

# Reconstruction of Objects Using Lineage

Eldar Sadikov  
Hector Garcia-Molina

June 13, 2009

## Abstract

We study the problem of object reconstruction based on lineage, using photographs as our driving application. In addition to standard forward reconstructions, our model allows inverse transformations, reconstructions that exploit properties (e.g., commutativity), and imperfect reconstructions. With these additions, our model provides many more options for recovering a lost object. However, to choose among many possibly imperfect reconstructions, we need to carefully account for the accompanying “degradation.” In this paper, we propose a model for measuring degradation and a set of composition rules that help us measure the quality of reconstructions. Given this model, we propose an efficient algorithm for finding reconstructions and illustrate how it strikes a balance between efficiency and the quality of the produced results.

## 1 Introduction

Lineage (or pedigree or provenance) describes where data comes from [11, 12]. Lineage plays an important role in data quality, and, as we will discuss here, lineage can also be useful in reconstructing lost, corrupted or unavailable data.

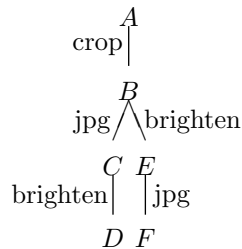


Figure 1: Photo Lineage Tree

To illustrate, consider the lineage tree shown in Figure 1. Object  $A$  represents an original photograph, say in TIFF (uncompressed) format. Photo  $A$  has been cropped, yielding photo  $B$ . Photo  $B$  was given to two people who independently (and in different order) converted the photo to JPG (compressed) format and brightened the image. (Although not shown in the figure, transformations may have parameters, e.g., the amount the image was brightened.) The data objects are represented by capital case letters in the figure. The edges represent transformations. The lineage tree does not describe the internals of these transformations, although some properties of these “black box” transformation (see below) may be known. Much of the work on lineage has been done in the context of relational databases, where the objects are relations and the

transformations are relational operators. However, here we do not restrict ourselves to a relational model.

Lineage can be used in a variety of ways:

- *Query*: Users may query the lineage to discover where the data came from. In our example, a photographer that sees photo  $D$  may want to find the original source photo  $A$ .
- *Lazy Evaluation*: Objects may be materialized on demand. For example, in Trio [15] confidences in a resulting table are computed as-needed, based on the lineage tree.
- *Updates*: If an object changes, the dependent objects can be recomputed. For example, if photo  $A$  changes, we can automatically recompute  $D$  and  $F$  objects, using the lineage.
- *Reconstruction*: If objects are lost or unavailable, they can be reconstructed using lineage. For example, if photo  $D$  is lost but a backup copy of  $B$  exists, we can apply the JPG and brightness transformations on  $B$  to recreate  $D$ . In a distributed environment, reconstructing an object may be an attractive alternative to fetching the object from a remote server. For example, if we have a local copy of  $B$  but need photo  $D$ , it may be faster to apply the JPG and brighten transformations on  $B$ , rather than fetching  $D$  from a remote site.

In this paper we focus on the use of lineage for reconstruction. Typically, reconstructions based on lineage proceed in a forward fashion, e.g., from  $B$  we can reconstruct  $C$ ,  $D$ ,  $E$  or  $F$ . However, in this paper we also allow:

- *Reconstructions that Use Inverses*: We may have available inverse transformations that “undo” the effect of the original. For example, if we have photo  $E$ , we may be able to “darken” the photo to compensate for the brighten action, giving us photo  $B$ .
- *Reconstructions that Exploit Properties*: For example, we may know that the JPG and brighten transformations have the same effect regardless of order. Then, if we need photo  $D$  we can use photo  $F$  instead.
- *Imperfect Reconstructions*: In many cases, the reconstructions illustrated in the previous two items may not be perfect. For example, if we take  $C$  and undo the JPG conversion (i.e., generate a TIFF file), the result is not exactly  $B$ . The result is very similar to  $B$ , so this reconstructed file could be used instead of  $B$  in many cases. As a second example, brightening a photo and then sharpening it is not exactly the same as sharpening it and then brightening it. But again, many people would not notice the difference.

As we will see, allowing inverse transformations and exploiting properties provides many more options for reconstructing objects, making lineage even more valuable. The more options for reconstruction we have, the more likely it is we will be able to recover a lost object, or the less data we will have to fetch remotely to obtain a desired object. However, it is essential to understand the accompanying “degradation” that may occur, so that end users can decide if a degraded version of an object is an acceptable replacement for a missing object. In particular, it is important to track how degradations accumulate. For example, say we reconstruct  $B$  from  $D$  by undoing the brighten and JPG transformations. Perhaps the degradation from each individual undo is acceptable to the user, but will the combined degradations be acceptable?

Coming up with the right model for “degradation” is very challenging. If we model degradation in a very fine, application dependent way, the model quickly becomes unwieldy. For instance, we could compute the degradation between a reconstructed photo and the desired original by computing the fraction of pixels that differ between objects. But unless we have the actual photos, it is very hard to estimate how transformations and their inverses impact such a metric.

Instead, we will take a coarse, user-centric view of degradation. We will assume that the user (or system administrator) will provide an *affinity factor* for each type of action that may degrade an image. For example, say that in Figure 1, photo  $D$  has been sharpened to obtain photo  $G$ . We have lost photo  $D$  and want to reconstruct it. One option is to apply an inverse to  $G$  (blur the photo) to obtain  $D'$ ; a second option is to use photo  $F$ . Say the user gives us an affinity factor of 0.9 for the blur inverse  $D'$ . This number describes how “acceptable”  $D'$  is as a substitute for  $D$ . An affinity value of 1 would mean  $D'$  was just as good as  $D$  for the user, but would not imply the images are identical. A value of 0 would mean that  $D'$  is simply not an option. Similarly, say the user gives an affinity factor of 0.95 for the transposition of brighten and JPG operations. Then we know that  $F$  is a more desirable replacement for  $D$  than  $D'$  (0.95 is larger than 0.9). As we will explain later, the affinity numbers themselves are not important; they are only useful for comparing reconstruction strategies. In other words, the affinity numbers are a compact way of encoding the user preferences.

In this paper we will use photo transformations as our driving application, although our model could be extended to other domains. There are many applications where data “objects” are transformed by black-box functions. In the scientific world, the objects can be results of experiments, satellite images, DNA sequences, etc. In the business world, objects can be spreadsheets, scanned reports, forms, video presentations, building designs, etc. Although our model could be applied in many of these domains, our work was motivated by our BioACT Project [2], where we are helping field biologists manage large numbers of photographs (of animals and plants). The biologists currently transform their photos in many ways (tagging them, cropping them, changing format, etc.), and share them with other biologists, who also edit the photos. Currently they do not keep lineage, and the result is quite frankly a mess: it is not clear where photos came from, how similar photos relate, or which of the many versions of a photo to use for a study, and where those photos are located. (We suspect many photographers are in this same situation.) Lineage can address all these issues. In particular, the transformation model we present in this paper can help biologists know what photos are redundant, what version to use for a study, and how to recover lost photos.

An important question that arises whenever lineage is discussed is: who records the lineage? Clearly, tools that manage objects (photos) need to record the transformations they make. In fact, formats like Adobe’s Extensible Metadata Platform (XMP) [17] already support lineage by enabling a “log” of the performed actions to be saved within the file. The feature of storing edit history within a file has existed in Photoshop [9] for a few years now. Moreover, Adobe, in one of its most recent products, Lightroom [1], has not only provided a way to save and view file’s edit history, but also provided an option to go back to earlier versions across different sessions. However, there have been no attempts from Adobe and others to support recovery of files from copies with a subset of edits or support imperfect reconstructions, in general.

Standards need to be developed so that different tools record lineage in shared formats. In this paper we do not address the lineage recording problem. We assume lineage is available, and we explore what can be achieved if lineage is available, hopefully incentivizing tool developers to continue the trend of recording lineage automatically.

In summary, the contributions of this paper are:

- A transformation model with properties that allows many possible reconstructions;
- An affinity model along with the rules for composition that captures “degradation” that may accompany reconstructions;
- An algorithm for the proposed transformation and affinity models for finding reconstructions; and
- An evaluation of the algorithm on a set of representative scenarios.

## 2 Model

### 2.1 Basic Definitions and Notation

We use “objects” and “images” interchangeably, since from now on we talk about objects strictly in the context of photographs. As discussed earlier, an image is either newly created or produced from another image via a sequence of one or more *transformations* such as cropping, brightness adjustment, sharpening, or format conversion. A transformation  $y$  of image  $X$  is a unary function that takes  $X$  as an argument and produces another image  $Y$  as output:  $Y = y(X)$ . A *production*,

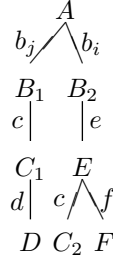


Figure 2: Production Tree

*tree* describes how an image is transformed. Its nodes represent versions of the image, and its edges represent transformations. Figure 2 shows one such tree where node  $A$  is the original source image, labels on the edges such as  $b_j, b_i, \dots, f$  are specific transformations, and nodes  $B_1, B_2, \dots, F$  are  $A$ 's versions. We use letters at the beginning of the alphabet for objects and transformations, and use letters at the end of the alphabet for variables representing objects and transformations. Capital case letters,  $A, B, C, \dots$ , will denote objects and lower case letters,  $a, b, c, \dots$ , will denote transformations, where  $a$  is a transformation that produced  $A$  (unless  $A$  has no parent as in the production tree of Figure 2),  $b$  is a transformation that produced  $B$ , etc.. Subscripts after the object name, e.g.,  $C_1$  and  $C_2$ , differentiate objects produced by the same transformation. Copies are treated as separate versions and copying is just another type of a transformation. Accordingly, if  $c$  is a copy transformation,  $C_1$  is a copy of  $B_1$  and  $C_2$  is a copy of  $E$ .

