Maximum Flows by Incremental Breadth-First Search

Andrew V. Goldberg[1], Sagi Hed[2], Haim Kaplan[2],
Robert E. Tarjan[3], and Renato F. Werneck[1]

[1] Microsoft Research Silicon Valley
{goldberg,renatow}@microsoft.com
[2] Tel Aviv University
{sagihed,haimk}@tau.ac.il
[3] Princeton University and HP Labs
ret@cs.princeton.edu

Abstract. Maximum flow and minimum s-t cut algorithms are used to solve several fundamental problems in computer vision. These problems have special structure, and standard techniques perform worse than the special-purpose *Boykov-Kolmogorov* (BK) algorithm. We introduce the *incremental breadth-first search* (IBFS) method, which uses ideas from BK but augments on shortest paths. IBFS is theoretically justified (runs in polynomial time) and usually outperforms BK on vision problems.

1 Introduction

Computing maximum flows is a classical optimization problem that often finds applications in new areas. In particular, the minimum cut problem (the dual of maximum flows) is now an important tool in the field of computer vision, where it has been used for segmentation, stereo images, and multiview reconstruction. Input graphs in these applications typically correspond to images and have special structure, with most vertices (representing pixels or voxels) arranged in a regular 2D or 3D grid. The source and sink are special vertices connected to all the others with varying capacities. See [2, 3] for surveys of these applications.

Boykov and Kolmogorov [2] developed a new algorithm that is superior in practice to general-purpose methods on many vision instances. Although it has been extensively used by the vision community, it has no known polynomial-time bound. No exponential-time examples are known either, but the algorithm performs poorly in practice on some non-vision problems.

The lack of a polynomial time bound is disappointing because the maximum flow problem has been extensively studied from the theoretical point of view and is one of the better understood combinatorial optimization problems. Known solutions to this problem include the augmenting path [9], network simplex [7], blocking flow [8, 15] and push-relabel [12] methods. A sequence of increasingly better time bounds has been obtained, with the best bounds given in [16, 11].

Experimental work on the maximum flow problem has a long history and includes implementations of algorithms based on blocking flows (e.g., [5, 13]) and on the push-relabel method (e.g., [6, 10, 4]), which is the best general-purpose

approach in practice. With the extensive research in the area and its use in computer vision, the *Boykov-Kolmogorov* (BK) algorithm is an interesting development from a practical point of view.

In this paper we develop an algorithm that combines ideas from BK with those from the shortest augmenting path algorithms. In fact, our algorithm is closely related to the blocking flow method. However, we build the auxiliary network for computing augmenting paths in an incremental manner, by updating the existing network after each augmentation while doing as little work as we can. Since for the blocking flow method network construction is the bottleneck in practice, this leads to better performance. Like BK, and unlike most other current algorithms, we build the network in a bidirectional manner, which also improves practical performance.

We call the resulting algorithm *Incremental Breadth First Search* (IBFS). It is theoretically justified in the sense that it gets good (although not the best) theoretical time bounds. Our experiments show that IBFS is faster than BK on most vision instances. Like BK, the algorithm does not perform as well as state-of-the-art codes on some non-vision instances. Even is such cases, however, IBFS appears to be more robust than BK. BK is heavily used to solve vision problems in practice. IBFS offers a faster and theoretically justified alternative.

2 Definitions and Notation

The input to the maximum flow problem is (G, s, t, u), where $G = (V, A)$ is a directed graph, $s \in V$ is the *source*, $t \in V$ is the *sink* (with $s \neq t$), and $u : A \Rightarrow [1, \ldots, U]$ is the *capacity function*. Let $n = |V|$ and $m = |A|$.

Let a^R denote the *reverse* of an arc a, let A^R be the set of all reverse arcs, and let $A' = A \cup A^R$. A function g on A' is *anti-symmetric* if $g(a) = -g(a^R)$. Extend u to be an anti-symmetric function on A', i.e., $u(a^R) = -u(a)$.

A flow f is an anti-symmetric function on A' that satisfies *capacity constraints* on all arcs and *conservation constraints* at all vertices except s and t. The capacity constraint for $a \in A$ is $0 \leq f(a) \leq u(a)$ and for $a \in A^R$ it is $-u(a^R) \leq f(a) \leq 0$. The conservation constraint for v is $\sum_{(u,v)\in A} f(u, v) = \sum_{(v,w)\in A} f(v, w)$. The *flow value* is the total flow into the sink: $|f| = \sum_{(v,t)\in A} f(v, t)$. A *cut* is a partitioning of vertices $S \cup T = V$ with $s \in S, t \in T$. The capacity of a cut is defined by $u(S, T) = \sum_{v\in S,w\in T,(v,w)\in A} u(v, w)$. The max-flow/min-cut theorem [9] says that the maximum flow value is equal to the minimum cut capacity.

