Globally Optimal Pixel Labeling Algorithms for Tree Metrics*

Pedro F. Felzenszwalb
University of Chicago
pff@cs.uchicago.edu

Gyula Pap
Cornell University
pap@cs.cornell.edu

Éva Tardos
Cornell University
eva@cs.cornell.edu

Ramin Zabih
Cornell University
rdz@cs.cornell.edu

Abstract

We consider pixel labeling problems where the label set forms a tree, and where the observations are also labels. Such problems arise in feature-space analysis with a very large label set, for instance in color image segmentation. In this case a tree of labels can be constructed via hierarchical clustering of the observations. This leads to an obvious distance function between two labels, namely their distance within the tree; such tree metrics have been extensively studied outside of computer vision [14]. We provide fast algorithms that use graph cuts to exactly minimize the energy function for pixel labeling problems with tree metrics. Our work substantially improves a facility location algorithm of Kolen [18], which is impractical for large label sets $\mathcal{L}$ since it requires $O(|\mathcal{L}|)$ min cuts on large graphs. Our main technical contribution is a new ordering of swap moves that reduces the running time to the equivalent of $O(\log |\mathcal{L}|)$ min cuts; as a result, we can handle realistic-sized color images in a few seconds.

1. Pixel Labeling Problems with Tree Metrics

Pixel labeling problems involve assigning a label from some set $\mathcal{L}$ to each pixel in an image. These problems are naturally phrased in terms of energy minimization, and are closely tied to Markov Random Fields. Energy minimization for pixel labeling has been extensively used in computer vision, especially since the introduction of fast algorithms a decade ago (see [26] for a recent overview). The general energy minimization problem is known to be NP-hard, even for a grid graph [4]. As a result, there has been considerable interest in restricted classes of energy functions where the globally optimal solution can be efficiently computed. In this paper we introduce a new class of problems and provide very efficient algorithms for their solution.

Consider a pixel labeling problem for an image with n pixels and label set $\mathcal{L}$. A solution will be written as $x = (x_1, \ldots, x_n)$ where $x_i \in \mathcal{L}$. Many pixel labeling problems involve minimizing an energy of the form

$$\sum_{i=1}^{n} \lambda_i D(x_i, o_i) + \sum_{(i,j)\in\mathcal{N}} \lambda_{ij} V(x_i, x_j). \qquad (1)$$

Here $o_i \in \mathcal{O}$ is an observation associated with the i-th variable. The data term, $D(x_i, o_i)$, ensures that x_i is consistent with o_i. The neighborhood system $\mathcal{N}$ specifies an undirected graph over the variables. The smoothness term, $V(x_i, x_j)$, ensures that adjacent variables in this graph are given similar labels. The weights λ_i, λ_{ij} specify the relative importance of an individual pixel's observation, and of the smoothness term for each adjacent pair of pixels.

While there are many variants of the pixel labeling problem, (1) is general enough to capture many textbook cases such image denoising. Note that in denoising the space of observations $\mathcal{O}$ is the same as $\mathcal{L}$, since o_i and x_i are both intensities or colors. Moreover, in denoising D and V typically have a similar form, such as $f(a, b) = \min((a - b)^2, const)$. This occurs because when the space of observations is the same as the space of labels, both D and V compare elements in the same space.

Efficiency is a huge challenge, especially when the label space is large. The two most popular modern methods for minimizing such energy functions, graph cuts [4] and loopy belief propagation (LBP) [21], do not generally scale well with $|\mathcal{L}|$. The most popular graph cut algorithm computes at least $|\mathcal{L}|$ min cuts on a graph with approximately n nodes. Discrete LBP requires space (and time) per variable that grows linearly in $|\mathcal{L}|$.

1.1. The energy minimization problem

We consider pixel labeling problems with a particular structure, where the both the labels and the observations are nodes of a tree T. Formally we assume that each node in T corresponds to a label, and that $\mathcal{O} \subseteq \mathcal{L}$. (For example, this situation arises naturally if we use hierarchical clustering on $\mathcal{O}$ to build $\mathcal{L}$.) Such a tree T induces a natural distance on two labels a, b, which is the length of the unique path in T between a and b. We will consider trees with non-negative

*This research has been supported by NSF grants IIS-0746569 and IIS-0803705 and by NIH grants R21EB008138 and P41RR023953. The work was done while P. Felzenszwalb was visiting Cornell University.

edge weights w_e associated with each edge, so the length of a path is the sum of the edge weights along the path. The distance between a and b, which we will write $d_T(a, b)$, is called a tree metric and has been extensively studied in theoretical computer science, as surveyed in [14]. In particular tree metrics can be used to approximate arbitrary metrics.