A *production* of object  $Z$  from a *base* object  $X$ , where both  $X$  and  $Z$  belong to the same production tree, is a sequence of transformations  $\langle a, b, \dots, z \rangle$  applied on  $X$  such that  $z(y(\dots(a(X)))) = Z$ . We use double struck lower case letters, e.g.,  $\mathfrak{q}$ , with a possible subscript to denote transformation sequences and double-struck capital case letter  $\mathbb{P}$  with a possible subscript to denote productions. When the order of transformations in  $\mathfrak{q}$  is not known, we use a bar accent. Hence,  $\bar{\mathfrak{q}}$  is a bag of transformations rather than an ordered list. When specifying a production's transformations, we omit parenthesis and write transformations in the order they were applied, thus representing  $z(y(\dots(a(X))))$  as  $Xa\dots yz$ . For example, in the production tree of Figure 2,  $D$ 's production from base  $C_1$  is  $C_1d$  and  $F$ 's production from base  $A$  is  $Ab_i e f$ . Alternatively, using double struck notation, we can replace  $Ab_i e f$  by  $\mathbb{P}$ , where  $\mathbb{P} = A\mathfrak{q}$  and  $\mathfrak{q} = b_i e f$ .

Transformations often use parameters, e.g. brightness is adjusted by 10% or by 20%. We refer to the parameters using a subscript after the transformation name, e.g.  $x_1$  is transformation  $x$  invoked with parameter 1. If there is more than one parameter, we combine them into a vector and refer to it with a lower case letter, e.g.  $i$ , also added as a subscript after the transformation name. For example, transformation  $x_i$  may be a crop using dimensions  $i$ . In Figure 2, both objects  $B_1$  and  $B_2$  were produced with the same transformation  $b$  but different parameter vectors  $j$  and  $i$ , respectively. Note that we use subscripts in two different ways: the subscript of an object (e.g.  $B_2$ ) is used to distinguish the object from related ones, while the subscript of a transformation (e.g.  $b_i$  or  $b_j$ ) refers to the parameters used by the transformation. When a transformation does

not take parameters or parameters are not important in our discussion, we omit them ( $c, d, e, f$  in our example).

## 2.2 Goal

Each version of an object in a production tree may be hosted on a different computer or device and that computer or device may not be available. Thus, a particular version  $X$  can be lost, unavailable, or too expensive to fetch at any given point in time. Accordingly, our goal is to reconstruct  $X$ , i.e., to produce  $X$  from other versions in its production tree with the minimum possible “loss”. Although cost (defined as cost of fetching an object or cost of performing transformations) may be an important factor when evaluating “loss”, in this work, we focus on minimizing lost quality or, as we will soon define, value to the end user.

What do we need for this type of reconstruction? Here are the basic requirements:

- Knowledge of the production tree. For each node - its location and identity (but not necessarily, its content); for each edge - identity of the transformation with its parameter values.
- Availability of one or more nodes in the tree.
- Ability to apply some of the transformations in the tree using the properties and rules defined in the next section.

Note that we do not put any restrictions on where or how this information is stored or retrieved. The production tree could either be stored with each version in full, or “lazily” computed from partial information stored with each version.

## 2.3 Transformation Model

In order to generate reconstructions, we can exploit certain properties of transformations, which we describe in the following sub-sections. Transformations are not required to have all or any of these properties.

### 2.3.1 Reproducibility

A transformation  $x$  is *reproducible* if its complete definition is available and  $x$  is *deterministic*, i.e., gives the same result when applied with the same parameters and input image. A transformation may not be reproducible, if for instance, it was done via a third party software which is not available, or if the transformation was applied by a human and yields non-deterministic results (e.g. manual “red eye” removal). In the rest of the paper, we only consider reproducible transformations. If some transformations are not reproducible, then they should be removed from consideration by the algorithms of Sections 3.2–3.4.

### 2.3.2 Reversibility

A transformation  $y$  has an *inverse transformation*  $y^{-1}$  if  $Xyy^{-1} = X$ . For example, if  $X$  is a photo in RGB color format, conversion to CYMK format has an inverse transformation that can give us back the original  $X$  with no loss. If the forward transformation  $y_i$  used parameter  $i$ , we assume that the inverse is invoked with the same parameter, i.e.,  $y_i^{-1}$ .

An inverse transformation may not always give us an identical copy of the original object. If we can tolerate some “loss in quality” in the reconstructed image, we will be able to consider many more reconstructions. To capture the notion of a lossy inverse, we introduce a conceptual function  $V(X', X)$  that describes how valuable or “useful”  $X'$  is to the end user in place of  $X$ . Function

$V(X', X)$  returns values in  $[0, 1]$ , where 1 indicates that  $X'$  has the same value to the users as  $X$ , and 0 indicates  $X'$  has no value. We then associate with a transformation  $y$  an *affinity factor*  $\alpha(y) \in [0, 1]$  such that  $\alpha(y) \leq V(Xyy^{-1}, X)$ . Note that affinity  $\alpha(y)$  is a lower bound on the value of  $Xyy^{-1}$  relative to  $X$ , and hence we are using affinities as beliefs in the classical Dempster-Shafer model [10]. (We chose this lower bound formulation because it simplifies operations with affinities, as we will see.) Note that absence of an inverse can be encoded in the  $\alpha$  value, that is, if there is no inverse for  $y$ ,  $\alpha(y) = 0$ .

In general,  $\alpha(y)$  may depend on the parameters used when invoking  $y$ , or even on the image on which  $y$  is used. For example, darken may be a good inverse (high affinity) for brighten, as long as the image was not brightened too much. We can work with parameter dependent affinities, i.e.,  $\alpha(y_i)$ , since when we are exploring reconstructions we know the parameters involved. However, we do not consider affinities that depend on the actual images, since when we are exploring reconstructions we do not have all images.

As discussed in the introduction, affinity values are a compact way for a user or a system administrator to define how desirable certain types of transformations are. Since affinity values are only used to compare reconstructions, we believe it is feasible for the user or administrator to give such general guidelines. For example, if  $y^{-1}$  introduces less degradation than  $z^{-1}$ , then  $\alpha(y)$  should be larger than  $\alpha(z)$ . We return to the problem of affinity selection in Section 2.3.6.

### 2.3.3 Compensation

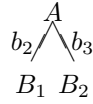


Figure 3: Compensation Example

A transformation  $z$  is *compensable* if for any  $z_i$  and  $z_j$ , there is a compensating  $z_k$  such that for any object  $X$ ,  $Xz_j = Xz_iz_k$ . To illustrate, consider the production tree in Figure 3. Let transformation  $b$  be brightness adjustment and, accordingly,  $b_2$  be increasing brightness of  $A$  by 2% and  $b_3$  be increasing brightness of  $A$  by 3%. Say brightness values can be added to produce the same result. Then we can compensate  $B_2 = Ab_2$  to  $B_3 = Ab_3$  via  $b_{3-2} = b_1$ . Applying  $b_1$  may be better than applying  $b_2^{-1}$  (with a possible degradation) followed by  $b_3$ .

The compensating parameter  $k$  in  $Xz_j = Xz_iz_k$ , may not always be calculated as a simple difference. Thus, we define a *compensating parameter function*  $f_z(i, j) = k$  for any transformation  $z$ , that returns a compensating  $k$  for any applied parameter  $i$  and desired parameter  $j$  or  $\phi$  if no compensation is necessary. For example,  $f_z(i, j)$  can be a numeric difference ( $j - i$ ) for transformations whose parameters can be simply added or it can be a set/bag difference ( $j - i$ ) for transformations like adding tags to an image. Alternatively, for transformations whose effect is determined by the last parameter applied such as resizing, renaming, changing resolution, or changing color pixel depth (posterizing) of an image, to compensate we always reproduce, so  $f_z(i, j) = j$  for any  $i$  and  $j$ .

As with inverse transformations, there may be a certain “loss” when applying a compensation, which we will describe with an *affinity factor*  $\beta \in [0, 1]$ . That is,  $\beta(z_i, z_j) \leq V(Xz_iz_k, Xz_j)$  for any object  $X$ , transformation  $z$  and  $k = f_z(i, j)$ . If  $z$  cannot be compensated, we say that  $\beta(z) = 0$ . If affinity is parameter dependent, we write  $\beta(z_i, z_j)$ .

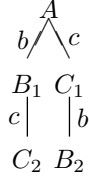


Figure 4: Commutativity Example

### 2.3.4 Commutativity

We can capture even more types of reconstructions, if in addition to the individual properties of transformations, we consider interactions between different transformations. More specifically, transformations  $y$  and  $z$  *commute* if  $Xyz = Xzy$  for any image  $X$ . For example, consider the production tree in Figure 4: say  $b$  is a crop and  $c$  is a JPG conversion. Using our notation,  $C_2 = Abc$  and  $B_2 = Acb$ . In this case, we know that  $b$  and  $c$  commute, thus if  $B_2$  is unavailable, we can use  $C_2$  in place of  $B_2$  as the two are identical.

In some cases, however, changing the order of two transformations may not give exactly the same result. For example, if we sharpen a photo and then adjust brightness, the result although quite similar will be different than if we adjust brightness and then sharpen. Let  $\gamma(y, z) \in [0, 1]$  be the affinity factor for moving transformation  $y$  ahead (to the right) of  $z$ . That is,  $\gamma(y, z) \leq V(Xyz, Xzy)$ . Note that  $V$  may not be symmetrical, thus  $V(Xyz, Xzy) \neq V(Xzy, Xyz)$  and  $\gamma(y, z) \neq \gamma(z, y)$ . As before, if affinity depends on parameters, we write  $\gamma(y_i, z_j)$ , and an affinity of zero means the transformations do not commute.

Commutativity can also be applied to individual transformations as well. Transformation  $y$  is *self-commutative* if it commutes with itself when invoked with different parameters:  $Xy_iy_j = Xy_jy_i$  where  $i$  and  $j$  are two different parameters. For example, adding tags to an image is a self-commutative transformation. If there is a loss, affinity factor  $\gamma(y_i, y_j) \leq V(Xy_iy_j, Xy_jy_i)$  applies.

### 2.3.5 Affinity Rules

In this section we define two fundamental rules for manipulating affinities. As a first step we define some notation. Let  $\mathbb{P}_1 \xrightarrow{\pi} \mathbb{P}_2$  mean that that some origin production  $\mathbb{P}_1$  provides at least  $\pi$  value in place of some destination production  $\mathbb{P}_2$ , i.e.,  $\mathbb{P}_1$  *derives*  $\mathbb{P}_2$  with affinity  $\pi$ . For example,  $\mathbb{P}_1 = \mathbb{P}_2$  will imply  $\mathbb{P}_1 \xrightarrow{1} \mathbb{P}_2$ . With this notation, the three properties we have defined for transformations can be summarized by the first three lines of Table 1.

