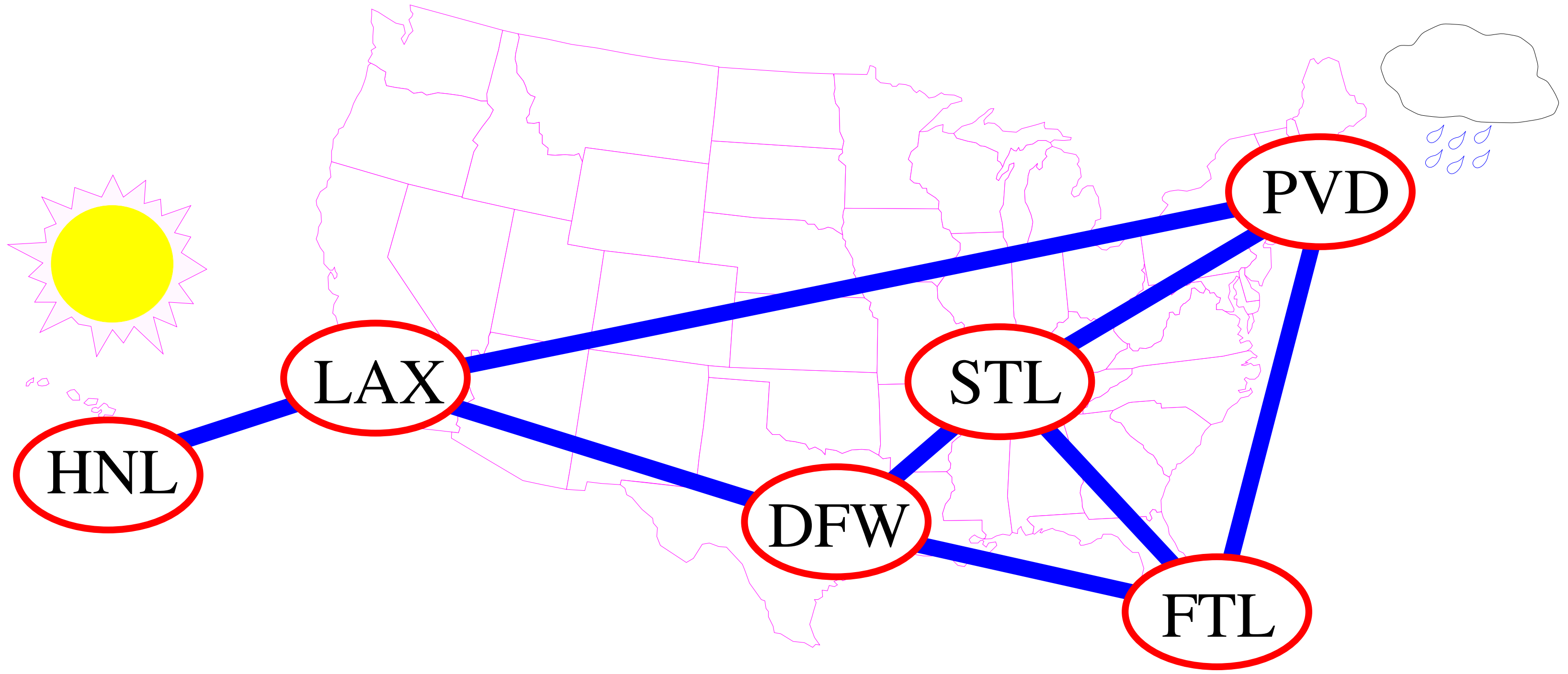
Winter 2016
COMP-250: Introduction to Computer Science

Lecture 16, March 10, 2016

GRAPHS

- Definitions
- Examples
- The Graph ADT



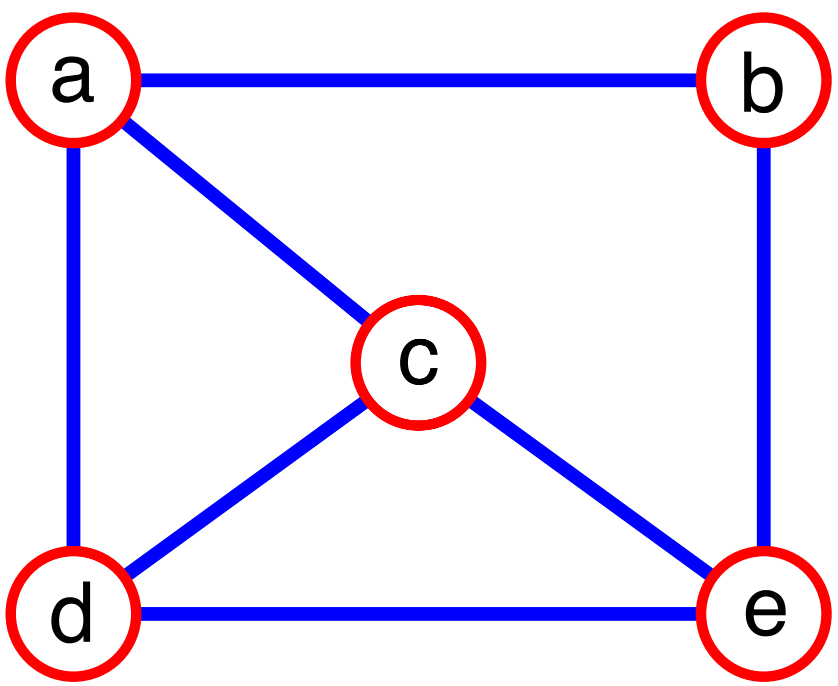
What is a Graph?

- A **graph G** = (**V**,**E**) is composed of:

  V: set of *vertices*

  E: set of *edges* connecting the *vertices* in **V**

- An **edge e** = (u,v) is a pair of vertices
- Example:

