Union-Find

A Data Structure for Disjoint Set Operations; Applications

Union – Find Structure

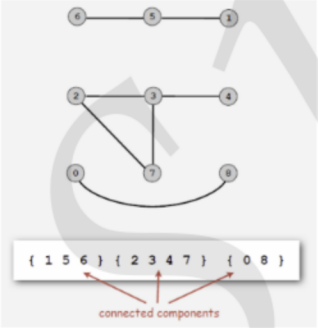
- A partition or union-find structure is a data structure supporting a collection of disjoint sets. We define the methods for this structure assuming we have a constant time way to access a node associated with an item, e. For instance, items could themselves be nodes or we could maintain some kind of 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".
  - 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 between nodes u and v. ← 1 union operation
  - IS-CONNECTED(u, v). Is there a path between u and v? ← 2 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.
  - Could 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.



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.

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

0 1 { 2 3 9 } { 5 6 } 7 { 4 8 } — subsets of connected grid points

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

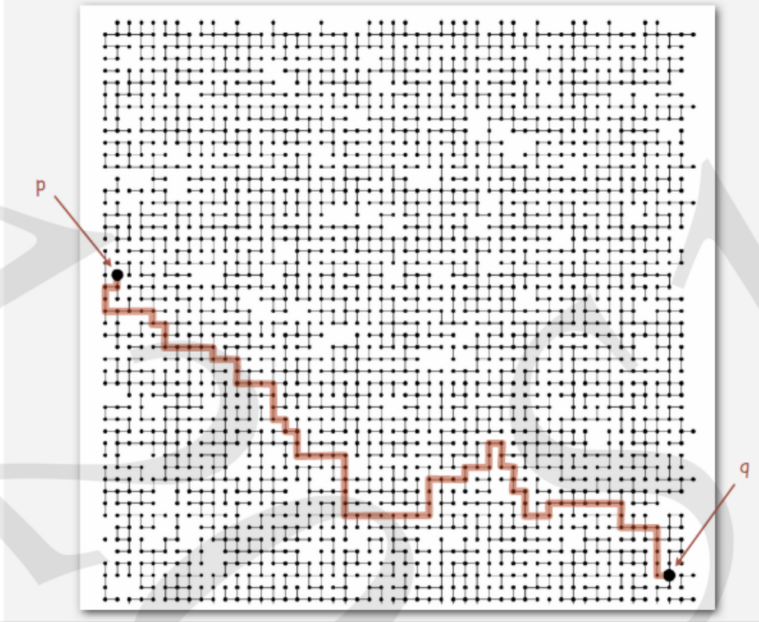
Think of a maze, or a connected component in a graph

0 1 { 2 3 9 } { 5-6 } 7 { 4-8 } — are two grid points connected?

- 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

Network connectivity: larger example



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.

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 $\Theta(\log_2 n)$ using balanced binary trees. Not really good
  - A sequence of M (Arbitrary unions and finds) operations takes $\Theta(M \log_2 N)$.
- **How to improve?**
  - Keep all equivalence sets in separate trees: 1 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).

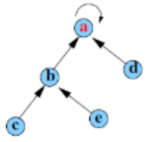
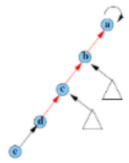


Figure: $S_i = \{a, b, c, d, e\}$.

FIND and UNION

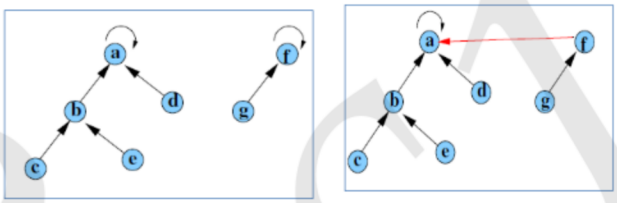
- **Find (x) :**
- To which set does a given element x belong) Find(x).
- Returns the root (representative) of the set that contain x.



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)

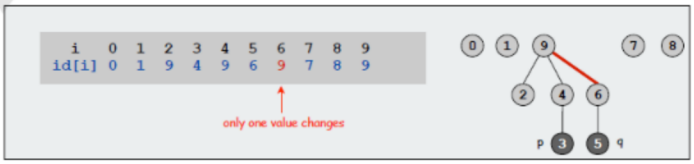
Quick Union [Lazy Approach]

- **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 is 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.

