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

Use as array index
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 $\cup$ 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 ($i$) $
\Rightarrow \ id(i) = i$ for all $i$.

Step 2: $\text{Union}(0, 1)$ $
\Rightarrow$ the id array is $[0\ 1\ 2\ 3\ 4\ 5]$

Step 3: $\text{Union}(0, 2), \text{Union}(0, 3)$

Step 4: $\text{Union}(4, 5)$

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id(i)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id(i)</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id(i)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id(i)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
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

- **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?

- **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

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

Think of a maze, or a connected component in a graph.
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 \( O(M \times N) \) 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 really 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 \text{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.
  - \text{FIND}(x).
  - \text{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 \( \Rightarrow \) Find(x)
  - Returns the root (representative) of the set that contain x

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

**UNION (c, g)**: (1) Find (c) \( \Rightarrow \) 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]...]]].

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

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[i]</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```

- **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.

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[i]</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```

3's root is 9; 5's root is 6

Only one value changes
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. …
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 ring any bell?
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:

```c
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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>3–4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>4–9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8–0</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2–3</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>5–6</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>5–9</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>7–3</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4–8</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6–1</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

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.

- 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.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log^* N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>$655536 = 2^{16}$</td>
<td>4</td>
</tr>
<tr>
<td>$2^{655536}$</td>
<td>5</td>
</tr>
</tbody>
</table>
A Naïve Algorithm for Equivalence Class Computation

Initially, put each element \( a \in S \), in its own equivalence class, i.e., for example, \( \text{EqClass}_a = \{ a \} \) for each pair of elements \((a,b)\); \hspace{1cm} // \mid S \mid = n \Rightarrow \text{there are} \ n(n - 1)/2 \text{ pairs} \hspace{1cm} \text{if} \hspace{1cm} (a \ R \ b)

\{\text{EqClass}_a = \text{find}(a); \hspace{0.5cm} \text{Eqclass}_b = \text{find}(b); \hspace{0.5cm} \text{EqClass}_{ab} = \text{union}(\text{EqClass}_a, \text{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?**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

//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.

Given

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

Solution exists

Now Solve the problem in a, say, $100 \times 200$ maze! Write the code?
Problem: Given an undirected graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$ is the set of $n$ nodes and $E = \{e_1, e_2, \ldots, e_m\}$ 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??].

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

Graph contains cycle: 0-1-3-2-0
How to find a cycle using Union – Find Structure?

- The `make_set` 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
```