Flexible Anonymization For Privacy Preserving Data Publishing: A Systematic Search Based Approach

Bijit Hore, Ravi Chandra Jammalamadaka, Sharad Mehrotra

April 5, 2007

Abstract

k-anonymity is a popular measure of privacy for data publishing: It measures the risk of identity disclosure for individuals whose personal information is released in some table along with other individuals (e.g. census data). Higher values of k denote higher level of privacy (smaller risk of disclosure). Existing techniques to achieve k-anonymity use a variety of "generalization" and "suppression" of cell values for multi-attribute data. But maximizing the privacy level is not the only goal, the released data needs to be as "information-rich" as possible to maximize its utility. Information loss becomes an even greater concern as more stringent privacy constraints are imposed on the released data [4]. The resulting optimization problems have proven to be computationally intensive for data sets with large attribute-domains. In this paper, we develop a systematic enumeration based search strategy that explores a much richer space of solutions than any previous method in literature. We also develop a flexible generic search strategy that accelerate the search process significantly. We provided sufficient empirical evidence of the superiority of our approach over previous ones through extensive experimentation.

1 Introduction

The problem of anonymizing multi-attribute personal data (*microdata*) for public release, has generated a lot of interest in the research community over the recent years. Privacy concerns arise due to danger of disclosure of confidential information when individual specific data sets are released to public. Since most data sets of interest contain one or more attributes which could be considered confidential by the owner, e.g., medical condition, financial information, ethnicity, level of education etc., if data sets are "insufficiently anonymized", they may be linked with other available information, thereby disclosing identity of individuals and possibly their confidential information. A simple approach to preventing information leakage is to k-anonymize the

released data set [8, 6]. A k-anonymized data set has the following properties: (i) All attributes that are explicitly identifying, example "Name", "Address", "Social security number" etc. are removed from the released set. (ii) A subset of the remaining attributes is determined (by an expert) to be the set of "Quasi Identifiers" (potentially identifying set): these attribute values taken together might form an unique combination and therefore could be linked with external data to identify the record of an individual in the published data. (iii) It contains one or more *sensitive* attributes, the association of which with an individual could be considered confidential, e.g., type of disease, financial details like credit worthiness etc. Now, assume there are q attributes in the quasi identifier set, then the goal of k-anonymization is to "minimally" modify the released data set, such that for each record, the q-tuple of values of its quasi-identifier attribute set is indistinguishable from at least $k-1$ other records in the released set. This is referred to as the anonymity set or equivalence class of the record.

The most commonly used technique for anonymization is that of *generalization*. Generalization refers to the action of replacing the original value by some more general value (e.g., replacing an exact numerical value by an interval). Occasionally some of the records might to be suppressed (i.e., dropped from the published table). Generalizing a data set results in *information loss* which can be captured quantitatively by an associated *cost* metric for each of these operations. Typically, the cost associated with suppression is higher than generalization, since it leads to a greater loss of information. A number of cost metrics have been proposed in literature [13, 1, 2, 4] which capture various notions of information loss, ranging from application specific measures to very generic ones.

Generalization techniques in anonymization can be classified into two categories [3]:

Single dimensional techniques: They are also referred to as *Attribute generalization* (AG) techniques. In such approaches, we fix one generalization for each attribute, e.g. for attributes with ordered domains this corresponds to some division of the domain into disjoint intervals. Then, recode the values of each data tuple according to the generalization fixed for the corresponding attribute. Such a strategy can be viewed as partitioning the multidimensional space of quasi-identifiers into a (possibly non-uniform) grid (see figure 1 (a) below) where each partition boundary cuts across the whole space. Generalization techniques are often mixed with *suppression* (i.e., records are completely dropped from the table) when good generalization is not possible. However, suppressions is not desirable due to the large information loss it results in.

Multidimensional partitioning techniques: In this approach more flexible generalization schemes of the data as compared to those allowed by single dimensional or attribute generalization approaches. Here, the rectangular partitions generated to generalize the data could belong to the

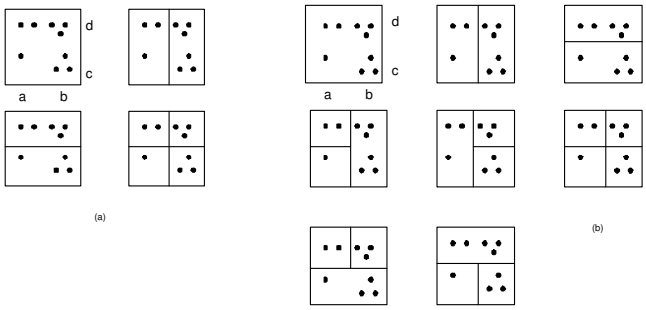

Figure 1: Solution space explored by (a) single dimensional (b) multidimensional schemes

family of *hierarchical* partitionings (also called *guillotine* partitionings) or *arbitrary rectangular partitionings*. Visualizing in a multidimensional space, a hierarchical partitioning of the space is generated by splitting the original (undivided) space into two disjoint rectangular subspaces using an available split along some attribute value (i.e., along a chosen dimension) and then recursively partitioning these two spaces, independent of each other. In such a strategy a partition boundary might not cut across the entire space (see figure 1 (b)). The class of arbitrary partitioning allows for any rectangular partitioning of the space and is a superset of the class of single dimensional and hierarchical partitions.

Our Approach: In this paper we will investigate the class of hierarchical partitioning in great depth. As argued in [3] there are numerous advantages of considering a generalization scheme such as the hierarchical partitioning scheme. The most important advantage of such a multidimensional approach is that it allows one to change the nature of partitioning in one region of the space, i.e. locally without affecting the partitions in the other regions. Another advantage of this localization property is that it helps avoid suppression in many instances where single dimensional schemes would have enforced suppression. For instance, consider the generalizations of the 9 data points in figure 1 (a) and (b). If the required anonymization level was $k = 3$, the latter scheme allows a generalization where each class has size exactly 3 and there is no suppression at all (5^{th} scheme) whereas the former is unable to do so[1]. These extra degrees of freedom in multidimensional schemes substantially reduces the information loss in the final anonymized data sets.

Specifically, we develop a enumeration based branch and bound approach that enables us to explore the space of hierarchical partitionings of a multidimensional space. One of the key challenges is how to enumerate all distinct partitionings without duplication. We devise a novel algorithm to enumerate all hierarchical partitionings in the form of an enumeration tree.

[1]This is optimum with respect to the "discrimination metric" (DM) presented later

We are not aware of any prior work on systematic enumeration[2] of this class of partitionings. Further, based on our enumeration tree, we develop a branch and bound method that exploits monotonicity of the cost functions for lower bounding and pruning of un-promising sets of solutions from the search space. We summarize the key contributions and advantages of our proposed solution methodology below:

Versatility of search: Our search strategy is not tied to a specific optimization problem, but is applicable to a much larger class of problem settings. For instance, recently privacy researchers have started addressing the inadequacies of the basic k-anonymity criteria and its effectiveness as a privacy constraint. Authors in [4] have recently proposed more robust measures like *entropy l-diversity*, *(c,l)-recursive diversity* etc. Even in the traditional settings of the k-anonymization problem, an user may want to impose additional privacy constraints, such as *length restrictions* of anonymity groups along certain attribute dimensions. For instance, the following restriction could be imposed "In the released data set, no individual's salary should be specified to an interval of length smaller than 10K" and/or "Age of any individual should not be specified to an interval less than 5 years" etc. Our solution approach can be seamlessly extended to these and many other new problem settings where the cost functions and problem constraints satisfy some generic monotonicity properties[3](to be discussed in section 4). Indeed, we will show the performance characteristics of our algorithm for a variety of such problem settings.

Progressive improvement in solution quality: Our priority queue based algorithm, at all time maintains a lower bound to the true optimum (global minimum) cost. This feature allows one to stop the algorithm as soon as a solution with cost within an user-specified approximation factor is reached. As the search progresses, better approximations to the optimum solution are generated as both, the solution costs are lowered and the lower bound estimates are tightened. Of course, if let to run till completion, the algorithm does a complete search of the solution space and guarantees finding all solutions with optimum cost.

Incorporation of search heuristics: The priority based approach has an added flexibility, it allows for variety of application driven search strategies/heuristics to be incorporated into the algorithm to enable one to find a good solution early. An early detection of a good solution benefits in two ways, first, it helps establish a good lower bound for pruning purposes and secondly it helps reach desired level of approximations quickly. We introduce a few of these heuristics in section 5 and report our findings in the experimental section.

[2]A scheme to enumerate each distinct partition exactly once.

[3]Most popular cost measures and constraints occurring in optimization problems related to anonymization satisfy these properties.

The overall techniques we develop provides for a powerful approach to exploring the enormous space of hierarchical partitioning based solutions for k-anonymization.

In the remainder of this paper, we first summarize some related work in section 2. In section 3 we present our enumeration algorithm, followed by description of the pruning based search strategy in section 4. In section 5 we describe how one can explore the large space of hierarchical partitions efficiently using a priority queue. Then, in section 6 we discuss experimental results and finally conclude in section 7.

2 Related Work

Here, we briefly summarize the related previous work and compare and contrast it with ours.

Single dimensional (AG) schemes: Most of the previous work on k-anonymization of microdata for public release falls in the category of single dimensional techniques (defined earlier) [6, 8, 2, 1, 11, 10, 14]. While some of these algorithms are designed to search for an optimal solution or some good approximations to the optimal [8, 6], others settle for "acceptable" levels of information loss (i.e. solutions whose cost is within some pre-specified upper bound) by using incomplete stochastic searches [1, 12]. These algorithms have been demonstrated to work only on data with small set of attribute values. Recently, more thorough search algorithms have been proposed for carrying out the search for optimal k-anonymization [2, 11]. Authors in [11] developed an algorithm to enumerate all possible k-anonymizations of a data set using full-domain generalization. This algorithm requires one to provide the domain-generalization hierarchies for each of its quasi identifier attributes. Authors in [2] made a clever observation wherein they noted that most attribute generalization techniques can be modelled as a set enumeration problem. This fact is used by the authors to develop an enumeration based branch and bound algorithm which uses cost-based pruning heuristics to cut down the search space. The effectiveness of the approach was shown empirically by finding actual optimal k-anonymization of a real life data set that attained global minima for a couple of generic cost metrics. We would like to point it out that, though both ours and the scheme proposed in [2] belong to the generic class of branch and bound algorithms, there is little that is common between the two approaches. While authors in [2] use a standard set enumeration technique, such a technique cannot be used to generate the solution space of hierarchical partitions. We instead develop a new geometric approach to enumerate a completely different class of space partitionings.

Multidimensional schemes: While in [2] the results empirically found using the branch and bound method were essentially optimal for single dimensional schemes, authors in [3] demonstrated that it was possible to lower

the cost further (for a specific cost metric) by allowing a multidimensional partitioning of the data (defined earlier). They use a greedy heuristic similar to a kd-tree construction algorithm [5] to partition the data set and derive the anonymity classes. But the work in [3] left a lot of questions un-answered and had some serious limitations mainly due to all the avenues that were left un-explored both via more extensive experimentation and theoretical deduction. We mention a couple of these here: (i) The worst-case approximation factor of the greedy algorithm does not provide a good clue as to how the quality of the solution actually varies with various changing parameters like the anonymity level k, the data dimensionality d etc. (ii) The performance of greedy approaches was not sufficiently explored (empirically or otherwise) beyond the basic k-anonymization problem and essentially one generic cost metric (i.e., DM metric). In general, in this paper we advance the state-of-the-art regarding the application of the multidimensional approach to a much more diverse class of anonymization problems and characterize its performance in a more comprehensive manner through extensive experimentations.

Other definitions of privacy: The weakness of the k-anonymization criteria is that the data set does not guarantee enough diversity in the sensitive attributes within each equivalence set, thereby potentially exposing the confidential attribute of all individuals in the set. To amend this problem, a more robust definition of privacy was proposed recently in [4], which is called the principle of *l-diversity*. The l-diversity criteria requires the data anonymization to be such that each equivalence class has at least l "well represented" distinct values of the sensitive attributes, thereby making it more difficult for an adversary to determine the true value.

3 Enumerating Partitions

In this section we describe the enumeration algorithm for hierarchical partitions of the space. From here onwards, for notational convenience we will use the term "partition" and "partitioning" interchangeably to refer to a complete partitioning scheme of the space. To refer to an undivided block of space within a partitioning scheme, we will use the term "partition block" or simply the term "block".

