Faster and More Dynamic Maximum Flow by Incremental Breadth-First Search

Andrew V. Goldberg[1,⋆], Sagi Hed[2,*], Haim Kaplan[2,*], Pushmeet Kohli[3], Robert E. Tarjan[4,*], and Renato F. Werneck[1,*]

[1] Amazon.com Inc.
{andgold,werneck}@amazon.com
[2] School of Computer Science, Tel Aviv University
{sagihed,haimk}@cs.tau.ac.il
[3] Microsoft Research
pkohli@microsoft.com
[4] Department of Computer Science, Princeton University and Intertrust Technologies
ret@cs.princeton.edu

Abstract. We introduce the *Excesses Incremental Breadth-First Search* (Excesses IBFS) algorithm for maximum flow problems. We show that Excesses IBFS has the best overall practical performance on real-world instances, while maintaining the same polynomial running time guarantee of $O(mn^2)$ as IBFS, which it generalizes. Some applications, such as video object segmentation, require solving a series of maximum flow problems, each only slightly different than the previous. Excesses IBFS naturally extends to this dynamic setting and is competitive in practice with other dynamic methods.

1 Introduction

The maximum flow problem and its dual, the minimum s–t cut problem, are fundamental optimization problems with applications in a wide range of areas such as network optimization, computer vision, and signal processing. We present a new robust algorithm for the maximum flow problem that is particularly suitable for dynamic applications. We prove a strongly polynomial running time bound for our algorithm and compare its performance to previous algorithms.

Experimental work has been done on Dinic's blocking flow algorithm [4,16], the Push-Relabel (PR) algorithm [5,12,13], and Hochbaum's Pseudoflow algorithm (HPF) [18,3,9]. All three have a strongly polynomial worst-case time bound. In contrast, the algorithm of Boykov and Kolmogorov (BK) [2] is purely practical: it has no strongly polynomial time bound, but is probably the most widely used algorithm in computer vision. The Incremental Breadth-First Search (IBFS) algorithm [14] shares some features with both BK and PR. It is competitive with BK in practice [14] and has the same strongly polynomial time bounds as PR: $O(mn^2)$ without sophisticated data structures and $O(mn \log(n^2/m))$ with dynamic trees, where m is the number of arcs and n is the number of vertices.

⋆ Work partly done while the author was at Microsoft Research Silicon Valley.

N. Bansal and I. Finocchi (Eds.): ESA 2015, LNCS 9294, pp. 619–630, 2015.
DOI: 10.1007/978-3-662-48350-3_52

The maximum flow problem has also been studied in a dynamic setting, where one solves a series of maximum flow instances, each obtained from the previous one by relatively few changes in the input. The naive approach is to solve each problem independently, but one can do better. Kohli and Torr [19,20,1] extend the BK algorithm to the dynamic setting for better performance.

Despite improvements over the years, there is still much room for obtaining faster running times in practice. A natural question is whether we can come up with a robust algorithm that is fast for all applications, both static and dynamic, and has a worst-case strongly polynomial time bound.

In this paper, we present the *Excesses IBFS (EIBFS)* algorithm, which generalizes IBFS. We show that EIBFS is the best overall algorithm in practice on real-world data. On most instances it is the fastest compared to all other algorithms (often by orders of magnitude); when it loses, it is by small factors. Unlike IBFS, Excesses IBFS naturally extends to the dynamic setting, where it is competitive in practice with the dynamic extension of BK [19].

Section 2 describes the EIBFS algorithm, proves its correctness, and shows that it has the same worst-case time bounds as IBFS: $O(mn \log(n^2/m))$ with dynamic trees and $O(mn^2)$ without. Section 3 describes improvements that can be implemented in both IBFS and EIBFS. Section 4 has an extensive experimental comparison of all the key players in solving maximum flow in practice. Our benchmark is a superset of previous benchmarks and as comprehensive as we could make it.

IBFS offered a faster, theoretically justified alternative to solving maximum flow. Excesses IBFS offers an even faster, still theoretically justified and dynamic alternative to all existing methods.

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

We assume that every arc a has a reverse arc a^R of capacity 0. A (feasible) flow f is an anti-symmetric function (i.e. $f(a) = -f(a^R)$) on $E \cup E^R$ that satisfies *capacity constraints* on all arcs and *conservation constraints* at all vertices except s and t. The capacity constraint for an arc (v, w) is that $f(v, w) \leq u(v, w)$. The conservation constraint for v is $\sum_{(u,v)\in E} f(u, v) = \sum_{(v,w)\in E} f(v, w)$. The *flow value* is the total flow into the sink: $|f| = \sum_{(v,t)\in E} f(v, t)$. A *cut* is a partitioning of vertices $S \cup T = V$ with $s \in S$ and $t \in T$. The capacity of a cut is defined as $u(S, T) = \sum_{(v,w)\in E | v\in S, w\in T} u(v, w)$. The max-flow/min-cut theorem [10] states that the maximum flow value is equal to the minimum capacity of a cut.

