Union-Find

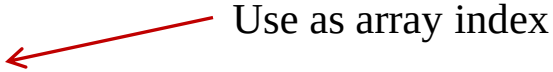
A Data Structure for Disjoint Set Operations; Applications

What is Disjoint-set data structure?

- Represents the mathematical concept of "Set".
- A disjoint-set data structure, also called a union–find data structure or merge–find set.
- A disjoint-set data structure that keeps track of a set of elements partitioned into a number of disjoint or non-overlapping subsets.
- It provides near-constant-time operations to add new sets, to merge existing sets, and to determine whether elements are in the same set.
- Plays a key role in Kruskal's algorithm to find the minimum spanning tree of a graph.
- This can also be used to detect cycle in the graph.
- Many other applications.

Problem Statement

- **Union-find** applications involve manipulating **objects** of all types.
  - Computers in a network.
  - Web pages on the Internet.
  - Transistors in a computer chip.
  - Variable name aliases.
  - Pixels in a digital photo.
  - Metallic sites in a composite system.
- When programming, convenient to name them 0 to N-1 (not always!)
  - Hide details not relevant to union-find.
  - Integers allow quick access to object-related info.
  - Could use symbol table to translate from object names



Union – Find Structure

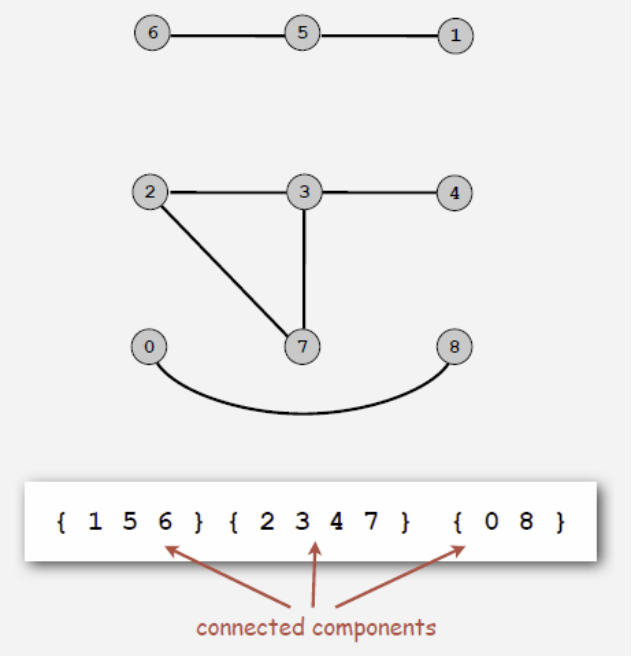
- A partition or union-find structure is a data structure supporting a **collection of disjoint sets**. We define the methods for this structure assuming we have a constant time way to access a node associated with an item, e. For instance, items could themselves be nodes or we could maintain some kind of lookup table or map for finding the node associated with an item, e, in constant time. Given such an ability, the methods support the following operations:
  - **make-Set**(e): Create a singleton set containing the element e and name this set "e".
  - **union**(A,B): Update A and B to create A ∪ B, naming the result as "A" **or** "B". Thus, each set forms a tree, which is known by the root.
  - **find**(e): Return the name of the set containing the element e.
- We refer to an implementation supporting these methods as a union-find structure.
- **Dynamic connectivity**. Given an initial empty graph G on n nodes, support the following queries:
  - ADD-EDGE(u, v). Add an edge [undirected] between nodes u and v. **union operation**
  - IS-CONNECTED(u, v). Is there a path between u and v? **find operations**
- Original motivation: Compiling EQUIVALENCE, DIMENSION, and COMMON statements in Fortran (1964)

Modeling the objects

- **Union-find** applications involve manipulating **objects** of all types.
  - Computers in a network, Web pages on the Internet, Transistors in a computer chip, Variable name aliases, Pixels in a digital photo, Metallic sites in a composite system.
- When programming, convenient to name them 0 to N-1 (not always!)
  - Hide details not relevant to union-find.
  - Integers allow quick access to object-related info.
  - use symbol table to translate from object names

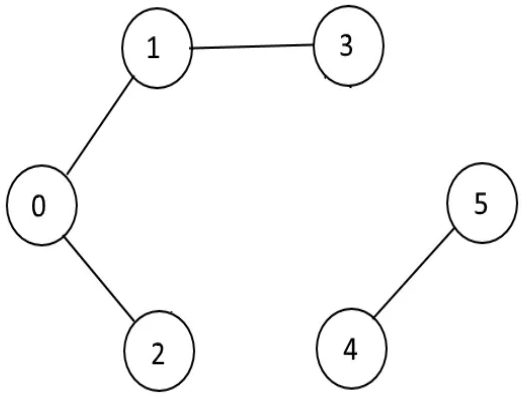
Modeling the connections:

- Transitivity. If p is connected to q and q is connected to r, then p is connected to r.
- Connected components. Maximal set of objects that are mutually connected.
- Being connected is an Equivalence Relation



How Disjoint Set is constructed

- A disjoint-set forest consists of a number of elements each of which stores an id, and a parent pointer.
- The parent pointers of elements are arranged to form one or more trees, each tree representing a different set.
- If an element's parent pointer points to no other element, then the element is the root of a tree and is the representative member of its set (called the identifier of the subset representing the tree).
- A set may consist of only a single element. However, if the element has a parent, the element is part of whatever set is identified by following the chain of parents upwards until a representative element (identifier) (one without a parent) is reached at the root of the tree.



Disjoint Sets

Set Id: 0, elements: [0, 1, 2, 3]

Set Id: 4, elements: [4, 5]

Sometimes, edges can be shown to be directed towards the parent; root node has its parent pointer pointing to itself.

How Disjoint Set is constructed: Example

Consider the previous example: V = {0, 1, 2, 3, 4, 5} and the given subsets are {0, 1, 2, 3} and {4, 5}.

Step 1: for i = 0 to 5 makeset (i) ⇒ id(i) = i for all i.

i	0	1	2	3	4	5
Id(i)	0	1	2	3	4	5

Step 2: Union (0, 1) ⇒ the id array is [0 0 2 3 4 5]

i	0	1	2	3	4	5
Id(i)	0	0	2	3	4	5

Step 3: Union (0, 2), Union(0, 3)

i	0	1	2	3	4	5
Id(i)	0	0	0	0	4	5

Step 4:Union (4, 5)

i	0	1	2	3	4	5
Id(i)	0	0	0	0	4	4

Union Find Abstractions

Simple model captures the essential nature of connectivity.

