D and E (Figure 14(b)). After the split, $B.PointList = \{\varnothing\}$, $B.PointsBG$ = 0, $B.PointsEx$ =6. Though there is no correlation invoking attribute A_3 in the event producing variable D, by chance the unexpectedness of $B = U$ (.125, 18, 6) = 1.7 is greater than the unexpectedness of the region containing B as well as the area behind B, (U(.25, 18, 8) = 1.2) thus repair will be needed to produce a more optimal partition. Also, the ConvertToLeaf operation will not be called on A, since Pr $(P(U(A, B)|(U(A', B') \geq 1)))$ is trivially high.

In Figure 14(c), $|E.PointList|$ = 18. First, E checks its neighbors to see if any will form a good combination. The only candidate node is D, since the node produced by a Combine operation must be larger than either of the combined nodes. To determine whether Combine is invoked, we find the most recent common ancestor of both E and D, which is A and check to see if we can combine. This is enough of a discrepancy in the internal distributions that we do not combine and so the Split operation is called, producing H, G, and F (Figure 14(d)).

In Figure 14(e), $|F.PointList|$ = 18. As described above, F checks its neighbors for potential combinations. B, C, G, and H are determined to be potential mates. For each of these, the most recent ancestor is determined and we check for a valid combination. H and G are rejected, since the events processed since H, G, and F were first created clearly show that F belongs with neither of them. Next we check B. The most recent ancestor of both B and F is A, and so all points in B and F are considered when determining whether to combine. There are found to be 9 events in both $(B \cup F) \cap B$ (which is just B) and $(B \cup F) \cap F$ (which is the lower half of F). There is no discrepancy to justify separating the two regions, and so Combine is invoked producing J, C and I (Figure 14(f)).

Similarly, in Figure 14(g) node D reaches 18 points. It is combined with H, producing K. At this point, the entire data set has been considered, and ABC allows nodes in the leaf network to combine freely. The configuration encountered in Figure 14(h) is common, where nodes that happen to cover regions of small probability and lower-than-expected event densities are fractured (like nodes C and I, and D and G), since they rarely acquire enough points to combine with one another. The oscillation stage allows such fractures to heal. During oscillation, these nodes combine, and the system enters the configuration shown in Figure 14(i). Finally, the algorithm terminates.

2.6 Sample Execution

We now illustrate the execution of ABC on an imaginary data set containing three integer-valued attributes. Individually, A_1, A_2 and A_3 are all uniformly distributed, but together A_1 and A_2 are normally distributed about the diagonal $A_1 = A_2$. MaxPoints is chosen as 18, and t is chosen to be 1.

