Review of Serial and Parallel Min-Cut/Max-Flow Algorithms for Computer Vision

Patrick M. Jensen, Niels Jeppesen, Anders B. Dahl, and Vedrana A. Dahl

Abstract—Minimum cut/maximum flow (min-cut/max-flow) algorithms solve a variety of problems in computer vision and thus significant effort has been put into developing fast min-cut/max-flow algorithms. As a result, it is difficult to choose an ideal algorithm for a given problem. Furthermore, parallel algorithms have not been thoroughly compared. In this paper, we evaluate the state-of-the-art serial and parallel min-cut/max-flow algorithms on the largest set of computer vision problems yet. We focus on generic algorithms, i.e., for unstructured graphs, but also compare with the specialized GridCut implementation. When applicable, GridCut performs best. Otherwise, the two pseudoflow algorithms, Hochbaum pseudoflow and excesses incremental breadth first search, achieves the overall best performance. The most memory efficient implementation tested is the Boykov-Kolmogorov algorithm. Amongst generic parallel algorithms, we find the bottom-up merging approach by Liu and Sun to be best, but no method is dominant. Of the generic parallel methods, only the parallel preflow push-relabel algorithm is able to efficiently scale with many processors across problem sizes, and no generic parallel method consistently outperforms serial algorithms. Finally, we provide and evaluate strategies for algorithm selection to obtain good expected performance. We make our dataset and implementations publicly available for further research.

Index Terms—Algorithms, computer vision, graph algorithms, graph-theoretic methods, parallel algorithms, performance evaluation of algorithms and systems

1 Introduction

Min-cut/max-flow algorithms are ubiquitous in computer vision, since a large variety of computer vision problems can be formulated as min-cut/max-flow problems. Example applications include image segmentation [1], [2], [3], [4], [5], [6], stereo matching [7], [8], surface reconstruction [9], surface fitting [10], [11], [12], [13], [14], [15], graph matching [16], and texture restoration [17]. In recent years, min-cut/max-flow algorithms have also found use in conjunction with deep learning methods — for example, to quickly generate training labels [18] or in combination with convolutional neural networks (CNNs) [19], [20], [21].

Greig *et al.* [22] were the first to use min-cut/max-flow algorithms to solve maximum a posterior Markov random field (MRF) problems in computer vision. Later, Boykov and Jolly [23] showed how this could be generalized and Boykov and Kolmogorov [1] proposed a fast min-cut/max-flow algorithm for computer vision problems. Min-cut/max-flow algorithms in computer vision are used to solve a large family of energy minimization problems, and the most commonly used energy function is of the form

$$\mathcal{E}(\mathbf{x}) = \sum_{i\in\mathcal{P}} \mathcal{E}_i(x_i) + \sum_{(i,j)\in\mathcal{N}} \mathcal{E}_{ij}(x_i, x_j), \tag{1}$$

where $\mathcal{P}$ is a set of indices for the binary variables $x_i \in \{0,1\}$, and $\mathcal{N}$ is a set of index pairs. A unary term $\mathcal{E}_i : \{0,1\} \to \mathbb{R}$ is associated with variable x_i, and a pairwise term $\mathcal{E}_{ij} : \{0,1\}^2 \to \mathbb{R}$ is associated with the pair of variables x_i, x_j. As the inputs to the energy terms are binary, the terms are often represented as lookup tables. In a typical application, such as binary segmentation with MRFs [24], $\mathcal{P}$ represents pixels in an image and x_i represents the assignment of pixel i. However, variables can also describe more abstract things [5], [10], [13], [14], [16], [25], e.g., candidate positions for mesh vertices.

For energy functions which are *submodular*, meaning that all pairwise energy terms satisfy the condition

$$\mathcal{E}_{ij}(0,0) + \mathcal{E}_{ij}(1,1) \le \mathcal{E}_{ij}(0,1) + \mathcal{E}_{ij}(1,0), \tag{2}$$

the minimization can be solved directly as a min-cut/max-flow problem [26], [27]. Submodular energies favor neighbors that have the same label, i.e., (x_i, x_j) having the labels (0,0) or (1,1) rather than (0,1) or (1,0). Therefore, submodularity imposes a local smoothness of the solution, which is useful in many computer vision problems. However, some important vision problems are not submodular. In such cases, one can use either a submodular approximation or an approach based on quadratic pseudo-Boolean optimization (QPBO) as described in [17], [28], [29], [30].

Due to the wide applicability of min-cut/max-flow in computer vision, several fast generic min-cut/max-flow algorithms have been developed [1], [31], [32]. In addition, more specialized algorithms have been created that exploit the grid structure of images to reduce memory usage and run time [33], [34], [35], [36], [37]. Furthermore, methods for dynamic problems [31], [38], [39], where a series of similar min-cut/max-flow problems are solved in succession, have

• *The authors are with the Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark. E-mail: {patmjen, niejep, abda, vand}@dtu.dk.*

Manuscript received 14 June 2021; revised 1 April 2022; accepted 19 April 2022. Date of publication 26 April 2022; date of current version 6 January 2023. (Corresponding author: Patrick M. Jensen.) Recommended for acceptance by T. Pock MSc. Digital Object Identifier no. 10.1109/TPAMI.2022.3170096

been proposed. For such problems, the result of the first solution can be reused to speed up computations of subsequent solutions. Finally, some papers [40], [41], [42], [43] have explored methods that also allow for distributed computation of min-cut/max-flow problems across several computational nodes. This approach is primarily suited for graphs too large to fit in physical memory.

In this paper, we focus on generic min-cut/max-flow algorithms, which do *not* make assumptions about the graph structure (e.g., requiring a grid structure). However, for comparison, we include the GridCut algorithm [35] in our evaluation on grid-based graphs. Furthermore, we consider only static problems where a solution is calculated once, without access to a previous solution (as opposed to dynamic problems). Finally, for parallel algorithms, we do not consider whether the algorithm works well in a distributed setting, but focus on the shared memory case where the complete graph can be loaded into the memory of one machine.

The goal is that our experimental results can help researchers understand the strengths and weaknesses of the current state-of-the-art min-cut/max-flow algorithms and help practitioners when choosing a min-cut/max-flow algorithm to use for a given problem.

1.1 Related Work

Serial Algorithms. Several papers [31], [44], [45], [46] provide comparisons of different serial min-cut/max-flow algorithms on a variety of standard benchmark problems. However, many of these benchmark problems are small w.r.t. the scale of min-cut/max-flow problems that can be solved today — especially when it comes to grid graphs. Also, graphs in which nodes are not based on an image grid are severely underrepresented. Furthermore, [44], [45], [46] do not include all current state-of-the-art algorithms, while other papers do not include initialization times for the min-cut computation. As shown by Verma and Batra [46], it is important for practical use to include the initialization time, as algorithm implementations may spend as much time on initialization as on the min-cut computation. Additionally, existing papers only compare reference implementations (i.e., the implementation released by the authors) of algorithms — the exception being that an optimized version of the BK algorithm is sometimes included, e.g., in [31]. However, as implementation details — i.e., choices that are left unspecified by the algorithm description — can significantly impact performance [46], a systematic investigation of their effect is also important. Finally, existing comparisons focus on determining the overall best algorithm, even though, as we show in this work, the best algorithm depends on the features of the given graph.

Parallel Algorithms. To our knowledge, parallel min-cut/max-flow algorithms have not been systematically compared. Papers introducing parallel algorithms only compare with serial algorithms [42], [43], [47] or a single parallel algorithm [48]. The most comprehensive comparison so far was made by Shekhovtsov and Hlaváč [41] who included a generic and grid-based parallel algorithm. However, no paper compares with the approach by Liu and Sun [47], as no public implementation is available, even though it is expected to be the fastest [41], [42]. Additionally, all papers use the same set of computer vision problems used to benchmark serial algorithms. This is not ideal, as the set lacks larger problems which we expect to benefit the most from parallelization [49]. Therefore, how big the performance benefits of parallelization are, and when to expect them, is still to be determined.

1.2 Contributions

We evaluate current state-of-the-art generic serial and parallel min-cut/max-flow algorithms on the largest set of computer vision problems so far. We compare the algorithms on a wide range of graph problems including commonly used benchmarks problems, as well as many new problem instances from recent papers — some of which are significantly larger than previous problems and expose weaknesses in the algorithms not seen with previous datasets. Since the performance of the algorithms varies between problems, we also provide concrete strategies on algorithm selection and evaluate the expected performance of these.

For the serial algorithms, we evaluate the reference implementations of the Hochbaum pseudoflow (HPF) [32], [50], the preflow push-relabel (PPR) [51], and the GridCut [34] algorithms. Moreover, to reduce the influence of implementation details, we evaluate different versions (including our own) of the Excesses Incremental Breadth First Search (EIBFS) [31] and the Boykov-Kolmogorov (BK) [1] algorithm. We chose these for an extended evaluation, as EIBFS is the most recent min-cut/max-flow algorithm and BK is still widely used in the computer vision community.

For the parallel algorithms, we provide the first comprehensive comparison of all major approaches. This includes our own implementation of the bottom-up merging algorithm by Liu and Sun [47], our own version of the dual decomposition algorithm by Strandmark and Kahl [42], the reference implementation of the region discharge algorithm by Shekhovtsov and Hlaváč [41], an implementation of the parallel preflow push-relabel algorithm by Baunstark *et al.* [48], and the parallel implementation of GridCut (P-GridCut) [34]. In our comparison, we evaluate not just the run time — including both the initialization time and the time for the min-cut/max-flow computations — but also the memory use of the implementations. Memory usage has not received much attention in the literature, despite it often being a limiting factor when working with large problems. Finally, we show that the current parallel algorithm implementations have unpredictable performance and unfortunately often perform worse than serial algorithms.

All tested C++ implementations (except GridCut [34]), including our new implementations of several algorithms, are available at https://github.com/patmjen/maxflow_algorithms and are archived at DOI:10.5281/zenodo.4903945 [52]. We also provide Python wrapper packages for several of the algorithms (including BK and HPF), which can be found at https://github.com/skielex/shrdr. All of our benchmark problems are available at DOI:10.11583/DTU.17091101.

2 Min-Cut/Max-Flow Algorithms for Computer Vision

To illustrate the use of min-cut/max-flow, we will sketch how a vision problem, image segmentation, can be solved using min-cut/max-flow. We start by introducing our notation and defining the min-cut/max-flow problem.

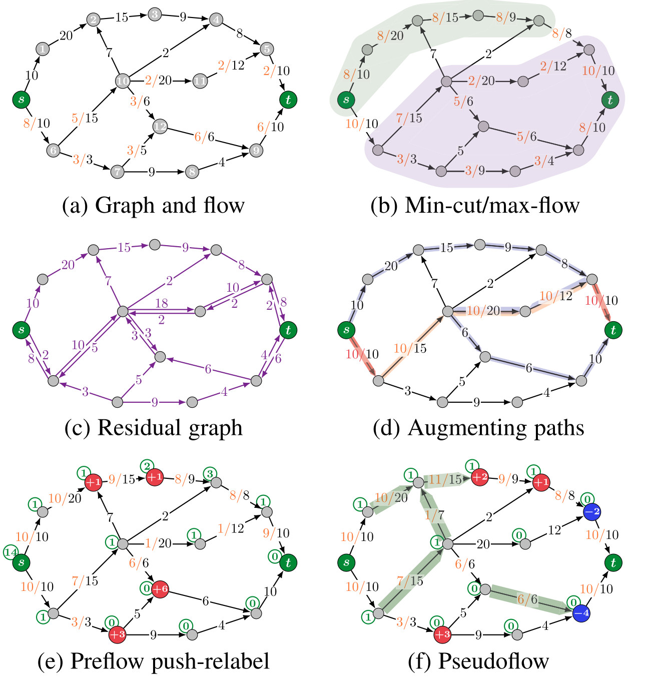


Fig. 1. *Graph basics and serial algorithms.* (a) An example of the graph and a feasible (non-maximal) flow. The flow and capacity for each arc is written as f_{ij}/c_{ij}, and (to reduce clutter) zero-values of the flow are omitted. The flow is 8, which is not maximal, so no s-t cut is evident. (b) The min-cut/max-flow with a value of 18, which all min-cut/max-flow algorithms will eventually arrive at. (c) Residual graph for the flow from (a). (d) An intermediate flow while running the AP algorithm. In the first iteration, 10 units are pushed along the path highlighted in orange and red, saturating two terminal arcs (red). In the next iteration, flow is pushed along the residual path highlighted in blue. (e) A preflow at an intermediate stage of a PPR algorithm. Nodes with excess are shown in red, and a label in green is attached to every node. (f) A pseudoflow at an intermediate stage of the HPF algorithm. Nodes with surplus/deficit are shown in red/blue, a label is attached to every node, and arcs of the tree structure are highlighted in green.

We define a directed graph $G = (V, E)$ by a set of *nodes*, V, and a set of directed *arcs*, E. We let n and m refer to the number of nodes and arcs, respectively. Each arc $(i, j) \in E$ is assigned a non-negative *capacity* c_{ij}. For min-cut/max-flow problems, we define two special *terminal nodes*, s and t, which are referred to as the *source* and *sink*, respectively. The source has only outgoing arcs, while the sink has only incoming arcs. Arcs to and from the terminal nodes are known as *terminal arcs*.

A *feasible flow* in the graph G is an assignment of non-negative numbers (flows), f_{ij}, to each arc $(i, j) \in E$. A feasible flow must satisfy the following two types of constraints: *capacity constraints*, $f_{ij} \leq c_{ij}$, and *conservation constraints*, $\sum_{i|(i,j)\in E} f_{ij} = \sum_{k|(j,k)\in E} f_{jk}$ for all nodes $j \in V \setminus \{s, t\}$. Capacity constraints ensure that the flow along an arc does not exceed its capacity. Conservation constraints ensure that the flow going into a node equals the flow coming out. See Fig. 1a for an example of the graph and a feasible flow. The *value* of the flow is the total flow out of the source or, equivalently, into the sink, and the *maximum flow* problem refers to finding a feasible flow that maximizes the flow value.

An *s-t cut* is a partition of the nodes into two disjoint sets S and T such that $s \in S$ and $t \in T$. The sets S and T are referred to as the source and sink set, respectively. The *capacity* of the cut is the sum of capacities of the arcs going from S to T. And the *minimum cut* problem refers to finding a cut that minimizes the cut capacity. Often, this partition of the nodes is all that is needed for computer vision applications. Therefore, some algorithms only compute the minimum cut, and an additional step would be needed to extract the flow value for every arc.

Finally, the max-flow min-cut theorem states that the value of the maximum flow is exactly the capacity of the minimum cut. Fig. 1b shows min-cut/max-flow on a small graph. This can be shown by formulating both problems as linear programs, which reveals that max-flow is the strong dual of the min-cut.

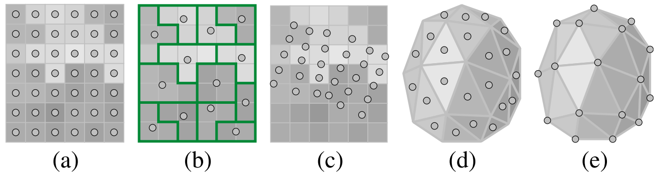


Fig. 2. *Some possibilities for associating graph nodes with entities used for segmentation.* Graph nodes (gray dots) associated with (a) image pixels, (b) superpixels, (c) positions in the image, (d) mesh faces, and (e) mesh vertices.

2.1 Image Segmentation

When formulating segmentation as a min-cut/max-flow problem, one modeling choice involves deciding which structures to represent as graph nodes. Often, nodes of the graph represent individual image pixels, but various other entities may be associated with graph nodes, some of which are illustrated in Fig. 2.

The energy formulation (1) is convenient when min-cut/max-flow algorithms are used to optimize MRFs. Here, each unary energy term is a likelihood energy (negative log likelihood) of a pixel being labeled 0 or 1. Likelihood terms are typically computed directly from image data. The pairwise terms are defined for pairs of pixels, so-called neighbors, and for 2D images, the neighborhood structure is usually given by a 4 or 8-connectivity.

