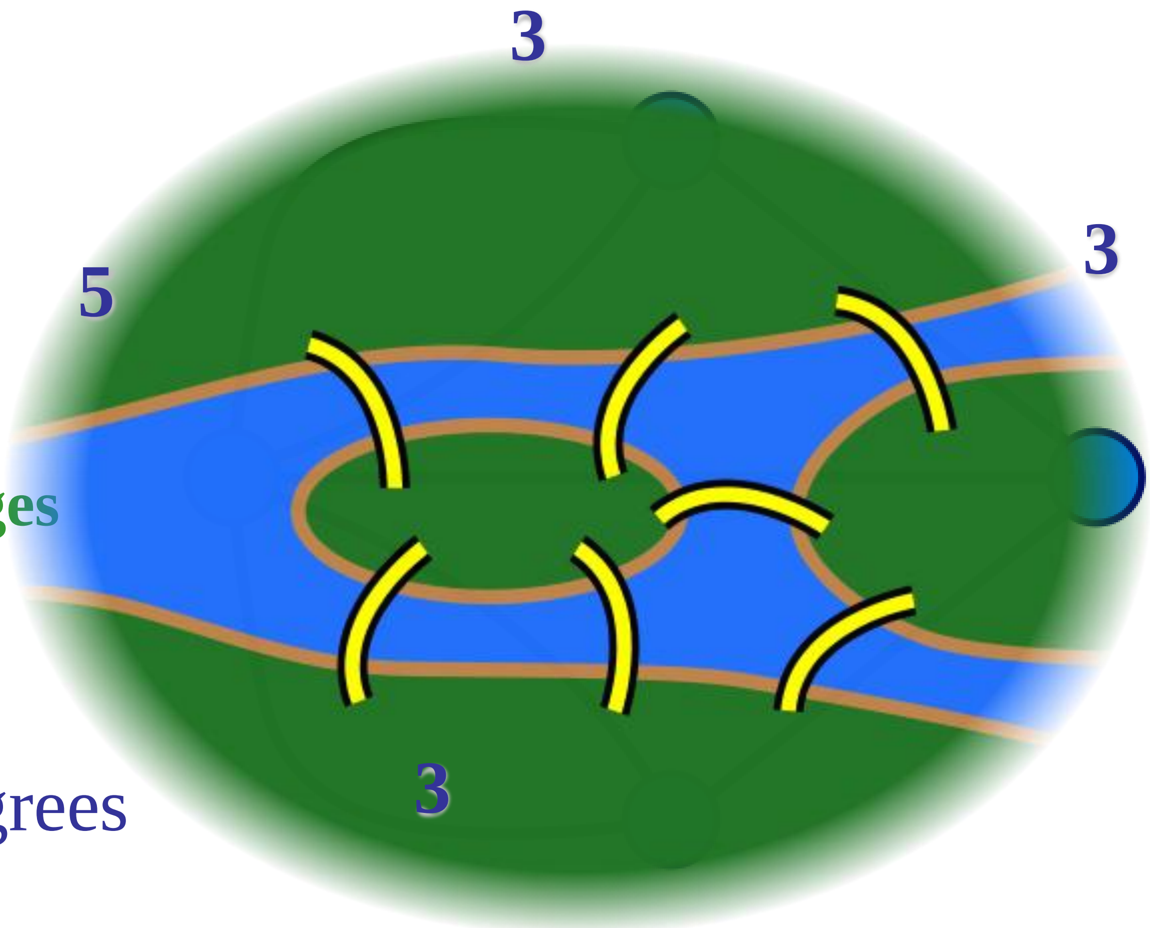
Graph Basics

- **Terminologies**
- **Representations**
- **Traversals**

Data Structures

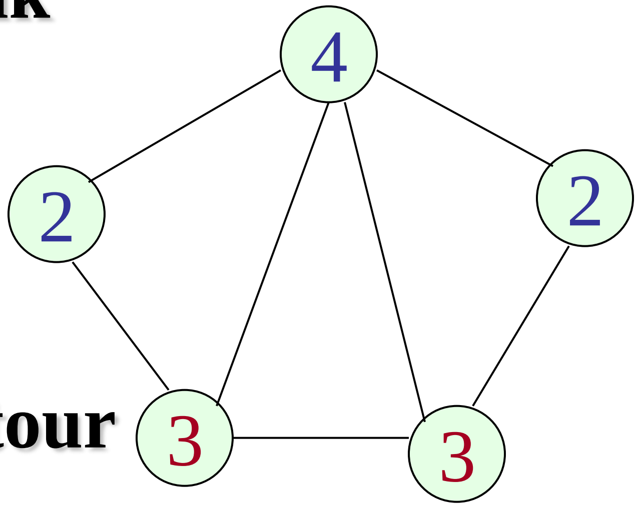
Seven Bridges of Königsberg

- G = {V, E}
  - V(G): vertex set
  - E(G): edge set
  - degree
    - Number of edges
- Vertex types
  - Odd or even degrees



Seven Bridges of Königsberg

- **Eulerian path (trial) / Euler walk**
  - visits every edge exactly once
  - 0 or 2 nodes with odd degrees



- **Eulerian circuit (cycle) / Euler tour**
  - begin and end at the same vertex
  - 0 node with odd degrees

{Euler walks} $\supseteq$ {Euler tours}

Wikipedia ©

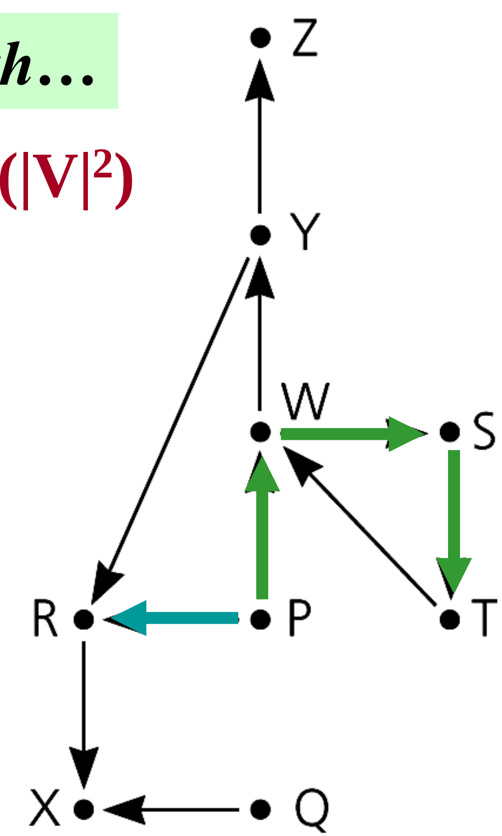
Adjacency Matrix: *Examples*

In-degree vs. Out-degree

	P	Q	R	S	T	W	X	Y	Z
P	0	0	1	0	0	1	0	0	0
Q	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	1	0	0
S	0	0	0	0	1	0	0	0	0
T	0	0	0	0	0	1	0	0	0
W	0	0	0	1	0	0	0	1	0
X	0	0	0	0	0	0	0	0	0
Y	0	0	1	0	0	0	0	0	1
Z	0	0	0	0	0	0	0	0	0

Traverse a path…

***traverse*(g): O(|V|^2)**

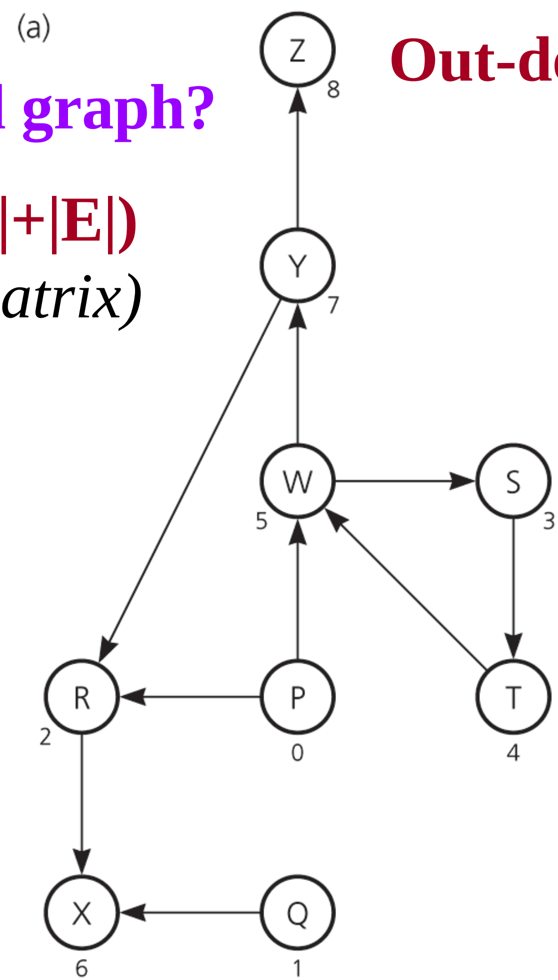


Adjacency List: *Examples*

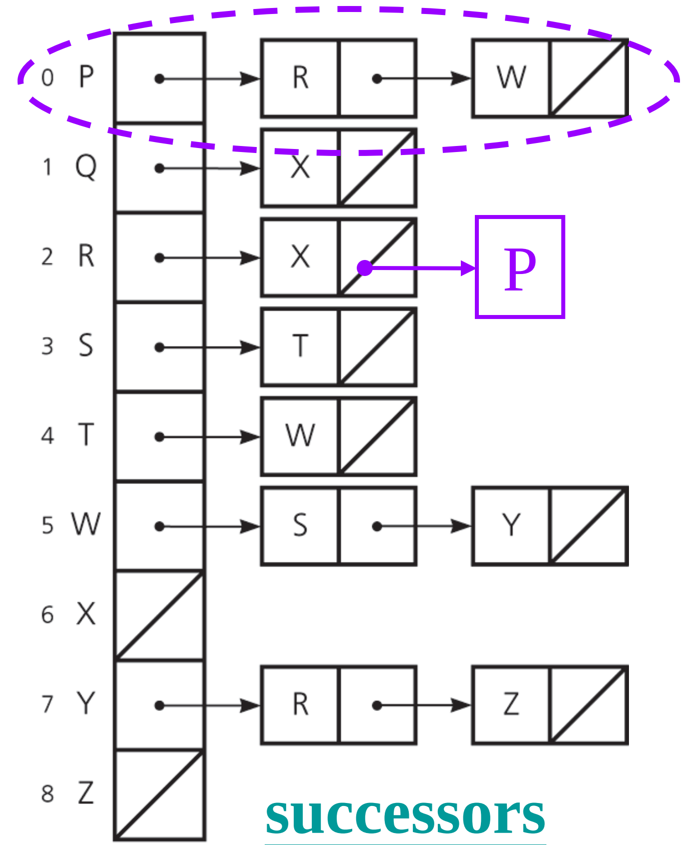
Q&A: undirected graph?

***traverse*(g): O(|V|+|E|)**

(if it is a sparse matrix)



Out-degree



successors

Other Graph Representations

- **Mapping from vertex labels to array indices**

  P Q R S T W X Y Z

  0 1 2 3 4 5 6 7 8

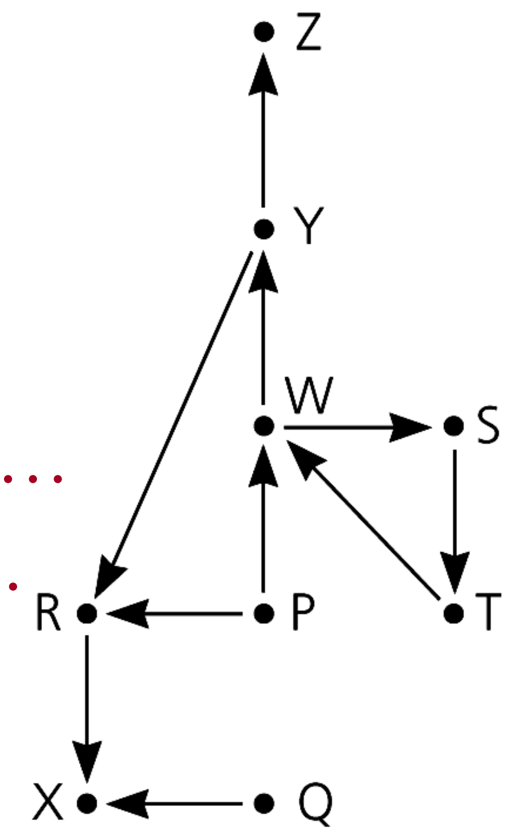
- **Sequential representation**

  – nodes + edges

  [0][1][2][3]…[7][8][9][10][11][12][13]…

  10 12 1314…19 20 20 2 5 6 6 ……

  Q➔X



20 = |V|+|E|+1

undirected graph: |V|+2|E|+1

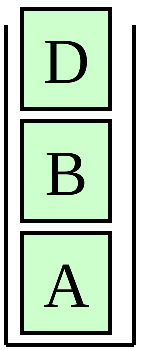
DFS in *iterative* form (stack)

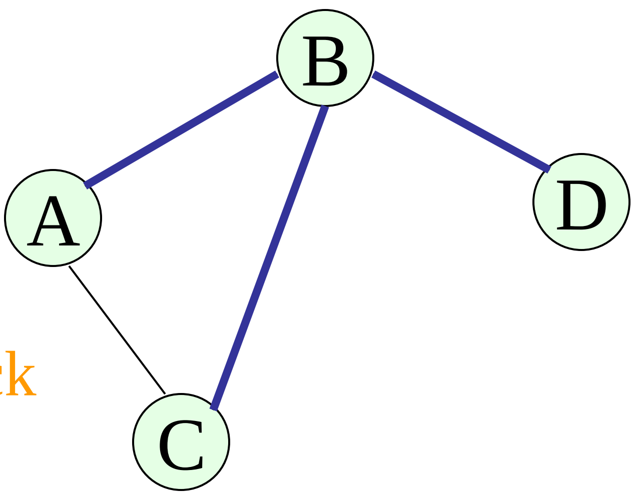
Data Structures

DFS traversal sequence: AB BC BD

```
iterativeDFS(Vertex v)
    s.createStack();
    s.push(v);
    Mark v as visited;
    while (!s.isEmpty())
    {     u = s.getTop();                          // at top of the stack
          if (unvisited vertex w is adjacent to u)
          {        s.push(w);
                   Mark w as visited;              // output
          }
          else     s.pop();                        // backtrack
    }
```

If (*any visited vertex adjacent to u*)
Cycle is found!





