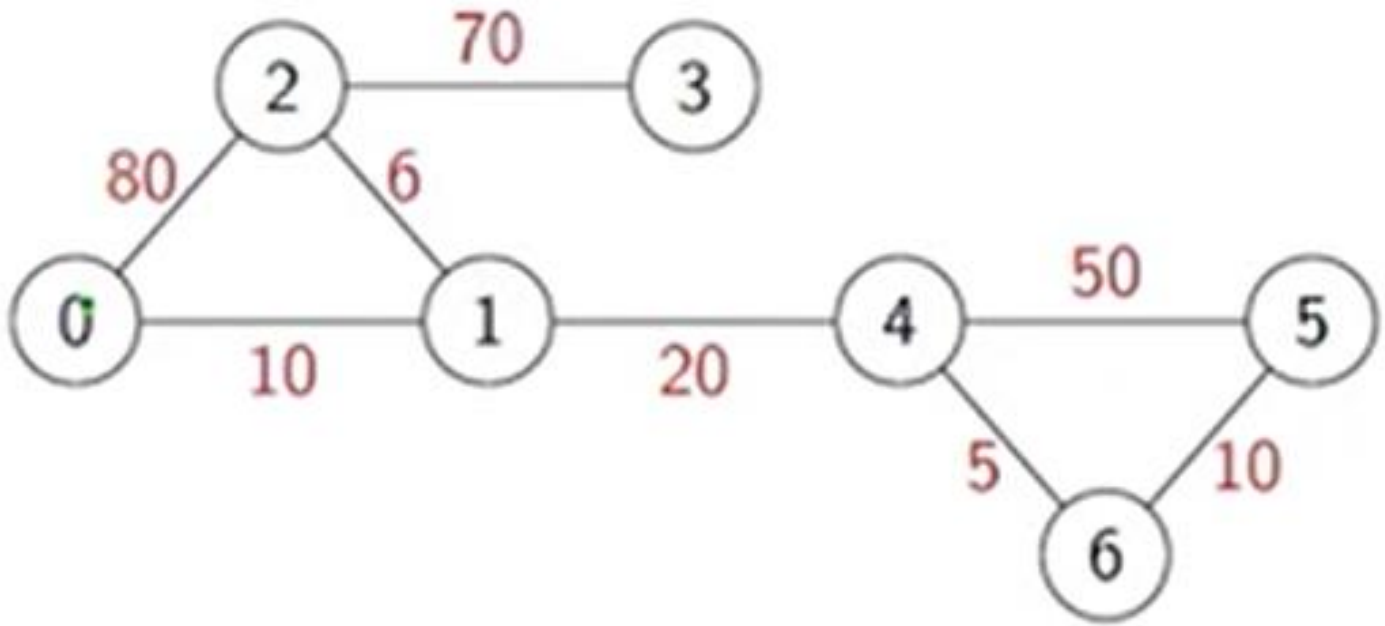
Design and Analysis of Algorithms
Module II: Greedy Method
Topic: Single Source Shortest Path(Dijkstra's Algorithm)

Prof. PV Nageswara Rao

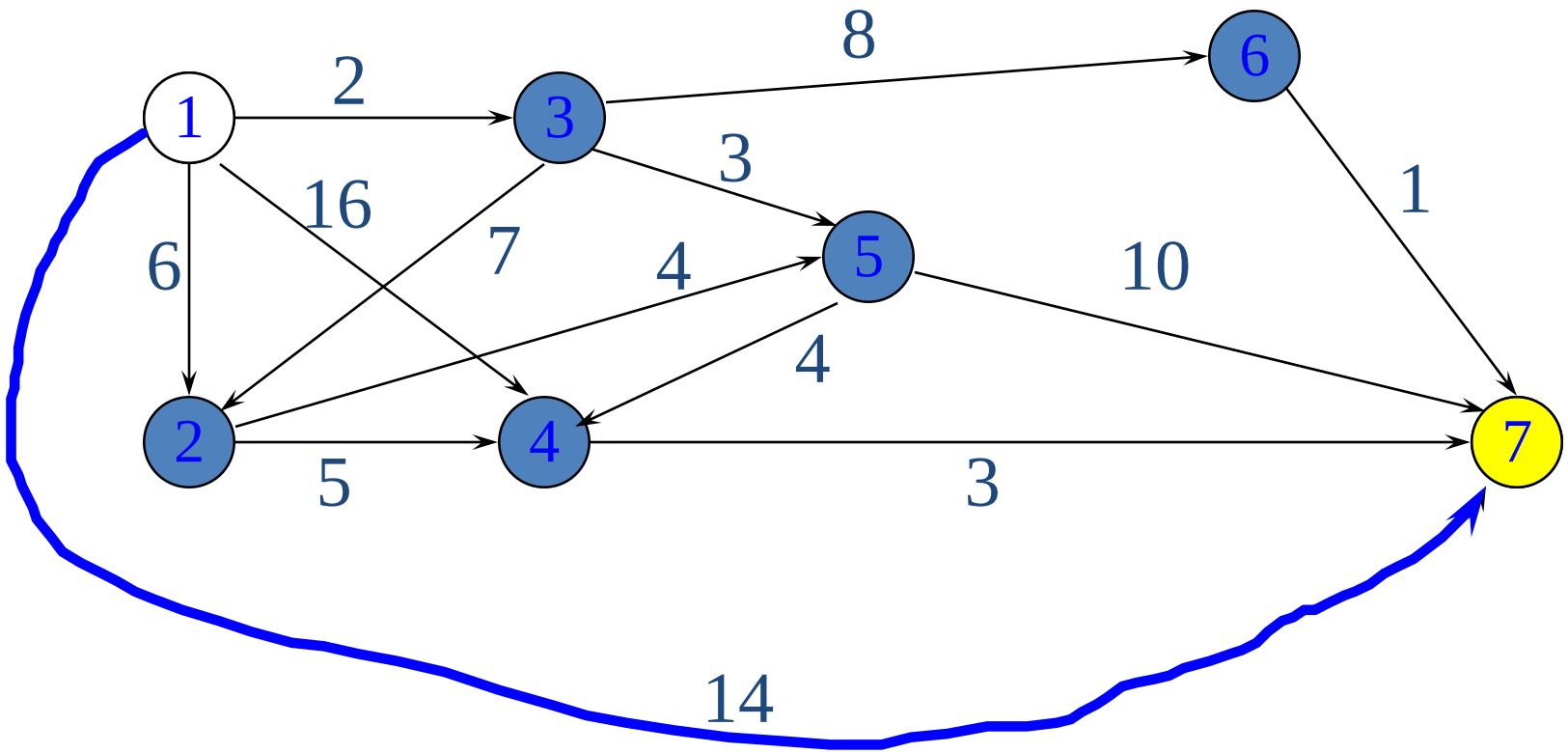
Dept. of CSE, GIT, GU, Visakhapatnam

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.


70
2
3
80
6
0
1
4
50
5
10
20
5
10
6

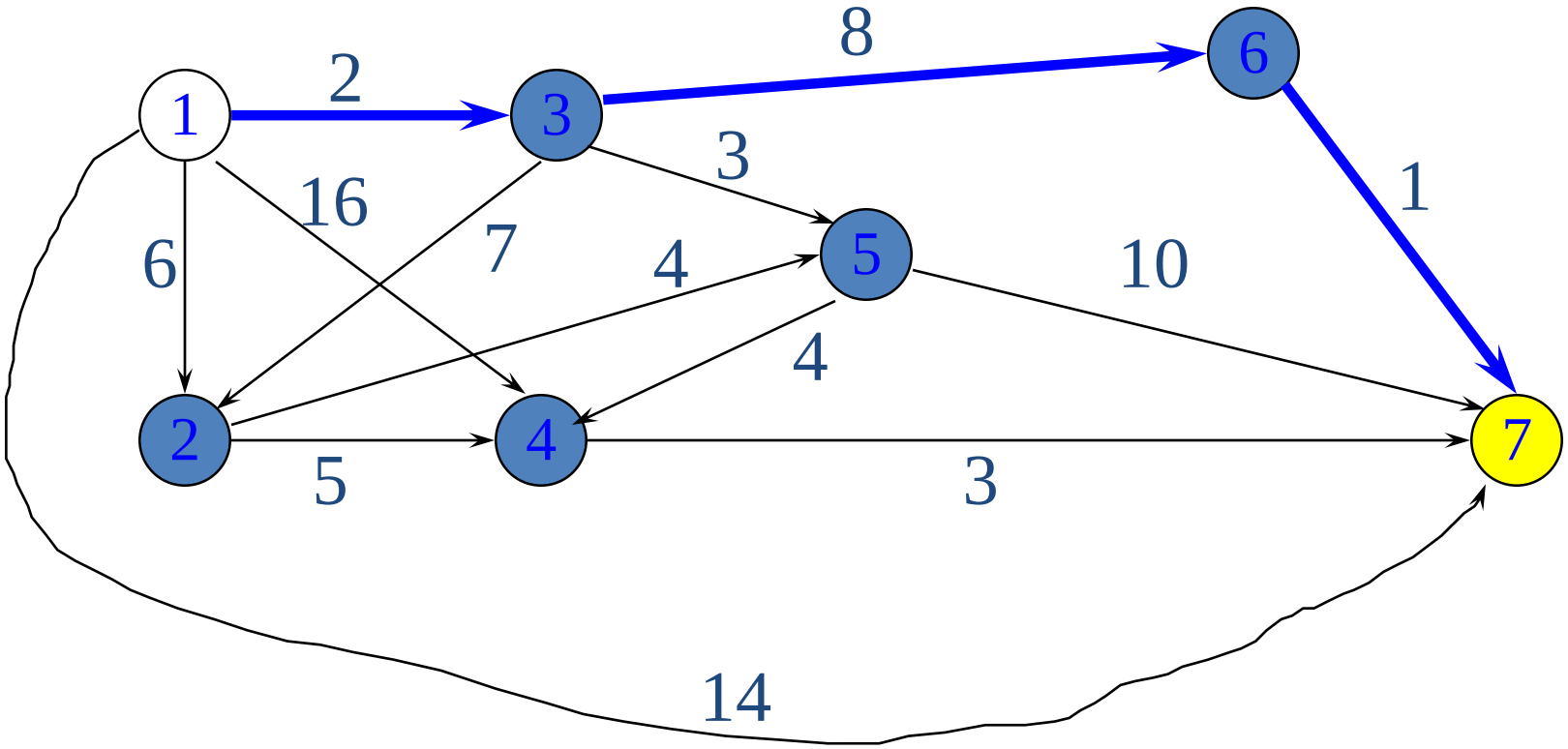
Example



A path from 1 to 7.

Path length is 14.

Example



Another path from 1 to 7.

Path length is 11.

