University of Illinois Fall 2018

ECE 544NA: Pattern Recognition

Lecture 13: Structured Inference (ILP & LP Relaxation): October 11 & 16

Lecturer: Alexander Schwing Scribe: Andy Lai

1 Introduction

Today we will consider our discussion on structured inference programs. Previously, we introduced the covered two algorithms, exhaustive search and dynamic programming. In the following lecture we will explore four new algorithms: integer linear programming, linear programming relaxation, message passing, and graph cuts.

1.1 Review of Structured Inference

We use the following equation to represent the inference program.

$$\mathbf{y}^* = \underset{\hat{\mathbf{y}}}{\operatorname{argmin}} \sum_r f_r(\hat{\mathbf{y}}_r) \tag{1}$$

Structured inference takes advantage of correlations in the output space, and rather than predicting each output variable independently. In this model, the unary output variables are subsumed into one vector, $\mathbf{y}^{(i)} = (y_1^{(i)}, y_2^{(i)}, \ldots, y_{\mathcal{D}}^{(i)})$. Here, the superscript index (i) pairs the output $\mathbf{y}^{(i)}$ with input data $x^{(i)}$, while the subscript index r enumerates the labels we want to predict.

Examples that we have gone over in class include handwriting prediction, where we can use known spelling patterns to better predict single letters; and semantic segmentation, where we can enforce a smoothness prior by looking at neighboring pixels.

1.2 Review of Algorithms

The two algorithms we have covered are dynamic programming and exhaustive search. In exhaustive search, we try every possible configuration $\hat{\mathbf{y}} \in \mathcal{Y}$ and select the highest scoring configuration. This has the advantage of being very simple to implement, which cannot be overlooked and is actually very useful in the implementation of structured inference model. However it is very slow as it must search through a space of size K^D, where K is the number of labels and D is the number of predictions made.

In contrast, dynamic programming is much faster. It works by interleaving functions and maximizations and taking advantage of a tree-like structure. However, it is restricted to trees, and most structured inference problems cannot be graphically represented as trees.

2 Integer Linear Programming

An integer linear program (LP) is an optimization problem over a set of integer valued variables, which this lecture refers to as $b_r(\mathbf{y}_r)$, where both the objectives and constraints are linear [2]. We will define both of these below.

Consider an inference problem posed as (1) with a unary output space of $2 \cdot 2$, i.e., we infer between two labels for two outputs. We then impose structure by also considering the pairwise term, $\mathbf{y}_{1,2}$.

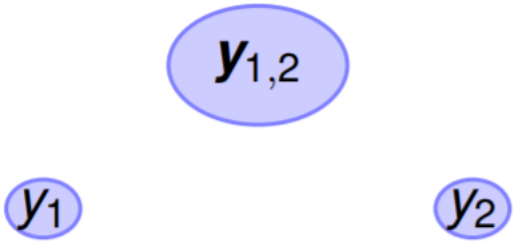


Figure 1: ILP graph example

Thus, the resultant output space has size $2 \cdot 2 + 2^2 = 8$, where the first addend represents the output space of the unary potentials and the second represents the output space of the pairwise potential.A graphical representation of this problem is seen in Fig. 1 which shows the two unary potentials, $\mathbf{y}_1$ and $\mathbf{y}_2$ as well as the pairwise potential, $\mathbf{y}_{1,2}$.

We introduce the optimization variables $b_r(\mathbf{y}_r)$, on which we will later impose constraints. We now formulate this problem as a maximization over $b_r(\mathbf{y}_r)$ as follows:

$$\max_{b_1,b_2,b_{12}} \begin{bmatrix} b_1(1) \\ b_1(2) \\ b_2(1) \\ b_2(2) \\ b_{12}(1,1) \\ b_{12}(1,2) \\ b_{12}(2,1) \\ b_{12}(2,2) \end{bmatrix}^T \begin{bmatrix} f_1(1) \\ f_1(2) \\ f_2(1) \\ f_2(2) \\ f_{12}(1,1) \\ f_{12}(1,2) \\ f_{12}(2,1) \\ f_{12}(2,2) \end{bmatrix} \tag{2}$$

$b_r(\mathbf{y}_r)$ is multiplied element-wise with $f_r(\mathbf{y}_r)$, and the resultant vector is summed up. In other words, we take the inner product between $b_r(\mathbf{y}_r)$ and $f_r(\mathbf{y}_r)$. This inner product is maximized with respect to $b_r(\mathbf{y}_r)$.

