CS 225

Data Structures

November 6 – Disjoint Sets Finale + Graphs

G Carl Evans

Disjoint Sets ADT

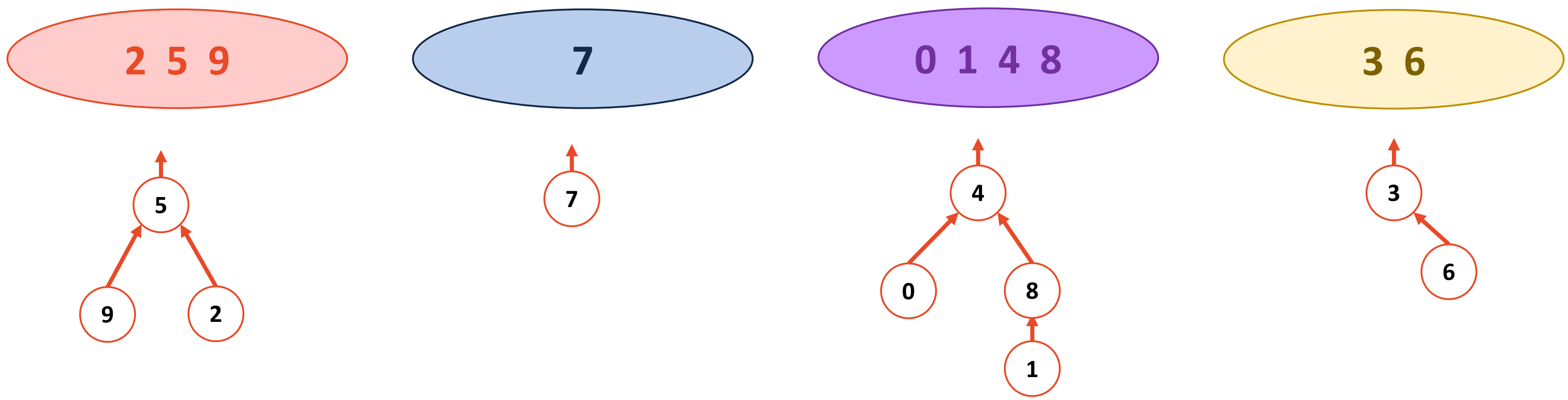
- Maintain a collection $S = \{s_0, s_1, \ldots s_k\}$

- Each set has a representative member.

- API:

```
void addelements(int num);
void union(int k1, int k2);
int find(int k);
```

Disjoint Sets



0	1	2	3	4	5	6	7	8	9
4	8	5	-1	-1	-1	3	-1	4	5

Disjoint Sets Find

```
int DisjointSets::find() {
  if ( s[i] < 0 ) { return i; }
  else { return _find( s[i] ); }
}
```

Running time?

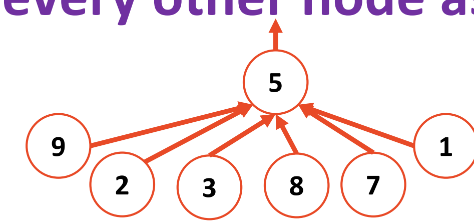
Structure: A structure similar to a linked list

Running time: O(h) < O(n)

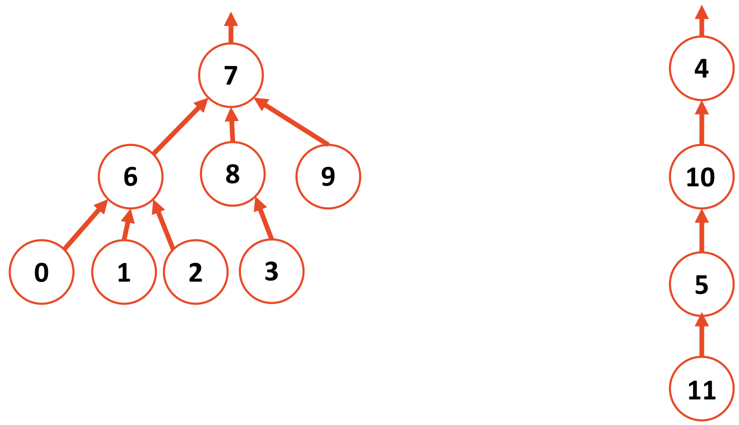
What is the ideal UpTree?

Structure: One root node with every other node as it's child

Running Time: O(1)



Disjoint Sets – Smart Union



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-3	7	7	4	5

Idea: *Keep the height of the tree as small as possible.*

Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-8	7	7	4	5

Idea: *Minimize the number of nodes that increase in height*

Both guarantee the height of the tree is:

Disjoint Sets Find and Union

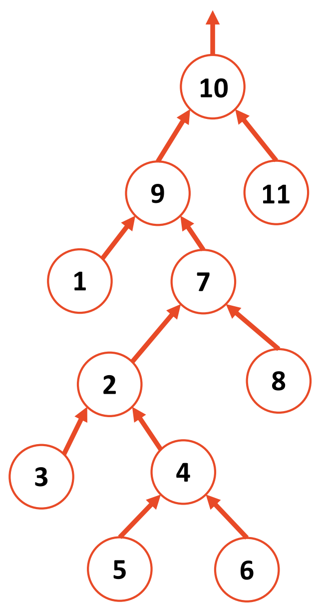
```
int DisjointSets::find(int i) {
  if ( arr_[i] < 0 ) { return i; }
  else { return _find( arr_[i] ); }
}
```

```
void DisjointSets::unionBySize(int root1, int root2) {
  int newSize = arr_[root1] + arr_[root2];

  // If arr_[root1] is less than (more negative), it is the larger set;
  // we union the smaller set, root2, with root1.
  if ( arr_[root1] < arr_[root2] ) {
    arr_[root2] = root1;
    arr_[root1] = newSize;
  }

  // Otherwise, do the opposite:
  else {
    arr_[root1] = root2;
    arr_[root2] = newSize;
  }
}
```

