COMP1002 DATA STRUCTURES AND ALGORITHMS

LECTURE 6: GRAPHS

Curtin University

Discipline of Computing

Last updated: 6 April 2020

Copyright Warning

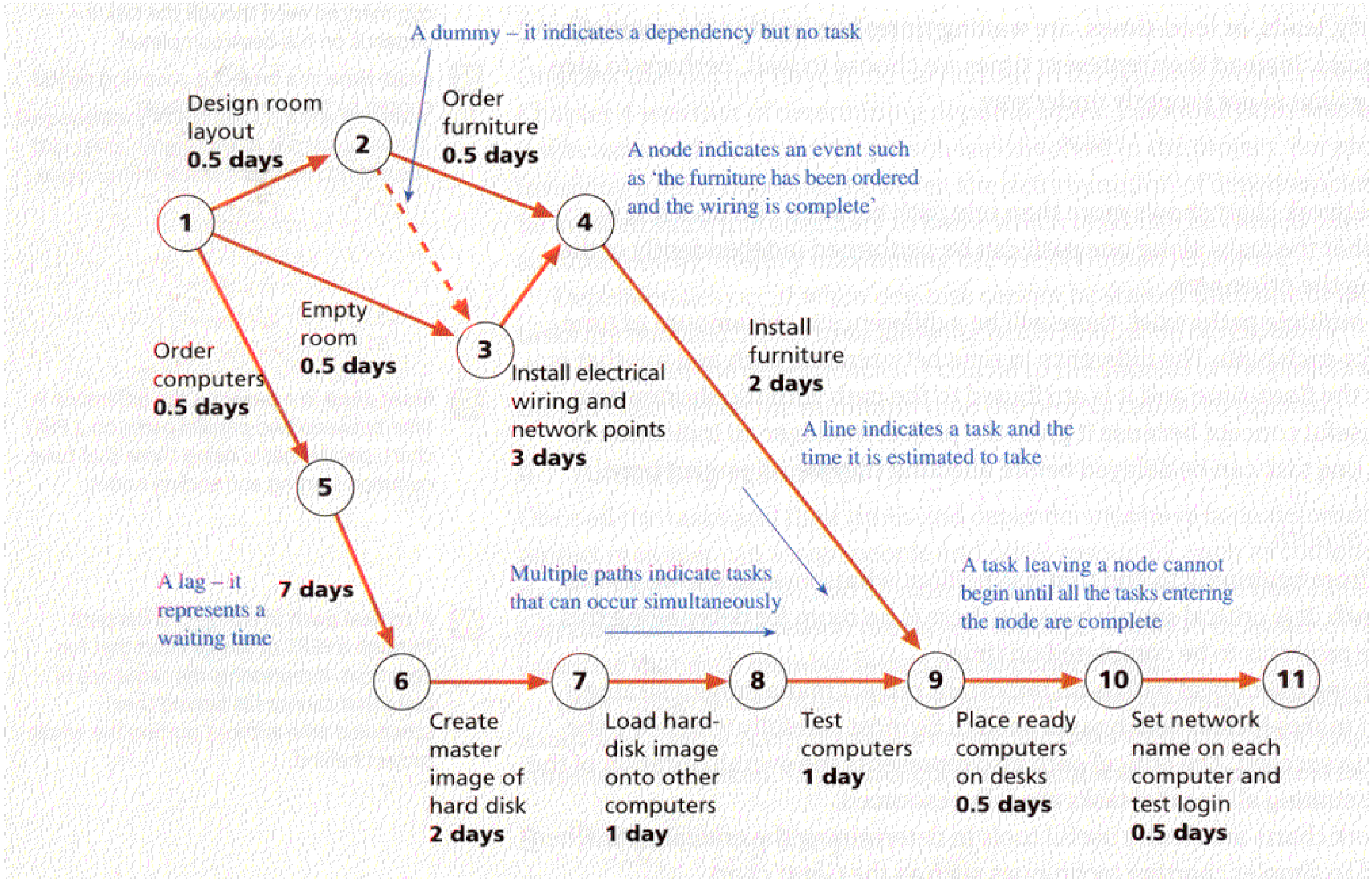
This Week

- Graph Terminology
- Graph Representation
- Graph Algorithms
  - Conceptual
- Big-O time complexity analysis

What is a Graph?

- A graph is a data structure representing connections between items
  - We're storing values with the potential for connections between any or all elements
- Examples of graphs in everyday life:
  - PERT charts
  - Flow charts
- Examples in computer science
  - Networks
  - Social Network diagrams
  - Executing a makefile

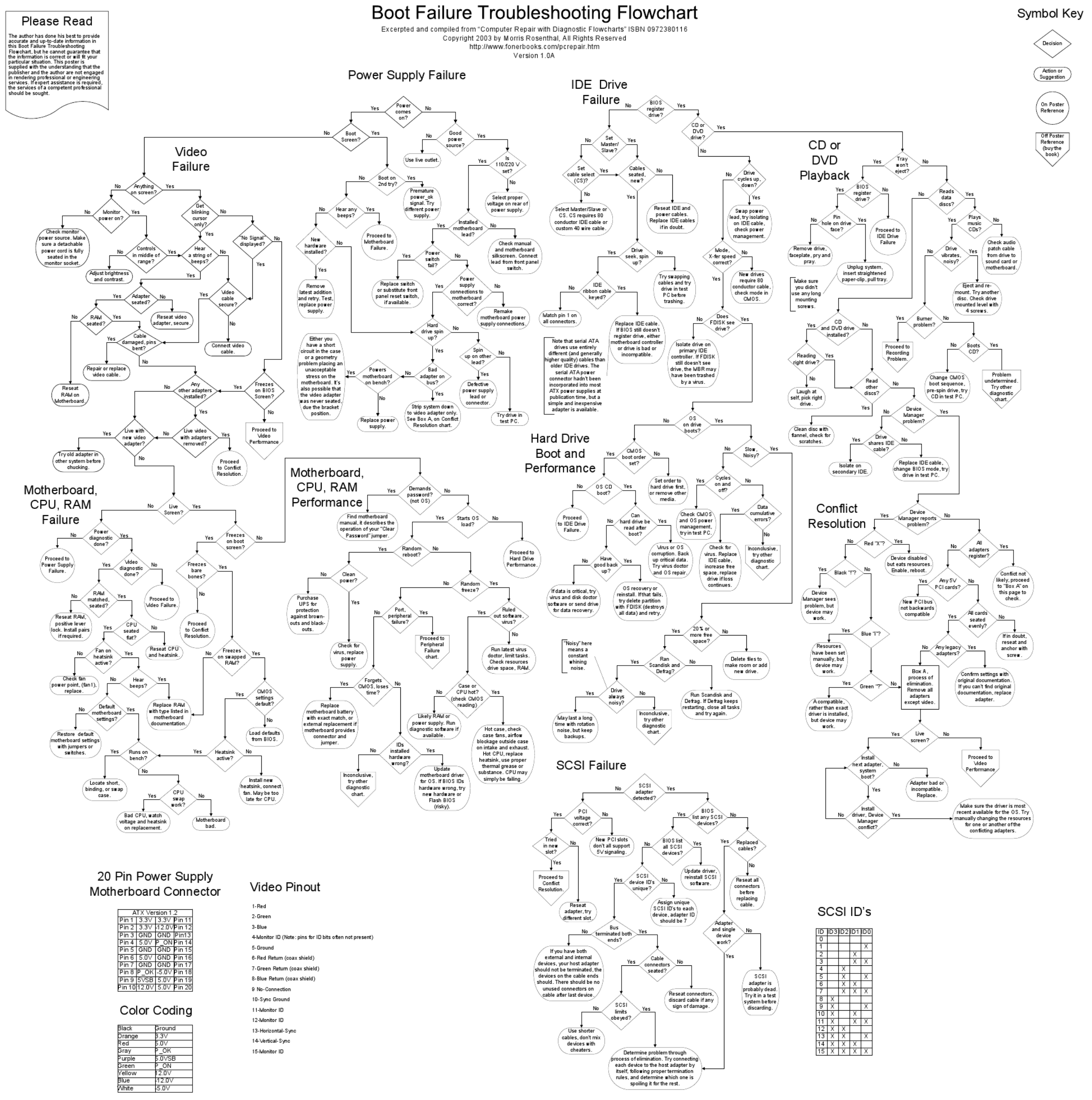
PERT Chart


A dummy – it indicates a dependency but no task
Design room layout 0.5 days
Order furniture 0.5 days
A node indicates an event such as 'the furniture has been ordered and the wiring is complete'
1
2
3
4
5
6
7
8
9
10
11
Empty room 0.5 days
Order computers 0.5 days
Install electrical wiring and network points 3 days
Install furniture 2 days
A line indicates a task and the time it is estimated to take
A lag – it represents a waiting time
7 days
Multiple paths indicate tasks that can occur simultaneously
A task leaving a node cannot begin until all the tasks entering the node are complete
Create master image of hard disk 2 days
Load hard-disk image onto other computers 1 day
Test computers 1 day
Place ready computers on desks 0.5 days
Set network name on each computer and test login 0.5 days

[8.19] A PERT chart shows tasks on the lines. These are also known as CPM and PERT/CPM charts.

http://it.ballaratsc.vic.edu.au/IT2/Theory%20topic%20notes/TheoryTopicJ-SubPagePertCharts.htm

Troubleshooting Flowcharts



Boot Failure Troubleshooting Flowchart

Excerpted and compiled from "Computer Repair with Diagnostic Flowcharts" ISBN 0972380116
Copyright 2003 by Morris Rosenthal, All Rights Reserved
http://www.fonerbooks.com/pcrepair.htm
Version 1.0A

Please Read

The author has done his best to provide accurate and up-to-date information in this Boot Failure Troubleshooting Flowchart, but he cannot guarantee that the information is correct or will fit your particular situation. This poster is supplied with the understanding that the publisher and the author are not engaged in rendering professional or engineering services. If expert assistance is required, the services of a competent professional should be sought.

20 Pin Power Supply Motherboard Connector

ATX Version 1.2			
Pin 1	3.3V	3.3V	Pin 11
Pin 2	3.3V	-12.0V	Pin 12
Pin 3	GND	GND	Pin13
Pin 4	5.0V	P_ON	Pin 14
Pin 5	GND	GND	Pin 15
Pin 6	5.0V	GND	Pin 16
Pin 7	GND	GND	Pin 17
Pin 8	P_OK	-5.0V	Pin 18
Pin 9	5VSB	5.0V	Pin 19
Pin 10	12.0V	5.0V	Pin 20

Color Coding

Black	Ground
Orange	3.3V
Red	5.0V
Gray	P_OK
Purple	5.0VSB
Green	P_ON
Yellow	12.0V
Blue	-12.0V
White	-5.0V

Video Pinout

1-Red
2-Green
3-Blue
4-Monitor ID (Note: pins for ID bits often not present)
5-Ground
6-Red Return (coax shield)
7-Green Return (coax shield)
8-Blue Return (coax shield)
9 No-Connection
10-Sync Ground
11-Monitor ID
12-Monitor ID
13-Horizontal-Sync
14-Vertical-Sync
15-Monitor ID

SCSI ID's

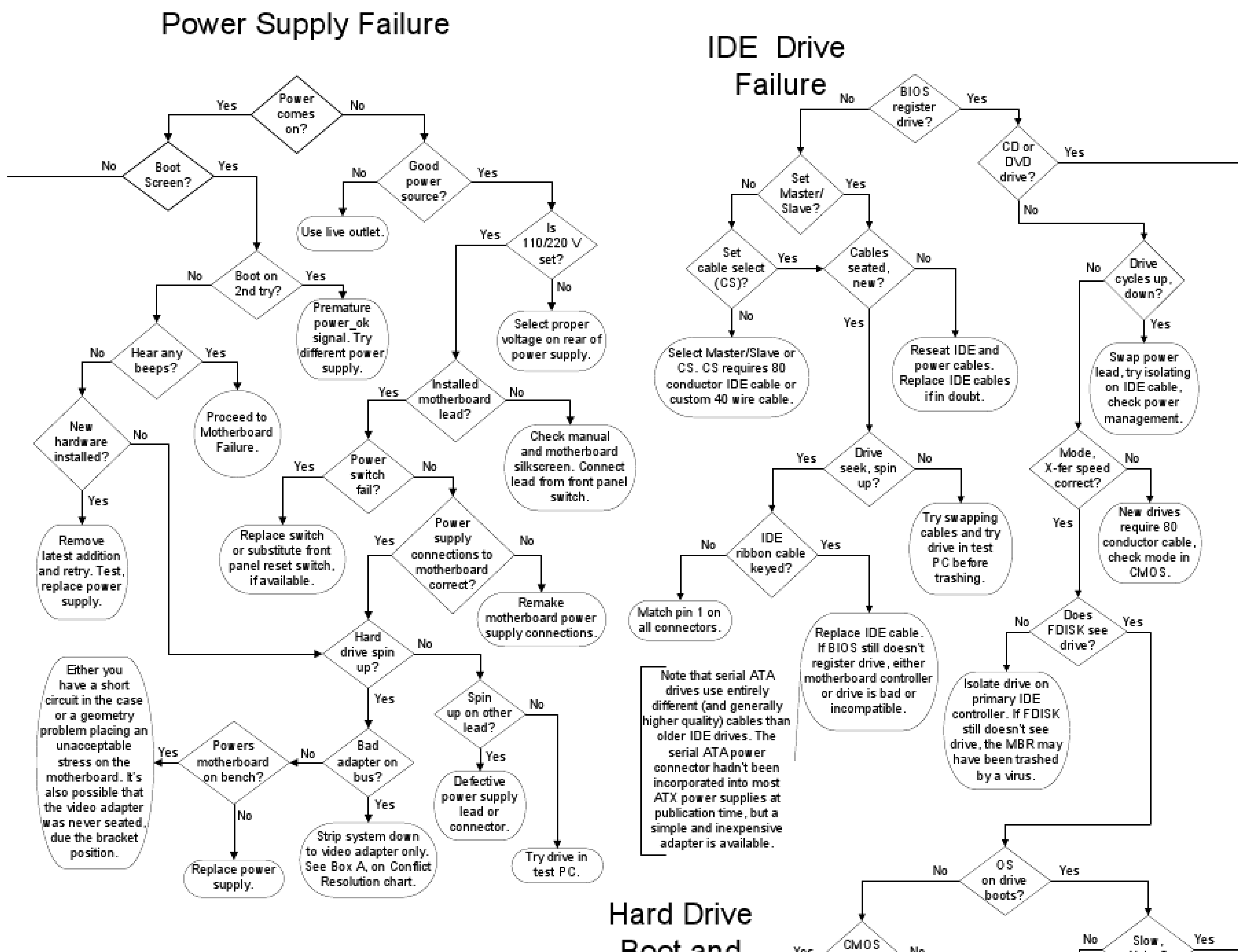
ID	ID3	ID2	ID1	ID0
0				
1				X
2			X	
3			X	X
4		X		
5		X		X
6		X	X	
7		X	X	X
8	X			
9	X			X
10	X		X	
11	X		X	X
12	X	X		
13	X	X		X
14	X	X	X	
15	X	X	X	X

http://www.fonerbooks.com/pcrepair.htm

Troubleshooting Flowcharts

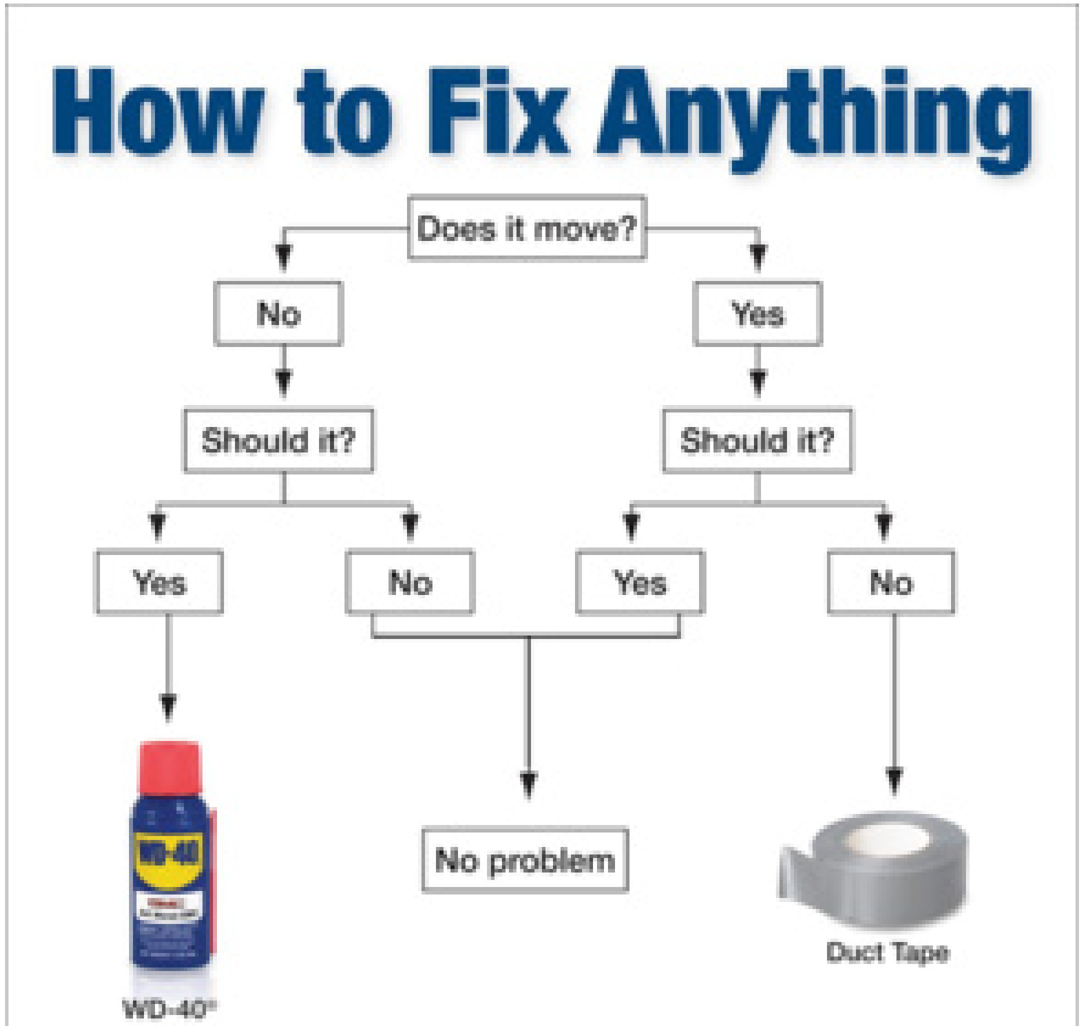
Boot Failure Troubleshooting Flowchart

Excerpted and compiled from “Computer Repair with Diagnostic Flowcharts” ISBN 0972380116
Copyright 2003 by Morris Rosenthal, All Rights Reserved
http://www.fonerbooks.com/pcrepair.htm
Version 1.0A

