

## Using edit distance to analyze card sorts

---

Katherine Deibel,<sup>1</sup> Richard Anderson<sup>1</sup> and Ruth Anderson<sup>2</sup>

(1) Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA

(2) Department of Computer Science, University of Virginia, Charlottesville, VA 22904, USA

**Abstract:** Card sorts are a knowledge elicitation technique in which participants are given a collection of items and are asked to partition them into groups based on their own criteria. Information about the participant's knowledge structure is inferred from the groups formed and the names used to describe the groups through various methods ranging from simple quantitative statistical measures (e.g. co-occurrence frequencies) to complex qualitative methods (e.g. content analysis on the group names). This paper introduces a new technique for analyzing card sort data that uses quantitative measures to discover rich qualitative results. This method is based upon a distance metric between sorts that allows one to measure the similarity of groupings and then look for clusters of closely related sorts across individuals. By using software for computing these clusters, it is possible to identify common concepts across individuals, despite the use of different terminology.

**Keywords:** sorting, card sorts, edit distance, clustering

### 1. Introduction

Many methods have been devised to collect data on a subject's representation of knowledge (Rugg & McGeorge, 1997). One of these methods is the *repeated single-criterion card sort*, or card sort for short. In this knowledge elicitation technique, a participant is given a collection of items (index cards, pictures etc.) that contain semantic information. The subject is asked to partition the cards into categories based upon a *criterion* of the subject's own choosing and is then asked to provide a name for each *category*. Typically, a subject will be asked to sort the cards

multiple times, using a new criterion for each sort. These repetitions reveal different facets of the subject's view of the domain being studied.

A simple example of a card sort is as follows. The subject is presented with a set of nine index cards on which the following animals are listed:

1. Lion
2. Pigeon
3. Dog
4. Ostrich
5. Frog
6. Gorilla
7. Rabbit
8. Penguin
9. Cow

For ease of reference, each card is associated with a number. On the first sort, the subject chooses to partition the cards into three categories: Mammals {1, 3, 6, 7, 9}, Amphibians {5} and Birds {2, 4, 8}. Next, the subject chooses to use the criterion 'Number of legs', which results in the categories Two-legged {2, 4, 6, 8} and Four-legged {1, 3, 5, 7, 9}. The process then repeats until the subject can no longer think of new criteria.

After collecting data from a number of subjects, the task is to interpret the sorts to infer how the subjects understand the domain. A wide range of techniques, both qualitative and quantitative, have been developed to do this. The contribution of this paper is to introduce a new analysis technique based on a measure of similarity between card sorts. By identifying sorts that are close to one another, it is possible to look at how different subjects view similar concepts. It is also possible to find other card sorts that are close to a selected sort, allowing the exploration of a data set and the discovery of new relationships. This analysis technique contrasts with other techniques that have been used, which tend to either depend upon linguistic analysis of category and criterion names or consider co-occurrence of items in groups.

The first part of our paper introduces this new analysis technique. The paper begins with a brief overview of current methods for analyzing card sort data. Section 3 defines the *edit distance* between card sorts and shows how it can be efficiently computed. Section 4 details three applications of the technique which can be used to analyze collections of card sorts: neighborhood search, clique finding, and probing. Section 5 then describes a computer tool developed by the authors for analyzing card sorts using edit distance.

The second part of the paper shows how these techniques are used by applying them to the Bootstrapping data set. This data set, as described in Section 6, was collected as part of an international study of introductory programming students (Sanders *et al.*, 2005). Automatic analysis techniques are particularly beneficial for data sets as large as this one, which contains 1199 sorts. Section 7 shows examples of how these techniques can be used to discover new knowledge about the structure of the data set. These examples include being able to find card sorts that are structurally similar despite having very different criteria and category names, and identifying underlying concepts that occur relatively frequently in the data set.

## 2. Analysis techniques

A number of different approaches have been used in the analysis of card sort data (Gerrard, 1995; Rugg & McGeorge, 1997; Martine, 2000; Gerrard & Dickinson, 2005; Sanders *et al.*, 2005). These approaches look at different aspects of the sorts and provide different types of both qualitative and quantitative information. Here are four different techniques that are commonly used.

- *Structural*: This purely quantitative approach looks at the ‘shape’ of the sorts. This can include looking at the number of categories in each sort as well as the distribution of the sizes of categories. The *names* of categories, criteria, and items placed in a particular category are ignored completely.
- *Item based*: Another quantitative measure looks at how often different items/cards are placed in the same category. While individual items that are often grouped together can be viewed as closely related, no attempt is made to describe the similarities between two entire sorts. Co-occurrence matrices and dendral diagrams (tree-based representations of co-occurrence matrices) are commonly used for this purpose. Again, names of categories, criterion and items are ignored.
- *Category based*: This approach looks at just the category names. As an example, the Bootstrapping study (Petre *et al.*, 2003) analyzed category names to determine which categories occurred most frequently across all the card sorts. This was accomplished through *gist analysis*, a qualitative method in which category (or criterion) names with the same meaning are grouped together. For example, the categories ‘Easy’ and ‘Beginner’ might be grouped together under a gist analysis done on category names. The actual items placed in a particular category are ignored.
- *Superordinate analysis*: This method (Gerrard & Dickinson, 2005) looks at both the category *and* criterion names used in a sort. Typically, this involves partitioning sorts into clusters in which card sorts that use the same criteria (as defined by a gist analysis of the criteria)

are grouped together. Because gist analysis is used, the actual criteria and categories of the sorts within a cluster can differ slightly. For example, a sort with categories ‘Novice’, ‘Amateur’ and ‘Expert’ would be placed with other sorts that used some phrasing of ‘Skill levels’ as a criterion. Note that sorts in the same superordinate cluster need not have the same number of categories. For example, a sort with the categories ‘Beginner’ and ‘Advanced’ would be in this cluster as well. Again, the individual items placed in a particular category are ignored.