The typical pairwise energy terms used in (1) are

$$\mathcal{E}_{ij}(0,0)=\mathcal{E}_{ij}(1,1)=0 \text{ and } \mathcal{E}_{ij}(0,1)=\mathcal{E}_{ij}(1,0)=\beta_{ij} \ . \qquad (3)$$

These terms penalize neighboring pixels having different labels by a fixed amount, β_{ij}, thus *encouraging* smoothness of the segmentation. In this case, the construction of the s-t graph which exactly represents the energy function is straightforward: The node set is $V = \mathcal{P} \cup \{s, t\}$. For terminal arc capacities, c_{si} and c_{it}, we use the unary terms $\mathcal{E}_i(0)$ and $\mathcal{E}_i(1)$, respectively. Meanwhile, pairwise energy terms correspond to non-terminal arc capacities, such that $c_{ij} = c_{ji} = \beta_{ij}$. Fig. 3a shows this construction for a 4-connected grid graph. The binary segmentation of the image corresponds directly to the binary labeling given by the minimum cut. Put in another way, the sets S and T give the optimal labeling of the nodes, and because we have a 1-to-1 mapping between non-terminal nodes and pixels, the node labeling is the segmentation. However, there are many more advanced ways to formulate image segmentation using binary energy optimization and s-t graphs, and ways to formulate other computer vision problems as well [27].

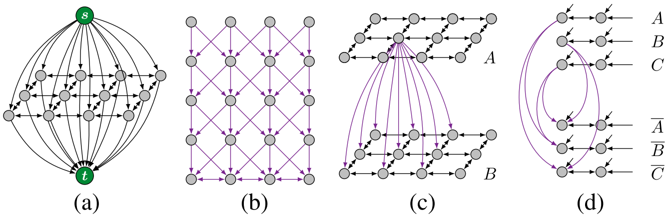


Fig. 3. *Some typical segmentation models.* Terminal arcs are shown only for the first example. Arcs drawn in purple have infinite capacity. (a) A classical MRF segmentation with 4-connected grid graph. (b) A multi-column graph used for segmenting layered structures. (c) Two-object segmentation with inclusion constraint. (d) Three-object segmentation with mutual exclusion using QPBO.

An example closely related to image segmentation is surface fitting, where [13] uses arcs of infinite capacity (i.e., infinite pairwise energy terms) to *impose* a structure to the optimal solution. In Fig. 3b, downward-pointing arcs ensure that if a pixel is in a source set, the column of pixels below it is also in the source set — so the optimal solution has to be a layer. The slanted arcs impose the smoothness of this layer.

It is also possible to formulate multi-label/multi-object segmentation problems that can be solved with a single s-t cut [4], [5], [13], [33], or by iteratively changing and computing the cut [3], [53]. For the single-cut Ishikawa method [4], it is common to duplicate the graph for each label, i.e., having a sub-graph per label. For example, in Fig. 3, each pixel is represented by two nodes: one for object A and one for object B, so a pixel may be segmented as belonging to A, B, both, or neither. The submodular interaction between the objects may be achieved by adding arcs between the sub-graphs. Fig. 3c shows submodular interaction, where arcs with infinite capacity ensure that if a pixel belongs to object A, this pixel and all its neighbors also belong to object B. This is known as *inclusion* or *containment* with a minimum margin of one.

In the examples covered so far, the arcs between the graph nodes correspond to submodular energy terms, which means the energy is lower when the nodes belong to the same set (S or T). Mutual exclusion, in the general case, requires non-submodular energies which are not directly translatable to arcs in the graphs shown so far. An alternative is to use QPBO [30], as illustrated in Fig. 3d, which can handle any energy function of the form in (1) — submodular or not. When using QPBO, we construct two sub-graphs for each object: one representing the object and another representing its complement. The exclusion of two objects, say A and B, is then achieved by adding inclusion arcs from A to $\overline{B}$ and from B to $\overline{A}$. However, there is no guarantee that the min-cut/max-flow solution yields a complete segmentation of the object as the object and its complement may disagree on the labeling of some nodes leaving them "unlabeled". The number of unlabeled nodes depends on the non-submodularity of the system. Extensions to QPBO, such as QPBO-P and QPBO-I [17], may be used to iteratively assign labels to the nodes that QPBO failed to label.

3 Serial Min-Cut/Max-Flow Algorithms

All min-cut/max-flow algorithms find the solution by iteratively updating a flow that satisfies the capacity constraints. Such a flow induces a *residual graph* with the set of *residual arcs*, R, given by

$$R = \{(i,j) \in V \times V \mid (i,j) \in E, f_{ij} < c_{ij} \text{ or } (j,i) \in E, f_{ji} > 0\}. \quad (4)$$

Each of the residual arcs has a *residual capacity* given by $c'_{ij} = c_{ij} - f_{ij}$ if $(i,j) \in E$ or $c'_{ij} = f_{ji}$ if $(j,i) \in E$. In other words, residual arcs tell us how much flow on the original arc we can increase or decrease, see Fig. 1c. If the graph contains bidirectional arcs, both conditions from (4) may be met, and the residual capacity then equals the sum of two contributions.

Serial min-cut/max-flow algorithms can be divided into three families: augmenting paths, preflow push-relabel, and pseodoflow algorithms. In this section, we provide an overview of how algorithms from each family work.

3.1 Augmenting Paths

The augmenting paths (AP) family of min-cut/max-flow algorithms is the oldest of the three families and was introduced with the Ford-Fulkerson algorithm [54]. An algorithm from the AP family always maintains a feasible flow. It works by repeatedly finding so-called *augmenting paths*, which are paths from s to t in the residual graph. When an augmenting path is found, a flow is pushed along the path. Pushing flow means increasing flow for each forward arc along the path, and decreasing flow for each reverse arc. To maintain the capacity constraints, the flow that is pushed equals the minimum residual capacity along the path. Conservation constrains are maintained as the algorithm only updates complete paths from s to t. The algorithm terminates when no augmenting paths can be found. Fig. 1d shows an intermediate stage of an AP algorithm.

The primary difference between various AP algorithms lies in how the augmenting paths are found. For computer vision applications, the most popular AP algorithm is the Boykov-Kolmogorov (BK) algorithm [1], which works by building search trees from both the source and sink nodes to find augmenting paths and uses a heuristic that favors shorter augmenting paths. The BK algorithm performs well on many computer vision problems, but its theoretical run time bound is worse than other algorithms [46].

In terms of performance, the BK algorithm has been surpassed by the Incremetal Breadth First Search (IBFS) algorithm by Goldberg *et al.* [55]. The main difference between the two algorithms is that IBFS maintains the source and sink search trees as breadth-first search trees, which results in both better theoretical run time and better practical performance [31], [55].

3.2 Preflow Push-Relabel

The second family of algorithms are the preflow push-relabel (PPR) algorithms, which were introduced by Goldberg and Tarjan [51]. These algorithms use a so-called *preflow*, which satisfies capacity constraints but allows nodes to have more incoming than outgoing flow, thus violating conservation constraints. The difference between the incoming and outgoing flows for a node, i, is denoted as its *excess*, $e_i \geq 0$.

The PPR algorithms work by repeatedly pushing flow along the individual arcs. To determine which arcs admit flow, the algorithms maintain an integer labeling (so-called *height*), d_i, for every node. The labeling provides a lower bound on the distance from the node to the sink and has a *no* *steep drop* property, meaning $d(i) - d(j) \leq 1$ for any residual arc (i, j).

An algorithm from the PPR family starts by saturating the source arcs and raising the source to $d(s) = n$. The algorithm then works by repeatedly selecting a node with excess (after selection called a *selected* node) and applying one of two actions [51], [56]: *push* or *relabel*. If there is an arc in the residual graph leading from the selected node to a lower-labeled node, *push* is performed. This pushes excess along the arc, until all excess is pushed or the arc is saturated. If no residual arc leads to a lower node, the *relabel* operation is used to lift the selected node (increase its label) by one. Fig. 1e shows an intermediate step of a PPR algorithm.

When there are no nodes with excess left, the preflow is the maximum flow. It is possible to terminate the algorithm earlier, when no nodes with excess have a label $d_i < n$. At this point, the minimum s-t cut can be extracted by inspecting the node labels. If $d_i \geq n$, then $i \in S$, otherwise $i \in T$. Extracting the maximum flow requires an extra step of pushing all excess back to the source. However, this work generally only represents a small part of the run time [46] and, for computer vision applications, we are typically only interested in the minimum cut anyway.

The difference between various PPR algorithms lies in the order in which the push and relabel operations are performed. Early variants used simple heuristics, such as always pushing flow from the node with the highest label or using a first-in-first-out queue to keep track of nodes with positive excess [44]. More recent versions [57], [58], [59] use sophisticated heuristics and a mix of local and global operations to obtain significant performance improvements over early PPR algorithms.

Unlike other serial algorithms, the algorithms from the PPR family operate locally on nodes and arcs. This, as we shall discuss later, has resulted in a whole family of parallel PPR algorithms.

3.3 Pseudoflow

The most recent family of min-cut/max-flow algorithms is the pseudoflow family, which was introduced with the Hochbaum pseudoflow (HPF) algorithm [32], [50]. These algorithms use a so-called *pseudoflow*, which satisfies capacity constraints but not the conservation constraints, as it has no constraints on the difference between incoming and outgoing flow. As with preflow, we refer to the difference between incoming and outgoing flow for a node as its excess, e_i. A positive excess is referred to as a *surplus* and a negative excess as a *deficit*.

During operation, HPF algorithms maintain two auxiliary structures: the forest of trees and a node labeling function. Only one node in every tree, the root, is allowed to have an excess. The algorithm works by repeatedly pushing the flow along the paths connecting the trees, and growing the trees.

A generic algorithm from the HPF family is initialized by saturating all terminal arcs. At this point, each graph node is a singleton tree in the forest. The algorithm then selects a tree with surplus and containing at least one node with the label less than n (the number of nodes in the graph). In this tree, i denotes the node with the lowest label. If there are no residual arcs from i to a node with a lower label, the label of i is incremented. If there is a residual arc (i, j) that leads to a node j with a lower label, a *merge* is performed. This operation involves pushing surplus along the path from the root of the tree containing i, over i, over j, and to the root of the tree containing j. If the arc capacities along this path allow it, the entire surplus will be pushed and the trees will be merged with j as the root. If the flow along the path saturates an arc (i', j'), a surplus will be collected in i', and a new tree rooted in i' will be created. In contrast to the AP algorithms, the only restrictions on how much flow to push are the individual arc capacities, not the path capacity.

The algorithm terminates when no selection can be made, at which point nodes labeled with n constitute the source set. Additional processing is needed to recover the maximum feasible flow. Fig. 1f shows an intermediate step of the HPF algorithm.

There are two main algorithms in this family: HPF and Excesses Incremental Breadth First Search (EIBFS) [31]. The main differences are the order in which they scan through nodes when looking for an arc connecting two trees in the forest, and how they push flow along the paths. Both have sophisticated heuristics for these choices, which makes use of many of the same ideas developed for PPR algorithms.

3.4 Implementation Details

As stressed by [46], the implementation details can significantly affect the measured performance of a given min-cut/max-flow algorithm. In this section, we will highlight the trends of modern implementations and how they differ.

3.4.1 Data Structures and Data Types

The implementations considered in this paper all use a variant of the adjacency list structure [60] to represent the underlying graph. The most common setup mimics the BK algorithm: there is a list of nodes and a list of directed (half-)arcs. Each `Node` structure stores a pointer to its first outgoing half-arc. Each `Arc` stores a pointer to the node it points to, a pointer to the next outgoing arc for the node it points from, a pointer to its reverse arc, and a residual capacity. For algorithms implemented with computer vision applications in mind (e.g., BK, IBFS, and EIBFS), the terminal arcs are stored as a single combined terminal capacity for each `Node`, instead of using the `Arc` structures. Other implementations simply keep track of the source and sink nodes and use `Arc` structures for all arcs. The HPF implementation uses a bidirectional `Arc` structure with a capacity, a flow, and a direction. It is also common to store auxiliary values such as excesses, labels, or more.

As a result of these differences, the memory footprint varies between implementations, as shown in Table 1. The footprint also depends heavily on the data types used to store the data, in particular references to nodes and arcs, as we discuss in the next subsection. For storing arc capacities, integers are common because they are computationally efficient and may use as little as 1 byte. However, some graph constructions involve large capacities to model hard constraints, and here some care must be taken to avoid overflow issues. With floats, this can be modeled using infinite capacity. However, floats are less efficient and some algorithms are not guaranteed to terminate with floats due to numerical errors.

TABLE 1
Summary of the Tested Implementations Including Their Memory Footprint

Serial algorithms	Algorithm type	Memory footprint	
HI-PR[a] [56]	Preflow push-relabel	$40n + 40m_T$	$+40m_N$
HPF[b] [32]	Pseudoflow	$104n + 48m_T$	$+48m_N$
EIBFS [31]	Pseudoflow		
↳ `EIBFS`[c]	Pseudoflow	$72n$	$+72m_N$
↳ `EIBFS-I`[*i]	Pseudoflow	$29n$	$+50m_N$
↳ `EIBFS-I-NR`[*i]	Pseudoflow	$49n$	$+24m_N$
BK [1]	Augmenting path		
↳ `BK`[d]	Augmenting path	$48n$	$+64m_N$
↳ `MBK`[*i]	Augmenting path	$23n$	$+24m_N$
↳ `MBK-R`[*i]	Augmenting path	$23n$	$+48m_N$
Parallel algorithms			
P-PPR[ei] [48]	Parallel PPR	$48n + 68m_T$	$+68m_N$
Liu-Sun[*i] [47]	Ada. bot.-up merging[†]	$25n$	$+24m_N$
Strandmark-Kahl[*i] [42]	Dual decomposition[†]	$29n$	$+24m_N$
P-ARD[a] [41]	Region discharge[†]	$40n$	$+32m_N$

The table shows the bytes required as a function of the number of nodes, n, number of terminal arcs, m_T, and number of neighbor arcs, m_N. We assume the common case of 32-bit capacities and 32-bit indices, which is also what we use for all of our experiments. Since HPF stores undirected arcs, we give all sizes as undirected arcs, i.e., for implementations using directed arcs the size per arc reported here is doubled. Note that the numbers depend on, but are not *the same as, the `Node` and `Arc` structure sizes, as the footprint reported includes all stored data (connectivity, capacity, and any auxiliary data).*
[†] *Uses BK (augmenting path).*
[*] *Implemented or updated by us: https://github.com/patmjen/maxflow_algorithms*
[i] *Assuming 32-bit indices*
[a] *https://cmp.felk.cvut.cz/~shekhovt/d_maxflow/index.html*
[b] *https://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html*
[c] *https://github.com/sydbarrett/AlphaPathMoves*
[d] *http://pub.ist.ac.at/~vnk/software.html*
[e] *https://github.com/niklasb/pbbs-maxflow*

As the size of the data structures influences how much the CPU can store in its caches, which has a large effect on performance, it is generally beneficial to keep the data structures small. Note that some compilers do not pack data structures densely by default, which may significantly increase the size of the `Arc` and `Node` data structures.

3.4.2 Indices versus Pointers

One way to reduce the size of the `Arc` and `Node` data structures on 64-bit system architectures is to use indices instead of pointers to reference nodes and arcs. As long as the indices can be stored using unsigned 32-bit integers, we can halve the size arc and node references by using unsigned 32-bit integers instead of pointers (which are 64-bit). This approach can significantly reduce the size of the `Arc` and `Node` data structures, as the majority of the structures consist of references to other arcs and nodes [35]. As the performance of min-cut/max-flow algorithms is mainly limited by memory speed, smaller data structures can often lead to improved performance. The downside of indices is that extra computations may be needed for every look-up, although this depends on the exact assembly instructions the compiler chooses to use.

Some grid-based algorithms [35] use 32-bit indices to reduce the size of their data structure. The generic algorithms we have investigated in this work all use pointers to store references between nodes and arcs. Some implementations avoid the extra memory requirement by compiling with 32-bit pointers. However, 32-bit pointers limit the size of the graph much more than 32-bit indices. The reason is that the 32-bit pointers only have 4 GiB of address space, and the `Node` and `Arc` structures they point to take up many bytes. For example, the smallest `Arc` structure we have tested, *c.f.* Table 1, uses 24 bytes, meaning that an implementation based on 32-bit *indices* could handle graphs with 24 times more arcs than an implementation based on 32-bit *pointers*.

