Author's freely distributed eddition. The final publication is available at http://www.springer.com
Int J Comput Vis DOI: 10.1007/s11263-012-0571-2

A Distributed Mincut/Maxflow Algorithm Combining Path Augmentation and Push-Relabel

Alexander Shekhovtsov · Václav Hlaváč

Recieved: / Accepted:

Abstract We propose a novel distributed algorithm for the minimum cut problem. Motivated by applications like volumetric segmentation in computer vision, we aim at solving large sparse problems. When the problem does not fully fit in the memory, we need to either process it by parts, looking at one part at a time, or distribute across several computers. Many MINCUT/MAXFLOW algorithms are designed for the shared memory architecture and do not scale to this setting. We consider algorithms that work on disjoint regions of the problem and exchange messages between the regions. We show that the region push-relabel algorithm of Delong and Boykov (2008) uses $\Theta(n^2)$ rounds of message exchange, where n is the number of vertices. Our new algorithm performs path augmentations inside the regions and push-relabel style updates between the regions. It uses asymptotically less message exchanges, $O(\mathcal{B}^2)$, where $\mathcal{B}$ is the set of boundary vertices. The sequential and parallel versions of our algorithm are competitive with the state-of-the-art in the shared memory model. By achieving a lower amount of message exchanges (even asymptotically lower in our synthetic experiments), they suit better for solving large problems using a disk storage or a distributed system.

Keywords maximum flow · minimum cut · distributed · parallel · push-relable · augmenting path · large scale

This work was supported by the EU project FP7-ICT-247870 NIFTi, FP7-ICT-247525 HUMAVIPS and GACR P103/10/0783.

Alexander Shekhovtsov · Václav Hlaváč
Czech Technical University in Prague
Center for Machine Perception
Technicka 2, 166 27 Praha 6, Czech Republic
http://cmp.felk.cvut.cz
E-mail: shekhole@fel.cvut.cz
hlavac@fel.cvut.cz

1 Introduction

Minimum s-t cut (MINCUT) is a classical combinatorial problem with applications in many areas of science and engineering. This research was motivated by the wide use of MINCUT/MAXFLOW problems in computer vision, where large sparse instances need to be solved. We start by a more detailed overview of models and optimization techniques in vision, where the MINCUT problem is employed and give examples of our test problems.

MINCUT in Computer Vision. In some cases, an applied problem is formulated directly as a MINCUT. More often, however, MINCUT problems in computer vision originate from the Energy minimization framework (maximum a posteriori solution in a Markov random field model). Submodular Energy minimization problems completely reduce to MINCUT (Ishikawa 2003; Schlesinger and Flach 2006). When the energy minimization is intractable, MINCUT is employed in relaxation and local search methods. The linear relaxation of pairwise Energy minimization with 0-1 variables reduces to MINCUT (Boros et al 1991; Kolmogorov and Rother 2007) as well as the relaxation of problems reformulated in 0-1 variables (Kohli et al 2008). Expansion-move, swap-move (Boykov et al 1999) and fusion-move (Lempitsky et al 2010) algorithms formulate a local improvement step as a MINCUT problem.

Many applications of MINCUT in computer vision use graphs of a regular structure, with vertices arranged into an N-D grid and edges uniformly repeated, *e.g.*, 3D segmentation models illustrated in Fig. 1(c), 3D reconstruction models, Fig. 1(b). Because of such regular structure, the graph itself need not be stored in the memory, only the edge capacities, allowing relatively large instances to be solved by a specialized implemen-

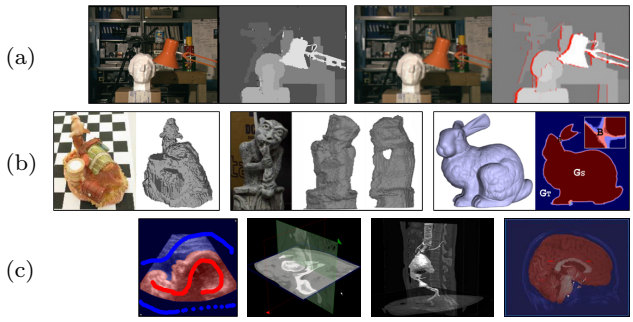


Fig. 1 Examples of labeling problems in computer vision solved via maxflow. (a) Stereo and stereo with occlusions (Boykov et al 1998), (Kolmogorov and Zabih 2001). (b) 3D reconstruction (Lempitsky et al 2006; Boykov and Lempitsky 2006) and surface fitting (Lempitsky and Boykov 2007). (c) 3D segmentation (Boykov and Jolly 2001; Boykov and Funka-Lea 2006; Boykov and Kolmogorov 2003). The instances are published at the University of Western Ontario web pages (2008) for benchmarking maxflow implementations.

tation. However, in many cases, it is advantageous to have a non-regular structure, *e.g.*, in stereo with occlusions in Fig. 1(a), in 3D reconstruction with adaptive tetrahedral volume (Labatut et al 2009; Jancosek and Pajdla 2011). Such applications would benefit from a large-scale generic MINCUT solver.

Distributed Computation. The previous research mostly focused on speeding up MINCUT by parallel computation in the shared memory model. We consider a *distributed* memory model, which assumes that the computation units have their own separate memory and exchanging the information between them is expensive. A distributed algorithm has therefore to divide the computation and the problem data between the units and keep the communication rate low. We will consider distributed algorithms, operating in the following two practical usage modes:

- Sequential (or *streaming*) mode, which uses a single computer with a limited memory and a disk storage, reading, processing and writing back a portion of data at a time.
- Parallel mode, in which the units are thought as computers in a network.

We propose new algorithms for both cases, prove their correctness and termination guarantees. In the assessment of complexity, we focus on the costly operations such as load-unload of the data in the streaming mode or message exchanges in the parallel mode. More specifically, we call a *sweep* the event when all units of a distributed algorithm recalculate their data once. Sweeps in our algorithms correspond to outer iterations and their number is roughly proportional to the amount of communication in the parallel mode or disk operations in the streaming mode. While there are algorithms with better bounds in terms of elementary operations, our algorithms achieve lower communication rates.

Previous Work. A variant of path augmentation algorithm was shown by Boykov and Kolmogorov (2004) to have the best performance on computer vision problems among sequential solvers. There were several proposals how to parallelize it. Partially distributed implementation (Liu and Sun 2010) augments paths within disjoint regions first and then merges regions hierarchically. In the end, it still requires finding augmenting paths in the whole problem. Therefore, it cannot be used to solve a large problem by distributing it over several computers or by using a limited memory and a disk storage. For the shared memory model Liu and Sun (2010) reported a near-linear speed-up with up to 4 CPUs for 2D and 3D segmentation problems.

Strandmark and Kahl (2010) obtained a distributed algorithm using a dual decomposition approach. The subproblems are MINCUT instances on the parts of the graph (regions) and the master problem is solved using the subgradient method. This approach requires solving MINCUT subproblems with real valued capacities and does not have a polynomial bound on the number of iterations. The integer algorithm proposed by Strandmark and Kahl (2010) is not guaranteed to terminate. Our experiments (§7.3) showed that it did not terminate on some of the instances in 1000 sweeps. In §10, we relate dual variables in this method to flows.

The push-relabel algorithm (Goldberg and Tarjan 1988) performs many local atomic operations, which makes it a good choice for a parallel or distributed implementation. A distributed version (Goldberg 1991) runs in $O(n^2)$ time using $O(n)$ processors and $O(n^2\sqrt{m})$ messages, where n is the number of vertices and m is the number of edges in the problem. However, for a good practical performance it is crucial to implement gap relabel and global relabel heuristics (Cherkassky and Goldberg 1994). The global relabel heuristic can be parallelized (Anderson and Setubal 1995), but it is difficult to distribute. We should note, however, that the global relabel heuristic was not essential in the experiments with computer vision problems we made (§7.2). Delong and Boykov (2008) proposed a coarser granulation of push-relabel operations, associating a subset of vertices (a region) to each processor. Push and relabel operations inside a region are decoupled from the rest of the graph. This allows to process several non-interacting regions in parallel or run in a limited memory, processing few regions at a time. The gap and relabel heuristics, restricted to the regions (Delong and Boykov 2008) are powerful and distributed at the same time. Our work was largely motivated by De-

long and Boykov (2008) and the notice that their approach might be extendible to augmenting path algorithms. However, our first attempt to prioritize augmentation to the boundary vertices by the shortest distance to the sink did not lead to a correct algorithm.

Contribution. In this work we revisit the algorithm of Delong and Boykov (2008) for the case of a fixed partition into regions. We study a sequential variant and a novel parallel variant of their algorithm, which allows computation on neighboring interacting regions to run concurrently using a conflict resolution similar to the asynchronous parallel push-relabel (Goldberg 1991). We prove that both variants have a tight $O(n^2)$ bound on the number of sweeps. The new algorithm we construct works with the same partition of the graph into regions but is guided by a different distance function than the push-relabel one.

Given a fixed partition into regions, we introduce a distance function which counts the number of region boundaries crossed by a path to the sink. Intuitively, it corresponds to the amount of costly operations – network communications or loads-unloads of the regions in the streaming mode. The algorithm maintains a labeling, which is a lower bound on the distance function. Within a region, we first augment paths to the sink and then paths to the boundary vertices prioritized by the lowest label. Thus the flow is pushed out of the region in the direction given by the distance estimate. We present a sequential and parallel versions of the algorithm which terminate in $O(|\mathcal{B}|^2)$ sweeps, where $\mathcal{B}$ is the set of all boundary vertices (incident to inter-region edges).

The proposed algorithms are evaluated on instances of MINCUT problems collected and published by the Computer Vision Research Group at the University of Western Ontario (illustrated in Fig. 1). The results are compared against the state-of-the-art sequential and parallel solvers. We also studied the behavior of the algorithms w.r.t. problem size, granularity of the partition, *etc*. Our implementation is publicly available at `http://cmp.felk.cvut.cz/~shekhovt/d_maxflow`.

Other Related Work. The following works do not consider a distributed implementation but are relevant to our design. The Partial Augment-Relabel algorithm (PAR) by Goldberg (2008) augments in each step a path of length k. It may be viewed as a lazy variant of push-relabel, where actual pushes are delayed until it is known that a sequence of k pushes can be executed. The algorithm by Goldberg and Rao (1998) incorporates the notion of a length function and a valid labeling w.r.t. this length. It can be seen that the labeling maintained by our algorithm corresponds to the length function assigning 1 to boundary edges and 0 to intra-region edges. Goldberg and Rao (1998) used such generalized labeling in the context of the blocking flow algorithm but not within the push-relabel.

2 MINCUT and Push-Relabel

We solve MINCUT problem by finding a maximum preflow[1]. In this section, we give basic definitions and introduce the push-relabel framework of Goldberg and Tarjan (1988). While we assume the reader is familiar with mincut/maxflow, we explain some known results using the notation adjusted for the needs of this paper.

In the classical framework of minimum cut and maximum flow, the flow augmentation transforms a minimum cut problem into an equivalent one on the residual network (preserving costs of all cuts up to a constant). However, there is no equivalent minimum cut problem corresponding to an augmentation of a *preflow*. In the push-relabel approach of Goldberg and Tarjan (1988), this is not essential, as only single residual arcs need to be considered and algorithms can be formulated as working with a pair of a network and a preflow. In this paper, we need to work with residual paths and the reachability in the residual network. We therefore use the extended definition of the minimum cut problem, which includes a flow excess (or supply) in every vertex. After this extension, the family of equivalent mincut problems becomes closed under preflow augmentations. This allows us to formulate algorithms more conveniently as working with the current residual network and constructing a preflow increment. This point is illustrated in Fig. 2.

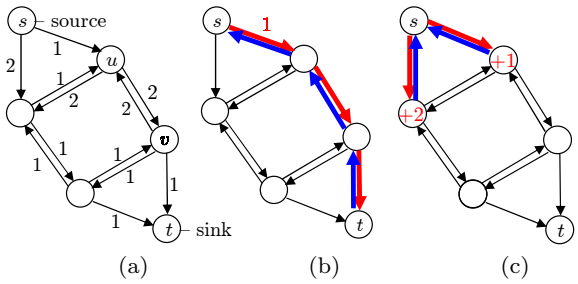


Fig. 2 (a) Example of a network with indicated edge capacity function. (b) Augmenting path approach: send flow from the source to the sink along a path. The residual network defines an equivalent min-cut problem. (c) Push-relabel approach: the preflow is pushed over arcs in all directions, prioritized by the shortest distance to the sink. The equivalent min-cut problem is defined by a network with excess.

[1] A maximum preflow can be completed to a maximum flow using the flow decomposition, in $O(m \log m)$ time. Because we are primarily interested in the minimum cut, we do not consider this step or whether it can be distributed.

By a *network* we call the tuple $G = (V, E, s, t, c, e)$, where V is a set of vertices; $E \subset V \times V$ is the set of edges, thus (V, E) is a directed graph; $s, t \in V$, $s \neq t$, are *source* and *sink*, respectively; $c\colon E \to \mathbb{N}_0$ is a capacity function; and $e\colon V \to \{0, 1, \ldots, \infty\}$, $e(t) = 0$, $e(s) = \infty$ is an *excess* function. We also denote $n = |V|$ and $m = |E|$.

For any sets $X, Y \subset V$ we will denote $(X, Y) = E \cap (X \times Y)$. For $C \subset V$ such that $s \in C$, $t \notin C$, the set of edges $(C, \bar{C})$, with $\bar{C} = V \backslash C$ is called an *s-t cut*. The MINCUT problem is

$$\min \Big\{ \sum_{(u,v)\in(C,\bar{C})} c(u,v) + \sum_{v\in\bar{C}} e(v) \,\Big|\, C \subset V,\ s \in C,\ t \in \bar{C} \Big\}. \quad (1)$$

The objective is called the *cost* of the cut. Note, that excess in this problem can be equivalently represented as additional edges from the source, but we prefer the explicit form. Without a loss of generality, we assume that E is symmetric – if not, the missing edges are added and assigned a zero capacity.

A *preflow* in G is the function $f\colon E \to \mathbb{Z}$ satisfying the following constraints:

$$f(u,v) \leq c(u,v) \quad \forall (u,v) \in E; \quad (2a)$$

(*capacity constraint*)

$$f(u,v) = -f(v,u) \quad \forall (u,v) \in E; \quad (2b)$$

(*antisymmetry*)

$$e(v) + \sum_{u \,|\, (u,v)\in E} f(u,v) \geq 0 \quad \forall v \in V. \quad (2c)$$

(*preflow constraint*)

The constraint (2b) removes the redundancy in the otherwise independent flow values on (u, v) and (v, u) (positive flows should naturally cancel each other) and shortens the equations at the same time.

A *residual network* w.r.t. preflow f is a network $G_f = (V, E, s, t, c_f, e_f)$ with the capacity and excess functions given by

$$c_f = c - f, \quad (3a)$$

$$e_f(v) = e(v) + \sum_{u \,|\, (u,v)\in E} f(u,v), \quad \forall v \in V \backslash \{s,t\}. \quad (3b)$$

By constraints (2) it is $c_f \geq 0$ and $e_f \geq 0$. The costs of all *s-t* cuts differ in G and G_f by a constant called the *flow value*, $|f| = \sum_{u \,|\, (u,t)\in E} f(u,t)$. This can be easily verified by substituting c_f and e_f into (1) and expanding. Network G_f is thus *equivalent* to network G up to the constant $|f|$ and since all cuts in G_f are non-negative, $|f|$ is a lower bound on the cost of a cut in G. The problem of maximizing this lower bound, *i.e.* finding a maximum preflow:

$$\max_f |f| \quad \text{s.t. constraints (2)} \quad (4)$$

is dual to MINCUT. The value of a maximum preflow equals to the cost of a minimum cut and the solutions are related as explained below.

We say that $w \in V$ is *reachable* from $v \in V$ in the network G_f if there is a path (possibly of length 0) from v to w composed of edges with strictly positive residual capacities c_f (a *residual path*). This relation is denoted by $v \to w$.

Let us consider a residual path from v to w such $e_f(v) > 0$. *Augmentation* increases the flow by $\Delta > 0$ on all forward edges of the path, and decreases it on all reverse edges, where Δ does not exceed the residual capacities of the forward arcs or $e_f(v)$. In the result, the excess $e_f(v)$ is decreased and excess $e_f(w)$ is increased. Augmenting paths to the sink increases the flow value. In the augmenting path approach, the problem (4) is solved by repeatedly augmenting residual paths from vertices having excess (*e.g.*, source) to the sink.

If w is not reachable from v in G_f we write $v \nrightarrow w$. For any $X, Y \subset V$, we write $X \to Y$ if there exist $x \in X$, $y \in Y$ such that $x \to y$. Otherwise we write $X \nrightarrow Y$.

A preflow f is maximum iff there is no residual path to the sink which can be augmented. This can be written as $\{v \,|\, e_f(v) > 0\} \nrightarrow t$ in G_f. In this case the cut $(\bar{T}, T)$ with $T = \{v \in V \,|\, v \to t \text{ in } G_f\}$ has value 0 in G_f. Because all cuts are non-negative it is a minimum cut.

2.1 General Push-relabel

A *Distance* function $d^*\colon V \to \{0, 1, \ldots, n\}$ in G_f assigns to $v \in V$ the length of the shortest residual path from v to t, or n if no such path exists. A shortest path cannot have loops, thus its length is not greater than $n - 1$. Let us denote $d^\infty = n$.

A *labeling* $d\colon V \to \{0, 1, \ldots, d^\infty\}$ is *valid* in G_f if $d(t) = 0$ and $d(u) \leq d(v) + 1$ for all $(u, v) \in E$ such that $c_f(u, v) > 0$. Any valid labeling is a lower bound on the distance d^* in G_f, however not every lower bound is a valid labeling.

A vertex v is called *active* w.r.t. (f, d) if $e_f(v) > 0$ and $d(v) < d^\infty$.

