# Fast, Approximately Optimal Solutions for Single and Dynamic MRFs*

Nikos Komodakis, Georgios Tziritas
University of Crete, Computer Science Department
{komod,tziritas}@csd.uoc.gr

Nikos Paragios
MAS, Ecole Centrale de Paris
nikos.paragios@ecp.fr

## Abstract

*A new efficient MRF optimization algorithm, called Fast-PD, is proposed, which generalizes $\alpha$-expansion. One of its main advantages is that it offers a substantial speedup over that method, e.g. it can be at least 3-9 times faster than $\alpha$-expansion. Its efficiency is a result of the fact that Fast-PD exploits information coming not only from the original MRF problem, but also from a dual problem. Furthermore, besides static MRFs, it can also be used for boosting the performance of dynamic MRFs, i.e. MRFs varying over time. On top of that, Fast-PD makes no compromise about the optimality of its solutions: it can compute exactly the same answer as $\alpha$-expansion, but, unlike that method, it can also guarantee an almost optimal solution for a much wider class of NP-hard MRF problems. Results on static and dynamic MRFs demonstrate the algorithm's efficiency and power. E.g., Fast-PD has been able to compute disparity for stereoscopic sequences in real time, with the resulting disparity coinciding with that of $\alpha$-expansion.*

## 1. Introduction

Discrete MRFs are ubiquitous in computer vision, and thus optimizing them is a problem of fundamental importance. According to it, given a weighted graph $\mathcal{G}$ (with nodes $\mathcal{V}$, edges $\mathcal{E}$ and weights $w_{pq}$), one seeks to assign a label $x_p$ (from a discrete set of labels $\mathcal{L}$) to each $p \in \mathcal{V}$, so that the following cost is minimized:

$$\sum\nolimits_{p\in\mathcal{V}} \mathfrak{c}_p(x_p) + \sum\nolimits_{(p,q)\in\mathcal{E}} w_{pq} d(x_p, x_q). \qquad (1)$$

Here, $\mathfrak{c}_p(\cdot)$, $d(\cdot,\cdot)$ determine the singleton and pairwise MRF potential functions respectively.

Up to now, graph-cut based methods, like $\alpha$-expansion [3], have been very effective in MRF optimization, generating solutions with good optimality properties [8]. However, besides solutions' optimality, another important issue is that of computational efficiency. In fact, this issue has recently been looked at for the special case of dynamic MRFs [5, 4], *i.e*. MRFs varying over time. Thus, trying to concentrate on both of these issues here, we raise the following questions: can there be a graph-cut based method, which will be more efficient, but equally (or even more) powerful, than $\alpha$-expansion, for the case of single MRFs? Furthermore, can that method also offer a computational advantage for the case of dynamic MRFs? With respect to the questions raised above, this work makes the following contributions.

**Efficiency for single MRFs:** $\alpha$-expansion works by solving a series of max-flow problems. Its efficiency is thus largely determined from the efficiency of these max-flow problems, which, in turn, depends on the number of augmenting paths per max-flow. Here, we build upon recent work of [6], and propose a new primal-dual MRF optimization method, called Fast-PD. This method, like [6] or $\alpha$-expansion, also ends up solving a max-flow problem for a series of graphs. However, unlike these techniques, the graphs constructed by Fast-PD ensure that the number of augmentations per max-flow decreases dramatically over time, thus boosting the efficiency of MRF inference. To show this, we prove a generalized relationship between the number of augmentations and the so-called *primal-dual gap* associated with the original MRF problem and its dual. Furthermore, to fully exploit the above property, 2 new extensions are also proposed: an *adapted max-flow algorithm*, as well as an *incremental graph construction* method.

**Optimality properties:** Despite its efficiency, our method also makes no compromise regarding the optimality of its solutions. So, if $d(\cdot,\cdot)$ is a metric, Fast-PD is as powerful as $\alpha$-expansion, *i.e*. it computes exactly the same solution, but with a substantial speedup. Moreover, it applies to a much wider class of MRFs[1], *e.g*. even with a non-metric $d(\cdot,\cdot)$, while still guaranteeing an almost optimal solution.

**Efficiency for dynamic MRFs:** Furthermore, our method can also be used for boosting the efficiency of dynamic MRFs (introduced to computer vision in [5]). Two works have been proposed in this regard recently [5, 4]. These methods can be applied to dynamic MRFs that are binary or have convex priors. On the contrary, Fast-PD naturally handles a much wider class of dynamic MRFs, and can do so by also exploiting information from a problem, which is dual to the original MRF problem. Fast-PD can thus be thought of as a generalization of previous techniques.

The rest of the paper is organized as follows. In sec. 2, we briefly review the work of [6] about using the primal-dual schema for MRF optimization. The Fast-PD algorithm is then described in sec. 3. Its efficiency for optimizing

*This work was partially supported from the French ANR-Blanc grant SURF (2005-2008) and Platon (2006-2007).

[1] Fast-PD requires only $d(a,b) \geq 0,\ d(a,b) = 0 \Leftrightarrow a = b$

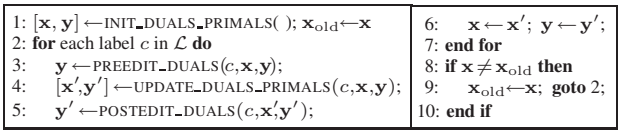



**Fig. 1:** The primal dual schema for MRF optimization.

single MRFs is further analyzed in sec. 4, where related results and some important extensions of Fast-PD are presented as well. Sec. 5 explains how Fast-PD can boost the performance of dynamic MRFs, and also contains more experimental results. Finally, we conclude in section 6.

## 2. Primal-dual MRF optimization algorithms

In this section, we review very briefly the work of [6]. Consider the primal-dual pair of linear programs, given by:

$$\text{PRIMAL: } \min \mathbf{c}^T\mathbf{x} \quad \text{s.t. } \mathbf{A}\mathbf{x}=\mathbf{b}, \mathbf{x}\geq\mathbf{0} \qquad \text{DUAL: } \max \mathbf{b}^T\mathbf{y} \quad \text{s.t. } \mathbf{A}^T\mathbf{y}\leq\mathbf{c}$$

One seeks an optimal primal solution, with the extra constraint of $\mathbf{x}$ being integral. This makes for an NP-hard problem, and so one can only hope for finding an approximate solution. To this end, the following schema can be used:

**Theorem 1 (Primal-Dual schema).** *Keep generating pairs of integral-primal, dual solutions* $(\mathbf{x^k}, \mathbf{y^k})$*, until the elements of the last pair, say* $\mathbf{x}, \mathbf{y}$*, are both feasible and have costs that are close enough, e.g. their ratio is* $\leq f_{\text{app}}$*:*

$$\mathbf{c}^T\mathbf{x} \leq f_{\text{app}} \cdot \mathbf{b}^T\mathbf{y} \tag{2}$$

*Then* $\mathbf{x}$ *is guaranteed to be an* $f_{\text{app}}$*-approximate solution to the optimal integral solution* $x^*$*, i.e.* $\mathbf{c}^T\mathbf{x} \leq f_{\text{app}} \cdot \mathbf{c}^T\mathbf{x}^*$.