Though often used and insightful, these methods are unsatisfactory for comparing a large number of card sorts in the hope of finding underlying concepts that occur commonly across different respondents. Both category-based and superordinate-construct-based analysis can identify some cases of similarity, but these qualitative methods are difficult to do automatically, largely due to the time-consuming nature of gist analysis. In the next section, we propose an automated approach for uncovering structural similarity among sorts. Once a cluster of sorts has been identified as similar, more costly qualitative approaches can be applied to the cluster to discover interesting correspondences.

## 3. Edit distance

The basis for the analysis introduced in this paper is a definition of a distance function (or metric (Rudin, 1976)) on card sorts that measures how far apart two card sorts are. The distance function is *edit distance*: the minimum number of operations needed to convert one card sort into another. The basic operation is to move an item from one group to another. Consider the following example with two sorts  $A$  and  $B$ , both consisting of four groups of cards:  $A = \{A_1, A_2, A_3, A_4\}$  and  $B = \{B_1, B_2, B_3, B_4\}$ . These sorts use a stimulus set of nine items numbered 1 through 9.

$$\begin{array}{ll} A_1 = \{1, 2, 3\} & B_1 = \{1, 2\} \\ A_2 = \{4, 5, 6\} & B_2 = \{3, 4\} \\ A_3 = \{7, 8, 9\} & B_3 = \{5, 6, 7\} \\ A_4 = \{\} & B_4 = \{8, 9\} \end{array}$$

The empty group  $A_4$  is included so the sorts have the same number of groups.

Sort  $A$  can be converted into sort  $B$  by moving items between groups. A minimum set of moves is as follows: move 3 from  $A_1$  to  $A_4$ , 4 from  $A_2$  to  $A_4$  and 7 from  $A_3$  to  $A_2$ .

After these moves, we have

$$A_1 = \{1, 2\}$$

$$A_2 = \{5, 6, 7\}$$

$$A_3 = \{8, 9\}$$

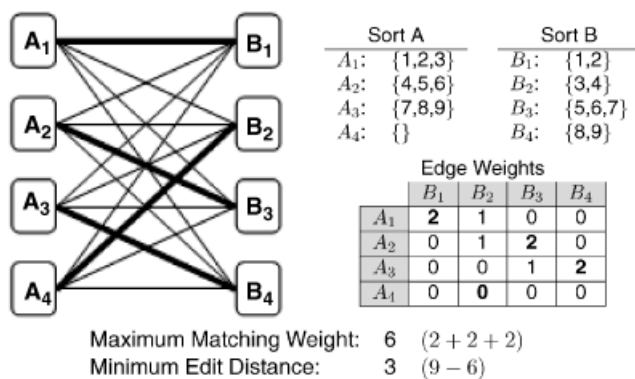
$$A_4 = \{3, 4\}$$

$A$  can now be converted into  $B$  by rearranging the groups. Formally, the rearrangement is specified by a permutation  $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$ , which specifies how the indices of  $A$  map to indices of  $B$ . In our example,  $\pi_1 = 1$ ,  $\pi_2 = 3$ ,  $\pi_3 = 4$  and  $\pi_4 = 2$ .

The edit distance between  $A$  and  $B$  is defined to be the minimum number of item moves to convert  $A$  to  $B$ . It is easy to verify that the edit distance is symmetric, positive, and satisfies the triangle inequality, so it is a *metric*.

There is an efficient algorithm to compute the edit distance. The number of items moved can be minimized by maximizing the number of items not moved. After moving some items between groups of  $A$ , we permute the groups of  $A$  to get  $B$ . An item is not moved between groups if, for some  $i$ , it is in  $A_i$  and  $B_{\pi_i}$ . To minimize the number of moves, a permutation is found that maximizes  $\sum_i |A_i \cap B_{\pi_i}|$ .

This problem can be solved by finding a maximum weight matching in a bipartite graph as shown in Figure 1. A graph on  $A_1 \dots A_n, B_1 \dots B_n$  is built with an edge between  $A_i$  and  $B_j$  of weight  $|A_i \cap B_j|$ . A complete matching associates each vertex in  $A$  with a different vertex in  $B$ . The total weight of the matching is the sum of the edge weights between matched pairs of vertices. The matching then defines a permutation. In the example above, the maximum weight matching is  $(A_1, B_1)$ ,  $(A_2, B_3)$ ,  $(A_3, B_4)$  and  $(A_4, B_2)$ , with edge weights 2, 2, 2 and 0, respectively. Maximum weight matching on bipartite graphs with  $n$  nodes can be solved in  $O(n^3)$  time using the Hungarian algorithm (Papadimitriou & Steglitz, 1982).



**Figure 1:** Using maximum weight matchings to calculate the edit distance between sorts. Edge weights are equal to  $|A_i \cap B_j|$  for  $1 \leq i, j \leq 4$ .

## 4. Applications of edit distance

The most immediate application of the edit distance metric is for determining the similarity between two sorts. This is particularly useful when looking at sorts that use similar criteria. However, the use of the metric is not limited to comparisons. With a way of measuring distance, one can identify neighborhoods of card sorts to find closely related sorts, as well as clusters of card sorts that correspond to any specified concept.

### 4.1. Neighborhoods

Given a sort, a simple approach is to determine which sorts are closest to it. This can be accomplished by enumerating all of the other sorts in order of increasing distance. The  $d$ -neighborhood of sort  $S$  is defined to be the set of all sorts with distance at most  $d$  of  $S$ .