The definitions of reachability and validity are given w.r.t. the residual network G_f, however expressions like "$v \to w$ in G" or "d is valid in G" are also correct, and will be needed later in the paper. In particular, we will consider algorithms making some large steps, where a

preflow increment f is computed and then applied to the initial network by assigning $G := G_f$. After that, the algorithm continues with G and resets f.

To ensure that residual paths do not go through the source and for reasons of efficiency, we make all edges from the source saturated during the following Init procedure (common to all algorithms in this paper).

Procedure Init

```
   /* saturate source edges                          */
1  f(s,v) := c(s,v), ∀(s,v) ∈ E;
2  G := G_f; f := 0;                    /* apply preflow */
3  d := 0, d(s) := d^∞;               /* initialize labels */
```

The generic push-relabel algorithm by Goldberg and Tarjan (1988) maintains a preflow f and a valid labeling d. It starts with `Init` and applies the following `Push` and `Relabel` operations while possible:

- `Push`(u, v), which is applicable if u is active and $c_f(u, v) > 0$ and $d(u) = d(v) + 1$. The operation increases $f(u, v)$ by Δ and decreases $f(v, u)$ by Δ, where $\Delta = \min(e_f(u), c_f(u, v))$.
- `Relabel`(u), which is applicable if u is active and $\forall v \mid (u, v) \in E,\ c_f(u, v) > 0$ it is $d(u) \leq d(v)$. It sets $d(u) := \min\big(d^\infty, \min\{d(v) + 1 \mid (u, v) \in E,\ c_f(u, v) > 0\}\big)$.

If u is active then either `Push` or `Relabel` operation is applicable to u. The algorithm preserves validity of the labeling and stops when there are no active vertices. Then for any u such that $e_f(u) > 0$, we have $d(u) = d^\infty$ and therefore $d^*(u) = d^\infty$ and $u \nrightarrow t$ in G_f, so f is a maximum preflow.

3 Region Discharge Revisited

We now review the approach of Delong and Boykov (2008) and reformulate it for the case of a fixed graph partition. We then introduce generic sequential and parallel algorithms which will be applied with both push-relabel and augmenting path approaches.

Delong and Boykov (2008) introduce the following operation. The *discharge* of a *region* $R \subset V \backslash \{s, t\}$ applies `Push` and `Relabel` to $v \in R$ until there are no active vertices left in R. This localizes computations to R and its *boundary*, defined as

$$B^R = \{w \mid \exists u \in R\ (u, w) \in E, w \notin R,\ w \neq s, t\}. \qquad (5)$$

When a `Push` is applied to an edge $(v, w) \in (R, B^R)$, the flow is sent out of the region. We say that two regions $R_1, R_2 \subset V \backslash \{s, t\}$ *interact* if $R_1 \cap R_2 \neq \emptyset$ or $R_1 \cap B^{R_2} \neq \emptyset$, that is they share vertices or they are connected by an edge. Because `Push` and `Relabel` operations work on the individual edges, discharges of non-interacting regions can be performed in parallel. The algorithm proposed by Delong and Boykov (2008) repeats the following steps until there are no active vertices in V:

1. Select several non-interacting regions, containing active vertices.
2. Discharge the selected regions in parallel, applying region-gap and region-relabel heuristics.
3. Apply the global gap heuristic.

All heuristics (global-gap, region-gap, region-relabel) serve to improve the distance estimate. They are very important in practice, but do not affect the theoretical properties and will be discussed in §5 devoted to the implementation.

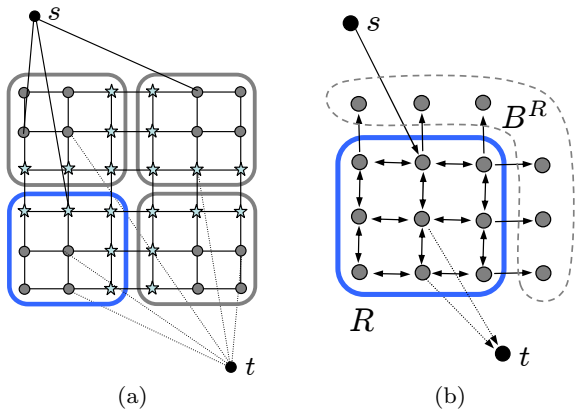


Fig. 3 (a) Partition of a network into 4 regions and the boundary set $\mathcal{B}$ depicted by stars. (b) The region network corresponding to the highlighted region in (a).

3.1 Region Network

We now take a different perspective on the algorithm. We consider each region discharge as a proper subproblem to be solved. Given the states of the boundary edges on the input (labels and excess), region discharge of region R returns a flow and a labeling. To define it formally, we first single out a subnetwork on which region discharge will work.

A *region network* $G^R = (V^R, E^R, s, t, c^R, e^R)$ has the set of vertices $V^R = R \cup B^R \cup \{s, t\}$; set of edges $E^R = (R \cup \{s, t\}, R \cup \{s, t\}) \cup (R, B^R) \cup (B^R, R^R)$; capacities $c^R(u, v) = c(u, v)$ if $(u, v) \in E^R \backslash (B^R, R)$ and 0 otherwise; and excess $e^R = e|_{R \cup \{s,t\}}$ (the restriction of function e to its subdomain $R \cup \{s, t\}$). This subnetwork is illustrated in Fig. 3(b). Note that the capacities of edges coming from the boundary, (B^R, R), are set to zero. Indeed, these edges belong to a neighboring region network. The region discharge operation of Delong and

Boykov (2008), which we refer to as Push-relabel Region Discharge (PRD), can now be defined as shown in Procedure PRD.

Procedure $(f, d) = \mathrm{PRD}(G^R, d)$

```
/* assume d: V^R → {0,...,d^∞} valid in G^R */
while ∃v ∈ R active do
    apply Push or Relabel to v; /* changes f and d */
    apply region gap heuristic (§5);  /* optional */
```

3.2 Generic Region Discharging Algorithms

While Delong and Boykov (2008) selected the regions dynamically, trying to divide the work evenly between CPUs in each iteration and cover the most of the active vertices, we restrict ourselves to a fixed collection of regions $(R^k)_{k=1}^K$ forming a partition (disjoint union) of $V \backslash \{s, t\}$. With respect to this partition we will use shortened notations B^k, G^k, V^k, E^k, c^k, e^k to denote the corresponding region boundary B^{R_k}, region network G^{R_k} and its respective components. We also define the *boundary* $\mathcal{B} = \bigcup_k B^k$, which is the set of all vertices incident to inter-region edges (Fig. 3(a)).

We now define generic sequential and parallel algorithms which use a black-box `Discharge` function as a subroutine. The sequential algorithm (Alg. 1) takes regions one-by-one from the partition and applies the `Discharge` operation to them until there are no active vertices in either region. The parallel algorithm (Alg. 2) calls `Discharge` for all regions concurrently and then resolves conflicts in the flow similarly to the asynchronous parallel push-relabel of Goldberg and Tarjan (1988). A conflict occurs if two interacting regions increase their labels on the vertices of a boundary edge (u, v) simultaneously and try pushing flow over it (in their respective region networks). In such a case, we accept the labels, but do not allow the flow to cross the boundary in one of the directions by the following construction. In step 5 of Alg. 2, boundary edges, where the validity condition is potentially violated, are assigned $\alpha = 0$ (here, $[\![\,]\!]$ is the Iverson bracket). The flow fusion in step 6 disables the flow on such edges (the flow going "upwards"). As will be proven later, this correction restores the validity. The actual implementation does not maintain the full network G, only the separate region networks. This is in contrast to Delong and Boykov (2008), who perform all operations in the global network G.

In the case when the abstract `Discharge` procedure is implemented by `PRD`, the sequential and parallel algorithms correspond to the push-relabel approach and will be referred to as S-PRD and P-PRD respectively.

Algorithm 1: Sequential Discharging

```
1 Init;
2 while there are active vertices do          /* a sweep */
3     for k = 1,...K do
4         Construct G^k from G;
5         (f', d') := Discharge(G^k, d|_{V^k});
6         G := G_{f'};                        /* apply f' to G */
7         d|_{R^k} := d'|_{R^k};              /* update labels */
8         apply global gap heuristic (§5); /* optional */
9 Compute the reachability v → t in G, ∀v (§5.2);
```

Algorithm 2: Parallel Discharging

```
1 Init;
2 while there are active vertices do          /* a sweep */
      /* discharge all regions in parallel                  */
3     (f'_k, d'_k) := Discharge(G^k, d|_{V^k}) ∀k;
4     d'|_{R^k} := d'_k|_{R^k} ∀k;                /* fuse labels */
      /* determine valid pairs                              */
5     α(u,v) := [[d'(u) ≤ d'(v) + 1]]  ∀(u,v) ∈ (B,B);
      /* fuse flows                                         */
6     f'(u,v) :=
      { α(v,u) f'_k(u,v) + α(u,v) f'_j(u,v)   if (u,v) ∈ (R^k, R^j)
      { f'_k(u,v)                             if (u,v) ∈ (R^k, R^k)
7     G := G_{f'};                            /* apply f' to G */
8     d := d';                                /* update labels */
9     global gap heuristic (§ 5);             /* optional */
10 Compute the reachability v → t in G, ∀v (§5.2);
```

S-PRD is a sequential variant of Delong and Boykov (2008) and P-PRD is a novel parallel variant. As was mentioned above, the original algorithm by Delong and Boykov (2008) allows to discharge only non-interacting regions in parallel (in this case there are no conflicts). To discharge all regions, this approach would require sequentially selecting subsets of non-interacting regions for processing.[2] Our parallel algorithm applies ideas of Goldberg and Tarjan (1988) to remove this limitation and process all regions in parallel.

We prove below that both S-PRD and P-PRD terminate with a valid labeling in at most $2n^2$ sweeps. Parallel variants of push-relabel (Goldberg 1987) have the same bound on the number of sweeps. However, they perform much simpler sweeps, processing every vertex only once, compared to S/P-PRD. A natural question is whether $O(n^2)$ bound cannot be tightened for S/P-PRD. In §9, we give an example of a graph, its partition into two regions and a valid sequence of `Push` and `Relabel` operations, implementing S/P-PRD which takes $\Omega(n^2)$ sweeps to terminate.[3] The number

[2] The number of sequential phases required in a general case is equal to the minimal coloring of the region interaction graph, *i.e.* 2 for bipartite graph and so on.

[3] An algorithm is said to be $\Omega(f(n))$ if for some numbers c' and n_0 and all $n >= n_0$, the algorithm takes at least $c' f(n)$

of inter-region edges in this example is constant, which shows that a better bound in terms of this characteristic is not possible.

3.3 Complexity of Sequential Push-relabel Region Discharging

Our proof follows the main idea of the similar result for parallel push-relabel by Goldberg (1987). The main difference is that we try to keep `Discharge` operation as abstract as possible. Indeed, it will be seen that proofs of termination of other variants follow the same pattern, using several important properties of the `Discharge` operation, abstracted from the respective algorithm. Unfortunately, to this end we do not have a unified proof, so we will analyze all cases separately.

Statement 1 (Properties of PRD)
Let $(f', d') = \texttt{PRD}(G^R, d)$, then
1. there are no active vertices in R w.r.t. (f', d') (*optimality*)
2. $d' \geq d$; $d'|_{B^R} = d|_{B^R}$ (*labeling monotony*)
3. d' is valid in $G^R_{f'}$ (*labeling validity*)
4. $f'(u,v) > 0 \Rightarrow d'(u) > d(v)$, $\forall (u,v) \in E^R$ (*flow direction*).

Proof 1. Optimality. This is the stopping condition of `PRD`.

2,3. Labeling validity and monotony: labels are never decreased and the `Push` operation preserves labeling validity (Goldberg and Tarjan 1988). Labels not in R^k are not modified.

4. Flow direction: let $f'(u,v) > 0$, then there was a push operation from u to v in some step. Let $\tilde{d}$ be the labeling in this step. We have $\tilde{d}(u) = \tilde{d}(v) + 1$. Because labels never decrease, $d'(u) \geq \tilde{d}(u) > \tilde{d}(v) \geq d(v)$. □

These properties are sufficient to prove correctness and the complexity bound of S-PRD. They are abstract from the actual sequence of `Push` and `Relabel` operation performed by `PRD` and for a given pair (f', d') they are easy to verify. For correctness of S-PRD we need to verify that it maintains a labeling, which is globally valid.

Statement 2 Let d be a valid labeling in G. Let f' be a preflow in G^R and d' be a labeling satisfying properties 2 and 3 of Statement 1. Extend f' to E by letting $f'|_{E \setminus E^R} = 0$ and extend d' to V by letting $d'|_{V \setminus V^R} = d|_{V \setminus V^R}$. Then d' is valid in $G_{f'}$.

time on *some* problem instance. Here we measure complexity in sweeps.

Proof We have that d' is valid in $G^R_{f'}$. For edges outside the region network, $(u,v) \in (V \setminus R, V \setminus R)$, it is $f'(u,v) = 0$ and d' coincides with d on $V \setminus R$. It remains to verify validity on the boundary edges $(v,u) \in (B^R, R)$ in the case $c^R_f(v,u) = 0$ and $c_f(v,u) > 0$. These are the incoming boundary edges which are zeroed in the network G^R. Because $0 = c^R_f(v,u) = c^R(v,u) - f(v,u) = -f(v,u)$, we have $c_f(v,u) = c(v,u)$. Since d was valid in G, $d(v) \leq d(u) + 1$. The new labeling d' satisfies $d'(u) \geq d(u)$ and $d'(v) = d(v)$. It follows that $d'(v) = d(v) \leq d(u) + 1 \leq d'(u) + 1$. Hence d' is valid in $G_{f'}$. □

Similar to Goldberg (1987), we introduce the *potential function*

$$\Phi = \max\{d(v) \mid v \in V,\ v \text{ is active in } G\}. \tag{6}$$

This value may increase and decrease during the algorithm run, but the total number of times it can change is bounded. We first show that for a region discharge on R its increase is bounded by the total increase of the labeling.

Statement 3 Let (f', d') satisfy properties 2-4 of Statement 1. Let f' be extended to E by setting $f'|_{E \setminus E^R} = 0$ and d' be extended to V by setting $d'|_{V \setminus V^R} = d|_{V \setminus V^R}$. Let $G' = G_{f'}$ and Φ' be the new potential computed for the network G' and labeling d'. Then

$$\Phi' - \Phi \leq \sum_{v \in R} [d'(v) - d(v)]. \tag{7}$$

Proof Let the maximum in the definition of Φ' be attained at a vertex v, so $\Phi' = d'(v)$. Then either $v \notin V^R$, in which case $\Phi' \leq \Phi$ (because the label and the excess of v in G and G' are the same), or $v \in V^R$ and there exists a path $(v_0, v_1, \ldots v_l)$, $v_l = v$, $v_0, \ldots v_{l-1} \in R$, such that $f'(v_{i-1}, v_i) > 0$, $i = 1 \ldots l$ and v_0 is active in G. We have $\Phi \geq d(v_0)$, therefore

$$\begin{aligned} \Phi' - \Phi &\leq d'(v_l) - d(v_0) = \\ &\sum_{i=1}^{l} [d'(v_i) - d'(v_{i-1})] + [d'(v_0) - d(v_0)] \\ &\overset{(a)}{\leq} \sum_{i=0}^{l} [d'(v_i) - d(v_i)] \\ &\overset{(b)}{\leq} \sum_{v \in R \cup B^R} [d'(v) - d(v)] \overset{(c)}{=} \sum_{v \in R} [d'(v) - d(v)], \end{aligned} \tag{8}$$

where inequality (a) is due to the flow direction property (statement 1.4) which implies $d'(v_{i-1}) > d(v_i)$. The inequality (b) is due to monotony property (statement 1.2) and due to $v_i \subset R \cup B^R$. The equality (c) is due to $d'|_{B^R} = d|_{B^R}$. □

We can now state the termination.

Theorem 1 *S-PRD terminates in at most $2n^2$ sweeps.*

Proof Labeling d does not exceed n for every vertex. Because there are n vertices, d can be increased n^2 times at most.

From Statement 3 it follows that the increase of Φ after the discharge of region R^k is bounded by the total increase of $d|_{R^k}$. Since regions are disjoint, the total increase of Φ after a sweep of S-PRD is bounded by the total increase of d.

If d has not increased during a sweep ($d' = d$) then Φ decreases at least by 1. Indeed, let us consider the set of vertices having the label greater or equal to the label of the highest active vertex, $H = \{v \mid d(v) \geq \Phi\}$. These vertices do not receive flow during all discharge operations due to the flow direction property. After discharging R^k, there are no active vertices in $R^k \cap H$ (statement 1.1). Therefore, there are no active vertices in H after the full sweep.

In the worst case, Φ can increase by one n^2 times and decrease by one n^2 times. Therefore, there are no active vertices in G in at most $2n^2$ sweeps and the algorithm terminates. □

When the algorithm terminates, it outputs a network G, equivalent to the initial one, a labeling d valid in G and guarantees that there are no active vertices w.r.t. d. This implies that in the current network G there are no paths form the vertices with excess to the sink and the cut $(\bar{T}, T)$, with $T = \{v \mid v \to t \text{ in } G\}$ is one of the minimum cuts. The issue how to compute the reachability $v \to t$ in G in a distributed fashion, utilizing d, rather than by breadth-first search in G is discussed in §5.2. This is the purpose of the last step in both of the algorithms.

3.4 Complexity of Parallel Push-relabel Region Discharging

We will now prove the validity and termination of the parallel algorithm using the results of previous section. Properties similar to Statement 1 will be proven for the fused flow and labeling (constructed at step 6 of Alg. 2) and the bound on the increase of the potential will follow for the whole network as if it was a single region.

