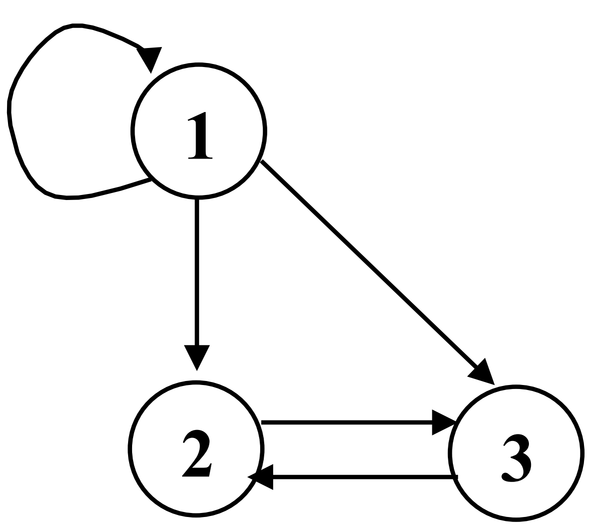
Data Structure:
Graph

graphs

- a graph $G = (V, E)$
  - V: a set of vertices (or nodes)
  - E: a set of edges (or arcs)

    each edge is represented as (v, w) where $v, w \in V$
- directed graph (Digraph): a graph with directed edges
- undirected graph: a graph with undirected edges

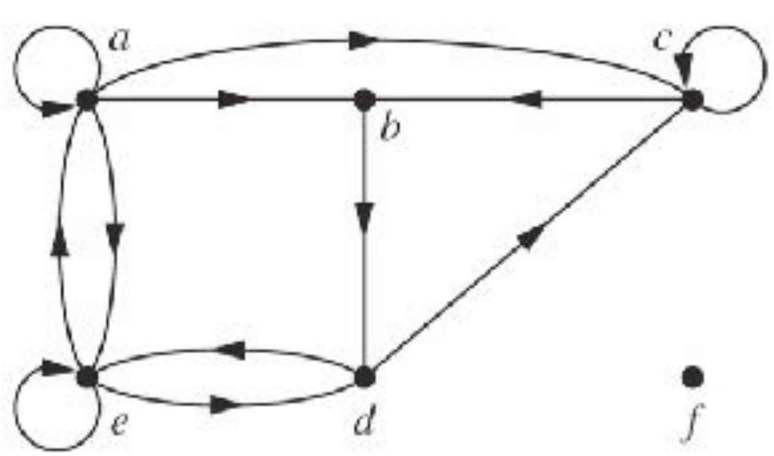
$V = \{1, 2, 3\}$
$E = \{(1,1), (1,2), (2,3), (3,2), (1,3)\}$



directed graphs

- Let G = (V, E) be an directed graph
  - edge (u, v)
    - u is adjacent to v
    - u is initial vertex of (u, v)
    - v is terminal vertex of (u, v)
- degree of edges
  - in-degree of a vertex v
    - the number of edges with v as their terminal vertex
  - out-degree of a vertex v
    - the number of edges with v as their initial vertex

$$\sum_{v \in V} indeg(v) = \sum_{v \in V} outdeg(v) = |E|$$



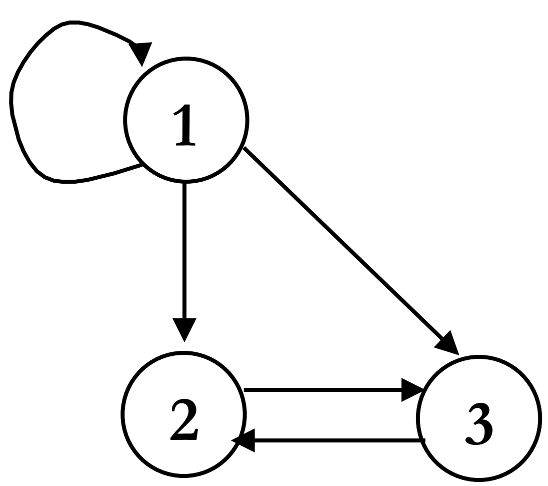
indeg(a) = 2 outdeg(a) = 4
indeg(b) = 2 outdeg(b) = 1
indeg(f) = 0 outdeg(f) = 0

connectivity of graphs

- For a digraph $G = (V, E)$, $n = |V|$, $e = |E|$, $e \leq n^2$
- path is a sequence of vertices $x_1, x_2, \ldots, x_{n-1}$
- the length of path is the number of edges in the path
- cycle begins and ends at the same vertex
- a path or cycle is simple if it does not contain the same edge more than once, the first and the last could be the same
- DAG (Directed Acyclic Graph): a digraph with no cycles.

graph representation: adjacency matrices

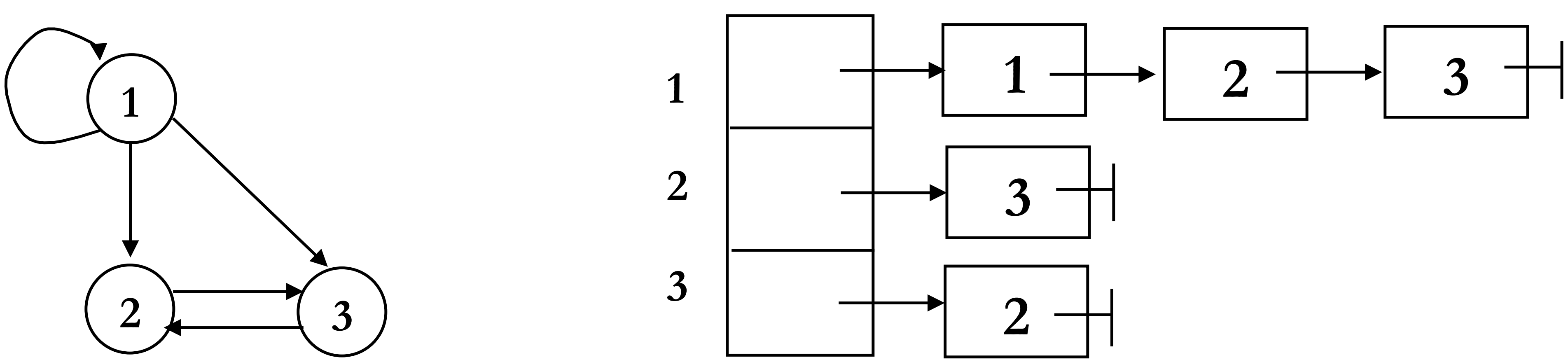
- $A[u][v] = w$ if an edge exists between vertices u and v
  $A[u][v] = 0$ otherwise.
- $w = 1$ or an arbitrary weight associated with edges
- use a table of size $|V|$ to store a mapping from vertex names to array indices
- simple but $\Theta(|V|^2)$ space is needed
- appropriate for dense graphs with $|E|$ approaching to $(|V|^2)$.



	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

graph representation: adjacency lists

- for each vertex, keep a list of adjacent vertices.
- space requirement: $O(\,|E| + |V|\,)$
- appropriate for sparse graphs.



partial orderings

- a relation R on a set S is partial order if it is reflexive, antisymmetric, and transitive
- (S, R): a set S with a partial ordering R is a partially ordered set (poset)

$\geq$ is a partial ordering on the set of integers

reflexive: $a \geq a$ for every integer a

antisymmetric: $a \geq b$ and $b \geq a$ then $a = b$

transitive: $a \geq b$ and $b \geq c$ imply $a \geq c$

total orderings

- if $(S, \preccurlyeq)$ is a poset and every two elements of S are comparable, S is a totally ordered, linearly ordered set, or chain

poset $(Z, \leq)$ is totally ordered because $a \leq b$ or $b \leq a$

poset $(Z^+, |)$ is not totally ordered because $5 \nmid 7$ and $7 \nmid 5$

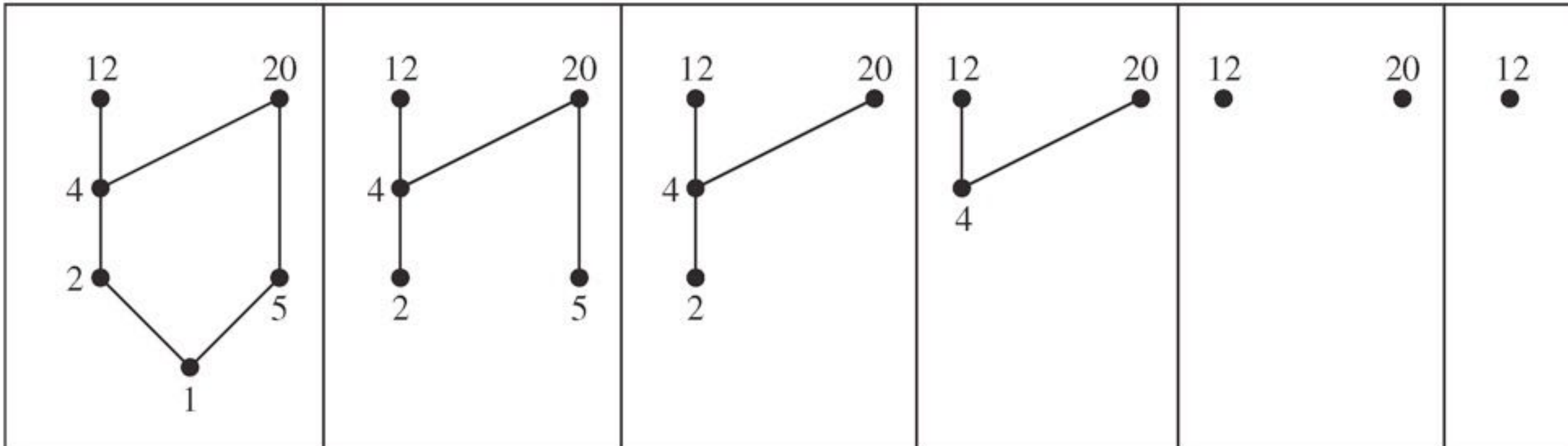
topological sorting

- topological sorting is constructing a compatible total ordering from a partial ordering

```
procedure topological sorting ((S, ≼))

k := 1
while S ≠ ∅
    a_k := a minimal elements of S
    S := S - {a_k}
    k := k + 1
return a_1, a_2, . . . a_n {a_1, a_2, . . ., a_n is a compatible total ordering of S}
```

topological sorting

Find a compatible total ordering for the poset({1, 2, 4, 5, 12, 20}, |)



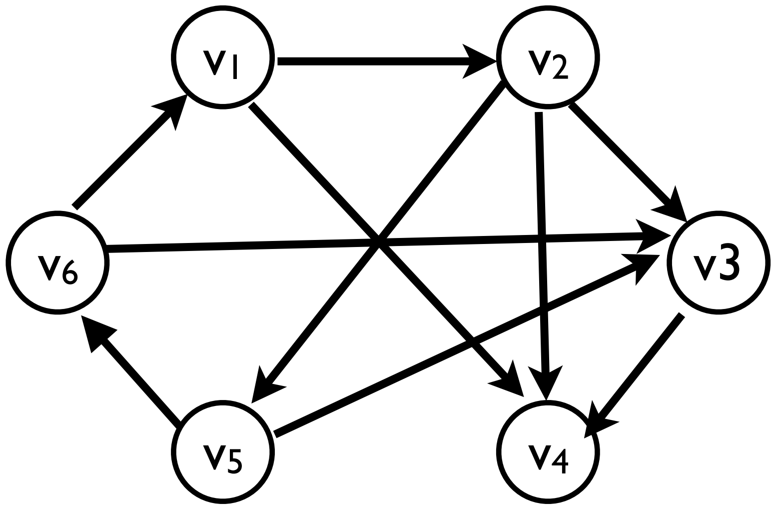
Minimal element chosen	1	5	2	4	20	12

1 < 5 < 2 < 4 < 20 <12

topological sort

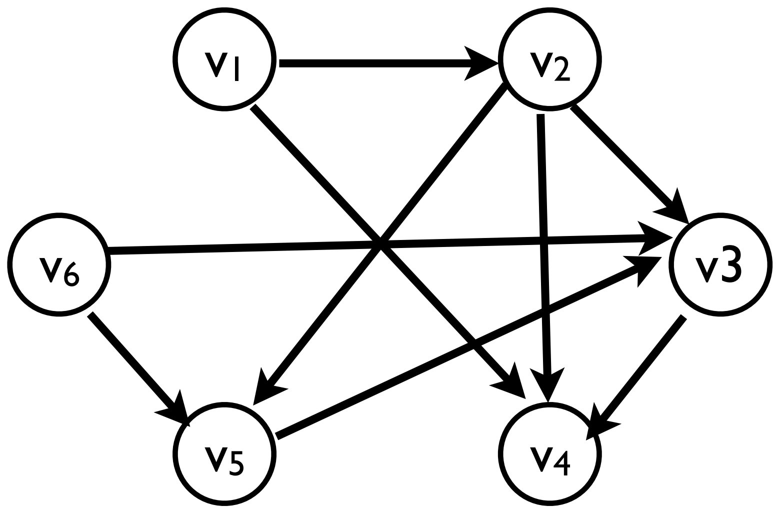
- ordering of vertices in a DAG, such that if there exists a path from v_i to v_j, then v_j appears after v_i
- Example: a topological ordering of courses
  - any course sequence that does not violate the prerequisite requirement

topological sort



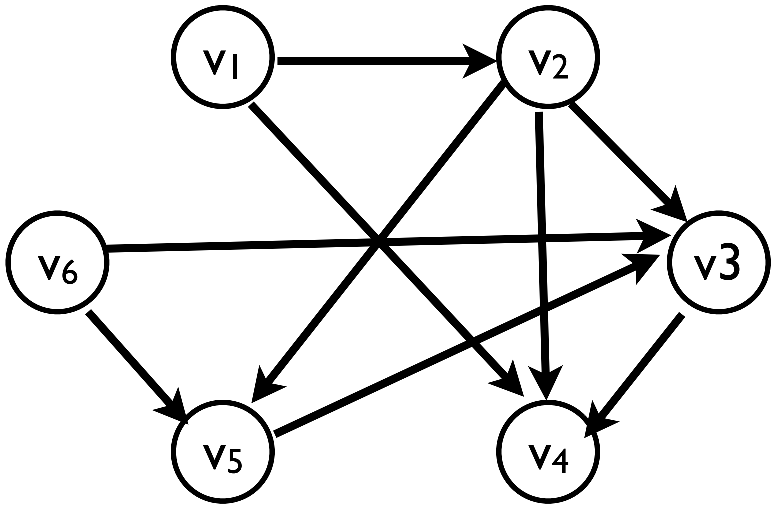
Can you perform topological sort?

topological sort

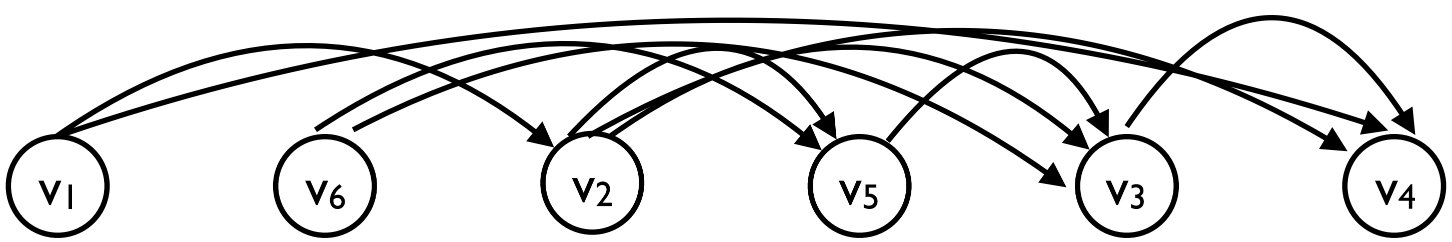


Can you perform topological sort?

