Union-Find Algorithms

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Subtext of today’s lecture (and this course)

Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.
dynamic connectivity
  • quick find
  • quick union
  • improvements
  • applications
Dynamic connectivity

Given a set of objects

• **Union**: connect two objects.
• **Find**: is there a path connecting the two objects?

```
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
  find(0, 2)  no
  find(2, 4)  yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
  find(0, 2)  yes
  find(2, 4)  yes

```
Network connectivity: larger example
Modeling the objects

Dynamic connectivity applications involve manipulating objects of all types.

- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

When programming, convenient to name objects 0 to N-1.

- Use integers as array index.
- Suppress details not relevant to union-find.

Can use symbol table to translate from object names to integers (stay tuned)
**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.

\[
\{ 1 \ 5 \ 6 \} \ { 2 \ 3 \ 4 \ 7 \} \ { 0 \ 8 \}
\]

connected components
Implementing the operations

Find query. Check if two objects are in the same set.

Union command. Replace sets containing two objects with their union.

\[ \{ 1 \ 5 \ 6 \} \{ 2 \ 3 \ 4 \ 7 \} \{ 0 \ 8 \} \]

\[ \{ 1 \ 5 \ 6 \} \{ 0 \ 2 \ 3 \ 4 \ 7 \ 8 \} \]

connected components
Goal. Design efficient data structure for union-find.

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

**Union-find data type (API)**

```java
public class UnionFind {
    public UnionFind(int N) {
        // create union-find data structure with $N$ objects and no connections
    }
    public boolean find(int p, int q) {
        // are $p$ and $q$ in the same set?
    }
    public void unite(int p, int q) {
        // replace sets containing $p$ and $q$ with their union
    }
}
```
• dynamic connectivity
• quick find
• quick union
• improvements
• applications
Quick-find [eager approach]

Data structure.
- Integer array \( \text{id[]} \) of size \( N \).
- Interpretation: \( p \) and \( q \) are connected if they have the same id.

\[
\begin{array}{ccccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{id}[i] & 0 & 1 & 9 & 9 & 9 & 6 & 6 & 7 & 8 & 9 \\
\end{array}
\]

5 and 6 are connected
2, 3, 4, and 9 are connected
Quick-find [eager approach]

Data structure.

- Integer array id[] of size N.
- Interpretation: \( p \) and \( q \) are connected if they have the same id.

\[
\begin{array}{cccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
  id[i] & 0 & 1 & 9 & 9 & 6 & 6 & 7 & 8 & 9 & \\
\end{array}
\]

Find. **Check if \( p \) and \( q \) have the same id.**

- 3 and 6 not connected

- 5 and 6 are connected
- 2, 3, 4, and 9 are connected
Quick-find [eager approach]

Data structure.
• Integer array \( \text{id}[\] \) of size \( N \).
• Interpretation: \( p \) and \( q \) are connected if they have the same id.

\[
\begin{array}{ccccccccccc}
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
id[i] & 0 & 1 & 9 & 9 & 9 & 6 & 6 & 7 & 8 & 9 \\
\end{array}
\]

Find. Check if \( p \) and \( q \) have the same id.

\[
\begin{array}{ccccccccccc}
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
id[i] & 0 & 1 & 6 & 6 & 6 & 6 & 6 & 7 & 8 & 6 \\
\end{array}
\]

Union. To merge sets containing \( p \) and \( q \), change all entries with \( \text{id}[p] \) to \( \text{id}[q] \).

\[
\begin{array}{ccccccccccc}
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
id[i] & 0 & 1 & 6 & 6 & 6 & 6 & 6 & 7 & 8 & 6 \\
\end{array}
\]

union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change
Quick-find example

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

problem: many values can change
Quick-find: Java implementation

```java
public class QuickFind {
    private int[] id;

    public QuickFind(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public boolean find(int p, int q) {
        return id[p] == id[q];
    }

    public void unite(int p, int q) {
        int pid = id[p];
        for (int i = 0; i < id.length; i++)
            if (id[i] == pid) id[i] = id[q];
    }
}
```

- Set id of each object to itself (N operations)
- Check if p and q have same id (1 operation)
- Change all entries with id[p] to id[q] (N operations)
Quick-find is too slow

Quick-find defect.
- Union too expensive (N operations).
- Trees are flat, but too expensive to keep them flat.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

Ex. May take $N^2$ operations to process $N$ union commands on $N$ objects.
Quadratic algorithms do not scale

Rough standard (for now).
• $10^9$ operations per second.
• $10^9$ words of main memory.
• Touch all words in approximately 1 second.