Path Compression



Disjoint Sets Find with Compression

```
int DisjointSets::find(int i) {
  // At root return the index
  if ( arr_[i] < 0 ) {
    return i;
  }

  // If not at the root recurse and on the return update parent
  // to be the root.
  else {
    int root = find( arr_[i] );
    arr_[i] = root;
    return root;
  }
}
```

Disjoint Sets Analysis

The **iterated log** function:

The number of times you can take a log of a number.

log*(n) =

0 , $n \le 1$

1 + log*(log(n)) , $n > 1$

What is **lg*(2^{65536})**?

Disjoint Sets Analysis

In an Disjoint Sets implemented with smart **unions** and path compression on **find**:

Any sequence of **m** **union** and **find** operations result in the worse case running time of O(____________),

where **n** is the number of items in the Disjoint Sets.

In Review: Data Structures

Array
- **Sorted Array**
- **Unsorted Array**
  - **Stacks**
  - **Queues**
  - **Hashing**
  - **Heaps**
    - **Priority Queues**
  - **UpTrees**
    - **Disjoint Sets**

Linked
- **Doubly Linked List**
- **Trees**
  - **BTree**
  - **Binary Tree**
    - **Huffman Encoding**
    - **kd-Tree**
    - **AVL Tree**

In Review: Data Structures

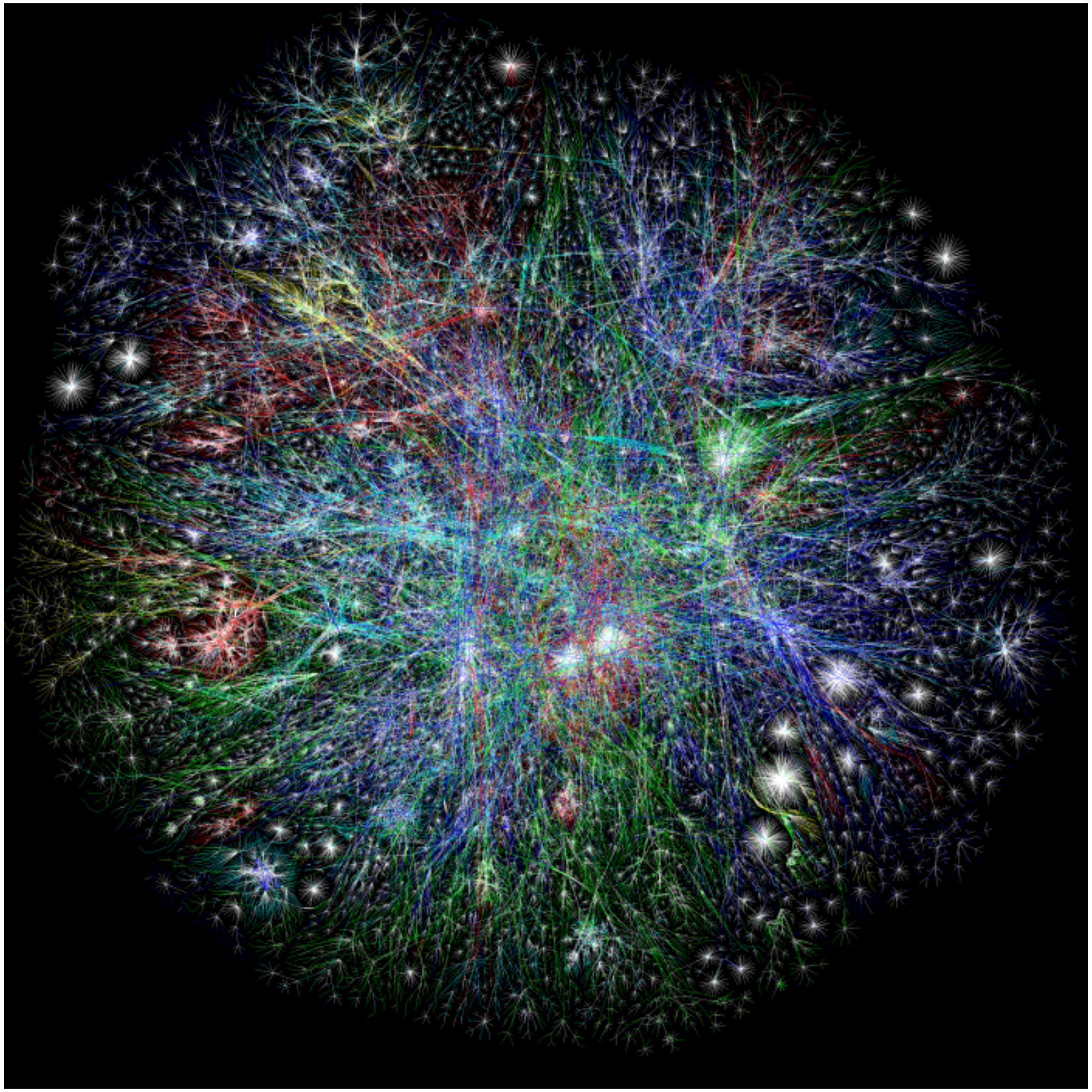
Array

- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
  - UpTrees
    - Disjoint Sets