In a pixel labeling problem with a tree metric, $d_T(a, b)$ is used both to measure the similarity between observation and labels, and the similarity between labels. Thus, the energy function we wish to minimize is

$$E(x_1, \ldots, x_n) = \sum_{i=1}^{n} \lambda_i d_T(x_i, o_i) + \sum_{(i,j)\in\mathcal{N}} \lambda_{ij} d_T(x_i, x_j). \quad (2)$$

Note that while T is a tree, the neighborhood system $\mathcal{N}$ is arbitrary and can include loopy graphs such as grids. The energy E is similar to classical image denoising formulations, in that D and V have identical forms; the key difference is that we ensure tractability by requiring a tree metric. The main contribution of our work is a very fast algorithm to exactly minimize this energy function.

1.2. Application: spatially coherent clustering

To make this problem more concrete, we consider an application of our algorithm for feature space analysis, a popular technique for solving early vision problems (see, for example, [5, 22, 30]). In feature space analysis each pixel is associated with a feature vector, and we will label each pixel with a cluster in feature space.

Our algorithm can be applied to give a fast method for constructing spatially coherent clusterings, which is a problem that several papers have recently addressed [31, 10, 25]. All of these methods rely on iterative techniques without provable error bounds, while our method computes a global minimum solution of an energy function.

If we write the feature vector associated with each pixel p as $v(p)$, then we can build our tree of labels T by performing hierarchical clustering on these vectors. This leads to a binary tree T, where the observations $\mathcal{O}$ (which are the feature vectors $v(p)$) are the leaves of the tree. The internal nodes of T are clusters of feature vectors.

Now, we can apply the labeling algorithm with T as the underlying tree of labels. Distances along T lead to a natural measure of similarity between labels and observations. For example, if a set of pixels S have similar feature vectors then we expect to have a cluster C corresponding to S, and the distance in T from C to each observation in S will be small. In particular, if the pixels in S are neighbors of each other the spatial smoothness terms in the energy function will push them to select the label C over their individual observations.

Figure 1 shows an example where we apply the algorithm for clustering using color information. In this example each pixel has one of 3 different colors: red, green and blue. The input image (left) has 4 regions, pixels in the first region are red, in the second are green and in the third are blue. Pixels in the last region have a random color. The tree T (middle) groups the 3 observed colors in a hierarchical fashion. The final labeling (right) accurately recovers a natural clustering of the pixels. The spatial terms in the energy function gives an advantage towards picking the same (or similar) label for groups of nearby pixels. Pixels in the first 3 regions keep their observed colors because they are spatially coherent. Pixels in the last region pick the "purple" label because that is the "mean" of red, green and blue.

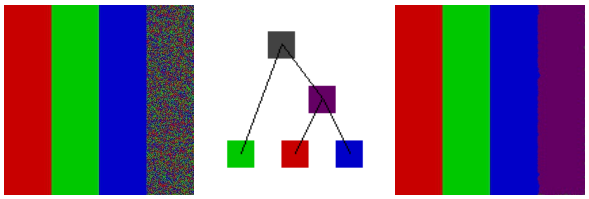

Figure 1. Left: input image, middle: hierarchical clustering of colors, right: optimal labeling.

An interesting property of our method is that even though the purple label is the parent of red and blue we have that some pixels take the purple label while other pixels keep the red and blue labels. Thus the tree-metric labeling algorithm adaptively picks labels for each pixel, without making hard clustering decisions. An area of the image that is very spatially incoherent will naturally be labeled with a cluster high-up in the hierarchy, while an area of the image that is more coherent will be labeled with a cluster lower-down in the hierarchy.

Note that in this application the set of observations $\mathcal{O}$, and thus $\mathcal{L}$, can be very since each pixel can have a different color. This makes it impractical to consider algorithms that require time per pixel that grows linearly in $|\mathcal{L}|$.

1.3. Summary of our results

The key tool in our approach is the swap move [4], one of the basic algorithms in graph cuts. In a swap move, we consider two labels a, b and restrict our attention to pixels that currently have one of these labels. [4] showed that we can efficiently re-allocate the pixels between these two labels for a large class of energy functions. A local minimum with respect to swap moves is a labeling whose energy cannot be decreased by any swap move. [4] showed how to determine the optimal (a, b)-swap move by computing the min cut on a graph with one node per variable. The min cut can be computed by max flow methods [17], which are very fast in practice; we will write the running time to compute max flow on a graph with n nodes as $f(n)$[1], which is a low-

[1] Assuming $\mathcal{N}$ is sparse all graphs considered by our algorithms will be sparse, so we don't include the number of edges as a parameter.

order polynomial. We will rely on a special case of swap moves where the two labels a, b are adjacent in the tree of labels T. We will refer to this as an *adjacent* swap move.

