LABORATOIRE MATHEMATIQUES APPLIQUEES AUX SYSTEMES

Fast Primal-Dual Strategies for MRF Optimization

Nikos Komodakis — Georgios Tziritas — Nikos Paragios

N° 0605

December 2006

Projet Orasis

Fast Primal-Dual Strategies for MRF Optimization

Nikos Komodakis[*] , Georgios Tziritas[†] , Nikos Paragios[‡]

Projet Orasis — Orasis

Rapport de recherche n° 0605 — December 2006 — 28 pages

Abstract: A new efficient MRF optimization algorithm, called Fast-PD, is proposed, which generalizes α-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 α-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. In addition, despite being fast, Fast-PD makes no compromise about the optimality of its solutions: it can compute exactly the same answer as α-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 α-expansion.

Key-words: Fast-PD, Static Markov Random Fields, Dynamic Markov Random Fields, Linear Programming, , Optimization

[*] nikos.komodakis@ecp.fr, komod@csd.uoc.gr - Ecole Centrale de Paris, University of Crete
[†] tziritas@csd.uoc.gr - University of Crete
[‡] nikos.paragios@ecp.fr - Ecole Centrale de Paris

Laboratoire MAS
Ecole Centrale Paris, Grande Voie des Vignes
92290 CHATENAY-MALABRY Cedex (France)
Téléphone : +33 1 41 13 17 11 — Télécopie : +33 1 41 13 17 35

Fast Primal-Dual Strategies for MRF Optimization

Résumé : Nous proposons un nouvel algorithme efficace d'optimisation basé sur les champs de Markov. Cet algorithme s'appelle Fast-PD, et généralise l'algorithme de α-expansion. Parmi ses principaux avantages, nous notons sa grande rapidité par rapport à la dernière méthode. Il peut être de 3 à 9 fois plus rapide que l'algorithme de α-expansion. Il puise son efficacité dans l'exploitation des informations qui émanent non seulement du champ de Markov primal, mais aussi du problème dual. En outre, son utilisation pour accélérer les champs de Markov statiques s'étend aussi à l'amélioration des performances des champs de Markov dynamiques, c'est à dire ceux qui varient au cours du temps. En plus de sa rapidité, Fast-PD ne fait aucun compromis quant à l'optimalité de ses solutions : il peut calculer les mêmes solutions que α-expansion, mais contrairement à cette méthode, il peut aussi garantir une solution presque optimale pour une classe beaucoup plus large de problèmes NP-difficiles de champs de Markov. Les résultats obtenus sur des champs de Markov statiques et dynamiques démontrent l'efficacité et la puissance de cet algorithme. A titre d'exemple, Fast-PD est utilisé pour calculer la disparité sur des séquences d'images stéréoscopiques en temps réel. La disparité obtenue est la même que celle qui résulte de l'algorithme de α-expansion.

Mots-clés : Fast-PD, Champs de Markov (Statiques et Dynamiques), programmation linéaire, optimisation.

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:

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

In the above formula, $\mathfrak{c}_p(\cdot)$, $d(\cdot,\cdot)$ determine the singleton and pairwise MRF potential functions respectively[1].

Up to now, graph-cut based methods, like α-expansion [2], have been very effective in MRF optimization, generating solutions with good optimality properties [7]. 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 [4, 3], 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 α-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: α-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 [5], and propose a new primal-dual MRF optimization method, called Fast-PD. This method, like [5] or α-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 this end, we also 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, for fully exploiting this property two extensions are also proposed: an *adapted max-flow algorithm*, as well as an *incremental graph construction* method.

Optimality properties: But, 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 α-expansion, i.e. it computes exactly the same solution, but with a substantial speedup. Moreover, it applies to a much wider class of MRFs[2], e.g. even with a non-metric $d(\cdot,\cdot)$, while still guaranteeing an almost optimal solution.

[1]Hereafter, $d(\cdot,\cdot)$ will be called simply a distance function
[2]Fast-PD requires only $d(a,b) \geq 0,\ d(a,b) = 0 \Leftrightarrow a = b$

Efficiency for dynamic MRFs: Furthermore, our method can also be used for boosting the efficiency of dynamic MRFs. We note here that many works have been proposed in this regard recently [4, 3]. 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 technical report is organized as follows. In sec. 2, we briefly review the work of [5] about using the primal-dual schema for MRF optimization. The Fast-PD algorithm is then described in sec. 3. Its efficiency for optimizing 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, while appendices A and B contain technical proofs for the theorems of this report.

2 Primal-dual MRF optimization algorithms

In this section, we review very briefly the work of [5]. Consider the primal-dual pair of linear programs, given by:

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

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_{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 [5], 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 (Linear Programs), the authors then show that, for Theorem 1 to be true with $f_{\text{app}}=2\frac{d_{\max}}{d_{\min}}$[3], it suffices that the next (so-called *relaxed complementary slackness*) conditions

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

```
1: $[\mathbf{x},\mathbf{y}] \leftarrow$ INIT_DUALS_PRIMALS( ); $\mathbf{x}_{\text{old}} \leftarrow \mathbf{x}$
2: **for** each label $c$ in $\mathcal{L}$ **do**
3:     $\mathbf{y} \leftarrow$ PREEDIT_DUALS$(c,\mathbf{x},\mathbf{y})$;
4:     $[\mathbf{x}',\mathbf{y}'] \leftarrow$ UPDATE_DUALS_PRIMALS$(c,\mathbf{x},\mathbf{y})$;
5:     $\mathbf{y}' \leftarrow$ POSTEDIT_DUALS$(c,\mathbf{x}',\mathbf{y}')$;
6:     $\mathbf{x} \leftarrow \mathbf{x}'$; $\mathbf{y} \leftarrow \mathbf{y}'$;
7: **end for**
8: **if** $\mathbf{x} \neq \mathbf{x}_{\text{old}}$ **then**
9:     $\mathbf{x}_{\text{old}} \leftarrow \mathbf{x}$; **goto** 2;
10: **end if**
```

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

hold true for the resulting primal and dual variables:

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

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

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

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 are 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:

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

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

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

Base on the above definition, conditions (4),(5) can be rewritten as:

$$\text{load}_{pq}(x_p, x_q) = w_{pq} d(x_p, x_q), \quad \forall pq\in\mathcal{E} \tag{8}$$

$$\text{load}_{pq}(a,b) \le 2w_{pq} d_{\max}, \quad \forall pq\in\mathcal{E}, a\in\mathcal{L}, b\in\mathcal{L} \tag{9}$$

and so, whenever we refer to conditions (4),(5) hereafter, we will implicitly refer to conditions (8),(9) as well.