Graphs

Linked

- Doubly Linked List
- Skip List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree

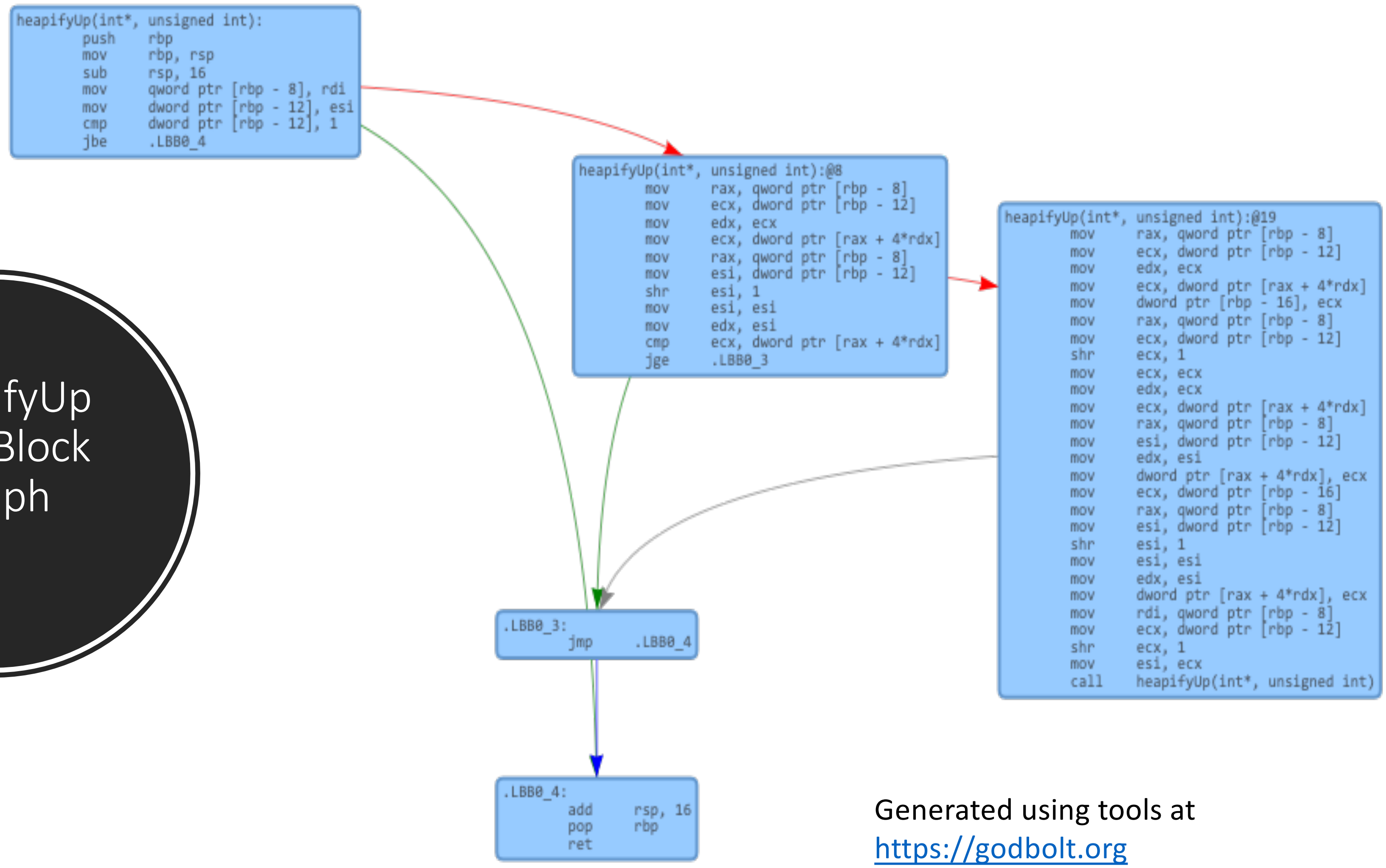


The Internet 2003

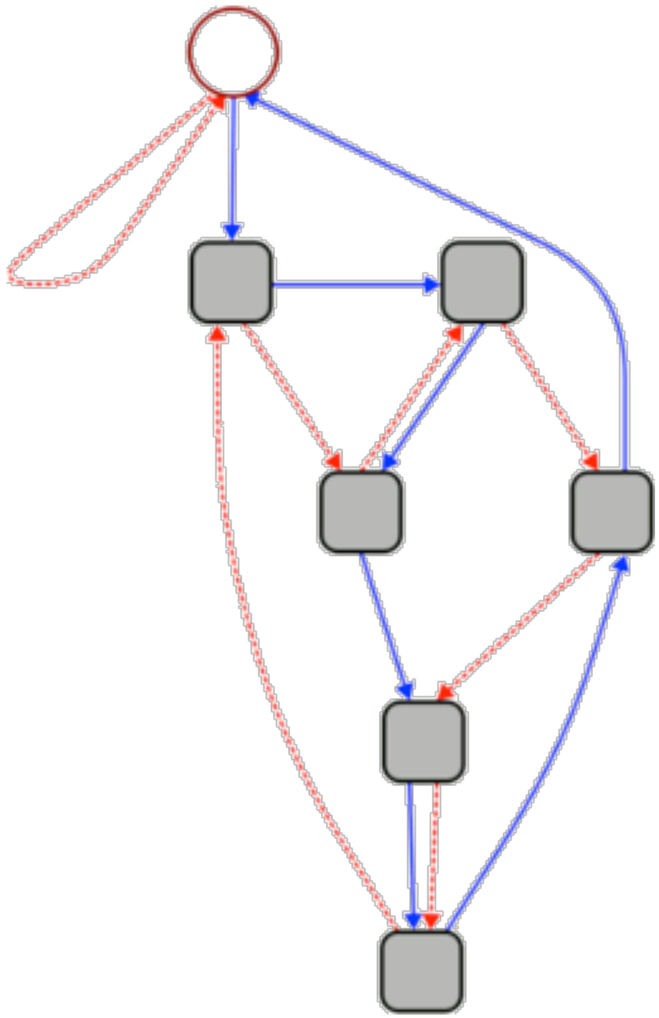
The OPTE Project (2003)

Map of the entire internet; nodes are routers; edges are connections.

HeapifyUp BasicBlock Graph



Generated using tools at https://godbolt.org



