Minimizing Energy Functions on 4-connected Lattices using Elimination

Peter Carr Richard Hartley
Australian National University and NICTA
Canberra, Australia

Abstract

We describe an energy minimization algorithm for functions defined on 4-connected lattices, of the type usually encountered in problems involving images. Such functions are often minimized using graph-cuts/max-flow, but this method is only applicable to submodular problems. In this paper, we describe an algorithm that will solve any binary problem, irrespective of whether it is submodular or not, and for multilabel problems we use alpha-expansion. The method is based on the elimination algorithm, which eliminates nodes from the graph until the remaining function is submodular. It can then be solved using max-flow. Values of eliminated variables are recovered using back-substitution. We compare the algorithm's performance against alternative methods for solving non-submodular problems, with favourable results.

1. Introduction

Most applications of energy minimization on Markov Random Fields (MRFs) in Vision involve an MRF defined on a 4-connected lattice derived from an image, but few algorithms use this structure specifically to find a minimum. The most popular general method of solving these problems is max-flow. However, this algorithm, which finds an optimal partition of a graph, can only be used on submodular problems [7]. We concentrate on binary (pseudo-boolean) problems. Even with this restriction, minimization of non-submodular functions is NP-hard, so we can not expect a workable optimal algorithm. However, we present an algorithm that defaults to max-flow for submodular problems, but gracefully degrades from optimality when the problem is not submodular. Since the most common source of non-submodular problems in vision is from multilabel problems, using alpha-expansion, we test the algorithm on multilabel problems, and compare it with other available algorithms, notably Fast PD [8, 9] and tree-reweighted message passing [5]. In most cases, our algorithm is either significantly faster, or achieves better results, or both.

The basis for our method is the elimination algorithm, a specialization of the Junction-tree algorithm. Although this method is useful on "thin" graphs, such as k-trees [1], it is not generally thought to be useful on graphs arising from images, because of their very large complexity. We show in this paper that along with a heuristic to limit the growth of complexity through an approximation (the mean-field approximation) the elimination algorithm is useful and gives very good results in practice.

Previous work. Since the problem is inherently intractable, all methods for solving non-submodular pseudo-boolean problems are of necessity heuristic. The best reference for algorithms that address this problem is [1]. Of particular note are the roof-dual methods, which have been applied to Vision by [11]. However, the reader is referred to [1] for a survey of many other methods, which it would be superfluous to repeat here.

For multilabel problems, notable approaches include alpha-expansion and alpha-beta swap [2] which reduce the problem to a sequence of binary problems. Alpha-expansion is better in practice, but is not always applicable. In this paper, we show how it may be adapted to be generally applicable to all problems, at least those defined on a 4-connected MRF grid. Other methods, such as Fast PD [8], and message-passing algorithms, such as tree-reweighted message passing [5], work directly on multilabel problems. We will compare their methods with ours.

As we will explain in the next section, our algorithm iteratively removes alternating vertices from a lattice. Szeliski [12] employed a similar strategy for preconditioning conjugate gradient optimization of continuous quadratic cost functions. Unlike our least squares approach, he modelled the edge weights as spring stiffness coefficients and computed new coefficients for the non-eliminated springs which best approximated the dynamics of the original problem.

2. The Elimination Algorithm

The elimination algorithm (the "Basic Algorithm" of [1]) gives a closed form method of minimizing any pseudo-boolean cost function without restriction. (In fact, it is also applicable to any multilabel cost function, but we con-

centrate on pseudo-boolean functions.) Of course, since this minimization problem is NP-hard, the elimination algorithm will run in non-polynomial time in general cases, but there are instances when it may be quite efficient. Nevertheless, for full 4-connected lattices, such as those associated with images, the complexity of elimination grows rapidly to a point where full elimination is not possible. In this paper, we explore the idea of limiting the growth in complexity during elimination, resulting in a linear-time algorithm, which gives an approximation of the true cost functions encountered as elimination progresses.

The elimination algorithm. $\mathcal{B}$ denotes the set $\{0,1\}$ and $\mathbb{R}$ the real numbers. A function $f : \mathcal{B}^{n+1} \to \mathbb{R}$ is called a pseudo-boolean function. Our task is to find the minimum of such an f. Writing $f(\mathbf{x}) = f(x_0, x_1, \ldots, x_n)$ we may break up this minimization task as

$$\begin{aligned} \min_{x_0,\ldots,x_n} f(x_0,\ldots,x_n) &= \min_{x_1,\ldots,x_n} \min_{x_0} f(x_0,\ldots,x_n) \\ &= \min_{x_1,\ldots,x_n} f(x_0^*, x_1,\ldots,x_n) \\ &= \min_{x_1,\ldots,x_n} f_1(x_1,\ldots,x_n) \end{aligned}$$

where x_0^* represents the value of x_0 at the optimal solution. To explain the last line, we observe that the optimal value x_0^* depends on the values of the other variables. In short, x_0^* is itself a function of $x_1, \ldots, x_n$. If we can specify this function, then we reduce our problem to minimizing the new function $f_1(x_1, \ldots, x_n)$. Continuing in this way, we ultimately arrive at a function of just one variable, which we may trivially minimize. This is the forward sweep of the algorithm.