The main goal of a MRF primal-dual method is to satisfy all conditions (3)-(5), and, to this end, the primal-dual variables are iteratively updated until all of these conditions become true. The basic structure of a primal-dual algorithm can be seen in Fig. 1. During an inner c-iteration (lines 3-6 in

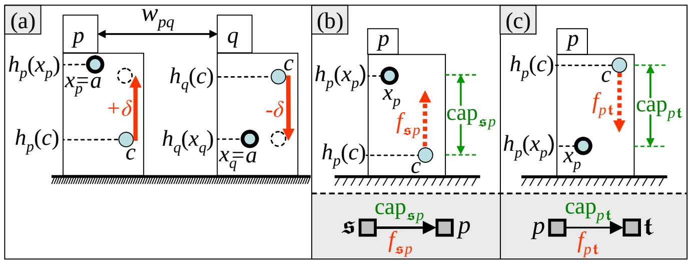


Fig. 2: (a) Dual variables' visualization for a simple MRF with 2 nodes $\{p, q\}$ and 2 labels $\{a, c\}$. A copy of labels $\{a, c\}$ exists for every node, and all these labels are represented by balls floating at certain heights. The role of the *height variables* $h_p(\cdot)$ is to specify exactly 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 during a c-iteration, 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 during a c-iteration, 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 label c will still remain above x_p).

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.

The main update of the primal and dual variables takes place in UPDATE_DUALS_PRIMALS during an inner iteration, and (as it was shown in [5]) 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.[4] Also, for simplicity's sake, note that we will hereafter refer to only one of the methods derived in [5], 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

[4]Throughout this technical report, we use the following convention for the notation of the dual variables during an inner-iteration: before the UPDATE_DUALS_PRIMALS routine, all dual variables are denoted without an accent, e.g. $y_{pq}(\cdot)$, $h_p(\cdot)$. After UPDATE_DUALS_PRIMALS has updated the dual variables, we always use an accent for denoting these variables, e.g. we write $y'_{pq}(\cdot)$, $h'_p(\cdot)$ in this case.

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 [5]: 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.

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 (see Fig. 3). Furthermore, the main movement of all c-labels (i.e. the main update of dual variables $y_{pq}(c)$ and $h_p(c)$ for all p, q) takes place in UPDATE_DUALS_PRIMALS, and this movement has been shown that it 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

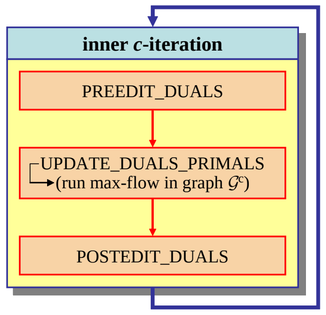


Fig. 3: The basic structure of an inner c-iteration is shown here. During such an iteration, only the c-labels are allowed to move (i.e. only them can change their heights). The main movement of the c-labels takes place inside the UPDATE_DUALS_PRIMALS routine, and this movement is simulated by pushing the maximum-flow through an appropriate directed graph $\mathcal{G}^c$. However, besides the movement during UPDATE_DUALS_PRIMALS, c-labels also move before and after that routine as well. This happens because routines PREEDIT_DUALS and POSTEDIT_DUALS also apply corrections to the dual variables, and these corrections take place before and after max-flow respectively.

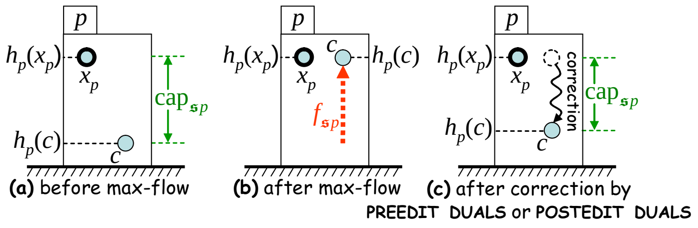


Fig. 4: **(a)** Label c at p is below x_p, and thus label c is allowed to raise itself in order to reach x_p. This means that p will be an $\mathfrak{s}$-linked node of graph $\mathcal{G}^c$, i.e. $\text{cap}_{\mathfrak{s}p} > 0$, and thus a non-zero flow $f_{\mathfrak{s}p}$ (representing the total raise of label c in UPDATE_DUALS_PRIMALS) 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 (i.e. after max-flow), 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 later to c-label's height by PREEDIT_DUALS or POSTEDIT_DUALS, 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)).

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 during UPDATE_DUALS_PRIMALS. In particular, the corresponding flows f_{pq}/f_{qp} 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 these edges are responsible to ensure (along with PREEDIT_DUALS, POSTEDIT_DUALS) that conditions (4), (5) hold true.

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 during UPDATE_DUALS_PRIMALS), as well as their capacities (which are left with ensuring condition (3) on their own), 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 during 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 $\text{cap}_{\mathfrak{s}p} = h_p(x_p) - h_p(c)$.

```
$[\mathbf{x},\mathbf{y}]\leftarrow$ INIT_DUALS_PRIMALS( ):  $\mathbf{x}\leftarrow$ random labels; $\mathbf{y}\leftarrow\mathbf{0}$;
    $\forall pq$, adjust $y_{pq}(x_p)$ or $y_{qp}(x_q)$ so that $\text{load}_{pq}(x_p,x_q)=w_{pq}d(x_p,x_q)$

$\mathbf{y}\leftarrow$ PREEDIT_DUALS$(c,\mathbf{x},\mathbf{y})$:
    $\forall pq$, if $\text{load}_{pq}(c,x_q)>w_{pq}d(c,x_q)$ or $\text{load}_{pq}(x_p,c)>w_{pq}d(x_p,c)$
        adjust $y_{pq}(c)$ so that $\text{load}_{pq}(c,x_q)=w_{pq}d(c,x_q)$

$[\mathbf{x}',\mathbf{y}']\leftarrow$ UPDATE_DUALS_PRIMALS$(c,\mathbf{x},\mathbf{y})$:  $\mathbf{x}'\leftarrow\mathbf{x}$; $\mathbf{y}'\leftarrow\mathbf{y}$;
    Construct $\mathcal{G}^c$ and apply max-flow to compute all flows $f_{\mathfrak{s}p}/f_{p\mathfrak{t}}$, $f_{pq}$
    $\forall pq$, $y'_{pq}(c)\leftarrow y_{pq}(c)+f_{pq}-f_{qp}$
    $\forall p$ , if an unsaturated path from $\mathfrak{s}$ to $p$ exists, then $x'_p\leftarrow c$

$\mathbf{y}'\leftarrow$ POSTEDIT_DUALS$(c,\mathbf{x}',\mathbf{y}')$: {We denote $\text{load}'_{pq}(\cdot,\cdot)=y'_{pq}(\cdot)+y'_{qp}(\cdot)$}
    $\forall pq$, if $\text{load}'_{pq}(x'_p,x'_q)>w_{pq}d(x'_p,x'_q)$ {This implies $x'_p=c$ or $x'_q=c$}
        adjust $y'_{pq}(c)$ so that $\text{load}'_{pq}(x'_p,x'_q)=w_{pq}d(x'_p,x'_q)$
```

Fig. 5: Fast-PD's pseudocode.