The *residual capacity* of an arc $a \in E \cup E^R$ is defined by $u_f(a) = u(a) - f(a)$. The *residual graph* $G_f = (V, E_f)$ is the graph induced by the arcs in $E \cup E^R$ with strictly positive residual capacity. A *valid distance labeling* from s is an integral function d_s on V that given a flow f satisfies $d_s(s) = 0$ and $d_s(w) \leq d_s(v) + 1$ for every arc $(v, w) \in E_f$. 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 $(v, w) \in E_f$ and $d_s(v) = d_s(w) - 1$, and *admissible w.r.t.* d_t if $(v, w) \in E_f$ and $d_t(w) = d_t(v) - 1$.

2 Excesses IBFS

Unlike IBFS and BK, which always maintain a feasible flow, Excesses IBFS is a generalization of IBFS that maintains a *pseudoflow*, a flow that observes capacity but not conservation constraints. For a vertex v, let $e_f(v) = \sum_{w|(w,v)\in E} f(w,v) - \sum_{w|(v,w)\in E} f(v,w)$. We say v is an *excess* if $e_f(v) > 0$ and a *deficit* if $e_f(v) < 0$. We define s and t to have infinite excess and deficit, respectively.

Pseudoflows often allow to efficiently restart an algorithm after solving a problem to solve related problems. For example, for the global minimum cut problem [17] and the parametric flow problem [11], one gets the same running time bound for a sequence of flow computations as that for a single computation.

EIBFS maintains a pair of vertex-disjoint forests S and T in the admissible subgraph. Each excess is a root of a tree in S, and a root in S must be an excess. Similarly, each deficit is a root of a tree in T, and a root in T must be a deficit. For a non-root vertex v in S or T, we let $p(v)$ be the parent of v in its respective forest. We call a vertex which is not in S nor in T a *free* vertex.

The algorithm maintains distance labels $d_s(v)$ and $d_t(v)$ for every vertex v. The forest arcs in S and T are admissible with respect to d_s and d_t, respectively. Initially, every root r in S or in T has $d_s(r) = 0$ or $d_t(r) = 0$, respectively. New excesses and deficits that form as the algorithm runs may have arbitrary distance labels, so the roots of the forests do not necessarily have zero distance label. Similar forests have been introduced before in an algorithm for finding a global minimum cut [17]. We also maintain $D_s = \max_{v\in S} d_s(v)$ and $D_t = \max_{v\in T} d_t(v)$.

Initially, S contains only s, T contains only t, $d_s(s) = d_t(t) = 0$, $D_s = D_t = 0$ and $p(v)$ is null for every vertex v. The algorithm proceeds in phases. Every phase is either a *forward phase* (where we grow the S forest) or a *reverse phase* (where we grow the T forest). Every phase executes growth steps, which may be interrupted by augmentation steps (when an augmenting path is found) followed by alternating adoption and augmentation steps.

We describe a forward phase; reverse phases are symmetric. The goal of a forward phase is to grow S by one level. If S has vertices at level $D_s + 1$ at the end of the phase, we increment D_s; otherwise we terminate.

We execute growth steps as in IBFS. When the phase starts we make all vertices v in S with $d_s(v) = D_s$ *active*. We then pick an active vertex v and scan v by examining residual arcs (v, w). If w is in S, we do nothing. If w is free, we add w to S, 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. We remember (v, w) as the outgoing arc that triggered the augmentation step. If v is still active after the augmentation step, we resume the scan of v from (v, w) to avoid re-scanning the preceding arcs. If (v, w) is still residual and connects the forests, we do more augmentation steps using it. After all arcs out of v have been scanned, v becomes inactive. When all vertices are inactive, the phase ends.

Augmentation steps differ from those of IBFS. When we find a connecting arc (v, w) with v in S and w in T we increase the flow on (v, w) by any feasible amount without violating the capacity constraint of (v, w) (we will discuss the best strategy for choosing the amount later). As a result of adding flow, an

excess may be created in T and a deficit may be created in S. We now alternate between augmentation steps and adoption steps as we describe below. Once all excesses have been drained or removed from T and all deficits have been drained or removed from S we continue to perform growth steps.

We describe how we handle excesses created in T. We handle deficits in S symmetrically. We call a vertex $v \in T$ an *orphan* if its parent arc $(v, p(v))$ is not admissible (possibly saturated) and $e_f(v) \geq 0$. We execute an augmentation step by picking an excess $v \in T$ and pushing flow out of v as described below, possibly creating orphans and more excesses in T. If the augmentation step created orphans, we run adoption steps to repair them. After orphans are repaired we execute another augmentation step from another excess. We stop when all excesses are drained or removed from T. The excesses can be picked in any arbitrary order; highest level order seems to work well in practice.

We push flow out of an excess $v \in T$ as follows. We traverse the tree path from v to the root r of its tree in T. For every arc (x, y) along this path, in turn, we increase the flow by $\min\{u_f(x, y), e_f(x)\}$. It follows that we either drain the entire amount of excess from x or saturate the arc (x, y), making x an orphan in T. Root r remains a deficit if we did not drain enough excess into it. Otherwise it has $e_f(r) \geq 0$ and becomes an orphan; it can no longer serve as a root in T.