In the second (back-substitution) phase of the algorithm, we compute the optimal values of each variable in reverse order. Thus, in the final step, knowing the values of the optimal choice of the variables $(x_1^*, \ldots, x_n^*)$, we may compute x_0^*. The process is very much analogous to the method of solving a set of linear equations by Gaussian elimination.

We now become more specific. Any pseudo-boolean function may be written as a polynomial in the variables $x_0, \ldots, x_n$ [1]. For pseudo-boolean functions arising from MRFs, the degree of this polynomial is equal to the maximum clique size. It is not difficult to show [1] that the minimization of a pseudo-boolean function of any degree may be reduced to the minimization of a quadratic function, though this involves introduction of auxiliary variables. Later, we will specifically consider quadratic pseudo-boolean functions, but for now we continue to address the general case.

Given a function $f(x_0, \ldots, x_n)$, we may separate out the terms involving x_0 and factor to write

$$f(x_0, x_1, \ldots, x_n) = g(x_1, \ldots, x_n) + x_0 \Delta_0(x_1, \ldots, x_n)\,.$$

(Note that since $x_i^2 = x_i$ for a boolean variable, there are no higher order terms in x_0.) The function $\Delta_0(x_1, \ldots, x_n)$ is called the derivative of f with respect to x_0. Suppose the values of variables $x_1, \ldots, x_n$ are known. Then to minimize $f(x_0, \ldots, x_n)$ we need to choose $x_0^* = 0$ if $\Delta_0(x_1, \ldots, x_n) > 0$ and $x_0^* = 1$ if $\Delta_0(x_1, \ldots, x_n) < 0$. If $\Delta_0 = 0$, then the choice of x_0^* is arbitrary. We proceed as follows.

1. Enumerate all possible assignments to the variables $x_1, \ldots, x_n$ and for each one compute the numerical value of $\Delta_0(x_1, \ldots, x_n)$. In the worst case, this involves the enumeration of 2^n different cases.

2. Let $x_0^* = 1$ if $\Delta_0(x_1, \ldots, x_n) < 0$ and $x_0^* = 0$ otherwise. Then, compute $x_0^* \Delta_0(x_1, \ldots, x_n)$ as a function of the variables $x_1, \ldots, x_n$. This function is conveniently expressed in tabular form. One may observe at this point that $x_0^* \Delta_0(x_1, \ldots, x_n)$ takes only non-positive values. The importance of this observation will become clearer later.

3. Replace $x_0^* \Delta_0(x_1, \ldots, x_n)$ in the original function f to obtain the new function $f_1(x_1, \ldots, x_n)$.

Elimination in MRFs. In the energy function arising from an MRF with small neighbourhood structure, such as a 4-connected lattice representing an image, the complexity of the elimination process is greatly reduced, at least at the initial stage. We restrict our attention to such a 4-connected lattice. In this case, the energy $f(\mathbf{x})$ may be written as a quadratic polynomial of the form

$$f(x_0, x_1, \ldots, x_n) = \sum_i a_i x_i + \sum_{i,j \in \mathcal{N}} b_{ij} x_i x_j \qquad (1)$$

where $\mathcal{N}$ is a set of all unordered pairs of "neighbouring" variables. A variable x_i associated with a vertex in a 4-connected lattice appears in at most 5 terms, and we may write the terms involving x_i as

$$x_i \Delta_i(x_u, x_r, x_d, x_\ell) = x_i(a_i + b_{iu} x_u + b_{ir} x_r + b_{id} x_d + b_{i\ell} x_\ell),$$

where u, r, d and ℓ stand for "up", "right", "down" and "left". Hence Δ_i involves only four variables. It is easy to enumerate the 16 values corresponding to the possible values of $x_u, \ldots, x_\ell$. (With a little thought one may see that this can be done with only 15 addition operations.)

Unfortunately, the function $x_i^* \Delta_i(x_u, \ldots, x_\ell)$ may be a complicated function of the four variables, and in general it is not quadratic. We consider the "co-occurrence graph" of a pseudo-boolean function which is defined to be a graph with one vertex per variable, and an edge between vertices if and only if the two variables occur together in some term of the polynomial representation of the pseudo-boolean function. In the case we are considering, the co-occurrence

graph of the original function f is a 4-connected lattice. When we eliminate a variable x_i in the interior of the lattice, the resulting function $x_i^* \Delta_i(x_u, \ldots, x_\ell)$ may in general become a complete graph on the four vertices $x_u, \ldots, x_\ell$. By eliminating the variable x_i, we have introduced a 4-clique in the neighbouring four vertices. See Fig 1. The function we are trying to minimize thus has become more complicated.