Neighborhoods are particularly useful for understanding individual sorts. If the sort has many close neighbors, it may be part of a common theme in the overall data set. Moreover, looking at the category names of close neighbors can help illuminate what sort  $S$  is getting at. In this case, it is often sufficient to just look at the closest  $K$  sorts, for some value  $K$ , instead of the entire  $d$ -neighborhood.

A choice needs to be made as to what values of  $d$  indicate that two sorts are similar. Generally, this value is a subjective choice by the researcher. However, the number of groups in the sort is a factor. Consider the matching used to calculate the edit distance. For each matched pair of groups, a few mismatches can be tolerated, and the groups will still appear to be similar. With more groups, more discrepancies can occur, increasing the overall distance. With a sort of two groups, values of  $d$  up to 3 or 4 might be of interest, while values of up to 7 or 8 might be appropriate in the case of four groups. For example, in the card sort data used in Section 7, 69% of the sorts of the 26 item stimulus set contained four or fewer categories. Thus, the analysis focused on values of  $d$  from 0 to 7. The appropriate value of  $d$  also depends on the number of items in the stimulus set.

### 4.2. Cliques

Neighborhoods are useful for looking at individual sorts but have a weakness when used for clustering together similar sorts. The set of sorts within  $r$  of a sort  $S$  can contain a pair of sorts at distance  $2r$ . It can easily be the case that sorts of distance  $r$  look similar but sorts of distance  $2r$  do not look similar. A stronger definition of a cluster of sorts is the  $d$ -clique. A  $d$ -clique is a set of sorts such that the distance between any pair of sorts in the set is at most  $d$ . Since the sorts are mutually close, the category names can be used to better understand the reasoning shared by the sorts.

```

FindClique(Graph  $G$ , Vertex  $c$ , int  $maxDist$ )
1: Let  $clique = \{c\}$ 
2: Let  $S_c = \{x \in G : x \neq c \text{ and } dist(c, x) \leq maxDist\}$ 
3: while(  $|S_c| > 0$ )
4:   choose Vertex  $v$  from  $S_c$  using a heuristic
5:    $clique = clique \cup \{v\}$ 
6:   Let  $S_v = \{x \in G : x \neq v \text{ and } dist(v, x) \leq maxDist\}$ 
7:    $S_c = S_c \cap S_v$ 
8: return  $clique$ 

```

**Figure 2:** The clique-finding algorithm. Line 4 is the heuristic step.

To find cliques, a complete weighted graph is defined in which each vertex represents a card sort. The weight of an edge between two respective sorts is the distance between those sorts. Finding a  $d$ -clique begins by removing all edges with weights larger than  $d$  and then searching for cliques within this reduced graph. Since finding maximum cardinality  $d$ -cliques is NP-complete (Garey & Johnson, 1979), we use the heuristic algorithm in Figure 2. The algorithm finds a clique that contains a specified sort.

The algorithm starts with a vertex  $c$  in the clique and maintains a set of vertices  $S_c$  that could be added to the clique while maintaining the maximum distance inside the clique to be at most  $d$ . Line 4 in Figure 2 is the heuristic step of the algorithm. Two heuristics have been implemented for selecting the vertex to add to the clique: adding a random vertex and adding a vertex by a greedy rule. For random selection, a vertex is chosen randomly from the set  $S_c$  of available vertices. The greedy selection rule is to choose a vertex from  $S_c$  that would reduce the size of  $S_c$  by the smallest amount when it is added to the clique. Experience with the two methods has identified different uses for each. In practice, the greedy method tends to find larger  $d$ -cliques than the random heuristic, while the random method is useful for finding a variety of cliques of different sizes.

#### 4.3. Probes

In order to use neighborhoods or cliques, it is necessary to choose a sort to begin with. This limits the researcher to using only the sort criteria found in the data set. A more flexible and disciplined approach for studying a set of card sorts is to create a *probe sort*. A probe sort is a sort specifically constructed by the researcher to represent a sort criterion of interest. Once added to the data set, the neighborhood and clique techniques are used with the probe sort as the focus. Through these means, one can explore the presence of the probe's criterion within the data set. Thus, probing allows the researcher to openly explore and question the data set.

One technique for creating probe sorts is to ask experts (defined based on the field of interest) to complete such

sorts. The respondent may be given only the criterion name, or the names of individual categories may be provided as well. Having multiple experts create the probes gives a richer idea of the criterion in question and may allow detection of different views of the criterion in the overall data set.

#### 4.4. Other clustering approaches

Other applications of edit distance are in the process of being explored. In particular, it is possible that edit distance can be used for clustering or partitioning the entire data set. One goal is to study the use of edit distance as an automatic classifier for superordinate analysis. Currently, two well-known clustering approaches,  $k$ -means clustering (Hartigan, 1975) and hierarchical clustering (Jain *et al.*, 1999) have been implemented. Further study is required to determine how effective these methods are for analyzing card sorts.

### 5. Analysis tool

To support ongoing research, several software tools for calculating and utilizing edit distance have been developed. These tools have been collected into a single, publicly available Windows application, the University of Washington Card Sort Analyzer.<sup>1</sup>

#### 5.1. Features

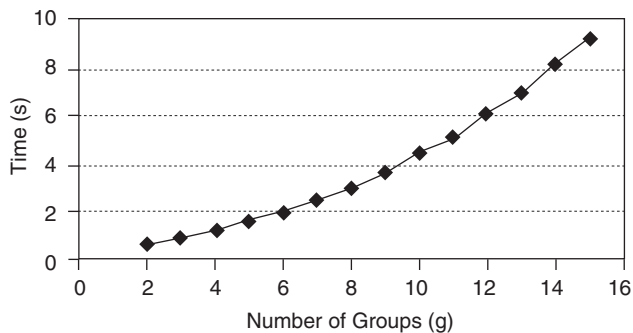
The Analyzer provides a workspace for exploring and studying card sort results. Once a collection of card sorts has been loaded, the user can select any number of sorts to analyze. Several tools are provided to help the user in selecting sorts, including selecting sorts based on the number of groups or based on a text search of the category names.