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[5]. 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 4, where, due to UPDATE_DUALS_PRIMALS (i.e. due to max-flow), a c-label has initially managed to reach an active label x_p, but it has again dropped below x_p, due to some correction 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.

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. 5):

[5]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 once again helps condition (3) to become true, and forms the main rationale behind the update of the primal variables $\mathbf{x}$ in UPDATE_DUALS_PRIMALS.

- 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 $\mathrm{load}_{pq}(c, x_q){>}w_{pq}d(c, x_q)$ or $\mathrm{load}_{pq}(x_p, c){>}w_{pq}d(x_p, c)$ (which, in practice, happens much more rarely). In this case, a modification is needed (see code in Fig. 5), 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:

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

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 (14)). 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 balance variables $y'_{pq}(x'_p)$ (with $x'_p = c$) or $y'_{qp}(x'_q)$ (with $x'_q = c$) are modified much more rarely by the new POSTEDIT_DUALS. So, whereas the previous version modified these variables (thereby changing the height of a c-label) whenever they were negative (which, in practice, happened most of the time), the new routine applies a modification only if $\mathrm{load}'_{pq}(x'_p, x'_q){>} w_{pq}d(x'_p, x'_q)$,[6] which may happen only in very seldom occasions (e.g. if the distance function $d(\cdot,\cdot)$ is a metric, one may then show that this can never happen). If the above inequality does hold true, then POSTEDIT_DUALS simply needs to reduce $\mathrm{load}'_{pq}(x'_p, x'_q)$ so as to just restore (4).

- 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:[7]

$$\mathrm{cap}_{pq} = \left[w_{pq}d(c, x_q){-}\mathrm{load}_{pq}(c, x_q)\right]^+ , \tag{10}$$

$$\mathrm{cap}_{qp} = \left[w_{pq}d(x_p, c){-}\mathrm{load}_{pq}(x_p, c)\right]^+ , \tag{11}$$

where $[x]^+{\equiv}\max(x, 0)$. Besides ensuring condition (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), no matter if (c, x_q) or (x_p, c) are the next labels assigned to them (which is good, since we will thus manage to avoid the need for a correction by POSTEDIT_DUALS for all but a few p, q). To see this, the crucial thing to observe is that if, say, (c, x_q) are the next labels for p and q, then capacity cap_{pq} can be shown to represent the increase of $\mathrm{load}_{pq}(c, x_q)$ after max-flow, i.e. :

$$\mathrm{load}'_{pq}(c, x_q) = \mathrm{load}_{pq}(c, x_q) + \mathrm{cap}_{pq}. \tag{12}$$

Hence, if the following inequality is true as well:

$$\mathrm{load}_{pq}(c, x_q) \le w_{pq}d(c, x_q) , \tag{13}$$

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

[6]As in (7), we define $\mathrm{load}'_{pq}(a, b){\equiv}y'_{pq}(a){+}y'_{qp}(b)$ for variable $\mathbf{y}'$.
[7]If $c{=}x_p$ or $c{=}x_q$, then $\mathrm{cap}_{pq}{=}\mathrm{cap}_{qp}{=}0$ as before, i.e. as in PD3$_a$.

exterior capacities	**interior capacities**			
$\text{cap}_{\mathfrak{s}p}=[h_p(x_p)-h_p(c)]^+$	$x_p \neq c$ $\wedge$	$\text{cap}_{pq}=[w_{pq}d(c,x_q)-\text{load}_{pq}(c,x_q)]^+$	$x_p = c$ $\vee$	$\text{cap}_{pq}=0$
$\text{cap}_{p\mathfrak{t}}=[h_p(c)-h_p(x_p)]^+$	$x_q \neq c$	$\text{cap}_{qp}=[w_{pq}d(x_p,c)-\text{load}_{pq}(x_p,c)]^+$	$x_q = c$	$\text{cap}_{qp}=0$

Fig. 6: Capacities of graph $\mathcal{G}^c$, as set by Fast-PD.

$$\begin{aligned}\text{load}'_{pq}(c,x_q) &\overset{(12),\,(10)}{=} \text{load}_{pq}(c,x_q)+[w_{pq}d(c,x_q)-\text{load}_{pq}(c,x_q)]^+ \\ &\overset{(13)}{=} \text{load}_{pq}(c,x_q)+[w_{pq}d(c,x_q)-\text{load}_{pq}(c,x_q)] = w_{pq}d(c,x_q) \qquad (14)\end{aligned}$$

But this means that a correction may need to be applied by POSTEDIT_DUALS only for pairs p, q violating (13) (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. 6 summarizes how Fast-PD sets the capacities for all edges of $\mathcal{G}^c$. As already mentioned, based on the interior capacities, one may show that UPDATE_DUALS_PRIMALS (with the help of PREEDIT_DUALS, POSTEDIT_DUALS in a few cases) ensures (4),(5), while, thanks to the exterior capacities, UPDATE_DUALS_PRIMALS can ensure (3). As a result, the next theorem holds (see appendix A for a complete proof):

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

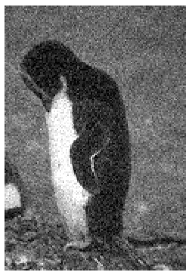
In fact, Fast-PD maintains all good optimality properties of the PD3$_a$ method. E.g. , for a metric $d(\cdot,\cdot)$, Fast-PD proves to be as powerful as α-expansion (see appendix B for a proof):