The above schema has been used in [6], for deriving approximation algorithms for a very wide class of MRFs. To this end, MRF optimization was first cast as an equivalent integer program and then, as required by the primal-dual schema, its linear programming relaxation and its dual were derived. Based on these LPs, the authors then show that, for Theorem 1 to be true with $f_{\text{app}} = 2\frac{d_{\max}}{d_{\min}}$[2], it suffices that the next (so-called *relaxed complementary slackness*) conditions hold true for the resulting primal and dual variables:

$$h_p(x_p) = \min_{a\in\mathcal{L}} h_p(a), \quad \forall p\in\mathcal{V} \tag{3}$$

$$y_{pq}(x_p)+y_{qp}(x_q) = w_{pq}d(x_p,x_q), \quad \forall pq\in\mathcal{E} \tag{4}$$

$$y_{pq}(a)+y_{qp}(b) \leq 2w_{pq}d_{\max}, \quad \forall pq\in\mathcal{E}, a\in\mathcal{L}, b\in\mathcal{L} \tag{5}$$

In these formulas, the primal variables, denoted by $\mathbf{x} = \{x_p\}_{p\in\mathcal{V}}$, determine the labels assigned to nodes (called *active labels* hereafter), *e.g.* $x_p$ is the active label of node $p$. Whereas, the dual variables are divided into *balance* and *height* variables. There exist 2 balance variables $y_{pq}(a), y_{qp}(a)$ per edge $(p,q)$ and label $a$, as well as 1 height variable $h_p(a)$ per node $p$ and label $a$. Variables $y_{pq}(a), y_{qp}(a)$ are also called *conjugate* and, for the dual solution to be feasible, these must be set opposite to each other, *i.e.*: $y_{qp}(\cdot) \equiv -y_{pq}(\cdot)$. Furthermore, the height variables are always defined in terms of the balance variables as follows:

[2] $d_{\max} \equiv \max_{a\neq b} d(a,b)$, $d_{\min} \equiv \min_{a\neq b} d(a,b)$

$$h_p(\cdot) \equiv \mathfrak{c}_p(\cdot) + \sum\nolimits_{q:qp\in\mathcal{E}} y_{pq}(\cdot). \tag{6}$$

Note that, due to (6), only the vector $\mathbf{y}$ (of all balance variables) is needed for specifying a dual solution. In addition, for simplifying conditions (4),(5), one can also define:

$$\text{load}_{pq}(a,b) \equiv y_{pq}(a)+y_{qp}(b). \tag{7}$$

The primal-dual variables are iteratively updated until all conditions (3)-(5) hold true. The basic structure of a primal-dual algorithm can be seen in Fig. 1. During an inner $c$-iteration (lines 3-6 in Fig. 1), a label $c$ is selected and a new primal-dual pair of solutions $(\mathbf{x}', \mathbf{y}')$ is generated based on the current pair $(\mathbf{x}, \mathbf{y})$. To this end, among all balance variables $y_{pq}(.)$, only the balance variables of $c$-labels (*i.e.* $y_{pq}(c)$) are updated during a $c$-iteration. $|\mathcal{L}|$ such iterations (*i.e.* one $c$-iteration per label $c$ in $\mathcal{L}$) make up an outer iteration (lines 2-7 in Fig. 1), and the algorithm terminates if no change of label takes place at the current outer iteration.

During an inner iteration, the main update of the primal and dual variables takes place inside UPDATE_DUALS_PRIMALS, and (as shown in [6]) this update reduces to solving a max-flow problem in an appropriate graph $\mathcal{G}^c$. Furthermore, the routines PREEDIT_DUALS and POSTEDIT_DUALS simply apply corrections to the dual variables before and after this main update, *i.e.* to variables $\mathbf{y}$ and $\mathbf{y}'$ respectively. Also, for simplicity's sake, note that we will hereafter refer to only one of the methods derived in [6], and this will be the so-called PD3$_a$ method.

## 3. Fast primal-dual MRF optimization

The complexity of the PD3$_a$ primal-dual method largely depends on the complexity of all max-flow instances (one instance per inner-iteration), which, in turn, depends on the number of augmentations per max-flow. So, for designing faster primal-dual algorithms, we first need to understand how the graph $\mathcal{G}^c$, associated with the max-flow problem at a $c$-iteration of PD3$_a$, is constructed. To this end, we also have to recall the following intuitive interpretation of the dual variables [6]: for each node $p$, a separate copy of all labels in $\mathcal{L}$ is considered, and all these labels are represented as balls, which float at certain heights relative to a reference plane. The role of the height variables $h_p(\cdot)$ is then to determine the balls' height (see Figure 2(a)). *E.g.* the height of label $a$ at node $p$ is given by $h_p(a)$. Also, expressions like "label $a$ at $p$ is below/above label $b$" imply $h_p(a) \lesseqgtr h_p(b)$. Furthermore, balls are not static, but may move in pairs through updating pairs of conjugate balance variables. *E.g.*, in Figure 2(a), label $c$ at $p$ is raised by $+\delta$ (due to adding $+\delta$ to $y_{pq}(c)$), and so label $c$ at $q$ has to move down by $-\delta$ (due to adding $-\delta$ to $y_{qp}(c)$ so that condition $y_{pq}(c) = -y_{qp}(c)$ still holds). Therefore, the role of balance variables is to raise or lower labels. In particular, the value of balance variable $y_{pq}(a)$ represents the partial raise of label $a$ at $p$ due to edge $pq$, while (by (6)) the total raise of $a$ at $p$ equals the sum of partial raises from all edges of $\mathcal{G}$ incident to $p$.

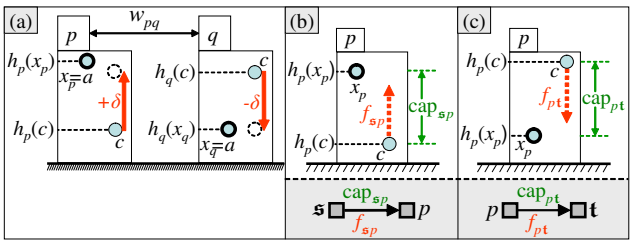



**Fig. 2:** **(a)** Dual variables' visualization for a simple MRF with 2 nodes $\{p, q\}$ and 2 labels $\{a, c\}$. A copy of labels $\{a, c\}$ exists for every node, and all these labels are represented by balls floating at certain heights. The role of the *height variables* $h_p(\cdot)$ is to specify exactly these heights. Furthermore, balls are not static, but may move (*i.e.* change their heights) in pairs by updating conjugate *balance variables*. E.g., here, ball $c$ at $p$ is pulled up by $+\delta$ (due to increasing $y_{pq}(c)$ by $+\delta$) and so ball $c$ at $q$ moves down by $-\delta$ (due to decreasing $y_{qp}(c)$ by $-\delta$). Active labels are drawn with a thicker circle. **(b)** If label $c$ at $p$ is below $x_p$, then (due to (3)) we want label $c$ to raise and reach $x_p$. We thus connect node $p$ to the source $\mathfrak{s}$ with an edge $\mathfrak{s}p$ (*i.e.* $p$ is an $\mathfrak{s}$-linked node), and flow $f_{\mathfrak{s}p}$ represents the total raise of $c$ (we also set $\text{cap}_{\mathfrak{s}p} = h_p(x_p) - h_p(c)$). **(c)** If label $c$ at $p$ is above $x_p$, then (due to (3)) we want label $c$ not to go below $x_p$. We thus connect node $p$ to the sink $\mathfrak{t}$ with edge $p\mathfrak{t}$ (*i.e.* $p$ is a $\mathfrak{t}$-linked node), and flow $f_{p\mathfrak{t}}$ represents the total decrease in the height of $c$ (we also set $\text{cap}_{p\mathfrak{t}} = h_p(c) - h_p(x_p)$ so that $c$ will still remain above $x_p$).