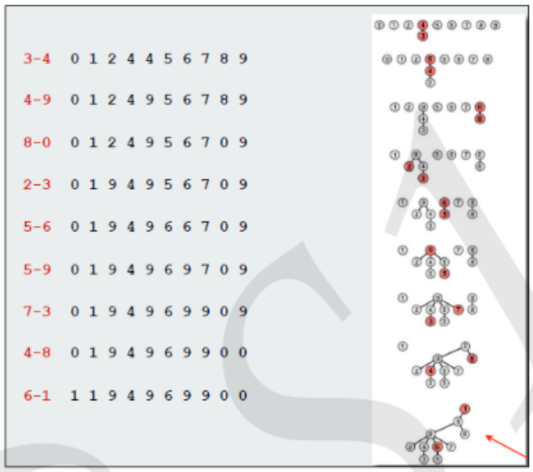


Union (3, 5)

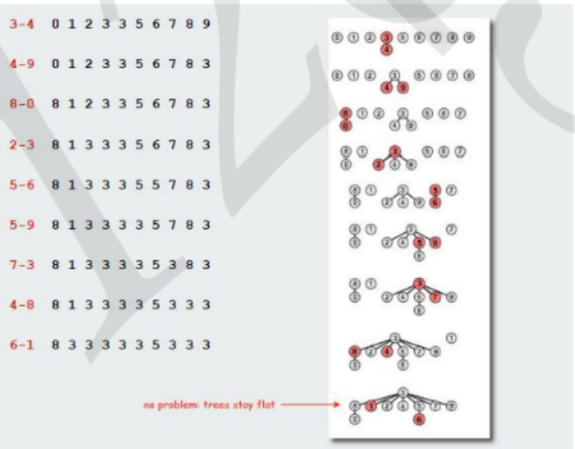
Examples

Note:

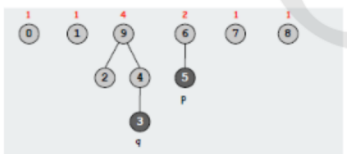
1. union(3,4) needs 3 operations
2. union (4,9) needs 3 operations
3. union (8,0) needs 3 operations
4. union (2,3)
5. …
6. union (6,1) id[6] =9, id[9] = 0, id[1]=1, so do id[0] = 1 4 operations.
7. **Problem:** trees can get arbitrarily large [O(N)] and find operations become very expensive. M arbitrary 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 make any difference in performance?



Union using rank (height)



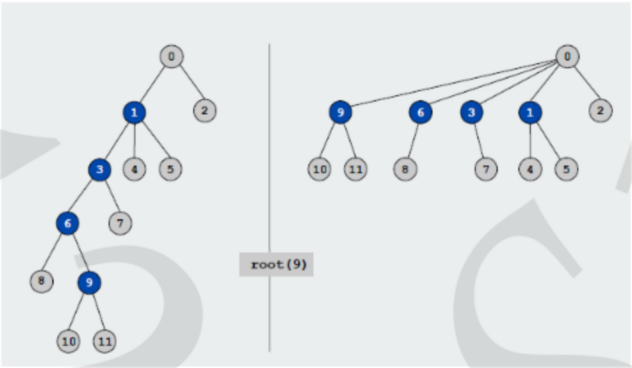
- Problem with the arbitrary root attachment strategy in the simple approach is that: the tree, in the worst-case, could just grow along one long (O(n)) path.
- Idea: Prevent formation of such long chains Enforce Union() to happen in a "balanced" way. We'll use rank or height of the tree as a metric. 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). [Need to traverse the find path twice].

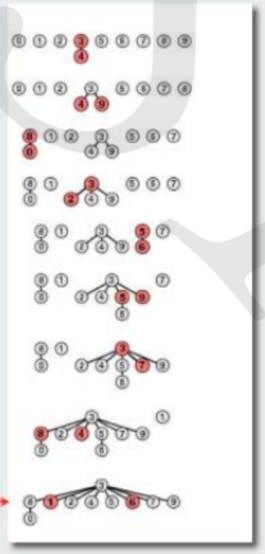


- Coding is simple: int root(i){while (i != id[i]){id[i] = id[id[i]]; i=id[i];} return i;}
- Advantage: Any future calls to find on x or its ancestors will return in constant time!

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



Performance

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

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

- Proof is very 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, a tighter estimate is given by *the inverse Ackerman function* of N, which is even more slowly increasing in N than log*N. While the analysis is complicated, we will show what an Ackerman function is.