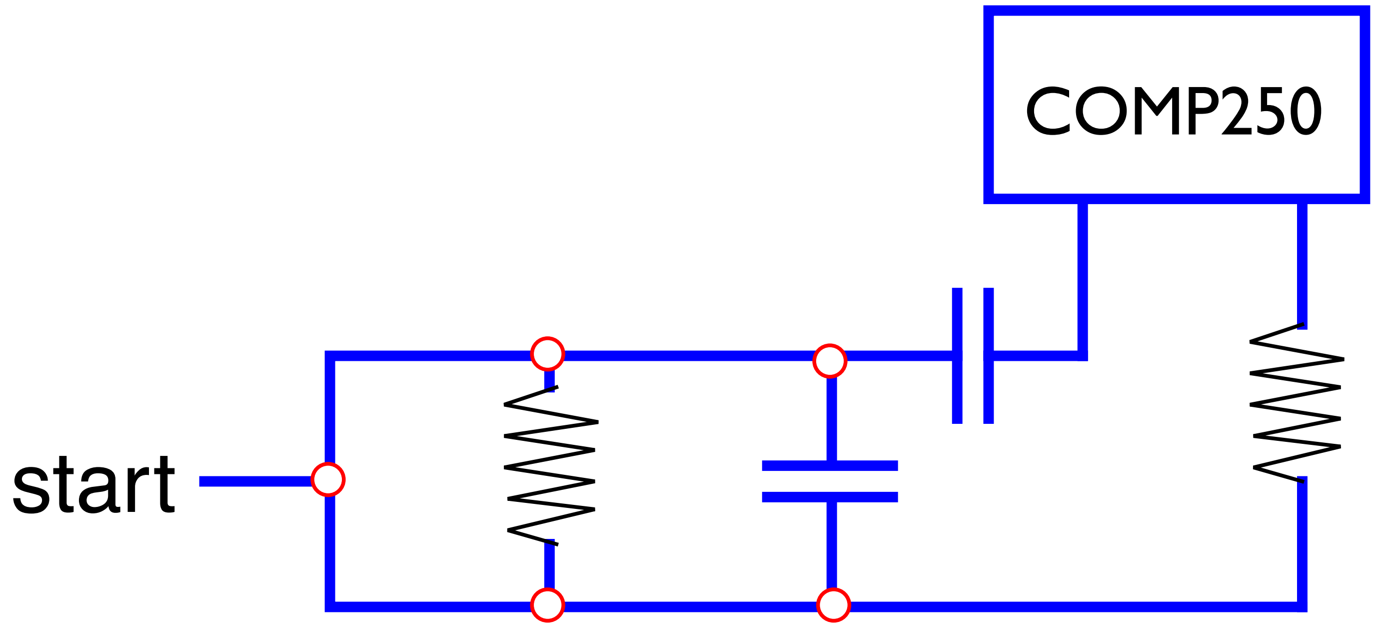


V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

Applications

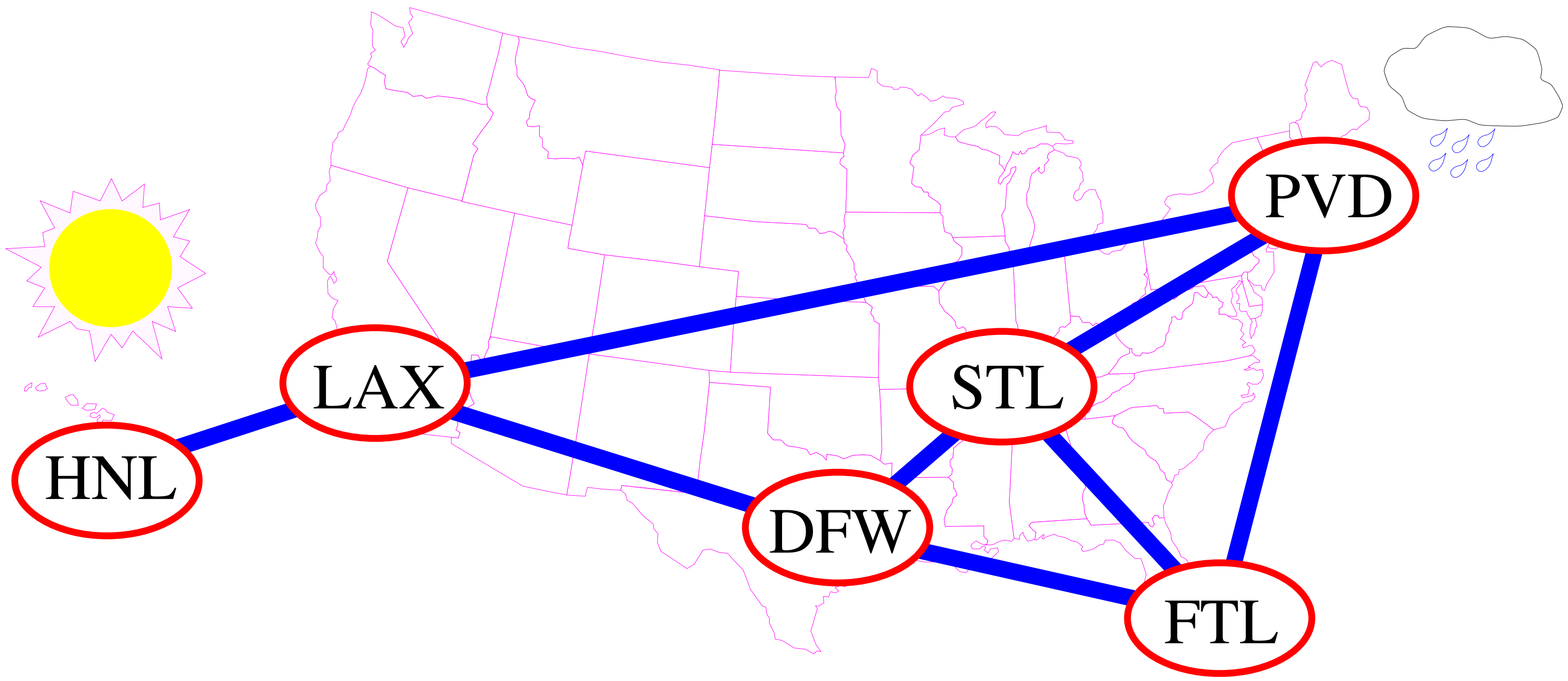
- electronic circuits



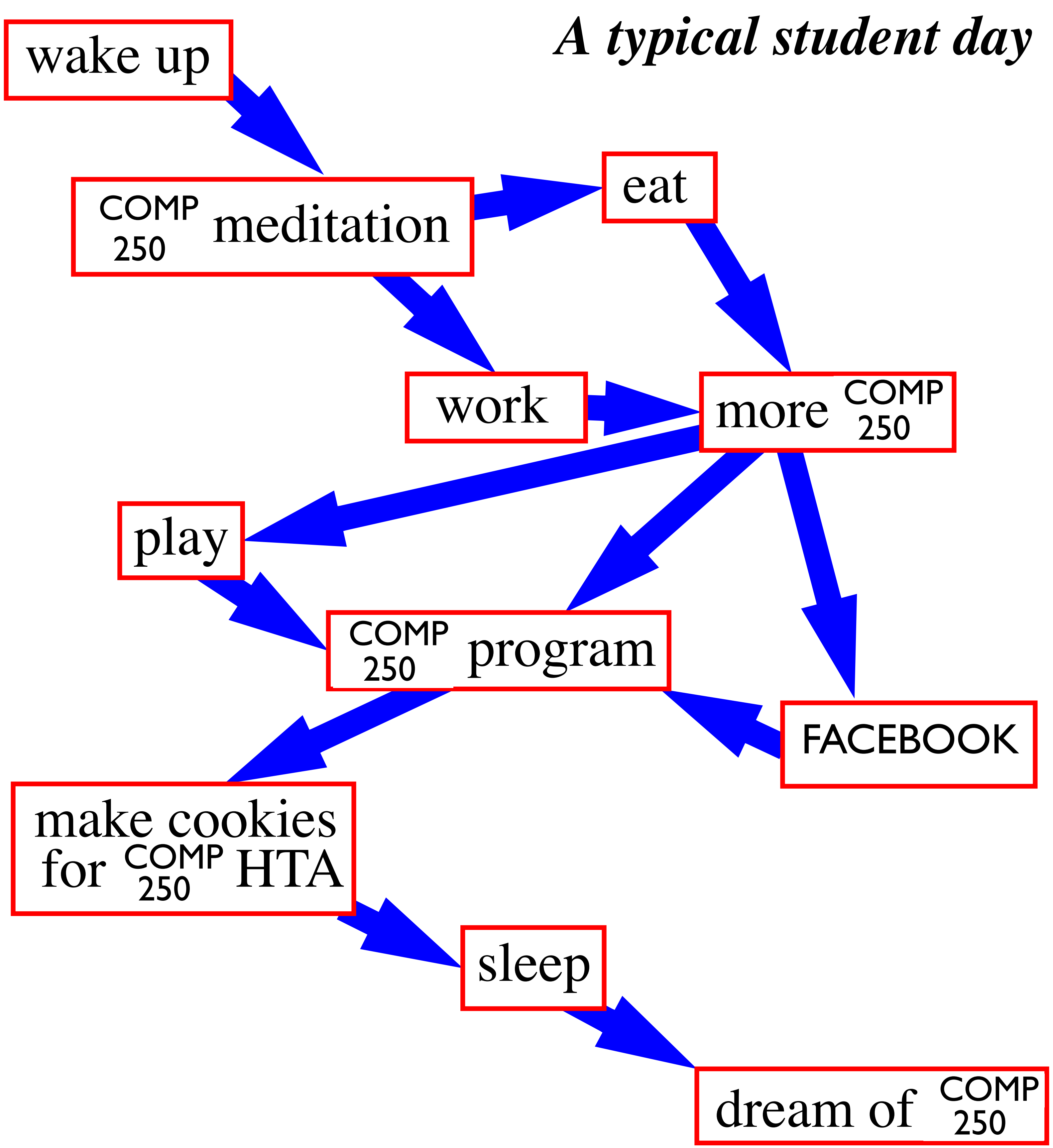
find the path of least resistance to COMP250

Applications

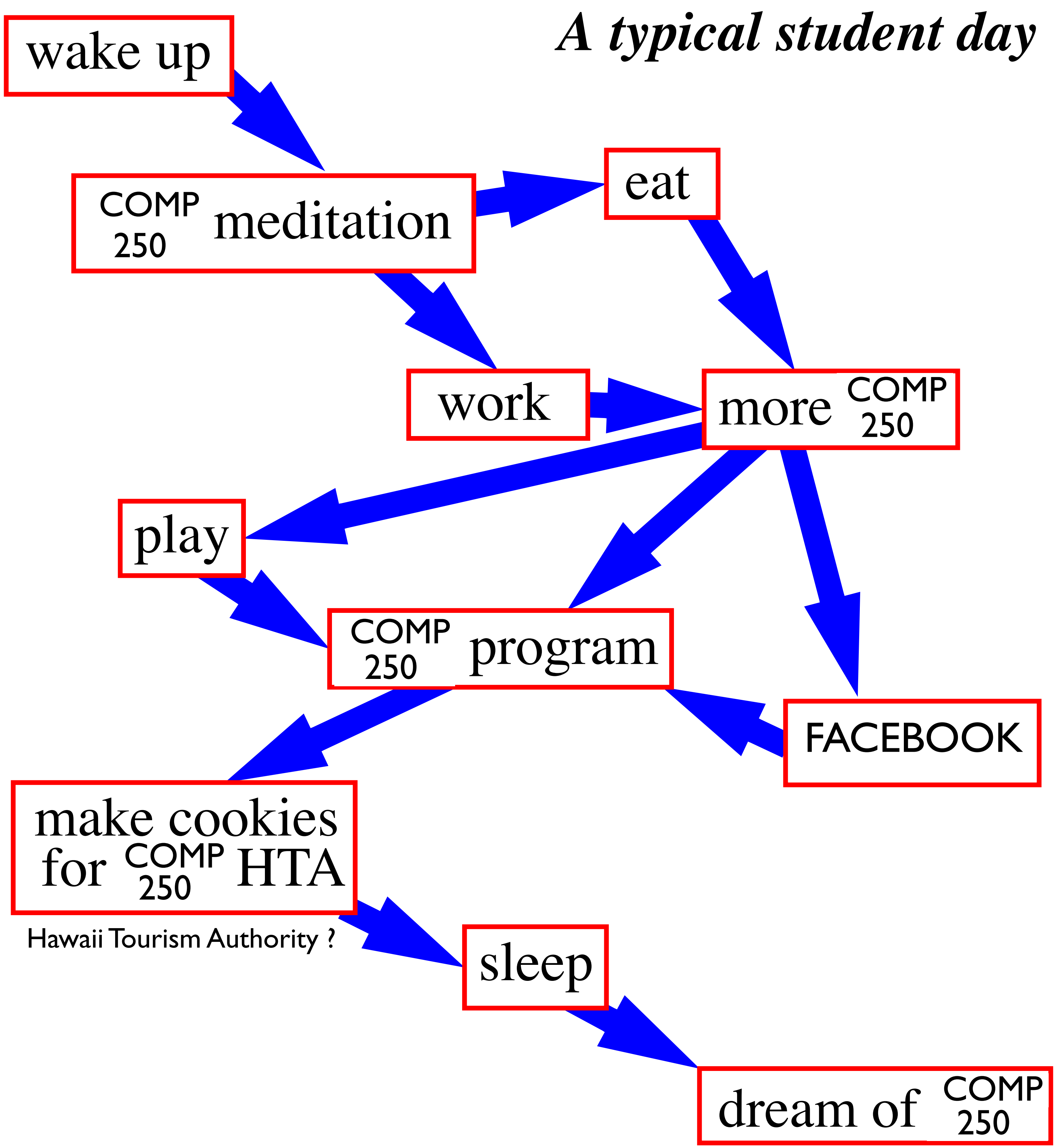
- networks (roads, flights, communications)



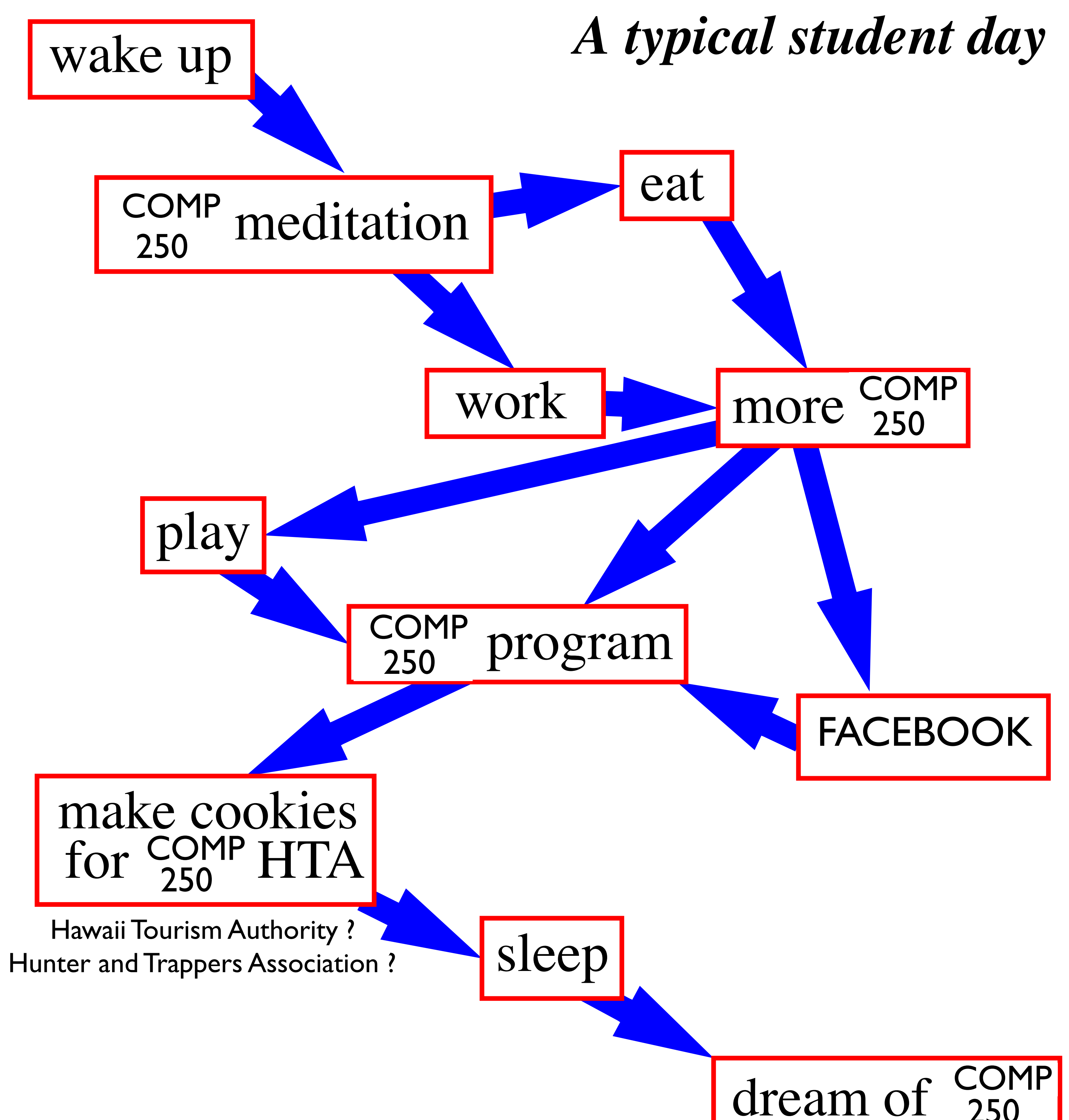
- scheduling (project planning)