BFS in *iterative* form (queue)

BFS traversal sequence: AB AC BD

DFS traversal sequence: AB BC BD

iterativeBFS(Vertex v)

q.createQueue();

q.**enqueue**(v);

Mark v as visited;

while (!q.isEmpty())

{ q.**dequeue**(u);

for (*each unvisited vertex w adjacent to u*)

{ *Mark w as visited*; // output

q.**enqueue**(w);

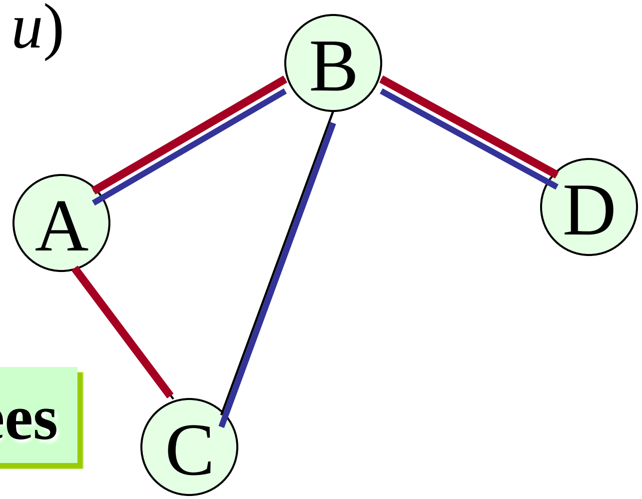
}

}

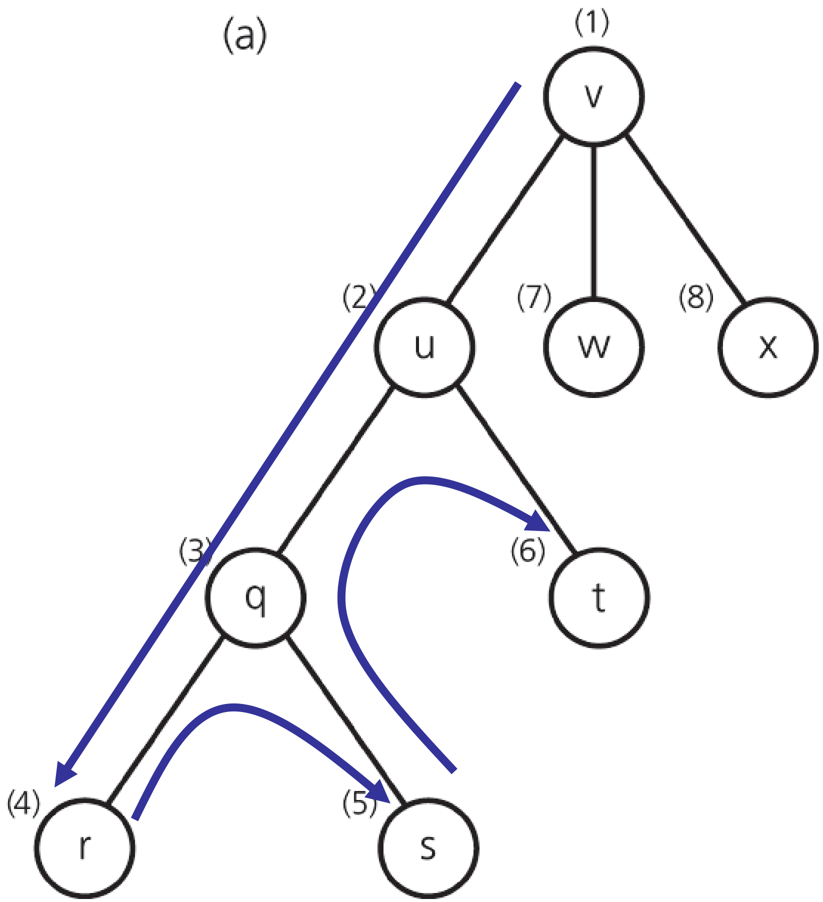
If (*any visited vertex adjacent to u*)
Cycle is found!

B	C	D

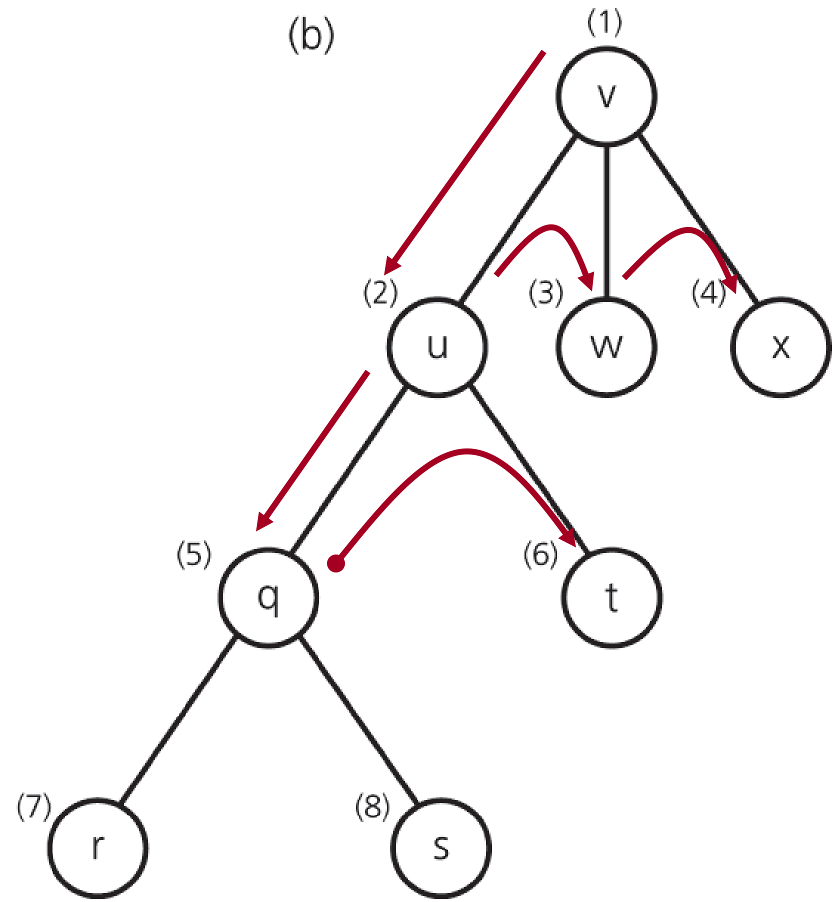
spanning trees



DFS and BFS Traversals



DFS

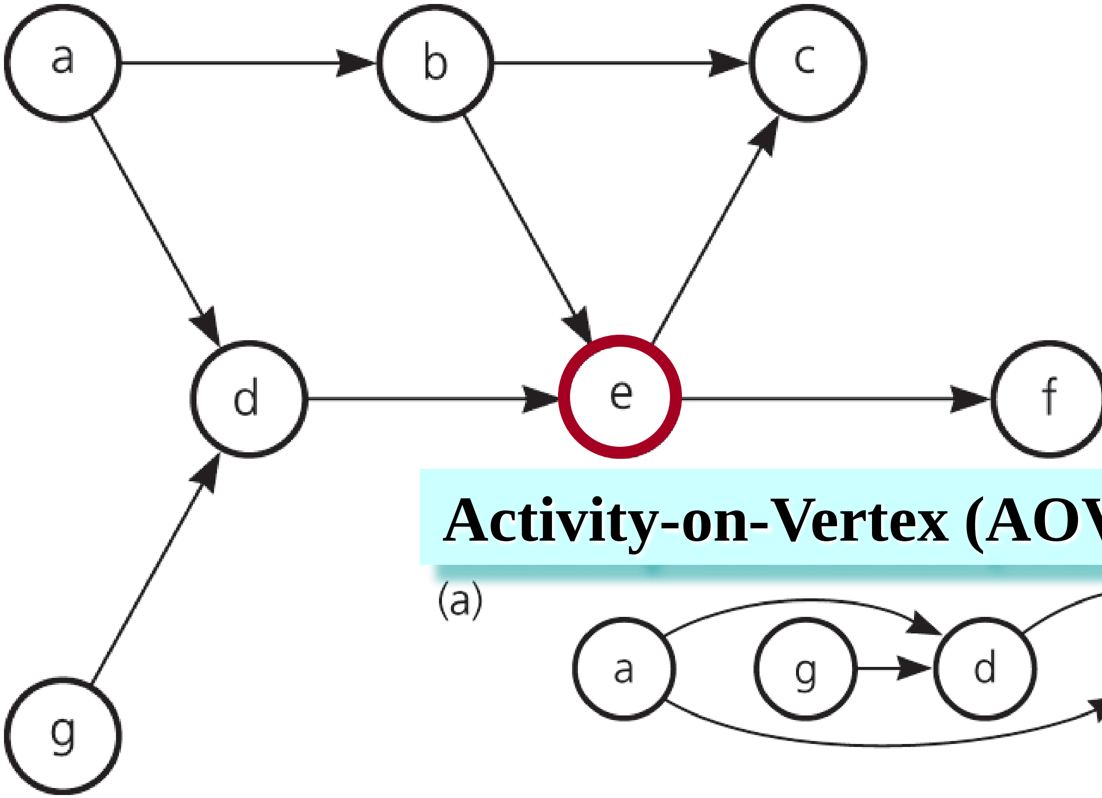


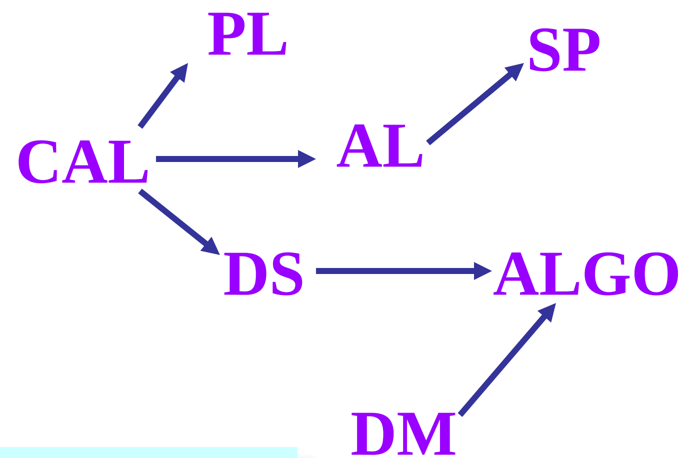
BFS

Graph App.