Table 1: Affinity Rules

| Rule Name             | Definition   |
|-----------------------|--|
| Reversibility         | $\mathbb{P}xx^{-1} \xrightarrow{\alpha(x)} \mathbb{P}$   |
| Compensation          | $\mathbb{P}x_ix_k \xrightarrow{\beta(x_i, x_j)} \mathbb{P}x_j$ , where $f_x(i, j) = k$   |
| Commutativity         | $\mathbb{P}xy \xrightarrow{\gamma(x, y)} \mathbb{P}yx$   |
| Affinity Conservation | If $\mathbb{P}_1 \xrightarrow{\pi} \mathbb{P}_2$ , then $\mathbb{P}_1x \xrightarrow{\pi} \mathbb{P}_2x$  |
| Affinity Transitivity | If $\mathbb{P}_1 \xrightarrow{\pi_1} \mathbb{P}_2$ and $\mathbb{P}_2 \xrightarrow{\pi_2} \mathbb{P}_3$ , then $\mathbb{P}_1 \xrightarrow{\pi_1\pi_2} \mathbb{P}_3$ |

To motivate our first rule, say, we have two objects, one is  $A$  and the other one produced via  $Abb^{-1}$ . We know that the value of  $Abb^{-1}$  in place of  $A$  is at least  $\alpha(b)$ . Now suppose we apply a cropping transformation  $c$  to both objects (with the same parameters). How valuable is  $Abb^{-1}c$  in

place of  $Ac$ ? Our first rule states that the affinity of the cropped objects continues to be  $\alpha(b)$ , i.e.,  $V(Abb^{-1}c, Ac)$  is still at least  $\alpha(b)$ . This indeed is reasonable, since  $c$  is the same transformation in both productions and the value function tells us “how desirable” one object is in place of the other. More formally, if some production  $\mathbb{P}_1$  provides at least  $\pi$  value in place of another production  $\mathbb{P}_2$ , then  $\mathbb{P}_1x$ , where  $x$  is some arbitrary transformation, provides at least  $\pi$  value in place of  $\mathbb{P}_2x$ . We call this rule *affinity conservation*, as shown in the fourth line of Table 1.

Our second rule, *affinity transitivity*, tells us how affinity values can be composed. For any productions  $\mathbb{P}_1$ ,  $\mathbb{P}_2$ , and  $\mathbb{P}_3$  and affinity values  $\pi_1$  and  $\pi_2$ , if  $\mathbb{P}_1 \xrightarrow{\pi_1} \mathbb{P}_2$  and  $\mathbb{P}_2 \xrightarrow{\pi_2} \mathbb{P}_3$ , it must be true that  $\mathbb{P}_1 \xrightarrow{\pi_1\pi_2} \mathbb{P}_3$  (fifth line of Table 1).

To see how we use this rule, again consider Figure 2 and suppose we want to reconstruct  $B_1 = Ab_j$  from  $E = Ab_i e$ . If  $e$  is a reversible transformation and  $b$  is a compensable transformation such that  $f_b(i, j) = k$ , one possible reconstruction is to apply  $e^{-1}b_k$  to  $E = Ab_i e$ . To see how desirable this reconstruction is, we need to bound the value of  $Ab_i e e^{-1}b_k$  with respect to  $Ab_j$ . We can bound this affinity in 4 steps:

1.  $Ab_i e e^{-1} \xrightarrow{\alpha(e)} Ab_i : V(Ab_i e e^{-1}, Ab_i) \geq \alpha(e)$   
Reversibility
2.  $Ab_i e e^{-1}b_k \xrightarrow{\alpha(e)} Ab_i b_k : V(Ab_i e e^{-1}b_k, Ab_i b_k) \geq \alpha(e)$  Affinity Conservation
3.  $Ab_i b_k \xrightarrow{\beta(b_i, b_j)} Ab_j : V(Ab_i b_k, Ab_j) \geq \beta(b_i, b_j)$   
Compensation
4.  $V(Ab_i e e^{-1}b_k, Ab_j) = V(Ab_i e e^{-1}b_k, Ab_i b_k)V(Ab_i b_k, Ab_j) \geq \alpha(e)\beta(b_i, b_j)$

Thus,  $Ab_i e e^{-1}b_k \xrightarrow{\alpha(e)\beta(b_i, b_j)} Ab_j$ .

Multiplying affinities when we compose transformations makes intuitive sense because we expect degradation to compound. However, there are other ways to combine affinities, but we do not explore them here. Multiplication is the simplest choice, and works well in our model. (as well as in the Dempster-Shafer belief model [10]).

### 2.3.6 Selecting Affinities

The  $\alpha$ ,  $\beta$ ,  $\gamma$  affinity values for the available transformations can be selected manually by a user or system administrator to capture how desirable images are. For example, the photo editor of a newspaper may provide the values to define what types of transformation sequences are good enough for the paper.

An alternative is for the user or editor to supply a set of “training examples.” For instance, the user can provide two ways to obtain  $Abc$ : from  $Acb$  and from  $Acbb^{-1}c^{-1}bc$ , and state that the former is preferable to the latter. This example then yields the constraint that  $\gamma(c, b)$  should be larger than  $\alpha(b)\alpha(c)$ . The user should give enough constraints so that we can solve for all the affinities. If there is no solution, the user can be warned that his preferences are not consistent.

We do not anticipate the task of selecting affinities to be a big burden for the user. We suspect that many domains are limited to only a few transformations, so the number of possible affinity factors should be small. For example, in the context of BioACT project [2], field biologists typically rely on only a handful number of transformations. In cases where there are many transformations, an automated way via training examples may be used.

## 2.4 Reconstruction Model

Recall that our ultimate goal is to reconstruct some node  $Z$  in a production tree from some accessible node  $Y$ . We call  $Y$  the origin and  $Z$  the destination. Figure 5 illustrates the scenario, where node  $X$  is the root of the tree. Node  $Y$ 's production from the root is  $Xq_Y$  and  $Z$ 's production is  $Xq_Z$ , where  $q_Y$  and  $q_Z$  are the corresponding transformation sequences.

Consequently, our problem is as follows: given an origin's production  $\mathbb{P}_Y = Xq_Y$  and destination's production  $\mathbb{P}_Z = Xq_Z$  where both  $Y$  and  $Z$  belong to the same production tree, we need to find a new transformation sequence  $q_W$  such that  $W = Yq_W = Xq_Yq_W$  is either equal or most similar to  $Z$ .

**Definition 1** A reconstruction is a 3-tuple  $\langle \mathbb{P}_Y, \mathbb{P}_Z, q_W \rangle$ , where  $q_W$  is the reconstruction sequence,  $\mathbb{P}_Y$  is the origin production and  $\mathbb{P}_Z$  is the destination production such that  $\mathbb{P}_Y$  and  $\mathbb{P}_Z$  share the same base (i.e.  $\mathbb{P}_Y = Xq_Y$  and  $\mathbb{P}_Z = Xq_Z$  for some object  $X$  and some transformations sequences  $q_Y$  and  $q_Z$ ). Sequence  $q_W$  can be empty, in which case we denote it by  $\phi$ , i.e.  $\langle \mathbb{P}_Y, \mathbb{P}_Z, \phi \rangle$ .

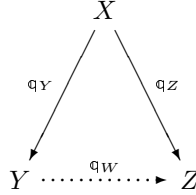


Figure 5: Reconstruction Problem

To illustrate, let us return to the example of Figure 2, where we want to reconstruct  $B_1 = Ab_j$  from  $E = Ab_ie$ . One possible reconstruction, as we showed earlier, is  $\langle Ab_ie, Ab_j, e^{-1}b_k \rangle$ . Another plausible reconstruction, if  $b$  is reversible, is  $\langle Ab_ie, Ab_j, e^{-1}b_i^{-1}b_j \rangle$ . To decide which is better, we can compare the affinity of each end result (i.e.  $\mathbb{P}_W$ ) to the desired destination.

In Section 2.3.5 we illustrated how to compute the affinity of the first reconstruction, i.e., we computed a bound for  $V(Ab_iee^{-1}b_k, Ab_j)$ . However, it turns out that the affinity bound we compute depends on the steps we use in the derivation. For example, consider this second sequence of steps for the same reconstruction  $\langle Ab_ie, Ab_j, e^{-1}b_k \rangle$ :

1.  $Ab_iee^{-1}b_k \xrightarrow{\gamma(e^{-1}, b_k)} Ab_ieb_ke^{-1}$
2.  $Ab_ieb_ke^{-1} \xrightarrow{\gamma(e, b_k)} Ab_ib_k e e^{-1}$
3.  $Ab_ib_k e e^{-1} \xrightarrow{\alpha(e)} Ab_ib_k$
4.  $Ab_ib_k \xrightarrow{\beta(b_i, b_j)} Ab_j$

In this case we apply the commutativity property before the reversibility property and the derivation results in the  $\gamma(e^{-1}, b_k)\gamma(e, b_k)\alpha(e)\beta(b_i, b_j)$  total affinity. This derivation has additional  $\gamma(e^{-1}, b_k)\gamma(e, b_k)$  factors compared to the first derivation we described in Section 2.3.5.

For a given reconstruction, each derivation yields a lower bound on the value of the reconstructed image. Thus, it seems natural that we take the best bound, over all possible derivations, when we consider how good a reconstruction is. This intuition is captured by the following definitions.

**Definition 2** A derivation, denoted via  $\mathbb{D}$  with a possible subscript, of length  $n$  is a sequence of  $n$  steps such that each  $i$ 'th step for  $1 \leq i \leq n$  is of the form  $\mathbb{P}_i \xrightarrow{\pi_i} \mathbb{P}_{i+1}$ , where  $\mathbb{P}_{i+1}$  is produced from  $\mathbb{P}_i$  via a single application of the properties and rules of Table 1.  $\mathbb{P}_1$  is called derivation's beginning and  $\mathbb{P}_{n+1}$  is called derivation's ending.

**Definition 3** Production  $\mathbb{P}$  derives another production  $\mathbb{P}'$ , denoted by  $\mathbb{P} \rightarrow \mathbb{P}'$ , if there is a derivation with beginning  $\mathbb{P}$  and ending  $\mathbb{P}'$ .

**Definition 4** Affinity of the derivation  $\mathbb{D}$  is equal to  $\prod_{i=1}^n \pi_i$ , where  $n$  is  $\mathbb{D}$ 's length and  $\pi_i$  is the affinity resulting from  $\mathbb{D}$ 's  $i$ 'th step for  $1 \leq i \leq n$ .