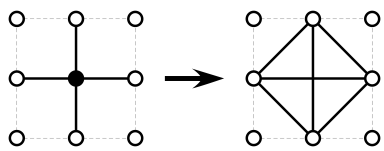


Figure 1. *Eliminating a node introduces a 4-clique into the co-occurence lattice.*

As we proceed with the elimination algorithm, we will ultimately be required to eliminate highly connected vertices in the co-occurrence graph. Elimination of a vertex with k neighbours in the co-occurrence graph leads in the worst case to the creation of a fully connected graph on the k neighbours – a k-clique. As elimination proceeds, the size of the cliques will increase to a point where it becomes impossible to enumerate the Δ_i functions associated with an eliminated variable x_i, since this requires the enumeration of 2^k possible values.

Fortunately, one may delay the evil day when the complexity gets out of hand by eliminating alternating vertices in the lattice. Considering the vertices as coloured black and white like the squares of a chess-board, we eliminate all the white vertices, thereby halving the number of vertices in the graph. The result is to eliminate every alternate vertex and join its neighbouring vertices in a 4-clique. It may be observed that each elimination operation here is independent of the others. The resulting co-occurrence graph is shown in Fig 2 (left).

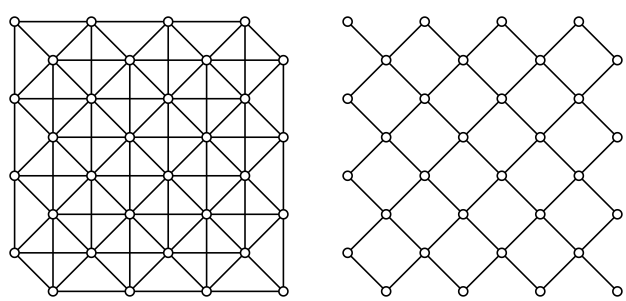


Figure 2. *Co-occurrence graph after elimination of alternating nodes from the lattice (left) and after approximation step (right).*

The approximation step. It may be observed at this stage that each vertex is now 8-connected. If we continue to eliminate vertices in the graph replacing them with cliques involving their neighbouring vertices, the complexity soon gets out of hand, and we come to a grinding halt. Therefore, to continue, drastic action is necessary. Our heuristic is to replace each clique of four vertices by a simpler graph obtained by eliminating the diagonal edges of the resulting squares. In graphical terms, this is illustrated in Fig 3. It is known as the *approximation step* in our algorithm.

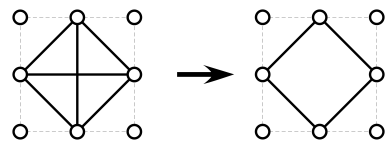


Figure 3. *The approximation step simplifies node dependencies in the lattice.*

We now explain what this means algebraically. The clique on four vertices $x_u, \ldots, x_\ell$ represents the function $x_i^* \Delta_i(x_u, \ldots, x_\ell)$ which we have said may be a complicated function of the four variables $x_u, \ldots, x_\ell$. The approximation step is to replace this function by a quadratic function of the form

$$\begin{aligned} c + a_u x_u + a_r x_r + a_d x_d + a_\ell x_\ell \\ + b_{ur} x_u x_r + b_{rd} x_r x_d + b_{d\ell} x_d x_\ell + b_{\ell u} x_\ell x_u \end{aligned} \tag{2}$$

where constant c and coefficients $a_u, \ldots, a_\ell, b_{ur}, \ldots, b_{\ell u}$ are to be chosen so as best to approximate the function $x_i^* \Delta_i$. This is a simple linear least-squares optimization problem involving the best choice of the 9 coefficients in (2) to approximate the function values for the 16 choices of $x_u, \ldots, x_\ell$. This involves a set of linear equations of the form $\mathtt{A}\mathbf{a} = \mathbf{c}$, where $\mathtt{A}$ is a 16×9 matrix independent of the data. One may therefore calculate its pseudo-inverse up front, so computing $a_u, \ldots, b_{\ell u}$ involves only a single matrix multiplication.

Recursion. The co-occurrence graph of the function resulting from the approximation step is shown in Fig 2 (right) obtained from Fig 2 (left) by eliminating the diagonals of the squares. We now observe that this is again a 4-connected lattice, rotated by $45°$ from the original orientation. We may now repeat a second round of elimination, deleting alternate vertices (equivalent to removing every other row of vertices in the original orientation) of the co-occurrence graph and applying the approximation step again. This time we get back to a 4-connected lattice in the original orientation with only one quarter of the original vertices remaining. By proceeding recursively in this way, we eventually arrive at a single variable x_n with coefficient a_n. This vertex can then be labelled as 0 or 1 depending on whether a_n is positive or negative.