Let us consider a d-dimensional space where each dimension corresponds to one attribute of the data set. For instance, the figure 2 shows a 2-dimensional space where one dimension is *Age* and the other *Salary* with domains [30yrs, 60yrs] and [50K, 200K] respectively. Let the split set comprise of the four splits: (1):Age=40, (2):Age = 50, (3):Salary = 100K and (4):Salary = 150K. The figure 2 shows the finest level of partition (where all splits are used and all of them cut across the whole space). A hierarchical partition of a d-dimensional space consists of a set of disjoint d-dimensional hyper-rectangles that cover the entire space. Such a partition can be gener-

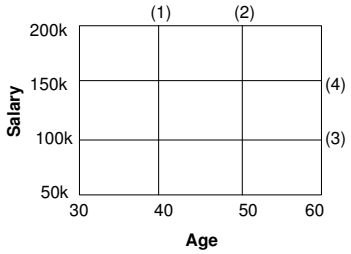

Figure 2: Finest partition of the space

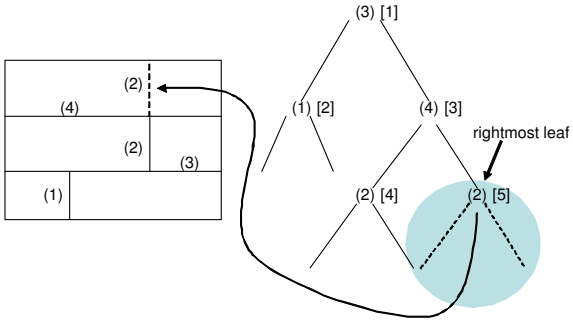

Figure 3: A sequence of splits generating a partition

ated by recursively splitting the space (we will give formal definition a little later). Such partitionings can be represented by binary trees, see figure 3 for example.

The objective of our enumeration algorithm is to generate the set of all possible hierarchical partitions. The difficulty arises in trying to generate it in duplicate free manner. To see the challenge of duplicates, consider the figure 3 again. The binary tree in the figure could have been created by the following sequence of splits: (3), (1), (4), (2) and (2), where the integer within '(' and ')' is the split-id and the integer within '[' and ']' is the timestamp at which the node was split. Here we follow the rule that nodes earlier in DFS order should be split before those that come later (if they are split at all that is). Now, notice that this is not the only way this partition could have been generated, it could have also been generated by the following sequence of splits: (4), (3), (1), (2) and (2). Though the final partitions are identical, the binary trees resulting from these two sequences are structurally very different.

There are many reasons to avoid duplicate generation, the two most compelling ones are the following: (i) Our approach, as in any optimization algorithm, will enumerate solutions and choose the best one based on cost. If duplicates are generated, the number of elements generated explodes. For instance take the simple case of one dimensional partitions with N split values. The true number of distinct partitions is 2^N, whereas if we did a blind

enumeration along each branch of the enumeration tree, we will end up with $N!$ total partitions, which is exponentially larger than 2^N. Though it is easy to eliminate duplicates by imposing a simple order on the splits it is much more complex in the case of general hierarchical partitions in arbitrary dimensional spaces. (ii) The second reason to avoid duplicates is also motivated by the nature of branch and bound solution approach we take, whose performance depends on the effectiveness of pruning. Duplicate generation severely degrades the prunability of branches, therefore practically rendering such an approach useless since many (all) good solutions will appear in all branches of the enumeration tree.

We now give some useful definitions before presenting our partition enumeration problem.

3.1 The Space of Hierarchical Partitions

Let us assume for ease of visualization that each attribute is numeric, i.e., coming from a totally ordered domain and has been appropriately discretized. (Extension of our methods to categorical attributes can be found are discussed in the next subsection). Then the *split set* is defined as follows.

Definition 3.1 (Split set): *The set of all (attribute, value) pairs at which the space is allowed to be partitioned, is called the split set and is denoted by S.*

Definition 3.2 (Split ordering): *We impose an absolute order on the elements of the split set defined above. Therefore each ordered pair: $(attribute, value)$ denotes an unique split from this set. A lower id of a split denotes a higher priority*[4]*. All split values of an attribute belong to a single* **group** *labelled by that attribute (i.e. groups $S_1, \ldots, S_m$ are the set of splits belonging to attributes $A_1, \ldots, A_m$ respectively, where $S = S_1 \cup S_2 \cup \ldots S_m$). In a geometric interpretation, splits in a group correspond to parallel planes in the space (In figure 2 splits (1) and (2) belong to one group and splits (3) and (4) belong to another group).*

If a common split runs across some rectangular subspace, we say it is a *cut* across that subspace. For example, in figure 3, split (3) is a cut across the whole space and split (4) is the cut of the subspace denoted by the right child of the root. A cut is formally defined as follows:

Definition 3.3 (Cut of a subspace): *A split* **s** *is said to form a* **s-cut of a subspace** *H if the union of* **s***-cuts across one or more of its constituent partition blocks also runs across the whole subspace (i.e. divides H into two disjoint parts).*

[4]For attributes with ordered domains, we assign ids to the splits sequentially, where splits corresponding to a higher values of the attribute get higher ids. Though any random sequence can be assigned as well.

Now, a hierarchical partition[5] of the (multidimensional) domain space of the data is defined as follows:

Definition 3.4 *(***Hierarchical Space Partition):** *Given a multidimensional space H and a set of candidate splits S such that each $s \in S$ is a potential s-cut of H, then a hierarchical partition of the space is achieved by either leaving H undivided or by choosing a cut s_0 of H and then recursively partitioning the two resultant subspaces H_1 and H_2 hierarchically using splits from the set $S - \{s_0\}$.*

A hierarchical partition can be represented by a labelled binary tree which we call the **partition tree**. We note that a partition tree belongs to a class of trees called the **kd-tree**[6] [5], which is a popular data structure to represent multidimensional partitioning of a space. We represent each partition in our solution space by a corresponding partition tree as defined below.

Definition 3.5 (Partition tree): *A partition tree is a labeled binary tree that represents a hierarchical partition of the space. Each internal node of the tree denotes a subspace that is further partitioned into two or more subspaces. A leaf denotes an undivided subspace (or a partition block as we call it). Each internal node is associated with a split-id denoting the split that was used to cut this subspace into two. Left and right side for a split s correspond to the side with lower split ids and higher split ids than s in its group respectively. Therefore leftchild and righchild are well defined for each node in a partition tree. Visualizing geometrically, the partition tree is same as a kd-tree [5].*

A new partition tree is generated from a parent tree by splitting one of its leaf nodes (i.e. a partition block), which then becomes an internal node with two newly generated leaves as it children. See figure 3 which shows how addition of a new split to an existing partition is reflected in the corresponding partition tree.

Generating partition trees: Our enumeration algorithm (to be described shortly) generates partitions incrementally, starting with the undivided space and splitting one partition block at a time. The starting state is denoted by a partition tree which has single node, the *root*. At this point the partition of space consists of just one block, to which all data belongs. Then, the first split, say s_i is chosen at the root which divides the space into two new blocks. The new partition is represented by a two level binary tree and the two leaves denote the *leftchild* and *rightchild* of the root. A split applied to a node n in the partition tree is denoted by the label "(s_i)" beside n in

[5]From here on we will omit using the qualifier "rectangular" for brevity

[6]There is an injective mapping (but not onto) from the set of our space partitions to the set of KD-trees with node labels corresponding to the attribute split values

the tree. The remaining set of splits ($S - \{s_i\}$) get propagated to the newly formed leaf nodes in the following manner: All the splits that are not in the same group as s_i (i.e., do not belong to the same attribute as s_i) are also available at each child. Out of the remaining splits in $group(s_i)$, the ones to the "left" of s_i go to the left child and the ones to its "right" go to the right child. These propagated splits called the set of *available split* at the corresponding nodes in the partition tree. For each internal node of the partition tree, the split at the node is restricted to the subspace denoted by the node. To generate a new partition from a given one, one of its leaves (i.e. partition blocks) is divided into two new blocks using a split available at that node and the remaining splits from this set get propagated as described above. Starting with the single node partition tree which denotes the undivided space and applying a sequence of one or more node splits to this tree generates a new partition of the hierarchical space.

Timestamped Partition Trees: We associate a logical *timestamp* with each internal node of the partition tree. The timestamp represents the order in which the internal nodes were split. The tree shown in figure 3 is a timestamped partition tree, where the root has the timestamp of 1 (shown as a label in square brackets), the left child of the root node has a timestamp 2, the right child has a timestamp of 3, and so on. This implies that in creating the partition tree, the root was split first, followed by the left child, then the right child, etc. As will become clear, the concept of timestamp is important in preventing duplicate enumeration of the same hierarchical partition. Henceforth, by partition tree, we will refer to trees in which internal nodes are labelled by timestamp that represent the order in which the node was split to create the tree.

Now we describe our algorithm for the systematic enumeration of the hierarchical partitions that can be represented uniquely as the leaves of a multi-way "partition enumeration tree".

3.2 Partition Enumeration Tree (PET)

We first give the definition of a partition enumeration tree and then discuss how to generate it.

Definition 3.6 (Partition Enumeration Tree): *The partition enumeration tree (PET) is the multi-way tree, in which each node (including the root and leaves) denotes a distinct hierarchical partition of the space. Each node n of PET corresponds to a unique timestamped partition tree denoted $P(n)$.*

Our objective is to generate a PET such that any two distinct nodes of the PET correspond to distinct space partitions. Nodes in PET are generated recursively by splitting one of the subspaces of the partition tree, $P(n)$, associated with a node n of the PET using one of the split values

that are available to split the subspace. We can avoid generating duplicate partition trees that are structurally similar to each other by exploiting the timestamps associated with nodes of the partition tree. This is achieved by imposing the following constraint on the partition trees associated with any node in the PET.

Constraint 1 *A partition tree associated with any node of a PET satisfies the constraint that a* **pre-order** *traversal of its nodes (i.e. the partition tree's nodes) lists their timestamps in the increasing order.*

The above constraint prevents PET from generating nodes whose associated partition tree are structurally identical. However, as we observed in the example in the beginning of the section, the same hierarchical partitioning could result from two partition trees that are not structurally identical. Notice that the above simple constraint on timestamps would not prevent generation of this duplicate partitioning. To prevent such duplicate generation of hierarchical partitioning, we need to define a concept of a in-sequence splits in a partitioning tree:

Definition 3.7 (In-Sequence splits): *Let P be a partition tree, n be a node of the tree, and A be the set of ancestors of n in P. The split s_n associated with n is* out of sequence *if there exists an ancestor $a \in A$ of n such that: s_n generates a s_n-cut across the subspace rooted at a and the split s_a associated with a has a lower id than s_n. If a split s_n is not out of sequence, it is deemed* in-sequence *for the partition tree.*

Duplicate generation of hierarchical partitioning in the PET can be prevented by imposing the following additional constraint on the partition tree $P(n)$ that correspond to the nodes of the PET.

Constraint 2 *A partition tree associated with nodes of a PET satisfies the constraint that the splits s_k corresponding to any of its internal node k are* in-sequence.

Consider the example of figure 3 again. Notice that the partition tree in the figure, seen without the node with timestamp [5] is legal. But, the final split (2), made at timestamp [5] would have been out of sequence since it results in a cut of the subspace rooted at its parent node (i.e., split(4)) while its id is smaller than the id of its parent.

Now, we describe our partition enumeration algorithm.

3.3 Enumeration Algorithm

In this section, we develop an algorithm to enumerate duplicate-free hierarchical partitioning. The basic approach ensures that the partition tree associated with the nodes of the PET satisfy Constraints 1 and 2. The algorithm generates nodes in the PET recursively. Let n be the current node

Detect-legal-split$(\mathbf{P(n)}, \mathbf{l}, \mathbf{s_l})$
Input: $P(n)$ is partition tree of node $n \in PET$
l is the leaf being split in $P(n)$
(i.e. l denotes an unsplit partition block)
s_l is a candidate split at l
Output: True(False) if split s_l is Legal(Illegal)

BEGIN
1. **If** $l == root(P(n))$ **Then**
2. **Return** TRUE
3. **End if**

4. **If** Pre-order listing of node time-stamps in
5. $(P(n) \cup s_l)$ NOT in increasing order **Then**
6. **Return** FALSE
7. **End if**

8. $a \leftarrow l$
9. **While** $a \neq root(P(n))$ **Do**
10. $a \leftarrow parent(a)$
11. **If** Is-split-a-cut$(P(n), a, s_l)$ **Then**
12. **If** $s_l \leq split(a)$ **Then**
13. **Return** FALSE
14. **End if**
15. **Else** /* s_l is not a cut of subspace of a */
16. **Return** TRUE
17. **End if**
18. **End While** /* now $a = root(P(n))$ */
19. **Return** TRUE
END

Figure 4: Detecting a legal split

of the PET being explored and let $P(n)$ be its corresponding partition tree. The partition tree associated with the child node of n consists of $P(n)$ augmented with an additional split of one of the subspaces in $P(n)$. Generating such partition trees for child nodes to satisfy Constraint 1 (i.e., timestamp ordering) is straightforward. The constraint can be verified at the time the child node is created.

Checking if the partition tree associated with the child node satisfies Constraint 2 is more involved. Since it requires "stitching" splits across subspaces to detect out-of-sequence splits as described below.