**Definition 5** A derivation  $\mathbb{D}$  covers a given reconstruction  $\langle \mathbb{P}_Y, \mathbb{P}_Z, \mathbb{Q}_W \rangle$ , if  $\mathbb{P}_Y \mathbb{Q}_W$  is  $\mathbb{D}$ 's beginning and  $\mathbb{P}_Z$  is  $\mathbb{D}$ 's ending.

**Definition 6** A derivation  $\mathbb{D}$  is equivalent to some derivation  $\mathbb{D}'$ , denoted by  $\mathbb{D} \leftrightarrow \mathbb{D}'$ , if  $\mathbb{D}$  and  $\mathbb{D}'$  have the same beginning and ending.

**Definition 7** A derivation  $\mathbb{D}$  is dominated by some equivalent derivation  $\mathbb{D}'$  if affinity of  $\mathbb{D}'$  is greater than the affinity of  $\mathbb{D}$ .

**Definition 8** Affinity of the reconstruction is the maximum affinity of all derivations that cover it. If there are no derivations that cover this reconstruction, its affinity is 0.

## 2.5 Derivation Rules

It is impractical to generate all possible derivations of a reconstruction, thus we need rules to prune “useless” derivations. In this section, we prove two theorems that allow us to drop equivalent derivations that are dominated by others (have a smaller affinity bound), and we introduce two heuristics that allow us to drop equivalent derivations that are *likely* (but not guaranteed) to be dominated by others. For brevity, we refer to derivations in one line as follows:  $\mathbb{P}_1 \xrightarrow{\pi_1} \mathbb{P}_2 \rightarrow \dots \xrightarrow{\pi_n} \mathbb{P}_{n+1}$ .

To motivate our first theorem, consider reconstruction  $\langle Abc, Acb, \phi \rangle$ . Two equivalent derivations that cover this reconstruction are: (1)  $Abc \xrightarrow{\gamma(b,c)} Acb$ ; (2)  $Abc \xrightarrow{\gamma(b,c)} Acb \xrightarrow{\gamma(c,b)} Abc \xrightarrow{\gamma(b,c)} Acb$ . Clearly, the second derivation has a lower affinity. Now, consider another reconstruction  $\langle Ab_i, Ab_jc, b_kc \rangle$ , where  $b$  is a compensable transformation such that  $f_b(i, j) = k$ . Two equivalent derivations for this reconstruction are: (1)  $Ab_ib_kc \xrightarrow{\beta(b_i, b_j)} Ab_jc$ ; (2)  $Ab_ib_kc \xrightarrow{\gamma(b_k, c)} Ab_icb_k \xrightarrow{\gamma(b_i, c)} Acb_ib_k \xrightarrow{\beta(b_i, b_j)} Acb_j \xrightarrow{\gamma(c, b_j)} Ab_jc$ . The second derivation with additional  $\gamma(b_k, c)\gamma(b_i, c)\gamma(c, b_j)$  affinity is clearly worse. We generalize these two cases in the following theorem:

**Theorem 1** Let  $\mathbb{D}$  be a derivation of length  $n$ , such that for some  $k$  and  $l$ ,  $1 \leq k < l \leq n$ , steps  $k$  through  $l$  result in  $X \dots \mathbb{p}_1 \mathbb{q}_1 \dots \rightarrow X \dots \mathbb{q}_2 \mathbb{p}_2 \dots \rightarrow X \dots \mathbb{p}_3 \mathbb{q}_3 \dots$ , where  $\mathbb{p}_1, \mathbb{p}_2, \mathbb{p}_3, \mathbb{q}_1, \mathbb{q}_2, \mathbb{q}_3$  are transformation sequences such that  $\mathbb{p}_1 \rightarrow \mathbb{p}_2 \rightarrow \mathbb{p}_3$  and  $\mathbb{q}_1 \rightarrow \mathbb{q}_2 \rightarrow \mathbb{q}_3$ , and  $\bar{\mathbb{p}}$  is the set of all transformations involved in  $\mathbb{p}_1 \rightarrow \mathbb{p}_3$ , whereas  $\bar{\mathbb{q}}$  is the set of all transformations involved in  $\mathbb{q}_1 \rightarrow \mathbb{q}_3$ . If no  $x \in \bar{\mathbb{p}} \cup \bar{\mathbb{q}}$  interacts with any  $y \notin \bar{\mathbb{p}} \cup \bar{\mathbb{q}}$  between the steps  $k$  and  $l$ , then there exists another derivation  $\mathbb{D}'$  with fewer than  $n$  steps, such that  $\mathbb{D} \leftrightarrow \mathbb{D}'$  and affinity of  $\mathbb{D}'$  is greater than the affinity of  $\mathbb{D}$ .

**Proof** Let the set of steps from  $k$  to  $l$  in  $\mathbb{D}$  be  $S$ . Let's remove steps of the form  $X \dots yz \dots \xrightarrow{\gamma(y,z)} X \dots zy \dots$  or  $X \dots zy \dots \xrightarrow{\gamma(y,z)} X \dots yz \dots$  for any  $y \in \bar{\mathbb{p}}$  and  $z \in \bar{\mathbb{q}}$  from  $S$ . For the remaining steps  $\mathbb{P}_i \rightarrow \mathbb{P}_{i+1}$

in  $S$  ( $k \leq i \leq l$ ), let's reorder transformations in  $\mathbb{P}_i$  and  $\mathbb{P}_{i+1}$  so that all transformations in  $\bar{\rho}$  precede transformations in  $\bar{q}$  while preserving the original order of transformations within  $\bar{\rho}$  and  $\bar{q}$ . For example, we would reorder some production  $X \cdots p_1 q_1 p_2 q_2 \cdots$ , where  $\{p_1, p_2\} \in \bar{\rho}$  and  $\{q_1, q_2\} \in \bar{q}$  as  $X \cdots p_1 p_2 q_1 q_2 \cdots$ . Now since the remaining steps in  $S$  can only be the steps belonging to derivations  $\rho_1 \rightarrow \rho_2 \rightarrow \rho_3$  and  $q_1 \rightarrow q_2 \rightarrow q_3$ , we can obtain a new legal derivation  $\mathbb{D}'$  by substituting steps  $k$  through  $l$  in  $\mathbb{D}$  with steps in  $S$ .  $\mathbb{D}'$  has the same beginning and ending as  $\mathbb{D}$ , and hence, is, by definition, equivalent to  $\mathbb{D}$  but with fewer steps. Since affinity factor of each derivation step in  $\mathbb{D}$  is within  $[0, 1]$ , derivation  $\mathbb{D}'$  having a subset of  $\mathbb{D}$ 's steps, must have a greater total affinity (calculated as a product of affinity factors contributed by each step). ■

Note how we used in the statement of our theorem constraint that no  $x \in \bar{\rho} \cup \bar{q}$  interacts with any  $y \notin \bar{\rho} \cup \bar{q}$  between the steps  $k$  and  $l$ . However, the theorem can be generalized to the case when transformations in  $\bar{\rho}$  and  $\bar{q}$  are initially intermixed with and/or freely commute with transformations not in  $\bar{\rho} \cup \bar{q}$ . However, this case requires another constraint that no transformation  $y \notin \bar{\rho} \cup \bar{q}$  that commutes with some  $x \in \bar{\rho} \cup \bar{q}$  can undergo compensation or reversal between the steps  $k$  and  $l$ . For the sake of clarity, we are not providing a proof for this here but one can easily verify this by observing that if any  $y$  commutes over some subset of transformations in  $\bar{\rho} \cup \bar{q}$ , it does not have to commute over this set in a specific order. Accordingly, we could reorder commutativity steps to first start with transformations in  $\bar{\rho}$  followed by  $\bar{q}$  for left to right movements, and to first start with transformations in  $\bar{q}$  followed by  $\bar{\rho}$  for right to left movements.

To motivate our second theorem, let us return to the example from Figure 2 of reconstruction  $\langle Ab_i e, Ab_j, e^{-1} b_k \rangle$ . Recall that there are two equivalent derivations for this reconstruction: (1)  $Ab_i e e^{-1} b_k \xrightarrow{\alpha(e)} Ab_i b_k \xrightarrow{\beta(b_i, b_j)} Ab_j$ ; (2)  $Ab_i e e^{-1} b_k \xrightarrow{\gamma(e^{-1}, b_k)} Ab_i e b_k e^{-1} \xrightarrow{\gamma(e, b_k)} Ab_i b_k e e^{-1} \xrightarrow{\alpha(e)} Ab_i b_k \xrightarrow{\beta(b_i, b_j)} Ab_j$ . As stated earlier, the second derivation results in additional  $\gamma(e^{-1}, b_k) \gamma(e, b_k)$  affinity and thus has a lower total affinity.

**Theorem 2** *Let  $\mathbb{D}$  be a derivation of length  $n$ , where for some  $k, l$ , and  $m$ , such that  $1 \leq k < l < m \leq n$ , steps  $k, l$  and  $m$  are either of the following two forms:*

1. Step  $k$ :  $X \cdots yz \cdots \xrightarrow{\gamma(y, z)} X \cdots zy \cdots$   
Step  $l$ :  $X \cdots zyz^{-1} \cdots \xrightarrow{\gamma(y, z^{-1})} X \cdots zz^{-1}y \cdots$   
Step  $m$ :  $X \cdots zz^{-1} \cdots \xrightarrow{\alpha(z)} X \cdots$
2. Step  $k$ :  $X \cdots zy \cdots \xrightarrow{\gamma(z, y)} X \cdots yz \cdots$   
Step  $l$ :  $X \cdots z^{-1}yz \cdots \xrightarrow{\gamma(z^{-1}, y)} X \cdots yz^{-1}z \cdots$   
Step  $m$ :  $X \cdots zz^{-1} \cdots \xrightarrow{\alpha(z)} X \cdots$

where  $X$  is some object and  $y, z, z^{-1}$  are the same transformation instances in steps  $k, l$  and  $m$ . If all the steps that involve  $z$  or  $z^{-1}$  between the steps  $k$  and  $m$  are commutativity steps and transformations that commute with  $z$  between steps  $k$  and  $l$  are involved only in commutativity steps between steps  $k$  and  $l$ , then there exists another derivation  $\mathbb{D}'$  with fewer than  $n$  steps, such that  $\mathbb{D} \leftrightarrow \mathbb{D}'$  and affinity of  $\mathbb{D}'$  is greater than the affinity of  $\mathbb{D}^1$ .