- **Topological Sort**
- **Spanning Tree**
- **Shortest Paths**

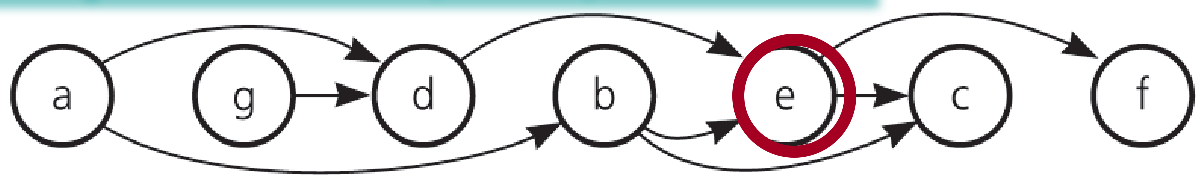
Topological Sort: *Examples*



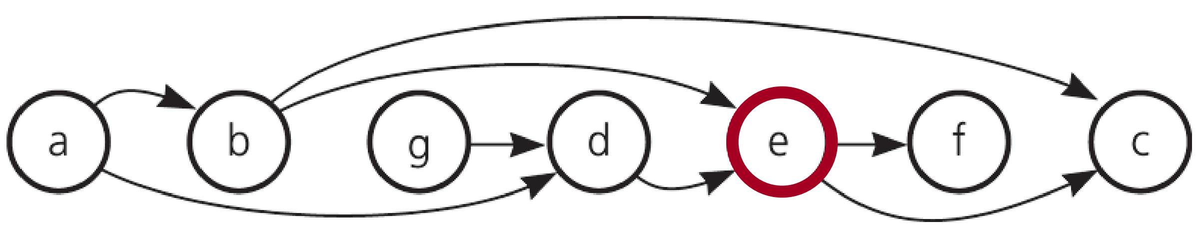


Activity-on-Vertex (AOV) Network

(a)

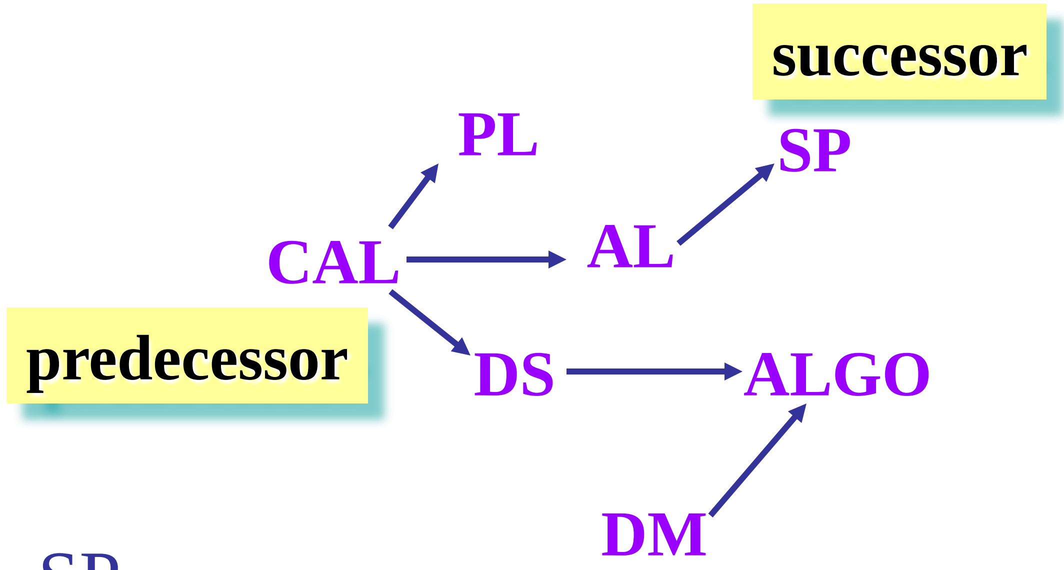


(b)



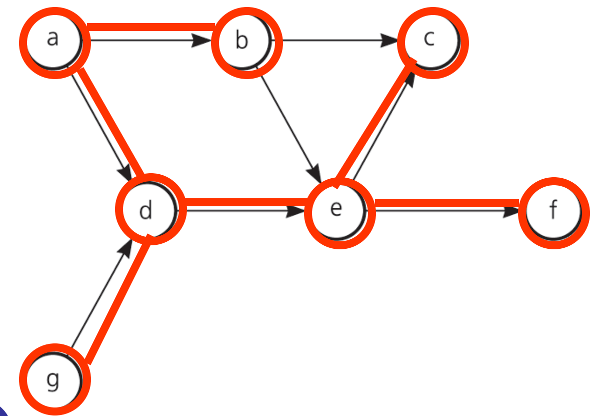
A Trace of *topSort1* (concept)

- **List**
  - SP
  - ALGO, SP
  - DM, ALGO, SP
  - PL, DM, ALGO, SP
  - DS, DM, ALGO, SP
  - AL, DS, DM, ALGO, SP
  - CAL, AL, DS, DM, ALGO, SP



A Trace of *topSort2*

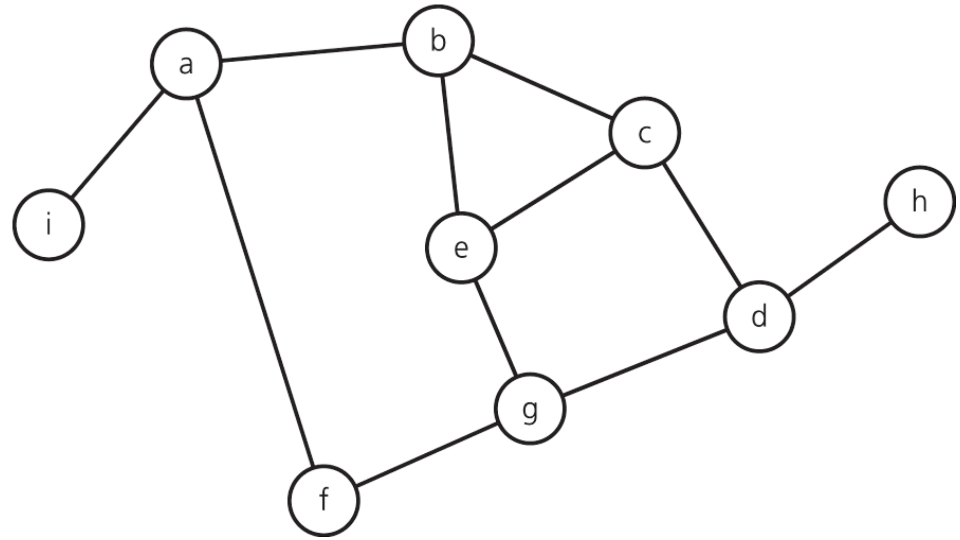
Action	Stack s (bottom to top)	List aList (beginning to end)
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	
Push c	a g d e c	
Pop c, add c to aList	a g d e	c
Push f	a g d e f	c
Pop f, add f to aList	a g d e	f c
Pop e, add e to aList	a g d	e f c
Pop d, add d to aList	a g	d e f c
Pop g, add g to aList	a	g d e f c
Push b	a b	g d e f c
Pop b, add b to aList	a	b g d e f c
Pop a, add a to aList	(empty)	a b g d e f c

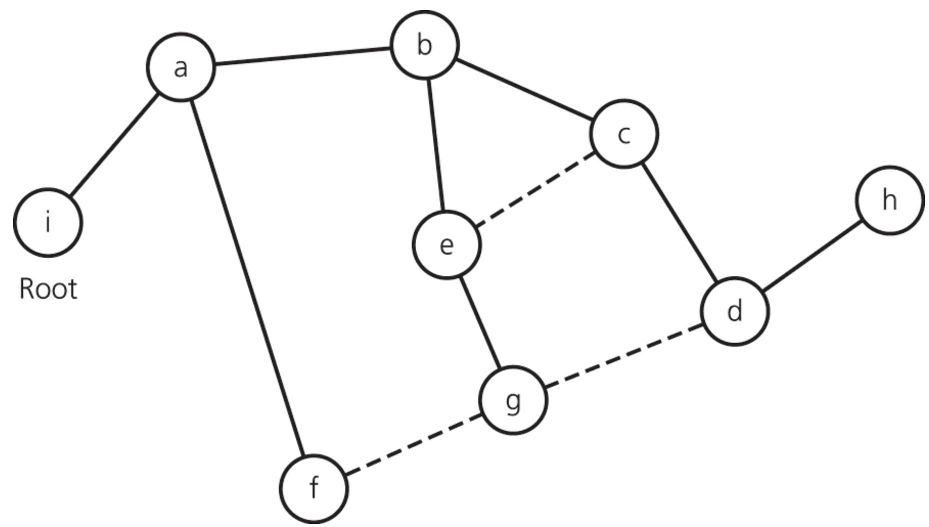


Spanning Tree: *Definition*

Data Structures

- **To obtain a spanning tree from a connected undirected graph with cycles**
  - Remove edges until there are no cycles



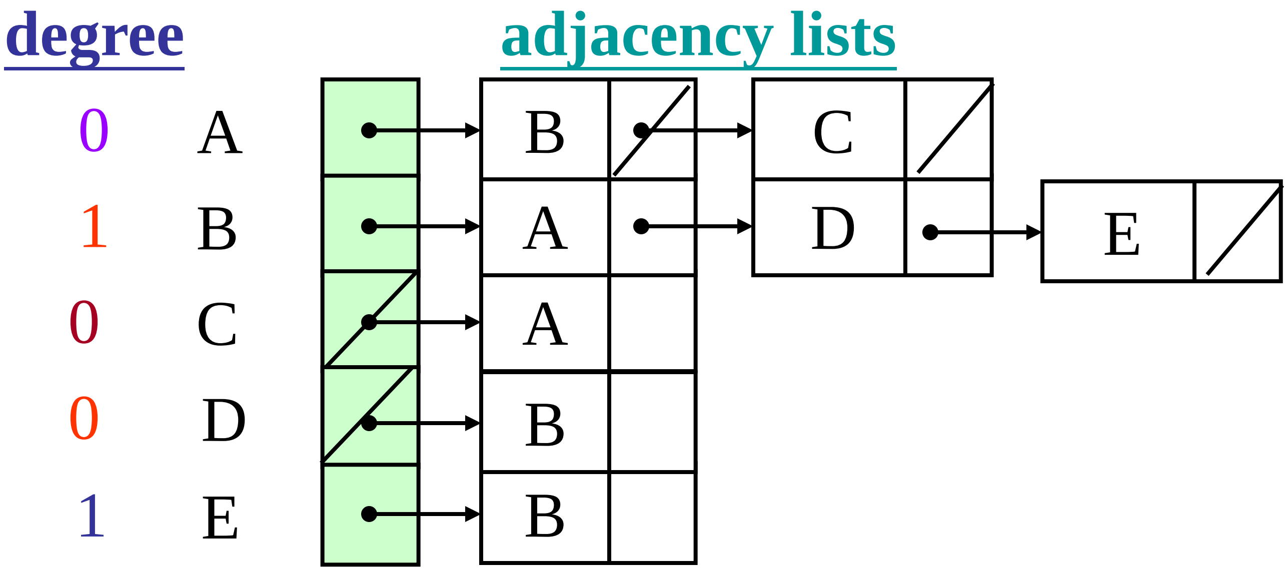


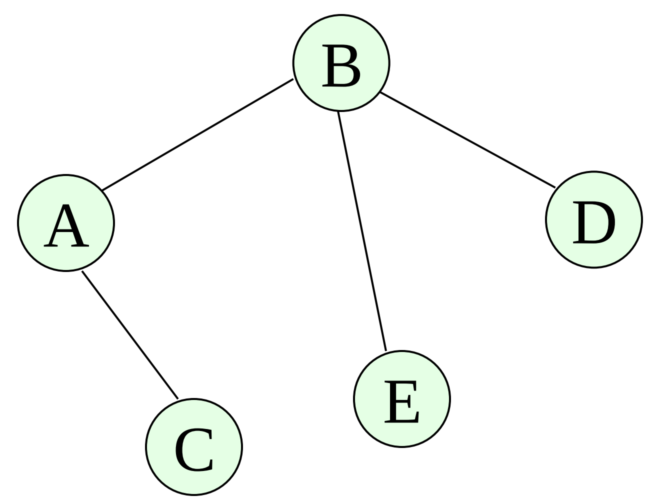
Prüfer Sequence

Prüfer sequence: A B B

degree: [A B C D E]

A	B	C	D	E
0	2	0	1	1
0	1	0	0	1





Q6. *Prüfer Sequence*

Data Structures

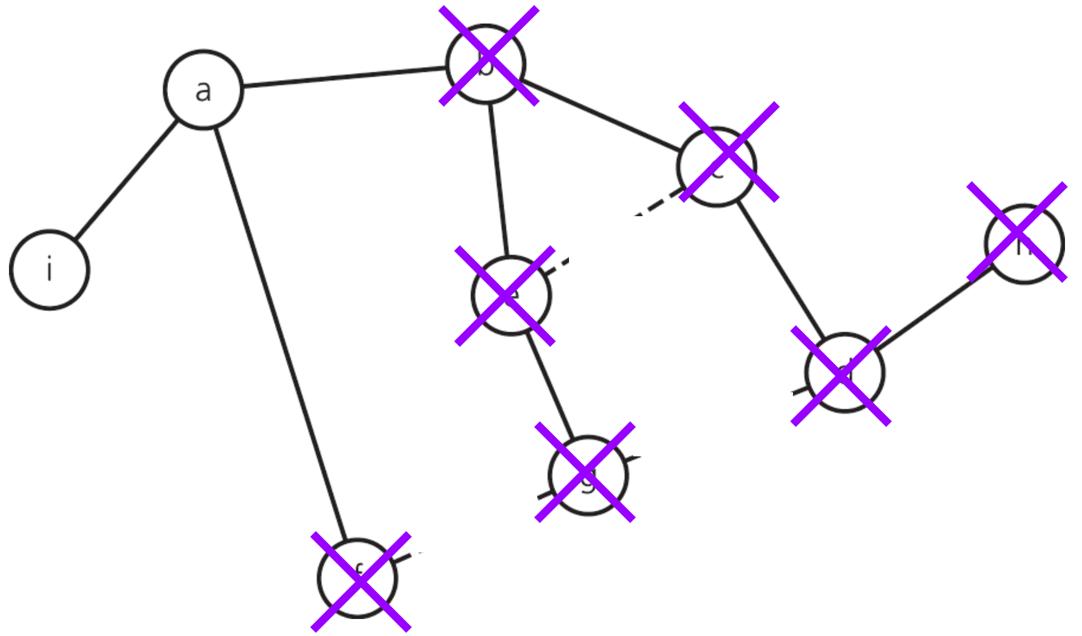
- What is the **Prüfer sequence** of the following graph?