Kolen [18] gave a method to minimize E in the context of a facility location task; he proved that a local minimum of E with respect to adjacent swap moves is a global minimum, and gave an algorithm whose running time is $O(f(n)|\mathcal{L}|)$, which is impractical for large label sets. We provide much faster algorithms for minimizing E by performing the swap moves in a different order; even though we still do $O(|\mathcal{L}|)$ swap moves, most of them are done on small graphs. As a result, our methods have a running time of $O(f(n)\log|\mathcal{L}|)$, and are extremely fast in practice.[2]

Our first algorithm (section 3) achieves this runtime in the important special case of balanced binary trees. This algorithm is straightforward to describe and analyze, as well as easy to implement. Our second algorithm (section 4) handles arbitrary trees using a divide and conquer approach. Experimental results are presented for the first algorithm.

2. Related Work

Since the general pixel labeling problem is NP-hard [4], a great deal of effort has been expended on finding tractable special cases. Perhaps the best known such example is a binary labeling problem that can be exactly solved with graph cuts [12, 11]. Special cases can be quite important in their own right; this binary labeling problem is directly used for important interactive segmentation tasks [2, 24]. Fast exact methods for special cases can also be used to approximately solve NP-hard problems; the same binary labeling problem is the inner loop for the main graph cut algorithms of [4].

Our divide and conquer approach is related to [19]. However, the method in [19] is a heuristic with no guarantees about the quality of the solution it finds. The key to our result is that for pixel labeling with tree metrics it is possible to set up a binary labeling problem to decide which of two sets contains the optimal label for each variable.

2.1. Polynomial time algorithms for pixel labeling

Another fast exact algorithm for graph cuts was originally proposed in [16], and then generalized by [15]. This method handles more than 2 labels, but restricts the form of V. The construction has been used by other graph cut algorithms as well [28, 7].

[13] gives an algorithm that has logarithmic dependency on $|\mathcal{L}|$ for certain kinds of problems. Within our setting their algorithm applies when T is a path. In this sense we generalize their results to arbitrary trees. We also note that we use standard max-flow where [13] uses parametric max-flow, which requires more complex algorithms.

[2]This bound assumes $|\mathcal{L}|$ is not too big such as when $|\mathcal{L}| \leq f(n)$. More generally this running time bound holds if $|\mathcal{L}|$ is $O(f(n))$.

If the neighborhood system $\mathcal{N}$ forms a tree we can use dynamic programming or belief propagation to solve the problem, independent of the form of D or V [23]. The runtime of these methods is $O(n|\mathcal{L}|^2)$ in general. In some cases, such as [8, 9] dynamic programming can be made even faster, but its still $O(n|\mathcal{L}|)$.

2.2. Kolen's results on facility location in trees

Our work builds on the results of Kolen [18], who gave a polynomial time algorithm to minimize E in a very different context, namely facility location on a tree. Kolen considered the "p-median problem with mutual communication", which in our notation is to minimize E from equation (2) where the neighborhood system $\mathcal{N}$ is the complete graph. While his motivating problem used a complete graph, his algorithm can handle an arbitrary neighborhood system by setting some of the λ_{ij} terms to 0.

Kolen proved that a local minimum with respect to adjacent swap moves is also a global minimum.

Theorem 2.1 *(Kolen's optimality theorem [18]) If x^* minimizes E with respect to (a, b)-swap moves for all labels a, b that are adjacent in T, then x^* globally minimizes E.*

Kolen proposed a simple algorithm to compute x^* by a series of adjacent swap moves. His algorithm begins assigning every variable x_i the label a, which is an arbitrary leaf in T whose parent is b. He then does an (a, b)-swap move. Kolen showed that this swap move provides information about the global optimum: any variable that kept the label a after this swap move has that label in the global optimum, while any variable that adopted the label b does not have the label a.

This procedure can the be done recursively, ignoring the variables that are known to be labeled with a and removing a from the tree of labels T

The running time of Kolen's algorithm unfortunately is $O(f(n)|\mathcal{L}|)$, which is impractical for large label sets. In the worst case, after each adjacent (a, b)-swap move every variable adopts the new label b, which means that the underlying graph never shrinks.

3. The Sweep Algorithm

We now describe our sweep algorithm. We initialize the algorithm by picking a root label, and assigning that label to every variable. We perform a sequence of adjacent swap moves, one for each edge in the tree of labels T. The order of the swaps is such that we always perform an adjacent (a, b)-swap move where the label a has been considered before but b has not. In this case we are simply relabeling some variables that currently have label a with label b. The algorithm is naturally performed by considering the labels in T in depth-first order starting at the root label.