Here, the structure for $b_r(\mathbf{y}_r)$ mirrors that of $f(\mathbf{y}_r)$. The subscript index r of the variable enumerates which output we are looking at, while the value inside the parentheses is the label that it takes on. In this example, since there are two labels we are inferring between, $\mathbf{y}_r$ can take on a value in $\{1, 2\}$.

Now, we introduce the following constraints on $b_r(\mathbf{y}_r)$.

$$b_r(\mathbf{y}_r) \in \{0, 1\} \qquad \forall r, \mathbf{y}_r \tag{3}$$

$$b_r(\mathbf{y}_r) \geq 0 \qquad \forall r, \mathbf{y}_r \tag{4}$$

$$\sum_{\mathbf{y}_r} b_r(\mathbf{y}_r) = 1 \qquad \forall r \tag{5}$$

$$\sum_{\mathbf{y}_p \setminus \mathbf{y}_r} b_p(\mathbf{y}_p) = b_r(\mathbf{y}_r) \qquad \forall r, \mathbf{y}_r, p \in P(r) \tag{6}$$

The first constraint, (3), restricts each variable in $b_r(\mathbf{y}_r)$ to take on a binary value. Meanwhile, the second and third constraints, (4) and (5), enforce that $b_r(\mathbf{y}_r)$ is a valid probability distribution. This probability distribution is local to each output $\mathbf{y}_r$. We can also think of it as being local to each node in the graph. Because $b_r(\mathbf{y}_r)$ is constrained to being a binary variable, this of course means

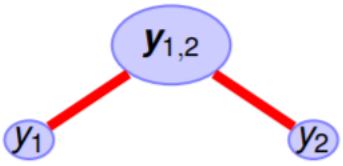


Figure 2: ILP graph example with parent-child representation

$b_r(\mathbf{y}_r) = 1$ for only one label for $\mathbf{y}_r$. For instance, we cannot have both $y_1 = 1$ and $y_1 = 2$, because $b_1(1) + b_1(2) = 1$. Lastly, the fourth constraint, (6) is a marginalization that takes advantage of the parent-child relationship in the graph. Consider $b_p(\mathbf{y}_p)$ as a joint distribution, of which $\mathbf{y}_r$ is one variable — then we are marginalizing over the variable $\mathbf{y}_r$ to find its distribution, $b_r(\mathbf{y}_r)$.

Fig. 2 shows the parent-child relationship for this example with directed edges from the parent to the children, and we can use it to understand the constraint given in (6). In this example, $\mathbf{y}_{1,2}$ is the parent node of its children, $\mathbf{y}_1$ and $\mathbf{y}_2$. In the general case, we can think of $\mathbf{y}_p$ as being the parent of a node $\mathbf{y}_r$. In this example, for the two children, we would have:

$$b_1(1) = b_{12}(1,1) + b_{12}(1,2)$$
$$b_2(1) = b_{12}(1,1) + b_{12}(2,1)$$

More generally, this constraint tells us to go through the entire region of the variable $b_r(\mathbf{y}_r)$ — graphically, we can think of this as iterating through all the nodes in the graph. Then, we enforce that, for any child $\mathbf{y}_r$, the joint distribution (or optimization variables) for its parent node marginalized over $\mathbf{y}_r$ must sum to the distribution (or optimization variable) of $\mathbf{y}_r$ itself. In other words, the relationship between any parent $\mathbf{y}_p$ and child $\mathbf{y}_r$ must be a valid joint probability distribution that obeys the law of total probability.

Thus, the constraints can be thought of as follows:

- $b_r(\mathbf{y}_r) \in \{0, 1\}$ (integer constraint)
- local probability b_r
- marginalization

More generally, we can rewrite (2), along with constraints (3)-(6), as follows:

$$\max_{b_r} \sum_{r,\mathbf{y}_r} b_r(\mathbf{y}_r) f_r(\mathbf{y}_r) \quad \text{s.t.} \quad \begin{aligned} b_r(\mathbf{y}_r) &\in \{0,1\} && \forall r, \mathbf{y}_r \\ b_r(\mathbf{y}_r) &\geq 0 && \forall r, \mathbf{y}_r \\ \sum_{\mathbf{y}_r} b_r(\mathbf{y}_r) &= 1 && \forall r \\ \sum_{\mathbf{y}_p \setminus \mathbf{y}_r} b_p(\mathbf{y}_p) &= b_r(\mathbf{y}_r) && \forall r, \mathbf{y}_r, p \in P(r) \end{aligned} \tag{7}$$

Integer LP has the advantage of always being able to find a global optimum for the inference problem. In addition, good solvers already exist and are available. However, integer LP, while faster than exhaustive search, is still very slow for large problems.

