A Randomized Maximum-Flow Algorithm

JOSEPH CHERIYAN TORBEN HAGERUP

Fachbereich Informatik, Universität des Saarlandes
D–6600 Saarbrücken, West Germany

The problem of finding a maximum flow in a network has significant theoretical ramifications and many practical applications. It has attracted much algorithmic research. The best known strongly polynomial algorithm for computing a maximum flow in a network with n vertices and m directed edges takes $O(nm\log(n^2/m))$ time [GT88]. We present a randomized maximum-flow algorithm, called the PLED (*Prudent Linking Excess Diminishing*) algorithm, whose expected running time is $O(nm + n^2(\log n)^3)$; this is $O(nm)$ for all except relatively sparse networks. The algorithm is always correct and in the worst case, which occurs with negligible probability, it takes $O(nm\log n)$ time.

The PLED algorithm combines a variant of the Ahuja-Orlin excess scaling algorithm [AO87], which is based on the generic maximum-flow algorithm of Goldberg and Tarjan [GT88], with the dynamic trees data structure [ST83, ST85]. The following is a brief sketch of the ideas and techniques used in the paper.

Following Ahuja and Orlin, we maintain a parameter Δ, which is a measure of the maximum flow excess of a vertex and of the maximum amount of flow sent by a single operation. Initially, Δ is less than or equal to the maximum edge capacity, and $\Delta = 0$ at termination. The execution of the PLED algorithm is partitioned into *phases* such that Δ stays fixed during each phase and decreases between consecutive phases. In order to achieve a bound on the number of phases that is independent of the maximum edge capacity, the algorithm decreases Δ by as large a factor (≥ 2) as possible, rather than by a constant factor.

We introduce the notion of a *premature-target-relabeling* (PTR) event. Each PTR event contributes $O(\log n)$ amortized time to the overall running time. There is a trivial $O(nm)$ bound on the number of PTR events ($\sharp ptr$). In order to improve upon this bound, we focus on the ordering of the adjacency lists of the network vertices. The implementation of the generic maximum-flow algorithm in [GT88, pp. 929] uses a fixed but arbitrary ordering. In contrast, the PLED algorithm randomly permutes each adjacency list at the start. Further, whenever the algorithm updates the label of a vertex v, it randomly permutes v's adjacency list. We model the execution of the algorithm by an adversary game on bipartite graphs and show that the algorithm can win by using a randomized strategy, i.e., we show that the expected value of $\sharp ptr$ is $O(nm/\beta + n^2\beta)$, for any $\beta = \Omega(\log n)$, if the adjacency lists are randomly permuted.

Several previous maximum-flow algorithms use the dynamic trees data structure, and all these algorithms invoke $\Omega(nm)$ dynamic trees operations. The PLED algorithm uses "prudent linking" (i.e., inserting an edge into the data structure only when a large amount of flow, relative to Δ, can be sent over the edge) to control the number of dynamic trees operations. We show that this number is $O(nm/\beta + n^2\beta\log n + \sharp ptr)$, for any β with $2 \leq \beta = n^{O(1)}$. Additional novel ideas are used in the analysis.

This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

1. Preliminaries

We assume that the reader is familiar with the generic maximum-flow algorithm in [GT88] and refer to [GT88] for definitions of the terms *network*, *source* s, *sink* t, *edge capacity* $cap(v,w)$, *flow*, *maximum flow*, *preflow* f, *flow excess* $e(v)$ of a vertex v, *residual capacity* $rescap(v,w)$, *valid labeling* d, *active vertex*, *residual graph*, *push*, *saturating push*, and *nonsaturating push*. Let $G = (V,E)$ denote the digraph (assumed symmetric) corresponding to the network. Let $n = |V|$, $m = |E|$ and note that $n \geq 2$.

Except for one minor difference, the PLED algorithm is an instance of the generic maximum-flow algorithm of Goldberg and Tarjan [GT88]. The difference (namely, the value of a push over an edge (v,w) may be less than $\min\{e(v), rescap(v,w)\}$) does not affect the essential properties of the generic algorithm. In particular, the PLED algorithm correctly computes a maximum flow.

The *dynamic trees* data structure of Sleator and Tarjan [ST83, ST85] maintains a set of vertex-disjoint rooted trees in which each tree edge has an associated real value. We shall need the following dynamic trees operations (each tree edge is considered to be directed toward the root, i.e., from child to parent):

find root(v):	Find and return the root of the tree containing vertex v.
find value(v):	Find and return the value of the edge from vertex v to its parent; if v is a tree root, then return ∞.
add value(v,c):	Add real number c to the value of every edge on the path from vertex v to *find root*(v).
link(v,w,c):	Combine the trees containing vertices v and w by making w the parent of v and giving the value c to the new edge from v to w. This operation does nothing if v and w are in the same tree or if v is not a tree root.
cut(v):	Break the tree containing vertex v into two trees by deleting the edge from v to its parent. This operation does nothing if v is a tree root.
find bottleneck(v,c):	If there is an edge on the path from vertex v to *find root*(v) whose value is $\leq c$, then return the nearest ancestor w of v with *find value*$(w) \leq c$; otherwise return *find root*(v).

A sequence of k dynamic trees operations, starting with a collection of n single-vertex trees, can be executed in $O(k\log n)$ time. (The *find bottleneck* operation is non-standard, but can be implemented within this time bound.)

The generic algorithm uses a valid labeling $d : V \to \mathbb{N} = \{0, 1, \ldots\}$. We call an edge (v, w) *eligible* if $rescap(v, w) > 0$ and $d(v) = d(w) + 1$. Each vertex $v \in V$ has an adjacency list, which consists of edges $(v, w) \in E$. For each $v \in V$, the first eligible edge in v's adjacency list is called its *current edge* and denoted by $ce(v)$ (possibly, $ce(v)$ does not exist).