3.1. Proof of correctness

When we perform an adjacent (a, b)-swap move, we can think of the tree of labels T as being divided into two trees by removing the (a, b) edge; we will write these trees as T_a, T_b, where $a \in T_a, b \in T_b$. We say two labelings x, y are *consistent* with respect to (a, b) if x_i and y_i are both in the same side of the edge (a, b) for all i. That is, if $x_i \in T_a \iff y_i \in T_a$ for all i, and similarly for T_b.

The main lemma we need states informally that if two labelings are consistent with respect to (a, b) then relabeling some pixels from b to a (or vice-versa) results in the same change in E, whether we start with x or with y.

Lemma 3.1 *Let x, y be two labelings that are consistent with respect to the adjacent labels (a, b), and let B be an arbitrary subset of variables that are given label b by both x and y. Let Δx be the change in E that results if we start at x and give all variables in B the label a, and let Δy be the change that results if we start at y. Then $\Delta x = \Delta y$.*

PROOF: We will show that the claimed equality holds actually for every term in E, instead of just for E itself.

First, consider a term of the form $\lambda_i d_T(o_i, x_i)$. If the ith variable is in B the term changes in the same way in both labelings. If the variable is not in B the term is unchanged in both labelings.

Second, consider a term of the form $\lambda_{ij} d_T(x_i, x_j)$, and its contribution to Δx. If neither variable is in B there is no contribution; this is also the case if both variables are in B (since they both had the label b, then both acquire the label a). In both cases there is also no contribution to Δy. Now, w.l.o.g., suppose the first variable is not in B while the second variable is. The contribution to Δx is $\lambda_{ij}(d_T(x_i, a) - d_T(x_i, b))$, and the contribution to Δy is $\lambda_{ij}(d_T(y_i, a) - d_T(y_i, b))$. Since a and b are neighbors in T, these quantities are $\pm\lambda_{ij} w_{ab}$, depending on whether x_i and y_i are closer to a or b. They are closer to a if they are in T_a and closer to b if they are in T_b. Either way, since the labelings are consistent the contributions are the same. ■

We prove correctness of the sweep algorithm based on this lemma. By Kolen's optimality theorem, all we need to show is that the output of our algorithm is optimal with respect to adjacent swap moves.

Let x^* be the output of the sweep algorithm.

Theorem 3.2 *The energy of x^* cannot be decreased by performing an adjacent swap move.*

PROOF: Suppose an adjacent swap move (a, b) would decrease the energy of x^*. Our algorithm performed an optimal (a, b)-swap move at some point, and let us write the labeling we obtained after this swap move as $\hat{x}$. Observe that x^* and $\hat{x}$ are consistent with respect to (a, b), since on any succeeding swap move the pixels labeled with a in $\hat{x}$ can only get a label in T_a, and similarly for b. Note also that any variable labeled a or b in x^* must have the same label in $\hat{x}$, since our swap moves will only relabel a variable with a label that hasn't been considered before. So by Lemma 3.1, the (a, b)-swap move that would decrease the energy of x^* would also decrease the energy of $\hat{x}$. This contradicts the construction of $\hat{x}$ by an optimal (a, b)-swap. ■

3.2. Running time

At first glance it appears that the running time of our algorithm is $O(f(n)|\mathcal{L}|)$, since we are doing one min cut per edge in T. However, a more careful analysis reveals that the min cuts are being computed on smaller and smaller graphs. Now we will show that our running time is actually $O(|\mathcal{L}| + f(n)k\Delta)$, where k is the depth of T and Δ is the maximum degree.[3] For any balanced tree of bounded degree this will give us a running time of $O(|\mathcal{L}| + f(n)\log|\mathcal{L}|)$.

Let G be the graph defined by the neighborhood system $\mathcal{N}$ on the variables. Note that when we do an (a, b)-swap we are simply relabeling some nodes currently labeled a with b. Thus we can concentrate on the subgraph on nodes A that currently have label a. Some care must be taken due to nodes in A that have neighbors outside of A. We can replace them by a single node that is forced to take label a without changing the result of the (a, b)-swap. In particular this means that the time it takes to do the min cut computation needed for the (a, b)-swap is $f(|A|)$.[4]

We also need to make sure we can set up the min cut problem efficiently. Nodes in A can have arbitrary observations. Consider a node v with observation c. The difference between the assignment cost when labeling this node a or b is exactly $\lambda_v w_{ab}$. The assignment cost of label a is smaller by $\lambda_v w_{ab}$, if c is in the subtree T_a, and the assignment cost of label b is smaller by $\lambda_v w_{ab}$, if c is in the subtree T_b. Hence we will connect v to either s or t with an edge of cost $\lambda_v w_{ab}$ depending on which subtree the label c is in.

