1 Overview

In the previous lecture we looked at blocking flows. The idea was to repeatedly find a blocking flow $f$, and augment our flow by $f$. Total time was $(\#\text{iterations}) \cdot (\text{time to find a blocking flow}) \leq (n - 1) \cdot O(mn)$.

The next few topics will be

- Link-cut trees
- Min cost max flow

2 Link-cut Trees [1]

These can be used to find a blocking flow in time $O(m \log n)$. They will store a collection of vertex disjoint rooted trees, and allow the following operations:

1. makeTree(): makes a new vertex and puts it in a singleton tree
2. findRoot($v$): returns the root of the tree containing $v$
3. cut($v$): destroys the edge $(v, \text{parent}(v))$
4. findMin($v$): returns the lowest capacity edge on the path from $v$ to its root. If there is a tie, return the edge closest to the root.
5. subtract($v, x$): subtracts $x$ from the capacity of every edge on the $v$-root path
6. addFlow($v, x$): adds $x$ to the flow variable for every edge no the $v$-root path
7. link(($v, w$), $x$): assumes $v$ is the root of its tree, and that $v, w$ are in different trees. Makes $v$ a child of $w$ with capacity $x$.

2.1 Blocking Flow Algorithm

Initially, makeTree() $n$ times  
while(true):  
    $v = \text{findRoot}(s)$  
    if ($v == t$)  
        //$(z, \text{parent}(z))$ is a min capacity edge with weight $x$
\[(z, x) = \text{findMin}(s)\]
\[\text{subtract}(s, x)\]
\[\text{cut}(z)\]
\[\text{delete}(z, \text{parent}(z)) \text{ from the level graph}\]

else

//try to advance
if \(v\) has an outgoing edge to some \(w\) in \(L\):
    \[\text{link}((v, w), \text{capacity}(v, w))\]
else
    if \((v == s)\): break
    else for every child \(y\) of \(v\)
        \[\text{cut}(y)\]
        \[\text{delete}(y, v) \text{ from } L\]

2.2 Discussion

The basic idea of link-cut trees is to store (potentially unbalanced) trees using balanced BSTs. Every operation will run in \(O(\log n)\) time (today, we get amortized \(O(\log n)\) time). For each tree, we will maintain a preferred path decomposition: every vertex will have a preferred child:

\[
\text{preferredChild}(v) = \begin{cases} 
\text{none} & \text{if } v \text{ was the last nodee accessed in its subtree} \\
\text{the child containing the subtree containing the last accessed node in } v\text{'s subtree} & \text{o.w.}
\end{cases}
\]

An edge leading to a preferred child will be called a preferred edge. A preferred path is a maximal chain of preferred edges. Then a preferred path decomposition is a tree on the preferred edges. Link-cut trees explicitly maintain this decomposition. Each preferred path will be stored in a splay tree, keyed by depth (a higher node in the tree is smaller). Call the splay tree to store a path an auxiliary tree. Call the actual tree \(T\) a represented tree. The root of each auxiliary tree will have a pathparent pointer, telling us the parent of the top node of the path in the represented tree.

2.3 Helper Operation

\text{access}(v)\) will make the root-\(v\) path in \(T\) preferred. The implementation is as follows:

make sure \(v\) is the root of the root of the tree of auxiliary trees, \(v\) is in some auxiliary tree
\text{splay}(v)
\(v\).right.pathparent = \(v\)
\(v\).right.parent = none
\(v\).right = none
let \(w\) be the pathparent of \(v\)'s auxiliary tree
\text{splay}(w)
\(w\).right.pathparent = \(w\)
\(w\).right.parent = none
\(w\).right = \(v\)
\(v\).pathparent = none
v.parent=w
splay(v)
v.pathparent=w.pathparent
w.pathparent=none

The runtime of acess depends on the number of preferred child changes after accessing w.

2.4 Implementation of Other Operations

- findRoot(v):
  access(v)
  return the loest depth eleemtn r in v’s auxiliary tree
  access(r)

- findMin(v):
  access(v)
  return the minimum value in v’s auxiliary tree

- cut(v):
  access(v)
  access(v)
  v.left.parent=none
  v.left=None

- link(v, w):
  access(v)
  access(w)
  v.left=w
  w.parent=v

Question for the next lecture/problem set: Why are all of these operations $O(\log n)$ amortized?

References