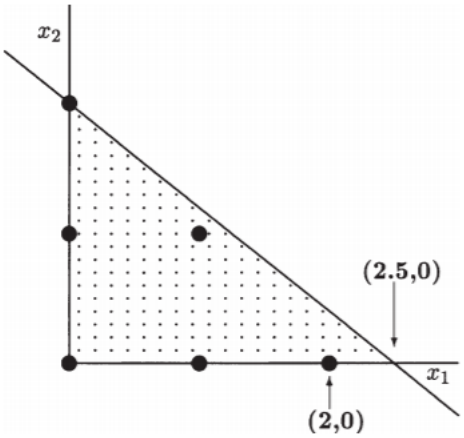


Figure 3: Example of LP relaxation with two variables

3 Linear Programming Relaxation

LP relaxation addresses the issues of speed that were present with integer LP. It *relaxes* the constraints of the integer LP problem by removing the integer constraint, (3). Thus we can simply rewrite (7) as follows:

$$\max_{b_r} \sum_{r,\mathbf{y}_r} b_r(\mathbf{y}_r) f_r(\mathbf{y}_r) \quad \text{s.t.} \quad \begin{aligned} b_r(\mathbf{y}_r) &\geq 0 && \forall r, \mathbf{y}_r \\ \sum_{\mathbf{y}_r} b_r(\mathbf{y}_r) &= 1 && \forall r \\ \sum_{\mathbf{y}_p \setminus \mathbf{y}_r} b_p(\mathbf{y}_p) &= b_r(\mathbf{y}_r) && \forall r, \mathbf{y}_r, p \in P(r) \end{aligned} \tag{8}$$

The optimization variables $b_r(\mathbf{y}_r)$ still must comprise a valid local probability density, and we still perform the marginalization.

LP relaxation reduces the complexity of integer LP, so it is a faster algorithm. The advantages are similar to that of integer LP: a global optimum for the LP relaxation problem can always be found, and good solvers exist for LP relaxation. However, a global optimum for the original integer LP problem is not guaranteed, and while faster than integer LP, the algorithm is still slow for larger problems.

Fig. 3 shows an example of LP relaxation over the variables x_1 and x_2 [1]. Unlike our case, where $\mathbf{y}_r$ is constrained to be a binary variable, here x_1 and x_2 are allowed to be any integer within the bounds delineated in the graph. The optimum solution of the original integer LP problem is found at $(2, 0)$ as Fig. 3 shows. However, when performing LP relaxation, the possible solutions are no longer constrained to the bold black dots but any point in the continuous shaded area. The new global optimum is at $(2.5, 0)$, which fits with our intuition that a global optimum is guaranteed for LP relaxation but it may not be the same optimum as the original integer LP problem.

4 Message Passing (Loopy Belief Propagation)

The third algorithm we cover today is message passing. Unlike dynamic programming, which only works on trees, message passing exploits any graphical structure which is defined via marginalized

constraints. Those marginalized constraints, which we interpreted above as parent-child representations, can be thought of as messages which are passed between nodes, leading to a stable inference at each node. This is also called loopy belief propagation as the marginal distribution, or "belief", is found using neighboring nodes; the term "loopy" refers to the application of this algorithm to cyclic graphs as well as trees.

Message passing is more efficient than the previously described algorithms, as it breaks down the LP relaxation problem into sub-problems that are easier to compute. We will see what this looks like below. However, message passing is prone to getting stuck in "corner-cases". Thus, special care must be taken to find the LP relaxation optimum.

Message passing operates by first calculating the dual of the LP relaxation problem, (8). Recall from Lecture 7: Optimization Dual that writing the dual program involves identifying the equality and inequality constraints, then assigning Lagrange multipliers λ_i and ν_i, respectively. The following steps are taken in computing the dual:

1. Bring primal program into standard form

$$\begin{aligned} \min_{\mathbf{w}} f_0(\mathbf{w}) \quad \text{s.t.} \quad & f_i(\mathbf{w}) \leq 0 \quad \forall\, i \in \{1, \ldots, C_1\} \\ & h_i(\mathbf{w}) = 0 \quad \forall\, i \in \{1, \ldots, C_2\} \end{aligned} \tag{9}$$