**Proof** Let's remove step  $k$  and in all steps  $\mathbb{P}_i \rightarrow \mathbb{P}_{i+1}$  between  $k$  and  $l$  (i.e.  $k < i \leq l$ ), move  $z$  ahead of  $y$  in both  $\mathbb{P}_i$  and  $\mathbb{P}_{i+1}$ . Since all steps that involve  $z$  between steps  $k$  and  $l$  are commutativity steps, any transformation  $x$  that commutes with  $z$  between steps  $k$  and  $l$  must either also commute with  $y$  in the same direction or commute again with  $z$  in the opposite direction. Either of these cases must be true because in step  $l$ , there is no transformation between  $y$  and  $z$  and transformations

<sup>1</sup>By symmetry, theorem holds if we replace  $z$  with  $z^{-1}$  and  $z^{-1}$  with  $z$ , i.e.  $z^{-1}$  commutes over  $y$  before  $z$ .

that commute with  $z$  do not undergo compensation or reversal. Now let's consider both cases. If  $x$  commutes with  $z$  in both directions, the two steps by Theorem 1 are redundant, so we can safely remove them. If on the other hand,  $x$  commutes with both  $z$  and  $y$ , we can simply reorder the steps, e.g. replace steps  $X_{\mathfrak{p}_1} x z y \mathfrak{q}_1 \xrightarrow{\gamma(x,z)} X_{\mathfrak{p}_2} z x y \mathfrak{q}_2 \xrightarrow{\gamma(x,y)} X_{\mathfrak{p}_2} z y x \mathfrak{q}_2$  with  $X_{\mathfrak{p}_1} x y z \mathfrak{q}_1 \xrightarrow{\gamma(x,y)} X_{\mathfrak{p}_2} y x z \mathfrak{q}_2 \xrightarrow{\gamma(x,z)} X_{\mathfrak{p}_2} y z x \mathfrak{q}_2$ .

Now we have a valid derivation up to step  $l$  (using original ordering of the steps). For some transformation sequences  $\mathfrak{p}$  and  $\mathfrak{q}$ , the left side of step  $l$  must be  $X_{\mathfrak{p}} y z z^{-1} \mathfrak{q}$ , if we are in case 1 of the theorem, and  $X_{\mathfrak{p}} z^{-1} z y \mathfrak{q}$ , if we are in case 2 of the theorem. In the first case, we modify step  $l$  as follows:  $X_{\mathfrak{p}} y z z^{-1} \mathfrak{q} \xrightarrow{\alpha(z)} X_{\mathfrak{p}} y \mathfrak{q}$ . On the other hand, in the second case we modify step  $l$  and add a new step after it:  $X_{\mathfrak{p}} y z^{-1} z \mathfrak{q} \xrightarrow{\gamma(z^{-1},z)} X_{\mathfrak{p}} y z z^{-1} \mathfrak{q} \xrightarrow{\alpha(z)} X_{\mathfrak{p}} y \mathfrak{q}$ . Finally, we remove the steps that involve either  $z$  or  $z^{-1}$  from step  $l$  to step  $m$  and remove  $z, z^{-1}$  from productions of all the remaining steps. This, in fact, produces a new legal derivation  $\mathbb{D}'$  with the same origin and destination as  $\mathbb{D}$ .

Note that we have not introduced any additional affinity that  $\mathbb{D}$  did not contain. Reversing  $z$  is equivalent to old step  $m$  that has been removed. Likewise, the step that swaps  $z$  and  $z^{-1}$  must already exist in  $\mathbb{D}$  among the steps  $l$  through  $m$  since step  $m$  has  $z$  and  $z^{-1}$  in the correct order. This step, like the step  $m$ , has been removed in  $\mathbb{D}'$ . Furthermore, note that  $\mathbb{D}'$  has at least two fewer steps than  $\mathbb{D}$ , since we removed steps where  $z$  and  $z^{-1}$  commute over  $y$ . Since affinity factors contributed by each step of derivation is within  $[0, 1]$ ,  $\mathbb{D}'$  having a subset of  $\mathbb{D}$ 's steps, must have a greater total affinity (calculated as a product of affinity factors contributed by each step). ■

Although the two theorems are helpful for pruning some of the equivalent derivations, there can still be more than one feasible derivation for any given reconstruction. For example, consider:  $\langle Ab_i, Acb_j, b_k c \rangle$ . Granted that  $b$  is a compensable transformation such that  $f_b(i, j) = k$ , two equivalent derivations that cover this reconstruction will be: (1)  $Ab_i b_k c \xrightarrow{\beta(b_i, b_j)} Ab_j c \xrightarrow{\gamma(b_j, c)} Acb_j$ ; (2)  $Ab_i b_k c \xrightarrow{\gamma(b_k, c)} Ab_i c b_k \xrightarrow{\gamma(b_i, c)} Acb_i b_k \xrightarrow{\beta(b_i, b_j)} Acb_j$ . None of our theorems rules out either of these derivations. Furthermore, if there are  $n$  compensated transformations like  $b$  that need to commute over  $c$ , there are  $2^n$  equally feasible derivations. If  $n$  is a large number, it may be impractical to consider all of them. Thus, we need *heuristics* to further limit the number of considered derivations.

As the example shows, there is always a choice between applying commutativity in one step and in two steps. Since each step introduces some degradation, the fewer steps seems naturally better. Our first heuristic states it is desirable to commute over a transformation in one step rather than in two steps (in the example above, derivation (1) will be better than (2)). Obviously, there may be exceptions to this rule for some transformations, but we believe this rule holds in most cases.

**Heuristic 1** For any transformation  $z$  and any compensable transformation  $y$ , such that  $f_y(i, j) = k$ ,

1.  $\gamma(y_j, z) \geq \gamma(y_i, z)\gamma(y_k, z)$  and  $\gamma(z, y_j) \geq \gamma(z, y_i)\gamma(z, y_k)$
2.  $\gamma(y_i, z) \geq \gamma(z, y_k)\gamma(y_j, z)$  and  $\gamma(z, y_i) \geq \gamma(y_k, z)\gamma(z, y_j)$
3.  $\gamma(y_k, z) \geq \gamma(z, y_i)\gamma(y_j, z)$  and  $\gamma(z, y_k) \geq \gamma(y_i, z)\gamma(z, y_j)$

To illustrate all three cases of Heuristic 1, consider the following three examples. Just like for the example used above, assume  $b$  is a compensable transformation such that  $f_b(i, j) = k$ :

1. Consider again reconstruction  $\langle Ab_i, Xcb_j, b_k c \rangle$  from above and its two equivalent derivations: (1)  $Ab_i b_k c \xrightarrow{\beta(b_i, b_j)} Ab_j c \xrightarrow{\gamma(b_j, c)} Acb_j$ ; (2)  $Ab_i b_k c \xrightarrow{\gamma(b_k, c)}$

$Ab_ikb_k \xrightarrow{\gamma(b_i,c)} Acb_ib_k \xrightarrow{\beta(b_i,b_j)} Acb_j$ . Using the first case of Heuristic 1, the affinity of the first derivation will always be at least as high as the affinity of the second derivation.

2. Consider reconstruction  $\langle Ab_ic, Acb_j, b_k \rangle$ . Two equivalent derivations that cover this reconstruction are: (1)  $Ab_ikb_k \xrightarrow{\gamma(b_i,c)} Acb_ib_k \xrightarrow{\beta(b_i,b_j)} Acb_j$ ; (2)  $Ab_ikb_k \xrightarrow{\gamma(c,b_k)} Ab_ibkc \xrightarrow{\beta(b_i,b_j)} Ab_jc \xrightarrow{\gamma(b_j,c)} Acb_j$ . Using the second case of Heuristic 1, the affinity of the first derivation will always be at least as high as the affinity of the second derivation.

3. Consider reconstruction  $\langle Ab_ic, Ab_jc, b_k \rangle$ . Two equivalent derivations that cover this reconstruction are: (1)  $Ab_ikb_k \xrightarrow{\gamma(c,b_k)} Ab_ibkc \xrightarrow{\beta(b_i,b_j)} Ab_jc$ ; (2)  $Ab_ikb_k \xrightarrow{\gamma(b_i,c)} Acb_ib_k \xrightarrow{\beta(b_i,b_j)} Acb_j \xrightarrow{\gamma(c,b_j)} Ab_jc$ . Using the third case of Heuristic 1, the affinity of the first derivation will always be at least as high as the affinity of the second derivation.

Our second heuristic helps us calculate affinity when commuting inverse transformations. For example, consider reconstruction  $\langle Abc, Ac, b^{-1} \rangle$ . Two equivalent derivations that cover this reconstruction are: (1)  $Abcb^{-1} \xrightarrow{\gamma(b,c)} Acbb^{-1} \xrightarrow{\alpha(b)} Ac$ ; (2)  $Abcb^{-1} \xrightarrow{\gamma(c,b^{-1})} Abb^{-1}c \xrightarrow{\alpha(b)} Ac$ . Which one results in a higher affinity? Intuitively, commuting the inverse transformation should be the same as commuting the forward transformation. However, it may be impractical to always consider both options, so for simplicity we will arbitrarily give precedence to the forward transformation (first derivation in our example) and define the following heuristic:

**Heuristic 2** For any transformation  $z$  and reversible transformation  $y$ ,  $\gamma(y, z) \geq \gamma(z, y^{-1})$  and  $\gamma(z, y) \geq \gamma(y^{-1}, z)$ .

### 3 Algorithms

In this section we present algorithms for finding reconstructions. We are not considering the problem of finding the best origin node for a given destination node but rather focusing on the problem of finding the best reconstruction sequence given a fixed origin and destination nodes. Each algorithm takes as input an origin production  $\mathbb{P}_O$  and a destination production  $\mathbb{P}_D$  with a common base and returns a pair  $\langle \mathfrak{q}, \pi \rangle$ , such that  $\langle \mathbb{P}_O, \mathbb{P}_D, \mathfrak{q} \rangle$  is a reconstruction with affinity of at least  $\pi$ . Using our example from Figure 2, if  $\mathbb{P}_O = Ab_ie$  (production of  $E$ ) and  $\mathbb{P}_D = Ab_j$  (production of  $B_1$ ),  $\mathfrak{q}$  returned by an algorithm could be  $e^{-1}b_i^{-1}b_j$  such that the resulting reconstruction is  $\langle Ab_ie, Ab_j, e^{-1}b_i^{-1}b_j \rangle$ .