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

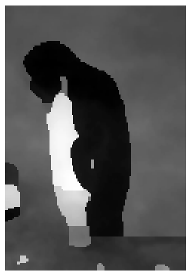
4 Efficiency of Fast-PD for single MRFs

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

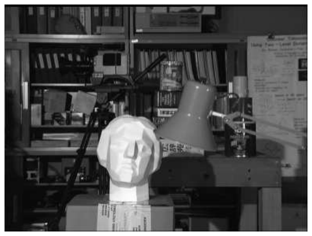
In fact, the computational efficiency for all methods of this kind is largely determined from the time taken by each max-flow problem, which, in turn, depends on the number of augmenting paths that need to be computed. For the case of Fast-PD, the number of augmentations per inner-iteration decreases dramatically, as the algorithm progresses. E.g. Fast-PD has been applied to the problem of image restoration, where, given a corrupted (by noise) image, one seeks to restore the original (uncorrupted) image back. In this case, labels correspond to intensities, while the singleton potential function $\mathfrak{c}_p(\cdot)$ was defined as a truncated squared difference $\mathfrak{c}_p(a) = \min\{|I_p - a|^2, 10^4\}$ between the label and the observed intensity I_p at pixel p. Fig. 7(b) 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 $D = 200$ - were also used in this case). Fig. 9(a) shows the



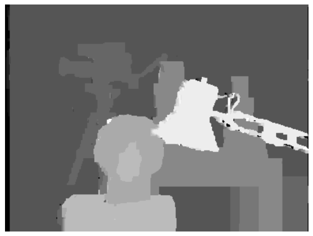
(a) Noisy "penguin"image



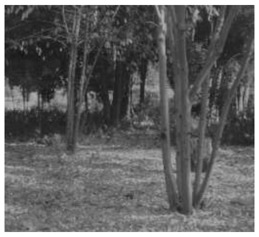
(b) Restoration of the "penguin"image by the Fast-PD algorithm



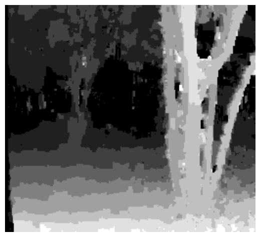
(c) "Tsukuba"image



(d) Corresponding disparity as estimated by Fast-PD



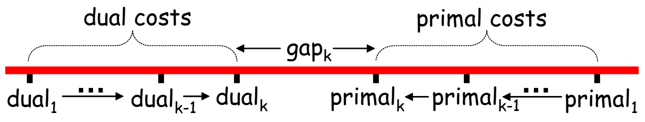
(e) "SRI tree"image



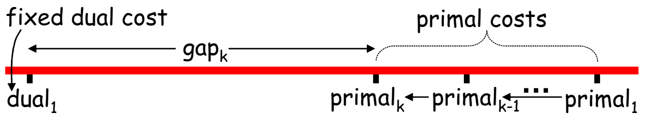
(f) Corresponding disparity as estimated by Fast-PD

Fig. 7: Image restoration and stereo matching results by the Fast-PD algorithm.

corresponding number of augmenting paths per outer-iteration (i.e. per group of $|\mathcal{L}|$ inner-iterations). Notice that, for both α-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^{th} and last outer-iteration respectively (obviously, as also shown in Fig. 10(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.



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



(b) High-level view of the α-expansion algorithm

Fig. 8: (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, α-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.

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-iteration. Intuitively, what happens is that Fast-PD ultimately wants to close the gap between the primal and the 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. 8(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 α-expansion, that works only in the primal domain, ignores dual solutions completely. It is, roughly speaking, as if α-expansion is resetting the dual solution to zero at the start of each inner-iteration, thus effectively forgetting that solution thereafter (see Fig. 8(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. However, not only the α-expansion, 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:

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.*

Proof. The same dual linear program as in [5] has been used, and so the cost of a dual solution is defined as:

$$\text{dual cost} = \sum_p \min_{a \in L} h_p(a) \ , \tag{15}$$

which implies that:

$$\text{dual cost} \leq \sum_p \min(h_p(c), h_p(x_p)) \tag{16}$$

Furthermore, in the case of the Fast-PD algorithm, it can be shown that the following equality will hold before the start of max-flow at an inner-iteration (see lemma B.1):

$$\text{primal cost} = \sum_p h_p(x_p) \tag{17}$$

Based on (16), (17), the following inequality then results:

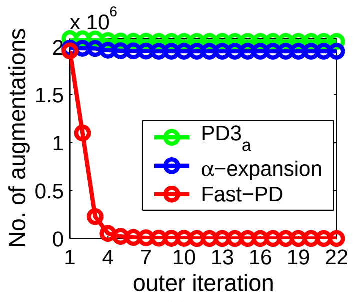
$$\begin{aligned} \text{primal dual gap} = \text{primal cost} - \text{dual cost} &\geq \sum_p h_p(x_p) - \sum_p \min(h_p(c), h_p(x_p)) \\ &= \sum_p [h_p(x_p) - h_p(c)]^+ = \sum_p \text{cap}_{\mathfrak{s}p}. \end{aligned} \tag{18}$$

But the quantity $\sum_p \text{cap}_{\mathfrak{s}p}$ obviously forms an upper-bound on the maximum flow during a c-iteration, which, in turn, upper-bounds the number of augmenting paths (assuming integral flows). In addition to that, the upper bound defined by $\sum_p \text{cap}_{\mathfrak{s}p}$ will not increase during any of the next c-iterations (which means that the number of augmentations will keep decreasing over time), and so the current primal-dual gap will be an approximate upper bound for the number of augmentations of the next c-iterations as well.

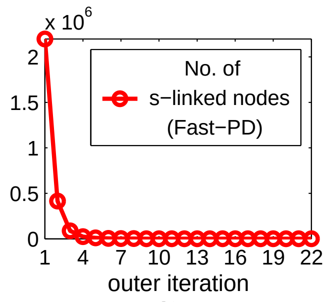
The fact that the upper bound $\sum_p \text{cap}_{\mathfrak{s}p} = \sum_p [h_p(x_p) - h_p(c)]^+$ will not increase during any of the next iterations may be justified by that any of the terms $[h_p(x_p) - h_p(c)]^+$ can increase only during either PREEDIT_DUALS or POSTEDIT_DUALS (it is easy to show that UPDATE_DUALS_PRIMALS may only decrease the value of these terms). However, both PREEDIT_DUALS and POSTEDIT_DUALS modify the height variables $h_p(\cdot)$ only in very rare occasions during the execution of Fast-PD (e.g. if $d(\cdot, \cdot)$ is a metric, one may prove that none of the height variables need to be altered by POSTEDIT_DUALS). Hence, the terms $[h_p(x_p) - h_p(c)]^+$ will typically not be altered by these routines (or they will be altered by a negligible amount at most), and so only UPDATE_DUALS_PRIMALS may modify these terms, thus decreasing their values.

