Announcements

• Reading for this lecture: Chapter 8.
Dijkstra’s Algorithm

S = {}; \ d[s] = 0; \ d[v] = infinity for v != s

While S != V

Choose v in V-S with minimum d[v]
Add v to S
For each w in the neighborhood of v

\[ d[w] = \min(d[w], d[v] + c(v, w)) \]

Assume all edges have non-negative cost
Simulate Dijkstra’s algorithm (starting from s) on the graph

<table>
<thead>
<tr>
<th>Round</th>
<th>Vertex Added</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>a</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Edsger Wybe Dijkstra was one of the most influential members of computing science's founding generation. Among the domains in which his scientific contributions are fundamental are

- algorithm design
- programming languages
- program design
- operating systems
- distributed processing
- formal specification and verification
- design of mathematical arguments
Why do we worry about negative cost edges??
Graph Algorithms / Data Structures

• Dijkstra’s Algorithm for Shortest Paths
  – Heaps, $O(m \log n)$ runtime

• Kruskal’s Algorithm for Minimum Spanning Tree
  – Union-Find data structure
Making Connections

You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

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

Q: Are nodes 2 and 4 (indirectly) connected?
Q: How about nodes 3 and 8?
Q: Are any of the paired connections redundant due to indirect connections?
Q: How many sub-networks do you have?
Making Connections

Answering these questions is much easier if we create disjoint sets of nodes that are connected:

Start:  \{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\} \{8\} \{9\}

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

**Q:** Are nodes 2 and 4 (indirectly) connected?

**Q:** How about nodes 3 and 8?

**Q:** Are any of the paired connections redundant due to indirect connections?

**Q:** How many sub-networks do you have?
Applications of Disjoint Sets

Maintaining disjoint sets in this manner arises in a number of areas, including:

– Networks
– Transistor interconnects
– Compilers
– Image segmentation
– Building mazes (this lecture)
– Graph problems
  • Minimum Spanning Trees (upcoming topic in this class)
Disjoint Set ADT

- Data: set of pairwise **disjoint sets**.
- Required operations
  - **Union** – merge two sets to create their union
  - **Find** – determine which set an item appears in
- A common operation sequence:
  - Connect two elements if not already connected:
    
    `if (Find(x) != Find(y)) then Union(x,y)`
Disjoint Sets and Naming

• Maintain a set of pairwise disjoint sets.
  – \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}

• Each set has a unique name: one of its members (for convenience)
  – \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
Union

• Union(x,y) – take the union of two sets named x and y
  – \{3,\fcolorbox{red}{5},7\} , \{4,2,\fcolorbox{red}{8}\}, \{9\}, \{1,6\}
  – Union(5,1)
    \{3,\fcolorbox{red}{5},7,1,6\}, \{4,2,\fcolorbox{red}{8}\}, \{9\},
Find

• Find(x) – return the name of the set containing $x$.
  – \{3,\textcolor{red}{5},\textcolor{red}{7},1,6\}, \{4,2,\textcolor{red}{8}\}, \{\textcolor{red}{9}\},
  – Find(1) = 5
  – Find(4) = 8
Example

\[
S = \{1,2,7,8,9,13,19\} \\
\{3\} \\
\{4\} \\
\{5\} \\
\{6\} \\
\{10\} \\
\{11,17\} \\
\{12\} \\
\{14,20,26,27\} \\
\{15,16,21\} \\
\ldots \\
\{22,23,24,29,39,32\} \\
\{33,34,35,36\}
\]

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

\[
S = \{1,2,7,8,9,13,19,14,20,26,27\} \\
\{3\} \\
\{4\} \\
\{5\} \\
\{6\} \\
\{10\} \\
\{11,17\} \\
\{12\} \\
\{15,16,21\} \\
\ldots \\
\{22,23,24,29,39,32\} \\
\{33,34,35,36\}
\]
Nifty Application: Building Mazes

Idea: Build a random maze by erasing walls.
Building Mazes

- Pick Start and End
Building Mazes

- Repeatedly pick random walls to delete.
Desired Properties

• None of the boundary is deleted (except at “start” and “end”).

