Graph Algorithms

Fall 2020

School of Software Engineering

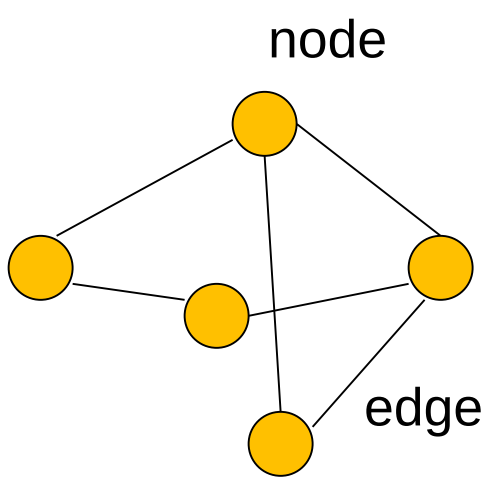
South China University of Technology

Definitions & Representations

Section 9.1

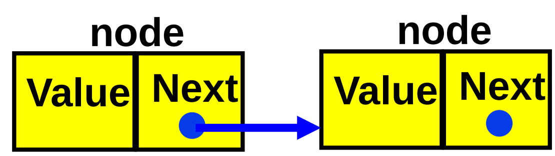
Graphs

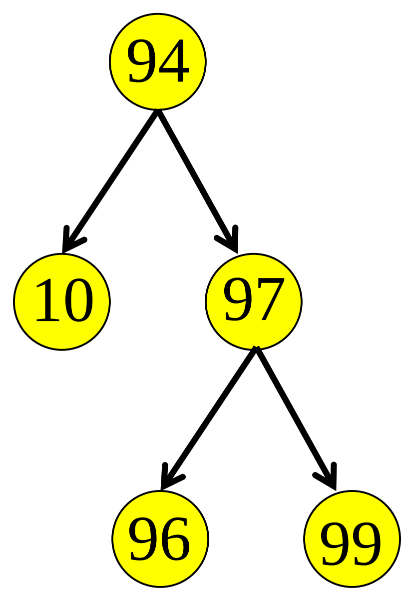
- Graphs are composed of
  - Nodes (vertices)
    - Labeled or unlabeled
  - Edges (arcs)
    - Directed or undirected
    - Labeled or unlabeled

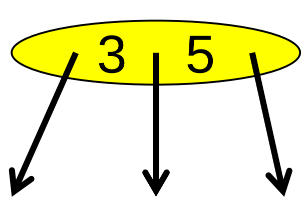


Motivation for Graphs

- Consider the data structures we have looked at so far…
- Linked list: nodes with 1 incoming edge + 1 outgoing edge
- Binary trees/heaps: nodes with 1 incoming edge + 2 outgoing edges
- B-trees: nodes with 1 incoming edge + multiple outgoing edges



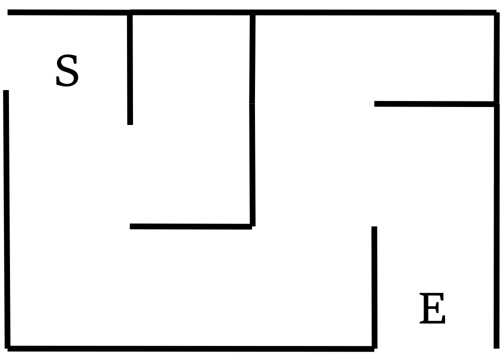


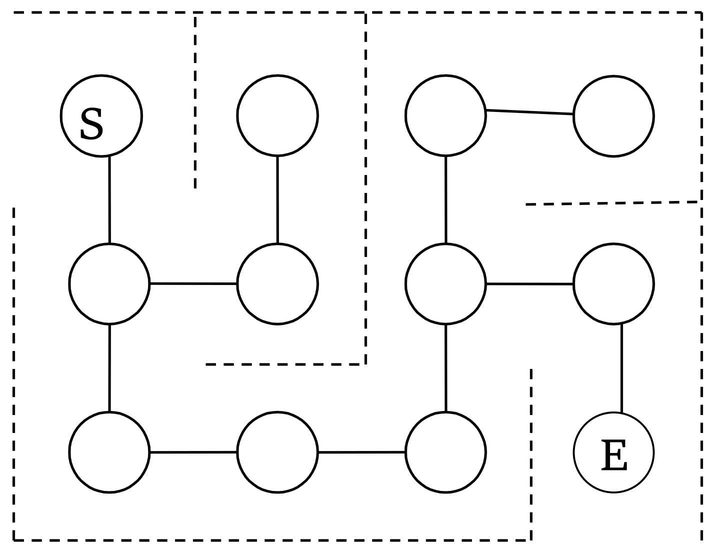


Motivation for Graphs

- How can you generalize these data structures?
- Consider data structures for representing the following problems…