We use the dynamic trees data structure to maintain a spanning forest F of G containing a subset of the current edges, where the value associated with an edge in F is its residual capacity. The algorithm stores $f(v, w)$ and $f(w, v)$ explicitly iff $(v, w) \notin F$ and $(w, v) \notin F$. Whenever a tree edge (v, w) is cut, we must find its associated value (i.e., $rescap(v, w)$) and then compute the current values of $f(v, w)$ and $f(w, v)$. Also, when the algorithm terminates, $f(v, w)$ and $f(w, v)$ must be computed for all $(v, w) \in F$.

We also use the Fibonacci heaps data structure [FT87] and need the following heap operations: *make heap*, *is empty*, *insert*, *delete*, *find min*, *delete min*, and *decrease key*. The total time for a sequence of k *delete* or *delete min* operations and l other heap operations, starting with no heaps, is $O(l + k \log n)$, where n is the maximum number of items in any heap.

Let $S(\Delta) = \{\Delta_1, \Delta_2, \ldots\}$ denote the set of positive values assigned to the parameter Δ in an execution of the PLED algorithm. A phase during which the parameter Δ has the value $\Delta_i \in S(\Delta)$ is called the Δ_i-*phase*. Every push that occurs during the Δ_i-phase has value $< 2\Delta_i$, and a push is called *large* if its value is $\geq \Delta_i/2$, otherwise it is called *small*.

2. The PLED algorithm

The implementation of the generic algorithm in [GT88, pp. 929] repetitively selects an active vertex and applies a *push/relabel* step to it. Rather than selecting an arbitrary active vertex, it turns out to be advantageous to select an active vertex with a relatively large flow excess. Focus on the Δ_i-phase, $\Delta_i \in S(\Delta)$, of the PLED algorithm. At the start of the phase $e(v) < 2\Delta_i$, for all $v \in V - \{s, t\}$. The algorithm selects an active vertex v only if $e(v) \geq \Delta_i$; further, among all such vertices it selects one with minimum distance label. It is easily seen that this selection rule, together with the bound of $2\Delta_i$ on the value of a push, guarantees that for all $v \in V - \{s, t\}$, $e(v) < 3\Delta_i$ always.

After selecting v, the algorithm applies the *macropush* procedure to v. The maximum amount of flow sent by *macropush* over a single edge emanating from v, denoted $limflow(v)$, is defined as follows: If $e(v) < 2\Delta_i$, then $limflow(v) = e(v)$, otherwise $limflow(v) = e(v)/2$. This achieves two things: Firstly, the amount of flow sent is $< 2\Delta_i$ always, as claimed above; secondly, if $e(v) \geq 2\Delta_i$, then the algorithm attempts to make $e(v)$ equal to zero during the Δ_i-phase by selecting v twice (this would not be achieved by letting $limflow(v) = \min\{e(v), 2\Delta_i\}$).

There are now two cases, depending on whether or not v is the root of its tree in F. If $v \neq findroot(v)$, then *macropush* uses the dynamic trees data structure to send flow from v, over a path of current edges, to $findbottleneck(v, limflow(v))$ by invoking a constant number of operations. The algorithm never executes a nonsaturating push operation over an individual edge; instead, it sends flow over a path in F by invoking operations on the data structure. Further, no edge in F becomes saturated while the algorithm is sending flow over a path in F; in other words, each saturating push operation is executed over an individual edge.

Now suppose that $v = findroot(v)$. Then *macropush* executes zero or more saturating pushes over edges emanating from v, possibly followed by a sequence consisting of a *link* operation on an edge (v, w) and a nonsaturating push over (v, w). After each saturating push notice that $e(v)$ decreases, and therefore $limflow(v)$ may also decrease. One alternative would be to continue *macropush* as long as $limflow(v) > 0$; however, this might possibly lead to too many nonsaturating pushes and, indirectly, to too many link operations. Another alternative would be to terminate *macropush* after executing one saturating push; however, this would lead to too many *select* steps overall. An efficient compromise turns out to be to terminate *macropush* when $limflow(v)$ becomes less than $\Delta_i/2$, thereby ensuring that every link operation is followed by a large nonsaturating push.

Two heaps are used to implement the *select* procedure efficiently. The *e_heap* is a Fibonacci heap containing all vertices $v \in V - \{s, t\}$ with $e(v) < \Delta$. The heap order is maintained according to the key $-e(v)$. The *e_heap* is needed for efficient updating of Δ. A Fibonacci heap is used because only $O(1)$ amortized time can be allowed per *decrease key* operation. The *d_heap* is an ordinary heap containing all vertices $v \in V - \{s, t\}$ with $e(v) \geq \Delta$. The heap order is maintained according to the key $d(v)$. The *d_heap* is needed for efficiently selecting a vertex v with minimum distance label among those with $e(v) \geq \Delta$.

The algorithm uses the following procedures:

initialize;
  Perform the standard initialization of the variables f, d and e [GT88, pp. 925] and of the *e_heap*, the *d_heap* and the dynamic trees data structure;
  For each $v \in V$, randomly permute the adjacency list of v;
  $\Delta := \max(\{e(v) \mid v \in V - \{s, t\}\} \cup \{0\})$;
  if $\Delta = 0$ then stop; /* the current flow f is maximum */
  for $v \in V - \{s, t\}$ do *heap insert*(v);

heap insert$(v : vertex)$;
  if $e(v) \geq \Delta$
  then *insert*(v, d_heap) /* key $= d(v)$ */
  else *insert*(v, e_heap); /* key $= -e(v)$; possibly $e(v) = 0$ */

treecut$(x : vertex)$;
  Let $(x, w) = ce(x)$;
  $f(x, w) := cap(x, w) - find\,value(x)$; $f(w, x) := -f(x, w)$;
  $cut(x)$;

