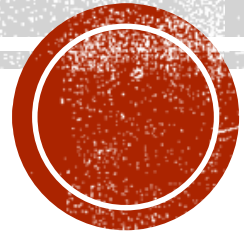
M5: GRAPHS AND GRAPH ALGORITHMS

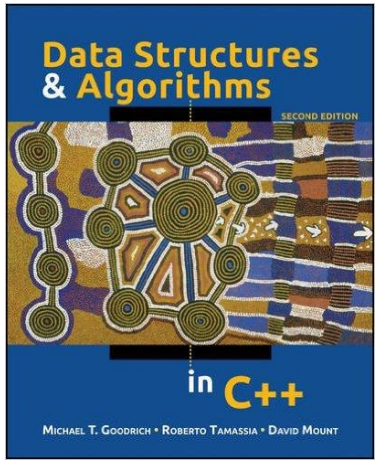
Summer 2019 – CSC 395 – ST: Algorithm & Data Structure Concepts



SOME RESOURCES

- See chapter 12 from https://legacy.gitbook.com/book/cathyatseneca/data-structures-and-algorithms/details
- See chapter 11 from http://people.cs.vt.edu/~shaffer/Book/C++3e20130328.pdf

- Another great source (not free!)
  - **Data Structures and Algorithms in C++** Hardcover – 2004/2011
  - by Michael T. Goodrich, Roberto Tamassia, David Mount



SOME RESOURCES (2)

- visualization:
  - graphs: https://visualgo.net/en/graphds?slide=1
  - liang - graphs: DFS/BFS: http://www.cs.armstrong.edu/liang/animation/web/GraphLearningTool.html

GRAPH - MOTIVATION

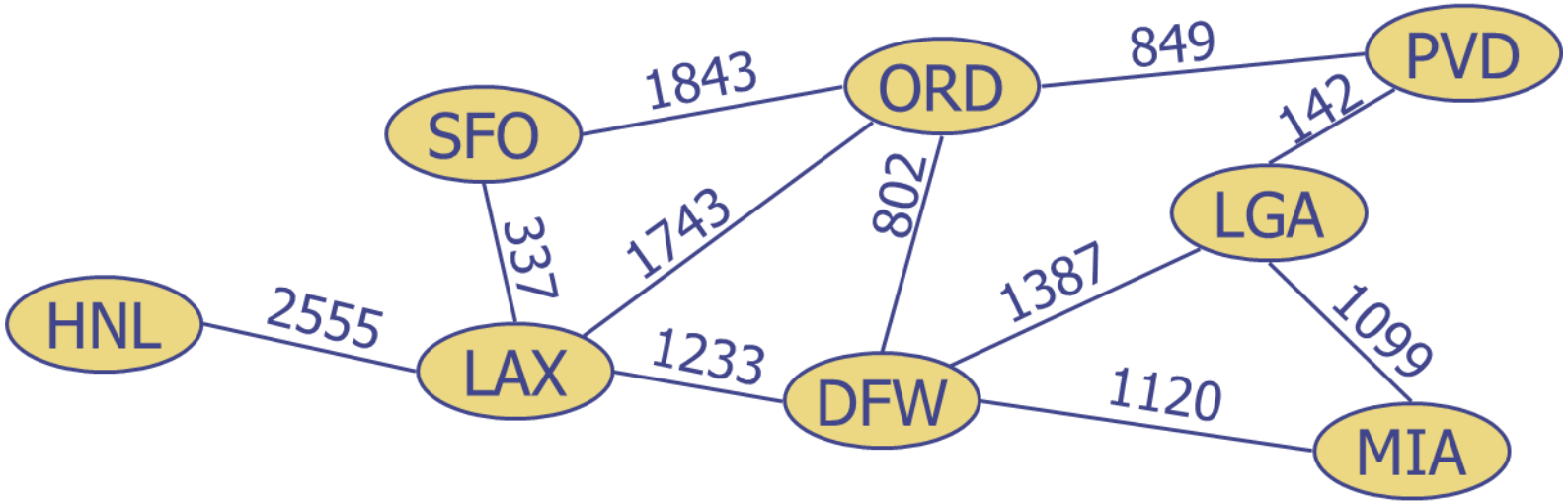
- Graphs provide the ultimate in data structure flexibility. Graphs can model both real-world systems and abstract problems, so they are used in hundreds of applications.

- Here is a small sampling of the range of problems that graphs are routinely applied to:
  1. Modeling connectivity in computer and communications networks.
  2. Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
  3. Modeling flow capacities in transportation networks.
  4. Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
  5. Modeling computer algorithms, showing transitions from one program state to another.
  6. Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
  7. Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

GRAPH - DEFINITION

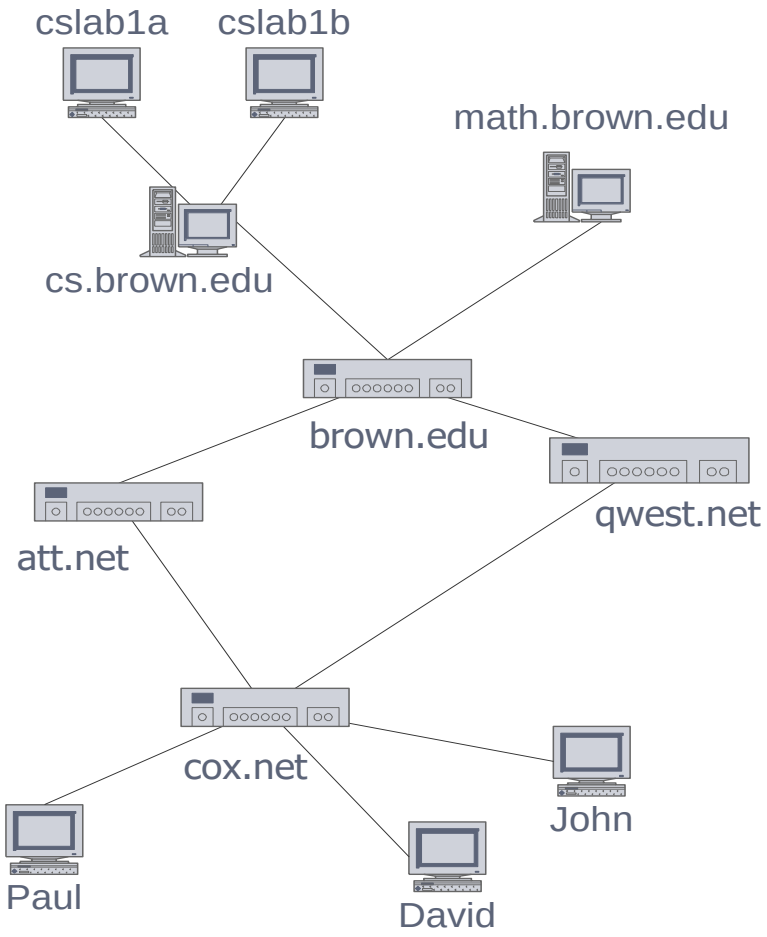
- A graph is a pair G=(V,E) where
  - V is set of nodes called vertices and
  - E is a set of pairs of vertices called edges.

- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



GRAPH - APPLICATIONS

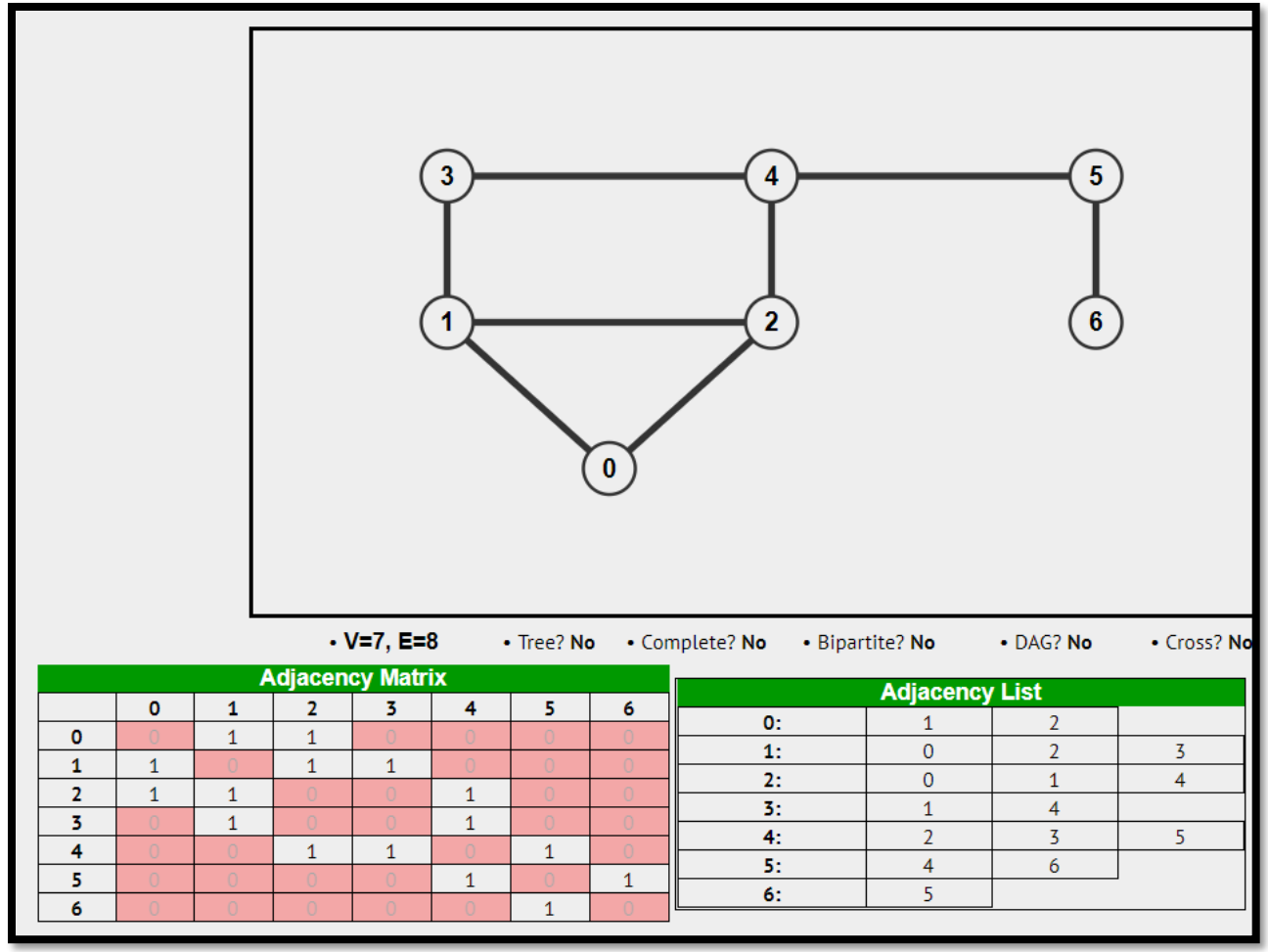
- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



REPRESENTATIONS OF GRAPHS

- There are two standard ways to represent a graph $G = (V, E)$:
  - as a collection of adjacency lists or
    - the adjacency list is an array of linked lists
  - as an adjacency matrix:
    - column j in row i is set to 1 if there is an edge from vi to vj and 0 otherwise.

    - https://visualgo.net/en/graphds



Adjacency Matrix

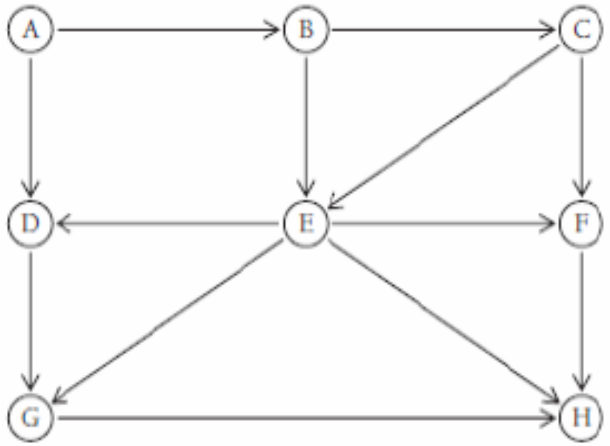
	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

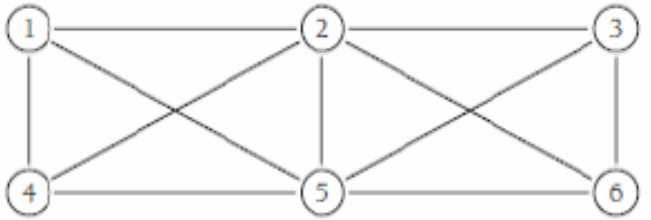
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

REPRESENTATIONS OF GRAPHS (2)

- The adjacency matrix/list is applicable to both directed and undirected graphs.
  - Directed graph (digraph): graph whose pairs are ordered
  - Undirected graph (graph): graph whose pairs are not ordered

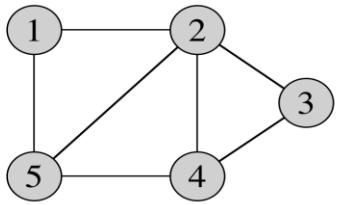


A Digraph (Directed Graph).

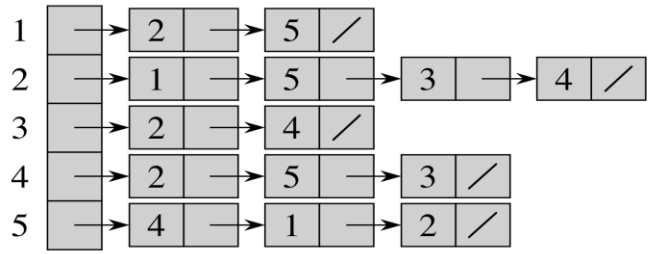


An Unordered Graph.

