A Background and Target Problem

For Problem (1), whether explicitly or not, existing methods use subgraph indicators as explanatory variables as shown in Figure 6. A subgraph indicator $I(x \subseteq g) \in \{0, 1\}$ indicates the presence or absence of a subgraph feature $x \in \mathcal{G}$ in the input graph $g \in \mathcal{G}$. However, we cannot explicitly obtain this matrix in practical situations, since the number of all possible subgraphs is intractably huge to enumerate, due to combinatorial explosion. Moreover, we need to test subgraph isomorphism to obtain each of 0 or 1 in the matrix, while we have to start with no column vectors and can obtain the column vector of $x_i$, only after we query a particular subgraph $x_i \in \mathcal{G}$ to a given set of $n$ graphs, $g_1, g_2, \ldots, g_n$. Thus, this is a good example of statistical learning for data with a combinatorially huge dimension where we cannot even enumerate all features beforehand.

Hence the main difficulty of Problem (1) is simultaneous learning of the model function $\mu$ and a finite set of relevant features $\{x_1, x_2, x_3, \ldots\} \subset \mathcal{G}$ with searching and identifying the feature subgraphs. In practice, most relevant subgraph features to a given problem should be selected but are unknown beforehand. Thus several methods have been proposed to learn a small number of relevant features from all possible subgraph features during the learning process of specific models [Kudo et al. 2005], [Tsuda and Kudo 2006], [Tsuda 2007], [Tsuda and Kurihara 2008], [Saigo et al. 2008], [Saigo and Tsuda 2008], [Saigo et al. 2009].

The primary objective of this paper is to give a coherent and unifying view to understand the principles, which are shared among these previous methods for different specific problems. Then we present a method to solve a general form of learning problems in the loss-function minimization framework by (2) and (3).

The form of (2) with the linear model (3) is a typical machine learning formulation that includes many existing methods with elastic-net regularization such as generalized linear models (GLMs) —including most fundamental statistical methods such as regression, logistic/Poisson/Cox regression, t-test, ANOVA, ANCOVA— and any smoothed hinge loss minimization including modified Huber loss minimization. In fact, our formulation (2) corresponds to several existing work:

- For $L = $ squared loss and $\lambda_2 = 0$, then we have the LASSO form for graphs. The path-following algorithm for LARS/LASSO for graphs was proposed in Tsuda [2007].

- For $L = $ exponential loss and $\lambda_2 = 0$, then we have an approximated form of Adaboost with early stopping. Adaboost over subgraph indicators was established in Kudo et al. [2005].

- For $L = $ hinge loss and $\lambda_1 = 0$, then we have the SVM form for graphs with all subgraph kernel. However this problem is well known to be computational infeasible in practice Gärtner et al. [2003].

- For $L = $ hinge loss and $\lambda_2 = 0$, then we have the 1-norm SVM Zhu et al. [2004] form for graphs. LPBoost for subgraph indicators targets this problem, and was already established as gBoost Saigo et al. [2009].
For $L = \text{logistic loss}$, then we have the logistic regression form with elastic-net penalty. For nominal labels $Y = \{T, F\}$, we can also define the logistic loss as in standard link-function formulation of GLM. This case is not well investigated in the literature, and is our main example of interest.

The hinge loss functions are not twice differentiable, but we can instead use the smoothed hinge loss functions or subgradient-type algorithms. We further note that Eq (2) has the elastic-net type regularizer: The third term of 2-norm $\|\beta\|^2_2$ encourages highly correlated features to be averaged, while the second term of 1-norm $\|\beta\|_1$, a sparsity-inducing regularizer, encourages a sparse solution in the coefficients of the averaged features by the 2-norm term.

### B Related Work

For Problem (1), the most well-used approach would be graph kernel method. So far various types of graph kernels have been developed and used in various applications successfully Kashima et al. [2003], Gärtner et al. [2003], Mahé and Vert [2009], Kondor and Borgwardt [2008], Vishwanathan et al. [2010], Shervashidze et al. [2011]. For example, we can compute the inner product of feature vectors of $g_i$ and $g_j$ in Figure 6, indirectly through a kernel trick as

$$k(g_i, g_j) := \# \text{ of common subgraphs between } g_i \text{ and } g_j,$$

which is, as well as many other proper graph kernels, a special case of $R$-convolution kernels Haussler [1999]. However Gärtner et al. [2003] showed that this all subgraph kernel is infeasible in practice. Hence, any practical graph kernel restricts the subgraph features to some limited types such as paths and trees, bounded-size subgraphs, or heuristically inspired subgraph features in individual applications.

In chemoinformatics, there have been many methods for computing the fingerprint descriptor (feature vector) of a given molecular graph Takigawa and Mamitsuka [2013]. The predefined set of subgraph features engineered with the domain knowledge would be still often used, but data-driven fingerprints are widely used in recent years: hashed fingerprints (such as Daylight\(^1\) and ChemAxon\(^2\),

---


extended connectivity fingerprints (ECFP) Rogers and Hahn [2010], frequent subgraphs Deshpande et al. [2005], and bounded-size graph fingerprint Wale et al. [2008]. The data-driven fingerprints adaptively choose the subgraph features from a limited type of subgraphs, resulting in that these fingerprints are similar to practical graph kernels.