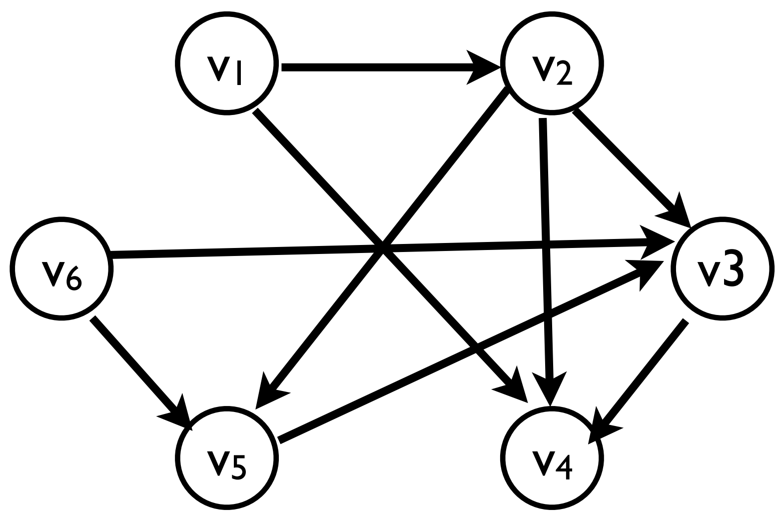
topological sort



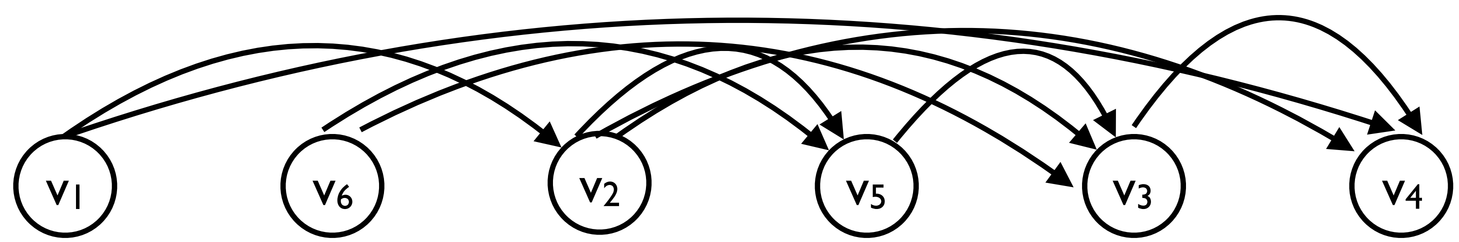
Can you perform topological sort?

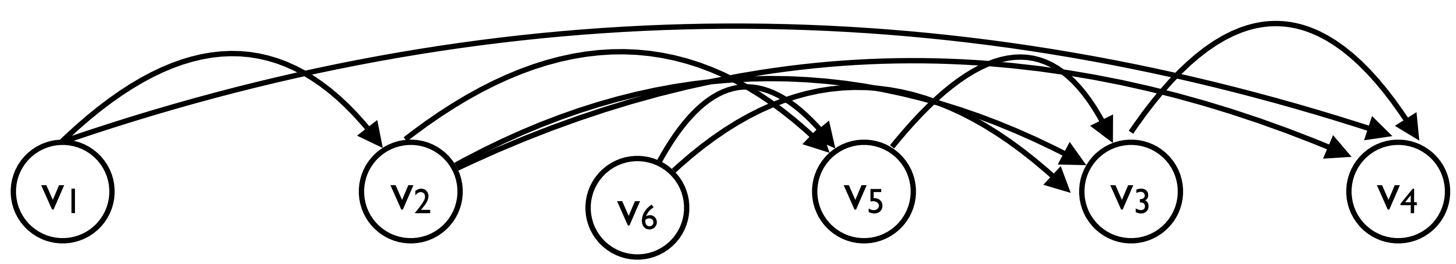


topological sort



Can you perform topological sort?



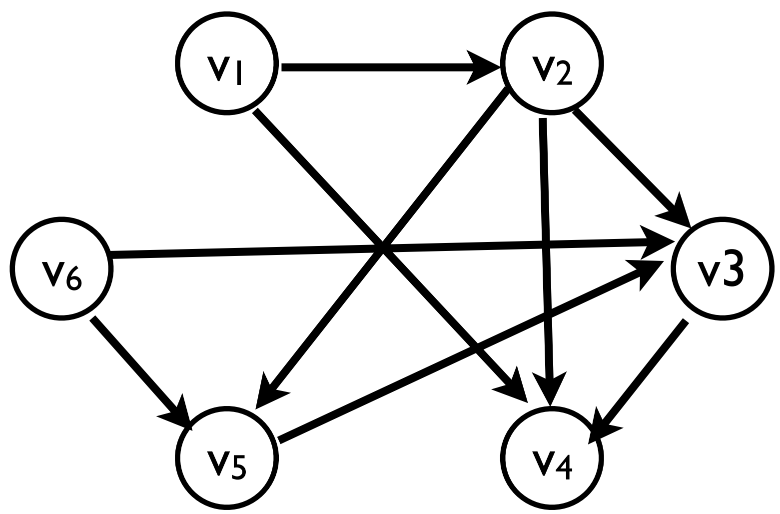


topological sort

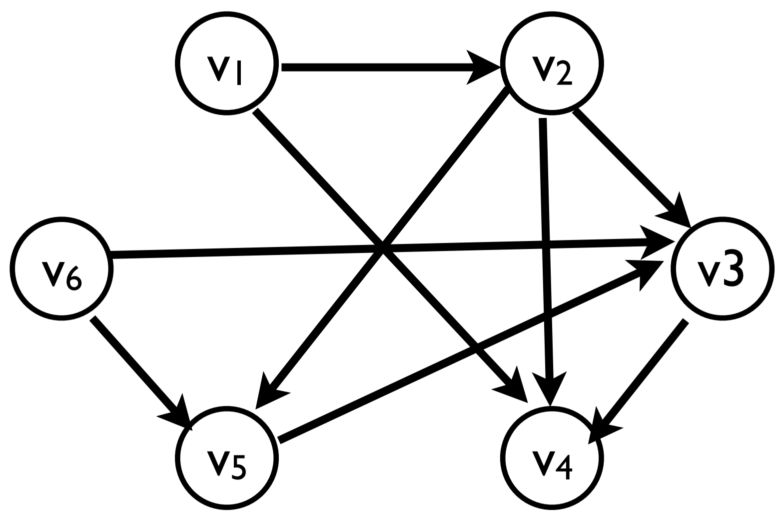
- algorithm:
  - for each vertex *v* whose in-degree is zero,
    - print *v*
    - remove *v* and its outgoing edges (which leads to decrementing the in-degree value of *v*'s adjacent vertices)

- use either stack or queue to keep track of vertices with in-degree = 0
- use adjacency list representation or matrix

topological sort

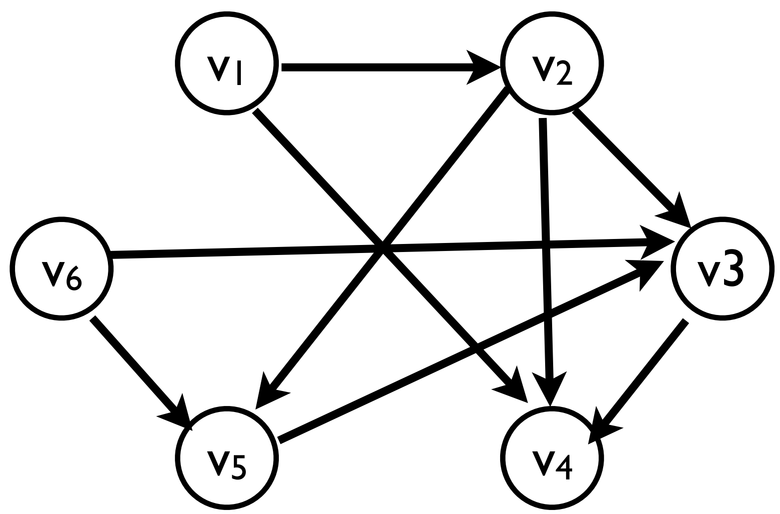


topological sort



	v1	v2	v3	v4	v5	v6
v1	0	1	0	1	0	0
v2	0	0	1	1	1	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	1	0	1	0

topological sort

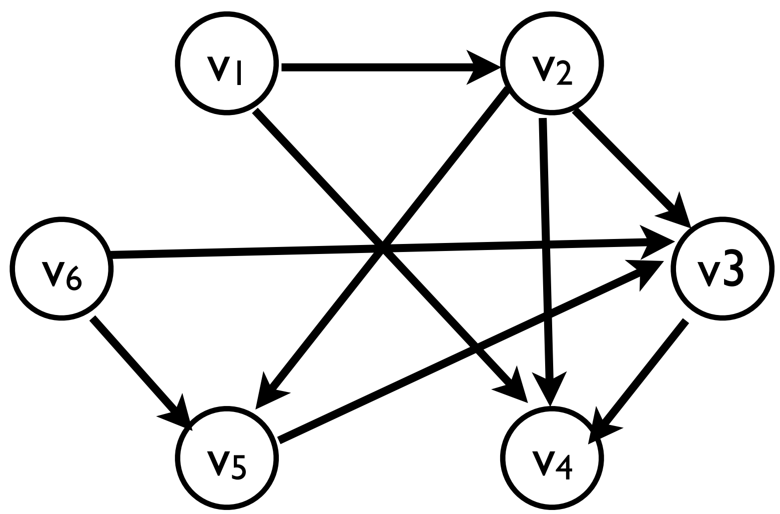


# of in-dgree

v1	0					
v2	1					
v3	3					
v4	3					
v5	2					
v6	0					
queue						
dequeue						

	v1	v2	v3	v4	v5	v6
v1	0	1	0	1	0	0
v2	0	0	1	1	1	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	1	0	1	0

topological sort

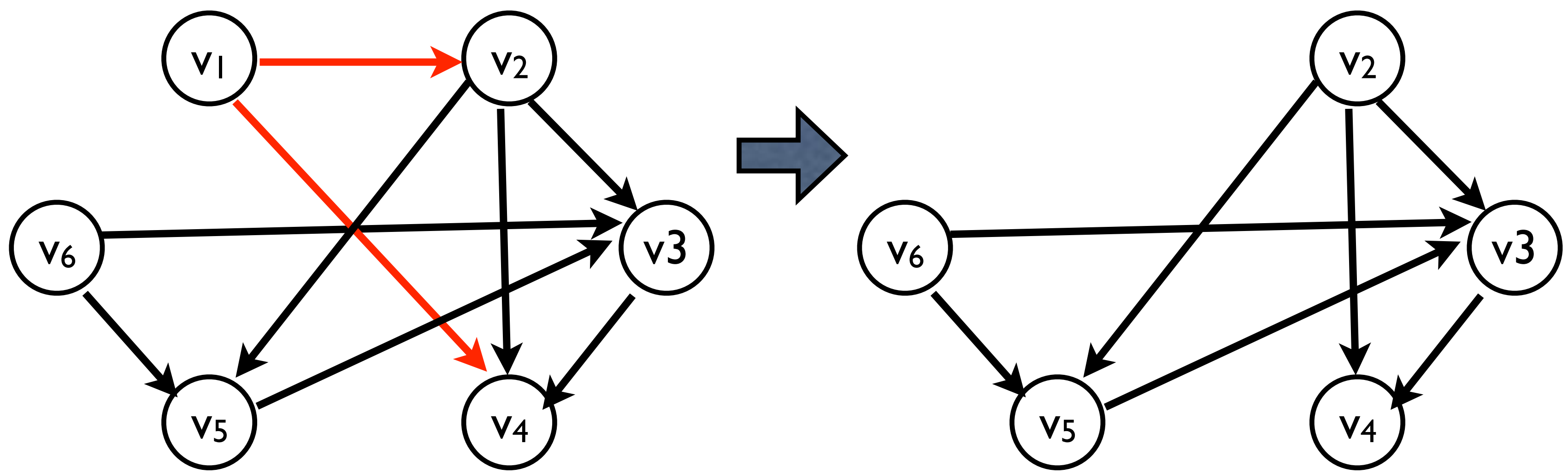


# of in-dgree

v1	0					
v2	1					
v3	3					
v4	3					
v5	2					
v6	0					
queue	**v1, v6**					
dequeue						

	v1	v2	v3	v4	v5	v6
v1	0	1	0	1	0	0
v2	0	0	1	1	1	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	1	0	1	0

topological sort



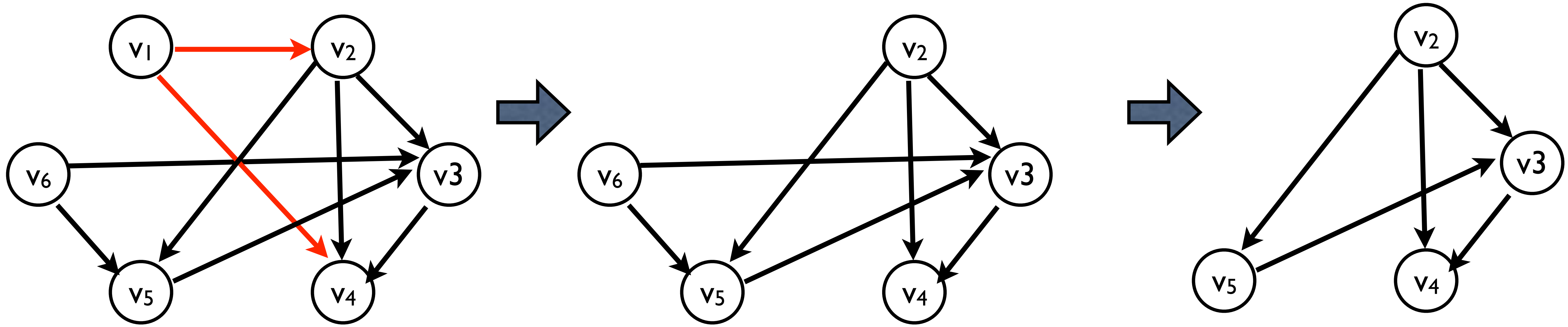
# of in-dgree

v1	0	0				
v2	1	0				
v3	3	3				
v4	3	2				
v5	2	2				
v6	0	0				
queue	v1, v6					
dequeue	**v1**					

queue v6

	v1	v2	v3	v4	v5	v6
v1	0	0	0	0	0	0
v2	0	0	1	1	1	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	1	0	1	0

topological sort



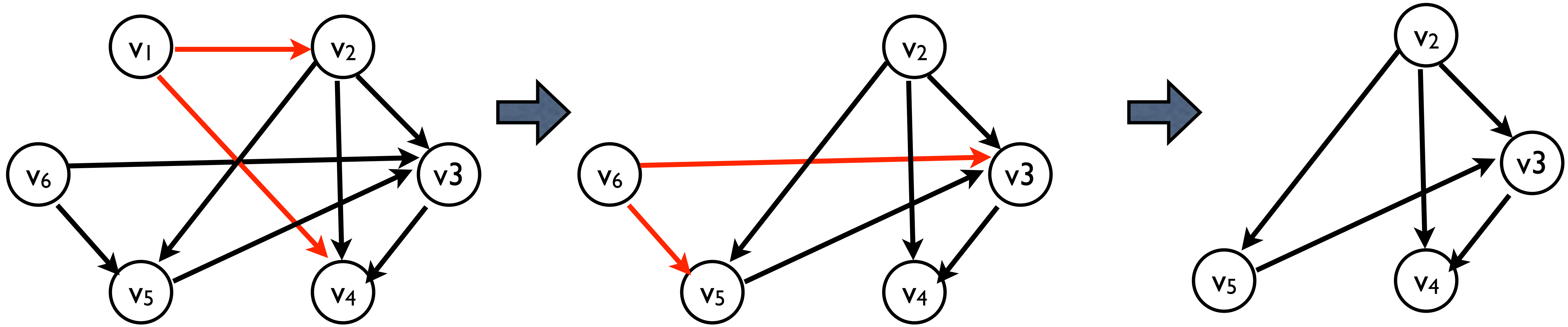
# of in-dgree

v1	0	0				
v2	1	0				
v3	3	3				
v4	3	2				
v5	2	2				
v6	0	0				
queue	v1, v6	v6, **v2**				
dequeue	**v1**					

queue v6

	v1	v2	v3	v4	v5	v6
v1	0	0	0	0	0	0
v2	0	0	1	1	1	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	1	0	1	0