3.4.3 Arc Packing

The order in which the arcs are stored may significantly affect performance. *Arc packing* is used to reduce CPU cache misses by storing the arcs in the same order that the algorithm will access them. For example, min-cut/max-flow algorithms often iterate over outgoing arcs from a node, making it beneficial to store outgoing arcs from the same node adjacent in memory. However, as arcs may be added to the graph in any order, packing the arcs usually incurs an overhead from maintaining the correct ordering or reordering all arcs as an extra step before computing the min-cut/max-flow. Similar to arc packing, *node packing* may improve performance. However, this is not done in practice as opposed to arc packing.

Of the serial reference implementations that we examined, only HI-PR [56], IBFS, and EIBFS implement arc packing. These all implement it as an extra step, where arcs are reordered after building the graph but before the min-cut/max-flow computations start. None of the examined implementations use node packing.

3.4.4 Arc Merging

In practice, it is not uncommon that multiple arcs between the same pair of nodes are added to the graph. Merging these arcs into a single arc with a capacity equal to the sum of capacities of the merged arcs may reduce the graph size significantly. As this decreases both the memory footprint of the graph and the number of arcs to be processed, it can provide substantial performance benefits [35], [41]. However, as redundant arcs can usually be avoided by careful graph construction and they should have approximately the same performance impact on all algorithms, we have not investigated the effects of this further.

4 Parallel Min-Cut/Max-Flow

Like serial algorithms, parallel algorithms for min-cut/max-flow problems can be split into families based on shared characteristics. A key characteristic is whether the algorithms parallelize over individual graph nodes (node-based parallelism) or split the graph into sub-graphs that are then processed in parallel (block-based parallelism). Other important algorithmic traits include whether the algorithm is distributed, which we do not consider in this paper, and the guarantees in terms of convergence, optimality, and completeness provided by the algorithm.

We should note that since many (but not all) min-cut/max-flow problems in computer vision are defined on grid graphs, several algorithms [34], [35], [36], [37] have exploited this structure to create very efficient parallel implementations.

However, many important computer vision problems are not defined on grid graphs, so in this paper we focus on generic min-cut/max-flow algorithms.

The category of node-based parallel algorithms is generally dominated by parallel versions of PPR algorithms. In the block-based category, we have identified three main approaches: adaptive bottom-up merging, dual decomposition, and region discharge, which we investigate. In the following sections, we give an overview of each approach and briefly discuss its merits and limitations.

4.1 Parallel Preflow Push-Relabel

PPR algorithms have been the target of most parallelization efforts [33], [37], [48], [61], [62], [63], [64], since both push and relabel are local operations, which makes them well suited for parallelization. Because the operations are local, the algorithms generally parallelize over each node — performing pushes and relabels concurrently. To avoid data races during these operations, PPR algorithms use either locking [61] or atomic operations [64]. As new excesses are created, the corresponding nodes are added to a queue from which threads can poll them. In [48], a different approach is applied, where pushes are performed in parallel, but excesses and labels are updated later in a separate step, rather than immediately after the push.

Since parallel PPR algorithms parallelize over every node, they can achieve good speed-ups and scale well to modern multi-core processors [48], or even GPUs [37]. However, these algorithms have not achieved dominance outside of large grid graphs for min-cut/max-flow problems [43]. Since GPU hardware has advanced considerably in recent years, it is unclear whether GPU method should remain restricted to grid graphs, but this question is not within the scope of this paper.

4.2 Adaptive Bottom-Up Merging

The adaptive bottom-up merging approach introduced by Liu and Sun [47] uses block-based parallelism and has two phases, which are summarized in Fig. 4. In phase one, the graph is partitioned into a number of disjoint sets (blocks), and arcs between blocks have their capacities set to 0 — effectively removing them from the graph. For each pair of blocks connected by arcs, we store a list of the connecting arcs (with capacities now set to 0) along with their original capacities. Disregarding s and t, the nodes in each block now belong to disjoint sub-graphs and we can compute the min-cut/max-flow solution for each sub-graph in parallel. The min-cut/max-flow computations are done with the BK algorithm — although one could in theory use any min-cut/max-flow algorithm.

In phase two, we merge the blocks to obtain the complete globally optimal min-cut/max-flow. To merge two blocks, we restore the arc capacities for the connecting arcs and then recompute the min-cut/max-flow for the combined graph. This step makes use of the fact that the BK algorithm can efficiently recompute the min-cut/max-flow when small changes are made to the residual graph for a min-cut/max-flow solution [38].

For merges in phase two to be performed in parallel, the method marks the blocks being merged as locked. The computational threads then scan the list of block pairs, which were originally connected by arcs, until they find a pair of unlocked blocks. The thread then locks both blocks, performs the merge, and unlocks the new combined block. To avoid two threads trying to lock the same block, a global lock prevents more than one thread from scanning the list of block pairs at a time.

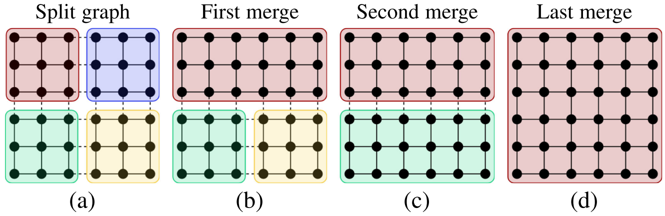


Fig. 4. *Illustration of the adaptive bottom-up merging approach for parallel min-cut/max-flow.* Terminal nodes and arcs are not shown. Note that the underlying graph does *not* have to be a grid graph. Phase one: (a) The graph is split into blocks and the min-cut/max-flow is computed for each block in parallel. Phase two: (b) The topmost blocks are locked, merged, and the min-cut/max-flow recomputed. (c) As the topmost block is locked, the next thread works on the bottom-most blocks (in parallel). (d) Last two blocks are merged and min-cut/max-flow recomputed to achieve the globally optimal solution.

As the degree of parallelism decreases towards the end of phase two — since there are few blocks left to merge — performance increases when computationally expensive merges are performed early in phase two. To estimate the cost of merging two blocks, [47] uses a heuristic based on the potential for new augmenting paths to be formed by merging two blocks. This heuristic determines the merging order of the blocks.

By using block-based, rather than node-based parallelism, adaptive bottom-up merging avoids much of the synchronization overhead that the parallel PPR algorithms suffer from. However, its performance depends on the majority of the work being performed in phase one and in the beginning of phase two, where the degree of parallelism is high.

4.3 Dual Decomposition

The dual decomposition (DD) approach was introduced by Strandmark and Kahl [42] and later refined by Yu *et al*. [43]. The approach was originally designed to allow for distributed computing, such that it is never necessary to keep the full graph in memory. Their algorithm works as follows: first, the nodes of the graph are divided into a set of overlapping blocks (see Fig. 5a). The graph is then split into disjoint blocks, where the nodes in the overlapping regions are duplicated in each block (see Fig. 5b). It is important that the blocks overlap such that if node i is connected to node j in block b_j and node k in block b_k, then i is also in both blocks b_j and b_k.

Once the graph has been partitioned into overlapping blocks, the algorithm proceeds iteratively. First, the min-cut/max-flow for each disjoint block is computed in parallel using the BK algorithm. Next, for each duplicated node, it is checked if all duplicates of that node are in the same s-t partitioned set, S or T. In that case, we say that the node duplicates agree on their assignment. If all duplicated nodes agree on their assignment, the computed solution is globally optimal and the algorithm terminates. If not, the terminal arc capacities for the disagreeing duplicated nodes are updated

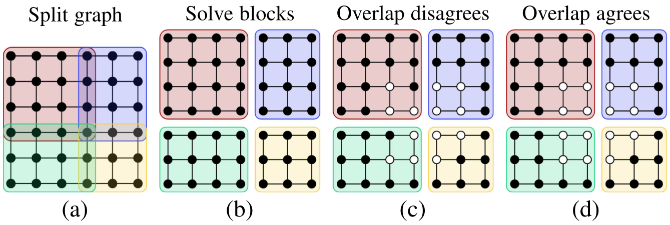


Fig. 5. *Illustration of the dual decomposition approach.* Terminal nodes and arcs are not shown. Note that the underlying graph does *not* have to be a grid graph. (a) Graph nodes are divided into a set of overlapping blocks. (b) The graph is split into disjoint sub-graphs and nodes in overlapping regions are duplicated into each blocks. (c) The min-cut/max-flow for each block is computed in parallel which gives an assignment to source set (black) or sink set (white). The source/sink capacities are then adjusted for disagreeing duplicated nodes. (d) The min-cut/max-flow is recomputed and capacities are adjusted until all duplicated nodes agree.

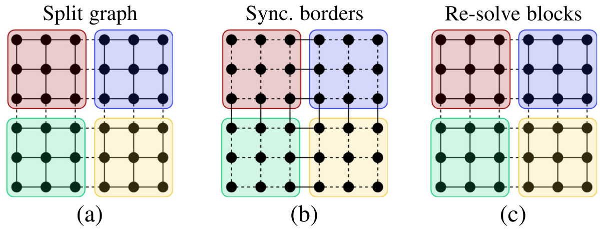


Fig. 6. *Illustration of the region discharge approach.* Terminal nodes and arcs are not shown. Note that the underlying graph does *not* have to be a grid graph. (a) Graph nodes are divided into a set of blocks and the region discharge operation is run on each block, which pushes flow to the sink or boundary. (b) Flow is synchronized between boundaries. (c) Region discharge is run again. The process repeats until no flow crosses the block boundaries.

according to a supergradient[1] ascent scheme and the min-cut/max-flow is recomputed. This process of updating terminal capacities and recomputing the min-cut/max-flow is repeated until all duplicated nodes agree on their assignment.

A limitation of the original dual decomposition approach is that convergence is not guaranteed. Furthermore, [43] and [41] have demonstrated that the risk of nonconvergence increases as the graph is split into more blocks. To overcome this, Yu *et al.* [43] introduced a new version with a simple strategy that guarantees convergence: if the duplicated nodes in two blocks do not belong to the same set, S or T, after a fixed number of iterations, the blocks are merged and the algorithm continues. This trivially guarantees convergence since, in the worst case, all blocks will be merged, at which point the global solution will be computed serially. However, performance significantly drops when merging is needed for the algorithm to converge, as merging only happens after a fixed number of iterations and all blocks may (in the worst case) have to be merged for convergence.

4.4 Region Discharge

The region discharge (RD) approach was introduced by Delong and Boykov [33] and later generalized by Shekhovtsov and Hlaváč [41]. The idea builds on the vertex discharge operation introduced for PPR in [51]. Similarly to DD by Strandmark and Kahl, RD was designed to allow for distributed computing. The method (summarized in Fig. 6) first partitions the graph into a set of blocks (called regions in [41] following the terminology of [33]). Each block R has an associated *boundary* defined as the set of nodes

$$B^R = \{v \in V \mid v \notin R, (u,v) \in E, u \in R, v \neq s,t\}. \quad (5)$$

Capacities for arcs going from a boundary node to a block node are set to zero. This means that flow can be pushed *out* of the block into the boundary, but not vice versa. Furthermore, each node is allowed to have an excess.

The method then performs the *region discharge* operation, which aims to push as much excess flow to the sink and/or the boundary nodes as possible (the source, s, is assumed to have infinite excess). This has been done with a PPR [33], [41] or an AP algorithm (specifically BK) [41]. When using a PPR algorithm, the discharge of a block is done by performing only push and relabel operations between nodes in the same block.

When using the BK algorithm, a distance labeling is maintained for the boundary nodes which gives an estimate of how many boundaries must be crossed to reach the sink. Initially, in each block, flow is pushed exclusively to the sink. Then, flow is pushed to the boundary nodes with distance labels less than 1, then less than 2, etc., until no more flow can be pushed. The BK implementation used by Shekhovtsov and Hlaváč has been slightly modified to allow excess in the boundary nodes and for flow to be pushed from the boundary nodes out of the block (but not back).

The discharge operation is performed on all blocks in parallel. Afterward, flow along boundary arcs is synchronized between neighboring blocks. This may create additional excesses in some blocks, since boundary nodes overlap with another block. The discharge and synchronization process is repeated until no new excesses are created, at which point the algorithm terminates. It is proved in [41] that this process terminates in at most $2n^2$ iterations of discharge and synchronization when using PPR and $2n_B^2+1$ when using AP, where n_B is the total number of boundary nodes.

The guarantee of convergence, without having to merge blocks, is beneficial, as it means that the algorithm can maintain a high degree of parallelism while computing the min-cut/max-flow solution. However, because flow must be synchronized between blocks, the practical performance of the method still depends on well-chosen blocks and may be limited by synchronization overhead. For details on the heuristics used in the algorithm, which are also important for its practical performance, see [41].

5 Performance Comparison

We now compare the performance of the algorithms discussed in the previous sections. For all experiments, the source code was compiled with the GCC C++ compiler version 9.2.0 with `-O3` optimizations on a 64-bit Linux-based operating system with kernel release 3.10. Experiments were run on a dual socket NUMA (Non-Uniform Memory Access) system with two Intel Xeon Gold 6226R processors with 16 cores each and HTT (Hyper-Threading Technology) disabled, for a total of 32 parallel CPU threads. The system has 756 GB of RAM, and for all experiments all data were kept in memory. All resources were provided by the DTU Computing Center [66].

1. Analogous to subgradients for convex functions [65].

For all parallel benchmarks, we prefer local CPU core and memory allocation. This means that for all parallel benchmarks with up to 16 threads, all cores are allocated on the same CPU/NUMA node. If the data fits in the local memory of the active node, we use this memory exclusively. If the data cannot fit in the local memory of one node, memory of both NUMA nodes is used. For benchmarks with more than 16 threads, both CPUs and their memory pools are used.

Run time was measured as the minimum time over three runs and no other processes (apart from the OS) were running during the benchmarks. We split our measured run time into two distinct phases: build time and solve time. Build time refers to the construction of the graph and any additional data structures used by an algorithm. If the algorithm performs arc packing or similar steps, this is included in the build time. To ensure that the build time is a fair representation of the time used by a given algorithm, we precompute a list of nodes and arcs and load these lists fully into memory before starting the timer. Solve time refers to the time required to compute the min-cut/max-flow. For the pseudoflow, PPR, and region discharge algorithms (*c.f.* Table 1), that only compute a minimum cut, we do not include the time to extract the full feasible maximum flow solution. The reason for this is that for most computer vision applications the minimum cut is of principal interest. Furthermore, converting to a maximum flow solution usually only adds a small overhead [46].

5.1 Datasets

We test the algorithms on the following benchmark datasets:

1) The commonly used University of Waterloo [67] benchmarks problems. Specifically, we use 6 stereo [7], [8], 36 3D voxel segmentation [23], [24], [68], 2 multi-view reconstruction [69], [70], and 1 surface fitting [9] problems.
2) The 4 super resolution [17], [71], 4 texture restoration [17], 2 deconvolution [17], 78 decision tree field (DTF) [72], and 3 automatic labelling environment (ALE) [73], [74], [75], [76] datasets from Verma's and Batra's survey [46].
3) New problems that use anisotropic MRFs [77] to segment blood vessels in large voxel volumes from [78]. We include 3 problems where the segmentation is applied directly to the image data and 3 to the output of a trained V-Net [79].
4) New problems that use MRFs to clean 3D U-Net [80] segmentations of prostate images from [81]. We contribute 4 benchmark problems.
5) New problems on mesh segmentation based on [82]. We contribute 8 benchmark problems. The original paper uses α-expansion and $\alpha\beta$-swaps [1], [23] to handle the multi-class segmentation problem. For our benchmarks, we instead use QPBO to obtain the segmentation with a single min-cut, which may lead to different results compared with the referenced method.
6) New problems using the recent Deep LOGISMOS [19] to segment prostate images from [81]. We contribute 8 problems.
7) New problems performing multi-object image segmentation via surface fitting from two recent papers [5], [83]. We contribute 9 problems using [83] and 8 using [5].
8) New problems performing graph matching from the recent paper [16]. The original matching problems can be found at https://vislearn.github.io/libmpopt/iccv2021. For each matching several QPBO sub-problems are solved. We contribute the QPBO subproblems (300 per matching problem) for each of the 316 matching problems.