- Objects.

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

grid points

- Disjoint sets of objects.

```
0  1  { 2 3 9 }  { 5 6 }  7  { 4 8 }
```

subsets of connected grid points

Consider a grid to get a feel; think of an equivalence class

- Find query: are objects 2 and 9 in the same set?

```
0  1  { 2 3 9 }  { 5-6 }  7  { 4-8 }
```

are two grid points connected?

Think of a maze, or a connected component in a graph

- Union command: merge sets containing 3 and 8.

```
0  1  { 2 3 4 8 9 }  { 5-6 }  7
```

add a connection between two grid points

Union Find Abstractions

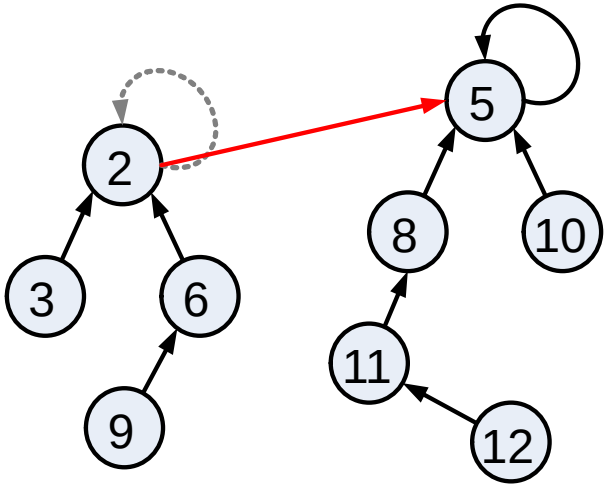
- Objects.
- Disjoint sets of objects.
- Find queries: are two objects in the same set?
- Union commands: replace sets containing two items by their union

Goal. Design efficient data structure for union-find.

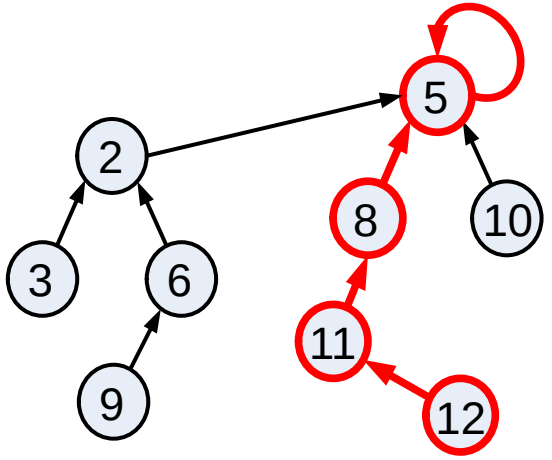
- Find queries and union commands may be intermixed.
- Number of operations M can be huge.
- Number of objects N can be huge.

Union – Find Operations

To do a union, simply make the root of one tree point to the root of the other.



To do a find, follow set name pointers from the starting node until reaching a node whose set-name pointer refers back to itself.



In Search of Efficient Data Structures

- **Approach 1:**
  - Keep the elements in the form of an array, where: A[i] stores the current set ID for element i. Assume N is the number of elements and M is the number of operations.
  - Find() will take O(1) time
  - Union() could take up to O(N) time in the worst case Bad
  - A sequence of m (union and find) operations could take O(M × N) in the worst case!!
- **Approach 2:**
  - Keep all equivalence sets in separate linked lists: 1 linked list for every set ID
  - Union() will take O(1) time [Assuming doubly linked lists]
  - Find can take O(N) time in the worst case; we can improve to $\Omega(\log_2 n)$ using balanced binary trees. Not good
  - A sequence of M (Arbitrary unions and finds) operations takes $\Omega(M \log_2 N)$.
- **How to improve?**
  - Keep all equivalence sets in separate trees: One tree for every set, **albeit a special kind.**
  - We will see the processes informally first.

The Union-Find data structure for n elements is a forest of k trees, where $1 \leq k \leq n$

Definitions

- $U \Rightarrow$ set of n elements and $S_i \Rightarrow$ a subset of U.
- S_1 and S_2 are disjoint if $S_1 \cap S_2 = \emptyset$.
- Maintains a dynamic collection $S_1, S_2, \ldots S_k$ of disjoint sets which together cover U.
- Each set is identified by a representative x.
- A set of algorithms that operate on this data structure is often referred to as a UNION-FIND algorithm.

- Each set is represented by a rooted tree, pointer towards root.
- The element in the root node is the representative of the set.
- Parent pointer $p(x)$ denotes the parent of node x.
- Two main operations.
  - FIND(x).
  - UNION(x, y).

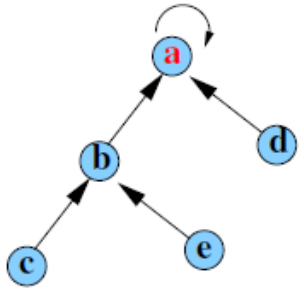
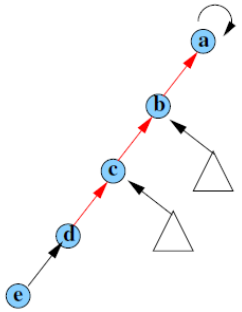


Figure: $S_i = \{a, b, c, d, e\}$.

The arrows really represent parent information, not to be confused with directed edges.