Hence, PD3$_a$ tries to iteratively move labels up or down, until all conditions (3)-(5) hold true. To this end, it uses the following strategy: it ensures that conditions (4)-(5) hold at each iteration (which is always easy to do) and is just left with the main task of making the labels' heights satisfy condition (3) as well in the end (which is the most difficult part, requiring each active label $x_p$ to be the lowest label for $p$). For this purpose, labels are moved in groups. In particular, during a $c$-iteration, only the $c$-labels are allowed to move. Furthermore, it was shown in [6] that the movement of all $c$-labels (*i.e.* the update of dual variables $y_{pq}(c)$ and $h_p(c)$ for all $p, q$) can be simulated by pushing the maximum flow through a directed graph $\mathcal{G}^c$ (which is constructed based on the current primal-dual pair $(\mathbf{x}, \mathbf{y})$ at a $c$-iteration). The nodes of $\mathcal{G}^c$ consist of all nodes of graph $\mathcal{G}$ (the *internal* nodes), plus 2 *external* nodes, the source $\mathfrak{s}$ and the sink $\mathfrak{t}$. In addition, all nodes of $\mathcal{G}^c$ are connected by two types of edges: *interior* and *exterior* edges. Interior edges come in pairs $pq$, $qp$ (with one such pair for every 2 neighbors $p, q$ in $\mathcal{G}$), and are responsible for updating the balance variables. In particular, the flows $f_{pq}/f_{qp}$ of these edges represent the increase/decrease of balance variable $y_{pq}(c)$, *i.e.* $y'_{pq}(c) = y_{pq}(c) + f_{pq} - f_{qp}$. Also, as we shall see, the capacities of interior edges are used together with PREEDIT_DUALS, POSTEDIT_DUALS to impose conditions (4), (5).

But for now, in order to understand how to make a faster primal-dual method, it is the exterior edges (which are in charge of the update of height variables), as well as their capacities (which are used for imposing the remaining condition (3)), that are of interest to us. The reason is that these edges determine the number of $\mathfrak{s}$-linked nodes, which, in turn, affects the number of augmenting paths per max-flow. In particular, each internal node connects to either the source $\mathfrak{s}$ (*i.e.* it is an $\mathfrak{s}$-*linked* node) or to the sink $\mathfrak{t}$ (*i.e.* it is a $\mathfrak{t}$-*linked* node) through one of these exterior edges, and this is done (with the goal of ensuring (3)) as follows: if label $c$ at $p$ is above $x_p$ during a $c$-iteration (*i.e.* $h_p(c) > h_p(x_p)$), then label $c$ should not go below $x_p$, or else (3) will be violated for $p$. Node $p$ thus connects to $\mathfrak{t}$ through directed edge $p\mathfrak{t}$ (*i.e.* $p$ becomes $\mathfrak{t}$-linked), and flow $f_{p\mathfrak{t}}$ represents the total decrease in the height of $c$ after UPDATE_DUALS_PRIMALS, *i.e.* $h'_p(c) = h_p(c) - f_{p\mathfrak{t}}$ (see Fig. 2(c)). Furthermore, the capacity of $p\mathfrak{t}$ is set so that label $c$ will still remain above $x_p$, *i.e.* $\text{cap}_{p\mathfrak{t}} = h_p(c) - h_p(x_p)$. On the other hand, if label $c$ at $p$ is below active label $x_p$ (*i.e.* $h_p(c) < h_p(x_p)$), then (due to (3)) label $c$ should raise so as to reach $x_p$, and so $p$ connects to $\mathfrak{s}$ through edge $\mathfrak{s}p$ (*i.e.* $p$ becomes $\mathfrak{s}$-linked), while flow $f_{\mathfrak{s}p}$ represents the total raise of ball $c$, *i.e.* $h'_p(c) = h_p(c) + f_{\mathfrak{s}p}$ (see Fig. 2(b)). In this case, we also set $\text{cap}_{\mathfrak{s}p} = h_p(x_p) - h_p(c)$.

This way, by pushing flow through the exterior edges of $\mathcal{G}^c$, all $c$-labels that are strictly below an active label try to raise and reach that label during UPDATE_DUALS_PRIMALS[3]. Not only that, but the fewer are the $c$-labels below an active label (*i.e.* the fewer are the $\mathfrak{s}$-linked nodes), the fewer will be the edges connected to the source, and thus the less will be the number of possible augmenting paths. In fact, the algorithm terminates when, for any label $c$, there are no more $c$-labels strictly below an active label (*i.e.* no $\mathfrak{s}$-linked nodes exist and thus no augmenting paths may be found), in which case condition (3) will finally hold true, as desired. Put another way, UPDATE_DUALS_PRIMALS tries to push $c$-labels (which are at a low height) up, so that the number of $\mathfrak{s}$-linked nodes is reduced and thus fewer augmenting paths may be possible for the next iteration.

However, although UPDATE_DUALS_PRIMALS tries to reduce the number of $\mathfrak{s}$-linked nodes (by pushing the maximum amount of flow), PREEDIT_DUALS or POSTEDIT_DUALS very often spoil that progress. As we shall see later, this occurs because, in order to restore condition (4) (which is their main goal), these routines are forced to apply corrections to the dual variables (*i.e.* to the labels' height). This is abstractly illustrated in Figure 3, where, as a result of pushing flow, a $c$-label initially managed to reach an active label $x_p$, but it again dropped below $x_p$, due to some correction applied by these routines. In fact, as one can show, the only point where a new $\mathfrak{s}$-linked node can be created is during either PREEDIT_DUALS or POSTEDIT_DUALS.

[3] Equivalently, if $c$-label at $p$ cannot raise high enough to reach $x_p$, UPDATE_DUALS_PRIMALS then assigns that $c$-label as the new active label of $p$ (*i.e.* $x'_p = c$), thus effectively making the active label go down. This helps condition (3) to become true, and forms the main rationale behind the update of the primal variables $\mathbf{x}$ in UPDATE_DUALS_PRIMALS.