Shortest Path Problems

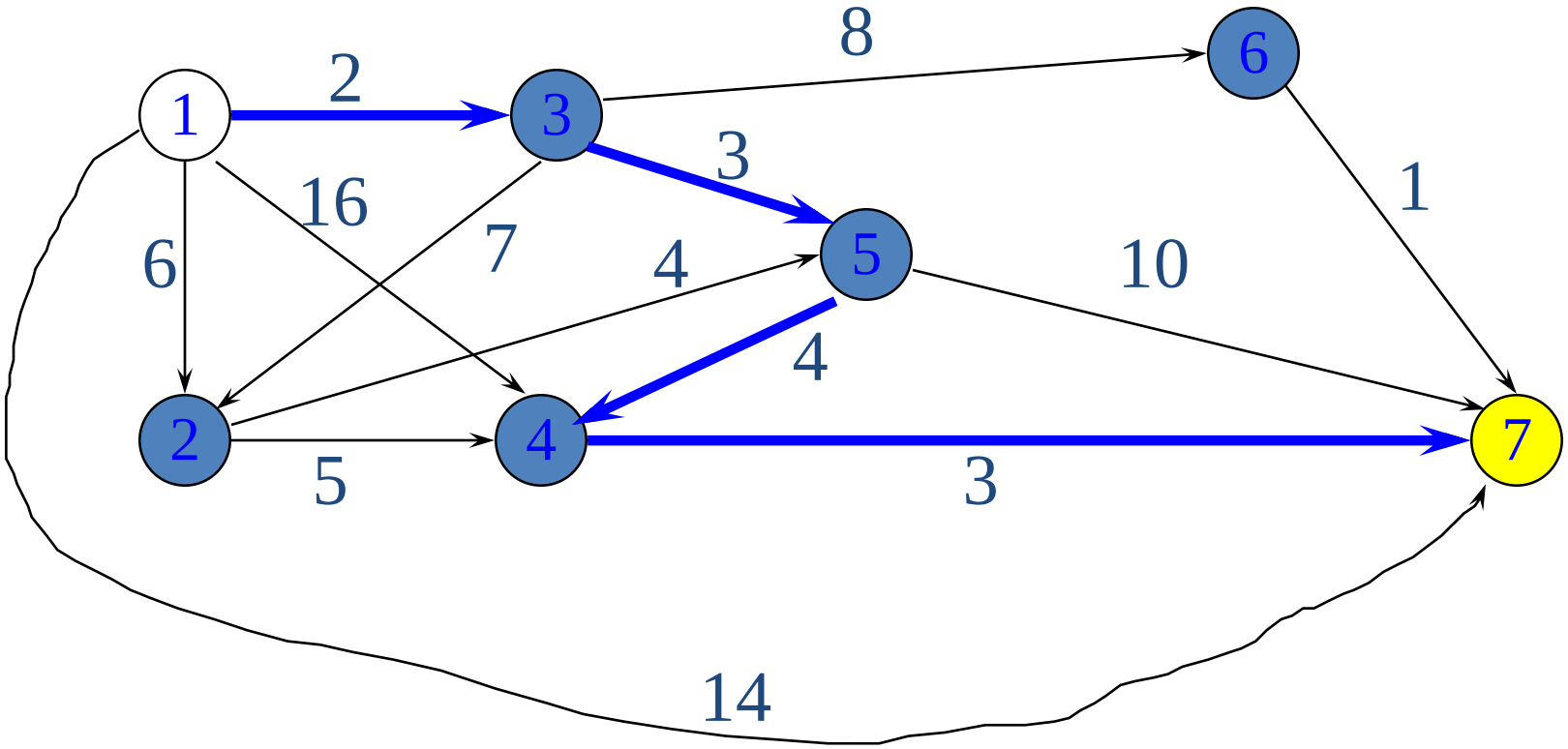
- Single source single destination.
- Single source all destinations.
- All pairs (every vertex is a source and destination).

Single Source Single Destination

Possible greedy 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.

Greedy Shortest 1 To 7 Path



Path length is 12.

Not shortest path. Algorithm doesn't work!

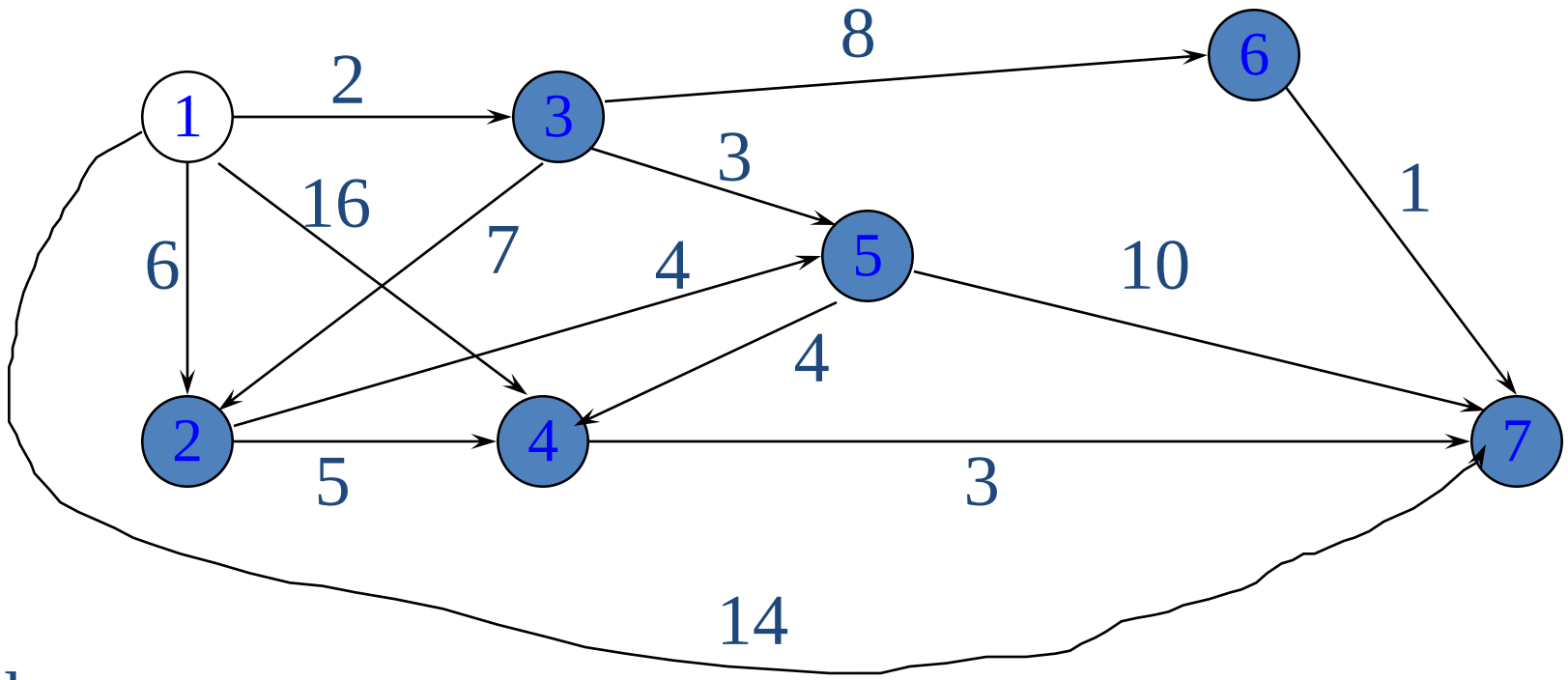
Single Source All Destinations

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

Greedy 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.

Greedy 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

Greedy Single Source All Destinations

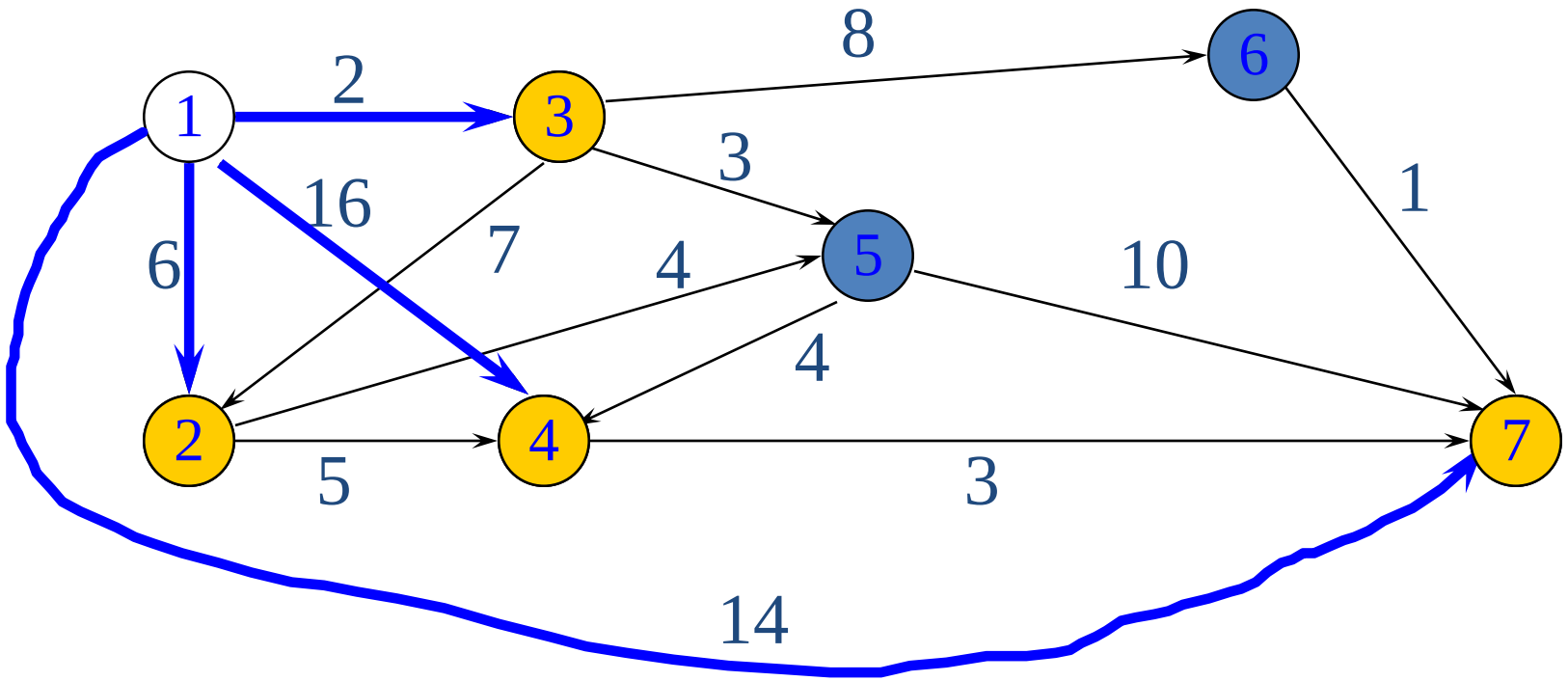
Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

- 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.

Greedy Single Source All Destinations

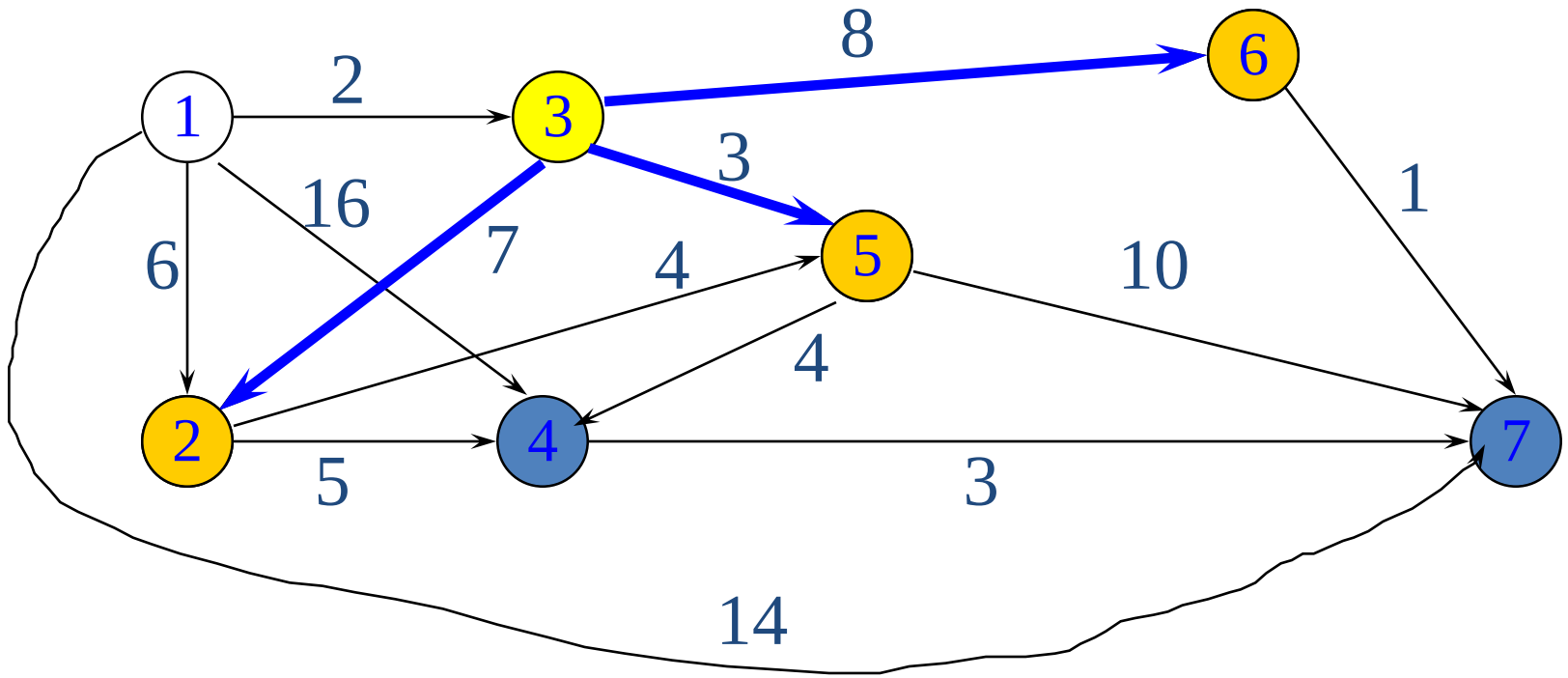
- Let d(i) (distanceFromSource(i)) be the length of a shortest one edge extension of an already generated shortest path, the one edge extension ends at vertex i.
- The next shortest path is to an as yet unreached vertex for which the d() value is least.
- Let p(i) (predecessor(i)) be the vertex just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