Back substitution. We can now back up through the levels of elimination. Suppose that a node x_i was eliminated at

some point in the forward sweep, and that the terms involving x_i were of the form

$$x_i\Delta_i(x_u,\ldots,x_\ell) = x_i(c + a_u x_u + \ldots + a_\ell x_\ell + b_{ur}x_u x_r + \ldots + b_{\ell u}x_\ell x_u) \quad (3)$$

Note that $x_u,\ldots,x_\ell$ are not the neighbouring variables in the original lattice, but the neighbours in the reduced lattice at the moment when x_i is due for elimination. During the back-substitution phase, we know the assignment $x_u^*,\ldots,x_\ell^*$ to each of the variables $x_u,\ldots,x_\ell$. We may now assign value 0 or 1 to x_i depending on whether $\Delta_i(x_u^*,\ldots,x_\ell^*)$ is positive or negative.

More about implementation. As became apparent in the description of the algorithm given above, the algebraic manipulation of the pseudo-boolean function is tightly connected to actions on the co-occurrence graph. Since we will at all times be dealing with quadratic functions, we may define a closer connection between the function being minimized and a weighted version of the co-occurrence graph. Weights are assigned both to the vertices and the edges of the graph. The vertices are associated with variables x_i, and a weight a_i on such a vertex corresponds to a term $a_i x_i$ in the function $f(x_0,\ldots,x_n)$. An edge between vertices x_i and x_j with weight b_{ij} represents the term $b_{ij}x_i x_j$ in the polynomial representation of the function f. Note that edges are undirected, and weights in general may be positive or negative.

Ignoring constant terms (which are irrelevant as far as optimization is concerned), there is evidently a one-to-one correspondence between quadratic pseudo-boolean functions and weighted graphs constructed in this way. The graphs we are interested in are 4-connected lattices. We now examine the main elimination/approximation step of the algorithm in terms of graphs.

When we eliminate a vertex x_i in a graph, we compute the function $x_i^*\Delta_i(x_u,\ldots,x_\ell)$, and approximate it by a function of the form (2). This results in the introduction of new edges corresponding to terms $x_u x_r,\ldots,x_\ell x_u$. Weights are "passed down" from the vertex x_i and its incident edges to the four neighbouring vertices $x_u,\ldots,x_\ell$ and the new edges. When this is done, the vertex and incident edges are removed from the graph. Weights generated in this way are added to the weights on the vertices and accumulated on the new edges. In general each new edge will receive weight from two adjacent vertices. We say that the vertices and edges *inherit* weights from the nodes being eliminated.

To push the analogy a little further, we say that the nodes adjacent to a node being eliminated are its *children*. Thus, the whole forward pass of the algorithm consists simply of eliminating vertices and passing down weights to the children and new edges created between the children.

Remarks. We will refer to the recursive elimination/approximation algorithm just described as Elimination (with a capital E). An important observation may be made. As the recursive elimination proceeds, there is a trend for edges weights towards negative and ultimately zero values. Usually, this occurs within two or three levels of elimination. We give a brief explanation of this phenomenon.

The vector $\mathbf{b} = x_i^*\Delta_i(x_u,\ldots,x_\ell)$ contains only zero or negative entries, since $x_i^* = 0$ if $\Delta_i(x_u,\ldots,x_\ell) > 0$. In the approximation step, we express this function in terms of positive functions – the matrix $\mathtt{A}$ contains only positive values. Therefore, we expect most of the resulting coefficients in (2) to be negative. This is not meant as a rigorous justification, but an explanation of an empirical phenomenon.

Ultimately, edge weights become zero (exactly). This may be explained as follows. The edge weights express the interactions between the nodes. As we descend through levels of elimination, the edges extend to increasingly distant nodes, and the interactions become weak, ultimately vanishing. A more mathematical analysis is possible, but since this is not essential to our further discussion, we omit details.

3. The LazyElimination Algorithm

In the Elimination algorithm, as described in section 2, nodes are eliminated from the lattice until only one remains. When each node is eliminated, some approximation of the function potentially takes place, resulting in a suboptimal solution. Although we can not in general hope to be able to find an absolute minimum, it is clear that it is best to avoid eliminating nodes from the graph where it is not necessary. This is the guiding principal of the LazyElimination algorithm to be described in this section.