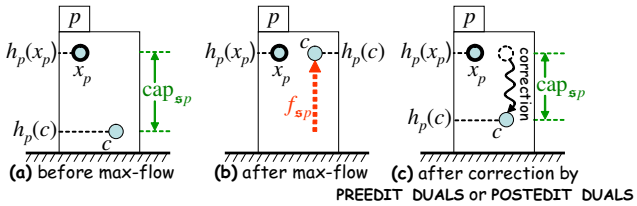



**Fig. 3:** **(a)** Label $c$ at $p$ is below $x_p$, and thus label $c$ is allowed to raise itself in order to reach $x_p$. This means that $p$ will be an $\mathfrak{s}$-linked node of graph $\mathcal{G}^c$, *i.e.* $\text{cap}_{\mathfrak{s}p} > 0$, and thus a non-zero flow $f_{\mathfrak{s}p}$ (representing the total raise of label $c$) may pass through edge $\mathfrak{s}p$. Therefore, in this case, edge $\mathfrak{s}p$ may become part of an augmenting path during max-flow. **(b)** After UPDATE_DUALS_PRIMALS, label $c$ has managed to raise by $f_{\mathfrak{s}p}$ and reach $x_p$. Since it cannot go higher than that, no flow can pass through edge $\mathfrak{s}p$, *i.e.* $\text{cap}_{\mathfrak{s}p} = 0$, and so no augmenting path may traverse that edge thereafter. **(c)** However, due to some correction applied to $c$-label's height, label $c$ has dropped below $x_p$ once more and $p$ has become an $\mathfrak{s}$-linked node again (*i.e.* $\text{cap}_{\mathfrak{s}p} > 0$). Edge $\mathfrak{s}p$ can thus be part of an augmenting path again (as in (a)).

To fix this problem, we will redefine PREEDIT_DUALS, POSTEDIT_DUALS so that they can now ensure condition (4) by using just a minimum amount of corrections for the dual variables, (*e.g.* by touching these variables only rarely). To this end, however, UPDATE_DUALS_PRIMALS needs to be modified as well. The resulting algorithm, called Fast-PD, carries the following main differences over PD3$_a$ during a $c$-iteration (its pseudocode appears in Fig. 4):

- the new PREEDIT_DUALS modifies a pair $y_{pq}(c), y_{qp}(c)$ of dual variables only when absolutely necessary. So, whereas the previous version modified these variables (thereby changing the height of a $c$-label) whenever $c{\neq}x_p$, $c{\neq}x_q$ (which could happen extremely often), a modification is now applied only if $\text{load}_{pq}(c,x_q) > w_{pq}d(c,x_q)$ or $\text{load}_{pq}(x_p,c) > w_{pq}d(x_p,c)$ (which, in practice, happens much more rarely). In this case, a modification is needed (see code in Fig. 4), because the above inequalities indicate that condition (4) will be violated if either $(c,x_q)$ or $(x_p,c)$ become the new active labels for $p,q$. On the contrary, no modification is needed if the following inequalities are true: $\text{load}_{pq}(c,x_q) < w_{pq}d(c,x_q)$, $\text{load}_{pq}(x_p,c) < w_{pq}d(x_p,c)$, because then, as we shall see below, the new UPDATE_DUALS_PRIMALS can always restore (4) (*i.e.* even if $(c,x_q)$ or $(x_p,c)$ are the next active labels - *e.g.*, see (12)). In fact, the modification to $y_{pq}(c)$ that is occasionally applied by the new PREEDIT_DUALS can be shown to be the minimal correction that restores exactly the above inequalities (assuming, of course, this restoration is possible).

- Similarly, the new POSTEDIT_DUALS modifies[4] balance variables $y'_{pq}(x'_p)$ (with $x'_p = c$) and $y'_{qp}(x'_q)$ (with $x'_q = c$) only if the inequality $\text{load}'_{pq}(x'_p,x'_q) > w_{pq}d(x'_p,x'_q)$ holds, in which case POSTEDIT_DUALS simply has to

[4]We recall that POSTEDIT_DUALS may modify only dual solution $\mathbf{y}'$. For that solution, we define $\text{load}'_{pq}(a,b) \equiv y'_{pq}(a) + y'_{qp}(b)$, as in (7).

```
[x, y]←INIT_DUALS_PRIMALS( ):  x←random labels; y←0;
 ∀pq, adjust y_pq(x_p) or y_qp(x_q) so that load_pq(x_p,x_q)=w_pq d(x_p,x_q)

y←PREEDIT_DUALS(c, x, y):
 ∀pq, if load_pq(c,x_q)>w_pq d(c,x_q) or load_pq(x_p,c)>w_pq d(x_p,c)
        adjust y_pq(c) so that load_pq(c,x_q)=w_pq d(c,x_q)

[x', y']←UPDATE_DUALS_PRIMALS(c, x, y):  x'←x; y'←y;
 Construct G^c and apply max-flow to compute all flows f_sp/f_pt, f_pq
 ∀pq, y'_pq(c)←y_pq(c)+f_pq−f_qp
 ∀p , if an unsaturated path from s to p exists, then x'_p←c

y'←POSTEDIT_DUALS(c, x', y'): {We denote load'_pq(·,·)=y'_pq(·)+y'_qp(·)}
 ∀pq, if load'_pq(x'_p,x'_q)>w_pq d(x'_p,x'_q)  {This implies x'_p=c or x'_q=c}
        adjust y'_pq(c) so that load'_pq(x'_p,x'_q)=w_pq d(x'_p,x'_q)
```

**Fig. 4:** Fast-PD's pseudocode.

reduce $\text{load}'_{pq}(x'_p,x'_q)$ for restoring (4). However, this inequality will hold true very rarely (*e.g.* for a metric $d(\cdot,\cdot)$, one may show that it can never hold), and so POSTEDIT_DUALS will modify a $c$-balance variable (thereby changing the height of a $c$-label) only in very seldom occasions.

- But, to allow for the above changes, we also need to modify the construction of graph $\mathcal{G}^c$ in UPDATE_DUALS_PRIMALS. In particular, for $c{\neq}x_p$ and $c{\neq}x_q$, the capacities of interior edges $pq, qp$ must now be set as follows:[5]

$$\text{cap}_{pq} = \big[w_{pq}d(c,x_q) - \text{load}_{pq}(c,x_q)\big]^+ , \qquad (8)$$

$$\text{cap}_{qp} = \big[w_{pq}d(x_p,c) - \text{load}_{pq}(x_p,c)\big]^+ , \qquad (9)$$

where $[x]^+ \equiv \max(x,0)$. Besides ensuring (5) (by not letting the balance variables increase too much), the main rationale behind the above definition of interior capacities is to also ensure that (after max-flow) condition (4) will be met by most pairs $(p,q)$, even if $(c,x_q)$ or $(x_p,c)$ are the next labels assigned to them (which is a good thing, since we will thus manage to avoid the need for a correction by POSTEDIT_DUALS for all but a few $p,q$). For seeing this, the crucial thing to observe is that if, say, $(c,x_q)$ are the next labels for $p$ and $q$, then capacity $\text{cap}_{pq}$ can be shown to represent the increase of $\text{load}_{pq}(c,x_q)$ after max-flow, *i.e.*:

$$\text{load}'_{pq}(c,x_q) = \text{load}_{pq}(c,x_q) + \text{cap}_{pq}. \qquad (10)$$

Hence, if the following inequality is true as well:

$$\text{load}_{pq}(c,x_q) \le w_{pq}d(c,x_q) , \qquad (11)$$

then condition (4) will do remain valid after max-flow, as the following trivial derivation shows:

$$\text{load}'_{pq}(c,x_q) \overset{(10),(8)}{=} \text{load}_{pq}(c,x_q) + \big[w_{pq}d(c,x_q) - \text{load}_{pq}(c,x_q)\big]^+ \overset{(11)}{=} w_{pq}d(c,x_q) \qquad (12)$$

But this means that a correction may need to be applied by POSTEDIT_DUALS only for pairs $p,q$ violating (11) (before max-flow). However, such pairs tend to be very rare in practice (*e.g.*, as one can prove, no such pairs exist when $d(\cdot,\cdot)$ is a metric), and thus very few corrections need to take place.

