J Real-Time Image Proc (2016) 11:589–609
DOI 10.1007/s11554-013-0344-3

ORIGINAL RESEARCH PAPER

A competitive study of the pseudoflow algorithm for the minimum *s–t* cut problem in vision applications

B. Fishbain · Dorit S. Hochbaum · Stefan Mueller

Received: 14 June 2012 / Accepted: 25 March 2013 / Published online: 11 April 2013

Abstract Rapid advances in image acquisition and storage technology underline the need for real-time algorithms that are capable of solving large-scale image processing and computer-vision problems. The *minimum s–t cut* problem, which is a classical combinatorial optimization problem, is a prominent building block in many vision and imaging algorithms such as video segmentation, co-segmentation, stereo vision, multi-view reconstruction, and surface fitting to name a few. That is why finding a real-time algorithm which optimally solves this problem is of great importance. In this paper, we introduce to computer vision the *Hochbaum's pseudoflow* (HPF) algorithm, which optimally solves the minimum *s–t* cut problem. We compare the performance of HPF, in terms of execution times and memory utilization, with three leading published algorithms: (1) Goldberg's and Tarjan's *Push-Relabel*; (2) Boykov's and Kolmogorov's *augmenting paths*; and (3) Goldberg's *partial augment-relabel*. While the common practice in computer-vision is to use either BK or PRF algorithms for solving the problem, our results demonstrate that, in general, HPF algorithm is more efficient and utilizes less memory than these three algorithms. This strongly suggests that HPF is a great option for many real-time computer-vision problems that require solving the minimum *s–t* cut problem.

Keywords Network flow algorithms · Maximum-flow · Minimum-cut · Segmentation · Stereo vision · Multi-view reconstruction · Surface fitting

B. Fishbain (✉)
Technion-Israel Institute of Technology, 32000 Haifa, Israel
e-mail: fishbain@technion.ac.il

D. S. Hochbaum
University of California, Berkeley, CA, USA
e-mail: hochbaum@ieor.berkeley.edu

S. Mueller
Technische Universitaet Berlin, Berlin, Germany
e-mail: ste.mu@arcor.de

1 Introduction

Rapid advances in image acquisition, processing and storage technologies have increased the need for faster *real-time* image processing and computer-vision algorithms that require lesser memory while being capable of handling large-scale imaging problems. The minimum $s-t$ cut problem, henceforth referred to as the *min-cut* problem, is a classical combinatorial optimization problem with applications in numerous areas of science and engineering [2].

The min-cut problem, given a finite undirected graph with nonnegative edge capacities and two nodes in the graph s and t, consists of partitioning the graph into two nonempty sets S and $\bar{S} = V \setminus S$ such that the sum of the capacities of edges connecting the two parts is minimum among all possible partitions and $s \in S$ and $t \in \bar{S}$. The broad applicability of the min-cut problem has resulted in a substantial amount of theoretical and experimental work on the subject. Since the seminal work of Ford and Fulkerson [18] defining the min-cut and its dual—the maximum-flow (max-flow) problems, many algorithms have been suggested for solving these problems. A cohort of these algorithms achieve better performance by applying certain assumptions on the problem such as a planar [8, 32] or bipartite graph [6]. However, these assumptions are not valid in most imaging and computer-vision applications.

In recent years, the min-cut problem has become a prominent building block in many vision and imaging algorithms. One can find examples of employing min-cut in applications such as image and video segmentation (e.g., [24, 27, 41], co-segmentation (e.g, [30], stereo vision (e.g., [42], multi-view reconstruction, (e.g., [45, 47], and surface fitting [50] to name a few.

In order to solve the min-cut problem in large instances, while complying with real-time constraints, one often employs a parallel implementation of the algorithm. The parallel implementation involves breaking the problem into a series of smaller sub-problems and solving in each separately the min-cut problem. Previous works utilized either BK [37, 44, 51] or PRF [15, 44, 52] for solving the sub-problems. The process is repeated until a threshold on the permissible run-time is reached or a global optimal solution is found. Typically, the runs end with a heuristic, or non-optimal, solution. Thus, this process incurs an error with respect to the original min-cut optimal solution. This error increases as the number of sub-problems increases and their sizes decrease. Therefore, the availability of a more efficient min-cut algorithm, such as HPF demonstrated here, has impact on such parallel algorithms. The availability of the extra efficient min-cut algorithm allows to parallelize the problem by breaking it into fewer parts, as compared to less efficient algorithms, and thus improve the accuracy of the solution.

Among algorithms for solving the min-cut and the max-flow problems without any assumptions in the problem, the push-relabel algorithm, PRF, of Goldberg and Tarjan [21]. Several studies have shown push-relabel to be computationally very efficient (e.g., [1, 4, 13, 16]). Boykov's and Kolmogorov's augmenting paths algorithm, BK, [10] for solving the min-cut problem, attempts to improve on standard augmenting path techniques on graphs in vision. Similarly to Ford–Fulkerson's algorithm [18], BK algorithm's complexity is only pseudo-polynomial. In this it differs from the other algorithms studied here, all of which have strongly polynomial time complexity. Despite that, it has been demonstrated in [10] that in practice on a set of vision problems, the algorithm works well. The Partial Augment-Relabel algorithm, PAR, devised by Goldberg [19], also searches for the shortest augmenting path, where at each stage of the algorithm it finds augmenting path of specific length.

We introduce here Hochbaum's pseudoflow algorithm, HPF [25], which optimally solves the min-cut problem, to vision problems. HPF was shown to be fastest in practice for general min-cut problems [12], however, its performance has not been evaluated in the context of vision and real-time vision problems and the practice has been that BK and PRF are the choice methods for the special class of vision problems. This motivates our study of comparing the performance of HPF with these algorithms for vision problems and real-time vision problems.

For evaluating the performance of HPF algorithm in vision problems, we compare it with the three aforementioned algorithms, which constitute the state-of-the-art: (1) the *Push-Relabel*, PRF, algorithm; (2) Boykov's and Kolmogorov's *augmenting paths* algorithm, BK, [10]; and (3) Goldberg's *partial augment-relabel*, PAR, algorithm [19]. The study consists of a benchmark of an extensive data set which includes four types of vision tasks: stereo vision, segmentation, multi-view reconstruction; and surface fitting, [14, 49].

The comparison of these algorithms was reported in [5, 10, 19]. The first, [10], compared BK algorithm only to PRF, and for a limited set of instances. The second report, [19], used the same limited set of instances, and compared PRF and PAR to HPF. The comparison provided by [19] to HPF compared the running times of the various algorithms, but excluded their initialization times. This invalidates the results of this comparison as our experiments show that the initialization times are significant with respect to the total running times and are different between the algorithms. The reason for this difference between the algorithms is that the logic computed in the initialization stage by each of the algorithms is different. The initialization process in BK and HPF algorithms only reads the problem file and initiates the corresponding graphs. The implementation of PRF, however, incorporates an additional logic which sorts the arcs of each node. Similar operations do exist in both HPF and BK, but they are incorporated in the main algorithm and are included in the running times reported in [19]. This also invalidates the results reported in [5], as the latter uses the times reported in [19] for comparison.

Here we provide, for the first time, a comprehensive review of the BK, HPF, PRF and PAR algorithms and a detailed comparison, including their memory usages and a breakdown of the run-times for the different stages of the algorithm (initialization, min-cut and max-flow). Note that all these algorithms solve the min-cut problem optimally. Thus, all produce the same optimal results and accuracy. For the vast majority of computer-vision applications only the min-cut solution is relevant (for recent examples see [3, 26, 27, 35, 45, 50]). Previous empirical studies made the comparison on the basis of max-flow computation times. For this reason, we report here the min-cut execution times and also the time it takes to generate the max-flow. The breakdown of the execution times, reported here, allows to evaluate the performance of the algorithms for these relevant computations by taking into account only the initialization and min-cut times. The manuscript also provides valuable practical information that should advance real-time usage of the algorithms evaluated in the paper.

Our results demonstrate that, in general, HPF algorithm is more efficient and utilizes less memory than BK, PRF

and PAR algorithms. In terms of memory utilization HPF is a lot more efficient than BK and PRF, and with up to 25 % less memory. In terms of computation times, our results show that HPF is faster than both PRF and PAR for all problem instances. BK runs slightly faster for stereo problems (6 instances). HPF's runtimes in these problems are, however, less than a fraction of a second slower. For the surface fitting problems (3 instances), which are of low degree, BK runs faster, but HPF is at most 10 % slower. Currently, BK and PRF are the common practice for solving the min-cut problem in computer vision (e.g., [9, 15, 35, 42, 45, 47]). As HPF is faster in general and large vision problems than all the other three algorithms, and its performance is at par with BK on subclass of problems, we believe that the introduction of HPF algorithm to real-time image-processing problems removes some of the obstacles of the road to real-time implementations of many computer-vision tasks which consist of the solving the min-cut problem.

The paper is organized as follows: Sect. 2 describes HPF algorithm and the three algorithms it is compared to. The experimental setup is presented in Sect. 3, followed by the comparison results, which are detailed in Sect. 4. Section 5 concludes the paper.

1.1 A graph representation of a vision problem

One way to present a vision problem is on an undirected graph $G = (V, E)$, where V is the set of vertices representing pixels and E is the set of edges connecting pairs of pixels that are considered neighbors according to a prescribed adjacency rule. Each edge has a similarity weight associated with it. For real-time vision application examples using such presentation see [7, 22, 33, 43]. Alternatively, in order to reduce the size of the problems and achieve shorter computation times, each node can represent a group of pixels. Such grouping can be done by some similarity measure [23, 40] or simply by dividing the image to square blocks of non-overlapping pixels (i.e., macro-blocks) [17, 38]. For regular grids, such as pixel-level or macro-block representation, the 4-neighbor setup is a commonly used adjacency rule with each pixel having 4 neighbors—two along the vertical axis and two along the horizontal axis. This setup forms a planar grid graph. The 8-neighbor arrangement is another setup, in which the planarity of the graph no longer holds, and consequently the complexity of various algorithms increases, sometimes significantly. Planarity also does not hold for 3-dimensional images or video. In the most general case of vision problems, no grid structure can be assumed and thus the respective graphs are not planar. Indeed, the algorithms presented here do not assume any specific property of the graph G—they work for general graphs.

The edges in the graph G representing the image carry *similarity* weights. There is a great deal of literature on how to generate similarity weights, and we do not discuss this issue here. We only use the fact that similarity is inversely increasing with the difference in attribute values between the pixels. In the graph G, each edge $\{i, j\}$ is assigned a similarity weight w_{ij} that increases as the two pixels i and j are perceived to be more similar. Low values of w_{ij} are interpreted as dissimilarity. In some contexts one might want to generate *dissimilarity* weights independently. In that case each edge has two weights, w_{ij} for similarity, and $\hat{w}_{ij}$ for dissimilarity.

1.2 Definitions and notation

We introduce here formal definitions of cuts and flows in a graph used in describing the algorithms in Sect. 2. The min-cut and max-flow problems are defined on a directed graph, $G = (V, A)$ where V is the set of nodes and A is the set of arcs connecting nodes in the graph. Using common notation, $n = |V|$ is the number of nodes and $m = |A|$ is the number of arcs in G. Each arc $(i, j) \in A$ has capacity u_{ij}.

The undirected graph $G = (V, A)$ representing a vision problem is converted to a directed graph, $G = (V, A)$, where there are two arcs, (i, j), (j, i) in A for every edge $[i, j] \in E$. The similarity weight w_{ij} of edge $[i, j]$ is the *capacity* u_{ij} and u_{ji} of (i, j) and (j, i).