This graph can be used to quickly calculate whether a given number is divisible by 7.

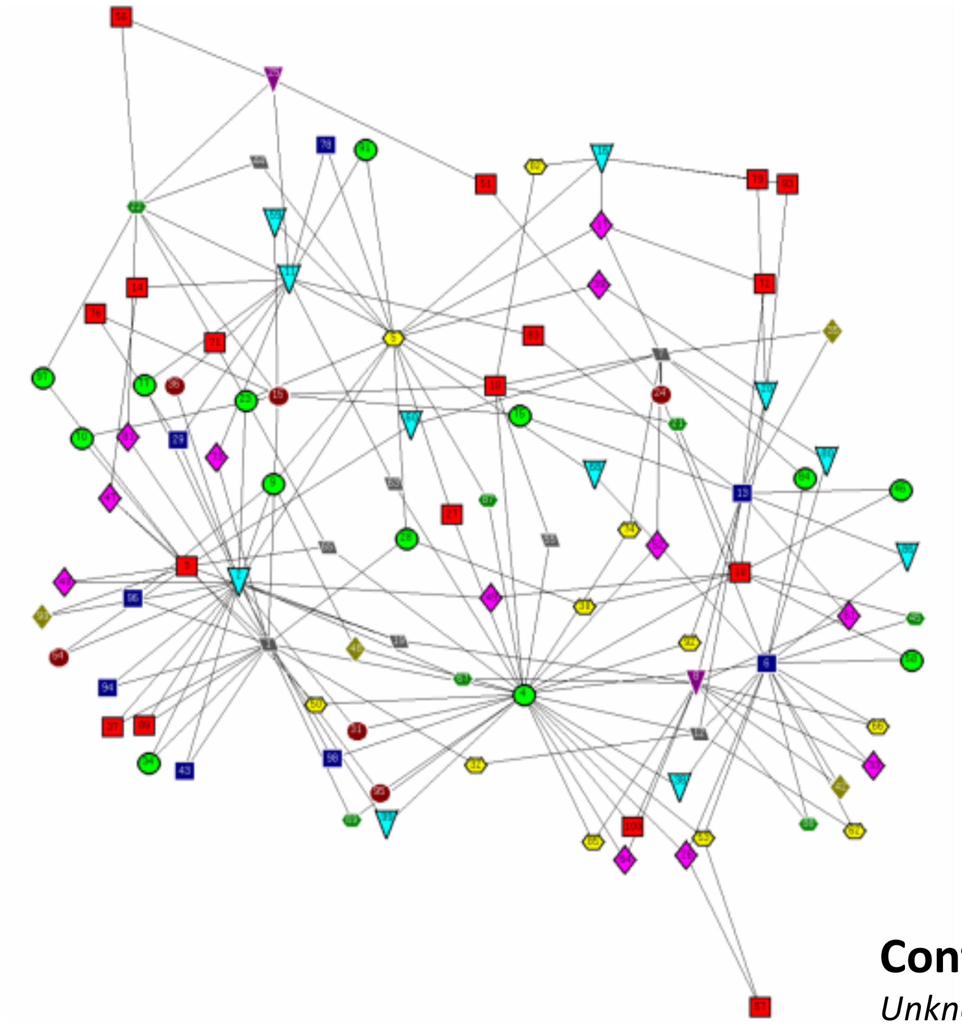
1. Start at the circle node at the top.
2. For each digit **d** in the given number, follow **d** blue (solid) edges in succession. As you move from one digit to the next, follow **1** red (dashed) edge.
3. If you end up back at the circle node, your number is divisible by 7.

