CSCI2100 Data Structures
Graph Algorithms

Irwin King

king@cse.cuhk.edu.hk
http://www.cse.cuhk.edu.hk/~king

Department of Computer Science & Engineering
The Chinese University of Hong Kong

Introduction

- Show several real-life problems using graphs.
- Give algorithms to solve several common graph problems.
- Show how the proper choice of data structures can drastically reduce the running time of these algorithms.
- See how depth-first search can be used to solve several seemingly nontrivial problems in linear time.

Examples

- Algorithms to find the minimum path between two nodes
- Algorithms to find whether a graph contains another graph
- Algorithms to find the maximum flow between two nodes

Definitions

- Graphs is an important mathematical structure.
- A graph $G = (V, E)$ consists of a set of **vertices** (or **nodes**), V, and a set of **edges**, E.
- Each edge is a pair (v,w), where $v,w \in V$. Edges are sometimes referred to as **arcs**.
- If $e = (v,w)$ is an edge with vertices v and w, the v and w are said to lie on e, and e is said to be **incident** with v and w.

Definitions

- If the pairs are unordered, then *G* is called an **undirected graph**; if the pairs are ordered, the *G* is called a **directed graph**.
- The term **directed graph** is often shortened to **digraph**, and the unqualified term **graph** usually means **undirected graph**.

Definition

- Vertex w is **adjacent** to v if and only if (v,w) ∈ E.
- In an undirected graph with edge (v,w), and hence (w,v), w is adjacent to v and v is adjacent to w.
- Sometimes an edge has a third component, known as either a **weight** or a **cost**.

Path Definition

- A path in a graph is a sequence of vertices w_1, $w_2, w_3, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$.
- The length of such a path is the number of edges on the path, which is equal to $n - 1$.
- We allow a path from a vertex to itself; if this path contains no edges, then the path length is 0.

Path Definition

- If the graph contains an edge (v,v) from a vertex to itself, then the path v, v is sometimes referred to as a **loop**.
- A **simple** path is a path such that all vertices are distinct, except that the first and last could be the same.

Cycle Definition

- A **cycle** in a directed graph is a path of length at least 1 such that $w_1 = w_n$; this cycle is simple if the path is simple.
- For undirected graphs, we require that the edges be distinct.
- Why?

  The logic of these requirements is that the path u, v, u in an undirected graph should not be considered a cycle, because (u, v) and (v, u) are the same edge.

Cycle Definition

- In a directed graph, these are different edges, so it makes sense to call this a cycle.

- A directed graph is **acyclic** if it has no cycles.

- A directed acyclic graph is sometimes referred to by its abbreviation, **DAG**.

Connectedness Definition

- An undirected graph is **connected** if there is a path from every vertex to every other vertex.
- A directed graph with this property is called **strongly connected**.
- If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be **weakly connected**.
- A **complete** graph is a graph in which there is an edge between every pair of vertices.

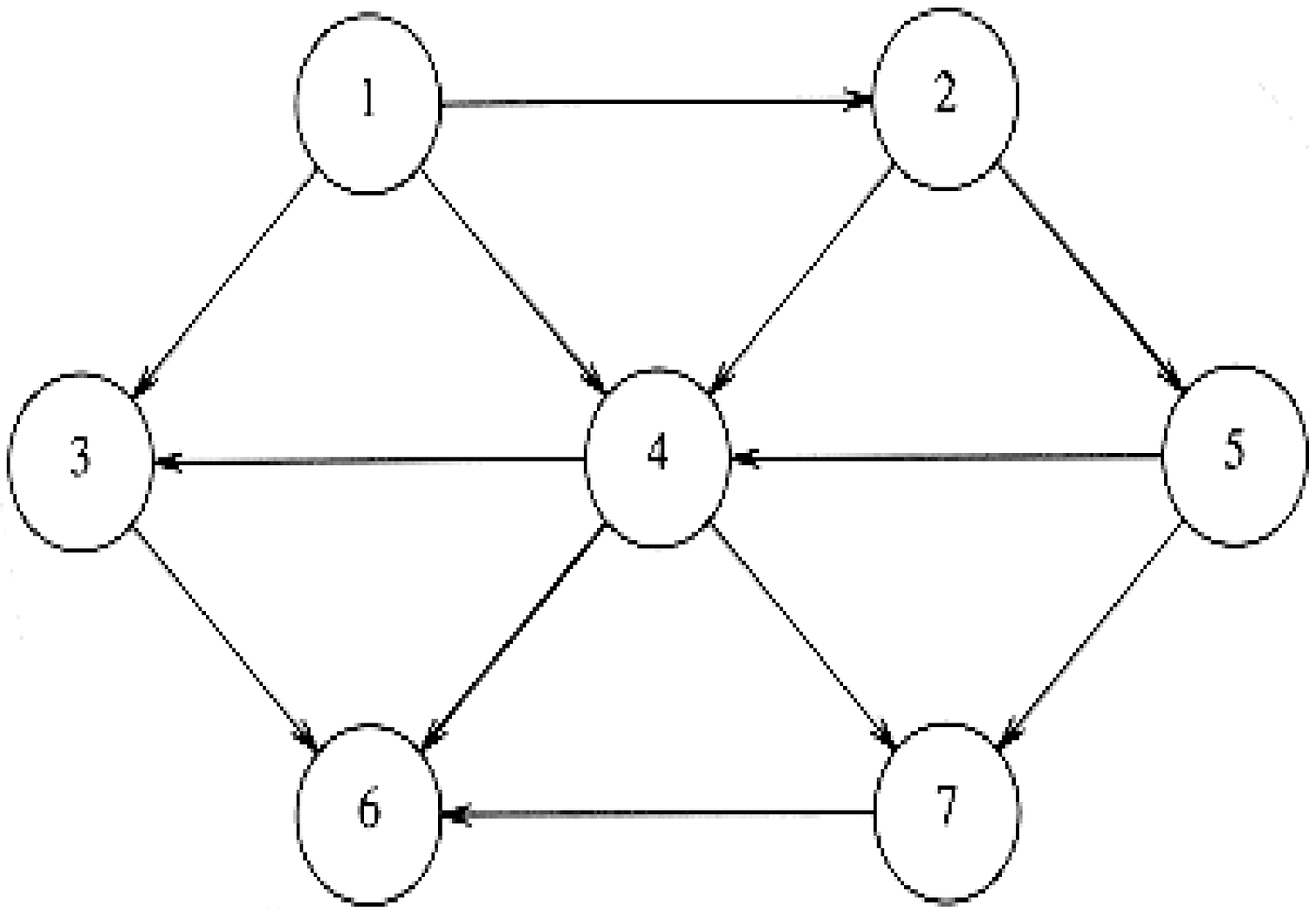
Airport Example

- Each airport is a **vertex**, and two vertices are connected by an **edge** if there is a nonstop flight from the airports that are represented by the vertices.
- The edge could have a **weight**, representing the time, distance, or cost of the flight.
- Such a graph is directed, since it might take longer or cost more to fly in different directions.

Example

- We would probably like to make sure that the airport system is **strongly connected**, so that it is always possible to fly from any airport to any other airport.
- We might also like to quickly determine the **best flight** between any two airports.
- "Best" could mean the path with the fewest number of edges or could be taken with respect to one, or all, of the weight measures.

Example



Implementation

- Use a two-dimensional array. This is known as an **adjacency matrix** representation.
- For each edge (u, v), we set a[u][v] = 1; otherwise the entry in the array is 0.
- If the edge has a weight associated with it, then we can set a[u][v] equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.

Example

- For instance, if we were looking for the cheapest airplane route, we could represent nonexistent flights with a cost of ∞.

- If we were looking, for some strange reason, for the most expensive airplane route, we could use - ∞ (or perhaps 0) to represent nonexistent edges.

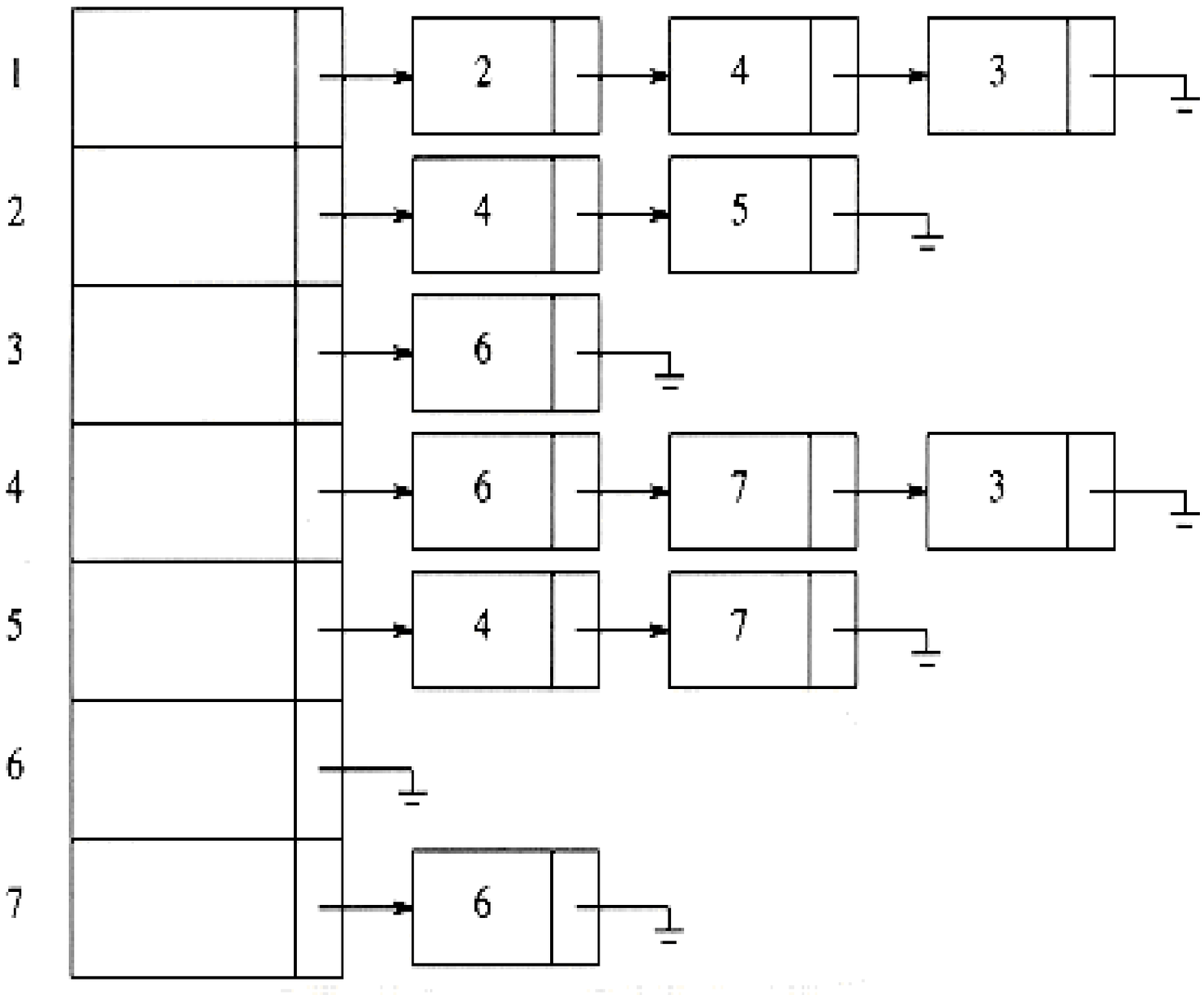
Notes

- The space requirement is $\Theta(|V|^2)$. This is unacceptable if the graph does not have very many edges.
- An adjacency matrix is an appropriate representation if the graph is dense: $|E| = \Theta(|V|^2)$.

Adjacency List

- If the graph is not dense, in other words, if the graph is **sparse**, a better solution is an **adjacency list representation**.
- For each vertex, we keep a list of all adjacent vertices.
- The space requirement is then $O(|E| + |V|)$.
- If the edges have weights, then this additional information is also stored in the cells.

Example



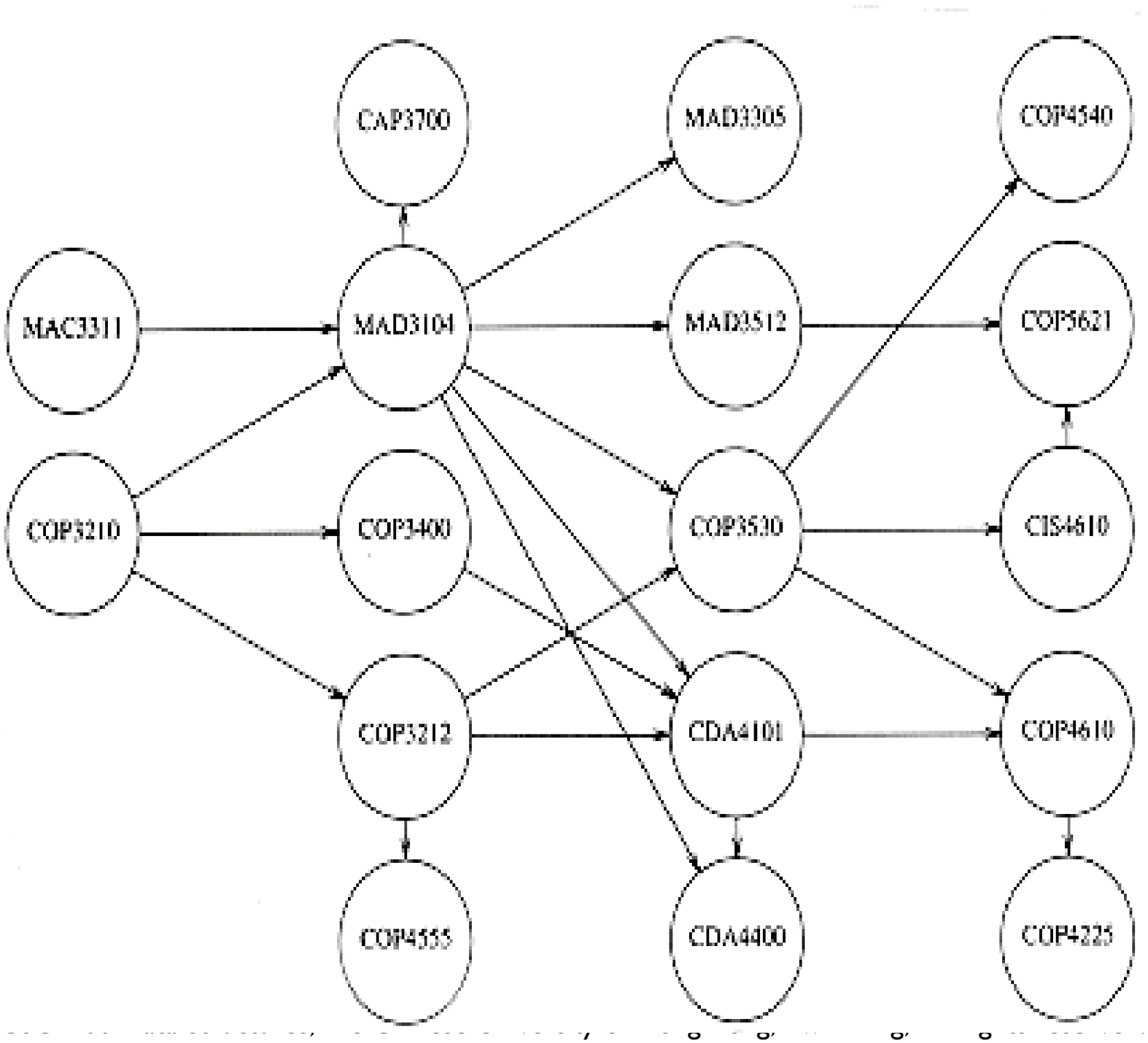
Topological

- A **topological sort** is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.

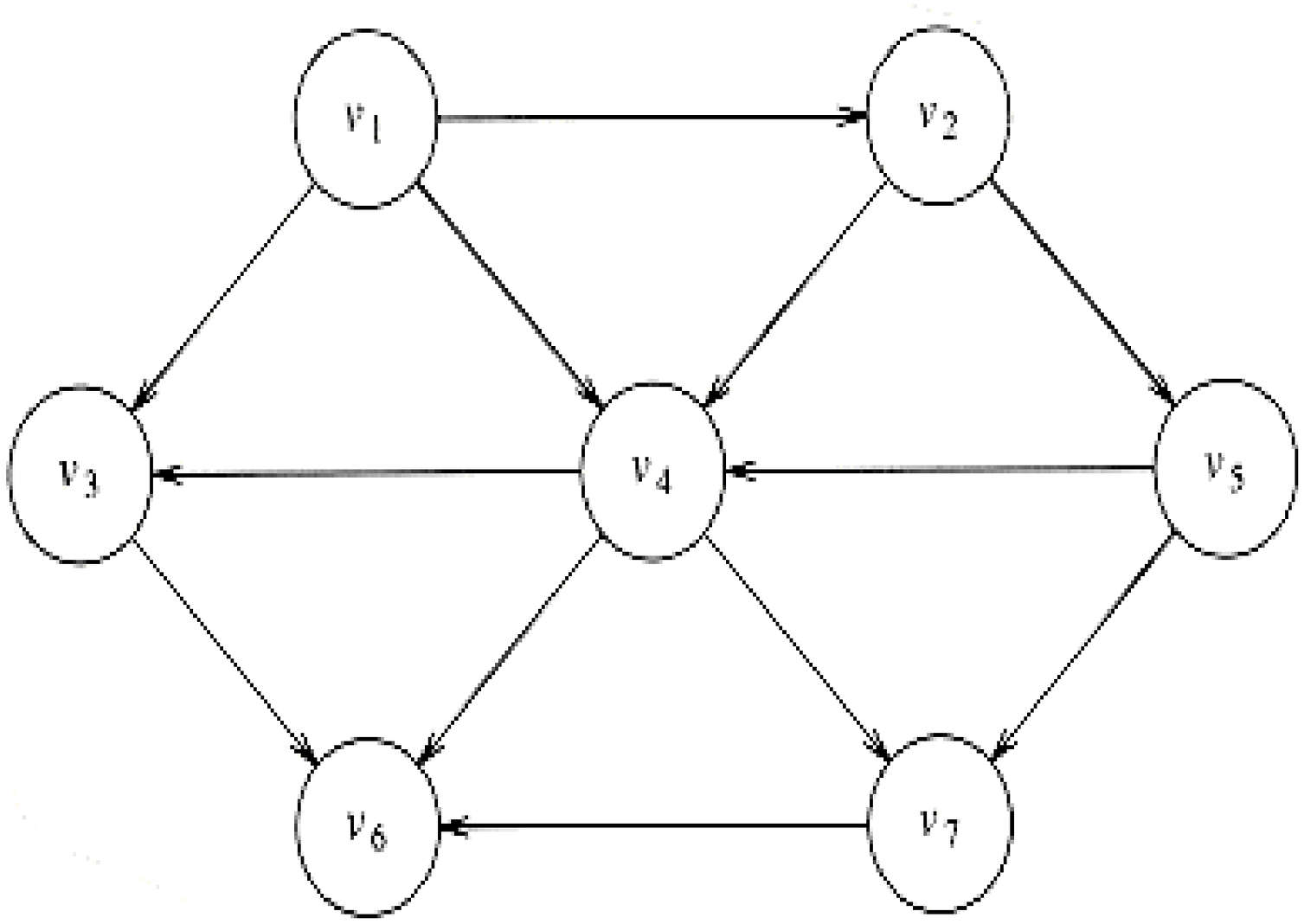
- A directed edge (v,w) indicates that course v must be completed before course w may be attempted.