To be able to set up the min cut problem efficiently, we preprocess the tree T using Depth-First-Search, allowing us to set up edges corresponding to assignment costs using a simple lookup in $O(1)$ time.

Theorem 3.3 *Using DFS to preprocess the label tree T in linear time, for any edge (a, b) of T, and any label c, we can decide in $O(1)$ time if c is in the subtree T_a or T_b.*

PROOF: Starting at the root label, we relabel the nodes in T by a DFS walk, i.e., the root is labeled 1, its first child is labeled 2, etc. For each node a we record the first and

[3] For a rooted tree we define the degree of a node as the number of children it has.

[4] The graph has 3 extra nodes, s, t and a node that was added to account for nodes that are not in A, but this does not affect the overall runtime.

last visit made to the node during the DFS. We call these $f(a)$ and $\ell(a)$. For example, the root r has $f(r) = 1$ and $\ell(r) = |\mathcal{L}|$, as the DFS visits the root again after visiting all tree nodes. Note that DFS visits the subtree rooted at a right after visiting a for any node a. Therefore, a label c is in the subtree rooted at a if and only if $f(a) \leq f(c) \leq \ell(a)$, which allows us to decide if c is in the subtree rooted at a in $O(1)$ time. Finally, notice that for an edge (a, b) if a is the parent of b, then T_b is exactly the subtree rooted at b. ■

To complete the runtime analysis we consider T one level at a time, and show that the work done to perform all min cuts in each level is bounded by $O(f(n))$.

For simplicity, let T be a binary tree whose root is the label a with two children b, c, where we perform an (a, b)-swap followed by an (a, c)-swap. We charge the work done for these two swaps to label a. Note that both (a, b) and (a, c) swaps are on graphs with at most n nodes. Usually the second swap is on a smaller graph, unless no nodes take label b. Even in the worst case the amount of work charged to a is at most $2f(n)$.

Let n_b be the number of nodes that adopt the label b and n_c be the number of nodes that adopt the label c. The two swaps charged to b will take at most $2f(n_b)$ work, while the swaps charged to c will take at most $2f(n_c)$ work. The key to the analysis is that while one of n_b or n_c might be large, $n_b + n_c \leq n$. Because the running time of min cut $f(n)$ is superlinear and monotonic, $f(n_b) + f(n_c) \leq f(n)$, so the total amount of work charged to *both* b and c is bounded by $2f(n)$. This same reasoning applies to each level of T. At each level below the root the total work charged to all nodes in that level is bounded by $2f(n)$.

This shows that for a binary tree of depth k the running time of all min cuts is $O(f(n)k)$. The same argument easily generalizes to any tree of bounded degree. For a tree of max degree Δ the running time for all min-cuts is $O(f(n)k\Delta)$ as we perform Δ swaps at every internal node of the tree.

4. The Divide-and-Conquer Algorithm

The sweep algorithm of the previous section works for arbitrary trees, but the running time has an $O(f(n)k\Delta)$ factor where k is the depth of the tree, and Δ is the maximum degree in the tree. For balanced binary trees k is $O(\log |\mathcal{L}|)$ and $\Delta = 2$, but both k and Δ can grow linearly with $|\mathcal{L}|$ for trees that are not balanced or binary.

In this section we give a divide-and-conquer algorithm that runs in $O((|\mathcal{L}|+f(n)) \log |\mathcal{L}|)$ time with arbitrary trees. We will do this in two steps. First we show that we can assume with no loss of generality that the tree is binary, and next we use divide and conquer to adapt the algorithm for balanced trees to the more general case.

4.1. Making the tree of labels binary

We solve the labeling problem for arbitrary trees by a reduction to binary trees. Here we assume T is rooted at an arbitrary label. Consider a tree T that is not binary, and for each node a with d children add a binary subtree with d leaves. For a node a with d children, this procedure adds $d - 2$ new nodes, which we will call $a^1, \ldots, a^{d-2}$. We set the weights of newly created edges to 0. Let T' denote the resulting tree, and $\mathcal{L}'$ denote the expanded label set. The resulting tree T' is binary, and all distances between labels in $\mathcal{L}$ remain the same as before. We will apply the sweep algorithm to the new tree T'.