When a candidate split $s \in S$ is being considered at a leaf node in some partition tree $P(n)$, it is possible that a bunch of s-cuts across adjacent subspaces of the current block (denoted by the leaf) when taken together, form a s-cut of a larger subspace. The following two-part check is carried out by the function **Detect-legal-split** (figure 4: The function first calls the routine **Is-split-a-cut**$(P(n), t, s)$ (figure 5) which detects if the candidate split s is a cut across a subspace denoted by node t in partition tree $P(n)$. If the candidate split is indeed a cut of the subspace denoted by node t, its

Is-split-a-cut$(\mathbf{P(n)}, \mathbf{t}, \mathbf{s})$
Input: $P(n)$ is partition tree of node $n \in PET$
t is an internal node of $P(n)$ (can be the root)
s is some splitting attribute value
Output: True if there is a s-cut across the
subspace denoted by t else False

/* Here we need a small test to determine
if some split s' is in the same group as
split s (i.e., belongs to same attribute) */

BEGIN
1. **If** $split(t) \in group(s)$ **Then**
2. **If** $split(t) == s$ **Then**
3. **Return** TRUE
4. **Else If** $split(t) < s$ **Then**
5. **Return** Is-split-a-cut$(P(n), rightchild(t), s)$
6. **Else** /* $split(t) > s$ */
7. **Return** Is-split-a-cut$(P(n), leftchild(t), s)$
8. **End if**
9. **Else** /* $split(t) \notin group(s)$ */
10. **If** $leftchild(t)$ is a leaf **OR**
11. $rightchild(t)$ is a leaf **Then**
12. **Return** FALSE
13. **Else**
14. **Return** (Is-split-a-cut$(P(n), leftchild(t), s)$
15. **&** Is-split-a-cut$(P(n), rightchild(t), s)$)
16. **End if**
17. **End if**

END

Figure 5: Detecting if a new split results in a cut across a larger enclosing sub-space

id is compared to the current node split at t and a violation is returned if $s < split(t)$ (i.e., s is out-of-sequence with t) else the same check is iteratively carried out at the parent of t and so on till either a violation is detected or t has no parent, i.e. node t is the root of $P(n)$. The following observation is the key to an efficient implementation of this check which simply uses the information in $P(n)$ at a node $n \in$ PET.

Observation 3.1 *The partition tree $P(a)$ at an ancestor a of node n in PET is simply a tree generated by deleting a proper subtree of $P(n)$. Therefore $P(n)$ encodes all the information of its ancestors (nodes along the root-to-node n path) in PET.*

The recursive enumeration algorithm **Enumerate** is given in figure 6. The algorithm is given the parameters: data set D, the ordered set of splits S, a node r of the PET and the partition tree $P(r)$ at node r. When **Enumerate** is invoked with r denoting the root of the PET, it generates the complete enumeration tree.

Enumerate$(\mathbf{D}, \mathbf{S}, \mathbf{r}, \mathbf{P(r)})$
Output: The enumeration tree rooted at r,
consisting of all distinct partitions legally
generatable from the partition $P(r)$

/* Below, s_l ranges over only the "available"
splits at the leaf l (i.e. only the splits relevant
to the partition-block denoted by l). */

BEGIN
1. **For Each** leaf l of $P(r)$ /* in pre-order sequence */
2. **For Each** available split s_l at l /* increasing order of id */
3. **If Detect-legal-split**$(P(r), l, s_l)$ **Then**
4. Generate new child node c
5. Generate partition-tree $P(c) \leftarrow P(r) \cup s_l$
6. Add pointers $r \rightarrow c$; $c \rightarrow r$
7. **Enumerate**$(D, S, c, P(c))$
8. **End if**
9. **End For**
10. **End For**

END

Figure 6: Enumerating all partitions of space

An example of a (partial) partition enumeration tree (PET) is shown in figure 7 for a data space with two attributes, one having 2 splits (1) and (2) and the other having a single split (3).

We can now state the main theorem that proves the correctness of our algorithm.

Theorem 3.1 *Algorithm* **Enumerate(D,S)** *systematically enumerates all distinct hierarchical partitions of the multidimensional space D generated using splits from the given split set S.*

Proof of the above theorem is given in appendix A.

3.4 Categorical Attributes

In the description of the enumeration algorithms above, we have implicitly assumed that the attributes have an order defined on their domains. But in reality many attributes of a table can be categorical with absolutely no order defined on the values in its domain. Alternatively, there could be some partial order defined in the form of a lattice structure. A popular way of specifying a partial order over categorical attributes, is via a taxonomy tree. An example of a taxonomy tree is shown in figure 8 for the attribute *working class* in the "Adult" data set [9]. The set of node labels of the taxonomy tree specifies the domain for the corresponding categorical attribute. For instance, the *working class* attribute of a tuple can take values only from the set of node labels in the taxonomy tree.

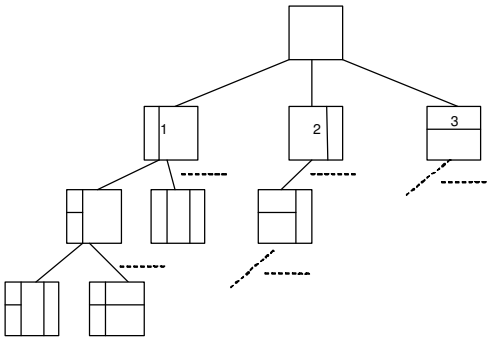

Figure 7: Partition enumeration tree

Our enumeration algorithm can handle categorical attributes easily with a couple of slight modifications to the approach outlined for numeric attributes. We categorize the categorical attributes into the following two classes and specify the corresponding modifications that are required to split along these dimensions of the space.

1. **No order:** In this case, no order is defined on the set of values taken by the attribute. As a result, all possible groupings are possible. For example, say the attribute takes 3 distinct values *a, b* and *c*, therefore blocks (anonymity groups) in a partition can cluster together these values in any of the following 7 ($= 2^3 - 1$) ways: {(a,b,c);(a,b);(a,c);(b,c);(a);(b);(c)}, where all values within '(' and ')' are indistinguishable. This case is equivalent to having three independent binary attributes instead of one with three values.

2. **Partial order:** If the categorical attribute is denoted by C, in this case we assume a taxonomy tree T_C is defined for the values that C can take. The following additional constraints need to be imposed while splitting a set of data points along the attribute C (refer to figure 8): (i) For a given generalization scheme, induced by some partition, all tuples in a partition block (anonymity group) share the same values of the quasi-identifier attributes. Therefore, attribute C for each tuple in a block l should correspond exactly to one node t_c of the taxonomy tree T_C (say "Government"). Let the set of these tuples be D_l. (ii) Subsequent splitting of the block l along attribute C should **simultaneously specialize** the C-attribute values of tuples in D_l to the children of node t_c in T_C. For example, splitting D_l would result in groups D_{l1}, D_{l2} and D_{l3} with *working class* values as "Fed", "State" and "Local" respectively, where $S = D_{l1} \cup D_{l2} \cup D_{l3}$ and $D_{li} \cap D_{lj} = \phi, \forall i \neq j$.

Tackling categorical attributes without any order whatsoever is easy

since such an attribute can be replaced by a set of binary attributes. In this paper, we assume each categorical attribute has a partial order defined on them, in the form of a taxonomy tree. Recall that in the enumeration algorithm, new partitions are generated from a given partition by splitting some partition block of the parent partition into two blocks by introducing exactly a single split along some chosen dimension. This allows us to represent each new partition also as a binary tree. But introducing the splitting constraint ((ii) in item 2 above) can result in more than two blocks being generated simultaneously when a partition block of the parent partition tree is split along a categorical attribute. Say, if the node corresponding to the current value of the categorical attribute C has m children in its taxonomy tree T_C, splitting along this attribute will result in m leaves simultaneously in the new partition tree. Therefore to represent this new partition in the standard format of our partition tree, we need to assign priorities to the m children of the corresponding node in the taxonomy tree. Two considerations need to be made while introducing these splits into the parent partition tree: (i) The "child priority" constraint 1 and "Out-of-sequence split" constraint 2 ordering constraints are not violated in the resulting partition tree due to introduction of these sibling splits (i.e. sibling nodes in T_C. (ii) Future splits in the newly generated blocks should not be wrongly discarded (i.e., for violating either the time-stamp constraint or the split priority constraint). These two requirements needs one to assign: (a) Consecutive split priorities to each set of sibling splits. (b) Also, assuming the "**left-to-right**" listing of the sibling nodes in T_C correspond to the actual left to right ordering of the splits in the space[7], the assigned priority should be **decreasing** amongst sibling splits going from **right to left** (as shown in figure 8); (c) The sibling splits (in T_C) need to be introduced simultaneously as noted above, but we also need to retain the binary tree form of the partition tree. As a result the sibling splits need to be introduced in their decreasing order of priority (i.e., high priority splits first). This results in creating a **left deep** binary subtree under the leaf being split and therefore complies with the time-stamp constraint. The example below illustrates the splitting technique.

Example: Assume we have a two dimensional data set with one dimension as numeric attribute *salary* with four values (intervals) and the other, the categorical attribute *working class* with a taxonomy (figure 8, part (b)) defined on it. Part (a) of figure 8 shows the space and all the available split values with their priorities assigned according to the rules mentioned above. Initially assume that the space is undivided therefore all data elements are indistinguishable from each other. Now let the first split be introduced along attribute *working class* which results in creation of the four blocks in one go (figure 9, part (a)) thereby adding 3 nodes to the partition tree, in the left-

[7]remember that each attribute has a well defined notion of "left end" and "right end" along the dimension corresponding to it in the data space

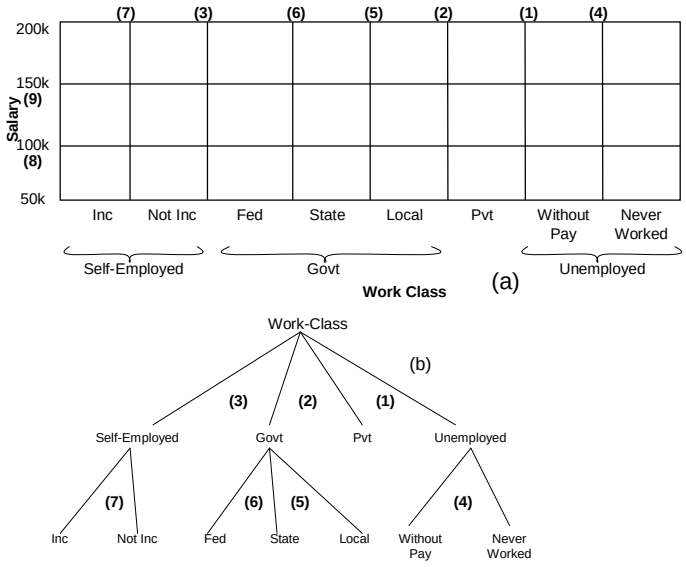

Figure 8: (a) Split priorities for a categorical attribute; (b) Priorities for taxonomy-tree nodes

deep manner (note this complies with the time-stamp ordering constraint as well as the out-of-sequence split constraint). In part (b) of figure 9, say a split along the *salary* attribute is chosen in the second block from left which adds node "(9)[4]" (split (9) and time-stamp [4]). Then, splitting the lower of the two newly generated blocks (shaded portion) along the categorical attribute, results in three more blocks in one go, i.e. 2 new nodes in the partition tree (time stamps [5] and [6]). $\diamondsuit$

4 Search for Optimal Partitioning Schemes

The anonymization problem can be viewed as a cost based optimization problem. Without loss of generality we will assume that the optimization is a constrained minimization problem. Abstractly, an optimization problem is specified using two entities, *Cost* and *Set of Constraints*.

Definition 4.1 *Cost* : $N \rightarrow \mathbb{R}$*, where* N *is the set of all nodes in PET, is a real function that associates each partition with a non-negative cost.*

Definition 4.2 Constraints*: A set of properties which determines the set of all feasible solutions. That is all potential solutions should minimally satisfy these constraints. We will also refer to an element from the feasible set as a* legal *solution.*

Different instances of these two parameters define different optimization problems. In context of the anonymization application, we define a few

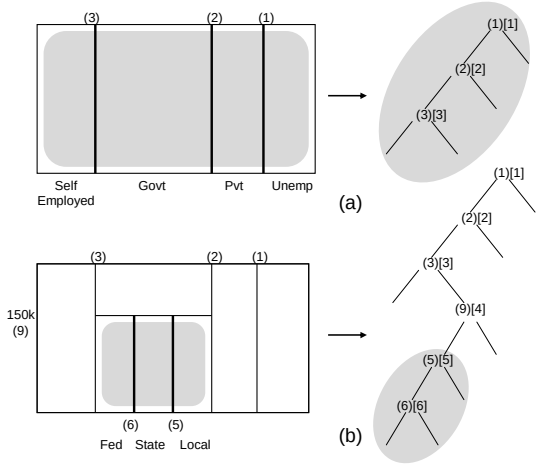

Figure 9: Splitting along categorical attributes can introduce multiple nodes in partition tree in one step

variants of the optimization problem using the following cost functions and constraints.