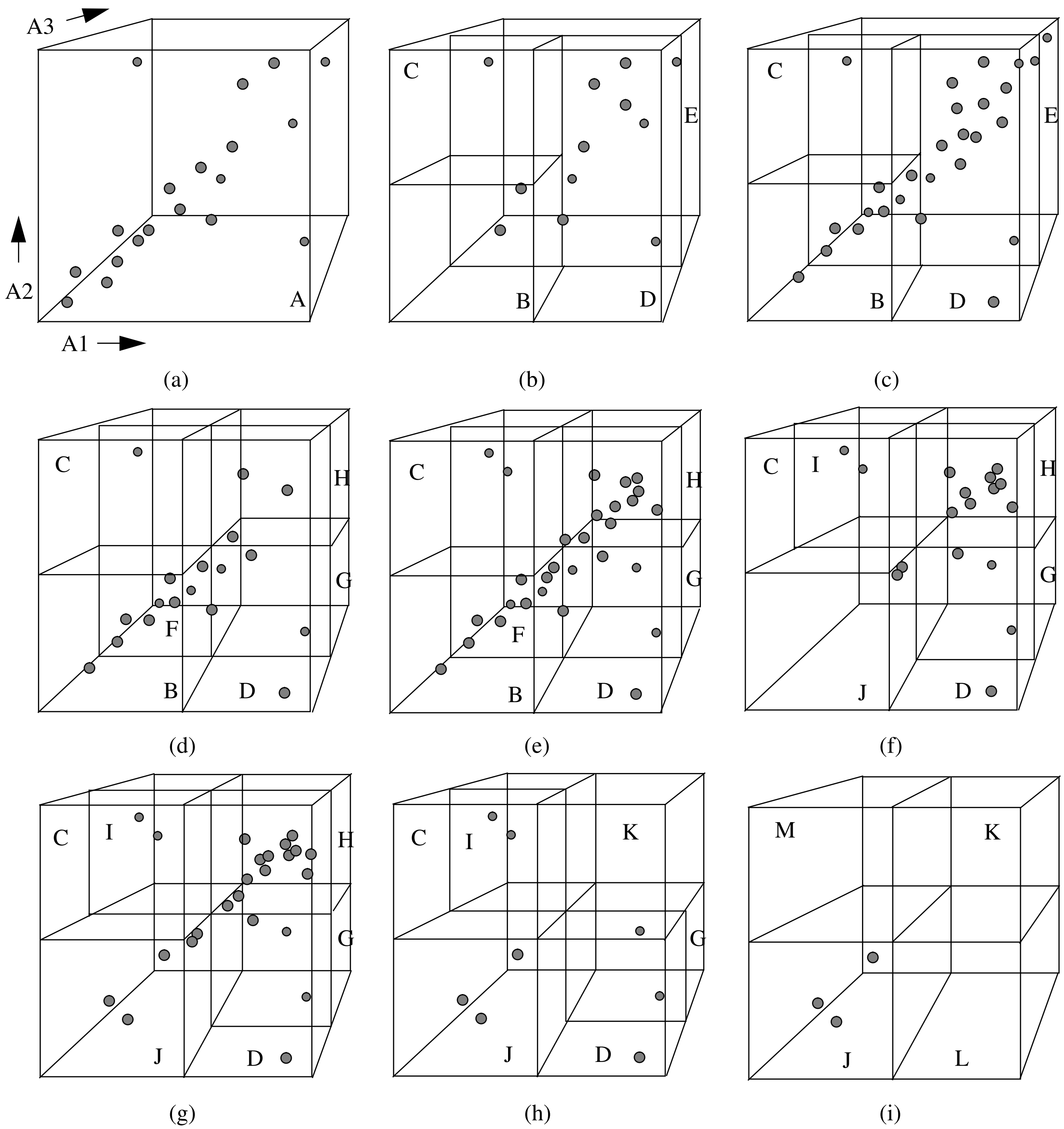


Figure 14: Sample execution of ABC.

As execution is begun, the first 18 tuples are inserted into node *A* (Figure 14(a)). After the 18th tuple has been inserted, the Split operation is invoked. Node *A* is split, producing node *B* as the primary leaf, as well as producing *C*,

Armed with this redefinition of a UR, we can describe the algorithm used to determine whether two URs may combine:

1) Traverse the UT to find the UR *Anc* that is the most recent ancestor of both *I* and *J* such that $I \subset Anc$ and $J \subset Anc$;

2) Disregard the background tuples inherited by both *I* and *J* from *Anc*;

3) Compute $I \cup J$ and I_{Res};

4) If $\Pr(U(I \cup J, I_{Res}) \big| (U((I \cup J)', I_{Res}') \geq t))$ is acceptably low, combine.

By first disregarding the tuples inherited from *Anc* when deciding whether to combine, we consider only those tuples processed since we first detected the trend that caused the regions covered by *I* and *J* to be separated. In the example of Figure 7, the 125 background tuples no longer in memory inherited by both I_{Res} and J_{Res} would then be ignored, and we would find that of the 19 tuples found to be in $I \cup J$ since the region was first fragmented, 18 (or 95%) have fallen in J_{Res}. This is clearly a significant trend and the two regions would not be combined.

5 Execution of Algorithm ABC

Execution of ABC is primarily concerned with insertion of tuples into the UT a tuple at a time. To insert a tuple into a node, first we check to see if the node is a leaf node. If not, we insert it into the correct child in *Children*. If it is a leaf node, we execute the following steps:

1) Insert the event into *PointList*;

2) If |*PointList*| > MaxPoints, check to see if the node can combine with any adjacent nodes in the UT. If so, invoke Combine;

3) If |*PointList*| > MaxPoints and the node was unable to combine, invoke the Split operation.

Finally, execution of ABC is as follows:

1) If attribute histograms are unavailable, make a pass through the data set *E* in order to construct *P*;

2) Make a pass through *E*, inserting the events in *E* into the root node one event at a time;

3) Oscillate. In this step, any adjacent leaf nodes *I* and *J* that meet the combination criteria are allowed to combine until the leaf network reaches a stable state where combinations have ceased;

4) Output the resulting leaf network as an approximate *t*-expected partition of S_D;

the actual data points in each UR are kept in memory. We demonstrate this in Figure 7.