In contrast to these two approaches, a series of inspiring studies for simultaneous feature learning via sparse modeling have been made Kudo et al. [2005], Tsuda and Kudo [2006], Tsuda [2007], Tsuda and Kurihara [2008], Saigo et al. [2008], Saigo and Tsuda [2008], Saigo et al. [2009]. They involve automatic selection of relevant features from all possible subgraphs during the learning process.

Triggered by the seminal work Kudo et al. [2005], it has been shown that we can perform the simultaneous learning on Adaboost Kudo et al. [2005], LARS/LASSO Tsuda [2007], sparse partial-least-squares (PLS) regression Saigo et al. [2008], sparse principal component analysis (PCA) Saigo and Tsuda [2008], and LPBoost Saigo et al. [2009]. This paper first aims to give a coherent and unifying view to understand principles, which are shared among these methods for specific problems. Then we present a method to solve a general problem of (2) including a wider class of important statistical problems such as GLMs.

Applying the glmpath algorithm Park and Hastie [2007], which extends LARS to path-following for GLM, to itemsets was recently proposed duVerle et al. [2013], while exact path-following algorithms like glmpath are known to be not scalable Friedman et al. [2010]. Thus direct-optimization-based learning methods of coordinate descent type, as we study in this paper, would be still in demand.

C Morishita-Kudo Bounds and Their Examples

The original bounds by Morishita [2002], Kudo et al. [2005] consider only the specific objective function of Adaboost, but many existing methods for other objectives, such as Tsuda [2007], Saigo et al. [2008, 2009], share the common idea as described by Morishita-Kudo bounds. We can say that most existing approaches are based on a branch-and-bound strategy with Morishita-Kudo bounds, which can be used for each class of learning problems with separable loss functions. We can see this fact in the three examples below.

C.1 Example 1: Maximizer of the weighted gain

Assume that we have a certain subgraph $x$. For the problem of classification with $y_i \in \{-1,1\}$, consider the upper bound of the gain attained by subgraph $z \supseteq x$

$$\sum_{i=1}^{n} w_i y_i (2I(z \subseteq g_i) - 1)$$

by $-1/1$-valued features of $2I(z \subseteq g_i) - 1$, where $w_i \geq 0$ Kudo et al. [2005]. Then, from Lemma 2, we have $x \subseteq z \Rightarrow 1(z) \subseteq 1(x)$. Thus we obtain a Morishita-Kudo upper bound

$$\sum_{i=1}^{n} w_i y_i (2I(z \subseteq g_i) - 1)$$

$$\leq \sum_{i \in 1(x)} \max\{w_i y_i (2 \cdot 0 - 1), w_i y_i (2 \cdot 1 - 1)\}$$

$$+ \sum_{i \in 0(x)} w_i y_i (2 \cdot 0 - 1) = \sum_{i \in 1(x)} w_i - \sum_{i \in 0(x)} w_i y_i.$$
In the original paper of Kudo et al. [2005], this bound was derived as
\[
\sum_{i=1}^{n} w_i y_i (2 I(z \leq g_i) - 1) = 2 \sum_{i=1}^{n} w_i y_i I(z \leq g_i) - \sum_{i=1}^{n} w_i y_i \\
\leq 2 \left\{ \sum_{i \in \{1\}} \max\{w_i y_i, 0\} + \sum_{i \in \{0\}} w_i y_i \cdot 0 \right\} - \sum_{i=1}^{n} w_i y_i \\
= 2 \sum_{i \in \{1\} \land y_i = 1} w_i - \sum_{i=1}^{n} w_i y_i.
\]

We can confirm that these two are equivalent by subtracting one from the other.

C.2 Example 2: Minimizer of the weighted classification error

Assume that we have a certain subgraph \( x \). For the problem of classification with \( y_i \in \{0, 1\} \), consider the lower bound of the weighted classification error attained by \( z(x) \) (6)
\[
\sum_{i=1}^{n} w_i I(I(z \leq g_i) \neq y_i),
\]
where \( w_i \geq 0 \). Similarly we have a Morishita-Kudo lower bound
\[
\sum_{i=1}^{n} w_i I(I(z \leq g_i) = y_i) \\
\geq \sum_{i \in \{1\}} \min\{w_i I(0 \neq y_i), w_i I(1 \neq y_i)\} + \sum_{i \in \{0\}} w_i I(0 \neq y_i) \\
= \sum_{i \in \{0\}} w_i I(0 \neq y_i) = \sum_{i \in \{0\} \land y_i = 1} w_i.
\]

In other words, the classification error depends only on elements taking \( y_i = 1 \) but predicted as \( I(x \leq g_i) = 0 \). Once the predictor value becomes 0, the objective function cannot be further improved by searching \( z \) such that \( z \geq x \). We can conclude that for \( x \) satisfying that the amount \( \sum_{i \in \{0\} \land y_i = 1} w_i \) is large, the possible value with \( z \) by further searching \( z \) satisfying that \( z \geq x \) is upper bounded by this amount.