3703

“Rule of 7”

Unknown Source

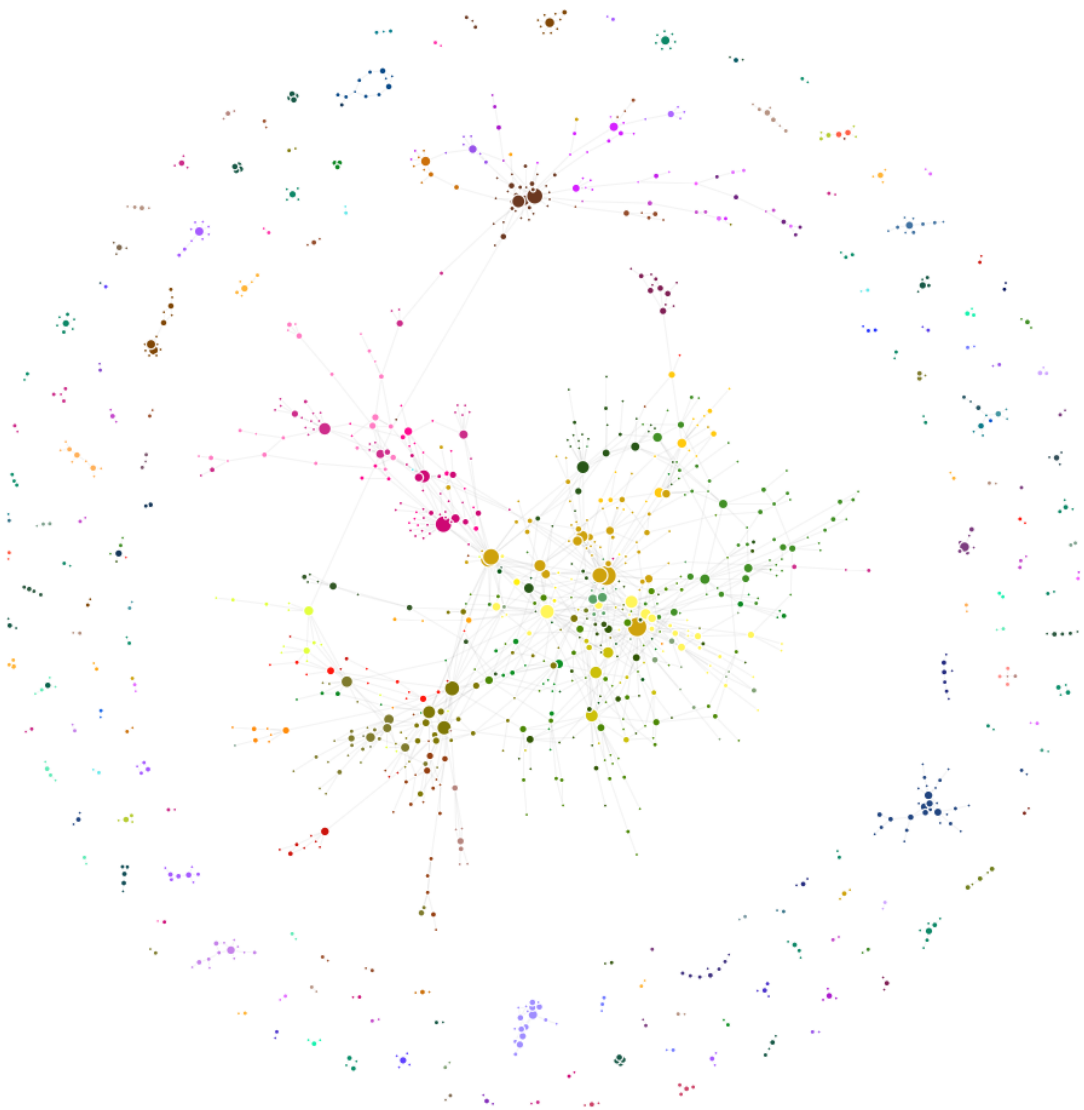
Presented by Cinda Heeren, 2016



Conflict-Free Final Exam Scheduling Graph

Unknown Source

Presented by Cinda Heeren, 2016

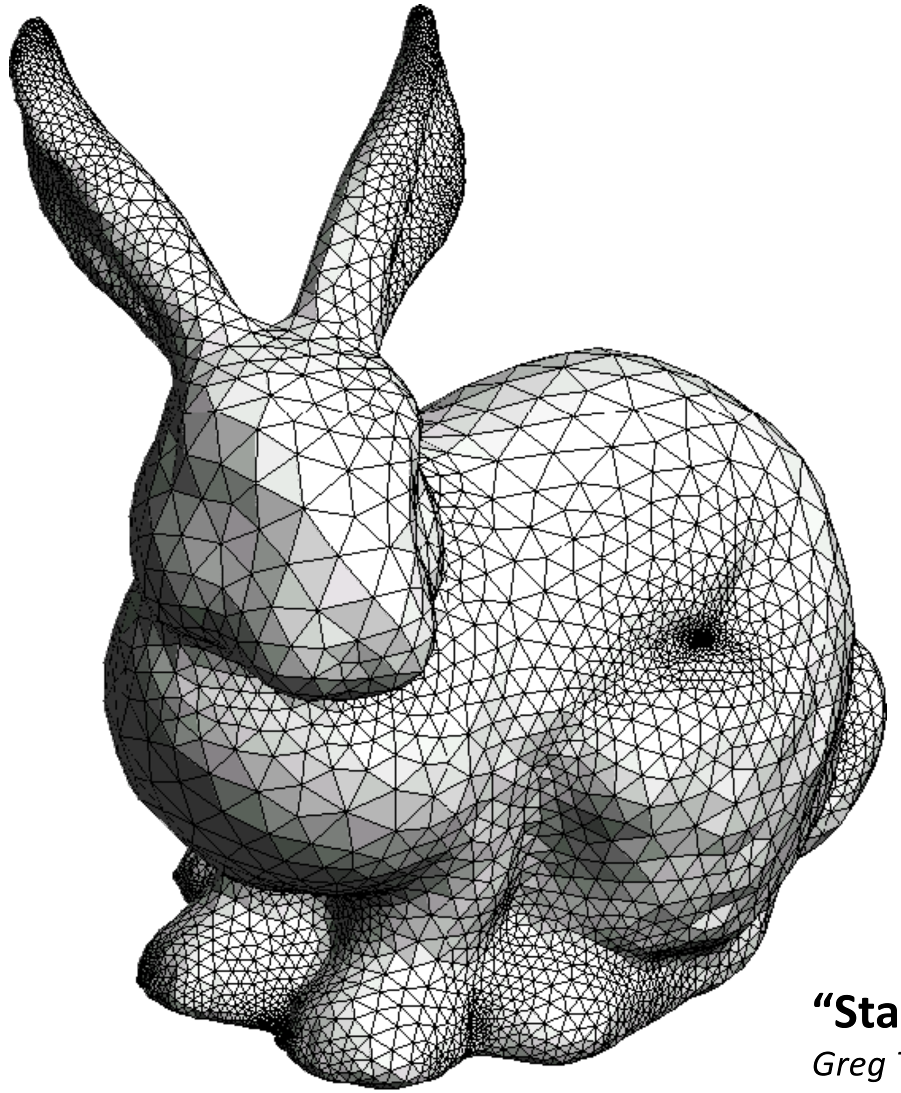


Class Hierarchy At University of Illinois Urbana-Champaign

A. Mori, W. Fagen-Ulmschneider, C. Heeren

Graph of every course at UIUC; nodes are courses, edges are prerequisites

