Graphs and Digraphs

Chapter 16

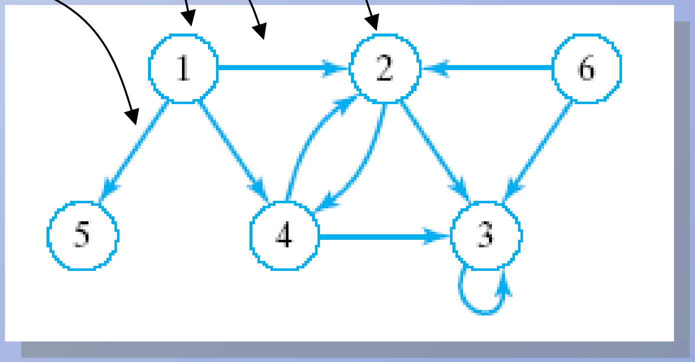
Chapter Contents

Chapter Objectives

- Introduce directed graphs (digraphs) and look at some of the common implementations of them
- Study some of the algorithms for searching and traversing digraphs
- See how searching is basic to traversals and shortest path problems in digraphs
- Introduce undirected graphs and some of their implementations

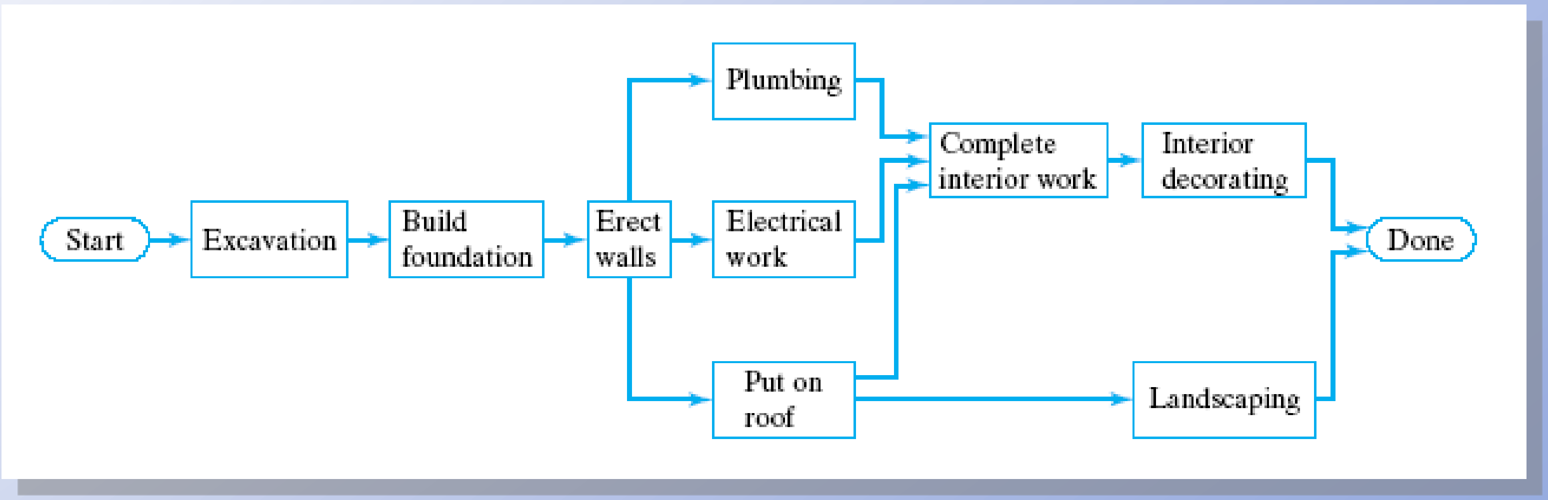
Directed Graphs

- Similar to a tree
- Consists of a finite set of elements
  - Vertices or nodes
- Together with finite set of directed
  - Arcs or edges
  - Connect pairs of vertices



Directed Graphs

- Applications of directed graphs
  - Analyze electrical circuits
  - Find shortest routes
  - Develop project schedules