C.3 Example 3: Maximizer of the correlation to the response

Assume that we have a certain subgraph \( x \). As seen in regression problems Saigo et al. [2009], consider the upper bound of the product-sum or uncentered-unnormalized correlation
\[
\left| \sum_{i=1}^{n} w_i I(z \leq g_i) \right| 
\]
between response \( w_i \in \mathbb{R} \) and indicator \( I(z \leq g_i) \) by subgraph feature \( z \) (\( \geq x \)). From \( |a| = \max\{a, -a\} \), the larger amount of
\[
\sum_{i=1}^{n} w_i I(z \leq g_i) \quad \text{or} \quad -\sum_{i=1}^{n} w_i I(z \leq g_i)
\]
gives the upper bound of the correlation. Here we have a Morishita-Kudo upper bound as

\[
\sum_{i=1}^{n} w_i I(z \leq g_i) \leq \sum_{i \in I(x)} \max\{w_i \cdot 0, w_i \cdot 1\} + \sum_{i \in 0(x)} w_i \cdot 0 \\
= \sum_{i \in I(x)} \max\{0, w_i\} = \sum_{i \in I(x) \cap w_i > 0} w_i.
\]

We can always have \(w_i I(x \leq g_i) = 0\) for \(i \in 0(x)\), and rewrite as

\[
\sum_{i=1}^{n} w_i I(z \leq g_i) \leq \sum_{i \in I(x) \cap w_i > 0} w_i = \sum_{w_i > 0} w_i I(x \leq g_i).
\]

Then, we can have the pruning bound in Saigo et al. [2009] as

\[
(6) \leq \max \left\{ \sum_{w_i > 0} w_i I(x \leq g_i), - \sum_{w_i < 0} w_i I(x \leq g_i) \right\}.
\]

D Proof of Lemma 7

For applying coordinate descent to \(T(\theta(t))\) of (5), we first set only the \(j\)-th element of \(\theta\) free, i.e. \(\theta_j = z\), and fix all the remaining elements by the current value of \(\theta(t)\) as \(\theta_k = \theta(t)_k\) for all \(k \neq j\). Then, since (5) can be written as

\[
\arg \min_{\theta_1, \theta_2, \ldots} \left[ \sum_{k=0}^{\infty} \nabla f(\theta(t))_k (\theta_k - \theta(t)_k) + \frac{1}{2} \sum_{k=0}^{\infty} \sum_{l=0}^{\infty} H(t)_{kl}(\theta_k - \theta(t)_k)(\theta_l - \theta(t)_l) + \sum_{k=1}^{\infty} \lambda_1 |\theta_k| \right],
\]

we have the following univariate (one-dimensional) optimization problem for \(z\)

\[
T(\theta(t))_j = \arg \min_{z} \left[ \nabla f(\theta(t))_j (z - \theta(t)_j) + \frac{1}{2} H(t)_{jj} (z - \theta(t)_j)^2 + \lambda_1 |z| \right].
\]

(7)

On the other hand, for any \(a \geq 0\), completing the square gives us

\[
\arg \min_{z \in \mathbb{R}} \left[ az^2 + bz + c + \lambda |z| \right] = \begin{cases} 
-\frac{1}{2a}(b + \lambda) & b < -\lambda \\
-\frac{1}{2a}(b - \lambda) & b > \lambda \\
0 & -\lambda \leq b \leq \lambda.
\end{cases}
\]

(8)

Then, by rewriting (7) in the form of (8), we obtain the result in the lemma.

E Proof of Theorem 8

Let \(L'_\mu(y, \mu) := \partial L(y, \mu) / \partial \mu\). Then since \(\partial \mu(g_i, \theta(t))/\partial \theta(t)_k = I(x_k \subseteq g_i)\), we have

\[
\sum_{i=1}^{n} \frac{\partial L(y_i, \mu(g_i; \theta(t)))}{\partial \theta(t)_k} = \sum_{i=1}^{n} L'_\mu(y_i, \mu(g_i; \theta(t)))I(x_k \subseteq g_i).
\]

(9)

It will be noticed that this function is separable in terms of values of \(I_{G_n}(x_k)\) because each \(L'_\mu(y_i, \mu(g_i; \theta(t)))\) is constant after we substitute the current value for \(\theta(t)\). From Lemma 2, for an \(n\)-dimensional Boolean vector \(I_{G_n}(x)\), we have \(1(I_{G_n}(x_k)) \leq 1(I_{G_n}(x_j))\) for \(x_k \in T(x_j)\). Thus we can obtain the
Morishita-Kudo bounds defined in Lemma 6 for (9) by letting \( v := I_{\varphi_n}(x_k) \) and \( u := I_{\varphi_n}(x_j) \) because \( 1(v) \subseteq 1(u) \). In reality, the Morishita-Kudo bounds for (9) can be computed as
\[
\begin{align*}
L_j(t) &:= \sum_{i \in 1(I_{\varphi_n}(x_j))} \min\{L'_\mu(y_i, \mu(g_i; \theta(t))), 0\}, \\
T_j(t) &:= \sum_{i \in 1(I_{\varphi_n}(x_j))} \max\{L'_\mu(y_i, \mu(g_i; \theta(t))), 0\},
\end{align*}
\]
and these two bounds satisfy, for any \( k \) such that \( x_k \in T(x_j) \),
\[
L_j(t) \leq \sum_{i=1}^{n} \frac{\partial L(y_i, \mu(g_i; \theta(t)))}{\partial \theta(t)_k} \leq T_j(t).
\]