saturate$((v, w) : edge)$;
  /* Push $rescap(v, w)$ units of flow over (v, w); v is not in either heap */
  $c := rescap(v, w)$;
  $f(v, w) := f(v, w) + c$; $f(w, v) := f(w, v) - c$;
  $e(w) := e(w) + c$; $e(v) := e(v) - c$;
  if $w \notin \{s, t\}$ then
    if $e(w) < \Delta$
    then *decrease key*(c, w, e_heap)
    else begin *delete*(w, e_heap); *insert*(w, d_heap); end;

treepush$(v : vertex;)$;
  /* Cut and saturate the tree edge in F nearest to v whose value is $\leq limflow(v)$, and then send $limflow(v)$ units of flow from v to $findroot(v)$; v is not in either heap */
  $x := findbottleneck(v, limflow(v))$;
  if $x \notin \{v, s, t\}$ then *delete*(x, e_heap);
  if $x \neq findroot(x)$ /* $ce(x)$ is the bottleneck */
    then begin *treecut*(x); *saturate*$(ce(x))$; end;
  if $x = v$ then exit(*treepush*);
  add value$(v, -limflow(v))$;
    /* a nonsaturating push from v to x */
  $e(x) := e(x) + limflow(v)$; $e(v) := e(v) - limflow(v)$;
  if $x \notin \{s, t\}$ then *heap insert*(x);

relabel$(v : vertex)$;
  for all $u \in V$ with $ce(u) = (u, v)$ do
    if $(u, v) \in F$ then *treecut*(u);
  $d(v) := 1 + \min\{d(w) \mid (v, w) \in E$ and $rescap(v, w) > 0\}$;
  Randomly permute the adjacency list of v;

```
lim flow(v : vertex) : real;
    return( if e(v) ≥ 2Δ then e(v)/2 else e(v) );

macropush(v : vertex);
    /* Send flow from v until v is relabeled or until lim flow(v)
    decreases to < Δ/2 due to saturating pushes or until flow is
    sent from v over a path in F or until cut(v) is executed */
    if v ≠ find root(v)
    then treepush(v) else
      begin
        while ce(v) ≠ nil and rescap(ce(v)) ≤ lim flow(v)
        do saturate(ce(v));
        if ce(v) = nil
        then relabel(v) else
          if lim flow(v) ≥ Δ/2 then
            begin /* rescap(ce(v)) > lim flow(v) ≥ Δ/2 */
              link(v, w, rescap(v, w)), where (v, w) = ce(v);
              treepush(v);
            end;
      end;
    heap insert(v);

select: vertex;
    /* Return an active vertex v with e(v) ≥ Δ and minimum d(v)
    or decrease Δ and then select v as before or stop */
    if is empty(d_heap) then
      begin /* e(v) < Δ for all active vertices v */
        Δ := min{Δ/2, −find min(e_heap)};
        if Δ = 0 then stop; /* the current flow f is maximum */
        while −find min(e_heap) ≥ Δ
        do begin v := delete min(e_heap); insert(v, d_heap); end;
      end;
    return(delete min(d_heap));

The PLED Algorithm;
    initialize;
    loop
      v := select;
      macropush(v);
    forever;
```

Theorem 1 [GT88]: The PLED algorithm correctly finds a maximum flow. ∎

3. Preliminaries for the analysis

Fact 1 [GT88]: The total number of saturating pushes is $O(nm)$. For all $v \in V$, $0 \le d(v) \le 2n-1$ throughout the execution, and $d(v)$ increases in each application of *relabel* to v. Hence, the total number of relabeling operations is $O(n^2)$. ∎

Fact 2: For each vertex $v \in V - \{s,t\}$, $e(v) < 2\Delta$ at the start of each phase, and $e(v) < 3\Delta$ always. The value of every push is $< 2\Delta$. ∎

Fact 3: Between consecutive phases, Δ decreases by a factor of 2 or more. ∎

Fact 4: For each $(v,w) \in E$, after a cut operation on (v,w) (i.e., $cut(v)$ with $(v,w) = ce(v)$) and before the next link operation (or before termination) (v,w) becomes noneligible. ∎

Fact 5: For each $(v,w) \in E$, after a link operation on (v,w) and before the next link operation (or before termination) there is a nonsaturating push over (v,w). ∎

Fact 6: All nonsaturating pushes are large and are executed by *treepush*. For each $(v,w) \in E$, a push over (v,w) is saturating if and only if $(v,w) \notin F$. ∎

Fact 7: For each $v \in V$, any increase of $e(v)$ while $v \neq find\,root(v)$ is due to saturating pushes over edges entering v. ∎

Fact 8:
(1) If a vertex $v \in V - \{s,t\}$ has $e(v) \ge \Delta$ after a call of *macropush*, then $e(v)$ does not change until v is selected.
(2) If a vertex $v \in V - \{s,t\}$ has $e(v) \ge \Delta$ and $v \neq find\,root(v)$ after a call of *macropush*, then $e(v)$ does not increase until either a $cut(v)$ operation is executed or until $e(v)$ decreases to zero due to one or two steps that select v. ∎

Fact 9: Let $\Delta_i \in \mathcal{S}(\Delta)$. For each $(v,w) \in E$, the increase of $f(v,w)$ during the Δ_i-phase and all subsequent phases is at most $20n^2\Delta_i$.

Proof sketch: Using the potential function $\Phi = \sum_{v \in V} e(v) \cdot d(v)$, it can be shown that the total increase of $f(v,w)$ during the Δ_i-phase is at most $10n^2\Delta_i$. The result follows by summing over the remaining phases and using Fact 3. ∎

The analysis uses the notion of an *undirected edge* $\{v,w\}$, i.e., a pair of (directed) edges (v,w) and (w,v). Define the capacity of an undirected edge $\{v,w\}$ to be $cap\{v,w\} = cap(v,w) + cap(w,v)$. For $v \in V$, let $deg(v)$ denote the number of undirected edges incident with v.