Using the sweep algorithm with T' creates an optimal solution, however, the resulting solution may also use the newly added labels $\mathcal{L}'$. If the labeling happens to only use the original labels $\mathcal{L}$, this solution is optimal. If some of the newly added labels are also used, then as a final step of the algorithm, we replace a new label a^k by the corresponding original label a. Note that this change in labeling does not change the energy value of the solution, as the distance $d_{T'}(a, a^k) = 0$, so it creates an alternate optimal solution only using the original labels $\mathcal{L}$.

4.2. Re-balancing unbalanced trees

By the previous subsection, we can assume that the label tree T is binary, but it may not be a balanced binary tree. Rather than doing a continuous sweep of the tree, we will process edges so as to create significantly smaller subtrees in each step. For example, in a path we start by processing the central edge, and then process the central edges of the two subpaths created. It is well known that binary trees have an edge that creates two roughly equal halves [27].

Lemma 4.1 *For any binary tree with n nodes there is an edge (a, b) such that the two subtrees created by deleting this edge have size at most $3n/4$.*

The algorithm is based on a recursive divide and conquer procedure. Let G be the graph defined by the neighborhood system $\mathcal{N}$ on the variables. We find a balanced edge (a, b) of T and temporarily label all nodes in G by a and perform an (a, b)-swap move. The goal of the swap move is to create two smaller subproblems. Let T_a and T_b denote the two subtrees created by removing edge (a, b) from T, with $a \in T_a$ and $b \in T_b$. Let A denote the nodes labeled a and B denote the nodes labeled b. We create two independent subproblems on two disjoint subgraphs G_a on A and G_b on B with label sets T_a and T_b, respectively. As before, some care must be taken due to nodes in A that have neighbors in B. All those neighbors will get labels from T_b so we can replace them by a single node that is forced to take label a without changing the minimal energy solution for G_a. Analogously for G_b we add a single new node replacing all

neighbors that nodes in B have in A, and force the new node to take label b.[5]

The algorithm consists of repeatedly applying the above step of a swap move, and creating two subproblems. Note that unlike the previous algorithm, we do not think of the nodes in G_a as labeled a or the nodes in G_b as labeled b, we think of them taking a label from T_a and T_b respectively.

We prove correctness of this algorithm we need to show is its output x^* is optimal.

Theorem 4.2 *The energy of x^* is optimal.*

PROOF: We proceed by induction on the size of the subtrees. The base case are trivial problems where we only have a single possible label. By the induction hypothesis, the algorithm finds the lowest energy labeling where all labels in the subgraphs are selected from the corresponding subtrees. We need to establish that there is a globally optimal labeling with this property. We can do this by reduction to the correctness of the sweep algorithm. Consider a run of the sweep algorithm that starts with the edge (a, b). Note that if we continue to run the sweep algorithm all the nodes in A will eventually get labels in T_a and all the nodes in B will get labels in T_b. This establishes that there is an optimal solution that is formed by combining optimal solutions to the subproblems. ∎

4.3. Running time

As a first step of the algorithm, we need to replace the original label tree T with a binary tree T'. Note that T has at most $|\mathcal{L}|$ leaves and T and T' have the same set of leaves. A binary tree with k leaves has less than $2k$ nodes, so T' has size at most $2|\mathcal{L}|$.

As we have done for the sweep algorithm, we use DFS to preprocess the tree, to allow us $O(1)$ time lookup which side label c falls of an edge (a, b).

Now consider one step of the algorithm, for a graph $\hat{G}$ with $\hat{n}$ nodes taking labels in $\hat{T}$. We need to select a central edge (a, b) of $\hat{T}$, whose existence is stated in Lemma 4.1. Finding this edge can be easily done in $O(|\hat{T}|)$ time. We temporarily label all nodes in $\hat{G}$ as a, and set up the min cut computations for an (a, b)-swap. This takes $O(\hat{n})$ time using the DFS data structure. We then solve the min cut problem in $f(\hat{n})$ time, and finally we prepare the subproblems, adding the two new nodes as required using $O(\hat{n})$ time. Note that $f(n) > n$, so the total running time for one step is $O(|\hat{T}| + f(\hat{n}))$.

The number of recursive levels will be $O(\log |\mathcal{L}|)$ since each iteration decreases the size of the label trees by at least a factor of 4/3.

[5]This process replaces an original edge by two edges connected to special nodes with forced labels. This will at most double the total number of edges over time because we never need to connect special nodes together.