REPRESENTATIONS OF GRAPHS (3)



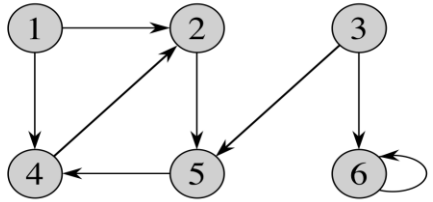
(a)



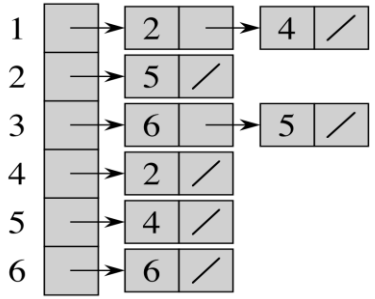
(b)

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

(c)



(a)

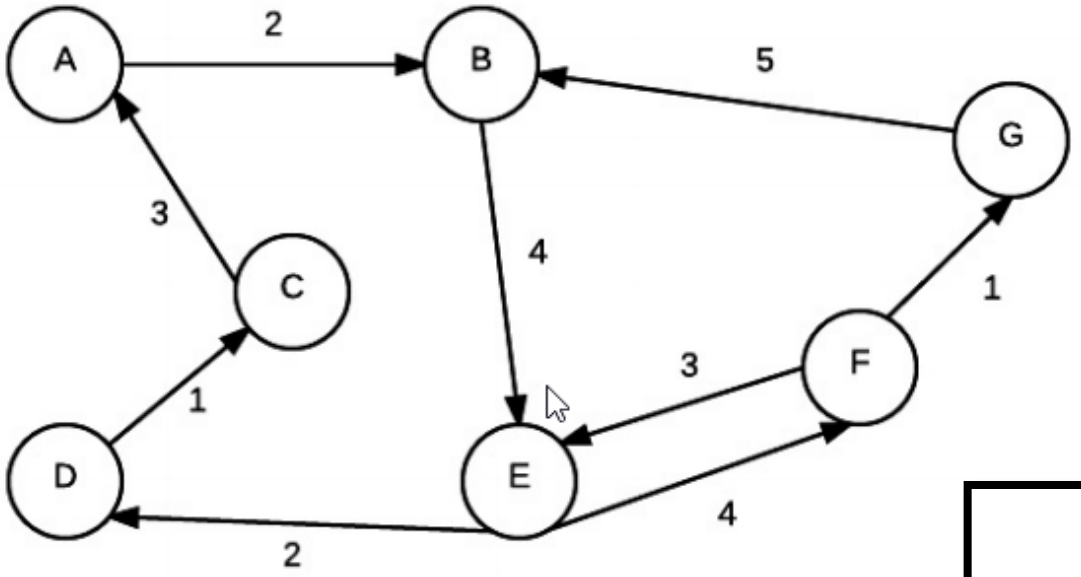


(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

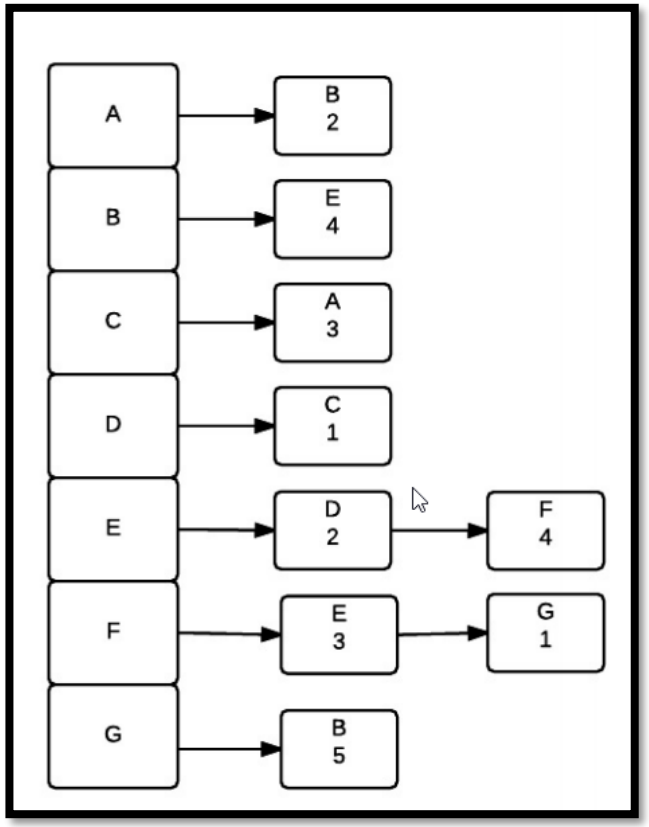
(c)

REPRESENTATIONS OF WEIGHTED GRAPHS (4)



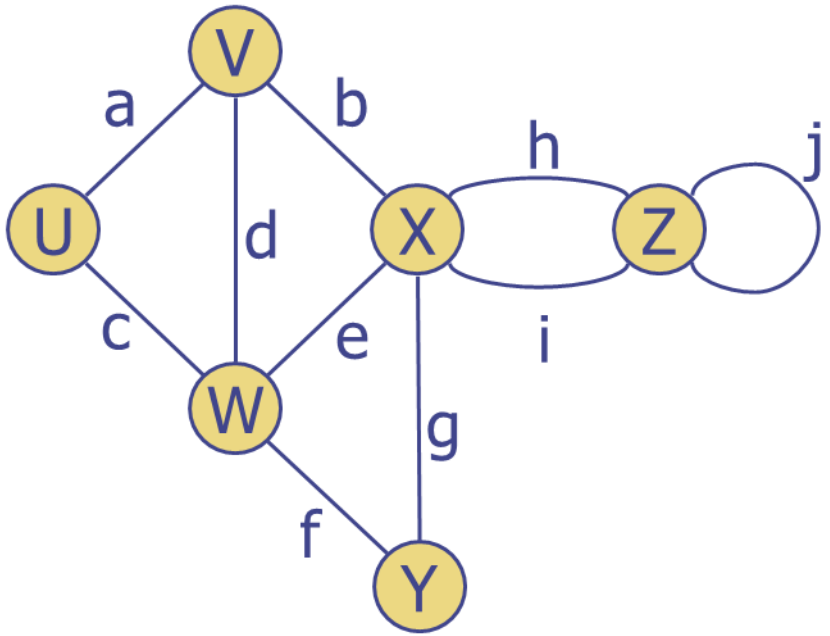
	A-0	B-1	C-2	D-3	E-4	F-5	G-6
A-0	0	2	0	0	0	0	0
B-1	0	0	0	0	4	0	0
C-2	3	0	0	0	0	0	0
D-3	0	0	1	0	0	0	0
E-4	0	0	0	2	0	4	0
F-5	0	0	0	0	3	0	1
G-6	0	5	0	0	0	0	0

Adjacency Matrix



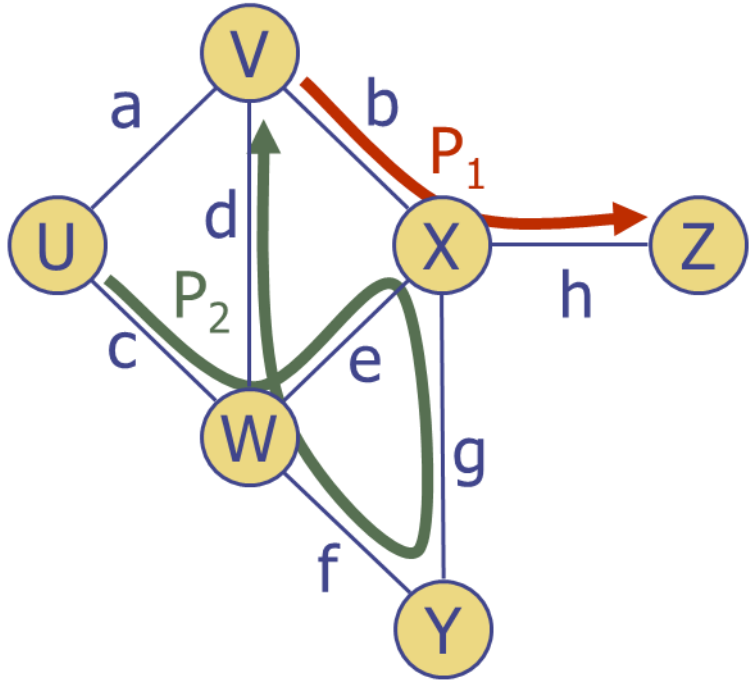
TERMINOLOGY

- **End vertices** (or **endpoints**) of an edge
  - U and V are the endpoints of a
- Edges **incident** on a vertex
  - a, d, and b are incident on V
  - **outgoing** edges of a vertex …
  - **incoming** edges of a vertex …
- **Adjacent vertices**
  - U and V are adjacent ←they are joined by an edge
- **Degree** of a vertex (# of incident edges)
  - X has degree 5 ←the number of edges incident on X is 5
  - **in-degree** …
  - **out-degree** …
- **Parallel** (or **multiple**) **edges**
  - h and i are parallel edges ←they have the same endpoints
- **Self-loop** (the two endpoints coincide)
  - j is a self-loop
- **Simple graph** (no parallel edges or self-loops)



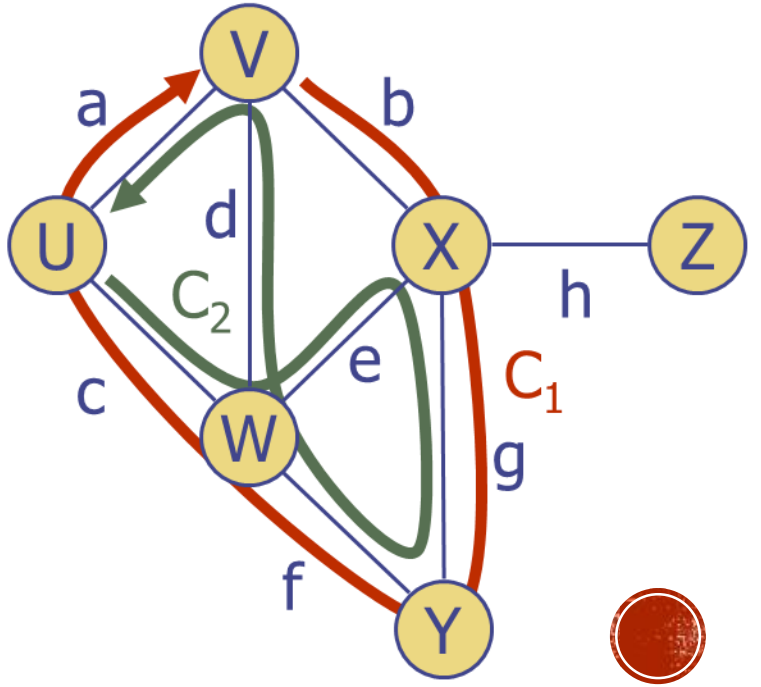
TERMINOLOGY (2)

- **Path** ← the **length of a path** is the number of edges …
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is incident to its predecessor and successor vertex
- **Simple path**
  - a path such that all its vertices are distinct
- **Cycle**
  - a path with at least one edge, with the same start and end vertices.
  - A cycle is simple if all its vertices are distinct
- Examples
  - **P_1=(V,b,X,h,Z)** is a simple path
  - **P_2=(U,c,W,e,X,g,Y,f,W,d,V)** is a path that is not simple
  - Examples of cycles?



TERMINOLOGY (4)

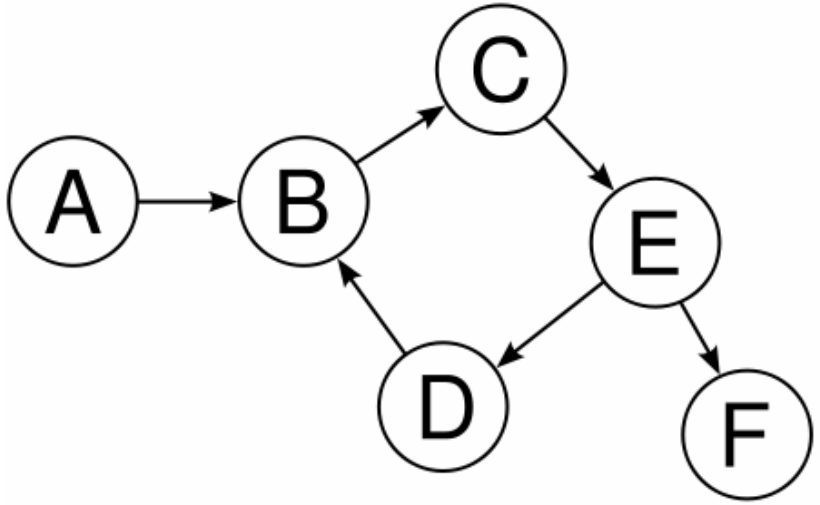
- **Directed path**
  - a path such that all edges are directed and are traversed along their direction.
- **Subgraph** (of a graph G)
  - a graph H whose vertices and edges are subsets of the vertices and edges of G.
- **Connected graph**
  - for any two vertices there is a path between them
- **Complete graph**
  - for any two vertices there is an edge between them
- **Forest**
  - a graph without cycles
- **(Free) Tree**
  - a connected forest
- **Connected components…** maximal connected subgraphs



TERMINOLOGY (5) – EXAMPLE

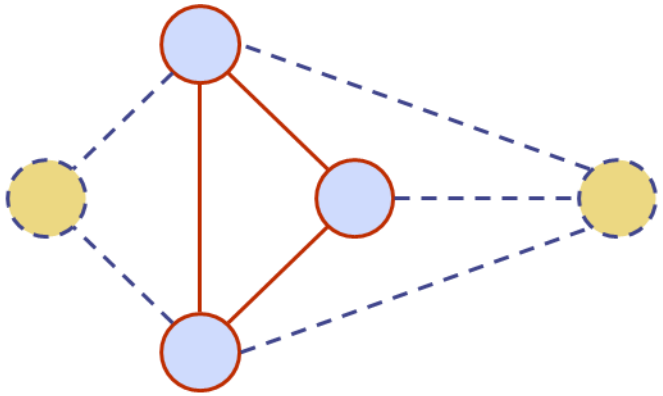
- This is not a complete graph (why?)
  - The path: A->B->C->E->F is a simple path. Its length is 4.
  - There is a cycle on the graph: B->C->E->D->B

- There are two distinct notions of connectivity in a directed graph.
  - A directed graph is weakly connected if there is an undirected path between any pair of vertices, and strongly connected if there is a directed path between every pair of vertices (Skiena 1990, p. 173).



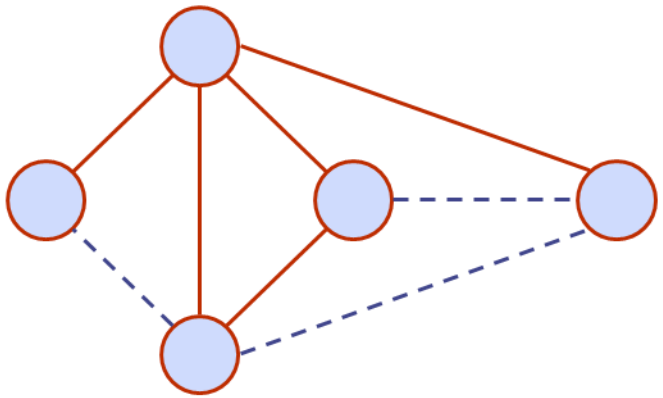
SUBGRAPHS

- A **subgraph** S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G



Subgraph

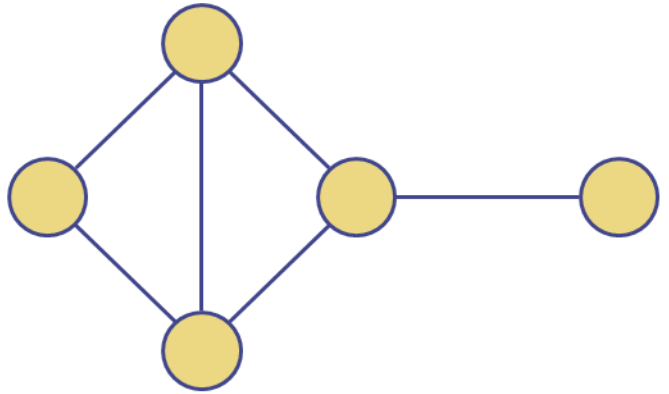
- A **spanning subgraph** of G is a subgraph that contains all the vertices of G



Spanning subgraph

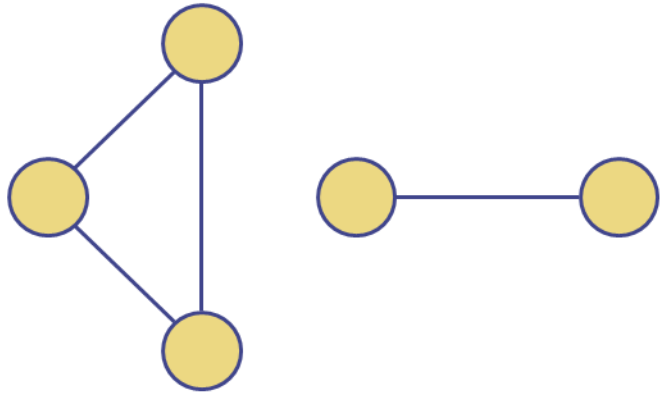
CONNECTIVITY

- A graph is **connected** if there is a path between every pair of vertices



Connected graph

- A **connected component** of a graph G is a maximal connected subgraph of G

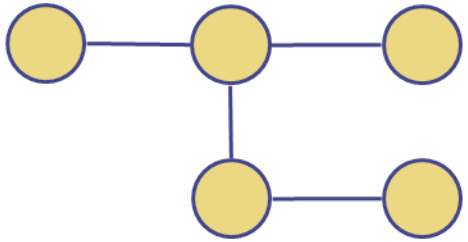


Non connected graph with two connected components

TREES AND FORESTS

- A (free) **tree** is an undirected graph T such that
  - T is connected
  - T has no cycles