In total, our benchmark includes 495 problems covering a variety of different computer vision applications. Note that some datasets consist of many small sub-problems that must be run in sequence. Here, we report the accumulated times. All the benchmark problems are available at: DOI:10.11583/DTU.17091101 [84].

For the parallel algorithm benchmarks, we only include a subset of all datasets. This is because parallelization is mainly of interest for large problems with long solve times. For the block-based algorithms, we split the graph into blocks in one of the following ways: For graphs based on an underlying image grid, we define blocks by recursively splitting the image grid along its longest axis. For the surface-based segmentation methods [5], [83], we define blocks such that nodes associated with a surface are in their own block. For mesh segmentation, we compute the geodesic distance between face centers and then use agglomerative clustering to divide the nodes associated with each face into blocks. For bottom-up merging, we use 64 blocks for the following dataset: the grid graphs, the mesh segmentation, and the cells, foam, and simcells. For the NT32_tomo data we use two blocks per object. For 4Dpipe we use a block per 2D slice. For P-GridCut we use the same blocks as for bottom-up merging. For dual decomposition and region discharge, we use one and two blocks per thread, respectively.

5.2 Tested Implementations

All tested implementations (except GridCut [34]) are available at https://github.com/patmjen/maxflow_algorithms and are archived at DOI:10.5281/zenodo.4903945 [52]. Beware that the implementations are published under different licenses — some open and some restrictive. See the links above for more information.

In the following, `typewriter` font refers to a specific implementation of a given algorithm. We use this for BK and EIBFS, where we test more that one implementation of each algorithm, e.g., BK refers to the algorithm, `BK` is the reference implementation, and `MBK` is one of our implementations.

BK [1] We test the reference implementation (`BK`) of the Boykov-Kolmogorov algorithm from http://pub.ist.ac.at/~vnk/software.html. Furthermore, we test our own implementation of BK (`MBK`), which contains several optimizations. Most notably, our version uses indices instead of pointers to reduce the memory footprint of the `Node` and `Arc` data structures. Finally, we test a second version (`MBK-R`), which reorders arcs so that all outgoing arcs from a node are adjacent in memory. This increases cache efficiency, but uses more memory (see Table 1) and requires an extra initialization step. The memory overhead from reordering could be reduced by ordering the arcs in-place; however, this may negatively impact performance. Therefore, we opt for the same sorting strategy as `EIBFS`, where arcs are copied during reordering.

EIBFS [31] We test a slightly modified version [3] (`EIBFS`) of the excesses incremental breadth first search algorithm originally implemented by [31] available from https://github.com/sydbarrett/AlphaPathMoves. This version uses slightly larger data structures to support non-integer arc capacities and larger graphs, compared to the implementation tested in [31]. Although these changes may slightly decrease performance, we think it is reasonable to use the modified version, as several of the other algorithms have made similar sacrifices in terms of performance. Additionally, we test our own modified version of `EIBFS` (`EIBFS-I`), which replaces pointers with indices to reduce the memory footprint. Finally, since both `EIBFS` and `EIBFS-I` perform arc reordering during initialization, we also test a version without arc reordering (`EIBFS-I-NR`) to better compare with other algorithms.

HPF [32] We test the reference implementation of Hochbaum pseudoflow (HPF) from https://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html. This implementation has four different configurations that we test:

1) Highest label with FIFO buckets (`HPF-H-F`).
2) Highest label with LIFO buckets (`HPF-H-L`).
3) Lowest label with FIFO buckets (`HPF-L-F`).
4) Lowest label with LIFO buckets (`HPF-L-L`).

HI-PR [56] We test the implementation of the preflow push-relabel algorithm from https://cmp.felk.cvut.cz/~shekhovt/d_maxflow/index.html.[2]

P-ARD [41] We test the implementation of parallel augmenting paths region discharge (P-ARD) from https://cmp.felk.cvut.cz/~shekhovt/d_maxflow/index.html. P-ARD is an example of the region discharge approach. It uses `BK` as the base solver. Note that, as the implementation is designed for distributed computing, it makes use of disk storage during initialization, which increases the build time.

Liu-Sun [47] Since no public reference implementation is available, we test our own implementation of the adaptive bottom-up merging approach based on the paper by Liu and Sun [47]. Our implementation uses `MBK` as the base solver.

P-PPR [48] We test the implementation of a recent parallel preflow push-relabel algorithm from https://github.com/niklasb/pbbs-maxflow.

Strandmark-Kahl [42] We test our own implementation of the Strandmark-Kahl dual decomposition algorithm based on the implementation at https://cmp.felk.cvut.cz/~shekhovt/d_maxflow/index.html.[3] The original implementation can only handle grid graphs with rectangular blocks, while our implementation can handle arbitrary graphs and arbitrary blocks at the cost of some additional overhead during graph construction. Our implementation uses `MBK` as the base solver. Note that our implementation does not implement the merging strategy proposed by [43] and, therefore, is not guaranteed to converge. We only include results for cases where the algorithm does converge.

2. Orignally from http://www.avglab.com/andrew/soft.html, but the link is no longer available.

3. Originally from https://www1.maths.lth.se/matematiklth/personal/petter/cppmaxflow.php but the link is no longer available.

GridCut [34], [35] We test both the serial and parallel versions of the highly optimized commercial GridCut implementation from https://gridcut.com. The primary goal is to show how much performance can be gained by using an implementation optimized for grid graphs. GridCut is only tested on problems with graph structures that are supported by the reference implementation, i.e., 4- and 8-connected neighbor grids in 2D, and 6- and 26-connected (serial only) neighbor grids in 3D.

Table 1 lists the tested implementations along with their type and memory footprint. Their memory footprint can be calculated based on the number of nodes and arcs in the graph and will be discussed further in Section 7.

5.3 Serial Algorithms

The primary experimental results for the serial algorithms are listed in Table 2 and Fig. 7. Table 2 shows a representative subset of the results, grouped by problem family, while Fig. 7 shows the distribution of the solve time and the total time for each algorithm on each dataset relative to the fastest algorithm on the dataset. Thus, for a given dataset, a relative performance score of 0.5 means that the algorithm used double the amount of time as the fastest algorithm on that dataset. The distribution of these scores indicates how well the different algorithms perform relative to each other.

From Fig. 7b, we see that EIBFS and HPF outperform the other two algorithms on the majority of the datasets in terms of solve time and total time, as the algorithms have most of their relative times close to 1. Looking at the median, EIBFS has a slightly better relative solve time than HPF, while HPF is faster w.r.t. total time. Furthermore, HPF has the best worst-case performance for both solve and total time. However, despite its overall good performance, HPF performs significantly worse on the oriented MRF and U-Net cleaning datasets. The performance of BK varies significantly depending on the benchmark problem. Although it has a median relative total time of just over 0.5, its relative performance is considerably more inconsistent than that of the three other algorithms. It performs particularly poorly on the 4Dpipe datasets, using over 6 hours on 4Dpipe_small, which both HPF and EIBFS completed in less than 30 seconds. For 4Dpipe_large, BK was not able to find the solution within 45 hours. HI-PR generally has the worst performance but does have the fastest solve time for a few datasets. However, measured on total time, it almost never manages a relative score of more than 0.5. It is worth noting that the distribution of relative times for all algorithms exhibits a bimodality. This indicates that all algorithms have datasets where they are poorly suited compared to the others. We further investigate this in Section 6.

5.3.1 Algorithm Variants

The different variants of each algorithm are compared in Fig. 8, which shows the relative performance of each implementation compared to a chosen "reference" implementation. For the BK algorithm, the `BK` implementation is used for reference, for the EIBFS algorithm, the `EIBFS` implementation is used as a reference, and for HPF the `HPF-H-F` configuration is used as reference, since it is the one

TABLE 2
Performance Comparison of Serial Algorithms Based on Both Their Solve and Total (build + solve) Times

Dataset	Nodes	Arcs	Solve	Total	Solve	Total	Solve	Total	Solve	Total	Solve	Total
3D segmentation: voxel-based			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
adhead.n26c100 [23], [24], [68]	12 M	327 M	65.81 s	92.57 s	22.60 s	33.93 s	24.29 s	29.03 s	225.38 s	424.67 s	25.19 s	**27.79 s**
adhead.n6c100 [23], [24], [68]	12 M	75 M	23.88 s	28.03 s	13.23 s	15.85 s	14.13 s	15.87 s	59.65 s	102.84 s	6.98 s	**7.31 s**
babyface.n26c100 [23], [24], [68]	5 M	131 M	82.29 s	92.87 s	30.13 s	**34.74 s**	54.47 s	56.71 s	183.60 s	228.09 s	53.21 s	54.11 s
babyface.n6c100 [23], [24], [68]	5 M	30 M	7.78 s	9.44 s	5.56 s	6.61 s	11.56 s	12.24 s	57.28 s	69.66 s	2.88 s	**3.00 s**
bone.n26c100 [23], [24], [68]	7 M	202 M	9.01 s	25.62 s	9.18 s	16.28 s	4.24 s	7.16 s	68.39 s	173.75 s	4.52 s	**5.88 s**
bone.n6c100 [23], [24], [68]	7 M	46 M	4.09 s	6.65 s	2.74 s	4.35 s	2.30 s	3.36 s	23.66 s	46.71 s	0.91 s	**1.12 s**
bone_subx.n6c100 [23], [24], [68]	3 M	23 M	4.10 s	5.36 s	2.38 s	3.11 s	1.28 s	1.81 s	10.34 s	21.49 s	1.34 s	**1.44 s**
bone_subx.n26c100 [23], [24], [68]	3 M	101 M	7.70 s	15.78 s	4.74 s	8.23 s	2.14 s	**3.61 s**	25.51 s	75.15 s	3.69 s	4.45 s
liver.n26c100 [23], [24], [68]	4 M	108 M	11.78 s	20.41 s	10.49 s	14.20 s	5.72 s	6.50 s	71.88 s	131.00 s	5.62 s	**6.21 s**
liver.n6c100 [23], [24], [68]	4 M	25 M	10.08 s	11.40 s	5.82 s	6.57 s	5.70 s	6.24 s	30.49 s	42.71 s	3.87 s	**3.99 s**
3D segmentation: oriented MRF			MBK [1]		EIBFS-I-NR [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
vessel.orimrf.256 [1], [77], [78]	16 M	66 M	1.84 s	2.95 s	1.13 s	2.03 s	3.19 s	6.80 s	4.11 s	30.99 s	0.40 s	**1.04 s**
vessel.orimrf.512 [1], [77], [78]	134 M	536 M	12.44 s	21.40 s	7.95 s	15.39 s	25.29 s	55.32 s	32.16 s	321.75 s	2.43 s	**7.73 s**
vessel.orimrf.900 [1], [77], [78]	688 M	2 B	75.23 s	121.82 s	48.13 s	88.09 s	147.22 s	300.79 s	177.38 s	1774.65 s	15.97 s	**44.70 s**
3D U-Net segmentation cleaning			MBK [1]		EIBFS-I-NR [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
clean.orimrf.256 [1], [77], [78]	16 M	66 M	0.97 s	2.09 s	0.69 s	1.61 s	3.21 s	6.89 s	3.93 s	30.83 s	0.13 s	**0.77 s**
clean.orimrf.512 [1], [77], [78]	134 M	536 M	7.87 s	17.03 s	5.51 s	13.51 s	27.10 s	58.22 s	31.40 s	320.87 s	0.91 s	**6.27 s**
clean.orimrf.900 [1], [77], [78]	688 M	2 B	35.83 s	81.92 s	25.96 s	64.22 s	130.22 s	280.73 s	163.88 s	1755.87 s	3.90 s	**31.43 s**
unet_mrfclean_2 [1]	8 M	32 M	0.47 s	1.01 s	0.29 s	0.74 s	3.55 s	5.36 s	9.80 s	22.82 s	62 ms	**0.36 s**
unet_mrfclean_3 [1]	15 M	63 M	0.82 s	1.88 s	0.52 s	1.37 s	5.68 s	9.14 s	20.59 s	46.69 s	0.11 s	**0.68 s**
unet_mrfclean_8 [1]	4 M	19 M	0.48 s	0.81 s	0.24 s	0.50 s	2.39 s	3.46 s	6.55 s	13.89 s	0.11 s	**0.28 s**
Surface fitting			MBK [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
LB07-bunny-lrg [9]	49 M	300 M	15.40 s	21.17 s	6.38 s	15.25 s	21.87 s	32.13 s	610.24 s	820.64 s	2.36 s	**3.75 s**
3D segmentation: sparse layered graphs (SLG)			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
4Dpipe_small [5]	14 M	124 M	6.03 h	6.03 h	2.06 s	**15.55 s**	17.91 s	28.85 s	202.49 s	266.01 s	-	-
4Dpipe_big [5]	143 M	1 B	-	-	20.59 s	**195.41 s**	222.09 s	332.06 s	2611.65 s	3436.43 s	-	-
NT32_tomo3_.raw_3 [5]	7 M	49 M	15.42 s	18.69 s	24.22 s	27.19 s	15.87 s	**18.33 s**	176.11 s	200.29 s	-	-
NT32_tomo3_.raw_10 [5]	22 M	154 M	52.86 s	63.15 s	50.82 s	60.01 s	36.46 s	**44.14 s**	645.33 s	741.98 s	-	-
NT32_tomo3_.raw_30 [5]	67 M	462 M	145.23 s	**176.37 s**	194.79 s	221.90 s	179.82 s	202.73 s	2939.04 s	3260.63 s	-	-
NT32_tomo3_.raw_100 [5]	183 M	1 B	778.39 s	860.71 s	553.50 s	627.08 s	520.26 s	**583.76 s**	9732.34 s	2.95 h	-	-
3D segmentation: seperating surfaces			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
cells.sd3 [83]	13 M	126 M	48.23 s	59.03 s	35.24 s	40.66 s	15.52 s	**21.84 s**	98.25 s	167.56 s	-	-
foam.subset.r160.h210 [83]	15 M	205 M	6.05 s	22.02 s	3.21 s	**12.52 s**	17.14 s	26.18 s	15.16 s	145.58 s	-	-
foam.subset.r60.h210 [83]	1 M	24 M	0.62 s	2.58 s	0.39 s	**1.49 s**	1.98 s	3.01 s	1.85 s	12.82 s	-	-
simcells.sd3 [83]	3 M	27 M	9.93 s	12.10 s	2.94 s	**4.12 s**	3.23 s	4.60 s	21.57 s	33.89 s	-	-
Deep LOGISMOS			MBK [1]		EIBFS-I [31]		HPF-H-F [32]		HI-PR [56]		GridCut [34], [35]	
deeplogismos.2 [19]	511 K	4 M	0.15 s	0.25 s	28 ms	**0.21 s**	0.12 s	0.31 s	0.16 s	1.29 s	-	-
deeplogismos.3 [19]	707 K	5 M	0.18 s	0.31 s	41 ms	**0.30 s**	0.18 s	0.45 s	0.24 s	1.90 s	-	-
deeplogismos.7 [19]	989 K	7 M	0.34 s	**0.54 s**	0.26 s	0.66 s	0.29 s	0.69 s	0.36 s	2.86 s	-	-
Super resolution			BK [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
super_res-E1 [17], [71]	10 K	62 K	2 ms	2 ms	1 ms	**2 ms**	2 ms	3 ms	1 ms	7 ms	-	-
super_res-E2 [17], [71]	10 K	103 K	4 ms	5 ms	2 ms	3 ms	2 ms	**3 ms**	2 ms	12 ms	-	-
super_res-Paper1 [17], [71]	10 K	62 K	2 ms	3 ms	1 ms	**2 ms**	2 ms	3 ms	1 ms	7 ms	-	-
superres_graph [17], [71]	43 K	742 K	62 ms	78 ms	10 ms	26 ms	7 ms	**12 ms**	19 ms	0.16 s	-	-
Texture			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
texture-Cremer [17]	44 K	783 K	1.54 s	1.58 s	0.35 s	0.37 s	0.17 s	0.19 s	42 ms	**0.19 s**	-	-
texture-OLD-D103 [17]	43 K	742 K	0.60 s	0.65 s	0.19 s	0.21 s	73 ms	**92 ms**	41 ms	0.19 s	-	-
texture-Paper1 [17]	43 K	742 K	0.65 s	0.69 s	0.19 s	0.21 s	76 ms	**95 ms**	36 ms	0.17 s	-	-
texture-Temp [17]	14 K	239 K	0.22 s	0.23 s	30 ms	34 ms	9 ms	**15 ms**	6 ms	32 ms	-	-
Automatic labelling envrionment (ALE)			MBK-R [1]		EIBFS-I-NR [31]		HPF-L-L [32]		HI-PR [56]		GridCut [34], [35]	
graph_1 (s) [74], [75], [76], [85]	185 K	5 M	16.80 s	18.52 s	0.35 s	**0.79 s**	1.00 s	1.60 s	1.58 s	10.60 s	-	-
graph_2 (s) [74], [75], [76], [85]	175 K	3 M	7.38 s	10.47 s	0.83 s	**1.64 s**	2.25 s	3.55 s	2.91 s	20.87 s	-	-
graph_3 (s) [74], [75], [76], [85]	179 K	7 M	27.68 s	35.55 s	2.69 s	**4.51 s**	4.63 s	6.96 s	6.49 s	43.73 s	-	-
Multi-view			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
BL06-camel-lrg [70]	18 M	93 M	107.53 s	111.42 s	28.55 s	31.54 s	24.44 s	**28.82 s**	291.71 s	337.91 s	-	-
BL06-gargoyle-lrg [70]	17 M	86 M	238.08 s	241.65 s	33.76 s	36.57 s	26.51 s	**30.61 s**	208.27 s	251.10 s	-	-
Deconvolution			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
graph3x3 [17]	2 K	47 K	9 ms	11 ms	3 ms	3 ms	1 ms	**1 ms**	1 ms	5 ms	-	-
graph5x5 [17]	2 K	139 K	62 ms	67 ms	6 ms	9 ms	3 ms	**4 ms**	2 ms	15 ms	-	-
Stereo 1			BK [1]		EIBFS-I-NR [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
BVZ-sawtooth (s) [7]	164 K	796 K	0.91 s	1.16 s	0.58 s	**0.69 s**	1.39 s	1.85 s	7.89 s	12.27 s	-	-
BVZ-tsukuba (s) [7]	110 K	513 K	0.49 s	0.58 s	0.35 s	**0.41 s**	0.66 s	0.84 s	4.69 s	6.64 s	-	-
BVZ-venus (s) [7]	166 K	795 K	1.72 s	2.03 s	1.30 s	**1.44 s**	1.94 s	2.46 s	15.00 s	20.11 s	-	-