2. Assign Lagrangian multipliers to a suitable set of constraints
3. Subsume all other constraints in $\mathcal{W}$
4. Write down the Lagrangian L

$$L(\mathbf{w}, \lambda, \nu) = f_0(\mathbf{w}) + \sum_{i=1}^{C_1} \lambda_i f_i(\mathbf{w}) + \sum_{i=1}^{C_2} \nu_i h_i(\mathbf{w}) \tag{10}$$

5. Minimize Lagrangian w.r.t. primal variables $\mathbf{w} \in \mathcal{W}$

Now we will compute the Lagrangian for the primal problem posed in (8).

$$\begin{aligned} L() &= \sum_{r, \mathbf{y}_r} b_r(\mathbf{y}_r) f_r(\mathbf{y}_r) + \sum_{r, p \in P(r), \mathbf{y}_r} \lambda_{r \to p}(\mathbf{y}_r) \left(\sum_{\mathbf{y}_p \setminus \mathbf{y}_r} b_p(\mathbf{y}_p) - b_r(\mathbf{y}_r) \right) \\ &= \sum_{r, \mathbf{y}_r} b_r(\mathbf{y}_r) \left(f_r(\mathbf{y}_r) - \sum_{p \in P(r)} \lambda_{r \to p}(\mathbf{y}_r) + \sum_{c \in C(r)} \lambda_{c \to r}(\mathbf{y}_c) \right) \end{aligned} \tag{11}$$

There are a few important things to note regarding the formulating of this Lagrangian. The first is that our original primal problem is posed as a maximization rather than a minimization. In this lecture, we continue to keep it as a maximization problem; thus, the signs are flipped. Secondly, only the marginalization constraint (6) is used to formulate the Lagrangian. The rest are subsumed in $\mathcal{W}$ and will be used to maximize $L()$. And lastly, though it is an equality constraint, this lecture uses the notation of λ as the Lagrangian multiplier for simplicity.

The subscript notation $r \to p$ denotes the edge between a node $\mathbf{y}_r$ and its parent $\mathbf{y}_p$. The equivalence in the second line of (11) introduces a new index variable, c, which refers to a "child", analogous to p which refers to a "parent". Thus, the subscript notation $c \to r$ denotes the edge between a node $\mathbf{y}_r$ and its child $\mathbf{y}_c$. When we factor out $\sum_{y, \mathbf{y}_r} b_r(\mathbf{y}_r)$ from the second term, we must make sure that we account for both the parents and children of any given node. This would not have bearing on a simple problem like the one posed in Fig. 2, but it is critical for more complex problems.

Now we can write the maximization of the Lagrangian as follows:

$$\max_b L() \quad \text{s.t.} \quad \begin{cases} b_r(\mathbf{y}_r) \geq 0 & \forall r, \mathbf{y}_r \\ \sum_{\mathbf{y}_r} b_r(\mathbf{y}_r) = 1 & \forall r \end{cases} \tag{12}$$

Because other two other constraints, (4) and (5), which enforce a valid local probability distribution, were subsumed in $\mathcal{W}$ rather than being explicitly written in the Lagrangian, we must now take them into account in the maximization.

Then our dual function, $g(\lambda)$, becomes:

$$g(\lambda) = \max_{\mathbf{y}_r} \left(f_r(\mathbf{y}_r) - \sum_{p \in P(r)} \lambda_{r \to p}(\mathbf{y}_r) + \sum_{c \in C(r)} \lambda_{c \to r}(\mathbf{y}_c) \right) \tag{13}$$

Recall from Lecture 8 that the dual lower bounds the optimal primal value; thus, we would typically want to maximize the dual to find the optimal. But in this case, we are dealing with a *maximization* problem to begin with, so we will instead want to *minimize* the dual.

Thus, we can write our overall dual program as follows:

$$\min_{\lambda} \left[\max_{\mathbf{y}_r} \left(f_r(\mathbf{y}_r) - \sum_{p \in P(r)} \lambda_{r \to p}(\mathbf{y}_r) + \sum_{c \in C(r)} \lambda_{c \to r}(\mathbf{y}_c) \right) \right] \tag{14}$$

