Summarizing User Action Sequences with Data Analysis

Exploiting Past Behaviour to Predict Future Actions

by

Bertrand Yilun Low

B.Sc., The University of British Columbia, 2003

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

October, 2007

Abstract

In the never-ending pursuit to enhance user interaction with computer systems, generating useful summaries can be highly beneficial. Summaries can provide the analyst with a map of past user behaviour that can aid in predicting future user actions. Tasks, such as forwarding an email or publishing and updating webpages, are composed of many individual user actions - as such, we view each task as a tree with the leaves of the tree representing the goal(s) of the task. We present a framework for modeling user actions as Navigation Trees and summarizing them into Summary Trees. The Summary Trees can be used to help streamline subsequent user action by acting as a guideline to semi-automate tasks. Using the concept of coverage and varying the number of attributes considered, we show how the quality of a Summary Tree can be adjusted. We also discuss five algorithms that approach summarization differently, compare their advantages and disadvantages, and provide an experimental study to empirically examine their individual characteristics.

Table of Contents

List of Tables

List of Figures

List of Programs

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Laks V.S. Lakshmanan, for his expert guidance, encouragement, and understanding throughout the course of my study. Laks' academic enthusiasm has been a constant source of inspiration. His comments and input have been instrumental toward the completion of this thesis. Thank you.

Many thanks also to Professor Rachel Pottinger who provided invaluable third-party feedback and in doing so, helped to make this work more accessible and relevant.

I have had the privilege of meeting a lot of great people in this journey. They have kindly shared their time and knowledge with me. They happen to be very intelligent too. I would like to thank Terence Ho, Xiaodong Zhou, Wesley Coelho, Wendy Hui Wang, Holly Kwan, Elaine Chang, Shaofeng Bu, and Timothy Chan for their friendship.

Finally, no amount of words can completely describe my sincere gratitude to my father, Rolland, my mother, Siok Hah, and my sister, Dorothy. Thank you all for your support and love in my endeavors.

God bless you all richly.

Long Live the UBC Database Lab!

For family and friends who laugh with us in the light, hold us in the darkness, and bring meaning to the breathing

Chapter 1

Introduction

As technology continues to advance, the amount of data a user deals with grows dramatically. Improvements in technology mean cheaper and larger hard drives, and a barrage of new software and files that contribute to a jungle of information the user has to navigate through. Even as the size of personal desktop-resident data is growing, the amount of remote data reachable by the user is also expanding quickly. The continuous growth of the Internet and the advent of file-sharing and remote-desktop-connection software yield a vast landscape of data available. Directory structure filing (i.e. creating the appropriate folders and placing relevant files in them) and browsing can be unsatisfactory and frustrating when dealing with such large amounts of data. Search tools (for both online and desktop use) aim to aid the user find desired data - however, search is only useful when the user remembers some feature of the data (e.g. keyword, date of creation/modification, author). Searching can be initiated by the user - as in the case of Google and Yahoo search engines - or, as in the case of some current research [10, 20], be initiated by the computer system. By capturing the current user context (e.g. email message) and identifying important features such as keywords and metadata, systems can proactively find other data that might be of interest to the user [10].

The research presented here purposes to enhance user interaction with a computer system by analyzing and summarizing past user actions. By looking at past behaviour, we can cut the search space by predicting what data the user might be interested in. By understanding the user, we can also anticipate the user's intentions and provide means to help the user complete a task faster and with less frustration. In this thesis, we present a framework for modeling user tasks, along with techniques to build summaries. Each summary can then be used as a tool to understand user behaviour or as a guideline for streamlining future user actions by way of semi-automated scripts.

1.1 Motivation

Typical users have certain tasks that they regularly perform on their computers. Examples of recurring tasks include downloading picture files from a digital camera into the hard drive, forwarding emails, and publishing blogs. Although the task itself does not change, the underlying data and frequency of recurrence varies. For example, the task of forwarding an email is common to most users but the data (i.e. the email being forwarded, the recipients, additions to the message body, etc.) usually differs. New photos are transferred between camera and disk, blogs are published with different subject lines, body, and at different frequencies (e.g. daily, weekly). The extent to which the data differs also varies. Politically-charged blogs, though individually unique in content, may have the same principal themes. We have certain expectations of the type of photos found in the camera of a sports photog-

rapher. The task itself, the data involved, and the amount of detail we wish to describe (i.e. summarize) said task, all affect the quality of the summary we can provide. We are able to make statements such as "These photos are of the NHL Vancouver Canucks" or, "These photos are of Game 7 of the '94 Stanley Cup Finals between the Vancouver Canucks and New York Rangers" (being more *specific*), or "These photos are of a sporting event" (being more *general*).

Consider the task of publishing course notes (a recurrent activity for academic instructors). A possible scheme could be:

1. Open presentation software and create course slides.
2. Export presentation to HTML format.
3. Export presentation to PDF format.
4. Export presentation to Word format.
5. Add links to the course notes (in the various file formats) on the course webpage. The various file formats allow students to choose the version they are most comfortable with or capable of viewing.

Suppose that our user performs the task of publishing course notes on a daily basis. Each day's slides would have different content and filename but we note that with appropriate software that is capable of exporting to the desired formats, steps 2-5 do not require further user input. In fact, if the system can recognize this daily task (by summarizing past instances of this task), it is possible to semi-automate the task so that for future course notes publishing, the user would just have to provide the system with the master

copy of the course slides created by the presentation software and the system then automatically goes about creating the various format versions and adds the appropriate links on the course webpage.

Given that users perform certain tasks routinely, our goal is to summarize the past instances of the user-performed task so that the summary can effectively aid future interactions with the personalized system. By creating a summary that captures past user action sequences, we allow for opportunities to create scriptable workflows that semi-automate recurring tasks, saving the user from performing tedious actions that do not require active participation (e.g. publishing course notes in other formats once the master copy is selected). The summary may also be used to anticipate the user's next action (by consolidating the number of times particular actions were taken) and predict the user's ultimate intention (i.e. the goal of performing a sequence of actions).

1.2 Contributions

The following contributions appear in this thesis:

1. A Framework for Modeling User Actions - We introduce the idea of the Navigation Tree.

2. Algorithms for Summarizing User Actions - We describe 5 different methods (known as Generalization Algorithms) for creating summaries and the concepts involved in doing so.

3. Experimental Evaluation of the Generalization Algorithms - We per-

form experiments on clickstream data to gain some insight on the performance and utility of the summaries created by the different algorithms.

1.3 Outline

In Chapter 2, we state the problems tackled in this research and describe the Navigation Tree model for user action representation, including the intuitive and technical challenges behind creating a useful summary. Following this, Chapter 3 describes the Generalization Algorithms for creating Summary Trees. We provide some related work and concepts in Chapter 4. Experimental results of the algorithms can be found in Chapter 5. Finally, Chapter 6 closes this thesis with our conclusions from this research.

Chapter 2

Problem Statement

This thesis work attempts to answer the following problem:

> Given n Navigation Trees, how can we create a useful Summary composed of n-or-less Summary Trees?

In order to answer the above problem, we need to explain what we mean by *Navigation Tree* and what constitutes a *useful* Summary. The following sections of this chapter discuss how we tackle these issues.

2.1 Navigation Trees

A Navigation Tree is a directed rooted tree, with edges pointing away from the root. Each node in the Navigation Tree represents a single Data Object (i.e. file). Edges represent the relationship between the two objects it connects - an edge is labeled with the user's reason for connecting the source Data Object to the target (See Subsection 2.1.2 for more information).

Consider the example Navigation Tree shown in Figure 2.1.

Figure 2.1 illustrates how the Navigation Tree models the task of publishing course notes on the course webpage. Each path in the tree represents the sequence of user actions that compose one part of the task. Also observe that each path is by itself a valid task - the user can choose to only publish

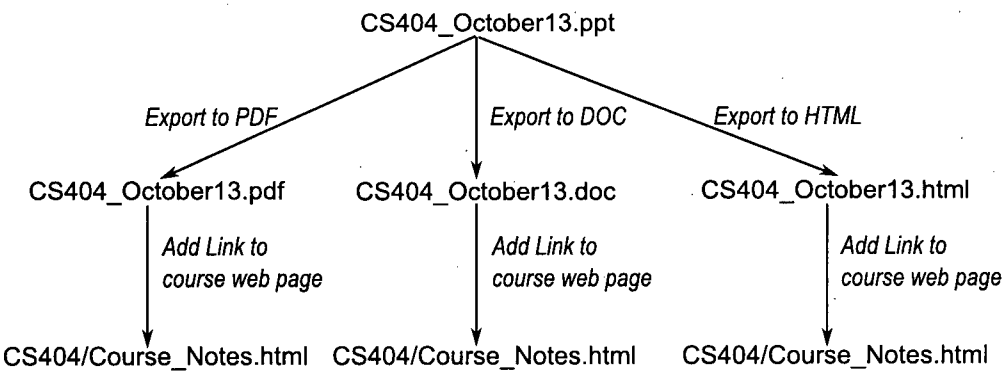


Figure 2.1: Navigation Tree Example. Modeling the task of course notes publishing with a Navigation Tree.

the `.pdf` version of the slides. The leaves of the tree can be understood as the various goals of the task. In this example, the goals are to publish CS404 notes on the course webpage for October 13 in the `.pdf`, `.doc`, and `.html` formats. A Navigation Tree models a past instance of a user action sequence. A Summary describes a set of Navigation Trees with varying detail, as analogous to the greater amount of detail that a longer - as opposed to shorter - summary of a literary work is capable of capturing. Later in Section 5.1, we show how Navigation Trees can be used to model clickstream data.

2.1.1 Modeling Tasks as Trees

We model tasks as trees because although actions are initiated one after the other (i.e. sequentially), the actions are not necessarily sequential from the logical perspective of the user. Consider the action sequence in Figure 2.2.

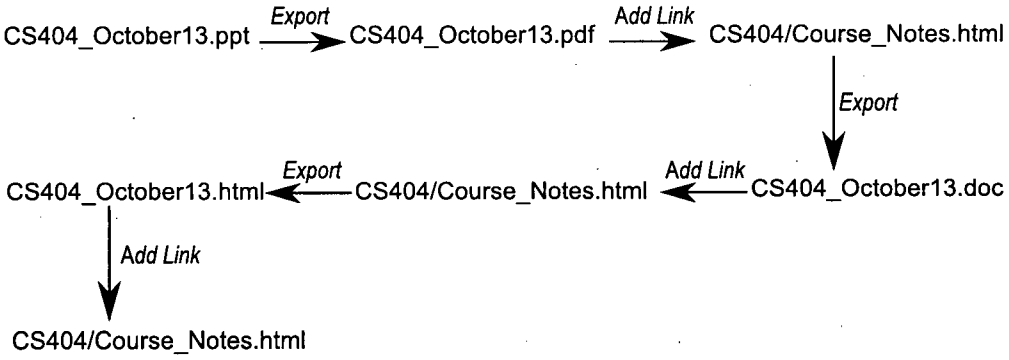


Figure 2.2: User Task Sequence Example. Modeling the task of course notes publishing using a sequence.

The Figure 2.2 sequence does not capture the logic that although the

export actions occurred in sequence, they really stem from the .ppt slideshow file and each export action can be performed independently of each other.

2.1.2 Edge Labels

We do not provide a strict definition of the *relationships* between objects as it is not the main focus of this work. However, we envision the edges being flexible enough to capture "event"-oriented relationships (as in the example of the export and add link edges in Figure 2.1) and also "characteristic"-oriented relationships. Event-oriented relationships refer to the class of actions that the user invokes to modify the source and/or target Data Object. Characteristic-oriented relationships apply to properties of the Data Objects themselves that piqued the user's interest in the target Data Object after viewing the source Data Object. An example of a "characteristic"-oriented relationship is connecting two objects because they contain the same keyword (this would represent the user opening the target Data Object because it contains the same keyword as the source). Another example would be connecting two email objects because they come from the same *sender* (representing the action of the user opening the target email object because it comes from the same *sender* as the source email). Table 2.1 lists possible relationships that might be of benefit to capture.

It is important to note that while edge labels help the system understand the user's logical flow better, they are not necessary. Edge labels provide more detail regarding the user's actions by discriminating between different contexts via which a user may access a Data Object (e.g. did the user select the Target because it was authored by the same person as the Source

Event-oriented

`Add Link` - Target links to the Source via a hyperlink.

`Compress` - Target is a compressed version of the Source.

`Export` - Target is an exported version of the Source.

`Open` - the "default" relationship if the relationship is not known or not specified by the user.

...

Characteristic-oriented

`Linked From` - Target was linked from the Source via a hyperlink.

`Same Author` - Source and Target share the same author.

`Same Directory` - Source and Target are located in the same directory.

`Same Keyword` - Source and Target share the same keyword(s) in either file name and/or content.

`Same Sender` - Source and Target emails share the same sender.

`Same Send Date` - Source and Target emails share the same "send date".

`Same Subject Title` - Source and Target emails share the same subject title. This could signify an email "thread" that the user is interested in.

...

Table 2.1: Example Relationships Between Data Objects

or because the Source and Target were located in the same directory?). By knowing the reason a user selected a Target after viewing the Source, we can make a more informed suggestion to the user when attempting to predict a future action. For example, if we are able to recognize that the user opened a `.doc` file because it shared the same author as the `.pdf` document that was viewed earlier, we may be able to present the user with a dynamically-updated list of `.doc` files that share the same author as any future `.pdf` document the user views. Event-oriented relationships can be incorporated into a personalized system as a semi-automated script that triggers when the system recognizes the user initiating a recurring task. Course notes publishing (Figure 2.1) is an example of a recurring task for which the system, upon realizing the user's intent to publish, can automatically create the various format versions (`.pdf`, `.doc`, `.html`) of the slides *and* add their corresponding links to the course notes webpage, without the user having to do more than identify the master copy (the `.ppt` presentation). However, if a relationship is not determinable or not defined by the user, the "default" `Open` label can be imposed and the summarization techniques in this thesis are still applicable on the Navigation Trees; the edge labels merely serve to enhance the resulting summary.

2.1.3 Attributes

Data Objects contain a lot more *attributes* than simply its file name; e.g. most files contain metadata like size, file type, and the creation, modified and accessed date. Further, there are certain attributes that are only applicable to certain file types. Video and audio files have various bitrates and codecs

associated with them whilst emails have fields such as *sender*, *recipients*, and *subject*.

It is also possible for the edge labels to contain attributes. For example, we could attach Time Invoked and Date Invoked attributes to the edge labels in the Navigation Tree in Figure 2.1 (representing the time and date the `export` and `add link` actions occurred). Note that our Data Object method of modeling files allows us to capture the inter-object relationships by treating the edge labels (and any corresponding edge label attributes) as attributes of the target Data Object. For example, the `CS404_October13.pdf` Data Object in Figure 2.1 would contain an additional "edge label" attribute with the value "`Export to PDF`" since that is the relationship stemming from its source (i.e. `CS404_October13.ppt`). Our focus is not in attaining the Navigation Trees, but rather, once attained, what we can do with them.

2.2 Summary Trees

We now introduce the concept of the Summary Tree and what we consider to be a *useful* Summary. Like Navigation Trees, a Summary Tree is a directed rooted tree, with edges pointing away from the root; the difference lies in that a node in a Summary Tree may not represent a single unique object. Whereas a Navigation Tree models a past instance of a user action sequence, the goal of a single Summary Tree is to represent, or "cover", a group of Navigation Trees, whilst retaining enough detail (i.e. attributes and edge labels) to be useful in predicting the user's future actions. We define a Summary, Ψ, to be a collection of Summary Trees. Consider the Navigation Trees, N_1 and

N_2, shown in Figure 2.3.

Each Navigation Tree corresponds to past actions taken by the user to publish a set of notes (one set for each of October 13 and October 15). Note that only the `file name` attribute is displayed. A possible Summary composed of one Summary Tree is shown in Figure 2.4. Due to the lack of detail (only the root and leaf Data Object's `file type` is captured), the Summary Ψ_1 in Figure 2.4 provides a very simplistic view of the user's actions ("The user opened a PPT file and exported it to some other file and then added the link to an HTML page"). Ψ_1 does not offer satisfactory predictive ability because it assumes that the user wants to perform `export` and `add link` actions to *all* `.ppt` files created.

Another possible Summary, Ψ_2 would be the collection of the 2 Navigation Trees (N_1, N_2) themselves (i.e. $\Psi_2 = \{S_1, S_2\}$ where $S_1 = N_1$, $S_2 = N_2$). Ψ_2, however, is too *specific* and is not helpful in anticipating future user actions (unless the user *only* deals with those very 9 Data Objects when performing the course notes publishing task). Unlike Summary Ψ_1, Ψ_2 is too precise and *covers* nothing more than the N_1 and N_2 task instances. Intuitively, some "middle-ground" between Ψ_1 and Ψ_2 is desirable.

Summary Ψ_3 (see Figure 2.5) is more precise than Ψ_1 but at the same time covers more than just N_1 and N_2. Ψ_3 recognizes that the user created a `.ppt` file with the common keyword CS404 (relating to the course CS404), exported that file to the `.pdf`, `.doc`, and `.html` formats, and finally added links to the exported files on the `CS404/Course_Notes.html` file. Ψ_3 allows the system to anticipate that the next time the user creates a slideshow for course CS404, the user would also want to create `.pdf`, `.doc`, and `.html`

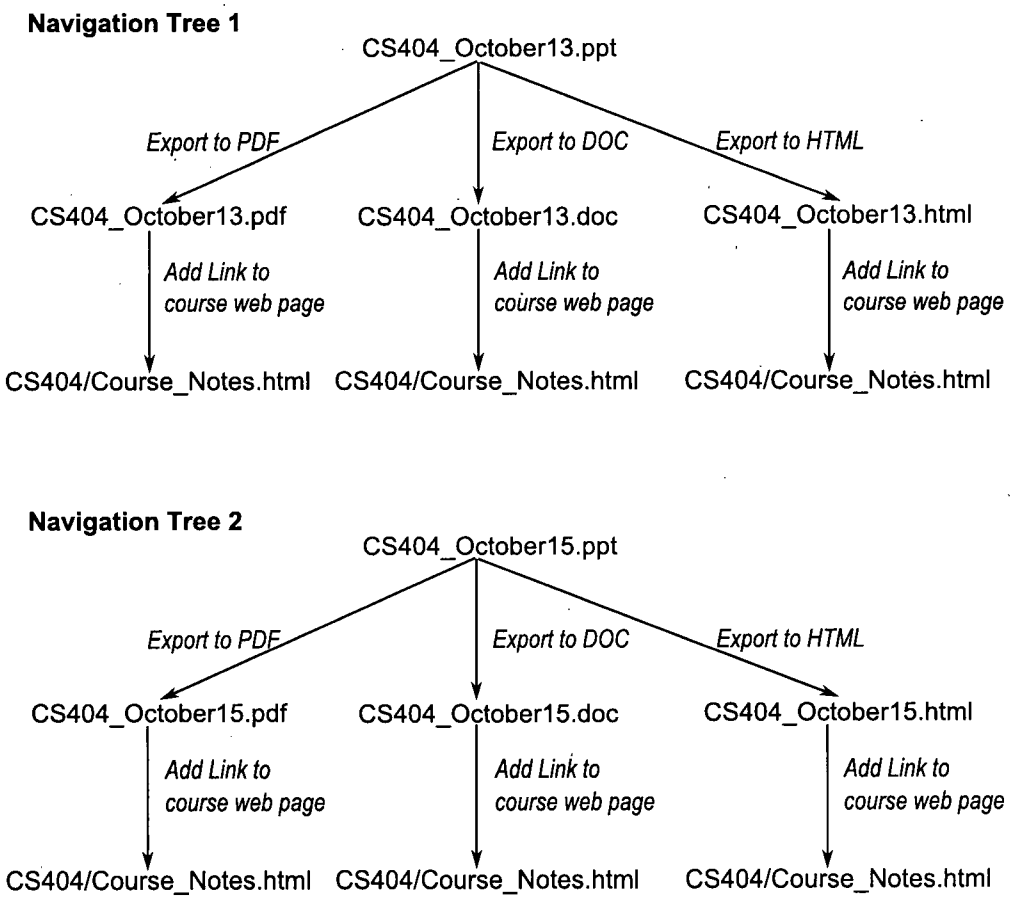


Figure 2.3: Navigation Trees for Publishing Course Notes. Here are 2 Navigation Trees, each representing a separate instance the user performed the task of publishing course notes for the course CS404

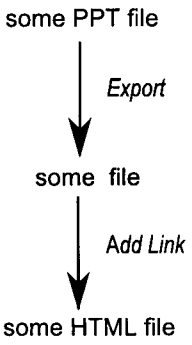


Figure 2.4: Summary Ψ_1 for Publishing Course Notes (Too General). Summary Ψ_1 summarizes the Navigation Trees in Figure 2.3 - however, it is too *general* and does not provide many details about the Data Objects other than the `file type`

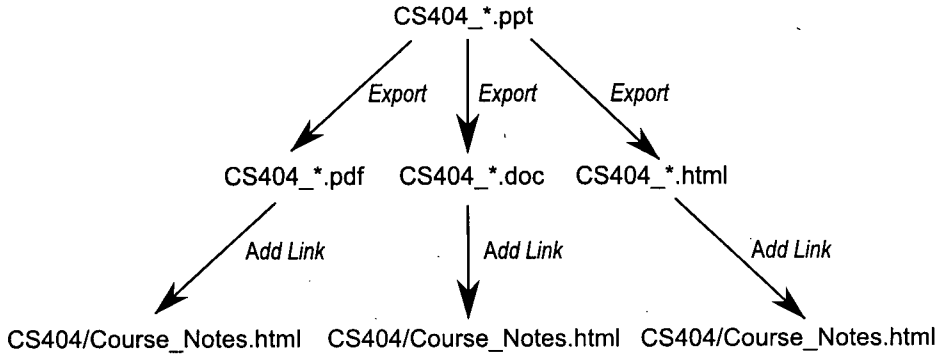


Figure 2.5: Summary Ψ_3 for Publishing Course Notes (Middle-ground). Summary Ψ_3 summarizes the Navigation Trees in Figure 2.3 and provides some precision w.r.t. N_1 and N_2 while being flexible enough to anticipate future user action sequences on similar files

counterparts to add to the course webpage - in doing so, the system can help the user by automating the export and add link actions.

The utility of a Summary greatly depends on the Navigation Trees it summarizes - if the Navigation Trees reflect tasks that are dissimilar (e.g. publishing course notes vs. forwarding email), the Summary may not be able to concisely describe the Navigation Trees without becoming either *too specific* or *too general*. Hence, we assume that Summary Trees summarize Navigation Trees that are similar in order that patterns and generalizations can be observed. To this regard, users can help by dictating the set of Navigation Trees to be summarized. Ideally, only Navigation Trees representing similar tasks are summarized into one Summary Tree (see Figure 2.3). However, note that it is *not* a requirement for the Navigation Trees to be isomorphic. By exploiting past instances of user-initiated tasks, a useful Summary can be created to semi-automate future tasks in order to enhance the user's experience with the system.

2.3 Definitions

Navigation Tree - A Navigation Tree, N, is a directed rooted tree with each node representing a single Data Object and edges representing the relationship between the source and target object (Section 2.1).

Summary Tree - A Summary Tree, S, is a directed rooted tree with each node representing one or more Data Objects and edges representing the relationship between the source and target object

(Section 2.2).

Summary - A Summary, Ψ, is a collection of Summary Trees $\{S_1, S_2, \ldots, S_m\}$

Given a Summary Tree S and a Navigation Tree N, $N \preccurlyeq S$ denotes "N generalizes to S" (see Subsection 2.3.1).

We say that Summary Ψ summarizes a collection of n Navigation Trees $(N_1, N_2, \ldots, N_n)$ if $\Psi = \{S_1, S_2, \ldots, S_m\}$ such that $m \leq n$ and $\forall N_i$ (where $i = 1, \ldots, n$), $\exists!$ $S \in \Psi$ such that $N_i \preccurlyeq S$.