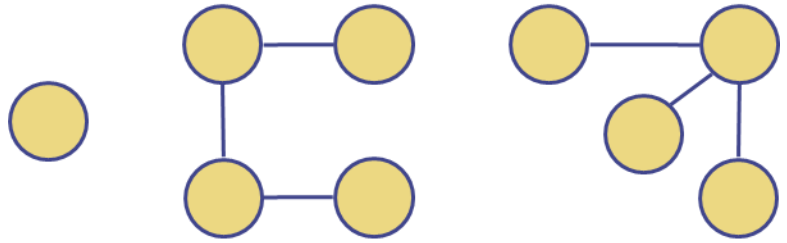
  This definition of tree is different from the one of a rooted tree



Tree

- A **forest** is an undirected graph without cycles

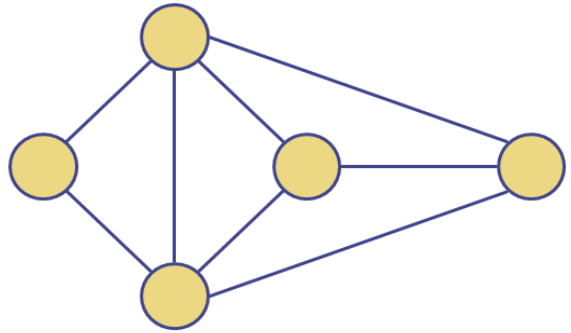
- The connected components of a forest are trees



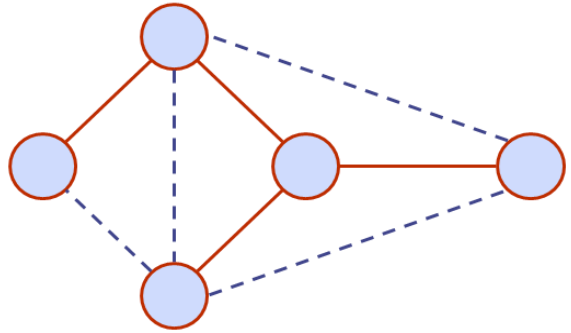
Forest

SPANNING TREES AND FORESTS

- **A spanning tree** of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A **spanning forest** of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

ANOTHER EXAMPLE …

- What does this look like in regards to graphs?
- What are the nodes?
- What do edges represent in here?
- Look at the values in this table (e.g. Boston-Atlanta & Atlanta- Boston): should we use a directed or undirected graph?
- Source: https://www.tripinfo.com/tips/Maps001.html

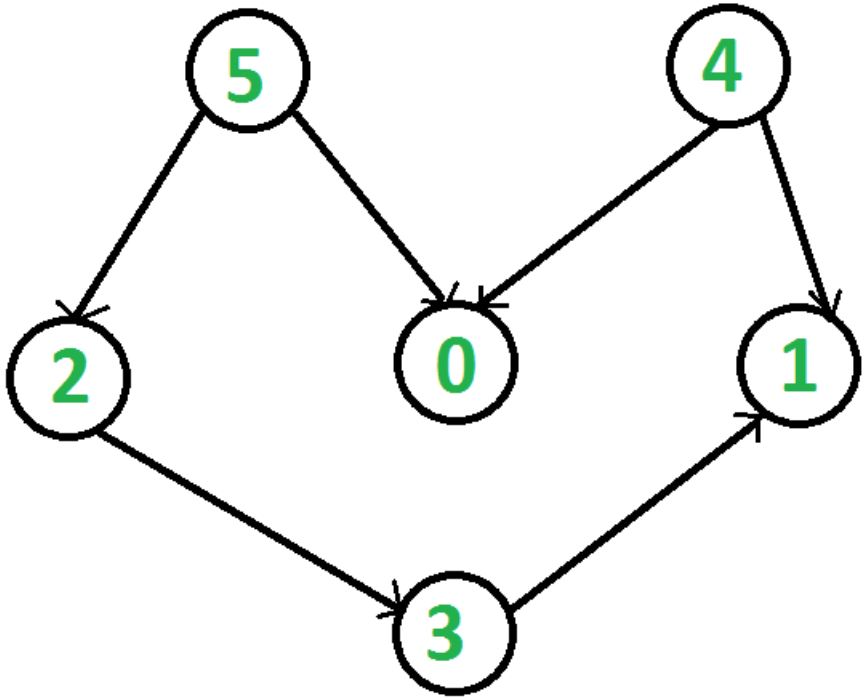
Approximate Mileages in the United States

	Atlanta, GA	Boston, MA	Chicago, IL	Cleveland, OH	Dallas, TX	Denver, CO	Detroit, MI	Houston, TX	Kansas City, MO	Las Vegas, NV	Los Angeles, CA	Miami, FL	Minneapolis, MN	New Orleans, LA	New York, NY	Philadelphia, PA	Phoenix, AZ	San Antonio, TX	San Diego, CA	San Francisco, CA	Seattle, WA	Washington, DC
Albuquerque, NM	1400	2200	1310	1590	640	440	1560	850	780	590	810	1970	1220	1200	2000	1930	460	730	810	1110	1450	1850
Atlanta, GA	-	1075	715	730	830	1520	745	800	820	1980	2185	665	1140	520	865	760	1860	1025	2130	2495	2690	620
Austin, TX	970	1960	1165	1380	195	1060	1390	160	735	1300	1380	1350	1175	510	1740	1660	1010	80	1300	1760	2385	1530
Baltimore, MD	680	405	700	365	1460	1625	525	1555	1070	2400	2725	1150	1110	1210	195	105	2335	1700	2680	2900	2795	40
Boston, MA	1080	-	980	645	1870	2010	710	1965	1445	2750	3130	1550	1400	1625	215	310	2750	2100	3285	3200	3165	445
Charlotte, NC	240	850	740	520	1060	1580	630	1030	970	2200	2410	740	1150	720	620	510	2030	1240	2410	2720	2740	380
Chicago, IL	715	980	-	345	940	1020	280	1075	505	1780	2190	1390	430	940	820	770	1840	1245	2280	2235	2050	700
Cleveland, OH	730	645	345	-	1225	1375	165	1360	800	2090	2490	1365	770	1135	505	425	2105	1490	2625	2565	2535	360
Columbus, OH	560	770	320	145	1120	1240	180	1215	670	2020	2360	1240	740	990	555	475	1945	1355	2260	2530	2500	390
Dallas, TX	830	1870	940	1225	-	800	1995	245	505	1230	1435	1395	940	495	1650	1565	1015	270	1430	1795	2225	1415
Denver, CO	1520	2010	1020	1375	800	-	1305	1040	605	760	1190	2130	875	1295	1855	1770	905	975	1245	1270	1430	1710
Detroit, MI	745	710	280	165	1195	1305	-	1330	755	2020	2450	1395	695	1145	635	590	2075	1460	2545	2495	2460	525
El Paso, TX	1450	2430	1450	1785	624	700	1755	755	945	720	785	2070	1380	1105	2210	2130	405	565	730	1210	1825	2055
Houston, TX	800	1965	1075	1360	245	1040	1330	-	750	1470	1565	1330	1185	360	1745	1650	1210	200	1580	1985	2445	1505
Indianapolis, IN	550	950	180	305	925	1050	275	1010	505	1840	2185	1220	600	835	730	650	1770	1175	2080	2355	2375	575
Kansas City, MO	820	1445	505	800	505	605	755	750	-	1370	1635	1485	435	810	1225	1145	1275	790	1655	1905	1985	1070
Las Vegas, NV	1980	2750	1780	2090	1230	760	2020	1470	1370	-	270	2570	1660	1730	2570	2480	290	1290	340	570	1180	2420
Los Angeles, CA	2185	3130	2190	2490	1435	1190	2540	1565	1635	270	-	2885	2035	1950	2915	2830	380	1390	125	420	1195	2755
Louisville, KY	435	990	310	355	900	1170	365	1005	530	1870	2215	1105	730	740	775	690	1800	1135	2090	2435	2495	640
Memphis, TN	415	1390	535	765	470	1055	755	580	460	1600	1880	1020	835	395	1175	1080	1495	725	1810	2190	2455	975
Miami, FL	665	1550	1390	1365	1395	2130	1395	1310	1485	2570	2885	-	179	885	1325	1235	2420	1435	2885	3240	3470	1100
Milwaukee, WI	815	1070	95	440	1030	1040	365	1150	585	1800	2150	1480	335	1020	935	850	1845	1310	2130	2190	2045	790
Minneapolis, MN	1140	1400	420	770	940	875	695	185	455	1660	2035	1790	-	1230	1265	1180	1670	1190	2085	2080	1695	1120
Nashville, TN	250	1170	445	545	710	1205	555	815	565	1810	2100	925	880	550	950	855	1765	945	2000	2410	2550	710
New Orleans, LA	520	1625	940	1135	495	1295	1145	360	810	1730	1950	885	1230	-	1410	1315	1555	575	1985	2300	2735	1165
New York, NY	865	215	820	505	1650	1855	635	745	1225	2570	2915	1325	1265	1410	-	90	2530	1890	2855	3085	3025	240
Oklahoma City, OK	850	1745	795	1105	210	630	1075	450	340	1110	1350	1575	775	725	1530	1445	1010	495	1330	1695	2155	1370
Omaha, NE	1030	1455	480	820	685	540	745	935	185	1300	1670	1690	375	995	1295	1215	1355	960	1630	1730	1825	1150
Philadelphia, PA	760	310	770	425	1565	1770	590	1650	1145	2480	2830	1235	180	1315	90	-	2445	1795	2780	2960	2945	145
Phoenix, AZ	1860	2750	1840	2105	1015	905	2075	1210	1275	290	380	2430	1670	1555	2530	2445	-	1005	360	810	1600	2370
Portland, OR	2660	3230	2250	2600	2145	1350	2525	2370	1955	1000	1020	3440	1830	2655	3090	3010	1385	2250	1090	535	175	2945
St. Louis, MO	585	1190	300	565	645	860	515	780	255	1620	1945	1295	555	690	970	890	1545	950	1830	2160	2240	920
Salt Lake City, UT	1960	2435	1415	1800	1240	540	1725	1505	1105	420	705	2625	1300	1735	2275	2195	650	1365	760	730	925	2130
San Antonio, TX	1025	2110	1245	1490	270	975	1460	200	790	1290	1390	1435	1190	575	1890	1795	1005	-	1375	1795	2305	1650
San Diego, CA	2230	3285	2280	2625	1430	1245	2545	1580	1655	340	125	2885	2085	1935	2855	2780	360	1375	-	525	1315	2705
San Francisco, CA	2495	3200	2125	2565	1795	1270	2495	1985	1905	570	420	3240	9080	2300	3085	2960	810	1795	525	-	790	2900
Seattle, WA	2690	3165	3185	2535	2225	1430	2460	2445	985	1180	1195	3470	1695	2735	3025	2945	1600	2305	1315	790	-	2880
Washington, DC	620	445	700	360	1415	1710	252	1505	1070	2420	2725	1100	1120	1165	240	145	2370	1650	2705	2900	2880	-

GRAPH CLASS IN C# ...

We want to be able to:

- Create a new Graph
- Add/Remove edges and vertices
- Display those values
- Perform some algorithms (we'll see more details soon).



GRAPH CLASS – VERTICES

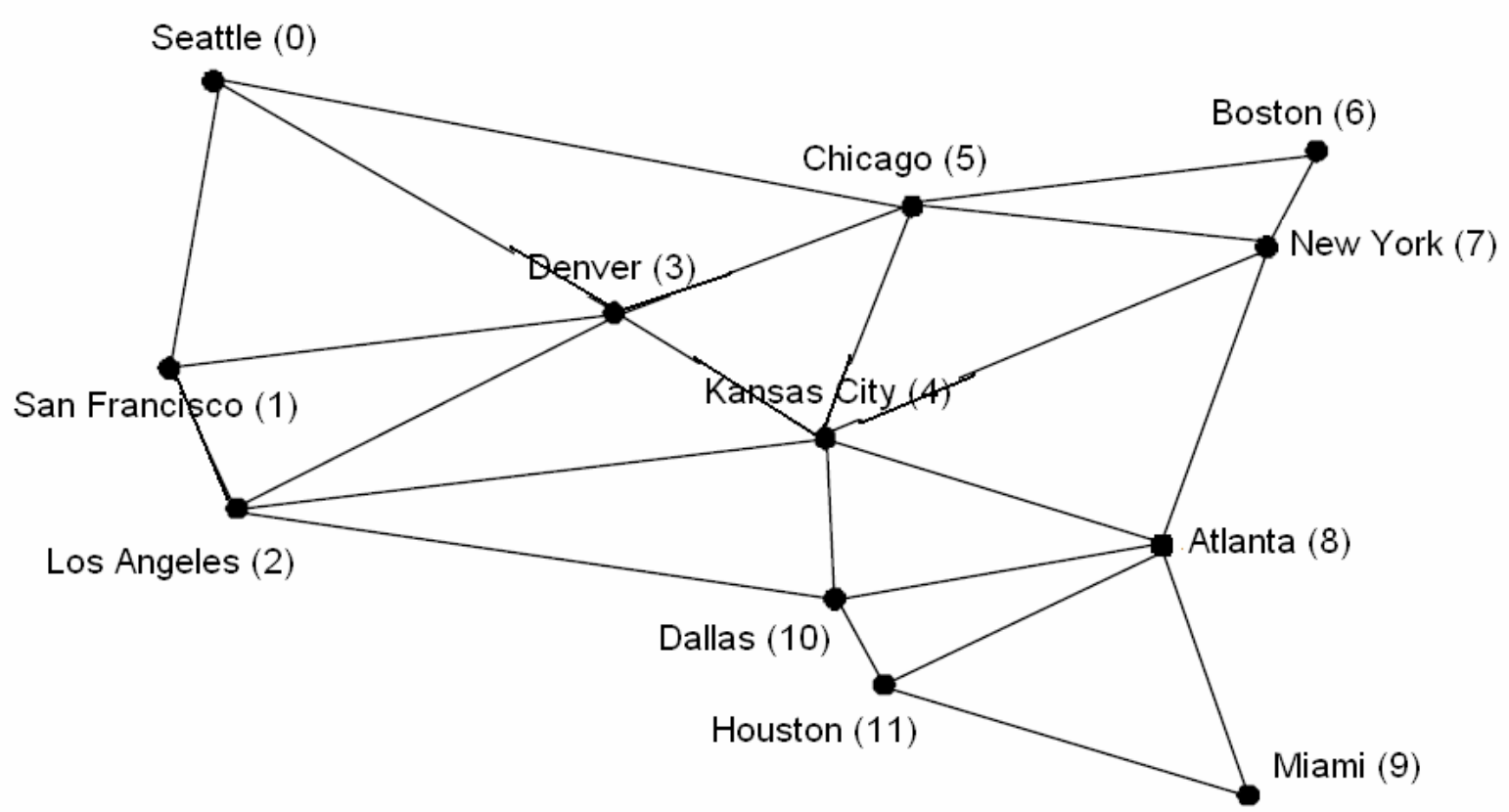
- The first step to build a Graph class is to build a Vertex class to store the vertices of a graph. Remember that "A graph is a pair G=(V,E)"
- We will store the list of vertices in an array and will reference them in the Graph class by their position in the array
- The Vertex class needs two data members:
  - One for the data that identifies the vertex – called label
  - The other is a Boolean member used to keep track of "visits" to the vertex – called wasVisited

```
public class Vertex
{
    public bool wasVisited;
    public string label;
    public Vertex(string label)
    {
        this.label = label;
        wasVisited = false;
    }
}
```