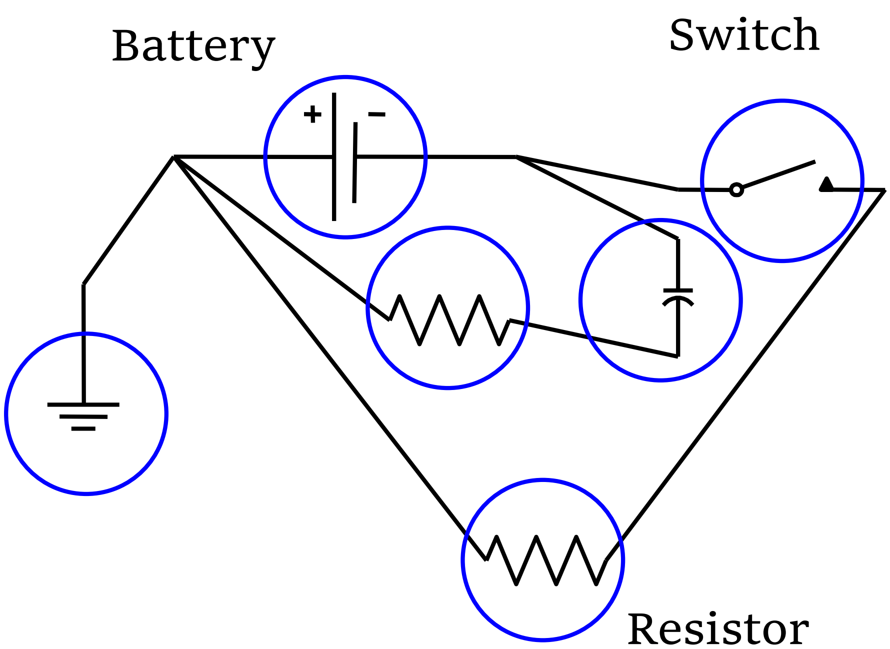
Representing a Maze





Nodes = rooms
Edge = door or passage

Representing Electrical Circuits

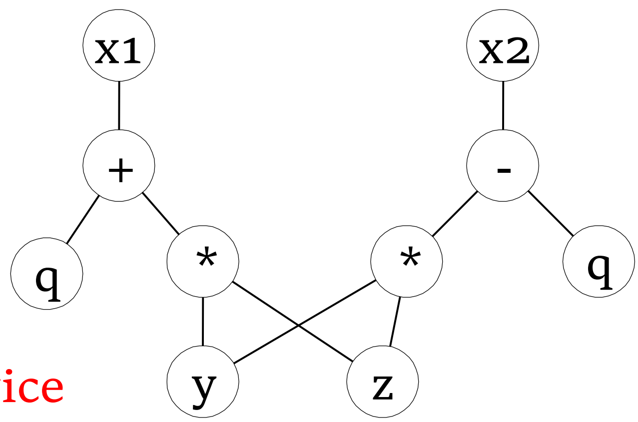


Nodes = battery, switch, resistor, etc.
Edges = connections

Program statements

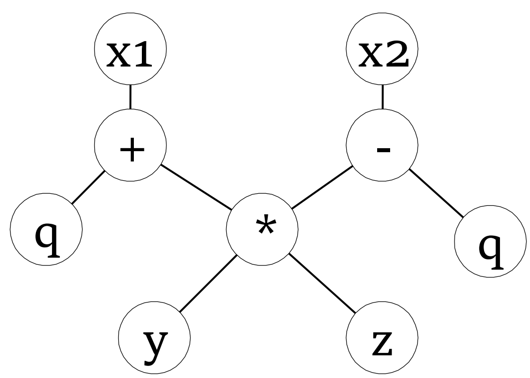
x1=q+y*z
x2=y*z-q

Naive:



y*z calculated twice

common
subexpression
eliminated:

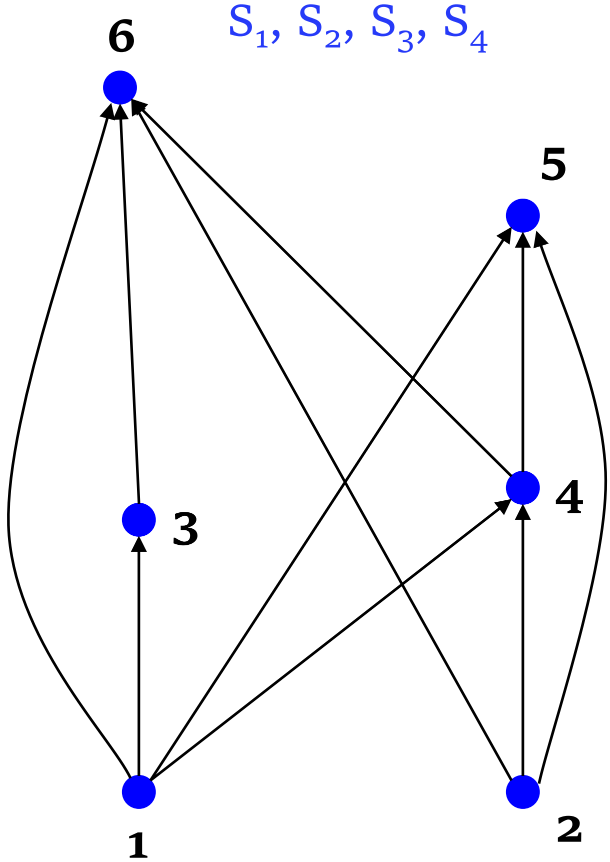


Nodes = symbols/operators
Edges = relationships

Precedence

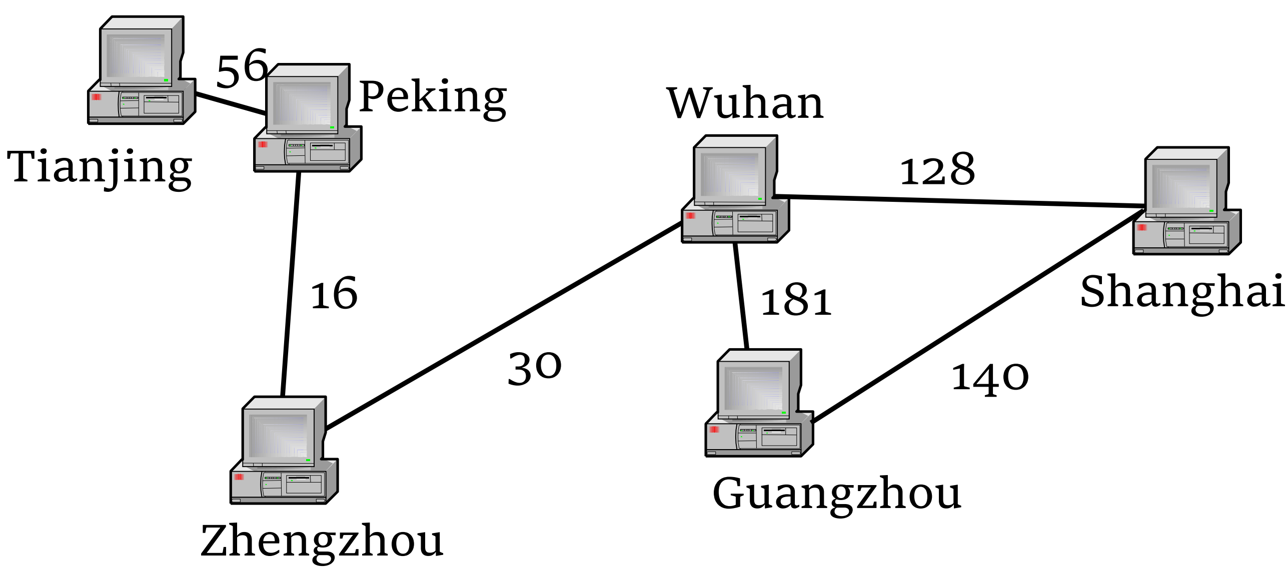
S_1 a=0;
S_2 b=1;
S_3 c=a+1
S_4 d=b+a;
S_5 e=d+1;
S_6 e=c+d;

Which statements must execute before S_6?
S_1, S_2, S_3, S_4



Nodes = statements
Edges = precedence requirements

Information Transmission in a Computer Network



Nodes = computers
Edges = transmission rates

Map

Nodes = stations

Edges = connecting lines

What is the shortest path from “越秀公园” to “动物园”？

Graph Definition

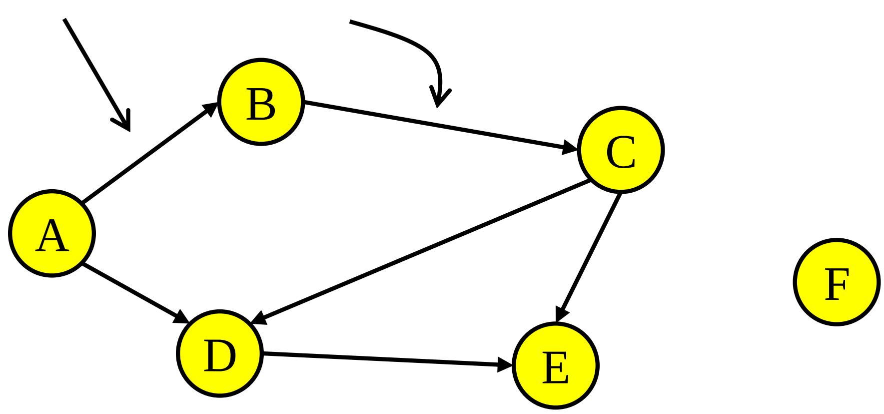
- A graph is simply a collection of nodes plus edges
  - Linked lists, trees, and heaps are all special cases of graphs
- The nodes are known as vertices (node = “vertex”)
- Formal Definition: A graph G is a pair (V, E) where
  - V is a set of vertices or nodes
  - E is a set of edges that connect vertices

- $|E|$ can range from 0 to $|V|^2 - |V|$

Graph Example

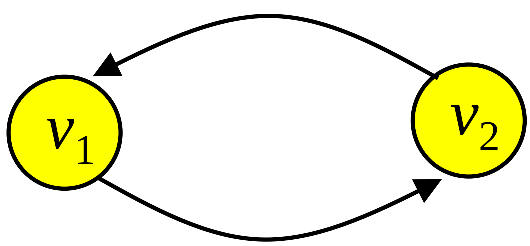
- Here is a directed graph $G = (V, E)$
  - Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V
  - V = {A, B, C, D, E, F}