Fig. 5 summarizes how Fast-PD sets the capacities for all edges of $\mathcal{G}^c$. As already explained, the interior capacities (with the help of PREEDIT_DUALS, POSTEDIT_DUALS

[5]If $c = x_p$ or $c = x_q$, then $\text{cap}_{pq} = \text{cap}_{qp} = 0$ as before, *i.e.* as in PD3$_a$.

in a few cases) allow UPDATE_DUALS_PRIMALS to impose conditions (4),(5), while the exterior capacities allow UPDATE_DUALS_PRIMALS to impose condition (3). As a result, the next theorem holds (see [1] for a complete proof):

**Theorem 2.** *The last primal-dual pair* $(\mathbf{x}, \mathbf{y})$ *of Fast-PD satisfies* (3)-(5)*, and so* $\mathbf{x}$ *is an* $f_{\text{app}}$*-approximate solution.*

In fact, Fast-PD maintains all good optimality properties of the PD3$_a$ method. *E.g.*, for a metric $d(\cdot,\cdot)$, Fast-PD proves to be as powerful as $\alpha$-expansion (see [1]):

**Theorem 3.** *If* $d(\cdot,\cdot)$ *is a metric, then the Fast-PD algorithm computes the best* $c$*-expansion after any* $c$*-iteration.*

## 4. Efficiency of Fast-PD for single MRFs

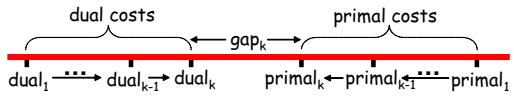
But, besides having all these good optimality properties, a very important advantage of Fast-PD over all previous primal-dual methods, as well as $\alpha$-expansion, is that it proves to be much more efficient in practice.

In fact, the computational efficiency for all methods of this kind is largely determined from the time taken by each max-flow problem, which, in turn, depends on the number of augmenting paths that need to be computed. For the case of Fast-PD, the number of augmentations per inner-iteration decreases dramatically, as the algorithm progresses. E.g. Fast-PD has been applied to the problem of image restoration, and fig. 7 contains a related result about the denoising of a corrupted (with gaussian noise) "penguin" image (256 labels and a truncated quadratic distance $d(a,b) = \min(|a-b|^2, D)$ - where $D = 200$ - has been used in this case). Also, fig. 8(a) shows the corresponding number of augmenting paths per outer-iteration (*i.e.* per group of $|\mathcal{L}|$ inner-iterations). Notice that, for both $\alpha$-expansion, as well as PD3$_a$, this number remains very high (*i.e.* almost over $2 \cdot 10^6$ paths) throughout all iterations. On the contrary, for the case of Fast-PD, it drops towards zero very quickly, *e.g.* only 4905 and 7 paths had to be found during the $8^{\text{th}}$ and last outer-iteration respectively (obviously, as also shown in Fig. 9(a), this directly affects the total time needed per outer-iteration). In fact, for the case of Fast-PD, it is very typical that, after very few inner-iterations, no more than 10 or 20 augmenting paths need to be computed per max-flow, which really boosts the performance in this case.

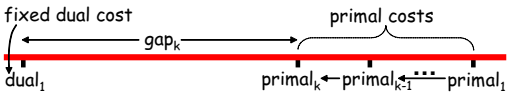
This property can be explained by the fact that Fast-PD maintains both a primal, as well as a dual solution throughout its execution. Fast-PD then manages to effectively use the dual solutions of previous inner iterations, so as to reduce the number of augmenting paths for the next inner-iterations. Intuitively, what happens is that Fast-PD ultimately wants to close the gap between the primal and the

| exterior capacities | interior capacities | | | |
|---|---|---|---|---|
| $cap_{sp}=[h_p(x_p)-h_p(c)]^+$ | $x_p \neq c$ $\wedge$ $x_q \neq c$ | $cap_{pq}=[w_{pq}d(c,x_q)-load_{pq}(c,x_q)]^+$ | $x_p = c$ $\vee$ $x_q = c$ | $cap_{pq}= 0$ |
| $cap_{pt}=[h_p(c)-h_p(x_p)]^+$ | | $cap_{qp}=[w_{pq}d(x_p,c)-load_{pq}(x_p,c)]^+$ | | $cap_{qp}= 0$ |

**Fig. 5:** Capacities of graph $\mathcal{G}^c$, as set by Fast-PD.




(a) High-level view of the Fast-PD algorithm




(b) High-level view of the $\alpha$-expansion algorithm

**Fig. 6:** **(a)** Fast-PD generates pairs of primal-dual solutions iteratively, with the goal of always reducing the primal-dual gap (*i.e.* the gap between the resulting primal and dual costs). But, for the case of Fast-PD, this gap can be viewed as a rough estimate for the number of augmentations, and so this number is forced to reduce over time as well. **(b)** On the contrary, $\alpha$-expansion works only in the primal domain (*i.e.* it is as if a fixed dual cost is used at the start of each new iteration) and thus the primal-dual gap can never become small enough. Therefore, no significant reduction in the number of augmentations takes place as the algorithm progresses.

dual cost (see Theorem 1), and, for this, it iteratively generates primal-dual pairs, with the goal of decreasing the size of this gap (see Fig. 6(a)). But, for Fast-PD, the gap's size can be thought of as, roughly speaking, an upper-bound for the number of augmenting paths per inner-iteration. Since, furthermore, Fast-PD manages to reduce this gap at any time throughout its execution, the number of augmenting paths is forced to decrease over time as well.

On the contrary, a method like $\alpha$-expansion, that works only in the primal domain, ignores dual solutions completely. It is, roughly speaking, as if $\alpha$-expansion is resetting the dual solution to zero at the start of each inner-iteration, thus effectively forgetting that solution thereafter (see Fig. 6(b)). For this reason, it fails to reduce the primal-dual gap and thus also fails to achieve a reduction in path augmentations over time, *i.e.* across inner-iterations. But the PD3$_a$ algorithm as well fails to mimic Fast-PD's behavior (despite being a primal-dual method). As explained in sec. 3, this happens because, in this case, PREEDIT_DUAL and POSTEDIT_DUAL temporarily destroy the gap just before the start of UPDATE_DUALS_PRIMALS, *i.e.* just before max-flow is about to begin computing the augmenting paths. (Note, of course, that this destruction is only temporary, and the gap is restored again after the execution of UPDATE_DUALS_PRIMALS).