We focus on the best reconstruction problem because its solution is key to solving many other problems. For instance, say we want to find the best origin node for a given destination node. One way to accomplish this would be to find the best reconstruction for each possible origin node and pick the one whose reconstruction has the highest affinity. Even if one does not want to evaluate each origin node by explicitly finding its best reconstruction, one would still wind up comparing production sequences of each node and, hence, apply the same rules and heuristics as we apply to the fixed origin reconstruction problem. As a second example, say we want to find the smallest set of nodes to produce all the other nodes in a production tree within some affinity threshold. Again, a solution to the best reconstruction problem would come to rescue. Specifically, for each node in a tree we could calculate its best reconstruction of all other nodes. After this, we could greedily select the nodes that reconstruct (within the affinity threshold) the most nodes until we are able to reconstruct the entire tree.

For simplicity, we assume that there is only one instance of each transformation in any production. This assumption is not essential to our algorithms but makes discussion clearer. Nonetheless,

even in this simplified model, as we will see, the problem of finding the best reconstruction is very hard.

We name our algorithms after transformers “Rewind”, “Fortress Maximus”, and “Optimus Prime” featured in the original “Transformers” cartoon [13]. Algorithm Rewind is a simple algorithm that works well in simple cases; the exponential-time Optimus algorithm finds an optimal solution by exhaustive search; and the polynomial-time Maximus algorithm strikes a balance between performance and the value of the reconstructed image.

### 3.1 Notation

Let us first discuss the data structures the algorithms will operate on and introduce notation we are going to use when presenting algorithms. In particular, we will view a transformation sequence as a doubly linked list that contains *head* and *tail* pointers and each node contains a transformation. So for example, for sequence  $q = b_i e$ ,  $q.head = b_i$ ,  $q.tail = e$ ,  $q.head.next = e$ , and  $q.tail.previous = b_i$ . Although a node in a transformation sequence does not exactly equal to a transformation it contains, we assume this is true for notational simplicity. In addition, a production will be viewed as a data structure containing a *base* object and a transformation sequence referred to as *seq*. For example, for production  $\mathbb{P} = Ab_i e$ ,  $\mathbb{P}.base = A$  and  $\mathbb{P}.seq = b_i e$ .

We will often need to iterate over transformations in a sequence, so we introduce notation for this. For example, to iterate over some  $q$  from head to tail, we will write the following:

**for each**  $t \in q$

If we need to iterate over  $q$  from tail to head, we will use the keyword *backward*:

**for each**  $t \in q$  **backward**

In some instances, instead of iterating over all transformations in a sequence, we may need to iterate over only a subset of them from, say, transformation node  $t1$  to some transformation node  $t2$  (by convention, iteration would include  $t1$  but not  $t2$ ). Accordingly, we are going to use the following notation to express this (which can be combined with *backward* for the backward iterations):

**for each**  $t \in q$  **from**  $t1$  **to**  $t2$

If we need to iterate from  $t1$  to the tail of the sequence, we omit the “to” part:

**for each**  $t \in q$  **from**  $t1$

Similarly, if we need to iterate from the head of the sequence to  $t2$ , we omit the “from” part:

**for each**  $t \in q$  **to**  $t2$

Finally, when iterating over some production  $\mathbb{P}$ ’s sequence, instead of saying  $\mathbb{P}.seq$ , we write  $\mathbb{P}$  to imply  $\mathbb{P}.seq$ :

**for each**  $t \in \mathbb{P}$

To avoid unnecessary complexity when searching through and managing transformation sequences, we use the basic functions described in Table 2.

### 3.2 Rewind Algorithm

The Rewind Algorithm (shown in Listing 1) illustrates the most naive reconstruction strategy. It reverses all transformations in  $\mathbb{P}_O$  and reproduces all transformations in  $\mathbb{P}_D$ . To see the algorithm in action, consider our favorite example from Figure 2, where  $\mathbb{P}_O = Ab_i e$  and  $\mathbb{P}_D = Ab_j$ . After initializing our result sequence  $q$  and reconstruction affinity  $\pi$ , in lines 4-6 of the algorithm, we

Table 2: Utility functions used by the algorithms

| Function Signature          | Description  |
|-----------------------------|--|
| initializeSeq()             | Initializes and returns an empty transformation sequence   |
| add( $x$ , $q$ )            | Adds $x$ to the tail of $q$  |
| find( $x$ , $q$ )           | Searches $q$ for $x$ , returns the corresponding node if it is found or <i>null</i> otherwise          |
| remove( $x$ , $q$ )         | Removes given transformation $x$ from $q$ if such is found   |
| isBefore( $q$ , $x$ , $y$ ) | Returns true if there is a node containing $x$ before $y$ in $q$ (granted $y \in q$ ), false otherwise |

iterate over all transformations in  $\mathbb{P}_O$  from tail to head, add their inverses to  $q$ , and penalize each one with its  $\alpha$  affinity factor. So in our example,  $q$  becomes  $e^{-1}b_i^{-1}$  and  $\pi$  becomes  $\alpha(e)\alpha(b_i)$ . Having reversed all transformations in the origin production, in lines 7-8 of the algorithm we now add all transformations in  $\mathbb{P}_D$  to  $q$  in the forward order (head to tail). In our example, we just add  $b_j$ , the only transformation in  $\mathbb{P}_D$ , to  $q$  and return the result:  $\langle e^{-1}b_i^{-1}b_j, \alpha(e)\alpha(b_i) \rangle$ . The affinity we return corresponds to the affinity of the following derivation:  $Ab_i e e^{-1}b_i^{-1}b_j \xrightarrow{\alpha(e)} Ab_i b_i^{-1}b_j \xrightarrow{\alpha(b_i)} Ab_j$ .

Listing 1: Rewind Algorithm

```

1 Rewind( $\mathbb{P}_O$ ,  $\mathbb{P}_D$ )
2    $q := \text{initializeSeq}()$ 
3    $\pi := 1$ 
4   for each  $x \in \mathbb{P}_O$  backward
5     add( $x^{-1}$ ,  $q$ )
6      $\pi := \pi * \alpha(x)$ 
7   for each  $x \in \mathbb{P}_D$ 
8     add( $x$ ,  $q$ )
9   return  $\langle q, \pi \rangle$ 

```

### 3.3 Maximus Algorithm

Rewind may not always yield the best results. For example, consider  $\mathbb{P}_O = Abcd$  and  $\mathbb{P}_D = Abce$ . It is obvious that we do not want to reverse all  $b, c, d$  but just  $d$ , followed by the reproduction of  $e$ . Hence, a better reconstruction sequence is  $q = d^{-1}e$  with the total affinity of  $\pi = \alpha(d)$  as opposed to  $q' = d^{-1}c^{-1}b^{-1}bce$  (returned by the Rewind) with the affinity of  $\pi' = \alpha(d)\alpha(c)\alpha(b)$ .

On the other hand, consider the same example but with the common part between  $\mathbb{P}_O$  and  $\mathbb{P}_D$  in a different order from each other. Specifically, consider  $\mathbb{P}_O = Acbd$  and  $\mathbb{P}_D = Abce$ . Sequence  $q$  may still be better than  $q'$ , but  $\pi$  needs to include now a new factor  $\gamma(c, b)$  to account  $b$  and  $c$  being out of order. Consequently,  $\pi = \alpha(d)\gamma(c, b)$ , and  $q$  will be better than  $q'$  if and only if  $\gamma(c, b) > \alpha(b)\alpha(c)$ .

It is less obvious whether “preserving” or reversing common transformations is advantageous when the “uncommon” transformations intermix with the common ones and vice versa. For example, consider  $\mathbb{P}_O = Adbc$  and  $\mathbb{P}_D = Abce$ . Now  $\mathbb{P}_O$  has the common transformations  $b$  and  $c$  preceded by the “uncommon” transformations  $d$ . Accordingly, if we decide to keep  $b$  and  $c$  in the production (i.e. not reverse them), then when applying  $d^{-1}$ , we need to be careful to account for the additional affinity:  $Adbc d^{-1}e \xrightarrow{\gamma(d,b)} Abcd d^{-1}e \xrightarrow{\gamma(d,c)} Abcd d^{-1}e \xrightarrow{\alpha(d)} Abce$ . Although by keeping  $b$  and  $c$  in the origin production we are avoiding the price of  $\alpha(b)\alpha(c)$ , there is now an

additional penalty of  $\gamma(d, b)\gamma(d, c)$ . Again, depending on whether  $\gamma(d, b)\gamma(d, c)$  or  $\alpha(b)\alpha(c)$  yields higher affinity, we may choose to reverse or preserve  $b$  and  $c$ .

Table 3: Transformation sequences in  $\mathbb{P}_O$  and  $\mathbb{P}_D$  in relation to some common transformation  $x$