An adoption step repairs an orphan v in T by either setting a new parent, $p(v)$ in T or removing v from T. There are different methods to performing adoption steps. The simplest one is *round robin* adoption described below. More advanced methods are described in Section 3. In either method, if v is removed from T and v still has excess, then v is added to S as a new root with distance label $d_s(v) = D_s + 1$ in a forward phase or $d_s(v) = D_s$ in a reverse phase.

The original IBFS algorithm can be seen as a restricted version of EIBFS with a specific strategy for choosing the amount of flow to push on a connecting arc (v, w) between S and T. This strategy is to always take the bottleneck residual capacity along the tree path from s to v, the arc (v, w), and the tree path from w to t. Such an augmentation step will never create additional excesses or deficits in its tree. As a result, the S and T forests will simply be BFS trees rooted in s and t, respectively.

Round-Robin Adoption. We describe adoption steps in T. The adoption steps in S are symmetric. For efficiency, we maintain for every vertex a *current arc*, which ensures that each arc incident to a vertex v is scanned at most once following each increase in $d_t(v)$. When a free vertex is added to T 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 of each vertex are not admissible.

The round robin method is based on the *relabel* operation of the push-relabel algorithm [15]. An adoption step on a vertex v works as follows. We first scan v's adjacency list starting from the current arc and stop when we find an admissible outgoing arc or reach the end of the list. If we find an admissible arc (v, u) we set the current arc of v to (v, u) and set $p(v) = u$. If we do not find such an arc, we apply the *orphan relabel* operation to v.

The orphan relabel operation scans v's adjacency list to find a new parent u for v. Vertex u qualifies to be a new parent of v if (1) u is a vertex of minimum $d_t(u)$ such that (v, u) is residual and (2) $d_t(u) < D_t$ in a forward phase and $d_t(u) \leq D_t$ in a reverse phase. If no vertex u qualifies as a new parent of v then we make v a free vertex if $e_f(v) = 0$ or add it to S as a new root if $e_f(v) > 0$.

If there is a vertex u that qualifies to be a parent of v then we choose u to be the first such vertex along v's adjacency list. We set the current arc of v to (v, u), set $p(v) = u$ and set $d_t(v) = d_t(u) + 1$. Every vertex w with $p(w) = v$ now becomes an orphan and needs to be repaired by adoption steps as well.

If v is active and we execute the orphan relabel operation on v, then we make v inactive (v is no longer in T or no longer with distance label $d_t(v) = D_t$).

Pushing Flow on Connecting Arcs. When a growth step finds an arc (v, w) with $v \in S$ and $w \in T$, we must decide how to increase the flow on (v, w). As we show later, the rule below ensures a strongly polynomial time bound.

Let r_v be the root of v's tree in S and let r_w be the root of w's tree in T. Let b_v be the bottleneck capacity along the path from v to r_v and let b_w be the bottleneck capacity along the path from w to r_w. Consider the following cases:

1. If $r_v = s$ and $r_w = t$, we push $u_f(v, w)$.
2. If $r_v = s$ and $r_w \neq t$, we push $\min\{b_v, u_f(v, w)\}$, thus creating no deficits in S (except v temporarily).
3. If $r_v \neq s$ and $r_w = t$, we push $\min\{u_f(v, w), b_w\}$, thus creating no excesses in T (except w temporarily).
4. If $r_v \neq s$ and $r_w \neq t$ we push $\min\{e_f(r_v), b_v, u_f(v, w), b_w, -e_f(r_w)\}$, thus creating no deficits or excesses in S or T (except v or w temporarily).

Correctness and Running Time. Correctness for EIBFS relies on the following lemma, which is the counterpart of Lemma 1 of IBFS [14]. See the full version of this extended abstract for a complete proof of correctness.

Lemma 1. *During a forward phase, 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 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 $d_s(u) = D_s$.*

Next we show that the worst-case time complexity of EIBFS is $O(mn^2)$, which can be improved to $O(mn \log(n^2/m))$ using dynamic trees and existing techniques. We first consider the invariants maintained by the algorithm. These are the counterparts of the invariants in Lemma 2 of IBFS [14].

Lemma 2. *The following invariants hold:*

1. *If (v, w) is residual with $v, w \in S$, then $d_s(w) \leq d_s(v) + 1$. If (v, w) is residual with $v, w \in T$, then $d_t(v) \leq d_t(w) + 1$.*
2. *For every vertex u in S, u's current arc precedes the first admissible arc to u or is equal to it. For every vertex u in T, u's current arc precedes the first admissible arc from u or is equal to it.*

3. *After an adoption step on u: if u is in S and $e_f(u) \leq 0$ then $(p(u), u)$ is admissible; if u is in T and $e_f(u) \geq 0$ then $(u, p(u))$ is admissible.*
4. *For every vertex v, $d_s(v)$ and $d_t(v)$ never decrease.*

Proof. The proof, by induction on the growth, augmentation and adoption steps, is the same as that of Lemma 2 for IBFS [14]. Although augmentation steps differ a little, the same arguments hold for EIBFS. The only addition is the case of removing an excess from T or a deficit from S during an adoption step.

