NAIST-IS-DD123456

Doctoral Dissertation

Incremental and Parallel Learning Algorithms for Data Stream Knowledge Discovery

Lei Zhu

February 1, 2018

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Lei Zhu

Thesis Committee:

Professor Kazushi Ikeda	(Supervisor)
Professor Keiichi Yasumoto	(Co-supervisor)
Professor Paul Pang	(Unitec)
Associate Professor Takatomi Kubo	(Co-supervisor)

Incremental and Parallel Learning Algorithms for Data Stream Knowledge Discovery*

Lei Zhu

Abstract

Incremental and parallel are two capabilities for machine learning algorithms to accommodate data from real world applications. Incremental learning addresses streaming data by constructing a learning model that is updated continuously in response to newly arrived samples. To solve the computational problems posed by large data sets, parallel learning distributes the computational efforts among multiple nodes within a cloud or cluster to speed up the calculation. With the rise of BigData, data become simultaneously large scale and streaming, which is the motivation to address incremental and parallel incremental (PI) learning in this work.

This research first considers the incremental learning alone, in the graph max-flow/min-cut problem. An augmenting path based incremental max-flow algorithm is proposed. The proposed algorithm handles graph changes in a chunking manner, updating residual graph via augmentation and de-augmentation in response to edge capacity increase, decrease, edge/node adding and removal. The theoretical guarantee of our algorithm is that incremental max-flow is always equal to batch retraining. Experiments show the deterministic computational cost save (i.e., gain) of our algorithm with respect to batch retraining in handling graph edge adding.

The proposed incremental max-flow is then applied to upgrade an existing batch semi-supervised learning algorithm known as graph minicuts to be incremental. In batch graph minicuts, a graph is learned from input labeled and

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, NAIST-IS-DD123456, February 1, 2018.

unlabeled data, and then a min-cut is conducted on that graph to make the classification decision. In the proposed modification, the graph is updated dynamically for accommodating online data adding and retiring. Then the proposed incremental max-flow algorithm is adopted to learn min-cut from the resulting non-stationary graph. Empirical evaluation on real world data reveals that the proposed algorithm outperforms state-of-the-art stream classification algorithms.

In the incremental max-flow, the training speed is not satisfactory when the data set is huge. A straightforward solution is to combine parallel data processing with incremental learning. Previously, parallel and incremental learning are often treated as two separate problems and solved one after another. Alternatively in this work, these two learning problems are solved in one process (i.e., PI integration).

To simplify the learning, this research considers a base model in which incremental learning can be implemented by merging knowledge from incoming data and parallel learning can be performed by merging knowledge from simultaneous learners (i.e., in knowledge mergeable condition). As a result, this work develops a parallel incremental wESVM (weighted Extreme Support Vector Machine) algorithm, in which the parallel incremental learning of the base model is completed within a single process of knowledge merging. Specifically, the wESVM is reformulated such that knowledge from subsets of training data can be merged via simple matrix addition. As such, the proposed algorithm is able to conduct parallel incremental learning by merging knowledge from data slices arriving at each incremental stage. Both theoretical and experimental studies show the equivalence of the proposed algorithm to batch wESVM in terms of learning effectiveness. In particular, the algorithm demonstrates desired scalability and clear speed advantages to batch retraining.

In the field of data stream knowledge discovery, this work investigates incremental machine learning and invents a wESVM-based parallel learning and incremental learning integrated system. The limitation of this work is that PI integration applies only to models that satisfy the knowledge mergeable condition. Future work should investigate how to release this constraint and expand PI integration to other models such as SVM and neural network.

Keywords:

Incremental learning, Max-flow, Min-cut, Augmenting Path, Graph minicuts, Parallel learning, Parallel incremental learning, Extreme SVM

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

The goal of machine learning can be stated as to build a computational model from what has been observed in the past [3]. In the early days, research studies and practices in machine learning focused on batch learning, in which the complete data set is available at once to the algorithm that generates a decision model. The assumption behind batch learning is that samples are generated randomly according to a stationary probability distribution. The learning objective is to estimate this distribution with the samples that are available.

Fast developments in information and communication technologies, data collection and processing methods have introduced dramatic changes. Currently, data are often presented in continuous streams, representing the current state of a stationary or non-stationary environment [4]. To learn from these data streams, incremental learning constructs a learning model that is updated continuously in response to newly arrived samples [5].

For the incremental model, the input samples $x_1, x_2, \ldots, x_n, \ldots$ arrive sequentially, item by item (known as online learning) or set by set (known as chunk learning) [4]. There are two types of incremental models:

1. Insert only model: only adding samples is allowed;

2. Insert-delete model: samples can be both added and retired.

The assumption behind insert only models is that samples are generated sequentially and randomly according to a stationary probability distribution. Thus the incremental learning of insert only models is to accumulate samples over time for improving the estimation of the underlying distribution [6]. On the other hand, insert-delete models assume a shifting probability distribution, and only

the most recent samples are useful for estimating the current distribution [7]. Thus the model is updated by accommodating knowledge from the newly arrived samples, while discarding knowledge from the samples that are no longer up-to-date.

In the literature, the activity of acquiring knowledge from new samples is also known as incremental learning (in the narrow sense), as distinguished from decremental learning, which is the operation of retiring knowledge from old samples.

Due to recent developments in data collection technologies, data sets are becoming increasingly larger. Processing these large scale data sets poses considerable difficulties, especially for computationally expensive machine learning algorithms. Parallel processing is an attractive technique for scaling up and speeding up algorithms, and this also applies for machine learning algorithms. Parallel learning accelerates the learning procedure by distributing the large computational efforts among a set of nodes within a cloud or cluster [8].

1.2 Research Roadmap

Incremental learning (IL) has been extensively studied in the literature and many approaches have been applied to achieve incremental capability. In general, existing IL models can be summarized into two categories in terms of the approach to deriving incremental capability: 1) model updating in which the current model is modified to incorporate the knowledge from newly appeared data samples, and 2) model ensemble in which a new model is built based on a chunk of incoming data and the knowledge is combined via an ensemble of individual models. In real-world application, incremental learning plays a major role in data analytics, big data processing, robotics, image processing, etc. [9].

In the era of BigData, data are being produced in variety of structured, semi-structured, and unstructured forms, and being presented as a mixture of numerical records, graphs, XML documents, text files, images, audio, video, etc. [10] [11]. Among all types of data, it is worth noting that graph data have been been occurring more frequently, representing communication network, power supply/consumption network, websites link structure, and users linkage in social network [12].

For graph modelling, max-flow is a fundamental model for solving many complex graph problems such as maximum cardinality bipartite matching and minimum path cover in directed acyclic graph [12]. Also, max-flow has been employed in variety of applications such as network bottleneck identification, energy minimization in computer vision, and graph-based clustering. For incremental learning of max-flow, existing algorithms focus on the push relabel mechanism which involves a great amount of operations in neighbour search, flow push, and node relabel, thus push relabel has higher empirical computational complexity. Augmenting path, the other track of max-flow, is still open for exploration.

For BigData processing, parallel incremental max-flow is a straightforward solution. The difficulty of parallelizing incremental max-flow lies at: 1) For augmenting path based incremental max-flow, it is an iterative path searching process followed by path de-augmentation or augmentation, where there is no existing solutions to parallelize the search for multiple edge disjoint paths; and 2) For push relabel based incremental max-flow, it is computationally very expensive to identify neighbour disjoint active nodes for parallel push and relabel. In general, the difficulty here is lack of sub-problems that one does not affect the other. In other words, we are not able to merge max-flow knowledge from sub-graphs.

For the parallelization of model-updating-based IL, we consider the following scenario: given a base model whose knowledge can be merged with that of other model, then IL can be implemented by merging knowledge from incoming datasets (each dataset generates one model), and parallel learning can be also performed by merging knowledge from a set of independent learners that work on different data slices.

More interestingly, in this scenario both parallel and incremental learning are achieved via an unified computing process. In other words, parallel and incremental learning are integrated into one system, a parallel incremental (PI) integrated system.

The advantage of a PI integrated system lies at:

1. The system is simplified as one data processing routine instead of two, for parallel and incremental function respectively; and

2. The system supports better distributed learning environment, because knowledge mergable condition ensures that the learning can be carried out with

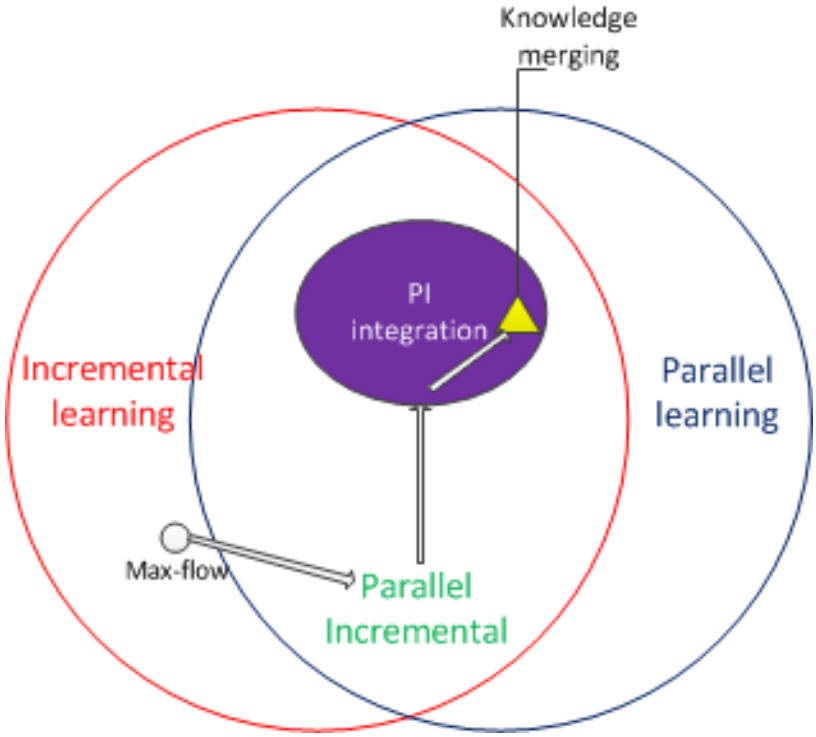


Figure 1.1: The research roadmap of this thesis.

no restriction on time and location.

As such, developing PI integrated algorithms is a significant work that we are going to address in this thesis. As a summary, Figure 1.1 presents the research roadmap of this thesis.

1.3 Contribution

Our contributions in this thesis are summarized as follows:

1. We derive an incremental max-flow algorithm based on augmenting path algorithm. The proposed algorithm is capable of handling all possible graph changes, with a theoretical guarantee that the incremental max-flow equals always to batch retraining. The proposed algorithm has deterministic computational cost savings with respect to batch retraining in handling graph

edge adding, and gives much faster converging speed compared to incremental push relabel.

2. We apply our incremental max-flow algorithm to upgrade an existing batch semi-supervised learning algorithm know as graph minicuts to be incremental. The proposed incremental graph mincuts is capable of accommodating both addition and retirement of labelel and unlabeled samples. The proposed system is found to be less sensitive to the amount of labelled data (in terms of the ratio to the whole training data) as compared to K-NN, SVM, and SVM self-training.

3. We raise PI integration (parallel and incremental integrated learning), a new concept of parallel incremental learning. PI integration deals with both parallel and incremental learning as one problem and solves the problem by applying one characteristic calculator (i.e., base model). The advantage of PI integration is that it simplifies the design and implementation of parallel incremental algorithms, and it suits real world distributed learning environments.

4. We propose a new concept of knowledge mergeable condition to judge if a learning model can be used as the base model of PI integration.

5. We develop the first PI integration algorithm based on wESVM (weighted Extreme Support Vector Machine). The proposed parallel incremental wESVM always gives the exactly same learning result as batch retraining, it scales well in response to both number of nodes and data size, and our incremental learning has clear speed advantage to batch learning.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives a comprehensive review the of max-flow problem and existing batch and incremental max-flow algorithms. Chapter 3 presents the proposed augmenting path based incremental max-flow algorithm, including algorithm derivation and evaluation. In Chapter 4, the proposed incremental max-flow algorithm is applied to upgrade an existing

batch semi-supervised learning algorithm, graph minicuts, to be incremental. Chapter 5 identifies a family of knowledge mergeable algorithms. In Chapter 6, we derive a PI integrated learning system. Chapter 7 contains the conclusions drawn from this thesis.

2 Max-flow Problem and Max-flow Algorithms

2.1 Introduction

Incremental max-flow is the first problem we address in this work. In this chapter, we first give an overview of max-flow, starting with the definition of the max-flow problem, followed by applications of max-flow. Then we give a comprehensive review of existing max-flow algorithms including batch and incremental ones.

2.1.1 Definitions and Problem Statement

A directed graph $G = \langle V, E, C \rangle$ is defined by a set of nodes V, a set of directed edges E, and a edge capacity function $C : E \to \mathbb{R}^+ \cup \{0\}$ which maps each edge (u, v) to a non-negative capacity value $C(u, v)$. In the context of max-flow/min-cut, a graph has two special nodes: source s and sink t, which is the start and end point of flow respectively.

A flow on G is a real valued function $f()$ if the following conditions are satisfied:

$$f(u, v) = -f(v, u), \quad \forall (u, v) \in V \times V; \tag{2.1a}$$

$$f(u, v) \leq C(u, v), \quad \forall (u, v) \in V \times V; \tag{2.1b}$$

$$\sum_u f(u, v) = 0, \quad \forall v \in V \setminus \{s, t\}. \tag{2.1c}$$

In literature 2.1a, 2.1b and 2.1c are known as flow antisymmetry, edge capacity and mass balance constraint, respectively. Let net flow $F = \sum_{u \in V} f(u, t)$ be the summation of flows into sink t. Then, the max-flow problem is to determine a flow from s to t with the maximum net flow F.

A s/t cut is a partitioning of the nodes in the graph into two disjoint subsets S and T, such that $s \in S$, $t \in V$ and $S \cup V = \emptyset$. For simplicity, the s/t cut is referred to as cut in the rest of this work. The cost of a cut $C(S,T)$ is defined as the total capacity of all boundary edges (u,v) where $u \in S$ and $v \in T$. The min-cut problem is to find a cut that has the minimum cost among all cuts.

According to the theorem of Ford and Fulkerson [13, 14], a max-flow from s to t saturates a set of edges in the graph dividing the nodes into two disjoint parts $\{S,T\}$ corresponding to a min-cut. Also, the flow value of the max-flow is equal to the cost of the min-cut. Thus, the min-cut and max-flow problems are equivalent, and the min-cut is normally solved by finding a max-flow.

Incremental max-flow is the incremental learning of max-flow. Given an initial graph and corresponding max-flow, which is stored in various forms according to the base algorithm (e.g., residual graph for augmenting path based algorithms, residual graph plus node distance labeling for push relabel based algorithms), the incremental max-flow subjects to update max-flow in response to graph changes in order to obtain the max-flow result for the updated graph. The advantage of incremental max-flow is to take advantage of the existing max-flow result and only learning from the graph changes to save computational costs in comparison with learning max-flow from scratch.

2.1.2 Max-flow Applications

As various real world problems can be abstracted into max-flow problems, or equivalent problems such as min-cut, there are vast applications of max-flow. The bottleneck identification for a city traffic network [15] is a known max-flow application in which a traffic network in a city is abstracted into a road graph. Applying max-flow computation on the road graph, a road with its total capacity taken to carry flow is considered as a bottleneck. Similarly, bottleneck identification for a power system has been used for computing a power system security index [16] in which power supply links are represented as edges, and factories or towns are denoted as nodes. Max-flow is also applied in a wireless mobile environment to optimize the association between wireless clients to access points by maximizing the traffic flow to clients [17] [18] [19].

Max-flow/min-cut has also been widely used in computer vision, particularly

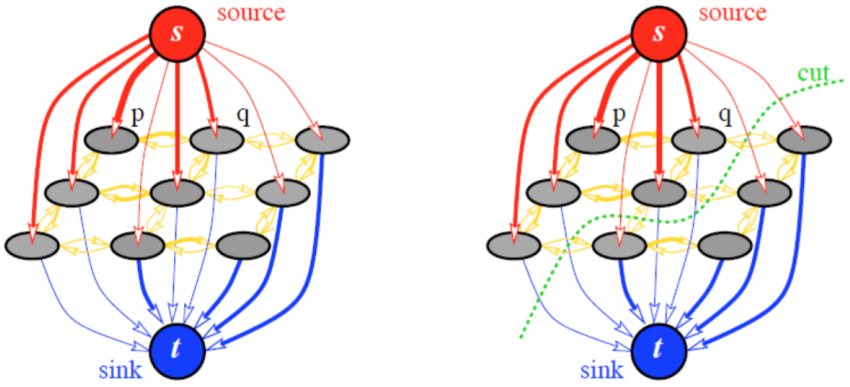


Figure 2.1: An example of graphs min-cut used for computer vision applications. Left is the graph constructed and the right is the min-cut result. [1]

for energy minimization problems such as image segmentation [20] [21] [22] [23], restoration [24] [25], stereo image processing [26] [27] [28] [29], shape reconstruction [30] [31] [32] [33], object recognition [34] [35] [36], augmented reality [37] [38] [39], and texture synthesis [40] [41]. For a comprehensive overview of max-flow/min-cut applications in computer vision, please refer to [1]. The graphs used for these typically have a specific structure, and the goal is to assign one of two labels to every pixel in an image.

The graph in these applications is generated based on a regular 2D grid where each node, except the source and the sink, represents an image pixel. The adjacent nodes are usually pixels that are adjacent in the corresponding image, as shown in the left part of Figure 2.1. The source and sink nodes, which are also known as the terminal nodes, are two special nodes. They represent the two possible labels which can be assigned to all the other nodes (i.e., pixels in the image). The terminal nodes are connected to all the other nodes with varying capacities [29] [42] [43].

The edge capacities between adjacent non-terminal nodes are set according to a penalty for discontinuity between the pixels associated with these nodes. They represent how well a label from one pixel would continue into the adjacent pixel. The edge capacities between the terminal nodes and the non-terminal nodes are set according to a penalty for assigning the corresponding label to the pixel. When this graph is generated, the $s - t$ min-cut on the graph is used for label

assignment. As shown in the right side of Figure 2.1, after the min-cut, nodes (pixels) connected with the source node s are labeled as one class and the others are labeled as another class.

Similar to its application in computer vision, min-cut is also applied to find clusters in various types of networks, which can be easily mapped to a graph, such as biological and sociological networks [44]. The Information Bottleneck (IB) method [45] [46], developed based on rate distortion theory [47], is representative of this track. It has been adopted in applications such as clustering word documents and gene expressions [48] [49], identifying modularity in synthetic and natural networks [50], classifying galaxies by their spectra formation [51], and community detection in social networks [52] [53] [54].

The applications of incremental max-flow, on the other hand, focus on learning in the changing environment. Grundmann et al. adopt incremental max-flow for image segmentation in continuous video frames [55]. A video can be seen as a series of images, so that each frame has little difference in comparison with the last frame. In this case, the graph constructed for image segmentation also has minor differences between neighboring frames. Thus the incremental learning of graph G_{i+1} based on the max-flow result of graph G_i, better suits this scenario in comparison with learning G_{i+1} from scratch. In this sense, incremental max-flow can be naturally applied to various computer vision tasks introduced before, for learning from video frames. Moreover, in the works [56] and [57], incremental max-flow is employed for graph-based clustering in dynamic settings, in which the graph represents real-world data that are changing continuously over time.

2.2 Batch Max-flow Algorithms

In solving max-flow problems, existing algorithms fall into two categories, namely preflow push method and augmenting path method [12]. Existing batch max-flow algorithms are introduced in this section.

2.2.1 Augmenting Path

Augmenting path [13] algorithm stores information about the distribution of the current $s-t$ flow F among the edges of G using a residual graph $R = (V, E, R)$.

The topology of R is identical to G (i.e., G and R share the same V and E), but $R(e)$, the capacity of edge e in R, reflects the residual capacity of the same edge in G given the amount of flow already in the edge. At the initialization, there is no flow from the source to the sink ($F = 0$) and edge capacities in the residual graph R are equal to the original capacities in G (i.e. $R(e) = C(e), \forall e \in E$).

Augmenting path algorithm is an iterative procedure of the following two steps:

1) Find $s - t$ path using Breadth-First Search (BFS). The resulting path P is a set of edges with positive residual capacity laid end to end connecting s to t, such as $P = \{(s, u), (u, v), (v, t) \mid R(s, u) > 0, R(u, v) > 0, R(v, t) > 0\}$.

2) Augment the $s - t$ path found above. We firstly find the maximum amount of flow that can go through this path P, which is termed augmentation value and denoted as Δ_P in the rest of this work. As it is a bottleneck problem here, Δ_P can be calculated as the minimum residual capacity of the whole path $\Delta_P = min(R(u, v) \mid \forall (u, v) \in P)$. Next, we send Δ_P flow through path P in R as,

$$\begin{aligned} R(u, v) &= R(u, v) - \Delta_P, \forall (u, v) \in P \\ R(v, u) &= R(v, u) + \Delta_P, \forall (u, v) \in P \end{aligned} \tag{2.2}$$

The above two steps are iteratively executed until no more $s - t$ paths can be found. Figure 2.2 gives an example of max-flow search through the augmenting path algorithm.

2.2.2 Boykov-Kolmogorov (BK) Algorithm

The BK algorithm [1] is developed based on the augmenting path method. It has been extensively utilized by the computer vision community, since it is superior in practice to others on many vision instances [58].

A difference from standard augmenting path algorithm, which finds the $s - t$ path via a single BFS search starting from source node s, the BK algorithm seeks the $s - t$ path in a bi-directional manner.

The BK algorithm maintains two non-overlapping search trees $\mathcal{S}$ and $\mathcal{T}$ rooted at s and t respectively. In $\mathcal{S}$, all edges from parents to children have positive residual capacity; and in $\mathcal{T}$, all edges from children to parents have positive residual capacity. Nodes in G but not in $\mathcal{S}$ or $\mathcal{T}$ are termed as free nodes, and

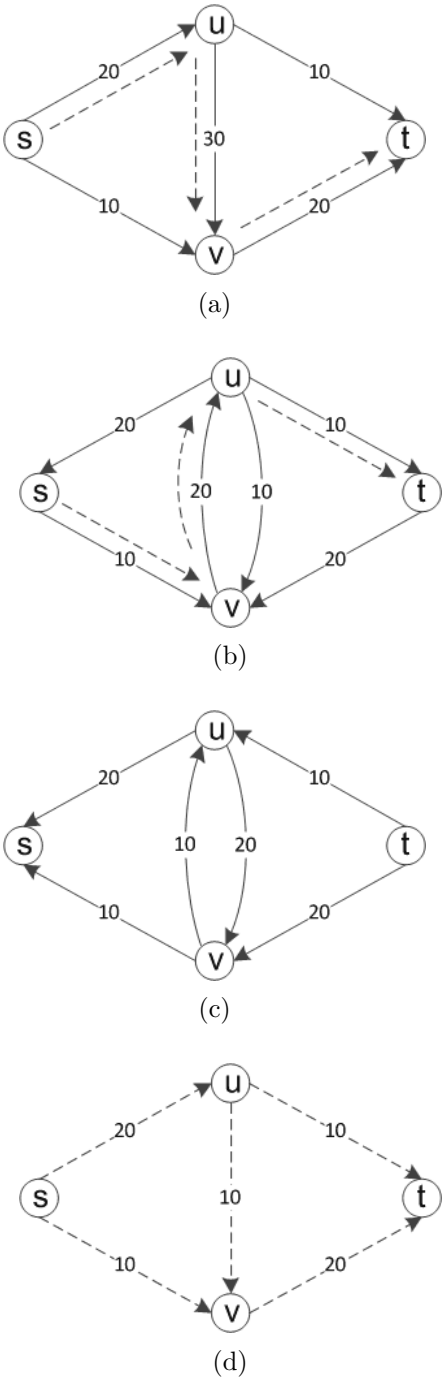


Figure 2.2: An example of max-flow search through augmenting path. (a) initial residual graph with the first found $s-t$ path shown in dotted lines; (b) augmented residual graph with newly found $s-t$ path; (c) current augmented residual graph, where augmenting path terminates since no more $s-t$ path can be found; (d) the obtained actual max-flow.

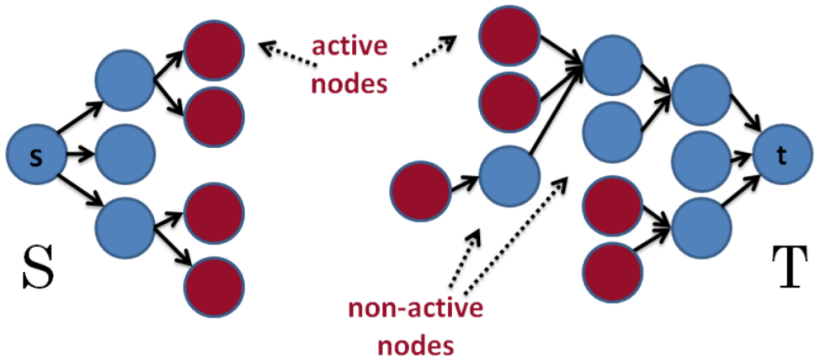


Figure 2.3: An example of trees $\mathcal{S}$ and $\mathcal{T}$ in BK algorithm. Non-active and active nodes are marked as blue and red respectively. [2]

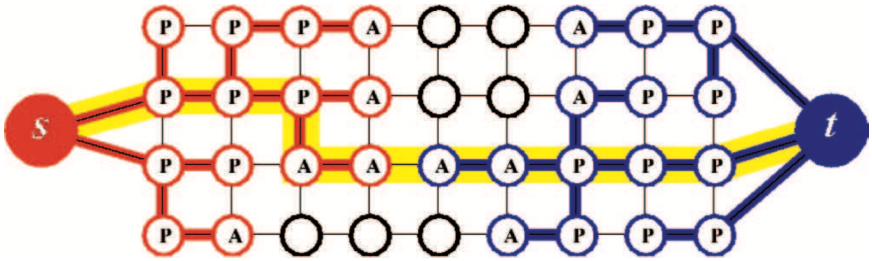


Figure 2.4: An example of the search trees $\mathcal{S}$ (red nodes) and $\mathcal{T}$ (blue nodes) at the end of the growth stage when a path (yellow line) from the source s to the sink t is found. Active and passive nodes are labeled by letters A and P, correspondingly. Free nodes appear in black. [1]

the nodes in $\mathcal{S}$ and $\mathcal{T}$ are either active (the outer boundary of each tree) or passive (the internal points of each tree). Figure 2.3 gives an example of the trees.

In the initial state, $\mathcal{S}$ has only one node s and $\mathcal{T}$ only contains t. Then the BK algorithm iteratively conducts operations in three stages: growth, augmentation, and adoption.

At the growth stage, the two search trees are expanded to find the $s-t$ path. Each active node explores adjacent edge with positive residual capacity ($R(u,v) > 0$ for $u \in \mathcal{S}$, and $R(v,u) > 0$ for $u \in \mathcal{T}$), and adds newly discovered free nodes into the tree as its children. The newly added nodes are set as active. When all neighbors of an active node have been scanned, the active node is set to be

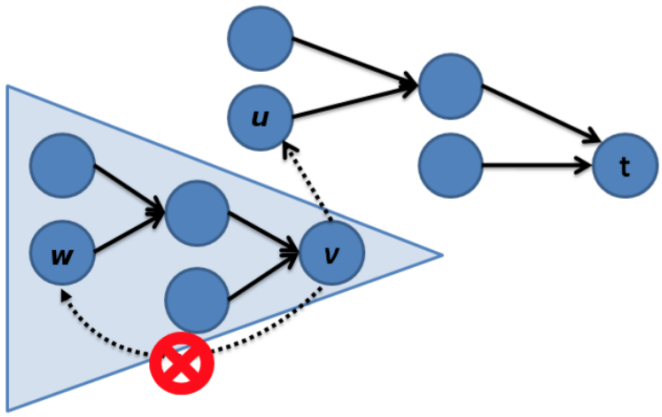


Figure 2.5: An example of adoption step on a node v. An orphan sub-tree is shown in the triangle. The solid edges are residual edges which are tree edges and the dashed edges are residual edges which are not tree edges. Node u can be selected as a parent of v. But w cannot be selected as a parent for v, because there is no residual path from w to t (the path terminates at v). [2]

passive. If no active node remains, the whole BK algorithm terminates. If a edge from $\mathcal{S}$ to $\mathcal{T}$ is found, which means there is a $s-t$ path, the augmentation stage starts. Figure 2.4 gives an example of when $\mathcal{S}$ meets $\mathcal{T}$ at the end of the growth stage.

At the augmentation stage, the $s-t$ path found in the growth stage is augmented by sending maximum possible flow through it. After the augmentation, some edges on the augmenting path become saturated (i.e., residual flow becomes zero). Thus some nodes in $\mathcal{S}$ and $\mathcal{T}$ become orphans, as the edges linking them to their parents are saturated. If edge (u, v) becomes saturated and both u and v are originally in tree $\mathcal{S}$, then v becomes a $\mathcal{S}$-orphan. If both u and v are originally in tree $\mathcal{T}$, then u becomes a $\mathcal{T}$-orphan. If u is in $\mathcal{S}$ and v is in $\mathcal{T}$, then no orphan is created in (u, v) saturation. In the other words, the augmentation operation may split trees $\mathcal{S}$ and $\mathcal{T}$ into forests, where s and t are still the root of $\mathcal{S}$ and $\mathcal{T}$ respectively, and the orphans form the roots of all other trees. Orphans created in the augmentation stage are placed in a list and handled in the adoption stage.

In the adoption stage, orphans are processed until there are no orphans left.

The BK algorithm tries to find a new valid parent in $\mathcal{S}$ for each $\mathcal{S}$-orphan, and similarly a parent in $\mathcal{T}$ for each $\mathcal{T}$-orphan. For instance, if we have a $\mathcal{S}$-orphan v, we seek such u that has $R(u, v) > 0$, $u \in \mathcal{S}$ and the tree path from s to u is valid (the whole path has positive residual capacity). If such a u is found, we make u to be the parent of v. If we can not find a new parent for v, we mark v as a free node and mark all former children of v as orphans. Then we examine all edges (u, v) have positive residual capacity, and make each $u \in \mathcal{S}$ active.

When the adoption stage is complete, the algorithm returns to the growth stage. The algorithm terminates when $\mathcal{S}$ and $\mathcal{T}$ can not grow (i.e., there are no active nodes) and the trees are separated by saturated edges (i.e., with zero residual capacity).

2.2.3 Incremental Breadth First Search (IBFS) Algorithm

The IBFS algorithm [59] [2] is a modification of the BK algorithm, where the $\mathcal{S}$ and $\mathcal{T}$ trees are maintained to be always breadth-first trees. As a result, any $s-t$ path found in the procedure is a shortest path, and the overall running time has a polynomial bound $O(n^2m)$.

The IBFS algorithm also maintains two trees $\mathcal{S}$ and $\mathcal{T}$, which are rooted at s and t respectively. At any given moment, a node can be in one of five states: $\mathcal{S}$-node, $\mathcal{T}$-node, $\mathcal{S}$-orphan, $\mathcal{T}$-orphan, or $\mathcal{N}$-node (which indicates the node is not in any tree). There is a parent pointer for each node indicating the parent of this node in the tree, which is empty for $\mathcal{N}$-nodes and orphans. A node u is called in $\mathcal{S}$ if u is a $\mathcal{S}$-node or a $\mathcal{S}$-orphan.

Distance labels $d_s(u)$ or $d_t(u)$ are maintained for each node u, representing the distance from s or t to u in the tree. If u is in $\mathcal{S}$, then only $d_s(u)$ is meaningful and $d_t(u)$ is unused. The situation for nodes in $\mathcal{T}$ are symmetric. For some values D_s and D_t, the nodes in tree $\mathcal{S}$ have no more than D_s from s and the nodes in tree $\mathcal{T}$ have up to D_t distance to t. Thus $L = D_s + D_t + 1$ is the lower bound of the length of any augmenting path.

Similar to the BK algorithm, IBFS also has three steps: growth, augmentation, and adoption. At the initial state, $\mathcal{S}$ has only s, $\mathcal{T}$ has only t, $d_s(s) = d_t(t) = 0$, and parent pointers for all nodes are empty.

The IBFS algorithm proceeds in passes. At the beginning of a pass, there are

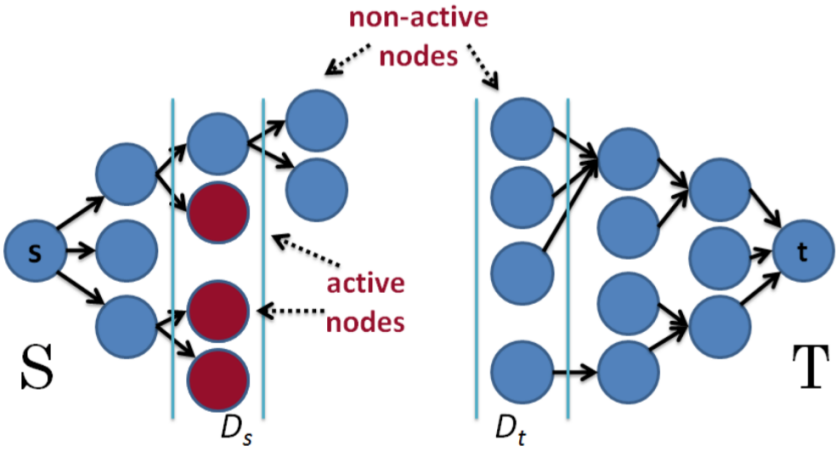


Figure 2.6: An example of a forward pass of IBFS. Note that during a forward $\mathcal{S}$ grown pass, active nodes exist only in level D_s of $\mathcal{S}$ [2]

no orphans. All nodes in $\mathcal{S}$ are $\mathcal{S}$-nodes, all nodes in $\mathcal{T}$ are $\mathcal{T}$-nodes, and the rest nodes are $\mathcal{N}$-nodes. In a pass of the growth stage, IBFS chooses a tree to grow (forward for $\mathcal{S}$ and backward for $\mathcal{T}$) for one level, this increases D_s (or D_t) and L by one. Figure 2.6 gives an example.

Taking a forward growth pass as an example, in which tree $\mathcal{S}$ is grown, the operation for a backward pass is symmetrical. Firstly, all nodes u in $\mathcal{S}$ with $d_s(u) = D_s$ are set to be active. The pass then executes growth steps. This may be interrupted by augmentation steps (when an augmenting path is found) followed by adoption steps (to fix the invariants violated when some arcs get saturated). At the end of the pass, if $\mathcal{S}$ has any nodes at level $D_s + 1$, D_s is incremented by 1; otherwise the algorithm terminates.

The growth step picks an active node v and scans all residual edges (v, w) leaving v. If w is a $\mathcal{S}$-node, nothing is done about it. If w is a $\mathcal{N}$-node, we mark w as a $\mathcal{S}$-node, set $p(w) = v$, and set $ds(w) = D_s + 1$. If w is in $\mathcal{T}$, an augmentation step is performed. Once all residual edges leaving v are scanned, v becomes inactive. If the scan on v is interrupted by augmentation, we record the outgoing residual edge that triggered the augmentation. If v is still active after the augmentation, the scan of v is resumed from that edge.

The augmentation step is performed when a residual edge (v, w) is found such

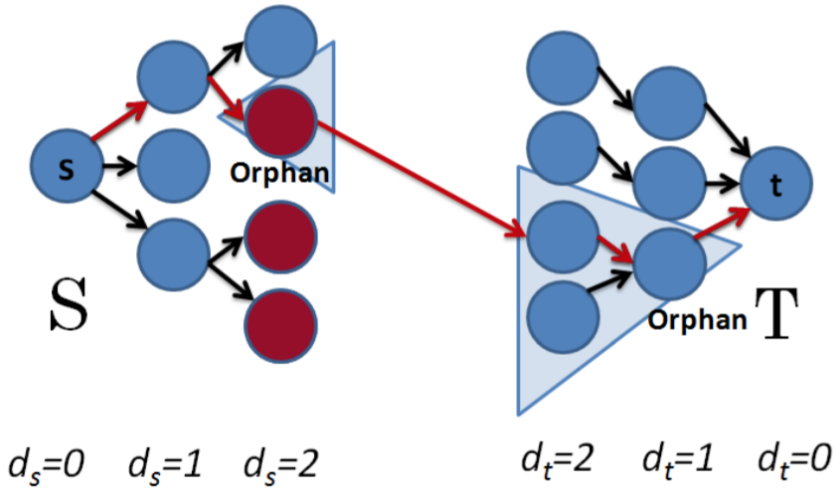


Figure 2.7: An example of augmentation step of IBFS [2]. The parent edges of the orphan nodes are saturated during this augmentation. The triangles represent the orphan sub trees that are created after the augmentation. Note the augmenting path is always a shortest $s-t$ path in the residual graph.

