Data Structures and Algorithms 2

Prof. Ahmed Guessoum
The National Higher School of AI

Chapter 8

Graphs

Main Points in this Chapter

- Show several real-life problems, which can be converted to problems on 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.
- Study minimal-path problems in a graph

Basic Definitions

- A **graph** $G = (V, E)$ consists of a set of **vertices,** 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 the pair is ordered, then the graph is **directed.** Directed graphs are sometimes referred to as **digraphs.**
- Vertex w is **adjacent** to v if and only if $(v, w) \in E$.
- 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.**

Basic Definitions

- 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, i.e. $N - 1$.
- We allow a path from a vertex to itself; if this path contains no edges, then the path length is 0.
- 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. (**The graphs we will consider will generally be loopless.)
- A **simple path** is a path such that all vertices are distinct (i.e. no cycles), except that the first and last vertices could be the same.

Basic Definitions

- 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, the edges must be distinct. (Note 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.)
- 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.**

Basic Definitions

- 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.**
- **Weakly connected graph:** a directed graph not strongly connected, but the underlying graph (without direction to the arcs) is connected**.** i.e. it is possible to reach any node starting from any other node by traversing edges in some direction (i.e., not necessarily in the direction they point).
- **Complete graph:** a graph in which there is an edge between every pair of vertices.

Practical Examples of Graphs

Airport system: 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 (logically) directed, since it might take longer or cost more to fly in different directions.
- Ideally, the airport system is strongly connected, so that it is always possible to fly from any airport to any other airport.
- We might like to quickly determine the *best* flight between any two airports. "Best": fewest number of edges or could be taken with respect to one, or all, of the weight measures.

Traffic flow: Each street intersection represents a vertex, and each street is an edge.

- The edge costs could represent a speed limit, a capacity (number of lanes), etc.
- We could ask for the shortest route or find the most likely location for bottlenecks.

Representation of Graphs

- A simple way to represent a graph is to use a two-dimensional array known as an **adjacency matrix** representation.
- For each edge (u, v), set $A[u][v]$ to true; otherwise to false.
- If the edge has a weight associated with it, then one can set $A[u][v]$ to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.
- Example, if we were looking for the cheapest airplane route, we could represent nonexistent flights with a cost of ∞. For the most expensive airplane route, we could use $-\infty$ to represent nonexistent edges.
- **Adjacency matrix** representation:
  - Advantage: extreme simplicity
  - Disadvantage: the space requirement is $\Theta(|V|^2)$ which can be prohibitive if the graph does not have very many edges.
- An adjacency matrix is an appropriate representation if the graph is **dense:** $|E| = \Theta(|V|^2)$.
- In most applications, not true. E.g.: a graph representing a street map. If 3000 intersections (3000 vertices), ~12000 edge entries, but array of size 9,000,000.

Alternative Graph Representation

- If the graph is **sparse** (i.e. not dense), a better solution is an **adjacency list** representation. It is the standard way to represent graphs.
- For each vertex, we keep a list of all adjacent vertices.
- The space requirement is then $O(|E| + |V|)$, which is linear in the size of the graph.
- If the edges have weights, then this additional information is also stored in the adjacency lists.
- Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage doubles.
- A common requirement in graph algorithms is to find all vertices adjacent to some given vertex v. This can be done, in time proportional to the number of such vertices found, by a simple scan down the appropriate adjacency list.

Adjacency List Representation of a Graph

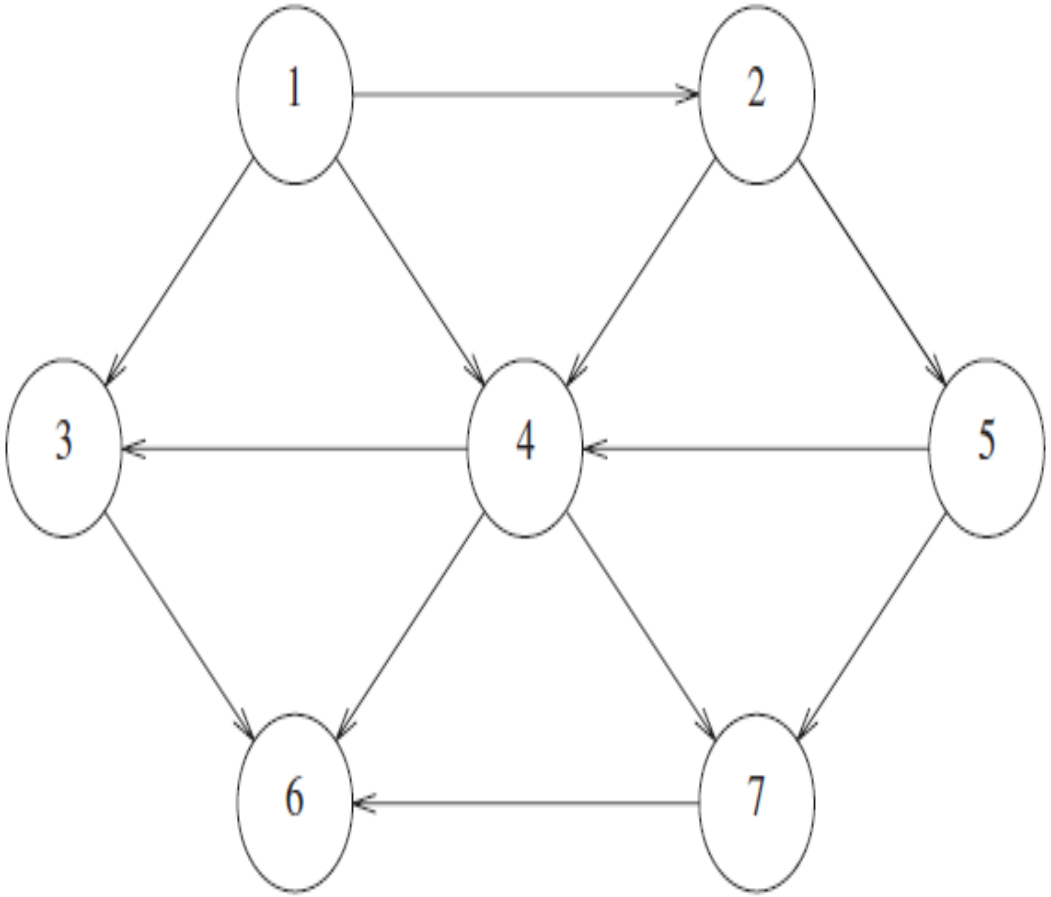


Figure 9.1 A directed graph

1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

Figure 9.2 An adjacency list representation of a graph

Maintaining the Adjacency Lists

- Several alternatives: the lists can be maintained in either vectors or lists.
- For sparse graphs, when using vectors, initialize each vector with a smaller capacity than the default to avoid wasted space.
- Quickly obtaining the list of adjacent vertices for any vertex is important, so the two basic options are
  1. to use **a map** in which the keys are vertices and the values are adjacency lists; or
  2. to maintain each adjacency list as a data member of a Vertex class.
- In the second scenario, if the vertex is a string (e.g. an airport name, or name of a street intersection), then a map can be used in which the key is the vertex name and the value is a Vertex (typically a pointer to a Vertex), and each Vertex object keeps a list of (pointers to the) adjacent vertices and perhaps also the original string name.

Topological Sort

- A **topological sort** is an ordering of vertices in a directed acyclic graph, such that if there is a path from *vi* to *vj*, then *vj* appears *after vi* in the ordering.
- The graph in Figure 9.3 represents the course prerequisite structure at a university.
- 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.