topological sort



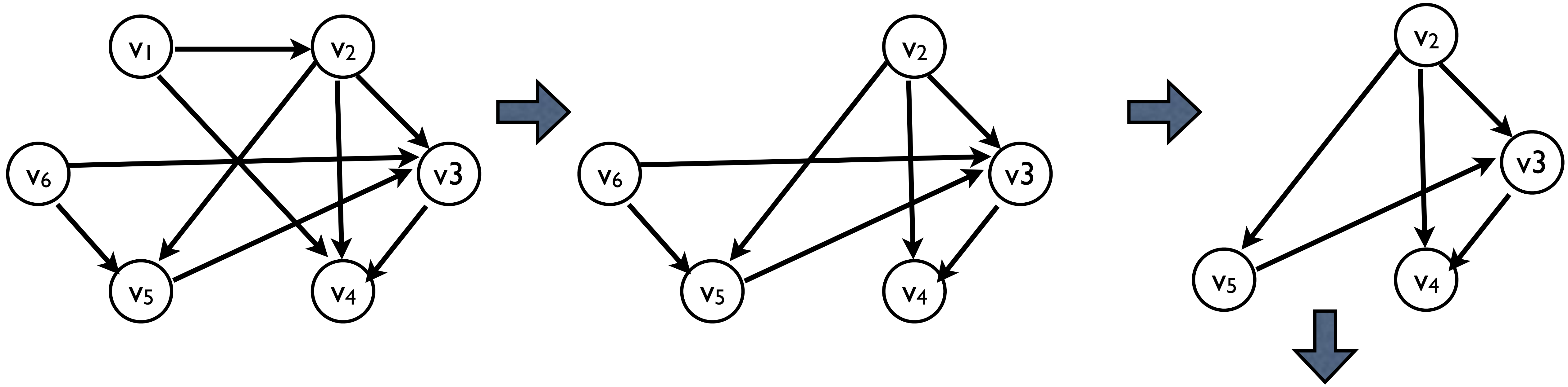
\# of in-dgree

v1	0	0				
v2	1	0				
v3	3	3				
v4	3	2				
v5	2	2				
v6	0	0				
queue	v1, v6	v6, **v2**				
dequeue	**v1**	v6				

queue v6 v2

	v1	v2	v3	v4	v5	v6
v1	0	0	0	0	0	0
v2	0	0	1	1	1	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	1	0	1	0

topological sort

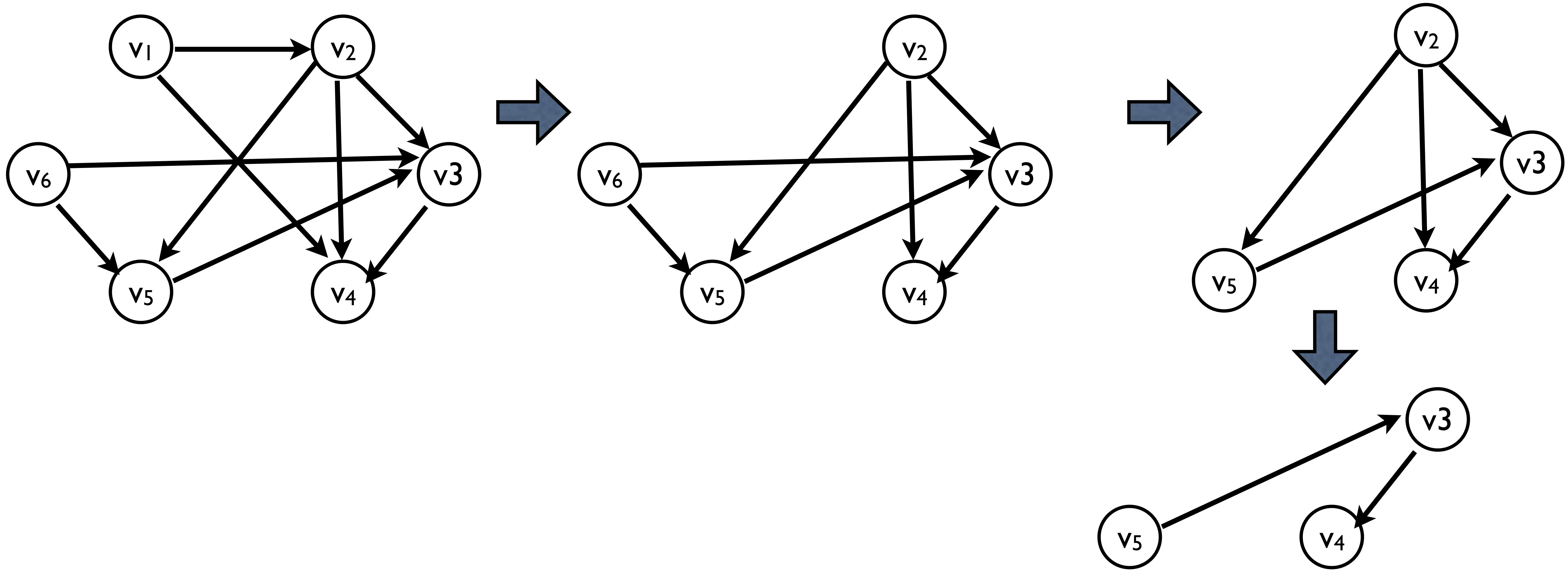


# of in-dgree

v1	0	0	0			
v2	1	0	0			
v3	3	3	2			
v4	3	2	2			
v5	2	2	1			
v6	0	0	0			
queue	v1, v6	v6, v2	v2			
dequeue	v1	v6	v2			
queue	v6	v2	none			

	v1	v2	v3	v4	v5	v6
v1	0	0	0	0	0	0
v2	0	0	1	1	1	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	0	0	0	0

topological sort

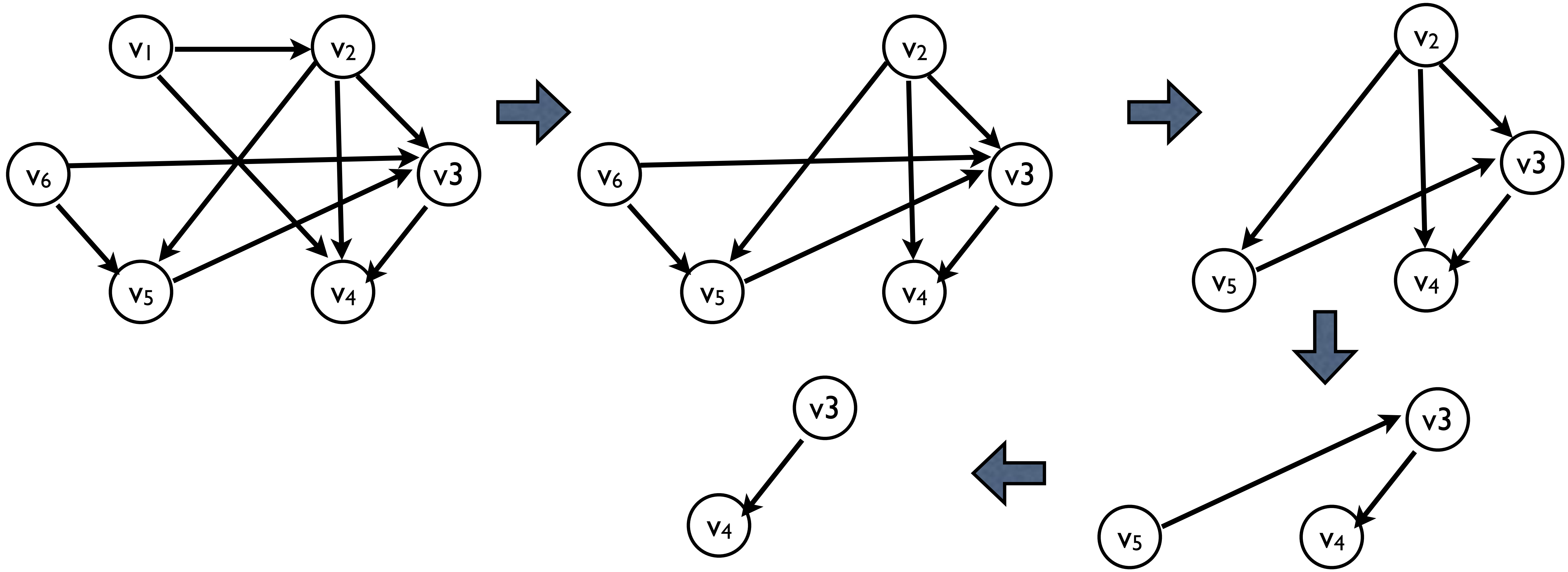


# of in-dgree

v1	0	0	0	0		
v2	1	0	0	0		
v3	3	3	2	1		
v4	3	2	2	1		
v5	2	2	1	0		
v6	0	0	0	0		
queue	v1, v6	v6, v2	v2	v5		
dequeue	v1	v6	v2	v5		
queue	v6	v2	none	none		

	v1	v2	v3	v4	v5	v6
v1	0	0	0	0	0	0
v2	0	0	0	0	0	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	1	0	0	0
v6	0	0	0	0	0	0

topological sort



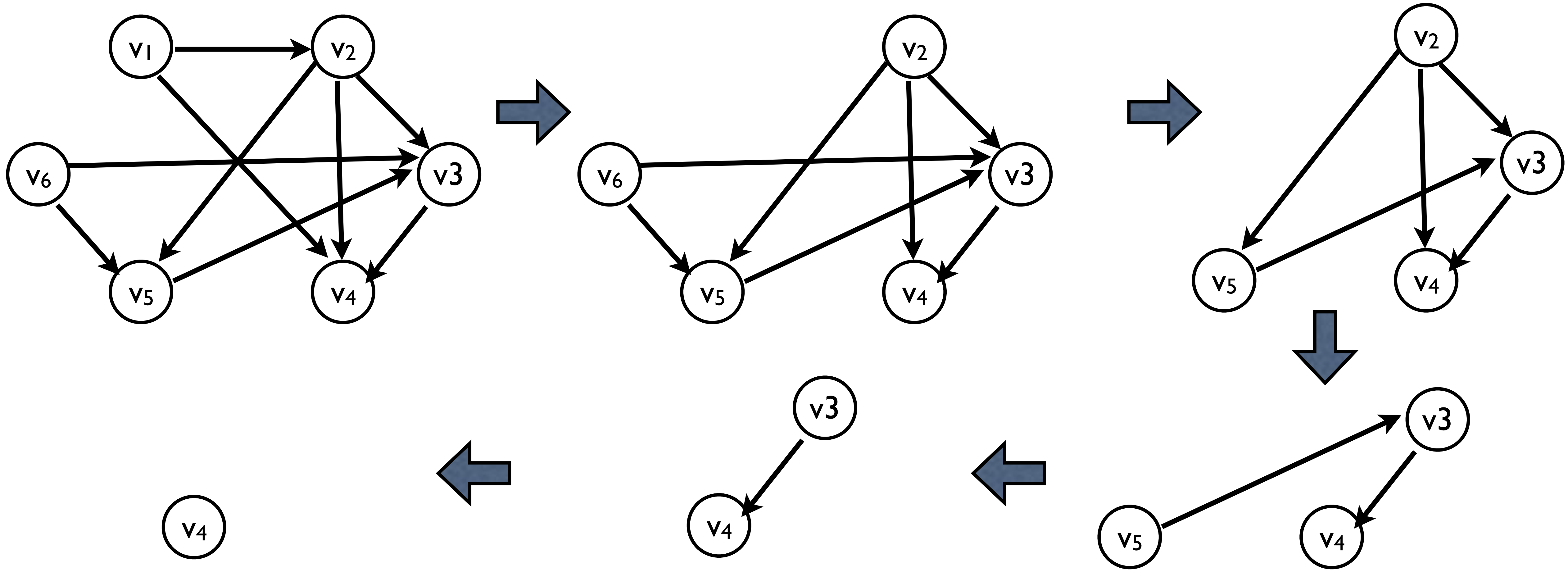
# of in-dgree

v1	0	0	0	0	0	
v2	1	0	0	0	0	
v3	3	3	2	1	0	
v4	3	2	2	1	1	
v5	2	2	1	0	0	
v6	0	0	0	0	0	
queue	v1, v6	v6, v2	v2	v5	v3	
dequeue	v1	v6	v2	v5	v3	

queue	v6	v2	none	none	none

	v1	v2	v3	v4	v5	v6
v1	0	0	0	0	0	0
v2	0	0	0	0	0	0
v3	0	0	0	1	0	0
v4	0	0	0	0	0	0
v5	0	0	0	0	0	0
v6	0	0	0	0	0	0

topological sort



# of in-dgree

v1	0	0	0	0	0	0
v2	1	0	0	0	0	0
v3	3	3	2	1	0	0
v4	3	2	2	1	1	0
v5	2	2	1	0	0	0
v6	0	0	0	0	0	0
queue	v1, v6	v6, v2	v2	v5	v3	v4
dequeue	v1	v6	v2	v5	v3	v4
queue	v6	v2	none	none	none	none

	v1	v2	v3	v4	v5	v6
v1	0	0	0	0	0	0
v2	0	0	0	0	0	0
v3	0	0	0	0	0	0
v4	0	0	0	0	0	0
v5	0	0	0	0	0	0
v6	0	0	0	0	0	0

topological sort

```
void Topsort (Graph G)
{
    Queue Q;
    Vertex V, W;
    int *Indegree;

    Q = CreateQueue();
    checkIndegree(Indegree);

    for each vertex V
        if( Indegree[V] == 0 )
            Enqueue(V, Q);

    while( !IsEmtpy(Q) )
    {
        V = Dequeue(Q);
        for each W adjacent to V
            if ( --Indegree[W] == 0 )
                Enqueue(W, Q);
    }

}
```

topological sort

- n = |V|, e = |E|
- the number of iterations of the while loop is at most n
- the number of iterations of the for loop is proportional to outdeg(v) and each iteration takes constant time
- since outdeg(v) may be zero and we need to spend some time updating loop variables, etc. in this case, the time is Θ(outdeg(v) + 1)
- the running time is

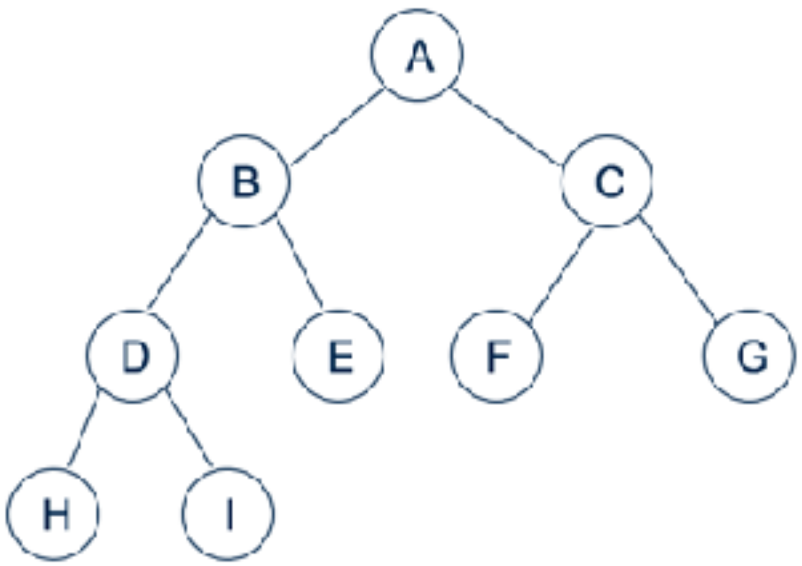
$$T(n) = n + \sum_{v \in V} (outdeg(v) + 1)$$

$$= n + \sum_{v \in V} outdeg(v) + \sum_{v \in V} 1 = n + e + n \in \Theta(n + e)$$