Cost functions: We consider the following cost functions:

- **Discernibility metric (DM):** Proposed in [2], it assigns a penalty to each tuple based on how many tuples in the transformed data set are indistinguishable from it. If a tuple belongs to an anonymity group S_i of size n (i.e. has $n-1$ other tuples present in it), then that tuple is assigned a penalty of n. If a tuple is suppressed a penalty equal to the size of the data set $\mid T \mid$ is assigned.

$$DM = \sum_{\forall i,\ |S_i| \geq k} \mid S_i \mid^2 + \sum_{\forall \text{ Suppressed Tuples}} \mid T \mid$$

- **Classification metric (CM):** Also proposed in [2, 1, 14], captures the notion of information loss for classification based mining tasks. Assume $|T| = N$, then the information loss is given by:

  $CM = \sum_{i=1}^{N} penalty(T(i))$ where $T(i)$ is the i^{th} tuple and $penalty(t) = 1$ if $class(t)$ is not the majority class in the anonymity group of tuple t.
- **Volume metric (VM):** We propose a new metric to capture the notion of "relative increase in uncertainty" associated with the representation of a data point in the anonymized data set. For each tuple in a given equivalence class, a penalty $V/unit\_volume$ is charged, where V is the "normalized volume" of the partition block to which the point belongs and $unit\_volume$ is the normalized volume of a basic unit cell of the data space (as a results of the initial discretization along the

dimensions of the space).

$VM = \sum_{i=1}^{N} Volume(block(T(i)))/unit\_volume$ where $Volume$ is the normalized volume function and $block(t)$ denotes the partitions block to which tuple t belongs.

Constraints: We experimented with three different constraint settings which are relevant to privacy problem in data publishing.

- **k-Anonymity:** For every tuple t' in the released table, there should at least be l distinct indices (including its own) $(i_1, i_2, \ldots i_l)$, where $l \geq k$, such that $t_{i_1}(Q) = t_{i_2}(Q) = \ldots t_{i_l}(Q) = t'$, where $t(Q)$ denotes the projection of tuple t on the quasi-identifier attribute set Q.
- **Entropy l-diversity:** Using the definition from [4], a table is said to be entropy l-diverse if for every equivalence class (partition block) b, the following holds.

$$-\sum_{r \in R} p_{(b,r)} log(p_{(b,r)}) \geq log(l)$$

  where $p_{(b,r)}$ denotes the fraction of tuples in the block b with sensitive attribute value equal to r, and R is the set of values (categorical) for the (single) sensitive attribute. This constraint ensures that each equivalence class has at least l well represented values of the sensitive class.
- **k-Anonymity with length restrictions:** Here in addition to the k-anonymity constraint we impose a minimum length constraint on the edge length(s) of a partition block in the final anonymized data set. Such constraints could reflect some security policy, wherein some of the quasi-identifying attributes are not allowed to be specified beyond a certain level of precision.

Definition 4.3 (Optimal Anonymization Problem): *Given the above set of cost functions and constraints, for any combination of the two, find a hierarchical partition of the multidimensional data set such that the constraint is not violated for any partition block (equivalence class) and the cost of the solution is minimized.*

Now, we will describe how the above optimization problem can be posed as a search problem and illustrate our branch and bound technique to find an optimum solution.

4.1 Searching for the Optimum Solution

Given an instance of the optimization problem, we need to look for an optimal solution. A simple solution would be simply call algorithm **Enumerate**

(figure 6) and return the least cost legal partition it generates[8]. The problem is that the search space for hierarchical partitions grows exponentially with the number of dimensions and number of splits, thereby making such a complete search impractical. More over not all enumerated partitions are necessarily legal, therefore to make the search tractable we employ cost lower bounding based node pruning. We also show how a nice structural property exhibited by our PET can be exploited to derive tight lower bounds.

It is clear that efficiency of the search algorithm critically depends on how much of the search space can be pruned away during enumeration. The following condition must hold for pruning a node n (at its generation time) in our PET:

Condition 4.1 (Condition for pruning) *The subtree rooted at n in the PET can be pruned if at least one of the following two conditions hold at each node in the subtree: (1) The partition corresponding to the node is not a feasible solution (violates one or more constraints); (2) The partition has a cost larger than the current_minimum_cost in the algorithm.*

The following monotonicity property of the constraints is essential for effective pruning of nodes in an enumeration tree based branch and bound approach.

Property 4.1 (Monotonicity of a Constraint) *Given optimization problem P, a constraint C and enumeration tree ET such that nodes $\in ET$ correspond to elements of the solution space of P, then C is said to be monotonic with respect to ET if whenever a node $n \in ET$ violates C, all its descendants in ET also violate C.*

If a node n in the search tree cannot be pruned for constraint violation, then we check for the second condition of prunability for the subtree rooted at n. To carry out this check an efficient (i.e., one that does not traverse the whole subtree) function LB is required to estimate a lower bound to the minimum cost in the subtree. In our case, the function LB is defined as follows.

Definition 4.4 *$LB : N \rightarrow \mathbb{R}$, where N is the set of nodes in PET, is a real function that estimates a lower bound to the minimum cost for a hierarchical partition "derivable" from the partition at n (i.e. all partitions in the subtree rooted at n in PET).*

The LB computation for a node can greatly benefit if the cost functions demonstrate the following monotonicity property with respect to the enumeration tree.

[8]This requires an additional constraint satisfaction test during generating a new partition in the Enumerate algorithm

Property 4.2 (Monotonicity of Cost) *The value of the cost metric decreases monotonically down any path from the root to a leaf in the PET.*

Notice, that the monotonicity property of cost functions is a desirable property but not a necessity for computing lower bounds. For instance, in [19] it is shown that in set mining problems, cost function like *variance* of values associated with items in a set do not display such monotonic behavior. The authors in [19] go on to show some efficient ways of find a lower bound in such cases, but we will not consider such cost functions in this paper. All the cost functions and constraints that we mention above in the data publishing context, exhibit the monotonicity properties 4.2 and 4.1. The measures DM and CM have been shown to follow monotonicity for single dimensional partitions and the results can be easily extended to the multi-dimensional case. The new volume metric that we propose also exhibits a similar property due of the additive relation between the volume of a space and that of its constituent subspaces. Similarly, the privacy constraints, i.e., k-anonymity and l-diversity have been previously shown to be monotonic in [2] and [4] respectively. It is easy to see that this property extends to our new constraint of "k-anonymity with length constraints" as well.

In the following subsection, we describe the lower bounding methodology in the PET. We use the DM cost measure (defined earlier) for illustrating the approach.

4.1.1 Cost based pruning in PET

In addition to the monotonicity of our PET has a nice structural property that allows one to compute good lower bounds for all the cost functions described above. This property is potentially beneficial for lower bounding cost functions belonging to a rather well studied class of functions, which happen to be some *spatial aggregates* over the data distribution across the partition blocks [16]. We describe this property next.

Property 4.3 (Spatial locality property)*: Constraint 1 imposes restrictions on the set of subspaces that can be further partitioned at any descendant of a node in the PET. Specifically, consider any ancestor-descendant pair n_a and n_d in the PET and their corresponding partition trees $P(n_a)$ and $P(n_d)$, then $P(n_d)$ can differ from $P(n_a)$ only in the subtree rooted at the rightmost, non-leaf node m_r of $P(n_a)$. That is, m_r denotes the last internal node in $P(n_a)$ on the path starting at its root (inclusive) and containing only right branches of nodes, till a leaf is reached.*

We now illustrate the lower bound computation for DM metric (the others can be computed similarly) using the example given below and present the corresponding algorithm LB_{DM} in figure 10.

LB$_{DM}$**(m)**
Input: m is some node in a partition tree
Output: Lower bound to the cost of any
partition of the space denoted by m
BEGIN
1. **If** m is a leaf **Then**
2. **Return** $MinCost(m)$
3. **End If**

4. $m_r \leftarrow$ right-most internal node $\in subtree(m)$
5. $L = \sum(Size(leaf))^2$, such that
$leaf \in subtree(m)$ & $leaf \notin subtree(m_r)$
6. $L \mathrel{+}= MinCost(m_r.rightchild)$
7. **If** $m_r.leftchild$ is a leaf
8. **Return** $L + MinCost(m_r.leftchild)$
9. **Else**
10. **Return** L + LB$_{DM}(m_r.leftchild)$
11. **End If**
END

/* A node m in a partition tree $P(n)$ denotes
a subspace. The subtree of $P(n)$ rooted at m,
referred to as $subtree(m)$, denotes the
restriction of the partition to this subspace.
$Size(leaf)$ is the number data points belonging
to the partition block denoted by $leaf$ */

Figure 10: Computing lower bound of DM

The lower bound of the cost metric for a leaf l of a partition tree is computed by the function call $MinCost(l)$ (illustrated below for the DM metric). It first generates the finest partition of the subspace denoted by l by using all the available splits at l (i.e. partitions the subspace into the smallest blocks possible) and then computes the value as follows:

$$MinCost(l) = \sum_{tuple \in subspace(l)} Penalty(tuple)$$

$$Penalty(t) = \begin{cases} \mid E(t) \mid & \text{where} \mid E(t) \mid > k \\ k & \text{otherwise} \end{cases}$$

Where $\mid E(t) \mid$ represents the size of the partition block to which the data point t belongs. That is, for each block having $m \geq k$ points, add m^2. For each block having $m' < k$ points, add km'. (Since the second set of blocks have less than k elements, the anonymity criteria would have required one to suppress these elements. This would lead to a much larger increase of the DM value, but since we are computing lower bounds, we simply charge each point a penalty of k, which is the minimum possible). We illustrate the lower bound computation using the example below (refer to figure 11).

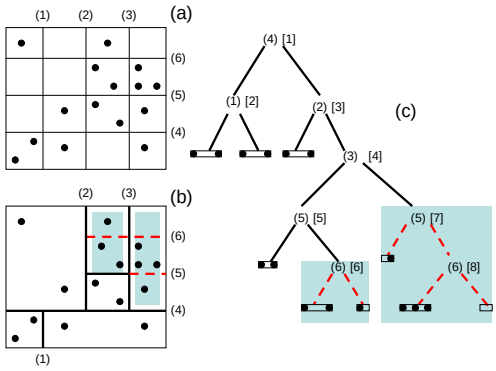

Figure 11: Visualizing lower bound computation

Example: Consider the data space shown in the figure 11(a). Assume a node n in PET corresponds to the partition shown in figure 11(b) without the "dashed" splits in the shaded region. Its partition tree is shown in figure 11(c) (visualized by replacing the 2 subtrees in the shaded region by 2 leaves instead). The splits used until this point are shown in bold, solid, black lines both in (b) and (c). The only legal partitions that our algorithm generates at any descendant of node n, will be by adding splits in the region shaded in blue in order to comply with the time-stamp constraint. The DM value of the partition $P(n)$ is given by: DM(n) $= 2^2+2^2+2^2+2^2+3^2+4^2 = 41$ (leaves traversed in depth-first order). Now, the lower bound at n is computed by first after adding all the available splits (generates 5 new leaves as shown in the shaded region) and summing the contribution from all the leaves of this partition: $\text{LB}_{DM}(n) = 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 1 * 2 + 1 * 2 + 3^2 + 0 = 33$. $\diamondsuit$

The lower bound computation for the other cost functions are similar to that of the DM metric. Now, we describe a novel optimization for improving the performance of our search algorithm.

5 Accelerating Search using Priorities

As it turns out, more often than not in spite of pruning, the search space continues to remain extremely large. As a result, the depth-first traversal order of nodes (as in the recursive algorithm) turns out to be an inefficient way to explore the solution space for most objective functions of interest. Here, by efficiency we mean "how soon the algorithm finds a solution close to optimal", say a partition whose cost is within a small factor α of the minimum cost. To do away with these limitations of the recursive algorithm, we propose a new, generic solution-space exploration scheme that is both theoretically sound (i.e. guarantees completeness of search just as the recursive enumeration algorithm) and at the same time, allows one to incorporate heuristics that accelerate the convergence to the optimum solution,

the side-effect being: good approximations to the optimal are generated much quicker. Now, instead of traversing the PET in a fixed order which is agnostic of the objective function, our new algorithm uses a flexible, *priority* based search scheme to direct its search at each step.

- $Priority : N \rightarrow \mathbb{R}$, where N is the set of nodes in PET, is a real function which assigns a numeric value that is used as the key to insert n into the priority queue. We also refer to it as the *priority* of node n.