Once one or more sorts have been selected, the software provides a variety of tools. If the user selects only one sort, the Analyzer can provide information such as the smallest and largest distances from this sort. Similarly, if the user selects several sorts, the user can then use the various edit distance applications mentioned in Section 4: comparing sorts, neighborhood-finding, clique-finding, probing etc.

#### 5.2. Performance

In practice, the primary bottleneck in the Analyzer is calculating the edit distance between the sorts. The Hungarian method for calculating the maximum weight matching between two sorts has a worst-case time complexity of  $O(g^3)$ , where  $g$  is the number of groups in the sort with more categories. In order to analyze  $n$  total sorts, distances between all  $\binom{n}{2}$  pairs are calculated. This task need only be performed once. The matching and

<sup>1</sup>Available at <http://www.cs.washington.edu/research/edtech/CardSorts/>



**Figure 3:** *The effect of the number of groups on computation time. Each data point involves 250 random sorts of 30 cards with the indicated number of groups. Data collected on a 2.8 GHz P4 machine.*

distances are stored in memory for later use. Despite this worst-case runtime of  $O(n^2g^3)$ , in practice this calculation scales well across various values of  $g$  and  $n$ . For example, calculating the pairwise distance for all 1199 sorts (with an average group count of 4.06 ( $\delta = 1.93$ )) in the Bootstrapping data set (see Section 6 for details) takes only 38 seconds on a 2.8 GHz P4 machine.

Scaling  $n$  clearly produces a quadratic increase in computation time, but since the effect of scaling  $g$  is not as clear, the following experiment was performed. For each value of  $g$  from 2 to 15, 250 sorts using 30 cards were generated at random with  $g$  groups each. The  $\binom{250}{2}$  pairwise distances were then calculated.

Figure 3 shows the effect that varying the average group count has on the time it takes to calculate all pairwise distances. As the graph shows, the total calculation time increases superlinearly as  $g$  increases. Plotting the data on a log-log graph reveals a line with a slope of 1.34. Thus, the computation time is modeled by a polynomial of degree significantly smaller than the theoretical cubic worst-case.

## 6. Card sort study

The Bootstrapping Research in Computer Science Education Study was an international study involving researchers at 22 different institutions. Card sorting was used to uncover the knowledge structures of introductory programming students. Each researcher collected card sorts from students at his or her institution using the stimulus set of 26 programming terms shown in Table 1. Further details on the background of the study and the methodology are given in Sanders *et al.* (2005).

Goals of the study were to collect a large data set representing a wide cross-section of institutions and students, and to use the data set to study the conceptual structures of the students. With 22 researchers involved, this goal was achieved; a total of 1199 card sorts was

**Table 1:** *Stimulus set used in the Bootstrapping study*

1	Function	10	Scope	19	Type
2	Method	11	List	20	Loop
3	Procedure	12	Recursion	21	Expression
4	Dependency	13	Choice	22	Tree
5	Object	14	State	23	Thread
6	Decomposition	15	Encapsulation	24	Iteration
7	Abstraction	16	Parameter	25	Array
8	If-then-else	17	Variable	26	Event
9	Boolean	18	Constant		

recorded from 276 individuals. The burden of analyzing a data set this large suggested automation. Thus, the edit distance metric was developed.

## 7. Results

This section contains results obtained by exploring the Bootstrapping data set using the edit distance analysis tools. Each section provides examples of the usefulness of applications of the edit distance metrics.

### 7.1. Edit distance

One of the first uses of the distance metric was to gain an overall picture of the Bootstrapping data set. The average pairwise distance between the sorts was 13.94 ( $\delta = 2.81$ ). Many of these pairs were of small distance; we discovered 22 pairs of distance 0, 48 pairs of distance 1 and 168 pairs of distance 2. Likewise, some pairs were of a large distance: 17 pairs were at a distance of 23.

Another basic use of edit distance relates to qualitative methods like gist and superordinate analysis. As mentioned in Section 2, these methods are used to cluster card sorts based on criterion and category names. Edit distance allows us to determine the degree of internal consistency within these clusters.

For example, one common theme found in the Bootstrapping study was that respondents often sorted the cards in terms of difficulty (i.e. easy versus hard). Using the edit distance analysis tool, gist analysis was simulated by automatically selecting any sort that used either 'easy' or 'hard' in either its criteria or category name. A total of 16 sorts was identified. Due to space limitations, Table 2 only shows the sorts with three groups. The reader should notice that the category names match up well. The first three sorts separate items into categories of 'easy', 'medium' and 'hard', while the last three use the categories of 'easy', 'hard' and 'don't know'.