1.2.1 Cuts

A bipartition of the set of nodes of a graph $S \cup \bar{S} = V$ ($\bar{S} = V \setminus S$), is associated with a set of arcs that go from S to $\bar{S}$, $(S, \bar{S}) = \{(i, j) \in A \mid i \in S, j \in \bar{S}\}$. The set of arcs, $(S, \bar{S})$, is called a *cut*. A schematic illustration of a cut is given in Fig. 1. The *capacity of a cut* $(S, \bar{S})$ is defined as the sum of the capacities of cut arcs going from S to

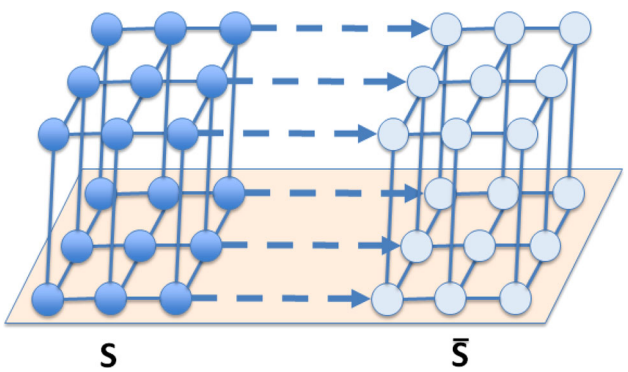


Fig. 1 A schematic illustration of a cut. The cut partitions the node set V into two disjoint sets: S and $\bar{S}$. The capacity $C(S, \bar{S})$ of the cut is the sum of the weights of the arcs that cross the cut, marked with a *dashed line* here

$\bar{S}, C(S,\bar{S}) = \sum_{i\in S, j\in\bar{S}} u_{ij}$. In the illustration given in Fig. 1 these arcs are represented in dashed lines.

Consider a graph that contains two specified nodes—a source node, s, and a sink node, t. This s, t-graph is denoted by $G_{st} = (V_{st}, A_{st})$, where $V_{st} = V \cup \{s, t\}$ and $A_{st} = A \cup A_s \cup A_t$ in which A_s and A_t are the source-adjacent and sink-adjacent arcs, respectively. The *minimum s, t cut* problem, referred to here as *min-cut*, is defined on G_{st}, and it is to find a bipartition of the nodes V_{st}—one set containing the source, s, and the other set containing the sink, t—such that the sum of capacities of the respective cut arcs, from the source set to the sink set, is minimized.

1.2.2 Flows

The *maximum-flow*, or *max-flow*, problem is defined on G_{st}, a directed capacitated graph. The problem is to find a feasible flow vector $f = \{f_{ij}\}_{(i,j)\in A_{st}}$ that maximizes the amount of flow that can be sent from the source to the sink while satisfying the following constraints:

1. Flow balance constraints: for each $j \in V$, $\sum_{(i,j)\in A_{st}} f_{ij} = \sum_{(j,k)\in A_{st}} f_{jk}$ [i.e., inflow(j) = outflow(j)], and
2. Capacity constraints: the flow value is between the lower bound and upper bound capacity of the arc, i.e., $\ell_{ij} \le f_{ij} \le u_{ij}$. We denote the vector of the upper bounds capacities of G_{st} by $\mathbf{u} = \{u_{ij}\}_{(i,j)\in A_{st}}$.

Without loss of generality one can assume that $\ell_{ij} = 0$. If not—one can apply a simple transformation setting $f'_{ij} = f_{ij} - \ell_{ij}$. The lower bound capacities in the transformed network, ℓ'_{ij}, become zero, while the objective function value changes by a constant that can be ignore when solving the problem (for further details see [2] (Section 2.4, p. 39).

The *value of the flow*, f is $\sum_{(i,j)\in A_{st}} f_{ij}$ and is denoted by $|f|$.

In 1956, Ford and Fulkerson [18] established the *max-flow min-cut theorem*, which states that the value of a max-flow in any network is equal to the value of a min-cut.

Given a capacity-feasible flow, hence a flow that satisfies (ii), an arc (i, j) is said to be a *residual arc* if $(i,j) \in A_{st}$ and $f_{ij} < u_{ij}$ or $(j,i) \in A_{st}$ and $f_{ji} > 0$. For $(i,j) \in A_{st}$, the residual capacity of arc (i, j) with respect to the flow f is $u^f_{ij} = u_{ij} - f_{ij}$, and the residual capacity of the reverse arc (j, i) is $u^f_{ji} = f_{ij}$. Let A^f denote the set of residual arcs with respect to flow f in G_{st} which consists of all arcs or reverse arcs with positive residual capacity.

A *preflow* is a relaxation of a flow that satisfies capacity constraints, but inflow into a node is allowed to exceed the outflow. The *excess* of a node $v \in V$ is the inflow into that node minus the outflow denoted by $e(v) = \sum_{(u,v)\in A_{st}} f_{uv} - \sum_{(v,w)\in A_{st}} f_{vw}$. Thus a preflow must have nonnegative excess.

A *pseudoflow* is a flow vector that satisfies capacity constraints, but may violate flow balance in either direction (inflow into a node needs not to be equal outflow). A negative excess is called a *deficit*.

2 Min-cut/max-flow algorithms

2.1 Hochbaum's pseudoflow algorithm

Hochbaum's pseudoflow algorithm (HPF), has a complexity of $O(nm \log \frac{n^2}{m})$ [29]. The algorithm was shown to be fast in theory [25] and in practice [12] for general benchmark problems. The following definitions are used in the description of HPF algorithm:

Current arc The set of current arcs in the graph, $\{\text{current arc}(w, v)\} \in A^f$, satisfies the following at the beginning of each iteration of the algorithm:

Property 1 ([12, 25])

(a) *The graph does not contain a cycle of current arcs.*
(b) *If* $e(v) \neq 0$, *then node* v *does not have a current arc.*

Root node A root node is defined recursively as follows: starting with node v, generate the sequence of nodes $\{v, v_1, v_2, \ldots, v_r\}$ defined by the current arcs $(v_1, v), (v_2, v_1), \ldots, (v_r, v_{r-1})$ until v_r has no current arc. Such root node v_r always exists (otherwise a cycle is formed, violating Property 1(a) [12, 25, 29]). Let the unique root of node v be denoted by root(v). Note that if node v has no current arc, then root(v) = v.

Distance label The algorithm associates each node $v \in V$ with a distance label $d(v)$ with the following property:

Property 2 ([12, 25]) *The node labels satisfy*:

(a) *For every arc* $(w, v) \in A^f, d(w) \le d(v) + 1$.
(b) *For every node* $v \in V$ *with strictly positive deficit*, $d(v) = 0$.

The above two properties imply that $d(v)$ is a lower bound on the distance (in terms of number of arcs) in the residual network of node v from a node with strict deficit.

Admissible arc A residual arc (w, v) is said to be *admissible* if $d(w) = d(v) + 1$.

Admissible path Given an admissible arc (w, v) with nodes w and v in different components, an *admissible path* is the path from root(w) to root(v) along the set of arcs from root(w) to w, the arc (w, v), and the set of arcs (in the reverse direction) from v to root(v).

Active node A node is said to be *active* if it has strictly positive excess.

HPF algorithm is initiated with any arbitrary initial *pseudoflow* (i.e., flow vector that may violate flow balance in either direction). Such initial pseudoflow can be

generated, for example, by saturating all source-adjacent and sink-adjacent arcs, $A_s \cup A_t$, and setting all other arcs to have zero flow. This creates a set of source-adjacent nodes with excess, and a set of sink-adjacent nodes with deficit. All other arcs have zero flow. This particular pseudoflow is called *simple initialization*. The procedure generating the simple initialization pseudoflow is called SimpleInit.

An iteration of HPF algorithm consists of choosing an active component, with root node label $<n$ and searching for an admissible arc from a *lowest labeled* node w in this component. Choosing a lowest labeled node for processing ensures that an admissible arc is never between two nodes of the same component.

By construction (see [25]), the root is the lowest labeled node in a component and node labels are non-decreasing with their distance from the root of the component. Thus, all the lowest labeled nodes within a component form a sub-tree rooted at the root of the component. Once an active component is identified, all the lowest labeled nodes within the component are examined for admissible arcs by performing a depth-first-search in the sub-tree starting at the root.

If an admissible arc (w, v) is found, a *merger* operation is performed. The merger operation consists of pushing the entire excess of root(w) towards root(v) along the admissible path and updating the excesses and the arcs in the current forest. The pseudocode of the merger operation is given in Fig. 2.

If no admissible arc is found, $d(w)$ is increased by 1 unit for all lowest label nodes w in the component.

The algorithm's first phase terminates when there are no active nodes with label $<n$. At termination all n labeled nodes form the source set of the min-cut.

The active component to be processed in each iteration can be selected arbitrarily. There are two variants of HPF: (1) lowest label HPF, where an active component with the lowest labeled root is processed at each iteration; and (2) highest label HPF, where an active component with the highest labeled root node is processed at each iteration. The highest label HPF algorithm was found to be most efficient and performs competitively with push-relabel on many general problem instances [12].

The first phase of HPF terminates with the min-cut and a pseudoflow. The second phase stage consists of flow recovery, converting this pseudoflow to a maximum feasible flow. This is done using *flow decomposition* in $O(m \log n)$ running time [25]. Our experiments, like the experiments in [12], indicate that the time spent in flow recovery is small compared to the time required to find the min-cut stage.

2.2 The push-relabel algorithm

The complexity of the push-relabel (PRF) algorithm is $O\left(nm \log \frac{n^2}{m}\right)$, using the dynamic trees data structure of Sleator and Tarjan [46]. In this section, we provide a sketch of a straightforward implementation of the algorithm. For a more detailed description, see [2, 21].

Goldberg's and Tarjan's [21] push-relabel algorithm, PRF, works with *preflows*, where a node with strictly positive excess is said to be *active*. Each node i is assigned a label $\ell(i)$ that satisfies (1) $\ell(t) = 0$, and (2) $\ell(i) \leq \ell(j) + 1$ for all $(i,j) \in A^f$. A residual arc $(i,j) \in A^f$ is said to be *admissible* if $\ell(i) = \ell(j) + 1$.

Initially, the source is assigned the label n, and all other nodes are assigned the label 0. Since all source-adjacent arcs are saturated, the set of source-adjacent nodes are all active (all other nodes have zero or negative excess). An

Fig. 2 The min-cut stage of HPF algorithm. At termination all nodes in label-n components are the source set of the min-cut

```
/*
Min-cut stage of HPF algorithm.
*/

procedure HPF (V_st, A_st, u):
    begin
        SimpleInit (A_s, A_t, u);
        while ∃ an active component T with root r, where d(r) < n, do
            w ← r;
            while w ≠ ∅ do
                if ∃ admissible arc (w, v) do
                    Merger (root(w), ··· , w, v, ··· , root(v));
                    w ← ∅;
                else do
                    if ∃y ∈ T : (current(y) = w) ∧ (d(y) = d(w)) do
                        w ← y;
                    else do {relabel}
                        d(w) ← d(w) + 1;
                        w ← current(w);
    end
```

iteration of the algorithm consists of selecting an active node in V, and attempting to push its excess to its neighbors along admissible arcs. If no such arc exists, the node's label is increased by 1. The algorithm terminates with a maximum preflow when there are no active nodes with label less than n. The set of nodes of label n then forms the source set of a min-cut and the current preflow is maximum in that it sends as much flow into the sink node as possible. This ends Phase 1 of the algorithm. In Phase 2, the algorithm transforms the maximum preflow into a maximum flow by pushing the excess back to s. The flow recovery consists of transforming the maximum preflow into a maximum feasible flow by pushing the excess back to s. Similarly to HPF, the flow recovery is considerably faster than computing the min-cut. A high-level description of PRF algorithm is shown in Fig. 3.

In the highest label and lowest label variants of the algorithm, an active node with highest and lowest labels, respectively, are selected for processing at each iteration. In the FIFO variant, the active nodes are maintained as a queue in which nodes are added to the queue from the rear and removed from the front for processing. The highest level variant of PRF was found to have the best performance in practice (see [2], p. 242, [13]). This variant of the algorithm is also referred to in the literature as HI_PR. In this paper, the highest label variant is used. Additionally, two heuristics are employed in practice and significantly improve the run-time of the algorithm: *Gap relabeling* and *Global relabeling* (see [12, 21] for details).