Our algorithm admits the usage of arbitrary priority generating functions. We experimented used the following functions to generate node priorities (i.e., instances of the "*Priority*" function defined above).

1) LB (lower bound function): LB was the primary priority generating function and was used in majority of the experimental runs. The rationale being that, the branch of PET that has lowest lower bound is the most promising one to explore.

2) $Cost$ (cost function): Cost of the solution at a node n was used to prioritize the search. The goal was to see how going down a path with the minimum cost affects the quality of solutions generated.

3) $LB/Cost$ (ratio of the lower bound and the cost): Similarly, the rationale behind $LB/Cost$ was to go down branches which had the highest potential of cost improvement.

The experimental results for these functions are summarized in the next section. Now, we describe the new search algorithm using the priority queue which was the algorithm used in all our experimental runs. There are two modes in which the priority algorithm can be run: (A) Geared towards finding **an optimum solution**, where the lower bound $LB(n)$ is used to decide whether to prune node n or not. (B) Geared towards finding a c**-approximate solution** to the optimum, where the ratio $\alpha(n) = LB(n)/Cost(n)$ is used to prune node n from the search space. For instance, if the user wants to find a 3-approximate solution, a node with α value greater than 1/3 can be safely pruned since one would have already seen a candidate solution if indeed the true minimum was same or higher that the lower bound at the node (the current global minima being one such candidate). Let us use the variable *bound* to denote either LB or α depending on the mode of the algorithm's run, then the basic priority queue does the following.

Priority-queue based search algorithm: The algorithm starts with the root node of the PET as the singleton node in the queue and the value in the variable *current_minimum_cost* set to $Cost(root)$, which is the cost when the whole data set is a single partition block. In each successive step,

Prioritized-Enumerate$(\mathbf{D}, \mathbf{root})$
Input: D is the data set to be partitioned
r is the root node in PET
Output: Output the optimal partition and its cost
BEGIN
1. $OPT_{tree} \leftarrow P(root)$
2. $OPT_{value} \leftarrow Cost(P(root))$
3. **If** $Opt_{value} < LB(root)$ **Then**
4. **Return** OPT_{value} and OPT_{tree}
5. **End If**

6. $PQ \leftarrow$ insert$(root, LB(root)$
7. **While** $PQ \neq \phi$ **Do**
8. **If** $|PQ| \geq$ MAXSIZE **Then**
9. **While** $|PQ| >$ MAXSIZE/2
10. $R_p \leftarrow$ A good partition beneath $min(PQ)$
11. $OPT_{value} \leftarrow Cost(R_p)$ and $OPT_{tree} \leftarrow R_p$
12. Delete all n' in PQ with $LB(n') > OPT_{value}$
13. **End While** /* Now $|PQ| \leq MAXSIZE/2$ */
14. **End If**

15. **While** $|PQ| \leq MAXSIZE$ & $PQ \neq \phi$ **Do**
16. $x \leftarrow PQ.pop()$
17. $X_c \leftarrow x.children$ /* x + a single split */
18. insert into PQ, all $y \in X_c$ s.t $LB(y) < OPT_{value}$
19. **If** $Min_{y \in x \cup X_c} Cost(y) < OPT_{value}$ **Then**
20. update OPT_{value} and OPT_{tree}
21. **End If**
22. **End While**
23. **End While**
24. **Return** OPT_{value} and OPT_{tree}
END

Figure 12: The prioritized search algorithm

the top element of the queue[9] is popped and all its children are generated. Some of these newly generated nodes using their priorities. For each new node, if the cost is found to be lower than the value in the variable *current_minimum_cost*, this value is replaced by the new minimum and the new optimal partition is recorded. Also, for each new node n $bound(n)$ depending on the mode in which it is being executed. If the $bound(n)$ is higher than the current global minima, this node is discarded[10]. The algorithm terminates when the priority queue is empty and the partition corresponding to the current minima is the optimal solution. The pseudo-code for the priorities based algorithm is given in figure 12.

Practical issues in using a priority queue: The downside of using a priority queue based approach is that the queue can grow very large in

[9]We use a MIN-priority queue, where the root node always has the lowest value of the key amongst all nodes

[10]It is quite possible that the partition corresponding to a discarded node might be the best solution seen so far

size on certain occasions. In such circumstances, the following probe-based algorithm is used.

Probe-based algorithm: When max-allowed size is reached, this algorithm does a "probe" which temporarily puts on hold further enumeration and instead invests time to find a node in the PET that has a cost smaller than the lower bound of a large number of nodes in the priority queue. Then, these nodes become useless and can be pruned from the queue. In the event such a good partition cannot be found by probing, the alternative is to forcefully drop a certain number of entries that are the least promising, thereby forfeiting the promise of optimality of the final solutions. Nonetheless, due to the inherent nature of the prioritized algorithm, a bound on the approximation ratio achieved by any candidate solution can always be derived. Note that the efficiency of the above algorithm depends upon the choice of parameter MAXSIZE and we will present some of our experimental findings regarding the performance with respect to MAXSIZE in the next section.

In the next section, we discuss the various experiments that we carried out.

6 Experiments

Experimental setup: All the experiments were run on a Pentium 4, 3.00 GHZ processor machine with 4 GB RAM. The machine was running windows XP operating system and our enumeration algorithm was implemented using the VC++ development environment.

Data sets: There were three different datasets that were used in the experiments. Two of the datasets are from the Irvine machine learning repository [9]. The first real data set used in the experiments is the *Adult* data set, which has in some ways become the benchmark for comparing anonymization algorithms. This gives us the opportunity to compare the quality of our solutions against other algorithms/techniques previously proposed. This data has 9 attributes (Age, Geography, Gender, Race, Working Class, Occupation, Education, Class, Marriage) and reports the actual census data. The data set has 30,162 tuples.

The second real data set used in our experiments is the *Coil 2000* data set from the Irvine machine learning repository. This data set contains information on customers of an insurance company. We have taken the first five dimensions of the coil data set namely customer subtype, number of houses, average size of household, average age and customer main type. This data set has 9882 tuples.

The other dataset used in our experiments was synthetically generated, which we will refer to as *SimpleNormal*. This dataset has two integer dimensions with values ranging from 0 to 100. The values follow a two dimensional

normal distribution.

Purpose of the experiments: The purpose of our experiments is to measure: a) scalability of our enumeration algorithm; and b) comparison of enumeration algorithm with other approaches in the literature. Primarily, we compare ourself with the greedy algorithm proposed in [3], as the this approach outperforms other previously suggested techniques.

Scalability: We empirically test the performance of our enumeration algorithm, specifically the time it takes to find the optimal (in most cases, we report how close it approximately gets the to the theoretical best-possible solution when it cannot find the optimal solution in a given maximum length of the run). Here, we will only give a summary of our findings due to space restrictions. We direct the interested reader to [20] for more comprehensive analysis of the scalability of the algorithm. The most critical parameter for the enumeration algorithm is the number of splits. The number of splits dictate the solution space. We have found that the enumeration algorithm finds optimal solution under 30 minutes when the number of distinct splits used are less than equal to 10. When the number of splits were increased, we needed to drop some solutions as they could not fit in the memory and hence could not guarantee the optimality of the final solution. When the splits were more than 10, the algorithm finds solutions within a factor of 3.5 of the theoretical lower bound to the cost. Lower bound is calculated by assigning the lowest penalty possible to every tuple. For instance, for the DM metric, the lower bound is $n * k$, where n is the number of tuples and k is the anonymity factor. Whereas our enumeration algorithm can determine the optimal solution or those within a specified factor of the lower-bound to the theoretical best given sufficient space and time, the greedy algorithms proposed thus far typically return a locally optimal solution.

Comparison with greedy algorithms: Comparison to the greedy is done for 3 different metrics (DM, VM, CM) and three different constraints (K-anonymity, L-diversity and K-anonymity with minimum length restrictions). Each of these constraints have a parameter that is varied, k for anonymity, l for diversity and *length* for the minimum allowed size of the range of attribute-values in anonymity classes. We applied a total of 4 different sets of constraints to each combination of dataset and cost metric. In total we carried out 3(different metrics) * 4(constraints) * # Datasets experiments. For lack of space, we will only report experiments that are done for the DM metric in this paper. For experiments on CM and VM metric please refer to the full version of the paper [20]. We use the following notation "(D, M, S)" to denote the instance experiment on a Dataset D, satisfying Metric M and satisfying Constraint set S. For all the following experiments the enumeration algorithm was run for maximum of 5 hours.

Experiment 1:-(SimpleNormal, DM, <k - anonymity>):

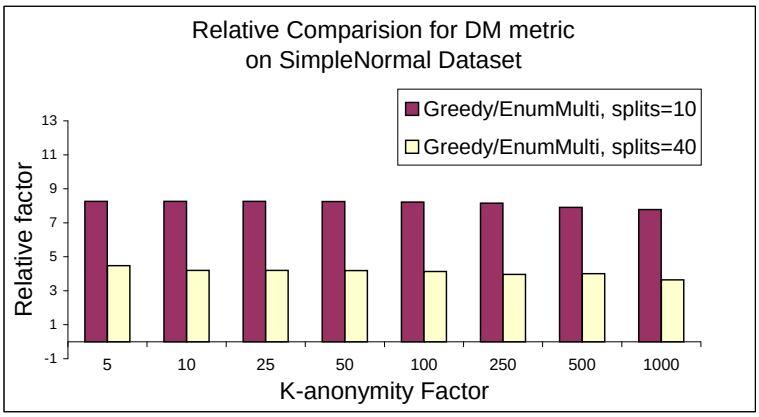

Figure 13: DM, SimpleNormal dataset

We used two different sets of splits (5×5 and 20×20) to partition the two dimensional space and ran both, our enumeration algorithm and the greedy algorithm for the DM metric. The subsequent split-widths were were chosen to be equidistant along both dimensions. Fig 13 shows the result. We plotted the ratio of the greedy value found to value of the solution found by the enumeration algorithm(represented as "EnumMulti" in the graphs). For a total of 10 splitsv(5×5), EnumMulti picked optimal solutions which were on a average 8.2 times better than the solutions found by the greedy algorithm. All the optimal solutions were found under 30 mins. For the *splits=40* (20×20) case, the solutions found by the enumeration algorithm was only 4 times better. As the number of splits increase, the difference between the enumeration algorithm and the greedy algorithm decreases.

Experiment 2 :- (Uniform, DM, <k-anonymity>):

For this experiment we have split the uniform dataset using two different sets of splits 1) 14 splits (i.e. 7 * 7) 2) 20 splits (i.e. 10 * 10). We have found optimal solutions for DM metric on the uniform dataset for 14 splits. All the optimal solutions were found under 20 mins. On a average, the enumeration algorithm does twice better than the greedy algorithm. Fig 14 shows the result. When the splits used were 20, we did not get optimal solutions, since we were forced to drop some solutions. Fig 15 shows the result of the experiment. For this run, the difference between the two approaches is converging, although still there is considerable gap. The greedy algorithm is doing better when the number of splits are increasing, as observed in the above experiment.