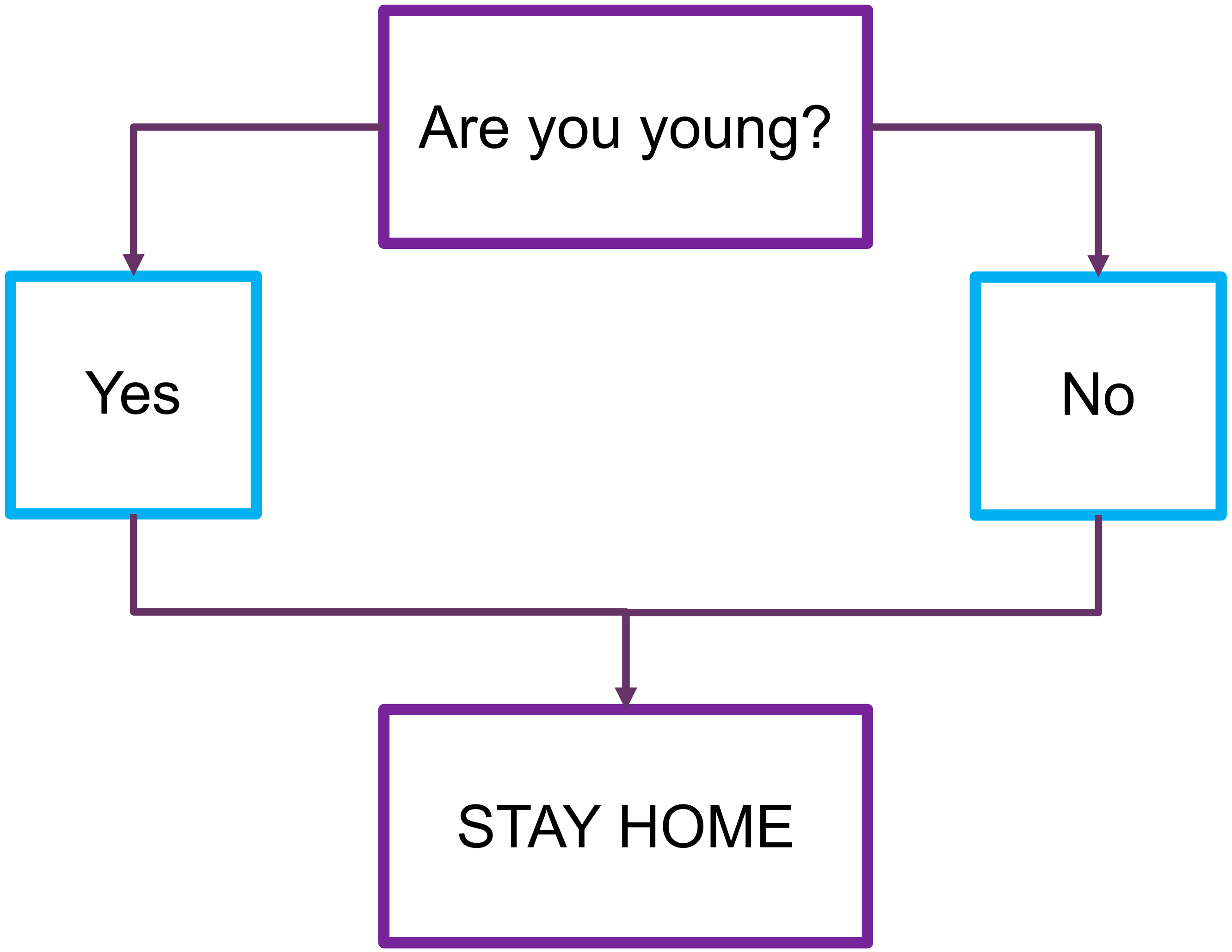

Power Supply Failure
Power comes on?
Yes
No
Boot Screen?
Good power source?
Use live outlet.
Is 110/220 V set?
Select proper voltage on rear of power supply.
Boot on 2nd try?
Premature power_ok signal. Try different power supply.
Hear any beeps?
Proceed to Motherboard Failure.
New hardware installed?
Remove latest addition and retry. Test, replace power supply.
Installed motherboard lead?
Check manual and motherboard silkscreen. Connect lead from front panel switch.
Power switch fail?
Replace switch or substitute front panel reset switch, if available.
Power supply connections to motherboard correct?
Remake motherboard power supply connections.
Hard drive spin up?
Spin up on other lead?
Defective power supply lead or connector.
Try drive in test PC.
Bad adapter on bus?
Strip system down to video adapter only. See Box A, on Conflict Resolution chart.
Powers motherboard on bench?
Replace power supply.
Either you have a short circuit in the case or a geometry problem placing an unacceptable stress on the motherboard. It's also possible that the video adapter was never seated, due the bracket position.
IDE Drive Failure
BIOS register drive?
CD or DVD drive?
Set Master/ Slave?
Set cable select (CS)?
Select Master/Slave or CS. CS requires 80 conductor IDE cable or custom 40 wire cable.
Cables seated, new?
Reseat IDE and power cables. Replace IDE cables if in doubt.
Drive seek, spin up?
Try swapping cables and try drive in test PC before trashing.
IDE ribbon cable keyed?
Match pin 1 on all connectors.
Replace IDE cable. If BIOS still doesn't register drive, either motherboard controller or drive is bad or incompatible.
Note that serial ATA drives use entirely different (and generally higher quality) cables than older IDE drives. The serial ATA power connector hadn't been incorporated into most ATX power supplies at publication time, but a simple and inexpensive adapter is available.
Drive cycles up, down?
Swap power lead, try isolating on IDE cable, check power management.
Mode, X-fer speed correct?
New drives require 80 conductor cable, check mode in CMOS.
Does FDISK see drive?
Isolate drive on primary IDE controller. If FDISK still doesn't see drive, the MBR may have been trashed by a virus.
OS on drive boots?
CMOS
Slow,
Hard Drive Boot and

http://www.fonerbooks.com/pcrepair.htm

Troubleshooting Flowcharts

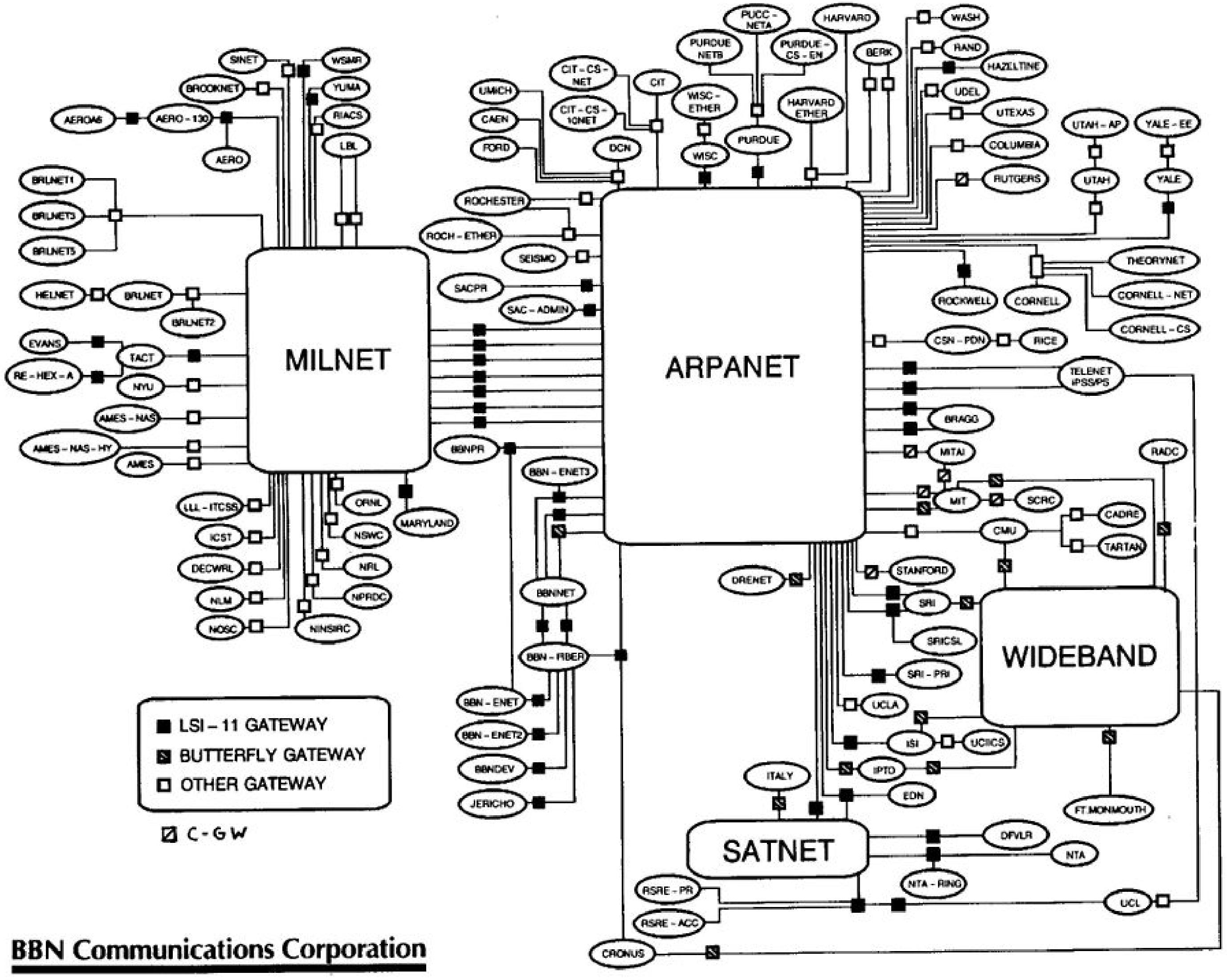


COVID-19



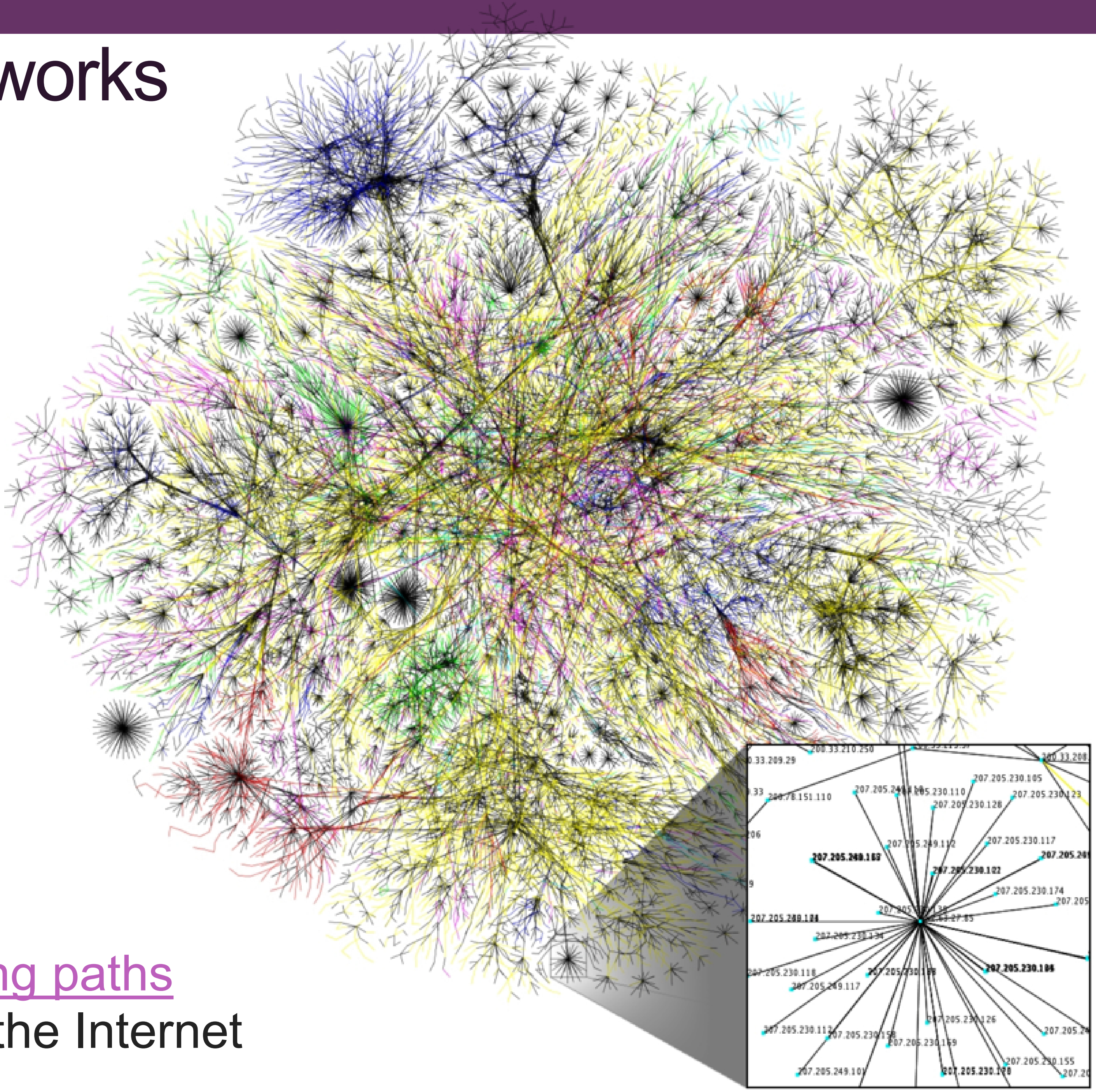
Computer Networks

TCP/IP Internet map of early 1986.