2.3 Boykov's and Kolmogorov's augmenting paths algorithm

For a maximum-flow vector f^* the max-flow value is denoted by $|f^*| = \sum_{(s,j)\in A_s} f^*_{sj}$. The theoretical complexity of Boykov's and Kolmogorov's (BK) algorithm is given by $O(mn^2|f^*|)$. Note that this complexity is *pseudo-polynomial*, in contrast to all other algorithms discussed here that have strongly polynomial complexity. At heart of the BK's approach is the use of search trees for detecting augmenting paths from s to t. Two such trees, one from the source, T_s, and the other from the sink, T_t are constructed, where $T_s \cap T_t = \varnothing$. The trees are constructed so that in T_s all edges from each parent node to its children are non-saturated and in T_t, edges from children to their parents are non-saturated.

Nodes that are not associated with a tree are called *free*. Nodes that are not free can be tagged as *active* or *passive*. Active nodes have edges to at least one free node, while passive nodes have no edges connecting them to a free node. Thus, trees can grow only by connecting, through a non-saturated edge, a free node to an active node of the tree. An augmenting path is found when an active node in either of the trees has a neighboring node found in the other tree.

At the initialization stage the search tree, T_s contains only the source node, s and the search tree T_t contains only the sink node t. All other nodes are free.

Each iteration of the algorithm consists of the following three stages:

Growth In this stage the search trees T_s and T_t expand. For all active nodes in a tree, T_s or T_t, adjacent free nodes, which are connected through non-saturated edge, are searched. These free nodes become members of the corresponding search tree. The growth stage terminates when the search for an active node from one tree, finds an adjacent (active) node that belongs to the other tree. Thus, an augmenting path from S to T was found.

Augmentation Upon finding the augmenting path, the maximum flow possible is being pushed from s to t. This implies that at least one edge will be saturated. Thus, for at least one node in the trees T_s and T_t the edge connecting it to its parent is no longer valid. The augmentation phase may split the search trees T_s and T_t into forests. Nodes for which the edges connecting them to their parent become saturated are called *orphans*.

```
/*
Generic push-relabel algorithm for maximum flow.
*/
procedure push-relabel(V_st, A_st, u)
    begin
        Set the label of s to n and that of all other nodes to 0;
        Saturate all arcs in A_s;
        while there exists an active node i ∈ V of label less than n do
            if there exists an admissible arc (i, j) do
                Push a flow of min{e(i), u_ij^f} along arc (i, j);
            else do
                Increase label of i by 1 unit;
    end
```

Fig. 3 High-level description of Phase I of the generic push-relabel algorithm. The nodes with label equal to n at termination form the source set of the min-cut

Adoption In this stage the tree structure of T_s and T_t is restored. For each orphan, created in the previous stage, the algorithm tries to find a new valid parent. The new parent should belong to the same set, T_s or T_t, as the orphan node and has a non-saturated edge to the orphan node. If no parent is found, then the orphan node and all its children become free and the tree structure rooted in this orphan is discarded. This stage terminates when all orphan nodes are connected to a new parent or are free.

The algorithm terminates when there are no more active nodes and the trees are separated by saturated edges. Thus, the max-flow is achieved and the corresponding min-cut is $S = T_s$ and $T = T_t$.

2.4 The partial augment-relabel

The Partial Augment-Relabel algorithm, PAR, devised by Goldberg [19], searches for the shortest augmenting path. PAR is distinguished from PRF in that PAR consist of pushing flows from active nodes along paths of length ℓ. These paths do not necessarily end at node t. PAR was shown to have a complexity of $O(n^2\sqrt{m})$.

At each iteration during stage ℓ there is a DFS search, from an active node, for an admissible path of length ℓ. Here, an arc (i, j) is admissible if the label of its associated nodes is equal, $d(i) = d(j)$. For a partial path, of length $<\ell$, from s to $i \in V$, if i has an admissible arc, (i, j), and j has not been visited as of yet, the path is extended to j. If no such admissible arc is found, the algorithm shrinks the path, making the predecessor of i on the path the current node and increasing its label, $d(i)$, by 1. At each iteration, the search terminates either if $j = t$, or if the length of the path reaches some predefined value, ℓ, or if i, the current node has no outgoing admissible arcs.

In order to achieve better performance in practice, the same gap and global heuristics mentioned in Sect. 2.2, for PRF, can be applied here for the PAR algorithm.

3 Experimental setup

PRF, HPF and BK algorithms are compared here by running them on the same problem instances and on the same hardware setup. The run-times of the highest level variant of PRF algorithm and of PAR are reported in [19] for a subset of the problems used here. Since the source code for the PAR implementation has not been made available, PAR's performance is evaluated here by normalizing the run-times of the highest level variant of PRF by the speedup factor of PAR compared to PRF reported in [19].

As suggested by Chandran and Hochbaum [12], we use the highest label version for HPF algorithm. The latest version of the code (version 3.23) is accessible at [28]. We use the highest-level PRF implementation Version 3.5, [20] since it was shown to be the best performance PRF variant [13]. Note that the latest implementation of the Push-Relabel method is denoted, in other papers, by HI_PR. We refer to this variant here as PRF, and it is the same algorithm which was reported in [13]. For BK algorithm, a library implementation was used [34]. In order to utilize the library for solving problems in DIMACS format, a wrapping code, wrapper, was written. This wrapper reads the DIMACS files and calls the library's functions for constructing and solving the problem. The part that reads the DIMACS files, under the required changes, is similar to the code used in the implementation of HPF. One should note that the compilation's setup and configuration of the library have great effect on the actual running times of the code. In our tests the shortest running times were achieved using the following compilation line `g++ -w -O4 -o <output_file_name> -DNDEBUG -DBENCHMARK graph.cpp maxflow.cpp <wrapper_implementation_file>`.

The run-times are reported here for the three different stages of the algorithm: initialization, min-cut and max-flow. Every problem instance was run three times for each stage and the average run-time and the standard deviation of the three runs were computed. The standard deviation was then normalized by the average run-time for the respected stage of the problem. The average of the normalized standard deviation across all problem instances and stages was 0.4 %, and the maximum value was 2 %. Thus, the run-times are stable. As detailed in Sect. 1, breaking down the run-times provides insight into the algorithms' performance and allows for better comparison. Since for many computer-vision applications only the min-cut solution is of importance, the most relevant evaluation is of the sum of initialization and min-cut times.

3.1 Computing environments

Our experiments were conducted on a machine with x86_64 Dual-Core AMD Opteron(tm) Processor at 2.4 GHz with 1024 KB level 2 cache and 32 GB RAM. The operating system used was GNU/Linux kernel release 2.6.18-53.el. The code of all three algorithms, PRF, HPF and BK, was compiled with gcc 4.1.2 with $-O4$ optimization flag.

One should note that the relatively large physical memory of the machine allows one to avoid memory swaps between the memory and the swap-file (on the disk) throughout the execution of the algorithms. Swaps are important to avoid since when the machine's physical memory is small with respect to the problem's size, the memory swap operation might take place very often. These swapping times, the wait times for the swap to take place,

can accumulate to a considerably long run-times. Thus, in these cases, the execution times are biased due to memory constraints, rather than measuring the algorithms' true computational efficiency. Therefore, we chose large physical memory which allows for more accurate and unbiased evaluation of the execution times.

3.2 Problem classes

The test sets used consist of problem instances that arise as min-cut problems in computer vision, graphics, and biomedical image analysis. All instances were made available from the Computer Vision Research Group at the University of Western Ontario [14]. The problem sets used are classified into four types of vision tasks: stereo vision, segmentation, multi-view reconstruction, and surface fitting. These are detailed in Sects. 3.2.1–3.2.4. The number of nodes n and the number of arcs m for each of the problems are given in Appendix 1, Tables 6, 7, 8, 9.

3.2.1 Stereo vision

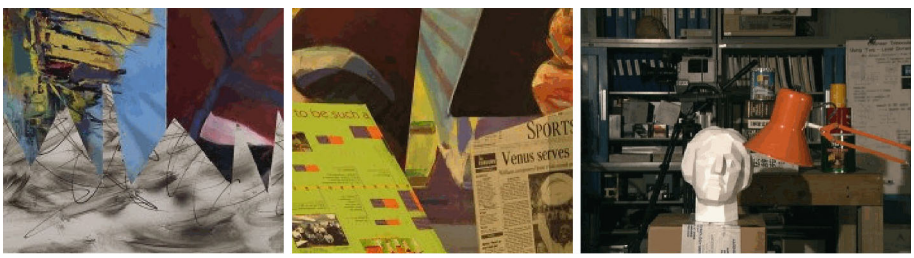
Stereo problems, as one of the classical vision problems, have been extensively studied. The goal of stereo is to compute the correspondence between pixels of two or more images of the same scene. We use the *Venus*, *Sawtooth* [42] and the *Tsukuba* [39] data sets. Each of the stereo problems, used in this study, consists of an image sequence, where each image in the sequence is a slightly shifted version of its preceding one. The Venus sequence consists of 22 images, Sawtooth of 20 and the Tsukuba sequence has 16 images. All images have the same size of 512×384 pixels. A corresponding frame for each sequence is given in Fig. 4. The segmentation problems' sizes are given in Table 6.

Often the best correspondence between the pixels of the input images is determined by solving a min-cut problem for each pair of images in the set. Thus in order to solve the stereo problem, one has to solve a sequence of min-cut subproblems all of approximately the same size. Previously reported run-times of these stereo problem [10, 19] disclosed, for each problem, only the summation of the run-times of its min-cut sub-problems. Presenting the summation of the run-times of the sub-problems as the time for solving the entire problem assumes linear asymptotic behavior of the run-times with respect to the input size. This assumption has not been justified. The run-times here, for the stereo problems, are reported as the *average* time it takes the algorithm to solve the min-cut sub-problem.

Each of the stereo min-cut sub-problems aims at matching corresponding pixels in two images. The graphs consist of two 4-neighborhood grids, one for each image. Each node, on every grid, has arcs connecting it to a set of nodes on the other grid. For each of the stereo problems there are two types of instances. In one type, indicated by KZ2 suffix, each node in one image is connected to at most two nodes in the other image. In the second type, indicated by BVZ suffix, each node in one image is connected to up to five nodes in the second image.

3.2.2 Multi-view reconstruction

A 3D reconstruction is a fundamental problem in computer vision with a significant number of applications (for recent examples see [31, 45, 47]). Specifically, graph theory-based algorithms for this problem were reported in [36, 48, 53]. The input for the multi-view reconstruction problem is a set of 2D images of the same scene taken from different perspectives. The reconstruction problem is to construct a 3D image by mapping pixels from the 2D images to voxels complex in the 3D space. The most intuitive example for such a complex would be a rectangular grid, in which the space is divided into cubes. In the examples used here a finer grid, where each voxel (neighborhood, NBH) is divided into 24 tetrahedral by six planes each passing through a pair of opposite cube edges, is used (see [36] for details). Two sequences are used in this class, *Camel* and *Gargoyle*. Each sequence was constructed in three different sizes (referred to as small, middle and large) [11]. The sizes here are of the inferred 3D grid size, thus the output resolution. Representing frames are presented in Fig. 5. Problems' images and the grid sizes are detailed in Table 7.



Sawtooth *Venus* *Tsukuba*

Fig. 4 Stereo test sequences (Source [14])

Camel *Gargoyle*

Fig. 5 Multi-view test sequences (source [14])

3.2.3 Surface fitting

3D reconstruction of an object's surface from sparse points containing noise, outliers, and gaps is also one of the most interesting problems in computer vision. The input is a set of point cloud in 3D space and the output is a smooth manifold which is close as possible to each of the points while imposing some shape priors either on the volume or the surface, such as spatial occupancy or surface area [35].