Statement 4 Let d be a valid labeling in the beginning of a sweep of P-PRD. Then the pair of fused flow and labeling (f', d') satisfies:

1. $d' \geq d$; *(labeling monotony)*
2. d' is valid in $G_{f'}$; *(labeling validity)*
3. $f'(u,v) > 0 \Rightarrow d'(u) > d(v)$, $\forall (u,v) \in E$; *(flow direction)*

Proof
1. We have $d'|_{R^k} \geq d|_{R^k}$ for all k.
2. We have to prove validity for the boundary edges, where the flow and the labeling are fused from different regions. It is sufficient to study the two regions case. Denote the regions R^1 and R^2. The situation is completely symmetric w.r.t. orientation of a boundary edge (u,v). Let $u \in R^1$ and $v \in R^2$. Let only $d'(v) \leq d'(u) + 1$ be satisfied and not $d'(u) \leq d'(v) + 1$. By the construction in step 6 of Alg. 2, flow f_2 is canceled and $f'(u,v) = f'_1(u,v) \geq 0$. Suppose $c_{f'_1}(u,v) > 0$, then we have that $d'_1(u) \leq d'_1(v) + 1$, because d'_1 is valid in $G^1_{f'_1}$. It follows that $d'(u) = d'_1(u) \leq d'_1(v) + 1 = d(v) + 1 \leq d'_2(v) + 1 = d'(v) + 1$, where we also used labeling monotonicity property. The inequality $d'(u) \leq d'(v) + 1$ is a contradiction, therefore it must be that $c_{f'}(u,v) = 0$. The labeling d' is valid on (u,v) in this case. Note that inequalities $d'(v) \leq d'(u) + 1$ and $d'(u) \leq d'(v) + 1$ cannot be violated simultaneously. In the remaining case, when both inequalities are satisfied, the labeling is valid for arbitrary flow on (u,v), so no flow is canceled in the flow fusion step.
3. If $f'(u,v) > 0$ then $f'_k(u,v) > 0$ and there was a push operation from u to v in the discharge of region $R^k \ni u$. Let $\tilde{d}_k$ be the labeling in G^k on this step. We have $d'(u) \geq \tilde{d}_k(u) = \tilde{d}_k(v) + 1 \geq d(v) + 1 > d(v)$. □

Theorem 2 *P-PRD terminates in at most $2n^2$ sweeps.*

Proof As before, the total increase of d is at most n^2. As shown above, the labeling monotony, labeling validity and flow direction are satisfied for the fused flow and labeling (f', d') on the region $R = V \backslash \{s, t\}$. Applying Statement 3, we get that the total increase of potential is bounded above by the total increase of d during a sweep.

If d has not increased during a sweep ($d' = d$) then $\alpha(u,v) = 1$ for all $(u,v) \in (\mathcal{B}, \mathcal{B})$ (all boundary pairs are valid). Flow direction property implies that the flow goes only "downwards" the labeling. So no flow is canceled on the fusion step. Let $H = \{v \mid d(v) \geq \Phi\}$. These vertices are above any active vertices, so they cannot receive flow. After the sweep, all active vertices which were in H are discharged and must become inactive. Because there is no active vertices with label Φ or above left, it is $\Phi' < \Phi$. It follows that the algorithm will terminate in at most $2n^2$ sweeps. □

4 Augmented path Region Discharge

In this section, we introduce the core of our new algorithm, which combines path augmentation and push-

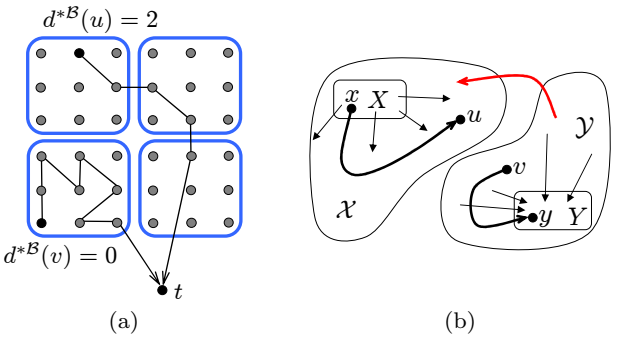


Fig. 4 (a) Illustration of region distance. (b) Illustration of Lemma 1: augmentation on paths from x to u or from v to y preserves $X \nrightarrow Y$, but not the augmentation on the red path.

relabel approaches. We will give a new definition to the distance function and validity of a labeling and introduce the new `Discharge` operation to be used within the generic sequential and parallel algorithms (Alg. 1 and 2). With these modifications the algorithms will be proven correct and posses a tighter bound on the number of sweeps.

4.1 A New Distance Function

Consider the fixed partition $(R^k)_{k=1}^K$. Let us introduce a distance function, which counts only inter-region edges and not inner edges. The *region distance* $d^{*\mathcal{B}}(u)$ in G is the minimal number of inter-region edges contained in a path from u to t, or $|\mathcal{B}|$ if no such path exists:

$$d^{*\mathcal{B}}(u) = \begin{cases} \min\limits_{P=((u,u_1),\ldots,(u_r,t))} |P \cap (\mathcal{B},\mathcal{B})| & \text{if } u \to t, \\ |\mathcal{B}| & \text{if } u \nrightarrow t. \end{cases} \tag{9}$$

This distance corresponds well to the number of region discharge operations required to transfer the excess to the sink (see Fig. 4(a)).

Statement 5 If $u \to t$ then $d^{*\mathcal{B}}(u) < |\mathcal{B}|$.

Proof Let P be a path from u to t given as a sequence of edges. If P contains a loop then it can be removed from P and $|P \cap (\mathcal{B},\mathcal{B})|$ will not increase. A path without loops goes through each vertex at most once. For $\mathcal{B} \subset V$ there is at most $|\mathcal{B}| - 1$ edges in the path which have both endpoints in $\mathcal{B}$. □

We now let $d^\infty = |\mathcal{B}|$ and redefine a valid labeling w.r.t. the new distance. A labeling $d\colon V \to \{0,\ldots,d^\infty\}$ is *valid* in G if $d(t) = 0$ and for all $(u,v) \in E$ such that $c_f(u,v) > 0$:

$$d(u) \le d(v) + 1 \qquad \text{if } (u,v) \in (\mathcal{B},\mathcal{B}), \tag{10}$$

$$d(u) \le d(v) \qquad \text{if } (u,v) \notin (\mathcal{B},\mathcal{B}). \tag{11}$$

Procedure ARD(G^R,d)

```
/* assume d: V^R → {0,...,d^∞} valid in G^R */
for i = 0,1,...,d^∞ do                        /* stage i */
    T_i = {t} ∪ {v ∈ B^R | d(v) < i};
    Augment(R, T_i);
/* Region-relabel */
```

$$d(u) := \begin{cases} \min\{i \mid u \to T_i\} & u \in R, u \to T_{d^\infty}, \\ d^\infty & u \in R, u \nrightarrow T_{d^\infty}, \\ d(u) & u \in B^R. \end{cases}$$

Procedure Augment(X,Y)

while *there exist a path* $(v_0, v_1, \ldots, v_l)$, $c_f(v_{i-1}, v_i) > 0$, $e_f(v_0) > 0$, $v_0 \in X$, $v_l \in Y$ **do**
  augment $\Delta = \min(e_f(v_0), \min\limits_i c_f(v_{i-1}, v_i))$ units along the path.

Statement 6 A valid labeling d is a lower bound on the region distance $d^{*\mathcal{B}}$.

Proof If $u \nrightarrow t$ then $d(u) \le d^{*\mathcal{B}}$. Otherwise, let $P = ((u,v_1),\ldots,(v_l,t))$ be one of the shortest paths w.r.t. $d^{*\mathcal{B}}$, *i.e.* $d^{*\mathcal{B}}(u) = |P \cap (\mathcal{B},\mathcal{B})|$. Applying the validity property to each edge in this path, we have $d(u) \le d(t) + |P \cap (\mathcal{B},\mathcal{B})| = d^{*\mathcal{B}}(u)$. □

4.2 New Region Discharge

In this subsection, reachability relations "$\to$", "$\nrightarrow$", residual paths, and labeling validity will be understood in the region network G^R or its residual G_f^R.

The new `Discharge` operation, called Augmented path Region Discharge (ARD), works as follows. It first pushes excess to the sink along augmenting paths inside the network G^R. When it is no longer possible, it continues to augment paths to vertices in the region boundary B^R in the order of their increasing labels. This is represented by the sequence of nested sets $T_0 = \{t\}$, $T_1 = \{t\} \cup \{v \in B^R \mid d(v) < 1\}$, $\ldots$, $T_{d^\infty} = \{t\} \cup \{v \in B^R \mid d(v) < d^\infty\}$. Set T_i is the destination of augmentations in stage i. As we prove below, in stage $i > 0$ residual paths may exist only to the set $T_i \backslash T_{i-1} = \{v \mid d(v) = i - 1\}$.

The labels on the boundary $d|_{B^R}$ remain fixed during ARD and the labels $d|_R$ inside the region do not participate in augmentations and therefore are updated only in the end.

We claim that ARD terminates with no active vertices inside the region, preserves validity and monotonicity of the labeling, and pushes flow from higher la-

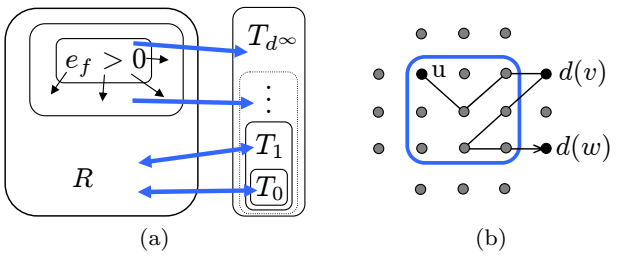


Fig. 5 (a) Reachability relations in the network G_f^R at the end of stage 1 of ARD: $\{v \,|\, e_f(v) > 0\} \nrightarrow T_1$; $T_{d^\infty} \setminus T_1 \nrightarrow R$. (b) Example of a path in the network G_f^R for which by Corollary 2 it must be $d(v) \le d(w)$. Note, such a path is not possible at the beginning of ARD, but in the middle it may exist since residual capacities of edges (B^R, R) may become non-zero.

bels to lower labels w.r.t. the new labeling. These properties will be required to prove finite termination and correctness of S-ARD. Before we prove them (Statement 10) we need the following intermediate results:

- Properties of the network G_f^R maintained by ARD (Statement 7, Corollaries 1 and 2).
- Properties of valid labellings in the network G_f^R (Statement 8).
- Properties of the labeling constructed by region-relabel (line 4 of `ARD`) in the network G_f^R (Statement 9).

Lemma 1 *Let $X, Y \subset V^R$, $X \cap Y = \emptyset$, $X \nrightarrow Y$. Then $X \nrightarrow Y$ is preserved after i) augmenting a path $(x, \dots, v)$ with $x \in X$ and $v \in V^R$; ii) augmenting a path $(v, \dots, y)$ with $y \in Y$ and $v \in V^R$.*

Proof (See Fig. 4(b)). Let $\mathcal{X}$ be the set of vertices reachable from X. Let $\mathcal{Y}$ be the set of vertices from which Y is reachable. Clearly $\mathcal{X} \cap \mathcal{Y} = \emptyset$, otherwise $X \to Y$. Therefore, the cut $(\mathcal{X}, \bar{\mathcal{X}})$ separates X and Y and has all edge capacities equal zero. Any residual path starting in X or ending in Y cannot cross the cut and its augmentation cannot change the edges of the cut. Hence, X and Y will stay separated. □

Statement 7 Let $v \in V^R$ and $v \nrightarrow T_a$ in G_f in the beginning of stage i_0 of ARD, where $a, i_0 \in \{0, 1, \dots d^\infty\}$. Then $v \nrightarrow T_a$ holds until the end of ARD, that is during all stages $i \ge i_0$.

Proof We need to show that $v \nrightarrow T_a$ is not affected by augmentations performed by ARD. If $i_0 \le a$, we first prove $v \nrightarrow T_a$ holds during stages $i_0 \le i \le a$. Consider augmentation of a path $(u_0, u_1, \dots, u_l)$, $u_0 \in R$, $u_l \in T_i \subset T_a$, $e_f(u_0) > 0$. Assume $v \nrightarrow T_a$ before augmentation. By Lemma 1 with $X = \{v\}$, $Y = T_a$ (noting that $X \nrightarrow Y$ and the augmenting path ends in Y), after the augmentation $v \nrightarrow T_a$. By induction, it holds till the end of stage a and hence in the beginning of stage $a + 1$.

We can assume now that $i_0 > a$. Let $A = \{u \in R \,|\, e_f(u) > 0\}$. At the end of stage $i_0 - 1$, we have $A \nrightarrow T_{i_0 - 1} \supset T_a$ by construction. Consider augmentation in stage i_0 on a path $(u_0, u_1 \dots, u_l)$, $u_0 \in R$, $u_l \in T_{i_0}$, $e_f(u_0) > 0$. By construction, $u_0 \in A$. Assume $\{v\} \cup A \nrightarrow T_a$ before augmentation. Apply Lemma 1 with $X = \{v\} \cup A$, $Y = T_a$ (we have $X \nrightarrow Y$ and $u_0 \in A \subset X$). After augmentation it is $X \nrightarrow T_a$. By induction, $X \nrightarrow T_a$ till the end of stage i_0. By induction on stages, $v \nrightarrow T_a$ until the end of the ARD procedure. □

Corollary 1 *If $w \in B^R$ then $w \nrightarrow T_{d(w)}$ throughout the ARD procedure.*

Proof At initialization, it is fulfilled by construction of G^R due to $c^R(B^R, R) = 0$. It holds then during ARD by Statement 7. □

In particular, we have $B^R \nrightarrow t$ during ARD.

Corollary 2 *Let $(u, v_1 \dots v_l, w)$ be a residual path in G_f^R from $u \in R$ to $w \in B^R$ and let $v_r \in B^R$ for some r. Then $d(v_r) \le d(w)$.*

Proof We have $v_r \nrightarrow T_{d(v_r)}$. Suppose $d(w) < d(v_r)$, then $w \in T_{d(v_r)}$ and because $v_r \to w$ it is $v_r \to T_{d(v_r)}$ which is a contradiction. □

The properties of the network G_f^R established by Statement 7 and Corollary 2 are illustrated in Fig. 5.

Statement 8 Let d be a valid labeling, $d(u) \ge 1$, $u \in R$. Then $u \nrightarrow T_{d(u)-1}$.

Proof Suppose $u \to T_0$. Then there exist a residual path $(u, v_1 \dots v_l, t)$, $v_i \in R$ (by Corollary 1 it cannot happen that $v_i \in B^R$). By validity of d we have $d(u) \le d(v_1) \le \dots \le d(v_l) \le d(t) = 0$, which is a contradiction.

Suppose $d(u) > 1$ and $u \to T_{d(u)-1}$. Because $u \nrightarrow T_0$, it must be that $u \to w$, $w \in B^R$ and $d(w) < d(u) - 1$. Let $(v_0 \dots v_l)$ be a residual path with $v_0 = u$ and $v_l = w$. Let r be the minimal number such that $v_r \in B^R$. By validity of d we have $d(u) \le d(v_1) \le \dots \le d(v_{r-1}) \le d(v_r) + 1$. By corollary 2 we have $d(v_r) \le d(w)$, hence $d(u) \le d(w) + 1$ which is a contradiction. □

Statement 9 For d computed on line 4 of Procedure ARD and any $u \in R$ it holds:

1. d is valid;
2. $u \nrightarrow T_a \Leftrightarrow d(u) \ge a + 1$.

Proof
1. Let $(u, v) \in E^R$ and $c(u, v) > 0$. Clearly $u \to v$. Consider four cases:

- case $u \in R$, $v \in B^R$: Then $u \to T_{d(v)+1}$, hence $d(u) \le d(v) + 1$.
- case $u \in R$, $v \in R$: If $v \nrightarrow T_{d^\infty}$ then $d(v) = d^\infty$ and $d(u) \le d(v)$. If $v \to T_{d^\infty}$, then $d(v) = \min\{i \mid v \to T_i\}$. Let $i = d(v)$, then $v \to T_i$ and $u \to T_i$, therefore $d(u) \le i = d(v)$.
- case $u \in B^R$, $v \in R$: By Corollary 1, $u \nrightarrow T_{d(u)}$. Because $u \to v$, it is $v \nrightarrow T_{d(u)}$, therefore $d(v) \ge d(u) + 1$ and $d(u) \le d(v) - 1 \le d(v) + 1$.
- case when $u = t$ or $v = t$ is trivial.

2. The "$\Leftarrow$" direction follows by Statement 8 applied to d, which is a valid labeling. The "$\Rightarrow$" direction: we have $u \nrightarrow T_a$ and $d(u) \ge \min\{i \mid u \to T_i\} = \min\{i > a \mid u \to T_i\} \ge a + 1$. □

Statement 10 (Properties of ARD) Let d be a valid labeling in G^R. The output (f', d') of ARD satisfies:

1. There are no active vertices in R w.r.t. (f', d');(*optimality*)
2. $d' \ge d$, $d'|_{B^R} = d|_{B^R}$; (*labeling monotonicity*)
3. d' is valid in $G^R_{f'}$; (*labeling validity*)
4. f' is a sum of path flows, where each path is from a vertex $u \in R$ to a vertex $v \in \{t\} \cup B^R$ and it is $d'(u) > d(v)$ if $v \in B^R$. (*flow direction*)

Proof 1. In the last stage, the ARD procedure augments all paths to T_{d^∞}. After this augmentation a vertex $u \in R$ either has excess 0 or there is no residual path to T_{d^∞} and hence $d'(u) = d^\infty$ by construction.

2. For $d(u) = 0$, we trivially have $d'(u) \ge d(u)$. Let $d(u) = a + 1 > 0$. By Statement 8, $u \nrightarrow T_a$ in G^R and it holds also in $G^R_{f'}$ by Statement 7. From Statement 9.2, we conclude that $d'(u) \ge a + 1 = d(u)$. The equality $d'|_{B^R} = d|_{B^R}$ is by construction.
3. Proven by Statement 9.1.
4. Consider a path from u to $v \in B^R$, augmented in stage $i > 0$. It follows that $i = d(v)+1$. At the beginning of stage i, it is $u \nrightarrow T_{i-1}$. By Statement 7, this is preserved till the end of ARD. By Statement 9.2, $d'(u) \ge i = d(v) + 1 > d(v)$. □

Algorithms 1 and 2 for `Discharge` being ARD will be referred to as S-ARD and P-ARD, respectively.