Public Domain, https://commons.wikimedia.org/w/index.php?curid=47069299

Computer Networks



An Opte Project visualization of routing paths through a portion of the Internet

By The Opte Project - Originally from the English Wikipedia; description page is/was here., CC BY 2.5, https://commons.wikimedia.org/w/index.php?curid=25698718

Social Networks

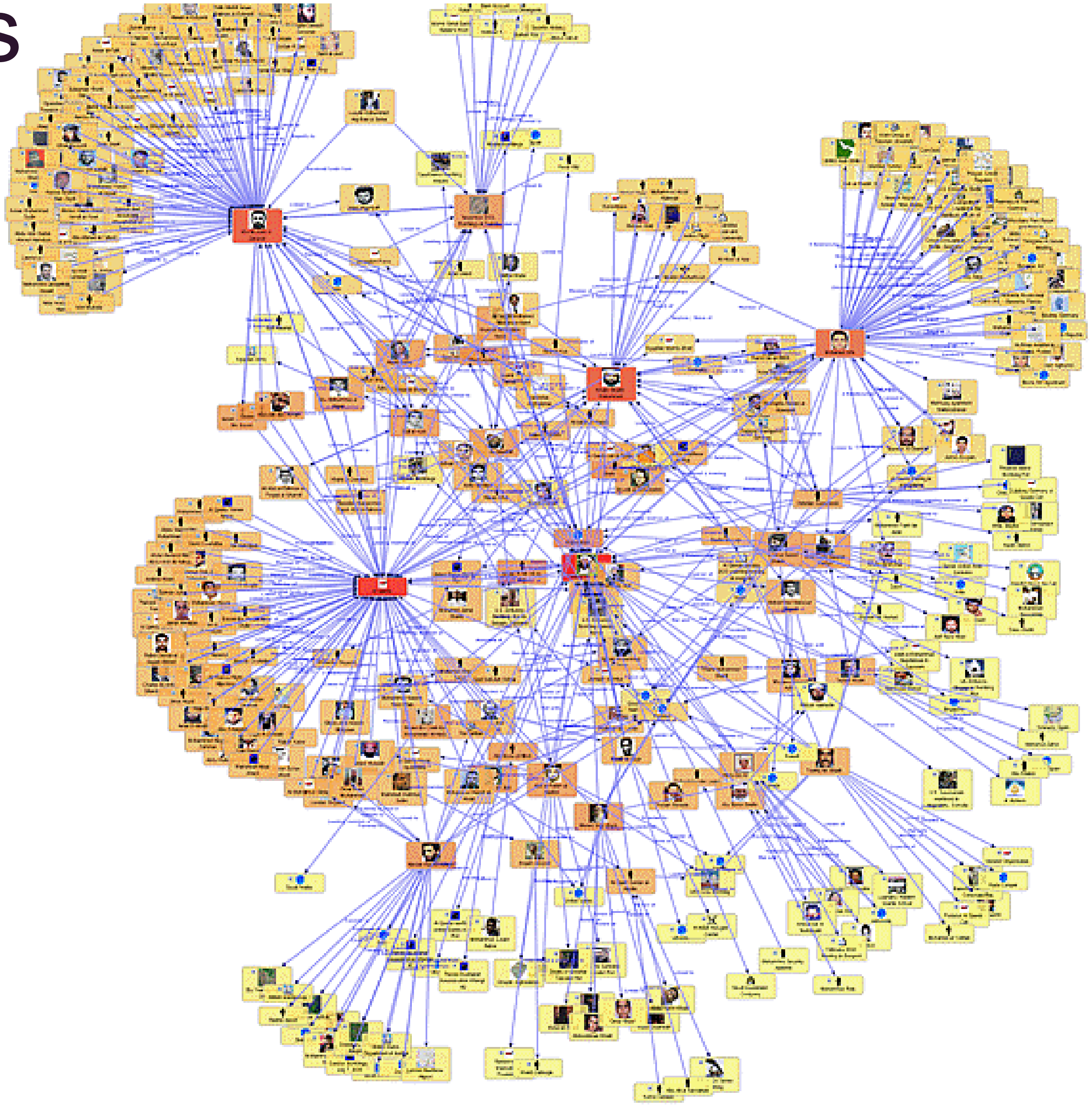
Using Social Network Analysis, you can answer:

•How highly connected is an entity within a network?

•What is an entity's overall importance in a network?

•How central is an entity within a network?

•How does information flow within a network?

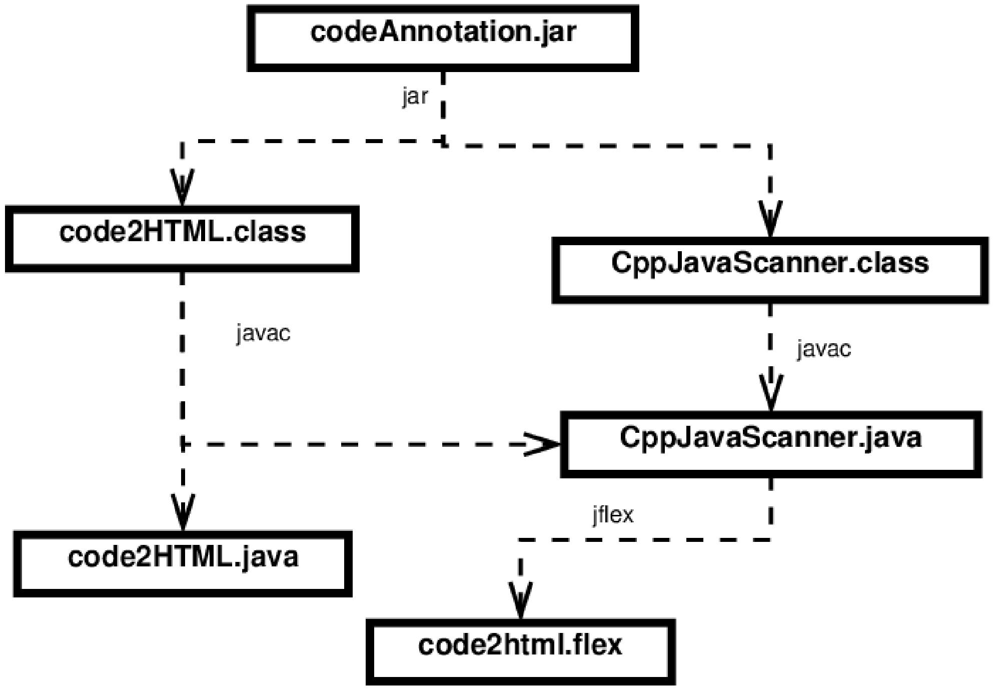


Sentinel Visualizer - http://www.fmsasg.com/SocialNetworkAnalysis/

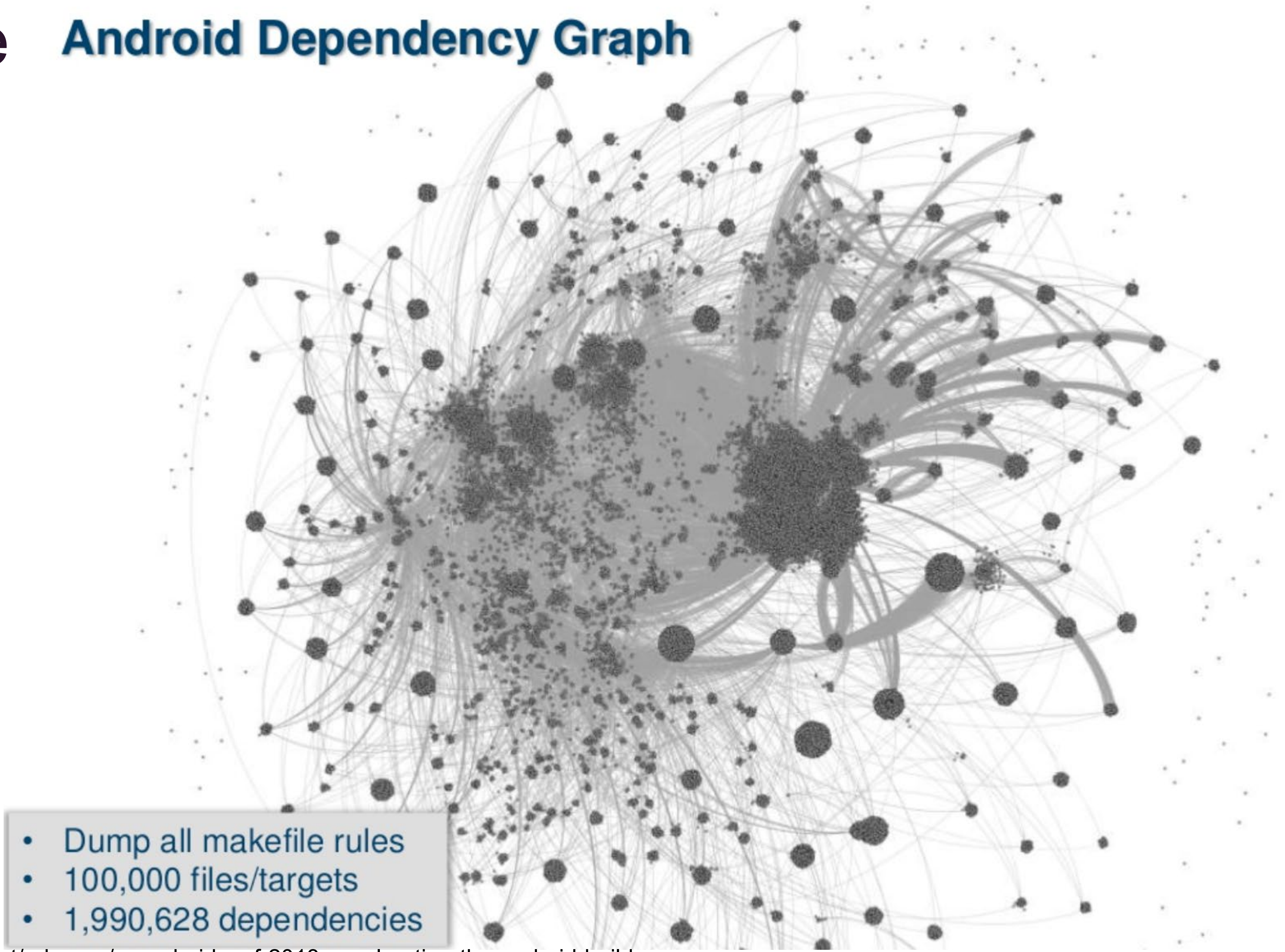
Makefiles

How make Works

- Construct the dependency graph from the target and dependency entries in the makefile
- Do a topological sort to determine an order in which to construct targets.
- For each target visited, invoke the commands if the target file does not exist or if any dependency file is newer
    - Relies on file modification dates



Makefile Rules



Android Dependency Graph

- Dump all makefile rules
- 100,000 files/targets
- 1,990,628 dependencies

https://www.slideshare.net/adrosen/onandroidconf-2013-accelerating-the-android-build

Set Theory

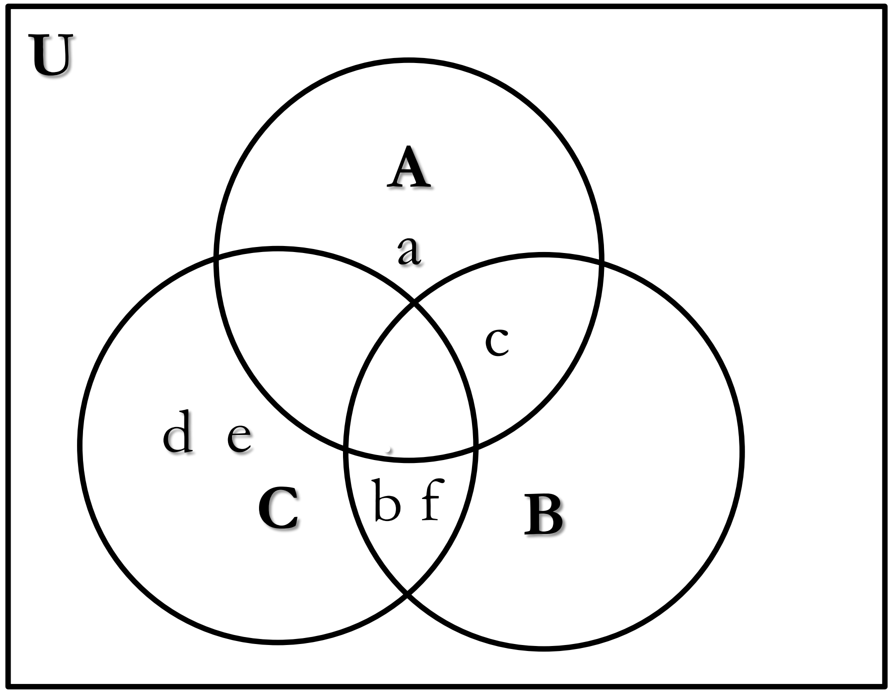
- A set is any collection of objects, e.g. a set of vertices
- The objects in a set are called the **elements** of the set

- Repetition and order are not important
  - {2, 3, 5} = {5, 2, 3} = {5, 2, 3, 2, 2, 3}

- Sets can be written in predicate form:
  - {1, 2, 3, 4} = {x **:** x is a positive integer less than 5}
  - Read the colon as "such that", also {x|x is a….}
- The empty set is {} = ∅, all empty sets are equal

Set Theory based on
http://www.maths.manchester.ac.uk/~avb/0n1_pdf/0N1_All.pdf

Elements and Subsets



- $x \in A$ means "x is an element of A"
  - $x \notin A$ means "x is not an element of A"
- $A \subseteq B$ means "A is a subset of B"
  - $x \not\subseteq A$ means "x is not a subset of A"
- If $A \subseteq B$ and $A \neq B$…
  A is a proper subset of B and we write $A \subset B$
- $A \subseteq A$ for every set A
  - Every set is a subset of itself

Subsets

- If A ⊆ B and B ⊆ C then A ⊆ C
- If A ⊆ B and B ⊆ A then A = B

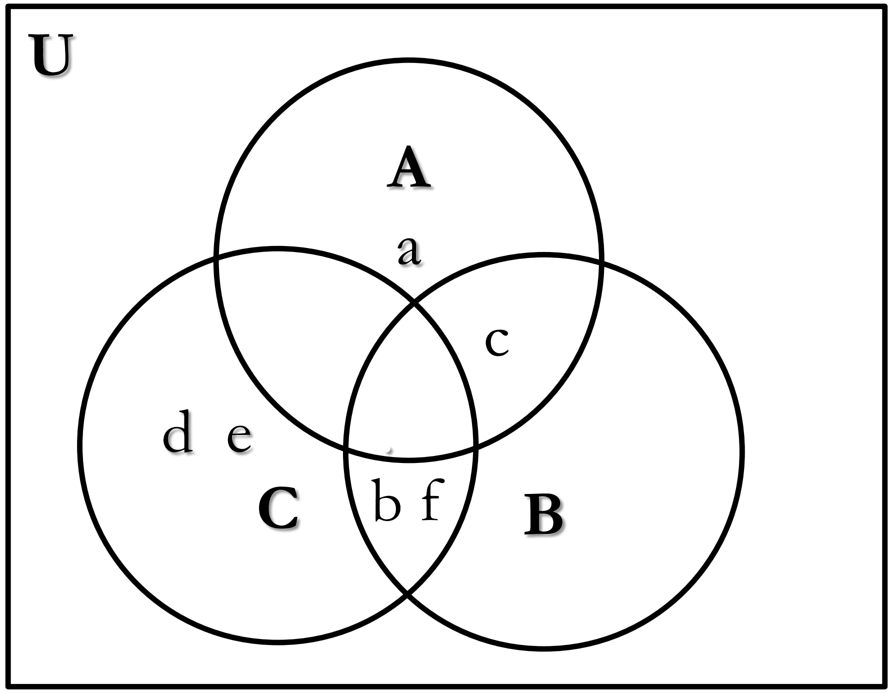
- The empty set is a subset of every set:

  ∅ ⊆ A for any A

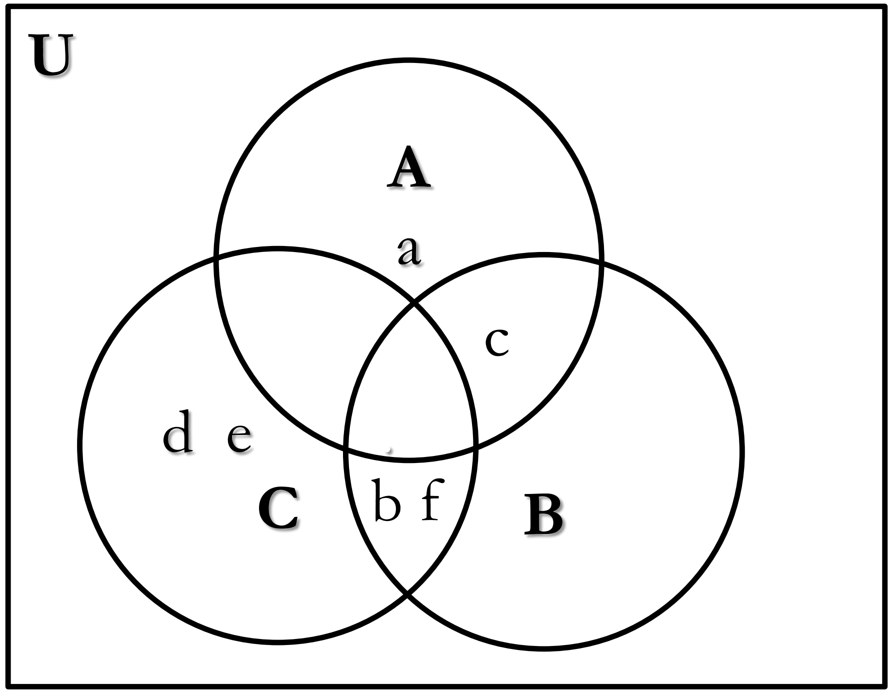
- The subsets of A ={1, 2, 3} are:

∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}

(Sometimes called the *powerset* of A, **P**(A))



Operations on Sets

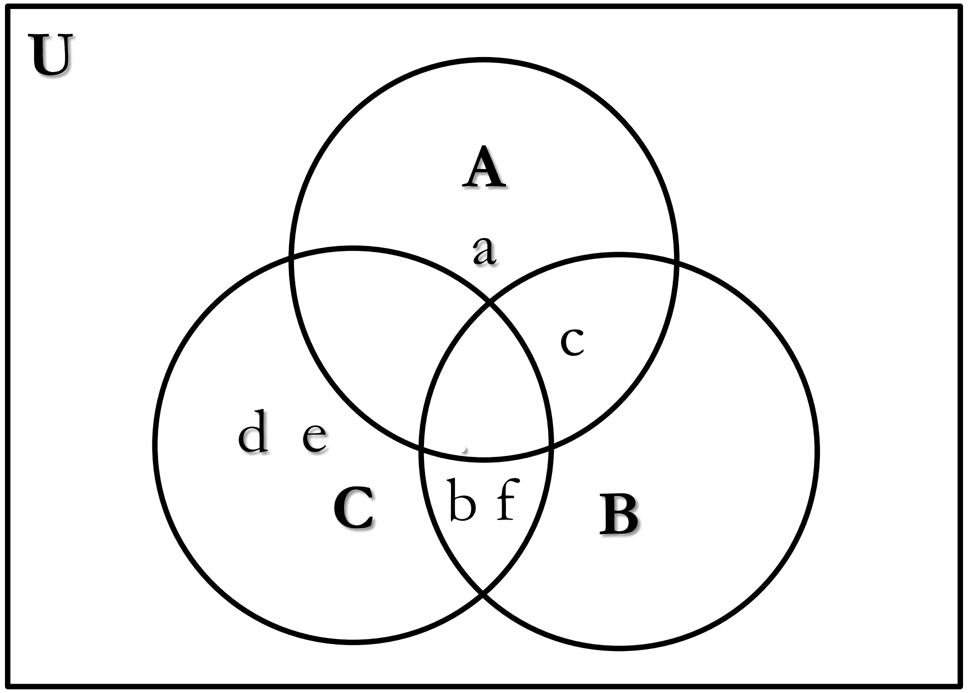


- Union
  - $A \cup B = \{ x : x \in A \text{ or } x \in B \}$
- Intersection
  - $A \cap B = \{ x : x \in A \text{ and } x \in B \}$
- Universal Set
  - All sets under consideration will be subsets of a background set, called the Universal Set, U
- Complement
  - $A' = \{ x : x \in U \text{ and } x \notin A \}$

Example

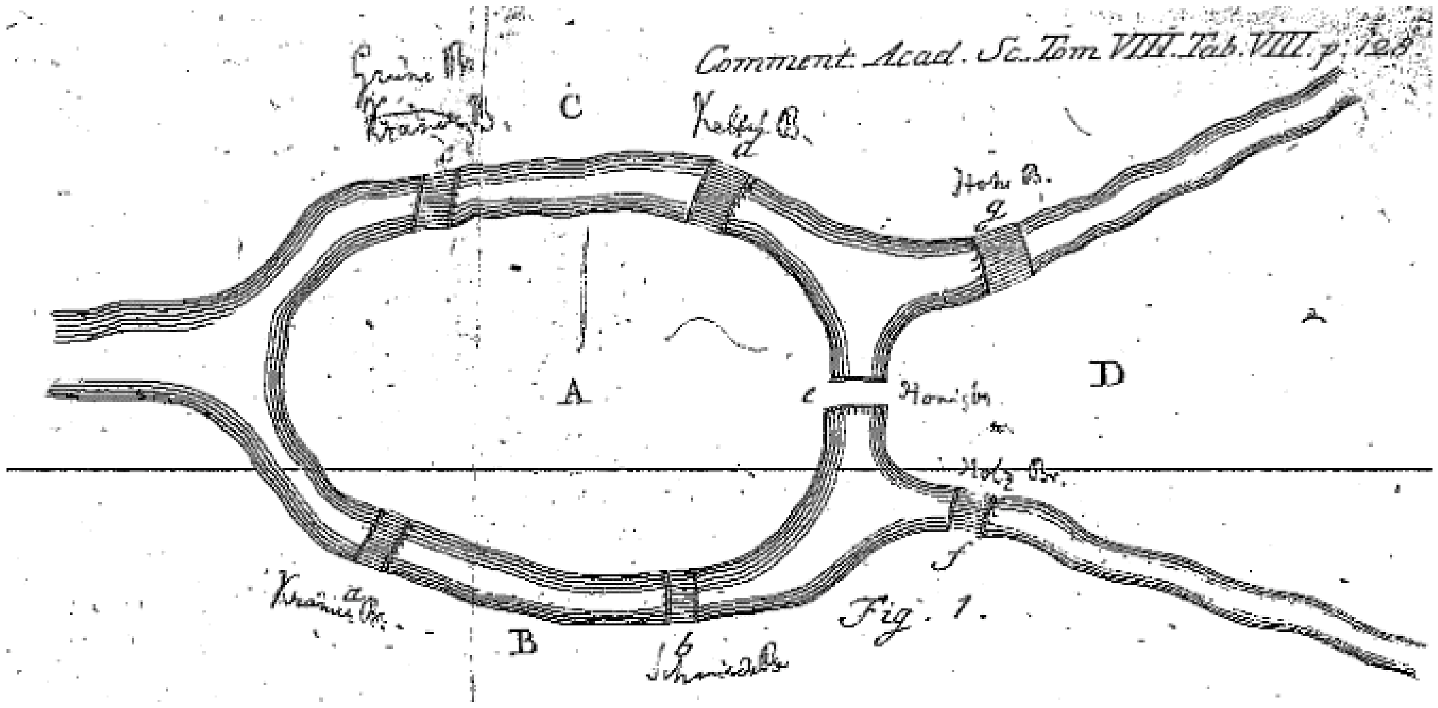
- Let:
  - U = {a, b, c, d, e, f}
  - A = {a, c}
  - B = {b, c, f}
  - C = {b, d, e, f}

- Then:
  - B ∪ C = {b, c, d, e, f}
  - A ∩ (B ∪ C) = {c}
  - A' = {b, d, e, f}
    = C
  - A' ∩ (B ∪ C) = C ∩ (B ∪ C) = {b, d, e, f} = C



Graph Theory - History

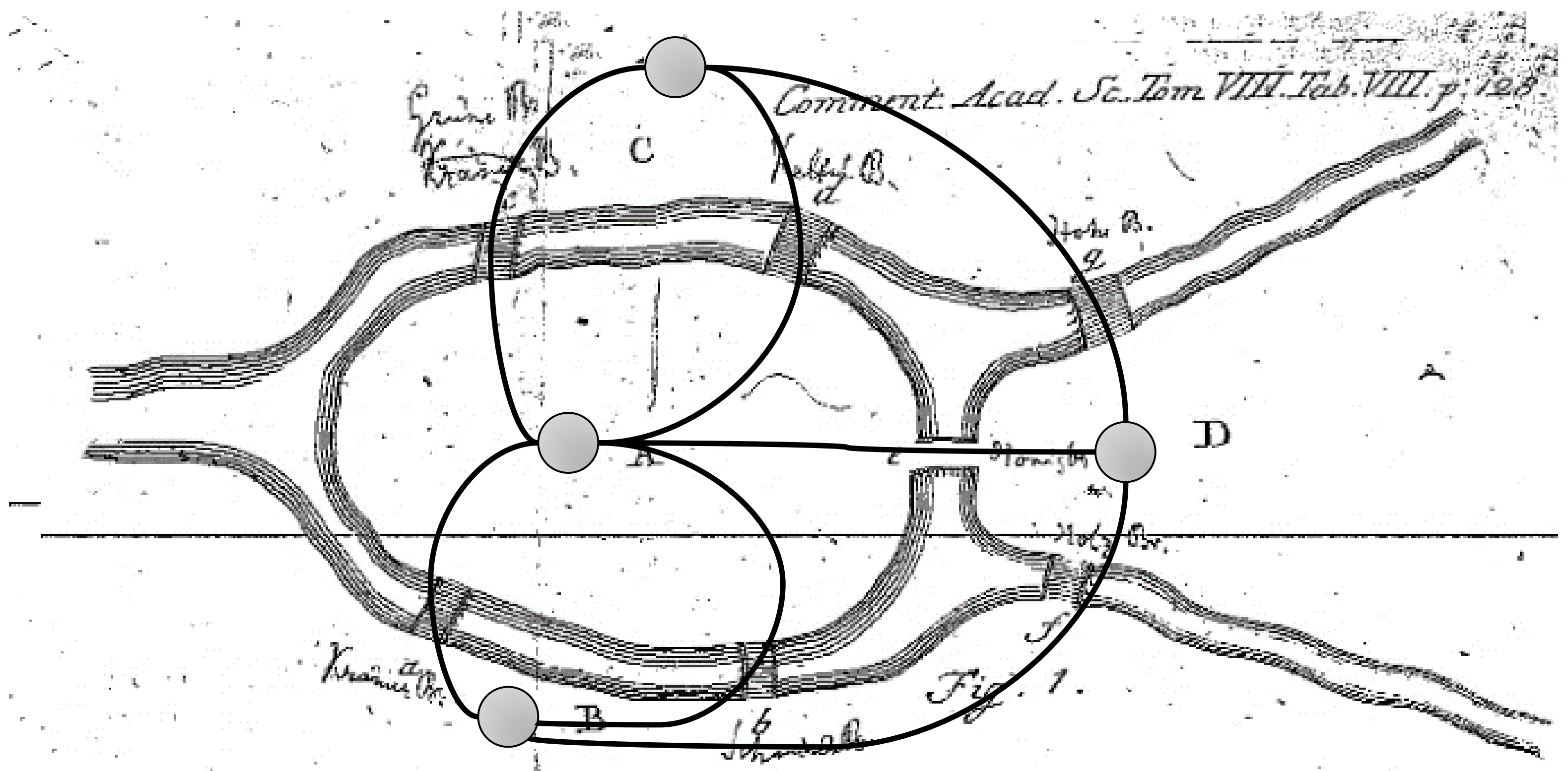
- The town of Konigsberg (now Kaliningrad) is on the banks of the Pregel River, and two islands connected by 7 bridges
- The locals puzzled over whether they could walk across all of the bridges exactly once and return to their starting point



Euler's Figure 1 from 'Solutio problematis ad geometriam situs pertinentis,' Eneström 53 [source: MAA Euler Archive]

Graph Theory - History

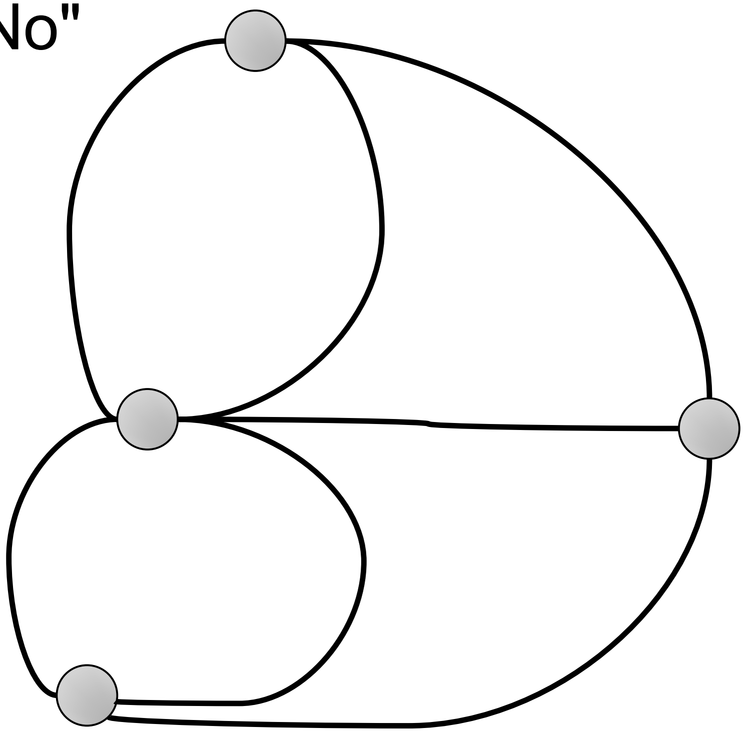
- In 1735, Leonhard Euler (1707-1783) published a paper with a solution to the problem – **proving** it was not possible
- He abstracted the problem into what became graph theory, based on an idea by Leibniz (1646-1716)



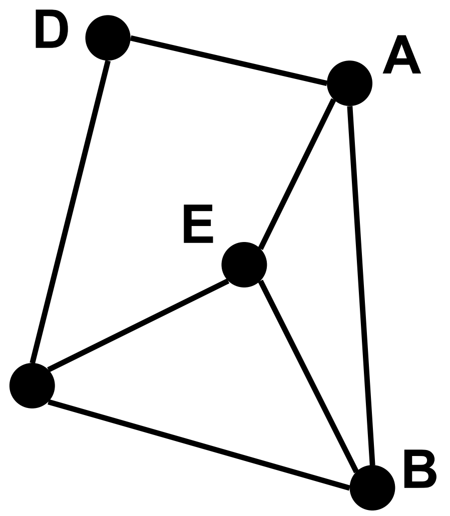
Euler's Figure 1 from 'Solutio problematis ad geometriam situs pertinentis,' Eneström 53 [source: MAA Euler Archive]

Graph Theory - History

– The solution:
  - "If a connected graph has more than two vertices of odd degree, then it cannot contain an Eulerian path", proved by Euler
  - "If a connected graph has no vertices of odd degree, or two such vertices, then it contains an Eulerian path", Carl Hierholzer, 1873
  - The answer is "No"

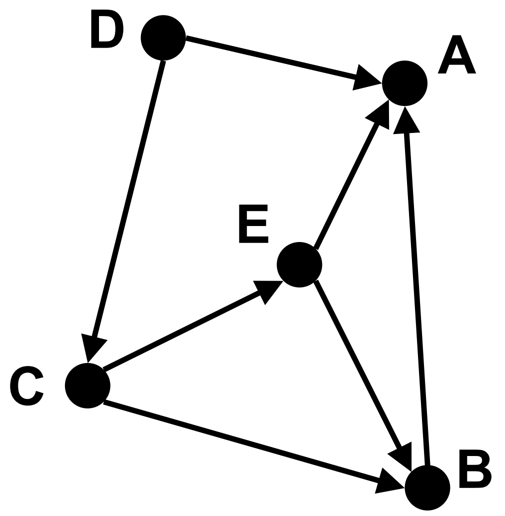


Graph Terminology



- Vertex: A labelled **node** in the graph
  - A, B, C, D, E
- Edge: An arc joining two vertices
  - {A, B}, {A, D}, {A, E}, {B, C}, {B, E},{C, D}, {C, E}
  - Edges can have **weight**, **direction** and **labels**
  - The two vertices in an edge are the **endpoints**
- Graph: made up of a set **V** of vertices and a set **E** of edges
  - V = {A, B, C, D, E}
  - E = {{A, B}, {A, D}, {A, E}, {B, C}, {B, E}, {C, D}, {C, E}}