Under this class we present a single test instance, *"Bunny"* (see Fig. 6), constructed in three different sizes (see Table 8) with six 3D connectivity scheme. Similarly to the multi-view-reconstruction problem set, size here refers to the size of the reconstructed 3D graph, hence the output 3D resolution. The sequence is part of the Stanford Computer Graphics Laboratory 3D Scanning Repository [49] and consists of 362,272 scanned points. The "bunny" corresponding graphs, on which the min-cut problem is solved, are characterized by particularly short paths from s to t [35].

3.2.4 Segmentation

Under this group a set of four 3D test cases, referred to as liver, adhead, babyface and bone are used. Each set consists of similar instances which differ in the graph size, neighborhood size, length of the path between s and t, regional arc consistency (noise), and arc capacity magnitude [9]. Liver problem instances have short s, t paths (32–400 arcs), while adhead are characterized by long s, t paths (several millions arcs). Babyface has noisy arcs capacities, while bone presents a classic MRI image. For all instances used in this group, the suffixes n and c represent the neighborhood type and maximum arc capacities, respectively. For example, *bone.n6.c10* and *babyface.n26.c100*, correspond to a 6 3D-neighborhood and a maximum arc capacity of 10 U and a 26 3D-neighborhood with maximum arc capacity of 100 U, respectively. The different *bone* instances differ in the number of nodes. The grid on the 3 axes x, y and z was made coarser by a factor of 2 on each, thus bone.*xy*, means that the original problem (bone) was decimated along the x, y axes and it is 1/4 of its original size; bone.*xyz*, means that the original problem was decimated along the x, y and z axes and it is 1/8 of its original size. The Segmentation problems' sizes are listed in Table 9.

OriginalImage *ReconstructedSurface*

Fig. 6 Bunny problem instance—surface fitting (Source[49])

4 Results

4.1 Run-times

In this study, the comparison of PRF's, HPF's and BK's run-times are indicated for the three stages of the algorithms: (1) initialization, t_{init}; (2) min-cut, t_{minCut}; and (3) max-flow, t_{maxflow}. As these data are unknown for PAR, the comparison of these three algorithms with respect to PAR is addressed differently, by running PRF on our setup and deducing PAR run-times by multiplying the measured PRF time by the speedup factor reported in [19]. This is explained in Sect. 4.2.

The relevant times for most computer-vision problems are the times it takes each of the algorithms to complete the computation of the min-cut, thus $t_{\text{init}} + t_{\text{minCut}}$. These are graphically presented in Fig. 7 and detailed in Tables 1, 2, 3, 4. Note that the bar chart figures give the run-times for multi-view, surface fitting and segmentation in logarithmic scale. The *Slowdown Factor*, reported in these tables for each algorithm, for every problem instance, is the ratio of the time it takes the algorithm to complete the computation of the min-cut divided by the minimum time it took any of the algorithms to complete this computation.

Figure 7a (Table 1) presents the run-times for the stereo vision problem sets. The input's size, for these problems is small, with respect the the other problem sets. For these small problem instances, BK algorithm does better than PRF (with average speedup factor of 2.86, which corresponds to average difference in the running time of 2.0 s)

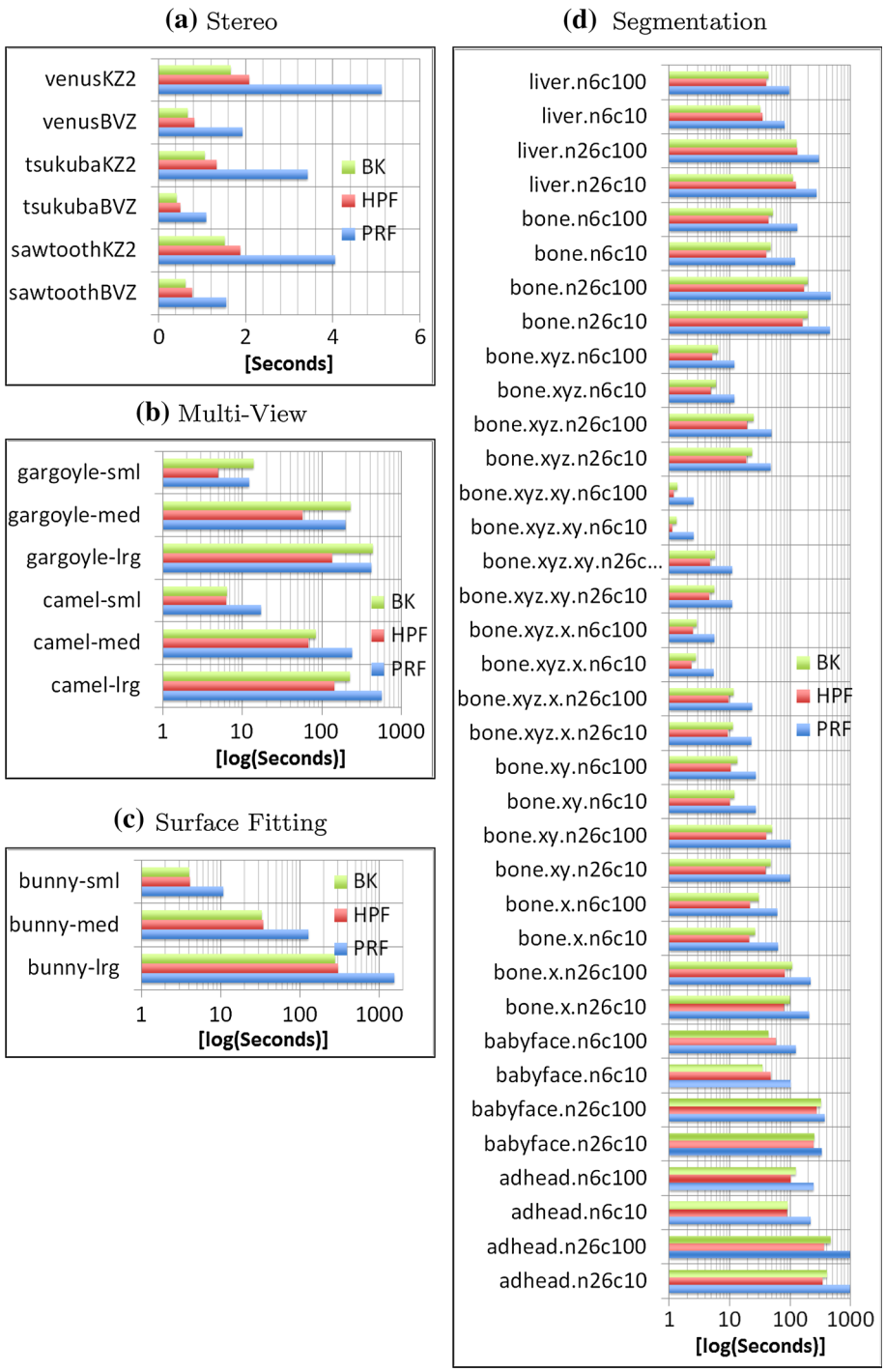


Fig. 7 Initialization and min-cut run-times in seconds: **a** stereo problems, **b** multi-view problems, **c** surface fitting, **d** segmentation

and slightly better than HPF (speedup factor of 1.24, which corresponds to a running time difference of 0.24 s). For the multi-view instances, presented in Fig. 7b (and Table 2) HPF is faster than both PRF and BK with average speedup factors of 1.46 with respect to BK and 3.19 with respect to PRF. These correspond to differences in the running times of 95 and 170 s, respectively. Figure 7c and Table 3 show the run-times for the surface fitting instances. This demonstrates that BK is faster for these instances due to particularly short s, t paths (of 32–400 arcs) that characterize these instances. In these instances, the slowdown factors of HPF and PRF are 1.05 (correspond to an average difference of 9 s) and 4.06 (454 s), respectively. The running times for the segmentation problems class are depicted in Fig. 7d and Table 4. There are 36 segmentation problems. In a subset of 5 segmentation problems BK

Table 1 Stereo problems: combined Initialization and min-cut run-times and their corresponding speedup factors

Stereo						
Instance	Run-times (S)			Slowdown factor		
	PRF	HPF	BK	PRF	HPF	BK
sawtoothBVZ	1.55	0.76	**0.62**	2.5	1.23	**1**
sawtoothKZ2	4.05	1.87	**1.52**	2.66	1.23	**1**
tsukubaBVZ	1.09	0.5	**0.41**	2.66	1.22	**1**
tsukubaKZ2	3.42	1.33	**1.06**	3.23	1.25	**1**
venusBVZ	1.93	0.82	**0.66**	2.92	1.24	**1**
venusKZ2	5.12	2.09	**1.66**	3.08	1.26	**1**

Each problem's fastest run-time is set in boldface. The speedup factor states how much an algorithm runs compared to the fastest algorithm

Table 2 Multi-view problems: combined initialization and min-cut run-times and their corresponding speedup factors

Multi-view						
Instance	Run-times (s)			Slowdown factor		
	PRF	HPF	BK	PRF	HPF	BK
camel-lrg	558.65	**143.96**	225.53	3.88	**1**	1.57
camel-med	240.3	**67.61**	83.43	3.55	**1**	1.23
camel-sml	17.16	**6.31**	6.46	2.72	**1**	1.02
gargoyle-lrg	413.26	**134.77**	432.31	3.07	**1**	3.21
gargoyle-med	196.26	**56.24**	226.43	3.49	**1**	4.03
gargoyle-sml	12.26	**5**	13.86	2.45	**1**	2.77

Table 3 Surface fitting problems: initialization and min-cut run-times and their corresponding speedup factors

Surface fitting						
Instance	Run-times (s)			Slowdown factor		
	PRF	HPF	BK	PRF	HPF	BK
bunny-lrg	1564.1	305.05	**277.02**	5.65	1.1	**1**
bunny-med	129.05	34.47	**32.82**	3.93	1.05	**1**
bunny-sml	10.89	4.12	**4.03**	2.7	1.02	**1**

achieves faster running times. These instances are characterized by short s, t paths [9, 10]. In this subset, BK's average speedup factors are 1.19 (9.24 s difference) and 2.62 (106 s difference in the running time) with respect to HPF and PRF, respectively. For the remainder of the 31 segmentation problems, HPF is the fastest, with speedup factors of 1.18 (14.22 s difference) compared to BK and 2.62 (101.39 s difference) compared to PRF.