4.3 Complexity of Sequential Augmented path Region Discharging

Statement 2 holds for S-ARD as well, so S-ARD maintains a valid labeling.

Theorem 3 *S-ARD terminates in at most* $2|\mathcal{B}|^2 + 1$ *sweeps.*

Proof The value of $d(v)$ does not exceed $|\mathcal{B}|$ and d is non-decreasing. The total increase of $d|_{\mathcal{B}}$ during the algorithm is at most $|\mathcal{B}|^2$.

After the first sweep, active vertices are only in $\mathcal{B}$. Indeed, discharging region R^k makes all vertices $v \in R^k$ inactive and only vertices in $\mathcal{B}$ may become active. So by the end of the sweep, all vertices $V \backslash \mathcal{B}$ are inactive.

Therefore, after the first sweep, the potential can be equivalently written as

$$\Phi = \max\{d(v) \mid v \in \mathcal{B},\ v \text{ is active in } G\}. \tag{12}$$

We will prove the following two cases for each sweep but the first one:

1. If $d|_{\mathcal{B}}$ is increased then the increase in Φ is no more than total increase in $d|_{\mathcal{B}}$. Consider discharge of R^k. Let Φ be the value before ARD on R^k and Φ' the value after. Let $\Phi' = d'(v)$. It must be that v is active in G'. If $v \notin V^k$, then $d(v) = d'(v)$ and $e(v) = e_{f'}(v)$ so $\Phi \ge d(v) = \Phi'$.
   Let $v \in V^k$. After the discharge, vertices in R^k are inactive, so $v \in B^k$ and it is $d'(v) = d(v)$. If v was active in G then $\Phi \ge d(v)$ and we have $\Phi' - \Phi \le d'(v) - d(v) = 0$. If v was not active in G, there must exist an augmenting path from a vertex v_0 to v such that $v_0 \in R^k \cap \mathcal{B}$ was active in G. For this path, the flow direction property implies $d'(v_0) \ge d(v)$. We now have $\Phi' - \Phi \le d'(v) - d(v_0) = d(v) - d(v_0) \le d'(v_0) - d(v_0) \le \sum_{v \in R^k \cap \mathcal{B}} [d'(v) - d(v)]$. Summing over all regions, we get the result.
2. If $d|_{\mathcal{B}}$ is not increased then Φ is decreased at least by 1. We have $d' = d$. Let us consider the set of vertices having the highest active label or above, $H = \{v \mid d(v) \ge \Phi\}$. These vertices do not receive flow during all discharge operations due to the flow direction property. After the discharge of R^k there are no active vertices left in $R^k \cap H$ (Statement 10.1). After the full sweep, there are no active vertices in H.

In the worst case, starting from sweep 2, Φ can increase by one $|\mathcal{B}|^2$ times and decrease by one $|\mathcal{B}^2|$ times. There are no active vertices left in at most $2|\mathcal{B}|^2 + 1$ sweeps. □

On termination, we have that the labeling is valid and there are no active vertices in G. Therefore, the accumulated preflow is maximal and a minimum cut can be found by analyzing the reachability in G (see discussion for S-PRD §3.3 and implementation details §5.2).

4.4 Complexity of Parallel Augmented Path Region Discharging

Statement 11 (Properties of Parallel ARD) Let d be a valid labeling at the beginning of a sweep of P-ARD. Then the pair of fused flow and labeling (f', d') satisfies:

1. Vertices in $V \backslash \mathcal{B}$ are not active in $G_{f'}$. (*optimality*)
2. $d' \geq d$; (*labeling monotony*)
3. d' is valid; (*labeling validity*)
4. f' is the sum of path flows, where each path is from a vertex $u \in V$ to a vertex $v \in \mathcal{B}$, satisfying $d'(u) \geq d(v)$. (*weak flow direction*)

Proof 1. For each k there are no active vertices in R^k w.r.t. (f'_k, d'_k). The fused flow f' may differ from f'_k only on the boundary edges $(u, v) \in (\mathcal{B}, \mathcal{B})$. So there are no active vertices in $V \backslash \mathcal{B}$ w.r.t. (f', d').
2. By construction.
3. Same as in P-PRD.
4. Consider the augmentation of a path from $u \in R^k$ to $v \in B^k$ during ARD on G^k and canceling of the flow on the last edge of the path during the flow fusion step. Let the last edge of the path be (w, v). We need to prove that $d'(u) \geq d(w)$. Let $\tilde{d}$ be the labeling in G^k right before augmentation, as if it was computed by region-relabel. Because $\tilde{d}$ is valid it must be that $\tilde{d}(w) \leq \tilde{d}(v) + 1$. We have $d'_k(u) > d(v) = \tilde{d}(v) \geq \tilde{d}(w) - 1 \geq d(w) - 1$. So $d'(u) \geq d(w)$. □

Theorem 4 *P-ARD terminates in* $2|\mathcal{B}|^2 + 1$ *sweeps.*

Proof As before, total increase of $d|_{\mathcal{B}}$ is at most $|\mathcal{B}|^2$. After the first sweep, active vertices are only in $\mathcal{B}$ by Statement 11.1.
For each sweep after the first one:

- If $d|_{\mathcal{B}}$ is increased then increase in Φ is no more than the total increase of $d|_{\mathcal{B}}$. Let Φ' be the value of the potential in the network $G' = G_{f'}$. Let $\Phi' = d'(v)$. It must be that v is active in G' and $v \in \mathcal{B}$.
  If v was active in G then $\Phi \geq d(v)$ and we have $\Phi' - \Phi \leq d'(v) - d(v)$.
  If v was not active in G then there must exist a path flow in f' from a vertex v_0 to v such that $v_0 \in \mathcal{B}$ was active in G. For this path, the weak flow direction property implies $d'(v_0) \geq d(v)$. We have $\Phi' - \Phi \leq d'(v) - d(v_0) = d'(v) - d(v) + d(v) - d(v_0) \leq d'(v) - d(v) + d'(v_0) - d(v_0) \leq \sum_{v \in \mathcal{B}} [d'(v) - d(v)]$.
- If $d|_{\mathcal{B}}$ is not increased then Φ is decreased at least by 1. In this case, f' satisfies the strong flow direction property and the proof of Theorem 3 applies.

After total of $2|\mathcal{B}|^2 + 1$ sweeps, there are no active vertices left. □

5 Implementation

In this section, we first discuss heuristics for improving the distance labeling (making it closer to the true distance at a cheap cost) commonly used in the push-relabel framework. They are essential for the practical performance of the algorithms. We then describe our base implementations of S-ARD/S-PRD and the solvers they rely on. In the next section, we describe an efficient implementation of ARD, which is more sophisticated but has a much better practical performance. All of the labeling heuristics can only increase the labels and preserve validity of the labeling. Therefore, they do not break theoretical properties of the respective algorithms.

5.1 Heuristics

Region-relabel heuristic computes labels $d|_R$ of the region vertices, given the distance estimate on the boundary, $d|_{B^R}$. There is a slight difference between PRD and ARD variants (using distance d^* and $d^{*\mathcal{B}}$, resp.), displayed by the corresponding "if" conditions.

Procedure $d|_R$ = Region-relabel(G^R,$d|_{B^R}$)

```
   /* init                                                    */
1  d(t) := 0; O := {t}; d|_R := d^∞; d^c := 0;
2  if ARD then d|_{B^R} := d|_{B^R} + 1;          /* (for ARD) */
   /* O is a list of open vertices, having the
      current label d^c                                       */
3  d^max := max{d(w) | w ∈ B^R, d(w) < d^∞};
4  while O ≠ ∅ or d^c < d^max do
       /* if O is empty raise d^c to the next seed            */
5      if O = ∅ then
       d^c := min{d(w) | w ∈ B^R, d(w) > d^c, d(w) < d^∞};
       /* add seeds to the open set                           */
6      O := O ∪ {w ∈ B^R | d(w) = d^c};
       /* find all unlabeled vertices from which O
          can be reached                                      */
7      O := {u ∈ R | (u,v) ∈ E^R, v ∈ O, c(u,v) > 0, d(u) =
       d^∞};
8      if PRD then d^c ← d^c + 1;                  /* (for PRD) */
9      d|_O := d^c;          /* label the new open vertices */
10 if ARD then d|_{B^R} := d|_{B^R} − 1;           /* (for ARD) */
```

In the implementation, the set of boundary vertices is sorted in advance, so that Region-relabel runs in $O(|E^R| + |V^R| + |B^R| \log |B^R|)$ time and uses $O(|V^R|)$ space. The resulting labeling d' is valid and satisfies $d' \geq d$ for arbitrary valid d.

Global gap heuristic. Let us briefly explain the global gap heuristic (Cherkassky and Goldberg 1994). It is a sufficient condition to identify that the sink is unreachable from a set of vertices. Let there be no vertices with label $g > 0$: $\forall v \in V \; d(v) \neq g$, and let $d(u) > g$. For a valid labeling d, it follows that there is no vertex

v for which $c(u,v) > 0$ and $d(v) < g$. Assuming there is, we will have $d(u) \leq d(v) + 1 \leq g$, which is a contradiction. Therefore the sink is unreachable from all vertices $\{u \mid d(u) > g\}$ and their labels may be set to d^∞.

Region gap heuristic of Delong and Boykov (2008) detects if there are no vertices inside region R having label $g > 0$. Such vertices can be connected to the sink in the whole network only through one of the boundary vertices, so they may be relabeled up to the closest boundary label. Here is the algorithm[4]:

Procedure $d|_R = \text{Region-gap}(G^R, d|_{R \cup B^R}, g)$

```
   /* Input: region network G^R, labeling d,        */
   /* gap g: ∀v ∈ R d(v) ≠ g                        */
1  d^next := min{d(w) | w ∈ B^R, d(w) > g.};
2  for v ∈ R such that g < d(v) < d^next do
3      d(v) := d^next+1;
```

If no boundary vertex is above the gap, then $d^{\text{next}} = d^\infty$ in step 1 and all vertices above the gap are disconnected from the sink in the network G. Interestingly, this sufficient condition does not imply a global gap. In our implementation of PRD, we detect the region-gap efficiently after every vertex relabel operation by discovering an empty bucket (see the implementation of S/P-PRD in §5.5).

5.2 Reachability/Exact distance Computation

At the termination of our algorithms (S/P-PRD, S/P-ARD), we have found a maximum preflow and we know that $(\bar{T}, T)$, defined by $T = \{v \mid v \to t \text{ in } G\}$, is a minimum cut. However, we only know the lower bound d on the true distance d^* (resp. $d^{*\mathcal{B}}$) and therefore the reachability relation $v \to t$ is not fully known at this point. When $d(v) = d^\infty$ we are sure that $v \nrightarrow t$ in G and hence v must be in the source set of a minimum cut, but if $d(v) < d^\infty$ it is still possible that $v \nrightarrow t$ in G. Therefore, we need to do some extra work to make d the exact distance and in this way to find the minimum cut. For this purpose we execute several extra sweeps, performing only region-relabel and gap heuristics until labels stop changing. We claim that at most d^∞ such extra sweeps are needed. We give a proof for the case of push-relabel distance.

Proof Let us call labels $d(v)$ *loose* if $d(v) < d^*(v)$ and exact if $d(v) = d^*(v)$. Consider the lowest loose label, $L = \min\{d(v) \mid d(v) < d^*(v)\}$ and the set of loose vertices having this label, $\mathcal{L} = \{v \mid L = d(v) < d^*(v)\}$. Let us show that after a sweep of region-relabel, the value of L increases at least by 1. Let $v \in \mathcal{L}$, $(v,w) \in E$ and $c(v,w) > 0$. If $d(w)$ is loose, we have $d(w) \geq L$ by construction. Assume that $d(w)$ is exact. Since $d(v) < d^*(v)$ and $d^*(v) \leq d^*(w) + 1$, we have $d(w) \geq d(v) = L$. Therefore, all neighbors of v have label L or above. After the elementary Relabel of v or Region-relabel of the region including v, its label will increase at least by 1 (recall that Relabel of v performs $d(v) := \min_w\{d(w) \mid (v,w) \in E,\ c(v,w) > 0\} + 1$). Because this holds for all vertices from $\mathcal{L}$, the value L will increase at least by 1 after elementary Relabel of all vertices or a sweep of Region-relabel. Because L is bounded above by d^∞, after at most d^∞ sweeps, d will be the exact distance. □

This proof can be suitably modified for the case of region distance (used in ARD) by replacing the pair (v,w) with a path from v to a boundary vertex w. In this case, we have the bound $d^\infty = |\mathcal{B}|$ sweeps. In the experiments, we observed that in order to compute the exact distance, only few extra sweeps were necessary (from 0 to 2) for S/P-ARD and somewhat more for S/P-PRD. Note, to compute the final reachability relation in S/P-PRD, the region distance and ARD Region-relabel could be employed. However, we did not implement this improvement. In §6 we describe how ARD Region-relabel is replaced by a dynamic data structure (search trees), allowing for quick recomputation during the sweeps.

5.3 Referenced Implementations

Boykov-Kolmogorov (BK). The reference augmenting path implementation by Boykov and Kolmogorov (2004) (v3.0, `http://www.cs.adastral.ucl.ac.uk/~vnk/software.html`). We will also use the possibility of dynamic updates in this code due to Kohli and Torr (2005). There is only a trivial $O(mn^2|C|)$ complexity bound known for this algorithm[5], where C is the cost of a minimum cut.

Highest level Push-Relabel (HIPR). The reference push-relabel implementation by Cherkassky and Goldberg (1994) (v3.6, `http://www.avglab.com/andrew/soft.html`). This implementation has two stages: finding the maximum preflow / minimum cut and upgrading the maximum preflow to a maximum flow. Only the first stage was executed and benchmarked. We tested two variants with frequency of

[4] Region-gap-relabel (Delong and Boykov 2008, fig. 10) seems to contain an error: only vertices above the gap should be processed in step 3.

[5] The worst-case complexity of breadth-first search shortest path augmentation algorithm is just $O(m|C|)$. The tree adaptation step, introduced by Boykov and Kolmogorov (2004) to speed-up the search, does not have a good bound and introduces an additional n^2 factor.

the global relabel heuristic (the frequency parameter roughly corresponds to the proportion of time spent on global updates versus push/relabel) equal to 0.5 (the default value in HIPR v3.6) and equal to 0. These variants will be denoted HIPR0.5 and HIPR0 respectively. HIPR0 executes only one global update at the beginning. Global updates are essential for difficult problems. However, HIPR0 was always faster than HIPR0.5 in our experiments with real test instances[6]. The worst case complexity is $O(n^2\sqrt{m})$.

5.4 S/P-ARD implementation

The basic implementation of `ARD` simply invokes BK solver as follows. On stage 0 we compute the maximum flow in the network G^R by BK, augmenting paths from source to the sink. On the stage i, infinite capacities are added from the boundary vertices having label $i-1$ to the sink, using the possibility of dynamic changes in BK. The flow augmentation to the sink is then continued, reusing the search trees. The Region-relabel procedure is implemented as described earlier in this section. In the beginning of next discharge, we clear the infinite link from boundary to the sink and repeat the above. Some parts of the sink search tree, linked through the boundary vertices, get destroyed, but the larger part of it and the source search tree are reused. A more efficient implementation is described in §6. It includes additional heuristics and maintenance of separate boundary search trees.

S-ARD. In the streaming mode, we keep only one region in the memory at a time. After a region is processed by `ARD`, all the internal data structures have to be saved to the disk and cleared from memory until the region is discharged next time. We manage this by allocating all the region's data into a fixed page in the memory, which can be saved and loaded preserving the pointers. By doing the load/unload manually (rather than relying on the system swapping mechanism), we can accurately measure the pure time needed for computation (CPU) and the amount of disk I/O. We also can use 32bit pointers with larger problems.

A region with no active vertices is skipped. The global gap heuristic is executed after each region discharge. Because it is based on labels of boundary vertices only, it is sufficient to maintain a label histogram with $|\mathcal{B}|$ bins to implement it. S-ARD uses $O(|\mathcal{B}| + |(\mathcal{B},\mathcal{B})|)$ "shared" memory and $O(|V^R + E^R|)$ "region" memory, to which regions are loaded one at a time.

To solve large problems, which do not fit in the memory, we have to create region graphs without ever loading the full problem. We implemented a tool called **splitter**, which reads the problem from a file and writes edges corresponding to the same region to the region's separate "part" file. Only the boundary edges (linking different regions) are withheld to the memory.

P-ARD. We implemented this algorithm for a shared-memory system using OpenMP language extension. All regions are kept in the memory, the discharges are executed concurrently in separate threads, while the gap heuristic and messages exchange are executed synchronously by the master thread.

5.5 S/P-PRD implementation

To solve region discharge subproblems in PRD in the highest label first fashion, we designed a special reimplementation of HIPR, which will be denoted **HPR**. We intended to use the original HIPR implementation to make sure that PRD relies on the state-of-the art core solver. It was not possible directly. A subproblem in PRD is given by a region network with fixed distance labels on the boundary (let us call them *seeds*). Distance labels in PRD may go up to n in the worst case. The same applies to region subproblems as well. Therefore, keeping an array of buckets corresponding to possible labels (like in HIPR), would not be efficient. It would require $O(|V|)$ memory and an increased complexity. However, because a region has only $|V^R|$ vertices, there are no more than $|V^R|$ distinct labels at any time. This allows to keep buckets as a doubly-linked list with at most $|V^R|$ entries. Highest label selection rule and the region-gap heuristic can then be implemented efficiently with just a small overhead. We tried to keep other details similar to HIPR (current arc data structure, etc.). HPR with arbitrary seeds has the worst case complexity $O(|V^R|^2\sqrt{|E^R|})$ and uses $O(|V^R| + |V^E|)$ space. When the whole problem is taken as a single region, HPR should be equivalent to HIPR0. Though the running time on the real instances can be somewhat different.