- A **topological ordering** of these courses is any course sequence that does not violate the prerequisite requirement.

Example


CAP3700
MAD3305
COP4540
MAC3311
MAD3104
MAD3512
COP5621
COP3210
COP3400
COP3530
CIS4610
COP3212
CDA4101
COP4610
COP4555
CDA4400
COP4225

Example



Notes

- Is topological ordering possible with a cyclic graph?
  - No, since for two vertices v and w on the cycle, v precedes w and w precedes v.
- Is the ordering unique?
  - It is not necessarily unique; any legal ordering will do. In the next graph, v_1, v_2, v_5, v_4, v_3, v_7, v_6 and v_1, v_2, v_5, v_4, v_7, v_3, v_6 are both topological orderings.

How to find the topological ordering?

- Define the **indegree** of a vertex v as the number of edges (u,v). We compute the indegrees of all vertices in the graph.
- Find any vertex with no incoming edges (or the indegree is 0).
- We can then print this vertex, and remove it, along with its edges, from the graph.
- Then we apply this same strategy to the rest of the graph.

Topological Sort Pseudocode

```
topsort( graph G )
{
     unsigned int counter;
     vertex v, w;
for(counter=0; counter<NUM_VERTEX; counter++){
  v = find_new_vertex_of_indegree_zero( );
  if( v = NOT_A_VERTEX ) {
    error("Graph has a cycle"); break;}
  top_num[v] = counter;
  for each w adjacent to v
  indegree[w]--;}}
```


Time Complexity

- How long will the algorithm take to find the topological ordering of a graph?
- It is a simple sequential scan of the indegree array, each call to it takes $O(|V|)$ time.
- Since there are $|V|$ such calls, the running time of the algorithm is $O(|V|^2)$.

Topological Sort Example

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v1	0	0	0	0	0	0	0
v2	1	0	0	0	0	0	0
v3	2	1	1	1	0	0	0
v4	3	2	1	0	0	0	0
v5	1	1	0	0	0	0	0
v6	3	3	3	3	2	1	0
v7	2	2	2	1	0	0	0
enqueue	v1	v2	v5	v4	v3	v7	v6
dequeue	v1	v2	v5	v4	v3	v7	v6

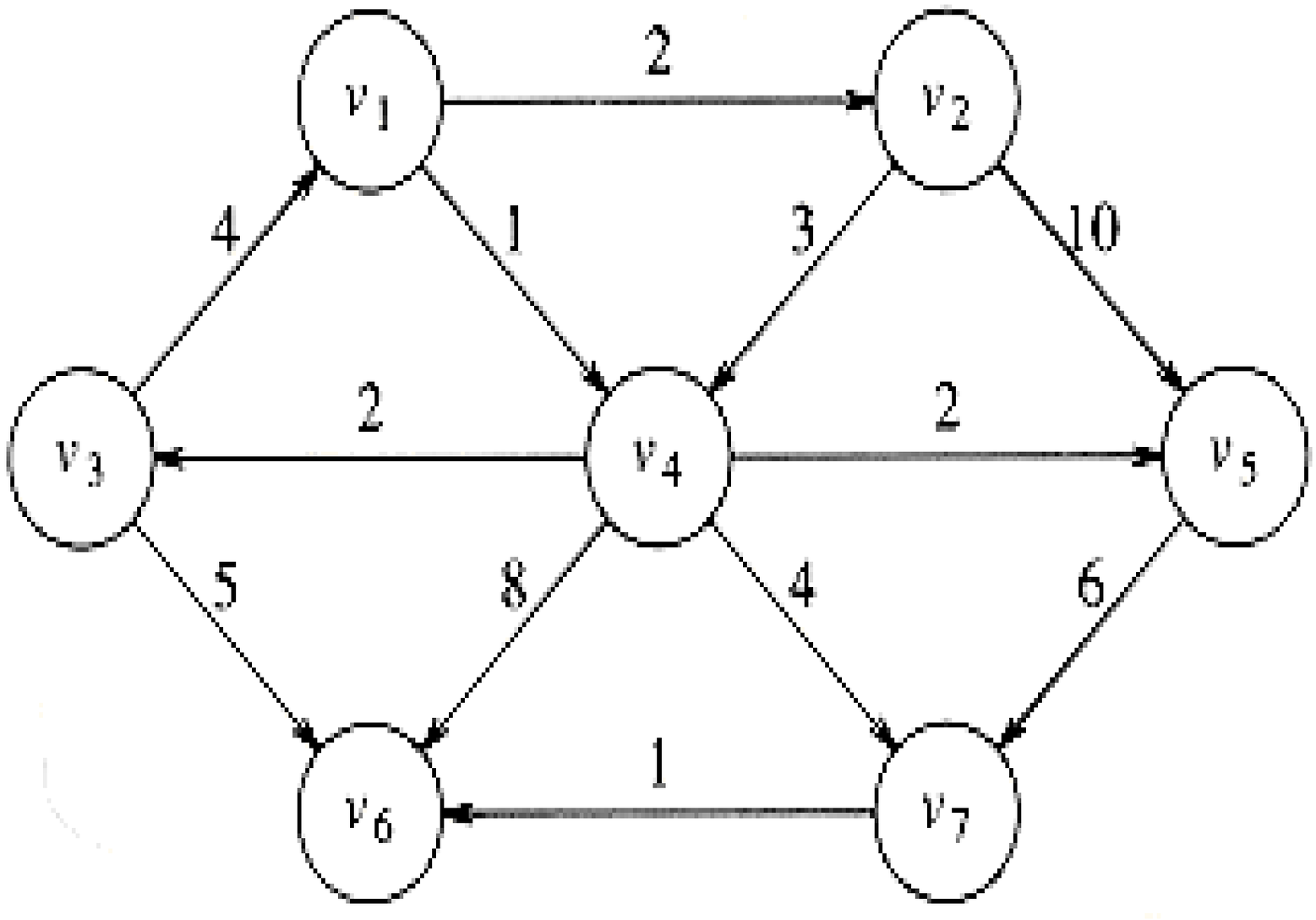
Shortest Path Algorithms

- The input is a weighted graph: associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the arc.
- The cost of a path $v_1 v_2 \ldots v_n$ is $\sum_{i=1}^{n-1} c_{i,i+1}$.
- This is referred to as the weighted path length.
- The unweighted path length is merely the number of edges on the path, namely, $n - 1$.

Single-Source Shortest-Path Problem

- Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, **s**, find the shortest weighted path from **s** to every other vertex in G.

Example

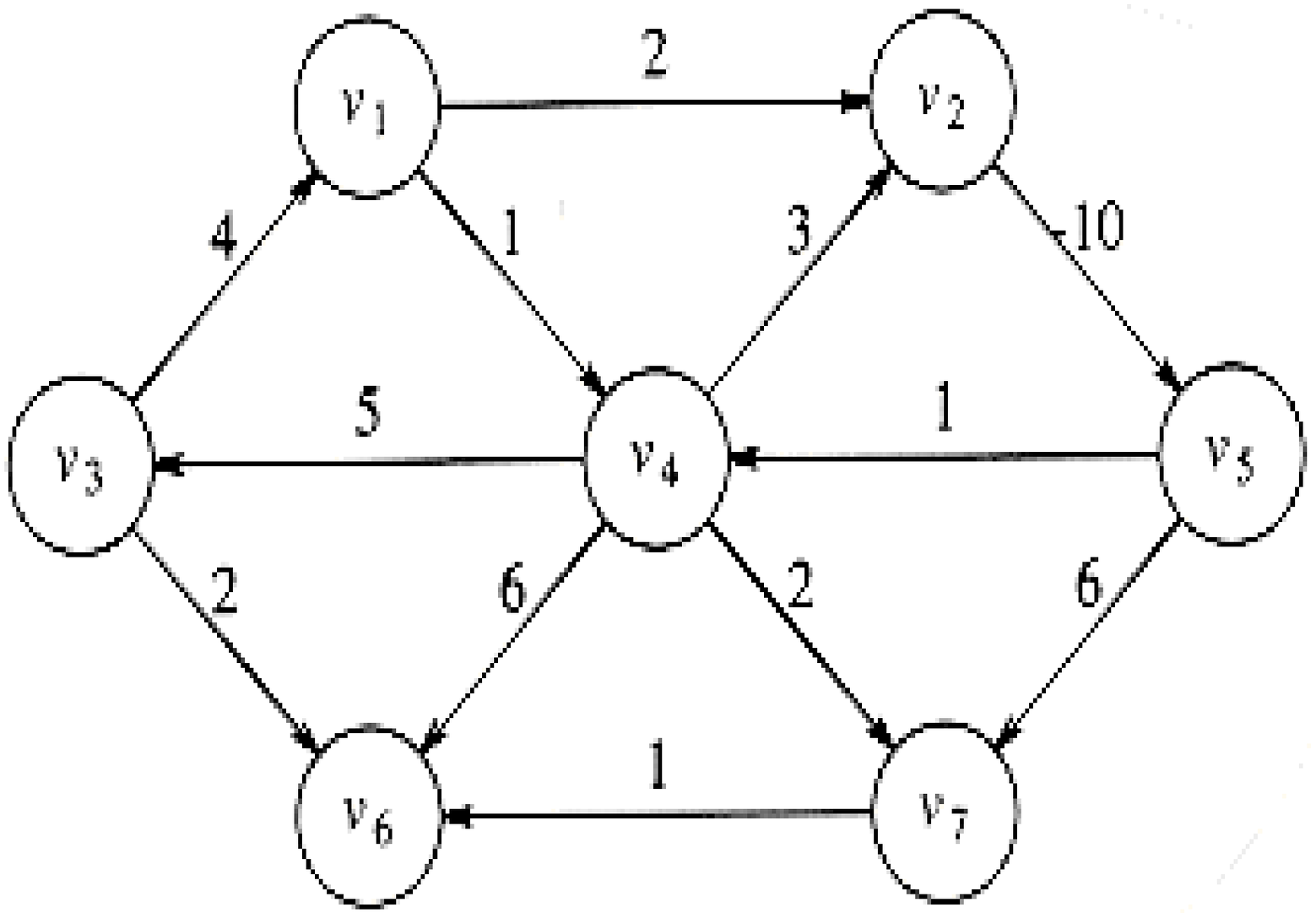


Example

- From the previous graph, the shortest weighted path from v_1 to v_6 has a cost of 6 and goes from v_1 to v_4 to v_7 to v_6.

- The shortest unweighted path between these vertices is 2.

What if the cost is negative?



Problem

- The path from v_5 to v_4 has cost 1, but a shorter path exists by following the loop v_5, v_4, v_2, v_5, v_4, which has cost -5.

- The shortest path between v_5 and v_4 is undefined.

- This loop is known as a **negative-cost cycle**; when one is present in the graph, the shortest paths are not defined.

Notes

- Currently there are no algorithms in which finding the path from **s** to one vertex is any faster (by more than a constant factor) than finding the path from **s** to all vertices.
- The intermediate nodes in a shortest path must also be the shortest path node from **s**.
- We will examine algorithms to solve four versions of this problem.
  - The unweighted shortest-path problem and show how to solve it in $O(|E| + |V|)$.

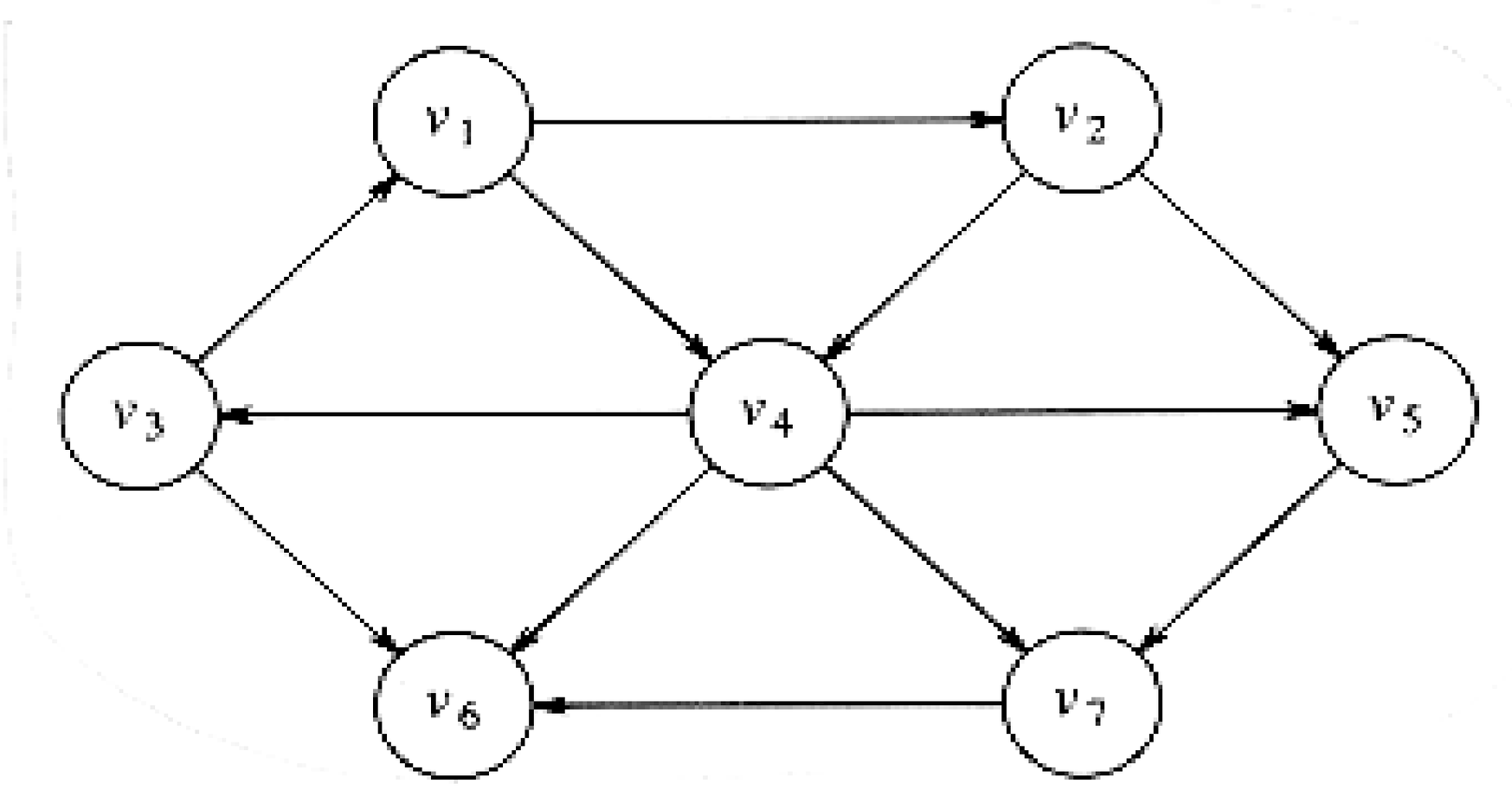
Notes

- The weighted shortest-path problem if we assume that there are no negative edges.
- The running time for this algorithm is $O(|E| \log |V|)$ when implemented with reasonable data structures.
- If the graph has negative edges, we will provide a simple solution, which unfortunately has a poor time bound of $O(|E| \cdot |V|)$.
- Finally, we will solve the weighted problem for the special case of acyclic graphs in linear time.

Unweighted Shortest Paths

- Using some vertex, `s`, which is an input parameter, we would like to find the shortest path from `s` to all other vertices.
- There are no weights on the edges.
- This is a special case of the weighted shortest-path problem, since we could assign all edges a weight of 1.