Graph Terminology - Direction

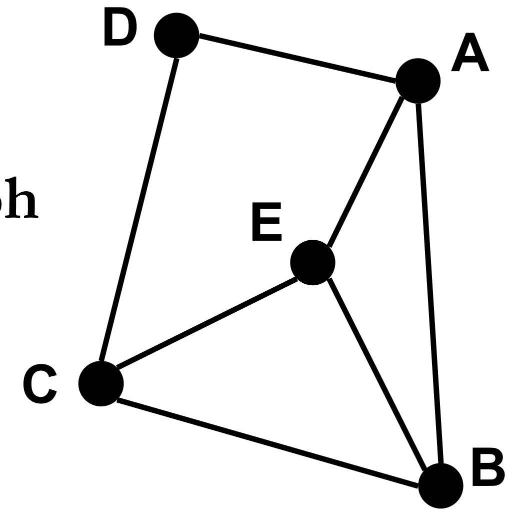


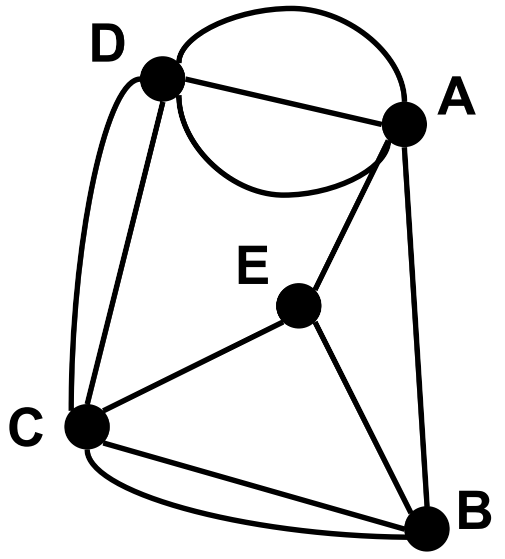
- A graph can be **directed** or **undirected**
- Undirected edges in the previous slide were:
  - {A, B}, {A, D}, {A, E}, {B, C}, {B, E} {C, D}, {C, E}
- The directed edge linking A and B is (B,A)
  - Order is important
  - B is the **source**, A is the **sink**
- Directed edges in this graph are:
  - E = {(D,A), (D,C), (B,A), (C,B), (C,E), (E,A), (E,B)}

Multigraphs

- Multigraphs allow multiple edges between the same pair of vertices
- Graphs that are not multigraphs are referred to as simple graphs

Simple Graph

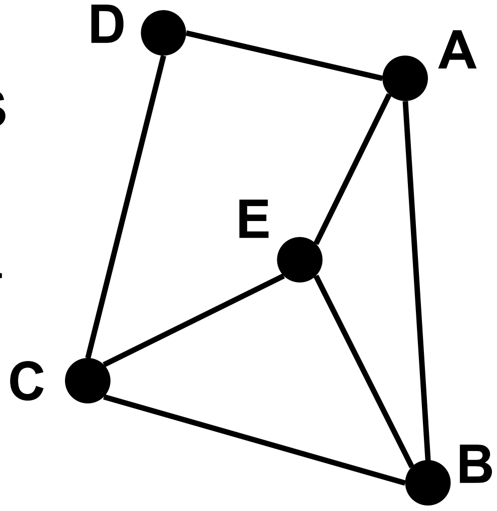




Multigraph

Graph Size

- Given the graph with V vertices and E edges
  - V = {A, B, C, D, E}
  - E = {{A, B},{A, D},{A, E},{B, C},{B, E},{C, D},{C, E}}

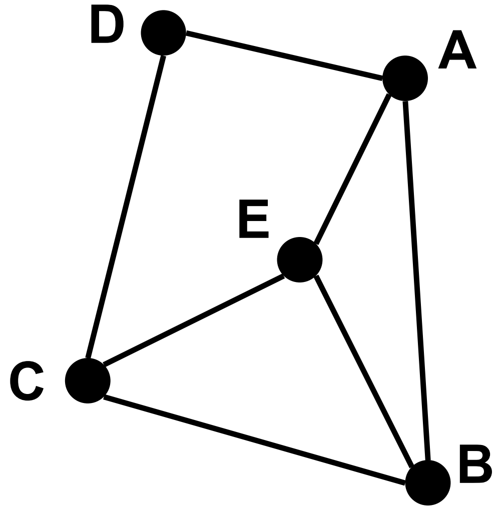


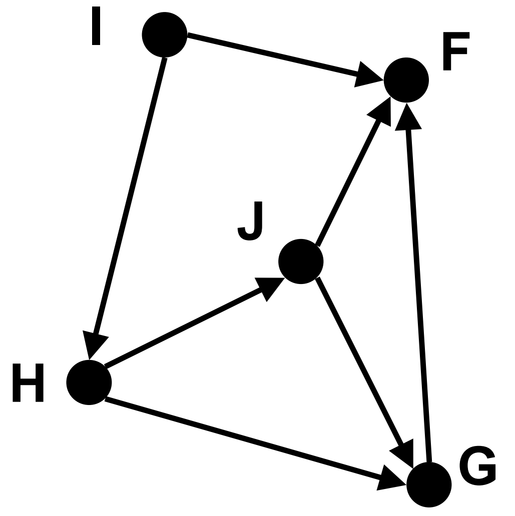
- **|V|** = size of V, often denoted **n**
- **|E|** = size of E, often denoted **m**
  - |V|= 5
  - |E|= 7

- Maximum number of edges in a simple graph is (n(n-1))/2 i.e O(n^2)

Vertex Adjacency and Degree

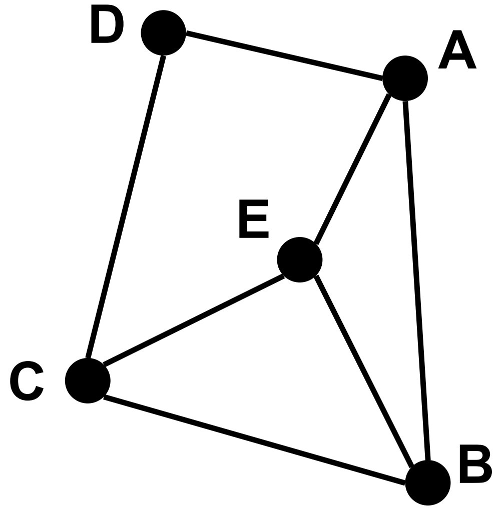
- Two vertices are **adjacent** if they are connected by an edge
  - D is adjacent to A
- The **degree** of a vertex, d(*v*), is the number of edges for which it is an endpoint
  - d(D) =2, d(A) = 3





- If the graph is directed…
  - **Outdegree** of a vertex: the number of edges for which it is the **source**
  - **Indegree** of a vertex: the number of edges for which it is the **sink**
    - ideg(I)=0, odeg(I) = 2, d(I) = 2
    - ideg(F) = 3, odeg(F) = 0, d(F) = 3

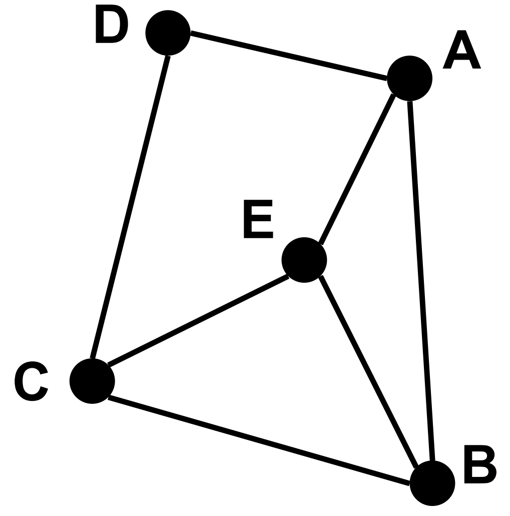
Paths and Cycles



- A **path** from vertex v_1 to v_2 is a sequence of vertices $v_1, v_2, \ldots v_k$, that are connected by edges $(v_1, v_2), (v_2, v_3)\ldots(v_{k-1}, v_k)$
  - Path from D to E: (D, A, B, E)
  - Edges in the path: (D,A), (A,B), (B,E)
- A path is **simple** if each vertex appears only once
- Vertex *u* is **reachable** from *v* if there is a path for *u* to *v*

- A **circuit** is a path whose first and last vertices are the same

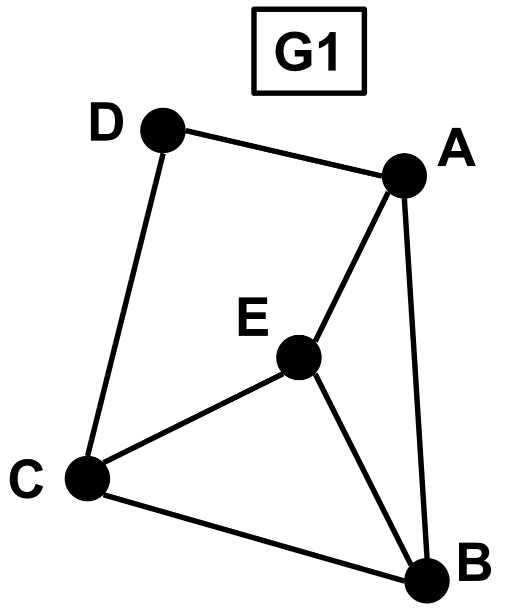
Paths and Cycles

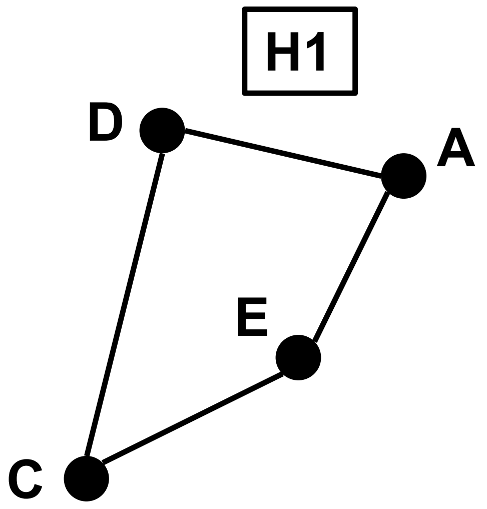


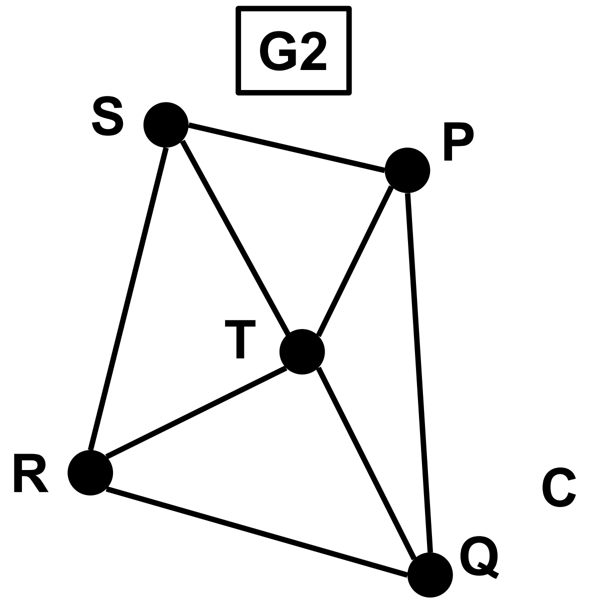
- A simple circuit is a **cycle** if, except for the first (and last) vertex, no other vertex appears twice
  - e.g. (A,B,E,A) or (D,A,B,E,C,D)
- A graph is **cyclic** if it has some path that contains the same node twice
  - Otherwise **acyclic or non-cyclic**
  - Trees and linked lists are acyclic
- A **Hamiltonian cycle** of graph G is a cycle that contains all the vertices of G:
  - (D,A,B,E,C,D)

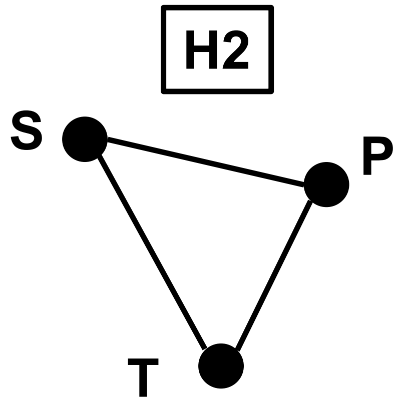
Subgraphs

- A **subgraph** of a graph G = (V, E) is a graph H = (U, F) such that U ⊆ V and F ⊆ E.
  - H1={ [U1: A, E, C, D], [F1: (A, E),(E, C),(C, D),(D, A)] } is subgraph of G1
  - H2={ [U2:S, P, T], [F2: (S, P),(S, T),(T, P)] } is a subgraph of G2A



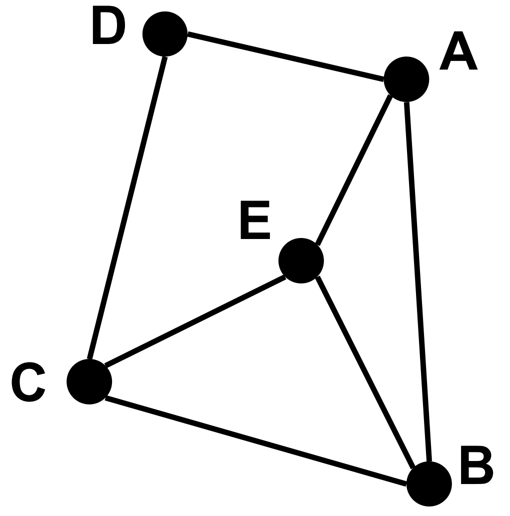


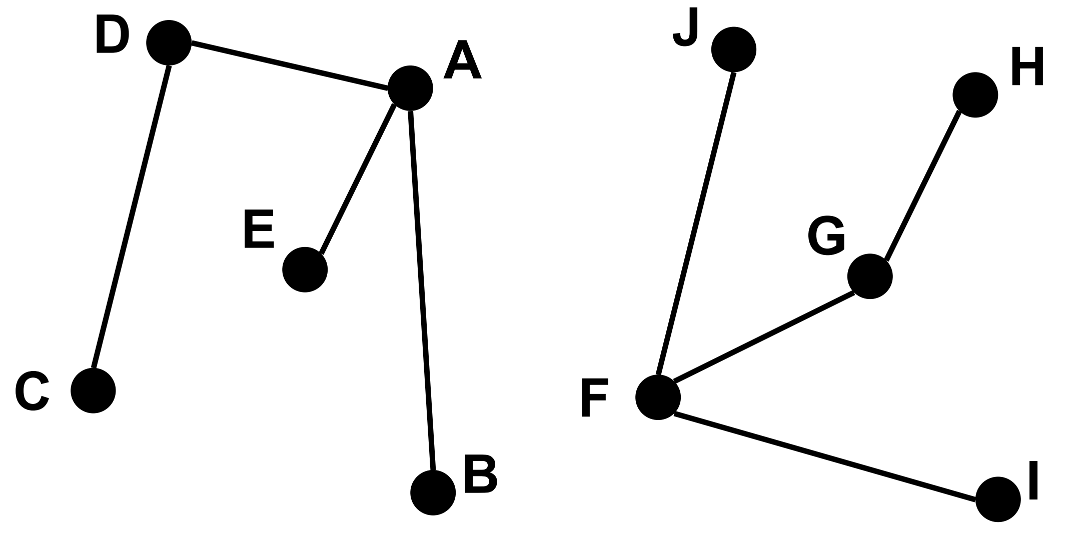




Graph Connectivity

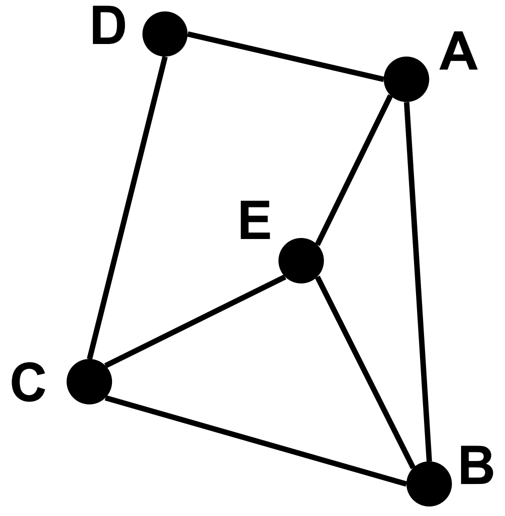
- A graph is said to be **connected** if there is a path from any vertex to any other vertex in the graph
- A **forest** is a graph that does not contain a cycle
- A **tree** is a **connected** forest
- A **spanning forest** of an undirected graph G is a subgraph of G that is a **forest** and contains all the vertices of G.