tree traversal

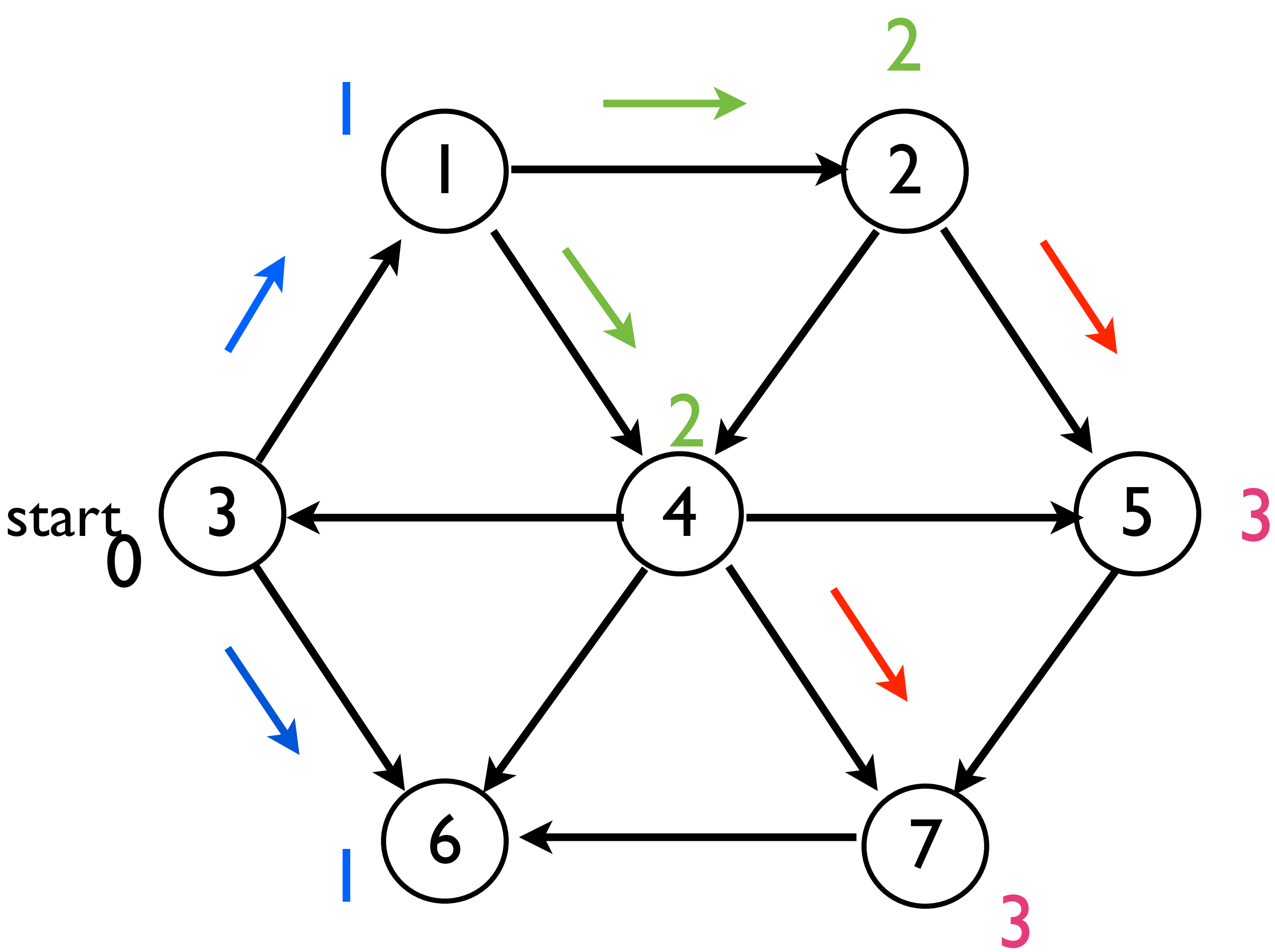
- level-order traversal

```
void levelOrder (Tree ptr)   {
        int front = rear = 0;
        Tree queue[MAX];
        if (! node)     return;
        addq(ptr);
        for (; ; )  {
                ptr = deleteq();
                if (ptr)  {
                        printf("%d", ptr->data);
                        if (ptr -> leftChild)
                                addq(ptr -> leftChild);
                        if (ptr -> rightChild)
                                addq(ptr -> rightChild);
                }
                else break;
        }
}
```



breadth-first search

- vertices closest to the start are evaluated first, and the most distant vertices are evaluated last
- use Queue to keep track of vertices to evaluate



1	6						

6	2	4					

2	4						

4	5						

5	7						

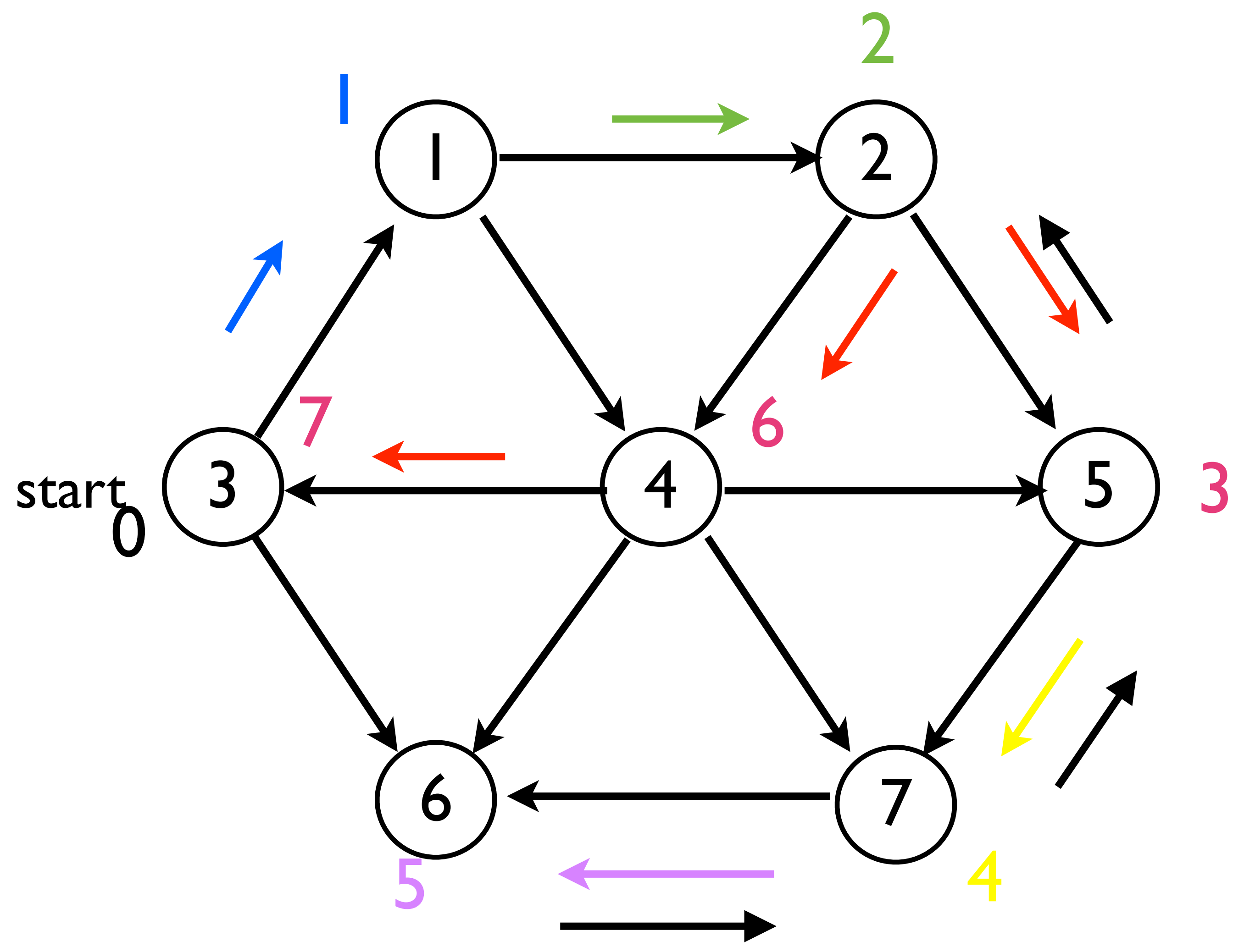
7							

breadth-first search

```
BFS(Table T)
{
    Q = CreateQueue(NumVertex);
    MakeEmpty(Q);
    Enqueue(s, Q);
    while( !IsEmtpy(Q) ) do
    {
        v = Dequeue(Q);
        for each w adjacent to v
            if( d[w] == infinity ) {        /* d[w] : depth,    d[w] == infinity means “not visited yet” */
                d[w] = d[v] + 1;
                pred[w] = v;
                Enqueue(w, Q);
            }
    }
    DisposeQueue(Q);
}
```

depth-first search

- travel as deep as possible from neighbour to neighbour before backtracking
- use Stack to keep track of vertices to evaluate



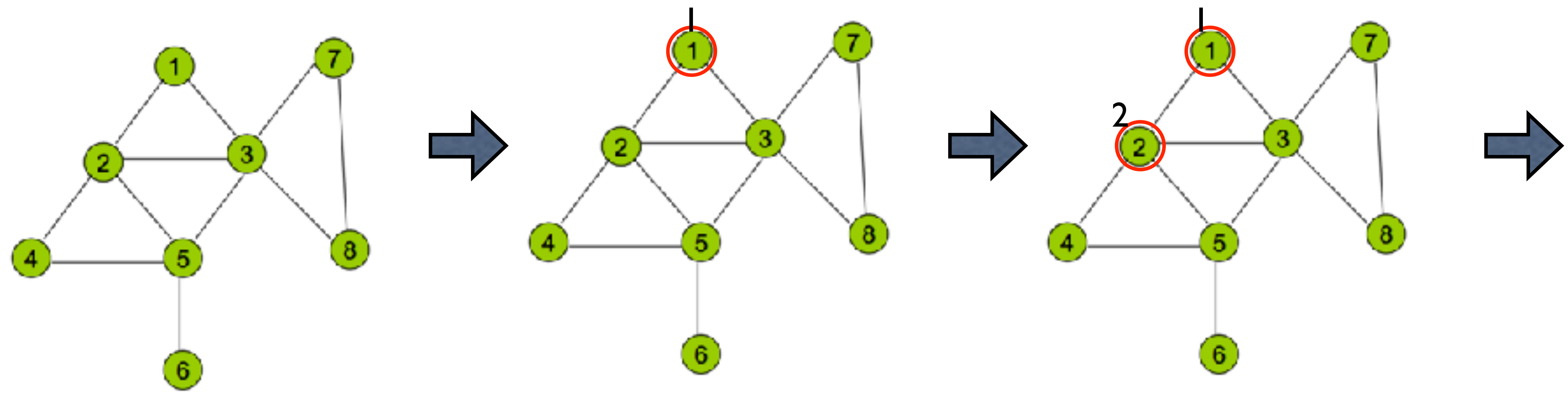
depth-first search: recursive implementation

```
void DFS (G, u){

    while u has an unvisited neighbour in G
        v = an unvisited neighbour of u
        mark v visited
        DFS(G, v)
}
```

depth-first search: iterative implementation using stack

```
void DFS (G, u){

    S = stack initialized
    S.push (u)
    while S is not empty
        v = S.pop()
        if v not visited
            mark  v as visited
            for  w is a neighbour of v
                S. push(w)
```

depth-first search: iterative implementation using stack



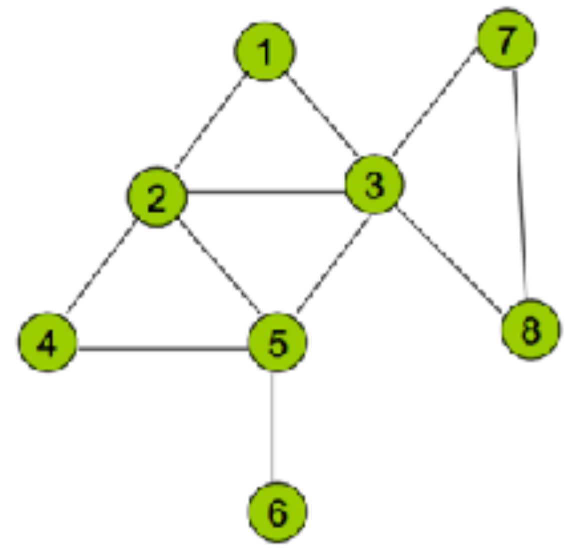
1							

3	2						

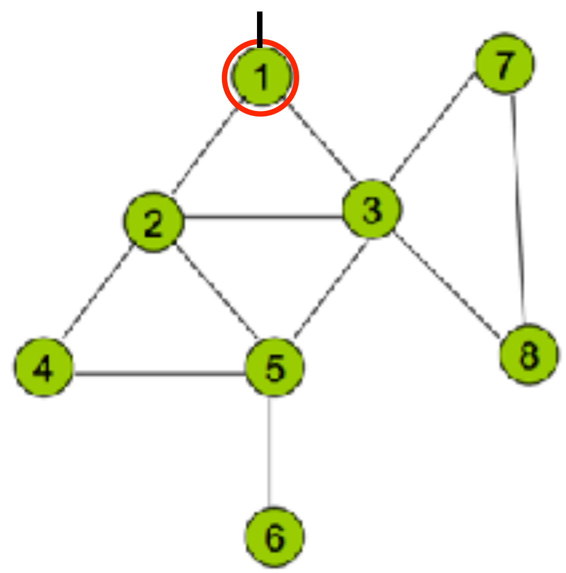
3	4	3	5	1			

```
void DFS (G, u){

    S = stack initialized
    S.push (u)
    while S is not empty
        v = S.pop()
        if v not visited
            mark  v as visited
            for  w is a neighbour of v
                S. push(w)
```

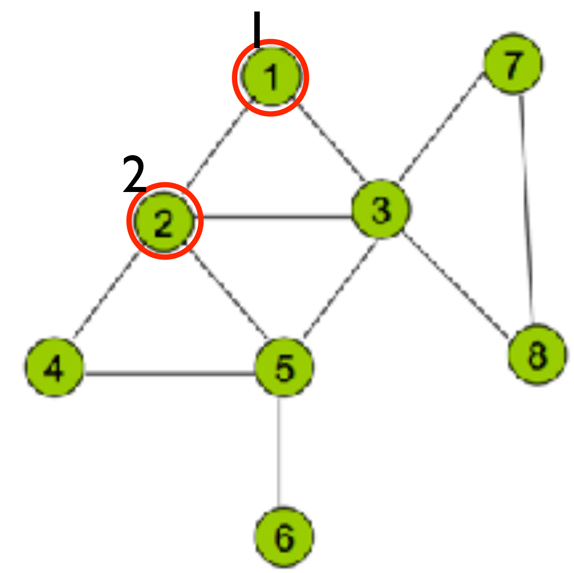
depth-first search: iterative implementation using stack



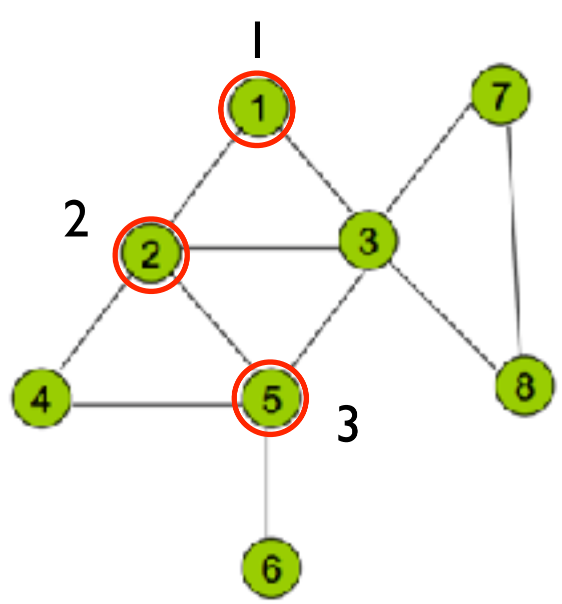
1							



3	2						



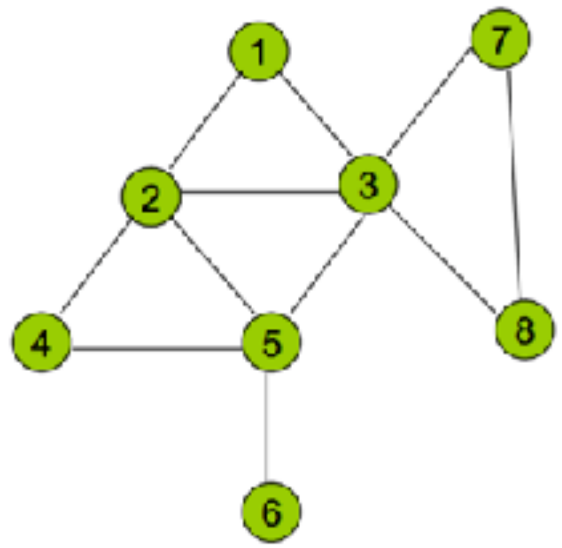
3	4	3	5	1			



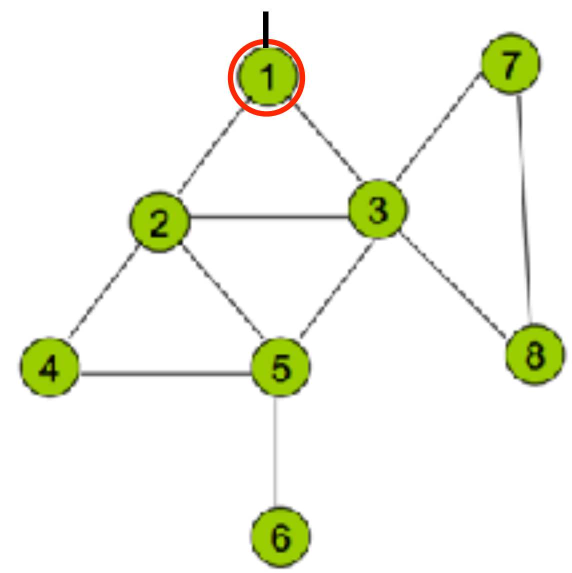
3	4	3	4	6	3	2	

```
void DFS (G, u){

    S = stack initialized
    S.push (u)
    while S is not empty
        v = S.pop()
        if v not visited
            mark  v as visited
            for  w is a neighbour of v
                S. push(w)
```

