ECE365: Data Structures and Algorithms II (DSA 2)

Graph Algorithms

Graphs

- A **graph**, G, consists of a set of **vertices** (a.k.a. *nodes*, *points*, or *items*), V, and a set of **edges** (a.k.a. *arcs*, *lines*, *links*, or *connections*), E
- Every edge can be expressed as a pair (v, w) such that v, w ε V
- By convention, you typically do not repeat the same edge twice in the set of edges, E
- Sometimes every edge will have an associated **weight** or *cost*; if so, we call the graph a *weighted graph*; otherwise, it is an *unweighted graph*
- Edges can be **directed** or **undirected**
- If the edges are directed, then we are dealing with a *directed graph*, a.k.a. a *digraph*; otherwise, we are dealing with an *undirected graph*
- It is also possible to have a mixed graph which contains both directed and undirected edges, although this is not common

Adjacency

- Vertex w is said to be **adjacent** to vertex v if and only if (v, w) ε E
- If the graph is undirected, and w is adjacent to v, then v will also be adjacent to w
- In a directed graph, it is possible for w to be adjacent to v without v being adjacent to w
- An edge connecting two vertices is said to be incident on the vertices
- You can talk about the *in-degree* or *out-degree* of a vertex in a directed graph
- The in-degree is the number of directed incoming edges coming into a vertex
- The out-degree is the number of directed outgoing edges leaving a vertex

Paths

- A **path** is a sequence of vertices $w_1, w_2, \ldots, w_N$ such that $(w_i, w_{i+1}) \varepsilon E$ for $1 \leq i < N$
- The *length* of the path is the number of edges on the path, which is N-1 above
- Our textbook allows a path from a vertex to itself; if this path contains no edges, then the path has length 0
- If there is an edge from a vertex to itself, 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 vertex and last vertex can be the same
- Some sources define a simple path to mean that all vertices are distinct, and they do not allow the first and last vertices to be the same

Cycles

- 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 (recall that our textbook allows the first and last vertex to be the same in a simple path)
- For undirected graphs, the definition of a cycle should also include the fact that the edges are all distinct
- For example, the path u, v, u in an undirected graph is not generally considered to be a cycle
- In a directed graph, if (u, v) and (v, u) are both edges, then the path u, v, u does represent a cycle
- A directed graph is *acyclic* if it has no cycles; a *directed acyclic graph* is also known as a *DAG*

Connected Graphs

- An undirected graph is **connected** if there is a path from every vertex to every other vertex
- A directed graph with this property is said to be *strongly connected*
- If a directed graph is not strongly connected, but the underlying undirected graph is connected, then the directed graph is said to be *weakly connected*
- An undirected graph that is not connected can be divided into *connected components*, which are maximal connected subgraphs
- This means that there is no path from a vertex in one subgraph to a vertex in another subgraph (but all vertices within a subgraph are connected)
- If there is an edge between every pair of vertices, the graph is said to be *complete*

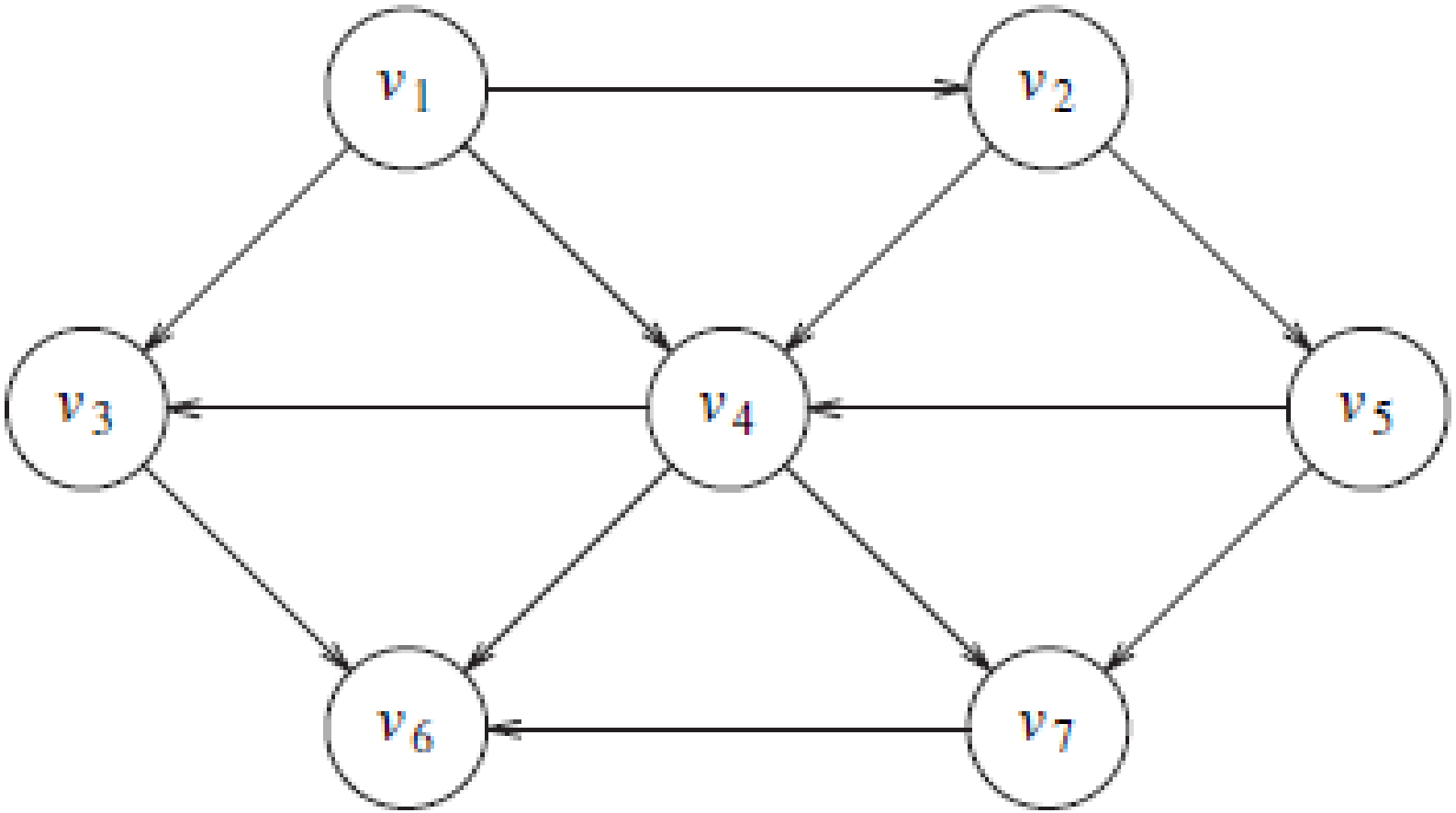
The Importance of Graph Algorithms

- We will see on the next slide that graphs can represent many important concepts
- Many DSA textbooks devote significantly more space than ours does to graph algorithms
- For example, consider the other DSA textbooks that I frequently reference:
  - "Algorithms, 3rd Edition, in C" (not the current edition) by Sedgewick is an extreme example
    - The textbook is divided into five "parts", with the fifth part being titled "Graph Algorithms"
    - This part of the book was so large that it was bound and published as a separate book, and sold separately
    - The other four parts, about the rest of DSA, was bound as a single book
  - "Algorithms, 4th Edition" by Sedgewick (only available for Java) is bound as a single book, and has only a single chapter on "Graphs", but that chapter is almost 200 pages long
  - "Introduction to Algorithms, 2nd Edition" by Cormen, Leiserson, Rivest, and Stein devotes five chapters to graph algorithms

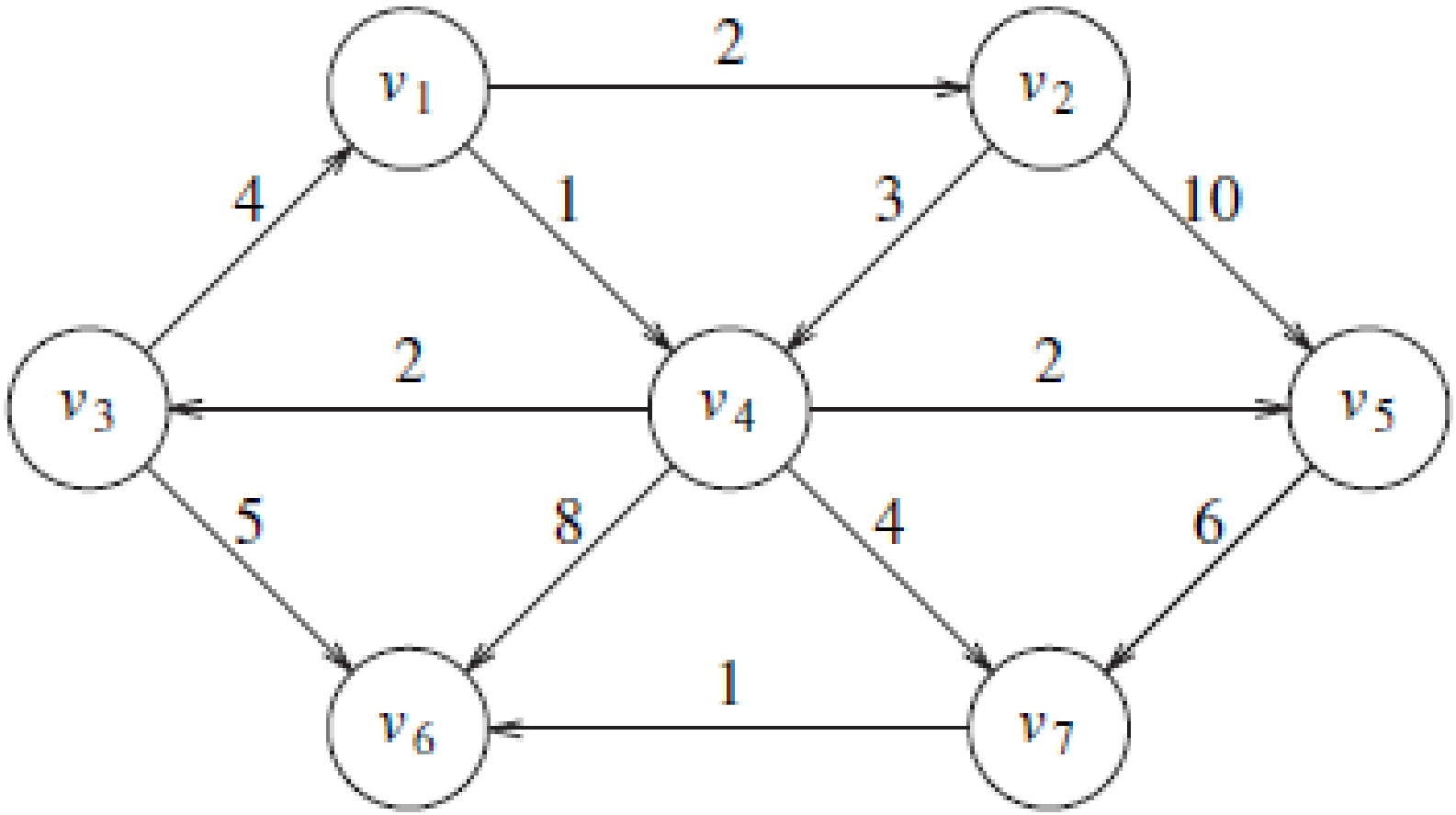
Applications of Graphs

- Some real-world concepts that can be represented as graphs include:
  - *Maps* - vertices may be cities, towns, street corners, etc.; edges might be streets or roads; or maybe they connect adjacent cities
  - *The World Wide Web* - vertices are the available web pages; edges are links; graph algorithms are essential components to WWW search engines
  - *Circuits* - vertices represent devices such as transistors, resistors, and capacitors; edges represent wires
  - *Schedules* - vertices represent tasks to be performed; edges represent constraints (e.g., that certain tasks can not start until others have finished)
  - *Transactions* - telephone example: vertices represent households and edges represent calls; finance example: vertices represent accounts and edges represent cash transfers
  - *Matching* - example: vertices represent candidates for jobs and available positions; edges represent assignments or potential assignments
  - *Networks* - vertices represent machines and edges represent direct connections between machines
  - *Program structure* - vertices represent functions and directed edges represent potential functions calls; used by compilers and other code analysis tools
- This list comes from " Algorithms, 3rd Edition, in C " by Sedgewick (mentioned on the previous slide)

Directed Graph Example (unweighted)



Directed Graph Example (weighted)



Analyzing Graph Algorithms

- When analyzing data structures and algorithms involving graphs, we will consider the number of both vertices and edges; that is |V| and |E|
- Recall that V and E are sets (of vertices and edges, respectively)
- The vertical bars around a set represent the size of the set, or the number of items in the set (in this case, the number of vertices or edges)
- When using big-Oh notation, as a shorthand, I may sometimes write V and E without the vertical bars, but this is just shorthand
- For example, O(V*E) should really be O(|V|*|E|)
- Of course, these two sizes (|V| and |E|) are related
  - For an undirected graph, $|E| \leq |V| * (|V| - 1) / 2$, assuming we do not allow edges from a vertex to itself, and we do not allow duplicate edges
  - For a directed graph, $|E| \leq |V|^2$, assuming we allow an edge from a vertex to itself, which is more commonly allowed for a directed graph, but no duplicate edges

Representing Graphs: Adjacency Matrices

- One way to represent a graph is with a two-dimensional array known as an **adjacency matrix**
- For each edge (u, v) in an unweighted graph, the slot A[u][v] is set to 1; all other slots are set to 0
- If edges have associated weights or costs, the array slot is set to the weight for existing edges
- A constant that depends on the application can be used to represent non-existent edges, or a separate Boolean can be used in addition to the weight
- For example, if the graph represents available airline flights and the weight of an edge is the cost of a flight, non-existent flights might be represented by a cost of infinity
- This matrix has an obvious space requirement of $\Theta(|V|^2)$
- This is appropriate if we are dealing with a **dense graph**; this means that $|E| \approx |V|^2$
- If we are dealing with an undirected graph, you only have to store half of the matrix; that is, the half above or below (perhaps inclusively) the main diagonal

Representing Graphs: Adjacency Lists

- For most of the applications we consider, we will be dealing with **sparse graphs**; this means that $|E|$ is significantly less than $|V|^2$
- The book talks about a street map for a city with a Manhattan-like structure; for such a sparse graph, $|E| \approx 4 * |V|$
- For sparse graphs, a better solution is to use an **adjacency list** representation
- For each vertex, we keep a list of all adjacent vertices
- This can be done using an array or vector of linked lists
- Header nodes (the slots of the array) can store additional information about the vertices
- The space requirement is $\Theta(|E| + |V|)$, a huge benefit over adjacency matrices for sparse graphs
- If edges have associated weights or costs, this information is also kept in each edge node
- One advantage of an adjacency matrix over an adjacency list is that you can determine in quick, constant time whether an edge exists

Named Vertices

- If the names of vertices are arbitrary strings, each vertex name must be mapped to a location in memory
- A *hash table* can be useful to store this mapping
- For example, the hash table might map a name to a pointer to a slot in an array or vertex storing the appropriate header node for an adjacency list representation
- When creating a graph, whenever a vertex name is encountered, we check if the name has already been mapped to a number
- If not, we assign it one, and add it to the end of the sequence of graph nodes
- You might also need a way to map indexes an array or vector of header nodes back to vertex names
- This can be done with a separate array of vector of vertex names
- Alternatively, the header nodes of the adjacency list representation of the graph can store the vertex names or they can store pointers into the hash table

Topological Sort

- The first graph algorithm task we will discuss is a **topological sort**
- A topological sort is an ordering of the vertices in a DAG such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering
- Two other ways to look at this (not mentioned in book) are:
  - Given a DAG, number the vertices such that every directed edge points from a lower numbered vertex to a higher numbered vertex
  - Given a DAG, rearrange the vertices on a horizontal line such that all directed edges point from left to right
- It should be obvious that a topological ordering is not possible if the graph has a cycle
- Often, there will be multiple valid orderings; generally, when performing a topological sort, any one of the valid orderings is acceptable
- Examples of applications of topological sort include graphs where:
  - The vertices represent tasks, and the edges represent constraints on the ordering of tasks
  - The vertices represent courses, and the edges represent prerequisites

Topological Sort Pseudo-code

- Two useful terms that will be helpful (not mentioned until a later subtopic in the textbook):
  - A **source** is a vertex with no incoming edges (i.e., the in-degree is 0)
  - A **sink** is a vertex with no outgoing edges (i.e., the out-degree is 0)
- A simple algorithm (my pseudo-code on right):
  - Find any source
  - Display or number this vertex
  - Remove the newly numbered vertex (along with its outgoing edges) from the graph
  - Repeat this process for the rest of the graph
- If the algorithm fails, meaning not every vertex gets numbered, then there is a directed cycle in the graph
- An analysis shows the time complexity is $\Theta(|V| + |E|)$ *if adjacency lists are used*
- This is because the body of the inner for loop is executed at most once per edge
- The textbook shows C++-style pseudo-code

```
TopologicalSort (Graph G)
  initialize Q as empty queue
  for each vertex v in G
    if in-degree[v] == 0
      push v onto Q
  i ← 0
  while Q is not empty
    u ← pop(Q)
    i ← i + 1
    assign u the number i
    for each edge from vertex u to
vertex w
      in-degree[w] ← in-degree[w] - 1
      if in-degree[w] == 0
        push w onto Q
```

Shortest-path Algorithms

- Next, we will discuss **shortest-path algorithms**
- If we are dealing with a weighted graph, and each edge (v_i, v_j) has an associated cost $c_{i,j}$, then the cost of a path is the sum of the costs of its edges
- This *path cost* is also referred to as the *weighted path length*
- The *unweighted path length* is the number of edges on the path
- One problem we will consider is referred to by the textbook as the **single-source shortest-path problem**; we will discuss multiple versions of this problem
- 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
- The problem is simplest if we can assume there are no negative costs for any edge
- The problem may not have an answer at all if there can be a negative cost cycle

Unweighted Path Length

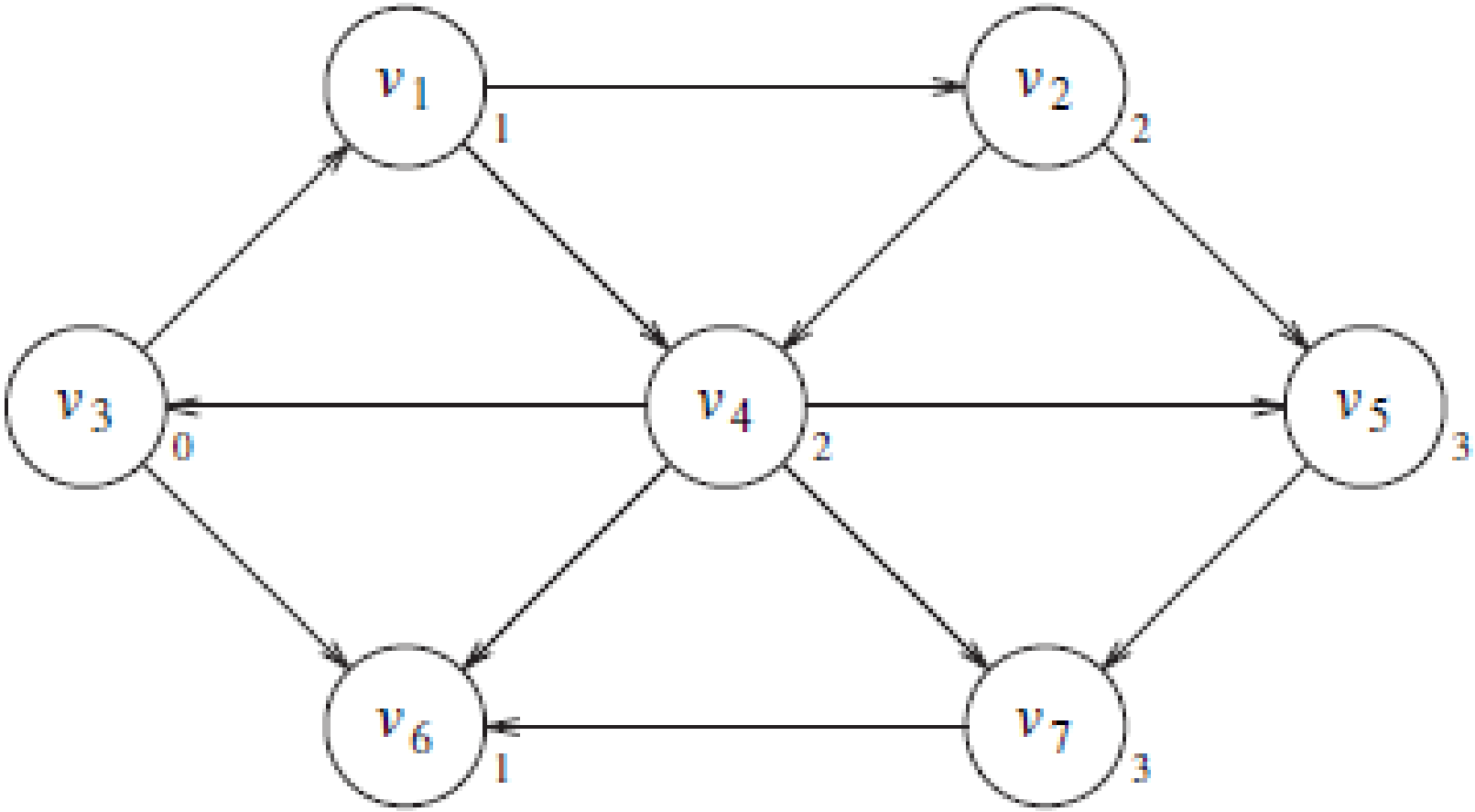
- The first algorithm we will consider assumes an unweighted graph
- We can define the path cost for an unweighted graph to be the number of edges on a path (i.e., we treat the graph as if every edge has a cost of one)
- A simple algorithm for this is as follows (pseudo-code is shown on the next page):
  - Assign the distance of the distinguished vertex, s, to 0 and push s onto a queue
  - As long as the queue is not empty, pop a vertex, v, from the queue
    - For every vertex w that can be reached from v, if the distance to w is not known:
      - Set the distance of w to be one greater than the distance of v, and consider w to be known
      - Push w onto the queue
- All we are really doing here is a *breadth-first search* using the starting node as the root
- With a slight modification, we can also retrieve the shortest unweighted path to each node

Unweighted Path Length Pseudo-code

- My pseudo-code is to the right
- At the end of the algorithm, each vertex stores its shortest distance
- The "prev" fields allow us to retrieve the shortest paths
- This running time is $\Theta(|E| + |V|)$ if adjacency lists are used
- That's the same time complexity as topological sort
- If an adjacency matrix is used, the running time is $\Theta(|V|^2)$

```
UnweightedPL (Graph G, Vertex s)
  initialize Q as empty queue
  for each vertex v in G
    v.distance ← ∞
  s.distance ← 0
  s.prev ← NULL
  push s onto Q
  while Q is not empty
    v ← pop(Q)
    for each edge from v to vertex w
      if w.distance == ∞
        w.distance ← v.distance + 1
        w.prev ← v
        push w onto Q
```

Unweighted Path Length Example (start at v_3)



Dijkstra’s Algorithm

- Next, we'll consider *a weighted graph with no negative-cost edges*
- We now want to find the shortest *weighted path* from a single source to all other nodes (as specified by the single-source shortest-path problem mentioned earlier)
- The algorithm we will cover is called **Dijkstra's algorithm**
- During the execution, we will keep track of vertices for which we have already computed the known minimum distance from the specified source, s
- We will refer to the vertices for which the final/shortest/best distances, d_v, are known as *known vertices*
- The vertices for which the final distances are not yet known are *unknown vertices*
- For each unknown vertex, v, we will also keep track of a tentative distance, d_v, which represents the shortest path from s to v using only known vertices
- We will also keep track of p_v, the previous vertex on the path to v leading to d_v, so that we will be able to recover the paths that led to the final computed distances

Dijkstra’s Algorithm Pseudo-code

- My pseudo-code for Dijkstra’s algorithm is to the right
- The start of Dijkstra’s algorithm performs initialization, including setting the distance to s to be 0
- The algorithm then proceeds through passes
- At the start of each pass, Dijkstra's algorithm selects the vertex v which has the smallest d_v among all vertices that do not already have their minimum cost decided
- I am purposely keeping that part of the algorithm vague (for now)
- This vertex becomes a known vertex; i.e., we assume its d_v has the correct final value
- The remainder of the pass cycles through the vertices, w, that are adjacent to v; i.e., the vertices for which (v, w) ε E
- The value of d_w is updated as: $d_w = \min(d_w, d_v + c_{v,w})$, where $c_{v,w}$ is the cost to travel from vertex v to vertex w
- The value of p_w is also updated if appropriate to v, meaning that v is the previous node on the best path to w found so far

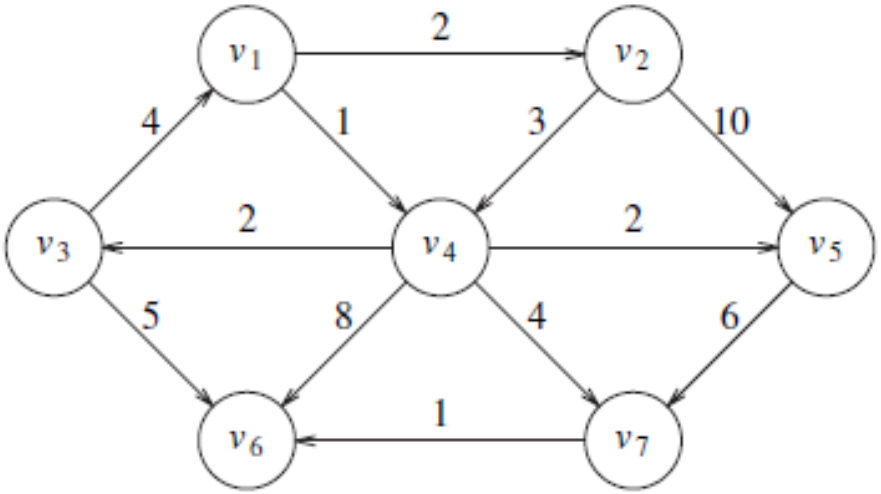
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (start at v_1)

v:

w:

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



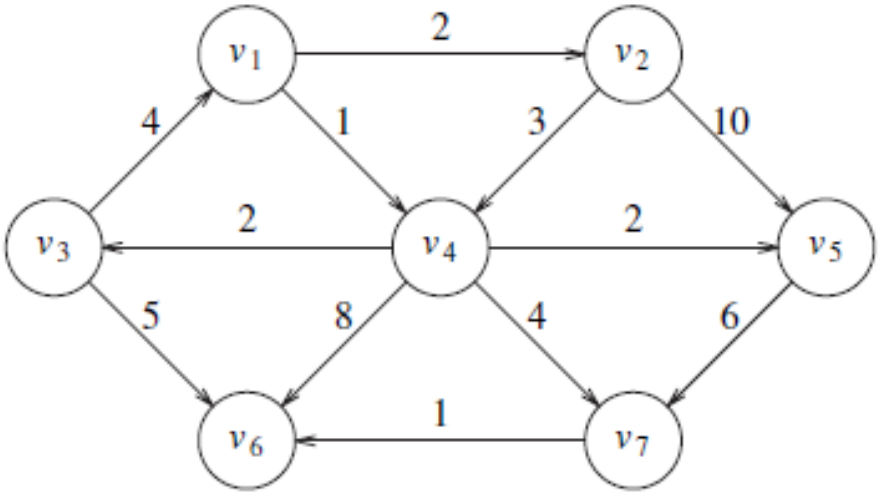

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
=> while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w:

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



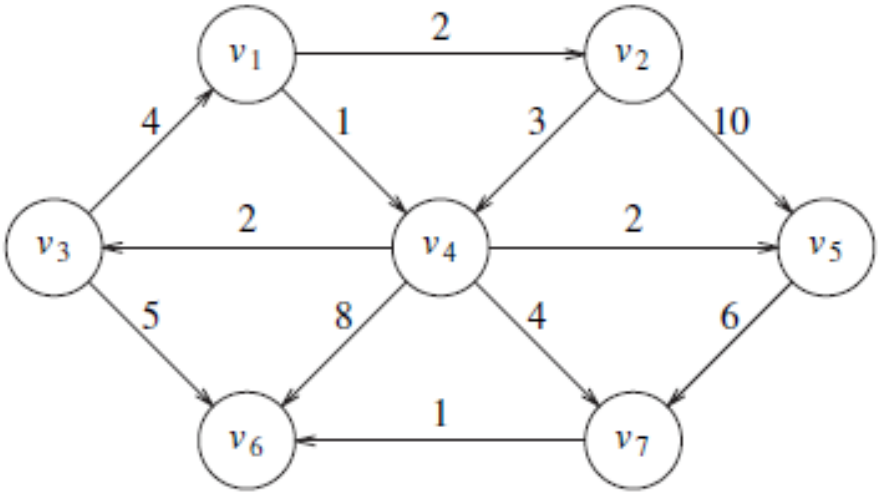

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
=>  v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	∞	
v_3	F	∞	
v_4	F	∞	
v_5	F	∞	
v_6	F	∞	
v_7	F	∞	



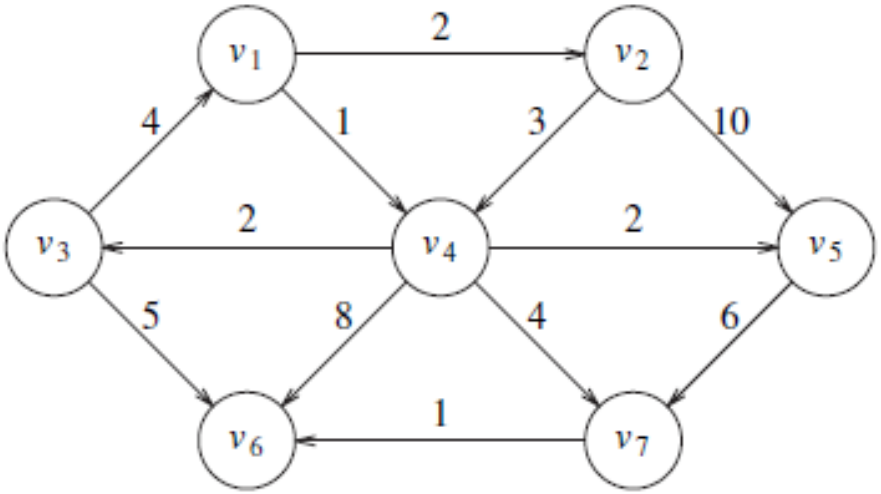

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
=>  known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_2

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	∞	
v_3	F	∞	
v_4	F	∞	
v_5	F	∞	
v_6	F	∞	
v_7	F	∞	



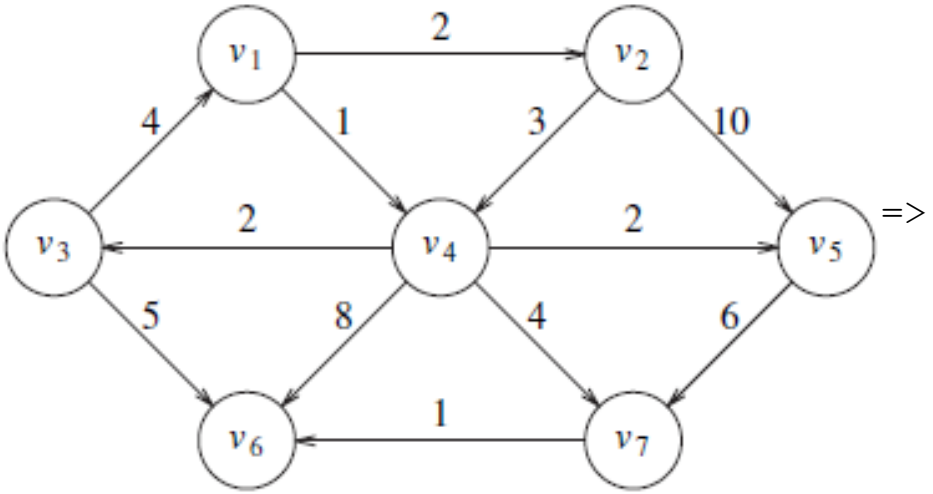

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_2

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	∞	
v_3	F	∞	
v_4	F	∞	
v_5	F	∞	
v_6	F	∞	
v_7	F	∞	



=>

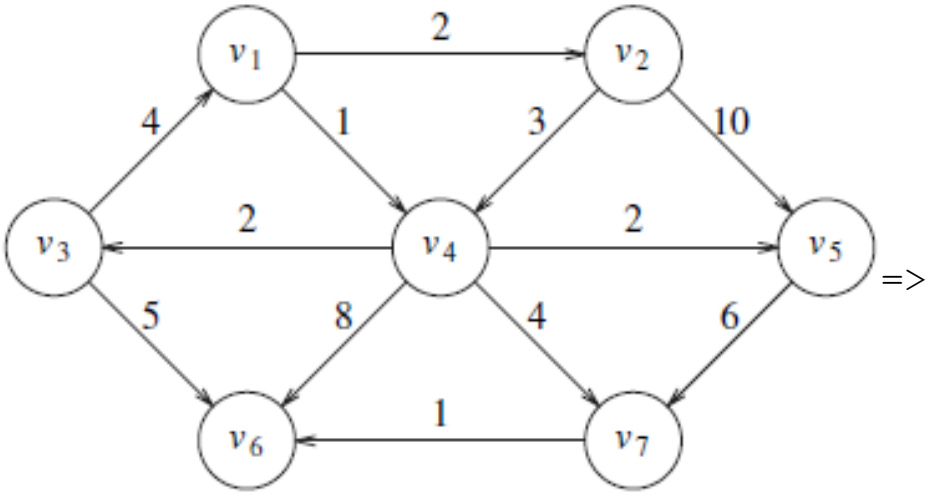
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_2

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



=>

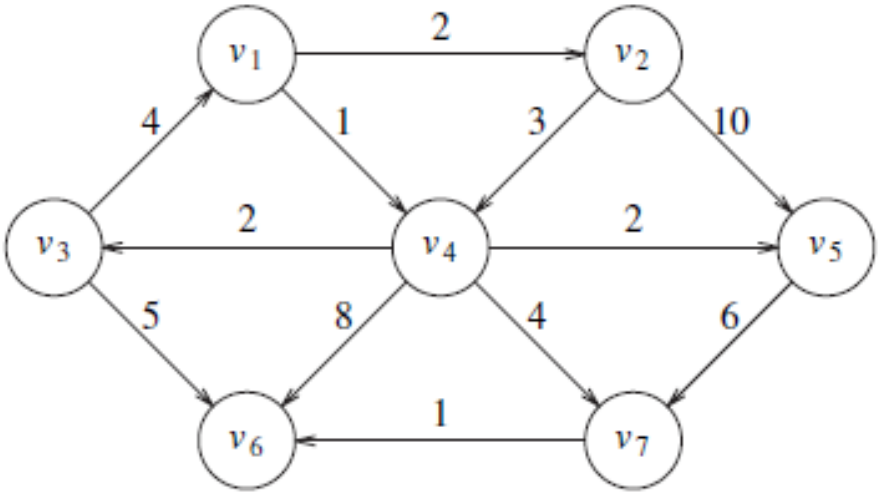
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_2

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



=>

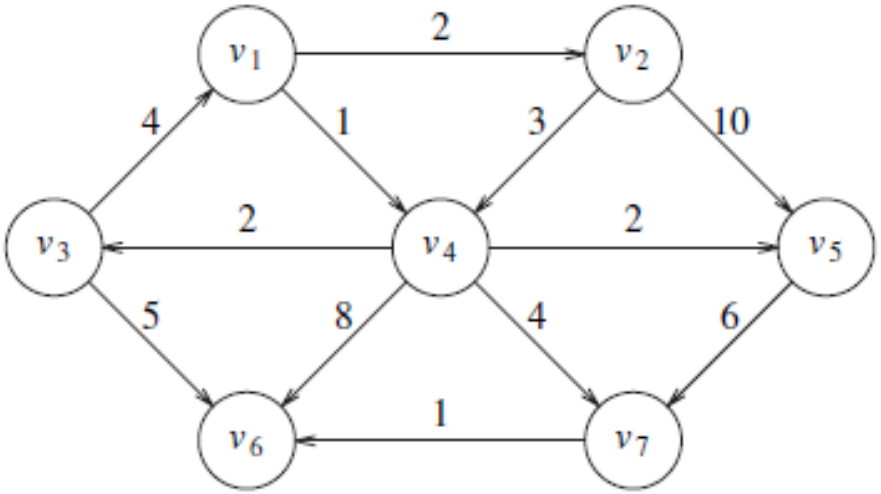
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_4

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



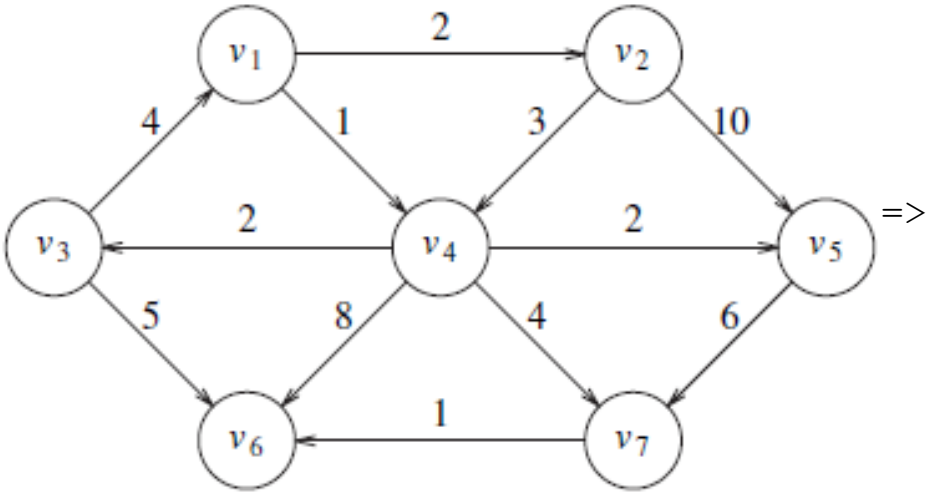

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_4

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



=>

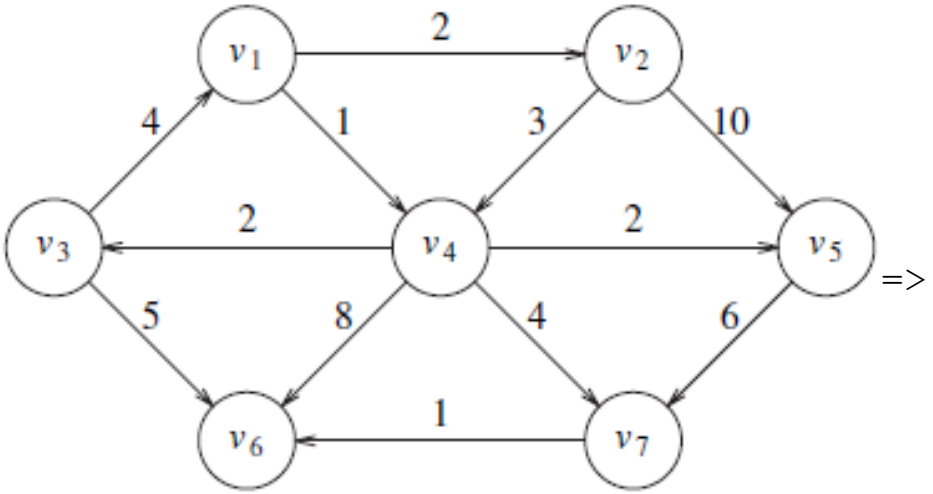
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_4

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



=>

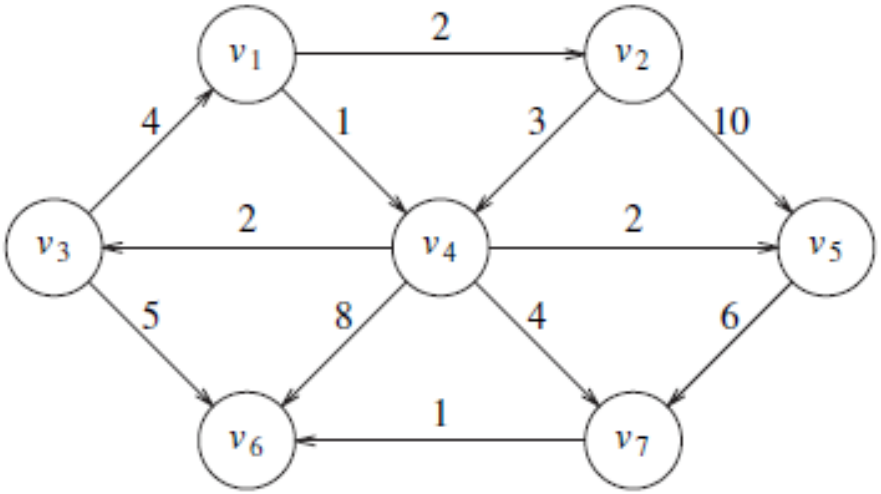
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_1

w: v_4

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



=>

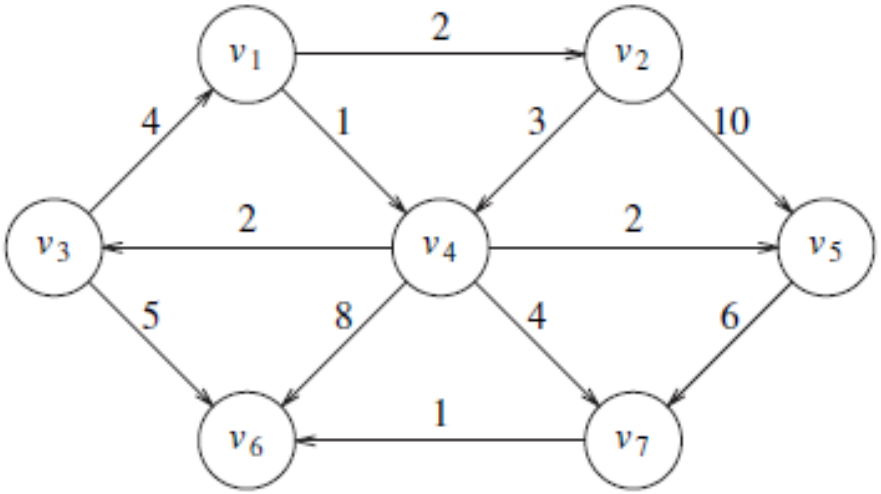
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v:

w:

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



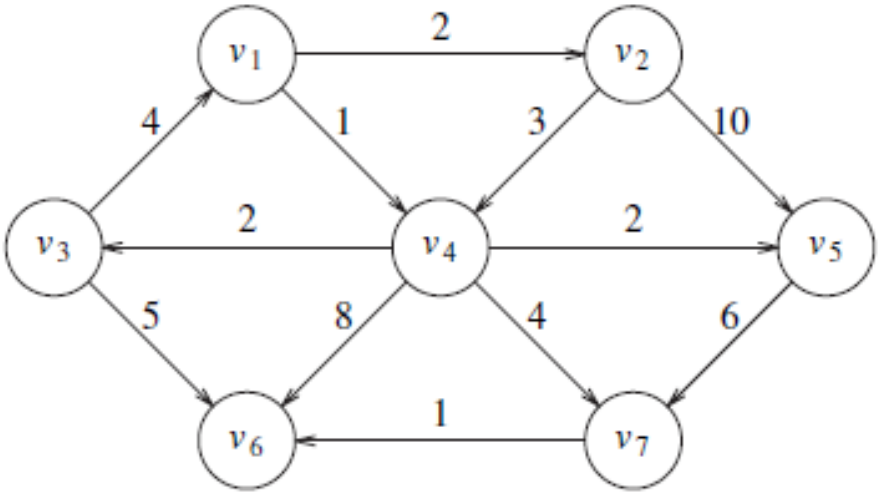

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
=> while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w:

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



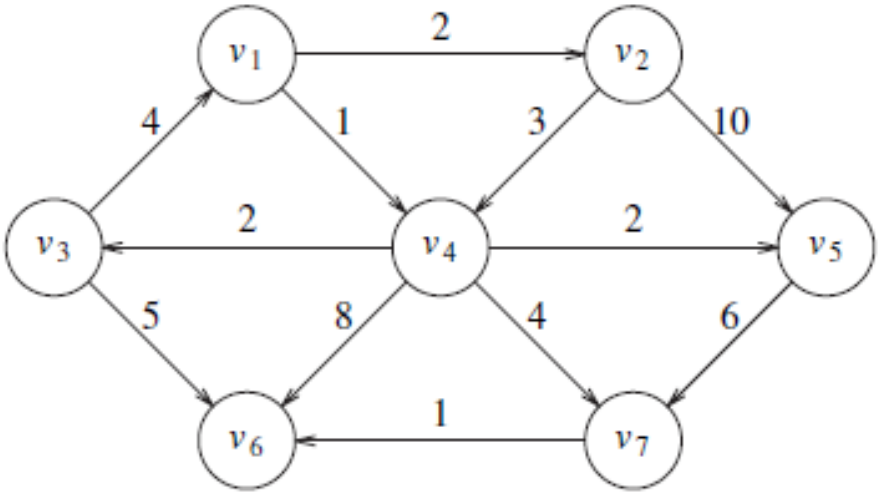

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
=>  v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w:

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



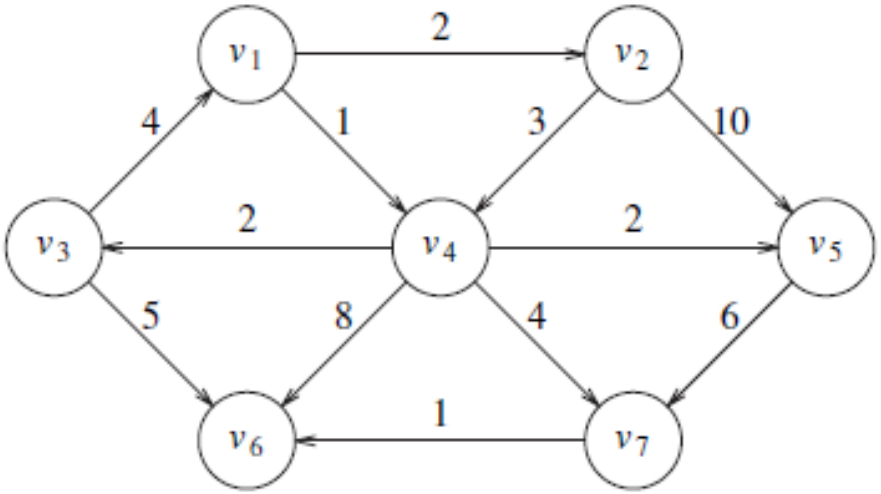

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
=>  known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_3

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



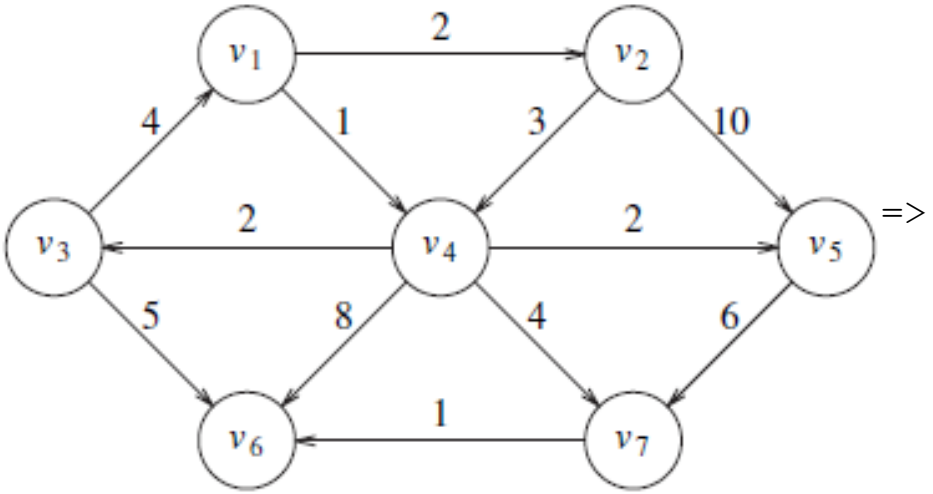

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_3

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



=>

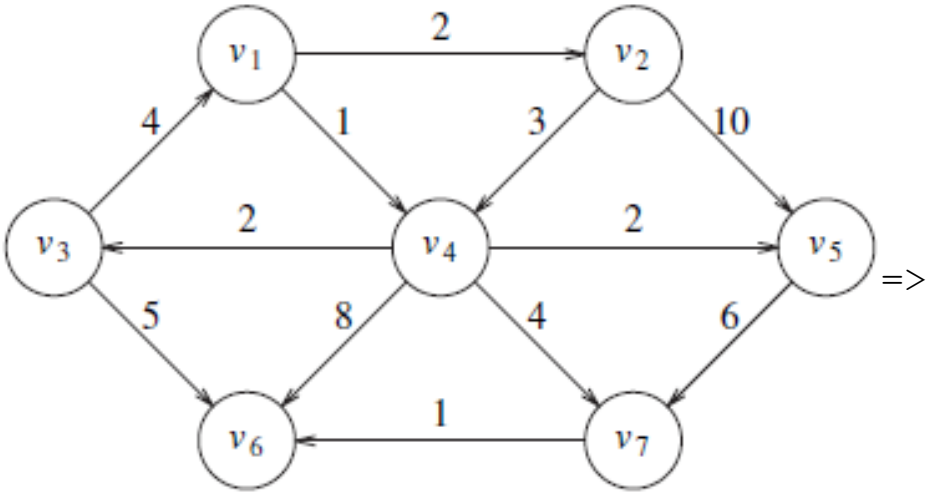
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_3

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	
v_4	T	1	v_1
v_5	F	∞	
v_6	F	∞	
v_7	F	∞	



=>

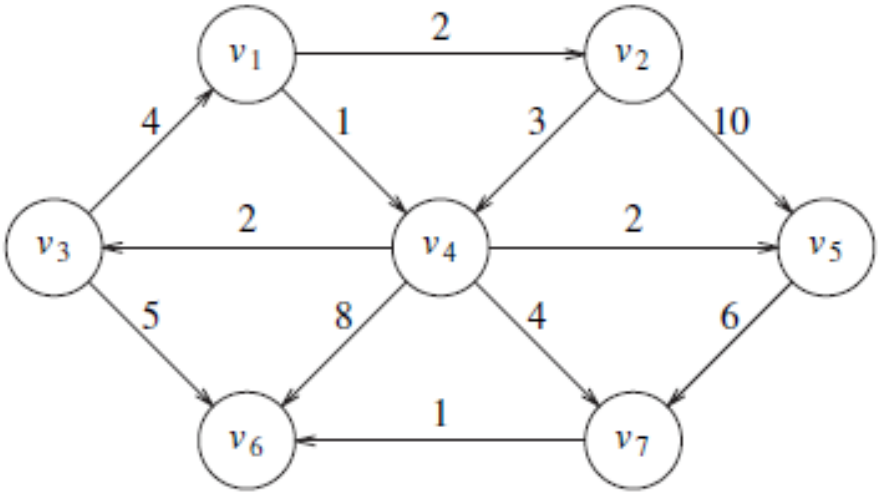
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_3

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	∞	
v_6	F	∞	
v_7	F	∞	



=>

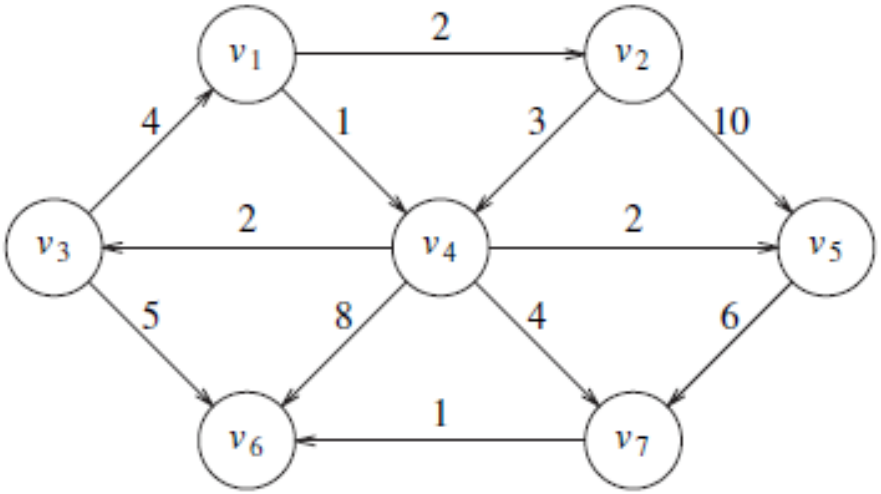
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_5

