Lecture 20: Union-Find and Shortest Path  
(CLRS 21.1-21.3, 24.0)

April 9, 2002

1 Union-Find  

- Last time we discussed Kruskal’s minimum spanning tree algorithm

<table>
<thead>
<tr>
<th>KRUSKAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = \emptyset$</td>
</tr>
<tr>
<td>FOR each vertex $v \in V$ DO</td>
</tr>
<tr>
<td>MAKE-SET($v$)</td>
</tr>
<tr>
<td>OD</td>
</tr>
<tr>
<td>Sort edges of $E$ in increasing order by weight</td>
</tr>
<tr>
<td>FOR each edge $e = (u, v) \in E$ in order DO</td>
</tr>
<tr>
<td>IF FIND-SET($u$) $\neq$ FIND-SET($v$) THEN</td>
</tr>
<tr>
<td>$T = T \cup {e}$</td>
</tr>
<tr>
<td>UNION-SET($u$, $v$)</td>
</tr>
<tr>
<td>FI</td>
</tr>
<tr>
<td>OD</td>
</tr>
</tbody>
</table>

- Kruskal’s algorithm uses a Union-Find data structure supporting:
  - MAKE-SET($v$): Create set consisting of $v$
  - UNION-SET($u$, $v$): Unite set containing $u$ and set containing $v$
  - FIND-SET($u$): Return unique representative for set containing $u$

- In the algorithm we performed $|V|$ MAKE-SET, $|V| - 1$ UNION-SET, and $2|E|$ FIND-SET operations.

- Simple solution to Union-Find problem (maintain set system under FIND-SET and UNION-SET)
  - Maintain elements in same set as a linked list with each element having a pointer to the first element in the list (unique representative)
Example:

Sets

- **MAKE-SET(v):** Make a list with one element \( \Rightarrow O(1) \) time
- **FIND-SET(u):** Follow pointer and return unique representative \( \Rightarrow O(1) \) time
- **UNION-SET(u, v):** Link first element in list with unique representative **FIND-SET(u)** after last element in list with unique representative **FIND-SET(v)** \( \Rightarrow O(|V|) \) time (as we have to update all unique representative pointers in list containing u)

- With this simple solution the \( |V| - 1 \) **UNION-SET** operations in Kruskal’s algorithm may take \( O(|V|^2) \) time.
- We can improve the performance of **UNION-SET** with a very simple modification: Always link the smaller list after the longer list \( \Rightarrow \) update the pointers of the smaller list
  - One **UNION-SET** operation can still take \( O(|V|) \) time, but the \( |V| - 1 \) **UNION-SET** operations take \( O(|V| \log |V|) \) time altogether (one **UNION-SET** takes \( O(\log |V|) \) time *amortized*):
    * Total time is proportional to number of unique representative pointer changes
    * Consider element u:
      After pointer for u is updated, u belongs to a list of size at least double the size of the list it was in before
      \[ \Downarrow \]
      After k pointer changes, u is in list of size at least \( 2^k \)
      \[ \Downarrow \]
      Pointer can be changed at most \( \log |V| \) times.
- With improvement, Kruskal’s algorithm runs in time \( O(|E| \log |E| + |V| \log |V|) = O((|E| + |V|) \log |E|) = O(|E| \log |V|) \) like Prim’s algorithm.
1.1 Improved Union-Find

- It turns out that Union-Find can be improved (but without leading to an improvement of Kruskal’s algorithm)
  - Linked list representation can also be viewed as trees of height 1

Example:

```
  3
 / \      /
1  2  10  6
```

- Instead of updating root pointers when performing UNION-SET, we could just link one tree below the root of the other

Example: UNION-SET(2,6)

```
  3
 / \      /
1  2  10  6
    /  /
   8  4  5  12
```

UNION-SET and FIND-SET takes \(O(\log |V|)\) time if we always insert small tree below larger tree (trees have height \(O(\log |V|)\))

\[|E|\] FIND-SET operations takes \(O(|E|\log |V|)\) time

- If we furthermore perform path-compression, \(|E|\) Find-set operations can be performed even faster

Path-compression: When following path during FIND-SET we link traversed nodes directly to the root:

Example:

```
  X
 / |
/  |
```

Note that a lot of paths are shortened (decreasing time spent on future FIND-SET operations) without using extra time
It can be shown that $O(|E| \log^* |V|)$ is the total time used on the $O(|E|)$ FIND-SET and UNION-SET operations