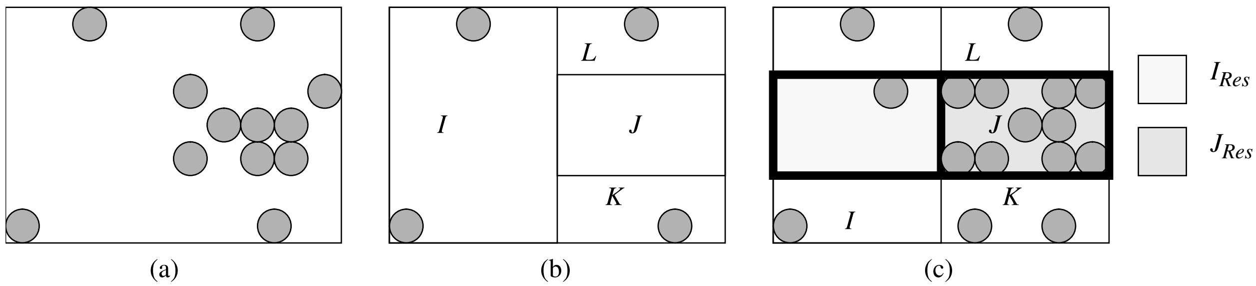


Figure 13: A UR (a) and the URs resulting from its split (b). (c) shows the configuration after 12 additional tuples have been added and the URs that would then be considered in deciding whether to combine I and J.

Figure 13(a) shows a UR containing 512 tuples, only 12 of which are in memory. This UR is then split, producing URs *I, J, K* and *L* in Figure 13(b). Of the 500 tuples not in memory, 250 are assigned to *I* and 125 to *J* since it assumed that the distribution function *P* is an adequate approximation for those 500 points. In Figure 13(b) then, *I* contains 252 tuples and *J* contains 133 tuples. 12 additional tuples are then added to the appropriate URs to produce Figure 13(c). Clearly, these 12 tuples have continued the trend first captured by the split: *J* receives a surprisingly high share of the data. Of the 24 tuples processed in Figure 13, *J* has received 18 or 67%, when only 25% were expected. However, if we compute the UR $I \cup J$ (outlined in bold in Figure 13(c)) as well as the URs I_{Res} and J_{Res}, we find that because the 500 original points have been discarded, *as far as we know*, J_{Res} contains 143 out of the 269 tuples in $I \cup J$, or only 53% when 50% were expected. This is a much less significant deviation. If we check whether *I* and *J* may combined using the process outlined above we may choose to combine them based on this less significant deviation, thereby losing an emerging trend. The difficulty is the 250 tuples we can only assume are distributed throughout $I \cup J$ according to *P*. Since they have been discarded, we cannot be sure of this.

Since it is imperative that we store only a fraction of the actual data in memory, this problem may become serious. To alleviate the situation we will break up the tuple count in a UR in such a way that we can identify those tuples which have been inherited, and so are approximate. We redefine the UR to be as follows:

(*Rect, PointsBG, PointsEx, PointList*)

where

- *PointsBG* is a real number denoting the number of *background* tuples inherited upon creation of this UR;
- *PointsEx* is an integer denoting the number of *exact* tuples verified as falling in this UR.

$J.Rect \subset I.Rect$, then denotes the probability that, given that the unexpectedness of *J* with respect to *I* is greater than *t in the entire data set E*, the unexpectedness of *J* with respect to *I* after a random subset of *N* tuples have been processed is less than or equal to *U (I, J).*

This probability provides us with one way to quantify the level of justification for keeping two URs separated in a UT. We illustrate this in the example below.

(a) (b)

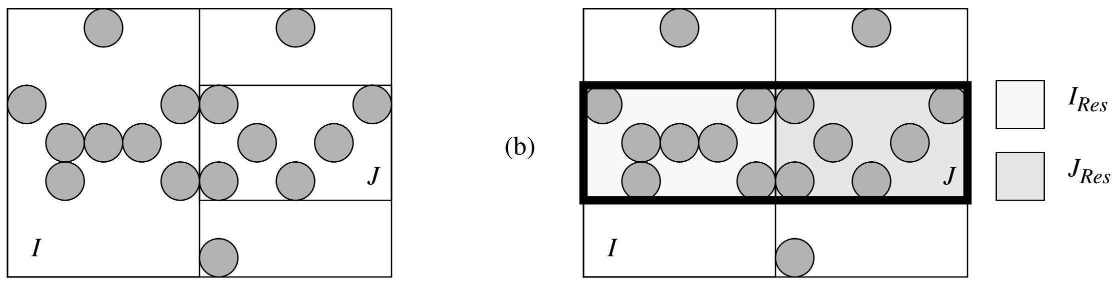
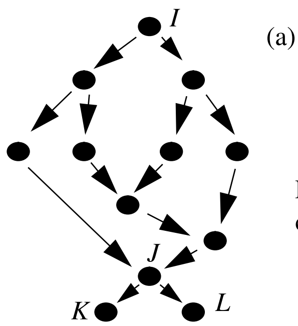


Figure 12. Rectangles *I* and *J* that are candidates for a combine (a) and the URs involved in the check for the validity of the combination (b).