We consider a vertex v with $e_f(v) > 0$ removed from T during an adoption step; the case of a deficit removed from S is symmetric. Since we reset the current arc of v, (2) holds. Since $e_f(v) > 0$, (3) does not apply. We assign v the highest possible label in S (either $D_s + 1$ for a forward phase or D_s for a reverse phase), so (4) holds. We are left to show that invariant (1) is maintained.

We assume there is a residual arc (u, v) with u in S, otherwise (1) holds vacuously. If u was in S at the beginning of the current phase (v was not) then by Lemma 1 u was on the last level of S and thus $d_s(u) = D_s$ at that time. By induction assumption of (4) we get that now $d_s(u) \geq D_s$ and thus $d_s(v) \leq d_s(u) + 1$. If u was not in S at that time, then by definition of the algorithm it could only have been added to S with a label $D_s + 1$. By induction assumption of (4) we get that now $d_s(u) \geq D_s + 1$ and thus $d_s(v) \leq d_s(u) + 1$. □

Adoption steps on a vertex charge their work to increases in the vertex's distance label. Lemma 3 is the counterpart of Lemma 5 in IBFS [14] and shows why this charging is possible. The proof is the same as in Lemma 5 [14]. Lemma 4 and 5 allow us to bound the maximum label assigned during the algorithm.

Lemma 3. *After an orphan relabel on v in S, $d_s(v)$ increases. After an orphan relabel on v in T, $d_t(v)$ increases.*

Lemma 4. *For every vertex v in S with $e_f(v) > 0$ we have $d_s(v) \leq n$. For every vertex v in T with $e_f(v) < 0$ we have $d_t(v) \leq n$.*

Proof. We prove the lemma for a vertex v in S. The proof for T is symmetric. We put new excesses in S only when we remove an excess from T during an adoption step that follows an augmentation step. Let (x, y) be the connecting arc between S and T that initiated this augmentation step.

Since we created excesses in T and by the definition of the flow increase on (x, y) we get that s is the root of x's tree. By applying Lemma 2 (1) to the path from s to v we get that at the time we initiated the augmentation step we had $d_s(x) \leq n - 1$. By definition of the algorithm we get that at the same time we had $d_s(x) \geq D_s$. It follows that $D_s \leq d_s(x) \leq n - 1$ and therefore $D_s + 1 \leq n$. Since we assign $d_s(v) = D_s + 1$ the lemma follows. □

Lemma 5. *For every vertex v in S we have $d_s(v) < 2n$. For every vertex v in T we have $d_t(v) < 2n$.*

Proof. We prove the lemma for a vertex v in S. The proof for T is symmetric. Let vertex r in S, $e_f(r) > 0$ be the root of v's tree. By applying Lemma 2 (1) to the path from r to v we get that $d_s(v) \leq d_s(r) + n - 1$. By Lemma 4 we get that $d_s(r) \leq n$. It follows that $d_s(v) \leq 2n - 1$. □

By Lemma 5 the maximum d_s or d_t label is $O(n)$. The following theorem follows using the same arguments as in the proof of Lemma 6 for IBFS [14].

Theorem 1. *Excesses IBFS runs in $O(n^2m)$ time.*

Dynamic Setting. We now consider the dynamic setting: after computing a maximum flow in the network, the capacities of some arcs change and we must recompute a maximum flow as fast as possible. IBFS does not seem to provide a robust method for recomputing a maximum flow other than starting the S and T trees from scratch. EIBFS, however, naturally lends itself to this setting. We first restore the invariants that were violated by changing the capacities, then run EIBFS normally continuing with the residual flow and forests from the previous computation.

Consider the network after changing some capacities. There are several types of violations to flow feasibility or to the invariants of the EIBFS that may follow:

1. An arc (v, w) such that now $f(v, w) > u(v, w)$.
2. A new residual arc (v, w) such that v is in S and w is in T.
3. A new residual arc (v, w) such that v and w are in S and $d_s(w) > d_s(v) + 1$, or the symmetric case for T.
4. A new residual arc (v, w) such that v and w are in S, $d_s(w) = d_s(v) + 1$, and (v, w) precedes the current arc of v, or the symmetric case for T.
5. A new residual arc (v, w) such that v is in S, $d_s(v) \leq D_s$ and w not in S, or the symmetric case for T.

Violation (4) can be fixed by reassigning the current arc of v. Violation (1) can be fixed by pushing flow on (w, v); (2), (3) and (5) can be fixed by saturating (v, w). In both cases the end result is the creation of new excesses or deficits in S and T. Excesses in S and deficits in T become new roots and need no further handling. Deficits in S and excesses in T are treated by alternating augmentation and adoption steps as when we find a connecting arc between S and T.