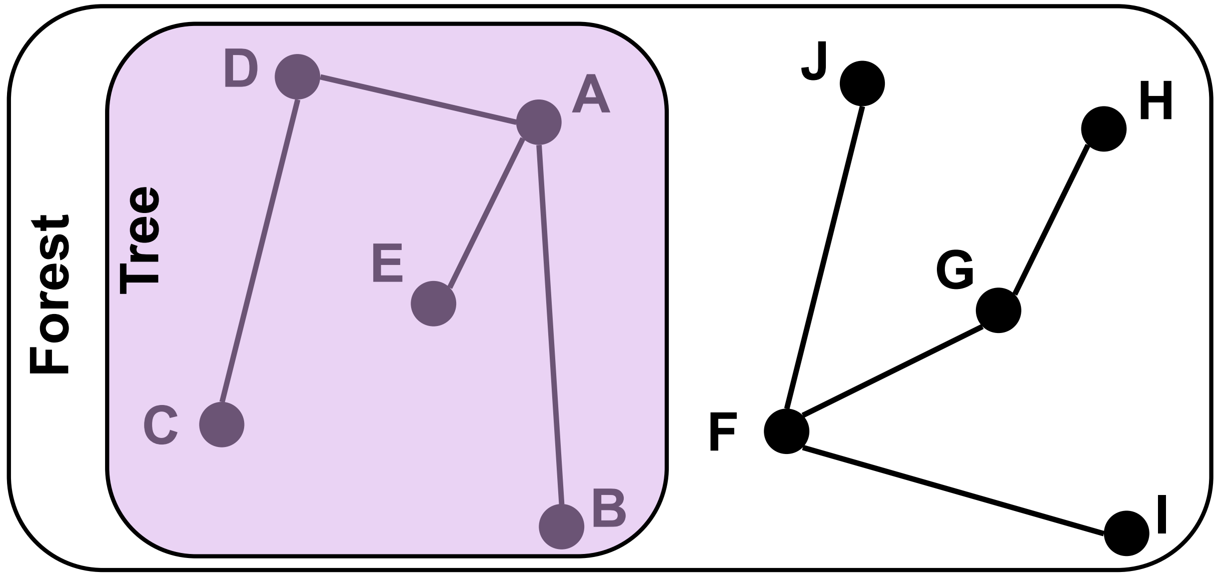




Graph Connectivity

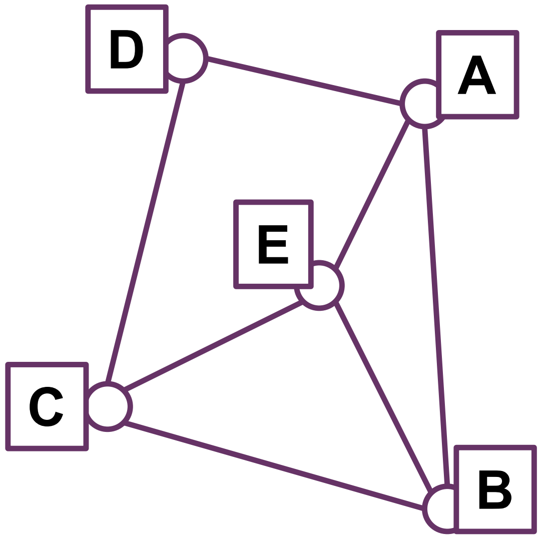
- A graph is said to be **connected** if there is a path from any vertex to any other vertex in the graph
- A **forest** is a graph that does not contain a cycle
- A **tree** is a **connected** forest
- A **spanning forest** of an undirected graph G is a subgraph of G that is a **forest** and contains all the vertices of G.

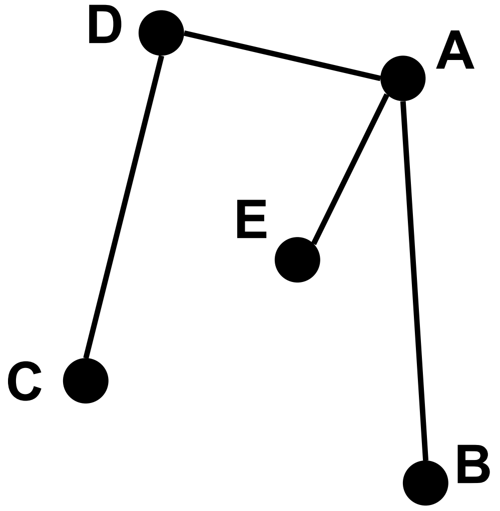


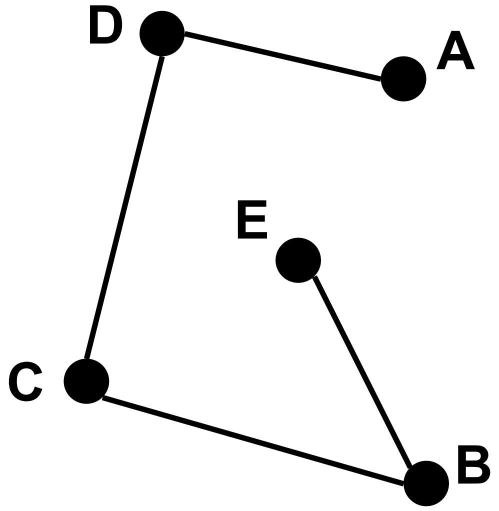


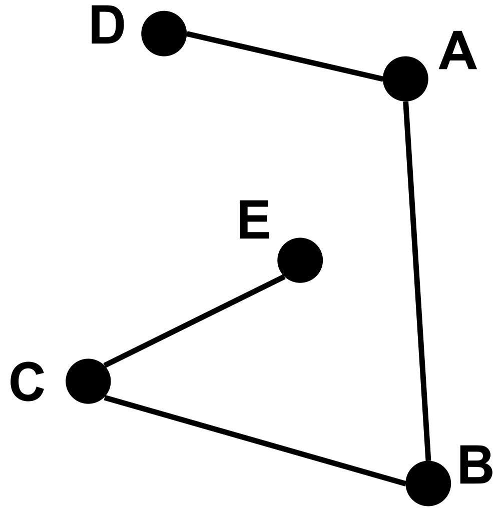
Spanning Trees

- A **spanning tree** of a graph G is a subgraph of G that is a tree and contains all the vertices of G
- Some spanning trees of this graph → are below:



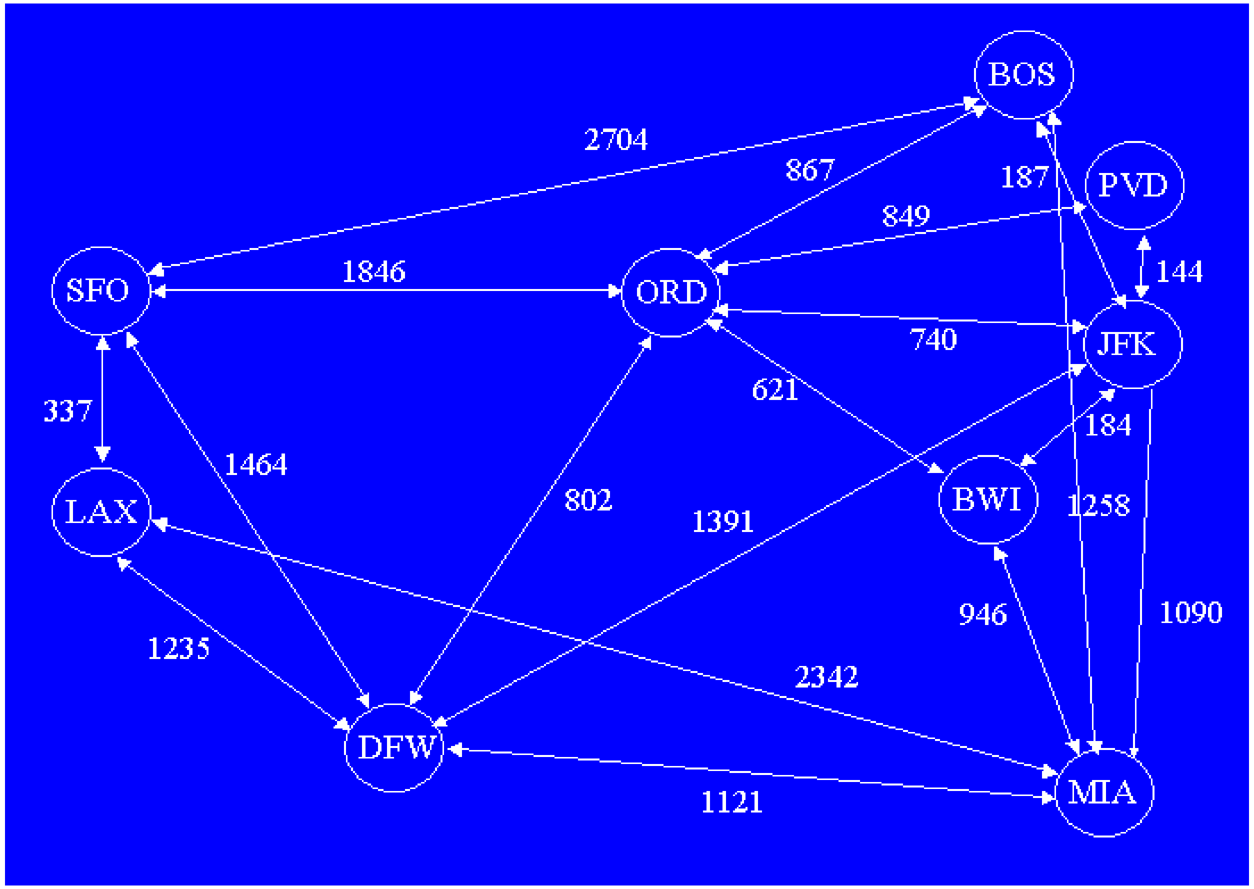






Example: Airport distances

- Nodes represent airports, edges represent flights
- Edge values represent the distance for each flight
- Could also do cost or time



Goodrich and Tamassia, "Data Structures and Algorithms in Java"

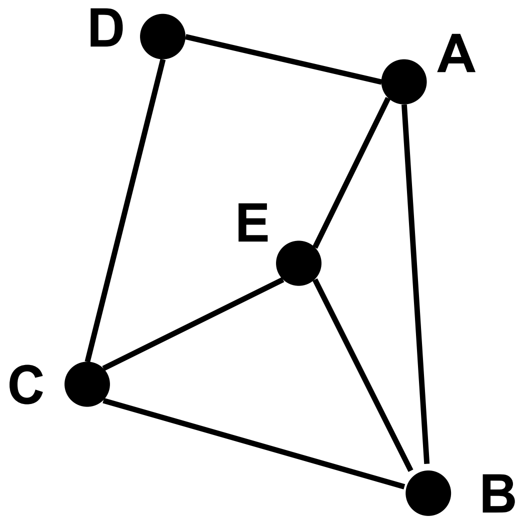
GRAPH REPRESENTATION AND IMPLEMENTATION

Adjacency Matrix

Adjacency List (options: links, nodes, edges)

Graph Representations

- Two main approaches:
  - Adjacency Lists
  - Adjacency Matrices
- Undirected Graphs – examples:



Adjacency List

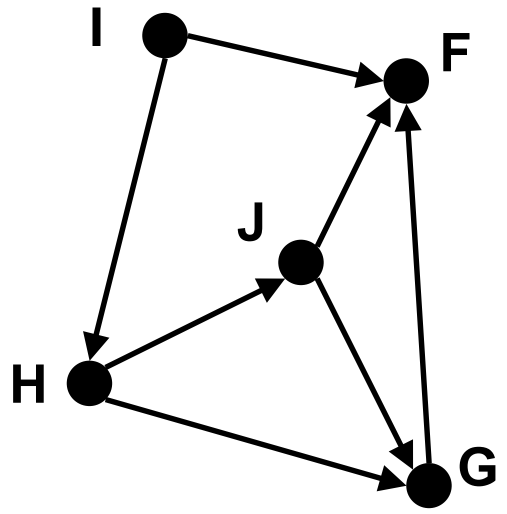
A	B	D	E
B	A	C	E
C	B	D	E
D	A	C	
E	A	B	C

Adjacency Matrix

	A	**B**	**C**	**D**	**E**
A	0	1	0	1	1
B	1	0	1	0	1
C	0	1	0	1	1
D	1	0	1	0	0
E	1	1	1	0	0

Graph Representations

- Directed Graphs - examples:



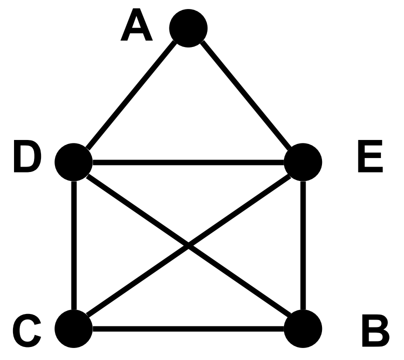
Adjacency List

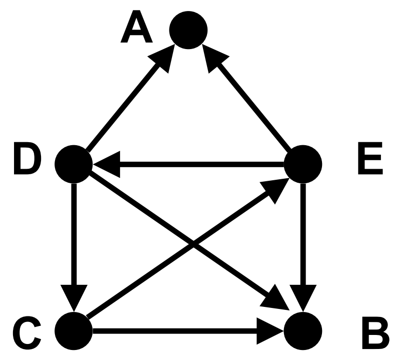
F			
G	F		
H	J	G	
I	F	H	
J	F	G	

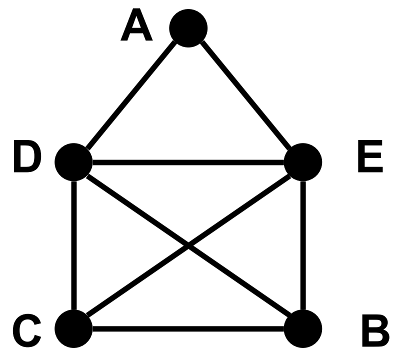
Adjacency Matrix

	F	**G**	**H**	**I**	**J**
F	0	0	0	0	0
G	1	0	0	0	0
H	0	1	0	0	1
I	1	0	1	0	0
J	1	1	0	0	0

PRACTICING GRAPH REPRESENTATIONS

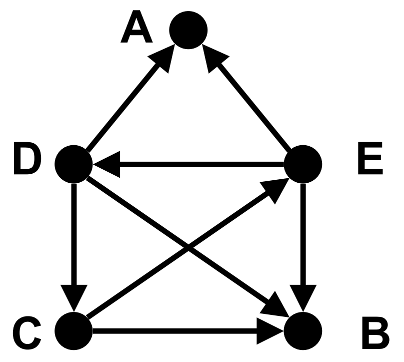






A	D	E		
B	C	D	E	
C	B	D	E	
D	A	B	C	E
E	A	B	C	D

	A	**B**	**C**	**D**	**E**
A	0	0	0	1	1
B	0	0	1	1	1
C	0	1	0	1	1
D	1	1	1	0	1
E	1	1	1	1	0



A			
B			
C	B	E	
D	A	B	C
E	A	B	D

	A	**B**	**C**	**D**	**E**
A	0	0	0	0	0
B	0	0	0	0	0
C	0	1	0	0	1
D	1	1	1	0	0
E	1	1	0	1	0

Graph Implementation

- Graphs can be put together much like trees
  - *i.e.,* a set of nodes linking to other nodes
    - Note: Depending on the application, may not need to implement the nodes – just work with adjacency list or matrix
  - Each node has zero or more links
    - We won't know how many when we are building the graph
  - Adjacency List:
    - Use a linked list **within each node** to connect to other nodes
  - Adjacency Matrix:
    - Use an **n x n matrix** of Boolean or ones and zeroes (or values to represent weights)

Implementation Analysis

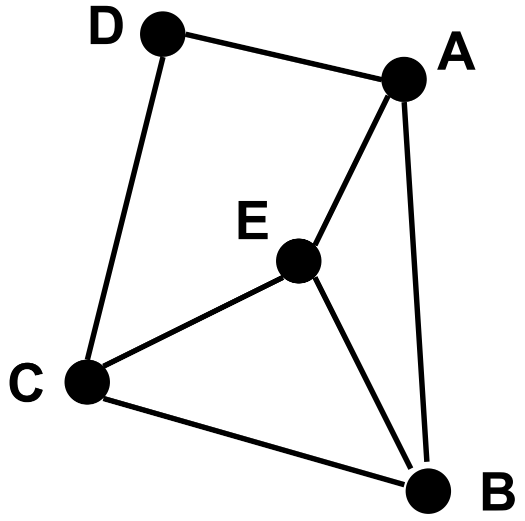
Operation/Space (*n* vertices & *m* edges)	**Adjacency List**	**Adjacency Matrix**
Contains edge (v_i, v_j)	O(# edges out of v_i)	O(1)
Iterate out-edges of v_i	O(# edges out of v_i)	O(n)
Iterate over all edges	O(n+m)	$O(n^2)$
Space	O(n+m)	$O(n^2)$

Graph as Adjacency Matrix

- Undirected or directed
- Maintain array to lookup labels (O(1))
- Adding vertex means increasing size of array, or start with default capacity and track size (like original stack/queue)...