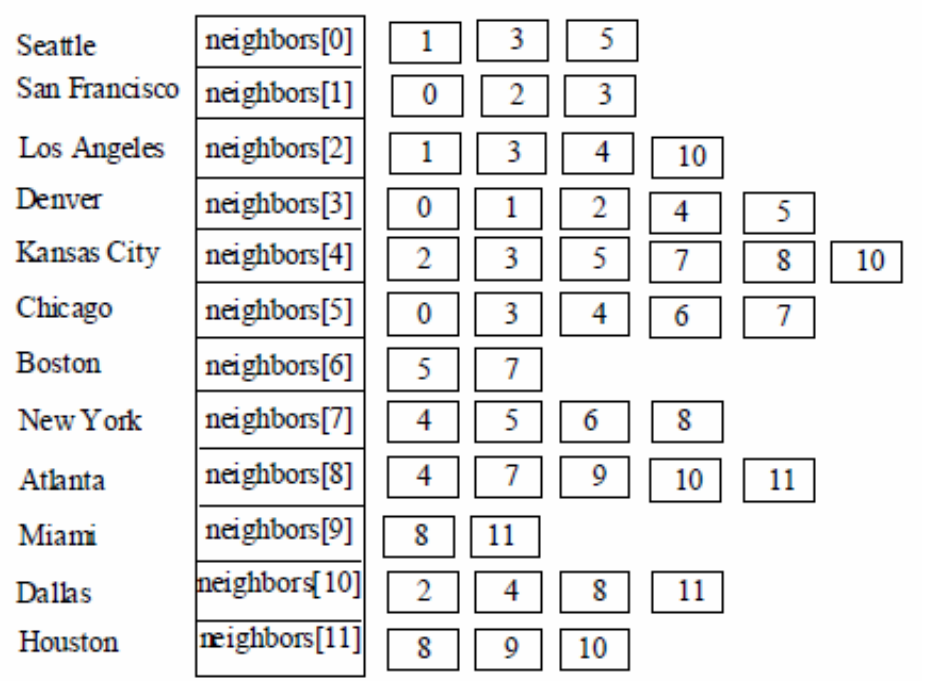
GRAPH CLASS – EDGES

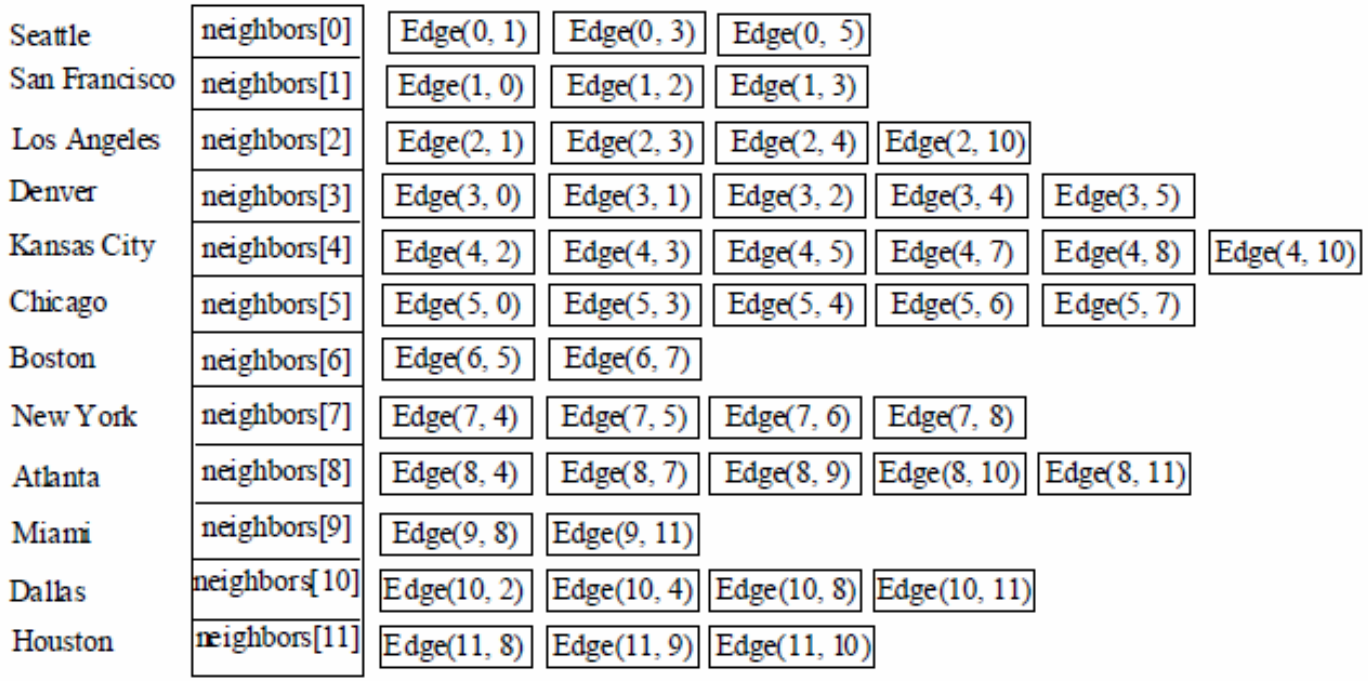
- Remember that "A graph is a pair G=(V,E)"
- The real information about a graph is stored in the edges
- Multiple ways to represent edges
  - Adjacency vertex lists: for a vertex i it contains the vertices that are adjacent to i
  - Adjacency edge lists: for a vertex i contains the edges that are adjacent to i.
  - Adjacency matrix: A two-dimensional array where the elements indicate whether an edge exists between two vertices. The vertices are listed as the headings for the rows and columns.
    - If an edge exists between two vertices, a 1 is placed in that position.
    - If an edge doesn't exist, a 0 is used.

GRAPH CLASS – EDGES (2)

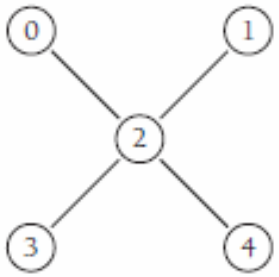


GRAPH CLASS – EDGES (3)



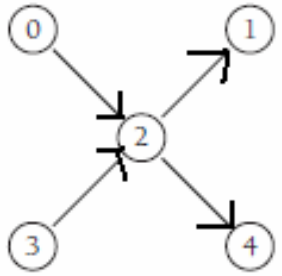


GRAPH CLASS – EDGES (4)



	V_0	V_1	V_2	V_3	V_4
V_0	0	0	1	0	0
V_1	0	0	1	0	0
V_2	1	1	0	1	1
V_3	0	0	1	0	0
V_4	0	0	1	0	0

An Adjacency Matrix.



	V_0	V_1	V_2	V_3	V_4
V_0	0	0	1	0	0
V_1	0	0	0	0	0
V_2	0	1	0	0	1
V_3	0	0	1	0	0
V_4	0	0	0	0	0

An Adjacency Matrix.

GRAPH CLASS – ADDING VERTICES

- First, we need to build a list of the vertices in the graph.

```
public void AddVertex(string label)
{
    vertices[numVerts] = new Vertex(label);
    numVerts++;
}
```

- Example, we can build a small graph by calling this method

```
Graph theGraph = new Graph(5);

theGraph.AddVertex("A");
theGraph.AddVertex("B");
theGraph.AddVertex("C");
theGraph.AddVertex("D");
theGraph.AddVertex("E");
```

GRAPH CLASS – ADDING EDGES

- Then we need to add the edges that connect the vertices

- Example:

```
//this is an sample matrix without a cycle
theGraph.AddEdge(0, 1);
theGraph.AddEdge(0, 2);
theGraph.AddEdge(0, 4);
theGraph.AddEdge(1, 3);
theGraph.AddEdge(2, 4);
theGraph.AddEdge(4, 1);
```

RUNNING TIMES?

- When implementing the code:
  - Do we want to use adjacency matrix or adjacency list?

- To simplify the code on the long run we'll use the adjacency matrix:
  - For this we need to have a max_size (or alternatively we could resize the array as needed)

- Discuss the running time … Note: we won't have only n, we will also have another variable (say m)

RUNNING TIMES?

for undirected graphs make sure you enter the same value for adjMatrix[i,j] and adjMatrix[j,i]

```
33  class Vertex
34  {
35      public string label;
36      public bool wasVisited;
37
38      public Vertex(string newLabel)
39      {
40          label = newLabel;
41          wasVisited = false;
42      }
43  }
44
45  class Graph
46  {
47      //A graph is a pair G=(V,E) where
48      int MAX_SIZE;       //we can hard code it or ask the use th specify one ...
49      int size;           //the actual size, always <= MAX_SIZE!
50      Vertex[] vertices;  //this is V
51      int[,] adjMatrix;       //this will be the adjacency matrix
52
53
54      //adding a new Vertex
55      public void addVertex(string label)
56      {
57          if (size < MAX_SIZE)
58          {
59              vertices[size] = new Vertex(label);
60              size++;
61          }
62          else
63              Console.WriteLine("Fatal error: the vertices array is full!");
64      }
65
66      //print vertices:
67      public void printVertices()
68      {
69          Console.Write("List of vertices: ");
70
71          for( int i=0; i<size; i++)
72              Console.Write(vertices[i].label + "  ");
73          Console.WriteLine();
74      }
```

```
76      //adding edges - to simplify the code let's use indexes instead of labels when adding edges ...
77      public void addEdge(int i, int j, int weight =1)
78      {
79          adjMatrix[i,j] = weight;
80      }
81
82
83      //print edges:
84      public void printEdges()
85      {
86          Console.Write("Printing adjacency list: \n");
87          for (int i = 0; i < size; i++)
88          {
89              for (int j = 0; j < size; j++)
90                  Console.Write(adjMatrix[i,j] + "  ");
91              Console.WriteLine();
92
93          }
94          Console.WriteLine();
95      }
96
97
98      //constructor
99      public Graph()
100     {
101         size = 0;
102         MAX_SIZE = 20;       //we can hard code it or ask the use th specify one ...
103         vertices = new Vertex[MAX_SIZE];
104         adjMatrix = new int[MAX_SIZE, MAX_SIZE];
105         //initialize the matrix to 0;
106         for (int i = 0; i < MAX_SIZE; i++)
107             for (int j = 0; j < MAX_SIZE; j++)
108                 adjMatrix[i,j] = 0;
109     }
110
111 }
```

GRAPH ALGORITHMS: TOPOLOGICAL SORTING

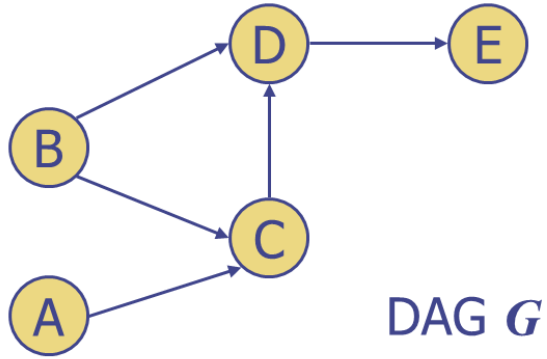
- A directed acyclic graph (DAG) is a digraph that has no directed cycles

- A topological ordering of a digraph is a numbering

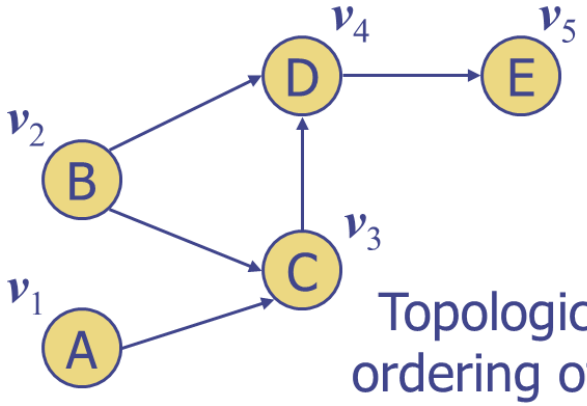
$$v_1, \ldots, v_n$$

of the vertices such that for every edge (v_i, v_j), we have $i < j$

- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints



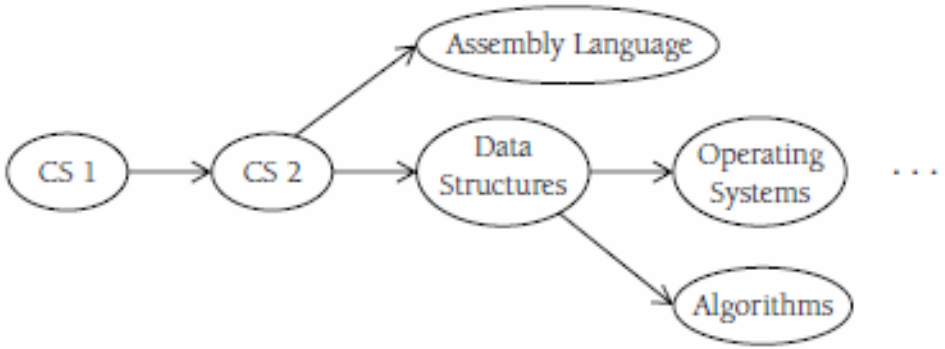
DAG G



Topological ordering of G

GRAPH ALGORITHMS: TOPOLOGICAL SORTING

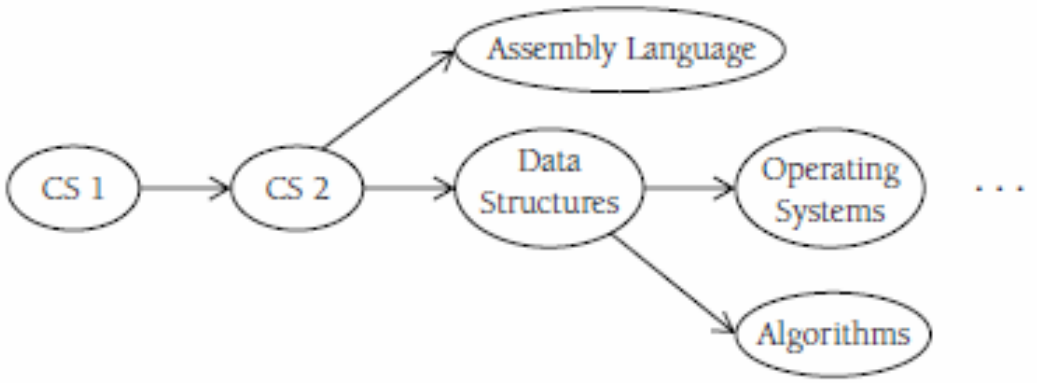
- Instead of numbering one could just display the specific order in which a sequence of vertices must be followed in a directed graph.
- The sequence of courses a college student must take on their way to a degree can be modeled with a directed graph.
- A student can't take the Data Structures course until they've had the first two introductory Computer Science courses.
- As an example, the Figure on the next page depicts a directed graph modeling part of the typical Computer Science curriculum.
- A topological sort of this graph would result in the following sequence:
  1. CS1
  2. CS2
  3. Assembly Language
  4. Data Structures
  5. Operating Systems
  6. Algorithms
- Courses 3 and 4 can be taken at the same time, as can 5 and 6



A Directed Graph Model of Computer Science Curriculum Sequence.