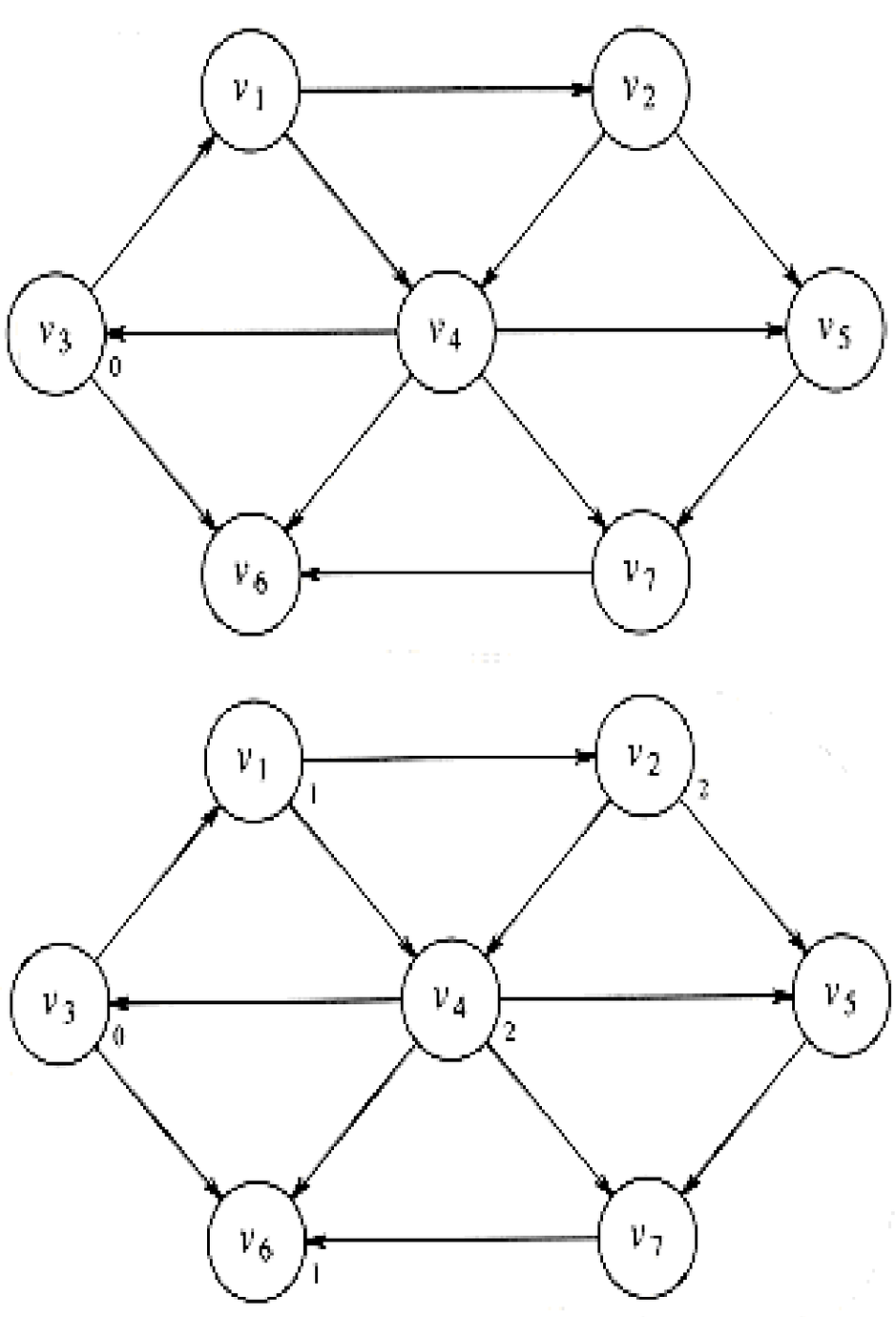
Unweighted Shortest Paths

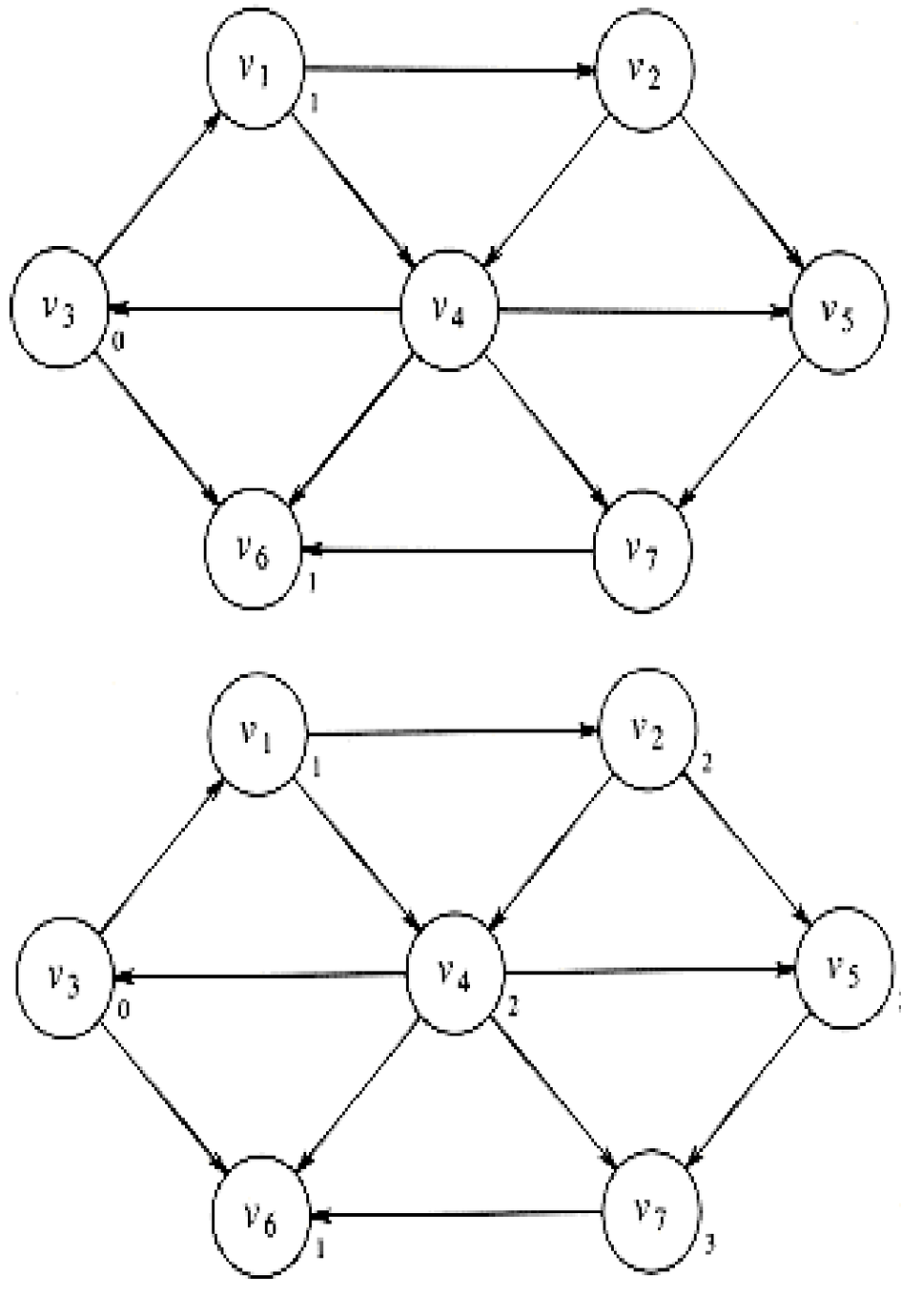


Algorithm

- Suppose we choose s to be v_3.
- The shortest path from s to v_3 is then a path of length 0.
- Now we can start looking for all vertices that are a distance 1 away from s.
  - These can be found by looking at the vertices that are adjacent to s.
  - Continue to look for vertices that are a distance 1

Example





ersity of H

Example

v	Known	d_v	p_v
v1	0	∞	0
v2	0	∞	0
v3	0	0	0
v4	0	∞	0
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

Example

	Initial State			v3 Dequeued			v1 Dequeued			v6 Dequeued		
v	Known	dv	pv	Known	dv	pv	Known	dv	pv	Known	dv	pv
v1	0	•	0	0	1	v3	1	1	v3	1	1	v3
v2	0	•	0	0	•	0	0	2	v1	0	2	v1
v3	0	0	0	1	0	0	1	0	0	1	0	0
v4	0	•	0	0	•	0	0	2	v1	0	2	v1
v5	0	•	0	0	•	0	0	•	0	0	•	0
v6	0	•	0	0	1	v3	0	1	v3	1	1	v3
v7	0	•	0	0	•	0	0	•	0	0	•	0
Q:		v3			v1,v6			v6,v2,v4			v2,v4	

Example

	v2 Dequeued			v4 Dequeued			v5 Dequeued			v7 Dequeued		
v	Known	dv	pv	Known	dv	pv	Known	dv	pv	Known	dv	pv
v1	1	1	v3	1	1	v3	1	1	v3	1	1	v3
v2	1	2	v1	1	2	v1	1	2	v1	1	2	v1
v3	1	0	0	1	0	0	1	0	0	1	0	0
v4	0	2	v1	1	2	v1	1	2	v1	1	2	v1
v5	0	3	v2	0	3	v2	1	3	v2	1	3	v2
v6	1	1	v3	1	1	v3	1	1	v3	1	1	v3
v7	0	•	0	0	3	v4	0	3	v4	1	3	v4
Q:		v4,v5			v5,v7			v7			empty	

Breadth-first Search

- This strategy for searching a graph is known as **breadth-first search**.
- It operates by processing vertices in layers: the vertices closest to the start are evaluated first, and the most distant vertices are evaluated last.
- This is much the same as a level-order traversal for trees.

Notes

- For each vertex, we will keep track of three pieces of information.
  - First, we will keep its distance from **s** in the entry d_v.
    Initially all vertices are unreachable except for **s**, whose path length is 0.
  - The entry in p_v is the bookkeeping variable, which will allow us to print the actual paths.
  - The entry **known** is set to 1 after a vertex is processed. Initially, all entries are unknown, including the start vertex.

Notes

- When a vertex is known, we have a guarantee that no cheaper path will ever be found, and so processing for that vertex is essentially complete.

- What is the running time of the algorithm?

  The running time of the algorithm is $O(|V|^2)$, because of the doubly nested for loops.

Dijkstra's Algorithm

- If the graph is weighted, the problem (apparently) becomes harder.
- Still we can use the ideas from the unweighted case.

Outline

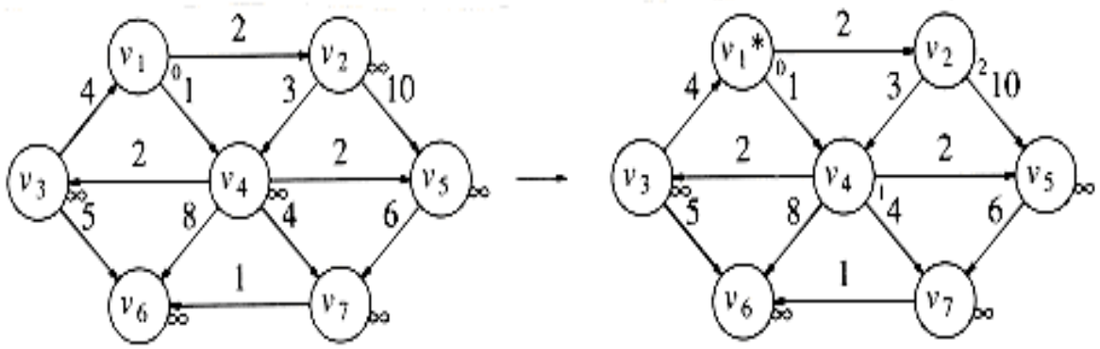
- Each vertex is marked as either **known** or **unknown**.
- A tentative distance d_v is kept for each vertex, as before.
- This distance turns out to be the shortest path length from s to v using only known vertices as intermediates.
- As before, we record p_v, which is the last vertex to cause a change to d_v.

Greedy Algorithm

- The general method to solve the single-source shortest-path problem is known as **Dijkstra's** algorithm.
- This thirty-year-old solution is a prime example of a **greedy algorithm**.
- Greedy algorithms generally solve a problem in stages by doing what appears to be the **best** thing at each stage.

Notes

- Dijkstra's algorithm proceeds in stages.
- At each stage, Dijkstra's algorithm selects a vertex v, which has the smallest d_v among all the unknown vertices, and declares that the shortest path from s to v is known.
- The remainder of a stage consists of updating the values of d_w.



Initial Configuration

v	Known	dv	pv
v1	0	0	0
v2	0	•	0
v3	0	•	0
v4	0	•	0
v5	0	•	0
v6	0	•	0
v7	0	•	0

After v_1 Is Known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	•	0
v4	0	1	v1
v5	0	•	0
v6	0	•	0
v7	0	•	0

After v_4 Is Known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	3	v4
v4	1	1	v1
v5	0	3	v4
v6	0	9	v4

After v_2 Is Known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	0	3	v4
v4	1	1	v1
v5	0	3	v4
v6	0	9	v4

After v_5 and v_3 Are Known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	0	8	v3
v7	0	5	v4

After v_7 Is Known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	0	6	v7

After v_6 Is Known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	1	6	v7

Graphs with Negative Edge Costs

- Dijkstra's algorithm does not work with negative edge costs.
- A combination of the weighted and unweighted algorithms will solve the problem, but at the cost of a drastic increase in running time.
- The running time is $O(|E| \cdot |V|)$ if adjacency lists are used.

Acyclic Graphs

- We can improve Dijkstra's algorithm by changing the order in which vertices are declared known, otherwise known as the **vertex selection rule**.
- The new rule is to select vertices in topological order.
- The algorithm can be done in one pass, since the selections and updates can take place as the topological sort is being performed.

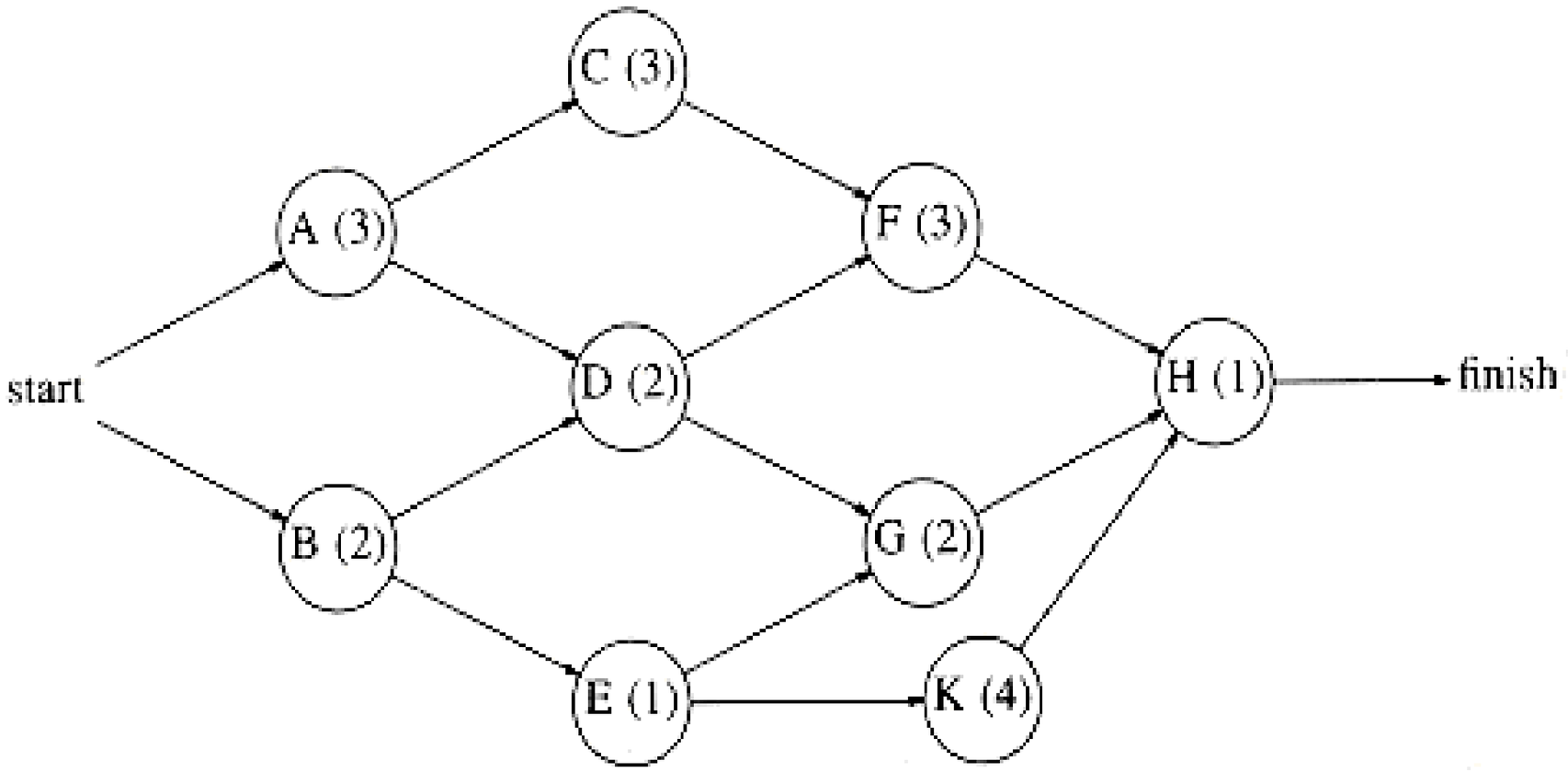
Acyclic Graphs

- Why does this selection rule work?
  - Because when a vertex v is selected, its distance, d_v, can no longer be lowered,
    - since by the topological ordering rule it has no incoming edges emanating from unknown nodes.
- The running time is $O(|E| + |V|)$, since the selection takes constant time.

Applications

- Downhill skiing problem
- Modeling of (nonreversible) chemical reactions.
- Critical path analysis
  - Each node represents an activity that must be performed, along with the time it takes to complete the activity.
  - This graph is thus known as an **activity-node graph**.

Activity-node Graph



The edges represent precedence relationships: An edge (v, w) means that activity v must be completed before activity w may begin.

Of course, this implies that the graph must be acyclic.

Notes

- Model construction projects
  - What is the earliest completion time for the project?
  - We can see from the graph that 10 time units are required along the path *A, C, F, H.*

Notes

- Another important question is to determine which activities can be delayed, and by how long, without affecting the minimum completion time.
  - For instance, delaying any of *A*, *C*, F, or *H* would push the completion time past 10 units.
  - On the other hand, activity *B* is less critical and can be delayed up to two time units without affecting the final completion time.