Label Lookup

0	A
1	B
2	C
3	D
4	E



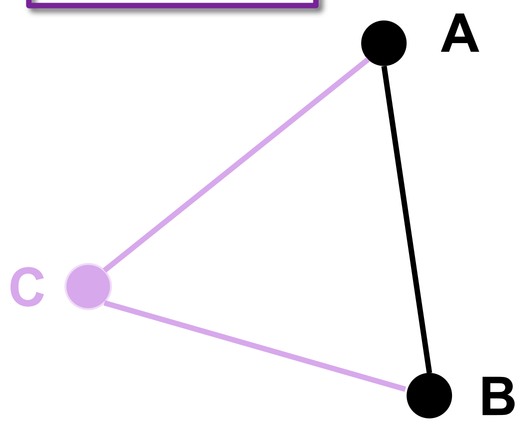
Adjacency Matrix

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

Graph as Adjacency Matrix

- Building a Graph:
  - May have list of vertices, then list of connections (edges)
  - Can also just have list of connections, if label not in list, add vertex before adding edge
- Input file:

```
A B
A C
B C
```



Lookup

0	A
1	B
2	
3	
4	

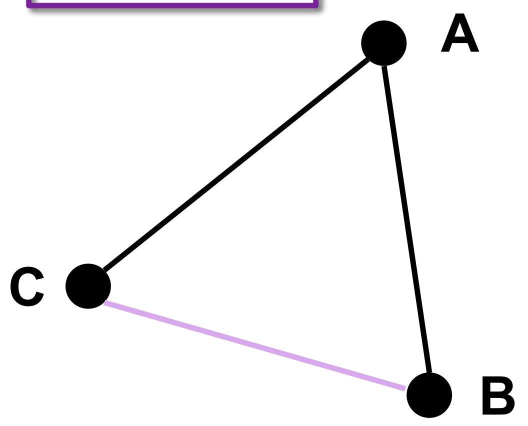
Adjacency Matrix

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

Graph as Adjacency Matrix

- Building a Graph:
  - May have list of vertices, then list of connections (edges)
  - Can also just have list of connections, if label not in list, add vertex before adding edge
- Input file:

```
A B
A C
B C
```



Lookup

0	A
1	B
2	C
3	
4	

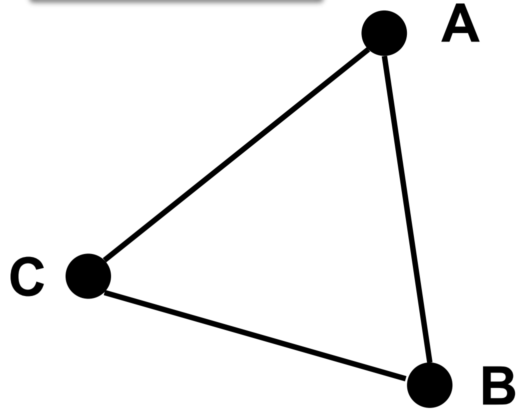
Adjacency Matrix

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

Graph as Adjacency Matrix

- Building a Graph:
  - May have list of vertices, then list of connections (edges)
  - Can also just have list of connections, if label not in list, add vertex before adding edge
- Input file:

```
A B
A C
B C
```



Lookup

0	A
1	B
2	C
3	
4	

Adjacency Matrix

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

DSAGraph Class – Adjacency Matrix

- Holds Vertices in 2D array, numbers in matrix are number of edges or a weight between two vertices

```
CLASS DSAGraph
FIELDS: matrix(n x n array), labels (lookup for labels, if needed)

CONSTRUCTOR DSAGraph IMPORTS NONE   // could import n, number of vertices

MUTATOR addVertex IMPORTS label[, value]  EXPORTS NONE
MUTATOR addEdge IMPORTS label1, label2 EXPORTS NONE
              // if undirected, add in both directions
ACCESSOR hasVertex IMPORTS label EXPORTS boolean
ACCESSOR getVertexCount IMPORTS NONE EXPORTS int
ACCESSOR getEdgeCount IMPORTS NONE EXPORTS int

ACCESSOR isAdjacent IMPORTS label1, label2 EXPORTS boolean
ACCESSOR getAdjacent IMPORTS label EXPORTS vertexList
ACCESSOR displayAsList IMPORTS NONE EXPORTS NONE
ACCESSOR displayAsMatrix IMPORTS NONE EXPORTS NONE
```

[brackets] indicate optional

DSAGraph Class – Adjacency Matrix

- Holds Vertices in 2D array, numbers in matrix are number of edges or a weight between two vertices

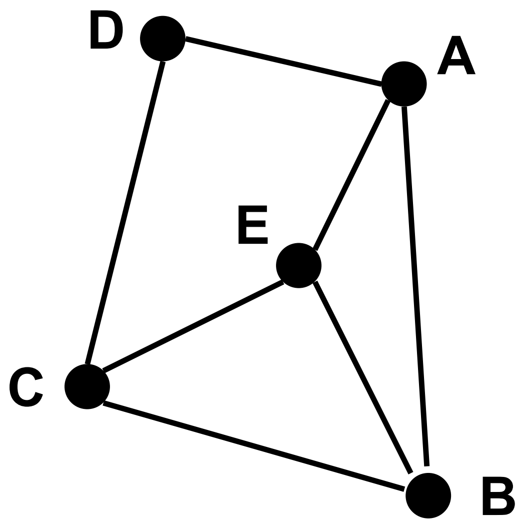
```
DSAGraph
--------------------------------------------
matrix(n x n matrix)
labels
--------------------------------------------
__init__() or DSAGraph()
+ addVertex(label)
+ addEdge(label1, label2)
+ hasVertex(label): boolean
+ getVertexCount(): int
+ getEdgeCount(): int
+ isAdjacent (label1, label2): boolean
+ getAdjacent (label): vertexList
+ displayAsList()
+ displayAsMatrix()
```

Graph as Adjacency List

- Complexity of implementation depends on program needs:
  - Lists and links (no data)
  - Lists of GraphNodes (data in nodes)
  - Lists of GraphNodes and Lists of Edges

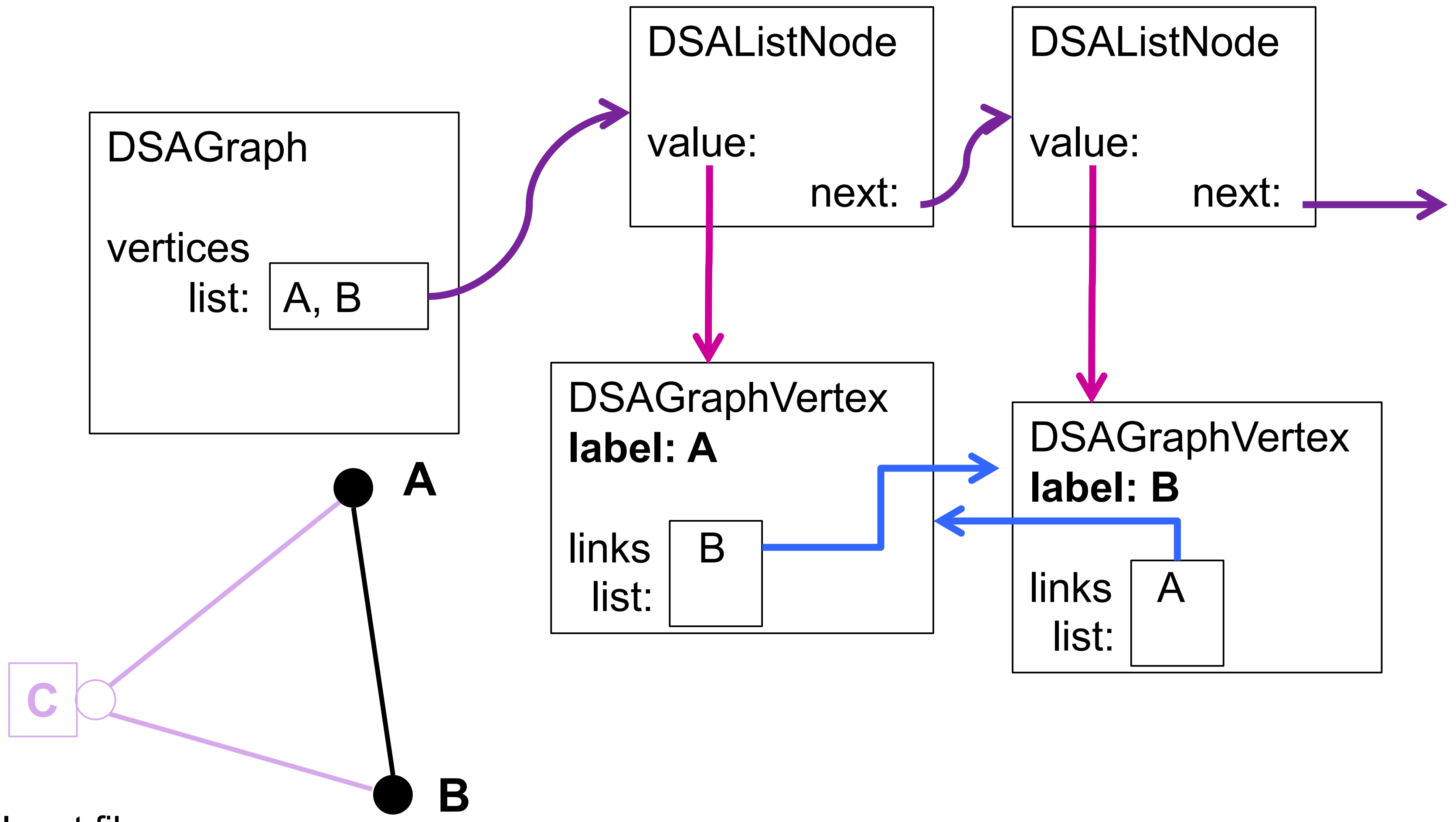
Adjacency List

A	B	D	E
B	A	C	E
C	B	D	E
D	A	C	
E	A	B	C



- If you only need to store the links, a very simple structure is possible:
  - Just a linked list of vertices
  - Link up by referencing itself
- We will use a List of GraphNodes to store the information for each vertex

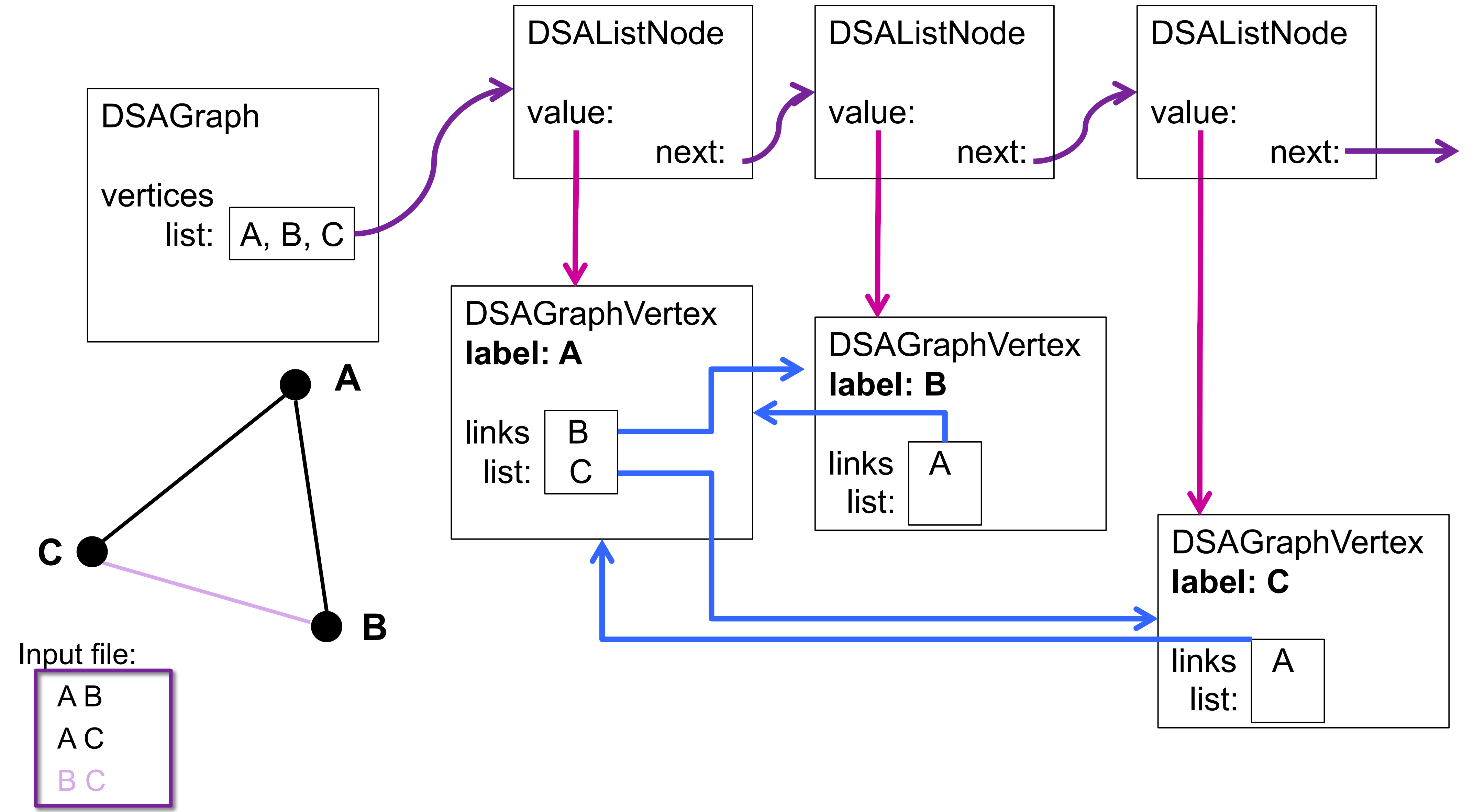
Graph as Adjacency List (use for Pracs)



Input file:

```
A B
A C
B C
```

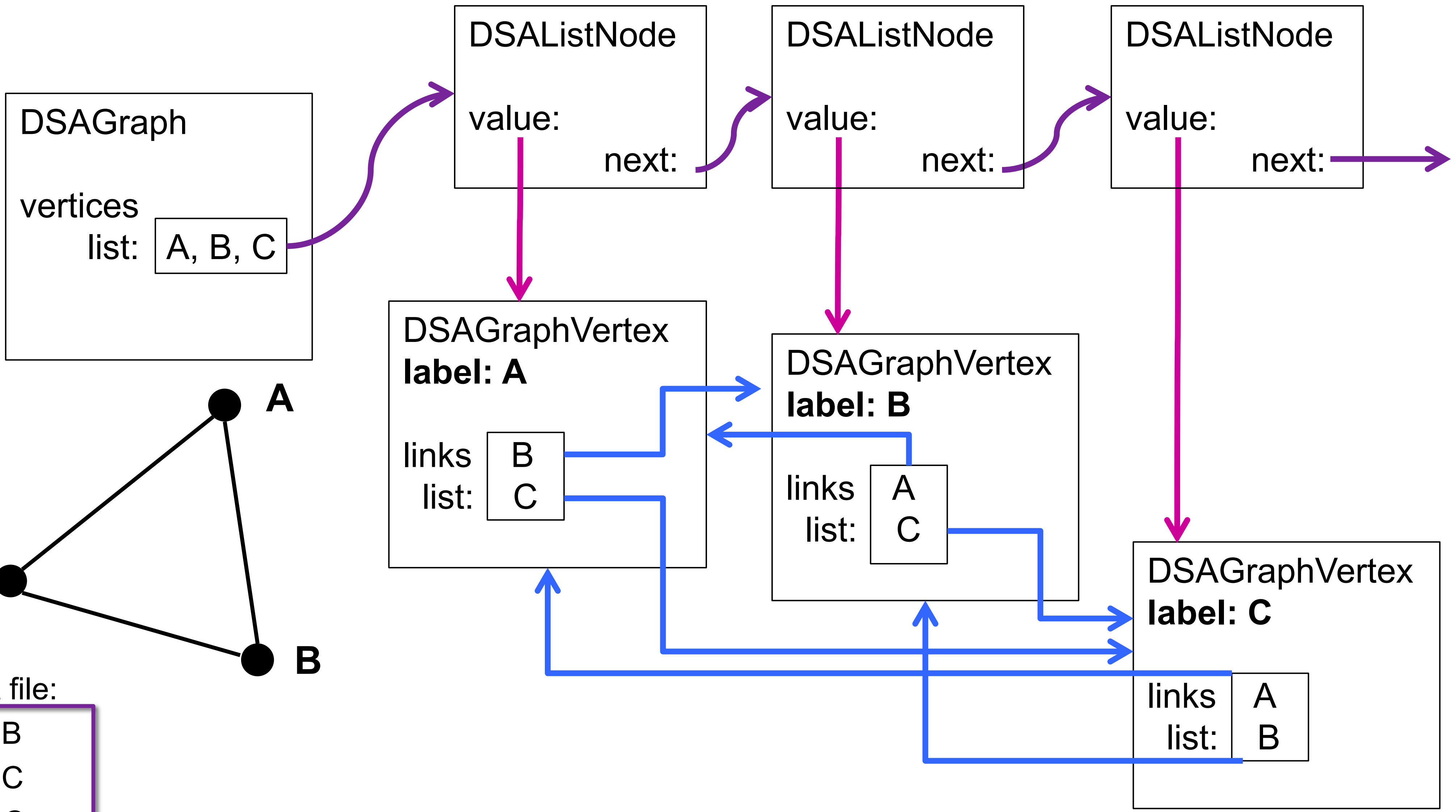
Graph as Adjacency List (use for Pracs)

