0.0.1 Shortest Augmenting Path

Strongly polynomial.
Instead of being directly "primal" greedy, tackle a "dual" function that bounds residual.

- Idea: if s, t far apart, not much flow can fit in network
- So try to push up s, t residual distance.

Lemma: For shortest augment, $(s.i)$ and (i, t) distance in residual graph non-decreasing.

- Among i that got closer to s
- Consider closest to s (after change)
- i has parent j on new shortest path
- j didn't get closer to s
- so (j, i) path got shorter
- so didn't used to have residual (j, i) edge
- so flow added from i to j
- so j was farther than i from s
- now they swapped places
- but j didn't get closer!
- so i must be farther—contra.

Lemma: at most $mn/2$ augmentations.

- Consider edge (i, j) saturated by augmenting path
- Before used again, must push flow on (j, i)
- In first aug, i was closer than j to s
- In next, j was closer than i
- Since no distances go down, must have increased distance of i.
- only happens n times per edge

Running time: $O(m^2 n)$.

- Strongly polynomial
- Note reason: distance is an integer $< n$

1 Blocking Flows

Extension of 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 graph: of **only** admissible arcs
- admissible path: made of admissible arcs
- find flow in admissible graph that saturates an arc on every admissible path
- don't need max-flow. so when saturate arc, **discard** (reverse arc not admissible. easier than max-flow.
- increases source-sink distance by 1
  - No longer have admissible path in layered graph
  - So every path uses at least one nonadmissible edge
  - Augmentations create no edge hopping a level
  - So cannot make up for distance lost traversing nonadmissible arc
  - So lose at least one unit distance on any path
- so n blocking flows will find a max-flow

How to find one?
2005 got here from point B

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). Why bother?

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

2011 lecture 8 start
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: $O(km + m^2/k) = O(m^{3/2})$
- 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
- (still better than shortest augmenting path)
- And, can say a little better:
  - each augment adds a unit of flow
  - so, if value of flow is f, total cost nf
  - even if not unit capacities
  - so, blocking flows are $O(mn + nf) = O((m + f)n)$
  - compare to naive augmenting path of $O(mf)$
  - good for f large, but not too large.
  - note: advances charged per blocking step, while augments amortized over all blocking steps

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 Blocking Flows

As before, do $\log U$ bit shifts.

- Then, use blocking flow algorithm to consume new residual flow
- Key benefit: total flow per phase small

Short analysis:

- Scaling phase introduces m new flow
- Above we saw blocking flow cost $O((m+f)n)$
- So, cost here is $O(mn)$
- over all phases, $O(mn \log U)$

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 other unit-cost bounds ($m^{3/2}$, $mn^{2/3}$) to capacitated graphs using scaling techniques.
1:25 from point B.

2 Push-Relabel

covered in recitation
Goldberg Tarjan. (earlier work by Karzanov, Dinic)
Two "improvements" on blocking flows:

- be lazy: update distance labels only when necessary

- make your work count: instead of waiting till find augmenting path, push flow along each augmentable edge you find (no augmentation work!).

Time bounds still $O(mn)$-like (no better/worse than blocking flows) but:

- some alternative approaches to get good time bounds without fancy data structures (or scaling)
- fantastic in practice—best choice for real implementations.

What did we use layered graph for?

- maintain distances from sink
- send flow on "admissible" arcs (v, w) have $d(v) = d(w) + 1$.
- when source-sink distance exceeds n, have max-flow

Distance Labels:

- lazy measure of distance from sink
- $d(t) = 0$
- if residual (v, w) has positive capacity, then $d(v) \leq d(w) + 1$
- lower bounds on actual distances
- so when $d(s) = n$, done
- arc is *admissible* if $d(v) = d(w) + 1$
- corresponds to level graph (good to push flow)
- if no admissible arc out of edge, can *relabel*, increasing distance, until get one.
- distances only increase, so n relabels per vertex
- allows same bounds as blocking flow $O(n^2m)$, without explicit bfs phases.