We found that in practice it pays off to reset the forests every $O(m)$ work. After a reset, the S and T forests are composed only of excesses or deficits, respectively, as roots with distance label 0. Note that this scans the nodes array once. This is similar in concept to Push-Relabel's global update operation [5].

3 Improvements to IBFS and Excesses IBFS

Forward or Reverse Phases. IBFS or EIBFS can use different strategies to alternate between forward and reverse phases. The original version of IBFS strictly alternated between them, producing trees with roughly the same height. As observed in [14], IBFS often spends the majority of its time on adoption steps. If arc capacities are distributed independently uniformly at random, balancing by height also tends to balance the amount of adoption work. In practice, however, this strategy is often far from optimal. A more robust alternative is to maintain an operation count that is indicative of the total amount of adoption work in

each forest. We run a reverse phase when this counter is higher for S than for T, and a forward phase otherwise. Counting the number of distinct orphans examined (which is fairly oblivious to the choice of adoption method) in every adoption process works well in practice.

Alternative Adoption Strategies. The round-robin adoption strategy tends to be quite fast, but in pathological cases it may process the same vertex a large number of times. We thus propose a *three-pass adoption* strategy, which looks at each arc adjacent to an orphan at most three times during the entire adoption process of one augmentation step. This is more robust, and cannot be outperformed by the round-robin method by more than a constant factor.

This strategy associates a bucket (linked list) with each distance label. We denote by $B(v)$ the distance label associated with the bucket containing v. The method works in two rounds: the first incurs at most one pass of the adjacency list for every orphan; the second round incurs at most two more.

We describe the adoption in T (S is symmetric). The first round examines every orphan v in T in ascending order of distance labels. We scan v's adjacency list starting from the current arc and stop as soon as we find a residual arc (v, u) with $d_t(u) = d_t(v) - 1$. If such a vertex u is found, we set $p(v) = u$ and set the current arc of v to (v, u). If no such u is found, we remove v from T (v becomes a free vertex), put v in bucket $d_t(v) + 1$, and make the children of v in T orphans.

The second round iterates over the buckets in ascending order of distance labels. We examine every orphan v in the bucket. If this is the second pass of v, then we perform an orphan relabel operation as in Section 2. If v finds a potential parent u in T (note that u is not an orphan) we move it to the bucket $d_t(u) + 1$. If vertex v did not find a potential parent, it remains free, but it may be reattached to T later in the round.

If this is the third pass of v, we scan v's adjacency list, performing two operations. The first operation is to find a parent u in T for which $d_t(u) = B(v) - 1$ and (v, u) is residual. At the time of the third pass we are guaranteed to find such a parent. At the end of the scan we set the current arc of v to be (v, u), set $p(v) = u$ and set $d_t(v) = B(v)$. The second operation applies to every neighbor w with (w, v) residual and w either free or in a bucket $B(w) > B(v) + 1$. We put w in the bucket $B(v) + 1$ and remove it from any other bucket.

In practice, we use a *hybrid method*, which works as follows for every adoption process of one augmentation step. It starts with the round-robin method while keeping count of the average number of times each orphan is examined. If this average exceeds 3, it processes all remaining orphans using the three-pass method. We found that this method combines the best of both worlds and outperforms the round-robin and the three-pass methods in practice.

4 Experimental Results

In our experiments we use the implementation of BK version 3.0.1 from `http://www.cs.ucl.ac.uk/staff/V.Kolmogorov/software.html`. That implementation allows for dynamic capacities on the arcs from the source and to

the sink. We added the option for dynamic capacities on all arcs. The dynamic version of BK was formulated by Kohli and Torr [19]. We also compare to UBK, an altered version of BK that maintains a consecutive arc structure: the arcs reside in an array grouped by the vertex they originate from (same as in [14]). We denote our implementation of Excesses IBFS with all the optimizations of Section 3 by EIBFS, and use IBFS to denote the implementation from [14]. The implementations of EIBFS and IBFS maintain a consecutive arc structure. We use the implementation of Hochbaum's pseudoflow (HPF) version 2.3 from `http://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html`. We run the highest label FIFO variant of HPF, as recommended by the download page. We run also an implementation of Two-Level Push-Relabel (P2R) [12]. Our implementation of EIBFS and benchmark data are available at `http://www.cs.tau.ac.il/~sagihed/ibfs/`.

We run our experiments on a 64-bit Windows 8 machine with 8 GB of RAM and an Intel i5-3230M 2.6 GHz processor (two physical cores, 256 KB L1 cache). We used the MinGW g++ compiler with -O3 optimization settings. We compile with 64-bit or 32-bit pointers depending on problem size. We report system times of the maximum flow computation obtained with *ftime*. On the rare occasions in which running times are too small to measure, we round them up to a millisecond. We report absolute times (in seconds) for EIBFS and relative (to EIBFS) times for all algorithms. Factors greater than 1 mean EIBFS was faster.

Batra and Verma [22] noted that initialization times may be significant. Therefore our reported times include any initialization time past the initial reading of arcs. In all implementations, this reading consists of only two operations: writing arcs consecutively to memory and advancing the count of vertex degrees.