Assume that the two attributes shown have a uniform distribution along each axis. Upon inspection it would seem that the partition shown in Figure 12(a) is suboptimal because of the heavy band of tuples through the center of the diagram. To check whether the URs *I* and *J* may combine to form a more optimal partition, we first compute the UR that would be created through their combination, $I \cup J$. This UR is outlined in bold in Figure 12(b). The next step is to determine the portions of *I* and *J* in this UR, which are labeled I_{Res} and J_{Res}. We then determine whether I_{Res} and J_{Res} have the potential to become highly unexpected in $I \cup J$. This is done by estimating the probability $\Pr(U(I \cup J, I_{Res}) | (U((I \cup J)', I_{Res}') \geq t))$, which quantizes the danger of I_{Res} becoming too unexpected with respect to the region outlined in bold. If this probability is sufficiently low, the danger of losing a region with high unexpectedness is low, and *I* and *J* may then combine with reasonable safety. In Figure 12, since I_{Res} and J_{Res} represent regions of equal probability and both contain 7 of the 14 tuples in $I \cup J$, the danger of either becoming too unexpected in $I \cup J$ is sufficiently low and they are combined.

However, in general we cannot correctly determine the validity of a combination using this method. Combining based on these criteria alone may cause the resultant UT to miss important trends in the data because not all of



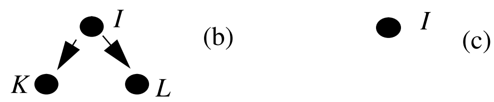


Figure 11: Two logically equivalent UTs, (a) and (b). (c) is the result of the application of the ConvertToLeaf operation on the tree of (b).

For an internal node *I*, Condense is as follows:

1) If there exists an internal node *J* such that forall nodes *K* on any walk from *I* to *J*, $K.Rect \subseteq I.Rect$ then;

2) Set *I.Children* to *J.Children*.

Implementation of the Condense operation in ABC is simple and rather elegant; since tuples are repeatedly inserted into leaf nodes via a recursive traversal of the UT, on any traversal downwards from an internal node *I* we can check the largest UR found. If it is equivalent to *I.Rect*, we condense.

The second operation on internal nodes is the counterpart to combination of leaf nodes. Just as leaf nodes use the Combine operation to repair poor decisions made due to the incremental strategy, internal nodes also have a repair mechanism, ConvertToLeaf. When an internal node recognizes that all of its children have low unexpectedness, it may absorb them and become a leaf node itself. This is the transformation from Figure 11(b) to Figure 11(c).

4 When to Repair?

Thus far we have described the primary mechanisms used in repair: the combination of leaf nodes and the conversion of a set of leaf nodes in the *Children* set of an internal node to a single leaf node through the ConvertToLeaf operation. A critical issue which we have not yet addressed is how to decide when these repair mechanisms should be invoked. Here, we discuss the issues involved in deciding when to combine two leaf nodes. The application of these issues to the task of deciding when to apply to ConvertToLeaf operation is then straightforward.

4.1 Checking For a Safe Combine

When deciding whether to combine two leaf nodes, we will use the following terminology:

Definition 2 Let *N* denote the number of tuples processed thus far. Given two URs *I* and *J* such that

1) *I* is converted from a leaf node to an internal nodes;

2) The URs in *NewRects* are inserted into *I.Children*;

3) All relevant *Nearby* sets are updated to reflect the new configuration.

This steps of a leaf node combination are illustrated in Figure 10.