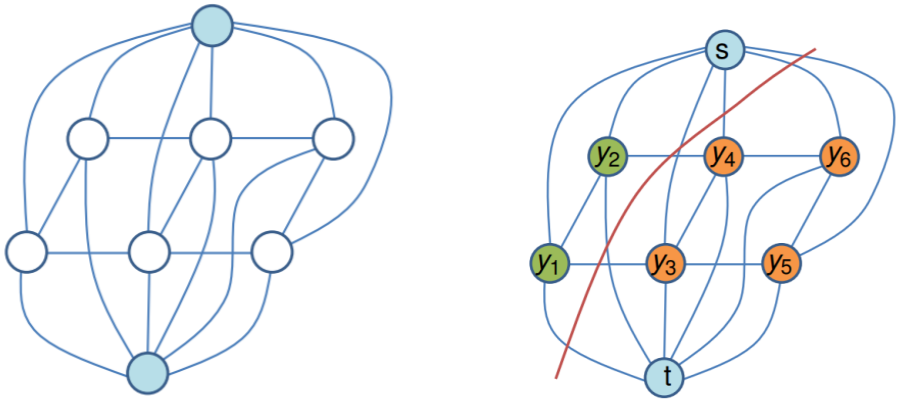
The properties of the dual program are as follows:

- Convex program
- Not strongly convex
- Piecewise linear
- **Unconstrained**
- Lagrange multipliers are messengers

Because this function is piecewise, it has a problem of falling into corner cases, as was mentioned above. Each of the Lagrange multipliers λ are messages on the edge of the graph, with the subscript denoting the specific edges. We can think of these messages as shifting 'energy' such that the local maximization in the dual will be equivalent to the global maximization in the primal. Thus, message passing breaks down the original problem into a series of linear functions which are less complex to analytically solve, while also removing constraints from the optimization. Calculating the message passing dual will still give a value equal to the original primal solution.

5 Graph-Cut

Graph-cut solvers are efficient algorithms which compute the minimum cut cost in a weighted graph that allows the maximum flow through a weighted graph. Fig. 4a shows us an example of one of these graphs for our binary problem where $y_d \in \{1, 2\}$. The output function nodes are arranged in a co-planar grid in the middle of the graph; in this example we can see that the unary and pairwise (between adjacent nodes) potentials are considered.



(a) Pre-cut auxiliary graph

(b) Auxiliary graph with minimum cost cut shown

Figure 4: Auxiliary graph with start and terminal nodes shaded in light blue

In addition, two nodes are added: a start node at the top, and a terminal node at the bottom. These correspond to our labels of $y_d \in \{1, 2\}$ and are both shaded in blue. There are weighted and directed edges pointing from the start node to each of the output function nodes, then from the output function nodes to the terminal node. We can think of this as water flowing down from the start to the terminal, and each of the edges representing a pipe, where the weight of the edge represents the flow rate through a pipe. The objective of the graph-cut algorithm is to find a cut where each output function node is connected to *only one of* the start or terminal nodes, but not both, such that the flow of water through the pipes is maximized. The cost of this is referred to as the minimum cut cost.

Fig. 4b shows an example of this minimum cost cut. The graph is cut in a way such that the edges between the green nodes and the start node remain, and the edges between the orange nodes and the terminal node remain. Thus, we can picture water flowing from the start node, to the green nodes, through the orange nodes, and finally down to the terminal node. But water cannot flow directly from the start to the orange nodes or from the green nodes to the terminal.

We can broadly simplify the graph-cut algorithm into two general steps.

- Convert scoring function F into auxiliary graph
- Compute a weighted cut cost corresponding to the labeling score

Note that this auxiliary graph is *not* the same graph from before, for example the graph in Fig. 2. It *must* have a start node s and terminal node t, and the edges are now weighted. The weights of these edges correspond to our labeling scores, $f_r(\mathbf{y_r})$. Each local scoring function is an array; for instance:

$$f_1(y_1) = \begin{bmatrix} f_1(y_1 = 1) & f_1(y_1 = 2) \end{bmatrix}$$

Fig. 5 shows these two labels displayed on the same graph in Fig. 4. Note that the inverse of these local scoring functions is taken because we want to find the minimum cut cost, which corresponds to a maximization of the scoring functions.

Consider the same example proposed in Section 2, with two binary outputs $y_1, y_2 \in \{1, 2\}$. We begin drawing the auxiliary graph as depicted in Fig. 6a, and then begin to compute the edges of our graph.