Table 1 reports results for static problems. We consider representative instances from a wide variety of families (see the full version of this extended abstract for a complete set of results). Multi-view reconstruction, 3D segmentation, stereo images, surface fitting and the first video segmentation family are provided by University of Western Ontario (`http://vision.csd.uwo.ca/maxflow-data`). Families of deconvolution, decision tree field (DTF), super resolution, texture restoration, automatic labeling environment (ALE) and synthetic segmentation are provided by `http://ttic.uchicago.edu/~dbatra/research/mfcomp/` [22]. Another synthetic family is from the DIMACS maximum flow challenge (`http://dimacs.rutgers.edu/Challenges/`). We run also on families of image and video segmentation with GMM models, multi-label image segmentation [1], lazy-brush image painting [21], road newtwork partitioning (PUNCH) [6], and graph bisection [7,8]. For stereo, ALE, PUNCH, and bisection, we report times summed over similar instances. Capacities are integral for all problems.

On real-world problems, EIBFS is the fastest overall algorithm, sometimes by orders of magnitude. It often improves the performance of IBFS but sometimes slows it down by marginal factors. EIBFS is the overall fastest the stereo, multi-view, 3D segmentation, surface fitting, and video frame families, losing occasionally to other algorithms only by small factors. On DTF problems HPF is the fastest, outperforming EIBFS by 20% on average. On lazy-brush problems EIBFS is fastest

Table 1. Performance on real-world and synthetic static inputs

FAMILY	INSTANCE NAME	$\frac{n}{1024}$	$\frac{m}{n}$	EIBFS TIME[s]	RELATIVE TIMES EIBFS	IBFS	BK	UBK	HPF	P2R
stereo	BVZ-tsukuba	108	4.0	0.249	1.00	**0.94**	1.02	1.09	3.50	7.49
	KZ2-venus	294	5.7	1.643	1.00	**0.92**	1.27	1.37	3.23	7.02
multi-view	camel-med	9450	4.0	7.125	**1.00**	1.95	2.67	2.75	1.52	4.44
	gargoyle-med	8640	4.0	8.516	1.00	1.62	11.73	9.35	**0.93**	2.35
3D	adhead6c100(64bit)	12288	6.0	11.250	**1.00**	1.08	2.38	1.95	1.71	1.95
segmentation	babyface6c100	4943	6.0	3.336	**1.00**	1.10	2.22	2.10	3.86	6.22
	bone6c100	7616	6.0	2.180	**1.00**	1.84	1.87	1.54	1.67	2.36
	bone_sx26c100(64bit)	3808	26.0	4.171	**1.00**	1.68	2.90	1.90	1.09	1.25
	bone_sx6c100	3808	6.0	1.492	**1.00**	1.27	2.73	2.28	1.79	1.82
	bone_sxyz26c10	960	26.0	0.742	1.00	1.98	2.04	1.24	**0.71**	1.53
	liver6c100	4064	6.0	3.391	**1.00**	1.16	2.78	2.43	2.15	2.79
surface fitting	bunny-med	6163	6.0	0.687	**1.00**	1.07	1.36	1.15	3.77	22.28
video1	car_32bins	77	9.7	0.015	**1.00**	1.01	9.41	2.92	1.99	5.61
(single frame)	person_16bins	107	9.9	0.015	**1.00**	1.10	154.74	19.74	1.21	5.41
	videoSegA	168	8.0	0.051	**1.00**	1.03	1.46	1.52	5.58	13.62
	videoSegB	225	8.0	0.015	1.00	1.01	**0.70**	1.10	1.58	3.31
	videoSegC	234	8.0	0.042	**1.00**	1.26	1.44	1.48	1.67	3.03
deconvolution	graph3x3	1	21.9	0.001	**1.00**	2.00	1.94	2.00	**1.00**	**1.00**
	graph5x5	1	67.7	0.004	1.00	1.00	7.50	5.19	0.65	**0.25**
DTF	printed_graph1	19	56.7	0.069	1.00	1.09	7.83	4.34	**0.77**	1.31
	printed_graph16	11	55.2	0.034	1.00	1.00	5.69	3.07	**0.80**	1.17
lazy-brush	lbrush-bird	2316	4.0	3.070	**1.00**	3.11	1.28	1.04	1.97	3.81
	lbrush-doctor	2317	4.0	1.140	**1.00**	18.00	1.47	1.15	1.10	9.45
	lbrush-mangagirl	579	4.0	0.273	1.00	4.89	1.23	**0.94**	1.43	8.38
	lbrush-elephant	2314	4.0	2.930	**1.00**	4.95	1.10	1.05	1.13	4.62
texture	texture_graph	42	15.1	0.010	1.00	1.14	**0.57**	1.28	0.86	1.72
resolution	superres_graph	42	15.1	0.007	1.00	1.00	**0.19**	1.19	0.99	1.98
segmentation	butterfly	453	8.0	0.084	**1.00**	1.11	1.52	1.59	5.96	8.22
	comp	236	8.0	0.078	**1.00**	1.40	2.20	2.21	4.01	3.89
	ferro	230	8.0	0.056	**1.00**	1.27	2.11	2.28	6.17	16.57
	flamingo2	468	8.0	0.106	**1.00**	1.06	1.38	1.35	3.94	5.08
PUNCH	punch-eu22p	1825	2.8	29.219	1.00	1.78	3.26	2.65	**0.50**	1.60
	punch-eu22u	1825	2.8	10.517	1.00	2.84	2.37	1.96	**0.89**	3.62
	punch-us22p	1596	2.8	47.189	1.00	1.96	2.98	2.38	**0.32**	0.76
	punch-us22u	1596	2.8	10.859	1.00	6.22	2.40	2.02	**0.85**	2.48
bisection	alue7065	32	3.2	0.110	**1.00**	2.00	1.29	1.42	1.58	4.83
	cal	1761	2.5	7.156	**1.00**	8.45	1.63	1.45	1.64	8.91
	horse	46	6.0	0.313	1.00	1.00	**0.55**	**0.55**	1.94	3.19
	rgg18	254	11.8	5.282	1.00	1.48	0.65	**0.46**	1.75	2.78
ALE	graph_2007_000033	168	27.3	0.915	**1.00**	1.02	175.23	11.32	27.29	24.03
	graph_2007_001288	161	29.0	0.904	**1.00**	1.02	186.73	6.77	69.02	19.55
segmentation	0.000099502487562_1	39	4.0	0.044	1.00	**0.78**	0.99	0.92	**0.78**	1.56
(synthetic)	0.002148473323752_1	39	45.0	0.059	**1.00**	1.06	9.34	6.49	3.22	1.58
	0.004631311287980_1	39	94.5	0.112	**1.00**	1.03	9.56	5.62	2.11	1.47
	0.021520481611665_1	39	432.4	0.584	1.00	1.12	9.52	4.51	**0.69**	**0.69**
	0.100000000000000_1	39	2001.6	3.278	1.00	0.93	9.65	3.93	**0.61**	0.75
DIMACS	ac.n1024	1	1019.0	0.034	1.00	1.07	37.76	4.71	0.91	**0.63**
(synthetic)	ac.n4096	4	4091.0	1.337	1.00	1.18	42.97	7.41	**0.42**	0.50
	rmf-long.n4	264	5.8	11.490	1.00	1.73	7.87	5.18	0.04	**0.02**
	rmf-wide.n4	120	5.8	2.696	1.00	1.21	19.63	13.34	**0.17**	0.30
	wash-line.n16384-64	64	127.7	1.034	1.00	2.68	4.98	3.32	0.51	**0.45**
	wash-line.n8192-45	32	89.8	0.218	1.00	3.15	7.51	4.54	**0.39**	0.53
	wash-rlg-long.n2048	128	6.0	1.621	1.00	6.97	16.42	12.74	**0.07**	**0.07**
	wash-rlg-wide.n2048	128	5.9	1.037	1.00	1.27	213.70	156.68	**0.19**	0.32