GRAPH ALGORITHMS: TOPOLOGICAL SORTING (2)

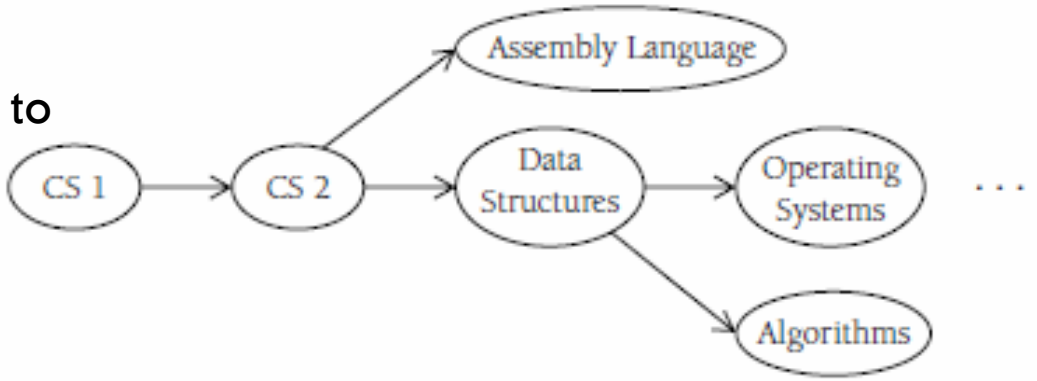
- The basic algorithm for topological sorting
  1. Find a vertex that has no successors.
  2. Add the vertex to a list of vertices.
  3. Remove the vertex from the graph.
  4. Repeat Step 1 until all vertices are removed.



A Directed Graph Model of Computer Science Curriculum Sequence.

GRAPH ALGORITHMS: TOPOLOGICAL SORTING (3)

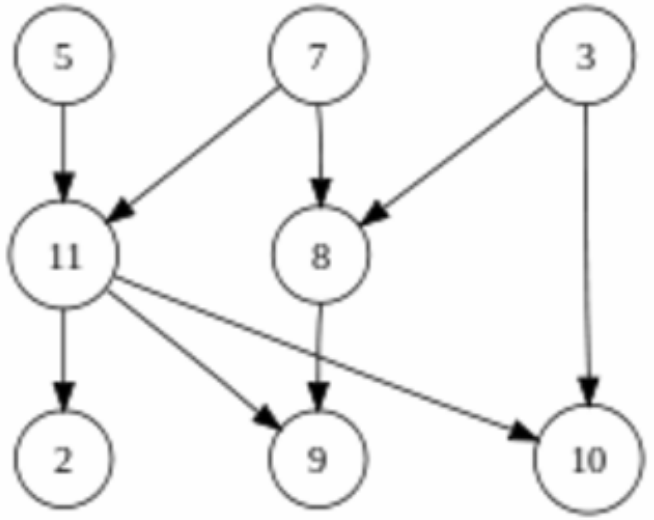
- The algorithm will work from the end of the directed graph to the beginning.
- Assuming that *Operating Systems* and *Algorithms* are the last vertices in the graph
  - Neither of them have successors and so they are added to the list and removed from the graph.
- Next come *Assembly Language* and *Data Structures.*
  - These vertices now have no successors and so they are removed from the list.
- Next will be *CS2*.
  - Its successors have been removed so it is added to the list.
  - Finally, we're left with CS1.



A Directed Graph Model of Computer Science Curriculum Sequence.

GRAPH ALGORITHMS: TOPOLOGICAL SORTING (4)

- Topological Sorting
  - If there are two or more than two vertices having no successors, how to choose a vertex from the candidates?



	2	3	5	7	8	9	10	11
2	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0
5	0	0	0	0	0	0	0	1
7	0	0	0	0	1	0	0	1
8	0	0	0	0	0	1	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	1	0	0	0	0	1	1	0

GRAPH ALGORITHMS: TOPOLOGICAL SORTING (5)

The graph has many valid topological sorts, including:

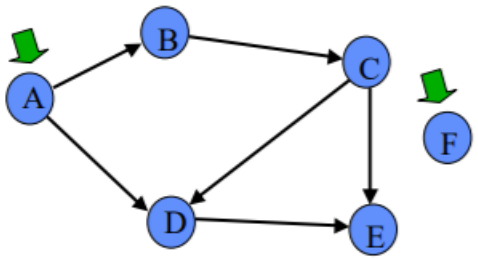
- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

ANOTHER ALGORITHM ... SKIPPED

- Next is another algorithm that ca be used to get a topological sorting
- https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

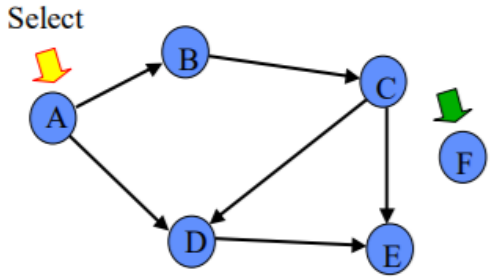
Topological Sort Algorithm

Step 1: Identify vertices that have no incoming edge
- The "**in-degree**" of these vertices is zero



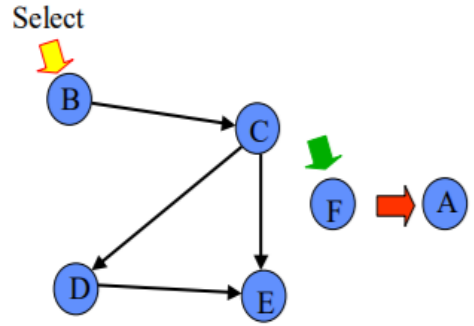
Topological Sort Algorithm

Step 1: Identify vertices that have no incoming edges
- Select one such vertex



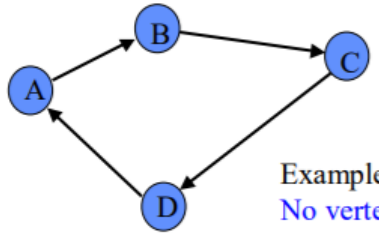
Topological Sort Algorithm

Repeat Steps 1 and Step 2 until graph is empty



Topological Sort Algorithm

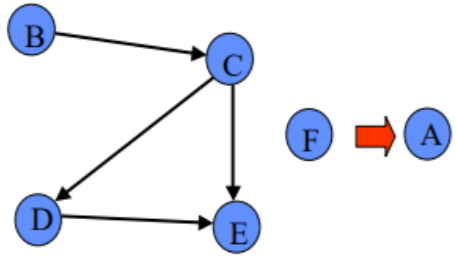
Step 1: Identify vertices that have no incoming edge
- If *no such edges*, graph has cycles (cyclic graph)



Example of a cyclic graph:
No vertex of in-degree 0

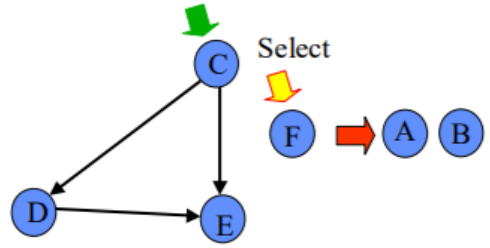
Topological Sort Algorithm

Step 2: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



Topological Sort Algorithm

Repeat Steps 1 and Step 2 until graph is empty

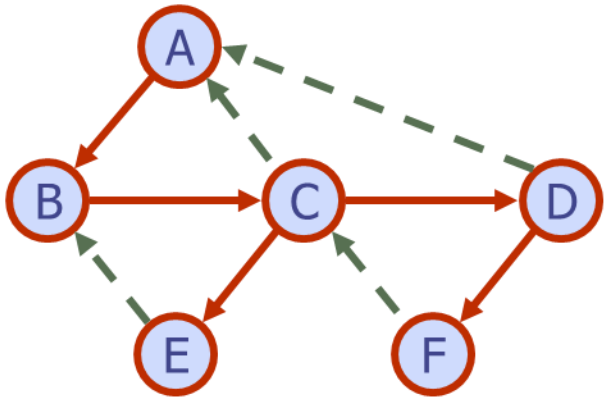


GRAPH TRAVERSALS

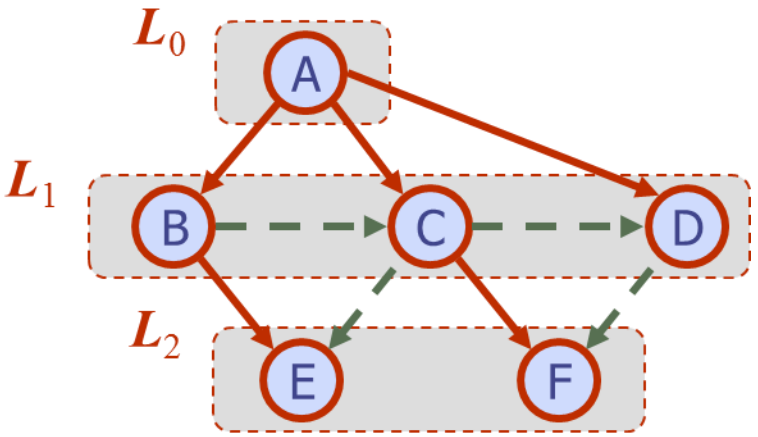
- a **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges.

DFS VS. BFS – DETAILS COMING NEXT …

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√



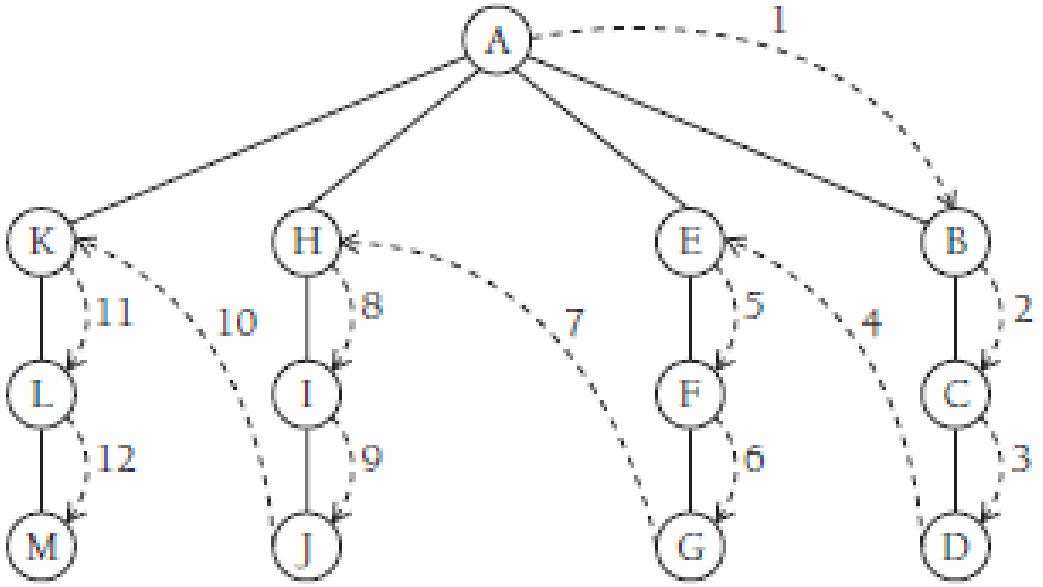
DFS



BFS

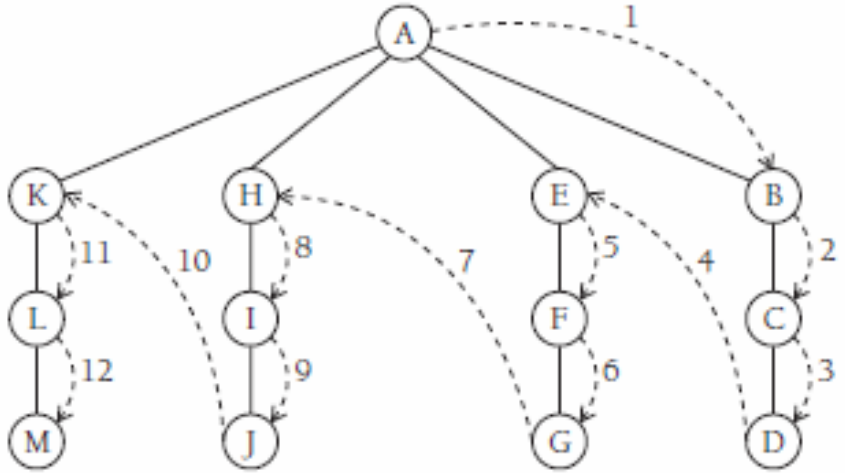
DEPTH-FIRST SEARCH (DFS)

- **Depth-first search (DFS)** involves following a path from the beginning vertex until it reaches the last vertex, then backtracking and following the next path until it reaches the last vertex, and so on until there are no more paths left.



Depth-First Search.

DFS - ALGORITHM



Depth-First Search.

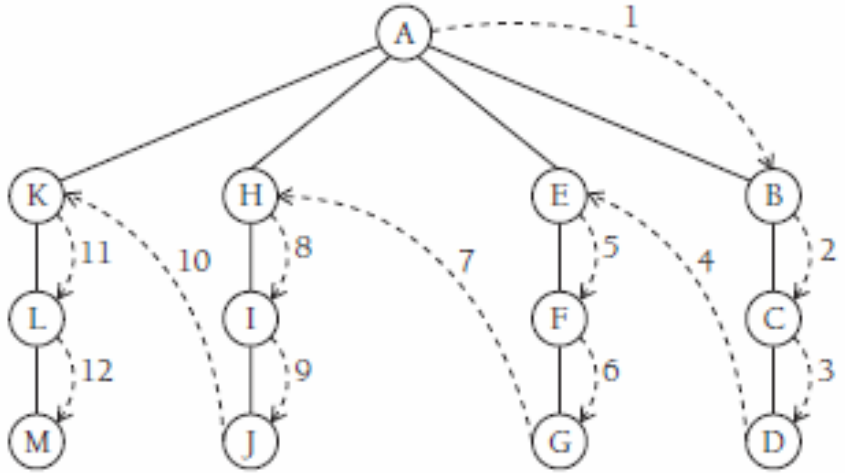
- First, pick a starting point, which can be any vertex.
  - Visit the vertex, push it onto a stack, and mark it as visited.
    - Reason for marking a node as visited: we don't end up with an infinite loop when we land on a cycle
  - Then you go to the next [adjacent] vertex that is unvisited, push it on the stack, and mark it.
  - This continues until you reach the last vertex.

- Then you check to see if the top vertex has any unvisited adjacent vertices.
  - If it doesn't, then you pop it off the stack and check the next vertex.
  - If you find one, you start visiting adjacent vertices until there are no more, check for more unvisited adjacent vertices, and continue the process.

- When you finally reach the last vertex on the stack and there are no more adjacent, unvisited vertices, you've performed a depth-first search.