The above mentioned relationship between primal-dual gap and number of augmenting paths is formally described in the next theorem (see [1] for a complete proof):

**Theorem 4.** *For Fast-PD, the primal-dual gap at the current inner-iteration forms an approximate upper bound for the number of augmenting paths at each iteration thereafter.*

***Sketch of proof.*** During a $c$-iteration, it can be shown that dual-cost $\leq \sum_p \min(h_p(c), h_p(x_p))$, whereas primal-cost$=$ $\sum_p h_p(x_p)$, and so the primal-dual gap upper-bounds the following quantity: $\sum_p [h_p(x_p) - h_p(c)]^+ = \sum_p \text{cap}_{sp}$.

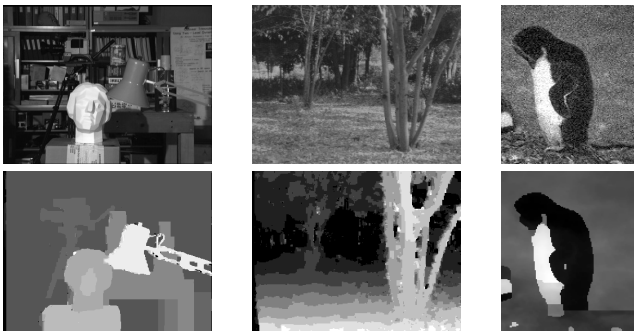


**Fig. 7: Left:** "Tsukuba"image and its disparity by Fast-PD. **Middle:** a "SRI tree" image and corresponding disparity by Fast-PD. **Right:** noisy "penguin" image and its restoration by Fast-PD.

But this quantity obviously forms an upper-bound on the maximum flow, which, in turn, upper-bounds the number of augmentations (assuming integral flows). □

Due to the above mentioned property, the time per outer-iteration decreases dramatically over time. This has been verified experimentally with virtually all problems that Fast-PD has been tested on. *E.g.* Fast-PD has been also applied to the problem of stereo matching, and fig. 7 contains the resulting disparity (of size $384 \times 288$ with 16 labels) for the well-known "Tsukuba" stereo pair, as well as the resulting disparity (of size $256 \times 233$ with 10 labels) for an image pair from the well-known "SRI tree" sequence (in both cases, a truncated linear distance $d(a,b) = \min(|a-b|, D)$ - with $D = 2$ and $D = 5$ - has been used, while the weights $w_{pq}$ were allowed to vary based on the image gradient at $p$). Figures 9(b), 9(c) contain the corresponding running times per outer iteration. Notice how much faster the outer-iterations of Fast-PD become as the algorithm progresses, e.g. the last outer-iteration of Fast-PD (for the "SRI-tree" example) lasted less than 1 msec (since, as it turns out, only 4 augmenting paths had to be found during that iteration). Contrast this with the behavior of either the $\alpha$-expansion or the PD3$_a$ algorithm, which both require an almost constant amount of time per outer-iteration, *e.g.* the last outer-iteration of $\alpha$-expansion needed more than 0.4 secs to finish (*i.e. it was more than 400 times slower than Fast-PD's iteration!*). Similarly, for the "Tsukuba" example, $\alpha$-expansion's last outer-iteration was more than 2000 times slower than Fast-PD's iteration.

**Max-flow algorithm adaptation:** However, for fully exploiting the decreasing number of path augmentations and reduce the running time, we had to properly adapt the max-flow algorithm. To this end, the crucial thing to observe was that the decreasing number of augmentations was directly related to the decreasing number of $\mathfrak{s}$-linked nodes, as already explained in sec. 3. *E.g.* fig. 8(b) shows how the number of $\mathfrak{s}$-linked nodes varies per outer-iteration for the "penguin" example (with a similar behavior being observed for the other examples as well). As can be seen, this number decreases drastically over time. In fact, as

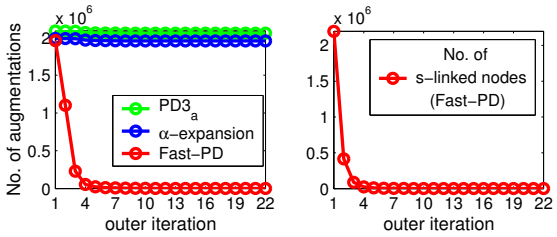
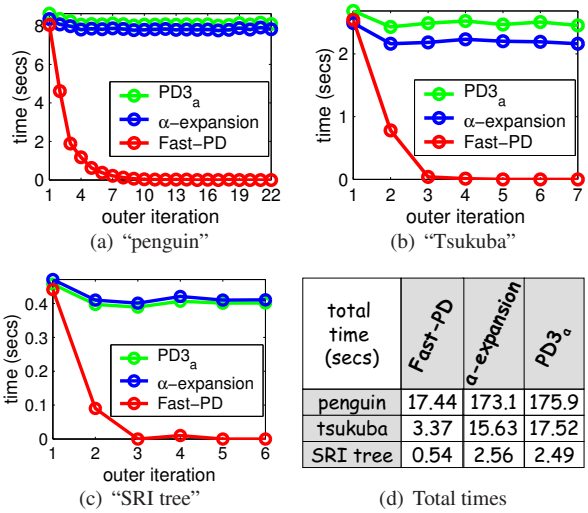



**Fig. 8: (a)** Number of augmenting paths per outer iteration for the "penguin" example (similar results hold for the other examples as well). Only in the case of Fast-PD, this number decreases dramatically over time. **(b)** This property of Fast-PD is directly related to the decreasing number of $\mathfrak{s}$-linked nodes per outer-iteration (this number is shown here for the same example as in (a)).




| total time (secs) | Fast-PD | α-expansion | PD3$_a$ |
|---|---|---|---|
| penguin | 17.44 | 173.1 | 175.9 |
| tsukuba | 3.37 | 15.63 | 17.52 |
| SRI tree | 0.54 | 2.56 | 2.49 |

(d) Total times

**Fig. 9:** Total time per outer iteration for the **(a)** "penguin", **(b)** "Tsukuba" and **(c)** "SRI tree" examples. **(d)** Total running times. For all experiments of this paper, a 1.6GHz laptop has been used.

implied by condition (3), no $\mathfrak{s}$-linked nodes will finally exist upon the algorithm's termination. Any augmentation-based max-flow algorithm striving for computational efficiency, should certainly exploit this property when trying to extract its augmenting paths. The most efficient of these algorithms [2] maintains 2 search trees for the fast extraction of these paths, a *source* and a *sink* tree. Here, the source tree will start growing by exploring non-saturated edges that are adjacent to $\mathfrak{s}$-linked nodes, whereas the sink tree will grow starting from all $\mathfrak{t}$-linked nodes. Of course, the algorithm terminates when no adjacent unsaturated edges can be found any more. However, in our case, maintaining the sink tree is completely inefficient and does not exploit the much smaller number of $\mathfrak{s}$-linked nodes. We thus propose maintaining only the source tree during max-flow, which will be a much cheaper thing to do here (*e.g.*, in many inner iterations, there can be fewer than 10 $\mathfrak{s}$-linked nodes, but many thousands of $\mathfrak{t}$-linked nodes). Moreover, due to the small size of the source tree, detecting the termination of the max-flow procedure can now be done a lot faster, *i.e.* with-