On the other hand, regarding the term \((\lambda_2 - H(t)_{kk})\theta(t)_k\), since
\[
(\lambda_2 - H(t)_{jj})\theta(t)_j = \begin{cases} 
0 & (\theta(t)_j = 0) \\
(\lambda_2 - H(t)_{jj})\theta(t)_j & (\theta(t)_j \neq 0)
\end{cases}
\]
and we already have the nonzero elements \( \theta(t)_j \) for \( j \) such that \( \theta(t)_j \neq 0 \), \( B_j(t) \) and \( B_j(t) \) are obtained just as the smallest and largest elements of the finite sets indexed by \( \{k \mid x_k \in T(x_j), \theta(t)_k \neq 0\} \). Thus we have \( B_j(t) \leq (\lambda_2 - H(t)_{kk})\theta(t)_k \leq B_j(t) \) directly by
\[
B_j(t) = \min \left( 0, \min_{l \in \{k \mid x_k \in T(x_j), \theta(t)_k \neq 0\}} (\lambda_2 - H(t)_{ll})\theta(t)_l \right),
\]
\[
B_j(t) = \max \left( 0, \max_{l \in \{k \mid x_k \in T(x_j), \theta(t)_k \neq 0\}} (\lambda_2 - H(t)_{ll})\theta(t)_l \right).
\]

It should be noted that as we see in Section F, the depth-first dictionary passing is an efficient way to obtain \( B_j(t) \) and \( B_j(t) \) during the traversal of enumeration tree \( T(G_n) \).

Now from the inequality (6), we have
\[
\left| \sum_{i=1}^{n} \frac{\partial L(y_i, \mu(g_i; \theta(t)))}{\partial \theta(t)_k} + (\lambda_2 - H(t)_{kk})\theta(t)_k \right| \leq \max\{T_j(t) + B_j(t), -L_j(t) - B_j(t)\}.
\]
The statement in Theorem 8 follows by combining this observation with Lemma 7.

## F Depth-First Dictionary Passing

In this subsection, we describe how to obtain \( B_j(t) \) and \( B_j(t) \) in Theorem 8 which are needed for computing \( \theta(t+1) \) from \( \theta(t) \). Since we already have the nonzero coordinates of \( \theta(t) \) for the index subset \( \{j \mid \theta(t)_j \neq 0\} \) at hand, these bounds are defined simply as the maximum and minimum of the finite candidates:
\[
B_j(t) := \min_{l \in \{k \mid x_k \in T(x_j)\}} (\lambda_2 - H(t)_{ll})\theta(t)_l = \min \left( 0, \min_{l \in \{k \mid x_k \in T(x_j), \theta(t)_k \neq 0\}} (\lambda_2 - H(t)_{ll})\theta(t)_l \right),
\]
\[
B_j(t) := \max_{l \in \{k \mid x_k \in T(x_j)\}} (\lambda_2 - H(t)_{ll})\theta(t)_l = \max \left( 0, \max_{l \in \{k \mid x_k \in T(x_j), \theta(t)_k \neq 0\}} (\lambda_2 - H(t)_{ll})\theta(t)_l \right).
\]
We can compute these values in a brute-force manner by checking \( x_j \subseteq x_k \) for all \( k \) such that \( \theta(t)_k \neq 0 \), but this is practically inefficient due to subgraph isomorphism test.
we write simply by $z$ the target traversal of passing for the search tree on the left (The output is shown on the right).

Build the mapping of (11) over all necessary $j$ deletes the pair $(j \mapsto x)$ with four operations of Get mapping as and it holds that $x$ such that $T$ next node of the depth-first traversal. In the pre-order operation, if we encounter $x$, we record, for every visited node $j$, the mapping $j \mapsto \{ \}$.

What we need instead is an efficient way to obtain, for each $j$, the index subset

$$\{ k \mid x_k \in T(x_j), \theta(t)_k \neq 0 \}. \quad (10)$$

Recall that in order to obtain $\{ j \mid \theta(t)_j \neq 0 \}$, we traverse the enumeration tree $T(G_n)$ to find all $j$ such that $T(\theta(t-1))_j \neq 0$ using the bounds in Theorem 8 (Figure 9). During this traversal, we can also record, for every visited node $x$, the mapping

$$j \mapsto \{ k \mid x_k \in T(x_j), \theta(t-1)_k \neq 0 \}. \quad (11)$$

If we have this mapping, then we can recursively obtain (10) for each $j$ as

$$\{ k \mid x_k \in T(x_j), \theta(t)_k \neq 0 \} = (\{ k \mid x_k \in T(x_j), \theta(t-1)_k \neq 0 \} \cup \{ k \mid x_k \in T(x_j), \theta(t-1)_k \neq 0 \}) \cap \{ k \mid \theta(t)_k \neq 0 \}. \quad (12)$$

Hence in what follows, we present the procedure, which we call depth-first dictionary passing, to build the mapping of (11) over all necessary $j$. Figure 7 shows an example of depth-first dictionary passing for the search tree on the left (The output is shown on the right).