    E = {(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)}

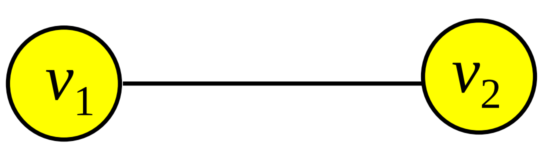


Directed vs Undirected Graphs

- If the order of edge pairs (v_1, v_2) matters, the graph is directed (also called a digraph): $(v_1, v_2) \neq (v_2, v_1)$



- If the order of edge pairs (v_1, v_2) does not matter, the graph is called an undirected graph: in this case, $(v_1, v_2) = (v_2, v_1)$

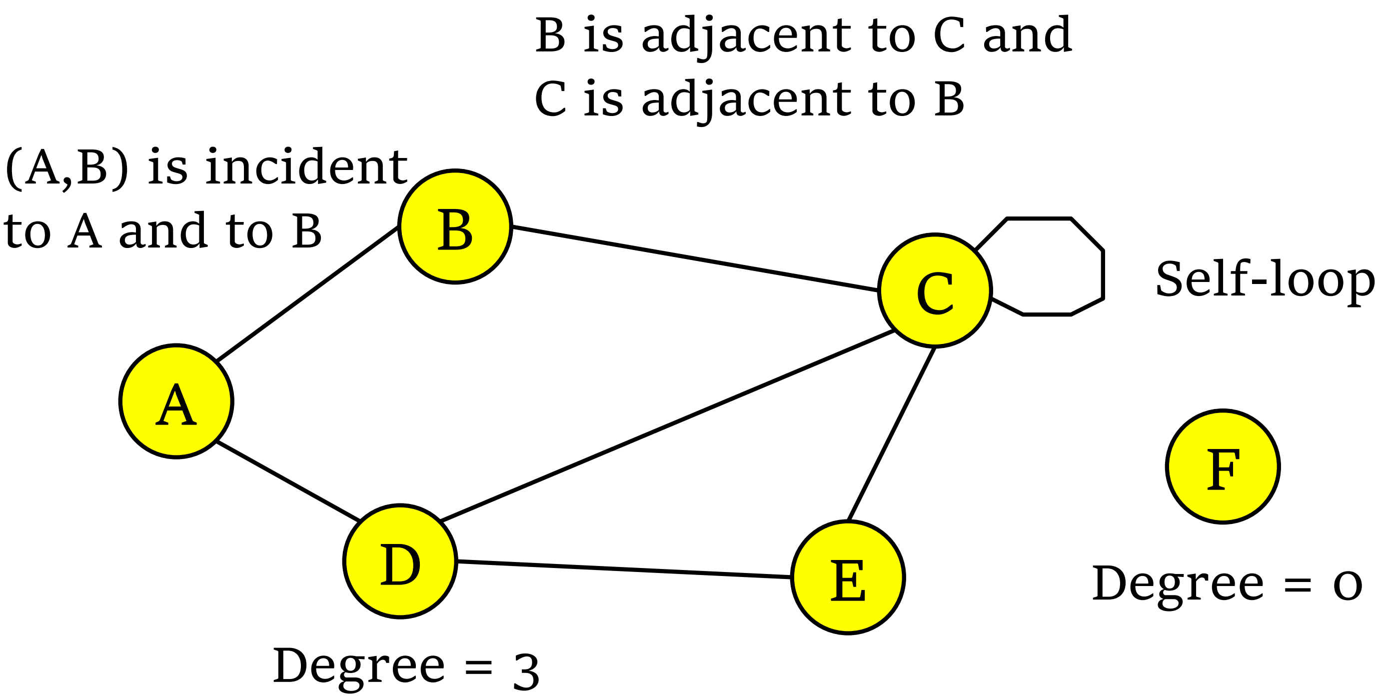


Undirected Terminology

- Two vertices u and v are adjacent in an undirected graph G if {u,v} is an edge in G
  - edge e = {u,v} is incident with vertex u and vertex v

- The degree of a vertex in an undirected graph is the number of edges incident with it
  - a self-loop counts twice (both ends count)
  - denoted with deg(v)