2.3.1 Generalization

Recall that Data Objects (re: files) have certain attributes associated with them (Subsection 2.1.3). Attributes can be metadata common to all Data Objects (e.g. name, size, creation date, modified date) or characteristics specific to the Data Object's *file type* (e.g. `.doc`, `.jpg`, `.avi`, `.mp3`, `.html`). We use the *file type* attribute to atomically separate Data Objects; that is, Data Objects of a particular *file type* should only be compared to other Data Objects of the same *file type*. Similarly, a Data Object of a certain *file type* can only be generalized to a Multi-Data Object of that same *file type*. Each attribute value (other than *file type*) can be generalized to a more general, less specific, value. As an example, the name attribute of the `CS404_October13.ppt` and `CS404_October15.ppt` data objects in Figure 2.3 can be generalized to `CS404_*.ppt` by replacing the date with the regular expression `*`. The `CS404_*.ppt` data object now semantically covers *all* `.ppt`

files that begin with the "CS404_" prefix. In general, each attribute has a corresponding generalization hierarchy (which can be user-specified or built using attribute-related ontology knowledge) that describes the set of possible values for that attribute. The creation-date attribute of a Data Object can be described by a date range hierarchy as in Figure 2.6

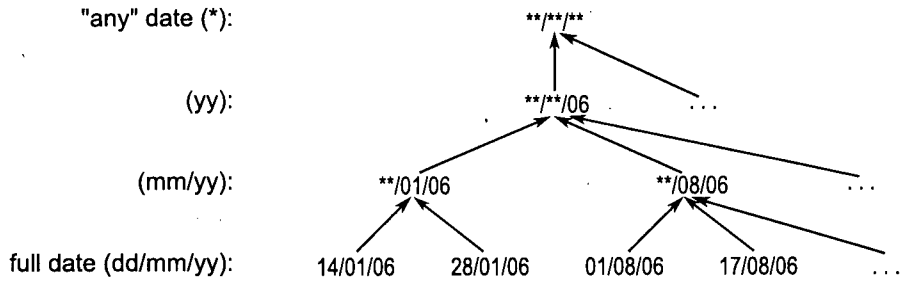


Figure 2.6: Generalization Hierarchy for the *Date* Attribute. The creation-date attribute can take any of the values given in the generalization hierarchy. Note that only a very small subset of all possible values are illustrated here.

Values become more general as we move up the generalization hierarchy with the top-most value representing "any" value, analogous to *suppressing* the attribute in classical relational database systems. Each attribute of any Data Object has a most-specific, ungeneralized value (i.e. one of the values on the lowest level of the generalization hierarchy). Each node in a Navigation Tree is "fully-specified" as each represents a single Data Object. The nodes in a Summary Tree are either fully-specified Data Objects or *generalized* Multi-Data Objects. A generalized Multi-Data Object has one or more of its attributes generalized and refers to one or more Data Objects. The most "general" any Data Object can become is to have all of its attributes (except

`file type`) fully generalized to the *any* value, thus representing all Data Objects of the same `file type`. For example, a fully-generalized `.ppt` Multi-Data Object would cover all `.ppt` Data Objects.

The idea of generalization applies to attributes, Data Objects, and Navigation Trees. If an attribute value, x, is a generalization (according to the generalization hierarchy) of another attribute value, y, then we say that $y \preccurlyeq x$. A Multi-Data Object, D_1, is a generalization of another (Multi-) Data Object, D_2, `iff` $\forall$ attributes $a_{D_1} \in D_1$, $\exists$ attribute $b_{D_2} \in D_2$, such that $b_{D_2} \preccurlyeq a_{D_1}$; we say $D_2 \preccurlyeq D_1$ (or equivalently, D_1 *covers* D_2). For a Navigation Tree N and a Summary Tree S, $N \preccurlyeq S$ `iff` $\forall$ Data Objects $D_N \in N$:

- $\exists!$ Multi-Data Object $D_S \in S$, such that $D_N \preccurlyeq D_S$ and
- $D_{N_{parent}}$ is the parent of $D_N \Leftrightarrow$ the Multi-Data Object generalization of $D_{N_{parent}}$ is the parent of D_S.

The Data objects in the Navigation Tree are generalized to Multi-Data Objects of the same depth in the Summary Tree. Each Multi-Data Object in the Summary Tree *covers* one or more Data Objects from the Navigation Tree. The statements "$N \preccurlyeq S$", "N generalizes to S", "S covers N", and "S summarizes N" are equivalent. Figure 2.7 shows an example of a Navigation Tree $N \preccurlyeq$ Summary Tree S.

2.3.2 Coverage Threshold

As illustrated in Section 2.2, the utility of a generated Summary depends on how wide a "target" it covers. A Summary that is too general (see Figure 2.4) covers too many possible Data Objects and leads to a poor understanding

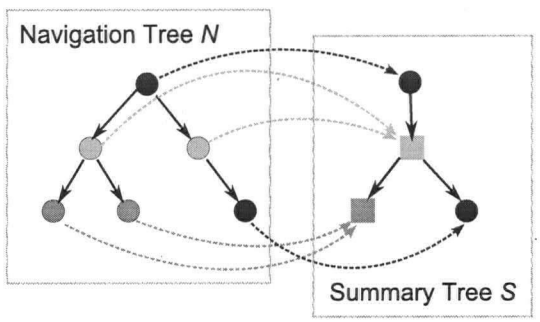


Figure 2.7: Navigation Tree $N \preccurlyeq$ Summary Tree S. The Data Objects of N are generalized into Multi-Data Objects in S

of the user's intentions (e.g. the user may not want to export and publish every `.ppt` slideshow created). However, a Summary that is too specific does not lend much to predicting a user's future actions (e.g. the user does not only publish the `CS404_October13.ppt` course notes, but other course notes as well). We introduce the notions of *Coverage* and *Coverage Threshold* to help quantitatively define how general or specific a Summary should be.

Coverage - the *coverage* of a Multi-Data Object X in a Summary Tree is defined as the number of Navigation Tree Data Objects that maps to it via a generalization μ (See Figure 2.8). The generalization μ takes as input a group of Data Objects and returns a single Multi-Data Object that covers each of the Data Objects. The ideal output goal of μ is to return the Multi-Data Object that retains the most attribute details. The algorithms presented in Chapter 3 uses generalizations of the Data Objects of Navigation Trees to find suitable Multi-Data Objects with which to construct the Summary Tree that is able to cover the Navigation Trees. We denote the coverage of X as $C(X) = |\mu^{-1}(X)|$

Coverage Threshold (CT) - given a collection of Navigation Tree Data Objects of the same `file type` (attained by capturing past user action sequences), each of whose parent maps to the same Multi-Data Object in the Summary Tree, the *Coverage Threshold* (CT) is the minimum percentage of Data Objects that must map to a single Multi-Data Object X. Let the Raw Coverage Threshold of X, $RCT(X) = (CT \times \#\text{Data Objects})$.

We claim that Multi-Data Object X satisfies the CT if $C(X) \geq$

RCT(X).

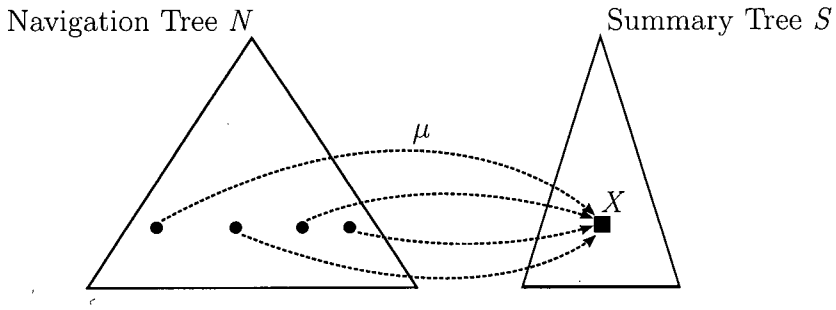


Figure 2.8: Coverage Described in Terms of Generalization Function Mapping. Each Data Object in N maps to one Multi-Data Object in S via a generalization μ

Figure 2.9 gives an example of how Coverage and Coverage Threshold relate to the summarization of Navigation Trees. The Summary Tree in Figure 2.9 is created with CT = 50%. At depth = 0, we have 3 `.ppt` Data Objects. Since CT=50%, and $\lceil 3 \times CT \rceil = 2$, each depth = 0 `.ppt` Multi-Data Object in Summary S must cover at least 2 or more Data Objects. However, since there are only 3 `.ppt` Data Objects at depth = 0, the corresponding Multi-Data Object in S would have to cover all 3 Data Objects (recall in Subsection 2.3.1 that a single Data Object can only map to one Multi-Data Object). Also note that at depth = 2, although there are 3 `.doc` Data Objects, the parent of one maps to a different `.pdf` Multi-Data Object than the parent of the other two. Hence, in S, only two `.doc` Data Objects end up as siblings (they are not generalized into a `.doc` Multi-Data Object since $\lceil 2 \times CT \rceil = 1$). Note that the Summary Tree is not unique - for example, at

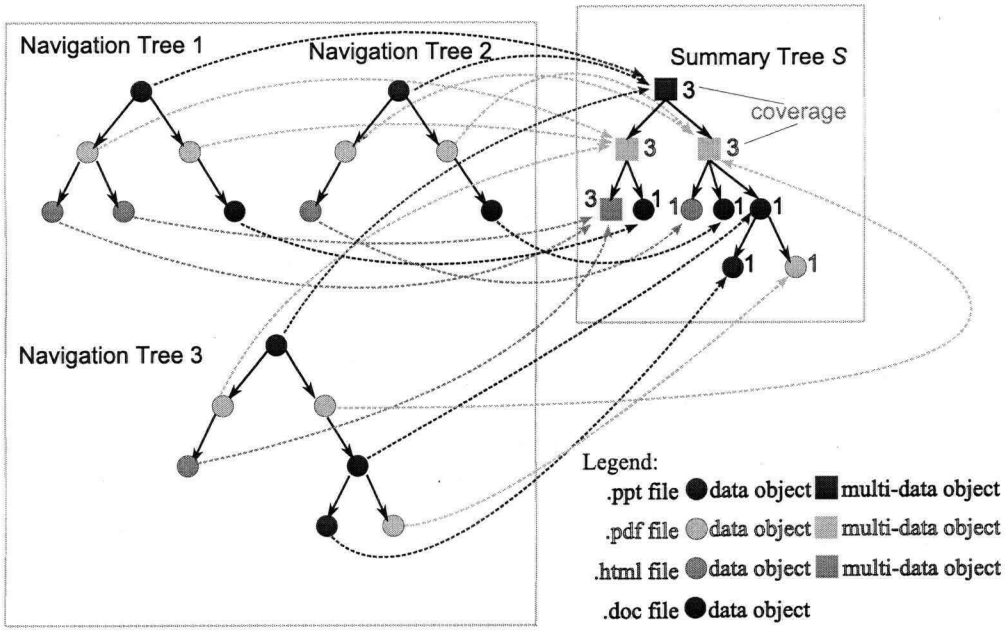


Figure 2.9: Example Applying Coverage and Coverage Threshold. The 3 Navigation Trees are summarized by Summary Tree S using a Coverage Threshold value of 50%. Note also that the Navigation Trees are not isomorphic

depth = 1, another Summary Tree $S\prime$ can be created by mapping a different 3-group combination of `.pdf` Data Objects to each of the `.pdf` Multi-Data Objects.

In general, the higher the Coverage Threshold and the more varied the Data Objects in the Navigation Trees are, the less specific the Summary Tree will be (i.e. the wider a target it captures) - this occurs because more Data Objects will have to map to the same Multi-Data Object. Hence, we can use the Coverage Threshold as a tool to control how general or specific the Summary Tree is, thus affecting the usefulness of the Summary Tree (See Section 2.2) - we seek some "middle-ground" between a "too-general" (high Coverage Threshold) Summary Tree and a "too-specific" (low Coverage Threshold) Summary Tree. Note that for any given Coverage Threshold, the more similar the Navigation Trees are to each other, both in structure and Data Objects, the more specific the Summary Tree will be; at this extreme end, if all the Navigation Trees are the same (i.e. $N_1 = N_2 = \ldots = N_n$) then the Summary Tree can be represented by any one of the Navigation Trees (i.e. $S = N_1 = N_2 = \ldots = N_n$).

We define *Thinness* as a measure for how wide a target a Summary Tree covers. Each Data Object in a Navigation Tree is mapped to a Multi-Data Object X in the Summary Tree as illustrated in Figure 2.8. Let Slack of X = Slack(X) = C(X) - RCT(X). We define the *Thinness* measure as follows:

$$\textbf{Thinness} = \Sigma_{X \in S}[Slack(X)]$$

For any given Coverage Threshold, the *thinner* the Summary Tree, the smaller the space of Data Objects it covers (i.e. the Multi-Data Objects

of a thinner Summary Tree will be more specific than those of a *fatter* Summary Tree). As illustrated by the Summaries in Figure 2.4 and Figure 2.5, a fat Summary is not desirable as it covers too many possible Data Objects and may result in over-anticipating the user. Similarly, a Summary that is too thin may not cover enough of the Data Objects that the user might be interested in. The best generalization likely lies between the fattest and thinnest Summary and probably varies from user to user. Thus, a personalized system that allows the user to vary the Coverage Threshold to fine-tune the thinness (i.e. coverage) of a resulting Summary would allow the most flexibility in meeting the user's needs.

2.4 Chapter Summary

The focus of the research presented here is not on how to attain Navigation Trees. However, we would like to know, when given n user-specific Navigation Trees, how to create a useful Summary Ψ composed of m Summary Trees ($m \leq n$). In this Chapter, we showed how Navigation Trees logically model tasks performed by the user. We also described the intuition behind creating a useful Summary that is not too specific (so as to not be able to predict future user actions on similar Data Objects as those in the Navigation Trees) and not too general (so as to wrongly predict user actions on Data Objects *not* similar to those in the Navigation Trees). The idea of generalization was introduced in this Chapter and we also showed how Coverage Threshold can be used to control the extent of detail with which a Summary Tree can describe a collection of Navigation Trees. A Summary S can be used to

understand user behaviour patterns, predict future actions, and streamline and improve the efficiency of interactions between the user and computer system.

Chapter 3

Summarizing Navigation Trees

Summary Trees are constructed by merging Navigation Trees together using generalization techniques. As mentioned in Subsection 2.3.1, the *file type* attribute allows us to distinguish between *mergeable* Data Objects - only Data Objects of the same *file type* are mergeable. Hence, given n Navigation Trees, we need to first group them by *file type* so that we only merge Navigation Trees with roots having the same *file type*. A *Typed* Navigation Tree Group is a group of Navigation Trees with the same root *file type*. Program 3.1 shows the pseudocode of the Merge algorithm.

Figures 3.1, 3.2, and 3.3 provide a partial walkthrough of the Merge algorithm. In Step 1 of the Merge algorithm, the Dummy Root effectively creates a Typed Sibling Group (TSG) consisting of all the roots of the Typed Navigation Trees (see Level 1 in Figure 3.1). The tree consisting of the Dummy Root and the Typed Navigation Trees is known as the Intermediate Summary. The Typed Sibling Group "`TSG 1`" and the Raw Coverage Threshold (RCT) is then passed to the *Generalization Algorithm* which then returns the representative Multi-Data Objects that *summarizes* the TSG. In our example, Figure 3.2 shows that the Generalization Algorithm mapped each of the Data Objects of `TSG 1` to one of two Multi-Data Objects (details of how the Multi-Data Objects in the Summary are determined can be found in the

Program 3.1: Pseudocode of Merge Algorithm for Summarizing *Typed* Navigation Trees

```
input: n Navigation Trees, Coverage Threshold CT
output: 1 Summary composed of m Summary Trees

1. Join Navigation Trees under Dummy Root to create
   Intermediate Summary
2. For each level (from top to bottom) under the Dummy Root
   in the Intermediate Summary:
  2a. Group Sibling Groups according to file type
  2b. For each Typed Sibling Group (TSG):
    2b.i. Calculate Raw Coverage Threshold
          (RCT = CT * |Typed Sibling Group|)
    2b.ii. Call Generalization Algorithm with arguments
           Typed Sibling Group and RCT
    2b.iii. Update Intermediate Summary with Merged Typed
            Sibling Group resulting from Generalization
      2b.iii.A. Appropriately update children of Merged TSG
          (i.e. the Children of Merged Nodes are now siblings)
3. Return Final Summary = Intermediate Summary
```

remaining sections of this chapter). The Intermediate Summary is then updated with the returned Multi-Data Objects, and the TSGs of the next level (Level 2) are considered (See Figures 3.2 and 3.3). At Level 2, we see from Figure 3.3 that there are multiple TSGs; for this particular example there are four TSGs (in general, there will be i TSGs). Each of the Level 2 TSGs (beginning with `TSG 1`) are then passed to the Generalization Algorithm in left-to-right order. The Generalization Algorithm will then return the Multi-Data Objects that cover the TSG it received as input and the Intermediate Summary is accordingly updated with said Multi-Data Objects. After all the Level 2 TSGs have been processed, the Merge Algorithm moves on to the next level (Level 3), and so on until all levels of the Intermediate Summary have been processed. The resulting Intermediate Summary is the final Summary returned by the Merge Algorithm. Note that the updating of any level of the Intermediate Summary creates new TSGs; that is, Data Objects that were not originally siblings might become siblings due to their parents being merged by the same representative Multi-Data Object (see Figure 3.2). The level-by-level merging of the TSGs in the Intermediate Summary creates a final Summary that is composed of m Summary Trees (each subtree below the Dummy Root represents a Summary Tree). The final Summary covers all the input Navigation Trees.

The number and detail-level of Representative Multi-Data Objects is determined by the Coverage Threshold (see Subsection 2.3.2) and the Generalization Algorithm. The goal of summarizing is to find the *thinnest* (as defined in Subsection 2.3.2) Summary that satisfies the Coverage Threshold. In the following sections of this chapter, we will present two different generaliza-

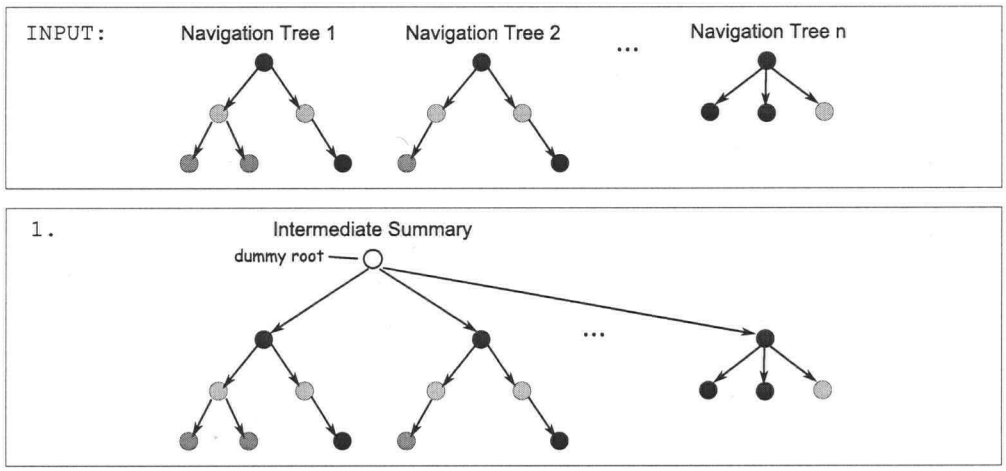


Figure 3.1: Example Illustrating Merge Algorithm Part 1 of 3. Step 1 of Merge Algorithm. Continued on Figure 3.2

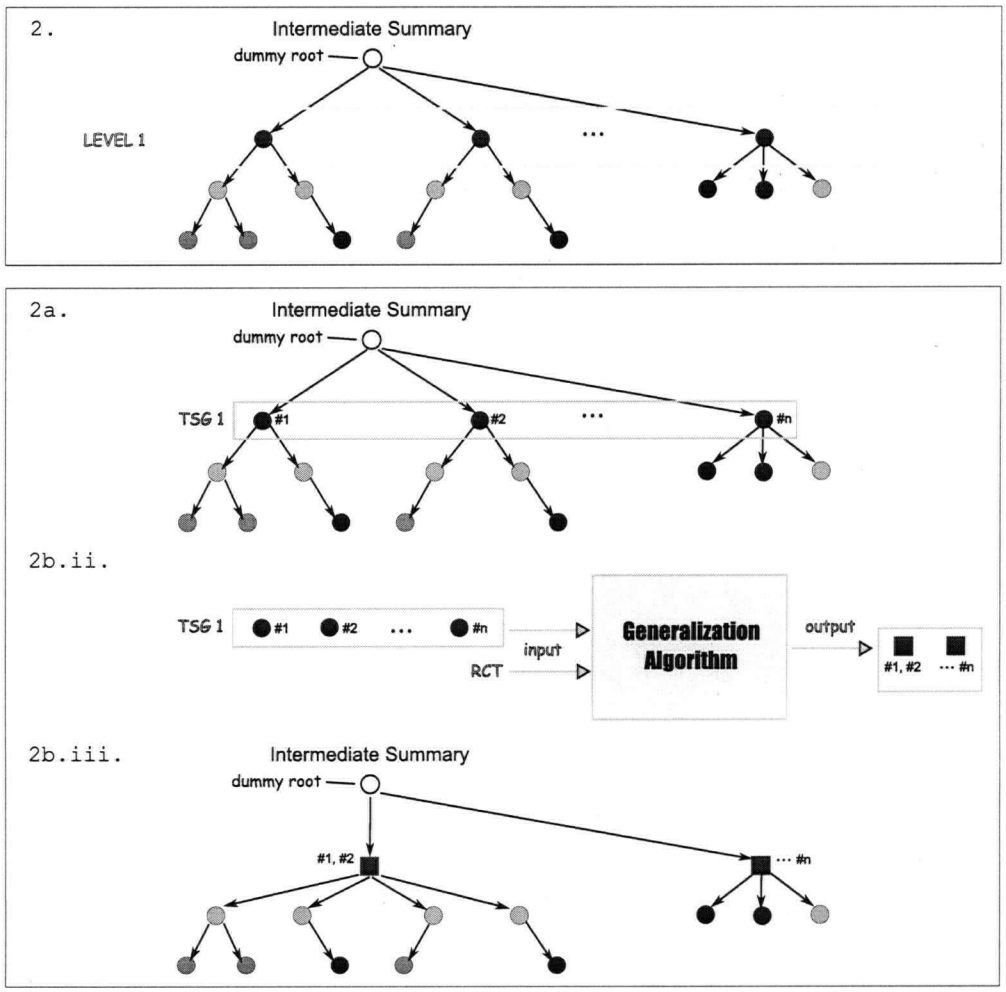


Figure 3.2: Example Illustrating Merge Algorithm Part 2 of 3. Continued from Figure 3.1. Step 2 of Merge Algorithm - Merge Level 1. Continued on Figure 3.3

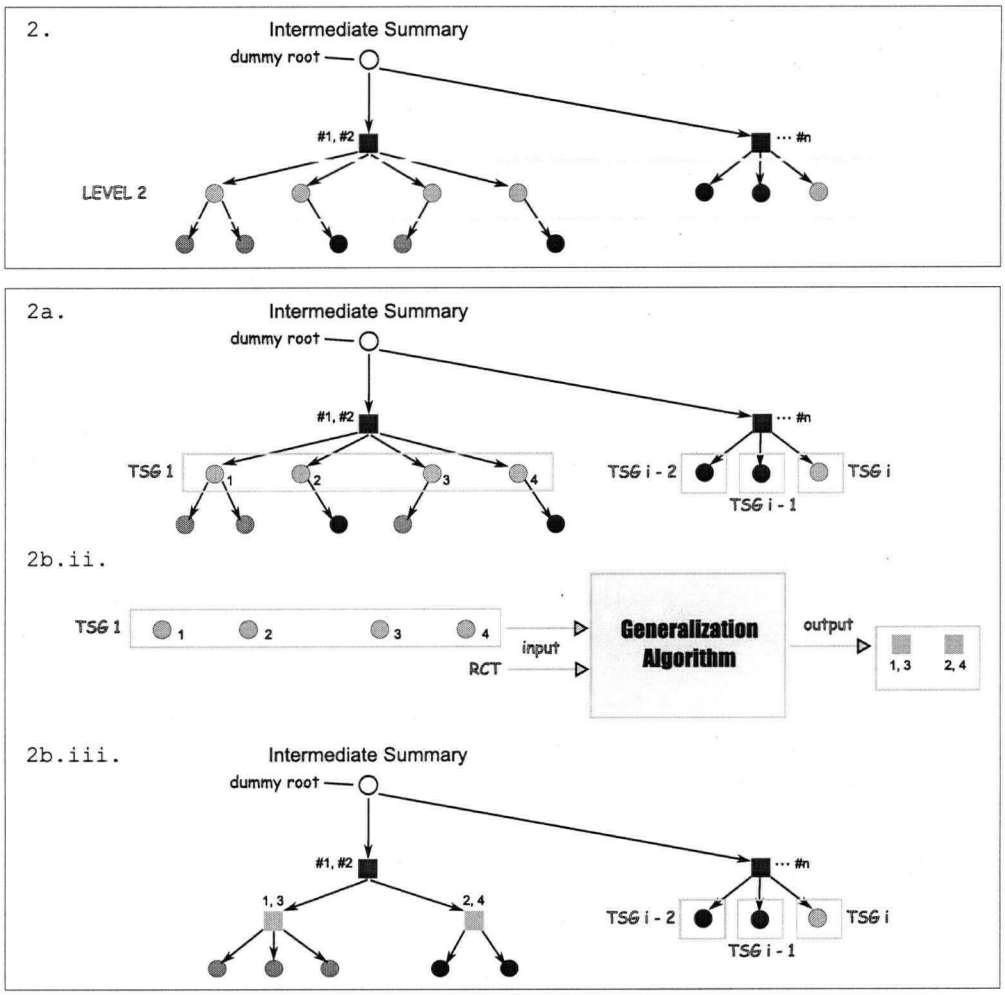


Figure 3.3: Example Illustrating Merge Algorithm Part 3 of 3. Continued from Figure 3.2. Step 2 of Merge Algorithm - Merge Level 2. Subsequent steps of the Merge Algorithm are not shown.

tion approaches used to merge Navigation Trees (Node-based Generalization Algorithms in Section 3.1 and Partition-based Generalization Algorithms in Section 3.2).