The *residual capacity* of an arc $a \in A'$ is defined by $u_f(a) = u(a) - f(a)$. Note that if f satisfies capacity constraints, then u_f is nonnegative. The *residual graph* $G_f = (V, A_f)$ is the graph induced by the arcs in A' with strictly positive residual capacity. An *augmenting path* is an s–t path in G_f.

When we talk about distances (and shortest paths), we mean the distance in the residual graph for the unit length function. A *distance labeling from s* is an integral function d_s on V that satisfies $d_s(s) = 0$. Given a flow f, we say that d_s is valid if for all $(v, w) \in A_f$ we have $d_s(w) \leq d_s(v) + 1$. A (valid) distance labeling to t, d_t, is defined symmetrically. We say that an arc (v, w) is

admissible w.r.t. d_s if $(v, w) \in A_f$ and $d_s(v) = d_s(w) - 1$, and *admissible w.r.t.* d_t if $(v, w) \in A_f$ and $d_t(w) = d_t(v) - 1$.

3 BK Algorithm

In this section we briefly review the BK algorithm [2]. It is based on augmenting paths. It maintains two trees of residual arcs, S rooted from s and T rooted into t. Initially S contains only s and T contains only t. At each step, a vertex is in S, in T, or *free*. Each tree has *active* and *internal* vertices. The outer loop of the algorithm consists of three stages: *growth*, *augmentation*, and *adoption*.

The growth stage expands the trees by scanning their active vertices and adding newly-discovered vertices to the tree from which they have been discovered. The newly-added vertices become active. Vertices become internal after being scanned. If no active vertices remain, the algorithm terminates. If a residual arc from S to T is discovered, then the augmentation stage starts.

The augmentation stage takes the path found by the growth stage and augments the flow on it by its bottleneck residual capacity. Some tree arcs become saturated, and their endpoints farthest from the corresponding root become *orphans*. If an arc (v, w) becomes saturated and both v and w are in S, then w becomes an S-orphan. If both v and w are in T, v becomes a T-orphan. If v is in S and w is in T, then a saturation of (v, w) does not create orphans. Orphans are placed on a list and processed in the adoption stage.

The adoption stage processes orphans until there are none left. Consider an S-orphan v (T-orphans are processed similarly). We examine residual arcs (u, v) in an attempt to find a vertex u in S. If we find such u, we check whether the tree path from u to s is valid (it may not be if it contains an orphan, including u). If a vertex u with a valid path is found, we make u the parent of v.

If we fail to find a new parent for v, we make v a free vertex and make all children of v orphans. Then we examine all residual arcs (u, v) and for each u in S, we make u active. Note that for each such u, the tree path from s to u contains an orphan (otherwise u would have been picked as v's parent) and this orphan may find a new parent. Making u active ensures that we find v again.

The only known way to analyze BK is as a generic augmenting path algorithm, which does not give polynomial bounds.

4 Incremental Breadth-First Search

The main idea IBFS is to modify BK to maintain breadth-first search trees, which leads to a polynomial time bound ($O(n^2m)$). Existing techniques can improve this further, matching the best known bounds for blocking flow algorithms.

The algorithm maintains distance labels d_s and d_t for every vertex. The two trees, S and T, satisfy the *tree invariants*: for some values D_s and D_t, the trees contain all vertices at distances up to D_s from s and up to D_t to t, respectively. We also maintain the invariant that $L = D_s + D_t + 1$ is a lower bound on the augmenting path length, so the trees are disjoint.

A vertex can be an S-vertex, T-vertex, S-orphan, T-orphan, or N-vertex (not in any tree). Each vertex maintains a parent pointer p, which is *null* for N-vertices and orphans. We maintain the invariant that tree arcs are admissible. During the adoption step, the trees are rebuilt and are not well-defined. Some invariants are violated and some orphans may leave the trees. We say that a vertex is in S if it is an S-vertex or an S-orphan. In a growth step, there are no orphans, so S is the set of S-vertices. Similarly for T.

If a vertex v is in S, $d_s(v)$ is the meaningful label value and $d_t(v)$ is unused. The situation is symmetric for vertices in T. Labels of N-vertices are irrelevant. Since at most one of $d_s(v)$ and $d_t(v)$ is used at any given time, one can use a single variable to represent both labels.