S-PRD. This is our reimplementation of the algorithm by Delong and Boykov (2008) for an arbitrary graph and a fixed partition, using HPR as a core solver. It uses the same memory model, paging mechanism and the splitter tool as S-ARD. The region discharge is always warm-started. We found it inefficient to run the region-relabel after every discharge. In the current experiments, motivated by performance of HIPR0, we run it once at the beginning and then only when a global gap is discovered. To detect a global gap, we keep a

[6] There is a discrepancy with Delong and Boykov (2008, Figure 4) regarding the results for the basic push-relabel. The main implementation difference is in the order of processing (HIPR versus FILO). It is also possible that their plot is illustrative and is not using the gap heuristic.

histogram of all labels, $O(n)$ memory, and update it after each region discharge (in $O(|V^R|)$ time). In practice, this $O(n)$ memory is not a serious limitation – labels are usually well below n. If they are not then we should consider a weaker gap heuristic with a smaller number of bins. Applying the gap (raising the corresponding vertices to d^∞) for all regions is delayed until they are loaded. So we keep the track of the best global gap detected for every region. Similar to how the sequential Alg. 1 represents both S-ARD and S-PRD, it constitutes a piece of generic code in our implementation, where the respective discharge procedure and gap heuristics are plugged.

P-PRD. This is our implementation of parallel `PRD` for shared-memory system with OpenMP.

6 Efficient ARD Implementation

The basic implementation of S-ARD, as described in the previous section, worked reasonably fast (comparable to BK) on simple problems like 2D stereo and 2D random segmentation (Sec. 7.1). However, on some 3D problems the performance was unexpectedly bad. For example, to solve `LB07-bunny-lrg` instance (Sec. 7.2) the basic implementation required 32 minutes of CPU time. In this section we describe an efficient implementation which is more robust and is comparable in speed with BK on all tested instances. In particular, to solve `LB07-bunny-lrg` it takes only 15 seconds of CPU time. The problem why the basic implementation is so slow is in the nature of the algorithm: sometimes it has to augment the flow to the boundary, without knowing of whether it is a useful work or not. If the particular boundary was selected wrongly, the work is wasted. This happens in `LB07-bunny-lrg` instance, where the data seeds are sparse. A huge work is performed to push the flow around in the first few iterations, before a reasonable labeling is established. We introduce two heuristics how to overcome this problem: the boundary-relabel heuristic and partial discharges. An additional speed-up is obtained by dynamically maintaining boundary search trees and the current labeling.

6.1 Boundary Relabel Heuristic

We would like to have a better distance estimate, but we cannot run a global relabel because implementing it in a distributed fashion would take several full sweeps, which would be too wasteful. Instead, we go for the following cheaper lower bound. Our implementation keeps all the boundary edges (including their flow and distance labels of the adjacent vertices) in the shared memory. Fig. 6(a) illustrates this boundary information. We want to improve the labels by analyzing only this boundary part of the graph, not looking inside the regions. Since we do not know how the vertices are connected inside the regions, we have to assume that every boundary vertex might be connected to any other one within the region, except of the following case. If u and v are in the same region R and $d(u) > d(v)$ then we know for sure that $u \nrightarrow v$ in G^R. It follows from the validity of labeling d (as defined for ARD in §4). We can calculate now a lower bound on the distance $d^{*\mathcal{B}}$ in G assuming that all the rest of the vertices are potentially connected within the regions.

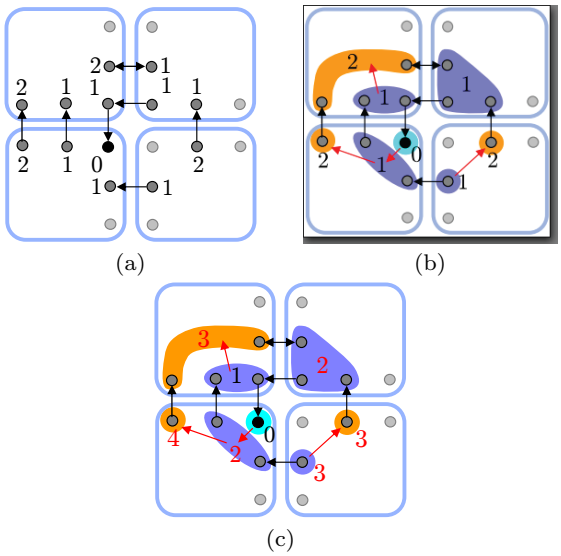


Fig. 6 Boundary relabel heuristic: (a) Boundary vertices of the network and a valid labeling. Directed arcs correspond to non-zero residual capacities. Vertices without numbers have label d^∞ and do not participate in the construction. (b) Vertices having the same label are grouped together within each region and arcs of zero length (of red color) are added from a group to the next label's group. It is guaranteed that *e.g.*, vertices with label 1 are not reachable from vertices with label 2 within the region, hence there is no arc 2→1. Black arcs have the unit length. (c) The distance in the auxiliary graph is a valid labeling and a lower bound on the distance in the original network.

We will now construct an auxiliary directed graph $\bar{G}$ with arcs having length 0 or 1 and show that the distance in this graph (according to the arc lengths) lower bounds $d^{*\mathcal{B}}$. If $d(v) = d(u)$ we have to assume that $v \to u$ and $u \to v$ in G^R, therefore the new lower bound for u and v will coincide. Hence we group vertices having the same label within a region together as shown in Fig. 6(b). In the case $d(v) < d(u)$, we know that $u \nrightarrow v$ but have to assume $v \to u$ in R. We thus add a directed arc of length zero from the group of v to the group of u (Fig. 6(b)). Let $d_1 < d_2 < d_3$ be labels

of groups within one region. There is no need to create an arc from d_1 to d_3, because two arcs from d_1 to d_2 and from d_2 to d_3 of length zero are an equivalent representation. Therefore it is sufficient to connect only groups having consecutive labels. We then add all residual edges (u, v) between the regions to $\bar{G}$ with length 1. We can calculate the distance to vertices with label 0 in $\bar{G}$ by running Dijkstra's algorithm. Let this distance be denoted d'. We then update the labels as

$$d(u) := \max\{d(u), d'(u)\}. \tag{13}$$

We have to prove the following two points:

1. d' is a valid labeling;
2. If d and d' are valid labellings, then $\max(d, d')$ is valid.

Proof 1. Let $c(u, v) > 0$. Let u and v be in the same region. It must be that $d(u) \leq d(v)$. Therefore either u and v are in the same group or there is an arc of length zero from group of u to group of v. It must be $d'(u) \leq d'(v)$ in any case. If u and v are in different regions, there is an arc of length 1 from group of u to group of v and therefore $d'(u) \leq d'(v) + 1$.

2. Let $l(u, v) = 1$ if $(u, v) \in (\mathcal{B}, \mathcal{B})$ and $l(u, v) = 0$ otherwise. We have to prove that if $c(u, v) > 0$ then

$$\max\{d(u), d'(u)\} \leq \max\{d(v), d'(v)\} + l(u, v). \tag{14}$$

Let $\max\{d(u), d'(u)\} = d(u)$. From validity of d we have $d(u) \leq d(v) + l(u, v)$. If $d(v) \geq d'(v)$, then $\max\{d(v), d'(v)\} = d(v)$ and (14) holds. If $d(v) < d'(v)$ then $d(u) \leq d(v) + l(u, v) < d'(v) + l(u, v)$ and (14) holds again.

The complexity of the boundary relabel heuristic is $O(|(\mathcal{B}, \mathcal{B})|)$. It is relatively inexpensive and can be run after each sweep. It does not affect the correctness and the worst case bound on the number of sweeps of S/P-ARD.

6.2 Partial Discharge

Another heuristic which proved very efficient was simply to postpone path augmentations to higher boundary vertices to further sweeps. This allows to save a lot of unnecessary work, especially when used in combination with boundary-relabel. More precisely, on sweep s the `ARD` procedure is allowed to execute only stages up to s. This way, in sweep 0 only paths to the sink are augmented and not any path to the boundary. Vertices which cannot reach the sink (but can potentially reach the boundary) get label 1. These initial labels may already be improved by boundary-relabel. In sweep 1 paths to the boundary with label 0 are allowed to be augmented and so on.

Note that this heuristic does not affect the worst case complexity of S/P-ARD. Because labels can grow only up to $|\mathcal{B}|$, after at most $|\mathcal{B}|$ sweeps the heuristic turns into full discharge. Therefore, the worst case bound of $O(|\mathcal{B}^2|)$ sweeps remains valid. In practice, we found it to increase the number of sweeps slightly, while significantly reducing the total computation time. Similarly, in the case of push-relabel, it would make sense to perform several sweeps of Region-relabel before doing any pushes to get a better estimate of the distance.

6.3 Boundary Search Trees

We now redesign the implementation of `ARD` such that not only the sink and source search trees are maintained but also the search trees of boundary vertices. This allows to save computation when the labeling of many boundary vertices remains constant during the consequent sweeps, with only a small fraction changing. Additionally, knowing the search tree for each inner vertex of the region determines its actual label, so the region-relabel procedure becomes obsolete. The design of the search tree data structures, their updates and other detail are the same as proposed by Kolmogorov (2004), only few changes to the implementation are necessary. For each vertex $v \in R$, we introduce a mark $\tilde{d}(v)$ which corresponds to the root label of its tree or is set to a special *free* mark if v is not in any tree. For each tree we keep a list of open vertices (called active by Kolmogorov (2004)). A vertex is *open* if it is not blocked by the vertices of the trees with the same or lower root label (more precisely, v is open if it is not free and there is a residual edge (u, v) such that u is free or its root label is higher than that of v). The trees may grow only at the open vertices.

Fig. 7 shows the correspondence between search trees and the labels. The sink search tree is assigned label -1. In the stage 0 of `ARD`, we grow the sink tree and augment all found paths if the sink tree touches the source search tree. Vertices, which are added to the sink tree are marked with label $\tilde{d} = -1$. In stage $i + 1$ of `ARD`, we grow trees with root at a boundary vertices w with label $d(w) = i$, all vertices added to the tree are marked with $\tilde{d} = i$. When the tree touches the source search tree, the found path is augmented. If the tree touches a vertex u with label $\tilde{d}(u) < i$, it means that u is already in the search tree with a lower root and no action is taken. It cannot happen that a vertex is reached with label $\tilde{d} > i$ during growth of a search tree

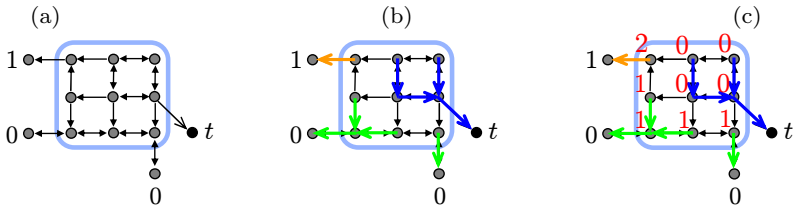


Fig. 7 Search trees. (a) A region with some residual arcs. The region has only 3 boundary vertices, for simplicity, the numbers correspond to the labels. (b) Search trees of the sink and boundary vertices: when a vertex can be reached by several trees, it choses the one with the lowest label of the root. The sink is assigned a special label -1. The source search tree is empty in this example. (c) Labels of the inner vertices are determined as their tree's root label+1.

with root label i, this would contradict to the properties of ARD. The actual label of a vertex v at any time is determined as $\tilde{d}(v)+1$ if $v \in R$ and $d(v)$ if $v \in B^R$.

Let us now consider the situation in which region R has built some search trees and the label of a boundary vertex w is risen from $d(w)$ to $d'(w)$ (as a result of update from the neighboring region or one of the heuristics). All the vertices in the search tree starting from w were previously marked with $d(w)$ and have to be declared as free vertices or adopted to any other valid tree with root label $d(w)$. The adaptation is performed by the same mechanism as in BK. The situation when a preflow is injected from the neighboring region and (a part of) a search tree becomes disconnected is also handled by the orphan adaptation mechanism.

The combination of the above improvements allows S-ARD to run in about the same time as BK on all tested vision instance (§7.2), sometimes being even significantly faster (154sec. vs. 245sec. on `BL06-gargoyle-lrg`).

7 Experiments

All experiments were conducted on a system with Intel Core 2 Quad CPU@2.66Hz, 4GB memory, Windows XP 32bit and Microsoft VC compiler. Our implementation and instructions needed to reproduce the experiments can be found at `http://cmp.felk.cvut.cz/~shekhovt/d_maxflow`. We conducted 3 series of experiments:

- Synthetic experiments, where we observe general dependencies of the algorithms, with some statistical significance, *i.e.* not being biased to a particular problem instance. It also serves as an empirical validation, as thousands of instances are solved. Here, the basic implementation of S-ARD was used.
- Sequential competition. We study sequential versions of the algorithms, running them on real vision instances. Only a single core of the CPU is utilized. We fix the region partition and study how much disk I/O it would take to solve each problem when only one region can be loaded in the memory at a time. In this and the next experiment we used the efficient implementation of ARD. Note, in the preceeding publication (Shekhovtsov and Hlavac 2011) we reported worse results with the earlier implementation.
- Parallel competition. Parallel algorithms are tested on the instances which can fully fit in 2GB of memory. All 4 cores of the CPU are allowed to be used. We compare our algorithms with two other state-of-the-art distributed implementations.

7.1 General Dependences: Synthetic Problems

We generated simple synthetic problems to validate the algorithms. The network is constructed as a 2D grid with a regular connectivity structure. Fig. 8(a) shows an example of such a network. The edges are added to the vertices at the following relative displacements $(0,1)$, $(1,0)$, $(1,2)$, $(2,1)$, $(1,3)$, $(3,1)$, $(2,3)$, $(3,2)$, $(0,2)$, $(2,0)$, $(2,2)$, $(3,3)$, $(3,4)$, $(4,2)$. By *connectivity* we mean the number of edges incident to a vertex far enough from the boundary. Adding pairs $(0,1)$, $(1,0)$ results in connectivity 4 and so on. Each vertex is given an integer excess/deficit distributed uniformly in the interval $[-500\ 500]$. A positive number means a source link and a negative number a sink link. All edges in the graph are assigned a constant capacity, called *strength*. The network is partitioned into regions by slicing it in s equal parts in both dimensions. Thus we have 4 parameters: the number of vertices, the connectivity, the strength and the number of regions. We generate 100 instances for each value of the parameters.

Let us first look at the dependence on the strength shown in Fig. 8(b). Problems with small strength are easy, because they are very local – long augmentation paths do not occur. On the other hand, long paths needs to be augmented for problems with large strength. However, finding them is easy because bottlenecks are un-

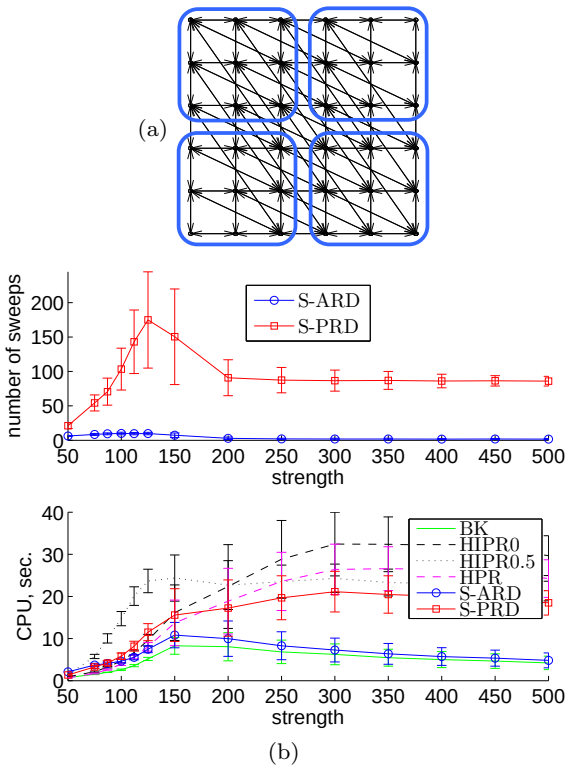


Fig. 8 (a) Example of a synthetic problem: a network of the size 6×6, connectivity 8, partitioned into 4 regions. The source and sink are not shown. (b) Dependence on the interaction strength, for size 1000×1000, connectivity 8 and 4 regions. Plots show mean values and intervals containing 70% of the samples.

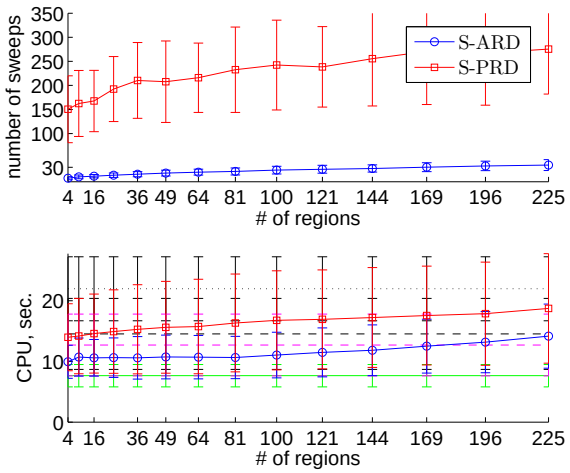


Fig. 9 Dependence on the number of regions, for size 1000×1000, connectivity 8, strength 150.

likely. Therefore BK and S-ARD have a maximum in the computation time somewhere in the middle. It is more difficult to transfer the flow over long distances for push-relabel algorithms. This is where the global relabel heuristic becomes efficient and HIPR0.5 outperforms HIPR0. The region-relabel heuristic of S-PRD allows it to outperform other push-relabel variants.