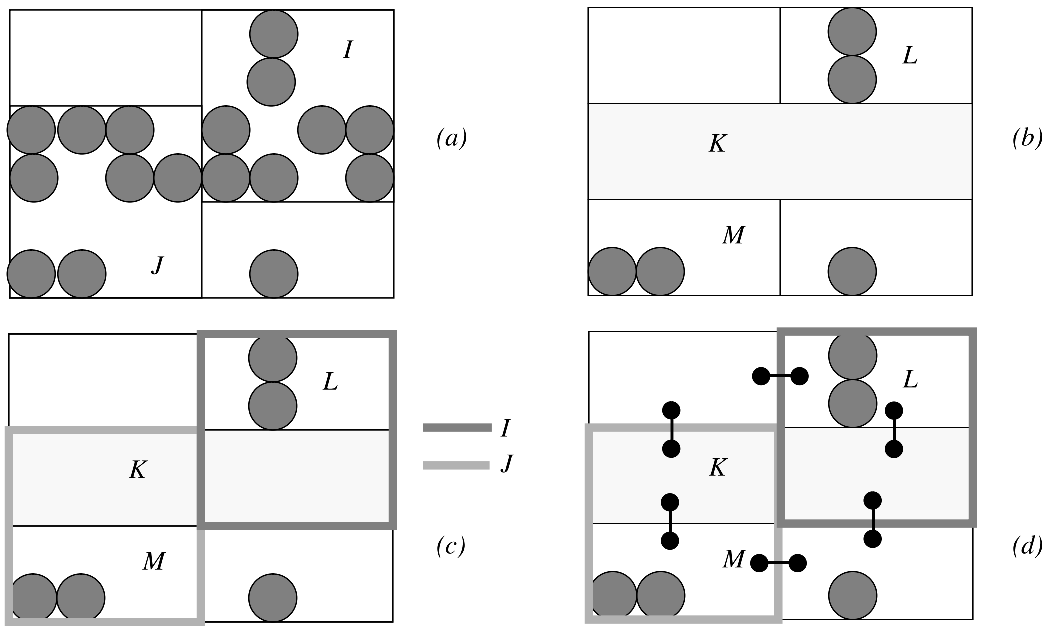


Figure 10: The steps involved in combining the leaf nodes *I* and *J* pictured in (a). First, the URs are combined to form *L, K*, and *M* (b). Then, *L, K* and *M* are inserted into leaf nodes and become children of *I* and *J* (c). Finally, the *Nearby* sets are updated (d).

3.2 Operations on Internal Nodes

Internal nodes are created as the result of the invocation of the split operation or the combine operation. When a leaf node is split, it is converted to an internal node and the leaf nodes created are stored in the *Children* set. When two leaf nodes are combined they are both converted into internal nodes and the leaf nodes created are placed into the *Children* sets of the appropriate parent. As a result of the combination of two nodes, it is possible, indeed common, for a single node to have more than one parent.

This creates the potential for redundant information to be stored in the UT. In Figure 11(a) below, much of the tree structure is not needed, and will slow down access to leaf nodes *K* and *L* from node *I*. This motivates the Condense operation that prunes a redundant portion of the UT and is responsible for the transformation from Figure 11(a) to Figure 11(b).

operation nonetheless. For a large data set we set MaxPoints to be around 400 events. This is a large enough number that even subtle trends in the data may be found, but small enough that the time required by the Split operation is not prohibitive.

3 The Unexpectedness Tree

URs alone lack the control structure necessary to produce an approximate minimal *t*-expected partition of the data set. Access to any given rectangle without some type of indexing structure would be prohibitively slow, requiring a linear search through the rectangles. URs would be unable to quickly find nearby URs to combine with. Ideally the search for neighbors would be a constant time operation because of the frequency with which combines will be considered. Also, without some sort of history, determining exactly when to combine with nearby rectangles will be difficult as well. To solve these problems we introduce an R-tree like structure known as the *unexpectedness tree* (UT). A UT is composed of the following data structures:

1) A *leaf node* is an ordered pair of the form (*Rectangle, Nearby*) where
   - *Rectangle* is an unexpectedness rectangle and
   - *Nearby* is a set of pointers to leaf nodes whose rectangles are adjacent to *Rectangle.Rect*.

2) An *internal node* is an ordered pair of the form (*Rectangle, Children*) where
   - *Rectangle* is an unexpectedness rectangle with |*Rectangle.PointList*| = 0 and
   - *Children* is a set of pointers to nodes whose rectangles are contained in *Rectangle.Rect*.

3.1 Combining and Splitting Leaf Nodes

When two leaf nodes *I* and *J* in a UT combine, first *I.Rectangle* is combined with *J.Rectangle* to produce a new set of URs we will call *NewRects* using the Combine operation for URs described previously. However, three additional steps are needed:

1) *I* and *J* are converted from leaf nodes to internal nodes;

2) The URs in *NewRects* are inserted into new leaf nodes and added to the *Children* sets of *I* and *J*;

3) All relevant *Nearby* sets are updated to reflect the new configuration.