As discussed in Sect. 1, the initialization run-times of PRF are considerably longer than those or either BK or HPF algorithms. The bar chart summarizing the

Table 4 Segmentation problems: initialization and min-cut run-times and their corresponding speedup factors

Segmentation						
Instance	Run-times (s)			Slowdown factor		
	PRF	HPF	BK	PRF	HPF	BK
adhead.n26c10	970.74	**344.31**	407.42	2.82	**1**	1.18
adhead.n26c100	971.2	**362.49**	476.04	2.68	**1**	1.31
adhead.n6c10	219.14	**90.17**	90.38	2.43	**1**	1
adhead.n6c100	242.43	**103.03**	123.22	2.35	**1**	1.2
babyface.n26c10	333.91	**245.1**	250.15	1.36	**1**	1.02
babyface.n26c100	378.94	**272.26**	321.2	1.39	**1**	1.18
babyface.n6c10	103.27	48.3	**35.3**	2.93	1.37	**1**
babyface.n6c100	126.28	58.77	**43.89**	2.88	1.34	**1**
bone.n26c10	451.35	**160.73**	196.26	2.81	**1**	1.22
bone.n26c100	472.34	**168.09**	198.73	2.81	**1**	1.18
bone.n6c10	119.76	**41.02**	47.79	2.92	**1**	1.17
bone.n6c100	132.06	**43.56**	51.88	3.03	**1**	1.19
bone.x.n26c10	209.03	**79.7**	100.23	2.62	**1**	1.26
bone.x.n26c100	218.57	**81.79**	107.43	2.67	**1**	1.31
bone.x.n6c10	64.17	**20.89**	26.57	3.07	**1**	1.27
bone.x.n6c100	61.18	**22.06**	30.29	2.77	**1**	1.37
bone.xy.n26c10	99.59	**39.51**	48.25	2.52	**1**	1.22
bone.xy.n26c100	101.13	**40**	51.04	2.53	**1**	1.28
bone.xy.n6c10	27.22	**10.04**	12.09	2.71	**1**	1.2
bone.xy.n6c100	27.66	**10.56**	13.62	2.62	**1**	1.29
bone.xyz.n26c10	48.07	**18.89**	23.69	2.54	**1**	1.25
bone.xyz.n26c100	48.83	**19.39**	25.41	2.52	**1**	1.31
bone.xyz.n6c10	11.95	**4.85**	5.95	2.46	**1**	1.23
bone.xyz.n6c100	12.05	**5.14**	6.54	2.34	**1**	1.27
bone.xyz.x.n26c10	22.96	**9.36**	11.27	2.45	**1**	1.2
bone.xyz.x.n26c100	23.55	**9.52**	11.55	2.47	**1**	1.21
bone.xyz.x.n6c10	5.47	**2.37**	2.75	2.31	**1**	1.16
bone.xyz.x.n6c100	5.56	**2.47**	2.89	2.25	**1**	1.17
bone.xyz.xy.n26c10	10.99	**4.63**	5.6	2.37	**1**	1.21
bone.xyz.xy.n26c100	11.06	**4.74**	5.79	2.33	**1**	1.22
bone.xyz.xy.n6c10	2.55	**1.14**	1.34	2.24	**1**	1.18
liver.n26c10	272.63	123.61	**112.4**	2.43	1.1	**1**
liver.n26c100	297.88	132.48	**128.6**	2.32	1.03	**1**
liver.n6c10	82.11	35.24	**32.02**	2.56	1.1	**1**
liver.n6c100	96.38	**40.36**	43.6	2.39	**1**	1.08

initialization times comparison is given in Fig. 8 and detailed in Appendix 2, Tables 10, 11, 12, 13. Figure 8 shows that for all problem instances, PRF's initialization times (t_{init}) are 2–3 times longer than BK's and HPF's respective initialization times. Although these times were excluded from the total execution times reported in [10] and [19], Fig. 8 strongly suggests that these initialization times are significant with respect to the min-cut computation times (t_{minCut}) and must not be disregarded.

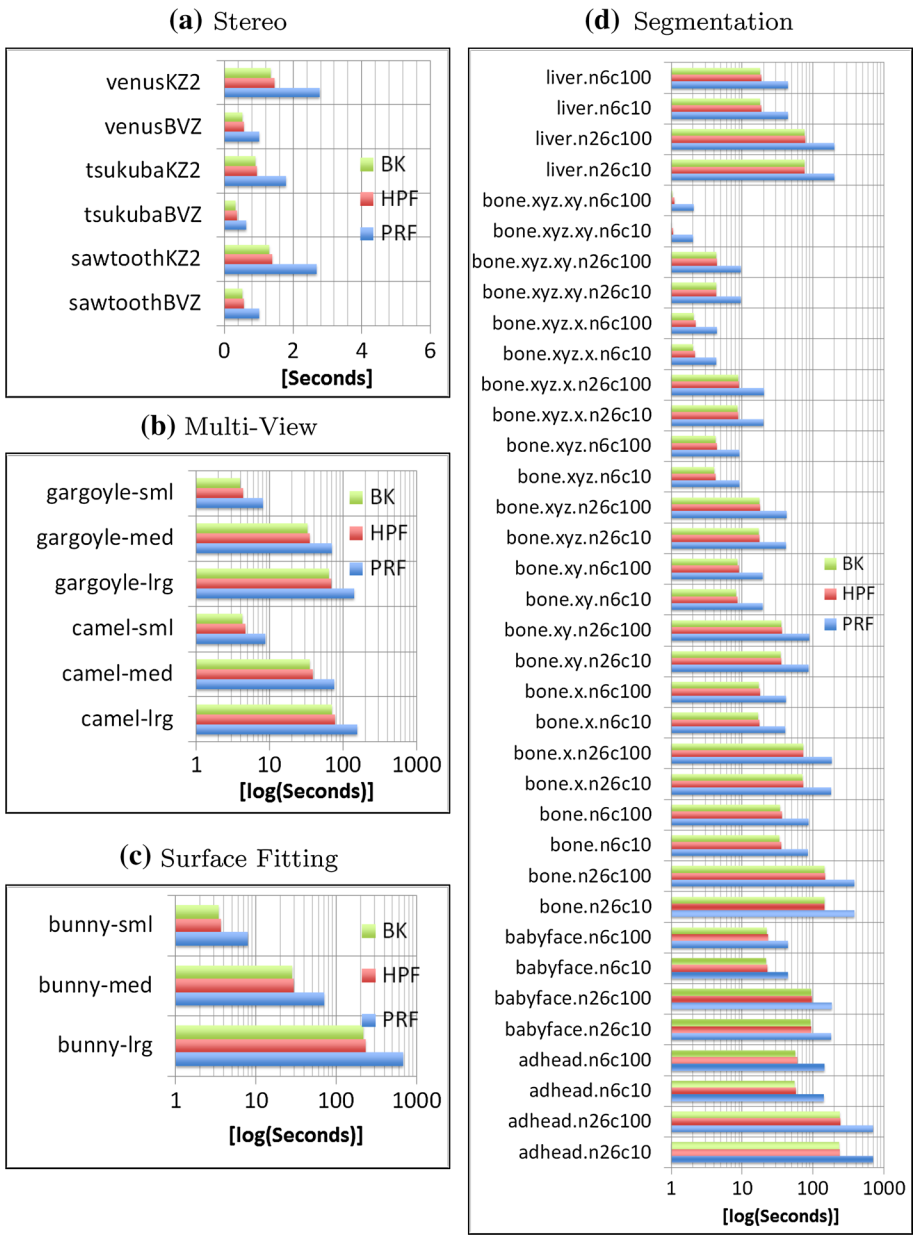


Fig. 8 Initialization run-times in seconds: **a** stereo problems; **b** multi-view problems; **c** surface fitting; **d** segmentation

The solution to the max-flow problem is of lesser significance in solving computer-vision problems. Yet, for the sake of completeness, the max-flow computation times of the algorithms ($t_{\text{init}} + t_{\text{minCut}} + t_{\text{maxFlow}}$) are reported in Fig. 9 and in Tables 14, 15, 16,17.

4.2 Comparison to partial augment-relabel

The PAR run-times, on our hardware setup, are deduced from the average speedup factor of PAR compared to the highest level variant of PRF for each problem instance reported in [19]. That paper reported only the summation of the min-cut and max-flow run-times, $t_{\text{minCut}} + t_{\text{maxFlow}}$, omitting the initialization times. Our results, given next, demonstrate that even when omitting the initialization times, both PRF and the faster version PAR, still have worse performance than HPF.

Let $t^G_{\text{PAR}}(I)$ and $t^G_{\text{PRF}}(I)$ be the run-times reported in [19] for PAR and PRF algorithms, respectively, for instance I. The estimated run-times of PAR in instance (I), $\hat{t}_{PAR}(I)$, on our hardware is:

$$\hat{t}_{\text{PAR}}(I) = \frac{t^G_{\text{PAR}}(I)}{t^G_{\text{PRF}}(I)} \left(t^{\text{PRF}}_{\text{minCut}}(I) + t^{\text{PRF}}_{\text{maxFlow}}(I)\right)$$

where $t^{\text{PRF}}_{\text{minCut}}(I)$ and $t^{\text{PRF}}_{\text{maxFlow}}(I)$ are the corresponding run-times of PRF algorithm measured on the hardware used in this study.

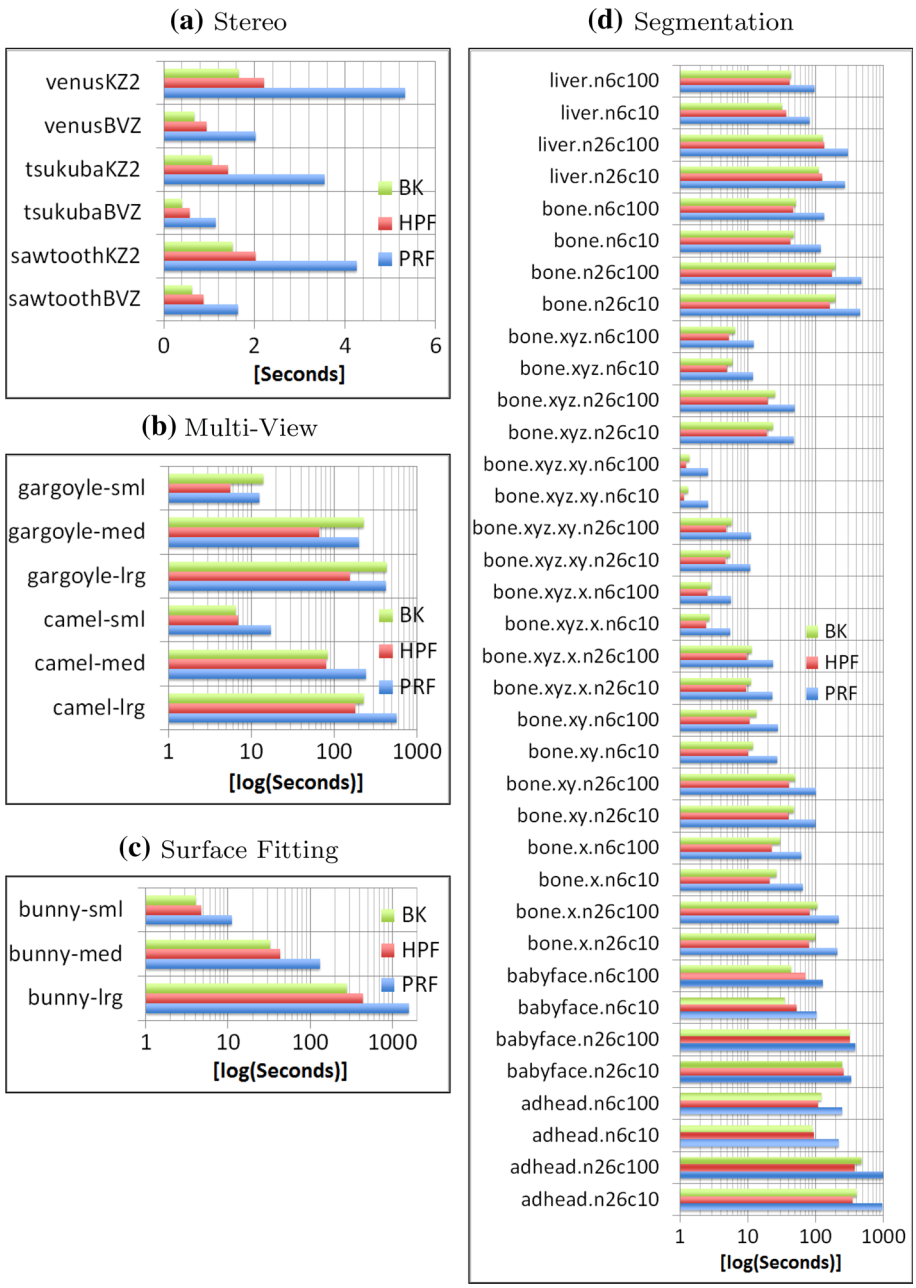


Fig. 9 Initialization, min-cut and max-flow run-times in seconds: **a** stereo problems; **b** multi-view problems; **c** surface fitting; **d** segmentation

The comparison results for the sum of the min-cut and max-flow run-times given in Fig. 10, for all problem instances reported in [19], demonstrate that HPF outperforms PAR for all problem instances. It is noted that although this comparison excludes the initialization times, PAR's performance is still inferior to that of HPF. If one were to add the initialization times (see Tables 10, 11, 12, 13), then the relative performance of PAR as compared to HPF would be much worse since PRF's initialization times are significantly longer than HPF's as shown in Fig. 8. In terms of comparing PAR to BK, Fig. 10 shows that PAR is inferior to BK for small problem instances, but performs better for larger instances.