It is well known [3] that a function of the form (1) is submodular, and hence can be solved using max-flow, if and only if $b_{ij} \leq 0$ for all i, j. Thus, positive values of b_{ij} need to be eliminated in order for the function (1) to be minimized using graph-cuts. We have observed in section 2 that eliminating nodes from the lattice has the effect that edge weights in the co-occurrence graph tend towards negative or zero values. Therefore, we propose an algorithm (called LazyElimination) based on the principle of eliminating only as many nodes as is necessary. In broad outline, the algorithm is as follows.

1. Eliminate nodes from the graph using elimination/approximation moves until there are no more edges with positive edge weight. In doing this, only remove those nodes that need to be removed and no more.

2. Now, solve the problem using max-flow on the graph representation of the function.

Since the amount of approximation is minimized, we expect this algorithm to work better than the full Elimination algorithm, which eliminates all but one node. This expectation is borne out by experiment. The algorithm embodying this principle will be called LazyElimination.

Details of LazyElimination. To describe the LazyElimination algorithm accurately, it is necessary to clarify the concept of levels of nodes in a 4-connected lattice, and their parent-child relationship.

Given coordinates (i, j) for a node in a rectangular lattice, the nodes for which $i + j$ is odd are said to be at level 0. These are the nodes that are eliminated in the first pass of the Elimination algorithm. They have no parents. The children of each such node are the four nodes adjacent horizontally or vertically. During an elimination/approximation step, these children, along with diagonal edges between them, inherit weight from the parent node being eliminated. Analogously, we say that nodes eliminated during the n-th pass of the Elimination algorithm are at level $n - 1$. Levels may be determined as follows.

- A node (i, j) is at level $2k$ if $i/2^k$ and $j/2^k$ are both integers, one even, one odd. Its children are 2^k positions away horizontally or vertically.
- The node is at level $2k + 1$ if $i/2^k$ and $j/2^k$ are both odd integers. Its children are 2^k positions away diagonally in all directions.
- The node $(0, 0)$ may be given a level higher than any other node in the graph. It has no children.

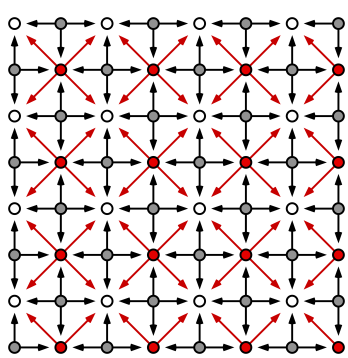


Figure 4. *Pointers (directed edges) from a node to its children. Level-*0 *nodes are coloured dark grey, level-*1 *nodes and their arrows in red. Edges originating at higher-level nodes (white) are not shown, to avoid cluttering the diagram. Every node has edges only to its four children. Edges emanating from nodes at level* 2 *or greater are not shown.*

We consider a graph constructed in this way with *directed* and weighted edges from each node to its four children, as shown in Fig 4. Note that the four parents of a node at level $k > 0$ are all at level $k - 1$, whereas two of the children of a node at level k are at level $k + 1$ and the other two children are at a higher level. Observe (see Fig 4) that the edges starting at nodes at level 0 account for all the edges of the original 4 connected lattice and are given the weight of the corresponding quadratic term $b_{ij}x_ix_j$ in the cost function. All the other edge weights starting at nodes at level greater than 0 initially have weight zero. These are just the edges that will be introduced into the graph by elimination/approximation moves. For instance, the edges attached to nodes at level 1 are those that are introduced during the first phase of the elimination/approximation.

As an implementation detail, the only storage requirement is an array of records one for each node in the lattice. For each node, we need to store the weight of the node itself, the weight of the edges to its four children, a flag indicating whether the node has been eliminated or not, and (implicitly by its position in the array), the coordinates of the node. We do not store pointers to the parents or children of a node, since these are easily and efficiently calculated.

The LazyElimination algorithm works on the principle of eliminating only nodes that are incident to edges with positive weight, subject to the principle that the parents of a node must be eliminated before the node itself, since they may contribute to the child's edge weights. The main routine of the elimation is given by a recursive subroutine called LazyEliminate. In this description, we ignore the issue that some of the parents or children of a node may lie outside the bounds of the lattice, and hence be non-existent. This needs to be checked when parent and children nodes are accessed.

Subroutine LazyEliminate (i, j)

1. If node (i, j) has already been eliminated, return.
2. If none of the edges from node (i, j) to its children has positive weight, return.
3. For each of the parents (i', j') of node (i, j), call **LazyEliminate** (i', j').
4. Check again: if none of the edges from node (i, j) to its children has positive weight, return.
5. If we have got this far, then eliminate the node (i, j), using an elimination/approximation step as follows:
   (a) Mark node (i, j) as eliminated.
   (b) Compute the inherited weights to be passed to children and their incident edges, based on the weight of the node (i, j) and its four edges.
   (c) Pass down the inherited weights to the children and incident edges.
   (d) For each of the two children (i', j') at the next level, call **LazyEliminate** (i', j').