DFS – C# CODE

```
200         public void DFS() //only works on a connected component!!!
201         {
202             Console.WriteLine("Printing the Depth-First Search (DFS) traversal:");
203             if (size == 0)//sanity check
204                 return;
205
206             Stack<int> s = new Stack<int>();//we'll put the index of each vertex we visit
207
208             //First, pick a starting point, which can be any vertex - let's start with the
209             //Visit the vertex - we'll display it, push it onto a stack, and mark it as vi
210             Console.WriteLine(vertices[0].label);   //display it
211             s.Push(0);                              //push it onto a stack
212             vertices[0].wasVisited = true;          //mark it as visited
213
214             //perform the DFS
215             while (s.Count > 0)
216             {
217                 //go to the next [adjacent] vertex that is unvisited, push it on the stack
218                 int v = FindUnvisitedSuccessor(s.Peek()); //If you find one, you start vis
219                 if (v != -1)
220                 {
221                     Console.WriteLine(vertices[v].label);   //display it
222                     s.Push(v);                              //push it onto a stack
223                     vertices[v].wasVisited = true;          //mark it as visited
224                 }
225                 else    //s.Peek() has no more unvisited successors hence pop it off the s
226                     s.Pop();
227             }//When you finally reach the last vertex on the stack and there are no more a
228
229             Console.WriteLine();
```



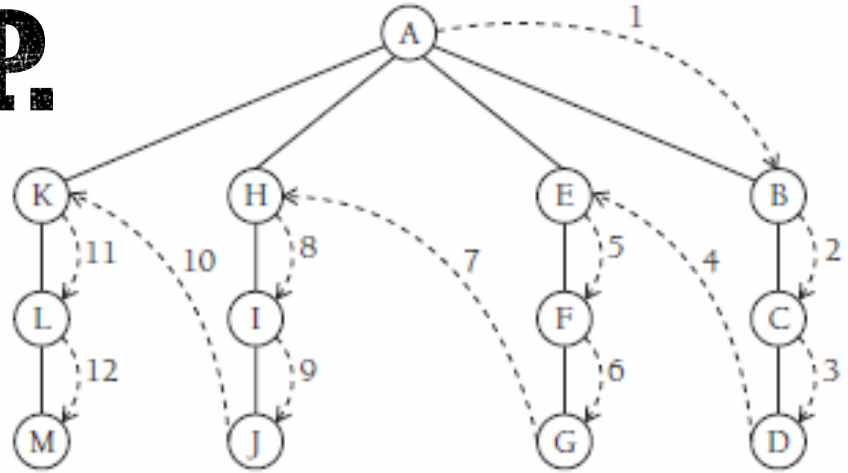
Depth-First Search.

how can we use the adjacency matrix to find successors of a vertex?

```
191         int FindUnvisitedSuccessor(int i) //for the Vertex vertices[i] finds
192         {
193             //look through all edges i->k, where k = 0, 1, ..., size -1
194             for (int k = 0; k < size; k++)
195                 if (adjMatrix[i, k] > 0 && vertices[k].wasVisited == false)
196                     return k;
197             return -1;
198         }
```

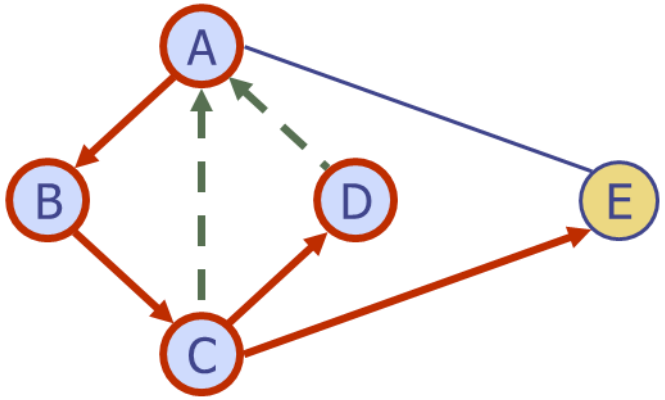
DFS – C# CODE – ALL COMP.

```
232         public void DFSAllComponents() //works for all connected components !!!
233         {
234             Console.WriteLine("Printing the Depth-First Search (DFS) traversal - all components:");
235             if (size == 0)//sanity check
236                 return;
237
238             for (int i = 0; i < size; i++)
239             {
240                 if (vertices[i].wasVisited == false)
241                 {
242                     Console.WriteLine("connected component: ");
243                     Stack<int> s = new Stack<int>();//we'll put the index of each vertex we visit
244
245                     //First, pick a starting point, which can be any vertex - let's start with the first vertex vertices[0]
246                     //Visit the vertex - we'll display it, push it onto a stack, and mark it as visited.
247                     Console.WriteLine(vertices[i].label);   //display it
248                     s.Push(i);                              //push it onto a stack
249                     vertices[i].wasVisited = true;          //mark it as visited
250
251                     //perform the DFS
252                     while (s.Count > 0)
253                     {
254                         //go to the next [adjacent] vertex that is unvisited, push it on the stack, and mark it.  This continu
255                         int v = FindUnvisitedSuccessor(s.Peek()); //If you find one, you start visiting adjacent vertices unti
256                         if (v != -1)
257                         {
258                             Console.WriteLine(vertices[v].label);   //display it
259                             s.Push(v);                              //push it onto a stack
260                             vertices[v].wasVisited = true;          //mark it as visited
261                         }
262                         else    //s.Peek() has no more unvisited successors hence pop it off the stack
263                             s.Pop();
264                     }//When you finally reach the last vertex on the stack and there are no more adjacent, unvisited vertices,
265                     Console.WriteLine();
266                 }
267             }
268         }
```



Depth-First Search.

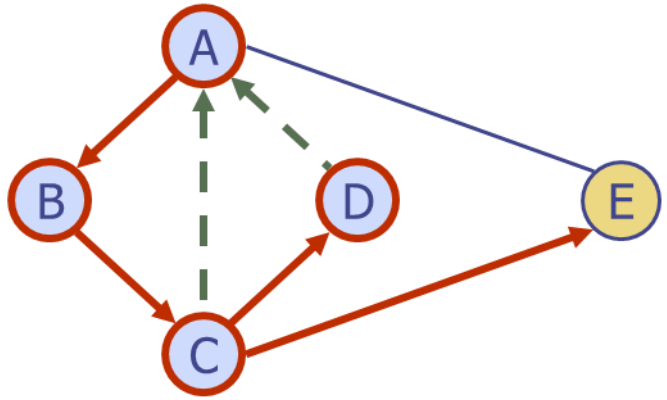
DFS – SKIPPED



- **Uses labeling and recursivity instead of a stack …**

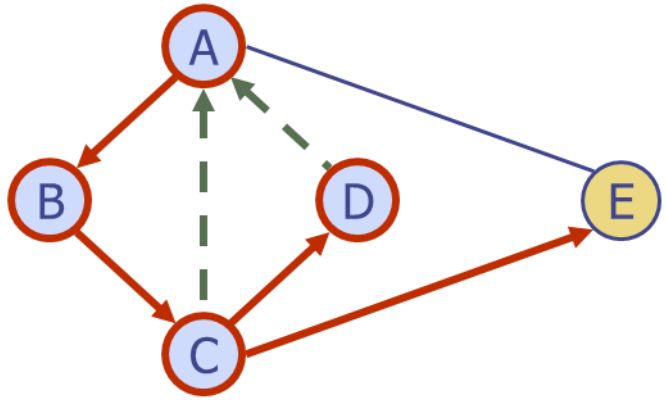
- **Depth-first search (DFS)** in an undirected graph G is analogous to wandering in a labyrinth with a string and a can of paint without getting lost.
  - We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as **"visited."** The vertex s is now our "current" vertex- call our current vertex u.
  - We then traverse G by considering an (arbitrary) edge (u,v) incident to the current vertex u.
    - If the edge (u, v) leads us to an already visited (that is, painted) vertex v, we immediately return to vertex u.
    - If, on the other hand, (u, v) leads to an unvisited vertex v, then we unroll our string, and go to v. We then paint v as " visited," and make it the current vertex, repeating the computation above.

DFS – SKIPPED



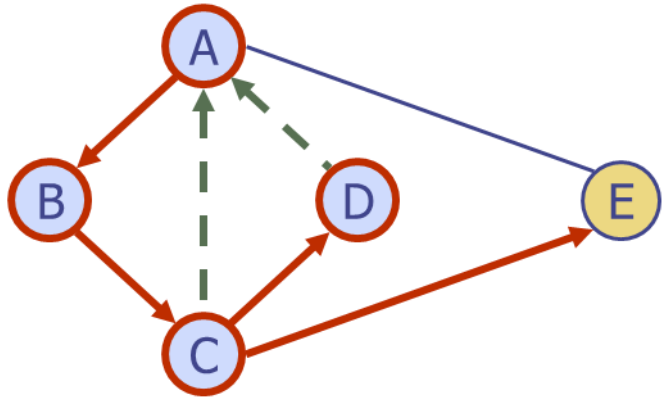
- Eventually, we get to a "dead end," that is, a current vertex u such that all the edges incident on u lead to vertices already visited.
  Thus, taking any edge incident on u causes us to return to u.
- To get out of this impasse, we roll our string back up, backtracking along the edge that brought us to u, going back to a previously visited vertex v.
- We then make v our current vertex and repeat the computation above for any edges incident upon v that we have not looked at before.
- If all of v's incident edges lead to visited vertices, then we again roll up our string and backtrack to the vertex we came from to get to v, and repeat the procedure at that vertex.
- Thus, we continue to backtrack along the path that we have traced so far until we find a vertex that has yet unexplored edges, take one such edge, and continue the traversal.
- The process terminates when our backtracking leads us back to the start vertex s, and there are no more unexplored edges incident on s.

DFS – IMPORTANT



- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G:
  - Visits all the vertices and edges of G
  - Determines whether G is connected
    - How?
  - Computes the connected components of G
    - How?
  - Computes a spanning forest of G
    - How?
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
    - How?
  - Find a cycle in the graph
    - How?

DFS – SKIP (LABELS EDGES TOO!!!)

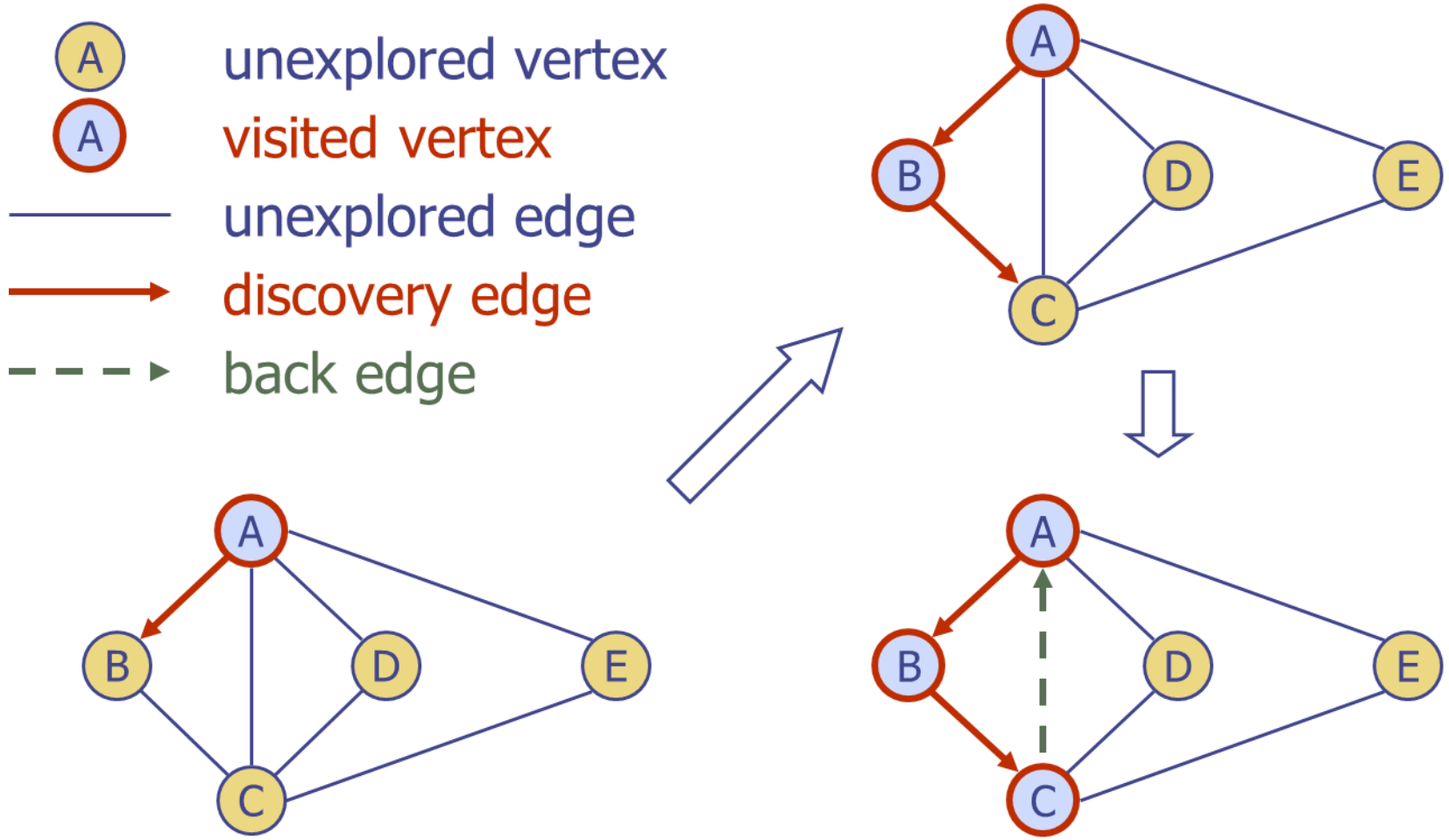


- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

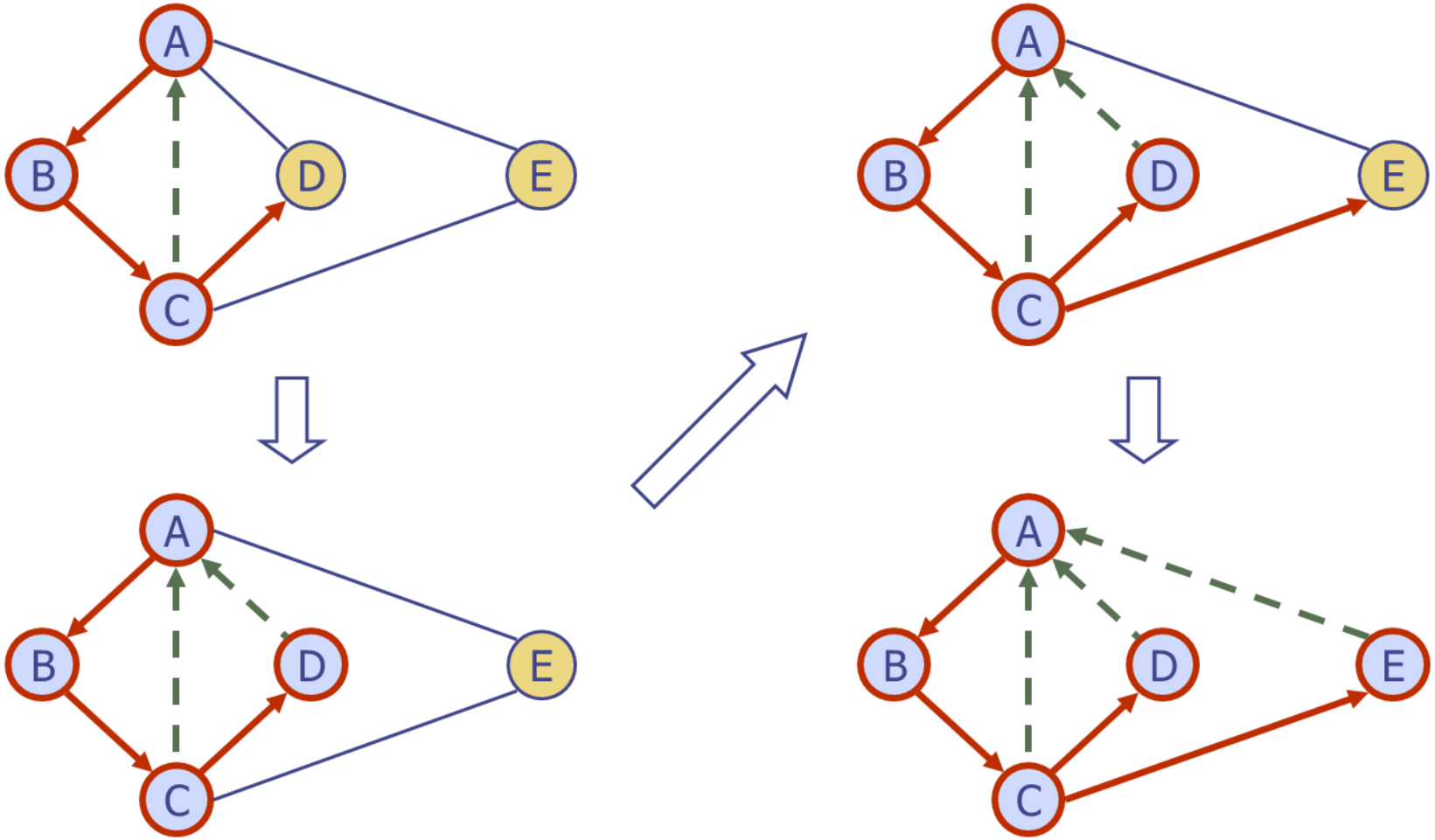
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()
    e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
    if v.getLabel() = UNEXPLORED
        DFS(G, v)
```