Undirected Terminology

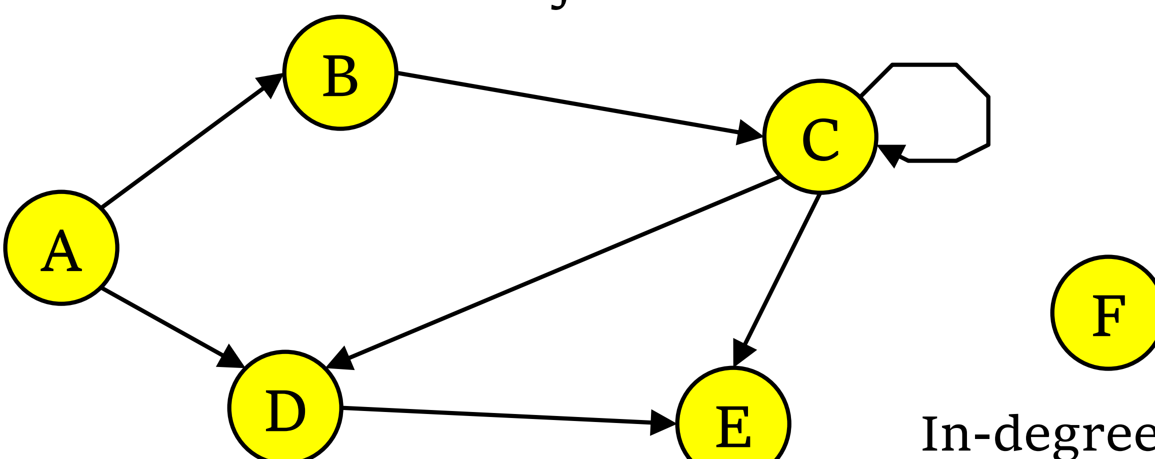


Directed Terminology

- Vertex u is adjacent to vertex v in a directed graph G if (u,v) is an edge in G
  - vertex u is the initial vertex of (u,v)
- Vertex v is adjacent from vertex u
  - vertex v is the terminal (or end) vertex of (u,v)
- Degree
  - in-degree is the number of edges with the vertex as the terminal vertex
  - out-degree is the number of edges with the vertex as the initial vertex

Directed Terminology

B adjacent to C and
C adjacent from B



In-degree = 2
Out-degree = 1

In-degree = 0
Out-degree = 0

Handshaking Theorem

- Let G=(V,E) be an undirected graph with |E|=e edges. Then

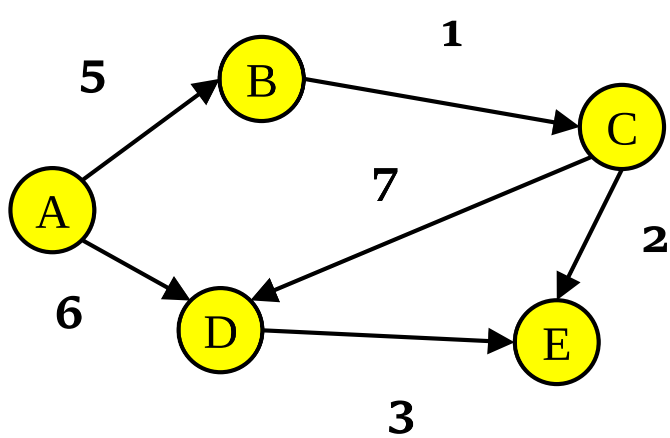
$$\sum_{v \in V} \deg(v) = 2e$$

  - Add up the degrees of all vertices

- Every edge contributes +1 to the degree of each of the two vertices it is incident with
  - number of edges is exactly half the sum of deg(v)
  - the sum of the deg(v) values must be even

Labeled Graph

- Each edge in a graph may be associated with a weight(cost). Such graph is called a weighted graph



Labeled Graph

Paths and Cycles

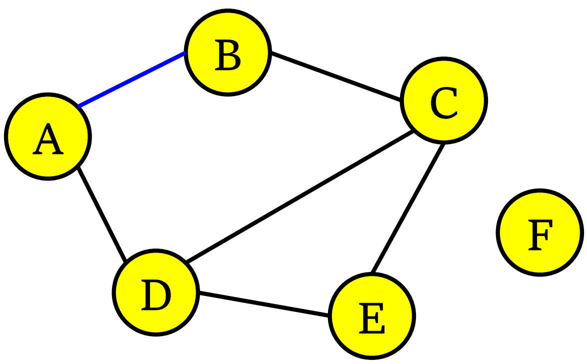
- Given a graph $G = (V,E)$, a path is a sequence of vertices $v_1,v_2, \ldots,v_k$ such that:
  - (v_i,v_{i+1}) in E for $1 \leq i < k$
  - path length = number of edges in the path
  - path weight = sum of weights of each edge
- A path is simple if all vertices on the path are distinct
- A path is a cycle if :
  - $k > 1$; $v_1 = v_k$
- G is acyclic if it has no cycles
  - A directed graph without cycles is called directed acyclic graph (DAG)

Graph Representations

- Space and time are analyzed in terms of:
  - Number of vertices = |V| and
  - Number of edges = |E|

- There are at least two ways of representing graphs:
  - The adjacency matrix representation
  - The adjacency list representation