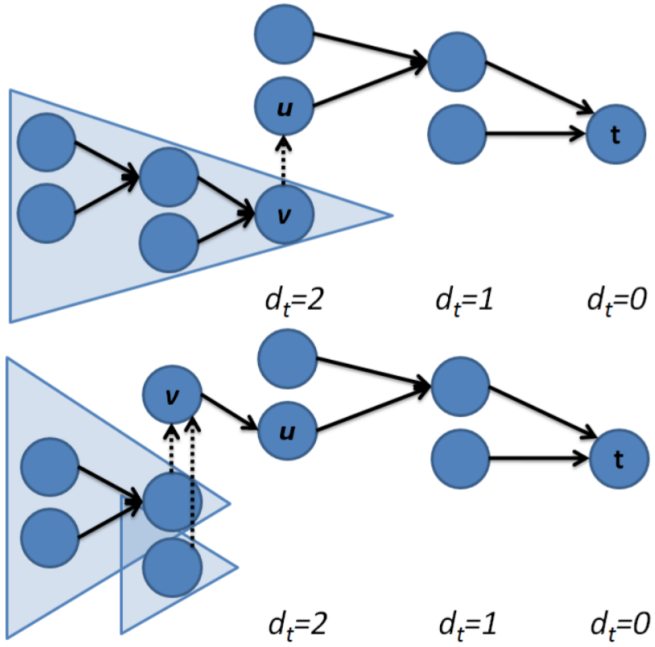


Figure 2.8: An example of adoption step of IBFS [2], in which orphan v is relabeled. The solid edges are residual edges which are tree edges. The dashed edges are residual edges which are not tree edges. The triangles represent orphan sub trees. Node v finds u as the lowest potential parent and performs a relabel with u as his new parent. The children of v then become orphans themselves and will be processed later during this adoption phase.

that v is in $\mathcal{S}$ and w is in $\mathcal{S}$, as shown in Figure 2.7. In such circumstances, the path P obtained by connecting the $s^\frown v$ path in $\mathcal{S}$, the edge (v, w), and the $w^\frown t$ path in $\mathcal{S}$ is an augmenting path. We conduct augmentation on P, saturating some of its edges. Saturating any arc (x, y) other than (v, w) creates orphans. Note that x and y are in the same tree. If they are both in $\mathcal{S}$, y is marked as an $\mathcal{S}$-orphan, otherwise x is marked as a $\mathcal{T}$-orphan. At the end of the augmentation step, there are two sets (possibly empty) of $\mathcal{S}$-orphans and $\mathcal{T}$-orphans. These sets are handled during the adoption step.

The adoption step recovers the $\mathcal{S}$ and $\mathcal{T}$ trees by eliminating orphans, Figure

2.8 gives an example. Here we assume $\mathcal{S}$ is grown thus we have $\mathcal{S}$-orphans to be processed. The adoption procedures for eliminating $\mathcal{T}$-orphans are symmetric. To process an $\mathcal{S}$-orphan v, the edges list is scanned starting from the current edge and the scan stops when a residual edge (u, v) with $d_s(u) = d_s(v) - 1$ is found. If such a u is found, v is marked as an $\mathcal{S}$-node, the current edge of v is set to be (v, u), and u is set to be the parent of v. If such a u can not be found, the orphan relabelling operation is applied to v. This relabel operation scans the whole edge list to find the u such that $d_s(u)$ is minimum and (u, v) has positive residual capacity. If no such u exists, or if $d_s(u) > D_s$, we make v as a $\mathcal{S}$-node and mark nodes w such that $p(w) = v$ as $\mathcal{S}$-orphans. Otherwise we choose u to be the first such node and set the current edge of v to be (v, u), set $p(v) = u$, set $d_s(v) = d_s(u) + 1$, make v an $\mathcal{S}$-node, and mark nodes w such that $p(w) = v$ as $\mathcal{S}$-orphans. If v was active and now has $d_s(v) = D_s + 1$, we make v inactive.

2.2.4 Push Relabel Algorithm

Push relabel algorithm is also known as preflow push algorithm, which is developed by Goldberg and Tarjan [60]. Different from algorithms in augmenting path family, push relabel algorithm do not keep the mass balance constraint hold at all time. The graph is flooded with excess in push relabel algorithm, these excess are either pushed towards sink t to form actual $s - t$ flow, or pushed back to source s for those can not reach t to satisfy mass balance constraint at the end of algorithm execution.

Excess $e(v)$ at a node v is defined as the amount of flow that incoming exceeds outgoing

$$e(v) = \sum_{(u,v)\in E} f(u, v) - \sum_{(v,w)\in E} f(v, w). \tag{2.3}$$

An node with positive excess is termed as active node. In order to estimate the distance of a node from s or t, distance label d is maintained for each node. For source s and sink t, we have $d(s) = n$ and $d(t) = 0$. For any edge (u, v) in the residual graph, have $d(u) \leq d(v) + 1$.

Push relabel algorithm consists two basic operations: push and relabel. The push operation pushes largest possible amount of flow through an admissible

edge, the procedure of of push operation is given in Algorithm 1. Note that line 1 of Algorithm 1 is the condition test to see if edge (u, v) is admissible for push operation.

Algorithm 1 Push Operation in Push Relabel Algorithm

Input: Graph $G = (V, E, C)$, residual R, distance labeling d, excess e, and the edge to push (u, v).

Output: Graph $G = (V, E, C)$, residual R, distance labeling d and excess e.

1: **if** $e(u) > 0$, $R(u, v) > 0$ and $d(u) = d(v + 1)$. **then**
2: Send $\delta = min(e(u), R(u, v))$ amount flow through edge (u, v) as:
$R(u, v) \leftarrow R(u, v) - \delta$; $R(v, u) \leftarrow R(v, u) + \delta$;
$e(u) \leftarrow e(u) - \delta$; $e(v) \leftarrow e(v) + \delta$.
3: **end if**

Relabel is another basic operation in push relabel algorithm. When there is no admissible edge to push, relabel operation adjusts distance labels to make push operation possible again. The procedure in relabel operation is stated in Algorithm 2. Note that line 1 in Algorithm 2 is the applicability test for relabeling u.

Algorithm 2 Relabel Operation in Push Relabel Algorithm

Input: Graph $G = (V, E, C)$, residual R, distance labeling d, excess e, and the node to relabel u.

Output: Graph $G = (V, E, C)$, residual R, distance labeling d and excess e.

1: **if** $e(u) > 0$ and $\forall R(u, v) > 0$ have $d(u) \leq d(v)$. **then**
2: Relabel $d(u) = min(d(v) : R(u, v) > 0) + 1$.
3: **end if**

At the initial state of push relabel algorithm, the residual graph is identical to input graph, the distance labeling is n for source s and 0 for all other nodes, the source node has infinite excess. The algorithm starts with a set of initial saturating push, in which for each edge (s, u) leaving source s is pushed with a flow that equals its capacity $C(s, u)$. Then the algorithm repeatedly perform push and relabel operations to push excess and modify distance labeling. For a push on (u, v), $\delta = min(e(u), R(u, v))$ flow is pushed from u to v, which increases

$R(v,u)$ and $e(v)$ by δ and decreases $R(u,v)$ and $e(u)$ by the same amount. For a relabel on u, the distance label $d(u)$ is set to be the largest value allowed by the valid labeling constraint. The algorithm terminates when there is no active node left. At this point, none nodes except s and t has excess, thus the mass balance constraint is satisfied and the preflow becomes flow which is the max-flow obtained.

An example of push relabel execution is given in Figure 2.9 and 2.10. Figure 2.9a shows the input residual graph and distance labeling, the residual graph is identical to input graph at this state. In Figure 2.9b, the initial saturating push is conducted over all edges leaving source s, 20 flow are pushed through (s,u) and (s,v) respectively. In Figure 2.9c, active node u is relabeled, so that it can push its 20 excess towards sink t and node v. The excess on u is pushed through edge (u,t) and (u,v) in Figure 2.9d and 2.9e, respectively. In Figure 2.9f, active node v is relabeled, so that it can push excess towards t. In Figure 2.9g, 20 excess on v is pushed to t via (v,t). In Figure 2.9h, active node v is relabeled as $d(v) = 2$, so that edge (v,u) becomes admissible.

In Figure 2.10a, the remaining 10 excess on v is pushed to u. At this state, there are two edges (u,s) and (u,v) leaving active node u with positive capacity. Since $d(s) = 4$ and $d(v) = 2$, u is relabeled as $d(u) = 3$ in Figure 2.10b. Then the excess on u is pushed to v via admissible edge (u,v) in Figure 2.10c. Next, in Figure 2.10d node v is relabeled as $d(v) = 4$ and the excess on v is pushed to u.

The steps of push relabel algorithm is summarized in Algorithm 3.

2.3 Incremental Max-flow Algorithms

Based on above batch algorithms, some incremental max-flow algorithms are propose for learning from dynamic graphs. In this section, we review the most important incremental max-flow algorithms.

2.3.1 Incremental Push Relabel

Kumar and Gupta propose an incremental max-flow algorithm in [61] based on push relabel mechanism. In their incremental push relabel algorithm, graph change is considered as inserting and deleting of edge which equivalent to edge

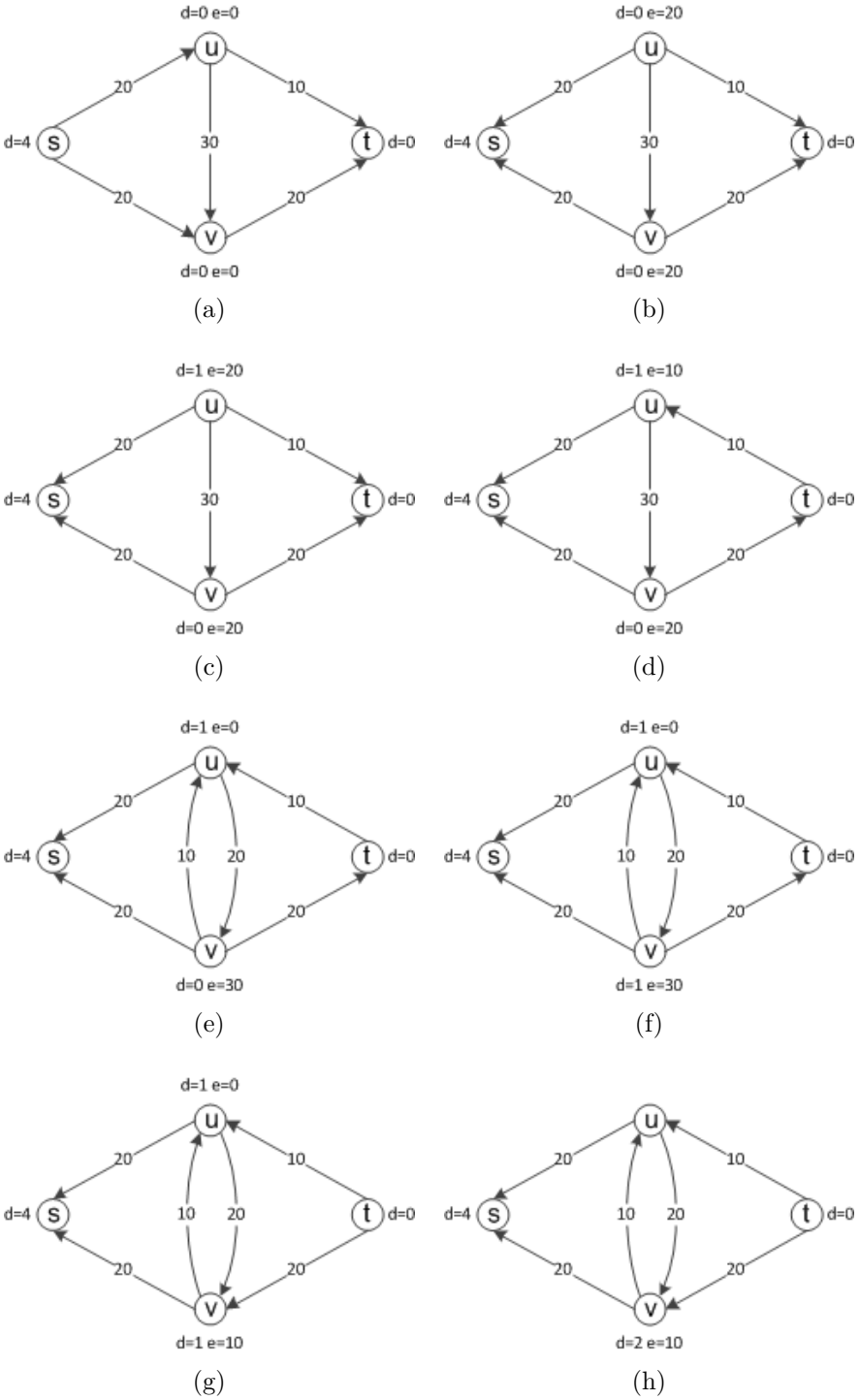


Figure 2.9: An example of max-flow search through push relabel. Part A

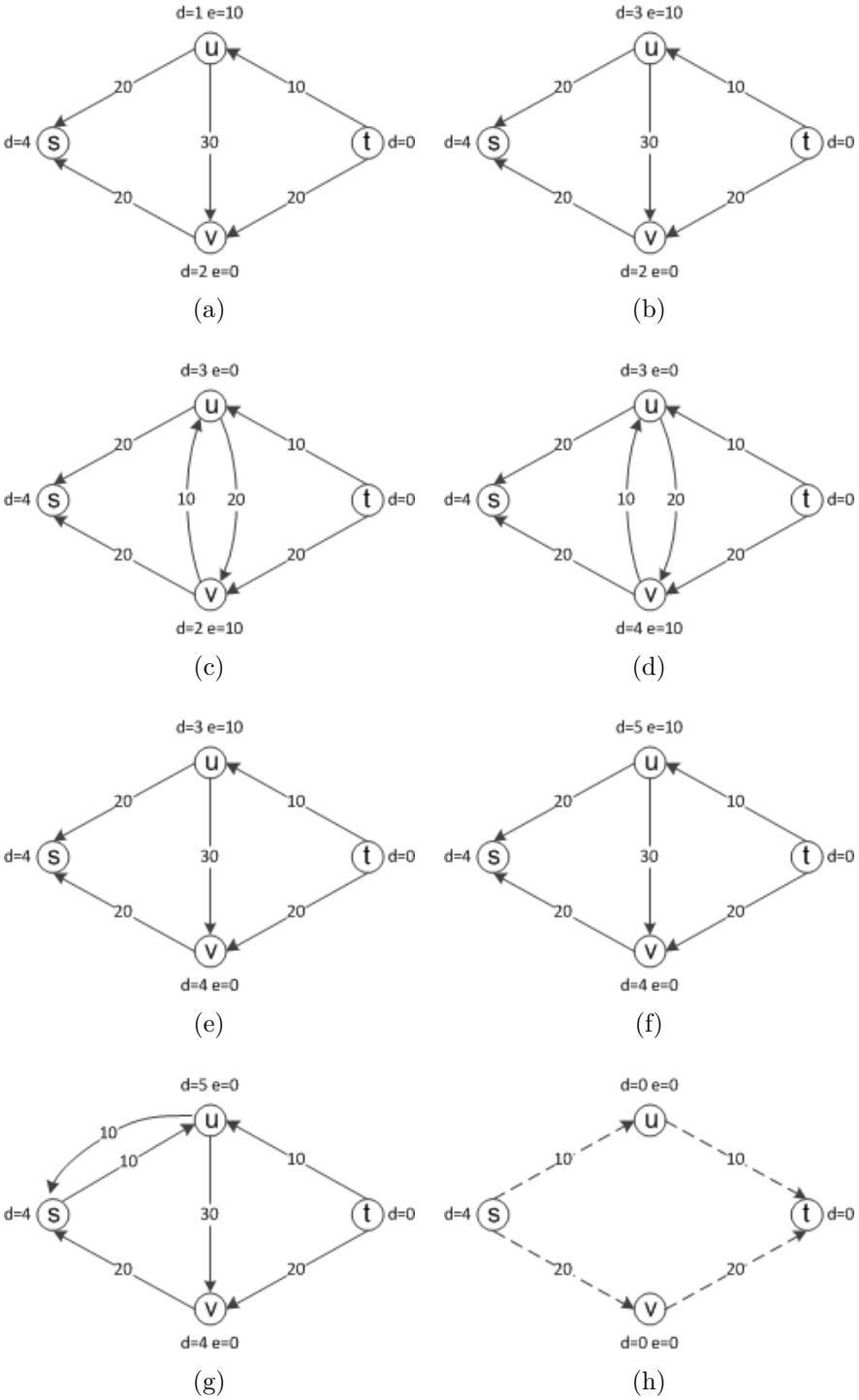


Figure 2.10: An example of max-flow search through push relabel. Part B

Algorithm 3 Push Relabel Algorithm

Input: Graph $G = (V, E, C)$.

Output: Graph $G = (V, E, C)$, residual R, distance labeling d and excess e.

1: Initialize residual graph as $R(u, v) = C(u, v)$, $\forall (u, v) \in E$;
2: Initialize distance label as $d(s) = n$ and $d(u) = 0$, $\forall u \in V \setminus \{s\}$;
3: Initialize excess as $e(s) = \infty$ and $e(u) = 0$, $\forall u \in V \setminus \{s\}$;
4: **for** Each (s, u) that $C(s, u) > 0$ **do**
5: Push $C(s, u)$ through (s, u) as:
$e(u) = C(s, u)$, $R(s, u) = R(s, u) - C(s, u)$ and
$R(u, s) = C(u, s) + C(s, u)$;
6: **end for**
7: **while** There exists a basic operation that applies **do**
8: Select a basic operation and perform it;
9: **end while**

capacity increase and decrease. These changes are addressed on one edge after another.

For any edge inserting or deleting, the incremental push relabel algorithm first finds the set of nodes that are affected by such change. In case of inserting a new edge u, v, the possible newly formed flow will go through augmenting paths that from source s to u and then v to sink t. The fist set of affected nodes which lie on the path of s to u are found by Backward Breadth First Search (BBFS) from u to s, and the second set of affected nodes which lie on the path of v to t are found by Forward Breadth First Search (FBFS). On the other hand, when a edge u, v is deleted, some existing augmenting path goes through $s - u - v - t$ may be affected. Thus the affected nodes are found by BBFS from u to s and FBFS from t to v. The algorithm of BBFS and FBFS are given in Algorithm 4 and 5 respectively.

Based on the push relabel algorithm discussed in Section 2.2.4, Kumar and Gupta give two lemmas that are important for helping identify affected nodes:

Lemma 1. *For any node u in the result of push relabel algorithm, if have $d(u) < n$, then no extra flow can be sent from source s to the node u.*

Algorithm 4 Backward Breadth First Search Algorithm

Input: Graph $G = (V, E, C)$, start node u, end node v.

Output: Affected node set AFF.

```
1: Initialize WORKSET = {u};
2: Initialize AFF = {u};
3: while WORKSET ≠ ∅ do
4:    Remove an element x from WORKSET;
5:    if x = v then
6:       break
7:    else
8:       for all edges (y, x) ∈ E do
9:          if C(y, x) > 0 then
10:            AFF = AFF ∪ {y};
11:            WORKSET = WORKSET ∪ {y};
12:         end if
13:      end for
14:   end if
15: end while
```

Lemma 2. *For any node u in the result of push relabel algorithm, if have $d(u) \geq n$, then no extra flow can be sent from the node u to sink t.*

The proof of Lemma 1 and 2 can be found in study [61].

Edge Insertion

For a edge (u, v) newly inserted, Kumar and Gupta's algorithm firstly find affected nodes and then use basic push relabel operation within the affected nodes (which is normally a subset of the whole graph) to find the newly formed $s - t$ flow. In finding the affected nodes, the following interpretation is given.

1. Each newly formed flow must starts from s and passes u in its way to t. Flows that do not lead to u and t are useless as they will return to the source s.

Algorithm 5 Forward Breadth First Search Algorithm

Input: Graph $G = (V, E, C)$, start node u, end node v.

Output: Affected node set AFF.

```
1: Initialize WORKSET = {u};
2: Initialize AFF = {u};
3: while WORKSET ≠ ∅ do
4:    Remove an element x from WORKSET;
5:    if x = v then
6:       break
7:    else
8:       for all edges (x, y) ∈ E do
9:          if C(x, y) > 0 then
10:            AFF = AFF ∪ {y};
11:            WORKSET = WORKSET ∪ {y};
12:         end if
13:      end for
14:   end if
15: end while
```

2. According to Lemma 1, on the way from s to u, only the nodes x that has $d(x) \geq n$ are the candidates of being affected nodes, because for those nodes y that have $d(y) < n$ can not receive extra flow from s.

3. According to Lemma 2, on the way from v to t, only the nodes x that has $d(x) < n$ are the candidates of being affected nodes. This is because for those nodes y that have $d(y) \geq n$ can not send extra flow to t.

4. Only edges (x, y) that have positive residual capacity $R(x, y) > 0$ can route extra flow after (u, v) insertion.

Based on above four points, the identification of affected nodes in case of edge (u, v) insertion consists two steps:

1. Add two extra conditions that $d(x) \geq n$ and $R(x, y) > 0$ at line 9 of Algorithm 4. Call above modified BBFS Algorithm with arguments that from u to s.

2. Add two extra conditions that $d(x) < n$ and $R(x, y) > 0$ at line 9 of Algorithm 5. Call above modified FBFS Algorithm with arguments that from v to t.

Having all affected nodes identified, the incremental push relabel algorithm then initialize the preflow f as:

1. The residual capacity for each edge leaving source node s to an affected nodes.

2. Zero for any other edges (between affected nodes and sink t).

This preflow initialization is followed by a modified push relabel operation, in which push and relabel process only applies on those affected nodes.

The incremental push relabel algorithm terminates when there is no more active nodes in the graph. At this point, if there is any new flow formed due to the insertion of (u, v), the new flow is found and added into the current flow. The steps for incremental push relabel algorithm to handle edge insertion are stated in Algorithm 6.

Algorithm 6 Incremental Push Relabel - Edge Insertion

Input: Graph $G = (V, E, C)$, residual graph R, edge (u, v) to be inserted.
Output: Residual graph R.

1: Call Algorithm 4 to conduct BBFS from u to s;
2: Call Algorithm 5 to conduct FBFS from v to t;
3: Initialize distance label as $d(s) = n$ and $d(u) = 0$, $\forall u \in V \setminus \{s\}$;
4: Initialize excess as $e(s) = \infty$ and $e(u) = 0$, $\forall u \in V \setminus \{s\}$;
5: **for** Each (s, u) that $R(s, u) > 0$ **do**
6: Push $R(s, u)$ through (s, u) as:
$e(u) = C(s, u)$, $R(s, u) = R(s, u) - C(s, u)$ and
$R(u, s) = C(u, s) + C(s, u)$;
7: **end for**
8: **while** There exists a basic operation that applies **do**
9: Select a basic operation and perform it;
10: **end while**

Edge Deletion

For a edge (v, w) removed from the original graph, there is a possibility of changing the max-flow only when there is flow goes through (v, w). In this case, when (v, w) is deleted, the flow $f(v, w)$ on this edge becomes zero and this amount of flow must be pushed back from t to w and then to v. Then the excess on v is pushed as much as possible towards t using alternate augmenting paths in the modified graph. If there is some excess on v can not be pushed to t, it will be pushed back to s.

In incremental push relabel algorithm, nodes v and w are set as active. The algorithm try to push their excess toward s and t, in which there are three operations to be performed:

1. Pushing flow from t to w;
2. Pushing flow from w to v;
3. Pushing flow from v and w towards t and s.

For the first operation which is a preflow push problem is the reversed direction, a reverse graph G^R is used. G^R is obtained by reversing all the residual edges in graph G. Since not all nodes in G^R are associated with the flow from t to w, the FBFS algorithm is called with arguments w and t to find the affected nodes. Then a standard push relabel algorithm is applied on these nodes to push flow from t to w.

Once the first operation is finished, $f(v, w)$ amount of excess is pushed from w to v. After this pushing, edge (v, w) can be safely removed since there is no flow goes through it.

When first two operation are accomplished, node v has $f(v, w)$ amount of excess which is pushed from w in step 2 and node w may have some excess as well. This is because over $f(v, w)$ amount of flow is pushed from t to w in step 1, as push relabel algorithm, different from augmenting path, can not control the total amount of flow pushed.

For the third operation, the algorithm conducts two separated push relabel procedure with active nodes v and w, respectively. The complete steps for incremental push relabel algorithm to handle edge deletion are stated in Algorithm 7.

Algorithm 7 Incremental Push Relabel - Edge Deletion

Input: Graph $G = (V, E, C)$, residual graph R, edge (v, w) to be deleted.

Output: Residual graph R.

1: Reverse residual graph R, obtain G^R
2: Call Algorithm 5 to conduct FBFS from t to w;
3: Call Algorithm 3 to push from t to w in G^R, with constraint that all operations are applied in affected nodes;
4: Push $f(v, w)$ excess from w to v in R
5: Delete (v, w) from G and R;
6: Call Algorithm 5 to conduct FBFS from v to t;
7: Call Algorithm 3 to push from v to t in G^R, with constraint that all operations are applied in affected nodes;
8: Call Algorithm 5 to conduct FBFS from w to t;
9: Call Algorithm 3 to push from w to t in G^R, with constraint that all operations are applied in affected nodes;

2.3.2 Excesses IBFS Algorithm

Excesses IBFS (EIBFS) algorithm [58] is developed based on IBFS algorithm and is capable of learning max-flow from the dynamic graphs.

On Static Graphs

Different from IBFS and BK, in which a feasible flow is always maintained, EIBFS is a generalized IBFS that maintains only a pseudoflow. Pseudoflow is a flow that follows capacity constraint but not conservation constraints.

A node v is known as an excess if $e(v) > 0$ and a deficit if $e(v) < 0$. In EIBFS, source s and sink t are defined to has infinite excess and deficit, respectively. EIBFS maintains two of node-disjoint forests $\mathcal{S}$ and $\mathcal{T}$. Each excess is a root of a tree in $\mathcal{S}$, and a root of tree in $\mathcal{S}$ must be an excess. Similarly, each deficit is a root of a tree in $\mathcal{T}$, and a root in $\mathcal{T}$ must be a deficit. For a non-root node v in $\mathcal{S}$ or $\mathcal{T}$, $p(v)$ is the parent of v in its respective forest. A node which is not in $\mathcal{S}$ nor in $\mathcal{T}$ a is called free node.

EIBFS also maintains distance labels $d_s(v)$ and $d_t(v)$ for every node v, just as

in IBFS. The edges in forests $\mathcal{S}$ and $\mathcal{T}$ are admissible with respect to d_s and d_t, respectively.

Initially, every root r in $\mathcal{S}$ or in $\mathcal{T}$ has $d_s(r) = 0$ or $d_t(r) = 0$, respectively. But new excesses and deficits formed in the algorithm execution may have arbitrary distance labels. Thus the roots of a tree do not necessarily have zero distance label. Similar to that in IBFS, D_s and D_t are maintained as

$$\begin{aligned} D_s &= \max_{v \in \mathcal{S}}(d_s(v)) \\ D_t &= \max_{v \in \mathcal{T}}(d_s(v)). \end{aligned} \tag{2.4}$$

At the initial state of EIBFS, $\mathcal{S}$ has only s, $\mathcal{S}$ has only t, $d_s(s) = d_t(t) = 0$, $D_s = D_t = 0$ and $p(v)$ is empty for every node v. The EIBFS algorithm is executed in phases. Each phase is either a forward phase (when the $\mathcal{S}$ forest is grown) or a backward phase (when the $\mathcal{T}$ forest is grown). Every phase first performs the growth steps, which may be interrupted by augmentation steps (when an augmenting path is found) and followed by alternating adoption and augmentation steps.

Let us take forward phase as an example to describe the procedures of EIBFS, the procedures in backward phase are symmetric. For a forward phase, the goal is to grow forest $\mathcal{S}$ by one level. If $\mathcal{S}$ has nodes at level $D_s + 1$ at the end of the phase, D_s is incremented by one; otherwise the algorithm terminates.

In the forward phase, growth steps are firstly performed. All nodes v in $\mathcal{S}$ that has $d_s(v) = D_s$ are marked as active. Then we pick an active node v and scan v by examining all residual edges (v, w) leaving v. If $w \in \mathcal{S}$, we do nothing. If w is a free node, we add w to $\mathcal{S}$, set $p(w) = v$, and set $d_s(w) = D_s + 1$. If $w \in \mathcal{T}$, which means an augmentation path from s to t is found, augmentation step is performed as described later. Edge (v, w) is recorded as the outgoing edge that triggered the augmentation step. If v is still active after the augmentation step, the scan of v is resumed from (v, w) to avoid re-scanning the edges processed. If (v, w) is still residual and connects the forests, we do more augmentation steps using it. After all edge out of v have been scanned, v becomes inactive. When all nodes are inactive, the phase ends.

The augmentation steps in EIBFS are different from those in IBFS. When a connecting edge (v, w) that has $v \in \mathcal{S}$ and $w \in \mathcal{S}$ is found, we increase the flow on (v, w) by any feasible amount without violating the capacity constraint of (v, w).

As a result of the flow added, an excess may be created in $\mathcal{T}$ and a deficit may be created in $\mathcal{S}$. We now alternate between augmentation steps and adoption steps as we describe below. Once all excesses have been drained or removed from $\mathcal{T}$ and all deficits have been drained or removed from $\mathcal{S}$ we continue to perform growth steps.

We now introduce how to handle excesses created in $\mathcal{T}$. the deficits in $\mathcal{S}$ are addressed symmetrically. A node v in $\mathcal{T}$ is called orphan if its parent arc $(v, p(v))$ is not admissible (possibly saturated) and have $e(v) \geq 0$. An augmentation step is executed by picking an excess v in $\mathcal{T}$ and pushing flow out of v as described below. This push may create orphans and more excesses in $\mathcal{T}$. If orphans are created, adoption steps are performed to repair them. After orphans are repaired we execute another augmentation step from another excess. Augmentation and adoption steps stop when all excesses are drained or removed from $\mathcal{T}$.

Flow is pushed out of an excess v in $\mathcal{T}$ as follows. The tree path from v to the root r in $\mathcal{T}$ are traversed. For every edge (x, y) in this path, we increase its flow by $minR(x, y), e(x)$. This means that we either drain the entire excess from x or saturate the edge (x, y), making x an orphan. Root r remains a deficit if not enough excess is drained into it. Otherwise it has $e(r) \geq 0$ and becomes an orphan, thus it can no longer serve as a root in $\mathcal{T}$.

In adoption step, an orphan v in $\mathcal{T}$ is repaired by either setting a new parent $p(v)$ in T or by removing v from $\mathcal{T}$. The adjacency list of v is scanned starting from the current arc and stop when an admissible outgoing edge is found or the end of the list is reached. If an admissible edge (v, u) is found, we set the current arc of v to be (v, u) and set $p(v) = u$. If no such edge can be found, we apply the orphan relabel operation to v.

The orphan relabel operation scans the adjacency list of v to find a new parent u for v. A node u is qualified to be a new parent of v if: 1) u is a node with minimum $d_t(u)$ such that (v, u) has positive residual capacity; and 2) $d_t(u) < D_t$ for a forward phase, or $dt(u) \leq Dt$ for a backward phase. If such u is found (may be more than one), we set the first such node to be u, set the current edge of v to be (v, u), set $p(v) = u$ and set $d_t(v) = d_t(u) + 1$. As a result, every node w with $p(w) = v$ becomes an orphan. These new orphans need to be repaired by adoption steps. If no such u is found, we make v a free vertex if $e(v) = 0$ or add

v to $\mathcal{S}$ as a new root if $e(v) > 0$.

On Dynamic Graphs

EIBFS algorithm considers graph change as violations to flow feasibility or to the invariants of the algorithm. The violations are summaries as following types:

(a) An edge (v, w) such that $f(v, w) > C(v, w)$, due to edge capacity reduce;

(b) A new residual edge (v, w) such that $v \in \mathcal{S}$ and $w \in \mathcal{T}$;

(c) A new residual edge (v, w) such that v and w are in $\mathcal{S}$ having $d_s(w) > d_s(v) + 1$, or the symmetric case in tree $\mathcal{T}$;

(d) A new residual edge (v, w) such that v and w are in $\mathcal{S}$, $d_s(w) = d_s(v) + 1$, and (v, w) precedes the current arc of v, or the symmetric case in tree $\mathcal{T}$;

(e) A new residual edge (v, w) such that $v \in \mathcal{S}$, $d_s(v) \leq D_s$ and w not in $\mathcal{S}$,or the symmetric case in tree $\mathcal{T}$.

These violations are fixed by some base operations introduced in the static graph setting. Specifically, (a) is resolved by pushing flow along (w, v); (b), (c) and (e) are fixed by by saturating edge (v, w). In these cases, new excesses or deficits are generated. These excesses and deficits are solved by alternating augmentation and adoption steps. The violation (d) is handled by simply reassigning the current arc of v.

2.4 Summary

Based on above observations, we find that handling edge capacity reduce or edge remove (i.e. the decremental learning) is the most difficult part for incremental max-flow. As this operation need to redirect current flow in alternative path or cancel current flow if it can not be redirected. The performance of an incremental max-flow is determined, to a large extent, by its decremental operation.

We also found that both existing incremental max-flow algorithms apply push-relabel style operation in handling edge capacity reduce. This method involves

great amount of operations in neighbor search, flow push and node relabel, thus has higher empirical computational complexity.

Alternatively, we decided to derive an incremental max-flow algorithm based on augmenting path. In the next chapter, we introduce the proposed augmenting path based incremental max-flow algorithm.

3 Proposed Incremental Max-flow Algorithm

3.1 Introduction

In this chapter, we present proposed incremental max-flow algorithm which is constructed base on augmenting path algorithm. The augmenting path mechanism is firstly introduced. Then we derived the max-flow updating procedures in response to all possible graph changes.

For the convenience of algorithm derivation and clarity of presentation, we summarize most notations used in this chapter in Table 3.1.

3.1.1 Motivation

We address max-flow problem because it is a fundamental algorithm in the graph theory and can be used to solve various problems such as min-cut, multi-source multi-sink max-flow, maximum edge-disjoint path etc [12]. Up to now, max-flow has been adopted in vast real-world applications, such as bottleneck identification for city traffic network [15] and bottleneck identification for power system security index computation [16]. Also in wireless mobile environment, max-flow is being applied to optimize the association between wireless clients and access points by maximizing the traffic flow to clients [17] [18]. In addition, it has been widely used in computer vision for image segmentation [20] [21] [24], stereo [27] [28] and shape reconstruction [30].

In big data era, data is becoming available quickly in a sequential manner, which requires system to process data in real time. For max-flow learning, if a huge graph changes frequently over time, then it is obviously not efficient to always retrain max-flow from scratch. Consider incremental max-flow, existing

Notation	Descriptions
G	weighted graph, $G = (V, E, C)$
V	node set
E	edge set
C	capacity set
u, v	node u, node v
s, t	source node, sink node
e	edge e
(u, v)	edge from node u to node v
$C(e)$, $C(u, v)$	capacity on edge e and (u, v)
R	residual graph, $R = (V, E, R)$
$R(e)$, $R(u, v)$	residual capacity on edge e and (u, v)
$f(u, v)$	flow value on edge (u, v)
F	net flow value, $F = \sum_{u \in V} f(u, t)$
P	path

Table 3.1: Notations

algorithms apply push-relabel style operation in handling edge capacity reduce, which is the key component that determines the performance. This push-relabel method involves great amount of operations in neighbor search, flow push and node relabel, thus has higher empirical computational complexity. Note that max-flow has the solution of either push relabel or augmenting path [12]. Apparently, incremental max-flow via augmenting path is still left as an open question.

3.2 Preliminary

Definition 3.2.1. *A flow on G is a real valued function $f()$ if the following conditions are satisfied:*

$$f(u,v) = -f(v,u), \quad \forall (u,v) \in V \times V; \tag{3.1a}$$

$$f(u,v) \leq C(u,v), \quad \forall (u,v) \in V \times V; \tag{3.1b}$$

$$\sum_{u} f(u,v) = 0, \quad \forall v \in V \setminus \{s,t\}. \tag{3.1c}$$

Let net flow $F = \sum_{u \in V} f(u,t)$ be the summation of flows into sink t. Then, the max-flow problem is to determine a flow from s to t with the maximum net flow F. In the rest of this work, we denote the direction from s to t as $s - t$.

Augmenting path algorithm stores information about the distribution of the current $s - t$ flow F among the edges of G using a residual graph $R = (V, E, R)$. The topology of R is identical to G (i.e., G and R share the same V and E), but $R(e)$ the capacity of edge e in R reflects the residual capacity of the same edge in G given the amount of flow already in the edge. At the initialization, there is no flow from the source to the sink ($F = 0$) and edge capacities in the residual graph R are equal to the original capacities in G i.e. $R(e) = C(e), \forall e \in E$.

Augmenting path algorithm is an iterative procedure of the following two steps: 1) find $s - t$ path using Breadth-First Search (BFS). The resulting path P is a set of edges with positive residual capacity laid end to end connecting s to t, such as $P = \{(s,u),(u,v),(v,t) \mid R(s,u) > 0, R(u,v) > 0, R(v,t) > 0\}$.

2) augment the $s - t$ path found above. We firstly find the max amount of flow can go through this path P, which is termed augmentation value and denoted as Δ_P in the rest of this work. As it is a bottleneck problem here, Δ_P can be calculated as the minimum residual capacity of the whole path $\Delta_P = min(R(u,v) \mid \forall (u,v) \in P)$. Next, we send Δ_P flow through path P in R as,

$$\begin{aligned} R(u,v) &= R(u,v) - \Delta_P, \forall (u,v) \in P \\ R(v,u) &= R(v,u) + \Delta_P, \forall (u,v) \in P \end{aligned} \tag{3.2}$$

The above two steps are iteratively executed until no more $s - t$ path can be found. Algorithm 8 gives the pseudo code of batch Augmenting Path.

As a result of max-flow search through Algorithm 8, we obtain a residual graph R. From max-flow to min-cut, we simply perform BFS or DFS on R to find the

Algorithm 8 Augmenting Path Max-flow Batch Learning

Input: $G = (V, E, C)$, s and t.

Output: R and F.

1: Initialize $R(e) = C(e), \forall e \in E$, $F = 0$;
2: Find a $s - t$ path P from the initial residual graph R;
3: **while** There is a $s - t$ path P **do**
4: Compute the amount of flow for augmentation: $\Delta_P = min(R(u, v) \mid \forall (u, v) \in P)$;
5: Augment the path P, via updating the residual graph as $R(u, v) = R(u, v) - \Delta_P, \forall (u, v) \in P$ and $R(v, u) = R(v, u) + \Delta_P, \forall (u, v) \in P$;
6: Update the flow value as $F = F + \Delta_P$;
7: Find a $s - t$ path P from the updated residual graph R.
8: **end while**

set of nodes S reachable from s, and define $T = V \setminus S$, then (S, T) is the $s - t$ min-cut.