All-Pairs Shortest Path

- Find the shortest paths between all pairs of vertices in the graph.
- Brute Force--Just run the appropriate single-source algorithm |V| times.
- In Chapter 10, there is an $O(|V|^3)$ algorithm to solve this problem for weighted graphs.
- On sparse graphs, of course, it is faster to run |V| Dijkstra's algorithms coded with priority queues.

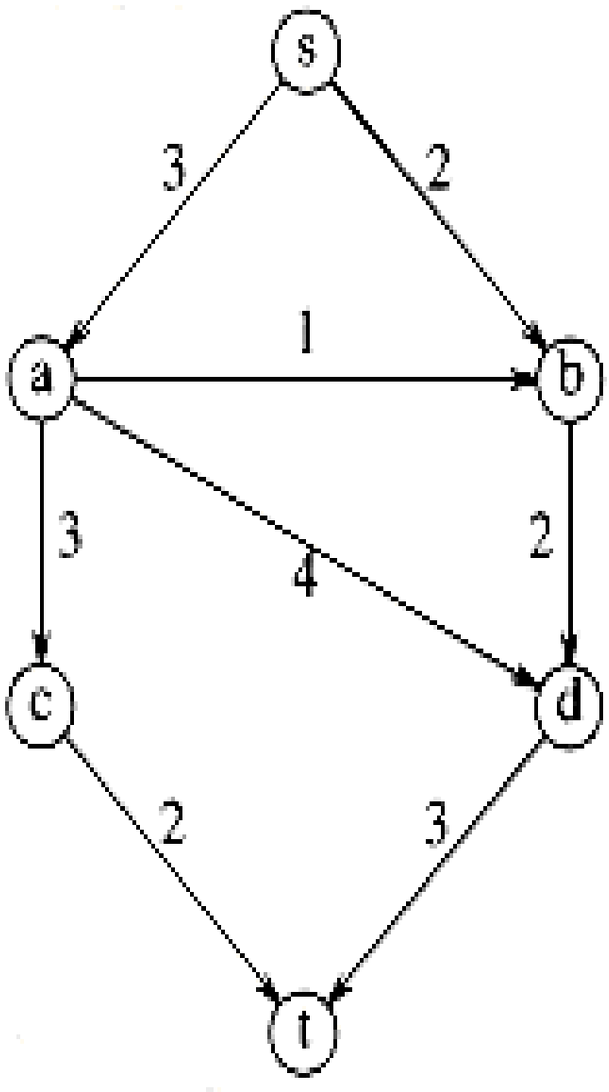
Network Flow Problems

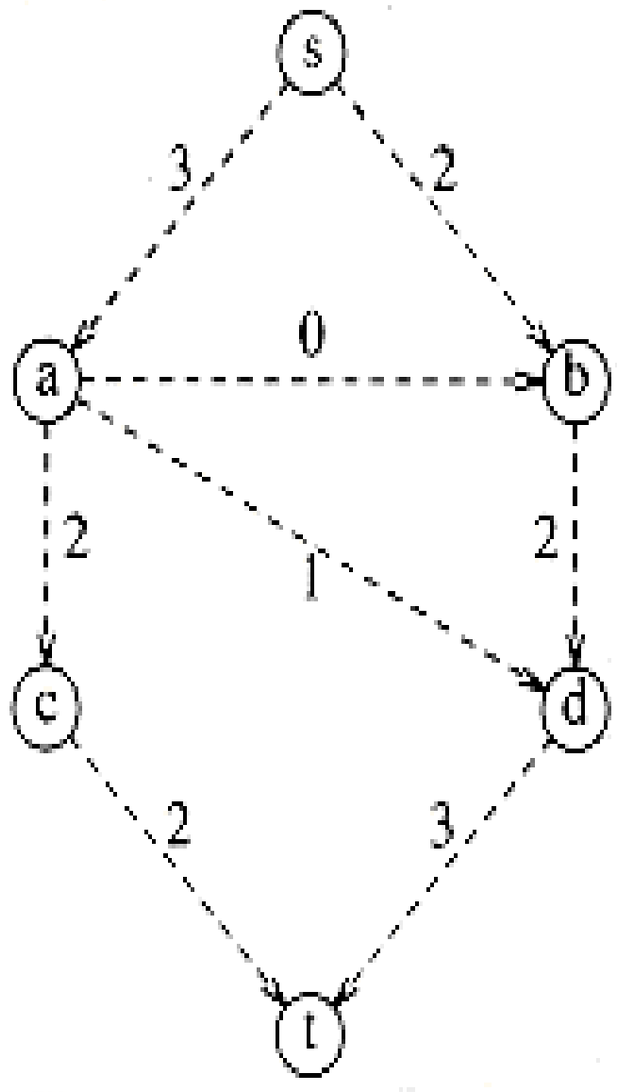
- Suppose we are given a directed graph $G = (V, E)$ with edge capacities $c_{v,w}$.
  - These capacities could represent the amount of water that could flow through a pipe or the amount of traffic that could flow on a street between two intersections.
- We have two vertices: s, which we call the **source**, and t, which is the **sink**.
- Through any edge, (v, w), at most $c_{v,w}$ units of "flow" may pass.

Problem

- At any vertex, v, that is not either s or t, the total flow coming in must equal the total flow going out.
- The **maximum flow problem** is to determine the maximum amount of flow that can pass from s to t.

Example





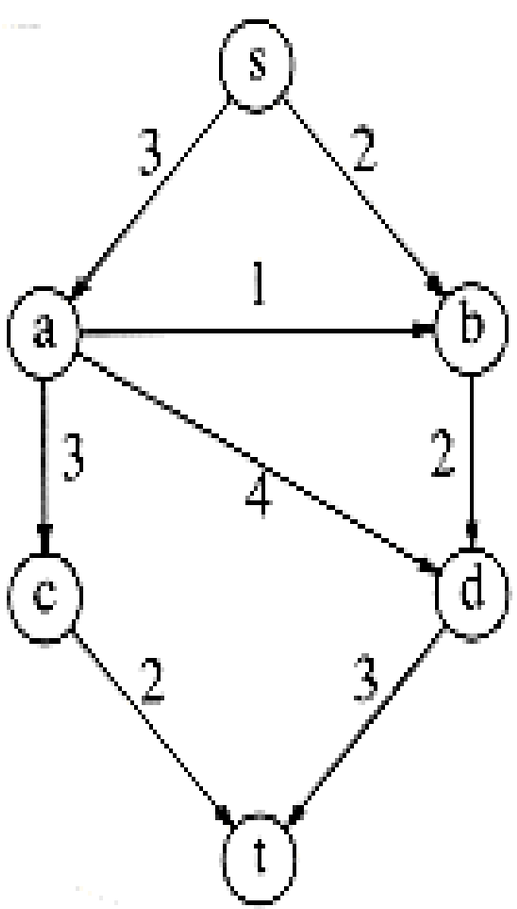
A Simple Maximum-Flow Algorithm

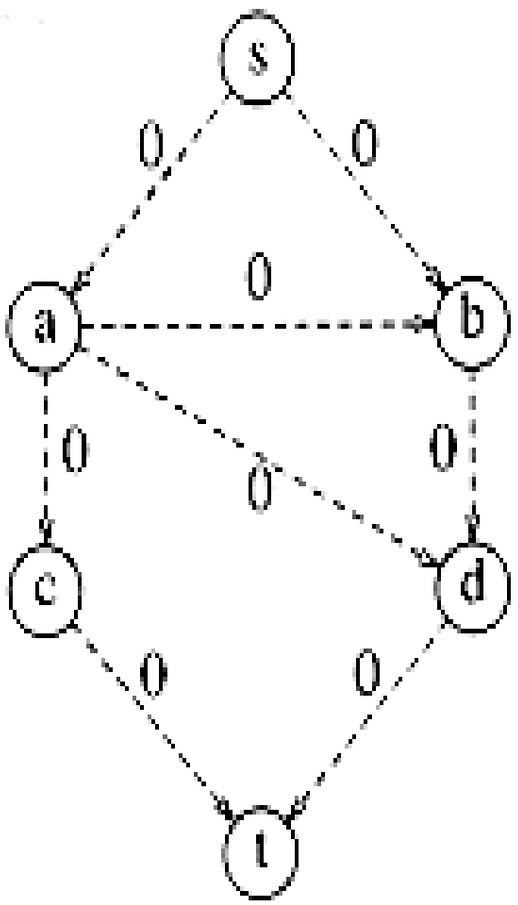
- G_f- a **flow graph**. It tells the flow that has been attained at any stage in the algorithm.
  - Initially all edges in G_f have no flow.
  - G_f should contain a maximum flow when the algorithm terminates.
- G_r- the **residual graph**. G_r tells, for each edge, how much more flow can be added.
  - We calculate this by subtracting the current flow from the capacity for each edge.
  - An edge in G_r is known as a **residual edge**.

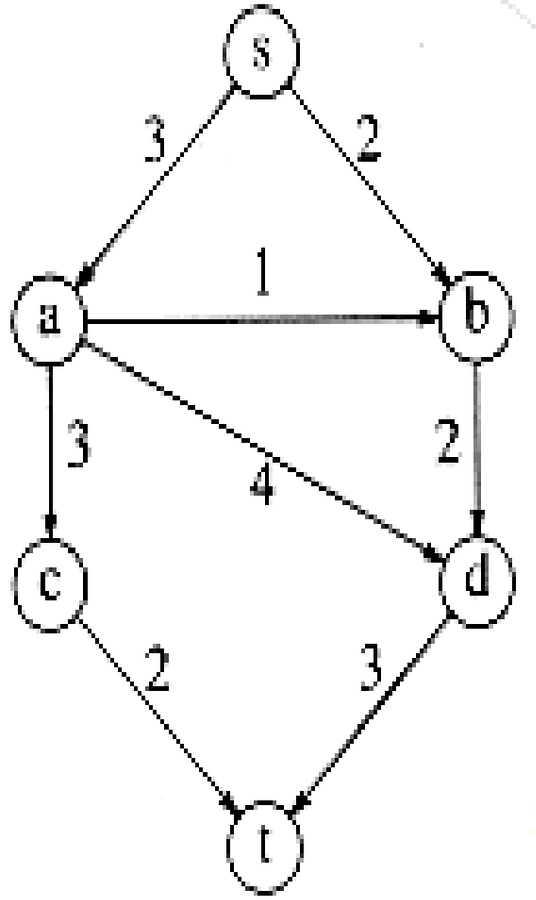
A Simple Maximum-Flow Algorithm

- At each stage, we find a path in G_r from s to t.
- This path is known as an **augmenting path**.
  - The minimum edge on this path is the amount of flow that can be added to every edge on the path.
- We do this by adjusting G_f and recomputing G_r.
- When we find no path from s to t in G_r, we terminate.
- This algorithm is **nondeterministic**, in that we are **free** to choose any path from s to t.