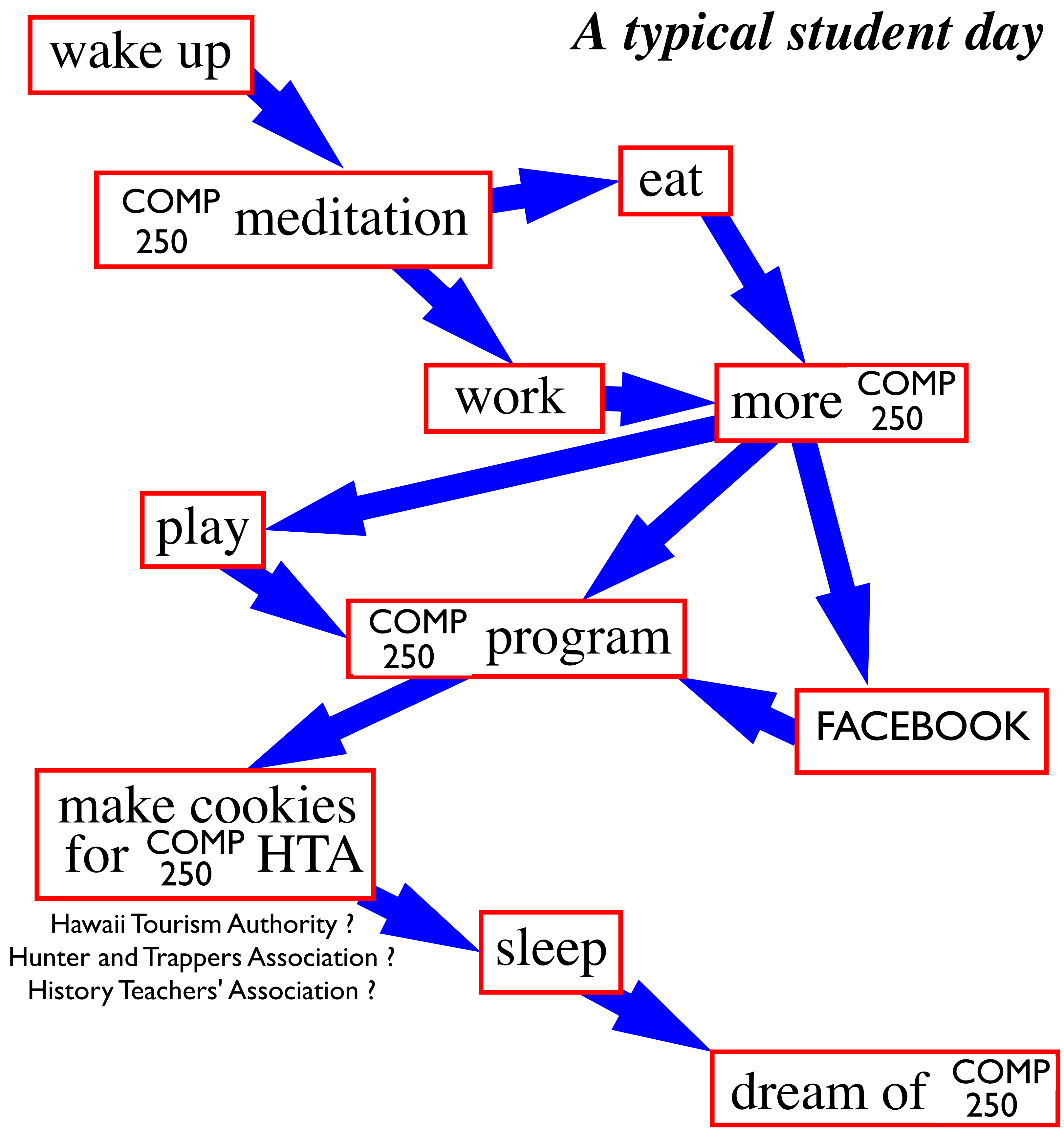
- scheduling (project planning)



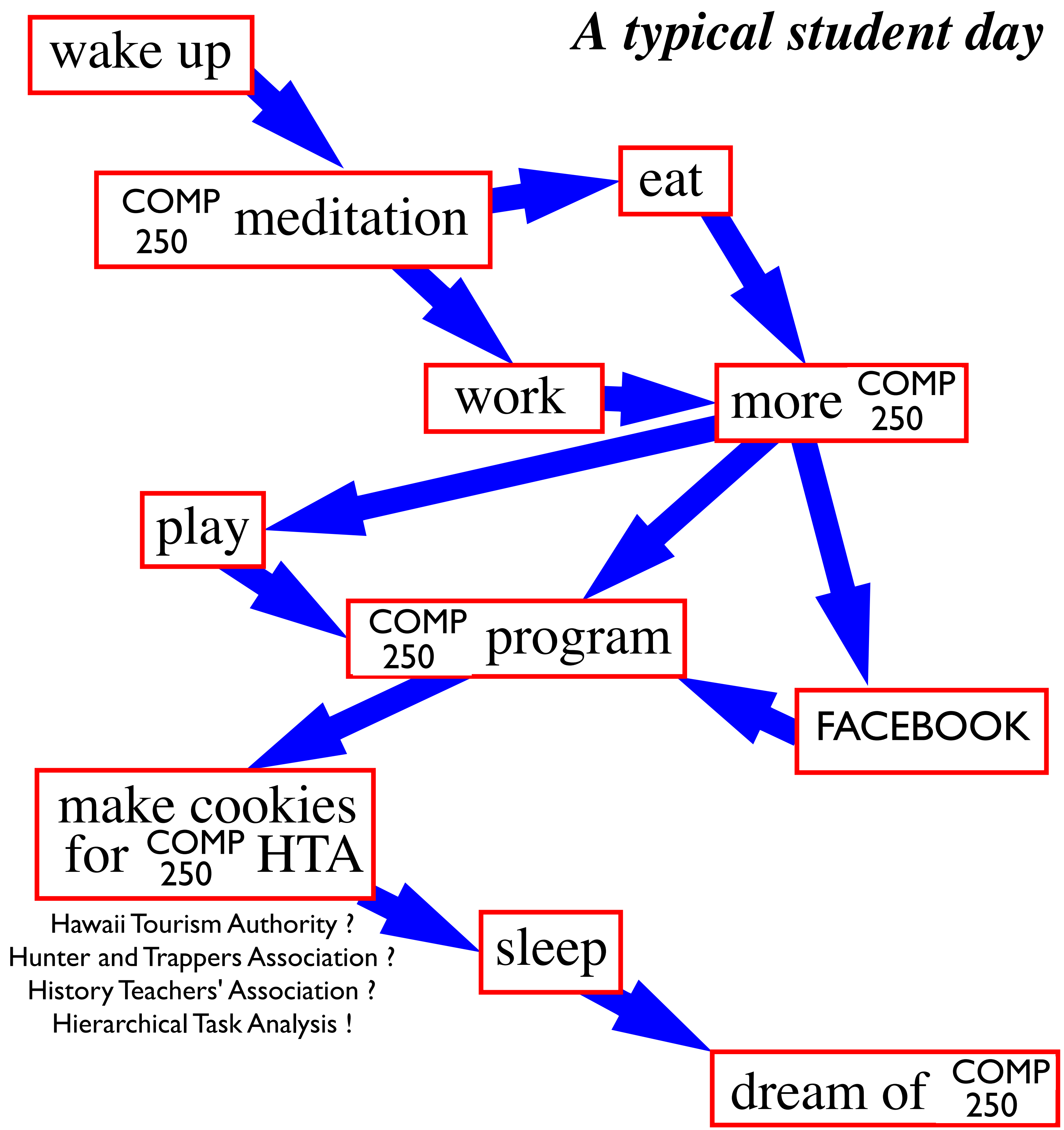
- scheduling (project planning)



- scheduling (project planning)



- scheduling (project planning)

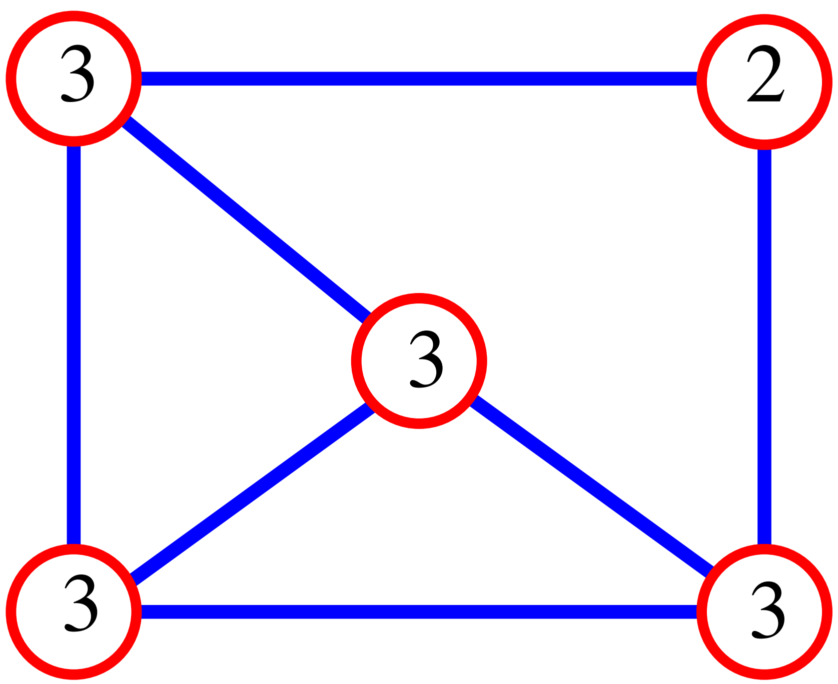


Applications

Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

Graph Terminology

- adjacent vertices: connected by an edge
- degree (of a vertex): # of adjacent vertices

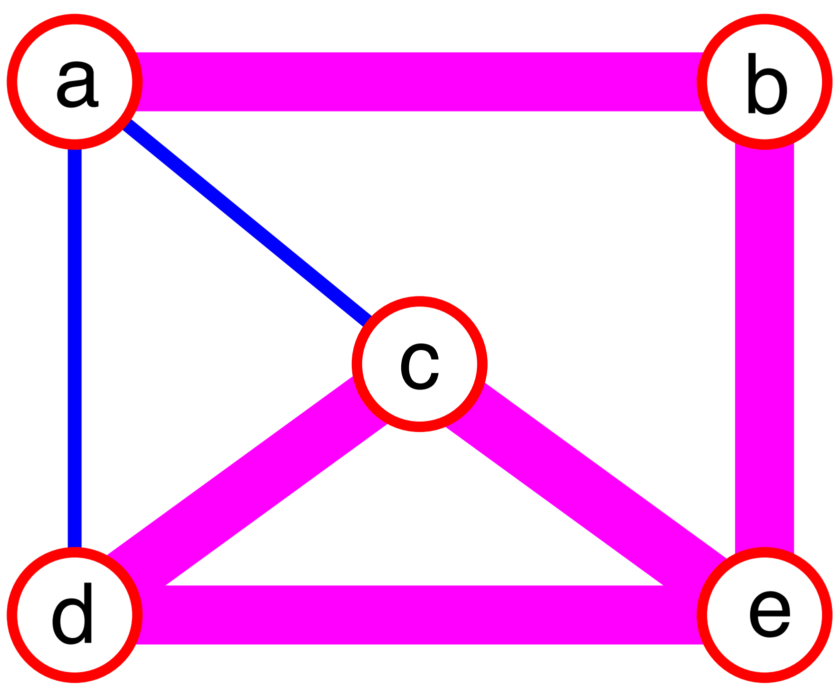


$$\sum_{v \in V} \deg(v) = 2(\#\ \text{edges})$$

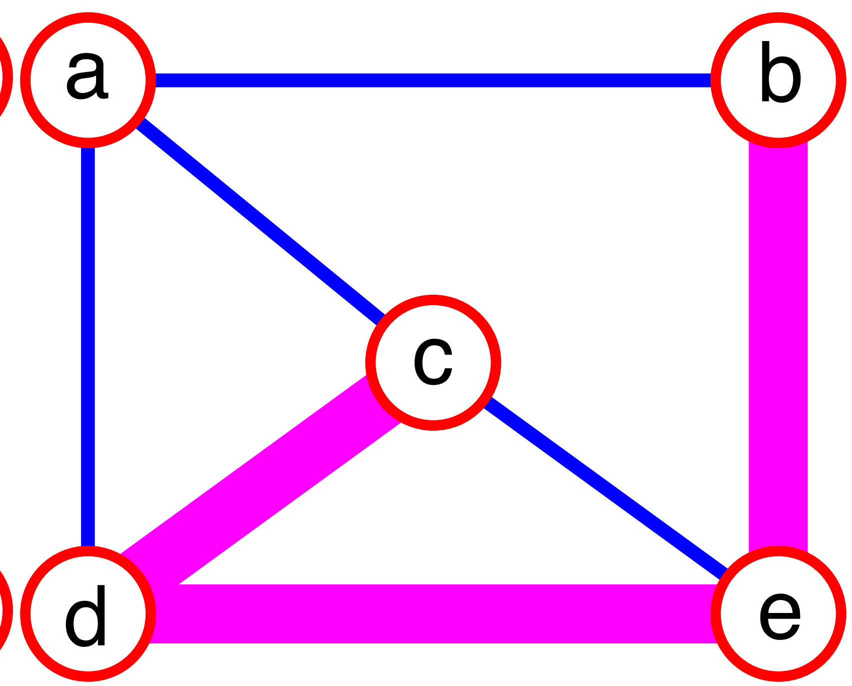
- Since adjacent vertices each count the adjoining edge, it will be counted twice