```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
    in the connected component of v
    as discovery edges and back edges
  v.setLabel(VISITED)
  for all e ∈ G.incidentEdges(v)
    if e.getLabel() = UNEXPLORED
      w ← e.opposite(v)
      if w.getLabel() = UNEXPLORED
        e.setLabel(DISCOVERY)
        DFS(G, w)
      else
        e.setLabel(BACK)
```

EXAMPLE - SKIP (LABELS EDGES TOO!!!)



EXAMPLE (CONT.) - SKIP (LABELS EDGES TOO!!!)



PATH FINDING – SKIP

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

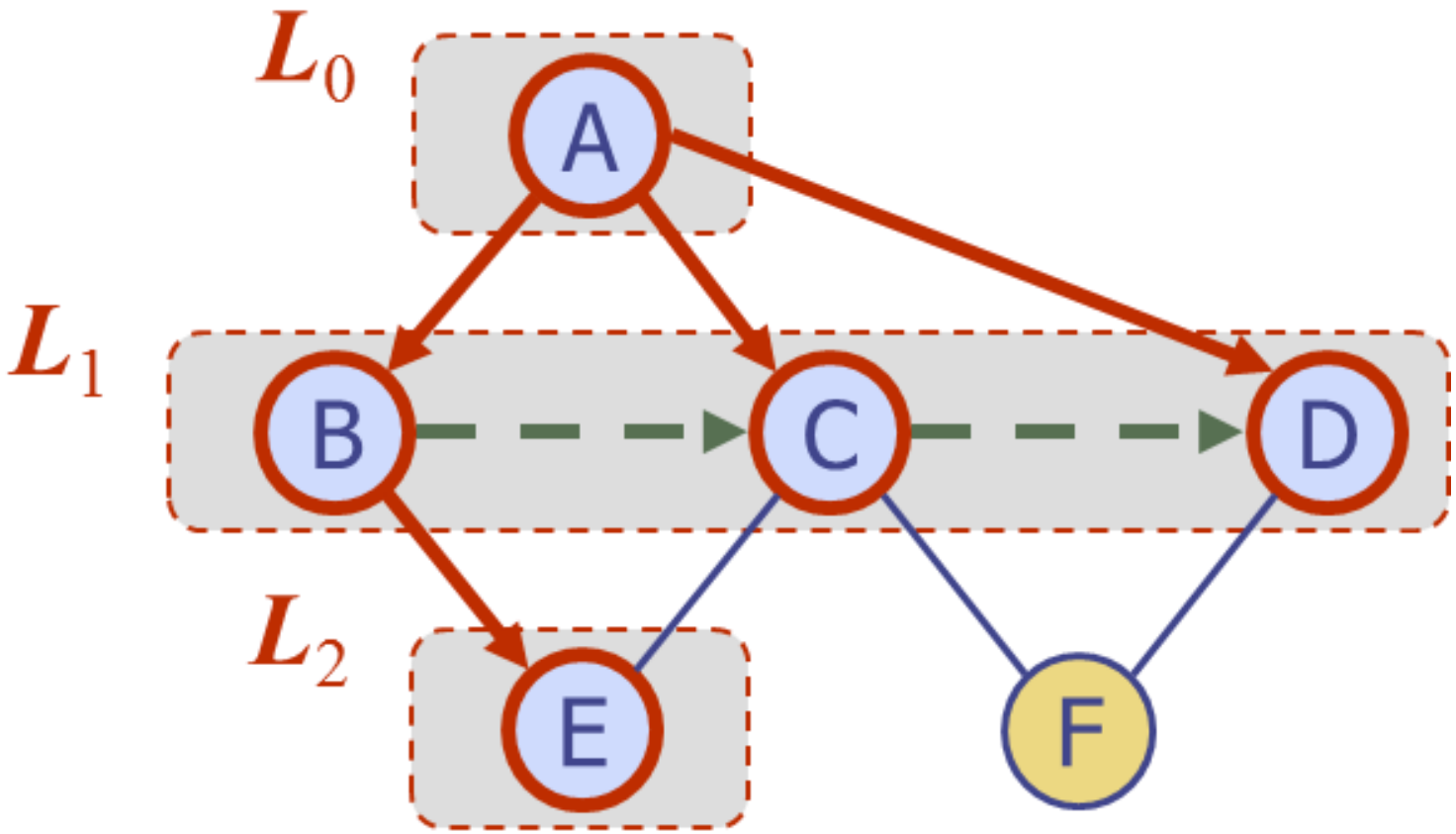
```
Algorithm pathDFS(G, v, z)
  v.setLabel(VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ v.incidentEdges()
    if e.getLabel() = UNEXPLORED
      w ← e.opposite(v)
      if w.getLabel() = UNEXPLORED
        e.setLabel(DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        e.setLabel(BACK)
  S.pop(v)
```

CYCLE FINDING - SKIPPED

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS(G, v)
    v.setLabel(VISITED)
    S.push(v)
    for all e ∈ v.incidentEdges()
        if e.getLabel() = UNEXPLORED
            w ← e.opposite(v)
            S.push(e)
            if w.getLabel() = UNEXPLORED
                e.setLabel(DISCOVERY)
                cycleDFS(G, w)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

BREADTH-FIRST SEARCH (BFS)



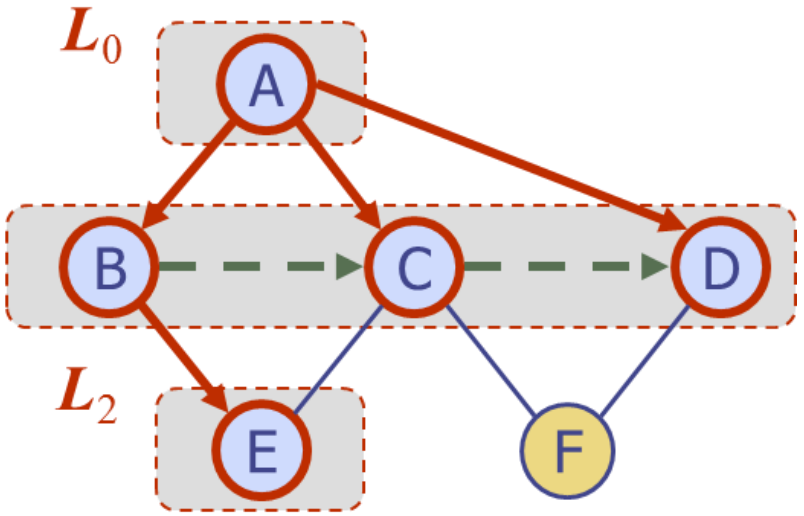
BREADTH-FIRST SEARCH (BFS)

- BFS starts at vertex s, which is at level 0 and defines the "anchor" for our string.

- In the first round, we let out the string the length of one edge and we visit all the vertices we can reach without unrolling the string any farther. In this case, we visit, and paint as "visited," the vertices adjacent to the start vertex s— these vertices are placed into level 1.

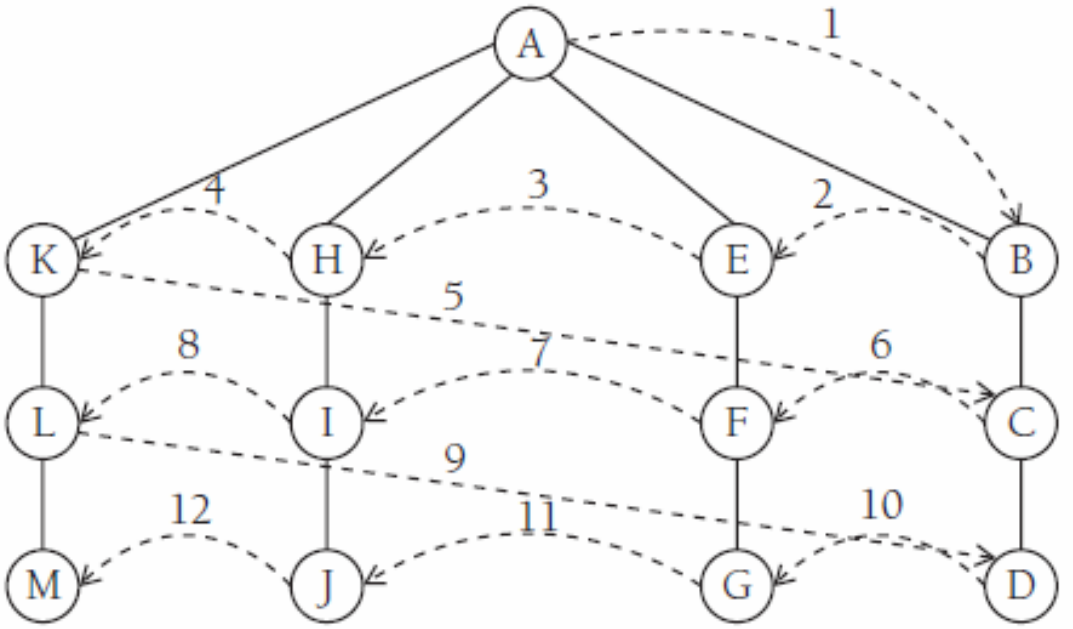
- In the second round, we unroll the string the length of two edges and we visit all the new vertices we can reach without unrolling our string any farther. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on.

- The BFS traversal terminates when every vertex has been visited.



BFS

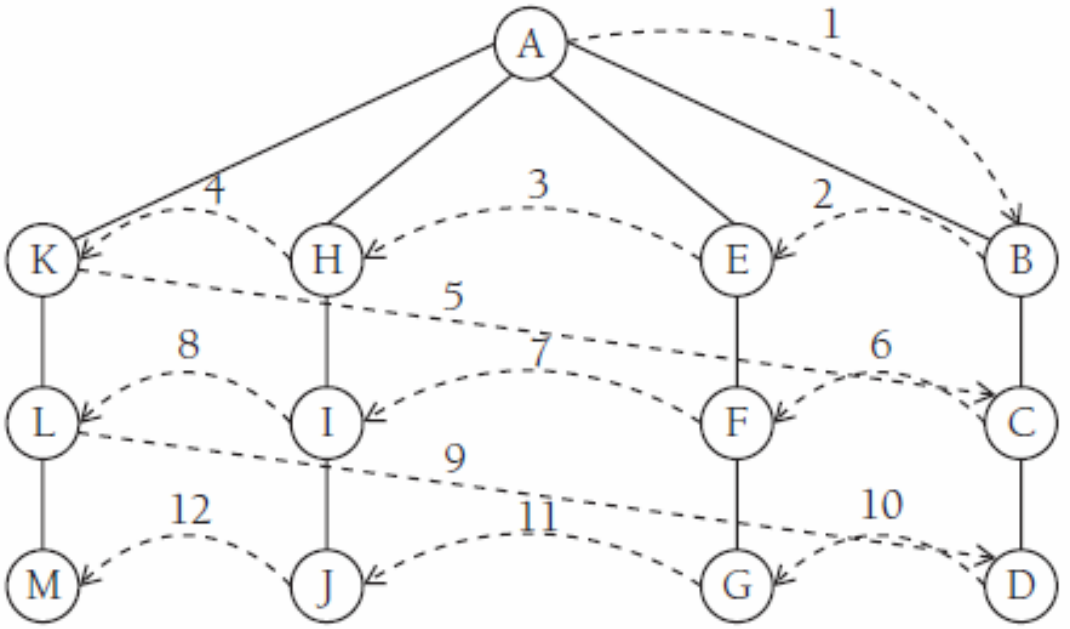
- A breadth-first search starts at a first vertex and tries to visit vertices as close to the first vertex as possible.
- In essence, this search moves through a graph layer by layer, examining the layers closer to the first vertex first and moving down to the layers farthest away from the starting vertex.
- The algorithm for breadth-first search uses a queue instead of a stack, though a stack could be used.



Breath-First Search.

BFS - ALGORITHM

- Find an unvisited vertex that is adjacent to the current vertex, mark it as visited, and add to a queue.
- If an unvisited, adjacent vertex can't be found, remove a vertex from the queue (as long as there is a vertex to remove), make it the current vertex, and start over.
- If the second step can't be performed because the queue is empty, the algorithm is finished



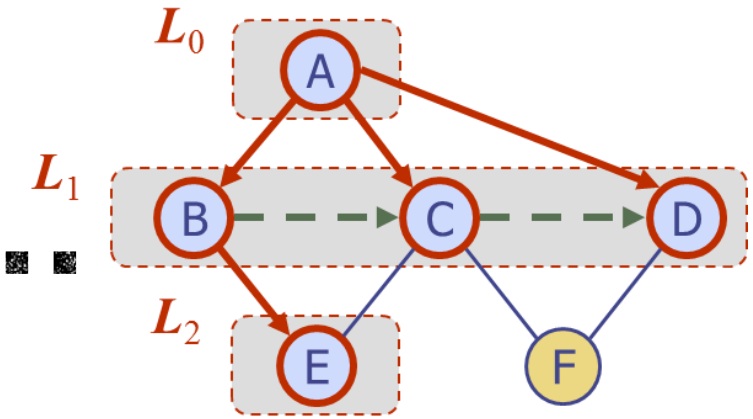
Breath-First Search.

BFS – C# IMPLEMENTATION

- There are two loops in this method.
- The outer loop runs while the queue has data in it, and the inner loop checks adjacent vertices to see if they've been visited.