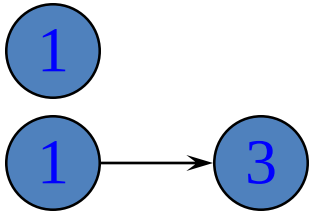
Greedy Single Source All Destinations



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

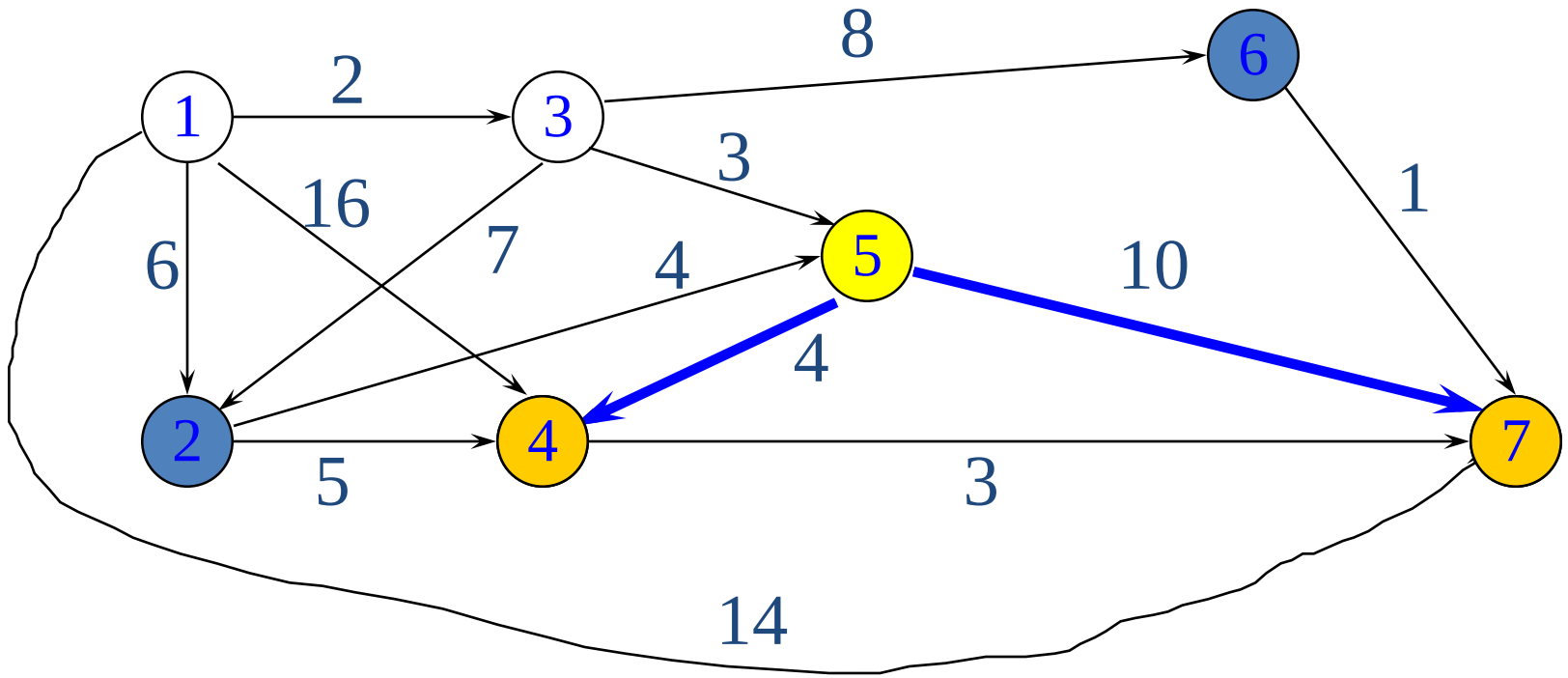
Greedy Single Source All Destinations

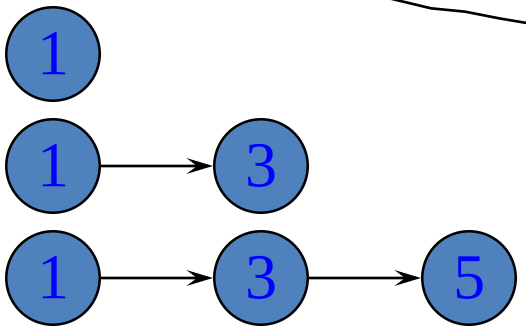




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	16	5	10	14
p	-	1	1	1	3	3	1

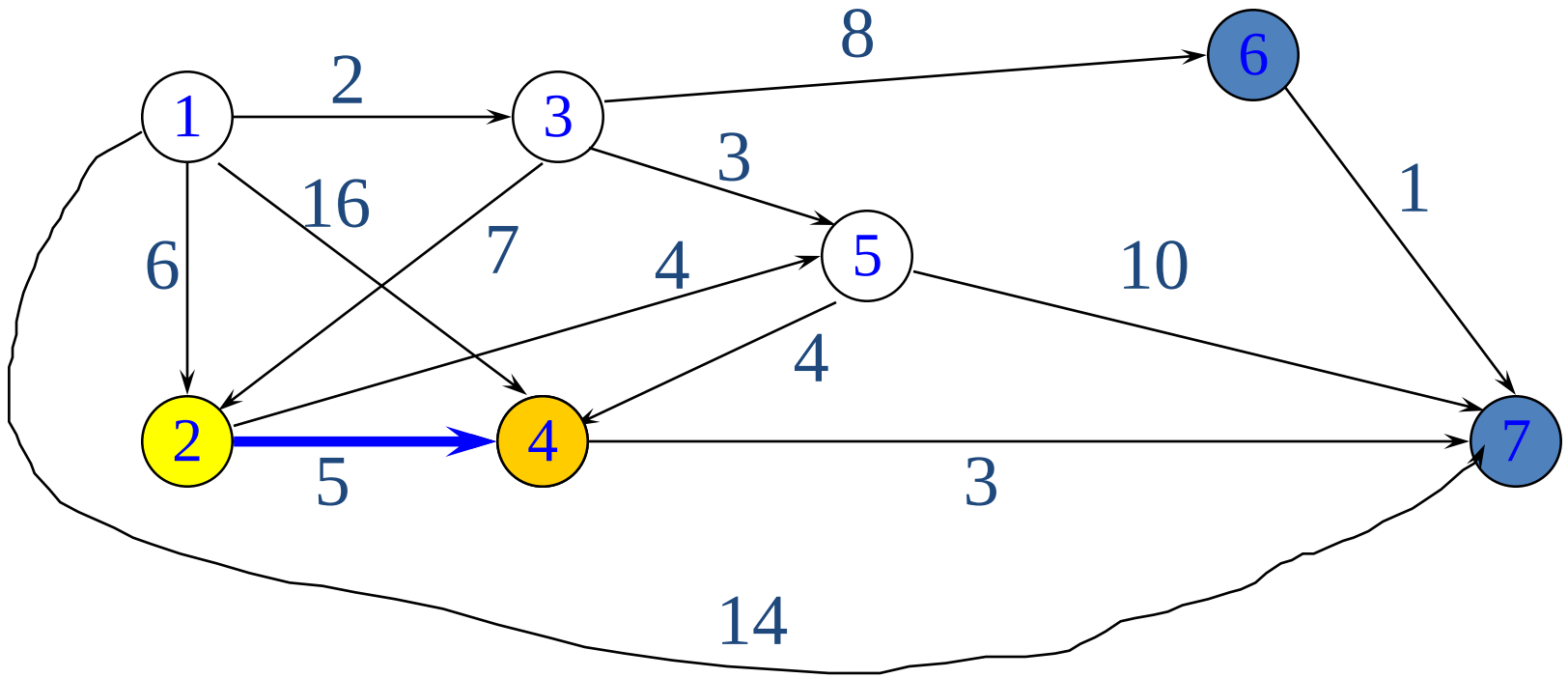
Greedy Single Source All Destinations

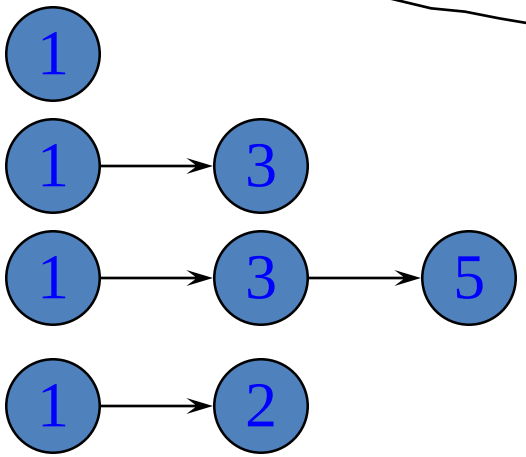




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

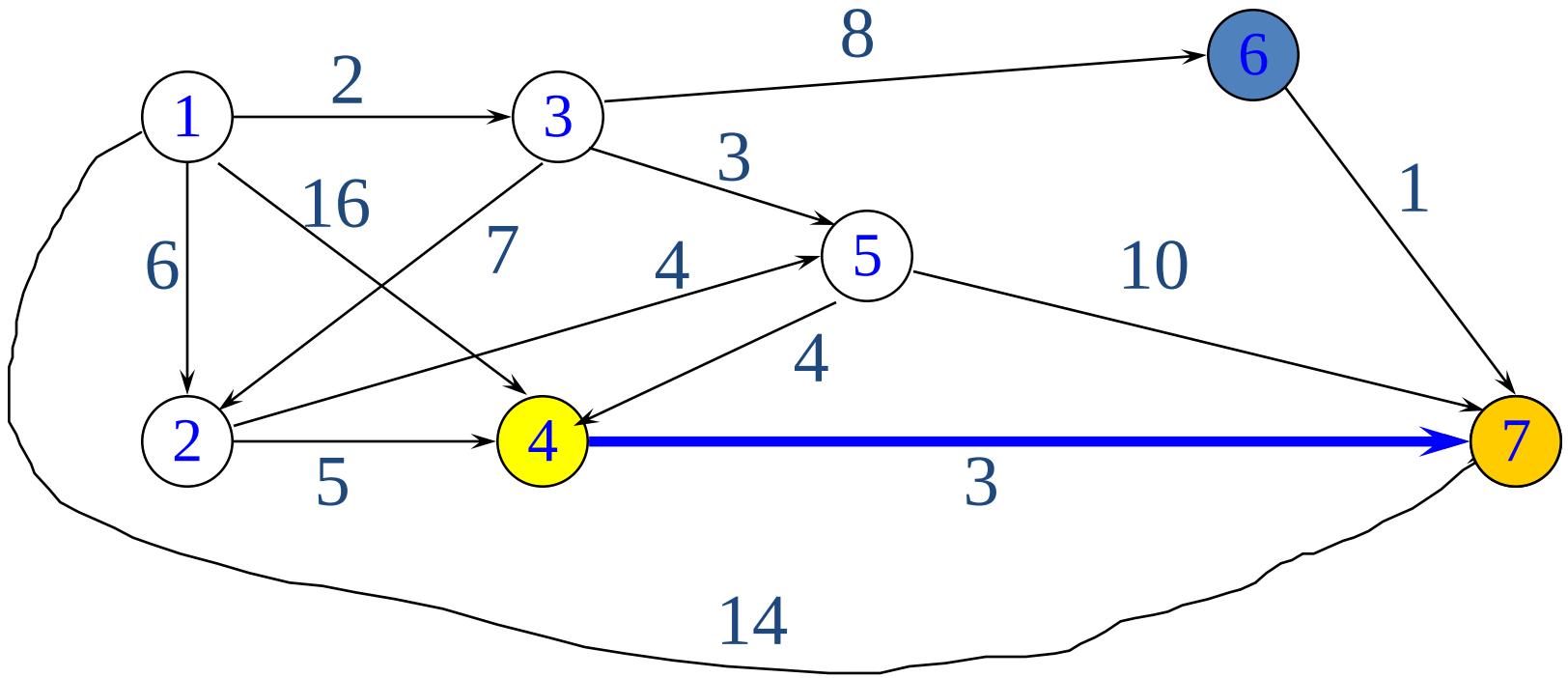
Greedy Single Source All Destinations

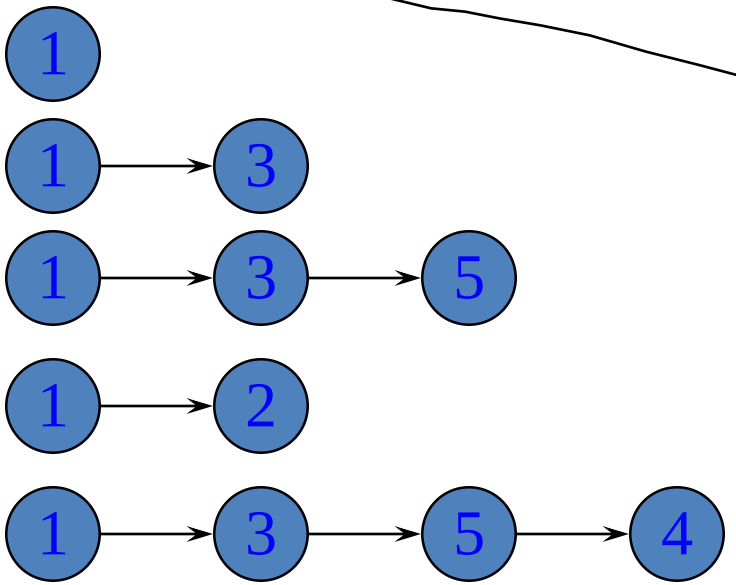




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

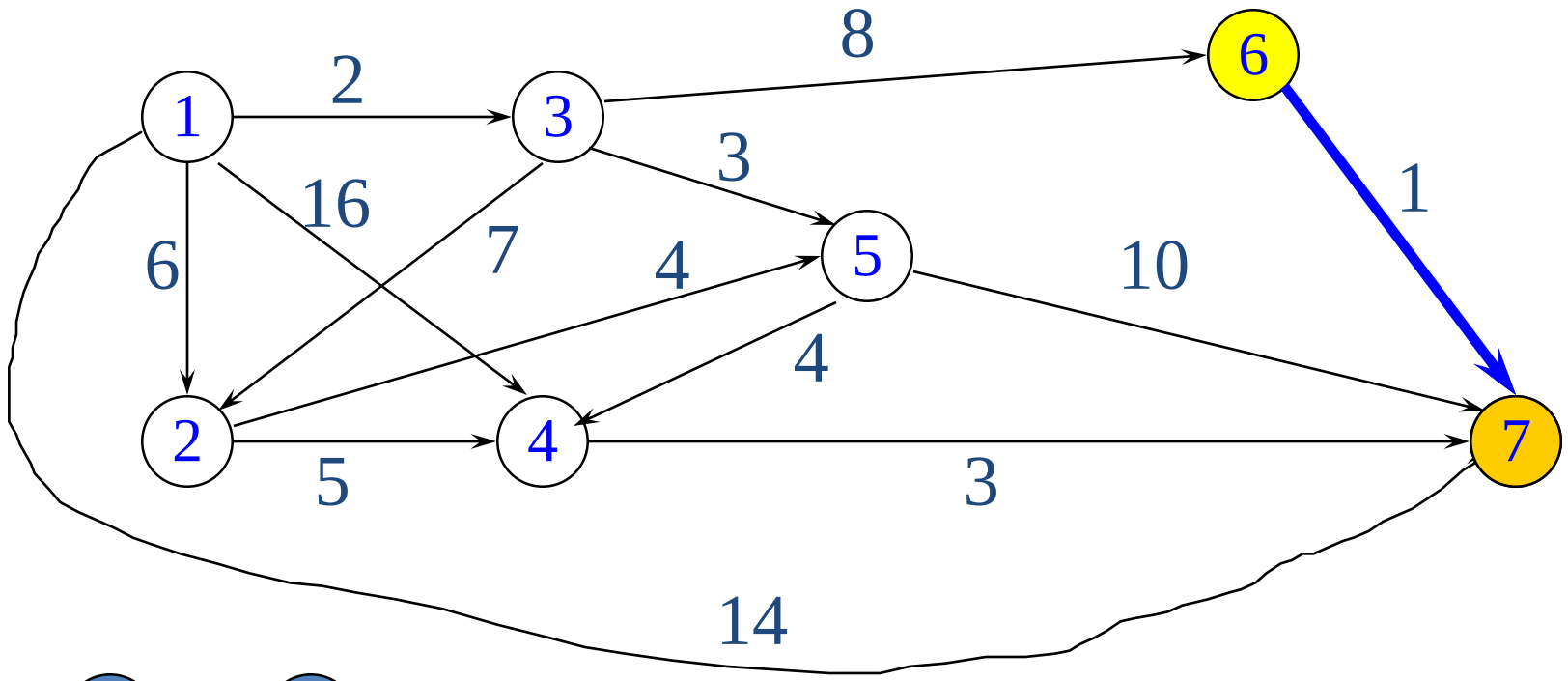
Greedy Single Source All Destinations

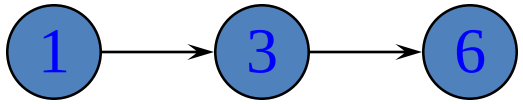




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	12
p	-	1	1	5	3	3	4

Greedy 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

Greedy 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

Data Structures For Dijkstra's Algorithm

- The greedy 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

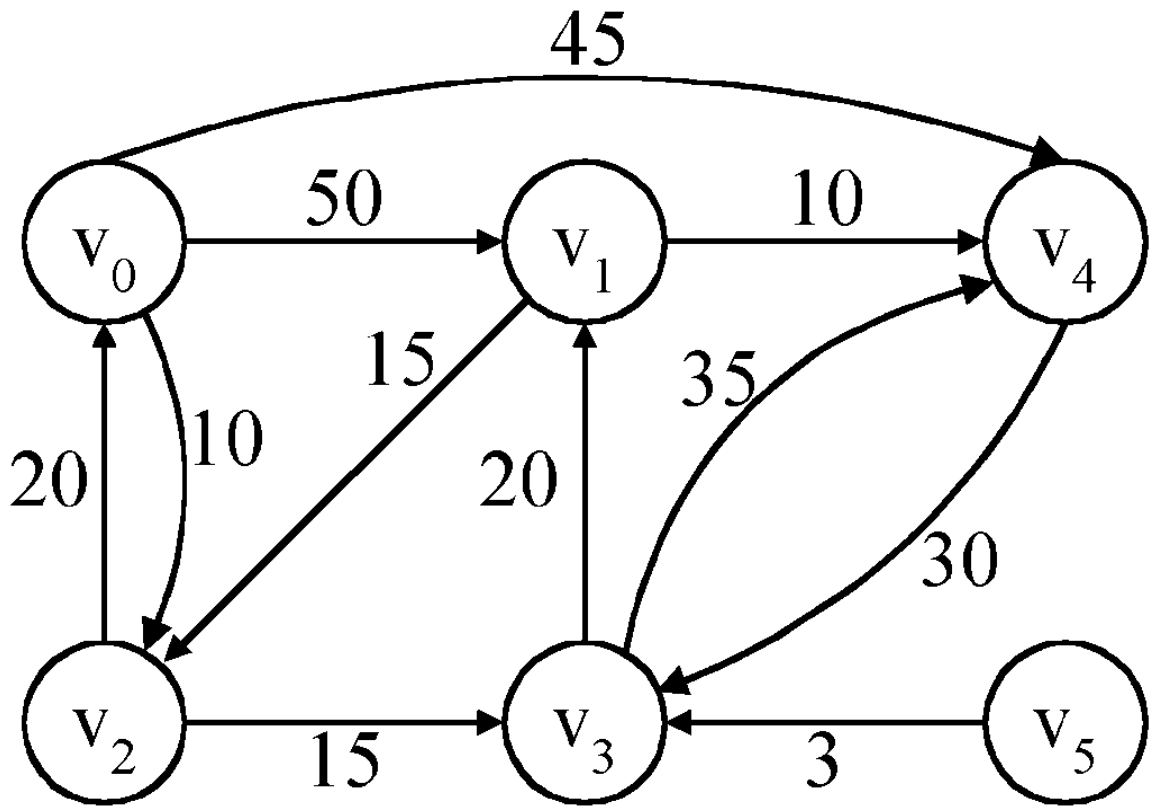
- O(n) to select next destination vertex.
- O(out-degree) to update d() and p() values when adjacency lists are used.
- O(n) to update d() and p() values when adjacency matrix is used.
- Selection and update done once for each vertex to which a shortest path is found.
- Total time is $O(n^2 + e) = O(n^2)$.

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 Shortest path: Example

- shortest paths from v_0 to all destinations

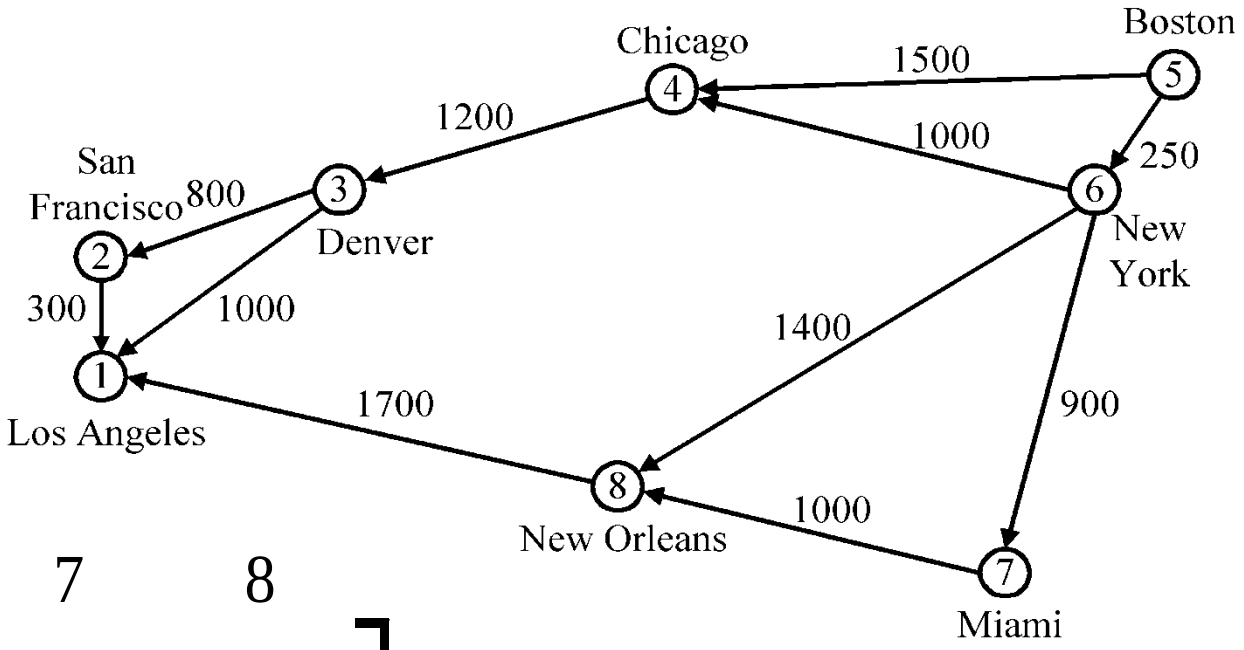


(a)

	Path	Length
1)	v_0v_2	10
2)	$v_0v_2v_3$	25
3)	$v_0v_2v_3v_1$	45
4)	v_0v_4	45

(b)

Dijkstra's Algorithm:



In the cost adjacency matrix, all entries not shown are $+\infty$.