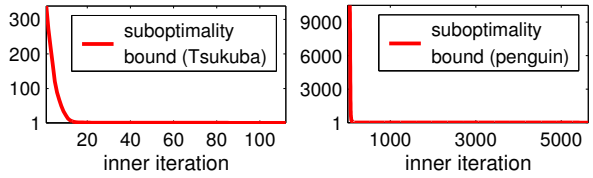



**Fig. 10:** Suboptimality bounds per inner iteration (for "Tsukuba" and "penguin"). These bounds drop to 1 very fast, meaning that the corresponding solutions have become almost optimal very early.

out having to fully expand the large sink tree (which is a very costly operation), thus giving a substantial speedup. In addition to that, for efficiently building the source tree, we keep track of all $\mathfrak{s}$-linked nodes and don't recompute them from scratch each time. In our case, this tracking can be done without cost, since, as explained in sec. 3, an $\mathfrak{s}$-linked node can be created only inside the PREEDIT_DUALS or the POSTEDIT_DUALS routine, and thus can be easily detected. The above simple strategy has been extremely effective for boosting the performance of max-flow, especially when a small number of augmentations were needed.

**Incremental graph construction:** But besides the max-flow algorithm adaptation, we may also modify the way graph $\mathcal{G}^c$ is constructed. *I.e.* instead of constructing the capacitated graph $\mathcal{G}^c$ from scratch each time, we also propose an incremental way of setting its capacities. The following lemma turns out to be crucial in this regard:

**Lemma 1.** *Let $\mathcal{G}^c$, $\bar{\mathcal{G}}^c$ be the graphs for the current and previous $c$-iteration. Let also $p, q$ be 2 neighboring MRF nodes. If, during the interval from the previous to the current $c$-iteration, no change of label took place for $p$ and $q$, then the capacities of the interior edges $pq, qp$ in $\mathcal{G}^c$ and of the exterior edges $\mathfrak{s}p, p\mathfrak{t}, \mathfrak{s}q, q\mathfrak{t}$ in $\mathcal{G}^c$ equal the residual capacities of the corresponding edges in $\bar{\mathcal{G}}^c$.*

The proof follows directly from the fact that if no change of label took place for $p, q$, then none of the height variables $h_p(x_p), h_q(x_q)$ or the balance variables $y_{pq}(x_p), y_{qp}(x_q)$ could have changed. Due to lemma 1, for building graph $\mathcal{G}^c$, we can simply reuse the residual graph of $\bar{\mathcal{G}}^c$ and only recompute those capacities of $\mathcal{G}^c$ for which the above lemma does not hold, thus speeding-up the algorithm even further.

**Combining speed with optimality:** Fig. 9(d) contains the running times of Fast-PD for various MRF problems. As can be seen from that figure, Fast-PD proves to be much faster than either the $\alpha$-expansion[6] or the PD3$_a$ method, *e.g.* Fast-PD has been more than 9 times faster than $\alpha$-expansion for the case of the "penguin" image (17.44 secs vs 173.1 secs). In fact, this behavior is a typical one, since Fast-PD has consistently provided at least a 3-9 times speedup for all the problems it has been tested on. However, besides its efficiency, Fast-PD does not make any compromise regarding the optimality of its solutions. On one hand, this is ensured by theorems 2, 3. On the other hand, Fast-PD, like any other primal-dual method, can also tell for free how well it performed by always providing a per-instance suboptimality bound for its solution. This comes at no extra cost, since any ratio between the cost of a primal solution and the cost of a dual solution can form such a bound. *E.g.* fig. 10 shows how these ratios vary per inner-iteration for the "tsukuba" and "penguin" problems (with similar results holding for the other problems as well). As one can notice, these ratios drop to 1 very quickly, meaning that an almost optimal solution has already been estimated even after just a few iterations (and despite the problem being NP-hard).

[6] Since $\alpha$-expansion cannot be used if $d(\cdot,\cdot)$ is not a metric, the method proposed in [7] had to be used for the cases of a non-metric $d(\cdot,\cdot)$.

$[\mathbf{x}, \mathbf{y}] \leftarrow$ INIT_DUALS_PRIMALS$(\bar{\mathbf{x}}, \bar{\mathbf{y}})$: $\mathbf{x} \leftarrow \bar{\mathbf{x}}$; $\mathbf{y} \leftarrow \bar{\mathbf{y}}$;
$\forall pq,\ y_{pq}(x_p) \mathrel{+}= w_{pq} d(x_p, x_q) - \bar{w}_{pq} \bar{d}(x_p, x_q)$;
$\forall p,\ \ h_p(\cdot) \mathrel{+}= \mathfrak{c}_p(\cdot) - \overline{\mathfrak{c}}_p(\cdot)$;

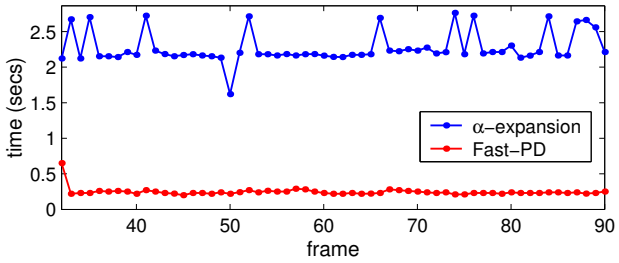
**Fig. 11:** Fast-PD's new pseudocode for dynamic MRFs.

## 5. Dynamic MRFs

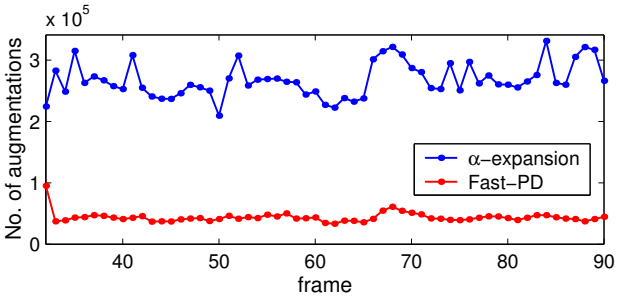
But, besides single MRFs, Fast-PD can be easily adapted to also boost the efficiency for dynamic MRFs [5], *i.e.* MRFs varying over time, thus showing the generality and power of the proposed method. In fact, Fast-PD fits perfectly to this task. The implicit assumption here is that the change between successive MRFs is small, and so, by initializing the current MRF with the final (primal) solution of the previous MRF, one expects to speed up inference. A significant advantage of Fast-PD in this regard, however, is that it can exploit not only previous MRF's primal solution (say $\bar{\mathbf{x}}$), but also its dual solution (say $\bar{\mathbf{y}}$). And this, for initializing current MRF's both primal and dual solutions (say $\mathbf{x}, \mathbf{y}$).