- $\log^* n$ is an extremely slow growing function
  
  - Consider $g(n) = \begin{cases} 2^1 & \text{if } i = 0 \\ 2^2 & \text{if } i = 1 \\ 2^2(n-1) & \text{if } i \geq 2 \end{cases}$
  
  \[ \downarrow \]
  
  $g(0) = 2$
  $g(1) = 2^2 = 4$
  $g(2) = 2^{2^2} = 2^4 = 16$
  $g(3) = 2^{2^{2^2}} = 2^{16} = 65536$
  
  \[ \vdots \]
  
  $g(i) = 2^{2^{i-2}}$ (2-stack of height $i$)
  
  \[ \downarrow \]
  
  $g(n)$ extremely fast growing function.

- Define $\log^{(i)} n = \begin{cases} n & \text{if } i = 0 \\ \log \log^{(i-1)} n & \text{otherwise} \end{cases}$
  
  \[ \downarrow \]
  
  $\log^* n = \min \{ i \geq 0 : \log^{(i)} n \leq 1 \}$
  
  \[ \downarrow \]
  
  $\log^* n$ is minimal number of times we need to take log to get below 1
  
  \[ \downarrow \]
  
  $\log^* n$ is inverse of $g(n)$
  
  \[ \downarrow \]
  
  $\log^* n$ extremely slow growing function

- $\log^* n \leq 5$ for all practical values of $n$

- One can even prove that with path-compression $O(|E| \cdot \alpha(|V|))$ is the total time spent on $|E|$ FIND-SET operations, where $\alpha(n)$ is a function growing even slower than $\log^* n$ (Inverse Ackerman function)
  
  * $\alpha(n) < 4$ for all practical values of $n$

## 2 Shortest path

- We will now consider a problem related to minimum spanning trees; shortest paths
  
  - We already discussed how BFS can be used to find shortest paths if the length of a path is defined to be the number of edges on it
  
  - In general we have weights on edges and we are interested in shortest paths with respect to the sum of the weights of edges on a path

Example: Finding shortest driving distance between two addresses (lots of www-sites with this functionality). Note that weight on an edge (road) can be more than just distance (weight can e.g. be a function of distance, road condition, congestion probability, etc).
• Formal definition of shortest path: \( G = (V,E) \) weighted graph. Weight of path \( P = <v_0, v_1, v_2, \ldots, v_k> \) is \( w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \). Shortest path \( \delta(u,v) \) from \( u \) to \( v \) has weight

\[
\delta(u,v) = \begin{cases} 
\min\{w(P) : P \text{ is path from } u \text{ to } v\} & \text{If path exists} \\
\infty & \text{Otherwise}
\end{cases}
\]

Example: Shortest path from \( a \) to \( e \) (of length 21)

- Note:
  - If \( P = <u = v_0, v_1, v_2, \ldots, v_k = v> \) is shortest path from \( u \) to \( v \) then for all \( i < k \) \( P' = <u = v_0, v_1, v_2, \ldots, v_i> \) is shortest path from \( u \) to \( v_i \)
  - Shortest path is not necessarily part of minimum spanning tree.

Example: Minimum spanning tree for example graph:

- No (unique) shortest path exists if graph has cycle with negative weight

Example: If we change weight of edge \((h,i)\) to \(-8\), we have a cycle \((i,h,g)\) with negative weight \((-1)\). Using this we can make the weight of path between \( a \) and \( e \) arbitrarily low by going through the cycle several times

On the other hand, the problem is well defined if we let edge \((h,i)\) have weight \(-7\) (no negative cycles)
- We will only consider graphs with non-negative weights
• Different variants of shortest path problem:
  
  - *Single pair shortest path:* Find shortest path from $u$ to $v$
  - *Single source shortest path (SSSP):* Find shortest path from source $s$ to all vertices $v \in V$
  - *All pair shortest path (APSP):* Find shortest path from $u$ to $v$ for all $u, v \in V$

• Note:
  
  - No algorithm is known for computing a single pair shortest path better than solving the ("bigger") SSSP problem
  - APSP can be solved by running SSSP $|V|$ times
    
    \[
    \downarrow
    \]
    
    We will concentrate on SSSP problem

• Next time we will discuss Dijkstra’s algorithm for the SSSP problem.