http://waf.cs.illinois.edu/discovery/class_hierarchy_at_illinois/

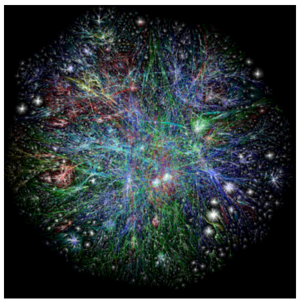


“Stanford Bunny”
Greg Turk and Mark Levoy (1994)

HAMLET

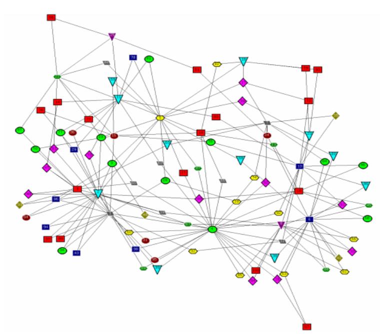
TROILUS AND CRESSIDA

Graphs



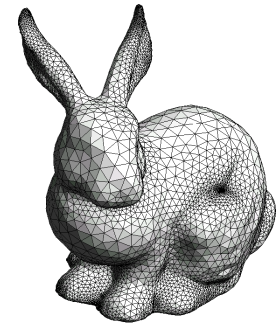
HAMLET

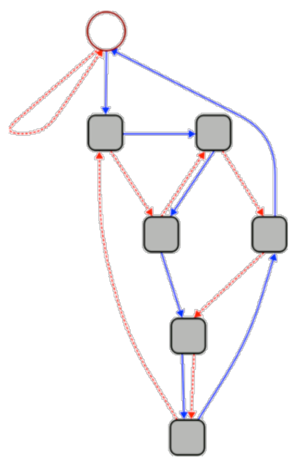
TROILUS AND CRESSIDA

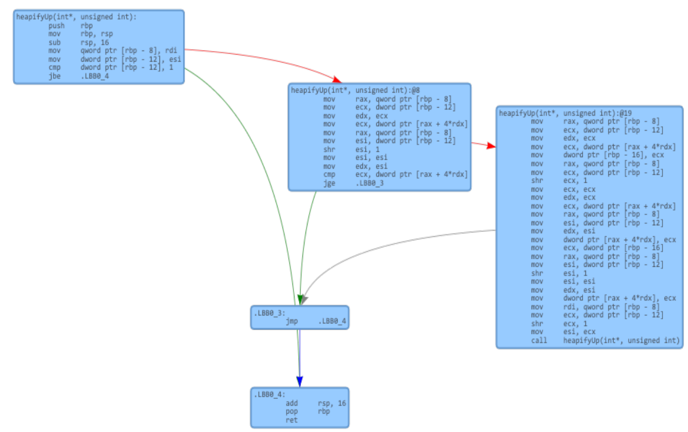


To study all of these structures:

1. A common vocabulary
2. Graph implementations
3. Graph traversals
4. Graph algorithms





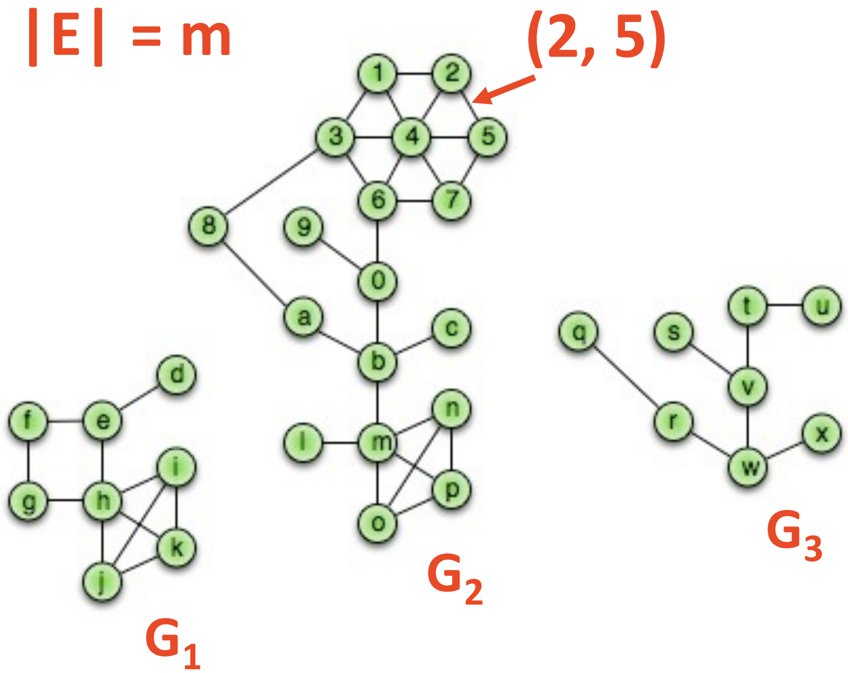


Graph Vocabulary

G = (V, E)

|V| = n

|E| = m



Incident Edges:
I(v) = { {x, v} in E }

Degree(v): |I|

Adjacent Vertices:
A(v) = { x : {x, v} in E }

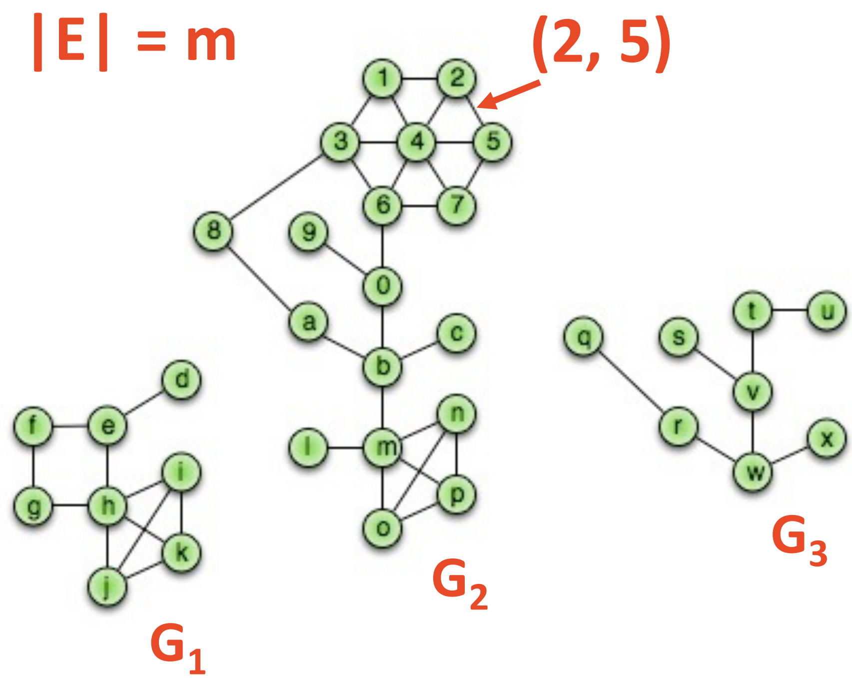
Path(G_2): Sequence of vertices connected by edges

Cycle(G_1): Path with a common begin and end vertex.

Simple Graph(G): A graph with no self loops or multi-edges.

Graph Vocabulary

G = (V, E)
|V| = n
|E| = m



Subgraph(G):
G' = (V', E'):
V' ∈ V, E' ∈ E, and
(u, v) ∈ E' → u ∈ V', v ∈ V'

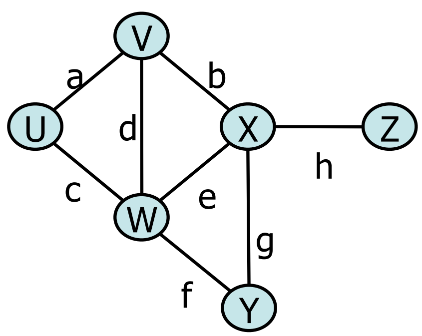
Complete subgraph(G)
Connected subgraph(G)
Connected component(G)
Acyclic subgraph(G)
Spanning tree(G)

Running times are often reported by **n**, the number of vertices, but often depend on **m**, the number of edges.

How many edges? **Minimum edges:**

Not Connected:

Connected*:



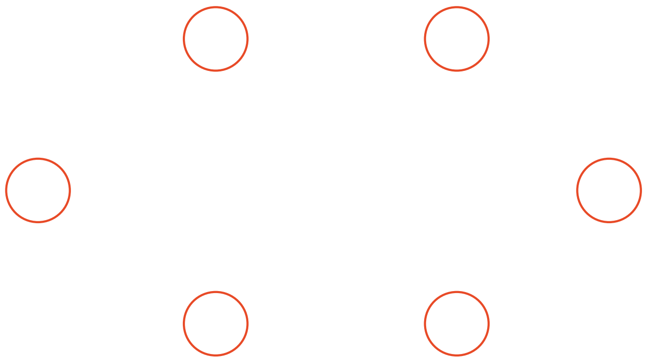
Maximum edges:

Simple:

Not simple:

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

Connected Graphs



Proving the size of a minimally connected graph

Theorem:
Every connected graph **G=(V, E)** has at least **|V|-1** edges.