1. a d a e b c b

2. a e b a b c d

3. a e b d c b a a

4. a e b d c b a



Minimum Spanning Tree: *Definition*

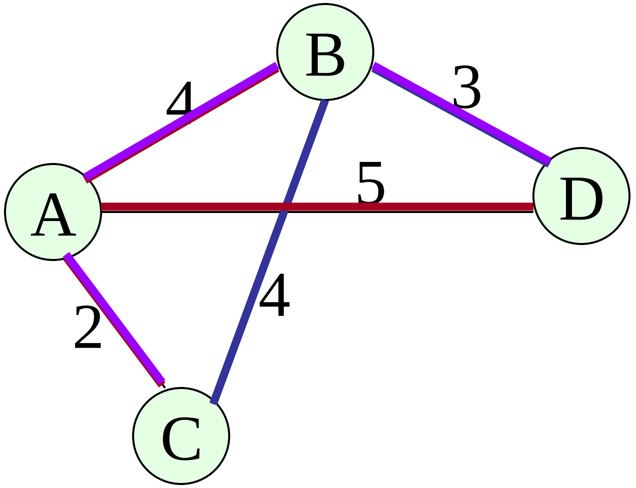
Data Structures

- **Cost of spanning tree**
  - Sum of the edge weights on a spanning tree
- **A *minimum spanning tree* of a connected undirected graph has a minimal edge-weight sum**
  - A particular graph could have *several* minimum spanning trees

DFS: 4+4+3=11

BFS: 4+2+5=11

MST: 4+2+3=9



Prim's Algorithm

Minimum Spanning Tree (MST): AC AB BD

BD BC CA

PrimAlgorithm(Vertex v)

Mark v as visited; count=0;

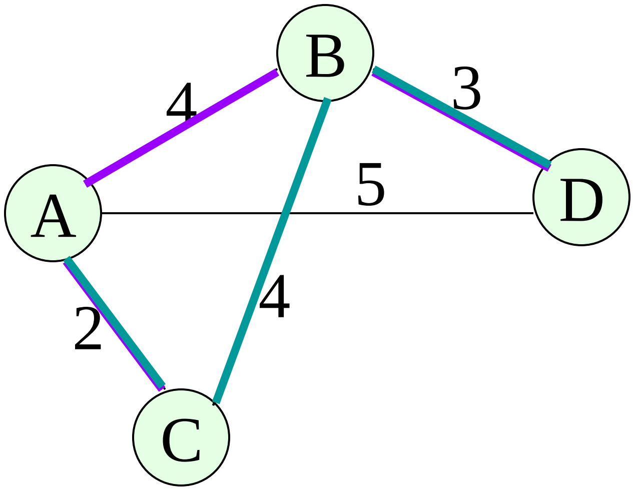
while (count < |V|-1)

(v,u) = the least-cost edge from visited to unvisited

Mark u as visited;

Add ***(v,u)*** into MST;

count++;



Kruskal's Algorithm

Minimum Spanning Tree (MST): AC BD AB

KruskalAlgorithm()

Assign a unique label to each vertex; count=0;

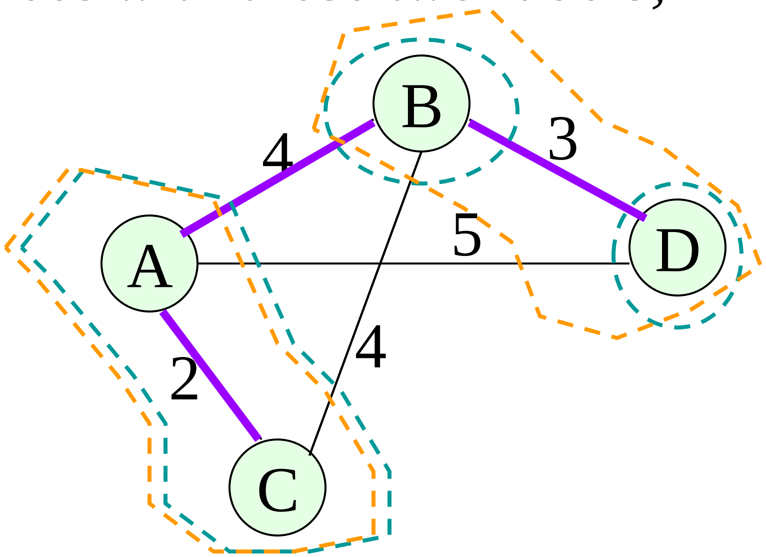
while (count < |V|-1)

(v,u)= the least-cost edge of two vertices with different labels

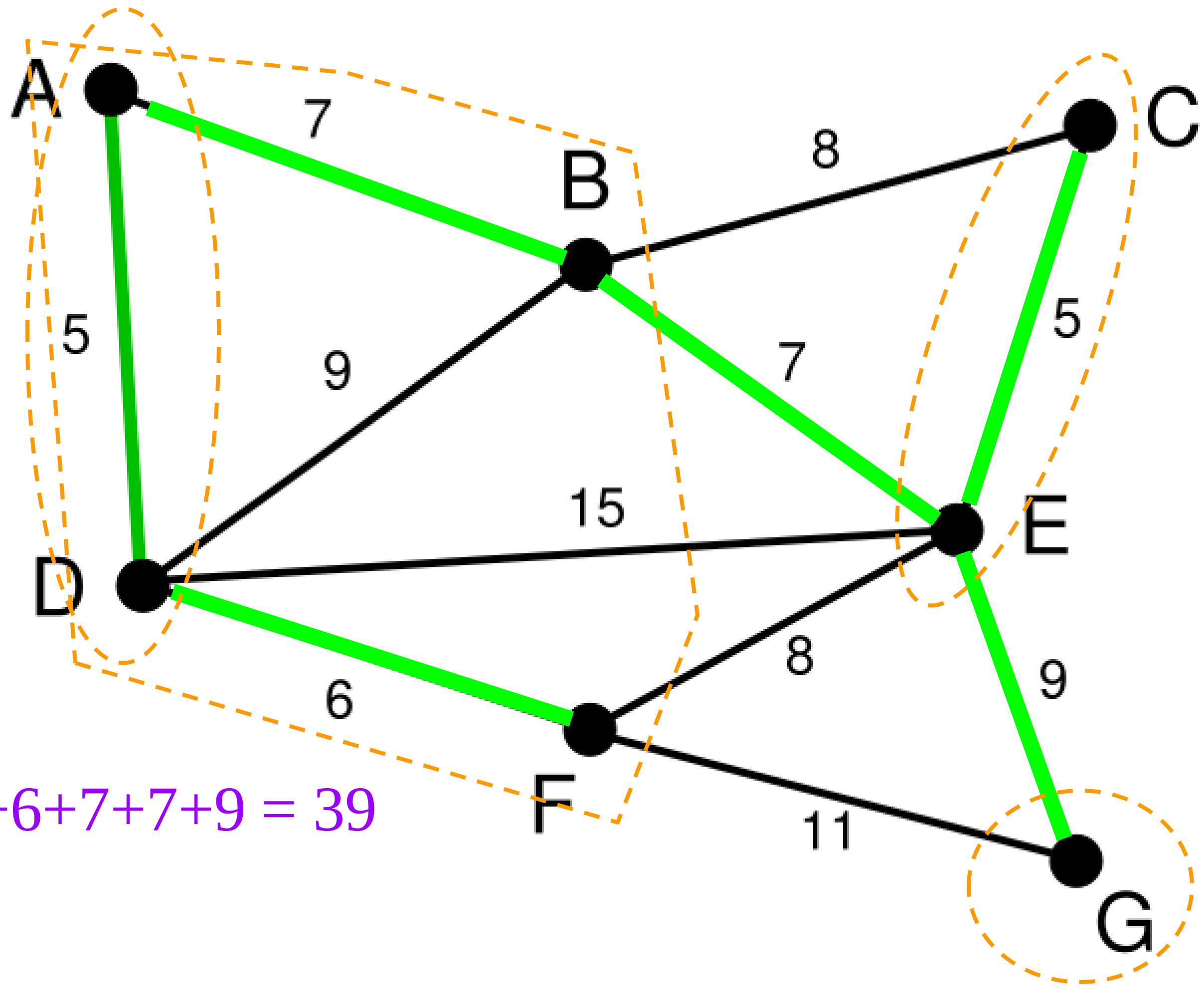
Assign the label min(u,v) to all vertices with these two labels;

Add ***(v,u)*** into MST;

count++;



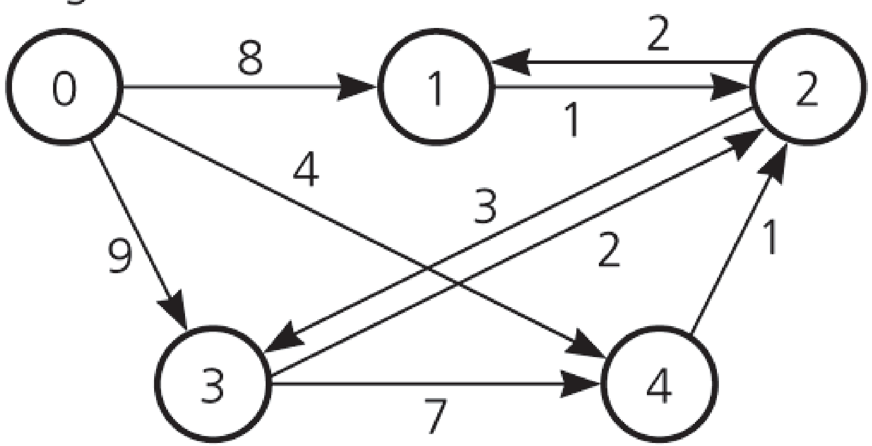
Kruskal's Algorithm



MST: 5+5+6+7+7+9 = 39

Shortest Paths: *Dijkstra's Algorithm*

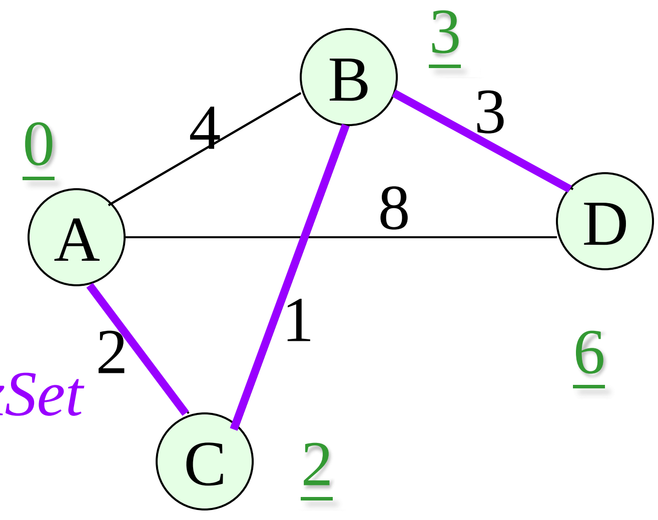
(a) Origin



Step	v	vertexSet	weight [0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4

Single-Source All-Destination Shortest Paths

Data Structures



DijkstraAlgorithm(Vertex v_0)

weight[*0..n*] = {0, ∞, …, ∞};

vertexSet = ∅; $v = v_0$;

do { *Add v into vertexSet;*

for *edge (v,u) where u is not in vertexSet*

weight[*u*] = ***min***{*weight*[*u*],

weight[*v*]+*edgeWeight*[*v,u*]};

cheapest = ∞**;**

for *vertex u not in vertexSet*

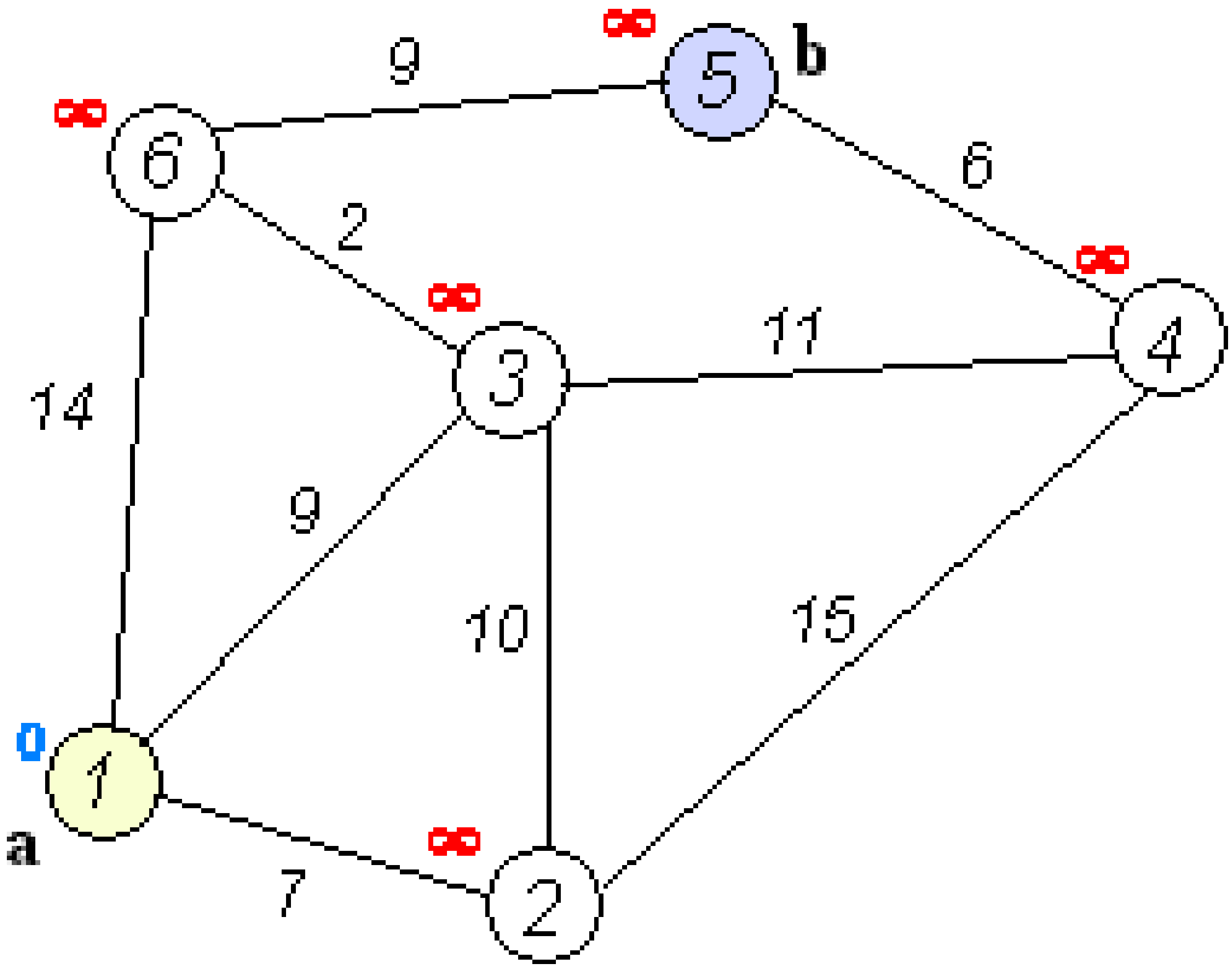
if (*weight*[*u*] < cheapest)