Splitting a leaf node *I* is analogous; first the Split operation for URs is used to split *I.Rectangle* into a new set of URs we again call *NewRects*. Then:

tics is illustrated in Figure 9.

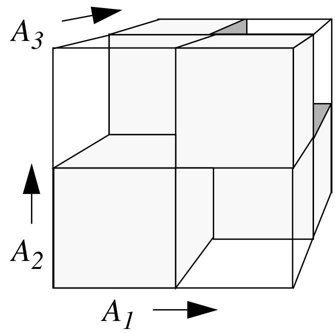


Figure 9: The shaded cubes indicate regions containing a uniform higher-than-expected density of tuple occurrences, while the unshaded regions indicate a uniform lower-than expected density.

In Figure 9, projection of the data to any proper subset of A_1, A_2 and A_3 will show a uniform distribution and suggest that there are no unexpected regions, when in fact there are. Using the construction suggested by Figure 4 it is possible to build a hypercube of any arbitrary dimensionality in such a way that there is a strong, imbedded relationship that is invisible in a projection along any proper subset or superset of the attributes in the hypercube. In general an optimal split may occur along any subset of attributes, at any value of the corresponding attribute, requiring a search that is exponential with respect to the number of attributes. A detailed discussion of the Split operation is beyond the scope of this paper. Because of the exponential time complexity in *d*, with a large number of attributes it is not practical to guarantee an optimal split. Thus our implementation of Split searches for an approximation to the optimal split and uses two auxiliary operations:

- GenCandidateAttributes (*I, AttSet*). Given a UR *I*, this operation generates a relatively small set of attributes, *AttSet*, that may potentially be involved in the split. The strategy is to use a set of heuristics in order to reduce the number of attributes under consideration. Our implementation of GenCandidateAttributes considers the rectangles formed by all pairs of events in *I.PointList* and counts the number of points in each of those rectangles. It runs in $O(d \times |I.PointList|^3)$ time.
- GenSplitRectangle (*I, attSet, J*). Given a UR *I* and a set of attributes *AttSet*, this operation attempts to find the subrectangle of *I.Rect* with maximal unexpectedness, subject to the constraint that it may split only along the attributes contained in *AttSet*. Our recursive implementation of GenSplitRectangle considers many of the combinations of the attributes in *AttSet* (others are pruned using a set of heuristics), counting the number of points in those combinations, and runs in $O(2^{|AttSet|} \times |I.PointList|^3)$ time.

Even though these auxiliary operations are highly optimized, the majority of ABC's running time is spent in the Split

and the subset of those tuples contained in *J.PointList* and *I.PointList* are as depicted in Figure 7(a). Since *I* and *J* have similar internal distributions of tuples, they are combined. Their combination produces the UR *K* of Figure 7(b) (also denoted $I \cup L$). Any tuples contained in *J.N* and *I.N* but not in *J.PointList* or *I.PointList* are distributed among the newly created URs according to *P*. In Figure 7(b), *P* is uniform so of the 12 tuples counted in *J.N* but not present in *J.PointList*, 6 are given to *K*. Upon creation, *K.N* is set to 24 and *K.PointList* is $\{\varnothing\}$ since the tuples in $I \cup L$ were distributed nearly uniformly.

2.3 Splitting Rectangles

In addition to combining with other rectangles, a UR may split into smaller rectangles when it finds that its internal distribution of events does not match closely with *P*. A rectangle *I* is split with the goal of creating a new rectangle *J* such that $J \subset I$ and *U* (*I, J*) is maximized over the set of all possible rectangles that could have been created. Other URs will be created during the split as regions that are contained within *I* but not within *J* are sheared off in order to reduce *I* to *J*.

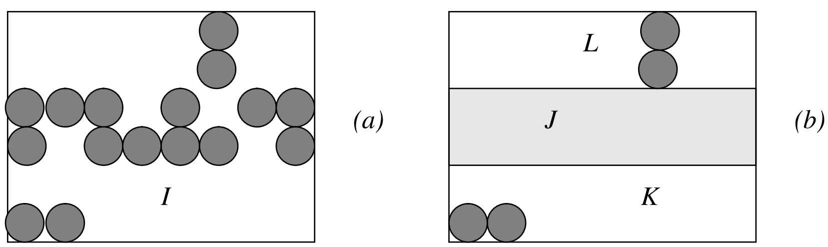


Figure 8. A UR before it is split (a) and after it is split to produce the UR J (b).