Before the further derivation, let's review some basic property of the residual graph R. Recall the augmentation procedure in Algorithm 8, for any edge (u, v) if Δ_P flow is sent through it, we reduce $R(u, v)$ and increase $R(v, u)$ by Δ_P which means Δ_P capacity is taken from $R(u, v)$ (what left is the capacity available for later use) and expanded for $R(v, u)$ (available capacity can be used for sending flow through (u, v)). Thus, in any state of Algorithm 8, have

$$R(u, v) + R(v, u) = C(u, v) + C(v, u), \quad \forall (u, v) \in E. \tag{3.3}$$

The flow go through any edge (u, v) can be traced by

$$f(u, v) = C(u, v) - R(u, v) = R(v, u) - C(v, u), \quad \forall (u, v) \in E. \tag{3.4}$$

Note that flow conditions (3.1) holds for (3.4).

Now we can give the condition for R being the residual graph of G.

Theorem 3. *Given a graph $G = (V, E, C)$ and a pseudo residual graph R, compute f through (3.4), if (3.3) and (3.1) are satisfied, R is the residual graph for G.*

Then we give the termination criteria for Algorithm 8 as

Theorem 4. *A flow F stored on R is a max-flow for G if and only if the residual graph R contains no $s-t$ path.*

The proof of above theorems can be found in [62] and [12].

3.3 Proposed Incremental and Decremental Max-flow Algorithm

3.3.1 Incremental Max-Flow Setup

According to the batch max-flow stated in Section 3.2, a max-flow model is represented as a residual graph R. Thus the goal of incremental decremental max-flow is to update R in response to graph update due to the changes of data.

Given graph $G = (V, E, C)$ and its updated graph $G' = (V', E', C')$. We observe four types of graph change: edges deleted, nodes deleted, nodes added, and edges added. For edges deleted, an edge with positive capacity indicates a sequential graph operation on G: reduce first the capacity of the edge to zero, followed by the deletion of this edge. Similarly for edges added, an edge with positive capacity implies a two-step graph operation: 1) add edge with zero capacity and 2) increase edge capacity to $C'(e)$. In general, we summarize the following six categories graph operation, by which G can be transformed to G'.

(a) a set of edges E_r have capacity C_r to be reduced (i.e., $C'(e) = C(e) - C_r(e), \quad \forall e \in E_r$);

(b) a set of edges E_d to be deleted (i.e., $e \in E \quad e \notin E', \quad \forall e \in E_d$);

(c) a set of nodes V_d to be deleted (i.e.,$v \in V \quad v \notin V', \quad \forall v \in V_d$);

(d) a set of nodes V_a to be added (i.e., $v \notin V \quad v \in V', \quad \forall v \in V_a$);

(e) a set of edges E_a to be added (i.e., $e \notin E \quad e \in E', \quad \forall e \in E_a$);

(f) a set of edges E_g with each edge e capacity to be increased by $C_g(e)$ (i.e., $C'(e) = C(e) + C_g(e), \quad \forall e \in E_g$).

As updating residual graph R in response to all changes of G' against G, we combine all above six categories operations into one task list. For each step graph update, we pick up a set of tasks from the list, apply the tasks to G, and conduct the corresponding graph update on R meanwhile always keeping R to be the residual graph of G. Afterwards, we remove the processed tasks from current task list. This iteration continues until current task list is empty (i.e., G becomes G'). Consequently, we obtain the updated residual graph R'. Here, we address decremental learning first instead of the other way around, because decremental learning deducts the scale of graph which reduces the complexity of incremental max-flow learning.

3.3.2 Decremental Max-Flow

In the case a), an edge $(u, v) \in E_r$ has capacity $C_r(u, v)$ to be reduced. Consider edge capacity has non-negativity constraint. Thus $C_r(u, v)$ is required to be no greater than initial capacity $C(u, v)$.

If $R(u, v) \geq C_r(u, v))$ which means there is enough "unused" capacity for this reduce, then we simply reduce $R(u, v)$ by $C_r(u, v)$ and we have R' as

$$\begin{aligned} R'(u, v) &= R(u, v) - C_r(u, v) \\ R'(e) &= R(e) \mid \forall e \in E \setminus (u, v). \end{aligned} \tag{3.5}$$

Lemma 5. *If $R(u, v) \geq C_r(u, v)$, then R' in (3.5) is the residual graph of $G' = (V, E, C')$, where $C'(u, v) = C(u, v) - C_r(u, v)$ and $C'(e) = C(e) \mid \forall e \in E \setminus (u, v)$. In simple capacity reduce, the max-flow value remains $F' = F$.*

Proof. As have $R'(u, v) = R(u, v) - C_r(u, v)$, $C'(u, v) = C(u, v) - C_r(u, v)$, $R'(e) = R(e) \mid \forall e \in E \setminus (u, v)$ and $C'(e) = C(e) \mid \forall e \in E \setminus (u, v)$, thus (3.3) holds for R' and $G' = (E, V, C')$. Also we have $f_{G'}(u, v) = C'(u, v) - R'(u, v) = C(u, v) - R(u, v) = f_G(u, v)$, thus $f_{G'}$ satisfies (3.1). Applying Theorem3, R' is the residual graph of $G' = (V, E, C')$. As $f_{G'}(e) = f_G(e) \mid \forall e \in E$, the max-flow value remains. □

If $R(u, v) < C_r(u, v)$, which follows that there is not enough residual capacity left to be reduced due to some capacity on (u, v) is occupied by the current flows, then we need to release the occupied capacity before reduce.

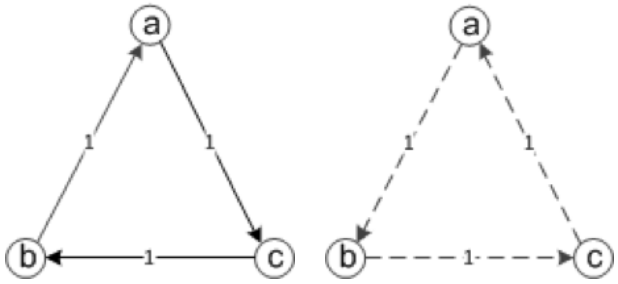


Figure 3.1: An example of cycle flow. (a) the residual graph; (b) the actual flow.

The current flow here can be either cycle flows or existing $s-t$ flows. Figure 3.1 gives an example of cycle flow. It is worth noting that the existence of a cycle flow may not only due to the input graph has a cycle. It is possible to form a cycle flow in the flow updating (adding and removing) process due to certain coincidences. Figure 3.2 gives an example of the formation of a cycle flow during continuous flow changing.

If capacity on (u, v) is occupied by cycle flows, then we release the capacity by canceling the cycles, for which we firstly find cycles by searching $u - v$ path with positive residual capacity from the residual graph, then cancel the located cycles by sending the same amount of flow in a revised direction along the cycles. Figure 3.3 gives an example of the cycle cancellation. Note that canceling a cycle flow does not change current $s-t$ flow, as cycle flow has no overlap with current $s-t$ flow.

In the case that capacity on (u, v) is occupied by $s-t$ flow. Let Σ be the amount of capacity to be released, clearly $\Sigma = C_r(u, v) - R(u, v)$. To release Σ capacity on (u, v), we send Σ flow from sink t to source s through a number (i.e. could be more than one) of paths go through (u, v). Note that a single $t - s$ path passing (u, v) may not be able to deliver required Σ flow.

As discussed before, augmentation is about sending flow from s to t. Contrarily, sending flow reversely from t to s means a reversed direction augmentation. We name the operation as de-augmentation. Similar to augmentation, de-augmentation is an iterative process, and each iteration consists of three steps: a) find a $t - v - u - s$ path P (i.e., a $t - v$ plus a $u - s$ path); b) determine the

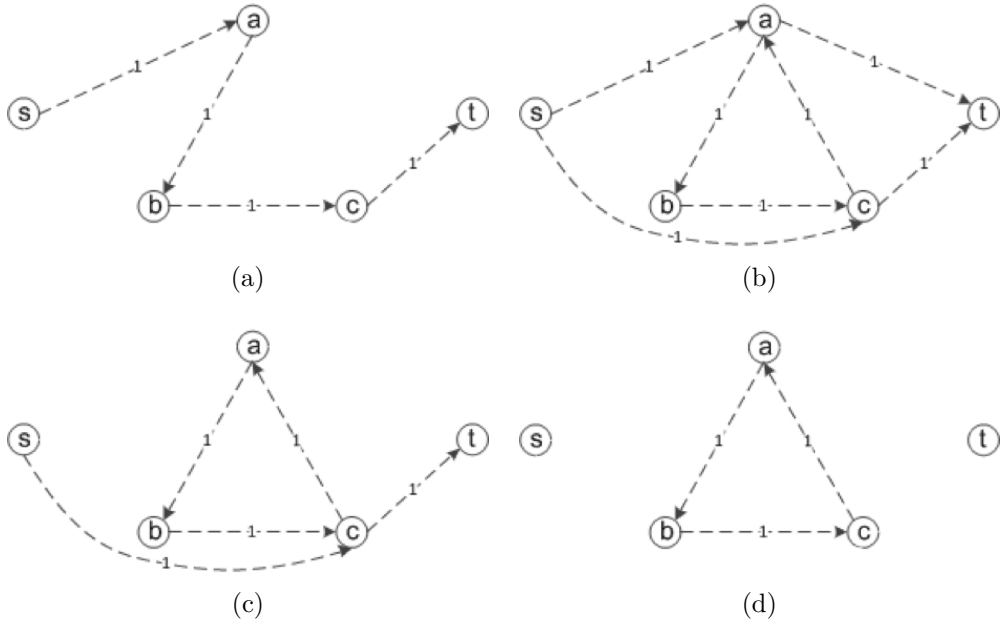


Figure 3.2: An example of the formation of a cycle flow in flow changing. (a) the initial $s-t$ flow $s-a-b-c-t$; (b) a new $s-t$ flow $s-c-a-t$ is added due to newly inserted edges; (c) flow $s-a-t$ removed due to edge removal; (d) flow $s-c-t$ removed due to edge removal, what left is a cycle flow $a-b-c-a$.

amount of flow to de-augment $\Omega = min(\{\Sigma, R(e) \mid \forall e \in P\})$; and c) sent Ω capacity through $t-v-u-s$ path P by updating residual graph.

The steps of one iteration cycle cancellation and de-augmentation are described in Algorithm 9.

Lemma 6. *Algorithm 9 output R' is the residual graph of $G = (V, E, C)$, and F' is the flow value on R'.*

Proof. As R' is initialized by residual R and only edges on path P are updated, so condition (3.3) and (3.1) hold for all remaining nodes and edges of R' except those on path P. As path P edges are updated by $R'(u,v) = R'(u,v) - \Omega, \forall (u,v) \in P$ and $R'(v,u) = R'(v,u) + \Omega, \forall (u,v) \in P$, all path P edges lose Ω capacity and their revised edges receive Ω capacity. This means, the cycle flow is canceled and the $s-t$ flow remains unchanged for cycle flow case. In de-augmentation case, a

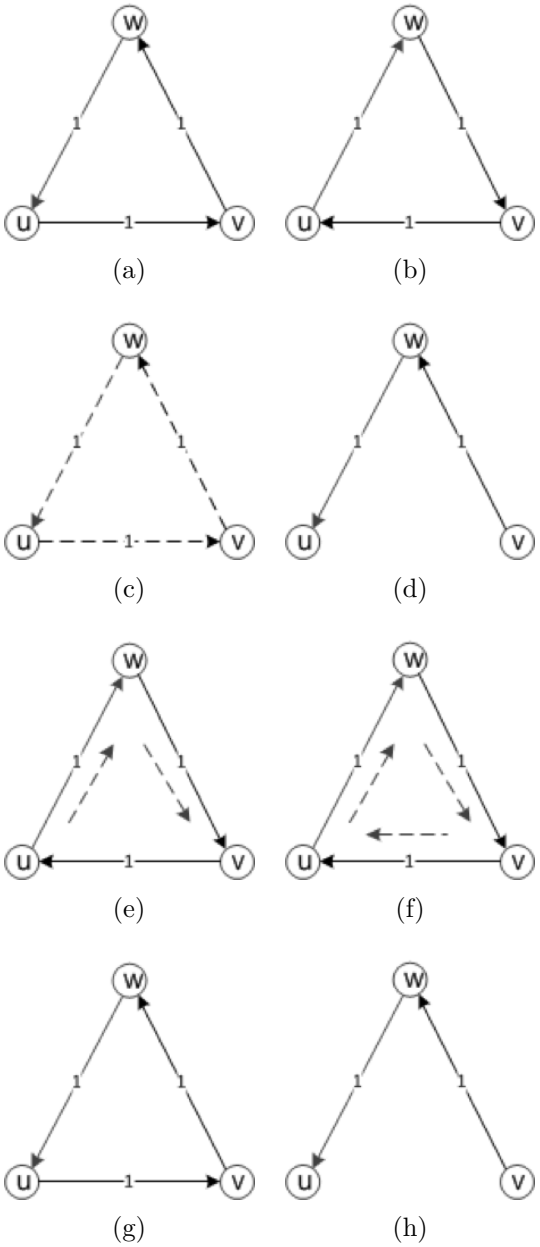


Figure 3.3: An example of cycle flow cancellation. (a) initial graph; (b) initial residual graph; (c) initial flow; (d) objective graph, where edge (u, v) need to be removed from initial graph; (e) a $u - v$ path is found in current residual graph; (f) the complete cycle $u - w - v - u$ is found; (g) the residual graph after sending 1 flow along cycle $u - w - v - u$; (h) now edge (u, v) can be removed safely.

Algorithm 9 Cycle Cancellation and De-augmentation

Input: $G = (V, E, C)$, R, F, s, t, (u, v) and Σ.

Output: R', F' and Ω.

1: Initialize $R' = R$;
2: **if** A $u - v$ path P_{u-v} can be found from R' **then**
3: Form a complete cycle path as $P = \{P_{u-v}, (v, u)\}$;
4: Compute the flow value in the cycle P as $\Omega = min(R(e) \mid \forall e \in P)$;
5: Cancel the cycle P, via updating the residual graph as $R'(u, v) = R'(u, v) - \Omega, \forall (u, v) \in P$ and $R'(v, u) = R'(v, u) + \Omega, \forall (u, v) \in P$;
6: $F' = F$;
7: **else if** A $t - v$ path P_{t-v} and a $u - s$ path P_{u-s} can be found from residual graph R' **then**
8: Form a complete path to de-augment as $P = \{P_{t-v}, (v, u), P_{u-s}\}$;
9: Compute the amount of flow to de-augment as $\Omega = min(\Sigma, \{R(e) \mid \forall e \in P\})$;
10: De-augment the path P by Ω, via updating the residual graph as $R'(u, v) = R'(u, v) - \Omega, \forall (u, v) \in P$ and $R'(v, u) = R'(v, u) + \Omega, \forall (u, v) \in P$;
11: Compute $F' = F - \Omega$.
12: **end if**

Ω flow is removed form the original $s-t$ flow, so that $F' = F - \Omega$. Also, condition (3.3) and (3.1) are satisfied for these path P nodes and edges. By Theorem 3, R' is a residual graph of $G = (V, E, C)$. □

Consdier we can only release Ω capacity on (u, v) by Algorithm 9. To release $\Sigma (\Sigma \geq \Omega)$ capacity, we call Algorithm 9 iteratively until Σ capacity is released in total. Next we apply simple reduce to $R(u, v)$. The complete procedure is given in Algorithm 10. Note that Algorithm 9 is called here just once when the graph is an unit graph. This is because one de-augmentation releases all flow on (u, v).

By Lemma5 and 6, we easily know that

Lemma 7. *Algorithm 10 output R' is the residual graph of $G' = (V, E, C')$, and F' is the flow value of R'. Here, $C'(u, v) = C(u, v) - C_r(u, v)$ and $C'(e) = C(e) \mid \forall e \in E \setminus (u, v)$.*

Algorithm 10 Capacity Reduce through Cycle Cancellation and De-augmentation

Input: $G = (V, E, C)$, R, F, s, t, (u, v) and $C_r(u, v)$

Output: $G' = (V, E, C')$, R' and F'.

1: Initialize $R' = R$, $G' = G$, $F' = F$;
2: Compute the total amount to de-augment $\Sigma = C_r(u, v) - R(u, v)$;
3: **while** $\Sigma > 0$ **do**
4: Conduct de-augmentation through Algorithm 9 to update R', F' and calculate Ω;
5: Update Σ by $\Sigma = \Sigma - \Omega$;
6: **end while**
7: Apply simple capacity reduce on (u, v) as $R'(u, v) = R'(u, v) - C_r(u, v)$ and $C'(u, v) = C'(u, v) - C_r(u, v)$

Remind that we have a set of edges E_r whose capacity needs to be reduced for max-flow update. Firstly, we perform simple reduce on those edges that are applicable and have $R(e) \geq C_r(e)$. For remaining edges, we conduct iteratively the following two operations until all edges in E_r are addressed: a) apply Algorithm 10 on an edge to release its capacity, and b) conduct simple reduce on those edges that newly become applicable due to operation a). Algorithm 11 presents the steps of capacity reduce on E_r.

In Algorithm 11, simple reduce (i.e., line 9 and 10) is conducted when Algorithm 10 is called. This is because that edges originally not applicable for simple capacity reduce may become applicable (i.e., satisfy the condition of line 9) after Algorithm 10 is executed. For instance, we have a $s - u - t$ path that carries unit flow. Here, neither (s, u) nor (u, t) is applicable for simple reduce because their capacities are occupied by the flow. When we apply Algorithm 10 to release residual capacity on (s, u), the algorithm releases actually the residual capacity of the whole $s - u - t$ path including (s, u) and (u, t). Therefore the capacity release on (s, u) turns (u, t) into being applicable for simple reduce.

Further based on Lemma 5 and 7, we have

Lemma 8. *Algorithm 11 output R' is the residual graph of $G' = (V, E, C')$, F' is the flow value in R'. Here $C'(e) = C(e) - C_r(e) \mid \forall e \in E_r$ and $C'(e) = C(e) \mid \forall e \in$*

Algorithm 11 Capacity Reduce on E_r

Input: $G = (V, E, C)$, R, F, s, t, E_r and C_r.

Output: $G' = (V, E, C')$, R' and F'.

1: Initialize $R' = R$, $C' = C$, $G' = (V, E, C')$, $F' = F$;
2: Find subset E_{sr} in E_r such that $R'(e) \geq C_r(e) \mid \forall e \in E_{sr}$;
3: Conduct simple capacity reduce on each edge in E_{sr} as $R'(e) = R(e) - C_r(e)$, $C'(e) = C(e) - C_r(e) \mid \forall e \in E_{sr}$;
4: Delete edges in E_{sr} from E_r as $E_r = E_r \setminus E_{sr}$;
5: **while** E_r is not empty **do**
6: Pick the first edge (u, v) from E_r;
7: Apply Algorithm 10 on (u, v) to obtain updated G', R' and F';
8: Delete edge (u, v) from E_r as $E_r = E_r \setminus (u, v)$;
9: Find subset E_{sr} in E_r such that $R'(e) \geq C_r(e) \mid \forall e \in E_{sr}$;
10: Conduct simple capacity reduce on each edge in E_{sr} as, $R'(e) = R(e) - C_r(e)$, $C'(e) = C(e) - C_r(e) \mid \forall e \in E_{sr}$;
11: Delete edges in E_{sr} from E_r as $E_r = E_r \setminus E_{sr}$
12: **end while**.

$E \setminus E_r$.

On the other hand, R' may not carry the max-flow of G', as new $s - t$ paths might be formed by newly released edges and those edges originally have no flow carried. To compute max-flow on G', according to Theorem 4 we simply augment $s - t$ paths in R' until no more $s - t$ paths are found. For simplicity, we postpone this augmentation until graph G is expanded with new edges and nodes (i.e, graph update case (d), (e) and (f)), and carry out all augmentation works in one batch.

Now we address graph update case (b). We can safely delete all E_d edges in G' and R', as the capacity of these edges have been reduced to zero. For graph update case (c), we simply delete all V_d nodes in G' and R' simultaneously, as any edge associated with these nodes has already been deleted in case (b).

Figure 3.4 gives an example max-flow decremental learning. Recall the example in Fig. 2.2. Let Fig. 6.5a be the initial graph, our objective here is to remove edge (u, v) from Fig.6.5a and find the max-flow of Fig. 3.4a by updating residual graph

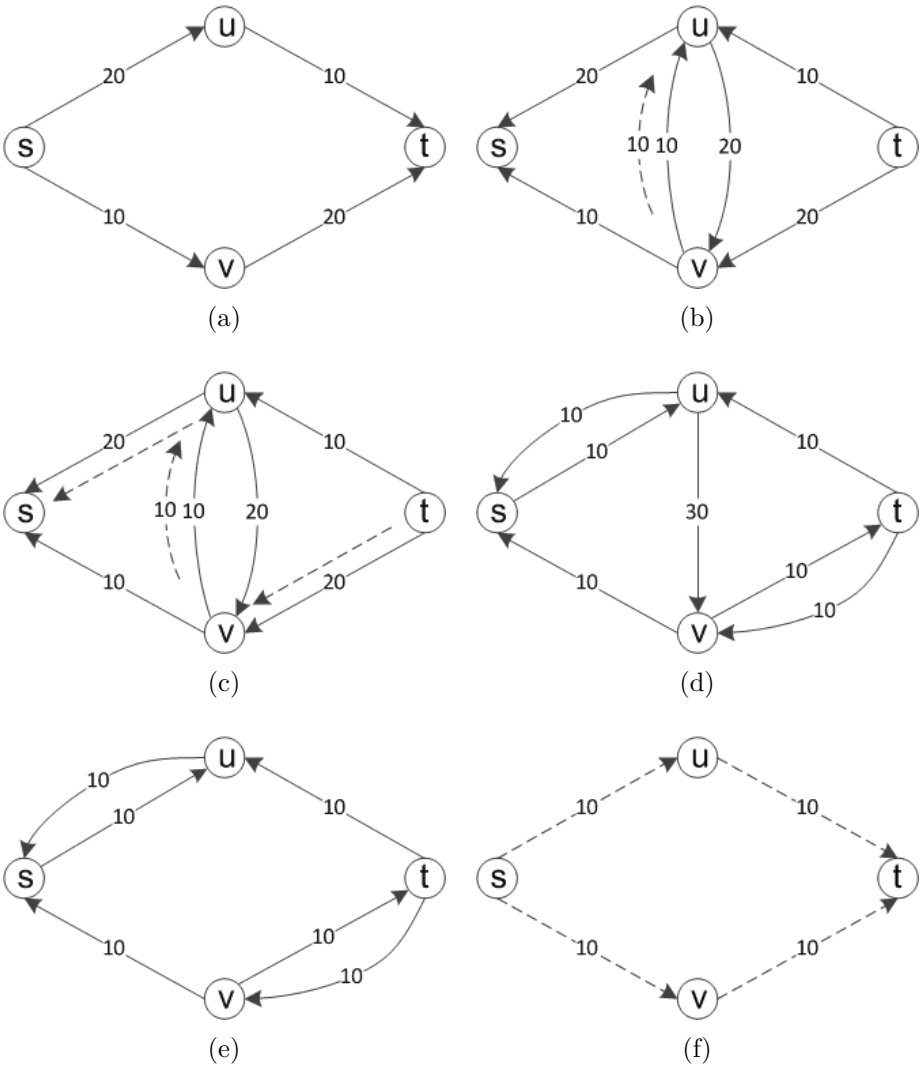


Figure 3.4: An example of decremental max-flow learning through de-augmentation. (a) objective graph; (b) initial residual graph, a flow of 10 should be sent from v to u in order to release the capacity on (u, v); (c) a $t - s$ path goes through (v, u) shown in dotted lines; (d) residual graph after de-augmenting the $t - s$ path, now (u, v) has enough capacity to be reduced; (e) reduce the capacity of (u, v) by 30 and remove the empty edge, and obtain the residual graph of (a); (f) the actual max-flow on (a).

Fig. 2.2c. Edge (u, v) has a capacity of 30 in which 10 is occupied by existing $s-t$ flow. Thus we firstly release 10 capacity on (u, v) through de-augmentation, then reduce the capacity of (u, v) to zero, finally remove the edge from the graph.

3.3.3 Incremental Max-flow

For graph update case (d), we simply add all V_a nodes into G' and R', respectively. Similarly for case (e), we expand G' and R' with edges in E_a. Note that R' is the residual graph of G' holds for case (b), (c), (d) and (e) since no capacity is changed in these updates and the topology of R' for all cases is kept the same as G'.

For graph update case (f), the capacity of each edge $(u, v) \in E_g$ is required to increase by $C_g(u, v)$. Thus, we increase $R(u, v)$ by $C_g(u, v)$,

$$\begin{aligned} R'(u, v) &= R(u, v) + C_g(u, v) \\ R'(e) &= R(e) \mid \forall e \in E \setminus (u, v). \end{aligned} \tag{3.6}$$

Lemma 9. *for (3.6) R' is the residual graph of $G' = (V, E, C')$, and the flow value keeps $F' = F$. Here $C'(u, v) = C(u, v) + C_g(u, v)$ and $C'(e) = C(e) \mid \forall e \in E \setminus (u, v)$.*

The proof of Lemma 9 is similar to that of Lemma 5, thus is omitted here.

Applying (3.6) to all edges in E_g, we have

$$\begin{aligned} R'(u, v) = R(u, v) + C_g(u, v) \mid \forall (u, v) \in E_g \\ R'(e) = R(e) \mid \forall e \in E \setminus E_g. \end{aligned} \tag{3.7}$$

Based on Lemma 9, we have the following guarantee on R',

Lemma 10. *for (3.7) graph R' is the residual graph of $G' = (V, E, C')$, and the flow value keeps $F' = F$. Here $C'(e) = C(e) + C_g(e) \mid \forall e \in E_g$ and $C'(e) = C(e) \mid \forall e \in E \setminus E_g$.*

As the result of graph update case (a) to (f), we have the updated G' and R'. By Lemma 8 and 10, R' is the residual graph of G'. In finding max-flow on G', we augment iteratively any $s-t$ path found on R' until no more $s-t$ path can be found. According to Theorem 4, the resulting R' from the augmentation carries the max-flow on G'. The complete procedure of incremental max-flow is given in Algorithm 12.

Algorithm 12 Incremental and Decremental Augmenting Path algorithm

Input: $G = (V, E, C)$, R, F, s, t, E_r, C_r, E_d, V_d, V_a, E_a, E_g, C_g.

Output: $G' = (V', E', C')$, R', F'.

1: Apply Algorithm 11 to conduct capacity reduce on E_r, and obtain $G' = (V, E, C')$, R' and F' ;
2: Delete edges in E_d from E as $E' = E \setminus E_d$
3: Delete nodes in V_d from V as $V' = V \setminus V_d$;
4: Add nodes in V_a into V' as $V' = V' \cup V_a$;
5: Add edges in E_a into E' as $E' = E' \cup E_a$;
6: Conduct simple capacity increase in E_g as (3.7);
7: Find a $s - t$ path P in the residual graph R';
8: **while** There is a $s - t$ path P **do**
9: Compute the amount of flow to augment as $\Delta_P = min(R'(u, v) \mid \forall (u, v) \in P)$;
10: Augment the path P, via updating the residual graph as $R'(u, v) = R'(u, v) - \Delta_P, \forall (u, v) \in P$ and $R'(v, u) = R'(v, u) + \Delta_P, \forall (u, v) \in P$;
11: Update the flow value as $F' = F' + \Delta_P$;
12: Find a $s - t$ path P from the updated residual graph R'
13: **end while**.

3.3.4 Complexity Analysis

According to [12], batch augmenting path algorithm takes $O(|V||E|^2)$ time to find a max-flow from a graph. Our graph as defined in Section 4.2.1 is basically an unit graph with most edges capacity as 1. In this case, batch augmenting path takes $O(F|E|)$ time for computing a max-flow. Here, F is the number of augmentation.

Consider decremental learning of max-flow. Proposed algorithm involves an iterative de-augmentation plus a followed iterative augmentation. For each de-augmentation iteration, BFS takes $O(|E_o|)$ time to find a $t - s$ path, where E_o is the set of edges occupied by current max-flow. The total number of de-augmentation is ΔF, which equals to the amount of flow lost in the de-augmentation step. In the worest case, the graph after de-augmentation requires additional ΔF augmentation to find the max-flow. Each augmentation

takes $O(|E|)$ time. Thus the overall complexity for decremental learning is $O(\Delta F|E_o| + \Delta F|E|) = O(\Delta F|E|)$.

For incremental learning of max-flow, proposed algorithm performs an iterative graph augmentation in response to newly added edges. The graph after edge adding has $|E'|$ edges in which $|E_o|$ edges are occupied by existing max-flow thus are not involved in the $s-t$ path search. In this sence, each augmentation costs $O(|E'| - |E_o|)$. Due to new edges added, $\Delta F = F' - F$ augmentations are required to caculate emerged flows, and the total cost on updating max-flow is $O(\Delta F(|E'| - |E_o|))$. As compared to batch retraining whoes complexity is $O(F'|E'|)$, proposed algorithm saves computational costs in reducing the number of augmentations and the scale of $s-t$ path search.

3.4 Experiments and Discussions

3.4.1 Experiment Setup

We compare proposed incremental max-flow with the preflow push based incremental max-flow in two scenarios: (1) graph continuously expanding; and (2) graph continuously shrinking. Meanwhile, we use batch augmenting path and batch preflow push max-flow as the baselines. All experiments are conducted on randomly generated unit graphs [12]. All algorithms are coded in Matlab and executed on a laptop with Intel i7 2.4GHz CPU and 8 MB memory.

For performance evaluation, we measure a set of variables on which we add "BA", "OA","BP" and "OP" to identify batch augmenting path, incremental augmenting path, batch preflow push and incremental preflow push algrorithms, respectively. The list of variable includes,

1. The ratio of edge number to node number in a graph, ENr;
2. The flow value of max-flow FV;
3. The number of augmentations conducted, $nAugBA$ and $nAugOA$;
4. The number of active nodes (i.e. the nodes that max-flow goes through) $nAnBA$ and $nAnOA$;

5. The ratio of active node number to total node number, $ANrBA$ and $ANrOA$;

6. The CPU time (in seconds) cost on finding max-flow tBA, tOA, tBP and tOP;

7. The $Gain$ of our algorithm with respect to batch augmenting path as

$$Gain = 1 - \frac{tOA}{tBA}; \tag{3.8}$$

8. The number of push and relabel operations $nPushBP$, $nRelabelBP$, $nPushOP$ and $nRelabelOP$.

3.4.2 Results of Learning Graphs Continuously Expanding

As learning expanding graphs, we start with an initial graph of 500 nodes and 500 edges. For each stage of expanding, additional 500 edges are added to the graph. Proposed incremental augmenting path and the incremental preflow push [63] are able to incrementally learn the graph while it is expanding. In contrast, the two batch max-flow algorithms have to learn from scratch for every stage of graph expanding.

Table 3.2 gives the comparison results in which the status of max-low leaning is observed in 20 stages of graph expanding. Consider proposed incremental max-flow is augmenting path based, we compare firstly proposed algorithm with batch augmenting path max-flow. As seen from the table, flow value (FV) grows consistently with the increase of ENr, which indicates more flows can be sent through the graph while it is expanding. After each stage of expanding, proposed algorithm identifies only those newly formed flows, whereas batch augmenting path has to find all flows in the expanded graph. This follows that the number of augmentations ($nAug$) for batch augmenting path, equals always to the accumulative sum of that for proposed incremental augmenting path,

$$nAugBA_i = nAugOA_1 + nAugOA_2 + \ldots nAugOA_i. \tag{3.9}$$

This is shown in column $nAugBA$ and $nAugOA$ of Table 3.2. Hereafter if any two valuables satisfy (3.9), we say these two valuables are accumulative equivalent.

ENr	FV	nAugBA	nAugOA	nAnBA	nAnOA	tBA	tOA	Gain	tBP	tOP	nPushBP	nRelabelBP	nPushOP	nRelabelOP
1	2	2	2	8	8	0.02	0.02	0.00	0.73	0.73	564	556	564	556
2	5	5	3	26	26	0.07	0.06	0.13	0.60	0.97	325	274	1174	1145
3	12	12	7	56	55	0.37	0.27	0.27	1.36	3.48	573	478	1853	1624
4	18	18	6	71	75	0.83	0.35	0.58	2.06	5.00	711	597	2094	1888
5	27	27	9	96	101	1.55	0.53	0.66	3.33	11.90	960	786	4155	3732
6	32	32	5	98	111	2.06	0.30	0.85	4.15	7.79	974	805	2318	2107
7	35	35	3	92	112	2.21	0.19	0.91	4.22	4.91	914	742	1365	1292
8	40	40	5	102	122	2.73	0.41	0.85	4.92	10.31	967	782	2452	2308
9	44	44	4	105	130	3.19	0.56	0.83	5.70	8.15	997	796	1832	1752
10	49	49	5	115	139	3.78	0.49	0.87	6.73	11.54	1129	898	2379	2258
11	53	53	4	116	149	4.06	0.39	0.90	6.68	8.95	1037	784	1688	1647
12	59	59	6	123	156	4.70	0.51	0.89	7.49	16.55	1075	816	2959	2812
13	64	64	5	131	163	5.32	0.51	0.90	8.63	15.26	1133	837	2503	2402
14	68	68	4	134	168	5.81	0.38	0.93	10.11	12.56	1219	894	1892	1837
15	71	71	3	136	172	6.27	0.33	0.95	10.37	12.27	1209	869	1750	1691
16	78	78	7	149	185	7.16	0.75	0.89	11.40	26.28	1242	872	3643	3536
17	81	81	3	155	193	7.57	0.40	0.95	13.06	14.29	1357	958	1859	1783
18	83	83	2	154	196	8.29	0.32	0.96	13.52	8.61	1325	923	1005	975
19	87	87	4	159	202	8.52	0.52	0.94	14.18	17.63	1333	915	2022	1966
20	92	92	5	168	207	9.41	0.62	0.93	14.80	30.94	1322	880	3402	3315

Table 3.2: Results for graph expanding on 500 nodes

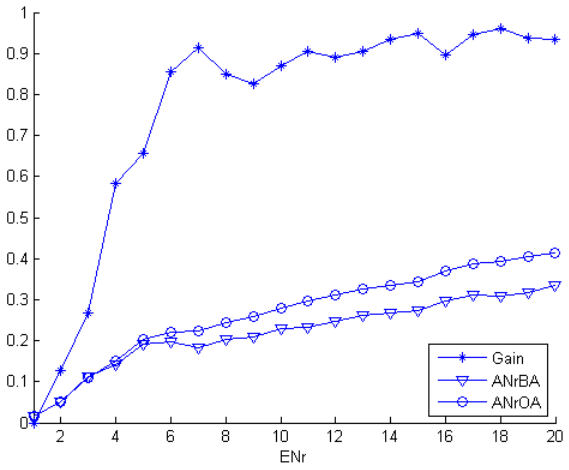


Figure 3.5: Gain and ANr for graph expanding

For CPU cost, the number of augmentations determines the running time. Consider $nAugBA$ and $nAugOA$ are accumulative equivalent. A quasi accumulative equivalence is expected on CPU cost, tBA and tOA. This expectation has been verified in corresponding columns of Table 3.2. By (3.8), the *Gain* of our algorithm increases steadily from 0 towards 1. This is demonstrated in both Table 3.2 and Fig. 3.5. However, the achieved *Gain* is found at the cost of a more complex max-flow (in terms of active node ratio), since $ANrOA$ in Fig. 3.5 is constantly higher than $ANrBA$. This can be explained that our algorithm seeks max-flow from edges observed so far (i.e., local optimal) at each stage, and combine all local results into the final max-flow.

Next we consider comparison to preflow push algorithms, we can see that proposed incremental augmenting path runs always faster than incremental preflow push. Surprisingly, we also find that the incremental preflow push executes even slower than batch retraining in most of cases. This phenomenon can be explained by the fact that incremental preflow push handles edge adding in a sequential way, one edge at a time. Each edge adding leads to a small set of push and relabel operations. When a large chunk of edges are added, the total number of push and relabel operations can be greater than that of batch retraining, which is shown in Table 3.2. This indicates that the incremental preflow push is inefficient when

ENr	FV	nAugBA	nAugOA	nDeAugOA	nAnBA	nAnOA	tBA	tOA	Gain
20	81	81	81	0	147	147	8.02	8.02	0.00
19	80	80	5	6	149	152	7.76	1.05	0.86
18	74	74	5	11	139	150	7.04	1.27	0.82
17	68	68	4	10	130	131	6.36	0.96	0.85
16	61	61	10	17	122	137	5.52	1.97	0.64
15	54	54	3	10	110	125	4.72	0.87	0.82
14	51	51	1	4	106	108	4.40	0.32	0.93
13	50	50	10	11	108	114	4.26	1.56	0.63
12	44	44	6	12	101	117	3.67	1.12	0.69
11	42	42	6	8	102	109	3.46	0.86	0.75
10	37	37	5	10	93	105	2.98	0.94	0.68
9	32	32	9	14	85	100	2.48	1.36	0.45
8	28	28	9	13	76	97	2.01	1.29	0.36
7	24	24	5	9	71	80	1.60	0.70	0.56
6	23	23	10	11	70	84	1.35	0.99	0.26
5	20	20	5	8	68	70	0.99	0.53	0.47
4	14	14	8	14	49	73	0.48	0.83	-0.75
3	11	11	8	11	50	60	0.32	0.46	-0.43
2	6	6	5	10	35	58	0.12	0.57	-3.81
1	1	1	0	5	0	0	0.00	0.04	-7.87

Table 3.3: Results for graph shrinking on 500 nodes

edges are added in a chunk manner.

3.4.3 Results of Learning Graphs Continuously Shrinking

As learning from shrinking graphs, we start with an initial graph of 500 nodes and $10,000$ edges. For each stage of shrinking, 500 edges are removed from the current graph. Proposed incremental augmenting path and the incremental preflow push are able to decrementally learn the graph while it is shrinking, whereas the two batch algorithms have to learn the shrinked graph from scratch.