FIND and UNION

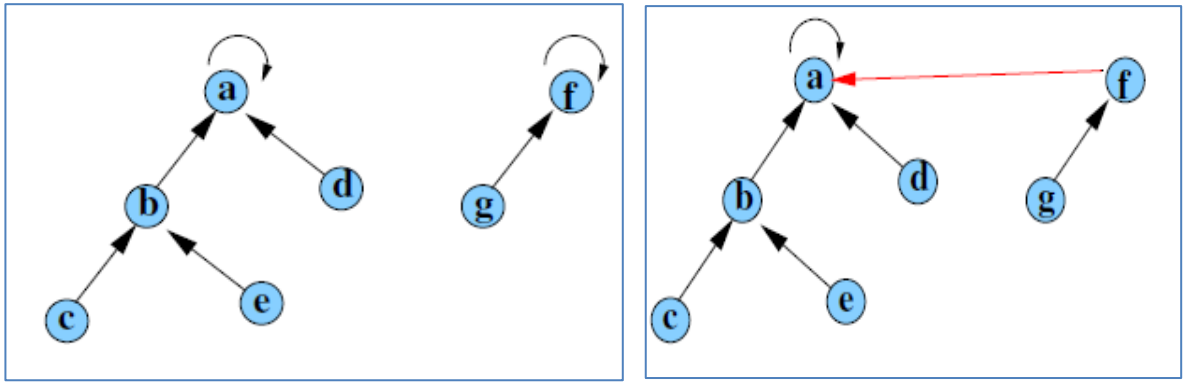
- **Find (x) :**
- To which set does a given element x belong) ⇒ Find(x)
- Returns the root (representative) of the set that contain x



Find (d)

UNION (x, y)

- Create a new set from the union of two existing sets containing x and y ⇒ Union(x; y).
- Change the parent pointer of one root to the other one.



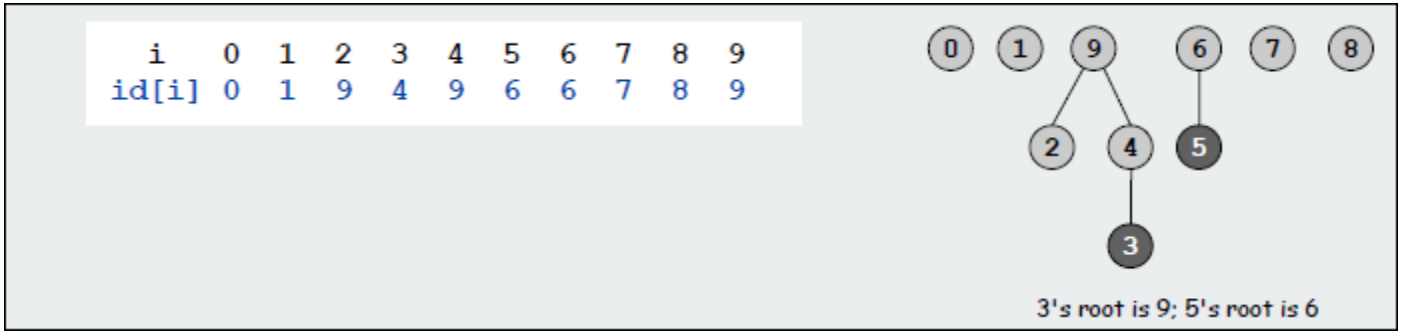
UNION (c, g): (1) Find (c) ⇒ find (c) says that the element c belongs to the set named a; (2) find (g) says g belongs to the set named f; (3) union (c,g) creates a new larger set named a by updating the parent pointer of f.

Quick Union [Lazy Approach]

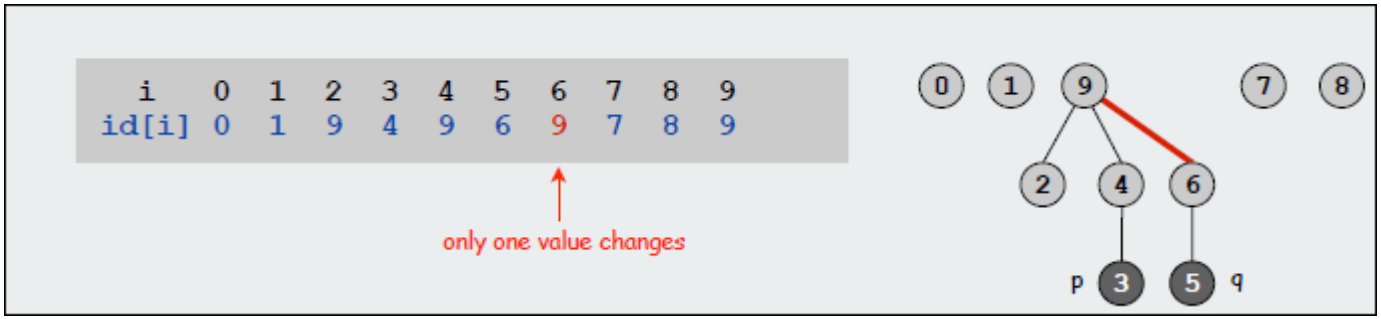
- **Data structure**.
  - Integer array id[] of size N.
  - Interpretation: id[i] is parent of i.
  - Root of i is id[id[id[...id[i]...]]].

Keep going until doesn't change; this may be expensive.

- **Find**. Check if p and q have the same root.