As was done for the sweep algorithm, we will consider the total running time at each recursive level combined. At a given recursive level, we have a number of subgraphs and subtrees to consider. The sum of the sizes of the subtrees adds up exactly to the size of the original tree T', so the part of the running time proportional to the size of the tree $\hat{T}$ will add up to $O(|\mathcal{L}|)$. Similarly the different subgraphs at one level are disjoint, and their sizes add up to at most $2n$ (due to the extra nodes added). So the total running time at one level of recursion is $O(|\mathcal{L}| + f(n))$. Since there are $O(\log |\mathcal{L}|)$ levels we get a total runtime of $O((|\mathcal{L}| + f(n)) \log |\mathcal{L}|)$.

5. Spatially coherent clusterings

Now we consider an application of the sweep algorithm for feature space analysis. Suppose we have an image I with pixels P and a feature vector $v(p)$ associated with each pixel $p \in P$. We would like to cluster the pixels using the information captured by the feature vectors. Classical methods cluster the feature vectors and label each pixel p according to the cluster that was assigned to $v(p)$. Our approach gives a method for constructing spatially coherent clusterings.

In this application the observations are the feature vectors associated with each pixel

$$\mathcal{O} = \{v(p) \mid p \in P\}.$$

We perform hierarchical clustering on $\mathcal{O}$ to get a tree of labels T. We then apply the sweep algorithm to label each pixel with a cluster in T. This leads to a spatially coherent labeling. Note that T is typically very large since each pixel can lead to a unique observation (these are the leafs in the clustering). In the experiments shown here $v(p)$ is a 3-dimensional vector of RGB color values, and we often get label sets with over 50,000 labels.

Figure 1 showed an example where there are only 3 observed colors: red, green and blue. Pixels in the first 3 columns of the image get labeled with their observation because those regions are coherent. Pixels in the fourth column get the "purple" label because that label is the closest, in terms of d_T, to the observations in that region.

For the experiments shown here we used a simple agglomerative clustering heuristic to generate T. The method is motivated by the classical agglomerative clustering algorithm based on Ward's variance criteria [29]. Usually one would repeatedly merge the "closest" pair of clusters, where the distance between clusters C and D reflects the increase in variance of the feature vectors in $C \cup D$ relative to the variance of C and D. However, the classical algorithm is too slow when there are a lot of initial clusters. Our method works in phases: in each phase we have a current set of top-level clusters that can be merged. We use a nearest-neighbor data structure [1] to find the k nearest neighbors

of each cluster, where distance is measured in terms of the Euclidean distance between the mean feature vector in each cluster. This leads to a set of candidate pairs (C, D) to be merged. We sort the pairs according to Ward's criteria and greedily merge a fraction of the pairs before moving on to the next phase.

This method fast because it relies on fast nearest neighbor computation. It also generates a balanced binary tree because the depth of the tree is bounded by the number of phases, and each phase reduces the number of top-level clusters by a significant factor.

Figure 2 shows some examples of the outputs we obtain on natural images. Note that our algorithm does not generate a full-fledged image segmentation. The output could be used to generate superpixels or as a preprocessing step for segmentation (similar for example to mean shift filtering [6]). For these examples we used $\lambda_i = 1$ and $\lambda_{ij} = 2$, independent of i and j. We used the standard 4-connected neighborhood system for $\mathcal{N}$.

Figures 3 and 4 show the effect of the pairwise strengths λ_{ij}. Greater values for λ_{ij} lead to solutions that are more spatially coherent. This can be clearly seen in the dock next to the boat in Figure 3, where multiple planks are visible with $\lambda_{ij} = 1$ but are merged when $\lambda_{ij} = 3$. Details such as the red numbers in the price tags in Figure 4 are smoothed out with $\lambda_{ij} = 3$. Note that the effect is local as pixels in other areas of that image are labeled with red colors.

We implemented the sweep algorithm and the hierarchical clustering method in C++. We used the publicly available ANN library [1] for nearest neighbor computations and the code from [3] for computing swap moves. The experiments were done on a 2.13 GHz Intel Core 2 Duo computer running Mac OS X 10.5 using the GNU C compiler. The running time to process the images in Figure 2 was between 5 and 10 seconds per image. A significant part of the time is due to hierarchical clustering. All the images have 481 by 321 pixels and the number of colors in each image ranges from about 20,000 to 60,000. For comparison this is approximately the same time it takes to run mean shift [6] under usual parameter settings.

6. Conclusions

We have described two new algorithms for finding exact solutions for a class of pixel labeling problems defined in terms of tree metrics. These algorithms are asymptotically much faster than Kolen's method, since they perform the equivalent of $O(\log |\mathcal{L}|)$ min cuts instead of $O(|\mathcal{L}|)$. Our sweep algorithm is also very fast in practice, and can be used for feature space analysis of color images in a few seconds, even with tens of thousands of labels.

