#32: Disjoint Sets Finale + Graphs Intro

November 6, 2019 · *G Carl Evans*

Smart Union Implementation:

DisjointSets.cpp (partial)

```
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;
  }
}
```

How do we improve this?

Running Time:

- Worst case running time of find(k)**:**
- Worst case running time of union(r1, r2), given roots:
- New function: “Iterated Log”:

  log*(n) :=

- Overall running time:
  - A total of **m** union/find operation runs in:

A Review of Major Data Structures so Far

Array-based	List/Pointer-based
- Sorted Array	- Singly Linked List
- Unsorted Array	- Doubly Linked List
- Stacks	- Skip Lists
- Queues	- Trees
- Hashing	- BTree
- Heaps	- Binary Tree
- Priority Queues	- Huffman Encoding
- UpTrees	- kd-Tree
- Disjoint Sets	- AVL Tree

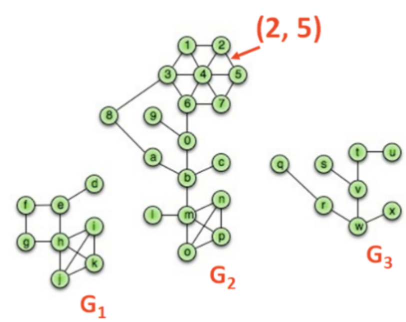
Motivation:

Graphs are awesome data structures that allow us to represent an enormous range of problems. To study these problems, we need:

1. A common vocabulary to talk about graphs
2. Implementation(s) of a graph
3. Traversals on graphs
4. Algorithms on graphs

Graph Vocabulary

Consider a graph **G** with vertices **V** and edges **E**, **G=(V,E)**.



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.

Subgraph(G): **G’ = (V’, E’)**:

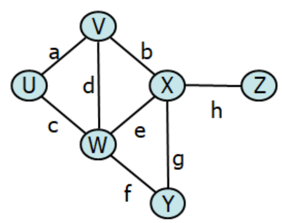
$V' \in V$, $E' \in E$, and $(u, v) \in E \rightarrow u \in V', v \in V'$

Graphs that we will study this semester include:
- Complete subgraph(G)
- Connected subgraph(G)
- Connected component(G)
- Acyclic subgraph(G)
- Spanning tree(G)

Size and Running Times

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

For arbitrary graphs, the <u>**minimum**</u> number of edges given a graph that is:



Not Connected:

Minimally Connected:*

The <u>**maximum**</u> number of edges given a graph that is:

Simple:

Not Simple:

The relationship between the degree of the graph and the edges:

Proving the Size of a Minimally Connected Graph

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

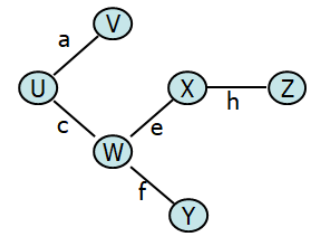
<u>**Proof of Theorem**</u>

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

<u>**Suppose |V| = 1:**</u>

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

<u>**Suppose |V| > 1:**</u>



1. Choose any vertex:
   -
   -
2. Partitions:
   -

Graph ADT

Data	Functions

1. Vertices
 	
2. Edges
 	
3. Some data structure maintaining the structure between vertices and edges.
	`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);`

CS 225 – Things To Be Doing:
1. Exam 4 next Friday; **Practice Exam Available Today!** 2. mp_mazes EC due Monday 3. Daily POTDs are ongoing!