Despite the similarity of their category names, the distance metric shows that these sorts are actually quite different from each other. The mean distance between the six sorts is 12.87 ( $\delta = 2.03$ ) with minimum and maximum distances of 9 and 15, respectively. Upon closer inspection

**Table 2:** Six card sorts using either 'easy' or 'hard' in their criterion or category names

Simple: 8, 9, 17, 18, 19, 21	Moderately difficult: 1, 2, 3, 4, 11, 13, 16, 20, 24, 25	Hard: 5, 6, 7, 10, 12, 14, 15, 22, 23, 26
Easy to understand: 1, 8, 17, 18, 20, 21, 23, 25	Moderately hard: 9, 11, 13, 14, 16, 22, 24, 26	Hard to understand: 2, 3, 4, 5, 6, 7, 10, 12, 15, 19
Easiest: 8, 9, 13, 14, 17, 18, 19, 20, 21, 24	Not so hard (medium): 1, 2, 3, 7, 10, 16	Things find hard: 4, 5, 6, 11, 12, 15, 22, 23, 25, 26
Easy to learn: 2, 3, 5, 8, 9, 10, 14, 16, 17, 18, 19, 20, 21, 26	Not taught: 11, 22, 23	Very hard to learn: 1, 4, 6, 7, 12, 13, 15, 24, 25
Easy part: 1, 2, 5, 8, 9, 10, 17, 18, 19, 20, 24, 26	Don't know: 4, 6, 13, 14	More challenging to work with: 3, 7, 11, 12, 15, 16, 21, 22, 23, 25
Easy to manipulate: 1, 2, 5, 11, 16, 17, 21, 22, 25	Don't know: 4, 6, 13, 14	Not easy to manipulate: 3, 7, 8, 9, 10, 12, 15, 18, 19, 20, 23, 24, 26

Each row represents a single card sort. Corresponding groups are aligned vertically.

**Table 3:** Subset of the 8-neighborhood of 'Players::Formations::Coaching::Unsure'

<i>Players:</i> 1, 2, 5, 9, 16, 17, 18	<i>Formations:</i> 8, 11, 12, 20, 21, 22, 23, 24, 25, 26	<i>Coaching:</i> 4, 6, 7, 10, 13, 14, 19	<i>Unsure:</i> 3, 15
Building blocks: 1, 2, 3, 9, 14, 16, 17, 18, 21	Medium level: 8, 11, 12, 13, 20, 22, 23, 24, 25, 26	Large system structure: 4, 5, 6, 7, 10, 15, 19	
Scope: 1, 3, 9, 10, 16, 17, 18, 19, 26	Object: 2, 5, 8, 11, 12, 20, 21, 22, 23, 24, 25	Encapsulation: 4, 6, 7, 13, 14, 15	
Data: 5, 9, 11, 16, 17, 18, 19, 22, 25	Flow of control: 1, 2, 3, 8, 12, 20, 21, 23, 24, 26	High level organization: 4, 6, 7, 10, 13, 14, 15	
Appear in memory: 2, 5, 9, 11, 17, 18, 22, 25	Appear inside code itself: 1, 3, 8, 12, 20, 21, 24, 26	Part of a running program: 4, 6, 7, 10, 13, 14, 15, 16, 19	n/a: 23
Modifiable things: 1, 2, 5, 9, 11, 16, 17, 18, 25	Unmodifiable: 8, 10, 12, 14, 15, 19, 20, 21, 23, 24, 26	Not familiar with: 3, 4, 6, 7, 13, 22	
Program parts: 1, 2, 3, 5, 9, 16, 17, 18, 25	Doesn't fit: 6, 8, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24	Program abilities: 4, 7, 15, 19, 26	

of the sorts in Table 2, this comes as no surprise. While the terms Variable (17) and If-then-else (8) are generally considered easy, there is little agreement about where Object (5), Iteration (24) and Array (25) belong. Thus, despite having used nearly the same criteria, these six sorts are in fact quite different from each other.

## 7.2. Neighborhoods

One approach for exploring a card sort data set is to scan the sort criteria for interesting sorts. Determining the neighborhood of the candidate can then reveal how frequently this partitioning of cards actually occurs. Consider the following sort from the Bootstrapping data set:

Doesn't fit	3, 14, 16
Don't know	6, 23, 26
Really simple	9, 17, 18
Takes more steps	1, 2, 8, 13, 21, 24
More complex	5, 10, 20, 25
Takes longer to learn	4, 11, 15, 19
Pain in the butt	7, 12, 22

This sort's criterion partitions the cards according to learning difficulty. While other sorts used a similar

criterion, this sort uses many more categories than the others. Despite the similarity of these criteria, this sort is isolated. Its closest neighbors are at a distance of 11. Despite the subject's use of a common sort criterion, this sort probably exhibits a granularity that was rarely, if ever, expressed in the rest of the data.

Neighborhoods are also useful in understanding the idea the respondent was trying to get at. For example, one intriguing sort produced in the study was 'Players::Formations::Coaching::Unsure'. Table 3 shows some of the sorts in its neighborhood. Although these sorts are at a distance of 6 to 8 from this sort, they reveal the meaning of the respondent's sports analogy. Players are the basic building blocks of code: variables, functions, parameters etc. Formations are how the players are organized: control flow, data structures, threads etc. Finally, coaching describes how the formations are chosen: design, object relationships, types etc.