Initially, S contains only s, T contains only t, $d_s(s) = d_t(t) = 0$, and all parent pointers are *null*. The algorithm proceeds in passes. At the beginning of a pass, all vertices in S are S-vertices, all vertices in T are T-vertices, and other vertices are N-vertices. The algorithm chooses a tree to grow in the pass, either S (*forward*) or T (*reverse*). Assume we have a forward pass; the other case is symmetric. The goal of a pass is to grow S by one level and to increase D_s (and L) by one. We make all vertices v of S with $d_s(v) = D_s$ *active*. The pass executes growth steps, which may be interrupted by augmentation steps (when an augmenting path is found) followed by adoption steps (to fix the invariants violated when some arcs get saturated). At the end of the pass, if S has any vertices at level $D_s + 1$, we increment D_s; otherwise we terminate.

For efficiency, we use the *current arc* data structure, which ensures that each arc into a vertex is scanned at most once between its distance label increases during the adoption step. When an N-vertex is added to the tree or when the distance label of a vertex changes, we set the current arc to the first arc in its adjacency list. We maintain the invariant that the arcs preceding the current arc on the adjacency list are not admissible.

The growth step picks an active vertex v and scans v by examining residual arcs (v, w). If w is an S-vertex, we do nothing. If w is an N-vertex, we make w an S-vertex, set $p(w) = v$, and set $d_s(w) = D_s + 1$. If w is in T, we perform an augmentation step as described below. Once all arcs out of v are scanned, v becomes inactive. If a scan of v is interrupted by an augmentation step, we remember the outgoing arc that triggered it. If v is still active after the augmentation, we resume the scan of v from that arc.

The augmentation step applies when we find a residual arc (v, w) with v in S and w in T. The path P obtained by concatenating the s–v path in S, the arc (v, w), and the w–t path in T is an augmenting path. We augment on P, saturating some of its arcs. Saturating an arc $(x, y) \neq (v, w)$ creates orphans. Note that x and y are in the same tree. If they are in S, we make y an S-orphan, otherwise we make x a T-orphan. At the end of the augmentation step, we have (possibly empty) sets O_s and O_t of S- and T-orphans, respectively. These sets are processed during the adoption step.

We describe the adoption step assuming we grow S (the case for T is symmetric). S has a partially completed level $D_s + 1$. To avoid rescanning vertices at level D_s, we allow adding vertices to this level during orphan processing.

Our implementation of the adoption step is based on the *relabel* operation of the push-relabel algorithm. To process an S-orphan v, we first scan the arc list starting from the current arc and stop as soon as we find a residual arc (u, v) with $d_s(u) = d_s(v) - 1$. If such a vertex u is found, we make v an S-vertex, set the current arc of v to (v, u), and set $p(v) = u$. If no such u is found, we apply the *orphan relabel* operation to v. The operation scans the whole list to find the vertex u for which $d_s(u)$ is minimum and (u, v) is residual. If no such u exists, or if $d_s(u) > D_s$, we make v an N-vertex and make vertices w such that $p(w) = v$ S-orphans. Otherwise we choose u to be the first such vertex and set the current arc of v to be (v, u), set $p(v) = u$, set $d_s(v) = d_s(u) + 1$, make v an S-vertex, and make vertices w such that $p(w) = v$ S-orphans. If v was active and now $d_s(v) = D_s + 1$, we make v inactive.

The adoption step for T-vertices is symmetric except we make v an N-vertex if $d_t(u) \geq D_t$ (not just $d_t(u) > D_t$) because we are in the forward pass. Once both adoption steps finish, we continue the growth step.

4.1 Correctness and Running Time

We now prove that IBFS is correct and bound its running time. When analyzing individual passes, we assume we are in a forward pass; the reverse pass is similar.

We start the analysis by considering what happens on tree boundaries.

Lemma 1. *If (u, v) is residual:*

1. *If $u \in S$, $d_s(u) \leq D_s$, and $v \notin S$, then u is an active S-vertex.*
2. *If $v \in T$ and $u \notin T$, then $d_t(v) = D_t$.*
3. *After the increase of D_s, if $u \in S$ and $v \notin S$, then $d_s(u) = D_s$.*

Proof. The proof is by induction on the number of growth, augmentation, and adoption steps and passes.

At the beginning of a pass, all S-vertices u with $d_s(u) = D_s$ are active. Moreover, (2) and (3) hold at the end of the previous pass (or after the initialization for the first pass). This implies (1) and (2).