3.1 Node-based Algorithms

Consider a single Data Object (i.e. node in the Navigation Tree) and the many ways it can be generalized. Each Data Object has a number of generalizable attributes associated with it. Recall from Subsection 2.3.1 that we treat the *file type* as fundamentally unalterable. All other attributes can be generalized to some degree (see Figure 2.6 for example). While each attribute has its own generalization hierarchy, the "any" generalization (aka *suppression*, see Figure 3.4 and Section 4.1) can be applied to all Data Objects. For example, just as the Date Attribute of Figure 2.6 may be fully-specified (`14/01/06`), partially-specified (`**/01/06`, `**/**/06`), or completely generalized to "any" date (`**/**/**`), so can other attributes such as Author (full name, first or last name, "any" name) and Size (123KB, 1**KB, "any" size). In the rest of this section, we will only consider "any" generalizations.

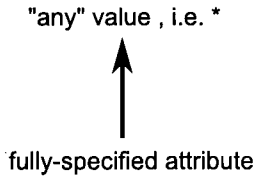


Figure 3.4: "ANY" generalization (aka suppression). The "any" generalization is applicable to all generalizable attributes

The various states of generalization for a single Data Object can be captured using a Lattice of Conditions (LOC). Figure 3.5 is an example of the node-based LOC for a text document. Note that the LOC can be significantly larger if hierarchical or range-based generalizations (e.g. Figure 2.6) are also considered. For the generalizations we consider, the number of generalized states per level for any node-based LOC is shown in Figure 3.6. Note that the higher (more general) the LOC one travels, the greater the coverage. Conversely, the lower (more specific) the LOC one travels, the smaller the coverage. The states found in node-based LOCs are composed of attribute/value pairs. Hence similarly "typed" Data Objects with the same attributes may still differ in their attribute values. The two Node-based Generalization Algorithms (Greedy and Enumerate All) use the LOC for selecting generalizations that satisfy a given coverage threshold.

3.1.1 Greedy

As illustrated by Figure 3.6, the LOC grows exponentially as the number of attributes increases. A Data Object (such as a Word document or mp3 file) can have upwards of 20 attributes (yielding a LOC with at least 1,048,576 nodes/generalization states). Thus, materializing the whole LOC is not desirable. In fact, because coverage is monotonically increasing with respect to the LOC's level number, it is not necessary to materialize the LOC completely. We can use the following logic to determine which portions of the LOC need to be materialized:

> Consider any node n (i.e. a generalized state) of the LOC. If the coverage of n satisfies the Raw Coverage Threshold, there is no

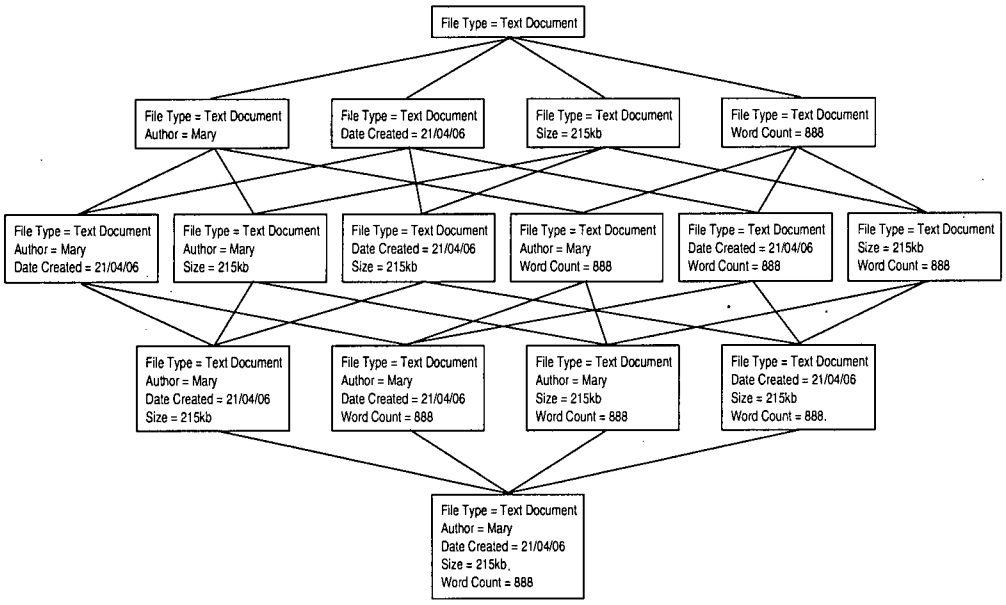


Figure 3.5: Example of a Lattice of Conditions. This Lattice is built for a text document Data Object. Note that only 4 of the 5 attributes of the Data Object are constrained since *file type* is immutable

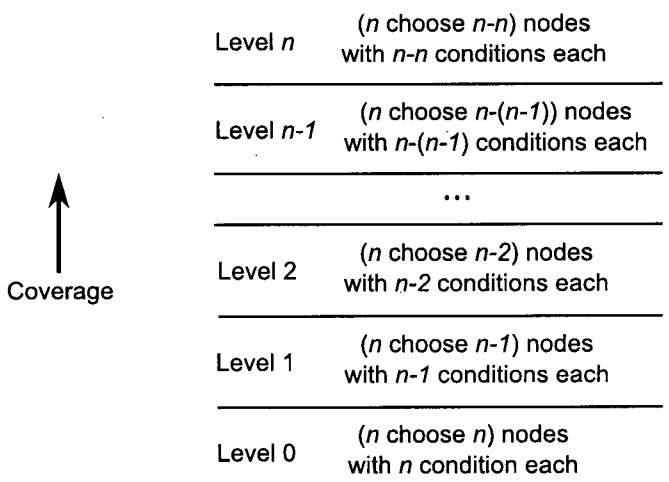


Figure 3.6: A General Lattice of Conditions. Coverage of a generalization in any LOC is proportional to the generalization's Level. The top and bottom of the LOC is the most general and the most specific respectively.

need to examine ancestors of n because their coverage cannot be lower than that of n. From our discussion of "best generalization" (Subsection 2.3.2), the generalization n is preferred over those of its ancestors because it is the most specific (i.e. thinnest) generalization that meets the coverage threshold. The only possible generalizations that might be preferred over n are those on the same level as n, and those on lower (more specific) levels than n. Any LOC node that satisfies the coverage threshold is *feasible*.

Thus, at each level of the LOC, we can label each node one of the following:

- *Under* - A node labeled *under* is not feasible.
- *Closest* - A node labeled *closest* is also not feasible but has the highest coverage among the infeasible nodes.
- *Above* - A node labeled *above* is feasible.
- *Thinnest* - A node labeled *thinnest* is also feasible but has the lowest coverage among the feasible nodes.

The above labels are each node's relative coverage with respect to the Raw Coverage Threshold. Note that there may be multiple nodes with the same label due to a "tie" in their coverage. An appropriate course of action can be taken for each node depending on its label: an ancestor a_n of a node n needs to be examined `iff` all descendants of a_n are infeasible. In other words, a single feasible descendant of a_n nullifies the need to check the coverage of a_n

- in this way, only necessary portions of the LOC need to materialized. Figure 3.7 summarizes the possible coverage labels for nodes and the appropriate respective actions.

As displayed in the worst case scenario of Figure 3.7, any ancestor of an *under* or *closest* node may contain the global *thinnest* node. Hence, in order to find the global thinnest node, the worst case scenario (each node is labeled either *under* or *closest*) forces us to materialize the whole LOC. The Greedy Algorithm avoids this complete materialization by selecting a local thinnest node. Given a Typed Sibling Group `TSG` and the Raw Coverage Threshold `RCT`, the Node-Based Greedy Generalization Algorithm will iteratively select a Data Object from the `TSG` that does not yet satisfy the `RCT` and generalize it using the LOC until the `RCT` is met. The algorithm uses the monotonically increasing property of the LOC to only materialize necessary portions. Program 3.2 displays the pseudocode for the Node-Based Greedy Generalization Algorithm.

The Greedy Algorithm generalizes a given Data Object by selecting a local *closest* node at each level of the LOC, beginning from level 0, until it reaches a local *thinnest* node. Only ancestors of the closest node selected at each level are materialized and their respective coverages checked. Note step `1h.`, which "force-merges" the remaining nodes in the `TSG` with `BestGen` if the number of remaining nodes is less than `RCT`. That is, in the case where it is not numerically possible for the remaining nodes to meet the `RCT`, these nodes are merged with the last `BestGen` node regardless of how similar they are to `BestGen`; apart from the `type` attribute, these nodes may very well be disjoint with `BestGen`. An example illustrating how the Greedy algorithm works can

Program 3.2: Pseudocode of Node-Based Greedy Generalization Algorithm

```
input: Typed Sibling Group TSG, Raw Coverage Threshold RCT
output: Merged Sibling Group MSG that satisfies RCT

MSG = {}

1. While there exists Node(s) in TSG {
1a.   Current Node = First Node of TSG
1b.   Remove Current Node from TSG
1c.   Let BestGen = Bottom Node of LOC (i.e. Current Node)
1d.   Let NextLevel = 1
1e.i. While (Coverage(BestGen) < RCT) {
1e.i.A. Let Gens = {all NextLevel generalizations of BestGen}
1e.i.B. BestGen = Gens[0]
1e.i.C. For(int i=1; i<Gens.size(); i++) {
          Let NodeI = Gens[i]
          If (Coverage(NodeI) > BestGen)
            { If (BestGen < RCT) { BestGen = NodeI } }
          Else If (Coverage(NodeI) >= RCT)
            { BestGen = NodeI }
        }
1e.i.D. NextLevel++
      }
1f.   MSG += BestGen
1g.   Remove from TSG all nodes covered by BestGen
1h.   If (TSG.size() < RCT)
        { ''Force-merge'' remaining TSG with BestGen }
   }
2. Return MSG
```

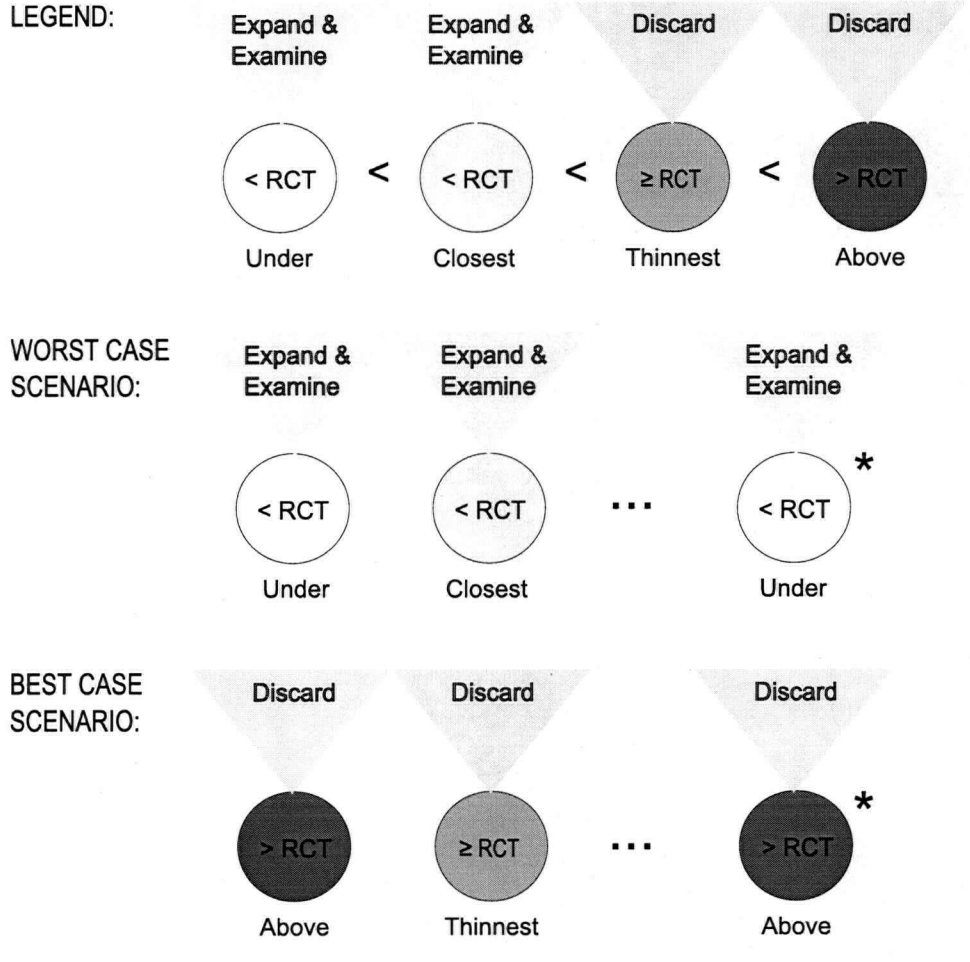


Figure 3.7: Summary of a Node's Relative Coverage and Appropriate Ancestor Action. The monotonic property of the LOC allows the determination of a node's relative coverage which in turn aids the Generalization Algorithm to determine which portions of the LOC need to be materialized.

be found in Figure 3.8 and Figure 3.9. The materialized portion of the LOC visited by the Greedy algorithm resembles a tree since it only expands one node per level. Hence, the complexity with respect to the number of visited nodes in the LOC $= n + (n-1) + (n-2) + ... + 1 = O(n^2)$.

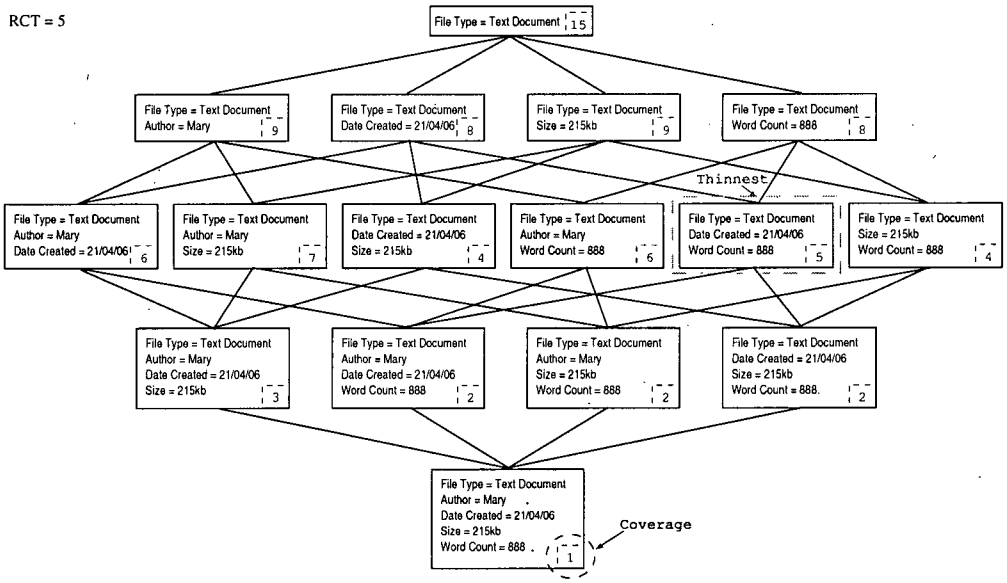


Figure 3.8: LOC with Coverages and Global Thinnest. RCT = 5

3.1.2 Enumerate All

Consider again the Greedy example in Figure 3.8. The Greedy algorithm works by traversing up the LOC starting at the bottom and iteratively selecting the *closest* node until it finds a local *thinnest* node. As illustrated in Figure 3.9, the Greedy algorithm does not select the global thinnest node on level 2 because the closest selection made on level 1 does not have it as one

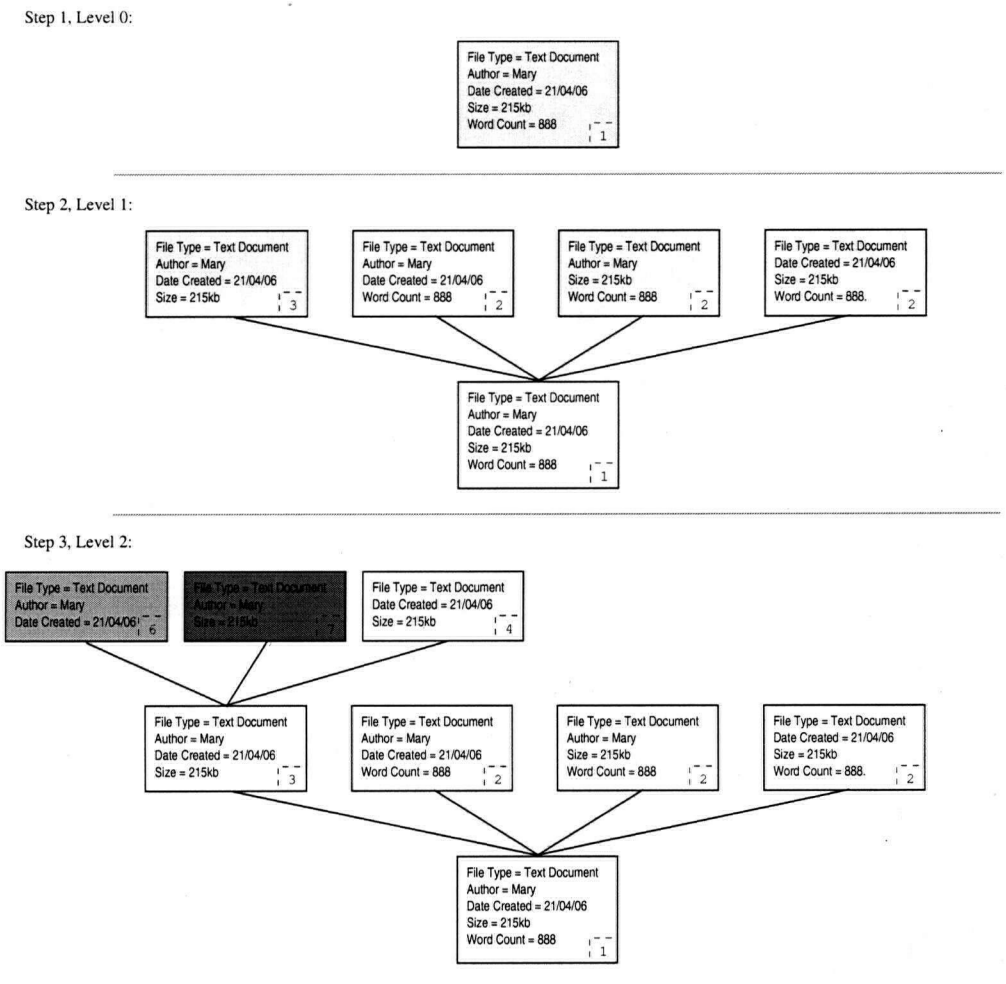


Figure 3.9: Example of Node-based Greedy Generalization Algorithm. This example illustrates how the Node-based Greedy Algorithm works given the LOC of Figure 3.8. Current generalizations being examined are coloured as per the relative coverage the node is labeled with (see Figure 3.7).

of its ancestors. The Node-based Enumerate All Generalization Algorithm finds the global thinnest node by materializing the whole level in consideration; its pseudocode is shown in Program 3.3. The key difference between the Node-Based Greedy and Enumerate All algorithm is in step `1e.i.A.`. Here, *all* `NextLevel` generalizations are considered, as opposed to only the `NextLevel` generalizations of the greedy-selected closest node from the previous iteration. The Enumerate All Generalization Algorithm is able to find the absolute thinnest LOC node in situations where its Greedy counterpart cannot due to Greedy selecting a closest choice whose ancestors do not include the global thinnest.

3.2 Partition-based Algorithms

The Node-based Algorithms of the previous section attack the problem of finding the thinnest summary from a micro point of view. Node-based algorithms examine one node of the TSG at any given time, without consideration for the other nodes. As a result, a poor choice of a TSG node to generalize can lead to a needlessly "fat" generalization of the entire TSG. Consider the example in Figure 3.10.