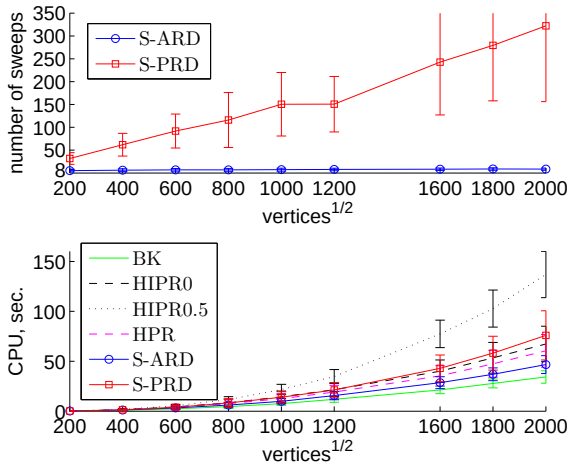


Fig. 10 Dependence on the problem size, for connectivity 8, strength 150, 4 regions.

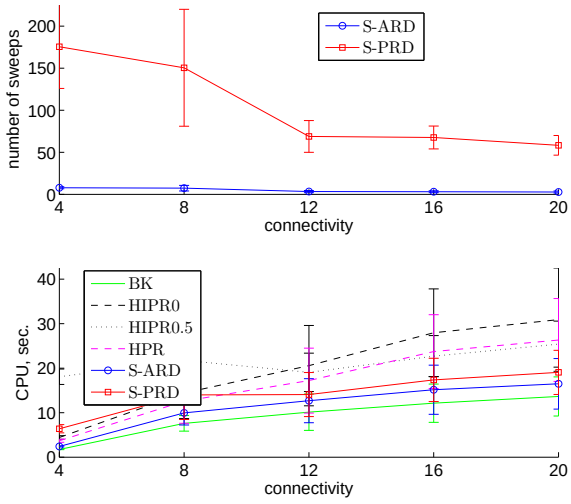


Fig. 11 Dependence on the connectivity, for size 1000×1000, strength = $(150 \cdot 8)/\text{connectivity}$, 4 regions.

In general, we think all such random 2D networks are too easy. Nevertheless, they are useful and instructive to show basic dependences. We now select the "difficult" point for BK with the strength 150 and study other dependencies:

- The number of regions (Fig. 9). For this problem family, both the number of sweeps and the computation time grows slowly with the number of regions.
- The problem size (Fig. 10). Computation efforts of all algorithms grow proportionally. However, the number of sweeps shows different asymptotes. It is almost constant for S-ARD but grows significantly for S-PRD.

– Connectivity (Fig. 11). Connectivity is not independent of the strength. Roughly, 4 edges with capacity 100 can transmit as much flow as 8 edges with capacity 50. Therefore while increasing the connectivity we also decrease the strength as $150 \cdot 8$ divided by connectivity in this plot.
– Workload (Fig. 7.1). This plot shows how much time each of the algorithms spends performing different parts of computation. Note that the problems are solved on a single computer with all regions kept in memory, therefor the time on sending messages should be understood as updates of dynamic data structure of the region w.r.t. the new labeling and flow on the boundary. For S-PRD more sweeps are needed, so the total time spent in messages and gap heuristic is increased. Additionally, the gap heuristic has to take into account all vertices, unlike only the boundary vertices in S-ARD.

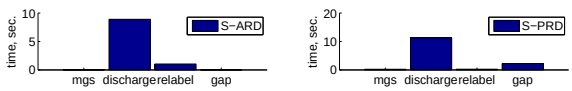


Fig. 12 Workload distribution, for size 1000×1000, connectivity 8, 4 regions, strength 150. *msg* – passing the messages (updating flow and labels on the boundary), *discharge* – work done by the core solver (BK for S-ARD and HPR for S-PRD), *relabel* – the region-relabel operation, *gap* – the global gap heuristic.

7.2 Sequential Competition

We tested our algorithms on the MAXFLOW problem instances in computer vision University of Western Ontario web pages (2008). The data consist of typical max-flow problems in computer vision, graphics, and biomedical image analysis. **Stereo** instances are sequences of subproblems (arising in the expansion move algorithm) for which the total time should be reported. There are two models: BVZ (Boykov et al 1998), in which the graph is a 4-connected 2D grid, and KZ2 (Kolmogorov and Zabih 2001), in which there are additional long-range links. **Multiview** 3D reconstruction models LB06 (Lempitsky et al 2006) and BL06 (Boykov and Lempitsky 2006). Graphs of these problems are cellular complexes subdividing the space into 3D cubes and each cube into 24 smaller cells. **Surface** fitting instances LB07 (Lempitsky and Boykov 2007) are 6-connected 3D grid graphs. And finally, there is a collection of volumetric **segmentation** instances BJ01 (Boykov and Jolly 2001), BF06 (Boykov and Funka-Lea 2006), BK03 (Boykov and Kolmogorov 2003) with 6-connected and 26-connected 3D grid graphs.

To test our streaming algorithms, we used the `regulargrid` hint available in the definition of the problems to select the regions by slicing the problem into 4 parts in each dimension – into 16 regions for 2D BVZ grids and into 64 regions for 3D segmentation instances. Problems KZ2 do not have such a hint (they are not regular grids), so we sliced them into 16 pieces just by the vertex number. The same we did for the multiview LB06 instances. Though they have a `size` hint, we failed to interpret the vertex layout correctly (the separator set, $\mathcal{B}$, was unexpectedly large when trying to slice along the dimensions). So we sliced them purely by the vertex number.

One of the problems we faced is pairing the arcs which are reverse of each other. While in stereo, surface and multiview problems, the reverse arcs are consequent in the files, and can be easily paired, in 3D segmentation they are not. For a generic algorithm, not being aware of the problem's regularity structure, it is actually a non-trivial problem requiring at least the memory to read all of the arcs first. Because our goal is a relative comparison, we did not pair the arcs in 3D segmentation. This means we kept twice as many arcs than necessary for those problems. This is seen in Table 1, *e.g.* for `babyface.n26c100`, which is 26-connected, but we construct a multigraph (has parallel arcs) with average vertex degree of 49. For some other instances, however, this is not visible, because there could be many zero arcs, *e.g.* `liver.n26c10` which is a 26-connected grid too, but has the average vertex degree of 10.4 with unpaired arcs. The comparison among different methods is correct, since all of them are given exactly the same multigraph.

The results are presented in Table 1. We did measure the real time of disk I/O. However, it depends on the hard drive performance, other concurrently running processes as well as on the system file caching (which has effect for small problems). We therefore report total bytes written/loaded and give an estimate of the full running time for the disk speed of 100MB/s (see Table 2). Note that disk I/O is not proportional to the number of sweeps, because some regions may be inactive during a sweep and thus skipped. For HIPR we do not monitor the memory usage. It is slightly higher than that of HPR, because of keeping initial arc capacities.

For verification of solvers, we compared the flow values to the ground truth solution provided in the dataset. Additionally, we saved the cut output from each solver and checked its cost independently. Verifying the cost of the cut is relatively easy: the cut can be kept in memory and the edges can be processed form the DIMACS problem definition file on-line. An independent check of (pre-)flow feasibility would be neces-

Table 1. Sequential Competition. CPU – the time spent purely for computation, excluding the time for parsing, construction and disk I/O. The total time to solve the problem is not shown. K – number of regions. RAM – memory taken by the solver; for BK in the case it exceeds 2GB limit, the expected required memory; for streaming solvers the sum of shared and region memory. I/O – total bytes read or written to the disk.

problem			BK	HIPR0	HIPR0.5	HPR	S-ARD			S-PRD		
name size	n(10^6)	m/n	CPU RAM	CPU RAM	CPU RAM	CPU RAM	CPU RAM	sweeps	K I/O	CPU RAM	sweeps	K I/O
stereo												
BVZ-sawtooth(20) 434×380, 14MB	0.2	4.0	0.68s 14MB	3.0s	7.7s	3.8s 17MB	0.63s 0.3+0.9MB	6	16 114MB	3.7s 0.8+1.1MB	32	16 0.7GB
BVZ-tsukuba(16) 384×288, 8.6MB	0.1	4.0	0.36s 9.7MB	1.9s	4.9s	2.6s 11MB	0.35s 0.2+0.6MB	5	16 71MB	2.1s 0.5+0.8MB	29	16 373MB
BVZ-venus(22) 434×383, 14MB	0.2	4.0	1.2s 15MB	5.7s	15s	6.2s 17MB	1.1s 0.3+0.9MB	6	16 119MB	6.6s 0.8+1.1MB	36	16 0.9GB
KZ2-sawtooth(20) 38MB	0.3	5.8	1.8s 33MB	7.1s	22s	6.1s 36MB	2.2s 1.2+2.0MB	6	16 280MB	7.4s 1.8+2.5MB	23	16 1.2GB
KZ2-tsukuba(16) 26MB	0.2	5.9	1.1s 23MB	5.3s	20s	4.4s 25MB	1.4s 1.1+1.4MB	6	16 186MB	5.9s 1.4+1.7MB	18	16 0.7GB
KZ2-venus(22) 38MB	0.3	5.8	2.8s 34MB	13s	39s	10s 37MB	3.4s 1.2+2.1MB	8	16 330MB	14s 1.9+2.5MB	36	16 1.8GB
multiview												
BL06-camel-lrg 100×75×105×24, 2.0GB	18.9	4.0	81s 1.6GB				63s 19+103MB	11	16 28GB	308s 86+122MB	418	16 0.6TB
BL06-camel-med 80×60×84×24, 1.0GB	9.7	4.0	25s 0.8GB	29s	77s	59s 1.0GB	20s 31+53MB	12	16 16GB	118s 46+63MB	227	16 225GB
BL06-camel-sml 40×30×42×24, 115MB	1.2	4.0	0.98s 106MB	1.5s	6.3s	1.8s 124MB	0.96s 8.0+7.0MB	9	16 1.4GB	4.2s 6.9+8.2MB	47	16 9.1GB
BL06-gargoyle-lrg 80×112×80×24, 1.8GB	17.2	4.0	245s 1.5GB			91s 1.7GB	154s 23+95MB	21	16 35GB	318s 82+112MB	354	16 0.8TB
BL06-gargoyle-med 64×90×64×24, 0.9GB	8.8	4.0	115s 0.8GB	17s	58s	37s 0.9GB	73s 37+50MB	16	16 14GB	143s 44+58MB	340	16 235GB
BL06-gargoyle-sml 32×45×32×24, 106MB	1.1	4.0	6.1s 97MB	1.2s	3.0s	1.7s 114MB	3.9s 9.3+6.6MB	10	16 1.3GB	4.4s 6.9+7.7MB	55	16 9.4GB
surface												
LB07-bunny-lrg 401×396×312, 6.6GB	49.5	6.0	5.7GB				15s 130+87MB	6	64 49GB	416s 226+99MB	43	64 276GB
LB07-bunny-med 202×199×157, 0.8GB	6.3	6.0	1.6s 0.7GB	20s	41s	26s 0.8GB	2.1s 33+12MB	10	64 6.5GB	16s 43+863MB	27	64 0.0MB
LB07-bunny-sml 102×100×79, 94MB	0.8	5.9	0.17s 95MB	0.80s	1.8s	1.1s 101MB	0.32s 8.2+1.6MB	9	64 0.8GB	0.86s 7.9+1.9MB	19	64 2.0GB
segm												
liver.n26c10 170×170×144, 2.1GB	4.2	10.4	6.4s 0.8GB	18s	18s	34s 0.7GB	14s 36+12MB	13	64 13GB	39s 30+13MB	157	64 82GB
liver.n26c100 170×170×144, 2.1GB	4.2	11.1	12s 0.8GB	26s	28s	39s 0.7GB	24s 38+13MB	15	64 16GB	35s 30+14MB	98	64 66GB
liver.n6c10 170×170×144, 498MB	4.2	9.8	7.2s 0.7GB	17s	25s	40s 0.7GB	14s 33+12MB	16	64 15GB	36s 28+13MB	151	64 79GB
liver.n6c100 170×170×144, 512MB	4.2	10.5	15s 0.8GB	30s	34s	44s 0.7GB	19s 35+12MB	17	64 14GB	32s 29+13MB	94	64 70GB
babyface.n26c10 250×250×81, 2.5GB	5.1	47.3	3.7GB				179s 156+56MB	38	64 102GB	222s 173+58MB	169	64 0.6TB
babyface.n26c100 250×250×81, 2.7GB	5.1	49.0	3.8GB				231s 156+56MB	44	64 115GB	262s 180+57MB	116	64 0.6TB
babyface.n6c10 250×250×81, 0.6GB	5.1	11.1	6.8s 1.0GB	38s	51s	68s 0.9GB	20s 22+16MB	17	64 19GB	100s 37+17MB	275	64 261GB
babyface.n6c100 250×250×81, 0.6GB	5.1	11.5	13s 1.0GB	71s	65s	87s 0.9GB	24s 22+16MB	19	64 18GB	74s 37+17MB	191	64 189GB
adhead.n26c10 256×256×192, 6.5GB	12.6	31.5	6.2GB				128s 153+83MB	17	64 84GB	224s 195+86MB	109	64 0.8TB
adhead.n26c100 256×256×192, 1.6GB	12.6	31.6	2.5GB				174s 34+36MB	21	64 36GB	269s 77+39MB	129	64 354GB
bone.n26c10 256×256×119, 3.9GB	7.8	32.3	4.0GB				25s 122+61MB	12	64 35GB	96s 147+63MB	148	64 470GB
bone.n26c100 256×256×119, 4.1GB	7.8	32.4	4.0GB				29s 122+61MB	14	64 39GB	68s 147+63MB	124	64 321GB
bone.n6c10 256×256×119, 0.9GB	7.8	11.5	7.7s 1.5GB	5.7s	17s	12s 1.4GB	7.2s 62+23MB	9	64 13GB	37s 52+25MB	195	64 188GB
bone.n6c100 256×256×119, 1.0GB	7.8	11.6	9.1s 1.6GB	9.1s	22s	14s 1.5GB	8.7s 62+23MB	10	64 13GB	23s 52+25MB	65	64 104GB
bone_subx.n6c100 128×256×119, 495MB	3.9	11.8	7.1s 0.8GB	6.3s	12s	6.4s 0.7GB	5.5s 39+12MB	12	64 7.1GB	9.4s 29+13MB	42	64 42GB
bone_subxy.n26c100 128×128×119, 1.0GB	1.9	32.2	5.9s 1.0GB	3.9s	6.1s	4.6s 0.8GB	7.3s 92+13MB	13	64 10GB	8.7s 50+16MB	33	64 39GB
abdomen_long.n6c10 512×512×551, 19GB	144.4	11.8	29GB				179s 410+403MB	11	196GB		> 35	>1TB
abdomen_short.n6c10 512×512×551, 19GB	144.4	11.8	29GB				82s 410+403MB	11	138GB			

problem	S-ARD		S-PRD	
	mem, MB	time	mem, MB	time
BVZ-sawtooth	1.1	1.7s	1.9	11s
BL06-camel-lrg	122	6min	208	1.8h
BL06-gargoyle-lrg	118	8.5min	194	2.4h
LB07-bunny-lrg	217	8.6min	325	54min
babyface.n26c10	212	20min	231	1.9h
adhead.n26c10	236	16min	281	2.3h
adhead.n26c100	70	9min	116	1h
bone.n26c10	183	6.3min	210	1.4h
abdomen long.n6c10	813	36min	∼800	>3h

Table 2 Estimated running time for the algorithms in the streaming mode, including the time for Disk I/O. The estimate is computed for a disk speed of 100MB/s and does not include initial problem splitting. The table also gives the total amount of memory used by the solver.

sary for full verification of a solver. However, it would require storing the full graph in the memory and was not implemented.

Our new algorithms computed flow values for all problems matching those provided in the dataset, except for the following cases:

- `LB07-bunny-lrg`: no ground truth solution available (we found flow/cut of cost 15537565).
- `babyfacen26c10` and `babyfacen26c100`: we found higher flow values than those which were provided in the dataset (we found flow/cut of cost 180946 and 1990729 resp.).

The latter problems appear to be the most difficult for S-ARD in terms of both time and number of sweeps. Despite this, S-ARD requires much fewer sweeps, and consequently much less disk I/O operations than the push-relabel variant. This means that in the streaming mode, where read and write operations take a lot of time, S-ARD is clearly superior. Additionally, we observe that the time it spends for computation is comparable to that of BK, sometimes even significantly smaller.

Next, we studied the dependency of computation time and number of sweeps on the number of regions in the partition. We selected 3 representative instances of different problems and varied the number of regions in the partition. The results are presented in the Fig. 13. The instance `BL06-gargoyle-sml` was partitioned by the vertex index and the remaining two problems were partitioned according to their 3D vertex layout, using variable number of slices in each dimension. The results demonstrate that the computation time required to solve these problems is stable over a large range of partitions and the number of sweeps required does not grow rapidly. Therefore, for the best practical performance the partition for S-ARD can be selected to meet other requirements: memory consumption, number of

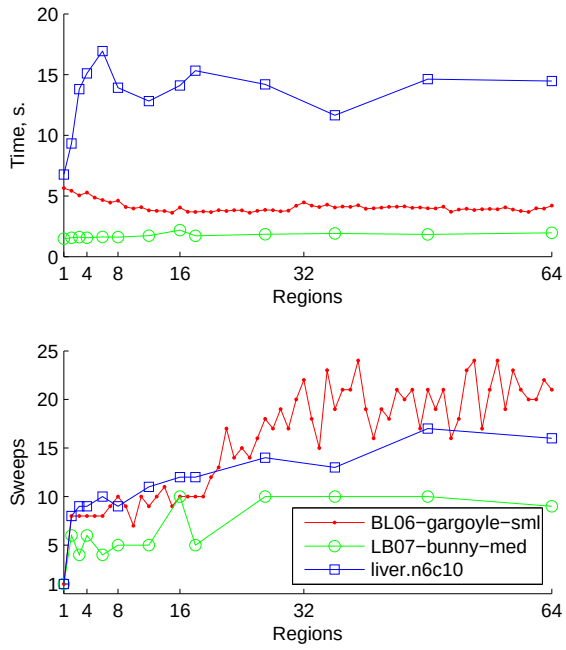


Fig. 13 Dependence on the number of regions for the representative instances of multiview, stereo and segmentation. Top: CPU time used. Bottom: number of sweeps.

computation units, *etc.* We should note however, that with refining the partition the amount of shared memory grows proportionally to the number of boundary edges. In the limit of single-vertex regions, the algorithm will turn into a very inefficient implementation of pure push-relabel.