• Every cell is reachable from every other cell.

• There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.
A Cycle
A Good Solution
A Hidden Tree
Number the Cells

We start with disjoint sets $S = \{ \{1\}, \{2\}, \{3\}, \{4\}, \ldots, \{36\} \}$. We have all possible walls between neighbors $W = \{ (1,2), (1,7), (2,8), (2,3), \ldots \}$ 60 walls total.

<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></td>
</tr>
</tbody>
</table>

**Idea**: Union-find operations will be done on cells.
Maze Building with Disjoint Union/Find

Algorithm sketch:
1. Choose wall at random.
   → Boundary walls are not in wall list, so left alone
2. Erase wall if the neighbors are in disjoint sets.
   → Avoids cycles
3. Take union of those sets.
4. Go to 1, iterate until there is only one set.
   → Every cell reachable from every other cell.
Pseudocode

- $S =$ set of sets of connected cells
  - Initialize to $\{\{1\}, \{2\}, \ldots, \{n\}\}$
- $W =$ set of walls
  - Initialize to set of all walls $\{\{1,2\}, \{1,7\}, \ldots\}$
- $\text{Maze} =$ set of walls in maze (initially empty)

While there is more than one set in $S$
  Pick a random non-boundary wall $(x,y)$ and remove from $W$
  $u =$ Find$(x)$;
  $v =$ Find$(y)$;
  if $u \neq v$ then
    Union$(u,v)$
  else
    Add wall $(x,y)$ to Maze
Add remaining members of $W$ to Maze
Example Step

Pick (8,14)

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

End

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

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}

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

Union(7,20)

S
{1,2,7,8,9,13,19,14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}

{22,23,24,29,39,32
33,34,35,36}
Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<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></td>
</tr>
</tbody>
</table>

Start

End

Pick (19, 20)

$S = \{1, 2, 7, 8, 9, 13, 19, 14, 20, 26, 27\}$

$\{3\}$

$\{4\}$

$\{5\}$

$\{6\}$

$\{10\}$

$\{11, 17\}$

$\{12\}$

$\{15, 16, 21\}$

$\{22, 23, 24, 29, 39, 32, 33, 34, 35, 36\}$
### Example at the End

<table>
<thead>
<tr>
<th></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></td>
</tr>
</tbody>
</table>

- **Start**
- **End**

\[ S \{1,2,3,4,5,6,7,...,36\} \]

Remaining walls in \( W \)

Previously added to Maze
Data structure for disjoint sets?

• Represent:  \{3,5,7\} , \{4,2,8\}, \{9\}, \{1,6\}
• Support: find(x), union(x,y)
Union/Find Trade-off

- Known result:
  - Find and Union cannot both be done in worst-case $O(1)$ time with any data structure.
- We will instead aim for good amortized complexity.
- For $m$ operations on $n$ elements:
  - Target complexity: $O(m)$ i.e. $O(1)$ amortized
Tree-based Approach

Each set is a tree

• Root of each tree is the set name.

• Allow large fanout (why?)
Up-Tree for DS Union/Find

**Observation:** we will only traverse these trees upward from any given node to find the root.

**Idea:** reverse the pointers (make them point up from child to parent). The result is an **up-tree**.

Initial state:

```
    1
   / \
  2   3
 / \ / \   
4   5 6   7
```

Intermediate state:

```
    1
   / \
  2   3
    / \
   5   6
      / \
    7   4
```

Roots are the names of each set.
Find Operation

Find(x) follow x to the root and return the root.
Union Operation

Union(i, j) - assuming i and j roots, point i to j.
Simple Implementation

- Array of indices

```
up = [1, 2, 3, 4, 5, 6, 7]
```

$\text{up}[x] = -1$ means $x$ is a root.
void Union(int x, int y) {
    assert(up[x]<0 && up[y]<0);
    up[x] = y;
}

int Find(int x) {
    while(up[x] >= 0) {
        x = up[x];
    }
    return x;
}

runtime for Union: runtime for Find:

Amortized complexity is no better.
A Bad Case

Union(1,2)

Union(2,3)

Union(n-1,n)

Find(1) n steps!!
Two Big Improvements

Can we do better? Yes!