Let β be a real number such that $2 \le \beta = n^{O(1)}$ and $\beta = \Omega(\log n)$; we fix $\beta = 2 + \log n$ to get the main result. Define the *status* of an undirected edge $\{v,w\}$ as follows: $\{v,w\}$ is said to be *small* if $cap\{v,w\} < \Delta/\beta$, *medium* if $\Delta/\beta \le cap\{v,w\} \le 40n^2\Delta$, and *huge* if $cap\{v,w\} > 40n^2\Delta$. (Note that the status may change during the execution.) Consider a huge edge $\{v,w\}$. By Fact 9, one of the directed edges (v,w) or (w,v), say (v,w), will never again be saturated. We call (v,w) a *forward huge edge* and (w,v) a *reverse huge edge*.

Denote by $\sharp links$ the total number of link operations executed on F, and by $\sharp selects$ the total number of *select* steps over the whole execution. Also, let $\sharp mediumlinks$ ($\sharp hugelinks$, respectively) denote the sum over all $\Delta_i \in \mathcal{S}(\Delta)$ of the number of link operations on edges that are medium (huge, respectively) during the Δ_i-phase.

Lemma 1: The running time of the PLED algorithm is $O(nm + \sharp selects \cdot \log n)$. ∎

It turns out that $\sharp selects$ depends on $\sharp links$. For this reason, we first count $\sharp links$.

4. Counting the number of link operations

It is easy to see that $\sharp links = \sharp mediumlinks + \sharp hugelinks$, because no small edge is ever inserted in F.

4.1. Bounding $\sharp mediumlinks$. The next lemma, which may be regarded as a generalization of Lemma 6 in [AO87], is the key lemma used to bound $\sharp mediumlinks$.

For $\Delta_i \in \mathcal{S}(\Delta)$, let $\mathcal{D}(\Delta_i)$ denote the increase of $\sum_{v \in V} d(v)$ during the Δ_i-phase. Note that $\sum_{\Delta_i \in \mathcal{S}(\Delta)} \mathcal{D}(\Delta_i) = O(n^2)$, by Fact 1.

Lemma 2: For $\Delta_i \in \mathcal{S}(\Delta)$, consider the Δ_i-phase. Let $V' \subseteq V$, $n' = |V'|$ and let l be a constant. Then the sum over all $u \in V'$, of the number of times $e(u)$ decreases by Δ_i/l plus the number of times $e(u)$ increases by Δ_i/l (due to one or more pushes over edges incident with u) during the Δ_i-phase is $O(n'n + \mathcal{D}(\Delta_i))$.

Proof: Let $V' = \{u_1, u_2, \ldots, u_{n'}\}$. For each vertex $v \in V$, define its *fooling height* to be

$$\phi'(v) = 2 \cdot \max_{i_1, i_2, \ldots, i_{n'} \in \mathbb{N}} |\{i_j : 1 \le j \le n' \text{ and } d(u_j) \le i_j < d(v)\}|.$$

(Intuitively, $\phi'(v)$ counts twice the maximum number of "possibly occupied distance levels" between v and t, where a vertex $u_j \in V'$ is allowed to occupy any one level numbered at least $d(u_j)$.)
Define $\phi(v) = \phi'(v) + (\text{if } v \in V' \text{ then 1 else 0})$. $\phi(v)$ has the following properties:
(1) $\forall v \in V : 0 \leq \phi(v) \leq 2n'$
(2) $\forall v, w \in V : d(v) > d(w) \Rightarrow \phi(v) \geq \phi(w)$
(3) $\forall u \in V', v \in V : d(u) > d(v) \Rightarrow \phi(u) > \phi(v)$
(4) $\forall u \in V', v \in V : d(v) > d(u) \Rightarrow \phi(v) > \phi(u)$
(5) For $v \in V$, suppose that $d(v)$ increases by k due to a relabeling of v. Then $\phi(v)$ increases by $\leq 2k$ and, further, for any $w \in V - \{v\}$ $\phi(w)$ does not increase.

Consider the potential function $\Phi = \sum_{v \in V} \phi(v) \cdot e(v)$. At the start of the Δ_i-phase, $\Phi \leq 4n'n\Delta_i$, and $\Phi \geq 0$ always (by Property (1) and Fact 2). Φ does not increase due to push operations (by Property (2)), and if a distance label increases by k, then Φ increases by $\leq 6k\Delta_i$ (by Property (5) and Fact 2). It follows that the total increase of Φ during the phase is $\leq 6\Delta_i \cdot \mathcal{D}(\Delta_i)$. Consequently, the total decrease of Φ during the phase is $\geq 4n'n\Delta_i + 6\Delta_i \cdot \mathcal{D}(\Delta_i)$.

For a vertex $u \in V'$, whenever $e(u)$ changes by an amount c (due to one or more pushes over edges incident with u), then the value of Φ decreases by at least c (by Properties (3) and (4)). Summing over all vertices $u \in V'$, the number of times $e(u)$ decreases by Δ_i/l plus the number of times $e(u)$ increases by Δ_i/l is $O(n'n + \mathcal{D}(\Delta_i))$. ∎

For $\Delta_i \in \mathcal{S}(\Delta)$ and $v \in V$, let $M(v, \Delta_i)$ denote the set of undirected edges incident with v that are medium during the Δ_i-phase. For $\Delta_i \in \mathcal{S}(\Delta)$ and $v \in V$, the Δ_i-phase is said to *hit* vertex v if there is a Δ_j-phase, $\Delta_j \in \mathcal{S}(\Delta)$, such that $|M(v, \Delta_j)| \geq deg(v)/\beta$ and $M(v, \Delta_i) \cap M(v, \Delta_j) \neq \emptyset$.