7.3 Parallel Competition

In this section, we test parallel versions of our algorithms and compare them with two state-of-the-art methods. The experiments are conducted on the same machine as above (Intel Core 2 Quad CPU@2.66Hz) but allowing the use of all 4 CPUs. The goal is to see how the distributed algorithms perform in the simplified setting when they are run not in the network but on a single machine. For P-ARD/PRD we expect that the total required work would increase compared to the sequential versions because the discharges are executed concurrently. The relative speed-up therefore would be sublinear even if we managed to distribute the work between CPUs evenly. The tests are conducted on small and medium size problems (taking under 2GB of memory). For P-ARD and P-PRD we use the same partition into regions as in Table 1. For other solvers, discussed next, we tried to meet better their requirements.

DD. The integer dual decomposition algorithm by Strandmark and Kahl (2010)[7] uses adaptive vertex-wise step rule and randomization. With or without randomization, this algorithm is not guaranteed to terminate. However, while without randomization there is an example with 4 vertices such that the algorithm never terminates, with randomization there is always a chance that it does. Interestingly, on all of the stereo problems the algorithm terminated in a small number of iterations. However, on larger problems partitioned into 4 regions it exceeded the internal iterations bound (1000) in many cases and returned without optimal flow/cut. In such a case it provides only an approximate solution to the problem. Whether such a solution is of practical value is beyond us. We tested it with partitions into 2 and 4 regions (denoted **DDx2** and **DDx4** resp.). Naturally, with 2 regions the algorithm can utilize only 2 CPUs. When the number of regions is increased, the random event of termination is expected to happen less likely, which is confirmed by our experiments.

RPR.A recently published implementation of Region Push Relabel (Delong and Boykov 2008) by Sameh Khamis (v1.01, `http://vision.csd.uwo.ca/code/`).

For RPR we constructed partition of the problem into smaller blocks. Because regions in RPR are composed dynamically out of blocks (default is 8 blocks per region) we partitioned 2D problems into $64 = 8^2$ blocks and 3D problems into $512 = 8^3$ blocks. This partition was also empirically faster than a coarser one. The parameter `DischargesPerBlock` was set by recommendation of authors to 500 for small problems (stereo) and to 15000 for big problems. The implementation is specialized for regular grids, therefore multiview and KZ2 problems which do not have `regulargrid` hint cannot be solved by this method. Because of the fixed graph layout in RPR, arcs which are reverse of each other are automatically grouped together, so RPR computes on a reduced graph compared to other methods. Let us also note that because of the dynamic regions, RPR is not fully suitable to run in a distributed system.

The method of Liu and Sun (2010) (parallel, but not distributed) would probably be the fastest one in this competition (as could be estimated from the results reported by Liu and Sun (2010)), however the implementation is not publicly available.

Results. The results are summarized in Table 3. The time reported is the wall clock time passed in the calculation phase, not including any time for graph construction. The number of sweeps for DD has the same meaning as for P-ARD/PRD, it is the number of times all regions are synchronously processed. RPR however is asynchronous and uses dynamic regions. For it, we define sweeps = `block_discharges`/`number_of_blocks`.

[7] Multi-threaded maxflow library `http://www.maths.lth.se/matematiklth/personal/petter/cppmaxflow.php`

Comparing to Table 1, we see that P-ARD on 4 CPUs is about $1.5 - 2.5$ times faster than S-ARD. The speed-up over BK varies from 0.8 on `livern6c10` to more than 4 on `gargoyle`.

We see that DD gets lucky some times and solves the problem really quickly, but often it fails to terminate. We also observe that our variant of P-PRD (based on highest first selections rule) is a relatively slow, but robust distributed method. RPR, which is based on LIFO selection rule, is competitive on the 3D segmentation problems but is slow on other problems, despite its compile-time optimization for the particular graph structure. It is also uses relatively higher number of blocks, The version we tested always returned the correct flow value but often a wrong (non-optimal) cut. Additionally, for 26 connected `bone_subxy.n26c100` it failed to terminated within 1 hour.

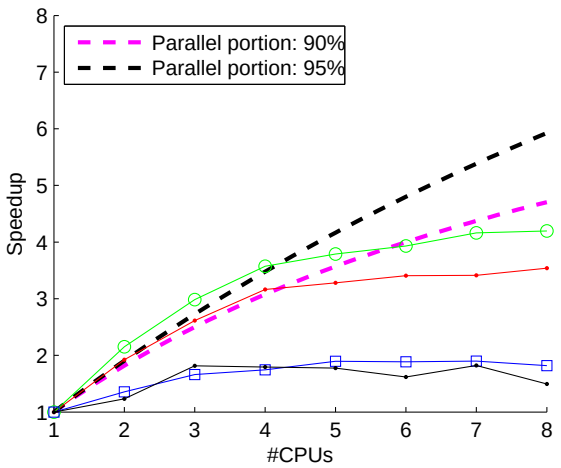
7.4 Scalability with Processors

We performed additional tests of P-ARD in the shared memory mode using 1-8 CPUs. This experiment is conducted on a system with Intel(R) Core(TM)i7 CPU 870@2.9GHz, 16GB memory, linux 64bit and gcc compiler. The plot in Fig. 14 shows the speedup of solving the problem (excluding initialization) using multiple CPUs over the time needed by a single CPU. For this test, we selected medium and large size problems of different kind which can fully fit in 16GB of memory. The two problems which were taking longer in the serial implementation scaled relatively well. On the other side, the largest `LB07-bunny` problem did not scale well. We believe that the limiting factor here is the memory bandwidth. We inspected that the sequential part of the computation (boundary relabel heuristic, synchronous message exchange) occupy less than 10% of the total time for all four problems. The fully parallel part should exhibit a linear speed-up in the ideal case of even load. The load for `LB07-bunny` should be relatively even, since we have enough regions (64) to be processed with 8 CPUs. Still, there is no speed-up observed in the parallel part of the first sweep (where most of the work is done) when scaling from 4 to 8 CPUs.

It is most probable that reducing memory requirements (e.g. by having dedicated graph implementation for regular grids) would also lead to a speed-up of the parallel solver. We also observed that 32bit compilation (pointers take 32bits) runs faster than 64bit compilation. It is likely that our implementation can be optimized further for the best parallel performance. We should however consider that preparing the data for the problem and splitting it into regions is another time-consuming part, which needs to be parallelized.

Table 3: Parallel Competition

problem	BK	DDx2		DDx4		P-ARD		P-PRD		RPR	
	time	time, sweeps									
stereo											
BVZ-sawtooth(20)	0.68s	0.52s	7	0.37s	11	0.30s	7	2.4s	31	4.8s	274
BVZ-tsukuba(16)	0.36s	0.28s	6	0.20s	8	0.17s	5	1.5s	33	2.1s	197
BVZ-venus(22)	1.2s	0.84s	7	0.59s	9	0.50s	7	4.9s	36	8.0s	466
KZ2-sawtooth(20)	1.8s	1.2s	11	0.91s	16	0.96s	6	4.9s	23		
KZ2-tsukuba(16)	1.1s	0.67s	7	0.52s	11	0.70s	8	4.9s	22		
KZ2-venus(22)	2.8s	1.9s	7	1.3s	12	1.5s	10	10s	39		
multiview											
BL06-camel-med	25s	18s	221	13s	260	8.7s	14	81s	322		
BL06-camel-sml	0.98s	0.63s	11	0.49s	27	0.49s	10	2.5s	70		
BL06-gargoyle-lrg	245s	120s	517	mem		58s	23	mem			
BL06-gargoyle-med	115s	59s	20	38s	50	27s	21	79s	219		
BL06-gargoyle-sml	6.1s	3.0s	19	1.9s	19	1.6s	10	2.4s	52		
surface											
LB07-bunny-med	1.6s	1.3s	11	1.1s	11	1.3s	13	12s	35	37s	349
LB07-bunny-sml	0.17s	0.12s	11	0.12s	11	0.21s	8	0.58s	21	3.5s	99
segm											
liver.n6c10	7.2s	X 7.6s	1000	X 22s	1000	8.9s	23	23s	164	5.1s	1298
liver.n6c100	15s	17s	31	X 21s	1000	12s	17	23s	102	7.3s	1722
babyface.n6c10	6.8s	8.8s	61	X 24s	1000	12s	22	61s	135	17s	4399
babyface.n6c100	13s	16s	338	X 20s	1000	17s	23	61s	179	22s	4833
bone.n6c10	7.7s	5.2s	22	X 8.2s	1000	4.9s	17	16s	182	6.3s	918
bone.n6c100	9.1s	5.3s	12	4.1s	17	6.2s	13	14s	70	7.9s	1070
bone_subx.n6c100	7.1s	6.3s	24	5.2s	34	3.9s	17	5.8s	48	1.5s	747
bone_subxy.n26c100	5.9s	3.4s	11	3.2s	12	5.8s	16	6.0s	37	hang	



problem	regions	MEM	1CPU	8CPUs
BL06-gargoyle-lrg	16	1.6GB	137.4s	38.82s
BL06-camel-lrg	16	1.8GB	67.14s	16.00s
LB07-bunny-lrg	64	12GB	12.94s	7.11s
liver.n6c100	64	0.9GB	18.41s	12.21s

Fig. 14 Speedup of P-ARD with the number of CPUs used. The extended legend shows the time to solve each problem with 1 and 8CPUs (does not include initialization). Dashed lines correspond to the speedup in the ideal case (Amdahl's law) when the parallel portion of the computation is 90% and 95%.

8 Region Reduction

In this section, we attempt to reduce the region network as much as possible by identifying and eliminating vertices which can be decided optimally regardless of the reminder of the full network outside of the region. If it was possible to decide about many vertices globally optimally inside a region network, the whole problem would simplify a lot. It would require less memory and could be potentially solved without distributing and or partitioned again into larger regions. We propose an improved algorithm for such a reduction and its experimental verification. This preprocessing is studied separately and was not applied in the tests of distributed algorithms above. Experiments with vision problems (Table 4) showed that while 2D problems can be significantly reduced, many of the higher-dimension problems do not allow a substantial reduction.

Some vertices become disconnected from the sink in the course of the studied algorithms (S/P-ARD, S/P-PRD). If they are still reachable from the source, they must belong to the source set of any optimal cut. Such vertices do not participate in further computations and the problem can be reduced by excluding them. Unfortunately, the opposite case, when a vertex must be strictly in the sink set is not discovered until the very end of the algorithms.

The following algorithm attempts to identify as many vertices as possible for a given region. It is based on the following simple consideration: if a vertex is disconnected from the sink in G^R as well as from the region boundary, B^R, then it is disconnected from the sink in G; if a vertex is not reachable from the source in G^R as well as from B^R then it is not reachable from the source in G.

Let us say that a vertex v is a *strong source vertex* (resp. a *strong sink vertex*) if for any optimal cut $(C, \bar{C})$, $v \in C$ (resp. $v \in \bar{C}$). Similarly, v will be called a *weak source vertex* (resp. *weak sink vertex*), if there exists an optimal cut $(C, \bar{C})$ such that $v \in C$ (resp. $v \in \bar{C}$).

Kovtun (2004) suggested to solve two auxiliary problems, modifying network G^R by adding infinite capacity links from the boundary vertices to the sink and in the second problem adding infinite capacity links from the source to the boundary vertices. In the first case, if v is a strong source vertex in the modified network G^R, it is also a strong source vertex in G. Similarly, the second auxiliary problem allows to identify strong sink vertices in G. It requires solving a maxflow problem on G^R twice. We improve this construction by reformulating it as the following algorithm finding a single flow in G^R.

Algorithm 3: Region Reduction (G^R, B^R)

```
   /* Input: network G^R, boundary B^R                    */
1  Augment(s, t);
2  B^S := {v | v ∈ B^R, s → v};   /* source boundary set */
3  B^T := {v | v ∈ B^R, v → t};    /* sink boundary set */
4  Augment(s, B^S);
5  Augment(B^T, t);
6  foreach v ∈ R do
7      if s → v then v is strong source vertex;
8      if v → t then v is strong sink vertex;
9      otherwise
10         if v ↛ B^R then v is weak source vertex;
11         if B ↛ v^R then v is weak sink vertex
```

Statement 12 Sets B^S and B^T constructed in step 2 are disjoint.

Proof We have $s \nrightarrow t$ after step 1, hence there cannot exist simultaneously a path from s to v and a path from v to t. □

After step 1, the network G^R is split into two disconnected networks: with vertices reachable from s and vertices from which t is reachable. Therefore, any augmentations occurring in steps 4 and 5 act on their respective subnetworks and can be carried independently of each other. On the output of Alg. 3, we have: $s \nrightarrow B^R \cup \{t\}$ and $B^R \cup \{s\} \nrightarrow t$. The classification of vertices is shown in Fig. 15.

Augmenting on (s, t) in step 1 and on (s, B^S) in the step 4 is the same work as done in ARD (where (s, B^S) paths are augmented in the order of labels of B^S). This is not a coincidence, these algorithms are very much related. However, the augmentation on (B^T, t) in step 5 cannot be executed during ARD. It would destroy validity of the labeling. We therefore consider Alg. 3 as a separate general preprocessing.

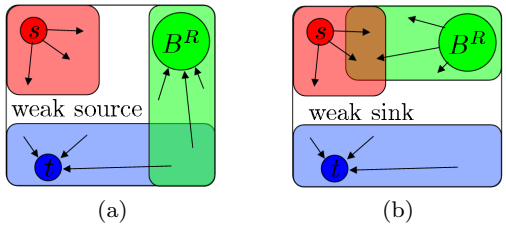


Fig. 15 Classification of vertices in V^R build by Alg. 3. Vertices reachable from s are strong source vertices. Vertices from which t is reachable are strong sink vertices. The remaining vertices can be classified as weak source (a) if they cannot reach boundary, or as weak sink (b) if they are not reachable from the boundary. Some vertices are both: weak source and weak sink, this means they can be on both sides of an optimal cut (but not independently).

If v is a weak source vertex, it follows that it is not a strong sink vertex. In the preflow pushing algorithms, we find the cut $(\bar{T}, T)$, where T is the set of all strong sink vertices in G. We consider that v is *decided* if it is a strong sink or a weak source vertex.

Table 4 gives the percentage of how many vertices are decided (and hence can be excluded from the problem) by Alg. 3 for computer vision problems. It is seen that in stereo problems, a large percent of vertices is decided. These problems are rather local and potentially can be fully solved by applying Alg. 3 on several overlapping windows. In contrast, only a small fraction can be decided locally for many other problems.

9 Tightness of $O(n^2)$ bound for PRD.

In this section, we give an example of a network, its partition into regions and a sequence of valid push and relabel operations, implementing PRD, such that S/P-PRD runs in $\Omega(n^2)$ sweeps.

We start by an auxiliary example, in which the preflow is transfered from a vertex to a boundary vertex with a higher label. In this example, some inner vertices of a region are relabeled, but not any of the boundary vertices. It will imply that the total number of sweeps cannot be bounded by the number of relabellings of boundary vertices alone.

Example 1 Consider a network of 6 regular vertices in Fig. 16. Assume all edges have infinite capacity, so only non-saturating pushes occur. There are two regions $R_1 = \{1, 2, 3, 4, 5\}$ and $R_2 = \{6\}$. Fig. 16 shows a sequence of valid push and relabel operations. We see that some vertices get risen due to relabel, but the net effect is that flow excess from vertex 1 is transfered to vertex 6, which had a higher label initially. Moreover, none of the boundary vertices (vertices 5,6) are relabeled.

BVZ-sawtooth(20)	80.0%	LB07-bunny-sml	15.6%	bone.n26c100	6.9%	bone_subxyz.n6c100	6.6%
BVZ-tsukuba(16)	72.8%	liver.n26c10	7.1%	bone.n6c10	8.8%	bone_subxyz_subx.n26c10	7.9%
BVZ-venus(22)	70.2%	liver.n26c100	5.3%	bone.n6c100	7.0%	bone_subxyz_subx.n26c100	6.6%
KZ2-sawtooth(20)	85.0%	liver.n6c10	7.2%	bone_subx.n26c10	6.6%	bone_subxyz_subx.n6c10	8.2%
KZ2-tsukuba(16)	69.9%	liver.n6c100	5.3%	bone_subx.n26c100	6.6%	bone_subxyz_subx.n6c100	6.6%
KZ2-venus(22)	75.8%	babyface.n26c10	29.3%	bone_subx.n6c10	6.3%	bone_subxyz_subxy.n26c10	11.3%
BL06-camel-lrg	2.0%	babyface.n26c100	30.9%	bone_subx.n6c100	6.3%	bone_subxyz_subxy.n26c100	9.5%
BL06-camel-med	2.3%	babyface.n6c10	35.4%	bone_subxy.n26c10	6.6%	bone_subxyz_subxy.n6c10	12.7%
BL06-camel-sml	4.6%	babyface.n6c100	33.7%	bone_subxy.n26c100	6.6%	bone_subxyz_subxy.n6c100	9.3%
BL06-gargoyle-lrg	6.0%	adhead.n26c10	0.3%	bone_subxy.n6c10	6.4%	abdomen_long.n6c10	1.7%
BL06-gargoyle-med	2.4%	adhead.n26c100	0.3%	bone_subxy.n6c100	6.3%	abdomen_short.n6c10	6.3%
BL06-gargoyle-sml	9.8%	adhead.n6c10	0.2%	bone_subxyz.n26c10	6.6%		
LB07-bunny-lrg	11.4%	adhead.n6c100	0.1%	bone_subxyz.n26c100	6.6%		
LB07-bunny-med	13.1%	bone.n26c10	8.7%	bone_subxyz.n6c10	6.6%		

Table 4 Percentage of vertices which can be decided by preprocessing. The problems are partitioned into regions the same way as in Table 1. For stereo problems the average number over subproblems is shown.