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

Example for the Shortest Path

(Continued)

Iteration	S	Vertex Selected	DIST	LA (1)	SF (2)	D (3)	C (4)	B (5)	NY (6)	M (7)	NO (8)
Initial		—									
1	5	6		$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
2	5,6	7		$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	5,6,7	4		$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
4	5,6,7,4	8		$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
5	5,6,7,4,8	3		3350	$+\infty$	2450	1250	0	250	1150	1650
6	5,6,7,4,8,3	2		3350	3250	2450	1250	0	250	1150	1650
	5,6,7,4,8,3,2			3350	3250	2450	1250	0	250	1150	1650

Single Source Shortest Path Algorithm(One to all) : Dijkstra's Algorithm

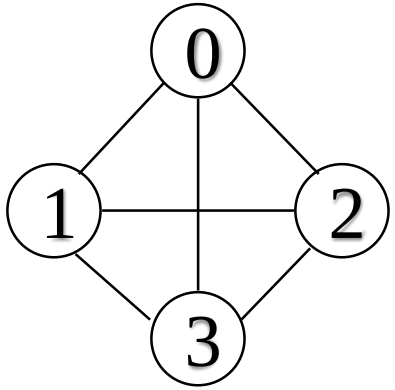
```
Algorithm shortestpaths(v, cost, dist, n)
//dist[j] 1<=j<=n is set to the length of the shortest path from vertex v to j
// in the digraph G with n vertices. dist[v] is set to zero. G is represented by
// its cost adjacency matrix cost[1:n,1:n].
{ for i:= 1 to n do
   { S[i]:=false; dist[i]=cost[v,i];}
 s[v] := true; dist[v]=0; // put v in S
 for num:=2 to n do
  {// determine n-1 paths from v;
    Choose u from among those vertices not in S sucht that dist[u] is minimum
     S[u]:=true; //put u in S
      for (each w adjacent to u with s[w]=false do
          if (dist[w]>dist[u]+cost[u,w]) then dist[w]:=dist[u]+cost[u,w]
   } // end of for
} // end of algorithm
```