The selection of the "outlier" node (i.e. `A`) with which to generalize the remaining TSG nodes yields a non-optimal summary. Partition-based Algorithms attempt to minimize the effect such "outliers" have with TSG summarization by looking at all the TSG nodes simultaneously, hence adopting a more macro point of view. The basic idea behind these algorithms is to partition the TSG into groups that are as thin as possible with respect to a

Program 3.3: Pseudocode of Node-Based Enumerate All Generalization Algorithm

```
input: Typed Sibling Group TSG, Raw Coverage Threshold RCT
output: Merged Sibling Group MSG that satisfies RCT
MSG = {}

1. While there exists Node(s) in TSG {
1a.   Current Node = First Node of TSG
1b.   Remove Current Node from TSG
1c.   Let BestGen = Bottom Node of LOC (i.e. Current Node)
1d.   Let NextLevel = 1
1e.i. While (Coverage(BestGen) < RCT) {
1e.i.A. Let Gens = {ALL NextLevel generalizations}
1e.i.B. BestGen = Gens[0]
1e.i.C. For(int i=1; i<Gens.size(); i++) {
          Let NodeI = Gens[i]
          If (Coverage(NodeI) > BestGen)
            { If (BestGen < RCT) { BestGen = NodeI } }
          Else If (Coverage(NodeI) >= RCT)
            { BestGen = NodeI }
        }
1e.i.D. NextLevel++
      }
1f.   MSG += BestGen
1g.   Remove from TSG all nodes covered by BestGen
1h.   If (TSG.size() < RCT)
        { ''Force-merge'' remaining TSG with BestGen }
   }
2. Return MSG
```

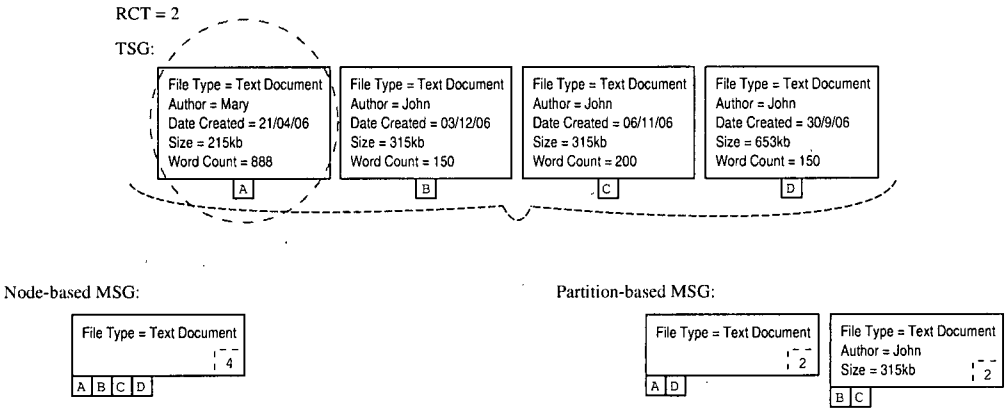


Figure 3.10: Node-based vs. Partition-based Algorithms. This example TSG illustrates the advantage Partition-based Algorithms have over their Node-based counterparts

RCT. Each of these groups are summarized by a representative Multi-Data Object. More formally, we say:

> $\forall$ group m of a given TSG, $\exists$ representative Multi-Data Object p such that
>
> $\forall$ Data Object $d \in m$, $d \preccurlyeq p$

Given a n-node TSG, we want to find $\lfloor n/RCT \rfloor$ representatives. Optimally, the TSG would be partitioned into equidepth "bins" with RCT number of items each. In general, a partition P is feasible if $\forall$ blocks $B \in P$, $|B| \geq RCT$. The Partition-based Algorithms presented in this section minimize the following cost-metric:

$$\text{Partition } P = \langle B_1, B_2, ..., B_n \rangle$$

$$\text{Cost}(P) = \sum_{i=1}^{n} |(|B_i| - RCT)|$$

In the Figure 3.10 example, a partition P_1 with blocks $B_{1_{P_1}} = \{\texttt{A, D}\}$ and $B_{2_{P_1}} = \{\texttt{B, C}\}$ is just as desirable as a partition P_2 with $B_{1_{P_2}} = \{\texttt{A, C}\}$ and $B_{2_{P_2}} = \{\texttt{B, D}\}$ because $Cost(P_1) = Cost(P_2) = 0$

In order to capture partition candidates when examining multiple nodes simultaneously, we introduce the concept of the Lattice of Attributes (LOA). Recall from Section 3.1 that the Lattice of Conditions is composed of nodes with attribute/value pairs. In contrast, the LOA is composed only of attribute combinations. See Figure 3.11 for an example of a LOA.

The top node of the LOA is the "any" partition which has no constraints on any of the available attributes, effectively grouping all the Data Objects of the TSG into a single block. The bottom node on the LOA has all attributes

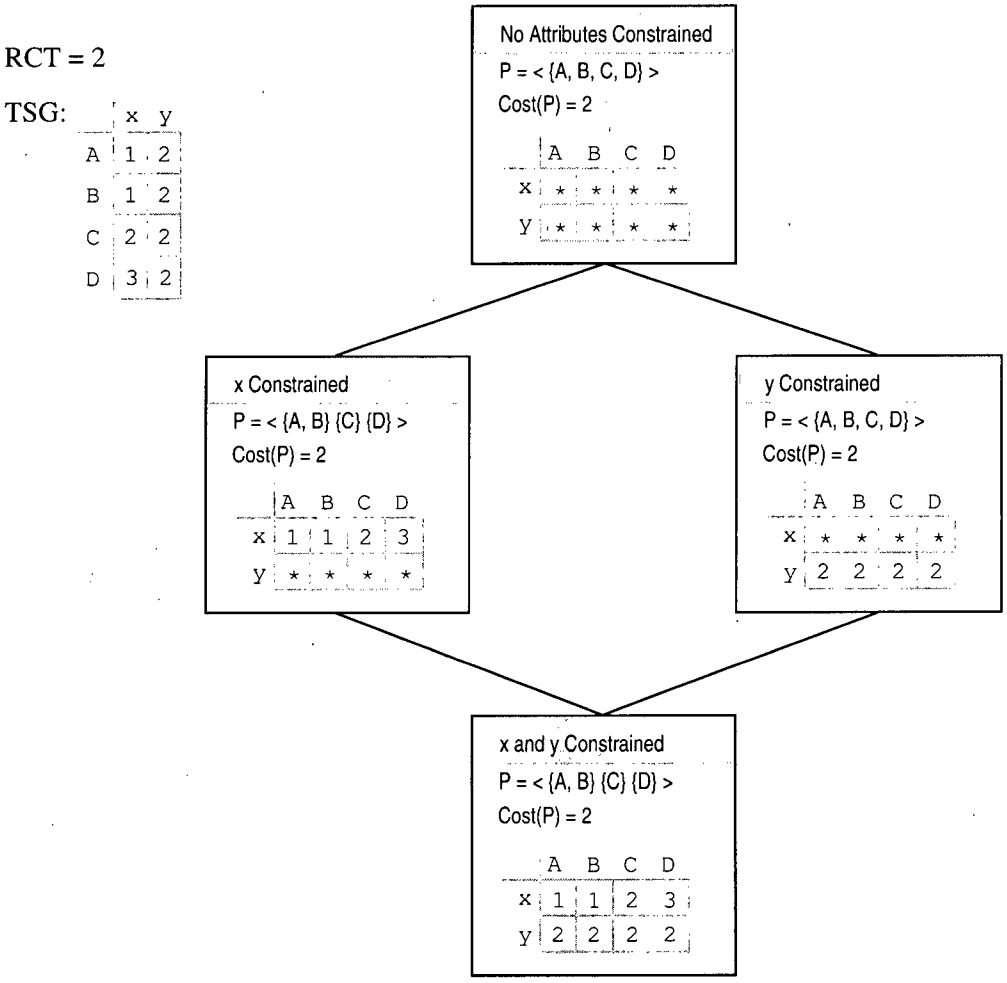


Figure 3.11: Example of a Simple Lattice of Attributes. The Lattice of Attributes (LOA) is based on different combinations of constrained attributes.

constrained and partitions the TSG into blocks of smallest possible size. If the Data Objects contain a unique ID attribute (or uniquely-identifying combination of attributes such as the conjunction of file name and path attributes) the bottom node of the LOA would be composed of single-item blocks. The LOA is traversed by Partition-based Generalization Algorithms in order to find a feasible partitioning of a TSG.

3.2.1 Greedy Bottom-Up

The Greedy Bottom-Up Partition Algorithm begins with a fully-constrained partition (i.e. bottom node of the LOA) and recursively "fattens" infeasible blocks of the current best partition until it arrives upon a feasible partition. Due to the recursive-"fattening" nature of the algorithm, the resulting LOA utilized is a more complex version of the Simple LOA introduced in Figure 3.11. Each partition (node) in a Simple LOA applies the same combination of attributes to each of its blocks. In contrast, a Mixed LOA allows a partition to contain blocks composed of different attribute combinations. An example of a Mixed LOA can be found in Figure 3.12. In this example, the algorithm begins with the bottom node at level 0 and attempts to fatten blocks {C} and {D} by relaxing one of the attributes (see dashed lines, solid lines represent the Simple LOA connections). The thinnest partition is discovered at the next level. The pseudocode for the Greedy Bottom-Up Partition Algorithm can be found in Program 3.4 and related Subroutine Program 3.5.

Step 1a of the Bottom-Up Partition Algorithm (Program 3.4) finds all groups of Data Objects that are exact matches (given the attributes consid-

Program 3.4: Pseudocode of Partition-Based Greedy Bottom-Up Generalization Algorithm

```
input: Typed Sibling Group TSG, Raw Coverage Threshold RCT
output: Best Partition BP of TSG that satisfies RCT

BP = TSG; GreedyChoicePartition GCP = {}

1a. FindExactMatches( BP )
1b. Let ITSG = Data Objects of infeasible blocks in BP
1c. Remove ITSG from BP
1d. BP += BUGA( ITSG, RCT )
2.  return BP
```

Program 3.5: Pseudocode of BUGA Subroutine

```
input: Typed Sibling Group TSG, Raw Coverage Threshold RCT
output: Best Partition BP of TSG that satisfies RCT

BP = TSG; GreedyChoicePartition GCP = {}

1. Let attributesToRelax = TSG[0].attributesConstrained
2. If ((attributesToRelax.size() == 0) || (Cost(BP) == 0))
     { return BP }
3a. Let Gens = all next level up generalizations of ITSG //relax attribute
3b. GCP = Gens[0]
3c. For (int i=0; i<Gens.size(); i++) {
3c.i  If (Number of Data Objects in Infeasible Blocks of Gen[i] >= RCT) {
3c.i.A. If (Cost(Gens[i]) < Cost(BP))
          { BP = GCP = Gens[i] }
3c.i.B. Else If (Cost(Gens[i]) < Cost(GCP))
          { GCP = Gens[i] }
      }
    }
4.  Let ITSG = Data Objects of infeasible blocks in BP
4a. If (ITSG.size() > 0) {
        Let greedyChoiceAttributesCombo = GCP[0].attributesConstrained
4a.i.   For (int j=0; j<ITSG.size(); j++)
          { ITSG[j].attributesConstrained = greedyChoiceAttributesCombo }
        Remove ITSG from BP
4a.ii.  BP += BUGA(ITSG, RCT)
    }
5.  return BP
}
```

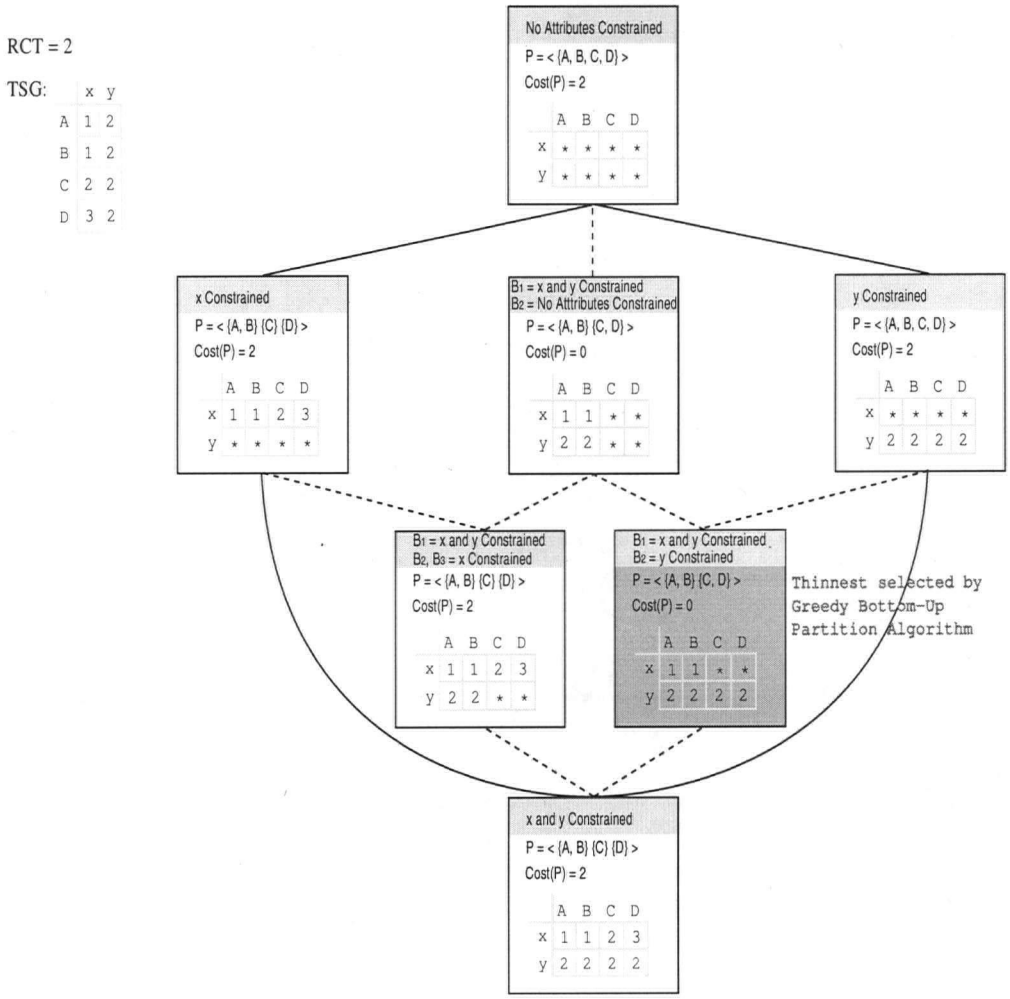


Figure 3.12: Example of a Mixed Lattice of Attributes. This Mixed Lattice of Attributes is used by the Greedy Bottom-Up Partition Algorithm. Note that the algorithm only needs to materialize part of the LOA.

ered) and retains the Multi-Data Object Representatives of those that are feasible. The Data Objects of the infeasible blocks are passed on to the BUGA (Bottom-Up Generalization Algorithm) subroutine. Step 1 and 2 of Subroutine BUGA (Program 3.5) determine which attributes of the TSG are still available for relaxation (since the algorithm moves bottom-up on the Mixed LOA, progressively relaxing attribute combination constraints). If there are no more attributes to relax, only the TYPE attribute can be salvaged and the ANY partition is returned. If the cost of the TSG is already minimal, the best partition is exactly TSG. The If statements in steps 3c.i and 3c.i.A check to see if the cost of the current generalization is better than the current Best Partition BP and also that the remaining number of Data Objects in infeasible blocks is able to satisfy the RCT (i.e. no amount of generalization can satisfy the RCT if the $|TSG| < RCT$). Step 4a checks if infeasible blocks exist in the current Best Partition BP. If there are infeasible blocks that require "fattening", each Data Object of these infeasible blocks is first updated with the cheapest combination of attributes greedily selected by the algorithm (step 4a.i, using the greedily-selected attribute combination determined from steps 3b and 3c.i.B). The algorithm is then called recursively on the TSG composed of the infeasible blocks (step 4a.ii).

3.2.2 Greedy Top-Down

The Greedy Top-Down Partition Algorithm is similar to its Bottom-Up counterpart but traverses down the Mixed LOA. Beginning at the top ANY partition, the algorithm increasingly constrains attributes until a partition with cost = 0 is found or no feasible cheaper partition is found at the next level

examined. Program 3.6 and Subroutine Program 3.7 show the pseudocode.

Program 3.6: Pseudocode of Partition-Based Greedy Top-Down Generalization Algorithm

```
input: Typed Sibling Group TSG, Raw Coverage Threshold RCT
output: Best Partition BP of TSG that satisfies RCT

BP = TSG; GreedyChoicePartition GCP = {}

1a. Let ANYPartition = new PartitionRep
1b. ANYPartition.attributesConstrained = {}
1c. ANYPartition.attributesRelaxed = {TSG[0].attributesConstrained}
2.  return BP = TDGA( BP, ANYPartition, RCT )
```

Step `1a, b, c` of Program 3.6 creates the `ANYPartition` which does not have any attributes constrained (i.e. the top of the Mixed LOA). The entire `TSG` is then passed to the `TDGA` Subroutine for "thinning". In the `TDGA` subroutine (Program 3.7), the `If` statement of step 2 checks the possibility of further splitting the `TSG` argument. If split, the `TSG` must separate into at least two feasible blocks. There also must be at least one remaining attribute that has yet to be relaxed, otherwise, the `TSG` is already as thin as possible. A key difference between the Top-Down and Bottom-Up Partition Algorithms is that the current Best Partition of the Top-Down algorithm is always feasible. Hence, a partition P with a lower cost than the current Best Partition is only selected if it does not contain any infeasible blocks (step `3c.i`). The benefit

Program 3.7: Pseudocode of TDGA Subroutine

```
input: Typed Sibling Group TSG, PartitionRep P, Raw Coverage Threshold RCT
output: Best Partition BP of TSG that satisfies RCT

BP = TSG; flag = false

1.  Let attributesToConstrain = P.attributesRelaxed
2.  If (( Cost(TSG) == 0 ) || ( TSG.size() < 2*actualCT ) ||
      ( attributesToConstrain.size() <= 0 ))
      { return BP } // Partition cannot be thinner
3a. Let Gens = all next level partitions (one attribute constrained)
3b. For (int i=0; i<Gens.size(); i++) {
3c.i. If (no infeasible blocks in Gens[i]) {
3c.i.A. If (Cost(Gens[i]) <= Cost(BP))
          { BP = Gens[i]; flag = true }
      }
    }
4.  If (flag) {
4a.   For each Fat Block j in BP {
        // a Fat Block contains >= 2*actualCT data objects
4a.i.   Remove j from BP
4a.ii.  PartitionRep PP.attributesConstrained = j[0].attributesConstrained
4a.iii. PP.attributesRelaxed = j[0].attributesRelaxed
4a.iv.  BP += TDGA(j, PP, RCT)
      }
    }
    Else { // no next level feasible partition found, BP still = TSG }
5.  return BP
```

of this characteristic of the Top-Down Algorithm is that a feasible solution can always be returned at any stage of the program's execution, without having to wait for the program to complete at the base cases. Also, unlike Bottom-Up, even if a next level partition has the same cost as the initial `TSG` argument, this next level partition then becomes the current Best Partition since it constrains more attributes, which we understand intuitively to be thinner/more specific and thus preferred (see Subsection 2.3.2). Hence, the cost comparison at step `3c.i.A` is not a strict inequality. Finally, if no feasible next-level lower-cost partition was found in step `3`, the greedy-choice Best Partition is the `TSG` itself (see `Else` statement of the `If (flag)` condition at step `4`). However, if a new Best Partition was found, its Fat Blocks are recursively passed to Subroutine `TDGA` for further thinning. For the example given in Figure 3.12, the Top-Down algorithm would return the partition `<A, B, C, D>` (where only the `Y` attribute is constrained) as the Best Partition.

3.2.3 Increasing Combination Size - ICS

The two Partition-based Generalization Algorithms presented thus far examine part of a Mixed LOA in order to greedily-select a cheap (i.e. thin) partition. We now present an algorithm inspired by the Incognito algorithm [17] for k-anonymity (Section 4.1) that uses the unmixed Simple LOA, as illustrated in Figure 3.11, to find a cheap partition. The ICS (Increasing Combination Size) algorithm makes use of the observation that the n-size attribute combination of a feasible partition is composed of k-size attribute combinations, where $1 \leq k \leq n$, that also yield feasible partitions. The observation stems from the fact that relaxing an attribute can only merge blocks

together but never split them. Thus, if we are given a feasible partition, relaxing any attributes of that partition only creates another feasible partition that is of equal or greater cost (due to the possibility of blocks being merged).

The ICS algorithm starts at the top of the Simple LOA and finds all 1-attribute-constrained combinations of feasible partitions. Each of these 1-attribute-constrained combinations are then used as candidates for finding feasible 2-attributes-constrained combinations. The algorithm iteratively uses *(k)*-size feasible combinations to find the *(k + 1)*-size feasible combinations. When the largest-sized feasible combination is found, the algorithm returns the partition allowed by the combination. The pseudocode for the ICS algorithm is shown in Program 3.8.

Step 1 of Program 3.8 determines which attributes are to be considered. Step 2 effectively materializes portions of the LOA downward, finding increasingly larger feasible attribute combinations. If a particular attribute is part of an infeasible combination and not part of any feasible combination (Step 2d), it is removed from the `AttributesList` before the next iteration uses the `AttributesList` to find a feasible combination that is 1 attribute larger. This follows the logic that all subsets of a feasible combination must also be feasible.

Unlike the Greedy Bottom-Up and Top-Down Partition algorithms, the ICS algorithm does not need to make any greedy decisions. Moreover, because all the blocks of the ICS-returned Best Partition are constrained by the same n-sized attribute combination rather than different attribute combinations (as in the greedy algorithms), all k-sized ($1 \leq k \leq n$) feasible attribute combinations are implied by the solution. Any subset of the attribute combi-

Program 3.8: Pseudocode of Partition-Based ICS Generalization Algorithm

```
input: Typed Sibling Group TSG, Raw Coverage Threshold RCT
output: Best Partition BP of TSG that satisfies RCT

1.   Let AttributesList = {TSG[0].attributesConstrained}
2a.  For(int i=1; i <= AttributesList.size(); i++) {
2b.    Let Gens = All size-i attribute combinations (i.e. generalizations)
                  from AttributesList
2c.    For(int j=0; j < Gens.size(); j++) {
2d.      If (Gens[j] is feasible) {
           Mark attributes of Gen[j] as DoNotRemove in AttributesList
         } Else {
           // Gens[j] is not feasible
           Mark attributes of Gen[j] as possiblyRemove in AttributesList
         }
       }
2e.    For each attribute m in AttributesList {
         If ((m marked possiblyRemove) && (m not marked DoNotRemove)) {
           Remove m from AttributesList
         }
       }
     }
3.   return BP = TSG grouped using AttributesList
```

nation solution also yields a feasible unmixed partition. The ICS algorithm is exponential in the size of the largest attribute combination considered at the bottom of the LOA, but it does return the global optimal *unmixed* attribute combination solution. However, the inflexibility of not allowing mixed attribute combinations can lead to a partition that is less-preferred than its mixed counterpart. Consider the example in Figure 3.11 where the ICS algorithm returns the partition composed of attribute `y Constrained`, which is not as thin as the Bottom-Up Greedy *mixed* solution (see Figure 3.12). In general, mixed attribute combination partitions can yield more specific solutions than unmixed partitions.

A common weakness that the ICS algorithm shares with the Bottom-Up and Top-Down Partition algorithms is that attribute combinations (i.e. generalizations) are not considered simultaneously for feasible disjoint blocks that when combined together, create a feasible partition, even if the blocks originated from infeasible partitions. Consider the example in Figure 3.13. The ideal solution consists of block $B_1 = \{\text{A, B}\}$ with `x Constrained` and block $B_2 = \{\text{C, D}\}$ with `y Constrained`. The ICS algorithm is not able to arrive at this optimal partition since it is a mixed partition. The Top-Down algorithm is also unable to reach the optimal partition because both partitions under the `ANY` partition are infeasible. However, the Bottom-Up algorithm is able to reach the partition but due to its greedy nature, does not retrieve the most-specific attribute combination for one of the blocks (see Figure 3.14).

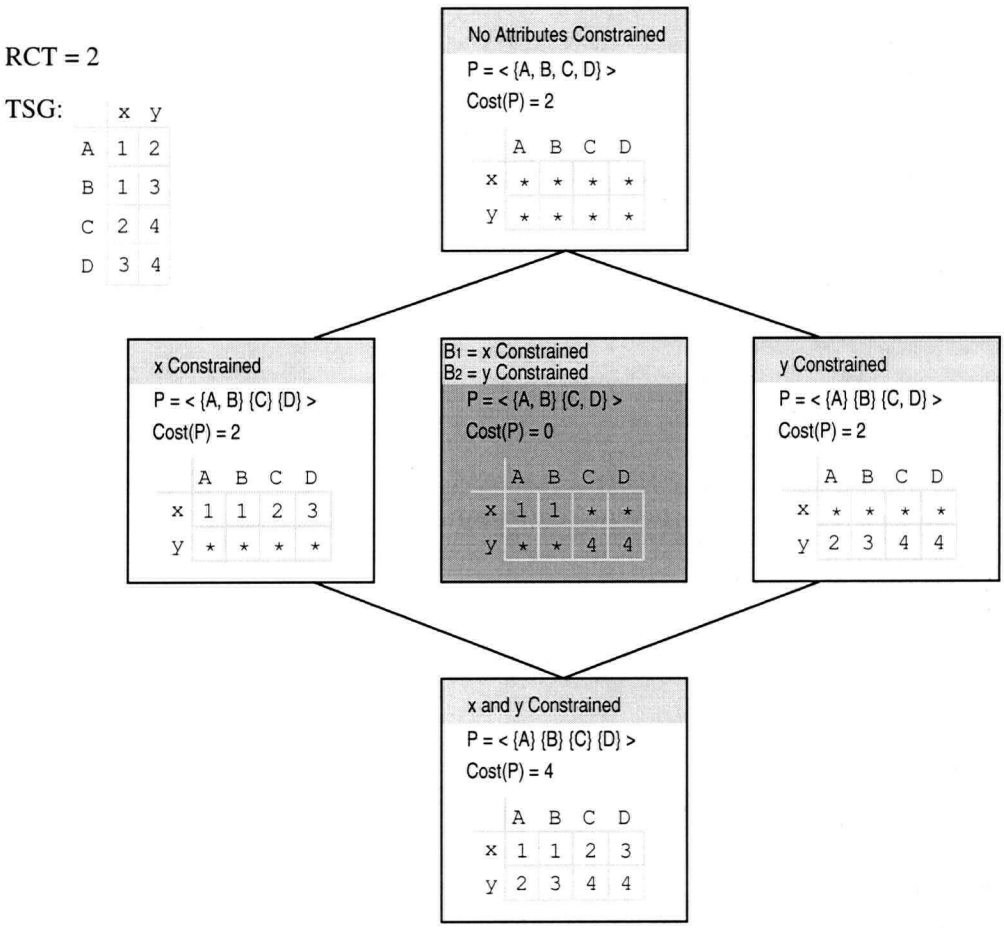


Figure 3.13: Common Weakness of Partition Algorithms. The Partition Algorithms do not consider piecing disjoint blocks from infeasible partitions in order to create a feasible partition.