TABLE 2
(*Continued*)

Dataset	Nodes	Arcs	Solve	Total	Solve	Total	Solve	Total	Solve	Total	Solve	Total
Stereo 2			BK [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
KZ2-sawtooth (s) [8]	294 K	1 M	2.59 s	3.40 s	1.14 s	**2.02 s**	3.30 s	4.66 s	23.79 s	36.55 s	-	-
KZ2-tsukuba (s) [8]	199 K	1 M	1.41 s	1.84 s	0.71 s	**1.12 s**	1.92 s	2.55 s	20.95 s	27.14 s	-	-
KZ2-venus (s) [8]	301 K	2 M	3.98 s	4.89 s	2.18 s	**3.16 s**	4.70 s	6.21 s	41.63 s	55.60 s	-	-
Decision tree field (DTF)			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
printed_graph1 [72]	20 K	1 M	0.63 s	0.73 s	0.13 s	0.17 s	40 ms	**51 ms**	51 ms	0.25 s	-	-
printed_graph16 [72]	11 K	683 K	0.24 s	0.29 s	44 ms	62 ms	16 ms	**22 ms**	25 ms	0.12 s	-	-
Graph matching: small			BK [1]		EIBFS-I-NR [31]		HPF-L-L [32]		HI-PR [56]		GridCut [34], [35]	
atlas1.dd (s) [16], [86]	1 K	5 K	37 ms	59 ms	34 ms	52 ms	21 ms	**39 ms**	23 ms	0.18 s	-	-
car1.dd (s) [16], [73], [87]	38	131	1 ms	1 ms	0 ms	**1 ms**	1 ms	1 ms	0 ms	3 ms	-	-
hassan1.dd (s) [16], [88], [89]	120	2 K	17 ms	30 ms	5 ms	25 ms	2 ms	**6 ms**	4 ms	65 ms	-	-
matching1.dd (s) [16], [90], [91]	38	380	10 ms	12 ms	5 ms	8 ms	2 ms	**4 ms**	6 ms	14 ms	-	-
Graph matching: big			MBK-R [1]		EIBFS-I [31]		HPF-H-L [32]		HI-PR [56]		GridCut [34], [35]	
pair1.dd (s) [16]	1 K	58 K	1.42 s	1.97 s	0.70 s	0.96 s	92 ms	**0.13 s**	0.82 s	1.68 s	-	-
Mesh segmentation			MBK-R [1]		EIBFS-I [31]		HPF-H-F [32]		HI-PR [56]		GridCut [34], [35]	
bunny.segment [82]	97 K	536 K	0.12 s	0.14 s	63 ms	**75 ms**	68 ms	91 ms	0.20 s	0.30 s	-	-
bunnybig.segment [82]	2 M	13 M	1.01 s	1.59 s	0.62 s	**1.23 s**	1.43 s	2.12 s	4.99 s	9.91 s	-	-
candle.segment [82]	159 K	959 K	87 ms	0.13 s	49 ms	**83 ms**	0.11 s	0.15 s	0.29 s	0.53 s	-	-
candlebig.segment [82]	1 M	5 M	0.51 s	0.72 s	0.26 s	**0.44 s**	0.60 s	0.91 s	2.03 s	3.70 s	-	-
chair.segment [82]	305 K	1 M	0.76 s	0.88 s	0.31 s	0.39 s	0.27 s	**0.37 s**	0.86 s	1.37 s	-	-
chairbig.segment [82]	3 M	26 M	1.62 s	2.89 s	1.02 s	**2.45 s**	3.23 s	4.59 s	9.98 s	20.92 s	-	-
handbig.segment [82]	248 K	1 M	0.15 s	0.19 s	71 ms	**0.11 s**	0.13 s	0.18 s	0.35 s	0.63 s	-	-
handsmall.segment [82]	15 K	69 K	4 ms	5 ms	2 ms	**3 ms**	4 ms	6 ms	10 ms	16 ms	-	-

We show a representative subset of the datasets, which have been grouped according to their problem family. For each problem family we only show the fastest variant of each algorithm measured in total time. The fastest solve time for each dataset has been underlined and the fastest total time has been marked with ***bold face****. Datasets which contain many sub-problems are marked with (s).*

recommended by the authors. As we are now measuring relative to a specific implementation, rather than the fastest implementation as in Fig. 7, it is possible to obtain a relative performance score of more than one.

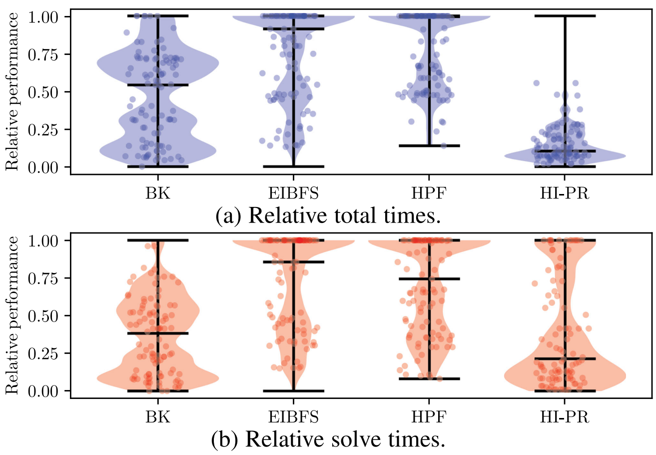
For the BK algorithm, both MBK and MBK-R overall perform similarly or slightly better than BK, when measured on total time. Looking at solve time, MBK-R shows a large speed-up over the other variants. This clearly reflects the effect of arc packing (reordering the arcs), in that it typically decreases solve at the cost of increased build time. From Table 2, we see that BK is generally best for smaller problems where the smaller memory footprint of the index-based variants is less of an advantage. However, the very small difference in absolute time for these small problems will in many cases render the choice of algorithm irrelevant.

For the EIBFS variants, the index-based version (EIBFS-I) consistently outperforms the reference implementation with a median improvement of more than 20%. Meanwhile, EIBFS-I-NR performs worse than EIBFS on almost all datasets w.r.t. to solve time, but better w.r.t. total time for the majority of the problems. In some cases, it also outperforms EIBFS-I, again showing that while arc packing generally significantly reduces the solve time, the additional overhead is not always worth it.

For the HPF algorithm, HPF-H-L consistently performs the best, while HPF-L-F andHPF-L-L perform worse than the reference HPF-H-F for the majority of datasets. However, for some datasets HPF-L-F and HPF-L-L show large speed-ups over the other variants. Table 2 reveals that the HPF-L variants seem to be better for graph matching and ALE datasets.

5.4 Parallel Algorithms

Our benchmark results for the parallel algorithms are shown in Table 3, where we compare the build and solve time for each algorithm on each dataset. The table includes the number of CPU threads used by each algorithm for the listed solve times. Furthermore, it includes the solve time of the best serial algorithm for each dataset for comparison. We focus on the solve time, as that is what reveals how successfully the



(a) Relative total times.

(b) Relative solve times.

Fig. 7. *Relative performance for the serial algorithms.* For each dataset, the solve and total times for each algorithm were compared to those of the fastest algorithm for that dataset and a relative time was computed. This shows how often an algorithm was fastest and, if it was not fastest, how much slower than the fastest it was. We oversample speed-ups from each problem family (*c.f.* Table 2) so all groups have the same number of entries. This is to avoid bias due to some problem groups having more entries than others. Finally, we overlay a random sample of the (oversampled) speed-ups as jittered points.

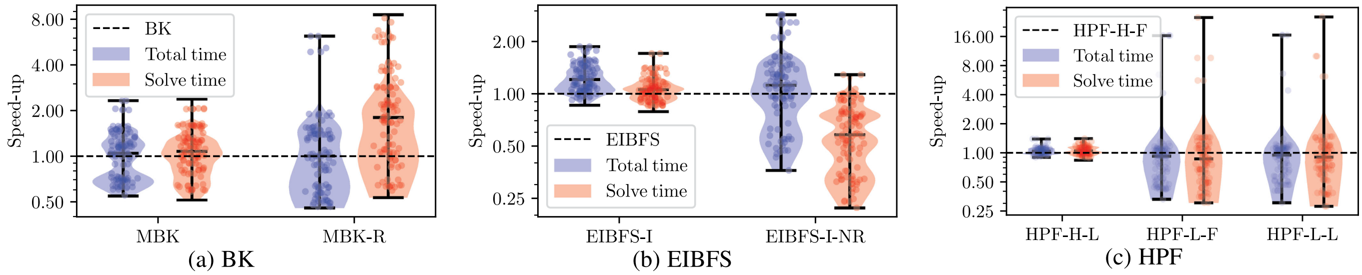


Fig. 8. *Performance comparison of serial algorithm variants.* The solve time and total time is compared against the times for the chosen reference algorithm for each dataset. The violin plots show a Gaussian kernel density estimate of the data and the horizontal bars indicate — from top to bottom — the maximum, median, and minimum. The values were re-sampled as described in Fig. 7.

algorithms distribute the work as more threads are added. Additionally, a lack of optimization leads to very long build times for some of the parallel implementations, especially P-PPR and P-ARD. Finally, some datasets are omitted for P-PPR due to run-time errors and for Strandmark-Kahl due to excessive run time.

From Table 3, it is clear that no algorithm is dominant, except P-GridCut for 6-connected grid graphs. Every algorithm has datasets where it is the fastest and a serial algorithm often gives the best or close to the best performance. The parallel algorithms show their strength for the large datasets with more than 1 M nodes where significant performance improvements are found. Curiously, only P-ARD shows a significant speed-up for smaller problems.

The parallel benchmarks are also summarized in Fig. 9. All algorithms have median speed-ups less than one. Liu-Sun generally performs best, giving a speed-up for almost half of the dataset and having the largest maximum speed-up. P-PPR and P-ARD still provide good speed-ups for some datasets. Strandmark-Kahl comes off the worst, as it rarely beats the best serial algorithm.

Finally, Fig. 10 shows the speed-up distribution of the parallel algorithms compared to their single-threaded performance. Only P-PPR improves consistently as more threads are added. Liu-Sun and P-ARD only show consistent improvements when looking at the maximum speed-up, and for over half of the datasets they have issues scaling beyond 12 threads.

6 Algorithm Selection

As the previous section shows, the performance of the individual min-cut/max-flow algorithms depends on the problem to be solved, i.e., the structure of the graph. Choosing the wrong algorithm may significantly increase the run time. In this section, we investigate strategies for selecting a min-cut/max-flow algorithm that maximize the expected performance given different levels of knowledge about the graph. To quantify the expected performance of a strategy, we will use the relative performance (RP), which we compute as follows: 1. Use the strategy to select an algorithm for each dataset. 2. For each dataset, compute the relative performance of the selected algorithm. For serial algorithms, this is the total time of the selected algorithm divided by the total time of the fastest algorithm for that dataset. For parallel algorithms, we use the solve time. This score shows the expected performance of a given strategy compared to choosing the fastest algorithm.

Scenario 1: No Graph Knowledge. If one has no knowledge of the graph to be solved, the best strategy is to choose the overall best algorithm. Table 4 shows summary statistics for the performance scores of each algorithm. To avoid bias in Fig. 8, we oversample scores from each problem family so that they all have the same number of samples.

For the serial algorithms, the best choice is by far GridCut if it is applicable. It is almost always the fastest option and never more than 36% slower than the best option. Otherwise, the best option is `HPF-H-L` in which case the expected performance 64% of the optimal. Another good option is `EIBFS-I` due to its high mean and high minimum RP scores. All implementations, except `EIBFS`, have a maximum RP of 1, meaning that they outperformed all other implementations on at least one problem instance.

For the parallel algorithms, GridCut again dominates when applicable. Otherwise, the best parallel option is Liu-Sun which is slightly better than P-PPR. Surprisingly, using the best serial algorithm for a dataset is the overall best option, although we should note that comparing to the *best* serial algorithm gives some advantage to the serial algorithms. If one compares to a single serial algorithm, the parallel algorithms do give an improvement — although the mean RP is only 1.8x higher in the best case.

Scenario 2: Known Problem Family. If one knows from which problem family the graph to be solved comes, a good strategy is to select the algorithm that performs well on that problem family. This could, for example, be established beforehand by running a set of benchmarks on example graphs.

Table 5 shows the best performing serial algorithm for each problem family. Note that, as opposed to Table 2, we split graph matching into sub-groups as papers use different energy functions for the matching. For all but four problem families, the best algorithm achieves a mean relative performance of 95% or higher. Furthermore, for most problem families, one algorithm is always the best. This indicates that the problem family is a strong predictor of algorithm performance. The problem family where this strategy performs the worst is 3D segmentation with sparse layered graphs (SLG). Here, the mean RP is only 81%, which is likely due to the large variation in graph size in this problem family.

Table 6 shows the best performing parallel algorithm for each problem family. For the 6-connected graphs, the parallel GridCut algorithm is clearly superior, but otherwise, the different families appear to favor different algorithms.

Scenario 3: Known Graph. Finally, we consider a strategy where the graph is known, but the problem family is not.