For $\Delta_i \in \mathcal{S}(\Delta)$, let $\mathcal{V}(\Delta_i)$ denote the set of vertices that are hit by the Δ_i-phase.

Lemma 3: For $v \in V$, the number of distinct phases that hit v is $O(\beta \log n)$. Hence, $\sum_{\Delta_i \in \mathcal{S}(\Delta)} |\mathcal{V}(\Delta_i)| = O(n\beta \log n)$. ∎

Define a *push bundle*, b, *incident* with the vertex $\nu(b) \in V$ and *finished* by the $\delta(b)$-phase, $\delta(b) \in \mathcal{S}(\Delta)$, to be a set of push operations over edges incident with $\nu(b)$, such that each of the pushes occurs before or during the $\delta(b)$-phase. A push operation belongs to at most one push bundle.

All push bundles of interest satisfy the following condition:
(B0) For each push bundle b, the sum of the values of the pushes in b is $\geq \delta(b)/4$.

In order to bound $\sharp mediumlinks$, we shall associate a particular kind of push bundle with every link on a medium edge. By Facts 5 and 6, every link on an edge (v, w) is followed by a large push over (v, w). Further, by Fact 4, there are $O(1)$ links on an undirected edge $\{v, w\}$ for a fixed value of $d(v)$. This leads us to consider push bundles that satisfy the following conditions:
(B1) For each push bundle b, all the pushes in b occur during the $\delta(b)$-phase.
(B2) Each push bundle b has an associated triple $(\nu(b), \kappa(b), \mu(b))$, where $\kappa(b) \in \mathbb{N}$, and $\mu(b)$ is an undirected edge incident with $\nu(b)$; one of the pushes in b is over $\mu(b)$ and when this push occurs, $\mu(b)$ is a medium edge and $d(\nu(b)) = \kappa(b)$. Further, the number of push bundles associated with a fixed triple is $O(1)$.

Lemma 4: Over the whole execution, the number of push bundles that satisfy conditions (B0), (B1) and (B2) is $O(nm/\beta + n^2\beta \log n)$.

Proof: We partition the push bundles into 2 classes.
Class 1: The $\delta(b)$-phase hits $\nu(b)$. For $\Delta_i \in \mathcal{S}(\Delta)$, consider the number of push bundles b such that $\delta(b) = \Delta_i$ and b is in this class. This number is $O(|\mathcal{V}(\Delta_i)|n + \mathcal{D}(\Delta_i))$, by Lemma 2, (B0) and (B1), where we take V' in Lemma 2 to be $\mathcal{V}(\Delta_i)$. Summing over all $\Delta_i \in \mathcal{S}(\Delta)$ and applying Lemma 3, there are $O(n^2\beta \log n + n^2) = O(n^2\beta \log n)$ push bundles in this class.
Class 2: The $\delta(b)$-phase does not hit $\nu(b)$. We partition the push bundles in this class into 2 subclasses. For each push bundle b, let $\delta'(b)$ denote the value of Δ when $d(\nu(b))$ is set to $\kappa(b)$ (i.e., the relabeling of $\nu(b)$ that precedes the push over $\mu(b)$ occurs during the $\delta'(b)$-phase).
Class 2.1: $\mu(b)$ is a medium edge during the $\delta'(b)$-phase. For each push bundle b in this subclass, there are $< deg(\nu(b))/\beta$ medium edges incident with $\nu(b)$ when $d(\nu(b))$ is set to $\kappa(b)$ (by (B1), (B2) and the definition of a phase hitting a vertex). Consequently, for a fixed $v \in V$ and a fixed $k \in \mathbb{N}$, this subclass contains $< deg(v)/\beta$ triples of the form $(v, k, \{v, w\})$, $(v, w) \in E$. Summing over all $v \in V$ and all $k \in \mathbb{N}$ and using (B2), there are $O(nm/\beta)$ push bundles in this subclass.
Class 2.2: $\mu(b)$ is a non-medium edge during the $\delta'(b)$-phase. The status of $\mu(b)$ changes between the $\delta'(b)$-phase and the $\delta(b)$-phase, by (B1) and (B2). There are $O(m)$ push bundles in this subclass, by (B2). ∎

Lemma 5: $\sharp mediumlinks = O(nm/\beta + n^2\beta \log n)$. ∎

4.2. Premature-target-relabeling events. A *premature-target-relabeling* event (PTR event) is defined to be the relabeling of the target w of a current edge (v, w). In other words, a PTR event may be viewed as a triple $(w, k, (v, w))$, where $w \in V$, $k \in \mathbb{N}$, and $(v, w) \in E$ is the current edge of $v \in V$ when vertex w, which currently has $d(w) = k$, gets relabeled. Denote by $\sharp ptr$ the total number of PTR events.

The significance of PTR events is that a vertex changes its current edge iff either a saturating push occurs over the edge or a PTR event occurs on the edge. In particular, a vertex whose current edge is a forward huge edge changes its current edge exactly when a PTR event occurs on the edge.

Lemma 6: Over the whole execution,
(1) the number of links on forward huge edges is $\leq n + \sharp ptr$,
(2) the number of saturating pushes over reverse huge edges is $\leq n + m + \sharp ptr$, and
(3) $\sharp hugelinks \leq 2n + m + 2\sharp ptr = O(n + m + \sharp ptr)$.