A growth step on u without an augmentation makes u inactive, but only after completing a scan of arcs (u, v) and adding all vertices $v \neq t$ with a residual arc (u, v) to S, so (1) is maintained. A growth step does not change T, so it cannot affect the validity of (2).

An augmentation can make an arc (u, v) non-residual, which cannot cause any claim to be violated. An augmentation can create a new residual arc (u, v) with $u \in S$, if flow is pushed along (v, u). In this case $v = p(u)$, so v must also be in S and (1) does not apply for (u, v). The symmetric argument shows that (2) does not apply for a new residual arc either.

An orphan relabel step can remove a vertex v from S. However, if a residual arc (u, v) exists with $u \in S$ and $d_s(u) \leq D_s$, then by definition of the orphan relabel step, v remains an S-vertex. So (1) is maintained after an orphan relabel step. The symmetric argument shows that (2) is maintained as well.

Finally, if there are no active vertices, then (u, v) can be a residual arc with $u \in S$ and $v \notin S$ only if $d_s(u) > D_s$. Since we grow the tree by one level, $d_s(u) = D_s + 1$. This implies that (3) holds after the increase of D_s. □

We now consider the invariants maintained by the algorithm.

Lemma 2. *The following invariants hold:*

1. *Vertices in S and T have valid labelings, d_s and d_t.*
2. *For every vertex u in S, u's current arc either precedes or is equal to the first admissible arc to u. For every vertex u in T, u's current arc either precedes or is equal to the first admissible arc from u.*
3. *If u is an S-vertex, then $(p(u), u)$ is admissible. If u is a T-vertex, then $(u, p(u))$ is admissible.*
4. *For every vertex v, $d_s(v)$ and $d_t(v)$ never decrease.*

Proof. The proof is by induction on the growth, augmentation and adoption steps. We prove the claim for S; the proof for T is symmetric.

Augmentations do not change labels and therefore (4) does not apply. An augmentation can create a new residual arc $(u, p(u))$ by pushing flow on $(p(u), u)$. Using the induction assumption of (3), however, $(p(u), u)$ is admissible, so $(u, p(u))$ cannot be admissible and thus (2) still applies. In addition, $d_s(p(u)) = d_s(u) - 1$, so (1) is maintained. An augmentation can make an arc $(p(u), u)$ non-admissible by saturating it. However, this cannot violate claims (1) or (2) and vertex u becomes an orphan, so (3) is not applicable.

Consider a growth step on u that adds a new vertex v to S. We set $d_s(v) = d_s(u) + 1 = D_s + 1$, so (3) holds. For every residual arc (w, v) with $w \in S$, w must be active by Lemma 1. Since the d_s value of every active vertex is D_s, we get $d_s(w) = D_s = d_s(v) - 1$, so (1) holds. The current arc of v is v's first arc, so (2) holds. Since v is added at the highest possible label, it is clear that the label of v did not decrease and (4) is maintained.

Consider an adoption step on v. The initial scan of the orphan's arc list does not change labels and therefore cannot break (1) or (4). An orphan scan starts from the current arc, which precedes the first admissible arc by the induction assumption of (2), therefore it will find the first admissible arc to v. So if v finds a new parent, the new current arc is the first admissible arc to v, as required by (2) and (3). An orphan relabel finds the first lowest label $d_s(u)$ such that (u, v) is residual. So the labeling remains valid and the current arc is the first admissible arc, as required by (1), (2) and (3). Using the induction assumption of (1), labeling validity ensures that an orphan relabel cannot decrease the label of a vertex, by definition, so (4) is maintained. □

At the end of the forward step there are no active vertices, so if the level $D_s + 1$ of S is empty, then by Lemma 1 there are no residual arcs from a vertex in S to a vertex not in S, and therefore the current flow is a maximum flow.

The following two lemmas are useful to provide some intuition on the algorithm. They are not needed for the analysis, so we state them without proofs.

Lemma 3. *During a growth phase, for every vertex $v \in S$, the path in S from s to v is a shortest path, and for every vertex $v \in T$, the path in T from v to t is a shortest path.*

Lemma 4. *The algorithm maintains the invariant that $L = D_s + D_t + 1$ is a lower bound on the augmenting path length, and always augments along the shortest augmenting path.*

The next lemma allows us to charge the time spent on orphan arc scans.

Lemma 5. *After an orphan relabel on v in S, $d_s(v)$ increases. After an orphan relabel on v in T, $d_t(v)$ increases.*

Proof. Consider an orphan relabel on an orphan $v \in S$. The analysis for an orphan $v \in T$ is symmetric.