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                 if not visited[v]])
    nextvlist = [v for v in range(rows)
                 if (not visited[v]) and
                    distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                      +WMat[nextv,v
  return(distance)
```

Definition

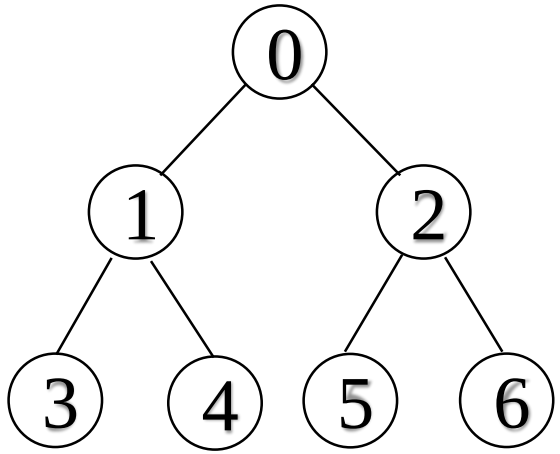
- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty, set of edges E(G)
  - G(V,E) represents a graph
- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $\langle v_0, v_1 \rangle$!= $\langle v_1, v_0 \rangle$

Examples for Graph

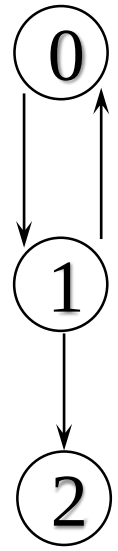


G_1

complete graph



G_2



incomplete graph

G_3

$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

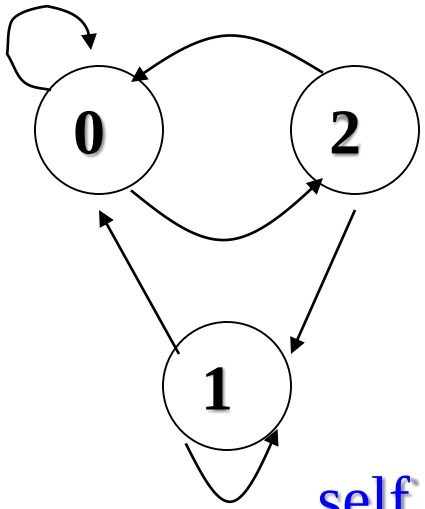
Complete Graph

- A complete graph is a graph that has the maximum number of edges
  - for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
  - for directed graph with n vertices, the maximum number of edges is n(n-1)
  - example: G1 is a complete graph

Adjacent and Incident

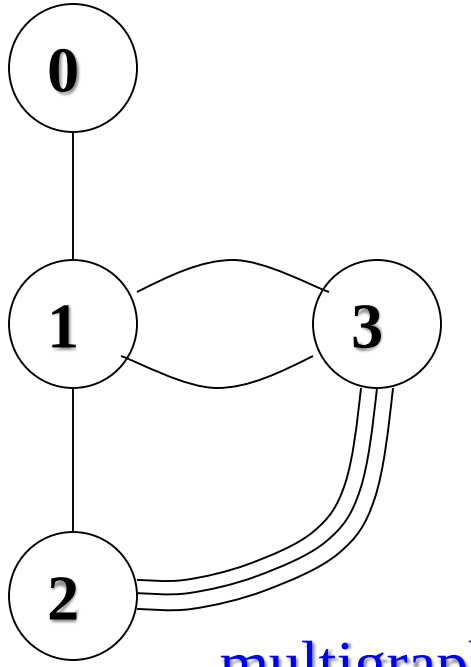
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

Example of a graph with feedback loops and a multigraph



self edge

(a)



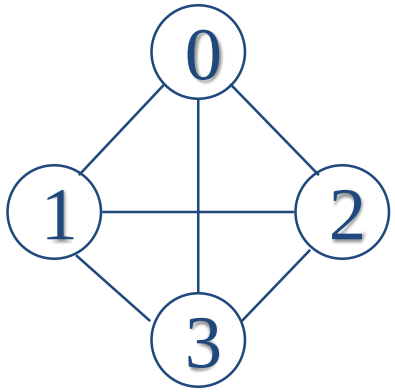
(b)

multigraph:
multiple occurrences
of the same edge

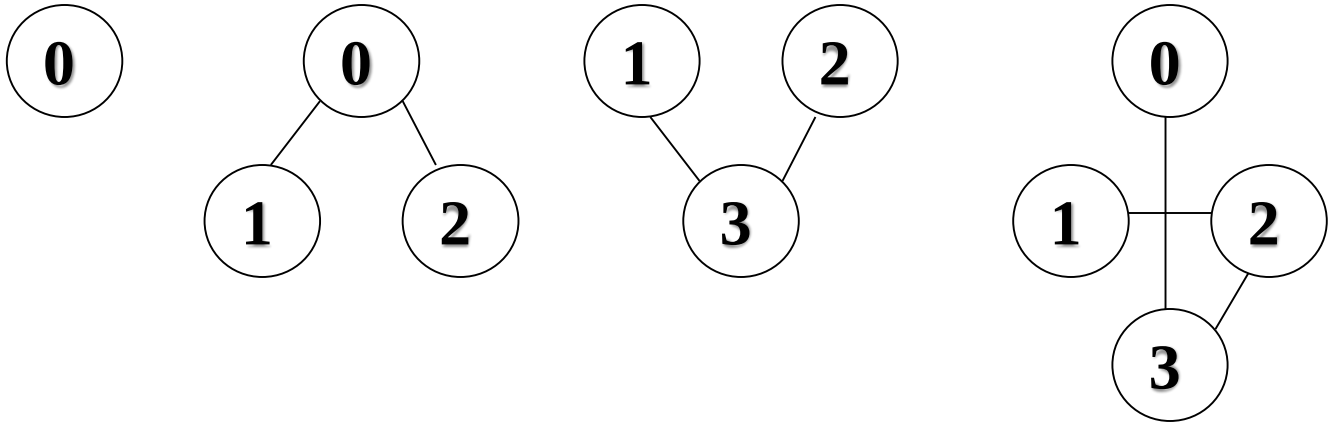
Subgraph and Path

- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph
- The length of a path is the number of edges on it

subgraphs of G_1 and G_3

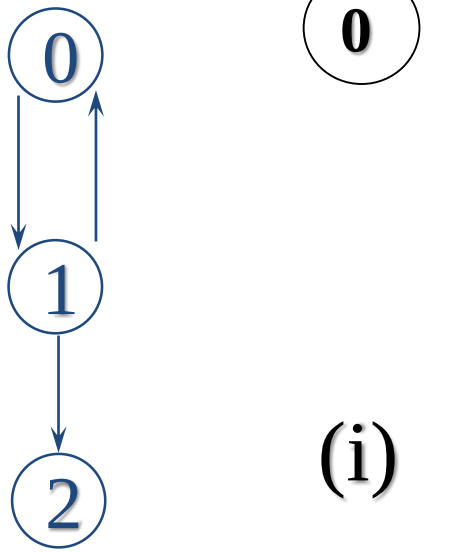


G_1

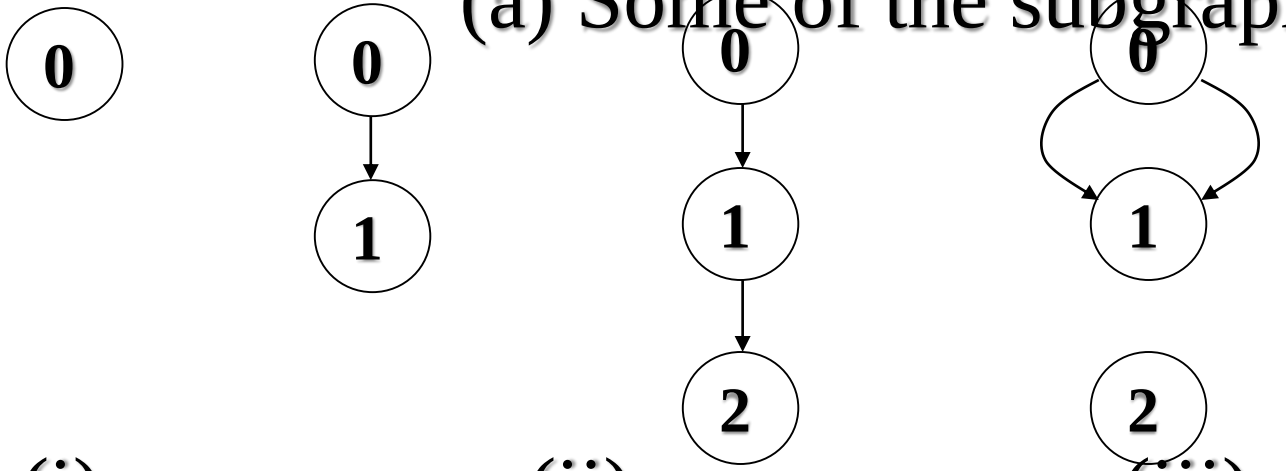


(i) (ii) (iii)

(a) Some of the subgraph of G_1



G_3



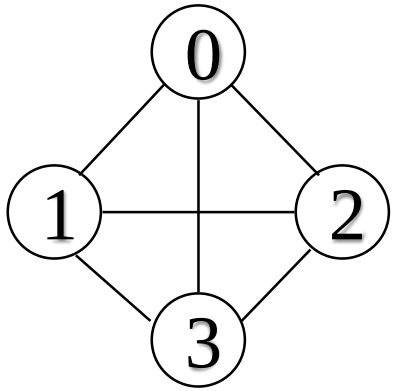
(i) (ii) (iii)

(b) Some of the subgraph of G_3

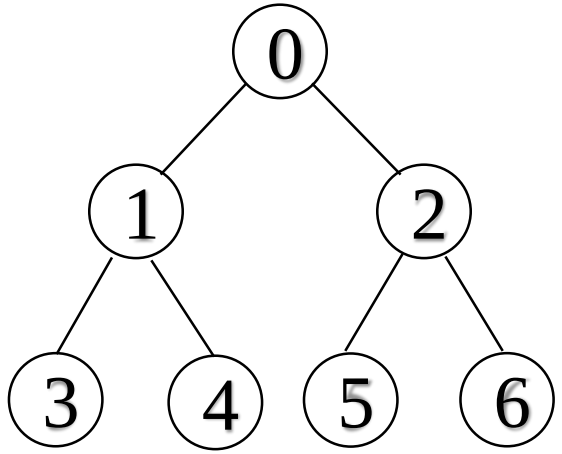
Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two vertices, v_0 and v_1, are connected if there is a path in G from v_0 to v_1
- An undirected graph is connected if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j

connected



G_1



G_2

tree (acyclic graph)

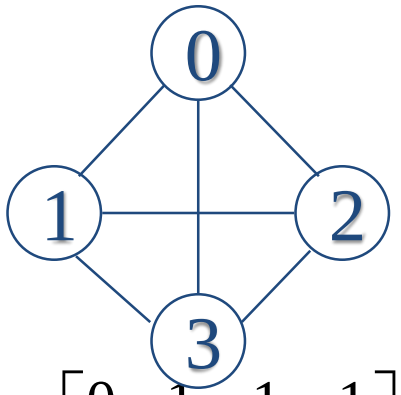
Connected Component