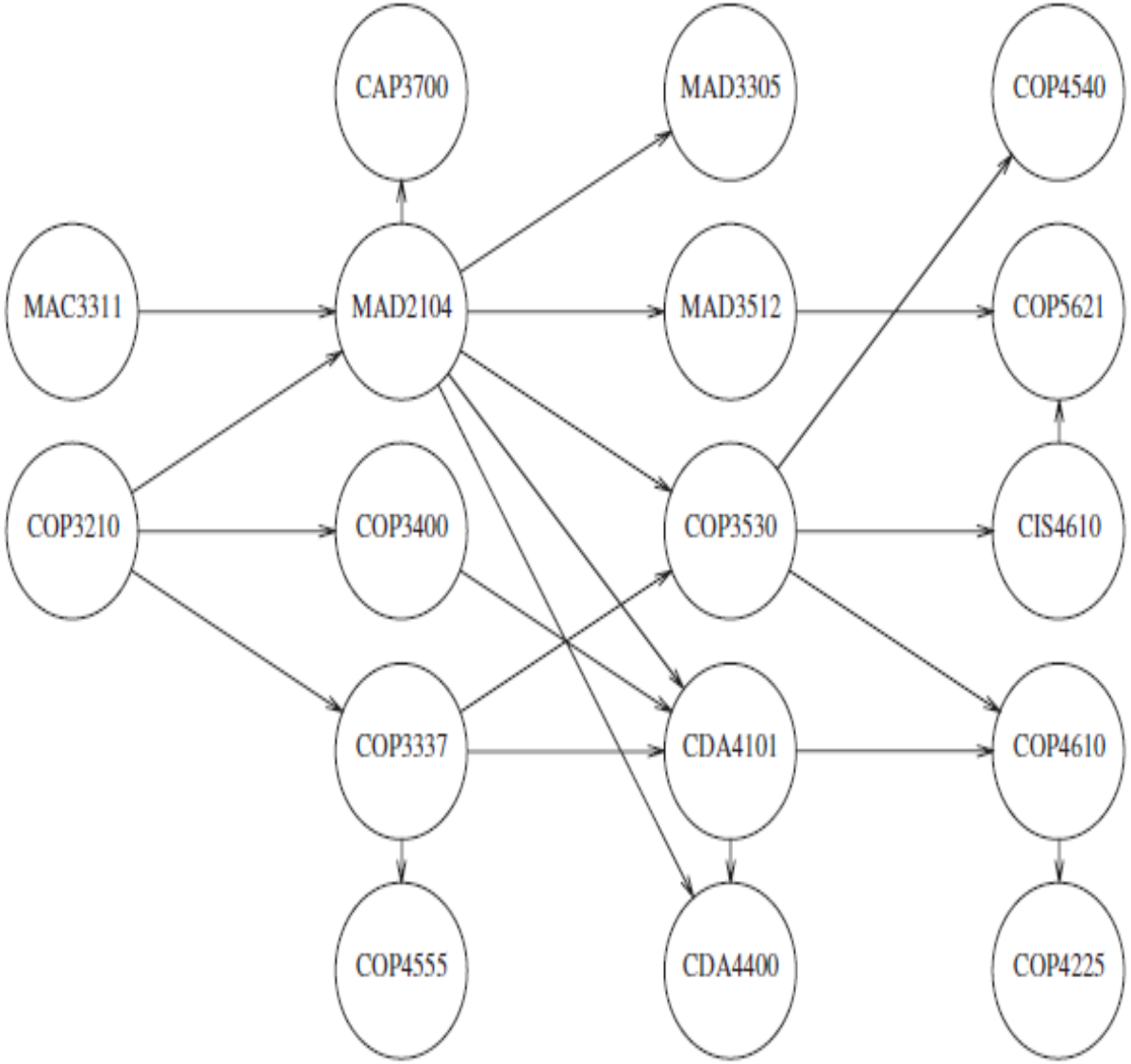


Figure 9.3 An acyclic graph representing course prerequisite structure

Topological Sort

- Topological ordering is not possible if the graph has a cycle, since for two vertices *v* and *w* on the cycle, *v* precedes *w* and *w* precedes *v*.
- The ordering is not necessarily unique; any legal ordering will do.
- In Figure 9.4, *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.
- Simple algorithm to find a topological ordering:
  1. Find any vertex with no incoming edges. Print it and remove it, along with its edges, from the graph.
  2. Apply this same strategy to the rest of the graph.
- Let us define the **indegree** of a vertex *v* as the number of edges (*u*, *v*).
- We compute the indegrees of all vertices in the graph.

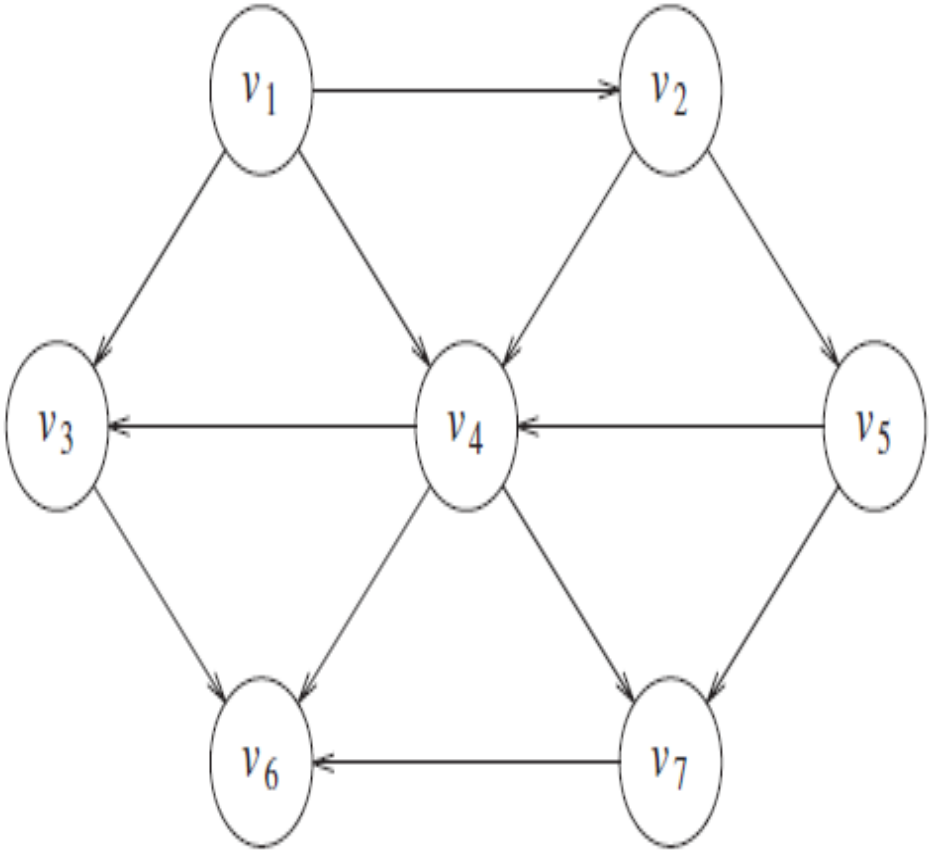


Figure 9.4 An acyclic graph

Topological Sort

1. The indegree is computed for every vertex.
2. All vertices of indegree 0 are placed on an initially empty queue (a stack could alternatively be used).
3. While the queue is not empty, a vertex *v* is removed, and all vertices adjacent to *v* have their indegrees decremented.
4. A vertex is put on the queue as soon as its indegree falls to 0.
5. The topological ordering then is the order in which the vertices dequeue.