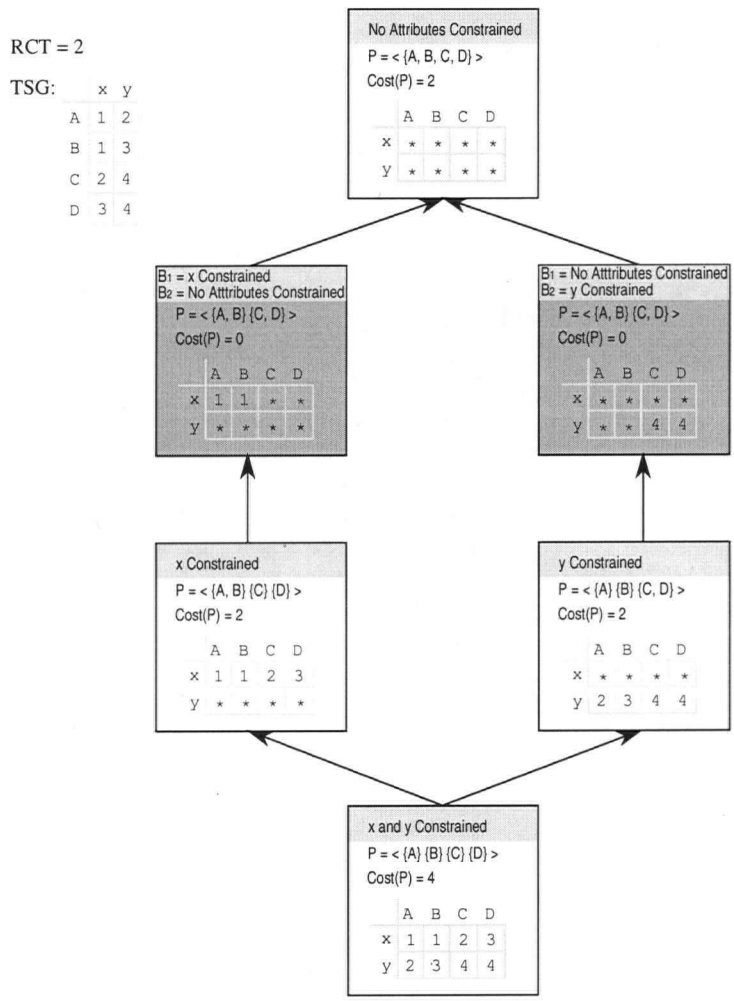


Figure 3.14: Bottom-Up Solution to Common Weakness of Partition Algorithms. The Greedy Bottom-Up Partition Algorithm is able to find the feasible partition of cost=0 but cannot retrieve the most specific attribute combination characteristics (See Figure 3.13).

3.3 Chapter Summary

In this chapter, we presented the concepts behind summarizing Navigation Trees. Using a "dummy" root, all Navigation Trees are brought together to create an Intermediate Summary. The Intermediate Summary is then processed, level-by-level beginning from the top, and Typed Sibling Group by Typed Sibling Group (TSG). Each TSG is passed to a Generalization Algorithm which returns Multi-Data Objects that represent the Data Objects of the TSG. We showed how the quality of the Summary is affected by the Generalization Algorithm and the Coverage Threshold. The five different Generalization Algorithms presented can be categorized as either Node-based (Greedy and Enumerate All) or Partition-based (Greedy Bottom-Up, Greedy Top-Down, and Increasing Candidate Size). When summarizing a TSG, Partition-based algorithms account for multiple Data Objects at a time while Node-based algorithms consider only one Data Object at a time.

Chapter 4

Related Work

In this Chapter, we will examine work that share concepts and techniques that bear interesting similarities to our work of summarizing Navigation Trees.

4.1 *k*-anonymity

K-anonymity was described in [28] as a standard in which sensitive data (such as government, medical, and bank records) can be released without violating the privacy of individuals. Samarati and Sweeney [22] showed that individuals can be identified using "linking attacks" that combine datasets which, by themselves, do not uniquely identify individuals. An example of a "linking attack" is given in [28] where voter registration data and hospital patient data uniquely relate a patient to his/her disease when joined together on the shared attributes *ZIP, Birth Date* and *Sex*. Sweeney explains that using only the *ZIP, Birth Date* and *Sex* attributes, the unique identities of 87% of the population of the United States can be determined [28]. The *k*-anonymity property is the requirement that each record in a dataset belongs to a group of at least k - 1 other records within which they are indistinguishable from each other with regards to the *quasi-identifier attribute set*. The *quasi-identifier attribute set* (also known as the *virtual identifier* [11, 31]) is

a subset of the total attributes of a record that is known to expose individual identities if linked with other datasets. For example, $\langle ZIP,\ Birth\ Date,\ Sex \rangle$ would be the quasi-identifier attribute set in the example involving voter registration and hospital patient data. Privacy can be retained by making sure that released data adheres to the k-anonymity property. Sweeney explained how k-anonymity can be achieved without violating data integrity by using attribute-value generalization and suppression [27, 28]. Suppression is simply the "removal" of the value altogether and can be understood as generalization to the ANY-value (indicated by "*") at the top of the attribute-value hierarchy (see Figure 2.6 for example). In our framework, we treat suppression as simply the ultimate form of generalization (Subsection 2.3.1).

The approach we take in summarizing Navigation Trees is analogous to creating a k-anonymized version of a table T. Recall that our Summary Tree is created by merging Navigation Trees (Section 3). The merging is facilitated by generalizing Data Objects in the Navigation Trees so that each Multi-Data Object in the Summary Tree *covers* at least CT (Coverage Threshold, Subsection 2.3.2) number of Data Objects. Essentially, we can view each Data Object as a tuple of a table T that we wish to partition into groups of at least CT tuples each and where each tuple in a group is indistinguishable from the others in that group. The Coverage Threshold is analogous to the k value in k-anonymity - as the value of CT (or k) increases, the larger the size of each group (i.e. partition) and hence, the less specific the Multi-Data Object that represents the group.

Many different definitions of *optimal* k-anonymization have been proposed. The intuition behind finding the optimal solution is to find the

anonymization that minimally *distorts* the data. Distortion is usually quantified with a cost metric that captures how much information is lost from generalization or suppression [3, 14, 15, 21, 27]. The hierarchy level to which a dataset must be generalized to is an example of such a cost metric and is used in some variation by Hundepool and Willenborg [14], Samarati [21], and Sweeney [27]. Fung et al. [11], Iyengar [15], and Wang et al. [31] tailored their cost metrics to solve the k-anonymity problem with a focus on using the k-anonymized data for classification purposes. The optimal K-anonymity problem was shown to be NP-hard in [1, 18]. Optimal algorithms that guarantee minimality of the resulting k-anonymization, such as Incognito [17] and K-OPTIMIZE [3] are exponential in the quasi-indentifier size. However, it is known that greedy heuristic algorithms such as "Datafly" [27] and Samarati's Binary Search [21] work well in practice despite having no optimality guarantee [3].

4.1.1 Global and Local Recoding

One of the major differences among proposed k-anonymity techniques is in the *recoding* model used. *Recoding* refers to the generalizations that attribute values can undergo. The two types of recodings possible are *Global Recoding* and *Local Recoding* [17]. Techniques that map all attribute values to generalized values on the same level of the generalization hierarchy are referred to as *global recoding* techniques. Techniques that allow individual instances to map to a generalized value on a different hierarchy level than other instances are known as *local recoding* techniques. To illustrate the concept more clearly, consider a table T (see Table 4.1) that we wish to k-anonymize

with $k = 2$. Let the quasi-identifier = $\langle$ *Date Modified* $\rangle$ and assume that the date hierarchy as shown in Figure 2.6 is used.

id	File Name	Date Modified
t1	CS404_October13.pdf	13/10/06
t2	CS404_October15_.pdf	15/10/06
t3	CS404_November25_corrected.pdf	25/11/06
t4	CS404_December05.pdf	05/12/06

Table 4.1: Table T to k-anonymize with $k = 2$

By generalizing the quasi-identifier to `**/mm/yy`, t1 and t2 satisfy the *2*-anonymity requirement but t3 and t4 generalize to **/11/06 and **/12/06 respectively. t3 and t4 have to be generalized to `**/**/yy` in order to be indistinguishable from at least 1 other record. Since `**/mm/yy` and `**/**/yy` are on different levels of the date hierarchy, Global Recoding would map all attribute values of T to their respective `**/**/yy` value (the "higher" generalization), as in Table 4.2. Local Recoding allows for attribute values to be on different generalization levels and hence permit the *2*-anonymized version of T as shown in Table 4.3.

Note that Local Recoding methods are more "flexible" than their Global Recoding counterparts and are able to provide k-anonymized views with less distortion. In the Globally-Recoded Table 4.2, all the tuples are merged into one set $\langle t1, t2, t3, t4 \rangle$ since they are non-unique with regards to the *Date Modified* quasi-identifier. The Locally-Recoded Table 4.3 effectively partitions T into two groups, $\langle t1, t2 \rangle$ and $\langle t3, t4 \rangle$ and retains more of the original

id	File Name	Date Modified
t1	CS404_October13.pdf	**/**/06
t2	CS404_October15_.pdf	**/**/06
t3	CS404_November25_corrected.pdf	**/**/06
t4	CS404_December05.pdf	**/**/06

Table 4.2: *2*-anonymized with Global Recoding

id	File Name	Date Modified
t1	CS404_October13.pdf	**/10/06
t2	CS404_October15_.pdf	**/10/06
t3	CS404_November25_corrected.pdf	**/**/06
t4	CS404_December05.pdf	**/**/06

Table 4.3: *2*-anonymized with Local Recoding

information in T (the fact that $t1$ and $t2$ were modified in October of 2006 is captured). The k-anonymity algorithms of Lefevre [17], Iyengar [15], Bayardo and Agrawal [3], Fung et al. [11], and Wang et al. [31] are examples of Global Recoding techniques. Local Recoding is used in the algorithms of Aggarwal et al. [1], Meyerson and Williams [18], and Sweeney [27]. Except for the ICS partition algorithm (Subsection 3.2.3), the generalization algorithms in this thesis can be classified as Local Recoding models (See Chapter 3).

4.1.2 Differences with Current k-anonymity Techniques

The following lists the areas in which the known k-anonymity techniques differ from our approach in summarizing Navigation Trees:

- *Quasi-identifier Size* - The size of the quasi-identifier greatly affects the runtime of k-anonymity algorithms [18]. The larger the quasi-identifier size, the more combinations of generalizations the algorithms have to examine, thus yielding a longer runtime. Thus far, algorithms only examined a small quasi-identifier size; As a reference, Lefevre examines up to a 9-attribute quasi-identifier for a 5.5MB table containing 45,222 records [17], as does Iyengar [15], and Bayardo and Agrawal [3] for a data set with 30,162 records. Fung et al [11] tests a 14-attribute quasi-identifier for up to only 1,000 records - this is the largest test quasi-identifier size known. In comparison, we test a large quasi-identifier size of 822 for our clickstream data (Section 5.1).

- *Number of Times Generalization Algorithm Invoked* - The k-anonymity algorithms are invoked only once per dataset. However, in the case of summarizing Navigation Trees, our generalization algorithm is run multiple times. In particular, our generalization algorithm has to be run for each *Typed Sibling Group* on each level of the intermediate Summary Tree (Chapter 3).

- *Outliers* - A number of k-anonymity approaches allow for tuples to be completely excluded from the k-anonymized view of the original dataset

[3, 21, 22]. These "excluded" tuples are considered to be outliers that would otherwise require significant generalization (i.e. distortion) of the dataset in order to satisfy the k-anonymity property. In summarizing Navigation Trees, recall that the Navigation Trees have been created with the aid of the user (Section 2.2). Thus, no subset of Data Objects in the Navigation Trees are treated as outliers since the user has deemed all of them important to consider.

- *Coverage Threshold as a Percentage* - We have chosen the Coverage Threshold (CT) to be a percentage (between 0% and 100%) rather than a natural number as defined by the k in k-anonymity. The benefit of using a percentage is that the CT is always *satisfiable* - it is not possible to encounter a dataset where $k >$ # Data Objects. Since our generalization algorithm is performed at least once per Typed Sibling Group (see Chapter 3), the Raw Coverage Threshold (Subsection 2.3.2) has to change according to the size of the group in order to ensure satisfiability.

- *Attribute Generalization Hierarchy Height* - Our work considers 2-level generalization hierarchies with the top level generalization being the "*" or *any* value (also equivalent to suppressing the attribute). We note that it is possible to increase the heights of the attribute generalization hierarchies by imposing domain-specific ontologies. Although our work deals with the 2-level hierarchies, it can be extended to multi-level hierarchies (the Lattice of generalizations would grow to include the multi-level possibilities, Chapter 3).

- *Data* - The data that we deal with in this research are files of all types as opposed to tuples of a relational database. However, we can view each file as a tuple by treating each file attribute as a field column of a relational database.

4.2 Workflow Mining

Workflow mining (also known as *process mining*) aims to discover workflow patterns from a log of recorded events (e.g. transaction logs) [2]. A workflow is an ordered set of events that is performed to accomplish a task. An *instance*, or *case*, of a workflow is an individual execution of the workflow. Each line in the transaction log corresponds to a particular step (e.g. "process book checkout request", "install back seats of vehicle") belonging to a specific instance of the workflow. Given a workflow transaction log, the goal of workflow mining is to extract the individual instances of the workflow and create a workflow *model* that best describes all of those extracted cases. As described in [30] and [25], workflows describe *sequential events* (e.g. "Event B follows Event A"), *alternate events* (e.g. "A or B"), *parallel events* (e.g. "A and B simultaneously"), and *repeating events* (i.e. cycles). In the process of workflow mining, outlier events can also be completely removed from the workflow model.

In contrast, the Navigation Trees in our work capture the type of data the user dealt with while performing a task - individual nodes in our model do not refer to actions or "steps" but to (Multi-)Data Objects (see Section 2.1). Also, workflow mining does not consider generalizations of workflow instances

when creating the model. Workflow instances either contain steps that are included exactly (i.e. without generalizing) in the workflow model or not included at all (i.e. labeled as an outlier). Techniques to detect workflow outliers include heuristics [32] and stochastic models [13]. Finally, while our Navigation and Summary Trees do not allow for cycles, duplicate Data Objects are allowed whereas the majority of workflow research assume that each workflow event is unique (see [29], [2], and [23]). Our work can be viewed as a combination of workflow mining with k-anonymity generalization - our Summary Trees try to describe all the Navigation Trees by generalizing their Data Objects. The resulting Summary Tree is similar to the workflow model in that it captures the order of events (re: Data Objects) performed by the user when accomplishing a particular task, but is also able to provide a description of the data encountered as well.

4.3 Personalization

The goal of Personalization is for the system to be able to automatically predict the user's intentions. Schlimmer describes Personalization systems as "self-customizing software" [24]. Work in this area has looked into predicting web usage [8, 12, 19], UNIX commands [9, 16], responsive actions to emails [6], news-content interest [4, 5], and even calculator keystrokes [7]. Probabilistic models and artificial intelligence techniques have been applied to the goal of predicting the user's "next step". The majority of work performed in this field can be broadly categorized as *event-based* and *content-based* prediction models. Event-based prediction models guess the type of

command/action that the user will next invoke (e.g. issuing a UNIX `ls` command after using `cp` to copy files into the current directory). On the other hand, *content-based* prediction models examine the content of the data in order to decide what other information might be of interest to the user (e.g. analyzing the contents of a particular news article in order to present other relevant news articles to the user). To our knowledge, the current personalization techniques do not construct complete task process models - prediction is based simply on the current step (e.g. the current UNIX command or news article being viewed) rather than the macroscopic view of where the current step lies in the process of task completion. Our work is unique in that we capture both the user action sequence *and* the type of data (i.e. content) encountered so as to be able to better predict user intention when performing recurring and similar tasks.

Chapter 5

Experiments

The algorithms in this thesis have been implemented and tested for the purpose of comparing their respective runtimes and scalability, and to evaluate their advantages and disadvantages. The space and time complexities of the algorithms are directly related to the portion of the LOC (or LOA) that needs to be materialized in order to find a satisfactory generalization. The Greedy Node-based and Partition-based algorithms have a polynomial complexity due to only having to materialize certain branches of the Lattice, while the Enumerate All and ICS algorithms are exponential since in the worst case, they materialize the entire Lattice. However, due to each algorithm's individual characteristics, the quality of the Summaries returned differs as well. Thus, the goal of our experiments is not only to act as a "proof of concept" but also to attempt to evaluate the quality of our Summaries. Specifically, we would like to examine the relationship between Coverage Threshold, number of attributes considered, and the Summary quality. The dataset we will execute our algorithms on is a combination of those found in workflow and k-anonymity. The Navigation Trees need to capture both the sequence (as in workflow) *and* the details (similar to k-anonymity databases) of the Data Objects. In general, workflow-like data is hard to attain as special application-specific software is required [25] for this purpose. However,

we have found clickstream data to meet the dataset requirements for our experiments.

5.1 Clickstream Data

We have gathered clickstream data from the `http://www.cs.ubc.ca` domain for the period of January 1, 2005 to May 31, 2006. This domain belongs to the Department of Computer Science at UBC and the data represents all client web requests to web objects located on that domain. The clickstream data of the period collected was stored in daily access log files and totaled 8.9GB compressed. The 541 access log files contain all the client requests for web objects. Each line of an access log provides the following information:

- Client IP
- Date and Time
- Requested Object
- Client's User Agent

In order to protect user privacy, the actual Client IP was mapped to a unique non-negative integer that remained consistent throughout the access logs. A total of 5,125,472 unique Client IPs accessed the domain during the period in question.

For our experiments, the Data Objects of concern are limited to web pages with textual content that are able to be rendered on standard Web browsers. Non-textual web objects such as picture, music, and video files

are not considered. A list of keywords was attained by combining the dictionary entries from `http://www.webopedia.com/Computer_Science/` with the names of UBC Computer Science Faculty and Course Codes (e.g. CPSC 100). Any duplicate entries were removed, resulting in a list of 822 keywords. These keywords act as the attributes of the web page Data Object. If a web page Data Object contains a keyword, the value of that particular keyword attribute is 1, and 0 otherwise. Thus, each web page has a keyword bit index associated with it that represents the keywords it contains. Shared keywords among a group of web pages can be determined by executing a bitwise `AND` ($\bigwedge$) between the respective keyword bit indexes (see Figure 5.1).

We will use Navigation Trees to represent user clickstreams. The clickstream captures the sequence of webpages that the user visited. For example, a small clickstream could show that the user went from the UBC Computer Science Research page to the UBC DB Lab home page, and eventually ended up at the DB Lab Issues webpage. In order to construct Navigation Trees from the clickstream data, it was necessary to first determine the keyword indexes for all web pages on the `cs.ubc.ca` domain. To do this, we first crawled the entire domain using SiteMapBuilder.NET [26] and filtered out unwanted web objects. A total of 12,960 relevant web pages were found. Keyword indexes were then determined for each of the web pages. At this point, we have all the Data Objects, including their full attribute keyword description, necessary for Navigation Tree construction. The next challenge is determining the sequence of Data Objects in the Navigation Trees.

Each Navigation Tree is user and session-specific. Hence, it is important to determine individual clients within the access log data. We note that the

URL1 contains: computer, database, machine, xml

URL2 contains: artificial intelligence, database, recognition, vision, xml

	artificial intelligence	computer	database	machine	recognition	vision	xml
URL1	0	1	1	1	0	0	1
URL2	1	0	1	0	1	1	1
URL1 ∧ URL2	0	0	1	0	0	0	1

Figure 5.1: Finding Shared Keywords Between Web Page Data Objects. With keywords represented by a bitmap, shared keywords can be determined with a bitwise AND ($\bigwedge$) on the keyword bit indexes of the web pages in question.

unique users as multiple users may share the same IP *and* User Agent.

Our hope is that summarizing the Navigation Trees of a user will provide a useful overview of the user's browsing patterns and interest with regards to particular web pages on the `cs.ubc.ca` domain. In our scenario, interest is suggested by the keywords found in the web pages of the computed summary. Hence, the summary may act as a filter to reduce the number of web pages the user might be interested in (based on past displayed interest). On top of suggesting the types of web pages that may be of interest, the summary also captures the sequence in which they were accessed. This summarization may also aid in predicting the user's future web request sequence by suggesting a complete or partial path (to allow "skipping" nodes and directly accessing the goal leaf/end node) of the summary that best matches the user's current clickstream.

5.2 Setup

Experiments were conducted on a system running SuSE 10.1 using an Intel P4 3Ghz processor and 2GB PC3200 RAM. The algorithms presented in Chapter 3 were implemented using Java. Experiments were performed on varying data sizes (1MB, 2MB, 5MB, 10MB, 25MB, and 50MB) and with varying coverage thresholds (10%, 20%, 30%, 40%, and 50%). In addition to testing the set of all 822 keywords, the number of attributes considered (known as the "quasi-identifier size" [17]) was also varied to: 4, 8, 12, 16, 20. We ran 5 trials for each scenario, removed the max and min runtimes, and report the average runtime of the remaining 3 trials.

In our clickstream data, there are over 5 million different Client IPs, which combined with User Agent, yield an even greater number of unique users. No single unique user's clickstream data over the collection period totaling more than 1MB was found. Hence for the purpose of creating our experiment datasets, the navigation trees of multiple unique users were grouped together. Table 5.1 lists the number of Navigation Trees each of the Datasets are composed of.

Dataset Size	# of Navigation Trees
1MB	814
2MB	1709
5MB	6692
10MB	16885
25MB	54688
50MB	104225

Table 5.1: Number of Navigation Trees Summarized Per Dataset Size

Recall that each Data Object is associated with 822 keyword attributes (Section 5.1). Due to the hardware limitations of our test system, only the Node-based Greedy and Partition-based Top-Down and Bottom-Up algorithms are able to process all 822 keywords. Both the Node-based Enumerate All and the ICS algorithm are unable to consider all of the possible keyword combinations (as is required in their worst-case scenario). Hence, we also compare the algorithms in an attribute-limited fashion; this is akin to varying the quasi-identifier (QI) size in [17]. The attribute-limited versions of our algorithms make an initial pass across the Typed Sibling Group (TSG)

of Data Objects to be generalized (see Chapter 3) and retain all the keywords that are shared by all Data Objects in the TSG. These retained keywords will be included in the Multi-Data Object(s) that the generalization algorithm returns. Then, the algorithm selects the top x most-occurring attributes (where $x = |\text{QI}| = 4, 8, 12, 16, 20$) from the remaining keywords that are not shared by all of the TSG members. The top x most-occurring keywords will then be the only attributes considered by the generalization algorithm for finding suitable Multi-Data Objects that satisfy the Coverage Threshold.

In order to compare the quality of the summaries computed by the various algorithms, we also report the *Average Slack* and *Average AvgDiff*. The definition of the *Average Slack* is as follows:

$$\textit{Average Slack} = \frac{\textit{Thinness}}{\#\textit{Multi} - \textit{DataObjects} \in \textit{Summary}}$$

with *Thinness* as defined in Subsection 2.3.2.