{ *v* = *u*; cheapest = *weight*[*u*];}

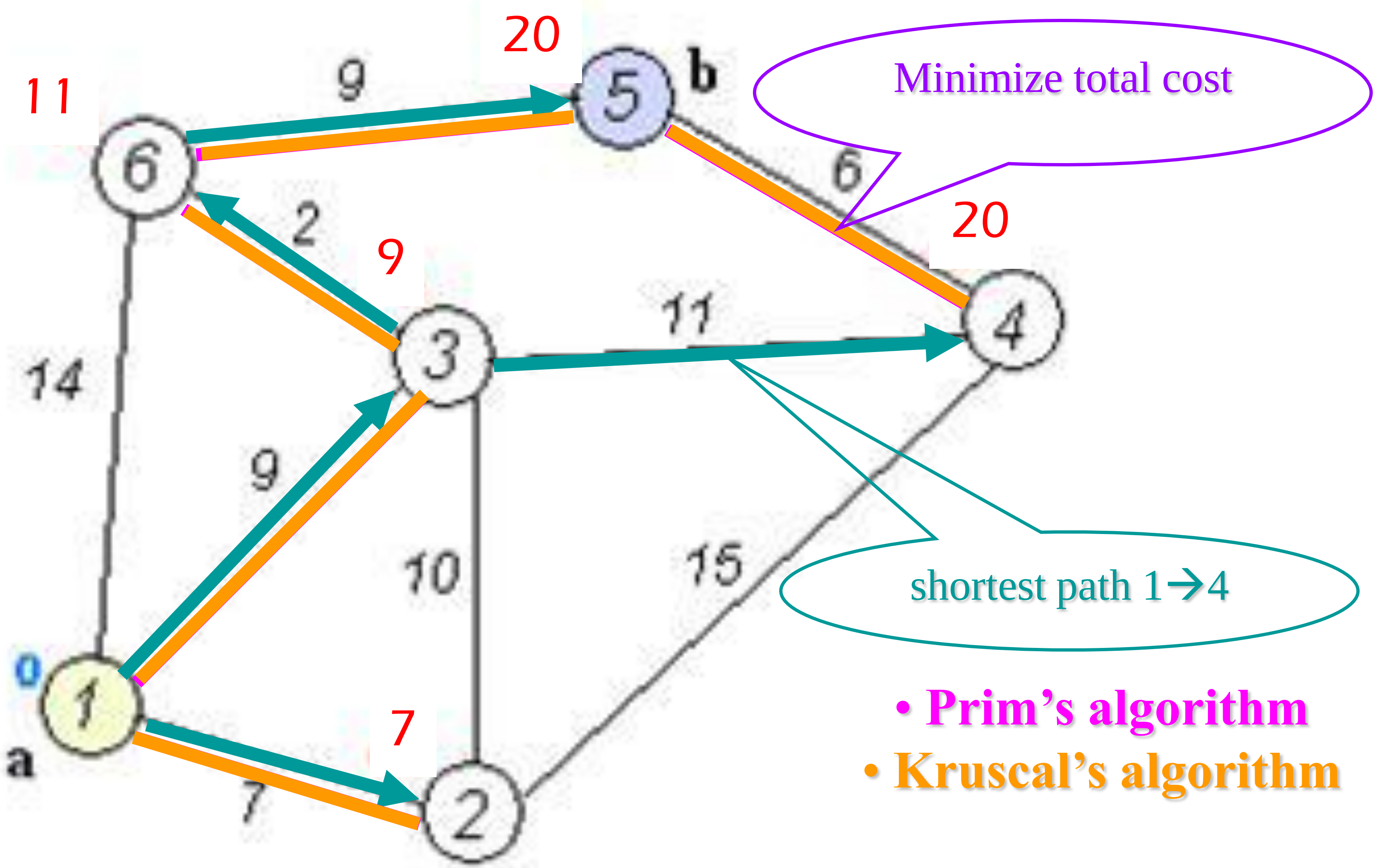
} while (cheapest < ∞);

Shortest Paths: *Demonstration*

Data Structures

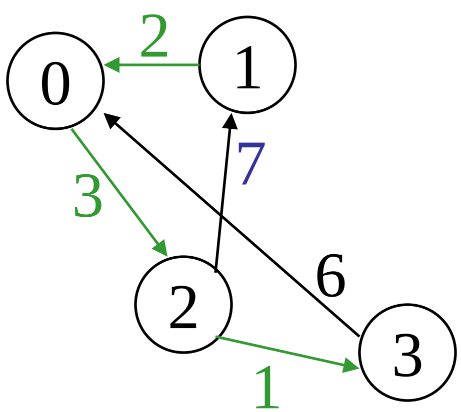


Shortest Path Tree vs. *Minimum Spanning Tree*



- **Prim's algorithm**
- **Kruscal's algorithm**

All-Pairs Shortest Paths



0,1	0→2→1	10
0,2	0→2	3
0,3	0→2→3	4
1,0	1→0	2
1,2	1→0→2	5
1,3	1→0→2→3	6
2,0	2→3→0	7
2,1	2→1	7
2,3	2→3	1
3,0	3→0	6
3,1	3→0→2→1	16
3,2	3→0→2	9

All-Pairs Shortest Paths: *Floyd's Algorithm*

Floyd–Warshall algorithm **[Robert Floyd, 1962][S. Warshall, 1962]**

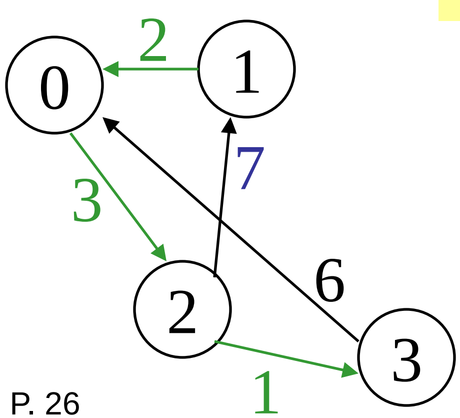
1. Initialize *distance matrix* D^{-1} = *adjacency matrix*;
2. **For** k = 0 to |V|-1

   $D^k \leftarrow D^{k-1}$; *// Add vertex k into vertexSet*

   For *i = 0* to *|V|*-1

   For *j = 0* to *|V|*-1

$D^0[1,2] = \min \{D^{-1}[1,2], D^{-1}[1,0]+D^{-1}[0,2]\}$



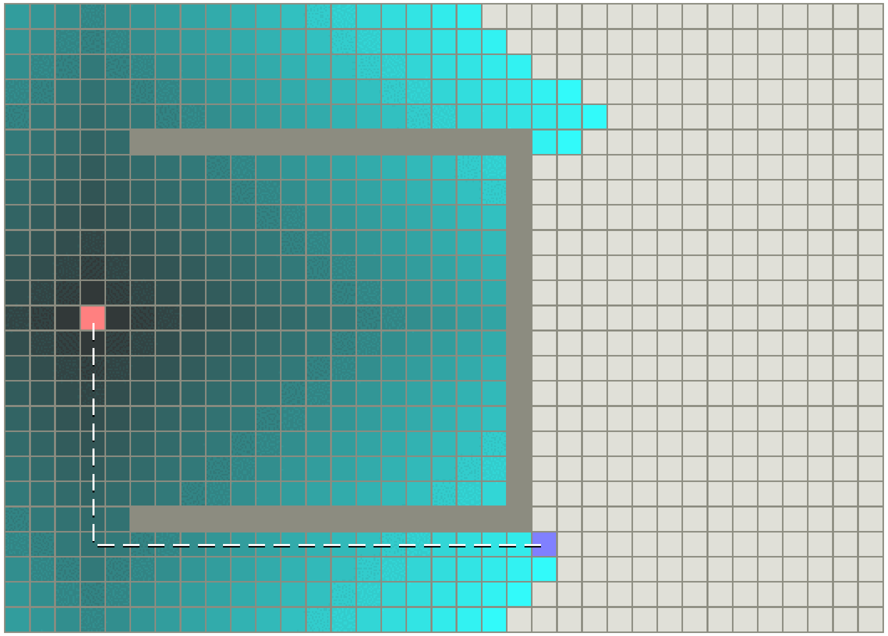
D^{-1}	0	1	2	3
0	0	∞	$\boxed{3}$	∞
1	$\boxed{2}$	0	∞	∞
2	∞	7	0	1
3	6	∞	∞	0

D^0	0	1	2	3
0	0	∞	3	∞
1	2	0	**5**	∞
2	∞	7	0	1
3	6	∞	∞	0

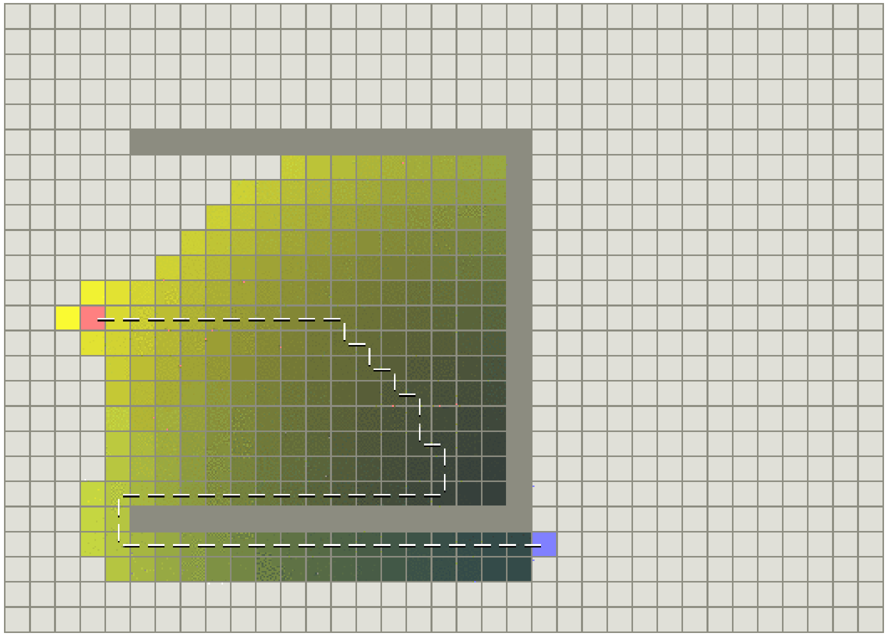
Path Finding: *Comparisons*

Data Structures

- *Dijkstra's algorithm*



- *A* algorithm*



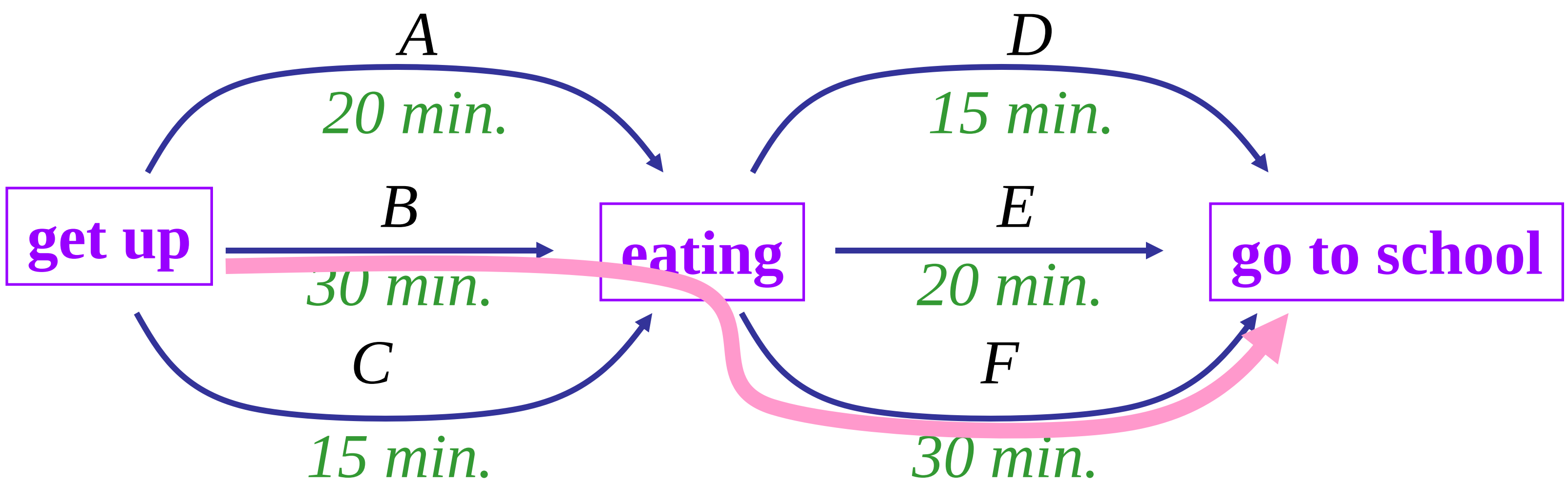
Graph Problems

- Critical Path Analysis
- Maximum Flow Problem
- Other Difficult Problems

Critical Path Analysis

Data Structures

- Earliest time of an activity/event: early(E) = 30
- Latest time of an activity/event: late(E) = 60 – 20 = 40
- **Critical activity**: late(F) = early(F) = 30