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

Ackerman Function*

- **Ackermann function**. A computable function that is not primitive recursive.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

- **Note:** There are other similar definitions (not quite equivalent).
- **Inverse Ackerman function**:

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

" I am not smart enough to understand this easily. "

— Raymond Seidel

*Inverse Ackerman Function**

Definition.

$$\alpha_k(n) = \begin{cases} 1 & \text{if } n = 1 \\ \lceil n/2 \rceil & \text{if } k = 1 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Ex.

- $\alpha_1(n) = \lceil n / 2 \rceil$.
- $\alpha_2(n) = \lceil \lg n \rceil$ = # of times we divide n by two, until we reach 1.
- $\alpha_3(n) = \lg^* n$ = # of times we apply the lg function to n, until we reach 1.
- $\alpha_4(n)$ = # of times we apply the iterated lg function to n, until we reach 1.

$$2 \uparrow 65536 = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \quad (65536 \text{ times})$$

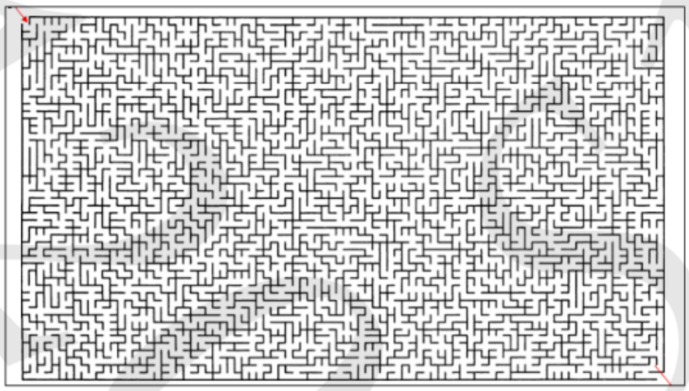
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	2^{16}	...	2^{65536}	...	$2 \uparrow 65536$
$\alpha_1(n)$	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	...	2^{15}	...	2^{65535}	...	*huge*
$\alpha_2(n)$	1	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	...	16	...	65536	...	$2 \uparrow 65535$
$\alpha_3(n)$	1	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	...	4	...	5	...	65536
$\alpha_4(n)$	1	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	...	3	...	3	...	4

Union-find applications

- Equivalence Class, Percolation.
- Games (Go, Hex).
- Network connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm, Cycle detection in graphs.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab's bwlabel() function in image processing.

A Naïve Algorithm for Equivalence 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

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

//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.

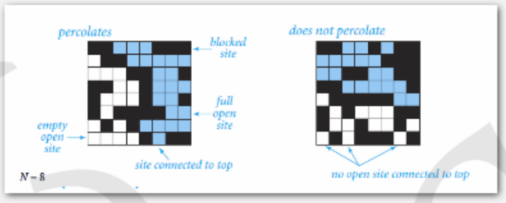
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Solution exists

Now Solve the problem in a, say, 100 × 200 maze! Write the code?
Revise the code to extract the path.

Percolation

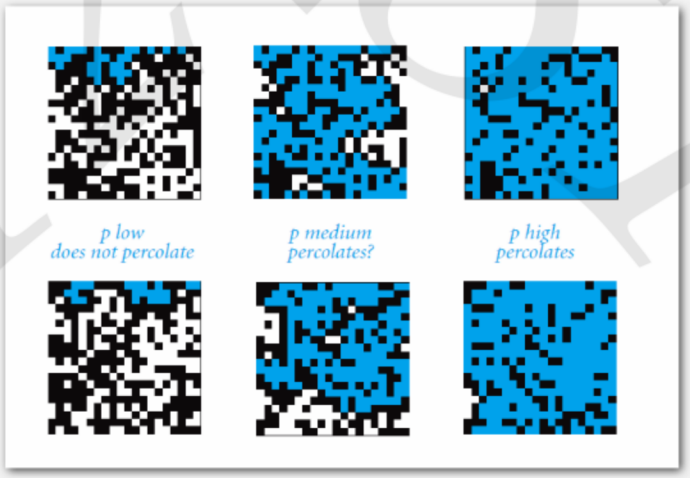
- A model for many physical systems:
  - N-by-N grid of sites.
  - Each site is open with probability p (or blocked with probability 1-p).
  - System percolates if top and bottom are connected by open sites.



model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Likelihood of percolation

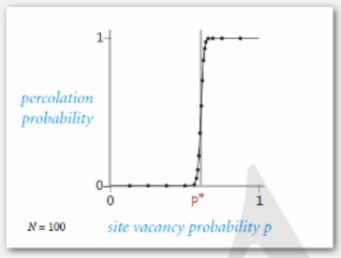
Depends on site vacancy probability p.