Directed Graphs

- Trees are special kinds of directed graphs
  - One of their nodes (the root) has no incoming arc
  - Every other node can be reached from the node by a unique path
- Graphs differ from trees as ADTs
  - Insertion of a node does not require a link (arc) to other nodes … or may have multiple arcs

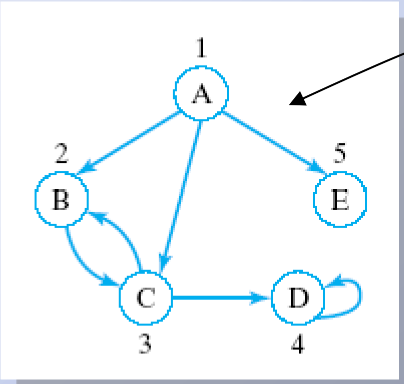
Directed Graphs

- A directed graph is defined as a collection of data elements:
  - Called nodes or vertices
  - And a finite set of direct arcs or edges
  - The edges connect pairs of nodes
- Operations include
  - Constructors
  - Inserts of nodes, of edges
  - Deletions of nodes, edges
  - Search for a value in a node, starting from a given node

Graph Representation

- <u>Adjacency matrix</u> representation
  - for directed graph with vertices numbered
    `1, 2, … n`
- Defined as `n` by `n` matrix named `adj`
- The `[i,j]` entry set to
  - `1` (true) if vertex `j` is adjacent to vertex `i` (there is a directed arc from `i` to `j`)
  - `0` (false) otherwise

Graph Representation



columns ***j***

rows ***i***

	1	*2*	*3*	*4*	*5*
1	0	1	1	0	1
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	1	0	0
5	0	0	0	0	0

- Entry [1, 5] set to true
- Edge from vertex 1 to vertex 5

Graph Representation

- Weighted digraph
  - There exists a "cost" or "weight" associated with each arc
  - Then that cost is entered in the adjacency matrix
- A complete graph
  - has an edge between each pair of vertices
  - N nodes will mean N * (N – 1) edges

Adjacency Matrix

- Out-degree of i^{th} vertex (node)
  - Sum of 1's (true's) in row `i`
- In-degree of j^{th} vertex (node)
  - Sum of the 1's (true's) in column `j`

- What is the out-degree of node 4?

- What is the in-degree of node 3?

Adjacency Matrix

- Consider the sum of the products of the pairs of elements from row ***i*** and column ***j***

adj

	1	*2*	*3*	*4*	*5*
1	0	1	1	0	1
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	1	0	0
5	0	0	0	0	0

Fill in the rest of adj^2

adj ^{2}

	1	*2*	*3*	*4*	*5*
1			1		
2					
3					
4					
5					

This is the number of paths of length 2 from node 1 to node 3

Adjacency Matrix

- Basically we are doing matrix multiplication
  - What is $\mathtt{adj}^{3}$?
- The value in each entry would represent
  - The number of paths of length 3
  - From node `i` to node `j`
- Consider the meaning of the generalization of $\mathtt{adj}^{n}$

Adjacency Matrix

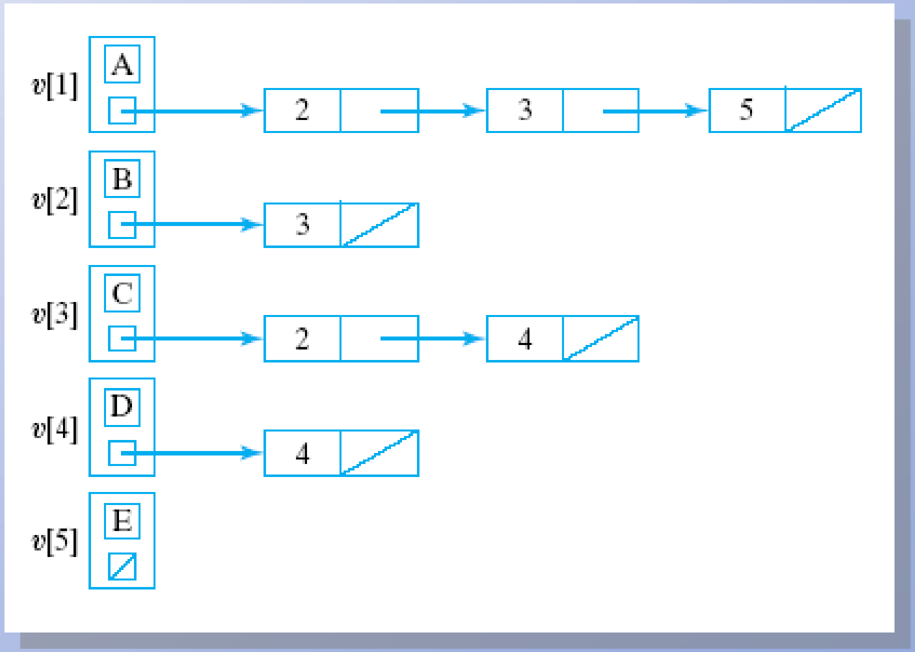
- Deficiencies in adjacency matrix representation
  - Data must be stored in separate matrix