Let U be the set of vertices u such that $u \in S$ and (u, v) is residual during the orphan relabel. By Lemma 2, v's current arc precedes the first admissible arc to v. Since during the orphan scan we did not find any admissible arc after v's current arc, there are no admissible arcs to v. By Lemma 2, the labeling is valid, so $d_s(u) \geq d_s(v) - 1$ for every $u \in U$. Since no admissible arc to v exists, we have that $d_s(u) \geq d_s(v)$ for every $u \in U$. So if the relabel operation does not remove v from S, it will increase $d_s(v)$.

Assume the relabel operation removes v from S. Let $d'_s(v)$ be the value of $d_s(v)$ when v was removed from S. Vertex v might be added to S later, during a growth step on some vertex $w \in S$. If $w \in U$, then $d_s(w)$ did not decrease since the relabel on v (by Lemma 2), so v will be added to S with a higher label. If $w \notin U$ then (w, v) became residual after v was removed from S. This means flow was pushed along (v, w) with $v \notin S$. This is only possible with $w \notin S$. So w was at some point removed from S and then added back to S at label $D_s + 1 \geq d'_s(v)$. Using Lemma 2, $d_s(w)$ did not decrease since that time, so when v is added to S, we get $d_s(v) = d_s(w) + 1 \geq d'_s(v) + 1$. □

We are now ready to bound the running time of the algorithm.

Lemma 6. *IBFS runs in $O(n^2m)$ time.*

Proof. There are three types of operations we must account for: adoption steps, growth steps with augmentations, and growth steps without augmentations.

Consider a growth step on v without an augmentation. We charge a scan of a single arc during the step to the label of v. Since we do not perform augmentations, v becomes inactive once the scan of its arcs is done. Vertex v can become active again only when its label increases. Thus every arc (v, u) scanned during such a growth step charges the distance label at most once. There are at most $n-1$ different label values for each side (S or T), so the total time spent scanning arcs in growth steps without augmentations is $O(\Sigma_v \text{ degree}(v) \cdot (n-1)) = O(nm)$.

We charge a scan of a single arc during an adoption step on v to the label of v. By Lemma 5 and Claim (4) of Lemma 2, after every orphan relabel $d_s(v)$ or $d_t(v)$ increases and cannot decrease afterwards. So every arc charges each label at most twice, once in an orphan scan and once in an orphan relabel. Since there are $O(n)$ labels, the time spent scanning arcs in adoption steps is also $O(nm)$.

We divide the work of a growth step with an augmentation on v into scanning arcs of v to find the arc to T and performing the augmentation. For the former, since we remember the arc used in the last augmentation, an arc of v

not participating in an augmentation is scanned only once per activation of v. An analysis similar to that for the growth steps without augmentation gives an $O(nm)$ bound on such work for the whole algorithm. For the latter, the work per augmentation is $O(n)$. If the saturated arc (u, v) is in S or T, the work can be charged to the previous scan of the arc after which it was added to the tree. It remains to account for augmentations that saturate an arc (u, v) with $u \in S$ and $v \in T$. We charge every such saturation to the label $d_s(u)$. While u remains active, (u, v) cannot be saturated again. As with growth steps without augmentations, u can only become active again when its label increases. So a saturation of (u, v) charges the label $d_s(u)$ at most once. There are at most $n - 1$ distinct label values, so the total number of such charges is $O(nm)$. An augmentation during a growth of v, including the scan of v's arcs until the augmentation, takes $O(n)$ time. So the total time spent on growth steps with augmentations is $O(n^2m)$. □

This bound can be improved to $O(nm \log n)$ using the dynamic trees data structure [17], but in practice the simple $O(n^2m)$ version is faster on vision instances.

5 Variants of IBFS

We briefly discuss two variants of IBFS, incorporating blocking flows and delays. According to our preliminary experiments, these variants have higher constant factors and are somewhat slower than the standard algorithm on vision instances, which are relatively simple. These algorithms are interesting from a theoretical viewpoint, however, and are worth further experimental evaluation as well.

A blocking flow version. Note that at the beginning of a growth pass, we have an auxiliary network on which we can compute a blocking flow (see e.g. [15]). The network is induced by the arcs (v, w) such that either both v and w are in the same tree and the arc is admissible, or v is in S and w is in T and (v, w) is residual. We get a blocking flow algorithm by delaying vertex relabelings: a vertex whose parent arc becomes saturated, or whose parent becomes an orphan, tries to reconnect at the same level of the same tree and becomes an orphan if it fails. In this case its distance from s (if it is an S-orphan) or from t (T-orphan) increased. We process orphans at the end of the growth/augment pass.