[Topological sort algorithm]

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

Figure 9.6 Result of applying topological sort to the graph in Figure 9.4

Complexity of the Topological Sort Algorithm

- Time complexity of this algorithm is $O(|E| + |V|)$ if adjacency lists are used.
- Indeed, the body of the for loop is executed at most once per edge.
- Computing the indegrees can be done with the code below; the same logic shows that the cost of this computation is $O(|E| + |V|)$

```
for each Vertex v
    v.indegree = 0;
for each Vertex v
    for each Vertex w adjacent to v
        w.indegree++;
```

- The queue operations are done at most once per vertex.
- The other initialization steps, including the computation of indegrees, also take time proportional to the size of the graph.

Shortest-Path Algorithms

- We now assume that the input is a weighted graph: Associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the edge.
- The **weighted path length** is the cost $\sum_{i=1}^{N-1} C_{i,i+1}$ of a path $v_1 v_2 \ldots v_N$
- The **unweighted path length** is the number of edges on the path, i.e. $N - 1$.
- **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: shortest weighted path from $v1$ to $v6$ is v_1, v_4, v_7, v_6 and has a cost of 6.
- The shortest unweighted path between v_1 and v_6 has a cost (length) of 2.

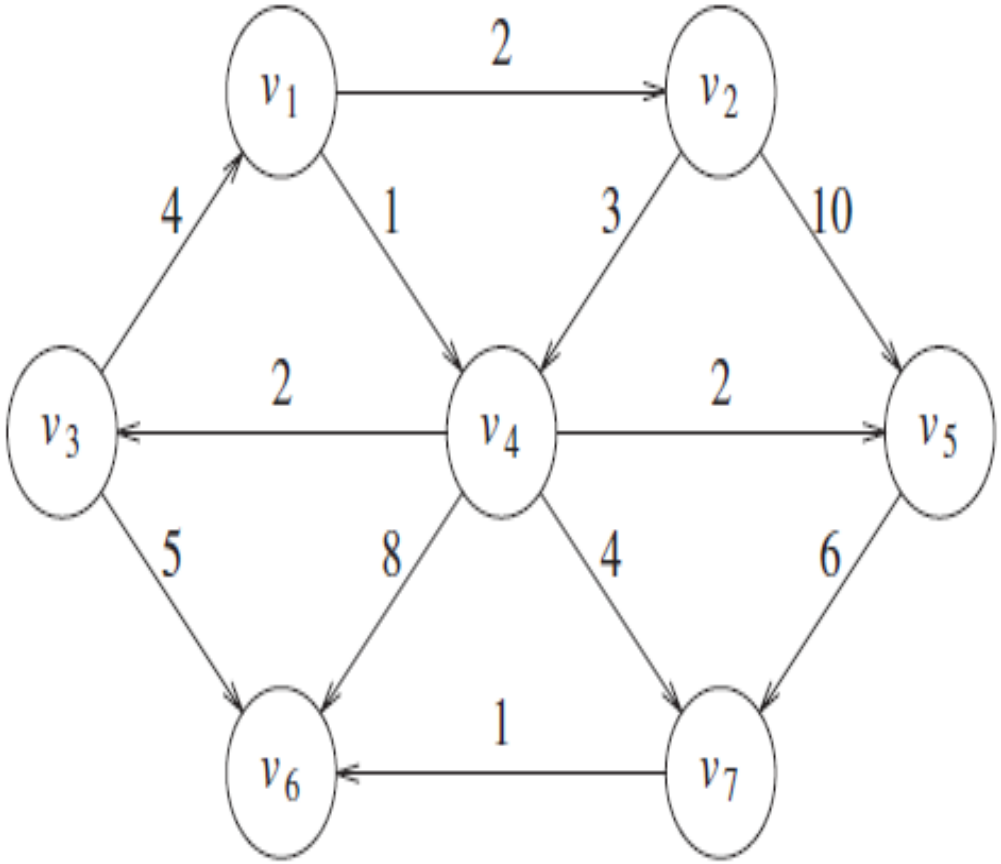


Figure 9.8 A directed graph G

Shortest-Paths & negative-cost cycles

- The graph in Figure 9.9 has an edge with negative cost. This can cause problems…
- Path *v*5 to *v*4 has cost 1, but the path (following the loop) *v*5, *v*4, *v*2, *v*5, *v*4 has cost −5.
- Note that we could stay in the loop arbitrarily long. Thus, the shortest path between these two points is undefined.
- Same applies to the shortest path from *v*1 to *v*6: undefined, as we can get into the same loop.
- This loop is known as a **negative-cost cycle;** when one is present, the shortest paths are not defined.
- For convenience, in the absence of a negative-cost cycle, the shortest path from *s* to *s* is zero.

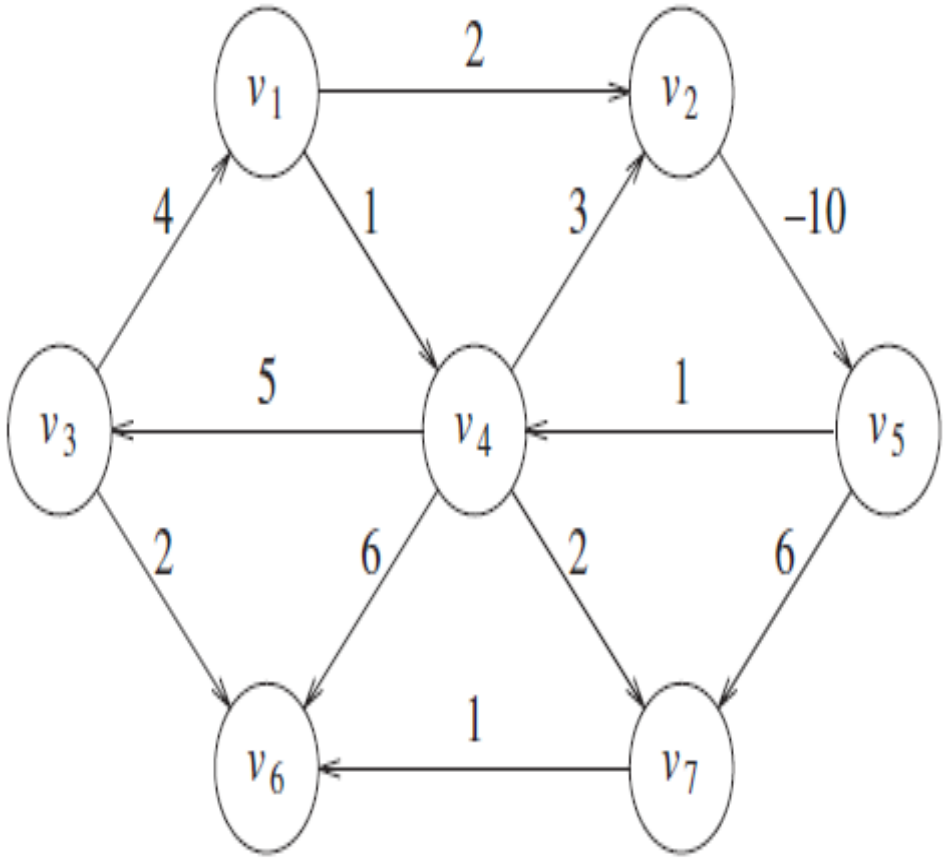


Figure 9.9 A graph with a negative-cost cycle

Example of Applications

Networks:

- **Vertices** represent computers; an edge represents a link between two computers;
- **Costs**: communication costs (phone bill per a megabyte of data), delay costs (number of seconds required to transmit a megabyte), or a combination of these and other factors,
- The **shortest-path algorithm** can be used to find the cheapest way to send electronic news from one computer to a set of other computers.
- **Modeling airplane (or other mass transit) routes** by **graphs**.
- Use a **shortestpath algorithm** to compute the best route between two points.
- In this and many practical applications, we might want to find the shortest path from one vertex, *s*, to only one other vertex, *t*.
- 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.