Graph Terminology

path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



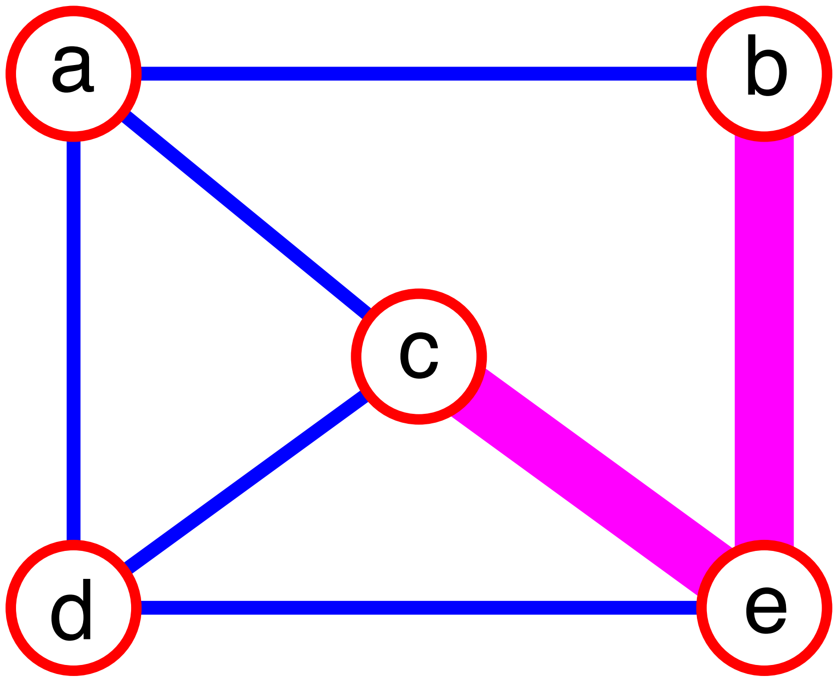
a b e d c



b e d c

More Graph Terminology

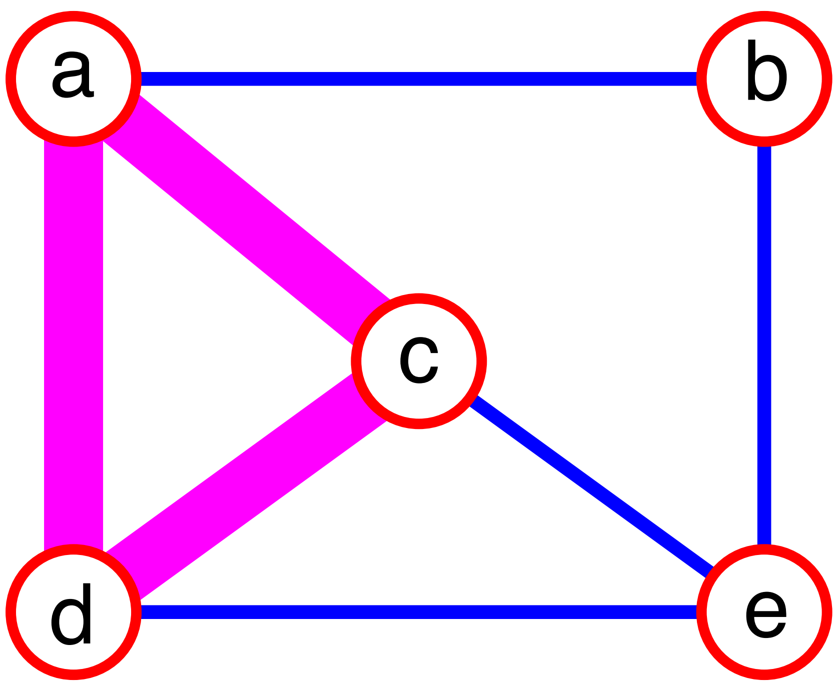
- simple path: no repeated vertices



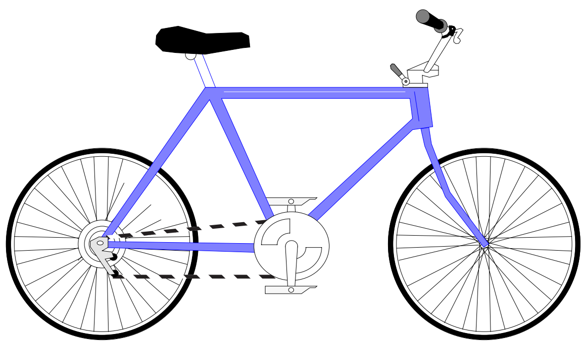
b e c

More Graph Terminology

- cycle: simple path, except that the last vertex is the same as the first vertex