It may be possible to match the bound on the number of iterations of the binary blocking flow algorithm bound [11].

A delayed version. The standard version of IBFS ignores some potentially useful information. For example, suppose that $D_s = D_t = 10$, $L = 21$, and for an S-vertex v, $d_s(v) = 2$. Then a lower bound on the distance from v to t is $21-2 = 19$. Suppose that, after an augmentation and an adoption step, v remains an S-vertex but $d_s(v) = 5$. Because distances to t are monotone, 19 is still a valid lower bound, and we can delay the processing of v until L increases to $5 + 19 = 24$.

The *delayed IBFS* algorithm takes advantage of such lower bounds to delay processing vertices known not to be on shortest paths of length L. Furthermore,

the algorithm is lazy: it does not scan delayed vertices. As a result, vertices reachable only through delayed vertices (not "touching" tree vertices) are implicitly delayed as well. Compared to standard IBFS, the delayed variant is more complicated, and so is its analysis: it maintains a lot of information implicitly, and more state transitions can occur.

6 Experimental Results

6.1 Implementation Details

We now give details of our implementation of IBFS, which we call *IB*. Instead of performing a forward or reverse pass independently, we grow both trees by one level simultaneously. This may result in augmenting paths one arc longer than shortest augmenting paths: for example, during the growth step of an S-vertex v with label D_s we may find a T-vertex w with label $D_t + 1$. Since the s–v path in S and the w–t path in T are shortest paths, one can still show that the distances are monotone and the analysis remains valid. Note that BK runs in the same manner, growing both trees simultaneously.

We process orphans in FIFO order. If an augmentation saturates a single arc (which is quite common), FIFO order means that all subsequent orphans (in the original orphan's subtree) will be processed in ascending order of labels.

We maintain *current arcs* implicitly. The invariants of IBFS ensure the current arc of v is either its parent arc or the first arc in its adjacency list. A single bit suffices to distinguish between these cases.

For each vertex v in a tree, we keep its children in a linked list, allowing them to be easily added to the list of orphans when v is relabeled.

During an orphan relabel step on a vertex v in S, if a potential parent u is found with $d_s(u) = d_s(v)$, then the scan halts and u is taken as the parent. It is easy to see that such a vertex u must have the minimum possible label. A similar rule is applied to vertices in T.

On vision instances, orphan relabels often result in increasing the label of the orphan by one. To make this case more efficient, we use the following heuristic. When an orphan v is relabeled, its children become orphans. For every child u of v, we make (v, u) the first arc in u's adjacency list. If v's label does increase by one, a subsequent orphan relabel step on u will find (u, v) immediately and halt (due to the previous heuristic), saving a complete scan of u's arc list.

We also make some low-level optimizations for improved cache efficiency. Every arc (u, v) maintains a bit stating whether the residual capacity of (v, u) is zero. This saves an extra memory access to the reverse arc during growth steps in T and during orphan steps in S. The bit is updated during augmentations, when residual capacities change. Moreover, we maintain the adjacency list of a vertex v in an array. To make the comparison fair, we make these low-level optimizations to BK as well. We compared our improved code, *UBK*, to BK version 3.0.1 from *http://www.cs.ucl.ac.uk/staff/V.Kolmogorov/software.html*. Overall, UBK is about 20% faster than the original BK implementation, although the speedup is not uniform and BK is slightly faster on some instances.

All implementations (BK, UBK, and IB) actually eliminate the source and target vertices (and their incident arcs) during a preprocessing step. For each vertex v, they perform a trivial augmentation along the path $(s, v) \cdot (v, t)$ and assign either a demand or an excess to v, depending on whether (s, v) or (v, t) is saturated. The running times we report to not include preprocessing.

6.2 Experiments

We ran our experiments on a 32-bit Windows 7 machine with 4 GB of RAM and a 2.13 GHz Intel Core i3-330M processor (64 KB L1, 256 KB L2, and 3 MB L3 cache). We used the Microsoft Visual C++ 6.0 compiler with default "Release" optimization settings. We report system times (obtained with the *ftime* function) of the maximum flow computation, which excludes the time to read and initialize the graph. For all problems, capacities are integral.