Avoid augmenting paths:

- consider advance/retreat process
- instead of waiting till hit sink to augment, augment as advance
- augment (v, w) amount is min of flow reaching v and (v, w)
- "unaugment" when retreat—augment reverse arc!
- means some vertices have more flow incomming than outgoing

Preflow:

- assigns flow to each edge
- obeys capacity constraints
- at all vertices except sink, net flow into vertex positive:

$$e(v) = \sum_w f(w,v) \geq 0.$$

- This quantity is called the *excess* at node v
- excess always nonnegative
- node with positive excess is *active*
- if no node has excess, preflow is a flow

Decomposition: any preflow is a collection of

- cycles
- paths from source to active nodes and sink

Push relabel algorithm:

- maintain valid distance labeling
- find active node (one with excess)
- push along admissible arc (towards sink)
- if no admissible arcs, relabel node (increasing distance)
- keep pushing flow down till reaches sink.

Initialize:

- saturate every arc leaving s (set $f(s,v) = u(s,v)$, so $e(v) = u(s,v)$
- set $d(s) = n$ (to absorb blocked flow—know will get there eventully)
- (creates valid distance labeling, since no residual arc from s to any vertex)
- gives preflow, make into flow by *pushes* and *relabels*

Push:

- applies if active node v has admissible outgoing arc (v,w)
- send $min(e(v), u_f(v,w))$ (residual capacity) from v to w
  - *saturating push* if send $u_f(v,w)$
  - nonsaturating if send $e(v) < u(v,w)$

Relabel:

- applies to active node v without admissible arcs
- set $d(v) = 1 + \min d(w)$ over all $(v, w) \in G_f$ (increases $d(v)$)

Generic algorithm: while can push or relabel, do so.

- generally 2 phases:
- first max-preflow
- then return excess to sink

Correctness: Any active vertex can push or relabel:

- if admissible arc, push
- if no admissible arc, then (since active) net flow in is positive, so residual arc out
- thus, relabel to larger than current

Correctness: if no active vertex, have max-flow

- no active vertex means have flow
- suppose have augmenting path in residual graph
- working backwards, each residual arc increases distance at most 1
- but we know $d(s) = n$, contra.

Analysis I: no distance label exceeds $2n$

- relabel only on active vertex
- decomposition of preflow shows residual path to source
- source has distance n
- v has distance only n more.

Deduce: $O(n)$ relabels per node,

- $O(n^2)$ relabels
- total cost $O(mn)$. (why?)

Analysis II: saturating pushes

- suppose saturating push on (v, w)
- can't push on (v, w) again till push on (w, v)
- can't do that till $d(w)$ increases

- then to push (v, w), $d(v)$ must increase
- $d(v)$ up by 2 each saturating push
- $O(n)$ saturating pushes per edge
- work $O(nm)$.
- (same arg as for shortest augmenting path)

Analysis III: nonsaturating pushes.

- potential function: active set S, function

$$\sum_{v \in S} d(v)$$

- intially 0
- over course of alg, relabels increase qtty by $O(n^2)$
- saturating push increases by at most $2n$ (new active vertex) (total $O(mn^2)$)
- nonsaturating push decreases by at least 1 (kills active vertex)
- so $O(mn^2)$ nonsaturating pushes

Summary: generic push-relabel does $O(mn^2)$ work.
Waitaminit, how find admissible arc?

- keep list of edges incident on vertex
- maintain "current arc" pointer for each vertex
- look for admissible arc by advancing current pointer
- when reach end, claim can relabel
- $O(n)$ relables per node, so each arc scanned $O(n)$ times
- $O(mn)$ work searching for current arc.

Discharge method:

- What bottleneck? Nonsaturating pushes.
- idea: bound nonsaturating pushes in terms of other operations
- discharge operation: push/relabel vertex till becomes inactive
- note ends with at most 1 nonsaturating push
- so if bound discharges, also bound bad pushes

FIFO:

- keep active vertices in queue
- go through queue, discharging vertices
- add new active vertices at end of queue

Analysis:

- phase 1: original queue vertices
- phase $i+1$: vertices enqueue in phase i
- only 1 nonsaturating push per vertex per phase (0 excess, remove from queue)
- claim $O(n^2)$ phases:
$$\phi = \max_{v \text{ active}} d(v)$$
- ϕ increases only during relabels: $O(n^2)$ total
- if no relabel, then at end of phase max distance decreases!
  - But total increase $O(n^2)$, so $O(n^2)$ relabel phase.
- $O(n)$ nonsaturating pushes per phase, so $O(n^3)$ nonsat pushes.