Table 3.3 gives the comparison results for 20 stages graph shrinking. We compare firstly proposed algorithm with batch augmenting path max-flow. As seen from the table, flow value (FV) reduces consistently with the decrease of ENr, which indicates fewer flows can be sent through the graph while it is shrinking. For each stage of shrinking, proposed algorithm conducts first de-augmentation to remove the flows that go through those $s-t$ paths with at least one edge removed. This reduces current flow value. Next, augmentation is carried out to find new $s-t$ paths through which new flows can be sent. This increases current flow value. Consider the case of unit graph, the actual flow value change is $\Delta FV = -nDeAugOA + nAugOA$, since a $s-t$ path carries always unit flow. Recall that batch augmenting path needs to find all flows, thus $nAugBA = FV$

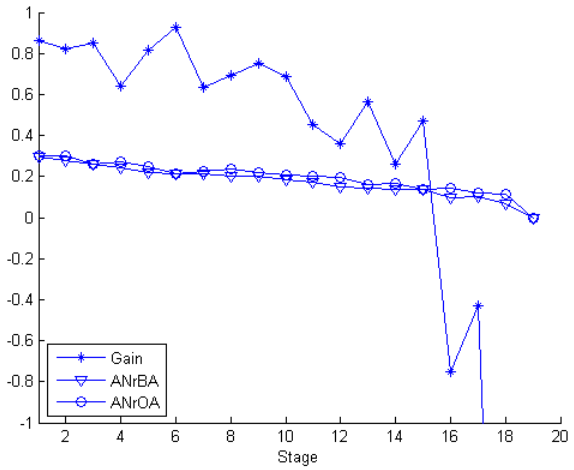


Figure 3.6: Gain and ANr for graph shrinking

holds for every stage of graph shrinking, and have

$$nAugBA_i = FV_i = FV_{i-1} - nDeAugOA_i + nAugOA_i. \tag{3.10}$$

This is shown in column FV to $nDeAugOA$ in Table 3.3.

For CPU cost, the running time for both batch and incremental augmenting path is mainly determined by the number of augmentation and de-augmentation. As seen in Table 3.3, $nAugOA+nDeAugOA$ fluctuates over stages, but $nAugBA$ reduces constantly with ENr. Correspondingly, tOA and tBA show the similar pattern of fluctuation and decrease, respectively. By (3.8), the $Gain$ of our algorithm decreases as graph shrinking, and become negative in the end. This is shown in both Table 3.3 and Fig. 3.6. Here, a straightforward conclusion can be made, proposed incremental augmenting path is effective when a small proportion of edges are removed from graph. When removal proportion goes above a certain threshold, incremental learning may take longer time than batch retraining. Similar to graph expanding, in handling graph shrinking proposed incremental augmenting path gives often more complex max-flow than that of batch retraining.

Secondly we compare proposed algorithm with two preflow push approaches. From our experiences, incremental preflow push converges often extremely slow on large graph decremental learning. Thus for this comparison, we carry out a

ENr	FV	nAugBA	nAugOA	nDeAugOA	nAnBA	nAnOA	tBA	tOA	Gain
10	37	37	37	0	39	39	0.29	0.29	0.00
9	36	36	6	7	39	39	0.27	0.10	0.63
8	33	33	5	8	39	39	0.24	0.10	0.59
7	29	29	5	9	37	37	0.20	0.10	0.50
6	24	24	5	10	36	37	0.16	0.10	0.38
5	20	20	4	8	30	35	0.13	0.08	0.37
4	13	13	3	10	23	26	0.08	0.07	0.07
3	10	10	4	7	18	18	0.05	0.06	-0.29
2	8	8	1	3	15	15	0.03	0.02	0.14
1	4	4	0	4	6	6	0.01	0.02	-0.57

ENr	FV	tBP	tOP	nPushBP	nRelabelBP	nPushOP	nRelabelOP
10	37	2.73	2.73	2237	1826	2237	1826
9	36	2.50	81.73	2161	1775	66320	58698
8	33	2.47	79.30	2135	1705	64826	56757
7	29	2.31	72.26	2106	1661	59376	51211
6	24	2.11	73.31	2028	1586	60285	50655
5	20	1.89	81.50	1847	1418	66911	54916
4	13	1.52	63.00	1611	1211	53884	42882
3	10	1.27	36.04	1447	1095	31499	24278
2	8	0.37	10.87	468	351	9966	7871
1	4	0.14	0.35	177	148	299	270

Table 3.4: Results for graph shrinking on 50 nodes

simple experiment on 50-node graph. Table 3.4 gives the results. As we can see, proposed incremental augmenting path excutes over 100 times faster than the incremental preflow push in handling graph shrinking, and the incremental preflow push is again found even slower than the batch preflow push retraining.

3.5 Summary

In this chapter, a novel augmenting path based incremental max-flow algorithm is developed to update max-flow whenever graph changes. The theoretical guarantee of proposed incremental max-flow is the learning effectiveness of incremental max-flow equals to that of batch max-flow retraining. This equivalence is proved both theoretically and experimentally. As compared to incremental preflow push, the converging speed of proposed algorithm is found much faster. This is because that our algorithm handles multiple graph changes in one batch, whereas incremental preflow push processes merely one graph change at a time.

4 Implement the Proposed Incremental Max-flow on Semi-supervised Learning

4.1 Introduction

In this chapter, the proposed incremental max-flow algorithm is applied to upgrade an existing batch semi-supervised learning algorithm know as graph minicuts to be an incremental algorithm.

In big data era data volume and velocity increase fast. Labeling a small sample of data is the only feasible way to learn classification from a real world big data. This causes that incoming data contains often a large portion of unlabeled instances. Semi-supervised learning is constructive in that unlabeled data can be utilized to facilitate machine learning and improve accuracy.

4.1.1 Semi-supervised Learning

In semi-supervised learning, unlabeled data helps modify or reprioritize hypotheses obtained from labeled data alone [64, 65]. Provided with the same amount of labelled data, semi-supervised learning gives often higher learning accuracy than supervised learning does when assumptions such as smoothness, cluster and manifold are met [66,67]. On the other hand, labeled instances are often difficult, expensive, or time consuming to obtain in real world applications, as they require the efforts of experienced human annotators. In this sense, semi-supervised learning is capable of enhancing the learning effectiveness with less human efforts.

Semi-supervised learning uses unlabeled data to facilitate learning accuracy.

This impact works also for incremental learning. Here, we review briefly recent works with a focus on how unlabeled data is being used for modeling. In [68], unlabeled samples that have the same distributions with the target domain are utilized with labeled data to train a transfer semi-supervised SVM. Zhang et al [6] cluster unlabeled data, then ensemble the obtained clusters with classifiers built on labeled data to deal with the concept drift of data streams. In [69] unlabeled data are utilized to train a Gaussian mixture model determining the voting weight for each weak classifier trained from labeled data received at different learning stage. Besides, there are also other ways for unlabeled data to be used in incremental semi-supervised learning [70–75]. All above works set the ratio of unlabelled data within the range of 90% to 99%, but haven't yet addressed even higher ratio.

4.1.2 Graph Mincuts Algorithm

Graph mincuts [76] is a graph based semi-supervised learning algorithm featured by its non-parametric, discriminative, and transductive nature. The basic assumption of graph mincuts is simple but concrete: samples with smaller distance are more likely to be in the same class. This is also called smoothness assumption in literature [65]. In graph mincuts, graph is constructed from both labeled and unlabeled data according to samples closeness (i.e., similarity). For classification, min-cut is applied to split the graph into two isolated parts by removing an edge set with minimum total weight. Here, min-cut ensures that minimized number of similar sample pairs are classified into distinct classes. Although graph mincuts shows great performance in learning from datasets with only small portion of samples labeled, graph mincuts, as a batch learner, can be only used for learning from static datasets.

Consider incremental learning of graph mincuts, a straightforward solution is to derive from an existing batch graph mincuts to its corresponding incremental mincuts.

The graph learned from labeled and unlabeled data is updated dynamically for accommodating data adding and retiring. For learning such non-stationary graph, proposed incremental max-flow addresses all possible graph changes in two categories: capacity decrease and increase on edges. As a response, our algorithm

de-augments paths to enable capacity decrease and augment paths after capacity increase to compute the up-to-date max-flow.

For the convenience of algorithm derivation and clarity of presentation, we summarize most notations used in this chapter in Table 4.1.

Notation	Descriptions
$\boldsymbol{X}$	instance matrix
$\boldsymbol{x}_i$	the i-th instance
L_+, L_-, U	index set of positive, negative, unlabeled samples
$\boldsymbol{D}$	distance matrix, $d_{i,j} = dist(\boldsymbol{x}_i, \boldsymbol{x}_j)$
G	weighted graph, $G = (V, E, C)$
R	residual graph, $R = (V, E, R)$

Table 4.1: Notations

4.2 Preliminary

Graph mincuts has been used for classification learning from both labelled and unlabelled data [76]. The idea is straightforward but concrete: samples with smaller distance are more likely to be the same class. Let $\boldsymbol{X}$ be a labeled and unlabeled dataset. We assume each sample of the dataset has a unique index, and L_+, L_- and U be the index set of positive, negative and unlabeled samples, respectively. Graph mincuts constructs a weighted graph G according to sample closeness (similarity) and then split G by removing an edge set with minimum total weight. For semi-supervised learning, min-cut ensures here minimized number of similar sample pairs are classified into distinct classes.

4.2.1 Graph Construction

We consider learning a weighted graph G from data $\boldsymbol{X}$. A weight graph $G = (V, E, C)$ consists of a finite node set V, an edge set $E \in V \times V$, and a wight function $C : E \to R^+$ (called capacity hereafter) which associates a positive weight value $C(e)$ with each edge $e \in E$.

As determining nodes V, each sample $\boldsymbol{x}_i$ either labeled or unlabelled is represented by a vertex v_i. To make the min-cut (max-flow) feasible, two virtual nodes v_+ and v_- are created for positive and negative class respectively. Thus $V = \{v_i, v_+, v_- | \forall i \in L_+ \cup L_- \cup U\}$.

As determining edges E and capacity C, two steps are taken in to connect nodes in V:

Firstly, each labeled node is connected to the virtual node with the same class label, where the edge has an infinite weight as

$$\begin{aligned} C(v_i, v_+) &= \infty, \quad \forall i \in L_+ \\ C(v_i, v_-) &= \infty, \quad \forall i \in L_-. \end{aligned} \tag{4.1}$$

This setup prevents labeled samples from being classified into the opposite class, as any cut associated with an infinite edge is not a min-cut [76].

Secondly, we calculate the pairwise distance matrix $\boldsymbol{D}$ where $d_{i,j} = dist(\boldsymbol{x}_i, \boldsymbol{x}_j)$ to measure the Euclidean similarity between any of two samples. Then, the remaining sample nodes are connected by edges of weight 1 according to the one of the following connecting rules defined on $\boldsymbol{D}$:

1. Mincut-N, each unlabeled sample is connected to its N nearest neighbors in terms of pairwise similarity shown in matrix $\boldsymbol{D}$. To avoid having isolated area in graph, one of the N-nearest neighbors is forced to be labeled. In other words, each unlabeled sample is connected to its nearest labeled neighbor and $N-1$ nearest unlabeled neighbors;

2. Mincut-δ, a pair of samples is connected if their distance $d_{i,j}$ is less than a given threshold δ. Here, δ is determined through multiple attempts to meet one of the following conditions,

   (a) Mincut-δ_0 is to choose the maximum δ on which graph G has a 0 valued min-cut;

   (b) Mincut-$\delta_{1/2}$ is to find δ forms the largest connected component by 1/2 number of data points.

4.2.2 Solve Min-Cut

Given G learned from a static dataset $\boldsymbol{X}$, we consider solving min-cut (i.e., to find a min-cut splitting G). According to [13] and [14], min-cut is equivalent to max-flow. Applying any max-flow algorithm such as augmenting path on G, the max-flow is obtained in form of a residual graph R.

From max-flow to min-cut, we simply perform Width First Search (BFS) or Depth First Search (DFS) on R to find the set of nodes S that are reachable from s, and define $T = V \setminus S$. Then (S, T) is the $s - t$ min-cut.

Having the min-cut result, the nodes in the graph are splitted into to two isolated sets, and so for the unlabeled samples. Thus we have all unlabeled samples classified.

4.3 Implement Incremental Decremental Max-flow on Incremental Semi-supervised Learning

Let $\mathcal{C}$, $\mathcal{A}$ and $\mathcal{R}$ be the set of sample index for current data, data to be added and removed respectively. Given newly acquired dataset $\boldsymbol{X}^{\mathcal{A}}$ and dataset $\boldsymbol{X}^{\mathcal{R}}$ to be retired from current data $\boldsymbol{X}^{\mathcal{C}}$. The goal of our work is to develop incremental decremental function $f()$ capable of updating min-cut on $\boldsymbol{X}^{\mathcal{C}}$ in response to data updates $\boldsymbol{X}^{\mathcal{A}}$ and $\boldsymbol{X}^{\mathcal{R}}$ as

$$M' = f(M, \boldsymbol{X}^{\mathcal{A}}, \boldsymbol{X}^{\mathcal{R}}) \tag{4.2}$$

where M is current min-cut on $\boldsymbol{X}^{\mathcal{C}}$ and M' is updated min-cut computed by incremental decremental learning on data updates $\boldsymbol{X}^{\mathcal{A}}$ and $\boldsymbol{X}^{\mathcal{R}}$. In principle, M' should be exact same as the batch min-cut on the updated dataset,

$$f(M, \boldsymbol{X}^{\mathcal{A}}, \boldsymbol{X}^{\mathcal{R}}) = g(\boldsymbol{X}^{\mathcal{C}} \setminus \boldsymbol{X}^{\mathcal{R}} \cup \boldsymbol{X}^{\mathcal{A}}) \tag{4.3}$$

$$\tag{4.4}$$

where $g()$ is the batch learning system corresponding Algorithm 8.

As described above, a batch graph mincuts system consists of two main steps: 1. construct an undirected graph from the labeled and unlabeled samples based

on their close neighbor relationship; and 2. conduct min-cut separation on above such graph through max-flow optimization. The objective of proposed incremental mincut, namely oMincut, is to update the batch min-cut in response to any newly acquired samples and/or samples retired. Accordingly, proposed incremental system is about the two steps updating, incremental graph updating and incremental decremental max-flow.

4.3.1 Graph Updating

The objective of graph updating is to update current graph in response to data updates. Corresponding to the steps of graph construction for batch min-cut learning, we update first the pairwise distance matrix.

For removing a subset $\boldsymbol{X}^{\mathcal{R}}$ from current data $\boldsymbol{X}^{\mathcal{C}}$, we simply calculate the residual index set $\mathcal{C} \setminus \mathcal{R}$ and apply to $\boldsymbol{D}^{\mathcal{C}}$, then we have the updated pairwise distance matrix as

$$\boldsymbol{D}' = \boldsymbol{D}^{\mathcal{C}\setminus\mathcal{R}} \tag{4.5}$$

For adding data $\boldsymbol{X}^{\mathcal{A}}$ into current data $\boldsymbol{X}^{\mathcal{C}}$, we first calculate $\boldsymbol{X}^{\mathcal{A}}$ pairwise distance matrix as $\boldsymbol{D}^{\mathcal{A}}$ in which $d^{\mathcal{A}}_{i,j} = dist(\boldsymbol{x}^{\mathcal{A}}_i, \boldsymbol{x}^{\mathcal{A}}_j)$. Next we calculate the distance matrix in between $\boldsymbol{X}^{\mathcal{C}}$ and $\boldsymbol{X}^{\mathcal{A}}$ as $\boldsymbol{D}^{\mathcal{C}\mathcal{A}}$ in which $d^{\mathcal{C}\mathcal{A}}_{i,j} = dist(\boldsymbol{x}^{\mathcal{C}}_i, \boldsymbol{x}^{\mathcal{A}}_j)$. Then, we have the updated pairwise distance matrix computed as

$$\boldsymbol{D}' = \begin{bmatrix} \boldsymbol{D}^{\mathcal{C}} & \boldsymbol{D}^{\mathcal{C}\mathcal{A}} \\ \boldsymbol{D}^{\mathcal{C}\mathcal{A}^T} & \boldsymbol{D}^{\mathcal{A}} \end{bmatrix}. \tag{4.6}$$

In practice, we address data adding and retiring in one batch. In other words, given data $\boldsymbol{X}^{\mathcal{C}}$, $\boldsymbol{D}^{\mathcal{C}}$, $\boldsymbol{X}^{\mathcal{R}}$, and $\boldsymbol{X}^{\mathcal{A}}$, the final updated pairwise distance matrix is calculated by a) apply (4.5) to remove $\boldsymbol{X}^{\mathcal{R}}$; b) let $\boldsymbol{D}^{\mathcal{C}} = \boldsymbol{D}'$ and $\boldsymbol{X}^{\mathcal{C}} = \boldsymbol{X}^{\mathcal{C}} \setminus \boldsymbol{X}^{\mathcal{R}}$; and c) calculate $\boldsymbol{D}'$ by (4.6) to add $\boldsymbol{X}^{\mathcal{A}}$.

Having the updated $\boldsymbol{D}'$, we construct the updated graph G' according to the connecting rules described in 4.2.1.

4.3.2 Min-cut Updating

Having the updated graph G', we compare it with the initial graph G. Then we obtain the following six categories graph operations, by which G can be trans-

formed to G'

(a) a set of edges E_r have capacity C_r to be reduced (i.e., $C'(e) = C(e) - C_r(e), \quad \forall e \in E_r$);

(b) a set of edges E_d to be deleted (i.e., $e \in E \quad e \notin E', \quad \forall e \in E_d$);

(c) a set of nodes V_d to be deleted (i.e.,$v \in V \quad v \notin V', \quad \forall v \in V_d$);

(d) a set of nodes V_a to be added (i.e., $v \notin V \quad v \in V', \quad \forall v \in V_a$);

(e) a set of edges E_a to be added (i.e., $e \notin E \quad e \in E', \quad \forall e \in E_a$);

(f) a set of edges E_g with each edge e capacity to be increased by $C_g(e)$ (i.e., $C'(e) = C(e) + C_g(e), \quad \forall e \in E_g$).

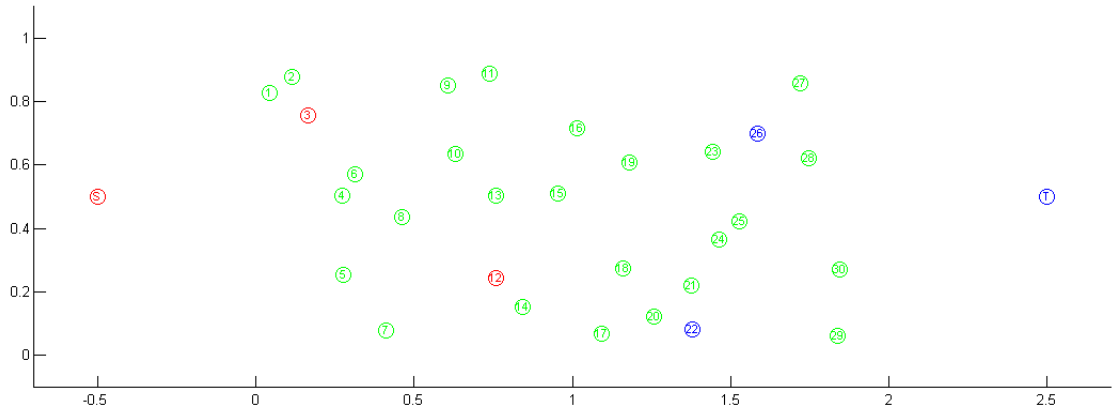
Apply our incremental max-flow algorithm proposed in Chapter **??**, we can update the max-flow on G into the max-flow on G'. The max-flow on G' is in form of a residual graph R'. To obtain the updated min-cut, we also conduct BFS on R' from node s and split R' into s-reachable S and non-reachable T. The (S, T) split is the min-cut used for classifying unlabeled samples.

4.4 Experiments and Discussions

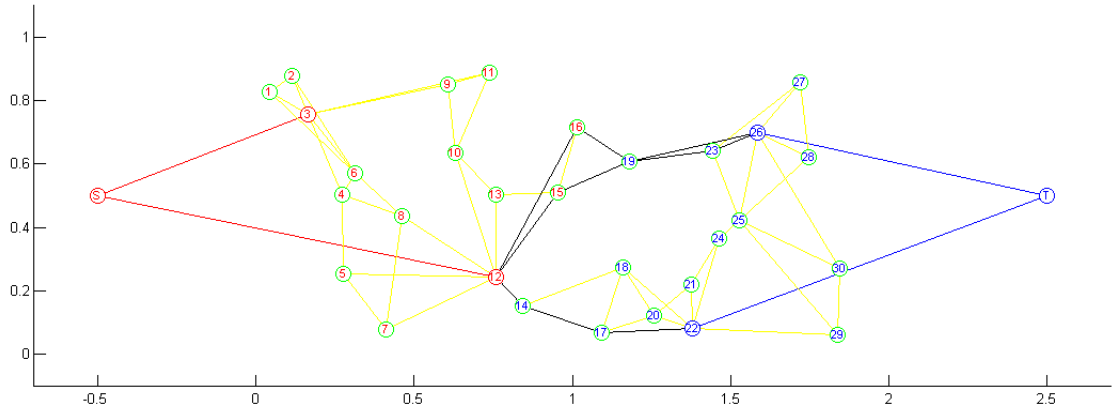
In this section, the effectiveness of our algorithms for semi-supervised learning is testified. We evaluate proposed oMincut algorithms (corresponding to batch Mincut defined in Section 4.2.1) on both artificial and real world datasets. Details regarding these datasets, and the performance has been obtained on them are presented in the following.

4.4.1 Graphical Demonstration

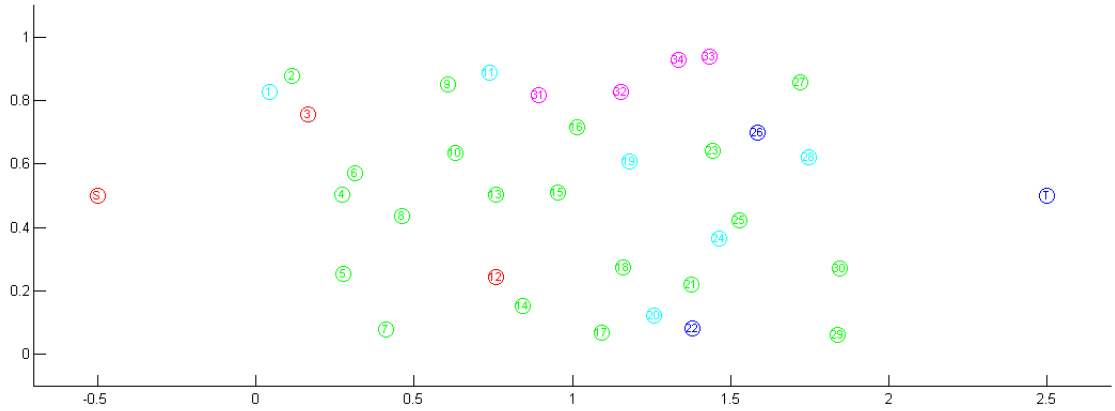
In this section, a graphical demonstration is given to show the incremental graph/model updating of proposed algorithm.



(a) Initial dataset

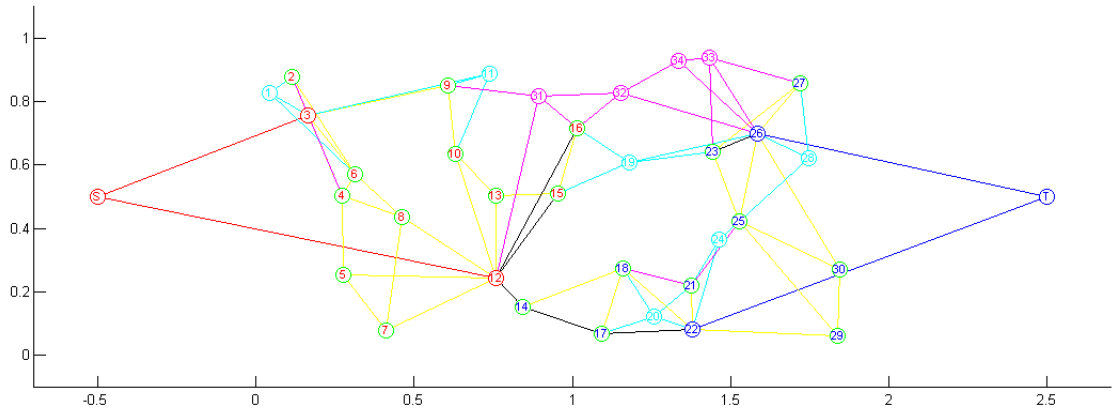


(b) Initial graph and max-flow model

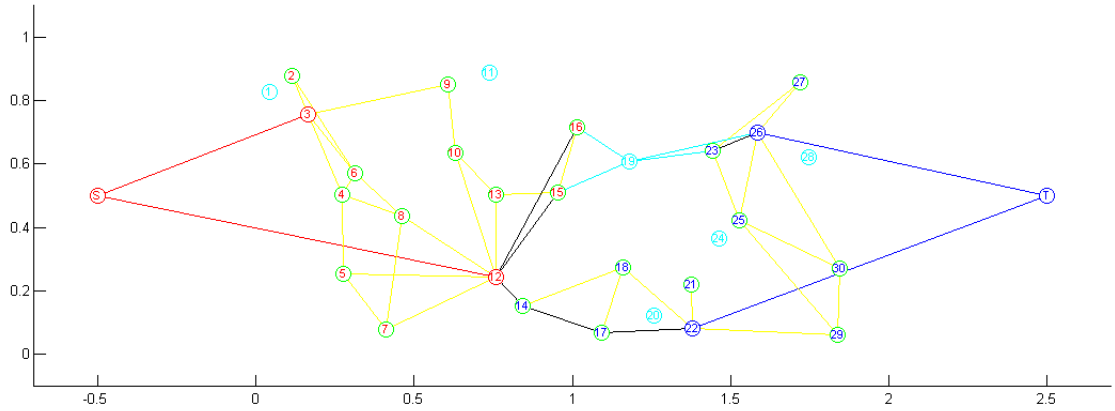


(c) Data update

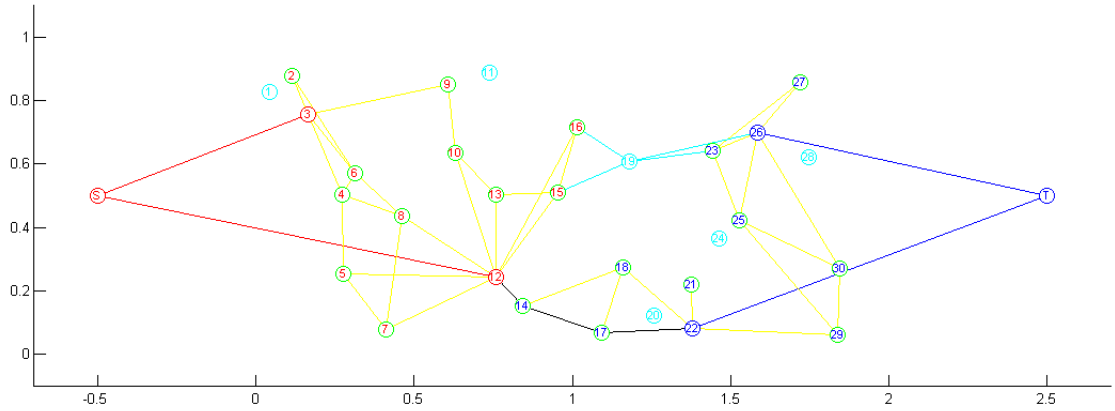
Figure 4.1: Graphical demonstration of the proposed algorithm. Part A



(a) Edges to be updated

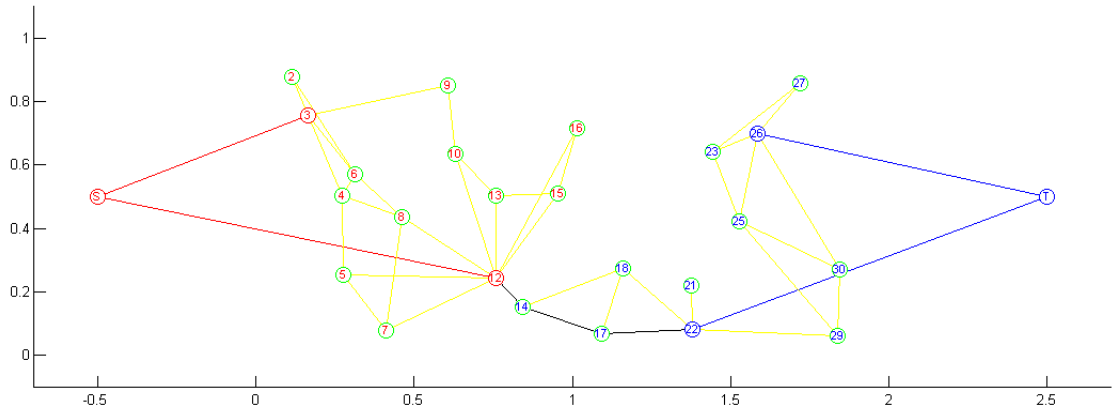


(b) Edges carry flow to be removed

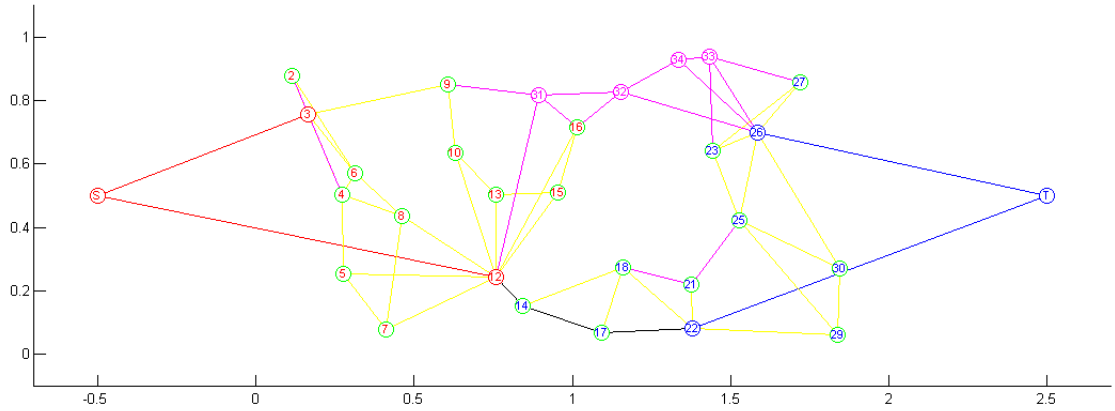


(c) Result of de-augmentation

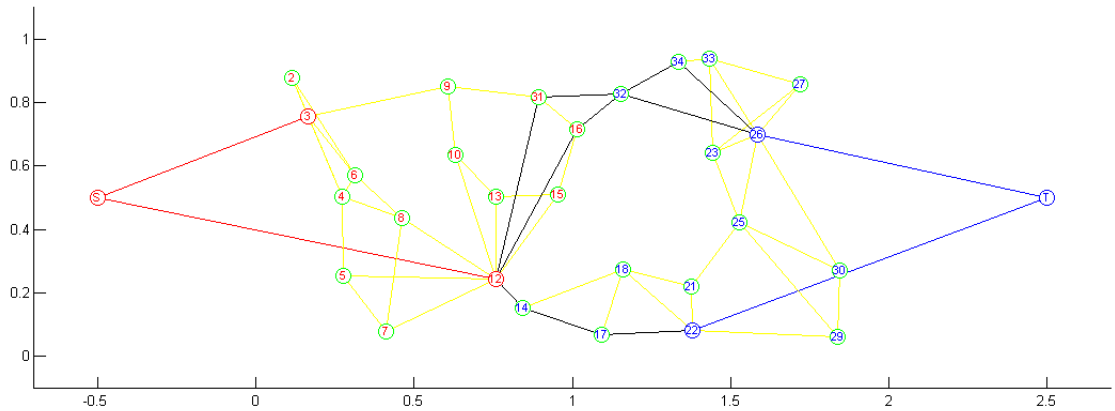
Figure 4.2: Graphical demonstration of the proposed algorithm. Part B



(a) Edge and node removal done



(b) Edge and node adding



(c) Augmenting path to find max-flow

Figure 4.3: Graphical demonstration of the proposed algorithm. Part C

Given a dataset consists of both labeled and unlabeled samples, an initial min-cut model is trained upon it. When new samples being added and old samples being retired, our algorithm conducts incremental and decremental learning, which updates the initial min-cut model to a new state.

Figure 4.1a gives an initial labeled and unlabeled dateset, where cycled points in red, blue, and green represent positive, negative, and unlabeled samples, and S and T denote the virtual source and sink nodes respectively. By a max-flow batch learning, we have Figure 4.1b as the obtained k-NN graph and the corresponding max-flow. In this figure, edges in red and blue refer to the infinite weighted edges connecting S to all positive labeled nodes and T to all negative labeled nodes respectively. The remaining edges are all 1 weighted nearest neighbor connections, in which edges in yellow carry no flow and black edges have the max-flow going through. As we can see from the figure, the max-flow value is 3, and the flow goes through: a) $S, 12, 14, 17, 22, T$, b) $S, 12, 15, 19, 23, 26, T$, and c) $S, 12, 16, 19, 26, T$. We then have the min-cut $\{(12, 14), (15, 19), (16, 19)\}$, which splits the whole graph into two isolated parts. Thus, all unlabeled nodes are naturally classified into positive and negative classes. Here, we utilize color of number to differentiate the classification of unlabeled nodes, red as positive and blue for negative respectively.

Figure 4.1c-4.3c describes how the graph is being updated, when new samples is added and/or old samples is retired. Consider a set of nodes are required to be removed and added which are drawn in light blue and purple respectively in Figure 4.1c. We construct a k-NN graph on the updated dataset, and compare the graph with the one before update (i.e., Figure 4.1b). Consequently, a set of edge update is located in Figure 4.2a, as the lines in light blue and purple, which corresponds to those edges to be removed and added respectively. For updating the min-cut, we do removal first. For those edges carry no flow, we simply remove them from the graph, since the removal of these edges causes no change on current max-flow model. The resulting graph is shown in Figure 4.2b. For those edges carry flow, such as $15 - 19$, $16 - 19$, $19 - 23$ and $19 - 26$, we de-augment path $S, 12, 15, 19, 23, 26, T$ and $S, 12, 16, 19, 26, T$ by Algorithm 9 and set edges free (i.e., edges no longer carry flow), as in Figure 4.2c, then remove all together light blue edges and nodes. Figure 4.3a gives the obtained graph from

removal. Next, we handle adding. We simply include those purple nodes and edges in Figure 4.3b, and expand them into the graph. To obtain the max-flow in this expanded graph, we iteratively augment any $s-t$ path found, and result in the final update max-flow model shown in Figure 4.3c.

4.4.2 Static Classification

In this experiment, we compare classic supervised SVM, semi-supervised SVM self-training and K Nearest Neighbor with proposed algorithms on a series of benchmark UCI datasets. Since proposed algorithms are applicable for two-class problems, several two-class data are selected, and the multiclass datasets in UCI are converted into two-class datasets by combining several classes into one. The name of the dataset used, the dimensionality of the dataset, and the number of instances from positive/negative class are summarized in the first column in Table 4.2.

For each dataset, we form our training datasets with both labelled and unlabelled data in which labeled instances are randomly selected from the original dataset, and unlabeled instances are adopt from those unused instances by hiding their label information. Here, we intend to explore the effect of label ratio on semi-supervised learning from 0.1 to 0.01 and further down to the level of 0.001. Thus, we set label ratio as 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1 and 0.2, which correspond to the column 3 to 10 in Table 4.2, respectively. In our experiments, both labeled and unlabeled data are employed to train semi-supervised learners, and only labeled data are used to train supervised learners. Each learner is tested by its performance on predicting the labels of all unlabeled instances (i.e., testing dataset). For each label ratio, 25 rounds independent tests are taken.

For performance evaultation, the mean accuracy and the standard deviation are caculated and shown in form of $mean \pm std$. Hypothesis test is also performed, where p-value is computed for each algorithm with respect to the algorithm that gives the highest mean accuracy. In Table 4.2, we categorize all p-values into $0.05 > p > 0.01$, $0.01 > p > 0.001$ and $0.001 > p$, and present as *, ** and *** respectively.