In Figure 8(a), *P* predicts a uniform distribution of tuples in the UR *I*. Clearly, this is not the case due to the heavy band of tuples through the center of *I*, and so *I* is split. This split has the by-product of creating *K* and *L* as well. Since splits create regions with uniformly heavy or light distributions of tuples, the tuples from *I.PointList* that fall in the newly created UR *J* are discarded.

In general there are severe difficulties involved in computing an optimal split. Producing such a split is a difficult, exponential optimization problem due to the fact that relationships among several attributes that could be exploited to produce an optimal split may be hidden and invisible in all of the projections onto lower dimensional data spaces, making it necessary to consider all subsets of the set of attributes. A dataset exhibiting these characteris-

The first is that a direct computation of this value is not practical in ABC due to the necessary time complexity involved, since direct computation requires the summation of $O(J.N)$ binomial probabilities for an overall time complexity of $O(J.N^2)$ with a very large hidden constant. Due to the large number of times this operation will be used, anything more than a constant running time is unacceptable. To overcome this problem we use a number of approximation techniques [ref] that allow a constant time computation of unexpectedness.

In order to compute U, we must also compute the function $P\colon \Upsilon \rightarrow [0,1]$ mapping the set of rectangles in S_D to corresponding values. P is implemented using a list of histograms where each histogram represents the probability distribution of an individual attribute. P accepts a single rectangle in S_D and returns the probability that any given event produced by D falls in that rectangle under the assumption that all the attributes in D are independent.

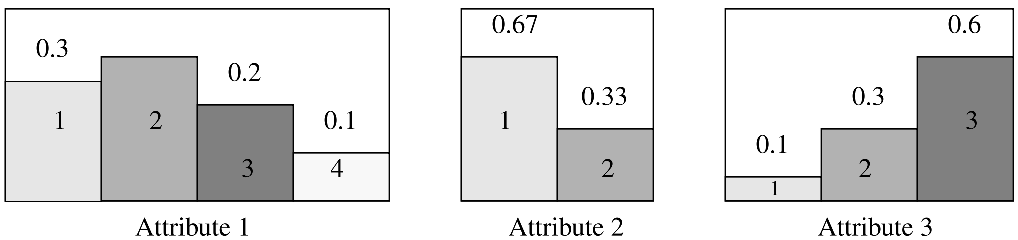


Figure 6: Histograms of attributes in a 3-dimensional data set

For example, consider the list of histograms in Figure 6. Using these histograms, P (((1, 1, 2), (3, 2, 4))) = $(0.3 + 0.35) \times (0.67) \times (0.3 + 0.6) = 0.392$.

2.2 Combining Rectangles

A fundamental ability of the UR is the ability to combine with other URs..

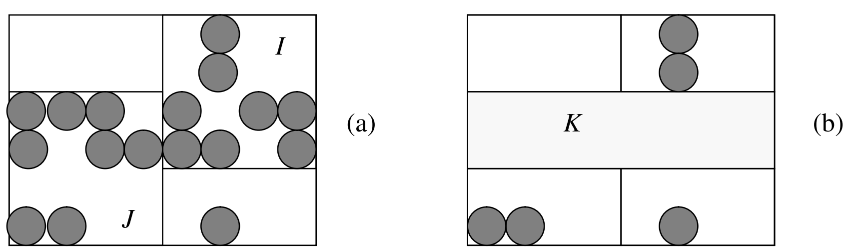


Figure 7. Rectangles I and J before combining (a) and after a combination produces the new UR K (b).

Typically, URs combine when the region produced by the combination will have a distribution that can be accurately approximated by P. In Figure 7, a uniform distribution is predicted by P. Both I and J contain 20 tuples,

isolate that subrectangle.

3) *Combine*. Since the algorithm is relies on local, incremental choices, splits may prove to be incorrect or unnecessary. To correct such errors, URs have the ability to combine with one another when is found that adjacent URs may join to form a single UR without any unexpected subrectangles.

This incremental splitting and combining of URs allows that an approximate minimal *t*-expected partition of the data set be kept at all times.

2 The Unexpectedness Rectangle

An unexpectedness rectangle is a tuple of the form

(*Rect, N, PointList*)

where:

- *Rect* is a rectangular subset of the dataspace S_D. It is expressed as an ordered pair of tuples, (*low, high*).
- *N* is a real number approximating the number of tuples processed thus far that fall in *Rect*.
- *PointList* is a subset of the tuples counted by *N*.