- A connected component of an undirected graph is a maximal connected subgraph.
- A tree is a graph that is connected and acyclic.
- A directed graph is strongly connected if there is a directed path from v_i to v_j and also from v_j to v_i.
- A strongly connected component is a maximal subgraph that is strongly connected.

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

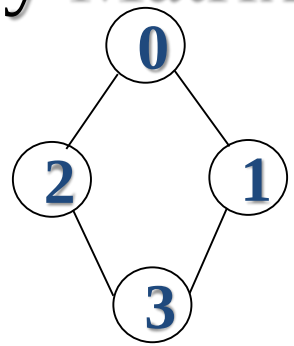
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

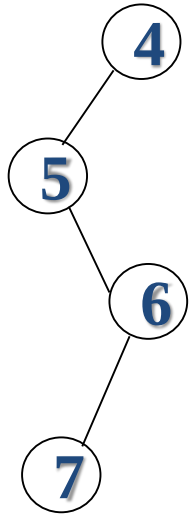
G_2

symmetric

undirected: $n^2/2$

directed: n^2





$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

Data Structures for Adjacency Lists

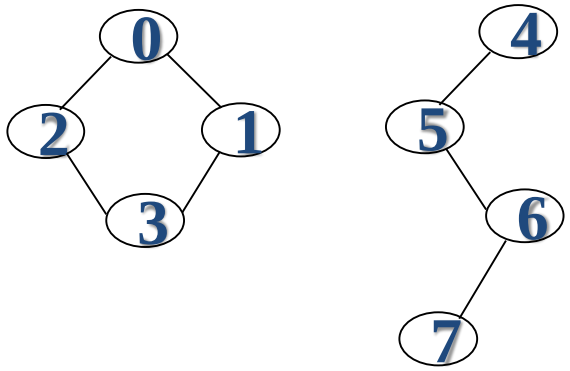
Each row in adjacency matrix is represented as an adjacency list.

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Interesting Operations

- degree of a vertex in an undirected graph
  - # of nodes in adjacency list
- # of edges in a graph
  - determined in O(n+e)
- out-degree of a vertex in a directed graph
  - # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - traverse the whole data structure

Compact Representation



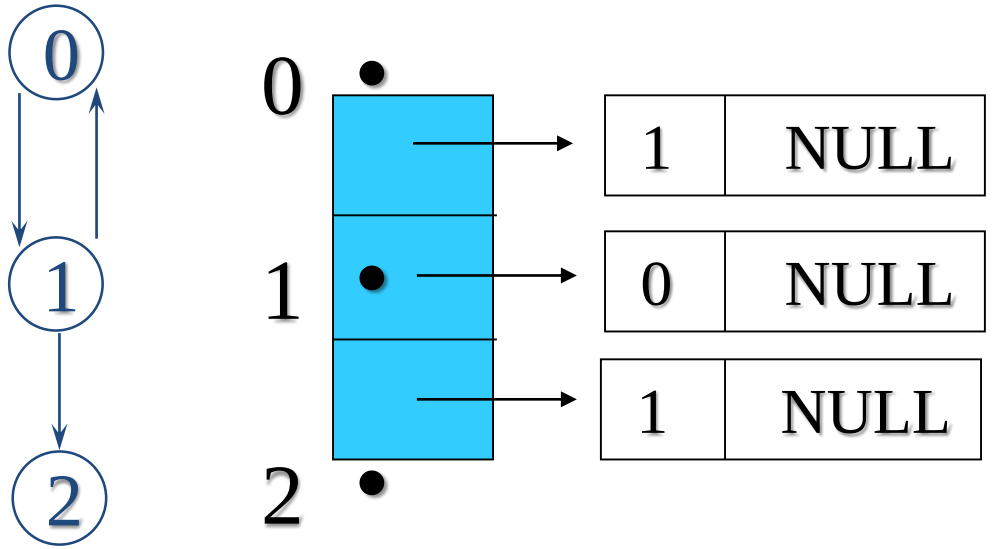
node[0] … node[n-1]: starting point for v

node[n]: n+2e+1

node[n+1] … node[n+2e]: head node of e

[0]	9		[8]	23		[16]	2	
[1]	11	0	[9]	1	4	[17]	5	
[2]	13		[10]	2	5	[18]	4	
[3]	15	1	[11]	0		[19]	6	
[4]	17		[12]	3	6	[20]	5	
[5]	18	2	[13]	0		[21]	7	
[6]	20		[14]	3	7	[22]	6	
[7]	22	3	[15]	1				

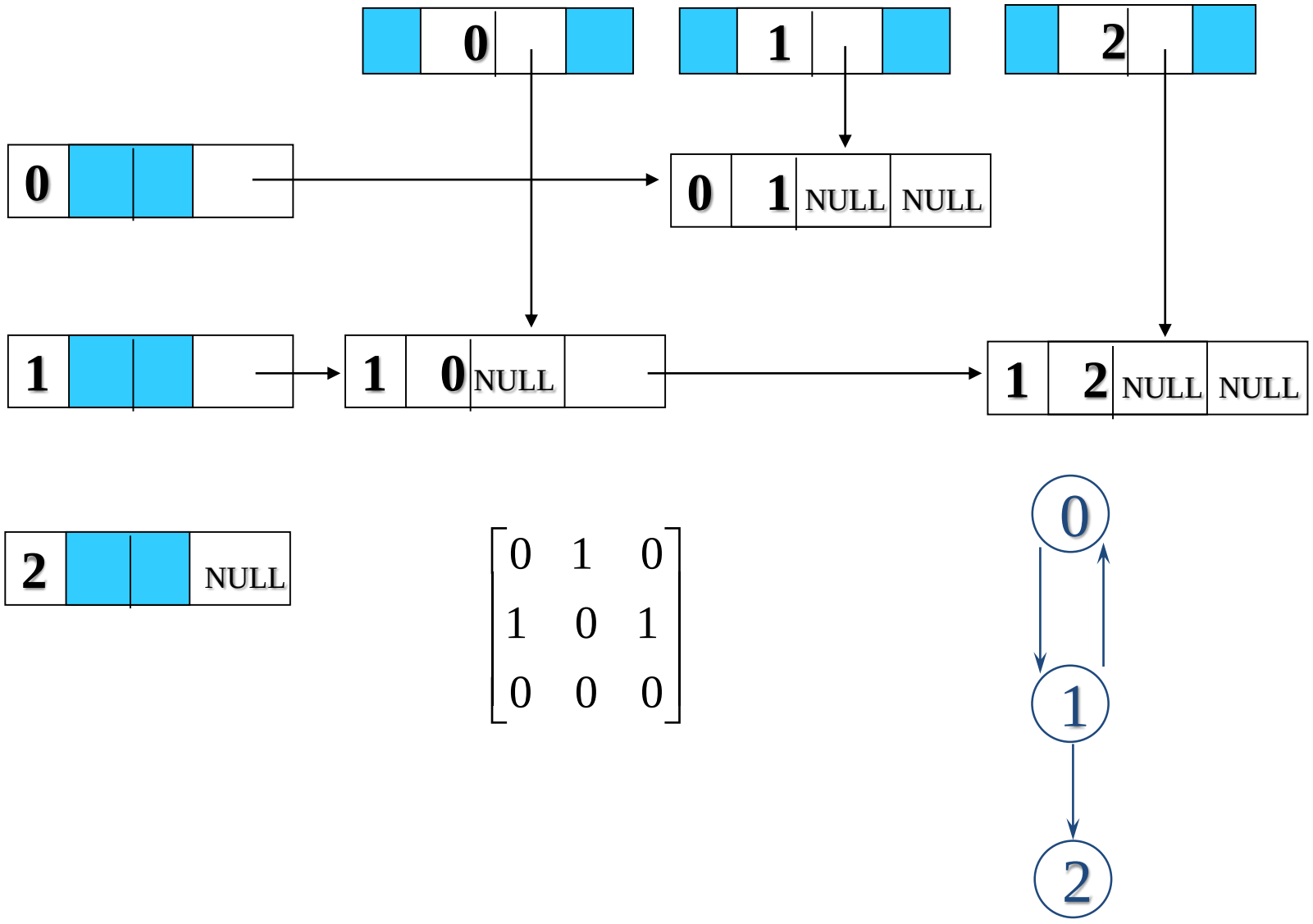
Inverse adjacency list for G_3



Determine in-degree of a vertex in a fast way.

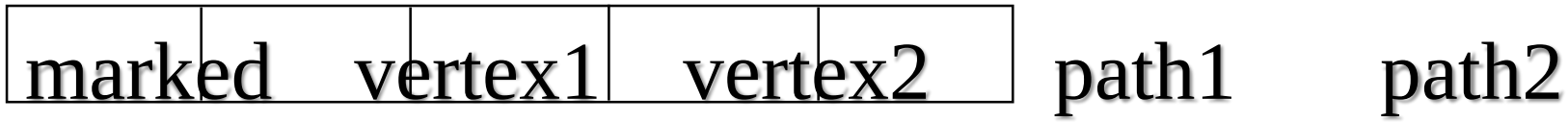
Alternate node structure for adjacency lists

Orthogonal representation for graph G_3



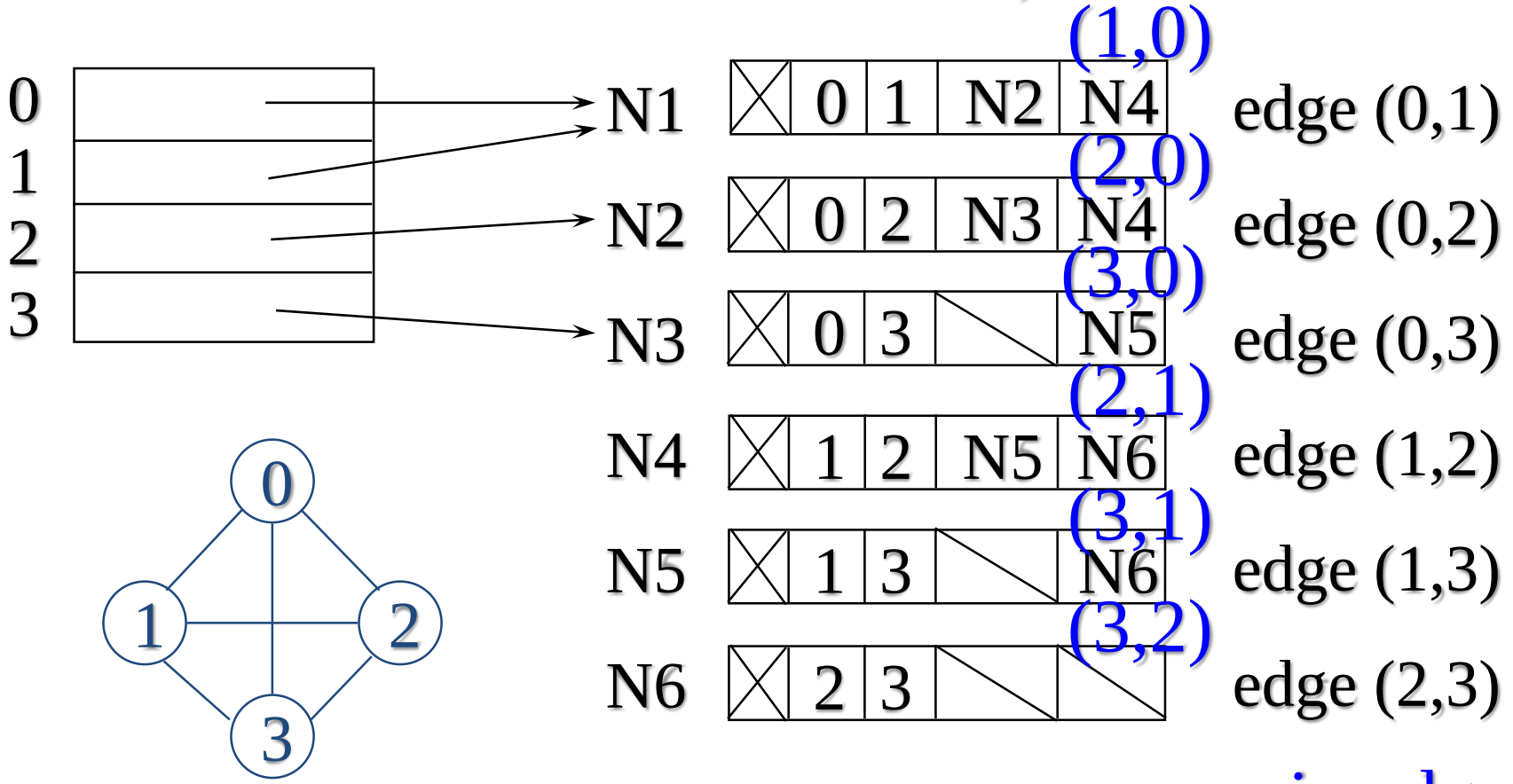
Adjacency Multilists

- An edge in an undirected graph is represented by two nodes in adjacency list representation.
- Adjacency Multilists
  - –lists in which nodes may be shared among several lists.
    (an edge is shared by two different paths)



Example for Adjacency Multlists

Lists: vertex 0: M1->M2->M3, vertex 1: M1->M4->M5
vertex 2: M2->M4->M6, vertex 3: M3->M5->M6



six edges

Adjacency Multilists