As seen from the table, oMincut learners outperform others on 4 out of 6 datasets at the level of 0.1 label ratio, which indicates that proposed learners are

Dataset	Algorithm	Ratio of Label							
		0.001	0.002	0.005	0.01	0.02	0.05	0.1	0.2
BankNote	oMincut-3	–	74.49 ± 8.85 ***	80.72 ± 7.99 ***	88.85 ± 4.98 ***	**94.64** ± 2.25 –	**97.66** ± 1.27 –	**99.06** ± 0.58 –	**99.61** ± 0.38 –
	oMincut-δ_0	–	93.24 ± 0.34 **	91.74 ± 7.14 *	**92.05** ± 3.92 –	92.08 ± 4.75 *	94.90 ± 2.42 ***	94.67 ± 2.21 ***	96.69 ± 1.41 ***
4 Features	oMincut-$\delta_{1/2}$	–	**93.65** ± 0.77 –	**92.04** ± 7.26 –	92.05 ± 4.91 ***	92.08 ± 4.75 *	94.90 ± 2.42 ***	94.66 ± 2.18 ***	96.69 ± 1.41 ***
610 PosInst	SVM	–	73.28 ± 8.24 ***	76.84 ± 9.23 ***	81.76 ± 6.99 ***	89.11 ± 3.11 ***	95.89 ± 1.52 ***	97.51 ± 0.47 ***	97.81 ± 0.52 ***
762 NegInst	SVM-self	–	73.32 ± 9.18 ***	76.67 ± 10.00 ***	81.41 ± 7.89 ***	89.89 ± 3.95 ***	96.67 ± 1.61 *	97.88 ± 0.54 ***	97.99 ± 0.61 ***
	K-NN	–	67.46 ± 7.90 ***	73.40 ± 9.47 ***	84.77 ± 6.47 ***	91.58 ± 2.56 ***	96.62 ± 1.28 ***	98.58 ± 0.60 ***	99.33 ± 0.52 **
CNAE9	oMincut-3	–	–	–	**93.37** ± 6.03 –	**95.13** ± 2.38 –	**96.54** ± 1.48 –	**97.35** ± 1.44 –	**98.22** ± 0.62 –
	oMincut-δ_0	–	–	–	87.91 ± 7.07 *	90.05 ± 0.99 ***	90.68 ± 0.68 ***	90.90 ± 0.67 ***	91.55 ± 0.31 ***
856 Features	oMincut-$\delta_{1/2}$	–	–	–	88.03 ± 6.75 *	90.05 ± 0.99 ***	90.68 ± 0.68 ***	90.90 ± 0.67 ***	91.55 ± 0.31 ***
120 PosInst	SVM	–	–	–	89.01 ± 0.10 **	89.01 ± 0.11 ***	89.09 ± 0.18 ***	89.54 ± 0.41 ***	90.55 ± 0.32 ***
960 NegInst	SVM-self	–	–	–	89.00 ± 0.09 **	89.01 ± 0.11 ***	89.08 ± 0.18 ***	89.54 ± 0.43 ***	90.55 ± 0.33 ***
	K-NN	–	–	–	92.10 ± 2.61	93.49 ± 2.70 **	94.45 ± 1.71 ***	95.63 ± 1.41 ***	97.39 ± 0.74 ***
InternetAdvS	oMincut-3	–	**87.64** ± 1.66 –	**87.78** ± 1.72 –	**87.92** ± 1.74 –	**87.92** ± 2.92 –	**90.75** ± 1.91 –	**91.60** ± 2.38 –	**93.58** ± 1.27 –
	oMincut-δ_0	–	84.65 ± 6.27 *	85.54 ± 2.87 ***	86.44 ± 0.54 ***	86.12 ± 0.52 **	86.57 ± 0.49 ***	86.78 ± 0.40 ***	87.09 ± 0.27 ***
1558 Features	oMincut-$\delta_{1/2}$	–	84.39 ± 4.59 **	85.17 ± 2.86 ***	86.30 ± 0.46 ***	86.12 ± 0.52 **	86.57 ± 0.49 ***	86.78 ± 0.40 ***	87.09 ± 0.27 ***
459 PosInst	SVM	–	86.02 ± 0.03 ***	86.08 ± 0.07 ***	86.09 ± 0.07 ***	86.17 ± 0.08 **	86.37 ± 0.15 ***	86.72 ± 0.18 ***	87.22 ± 0.17 ***
2820 NegInst	SVM-self	–	86.02 ± 0.03 ***	86.08 ± 0.07 ***	86.09 ± 0.07 ***	86.17 ± 0.08 **	86.37 ± 0.15 ***	86.71 ± 0.18 ***	87.22 ± 0.17 ***
	K-NN	–	86.00 ± 0.00 ***	86.59 ± 0.73 **	86.79 ± 0.90 **	87.42 ± 0.99	89.40 ± 0.58 **	91.16 ± 0.66	93.35 ± 0.58
ionosphere	oMincut-3	–	–	–	66.06 ± 7.01 ***	71.95 ± 7.04 **	75.10 ± 9.79 **	79.25 ± 7.73 ***	82.91 ± 6.23 ***
	oMincut-δ_0	–	–	–	79.94 ± 12.92 **	80.10 ± 10.67 **	81.22 ± 10.48	74.15 ± 13.80 ***	69.06 ± 9.65 ***
34 Features	oMincut-$\delta_{1/2}$	–	–	–	**83.62** ± 13.38 –	**82.82** ± 11.61 –	81.04 ± 10.28	74.22 ± 13.86 ***	69.00 ± 9.51 ***
225 PosInst	SVM	–	–	–	66.34 ± 3.69 ***	69.90 ± 7.52 ***	83.49 ± 7.80	**89.64** ± 3.79 –	**92.30** ± 1.49 –
126 NegInst	SVM-self	–	–	–	66.38 ± 3.65 ***	69.90 ± 7.50 ***	**83.60** ± 7.77 –	89.55 ± 3.87	92.30 ± 1.52
	K-NN	–	–	–	65.31 ± 5.57 ***	68.29 ± 5.67 ***	72.05 ± 5.86 ***	76.97 ± 6.69 ***	81.91 ± 3.76 ***
Musk	oMincut-3	–	–	53.23 ± 5.53 ***	54.10 ± 5.74 ***	57.00 ± 4.00 ***	63.80 ± 4.48 *	**68.48** ± 4.07 –	**75.47** ± 2.91 –
	oMincut-δ_0	–	–	**60.68** ± 5.11 –	64.56 ± 4.64	63.66 ± 4.36	**66.38** ± 3.65 –	65.33 ± 5.30 *	65.36 ± 4.57 ***
166 Features	oMincut-$\delta_{1/2}$	–	–	59.83 ± 6.91	**64.74** ± 4.35 –	**63.93** ± 4.39 –	66.38 ± 3.65 ***	65.33 ± 5.30 *	65.36 ± 4.57 ***
207 PosInst	SVM	–	–	54.90 ± 6.09 **	58.00 ± 8.38 **	59.34 ± 1.59 ***	61.06 ± 1.16 ***	65.53 ± 1.69 **	71.95 ± 2.30 ***
269 NegInst	SVM-self	–	–	54.88 ± 6.05 **	57.91 ± 8.31 **	59.14 ± 1.56 ***	60.80 ± 1.12 ***	65.35 ± 1.75 **	71.71 ± 2.32 ***
	K-NN	–	–	49.90 ± 4.02 ***	49.91 ± 4.20 ***	55.44 ± 3.75 ***	59.44 ± 4.25 ***	65.50 ± 2.79 **	72.46 ± 2.00 ***
waveform	oMincut-3	**72.26** ± 4.01 –	**77.02** ± 4.58 –	**80.60** ± 2.09 –	81.22 ± 1.92 ***	82.65 ± 1.18 ***	83.63 ± 0.69 ***	84.33 ± 0.51 ***	84.24 ± 0.69 ***
	oMincut-δ_0	64.48 ± 4.79 ***	66.76 ± 1.10 ***	66.29 ± 1.32 ***	66.62 ± 0.71 ***	66.97 ± 0.34 ***	66.99 ± 0.17 ***	66.94 ± 0.09 ***	66.97 ± 0.10 ***
21 Features	oMincut-$\delta_{1/2}$	64.17 ± 4.73 ***	66.76 ± 1.10 ***	66.29 ± 1.32 ***	66.62 ± 0.71 ***	66.97 ± 0.34 ***	66.99 ± 0.17 ***	66.94 ± 0.09 ***	66.97 ± 0.10 ***
1657 PosInst	SVM	67.27 ± 1.37 ***	71.94 ± 5.09 ***	79.68 ± 3.82	84.37 ± 2.59	87.20 ± 1.17 *	88.31 ± 0.58	88.99 ± 0.30 *	89.45 ± 0.36 ***
3343 NegInst	SVM-self	67.19 ± 1.24 ***	71.70 ± 5.21 ***	79.62 ± 4.03	**84.41** ± 2.66 –	**87.28** ± 1.13 –	**88.36** ± 0.65 –	**89.05** ± 0.33 –	**89.58** ± 0.39 –
	K-NN	70.41 ± 5.13 *	74.66 ± 4.77 *	80.02 ± 2.24	81.44 ± 1.34 ***	82.10 ± 1.50 ***	83.51 ± 0.79 ***	84.42 ± 0.71 ***	84.84 ± 0.58 ***

Table 4.2: Accuracy comparison on UCI datasets

competitive to benchmark learners if no less than 10% data are labeled. When the label ratio is 10 times smaller, proposed learners achieve the best performance for 5 out of 6 datasets. It is worth noting that at the lowest level of 0.001, proposed learners are giving consistently the highest prediction accuracy for all applicable datasets. This follows that given 0.1% of data labelled, proposed leaner is still giving the best performance. Further, the oMincut-3's superiority on class-imbalanced classification is demonstrated in Table 4.3 in which recall is evaluated on each of the binary classes.

4.4.3 Drifting Concept Tracing

In this experiment, we demonstrate how proposed algorithm handles drifting concepts while incremental and decremental learning in Figure 4.4.

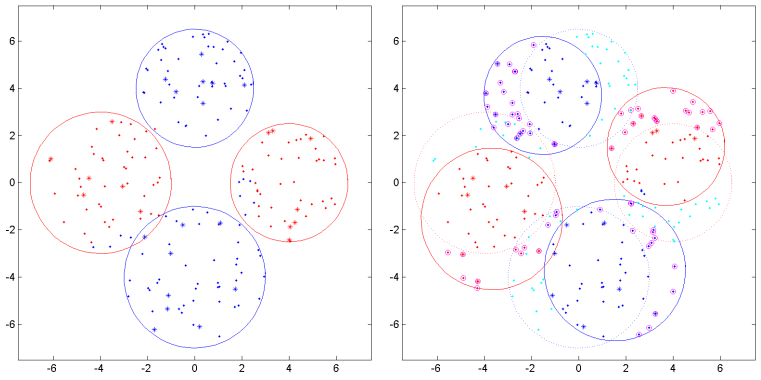
Figure 4.4a shows the initial learning stage. In a two-dimensional feature space, there are two opposite concepts (i.e., data distribution) identified in blue and red colors respectively. Each concept consists of two sub-concepts with their data distribution bounded by circles. In this figure, dots and stars represent labeled and unlabeled samples respectively; and the color of symbols shows the class label of labeled samples or predicted label of unlabeled sample.

Let two concepts start drifting by rotating all four sub-concepts against $(0, 0)$ for 5 stages. For each step rotation, we keep constant the scale of each sub-concept (i.e., the radius) and its distance to to $(0, 0)$, then turn all circles around $(0, 0)$ for 18 degrees. Consequently, all sub-concepts rotate 90 degrees, which follows that the two concepts exchange their positions. Figure 4.4b-4.4e show the procedure of concept drifting in which solid circles represent current and dashed ones represent the past sub-concepts. As the result of concept drifting, we thus form a dynamic data stream, whose samples that are no longer in the scope of current concept are being removed; and new samples fall into current concept are being added in.

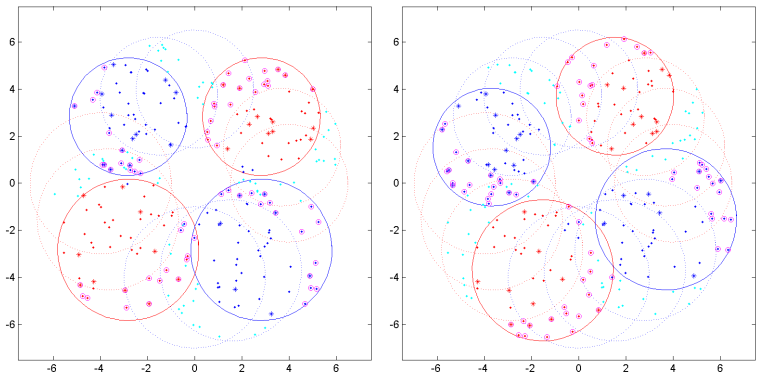
Through five stages concept drift, we trace the error rate of oMincut at each stage, and compare the final stage rate with that of batch learning. As seen from the parentheses below the plots, the transductive error rates of oMincut are smaller than 7% for all five stages, and the final stage rate is exactly the same as the rate from the batch learning.

Dataset	Class	Algorithm	Ratio of Label 0.002	0.005	0.01	0.02	0.05	0.1	0.2
CNAE9	Pos	oMincut-3	–	–	**52.10** ± 26.71 –	**56.38** ± 21.60 –	**69.26** ± 13.71 –	**77.22** ± 13.16 –	**84.63** ± 5.88 –
		oMincut-δ_0	–	–	10.27 ± 11.72 ***	10.09 ± 8.99 ***	16.14 ± 6.14 ***	18.07 ± 6.02 ***	23.96 ± 2.77 ***
		oMincut-$\delta_{1/2}$	–	–	10.14 ± 11.45 ***	10.09 ± 8.99 ***	16.14 ± 6.14 ***	18.07 ± 6.02 ***	23.96 ± 2.77 ***
		SVM	–	–	0.58 ± 0.87 ***	0.68 ± 0.96 ***	1.79 ± 1.65 ***	5.89 ± 3.72 ***	14.96 ± 2.91 ***
		SVM-self	–	–	0.47 ± 0.85 ***	0.68 ± 0.96 ***	1.68 ± 1.64 ***	5.85 ± 3.84 ***	14.96 ± 3.01 ***
		K-NN	–	–	28.51 ± 23.74 ***	41.50 ± 24.75 **	50.28 ± 15.53 ***	61.74 ± 13.12 ***	77.75 ± 7.28 ***
	Neg	oMincut-3	–	–	98.49 ± 7.27	99.95 ± 0.12	99.95 ± 0.11 *	99.87 ± 0.17 ***	99.92 ± 0.11 **
		oMincut-δ_0	–	–	97.56 ± 8.94	**100.00** ± 0.00 –	**100.00** ± 0.00 –	**100.00** ± 0.00 –	**100.00** ± 0.00 –
		oMincut-$\delta_{1/2}$	–	–	97.71 ± 8.53	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
		SVM	–	–	**100.00** ± 0.00 –	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
		SVM-self	–	–	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
		K-NN	–	–	100.00 ± 0.02	99.96 ± 0.06 **	99.97 ± 0.09	99.87 ± 0.19 **	99.84 ± 0.16 ***
InternetAdvS	Pos	oMincut-3	**14.75** ± 8.67 –	**15.76** ± 9.05 –	**16.95** ± 9.41 –	**26.00** ± 8.50 –	**43.84** ± 7.60 –	**53.75** ± 4.71 –	**62.53** ± 3.97 –
		oMincut-δ_0	3.55 ± 5.73 ***	2.85 ± 3.47 ***	3.47 ± 4.59 ***	2.33 ± 1.32 ***	4.78 ± 3.03 ***	6.26 ± 2.71 ***	8.48 ± 1.94 ***
		oMincut-$\delta_{1/2}$	14.05 ± 20.64	5.82 ± 13.08 **	4.51 ± 9.45 ***	2.33 ± 1.32 ***	4.78 ± 3.03 ***	6.26 ± 2.71 ***	8.48 ± 1.94 ***
		SVM	0.11 ± 0.20 ***	0.46 ± 0.52 ***	0.60 ± 0.46 ***	1.10 ± 0.58 ***	2.72 ± 1.07 ***	5.31 ± 1.29 ***	9.01 ± 1.26 ***
		SVM-self	0.12 ± 0.23 ***	0.48 ± 0.54 ***	0.61 ± 0.47 ***	1.08 ± 0.62 ***	2.72 ± 1.10 ***	5.28 ± 1.30 ***	9.05 ± 1.28 ***
		K-NN	0.00 ± 0.00 ***	4.41 ± 5.26 ***	5.85 ± 6.64 ***	10.34 ± 7.27 ***	25.18 ± 4.15 ***	38.65 ± 5.09 ***	54.15 ± 4.42 ***
	Neg	oMincut-3	99.49 ± 1.15 *	99.50 ± 1.15 *	99.46 ± 1.16 *	97.98 ± 3.45 **	98.39 ± 2.40 **	97.76 ± 2.78 ***	98.63 ± 1.65 ***
		oMincut-δ_0	97.85 ± 7.77	98.98 ± 3.31	99.94 ± 0.18	99.73 ± 0.55 *	99.88 ± 0.25	99.88 ± 0.12 **	99.87 ± 0.12 **
		oMincut-$\delta_{1/2}$	95.84 ± 6.38 **	98.07 ± 4.36 *	99.60 ± 1.84	99.73 ± 0.55 *	99.88 ± 0.25	99.88 ± 0.12 **	99.87 ± 0.12 **
		SVM	100.00 ± 0.01	99.99 ± 0.02	**100.00** ± 0.01 –	99.99 ± 0.02	**99.98** ± 0.02 –	**99.97** ± 0.02 –	**99.94** ± 0.05 –
		SVM-self	100.00 ± 0.01	**100.00** ± 0.01 –	100.00 ± 0.01	**99.99** ± 0.02 –	99.98 ± 0.02	99.97 ± 0.02	99.94 ± 0.05
		K-NN	**100.00** ± 0.00 –	99.95 ± 0.13	99.96 ± 0.10 *	99.95 ± 0.06 **	99.85 ± 0.27 *	99.70 ± 0.31 ***	99.73 ± 0.22 ***

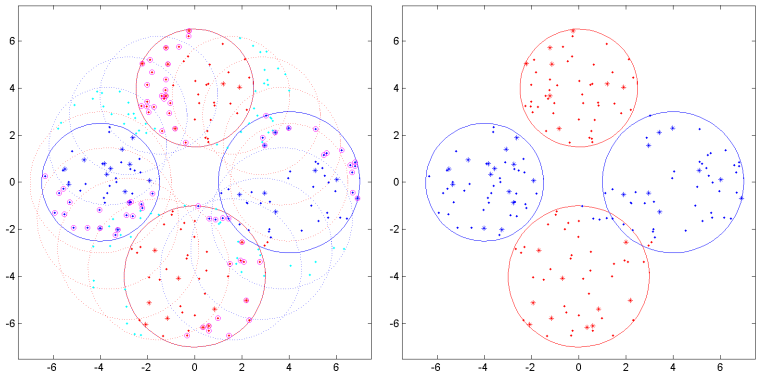
Table 4.3: Two-class recall comparison on imbalanced datasets



(a) Stage1 (b) Stage2



(c) Stage3 (d) Stage4



(e) Stage5 (f) Batch Learning on Stage 5

Figure 4.4: oMincut incremental and decremental learning on five stages concept drift, with the final stage compared to batch learning. The transductive performance in terms of classification error rate is given at each stage in parentheses as (a) (6.43 percent), (b) (3.59 percent), (c) (4.46 percent), (d) (3.20 percent), (e) (4.40 percent) and (f) (4.40 percent).

4.4.4 Stream Learning

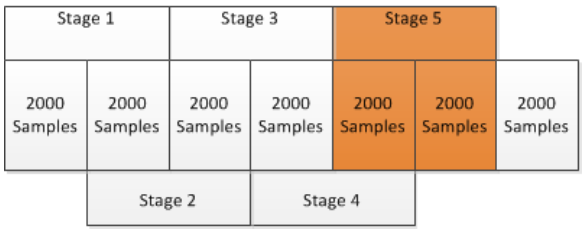
In this section, we compare proposed algorithms with SVM, SVM self-training and K-NN on a real-world data stream learning. The data used here is the KDD99 Intrusion Detection stream, which consists of 122 features and over 125k samples. Each sample corresponds to a TCP connection, which is either a normal connection or an attack. In this experiment, two incremental learning scenarios are adopted :

a) Sliding Window Snapshot. Consider learning from a data stream with concept drifts, we let algorithms learn at each stage from only a segment of data stream bounded by a sliding window. Thus, all learners retain at all time an up-to-date view of the drifting concepts. Figure 4.5a illustrates the setup of this stream learning, where the size of sliding window is 4000 and the length of sliding step is 2000.

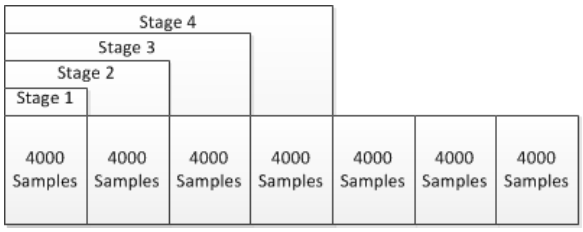
b) Data Accumulation. Assume the class concepts are constant for the entire data stream, we conduct only incremental learning at each stage to reinforce learning effectiveness by accommodating new chunk of data. The training data is fed as in Figure 4.5b, where we set the chunk size as 4000.

For each scenario, we conduct experiments with the same label ratios as the ones used in Section 4.4.2; and we use the testing accuracy as performance measurement.

For sliding window snapshot scenario, Figure 4.6 compares all learners performance under the label ratio of 0.001, 0.01 and 0.1, respectively. As seen from the figure, the superiority of proposed oMincut-3 is observed across all three label ratios. When the ratio is 0.1, oMincut-3 has visible superiority at 16 out of total 40 stages. Such superiority become apparent in the case of lower ratio 0.01. The absolute superiority occurs when the ratio is further 10 times lower. Apparently, all learners perform sensitive to the label ratio. Figure 4.7 shows the performance (in terms of accuracy mean and standard deviation) variation of learners against the label ratio. We can see that with the decrease of label ratio, all learners lose classification capability performing with lower accuracy meanwhile higher variance (indicating system stability reduced). However, proposed oMincut-3 gives



(a) Sliding Window Snapshot



(b) Data Accumulation

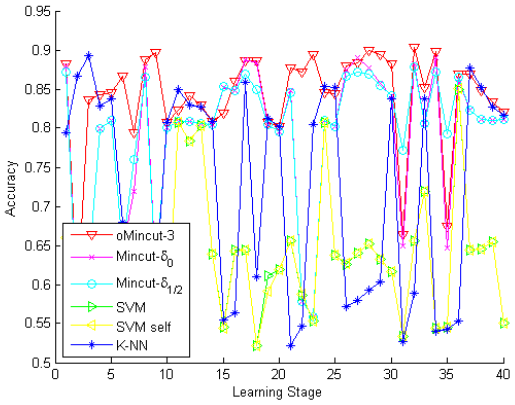
Figure 4.5: Two stream learning scenarios

the slowest performance reduction in terms of both accuracy and stability, when the label ratio decreases.

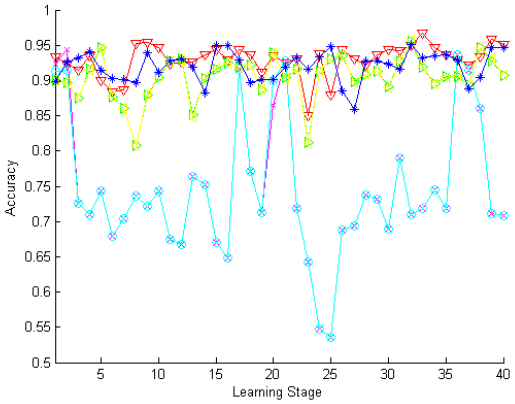
In the mode of data accumulation learning, learners are trained incrementally from an accumulating dataset. We measure learner performance as stage mean accuracy and final stage accuracy. Figure 4.8 compares the performance of learners at different label ratios. Similar to the result from the above snapshot learning, proposed oMincut-3 gives the lowest performance reduction to the decrease of label ratio.

4.5 Summary

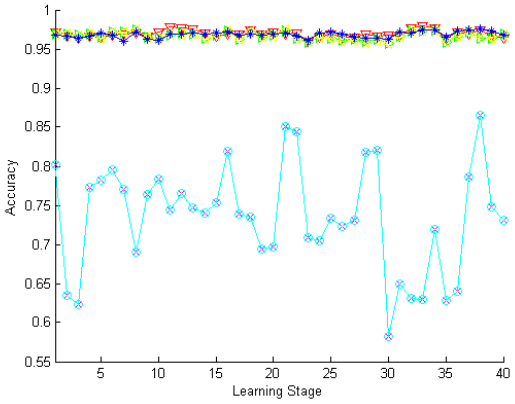
For learning from data streams with both labeled and unlabeled samples, an incremental decremental max-flow based incremental semi-supervised learning system is proposed. The basic assumption for this work is that samples with smaller distance are more likely to be in the same class. To conduct semi-supervised learning, a K-NN based sample network is built based on both labeled and unlabeled data to describe the "close to" relationship between samples. Then a min-cut is applied to split the whole set into two classes by removing the mini-



(a) R=.001

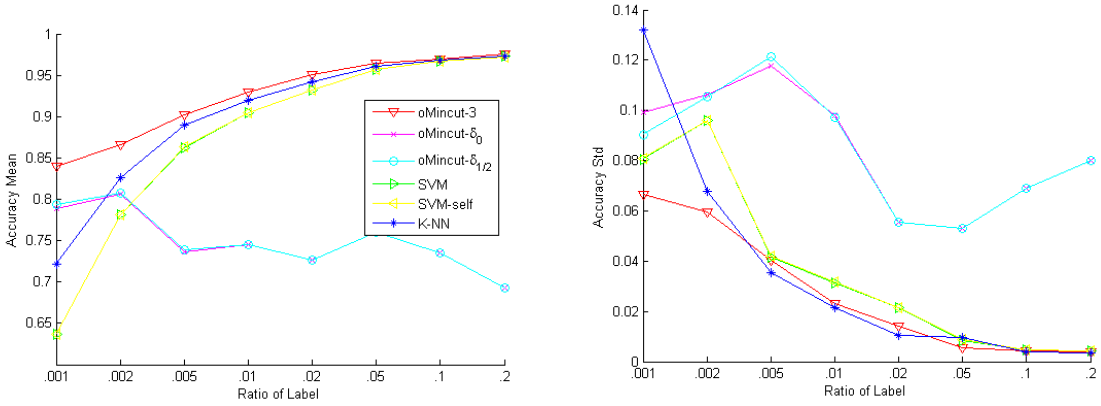


(b) R=.01



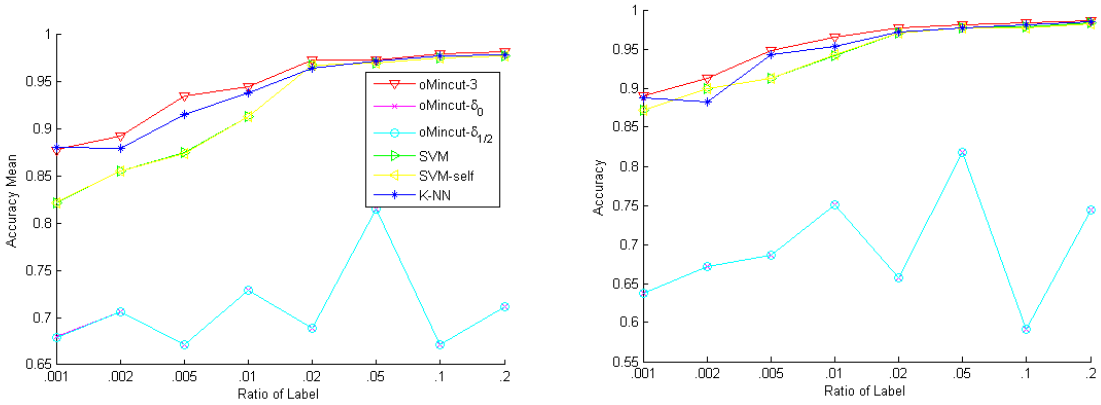
(c) R=.1

Figure 4.6: Sliding window snapshot learning on KDD streams



(a) Accuracy Mean (b) Accuracy Standard Deviation

Figure 4.7: The variation of learners performance against the label ratio.



(a) Stage Mean Accuracy (b) Final Stage Accuracy

Figure 4.8: The variation of learners performance against the label ratio.

mal number of "close to" relation. To find such min-cut, the max-flow problem is required to be solved on the sample network.

In this work, we derive incremental max-flow for semi-supervised learning by proposing 1) an incremental sample network whose pair-wised sample distance matrix is being updated for every sample adding/retiring, and more importantly 2) our incremental/decremental max-flow algorithm is applied to update the max-flow whenever the sample network changes. Proposed incremental semi-supervised learning system is demonstrated capable of accommodating new samples adding and old samples retiring, with a theoretical guarantee that incremental learning result equals always that of batch retraining. Experiments on UCI benchmark datasets and KDD data stream show that proposed system is less sensitive to the amount of labelled data (in terms of the ratio to the whole training data) as compared to K-NN, SVM and SVM self-training. Actually, with only 0.1% training data labelled, our algorithm still be able to achieve relatively high accuracy.

The proposed algorithm is able to retire unwanted samples, but in real-world application it is rare to have the priori knowledge of which set of data is no longer needed [6] [69]. Thus, incorporating concept drift detection mechanism is an interesting work.

5 Parallel Incremental Learning Integration

5.1 Introduction

In our study of incremental max-flow and graph mincuts, we found that the training speed is not in good satisfactory when data is huge. A straightforward solution is to combine parallel data processing with incremental learning.

We attempt to parallelize incremental max-flow, and find the difficulty lies at 1) For the augmenting path mechanism, the augmentation and de-augmentation can be done in parallel, if we can find a set of paths that are edge disjoint. However, there is no existing solutions for parallelizing the search for such paths. Moreover, such edge disjoint paths may even not exists, such as in a graph with bottleneck edge where most flows go through that bottleneck. 2) For the push relabel mechanism, the push and relabel operation can be parallelized if we can identify a set of active nodes that are neighbor disjoint. But it is computationally very expensive for such identification. Similarly, in case of graph with bottleneck, such neighbor disjoint active nodes can not be found in certain stage. All in all, we are not able to merge max-flow knowledge from sub-graphs.

5.2 Motivation: PI Integration via Knowledge Merging

For developing parallel incremental learning algorithm, three straightforward route maps are considered: a) parallelize an existing incremental algorithm, for example [77] recently proposed recently a gossip-based approach that parallelizes

online prediction and stochastic optimization; b) renovate an existing parallel algorithm to be incremental, but few research study can be found in this category; and c) derive a parallel incremental algorithm from scratch. In this category, existing works normally treat incremental and parallel learning as two separate problems. One either first parallelizes learning then designs the incremental rule (e.g., [78], [79], [80]), or the other way around (e.g., [81], [82], [83], [84]). Alternatively, we intend to solve these two learning problems in one process.

Definition 5.2.1. *Knowledge mergeable condition: Given dataset D_a and D_b such that $D = D_a \cup D_b$, a learning model T is knowledge mergeable if $K_D = M(K_{D_a}, K_{D_b})$, where $M()$ is the merging rule and $K_D = T(D)$ is the learning knowledge from D.*

Based on a knowledge mergeable learning model, the incremental learning of the model can be implemented as updating current model through continuously adding new knowledge from incoming data. Similarly, parallel learning can be seen as parallel knowledge extraction on data slices followed by adding up knowledge from all data slices. In this sense, parallel and incremental learning can be unified by a knowledge merging process running over temporal and spatial domains. Therefore, we have

Definition 5.2.2. *PI integration: If base model T satisfies the knowledge mergeable condition; the parallel learning of T, $P_T(D) = M(K_{D_1}, \ldots, K_{D_n})$ where $D = D_1 \cup \ldots \cup D_n$; and the incremental learning of T, $I_T(D') = M(K_D, K_{D'})$ where D' is the data newly arrived, then the parallel learning and incremental learning of T forms a PI integration, where $M()$ is the common core operation.*

by which, parallel learning and incremental learning of T are encapsulated into one PI integrated system. In the remaining chapter, we review a family of algorithms that satisfy the knowledge mergeable condition.

5.3 Knowledge Mergeable Algorithms

5.3.1 LPSVM

Given a training set $\boldsymbol{\mathcal{S}} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$, having instance matrix $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \end{bmatrix}' \in \mathbb{R}^{n \times d}$ and label vector $\boldsymbol{Y} = \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix}' \in \{+1, -1\}^{n \times 1}$.

A classic Support Vector Machine (SVM) [85] solves the following optimization

$$\begin{aligned} \min \quad & C\sum_{i=1}^{n}\xi_i + \frac{\|\boldsymbol{w}\|^2}{2} \\ \text{s.t.} \quad & y_i(\boldsymbol{x}_i'\boldsymbol{w}+b)+\xi_i \geq 1 \\ & \xi_i \geq 0 \quad \forall i \in \{1,\cdots,n\}, \end{aligned} \tag{5.1}$$

in order to learn a separation plane

$$\boldsymbol{x}'\boldsymbol{w}+b=0, \tag{5.2}$$

which located in midway of two bounding planes

$$\begin{aligned} \boldsymbol{x}'\boldsymbol{w}+b&=+1 \\ \boldsymbol{x}'\boldsymbol{w}+b&=-1. \end{aligned} \tag{5.3}$$

Training samples from two classes are bounded by (5.3) with some non-negative slacks ξ_i

$$\begin{aligned} \boldsymbol{x}_i'\boldsymbol{w}+b+\xi_i &\geq +1 \quad \text{for} \quad y_i=+1 \\ \boldsymbol{x}_i'\boldsymbol{w}+b-\xi_i &\leq -1 \quad \text{for} \quad y_i=-1. \end{aligned} \tag{5.4}$$

$\frac{2}{\|\boldsymbol{w}\|}$ is the distance between bounding planes in (5.3), also know as margin in literature [86]. In optimization (5.1), the margin is maximized by minimizing $\frac{\|\boldsymbol{w}\|^2}{2}$, the total slacks is minimized by minimizing $\sum_{i=1}^{n}\xi_i$, and the importance of margin maximization and total slacks minimization is balanced by parameter C [87, 88].

Unlike classic SVM [85], Linear Proximal SVM (LPSVM) simplifies above binary classification as an regularized least square problem, thus the training of LPSVM becomes more efficient [89] than the classic SVM. Specifically, LPSVM solves the following optimization

$$\begin{aligned} \min \quad & \frac{1}{2}(\|\boldsymbol{w}\|^2+b^2)+\frac{C}{2}\|\boldsymbol{\xi}\|^2 \\ \text{s.t.} \quad & \boldsymbol{D}(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{e}b)+\boldsymbol{\xi}=\boldsymbol{e}, \end{aligned} \tag{5.5}$$

where $\boldsymbol{\xi}$ is a $n\times 1$ slack vector, $n\times n$ diagonal matrix $\boldsymbol{D}=diag(\boldsymbol{Y})$ represents class labels, and $\boldsymbol{e}$ is a $n\times 1$ vector of ones. Through solving (5.5), LPSVM obtains a separating plane

$$\boldsymbol{x}'\boldsymbol{w}-b=0 \tag{5.6}$$

which lies in the middle of two proximal planes

$$\begin{aligned} \boldsymbol{x}'\boldsymbol{w} - b &= +1 \\ \boldsymbol{x}'\boldsymbol{w} - b &= -1. \end{aligned} \tag{5.7}$$

As compared to the classic SVM optimization (5.1) which has inequality constraint, LPSVM applies in (5.5) an equality constraint. As a result, the planes (5.7) no longer bound training samples, but become the proximal planes with data points of each class clustered around. In LPSVM optimization (5.5), margin between proximal planes are maximized by minimizing term $(\|\boldsymbol{w}\|^2 + b^2)$, the total slack is minimized by minimizing $\|\boldsymbol{\xi}\|^2$, and the importance of these two objectives are balanced by parameter C.

The solution of LPSVM can be given explicitly as

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = \left[\frac{\boldsymbol{I}}{C} + \begin{bmatrix} \boldsymbol{X}' \\ -\boldsymbol{e}' \end{bmatrix} \begin{bmatrix} \boldsymbol{X} & -\boldsymbol{e} \end{bmatrix} \right]^{-1} \begin{bmatrix} \boldsymbol{X}'\boldsymbol{D}\boldsymbol{e} \\ -\boldsymbol{e}'\boldsymbol{D}\boldsymbol{e} \end{bmatrix}. \tag{5.8}$$

For the derivation of (5.8), please refer to the work of [89] Let $\boldsymbol{F} = \begin{bmatrix} \boldsymbol{X} & -\boldsymbol{e} \end{bmatrix}$, we have (5.8) simplified as

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = (\frac{\boldsymbol{I}}{C} + \boldsymbol{F}'\boldsymbol{F})^{-1}\boldsymbol{F}'\boldsymbol{D}\boldsymbol{e}, \tag{5.9}$$

where $\boldsymbol{I}$ is a $(d+1) \times (d+1)$ identity matrix.

The steps of batch LPSVM training is summarized in Algorithm 13.

Once $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$ is computed, classification decision is made by

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{x}'\boldsymbol{w} - b \\ &= \begin{bmatrix} \boldsymbol{x}' & -1 \end{bmatrix} \begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} \begin{cases} > 0 & \text{then} \quad y = +1 \\ < 0 & \text{then} \quad y = -1. \end{cases} \end{aligned} \tag{5.10}$$

5.3.2 ESVM

Extreme SVM (ESVM) [90] is essentially an LPSVM in Extreme Learning Machine (ELM) [91,92] feature space. For nonlinear classification, ESVM introduces a nonlinear mapping function $\Phi(\boldsymbol{x})$, through which d dimensional input samples

Algorithm 13 LPSVM

Input: $\boldsymbol{X} \in \mathbb{R}^{n\times d}$, $\boldsymbol{Y} \in \{+1,-1\}^{n\times 1}$, $C \in \mathbb{R}^{+}$.

Output: $\begin{bmatrix}\boldsymbol{w}\\ b\end{bmatrix}$.

1: Generate $\boldsymbol{e} \in \mathbb{R}^{n\times 1}$;
2: Generate $\boldsymbol{F}$, as $\boldsymbol{F} = \begin{bmatrix}\boldsymbol{X} & -\boldsymbol{e}\end{bmatrix}$;
3: Transform $\boldsymbol{Y}$ into $\boldsymbol{D}$, as $\boldsymbol{D} = diag(\boldsymbol{Y})$;
4: Generate $\boldsymbol{I} \in \mathbb{R}^{(d+1)\times(d+1)}$;
5: Compute $\begin{bmatrix}\boldsymbol{w}\\ b\end{bmatrix}$ as (5.9).

are mapped explicitly into a $\tilde{d}$ dimensional feature space, so that an LPSVM linear separation can be conducted in feature space to achieve nonlinear classification of input space.