The main eliminate routine consists now of calling the routine **LazyEliminate**(i, j) for each level-0 node (i, j) in the lattice. For efficiency, it is valuable to note that when inherited weights are passed down to child nodes and edges, they may overwrite the existing values on those nodes. The original values will not be needed any more, including in the back-substitution phase of the algorithm. For back substitution at some node, only the weights of the node and its incident edges are needed, and these are not changed by eliminating that node.

This elimination phase terminates with a situation where there are no remaining positive edges in the graph. The remaining optimization problem can then be solved using a max-flow algorithm. The graph used for max-flow contains only the remaining (non-eliminated) vertices in the co-occurrence graph and non-zero edges between non-eliminated vertices. After some of the node labels are computed using max-flow, the others are computed using back-substitution.

4. Experiments

We evaluated the performance of Elimination and LazyElimination against implementations of: max-flow [2], roof-dual [11], Fast PD [8], tree reweighted message passing [13], and max-product belief propagation [13]. All trials involved non-submodular energy functions. For max-flow, we truncated the non-submodular terms of the energy function [6].

Synthetic Non-Submodular Problems. A submodular pseudo-boolean function defined on a 4-connected lattice was generated by sampling values for the coefficients a_i and b_{ij} in (1) from uniform distributions $[0, 1]$ and $[-1, 0]$ respectively. A random subset of the variables $\mathbf{x}$ were replaced with their complements $1 - \bar{x}_i$. In this substitution, the coefficient b_{ij} of any quadratic term where only one of x_i or x_j was complemented became positive. The resulting function (achieved by interpreting complements as new variables) was non-submodular, and was minimized using roof-dual, max-flow, Elimination and LazyElimination.

Since roof-dual is invariant to complementing variables, it will always recover an optimal labelling for the particular non-submodular problems just described. (Recall that it is unable to produce an optimal assignment for every variable in general). Therefore, we quantified the quality of the remaining three methods by counting the number of assignments $x_i^\star$ which were different from that of roof-dual (see Figs 5 and 6).

Real Multilabel Problems. The binary algorithms max-flow, roof-dual, Elimination and LazyElimination can be extended to multilabel problems using alpha-expansion. Our multilabel experiments (to be described shortly) all formulate energy using data terms $E_i(x_i)$, smoothness terms $E_{ij}(x_i, x_j)$, and a regularization parameter λ:

$$E(\mathbf{x}) = \sum_i E_i(x_i) + \lambda \sum_{i,j} E_{ij}(x_i, x_j) \qquad (4)$$

Every algorithm (including those that can deal directly with multilabel variables, such as Fast PD) was run for three iterations. In the case of alpha-expansion, this consisted of solving 3ℓ binary problems, where ℓ is the number of labels. The initial configuration for α-expansion was determined by optimizing the data functions $E_i(x_i)$ independently. The label order used for α-expansion was the same for all pseduo-boolean optimization algorithms.

Denoising. We quantized an 8-bit image into 64 grey levels and added synthetic Gaussian noise ($\sigma = 4$ labels). The data cost was the square difference between the noisy input $\mathbf{I}_i$ and the hypothesized enhanced label x_i, limited to a maximum difference of 2σ: $E_i(x_i) = \min(|\mathbf{I}_i - x_i|, 2\sigma)^2$. The smoothness term was also a truncated convex function

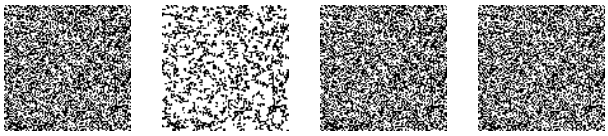


Figure 5. *Typical results for a synthetic non-submodular function created by complementing a random subset of variables of a submodular function. In this specific example, half the variables are complemented. Roof-dual produces the optimal labelling. The effect of truncation is clearly visible in the max-flow result.*

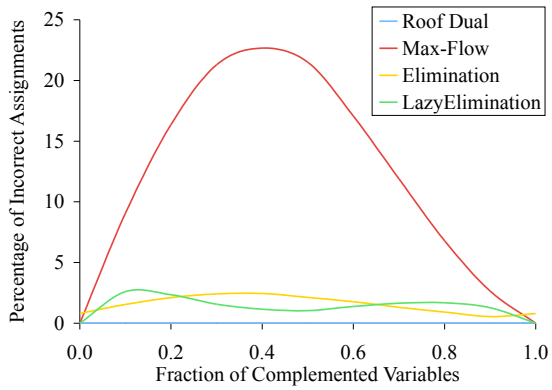


Figure 6. *The average number of incorrect variable assignments for ten random pseudo-boolean functions is shown for each optimization algorithm. Roof-dual calculates the optimal labelling and has no incorrect assignments. Max-flow produces a large number of incorrect labels because of truncation. Elimination is unable to produce an optimal labelling when the function is submodular, but maintains a low error rate when non-submodular. LazyElimination combines the benefits of both max-flow and Elimination, and is always able to produce a solution for every variable in a non-submodular problem.*

$E_{ij}(x_i, x_j) = \min(|x_i - x_j|, 2\sigma)^2$, with the regularization parameter λ manually tuned to 0.4. Results are summarized in Table 1. Since all algorithms produced similar answers, only a subset of results are shown in Fig 7.