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	∞	
v_6	F	∞	
v_7	F	∞	



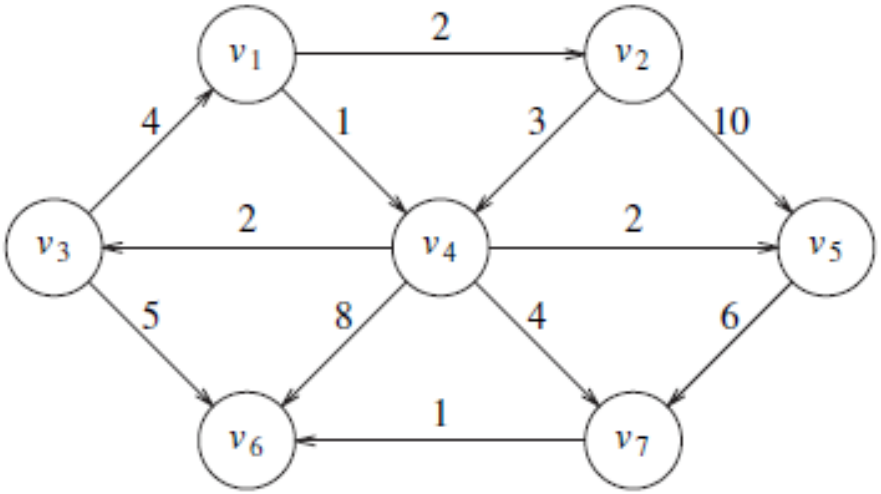

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_5

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	∞	
v_6	F	∞	
v_7	F	∞	



=>

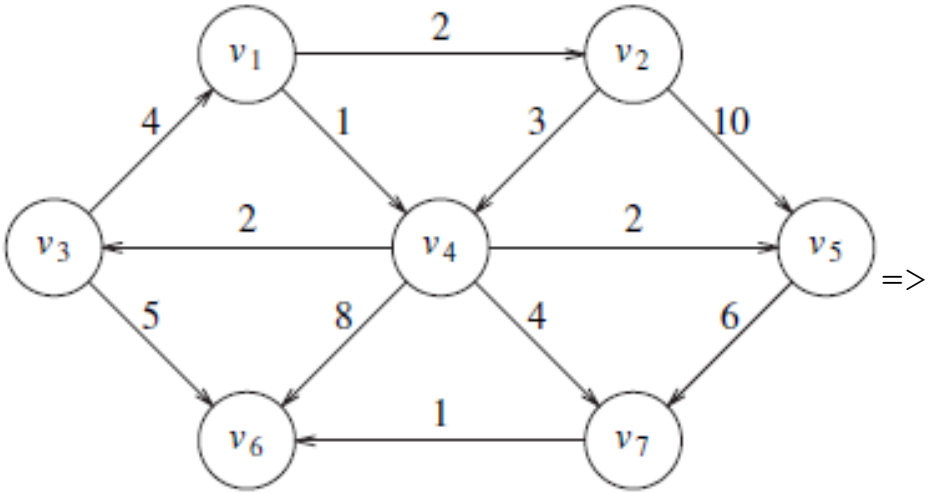
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_5

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	
v_6	F	∞	
v_7	F	∞	



=>

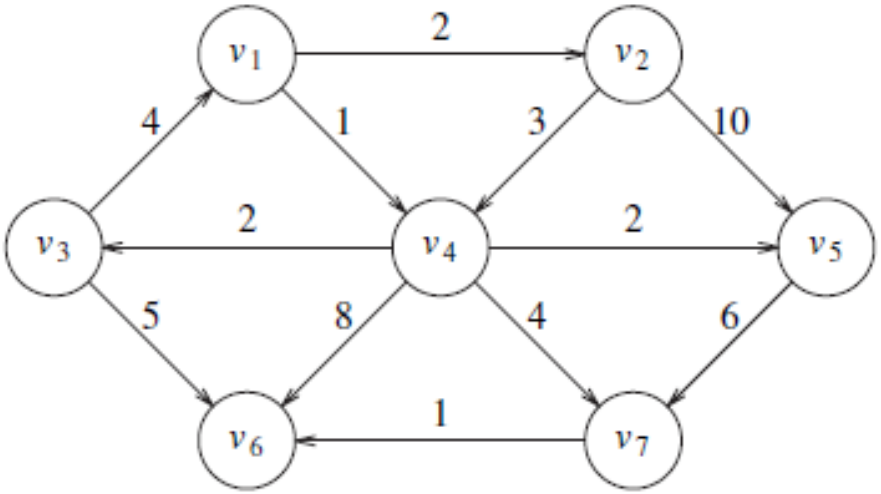
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_5

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	∞	
v_7	F	∞	



=>

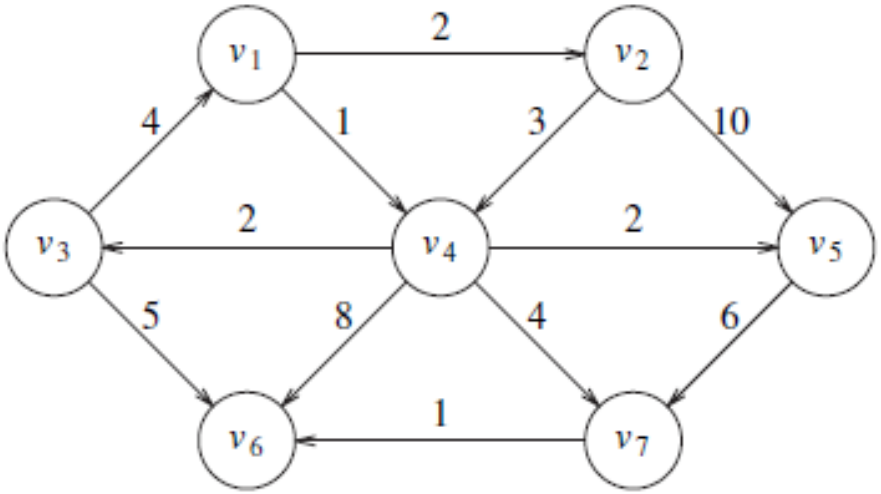
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	∞	
v_7	F	∞	



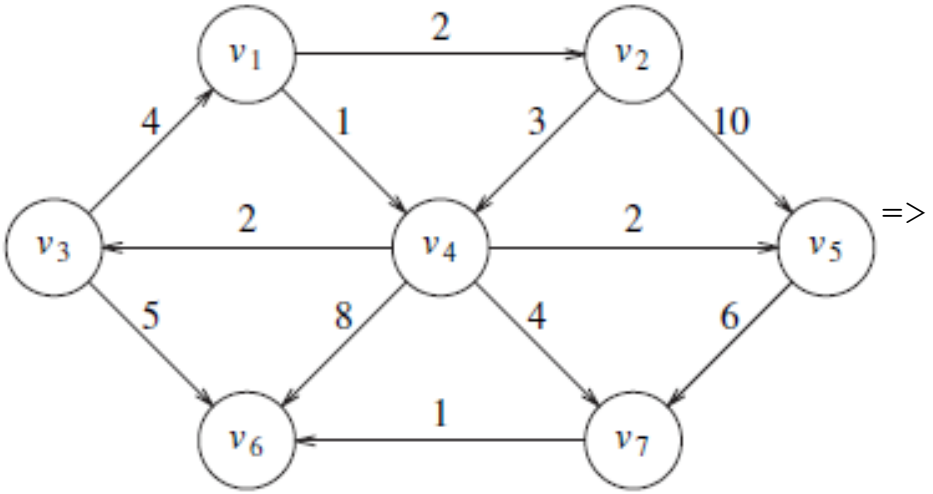

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	∞	
v_7	F	∞	



=>

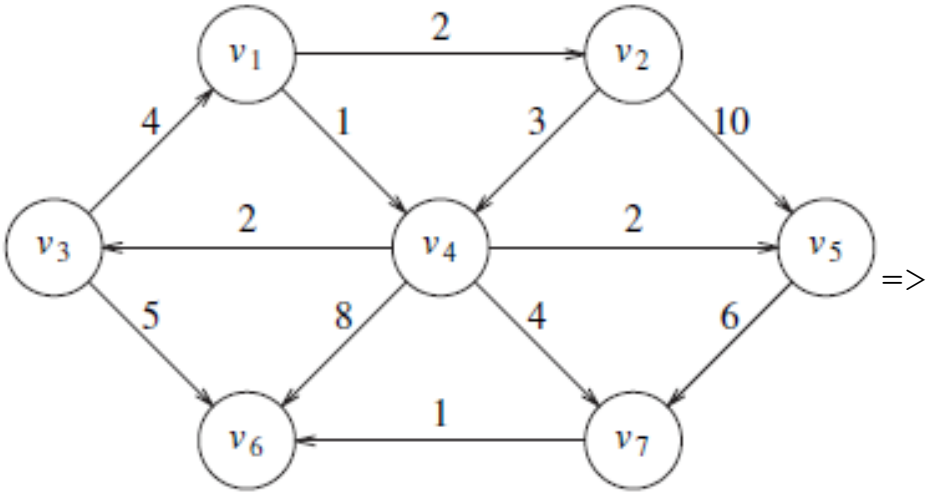
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	
v_7	F	∞	



=>

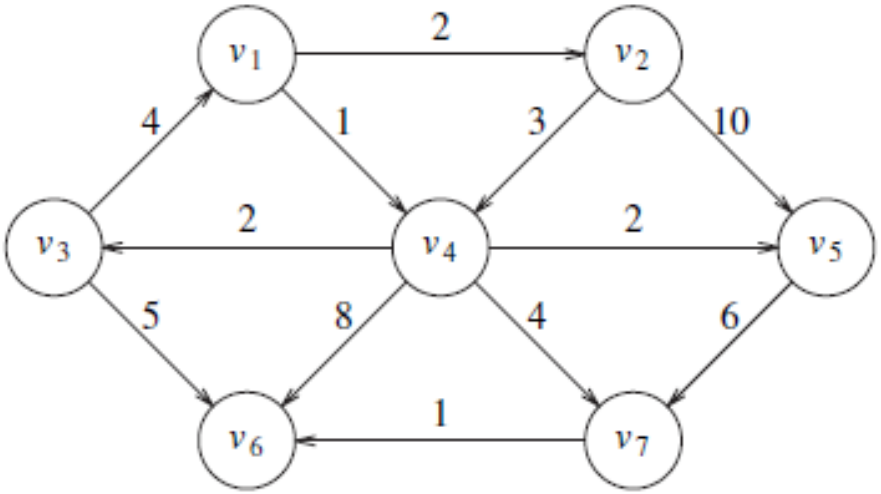
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	∞	



=>

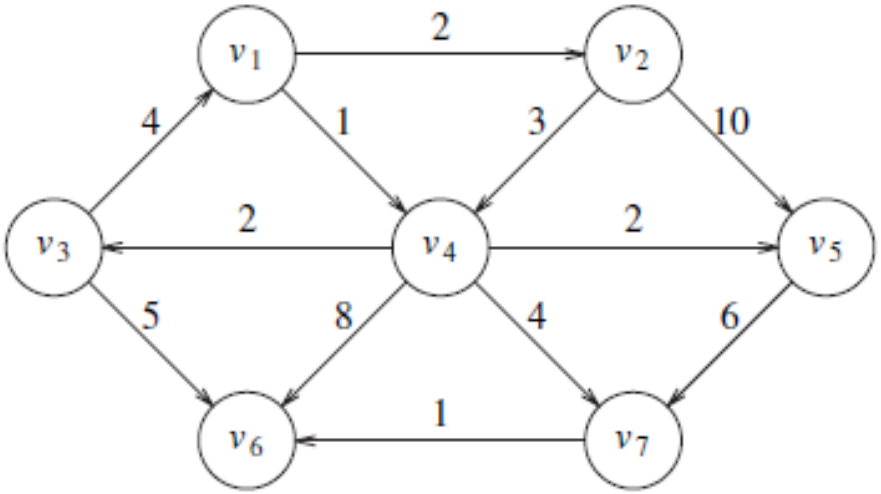
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_7

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	∞	



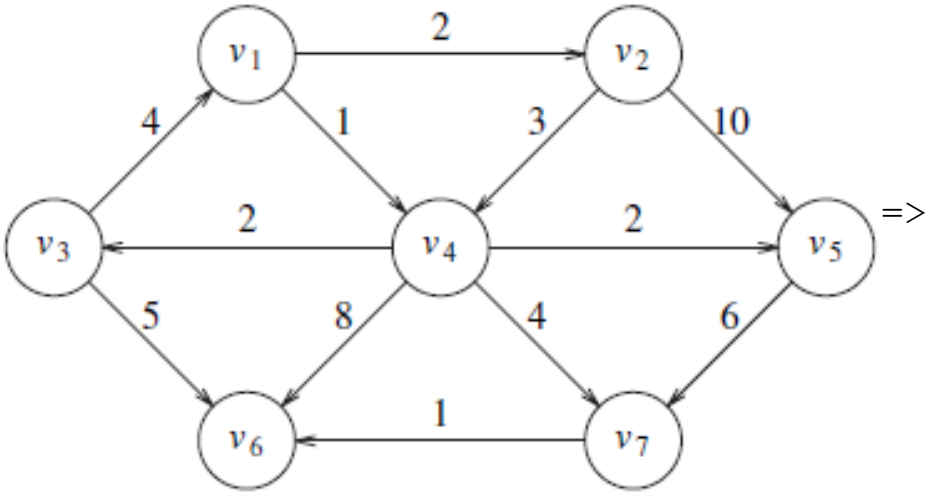

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_7

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	∞	



=>

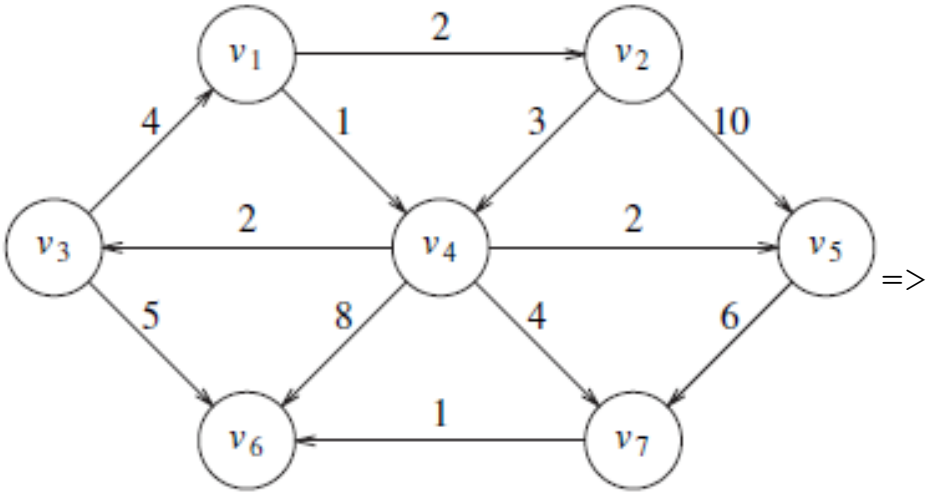
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_7

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	



=>

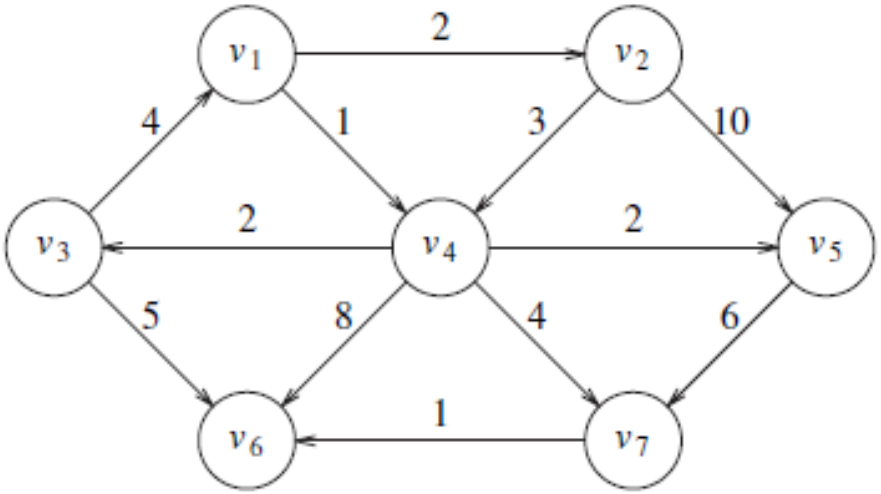
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_4

w: v_7

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



=>

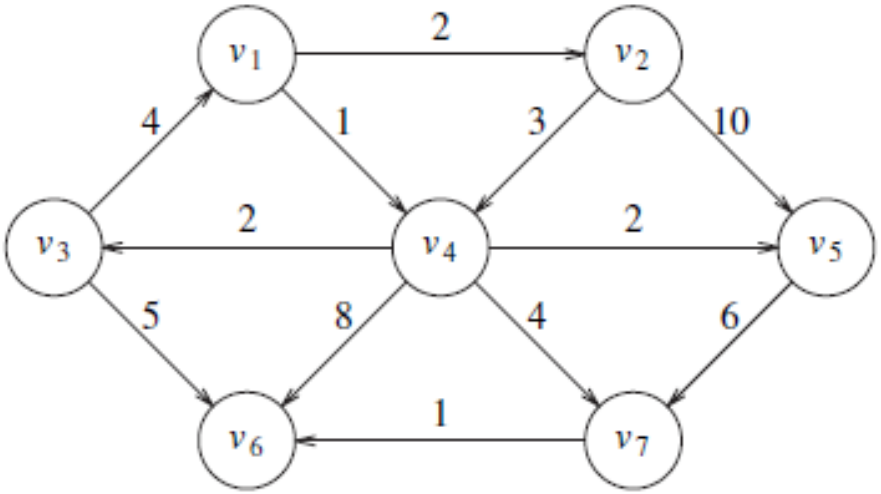
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v:

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



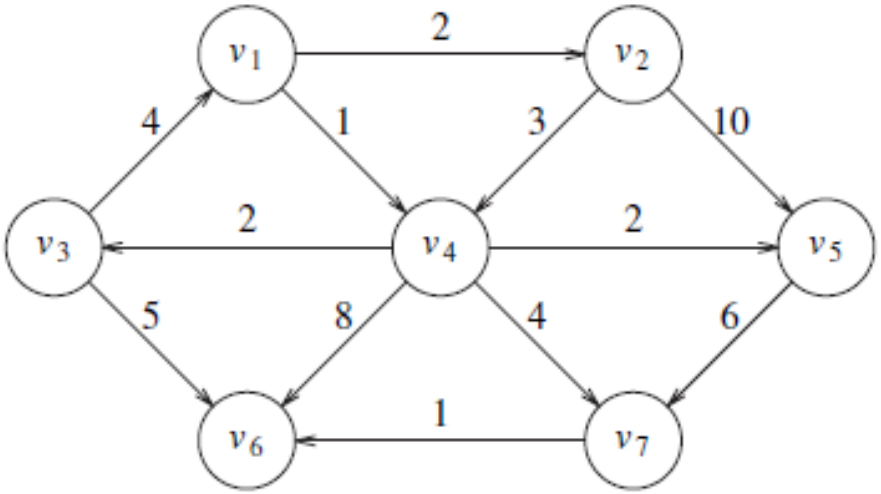

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
=> while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_2

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



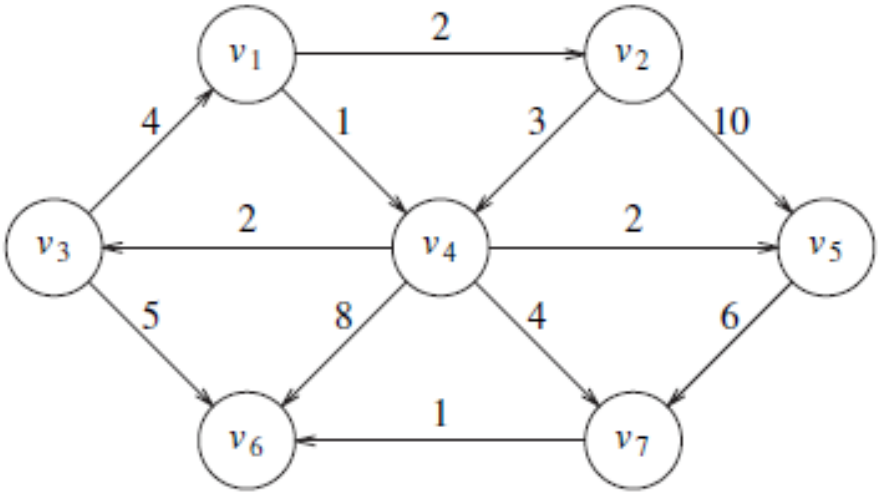

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
=>  v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_2

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



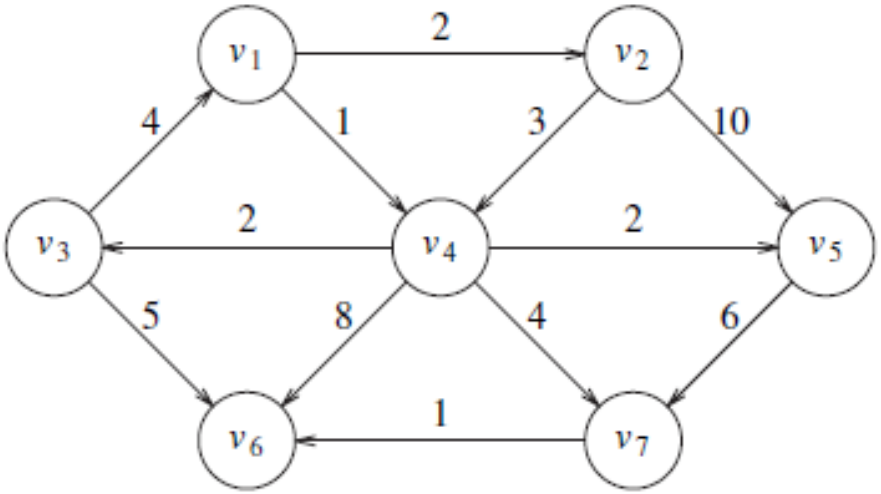

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
=>  known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_2

w: v_4

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



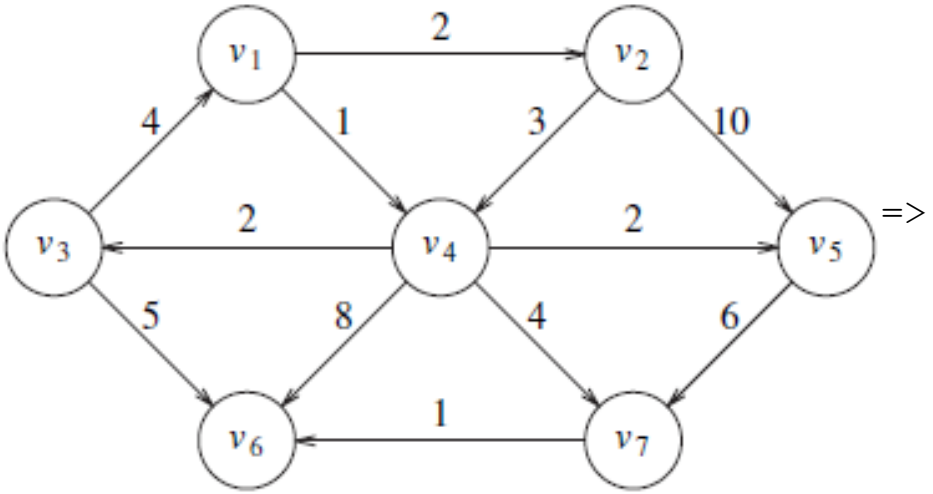

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_2

w: v_4

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



=>

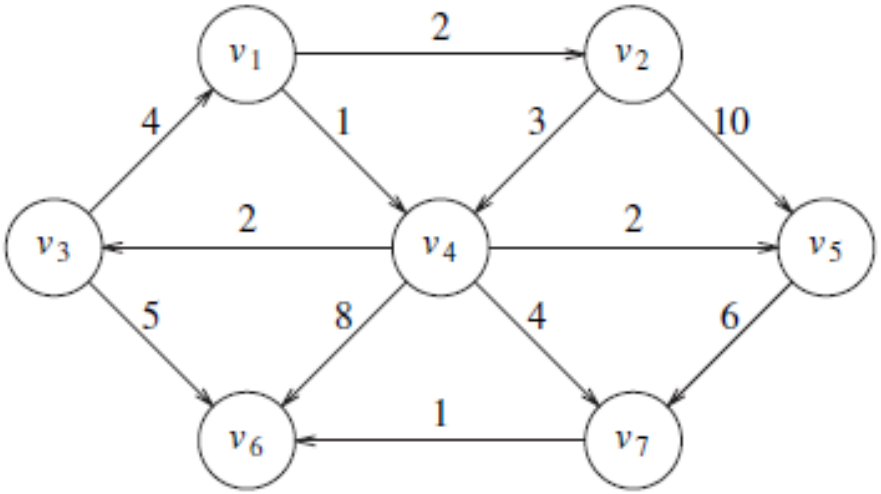
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_2

w: v_5

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



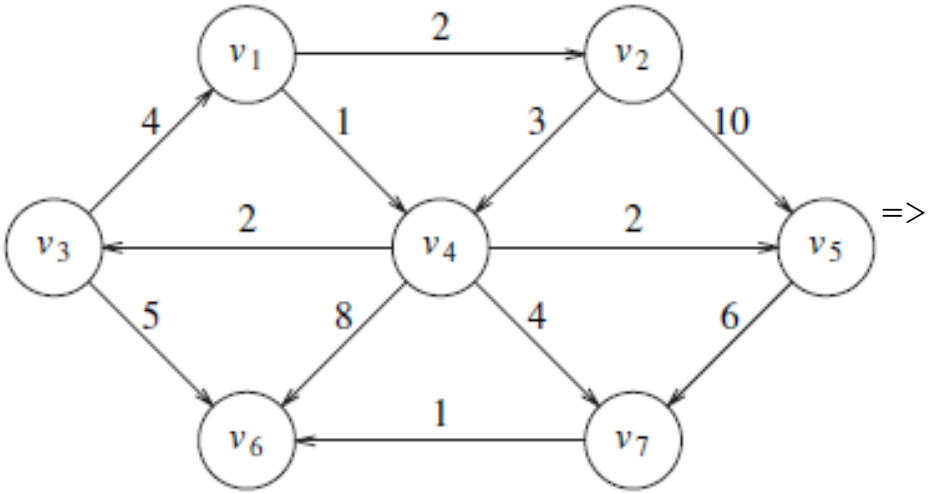

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_2

w: v_5

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



=>

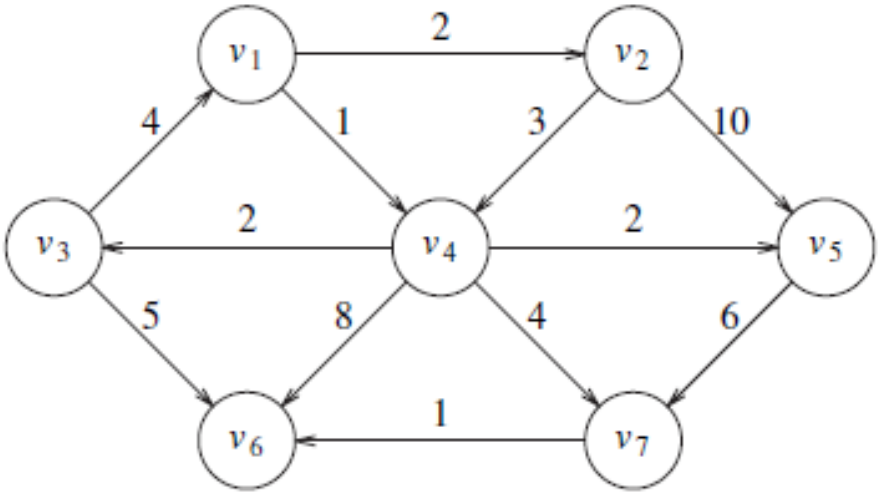
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v:

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



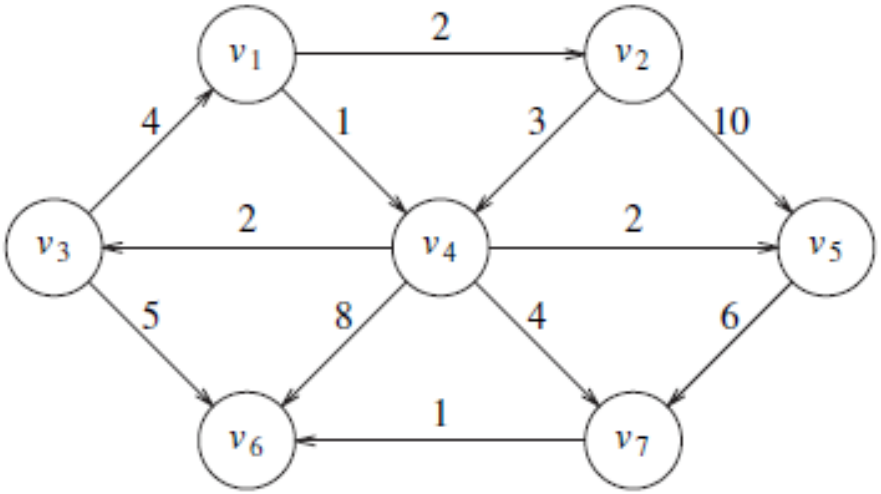

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
=> while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



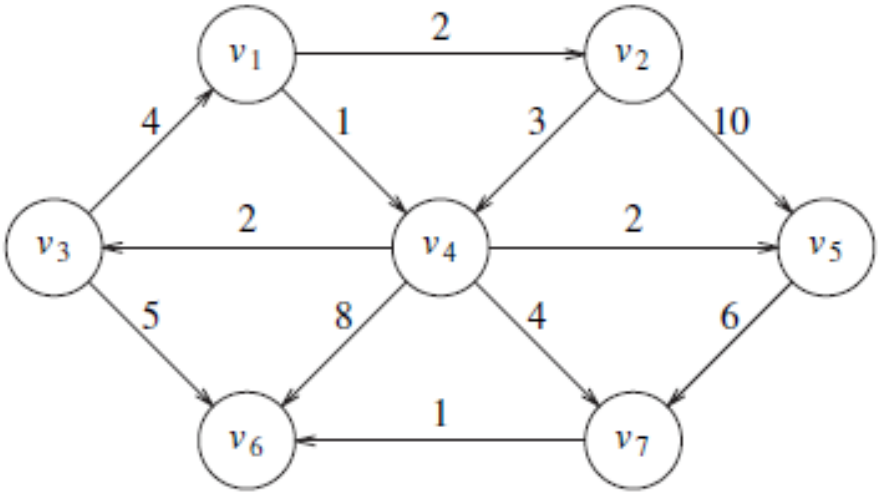

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
=>  v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



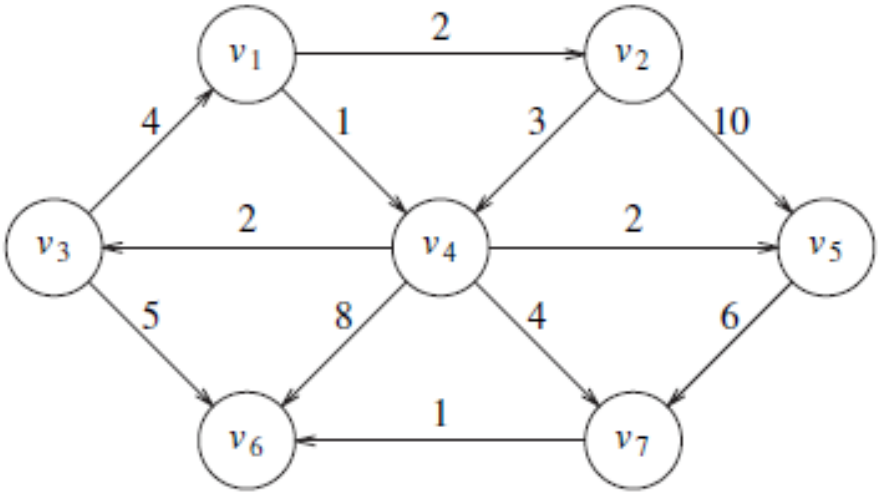

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
=>  known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w: v_1

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



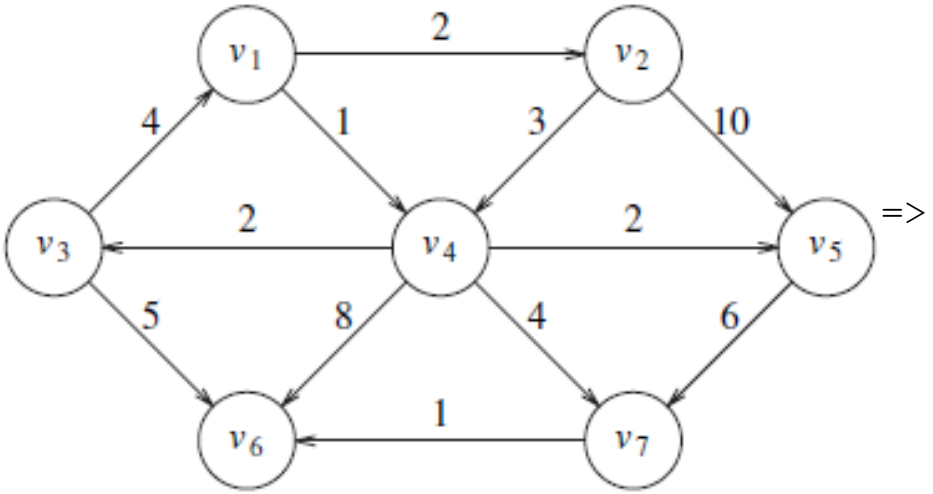

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w: v_1

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



=>

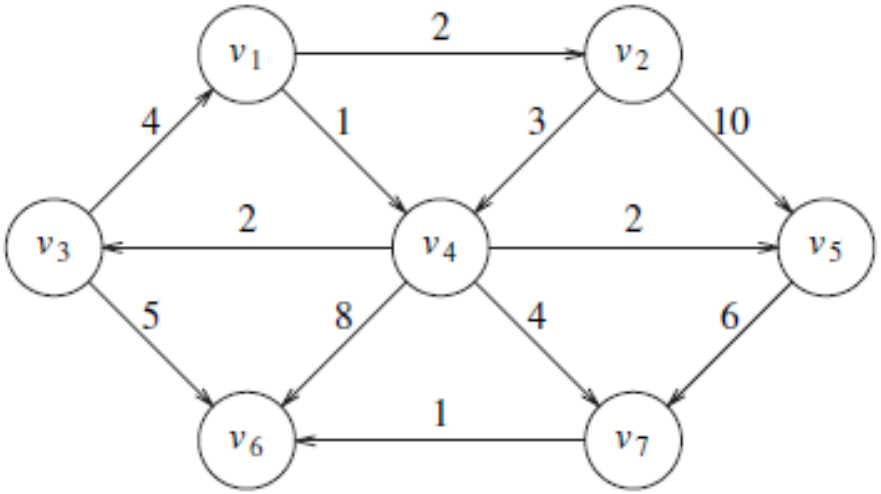
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



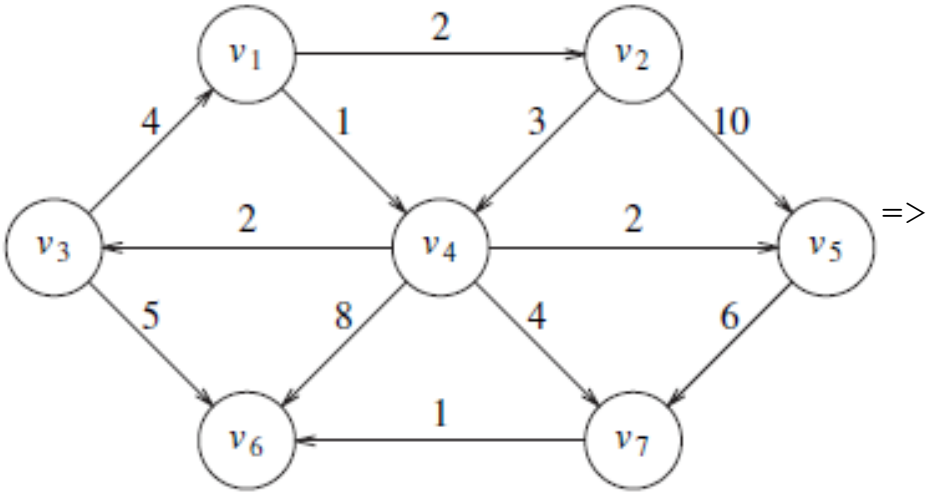

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4



=>

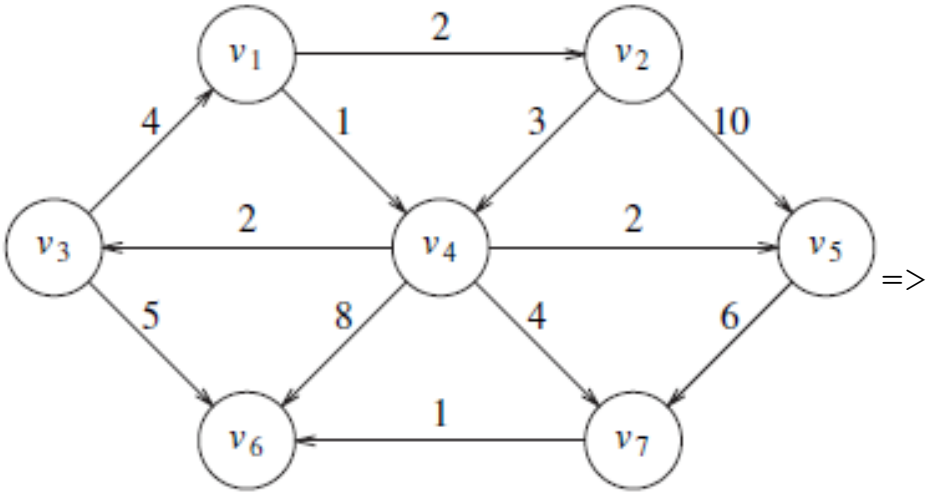
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	8	v_4
v_7	F	5	v_4



=>

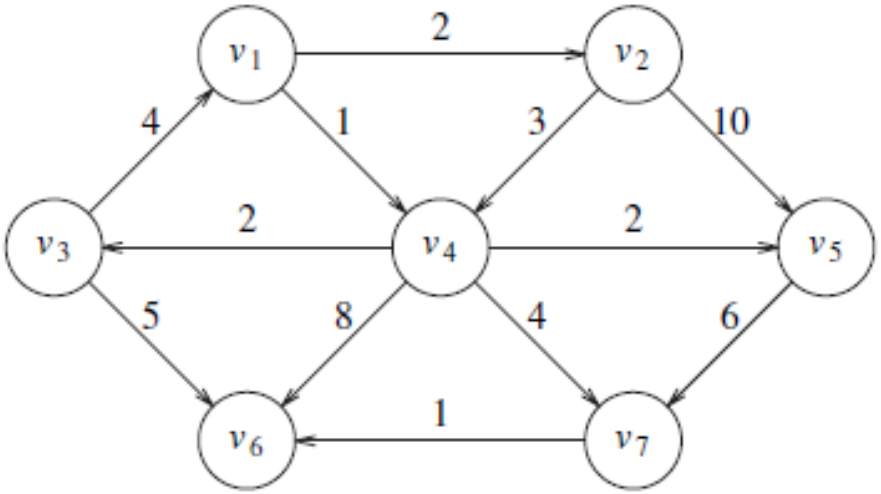
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_3

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4



=>

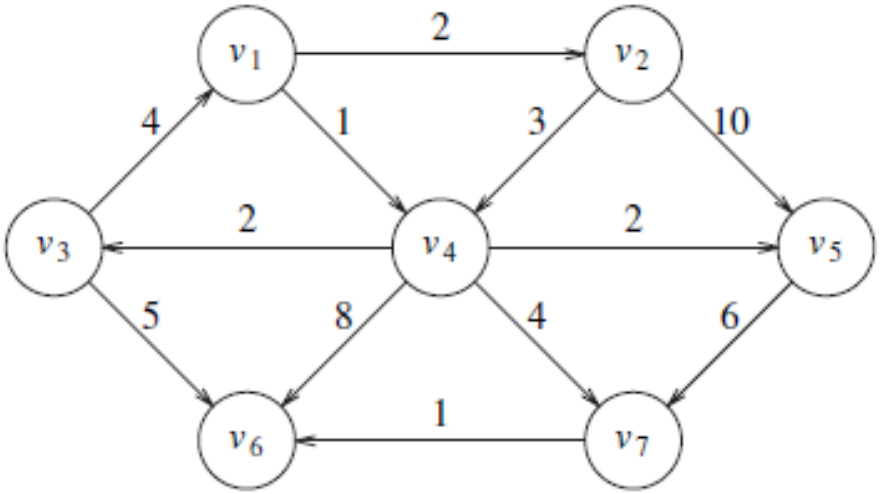
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v:

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4



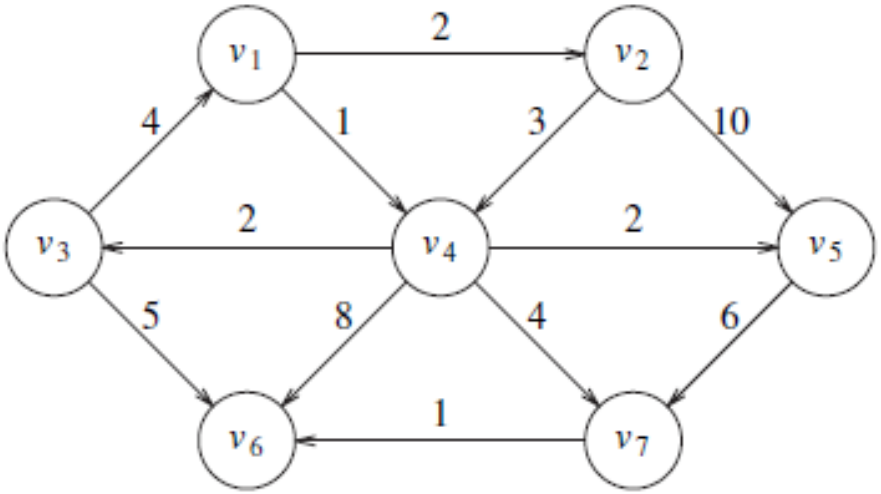

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
=> while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_5

w:

v	known	d_v	p_v
v_1	T	0	NULL
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4



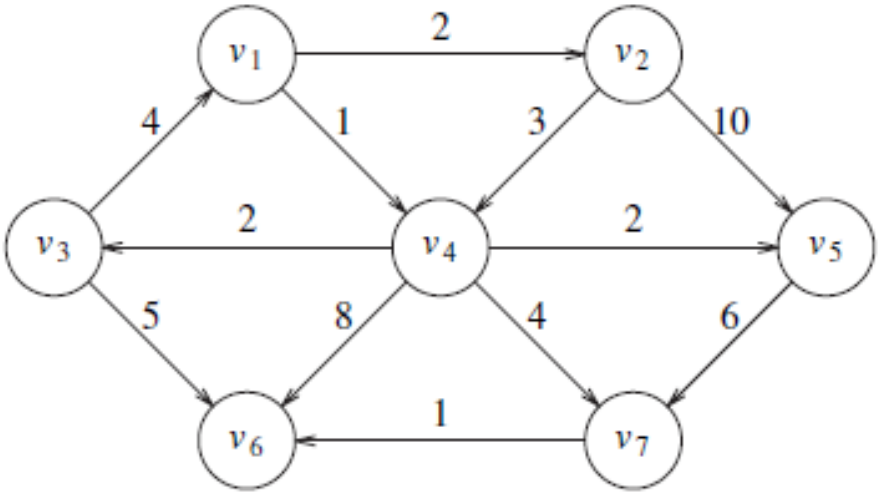

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
=>  v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_5

w:

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	F	5	v_4



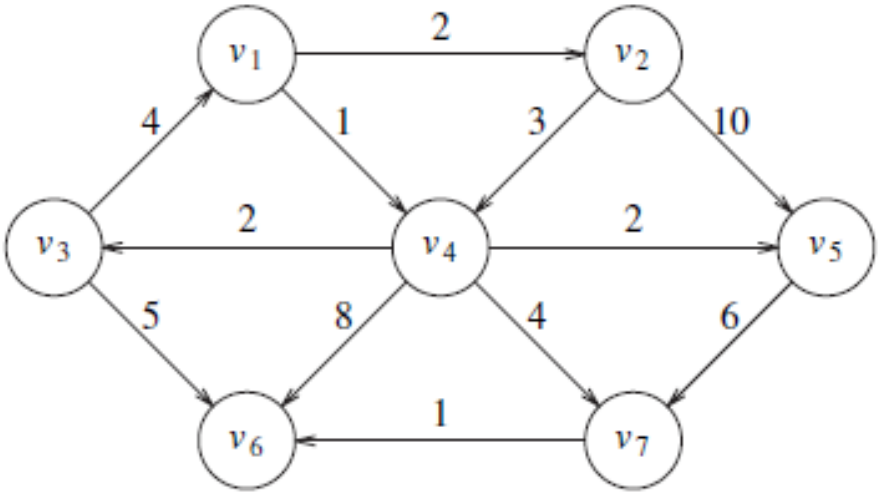

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
=>  known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_5

w: v_7

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	F	5	v_4



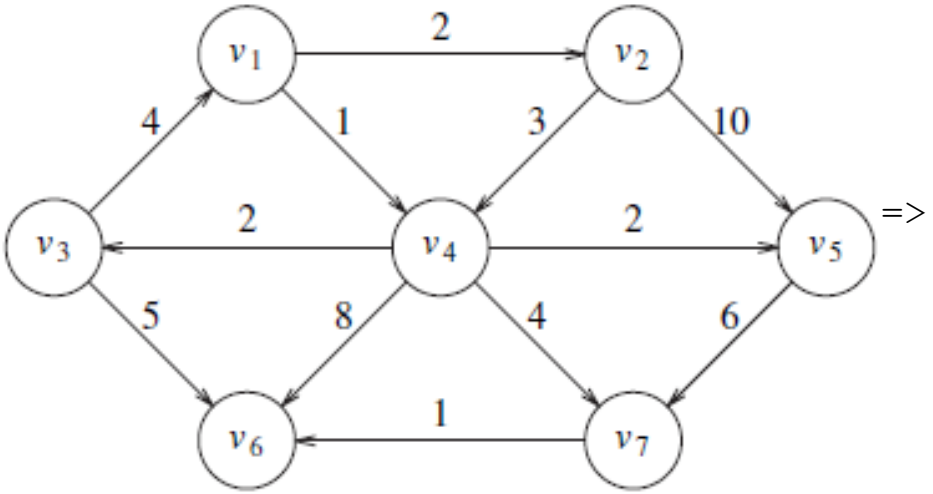

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_5

w: v_7

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	F	5	v_4



=>

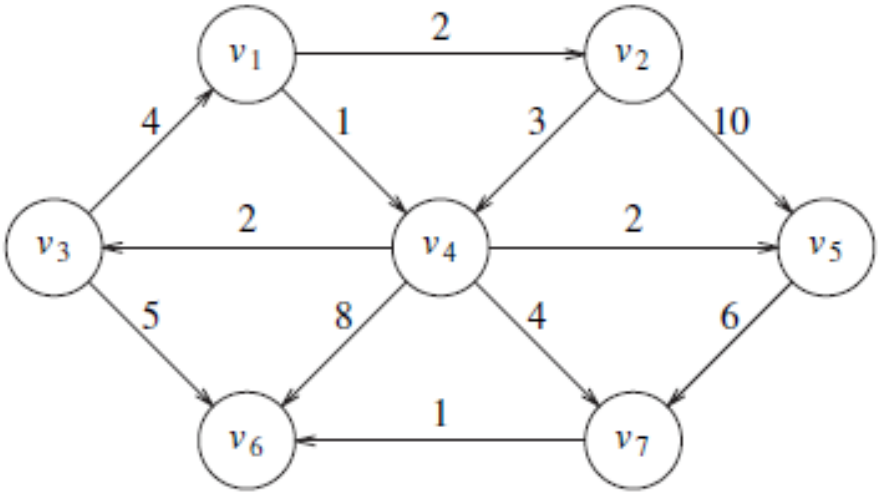
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v:

w:

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	F	5	v_4



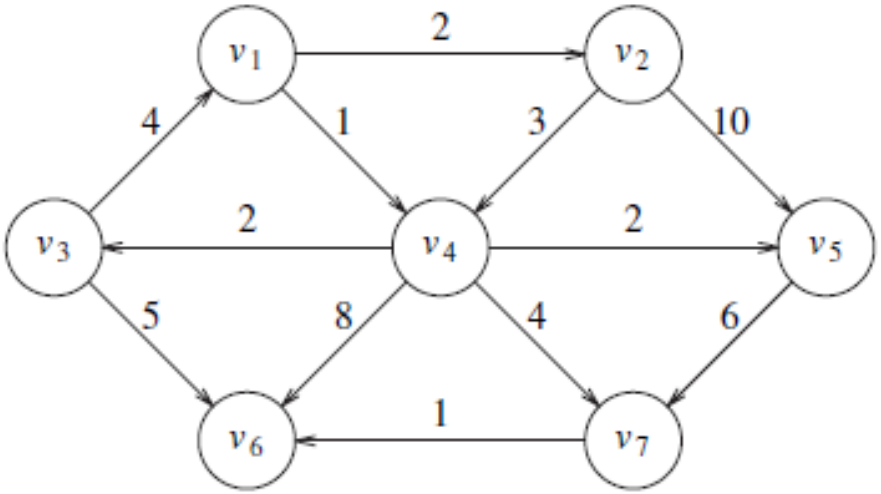

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
=> while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_7

w:

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	F	5	v_4



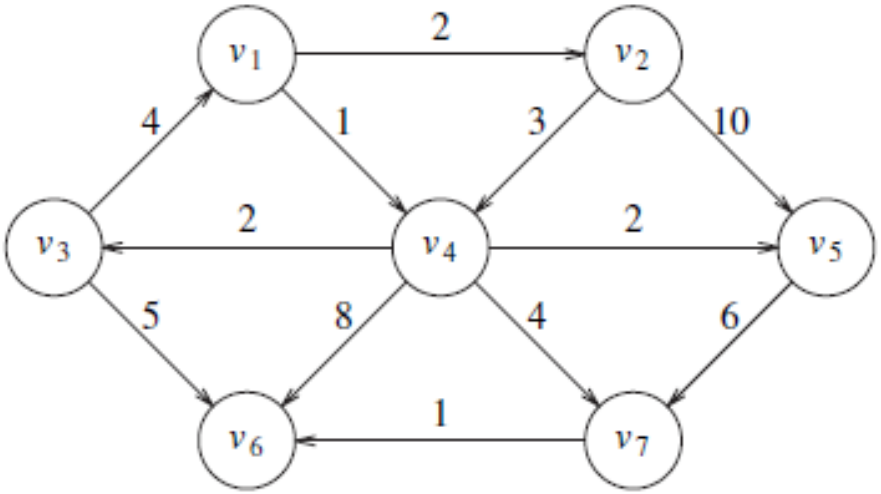

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
=>  v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_7

w:

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	T	5	v_4



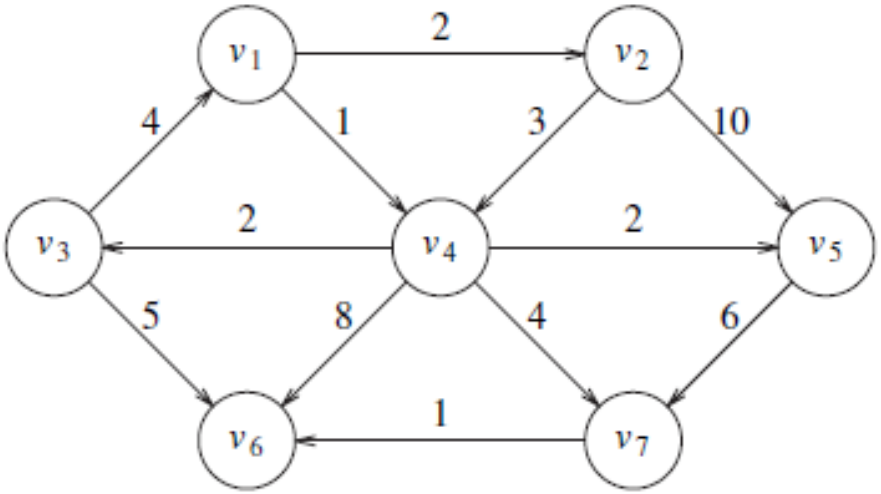

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
=>  known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_7

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	T	5	v_4



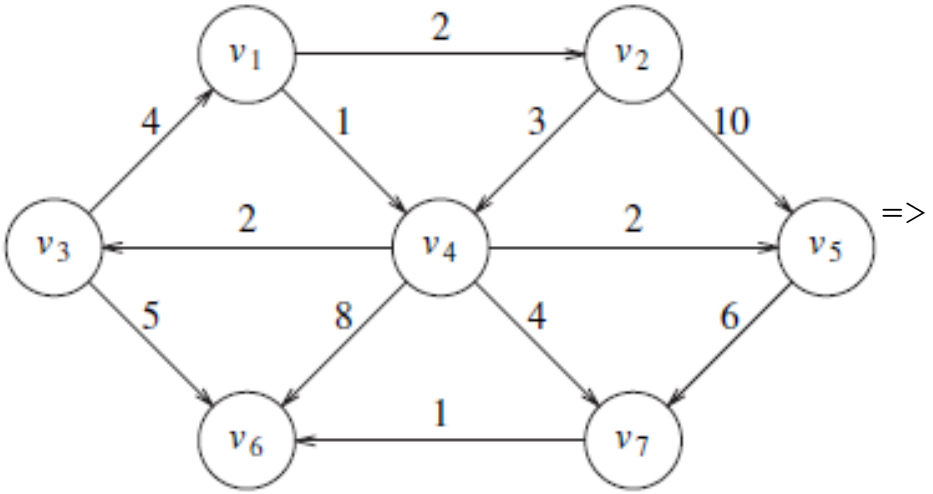

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_7

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
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	F	8	v_3
v_7	T	5	v_4



=>

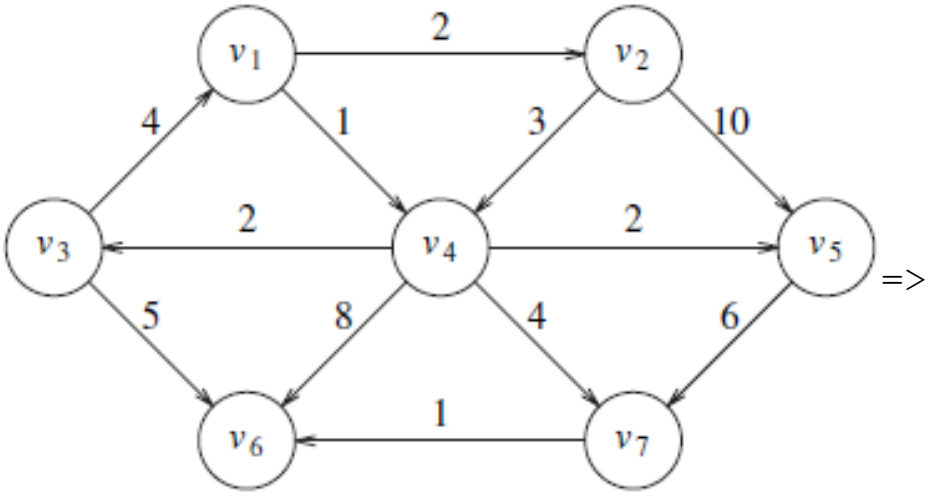
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_7

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
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	F	6	v_3
v_7	T	5	v_4



=>

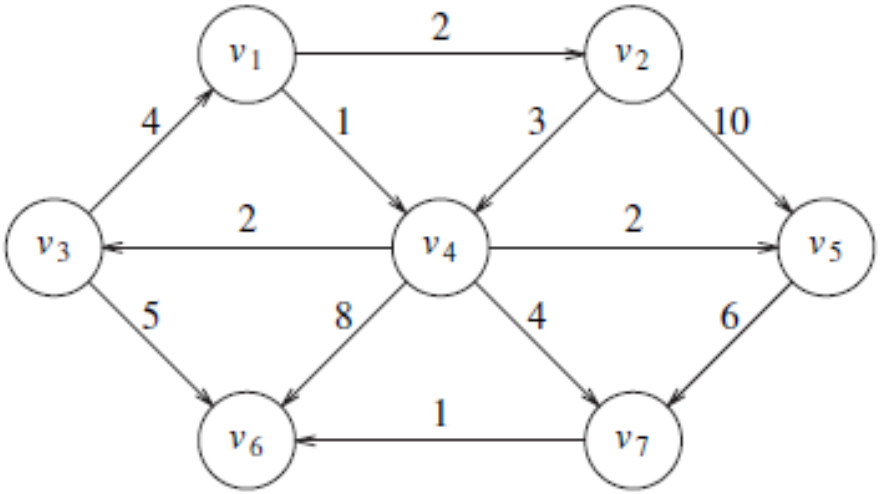
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_7

w: v_6

v	known	d_v	p_v
v_1	T	0	NULL
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	F	6	v_7
v_7	T	5	v_4



=>

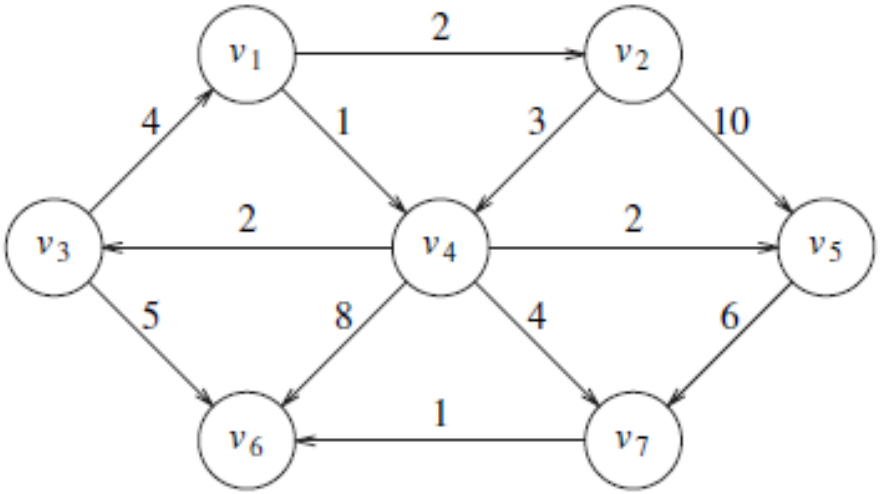
```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v:

w:

v	known	d_v	p_v
v_1	T	0	NULL
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	F	6	v_7
v_7	T	5	v_4



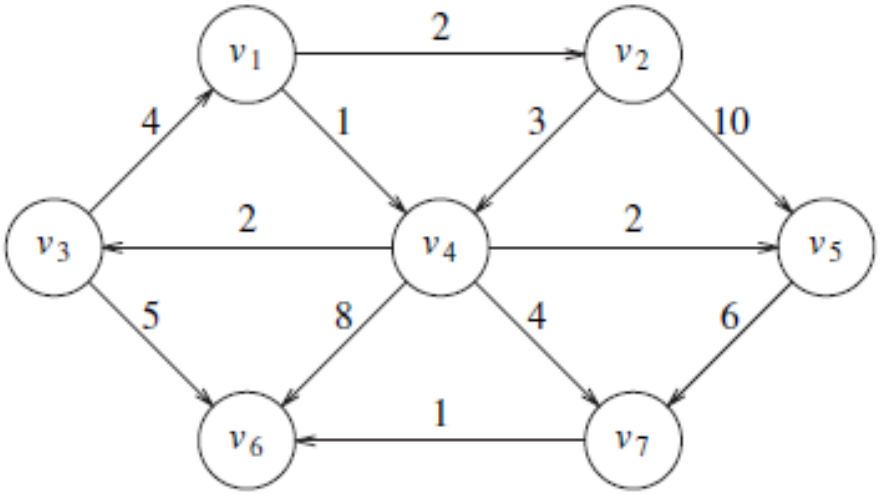

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
=> while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_6

w:

v	known	d_v	p_v
v_1	T	0	NULL
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	F	6	v_7
v_7	T	5	v_4



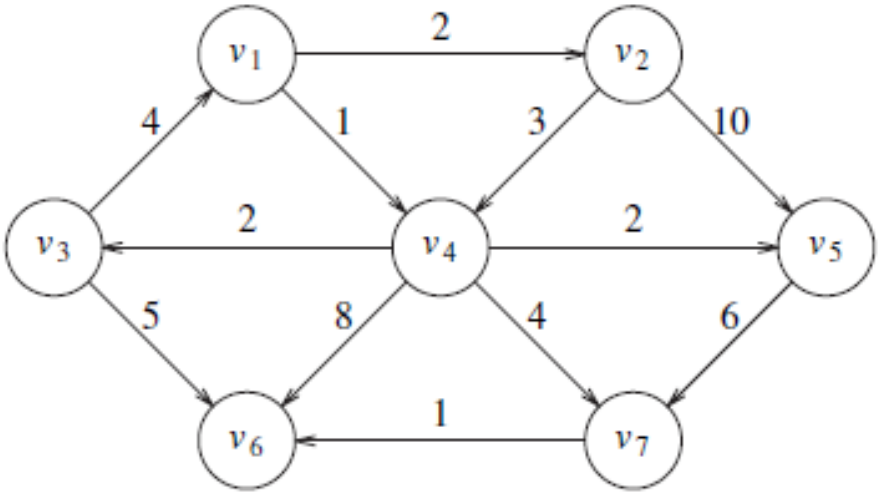

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
=>  v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_6

w:

v	known	d_v	p_v
v_1	T	0	NULL
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



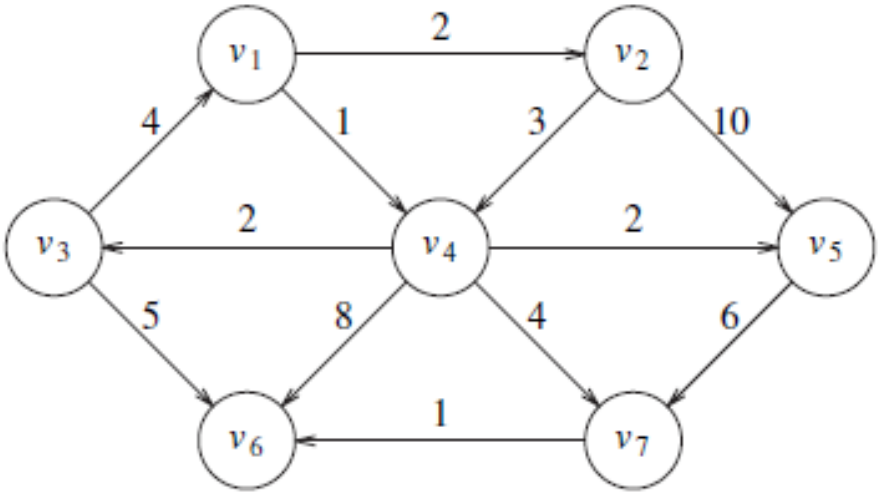

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
=>  known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

v: v_6

w:

v	known	d_v	p_v
v_1	T	0	NULL
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

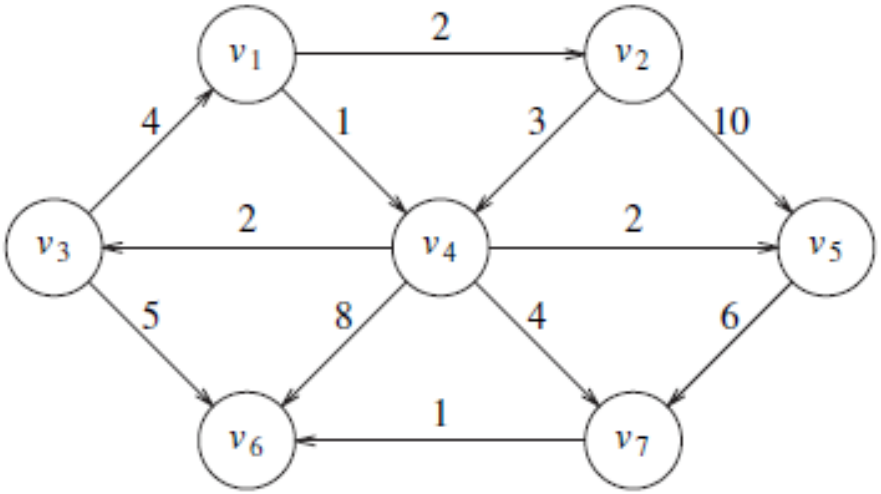



```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
=>  for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Dijkstra's Algorithm Example (cont.)

DONE!

v	known	d_v	p_v
v_1	T	0	NULL
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




```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Time Complexity of Dijkstra's Algorithm

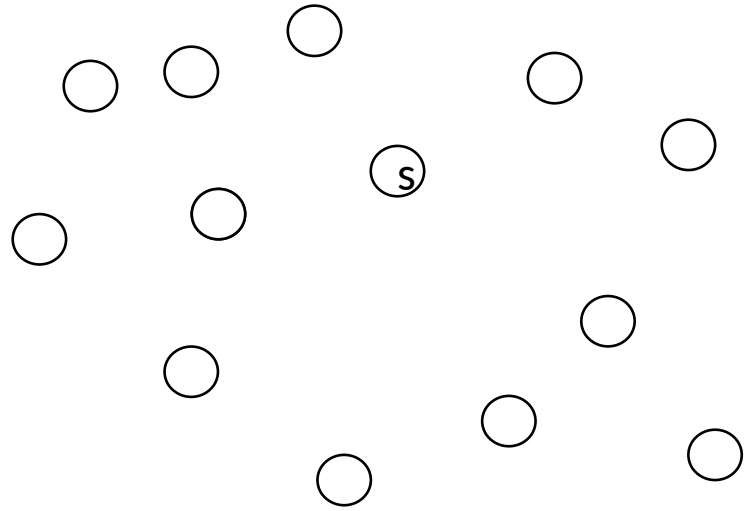
- Assume for now that it takes time linear with respect to $|V|$ to find the unknown vertex with the smallest cost, and that it also takes linear time to find all outgoing edges from a vertex
- Then, this would clearly lead to an $\Theta(|V|^2)$ algorithm (two consecutive linear parts inside a linear loop), which is the best we can do if the graph is dense, since $|E| \approx |V|^2$ in that case
- If the graph is sparse, especially if $|E| = \Theta(|V|)$, we can do much better
- Assume that we use adjacency lists to store the outgoing edges from each vertex
- To find the unknown vertex with the smallest d-value, we will rely on a **priority queue** (we'll assume a binary heap that includes a hash table to efficiently find items based on their ids)
- Consider the line: `v ← the unknown vertex with the smallest d-value`
- This is accomplished by a *deleteMin* operation, which takes time $O(\log |V|)$
- Now consider the line: $d_w \leftarrow d_v + c_{v,w}$
- This is accomplished by a *decreaseKey* operation, which also takes time $O(\log |V|)$
- Note also that the inner loop will execute at most once per edge, so the total running time of the algorithm will be $\theta(|V| * \log |V| + |E| * \log |V|) = \theta(|E| * \log |V|)$

Proving the Optimality of Dijkstra’s Algorithm

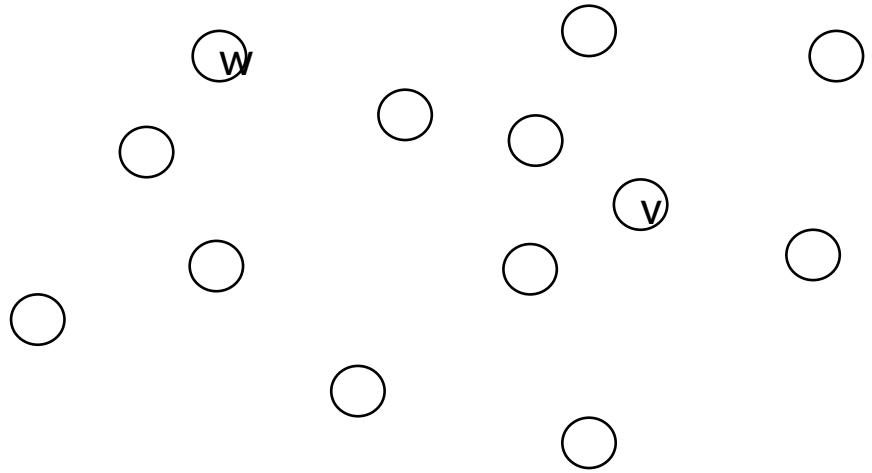
- Dijkstra's algorithm is an example of a **greedy algorithm** (we will talk about greedy algorithms in general later in the course)
- Not all greedy algorithms lead to the best possible solution for the problem they are solving (i.e., they are not all optimal)
- We will use an inductive proof to show that *Dijkstra's algorithm is optimal if there are no negative cost edges* in the graph
- Our textbook mentions an indirect proof to prove the same thing, but leaves it as an exercise
- Clearly, the best cost for the starting vertex is the zero cost to which it is initialized
- During the first pass of the algorithm, the starting vertex becomes the first known node, and vertices adjacent to it are assigned their appropriate distances
- Clearly, all the d-values are correct after this first pass (for both known and unknown vertices, with meanings previously defined)
- Now assume that after some pass, we know the shortest possible paths to all known vertices, and we know the shortest path to each unknown vertex starting at the starting vertex and only going through known vertices (both are true after the first pass)
- Then, unknown vertex with the smallest d-value must have that cost be its best possible cost
- Otherwise, there would have to be another unknown vertex with a smaller achievable cost (assuming no negative cost edges)
- Also, each remaining unknown node only needs to be updated if there is a better weighted path cost starting from s, traveling through only known vertices with the new one last, and then moving to the unknown node
- Therefore, adding the selected vertex to the known list and updating the costs of paths to adjacent unknown vertices, as specified by the algorithm, keeps the stated assumptions true after the next pass

Proof of Dijkstra’s (inductive step, part 1)

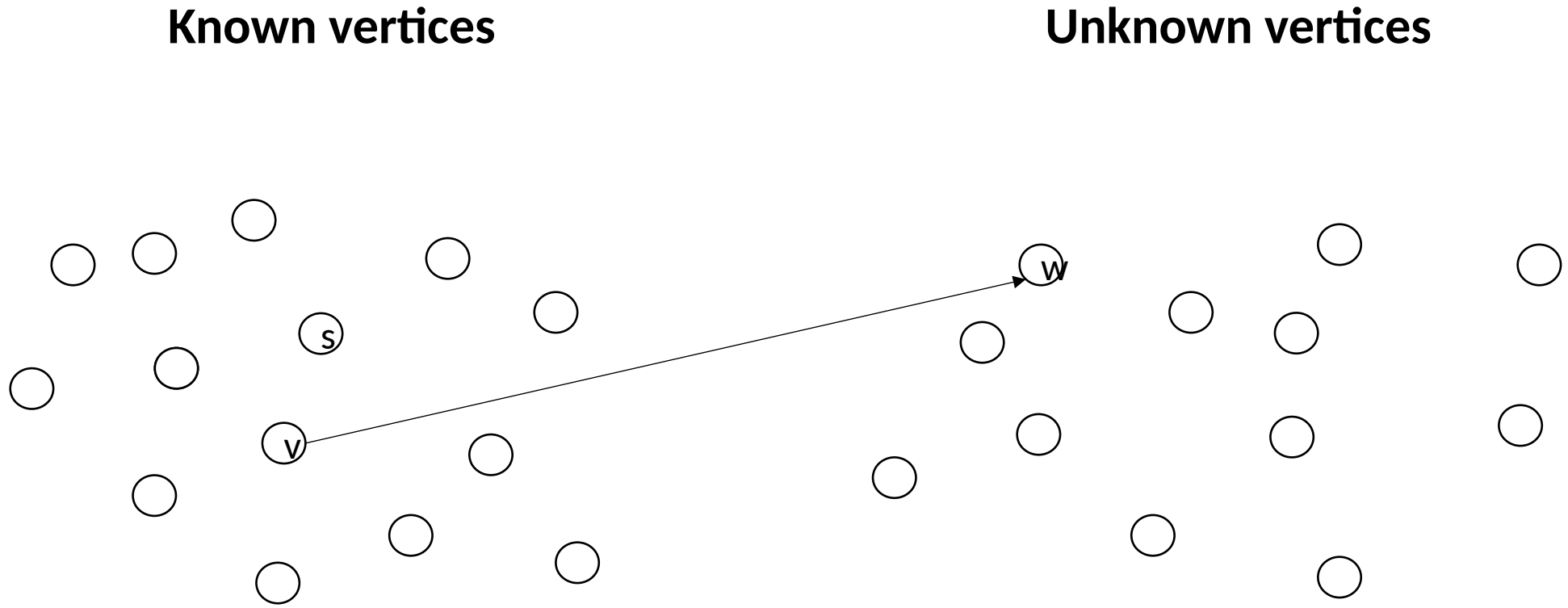
Known vertices



Unknown vertices



Proof of Dijkstra’s (inductive step, part 2)



Bellman-Ford algorithm

- Next, we'll consider a weighted graph that may include negative-cost edges
- Recall that if there are negative-cost cycles, the problem does not have a solution
- We still want to find the shortest weighted path from a single source to all other nodes (i.e., this is another version of the single-source shortest-path problem)
- The textbook shows pseudo-code for a solution that has a worst-case time complexity of O(|V| * |E|), but it might run considerably faster if the graph is sparse
- The book's solution uses a regular queue and falls into an infinite loop if negative cycles are present, although the authors state how this can be remedied
- The book's solution is a variation of a solution known as the **Bellman-Ford algorithm** (although they don't specify the name)
- We are going to look at different version of the Bellman-Ford algorithm that also has a time complexity of O(|V| * |E|) and always takes that long
- Note that this is significantly slower than Dijkstra's algorithm for a sparse graph, but that is necessary if we are allowing negative cost-edges
- Our solution will be simpler than the one from the book, and it will never fall into an infinite loop; it will return FALSE if a negative cycle is present, and TRUE otherwise

Bellman-Ford Algorithm Pseudo-code

- The time complexity of the algorithm, O(|V| * |E|), should be obvious
- After the k^{th} pass of the outer loop, all paths from s with up to k edges will have been discovered
- After the k^{th} pass, we will have found the best way to get from s to every other node using at most k edges
- If we assume no negative-cost cycles, the optimal path from one vertex to another uses at most |V| - 1 edges
- Therefore, after |V| - 1 passes, all optimal paths from s will be discovered, unless there is a negative-cost cycle
- This can be checked with one more loop to see if further improvements are possible
- If so, the algorithm returns false
- In this case, not all the lengths and paths determined by the algorithm are optimal
- The algorithm returns TRUE otherwise, meaning that every discovered path is optimal