Unweighted Shortest Paths

- Given some vertex, *s*, find the shortest path from *s* to all other vertices of an unweighted graph.
- For now, suppose we are interested only in the length of the shortest paths, not in the actual paths themselves. (Easy to keep track of the actual paths.)
- If *s* is chosen as *v*3. Immediately, we can tell that the shortest path from *s* to *v*3 has length 0.
- We mark this information on the graph.
- Now we look 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*.
- One can see that *v*1 and *v*6 are one edge from *s*.

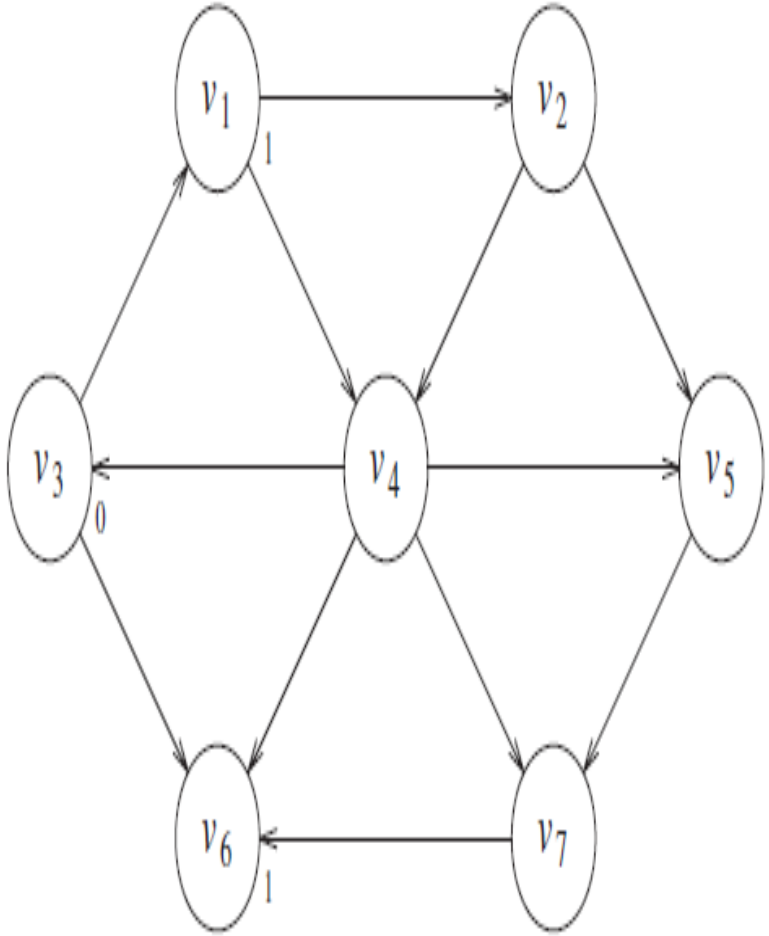


Figure 9.12 Graph after finding all vertices whose path length from *s* is 1

Unweighted Shortest Paths

- Now find vertices whose shortest path from *s* is exactly 2, by finding all the vertices adjacent to *v*1 and *v*6 (the vertices at distance 1), whose shortest paths are not already known.

  ➔ the shortest path to *v*2 and *v*4 is 2.

- Finally, examine the vertices adjacent to the recently evaluated *v*2 and *v*4.

  ➔ *v*5 and *v*7 have a shortest path of 3 edges.

- All vertices have now been calculated,
- This strategy for searching a graph is known as **breadth-first search**.

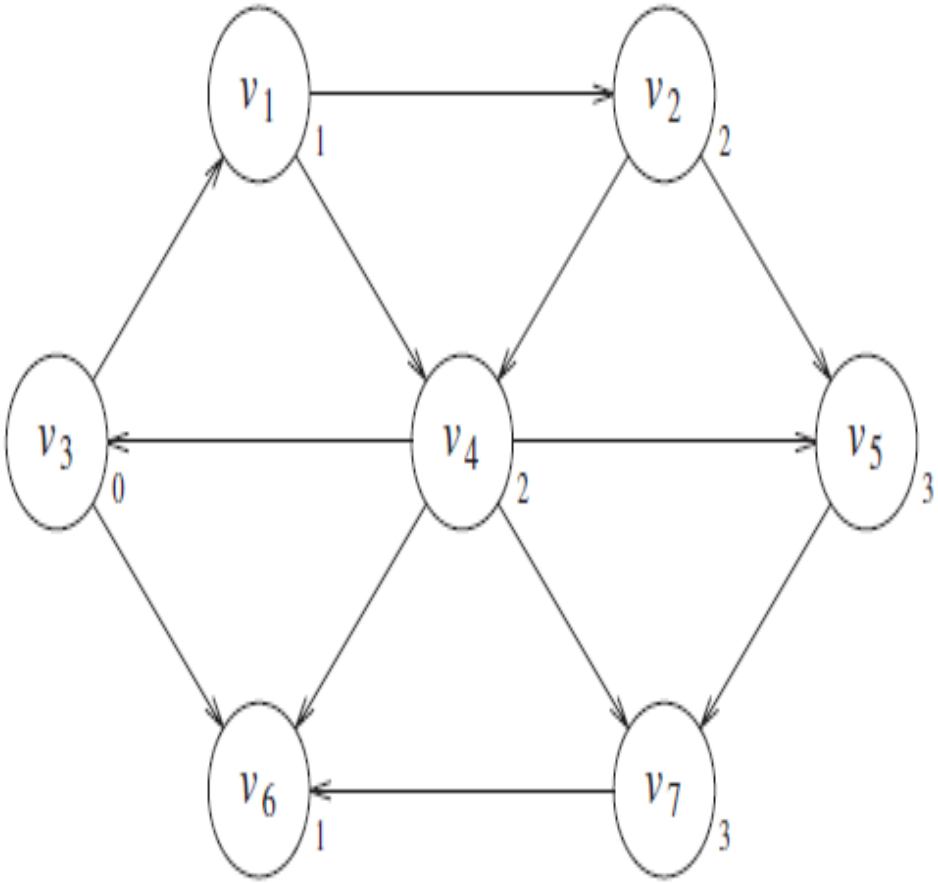


Figure 9.14 Final shortest paths

Unweighted Shortest Paths

Initial configuration of the table that the algorithm will use to keep track of its progress with 3 pieces of information for each vertex:

- Entry d_v : distance of vertex from *s*. Initially all vertices are unreachable except for *s*, whose path length is 0.
- Entry p_v: the bookkeeping variable; it will allow to print the actual paths.
- Entry *known*: set to true after a vertex is processed. (Initially, all entries are not *known*, including the start vertex. When a vertex is marked *known*, we know that no cheaper path will ever be found, and so processing for that vertex is essentially complete.)

  A more efficient implementation skips this *known*

Pseudocode for unweighted shortest-path algorithm

- Using the same analysis as was performed for topological sort, we get that the running time is $O(|E| + |V|)$, as long as adjacency lists are used.

v	Known	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.19 How the data change during the unweighted shortest-path algorithm

Dijkstra's Algorithm

- If the graph is weighted, the problem (apparently) becomes harder, but we can still use the ideas from the unweighted case.
- Each vertex is still marked as either *known* or *unknown*, distance *dv* is kept for each vertex, as well as *pv*, the last vertex to cause a change to *dv*.
- **Dijkstra's algorithm:** the general method to solve the single-source shortest-path problem**.** It is an 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. But do not guarantee to always get the best solution.
- At each stage, Dijkstra's algorithm :
  - selects a vertex, *v*, which has the smallest *dv* among all the *unknown* vertices;
  - declares that the shortest path from *s* to *v* is *known;*
  - updates the values of *dw*.