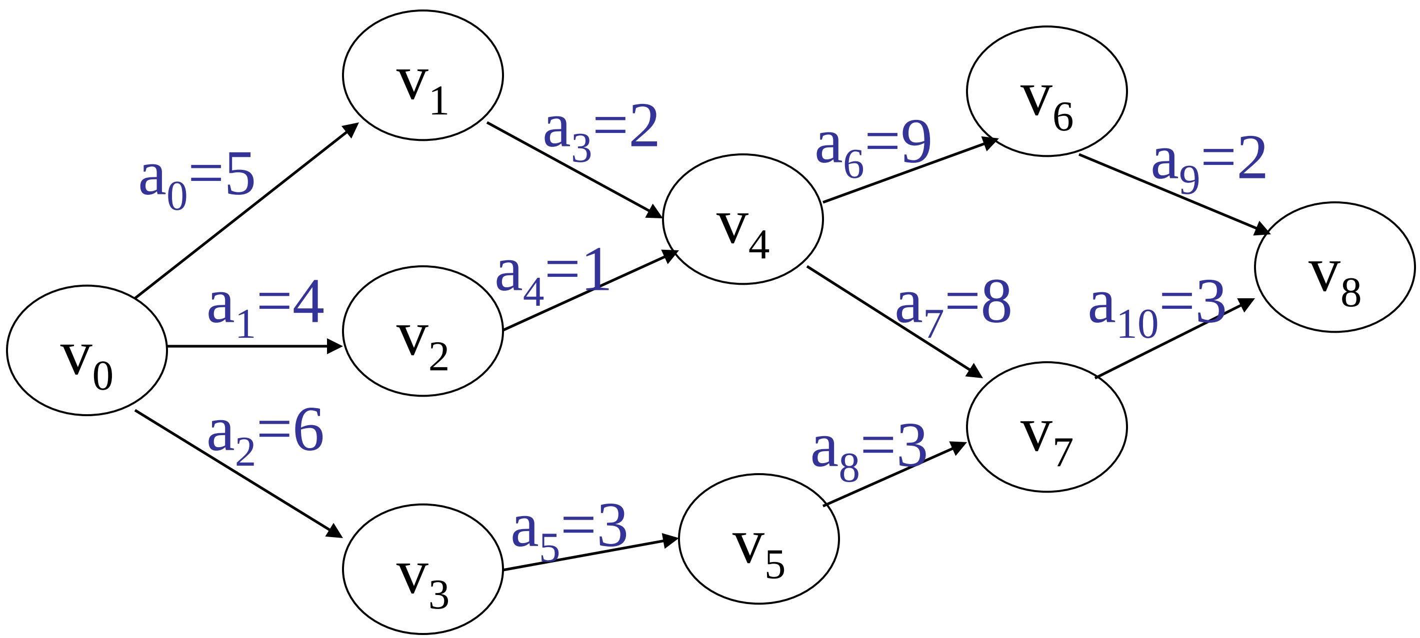
Critical Path Analysis: *Example*

Data Structures

- **Activities: a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 $a_9 a_{10}$**
- **Dependencies among Activities**

  $a_0 \rightarrow a_3$, $a_1 \rightarrow a_4$, $a_2 \rightarrow a_5$, $a_5 \rightarrow a_8$, $a_6 \rightarrow a_9$

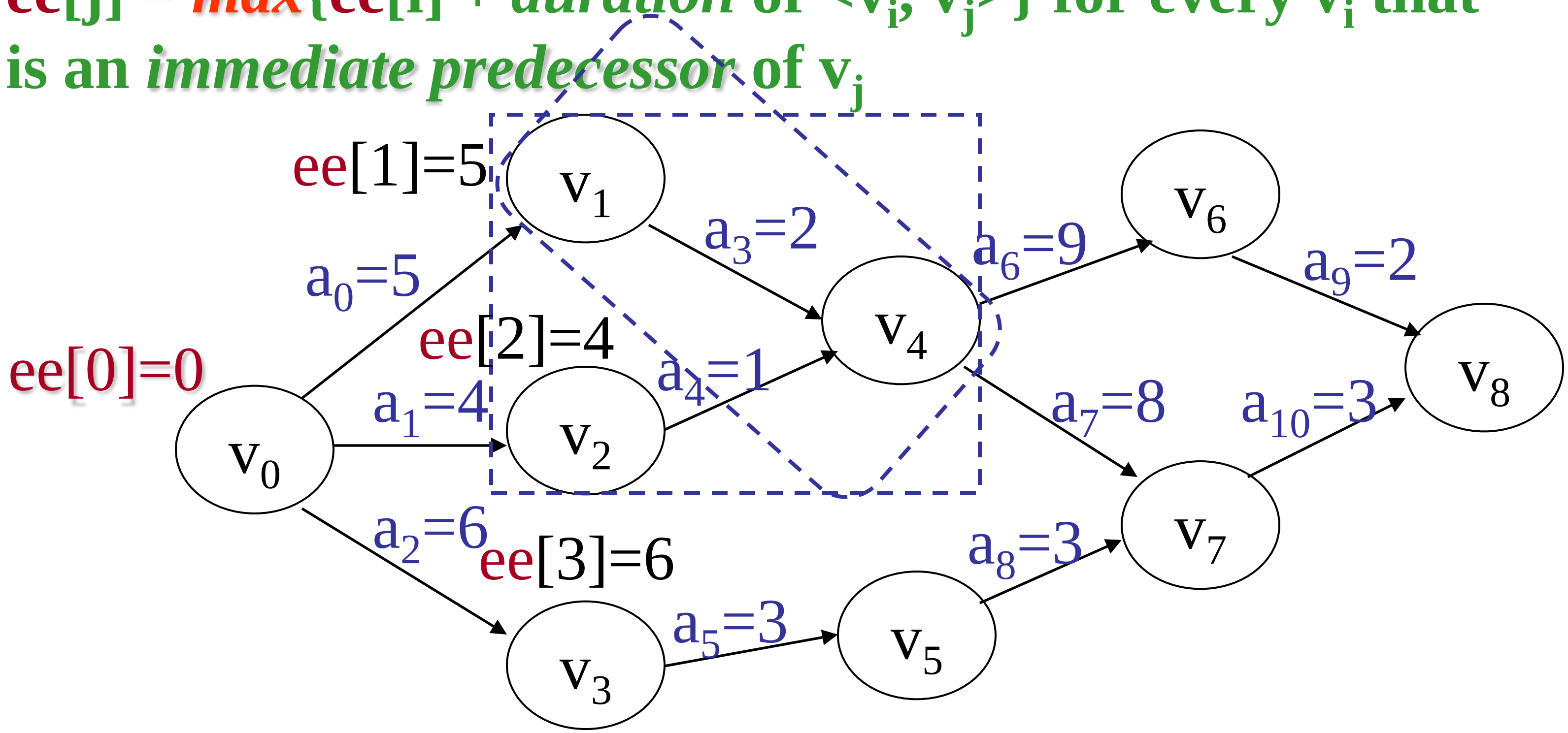
  $a_3\ a_4 \rightarrow a_6\ a_7$, $a_7\ a_8 \rightarrow a_{10}$



Critical Path Analysis: *Forward*

- **earliest time of an activity: ea[0..10]**
  - earlist time of an event: ee[0..8]
    - **ea[x] = ee[i] if a_x is on the edge $<v_i, v_j>$**
    - **ee[j] = *max*{ee[i] + *duration* of $<v_i, v_j>$} for every v_i that is an *immediate predecessor* of v_j**