$$\texttt{data} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}.$$

  - When there are few edges the matrix is sparse (wasted space)

Adjacency-List Representation

- Solving problem of wasted space
  - Better to use an array of pointers to linked row-lists
- This is called an Adjacency-list representation
- View source code for class template



Searching a Graph

- Recall that with a tree we search from the root
- But with a digraph …
  - may not be a vertex from which every other vertex can be reached
  - may not be possible to traverse entire digraph (regardless of starting vertex)

Searching a Graph

- We must determine which nodes are reachable from a given node
- Two standard methods of searching:
  - Depth first search
  - Breadth first search

Depth-First Search

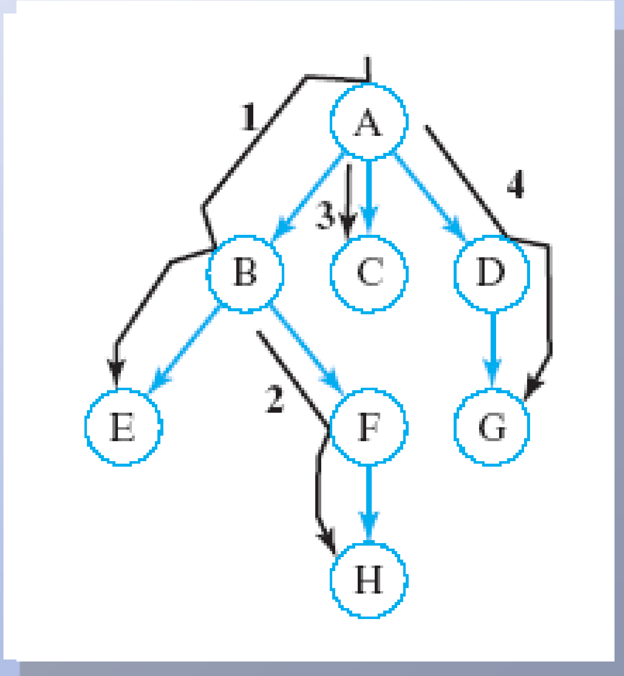
- Start from a given vertex *v*
- Visit first neighbor *w*, of *v*
- Then visit first neighbor of *w* which has not already been visited
- etc. … Continues until
  - all nodes of graph have been examined
- If dead-end reached
  - backup to last visited node
  - examine remaining neighbors

Depth-First Search

- Start from node 1
- What is a sequence of nodes which would be visited in DFS?

Click for answer

A, B, E, F, H, C, D, G



Depth-First Search

- DFS uses <u>backtracking</u> when necessary to return to some values that were
  - already processed or
  - skipped over on an earlier pass
- When tracking this with a stack
  - pop returned item from the stack
- Recursion is also a natural technique for this task
- Note: DFS of a tree would be equivalent to a preorder traversal

Depth-First Search

Algorithm to perform DFS search of digraph

1. Visit the start vertex, ***v***
2. For each vertex ***w*** adjacent to ***v*** do:

   If w has not been visited,

   apply the depth-first search algorithm with ***w*** as the start vertex.

Note the recursion

Breadth-First Search

- Start from a given vertex `v`
- Visit all neighbors of `v`
- Then visit all neighbors of first neighbor `w` of `v`
- Then visit all neighbors of second neighbor *x* of `v` … etc.