| Seq               | Definition   | Effect on $\pi_p$ and $\pi_r$ of $x$  |
|-------------------|--|---|
| $\circ_l$         | Transformations unique to $\mathbb{P}_O$ to the left of $x$                                    | We know for certain that these transformations will be reversed since they do not belong to $\mathbb{P}_D$ . If $x$ is reversed, there is no additional affinity. If $x$ is preserved, additional $\gamma(\circ_l, x)$ will be necessary to reverse $\circ_l$ , thus $\gamma(\circ_l, x)$ is included in $\pi_p$ .  |
| $\circ_r$         | Transformations unique to $\mathbb{P}_O$ to the right of $x$                                   | These transformations should be reversed by the time we get to $x$ (they do not belong to $\mathbb{P}_D$ ), so we can ignore them.  |
| $\circ_l$         | Transformations unique to $\mathbb{P}_D$ to the left of $x$                                    | These transformations have to be placed before $x$ . If $x$ is reversed, no additional affinity is required. If $x$ is preserved, we have to account for $\gamma(x, \circ_l)$ , hence the latter is included in $\pi_p$ .   |
| $\circ_r$         | Transformations unique to $\mathbb{P}_D$ to the right of $x$                                   | Regardless of whether we reverse or preserve $x$ , these transformations can be applied after $x$ , thus there is no additional affinity in either case.  |
| $\mathbb{C}_{lr}$ | Transformations to the left of $x$ in $\mathbb{P}_O$ but to the right of $x$ in $\mathbb{P}_D$ | We do not know yet whether we are going to preserve or reverse these transformations (assuming traversal of $\mathbb{P}_D$ from right to left). If $x$ is preserved, regardless of whether transformations in $\mathbb{C}_{lr}$ are preserved or reversed, we pay the penalty of $\gamma(\mathbb{C}_{lr}, x)$ . However, if $x$ is reversed, we incur $\gamma(\mathbb{C}_{lr}, x)$ only if $\mathbb{C}_{lr}$ transformations are preserved. Assuming the best case scenario for both, we include $\gamma(\mathbb{C}_{lr}, x)$ in $\pi_p$ but not in $\pi_r$ . |
| $\mathbb{C}_{ll}$ | Transformations to the left of $x$ in both $\mathbb{P}_O$ and $\mathbb{P}_D$                   | We do not know yet whether we are going to preserve or reverse these transformations. If transformations in $\mathbb{C}_{ll}$ are preserved, there is no additional commutativity if we preserve or reverse $x$ . If transformations in $\mathbb{C}_{ll}$ are reversed, preserving $x$ will result in additional $\gamma(\mathbb{C}_{ll}, x)\gamma(x, \mathbb{C}_{ll})$ affinity. Assuming best case for both $\pi_p$ and $\pi_r$ , we do not penalize either.  |
| $\mathbb{C}_{rl}$ | Transformations to the right $x$ in $\mathbb{P}_O$ but to the left of $x$ in $\mathbb{P}_D$    | Some of the transformations in $\mathbb{C}_{rl}$ have been reversed but others have been preserved. While $\gamma(x, \mathbb{C}_{rl})$ for the preserved transformations has already been accounted (see $\mathbb{C}_{lr}$ case), $\gamma(x, \mathbb{C}_{rl})$ for the reversed transformations is yet to be included if $x$ is preserved. Accordingly, we are going to include $\gamma(x, \mathbb{C}_{rl}^r)$ in $\pi_p$ where $\mathbb{C}_{rl}^r$ denotes reversed transformations in $\mathbb{C}_{rl}$ .   |
| $\mathbb{C}_{rr}$ | Transformations to the right of $x$ in both $\mathbb{P}_O$ and $\mathbb{P}_D$                  | Although some of the transformations in $\mathbb{C}_{rr}$ have been reversed, we only need to consider the ones that have been preserved. If $x$ is preserved, there is no additional affinity, while if $x$ is reversed, it will result in extra $\gamma(x, \mathbb{C}_{rr})\gamma(\mathbb{C}_{rr}, x)$ affinity. Accordingly, we need to penalize $\pi_r$ with $\gamma(x, \mathbb{C}_{rr}^p)\gamma(\mathbb{C}_{rr}^p, x)$ , where $\mathbb{C}_{rr}^p$ denotes preserved transformations in $\mathbb{C}_{rr}$ .  |

Intuitively, it seems that the decision to preserve or reverse a common transformation should be made quantitatively on an individual basis. The strategy we could use is traversing the origin production from right to left and for every observed common transformations  $x$  deciding whether to reverse it or preserve it. To make this decision, we could compare the affinity of preserving  $x$ , which we are going to denote  $\pi_p$ , to the affinity of reversing  $x$ , which we are going to denote  $\pi_r$ .

In order to estimate  $\pi_p$  and  $\pi_r$ , we are going to classify all transformations in  $\mathbb{P}_O$  and  $\mathbb{P}_D$  in terms of their placement with respect to  $x$ . As shown in Table 3, we are going to treat transformations differently depending on whether they are unique to  $\mathbb{P}_O$  or  $\mathbb{P}_D$ , and depending on whether they are to the left of or to the right of  $x$  in each production. Using this notation, we could describe  $\mathbb{P}_O$  and  $\mathbb{P}_D$  as follows:  $\mathbb{P}_O = \overline{\mathbb{C}_{lr}\mathbb{C}_{ll}\circ_l x \mathbb{C}_{rl}\mathbb{C}_{rr}\circ_r}$  and  $\mathbb{P}_D = \overline{\mathbb{C}_{ll}\mathbb{C}_{rl}\circ_l x \mathbb{C}_{lr}\mathbb{C}_{rr}\circ_r}$ . Note that we are using a bar on top of the transformation sequences to denote that transformations from each sequence can be interleaved, thus a group of transformation sequences should be viewed as a bag rather

than a sequence of transformations.

Now that we have classified transformations in  $\mathbb{P}_O$  and  $\mathbb{P}_D$ , we are going to define  $\pi_p$  and  $\pi_r$  each as affinity of a set of derivation steps. As explained in Table 3,  $\pi_p$  will be defined as affinity of these four steps:

1.  $\overline{\mathfrak{c}_{lr}\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{lr}\mathfrak{C}_{rr}\mathfrak{d}_l}} \xrightarrow{\gamma(\mathfrak{c}_{lr},x)} \overline{\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{lr}\mathfrak{C}_{rr}\mathfrak{d}_l}}$
2.  $\overline{\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{lr}\mathfrak{C}_{rr}\mathfrak{d}_l}} \xrightarrow{\gamma(x,\mathfrak{e}_{rl}^r)} \overline{\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{lr}\mathfrak{C}_{rr}\mathfrak{d}_l}}$
3.  $\overline{\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{lr}\mathfrak{C}_{rr}\mathfrak{d}_l}} \xrightarrow{\gamma(\mathfrak{O}_l,x)} \overline{\mathfrak{c}_{ll}\mathfrak{C}_{rl}^p\overline{x\mathfrak{O}_l\mathfrak{C}_{lr}\mathfrak{C}_{rr}\mathfrak{d}_l}}$
4.  $\overline{\mathfrak{c}_{ll}\mathfrak{C}_{rl}^p\overline{x\mathfrak{O}_l\mathfrak{C}_{lr}\mathfrak{C}_{rr}\mathfrak{d}_l}} \xrightarrow{\gamma(x,\mathfrak{d}_l)} \overline{\mathfrak{c}_{ll}\mathfrak{C}_{rl}^p\overline{\mathfrak{d}_l x\mathfrak{O}_l\mathfrak{C}_{lr}\mathfrak{C}_{rr}}}$

On the other hand,  $\pi_r$  will be affinity of the following three steps:

1.  $\overline{\mathfrak{c}_{lr}\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{rr}^p x^{-1}x}} \xrightarrow{\gamma(x,\mathfrak{e}_{rr}^p)} \overline{\mathfrak{c}_{lr}\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{rr}^p x x^{-1}x}}$
2.  $\overline{\mathfrak{c}_{lr}\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{rr}^p x x^{-1}x}} \xrightarrow{\alpha(x)} \overline{\mathfrak{c}_{lr}\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{rr}^p x}}$
3.  $\overline{\mathfrak{c}_{lr}\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{rr}^p x}} \xrightarrow{\gamma(\mathfrak{e}_{rr}^p;x)} \overline{\mathfrak{c}_{lr}\mathfrak{c}_{ll}\mathfrak{O}_l\mathfrak{C}_{rl}^p\overline{x\mathfrak{C}_{rr}^p x\mathfrak{C}_{rr}^p}}.$

As the reader may have noticed, both  $\pi_p$  and  $\pi_r$  of a common transformation, depend on whether we reverse or preserve common transformations to the left of it in  $\mathbb{P}_O$ . More specifically, there are two classes of such transformations:  $\mathfrak{c}_{lr}$  and  $\mathfrak{c}_{ll}$ . Both  $\mathfrak{c}_{lr}$  and  $\mathfrak{c}_{ll}$  have not yet been traversed (assuming right to left traversal order), so we do not know yet whether they are preserved or reversed. Consequently, we need to make assumptions. As described in Table 3, we are optimistic and assume the best case scenario for both  $\pi_p$  and  $\pi_r$ , which is preservation of  $\mathfrak{c}_{lr}$  and  $\mathfrak{c}_{ll}$  for  $\pi_p$  and reversal of  $\mathfrak{c}_{lr}$  and  $\mathfrak{c}_{ll}$  for  $\pi_r$ . We are effectively trying to estimate best case for reversing  $x$  and best case for preserving  $x$  and make a greedy choice in favor of the better one.

Listing 2: Maximus Algorithm

```

1 Maximus( $\mathbb{P}_O$ ,  $\mathbb{P}_D$ )
2    $q := \text{initializeSeq}()$ 
3    $\pi := 1$ 
4   for each  $x \in \mathbb{P}_O$  backward
5      $\pi_r := \alpha(x)$ 
6     if ( $\text{find}(x, \mathbb{P}_D.\text{seq}) \neq \text{null}$ )
7        $\pi_p := 1$ 
8       for each  $y \in \mathbb{P}_O$  to  $x$ 
9         if ( $\text{!isBefore}(\mathbb{P}_D.\text{seq}, y, x)$ )
10           $\pi_p := \pi_p * \gamma(y, x)$ 
11        for each  $y \in \mathbb{P}_D$  to  $x$ 
12          if ( $\text{find}(y, \mathbb{P}_O.\text{seq}) = \text{null}$ )
13             $\pi_p := \pi_p * \gamma(x, y)$ 
14          for each  $y \in \mathbb{P}_O$  from  $x.\text{next}$ 
15            if ( $\text{isBefore}(\mathbb{P}_D.\text{seq}, x, y)$ )
16               $\pi_r := \pi_r * \gamma(x, y) * \gamma(y, x)$ 
17          if ( $\pi_p > \pi_r$ )
18             $\pi := \pi * \pi_p$ 
19          continue
20         $\pi := \pi * \pi_r$ 
21      add( $x^{-1}$ ,  $q$ )
22      remove( $x$ ,  $\mathbb{P}_O.\text{seq}$ )

```

```

23   for each  $x \in \mathbb{P}_D$ 
24     if ( $\text{find}(\mathbb{P}_O, x) = \text{null}$ )
25        $\text{add}(x, \mathfrak{q})$ 
26   return  $\langle \mathfrak{q}, \pi \rangle$ 

```

Our reasoning above demonstrates the intuition behind the Maximus Algorithm. Shown in Listing 2, the algorithm consists of two main loops in lines 4-22 and lines 23-25, respectively. In the first loop, the algorithm iterates over  $\mathbb{P}_O$  from right to left and for each transformation  $x$  in  $\mathbb{P}_O$  checks if affinity  $\pi_p$  of preserving  $x$  is greater than the affinity  $\pi_r$  of reversing  $x$  ( $\pi_p$  is calculated only if  $x$  is common between  $\mathbb{P}_O$  and  $\mathbb{P}_D$ ; if  $x$  is unique to  $\mathbb{P}_O$ , we always choose the option of reversing). Every transformation that gets reversed is removed from  $\mathbb{P}_O$ , so at the termination of the first loop,  $\mathbb{P}_O$  contains only preserved transformations. Then, in the second loop, the algorithm iterates over  $\mathbb{P}_D$  and applies only those transformations in  $\mathbb{P}_D$  that are not yet in  $\mathbb{P}_O$ , i.e. transformations unique to  $\mathbb{P}_D$  and transformations common with  $\mathbb{P}_O$  but not preserved.