TABLE 3
Performance of Parallel Algorithms based on Build and Solve Times

	Liu-Sun [47]			P-PPR [48]			Strandmark-Kahl [42]			P-ARD [41]			P-GridCut [34], [35]			Best serial	
Dataset	Build	Best solve		Build	Best solve		Build	Best solve		Build	Best solve		Build	Best solve		Algo.	Solve
3D segmentation: voxel-based																	
adhead.n26c10 [23], [24], [68]	6.90	17.41	8T	42.71	14.37	12T	26.14	25.40	2T	78.20	35.17	4T	-	-	-	GridCut	**13.21**
adhead.n26c100 [23], [24], [68]	6.86	20.87	8T	41.79	**8.41**	32T	27.65	19.30	6T	75.37	42.91	4T	-	-	-	EIBFS	22.60
babyface.n26c100 [23], [24], [68]	2.89	72.26	32T	15.99	**7.93**	32T	9.15	40.86	4T	32.01	61.20	32T	-	-	-	EIBFS	30.13
bone.n26c100 [23], [24], [68]	4.36	3.68	32T	25.63	4.01	32T	16.38	5.04	8T	48.90	11.21	4T	-	-	-	HPF	**3.48**
bone_subx.n26c100 [23], [24], [68]	2.30	4.04	16T	12.48	2.34	32T	8.03	4.43	8T	24.07	11.71	16T	-	-	-	HPF	**2.14**
liver.n26c10 [23], [24], [68]	2.37	10.79	6T	14.42	7.91	32T	0.89	3.49	1T	21.93	14.95	1T	-	-	-	GridCut	**2.95**
liver.n26c100 [23], [24], [68]	2.36	18.24	6T	13.94	**5.45**	32T	0.86	6.69	1T	27.07	25.74	24T	-	-	-	GridCut	5.62
liver.n6c100 [23], [24], [68]	0.53	7.62	6T	4.78	3.68	16T	0.51	7.24	1T	6.18	7.74	32T	0.11	**2.70**	6T	GridCut	3.87
adhead.n6c100 [23], [24], [68]	1.59	11.17	8T	14.49	4.72	32T	2.52	7.17	4T	15.46	14.41	2T	0.31	**3.83**	4T	GridCut	6.98
babyface.n6c10 [23], [24], [68]	0.66	2.70	32T	5.30	3.09	24T	0.73	3.55	1T	8.87	5.35	1T	0.12	**0.88**	32T	GridCut	1.52
babyface.n6c100 [23], [24], [68]	0.66	5.34	32T	6.53	3.63	24T	0.98	5.24	4T	7.29	7.28	32T	0.12	**1.66**	16T	GridCut	2.88
bone.n6c100 [23], [24], [68]	0.99	0.79	24T	8.30	2.66	32T	1.23	2.01	2T	11.18	2.08	4T	0.20	**0.17**	12T	GridCut	0.91
3D segmentation: oriented MRF																	
vessel.orimrf.256 [1]	1.22	0.69	6T	14.67	1.69	32T	1.89	1.04	2T	16.24	0.82	32T	0.52	**0.14**	12T	GridCut	0.40
vessel.orimrf.512 [1]	9.72	4.92	8T	-	-	-	15.08	6.49	2T	100.95	5.54	16T	4.26	**0.43**	32T	GridCut	2.43
vessel.orimrf.900 [1]	49.62	28.66	8T	-	-	-	79.82	38.45	2T	599.09	24.02	32T	21.98	**2.49**	32T	GridCut	15.97
3D U-Net segmentation cleaning																	
clean.orimrf.256 [1]	1.23	0.39	12T	16.50	2.48	24T	2.39	0.48	4T	15.96	0.84	8T	0.52	**0.06**	32T	GridCut	0.13
clean.orimrf.512 [1]	9.73	2.45	16T	-	-	-	18.26	3.81	4T	131.62	4.81	8T	4.37	**0.27**	32T	GridCut	0.91
clean.orimrf.900 [1]	50.52	12.36	16T	-	-	-	85.77	18.66	4T	578.09	20.17	16T	23.64	**0.82**	32T	GridCut	3.90
unet_mrfclean_3 [1]	1.15	0.27	32T	-	-	-	2.15	0.45	4T	12.66	0.53	4T	0.46	**0.04**	32T	GridCut	0.11
unet_mrfclean_8 [1]	0.37	0.21	8T	-	-	-	0.64	0.23	4T	4.01	0.28	2T	0.14	**0.04**	16T	GridCut	0.11
Surface fitting																	
LB07-bunny-lrg [9]	6.14	1.86	16T	55.88	24.27	32T	7.73	4.14	4T	72.02	4.31	16T	1.24	**0.32**	24T	GridCut	2.36
3D segmentation: sparse layered graphs (SLG)																	
4Dpipe_small [5]	9.93	9.39	12T	-	-	-	-	-	-	47.53	421.60	24T	-	-	-	EIBFS	**2.06**
4Dpipe_big [5]	122.43	86.18	16T	-	-	-	-	-	-	570.44	7485.11	4T	-	-	-	EIBFS	**20.59**
NT32_tomo3_.raw_10 [5]	4.99	18.58	12T	34.90	**14.39**	32T	22.02	85.49	1T	43.95	15.40	12T	-	-	-	HPF	36.46
NT32_tomo3_.raw_30 [5]	14.93	45.70	16T	111.70	59.81	32T	66.56	363.06	1T	132.41	**36.13**	32T	-	-	-	BK	145.23
NT32_tomo3_.raw_100 [5]	38.93	**95.24**	32T	-	-	1T	170.38	1189.78	1T	365.60	158.92	24T	-	-	-	HPF	498.94
3D segmentation: seperating surfaces																	
cells.sd3 [83]	4.18	10.33	16T	25.93	**9.47**	32T	23.90	76.40	1T	21.84	44.98	1T	-	-	-	HPF	15.52
foam.subset.r160.h210 [83]	5.91	8.59	32T	37.08	3.61	32T	52.11	17.24	1T	33.72	7.32	1T	-	-	-	EIBFS	**3.21**
simcells.sd3 [83]	0.69	2.28	16T	5.60	1.99	32T	1.49	2.82	32T	4.96	**0.89**	32T	-	-	-	EIBFS	2.94
Multi-view																	
BL06-camel-lrg [70]	3.99	57.41	8T	-	-	-	1.87	75.56	1T	13.76	95.40	1T	-	-	-	HPF	**24.44**
BL06-gargoyle-lrg [70]	3.70	29.28	16T	-	-	-	1.70	190.07	1T	12.20	102.31	2T	-	-	-	HPF	**26.51**
Mesh segmentation																	
bunnybig.segment [82]	0.30	**0.37**	12T	2.81	1.15	32T	1.12	1.89	1T	3.66	0.41	32T	-	-	-	EIBFS	0.62
chairbig.segment [82]	0.60	0.64	24T	6.10	1.76	32T	2.62	3.29	1T	6.89	**0.57**	32T	-	-	-	EIBFS	1.02
handbig.segment [82]	0.02	0.11	8T	0.25	0.25	16T	0.04	0.16	1T	0.40	0.11	32T	-	-	-	EIBFS	**0.07**

We show a representative subset of the datasets grouped according to their problem family. See Table 2 for the number of nodes and arcs. The algorithms were run with 1, 2, 4, 6, 8, 12, 16, 24, and 32 threads. Only the best time is shown along with the thread count for that run. For comparison, the solve time for the fastest serial algorithm is also included. All times are in seconds. The fastest solve time for each dataset has been marked with bold face.

Here, our strategy is to train a simple decision tree to predict the best algorithm given a feature vector that describes the graph to be solved. Although a single decision tree is not the strongest classifier, it has the benefit of being easily interpretable.

The first components of our feature vector consist of the number of nodes, the number of terminal arcs, the number of neighbor arcs, and whether the graph is a grid graph. Then we include mean, standard deviation, and standard deviation of non-zero values for a number of arc and node properties. For arc properties, we use: source, sink, terminal (source and sink combined), and neighbor capacities. Finally, for node properties we use: sum of in-going neighbor capacities, sum of out-going neighbor capacities, sum of neighbor capacities, degrees, out degrees, and in degrees counts only non-zero arcs. Note that these statistics can be computed efficiently during graph construction. We normalize all capacity statistics by the mean over all arc capacities. In total, our feature vector has 31 entries per graph. Fig. 11 shows a UMAP embedding [94] of the feature vectors for all benchmark datasets. Similar problem families cluster together, despite UMAP receiving no information on this. This suggests the feature vectors provide a good description of the graphs.

We train the decision tree using Scikit-learn [95] version 0.23.1. We use Gini impurity as the split criterion and reduce the tree using minimal cost-complexity pruning [96]. The optimal amount of pruning is determined with 5-fold cross validation. We split each problem family evenly into the folds (if it contains at least 5 datasets). When fitting, each dataset is weighed by one over the number of datasets

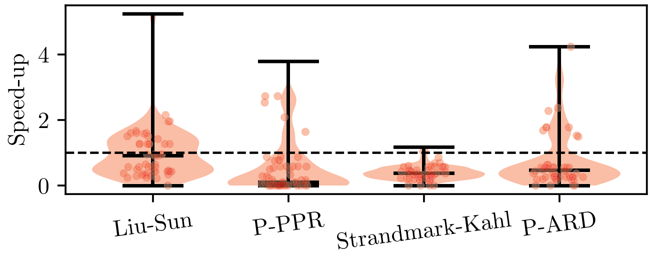


Fig. 9. Speed-up of the parallel algorithms relative to the best serial solve time for each dataset. The values were re-sampled as described in Fig. 7.

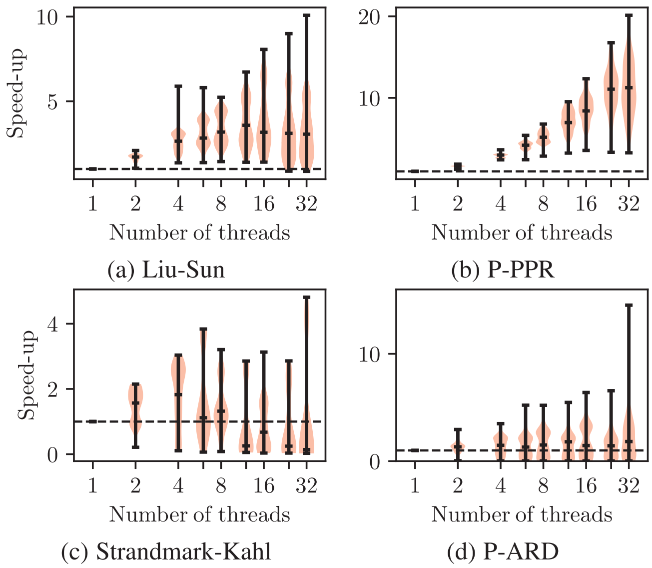


Fig. 10. Speed-up of the the parallel algorithms compared to their single-threaded performance. For each number of threads, the distribution of the speed-ups over all datasets is shown. The values were re-sampled as described in Fig. 7.

in its problem family. When evaluating, we oversample the validation data, so that each problem family has the same number of entries. This indicates how well the decision tree will perform with representative training data. We also perform an additional evaluation where we hold out one problem family, fit on the rest, and then evaluate on the held out family. This indicates how well the decision tree will perform for a problem family that it has not yet encountered. We use the mean RP as validation metric.

We first train a decision tree for the serial algorithms; the result is shown in Fig. 12. It achieves a mean RP of 0.82 and 0.82 in the two evaluations, respectively. This means that the tree is significantly better than naively choosing the overall best algorithm but not as good as knowing the best algorithm for a problem family.

Next, we train a decision tree for the parallel algorithms. We include a category 'Serial', which means that choosing a serial algorithm would be faster. For simplicity, we do not specify which serial algorithm to choose in this scenario. The result is shown in Fig. 13. The decision tree achieves a mean RP of 0.56 and 0.57 in the two evaluations, respectively. Thus, the tree is slightly better than simply choosing the overall best algorithm. However, the best option is to choose the best algorithm for a given category.

TABLE 4
Summary of Relative Performance (RP) Scores for Each of the min-cut/max-flow Algorithm Variants

Serial algorithms	Mean RP $\pm$ Std. RP	Min RP	Max RP
`EIBFS-I`	0.59 ± 0.28	**0.1309**	**1.00**
`EIBFS-I-NR`	0.56 ± 0.32	0.0535	**1.00**
`EIBFS`	0.47 ± 0.23	0.1288	0.94
`HI-PR`	0.16 ± 0.17	0.0046	**1.00**
`HPF-H-F`	0.59 ± 0.33	0.0279	**1.00**
`HPF-H-L`	$\mathbf{0.64 \pm 0.36}$	0.0393	**1.00**
`HPF-L-F`	0.49 ± 0.29	0.0313	**1.00**
`HPF-L-L`	0.53 ± 0.31	0.0312	**1.00**
`MBK-R`	0.27 ± 0.20	0.0006	**1.00**
`BK`	0.27 ± 0.24	0.0005	**1.00**
`MBK`	0.28 ± 0.22	0.0005	**1.00**
GridCut*	$\mathbf{0.99} \pm 0.03$	**0.6419**	**1.00**
Parallel algorithms			
Liu-Sun	$\mathbf{0.48 \pm 0.30}$	0.0667	**1.00**
P-PPR	0.46 ± 0.38	0.0133	**1.00**
Strandmark-Kahl	0.23 ± 0.16	0.0667	0.85
P-ARD	0.35 ± 0.32	0.0028	**1.00**
P-GridCut*	$\mathbf{1.00} \pm \mathbf{0.00}$	**1.0000**	**1.00**
Best serial	$\mathbf{0.59} \pm \mathbf{0.33}$	**0.1365**	**1.00**

* *Only grid graphs included (6- and 26-conn. for serial, 6-conn for parallel). The best score (higher is better) in each column has been marked with* bold *face. Results were oversampled as described in Fig. 7. We only include results where the algorithm ran to completion.*

TABLE 5
Relative Performance (RP) Scores for the Best Serial Algorithm Variant for Each Problem Family

Problem family	Algorithm	Mean RP
3D segmentation: SLG [5]	`HPF-H-L`	0.81
Multi-view [70]	`HPF-H-L`	1.00
Surface fitting [9]	GridCut	1.00
3D segmentation: voxel-based [23], [24], [68]	GridCut	0.98
Mesh segmentation [82]	`EIBFS-I`	0.95
3D segmentation: sep. surfaces [83]	`EIBFS-I`	0.92
3D MRF [1]	GridCut	1.00
Deep LOGISMOS [19]	`EIBFS-I-NR`	0.96
Deconvolution [17]	`HPF-H-L`	0.96
DTF [72]	`HPF-H-L`	1.00
Super resolution [17], [71]	`EIBFS-I`	0.87
Stereo 1 [7]	`EIBFS-I`	0.99
Stereo 2 [8]	`EIBFS-I`	1.00
ALE [75], [76], [85]	`EIBFS-I-NR`	1.00
Graph matching: small [16], [86]	`HPF-L-L`	1.00
Graph matching: small [16], [73], [87]	`EIBFS-I-NR`	0.91
Graph matching: small [16], [88], [89]	`HPF-L-F`	1.00
Graph matching: small [16], [92], [93]	`HPF-L-L`	1.00
Graph matching: small [16], [90], [91]	`HPF-L-F`	1.00
Graph matching: big [16]	`HPF-H-L`	1.00
Mean $\pm$ std.		0.97 ± 0.05

Almost all problem families have one dominant algorithm.

7 Discussion

In this section, we discuss the most interesting findings from our experiments.

7.1 Serial Algorithms

Our results clearly show that GridCut is superior to the other tested algorithms for min-cut/max-flow problems with fixed

TABLE 6
Relative Performance (RP) Scores for the Best Parallel Algorithm for Each Problem Family

Problem family	Algorithm	Mean RP
3D segmentation: SLG [5]	Liu-Sun	0.63
Multi-view [70]	Serial	1.00
Surface fitting [9]	P-GridCut	1.00
3D seg.: voxel-based [23], [24], [68] (26-conn.)	Serial	0.86
3D seg.: voxel-based [23], [24], [68] (6-conn.)	P-GridCut	1.00
Mesh segmentation [82]	P-ARD	0.88
3D segmentation: sep. surfaces [83]	P-PPR	0.74
3D MRF [1]	P-GridCut	1.00
Mean ± std.		0.89 ± 0.14

Since the parallel GridCut implementation can only handle 6-connected graphs '3D segmentation: voxel based' has been split into two subgroups: 6-connected graphs and 26-connected graphs. If an algorithm did not run to completion on a dataset we count the RP as 0.