Experiment 3:- (Adult,DM,<k-anonymity>):

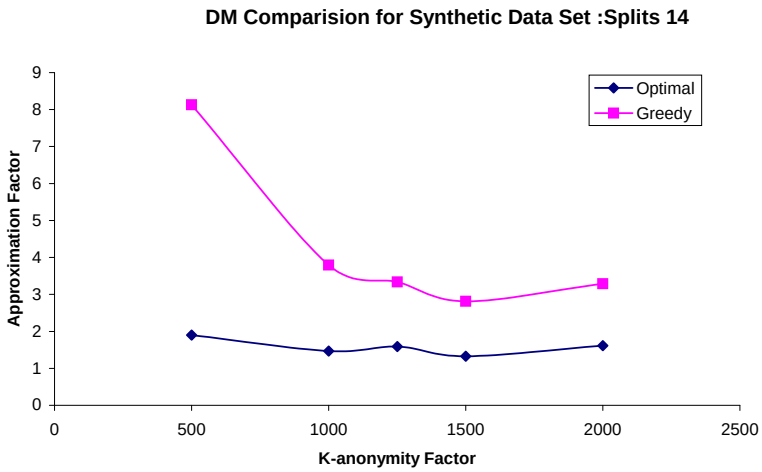

Figure 14: DM, Uniform

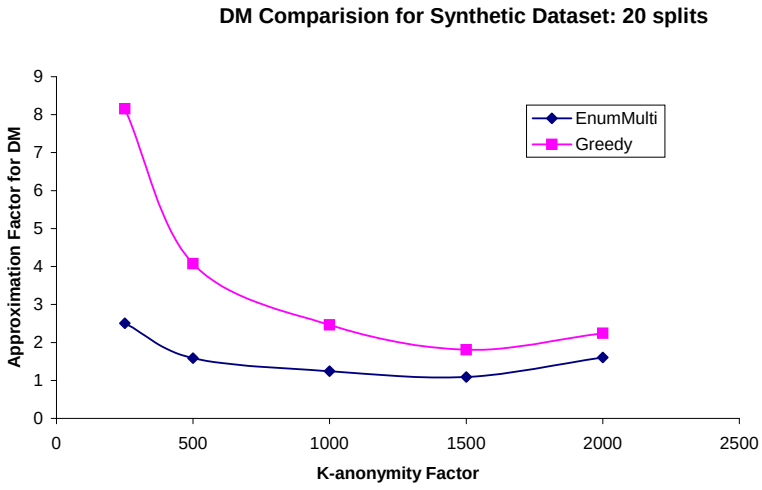

Figure 15: DM, Uniform

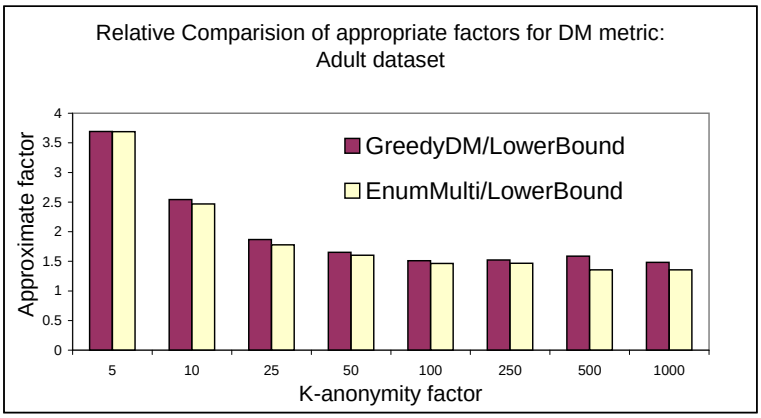

Figure 16: DM, Adult dataset

We used 101 splits to partition the adult dataset. Our enumeration technique outperforms the greedy approach at higher values of k(greater than 100) and does marginally better at lower values of k. To capture how close we are getting to the optimum solution, we calculated the approximation factor which is the ratio of the best solution found to the lowest lower bound of a partition tree enumerated by our algorithm[11]. Similarly, we have calculated the approximation factor for the greedy approach and fig 16 shows the comparison of the approximate factors for the two approaches. We have got as close as 1.35 times the lowerbound (for K=1000).

The most important result to come out of this experiment is the fact that greedy algorithm comes up with close to optimal solutions when the dataset is partitioned using a large number of splits. This confirms the intuition that we have gained from the previous experiment, where greedy algorithm was getting closer to the optimal solutions, when the number of splits increase. The greedy algorithm at every stage tries to find the best split to break a partition into two pieces. If there are a large number of splits at its disposal, even at lower levels the greedy algorithm will find a good candidate split. Since the DM metric is monotonically decreasing, as long as the greedy algorithm finds a split, the value of the DM metric value will decrease.

Experiment 4:- (Coil,DM,<k-anonymity>):

To confirm the above result we ran our enumeration algorithm for the DM metric on the Coil dataset. We are doing better in comparison with the greedy approach in the Coil dataset than the adult dataset. Fig 17 shows the comparison of the approximate factors of the two approaches and clearly there is considerable gap between the two data series. But even here the

[11]This is also lower bound of the optimal solution

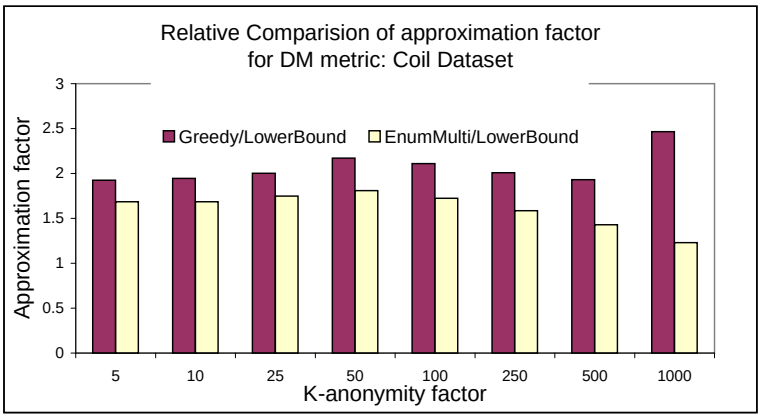

Figure 17: DM, Coil dataset

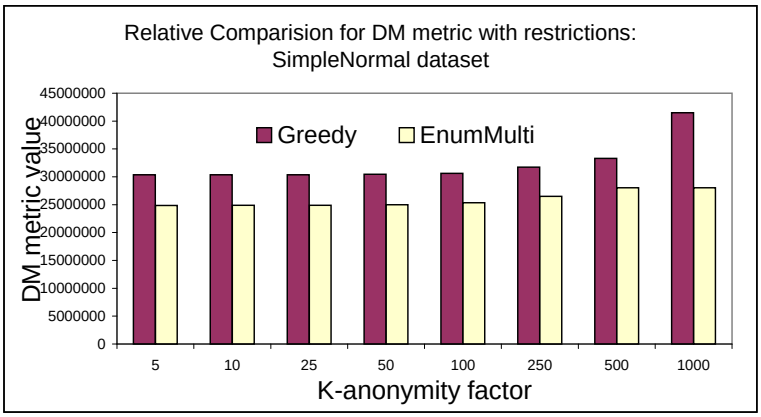

Figure 18: DM,SimpleNormal, with length constraints

greedy approach gets closer to the enumeration algorithm for lower values for k.

Experiment 5:-(SimpleNormal, DM,<k-anonymity,Length>):

For this experiment the SimpleNormal dataset was partitioned using 50×50 splits. We also imposed length restrictions of size 10 units along each dimension (i.e., "no partition-block should have an edge-length of less than 10 units"). Fig 18 shows the result. The enumeration algorithm outperforms the greedy algorithm for all values of k by a significant margin. This illustrates how the greedy is more likely to get caught in local-minima as the set of constraints get more diverse.

When constraints apply, the greedy algorithm does not have many splits at its disposal at each stage due to the restrictions imposed by minimum length requirement. This is specially true at the lower levels of the partition

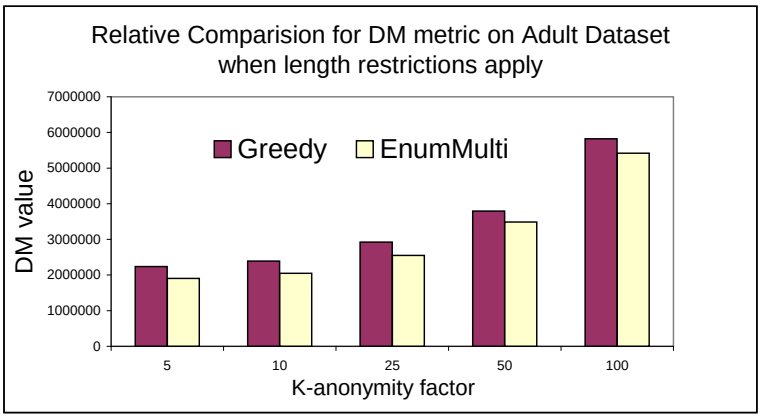

Figure 19: DM, Adult dataset, with length constraints

tree where the effect of the length restrictions is the greatest. The length restrictions reduce the solution space at each subsequent stage and since the greedy approaches don't use any back-tracking, they get caught in some local minima most of the time. In comparison EnumMulti systematically enumerates this reduced solution space, and is more likely to do much better than the greedy.

Experiment 6:-(Adult, DM, <k-anonymity,Length>):

Here we compared the cost of solutions using EnumMulti and greedy on the Adult dataset by imposing minimum length restrictions. In seven out of the nine attributes (Age, Geography, Race, Working_class, Occupation, Education, Marriage) we imposed a minimum length requirement equal to twice the inter-split distance. The other two dimensions were left out since twice the inter-split distance along these dimensions would amount to the total range of the dimension. Fig 19 shows the results. Even for this dataset, the enumeration algorithm does well for all values of k against the greedy approach, therefore confirming our intuition from the previous experiment.

Experiment 7:-(Coil, DM, <k-anonymity,Length>):

In this experiment a minimum length equal to twice the inter-split distance(minimum possible) was imposed on all the five dimensions of the Coil dataset. Fig 20 shows the result.

Experiment 8:-(Adult,DM, <k-anonymity,l-diversity>):

We compared both the approaches when L-diversity restriction, together with the k-anonymity is applied on adult dataset. We have taken *occupation*

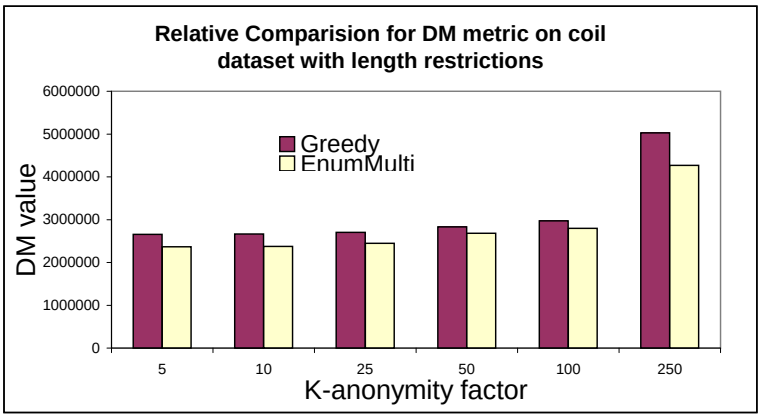

Figure 20: DM, Coil dataset

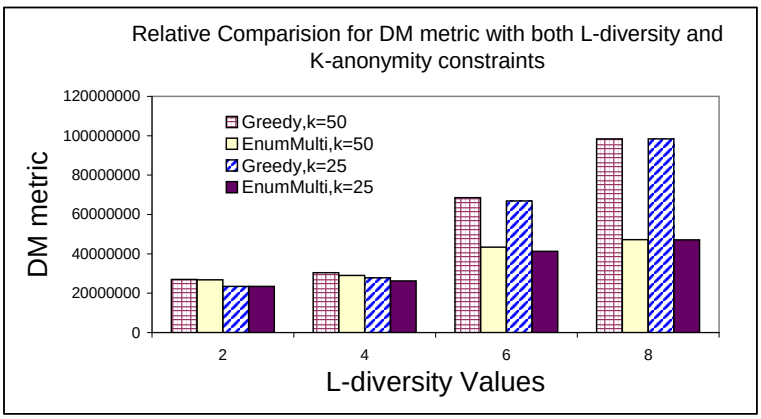

Figure 21: L-diversity, Adult dataset

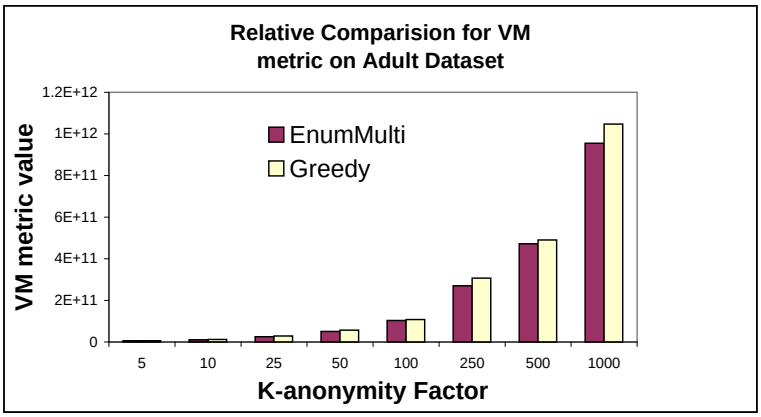

Figure 22: VM, Adult dataset

dimension as the sensitive attribute which has 14 different values, similar to [4]. For this experiment we have kept the anonymity constraint k constant and varied the l-diversity. This was done for two values of k (i.e. 25 and 50). Fig 21 shows the result. Our enumeration algorithm outperforms the greedy approach when L-diversity desired is more than 4 for both values of k. Diversity of 4 is easier to achieve in this dataset.

Experiment 9:- (Adult,VM, <k-anonymity>):

For this experiment we have used 101 splits, similar to the run where the DM metric is optimized. The enumeration algorithm mimics the result of the DM metric for the VM metric. The enumeration algorithm does better than greedy for higher values of k, but the greedy algorithm comes close to the enumeration algorithm for lower values of k. Fig 22 shows the result of the experiment.

Experiment 10:- (Adult, CM, <k-anonymity>):

For this experiment we have used 101 splits, similar to the run where the DM metric is optimized. The enumeration algorithm mimics the result of the DM metric for the CM metric. The enumeration algorithm does better than greedy for higher values of k, but the greedy algorithm comes close to the enumeration algorithm for lower values of k. Fig 23 shows the result of the experiment.