Example

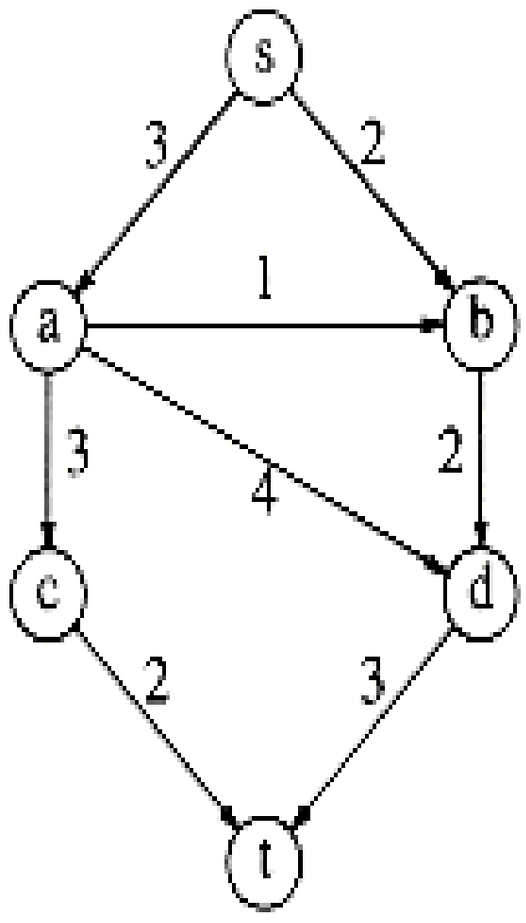


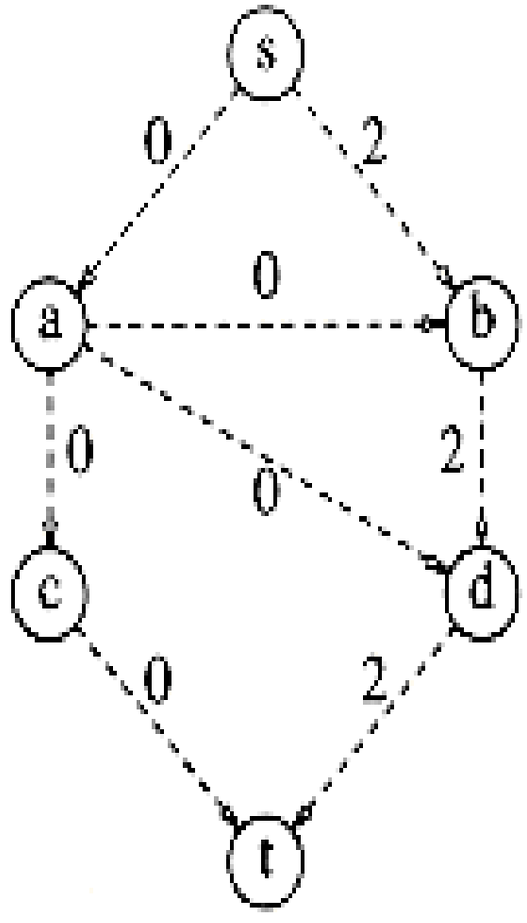


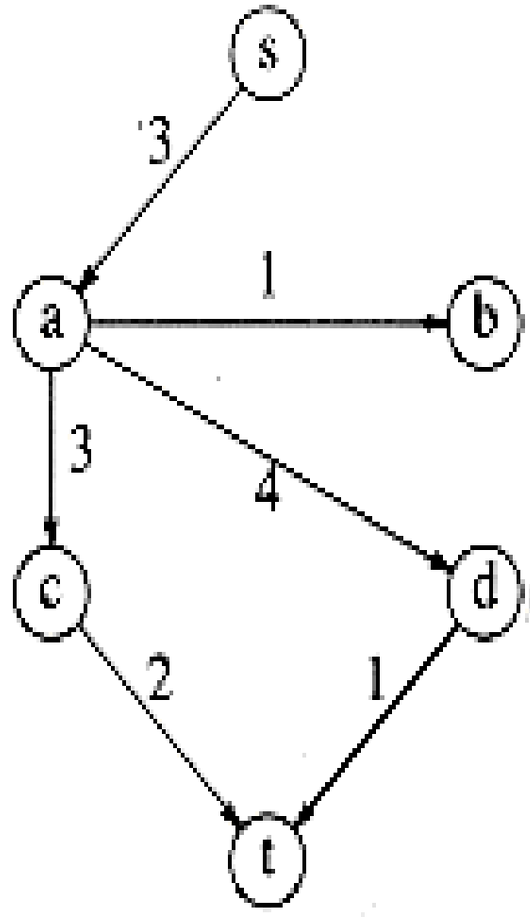


Initial stages of the graph, flow graph, and residual graph

Example

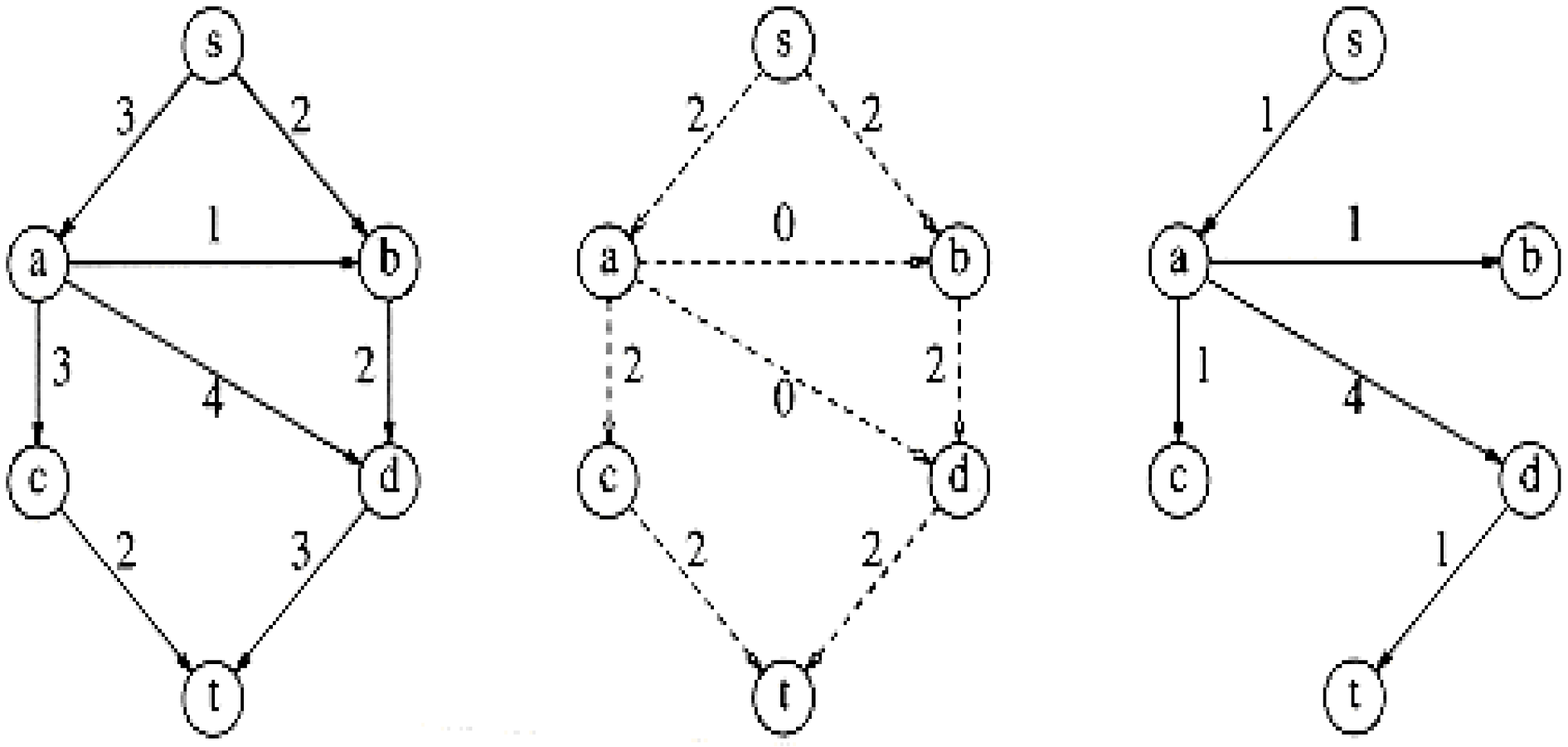






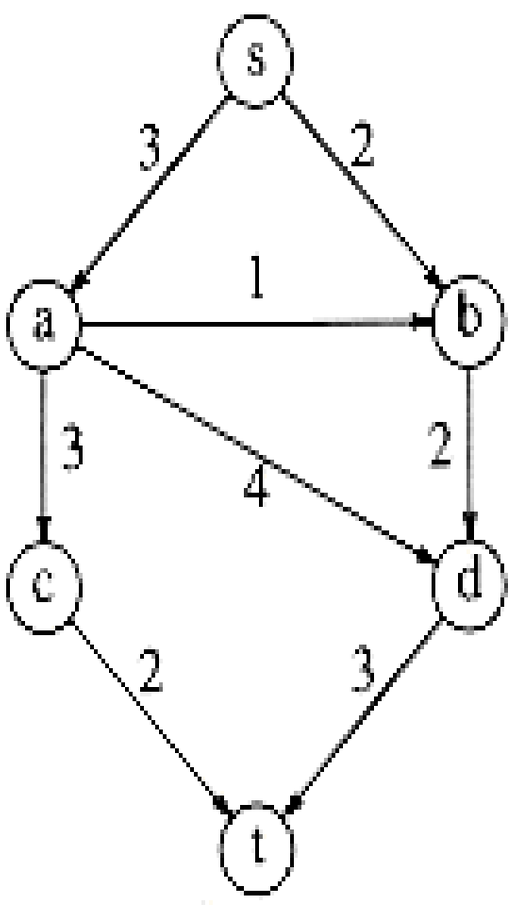
G, G_f, G_r after two units of flow added along s, b, d, t

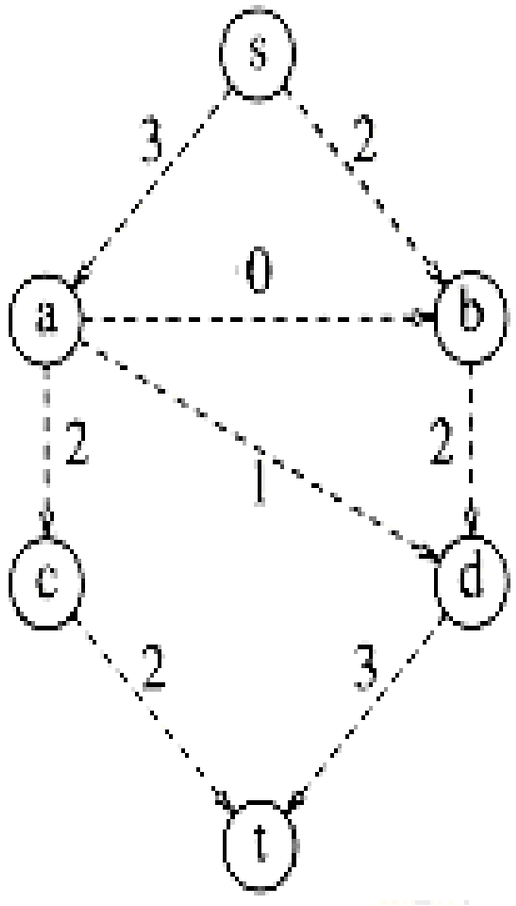
Example

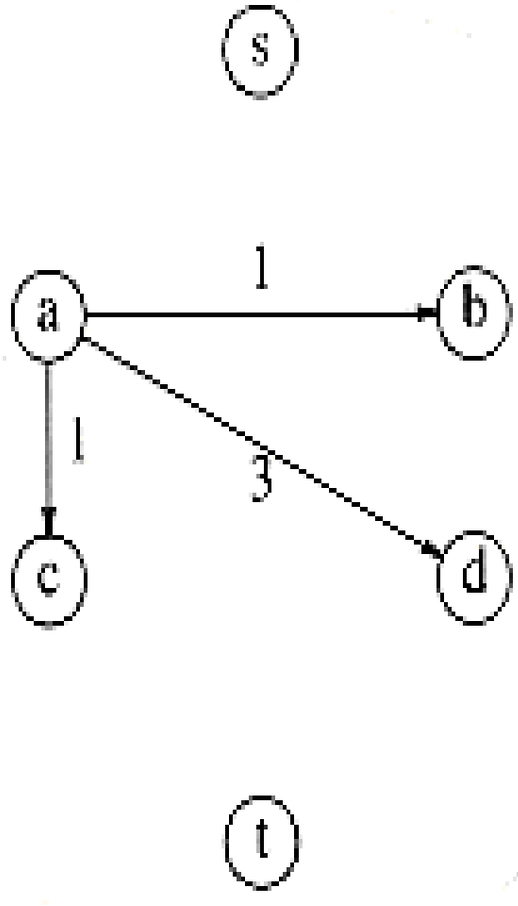


G, G_f, G_r after two units of flow added along s, a, c, t

Example







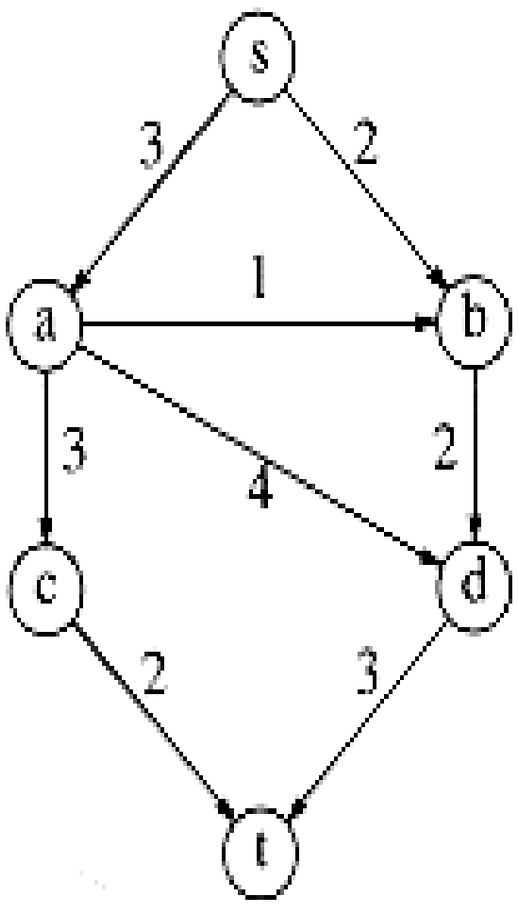
G, G_f, G_r after one unit of flow added along s, a, d, t --algorithm terminates

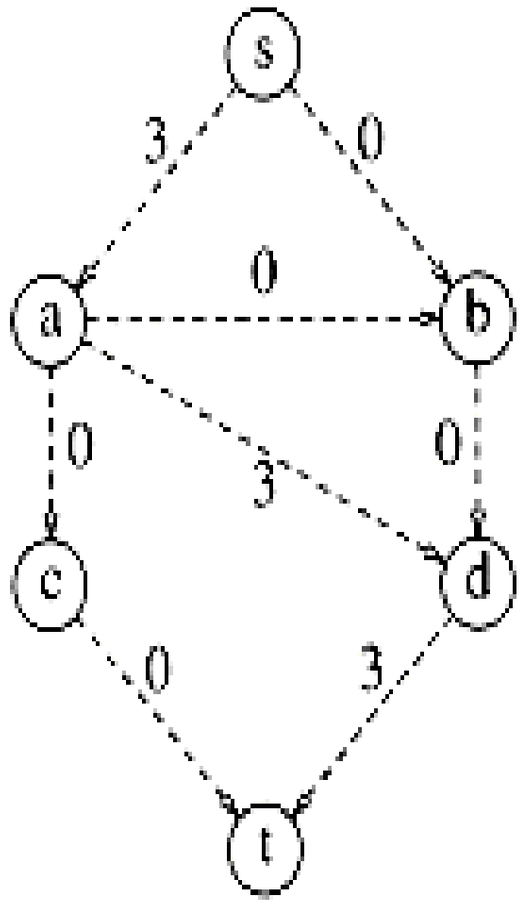
Problem

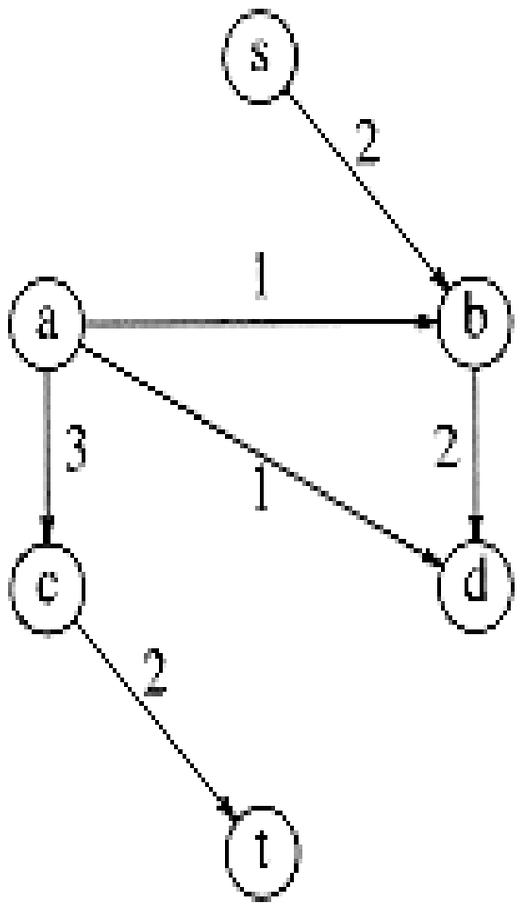
- When t is unreachable from s the algorithm terminates.
- The resulting flow of 5 happens to be the maximum.

- Problem of Not Being Optimal
  - Suppose that with our initial graph, we chose the path s, a, d, t.
  - The result of this choice is that there is now no longer any path from s to t in the residual graph.

Example





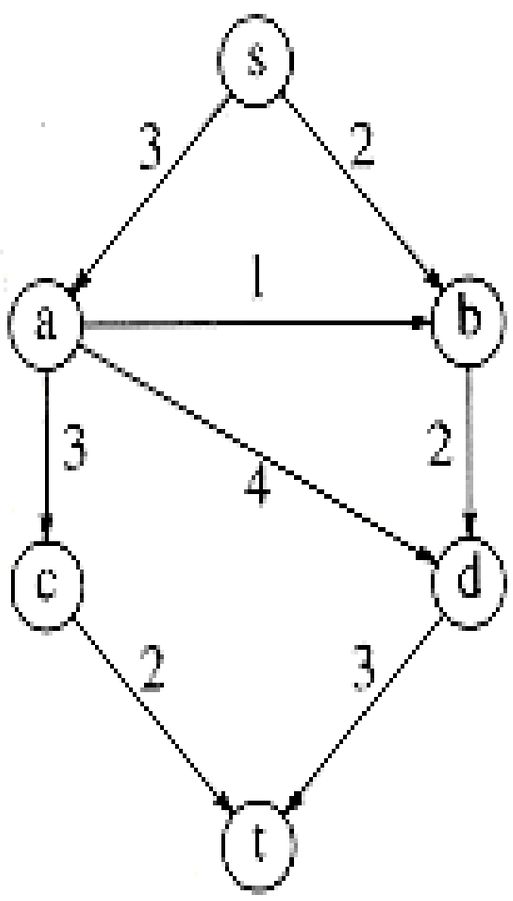


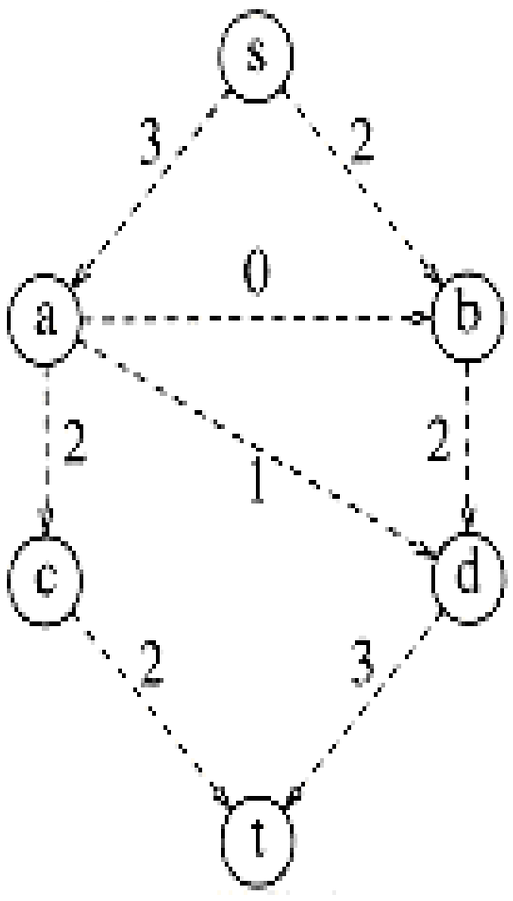
G, G_f, G_r if initial action is to add three units of flow along s, a, d, t -- algorithm terminates with suboptimal solution

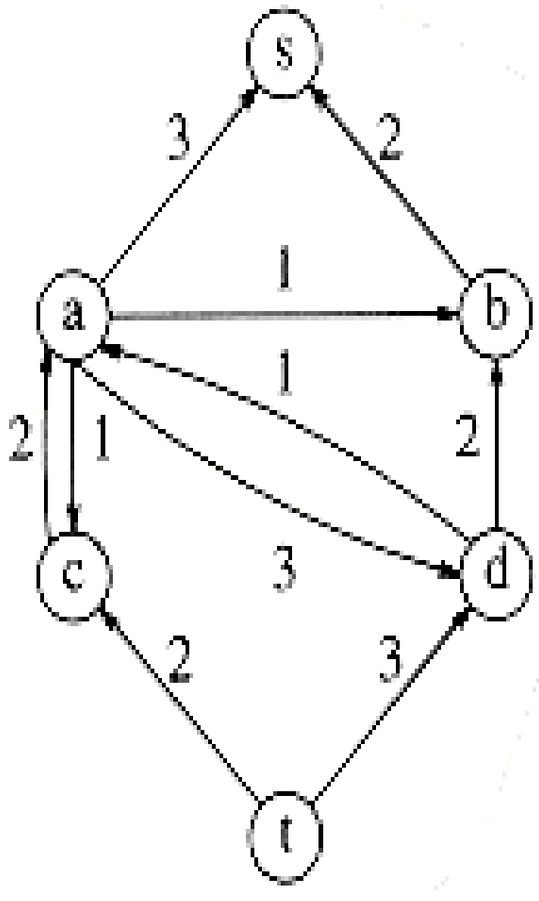
How To Make It Optimal

- In order to make this algorithm work, we need to allow the algorithm to **change its mind**.
- To do this, for every edge (v, w) with flow $f_{v,w}$ in the flow graph, we will add an edge in the residual graph (w, v) of capacity $f_{v,w}$.
- In effect, we are allowing the algorithm to undo its decisions by sending flow back in the opposite direction.

