No class tuesday.
So, no homework today.
Instead, homework thursday, due **a week from** tuesday.

1 Blocking Flows

Last time, studied shortest augmenting path.

- Strongly polynomial bound
- increasing source-sink distance
- wait a minute: can we benefit more from our shortest path computation?

Dinic's algorithm

- layered graph, based on distance from sink
- admissible arcs: those pointing toward sink
- admissible path: made of admissible arcs
- find flow that saturated an arc on every admissible path
- (idea from last time: when saturate arc, **discard** (reverse arc not admissible).
- increasesd source-sink distance by 1
- so n blocking flows will find a max-flow

How to find one?

1.1 Unit Blocking Flows

Will start by considering special case: unit capacity edges.

- dfs, like search for augmenting path
- change: conserve information about edges once traversed
- advance: follow some outgoing edge from current vertex
- retreat: current node blocked from sink. move back to parent
- eventually, reach sink: augment along current path
- seems much like aug path algorithm
- but can save info since don't create residual arcs
- once vertex is blocked, stays that way

- so when retreat on edge, can "delete" edge
- when vertex has no outgoing arcs, know it is blocked
- when augment along path, can also "delete" edges
- so total cost of blocking flow is $O(m)$.
- so find flow in $O(mn)$

Wait a minute, augmenting path is also $O(mn)$ on unit capacity! (not if have parallel edges). Wht bother?

- get other nice bounds for uncapacitated
- get similar bounds for capacitated
- better in practice

Other unit bounds:

- suppose do k blocking flows
- consider max-flow in residual graph
- decompose into paths (number=value of residual flow)
- each has length k
- paths are disjoint
- so number of paths at most m/k
- so m/k more blocking flows (or aug paths) suffice
- total time: $O(km + m/k) = O(m^{3/2})$
- similar argument gives a bound of $O(mn^{2/3})$

Bipartite matching:

- recall problem and reduction
- initial and residual graphs are *unit graphs*: every vertex either has indegree 1 or outdegree 1
- do k blocking flows, decompose as above
- note paths are *vertex* disjoint
- deduce $O(n/k)$ flow remains
- balance to get $O(m\sqrt{n})$ runtime

What breaks in general graphs?

- basic idea of advance/retreat/block still valid
- every advance is paid for by retreat or augment, ignore
- still $O(m)$ retreats in a phase
- unfortunately, augment only zaps one edge (min-capacity on path)
- must charge n (augmenting path work) to zapped edge
- $O(mn)$ time bound for blocking flow
- $O(mn^2)$ for max-flow

1.2 Data Structures

goal: preserve info:

- zapped edge breaks aug path into 2 pieces
- both pieces still legitimate for aug.
- if encounter vertex on piece, want to jump to head of piece and continue from there
- still problem if must traverse all edges to do augment, so also want to augment (reducing all edge capacities and splitting path) in constant time?

details:

- maintain in-forest of augmentable (nonsaturated) edges
- initially all vertices isolated
- "current" vertex always a root of tree containing source
- advance:
  - "link" current (root) vertex to head of arc
  - merges two trees
  - jump to root of (new) current tree
- retreat:
  - "cut" trees into separate pieces
  - tail of cut edge becomes root
- augment:
  - occurs when reach sink
  - source/sink in same tree

  - find min-capacity c on tree path from source to sink
  - decrease all capacities on this path by c
  - cut at edge that drops to 0 capacity
- four operations: link, cut, min-path, add-path
- supported by *Dynamic Tree* data structure of (surprise) Sleator-Tarjan
- basic idea: path
  - maintain ordered list of vertices on path in balanced search tree
  - store "deltas" so that true value of node is sum of values on path to it
  - easy to maintain under rotations
  - to add x to path from v, splay successor of v to root, add x to root of left subtree
  - similarly, maintain at each node min of its subtree

1.3 Scaling

[Gabow '85]. Also [Dinitz '73], but appeared only in Russian so, as often the case in this area, the american discovery was much later but independent.
General principle for applying "unit case" to "general numeric case".
Idea: number is bits; bits are unit case!
Scaling is reverse of rounding.

- start with rounded down value (drop low order bits)
- put back low order its, fixup solution

big benefit: aside from scaling phase, often as simple as unit case (eg no data structures!)
Basic approach:

- capacities are $\log U$ bit numbers
- start with all capacities 0 (max-flow easy!)
- roll in one bit of capacity at a time, starting with high order and working down.
- after rollin, update max-flow by finding max-flow in residual graph
- effect of rollin:
  - double all capacities and flow values
  - double all residual capacities
  - add 1 to some residual capacities

- after $\log U$ roll-ins, all numbers are correct, so flow is max-flow

Analysis of one scaling phase:

- In blocking flow, we saw 2 costs: retreats and augments
- bounded retreat cost by $O(m)$ per blocking flow, $O(mn)$ total
- now bound augment cost.
- claim: at start of phase, residual graph flow is $O(m)$
- each augment step reduces residual flow by 1
- thus, over whole phase, $O(m)$ augments
- pay n for each, total $O(mn)$
- proof of claim:
  - before phase, residual graph had a capacity 0 cut $(X, \overline{X})$
  - each edge crossing it has capacity 0
  - then roll in next bit
  - each edge crossing cut has capacity increase to at most 1
  - cut capacity at most m, bound flows value.
- Summary: $O(mn)$ for retreats and augments in a phase.
- $O(\log U)$ phase
- $O(mn \log U)$ time bound for flows.

In recent work, Goldberg-Rao have extended the unit-cost bounds to capacitated graphs using scaling techniques.