The Slack gives us an idea of how well each algorithm was able to meet the Coverage Threshold, without surpassing it significantly. A lower Slack is preferred as this indicates a thinner summary. The *AvgDiff* measure reflects the number of keywords (i.e. level of detail) a Multi-Data Object was able to retain after summarizing the Typed Sibling Group from which it was derived. More formally, for a Multi-Data Object X in a Summary and the Typed Sibling Group TSG it summarizes, the AvgDiff of X is defined as:

$$AvgDiff_X = \sum_{y \in TSG} [\mid keywordsON_y \mid - \mid keywordsON_X \mid]$$

where $keywordsON_i$ is the set of keywords found in (Multi-)Data Object i (i.e. the bit index location for the keyword is set to ON,

or 1). Note also that $| keywordsON_y | \geq | keywordsON_X |$ since X is a generalization of y.

Then, for all the Multi-Data Objects in the summary, we define

$$Average\ AvgDiff = \frac{\sum_{X \in Summary} AvgDiff_X}{\#Multi - DataObjects \in Summary}$$

The lower the AvgDiff, the more similar the Multi-Data Object is with the Data Objects it generalizes. Lower AvgDiff values also suggest that the summary does not "over-capture" too wide a target which is undesirable using the same argument for *thinness* (Subsection 2.3.2).

5.3 Results

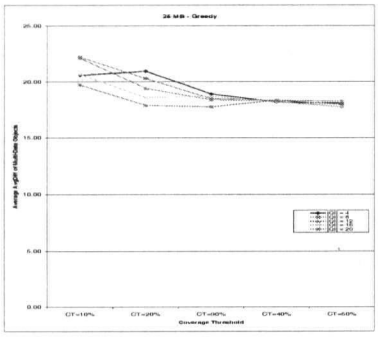
Due to the size of our complete experimental results, only select tables and graphs will be displayed in this section. Please refer to Appendix A for more complete detailed results.

Our experimental results support the following observations:

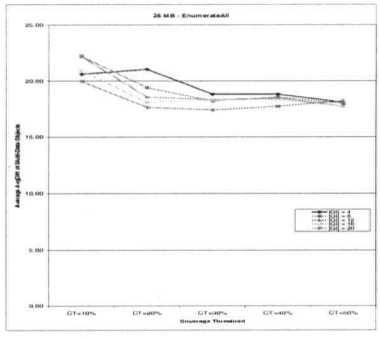
- For our dataset, there was no significant improvement in summary quality when considering more than 20 attributes. The most significant improvement can be found when increasing the attributes considered (i.e. |QI|) from 4 to 8. Quality appears to converge as |QI| and Coverage Threshold (CT) increase. The graphs in Figure 5.2 and Figure 5.3 show that as CT (x-axis) increases, the |QI| has less of an effect on the Average AvgDiff/Slack (y-axis in the corresponding graphs); this is indicated by the converging lines which represent different |QI|. All

the algorithms except for ICS suffer an upwards spike in Average Slack when CT = 50%; this is likely due to the Navigation Trees not being able to be summarized exactly into half, thus resulting in numerous occasions where a Typed Sibling Group (TSG) is represented by a single Multi-Data Object. Note that ICS does not exhibit this pattern as it does not consider Slack cost when summarizing TSGs and hence has very high Average Slack values even at low CTs.

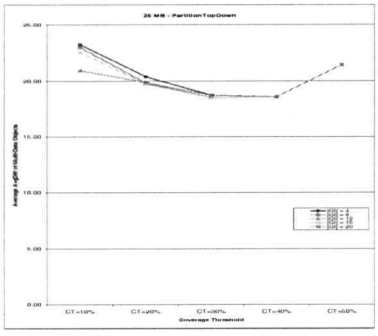
- In general, the results of the Enumerate All algorithm follow very closely with that of the Greedy algorithm. Please see Figures 5.2(a), 5.2(b) and 5.3(a), 5.3(b) for example. The execution times of Enumerate All are always at least as high as that of Greedy and very often they are arguably equal (refer to Tables A.2-A.6 and A.8-A.12 in Appendix A). Instances where Enumerate All take considerably longer than Greedy can be due to the algorithm having to climb much higher up the Lattice of Conditions (LOC) for the *thinnest* generalization or that by progressively selecting the thinnest, the algorithm has to be invoked more often per TSG on the way to finding more representative Multi-Data Objects. The Average Slack and AvgDiff quality of Enumerate All are often not of vast improvement to warrant the possibility of greater runtime. At CT = 50%, Enumerate All and Greedy have almost identical runtimes and quality. Furthermore, there are a very small number of cases where Greedy slightly edges Enumerate All in quality (i.e. by hundreths or tenths). This phenomenon is likely attributed to Enumerate All building its LOC by selecting a Data Object which is more dissimilar to the remaining TSG members than its



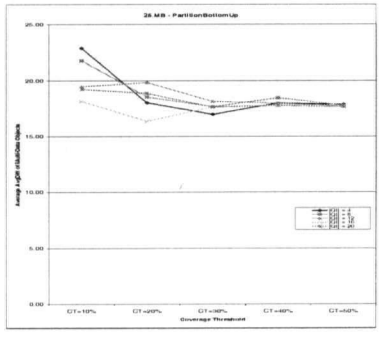
(a) 25MB, Greedy



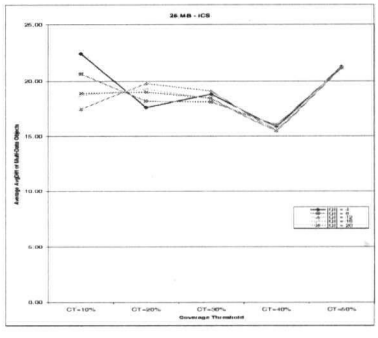
(b) 25MB, Enumerate All



(c) 25MB, Partition Top-Down

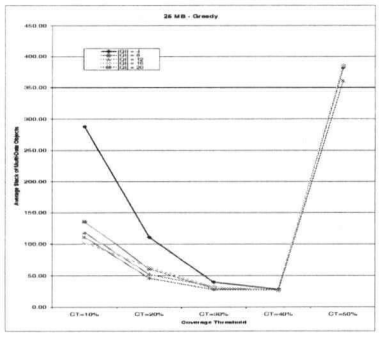


(d) 25MB, Partition Bottom-Up

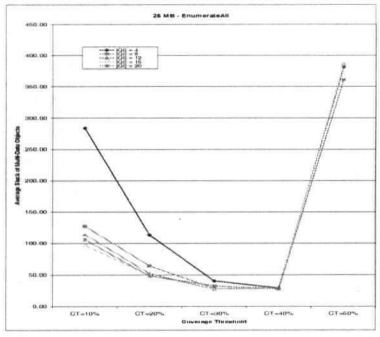


(e) 25MB, ICS

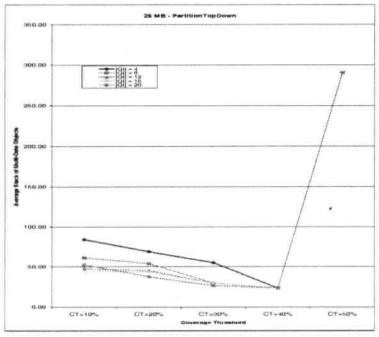
Figure 5.2: Average AvgDiff for 25MB



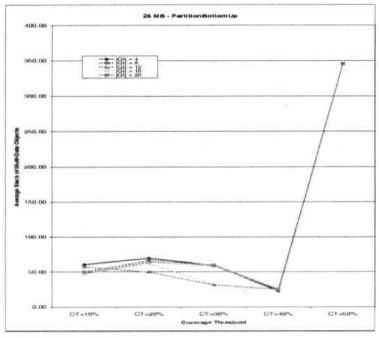
(a) 25MB, Greedy



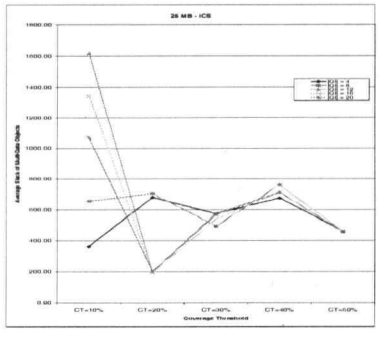
(b) 25MB, Enumerate All



(c) 25MB, Partition Top-Down



(d) 25MB, Partition Bottom-Up



(e) 25MB, ICS

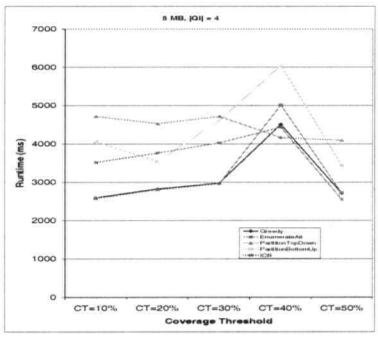
Figure 5.3: Average Slack for 25MB

Greedy counterpart.

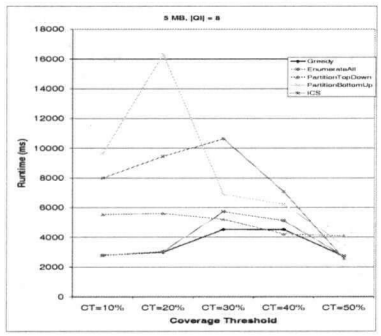
- The attribute-limited Partition-Based Bottom-Up algorithm has very competitive runtimes for the 1MB and 2MB datasets. For the larger datasets, however, the runtimes are significantly higher than the other algorithms when considering |QI| = 12, 16, 20, 822 at the lower Coverage Thresholds (i.e. 10% and 20%). Please see Figure 5.4 for example. While the Bottom-Up algorithm exhibits relatively long execution times for the the 5MB, 10MB, 25MB, and 50MB datasets, the quality of its summaries is still in the top-2. The Partition-Based Bottom-Up algorithm also displays the most markedly-increasing runtimes as |QI| increases. This algorithm considers the whole TSG simultaneously from bottom-up as opposed to the Node-based algorithms which consider generalizing one Data Object at a time, and the Partition-Based Top-Down and ICS algorithms which work from top-down by progressively "thinning" fat blocks. As with all the algorithms, relative results vary with different datasets. For example, the ICS algorithm displays the worst execution times for the 1MB and 2MB datasets.

- In general, the ICS algorithm has the highest Average Slack amongst the evaluated algorithms. However, this is expected as the ICS algorithm does not attempt to minimize Slack in anyway. Instead, it is designed to find all *unmixed* attribute combinations that satisfy the CT, thus effectively returning one Multi-Data Object per TSG.

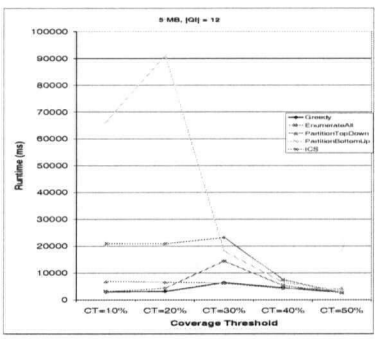
- The ICS and Partition-Based Bottom-Up algorithms generally have the lowest-2 Average AvgDiff values for the 5MB, 10MB and 25MB



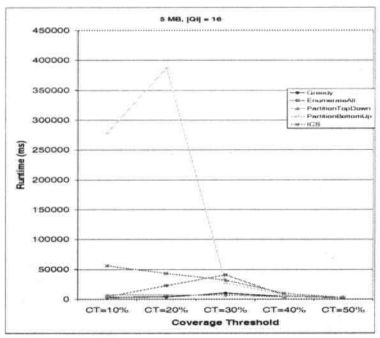
(a) 5MB, |QI| = 4



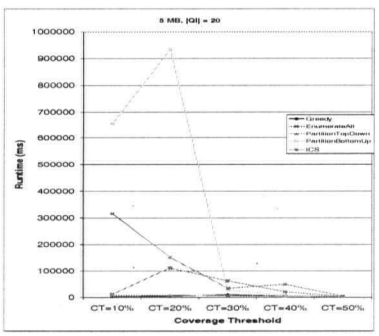
(b) 5MB, |QI| = 8



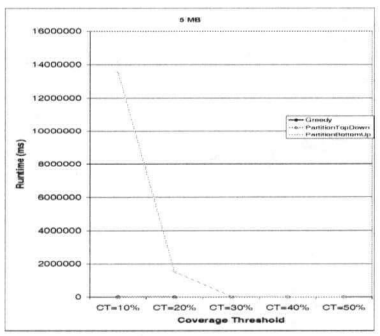
(c) 5MB, |QI| = 12



(d) 5MB, |QI| = 16



(e) 5MB, |QI| = 20



(f) 5MB

Figure 5.4: Runtimes for 5MB

datasets. For the smaller datasets, Bottom-Up tends to obtain the best Average AvgDiff value, with the Node-based algorithms occupying the next 2-lowest positions. Exceptions to this general behaviour can be found at 1MB with |QI| = 4, 8 and CT = 10%, 20%, 30%.

- Generally, for the 10MB and 25MB datasets, the Partition-Based Bottom-Up algorithm achieved the best Average Slack values with CT=10% and |QI|=4 and 8. The respective Average AvgDiff for the same specifications were also very competitive (scoring second-best to ICS in the majority of these occasions).

- Results are very dependent on the dataset characteristics. Some datasets react better to being summarized top-down while others have slightly more similar Data Objects and can be effectively approached from bottom-up. Also, there is not a "one-size-fits-all" CT that *always* yields the best quality for the smallest runtime. Generally speaking, lower CT values yield better results but cause the algorithms to execute longer. In this regard, the mid-way CT value of 30% appears to be a good starting point from which the user can decide if the returned summary is satisfactory or a lower CT is required.

- For almost all our experimental conditions, both the Bottom-Up and Top-Down Partition-based algorithms yield better-quality summaries than their Node-based counterparts, with the discrepancy being more significant for the 10MB and 25MB datasets. Examples of this behaviour can be found in Figures 5.2, 5.3, and Tables A.20-A.24 in Appendix A.

- As mentioned in Section 5.1, the clickstream data used in these experiments do not allow for error-free unambiguous discernment of unique users and their respective Navigation Trees. Also, we have had to merge the Navigation Trees of users known to be different for the purpose of creating our test datasets. Hence, a TSG is much more likely to contain dissimilar, and possibly disjoint, Data Objects which may not be able to be summarized as effectively as when considering similar-task Navigation Trees from a single unique user. Our generally-observed comparatively-longer runtimes of the algorithms that approach the LOC (or LOA) from bottom-up support our suspicion that these algorithms have to climb closer to the top of the lattice in order to find a satisfactory generalization (refer to Table 5.2 for instance).

- For the most part, the Top-Down Partition-based algorithm seems to be a good compromise on quality and runtime (see Figure 5.4 and Tables A.13-A.18 in Appendix A for example).

- For a given dataset size and a higher CT of either 40% or 50%, the number of attributes considered does not greatly affect the execution time (see Table 5.2). However, at lower CT values, the general observed behaviour of the algorithms is that larger sizes of |QI| increase the execution time. The algorithms do not always scale linearly across dataset size, especially with the larger sizes (10MB, 25MB, and 50MB) at lower CT values (10% and 20%). However, this behaviour is to be expected as having a twice-as-large dataset does not imply having to do only twice as many generalizations since the number of TSGs may

increase by more than double. Representative runtimes are displayed in Table 5.3.

Node-Based GREEDY

	10%	20%	30%	40%	50%
|QI| = 4	14160	17698	17738	37121	13409
|QI| = 8	17607	19824	72995	37226	13336
|QI| = 12	23829	20418	73575	37525	13395
|QI| = 16	24237	20907	73749	37460	13440
|QI| = 20	24496	39809	73893	37698	13484

Node-Based ENUMERATE ALL

	10%	20%	30%	40%	50%
|QI| = 4	14191	17594	17748	36960	13501
|QI| = 8	17686	21023	116100	37497	13527
|QI| = 12	26437	24307	119655	37556	13460
|QI| = 16	26625	34793	139334	38838	13808
|QI| = 20	38567	260614	349361	48993	14655

Partition-Based GREEDY TOP-DOWN

	10%	20%	30%	40%	50%
|QI| = 4	52497	47531	52139	33378	24969
|QI| = 8	61762	54410	48714	33368	24889
|QI| = 12	75804	57689	51619	33458	25106
|QI| = 16	74730	55289	52617	33543	25211
|QI| = 20	78361	62912	52501	33566	25142

Partition-Based GREEDY BOTTOM-UP

	10%	20%	30%	40%	50%
|QI| = 4	61680	32770	45265	43441	21360
|QI| = 8	72194	101890	69731	45414	21382
|QI| = 12	342360	480370	148636	45454	21371
|QI| = 16	1315672	1709049	162261	45132	21532
|QI| = 20	3421940	4373999	165688	45540	21722

ICS

	10%	20%	30%	40%	50%
|QI| = 4	14754	16376	17830	18653	9176
|QI| = 8	37085	44723	53730	21673	9113
|QI| = 12	133054	97277	90822	21756	9372
|QI| = 16	239875	194063	97100	23663	9528
|QI| = 20	513581	456350	150345	58138	9990

Table 5.2: Runtime (in ms) Across Varying |QI| for 10MB Dataset

Node-Based GREEDY

	10%	20%	30%	40%	50%
1MB	390	474	536	614	666
2MB	815	1016	976	1168	1024
5MB	2800	2992	4531	4530	2714
10MB	17607	19824	72995	37226	13336
25MB	714433	1801255	443535	286312	173917

Node-Based ENUMERATE ALL

	10%	20%	30%	40%	50%
1MB	409	512	599	638	670
2MB	824	1227	1019	1548	1020
5MB	2787	3040	5723	5123	2727
10MB	17686	21023	116100	37497	13527
25MB	699198	2762072	436421	281694	170980

Partition-Based GREEDY TOP-DOWN

	10%	20%	30%	40%	50%
1MB	451	521	567	626	661
2MB	973	995	1031	1170	1075
5MB	5538	5607	5213	4227	4084
10MB	61762	54410	48714	33368	24889
25MB	980582	836657	768169	501911	245478

Partition-Based GREEDY BOTTOM-UP

	10%	20%	30%	40%	50%
1MB	579	765	832	949	675
2MB	1511	2387	1800	2036	1096
5MB	9637	16308	6894	6218	3471
10MB	72194	101890	69731	45414	21382
25MB	1349348	1459566	1698772	707125	289608

ICS

	10%	20%	30%	40%	50%
1MB	1441	1338	1482	1536	718
2MB	3801	2688	2724	3246	1091
5MB	7998	9451	10639	7069	2566
10MB	37085	44723	53730	21673	9113
25MB	526637	584812	557451	240857	91380

Table 5.3: Runtime (in ms) Across Varying Dataset Sizes with $|QI| = 8$

Chapter 6

Conclusion

In an effort toward improving user-computer interaction, this thesis proposed a framework to aid in streamlining user actions. In order to do so, we showed how Navigation Trees can capture both the context and details (in the form of attributes) of the Data Objects that make up single user actions. These Navigation Trees act as learning data that is analyzed and summarized to create Summary Trees which in turn aid the system in predicting future user actions. The Summary Trees act as a template to which current user actions can be compared so that the system can provide useful suggestions for the user's next step.

We have also presented five different generalization algorithms for performing the summarization. The summarization process consists of progressively merging a set of Navigation Trees in a top-down manner. The dynamically-evolving intermediate summary contains numerous Typed Sibling Groups (TSG) which are passed to the generalization algorithm to compute representative Multi-Data Objects and create the next-level TSGs. The two main types of generalization algorithms are Node-based and Partition-based. The Node-based approach (Greedy and Enumerate All) examine individual members of the TSG, independent of each other, in order to find a local-thinnest satisfactory generalization. On the other hand, Partition-

based algorithms (Greedy Top-Down, Greedy Bottom-Up, and ICS) examine the entire TSG simultaneously to find a partition that satisfies the Coverage Threshold.

Extensive experiments were conducted for comparing the runtimes and quality of the summaries produced by each of the presented methods. We examined the effect of varying the Coverage Threshold (CT) and the number of attributes considered on different sizes of clickstream data. While there was no unanimous "best" CT for any dataset, a CT of 30% appears to be a good starting point for further refinement. Moreover, we showed that the size of the attributes considered has greater effect when considering lower values of CT. Our results also suggest that in general, the Partition-based Greedy Top-Down and Bottom-Up algorithms yield better quality summaries than ICS and the Node-based approach.

This thesis presented a first step to summarizing user action sequences. There is much room for future work on improving the efficiency and effectiveness of the presented algorithms. Also, as identified at the end of Subsection 3.2.3, newer approaches with the ability to piece disjoint blocks with mixed attribute combinations still need to be discovered. Such methods may be able to determine a global-thinnest solution and may result in vastly-improved summary quality. Finally, it would be beneficial to test our algorithms on data that more closely resemble our framework's Navigation Trees and also examine the utility of our summaries through real-life user feedback interaction. Doing so would involve building a system specifically for capturing individual user action sequences without discrepancies in distinguishing unique users and grouping related actions.

Bibliography

[1] Gagan Aggarwal, Tomás Feder, Krishnaram Kenthapadi, Rajeev Motwani, Rina Panigrahy, Dilys Thomas, and An Zhu. Anonymizing tables. In *ICDT*, pages 246–258, 2005.

[2] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. volume 1377 of *Lecture Notes in Computer Science*, pages 469–483, 1998.

[3] Roberto J. Bayardo and Rakesh Agrawal. Data privacy through optimal k-anonymization. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 217–228, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Daniel Billsus and Michael J. Pazzani. A hybrid user model for news story classification. In *UM '99: Proceedings of the seventh international conference on User modeling*, pages 99–108, Secaucus, NJ, USA, 1999. Springer-Verlag New York, Inc.

[5] Ricardo Carreira, Jaime M. Crato, Daniel Gonçalves, and Joaquim A. Jorge. Evaluating adaptive user profiles for news classification. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 206–212, New York, NY, USA, 2004. ACM Press.

[6] Laura A. Dabbish, Robert E. Kraut, Susan Fussell, and Sara Kiesler. Understanding email use: predicting action on a message. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 691–700, New York, NY, USA, 2005. ACM Press.

[7] John J. Darragh, Ian H. Witten, and Mark L. James. The reactive keyboard: A predictive typing aid. *Computer*, 23(11):41–49, 1990.

[8] Brian D. Davison. Predicting web actions from html content. In *HYPERTEXT '02: Proceedings of the thirteenth ACM conference on Hypertext and hypermedia*, pages 159–168, New York, NY, USA, 2002. ACM Press.

[9] Brian D. Davison and Haym Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Approaches to Time-Series Problems*, pages 5–12, Madison, WI, 1998. AAAI Press. Proceedings of AAAI-98/ICML-98 Workshop, published as Technical Report WS-98-07.

[10] Susan Dumais, Edward Cutrell, Raman Sarin, and Eric Horvitz. Implicit queries (iq) for contextualized search. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 594–594, New York, NY, USA, 2004. ACM Press.

[11] Benjamin C. M. Fung, Ke Wang, and Philip S. Yu. Top-down specialization for information and privacy preservation. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 205–216, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Ernst Georg Haffner, Uwe Roth, II Andreas Heuer, Thomas Engel, and Christoph Meinel. What do hyperlink-proposals and request-prediction have in common? In *ADVIS '00: Proceedings of the First International Conference on Advances in Information Systems*, pages 285–293, London, UK, 2000. Springer-Verlag.

[13] Joachim Herbst and Dimitris Karagiannis. Workflow mining with inwolve. *Comput. Ind.*, 53(3):245–264, 2004.

[14] A. J. Hundepool and L. C. R. J. Willenborg. Mu- and tau-argus: Software for statistical disclosure control. In *In Third International Seminar on Statistical Confidentiality at Bled*, 1996.

[15] Vijay S. Iyengar. Transforming data to satisfy privacy constraints. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 279–288, New York, NY, USA, 2002. ACM Press.

[16] Benjamin Korvemaker and Russell Greiner. Predicting unix command lines: Adjusting to user patterns. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 230–235. AAAI Press / The MIT Press, 2000.

[17] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Incognito: efficient full-domain k-anonymity. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 49–60, New York, NY, USA, 2005. ACM Press.