Algorithm	Energy ($\times 10^5$)	Time (s)	RMS Error
Max-Flow	8.0006	10.70	2.37
Roof-Dual	8.0002	17.81	2.37
Elimination	8.0103	10.05	2.39
LazyElimination	8.0836	14.43	2.40
Fast PD	8.4940	8.23	2.64
TRW-S	7.9761	176.13	2.36
MaxProd BP	8.5962	152.18	2.62

Table 1. *Every algorithm was able to find a suitable enhancement of the noisy image. The methods based on graph-cuts were quick and used a minimal amount of memory. The implementation of Fast PD has been optimized for speed and used approximately 100× more memory. Tree reweighted message passing was able to find the lowest energy but took significantly longer. Although the algorithm had a competitive solution after its first iteration, even this took significantly longer (59.02s) than the other methods.*

Figure 7. *The corrupted input image was created by adding Gaussian noise ($\sigma = 4$ labels) to a 256×256 image quantized to 64 labels/grey levels. A truncated convex function was used for both the data and smoothness terms. The enhancement of LazyElimination is shown on the right. All algorithms were run for three iterations, and each produced a similar solution.*

Number of eliminations. The number of nodes eliminated during α-expansion with LazyElimination is quite small (see Fig 8). This observation also explains why max-flow is able to do so well under the truncation assumption.

Corridor. Sometimes, only specific labellings are viable. In [10], for instance, the set of labels {left, right, centre, top, bottom} corresponds to the predominant planes in the view of a corridor. The geometry of the scene induces asymmetric spatial constraints between the labels. For example, a pixel labelled 'bottom' (corresponding to the floor of the corridor) can not occur above any other pixel not labelled 'bottom', as it is impossible to have the floor of the corridor physically above the walls or ceiling. Non-sensical labellings (such as one which has a label 'bottom' existing above a label 'top') are avoided by assigning them high costs. However, this restriction causes the binary functions encountered in α-expansion to be non-submodular.

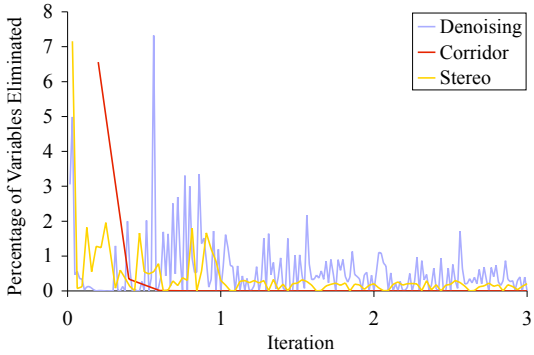


Figure 8. *Number of nodes eliminated per run of LazyElimination during α-expansion. Recall that the number of binary problems per iteration depends on the size of the label set.*

We generate a synthetic 256×256 corridor labelling $\mathbf{I}$, and then randomly assign new labels (where the five labels are equally probable) to three quarters of the pixels. Both the data and smoothness terms are Potts models, and are given equal weighting — i.e., $\lambda = 1.0$. In both cases, the cost associated to a pair of non-indentical labels is 1.0. The smoothness term also checks to see if the spatial layout of the two labels is permissible; invalid orderings are assigned a high cost. Results for the various algorithms are shown in Fig 9.

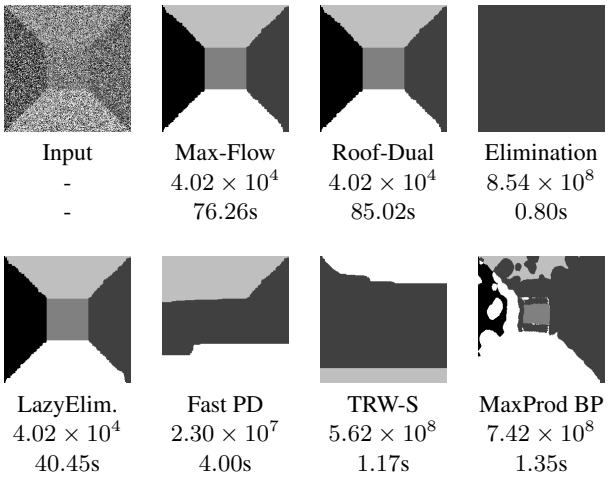
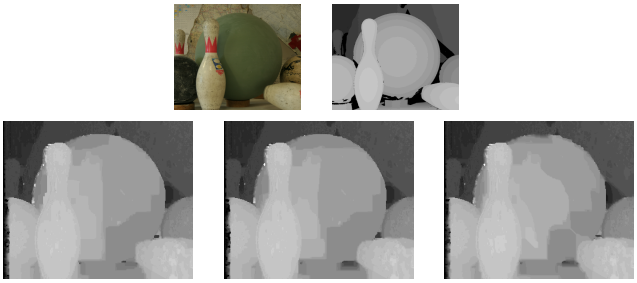