There are very few exact algorithms for solving pixel labeling problems. While pixel labeling with tree metrics has not been previous studied in computer vision, we are hopeful that our results will lead researchers to look for new applications. In particular, researchers have typically only considered problems with relatively small label sets due to computational restrictions. Our algorithms make it possible to consider new applications that may require very large label sets. We also hope that our algorithms might be useful as a subroutine to solve more general problems.

Figure 3. Results with $\lambda_{ij} = 1$ (middle) and $\lambda_{ij} = 3$ (right). Higher values for λ_{ij} lead to more spatial coherence in the final labeling. Note how more gaps in the dock are filled in on the image on the right hand side.

Figure 4. Results with $\lambda_{ij} = 1$ (middle) and $\lambda_{ij} = 3$ (right). Higher values for λ_{ij} lead to more spatial coherence in the final labeling. This causes some of the details such as the red numbers in the price tags to "disappear" at large values of λ_{ij} even though many pixels are labeled red in other areas of the image.

References

[1] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *JACM*, 45(6):891–923, 1998.

[2] Y. Boykov and M.-P. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in N-D images. In *ICCV*, pages I: 105–112, 2001.

[3] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *PAMI*, 26(9):1124–1137, 2004.

[4] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *PAMI*, 23(11):1222–1239, 2001.

[5] C. Carson, S. Belongie, H. Greenspan, and J. Malik. Blobworld: Image segmentation using expectation-maximization

Figure 2. Some images from the Berkeley segmentation dataset [20] (first row) and the result of our method (second row). Note that our output is not a full-fledged segmentation. It could be used to generate superpixels or as a preprocessing step for segmentation.

and its application to image querying. *PAMI*, 24(8):1026–1038, 2002.

[6] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *PAMI*, 24(5):603–619, 2002.

[7] A. Delong and Y. Boykov. Globally optimal segmentation of multi-region objects. In *ICCV*, pages 1–8, 2009.

[8] P. F. Felzenszwalb and D. P. Huttenlocher. Pictorial structures for object recognition. *IJCV*, 61(1), 2005.

[9] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. *IJCV*, 70(1), 2006.

[10] M. Figueiredo, D. Cheng, and V. Murino. Clustering under prior knowledge with application to image segmentation. In *NIPS*, 2007.

[11] D. Greig, B. Porteous, and A. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society, Series B*, 51(2):271–279, 1989.

[12] P. Hammer. Some network flow problems solved with pseudo-boolean programming. *OR*, 13:388–399, 1965.

[13] D. S. Hochbaum. An efficient algorithm for image segmentation, Markov Random Fields and related problems. *JACM*, 48(4):686–701, 2001.

[14] P. Indyk. Algorithmic aspects of geometric embeddings. Tutorial presented at FOCS, 2001.

[15] H. Ishikawa. Exact optimization for Markov Random Fields with convex priors. *PAMI*, 25(10):1333–1336, 2003.

[16] H. Ishikawa and D. Geiger. Segmentation by grouping junctions. In *CVPR*, pages 125–131, 1998.

[17] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2005.

[18] A. J. W. Kolen. *Tree Network and Planar Rectilinear Location Theory*, volume 25 of *CWI Tracts*. CWI, 1986.

[19] V. Lempitsky, C. Rother, S. Roth, and A. Blake. Fusion moves for Markov Random Field optimization. *PAMI*, 2010.

[20] D. Martin, C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *PAMI*, pages 530–549, 2004.

[21] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: an empirical study. In *UAI*, 1999.

[22] D. Murray and B. Buxton. Scene segmentation from visual motion using global optimization. *PAMI*, 9(2), 1987.

[23] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[24] C. Rother, V. Kolmogorov, and A. Blake. "GrabCut" - interactive foreground extraction using iterated graph cuts. *SIGGRAPH*, 23(3):309–314, 2004.

[25] G. Sfikas, C. Nikou, and N. Galatsanos. Edge preserving spatially varying mixtures for image segmentation. In *CVPR*, pages 1–7, 2008.

[26] 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. *PAMI*, 30(6):1068–1080, 2008.

[27] L. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Comput.*, 30(2):135–140, 1981.

[28] O. Veksler. Graph cut based optimization for MRFs with truncated convex priors. In *CVPR*, pages 1–8, 2007.

[29] J. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[30] Y. Weiss and E. H. Adelson. A unified mixture framework for motion segmentation: Incorporating spatial coherence and estimating the number of models. In *CVPR*, pages 321–326, 1996.

[31] R. Zabih and V. Kolmogorov. Spatially coherent clustering using graph cuts. In *CVPR*, pages II: 437–444, 2004.