Proof:
(1) The number of links on forward huge edges is at most n plus the sum over all $v \in V$ of the number of times v changes its current edge while its current edge is a forward huge edge, which is $\leq n + \sharp ptr$.
(2) Between two consecutive saturating pushes over a reverse huge edge, say (w, v), there must be a nonsaturating push, and therefore also a link (recall Fact 6), over the forward huge edge (v, w). Hence, the number of saturating pushes over reverse huge edges is at most m plus the number of links on forward huge edges, which is $\leq m + n + \sharp ptr$.
(3) $\sharp hugelinks$ is at most n plus the number of saturating pushes over reverse huge edges plus the sum over all $v \in V$ of the number of PTR events on edges emanating from v, which is $\leq 2n + m + 2\sharp ptr$. ∎

5. Counting the number of select steps

While bounding $\sharp selects$, we shall come across the following situation: For $v \in V$, while $v \neq find\,root(v)$, $e(v)$ increases from zero to $\geq \Delta$ due to saturating pushes over edges entering v. These saturating pushes may occur during several distinct phases. However, for each of these saturating pushes notice that the value of $d(v)$ is

the same. This leads us to consider push bundles that satisfy one or both of the following conditions:

(B3) For each push bundle b, all the pushes in b are saturating pushes.

(B4) Each push bundle b has an associated number $\kappa(b) \in \mathbb{N}$, such that $d(\nu(b)) = \kappa(b)$ at the start of the $\delta(b)$-phase (recall that a push bundle b is incident with the vertex $\nu(b)$ and is finished by the $\delta(b)$-phase). Further, for each push in b, the push occurs before the start of the $\delta(b)$-phase and $d(\nu(b)) = \kappa(b)$ when the push occurs.

Lemma 7: Over the whole execution,

(1) the number of push bundles that satisfy conditions (B0), (B1) and (B3) is $O(nm/\beta + n^2\beta \log n + \sharp ptr)$, and

(2) the number of push bundles that satisfy conditions (B0), (B3) and (B4) is $O(n + m \log n + nm/\beta + \sharp ptr)$.

Proof: Clearly, there are $O(nm/\beta)$ push bundles such that the number of pushes in a push bundle is $\geq \beta/4$, by Fact 1 and (B3). For each push bundle b, if the number of pushes in b is $< \beta/4$, then note that there is a push in b whose value is $> \delta(b)/\beta$ (by (B0)), and therefore the undirected edge over which this push occurs is non-small during the $\delta(b)$-phase; let $\mu(b)$ denote one such edge, and let the push over $\mu(b)$ occur during the $\delta'(b)$-phase, $\delta'(b) \in \mathcal{S}(\Delta)$. It is easily seen that there are $O(n + m + \sharp ptr)$ push bundles b such that $\mu(b)$ is a huge edge during the $\delta'(b)$-phase, by Lemma 6 and (B3). Focus on the remaining push bundles, and consider separately parts (1) and (2) of the lemma.

(1) For each remaining push bundle b, the number of pushes in b is $< \beta/4$ and $\mu(b)$ is a medium edge during the $\delta'(b)$-phase. Also, note that $\delta'(b) = \delta(b)$, by (B1). Since there are $O(1)$ saturating pushes over $\mu(b)$ for a fixed value of $d(\nu(b))$, it follows that the remaining push bundles satisfy (B2). Therefore the number of remaining push bundles is $O(nm/\beta + n^2\beta \log n)$, by Lemma 4.

(2) We partition the remaining push bundles into 2 classes. For each remaining push bundle b, notice that $\delta'(b) \neq \delta(b)$, by (B4).

Class 1: $\mu(b)$ is a medium edge during both the $\delta'(b)$-phase and the $\delta(b)$-phase. For each push bundle b in this class, notice that $\delta'(b) \geq 2\,\delta(b)$ (by Fact 3). There are $O(m \log n)$ push bundles in this class by (B3), (B4) and the definition of a medium edge.

Class 2: The status of $\mu(b)$ changes between the $\delta'(b)$-phase and the $\delta(b)$-phase. There are $O(m)$ push bundles in this class, by (B3) and (B4). ∎

Lemma 8: $\sharp selects = O(nm/\beta + n^2\beta \log n + \sharp ptr)$.

Proof: We partition the select steps into 2 classes. For each select step z, let $\nu(z)$ denote the vertex selected by the step, and let $\delta(z)$ denote the value of Δ when the step occurs.

Class 1: $\nu(z) \neq$ *find root*$(\nu(z))$. For a vertex $v \in V$, consider the number of times v is selected between a *link* operation on an edge emanating from v and the next *cut*(v) operation. This number is at most two plus twice the number of times $e(v)$ increases from zero to $\geq \Delta$, by Fact 8 (an increase of $e(v)$ from zero to $\geq \Delta$ may be due to pushes that occur during several distinct phases). Consequently, the number of select steps in this class is at most $2\,\sharp links$ plus the sum over all $v \in V$ of the number of times $e(v)$ increases from zero to $\geq \Delta$ while $v \neq$ *find root*(v).

For a vertex $v \in V$, an increase of $e(v)$ from zero to $\geq \Delta$ while $v \neq$ *find root*(v) is due to saturating pushes over edges entering v (by Fact 7). By applying Lemma 7(1) and 7(2), it can be shown that the sum over all $v \in V$ of the number of times $e(v)$ increases from zero to $\geq \Delta$ while $v \neq$ *find root*(v) is $O(nm/\beta + n^2\beta \log n + \sharp ptr)$. Since $\sharp links = O(nm/\beta + n^2\beta \log n + \sharp ptr)$ (by Lemmas 5 and 6), it follows that the number of select steps in this class is $O(nm/\beta + n^2\beta \log n + \sharp ptr)$.