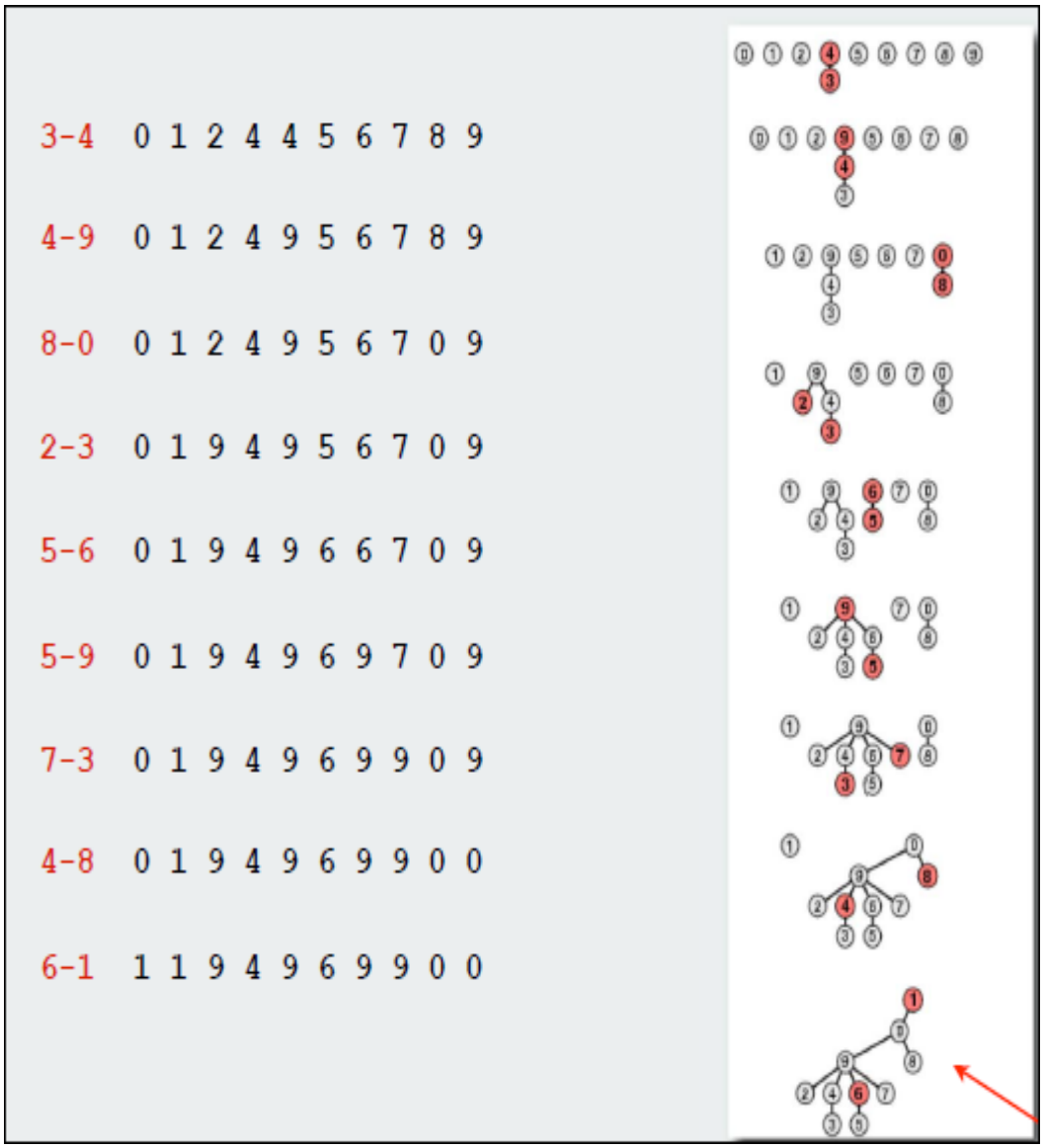
- **Union**. Set the id of q's root to the id of p's root. **Note** for each union we need to do 2 finds.



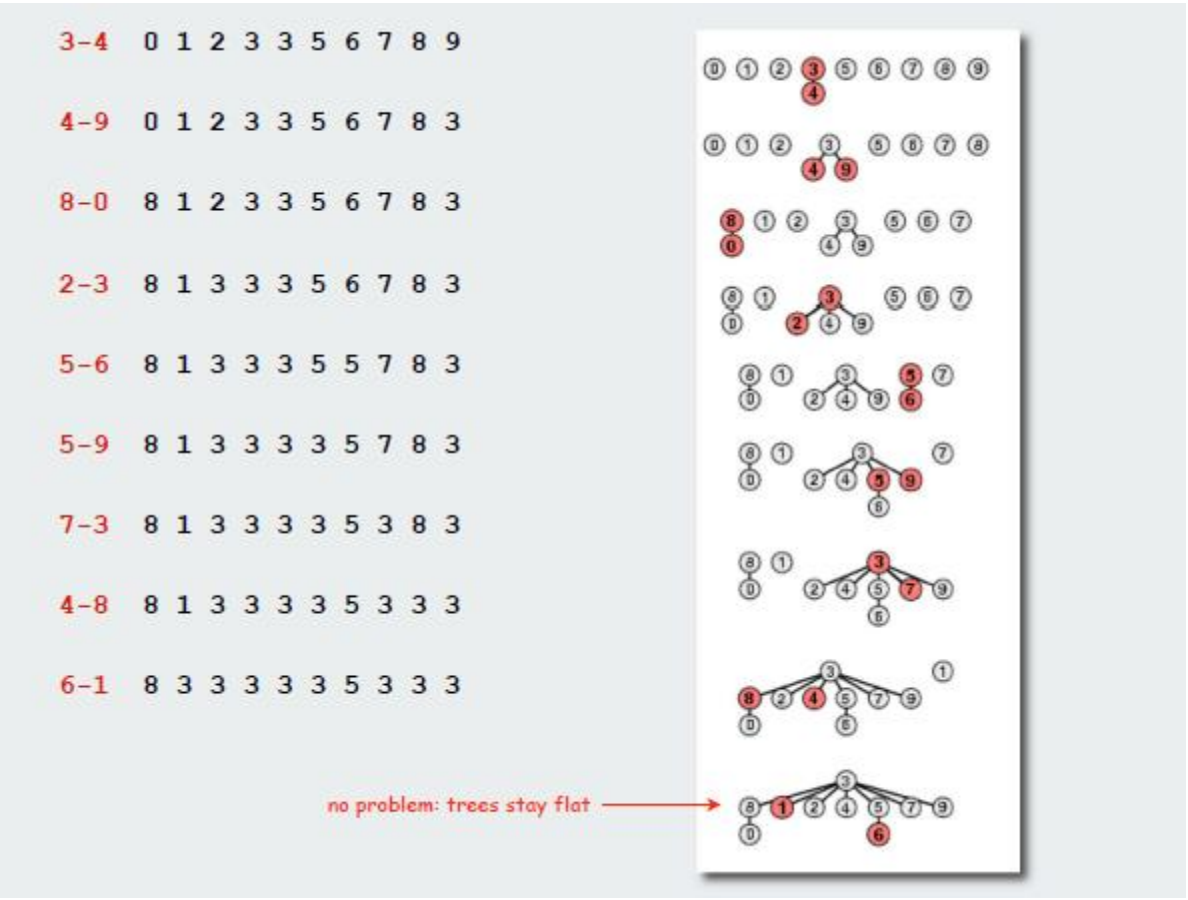
Examples

Note:

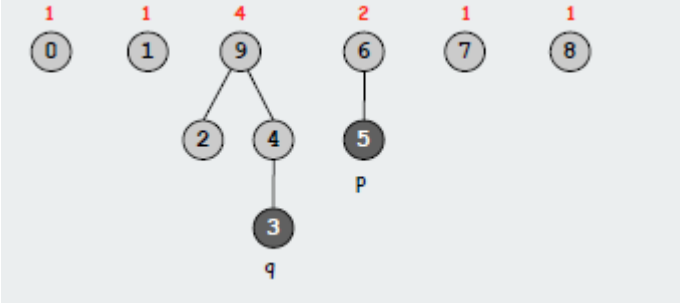
1. union(3,4) needs 3 operations
2. union (4,9) needs 3 operations
3. union (8,0) needs 3 operations
4. union (2,3) ⇒
5. …
6. union (6,1) ⇒ id[6] =9, id[9] = 0, id[1]=1, so do id[0] = 1 ⇒ 4 5 operations.
7. **Problem:** trees can get arbitrarily high [O(N)] and find operations become very expensive. M arbitrary operations may tend to O(MN). Time for find(p) is proportional to the depth of p in its tree. Time for union(p, q) is proportional to depth(p) + depth(q) ??
8. Also, we have arbitrarily chosen who becomes the root after union is executed – does that choice ring any bell?



Union using rank (height)



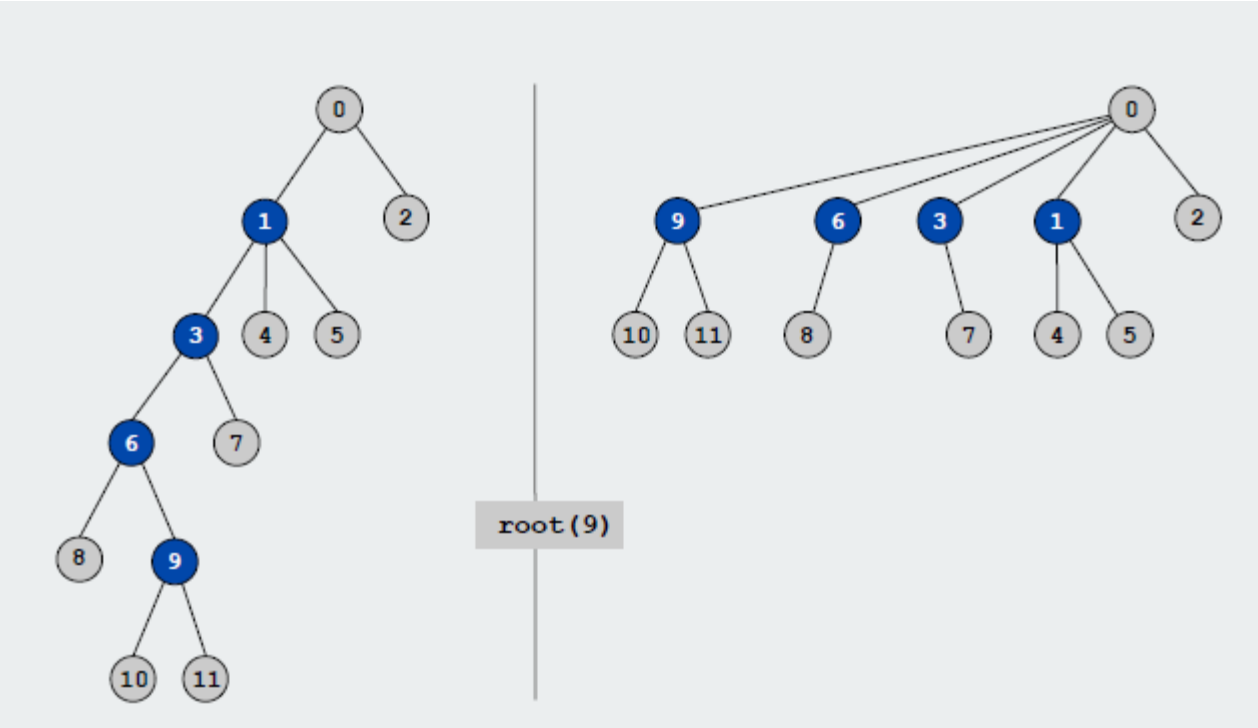
- Problem with the arbitrary root attachment strategy in the simple approach is that: the tree, in the worst-case, could just grow along one long (O(n)) path.
- Idea: Prevent formation of such long chains ⇒ Enforce Union() to happen in a "balanced" way. We'll use rank or height of the tree as a metric. union (p,q) will attach the tree of q to p's root if the tree of p is higher than that of q.
- Union of 5 and 3 will attach 6 to 9.



Both find and union has worst case time of $O(\log_2 N)$

Further Improvement by Path Compression

- Path compression. Just after computing the root of i, set the id of each examined node to root(i).

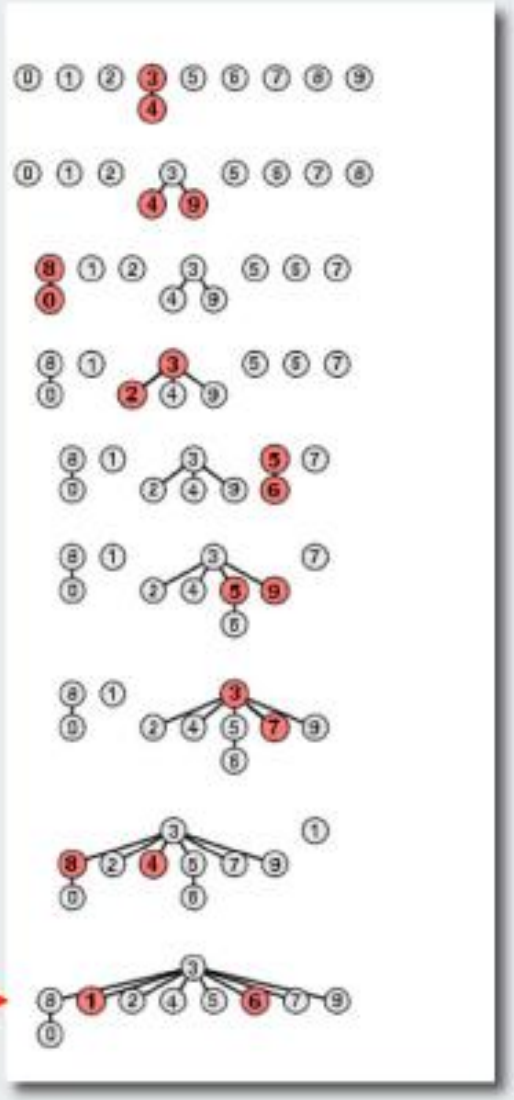


- Coding is simple: int root(i){while (i != id[i]){id[i] = id[id[i]]; i=id[i];} return i;}
- Advantage: Any future calls to find on x or its ancestors will return in constant time!