Another exploratory use of the edit distance metric is to determine how representative a sort is of the data set. For example, consider another interesting sort, 'Words I hate::Words I don't hate', shown in Table 4. This sort is an example of the *emotional response* sorts discussed in Sanders *et al.* (2005). Surprisingly, this sort is structurally similar to many other sorts. In its neighborhood, there are

**Table 4:** 2-neighborhood of ‘Words I hate::Words I don’t hate’

<i>Words I hate:</i> 4, 6, 7, 10, 12, 14, 15, 24, 26	<i>Words I don’t hate:</i> 1, 2, 3, 5, 8, 9, 11, 13, 16, 17, 18, 19, 20, 21, 22, 23, 25	
Abstraction: 4, 6, 7, 10, 14, 15, 26	Physically exists: 1, 2, 3, 5, 8, 9, 11, 12, 13, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25	
Methodologies: 4, 6, 7, 10, 12, 13, 14, 15, 23, 24, 26	Components used when writing a program: 1, 2, 3, 5, 8, 9, 11, 16, 17, 18, 19, 20, 21, 22, 25	
Concepts not in the code: 4, 6, 7, 10, 12, 15, 24	Concepts that are coded: 1, 2, 3, 5, 8, 9, 11, 13, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26	Don’t know: 14

**Table 5:** A 4-clique for the ‘Words I hate::Words I don’t hate’ sort

<i>Words I hate:</i> 4, 6, 7, 10, 12, 14, 15, 24, 26	<i>Words I don’t hate:</i> 1, 2, 3, 5, 8, 9, 11, 13, 16, 17, 18, 19, 20, 21, 22, 23, 25	
Are not predefined: 4, 6, 7, 10, 12, 14, 15, 16, 21, 24	Predefined types or keywords: 1, 2, 3, 5, 8, 9, 11, 13, 17, 18, 19, 20, 22, 23, 25, 26	
Principles: 4, 6, 7, 12, 15, 24	Not principles: 1, 2, 3, 5, 8, 9, 11, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26	
Design concepts: 4, 6, 7, 12, 15, 24	Core programming terms: 1, 2, 3, 5, 8, 9, 10, 11, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26	
Ways to view it: 4, 6, 7, 10, 12, 14, 15, 19, 23, 24	Object: 1, 2, 3, 5, 8, 9, 11, 13, 16, 17, 18, 20, 21, 22, 25, 26	
How it is implemented: 4, 6, 7, 10, 14, 24	How you implement it: 1, 2, 3, 5, 8, 9, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26	
Consider when designing: 4, 7, 10, 12, 15, 24	Should not consider when designing: 1, 2, 3, 5, 6, 8, 9, 11, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26	
Concepts not in the code: 4, 6, 7, 10, 12, 15, 24	Concepts that are coded: 1, 2, 3, 5, 8, 9, 11, 13, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26	Don’t know: 14
General ideas: 4, 6, 7, 12, 15, 24	Leftovers: 1, 2, 3, 5, 8, 9, 10, 11, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26	
Non-tangible things: 4, 6, 7, 10, 12, 13, 14, 15	Tangible things: 1, 2, 3, 5, 8, 9, 11, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26	

Mean distance 3.11 ( $\delta = 1.12$ ).

three sorts of distance 2, nine of distance 3 and 16 of distance 4. All in all, there are 82 sorts within a distance of 6 of this emotional sort.

### 7.3. Cliques

Having identified ‘Words I hate::Words I don’t hate’ as having many close neighbors, cliques are used to better understand the reasoning behind the sort. While the neighborhood in Table 4 suggests that the subject dislikes abstract concepts, one must remember that these sorts are not necessarily mutually close. As it turns out, the three other sorts in that neighborhood are at a distance of 4 from each other, making it really a clique of maximum distance 4. Table 5 shows another clique with a maximum distance of 4. Both cliques show a separation between easy and advanced concepts. The ‘non-hated’ words are implementation terms and concepts that introductory programming students commonly and directly work with: variables, loops etc. ‘Hated’ words, instead, are intangible, more advanced topics that require more thought and work by

the student in order to be properly used: dependency, abstraction, recursion etc.

Cliques are also useful for finding sorts that are structurally similar despite having very different category and criterion names. For example, consider the card sort ‘Mathematics::Computer science’ that is the focus of the 5-clique in Table 6. The clique shows that this criterion is close to a wide variety of other criteria, including ‘Implementation or design issues’, ‘Object-oriented’ and ‘Concrete or abstract data structures’.

This clique is interesting for several reasons. First, consider the match up of categories across the sorts. Computer science students often complain about mathematics requirements, so it is interesting that this clique would match ‘Concrete’ and ‘Things I already know’ with the category ‘Mathematics’. Second, the Mathematics category contains several terms (e.g. variable, function etc.) that have radically different definitions in math and computer science. These different connotations are known to cause problems for students. In particular, numerous student misconceptions about variables have been identi-

**Table 6:** A 5-clique for the 'Mathematics::Computer science' sort

<i>Mathematics:</i> 1, 3, 7, 9, 11, 12, 16, 17, 18, 19, 20, 21, 22, 24, 25	<i>Computer science:</i> 2, 4, 5, 6, 8, 10, 13, 14, 15, 23, 26
Implementation level issues: 8, 9, 10, 11, 12, 16, 17, 18, 19, 20, 21, 22, 24, 25	Design level issues: 1, 2, 3, 4, 5, 6, 7, 13, 14, 15, 23, 26
Things that are: 1, 3, 8, 9, 11, 12, 16, 17, 18, 20, 22, 24, 25	Things that are not: 2, 4, 5, 6, 7, 10, 13, 14, 15, 19, 21, 23, 26
Explicitly stated in text: 1, 3, 8, 9, 11, 12, 16, 17, 18, 19, 20, 21, 24, 25	Not actually stated in program: 2, 4, 5, 6, 7, 10, 13, 14, 15, 22, 23, 26
Concrete data structures: 1, 3, 8, 9, 11, 12, 13, 16, 17, 18, 19, 20, 21, 22, 24, 25	Abstract data structures: 2, 4, 5, 6, 7, 10, 14, 15, 26
How you can manipulate objects (physical things): 1, 2, 5, 8, 9, 11, 12, 16, 17, 18, 19, 20, 21, 22, 24, 25	Properties of objects (ideas): 3, 4, 6, 7, 10, 13, 14, 15, 23, 26
Group I already know: 1, 8, 9, 11, 12, 13, 16, 17, 18, 19, 20, 21, 24, 25	Group I'd like to learn more about: 2, 3, 4, 5, 6, 7, 10, 14, 15, 22, 23, 26
Not object-oriented: 1, 3, 8, 9, 11, 12, 13, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25	Object-oriented: 2, 4, 5, 6, 7, 10, 14, 15, 26

