Announcements

- Homework 3 due in ONE week…Wednesday July 22nd!

- TA Sessions will remain the same time.

- Midterm on Friday
  - Exam review Thursday 5-6 pm in EEB 125
The plan

Last lecture:

• Disjoint sets
• The union-find ADT for disjoint sets

Today’s lecture:

• Finish maze application
• Basic implementation of the union-find ADT with “up trees”
• Optimizations that make the implementation much faster
Example application: maze-building

- Build a random maze by erasing edges
  - Possible to get from anywhere to anywhere
    - Including “start” to “finish”
  - No loops possible without backtracking
    - After a “bad turn” have to “undo”
The algorithm

- \( P = \text{disjoint sets} \) of connected cells
  - initially each cell in its own 1-element set
- \( E = \text{set} \) of edges not yet processed, initially all (internal) edges
- \( M = \text{set} \) of edges kept in maze (initially empty)

while \( P \) has more than one set {
  - Pick a random edge \((x,y)\) to remove from \( E \)
  - \( u = \text{find}(x) \)
  - \( v = \text{find}(y) \)
  - if \( u == v \)
    add \((x,y)\) to \( M \) // same subset, leave edge in maze, do not create cycle
  else
    \text{union}(u,v) \) // connect subsets, remove edge from maze
}
Add remaining members of \( E \) to \( M \), then output \( M \) as the maze
Example

Pick edge (8,14)
Find(8) = 7
Find(14) = 20
Union(7,20)

\[
P = \{1,2,7,8,9,13,19\}
\{3\}
\{4\}
\{5\}
\{6\}
\{10\}
\{11,17\}
\{12\}
\{14,20,26,27\}
\{15,16,21\}
\{18\}
\{25\}
\{28\}
\{31\}
\{22,23,24,29,30,32,33,34,35,36\}
\]
**Example**

\[ P \]

\{1,2,7,8,9,13,19\}
\{3\}
\{4\}
\{5\}
\{6\}
\{10\}
\{11,17\}
\{12\}
\{14,20,26,27\}
\{15,16,21\}
\{18\}
\{25\}
\{28\}
\{31\}
\{22,23,24,29,30,32,33,34,35,36\}

\[ P \]

\{1,2,7,8,9,13,19,14,20,26,27\}
\{3\}
\{4\}
\{5\}
\{6\}
\{10\}
\{11,17\}
\{12\}
\{15,16,21\}
\{18\}
\{25\}
\{28\}
\{31\}
\{22,23,24,29,30,32,33,34,35,36\}

Find(8) = 7
Find(14) = 20

Union(7,20)

\[ \]
**Example: Add edge to M step**

Pick edge \((19,20)\)
Find \((19) = 7\)
Find \((20) = 7\)
Add \((19,20)\) to \(M\)

<table>
<thead>
<tr>
<th>Start</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>End</td>
</tr>
</tbody>
</table>

\(P\)
\{'1,2,\textcolor{red}{7},8,9,13,19,14,20,26,27\}
\{'3\}'
\{'4\}'
\{'5\}'
\{'6\}'
\{'10\}'
\{'11,17\}'
\{'12\}'
\{'15,16,21\}'
\{'18\}'
\{'25\}'
\{'28\}'
\{'31\}'
\{'22,23,24,29,30,32,33,\textcolor{red}{34},35,36\}'
At the end of while loop

- Stop when \( P \) has one set (i.e. all cells connected)
- Suppose green edges are already in \( M \) and black edges were not yet picked
  - Add all black edges to \( M \)

\[
P = \{1,2,3,4,5,6,7,\ldots,36\}
\]
Union-Find ADT

- **create** an initial partition of a set
  - Typically each item in its own subset: \{a\}, \{b\}, \{c\}, …
  - Name each subset by choosing a *representative element*

- **find** takes an element of \(S\) and returns the representative element of the subset it is in

- **union** takes two subsets and (permanently) makes one larger subset
Implementation – our goal

• Start with an initial partition of $n$ subsets
  – Often 1-element sets, e.g., \{1\}, \{2\}, \{3\}, …, \{n\}

• May have $m$ find operations

• May have up to $n-1$ union
  – After $n-1$ union operations, every find returns same 1 set
Up-tree data structure