Example

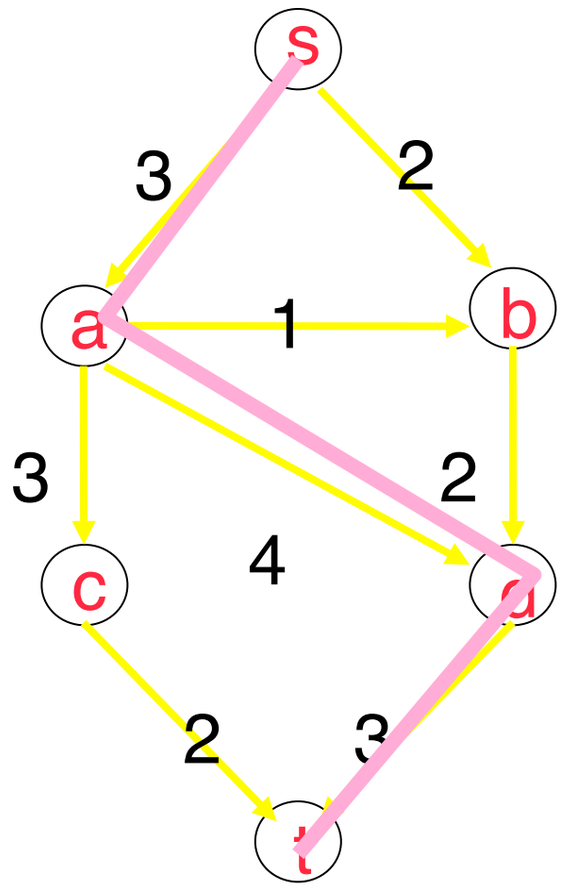


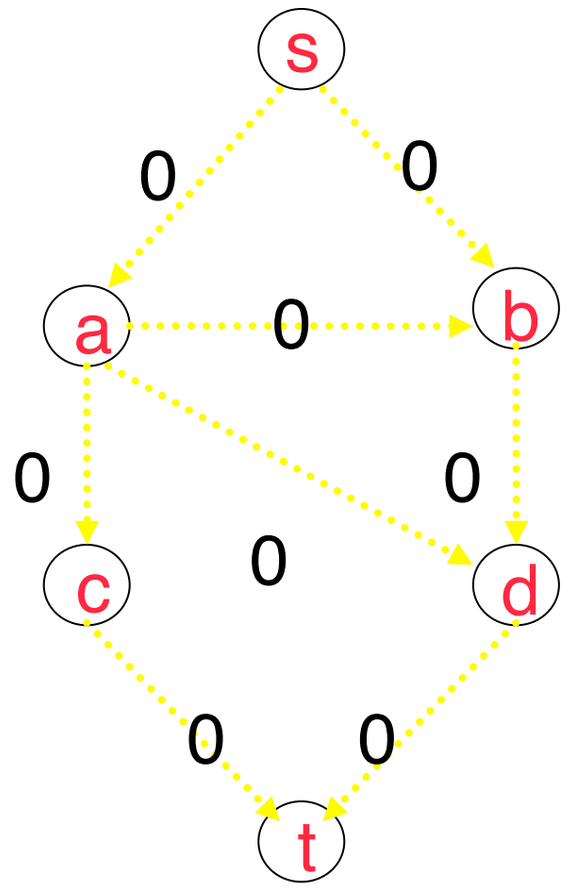


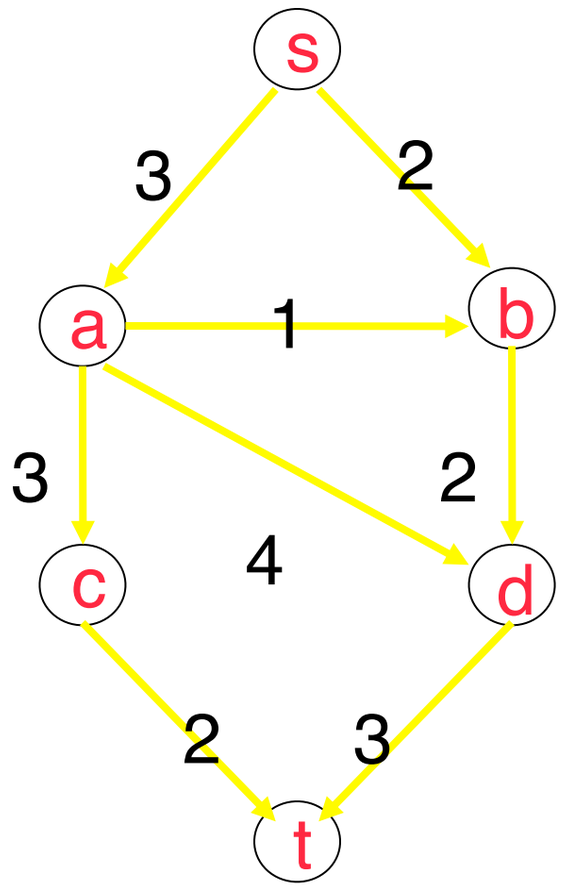


Graphs after three units of flow added along s, a, d, t using correct algorithm

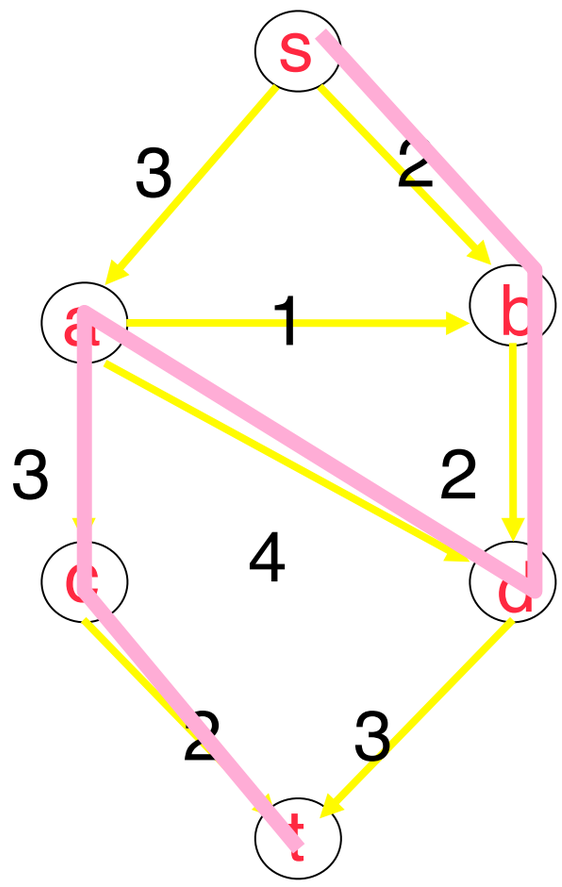
Example

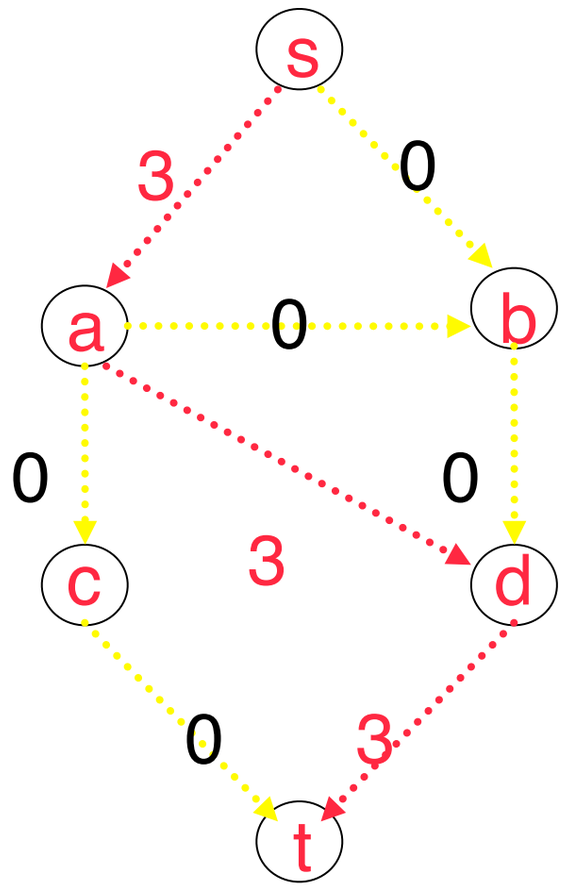


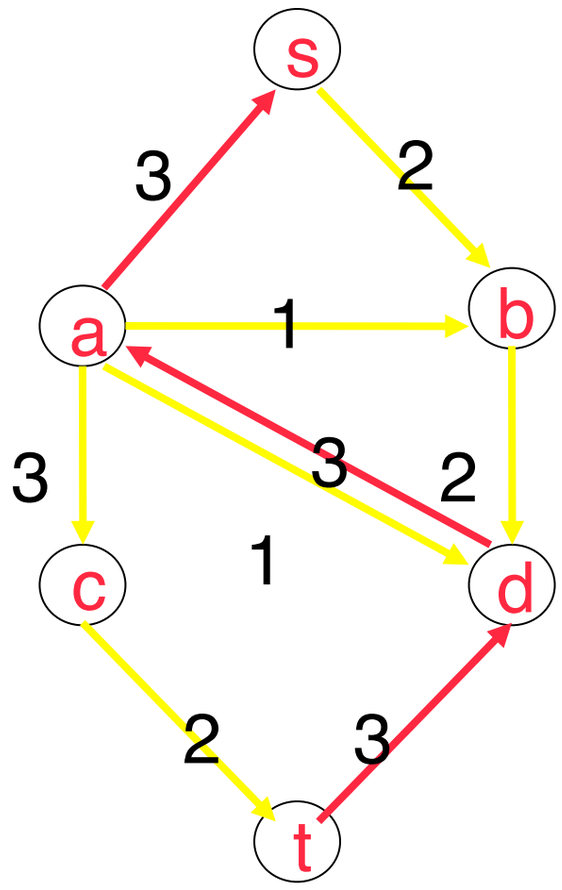




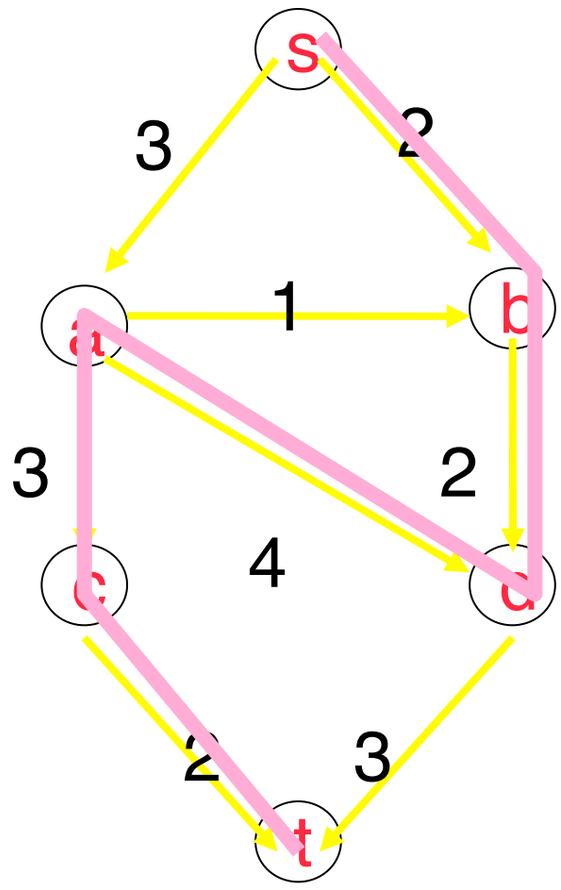
Example

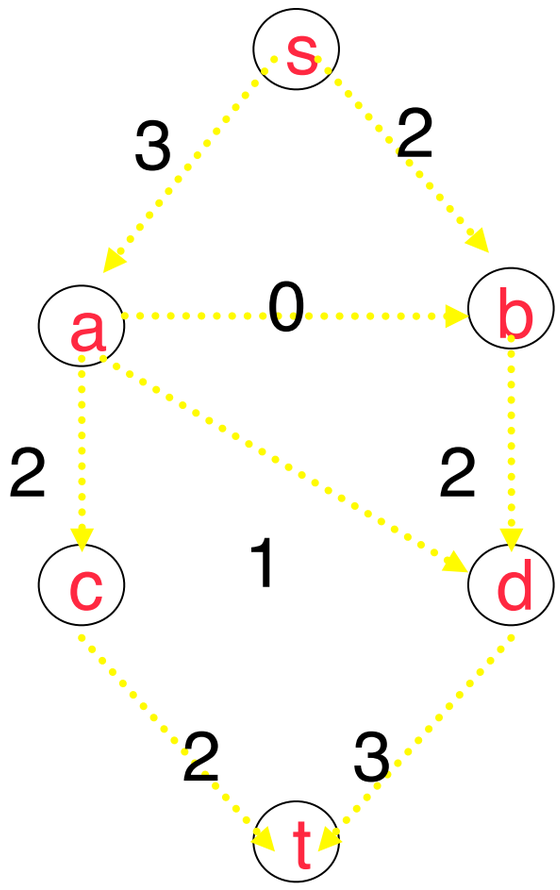


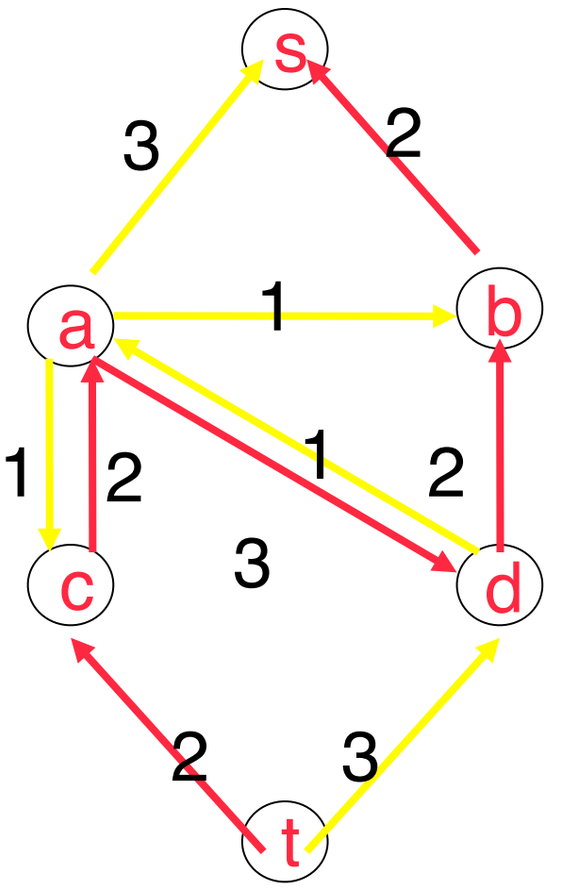




Example







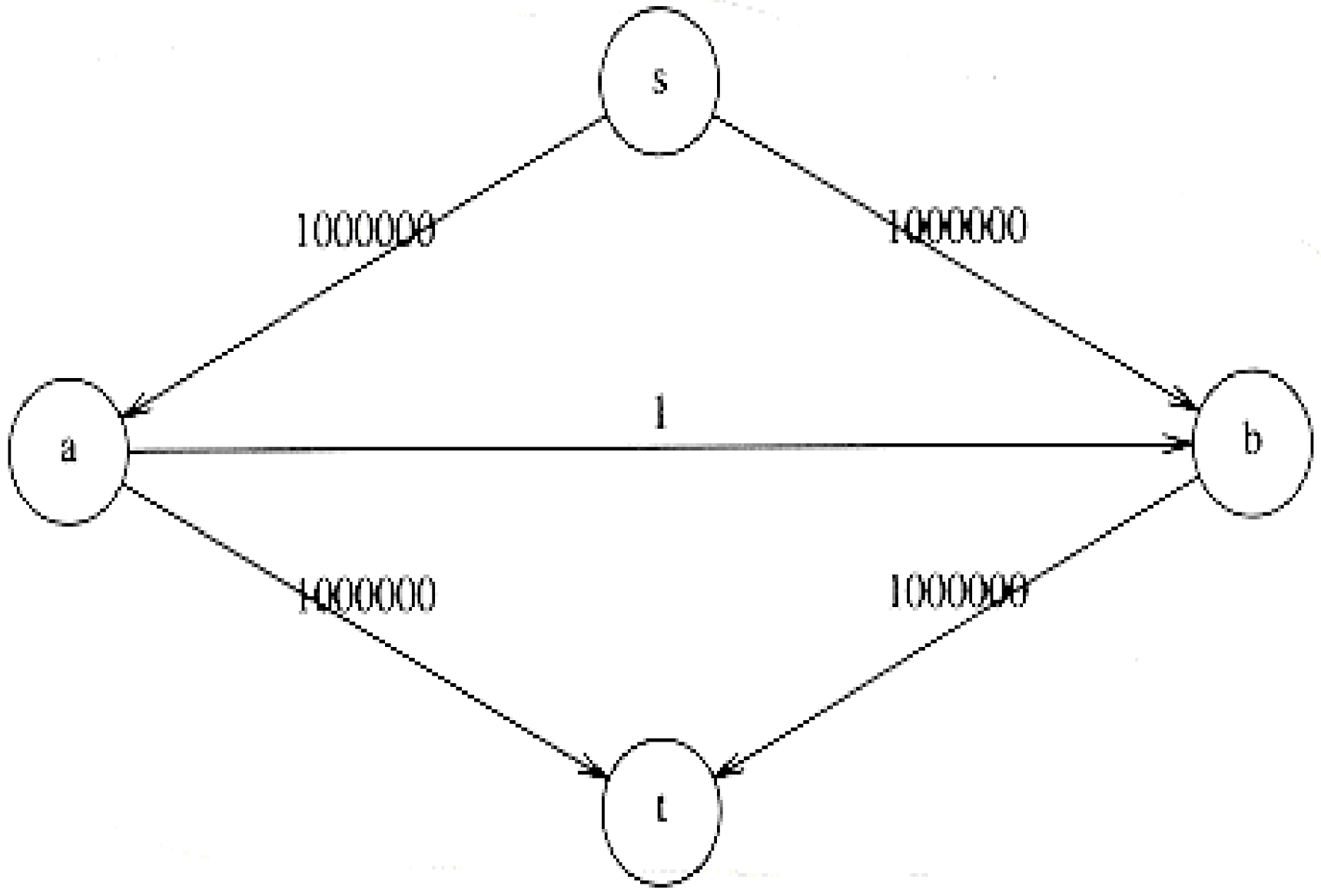
Notes

- In the residual graph, there are edges in **both directions** between a and d.
- Either one more unit of flow can be pushed from a to d, or up to three units can be pushed back--**we can undo flow**.
- Now the algorithm finds the augmenting path s, b, d, a, c, t, of flow 2.
- By pushing two units of flow from d to a, the algorithm takes two units of flow away from the edge (a, d) and is essentially **changing its mind**.