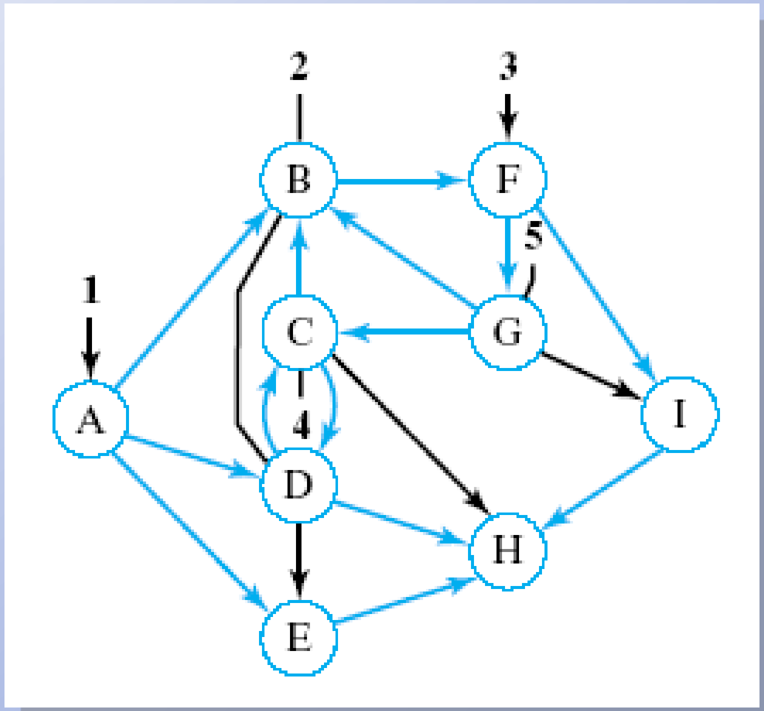
- BFS visits nodes by level

Breadth-First Search

- Start from node containing A
- What is a sequence of nodes which would be visited in BFS?

Click for answer

A, B, D, E, F, C, H, G, I



Breadth-First Search

- While visiting each node on a given level
  - store it so that
  - we can return to it after completing this level
  - so that nodes adjacent to it can be visited
- First node visited on given level should be First node to which we return

What data structure does this imply?

A queue

Breadth-First Search

Algorithm for BFS search of a diagraph

1. Visit the start vertex
2. Initialize queue to contain only the start vertex
3. While queue not empty do
   a. Remove a vertex ***v*** from the queue
   b. For all vertices ***w*** adjacent to ***v*** do:
      If ***w*** has not been visited then:
      i. Visit ***w***
      ii. Add ***w*** to queue

   End while

Graph Traversal

Algorithm to traverse digraph must:

- visit each vertex exactly once
- BFS or DFS forms basis of traversal
- Mark vertices when they have been visited

1. Initialize an array (vector) **`unvisited`** **`unvisited[i]`** false for each vertex **`i`**
2. While some element of **`unvisited`** is false
   a. Select an unvisited vertex **`v`**
   b. Use BFS or DFS to visit all vertices reachable from **`v`**

   End while

Paths

- Routing problems – find an optimal path in a network
  - a shortest path in a graph/digraph
  - a cheapest path in a weighted graph/digraph
- Example – a directed graph that models an airline network
  - vertices represent cities
  - direct arcs represent flights connecting cities
- Task: find most direct route (least flights)

Paths

- Most direct route equivalent to
  - finding length of shortest path
  - finding minimum number of arcs from start vertex to destination vertex
- Search algorithm for this shortest path
  - an easy modification of the breadth-first search algorithm

Shortest Path Algorithm

1. Visit `start` and label it with a `0`
2. Initialize `distance` to `0`
3. Initialize a queue to contain only `start`
4. While `destination` not visited and the queue not empty do:
   a. Remove a vertex `v` from the queue
   b. If label of `v > distance`, set `distance++`
   c. For each vertex `w` adjacent to `v`
      If ***w*** has not been visited then
      i. Visit `w` and label it with `distance + 1`
      ii. Add `w` to the queue
      End for
   End while