```
WeightedPLBellmanFord (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
  d_s ← 0
  p_s ← NULL
  repeat |V| - 1 times
    for every edge (v, w)
      if d_v + c_v,w < d_w
        d_w ← dv + c_v,w
        p_w ← v
  for every edge (v, w)
    if d_v + c_v,w < d_w
      return FALSE
  return TRUE
```

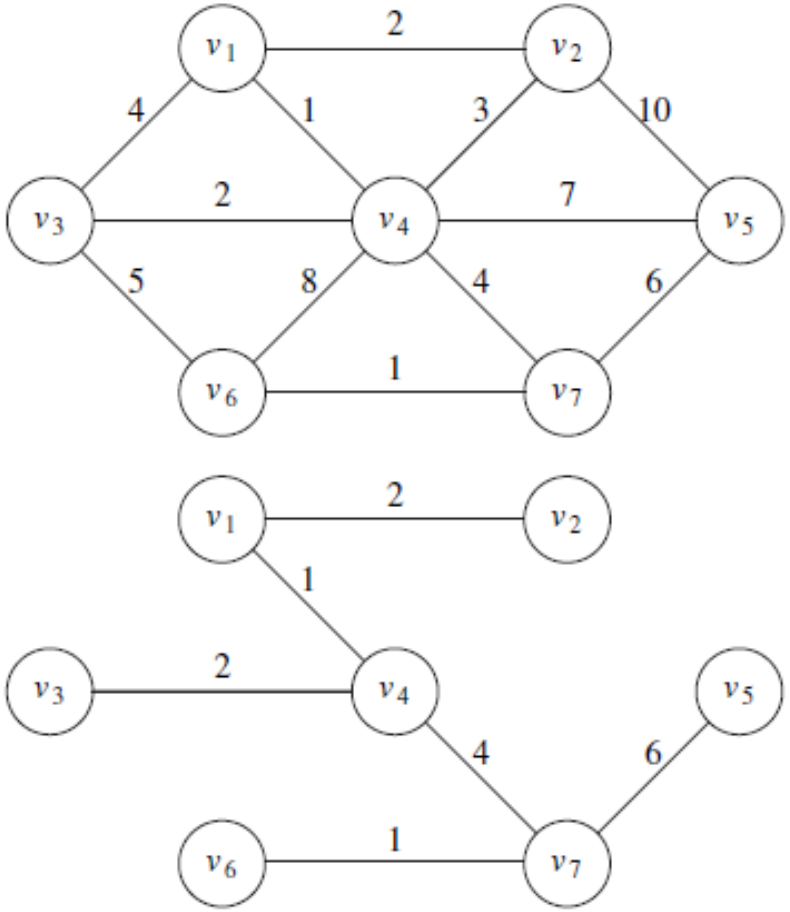
Other Shortest-Path Algorithms (briefly)

- The textbook describes another solution to the single-source shortest-path problem that can be used for directed acyclic graphs (DAGs)
  - Because we know there are no cycles, there is an optimal solution that is more efficient than Dijkstra's algorithm, even if negative-cost edges are present
  - One application of this algorithm involves critical path analysis
- Sometimes, we want to find the shortest path from a starting vertex, s, to a single destination vertex, d (not mentioned in the textbook)
  - One possibility is to apply Dijkstra's algorithm and stop when d becomes known
  - Another possibility, often discussed in a course on artificial intelligence, is to use A* search
  - A* search relies on a domain-specific heuristic to estimate the path cost from each other node to d
- The textbook also discusses the all-pairs shortest path problem
  - The goal of this problem is to find the shortest path from each node to every other node
  - If we can assume there are no negative-cost edges and the graph is sparse, I don't think we can do much better than applying Dijkstra's algorithm |V| times
  - If negative-cost edges are allowed, one possibility is to apply the Bellman-Ford algorithm |V| times; however, the cost would likely be prohibitive
  - The textbook discusses another solution that relies on dynamic programming in a later chapter

Minimum Spanning Trees

- A **spanning tree** for an undirected graph, G, is a tree that connects all the vertices in G
- The **minimum spanning tree** is the spanning tree such that the sum of the costs of all edges in the tree is as low as possible
- The concept of a minimum spanning tree for a directed graph exists, but the analogous problem is less common, and the algorithms are different (we will not cover that)
- Often there are multiple minimum spanning trees that share the same cost
- If so, algorithms to determine a minimum spanning tree are generally free to pick any one of the optimal spanning trees
- A minimum spanning tree only exists if the graph is *connected*
- A robust algorithm would determine if the graph is not connected, but the book's algorithms assume a connected graph
- From another source, a few applications of minimum spanning trees relate to communication networks, transportation networks, electrical grids, approximation algorithms, etc.
- Out textbook's example is wiring a house with minimal wire, but I don't think that is a great example
- The costs of edges can represent different things for different domains; for electrical grids, for example, they could represent distance, or they could represent monetary cost

Minimum Spanning Tree Example



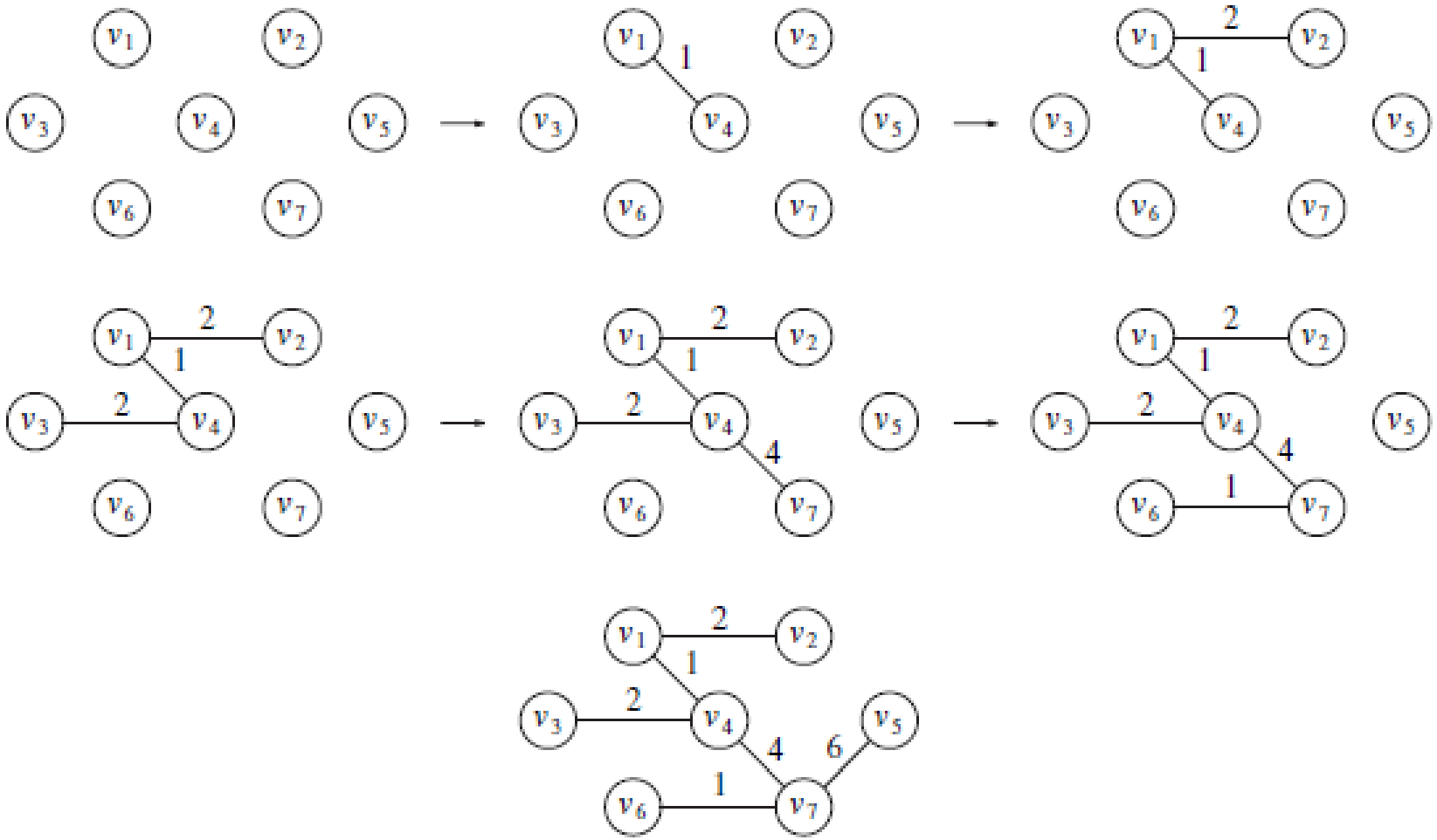
Facts About Minimum Spanning Trees

- The number of edges in any minimum spanning tree, or a more general spanning tree, is |V| - 1
- If any edge, e, is added to any spanning tree, T, a cycle is created; the removal of any edge on the cycle reinstates the spanning tree property
- If e has a lower cost than the removed edge, then the cost of the spanning tree is lowered
- There are certain algorithms that, following specific rules, iteratively select a minimum remaining edge according to the rules, constructing a minimum spanning tree in the process
- Two important concepts that are not mentioned in this context in our textbook are cuts and crossing edges
- Let G be an undirected weighted graph partitioned into two sets of vertices V_1 and V_2; this is often referred to as a **cut** in a graph
- Any edge that connects two vertices across a cut is called a **crossing edge**
- Let e be the minimum-weight crossing edge that connects V_1 and V_2 for some cut; then e is a valid edge to add as we are constructing a minimum spanning tree, T
- We can easily prove this with an indirect proof
  - Add e, the minimum crossing edge across some cut of a graph, to any spanning tree, T', that does not contain e
  - This creates a cycle containing at least one other crossing edge, e', where e' must have a cost greater than or equal to e
  - Removing the other crossing edge leads to another spanning tree with at least as low of a cost as T'

Prim’s Algorithm

- The algorithm we will cover to determine a minimum spanning tree is known as **Prim's algorithm**, a.k.a. *the Prim-Jarnik algorithm*
- The textbook also covers a second algorithm, called Kruskal's algorithm, but we will skip that
- The steps of Prim's algorithm are very simple:
  - Start with any vertex, s, and add it as the first vertex (we can consider it the root) of the spanning tree
  - Then repeatedly add to the spanning tree the minimum edge connecting a vertex already in the spanning tree to a vertex not yet included in the spanning tree
  - Also add the vertex on the other end of this edge to the spanning tree
  - Continue this procedure until all vertices are in the spanning tree
- Note that this is guaranteed to lead to a valid spanning tree because of what we previously said about cuts and crossing edges
  - At the start of any pass, we can say that V_1 is the set of all vertices in the spanning tree so far, and V_2 is the set of all other vertices
  - At every step, we are adding the edge that is the minimum crossing edge across the cut formed by V_1 and V_2

Prim’s Algorithm Example (starting with v_1)



Prim’s Algorithm vs. Dijkstra’s Algorithm

- An astute student may notice that the implementation of Prim’s algorithm would be extremely similar to that of Dijkstra’s algorithm
- We can consider the vertices already in the spanning tree to be known, and the others to be unknown
- The d-value for each unknown vertex represents the cost of the minimum edge connecting it to any known vertex
- For known nodes, the d-values are irrelevant to the algorithm, but they represent the cost of the edge that was used to add the vertex to the spanning tree
- The p-value for each unknown vertex is the known vertex responsible for the d-value
- The p-values for each known vertex is its parent of the vertex in the spanning tree
- The update rule becomes: $d_w = \min(d_w, c_{v,w})$
- Other than the interpretation of the variables, the update rule is the only thing that changes
- Therefore, as with Dijkstra’s algorithm, if we use a binary heap and adjacency lists, the running time will be $O(|E| * \log |V|)$
- Since we are now dealing with an undirected graph, each edge is stored in two adjacency lists

Prim's Algorithm Pseudo-Code (vs. Dijkstra)

```
MinimumSpanningTreePrim (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if c_v,w < d_w
        d_w ← c_v,w
        p_w ← v
```

```
WeightedPLDijkstra (Graph G, Vertex s)
  for each vertex v in G
    d_v ← ∞
    known_v ← FALSE
  d_s ← 0
  p_s ← NULL
  while there are still unknown vertices
    v ← the unknown vertex with the
        smallest d-value
    known_v ← TRUE
    for each edge from v to vertex w
      if d_v + c_v,w < d_w
        d_w ← d_v + c_v,w
        p_w ← v
```

Network Flow Problems

- The next subtopic involves what are known as **network flow problems**
- The *directed graphs* used for these problems are sometimes called **flow networks**
- We will cover this topic in more detail than the textbook
- When dealing with a flow network, two vertices are designated to be the **source**, s, where *material* is produced and the **sink**, t, where material is consumed
- We are using the term material very generally; e.g., it could be information
- The source produces material at a steady rate, and the sink consumes material at the same rate; the material is said to *flow* from the source to the sink
- Each directed edge can be thought of as a conduit for a material; vertices represent conduit junctions
- The edge weights, $c_{v,w}$, represent *capacities*
- Each edge also has a current flow, $f_{v,w}$; it must be the case that: $f_{v,w} \leq c_{v,w}$
- By convention, we will also specify that: $f_{v,w} = -f_{w,v}$

Conservation of Flow

- As previously mentioned, each directed edge can be thought of as a conduit for a material, and vertices represent conduit junctions
- Recall that material is produced at the source and consumed at the sink
- Other than the source and the sink, material flows through vertices without collecting in them
- In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex
- This is known as the property of *flow conservation*
- Formally, flow conservation can be expressed as: $\forall_{v \epsilon V}, \sum_{w \epsilon V} f_{v,w} = 0$
- Note that this formula relies on the convention that $f_{v,w} = -f_{w,v}$

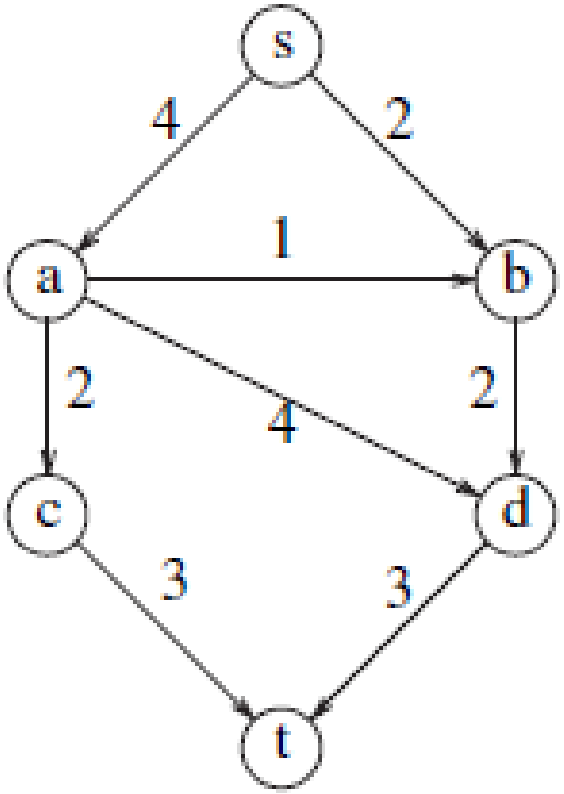
Flow Network Applications

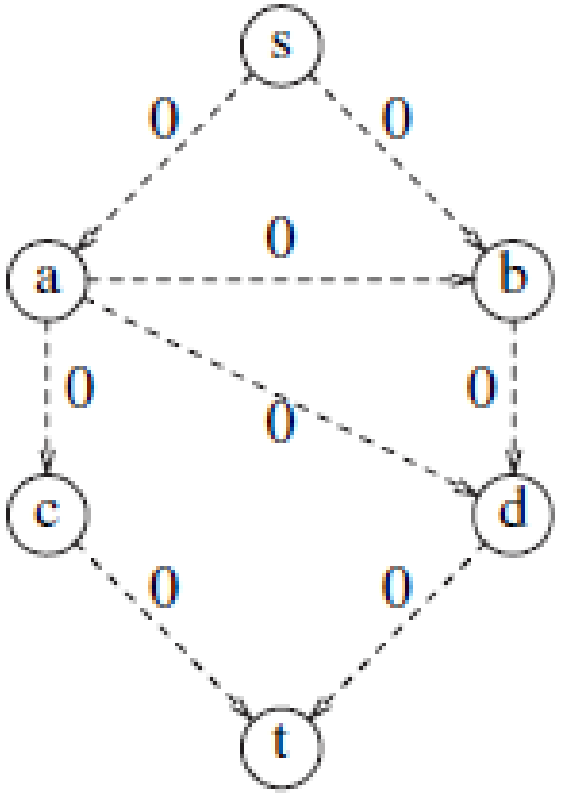
- A flow network can be used to model various things; for example:
  - Liquids flowing through pipes
  - Parts through assembly lines
  - Current through electrical networks
  - Information flowing through communication networks
  - Traffic flowing between intersections
  - Merchandise that must be transported to a destination
- The algorithm we consider assumes we are trying to maximize the flow of material from the source, s, to the sink, t, in a flow network

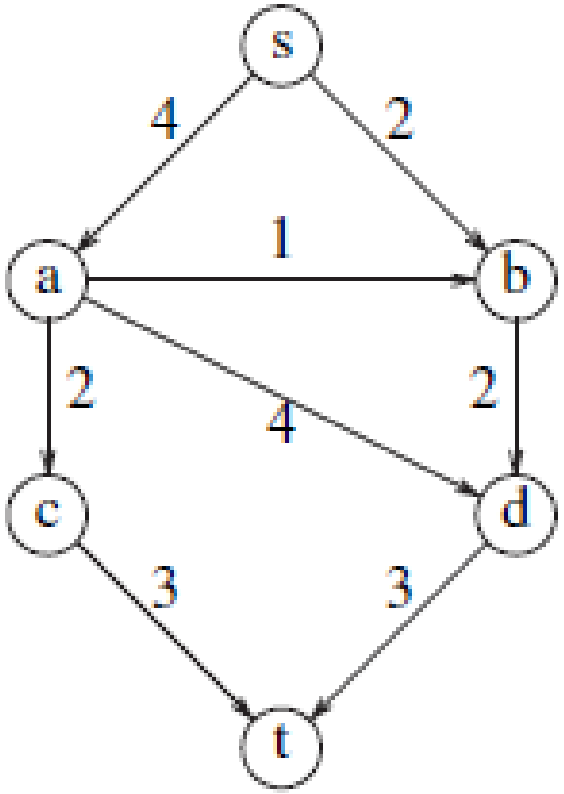
The Maximum Flow Problem

- The **maximum-flow problem** asks us to determine the greatest rate at which material can be shipped from s to t without violating any constraints
- The textbook uses three types of graphs to discuss the steps for finding a solution:
  - The **graph** is just the given flow network; it indicates the capacities of edges, and this graph never changes during the execution of the algorithm we will cover
  - The **flow graph** indicates the current flow along each edge; I have seen some other sources combine the graph and the flow graph
  - The **residual graph**, a.k.a. *residual network*, shows how much additional flow can be pumped along each edge; all the edges in this graph have positive weights
- Initially, the flow graph has zeros for every edge, and the residual graph is the same as the original graph

Example of Initial Graphs



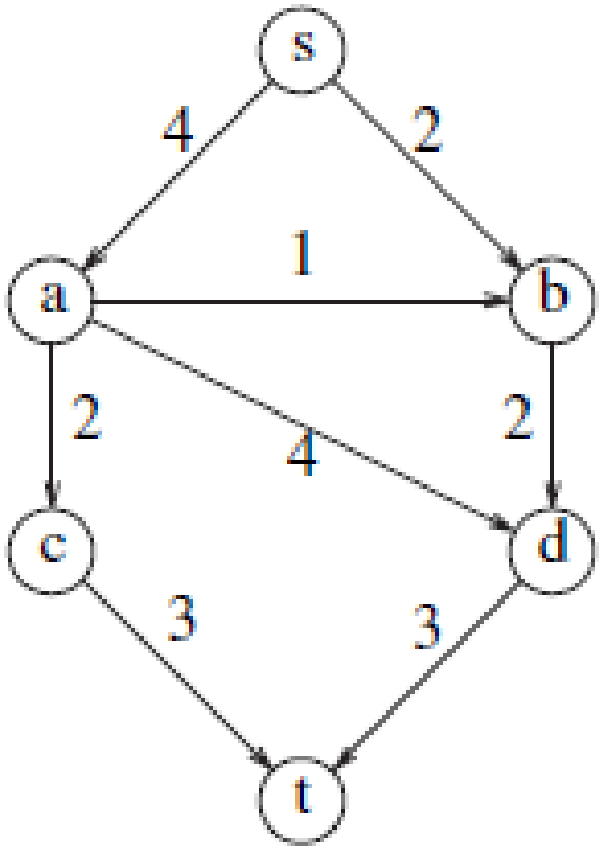


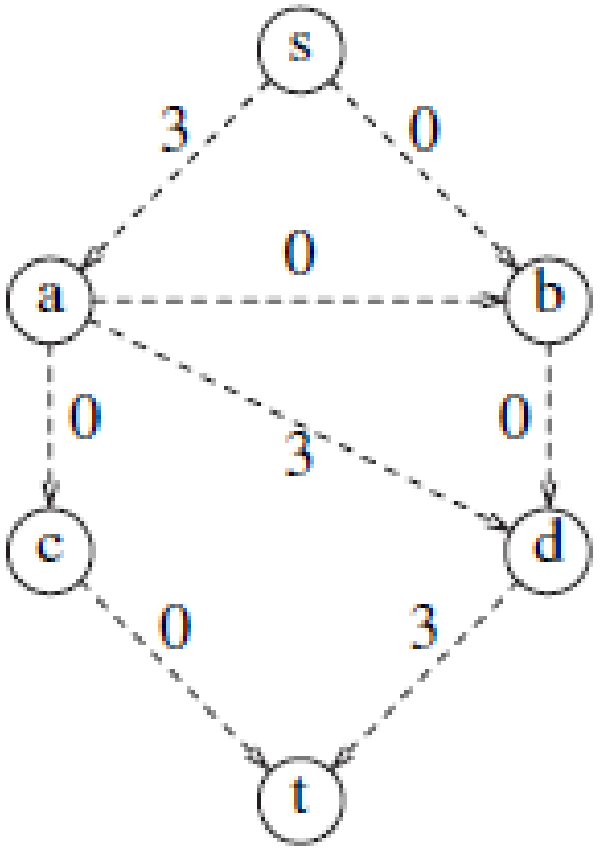


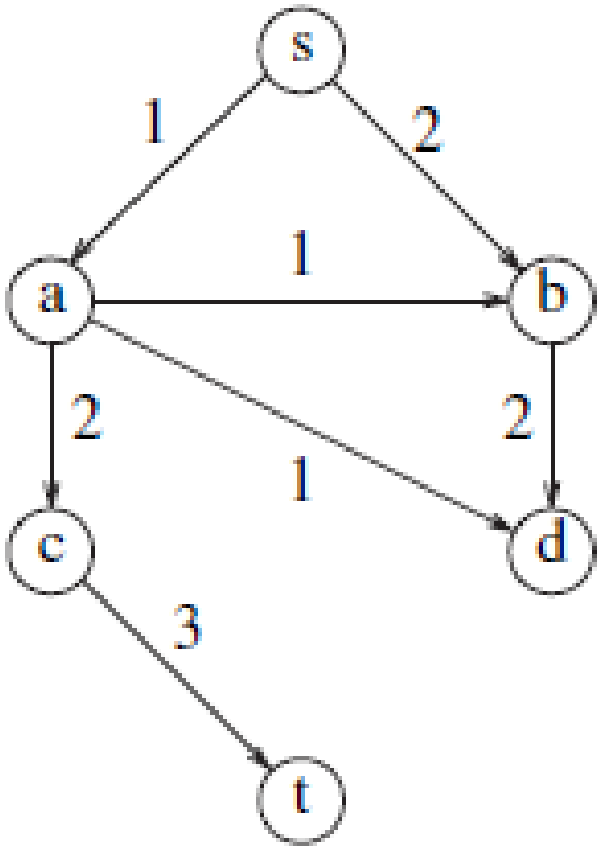
Augmenting Paths

- At any time, a path from the source to the sink in the residual graph is called an **augmenting path**
- Any augmenting path represents a valid way to push more flow from the source to the sink
- We will see from an example that when we pump material along a path, we need to add additional *back edges* in the residual graph
- The back edges indicates that we are allowed to undo flow, which is equivalent to pumping flow in the opposite direction
- If we do not allow this, in certain circumstances, we might saturate our graph with a suboptimal solution
- For example, consider what happens when we start by pumping 3 units of material along the augmenting path s 🡒 a 🡒 d 🡒 t in our example graph
  - Without the back edges, only one additional unit of flow can be pumped along the path s 🡒 a 🡒 c 🡒 t, leading to a total flow of 4, which is suboptimal (the figure on the next slide helps to visualize this)
  - With the back edges, 2 additional units of flow can be pumped along the path s 🡒 b 🡒 d 🡒 a 🡒 c 🡒 t, leading to a total flow of 5, which is optimal (the figure in two slides helps to visualize this)

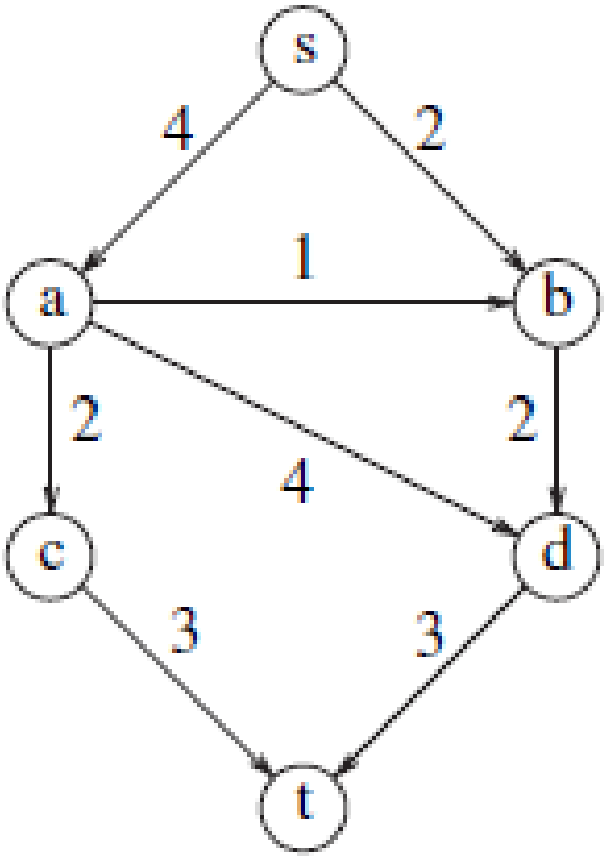
Residual Graph without Back Edges

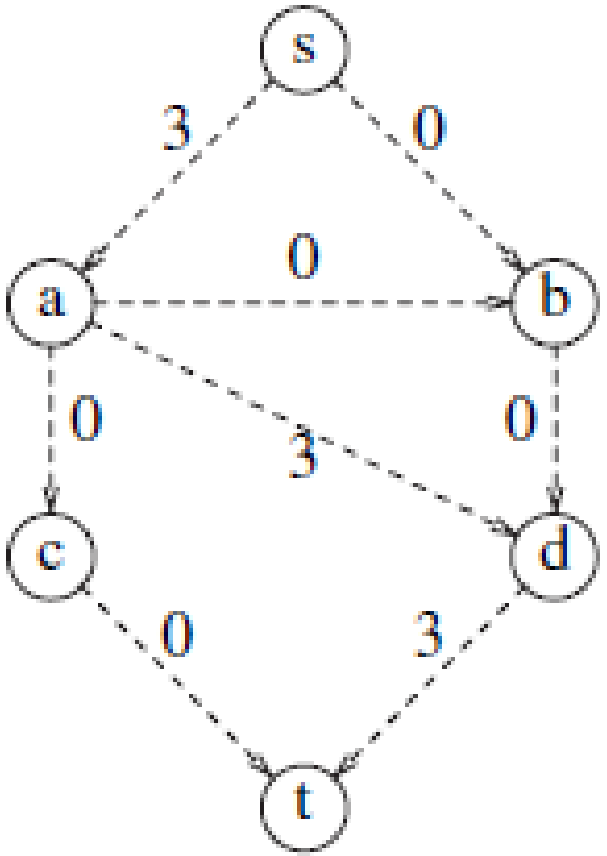


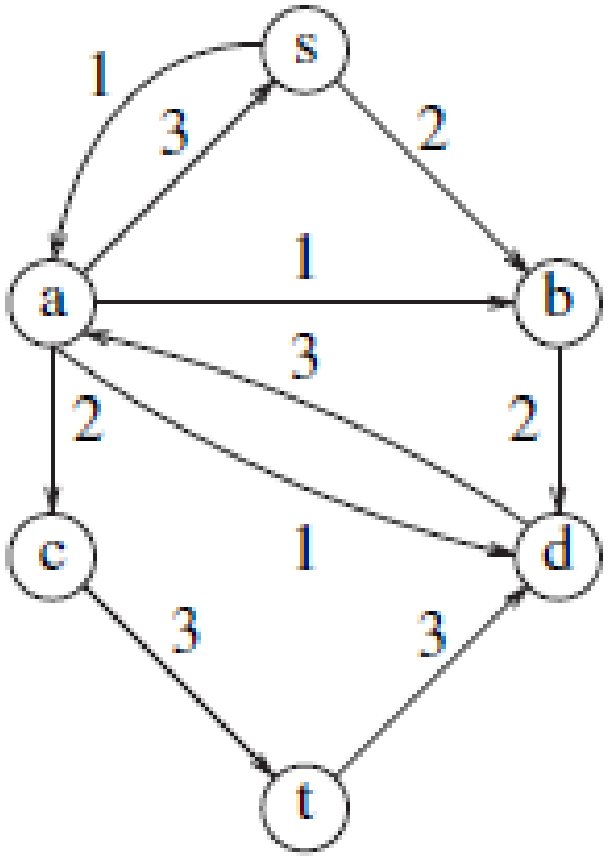




Residual Graph With Back Edges







The Ford-Fulkerson Method

- An effective approach for solving the maximum-flow problem is known as the **Ford-Fulkerson method**, a.k.a. the *augmenting-path method* (the name is not mentioned in the textbook)
- Note that this is called a method instead of an algorithm, because it is quite general, and there are actually several algorithms that can be used to implement this method
- My pseudo-code for the method is as follows:

```
FordFulkersonMethod (Graph G, source s, sink t)
  initialize all flows to 0
  while there exists an augmenting path p
    augment the flow f along p as much as possible
```

- We still need to decide which augmenting path to use when multiple such paths exist
- No matter how we choose the paths, the final solution will be optimal (we will discuss why soon)
- However, some methods of choosing the augmenting paths will lead to more efficient running times than others (we will mention a couple of reasonable possibilities shortly)

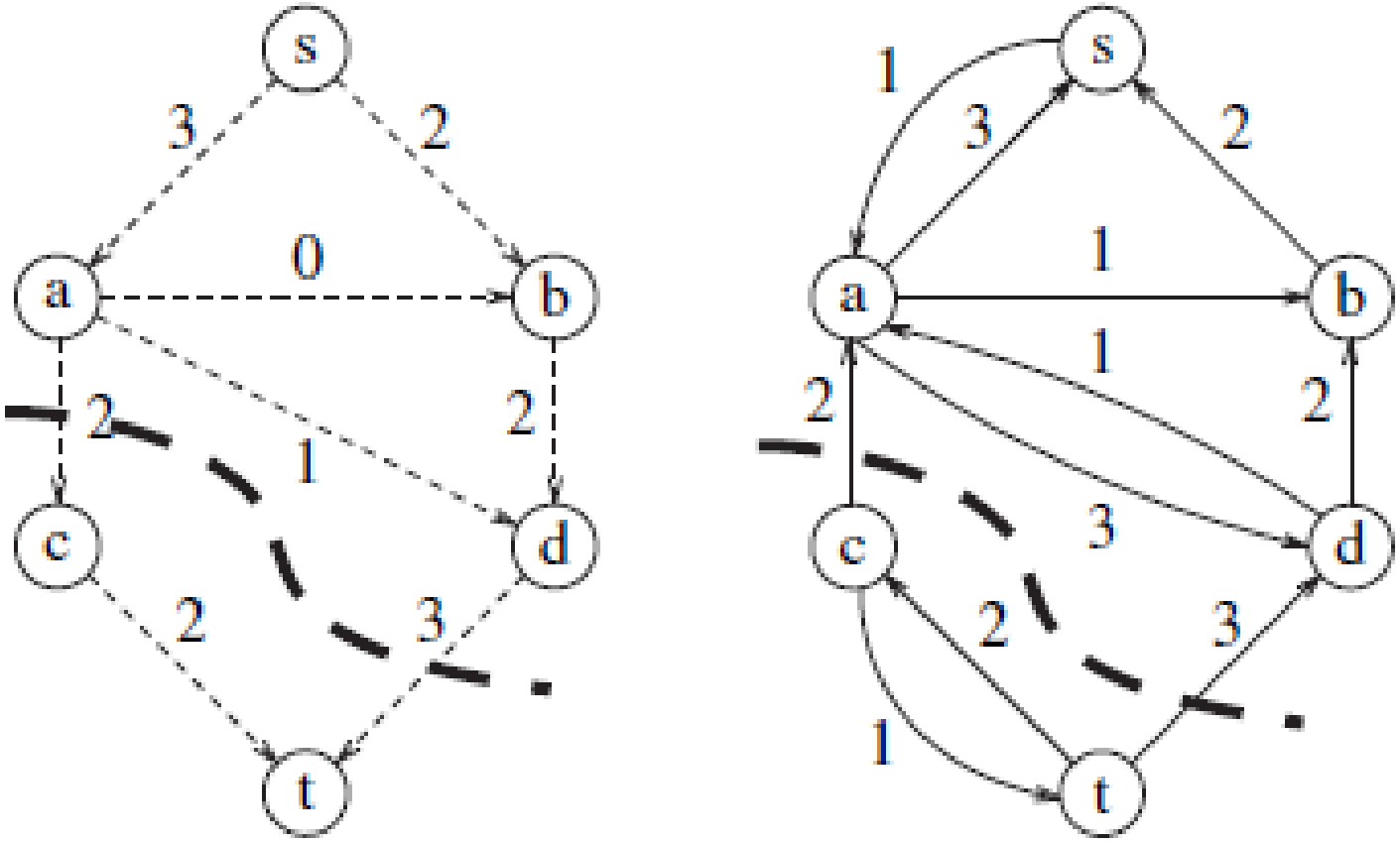
Cuts in Flow Networks

- Remember the definition of a **cut**, which is a partitioning of the vertices of a graph into two sets
- Let (S, T) be any cut of the graph G such that the source, s, is a member of S and the sink, t, is a member of T
- The flow from s to t is also the net flow across any such cut
- There are $2^{|V|-2}$ such cuts
- This means that the value of any flow in a flow network G is bounded by the capacity of any such (S, T) cut of G

The Max-flow, Min-cut Theorem

- The **max-flow, min-cut theorem** (not mentioned in our textbook) states that the maximum possible flow in a flow network is equal to the minimum capacity of all (S, T) cuts
- We are not going to fully prove the max-flow, min-cut theorem, but we will use it to prove that the Ford-Fulkerson method leads to a correct solution to the maximum-flow problem
- Also, we have already discussed why half of what the max-flow, min-cut theorem states is obvious, and that is the only part of the theorem that we need to rely on here
- Consider the situation after the Ford-Fulkerson method terminates
  - Then consider the cut (S, T) such that S contains all the vertices that can be reached starting form s (the source) in the residual network and T contains all other vertices
  - Clearly, the sink, t, must be in T, or else there would be an augmenting path in the residual graph and the Ford-Fulkerson method would not yet terminate
  - The flow through this cut must be equal to the current flow through the graph
  - Also, this cut must be at its full capacity (it is said to be saturated), or else there would be some vertex in T that should have been added to S
  - This must be the maximum possible flow, since we cannot exceed the capacity of any cut
- The figure on the next slide displays the flow graph, along with a saturated cut in the residual graph, at the end of the process for our sample graph

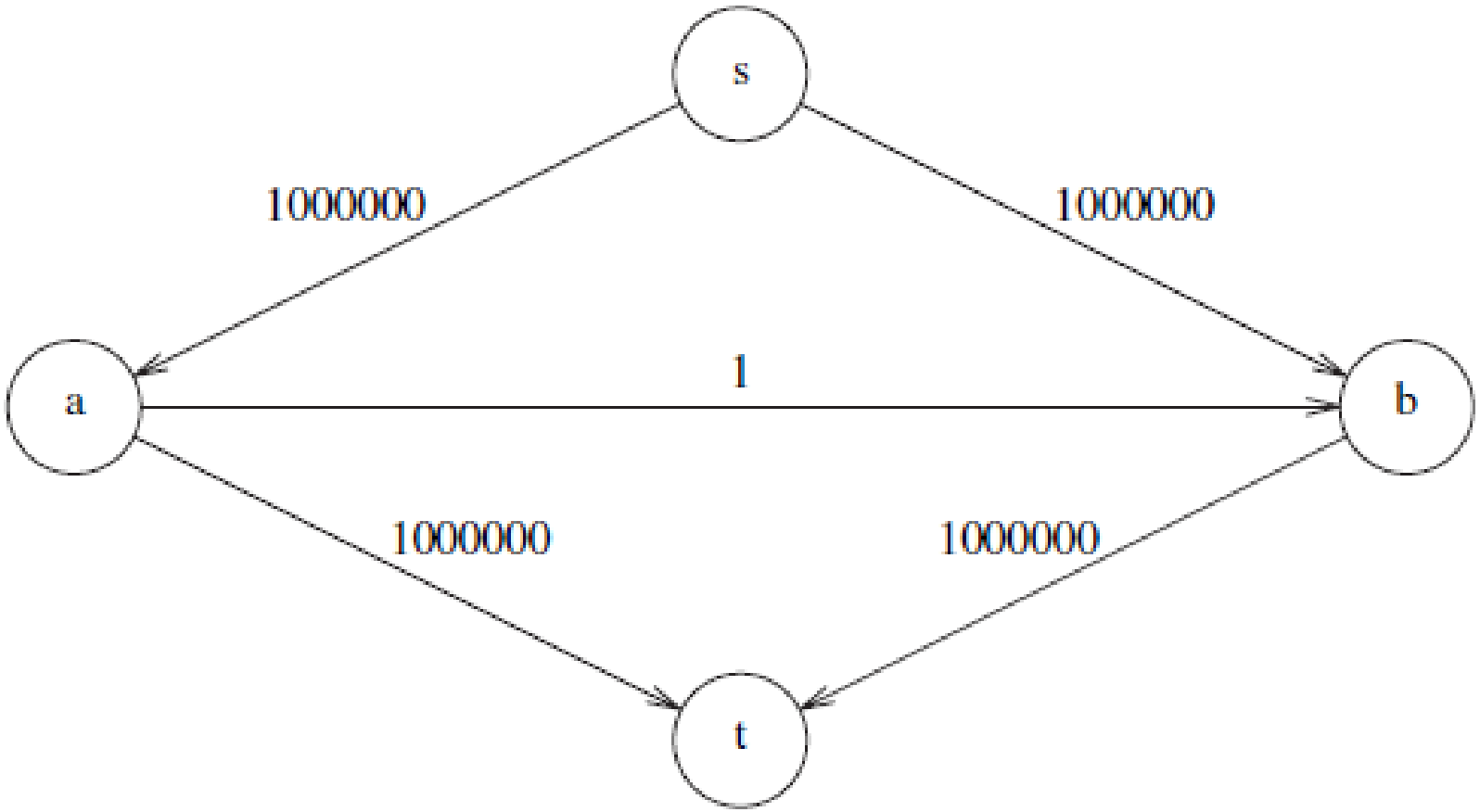
Saturated Cut Example



How to Choose Augmenting Paths

- We still need a good way to choose which augmenting path to use for the for Ford-Fulkerson method, as the previous example illustrates
- An example of a bad strategy would be to choose the path with the largest number of nodes, as demonstrated by the graph on the next slide
- Two reasonable solutions are:
  1. Choose the augmenting path that leads to the biggest increase in flow (this is known as the *Ford-Fulkerson algorithm*)
  2. Choose the augmenting path with the fewest number of nodes (this is known as the *Edmunds-Karp algorithm*)
- Implementations using choice 2 involve a *breadth-first search* from the source to find the augmenting path
- We will not prove it, but it can be shown that an algorithm implemented this way has a worst-case running time that is $O(|V| * |E|^2)$
- More advanced methods have been found that improve the running time; according to our textbook, the best-known bound for this problem is $O(|E| * |V|)$
- Given any reasonable implementation of the Ford-Fulkerson method, if all weights are integers, it is simple to show that the running time is at most $O(|E| * f)$, where f is the maximum flow
- This assumes that each iteration will examine at most $|E|$ edges, finding an augmenting path in $O(|E|)$ time, and will increase the flow by at least one

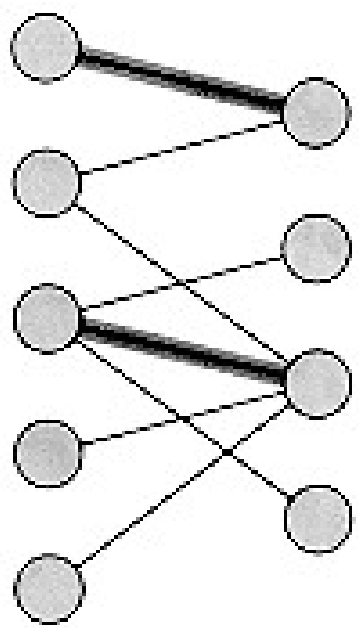
Contrived Flow Network



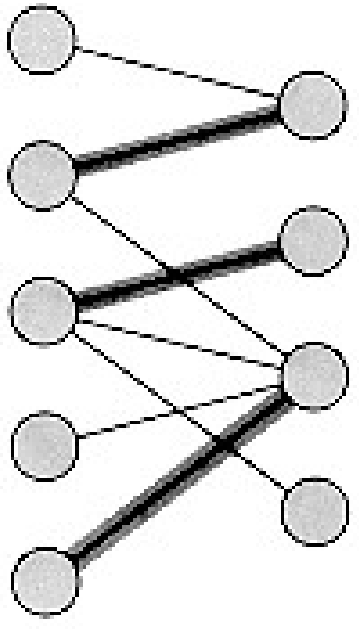
The Maximum-Bipartite-Matching Problem

- Next, we will discuss the **maximum-bipartite-matching problem** (this problem is not mentioned in our textbook)
- A **bipartite graph** is a graph for which the vertices can be decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent
- Consider two columns of vertices; some vertices in the left column are connected by undirected edges to some vertices in the right column
- An example of a bipartite graph is shown on the next slide
- The two sets of vertices might represent employees and jobs, teachers and classes, etc.
- An edge represents a potential mutually beneficial match (e.g., an employee who can do a job, or a teacher who can teach a class)
- However, each vertex on the left can only be matched to a single vertex on the right, and each vertex on the right can only be matched to a single vertex on the left
- A *matching* is a set of edges that touch each vertex at most once (so no two edges are incident on the same vertex)
- The problem is to determine the matching that uses the maximum number of edges

Example from “Introduction to Algorithms”



(a)



(b)

Straight-forward Approaches Don’t Work Well

- It may seem like a solution is to start with an empty matching, and to keep adding edges until no more can be added without violating a constraint
- The previous simple example shows us that this is not guaranteed to lead to an optimal solution
- Note that the matching in part (a) has a size of 2, and no more edges can be added to this matching, but this is suboptimal
- The matching in part (b) has size 3, and this is optimal for the example bipartite graph
- One possible method to find the optimal solution would be to cycle though all possible subsets of edges to find the optimal, valid subset
- This algorithm, however, would require exponential time

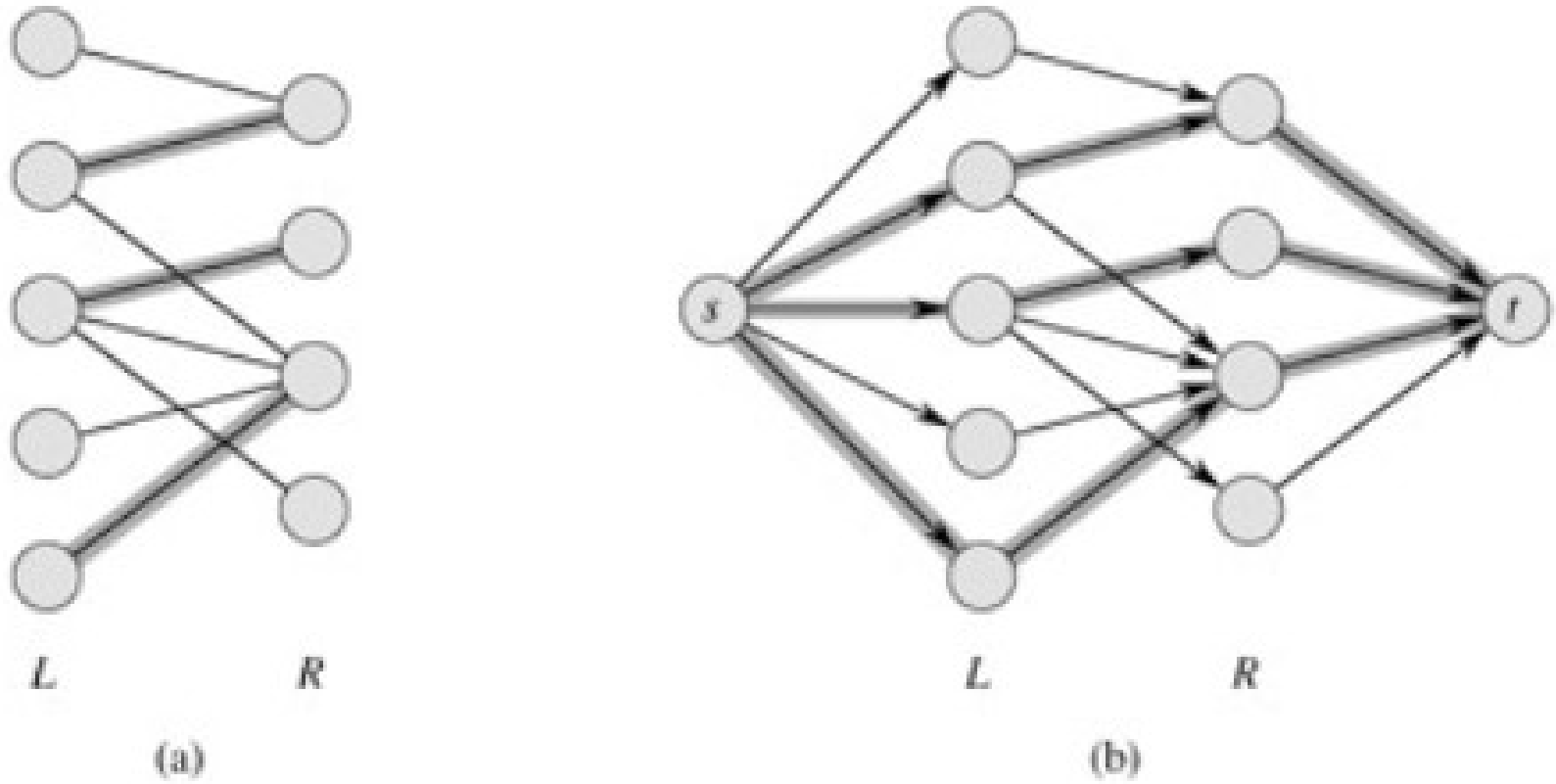
Reductions

- In computer science, it is often the case that an instance of one problem can be *reduced* to an instance of another
- The process is called **reduction**
- If a new problem is reduced to an already-solved problem, a solution to the solved problem can be applied to the new problem as well
- Sometimes, the two problems will not appear similar to each other on the surface
- This notion will be very important when we discuss *complexity classes* and *NP-complete problems* later in the course

An Interesting Reduction

- Without much difficulty, *we can reduce the maximum-bipartite-matching problem to the maximum-flow problem*!
- First, consider all edges to be directed from left to right
- Then, add two new vertices:
  - A source, s, with outgoing directed edges to all left vertices
  - A sink, t, with incoming directed edges from all right vertices
- Finally, assign every edge a weight, or capacity, of one
- Now, the edges used to solve this instance of the maximum-flow problem also solves the maximum-bipartite-matching problem
- An example of this reduction is shown on the next slide

Example Reduction



Analyzing the Solution

- Remember that every edge has been assigned a capacity of 1
- Any solution to the maximum-bipartite-matching problem with n edges will also lead to a flow of n through the flow network
- Any solution to the maximum-flow problem with a flow of n will leads to a matching of size n
- Recall that when all weights are integers, any reasonable implementation of the Ford-Fulkerson method has a running time is at most O(|E| * f)
- Here, f is the maximum flow through the flow network
- For flow networks resulting from reductions from the maximum-bipartite matching problem, the maximum flow will have a size that is O(|V|)
- Therefore, any reasonable implementation of the Ford-Fulkerson method, applied to these graphs, will have a running time that is O(|E| * |V|)

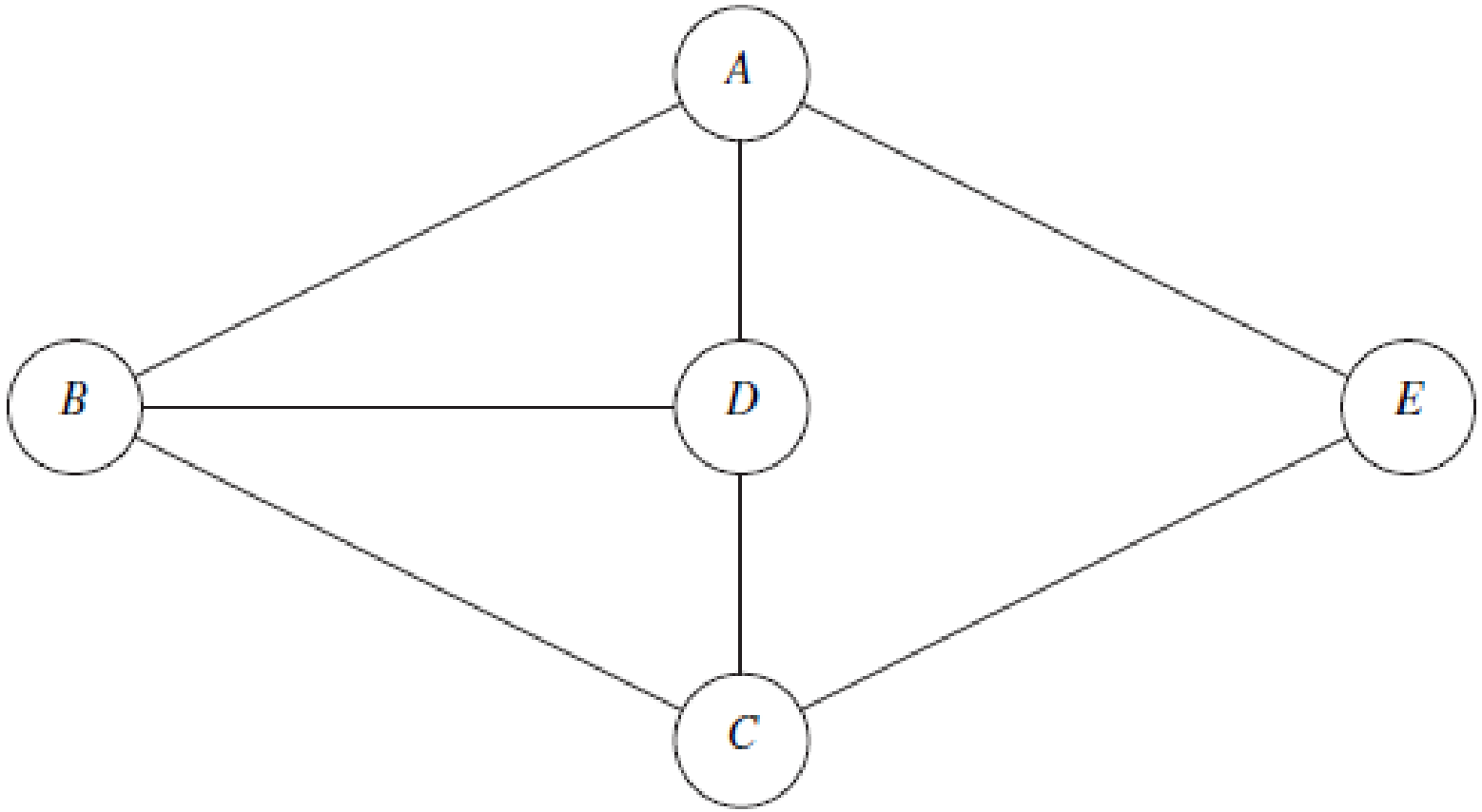
Depth-first Search

- The last subtopic involving graphs that we will discuss is **depth-first search** (**DFS**)
- A depth-first search of a graph is very similar to a *preorder traversal*
- We visit the current node first, then we recursively visit all its adjacent neighbors that have not already been visited
- To avoid infinite loops and generally improve the time complexity of DFS, we must have a way to mark a node as visited
- For a finite graph in which all the nodes have their own data structure, we can simply use a Boolean field for this purpose
- We will consider depth-first search applied to undirected graphs
- We will assume that adjacency lists are used, and that each edge exists in both incident vertices' adjacency lists
- Assuming these things, every node will be visited at most once and every edge will be traversed at most once
- A simple, usually recursive, implementation will have a worst-case running time that is O(|V| + |E|)