Dijkstra's Algorithm

- Recall the unweighted case: $dw = dv + 1$ if $dw = \infty$. i.e. the value of dw *is* lowered if vertex v offered a shorter path.
- In **Dijkstra's algorithm**, with the same logic, set $dw = dv + C_{v,w}$ if this new value for dw would be an improvement.
- i.e. the algorithm decides if it is a good idea or not to use v on the path to w.
- The cost calculated above is the cheapest path using v (and only *known* vertices).

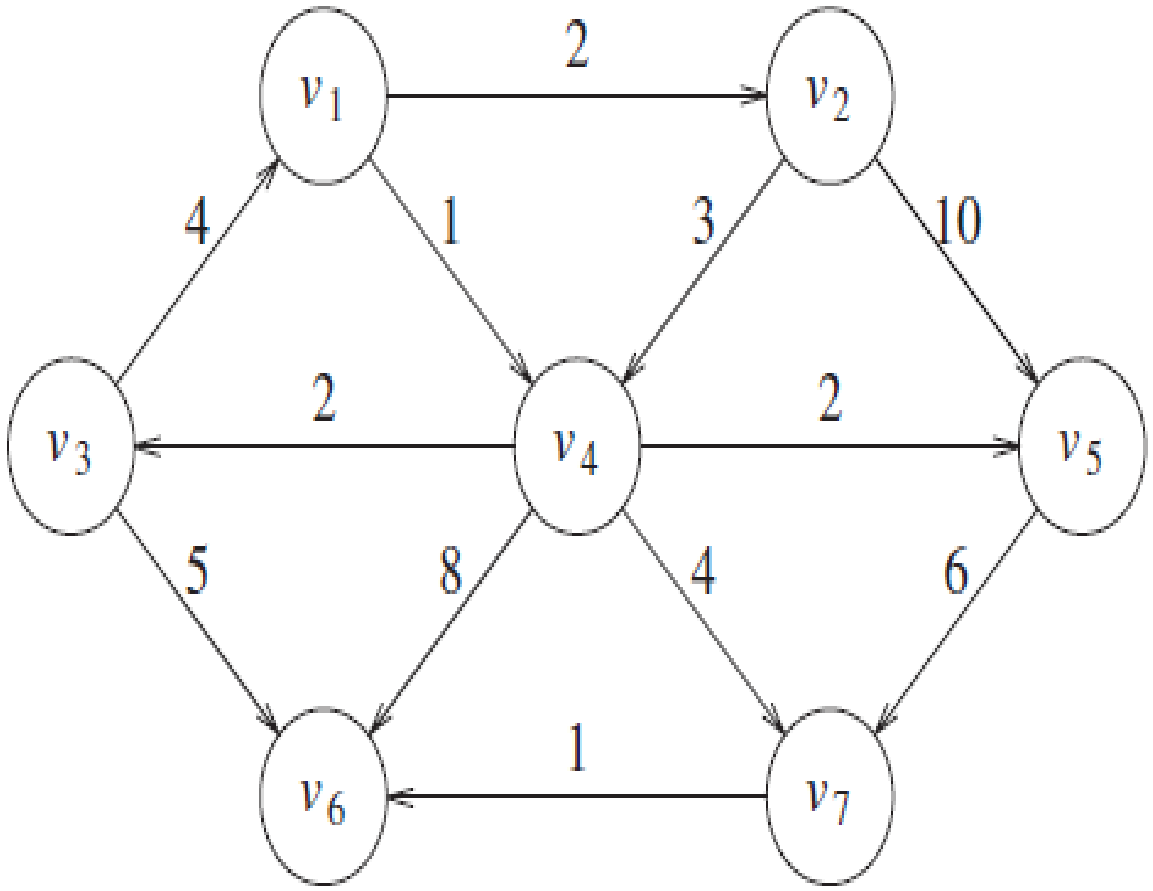


Figure 9.20 The directed graph G (again)

Dijkstra's Algorithm

- Assume that the start node, *s*, is *v*1.
- Figure 9.21 represents the initial configuration,
- The first vertex selected is *v*1, with path length 0. It is marked *known*.
- Now that *v*1 is *known*, some entries need to be adjusted. The vertices adjacent to *v*1 are *v*2 and *v*4.
- Both these vertices get their entries adjusted, as indicated in Figure 9.22.
- We continue like this….

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.22

Dijkstra's Algorithm

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

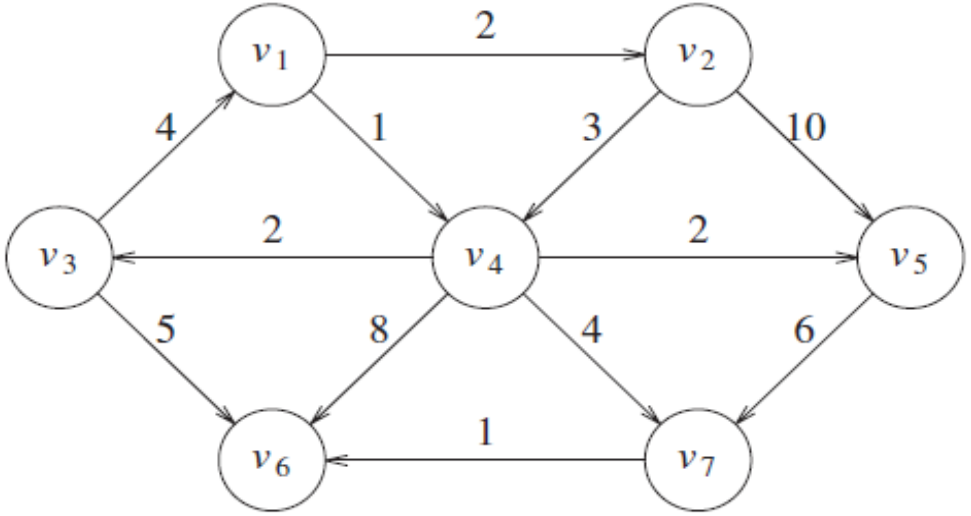
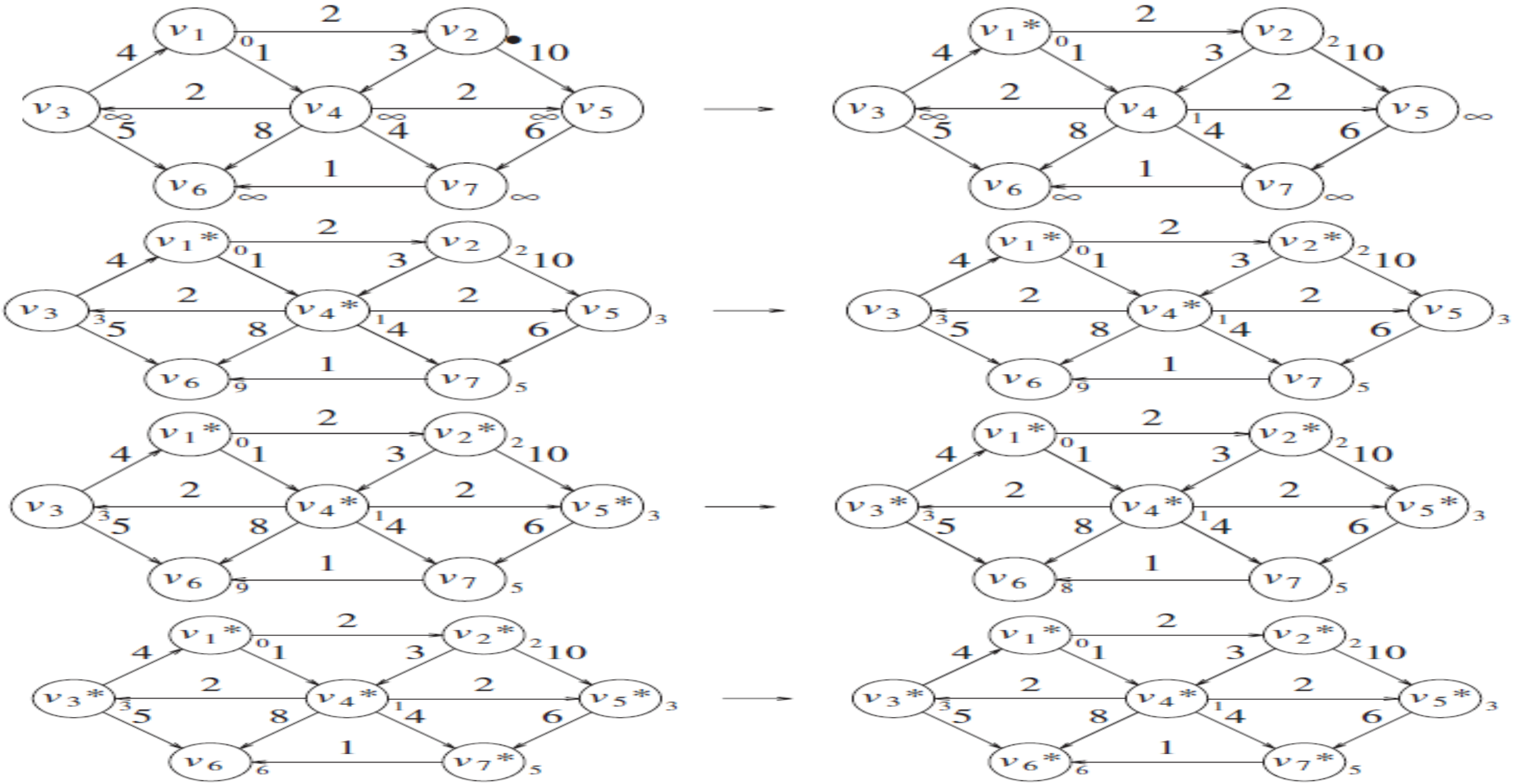