Shortest Path Algorithm

5. If `destination` has not been visited then display "Destination not reachable from start vertex"

else

Find vertices `p[0]` … `p[distance]` on shortest path as follows

a. Initialize `p[distance]` to `destination`

b. For each value of ***k*** ranging from `distance – 1` down to `0`
Find a vertex `p[k]` adjacent to `p[k+1]` with label `k`

End for

- Note source code of Digraph Class Template, Fig. 16.1
- View program to find shortest paths in a network, Fig. 16.2

NP-Complete Problems

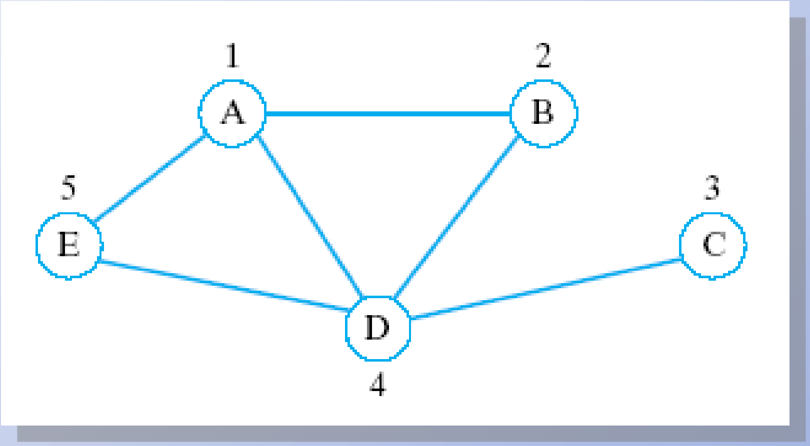
- Nondeterministic polynomial problems
  - Problems for which a solution can be guessed, then checked with an algorithm
  - Algorithm has computing time O(P(n)) for some polynomial P(n)
- Contrast deterministic polynomial (or P) problems
  - Can be solved by algorithms in polynomial time

NP-Complete Problems

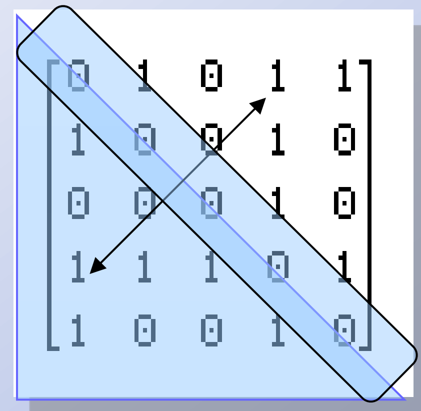
- These are applied to shortest path problems
  - Example is traveling salesman problem
  - Find route to all destinations with least cost
- NP-Complete problems
  - If a polynomial time algorithm that solves any one of these problems can be found
  - Then the existance of polynomial time algorithms for <u>all</u> NP problems is guaranteed

Graphs

- Like a digraph
  - Except no direction is associated with the edges
  - No edges joining a vertex to itself allowed