4.3 Memory utilization

Measuring the actual memory utilization of different algorithms is of growing importance, as advances in acquisition systems and sensors allow higher image resolution, thus larger problem sizes and more significant use of memory. The memory usage was read directly out of the/*proc/[process]/statm* file for each implementation and for

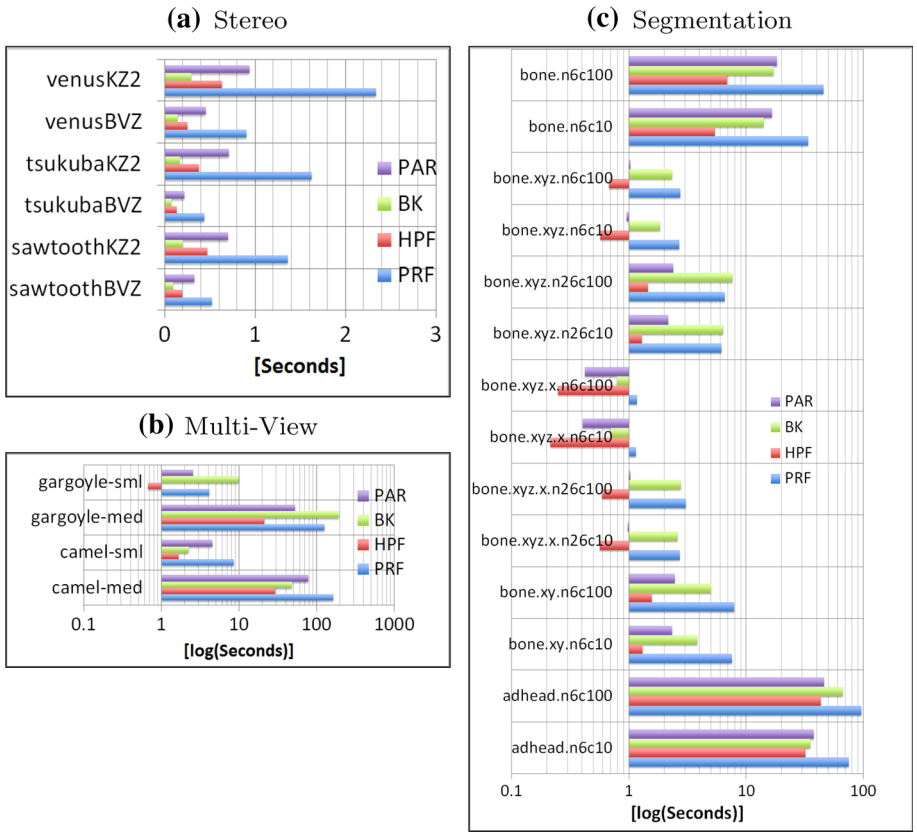


Fig. 10 Min-cut and max-flow run-times in seconds for PAR, PRF, HPF and BK: **a** stereo problems; **b** multi-view problems; **c** segmentation

Table 5 Executables memory utilization

Algorithm	Text	Data	Total
BK	13,549	708	14,305
PRF	13,963	620	14,903
HPF	9,433	600	10,145

each problem instance. The granularity of the readings is a page-file size, thus 4,096 Bytes, which is 0.005 % of the total memory consumption of the smallest problem. These readings include the actual machine instructions (*text section*), all initialized variables declared in the program (*data section*) and the dynamic data which is allocated only at runtime. The text and data sections are the same for all runs and are given in Table 5.

Figure 11 summarizes the results of the memory utilization for BK (green dashed line), HPF (dotted red) and PRF (blue solid) algorithms. These are detailed in "Appendix 3", Tables 18, 19, 20, 21. The *X* axis in Fig. 11 is the input size. A Problem's input size is the number of nodes, n plus the number of arcs, m, in the problem's corresponding graph: input size $= n + m$. The number of nodes, n, and the number of arcs, m, for each of the problems are given in Tables 6, 7, 8, 9. The *Y* axis is the memory utilization in mega-bytes. The graph also shows how memory utilization increases as a function of problem size.

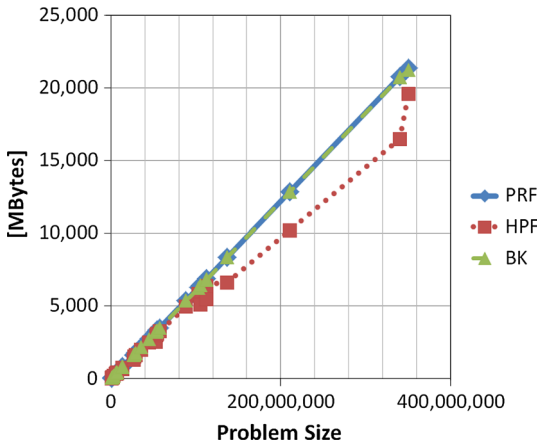


Fig. 11 Memory utilization vs. input size

Both BK and PRF algorithms use on average 10 % more memory than HPF algorithm. For problem instances with large number of arcs, PRF and BK require 25 % more memory. This becomes critical when the problem size is considerably large, with respect to the machine's physical memory. In these cases the execution of the algorithms requires a significant amount of swapping memory pages between the physical memory and the disk, resulting in longer execution times.

The differences in memory utilization between the algorithms may be attributed to two factors: (1) data

Table 6 Problems' sizes: stereo problems

Stereo				
Name	Image size	Frames	Nodes	Arcs
sawtoothBVZ	512 × 384	16	173,602	838,635
sawtoothKZ2	512 × 384	16	310,459	2,059,153
tsukubaBVZ	512 × 384	16	117,967	547,699
tsukubaKZ2	512 × 384	16	213,144	1,430,508
venusBVZ	512 × 384	16	174,139	833,168
venusKZ2	512 × 384	16	315,972	2,122,772

structures each algorithm maintains; and (2) dynamic memory allocation. BK's data structures are more complex and require more memory than HPF (see Table 5). For example, both HPF and BK algorithms, in conjunction with the graph topology information, must maintain a flow vector f. Additionally, BK maintains a list of active nodes and a list of all orphans (see Sect. 2.3). HPF algorithm, on the other hand, adds only three features (or fields) to each node ("is it a root node?" boolean flag, the node's excess/deficit and the node's current arc), which does not increase significantly the memory required to describe the graph topology. This in itself, however, is not sufficient to determine memory utilization as dynamic allocation must be evaluated empirically. It is important to note that neither factors grow linearly with the data.

5 Conclusions

This paper introduces Hochbaum's pseudoflow algorithm (HPF), that solves optimally the s, t min-cut and max-flow problems, to the field of computer vision. A comprehensive computational study is presented here, comparing HPF with the three leading algorithms: (1) Goldberg's and Tarjan's *Push-Relabel* (PRF); (2) Boykov's and Kolmogorov's *augmenting paths* (BK); and (3) Goldberg's *partial augment-relabel* (PAR).

A total of 51 computer-vision problem instances were tested within the scope of this study. In all problem instances both BK and HPF were faster than PAR and PRF. In 37 instances (out of the 51) HPF had the best running times. In the remaining 14 cases, HPF was slightly slower than BK. Six out of the 14 instances were small, with an average problem size for this group of 1.5 million graph elements, whereas the remaining problems had average problem sizes that were 50 times larger. The remaining 8 instances out of the 14 had shorter length s, t paths (consisting of up to 400 arcs) than other problem instance (which had s, t paths that are tens of thousands arcs long).

Indeed for problem instances with either small problem size or with extremely short s, t paths one may choose BK algorithm. For all other types of instances HPF should be the algorithm of choice. Furthermore, HPF algorithm's source code is readily available [28]. This makes HPF the perfect tool for the growing number of computer-vision applications that incorporate the min-cut problem as a subroutine. Our results are significant since they contrast the widely accepted belief that both BK and PRF algorithms were the fastest algorithms in practice for the min-cut problem. This was shown not to hold in general [12], and here for computer vision in particular.

The first author was partially funded by the New York Metropolitan and the Technion's Security Science and Technology research funds, The German-Israeli Foundation for Scientific Research and Development (GIF) Young Scientist Program, the Technion Center of Excellence in Exposure Science and Environmental Health and the CITI-SENSE project of the 7th European Framework Program (FP7), ENV.2012.6.5-1. The second author was supported in part by NSF awards No. CMMI-1200592 and CBET-0736232.

Appendix1: Problem sizes

See Tables 6, 7, 8, 9

Table 7 Problems' sizes—multi-view problems

Multi-view						
Name	Image size	Views	Grid	NBH	Nodes	Arcs
camel-lrg	189 × 220	20	100 × 75 × 105	24	18,900,002	93,749,846
camel-med	189 × 220	20	80 × 60 × 84	24	9,676,802	47,933,324
camel-sml	189 × 220	20	40 × 30 × 42	24	1,209,602	5,963,582
gargoyle-lrg	486 × 720	16	80 × 112 × 80	24	17,203,202	86,175,090
gargoyle-med	486 × 720	16	64 × 90 × 64	24	8,847,362	44,398,548
gargoyle-sml	486 × 720	16	32 × 45 × 32	24	1,105,922	5,604,568

Views is the number of different views taken for the 3D reconstruction. *grid* is the size of the reconstructed 3D grid, *NBH* is the oriented neighborhood around each pixel (see [36])

Table 8 Problems' sizes: surface fitting problems

Surface fitting					
Name	Points	Grid	Conn	Nodes	Arcs
bunny-lrg	362,272	401 × 396 × 312	6	49,544,354	300,838,741
bunny-med	362,272	401 × 396 × 312	6	6,311,088	38,739,041
bunny-sml	362,272	401 × 396 × 312	6	805,802	5,040,834

Points is the number of scanned points. *grid* is the size of the reconstructed 3D grid, *Conn* is the 3D connectivity scheme used

Table 9 Segmentation problems' sizes

Segmentation			
Name	3D Image size	Nodes	Arcs
adhead.n26c10	256 × 256 × 192	12,582,914	327,484,556
adhead.n26c100	256 × 256 × 192	12,582,914	327,484,556
adhead.n6c10	256 × 256 × 192	12,582,914	75,826,316
adhead.n6c100	256 × 256 × 192	12,582,914	75,826,316
babyface.n26c10	250 × 250 × 81	5,062,502	131,636,370
babyface.n26c100	250 × 250 × 81	5,062,502	131,636,370
babyface.n6c10	250 × 250 × 81	5,062,502	30,386,370
babyface.n6c100	250 × 250 × 81	5,062,502	30,386,370
bone.n26c10	256 × 256 × 119	7,798,786	202,895,861
bone.n26c100	256 × 256 × 119	7,798,786	202,895,861
bone.n6c10	256 × 256 × 119	7,798,786	46,920,181
bone.n6c100	256 × 256 × 119	7,798,786	46,920,181
bone.x.n26c10	128 × 256 × 119	3,899,394	101,476,818
bone.x.n26c100	128 × 256 × 119	3,899,394	101,476,818
bone.x.n6c10	128 × 256 × 119	3,899,394	23,488,978
bone.x.n6c100	128 × 256 × 119	3,899,394	23,488,978
bone.xy.n26c10	128 × 128 × 119	1,949,698	50,753,434
bone.xy.n26c100	128 × 128 × 119	1,949,698	50,753,434
bone.xy.n6c10	128 × 128 × 119	1,949,698	11,759,514
bone.xy.n6c100	128 × 128 × 119	1,949,698	11,759,514
bone.xyz.n26c10	128 × 128 × 60	983,042	25,590,293
bone.xyz.n26c100	128 × 128 × 60	983,042	25,590,293
bone.xyz.n6c10	128 × 128 × 60	983,042	5,929,493
bone.xyz.n6c100	128 × 128 × 60	983,042	5,929,493
bone.xyz.x.n26c10	64 × 128 × 60	491,522	12,802,789
bone.xyz.x.n26c100	64 × 128 × 60	491,522	12,802,789
bone.xyz.x.n6c10	64 × 128 × 60	491,522	2,972,389
bone.xyz.x.n6c100	64 × 128 × 60	491,522	2,972,389
bone.xyz.xy.n26c10	64 × 64 × 60	245,762	6,405,104
bone.xyz.xy.n26c100	64 × 64 × 60	245,762	6,405,104
bone.xyz.xy.n6c10	64 × 64 × 60	245,762	1,489,904
bone.xyz.xy.n6c100	64 × 64 × 60	245,762	1,489,904
liver.n26c10	128 × 128 × 119	4,161,602	108,370,821
liver.n26c100	128 × 128 × 119	4,161,602	108,370,821
liver.n6c10	128 × 128 × 119	4,161,602	25,138,821
liver.n6c100	128 × 128 × 119	4,161,602	25,138,821