Example

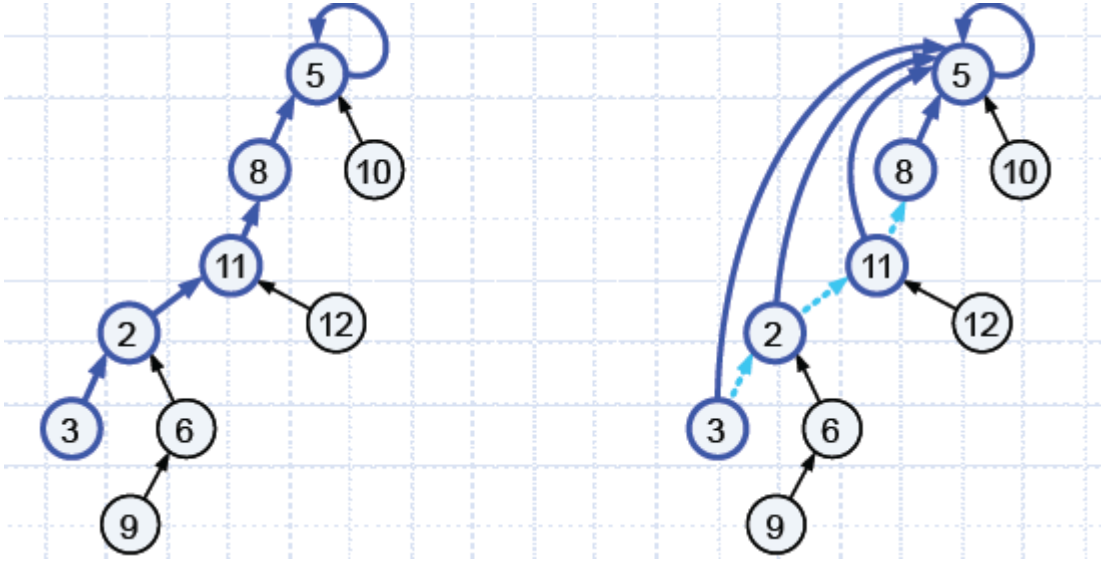
3-4	0 1 2 3 3 5 6 7 8 9
4-9	0 1 2 3 3 5 6 7 8 3
8-0	8 1 2 3 3 5 6 7 8 3
2-3	8 1 3 3 3 5 6 7 8 3
5-6	8 1 3 3 3 5 5 7 8 3
5-9	8 1 3 3 3 3 5 7 8 3
7-3	8 1 3 3 3 3 5 3 8 3
4-8	8 1 3 3 3 3 5 3 3 3
6-1	8 3 3 3 3 3 3 3 3 3

no problem: trees stay VERY flat



Path Compression

- Path compression: After performing a find, compress all the pointers on the path just traversed so that they all point to the root
- Implies a fast "almost linear" time for n union-find operations.



Performance

- Theorem. Starting from an empty data structure, any arbitrary sequence of M union and find operations on N objects takes O(N + M log* N) time.

number of times needed to take the log of a number until reaching 1

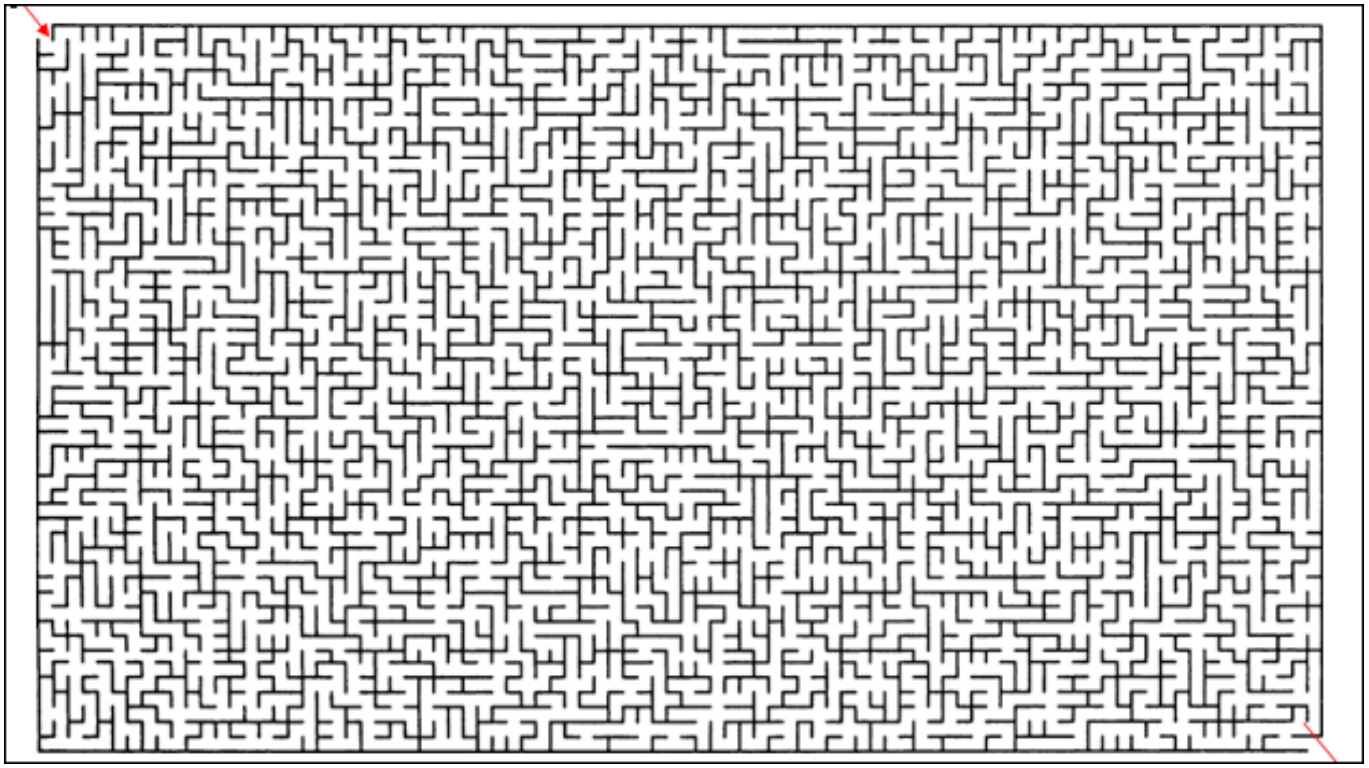
- Proof is extremely difficult, but the algorithm is still extremely simple.
- Cost within constant factor of reading in the data. In theory, the performance is not quite linear, but linear in this universe.