□

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. In this case, the conventional measure of SSD (sum of squared differences) or SAD (sum of absolute differences) has been used for the singleton potentials $\mathfrak{c}_p(\cdot)$. Fig. 7(d) contains the resulting disparity (of size 384×288

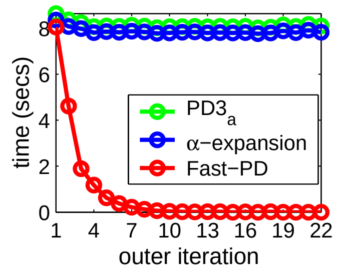


(a)

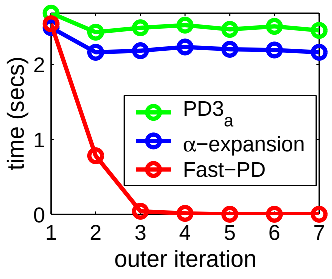


(b)

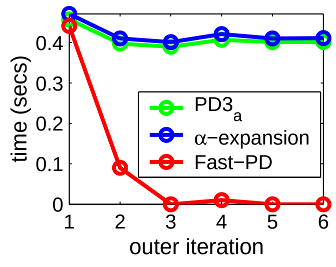
Fig. 9: (a) Number of augmenting paths per outer iteration for the "penguin" example (similar results hold for the other examples as well). Only in the case of Fast-PD, this number decreases dramatically over time. **(b)** This property of Fast-PD is directly related to the decreasing number of $\mathfrak{s}$-linked nodes per outer-iteration (this number is shown here for the same example as in (a)).



(a) "penguin"



(b) "Tsukuba"



(c) "SRI tree"

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

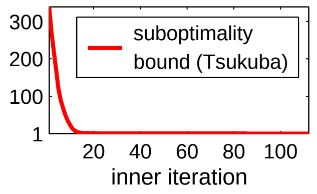
Fig. 10: Total time per outer iteration for the **(a)** "penguin", **(b)** "Tsukuba" and **(c)** "SRI tree" examples. **(d)** Resulting total running times for the same examples. (We note that for all experiments of this paper, a 1.6GHz laptop has been used).

with 16 labels) for the well-known "Tsukuba" stereo pair, while fig. 7(f) contains the resulting disparity (of size 256×233 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 10(b), 10(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 α-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 α-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, α-expansion's last outer-iteration was more than 2000 times slower than Fast-PD's iteration.

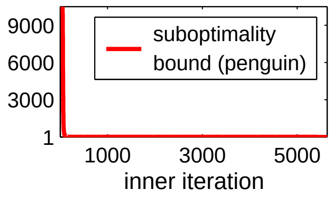
4.1 Max-flow algorithm adaptation

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. 9(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 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 [1] 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. without 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.



(a) Suboptimality bounds for stereo matching



(b) Suboptimality bounds for image restoration

Fig. 11: Suboptimality bounds (per inner iteration) for a stereo matching problem ("Tsukuba" example), as well as for an image restoration problem ("penguin" example). As can be seen, in both cases the bounds drop to 1 very fast, meaning that the corresponding solutions have become almost optimal very early (i.e. in very few iterations).

4.2 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 the above lemma, 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. This way, an additional speedup can be obtained in some cases.

4.3 Combining speed with optimality

Fig. 10(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 α-expansion[8] or the PD3$_a$ method, e.g. Fast-PD has been more than 9 times faster than α-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

[8]We note that the publicly available implementation of [7] has been used for the α-expansion algorithm. Furthermore, since α-expansion cannot be applied when $d(\cdot,\cdot)$ is not a metric, the extension proposed in [6] has been used for the cases where a non-metric distance function $d(\cdot,\cdot)$ was needed.

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. 11 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). Before proceeding, we should also note that no special tuning of either the singleton or the pairwise potential functions took place for deriving the results in Figure 7. Therefore, by properly adjusting these functions with more care, even better results may be obtained by the Fast-PD algorithm. E.g. Figure 12 displays the resulting disparity (for the "Tsukuba" image pair), when a Potts function (instead of a truncated linear function) has been used as the distance function $d(\cdot,\cdot)$.

5 Dynamic MRFs

But, besides single MRFs, Fast-PD can be easily adapted to also boost the efficiency for dynamic MRFs [4], 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

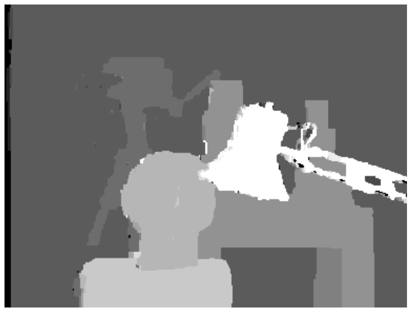
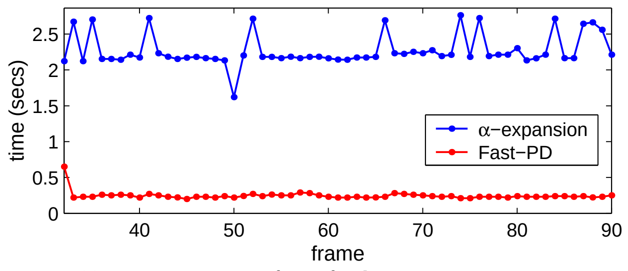
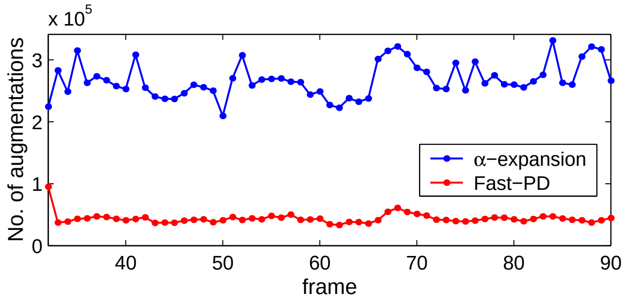


Fig. 12: Disparity for the "Tsukuba" image as estimated by the Fast-PD algorithm in the case where a Potts function has been used for the distance $d(\cdot,\cdot)$.



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



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

Fig. 13: Statistics for the “SRI tree” sequence.

can simply set $y_{pq}(x_p) {+}{=} w_{pq}d(x_p, x_q) {-} \bar{w}_{pq}\bar{d}(x_p, x_q)$. Finally, for taking into account the possibly different singleton potentials between successive MRFs, the new heights will obviously need to be updated as $h_p(\cdot) {+}{=} \mathfrak{c}_p(\cdot) {-} \bar{\mathfrak{c}}_p(\cdot)$, where $\bar{\mathfrak{c}}_p(\cdot)$ 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. 14.