Thm: Every connected graph **G=(V, E)** has at least **|V|-1** edges.

Proof: Consider an arbitrary, connected graph **G=(V, E)**.

Suppose |V| = 1:

Definition: A connected graph of 1 vertex has 0 edges.

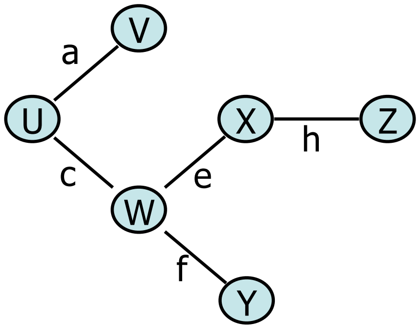
Theorem: |V|-1 edges ➔ 1-1 = 0.

Inductive Hypothesis: For any **j < |V|**, any connected graph of **j** vertices has at least **j-1** edges.

Suppose |V| > 1:

1. Choose any edge:

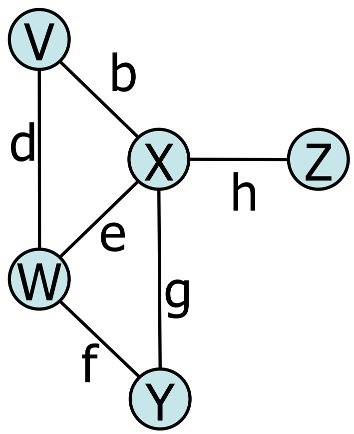
2. Partition:



Graph ADT

Data:

- Vertices
- Edges
- Some data structure maintaining the structure between vertices and edges.



Functions:

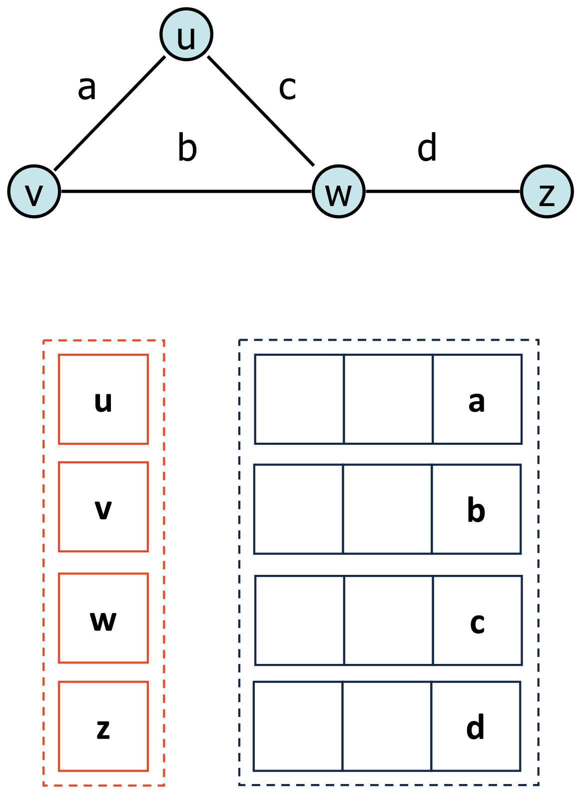
- insertVertex(K key);
- insertEdge(Vertex v1, Vertex v2, K key);

- removeVertex(Vertex v);
- removeEdge(Vertex v1, Vertex v2);

- incidentEdges(Vertex v);
- areAdjacent(Vertex v1, Vertex v2);

- origin(Edge e);
- destination(Edge e);

Graph Implementation: Edge List



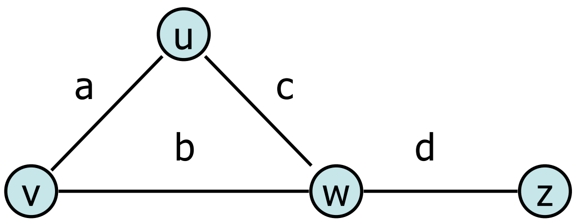
insertVertex(K key);

removeVertex(Vertex v);

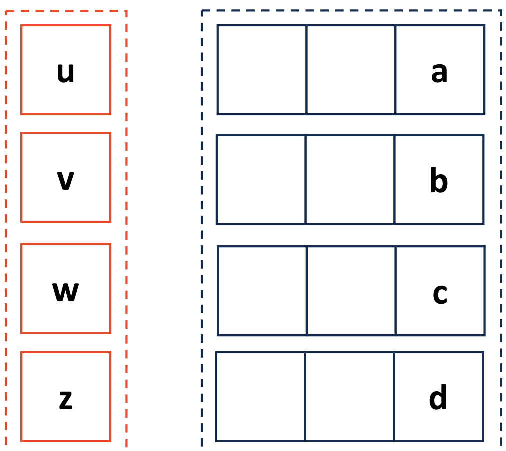
areAdjacent(Vertex v1, Vertex v2);

incidentEdges(Vertex v);

Graph Implementation: Adjacency Matrix



insertVertex(K key);
removeVertex(Vertex v);
areAdjacent(Vertex v1, Vertex v2);
incidentEdges(Vertex v);



	u	v	w	z
u				
v				
w				
z				