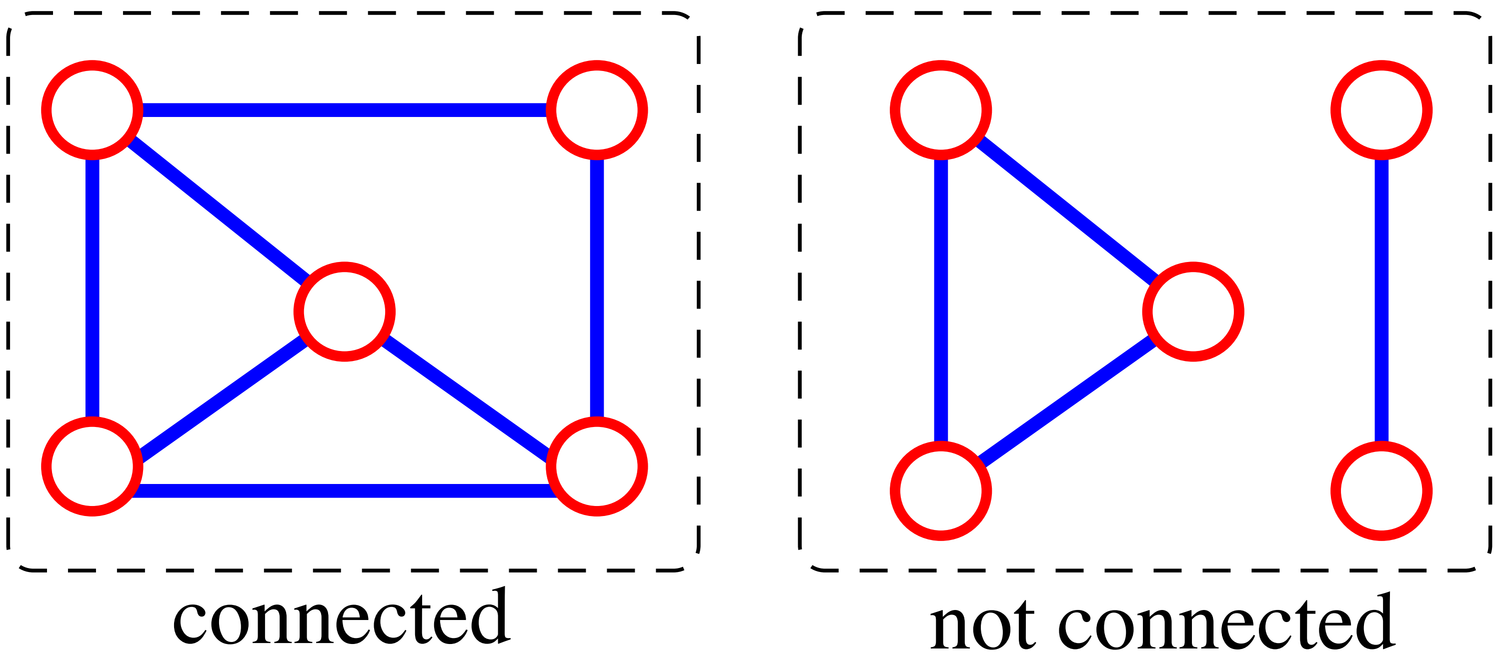


a c d a



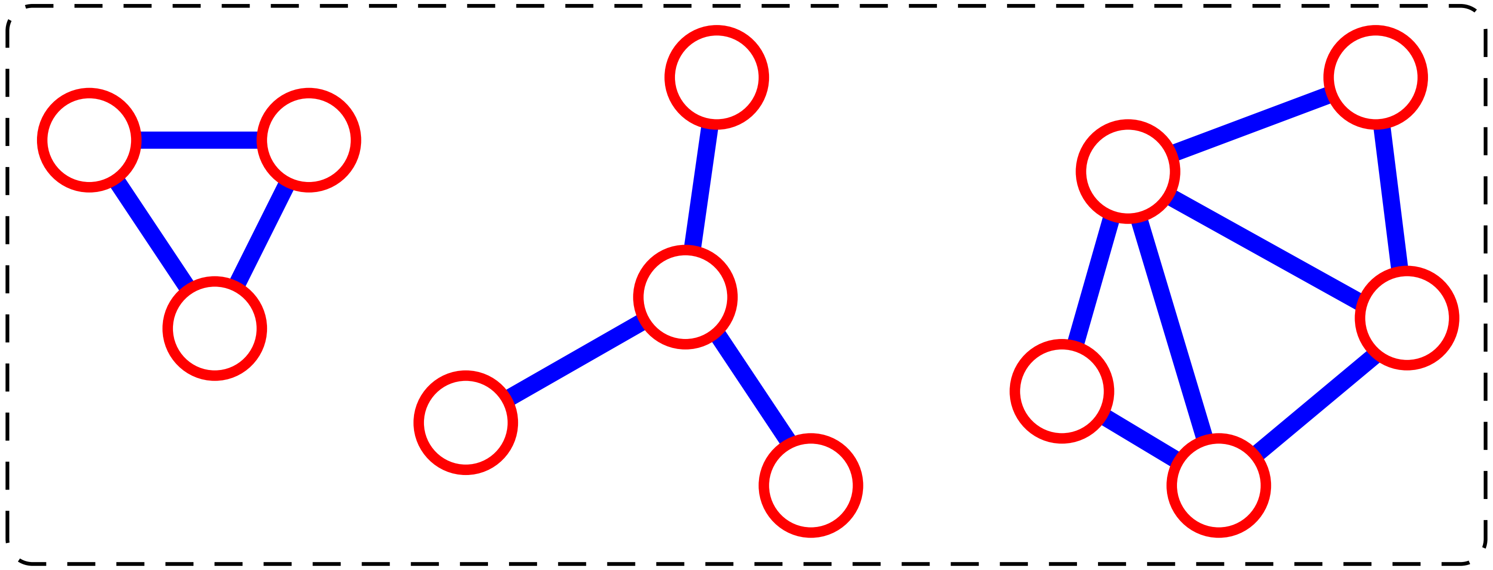
Even More Terminology

- connected graph: any two vertices are connected by some path



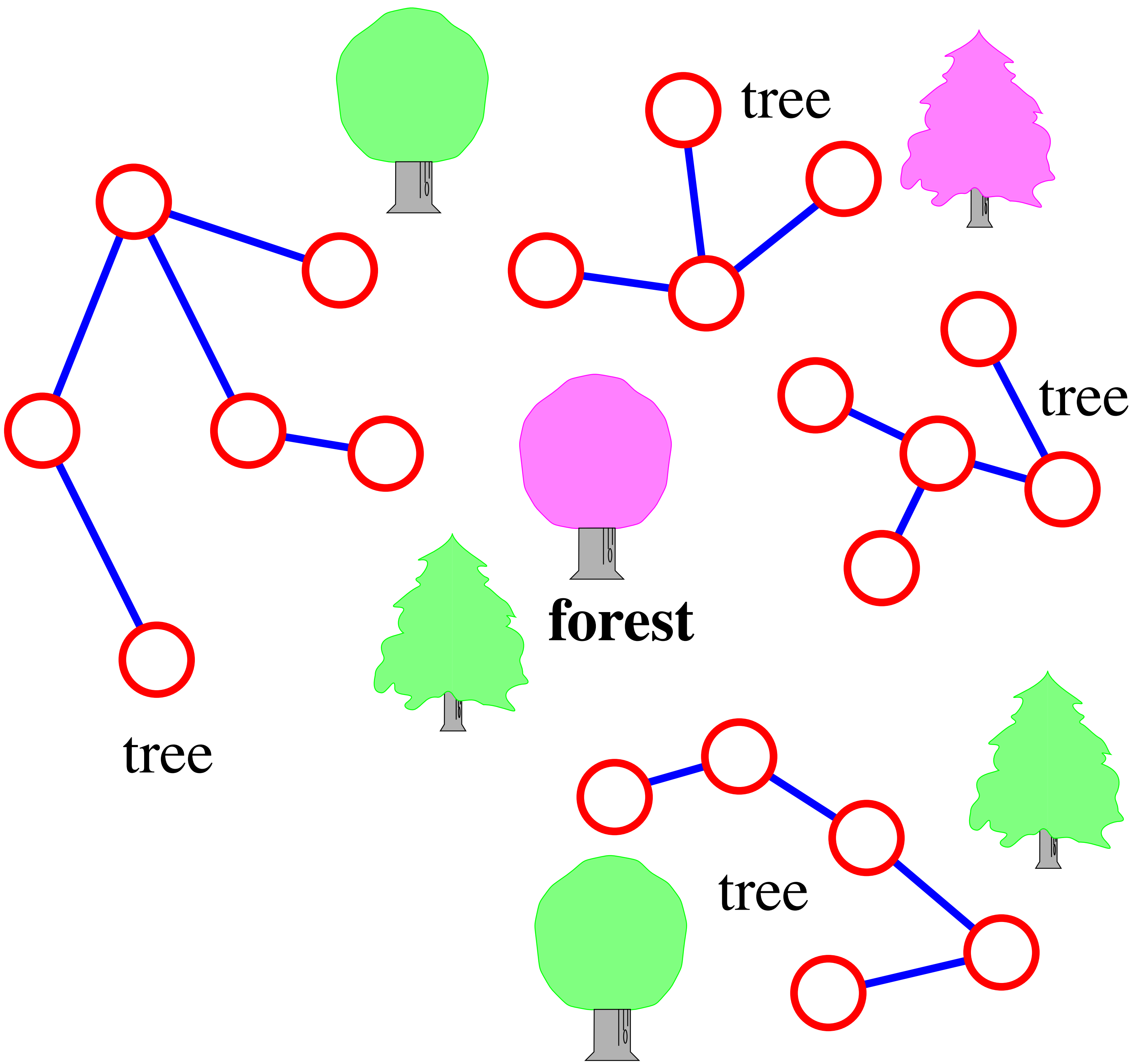
More Graph Terminology

- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Yet another Terminology Slide!

- (free) tree - connected graph without cycles
- forest - collection of trees



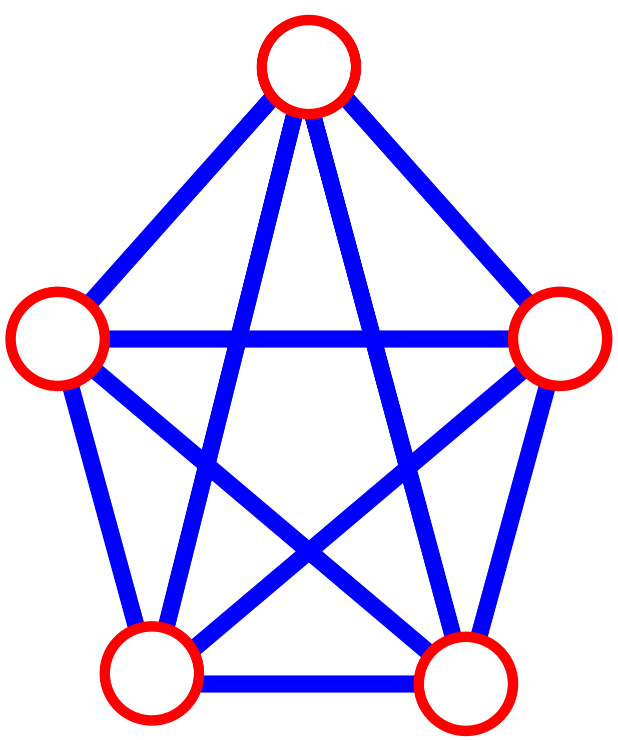
Connectivity

Let **n** = #vertices
m = #edges

- complete graph - all pairs of vertices are adjacent

$$\mathbf{m} = (1/2)\sum_{v \in \mathbf{V}} \deg(\mathbf{v}) = (1/2)\sum_{v \in V} (\mathbf{n} - 1) = \mathbf{n}(\mathbf{n}-1)/2$$

- Each of the **n** vertices is incident to **n** - 1 edges, however, we would have counted each edge twice!!! Therefore, intuitively, **m** = **n**(**n**-1)/2.

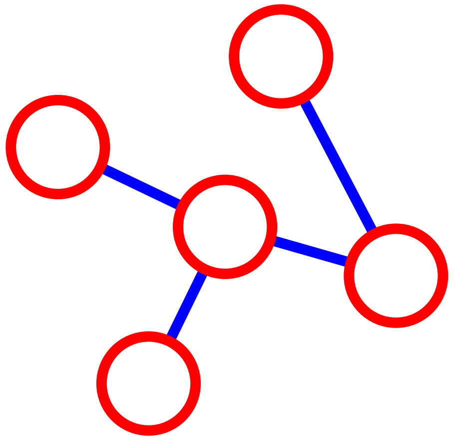


n = 5
m = (5 ∗ 4)/2 = 10

More Connectivity

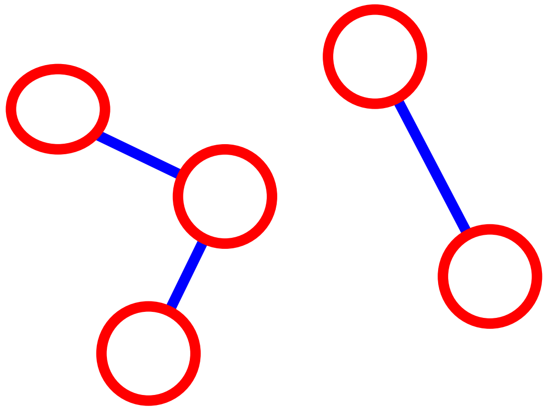
n = #vertices
m = #edges

- For a tree **m** = **n** - 1



n = 5
m = 4

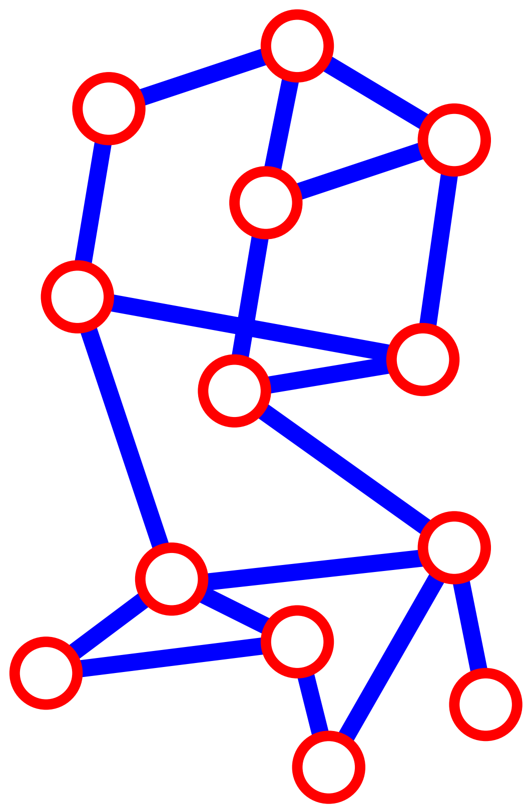
- If **m** < **n** - 1, G is not connected



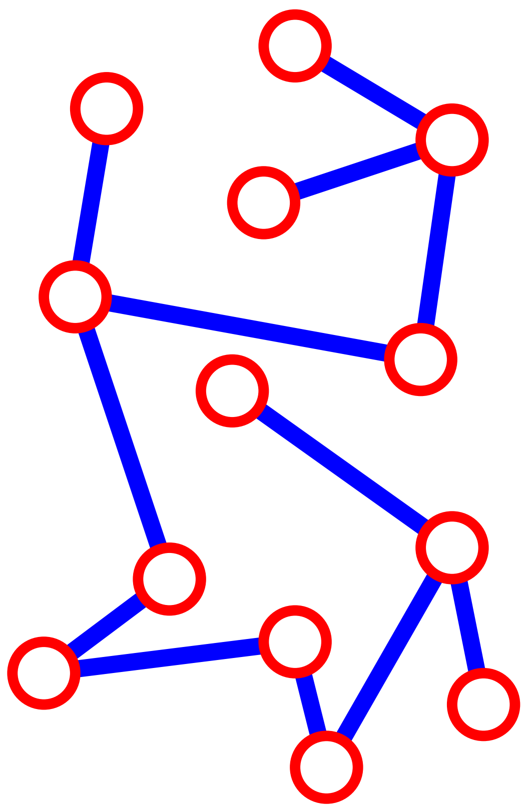
n = 5
m = 3

Spanning Tree

- A **spanning tree** of **G** is a subgraph which
  - is a tree
  - contains all vertices of **G**