Ex. Huge problem for quick-find.
• $10^9$ union commands on $10^9$ objects.
• Quick-find takes more than $10^{18}$ operations.
• 30+ years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.
• New computer may be 10x as fast.
• But, has 10x as much memory so problem may be 10x bigger.
• With quadratic algorithm, takes 10x as long!
› dynamic connectivity
› quick find
› quick union
› improvements
› applications
Quick-union [lazy approach]

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

<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>

3’s root is 9; 5’s root is 6

Keep going until it doesn’t change
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.

3's root is 9; 5's root is 6
3 and 5 are not connected
**Quick-union** [lazy approach]

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

<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>

**Find.** Check if \( p \) and \( q \) have the same root.

**Union.** To merge subsets containing \( p \) and \( q \), set the id of \( q \)'s root to the id of \( p \)'s 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>9</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

3's root is 9; 5's root is 6
3 and 5 are not connected

only one value changes
keep going until it doesn't change
Quick-union example

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

problem: trees can get tall
public class QuickUnion
{
    private int[] id;

    public QuickUnion(int N)
    {
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
    }

    private int root(int i)
    {
        while (i != id[i]) i = id[i];
        return i;
    }

    public boolean find(int p, int q)
    {
        return root(p) == root(q);
    }

    public void unite(int p, int q)
    {
        int i = root(p), j = root(q);
        id[i] = j;
    }
}
Quick-union is also too slow

Quick-find defect.
• Union too expensive (N operations).
• Trees are flat, but too expensive to keep them flat.

Quick-union defect.
• Trees can get tall.
• Find too expensive (could be N operations).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Union</th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N *</td>
<td>N</td>
</tr>
</tbody>
</table>

* includes cost of finding root
- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Improvement 1: weighting

Weighted quick-union.
- Modify quick-union to avoid tall trees.
- Keep track of size of each subset.
- Balance by linking small tree below large one.

Ex. Union of 3 and 5.
- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.
Weighted quick-union 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 7 8 3
5–9  8 1 3 3 3 3 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 5 3 3 3

no problem: trees stay flat
Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array $sz[i]$ to count number of objects in the tree rooted at $i$.

Find. Identical to quick-union.

```
return root(p) == root(q);
```

Union. Modify quick-union to:

• Merge smaller tree into larger tree.
• Update the $sz[]$ array.

```
int i = root(p);
int j = root(q);
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```
Weighted quick-union analysis

Analysis.
• Find: takes time proportional to depth of p and q.
• Union: takes constant time, given roots.
• Fact: depth is at most $\lg N$. [needs proof]

Q. How does depth of x increase by 1?
A. Tree $T_1$ containing x is merged into another tree $T_2$.
   • The size of the tree containing x at least doubles since $|T_2| \geq |T_1|$.
   • Size of tree containing x can double at most $\lg N$ times.
Weighted quick-union analysis

Analysis.

- **Find**: takes time proportional to depth of p and q.
- **Union**: takes constant time, given roots.
- **Fact**: depth is at most \( \lg N \). \([\text{needs proof}]\)

<table>
<thead>
<tr>
<th>algorithm</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>( N )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>quick-union</td>
<td>( N \ast )</td>
<td>( N )</td>
</tr>
<tr>
<td>weighted QU</td>
<td>( \lg N \ast )</td>
<td>( \lg N )</td>
</tr>
</tbody>
</table>

* includes cost of finding root

**Q.** Stop at guaranteed acceptable performance?

**A.** No, easy to improve further.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of $p$, set the id of each examined node to $\text{root}(p)$. 
Path compression: Java implementation

**Standard implementation:** add second loop to `root()` to set the id of each examined node to the root.

**Simpler one-pass variant:** halve the path length by making every other node in path point to its grandparent.