First, we define $T_{visited}(G_n)$ as the tree consisting of all visited nodes during the depth-first traversal of $T(G_n)$ with pruning based on Theorem 8. Note that $T_{visited}(G_n)$ is a subtree of $T(G_n)$, and it holds that $x_j \in T_{visited}(G_n)$ for all $x_j$ such that $T(\theta(t))_j \neq 0$. We then define an auxiliary mapping as

$$h'(t) := \{(j \mapsto \{ k \mid x_k \in T(x_j), T(t)_k \neq 0 \}) \mid x_j \in T_{visited}(G_n)\} \quad (13)$$

with four operations of Get, Put, Keys, and DeleteKey: For a mapping $h : j \mapsto z$, Get returns the target $z$ of indicated $j$, which we write by $h[j]$. Put registers a new pair $(j \mapsto z)$ to $h$, which we write simply by $h[j] \leftarrow z$. Keys returns all registered keys in $h$ as $Keys(h) = \{ j \mid (j \mapsto z) \in h \}$. DeleteKey deletes the pair $(j \mapsto z)$ indicated by $j$ from $h$ as $DeleteKey(h, j) = h - \{ (j \mapsto z) \}$.

During the depth-first traversal of $T(G_n)$ at time $t$, we keep a tentative mapping, $h_{tmp}(t)$, to build $h'(t)$ of (13) at the end. We start with an empty $h_{tmp}(t)$. Then we update and pass $h_{tmp}(t)$ to the next node of the depth-first traversal. In the pre-order operation, if we encounter $x_j \in T_{visited}(G_n)$ such that $T(t)_j \neq 0$, we add this $j$ to all $h_{tmp}[i]$ for $i \in Keys(h_{tmp})$ by $h_{tmp}[i] \leftarrow h_{tmp}[i] \cup \{ j \}$. This informs all the ancestors that $T(t)_j \neq 0$. Then, we register $j$ itself to $h_{tmp}$ by initializing as $h_{tmp}[j] \leftarrow \{ \}$. In the post-order operation, if we have $h_{tmp}[j] \neq \{ \}$, then it implies that $x_j$

![Figure 7: An example of depth-first dictionary passing. Depth-first dictionary passing is a procedure to produce the mapping $h$ on the right after the depth-first traversal of the tree on the left. Each $h[j]$ is the indices of nonzero nodes below node $j$.](image)
Figure 8: The transition of tentative mapping $h_{tmp}$ in the depth-first dictionary passing in Figure 7. The order follows the depth-first order of traversal, and each shaded mapping at post-order operations constitutes the final output $h$. At the end of traversal, we have the mapping $h$ on the right in Figure 7 as the collection of all shaded mappings.

has descendants $x_k \in T(x_j)$ such that $T(\theta(t))_k \neq 0$ and $k \in h_{tmp}[j]$. Thus, this is an element of our target set $h'(t)$, and we finalize the $j$-th element $h'(t)[j]$ by setting $h'(t)[j] \leftarrow h_{tmp}[j]$. At this point, we also remove $j$ from $h_{tmp}$ by DELETE_KEY($h_{tmp}, j$) because $j$ cannot be an ancestor of any forthcoming nodes in the subsequent depth-first traversal after $x_j$. In this way, we can obtain $h'(t)$ defined in (13) at the end of traversal.

The depth-first dictionary passing finally gives us the mapping (10) as

$$h(t) := \{ (j \mapsto \{ k \mid x_k \in T(x_j), \theta(t)_k \neq 0 \}) \mid x_j \in T_{visited}(G_n) \}$$

from $h'(t-1), h(t-1)$ and the nonzero coordinates of $\theta(t)$ by using a recursive formula of (12). This also gives $B_j(t)$ and $B_j(t)$ in Theorem 8 which are needed for computing $\theta(t+1)$ from $\theta(t)$.