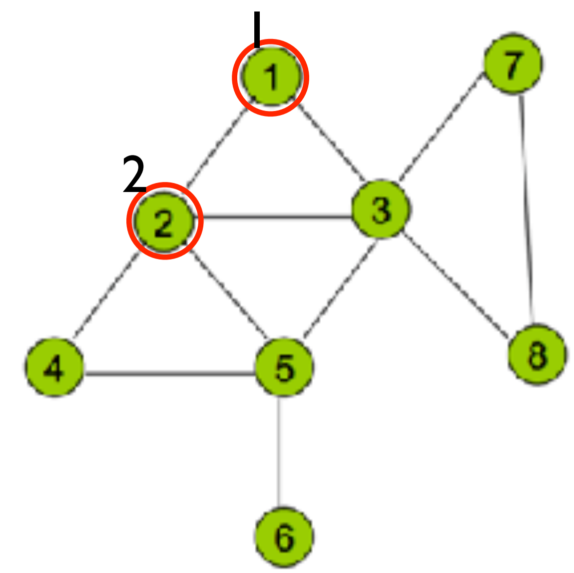
depth-first search: iterative implementation using stack



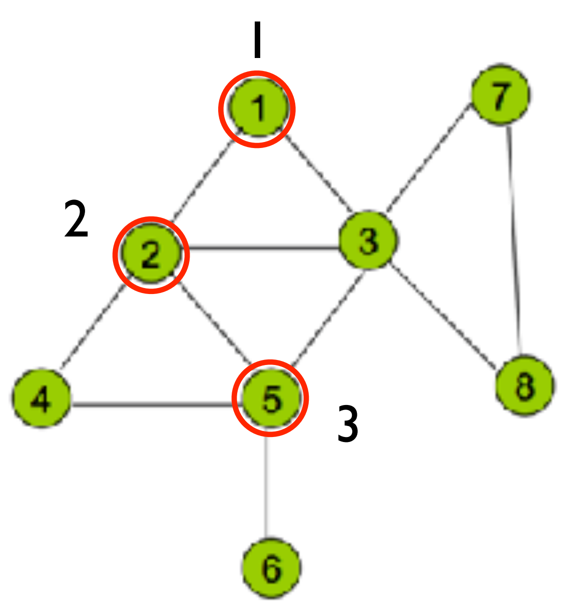
1							



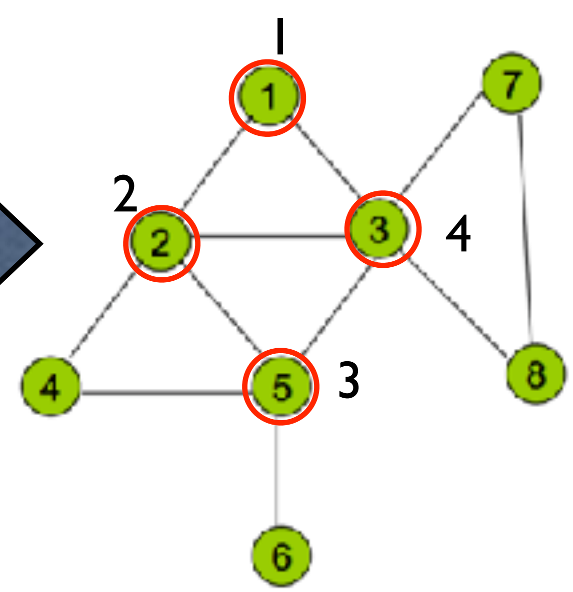
3	2						



3	4	3	5	1			



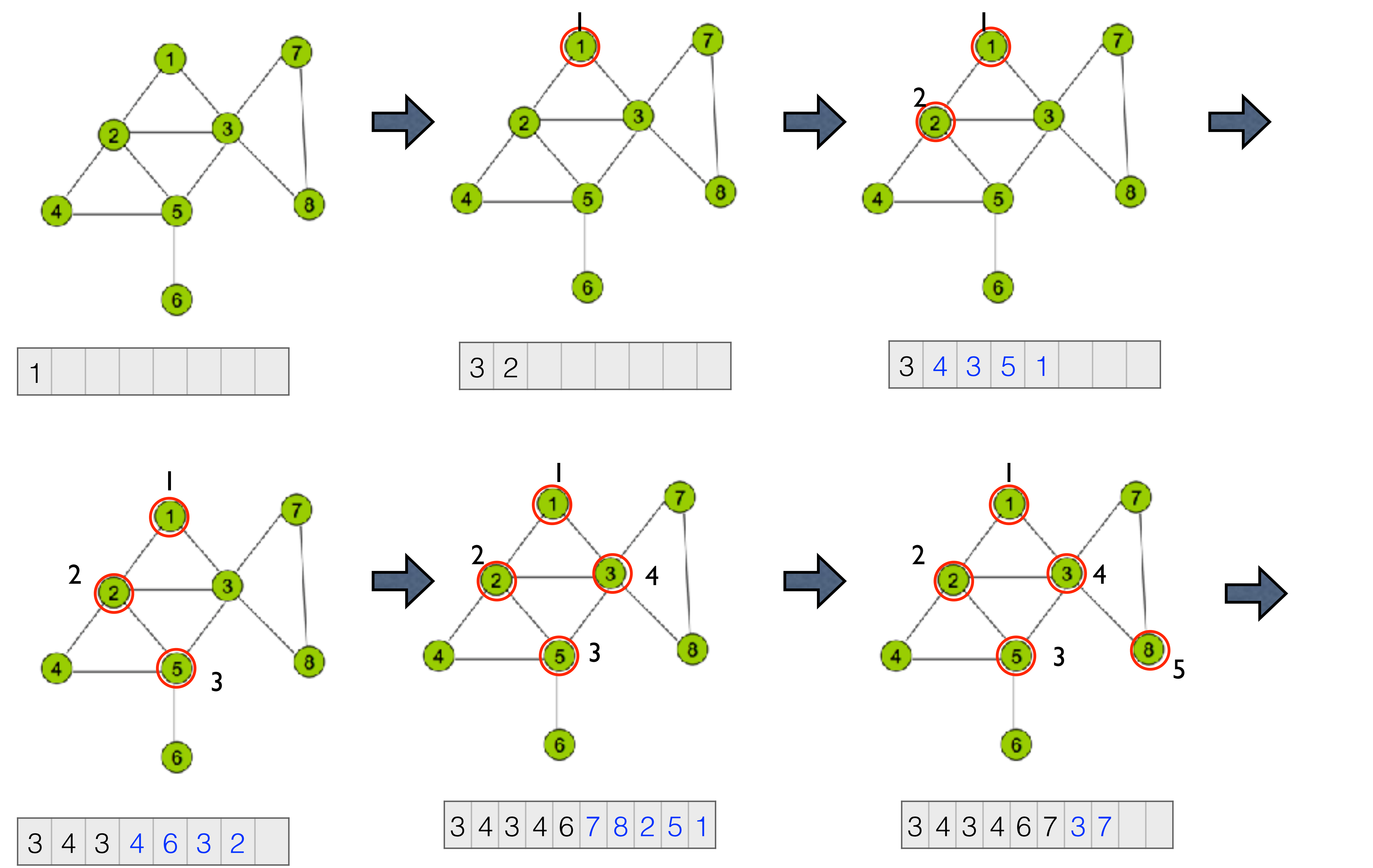
3	4	3	4	6	3	2	



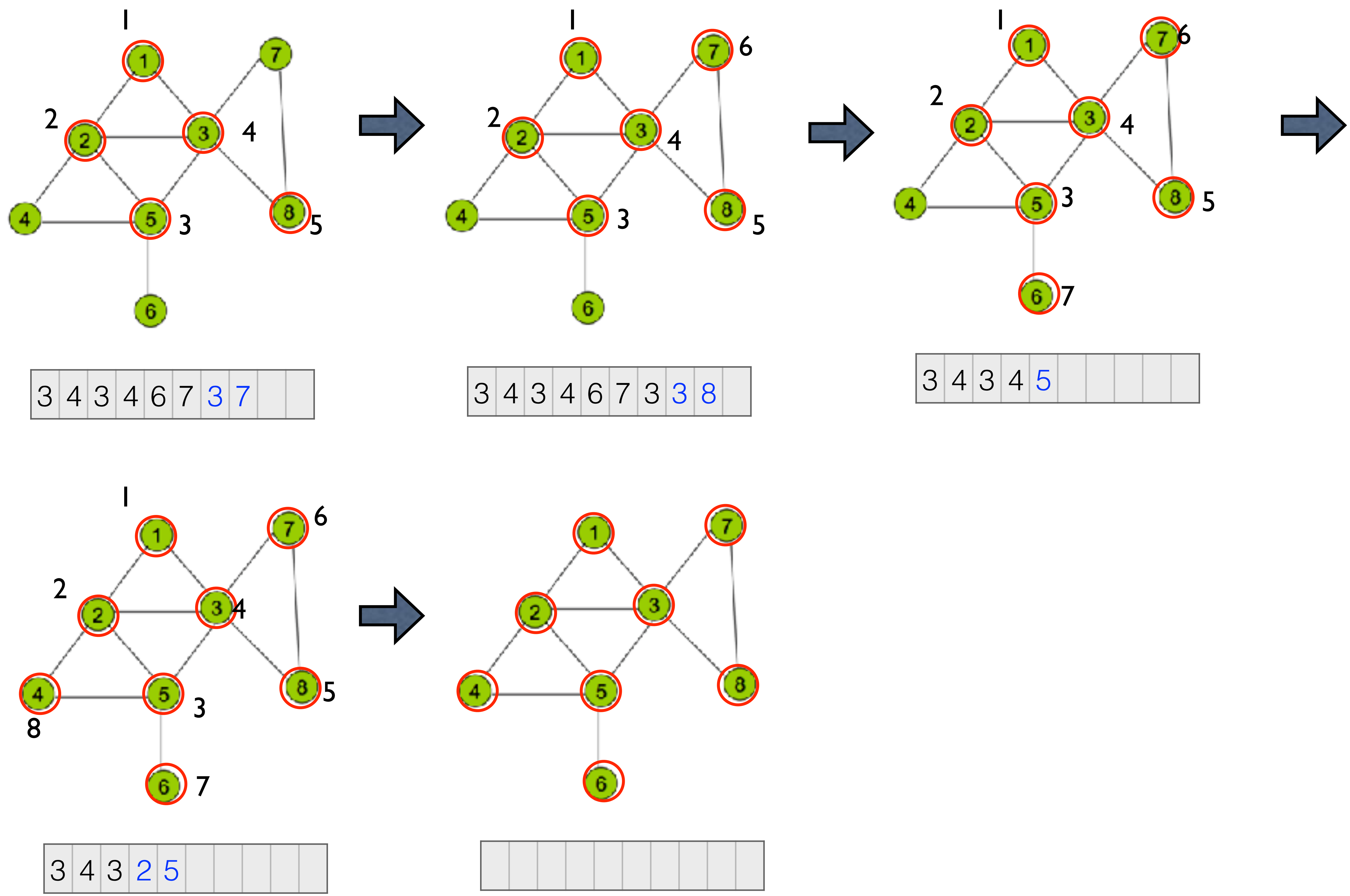
3	4	3	4	6	7	8	2	5	1

```
void DFS (G, u){

    S = stack initialized
    S.push (u)
    while S is not empty
        v = S.pop()
        if v not visited
            mark  v as visited
            for  w is a neighbour of v
                S. push(w)
```

depth-first search: iterative implementation using stack



depth-first search: iterative implementation using stack



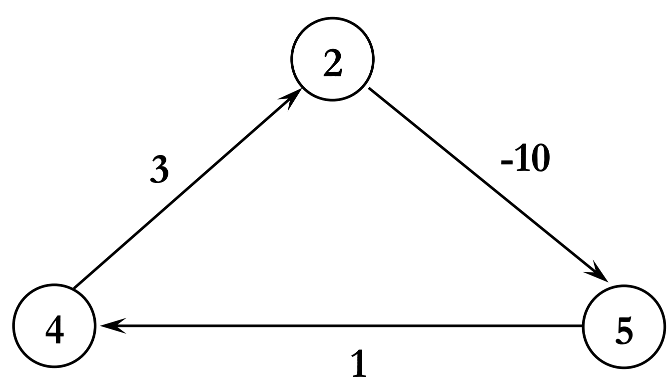
shorted path algorithms

- a weighted graph: a graph G = (V, E), where a cost c_{ij} is associated with each edge
  - weighted path length: $\sum_{i=1}^{n-1} c_{i,i+1}$ for the path $v_1, v_2, \ldots v_n$
  - unweighted path length: the number of edges on the path, i.e. $c_{i,i+1} = 1$
- Single source shortest-path problem
  - given as input a weighted graph G and a distinguished vertex s as source
  - find the shortest weighted path from s to every other v vertex in G

shorted path algorithms

- In many practical applications, we consider finding the shortest path from one vertex s to another t
  - currently no algorithm can find the path from one source to one vertices (ie. s to t) any faster than finding the path from one source to all vertices

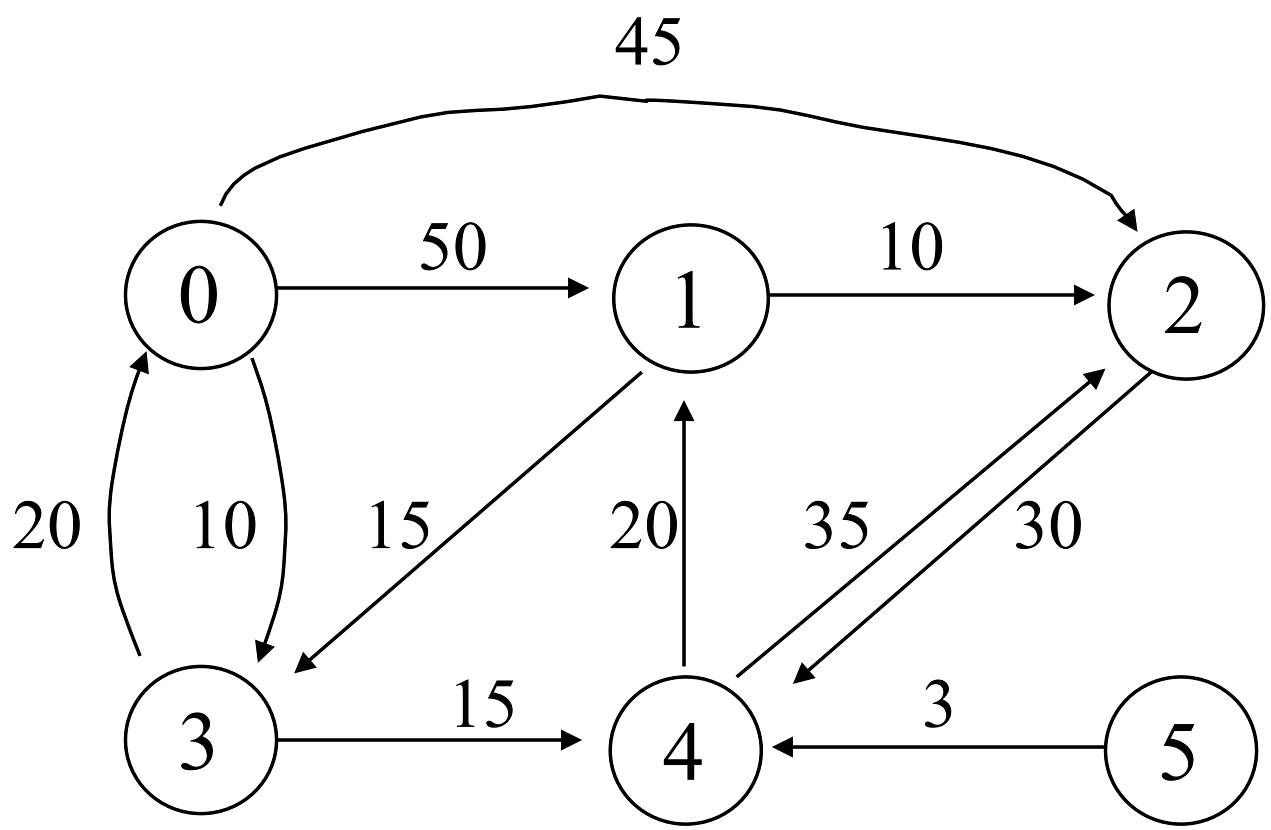
- When negative-cost cycles are present in the graph, the shortest path may be undefined