3 Fancy Push Relabel Algorithms

3.1 Excess Scaling

Way to achieve $O(nm)$ without data structs, but must discard strong polynomiality.
Basic idea: make sure your pushes send lots of flow.
Instead of highest level, do lowest level!
Can explain by bit shifts, but slightly cleaner to talk about Δ-phases:

- starts with all excesses below Δ
- ends with all excesses below $\Delta/2$
- initially $\Delta = U$
- when $\Delta < 1$, done.
- $O(\log U)$ phases
- each takes $O(nm)$ time
- so $O(nm \log U)$.

Doing a phase: make sure pushes are big

- *large excess* nodes have $e(v) \geq \Delta/2$
- push maximum possible without exceeding Δ excess at destination
- (turns some potentially saturating pushes nonsaturating)
- to ensure big push, always push from large excess with smallest label
- if push nonsaturating, has value at least $\Delta/2$
  - large excess source has at least $\Delta/2$,
  - small excess dest can receieve at least this much without going over Δ

Claim: $O(n^2)$ nonsaturating pushes per phase:

- potential function
$$\Phi = \sum d(i)e(i)/\Delta$$
- relabel increases by total of $O((n^2\Delta)/\Delta) = O(n^2)$
- saturating push decreases
- nonsaturating push sense $\Delta/2$ downhill: decrease by 1/2
- so $O(n^2)$ nonsaturating pushes.
- note: in this alg, *saturating pushes* form bottleneck.

Deduce: $O(nm + n^2 \log U)$ running time.

3.2 Highest Label

Highest label (more sophisticated fifo):

- idea: avoid sending nonsaturating pushes down a path more than once
- keep vertices arranged by distance label (in buckets)
- always discharge from highest label (flow "accumulates" into fewer piles as moves towards sink)
- easy analysis: if n discharges without relabel, done.
- so 1 relabel every n discharges
- so $O(n^3)$ discharges/nonsaturating pushes.
- so $O(n^3)$ time since relabels, sat pushes $O(nm)$.

Keeping track of level:

- like bucketing shortest paths algorithms

- keep pointer to current highest level
- raise when relabel if necessary
- advance downward to find next nonempty bucket
- total raising $O(n^2)$
- also bound total descent.

Better analysis:

- Basic idea:
  - block of excess "originates" with some saturating push or relabel
  - then flows downhill on nonsaturating pushes
  - account by combination bound on originating pushes and distance travelled downhill.
- Consider phase between two relabels
- Current arcs
  - each vertex has a "current arc" along which most recently pushed.
  - within phase, these form a forest.
  - all nonsaturating pushes travel along forest edges
- Consider a nonsaturating push from a max-label node
- work backwards along current arcs until get to leaf
- why is it a leaf?
  - either flow arrived because of a saturating push
  - or came with relabel of leaf
- "blame" push we just sent on this leaf

Study "trajectories" of flow.

- Saturating pushes and relabels "originate" some excess
- then it flows down nonsaturating pushes
- until participates in new saturating push or relabel.
- note nonsaturating push never "splits" excess (all goes in one push)
- so nonsaturating pushes of a block of excess form a path— "trajectory"
- trajectories might merge.

  - If so, highest label rule says excesses will merge too
  - so end one trajectory
  - So can consider trajectories vertex disjoint

- Two kinds of pushes. “Short” within $n/\sqrt{m}$ of originating event, “long” otherwise.

- short pushes

  - short path has $O(n/\sqrt{m})$ nonsat pushes
  - each starts with one of $O(nm)$ sat pushes or relabels
  - so $O(n^2\sqrt{m})$ total nonsat pushes

- long pushes

  - Consider long push. Work backwards along current arcs of nonsat. pushes excess followed, till get to leaf
  - Why is it a leaf? because a sat. push or relabel delivered excess there
  - so, leaf must be more than $n/\sqrt{m}$ distance from excess (else short push)
  - but, these trajectories are vertex disjoint!
  - so, at most *sqrtm* distinct trajectories in phase
  - define *length* of phase as total drop in maximum distance
  - claim: sum of phase lengths $O(n^2)$:
    * decreases must be balanced by increases
    * total increase (relabels) $O(n^2)$
  - number of long phases at most $n^2/(n/\sqrt{m}) = O(n\sqrt{m})$
  - phase has only n pushes
  - so total $O(n^2\sqrt{m})$

Best known strong poly bound for push-relabel without fancy data structs.

3.3 Wrapup

Text discusses practical choice, argues for:

- shortest aug path simple, often good enough
- highest label best in practice if time to code
- excess scaling also good.

Open: $O(nm)$-ish without scaling, data structs