- We will discuss two applications of DFS, but there are many others; e.g., in my AI course, DFS is an important component of the main algorithm discussed for implementing strategic, AI game-playing programs

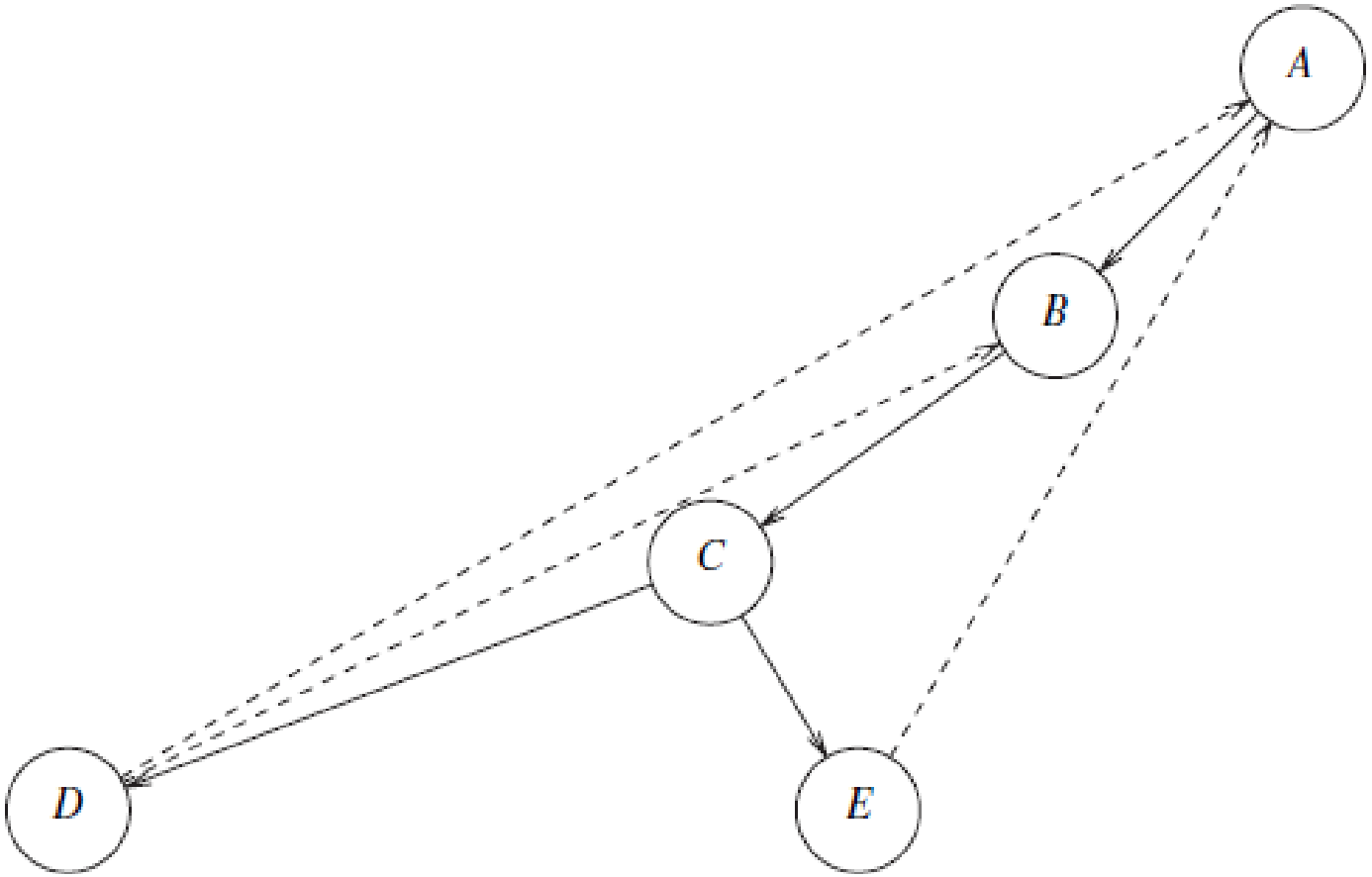
Depth-first Spanning Trees

- As we traverse the nodes of a graph using a depth-first search, we will call the edges that are actually followed **tree edges**
- We will call any edge that would lead back to a node that was already visited a **back edge**
- The existence of any back edge indicates a cycle in the graph
- We will graphically draw a tree using the tree edges, and depict the (not followed) back edges using dashed lines
- This graphical depiction that demonstrates the order in which nodes are visited is called a **depth-first spanning tree**
- During the depth first search, assuming the graph is connected, every edge becomes either a tree edge or a back edge in the depth-first spanning tree
- An example of a graph and its depth-first spanning tree are shown on the next two slides

Undirected Graph Example



Depth-first Spanning Tree Example



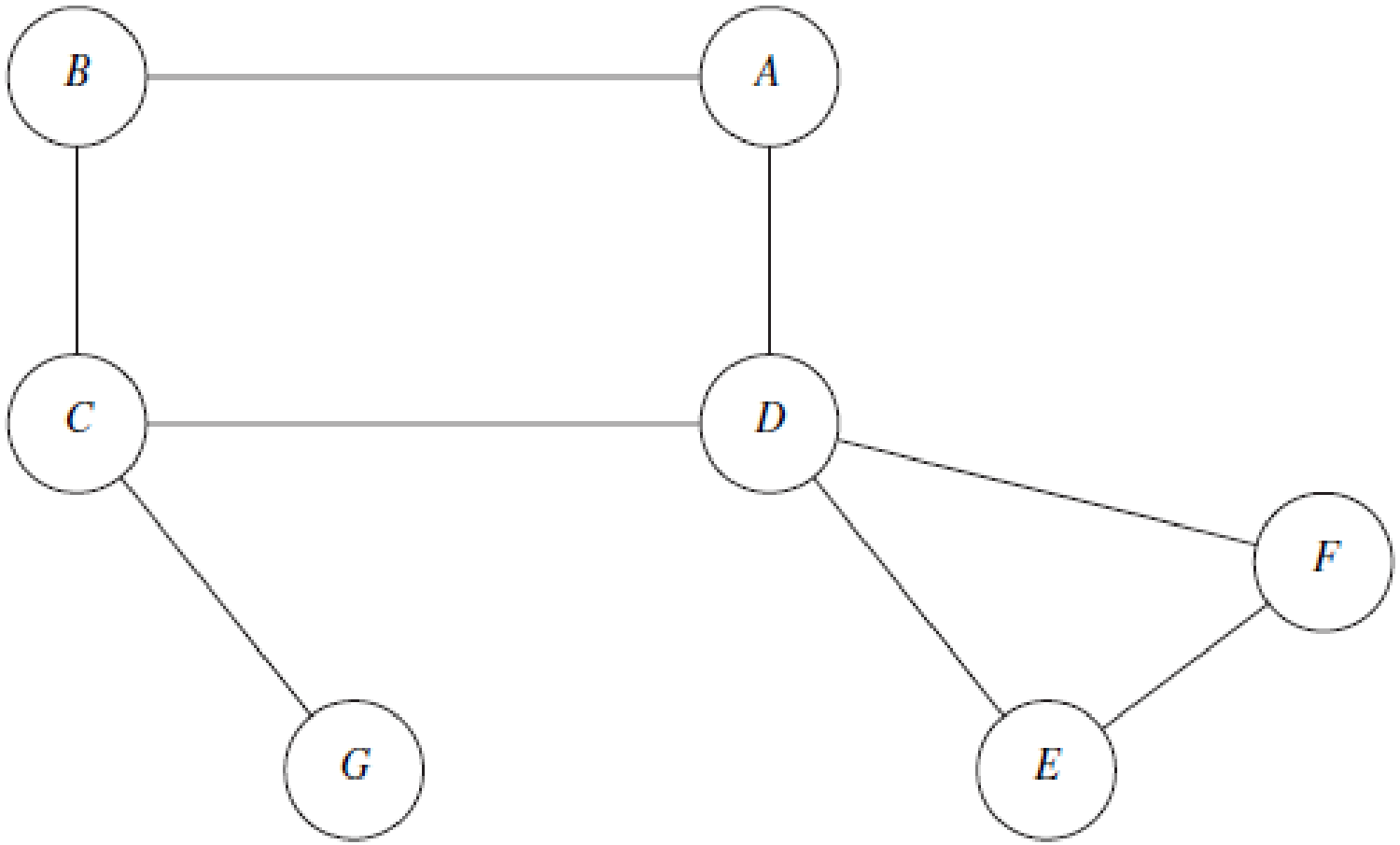
Biconnectivity

- One simple use of a DFS is to check if an undirected graph is connected; simply perform a DFS and see if there is any node that is not visited
- A harder problem is to check for **biconnectivity** in an undirected graph
- A graph is biconnected if and only if there are no vertices whose removal disconnects the rest of the graph
- If the graph is not biconnected, the vertices whose removal would disconnect the graph are known as **articulation points**
- Therefore, a graph is biconnected if and only if there are no articulation points
- A possible application: Determine if there are there any computers in a network that, if they went down, would cause communication failures

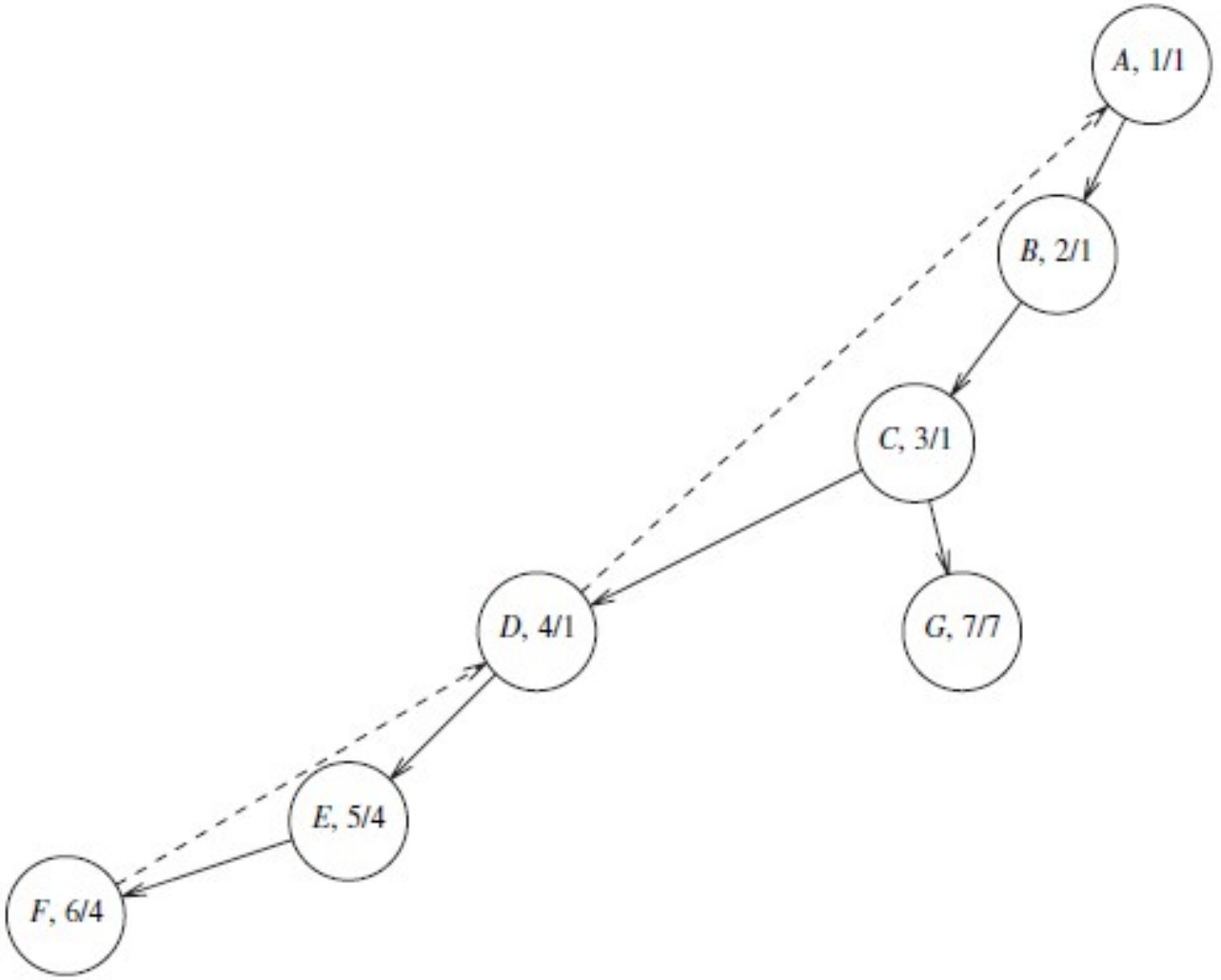
Num and Low Values

- For each vertex, v, in a depth-first spanning tree, we will define the following two parameters:
  - Num(v) = the position of v if you number the vertices based in the order in which they are visited in the DFS (the root is assigned the value 1)
  - Low(v) = the lowest number, Num(w), of all vertices w that can be reached starting at v using zero or more tree edges and optionally one back edge (which must be used last)
- Equivalently, we can define Low(v) to be the minimum of these three values:
  1. Num(v)
  2. The lowest Num(w) among all back edges (v, w)
  3. The lowest Low(w) among all tree edges (v, w)
- The values defined in 1 and 2 above can be evaluated by looking at v and the edges leaving v (stored in an adjacency list)
- The definition in 3 is recursive
- We need to evaluate the Low of the children before the Low of the current node; this requires a *postorder traversal*, which is also based on a DFS
- An example of a graph and its depth-first spanning tree including Num and Low values are shown on the next two slides

Another Undirected Graph Example



Example with Num and Low Values

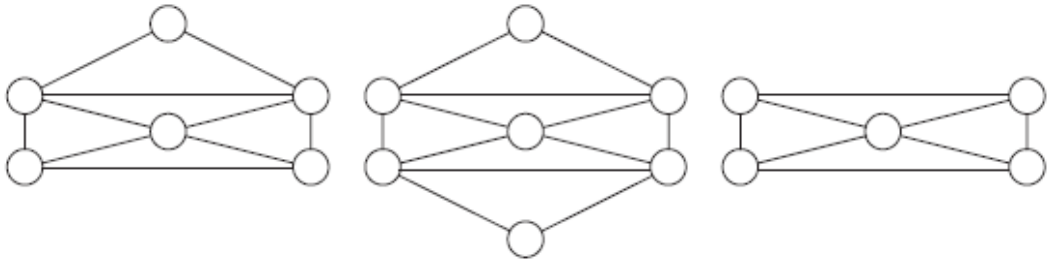


Detecting Articulation Points

- The root of the DFS spanning tree is an articulation point if and only if it has more than one child (the reasoning for this should be clear)
- For any other vertex, v, v is an articulation point if and only if v has some child w such that Low(w) ≥ Num(v)
  - The "if" part is clear enough, and applies to both C and D in the previous example
  - That is, for any child, w, of a node, v, such that Low(w) ≥ Num(v), w can only get to ancestors of v by going through v
  - The "only if" part may be less obvious; that is, why does this algorithm find all articulation points?
  - The reason is that if all the children have Low(w) < Num(v), then they (and their descendants) have alternate routes to the top part of the tree

Euler Tours and Euler Circuits

- An **Euler tour**, a.k.a. an *Euler path*, is a path in a graph that visits (i.e., traverses) every edge exactly once
- An **Euler circuit**, a.k.a. an *Euler cycle*, is an Euler tour that starts and ends on the same vertex
- An Euler tour can only exist in an undirected graph if zero or exactly two vertices have an odd degree (i.e., an odd number of incident edges)
- It turns out that this is also a sufficient condition, although we won't prove it
- An Euler cycle can only exist in an undirect graph if every vertex has an even degree; again, we won't prove it, but this is also a sufficient condition
- Clearly, we can check for these conditions in O(|E| + |V|) time
- In the following graphs, the first has only Euler tours possible, the second has Euler tours and circuits possible, and the third has neither possible:



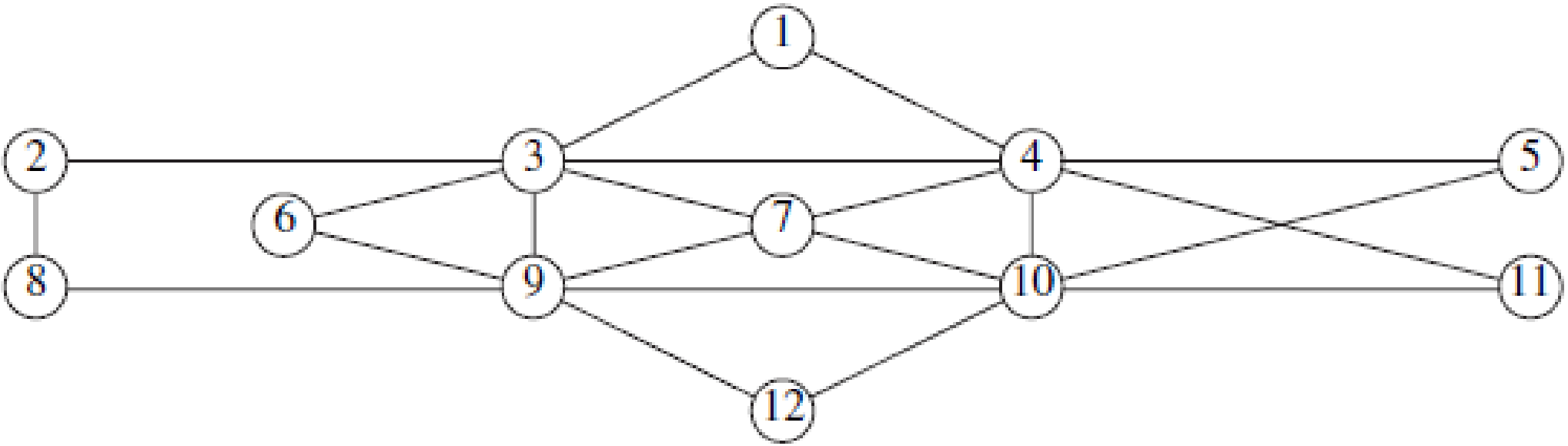
Finding an Euler circuit

- We will focus on the problem of finding Euler circuits (but the algorithm for finding Euler tours is very similar)
- If an Euler circuit is possible, which we check for first, we can then apply the following algorithm (we will step through an example shortly):
  - Start at any vertex
  - Apply a depth-first search, stopping when we get back to the starting vertex
  - Note that we will never get stuck along the way, because every vertex has an even number of incident edges
  - When we return to the starting vertex, we might be stuck at a dead end without having visited every edge
  - If we are not stuck, we continue and find a larger cycle
  - If we are stuck, we find the first vertex, v, on the cycle that has additional unexplored edges
  - We then apply another DFS to find a cycle starting and ending at v
  - We splice this cycle into the previous cycle
  - We repeat the last few steps until we have a Euler cycle

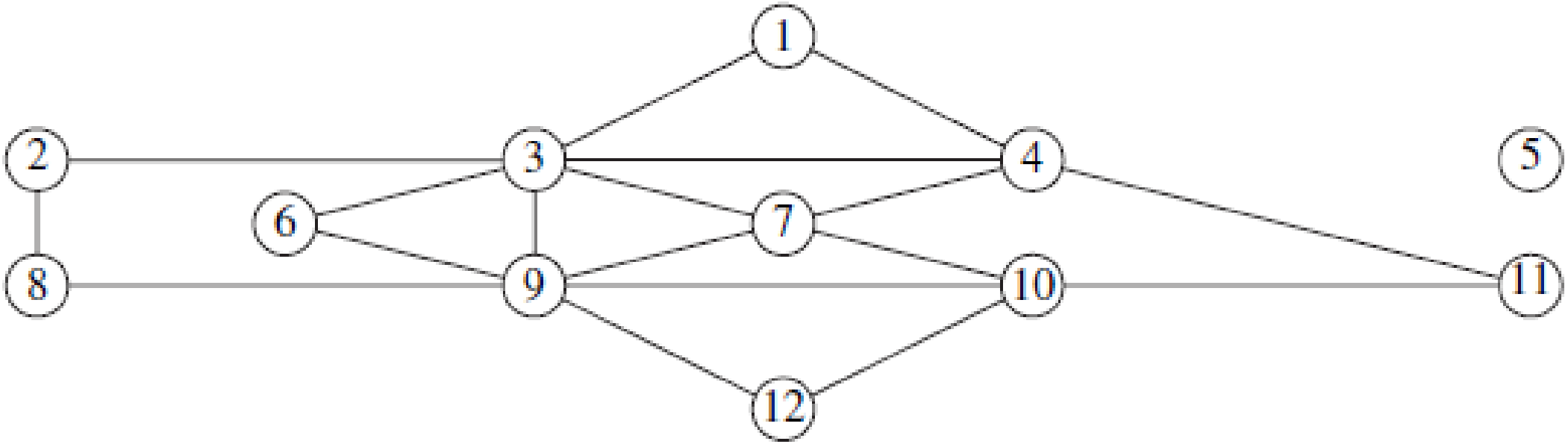
Euler Circuit Algorithm Example

- An example graph is shown on the next slide
- Suppose we start at vertex 5 and find the cycle 5, 4, 10, 5; the figure two slides forward shows the graph with the edges so far removed
- We are stuck, so we find the first vertex in the cycle that has unexplored edges; this is 4
- We apply a DFS from there; suppose we find the cycle 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4; the figure three slides forward shows the updated graph
- Splicing this into the first cycle where four was encountered gives us 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5
- We are still not done; now, the first vertex in the cycle with unexplored edges is 3
- We apply a DFS from there; suppose we find the cycle 3, 2, 8, 9, 6, 3; see the updated graph in four slides
- Splicing this into the larger cycle gives 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5
- We are still not done; now, the first vertex in the cycle with unexplored edges is 9
- We apply a DFS from there; suppose we find 9, 12, 10, 9 (after which all edges would be removed)
- Splicing this into the larger cycle gives 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5
- Now, all edges have been traversed, and we have found an Euler circuit

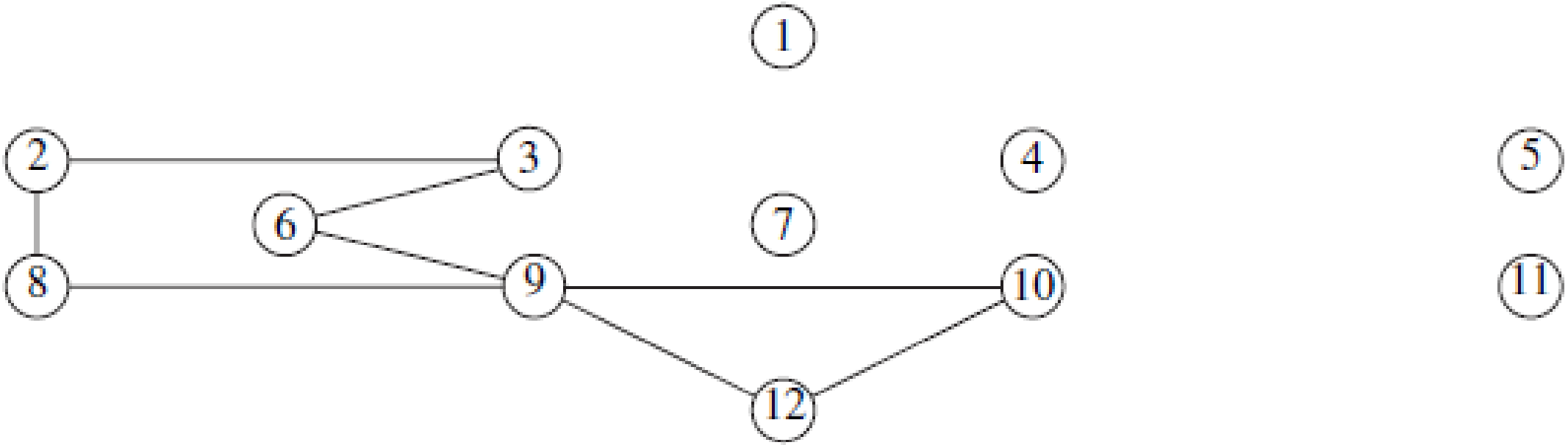
Sample Graph with Euler Circuit Possible



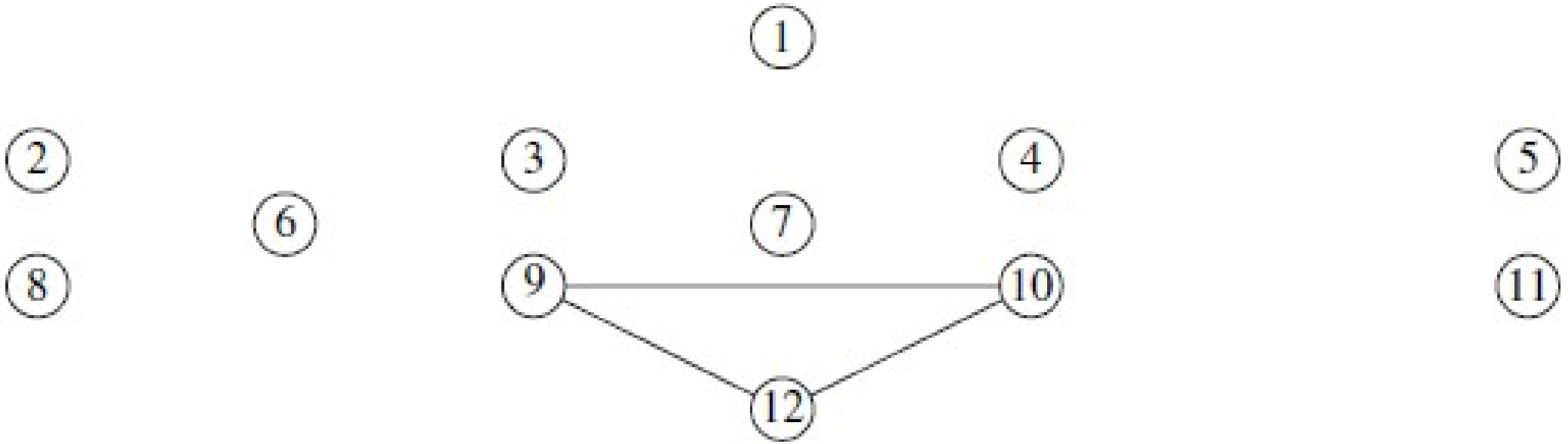
Sample Graph after One Iteration



Sample Graph after Two Iterations



Graph After Three Iterations



Some Implementation Details

- The textbook suggests using linked lists to store the cycles since splicing of shorter cycles into longer cycles is necessary
- Each node should maintain a pointer to the last explored edge in its adjacency lists to avoid repetitive scanning
- Each search to find the first vertex with unexplored edges should start at the point where the last one was found
- The textbook states that with the appropriate data structures, the total running time of the algorithm is $O(|V| + |E|)$

Hamiltonian Paths and Hamiltonian Cycles

- A **Hamiltonian path** is a path in a graph that visits every vertex exactly once
- A **Hamiltonian cycle** is a Hamiltonian path that it starts and ends on the same vertex
- The **Hamiltonian cycle problem** asks us to determine if a given graph has a Hamiltonian cycle
- This may sound very similar to the problem that asks us to find an Euler circuit in a graph, for which we have just seen an efficient solution
- We have seen that it is even simpler to determine if a Euler circuit for a graph exists, without actually finding it
- However, there is no known polynomial-time solution for the Hamiltonian cycle problem, and it is believed by most computer scientists that no polynomial solution exists
- The Hamiltonian cycle problem is an **NP-complete problem**; we will talk more about such problems as part of our final topic in the course
- A related, but even harder, problem is the **traveling salesman problem**, which asks to find the Hamiltonian cycle with the lowest cost