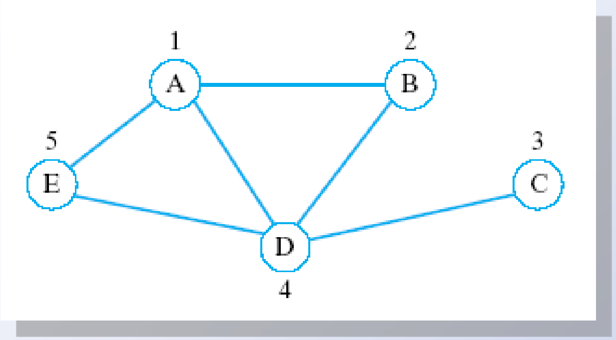
Undirected Graph Representation



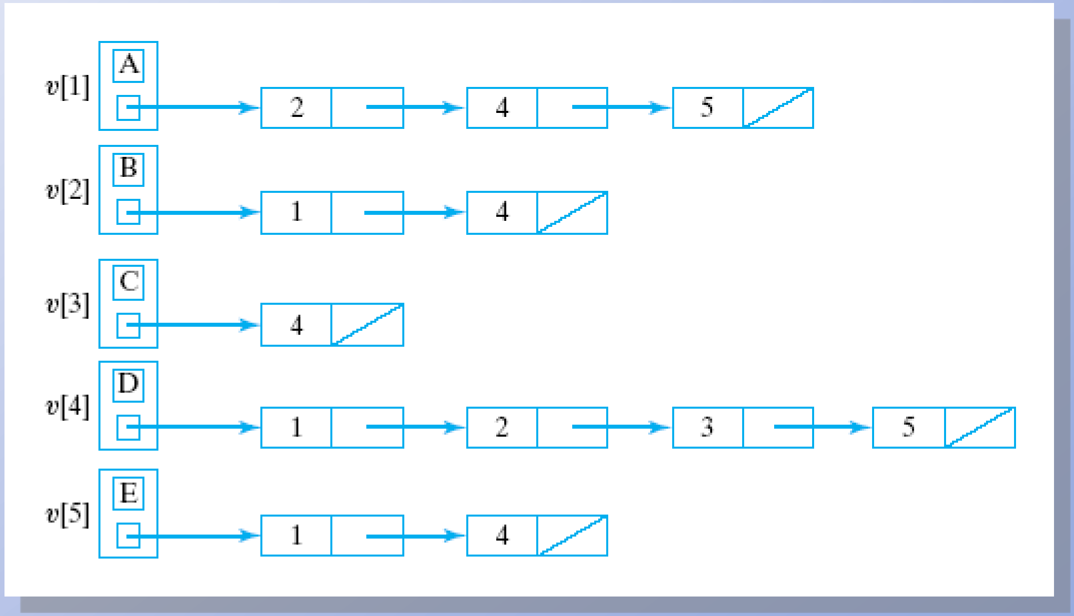
- Can be represented by
  - Adjacency matrices
- Adjacency matrix will always be <u>symmetric</u>
  - For an edge from `i` to `j`, there must be
  - An edge from `j` to `i`
  - Hence the entries on one side of the matrix diagonal are redundant
- Since no loops,
  - the diagonal will be all 0's

Undirected Graph Representation

- Given



- Adjacency-List representation

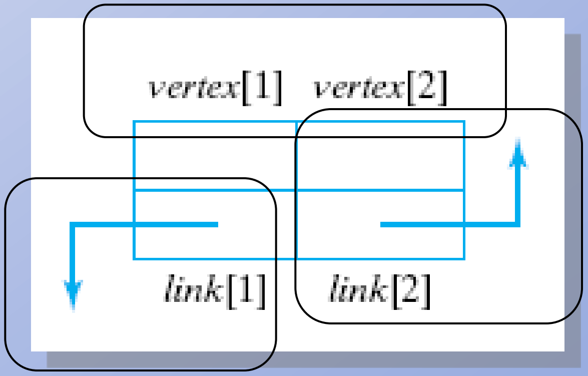


Edge Lists

- Adjacency lists suffer from the same redundancy
  - undirected edge is repeated twice
- More efficient solution
  - use <u>edge lists</u>
- Consists of a linkage of edge nodes
  - one for each edge
  - to the two vertices that serve as the endpoints