Mean distance 3.96 ( $\delta = 1.02$ ).

fied (Postner & Turns, 2002). Many of these stem from the differences between how mathematics and computer science view the '=' operator (Knuth, 1996). Similar problems occur with the notion of a 'function'. Given all these misconceptions, it might be expected that the Mathematics group would match up with the groups signifying difficulty or abstraction – 'Design level issues', 'Group I'd like to learn more about' – but instead it matches with groups corresponding to simplicity and concreteness.

#### 7.4. Probes

As discussed in Section 4, probing is using specifically constructed sorts to explore the data set. This enables a researcher to ask questions of the data without having to first find an interesting sort to focus on. One question asked by the Bootstrapping study concerned the order that concepts and terminology were learned in introductory computer programming courses. In computer science education, there is an ongoing debate over how to teach introductory programming, focusing heavily on the order that topics are introduced (Bruce, 2004; Astrachan, 2005). The ACM's Computing Curriculum 2001 identifies six distinct pedagogical approaches for introductory programming, based on the initial emphasis of the course (Joint Task Force on Computing Curricula, 2001). Thus, it was of interest to see how views of educators from the different camps were reflected in the data set.

Probing was used to investigate learning order. Two programming instructors were asked to sort the cards according to the criteria 'CS1::CS2::Other courses', where CS1 and CS2 refer to the first two programming courses taken by undergraduate majors in the USA. The two instructors differed greatly in their teaching philosophies. The first instructor followed the objects-first approach,

which focuses on teaching object-oriented programming principles (abstraction, dependency etc.) from the beginning. He produced the following sort:

CS1	1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 24, 25
CS2	4, 10, 22, 26
Other courses	23

The other instructor used a back-to-basics approach, which emphasizes mastery of fundamental notions of programming (iteration conditionals etc.). This view resulted in the sort

CS1	1, 2, 3, 5, 6, 8, 9, 11, 13, 16, 17, 18, 19, 20, 21, 24, 25
CS2	4, 7, 10, 12, 14, 15, 22
Other courses	23, 26

In general, these sorts are fairly similar and have a minimum edit distance of 5. However, there is a substantial difference in the category sizes: 21:4:1 for the objects-first approach and 17:7:2 for back to basics.

5-cliques formed from these probes are shown in Tables 7 and 8. The objects-first clique contains several sorts that separate the cards according to specific advanced programming behaviors: runtime, events, multi-tasking etc. The groups dealing with these technical topics match up with the CS2 group. The other categories, including the CS1 category, are more focused on the general principles of producing code. These principles include both implementation and design terms.

In contrast, the back-to-basics approach shows a separation between concrete and abstract aspects of programming. Under CS2, intangible notions are not just limited to behavior but also concepts like design and recursion. The CS1 group focuses on the concrete aspects

**Table 7:** *A 5-clique found through probing*

<i>Objects-first CS1:</i> 1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 24, 25	<i>Objects-first CS2:</i> 4, 10, 22, 26	<i>Other courses:</i> 23
Writing code: 1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 24, 25, 26	Database and directories: 4, 6, 14, 22, 23	
Not necessarily runtime considerations: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25	Runtime considerations: 14, 26	
General: 1, 3, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25	Windows specific: 2, 23, 26	n/a: 4, 6
Not event handling: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25	Event handling: 23, 26	
Not necessarily multi-tasking: 1, 2, 3, 7, 8, 9, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25	Multi-tasking: 4, 5, 10, 14, 23, 26	Doesn't fit: 6
Mean distance 4.6 ( $\delta = 0.88$ ).		

**Table 8:** *A 5-clique found through probing*

<i>Back-to-basics CS1:</i> 1, 2, 3, 5, 6, 8, 9, 11, 13, 16, 17, 18, 19, 20, 21, 24, 25	<i>Back-to-basics CS2:</i> 4, 7, 10, 12, 14, 15, 22	<i>Other courses:</i> 23, 26
Internal programming structures: 1, 2, 3, 5, 8, 9, 11, 13, 14, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26	Not programming structures: 4, 6, 7, 10, 12, 15, 23	
Can point out in a program: 1, 2, 3, 5, 8, 9, 11, 12, 13, 16, 17, 18, 19, 20, 21, 22, 24, 25	Cannot point out in code: 4, 6, 7, 10, 14, 15, 23, 26	
Physically exists: 1, 2, 3, 5, 8, 9, 11, 12, 13, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25	Abstraction: 4, 6, 7, 10, 14, 15, 26	
Words I don't hate: 1, 2, 3, 5, 8, 9, 11, 13, 16, 17, 18, 19, 20, 21, 22, 23, 25	Words I hate: 4, 6, 7, 10, 12, 14, 15, 24, 26	
Tangible things: 1, 2, 3, 5, 8, 9, 11, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26	Non-tangible things: 4, 6, 7, 10, 12, 13, 14, 15	
Mean distance 3.76 ( $\delta = 1.06$ ).		

of coding: variables, if-then-else etc. While this is similar to the objects-first case, the back-to-basics view of CS1 limits itself only to tangible implementation concepts.