1. Union-by-size
   - Improve \texttt{Union} so that \texttt{Find} only takes worst case time of $\Theta(\log n)$.

2. Path compression
   - Improve \texttt{Find} so that, with Union-by-size, \texttt{Find} takes amortized time of almost $\Theta(1)$. 
Union-by-Size

Union-by-size
– Always point the smaller tree to the root of the larger tree

S-Union(7,1)
Example Again

\[ S - \text{Union}(1,2) \]
\[ S - \text{Union}(2,3) \]
\[ \vdots \]
\[ S - \text{Union}(n-1,n) \]

Find(1) constant time
Analysis of Union-by-Size

• Theorem: With union-by-size an up-tree of height $h$ has size at least $2^h$.

• Proof by induction
  – Base case: $h = 0$. The up-tree has one node, $2^0 = 1$
  – Inductive hypothesis: Assume true for $h-1$
  – Observation: tree gets taller only as a result of a union.
Analysis of Union-by-Size

• What is worst case complexity of Find(x) in an up-tree forest of $n$ nodes?

• (Amortized complexity is no better.)
Worst Case for Union-by-Size

n/2 Unions-by-size

n/4 Unions-by-size
Example of Worst Cast (cont’)

After $n - 1 = n/2 + n/4 + \ldots + 1$ Unions-by-size

If there are $n = 2^k$ nodes then the longest path from leaf to root has length $k$. 
Array Implementation

Can store separate size array:

<table>
<thead>
<tr>
<th>up size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>-1</td>
</tr>
</tbody>
</table>

47
Better, store sizes in the up array:

```
1  2  3  4  5  6  7
up: -2  1  -1  7  7  5  -4
```

Negative up-values correspond to sizes of roots.
Code for Union-by-Size

S-Union(i, j) {
    // Collect sizes
    si = -up[i];
    sj = -up[j];

    // verify i and j are roots
    assert(si >= 0 && sj >= 0)
    // point smaller sized tree to
    // root of larger, update size
    if (si < sj) {
        up[i] = j;
        up[j] = -(si + sj);
    } else {
        up[j] = i;
        up[i] = -(si + sj);
    }
}
Path Compression

- To improve the amortized complexity, we’ll borrow an idea from splay trees:
  - When going up the tree, *improve nodes on the path!*
- On a Find operation point all the nodes on the search path directly to the root. This is called “path compression.”
Self-Adjustment Works

PC-Find(x)
Draw the result of Find(5):
Code for Path Compression Find

PC-Find(i) {
    // find root
    j = i;
    while (up[j] >= 0) {
        j = up[j];
        root = j;
    }

    // compress path
    if (i != root) {
        parent = up[i];
        while (parent != root) {
            up[i] = root;
            i = parent;
            parent = up[parent];
        }
    }
    return(root)
}
Complexity of Union-by-Size + Path Compression

• Worst case time complexity for…
  – …a single Union-by-size is:
  – …a single PC-Find is:

• Time complexity for \( m \geq n \) operations on \( n \) elements has been shown to be \( O(m \log^* n) \).
  [See Weiss for proof.]
  – Amortized complexity is then \( O(\log^* n) \)
  – What is \( \log^* \) ?
**log* $n$**

log* $n$ = number of times you need to apply log to bring value down to at most 1

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* $2^{65536}$ = ............... ≈ log* ($2 \times 10^{19,728}$) = 5

log * $n \leq 5$ for all reasonable $n$. 
In fact, Tarjan showed the time complexity for \( m \geq n \) operations on \( n \) elements is:

\[
\Theta(m \alpha(m, n))
\]

Amortized complexity is then \( \Theta(\alpha(m, n)) \).

What is \( \alpha(m, n) \)?

– Inverse of Ackermann’s function.

– For reasonable values of \( m, n \), grows even slower than \( \log * n \). So, it’s even “more constant.”

Proof is beyond scope of this class. A simple algorithm can lead to incredibly hardcore analysis!