Figure 9.20 The directed graph G (again)

After v6 is declared known and algorithm terminates

Fig. 9.28: Stages of Dijkstra's algorithm



Pseudocode for Dijkstra's algorithm

- Vertex class for Dijkstra's algorithm (pseudocode) and printPath
- Pseudocode for Dijkstra's algorithm

Analysis of Dijkstra's algorithm

- A proof by contradiction will show that the algorithm always works as long as no edge has a negative cost. (If any edge has a negative cost, the algorithm could produce the wrong answer).
- The running time depends on how the vertices are manipulated.
- If the vertices are scanned sequentially to find the minimum *dv*, each phase will take $O(|V|)$ time to find the minimum *dv* vertex, and thus the algorithm will take $O(|V|^2)$ time for finding the minimum.
- The time for updating *dw* is constant per update, and there is at most one update per edge for a total of $O(|E|)$.
- Thus, the total running time is $O(|E|+|V|^2) = O(|V|^2)$.
- If the graph is dense, with $|E| = (|V|^2)$, this algorithm is not only simple but also essentially optimal, since it runs in time linear in the number of edges.

Analysis of Dijkstra's algorithm

- If the graph is sparse, with $|E| = (|V|)$, the algorithm is too slow.
- The distances would thus need to be kept in a priority queue.
- Two ways to do this
  1. Selection of the vertex *v* is a deleteMin operation, since once the unknown minimum vertex is found, it is no longer unknown and must be removed from future consideration.
  2. The update of w's distance can be implemented in two ways.
     a. Treat the update as a decreaseKey operation.
     b. Insert *w* and the new value d_w into the priority queue every time *w*'s distance changes. (There may be more than one representative for each vertex in the priority queue; handled appropriately)

     (See the textbook for the details)

     Both yield a O(|E| log |V|) time complexity to the algorithm.

Notes on Dijkstra's algorithm

- For the typical problems, such as computer mail and people (resp., goods) transportation (resp., delivery), the graphs are typically very sparse because most vertices have only a couple of edges.

➔ it is important in many applications to use a priority queue to solve this problem.

- There are better time bounds possible using Dijkstra's algorithm if different data structures are used.

Graphs with Negative Edge Costs

- If the graph has negative edge costs, then Dijkstra's algorithm does not work.
- The problem: once a vertex, *u*, is declared *known*, it is possible that from some other *unknown* vertex, *v*, there is a path back to *u* that is very negative such that taking a path from *s* to *v* back to *u* is better than going from *s* to *u* without using *v*.
- A working solution, but with a drastic increase in running time, is a combination of the weighted and unweighted algorithms:
  - No use of the concept of *known* vertices
  - Begin by placing *s* on a queue.
  - At each stage, dequeue a vertex *v*.
  - Find all vertices *w* adjacent to *v* such that $d_w > d_v + C_{v,w}$. Update d_w and p_w, and place *w* on a queue if it is not already there. (A bit can be set for each vertex to indicate presence in the queue.)
  - Repeat the process until the queue is empty.

Graphs with Negative Edge Costs

Pseudocode for weighted shortest-path algorithm with negative edge costs

- This algorithm works if there are no negative-cost cycles.
- It is no longer true that the code in the inner for loop is executed once per edge.
- Each vertex can *dequeue* at most $|V|$ times, so the running time is $\boldsymbol{O(|E| \cdot |V|)}$ if adjacency lists are used (Exercise 9.7(b)).
- This is quite an increase from Dijkstra's algorithm
- **BUT** it is fortunate that, in practice, edge costs are non-negative.
- If negative-cost cycles are present, then
  - the algorithm as written will loop indefinitely.
  - By stopping the algorithm after any vertex has *dequeued* $|V| + 1$ times, we can guarantee termination.

Acyclic Graphs

- If the graph is known to be acyclic, Dijkstra's algorithm can be improved by changing the order in which vertices are declared *known*, called 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.
- This selection rule works because when a vertex *v* is selected, its distance, *dv*, can no longer be lowered, since by the topological ordering rule it has no incoming edges emanating from *unknown* nodes.
- There is no need for a priority queue with this selection rule;
- The running time is $O(|E| + |V|)$, since the selection takes constant time.

Examples of Applications of Acyclic Graphs

- Modelling some downhill skiing problem: getting from point *a* to *b*, but can only go downhill, so clearly there are no cycles.
- Modelling of (nonreversible) chemical reactions:
  - each vertex could represent a particular state of an experiment,
  - edges would represent a transition from one state to another, and
  - the edge weights might represent the energy released.
  - If only transitions from a higher energy state to a lower one are allowed, the graph is acyclic.
- **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.
  - The edges represent precedence relationships: An edge (*v*, *w*) means that activity *v* must be completed before activity *w* may begin. ➔ the graph must be acyclic.
  - It is assumed that any activities that do not depend (directly or indirectly) on each other can be performed in parallel by different servers.

Applications of Acyclic Graphs – Critical Path Analysis

Critical path analysis:

- Such a graph could be (and frequently is) used to model construction projects.
- Important questions:
  - What is the earliest completion time for the project? E.g. We can see from the graph that 10 time units are required along the path *A*, *C*, *F*,*H*.
  - Which activities can be delayed, and by how long, without affecting the minimum completion time?
    E.g. delaying any of *A*, *C*, *F*, or *H* would push the completion time past 10 units.