Figure 8 shows an example of building $h_{tmp}$ (each key-list table) and registration to $h'$ (shown as the shaded colored key-list pairs) for the case of Figure 7. First, we start a depth-first traversal from node 1. In the pre-order operation at node 1, we just register key 1 to $h_{tmp}$ as $h_{tmp}[1] \leftarrow \{ \}$. The next node 2 is shaded, which indicates that $T(\theta(t))_2 \neq 0$, then we inform node 2 to all ancestors by $h_{tmp}[1] \leftarrow h_{tmp}[1] \cup \{ 2 \}$ and also initialize as $h_{tmp}[2] \leftarrow \{ \}$. At node 3 and 4, we just add $h_{tmp}[3] \leftarrow \{ \}$ and $h_{tmp}[4] \leftarrow \{ \},$ but at node 4, we also need the post-order operation of registering the mapping of 4 to $h'$ and then removing key 4. As a result, at the shaded node of 5, we have keys of 1, 2 and 3 in $h_{tmp}$. In the pre-order operation at node 5, we add 5 to all of these as $h_{tmp}[1] \leftarrow h_{tmp}[1] \cup \{ 5 \}, h_{tmp}[2] \leftarrow h_{tmp}[2] \cup \{ 5 \}$ and $h_{tmp}[3] \leftarrow h_{tmp}[3] \cup \{ 5 \}$. In this way, we keep $h_{tmp}$, and obtain $h'$ at the end (as the set of the shaded key-list pairs in tables of Figure 8).

G Numerical Case Study: L1-Penalized Logistic Regression

As a case study of our proposed framework, we take an example of L1-penalized logistic regression. Logistic regression is a fundamental model for classification, and gives a baseline understanding
Figure 9: A schematic example of corresponding nonzero coordinates of \( \theta(t) \), \( T(\theta(t)) \), \( d(t) \), and \( \theta(t+1) \) in the enumeration tree. The nonzero coordinates of \( \theta(t+1) \) is computed from those of \( \theta(t) \) through \( T(\theta(t)) \) and \( d(t) \), and identifying \( T(\theta(t)) \neq 0 \) is the determining step. Since \( d(t) = T(\theta(t)) - \theta(t) \), we have \( \{ j \mid d(t)_j \neq 0 \} \subseteq \{ j \mid \theta(t)_j \neq 0 \} \cup \{ j \mid T(\theta(t))_j \neq 0 \} \). Since \( \theta(t+1) = \theta(t) + \alpha(t)d(t) \), we can also see \( \{ j \mid \theta(t+1)_j \neq 0 \} \subseteq \{ j \mid \theta(t)_j \neq 0 \} \cup \{ j \mid T(\theta(t))_j \neq 0 \} \).

about the linear separability of the data. The proposed algorithm is, to our knowledge, the first exact method to directly learn logistic regression over all subgraph features under elastic-net regularization (a possibility of generalizing a boosting-based approach was already discussed Nowozin [2009]).

We numerically examine the properties and performance of L1-penalized logistic regression (L1-LogReg) by using our algorithm shown in Figure 2, which is, in Problem 2, \( \lambda_2 = 0 \) and

\[
L(y, \mu) = y \log(1 + \exp(-\mu)) + (1 - y) \log(1 + \exp(\mu))
\]

for \( y \in \{0, 1\} \). Our results are compared to those of two existing methods mentioned in Sections 1 and 2: Adaboost Kudo et al. [2005] and LPBoost Saigo et al. [2009]. In this paper we focus on optimization, and so rather than predictive performance, our interests are in comparing

- **Learning curves**: Rate of convergence should be higher at each iteration till the convergence to the optimal solution.
- **Selected subgraph features**: Selected features should be more understandable in terms of knowledge discovery. For example, they should be larger and the number of selected features should be smaller if the performance is equivalent.

According to the example of Yun and Toh [2011], we set

\[
H(t)_{jj} := \min \{ \max \{ \nabla^2 f(\theta(t))_{jj}, 10^{-10} \}, 10^{10} \}
\]

and

\[
\sigma = 0.1, c = 0.5, \gamma = 0, \alpha_{\text{init}}(0) = 1,
\]

\[
\alpha_{\text{init}}(t) = \min \left\{ \frac{\alpha(t-1)}{c^5}, 1 \right\}, v(t) = 0.9.
\]

For systematic evaluation on binary classification, we newly developed a random graph generator, which, from a given seed-graph pool (a set of subgraph features), generates a graph by probabilistically combining small graphs from the pool as described in the subsequent section. In this manner, we can control the property, size, and number of graph samples as well as the training and test error of the model generated from the data. We generated two sets of graphs which can be shown...
Figure 10: The data matrix for two graph sets (i) and (ii) for binary-classification benchmarking

Graph set (i)

\[ P(I_{ij} = 1) = p_1 \]

Graph set (ii)

\[ P(I_{ij} = 1) = p_2 \]

\[ L_{ij} = 0 \text{ or } 1 \]

Figure 11: The scheme and settings of data generation for numerical evaluation. We generated 100 sets of 1,000 random graphs consisting of 500/500 graphs for (i)/(ii) in Figure 10, respectively, sharing the same generative rule.

as a design matrix in Figure 10. A big advantage of this manner of evaluation is that we know “true” discriminative subgraph features embedded in given (observed) graphs, and we can compare the selected subgraph features by each learning algorithm to these unobserved “true” features. We primarily employ this approach since it is difficult to evaluate these properties of selected features by real datasets with unknown background.

L1-LogReg was implemented by C++ entirely from the scratch. For Adaboost and LPBoost, we used the implementations in C++ of Saigo et al. [2009], the source code being obtained from the first author's website\(^3\), since this is faster than other implementations. We also set minsup=1 and maxpat=∞ in the original code, to run Adaboost and LPBoost over all possible subgraph features.