Class 2: $\nu(z) =$ *find root*$(\nu(z))$. One of the following occurs during the $\delta(z)$-phase and between the selection of $\nu(z)$ and the next select step: either a link on an edge emanating from $\nu(z)$ is performed, or $\nu(z)$ is relabeled, or $e(\nu(z))$ decreases from $\geq \delta(z)$ to $< \delta(z)/2$ due to saturating pushes over edges emanating from $\nu(z)$. We partition the select steps in this class into 3 subclasses, according to these 3 cases. The number of select steps in the first subclass is $O(nm/\beta + n^2\beta \log n + \sharp ptr)$, by Lemmas 5 and 6, and the number in the second subclass is $O(n^2)$, by Fact 1. By applying Lemma 7(1) to the third subclass, it can be shown that the number of select steps in this subclass is $O(nm/\beta + n^2\beta \log n + \sharp ptr)$. ∎

6. Decreasing $\sharp ptr$: A game on bipartite graphs

In order to investigate the distribution of $\sharp ptr$, we first study a game on bipartite graphs played between two players, Algy and an adversary. In the following, let $G' = (U', V', E')$ be a fixed undirected bipartite graph and let $N_u = |U'|$, $N_v = |V'|$, $N = N_u + N_v$ and $M = |E'|$. Let $U' = \{u_1, \ldots, u_{N_u}\}$ and $V' = \{v_1, \ldots, v_{N_v}\}$ and for $i = 1, \ldots, N_u$, denote by $\Gamma(u_i)$ the set of neighbours of u_i in G' and by $d_i = |\Gamma(u_i)|$ the degree of u_i in G'. For any finite set A, let Π_A be the set of all permutations of A, i.e., the set of all bijections $\pi : \{1, \ldots, |A|\} \to A$.

The game between Algy and the adversary on G' is played according to the following rules: Algy has a single move, which starts the game and in which he fixes an ordering of the adjacency lists of all vertices in U', i.e., he chooses an element $\xi = (\xi_{u_1}, \ldots, \xi_{u_{N_u}})$ of

$$\Xi = \Pi_{\Gamma(u_1)} \times \cdots \times \Pi_{\Gamma(u_{N_u})}.$$

The remainder of the game is a sequence of moves by the adversary, in which he modifies the graph and attempts to collect as many *game points* as possible. For $j = 0, 1, \ldots$, denote the graph resulting from the jth move by the adversary by $G_j = (U_j, V_j, E_j)$ $(G_0 = G')$. The adversary has three kinds of moves. In the jth move, for $j = 1, 2, \ldots$, he may

(1) Remove a vertex $v \in V_{j-1}$ together with its incident edges, i.e., let $G_j = G_{j-1} - \{v\}$. The adversary obtains for this move a number of game points equal to the number of vertices in G_{j-1} whose first remaining edge (in the ordering implied by ξ) is incident with v, i.e., equal to $|\{u_i \in U_{j-1} \mid \{u_i, v\} \in E_{j-1} \text{ and } \{u_i, \xi_{u_i}(k)\} \notin E_{j-1}, \text{ for all } k \text{ with } 1 \leq k < \xi_{u_i}^{-1}(v)\}|$.

(2) Remove an edge $e \in E_{j-1}$, i.e., let $G_j = G_{j-1} - \{e\}$. The adversary obtains zero game points for this move.

(3) End the game. If $V_{j-1} = \emptyset$, this is the only possible move.

For $\xi \in \Xi$, define the *penalty* of ξ, denoted *penalty*(ξ), to be the maximum number of game points obtainable by the adversary in response to Algy's choice of the move ξ. In order to study the distribution of *penalty*(ξ) when ξ is chosen randomly, we first give another characterization of *penalty*(ξ).

For $1 \leq i \leq N_u$, $\xi = (\xi_1, \ldots, \xi_{N_u}) \in \Xi$ and $\sigma \in \Pi_{V'}$, denote by $\lambda_i(\xi, \sigma)$ the length of a longest (not necessarily contiguous) ascending subsequence of the sequence $\sigma^{-1}(\xi_{u_i}(1)), \ldots, \sigma^{-1}(\xi_{u_i}(d_i))$ (i.e., the sequence given by ordering the elements of $\Gamma(u_i)$ by ξ_{u_i} and numbering them according to σ).

Lemma 9: For all $\xi \in \Xi$, *penalty*$(\xi) = \max_{\sigma \in \Pi_{V'}} \sum_{i=1}^{N_u} \lambda_i(\xi, \sigma)$. ∎

Lemma 10: Suppose that ξ is drawn randomly from the uniform distribution over Ξ. Then for any constant $\alpha > 0$ and for any $\beta = \Omega(\log N)$, *penalty*$(\xi) = O(M/\beta + N\beta)$ with probability at least $1 - N^{-\alpha N}$.

Proof: Fix $\sigma \in \Pi_{V'}$ and let $\Lambda_i = \lambda_i(\xi, \sigma)$, for $i = 1, \ldots, N_u$, and $S = \sum_{i=1}^{N_u} \Lambda_i$. Then for any $k \in \mathbb{N}$,

$$Prob(\Lambda_i \geq k) \leq \frac{\binom{d_i}{k}^2 (d_i - k)!}{d_i!} \leq \frac{d_i^k}{(k!)^2} \leq \left(\frac{e^2 d_i}{k^2}\right)^k,$$

and for any $a_i \in \mathbb{N}$,

$$E(e^{\Lambda_i}) = \sum_{k=0}^{\infty} e^k Prob(\Lambda_i = k)$$

$$\leq e^{a_i} + \sum_{k=a_i+1}^{\infty} e^k Prob(\Lambda_i \geq k) \leq e^{a_i} + \sum_{k=a_i+1}^{\infty} \left(\frac{e^3 d_i}{k^2}\right)^k.$$

Choose $a_i = \lfloor \sqrt{2e^3 d_i} \rfloor$. Then

$$E(e^{\Lambda_i}) \leq e^{a_i} + \sum_{k=a_i+1}^{\infty} 2^{-k} = e^{a_i} + 2^{-a_i} \leq 2e^{\sqrt{2e^3 d_i}}.$$