- Tree with:
  - No limit on branching factor
  - References from *children* to *parent*

- Start with *forest* of 1-node trees
  
- Possible forest after several unions:
  - Will use roots for set names
Find

\texttt{\textbf{find}(x)}:
- \textit{Assume} we have \(O(1)\) access to each node
  - Will use an array where index \(i\) holds node \(i\)
- Start at \(x\) and follow parent pointers to root
- Return the root

\texttt{\textbf{find}(6)} = 7
**Union**

union(x,y):
- Assume x and y are roots
  - Else find the roots of their trees
- Change root of one to have parent be the root of the other
  - Notice no limit on branching factor

union(1,7)
Simple implementation

• If set elements are contiguous numbers (e.g., 1,2,…,n), use array of length $n$ called $\text{up}$
  – Starting at index 1 on slides
  – Put in array index of parent, with 0 (or -1, etc.) for a root

• Example:

\begin{itemize}
  \item Example: \begin{tikzpicture}
    \node (1) at (0,0) {1};
    \node (2) at (1,-1) {2};
    \node (3) at (1,1) {3};
    \node (4) at (2,0) {4};
    \node (5) at (2,2) {5};
    \node (6) at (3,0) {6};
    \node (7) at (2,4) {7};
    \draw (1) -- (2);
    \draw (1) -- (3);
    \draw (2) -- (4);
    \draw (2) -- (5);
    \draw (3) -- (5);
    \draw (5) -- (6);
    \draw (5) -- (7);
  \end{tikzpicture}

  \begin{array}{cccccccc}
    1 & 2 & 3 & 4 & 5 & 6 & 7 \\
    0 & 0 & 0 & 0 & 0 & 0 & 0
  \end{array}

  \begin{array}{cccccccc}
    1 & 2 & 3 & 4 & 5 & 6 & 7 \\
    0 & 1 & 0 & 7 & 7 & 5 & 0
  \end{array}
\end{itemize}
Implement operations

// assumes x in range 1,n
int find(int x) {
    while(up[x] != 0) {
        x = up[x];
    }
    return x;
}

// assumes x,y are roots
void union(int x, int y) {
    up[y] = x;
}

- Worst-case run-time for union? \(O(1)\)
- Worst-case run-time for find? \(O(n)\)
- Worst-case run-time for \(m\) finds and \(n-1\) unions? \(O(m*n)\)
Two key optimizations

1. Improve union so it stays $O(1)$ but makes find $O(\log n)$
   - So $m$ finds and $n-1$ unions is $O(m \log n + n)$
   - Union-by-size: connect smaller tree to larger tree

2. Improve find so it becomes even faster
   - Make $m$ finds and $n-1$ unions almost $O(m + n)$
   - Path-compression: connect directly to root during finds
The bad case to avoid

1  2  3  ...  n

union(2,1)
union(3,2)
:  
union(n,n-1)

find(1) = n steps!!
Union-by-size

Union-by-size:
  - Always point the *smaller* (total # of nodes) tree to the root of the larger tree

union(1,7)
Union-by-size

- Always point the \textit{smaller} (total \# of nodes) tree to the root of the larger tree

\begin{itemize}
  \item union(1,7)
\end{itemize}
Array implementation

Keep the size (number of nodes in a second array)
  – Or have one array of objects with two fields
**Nifty trick**

Actually we do not need a second array…

- Instead of storing 0 for a root, store negation of size
- So up value < 0 means a root
The Bad case? Now a Great case…

\[ \text{union}(2,1) \]

\[ \text{union}(3,2) \]

\[ \vdots \]

\[ \text{union}(n, n-1) \]

\[ \text{find}(1) \text{ constant here} \]
General analysis

• Showing one worst-case example is now good is *not* a proof that the worst-case has improved

• So let’s prove:
  – `union` is still $O(1)$ – this is “obvious”
  – `find` is now $O(\log n)$

• Claim: If we use union-by-size, an up-tree of height $h$ has at least $2^h$ nodes
  – Proof by induction on $h$…
Exponential number of nodes

\[ P(h) = \text{With union-by-size, up-tree of height } h \text{ has at least } 2^h \text{ nodes} \]