As expected, for dynamic MRFs, the speedup provided by Fast-PD is even greater than single MRFs. E.g. Fig. 13(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 α-expansion in this case (requiring on average 0.22 secs per frame, whereas α-expansion required 2.28 secs on average). Fast-PD can thus run on about 5 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. 13(b)

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

Fig. 14: Fast-PD’s new pseudocode for dynamic MRFs.

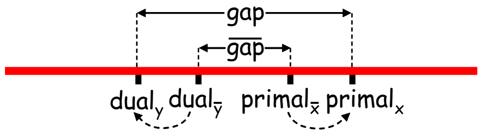


Fig. 15: The final costs $\text{primal}_{\bar{\mathbf{x}}}$, $\text{dual}_{\bar{\mathbf{y}}}$ of the previous MRF are slightly perturbed to give the initial costs $\text{primal}_{\mathbf{x}}$, $\text{dual}_{\mathbf{y}}$ of the current MRF. Therefore, the initial primal-dual gap of the current MRF will 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.

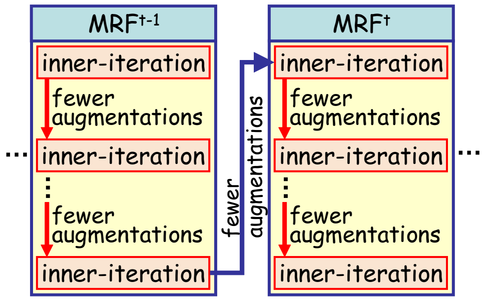


Fig. 16: Fast-PD reduces the number of augmenting paths in 2 ways: internally, i.e. across iterations of the same MRF (see red arrows), as well as externally, i.e. across different MRFs (see blue arrow).

shows the corresponding number of augmenting paths per frame for the "SRI tree" image sequence (for both α-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. 15. Fast-PD has already managed to close the gap between the costs $\text{primal}_{\bar{\mathbf{x}}}$, $\text{dual}_{\bar{\mathbf{y}}}$ of the final primal-dual solutions $\bar{\mathbf{x}}$, $\bar{\mathbf{y}}$ 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 costs $\text{primal}_{\mathbf{x}}$, $\text{dual}_{\mathbf{y}}$ for the initial solutions $\mathbf{x}, \mathbf{y}$ of the current MRF. 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 $\text{primal}_{\bar{\mathbf{x}}}$, $\text{dual}_{\bar{\mathbf{y}}}$) and 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. 16), whereas the second axis extends across time, i.e. across different MRFs (e.g. see blue arrow in Fig. 16, connecting last iteration of MRF^{t-1} to first iteration of MRF^{t}).

6 Conclusions

In conclusion, a new graph-cut based method for MRF optimization has been proposed. It generalizes α-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 the Fast-PD algorithm can prove to be an extremely valuable tool for many problems in computer vision in the years to come.

References

[1] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *In IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 26(9), 2004.

[2] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *In IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 23(11), 2001.

[3] O. Juan and Y. Boykov. Active graph cuts. *In Proc. Computer Vision and Pattern Recognition (CVPR)*, 2006.

[4] P. Kohli and P. H. Torr. Efficiently solving dynamic markov random fields using graph cuts. *In Proc. International Conference on Computer Vision (ICCV)*, 2005.

[5] N. Komodakis and G. Tziritas. A new framework for approximate labeling via graph-cuts. *In Proc. International Conference on Computer Vision (ICCV)*, 2005.

[6] C. Rother, S. Kumar, V. Kolmogorov, and A. Blake. Digital tapestry. *In Proc. Computer Vision and Pattern Recognition (CVPR)*, 2005.

[7] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields. *In Proc. European Conference on Computer Vision (ECCV)*, 2006.

[8] V. Kolmogorov and R. Zabih. What Energy Functions can be Minimized via Graph Cuts? *In IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 26(2), 2004.

Appendix A: Proof of Theorem 2 about the optimality of Fast-PD's solutions

The purpose of this section is to provide the proof for Theorem 2, which certifies that the solutions estimated by the Fast-PD algorithm have guaranteed optimality properties. But before that, the following 3 lemmas need to proved first:

Lemma A.1
During a c-iteration, the following inequalities hold true exactly after UPDATE_DUALS_PRIMALS*:*

$$y'_{pq}(c) \le y_{pq}(c) + \mathrm{cap}_{pq} \tag{19}$$
$$y'_{qp}(c) \le y_{qp}(c) + \mathrm{cap}_{qp} \tag{20}$$

Proof. An intuitive proof comes from the fact that flows f_{pq} and f_{qp} represent the increase of the balance variables $y_{pq}(c)$ and $y_{qp}(c)$ respectively during UPDATE_DUALS_PRIMALS. Since it is always true that:

$$f_{pq} \le \mathrm{cap}_{pq} \ ,$$
$$f_{qp} \le \mathrm{cap}_{qp} \ ,$$

the lemma then follows directly. □

Lemma A.2 *During a c-iteration, the following entailments hold true:*

$$\mathrm{load}_{pq}(c, \bar{x}_q) \le w_{pq} d(c, \bar{x}_q) \Rightarrow \mathrm{load}'_{pq}(c, \bar{x}_q) \le w_{pq} d(c, \bar{x}_q) \ , \tag{21}$$
$$\mathrm{load}_{pq}(\bar{x}_p, c) \le w_{pq} d(\bar{x}_p, c) \Rightarrow \mathrm{load}'_{pq}(\bar{x}_p, c) \le w_{pq} d(\bar{x}_p, c) \ , \tag{22}$$

where $\bar{\mathbf{x}}$ can be any labeling which is a c-expansion of the primal solution $\mathbf{x}$ at the start of the current c-iteration. (In the above entailments, quantities $\mathrm{load}_{pq}(c, \bar{x}_q)$, $\mathrm{load}_{pq}(\bar{x}_p, c)$ are supposed to have been estimated using the value of the balance variables exactly after PREEDIT_DUALS*).*