For any $r \in \mathbb{R}$,

$$Prob(S \geq r) = e^{-r} e^r Prob(e^S \geq e^r) \leq e^{-r} E(e^S).$$

Since $\Lambda_1, \ldots, \Lambda_{N_u}$ are independent, $E(e^S) = \prod_{i=1}^{N_u} E(e^{\Lambda_i})$. Consequently, for any $r \in \mathbb{R}$,

$$Prob(S \geq r) \leq e^{-r} \prod_{i=1}^{N_u} E(e^{\Lambda_i}) \leq e^{-r} 2^{N_u} e^{\sqrt{2e^3} \sum_{i=1}^{N_u} \sqrt{d_i}}.$$

Since $\sqrt{d_i} \geq \beta \Rightarrow \sqrt{d_i} \leq d_i/\beta$, clearly

$$\sum_{i=1}^{N_u} \sqrt{d_i} \leq M/\beta + N\beta.$$

Let $r = N_u + \sqrt{2e^3}(M/\beta + N\beta) + (\alpha+1)N \log N = O(M/\beta + N\beta)$. We find that

$$Prob(S \geq r) \leq e^{-(\alpha+1)N \log N} = N^{-(\alpha+1)N}.$$

Since the above analysis holds for all $\sigma \in \Pi_{V'}$, we may conclude that $penalty(\xi) \geq r$ with probability at most $|\Pi_{V'}| \cdot N^{-(\alpha+1)N} \leq N^{-\alpha N}$. ∎

Lemma 11: For any constant $\alpha > 0$ and for any $\beta = \Omega(\log n)$, $\sharp ptr = O(nm/\beta + n^2\beta)$ with probability at least $1 - n^{-\alpha n^2}$.

Proof: For a network digraph $G = (V, E)$, let the corresponding game graph be $G' = (U', V', E')$, where

$$U' = V' = \{\langle v, k \rangle \mid v \in V \text{ and } 0 \leq k \leq 2n-1\}, \text{ and}$$

$$E' = \{\{\langle v, k \rangle, \langle w, k-1 \rangle\} \mid (v, w) \in E, \langle v, k \rangle \in U', 1 \leq k \leq 2n-1\}.$$

Note that $N = |U'| + |V'| = \Theta(n^2)$ and $M = |E'| = \Theta(nm)$.

Given an execution of the PLED algorithm on G, we now construct a play of the game on G' such that the following holds:

Claim: $\sharp ptr$ for the execution at most equals the number of game points obtained by the adversary in the play.

(1) Algy chooses a move ξ such that for all $v \in V$ and all k, $0 \leq k \leq 2n-1$,
   (a) If $d(v) = k$ at some step of the execution, then $\xi_{\langle v,k \rangle}$ (i.e., the ordering of the adjacency list of $\langle v, k \rangle$ implied by ξ) is chosen to agree with the ordering of the adjacency list of v fixed by the PLED algorithm when it sets $d(v) = k$ (either initially or while relabeling v).
   (b) If $d(v) = k$ does not occur during the execution, then $\xi_{\langle v,k \rangle}$ is drawn randomly from the uniform distribution over $\Pi_{\Gamma(\langle v,k \rangle)}$.

It is easy to see that ξ is drawn randomly from the uniform distribution over Ξ.

The move sequence by the adversary is constructed such that the remainder of the play simulates the execution of the PLED algorithm. For ease of description, pretend that the execution of the algorithm and the play take place concurrently.

(2) For a network vertex $v \in V$, when the PLED algorithm sets $d(v) = k$, the adversary removes all game edges $\{\langle v, k \rangle, \langle w, k-1 \rangle\}$ such that the network edge (v, w) is not currently eligible.
(3) When the PLED algorithm relabels a network vertex $v \in V$, then the adversary removes the game vertex $\langle v, k \rangle \in V'$, where $d(v) = k$ prior to the relabeling operation.
(4) When the PLED algorithm performs a saturating push over a network edge $(v, w) \in E$, then the adversary removes the game edge $\{\langle v, k \rangle, \langle w, k-1 \rangle\}$, where $k = d(v)$.

It can be seen that the claim holds. Hence, $\sharp ptr \leq penalty(\xi)$. The desired result now follows from Lemma 10, which is applicable since ξ is chosen randomly. ∎

By Lemmas 8 and 11, $\sharp selects = O(nm/\beta + n^2\beta \log n)$ with high probability. The main result follows from Lemma 1 by fixing $\beta = 2 + \log n$.

Theorem 2: For any constant $\alpha > 0$, the PLED algorithm finds a maximum flow in time $O(nm + n^2(\log n)^3)$ with probability at least $1 - n^{-\alpha n^2}$. The worst-case running time is $O(nm \log n)$. ∎

Acknowledgment: It is a pleasure to thank Kurt Mehlhorn for his interest in and enthusiasm for this work.

References

[AO87]: R. K. Ahuja and J. B. Orlin: "A Fast and Simple Algorithm for the Maximum Flow Problem". Sloan Working Paper No. 1905–87, MIT, 1987. Also to appear in *Operations Research*.

[FT87]: M. L. Fredman and R. E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". *Journal of the ACM* **34** (1987), 596–615.

[GT88]: Andrew V. Goldberg and Robert E. Tarjan: "A New Approach to the Maximum-Flow Problem". *Journal of the ACM* **35** (1988), 921–940.

[ST83]: D. D. Sleator and R. E. Tarjan: "A Data Structure for Dynamic Trees". *Journal of Computer and System Sciences* **26** (1983), 362–391.

[ST85]: D. D. Sleator and R. E. Tarjan: "Self-Adjusting Binary Search Trees". *Journal of the ACM* **32** (1985), 652–686.