Table 2. Performance on real-world dynamic input

INSTANCE				DYNAMIC EIBFS		RELATIVE TIMES			
FAMILY	NAME	$\frac{n}{1024}$	$\frac{m}{n}$	ITERS	TIME[s]	EIBFS	BK	UBK	NIBFS
bisection	alue7065	32	3.2	9.0	0.074	1.00	**0.47**	0.50	0.62
	cal	1760	2.5	8.0	5.846	1.00	1.26	1.03	**0.86**
	horse	46	6.0	9.0	0.072	1.00	0.78	**0.65**	0.95
	rgg18	254	11.8	9.0	1.753	1.00	0.52	**0.48**	1.34
video1	car_32bins.inc	77	9.7	4320.0	2.691	**1.00**	2.27	1.10	7.17
	person_16bins.inc	107	9.9	28380.0	21.502	**1.00**	11.78	2.92	9.17
	videoSegA.inc	168	8.0	49.0	0.867	**1.00**	1.96	1.91	1.41
	videoSegB.inc	225	8.0	49.0	0.236	1.00	0.61	**0.49**	1.41
video2	gir.inc	405	8.0	4.0	0.078	**1.00**	2.00	1.64	1.36
	highway.inc	75	8.0	40.0	0.177	**1.00**	1.90	1.91	1.31
	office.inc	84	8.0	46.0	0.279	1.00	1.78	1.46	**0.92**
	pedestrians.inc	84	8.0	50.0	0.083	**1.00**	1.13	1.35	1.73
multi-label	cowInc00	405	8.0	16.0	0.001	**1.00**	1.50	**1.00**	29.60
	gardenInc00	20	7.9	28.0	0.001	**1.00**	**1.00**	**1.00**	4.60

and improves IBFS considerably. BK is the fastest on texture restoration and super resolution problems, since most of the running time is taken up by initialization (as seen by comparing BK to UBK). On image segmentation problems EIBFS is fastest. On PUNCH problems HPF is fastest, outperforming EIBFS by 25% on average. On bisection problems EIBFS and BK/UBK are competitive. On ALE problems EIBFS is faster by orders of magnitude compared to all other algorithms except IBFS. On synthetic problems, EIBFS is faster than IBFS but can still lose by orders of magnitude to HPF and P2R (especially on DIMACS instances). We note that some have very large vertex degrees, with most of the time used for initialization of the arc structure.