Adjacency Matrix

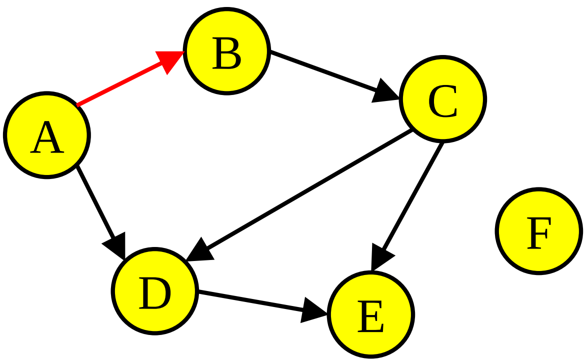


$$\begin{array}{c c}
 & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}
\end{array}$$

$$M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

Space = $|V|^2$

Adjacency Matrix for a Digraph



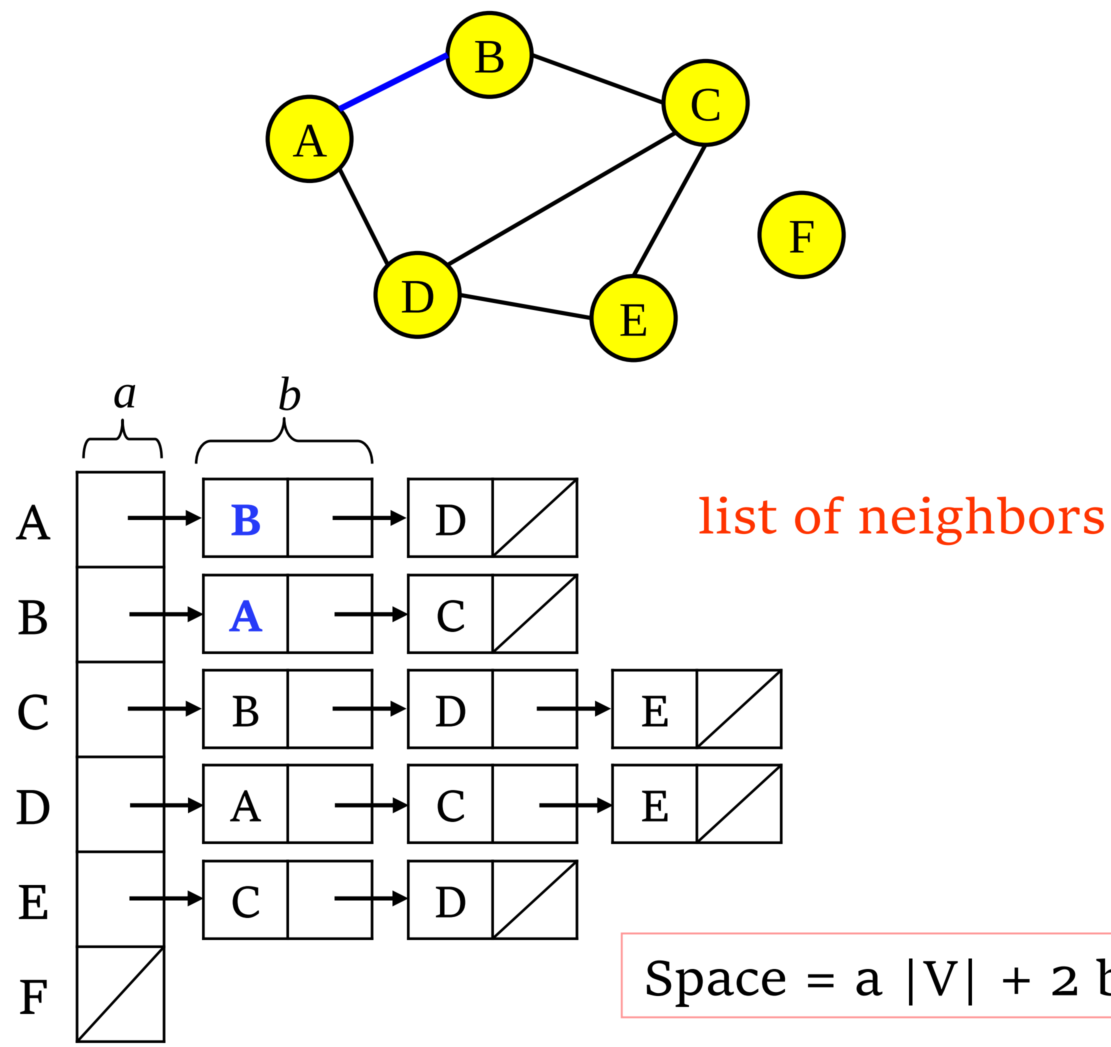
$$
\begin{array}{c|cccccc}
 & A & B & C & D & E & F \\
\hline
A & 0 & 1 & 0 & 1 & 0 & 0 \\
B & 0 & 0 & 1 & 0 & 0 & 0 \\
C & 0 & 0 & 0 & 1 & 1 & 0 \\
D & 0 & 0 & 0 & 0 & 1 & 0 \\
E & 0 & 0 & 0 & 0 & 0 & 0 \\
F & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

$$M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

Space = $|V|^2$

Adjacency List

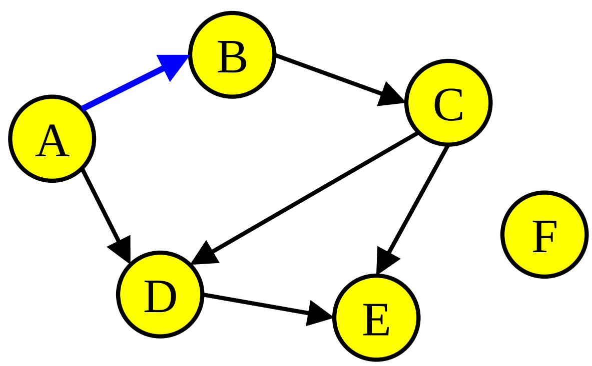
For each v in V, $L(v)$ = list of w such that (v, w) is in E