The mapping $\Phi(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}^{\tilde{d}}$ is performed as

$$\begin{aligned}\Phi(\boldsymbol{x}) =& G(\boldsymbol{W}\boldsymbol{x}^1)\\ =& (g(\sum_{i=1}^{d}\boldsymbol{W}_{1i}\boldsymbol{x}_i + \boldsymbol{W}_{1(d+1)}), \ldots,\\ & g(\sum_{i=1}^{d}\boldsymbol{W}_{\tilde{d}i}\boldsymbol{x}_i + \boldsymbol{W}_{\tilde{d}(d+1)}))'.\end{aligned} \tag{5.11}$$

where $\boldsymbol{x} \in \mathbb{R}^{d\times 1}$ is the sample of input space, $\boldsymbol{x}^1 = [\boldsymbol{x}', 1]'$, $\boldsymbol{W} \in \mathbb{R}^{\tilde{d}\times(d+1)}$ is a weighting matrix whose elements are randomly generated, and $\Phi(\boldsymbol{x})$ is the projection of $\boldsymbol{x}$ in $\mathbb{R}^{\tilde{d}}$. Here, $G(\cdot)$ stands for a mapping function that projects each element z_{ij} of input matrix $\boldsymbol{Z}$ into corresponding $g(z_{ij})$ of output matrix $G(\boldsymbol{Z})$, where $g(\cdot)$ is an activation function specified by user such as sigmoidal function. From the view point of ELM, $\Phi(\boldsymbol{x})$ can be seen as the output of $\boldsymbol{x}$ through the hidden layer, also $\boldsymbol{W}$ and $\boldsymbol{x}^1$ can be taken as the input weights and vector of the hidden Layer respectively.

Applying (5.11) to every instance $\boldsymbol{x_i}$ of $\boldsymbol{X}$, we have the projection of the entire instance matrix $\boldsymbol{X}$ as

$$\Phi(\boldsymbol{X}) = [\Phi(\boldsymbol{x}_1'), \ldots, \Phi(\boldsymbol{x}_n')]'. \tag{5.12}$$

Having samples mapped into feature space by (5.12), ESVM proceeds with an LPSVM linear separation on projected samples in feature space. Mathematically, the optimization of ESVM is formulated as

$$\begin{aligned} \min \quad & \frac{C}{2}\left\|\boldsymbol{\xi}\right\|^2 + \frac{1}{2}(\boldsymbol{w}'\boldsymbol{w} + b^2) \\ \text{s.t.} \quad & \boldsymbol{D}(\Phi(\boldsymbol{X})\boldsymbol{w} - \boldsymbol{e}b) + \boldsymbol{\xi} = \boldsymbol{e}. \end{aligned} \tag{5.13}$$

Solving (5.13) as in LPSVM [89], the following solution is obtained for nonlinear ESVM

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = \left[\frac{\boldsymbol{I}}{C} + \begin{bmatrix} \Phi(\boldsymbol{X})' \\ -\boldsymbol{e}' \end{bmatrix} \begin{bmatrix} \Phi(\boldsymbol{X}) & -\boldsymbol{e} \end{bmatrix} \right]^{-1} \begin{bmatrix} \Phi(\boldsymbol{X})'\boldsymbol{D}\boldsymbol{e} \\ -\boldsymbol{e}'\boldsymbol{D}\boldsymbol{e} \end{bmatrix}. \tag{5.14}$$

Let $\boldsymbol{E} = \begin{bmatrix} \Phi(\boldsymbol{X}) & -\boldsymbol{e} \end{bmatrix} \in \mathbb{R}^{n\times(\tilde{d}+1)}$, (5.14) can be simplified as

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = (\frac{\boldsymbol{I}}{C} + \boldsymbol{E}'\boldsymbol{E})^{-1}\boldsymbol{E}'\boldsymbol{D}\boldsymbol{e}. \tag{5.15}$$

The steps of nonlinear ESVM training is stated in Algorithm 14.

Algorithm 14 ESVM

Input: $\boldsymbol{X} \in \mathbb{R}^{n\times d}$, $\boldsymbol{Y} \in \{+1, -1\}^{n\times 1}$, $C \in \mathbb{R}^+$.

Output: $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$.

1: Generate $\boldsymbol{e} \in \mathbb{R}^{n\times 1}$;
2: Compute $\Phi(\boldsymbol{X})$ as (5.11) and (5.12);
3: Generate $\boldsymbol{E}$, as $\boldsymbol{E} = \begin{bmatrix} \Phi(\boldsymbol{X}) & -\boldsymbol{e} \end{bmatrix}$;
4: Transform $\boldsymbol{Y}$ into $\boldsymbol{D}$, as $\boldsymbol{D} = diag(\boldsymbol{Y})$;
5: Generate $\boldsymbol{I} \in \mathbb{R}^{(\tilde{d}+1)\times(\tilde{d}+1)}$;
6: Compute $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$ as (5.15).

For any incoming unseen sample $\boldsymbol{x}$, ESVM classification is conducted as

$$\begin{aligned} f(\boldsymbol{x}) &= \Phi(\boldsymbol{x})'\boldsymbol{w} - b \\ &= \begin{bmatrix} \Phi(\boldsymbol{x})' & -1 \end{bmatrix} \begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} \begin{cases} > 0 & \Rightarrow y = +1 \\ < 0 & \Rightarrow y = -1. \end{cases} \end{aligned} \tag{5.16}$$

5.3.3 wLPSVM and wESVM

Class imbalance harms the classification capability when the size of one class known as the minority class is much smaller than that of the other class, which is known as majority class. Recall the optimization for LPSVM (5.5), in which margin is maximized by minimizing $\|\boldsymbol{w}\|^2 + b^2$ and proximal planes is drawn respectively to the center of classes by minimizing total slacks $\|\boldsymbol{\xi}\|^2$. When class imbalance exists, the total slacks from minority (positive) class is much smaller than from majority (negative) class, i.e., $\|\boldsymbol{\xi}_+\|^2 << \|\boldsymbol{\xi}_-\|^2$. Thus the force of drawing proximal planes towards class centers is much stronger for negative class. As a result, the proximal plane for negative class tends to stay at the center of the class, the plane for minority class however is pushed away by the force of margin maximization. Consequently, the separating plane which lies in the middle of two proximal planes is biased to the minority class, which results in low recall for the minority (positive) class.

To mitigate the class imbalance problem discussed above, [93] and [94] propose weighted LPSVM (wLPSVM), in which the optimization (5.5) is revised as

$$\begin{aligned} \min \quad & \frac{C}{2}\boldsymbol{\xi}'\boldsymbol{N}\boldsymbol{\xi} + \frac{1}{2}(\boldsymbol{w}'\boldsymbol{w} + b^2) \\ \text{s.t.} \quad & \boldsymbol{D}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{e}b) + \boldsymbol{\xi} = \boldsymbol{e}, \end{aligned} \tag{5.17}$$

where $\boldsymbol{N}$ is a diagonal weighting matrix

$$\boldsymbol{N}_{ii} = \begin{cases} \sigma_+ & \text{if } y_i = +1 \\ \sigma_- & \text{if } y_i = -1. \end{cases} \tag{5.18}$$

According to the level of imbalance [95], class weights σ_+ and σ_- are determined as

$$\begin{aligned} \sigma_+ &= l_-/(l_+ + l_-) \\ \sigma_- &= l_+/(l_+ + l_-), \end{aligned} \tag{5.19}$$

where l_+ and l_- are the number of instances from positive and negative class respectively.

Similar to (5.13), (5.17) can be solved as in LPSVM [89], and the solution of wLPSVM is given as

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = (\frac{\boldsymbol{I}}{C} + \boldsymbol{F}'\boldsymbol{N}\boldsymbol{F})^{-1}\boldsymbol{F}'\boldsymbol{D}\boldsymbol{N}\boldsymbol{e}, \tag{5.20}$$

where $\boldsymbol{F} = \begin{bmatrix} \boldsymbol{X} & -\boldsymbol{e} \end{bmatrix}$.

For nonlinear case, above weighting scheme can be easily adopted into ESVM to facilitate it with class imbalance robustness. Simply replacing $\boldsymbol{X}$ with $\Phi(\boldsymbol{X})$ in (5.17), we have the optimization for wESVM

$$\begin{aligned} \min \quad & \frac{C}{2}\boldsymbol{\xi}'\boldsymbol{N}\boldsymbol{\xi} + \frac{1}{2}(\boldsymbol{w}'\boldsymbol{w} + b^2) \\ \text{s.t.} \quad & \boldsymbol{D}(\Phi(\boldsymbol{X})\boldsymbol{w} - \boldsymbol{e}b) + \boldsymbol{\xi} = \boldsymbol{e}. \end{aligned} \tag{5.21}$$

Again solving (5.21) as in LPSVM [89], we have the solution of wESVM as

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = (\frac{\boldsymbol{I}}{C} + \boldsymbol{E}'\boldsymbol{N}\boldsymbol{E})^{-1}\boldsymbol{E}'\boldsymbol{D}\boldsymbol{N}\boldsymbol{e}, \tag{5.22}$$

where $\boldsymbol{E} = \begin{bmatrix} \Phi(\boldsymbol{X}) & -\boldsymbol{e} \end{bmatrix}$. For simplicity, we let

$$\begin{aligned} \boldsymbol{M} &= \boldsymbol{E}'\boldsymbol{N}\boldsymbol{E} \\ \boldsymbol{v} &= \boldsymbol{E}'\boldsymbol{D}\boldsymbol{N}\boldsymbol{e}, \end{aligned} \tag{5.23}$$

then (5.22) can be written as

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = (\frac{\boldsymbol{I}}{C} + \boldsymbol{M})^{-1}\boldsymbol{v}. \tag{5.24}$$

Note that wLPSVM is formulated very similar to wESVM, where the only difference is that wLPSVM uses original input data $\boldsymbol{X}$ instead of data projection $\Phi(\boldsymbol{X})$. For the rest of this work, we address only nonlinear wESVM without loss of generality.

The complete procedure of batch wESVM training is stated in Algorithm 15.

5.4 Summary

Based on above observation, we found that ESVM is essentially an LPSVM in ELM feature space, thus ESVM is a more generalized LPSVM. We also found that, wESVM is a more generalized ESVM because ESVM can be seen as a special case of wESVM in which the two classes are always equally weighted.

Thus in the following Chapter, we intend to conduct our PI integration study on the most generalized model, the wESVM.

Algorithm 15 Batch wESVM Algorithm

Input: $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, $\boldsymbol{Y} \in \{+1, -1\}^{n \times 1}$, $C \in \mathbb{R}^{+}$.

Output: $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$.

1: Count the number of instances l_+ and l_- for both classes;
2: Compute σ_+ and σ_- as (5.19);
3: Generate $\boldsymbol{N}$ as (5.18);
4: Compute $\Phi(\boldsymbol{X})$ as (5.11) and (5.12);
5: Generate $\boldsymbol{E}$, as $\boldsymbol{E} = \begin{bmatrix} \Phi(\boldsymbol{X}) & -\boldsymbol{e} \end{bmatrix}$;
6: Transform $\boldsymbol{Y}$ into $\boldsymbol{D}$, as $\boldsymbol{D} = diag(\boldsymbol{Y})$;
7: Generate $\boldsymbol{I} \in \mathbb{R}^{(\tilde{d}+1) \times (\tilde{d}+1)}$;
8: Compute $\boldsymbol{M}$ and $\boldsymbol{v}$ as (5.23)
9: Compute $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$ as (5.24).

6 Proposed PI integrated algorithm

6.1 Introduction

In this chapter, we apply the knowledge merging on wESVM to develop a PI integrated system. The idea is to enable merging of wESVMs on subsets of data into one model whose learning result equals that from the whole dataset. In doing that, we derive a new formula of wESVM in which knowledge is represented as a set of class-wised matrices, and we prove that the merging of wESVMs can be performed through simple matrix addition. Through such knowledge merging, parallel learning can be implemented by letting multiple nodes learning simultaneously from data slices, then combining knowledge obtained; also incremental learning can be carried out by adding up knowledge acquired at different incremental stages. Thus, wESVM is transformed without information lost for PI integration.

The proposed algorithm is implemented in MapReduce environment. The correctness, efficiency and effectiveness of our algorithm is evaluated in experiment. Our algorithm is also compared with other parallel and parallel incremental algorithms in terms of of classification capability and training time.

6.2 Preliminary

MapReduce is a state of the art framework designed for parallel computation [96, 97]. It is originally developed by Google as a platform for processing very large scale data such as web pages obtained by crawlers and logs of web search request [98]. MapReduce supports parallelism by dealing with some common

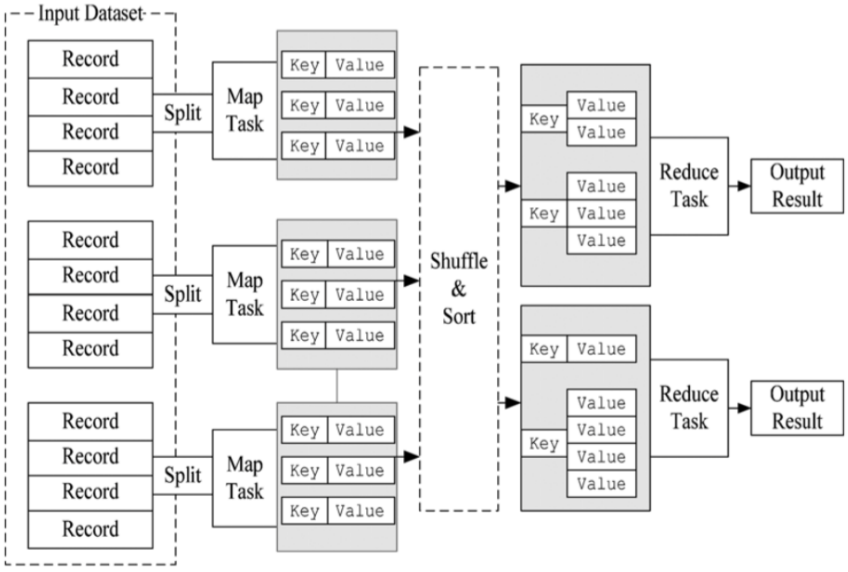


Figure 6.1: MapRedce work flow

issues related to distributed and parallel programming such as load balancing, network communication, fault tolerance etc. Thus programmers can abstract from these issues and concentrate on data processing design. Figure 6.1 presents the work flow for a standard MapReduce execution.

There are three major phases in a MapReduce program execution: Map, Shuffle and Reduce. At the Map phase, input data are firstly splitted into several slices and each slice is then processed independently by a Map task. Each Map task takes one input record at a time and generates one or a group of intermediate key/value $< k_i, v_i >$ pairs as a part of the Map phase output. The processing within the Map task is specified by user written Map function. In the Shuffle phase, all works are conducted by MapReduce framework. All intermediate key/value pairs are firstly sorted by their keys, then all values associated with the same key are grouped together. The output of Shuffle phase is in form of set of unique keys and each of which is followed by a list of values that the key is associated with $< k_i, V_i >$. At the Reduce phase, one Reduce task picks one key at a time and the list of values associated with that key, and then merges all these values into the final result. The merging process is defined in Reduce function, which is also specified by user.

MapReduce applications are executed completely in parallel at two phases. At the Map phase, it is executed independently for all map tasks, since each of Map tasks works on its own input data slice. At Reduce phase, all Reduce tasks are still executed independently, although one Reduce task may depend on the outputs from various Map tasks. This is explained that each Reduce task works on one key at a time, and all values associated with that key have already been grouped at the Shuffle phase.

6.3 Proposed PI Integrated wESVM

Recall that training a batch wESVM involves two steps: the calculation of $\boldsymbol{M}$ and $\boldsymbol{v}$, and (5.24) execution. According to (5.23), the computational complexities of $\boldsymbol{M}$ and $\boldsymbol{v}$ increase linearly to the number of samples. The execution of (5.24) has the complexity of $O(\tilde{d}^3)$ which is fixed for any given $\tilde{d}$. Thus, the computational cost on calculating $\boldsymbol{M}$ and $\boldsymbol{v}$ is dominant, given a large scale dataset for wESVM training. Therefore, we address only the parallel updating of $\boldsymbol{M}$ and $\boldsymbol{v}$ for the derivation of PI integrated wESVM.

As discussed before, we intend to formulate the parallel and incremental learning of $\boldsymbol{M}$ and $\boldsymbol{v}$ as a knowledge merging problem. Let us take learning $\boldsymbol{M}$ as an example. Assume that we have dataset $\boldsymbol{\mathcal{S}}_a$, $\boldsymbol{\mathcal{S}}_b$ and $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}_a \cup \boldsymbol{\mathcal{S}}_b$, $\boldsymbol{M}_a$ and $\boldsymbol{M}_b$ learned from $\boldsymbol{\mathcal{S}}_a$ and $\boldsymbol{\mathcal{S}}_b$ respectively. If we can merge $\boldsymbol{M}_a$ and $\boldsymbol{M}_b$ into $\tilde{\boldsymbol{M}}$ that equals to $\boldsymbol{M}$ learned directly from data $\boldsymbol{\mathcal{S}}$, then we have both incremental and parallel learning problem solved. This is because, for incremental learning we can always have the updated $\tilde{\boldsymbol{M}}$ by merging current $\boldsymbol{M}$ with $\boldsymbol{M_t}$, which is the learning result from newly arrived dataset $\boldsymbol{\mathcal{S}_t}$. Also, for parallel learning we can obtain $\boldsymbol{M}$ by merging all $\{\boldsymbol{M}_1, \ldots \boldsymbol{M}_n, \ldots \boldsymbol{M}_N\}$ learned simultaneously from data slices. Note that the above principle also applies to the learning of $\boldsymbol{v}$. Thus for developing PI integrated wESVM, we turn to merge $\boldsymbol{M}$ and $\boldsymbol{v}$ obtained from different datasets.

6.3.1 wESVM Reformulation for Merging

Consider merging $\boldsymbol{M}$ and $\boldsymbol{v}$. We have the following knowledge merging rules

Lemma 11. *Let* $\begin{bmatrix} \boldsymbol{X} & \boldsymbol{Y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{X}_a & \boldsymbol{Y}_a \\ \boldsymbol{X}_b & \boldsymbol{Y}_b \end{bmatrix}$, $\boldsymbol{E} = \begin{bmatrix} \boldsymbol{E}_a \\ \boldsymbol{E}_b \end{bmatrix} = \begin{bmatrix} \Phi(\boldsymbol{X}_a) & -\boldsymbol{e} \\ \Phi(\boldsymbol{X}_b) & -\boldsymbol{e} \end{bmatrix}$, $\boldsymbol{D} = \begin{bmatrix} \boldsymbol{D}_a & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{D}_b \end{bmatrix}$ *and* $\boldsymbol{N} = \begin{bmatrix} \boldsymbol{N}_a & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{N}_b \end{bmatrix}$. *Then,*

$$\begin{aligned} \boldsymbol{M} &= \boldsymbol{E}'\boldsymbol{N}\boldsymbol{E} \\ &= \boldsymbol{E}'_a\boldsymbol{N}_a\boldsymbol{E}_a + \boldsymbol{E}'_b\boldsymbol{N}_b\boldsymbol{E}_b \\ &= \boldsymbol{M}_a + \boldsymbol{M}_b \\ \boldsymbol{v} &= \boldsymbol{E}'\boldsymbol{D}\boldsymbol{N}\boldsymbol{e} \\ &= \boldsymbol{E}'_a\boldsymbol{D}_a\boldsymbol{N}_a\boldsymbol{e} + \boldsymbol{E}'_b\boldsymbol{D}_b\boldsymbol{N}_b\boldsymbol{e}, \\ &= \boldsymbol{v}_a + \boldsymbol{v}_b \end{aligned} \tag{6.1}$$

$$\begin{aligned} \boldsymbol{E}'\boldsymbol{E} &= \boldsymbol{E}'_a\boldsymbol{E}_a + \boldsymbol{E}'_b\boldsymbol{E}_b \\ \boldsymbol{E}'\boldsymbol{e} &= \boldsymbol{E}'_a\boldsymbol{e} + \boldsymbol{E}'_b\boldsymbol{e} \end{aligned} \tag{6.2}$$

□

Note that (6.1) holds only when weighting matrix $\boldsymbol{N}_a$, $\boldsymbol{N}_b$ and $\boldsymbol{N}$ share the same class weights. This implies that each class weight must be a constant for merging multiple $\boldsymbol{M}$ or $\boldsymbol{v}$ terms, which is problematic for either incremental or parallel learning of wESVM.

Consider incremental learning. The level of imbalance varies over time, since new data are being presented continuously. Then class weights must be changing accordingly, as class weight by (5.19) is a function of class imbalance. Thus , Knowledge merging via (6.1) is not a solution to incremental wESVM.

Consider parallel learning. If we put slave nodes on learning local data which normally has different class imbalance, then we are not able to merge the knowledge (i.e., $\boldsymbol{M}$ and $\boldsymbol{v}$) from individual slave nodes as the knowledge on those nodes are with different weights. Alternatively, if we force all slave nodes to use the same class weights, then the only option is to use the weights calculated from the global imbalance (i.e., the class imbalance of whole dataset). Consider the global imbalance is unknown for individual slave node, since each node has only access to local data. Thus knowledge merging via (6.1) is also problematic for parallel learning of wESVM.

Now, we seek an alternative solution to enable both knowledge merging and weight updating. In doing so, we reformulate the original wESVM (i.e., (5.22)) as follows.

Let partition a consist of one sample. We apply Lemma 11 to split partition b iteratively until only one sample is left. Then, we have

Lemma 12.

$$\begin{aligned} \boldsymbol{E}'\boldsymbol{N}\boldsymbol{E} &= \boldsymbol{E}_1'\boldsymbol{N}_{11}\boldsymbol{E}_1 + \ldots + \boldsymbol{E}_n'\boldsymbol{N}_{nn}\boldsymbol{E}_n \\ &= \sum_{i=1}^{n} \boldsymbol{N}_{ii} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix} \\ \boldsymbol{E}'\boldsymbol{D}\boldsymbol{N}\boldsymbol{e} &= \boldsymbol{E}_1'\boldsymbol{D}_{11}\boldsymbol{N}_{11} + \ldots + \boldsymbol{E}_n'\boldsymbol{D}_{nn}\boldsymbol{N}_{nn} \\ &= \sum_{i=1}^{n} y_i \boldsymbol{N}_{ii} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}', \end{aligned} \tag{6.3}$$

$$\begin{aligned} \boldsymbol{E}'\boldsymbol{E} &= \boldsymbol{E}_1'\boldsymbol{E}_1 + \ldots + \boldsymbol{E}_n'\boldsymbol{E}_n = \sum_{i=1}^{n} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix} \\ \boldsymbol{E}'\boldsymbol{e} &= \boldsymbol{E}_1' + \ldots + \boldsymbol{E}_n' = \sum_{i=1}^{n} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}'. \end{aligned} \tag{6.4}$$

□

For terms on the right-hand side of (6.3), if we group all terms that have

$y_i = +1$ and $y_i = -1$ respectively, then have

$$\begin{aligned}
\boldsymbol{E}'\boldsymbol{N}\boldsymbol{E} =& \sum_{y_i=+1} \boldsymbol{N}_{ii} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix} + \\
& \sum_{y_j=-1} \boldsymbol{N}_{jj} \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix} \\
=& \sigma_+ \sum_{y_i=+1} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix} + \\
& \sigma_- \sum_{y_j=-1} \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix} \\
\boldsymbol{E}'\boldsymbol{D}\boldsymbol{N}\boldsymbol{e} =& \sum_{y_i=+1} y_i \boldsymbol{N}_{ii} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' + \\
& \sum_{y_j=-1} y_j \boldsymbol{N}_{jj} \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix}' \\
=& \sigma_+ \sum_{y_i=+1} y_i \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' + \\
& \sigma_- \sum_{y_j=-1} y_j \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix}',
\end{aligned} \tag{6.5}$$

since by (5.18) $\boldsymbol{N}_{ii}$ equals to σ_+ when $y_i = +1$ and σ_- when $y_i = -1$. Applying (6.4) to (6.5), we reformulate $\boldsymbol{M}$ and $\boldsymbol{v}$ as

Lemma 13. *Let $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ be the instance matrix of positive and negative class respectively, $\boldsymbol{E}_+ = \begin{bmatrix} \Phi(\boldsymbol{X}_+) & -\boldsymbol{e} \end{bmatrix}$, $\boldsymbol{E}_- = \begin{bmatrix} \Phi(\boldsymbol{X}_-) & -\boldsymbol{e} \end{bmatrix}$, then*

$$\begin{aligned}
\boldsymbol{M} = \boldsymbol{E}'\boldsymbol{N}\boldsymbol{E} =& \sigma_+ \sum_{y_i=+1} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix} + \\
& \sigma_+ \sum_{y_j=-1} \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix}' \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix} \\
=& \sigma_+ \boldsymbol{E}_+{}'\boldsymbol{E}_+ + \sigma_- \boldsymbol{E}_-{}'\boldsymbol{E}_- \\
\boldsymbol{v} = \boldsymbol{E}'\boldsymbol{D}\boldsymbol{N}\boldsymbol{e} =& \sigma_+ \sum_{y_i=+1} \begin{bmatrix} \Phi(x_i) & -1 \end{bmatrix}' + \\
& \sigma_- \sum_{y_j=-1} y_j \begin{bmatrix} \Phi(x_j) & -1 \end{bmatrix}' \\
=& \sigma_+ \boldsymbol{E}_+{}'\boldsymbol{e} - \sigma_- \boldsymbol{E}_-{}'\boldsymbol{e}
\end{aligned} \tag{6.6}$$

□

Let $\boldsymbol{M}_+ = \boldsymbol{E}_+{}'\boldsymbol{E}_+$, $\boldsymbol{M}_- = \boldsymbol{E}_-{}'\boldsymbol{E}_-$, $\boldsymbol{v}_+ = \boldsymbol{E}_+{}'\boldsymbol{e}$ and $\boldsymbol{v}_- = \boldsymbol{E}_-{}'\boldsymbol{e}$, we reformulate the original wESVM solution (5.22) as

$$\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix} = (\frac{\boldsymbol{I}}{C} + \sigma_+\boldsymbol{M}_+ + \sigma_-\boldsymbol{M}_-)^{-1}(\sigma_+\boldsymbol{v}_+ - \sigma_-\boldsymbol{v}_-). \tag{6.7}$$

As a result, $\boldsymbol{M}_+$, $\boldsymbol{M}_-$, $\boldsymbol{v}_+$ and $\boldsymbol{v}_-$ can be merged according to (6.2). Also, the class weights σ_+ and σ_- can be updated in response to current global class imbalance since they are now real value coefficients.

In above reformulated wESVM, we let

$$\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\} \tag{6.8}$$

be the knowledge of wESVM on $\boldsymbol{\mathcal{S}}$. Algorithm 16 gives the steps of wESVM knowledge extraction.

Algorithm 16 wESVM Knowledge Extraction

Input: $\boldsymbol{\mathcal{S}} = \{\boldsymbol{X}, \boldsymbol{Y}\}$.

Output: $\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\}$.

1: Split $\boldsymbol{X}$ into $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ according to Y;
2: Count the number of instances l_+ and l_- in $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ respectively;
3: Compute $\Phi(\boldsymbol{X}_+)$ and $\Phi(\boldsymbol{X}_-)$ as (5.11) and (5.12);
4: Generate $\boldsymbol{E}_+$ and $\boldsymbol{E}_-$ as $\boldsymbol{E} = \begin{bmatrix} \Phi(\boldsymbol{X}) & -\boldsymbol{e} \end{bmatrix}$;
5: Compute $\boldsymbol{M}_+ = \boldsymbol{E}_+{}'\boldsymbol{E}_+$, $\boldsymbol{M}_- = \boldsymbol{E}_-{}'\boldsymbol{E}_-$, $\boldsymbol{v}_+ = \boldsymbol{E}_+{}'\boldsymbol{e}$ and $\boldsymbol{v}_- = \boldsymbol{E}_-{}'\boldsymbol{e}$;
6: Return $\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\}$.

Given $\boldsymbol{\mathcal{K}}$ on $\boldsymbol{\mathcal{S}}$, Algorithm 17 describes the steps of wESVM classifier $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$ calculation.

Algorithm 17 wESVM Knowledge to Solution

Input: $\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\}$.

Output: $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$.

1: Compute weights σ_+ and σ_- as (5.19);
2: Compute $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$ as (6.7).

6.3.2 Incremental and Decremental wESVM

In this section, we discuss the wESVM updating according to newly arrived samples and samples no longer useful, respectively.

Given dataset $\boldsymbol{\mathcal{S}}$ and its wESVM knowledge $\boldsymbol{\mathcal{K}}$. Let $\boldsymbol{\mathcal{S}}_r$ be the set of data to be retired and $\boldsymbol{\mathcal{S}}_l$ be the remaining data such that

$$\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}_l \cup \boldsymbol{\mathcal{S}}_r. \tag{6.9}$$

Let $\boldsymbol{\mathcal{S}}_a$ denotes the newly arrived data to be added into the training set, $\tilde{\boldsymbol{\mathcal{S}}}$ denotes the data after adding and retiring. We have

$$\tilde{\boldsymbol{\mathcal{S}}} = \boldsymbol{\mathcal{S}}_l \cup \boldsymbol{\mathcal{S}}_a. \tag{6.10}$$

Thus given $\boldsymbol{\mathcal{K}}$ on $\boldsymbol{\mathcal{S}}$, we update $\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\}$ in response to data updates $\boldsymbol{\mathcal{S}}_a$ and $\boldsymbol{\mathcal{S}}_r$, respectively. The objective of incremental and decremental wESVM is to obtain updated wESVM knowledge that equals the batch wESVM $\tilde{\boldsymbol{\mathcal{K}}} = \{\tilde{\boldsymbol{M}}_+, \tilde{\boldsymbol{M}}_-, \tilde{\boldsymbol{v}}_+, \tilde{\boldsymbol{v}}_-, \tilde{l}_+, \tilde{l}_-\}$ on $\tilde{\boldsymbol{\mathcal{S}}}$.

For updating $\boldsymbol{M}_+$ and $\boldsymbol{v}_+$, we have the following according to Lemma 11,

$$\begin{aligned} \boldsymbol{M}_+ &= \boldsymbol{E}_+{}'\boldsymbol{E}_+ = \boldsymbol{E}_{l+}{}'\boldsymbol{E}_{l+} + \boldsymbol{E}_{r+}{}'\boldsymbol{E}_{r+} \\ \boldsymbol{v}_+ &= \boldsymbol{E}_+{}'\boldsymbol{e} = \boldsymbol{E}_{l+}{}'\boldsymbol{e} + \boldsymbol{E}_{r+}{}'\boldsymbol{e}, \end{aligned} \tag{6.11}$$

$$\begin{aligned} \tilde{\boldsymbol{M}}_+ &= \tilde{\boldsymbol{E}}_+{}'\tilde{\boldsymbol{E}}_+ = \boldsymbol{E}_{l+}{}'\boldsymbol{E}_{l+} + \boldsymbol{E}_{a+}{}'\boldsymbol{E}_{a+} \\ \tilde{\boldsymbol{v}}_+ &= \tilde{\boldsymbol{E}}_+{}'\boldsymbol{e} = \boldsymbol{E}_{l+}{}'\boldsymbol{e} + \boldsymbol{E}_{a+}{}'\boldsymbol{e}, \end{aligned} \tag{6.12}$$

since (6.9) and (6.10) applies for both positive and negative classes. Substituting (6.11) into (6.12), we have the updating rule for $\boldsymbol{M}_+$ and $\boldsymbol{v}_+$ as

$$\begin{aligned} \tilde{\boldsymbol{M}}_+ &= \boldsymbol{M}_+ - \boldsymbol{E}_{r+}{}'\boldsymbol{E}_{r+} + \boldsymbol{E}_{a+}{}'\boldsymbol{E}_{a+} \\ &= \boldsymbol{M}_+ - \boldsymbol{M}_{r+} + \boldsymbol{E}_{a+} \\ \tilde{\boldsymbol{v}}_+ &= \boldsymbol{v}_+ - \boldsymbol{E}_{r+}{}'\boldsymbol{e} + \boldsymbol{E}_{a+}{}'\boldsymbol{e} \\ &= \boldsymbol{v}_+ - \boldsymbol{v}_{r+} + \boldsymbol{v}_{a+}. \end{aligned} \tag{6.13}$$

Similar to (6.11) to (6.13), we have the updating rule for $\boldsymbol{M}_-$ and $\boldsymbol{v}_-$ as

$$\begin{aligned}\tilde{\boldsymbol{M}}_- &= \boldsymbol{M}_- - \boldsymbol{E}_{r-}{}'\boldsymbol{E}_{r-} + \boldsymbol{E}_{a-}{}'\boldsymbol{E}_{a-} \\ &= \boldsymbol{M}_- - \boldsymbol{M}_{r-} + \boldsymbol{E}_{a-} \\ \tilde{\boldsymbol{v}}_- &= \boldsymbol{v}_- - \boldsymbol{E}_{r-}{}'\boldsymbol{e} + \boldsymbol{E}_{a-}{}'\boldsymbol{e} \\ &= \boldsymbol{v}_- - \boldsymbol{v}_{r-} + \boldsymbol{v}_{a-}.\end{aligned} \tag{6.14}$$

For updating l_+ and l_-, by (6.9) and (6.10) we naturally have

$$\begin{aligned}\tilde{l}_+ &= l_+ - l_{r+} + l_{a+} \\ \tilde{l}_- &= l_- - l_{r-} + l_{a-}.\end{aligned} \tag{6.15}$$

In (6.13)-(6.15), $\boldsymbol{M}_{r+}$, $\boldsymbol{M}_{r-}$, $\boldsymbol{v}_{r+}$, $\boldsymbol{v}_{r-}$, l_{r+}, l_{r-} are obtained from $\boldsymbol{\mathcal{K}}_r$ (wESVM knowledge on $\boldsymbol{\mathcal{S}}_r$), and $\boldsymbol{M}_{a+}$, $\boldsymbol{M}_{a-}$, $\boldsymbol{v}_{a+}$, $\boldsymbol{v}_{a-}$, l_{a+}, l_{a-} are from $\boldsymbol{\mathcal{K}}_a$. $\boldsymbol{\mathcal{K}}_r$ and $\boldsymbol{\mathcal{K}}_a$ are the result of Algorithm 16 execution with $\boldsymbol{\mathcal{S}}_r$ and $\boldsymbol{\mathcal{S}}_a$ as input respectively.

Applying knowledge merging rule (6.13)-(6.15), we have the updated knowledge $\tilde{\boldsymbol{\mathcal{K}}}$ by merging $\boldsymbol{\mathcal{K}}$ with $\boldsymbol{\mathcal{K}}_r$ and $\boldsymbol{\mathcal{K}}_a$. Consequently, we obtain the corresponding updated wESVM classifier $\begin{bmatrix}\tilde{\boldsymbol{w}} \\ \tilde{b}\end{bmatrix}$ by executing Algorithm 17 on $\tilde{\boldsymbol{\mathcal{K}}}$. The steps for proposed incremental/decremental wESVM is shown in Algorithm 18.

Algorithm 18 Proposed Incremental/Decremental wESVM Algorithm

Input: Initial dataset $\boldsymbol{\mathcal{S}} = \{\boldsymbol{X}, \boldsymbol{Y}\}$ and C for the first batch.

Or, initial model $\{\begin{bmatrix}\boldsymbol{w}\\ b\end{bmatrix}, \boldsymbol{\mathcal{K}}\}$, $\boldsymbol{\mathcal{S}_r} = \{\boldsymbol{X}_r, \boldsymbol{Y}_r\}$ to be retired, $\boldsymbol{\mathcal{S}_a} = \{\boldsymbol{X}_a, \boldsymbol{Y}_a\}$ to be added, and C for the rest stages.

Output: Initial model $\{\begin{bmatrix}\boldsymbol{w}\\ b\end{bmatrix}, \boldsymbol{\mathcal{K}}\}$ for the first batch,

or updated model $\{\begin{bmatrix}\tilde{\boldsymbol{w}}\\ \tilde{b}\end{bmatrix}, \tilde{\boldsymbol{\mathcal{K}}}\}$ for the rest stages.

1: **if** there is no initial model (this is the first batch) **then**
2: Call Algorithm 16 with $\boldsymbol{\mathcal{S}} = \{\boldsymbol{X}, \boldsymbol{Y}\}$ as input, obtain $\boldsymbol{\mathcal{K}}$
3: Call Algorithm 17 with $\boldsymbol{\mathcal{K}}$ as input, obtain $\begin{bmatrix}\boldsymbol{w}\\ b\end{bmatrix}$;
4: Return $\{\begin{bmatrix}\boldsymbol{w}\\ b\end{bmatrix}, \boldsymbol{\mathcal{K}}\}$
5: **else**
6: Call Algorithm 16 with $\boldsymbol{\mathcal{S}_r} = \{\boldsymbol{X}_r, \boldsymbol{Y}_r\}$ as input, obtain $\boldsymbol{\mathcal{K}_r}$;
7: Call Algorithm 16 with $\boldsymbol{\mathcal{S}_a} = \{\boldsymbol{X}_a, \boldsymbol{Y}_a\}$ as input, obtain $\boldsymbol{\mathcal{K}_a}$;
8: Merge $\boldsymbol{\mathcal{K}}$ with $\boldsymbol{\mathcal{K}_r}$ and $\boldsymbol{\mathcal{K}_a}$ using (6.13) to (6.15), obtain $\tilde{\boldsymbol{\mathcal{K}}}$;
9: Call Algorithm 17 with $\tilde{\boldsymbol{\mathcal{K}}}$ as input, obtain $\begin{bmatrix}\tilde{\boldsymbol{w}}\\ \tilde{b}\end{bmatrix}$;
10: Return $\{\begin{bmatrix}\tilde{\boldsymbol{w}}\\ \tilde{b}\end{bmatrix}, \tilde{\boldsymbol{\mathcal{K}}}\}$.
11: **end if**

6.3.3 PI Integrated wESVM

The idea of our parallelization of incremental wESVM is to parallelize Algorithm 18 via knowledge merging. To be more specific, we parallelize the knowledge extraction step, Algorithm 16, as it shares the biggest portion of total computational costs.