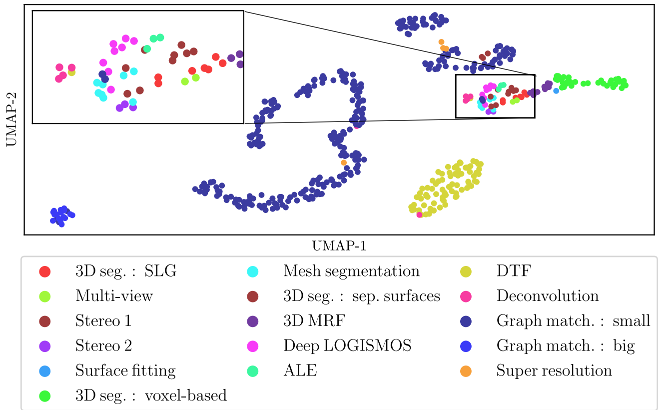


Fig. 11. *UMAP embedding [94] of the extracted graph features.* Each point correspond to a benchmark dataset and is colored according to its problem family. When a benchmark consists of multiple sub-problem we use the mean feature vector. Notice that points from the same problem family tend to cluster together.

neighborhood grids. This is not surprising since GridCut has been designed and optimized specifically for this type of graph. However, as shown in Table 2, the performance benefit of GridCut decreases significantly when moving from 6-connected to 26-connected graphs. Actually, both EIBFS and HPF manage to beat GridCut on a couple of the 26-connected problems. This indicates that the benefit of using GridCut significantly decreases for graphs with high connectivity, perhaps because it is an AP algorithm.

In general, the pseudoflow algorithms have the best overall performance. Measured on solve time, EIBFS performs the best, which aligns with existing literature [31]. However, looking at the total time, HPF performs better overall, slightly contradicting previous benchmarks [31]. The reason for the difference between the results may be that [31] compared EIBFS with `HPF-H-F`, which we show to be inferior to `HPF-H-L`. Also, in [31] they use 32-bit pointer for most datasets, which may also provide slightly improved performance. Finally, the hardware used in [31] may have different performance characteristics than ours. We observed that EIBFS actually performed better on an older system than on the one we used for our experiments. We speculate if this could be due to the lower cache and memory latency (estimated using Intel ® Memory Latency Checker v3.9a) on the

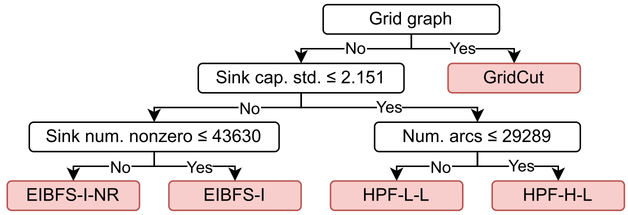


Fig. 12. Decision tree trained to select the best serial algorithm. Note that capacity statistics are normalized, *c.f.* Section 6.

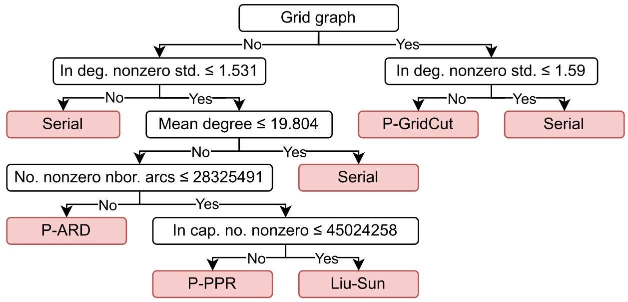


Fig. 13. Decision tree trained to select the best parallel algorithm. 'Serial' means a serial algorithm would be the best option. Note that capacity statistics are normalized, *c.f.* Section 6.

older system compared to the one used for our benchmarks. In any case, this raises the question whether HPF implemented with arc packing could outperform EIBFS on even more problems.

Our results also show that the performance of the different algorithm variants varies, and the choice of variant can significantly affect the run time. Optimizing for cache efficiency seems to be of particular importance, since optimizations such as arc packing and smaller data structures have large effects on the solve times for both BK and EIBFS.

As shown in Section 6, for non-grid problems, the best algorithm most often comes down to a choice between EIBFS or HPF. From Fig. 12 it seems that HPF is faster when the sink (or, more likely, terminal) arc capacities vary a lot. As expected, `EIBFS-I-NR` is preferred for small graphs, while the preferred HPF variant for small graphs appears to be `HPF-L-L`, which aligns with the results in Table 5. However, the best strategy is to test several algorithms on a set of problems from the family at hand.

7.2 Parallel Algorithms

P-GridCut provides the best performance of the parallel algorithms for 6-connected grid graph problems and scales well with many threads. Of the other parallel algorithms, Liu-Sun is overall the best, closely followed by P-PPR, which aligns with previous results [47] and expectations [41], [42]. However, all the block-based algorithms only scale well for large graphs. For small to medium problems, they do not scale to many threads, but seem to peak at 8-12 threads, *c.f.* Fig. 10. This also means that choosing an optimal thread count may be difficult. Only P-PPR scaled consistently with up to 32 threads. In addition, all parallel algorithms were often outperformed by a serial algorithm except on large graphs. In fact, as Table 4 shows, selecting a good serial algorithm has better expected performance than selecting any of the parallel algorithms.

For practical use, only the Liu-Sun, P-PPR, and P-ARD algorithms seem to be relevant as is. However, the block-based algorithms have the additional challenge of dividing the graph into blocks — the result of which significantly affects the run time of the algorithms. This was also shown in [41], where it was noticed that the multiview problems would scale better with more processors when partitioned on vertex numbers versus the grid. While the graphs tested in this work have a natural way to be split, this may not always be the case. Meanwhile, even though this problem is avoided with P-PPR, it does not perform as well as the block-based algorithms overall, as shown in Fig. 9.

Finally, while all parallel algorithms had datasets where they were best, selecting the best parallel algorithm is difficult (except for 6-connected grid graphs). No algorithm showed dominant performance — neither globally nor per problem family. Furthermore, using the decision tree only gives a small improvement over selecting the best overall algorithm. Fig. 13 indicates that for grid and low-degree graphs, a serial algorithm or GridCut performs best. Otherwise, the choice comes down to graph size, with P-ARD doing better for the smaller graphs, P-PPR being faster for the medium-sized ones, and Liu-Sun performing the best for the largest graphs. However, as Table 6 shows, the best strategy is again to test on a number of graphs from the problem family at hand.

8 Conclusions and Perspectives

We now summarize our findings for the serial and parallel algorithms tested in this work. We also provide perspectives on possible future developments of min-cut/max-flow methods, as well as how these may fit into the future of computer vision.

8.1 Serial Algorithms

For the serial min-cut/max-flow algorithms, we have tested a total of 12 different variants across five of the fastest and most popular algorithms: PPR, BK, EIBFS, HPF, and GridCut. These include representatives for the three families of min-cut/max-flow algorithms: augmenting paths, push-relabel, and pseudoflow.

Our results clearly show that, for simple grid graphs, GridCut has the best performance. In most other cases, the two pseudoflow algorithms, EIBFS and HPF, are significantly faster than the other algorithms and thus should be the first choice for anyone looking for a fast serial min-cut/max-flow algorithm for static computer vision problems. For dynamic problems, we refer to [31].

Contrary to existing literature, we recommend the HPF algorithm in the H-LIFO configuration as the default, since it has the best overall performance. However, the EIBFS algorithm (`EIBFS-I` implementation) is a very close contender and can easily replace HPF with little impact on performance — and indeed may perform better on some problem families. If memory usage is of chief concern, the `MBK` and `EIBFS-I-NR` implementations are both good options, as they use significantly less memory than the reference EIBFS and HPF implementations.

Furthermore, we think significant performance improvements may be gained from further improving the algorithm implementations — especially with a focus on memory use and cache efficiency. In particular, faster and more memory efficient methods for arc (and node) packing could result in significant benefits, since the extra initialization step incurs a large memory and run time overhead. We would like to see a reimplementation of HPF with a half-arc data structure and arc packing.

Finally, we found significant gains through automatic algorithm selection. Based on our results, it seems likely that one could train a robust classifier for selecting the appropriate algorithm based on the min-cut/max-flow problem to be solved. By selecting the right algorithm for the job, run time could in many cases be significantly reduced without the need for new algorithms or implementations. In general, we find it unlikely that a single algorithm will ever be dominant for all types of graphs.

8.2 Parallel Algorithms

We tested five different parallel algorithms for min-cut/max-flow problems: parallel PPR (P-PPR), adaptive bottom-up merging (Liu-Sun), dual decomposition (Strandmark-Kahl), region discharge (P-ARD), and parallel GridCut (P-GridCut).

If the graph is a simple grid, P-GridCut significantly outperforms all other algorithms. For other graphs, we found adaptive bottom-up merging, as proposed by Liu and Sun [47], to be the best overall parallel approach. However, each parallel algorithm had an area in which it was the best, and it is difficult to predict the best parallel algorithm for a graph (except for 6-connected grid graphs).

Of the parallel algorithms, only P-GridCut and P-PPR improved consistently with more threads. All block-based algorithms failed to scale beyond 12 threads, except on large graphs. Furthermore, except for P-GridCut, all parallel algorithms were often outperformed by a serial algorithm, and consistent improvements over serial algorithms were obtained only for large graphs. These issues reveal a major deficiency in the state of current parallel min-cut/max-flow algorithms and deserve further study. While providing good scaling on any type of graph may be unreachable as min-cut/max-flow is P-complete and therefore hard to parallelize [97], computer vision graphs often come with additional structure. Therefore, it seems highly likely that further improvements in practical performance can be achieved. However, at this time, we only recommend using a parallel algorithm for graphs with more than 5 M nodes or where a serial algorithm uses at least 5 seconds.

To improve the parallel min-cut/max-flow algorithms, one could try to replace BK, which is currently used in all the tested block-based parallel algorithms, with a pseudoflow algorithm. However, this may not be trivial. In [49], results for a Liu-Sun implementation using EIBFS instead of BK showed a significant performance decrease compared to serial EIBFS. Still, given the superior performance of pseudoflow algorithms, this is an important area to investigate. Furthermore, parallelized graph construction is currently only available for P-GridCut. As the build time is a significant part of the total time, reducing build time will be important — especially as solve time decreases.

Finally, choosing an optimal blocking strategy remains an open problem. Generally, when nodes correspond to spatial positions (e.g., pixels or mesh vertices), we find that grouping based on spatial distance works well. However, we recommend that practitioners experiment with different blocking strategies since it can significantly affect the performance. Furthermore, a general method that only considers the graph structure would be of high interest, as this would also make the algorithms more accessible to the average user. An alternative would be to focus on P-PPR algorithms that do not rely on blocking. Further improvements in these areas could also open the door to GPU-based implementations for solving general min-cut/max-flow problems.

8.3 Min-Cut/Max-Flow in Modern Computer Vision

It is no secret that the field of computer vision is currently dominated by deep learning. In this context, it is highly relevant to consider the future role of traditional computer vision tools, such as min-cut/max-flow algorithms.

For 3D images used in medical imaging and materials science research [98], it is common to have images where no relevant training data are available. Here, segmentation methods based on min-cut/max-flow continue to play an important role, as they work without training data and allow geometric prior knowledge to be incorporated. Furthermore, while modern 3D images can already be very large (many GB per image), dynamic imaging (3D + time) with high acquisition rates is now also possible [99], [100]. Computational efficiency is paramount to be able to process this ever increasing amount of data, and for this, parallel min-cut/max-flow algorithms could prove particularly useful.

Finally, as mentioned in [20], there is agreement that the performance of deep learning-based segmentation methods has started to plateau, and investigating how to integrate CNNs with 'classical' approaches should be pursued. Already, combinations with active contours have shown promising results [101], [102], [103] and a combination of CNNs and min-cut/max-flow methods could lead to new advances. As deep learning involves repeated forward and backward passes through a model, it is crucial that the min-cut/max-flow algorithms are fast and efficient. While not the focus of this work, this is also an area where dynamic min-cut/max-flow algorithms can be of great importance, as they are effective at handling repeated solves of graphs where capacities do not change drastically between successive solves.

References

[1] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 20, pp. 1124–1137, Sep. 2004.

[2] X. Chen and L. Pan, "A survey of graph cuts/graph search based medical image segmentation," *IEEE Rev. Biomed. Eng.*, vol. 11, pp. 112–124, 2018.

[3] H. Isack, O. Veksler, I. Oguz, M. Sonka, and Y. Boykov, "Efficient optimization for hierarchically-structured interacting segments (HINTS)," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 4981–4989.

[4] H. Ishikawa, "Exact optimization for Markov random fields with convex priors," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 10, pp. 1333–1336, Oct. 2003.

[5] N. Jeppesen, A. N. Christensen, V. A. Dahl, and A. B. Dahl, "Sparse layered graphs for multi-object segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 12774–12782.

[6] B. Peng, L. Zhang, and D. Zhang, "A survey of graph theoretical approaches to image segmentation," *Pattern Recognit.*, vol. 46, no. 3, pp. 1020–1038, 2013.

[7] Y. Boykov, O. Veksler, and R. Zabih, "Markov random fields with efficient approximations," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 1998, pp. 648–655.

[8] V. Kolmogorov and R. Zabih, "Computing visual correspondence with occlusions using graph cuts," in *Proc. Int. Conf. Comput. Vis.*, 2001, pp. 508–515.

[9] V. Lempitsky and Y. Boykov, "Global optimization for shape fitting," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2007, pp. 1–8.

[10] J. Egger *et al.*, "Nugget-cut: A segmentation scheme for spherically-and elliptically-shaped 3D objects," in *Proc. Joint Pattern Recognit. Symp.*, 2010, pp. 373–382.

[11] S. Kashyap, H. Zhang, and M. Sonka, "Accurate fully automated 4D segmentation of osteoarthritic Knee MRI," *Osteoarthritis Cartilage*, vol. 25, pp. S227–S228, 2017.

[12] K. Lee *et al.*, "Multiresolution LOGISMOS graph search for automated choroidal layer segmentation of 3D macular OCT scans," in *Proc. SPIE Image Process. Med. Imag.*, 2020, Art. no. 113130B.

[13] K. Li, X. Wu, D. Z. Chen, and M. Sonka, "Optimal surface segmentation in volumetric images-a graph-theoretic approach," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 1, pp. 119–134, Jan. 2006.

[14] Y. Wang and R. Beichel, "Graph-based segmentation of lymph nodes in CT data," in *Proc. Int. Symp. Vis. Comput.*, 2010, pp. 312–321.

[15] Y. Yin, X. Zhang, R. Williams, X. Wu, D. D. Anderson, and M. Sonka, "LOGISMOS–Layered optimal graph image segmentation of multiple objects and surfaces: Cartilage segmentation in the knee joint," *IEEE Trans. Med. Imag.*, vol. 29, no. 12, pp. 2023–2037, Dec. 2010.

[16] L. Hutschenreiter, S. Haller, L. Feineis, C. Rother, D. Kainmüller, and B. Savchynskyy, "Fusion moves for graph matching," in *Proc. Int. Conf. Comput. Vis.*, 2021, pp. 6270–6279.

[17] C. Rother, V. Kolmogorov, V. Lempitsky, and M. Szummer, "Optimizing binary MRFs via extended roof duality," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2007, pp. 1–8.

[18] A. Khoreva, R. Benenson, J. Hosang, M. Hein, and B. Schiele, "Simple does it: Weakly supervised instance and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 1665–1674.

[19] Z. Guo *et al.*, "Deep LOGISMOS: Deep learning graph-based 3D segmentation of pancreatic tumors on CT scans," in *Proc. IEEE Int. Symp. Biomed. Imag.*, 2018, pp. 1230–1233.