Note that activity *B* is less critical and can be delayed up to two time units without affecting the final completion time.

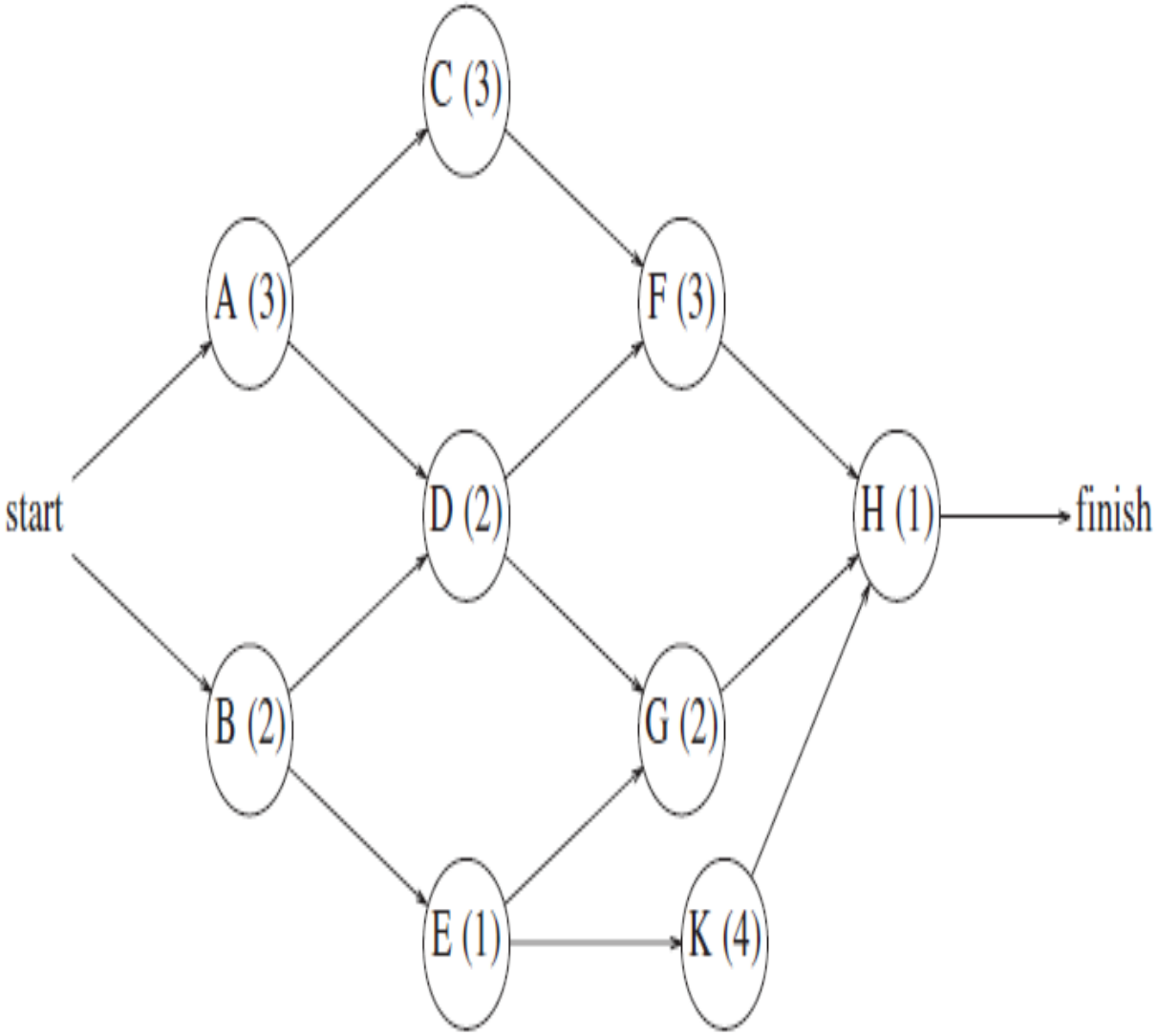


Figure 9.33 Activity-node graph

Applications of Acyclic Graphs – Critical Path Analysis

- To perform these calculations, the activity-node graph is converted to an **event-node graph**.
- Each event corresponds to the completion of an activity and all its dependent activities.
- Events reachable from a node *v* in the event-node graph may not commence until after the event *v* is completed.
- This graph can be constructed automatically or by hand.
- Dummy edges and nodes may need to be inserted in the case where an activity depends on several others.
- This is necessary in order to avoid introducing false dependencies (or false lack of them).
- The event-node graph corresponding to the previous graph is shown in Figure 9.34.

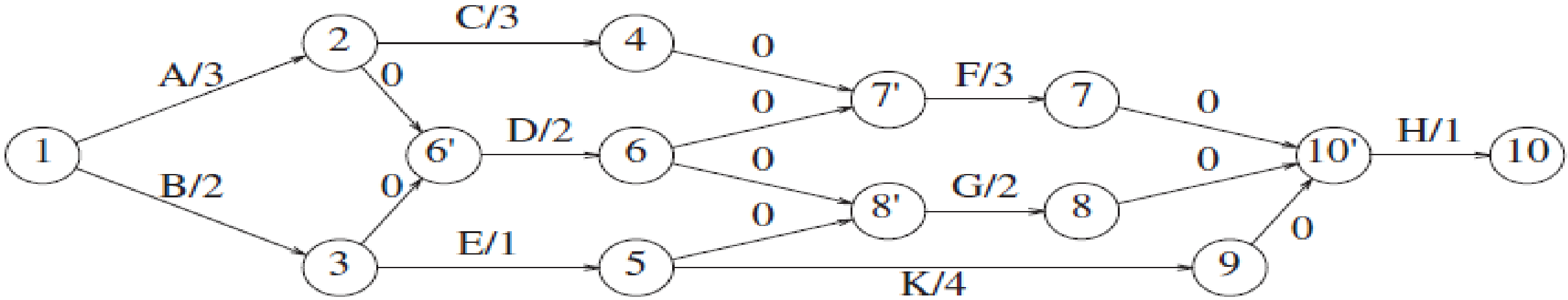


Figure 9.34 Event-node graph

Critical Path Analysis - Event-node graph

- To find the earliest completion time of the project, simply find the length of the *longest* path from the first event to the last event.
- Since the event-node graph is acyclic, we need not worry about cycles.
- In this case, it is easy to adapt the shortest-path algorithm to compute the *earliest completion time* for all nodes in the graph.
- If EC_i is the **earliest completion time** for node i, then the applicable rules are