Table 1 has the results. For each instance, we give the number of vertices, n, and density, m/n. We then report the running times (in seconds) of IB and UBK, together with the relative speedup (SPD), i.e., the ratio between them. Values greater than 1.0 favor IB. The remaining columns contain some useful operation counts. PU is the combined length of all augmenting paths. GS is the number of arc scans during growth steps. OS is the number of arc scans during orphan steps. Finally, OT is the number of arcs scanned by UBK when traversing the paths from a potential parent to the root of its tree (these are not included in OS). Note that all counts are *per vertex* (i.e, they are normalized by n).

The instances in the table are split into six blocks. Each represents a different family: image segmentation using scribbles, image segmentation, surface fitting, multiview reconstruction, stereo images, and a hard DIMACS family.

The first five families are vision instances. The scribble instances were created by the authors, and are available upon request. The four remaining vision families are available at *http://vision.csd.uwo.ca/maxflow-data/*, together with detailed descriptions. (Other instances from these families are available as well; we took a representative sample due to space constraints.) Note that each image segmentation instance has two versions, with maximum capacity 10 or 100. For the vision problems, the running times are the average of three runs for every instance. Because stereo image instances are solved extremely fast, we present the total time for solving all instances of each subfamily.

Note that IB is faster than BK on all vision instances, except `bone10` and `bone100`. The speedup achieved by IB is usually modest, but can be close to an order of magnitude in some cases (such as `gargoyle`). IB is also more robust. It has similar performance on `gargoyle` and `camel`, which are problems from the same application and of similar size; in contrast, UBK is much slower on `gargoyle` than on `camel`.

Operation counts show that augmenting on shortest paths leads to fewer arc flow changes and growth steps, but to more orphan processing. This is because IB has more restrictions on how a disconnected vertex can be reconnected. UBK also performs OT operations, which are numerous on some instances (e.g., `gargoyle`).

Table 1. Performance of IBFS and BK on various instances.

INSTANCE			TIME [s]			PU		GS		OS		OT
NAME	n	$\frac{m}{n}$	IB	UBK	SPD	IB	UBK	IB	UBK	IB	UBK	UBK
diggedshweng	301035	5.0	0.42	1.26	3.00	16.9	160.0	6.7	7.7	87.8	7.7	38.4
hessi1a	494402	5.0	5.81	6.43	1.11	108.4	353.2	7.3	25.4	601.7	43.9	126.5
monalisa	789419	5.0	2.92	4.33	1.48	30.9	181.9	8.1	11.7	239.4	17.1	59.2
house	967874	5.0	2.54	3.16	1.24	33.0	122.2	6.3	10.2	129.6	13.3	43.7
anthra	1061920	5.0	6.28	6.73	1.07	53.5	153.0	6.8	17.3	348.3	27.3	83.3
bone_subx10	3899394	7.0	2.73	3.20	1.17	0.6	1.3	6.6	8.2	25.0	5.5	11.7
bone_subx100	3899394	7.0	3.30	5.32	1.61	2.8	10.9	6.8	8.8	30.1	6.8	23.0
liver10	4161602	7.0	4.91	5.98	1.22	1.0	2.1	6.5	9.6	45.6	8.7	22.2
liver100	4161602	7.0	6.62	14.21	2.15	7.5	23.2	6.9	12.3	56.0	13.6	66.5
babyface10	5062502	7.0	4.98	5.72	1.15	0.5	1.0	6.4	9.3	38.6	7.0	15.4
babyface100	5062502	7.0	6.44	11.33	1.76	4.5	12.7	6.6	10.7	46.3	9.5	39.5
bone10	7798786	7.0	6.24	4.21	0.67	0.1	0.1	6.9	7.5	30.7	3.6	3.6
bone100	7798786	7.0	7.01	5.56	0.79	0.5	2.0	6.9	8.1	35.6	5.1	7.0
bunny-med	6311088	7.0	1.04	1.28	1.23	0.3	0.5	6.2	6.2	0.6	0.4	0.6
gargoyle-sml	1105922	5.0	0.89	8.56	9.57	7.8	212.8	7.5	6.8	33.5	10.7	143.2
gargoyle-med	8847362	5.0	22.58	139.06	6.16	22.7	337.2	8.7	12.1	121.6	20.7	250.5
camel-sml	1209602	5.0	0.84	1.31	1.56	5.3	27.6	6.6	6.8	27.5	8.0	23.1
camel-med	9676802	5.0	21.00	32.33	1.54	20.4	74.0	6.8	9.4	92.4	13.0	61.2
BVZ-tsukuba	—	—	0.42	0.45	1.09	1.2	1.7	5.1	5.5	10.8	3.9	2.8
BVZ-sawtooth	—	—	0.70	0.84	1.20	1.6	2.5	5.1	5.5	6.1	3.7	2.7
BVZ-venus	—	—	1.06	1.19	1.11	2.3	4.1	5.7	6.2	13.5	6.0	5.1
KZ2-sawtooth	—	—	1.68	2.49	1.48	2.6	4.3	8.1	9.3	7.5	8.8	4.0
KZ2-venus	—	—	2.98	4.14	1.39	3.3	6.2	8.8	11.2	18.0	13.5	8.1
rmf-wide-14	16807	6.6	0.17	0.57	3.35	99.6	385.5	57.1	113.7	492.1	339.9	1659.5
rmf-wide-16	65025	6.7	2.06	13.22	6.43	184.6	1339.2	97.7	413.0	1161.0	982.6	8835.8
rmf-wide-18	259308	6.8	25.37	641.83	25.30	334.3	5923.9	150.4	3417.3	2626.8	6635.4	85807.3