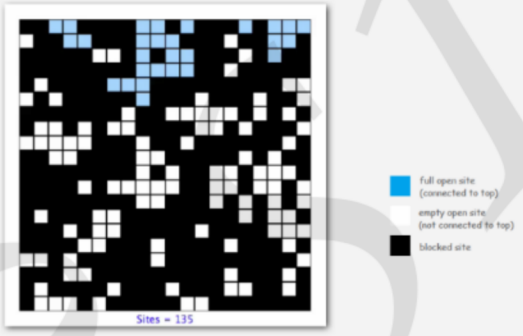
N = 20

Percolation phase transition

- Theory guarantees a sharp threshold p* (when N is large).
  - p > p*: almost certainly percolates.
  - p < p*: almost certainly does not percolate.
- **Question**. What is the value of p* ?

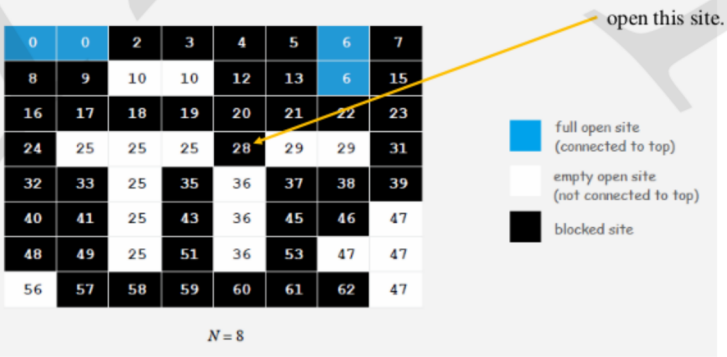


- **Monte Carlo simulation**
  - Initialize N-by-N whole grid to be blocked.
  - Make random sites open until top connected to bottom.
  - Vacancy percentage estimates p*.



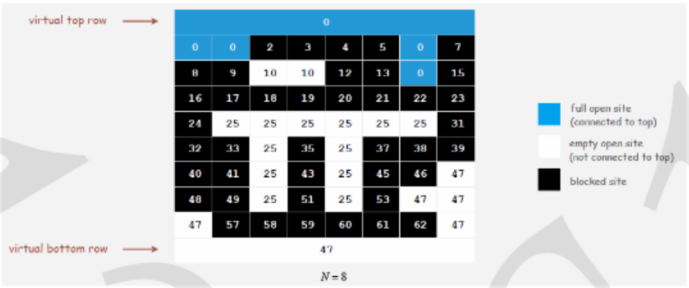
UF solution to find percolation threshold

- How to check whether system percolates?
- Create object for each site.
- Sites are in same set if connected by open sites.
- Percolates if any site in top row is in same set as any site in bottom row [brute force algorithm would need to check N^2 pairs].
- How to declare a new site open? – Take union of new site and all adjacent open sites.

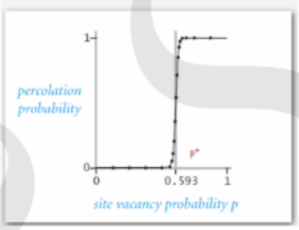


UF solution: a critical optimization

- How to avoid checking all pairs of top and bottom sites? Create a virtual top and bottom objects; system percolates when virtual top and bottom objects are in same set.



Percolation Threshold: What is percolation threshold p* ? About 0.592746 for large square lattices.



Union Find applications in graph theory problems

- Union Find structure can be used to check whether an undirected graph contains a cycle or not. This method assumes that graph doesn't contain any self loops.
- We have already seen application of union find structures in the greedy Kruskal's algorithm to find the minimal spanning tree of a graph.

```
1: S ← ∅
2: for each x ∈ V do
3:     MAKESET(x)
4: for each (x, y) ∈ E do
5:     if FIND(x) ≠ FIND(y) then
6:         UNION(x, y)
7:         S ← S ∪ {(x, y)}
```

- Note that if the edges are processed by increasing weight then this algorithm is Kruskal's algorithm.