$$EC_1 = 0 \quad \text{and} \quad EC_w = max_{(v,w) \in E} (EC_v + C_{v,w})$$

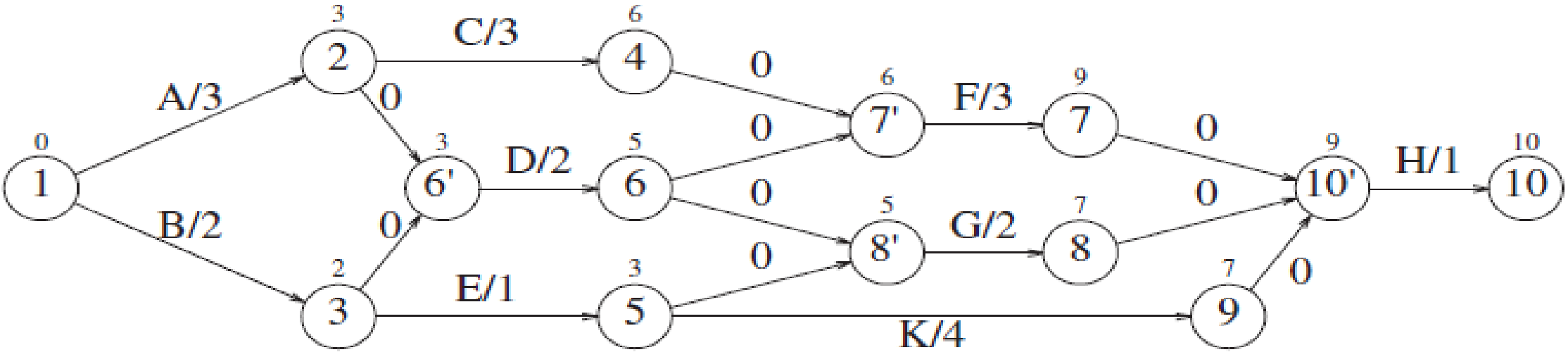


Figure 9.35 Earliest completion times

Critical Path Analysis - Event-node graph

- One can also compute the **latest completion time**, LC_i, that each event can finish without affecting the final completion time. The formulas to do this are:

$$LC_n = EC_n \quad and \quad LC_v = min_{(v,w) \in E} (LC_w - C_{v,w})$$

- These values can be computed in linear time by maintaining, for each vertex, a list of all adjacent and preceding vertices.
- The earliest completion times are computed for vertices by their topological order, and the latest completion times are computed by reverse topological order.

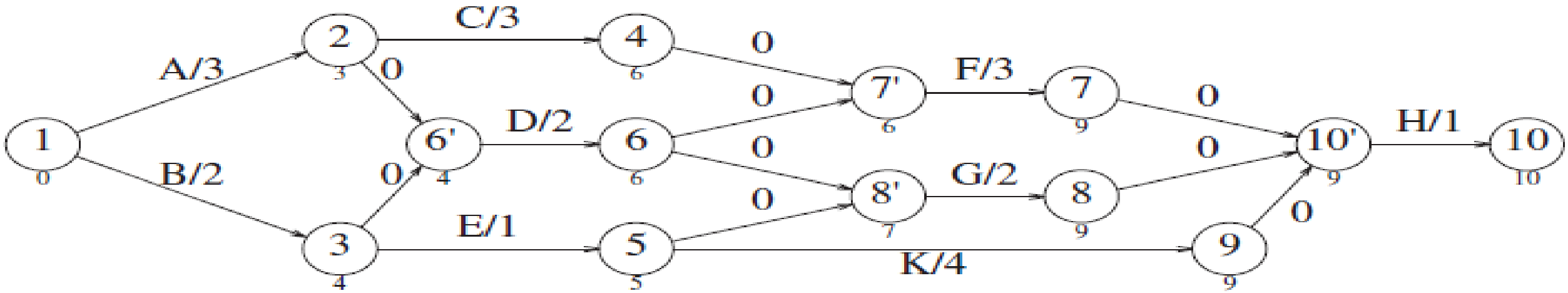


Figure 9.36 Latest completion times

Critical Path Analysis - Event-node graph

- The slack time for each edge in the event-node graph represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion.
- It is easy to see that $Slack_{(v\ w)} = LC_w - EC_v - C_{v,w}$
- In the figure, for each vertex: Earliest completion time: the top number; Latest completion time: the bottom entry. Slack time: the 3rd entry in the edge label.
- Some activities have zero slack. These are critical activities, which must finish on schedule.
- There is at least one path consisting entirely of zero-slack edges; such a path is a **critical path.**

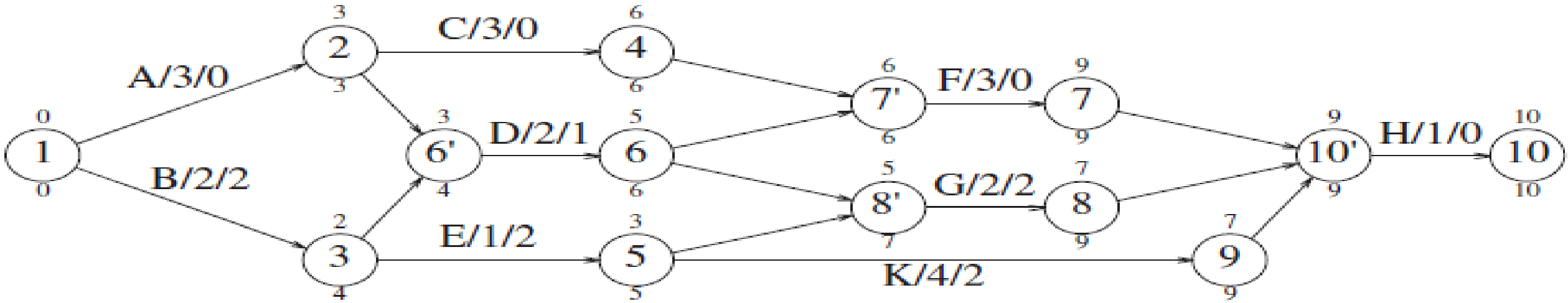
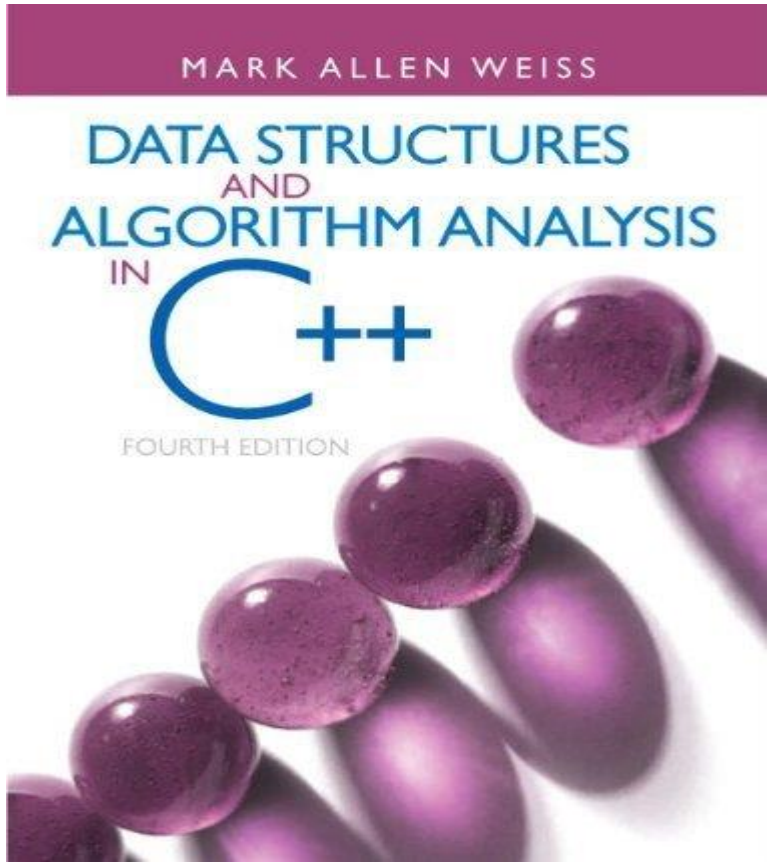


Figure 9.37 Earliest completion time, latest completion time, and slack

Slides based on the textbook



Mark Allen Weiss, (2014) Data Structures and Algorithm Analysis in C++, 4th edition, Pearson.

Acknowledgement: This **course PowerPoints** make substantial (non-exclusive) use of the PPT chapters prepared by Prof. Saswati Sarkar from the University of Pennsylvania, USA, themselves developed on the basis of the course textbook. Other references, if any, will be mentioned wherever applicable.