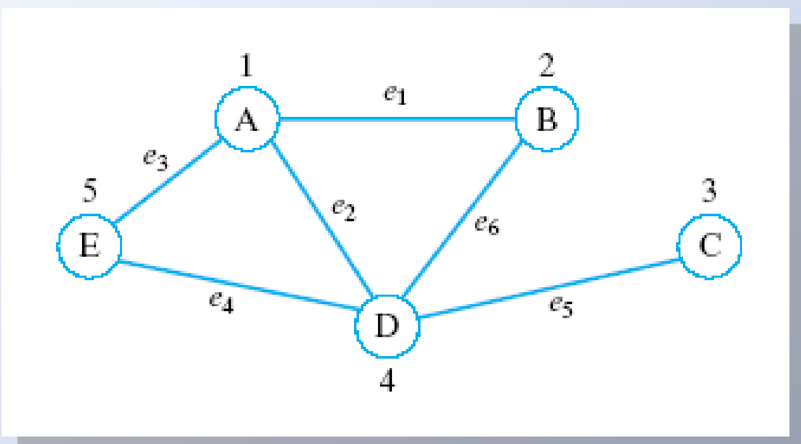
Edge Nodes

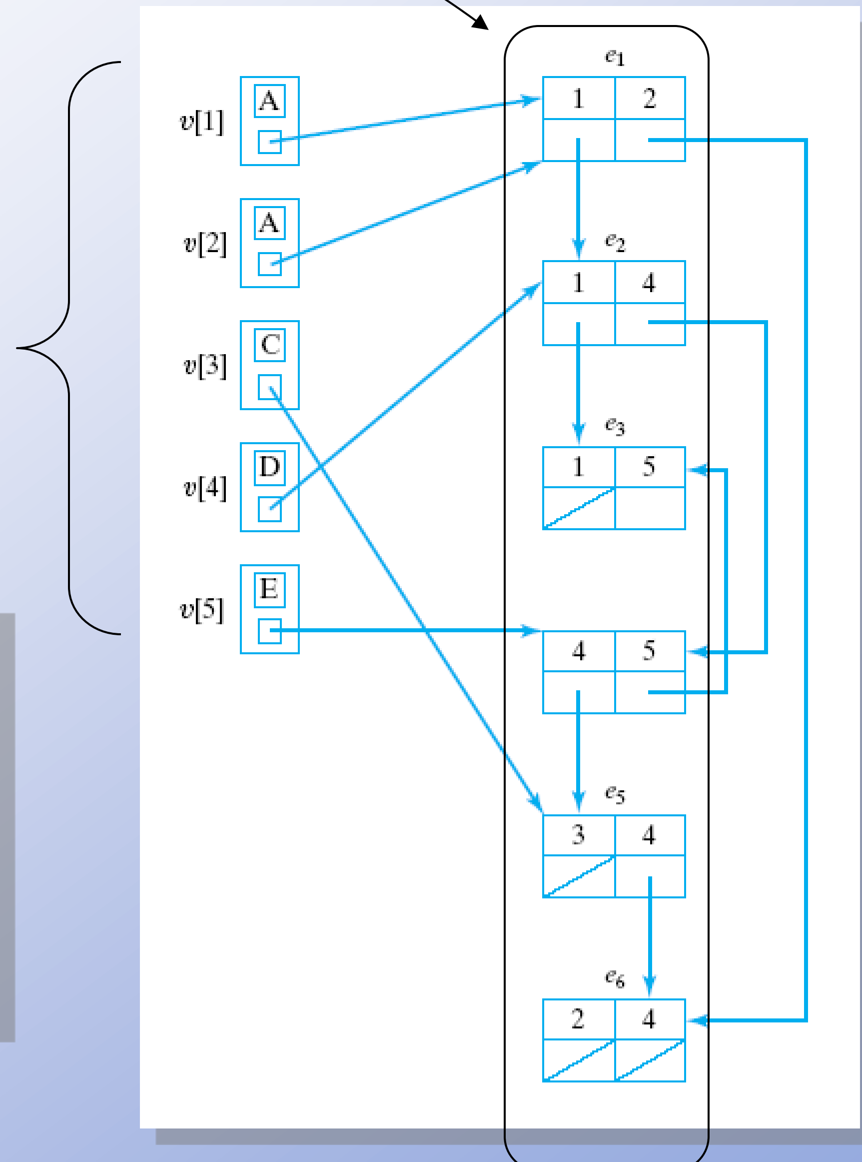
- Each edge node represents one edge
  - `vertex[1]` and `vertex[2]` are vertices connected by this edge
  - `link[1]` points to another edge node having `vertex[1]` as one end point
  - `link[2]` points to another edge node having `vertex[2]` as an endpoint



Edge List

- Vertices have pointers to one edge





Graph Operations

- DFS, BFS, traversal, etc. are similar as those for digraphs
- Note class template Graph, Fig. 16.4
  - Uses edge-list representation of graphs as just described

Connectedness

- Connected defined
  - A path exists from each vertex to every other vertex
- Note the `isConnected()` function in `Graph` class template
  - Uses a DFS, marks all vertices reachable from vertex 1
- View program in Fig. 16-5
  - Exercises this function