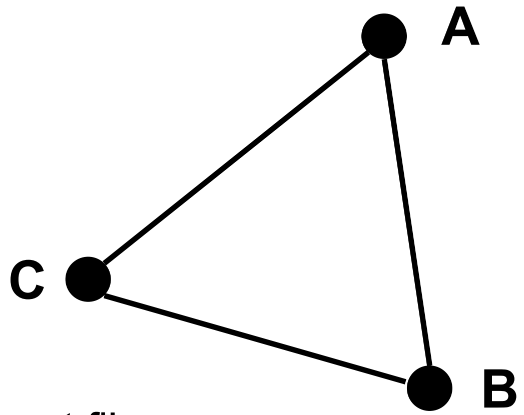


Input file:

A B
A C
B C

Graph as Adjacency List (use for Pracs)





Input file:

```
A B
A C
B C
```

DSAGraph Class – Adjacency List

- Holds Vertices in linked list
- Each Vertex has a label, a possible value and a linked list of neighbours

```
CLASS DSAGraph
FIELDS: vertices (DSALinkedList)

CONSTRUCTOR DSAGraph IMPORTS NONE

MUTATOR addVertex IMPORTS label [, value] EXPORTS NONE
MUTATOR addEdge IMPORTS label1, label2 EXPORTS NONE
              // if undirected, add in both directions
ACCESSOR hasVertex IMPORTS label EXPORTS boolean
ACCESSOR getVertexCount IMPORTS NONE EXPORTS int
ACCESSOR getEdgeCount IMPORTS NONE EXPORTS int

ACCESSOR getVertex IMPORTS label EXPORTS vertex
ACCESSOR getAdjacent IMPORTS label EXPORTS vertexList
ACCESSOR isAdjacent IMPORTS label1, label2 EXPORTS boolean
ACCESSOR displayAsList IMPORTS NONE EXPORTS NONE
ACCESSOR displayAsMatrix IMPORTS NONE EXPORTS NONE
```

Vertices are set up similarly to nodes in linked lists

DSAGraphVertex Class – Adjacency List

- Label is usually a String or an int
- Optional value could be an Object

```
CLASS DSAGraphVertex
FIELDS: label, [value,] links, visited

CONSTRUCTOR DSAGraphVertex IMPORTS inLabel[, inValue]

ACCESSOR getLabel IMPORTS NONE EXPORTS label
ACCESSOR getValue IMPORTS NONE EXPORTS value
ACCESSOR getAdjacent IMPORTS NONE EXPORTS vertexList

MUTATOR addEdge IMPORTS vertex EXPORTS NONE

MUTATOR setVisited IMPORTS NONE EXPORTS NONE    //for searching (later)
MUTATOR clearVisited IMPORTS NONE EXPORTS NONE
ACCESSOR getVisited IMPORTS NONE EXPORTS Boolean

ACCESSOR toString IMPORTS NONE EXPORTS string
```

DSAGraph Class – with Edges

- Most complex version
- Has Vertex and Edge objects in lists

```
CLASS DSAGraph
FIELDS: vertices, edges (DSALinkedList)    // edges have their own class

CONSTRUCTOR DSAGraph IMPORTS NONE

MUTATOR addVertex IMPORTS label, value EXPORTS NONE
MUTATOR addEdge IMPORTS label1, label2 [edgeLabel, value] EXPORTS NONE

ACCESSOR hasVertex IMPORTS label EXPORTS boolean
ACCESSOR getVertexCount IMPORTS NONE EXPORTS int
ACCESSOR getEdgeCount IMPORTS NONE EXPORTS int
ACCESSOR getVertex IMPORTS label EXPORTS vertex
ACCESSOR getEdge IMPORTS label EXPORTS edge

ACCESSOR getAdjacent IMPORTS label EXPORTS vertexList
ACCESSOR getAdjacentE IMPORTS label EXPORTS edgeList
```

DSAGraphVertex Class

```
CLASS DSAGraphVertex
FIELDS: label, value, [links,] visited
   # edges may be sole keepers of connections -> no "links" in vertex

CONSTRUCTOR DSAGraphVertex IMPORTS inLabel, inValue

ACCESSOR getLabel IMPORTS NONE EXPORTS label

ACCESSOR getValue IMPORTS NONE EXPORTS value
ACCESSOR getAdjacent IMPORTS NONE EXPORTS vertexList
[ACCESSOR getAdjacentE IMPORTS NONE EXPORTS edgeList]

[MUTATOR addEdge IMPORTS edge/vertex EXPORTS NONE]

MUTATOR setVisited IMPORTS NONE EXPORTS NONE   //later
MUTATOR clearVisited IMPORTS NONE EXPORTS NONE
ACCESSOR getVisited IMPORTS NONE EXPORTS Boolean

ACCESSOR toString IMPORTS NONE EXPORTS string
```

DSAGraphEdge Class

- May be required if storing weights or other information
- Label is usually a String or an int
  - *e.g.*, weight or relationship are useful labels (uniqueness not important)

- Value can be an int or a more complex Object

```
CLASS DSAGraphEdge
FIELDS: from, to, label, value    # can have label and/or value

CONSTRUCTOR DSAGraphEdge IMPORTS fromVertex, toVertex, inLabel, inValue

ACCESSOR getLabel IMPORTS NONE EXPORTS label
ACCESSOR getValue IMPORTS NONE EXPORTS value

ACCESSOR getFrom IMPORTS NONE EXPORTS vertex
ACCESSOR getTo IMPORTS NONE EXPORTS vertex

ACCESSOR isDirected IMPORTS NONE EXPORTS boolean

ACCESSOR toString IMPORTS NONE EXPORTS string
```

Adding Vertices and Edges

- Adding vertices
  - Adjacency **lists** is trivial – they don't need to be connected to the rest of the graph straight away
  - With adjacency **matrix**, add to label list (resize matrix if needed)
- Adding edges
  - Adding edges to an adjacency **matrix** is trivial (change a value)
  - When using adjacency **lists** means finding both vertices, and adding the vertex or edge to the adjacency list (both directions, if undirected)

DSAGraph Class – optional & additional methods

```
<may be added as needed, could vary imports/exports, some should be private>

ACCESSOR degree IMPORTS vertex EXPORTS int
ACCESSOR hasValue IMPORTS value EXPORTS boolean
ACCESSOR isAdjacent IMPORTS label1, label2 EXPORTS boolean

ACCESSOR nextVertex IMPORTS NONE EXPORTS vertex

      - can use iterator to go through vertices
      - may need to sort them to have a consistent order...

ACCESSOR sort IMPORTS NONE EXPORTS none

      - often alphabetical or numeric
      - can include sort as part of insert to do incrementally

ACCESSOR display IMPORTS NONE EXPORTS string
      Need at least one display method
         Variations:
            - as adjacency lists
            - as adjacency matrices
            - traverse using DFS or BFS and convert
```

SEARCHING AND TRAVERSING

Now you have your graph,
what are you going to do with it…?

Finding in a Graph

- More tricky than trees and lists as no leaf nodes or tail
- May have cycles, so could go on forever
- Need to keep track of vertices visited
- Approach:
  - Move through the graph, marking vertices as **visited**
  - On each step, continue through the not visited vertices until all vertices have been visited
- Two basic graph traversal algorithms:
  - Depth-First Search
  - Breadth-First Search
- Later units will cover other graph algorithms

Graph Traversal

- To find a value in a graph, you need to traverse the entire graph in an orderly way
- You may also want to display or store all values in the graph

- **Depth-first search** is similar to a pre-order traversal of a tree – go as deep along each path as possible before returning
- **Breadth-first search** goes level by level as you move away from the starting point

- Need to mark vertices as visited for algorithms to work

Depth-first Search

- **depthFirstSearch**( )

- **Uses**: G = (V, E) in adjacency list format
  v = node on top of stack S
  L[v] refers to the adjacency list of v

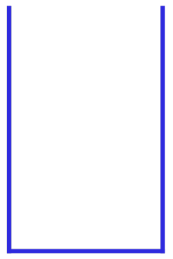
- **Output**: The DFS tree *T*
  Could be a queue of objects, or string of labels
  or a new graph (that's a tree)…

Depth-first Search - Algorithm

```
1.  mark all vertices new and set T = { }
2.  mark any one vertex v to old        // Alpha order preferred
3.  push (S, v)                         // push v onto stack S
4.  while S is nonempty do
5.      while exists a new vertex w in L[v] do
6.          T = T ∪ (v, w)              // add to traversal tree
7.          mark w as old               // set visited flag to true
8.          push (S, w)                 // store for later...
9.          v = w                       // move along branch...
10.     v = pop S                       // backtracking...
```

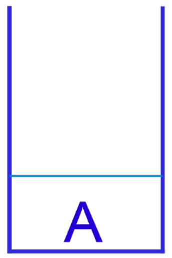
Depth-first Search

1



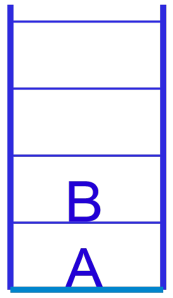
new = {A,B,C,D,E}
old = {}
T = {}

2



new = {B,C,D,E}
old = {A}
L[A] = {B,D,E}
T = {}

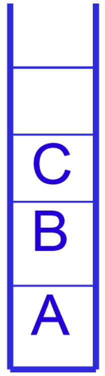
3



new={C,D,E}

L[B] = {A,C,E}
old = {A,B}
T = {(A,B)}

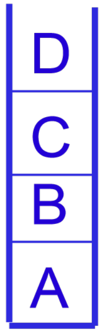
4



new={D,E}

L[C] = {B,D,E}
old = {A,B,C}
T = {(A,B),(B,C)}

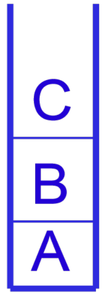
5



New ={E}

L[D] ={A,C}
old = {A,B,C,D}
T={(A,B),(B,C),(C,D)}

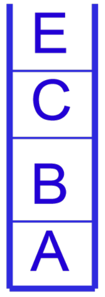
6



new ={E}

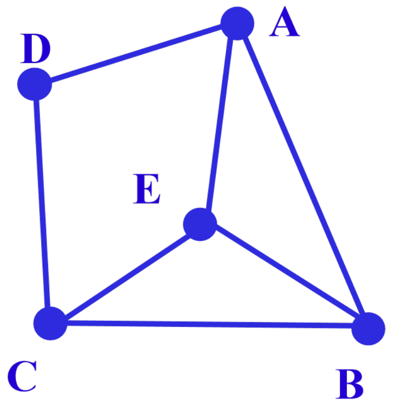
L[C] = {B,D,E}
old = {A,B,C,D}
T={(A,B),(B,C),(C,D),(C,E)}

7



new = {}

L[E]={A,B,C}
old = {A,B,C,D,E}
T = {(A,B),(B,C),(C,D),(C,E)}



Breadth-first Search

- **breadthFirstSearch**( )

- **Uses:** G = (V, E)
  v = value at front of queue Q
  L[v] refers to the adjacency list of v

- **Output:** The BFS tree T

Breadth-first Search – Algorithm

```
1. Mark all vertices new and set T = { }
2. Mark the start vertex v = old
3. insert (Q, v)                         // Q is a queue
4. while Q is nonempty do
5.     v = dequeue (Q)
6.     for each vertex w in L[v] marked new do
7.         T = T ∪ {v,w}
8.         mark w = old
9.         insert (Q,w)
```

Breadth-first Search

1

Q | |

new = {A,B,C,D,E}
old = {}
T = {}

2

Q | A |

L[A]={B,D,E}
new = {B,C,D,E}
old = {A}
T = {}

3

Q | B | D | E |

L[B] = {A, C, E}
new = {C}
old = {A,B,D,E}
T = {(A,B),(A,D),(A,E)}

4

Q | D | E | C |

L[D]={A,C}
new = {}
old = {A,B,D,E,C}
T = {(A,B),(A,D),
(A,E),(B,C)}

5

Q | E | C |

L[E]={A,B,C}
new = {}
old = {A,B,D,E,C}
T = {(A,B),(A,D),
(A,E),(B,C)}

6

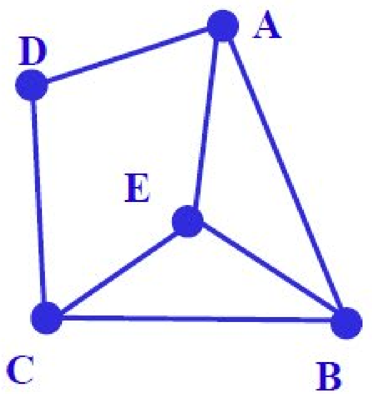
Q | C |

L[C]={B,D,E}
new = {}
old = {A,B,D,E,C}
T = {(A,B),(A,D),
(A,E),(B,C)}

7

Q | |

new = {}
old = {A,B,D,E,C}
T = {(A,B),(A,D),
(A,E),(B,C)}



MOTIVATIONAL SLIDES

Graph-based Assignments in DSA…

Graph Applications from Previous Semesters

Assignment Semester 2, 2017

- You are tasked with developing a program to hold information about locations and distances/times by various modes of transportation.
- You will be calculating the shortest path between locations, taking into consideration the transportation options the user requests, and any weather or traffic conditions that exist.
- For each pair of locations, there may be multiple distances and times for a range of travel types: walk, cycle, drive, bus, train, plane etc. Users can select which modes they are interested in travel times for.
- There will also be peak and offpeak times which will be selected based on the indicated mode.

Graph Applications from Previous Semesters

Assignment Semester 1, 2018

- It is 2050. A team of planetary scientists have discovered that there is a network of tunnels underneath the surface of Phoebe, one of the more unusual moons of Saturn.
- They have sent a team of identical robotic probes to Phoebe to map this network of tunnels. Each robot lands at a random point on the surface of Phoebe, drills down through the surface until it hits the tunnel network, then randomly moves through the network building a map. Each robot has sensors that detect various characteristics of the tunnels and their intersections. Oddly enough, they cannot detect each other. The robots regularly upload their maps to Earth for the scientists to process. It is your job to write the program that will process the data from this team of robots and merge it all together.

Graph Applications from Previous Semesters

Assignment Semester 2, 2018

- It is election time 2016. Australia is voting for a new government and Prime Minister to take us through the next 3 years...
- The campaign management team need to get overall information about their position in the lead-up to the election (we will use the 2016 election results as the "poll data")
- They also need to know which divisions should be visited to get maximum impact, and provide an itinerary based on the distances/time between the locations.

Graph Application

Assignment Semester 2, 2019

- You are developing a model to support research into the spread of information through a social network
- The model will need to load in data for a network from a file
- Your program will then simulate the spread of “news" over time, outputting statistics for each run
- Applications could include:
    - Fake news in social media
    - Measles / Ebola outbreaks
    - Zombie apocalyse modelling
- This scenario should allow you to reuse parts of many of your practicals, which you must self-cite

Graph Application

Assignment Semester 1, 2020 - Preview

- *Was going to be “A Day in the Life”*
- Given the current pandemic, we will now model **spread of disease**
- You will need to base it on established models, e.g. SIR
- More complex models will get more marks (see Shiflet ref.)
- These are often “systems models”, whereas this assignment requires a **graph-based** model of population connectedness
- Your code will need to take in parameters including:
  - # Susceptible, Infected population size/proportion (for SIR model)
  - Infectiousness of disease
  - Recovery rate
  - Interventions in place – e.g. social distancing, isolation, travel restrictions
- This scenario should allow you to reuse parts of many of your practicals, which you must self-cite