5 → 4: 1
5 → 4 → 2 → 5 → 4: -5

shorted path algorithms

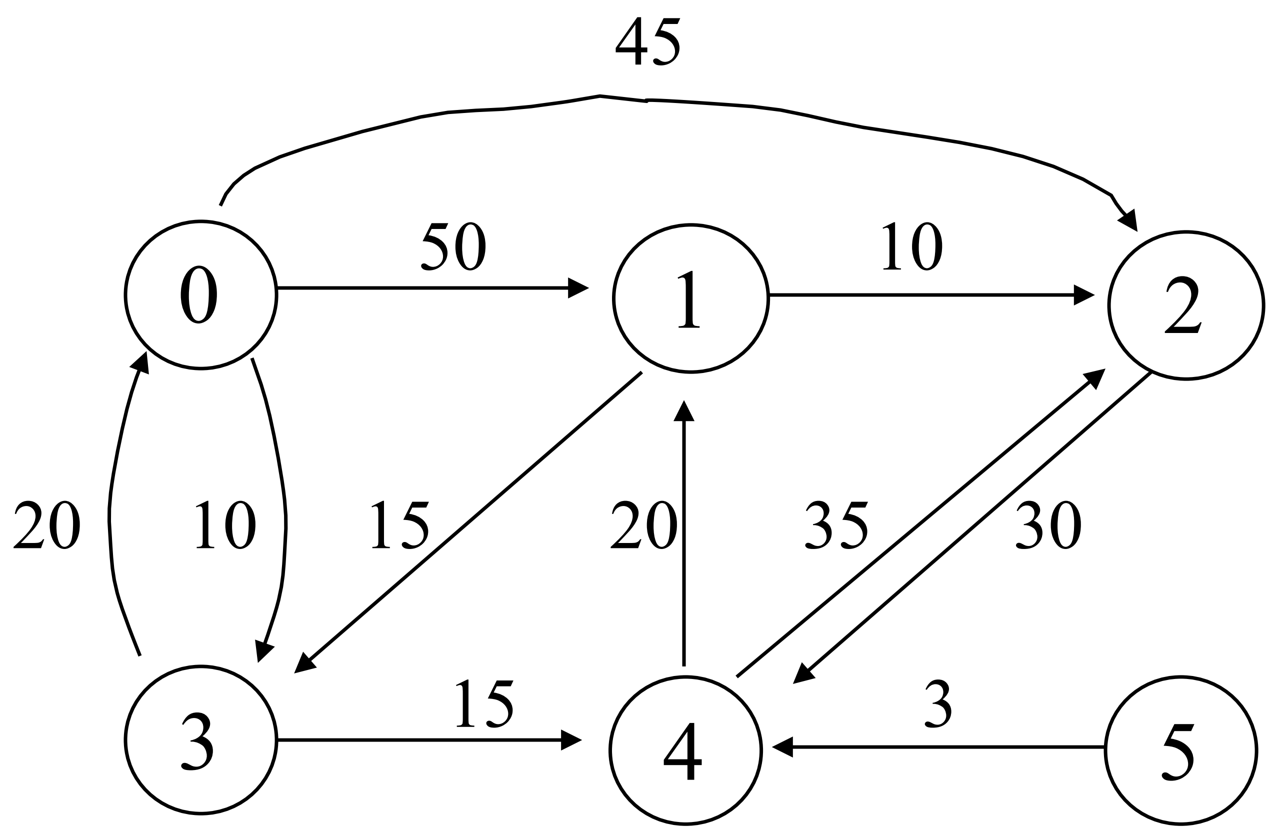
shortest path from 0 to 1?



(a)

shorted path algorithms

shortest path from 0 to 1?



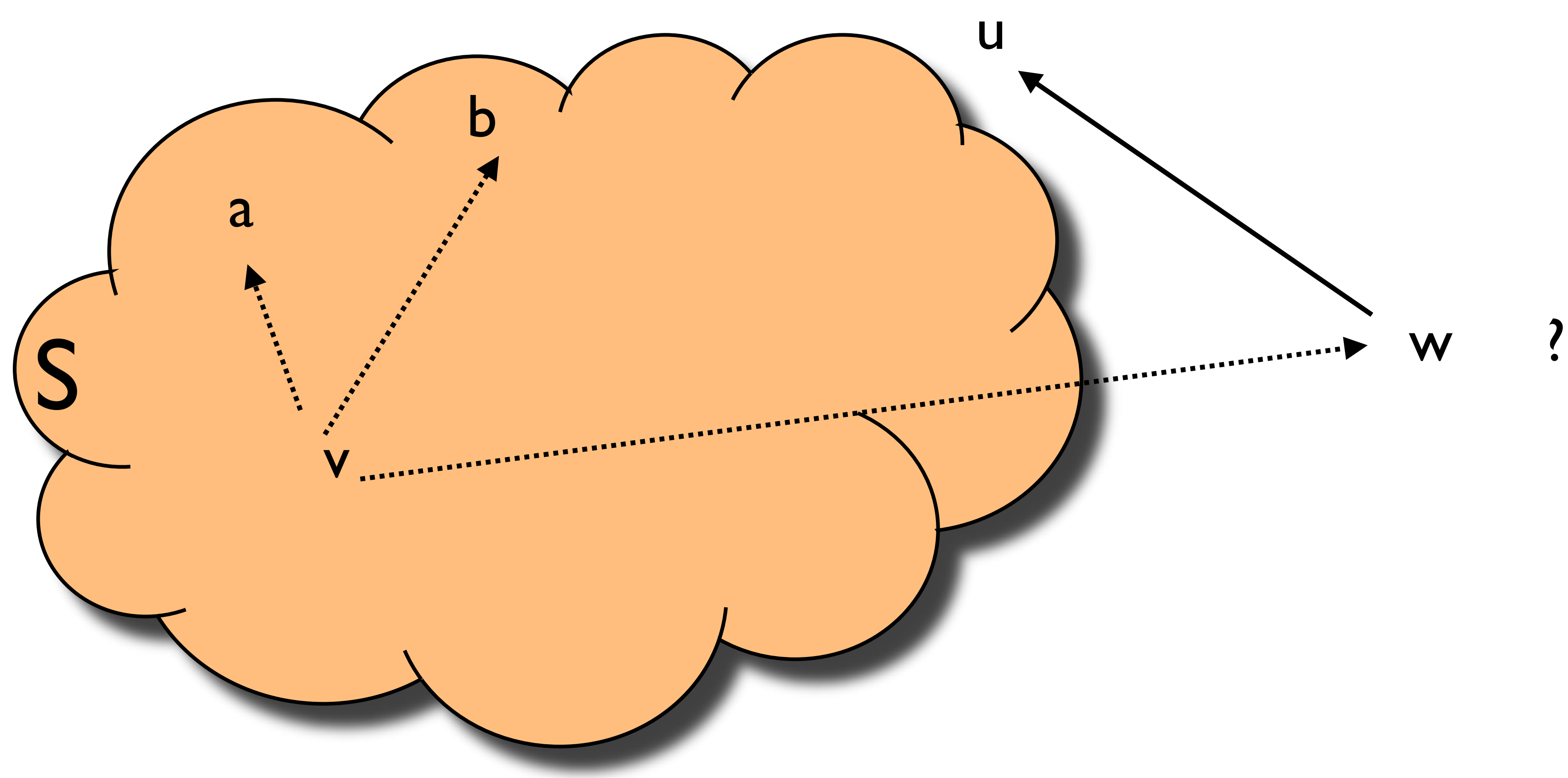
(a)

path	length
1) 0,3	10
2) 0,3,4	25
3) 0,3,4,1	45
4) 0,2	45

(b)

shorted path algorithms

- S is a set of vertices that have the shortest path from v to those vertices
- We generate the paths in non-descending order of length
- When the next shortest path is to vertex u, it possible to have a shortest path v to u through w?
- if vertex u is chosen, it has the minimum distance among all the vertices not in S



weighted single-source shorted path : Dijkstra's algorithm

- length of a path: sum of edge weights along the path
- finding minimum length of the path from u to v: $\delta(s, v)$
- given a directed graph with non-negative edge weights $G = (V, E)$, and a special source vertex $s \in V$, determine the distance from the source vertex to every vertex in G
  - d[v]: shortest path from the source to v
  - pred[v]: previous vertex of v in the path
- each node is one of the status, permanent or temporary
  - the status of a node is permanent if its distance value is equal to the shortest distance from node s
  - otherwise, the status of a node is temporary

weighted single-source shorted path : Dijkstra's algorithm

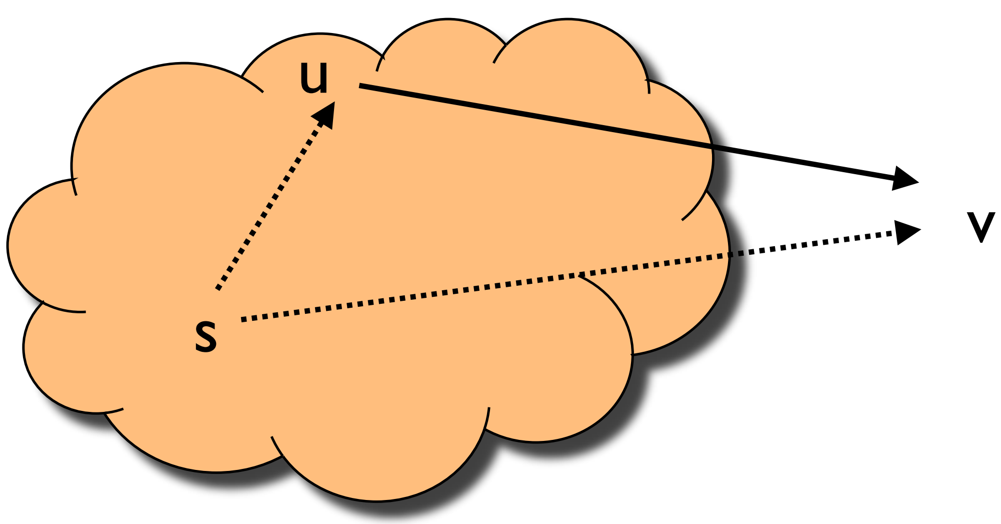
- how does the algorithm work
  - start by assigning some initial values for the distance D[v] from a node s to every other node v in the graph
  - at each step, update the distance to every node and determine a node j that has the smallest distance value d_i among all nodes in the temporary sets
  - if all nodes are labeled as permanent, stop

weighted single-source shorted path : Dijkstra's algorithm

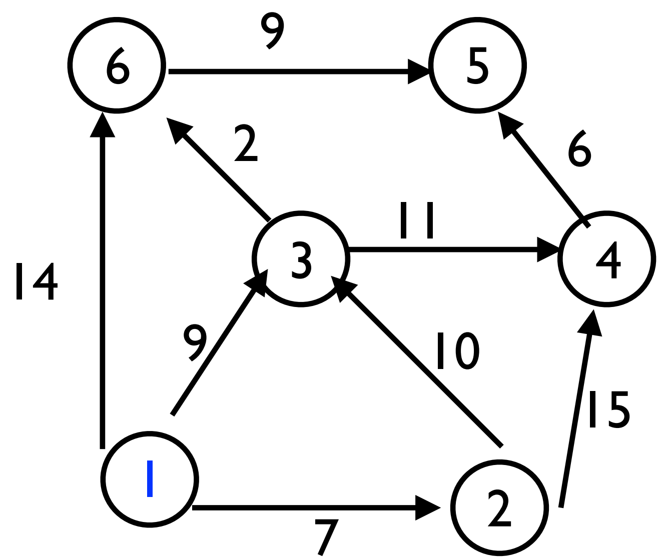
- relaxation (update) process
  - d[v]: shortest path from the source to v
  - pred[v]: previous vertex of v in the path

  - initially d[s] = 0, d[v] = ∞   v: all other nodes except the starting node
  - d[v] is updated until d[v] is converged to minimum distance δ(s, v)
  - implemented with a priority queue: every operation (insert, delete_min, decrease_key) can be done in Θ (log n) time

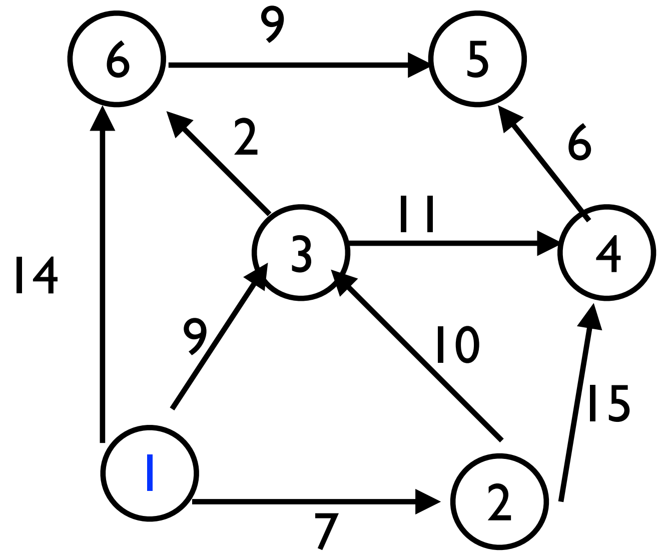
```
if (d[u] + w[u, v] < d[v])
{
   d[v] = d[u] + w[u, v];
   pred[v] = u;
}
```