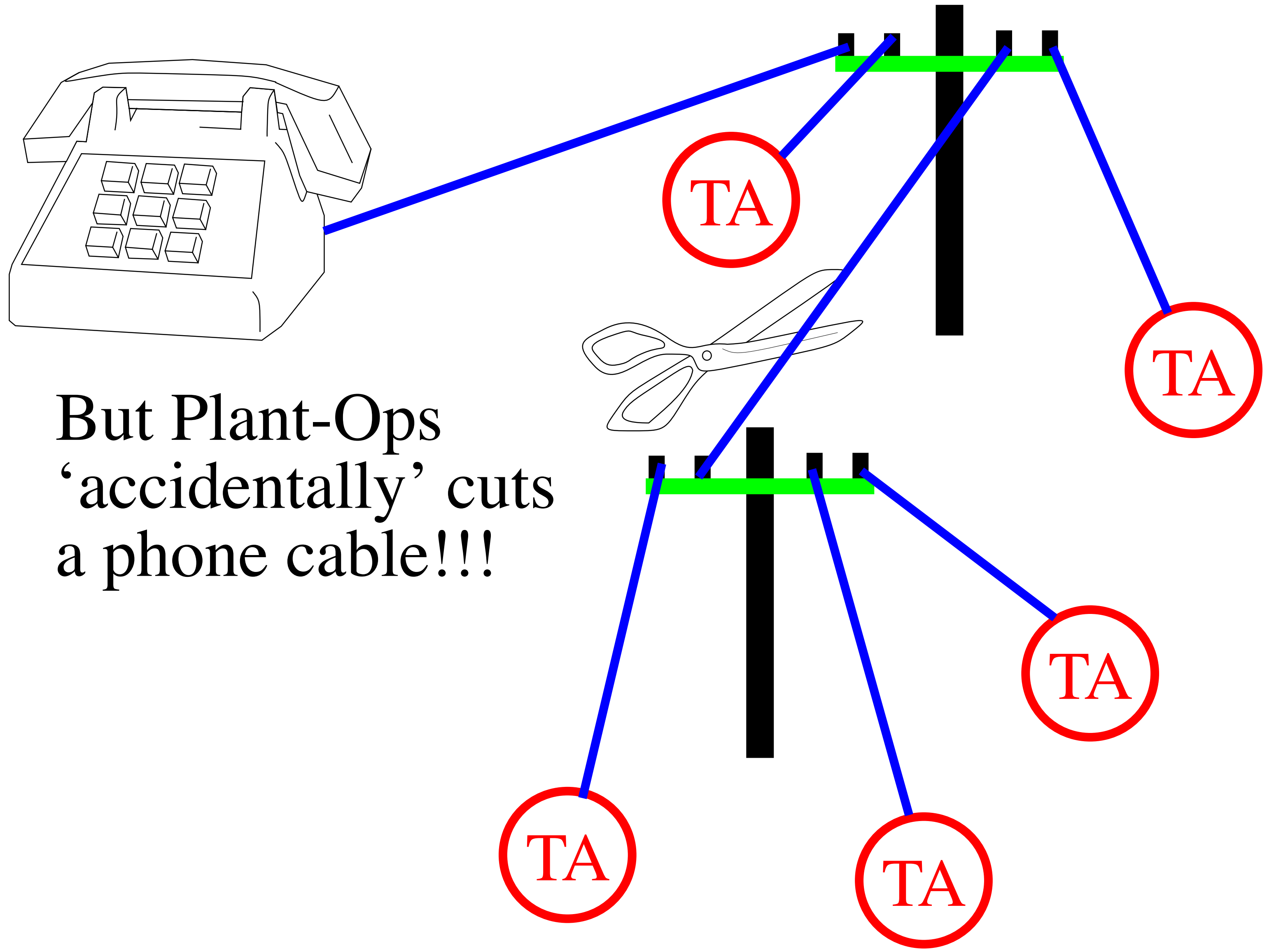
G



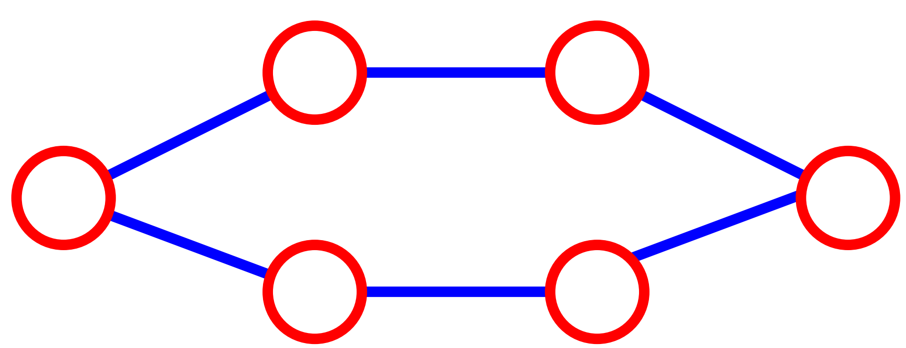
spanning tree of **G**

- Failure on any edge disconnects system (least fault tolerant)

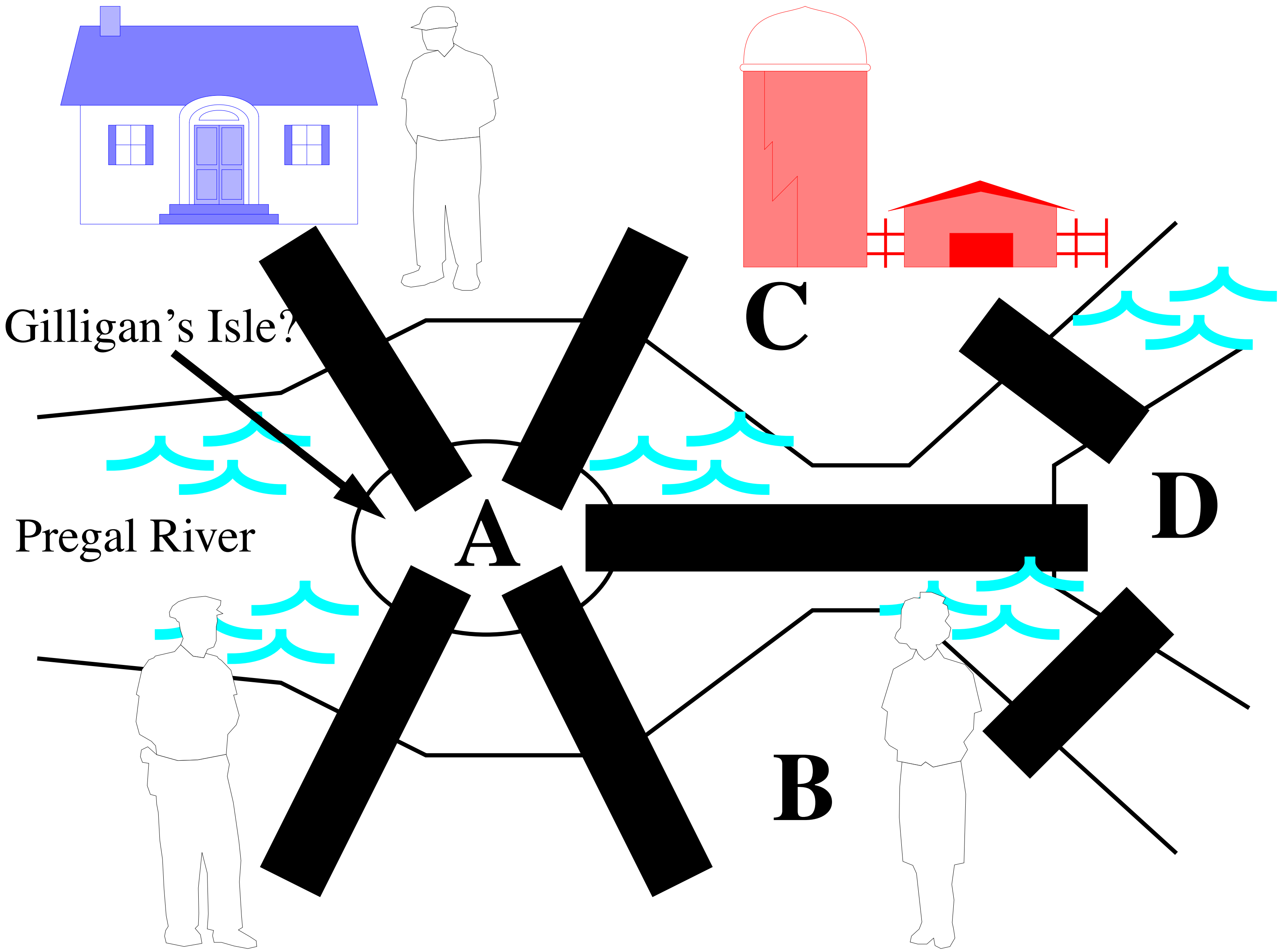
- Roberto wants to call the TA's to suggest an extension for the next program...



- One fault will disconnect part of graph!!
- A cycle would be more fault tolerant and only requires **n** edges



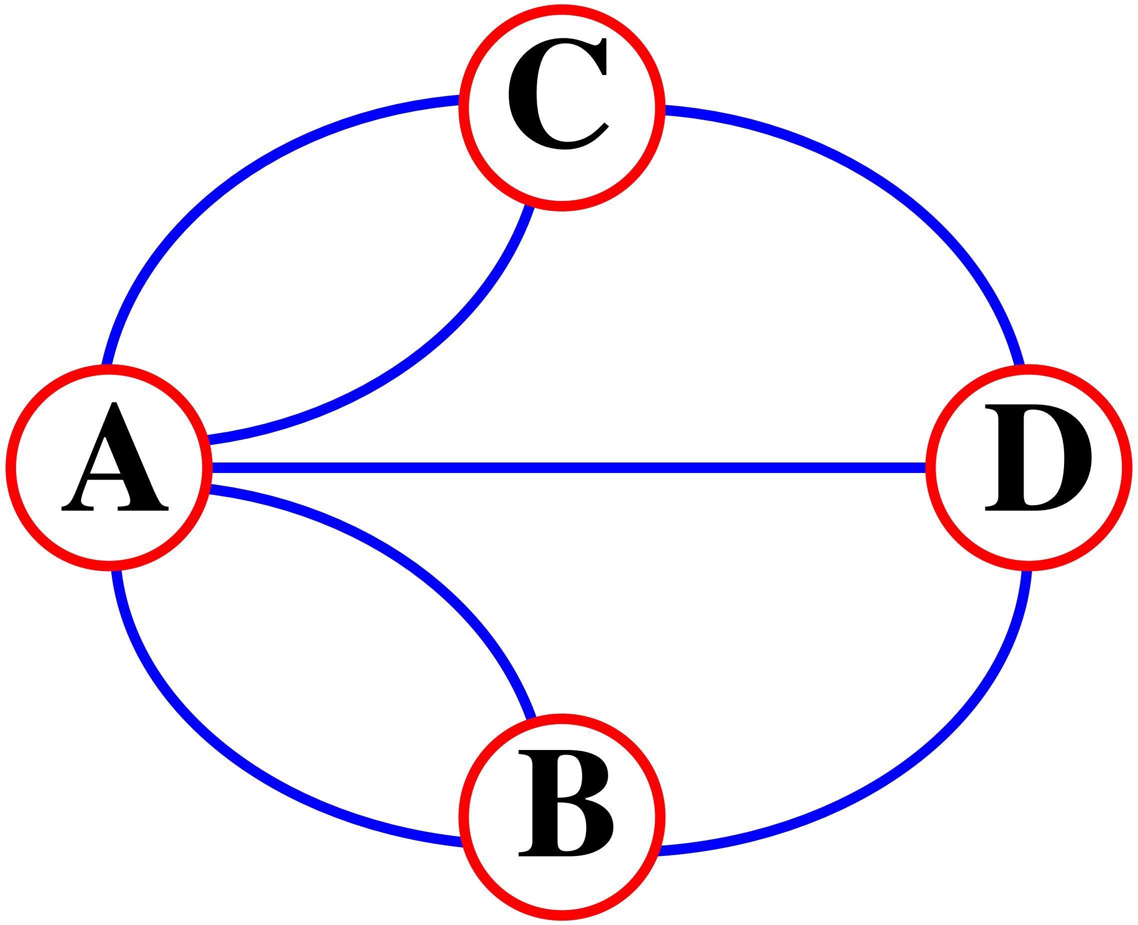
Koenigsberg



Can one walk across each bridge exactly once and return at the starting point?

- Consider if you were a UPS driver, and you didn't want to retrace your steps.
- In 1736, Euler proved that this is not possible

Graph Model(with parallel edges)



- Eulerian Tour: path that traverses every edge exactly once and returns to the first vertex
- Euler's Theorem: A graph has a Eulerian Tour if and only if all vertices have even degree

The Graph ADT

- The **Graph ADT** is a positional container whose positions are the vertices and the edges of the graph.

  - size() Return the number of vertices plus the number of edges of *G*.
  - isEmpty()
  - elements()
  - positions()
  - swap()
  - replaceElement()

The Graph ADT (contd.)

Notation: Graph *G*; Vertices *v*, *w*; Edge *e*; Object *o*

- numVertices()
  Return the number of vertices of *G*.
- numEdges()
  Return the number of edges of *G*.
- vertices() Return an enumeration of the vertices of *G*.
- edges() Return an enumeration of the edges of *G*.

The Graph ADT (contd.)

- directedEdges()

  Return an enumeration of all directed edges in G.
- undirectedEdges()

  Return an enumeration of all undirected edges in G.
- incidentEdges(v)

  Return an enumeration of all edges incident on v.
- inIncidentEdges(v)

  Return an enumeration of all the incoming edges to v.
- outIncidentEdges(v)

  Return an enumeration of all the outgoing edges from v.

The Graph ADT (contd.)

- opposite(*v*, *e*)

  Return an endpoint of *e* distinct from *v*
- degree(*v*)

  Return the degree of *v*.
- inDegree(*v*)

  Return the in-degree of *v*.
- outDegree(*v*)

  Return the out-degree of *v*.

More Methods ...

- adjacentVertices(*v*)

  Return an enumeration of the vertices adjacent to *v*.