```
277         public void BFS()//only works on a connected component!!!
278         {
279             Console.WriteLine("Printing the Breadth-First Search (BFS) traversal:");
280             if (size == 0)//sanity check
281                 return;
282
283             Queue<int> q = new Queue<int>();
284
285             // A breadth-first search starts at a first vertex
286             Console.WriteLine(vertices[0].label);   //display it
287             q.Enqueue(0);
288             vertices[0].wasVisited = true;          //mark it as visited
289
290             while(q.Count>0)
291             {
292                 Console.WriteLine();
293
294                 // ... and tries to visit vertices as close to the first vertex in the queue as possible.
295                 int top = q.Dequeue(); //dequeue the vertex
296
297                 for (int i = 0; i < size; i++)
298                     if (adjMatrix[top, i] > 0 && vertices[i].wasVisited == false)
299                     {
300                         Console.WriteLine(vertices[i].label);   //display it
301                         q.Enqueue(i);
302                         vertices[i].wasVisited = true;          //mark it as visited
303                     }
304             }
305         }
306
```

BFS - USES EDGE LABELING ...



- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

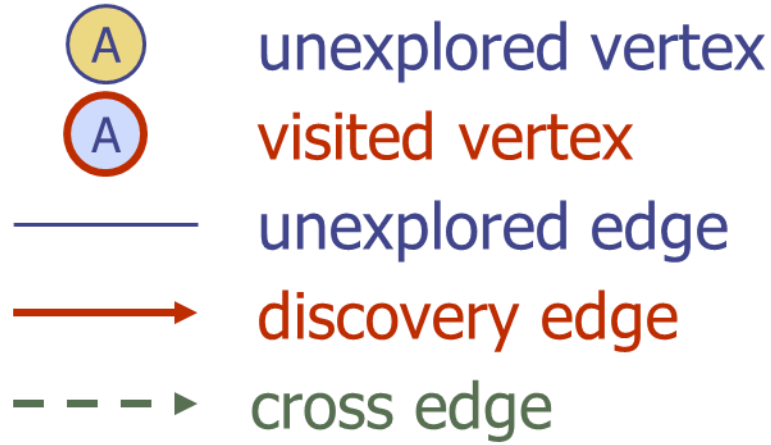
BFS ALGORITHM - SKIP - USES EDGE LABELING ...

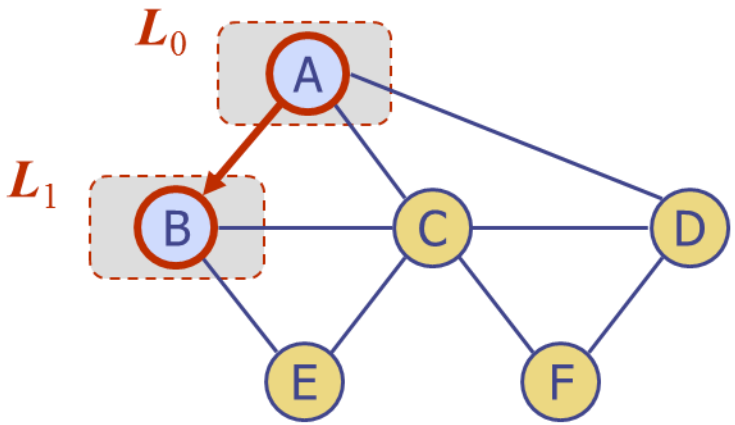
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

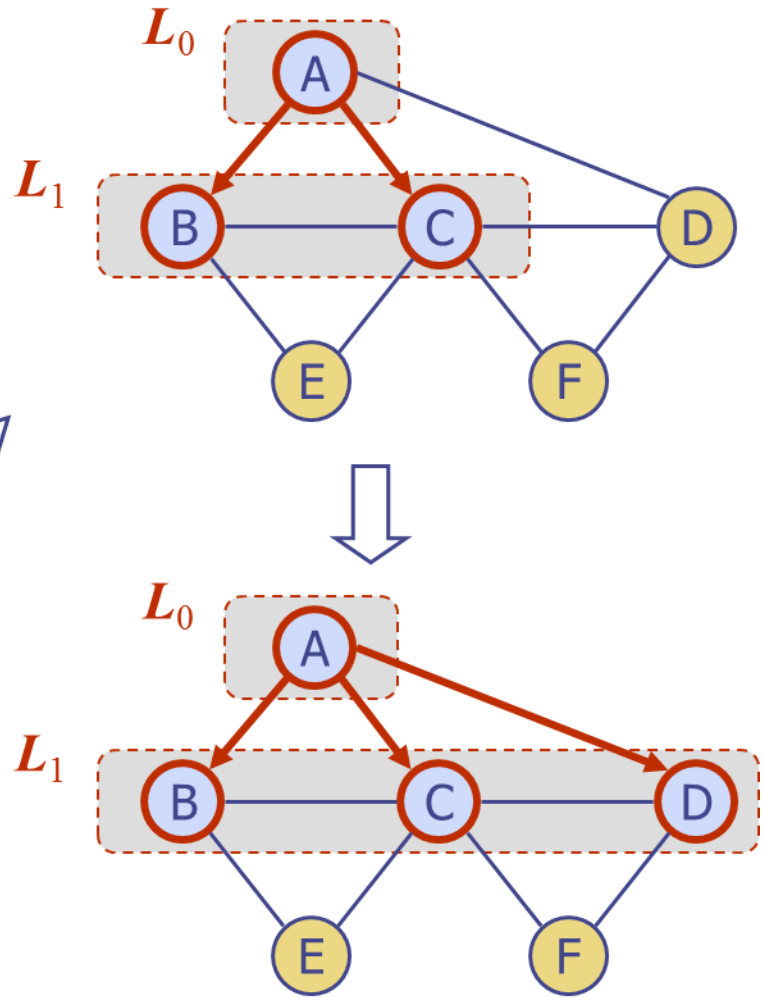
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()
    e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
    if v.getLabel() = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertBack(s)
  s.setLabel(VISITED)
  i ← 0
  while ¬Li.empty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ v.incidentEdges()
        if e.getLabel() = UNEXPLORED
          w ← e.opposite(v)
          if w.getLabel() = UNEXPLORED
            e.setLabel(DISCOVERY)
            w.setLabel(VISITED)
            Li+1.insertBack(w)
          else
            e.setLabel(CROSS)
    i ← i +1
```

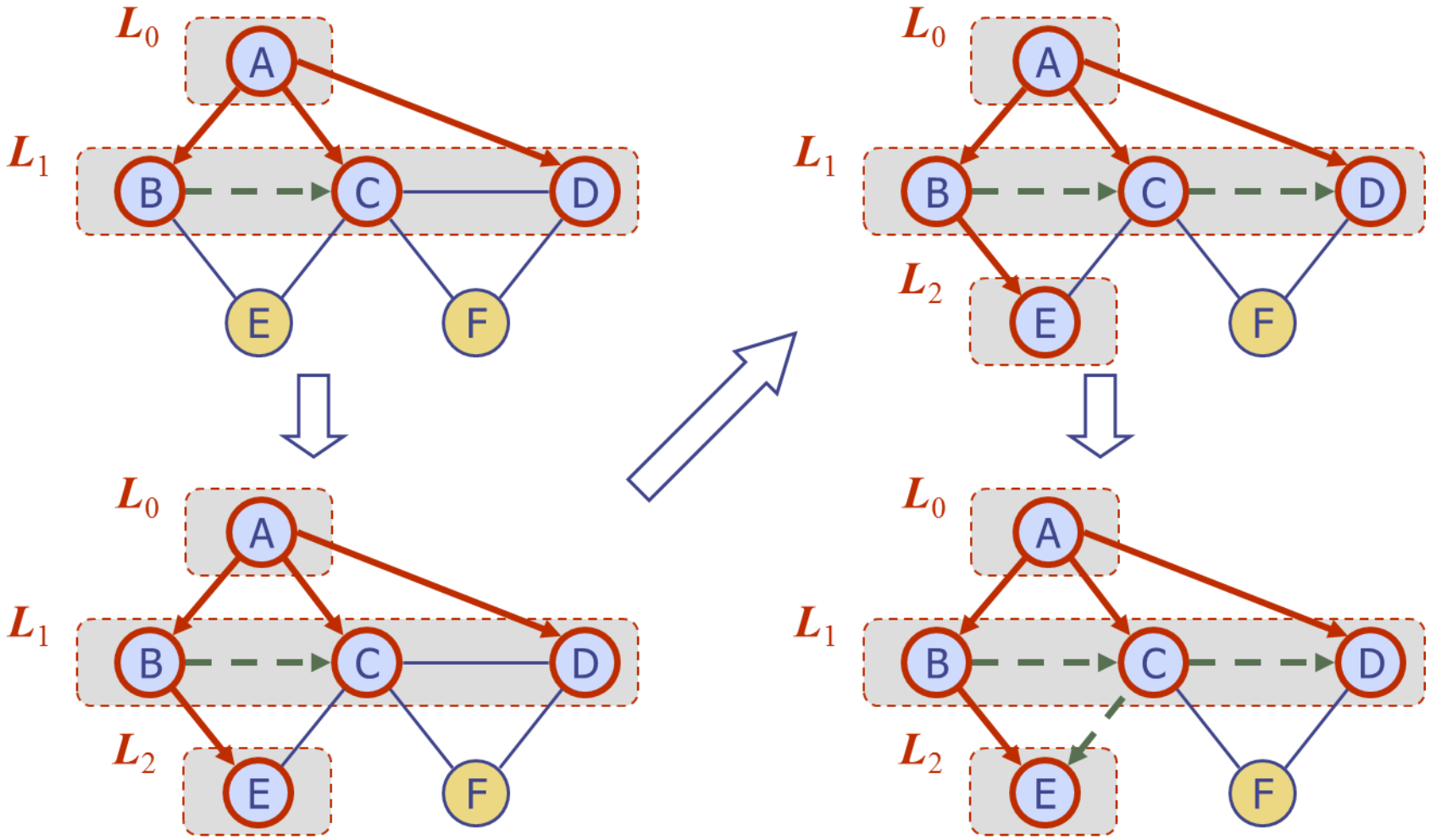
EXAMPLE – SKIP



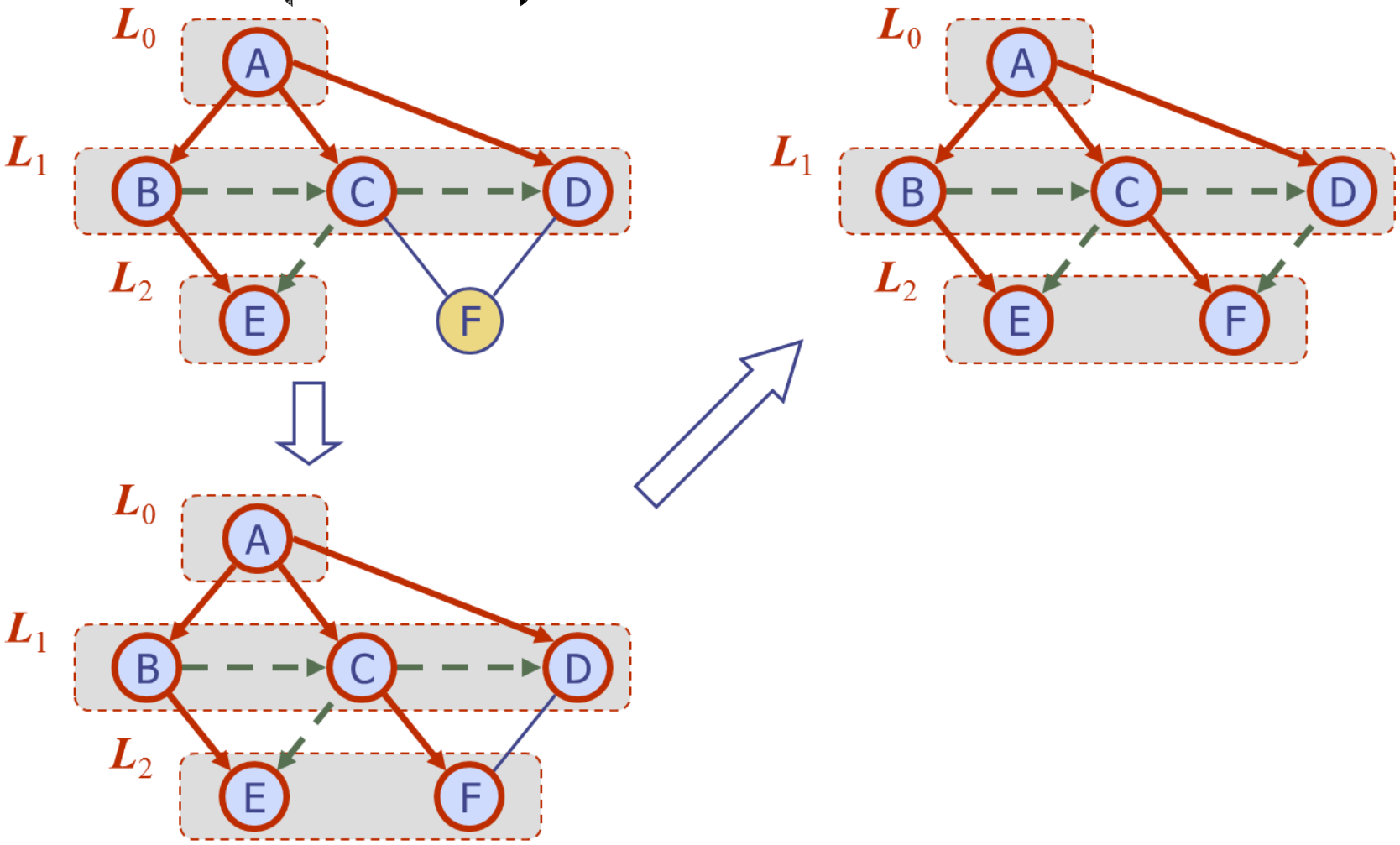




EXAMPLE (CONT.) - SKIP

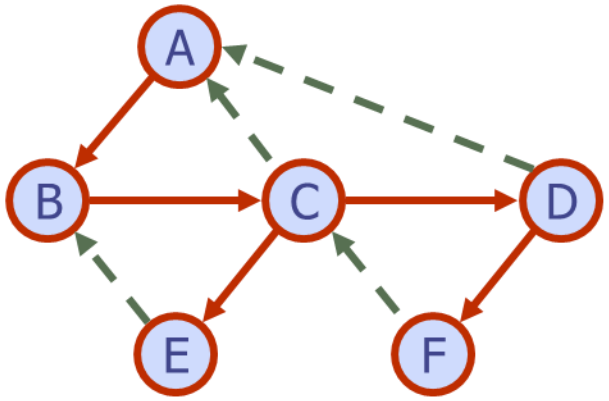


EXAMPLE (CONT.) - SKIP

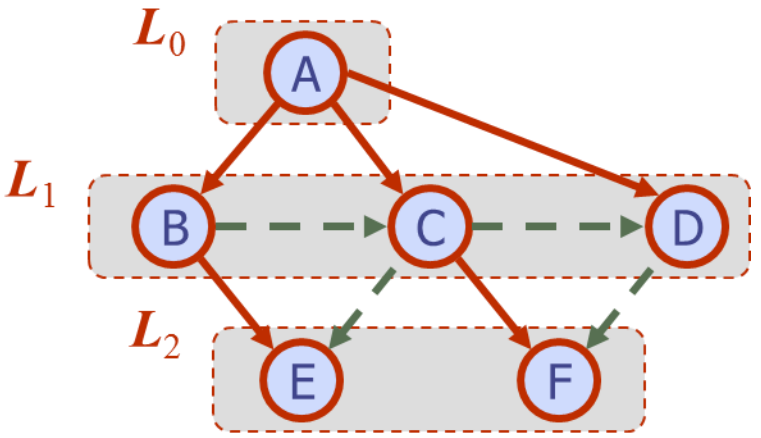


DFS VS. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√



DFS



BFS

INTERVIEW QUESTIONS (JUST FOR PRACTICE!)

- Which data structures are used for BFS and DFS of a graph?

INTERVIEW PROBLEMS (JUST FOR PRACTICE!)

- Given a graph, how would you visit every node only once (without visiting any previous nodes)? [Glassdoor.com: Microsoft interview question]

HOMEWORK FOR MODULE 5

- DUE: see Moodle, 11:59 pm

- **Required Homework**:
  - Rewrite the Graph class we wrote in class so it uses an **Adjacency List** instead of an **Adjacency Matrix**. For the adjacency list feel free to use array-lists (recommended), linked-lists, or any other container class (other than matrices!).
    - look up the notes on **array lists** (or google them) to see how to use array lists.
    - **code that does not compile or crashes at start is automatically graded with 0!**
- **Bonus-Points Homework**:
  - [20 bonus points] write a 2-page essay about **Dijkstra's algorithm**. Make sure your work contains a title, your name, and references. You should not copy & paste your work, make sure you rewrite everything you put in your essay and that you do understand it.
  - [20 bonus points - **ALL OR NOTHING**] Write a C# implementation of the **Dijkstra's algorithm** and make sure your main method contains some testing of the algorithm. In order to get credit your code has to be complete and correct (no compiling errors and no crashing).
  - [20 bonus points] write a 2-page essay about **greedy algorithms**. Make sure your work contains a title, your name, and references.