Most vision instances are easy, with few operations per vertex. To see what happens on harder problems, and to observe asymptotic trends, we use the DIMACS [14] family that is hardest for modern algorithms, `rmf-wide`. In this case, each entry in the table is the average of five instances with the same parameters and different seeds. On this family, IB is asymptotically faster than UBK, but not competitive with good general-purpose codes [10]. For larger instances, UBK performs more operations of every kind, including orphan processing. In addition, it performs a large number of OT operations.

We also experimented with other DIMACS problem families. On all of them IBFS outperformed UBK, in some cases by a very large margin.

7 Concluding Remarks

We presented a theoretically justified analog to the BK algorithm and showed that it is more robust in practice. We hope that the algorithm will be adopted by the vision community. Recently, Arora et al. [1] presented a new push-relabel algorithm that runs in polynomial time and outperforms BK on vision instances. It may outperform ours on some instances as well, but unfortunately we were unable to perform a direct comparison.

Note that our algorithm also applies in the semi-dynamic setting where we want to maintain shortest path trees when arbitrary arcs can be deleted from the graph, and arcs not on shortest paths can be added. We believe that the

variants of the IB algorithm introduced in Section 5 are interesting and deserve further investigation.

References

1. C. Arora, S. Banerjee, P. Kalra, and S. Maheshwari. An Efficient Graph Cut Algorithm for Computer Vision Problems. In K. Daniilidis, P. Maragos, and N. Paragios, editors, *Computer Vision–ECCV 2010*, volume 6313 of *Lecture Notes in Computer Science*, pages 552–565. Springer, 2010.
2. Y. Boykov and V. Kolmogorov. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004.
3. Y. Boykov and O. Veksler. Graph Cuts in Vision and Graphics: Theories and Applications. In N. Paragios, Y. Chen, and O. Faugeras, editors, *Handbook of Mathematical Models in Computer Vision*, pages 109–131. Springer, 2006.
4. B. Chandran and D. Hochbaum. A Computational Study of the Pseudoflow and Push-Relabel Algorithms for the Maximum Flow Problem. *Operations Research*, 57:358–376, 2009.
5. B. V. Cherkassky. A Fast Algorithm for Computing Maximum Flow in a Network. In A. V. Karzanov, editor, *Collected Papers, Vol. 3: Combinatorial Methods for Flow Problems*, pages 90–96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Trans., Vol. 158, pp. 23–30, 1994.
6. B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.
7. G. B. Dantzig. Application of the Simplex Method to a Transportation Problem. In T. C. Koopmans, editor, *Activity Analysis and Production and Allocation*, pages 359–373. Wiley, New York, 1951.
8. E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
9. J. Ford, L. R. and D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Math.*, 8:399–404, 1956.
10. A. V. Goldberg. Two-Level Push-Relabel Algorithm for the Maximum Flow Problem. In *Proc. 5th Alg. Aspects in Info. Management*, volume 5564 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2009.
11. A. V. Goldberg and S. Rao. Beyond the Flow Decomposition Barrier. *J. Assoc. Comput. Mach.*, 45:753–782, 1998.
12. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.
13. D. Goldfarb and M. D. Grigoriadis. A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow. *Annals of Oper. Res.*, 13:83–123, 1988.
14. D. S. Johnson and C. C. McGeoch. *Network Flows and Matching: First DIMACS Implementation Challenge*. AMS, 1993.
15. A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dok.*, 15:434–437, 1974.
16. V. King, S. Rao, and R. Tarjan. A Faster Deterministic Maximum Flow Algorithm. *J. Algorithms*, 17:447–474, 1994.
17. D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *J. Comput. System Sci.*, 26:362–391, 1983.