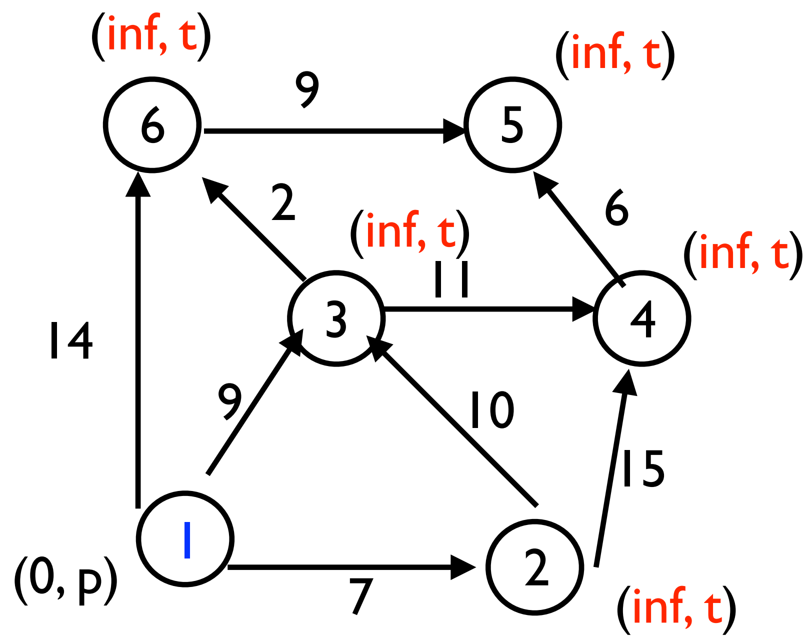


Dijkstra’s algorithm

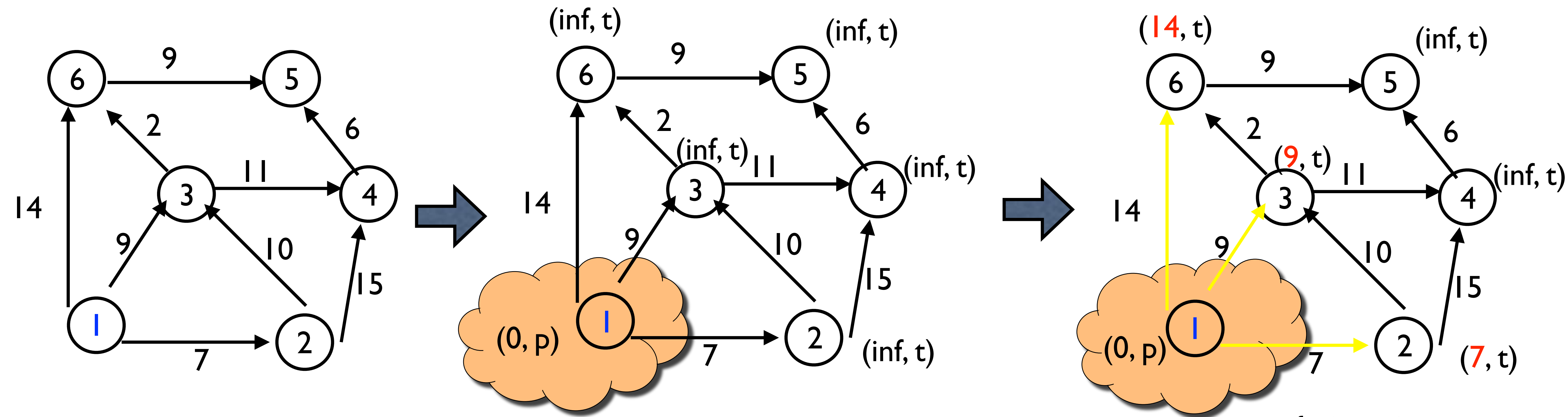


Dijkstra's algorithm





Dijkstra's algorithm

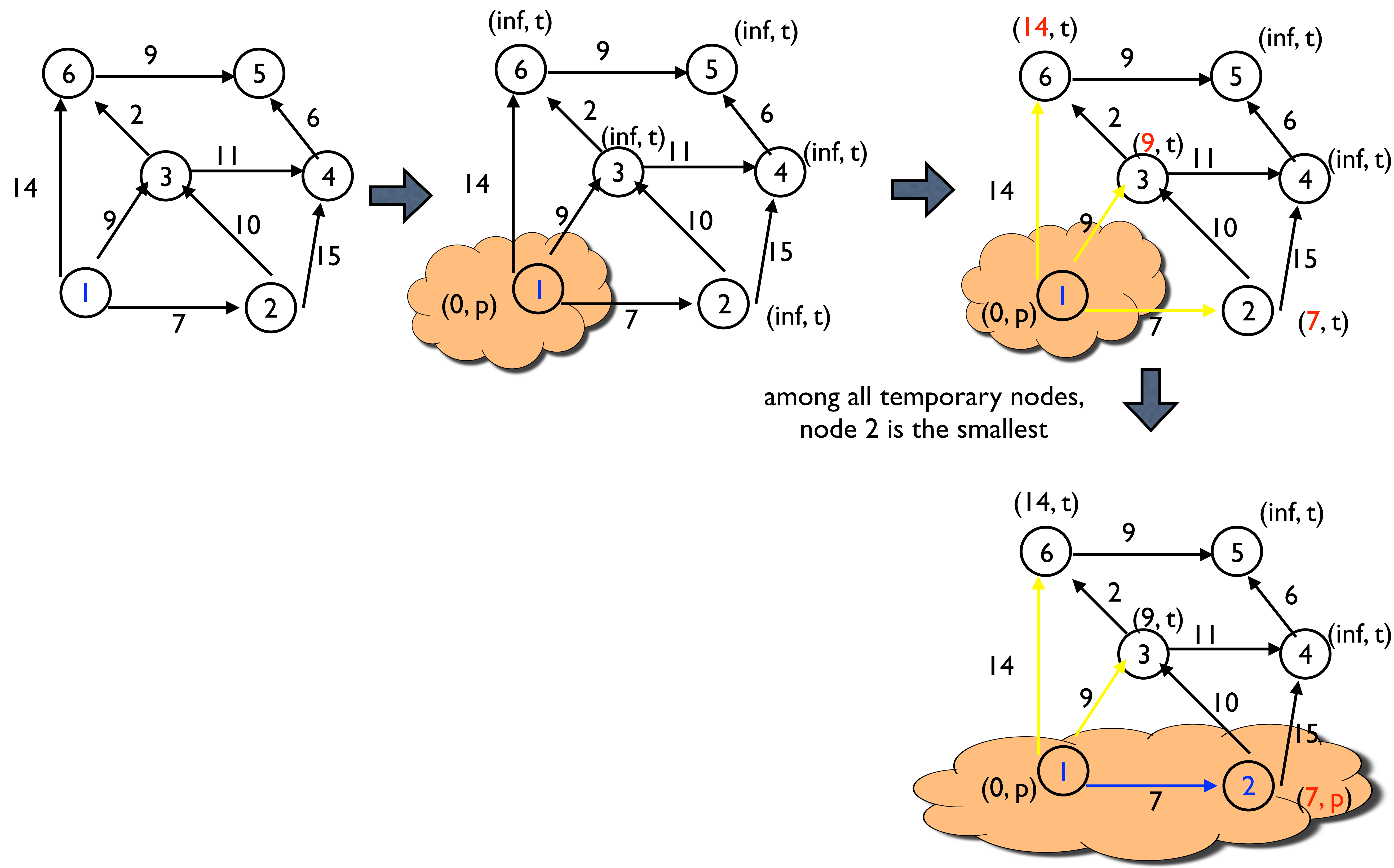


d2 = min{inf, 0+7) = 7

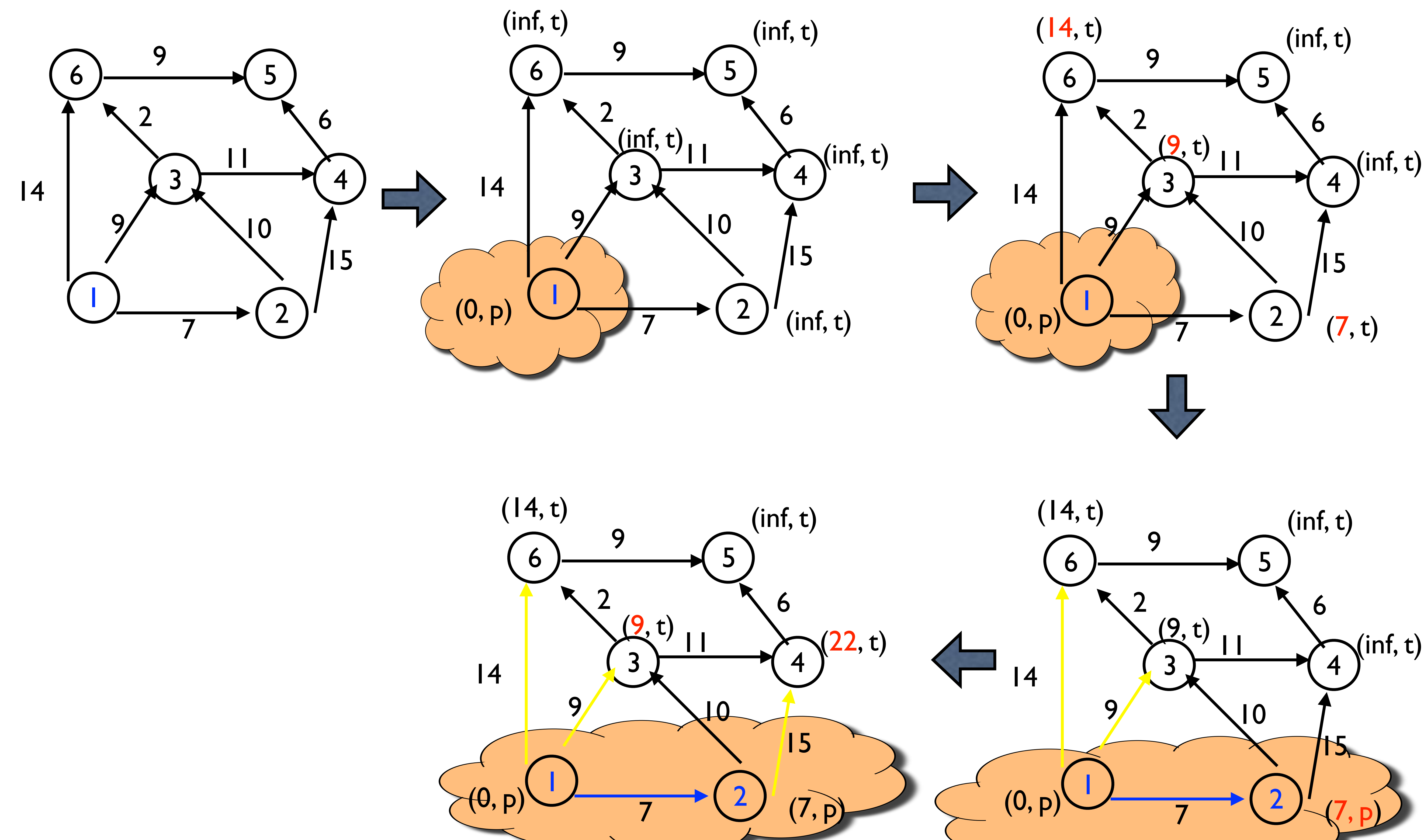
d3 = min{inf, 0+9) = 9

d6 = min{inf, 0+14) = 14

Dijkstra’s algorithm



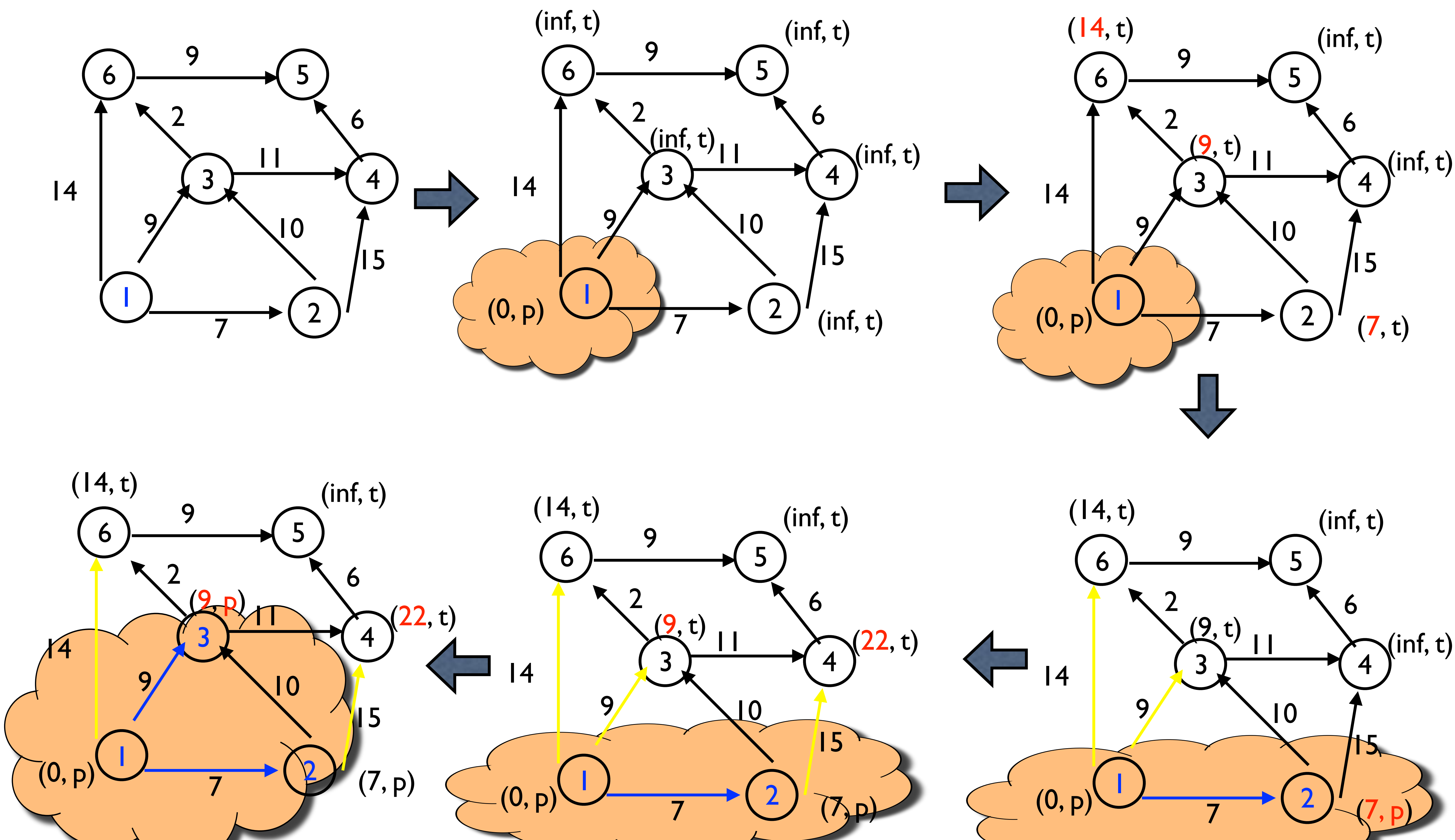
Dijkstra's algorithm



d3 = min{9, 7+10) = 9

d4 = min{inf, 7+15) = 22

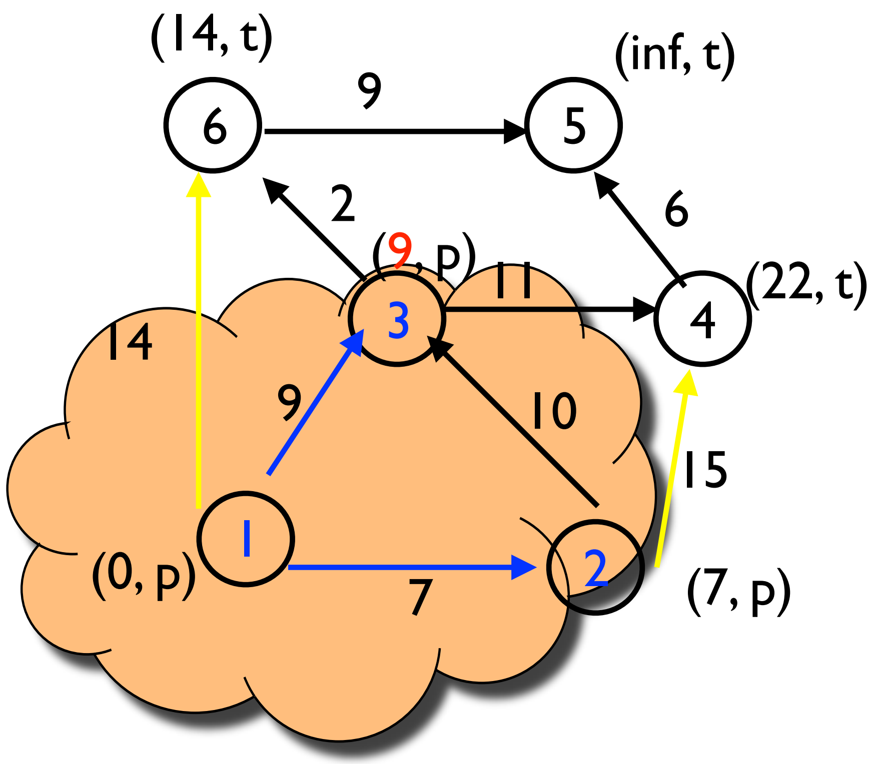
Dijkstra’s algorithm



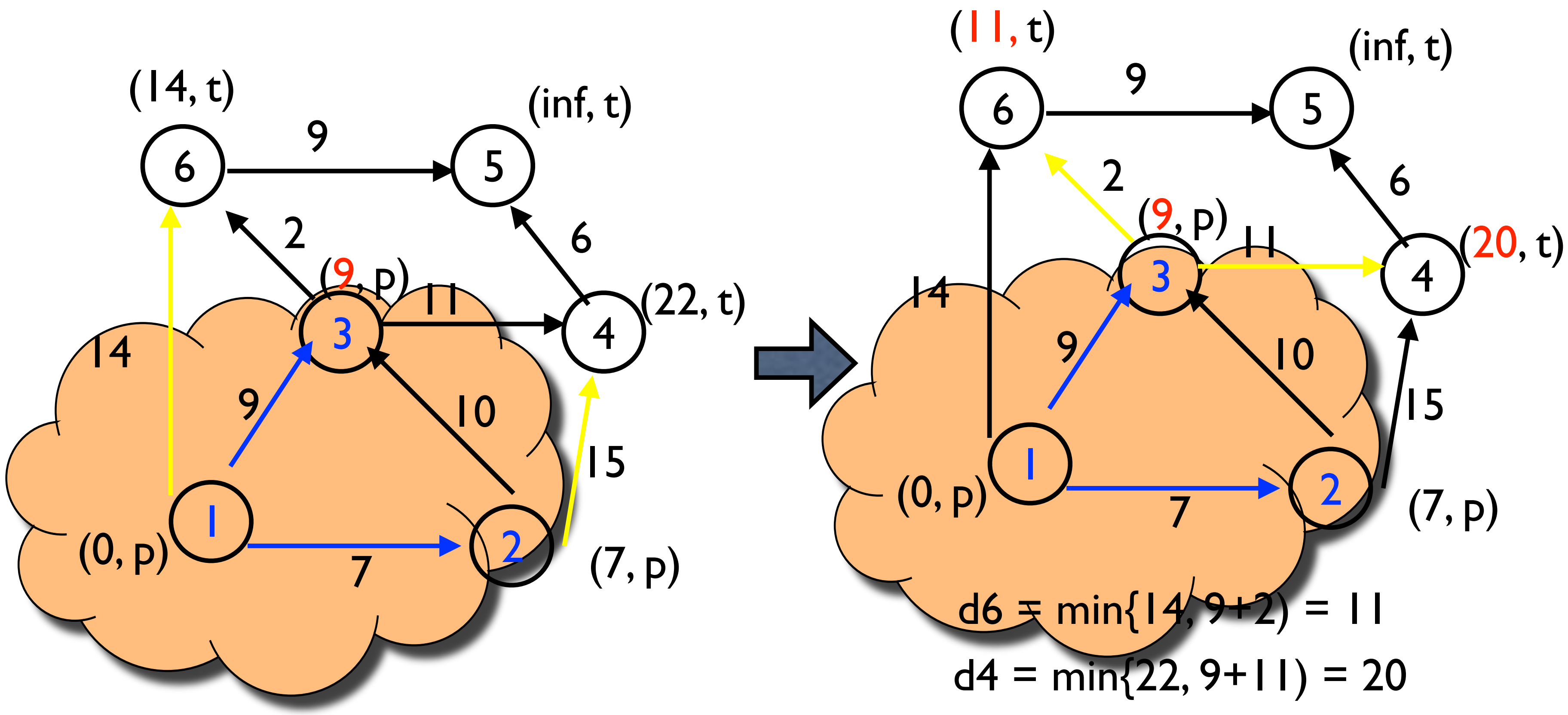
d3 = min{9, 7+10) = 9

d4 = min{inf, 7+15) = 22

Dijkstra's algorithm



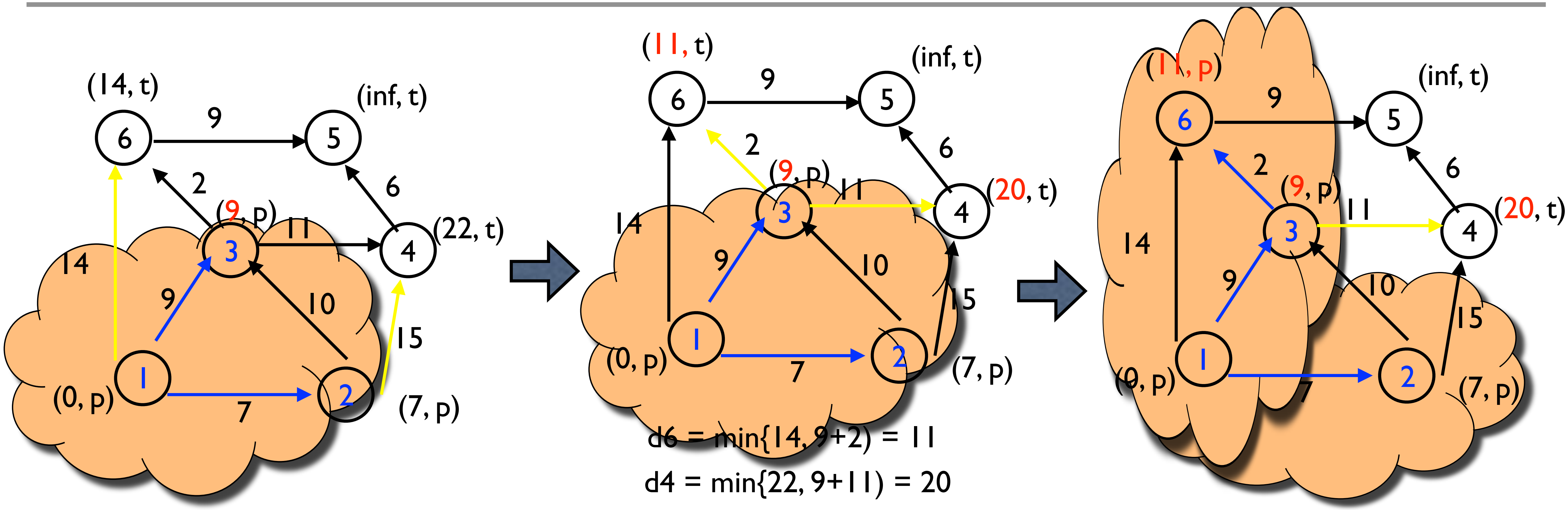
Dijkstra’s algorithm



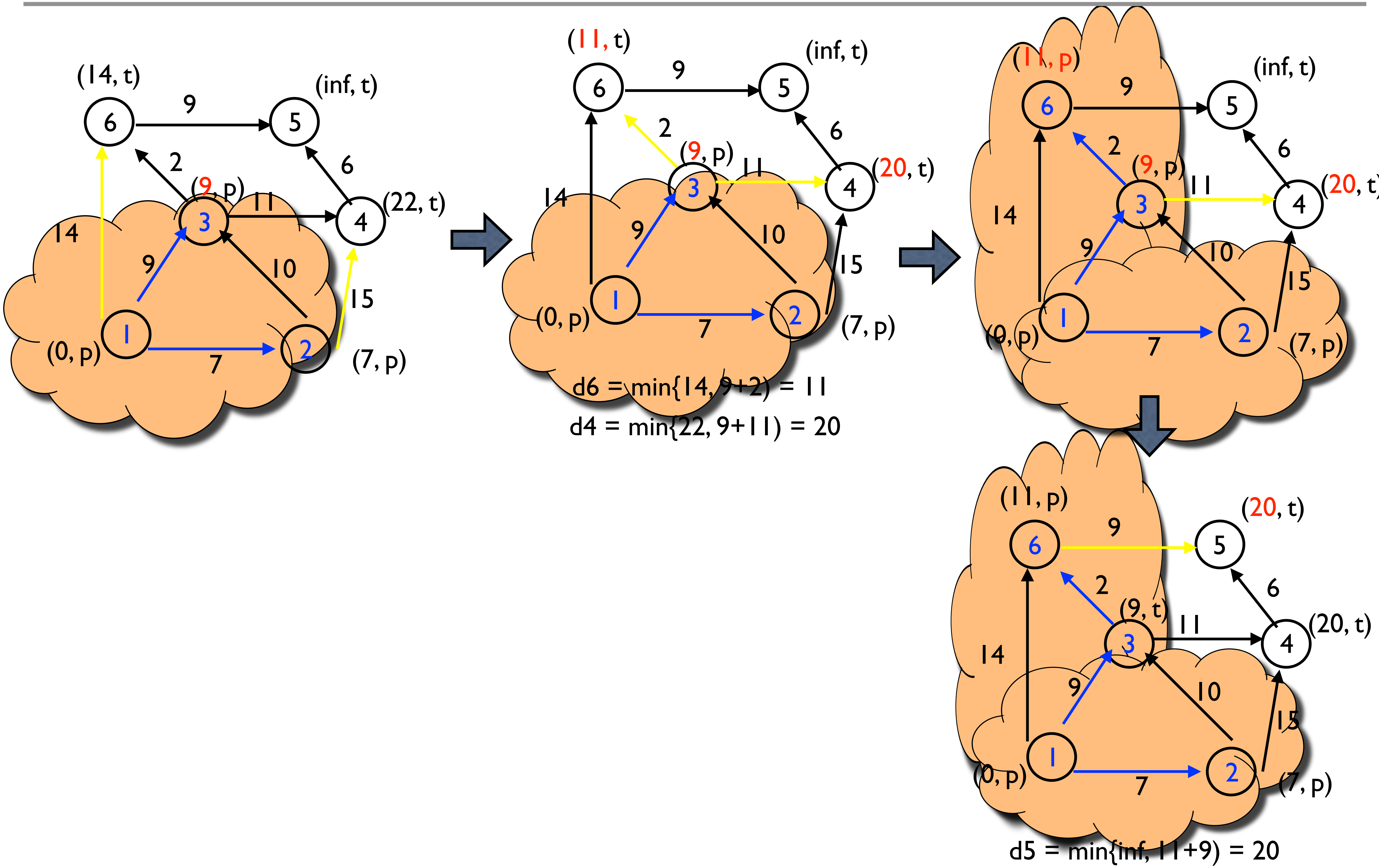
d6 = min{14, 9+2) = 11

d4 = min{22, 9+11) = 20

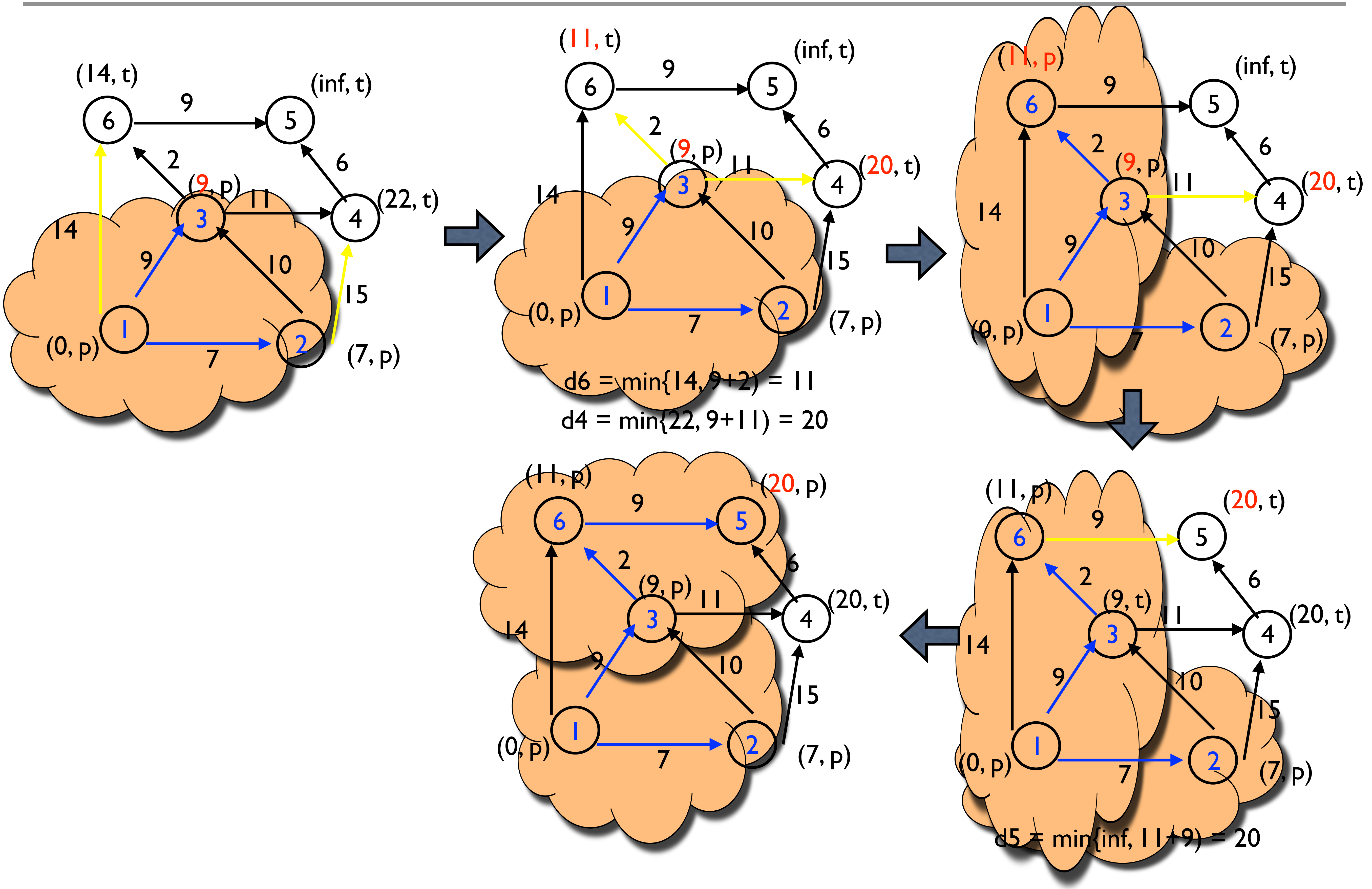
Dijkstra’s algorithm



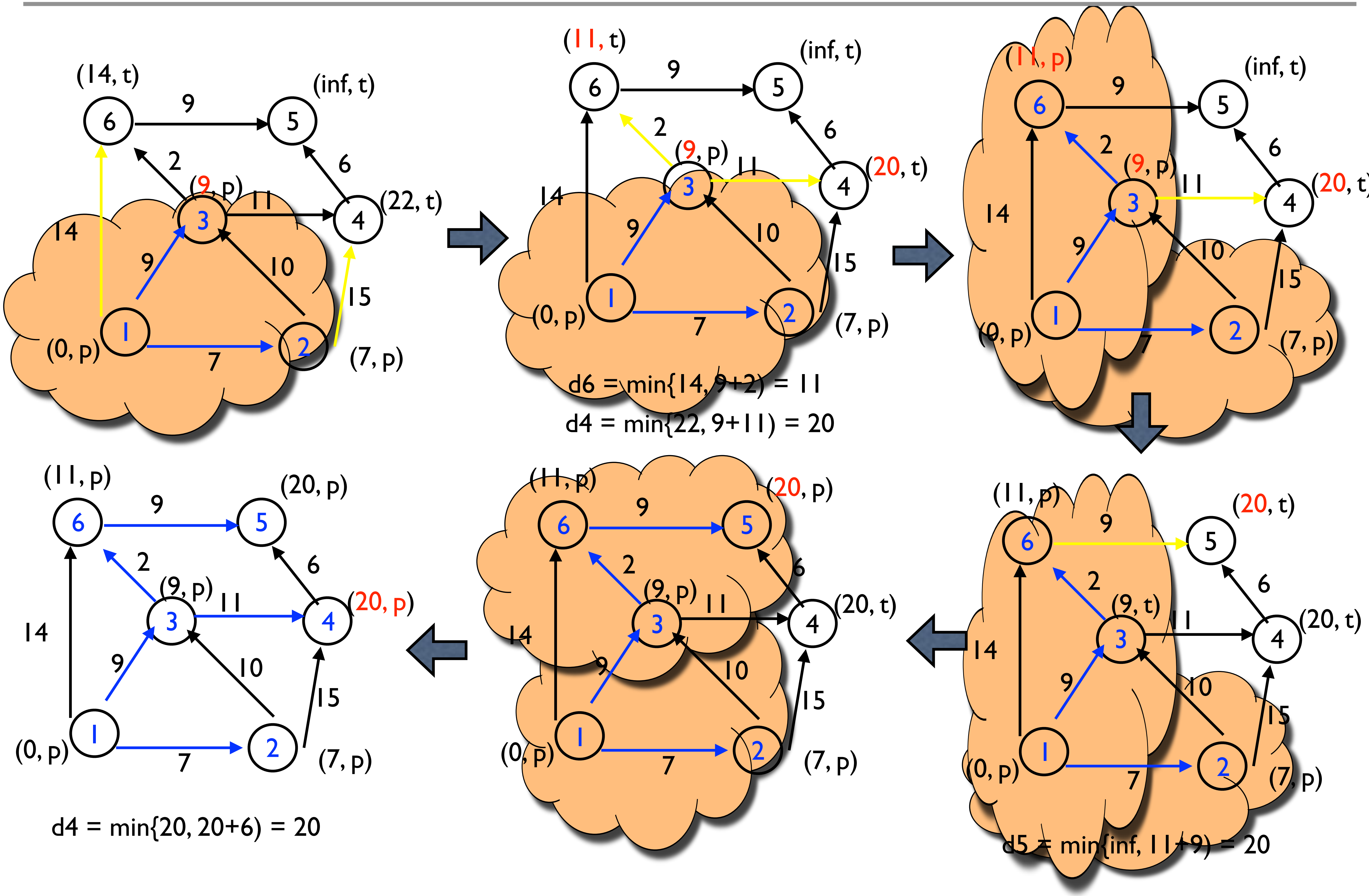
Dijkstra’s algorithm



Dijkstra’s algorithm



Dijkstra's algorithm



Dijkstra's algorithm

```
Dijkstra(G=(V, E, w), s)
{
      SP = {};
      for each v in V do  {
           d[v] = +infinity;      pred[v] = nil;
      }
      d[s] = 0;
      for each v in Adj[s] do {
           d[v] = w[s, v];        pred[v] = s;
      }

      Add each vertex to priority queue Q;

      While (Q is not empty) do  {      /* for each node*/
           u = Delete_Min(Q);
           SP = SP + {u};
           for each v in Adj[u] do   {   /* for each outdeg(u)*/
              if (d[u] + w(u, v) < d[v]) then  {
                  d[v] = d[u] + w(u, v);
                  pred[v] = u;
                  Decrease_Priority(Q, v);
              }
           }
      }
}
```

$\Theta(n)$

$\Theta(\log n)$

$\Theta(\log n)$

$$\boxed{\begin{aligned}&\sum_{u \in V} (\log n + 1 + outdeg(u) \times \log n) = \\ &n \log n + n + (\sum_{u \in V} outdeg(u)) \log n \in \Theta((n + e) \log n)\end{aligned}}$$

Dijkstra's algorithm