Obviously, for initializing $\mathbf{x}$, one can simply set $\mathbf{x} = \bar{\mathbf{x}}$. Regarding the initialization of $\mathbf{y}$, however, things are slightly more complicated. For maintaining Fast-PD's optimality properties, it turns out that, after setting $\mathbf{y} = \bar{\mathbf{y}}$, a slight correction still needs to be applied to $\mathbf{y}$. In particular, Fast-PD requires its initial solution $\mathbf{y}$ to satisfy condition (4), *i.e.* $y_{pq}(x_p) + y_{qp}(x_q) = w_{pq} d(x_p, x_q)$, whereas $\bar{\mathbf{y}}$ satisfies $\bar{y}_{pq}(x_p) + \bar{y}_{qp}(x_q) = \bar{w}_{pq} \bar{d}(x_p, x_q)$, *i.e.* condition (4) with $w_{pq} d(\cdot,\cdot)$ replaced by the pairwise potential $\bar{w}_{pq} \bar{d}(\cdot,\cdot)$ of the previous MRF. The solution for fixing that is very simple: *e.g.* we can simply set $y_{pq}(x_p) \mathrel{+}= w_{pq} d(x_p, x_q) - \bar{w}_{pq} \bar{d}(x_p, x_q)$. Finally, for taking into account the possibly different singleton potentials between successive MRFs, the new heights will obviously need to be updated as $h_p(\cdot) \mathrel{+}= \mathfrak{c}_p(\cdot) - \overline{\mathfrak{c}}_p(\cdot)$, where $\overline{\mathfrak{c}}_p(\cdot)$ are the singleton potentials of the previous MRF. These are the only changes needed for the case of dynamic MRFs, and thus the new pseudocode appears in Fig. 11.

As expected, for dynamic MRFs, the speedup provided by Fast-PD is even greater than single MRFs. *E.g.* Fig. 12(a) shows the running times per frame for the "SRI tree" image sequence. Fast-PD proves to be be more than 10 times faster than $\alpha$-expansion in this case (requiring on average 0.22 secs per frame, whereas $\alpha$-expansion required 2.28 secs on average). Fast-PD can thus run on about 5




(a) Running times per frame for the "SRI tree" sequence



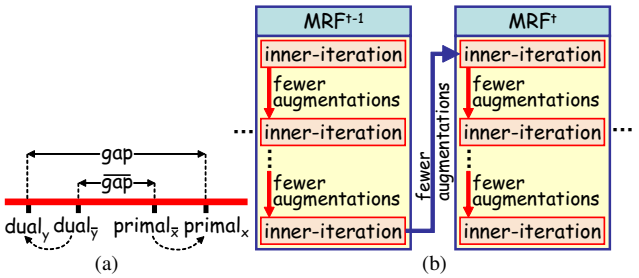

(b) Augmenting paths per frame for the "SRI tree" sequence

**Fig. 12:** Statistics for the "SRI tree" sequence.

frames/sec, *i.e*. it can do stereo matching almost in real time for this example (in fact, if successive MRFs bear greater similarity, even much bigger speedups can be achieved). Furthermore, fig. 12(b) shows the corresponding number of augmenting paths per frame for the "SRI tree" image sequence (for both $\alpha$-expansion and Fast-PD). As can be seen from that figure, a substantial reduction in the number of augmenting paths is achieved by Fast-PD, which helps that algorithm to reduce its running time.

This same behavior has been observed in all other dynamic problems that Fast-PD has been tested on as well. Intuitively, what happens is illustrated in Fig. 13(a). Fast-PD has already managed to close the gap between the final primal-dual costs $\text{primal}_{\bar{\mathbf{x}}}$, $\text{dual}_{\bar{\mathbf{y}}}$ of the previous MRF. However, due to the possibly different singleton (*i.e*. $\mathfrak{c}_p(\cdot)$) or pairwise (*i.e*. $w_{pq}d(\cdot,\cdot)$) potentials of the current MRF, these costs need to be perturbed to generate the new initial costs $\text{primal}_{\mathbf{x}}$, $\text{dual}_{\mathbf{y}}$. Nevertheless, as only slight perturbations take place, the new primal-dual gap (*i.e*. between $\text{primal}_{\mathbf{x}}$, $\text{dual}_{\mathbf{y}}$) will still be close to the previous gap (*i.e*. between $\text{primal}_{\bar{\mathbf{x}}}$, $\text{dual}_{\bar{\mathbf{y}}}$). As a result, the new gap will remain small. Few augmenting paths will therefore have to be found for the current MRF, and thus the algorithm's performance is boosted.

Put otherwise, for the case of dynamic MRFs, Fast-PD manages to boost performance, *i.e*. reduce number of augmenting paths, across two different "axes". The first axis lies along the different inner-iterations of the same MRF (*e.g*. see red arrows in Fig. 13(b)), whereas the second axis extends across time, *i.e*. across different MRFs (*e.g*. see blue arrow in Fig. 13(b), connecting the last iteration of $\text{MRF}^{t-1}$ to the first iteration of $\text{MRF}^{t}$).




(a) (b)

**Fig. 13:** **(a)** The final costs $\text{primal}_{\bar{\mathbf{x}}}$, $\text{dual}_{\bar{\mathbf{y}}}$ of the previous MRF are slightly perturbed to give the initial costs $\text{primal}_{\mathbf{x}}$, $\text{dual}_{\mathbf{y}}$ of the current MRF. Therefore, the initial primal-dual gap of the current MRF will be close to the final primal-dual gap of the previous MRF. Since the latter is small, so will be the former, and thus few augmenting paths will need to be computed for the current MRF. **(b)** Fast-PD reduces the number of augmenting paths in 2 ways: internally, *i.e*. across iterations of the same MRF (see red arrows), as well as externally, *i.e*. across different MRFs (see blue arrow).

## 6. Conclusions

In conclusion, a new graph-cut based method for MRF optimization has been proposed. It generalizes $\alpha$-expansion, while it also manages to be substantially faster than this state-of-the-art technique. Hence, regarding optimization of static MRFs, this method provides a significant speedup. In addition to that, however, it can also be used for boosting the performance of dynamic MRFs. In both cases, its efficiency comes from the fact that it exploits information not only from the "primal" problem (*i.e*. the MRF optimization problem), but also from a "dual" problem. Moreover, despite its speed, the proposed method can nevertheless guarantee almost optimal solutions for a very wide class of NP-hard MRFs. Due to all of the above, and given the ubiquity of MRFs, we strongly believe that Fast-PD can prove to be an extremely useful tool for many problems in computer vision in the years to come.

## References

[1] N. Komodakis, G. Tziritas and N. Paragios. Fast Primal-Dual Strategies for MRF Optimization. Technical report, 2006. 5

[2] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *PAMI*, 26(9), 2004. 6

[3] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *PAMI*, 23(11), 2001. 1

[4] O. Juan and Y. Boykov. Active graph cuts. In *CVPR*, 2006. 1

[5] P. Kohli and P. H. Torr. Efficiently solving dynamic markov random fields using graph cuts. In *ICCV*, 2005. 1, 7

[6] N. Komodakis and G. Tziritas. A new framework for approximate labeling via graph-cuts. In *ICCV*, 2005. 1, 2, 3

[7] C. Rother, S. Kumar, V. Kolmogorov, and A. Blake. Digital tapestry. In *CVPR*, 2005. 7

[8] R. Szeliski, *et al*. A comparative study of energy minimization methods for markov random fields. In *ECCV*, 2006. 1