Let $\boldsymbol{\mathcal{S}}_1, \ldots, \boldsymbol{\mathcal{S}}_t$ be the t slices of data $\boldsymbol{\mathcal{S}}$ such that $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}_1} \cup \ldots \cup \boldsymbol{\mathcal{S}_t}$. Then the parallelization of Algorithm 16 consists of two steps: execute in parallel multiple instances of Algorithm 16 on different data slice $\boldsymbol{\mathcal{S}}_i$, and merge knowledge

obtained $\boldsymbol{\mathcal{K}_i}$ to generate the global knowledge $\boldsymbol{\mathcal{K}}$ on $\boldsymbol{\mathcal{S}}$.

Given $\boldsymbol{\mathcal{K}_i} = \{\boldsymbol{M_{i+}}, \boldsymbol{M_{i-}}, \boldsymbol{v_{i+}}, \boldsymbol{v_{i-}}, l_{i+}, l_{i-}\}$ for $i = 1 \dots t$, by Lemma 11 we merge $\boldsymbol{M}$ and $\boldsymbol{v}$ terms as

$$\begin{aligned} \boldsymbol{M_+} &= \sum_{i=1}^{t} \boldsymbol{M_{i+}} \qquad \boldsymbol{v_+} = \sum_{i=1}^{t} \boldsymbol{v_{i+}} \\ \boldsymbol{M_-} &= \sum_{i=1}^{t} \boldsymbol{M_{i-}} \qquad \boldsymbol{v_-} = \sum_{i=1}^{t} \boldsymbol{v_{i-}}. \end{aligned} \tag{6.16}$$

Naturally, the class sizes can be merged as

$$\begin{aligned} l_+ &= \sum_{i=1}^{t} l_{i+} \\ l_- &= \sum_{i=1}^{t} l_{i-}. \end{aligned} \tag{6.17}$$

The steps for extracting $\boldsymbol{\mathcal{K}_i}$ and merging all $\boldsymbol{\mathcal{K}_i}$ into $\boldsymbol{\mathcal{K}}$ are summarized in Algorithm 19 and 20, respectively. Next, we form the parallel wESVM algorithm as parallel knowledge extraction by Algorithm 19 - 20, followed by wESVM classifier calculation via Algorithm 17. The steps of parallel wESVM algorithm is summarized in Algorithm 21. Replacing Algorithm 16 step in Algorithm 18 with Algorithm 19 - 20, we have our PI integrated wESVM algorithm, whose steps are summarized in Algorithm 22.

Algorithm 19 Extracting wESVM Knowledge from Data Slices at Slave Nodes

Input: $\boldsymbol{\mathcal{S}_i} = \{\boldsymbol{X}_i, \boldsymbol{Y}_i\}$.

Output: $\boldsymbol{\mathcal{K}_i} = \{\boldsymbol{M_{i+}}, \boldsymbol{M_{i-}}, \boldsymbol{v_{i+}}, \boldsymbol{v_{i-}}, l_{i+}, l_{i-}\}$.

1: Call Algorithm 16) with $\boldsymbol{\mathcal{S}_i}$ as input, obtain $\boldsymbol{\mathcal{K}_i}$;

2: Return $\boldsymbol{\mathcal{K}_i}$ to master node.

Algorithm 20 Merging wESVM Knowledge from Data Slices at Master Node

Input: $\boldsymbol{\mathcal{K}_i}$ from all slave nodes.

Output: $\boldsymbol{\mathcal{K}}$

1: Compute $\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-$ as (6.16);

2: Compute l_+, l_- as (6.17);

3: Return $\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\}$.

Algorithm 21 Proposed Parallel wESVM Algorithm

Input: Data slices $\boldsymbol{\mathcal{S}_i} = \{\boldsymbol{X_i}, \boldsymbol{Y_i}\} \quad i = 1, \ldots, t$ and C.

Output: $\{\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}, \boldsymbol{\mathcal{K}}\}$.

1: Call Algorithm 19 in parallel with $\boldsymbol{\mathcal{S}_i} = \{\boldsymbol{X_i}, \boldsymbol{Y_i}\}$ as input, obtain $\boldsymbol{\mathcal{K}_i}$;

2: Call Algorithm 20 with all $\boldsymbol{\mathcal{K}_i}$ as input, obtain $\boldsymbol{\mathcal{K}}$;

3: Call Algorithm 17 with $\boldsymbol{\mathcal{K}}$ as input, obtain $\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$;

4: Return $\{\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}, \boldsymbol{\mathcal{K}}\}$

Algorithm 22 Proposed PI integrated wESVM Algorithm

Input: Initial model $\{\begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}, \boldsymbol{\mathcal{K}}\}$, $\boldsymbol{\mathcal{S}_{ir}} = \{\boldsymbol{X_{ir}}, \boldsymbol{Y_{ir}}\} \quad i = 1, \ldots, t$ to be retired, $\boldsymbol{\mathcal{S}_{ia}} = \{\boldsymbol{X_{ia}}, \boldsymbol{Y_{ia}}\} \quad i = 1, \ldots, t$ to be added, and C.

Output: Updated model $\{\begin{bmatrix} \tilde{\boldsymbol{w}} \\ \tilde{b} \end{bmatrix}, \tilde{\boldsymbol{\mathcal{K}}}\}$.

1: Call Algorithm 19 in parallel with $\boldsymbol{\mathcal{S}_{ir}} = \{\boldsymbol{X_{ir}}, \boldsymbol{Y_{ir}}\}$ as input, obtain $\boldsymbol{\mathcal{K}_{ir}}$;

2: Call Algorithm 20 with all $\boldsymbol{\mathcal{K}_{ir}}$ as input, obtain $\boldsymbol{\mathcal{K}_r}$;

3: Call Algorithm 19 in parallel with $\boldsymbol{\mathcal{S}_{ia}} = \{\boldsymbol{X_{ia}}, \boldsymbol{Y_{ia}}\}$ as input, obtain $\boldsymbol{\mathcal{K}_{ia}}$;

4: Call Algorithm 20 with all $\boldsymbol{\mathcal{K}_{ia}}$ as input, obtain $\boldsymbol{\mathcal{K}_a}$;

5: Merge $\boldsymbol{\mathcal{K}}$ with $\boldsymbol{\mathcal{K}_r}$ and $\boldsymbol{\mathcal{K}_a}$ using (6.13) to (6.15), obtain $\tilde{\boldsymbol{\mathcal{K}}}$;

6: Call Algorithm 17 with $\tilde{\boldsymbol{\mathcal{K}}}$ as input, obtain $\begin{bmatrix} \tilde{\boldsymbol{w}} \\ \tilde{b} \end{bmatrix}$;

7: Return $\{\begin{bmatrix} \tilde{\boldsymbol{w}} \\ \tilde{b} \end{bmatrix}, \tilde{\boldsymbol{\mathcal{K}}}\}$.

In our PI integrated learning algorithm, the knowledge of each slice of new data is represented by $\boldsymbol{\mathcal{K}_{ir}}$, whose size is independent from the data size. Thus our algorithm is suitable for distributed stream learning, when the training data are in multiple distributed streams. In this scenario, we can allocate one learner (running Algorithm 19) at each data source to obtain the local knowledge, and

then transmit the local knowledge, with very low bandwidth cost, to the master learner to obtain a global classifier (using Algorithms 20 and 22).

We also notice that different streams may lead to independent models in the context of distributed stream learning. For instance, assume we have streams A - E where streams A and C belong to the same model and the other three streams lead to another model. In this case, our algorithm can aggregate streams (A, C) and (B, D, E) respectively using the mechanism mentioned above to obtain two independent models. But we still need prior knowledge or another detection algorithm to help determine that (A, C) and (B, D, E) belong to two different models.

6.3.4 MapReduce based Implementation

In this section, we implement proposed PI integrated wESVM in MapReduce environment. To be more specific, we implement in Hadoop parallel wESVM knowledge extraction, which includs Algorithm 19 and 20.

Given dataset $\boldsymbol{\mathcal{S}}$ stored on Hadoop distributed file system (HDFS) as a sequence of $< key, value >$ pairs, where each pair stands for one record $(\boldsymbol{x}_i, y_i)$. Here key is the offset of record to the start point of data file, and $value$ is the $(\boldsymbol{x}_i, y_i)$ in string format.

As introduced before, each Map function processes only one record at a time. To extract knowledge $\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\}$ from the entire dataset $\boldsymbol{\mathcal{S}}$, we extract knowledge from each individual sample, and merge the knowledge obtained as follows.

For $\boldsymbol{M}$ and $\boldsymbol{v}$, by (6.4) we have

$$\begin{aligned}
\boldsymbol{M}_+ &= \boldsymbol{E}_+{}'\boldsymbol{E}_+ = \sum_{y_i=+1} \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix}' \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix} = \sum_{y_i=+1} \boldsymbol{M}_i \\
\boldsymbol{M}_- &= \boldsymbol{E}_-{}'\boldsymbol{E}_- == \sum_{y_i=-1} \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix}' \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix} = \sum_{y_i=-1} \boldsymbol{M}_i \\
\boldsymbol{v}_+ &= \boldsymbol{E}_+{}'\boldsymbol{e} = \sum_{y_i=+1} \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix}' = \sum_{y_i=+1} \boldsymbol{v}_i \\
\boldsymbol{v}_- &= \boldsymbol{E}_-{}'\boldsymbol{e} = \sum_{y_i=-1} \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix}' = \sum_{y_i=-1} \boldsymbol{v}_i.
\end{aligned} \tag{6.18}$$

For l_+ and l_-, we have

$$l_+ = \sum_{y_i=+1} l_i \quad l_- = \sum_{y_i=-1} l_i, \tag{6.19}$$

where $l_i = 1$. The Hadoop implementation of (6.18) and (6.19) includes three phases: Map, Shuffle and Reduce.

At Map phase, for each input sample $(\boldsymbol{x}_i, y_i)$, we compute $\boldsymbol{M}_i$ and $\boldsymbol{v}_i$ as

$$\begin{aligned} \boldsymbol{M}_i &= \begin{bmatrix} \Phi(\boldsymbol{x}_i)' & -1 \end{bmatrix}' \begin{bmatrix} \Phi(\boldsymbol{x}_i)' & -1 \end{bmatrix} \\ \boldsymbol{v}_i &= \begin{bmatrix} \Phi(\boldsymbol{x}_i)' & -1 \end{bmatrix}' \end{aligned} \tag{6.20}$$

and set $l_i = 1$. To transmit square matrix $\boldsymbol{M}_i$, column vector $\boldsymbol{v}_i$ and count number $l_i = 1$ from Map to Shuffle phase, we define two types of $< key, value >$ pair as

1. Vector pair in which the *key* is $(y_i, vector, idx)$ where *vector* indicates this pair transmits a column vector and idx is the index of the column transmitted; and the *value* is a $(\tilde{d}+1) \times 1$ column vector which equals the idx-th column of $\boldsymbol{M}_i \in \mathbb{R}^{(\tilde{d}+1)\times(\tilde{d}+1)}$ for $1 \leq idx \leq (\tilde{d}+1)$ or $\boldsymbol{v}_i \in \mathbb{R}^{(\tilde{d}+1)\times 1}$ for $idx = \tilde{d}+2$.

2. Count pair in which the *key* is $(y_i, count)$ where *count* indicates that this pair transmits a count number; and the *value* is an integer 1.

Note that y_i is inserted in the *key* here, because $\boldsymbol{M}_i$, $\boldsymbol{v}_i$ and l_i by (6.18) and (6.19) are summed according to y_i.

The steps of Map function are shown in Algorithm 23, through which $\tilde{d}+2$ $< key, value >$ vector pairs and 1 count pair are generated for each $(\boldsymbol{x}_i,\ y_i)$. Thus for the entire dataset $\boldsymbol{\mathcal{S}} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, we have the output of Map phase as $n(\tilde{d}+3) < key, value >$ pairs with $2\tilde{d}+6$ unique *key*s.

Algorithm 23 MapReduce Implementation of Algorithm 7 - Map Function

Input: The offset key, the sample $(\boldsymbol{x}_i,\ y_i)$ $value$.

Output: Vector type $< key, value >$ pairs, count type $< key, value >$ pair.

1: Compute $\Phi(\boldsymbol{x}_i)$ as (5.11);
2: Compute $\boldsymbol{M}_i = \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix}' \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix}$ and $\boldsymbol{v}_i = \begin{bmatrix}\Phi(\boldsymbol{x}_i)' & -1\end{bmatrix}'$;
3: **for** $idx = 1 : (\tilde{d} + 1)$ **do**
4: Set key as $(y_i, vector, idx)$;
5: Set $value$ as the idx-th column of $\boldsymbol{M}_i$;
6: Emit $< key, value >$ pair;
7: **end for**
8: Set key as $(y_i, vector, (\tilde{d} + 2))$;
9: Set $value$ as the $\boldsymbol{v}_i$;
10: Emit $< key, value >$ pair;
11: Set key as $(y_i, count)$;
12: Set $value$ as 1;
13: Emit $< key, value >$ pair.

At shuffle phase, the $< key, value >$ pairs from Map phase are grouped according to their key by Hadoop framework. As a result, $2\tilde{d} + 6 < key, V >$ pairs are forwarded to Reduce phase as the output of shuffle phase, where V is the set of $value$ that associated with the key.

At Reduce phase, the sum operation of (6.18) and (6.19) is conducted to obtain $\boldsymbol{\mathcal{K}} = \{\boldsymbol{M}_+, \boldsymbol{M}_-, \boldsymbol{v}_+, \boldsymbol{v}_-, l_+, l_-\}$. For each input $< key, V >$, all $value$s in V are summed up as the output $value$. The steps of Reduce function are stated in Algorithm 24. The output of Reduce phase consists of 2 count pairs which output l_+ and l_-, and $2\tilde{d} + 4$ column pairs which give $\boldsymbol{M}_+$, $\boldsymbol{M}_-$, $\boldsymbol{v}_+$ and $\boldsymbol{v}_-$.

Algorithm 24 MapReduce Implementation of Algorithm 8 - Reduce Function

Input: Vector or count *key*, *value* set V with the same *key*.

Output: Vector or count $< key, value >$ pairs, where *value* is either a sample count number, a colume of $\boldsymbol{M}$ matrix or a $\boldsymbol{v}$ vector.

1: **if** *key* is vector type **then**
2: Initialize a vector *temp*;
3: **else if** *key* is count type **then**
4: Initialize an integer *temp*;
5: **end if**
6: **while** $V.hasNext()$ **do**
7: Compute $temp = temp + V.next()$;
8: **end while**
9: Set *value* as *temp*;
10: Emit $< key, value >$ pair.

6.3.5 Speedup Analysis

In this section, we estimate the training time and speedup of our MapReduce implementation of proposed algorithms in response to different number of nodes and data sizes. This estimation is based on pure computational complexity analysis, where time costs on nodes communication and coordination are not counted.

The computation at Map phase includes mapping all $\boldsymbol{x}_i$ into $\Phi(\boldsymbol{x}_i)$ which takes $2\tilde{d}(d+1)n$ operations, and computing $\boldsymbol{M}_i$ for all $\boldsymbol{x}_i$ which has the complexity of $(\tilde{d}+1)^2 n$. Here n is the number of sample, d and $\tilde{d}$ is the dimensionality of $\boldsymbol{x}_i$ and $\Phi(\boldsymbol{x}_i)$ respectively. The computation at Reduce phase is about the summing of $\boldsymbol{M}_i$ and $\boldsymbol{v}_i$ which takes $(\tilde{d}+1)^2 n$ and $(\tilde{d}+1)n$ operations respectively. Consider $d \ll n$ and $\tilde{d} \ll n$. The cost for both Map (Algorithm 23) and Reduce (Algorithm 24) phase is $O(n)$. For serial part (Algorithm 17) which includes $(\tilde{d}+1) \times (\tilde{d}+1)$ matrix addition and inversion, the cost is $O(\tilde{d}^3)$.

Let $T(n,t)$ be the total execution time for learning a dataset with n samples using t nodes, then we have

$$T(n,t) = T_M(n,t) + T_R(n,t) + T_S, \tag{6.21}$$

where $T_M(n,t)$, $T_R(n,t)$ and T_S are the execution time for Map, Reduce and serial part respectively.

Let $S_M(n,t)$ and $S_R(n,t)$ be the speedup at Map and Reduce phase execution. The computational cost of Map and Reduce phase is evenly distributed on all nodes, thus we have linear speedup against number of nodes

$$\begin{aligned} S_M(n,t) &= \frac{T_M(n,1)}{T_M(n,t)} = t \\ S_R(n,t) &= \frac{T_R(n,1)}{T_R(n,t)} = t. \end{aligned} \tag{6.22}$$

As discussed above, the complexity of both Map and Reduce phases is $O(n)$, then we have linear execution time against data size

$$\begin{aligned} \frac{T_M(n,t)}{T_M(1,t)} &= n \\ \frac{T_R(n,t)}{T_R(1,t)} &= n. \end{aligned} \tag{6.23}$$

Let $S(n,t)$ be the overall speedup for learning n samples using t nodes, then we have

$$S(n,t) = \frac{T(n,1)}{T(n,t)} = \frac{T_M(n,1) + T_R(n,1) + T_S}{T_M(n,t) + T_R(n,t) + T_S}. \tag{6.24}$$

According to (6.22) and (6.24), it is easy to know that

$$\begin{aligned} S(n,t) &< \frac{T_M(n,1) + T_R(n,1) + tT_S}{T_M(n,t) + T_R(n,t) + T_{Ser}} \\ &= \frac{tT_M(n,t) + tT_R(n,t) + tT_S}{T_M(n,t) + T_R(n,t) + T_S} = t, \end{aligned} \tag{6.25}$$

when $t > 1$. This indicates the overall speedup is less than linear against t.

Given t_1 and t_2 with $1 < t_1 < t_2$. According to (6.22) and (6.24), we know that

$$\begin{aligned} \frac{S(n,t_1)}{S(n,t_2)} &= \frac{T_M(n,1) + T_R(n,1) + T_S}{T_M(n,t_1) + T_R(n,t_1) + T_S} \Big/ \frac{T_M(n,1) + T_R(n,1) + T_S}{T_M(n,t_2) + T_R(n,t_2) + T_S} \\ &= \frac{T_M(n,t_2) + T_R(n,t_2) + T_S}{T_M(n,t_1) + T_R(n,t_1) + T_S} \\ &= \frac{t_2(T_M(n,1) + T_R(n,1) + t_2T_S)}{t_1(T_M(n,1) + T_R(n,1) + t_1T_S)} \\ &> \frac{t_2(T_M(n,1) + T_R(n,1) + t_1T_S)}{t_1(T_M(n,1) + T_R(n,1) + t_1T_S)} = \frac{t_2}{t_1} > 1, \end{aligned} \tag{6.26}$$

showing that the overall speedup decreases when the number of nodes grows.

According to (6.21) and (6.23), we know that

$$\begin{aligned}\frac{T(n,t)}{T(1,t)} &= \frac{T_M(n,t)+T_R(n,t)+T_S}{T_M(1,t)+T_R(1,t)+T_S} \\ &= \frac{nT_M(1,t)+nT_R(1,t)+T_S}{T_M(1,t)+T_R(1,t)+T_S} \\ &< \frac{nT_M(1,t)+nT_R(1,t)+nT_S}{T_M(1,t)+T_R(1,t)+T_S} = n,\end{aligned} \tag{6.27}$$

indicating a less than linear ratio between the overall training time and data size n. In other words, learning a n times larger dataset costs less than n times of training duration.

Given data sizes $n_2 > n_1 > 1$, according to (6.24) and (6.23) we have

$$\begin{aligned}\frac{S(n_2,t)}{S(n_1,t)} &= \frac{T_M(n_2,1)+T_R(n_2,1)+T_S}{T_M(n_2,t)+T_R(n_2,t)+T_S} \Big/ \frac{T_M(n_1,1)+T_R(n_1,1)+T_S}{T_M(n_1,t)+T_R(n_1,t)+T_S} \\ &= \frac{n_2(T_M(1,1)+T_R(1,1))+T_S}{n_2(T_M(1,t)+T_R(1,t))+T_S} \frac{n_1(T_M(1,t)+T_R(1,t))+T_S}{n_1(T_M(1,1)+T_R(1,1))+T_S} \\ &= \frac{n_2(T_M(1,1)+T_R(1,1))+T_S}{\frac{n_2}{t}(T_M(1,1)+T_R(1,1))+T_S} \frac{\frac{n_1}{t}(T_M(1,1)+T_R(1,1))+T_S}{n_1(T_M(1,1)+T_R(1,1))+T_S} \\ &= \frac{n_2(T_M(1,1)+T_R(1,1))+T_S}{n_2(T_M(1,t)+T_R(1,t))+tT_S} \frac{n_1(T_M(1,t)+T_R(1,t))+tT_S}{n_1(T_M(1,1)+T_R(1,1))+T_S} \\ &= \frac{n_1n_2(T_M(1,1)+T_R(1,1))^2+(n_2t+n_1)(T_M(1,1)+T_R(1,1))T_S+{T_S}^2}{n_1n_2(T_M(1,1)+T_R(1,1))^2+(n_1t+n_2)(T_M(1,1)+T_R(1,1))T_S+{T_S}^2} \\ &> 1,\end{aligned} \tag{6.28}$$

since $n_2t + n_1 > n_1t + n_2$ when $t > 1$. This indicates the higher speedup is obtained for larger datasets learning.

6.4 Experiments

In the experiments, we evaluate the correctness, efficiency and effectiveness of the proposed algorithms. Our algorithm is also compared with other parallel and parallel incremental algorithms in terms of of classification capability and training time.

Our experiments are conducted on a cluster of 4 computers. Each computer is equipped with 8 GB memory and four 2.8 GHz cores. For MapReduce environment, we have Hadoop version 2.7.2 installed on top of Java 1.8.0_91. The datasets used are downloaded from the public UCI repository [99], where those multi-class datasets are transformed into binary in one-class-against-rest manner since proposed classifiers are binary.

6.4.1 Equivalence to Batch Retraining

To verify proposed parallel and incremental learning equivalence to batch retraining, we compare respectively the learning outcomes of proposed incremental, parallel, and PI integrated algorithm with that of batch retraining. Consider in (6.7) the learning outcome of wESVM is a column vector. We measure the learning outcome differences by calculating the Euclidean distance between two vectors, one from proposed algorithm and the other from batch retraining. Without loss of generality, we simply set the number of incremental stages as 2 and utilize 2 nodes for parallel computing. All learning differences are measured over four datasets with varied scales and dimensionalities, and the results are shown in Table 6.1. As we can see, no learning outcome differences are found among three algorithms for all four datasets. This indicates that proposed incremental, parallel and PI integrated wESVM is identical to batch retraining in terms of final learning outcomes.

Table 6.1: Incremental (Inc), parallel (Par) and PI integrated (PI) wESVM learning outcome differences against that of batch wESVM.

Dataset	# Feature	# Instance	Inc	Par	PI
Heart	75	303	0	0	0
BreastWisconsin	32	569	0	0	0
AnonymousWeb	294	37,711	0	0	0
CoverType	54	581,012	0	0	0

6.4.2 Parallel Efficiency Evaluation

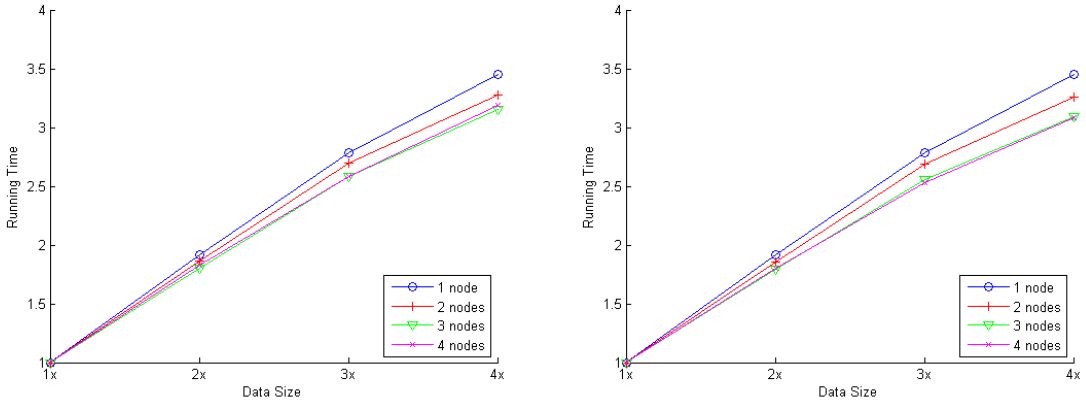
In order to evaluate the efficiency of proposed parallelization, we measure the variation of time cost against the number of nodes utilized and data sizes. We set the number of nodes varies from one to four. For data size, we duplicate the CoverType dataset up to three times to obtain four training datasets in the size of 1×, 2×, 3× and 4× of original dataset, respectively. For each combination of data size and number of nodes, we perform parallel wESVM training for 10 times, and record the time costs.

Consider the MapReduce implementation of our algorithm consists of three steps: Map, Reduce and serial part. For performance evaluation, we adopt from Hadoop JobTrackers [100] four basic measurements: max Map time, mean Map time, max Reduce time and mean Reduce time, where max Map time represents the longest execution time of all Map nodes, mean Map time calculates the average execution time of all Map nodes, and so for max Reduce time and mean Reduce time, respectively. Further for this experiment, we define total execution time as the sum of max Map time, max Reduce time and serial execution time, and define node average time as the sum of mean Map time, mean Reduce time and serial execution time.

Response to Data Size

To evaluate the parallel efficiency of our implementation, we observe the running time variation to data size. We set training time on 1× dataset as the basis, and calculate how many times longer the system costs to learn 2×, 3× and 4× datasets. We plot the curve of the total execution and node average time to data size in Figure 6.2.

As seen from the figure, t-times bigger dataset consumes less than t-times training time no matter how many nodes are used. This agrees with our estimation in (6.27), showing that the running time is sub-linear to data size. Also we can see that the more nodes we use, the fewer times of time spent on learning the same size data. This indicates that learning efficiency of our implementation increases with the number of nodes.



(a) Total execution time

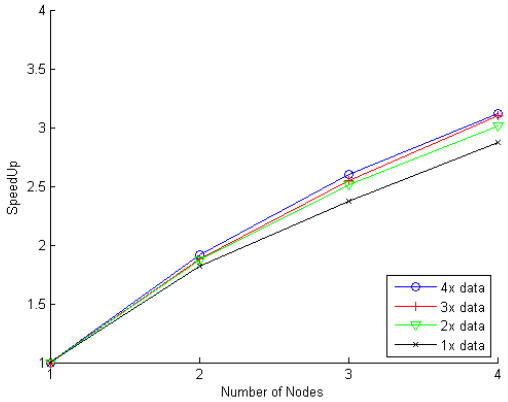
(b) Node average time

Figure 6.2: Running time against data size

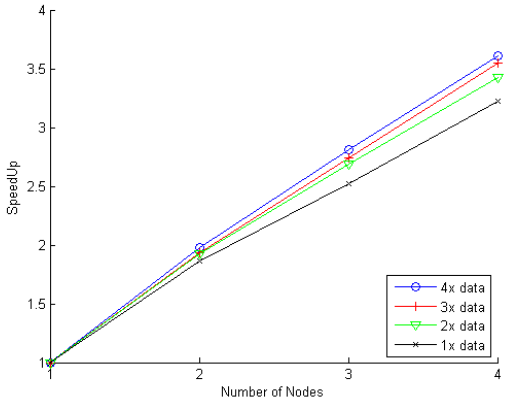
Response to Number of Nodes

To observe system speedup response to number of nodes, we have the curve of speedup to number of nodes shown in Figure 6.3a and 6.3b. As we can see, the training is accelerated by investing more nodes. The speedup is shown sub-linear to the number of nodes, which means t-node gives less than t times speedup. This result again agrees with our estimation in (6.25). Further as predicted in (6.28), the larger dataset is seen giving the better speedup, which indicates proposed algorithm is more sustainable for large dataset learning.

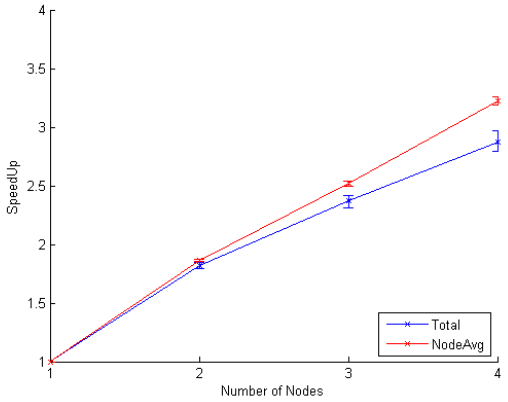
In checking the stability of our Hadoop implementation, we calculate the mean and variation (minimum and maximum) of speedup in terms of total execution time and node average time, respectively. Figure 6.3c to 6.3f give the comparison results. As seen, the speedup in terms of node average time is constantly higher than that of total execution time. This is because total execution time is determined by the time cost of slowest node, not the average node execution time. Also, the above speedup difference is seen increasing with the number of nodes, and so for the speedup variance in terms of total execution time. This indicates that the more nodes in use the more likely to have slow node, and the longer delay is caused by those slow nodes. In other word, our Hadoop implementation is more stable for smaller number of nodes.



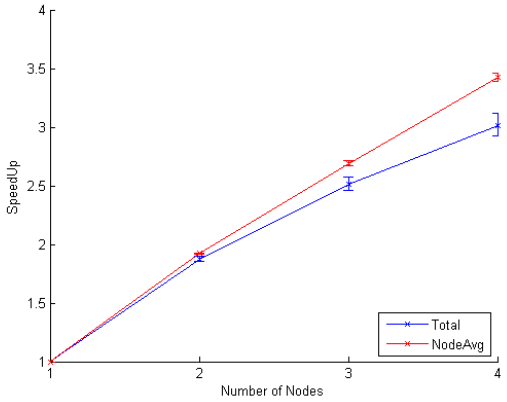
(a) Total execution time



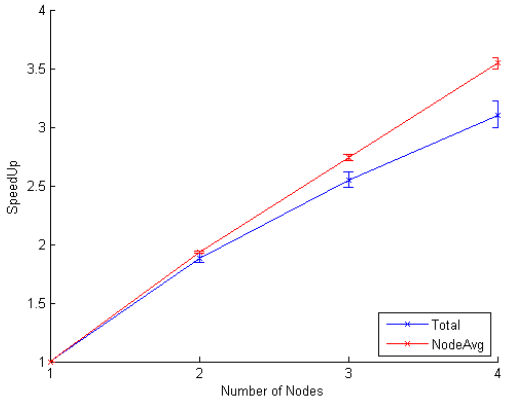
(b) Node average time



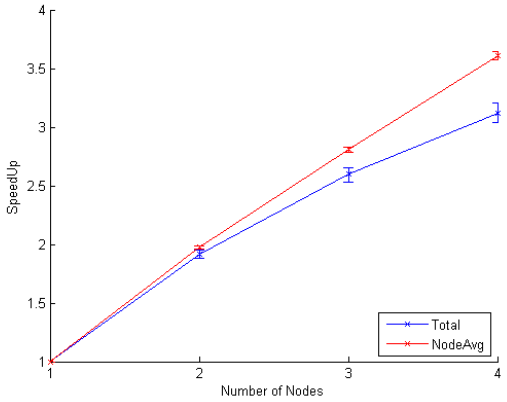
(c) Total vs node average at 1x data



(d) Total vs node average at 2x data

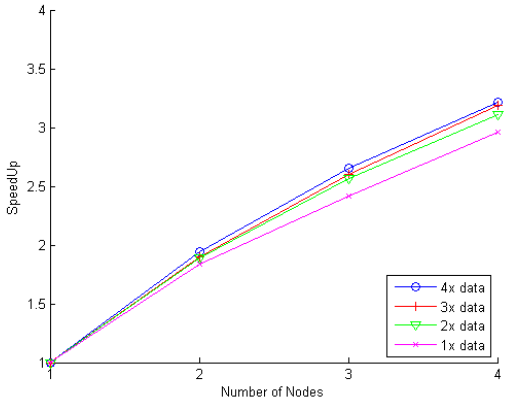


(e) Total vs node average at 3x data

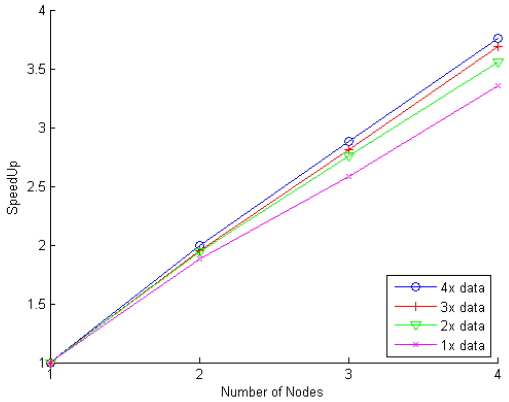


(f) Total vs node average at 4x data

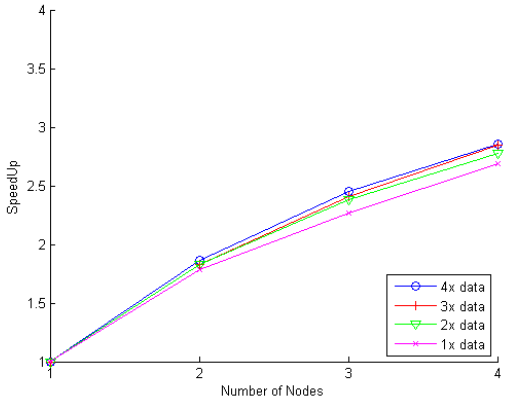
Figure 6.3: Speedup in terms of total execution time and node average time



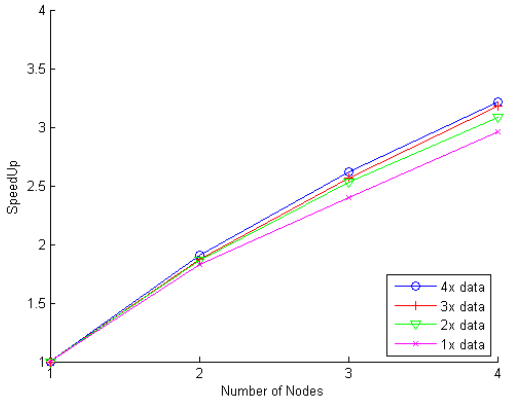
(a) max Map time



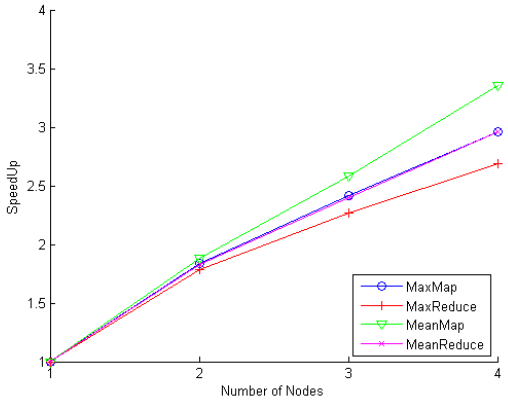
(b) mean Map time



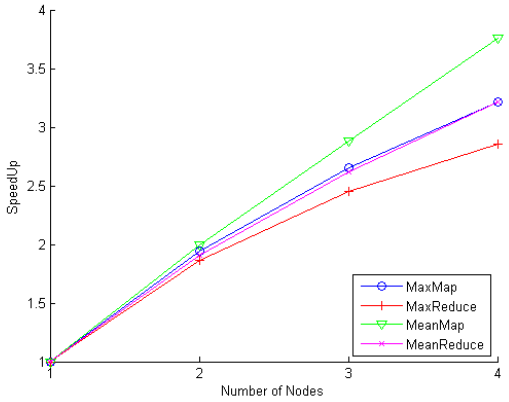
(c) max Reduce time



(d) mean Reduce time

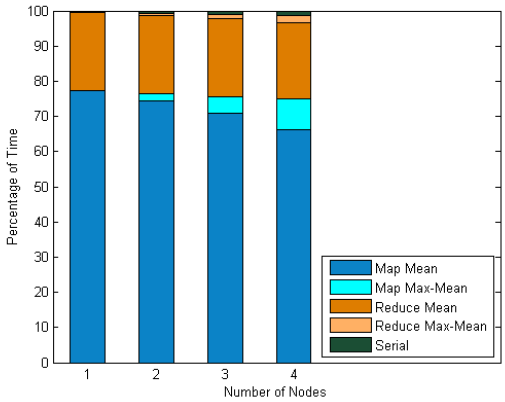


(e) Map and Reduce at 1x data

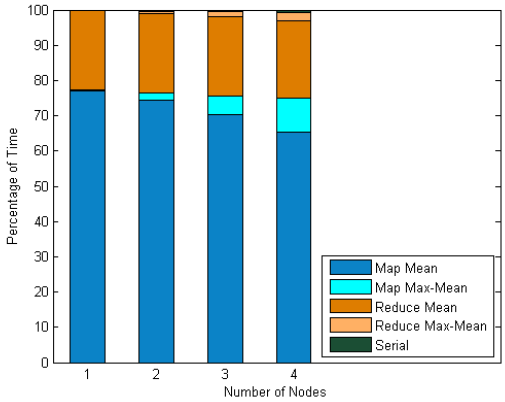


(f) Map and Reduce at 4x data

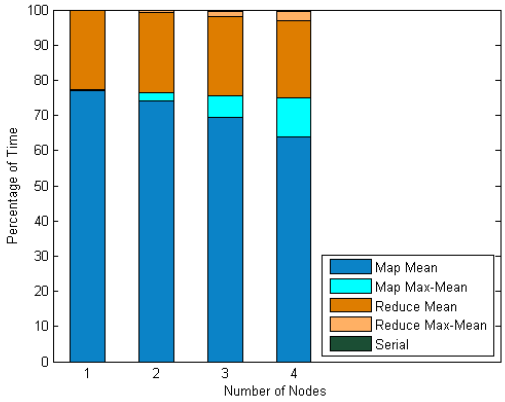
Figure 6.4: Speedup at Map and Reduce phases



(a) 1x data



(b) 2x data



(c) 4x data

Figure 6.5: Percentage of time taken by Map and Reduce at different data sizes

Table 6.2: Training time of batch, incremental (Inc), iarallel (Par) and PI integrated (PI) wESVM algorithm

Stage	# New Instances	# Accumulated Instances	Batch	Inc	Par	PI
1	300 K	300 K	41.35s	41.33s	19.98s	19.96s
2	150 K	450 K	62.55s	21.19s	29.94s	11.03s
3	75 K	525 K	74.63s	11.19s	35.46s	5.47s
4	37.5 K	562.5 K	80.23s	5.58s	37.72s	2.27s

To further discover the speedup contribution of individual phases, we calculate speedup in terms of max Map, mean Map, max Reduce and mean Reduce time, and give the results in Figure 6.4a to 6.4d respectively. As seen from the figures, the speedup of Map and Reduce phase are both sub-linear to number of nodes, which is lower than our estimation in (6.22). This is because that Hadoop introduces extra time cost (beyond computation) on nodes communication and coordination, and this cost increases with the number of nodes utilized. Also, the larger data leads to the better speedup at both Map and Reduce phases. This can be explained that the extra time cost increases sub-linearly to data size, thus less speedup is neutralized for larger size dataset. Figure 6.5 discloses the fraction of total execution time taken by Map and Reduce phase, respectively.