Appendix2: Run-times

See Tables 10, 11, 12, 13, 14, 15, 16, 17

Table 10 Initialization stage run-times: *Stereo Vision* problems

Stereo			
Instance	PRF	HPF	BK
sawtoothBVZ	1.02	0.57	0.53
sawtoothKZ2	2.69	1.40	1.32
tsukubaBVZ	0.65	0.37	0.34
tsukubaKZ2	1.79	0.96	0.90
venusBVZ	1.02	0.57	0.53
venusKZ2	2.79	1.45	1.36
Average	**1.66**	**0.89**	**0.83**

Table 11 Initialization stage run-times: *Multi-View* problems

Multi-view			
Instance	PRF	HPF	BK
camel-lrg	155.25	76.97	70.22
camel-med	76.40	38.44	35.15
camel-sml	8.66	4.65	4.23
gargoyle-lrg	141.81	68.79	63.50
gargoyle-med	70.79	34.97	32.33
gargoyle-sml	8.11	4.33	3.94
Average	**76.84**	**38.02**	**34.89**

Table 12 Initialization stage run-times: *Surface Fitting* problems

Surface fitting			
Instance	PRF	HPF	BK
bunny-lrg	687.01	230.81	219.08
bunny-med	70.87	29.25	27.95
bunny-sml	7.99	3.70	3.47
Average	**255.29**	**87.92**	**83.50**

Table 13 Initialization stage run-times: *Segmentation* problems

Segmentation			
Instance	PRF	HPF	BK
adhead.n26c10	697.57	238.71	233.34
adhead.n26c100	702.45	242.09	239.02
adhead.n6c10	144.05	58.00	55.12
adhead.n6c100	146.12	59.82	56.47
babyface.n26c10	180.35	94.24	92.50
babyface.n26c100	182.76	95.88	94.62
babyface.n6c10	44.31	22.77	21.71
babyface.n6c100	44.85	23.37	22.35
bone.n26c10	381.60	146.00	144.64
bone.n26c100	382.59	149.93	145.70
bone.n6c10	85.74	35.58	33.66
bone.n6c100	86.26	36.68	34.72
bone.x.n26c10	182.47	72.52	71.59
bone.x.n26c100	183.88	73.70	72.54
bone.x.n6c10	41.02	17.62	16.78
bone.x.n6c100	41.46	18.18	17.28
bone.xy.n26c10	87.33	36.05	35.32
bone.xy.n26c100	88.08	36.37	35.89
bone.xy.n6c10	19.62	8.73	8.26
bone.xy.n6c100	19.77	8.99	8.60
bone.xyz.n26c10	41.91	17.60	17.31
bone.xyz.n26c100	42.26	17.93	17.74
bone.xyz.n6c10	9.26	4.28	4.10
bone.xyz.n6c100	9.29	4.46	4.21
bone.xyz.x.n26c10	20.24	8.80	8.66
bone.xyz.x.n26c100	20.47	8.93	8.76
bone.xyz.x.n6c10	4.32	2.16	2.05
bone.xyz.x.n6c100	4.39	2.23	2.10
bone.xyz.xy.n26c10	9.63	4.38	4.30
bone.xyz.xy.n26c100	9.68	4.47	4.34
bone.xyz.xy.n6c10	2.05	1.06	1.02
bone.xyz.xy.n6c100	2.08	1.11	1.05
liver.n26c10	200.28	76.43	75.65
liver.n26c100	200.64	77.25	76.02
liver.n6c10	44.58	18.68	17.90
liver.n6c100	44.76	18.91	17.99
Average	**122.45**	**48.44**	**47.31**

Table 14 Total run-times of the initialization and min-cut and max-flow stages: *Stereo Vision* problems

Stereo			
Instance	PRF	HPF	BK
sawtoothBVZ	1.64	0.88	0.62
sawtoothKZ2	4.26	2.02	1.52
tsukubaBVZ	1.15	0.57	0.41
tsukubaKZ2	3.55	1.42	1.06
venusBVZ	2.02	0.94	0.66
venusKZ2	5.32	2.22	1.66
Average	**2.99**	**1.34**	**0.99**

Table 15 Total run-times of the initialization and min-cut and max-flow stages: *Multi-View* problems

Multi-view			
Instance	PRF	HPF	BK
camel-lrg	563.52	180.57	225.53
camel-med	242.74	80.56	83.43
camel-sml	17.44	6.92	6.46
gargoyle-lrg	417.75	154.40	432.31
gargoyle-med	198.56	66.66	226.43
gargoyle-sml	12.54	5.61	13.86
Average	**242.09**	**82.45**	**164.67**

Table 16 Total run-times of the initialization and min-cut and max-flow stages: *Surface Fitting* problems

Surface fitting			
Instance	PRF	HPF	BK
bunny-lrg	1595.21	440.59	277.02
bunny-med	131.78	43.50	32.82
bunny-sml	11.18	4.74	4.03
Average	**579.39**	**162.94**	**104.62**

Table 17 Total run-times of the initialization and min-cut and max-flow stages: *Segmentation* problems

Segmentation			
Instance	PRF	HPF	BK
adhead.n26c10	982.98	356.07	407.42
adhead.n26c100	985.39	383.55	476.04
adhead.n6c10	223.30	94.54	90.38
adhead.n6c100	248.06	110.58	123.22
babyface.n26c10	341.32	264.41	250.15
babyface.n26c100	392.62	325.34	321.20
babyface.n6c10	105.48	52.74	35.30
babyface.n6c100	129.39	72.35	43.89
bone.n26c10	456.03	163.58	196.26
bone.n26c100	477.26	174.07	198.73
bone.n6c10	121.31	42.74	47.79
bone.n6c100	133.96	47.02	51.88
bone_subx.n26c10	211.19	80.94	100.23
bone_subx.n26c100	220.91	83.42	107.43
bone_subx.n6c10	64.91	21.41	26.57
bone_subx.n6c100	62.00	22.52	30.29
bone_subxy.n26c10	100.65	40.13	48.25
bone_subxy.n26c100	102.25	40.87	51.04
bone_subxy.n6c10	27.58	10.30	12.09
bone_subxy.n6c100	28.05	10.83	13.62
bone_subxyz.n26c10	48.60	19.15	23.69
bone_subxyz.n26c100	49.39	19.71	25.41
bone_subxyz.n6c10	12.13	4.98	5.95
bone_subxyz.n6c100	12.24	5.28	6.54
bone_subxyz_subx.n26c10	23.23	9.48	11.27
bone_subxyz_subx.n26c100	23.82	9.68	11.55
bone_subxyz_subx.n6c10	5.56	2.43	2.75
bone_subxyz_subx.n6c100	5.65	2.54	2.89
bone_subxyz_subxy.n26c10	11.12	4.67	5.60
bone_subxyz_subxy.n26c100	11.20	4.80	5.79
bone_subxyz_subxy.n6c10	2.60	1.16	1.34
bone_subxyz_subxy.n6c100	2.64	1.24	1.39
liver.n26c10	275.68	126.03	112.40
liver.n26c100	301.11	135.20	128.60
liver.n6c10	83.42	36.99	32.02
liver.n6c100	97.88	42.30	43.60
Average	**177.25**	**78.42**	**84.79**

Appendix3: Memory utilization

See Tables 18, 19, 20, 21

Table 18 Memory utilization in (MBytes) for *Stereo* problems

Stereo			
Instance	PRF	HPF	BK
sawtoothBVZ	62.28	58.50	69.27
sawtoothKZ2	141.08	125.81	147.55
tsukubaBVZ	41.62	39.55	48.79
tsukubaKZ2	97.72	87.12	104.57
venusBVZ	62.27	58.65	69.24
venusKZ2	145.76	129.76	152.22

Table 19 Memory utilization in (MBytes) for *multi-view* problems

Multi-view			
Instance	PRF	HPF	BK
camel-lrg	6,879.30	6,392.60	6,814.80
camel-med	3,519.90	3,272.50	3,490.60
camel-sml	441.50	411.30	444.50
gargoyle-lrg	6,313.40	5,851.80	6,255.40
gargoyle-med	3,253.60	3,015.00	3,227.40
gargoyle-sml	413.30	382.52	416.60

Table 20 Memory utilization in (MBytes) for *Surface Fitting* problems

Surface fitting			
Instance	PRF	HPF	BK
bunny-lrg	21,389.40	19,600.30	21,208.00
bunny-med	2,753.30	2,515.00	2,736.80
bunny-sml	360.50	327.70	365.10

Table 21 Memory utilization in (MBytes) for *Segmentation* problems

Segmentation			
Instance	PRF	HPF	BK
adhead.n26c10	20,759.80	16,480.20	20,719.40
adhead.n26c100	20,759.80	16,480.20	20,719.40
adhead.n6c10	5,399.80	4,960.20	5,359.40
adhead.n6c100	5,399.80	4,960.20	5,359.40
babyface.n26c10	8,347.10	6,628.00	8,335.40
babyface.n26c100	8,347.10	6,628.00	8,335.40
babyface.n6c10	2,167.30	1,993.20	2,155.60
babyface.n6c100	2,167.30	1,993.20	2,155.60
bone.n26c10	12,863.50	10,212.70	12,841.30
bone.n26c100	12,863.50	10,212.70	12,841.30
bone.n6c10	3,343.50	3,072.70	3,321.30
bone.n6c100	3,343.50	3,072.70	3,321.30
bone.x.n26c10	6,435.30	5,109.30	6,428.10
bone.x.n26c100	6,435.30	5,109.30	6,428.10
bone.x.n6c10	1,675.30	1,539.30	1,668.10
bone.x.n6c100	1,675.30	1,539.30	1,668.10
bone.xy.n26c10	3,220.40	2,557.04	3,220.60
bone.xy.n26c100	3,220.40	2,557.06	3,220.60
bone.xy.n6c10	840.40	772.00	840.60
bone.xy.n6c100	840.40	772.00	840.60
bone.xyz.n26c10	1,625.60	1,291.00	1,629.40
bone.xyz.n26c100	1,625.60	1,291.00	1,629.40
bone.xyz.n6c10	425.60	391.10	429.40
bone.xyz.n6c100	425.60	391.10	429.40
bone.xyz.x.n26c10	815.10	647.60	820.80
bone.xyz.x.n26c100	815.10	647.60	820.80
bone.xyz.x.n6c10	215.10	197.70	220.80
bone.xyz.x.n6c100	215.10	197.70	220.80
bone.xyz.xy.n26c10	409.60	325.80	416.30
bone.xyz.xy.n26c100	409.60	325.80	416.30
bone.xyz.xy.n6c10	109.60	100.80	116.30
bone.xyz.xy.n6c100	109.60	100.80	116.30
liver.n26c10	6,872.10	5,455.20	6,863.80
liver.n26c100	6,872.10	5,455.20	6,863.80
liver.n6c10	1,792.00	1,645.20	1,783.80
liver.n6c100	1,792.00	1,645.20	1,783.80