Figure 9. *The geometry of a corridor induces asymmetric spatial constraints between labels. An initial labelling was produced by randomly re-labelling three-quarters of the variables. Here, only max-flow, roof-dual and LazyEliminaton are able to find a correct labelling. Of these three, LazyElimination is roughly 2× faster.*

The high cost associated with invalid labellings distorts the approximation step of the Elimination algorithm. As a result, Elimination and LazyElimination move between invalid labellings before converging on a uniform labelling (which max-flow and roof-dual do immediately). Elimination is unable to progress beyond this point, but maintains a consistent time (compared to the denoising experiment) of

0.05s per 256×256 binary problem. LazyElimination is able to calculate moves from one valid labelling to another faster than both max-flow and roof-dual. The other algorithms are unable to converge to reasonable solutions even when initialized with a valid uniform labelling.

Stereo. Given images of the left $\mathbf{L}$ and right $\mathbf{R}$ views of the scene, the cost of assigning pixel i disparity x_i is determined by the difference in appearance between the two pixels: $E_i(x_i) = \|\mathbf{L}(i) - \mathbf{R}(i - x_i)\|^2$. Like denoising, we employed a truncated convex function to enforce smoothness between disparity values, with truncation manually tuned to a maximum difference of 4: $E_{ij}(x_i, x_j) = \min(4, |x_i - x_j|)^2$. The regularization parameter λ was set at 12. We use 32 labels (allowing disparity estimations to the nearest half pixel) and 332×277 scaled versions of the Middlebury 2006 dataset.



LazyElim.	Fast PD	TRW-S
3.18×10^6	3.19×10^6	3.17×10^6
12.32s	5.01s	73.58s

Figure 10. *The 'bowling2' trial from the Middlebury 2006 dataset. The data term is based on the appearance difference between a pair of hypothesized corresponding pixels. The smoothness term incorporates a truncated convex cost function.*

5. Conclusions

The LazyElimination algorithm described here achieves excellent results on minimizing arbitrary pseudo-boolean energy functions defined on a 4-connected lattice. When combined with alpha-expansion, it is able to obtain quality solutions to multilabel problems. It is faster (or at least on par) with all but Fast PD, which has been optimized for speed at the expense of memory (such as pre-computing and caching individual energy values). With optimizations like dynamic graph-cuts [4], we expect at least to match the speed of Fast PD.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

We would like to thank Fangfang Lu for her assistance with the Elimination algorithm.

References

[1] E. Boros and P. L. Hammer. Pseudo-boolean optimization. *Discrete Appl. Math.*, 123:155 – 225, 2002. 1, 2

[2] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001. 1, 6

[3] D. Freedman and P. Drineas. Energy minimization via graph cuts: Settling what is possible. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2*, pages 939–946, Washington, DC, USA, 2005. IEEE Computer Society. 4

[4] P. Kohli and P. H. S. Torr. Dynamic graph cuts for efficient inference in markov random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(12):2079–2088, 2007. 8

[5] V. Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1568 – 1583, October 2006. 1

[6] V. Kolmogorov and C. Rother. Minimizing nonsubmodular functions with graph cuts — a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(7):1274–1279, 2007. 6

[7] V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2):147–159, 2004. 1

[8] N. Komodakis and G. Tziritas. Approximate labeling via graph cuts based on linear programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(8):1436–1453, 2007. 1, 6

[9] N. Komodakis, G. Tziritas, and N. Paragios. Performance vs computational efficiency for optimizing single and dynamic MRFs: Setting the state of the art with primal-dual strategies. *Comput. Vis. Image Underst.*, 112(1):14–29, 2008. 1

[10] X. Liu, O. Veksler, and J. Samarabandu. Graph cut with ordering constraints on labels and its applications. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, 2008. 7

[11] C. Rother, V. Kolmogorov, V. Lempitsky, and M. Szummer. Optimizing binary mrfs via extended roof duality. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, pages 1 – 8, 2007. 1, 6

[12] R. Szeliski. Locally adaptive hierarchical basis preconditioning. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 25(3):1135–1143, 2006. 1

[13] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparitive study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(6):1068–1080, 2008. 6