[18] Adam Meyerson and Ryan Williams. On the complexity of optimal k-anonymity. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 223–228, New York, NY, USA, 2004. ACM Press.

[19] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, 1996.

[20] B. J. Rhodes and P. Maes. Just-in-time information retrieval agents. *IBM Syst. J.*, 39(3-4):685–704, 2000.

[21] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[22] Pierangela Samarati and Latanya Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, page 188, New York, NY, USA, 1998. ACM Press.

[23] Guido Schimm. Process miner - a tool for mining process schemes from event-based data. In *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 525–528, London, UK, 2002. Springer-Verlag.

[24] J. C. Schlimmer and L. A. Hermens. Software agents: Completing patterns and constructing user interfaces. *Journal of Artificial Intelligence Research*, 1:61–89, November 1993.

[25] Ricardo Silva, Jiji Zhang, and James G. Shanahan. Probabilistic workflow mining. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 275–284, New York, NY, USA, 2005. ACM Press.

[26] Google SiteMapBuilder.NET. www.sitemapbuilder.net.

[27] Latanya Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):571–588, 2002.

[28] Latanya Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, 2002.

[29] W. M. P. van der Aalst and B. F. van Dongen. Discovering workflow performance models from timed logs. In *EDCIS '02: Proceedings of the First International Conference on Engineering and Deployment of Cooperative Information Systems*, pages 45–63, London, UK, 2002. Springer-Verlag.

[30] W. M. P. van der Aalst and A. J. M. M. Weijters. Process mining: a research agenda. *Comput. Ind.*, 53(3):231–244, 2004.

[31] Ke Wang, Philip S. Yu, and Sourav Chakraborty. Bottom-up generalization: A data mining solution to privacy protection. In *ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining*

(ICDM'04), pages 249–256, Washington, DC, USA, 2004. IEEE Computer Society.

[32] A. J. M. M. Weijters and W. M. P. van der Aalst. Process mining: Discovering workflow models from eventbased data, 2001.

Appendix A

Complete Experimental Results

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1510	1260	944	703	658
Max AvgDiff	86.00	79.00	70.17	83.50	82.00
Min AvgDiff	0.00	0.00	0.00	0.00	4.33
Average AvgDiff	18.32	21.03	21.70	19.94	23.94
Max Slack	208	268	249	149	407
Min Slack	0	0	0	0	0
Average Slack	8.00	6.89	5.50	2.97	49.03

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1324	886	705	658	650
Max AvgDiff	88.00	91.80	88.00	88.00	74.50
Min AvgDiff	0.00	0.00	1.00	0.00	0.00
Average AvgDiff	25.27	25.93	24.54	19.81	21.11
Max Slack	244	372	290	122	407
Min Slack	0	0	0	0	0
Average Slack	7.84	8.04	4.80	1.77	15.89

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	74747	21046	6176	1304	689
Max AvgDiff	85.50	70.33	53.67	70.31	48.50
Min AvgDiff	0.00	0.00	0.00	0.00	1.00
Average AvgDiff	16.76	19.05	20.11	19.31	20.90
Max Slack	208	161	199	119	407
Min Slack	0	0	0	0	0
Average Slack	6.51	5.91	3.80	1.91	14.17

Table A.1: 1 MB, 814 Navigation Trees - no limitation on Number of Attributes considered

Node-Based GREEDY

	10%	20%	30%	40%	50%
Runtime (ms)	415	440	477	549	671
Max AvgDiff	98.33	77.53	85.00	79.67	86.00
Average AvgDiff	27.54	25.81	23.12	20.14	23.99
Max Slack	391	310	315	169	407
Average Slack	20.71	12.19	6.69	3.42	47.59

Node-Based ENUMERATE ALL

	10%	20%	30%	40%	50%
Runtime (ms)	417	441	480	550	667
Max AvgDiff	98.33	77.53	85.00	79.67	86.00
Average AvgDiff	27.54	25.87	23.09	20.14	23.99
Max Slack	391	310	315	169	407
Average Slack	20.71	12.29	6.69	3.42	47.59

Partition-Based GREEDY TOP-DOWN

	10%	20%	30%	40%	50%
Runtime (ms)	399	495	515	582	662
Max AvgDiff	107.25	88.00	88.00	88.00	74.50
Average AvgDiff	29.47	27.61	24.39	19.81	21.11
Max Slack	352	372	290	122	407
Average Slack	13.47	9.91	5.12	1.77	15.89

Partition-Based GREEDY BOTTOM-UP

	10%	20%	30%	40%	50%
Runtime (ms)	372	377	567	635	669
Max AvgDiff	88.00	84.04	88.00	70.13	68.50
Average AvgDiff	29.42	26.63	23.29	20.09	20.81
Max Slack	260	231	269	86	407
Average Slack	11.02	8.63	3.89	2.08	13.50

ICS

	10%	20%	30%	40%	50%
Runtime (ms)	525	595	848	862	665
Max AvgDiff	68.14	56.35	36.53	49.00	50.00
Average AvgDiff	27.37	24.70	24.00	22.74	21.99
Max Slack	732	651	569	488	407
Average Slack	26.85	20.82	110.00	72.98	23.54

Table A.2: 1 MB, 814 Navigation Trees: # Attributes Considered = 4

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	390	474	536	614	666
Max AvgDiff	77.00	85.09	81.00	83.50	83.00
Average AvgDiff	25.59	25.91	23.01	19.79	23.97
Max Slack	435	310	249	149	407
Average Slack	15.02	8.36	5.77	2.82	49.03

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	409	512	599	638	670
Max AvgDiff	77.00	85.09	81.00	80.50	83.00
Average AvgDiff	25.88	25.76	22.93	19.78	23.97
Max Slack	435	310	249	149	407
Average Slack	15.37	8.20	5.59	2.82	49.03

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	451	521	567	626	661
Max AvgDiff	88.00	91.80	88.00	88.00	74.50
Average AvgDiff	27.98	26.90	24.61	19.81	21.11
Max Slack	244	372	290	122	407
Average Slack	8.91	8.20	4.82	1.77	15.89

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	579	765	832	949	675
Max AvgDiff	119.00	87.00	120.00	66.67	71.50
Average AvgDiff	27.89	25.40	23.12	19.61	21.07
Max Slack	208	253	199	123	407
Average Slack	9.09	7.98	3.89	1.88	14.32

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1441	1338	1482	1536	718
Max AvgDiff	89.00	69.00	68.25	45.80	58.50
Average AvgDiff	28.12	23.42	21.86	23.62	22.52
Max Slack	732	651	569	488	407
Average Slack	38.92	27.02	35.98	75.71	23.15

Table A.3: 1 MB, 814 Navigation Trees: # Attributes Considered = 8

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	421	582	646	678	671
Max AvgDiff	89.00	120.00	77.00	83.50	82.00
Average AvgDiff	26.60	24.94	22.08	19.70	23.94
Max Slack	210	295	249	149	407
Average Slack	12.00	9.27	5.27	2.84	49.03

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	457	888	1434	944	669
Max AvgDiff	89.00	79.00	77.00	79.50	81.00
Average AvgDiff	25.80	23.35	22.25	20.08	23.90
Max Slack	210	128	249	149	407
Average Slack	12.87	5.35	6.53	3.06	49.03

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	566	578	626	641	666
Max AvgDiff	88.00	91.80	88.00	88.00	74.50
Average AvgDiff	27.31	26.08	24.54	19.81	21.11
Max Slack	244	372	290	122	407
Average Slack	8.52	8.14	4.80	1.77	15.89

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	2112	2244	2396	1117	681
Max AvgDiff	115.00	84.00	83.00	71.10	67.50
Average AvgDiff	23.56	21.60	22.93	18.69	21.03
Max Slack	208	228	199	119	407
Average Slack	7.97	6.64	3.84	1.78	14.17

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	5033	4116	3361	2084	1115
Max AvgDiff	62.06	65.50	42.00	41.00	54.50
Average AvgDiff	28.29	23.35	20.89	23.47	22.38
Max Slack	732	651	569	488	407
Average Slack	33.97	24.14	35.61	77.51	23.15

Table A.4: 1 MB, 814 Navigation Trees: # Attributes Considered = 12

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	454	754	829	706	670
Max AvgDiff	89.00	74.00	73.00	83.50	82.00
Average AvgDiff	25.32	23.32	22.36	19.68	23.94
Max Slack	210	278	249	149	407
Average Slack	10.81	6.85	5.43	2.84	49.03

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	669	1904	15116	1883	704
Max AvgDiff	89.00	85.00	84.21	79.50	81.00
Average AvgDiff	25.30	22.90	22.51	20.04	23.90
Max Slack	210	128	195	149	407
Average Slack	11.09	4.95	4.88	3.08	49.03

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	634	636	659	650	661
Max AvgDiff	88.00	91.80	88.00	88.00	74.50
Average AvgDiff	26.95	26.08	24.54	19.81	21.11
Max Slack	244	372	290	122	407
Average Slack	8.10	8.14	4.80	1.77	15.89

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3999	5408	4497	1233	693
Max AvgDiff	112.00	80.00	85.00	70.31	63.50
Average AvgDiff	23.24	20.99	20.31	19.96	20.97
Max Slack	208	228	199	119	407
Average Slack	7.09	6.76	3.83	1.91	14.17

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	18655	16861	13041	3514	6034
Max AvgDiff	49.71	55.50	59.00	37.00	53.50
Average AvgDiff	23.65	24.34	21.02	23.39	22.31
Max Slack	732	651	569	488	407
Average Slack	123.23	32.08	17.61	77.51	23.15

Table A.5: 1 MB, 814 Navigation Trees: # Attributes Considered = 16

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	488	823	873	715	667
Max AvgDiff	89.00	70.00	70.17	83.50	82.00
Average AvgDiff	24.73	21.80	22.19	19.97	23.94
Max Slack	208	282	249	149	407
Average Slack	10.41	5.81	5.43	2.97	49.03

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1254	42046	182804	2237	670
Max AvgDiff	89.00	79.00	84.21	79.50	81.00
Average AvgDiff	24.69	22.17	22.25	20.01	23.90
Max Slack	208	128	195	149	407
Average Slack	10.65	4.59	4.97	3.08	49.03

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	697	702	672	655	661
Max AvgDiff	88.00	91.80	88.00	88.00	74.50
Average AvgDiff	26.45	26.02	24.54	19.81	21.11
Max Slack	244	372	290	122	407
Average Slack	8.27	8.07	4.80	1.77	15.89

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	7484	9157	4210	1287	689
Max AvgDiff	110.00	87.00	86.00	70.31	59.50
Average AvgDiff	22.97	20.95	20.23	19.63	20.98
Max Slack	208	228	199	119	407
Average Slack	6.85	6.11	3.81	1.92	14.17

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	80925	199634	142693	3723	84477
Max AvgDiff	52.00	92.00	55.00	43.45	51.50
Average AvgDiff	26.91	24.61	20.93	23.30	22.22
Max Slack	732	158	569	488	407
Average Slack	100.75	21.11	17.67	75.71	23.15

Table A.6: 1 MB, 814 Navigation Trees: # Attributes Considered = 20

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	7176	3713	1850	1277	1014
Max AvgDiff	86.00	81.33	56.67	74.50	44.80
Average AvgDiff	17.59	19.52	20.22	19.46	22.48
Max Slack	656	340	616	397	854
Average Slack	14.48	9.09	9.91	6.31	72.33

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3012	1904	1506	1214	1084
Max AvgDiff	88.00	96.67	88.35	60.27	42.00
Average AvgDiff	24.02	23.70	22.88	19.47	20.28
Max Slack	412	868	203	196	854
Average Slack	12.77	12.07	7.74	4.00	25.69

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	282404	124927	12679	2214	1094
Max AvgDiff	76.50	95.33	76.56	62.39	46.00
Average AvgDiff	19.83	20.21	20.23	18.67	20.03
Max Slack	431	528	543	233	854
Average Slack	12.58	17.26	10.57	4.78	23.90

Table A.7: 2 MB, 1709 Navigation Trees - no limitation on Number of Attributes considered

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	776	826	880	958	1026
Max AvgDiff	85.00	84.00	79.41	79.00	46.67
Average AvgDiff	24.78	24.24	21.92	19.76	22.52
Max Slack	398	538	367	397	854
Average Slack	29.09	18.19	10.87	7.33	64.00

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	776	823	884	974	1040
Max AvgDiff	85.00	84.00	79.41	57.00	46.67
Average AvgDiff	24.78	24.24	22.10	19.49	22.52
Max Slack	398	538	367	397	854
Average Slack	29.09	18.19	10.99	8.00	64.00

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	819	883	917	1004	1076
Max AvgDiff	131.00	88.00	88.35	60.27	42.00
Average AvgDiff	26.95	24.93	22.92	19.47	20.28
Max Slack	456	868	203	196	854
Average Slack	18.15	13.34	7.86	4.00	25.69

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	805	921	1049	1246	1089
Max AvgDiff	88.00	119.00	88.00	68.50	46.00
Average AvgDiff	26.00	24.15	21.94	19.85	19.99
Max Slack	334	430	295	301	854
Average Slack	17.31	18.05	9.26	4.43	23.38

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	979	1267	1376	1463	1036
Max AvgDiff	82.00	80.33	33.17	33.17	68.50
Average AvgDiff	25.86	24.70	24.05	22.27	20.60
Max Slack	845	1367	1196	1025	854
Average Slack	30.08	65.83	176.59	121.40	27.54

Table A.8: 2 MB, 1709 Navigation Trees: # Attributes Considered = 4

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	815	1016	976	1168	1024
Max AvgDiff	90.80	81.00	81.00	60.61	44.80
Average AvgDiff	24.91	23.65	21.60	19.43	22.81
Max Slack	438	375	334	397	854
Average Slack	21.59	14.18	9.75	6.94	71.41

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	824	1227	1019	1548	1020
Max AvgDiff	90.80	81.00	81.00	70.33	44.80
Average AvgDiff	24.44	23.07	21.06	19.72	22.80
Max Slack	400	375	367	397	854
Average Slack	18.77	14.02	9.10	7.85	71.41

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	973	995	1031	1170	1075
Max AvgDiff	88.00	96.67	88.35	60.27	42.00
Average AvgDiff	25.67	24.68	22.95	19.47	20.28
Max Slack	412	868	203	196	854
Average Slack	14.06	12.48	7.75	4.00	25.69

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1511	2387	1800	2036	1096
Max AvgDiff	86.00	87.00	87.00	64.97	52.33
Average AvgDiff	22.29	21.33	20.33	19.61	20.07
Max Slack	331	485	326	233	854
Average Slack	15.70	17.77	9.28	4.49	24.12

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3801	2688	2724	3246	1091
Max AvgDiff	49.33	68.67	51.33	33.17	67.50
Average AvgDiff	26.54	24.01	22.16	22.15	20.43
Max Slack	1538	1367	1196	1025	854
Average Slack	154.56	57.25	44.06	118.84	27.72

Table A.9: 2 MB, 1709 Navigation Trees: # Attributes Considered = 8

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	906	1080	1087	1202	1025
Max AvgDiff	80.00	82.06	77.00	60.61	44.80
Average AvgDiff	23.47	23.71	21.29	19.49	22.60
Max Slack	430	375	319	397	854
Average Slack	18.04	13.83	8.33	7.30	72.33

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1106	1606	1930	2158	1022
Max AvgDiff	80.00	80.25	77.20	70.33	44.80
Average AvgDiff	23.81	23.34	20.98	20.37	22.59
Max Slack	406	375	367	397	854
Average Slack	16.73	13.17	8.73	7.83	72.33

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1137	1094	1192	1195	1077
Max AvgDiff	88.00	96.67	88.35	60.27	42.00
Average AvgDiff	25.00	23.81	22.91	19.47	20.28
Max Slack	412	868	203	196	854
Average Slack	13.19	12.05	7.72	4.00	25.69

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	5520	9551	4854	2115	1097
Max AvgDiff	87.00	117.00	84.74	61.39	46.00
Average AvgDiff	21.88	22.53	21.74	19.38	19.99
Max Slack	331	865	487	233	854
Average Slack	14.73	17.99	9.51	4.88	24.12

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	9964	5662	6567	3997	1345
Max AvgDiff	52.00	44.50	54.00	33.17	63.50
Average AvgDiff	25.59	22.77	21.24	23.06	20.33
Max Slack	1538	1367	1196	1025	854
Average Slack	167.67	99.76	44.62	135.61	27.72

Table A.10: 2 MB, 1709 Navigation Trees: # Attributes Considered = 12

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1084	1215	1179	1242	1028
Max AvgDiff	78.00	94.00	98.00	74.50	44.80
Average AvgDiff	23.81	22.63	20.76	19.51	22.55
Max Slack	453	376	319	397	854
Average Slack	17.29	12.22	8.22	6.31	72.33

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1853	3663	7013	2574	1034
Max AvgDiff	78.00	74.00	74.00	71.50	44.80
Average AvgDiff	22.74	21.10	20.92	20.32	22.53
Max Slack	383	376	367	397	854
Average Slack	15.69	11.26	8.66	6.65	72.33

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1358	1276	1279	1209	1100
Max AvgDiff	88.00	96.67	88.35	60.27	42.00
Average AvgDiff	24.84	23.70	22.91	19.47	20.28
Max Slack	412	868	203	196	854
Average Slack	13.17	12.07	7.72	4.00	25.69

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	11496	12579	6142	2184	1104
Max AvgDiff	112.00	70.00	85.00	62.39	46.00
Average AvgDiff	20.53	21.80	20.10	18.54	20.03
Max Slack	431	528	296	233	854
Average Slack	14.60	17.96	7.80	4.67	23.90

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	43191	19804	25361	4367	2459
Max AvgDiff	46.57	65.33	53.25	33.17	59.50
Average AvgDiff	24.64	23.21	20.91	23.00	20.29
Max Slack	1538	1367	1196	1025	854
Average Slack	137.70	84.27	43.40	135.61	27.72

Table A.11: 2 MB, 1709 Navigation Trees: # Attributes Considered = 16

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1191	1349	1501	1288	1026
Max AvgDiff	100.50	94.00	69.00	74.50	44.80
Average AvgDiff	22.40	22.49	20.59	19.49	22.51
Max Slack	488	376	616	397	854
Average Slack	17.04	11.95	9.81	6.31	72.33

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	6909	27009	11477	4100	1027
Max AvgDiff	89.00	70.20	69.00	71.50	44.80
Average AvgDiff	21.41	20.71	20.30	20.18	22.49
Max Slack	383	376	319	397	854
Average Slack	14.91	11.62	9.26	6.58	72.33

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1547	1398	1354	1216	1079
Max AvgDiff	88.00	96.67	88.35	60.27	42.00
Average AvgDiff	24.79	23.70	22.88	19.47	20.28
Max Slack	412	868	203	196	854
Average Slack	13.11	12.07	7.74	4.00	25.69

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	27305	17954	9976	2201	1103
Max AvgDiff	86.00	48.50	86.00	62.39	46.00
Average AvgDiff	20.97	20.88	20.92	18.75	20.03
Max Slack	431	528	200	233	854
Average Slack	13.65	17.53	8.24	4.86	23.90

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	267662	75959	88405	48780	4968
Max AvgDiff	59.67	43.50	48.00	33.17	56.50
Average AvgDiff	24.79	23.50	21.38	22.76	20.27
Max Slack	1538	1367	1196	1025	854
Average Slack	139.02	106.31	42.75	135.69	27.72

Table A.12: 2 MB, 1709 Navigation Trees: # Attributes Considered = 20

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	18881	8080	10669	4488	2667
Max AvgDiff	80.80	85.09	90.80	47.00	38.57
Average AvgDiff	17.37	18.70	19.76	18.89	16.35
Max Slack	1317	1041	879	1210	3346
Average Slack	41.74	33.92	21.72	32.18	94.05

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	18797	9816	6542	4174	4029
Max AvgDiff	88.00	88.00	88.00	56.33	60.25
Average AvgDiff	22.41	21.50	19.88	18.79	16.56
Max Slack	1329	1003	884	1210	3346
Average Slack	39.87	28.07	19.37	24.98	88.47

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	13543483	1513646	32624	7520	3444
Max AvgDiff	77.50	77.50	73.00	57.00	58.33
Average AvgDiff	16.70	18.18	19.91	18.93	15.74
Max Slack	1317	999	791	1077	3346
Average Slack	41.63	32.73	19.86	17.13	89.00

Table A.13: 5 MB, 6692 Navigation Trees - no limitation on Number of Attributes considered

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	2595	2826	2978	4510	2707
Max AvgDiff	69.85	73.00	74.50	50.00	38.57
Average AvgDiff	22.01	20.91	19.51	18.90	16.27
Max Slack	4582	1027	883	1210	3346
Average Slack	83.92	40.70	24.24	44.81	94.73

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	2577	2805	2962	5012	2717
Max AvgDiff	69.85	73.00	74.50	61.00	38.57
Average AvgDiff	22.01	20.91	19.43	19.38	16.27
Max Slack	4582	1027	883	1210	3346
Average Slack	83.92	40.70	23.71	41.61	94.73

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	4721	4528	4719	4163	4098
Max AvgDiff	88.00	88.00	88.00	56.33	60.25
Average AvgDiff	23.46	22.10	19.65	18.79	16.56
Max Slack	2538	1003	884	1210	3346
Average Slack	51.94	30.23	20.90	24.98	88.47

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	4060	3541	4611	6046	3431
Max AvgDiff	85.00	66.00	80.50	69.00	50.00
Average AvgDiff	23.31	16.56	18.88	19.61	16.22
Max Slack	2252	1000	1619	1077	3346
Average Slack	57.09	46.74	25.77	18.24	90.21

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3522	3762	4030	4434	2547
Max AvgDiff	78.00	52.50	47.33	41.00	37.80
Average AvgDiff	19.68	18.68	14.74	14.04	16.14
Max Slack	6022	5353	4684	4015	3346
Average Slack	187.84	115.55	162.53	179.98	104.07

Table A.14: 5 MB, 6692 Navigation Trees: # Attributes Considered = 4

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	2800	2992	4531	4530	2714
Max AvgDiff	82.00	81.00	76.56	54.50	38.57
Average AvgDiff	22.26	20.48	20.40	18.69	16.35
Max Slack	1692	1040	885	1210	3346
Average Slack	61.80	38.10	21.80	42.55	94.05

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	2787	3040	5723	5123	2727
Max AvgDiff	82.00	81.38	76.56	42.57	38.57
Average AvgDiff	22.18	20.79	19.23	18.42	16.35
Max Slack	1692	1040	886	1210	3346
Average Slack	61.99	32.55	21.06	40.22	94.05

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	5538	5607	5213	4227	4084
Max AvgDiff	88.00	88.00	88.00	56.33	60.25
Average AvgDiff	23.40	22.03	19.87	18.79	16.56
Max Slack	1329	1003	884	1210	3346
Average Slack	44.56	28.65	19.31	24.98	88.47

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	9637	16308	6894	6218	3471
Max AvgDiff	77.00	61.00	67.00	60.50	58.33
Average AvgDiff	19.70	19.46	21.35	19.50	16.30
Max Slack	1320	1355	610	1077	3346
Average Slack	54.06	40.98	20.95	17.06	89.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	7998	9451	10639	7069	2566
Max AvgDiff	97.50	51.00	46.00	32.85	37.80
Average AvgDiff	22.31	19.51	14.96	12.75	16.14
Max Slack	6022	5353	4684	4015	3346
Average Slack	112.88	101.44	147.23	155.07	104.08

Table A.15: 5 MB, 6692 Navigation Trees: # Attributes Considered = 8

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	2951	3149	6430	4498	2704
Max AvgDiff	81.00	85.09	76.56	47.00	38.57
Average AvgDiff	21.12	20.19	19.64	19.27	16.35
Max Slack	1705	1041	880	1210	3346
Average Slack	58.63	35.26	20.72	33.36	94.05

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3107	4340	14419	5357	2715
Max AvgDiff	81.00	81.38	76.56	45.00	38.57
Average AvgDiff	20.74	20.20	19.60	19.00	16.35
Max Slack	1705	1041	880	1210	3346
Average Slack	57.19	34.00	20.69	30.29	94.05