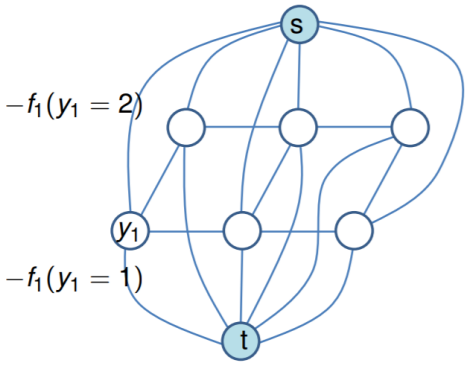
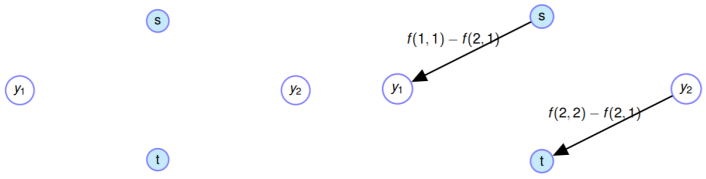
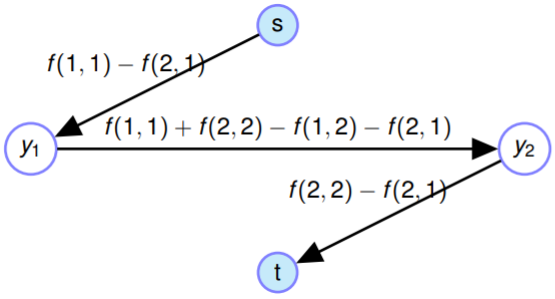


Figure 5: Auxiliary graph with one set of edge weights labeled



(a) Beginning of auxiliary graph (b) Unary potential edges



(c) Unary and pairwise edges

Figure 6: Graph-cut solvers compute a minimum cost cut

Now, taking advantage of the fact that we can write our local scoring functions in a matrix, we rewrite the labeling score function F as follows:

$$\begin{aligned}\begin{bmatrix} f_{12}(1,1) & f_{12}(1,2) \\ f_{12}(2,1) & f_{12}(2,2) \end{bmatrix} &= f(1,1) - f(2,1) + f(2,2) \\ &+ \begin{bmatrix} 0 & 0 \\ f(2,1) - f(1,1) & f(2,1) - f(1,1) \end{bmatrix} \\ &+ \begin{bmatrix} f(2,1) - f(2,2) & 0 \\ f(2,1) - f(2,2) & 0 \end{bmatrix} \\ &+ \begin{bmatrix} 0 & f(1,2) + f(2,1) - f(1,1) - f(2,2) \\ 0 & 0 \end{bmatrix} \end{aligned} \tag{15}$$

Simple matrix arithmetic confirms that these two expressions are equivalent, but the expression on the right breaks down how to weight our edges. The first term is a scalar, so does not factor into

our edge weights. The second and third terms determine the edge weights that we have added in Fig. 6b. Because they have constant rows or columns, they correspond to unary potentials; thus, the corresponding edges connect an output node with s or t, but will not connect two output nodes.

The second term has constant columns, so it only depends on y_1; thus, we draw an edge between y_1 and either s or t. Whether we choose the start or terminal node depends on the relation between $f(2,1)$ and $f(1,1)$. For $f_1(1) > f_1(2)$, this corresponds to an edge between y_1 and s, which is represented in Fig. 6b. The same is done for the second term; this term has constant rows, meaning it depends only on y_2. Our representation here, where the edge is drawn between y_2 and t, represents $f_2(2) > f_2(1)$ [3].

The final term does not have constant columns or rows; thus, it depends on both y_1 and y_2 and must be a pairwise term corresponding to an edge between the nodes. Note that all of these edges have their signs reversed from the matrix notation of the scoring functions, as discussed earlier. The final cut graphs in Fig. 6 are not definitive but correspond only to the case where $f_1(1) > f_1(2)$ and $f_2(2) > f_2(1)$. Due to regularization of the cost function F, the weight $f(1,1) + f(2,2) - f(1,2) - f(2,1) \geq 0$ is non-negative [3]. In addition, for a graph-cut algorithm to have guaranteed optimality, the pairwise terms must be positive.

For higher order functions, we would have more complicated graph constructions, but the general structure of the auxiliary graph would remain the same. Problems with more than two labels can be solved using a move-making algorithm. Move making algorithms operate by starting with an initial labeling guess, then iteratively lowering the energy in the solutions with an optimal change at each iteration. These algorithms can be solved using an st-cut like the one we have seen [4].

References

[1] C. A. Floudas and P. M. Pardalos. *Encyclopedia of Optimization*. Springer US, 2009.

[2] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, Cambridge, Massachusetts, 2009.

[3] V. Kolmogorov and R. Zabih. What energy functions can be maximized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2), February 2004.

[4] K. Pushmeet, M. P. Kumar, and P. H. Torr. $\mathcal{P}^3$ and beyond: Move making algorithms for solving higher order functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(9), September 2009.