Strictly speaking, the performance is given by *the inverse Ackerman function* of N, which is even more slowly increasing in N than log*N. That is way outside of our scope in this class.

N	log*N
1	0
2	1
4	2
16	3
$655536 = 2^{16}$	4
2^{655536}	5

A Naïve Algorithm for Equivalence Class Computation

- Initially, put each element a ϵ S, in its own equivalence class, i.e., for example, $EqClass_a = \{a\}$ for each pair of elements (a,b); // |S| = n ⇒ there are n(n – 1)/2 pairs if (a R b)
- {$EqClass_a$ = find(a); $Eqclass_b$ = find(b); $EqClass_{ab}$ = union($EqClass_a$, $Eqclass_b$);}
- Run time is $O(n^2 \log^* n)$ [Better solutions using other data structures/techniques could exist depending on the application]
- Let us see how this solves a maze problem; we will use a smaller rectangle to illustrate.



Is there a path from entry to exit?

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

no cells are connected

//**Note**: understand if 2 cells colored same they are reachable

Strategy: //say, entry and exit cells are 0 and 24

As you find cells that are connected,
collapse them into equivalent class

If no more collapses are possible,
examine if entry and exit cells are in same set;
if yes, we have a solution
else no solution.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Given

⇒

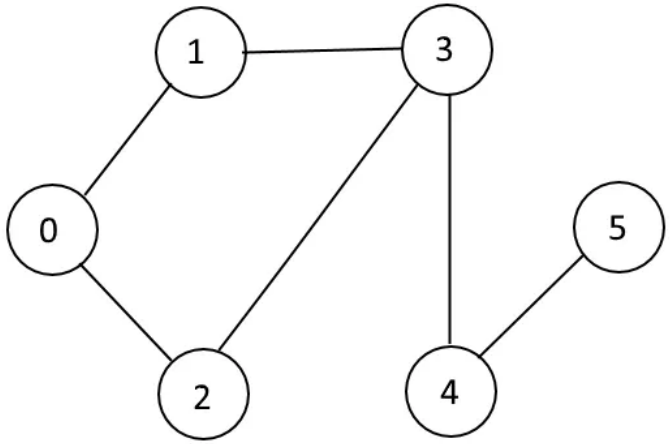
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Solution exists

Now Solve the problem in a, say, 100 × 200 maze! Write the code?

Union Find for Cycle Detection in a Symmetric Graph

- Problem: Given an undirected graph G = (V, E), where V = {1, 2, …, n} is the set of n nodes and E = {e_1, e_2, …, e_m} is the set of m edges [the edges are specified as node pairs], the job is to check if the graph contains any cycle [Recall what is a cycle in an undirected graph].
- The graph contains a cycle {(0,1), (1,3), (3,2), (2,0)}.
- Remember we have already discussed how to detect a cycle using DFS search in an undirected graph [have we??].



Graph contains cycle : 0-1-3-2-0

Can you find the cycle using DFS algorithm given the input?

How to find a cycle using Union – Find Structure?

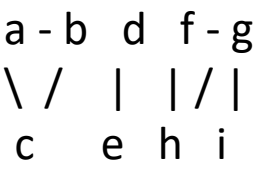
- The makeset operation makes a new set by creating a new element with a parent pointer to itself.
- Then process each edge of the graph and perform Find and Union operations to make subsets using both vertices of the edge.
- If find operation on both the vertices returns the same parent (means both vertices belongs to the same subset) then cycle is detected.
- Example: Consider a graph, V = {0, 1, 2, 3, 4}, E = {(0,1), (1,3), (3,4), (3,2), (0,2)}.
- Writing the code is straightforward.

Connected Components of an undirected graph.

Connected components: {a, b, c}, {d, e}, {f, g, h, i}

Goal: Given vertices v_1 and v_2 determine whether they belong to the same component

Question: How do you use disjoint sets to solve this problem?




```
Connected_Components(G):
  for each vertex v \in V[G]
      do MAKE_SET(v)
  for each edge (u,v) \in E[G]
      do if FIND_SET(u) \neq FIND_SET(v)
         then UNION(u,v)

Same_Component(u,v):
  if FIND_SET(u) = FIND_SET(v)
  then return true
  else return false
```

Ackermann Function☆

The version of the Ackermann function we use is based on an indexed function, A_i, which is defined as follows, for integers $x \geq 0$ and $i > 0$:

$$\begin{aligned} A_0(x) &= x + 1 \\ A_{i+1}(x) &= A_i^{(x)}(x), \end{aligned}$$

where $f^{(k)}$ denotes the k-fold composition of the function f with itself. That is,

$$\begin{aligned} f^{(0)}(x) &= x \\ f^{(k)}(x) &= f(f^{(k-1)}(x)). \end{aligned}$$

So, in other words, $A_{i+1}(x)$ involves making x applications of the A_i function on itself, starting with x. This indexed function actually defines a progression of functions, with each function growing much faster than the previous one:

- $A_0(x) = x + 1$, which is the increment-by-one function
- $A_1(x) = 2x$, which is the multiply-by-two function
- $A_2(x) = x2^x \geq 2^x$, which is the power-of-two function
- $A_3(x) \geq 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ (with x number of 2's), which is the tower-of-twos function
- $A_4(x)$ is greater than or equal to the tower-of-tower-of-twos function
- and so on.