```java
public int root(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```

In practice. No reason not to! Keeps tree almost completely flat.
Weighted quick-union with path compression 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 7 8 3
5–9  8 1 3 3 3 5 7 8 3
7–3  8 1 3 3 3 3 3 3 3
4–8  8 1 3 3 3 3 3 3
6–1  8 3 3 3 3 3 3 3

no problem: trees stay VERY flat
Theorem. [Tarjan 1975] Starting from an empty data structure, any sequence of $M$ union and find operations on $N$ objects takes $O(N + M \lg^* N)$ time.

- Proof is very difficult.
- But the algorithm is still simple!

Linear algorithm?
- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

Amazing fact. No linear-time linking strategy exists.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>65536</td>
<td>4</td>
</tr>
<tr>
<td>$2^{65536}$</td>
<td>5</td>
</tr>
</tbody>
</table>

because $\lg^* N$ is a constant in this universe

Actually $O(N + M \alpha(M, N))$

see COS 423
Summary

**Bottom line.** WQUPC makes it possible to solve problems that could not otherwise be addressed.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case time</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>$M N$</td>
</tr>
<tr>
<td>quick-union</td>
<td>$M N$</td>
</tr>
<tr>
<td>weighted QU</td>
<td>$N + M \log N$</td>
</tr>
<tr>
<td>QU + path compression</td>
<td>$N + M \log N$</td>
</tr>
<tr>
<td>weighted QU + path compression</td>
<td>$N + M \lg^* N$</td>
</tr>
</tbody>
</table>

*M union-find operations on a set of N objects*

**Ex.** [10⁹ unions and finds with 10⁹ objects]
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.
dynamic connectivity
quick find
quick union
improvements
applications
Union-find applications

- 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.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab's `bwlabel()` function in image processing.
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.
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.

<table>
<thead>
<tr>
<th>model</th>
<th>system</th>
<th>vacant site</th>
<th>occupied site</th>
<th>percolates</th>
</tr>
</thead>
<tbody>
<tr>
<td>electricity</td>
<td>material</td>
<td>conductor</td>
<td>insulated</td>
<td>conducts</td>
</tr>
<tr>
<td>fluid flow</td>
<td>material</td>
<td>empty</td>
<td>blocked</td>
<td>porous</td>
</tr>
<tr>
<td>social interaction</td>
<td>population</td>
<td>person</td>
<td>empty</td>
<td>communicates</td>
</tr>
</tbody>
</table>
Likelihood of percolation

Depends on site vacancy probability $p$.

$p_{\text{low}}$ does not percolate
$p_{\text{medium}}$ percolates?
$p_{\text{high}}$ percolates

$N = 20$
Theory guarantees a sharp threshold $p^*$ (when $N$ is large).

- $p > p^*$: almost certainly percolates.
- $p < p^*$: almost certainly does not percolate.

Q. 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*.
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 alg would need to check $N^2$ pairs

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>28</td>
<td>29</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>25</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>25</td>
<td>43</td>
<td>36</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>25</td>
<td>51</td>
<td>36</td>
<td>53</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>47</td>
</tr>
</tbody>
</table>

$N = 8$
Q. How to declare a new site open?

UF solution to find percolation threshold

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>28</td>
<td>29</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>25</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>25</td>
<td>43</td>
<td>36</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>25</td>
<td>51</td>
<td>36</td>
<td>53</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>47</td>
</tr>
</tbody>
</table>

open this site

full open site (connected to top)
empty open site (not connected to top)
blocked site

N = 8
Q. How to declare a new site open?
A. Take union of new site and all adjacent open sites.

UF solution to find percolation threshold

N = 8
**Q.** How to avoid checking all pairs of top and bottom sites?

**A.** Create a virtual top and bottom objects; system percolates when virtual top and bottom objects are in same set.

---

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>25</td>
<td>35</td>
<td>25</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>25</td>
<td>43</td>
<td>25</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>25</td>
<td>51</td>
<td>25</td>
<td>53</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>47</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>47</td>
</tr>
</tbody>
</table>

**UF solution:** a critical optimization

- **virtual top row**
- **virtual bottom row**

N = 8
**Percolation threshold**

**Q.** What is percolation threshold $p^*$?

**A.** About 0.592746 for large square lattices.

---

**Diagram:**

- **Percolation probability**
- **Site vacancy probability $p$**

**Note:** Percolation constant known only via simulation.
Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.