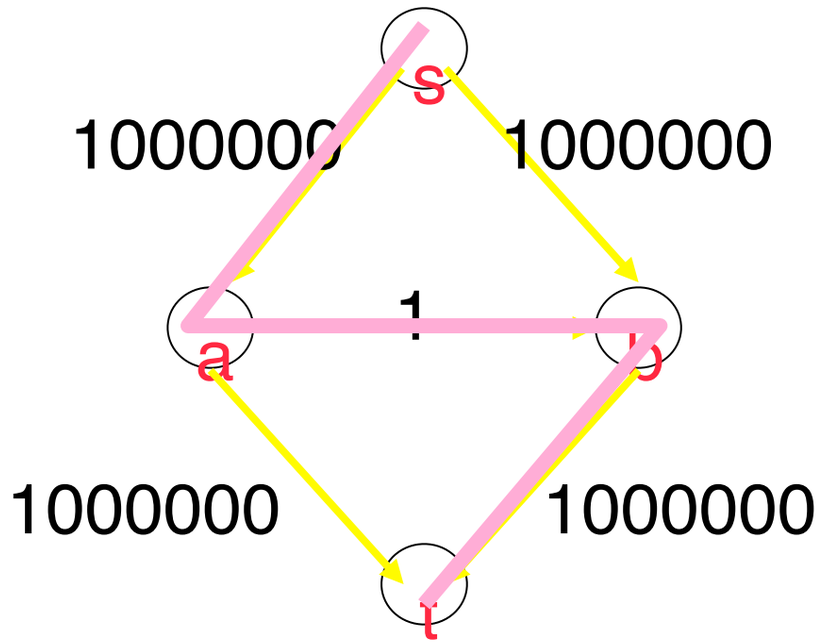
Notes

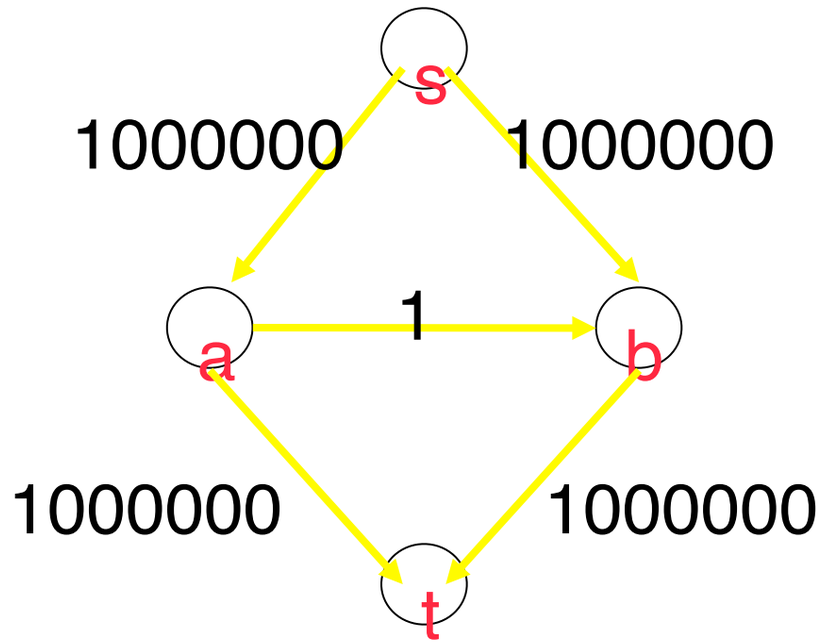
- If the capacities are all integers and the maximum flow is f, then, since each augmenting path increases the flow value by at least 1, f stages suffice.
- The total running time is $O(f \cdot |E|)$, since an augmenting path can be found in $O(|E|)$ time by an unweighted shortest-path algorithm.

Bad Case for Augmenting

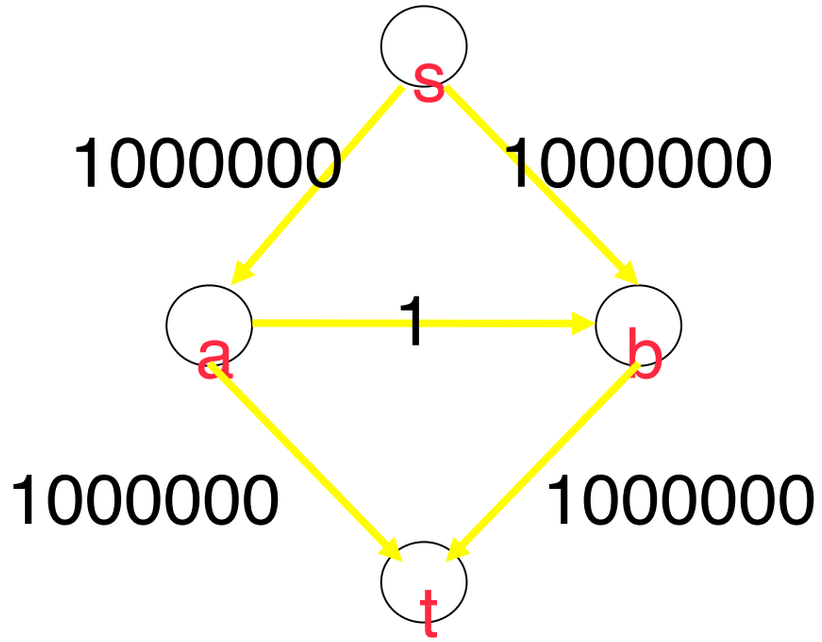


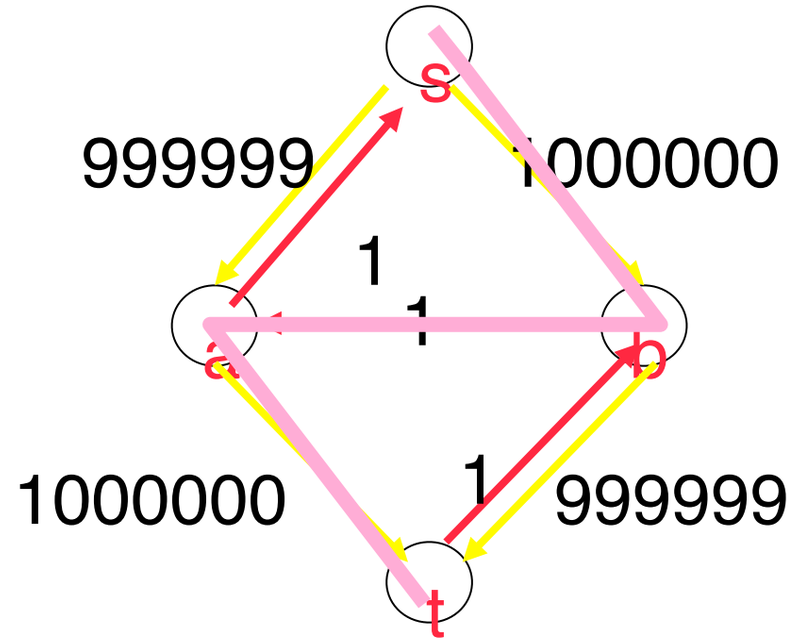
Example



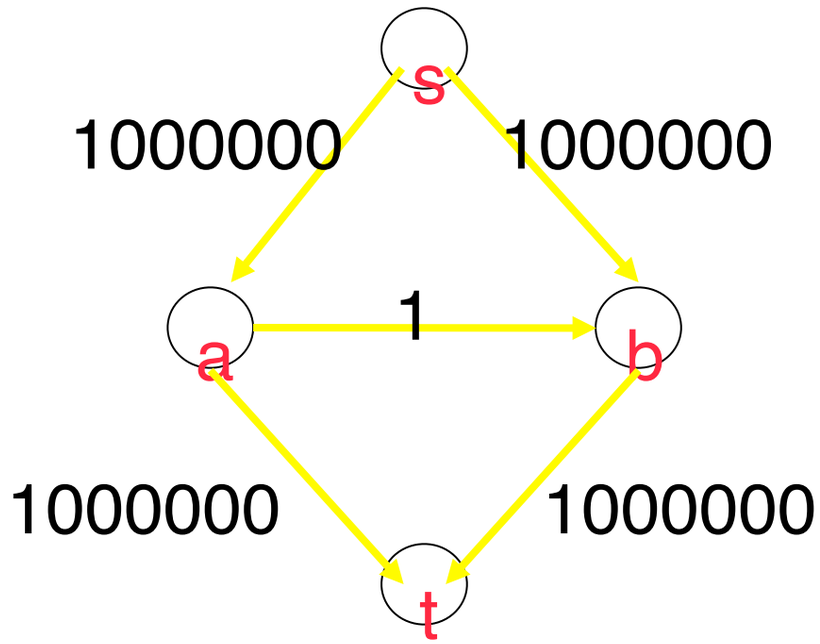


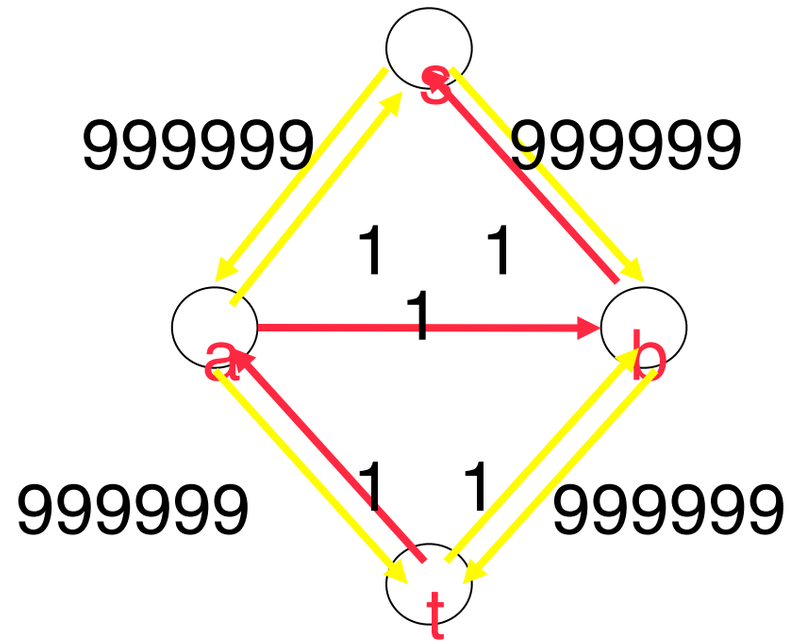
Example





Example





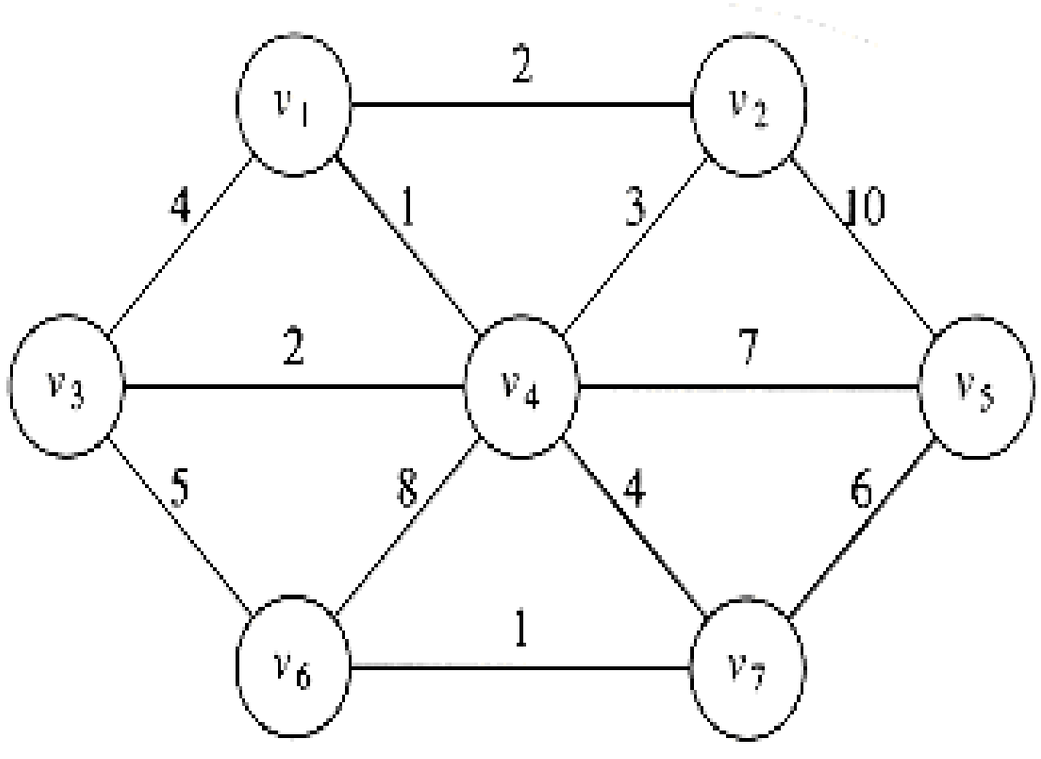
Problem

- The maximum flow is seen by inspection to be 2,000,000 by sending 1,000,000 down each side.
- Using the algorithm, 2,000,000 augmentations would be required, when we could get by with only 2.
- **Solution**--Always to choose the augmenting path that allows the **largest increase** in flow.
- Finding such a path is similar to solving a **weighted shortest-path** problem.

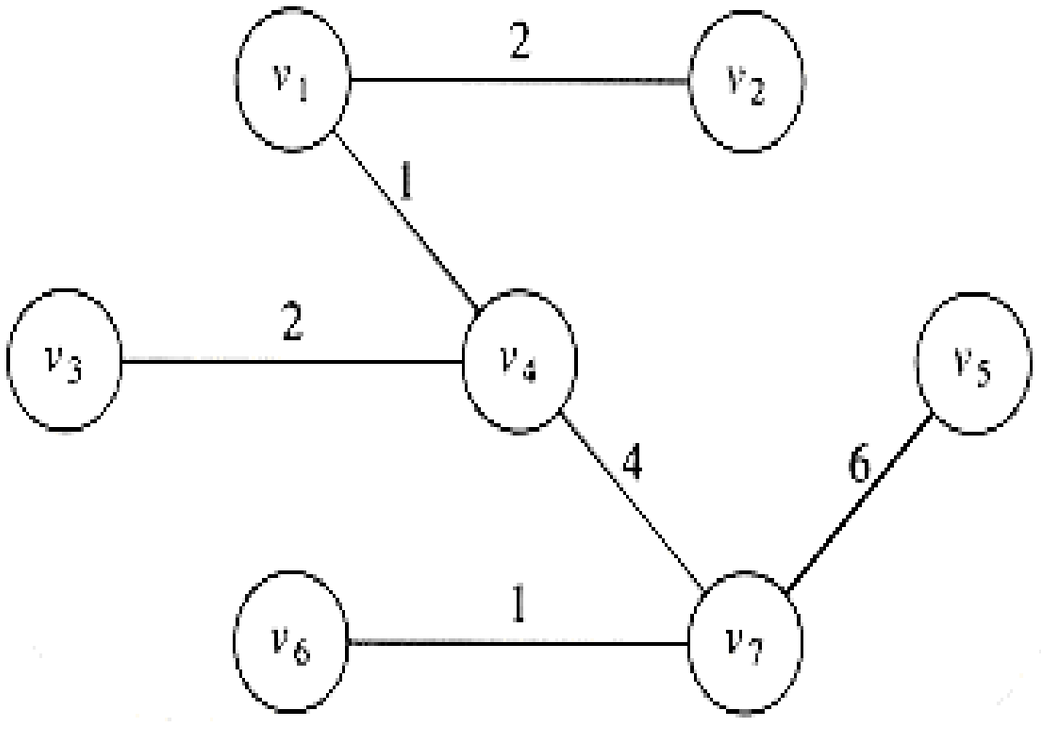
Minimum Spanning Tree

- Finding a minimum spanning tree in an undirected graph.
- Informally, a minimum spanning tree of an undirected graph *G* is a **tree** formed from graph edges that connects **all** the vertices of *G* at **lowest total cost**.
- A minimum spanning tree exists if and only if *G* is connected.
- Application--wiring of a house

Example



Is the minimum spanning tree unique?



Notes

- Notice that the number of edges in the minimum spanning tree is |V| - 1.
- The minimum spanning tree is a tree because it is **acyclic**.
- It is **spanning** because it covers every edge, and it is **minimum** because the sum of all cost is the minimum.

Prim's Algorithm

- It is to grow the tree in successive stages.
- In each stage, one **node** is picked as the root, and we add an **edge**, and thus an associated vertex, to the tree.
- The algorithm then finds, at each stage, a new vertex to add to the tree by choosing the edge (u, v) such that the cost of (u, v) is the smallest among all edges where u is in the tree and v is not.