H Random Graph Generation of Binary Classification Task

Figure 12 presents the procedure for generating two set of graphs\(^4\). These two sets correspond to graph sets (i) and (ii) in Figure 10 which are generated by a consistent probabilistic rule: each graph in (i) includes each subgraph feature in subgraph set (A) with a probability of \( p_1 \) and similarly each feature in (B) with \( q_1 \); Each graph in (ii) includes each subgraph feature in (A) with a probability of \( p_2 \) and each feature in (B) with \( q_2 \) (as summarized on the right in Figure 10), where a seed-graph pool of (A) and (B) are also generated randomly. When we set \( p_1 > q_1 \) and \( q_2 > p_2 \), we can

\(^3\)http://www.bio.kyutech.ac.jp/~saigo/publications.html

\(^4\)In the pseudo code of Figure 12, the subroutine \texttt{MinDFSCode}(g) computes a canonical representation of graph \( g \) called the minimum DFS code (See Yan and Han [2002] for technical details).
assume $V$ and $W$ dominant subgraph features that can discriminate between (i) and (ii). By using these two sets (i) and (ii) as positive and negative samples respectively, we define a systematic binary classification task with control parameters $V, W, N, M, p_1, p_2, q_1, q_2$ and the Poisson mean parameter. This procedure, which was inspired by Kuramochi and Karypis [2004], first generates $V + W$ random graphs as $X := \{x_1, x_2, \ldots, x_{V+W}\}$ for a seed-graph pool, and then generates $N + M$ output graphs by selecting and combining these seed graphs in $X$ with the probabilistic rule: as indicated in Figure 10, the probability that output graph $i$ contains subgraph feature $j$ or not ($I_{i,j} = 1$ or $0$) is determined block by block. To generate each output graph, the selected seed subgraphs are combined by the procedure $\text{combine}$ in Figure 13 which finally produces the output graphs (i) and (ii) having the data matrix in Figure 10. The mean parameter for the Poisson distribution is set to 3, and the number of node and edge labels to 5 throughout the numerical experiments.

Algorithm:

1. Generate $V + W$ Poisson random numbers $a_1, a_2, \ldots, a_{V+W}$ (resample if $a_i < 2$);
2. Initialize dictionary $d$;
3. Initialize seed-graph pool $X \leftarrow \{}$;
4. for $i = 1, 2, \ldots, V + W$ do
   1. Generate a graph $g$ with two node and an edge between them. (node and edge are labeled at random);
   2. repeat $a_i$ times do
      1. Select a node $\alpha$ from $g$ at random;
      2. Select one of the two choice at random (to ensure the connectivity of $g$);
         i) Add an edge from $\alpha$ to a node of $g$ that is not adjacent to $\alpha$;
         ii) Add a new node $\alpha'$ to $g$ and an edge between $\alpha$ and $\alpha'$;
   3. if $\text{MinDFSCode}(g)$ in $d$ then
      1. continue;
   else
      1. Register $\text{MinDFSCode}(g)$ to $d$;
      2. $X \leftarrow X \cup \{g\}$;
5. end
6. end
7. Set the first $V$ graphs in $X$ as $A$, and the last $W$ graphs in $X$ as $B$;
8. repeat $N$ times to generate $N$ positive samples do
   1. Select $S_V \subseteq A$ with $P(x_i \in S_V) = p_1$ for $x_i \in A, i = 1, 2, \ldots, V$;
   2. Select $S_W \subseteq B$ with $P(x_i \in S_W) = q_1$ for $x_i \in B, i = 1, 2, \ldots, W$;
   3. $g \leftarrow \text{combine}(S_V \cup S_W)$;
   4. Add $g$ to a set of positive examples;
9. end
10. repeat $M$ times to generate $M$ negative samples do
   1. Select $S_V \subseteq A$ with $P(x_i \in S_V) = p_2$ for $x_i \in A, i = 1, 2, \ldots, V$;
   2. Select $S_W \subseteq B$ with $P(x_i \in S_W) = q_2$ for $x_i \in B, i = 1, 2, \ldots, W$;
   3. $g \leftarrow \text{combine}(S_V \cup S_W)$;
   4. Add $g$ to a set of negative examples;
11. end

Figure 12: The procedure for class-labeled random graph generation.

\[\text{Kuramochi and Karypis [2001]}\] proposed another way to combine the seed graphs to maximize the overlapped subgraphs. In either way, the procedure in Figure 12 produces the data matrix as in Figure 10.
**Function: combine**($x_1, x_2, x_3, \ldots$)

\[
g' \leftarrow x_1;
\]

\[
\text{foreach} \ g_i \text{ in } x_2, x_3, \ldots \text{ do}
\]

\[
\quad \text{Select a node of } g' \text{ and a node of } g_i \text{ at random;}
\]

\[
\quad \text{Add an edge between them, and replace } g' \text{ by the combined graph;}
\]

\[
\text{return } g'
\]

Figure 13: The algorithm for combining several graphs into a single connected graph.