- inAdjacentVertices(*v*)

  Return an enumeration of the vertices adjacent to *v* along incoming edges.

- outAdjacentVertices(*v*)

  Return an enumeration of the vertices adjacent to *v* along outgoing edges.

- areAdjacent(*v*,w)

  Return whether vertices *v* and w are adjacent.

More Methods ...

- endVertices(e)

  Return an array of size 2 storing the end vertices of e.

- origin(e)

  Return the end vertex from which e leaves.

- destination(e)

  Return the end vertex at which e arrives.

- isDirected(e)

  Return true iff e is directed.

Update Methods

- makeUndirected(e)
  Set e to be an undirected edge.
- reverseDirection(e)
  Switch the origin and destination vertices of e.
- setDirectionFrom(e, v)
  Sets the direction of e away from v, one of its end vertices.
- setDirectionTo(e, v)
  Sets the direction of e toward v, one of its end vertices.

Update Methods

- insertEdge(*v*, *w*, *o*)

  Insert and return an undirected edge between *v* and *w*, storing *o* at this position.

- insertDirectedEdge(*v*, *w*, *o*)

  Insert and return a directed edge between *v* and *w*, storing *o* at this position.

- insertVertex(*o*)

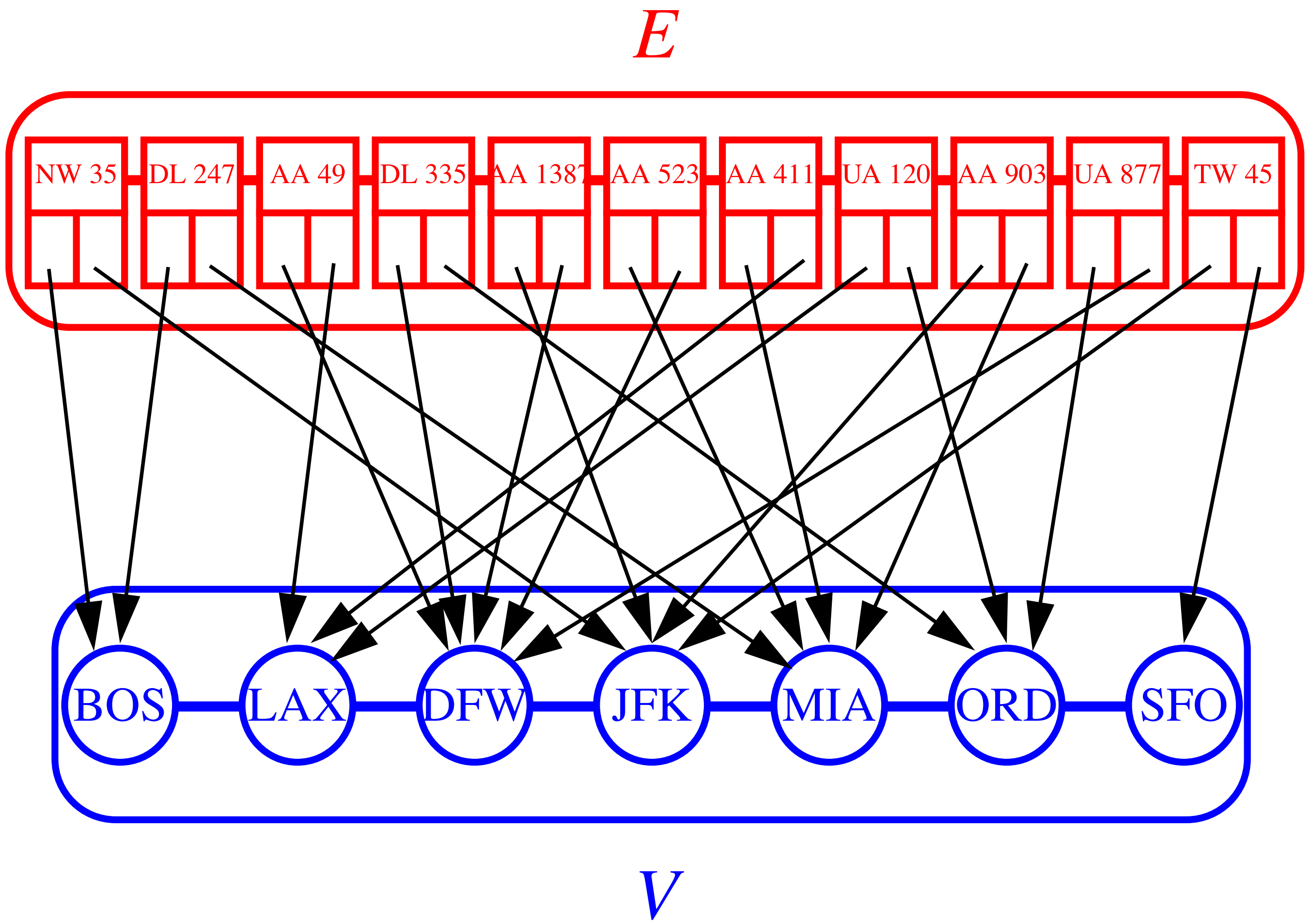
  Insert and return a new (isolated) vertex storing *o* at this position.

- removeEdge(*e*)

  Remove edge *e*.

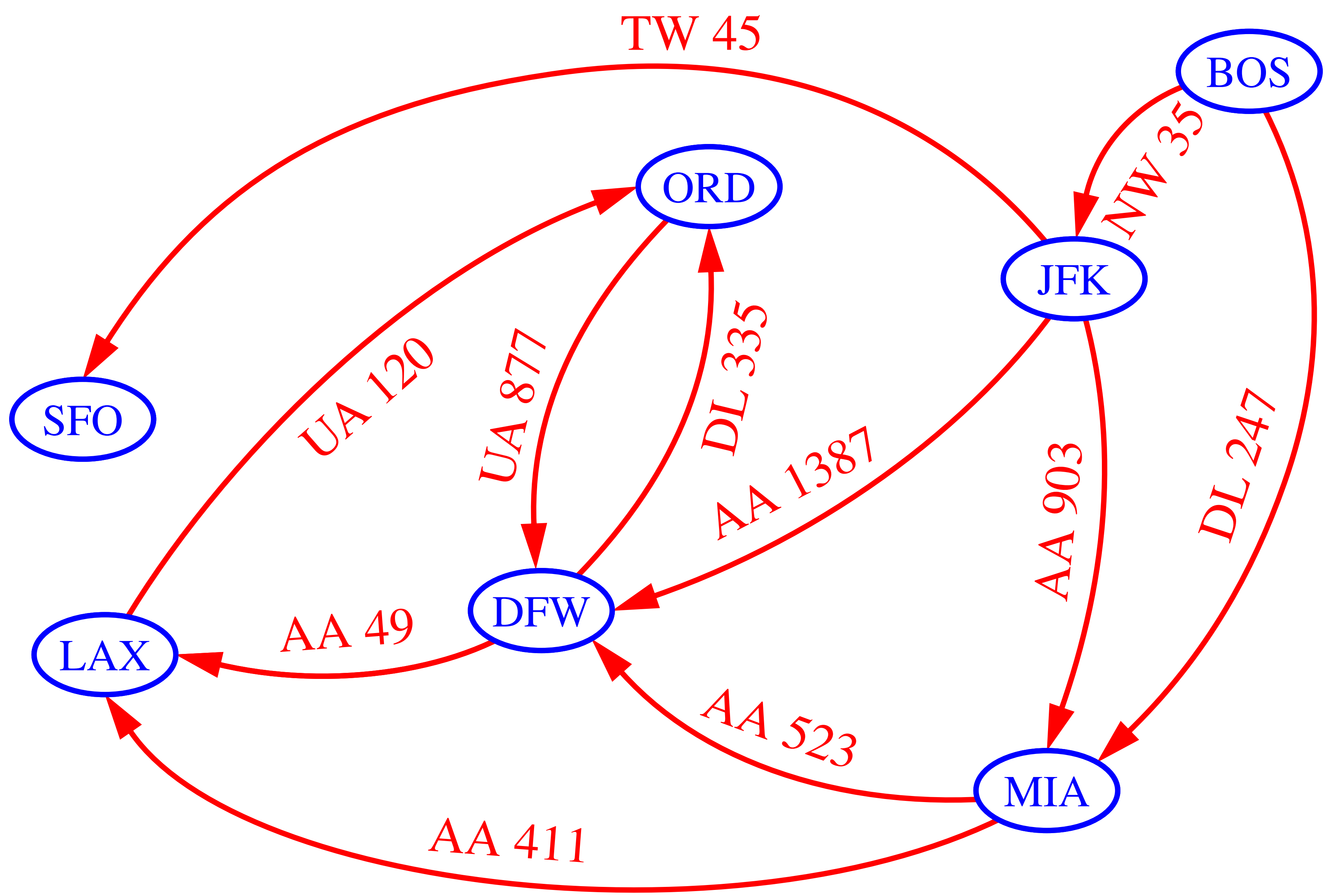
DATA STRUCTURES FOR GRAPHS

- Edge list
- Adjacency lists
- Adjacency matrix



Data Structures for Graphs

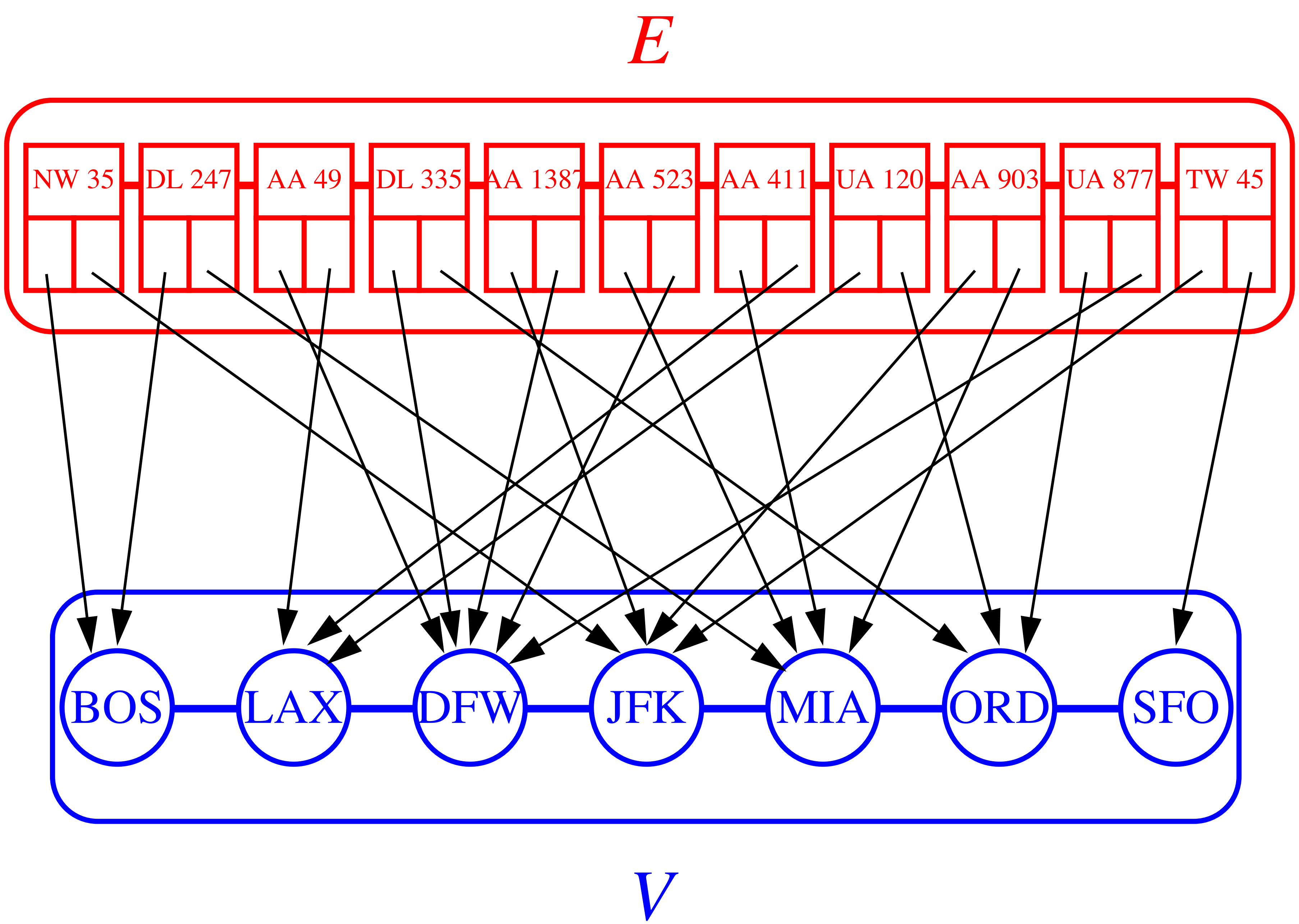
- To start with, we store the vertices and the edges into two containers, and each edge object has references to the vertices it connects.