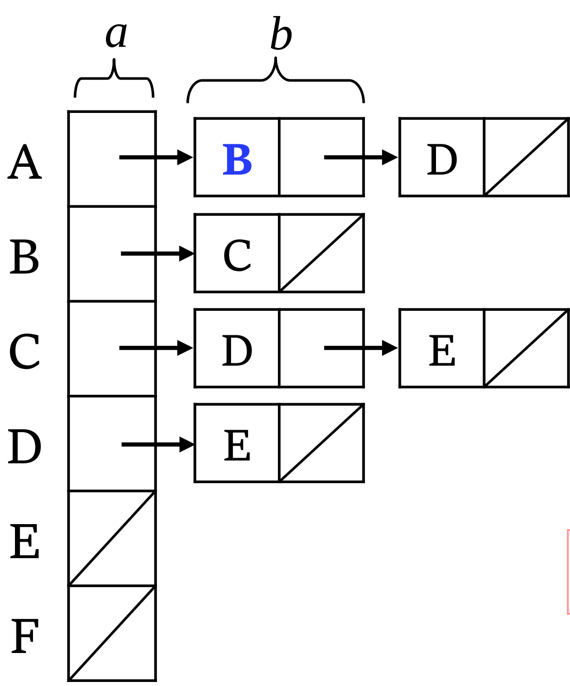


Space = a |V| + 2 b |E|

Adjacency List for a Digraph

For each *v* in *V*, *L*(*v*) = list of *w* such that (*v*, *w*) is in *E*





list of neighbors

Space = a |V| + b |E|

Graph Representations

- Space requirement: Undirected graphs vs. directed graphs
  - The adjacency matrix is symmetric
  - The size of the adjacency list is roughly twice the size of the adjacency list for the corresponding directed graph

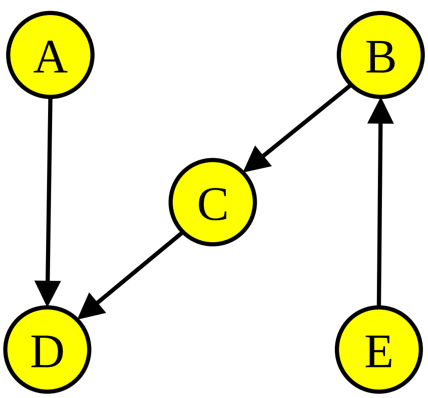
Graph Representations

- Space requirement：adjacency matrix vs. adjacency list
  - The adjacency matrix $\Theta(|V|^2)$ vs. the adjacency list $\Theta(|V|+|E|)$
  - Is the adjacency list more space efficient?

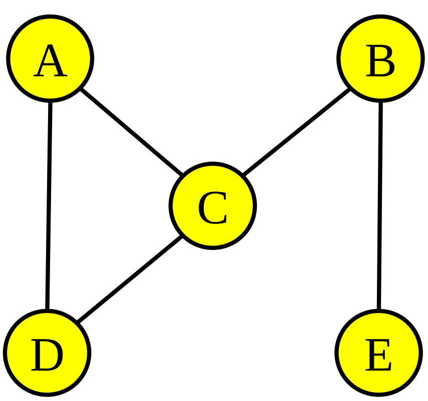
- It depends on the number of edges in the graph
  - The adjacency matrix requires no overhead for pointers
  - As the graph becomes denser ($|E| = (\Theta|V|^2)$), the adjacency matrix becomes relatively more efficient
  - It is more efficient to use the adjacency list to represent sparse graphs

Graph Representations

- Example: assume that a vertex index requires 2 bytes, a pointer requires 4 bytes, and an edge weight requires 2 bytes.



- The adjacency matrix requires $2|V|^2 = 50$ bytes
- The adjacency list requires $4|V|+6|E|= 44$ bytes



- The adjacency matrix requires $2|V|^2 = 50$ bytes
- The adjacency list requires $4|V|+6|E*2|= 92$ bytes

Graph implementations

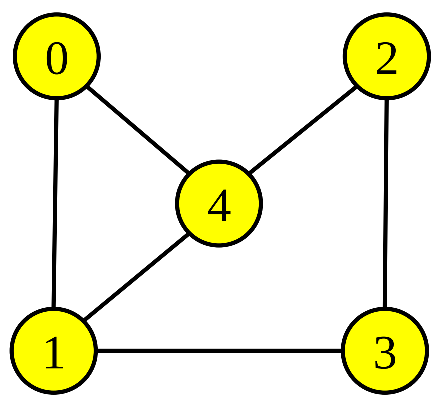
```
//A general-purpose graph abstract class

class Graph {
public:
  //initialize a graph with n vertices
  virtual void Init(int n) =0;

  // return #vertices
  virtual void int n() =0;
  // return #edges
  virtual int e() =0;

  // Return v's first and next neighbor
  virtual int first(int v) =0;
  virtual next(int v, int w) =0;

 // Set or return the weight for an edge (v1, v2)
  virtual void setEdge(int v1, int v2, int wgt) =0;
  virtual int weight(int v1, int v2) =0;

  // Delete the edge (v1, v2)
  virtual void delEdge(int v1, int v2) =0;

  //determine if an edge (v1, v2) is in the graph
  virtual bool isEdge(int v1, int v2) =0;

  //Get and set the mark value for a vertex
  virtual int getMark(int v) =0;
  virtual void setMark(int v, int val) =0;
};
```