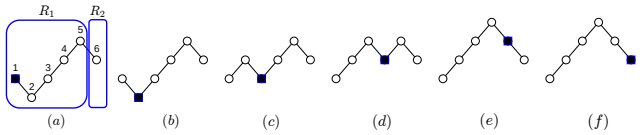


Fig. 16 Steps of Example 1. The height of a vertex correspond to its label. The black box shows the vertex with excess in each step. The source and the think vertices are not shown. (a)-(b) flow excess is pushed to vertex 2; (c) vertex 2 is relabeled, so that two pushes are available and excess is pushed to vertex 3; (d-f) similar.

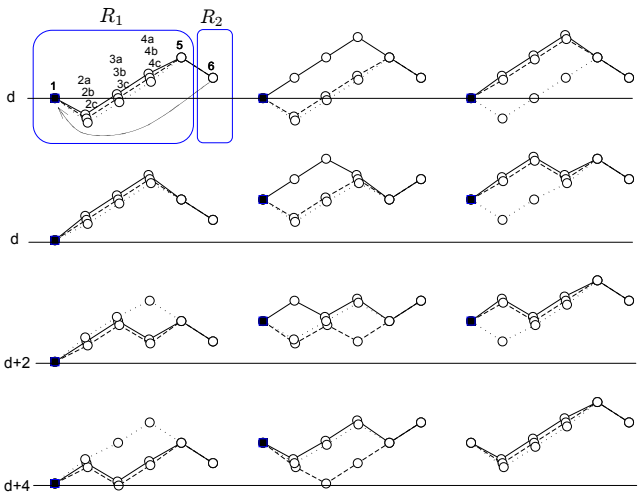


Fig. 17 Steps of Example 2. Top left: a network with several chains of vertices like in Example 1. Nodes 1,5,6 are common for all chains but there are separate copies of vertices 2,3,4 denoted by letters. In addition, there is a reverse arc from vertex 6 to vertex 1. From left to right, top to bottom: one step of transferring a flow from vertex 1 to vertex 6 using one of the chains and then pushing it through the arc (6,1), relabeling 6 when necessary. The label of the first vertex is increased three times by 2.

Example 2 Consider the network in Fig. 17. The first step corresponds to a sequence of push and relabel operations (same as in Fig. 16) applied to the chain $(1, 2a, 3a, 4a, 5, 6)$. Each next step starts with the excess at vertex 1. Chains are selected in turn in the order a, b, c. It can be verified from the figure that each step is a valid possible outcome of PRD applied first to R_1 and then to R_2. The last configuration repeats the first one with all labels raised by 6, so exactly the same loop may be repeated many times.

It is seen that vertices $1, 5, 6$ are relabeled only during pushes in the chains a and b and never during pushes in chain c. If there were more chains like chain c, it would take many iterations (= number of region discharge operations) before boundary vertices are risen. Let there be k additional chains in the graph (denoted d, e,...) handled exactly the same way as chain c. The total number of vertices in the graph is $n = 3k + \text{const}$. Therefore, it will take $\Omega(n)$ region discharges to complete each loop, raising all vertices by a constant value. The number of discharges needed in order that vertex 1 reaches a label D, is $\Omega(nD)$. To make the example complete, we add a chain of vertices initially having labels 1,2,3,... D to the graph such that there is a path from vertex 1 to the sink through a vertex with label D. Clearly, we can arrange that $D = \Omega(n)$. The algorithm needs $\Omega(n^2)$ discharges on this example.

Because there is only one active vertex at any time, the example is independent of the rule used to select the active vertex (highest label, FIFO, *etc.*). By the same reason, it also applies to parallel PRD. Because the number of regions is constant, the number of sweeps required is also $\Omega(n^2)$.

For comparison, noting that the number of boundary vertices is 3, we see that S-ARD algorithm will terminate on this example in a constant number of sweeps for arbitrary k.

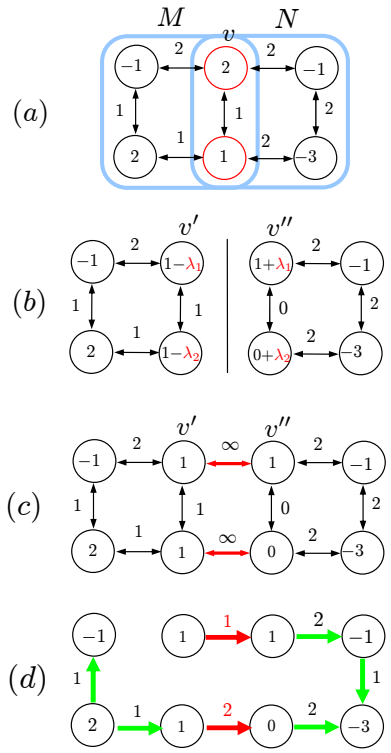


Fig. 18 Interpretation of the dual decomposition. (a) Example of a network with denoted capacities. Terminal capacities are shown in circles, where "+" denotes s-link and "−" denotes t-link. $M \cap N$ is a separator set. (b) The network is decomposed into two networks holding copies of the separator set. The associated capacities are divided (not necessarily evenly) between two copies. The variable λ_1 is the Lagrangian multiplier of the constraint $x_v = y_v$. (c) Introducing edges of infinite capacity enforces the same constraint, that v' and v'' are necessarily in the same cut set of any optimal cut. (d) A maximum flow in the network (c), the flow value on the red edges corresponds to the optimal value of the dual variables λ.

10 Relation to Dual Decomposition

In our approach, we partition the set of vertices into regions and couple the regions by sending the flow through the inter-region edges. In the dual decomposition for MINCUT (Strandmark and Kahl 2010) detailed below, a separator set of the graph is selected, each subproblem gets a copy of the separator set and the coupling is achieved via the constraint that the cut of the separator set must be consistent across the copies. We now show how the dual variables of Strandmark and Kahl (2010) can be interpreted as flow, thus relating their approach to ours.

Decomposition of the MINCUT problem into two parts is formulated by Strandmark and Kahl (2010) as follows. Let $M, N \subset V$ are such that $M \cup N = V$, $\{s,t\} \subset M \cap N$ and there are no edges in E from $M \backslash N$ to $N \backslash M$ and vice-versa. Let $x\colon M \to \{0,1\}$ and $y\colon N \to \{0,1\}$ be the indicator variables of the cut set, where 0 corresponds to the source set. Then the MINCUT problem without excess can be reformulated as:

$$\min_{x,y} C^M(x) + C^N(y),$$
$$\text{s.t.} \begin{cases} x_s = y_s = 0, \\ x_t = y_t = 1, \\ x_i = y_i, \quad \forall i \in M \cap N, \end{cases} \tag{15}$$

where

$$C^M(x) = \sum_{(i,j)\in E^M} c^M(i,j)(1-x_i)x_j, \tag{16a}$$
$$C^N(y) = \sum_{(i,j)\in E^N} c^N(i,j)(1-y_i)y_j; \tag{16b}$$

$$c^M(i,j) + c^N(i,j) = c(i,j), \tag{17a}$$
$$c^M(i,j) = 0 \quad \forall i,j \in N \backslash M, \tag{17b}$$
$$c^N(i,j) = 0 \quad \forall i,j \in M \backslash N, \tag{17c}$$

$E^M = (M,M) \stackrel{\text{def}}{=} (M \times M) \cap E$ and $E^N = (N,N)$. The minimization over x and y decouples once the constraint $x_i = y_i$ is absent. The dual decomposition approach is to solve the dual problem:

$$\max_{\lambda} \Big[\min_{\substack{x \\ x_s=0 \\ x_t=1}} \Big(C_M(x) + \sum_{i \in M \cap N} \lambda_i (1-x_s) x_i \Big) + \min_{\substack{y \\ y_s=0 \\ y_t=1}} \Big(C_N(y) - \sum_{i \in M \cap N} \lambda_i (1-y_s) y_i \Big) \Big], \tag{18}$$

where the dual variable λ is multiplied by the extra terms $(1-x_s) = (1-y_s) = 1$ to show explicitly that the inner minimization problems are instances of the minimum cut problem.

We observe that dual variables λ correspond to the flow on the artificial edges of infinite capacity between the copies of the vertices of the separator set as illustrated by Fig. 18. Indeed, consider a vertex v in the separator set. The dual variable λ_v contributes to the increase of the terminal link (v', t) in the subproblem M and to the decrease of the terminal link (v'', t) in the subproblem N. This can be equivalently represented as an augmentation of flow of λ_v on the cycle v', t', v'' in the network Fig. 18(c). The optimal flow in the network Fig. 18(c) on the constraint edges will therefore correspond to the optimal λ. This construction could be easily extended to the case when a vertex v from the separator set is shared by more than two subproblems.

There exist an integer optimal flow for a problem with integer capacities. This observation provides an alternative proof of the theorem (Strandmark and Kahl

2010, Theorem 2)[8], stating that there exist an integer optimal λ. Despite the existence of an integer solution, the integer subgradient algorithm (Strandmark and Kahl 2010) is not guaranteed to find it.

The algorithm we introduced could be applied to such a decomposition by running it on the extended graph Fig. 18(c), where vertices of the separator set are duplicated and linked by additional edges of infinite capacity. It could be observed, however, that this construction does not allow to reduce the number of boundary vertices or the number of inter-region edges, while the size of the regions increases. Therefore it is not beneficial with our approach.

11 Conclusion

We developed a new algorithm for MINCUT problem on sparse graphs, which combines augmenting paths and push-relabel approaches. We proved the worst case complexity guarantee of $O(|\mathcal{B}|^2)$ sweeps for the sequential and parallel variants of the algorithm (S/P-ARD). While there are many algorithms in the literature with complexities in terms of elementary arithmetic operations better than we could possibly prove, we showed that our algorithms are fast and competitive in practice, even in the shared memory model. We proposed an improved algorithm for local problem reduction (§8) and determined that most of our test instances are difficult enough in the sense that very few vertices can be decided optimally by looking at individual regions. The result that S/P-ARD solves test problem in few tens of sweeps is thus non-trivial. We also gave a novel parallel version of the region push-relabel algorithm of Delong and Boykov (2008) and a number of auxiliary results, relating our approach to the state-of-the.

Both in theory and practice (randomized test), S-ARD has a better asymptote in the number of sweeps than the push-relabel variant. Experiments on real instances showed that when run on a single CPU and the whole problem fits into the memory, S-ARD is comparable in speed with the non-distributed BK implementation, and is even significantly faster in some cases. When only a single region is loaded into memory at a time, S-ARD uses much fewer disk I/O than S-PRD. We also demonstrated that the running time and the number of sweeps are very stable with respect to the partition of the problem into up to 64 regions. In the parallel mode, using 4 CPUs, P-ARD achieves a relative speedup of about $1.5 - 2.5$ times over S-ARD and uses just slightly larger number of sweeps. P-ARD compares favorably to other parallel algorithms, being a robust method suitable for a use in a distributed system.

Our algorithms are implemented for generic graphs. Clearly, it is possible to specialize the implementation for grid graphs, which would reduce the memory consumption and might reduce the computation time as well.

A practically useful mode could be actually a combination of a parallel and sequential processing, when several regions are loaded into the memory at once and processed in parallel. There are several particularly interesting combinations of algorithm parallelization and hardware, which may be exploited: 1) parallel on several CPUs, 2) parallel on several network computers, 3) sequential, using Solid State Drive, 4) sequential, using GPU for solving region discharge.

There is the following simple way how to allow region overlaps in our framework. Consider a sequential algorithm, which is allowed to keep 2 regions in memory at a time. It can then load pairs of regions $(1, 2)$, $(2, 3)$, $(3, 4)\ldots$, and alternate between the regions in a pair until both are discharged. With PRD, this is efficiently equivalent to discharging twice larger regions with a 1/2 overlap and may significantly decrease the number of sweeps required. In the case of a 3D grid, it would take 8 times more regions to allow overlaps in all dimensions. However, to meet the same memory limit, the regions have to be 8 times smaller. It has to be verified experimentally whether it is beneficial. In fact, the RPR implementation of Delong and Boykov (2008) uses exactly this strategy: a dynamic region is composed out of a number of smaller blocks and blocks are discharged until the whole region is not discharged. It is likely that with this approach we could further reduce the disk I/O in the case of the streaming solver.

Future Work. We plan to provide also a distributed MPI-based implementation as well as address the question of dynamic updates of the problem and implement a more efficient input-output interface for the use in real applications. The memory-efficient representation of graphs having repetitive structure is also possible.

Acknowledgements We would like to thank the reviewers for their help in verifying the correctness and great advice on improving the presentation of this work.

[8] Strandmark and Kahl (2010) stated their theorem for even integer costs in the case of two-subproblem separator sets. They remarked that a multiple of 4, resp., 8 is needed in the cases of decompositions for 2D and 3D grids. However, this multiplication is unnecessary if we chose to split the cost unevenly but preserving the integrality (like we did in the example).

References

Anderson R, Setubal JC (1995) A parallel implementation of the push-relabel algorithm for the maximum flow problem. J Parallel Distrib Comput 29(1) 2

Boros E, Hammer PL, Sun X (1991) Network flows and minimization of quadratic pseudo-Boolean functions. Tech. Rep. RRR 17-1991, RUTCOR 1

Boykov Y, Funka-Lea G (2006) Graph cuts and efficient N-D image segmentation. Int J Comput Vision 70 2, 19

Boykov Y, Jolly MP (2001) Interactive graph cuts for optimal boundary & region segmentation of objects in N-D images. In: ICCV 2, 19

Boykov Y, Kolmogorov V (2003) Computing geodesics and minimal surfaces via graph cuts. In: ICCV 2, 19

Boykov Y, Kolmogorov V (2004) An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. In: PAMI, vol 26 2, 13

Boykov Y, Lempitsky V (2006) From photohulls to photoflux optimization. In: BMVC 2, 19

Boykov Y, Veksler O, Zabih R (1998) Markov random fields with efficient approximations. In: CVPR 2, 19

Boykov Y, Veksler O, Zabih R (1999) Fast approximate energy minimization via graph cuts. In: ICCV 1

Cherkassky BV, Goldberg AV (1994) On implementing push-relabel method for the maximum flow problem. Tech. rep. 2, 12, 13

Delong A, Boykov Y (2008) A scalable graph-cut algorithm for N-D grids. In: CVPR 1, 2, 3, 5, 6, 13, 14, 22, 27

Goldberg A (1987) Efficient graph algorithms for sequential and parallel computers. PhD thesis, Massachusetts Institute of Technology 6, 7

Goldberg AV (1991) Processor-efficient implementation of a maximum flow algorithm. Inf Process Lett 38(4) 2, 3

Goldberg AV (2008) The partial augment–relabel algorithm for the maximum flow problem. In: Proceedings of the 16th annual European symposium on Algorithms 3

Goldberg AV, Rao S (1998) Beyond the flow decomposition barrier. J ACM 3

Goldberg AV, Tarjan RE (1988) A new approach to the maximum flow problem. Journal of the ACM 35 2, 3, 5, 6, 7

Ishikawa H (2003) Exact optimization for Markov random fields with convex priors. PAMI 25(10) 1

Jancosek M, Pajdla T (2011) Robust, accurate and weakly-supported-surfaces preserving multi-vew reconstruction. In: CVPR 2

Kohli P, Torr P (2005) Efficiently solving dynamic markov random fields using graph cuts. In: ICCV05, vol 2, pp 922–929 13

Kohli P, Shekhovtsov A, Rother C, Kolmogorov V, Torr P (2008) On partial optimality in multi-label MRFs. In: ICML 1

Kolmogorov V (2004) Graph based algorithms for scene reconstruction from two or more views. PhD thesis, Ithaca, NY, USA, aAI3114475 16

Kolmogorov V, Rother C (2007) Minimizing non-submodular functions with graph cuts – a review. PAMI 1

Kolmogorov V, Zabih R (2001) Computing visual correspondence with occlusions via graph cuts. In: ICCV 2, 19

Kovtun I (2004) Image segmentation based on sufficient conditions of optimality in np-complete classes of structural labelling problem. PhD thesis, in Ukrainian 24

Labatut P, Pons JP, Keriven R (2009) Robust and efficient surface reconstruction from range data. Computer Graphics Forum 2

Lempitsky V, Boykov Y (2007) Global optimization for shape fitting. In: CVPR 2, 19

Lempitsky V, Boykov Y, Ivanov D, Ivanov D (2006) Oriented visibility for multiview reconstruction. In: ECCV 2, 19

Lempitsky V, Rother C, Roth S, Blake A (2010) Fusion moves for markov random field optimization. PAMI 1

Liu J, Sun J (2010) Parallel graph-cuts by adaptive bottom-up merging. In: CVPR 2, 22

Schlesinger D, Flach B (2006) Transforming an arbitrary min-sum problem into a binary one. Research report, Dresden University of Technology 1

Shekhovtsov A, Hlavac V (2011) A distributed min-cut/maxflow algorithm combining path augmentation and push-relabel. In: Proceedings of the 8th International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition (EMMCVPR), Springer, Lecture Notes in Computer Science, p 14 17

Strandmark P, Kahl F (2010) Parallel and distributed graph cuts by dual decomposition. In: CVPR 2, 22, 26, 27

University of Western Ontario web pages (2008) Computer vision research group. max-flow problem instances in vision. `http://vision.csd.uwo.ca/maxflow-data/` 2, 19