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	6850	6640	6212	4248	4126
Max AvgDiff	88.00	88.00	88.00	56.33	60.25
Average AvgDiff	23.37	21.99	19.88	18.79	16.56
Max Slack	1329	1003	884	1210	3346
Average Slack	43.98	28.14	19.37	24.98	88.47

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	66419	91174	18129	6532	3418
Max AvgDiff	89.74	108.00	86.00	67.00	58.33
Average AvgDiff	19.44	17.96	18.11	19.57	16.23
Max Slack	1317	999	883	1077	3346
Average Slack	55.16	39.53	19.56	17.11	89.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	20947	20878	23211	7516	2595
Max AvgDiff	51.00	85.00	75.00	32.85	37.80
Average AvgDiff	18.46	19.68	14.45	12.71	16.16
Max Slack	6022	5353	4684	4015	3346
Average Slack	211.90	71.05	133.89	155.07	103.26

Table A.16: 5 MB, 6692 Navigation Trees: # Attributes Considered = 12

Node-Based GREEDY

	10%	20%	30%	40%	50%
Runtime (ms)	3088	4180	10181	4511	2707
Max AvgDiff	78.00	85.09	77.00	47.00	38.57
Average AvgDiff	20.84	20.01	20.11	19.12	16.35
Max Slack	1709	1041	879	1210	3346
Average Slack	55.79	32.80	20.17	33.31	94.05

Node-Based ENUMERATE ALL

	10%	20%	30%	40%	50%
Runtime (ms)	4452	22723	40588	6671	2718
Max AvgDiff	78.00	91.72	76.56	42.00	38.57
Average AvgDiff	20.91	19.75	19.34	18.71	16.35
Max Slack	1709	1041	880	1210	3346
Average Slack	52.41	31.14	19.57	31.22	94.05

Partition-Based GREEDY TOP-DOWN

	10%	20%	30%	40%	50%
Runtime (ms)	7493	7456	6534	4230	4101
Max AvgDiff	88.00	88.00	88.00	56.33	60.25
Average AvgDiff	23.42	21.90	19.88	18.79	16.56
Max Slack	1329	1003	884	1210	3346
Average Slack	44.21	28.13	19.37	24.98	88.47

Partition-Based GREEDY BOTTOM-UP

	10%	20%	30%	40%	50%
Runtime (ms)	278707	387286	27335	6864	3443
Max AvgDiff	68.00	87.75	55.50	57.00	58.33
Average AvgDiff	19.22	19.33	19.93	19.57	16.13
Max Slack	1317	999	791	1077	3346
Average Slack	50.81	32.05	20.78	16.81	89.00

ICS

	10%	20%	30%	40%	50%
Runtime (ms)	56021	43034	31840	9767	2848
Max AvgDiff	53.00	26.00	45.00	32.85	37.80
Average AvgDiff	18.12	15.75	14.03	12.68	16.14
Max Slack	6022	5353	4684	4015	3346
Average Slack	230.95	335.03	146.90	155.07	103.26

Table A.17: 5 MB, 6692 Navigation Trees: # Attributes Considered = 16

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3579	5098	10303	4528	2715
Max AvgDiff	77.00	85.09	90.80	47.00	38.57
Average AvgDiff	20.55	19.34	19.81	19.09	16.35
Max Slack	1709	1041	879	1210	3346
Average Slack	56.51	32.95	20.57	33.31	94.05

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	12162	111601	62276	21237	2708
Max AvgDiff	77.00	112.00	76.56	41.00	38.57
Average AvgDiff	20.41	18.62	19.18	18.63	16.35
Max Slack	1709	1041	880	1210	3346
Average Slack	50.22	30.39	19.72	31.22	94.05

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	8572	8599	6565	4217	4044
Max AvgDiff	88.00	88.00	88.00	56.33	60.25
Average AvgDiff	23.38	21.48	19.88	18.79	16.56
Max Slack	1329	1003	884	1210	3346
Average Slack	43.80	28.22	19.37	24.98	88.47

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	657418	934456	28009	7415	3442
Max AvgDiff	86.00	70.00	73.00	57.00	58.33
Average AvgDiff	19.43	18.66	19.28	19.27	16.05
Max Slack	1317	999	791	1077	3346
Average Slack	51.94	32.23	20.32	17.15	89.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	316148	151577	34334	49402	4360
Max AvgDiff	48.00	43.13	44.00	32.85	37.80
Average AvgDiff	17.40	19.14	14.54	12.64	16.14
Max Slack	6022	5353	4684	4015	3346
Average Slack	235.11	121.35	147.32	155.07	104.08

Table A.18: 5 MB, 6692 Navigation Trees: # Attributes Considered = 20

Node-Based GREEDY

	10%	20%	30%	40%	50%
Runtime (ms)	224601	102234	71105	36039	12925
Max AvgDiff	101.60	87.08	61.00	69.67	54.00
Average AvgDiff	18.24	18.29	18.10	18.82	20.87
Max Slack	2260	1681	1010	2650	8442
Average Slack	33.75	23.55	21.95	18.39	165.87

Partition-Based GREEDY TOP-DOWN

	10%	20%	30%	40%	50%
Runtime (ms)	128387	71200	51045	30686	24425
Max AvgDiff	131.00	102.00	78.00	81.00	51.00
Average AvgDiff	22.07	20.30	18.71	18.32	16.06
Max Slack	2302	3653	1295	2237	8442
Average Slack	25.61	31.60	21.65	15.47	232.88

Partition-Based GREEDY BOTTOM-UP

	10%	20%	30%	40%	50%
Runtime (ms)	>17449254	17449254	165088	44798	20384
Max AvgDiff	n/a	67.50	85.71	67.25	37.00
Average AvgDiff	n/a	15.42	17.38	18.14	15.37
Max Slack	n/a	2075	6062	2702	8442
Average Slack	n/a	32.59	28.16	16.29	265.00

Table A.19: 10 MB, 16885 Navigation Trees - no limitation on Number of Attributes considered

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	14160	17698	17738	37121	13409
Max AvgDiff	60.00	63.00	88.50	78.00	48.33
Average AvgDiff	21.90	20.47	18.54	18.37	17.76
Max Slack	10296	3147	1006	2650	8442
Average Slack	132.57	64.12	34.90	21.67	180.17

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	14191	17594	17748	36960	13501
Max AvgDiff	60.00	63.00	88.50	78.00	48.33
Average AvgDiff	21.90	20.53	18.54	18.35	17.76
Max Slack	10296	3147	1006	2650	8442
Average Slack	132.57	61.25	34.90	21.52	180.17

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	52497	47531	52139	33378	24969
Max AvgDiff	107.00	102.00	78.00	81.00	51.00
Average AvgDiff	24.12	21.28	18.95	18.32	16.06
Max Slack	4880	3653	1295	2237	8442
Average Slack	51.41	40.83	22.33	15.47	232.88

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	61680	32770	45265	43441	21360
Max AvgDiff	88.00	81.00	67.00	88.00	44.50
Average AvgDiff	24.46	19.63	17.71	17.99	15.77
Max Slack	4359	1691	4919	2702	8442
Average Slack	38.28	49.65	25.89	16.05	265.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	14754	16376	17830	18653	9176
Max AvgDiff	69.00	68.67	35.15	57.00	52.33
Average AvgDiff	21.55	17.54	18.54	20.44	18.40
Max Slack	15196	13508	11819	10131	8442
Average Slack	298.71	209.92	337.63	213.02	203.31

Table A.20: 10 MB, 16885 Navigation Trees: # Attributes Considered = 4

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	17607	19824	72995	37226	13336
Max AvgDiff	74.50	64.00	84.33	78.00	48.33
Average AvgDiff	22.42	20.36	18.48	19.11	17.76
Max Slack	4835	3171	1005	2650	8442
Average Slack	95.68	49.39	28.45	20.43	180.17

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	17686	21023	116100	37497	13527
Max AvgDiff	74.50	74.00	84.33	78.00	48.33
Average AvgDiff	22.31	20.70	18.53	19.00	17.76
Max Slack	4835	3171	1005	2650	8442
Average Slack	87.48	46.83	28.52	20.39	180.17

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	61762	54410	48714	33368	24889
Max AvgDiff	107.50	102.00	78.00	81.00	51.00
Average AvgDiff	24.38	21.09	18.80	18.32	16.06
Max Slack	3211	3653	1295	2237	8442
Average Slack	42.28	34.98	22.00	15.47	232.88

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	72194	101890	69731	45414	21382
Max AvgDiff	120.00	91.00	74.50	75.75	37.00
Average AvgDiff	21.82	18.69	18.56	18.20	15.75
Max Slack	4266	3862	6054	2702	8442
Average Slack	31.20	51.90	36.11	16.06	265.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	37085	44723	53730	21673	9113
Max AvgDiff	80.00	65.00	54.20	50.67	52.33
Average AvgDiff	19.55	18.44	15.76	15.78	18.45
Max Slack	15196	13508	11819	10131	8442
Average Slack	300.84	189.54	300.29	232.60	204.87

Table A.21: 10 MB, 16885 Navigation Trees: # Attributes Considered = 8

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	23829	20418	73575	37525	13395
Max AvgDiff	84.22	79.70	61.00	78.00	57.00
Average AvgDiff	21.56	19.43	18.64	19.11	18.01
Max Slack	4859	3197	1010	2650	8442
Average Slack	86.94	40.60	22.43	20.49	234.68

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	26437	24307	119655	37556	13460
Max AvgDiff	76.60	87.80	67.50	78.00	57.00
Average AvgDiff	21.43	19.32	18.03	18.77	18.01
Max Slack	4859	3197	1010	2650	8442
Average Slack	86.12	39.60	22.30	20.35	234.68

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	75804	57689	51619	33458	25106
Max AvgDiff	131.00	102.00	78.00	81.00	51.00
Average AvgDiff	23.53	20.62	18.79	18.32	16.06
Max Slack	2302	3653	1295	2237	8442
Average Slack	33.59	33.45	21.73	15.47	232.88

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	342360	480370	148636	45454	21371
Max AvgDiff	87.00	68.50	76.50	65.25	37.00
Average AvgDiff	18.97	17.84	17.36	18.36	15.63
Max Slack	5260	3681	6062	2702	8442
Average Slack	30.63	44.92	26.41	15.92	265.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	133054	97277	90822	21756	9372
Max AvgDiff	44.67	56.00	53.50	50.67	52.33
Average AvgDiff	17.49	18.95	16.05	15.74	18.63
Max Slack	15196	13508	11819	10131	8442
Average Slack	844.54	153.06	262.08	232.60	201.89

Table A.22: 10 MB, 16885 Navigation Trees: # Attributes Considered = 12

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	24237	20907	73749	37460	13440
Max AvgDiff	71.33	98.06	61.00	78.00	54.00
Average AvgDiff	21.75	18.61	18.25	18.84	20.87
Max Slack	4866	3198	1010	2650	8442
Average Slack	73.68	36.61	22.10	18.39	165.87

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	26625	34793	139334	38838	13808
Max AvgDiff	70.00	97.06	73.50	78.00	54.00
Average AvgDiff	21.55	18.31	18.01	18.74	21.37
Max Slack	4866	3198	1010	2650	8442
Average Slack	79.28	35.18	20.63	18.51	188.92

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	74730	55289	52617	33543	25211
Max AvgDiff	131.00	102.00	78.00	81.00	51.00
Average AvgDiff	23.07	20.47	18.71	18.32	16.06
Max Slack	2302	3653	1295	2237	8442
Average Slack	32.67	32.69	21.65	15.47	232.88

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1315672	1709049	162261	45132	21532
Max AvgDiff	78.00	64.50	74.59	66.00	37.00
Average AvgDiff	19.14	16.76	17.23	18.31	15.51
Max Slack	5260	2823	6062	2702	8442
Average Slack	29.80	43.69	31.81	16.05	265.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	239875	194063	97100	23663	9528
Max AvgDiff	74.00	76.00	44.00	50.67	52.33
Average AvgDiff	18.30	19.16	15.99	15.75	18.61
Max Slack	15196	13508	11819	10131	8442
Average Slack	557.57	118.08	260.12	230.93	201.89

Table A.23: 10 MB, 16885 Navigation Trees: # Attributes Considered = 16

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	24496	39809	73893	37698	13484
Max AvgDiff	81.50	78.50	61.00	69.67	54.00
Average AvgDiff	21.81	18.50	18.08	18.85	20.87
Max Slack	4866	2010	1010	2650	8442
Average Slack	68.61	25.89	21.85	18.39	165.87

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	38567	260614	349361	48993	14655
Max AvgDiff	89.60	92.12	73.50	69.67	54.00
Average AvgDiff	20.57	18.01	17.66	18.74	21.37
Max Slack	4866	2010	1010	2650	8442
Average Slack	71.40	26.11	20.64	18.51	188.92

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	78361	62912	52501	33566	25142
Max AvgDiff	131.00	102.00	78.00	81.00	51.00
Average AvgDiff	22.81	20.42	18.71	18.32	16.06
Max Slack	2302	3653	1295	2237	8442
Average Slack	31.61	32.15	21.65	15.47	232.88

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3421940	4373999	165688	45540	21722
Max AvgDiff	64.00	68.00	80.33	65.00	37.00
Average AvgDiff	19.87	18.05	17.24	18.20	15.42
Max Slack	5260	2757	6062	2702	8442
Average Slack	30.54	34.04	27.98	15.97	265.00

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	513581	456350	150345	58138	9990
Max AvgDiff	71.50	76.00	41.00	50.67	52.33
Average AvgDiff	21.08	18.25	15.83	15.72	18.60
Max Slack	15196	13508	11819	10131	8442
Average Slack	247.59	144.12	263.76	230.93	201.89

Table A.24: 10 MB, 16885 Navigation Trees: # Attributes Considered = 20

Node-Based GREEDY

	10%	20%	30%	40%	50%
Runtime (ms)	32116599	6812556	903526	278610	169725
Max AvgDiff	83.40	73.00	85.00	79.67	49.50
Average AvgDiff	17.43	17.88	18.27	18.12	17.73
Max Slack	11813	5360	3200	7006	27344
Average Slack	42.81	34.49	31.85	26.08	386.14

Partition-Based GREEDY TOP-DOWN

	10%	20%	30%	40%	50%
Runtime (ms)	2428575	1424873	873666	459948	234434
Max AvgDiff	132.00	83.00	75.50	79.00	60.33
Average AvgDiff	20.70	19.63	18.53	18.60	21.42
Max Slack	7718	11938	3350	5886	27344
Average Slack	31.73	43.34	29.18	23.49	290.52

Partition-Based GREEDY BOTTOM-UP

	10%	20%	30%	40%	50%
Runtime (ms)	>97340527	97340527	5828499	683133	283025
Max AvgDiff	n/a	79.00	83.50	65.80	57.50
Average AvgDiff	n/a	17.84	17.63	17.79	17.41
Max Slack	n/a	9583	2863	7894	27344
Average Slack	n/a	45.15	33.08	24.72	345.59

Table A.25: 25 MB, 54688 Navigation Trees - no limitation on Number of Attributes considered

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	311992	307357	358565	277241	168699
Max AvgDiff	85.41	109.00	82.50	92.50	50.50
Average AvgDiff	20.58	20.96	18.91	18.19	18.09
Max Slack	42840	25693	4387	7006	27344
Average Slack	287.88	110.86	39.45	27.96	381.69

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	321734	315359	368640	284300	173422
Max AvgDiff	85.41	97.00	82.50	92.50	50.50
Average AvgDiff	20.62	21.06	18.82	18.82	18.09
Max Slack	42840	25693	4387	7006	27344
Average Slack	284.05	113.45	40.35	29.02	381.69

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	753069	659811	511513	483531	238827
Max AvgDiff	100.53	83.00	63.50	79.00	60.33
Average AvgDiff	23.24	20.39	18.74	18.60	21.42
Max Slack	16077	11938	20224	5886	27344
Average Slack	84.27	69.11	55.10	23.49	290.52

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	890450	620233	561474	716886	278158
Max AvgDiff	101.00	88.00	94.00	75.50	69.00
Average AvgDiff	22.89	18.03	16.98	18.00	17.87
Max Slack	13821	15964	19770	7894	27344
Average Slack	60.36	69.61	59.17	22.66	345.59

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	180408	190810	183617	210786	92766
Max AvgDiff	63.67	60.50	56.25	37.50	46.33
Average AvgDiff	22.43	17.67	18.84	15.90	21.27
Max Slack	49219	43750	38281	32812	27344
Average Slack	364.31	681.80	573.84	676.92	460.59

Table A.26: 25 MB, 54688 Navigation Trees: # Attributes Considered = 4

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	714433	1801255	443535	286312	173917
Max AvgDiff	113.00	91.00	75.50	79.67	49.50
Average AvgDiff	22.23	20.30	18.49	18.36	17.97
Max Slack	32195	17051	3134	7006	27344
Average Slack	136.03	60.92	29.35	26.27	383.91

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	699198	2762072	436421	281694	170980
Max AvgDiff	113.00	76.00	116.00	67.33	46.00
Average AvgDiff	22.24	19.41	18.22	18.55	17.97
Max Slack	32195	11484	3134	7006	27344
Average Slack	127.41	64.44	28.61	27.72	383.91

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	980582	836657	768169	501911	245478
Max AvgDiff	101.00	83.00	78.50	79.00	60.33
Average AvgDiff	23.01	19.84	18.54	18.60	21.42
Max Slack	12588	11938	3350	5886	27344
Average Slack	61.51	53.98	29.86	23.49	290.52

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1349348	1459566	1698772	707125	289608
Max AvgDiff	125.00	77.67	76.50	70.33	66.00
Average AvgDiff	21.78	18.56	17.66	18.48	17.78
Max Slack	13581	16346	13654	7894	27344
Average Slack	49.75	66.17	59.69	25.52	345.59

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	526637	584812	557451	240857	91380
Max AvgDiff	56.00	74.33	53.50	64.00	46.33
Average AvgDiff	20.65	18.22	18.15	16.10	21.18
Max Slack	49219	43750	38281	32812	27344
Average Slack	655.92	705.89	493.70	761.29	457.38

Table A.27: 25 MB, 54688 Navigation Trees: # Attributes Considered = 8

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1959112	1775607	744359	280722	170857
Max AvgDiff	113.00	72.42	79.00	79.67	49.50
Average AvgDiff	22.13	19.42	18.39	18.26	17.76
Max Slack	36640	17051	3200	7006	27344
Average Slack	118.34	52.01	32.29	26.30	386.14

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	4033531	2776798	1006939	282004	170983
Max AvgDiff	111.00	66.50	92.50	67.33	45.00
Average AvgDiff	22.20	18.57	18.34	18.47	17.75
Max Slack	36640	11484	3134	7006	27344
Average Slack	114.27	52.42	26.88	27.84	386.14

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1217027	966449	874992	470849	240728
Max AvgDiff	128.00	83.00	78.50	79.00	60.33
Average AvgDiff	22.97	19.79	18.53	18.60	21.42
Max Slack	7718	11938	3350	5886	27344
Average Slack	47.40	45.36	29.40	23.49	290.52

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3824181	4140974	4558053	707005	283963
Max AvgDiff	124.00	77.44	71.50	66.00	62.00
Average AvgDiff	19.48	19.85	18.15	17.98	17.73
Max Slack	10569	14583	18689	7894	27344
Average Slack	47.58	63.80	58.36	24.46	345.59

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	2110228	1175701	1252214	239779	91358
Max AvgDiff	37.33	109.00	55.00	60.00	46.33
Average AvgDiff	17.53	19.80	19.13	15.54	21.15
Max Slack	49219	43750	38281	32812	27344
Average Slack	1619.11	199.94	568.94	713.37	457.38

Table A.28: 25 MB, 54688 Navigation Trees: # Attributes Considered = 12

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1961663	2525915	930180	280724	170818
Max AvgDiff	113.00	87.00	79.00	79.67	49.50
Average AvgDiff	20.66	18.58	18.72	18.15	17.73
Max Slack	36640	21204	3200	7006	27344
Average Slack	102.06	63.83	31.93	26.08	386.14

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	4138222	6418700	1807194	283246	171234
Max AvgDiff	71.78	79.00	91.50	67.33	45.00
Average AvgDiff	20.93	18.08	18.29	18.34	17.72
Max Slack	36640	13210	3134	7006	27344
Average Slack	97.20	49.34	26.92	27.64	386.14

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1256828	1029784	899539	485824	242311
Max AvgDiff	128.00	83.00	78.50	79.00	60.33
Average AvgDiff	22.54	19.71	18.50	18.60	21.42
Max Slack	7718	11938	3350	5886	27344
Average Slack	44.70	43.53	29.37	23.49	290.52

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	11005651	13109214	5855111	708125	283935
Max AvgDiff	86.00	66.00	92.50	70.00	60.00
Average AvgDiff	18.15	16.37	17.64	17.82	17.69
Max Slack	9562	14583	2863	7894	27344
Average Slack	46.50	61.79	31.64	24.96	345.59

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3288947	2075728	1439545	242236	91735
Max AvgDiff	79.00	97.50	60.00	56.00	46.33
Average AvgDiff	18.73	19.31	18.42	15.50	21.15
Max Slack	49219	43750	38281	32812	27344
Average Slack	1345.22	193.25	537.98	713.37	457.38

Table A.29: 25 MB, 54688 Navigation Trees: # Attributes Considered = 16

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	3241516	2906904	881323	265175	159754
Max AvgDiff	70.00	74.00	86.00	93.50	48.00
Average AvgDiff	19.72	17.90	17.76	18.35	18.26
Max Slack	42451	19078	3397	10443	27344
Average Slack	111.11	46.03	27.39	27.06	360.65

Node-Based ENUMERATE ALL

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	8418184	12473990	1862795	281407	170180
Max AvgDiff	79.00	86.00	74.50	88.50	48.50
Average AvgDiff	19.97	17.66	17.42	17.74	18.23
Max Slack	36640	15534	4836	10443	27344
Average Slack	105.95	48.38	32.59	27.55	360.65

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	1263859	1116254	868199	448409	232059
Max AvgDiff	106.00	91.00	75.50	79.00	60.33
Average AvgDiff	20.92	19.88	18.72	18.60	21.42
Max Slack	25490	11938	3350	5886	27344
Average Slack	52.60	37.64	26.61	23.49	290.52

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	24979105	38908230	5398037	700310	270852
Max AvgDiff	117.00	85.00	83.00	65.80	57.50
Average AvgDiff	19.24	18.88	17.63	17.77	17.65
Max Slack	29436	17170	2863	7894	27344
Average Slack	56.94	49.55	31.31	24.83	345.59

ICS

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	4380736	3448144	1391588	242873	90958
Max AvgDiff	101.00	85.00	55.00	52.00	46.33
Average AvgDiff	18.91	19.03	18.48	15.46	21.15
Max Slack	49219	43750	38281	32812	27344
Average Slack	1068.95	201.21	575.53	713.37	457.38

Table A.30: 25 MB, 54688 Navigation Trees: # Attributes Considered = 20

Node-Based GREEDY

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	125710536	27261010	2333955	1026144	618838
Max AvgDiff	87.50	95.50	92.33	98.00	61.67
Average AvgDiff	18.34	18.89	19.48	19.30	18.56
Max Slack	23358	21611	32121	11521	52112
Average Slack	39.30	40.22	48.14	24.21	274.23

Partition-Based GREEDY TOP-DOWN

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	10325656	5772992	3778416	1849491	833396
Max AvgDiff	119.00	93.50	89.50	89.33	54.67
Average AvgDiff	21.14	20.86	19.95	19.86	17.04
Max Slack	15316	23426	5825	11521	52112
Average Slack	42.98	44.04	27.68	22.32	250.26

Partition-Based GREEDY BOTTOM-UP

	10%	**20%**	**30%**	**40%**	**50%**
Runtime (ms)	>125710536	55456082	14939727	2809191	1329151
Max AvgDiff	n/a	84.00	87.50	83.00	60.00
Average AvgDiff	n/a	18.65	18.97	19.02	17.65
Max Slack	n/a	11719	36268	15837	52112
Average Slack	n/a	34.40	50.02	20.58	255.90

Table A.31: 50 MB, 104225 Navigation Trees - no limitation on Number of Attributes considered