Graph Implementations

- Use functions first() and next() to visit the neighbors of a vertex v

```
for(w=G->first(v); w<G->n(); w=G->next(v,w))
```

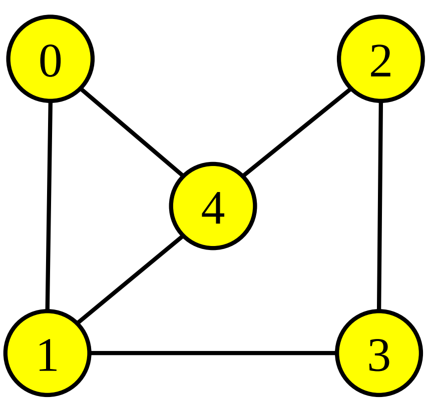
- If v = 1, w = 0, 3, 4



Graph Implementations

- For adjacency matrix implementation
  - Function first() locates the first edge of vertex i by beginning with edge (i, 0) and scanning through row i until an edge is found

  - Function next() locates the edge following edge (i, j) by continuing down the row i starting at position j+1.



$$
\begin{array}{c c}
 & \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$

Graph Implementations

```
//Adjacency matrix implementation

class Graphm: public Graph {
Private:
  int numVertex, numEdge; //#vertices & #edges
  int **matrix;           //Pointer to adjacency matrix
  int *mark;              //Pointer to mark array
public:
  void Init(int n) { //Initialize the graph
    int i;
    numVertex = n;
    numEdge = 0;

    mark = new int[n];      //Initialize mark array
    for (i=0; i<numVertex; i++)
        mark[i] = UNVISITED;

    matrix = (int**) new int*[numVertex]; //create

    for (i=0; i<numVertex; i++)
       matrix[i] = new int[numVertex];

    for (i=0; i<numVertex; i++) //Initial to 0 weights
      for (int j=0; j<numVertex; j++)
       matrix[i][j] = 0;
  }
```

Graph Implementations

```
//Adjacency matrix implementation (cont.)

int n() { return numVertex; } //#vertices
int e() { return numEdge; }    //#edges

int first(int v) {  //return first neighbor of v
  for (int i=0; i<numVertex; i++)
      if (matrix[v][i] != 0) return i;
  return numVertex;
}

//return v's next neighbor after w
int next(int v, int w){
  for(int i=w+1; i<numVertex; i++)
      if (matrix[v][i] != 0) return i;
  return numVertex;
}
```

Graph Implementations

```
//Adjacency matrix implementation (cont.)

void setEdge(int v1, int v2, int wt) { //edge (v1, v2)
   Assert(wt>0, "Illegal weight value");
   if (matrix[v1][v2] == 0) numEdge++;
   matrix[v1][v2] = wt;
}

void delEdge(int v1, int v2) {  //Delete edge (v1, v2)
   if(matrix[v1][v2] != 0) numEdge--;
   matrix[v1][v2] = 0;
}

bool isEdge(int i, int j) {
   return matrix[i][j] != 0;
}


};
```

Graph Implementations

```
//Adjacency list implementation

//Edge class for adjacency list implementation
Class Edge {
  int vert;   //the vertex pointed to by the edge
  int wt;     //the weight of the edge;

Public:
  Edge() { vert = -1; wt = -1;  }
  Edge(int v, int w) {vert = v, wt = w; }

  int vertex() {return vert; }
  int weight() {return wt; }
}
```

Graph Implementations

```
//Adjacency list implementation (cont)

class Graphl: public Graph{
private:
  List<Edge>** vertex;   //List headers
  int numVertex, numEdge;
  int *mark;

public:
  void Init(int n) {
    int i;
    numVertex = n;
    numEdge = 0;

    mark = new int[n];
    for (i=0; i<numVertex; i++) mark[i] = UNVISITED;

    //create and initialize adjacency list
    vertex = (List<Edge>**) new List<Edge>*[numVertex];
    for (i=0; i<numVertex; i++)
      vertex[i] = new Llist<Edge>();
  }
}
```

Graph Implementations

```
//Adjacency list implementation (cont)

int first(int v) { //return first neighbor of v
  if (vertex[v]->length() == 0) //return V's Llist size
     return numVertex;    //no neighbor
  vertex[v]->moveToStart();
  Edge it = vertex[v]->getValue();
  return it.vertex();
}

int next(int v, int w) { //get v's next neighbor after w
  Edge it;
  if (isEdge(v, w)) {
    if ((vertex[v]->currPos()+1) < vertex[v]->length())
    {
       vertex[v]->next();
      it = vertex[v]->getValue();
      return it.vertex();
    }
  }
  return n();  //no neighbor
}
```