Because we wish to minimze the portion of the dataset that must be kept in memory, not all of the tuples falling in a UR are stored in *PointList*. Those events which are not contained in *PointList* but have been counted in *N* are assumed to be distributed throughout *Rect* according to the probability distribution *P*. This assumption is justified by the definition of a minimal *t*-expected partition: URs are created in such a way that *P* is an adequate description of the event distribution within each rectangle. Later, when reorganization of a set of URs is needed it is done under the assumption that *P* is accurate within each rectangle for those points not contained in *PointList*. Because of the resultant error *N* is only an approximation of the events contained in *Rect*.

Now we discuss the three basic UR operations: *U*, Split, and Combine.

2.1 Computing the Unexpectedness of a UR

First we define the unexpectedness function over unexpectedness rectangles.

Definition 1 Given URs *I* and *J* such that $J \subset I$, ***U* (*I, J*)** = *U* (*P* (*J.Rect*) / *P* (*I.Rect*), *J.N, I.N*). This is the unexpectedness of *I* with respect to *J*.

There are two important considerations in computing *U*(*I, J*).

1 Introduction

The time and space complexity issues involved in computing a minimum *t*-expected partition of S_D are daunting. It is likely that even verification that given a rectangle R is *t*-expected in polynomial time with respect to the number of attributes in S_D is not possible in the general case since unexpected relationships may be hidden in any set of attributes and be undetectable in a subset or superset of those attributes. Further complications arise because ideally we would like to be able to compute a minimal *t*-expected partition in such a way that computation time scales linearly with the size of the data set. Our computation should also adequately handle domains having high dimensionality and we would like our computation to operate successfully on a dataset too large too fit in memory. Given these constraints, it seems clear that computing an exact minimal *t*-expected partition is not feasible. The algorithm proposed here, ABC, is designed to compute an *approximate t*-expected partition; that is, a partition that is both small in size and where the rectangles in the partition are *t*-expected with respect to most or all of their subrectangles.

ABC is an incremental algorithm, at all times maintaining an approximate *t*-expected partition of the portion of the data set that has been processed thus far. As tuples are processed, the partition is constantly adjusted in order to ensure that it remains approximately t-expected, as summarized in Figure 5. The current partition of the dataset is

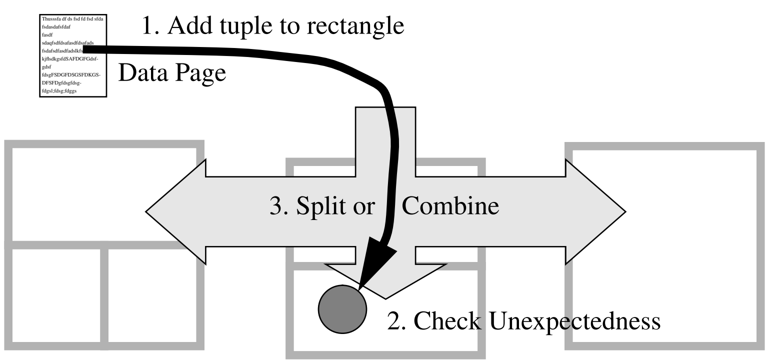


Figure 5: The processing of a tuple by ABC. Tuples are inserted into rectangles one at a time. Each tuple insertion may cause a rectangle to split or combine based on the rectangle's unexpectedness.

represented using a set of *unexpectedness rectangles* (URs). There are three basic operations on a UR:

1) *Compute unexpectedness*. From the information associated with a UR, it is possible to estimate the unexpectedness of the rectangle associated with it, as well as any subrectangle contained in it.

2) *Split*. When it is determined that a UR contains a subrectangle with high unexpectedness, the UR may split to

Elm: An Algorithm to Approximate a *t*-Expected Partition

Christopher Jermaine

Department of Computer Science
University of Arizona
Gould-Simpson Building, Room 721
Tuscon, AZ, 85721
jermaine@cs.arizona.edu
(520) 621-6613
(520) 621-4246 [FAX]

Renée J. Miller

Department of Computer Science
The Ohio State University
2015 Neil Avenue
Columbus, OH, 43210
rjmiller@cis.ohio-state.edu
(614) 292-7027
(614) 292-2911 [FAX]

Abstract

We have previously proposed a new metric of association between values called *unexpectedness* that is based on information content. This paper describes Elm, a novel, tree-based algorithm that builds a model of the data such that the information content, or unexpectedness, of any trend present in the actual data but not in the model is bounded.