Proof by induction on \( h \)…

- **Base case:** \( h = 0 \): The up-tree has 1 node and \( 2^0 = 1 \)
- **Inductive case:** Assume \( P(h) \) and show \( P(h+1) \)
  - A height \( h+1 \) tree \( T \) has at least one height \( h \) child \( T_1 \)
  - \( T_1 \) has at least \( 2^h \) nodes by induction (assumption)
  - And \( T \) has *at least* as many nodes not in \( T_1 \) than in \( T_1 \)
    - Else union-by-size would have
      had \( T \) point to \( T_1 \), not \( T_1 \) point to \( T \) (!!)
    - So total number of nodes is *at least* \( 2^h + 2^h = 2^{h+1} \)
The key idea

Intuition behind the proof: No one child can have more than half the nodes

As usual, if number of nodes is exponential in height, then height is logarithmic in number of nodes

So \text{find} is \(O(\log n)\)
The new worst case

n/2 Unions-by-size

n/4 Unions-by-size

n/8 Unions-by-size
The new worst case (continued)

After $\frac{n}{2} + \frac{n}{4} + \ldots + 1$ Unions-by-size:

Height grows by 1 a total of $\log n$ times
What about union-by-height

We could store the height of each root rather than size

• Still guarantees logarithmic worst-case find
  – Proof left as an exercise if interested

• But does not work well with our next optimization
Two key optimizations

1. Improve `union` so it stays $O(1)$ but makes `find` $O(\log n)$
   - So $m$ finds and $n-1$ unions is $O(m \log n + n)$
   - *Union-by-size*: connect smaller tree to larger tree

2. Improve `find` so it becomes even faster
   - Make $m$ finds and $n-1$ unions *almost* $O(m + n)$
   - *Path-compression*: connect directly to root during finds
**Path compression**

- Simple idea: As part of a `find`, change each encountered node’s parent to point directly to root
  - Faster future `finds` for everything on the path (and their descendants)
Pseudocode

// performs path compression
int find(i) {

    // find root
    int r = i
    while (up[r] > 0) {
        r = up[r]
    }

    // compress path
    if i==r {
        return r;
    }

    int old_parent = up[i]
    while (old_parent != r) {
        up[i] = r
        i = old_parent;
        old_parent = up[i]
    }

    return r;
}
So, how fast is it?

A single worst-case `find` could be $O(\log n)$
- But only if we did a lot of worst-case unions beforehand
- And path compression will make future finds faster

Turns out the amortized worst-case bound is much better than $O(\log n)$
- We won’t prove it – see text if curious
- But we will understand it:
  - How it is almost $O(1)$
  - Because total for $m$ finds and $n-1$ unions is almost $O(m+n)$
A really slow-growing function

\( \log^* x \) is the minimum number of times you need to apply “\( \log \) of \( \log \) of \( \log \) of” to go from \( x \) to a number \( \leq 1 \)

For just about every number we care about, \( \log^* x \) is less than or equal to 5 (!)
If \( x \leq 2^{65536} \) then \( \log^* x \leq 5 \)
- \( \log^* 2 = 1 \)
- \( \log^* 4 = \log^* 2^2 = 2 \)
- \( \log^* 16 = \log^* 2^{(2^2)} = 3 \) (\( \log \log \log 16 = 1 \))
- \( \log^* 65536 = \log^* 2^{(2^2)^2} = 4 \) (\( \log \log \log \log 65536 = 1 \))
- \( \log^* 2^{65536} = \ldots \ldots = 5 \)
Almost linear

- Turns out total time for $m$ finds and $n-1$ unions is $O((m+n) \times (\log^* (m+n)))$
  - Remember, if $m+n < 2^{65536}$ then $\log^* (m+n) < 5$
    so effectively we have $O(m+n)$
- Because $\log^*$ grows soooo slowly
  - For all practical purposes, amortized bound is constant, i.e.,
    cost of find is constant, total cost for $m$ finds is linear
  - We say “near linear” or “effectively linear”
- Need union-by-size and path-compression for this bound
  - Path-compression changes height but not weight, so they
    interact well
- As always, asymptotic analysis is separate from “coding it up”