[20] S. Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, early access, Feb. 17, 2021, doi: 10.1109/TPAMI.2021.3059968.

[21] N. Tajbakhsh, L. Jeyaseelan, Q. Li, J. N. Chiang, Z. Wu, and X. Ding, "Embracing imperfect datasets: A review of deep learning solutions for medical image segmentation," *Med. Image Anal.*, vol. 63, 2020, Art. no. 101693.

[22] D. M. Greig, B. T. Porteous, and A. H. Seheult, "Exact maximum a posteriori estimation for binary images," *J. Roy. Statist. Soc.. Ser. Methodol.*, vol. 51, no. 2, pp. 271–279, 1989.

[23] Y. Y. Boykov and M.-P. Jolly, "Interactive graph cuts for optimal boundary & region segmentation of objects in N-D images," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2001, pp. 105–112.

[24] Y. Boykov and G. Funka-Lea, "Graph cuts and efficient ND image segmentation," *Int. J. Comput. Vis.*, vol. 70, pp. 109–131, 2006.

[25] X. Wu and D. Z. Chen, "Optimal net surface problems with applications," in *Proc. Int. Colloq. Automata Lang. Program.*, 2002, pp. 1029–1042.

[26] D. Freedman and P. Drineas, "Energy minimization via graph cuts: Settling what is possible," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2005, pp. 939–946.

[27] V. Kolmogorov and R. Zabin, "What energy functions can be minimized via graph cuts?" *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 2, pp. 147–159, Feb. 2004.

[28] E. Boros, P. L. Hammer, and X. Sun, "Network flows and minimization of quadratic pseudo-boolean functions," RUTCOR, Piscataway, NJ, USA, Tech. Rep. 17–1991, 1991.

[29] P. L. Hammer, P. Hansen, and B. Simeone, "Roof duality, complementation and persistency in quadratic 0–1 optimization," *Math. Program.*, vol. 28, no. 2, pp. 121–155, 1984.

[30] V. Kolmogorov and C. Rother, "Minimizing nonsubmodular functions with graph cuts–A review," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 7, pp. 1274–1279, Jul. 2007.

[31] A. V. Goldberg, S. Hed, H. Kaplan, P. Kohli, R. E. Tarjan, and R. F. Werneck, "Faster and more dynamic maximum flow by incremental breadth-first search," in *Proc. Eur. Symp. Algorithms*, 2015, pp. 619–630.

[32] D. S. Hochbaum, "The pseudoflow algorithm: A new algorithm for the maximum-flow problem," *Oper. Res.*, vol. 56, no. 4, pp. 992–1009, 2008.

[33] A. Delong and Y. Boykov, "A scalable graph-cut algorithm for ND grids," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2008, pp. 1–8.

[34] O. Jamriška and D. Sýkora, "GridCut. Version 1.3," 2015, Accessed: Jun. 12, 2020. [Online]. Available: https://gridcut.com

[35] O. Jamriška, D. Sýkora, and A. Hornung, "Cache-efficient graph cuts on structured grids," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2012, pp. 3673–3680.

[36] Y. Peng, L. Chen, F. X. Ou-Yang, W. Chen, and J. H. Yong, "JF-Cut: A parallel graph cut approach for large-scale image and video," *IEEE Trans. Image Process.*, vol. 24, no. 2, pp. 655–666, Feb. 2015.

[37] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2008, pp. 1–8.

[38] P. Kohli and P. H. Torr, "Dynamic graph cuts for efficient inference in Markov random fields," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 12, pp. 2079–2088, Dec. 2007.

[39] M. Yu, S. Shen, and Z. Hu, "Dynamic graph cuts in parallel," *IEEE Trans. Image Process.*, vol. 26, no. 8, pp. 3775–3788, Aug. 2017.

[40] F. Halim, R. H. Yap, and Y. Wu, "A MapReduce-based maximum-flow algorithm for large small-world network graphs," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2011, pp. 192–202.

[41] A. Shekhovtsov and V. Hlaváč, "A distributed mincut/maxflow algorithm combining path augmentation and push-relabel," *Int. J. Comput. Vis.*, vol. 104, no. 3, pp. 315–342, 2013.

[42] P. Strandmark and F. Kahl, "Parallel and distributed graph cuts by dual decomposition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2010, pp. 2085–2092.

[43] M. Yu, S. Shen, and Z. Hu, "Dynamic parallel and distributed graph cuts," *IEEE Trans. Image Process.*, vol. 25, no. 12, pp. 5511–5525, Dec. 2016.

[44] B. G. Chandran and D. S. Hochbaum, "A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem," *Oper. Res.*, vol. 57, no. 2, pp. 358–376, 2009.

[45] B. Fishbain, D. S. Hochbaum, and S. Mueller, "A competitive study of the pseudoflow algorithm for the minimum S–T cut problem in vision applications," *J. Real-Time Image Process.*, vol. 11, no. 3, pp. 589–609, 2016.

[46] T. Verma and D. Batra, "MaxFlow revisited: An empirical comparison of maxflow algorithms for dense vision problems," in *Proc. Brit. Mach. Vis. Conf.*, 2012, pp. 1–12.

[47] J. Liu and J. Sun, "Parallel graph-cuts by adaptive bottom-up merging," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2010, pp. 2181–2188.

[48] N. Baumstark, G. Blelloch, and J. Shun, "Efficient implementation of a synchronous parallel push-relabel algorithm," in *Proc. Eur. Symp. Algorithms*, 2015, pp. 106–117.

[49] N. Jeppesen, P. M. Jensen, A. N. Christensen, A. B. Dahl, and V. A. Dahl, "Faster multi-object segmentation using parallel quadratic pseudo-boolean optimization," in *Proc. Int. Conf. Comput. Vis.*, 2021, pp. 6260–6269.

[50] D. S. Hochbaum and J. B. Orlin, "Simplifications and speedups of the pseudoflow algorithm," *Netw.*, vol. 61, no. 1, pp. 40–57, 2013.

[51] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, 1988.

[52] P. M. Jensen and N. Jeppesen, "Max-flow/min-cut algorithms," Dec. 12, 2021. Accessed: Jun. 08, 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4903945

[53] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, no. 11, pp. 1222–1239, Nov. 2001.

[54] L. R. Ford Jr. and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton Univ. Press, 1962.

[55] A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck, "Maximum flows by incremental breadth-first search," in *Proc. Eur. Symp. Algorithms*, 2011, pp. 457–468.

[56] B. V. Cherkassky and A. V. Goldberg, "On implementing the push—Relabel method for the maximum flow problem," *Algorithmica*, vol. 19, no. 4, pp. 390–410, 1997.

[57] C. Arora, S. Banerjee, P. Kalra, and S. Maheshwari, "An efficient graph cut algorithm for computer vision problems," in *Proc. Eur. Conf. Comput. Vis.*, 2010, pp. 552–565.

[58] A. V. Goldberg, "The partial augment–relabel algorithm for the maximum flow problem," in *Proc. Eur. Symp. Algorithms*, 2008, pp. 466–477.

[59] A. V. Goldberg, "Two-level push-relabel algorithm for the maximum flow problem," in *Proc. Int. Conf. Algorithmic Appl. Manage.*, 2009, pp. 212–225.

[60] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.

[61] R. Anderson and J. C. Setubal, "A parallel implementation of the push-relabel algorithm for the maximum flow problem," *J. Parallel Distrib. Comput.*, vol. 29, no. 1, pp. 17–26, 1995.

[62] D. A. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," Georgia Inst. Technol., CSE Tech. Rep., 2006.

[63] A. V. Goldberg, "Processor-efficient implementation of a maximum flow algorithm," *Inf. Process. Lett.*, vol. 38, no. 4, pp. 179–185, 1991.

[64] B. Hong and Z. He, "An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 1025–1033, Jun. 2011.

[65] S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge, U.K.: Cambridge Univ. Press, 2004.

[66] DTU Computing Center, "DTU computing center resources," 2021. [Online]. Available: https://doi.org/10.48714/DTU.HPC.0001

[67] University of Waterloo, "Max-flow problem instances in vision." Accessed: Feb. 05, 2021. [Online]. Available: https://vision.cs.uwaterloo.ca/data/maxflow

[68] Y. Boykov and V. Kolmogorov, "Computing geodesics and minimal surfaces via graph cuts," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2003, pp. 26–33.

[69] V. Lempitsky, Y. Boykov, and D. Ivanov, "Oriented visibility for multiview reconstruction," in *Proc. Eur. Conf. Comput. Vis.*, 2006, pp. 226–238.

[70] Y. Boykov and V. S. Lempitsky, "From photohulls to photoflux optimization," in *Proc. Brit. Mach. Vis. Conf.*, 2006, Art. no. 27.

[71] W. T. Freeman, E. C. Pasztor, and O. T. Carmichael, "Learning low-level vision," *Int. J. Comput. Vis.*, vol. 40, pp. 25–47, 2000.

[72] S. Nowozin, C. Rother, S. Bagon, T. Sharp, B. Yao, and P. Kohli, "Decision tree fields," in *Proc. Int. Conf. Comput. Vis.*, 2011, pp. 1668–1675.

[73] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL visual object classes challenge 2007 (VOC2007) results." [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2007/

[74] L. Ladický and P. H. Torr, "The automatic labelling environment." Accessed: Nov. 24, 2021. [Online]. Available: https://www.robots.ox.ac.uk/~phst/ale.htm

[75] L. Ladický, C. Russell, P. Kohli, and P. H. S. Torr, "Associative hierarchical CRFs for object class image segmentation," in *Proc. Int. Conf. Comput. Vis.*, 2009, pp. 739–746.

[76] L. Ladický, C. Russell, P. Kohli, and P. H. S. Torr, "Graph cut based inference with co-occurrence statistics," in *Proc. Eur. Conf. Comput. Vis.*, 2010, pp. 239–253.

[77] V. Grau, J. C. Downs, and C. F. Burgoyne, "Segmentation of trabeculated structures using an anisotropic Markov random field: Application to the study of the optic nerve head in glaucoma," *IEEE Trans. Med. Imag.*, vol. 25, no. 3, pp. 245–255, Mar. 2006.

[78] M. Reichardt *et al.*, "3D virtual histopathology of cardiac tissue from Covid-19 patients based on phase-contrast X-ray tomography," *Elife*, vol. 10, 2021, Art. no. e71359.

[79] F. Milletari, N. Navab, and S.-A. Ahmadi, "V-Net: Fully convolutional neural networks for volumetric medical image segmentation," in *Proc. Int. Conf. 3D Vis.*, 2016, pp. 565–571.

[80] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3D U-Net: Learning dense volumetric segmentation from sparse annotation," in *Proc. Int. Conf. Med. Image Comput. Comput. Assist. Interv.*, 2016, pp. 424–432.

[81] A. L. Simpson *et al.*, "A large annotated medical image dataset for the development and evaluation of segmentation algorithms," 2019, *arXiv:1902.09063*.

[82] L. Liu, Y. Sheng, G. Zhang, and H. Ugail, "Graph cut based mesh segmentation using feature points and geodesic distance," in *Proc. Int. Conf. Cyberworlds*, 2015, pp. 115–120.

[83] P. M. Jensen, A. B. Dahl, and V. A. Dahl, "Multi-object graph-based segmentation with non-overlapping surfaces," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2020, pp. 976–977.

[84] P. M. Jensen, N. Jeppesen, A. B. Dahl, and V. A. Dahl, "Min-cut/max-flow problem instances for benchmarking," Mar. 03, 2022. Accessed: Nov. 29, 2021. [Online]. Available: https://doi.org/10.11583/DTU.17091101

[85] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL visual object classes challenge 2010 (VOC2010) results." [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2010/

[86] D. Kainmueller, F. Jug, C. Rother, and G. Myers, "Active graph matching for automatic joint segmentation and annotation of C. elegans," in *Proc. Int. Conf. Med. Image Comput. Comput. Assist. Interv.*, 2014, pp. 81–88.

[87] M. Leordeanu, R. Sukthankar, and M. Hebert, "Unsupervised learning for graph matching," *Int. J. Comput. Vis.*, vol. 96, pp. 28–45, 2012.

[88] H. A. Alhaija, A. Sellent, D. Kondermann, and C. Rother, "GraphFlow–6D large displacement scene flow via graph matching," in *Proc. German Conf. Pattern Recognit.*, 2015, pp. 285–296.

[89] P. Swoboda, C. Rother, H. Abu Alhaija, D. Kainmuller, and B. Savchynskyy, "A study of lagrangean decompositions and dual ascent solvers for graph matching," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 7062–7071.

[90] N. Komodakis and N. Paragios, "Beyond loose LP-relaxations: Optimizing MRFs by repairing cycles," in *Proc. Eur. Conf. Comput. Vis.*, 2008, pp. 806–820.

[91] J. H. Kappes *et al.*, "A comparative study of modern inference techniques for structured discrete energy minimization problems," *Int. J. Comput. Vis.*, vol. 115, pp. 155–184, 2015.

[92] L. Torresani, V. Kolmogorov, and C. Rother, "Feature correspondence via graph matching: Models and global optimization," in *Proc. Eur. Conf. Comput. Vis.*, 2008, pp. 596–609.

[93] T. S. Caetano, J. J. McAuley, L. Cheng, Q. V. Le, and A. J. Smola, "Learning graph matching," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 6, pp. 1048–1058, Jun. 2009.

[94] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform manifold approximation and projection for dimension reduction," 2018, *arXiv:1802.03426*.

[95] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[96] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Evanston, IL, USA: Routledge, 2017.

[97] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation: P-Completeness Theory*, London, U.K.: Oxford Univ. Press, 1995.

[98] E. Maire and P. J. Withers, "Quantitative X-ray tomography," *Int. Mater. Rev.*, vol. 59, no. 1, pp. 1–43, 2014.

[99] F. Garcia-Moreno *et al.*, "Using X-ray tomoscopy to explore the dynamics of foaming metal," *Nature Commun.*, vol. 10, no. 1, pp. 1–9, 2019.

[100] R. Mokso *et al.*, "GigaFRoST: The gigabit fast readout system for tomography," *J. Synchrotron Radiat.*, vol. 24, no. 6, pp. 1250–1259, 2017.

[101] L. Guo, Y. Liu, Y. Wang, Y. Duan, and X.-C. Tai, "Learned snakes for 3D image segmentation," *Signal Process.*, vol. 183, 2021, Art. no. 108013.

[102] S. Peng, W. Jiang, H. Pi, X. Li, H. Bao, and X. Zhou, "Deep snake for real-time instance segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 8530–8539.

[103] U. Wickramasinghe, E. Remelli, G. Knott, and P. Fua, "Voxel2Mesh: 3D mesh model generation from volumetric data," in *Proc. Int. Conf. Med. Image Comput. Comput. Assist. Interv.*, 2020, pp. 299–308.

Patrick M. Jensen received the BScEng and MScEng degrees in applied mathematics from the Technical University of Denmark (DTU), Kgs. Lyngby, Denmark, in 2017 and 2019, respectively. He is currently working toward the PhD degree in 3D image analysis with the Visual Computing Group, Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kgs. Lyngby, Denmark. His research interests lie in 3D image segmentation.

Niels Jeppesen received the PhD degree in image analysis of 3D structures from the Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU), Kgs. Lyngby, Denmark. He is an image analysis and machine learning specialist with FORCE Technology. His research interests lie in min-cut/max-flow algorithms and quantitative analysis of structures in 3D images. He applies these methods for automated quality control of structures and materials, in particular, in the wind turbine industry.

Anders B. Dahl is professor in 3D image analysis, and a head of the Section for Visual Computing, Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU), Kgs. Lyngby, Denmark. He is heading the Center for Quantification of Imaging Data from MAX IV, focusing on quantitative analysis of 3D images. His research is focused on image segmentation and its applications.

Vedrana A. Dahl is an associate professor with the Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU), Kgs. Lyngby, Denmark. Her primary research interest include the use of geometric models for the analysis of volumetric data. This includes volumetric segmentation and methods based on deformable meshes. She developed analysis tools with applications in material science, industrial inspection, and biomedicine.