ea
[0]: 0
[1]: 0
[2]: 0

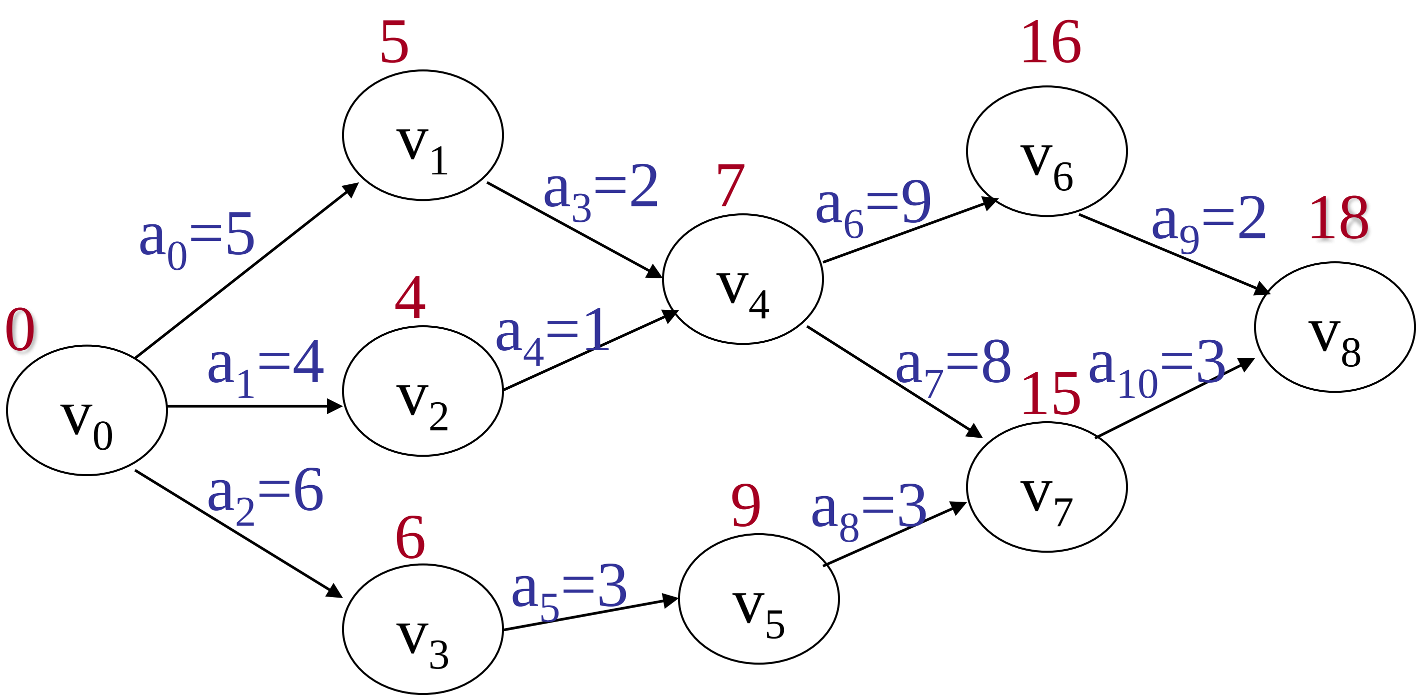


Critical Path Analysis: *Forward*

ea

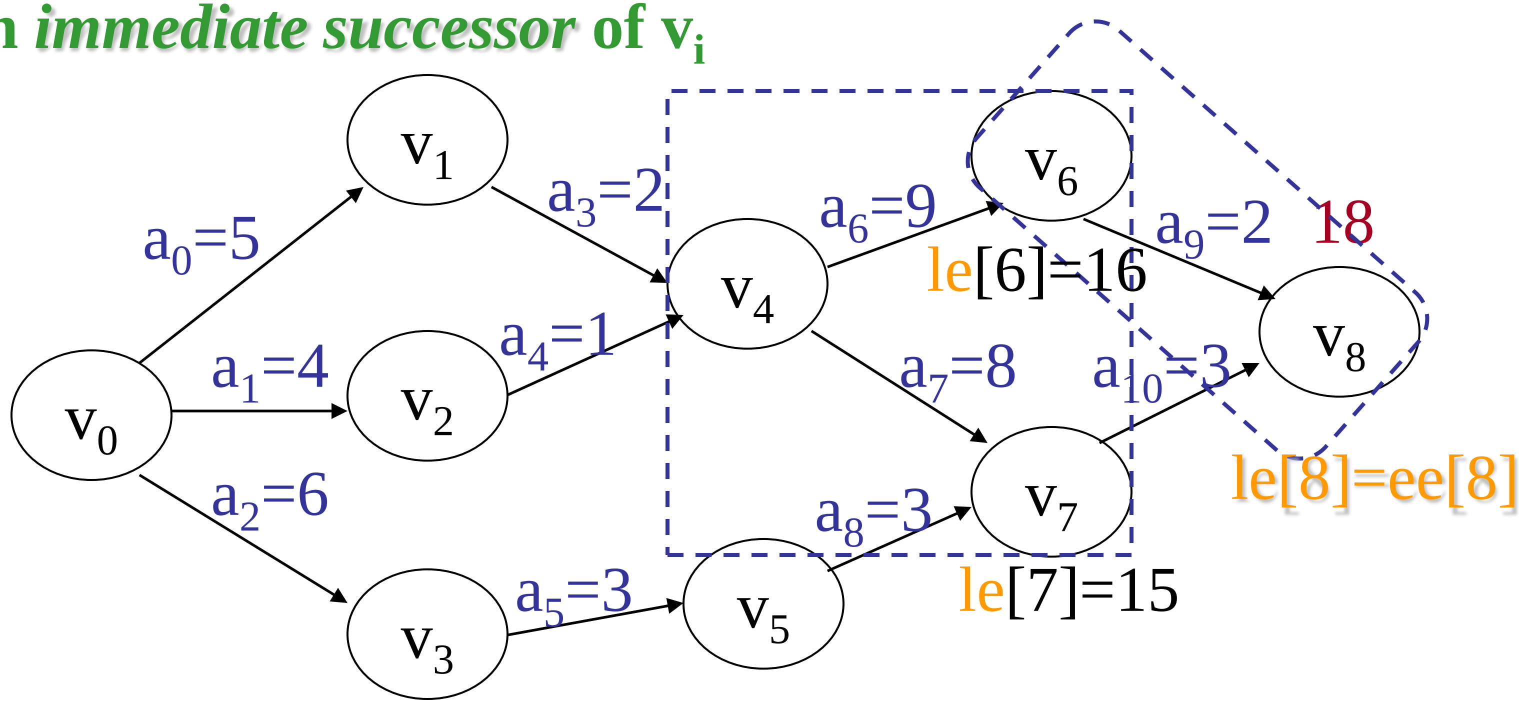
[0]: 0
[1]: 0
[2]: 0
[3]: 5
[4]: 4
[5]: 6
[6]: 7
[7]: 7
[8]: 9
[9]: 16
[10]: 15

- **ea[x] = ee[i] if a_x is on the edge <v_i, v_j>**
- **ee[j] = *max*{ee[i] + *duration* of <v_i, v_j>} for every v_i that is an *immediate predecessor* of v_j**



Critical Path Analysis: *Backward*

- **latest time of an activity: la[0..10]**
  - latest time of an event: le[0..8]
    - **la[x] = le[j] - *duration* of <v_i, v_j>, where a_x is on <v_i, v_j>**
    - **le[i] = *min*{le[j] - *duration* of <v_i, v_j>} for every v_j that is an *immediate successor* of v_i**



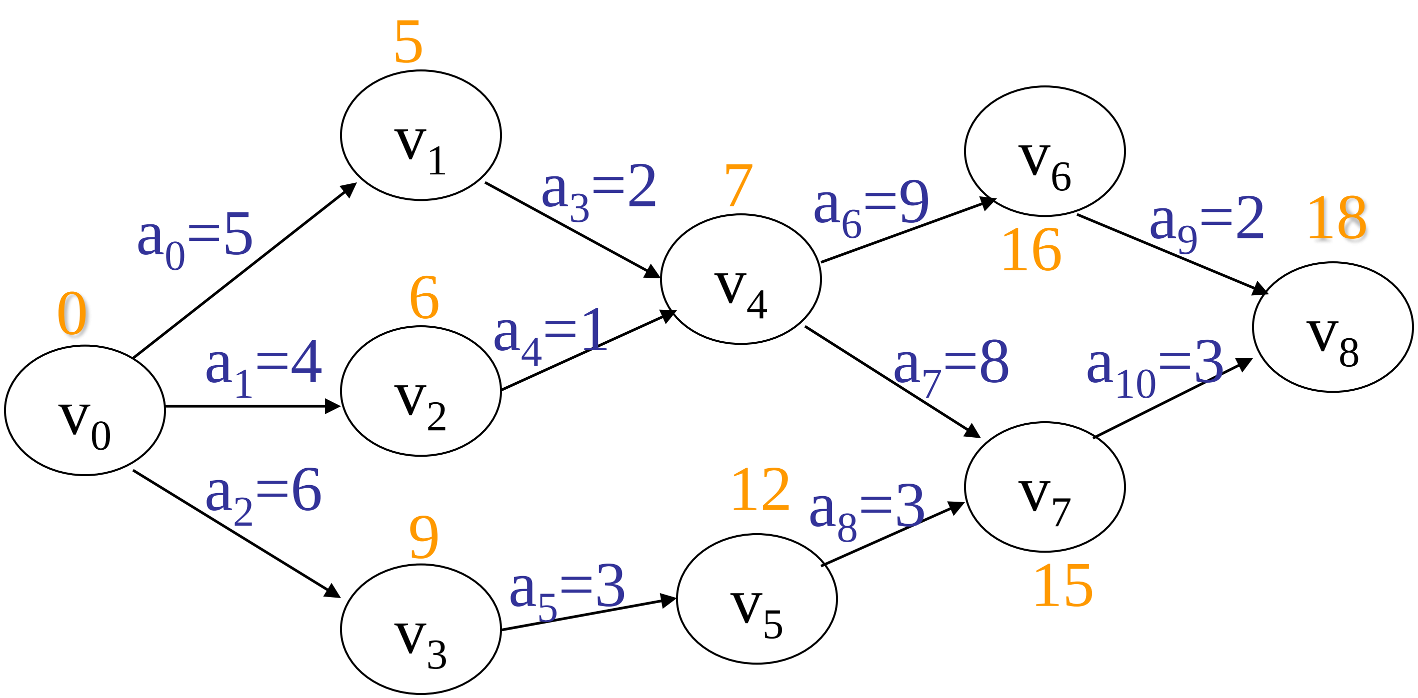
[9]: 16
[10]: 15
la

Critical Path Analysis: *Backward*

Data Structures

- $la[x] = le[j]$ - *duration* of $<v_i, v_j>$, where a_x is on $<v_i, v_j>$
- $le[i] = \textit{min}\{le[j]$ - *duration* of $<v_i, v_j>\}$ for every v_j that is an ***immediate successor*** of v_i

[0]: 0
[1]: 2
[2]: 3
[3]: 5
[4]: 6
[5]: 9
[6]: 7
[7]: 7
[8]: 12
[9]: 16
[10]: 15
la

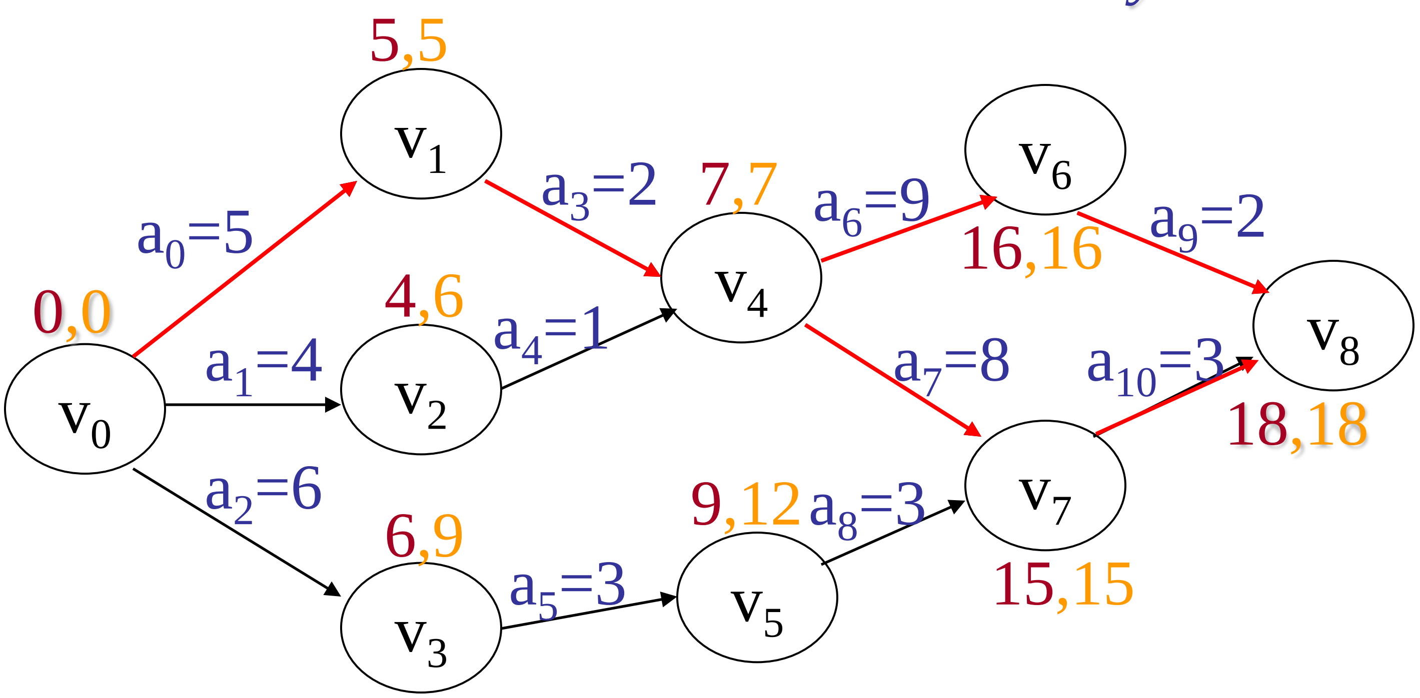


Critical Path Analysis: *Results*

	ea	la	la-ea
[0]:	0	0	**0**
[1]:	0	2	**2**
[2]:	0	3	**3**
[3]:	5	5	**0**
[4]:	4	6	**2**
[5]:	6	9	**3**
[6]:	7	7	**0**
[7]:	7	7	**0**
[8]:	9	12	**3**
[9]:	16	16	**0**
[10]:	15	15	**0**

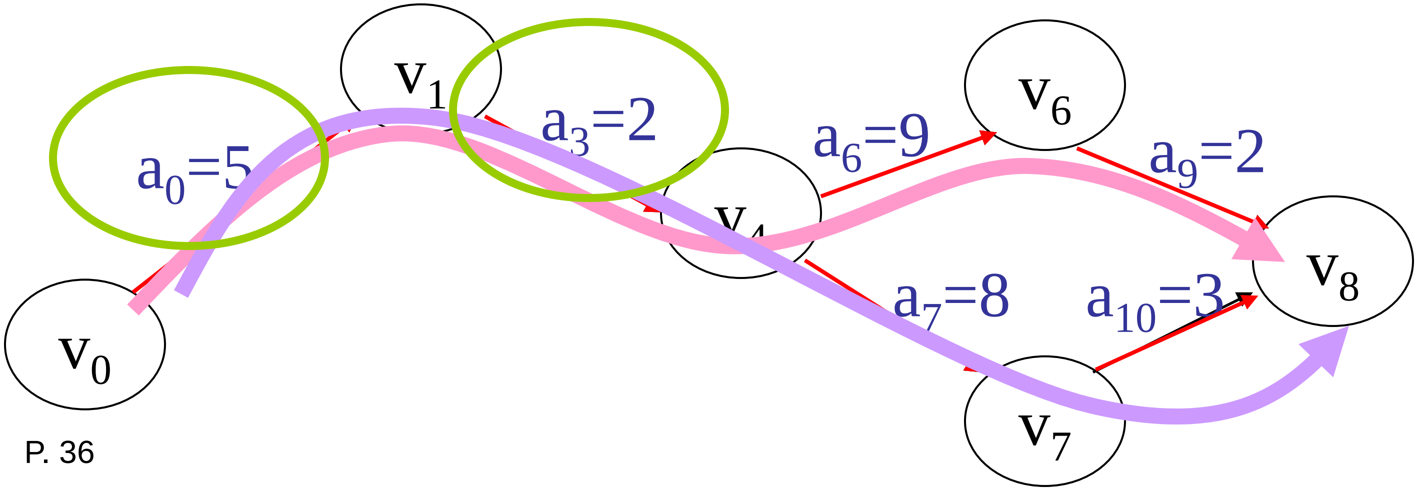
- **la-ea is called (total) *float* or *slack***
  - amount of time that a task can be delayed without causing a delay to project completion time