Ackermann Function☆

We then define the **Ackermann function** as

$$A(x) = A_x(2),$$

which is an incredibly fast-growing function.

- To get some perspective, note that A(3) = 2048 and A(4) is greater than or equal to a tower of 2048 twos, which is much larger than the number of subatomic particles in the universe.

Likewise, its inverse, which is pronounced “alpha of n”,

$$\alpha(n) = \min\{x: A(x) \geq n\},$$

is an incredibly slow-growing function. Even though α(n) is indeed growing as n goes to infinity, for all practical purposes, $\alpha(n) \leq 4$.

Fast Amortized Time Analysis

- For each node v in the union tree that is a root
  - define n(v) to be the size of the subtree rooted at v (including v)
  - identified a set with the root of its associated tree.
- We update the size field of v each time a set is union-ed into v. Thus, if v is not a root, then n(v) is the largest the subtree rooted at v can be, which occurs just before we union v into some other node whose size is at least as large as v's.
- For any node v, then, define the rank of v, which we denote as r(v), as $r(v) = \lceil \log n(v) \rceil + 2$
- Thus, $n(v) \geq 2^{r(v)-2}$.
- Also, since there are at most n nodes in the tree of v, $r(v) \leq [\log n]+2$, for each node v.

Fast Amortized Time Analysis

- For each node v with parent w: r (v) < r (w)

Proof: We make v point to w only if the size of w before the union is at least as large as the size of v. Let $n(w)$ denote the size of w before the union and let $n'(w)$ denote the size of w after the union. Thus, after the union we get

$$\begin{aligned} r(v) &= \lfloor \log n(v) \rfloor + 2 \\ &< \lfloor \log n(v) + 1 \rfloor + 2 \\ &= \lfloor \log 2n(v) \rfloor + 2 \\ &\leq \lfloor \log(n(v) + n(w)) \rfloor + 2 \\ &= \lfloor \log n'(w) \rfloor + 2 \\ &\leq r(w). \end{aligned}$$

■

- Thus, ranks are strictly increasing as we follow parent pointers.

Fast Amortized Time Analysis

- Claim: There are at most $n/ 2^{s-2}$ nodes of rank s.
- Proof:
  - Since r(v) < r(w), for any node v with parent w, ranks are monotonically increasing as we follow parent pointers up any tree.
  - Thus, if r(v) = r(w) for two nodes v and w, then the nodes counted in n(v) must be separate and distinct from the nodes counted in n(w).
  - If a node v is of rank s, then $n(v) \geq 2^{s-2}$.
  - Therefore, since there are at most n nodes total, there can be at most $n/2^{s-2}$ that are of rank s.

Fast Amortized Time Analysis

For the sake of our amortized analysis, let us define a ***labeling function***, $L(v)$, for each node v, which changes over the course of the execution of the operations in σ. In particular, at each step t in the sequence σ, define $L(v)$ as follows:

$$L(v) = \text{the largest } i \text{ for which } r(p(v)) \geq A_i(r(v)).$$

Note that if v has a parent, then $L(v)$ is well-defined and is at least 0, since

$$r(p(v)) \geq r(v) + 1 = A_0(r(v)),$$

because ranks are strictly increasing as we go up the tree U. Also, for $n \geq 5$, the maximum value for $L(v)$ is $\alpha(n) - 1$, since, if $L(v) = i$, then

$$\begin{aligned} n &> \lfloor \log n \rfloor + 2 \\ &\geq r(p(v)) \\ &\geq A_i(r(v)) \\ &\geq A_i(2). \end{aligned}$$

Or, put another way,

$$L(v) < \alpha(n),$$

for all v and t.

Fast Amortized Time Analysis

- Let v be a node along a path, P, in the union tree. Charge 1 cyber-dollar for following the parent pointer for v during a find:
  - If v has an ancestor w in P such that L(v) = L(w), at this point in time, then we charge 1 cyber-dollar to v itself.
  - If v has no such ancestor, then we charge 1 cyber-dollar to this find.
- Since there are most $\alpha(n)$ rank groups, this rule guarantees that any find operation is charged at most $\alpha(n)$ cyber-dollars.

Fast Amortized Time Analysis

- After we charge a node v then v will get a new parent, which is a node higher up in v ’s tree.
- The rank of v ’s new parent will be greater than the rank of v ’s old parent w.
- Any node v can be charged at most r(v) cyber-dollars before v goes to a higher label group.
- Since L(v) can increase at most $\alpha(n)-1$ times, this means that each vertex is charged at most $r(n)\alpha(n)$ cyber-dollars.

Fast Amortized Time Analysis

- Combining this fact with the bound on the number of nodes of each rank, this means there are at most

$$s\,\alpha(n)\frac{n}{2^{s-2}} = n\,\alpha(n)\frac{s}{2^{s-2}}$$

- Summing over all possible ranks, the total number of cyber-dollars charged to all nodes is at most

$$\sum_{s=0}^{\log n+2} n\alpha(n)\frac{s}{2^{s-2}} \le \sum_{s=0}^{\infty} n\alpha(n)\frac{s}{2^{s-2}} = n\alpha(n) = n\alpha(n)\sum_{s=0}^{\infty}\frac{s}{2^{s-2}} \le 8n\alpha(n)$$

- So, the total time for m union-find operations, starting with n singleton sets is O((n+m)α(n)).

Longest Zig-Zag Subsequence

- http://www.geeksforgeeks.org/longest-zig-zag-subsequence/
- Given an array of integers, compute the length of the longest subarray(s) where all elements are alternating. http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/

Test

- Write Something; Pradip Srimani Write Something; Pradip Srimani Monai, Rono

Longest Alternating Subsequence

- http://www.geeksforgeeks.org/longest-zig-zag-subsequence/
- Given an array of integers, compute the length of the longest subarray(s) where all elements are alternating. http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/

References for Union-Find

- https://algorithms.tutorialhorizon.com/disjoint-set-data-structure-union-find-algorithm/
- https://www.techiedelight.com/union-find-algorithm-cycle-detection-graph/
- http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut07.txt
- http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut07.txt connected components of an undirected graph.