Proof. If $\bar{x}_q = c$ then (21) is trivial to prove. We may therefore assume that $\bar{x}_q = x_q \ne c$ (since $\bar{\mathbf{x}}$ is a c-expansion of $\mathbf{x}$). So, in order to prove (21), let us then also assume that:

$$\mathrm{load}_{pq}(c, x_q) \le w_{pq} d(c, x_q) \tag{23}$$

But then, by combining Lemma A.1 with the definition of capacity cap_{pq} in (10), we get:

$$\begin{aligned} y'_{pq}(c) \overset{(19)}{\le} y_{pq}(c) + \mathrm{cap}_{pq} &\overset{(10)}{=} y_{pq}(c) + [w_{pq} d(c, x_q) - \mathrm{load}_{pq}(c, x_q)]^+ \\ &\overset{(23)}{=} y_{pq}(c) + w_{pq} d(c, x_q) - \mathrm{load}_{pq}(c, x_q) \\ &= w_{pq} d(c, x_q) - y_{qp}(x_q) \overset{x_q \ne c}{=} w_{pq} d(c, x_q) - y'_{qp}(x_q) \end{aligned}$$

which thus proves (21). The proof for (22) proceeds similarly. □

Lemma A.3 *At the last c-iteration of the Fast-PD algorithm, the following inequalities hold (for any p, q):*

$$\mathrm{load}'_{pq}(c, x'_q) \leq w_{pq} d_{\max} \tag{24}$$

$$\mathrm{load}'_{pq}(x'_p, c) \leq w_{pq} d_{\max} \tag{25}$$

Proof. The lemma is trivial if either $c = x'_p$ or $c = x'_q$, and so we will hereafter assume that $c \neq x'_p$ and $c \neq x'_q$. Furthermore, since this is the last c-iteration, no label change takes place, and so:

$$x'_p = x_p, \quad x'_q = x_q. \tag{26}$$

CASE 1: If the following two inequalities hold true:

$$\mathrm{load}_{pq}(c, x_q) \leq w_{pq} d(c, x_q) \ , \tag{27}$$

$$\mathrm{load}_{pq}(x_p, c) \leq w_{pq} d(x_p, c) \ , \tag{28}$$

then the lemma follows directly from Lemma A.2.

CASE 2: It thus remains to consider the case where at least one of the inequalities (27), (28) is violated. Then (and only then), PREEDIT_DUALS (by definition) will adjust $y_{pq}(c)$ so that:

$$\mathrm{load}_{pq}(c, x_q) = w_{pq} d(c, x_q) \tag{29}$$

Hence, condition (27) will be restored after the adjustment. We may then assume that (28) will remain violated after the adjustment (or else we would fall back to case 1), i.e. we may assume that:

$$\mathrm{load}_{pq}(x_p, c) > w_{pq} d(x_p, c) \tag{30}$$

Based on (29), (30) and the definition of capacities in (10), (11), it then results that $\mathrm{cap}_{pq} = \mathrm{cap}_{qp} = 0$. This implies that $y'_{pq}(c) = y_{pq}(c)$, and it is then easy to show that:

$$\mathrm{load}'_{pq}(c, x_q) = \mathrm{load}_{pq}(c, x_q) \tag{31}$$

$$\mathrm{load}'_{pq}(x_p, c) = \mathrm{load}_{pq}(x_p, c) \tag{32}$$

But then:

$$\mathrm{load}'_{pq}(c, x_q) \overset{(31)}{=} \mathrm{load}_{pq}(c, x_q) \overset{(29)}{=} w_{pq} d(c, x_q) \leq w_{pq} d_{\max} \tag{33}$$

and also:

$$\mathrm{load}'_{pq}(x_p, c) \overset{(32)}{=} \mathrm{load}_{pq}(x_p, c) = [\mathrm{load}_{pq}(x_p, c) + \mathrm{load}_{pq}(c, x_q)] - \mathrm{load}_{pq}(c, x_q) \tag{34}$$

$$= \mathrm{load}_{pq}(x_p, x_q) - \mathrm{load}_{pq}(c, x_q) \tag{35}$$

$$\overset{(4),(29)}{=} w_{pq} d(x_p, x_q) - w_{pq} d(c, x_q) \leq w_{pq} d_{\max}, \tag{36}$$

with equality (35) being true due to the identity $\mathrm{load}_{pq}(x_p, c) + \mathrm{load}_{pq}(c, x_q) = \mathrm{load}_{pq}(x_p, x_q)$. □

We may now proceed to prove Theorem 2, which (as already mentioned) forms the main goal of this section.

Proof for Theorem 2. To complete the proof of this theorem, we need to show that each one of the complementary slackness conditions (3)-(5) will hold true by the time Fast-PD terminates:

Condition (4)**:** As already explained in section 3, the UPDATE_DUALS_PRIMALS routine can restore condition (4) for most pairs (p, q) during any inner-iteration. However, even if there do exist pairs that violate this condition after UPDATE_DUALS_PRIMALS, then the POSTEDIT_DUALS routine can, by definition, always restore condition (4) for them.

Condition (5)**:** Based on Lemma A.3, it follows that, given any label a, the following inequality will hold true after the last a-iteration:

$$\text{load}_{pq}(a, x_q) \le w_{pq} d_{\max}. \tag{37}$$

Similarly, given any label b, the following inequality will also hold true after the last b-iteration:

$$\text{load}_{pq}(x_p, b) \le w_{pq} d_{\max}. \tag{38}$$

Combining these inequalities with the identity:

$$\text{load}_{pq}(a, b) + \text{load}_{pq}(x_p, x_q) = \text{load}_{pq}(a, x_q) + \text{load}_{pq}(x_p, b), \tag{39}$$

we get that:

$$\begin{aligned}\text{load}_{pq}(a, b) &= [\text{load}_{pq}(a, b) + \text{load}_{pq}(x_p, x_q)] - \text{load}_{pq}(x_p, x_q)\\ &\overset{(39)}{=} [\text{load}_{pq}(a, x_q) + \text{load}_{pq}(x_p, b)] - \text{load}_{pq}(x_p, x_q)\\ &\overset{(37),\,(38)}{\le} 2w_{pq} d_{\max} - \text{load}_{pq}(x_p, x_q),\end{aligned}$$

and then condition (5) follows trivially, since $\text{load}_{pq}(x_p, x_q) = d(x_p, x_q) \ge 0$ by (4).