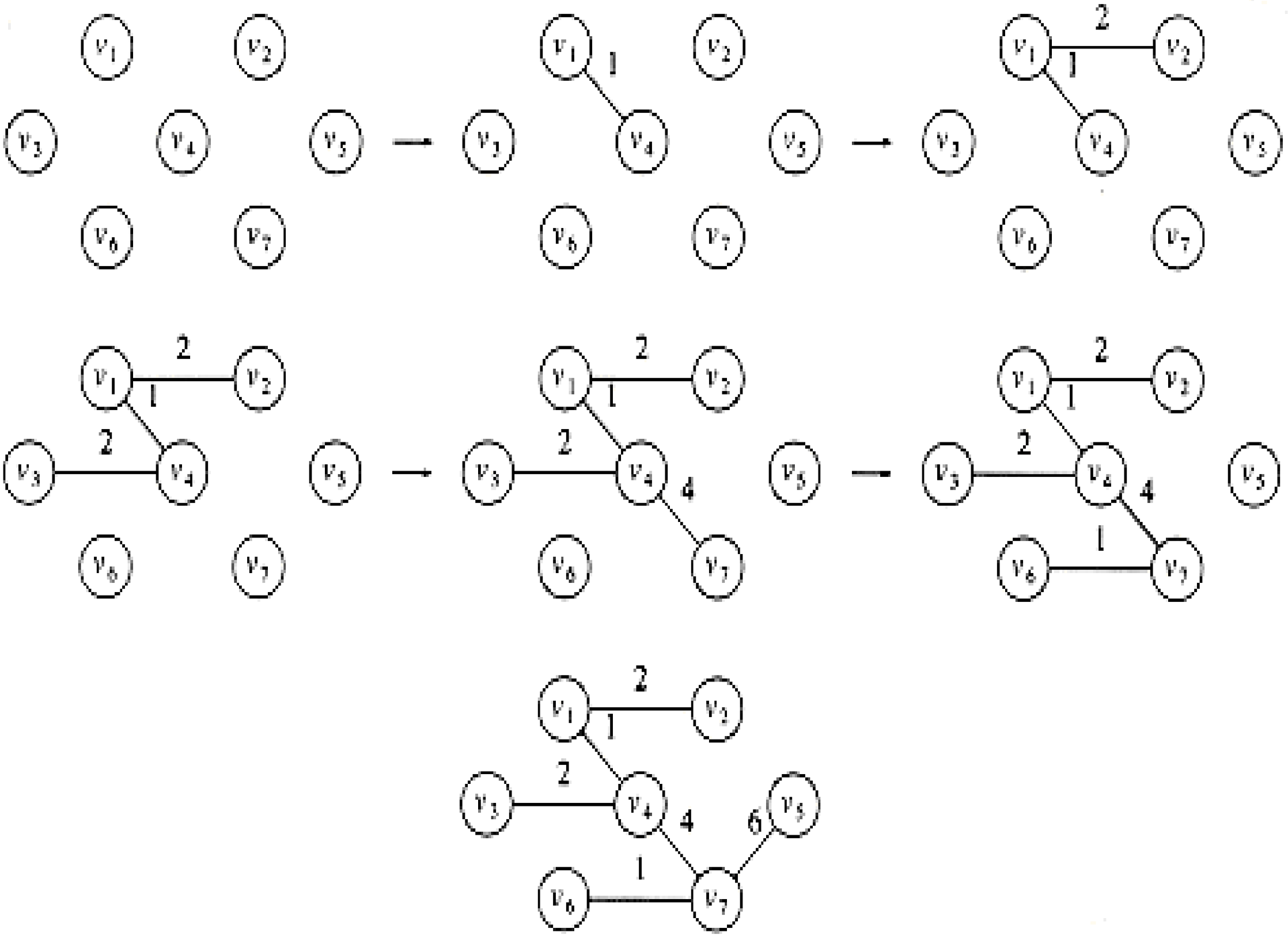
Notes

- Prim's algorithm is essentially identical to Dijkstra's algorithm for shortest paths.
- For each vertex we keep values d_v and p_v and an indication of whether it is known or unknown.

  d_v is the weight of the shortest arc connecting v to a known vertex.

  p_v, as before, is the last vertex to cause a change in d_v .

- After a vertex v is selected, for each unknown w adjacent to v, $d_v = \min(d_w, c_{w,v})$.



Example

v	Known	dv	pv
v1	0	0	0
v2	0	•	0
v3	0	•	0
v4	0	•	0
v5	0	•	0
v6	0	•	0
v7	0	•	0

After V_1 is Known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	4	v1
v4	0	1	v1
v5	0	•	0
v6	0	•	0
v7	0	•	0

Example

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	2	v4
v4	1	1	v1
v5	0	7	v4
v6	0	8	v4
v7	0	4	v4

After V_2 and V_3 are Known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	2	v4
v4	1	1	v1
v5	0	7	v4
v6	0	5	v3
v7	0	4	v4

After V_7 is Known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	2	v4
v4	1	1	v1
v5	0	6	v7
v6	0	1	v7
v7	1	4	v4

After V_5 and V_6 are Known

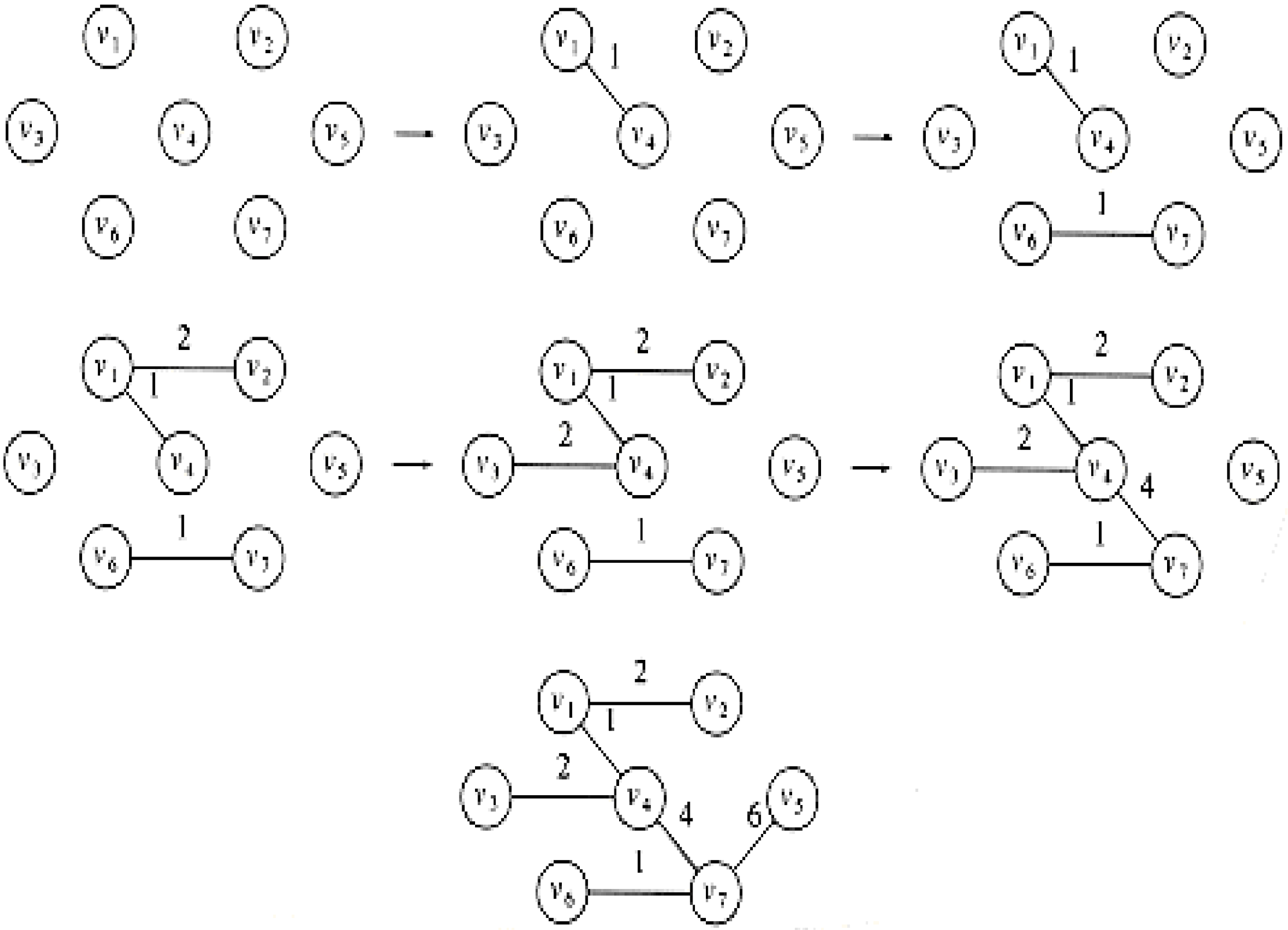
v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	2	v4
v4	1	1	v1
v5	1	6	v7
v6	1	1	v7
v7	1	4	v4

Notes

- Be aware that Prim's algorithm runs on undirected graphs, so when coding it, remember to put every edge in two adjacency lists.
- The running time is $O(|V|^2)$ without heaps, which is optimal for dense graphs, and $O(|E| \log |V|)$ using binary heaps, which is good for sparse graphs.

Kruskal's Algorithm

- A second greedy strategy is continually to select the edges in order of smallest weight and accept an edge if it does not cause a cycle.
- It maintains a forest-- a collection of trees.
- Initially, there are |V| single-node trees.
- Adding an edge merges two trees into one. When the algorithm terminates, there is only one tree, and this is the minimum spanning tree.



Notes

- The algorithm terminates when enough edges are accepted. It is simple to decide whether edge (u,v) should be accepted or rejected.
- The appropriate data structure is the **union**/**find** algorithm.
- The invariant we will use is that at any point in the process, two vertices belong to the same set if and only if they are connected in the current spanning forest.

Notes

- Thus, each vertex is initially in its own set.
- If u and v are in the same set, the edge is rejected, because since they are already connected, adding (u, v) would form a cycle.
- Otherwise, the edge is accepted, and a **union** is performed on the two sets containing u and v.

Example

Edge	Weight	Action
(v1,v4)	1	Accepted
(v6,v7)	1	Accepted
(v1,v2)	2	Accepted
(v3,v4)	2	Accepted
(v2,v4)	3	Rejected
(v1,v3)	4	Rejected
(v4,v7)	4	Accepted
(v3,v6)	5	Rejected
(v5,v7)	6	Accepted

Notes

- The worst-case running time of this algorithm is $O(|E| \log |E|)$, which is dominated by the heap operations.
- Notice that since $|E| = O(|V|^2)$, this running time is actually $O(|E| \log |V|)$.
- In practice, the algorithm is much faster than this time bound would indicate.

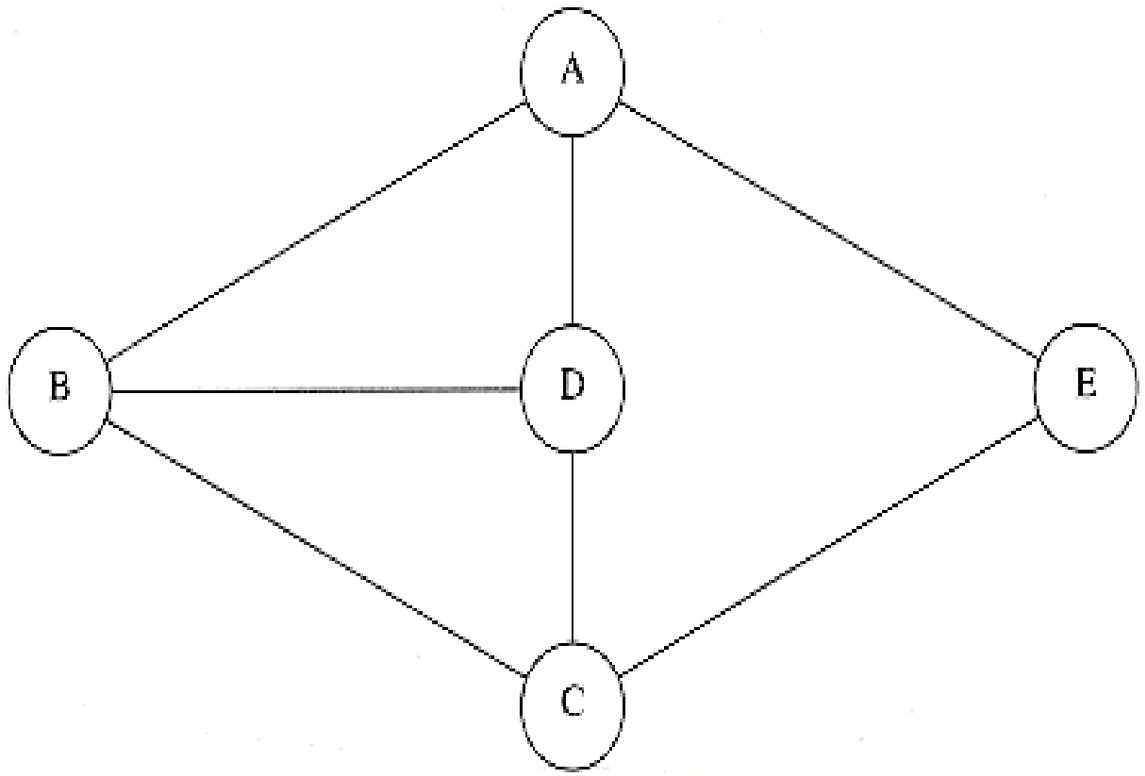
Depth-First Search

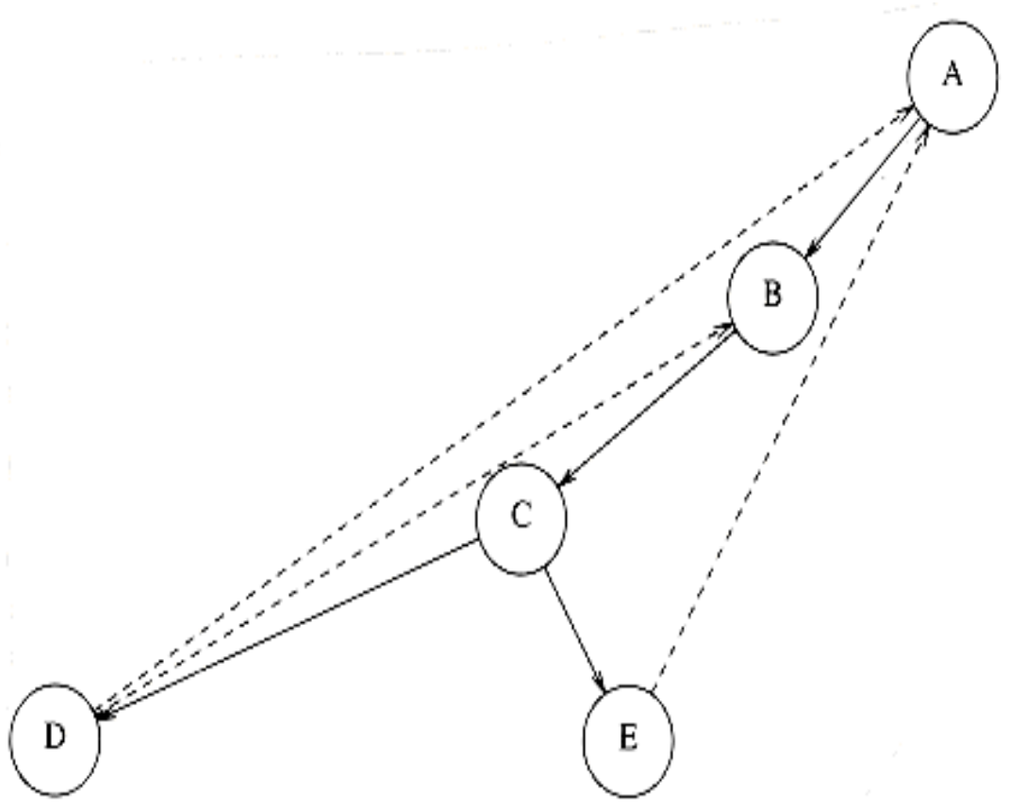
- Depth-first search is a generalization of preorder traversal.
- Starting at some vertex, v, we process v and then recursively traverse all vertices adjacent to v.
- If this process is performed on a tree, then all tree vertices are systematically visited in a total of $O(|E|)$ time.

Depth-First Search

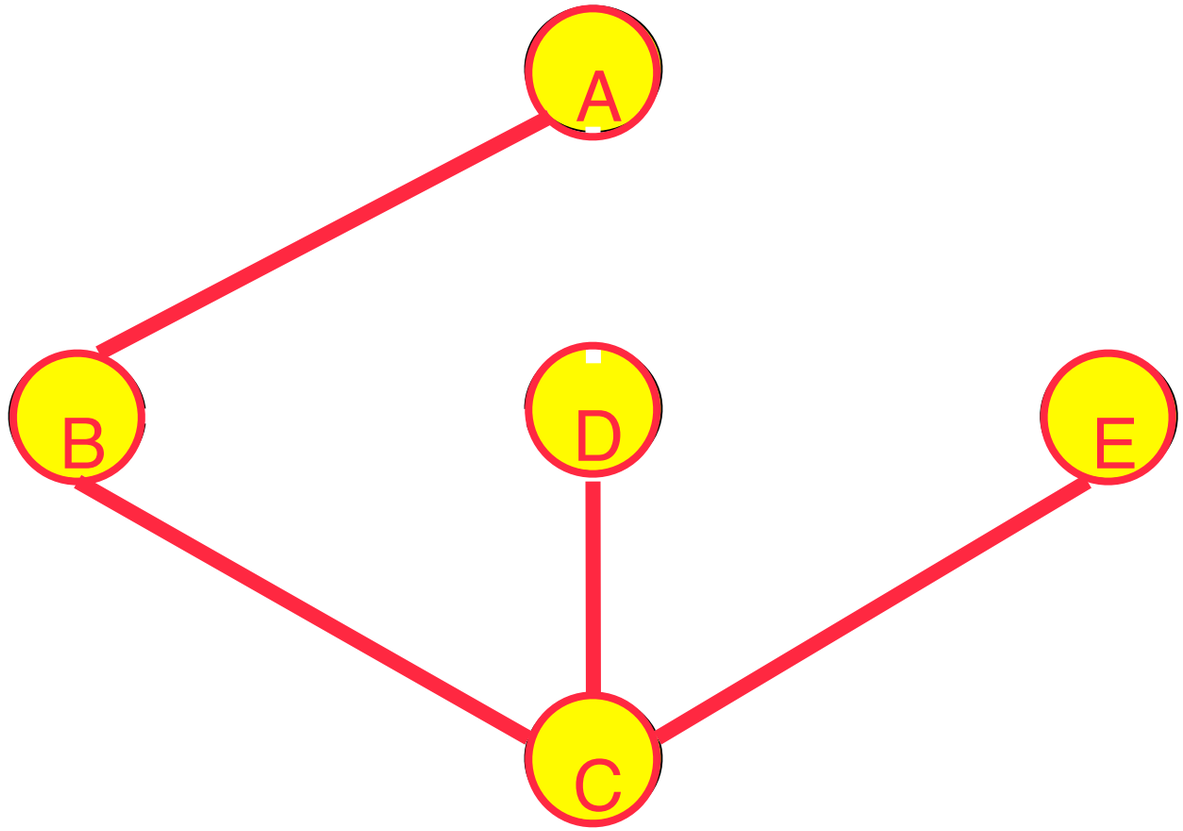
- If we perform this process on an arbitrary graph, we need to be careful to avoid cycles.
- To do this, when we visit a vertex v, we mark it visited, since now we have been there, and recursively call depth-first search on all adjacent vertices that are not already marked.

Undirected Graph





DFS Example



BFS Example