References

1. Ahuja, R.K., Kodialam, M., Mishra, A.K., Orlin, J.B.: Computational investigations of maximum flow algorithms. Eur. J. Oper. Res. **97**(3), 509–542 (1997)
2. Ahuja, R.K., Magnanti T.L., Orlin J.B.: Network flows: theory, algorithms, and applications. Prentice-Hall, Englewood Cliffs (1993)
3. Ali, S., Shah, M.: Human action recognition in videos using kinematic features and multiple instance learning. IEEE Trans Pattern Anal. Mach. Intell. **32**(2), 288–303 (2010)
4. Anderson, R.J., Setubal, J.C: Goldberg's algorithm for maximum flow in perspective: a computational study. In: Network flows and matching: First DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 12, pp. 123–133 (1991)
5. Arora, C., Banerjee, S., Kalra, P., Maheshwari, S.: An efficient graph cut algorithm for computer vision problems. In: Kostas, D., Petros, M., Nikos P., (eds.) Computer Vision ECCV 2010. Lecture Notes in Computer Science, vol 6313, pp. 552–565. Springer, Heidelberg (2010)
6. Azar, Y., Madry, A., Moscibroda, T., Panigrahi, D., Srinivasan, A.: Maximum bipartite flow in networks with adaptive channel width. Theor. Comput. Sci. **412**(24), 2577–2587 (2011)
7. Bai, X., Wang, J., Simons, D., Sapiro, G.: Video snapcut: robust video object cutout using localized classifiers. ACM Trans. Graph. **28**(3), 70:1–70:11 (2009)
8. Borradaile, G., Klein, P.: An o(n log n) algorithm for maximum st-flow in a directed planar graph. J. ACM **56**(2), 9:1–9:30 (2009)
9. Boykov, Y., Funka-Lea, G.: Graph cuts and efficient n-d image segmentation. Int. J. Comput. Vis. **70**(2), 109131 (2006)
10. Boykov, Y., Kolmogorov, V.: An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. IEEE Trans. Pattern Anal. Mach. Intell. **26**(9), 1124–1137 (2004)
11. Boykov, Y., Lempitsky, V.: From photohulls to photoflux optimization. In: British Machine Vision Conference (BMVC), vol. III, pp. 1149–1158 (2006)
12. Chandran, B.G., Hochbaum, D.S.: A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. Oper. Res. **57**(2), 358–376 (2009)
13. Cherkassky, B.V., Goldberg A.V.: On implementing the push—relabel method for the maximum flow problem. Algorithmica **19**(4), 390–410 (1997)
14. Computer Vision Research Group. Max-flow problem instances in vision. Technical report, University of Western Ontario. http://vision.csd.uwo.ca (2009). Accessed Oct 2009.
15. Delong, A., Boykov, Y.: A scalable graph-cut algorithm for n-d grids. In: IEEE computer society conference on computer vision and pattern recognition, pp. 1–8 (2008)
16. Derigs, U., Meier, W.: Implementing Goldberg's max-flow-algorithm a computational investigation. Math. Methods Oper. Res. **33**(6), 383–403 (1989)
17. Fishbain, B., Hochbaum, D.S., Yang, Y.T.: Graph-cuts target tracking in videos through joint utilization of color and coarse motion data. UC Berkeley Manuscript (2012)
18. Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. Can. J. Math. **8**(3), 339–404 (1956)
19. Goldberg, A.V.: The partial augment–relabel algorithm for the maximum flow problem. Algorithms-ESA 2008, pp. 466–477 (2008)
20. Goldberg, A.V.: Hi-level variant of the push-relabel (ver. 3.5) (2010). Accessed Jan 2010
21. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM **35**(4), 921–940 (1988)
22. Gracias, N., Mahoor, M., Negahdaripour, S., Gleason, A.: Fast image blending using watersheds and graph cuts. Image Vis. Comput. **27**(5), 597–607 (2009) [The 17th British Machine Vision Conference (BMVC 2006)]
23. Grundmann, M., Kwatra, V., Mei Han, and Essa, I.: Efficient hierarchical graph-based video segmentation. In: 2010 IEEE conference on computer vision and pattern recognition (CVPR), pp. 2141–2148 (2010)
24. Hochbaum D.S.: An efficient algorithm for image segmentation, markov random fields and related problems. J. ACM **48**(4), 686–701 (2001)
25. Hochbaum, D.S.: The pseudoflow algorithm: a new algorithm for the maximum-flow problem. Oper. Res. **56**(4), 992–1009 (2008)

26. Hochbaum, D.S.: Efficient and effective image segmentation interactive tool. In: BIOSIGNALS 2009-international conference on bio-inspired systems and signal processing, pp. 459–461 (2009)
27. Hochbaum, D.S.: Polynomial time algorithms for ratio regions and a variant of normalized cut. IEEE Trans. Pattern Recognit. Mach. Intell. **32**(5), 889–898 (2009)
28. Hochbaum, D.S.: HPF Implementation Ver. 3.3. (2010). Accessed Jan 2010
29. Hochbaum, D.S., Orlin J.B.: Simplifications and speedups of the pseudoflow algorithm. Networks (2012, to appear)
30. Hochbaum, D.S., Singh, V.: An efficient algorithm for co-segmentation. In: International conference on computer vision (ICCV) (2009)
31. Ideses, I., Yaroslavsky, L., Fishbain, B.: Real-time 2d to 3d video conversion. J. Real Time Image Process. **2**(1), 3–9 (2007)
32. Italiano, G.F., Nussbaum, Y., Sankowski, P., and Wulff-Nilsen, C.: Improved algorithms for min cut and max flow in undirected planar graphs. In: Proceedings of the 43rd annual ACM symposium on theory of computing, STOC '11, pp. 313–322. ACM, New York (2011)
33. Kalarot, R., Morris J.: Comparison of fpga and gpu implementations of real-time stereo vision. In: 2010 IEEE computer society conference on computer vision and pattern recognition workshops (CVPRW), pp. 9 –15 (2010)
34. Kolmogorov, V.: An implementation of the maxflow algorithm. http://www.cs.ucl.ac.uk/staff/V.Kolmogorov/software.html (2010). Accessed Jan 2010
35. Lempitsky, V., Boykov, Y.: Global optimization for shape fitting. In: Proceedings of IEEE conference on computer vision and pattern recognition CVPR '07, pp. 1–8 (2007)
36. Lempitsky, V., Boykov, Y., Ivanov, D.: Oriented visibility for multiview reconstruction. In: Leonardis, A, Bischof, H., Pinz, A. (eds.) Computer Vision ECCV 2006. Lecture Notes in Computer Science, vol. 3953, pp. 226–238. Springer, Heidelberg (2006)
37. Liu, J., Sun, J.: Parallel graph-cuts by adaptive bottom-up merging. In: 2010 IEEE conference on computer vision and pattern recognition (CVPR), pp. 2181–2188 (2010)
38. Mu, Y., Zhang, H., Wang, H., Zuo, W.: Automatic video object segmentation using graph cut. In: IEEE international conference on image processing, 2007 (ICIP 2007), vol. 3, pp. III–377–III–380 (2007)
39. Nakamura Y., Matsuura T., Satoh K., and Ohta Y. (1996) Occlusion detectable stereo—occlusion patterns in camera matrix. In: IEEE computer society conference on computer vision and pattern recognition 0:371
40. Ngo, C.-W., Ma, Y.-F., Zhang, H.-J.: Video summarization and scene detection by graph modeling. IEEE Trans. Circuits Syst. Video Technol. **15**(2),. 296–305 (2005)
41. Qranfal, J., Hochbaum, D.S., Tanoh, G.: Experimental analysis of the mrf algorithm for segmentation of noisy medical images. Algorithmic Oper. Res. **6**(2) (2012)
42. Scharstein, D., Szeliski, R., Zabih, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In: Proc. IEEE workshop on stereo and multi-baseline vision (SMBV 2001), pp. 131–140 (2001)
43. Sharon, E., Galun, M., Sharon, D., Basri, R., Brandt, A.: Hierarchy and adaptivity in segmenting visual scenes. Nature **442**(7104), 810–813 (2006)
44. Shekhovtsov, A., Hlavac V.: A distributed mincut/maxflow algorithm combining path augmentation and push-relabel. In: Boykov, Y., Kahl, F., Lempitsky, V., Schmidt, F., (eds.) Energy Minimization Methods in Computer Vision and Pattern Recognition. Lecture Notes in Computer Science, vol. 6819, pp. 1–16. Springer, Heidelberg (2011)
45. Sinha, S.N., Steedly, D., Szeliski, R., Agrawala, M., Pollefeys, M.: Interactive 3d architectural modeling from unordered photo collections. In: SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers, pp. 1–10. ACM, New York (2008)
46. Sleator, D.D., Tarjan R.E.: A data structure for dynamic trees. In: Proceedings of the thirteenth annual ACM symposium on Theory of computing (STOC '81) pp. 114–122. ACM, New York (1981)
47. Snavely, N., Seitz, S.M., Szeliski, R.: Photo tourism: exploring photo collections in 3d. ACM Trans. Graphics **25**(3) (2006)
48. Snow, D., Viola, P., Zabih, R.: Exact voxel occupancy with graph cuts. In: Proceedings of IEEE conference on computer vision and pattern recognition, vol. 1, pp. 345–352 (2000)
49. Stanford Computer Graphics Laboratory. "the stanford 3d scanning repository". Technical report, Stanford, Palo-Alto, CA, USA, http://graphics.stanford.edu/data/3Dscanrep/ (2009). Accessed Oct 2009
50. Starck, J., Hilton, A.: Surface capture for performance-based animation. IEEE Comput. Graphics Appl. **27**(3), 21–31 (2007)
51. Strandmark, P., Kahl, F.: Parallel and distributed graph cuts by dual decomposition. In: 2010 IEEE conference on computer vision and pattern recognition (CVPR), pp. 2085–2092 (2010)
52. Vineet, V., Narayanan, P.J.: Cuda cuts: Fast graph cuts on the gpu. In: IEEE computer society conference on computer vision and pattern recognition workshops, 2008. CVPRW '08. pp. 1–8 (2008)
53. Vogiatzis, G., Torr, P.H.S., Cipolla, R.: Multi-view stereo via volumetric graph-cuts. In: Computer Vision and Pattern Recognition (CVPR), vol. 2, pp. 391–398 (2005)

Author Biographies

Barak Fishbain is an assistant Professor at the Environmental, Water and Agricultural Engineering Division, Faculty of Civil & Environmental Engineering in the Technion-Israel Institute of Technology Haifa, Israel. Prior to his arrival to the Technion, Professor Fishbain served as an associate director at the Integrated Media Systems Center (IMSC), Viterbi school of engineering, University of Southern California (USC) and did his post-doctoral studies at the department of Industrial Engineering and Operations Research (IEOR) in University of California at Berkeley. Professor Fishbain's research focuses on Enviromatics, a new research field which aims at devising mathematical programming methods for machine understanding of trends and behaviors of built and natural environments. This includes environmental distributed sensing (i.e., distributed air and water quality monitoring), road safety and traffic data realization and structural sensory networks.

Dorit S. Hochbaum is a full professor and Chancellor chair at the University of California at Berkeley in the department of Industrial Engineering and Operations Research (IEOR). Her research interests are in the areas of approximation algorithms and design and analysis of computer algorithms and discrete and continuous optimization. Her recent work focuses on efficient techniques for network flow related problems, ranking, data mining and image segmentation problems. Professor Hochbaum is the author of over 140 papers that appeared in the Operations Research, Management Science and Theoretical Computer Science literature. Professor Hochbaum was named in 2004 an honorary doctorate of Sciences of the University of Copenhagen, for her work on approximation algorithms. Professor Hochbaum was awarded in 2005 the title of INFORMS fellow. She is the winner of the 2001 INFORMS Computing Society prize for best paper dealing with the Operations Research/Computer Science interface.

Stefan Mueller studied Mathematics at the Technische Universität Berlin and at University of California, Berkeley. His main interests are Combinatorial Optimization and Geometry. In 2011 he finished his master's thesis on "Confluent Network flows".