```
typedef struct edge *edge_pointer;
typedef struct edge {
    short int marked;
    int vertex1, vertex2;
    edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
```

Minimum cost spanning Tree: Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time
- Select the edges for inclusion in T in nondecreasing order of the cost
- An edge is added to T if it does not form a cycle
- Since G is connected and has n > 0 vertices, exactly n-1 edges will be selected

Examples for Kruskal's Algorithm

0 10 5

2 12 3

1 14 6

1 16 2

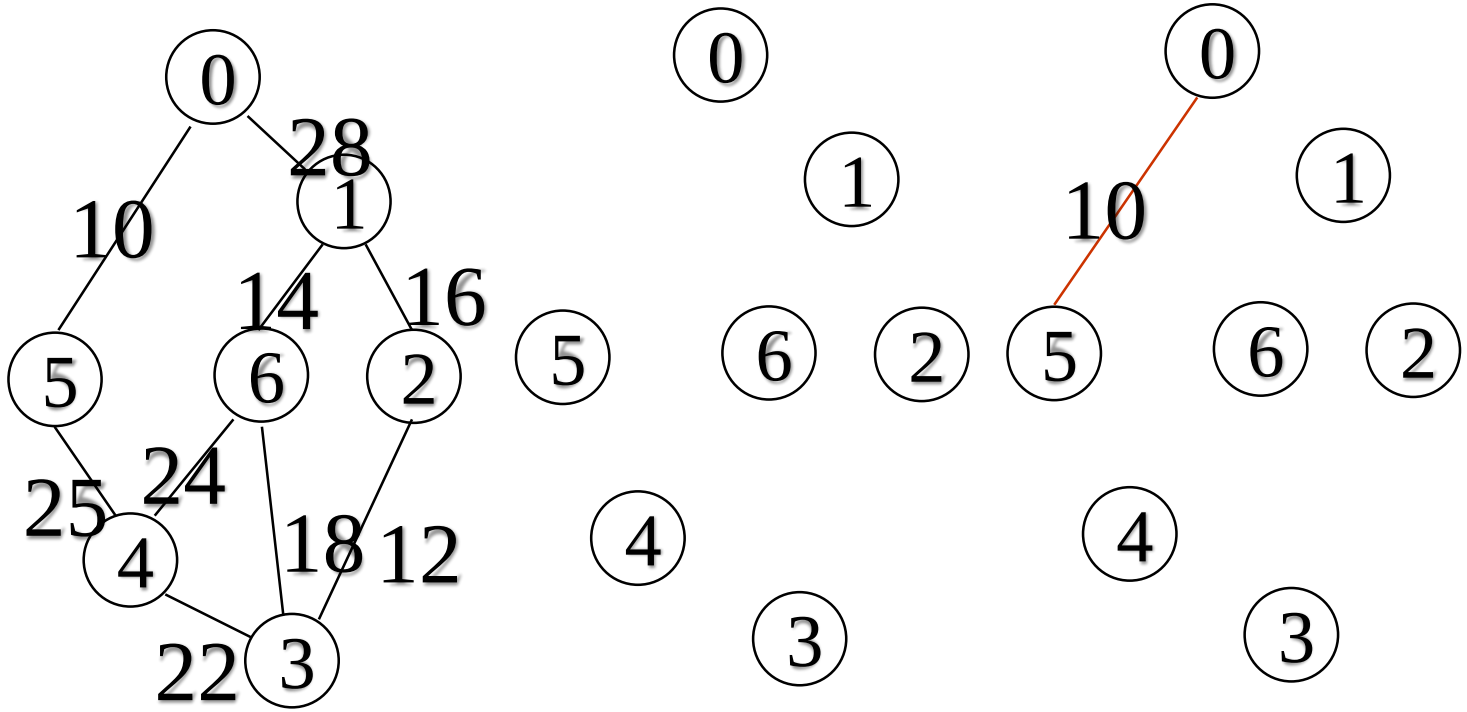
3 18 6

3 22 4

4 24 6

4 25 5

0 28 1



6/9

0 10 5

2 12 3

1 14 6

1 16 2

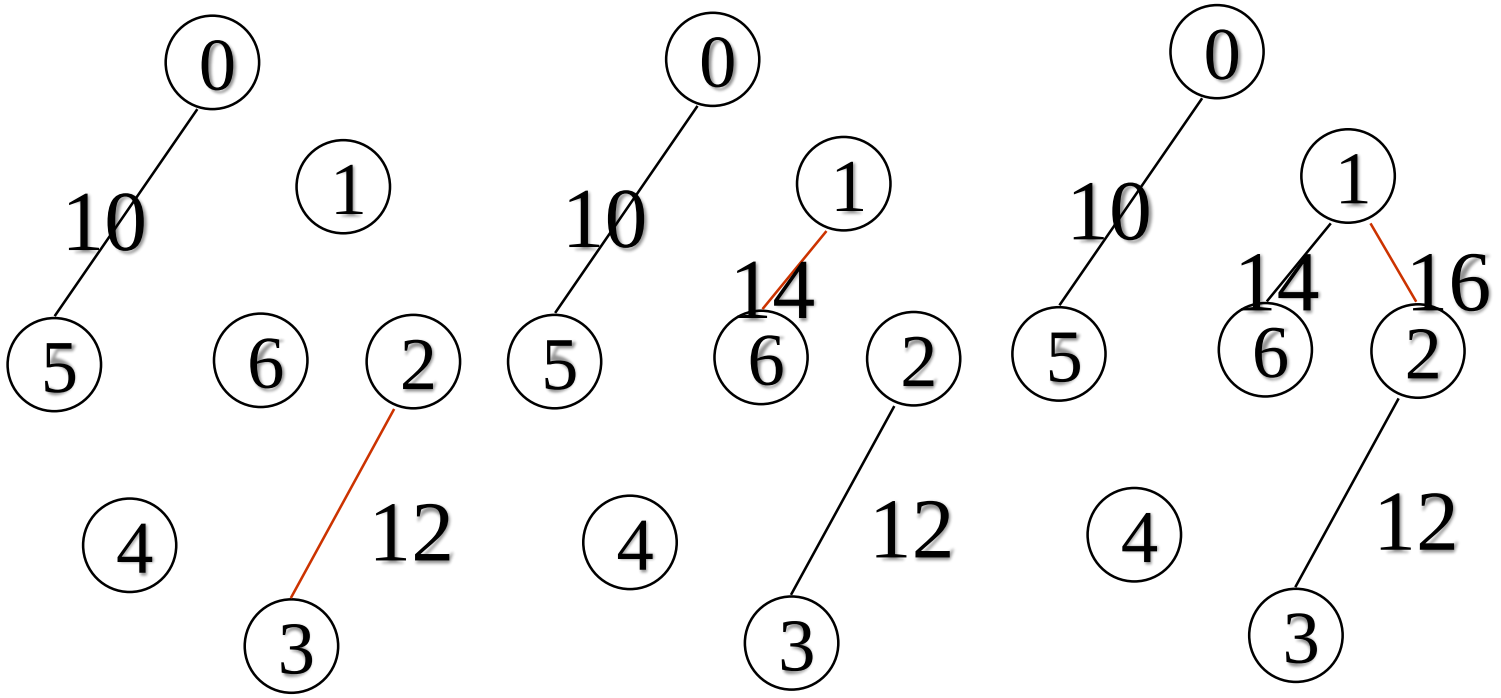
3 18 6

3 22 4

4 24 6

4 25 5

0 28 1



+ 3 6

54

cycle

0 10 5

2 12 3

1 14 6

1 16 2

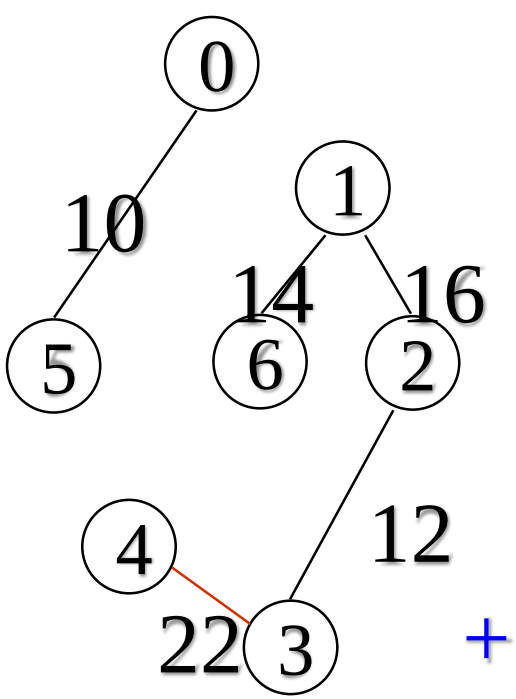
3 18 6

3 22 4

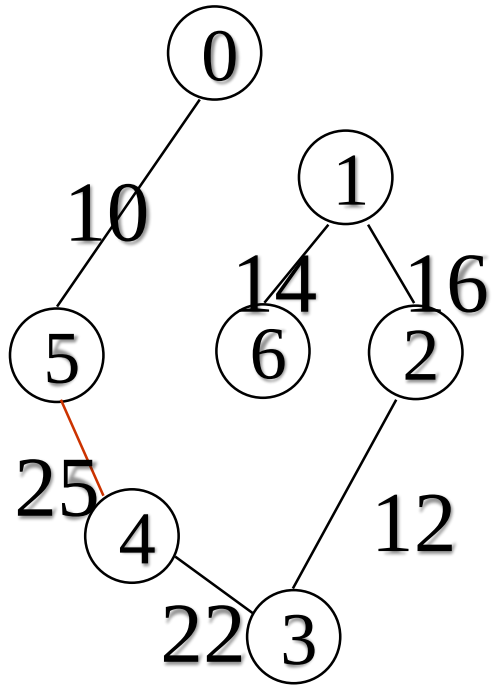
4 24 6

4 25 5

0 28 1



+ 4 — 6



cycle cost = 10 +25+22+12+16+14

Early form of Kruskal's Algorithm

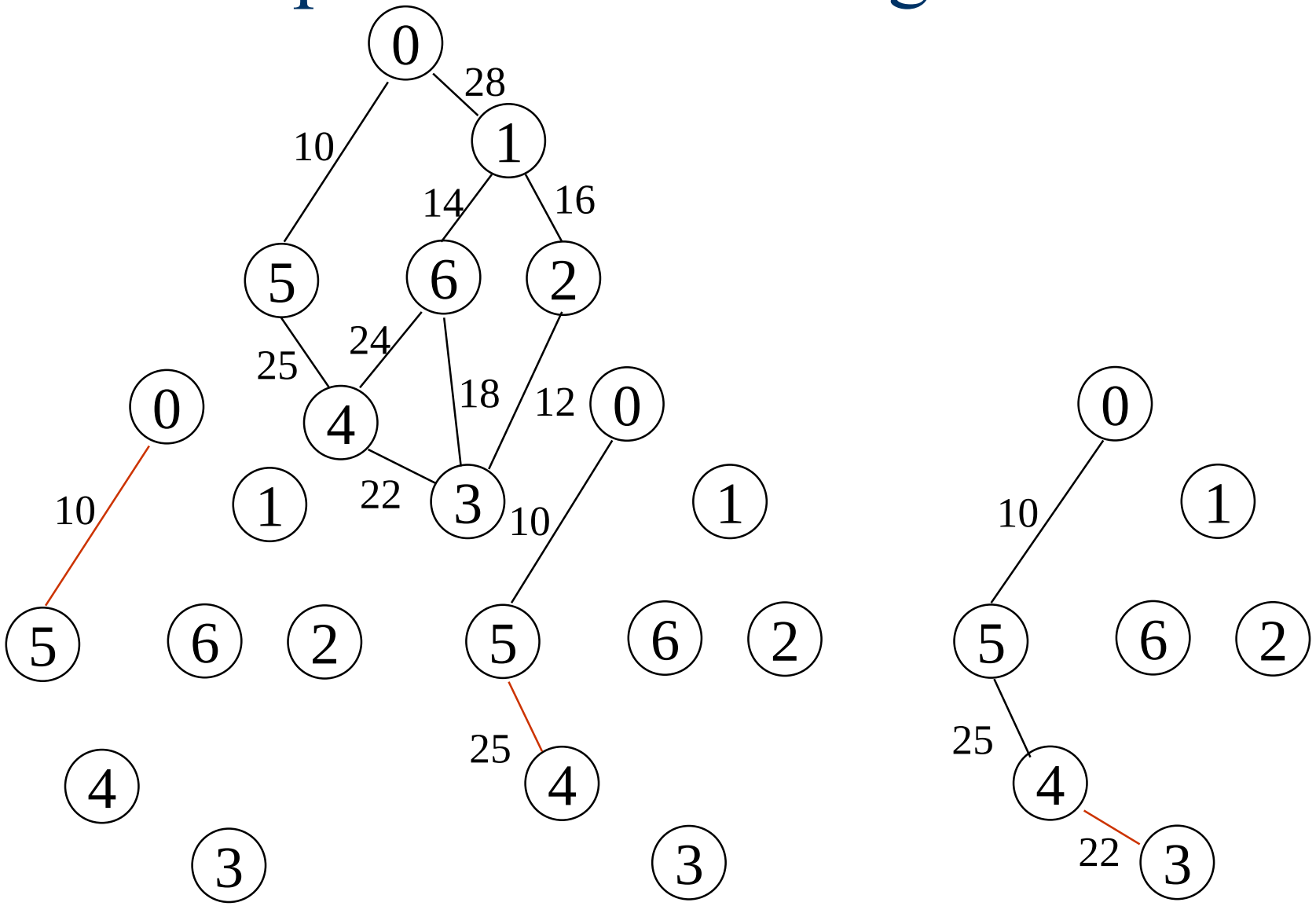
```
1    t := ∅;
2    while ((t has less than n − 1 edges)  and  (E ≠ ∅)) do
3    {
4        Choose an edge (v, w) from E of lowest cost;
5        Delete (v, w) from E;
6        if (v, w) does not create a cycle in t then add (v, w) to t;
7        else discard (v, w);
8    }
```

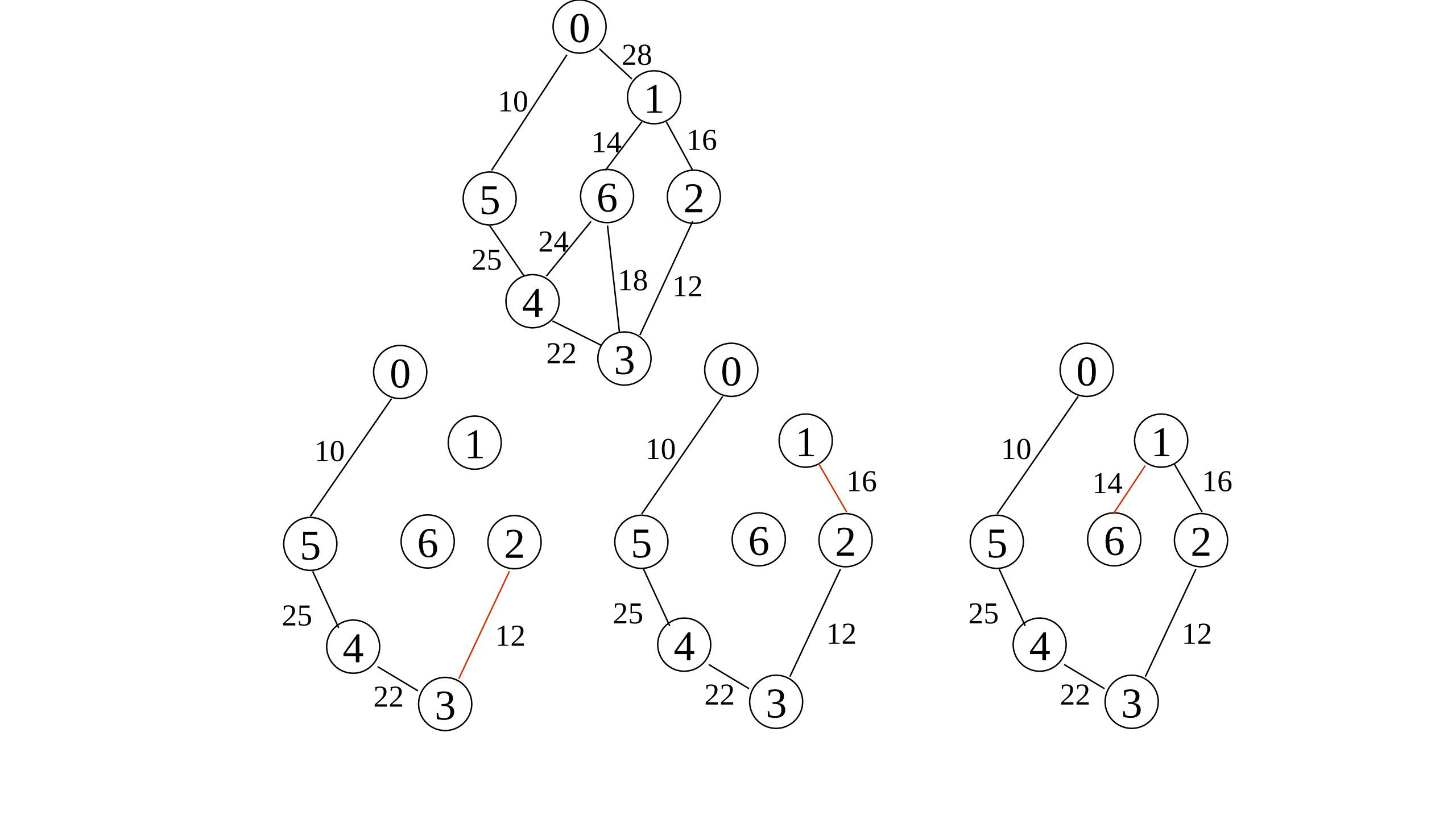
```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