la-ea==0 means a *critical activity*



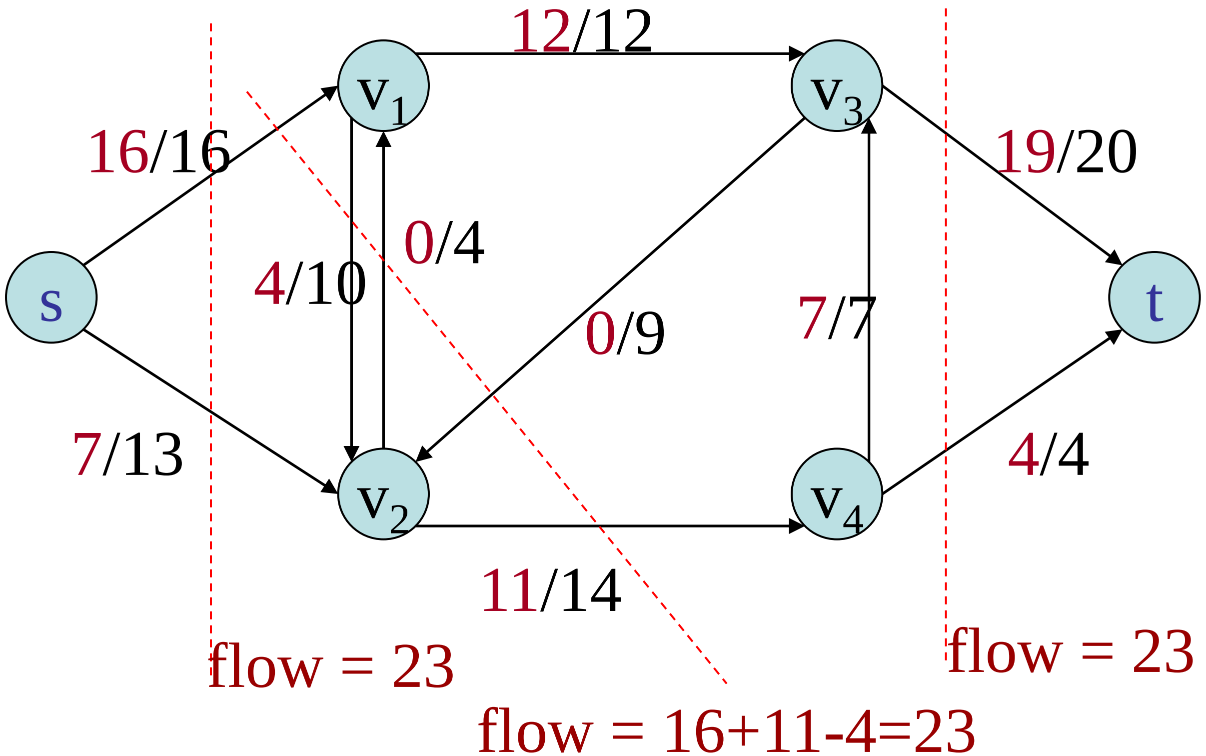
Critical Path Analysis: *Results*

- **Determine Critical Paths**
  - Delete all non-critical activities (*nonzero slack*)
  - Generate all the paths from the start to the end
- **Speed up the activities on all critical paths**
  - *resource* can be concentrated on these activities in an attempt to reduce the project completion time



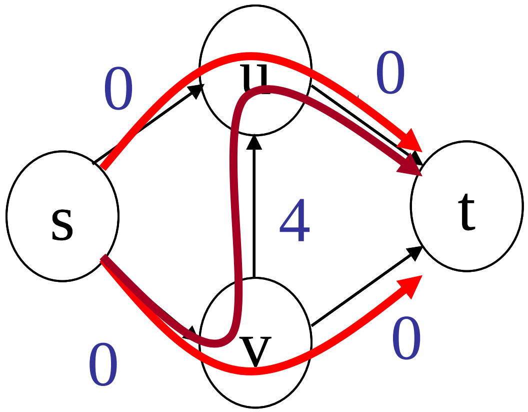
Maximum Flow Problem

- *What is the maximum flow?*



flow f(u,v) / capacity c(u,v)

Residual Graph

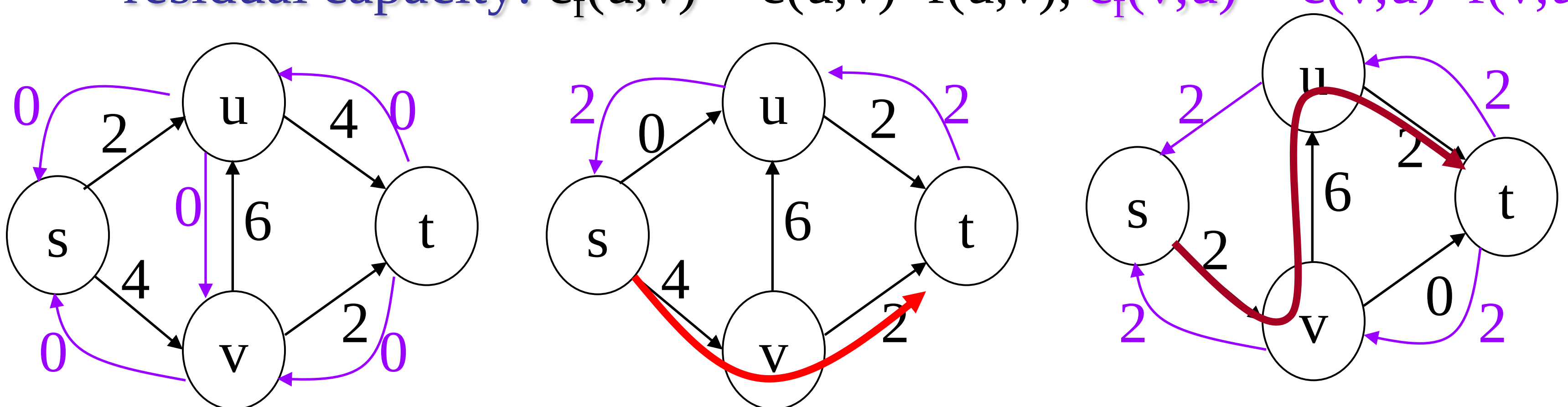


s→u→t: c(s,u) = 2, c(u,t) = 4 ➔ flow(u,v) = 2

s→v→t: c(s,v) = 4, c(v,t) = 2 ➔ flow(u,v) = 2

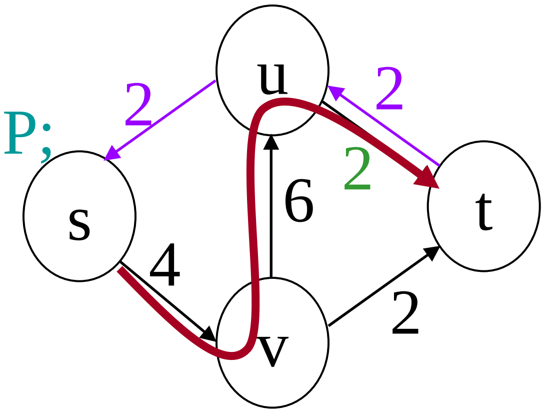
s→v→u→t: flow(u,v) = *min*{2,6,2} = 2

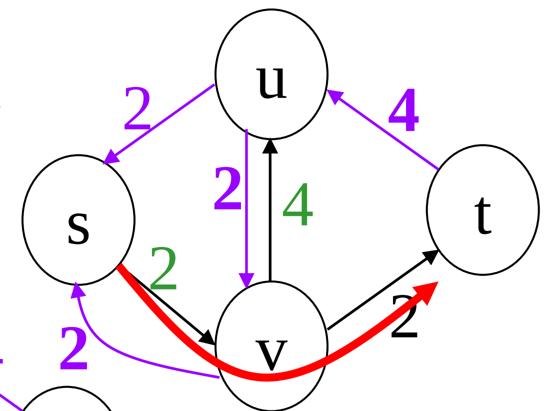
- **Residual graph**
  - residual capacity: $c_f(u,v) = c(u,v)–f(u,v)$, $c_f(v,u) = c(v,u)–f(v,u)$



Ford-Fulkerson algorithm

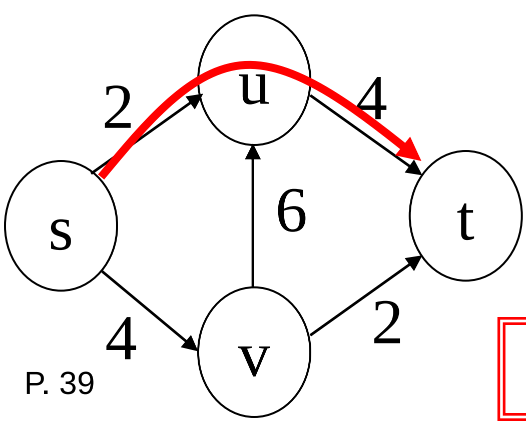
1. Initialize $c_f(u,v)$ for every edge;
2. Find a path P from s to t ∋ $c_f(u,v)>0\ \forall(u,v)\in P$;
3. $c_f(P) = \mathit{min}\{c_f(u,v): (u,v)\in P\}$;
4. For each edge $(u,v)\in P$
   - **$c_f(u,v) = c_f(u,v) - c_f(P)$;**
   - **$c_f(v,u) = c_f(v,u) + c_f(P)$;**



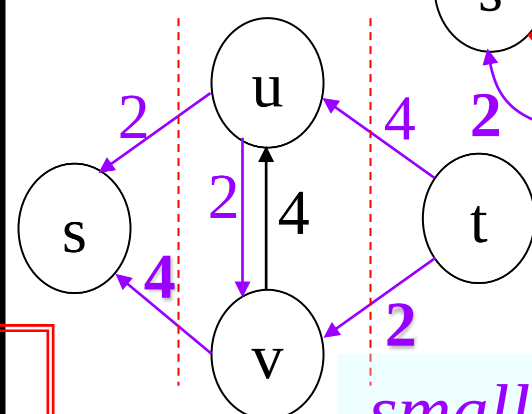


maximum flow

= 4 + 2 = 6



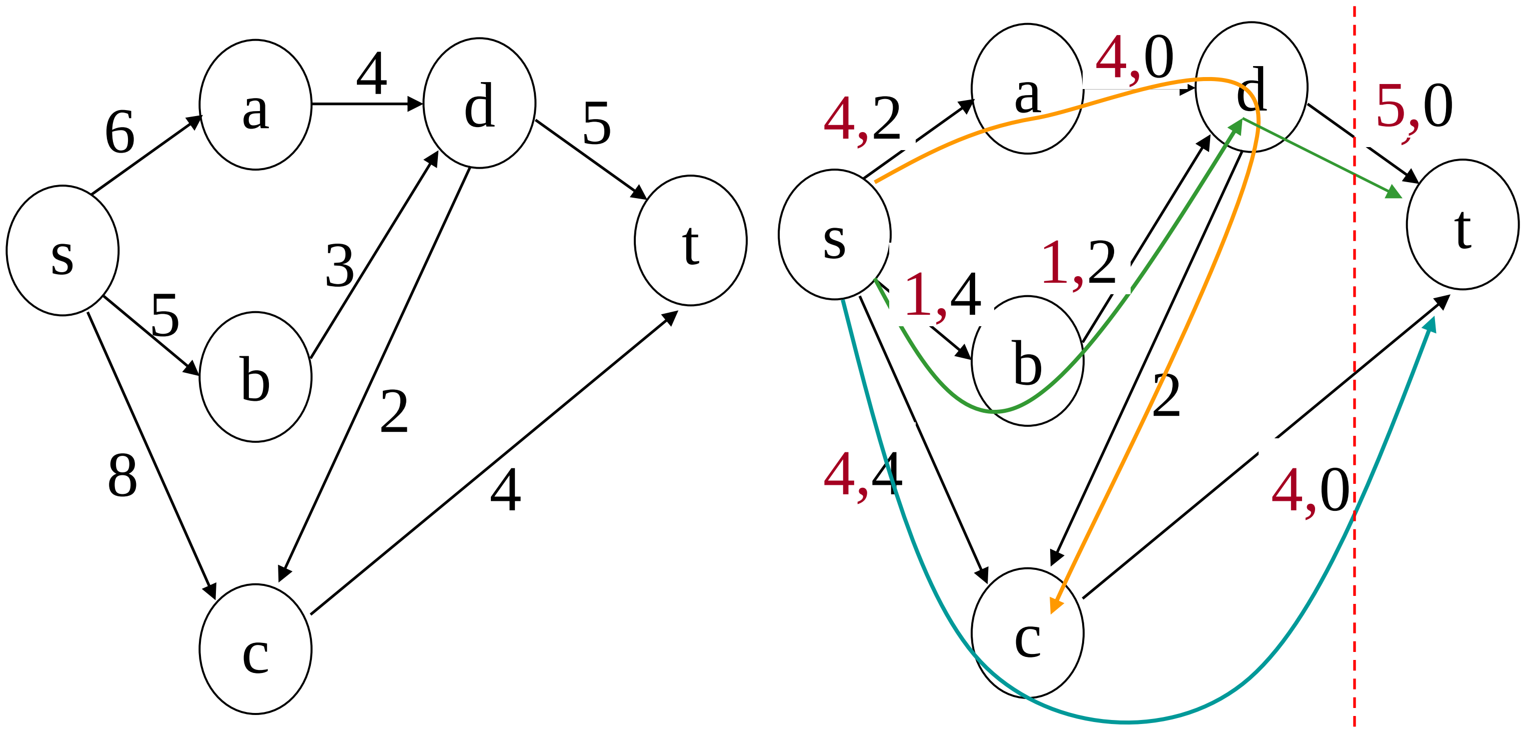
c_f	s	u	v	t
s	0	0	0	0
u	2	0	2	0
v	4	4	0	0
t	0	4	2	0



smallest label first

Edmonds-Karp algorithm

Data Structures



maximum capacity first

Bi-connected Graph: *Definitions*

- **Finding the articulation points**
  - DFS-tree based algorithm