Miscellenaous experiments:

Another interesting measurable factor was the priority function(key) of the priority queue. Changing the key of the priority queue leads to interesting search patterns on the partition enumeration tree and could potentially

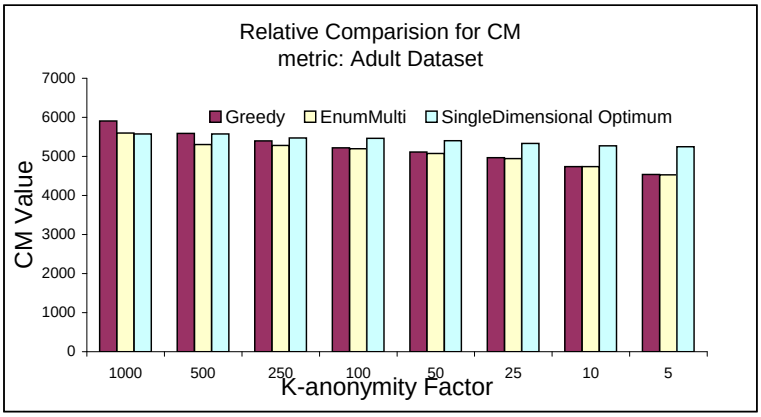

Figure 23: CM, Adult dataset

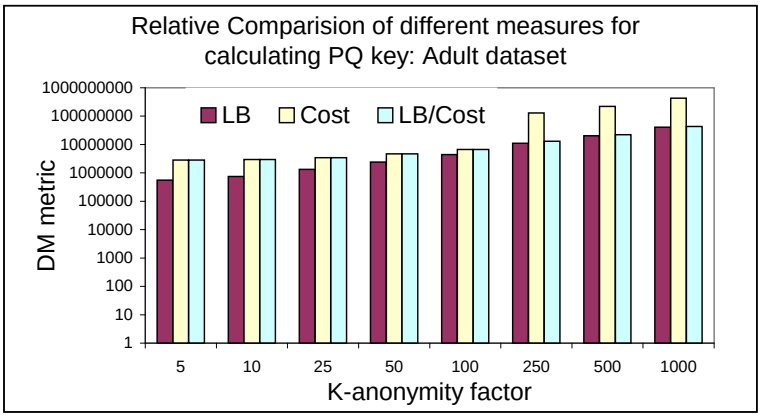

Figure 24: Biasing Priority queue

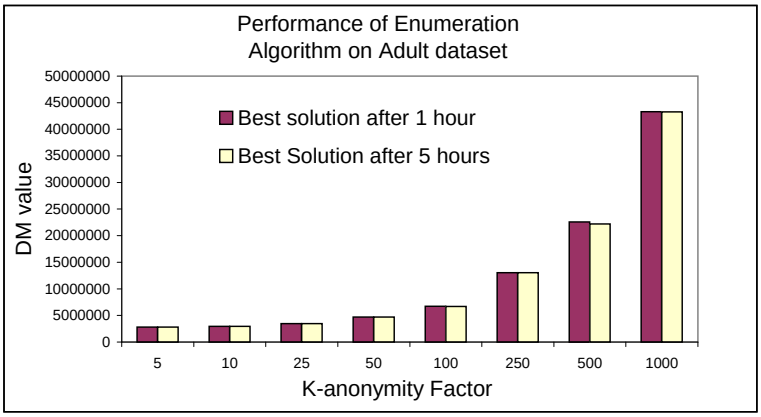

Figure 25: Performance, Adult Dataset

increases the chances of finding good solutions quickly. Previously, we introduced the following three different priority functions a) LowerBound (LB) b) Cost of the metric ($Cost$) and c) $LB/Cost$ and stated their rationale. Fig 24 shows the result for a run on the adult dataset optimizing the DM metric using all the three priority functions(keys) under consideration. LB priority function performs the best. A good solution at the highest level of the PET does not necessarily lead to a good solution at the lower levels. Therefore, $Cost$ priority function does not do as well the LB priority function. In the case of $LB/Cost$ priority function, the Cost of the metric ($Cost$) dominates the value and the priority function follows nearly the same search pattern as the $Cost$ priority function.

One interesting fact about the enumeration algorithm is that it does not need to run for a long time to get good solutions. In Fig 25 we have plotted the best solution found after 1 hour and after 5 hours for a run on the adult dataset optimizing the DM metric. It is quite clear from Fig 25 that the metric value of the solution found after 1 hour is very close to the metric value of the solution found 5 hours. This characteristic is found in almost all of our experiments.

In summary, the following are the significant results of our experiments;

- The enumeration algorithm has found optimal solutions for up to 14 splits under 30 mins.
- The enumeration algorithm performs better than greedy in all experiments and performs significantly better than greedy for higher values of k (usually more than 100), when constraints are not applied.
- For lower values of k (less than 100), greedy comes close to solutions found by the enumeration algorithm, when constraints are not applied.
- When constraints are applied, enumeration algorithm outperforms the greedy techniques for all values of k.
- The enumeration algorithm picks close to optimal solutions under an hour.

7 Summary & Conclusion

In this paper, we explored the problem of computing optimal k-anonymization of multidimensional data using hierarchical partitioning of the space. The approach we took was one of a complete search using systematic enumeration of the solution space. We have also developed a neat adaptation of the algorithm for categorical attributes, though we could not describe it in this paper due to space constraints and the interested reader can refer to [20]. To

our knowledge, there is no work that proposes a technique to systematically enumerate all the hierarchical partitioning of a multidimensional space.

In order to make the search for the optimal solution feasible in this huge space of solutions, we employed pruning heuristics to reduce the search space to a manageable size. Our enumeration tree exhibits a nice spatial locality property that allows us to make tight lower bound estimates efficiently by considering local distribution of the data in any region of interest.

Subsequently, we propose a flexible priority queue based algorithm that implements a prioritized search to get to good solutions quickly. In fact, this algorithm can execute in two modes, where one is directed towards finding an optimum solution while the other mode optimizes for finding solutions within a target approximation ratio.

We did extensive experimentation using a variety of combinations of the cost metrics and constraints motivated by some popular measures of privacy and information loss in privacy-preserving data mining applications. In our experiments besides characterizing the performance of our enumeration algorithm, we also carry out exhaustive comparison with some basic greedy heuristics and in the process discover some interesting trends that deviate significantly from what was predicted previously. In the this paper, we report results using the popular DM metric as the cost and k-anonymity, l-diversity and length restrictions as constraints. We also give performance results by varying the priority queue parameters like the priority generating functions and maximum allowed size of the queue.

Other related work & future directions: Finally, we note that while this paper is centrally motivated by the need for privacy preservation for data publishing, the problem of generalization based data anonymization is in many ways similar to the problem of optimal histogram construction and other related data partitioning problems. A wide variety of similar optimization problems have been studied earlier in different contexts e.g., query optimization, image compression, parallel computing etc., where similar families of partitionings have been considered [15, 16, 17, 18]. All these applications could also benefit from the current work. In fact, the turn many approximation algorithms proposed in them could perhaps be adapted for the class of anonymization problems we mentioned here. Although we did not pursue these issues in the current work, these remain as some attractive avenues to explore in the future.

References

[1] Vijay S. Iyengar. Transforming Data to Satisfy Privacy Constraints. In the proceeding of SIGKDD'02 Edmonton, Alberta, Canada.

[2] Roberto J.Bayardo, Rakesh Agrawal. Data Privacy Through Optimal K-anonymization. In the Proceedings of ICDE 2005.

[3] Kristen LeFevre, David DeWitt, Raghu Ramakrishnan. Mondrian Multidimensional K-Anonymity. ICDE 2006

[4] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, Muthuramakrishnan Venkitasubramaniam. L-Diversity: Privacy Beyond K-Anonymity. ICDE 2006

[5] Bentley, J., L. Multidimensional binary search trees using associative searching. Communication of the ACM, 18(9):509-517, Sept. 1975.

[6] L.Sweeney. Achieving K-anonymity privacy protecting using generalization and Protection. International Journal on Uncertainity, Fuzziness and Knowledge-Base Systems, 2002.

[7] P.Samarati. Protecting respondents identities in microdata release. IEEE Transactions on Knowledege and Data Engineering, 2001.

[8] Pierangela Samarati, Latanya Sweeney. Protecting Privacy When Disclosing Information: K-anonymity and its Enforcement through Generalization and Suppression.International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, Vol. 10, No. 5 (2002) 571-588

[9] UCI Machine Learning Repository. http:\\kdd.ics.uci.edu

[10] Benjamin C. M. Fung, Ke Wang, and Philip S. Yu. "Top-Down Specialization for Information and Privacy Preservation", ICDE 2005, Tokyo, Japan, April 5-8, 2005

[11] LeFevre, K., DeWitt, D., Ramakrishnan, R. "Incognito: Efficient Full-Domain K-Anonymity", In Proc. of ACM-SIGMOD 2005, Baltimore, MD, USA.

[12] Winkler, W., E. "Using Simulated Annealing for k-anonymity", Research Report Series (Statistics #2002-7), U.S Census Bureau 2002.

[13] Willenborg, L., Waal, T., D. "Elements of Statistical Disclosure Control", Lecture Notes in Statistics, 155, Springer-Verlag, New York.

[14] Aggarwal, C., Yu, P. "A Condensation Approach for Privacy Preserving Data Mining", EDBT 2004, Heraklion, Crete, Greece.

[15] Anily, S., Federgruen, A. "Structured Partitioning Problems", Operations Research Vol. 39, Issue 1, 1991, pp. 130 - 149.

[16] Muthakrishnan, S., Poosala, V., Suel, T. "On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity and Applications", In Proc. of the 7^{th} International Conference on Database Theory, 1997.

[17] Berman, P., Dasgupta, B., Muthukrishnan, S. "Exact Size of Binary Space Partitionings and Improved Rectangle Tiling Algorithms", In SIAM J. Discrete Math, Vol. 15., No. 2, pp. 252-267.

[18] Muthukrishnan, S., Suel, T. "Approximation Algorithms for Array Partitioning Problems", Journal of Algorithms 54, 2005, pp. 85-104.

[19] Kifer, D., Gehrke, J., Bucila, C, Walkher, W., "How to Quickly Find a Witness", PODS 2003, pp 272-283.

[20] Hore., B., Jammalamadaka, R., C., Mehrotra, S. "Systematic Search for Optimal k-Anonymization", UCI-ICS 2006, tech-report .

A Proof of Correctness of Algorithm "Enumerate"

Here, we present the proof of theorem A.1 of section 3.2.

Theorem A.1 *Algorithm* **Enumerate(D,S)** *systematically enumerates all distinct hierarchical partitions of the multidimensional space D generated using splits from the given split set S.*

Specifically, we prove the following two properties of the enumeration algorithm: (i) **Completeness**: It correctly generates all the distinct hierarchical partitions of the multi-dimensional space using the given set of splits. (ii) **Uniqueness**: It does so without duplication, i.e. there is a one-to-one correspondence between the nodes in partition enumeration tree (PET) and the set of hierarchical partitions of the space. The proof outline is given below.

1. First, we look at a simplified version of the algorithm **Enumerate** (figure 6) wherein the constraint checking is step (line 3) is simplified to only check compliance with Constraint 1. We call this the **MultiEnumerate** algorithm (figure 26). That is, it generates a multi-way tree of all timestamped KD-trees constructible using the given set of splits which comply with Constraint 1 (i.e. left subspace of a KD-tree node has been partitioned before the right subspace). Recall that a single hierarchical partition may be denoted by several different KD-trees. Since the simplified algorithm generates all KD-trees representing the same partition, we call it the **MultiEnumerate** algorithm. Therefore, the set of hierarchical partitions induce an equivalence relation over this set of KD-trees, where each equivalence class represents a distinct hierarchical partition of the space. We will use the method of induction to prove the following claim:

   Claim A.1 *Algorithm* **MultiEnumerate** *enumerates all members of each equivalence class, therefore ensuring completeness.*

2. Second, we look the effect of introducing the check for compliance with Constraint 2 into algorithm **MultiEnumerate**. The "Detect-Legal-Split" (DLS) routine in algorithm **Enumerate** (figure 6) first checks for Constraint 1 which is already implemented by **MultiEnumerate**. Therefore we only concentrate on the effect of enforcing compliance with Constraint 2 by the DLS routine. We explain in detail, how avoiding "out-of-sequence splits" removes all duplicates and retains exactly one "copy" of each hierarchical partition. Specifically we will prove the following claim.

Claim A.2 *DLS returns TRUE for exactly one member from each equivalence class induced by the set of hierarchical partitions. This unique KD-tree is the only "legal" representative of the hierarchical partition corresponding to the class (this KD-tree is called the "partition tree" for that partition).*

3. Lastly, we make the following observation.

   Observation A.1 *Algorithm* **Enumerate** *generates new partitions by adding a single split at a time to an existing partition. As a result, legal KD-trees (corresponding to child nodes) can only be derived from a parent KD-tree that is legal (i.e. complies with Constrain 1 and 2).*

   From the above observation, we are guaranteed completeness by simply retaining legal KD-trees at any stage in the enumeration process. This allows us to safely drop all trees derived from a legal KD-tree that do not meet the DLS criteria.