6.4.3 Incremental Effectiveness

In this experiment, we expand the training set progressively by adding a set of samples at every incremental stage, meanwhile record the time consumed by the batch and proposed algorithms. We split CoverType dataset into four chunks, and we add one chunk data at each stage into the training set. Here, parallel and PI integrated wESVM are executed on a cluster of four nodes. Table 6.2 gives the number of samples added and training time costs for each algorithm in comparison.

As seen in Table 6.2, the training time for proposed incremental and PI integrated wESVM decreases consistently when incoming data size reduces over stages. As a reference, batch and parallel algorithms increases over incremental data size because non-incremental algorithms have to train all data arrived so

far. Specifically at stage 4, there are 37.5K samples newly arrived. Proposed incremental and PI integrated wESVM costs 5.58s and 2.27s respectively to learn the data. In contrast, batch and parallel wESVM spends 80.23s and 37.72s for stage 4 learning, which is far more than that of the two incremental algorithms. This is because that batch and parallel algorithms have to learn the total 562.5K instances, which is an accumulation of all incremental stages.

6.4.4 Comparison with Other Algorithms

In this experiment, we compare our PI integrated algorithm (PI) with parallel incremental ESVM (PIESVM) [101], Bagging SVM (Bagging) [102] and Distributed SMO (DSMO) [103] in terms of classification capability and training time.

Classification Capability

The four datasets we used and their characteristics are listed in Table 6.3. For each dataset, we conduct two experiments to test the performance of all algorithms with and without class-imbalance. The classification capability is measured by testing accuracy and G-mean, where G-mean is defined as

$$\text{G-mean} = \sqrt{\text{Sensitivity} \times \text{Specificity}}. \tag{6.29}$$

The results of the two experiments are shown in Tables 6.4 and 6.5 respectively, where performance on each measurement is shown as 'average value $\pm$ standard division'. Table 6.4 shows that when there is no class-imbalance, all four algorithms show similar performance. When there is heavy class-imbalance, in Exp. 2 shown in Table 6.5, the classification capability decreases for all four algorithms, resulting in a lower G-mean. This is due to algorithms tend to classify most samples as the majority class to increase the training accuracy. However, the proposed algorithm has a much smaller performance decrease in comparison with other algorithms, thus it always gives the highest accuracy and G-mean. This indicates the robustness of our algorithm against class-imbalance.

Table 6.3: Datasets description

Dataset	Class Min. / Maj.	#Var.	Exp.	Imbalance Ratio #Maj./#Min.	Training Set		Testing Set	
					#Pos.	#Neg.	#Pos.	#Neg.
BreastWisconsin	Abnormal / Normal	9	1	1	20	20	40	40
			2	20	400	20		
Car	Very Good / Remainder	6	1	1	20	20	40	40
			2	40	800	20		
Abalone	Less than 5 rings/ Remainder	8	1	1	20	20	40	40
			2	60	1200	20		
CoverType	Type 5/remainder	54	1	1	1000	1000	5000	5000
			2	80	80000	1000		
BankNote	True/False	4	1	1	100	100	100	100
			2	20	30	600		
CNAE9	C1/Rest	856	1	1	50	50	50	50
			2	10	50	500		
InternetAd	Ad./Not	1558	1	1	100	100	250	250
			2	16	100	1600		
Cardiotocography	Pathologic/remainder	21	1	1	50	50	100	100
			2	32	50	1600		

Table 6.4: Classification capability without class-imbalance. Exp. 1

Dataset	Measurement	Algorithm			
		PI	PIESVM	Bagging	DSMO
BreastWisconsin	Accuracy	94.46 ± 3.13	94.46 ± 3.13	93.86 ± 5.48	$\mathbf{94.78} \pm 4.43$
	G-mean	$\mathbf{94.72} \pm 3.43$	$\mathbf{94.72} \pm 3.43$	93.12 ± 5.32	94.35 ± 3.92
Car	Accuracy	92.77 ± 5.21	92.77 ± 5.21	$\mathbf{94.55} \pm 2.42$	92.73 ± 3.21
	G-mean	92.33 ± 3.43	92.33 ± 3.43	$\mathbf{94.52} \pm 3.12$	92.65 ± 3.51
Abalone	Accuracy	$\mathbf{95.72} \pm 3.24$	$\mathbf{95.72} \pm 3.24$	94.86 ± 3.32	95.41 ± 3.67
	G-mean	95.46 ± 3.41	95.46 ± 3.41	94.54 ± 3.35	$\mathbf{95.64} \pm 3.43$
CoverType	Accuracy	84.91 ± 5.11	84.91 ± 5.11	$\mathbf{86.70} \pm 6.21$	85.26 ± 6.02
	G-mean	85.21 ± 5.42	85.21 ± 5.42	85.15 ± 6.18	$\mathbf{85.62} \pm 5.98$
BankNote	Accuracy	$\mathbf{97.21} \pm 3.13$	$\mathbf{97.21} \pm 3.13$	96.73 ± 3.22	97.01 ± 3.02
	G-mean	$\mathbf{97.27} \pm 3.22$	$\mathbf{97.27} \pm 3.22$	96.37 ± 3.28	95.98 ± 2.96
CNAE9	Accuracy	97.41 ± 3.21	97.41 ± 3.21	$\mathbf{97.70} \pm 3.31$	95.25 ± 3.02
	G-mean	97.12 ± 3.42	97.12 ± 3.42	97.15 ± 3.20	$\mathbf{97.62} \pm 2.88$
InternetAd	Accuracy	$\mathbf{94.12} \pm 2.10$	$\mathbf{94.12} \pm 2.10$	93.22 ± 3.65	93.75 ± 3.11
	G-mean	$\mathbf{93.97} \pm 2.12$	$\mathbf{93.97} \pm 2.12$	91.55 ± 3.33	93.06 ± 2.97
Cardiotocography	Accuracy	90.42 ± 3.43	90.42 ± 3.43	$\mathbf{91.21} \pm 3.11$	90.21 ± 3.22
	G-mean	90.12 ± 3.42	90.12 ± 3.42	$\mathbf{91.15} \pm 3.12$	90.63 ± 2.97

Table 6.5: Classification capability with class-imbalance. Exp. 2

Dataset	Measurement	Algorithm			
		PI	PIESVM	Bagging	DSMO
BreastWisconsin	Accuracy	**93.71** $\pm$ 4.10	88.07 $\pm$ 5.43	86.83 $\pm$ 5.65	86.03 $\pm$ 4.98
	G-mean	**93.54** $\pm$ 3.93	87.32 $\pm$ 5.32	86.12 $\pm$ 5.51	87.36 $\pm$ 4.88
Car	Accuracy	**89.71** $\pm$ 4.58	59.07 $\pm$ 5.62	66.83 $\pm$ 5.31	76.43 $\pm$ 5.17
	G-mean	**89.54** $\pm$ 4.32	44.27 $\pm$ 4.76	58.01 $\pm$ 5.30	73.04 $\pm$ 4.71
Abalone	Accuracy	**95.02** $\pm$ 3.11	52.72 $\pm$ 3.98	60.33 $\pm$ 5.56	62.19 $\pm$ 4.68
	G-mean	**95.31** $\pm$ 3.22	20.18 $\pm$ 4.21	44.72 $\pm$ 3.31	50.11 $\pm$ 4.12
CoverType	Accuracy	**82.32** $\pm$ 4.58	50.00 $\pm$ 0.00	50.00 $\pm$ 0.00	50.00 $\pm$ 0.00
	G-mean	**82.02** $\pm$ 5.42	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
BankNote	Accuracy	**92.41** $\pm$ 3.55	59.40 $\pm$ 5.12	65.47 $\pm$ 2.03	67.37 $\pm$ 4.02
	G-mean	**92.12** $\pm$ 3.41	44.52 $\pm$ 6.02	55.32 $\pm$ 3.01	59.88 $\pm$ 4.77
CANE9	Accuracy	**96.81** $\pm$ 3.21	50.00 $\pm$ 0.00	50.00 $\pm$ 0.00	50.00 $\pm$ 0.00
	G-mean	**96.47** $\pm$ 3.11	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
InternetAd	Accuracy	**92.13** $\pm$ 2.51	87.03 $\pm$ 3.02	88.75 $\pm$ 3.87	88.64 $\pm$ 2.99
	G-mean	**91.76** $\pm$ 3.01	86.11 $\pm$ 3.12	87.83 $\pm$ 3.23	87.87 $\pm$ 3.04
Cardiotocography	Accuracy	**90.31** $\pm$ 3.58	67.42 $\pm$ 4.03	82.33 $\pm$ 3.11	84.21 $\pm$ 4.28
	G-mean	**90.02** $\pm$ 3.64	59.87 $\pm$ 4.33	80.03 $\pm$ 3.44	83.08 $\pm$ 4.76

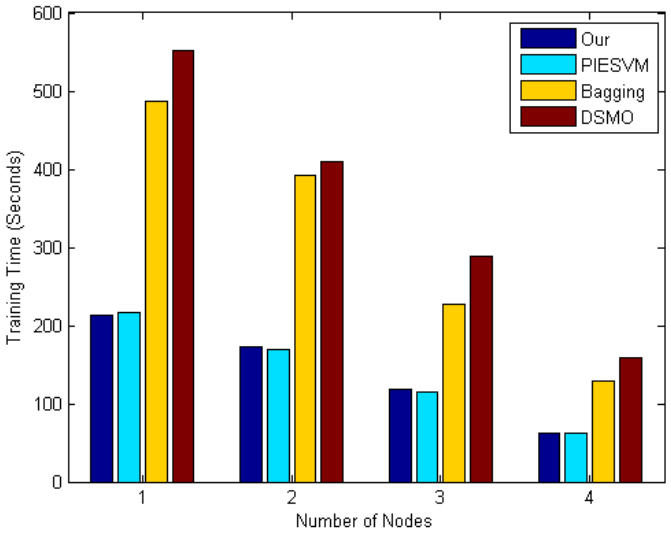


Figure 6.6: Training time comparison

Training Time

In this comparison, all four algorithms learn the 4× CoverType dataset with a different number of nodes involved. The training time is reported in Figure 6.6.

Figure 6.6 shows that our algorithm runs much faster than the Bagging and DSMO algorithms. This can be explained by the simplicity nature of ESVM compared with standard SVM. PIESVM consumes about the same amount of time as our proposed algorithm. This is because these two algorithms share some similar operations and, as we mentioned before, PIESVM can be seen as a special case of our algorithm.

6.5 Summary

In this work, we propose PI integrated wESVM. To enable wESVM knowledge merging, we reformulate wESVM such that knowledge from each class is represented as two class-wised matrices, and each class weight is formulated as a real value coefficient multiplying its corresponding class-wised matrices. As a result, knowledge merging can be performed through simple matrix addition and

class weights are updated easily by renewing the coefficients in response to class-imbalance. Based on this, we developed the PI integrated wESVM, in which incremental learning is achieved via merging knowledge from different incremental stages, and the parallel learning is implemented as merging knowledge from multiple data slices.

We implement the proposed algorithms in MapReduce environment. Experimental results show that proposed algorithms give always the exactly same learning result as batch retraining, our parallel learning scales well in response to both number of nodes and data size, and our incremental learning has clear speed advantage to batch learning. In comparison with other algorithms, the proposed algorithm also show advantages in classification capability and training time.

7 Conclusion and Future Works

7.1 Conclusion

In developing parallel incremental learning for BigData processing, we address first the incremental learning of max-flow in that max-flow has wide applications to the analysis of the Internet and social networks. The challenges of incremental max-flow modeling includes: the graph changes are dynamic, the dilemma of edge delete request and edge occupied by existing flows, and how to handle the cycle flow is another difficulty.

In the proposed incremental max-flow algorithm, all possible graph changes are abstracted into two key changes: edge capacity increase and decrease. To handle edge capacity decrease, we enable the decrease by releasing the capacity occupied by existing flows. When the capacity is occupied by cycle flow, we release the capacity by cancelling the cycle. When the capacity is occupied by $s-t$ flow, we release the capacity by de-augmenting the $s-t$ path. To handle edge capacity increase, we conduct path searching and augmentation to identify the newly emerged flow. The theoretical guarantee of the proposed algorithm is that incremental max-flow is always equal to that of batch retraining.

We apply the proposed incremental max-flow to graph minicuts, an existing batch semi-supervised learning algorithm, and develop the incremental graph minicuts. In batch graph minicuts, the input labeled and unlabeled samples are first represented as a graph, in which each sample is denoted by a node and the similarity of two samples determines the edge capacity between two corresponding nodes. The classification decision is then made by conducting a min-cut, which can be easily computed from the max-flow on the graph. In our incremental graph minicuts, we update the graph dynamically to accommodate sample adding and retiring. Then we employ the proposed incremental max-flow algorithm to learn

the max-flow from the resulting non-stationary graph. In the end, we compute the min-cut from the max-flow to make the classification decision on the updated labeled and unlabeled dataset.

As compared to incremental learning, parallel incremental learning presents the advantage of speed acceleration. The traditional approach achieves this by either developing parallel learning of an incremental learning model, or the other way around, which is developing incremental learning on top of a parallel learning model. Alternatively, we propose the concept of PI integration, in which parallel and incremental learning are merged by interpolating two different types of learning as the same knowledge merging process.

To achieve PI integration, we define a knowledge mergeable condition, which requires the base learning model knowledge from one dataset can be merged with that of another. As such, incremental learning can be implemented by merging knowledge from data that arrived at different learning stages, and parallel learning can be performed by merging knowledge from simultaneous learners on different data slices. For algorithm implementation, we identify a family of algorithms that satisfy the knowledge mergeable condition. These include LPSVM, ESVM, wLPSVM, and wESVM, where wESVM is a type of generalized weighted kernel LPSVM and the remaining algorithms are just special cases of wESVM.

As the first PI integrated learning system, we develop the PI integrated wESVM, where the wESVM is reformulated to enable learning knowledge to be mergable. Because the knowledge from one dataset can be merged with that from another dataset via simple matrix addition, the proposed PI integrated wESVM is capable of conducting parallel incremental learning by continuously merging knowledge from data slices arriving at each incremental stage, and instantly merging knowledge from data slices partitioned for parallel calculation.

7.2 Future Works

To achieve PI integration, the base model is required to satisfy the knowledge mergeable condition. This gives the limitation that PI integration can be only applicable to a set of models whose learning knowledge is mergable among different datasets. So far, the proposed PI integrated wESVM gives an example

implementation of PI integration.

For our future works, we consider:

1. To further explore the knowledge mergeable condition, and discover other base learning models for PI integration. As the wESVM family is not likely to be the only set of knowledge mergeable algorithms; and more interestingly,

2. To relax the constraint of the knowledge mergeable condition and discover other potential mechanisms for PI integration, so that PI integration can be generalized to work among an extensive scope of algorithms.

References

[1] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, pp. 1124–1137, Sept 2004.

[2] S. Hed, "Maximum flows by incremental breadth-first search," Ph.D. dissertation, Tel-Aviv University, 2011.

[3] J. Gama, "A survey on learning from data streams: current and future trends," *Progress in Artificial Intelligence*, vol. 1, no. 1, pp. 45–55, 2012.

[4] S. Muthukrishnan *et al.*, "Data streams: Algorithms and applications," *Foundations and Trends® in Theoretical Computer Science*, vol. 1, no. 2, pp. 117–236, 2005.

[5] P. Joshi and P. Kulkarni, "Incremental learning: Areas and methods-a survey," *International Journal of Data Mining & Knowledge Management Process*, vol. 2, no. 5, p. 43, 2012.

[6] P. Zhang, X. Zhu, J. Tan, and L. Guo, "Classifier and cluster ensembles for mining concept drifting data streams," in *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, Dec 2010, pp. 1175–1180.

[7] R. Elwell and R. Polikar, "Incremental learning of concept drift in non-stationary environments," *IEEE Transactions on Neural Networks*, vol. 22, no. 10, pp. 1517–1531, 2011.

[8] S. R. Upadhyaya, "Parallel approaches to machine learning a comprehensive survey," *J. Parallel Distrib. Comput.*, vol. 73, no. 3, pp. 284–292, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.11.001

[9] A. Gepperth and B. Hammer, "Incremental learning algorithms and applications," in *European Symposium on Artificial Neural Networks (ESANN)*, 2016.

[10] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, "Critical analysis of big data challenges and analytical methods," *Journal of Business Research*, vol. 70, no. Supplement C, pp. 263 – 286, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S014829631630488X

[11] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, no. 2, pp. 137 – 144, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0268401214001066

[12] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[13] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network." *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956. [Online]. Available: http://www.rand.org/pubs/papers/P605/

[14] ——, *Flows in Networks*. Princeton University Press, 1962.

[15] D. Shengwu and Z. Yi, "Research on method of traffic network bottleneck identification based on max-flow min-cut theorem," in *Transportation, Mechanical, and Electrical Engineering (TMEE), 2011 International Conference on*, Dec 2011, pp. 1905–1908.

[16] O. Kosut, "Max-flow min-cut for power system security index computation," in *Sensor Array and Multichannel Signal Processing Workshop (SAM), 2014 IEEE 8th*, June 2014, pp. 61–64.

[17] S. Dandapat, B. Mitra, N. Ganguly, and R. Choudhury, "Fair bandwidth allocation in wireless mobile environment using max-flow," in *High Performance Computing (HiPC), 2010 International Conference on*, Dec 2010, pp. 1–10.

[18] S. Dandapat, B. Mitra, R. Choudhury, and N. Ganguly, "Smart association control in wireless mobile environment using max-flow," *Network and Service Management, IEEE Transactions on*, vol. 9, no. 1, pp. 73–86, March 2012.

[19] Y. Li, Z. Wang, D. Jin, and S. Chen, "Optimal mobile content downloading in device-to-device communication underlaying cellular networks," *IEEE Transactions on Wireless Communications*, vol. 13, no. 7, pp. 3596–3608, July 2014.

[20] Y. Duan, W. Huang, and H. Chang, "Shape prior regularized continuous max-flow approach to image segmentation," in *Pattern Recognition (ICPR), 2012 21st International Conference on*, Nov 2012, pp. 2516–2519.

[21] T. Weglinski and A. Fabijanska, "Min-cut/max-flow segmentation of hydrocephalus in children from ct datasets," in *Signals and Electronic Systems (ICSES), 2012 International Conference on*, Sept 2012, pp. 1–6.

[22] Y. Boykov and G. Funka-Lea, "Graph cuts and efficient nd image segmentation," *International journal of computer vision*, vol. 70, no. 2, pp. 109–131, 2006.

[23] ——, "Optimal object extraction via constrained graph-cuts," *International Journal of Computer Vision (IJCV)*, 2004.

[24] Y. Boykov and M.-P. Jolly, "Interactive graph cuts for optimal boundary amp; region segmentation of objects in n-d images," in *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, vol. 1, 2001, pp. 105–112 vol.1.

[25] N. Lermé, F. Malgouyres, D. Hamoir, and E. Thouin, "Bayesian image restoration for mosaic active imaging," *Inverse Problems and Imaging*, pp. to–appear, 2014.

[26] Y. Boykov, O. Veksler, and R. Zabih, "Markov random fields with efficient approximations," in *Computer vision and pattern recognition, 1998. Proceedings. 1998 IEEE computer society conference on.* IEEE, 1998, pp. 648–655.

[27] R. Shade and P. Newman, "Choosing where to go: Complete 3d exploration with stereo," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 2806–2811.

[28] H. Isack and Y. Boykov, "Energy based multi-model fitting amp;amp; matching for 3d reconstruction," in *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, June 2014, pp. 1146–1153.

[29] D. M. Greig, B. T. Porteous, and A. H. Seheult, "Exact maximum a posteriori estimation for binary images," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 271–279, 1989.

[30] D. Snow, P. Viola, and R. Zabih, "Exact voxel occupancy with graph cuts," in *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, vol. 1, 2000, pp. 345–352 vol.1.

[31] S. Paris, F. X. Sillion, and L. Quan, "A surface reconstruction method using global graph cut optimization," *International Journal of Computer Vision*, vol. 66, no. 2, pp. 141–161, 2006.

[32] T. Yu, N. Ahuja, and W.-C. Chen, "Sdg cut: 3d reconstruction of non-lambertian objects using graph cuts on surface distance grid," in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 2. IEEE, 2006, pp. 2269–2276.

[33] P. Labatut, J.-P. Pons, and R. Keriven, "Efficient multi-view reconstruction of large-scale scenes using interest points, delaunay triangulation and graph cuts," in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on.* IEEE, 2007, pp. 1–8.

[34] Y. Boykov and D. P. Huttenlocher, "A new bayesian framework for object recognition," in *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, vol. 2. IEEE, 1999, pp. 517–523.

[35] A. Suga, K. Fukuda, T. Takiguchi, and Y. Ariki, "Object recognition and segmentation using sift and graph cuts," in *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on.* IEEE, 2008, pp. 1–4.

[36] L. Ladicky, C. Russell, P. Kohli, and P. H. Torr, "Graph cut based inference with co-occurrence statistics," in *European Conference on Computer Vision*. Springer, 2010, pp. 239–253.

[37] B. Thirion, B. Bascle, V. Ramesh, and N. Navab, "Fusion of color, shading and boundary information for factory pipe segmentation," in *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, vol. 2. IEEE, 2000, pp. 349–356.

[38] J. Mooser, S. You, and U. Neumann, "Real-time object tracking for augmented reality combining graph cuts and optical flow," in *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*. IEEE Computer Society, 2007, pp. 1–8.

[39] Y. Tian, T. Guan, and C. Wang, "Real-time occlusion handling in augmented reality based on an object tracking approach," *Sensors*, vol. 10, no. 4, pp. 2885–2900, 2010.

[40] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, "Graphcut textures: image and video synthesis using graph cuts," in *ACM Transactions on Graphics (ToG)*, vol. 22, no. 3. ACM, 2003, pp. 277–286.

[41] N. Gracias, M. Mahoor, S. Negahdaripour, and A. Gleason, "Fast image blending using watersheds and graph cuts," *Image and Vision Computing*, vol. 27, no. 5, pp. 597–607, 2009.

[42] V. Kolmogorov and R. Zabih, "Computing visual correspondence with occlusions using graph cuts," in *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, vol. 2. IEEE, 2001, pp. 508–515.

[43] ——, "Multi-camera scene reconstruction via graph cuts," *Computer Vision—ECCV 2002*, pp. 8–40, 2002.

[44] A. Raj and C. H. Wiggins, "An information-theoretic derivation of min-cut-based clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 6, pp. 988–995, June 2010.

[45] N. Tishby, F. C. Pereira, and W. Bialek, "The information bottleneck method," *arXiv preprint physics/0004057*, 2000.

[46] N. Tishby and N. Slonim, "Data clustering by markovian relaxation and the information bottleneck method," in *Advances in neural information processing systems*, 2001, pp. 640–646.

[47] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[48] N. Slonim, "The information bottleneck: Theory and applications," Ph.D. dissertation, Hebrew University of Jerusalem, 2002.

[49] N. Slonim and N. Tishby, "Document clustering using word clusters via the information bottleneck method," in *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2000, pp. 208–215.

[50] E. Ziv, M. Middendorf, and C. H. Wiggins, "Information-theoretic approach to network modularity," *Physical Review E*, vol. 71, no. 4, p. 046117, 2005.

[51] N. Slonim, R. Somerville, N. Tishby, and O. Lahav, "Objective classification of galaxy spectra using the information bottleneck method," *Monthly Notices of the Royal Astronomical Society*, vol. 323, no. 2, pp. 270–284, 2001.

[52] Z. Mu, M. Fanrong, and Z. Yong, "Density-based link clustering algorithm for overlapping community detection," *Journal of Computer Research and Development*, vol. 12, p. 006, 2013.

[53] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos, "Community detection in social media," *Data Mining and Knowledge Discovery*, vol. 24, no. 3, pp. 515–554, 2012.

[54] Y. Ruan, D. Fuhry, and S. Parthasarathy, "Efficient community detection in large networks using content and links," in *Proceedings of the 22nd international conference on World Wide Web*. ACM, 2013, pp. 1089–1098.

[55] M. Grundmann, V. Kwatra, M. Han, and I. Essa, "Efficient hierarchical graph-based video segmentation," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 2010, pp. 2141–2148.

[56] B. Saha and P. Mitra, "Fast incremental minimum-cut based algorithm for graph clustering," in *Proc. ICDM*, 2006, pp. 207–211.

[57] Y. Zhou, H. Cheng, and J. X. Yu, "Clustering large attributed graphs: An efficient incremental approach," in *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 689–698.

[58] A. V. Goldberg, S. Hed, H. Kaplan, P. Kohli, R. E. Tarjan, and R. F. Werneck, "Faster and more dynamic maximum flow by incremental breadth-first search," vol. 9294. Sorubger, September 2015, pp. 619–630. [Online]. Available: https://www.microsoft.com/en-us/research/publication/faster-dynamic-maximum-flow-incremental-breadth-first-search/

[59] A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck, "Maximum flows by incremental breadth-first search," in *Proceedings of the 19th European Conference on Algorithms*, ser. ESA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 457–468. [Online]. Available: http://dl.acm.org/citation.cfm?id=2040572.2040623

[60] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, Oct. 1988. [Online]. Available: http://doi.acm.org/10.1145/48014.61051

[61] S. Kumar and P. Gupta, "An incremental algorithm for the maximum flow problem," *Journal of Mathematical Modelling and Algorithms*, vol. 2, no. 1, pp. 1–16, Mar 2003. [Online]. Available: https://doi.org/10.1023/A:1023607406540

[62] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[63] S. Kumar and P. Gupta, "An incremental algorithm for the maximum flow problem," *Journal of Mathematical Modelling and Algorithms*, vol. 2, no. 1, pp. 1–16, 2003.

[64] P. Kumar Mallapragada, R. Jin, A. Jain, and Y. Liu, "Semiboost: Boosting for semi-supervised learning," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 31, no. 11, pp. 2000–2014, Nov 2009.

[65] O. Chapelle, B. Schlkopf, and A. Zien, *Semi-Supervised Learning*. The MIT Press, 2006.

[66] I. Cohen, "Semisupervised learning of classifiers with application to human-computer interaction," Ph.D. dissertation, Champaign, IL, USA, 2003, aAI3101819.

[67] J. Ortigosa-Hernández, I. Inza, and J. A. Lozano, "On the optimal usage of labelled examples in semi-supervised multi-class classification problems," UPV/EHU, Tech. Rep., April 2015. [Online]. Available: https://addi.ehu.es/handle/10810/15004

[68] P. Zhang, X. Zhu, and L. Guo, "Mining data streams with labeled and unlabeled training examples," in *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, 2009, pp. 627–636.

[69] G. Ditzler and R. Polikar, "Semi-supervised learning in nonstationary environments," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, July 2011, pp. 2741–2748.

[70] G. Carneiro and J. Nascimento, "Incremental on-line semi-supervised learning for segmenting the left ventricle of the heart from ultrasound data," in *Computer Vision (ICCV), 2011 IEEE International Conference on*, Nov 2011, pp. 1700–1707.

[71] B. Yver, "Online semi-supervised learning: Application to dynamic learning from radar data," in *Radar Conference - Surveillance for a Safer World, 2009. RADAR. International*, Oct 2009, pp. 1–6.

[72] S. Chu, S. Narayanan, and C.-C. Kuo, "A semi-supervised learning approach to online audio background detection," in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, April 2009, pp. 1629–1632.

[73] A. Bamdadian, C. Guan, K. K. Ang, and J. Xu, "Online semi-supervised learning with kl distance weighting for motor imagery-based bci," in *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, Aug 2012, pp. 2732–2735.

[74] K. K. Ang, Z. Y. Chin, H. Zhang, and C. Guan, "Filter bank common spatial pattern (fbcsp) algorithm using online adaptive and semi-supervised learning," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, July 2011, pp. 392–396.

[75] S. Imangaliyev, B. Keijser, W. Crielaard, and E. Tsivtsivadze, "Online semi-supervised learning: Algorithm and application in metagenomics," in *Bioinformatics and Biomedicine (BIBM), 2013 IEEE International Conference on*, Dec 2013, pp. 521–525.

[76] A. Blum and S. Chawla, "Learning from labeled and unlabeled data using graph mincuts," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 19–26.

[77] K. I. Tsianos and M. G. Rabbat, "Efficient distributed online prediction and stochastic optimization with approximate distributed averaging," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 2, no. 4, pp. 489–506, Dec 2016.

[78] J.-H. Böse, A. Andrzejak, and M. Högqvist, "Beyond online aggregation: Parallel and incremental data mining with online map-reduce," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, ser. MDAC '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/1779599.1779602

[79] K. Yue, Q. Fang, X. Wang, J. Li, and W. Liu, "A parallel and incremental approach for data-intensive learning of bayesian networks," *IEEE Transactions on Cybernetics*, vol. 45, no. 12, pp. 2890–2904, Dec 2015.

[80] X. Zhao, X. Li, Z. Zhang, C. Shen, Y. Zhuang, L. Gao, and X. Li, "Scalable linear visual feature learning via online parallel nonnegative matrix factorization," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 12, pp. 2628–2642, Dec 2016.

[81] T. N. Doan, T. N. Do, and F. Poulet, "Parallel incremental svm for classifying million images with very high-dimensional signatures into thousand classes," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, Aug 2013, pp. 1–8.

[82] J. S. Yoo and D. Boulware, "Incremental and parallel spatial association mining," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 75–76.

[83] K. Chen and Q. Huo, "Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2016, pp. 5880–5884.

[84] L. Xu and Z. Yun, "A novel parallel algorithm for frequent itemset mining of incremental dataset," in *2015 2nd International Conference on Information Science and Control Engineering*, April 2015, pp. 41–44.

[85] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Berlin, Germany: Springer-Verlag, 1995.

[86] T. Yamasaki and K. Ikeda, "Incremental svms and their geometrical analyses," in *Neural Networks and Brain, 2005. ICNN B '05. International Conference on*, vol. 3, oct. 2005, pp. 1734 –1738.

[87] G. Cauwenberghs and T. Poggio, "Incremental and decremental support vector machine learning," in *NIPS'00*, 2000, pp. 409–415.

[88] M. Karasuyama and I. Takeuchi, "Multiple incremental decremental learning of support vector machines," *Neural Networks, IEEE Transactions on*, vol. 21, no. 7, pp. 1048 –1059, july 2010.

[89] G. Fung and O. L. Mangasarian, "Proximal support vector machine classifiers," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, San Francisco, California, 2001, pp. 77–86.

[90] Q. Liu, Q. He, and Z. Shi, "Extreme support vector machine classifier," in *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 2008, pp. 222–233.

[91] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, vol. 2, July 2004, pp. 985–990 vol.2.

[92] G. B. Huang, Q. Y. Zhu, and C. K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1–3, pp. 489 – 501, 2006. [Online]. Available: //www.sciencedirect.com/science/article/pii/S0925231206000385

[93] G. M. Fung and O. L. Mangasarian, "Multicategory proximal support vector machine classifiers," *Mach. Learn.*, vol. 59, pp. 77–97, 2005.

[94] D. Zhuang, B. Zhang, Q. Yang, J. Yan, Z. Chen, and Y. Chen, "Efficient text classification by weighted proximal svm," in *Data Mining, Fifth IEEE International Conference on*, nov. 2005, p. 8 pp.

[95] X. Tao and H. Ji, "A modified psvm and its application to unbalanced data classification," in *Natural Computation, 2007. ICNC 2007. Third International Conference on*, vol. 1, aug. 2007, pp. 488 –490.

[96] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng, "Map-reduce for machine learning on multicore," in *Advances in Neural Information Processing Systems 19*,

P. B. Schölkopf, J. C. Platt, and T. Hoffman, Eds. MIT Press, 2006, pp. 281–288. [Online]. Available: http://papers.nips.cc/paper/3150-map-reduce-for-machine-learning-on-multicore.pdf

[97] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[98] R. Lammel, "Google's MapReduce Programming Model – Revisited," *Science of Computer Programming*, 2008.

[99] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[100] M. R. Ghazi and D. Gangodkar, "Hadoop, mapreduce and hdfs: A developers perspective," *Procedia Computer Science*, vol. 48, pp. 45 – 50, 2015, international Conference on Computer, Communication and Convergence (ICCC 2015). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915006171

[101] Q. He, C. Du, Q. Wang, F. Zhuang, and Z. Shi, "A parallel incremental extreme svm classifier," *Neurocomputing*, vol. 74, no. 16, pp. 2532 – 2540, 2011, advances in Extreme Learning Machine: Theory and Applications Biological Inspired Systems. Computational and Ambient Intelligence. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231211002566

[102] J. Zhao, Z. Liang, and Y. Yang, "Parallelized incremental support vector machines based on mapreduce and bagging technique," in *2012 IEEE International Conference on Information Science and Technology*, March 2012, pp. 297–301.

[103] G. Caruana, M. Li, and M. Qi, "A mapreduce based parallel svm for large scale spam filtering," in *2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, vol. 4, July 2011, pp. 2659–2662.

Publication List

Journal Paper

1. **Lei Zhu**, Kazushi Ikeda, Shaoning Pang, Tao Ban and Abdolhossein Sarrafzade, "Merging Weighted SVMs for Parallel Incremental Learning", accepted by *Neural Networks*. *(related to Chapter 6)*

2. Shaoning Pang, Dan Komosny, **Lei Zhu**, Ruibin Zhang, Abdolhossein Sarrafzadeh, Tao Ban, and Daisuke Inoue, "Malicious events grouping via behavior based darknet traffic flow analysis," *Wireless Personal Communications*, vol. 96, no.4, pp. 5335–5353, Oct 2017.

3. **Lei Zhu**, Shaoning Pang, Abdolhossein Sarrafzadeh, Tao Ban, and Daisuke Inoue, "Incremental and decremental max-flow for online semi-supervised learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 8, pp. 2115–2127, Aug 2016. *(related to Chapter 3 and 4)*

4. Shaoning Pang, **Lei Zhu**, Gang Chen, Abdolhossein Sarrafzadeh, Tao Ban, and Daisuke Inoue, "Dynamic class imbalance learning for incremental LPSVM," *Neural Networks*, vol. 44, no. Supplement C, pp. 87 – 100, 2013. *(related to Chapter 6)*

Conference Paper

1. Shaoning Pang, **Lei Zhu**, Tao Ban, Kazushi Ikeda, Wangfei Zhang and Abdolhossein Sarrafzadeh, "Online Max-flow Learning via Augmenting and De-augmenting Path", submitted to *International Joint Conference on Neural Networks (IJCNN)*, under review. *(related to Chapter 3)*

2. Tao Ban, **Lei Zhu**, Jumpei Shimamura, and Koji Nakao, "Detection of Botnet Activities Through the Lens of a Large-Scale Darknet" in *Proceedings of 24th International Conference on Neural Information Processing (ICONIP)*, 2017, pp. 442-451.

3. **Lei Zhu**, Tao Ban, Kazushi Ikeda, Paul Pang, and Abdolhossein Sarrafzadeh, "Distributed incremental wLPSVM learning." in *Proceedings of*

2016 IEEE Symposium Series on Computational Intelligence (SSCI), Dec 2016, pp. 1–8. *(related to Chapter 6)*

4. **Lei Zhu**, Kazushi Ikeda, Paul Pang, Ruibin Zhang, and Abdolhossein Sarrafzadeh, "A Brief Review of Spin-Glass Applications in Unsupervised and Semi-supervised Learning." in *Proceedings of 23rd International Conference on Neural Information Processing (ICONIP)*, 2016, pp. 579–586.

5. Ruibin Zhang, **Lei Zhu**, Xiaosong Li, Shaoning Pang, Abdolhossein Sarrafzadeh, and Dan Komosny, "Behavior Based Darknet Traffic Decomposition for Malicious Events Identification." in *Proceedings of 22nd International Conference on Neural Information Processing (ICONIP)*, 2015, pp. 251–260.

6. Simon Dacey, Lei Song, **Lei Zhu**, and Shaoning Pang, "Analysis and Configuration of Boundary Difference Calculations." in *Proceedings of 21st International Conference on Neural Information Processing (ICONIP)*, 2014, pp. 333–340.

7. Tao Ban, **Lei Zhu**, Junpei Shimamura, Shaoning Pang, Daisuke Inoue, and Koji Nakao, "Behavior Analysis of Long-term Cyber Attacks in the Darknet." in *Proceedings of 19th International Conference on Neural Information Processing (ICONIP)*, 2012, pp. 620–628.

8. **Lei Zhu**, Shaoning Pang, Gang Chen, and Abdolhossein Sarrafzadeh, "Class imbalance robust incremental LPSVM for data streams learning," in *Proceedings of The 2012 International Joint Conference on Neural Networks (IJCNN)*, June 2012, pp. 1–8. *(related to Chapter 6)*