Condition (3)**:** It turns out that the UPDATE_DUALS_PRIMALS routine can finally ensure condition (3) due to the way that the exterior capacities of graph $\mathcal{G}^c$ are defined. Since Fast-PD uses the same definition as PD3$_a$ for these capacities, the corresponding proof (that has been used for the case of the PD3$_a$ algorithm) in [5] applies here as well. □

Appendix B: Proof of Theorem 3 about the equivalence of Fast-PD and α-expansion in the case of a metric distance function $d(\cdot,\cdot)$

In this section, we will provide the proof for Theorem 3, which shows that when distance $d(\cdot,\cdot)$ is a metric, then Fast-PD can compute exactly the same solution as the α-expansion algorithm. To this end, we will make use of the following two lemmas:

Lemma B.1 *Let us define:*

$$\mathrm{primal}(\mathbf{x}) \equiv \text{MRF energy of labeling } \mathbf{x} \; ,$$

and let also $\mathbf{x}$ *be any primal solution generated during an inner-iteration of the Fast-PD algorithm. It then holds that:*

$$\mathrm{primal}(\mathbf{x}) = \sum_p h_p(x_p) \tag{40}$$

Proof.

$$\begin{aligned}\mathrm{primal}(\mathbf{x}) &\overset{(1)}{=} \sum_p \mathfrak{c}_p(x_p) + \sum_{pq\in\mathcal{E}} w_{pq} d(x_p, x_q)\\ &\overset{(4)}{=} \sum_p \mathfrak{c}_p(x_p) + \sum_{pq\in\mathcal{E}} \mathrm{load}(x_p, x_q) \overset{(7)}{=} \sum_p \mathfrak{c}_p(x_p) + \sum_{pq\in\mathcal{E}} \big(y_{pq}(x_p) + y_{qp}(x_q)\big)\\ &= \sum_p \mathfrak{c}_p(x_p) + \sum_p \sum_{q:pq\in\mathcal{E}} y_{pq}(x_p) = \sum_p \big(\mathfrak{c}_p(x_p) + \sum_{q:pq\in\mathcal{E}} y_{pq}(x_p)\big) \overset{(6)}{=} \sum_p h_p(x_p)\end{aligned}$$

□

Lemma B.2 *Let the distance function* $d(\cdot,\cdot)$ *be a metric. Let* $\mathbf{x}$ *be the primal solution at the start of the current* c*-iteration, and let also* $\bar{\mathbf{x}}$ *be any solution which coincides with a* c*-expansion of solution* $\mathbf{x}$*. It will then hold that:*

$$\mathrm{load}'_{pq}(\bar{x}_p, \bar{x}_q) \le w_{pq} d(\bar{x}_p, \bar{x}_q) \tag{41}$$

Proof. If either $\bar{x}_p = \bar{x}_q = c$ or $\bar{x}_p = x_p$, $\bar{x}_q = x_q$, the lemma is trivial to prove. So let us assume that $\bar{x}_p = x_p$, $\bar{x}_q = c$ (the case $\bar{x}_p = c$, $\bar{x}_q = x_q$ can be handled similarly). In this case, we need to show that:

$$\mathrm{load}'_{pq}(x_p, c) \le w_{pq} d(x_p, c) \tag{42}$$

Due to entailment (22) in Lemma A.2, it then suffices to show that the following condition will hold true after PREEDIT_DUALS:

$$\mathrm{load}_{pq}(x_p, c) \le w_{pq} d(x_p, c). \tag{43}$$

Regarding inequality (43), this will always hold if PREEDIT_DUALS has to apply no adjustment to $y_{pq}(c)$ (this results from the definition of PREEDIT_DUALS). However, even if PREEDIT_DUALS must adjust the value of $y_{pq}(c)$, inequality (43) will still hold true, provided that $d(\cdot,\cdot)$ is a metric.

To see that, it suffices to observe that after the adjustment made by PREEDIT_DUALS, it will then hold:

$$\text{load}_{pq}(c, x_q) = w_{pq} d(c, x_q) \tag{44}$$

and so:

$$\begin{aligned}\text{load}_{pq}(x_p, c) &= [\text{load}_{pq}(x_p, c) + \text{load}_{pq}(c, x_q)] - \text{load}_{pq}(c, x_q) \\ &= \text{load}_{pq}(x_p, x_q) - \text{load}_{pq}(c, x_q) \overset{(4),(44)}{=} w_{pq} d(x_p, x_q) - w_{pq} d(c, x_q) \le w_{pq} d(x_p, c) \ ,\end{aligned}$$

where the last inequality holds due to that $d(\cdot,\cdot)$ is a metric and thus has to satisfy the triangle inequality. □

We may now proceed to the main goal of this section, which is the proof of Theorem 3.

Proof for Theorem 3. Let $\mathbf{x}$ be the primal solution at the start of the current c-iteration, let $\mathbf{x}'$ be the solution selected by Fast-PD at the end of the current c-iteration, and let also $\bar{\mathbf{x}}$ be any solution which coincides with a c-expansion of solution $\mathbf{x}$.

To prove the theorem, we need to show that:

$$\text{primal}(\mathbf{x}') \le \text{primal}(\bar{\mathbf{x}}) \tag{45}$$

To this end, it suffices to show that the following conditions hold true:

$$\text{primal}(\mathbf{x}') = \sum_p h'_p(x'_p) \tag{46}$$

$$\sum_p h'_p(x'_p) \le \sum_p h'_p(\bar{x}_p) \tag{47}$$

$$\sum_p h'_p(\bar{x}_p) \le \text{primal}(\bar{\mathbf{x}}) \tag{48}$$

Regarding equation (46), this follows directly by applying Lemma B.1 to the primal solution $\mathbf{x}'$ generated by the Fast-PD algorithm.

To prove inequality (47), one can first show that $h'_p(x'_p) = \min\{h'_p(x_p), h'_p(c)\}$. In addition to that, it will also hold either $\bar{x}_p = x_p$ or $\bar{x}_p = c$ (since $\bar{\mathbf{x}}$ is a c-expansion of $\mathbf{x}$). By combining these facts, it then results that $h'_p(x'_p) \le h'_p(\bar{x}_p)$, and thus (47) follows directly.

Finally, inequality (46) will hold true because:

$$\begin{aligned}\text{primal}(\bar{\mathbf{x}}) &= \sum_p \mathfrak{c}_p(\bar{x}_p) + \sum_{pq \in \mathcal{E}} w_{pq} d(\bar{x}_p, \bar{x}_q) \overset{(41)}{\ge} \sum_p \mathfrak{c}_p(\bar{x}_p) + \sum_{pq \in \mathcal{E}} \text{load}'(\bar{x}_p, \bar{x}_q) \\ &= \sum_p \mathfrak{c}_p(\bar{x}_p) + \sum_{pq \in \mathcal{E}} \big(y'_{pq}(\bar{x}_p) + y'_{qp}(\bar{x}_q)\big) = \sum_p \mathfrak{c}_p(\bar{x}_p) + \sum_p \sum_{q: pq \in \mathcal{E}} y'_{pq}(\bar{x}_p) \\ &= \sum_p \big(\mathfrak{c}_p(\bar{x}_p) + \sum_{q: pq \in \mathcal{E}} y'_{pq}(\bar{x}_p)\big) \overset{(6)}{=} \sum_p h'_p(\bar{x}_p)\end{aligned}$$

$\square$