- Prim’s Method for finding the minimum cost spanning tree.

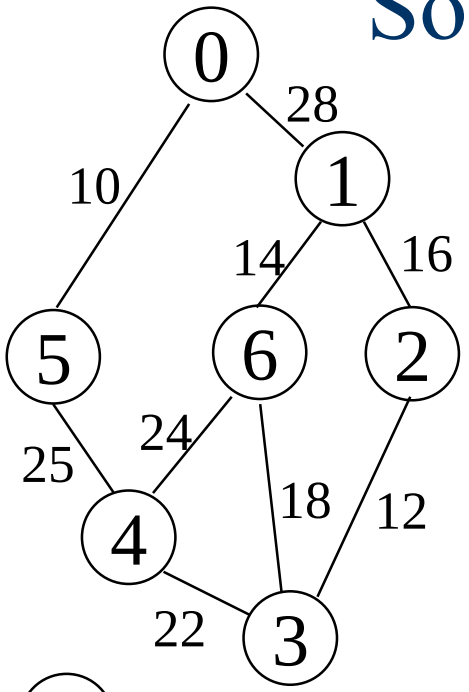
```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9        Let (k, l) be an edge of minimum cost in E;
10       mincost := cost[k, l];
11       t[1, 1] := k; t[1, 2] := l;
12       for i := 1 to n do  // Initialize near.
13            if (cost[i, l] < cost[i, k]) then near[i] := l;
14            else near[i] := k;
15       near[k] := near[l] := 0;
16       for i := 2 to n − 1 do
17       { // Find n − 2 additional edges for t.
18            Let j be an index such that near[j] ≠ 0 and
19            cost[j, near[j]] is minimum;
20            t[i, 1] := j; t[i, 2] := near[j];
21            mincost := mincost + cost[j, near[j]];
22            near[j] := 0;
23            for k := 1 to n do // Update near[ ].
24                 if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                      then near[k] := j;
26       }
27       return mincost;
28  }
```

Examples for Prim's Algorithm




0
28
10
1
14
16
5
6
2
24
25
18
12
4
22
3
0
10
1
5
6
2
25
12
4
22
3
0
10
1
16
5
6
2
25
12
4
22
3
0
10
1
14
16
5
6
2
25
12
4
22
3

Sollin's Algorithm



vertex	edge
0	0 -- 10 --> 5, 0 -- 28 --> 1
1	1 -- 14 --> 6, 1-- 16 --> 2, 1 -- 28 --> 0
2	2 -- 12 --> 3, 2 -- 16 --> 1
3	3 -- 12 --> 2, 3 -- 18 --> 6, 3 -- 22 --> 4
4	4 -- 22 --> 3, 4 -- 24 --> 6, 5 -- 25 --> 5
5	5 -- 10 --> 0, 5 -- 25 --> 4
6	6 -- 14 --> 1, 6 -- 18 --> 3, 6 -- 24 --> 4