## I Evaluating Learning Curves

We investigated the convergence property by the learning curves of the three methods on the same dataset. Figure 11 shows a schematic manner of generating an evaluation dataset by using the random graph generator. First we generated 100,000 graphs with the fixed parameters of $p_1, p_2, q_1,$ and $q_2$, and then divided them into 100 sets, each containing 1,000 graphs (500 positives and 500 negatives). We estimated the expected *training error* by computing each training error of model $i$ with data set $i$ that was used to train the model, and averaging over those 100 values obtained from 100 sets. Since all 100 sets share the same probabilistic rule behind their generation, we estimated the expected *test error* by first randomly choosing 100 pairs of set $i$ and model $j$ ($i \neq j$), and computing the test error of the model $i$ with data set $j$ that was not used to train the model, and averaging over those 100 values obtained from 100 pairs. We used the same fixed 100 pairs for evaluating all three methods of Adaboost, LPBoost, and L1-LogReg. For the three methods, we computed the training and test errors at each iteration, which gave us the learning curves of the three methods regarding the training error as well as the test error. We used four different settings for control parameters, which are shown in Figure 11. The setting 2 corresponds to the RAND dataset referred in the main paper. Table 4 shows the statistics on the datasets generated by these four settings. Using each set of 1,000 graphs, we trained the model by each of Adaboost, LPBoost, and L1-LogReg.

It should be noted that when we draw an averaged learning curve, we need to perform 100 trainings and 100 testings for 1,000 graphs *at each iteration*. For example, the learning curves of 150 iterations require $150 \times 100$ trainings and $150 \times 100$ testings for 1,000 graphs.

Figure 14 to 17 show the learning curves for all four settings with fixing the tuning parameters of learning algorithms at the best parameters. Figure 18 to 21 show the changes of learning curves by different values of the tuning parameters.

<table>
<thead>
<tr>
<th>data set</th>
<th># graphs</th>
<th># edges</th>
<th># nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>max</td>
<td>min</td>
</tr>
<tr>
<td>setting-1</td>
<td>100,000</td>
<td>193</td>
<td>49</td>
</tr>
<tr>
<td>setting-2 (RAND)</td>
<td>100,000</td>
<td>219</td>
<td>32</td>
</tr>
<tr>
<td>setting-3</td>
<td>100,000</td>
<td>246</td>
<td>33</td>
</tr>
<tr>
<td>setting-4</td>
<td>100,000</td>
<td>221</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 4: Statistics on the generated datasets.
J Correctness of implementations: How should we debug them?

It is generally hard to make sure that a learning algorithm for graphs works correctly. There is no easy way to visualize a large set of graphs and subgraph features in a consistent form. This originates from the difficulty to visually find out graph isomorphism and subgraph isomorphism. Thus it would be an important problem how to check the correctness of the implemented algorithm. We know that the complete check is impossible in practice, and thus here we simply emphasize the importance of careful double check with systematically generated dataset.

For example, our implementation, fully written in C++ from the scratch, is based on the gSpan algorithm to implicitly build and traverse an enumeration tree $T(G_n)$. We ensure that our implementation of gSpan works in the exactly same way to three other existing implementations with randomly generated graphs of various types\textsuperscript{6}. Similarly, we ensure that our implementation of Figure 2 works as expected using a simple double check as follows. When we limit the enumeration tree down to the $i$-th level, it is relatively easy to enumerate all bounded-size subgraphs in $X(G_n)$ to a certain level of $i$. Even if we can check only up to $i = 5$ or so, it should be valuable to check the identity to the case where we first generate the data matrix explicitly and apply the block coordinate gradient descent to the data matrix. Thus, for this level-limited enumeration tree up to the $i$-th level, we can explicitly generate the data matrix in the form of Figure 6. Applying the block coordinate gradient descent to this matrix is supposed to produce exactly the same coefficients and function values at each iteration as those which can be obtained when we limit $T(G_n)$ by size. Using the original implementation of block coordinate gradient descent by Yun and Toh [2011], we double-check that the nonzero coefficient values and their indices are exactly the same at each iteration as our implementation with the level limitation. In addition, we double-check by applying glmnet Friedman et al. [2010] to the data matrix and confirming if the values of the objective function at convergence are the same (because both glmnet and block coordinate gradient descent can be applied to the same problem of 1-norm penalized logistic regression). Note that glmnet in R first automatically standardizes the input variables, and thus we need to ignore this preprocessing part.

\textsuperscript{6}We developed this faster implementation in the previous project. See Takigawa and Mamitsuka [2011] for details.
Figure 14: Learning curves for setting-1 (average over 100 trials).

Figure 15: Learning curves for setting-2, RAND (average over 100 trials).
Figure 16: Learning curves for setting-3 (average over 100 trials).

Figure 17: Learning curves for setting-4 (average over 100 trials).
Figure 18: Average learning curves for different parameters (setting-1).

Figure 19: Average learning curves for different parameters (setting-2, RAND).
Figure 20: Average learning curves for different parameters (setting-3).

Figure 21: Average learning curves for different parameters (setting-4).
References


Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM)*, pages 721–724, Washington, DC, USA, 2002.