## 8. Conclusion

This paper has introduced a new quantitative technique for analyzing data from card sort experiments. The central idea is to define a distance metric between two card sorts and then to use this metric to look for structural similarity between card sorts. The distance metric that is used is edit distance, which counts the number of items that need to be moved to convert one card sort into another. This distance metric can be computed efficiently, making the technique appropriate for large data sets.

The main use of the technique is to explore a data set based on the structure of the sorts, as opposed to the names of the criteria or categories used. Structurally similar items can be identified, even if they have been labeled differently. This could allow one, for example, to find identical concepts that have been expressed with different terminology. The paper introduced three specific applications of using edit distance for card sort analysis: neighborhood

finding, clique finding, and probing. Neighborhood finding is looking at all sorts within a fixed distance of a given sort. This technique can identify structurally similar sorts in a data set and could be used to try to understand the meaning of a particular sort. Clique finding is similar, except a group of mutually close sorts containing a given sort is found. Clique finding can be thought of as a mechanism for identifying concepts for a given sort. Probing is creating specific 'query sorts' to see how close they are to sorts in the data set. An example of probing would be to have an expert create a sort corresponding to a concept and then see how this concept was expressed in the data set.

These techniques have been applied to a large card sort data set, the Bootstrapping data set, which contains 1199 card sorts of a set of 26 programming terms. The tools allowed exploration of the data set in a way that would not have been possible by hand and identified many correspondences that had not been recognized previously.

## References

- ASTRACHAN, O. (2005) Resolved: objects early has failed, in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St Louis, MO, February.

- BRUCE, K. (2004) Controversy on how to teach CS1: a discussion on the SIGCSE-members mailing list, *ACM SIGCSE Bulletin*, **36** (4), 29–34.
- GAREY, M. and D.S. JOHNSON (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco, CA: W. H. Freeman.
- GERRARD, S. (1995) The working wardrobe: perceptions of women's clothing at work, Master's thesis, Birkbeck College, London.
- GERRARD, S. and J. DICKINSON (2005) Women's working wardrobes: a study using card sorts, *Expert Systems*, this issue.
- HARTIGAN, J.A. (1975) *Clustering Algorithms*, New York: Wiley.
- JAIN, A.K., M.N. MURTY and P.J. FLYNN (1999) Data clustering: a review, *ACM Computing Surveys*, **31** (3), 264–323.
- JOINT TASK FORCE ON COMPUTING CURRICULA (2001) Computing curricula 2001, *Journal of Educational Resources in Computing*, **1** (3).
- KNUTH, D. (1996) Algorithms in modern mathematics and computer science, in *Selected Papers on Computer Science*, Cambridge: Cambridge University Press, Ch. 4, pp. 110–111.
- MARTINE, G. (2000) Quantification of 'look and feel' copyright infringement in Web page design, Master's thesis, University College Northampton, UK.
- PAPADIMITRIOU, C.H. and K. STEGLITZ (1982) *Combinatorial Optimization: Algorithms and Complexity*, Englewood Cliffs, NJ: Prentice-Hall.
- PETRE, M., S. FINCHER, J. TENENBERG, R. ANDERSON, R. ANDERSON, D. BOUVIER, S. FITZGERALD, A. GUTSCHOW, S. HALLER, M. JADUD, G. LEWANDOWSKI, R. LISTER, R. MCCAULEY, J. MCTAGGART, B. MORRISON, L. MURPHY, C. PRASAD, B. RICHARDS, K. SANDERS, T. SCOTT, D. SHINNERS-KENNEDY, L. THOMAS, S. WESTBROOK and C. ZANDER (2003) 'My criterion is: is it a Boolean?': a cardsort elicitation of students' knowledge of programming constructs, Technical Report 6-03, University of Kent.
- POSTNER, L. and J. TURNS (2002) Using facet-based assessment to understand introductory programming students' knowledge, in *Proceedings of FIE 2002*, Boston, MA, November.
- RUDIN, W. (1976) *Principles of Mathematical Analysis*, 3rd edn, New York: McGraw-Hill.
- RUGG, G. and P. MCGEORGE (1997) The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts, *Expert Systems*, **14** (2), 80–93; also this issue.
- SANDERS, K., S. FINCHER, D. BOUVIER, G. LEWANDOWSKI, B. MORRISON, L. MURPHY, M. PETRE, B. RICHARDS, J. TENENBERG, L. THOMAS, R. ANDERSON, R. ANDERSON, S. FITZGERALD, A. GUTSCHOW, S. HALLER, R. LISTER, R. MCCAULEY, J. MCTAGGART, C. PRASAD, T. SCOTT, D. SHINNERS-KENNEDY,

S. WESTBROOK and C. ZANDER (2005) A multi-institutional, multinational study of programming concepts using card sort data, *Expert Systems*, this issue.

## The authors

### Katherine Deibel

Katherine Deibel received a BSc in computer science and mathematics from Butler University in 2001 and an MSc from the University of Washington in 2003. She is currently a PhD candidate at the University of Washington. Her research interests include accessibility issues, educational technology and human–document interaction with a focus on developing tools for supporting people with reading disabilities.

### Richard Anderson

Richard Anderson received his PhD from Stanford University in 1986 and has been on the Computer Science and Engineering Faculty at the University of Washington since 1986. He has worked in many areas of computer science, including analysis of algorithms, parallel computation, scientific computation and symbolic model checking. His current research interests center around computer science education and educational technology.

### Ruth Anderson

Ruth Anderson received a BSc in computer science from the University of North Carolina in 1991 and an MSc in computer science from the University of Washington in 1994. She has been a lecturer in the Department of Computer Science at the University of Virginia since September 2000. Her current research interests are educational technology and computer science education.