Table 2 considers dynamic problems. Dynamic video segmentation aligns maximum flow problems from consecutive video frames as one dynamic maximum flow set. Dynamic multi-label image segmentation aligns maximum flow problems from consecutive alpha expansion iterations over the same label. Dynamic bisection aligns maximum flow problems from nearby branches of a branch-and-bound tree. The table shows that, for dynamic applications, EIBFS is competitive with UBK, which in turn tends to be faster than BK. We also include NIBFS, a more naive implementation of IBFS for the dynamic setting. After every set of incremental changes, it only fixes violations on arcs where the flow is greater than the capacity; it then resets the S and T forests as in the periodic update of dynamic EIBFS. The results show that EIBFS is much more robust: it can outperform NIBFS by large factors but the converse is false.

References

1. Alahari, K., Kohli, P., Torr, P.H.S.: Dynamic hybrid algorithms for MAP inference in discrete mrfs. IEEE PAMI 32(10), 1846–1857 (2010)
2. Boykov, Y., Kolmogorov, V.: An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. IEEE PAMI 26(9), 1124–1137 (2004)

3. Chandran, B., Hochbaum, D.: A computational Study of the Pseudoflow and Push-Relabel Algorithms for the Maximum flow Problem. Operations Research 57, 358–376 (2009)
4. Cherkassky, B.V.: A Fast Algorithm for Computing Maximum Flow in a Network. In: Karzanov, A.V. (ed.) Collected Papers, Vol. 3: Combinatorial Methods for Flow Problems, pp. 90–96. The Institute for Systems Studies, Moscow (1979) (in Russian) English translation appears in AMS Trans., 158, 23–30 (1994)
5. Cherkassky, B.V., Goldberg, A.V.: On Implementing Push-Relabel Method for the Maximum Flow Problem. Algorithmica 19, 390–410 (1997)
6. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph partitioning with natural cuts. In: 25th IEEE IPDPS, pp. 1135–1146 (2011)
7. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Exact combinatorial branch-and-bound for graph bisection. In: ALENEX, pp. 30–44 (2012)
8. Delling, D., Werneck, R.F.: Better bounds for graph bisection. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 407–418. Springer, Heidelberg (2012)
9. Fishbain, B., Hochbaum, D.S., Mueller, S.: Competitive analysis of minimum-cut maximum flow algorithms in vision problems. CoRR, abs/1007.4531 (2010)
10. Ford Jr., L.R., Fulkerson, D.R.: Maximal Flow Through a Network. Canadian Journal of Math. 8, 399–404 (1956)
11. Gallo, G., Grigoriadis, M.D., Tarjan, R.E.: A Fast Parametric Maximum Flow Algorithm and Applications. SIAM J. Comput. 18, 30–55 (1989)
12. Goldberg, A.: Two Level Push-Relabel Algorithm for the Maximum Flow Problem. In: Proc. 5th Alg. Aspects in Info. Management. Springer, New York (2009)
13. Goldberg, A.V.: The partial augment–relabel algorithm for the maximum flow problem. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 466–477. Springer, Heidelberg (2008)
14. Goldberg, A.V., Hed, S., Kaplan, H., Tarjan, R.E., Werneck, R.F.: Maximum flows by incremental breadth-first search. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 457–468. Springer, Heidelberg (2011)
15. Goldberg, A.V., Tarjan, R.E.: A New Approach to the Maximum Flow Problem. J. Assoc. Comput. Mach. 35, 921–940 (1988)
16. Goldfarb, D., Grigoriadis, M.: A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow. Ann. Op. Res. 13, 83–123 (1988)
17. Hao, J., Orlin, J.B.: A Faster Algorithm for Finding the Minimum Cut in a Directed Graph. J. Algorithms 17, 424–446 (1994)
18. Hochbaum, D.S.: The pseudoflow algorithm: A new algorithm for the maximum-flow problem. Operations Research 56(4), 992–1009 (2008)
19. Kohli, P., Torr, P.H.S.: Dynamic graph cuts for efficient inference in markov random fields. IEEE Trans. Pattern Anal. Mach. Intell. 29(12), 2079–2088 (2007)
20. Kohli, P., Torr, P.H.S.: Measuring uncertainty in graph cut solutions. Computer Vision and Image Understanding 112(1), 30–38 (2008)
21. Sýkora, D., Dingliana, J., Collins, S.: Lazybrush: Flexible painting tool for hand-drawn cartoons. Comput. Graph. Forum 28(2), 599–608 (2009)
22. Verma, T., Batra, D.: Maxflow revisited: An empirical comparison of maxflow algorithms for dense vision problems. In: BMVC, pp. 1–12 (2012)