- Additional structures can be used to perform efficiently the methods of the Graph ADT

Edge List

- The **edge list** structure simply stores the vertices and the edges into unsorted sequences.
- Easy to implement.
- Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence

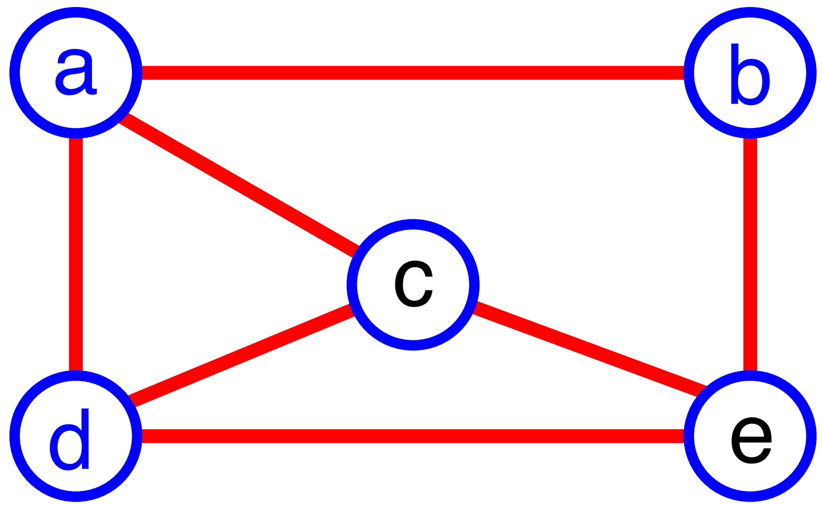


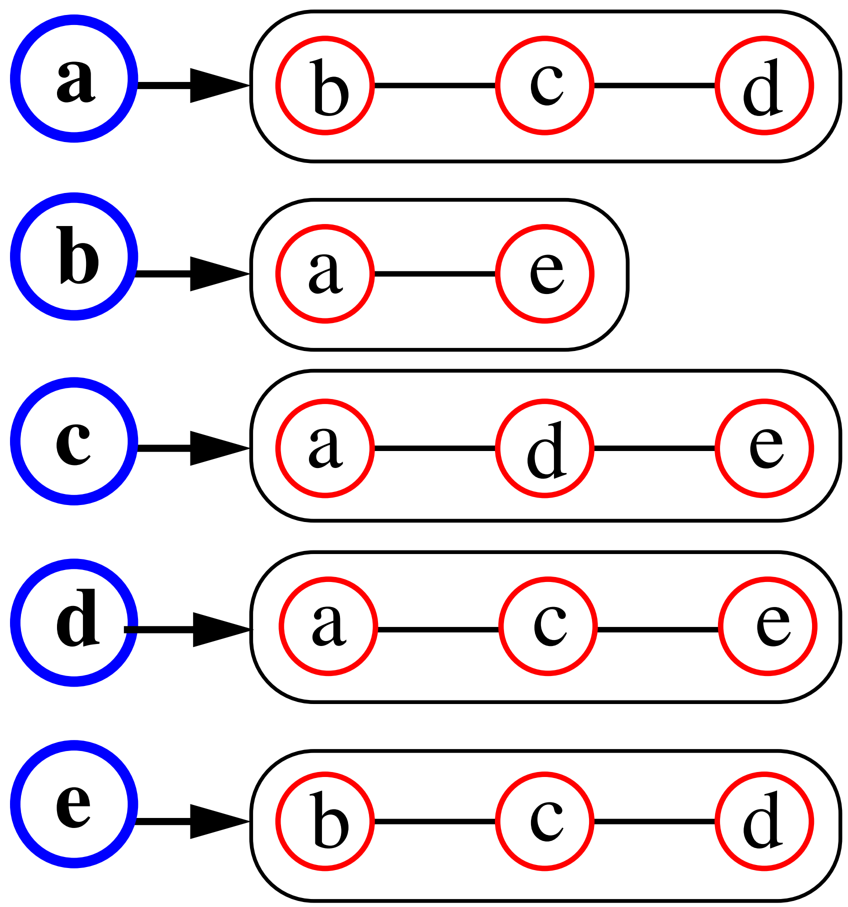
Performance of the Edge List Structure

Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices, areAdjacent, degree, inDegree, outDegree	O(m)
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
removeVertex	O(m)

Adjacency List (traditional)

- adjacency list of a vertex v:
  sequence of vertices adjacent to v
- represent the graph by the adjacency lists of all the vertices

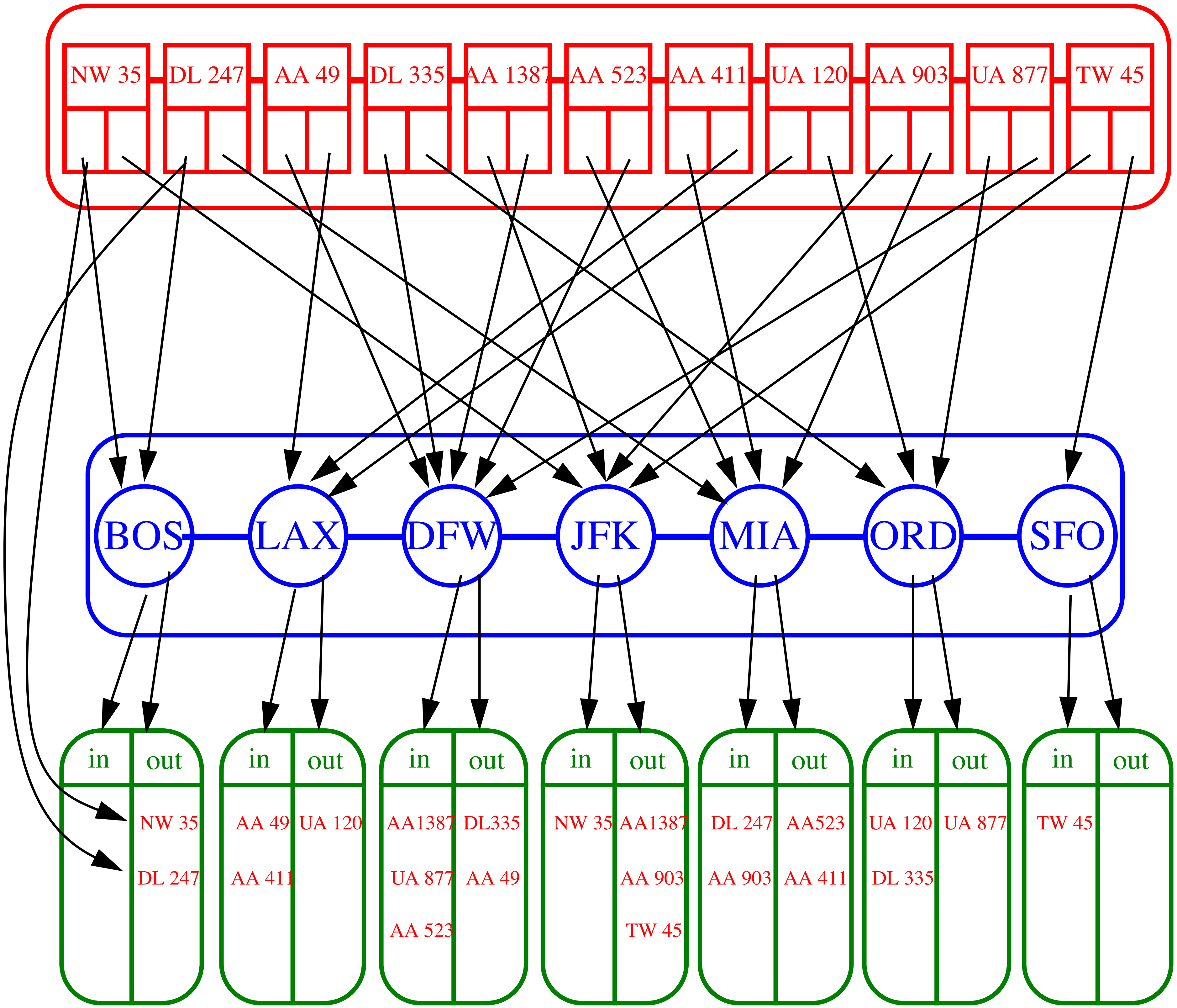




- Space = $\Theta(\mathbf{N} + \sum \text{deg}(\mathbf{v})) = \Theta(\mathbf{N} + \mathbf{M})$

Adjacency List (modern)

- The **adjacency list** structure extends the edge list structure by adding incidence containers to each vertex.

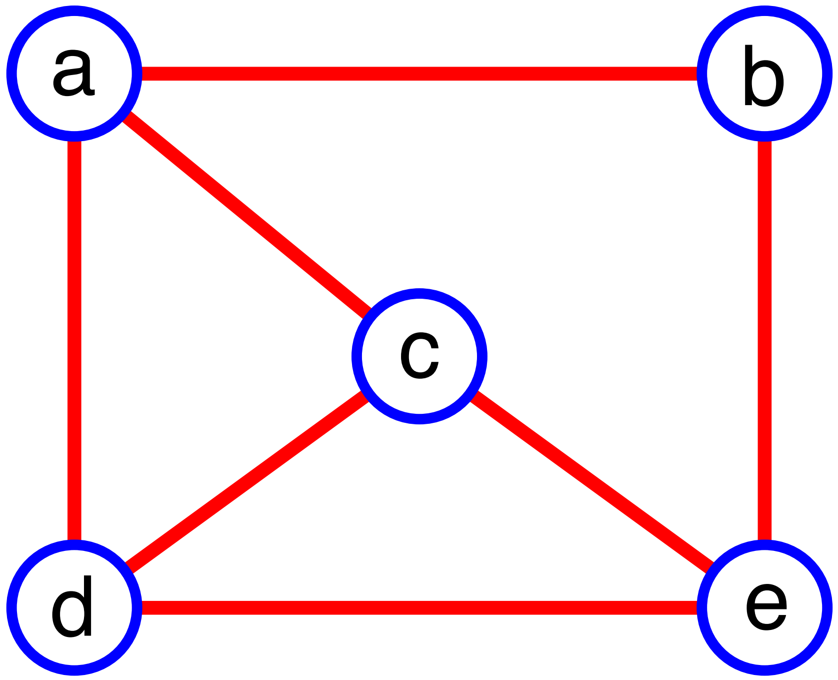


- The space requirement is O(n + m).

Performance of the Adjacency List Structure

Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v)	O(deg(v))
areAdjacent(u, v)	O(min(deg(u), deg(v)))
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection,	O(1)
removeVertex(v)	O(deg(v))

Adjacency Matrix (traditional)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- matrix M with entries for all pairs of vertices
- M[i,j] = true means that there is an edge (i,j) in the graph.
- M[i,j] = false means that there is no edge (i,j) in the graph.
- There is an entry for every possible edge, therefore:

  Space = $\Theta(\mathbf{N^2})$

Adjacency Matrix (modern)

- The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.

	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

BOS	DFW	JFK	LAX	MIA	ORD	SFO
0	1	2	3	4	5	6

- The space requirement is $O(n^2 + m)$

Performance of the Adjacency Matrix Structure

Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices,	O(n)
areAdjacent	O(1)
insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
insertVertex, removeVertex	$O(n^2)$

Winter 2016
COMP-250: Introduction to Computer Science

Lecture 16, March 10, 2016