Combining the three arguments made above, it follows that algorithm **Enumerate** (in figure 6) generates all distinct hierarchical partitions of the space using the set of splits and does so without any duplication. We now prove the claims in item 1 and 2 above, the claim in item 3 is clear from the arguments provided there.

A.1 Proof of Claim 1

The **MultiEnumerate** algorithm is given below in figure 26. We use induction on n, cardinality of the set $Seq_{[n]}$ of splits and prove that **MultiEnumerate** generates the complete set of KD-trees denoting the set of hierarchical partitions.
Note that we only need to prove the correctness for those instances of **MultiEnumerate** where the input parameter $T(r)$ denotes a degenerate KD-tree, i.e. $T(r)$ denotes an undivided subspace. We will denote such a tree by T_ϕ. Making the hypothesis for the more general case where $T(r)$ can be any KD-tree unnecessarily complicates the proof procedure, hence we make the simpler hypothesis which suffices. The proof by induction is given below.

For $n = 1$, the algorithm generates the enumeration tree containing two nodes, one denoting the undivided space and the other denoting the partition having just one split that dividing the space into two subspaces. Now we make the induction hypothesis for all values of n from 1 to k:
Induction hypothesis: For all values of $n = 1, \ldots, k$ Algorithm **MultiEnumerate** $(D, Seq_{[n]}, r, T_\phi)$ generates the enumeration tree rooted at r, consisting of all distinct timestamped KD-trees that are constructible on

the domain space of data set D, using zero or more splits from the set $Seq_{[n]}$. Additionally each tree generated complies with Constraint 1.

MultiEnumerate$(\mathbf{D}, \mathbf{Seq}, \mathbf{r}, \mathbf{T(r)})$
Output: Construct the enumeration tree
(rooted at r) of all KD-trees
extending $T(r)$ and comply with
Constraint 1
BEGIN
1) **For Each** leaf $l \in T(r)$ s.t. $l \models$ Constraint 1
2) **For Each** available split $s_l \in Seq$ at l
3) Generate new child node c
4) Generate KD-tree $T(c) \leftarrow T(r) \cup s_l$
5) Add pointers $r \rightarrow c$; $c \rightarrow r$
6) **MultiEnumerate**$(D, Seq, c, T(c))$
7) **End For**
8) **End For**
9) **Return** r
END

Figure 26: Generating the complete set of KD-trees using n splits

Induction step: To prove that **MultiEnumerate** generates the complete set of timestamped KD-trees when called with any split set of size $k+1$.

Proof of induction step: Consider the following instance of the algorithm: **MultiEnumerate** $(D, Seq_{[k+1]}, r, T_\phi)$. Any such instance of the **MultiEnumerate** algorithm can be re-written in a non-constructive but equivalent form as the new algorithm **InductiveEnumerate** shown in figure 27. Note that the two algorithms are completely equivalent for the class of instances of **MultiEnumerate** that we are interested in (i.e. where $T(r) = T_\phi$ always). Note that equivalent algorithm does not require the inputs r and $T(r)$ and outputs the complete set of KD-trees instead of an enumeration tree (hence we call it non-constructive). The key observation is that the split that is chosen first cuts across the whole space, therefore it cannot be used again and hence can be dropped from the set of available splits in both the recursive calls to the function. The algorithm introduces the first split and calls itself recursively on the space to the left and right of the first split with a split set of size one less than the one in its input. Therefore the proof follows from our inductive hypothesis and the equivalence of the two algorithms **MultiEnumerate** and **InductiveEnumerate**.

A.2 Proof of Claim 2

From here onwards, we will assume that all KD-trees comply with Constraint 1 unless otherwise stated. The first part of the DLS routine checks

InductiveEnumerate$(\mathbf{D}, \mathbf{Seq_{[k+1]}})$
Output: The set $TREES$ of all KD-trees
BEGIN
1) $TREES \leftarrow \{T_\phi\}$
2) **For Each** split $s_i \in Seq_{[k+1]}$
3) $\mathfrak{T}_{left} \leftarrow$ **InductiveEnumerate** $(D_{left(s_i)}, Seq_{[k+1]} - \{s_i\})$
4) $\mathfrak{T}_{right} \leftarrow$ **InductiveEnumerate** $(D_{right(s_i)}, Seq_{[k+1]} - \{s_i\})$
5) **For Each** $T^l \in \mathfrak{T}_{left}$
6) Add 1 to the timestamp of each node in T^l
7) **For Each** $T^r \in \mathfrak{T}_{right}$
8) Add **MAX** {timestamp($node \in T^l$)} to timestamp of each node in T^r
9) Generate $T_{new} \leftarrow T_\phi$
10) $T_{new}.root.split \leftarrow (s_i)$
11) $T_{new}.root.timestamp \leftarrow [1]$
12) $T_{new}.root.leftchild \leftarrow T^l$
13) $T_{new}.root.rightchild \leftarrow T^r$
14) $TREES \leftarrow TREES \cup \{T_{new}\}$
15) **End For**
16) **End For**
17) **End For**
18) **Return** $TREES$
END

Figure 27: Re-writing MultiEnumerate for inductive proof

compliance with Constraint 1 and since that is assumed to be taken care of, the phrase "DLS returns true" will be used to imply compliance with Constraint 2 (i.e. splits at nodes are in-sequence or not).

Nature of DLS routine and the enumeration process: Observe that the DLS routine, invoked at any node t of a KD-tree T checks whether split $s(t)$ at t is out-of-sequence with the split $s(a)$ at any ancestor a of t. T is considered legal (i.e. T is a partition tree) if and only if the DLS routine returns true for each node in T. Since algorithm **Enumerate** generates new trees by extending existing partition trees by splitting a single partition block at a time, it is only necessary to invoke DLS once for every new partition generated from the parent node in the enumeration tree. This also implies that there can be no partition tree (i.e. legal KD-tree) that is derived from an "illegal" parent KD-tree and therefore only legal KD-trees need be generated and retained during the entire enumeration process.

Parent-child switching of splits: We now introduce an operation called "parent-child switch" which can be performed between a node and its children (or a child, it will become clear soon) in a KD-tree under certain conditions. The important property of such an operation is that, they generate a new KD-tree that denotes the same partition, but simply changes the sequence in which the cutting splits are introduced in some subspace.

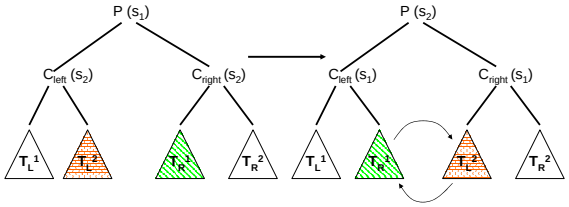

Figure 28: Parent-child switch for orthogonal splits (case 1)

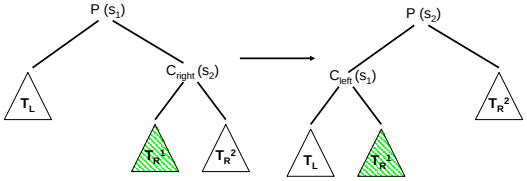

Figure 29: Parent-child switch for parallel splits (case 2)

We give the definition below and provide an example to illustrate the switch operation.

Definition A.1 Parent-child switch: *There are two distinct cases in which a switch can be made:*

Case 1) In a KD-tree T, a node p and its two children c_{left} and c_{right} have node splits such that $s(c_{left}) = s(c_{right}) = s_2 \neq s(p) = s_1$, then a parent-child switch (or simply a switch) refers to the atomic operation that switches the positions of splits s_1 with s_2, followed by switching the positions of the right subtree of c_{left} with the left subtree of c_{right}. Note that visualizing geometrically, this case holds when the two splits are orthogonal to each other (i.e. along two different dimensions (attributes)). This case is illustrated in figure 28.

Definition A.2 *Case 2) This corresponds to splits s_1 and s_2 that are parallel in space. Say $s(p) = s_1 \neq s(c_{right}) = s_2$ (when s_2 is a split to the right of s_1 in the space), then a switch refers to the atomic operation that "left rotates" the nodes p and c_{right}. That is, c_{right} becomes the parent with split s_2, p becomes the left child of c_{right} and positions of the left subtree of c_{right} is switched with that of the right subtree of p. The "right rotate" operations is symmetric. This case is illustrated in figure 29.*

Two part proof: We give a constructive proof and show that each equivalence class of KD-trees always has exactly one tree that satisfies the DLS routine at all of its nodes. The proof has two part: 1) We show that for any KD-tree with an out-of-sequence pair of splits, say $< (s), (s') >$ such that

(s) is the split introduced earlier, we can always derive a new tree using a sequence of parent-child switch operations such that it represents the same hierarchical partition and has (s') as the earlier of the two splits. Specifically we will show using straight forward induction that all conflicts (i.e. pairs of out-of-sequence splits) can be resolved for any given KD-tree, thereby generating a conflict-free KD-tree. 2) We show there can be at most one such tree for every equivalence class.

Proof of the first part: The proof is based on the following three observations, first of which we have already stated above.

Observation A.2 *Any parent-child switch operation is "partition preserving". That is, the KD-tree before and after a switch is made, represents the same hierarchical partition.*

Observation A.3 *A parent-child switch operation between the node p and its child node(s) never generates a new conflict with any ancestor of p, i.e. never generates new out-of-sequence pairs with split at an ancestor*[12] *of p.*

Observation A.4 *If s^* is the split at a KD-tree node n that forms a cut of the subspace denoted by some ancestor a of n, then s^* is also a cut across the subspace denoted by each node on the path $a \rightsquigarrow n$.*

Now look at the **MultiEnumerate** algorithm, in each iteration it starts with a legal KD-tree (i.e. conflict free) in the enumeration tree, call it the parent tree and generates a new candidate tree by adding a single split to some leaf of the parent tree. One only needs to check if there is a conflict between this split added at the newly formed node and the split at each of its ancestor nodes. We use induction as mentioned above to show that a conflict free tree exists for the given partition.

It is easy to see that when the parent tree has one node (i.e. one root node and two leaves), a conflicting split at a newly split node can be resolved using a switch operations with the root. Let us assume that all conflicts can be resolved for a KD-tree with up to k nodes. Therefore we can always have a legal KD-tree with k nodes. Now say a new node n_{new} is generated thereby making it a tree with $k+1$ nodes. Now if the highest ancestor a of this node, with which it conflicts is a node other than the root (i.e. $a \neq root$), by induction hypothesis we can always resolve the conflict in the subtree rooted at a since it is a tree with strictly $\leq k$ nodes. From observation A.3, the conflict can be resolved without introducing any new conflicts with any ancestor of a. Therefore there exists a legal KD-tree representing the new partition. Else if, the new node conflicts with the root of the tree,

[12] A switch operation may generate new conflicting pairs in either or both of the subtrees rooted at p.

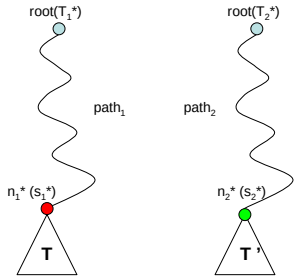

Figure 30: Uniqueness of the conflict-free KD-tree

using observations A.2, A.3 and A.4, we can always resolve the conflict with the root node, which will leave us with at most two "conflict-prone" subtrees (left and right subtree of root node), which can again be made conflict free using our induction hypothesis. Hence the first part is proved.

Proof of the second part: Assume we are given two randomly chosen KD-trees T_1 and T_2 belonging to the same equivalence class. Now, using parent-child switch operations let T_1 and T_2 be transformed into the conflict-free trees T_1^* and T_2^* respectively. Now, since these trees represent the same hierarchical partitions, they will have the same number of leaves and in fact there is a 1-to-1 correspondence of the set of leaves in T_1^* to those in T_2^*. Let us assume that these two final trees are distinct, then we can find at least one pair of leaves (corresponding to the same partition block in each tree) such that the sequence of splits along the root-to-leaf path in one tree is different from the other (see figure 30). Let us call these paths $path_1$ and $path_2$. Since these two are not the same, let n_1^* and n_2^* be the first nodes in each path, where the splits do not match, say denoted by (s_1^*) and (s_2^*). But both n_1^* and n_2^* denote the same subspace and the splits chosen at these nodes are cuts of this subspace. And now, since one of the two splits (s_1^*) or (s_2^*) has a higher priority than the other, only the path with a node-split of a higher priority could be a valid path in a conflict-free tree. Thereby contradicting our initial assumption that T_1^* and T_2^* are distinct. Hence we prove that at most one conflict free tree is there for each equivalence class.

Now the proof of Claim 2 is complete.