Although the strategy used by the algorithm is intuitively reasonable and effective, it may not always produce a reconstruction with the highest affinity. Using the example of  $\mathbb{P}_O = Adbc$  and  $\mathbb{P}_D = Abce$  from before, suppose  $\alpha(c)\alpha(b) > \gamma(d, c)\gamma(d, b)$ , but  $\alpha(c) < \gamma(d, c)$ . From the first condition, we have  $\alpha(c)\alpha(b)\alpha(d) > \gamma(d, c)\gamma(d, b)\alpha(d)$  and, hence, reconstruction  $\langle Adbc, Abce, c^{-1}b^{-1}d^{-1}bce \rangle$  has a higher affinity than reconstruction  $\langle Adbc, Abce, d^{-1}e \rangle$ . However, because  $\alpha(c) < \gamma(d, c)$ , our algorithm when considering whether to reverse or preserve  $c$  will choose to preserve  $c$  (since it assumes that  $b$  is preserved when estimating  $\pi_p$ ) and may ultimately generate a suboptimal reconstruction  $\langle Adbc, Abce, d^{-1}e \rangle$  (granted  $\gamma(d, b) > \alpha(b)\gamma(b, c)\gamma(c, b)$ ). Clearly, the optimistic assumptions we make when estimating  $\pi_r$  and  $\pi_p$ , although allowing us to make a decision of reversing or preserving a transformation without generating all possible reconstructions, may, in general, lead to a suboptimal solution. Nonetheless, the algorithm by exploiting common transformations between the two productions should on average outperform the naive Rewind strategy.

### 3.3.1 Including compensation

As the reader might have noticed, throughout the discussion of Maximus algorithm, we have completely ignored compensation. Specifically, if the origin production contained  $a_1$  and the destination production contained  $a_2$ , we did not consider compensating  $a_1$  to  $a_2$  and, instead, treated the two  $a$  invocations as two distinct transformations (hence, we always reversed  $a_1$  and reproduced  $a_2$ ). In this section, we will show how Maximus algorithm can be extended to handle compensation.

We are going to make a few changes to our notation. First, we assume that the utility function  $\text{find}(x_i, \mathfrak{q})$  searches the given sequence  $\mathfrak{q}$  for  $x$  invocations independently of the parameter  $i$ , so that if  $\mathfrak{q}$  contains  $x_j$  for some  $j \neq i$ ,  $\text{find}(x_i, \mathfrak{q})$  returns  $x_j$ . Secondly, for simplicity, we say that  $\beta(x_i, x_i) = 1$  and  $\gamma(x_i, y_\phi) = 1$ , where  $x$  and  $y$  are transformations,  $i$  is  $x$ 's parameter, and  $\phi$  is a null parameter such that  $y$  applied with  $\phi$  has no effect on a base object. Finally, whenever we refer to a transformation, we include its parameter, e.g.  $x_i$  instead of  $x$ .

Now for each transformation  $x_i$  in  $\mathbb{P}_D$ , we calculate  $\pi_p$  and  $\pi_r$  using the same reasoning as provided in Table 3 but paying attention this time to the transformation parameters. Every time we refer to a transformation in  $\mathbb{P}_O$ , we assume it has a parameter  $i$ ; every time we refer to a transformation in  $\mathbb{P}_D$ , we assume it has a parameter  $j$ . The new algorithm is shown in Listing 3. In addition to careful handling of transformation parameters and including  $\beta(x_i, x_j)$  in  $\pi_p$ , the only real change applies to the case of  $\mathfrak{c}u$ . We include in  $\pi_p$  now the extra  $\gamma(x_i, y_k)$  commutativity affinity for any transformation  $y$  to the left of  $x$  in both  $\mathbb{P}_O$  and  $\mathbb{P}_D$  where  $y$  is applied with parameter  $i$  in  $\mathbb{P}_O$  and parameter  $j$  in  $\mathbb{P}_D$  and  $k = f_y(i, j)$ . The rest of the changes are straight-forward, so we omit the details here.

Listing 3: Maximus Algorithm with Compensation

```

1 Maximus( $\mathbb{P}_O$ ,  $\mathbb{P}_D$ )
2    $q := \text{initializeSeq}()$ 
3    $\pi := 1$ 
4   for each  $x_i \in \mathbb{P}_O$  backward
5      $\pi_r := \alpha(x_i)$ 
6      $x_j = \text{find}(x_i, \mathbb{P}_D.\text{seq})$ 
7     if ( $x_j \neq \text{null}$ )
8        $\pi_p := \beta(x_i, x_j)$ 
9       for each  $y_i \in \mathbb{P}_O$  to  $x_i$ 
10         $y_j := \text{find}(y_i, \mathbb{P}_D.\text{seq})$ 
11        if ( $y_j = \text{null}$ )
12           $\pi_p := \pi_p * \gamma(y_i, x_i)$ 
13        else
14          if (!isBefore( $\mathbb{P}_D.\text{seq}, y_j, x_j$ ))
15             $\pi_p := \pi_p * \gamma(y_i, x_j)$ 
16        for each  $y_j \in \mathbb{P}_D$  to  $x_j$ 
17           $y_i := \text{find}(y_j, \mathbb{P}_O.\text{seq})$ 
18          if ( $y_i = \text{null}$ )
19             $\pi_p := \pi_p * \gamma(x_i, y_j)$ 
20          if (isBefore( $\mathbb{P}_O.\text{seq}, y_i, x_i$ ))
21             $k := f_y(i, j)$ 
22             $\pi_p := \pi_p * \gamma(x_i, y_k)$ 
23          for each  $y_i \in \mathbb{P}_O$  from  $x_i.\text{next}$ 
24             $y_j = \text{find}(y_i, \mathbb{P}_D.\text{seq})$ 
25            if ( $y_j \neq \text{null}$  and isBefore( $\mathbb{P}_D.\text{seq}, x_j, y_j$ ))
26               $\pi_r := \pi_r * \gamma(x_i, y_i) * \gamma(y_i, x_j)$ 
27          if ( $\pi_p > \pi_r$ )
28             $\pi := \pi * \pi_p$ 
29          continue
30         $\pi := \pi * \pi_r$ 
31        add( $x^{-1}$ ,  $q$ )
32        remove( $x$ ,  $\mathbb{P}_O.\text{seq}$ )
33      for each  $x \in \mathbb{P}_D$ 
34        if (find( $\mathbb{P}_O, x$ )= $\text{null}$ )
35          add( $x$ ,  $q$ )
36      return  $\langle q, \pi \rangle$ 

```

### 3.4 Optimus Algorithm

The implementation of Optimus Algorithm is very straight-forward, so we do not present the code for it here. In summary, the algorithm generates all possible reconstruction sequences (for a total of  $2^n$  sequences, where  $n$  is the number of common transformations) by either reversing or preserving each common transformation between  $\mathbb{P}_O$  and  $\mathbb{P}_D$ , reversing transformations unique to  $\mathbb{P}_O$ , and reproducing transformations unique to  $\mathbb{P}_D$ . Then for all generated reconstruction sequences, the algorithm calculates associated affinity using the theorems and heuristics defined in Section 2.5 (inverses are applied first followed by reproductions and compensations) and returns reconstruction sequence with the best affinity.

## 4 Representative Scenarios

In this section we illustrate how our algorithms perform in some representative scenarios. First of all, note that the shape of a production tree has no effect on the algorithms' performance since the algorithms operate on the transformation sequences rather than on the tree itself. Now given an origin and destination transformation sequences, what are the possible options? There are really three options: (1) an origin sequence is a subset of the destination sequence, (2) a destination sequence is a subset of the origin sequence, or (3) both origin and destination sequences share transformations but neither is a subset of the other.

The first and the second cases are symmetrical. To see why, consider  $\mathfrak{q}_O^1 = bd$  and  $\mathfrak{q}_D^1 = bcd$  ( $\mathfrak{q}_O^1 \subset \mathfrak{q}_D^1$ ), and  $\mathfrak{q}_O^2 = bcd$  and  $\mathfrak{q}_D^2 = bd$  ( $\mathfrak{q}_D^2 \subset \mathfrak{q}_O^2$ ). Regardless of whether  $c$  is missing as shown in the first case or extra as shown in the second case, if we do not reverse  $d$ , we need to swap  $c$  and  $d$  (in the first case to apply  $c$ , in the second case to reverse  $c$ ). This illustrates how transformations in  $\mathfrak{q}_D - \mathfrak{q}_O$  and  $\mathfrak{q}_O - \mathfrak{q}_D$  have the same effect on the reconstruction strategy. Thus, we do not need to study the second scenario: the results are analogous to the ones for the first scenario. The third case, on the other hand, is the combination of the first two cases, so the insights one learns from the third case are simply extensions of the conclusions for the first case. Thus, for brevity, in this paper we only study the first case, i.e. when an origin sequence is a subset of the destination sequence.

The first set of experiments (described in Section 4.1) shows the difference in strategy selection used by each algorithm, while the second set of experiments (described in Section 4.2) evaluates performance of algorithms in terms of achieved affinity. Although affinity values by themselves are not significant, we expect that small differences in affinity values imply small differences in quality. Thus, we would like to have algorithms that find reconstructions with affinities that are very close to the affinity of the optimal reconstruction. Accordingly, in Section 4.2 we evaluate our algorithms by seeing how closely they achieve optimal affinity, and in how many instances they actually get the optimal affinity (hence, the best reconstruction). We have conducted other experiments beyond the two sets we report on here, but what we have learned is similar to our conclusions for these two sets.

### 4.1 Strategy Selection

Consider the production tree in Figure 6. In this scenario, two similar but slightly different transformation sequences have been applied to the base image  $A$ . For example,  $B_1$  could be version of  $A$  for web and  $R_2$  be version of  $A$  for printing. Suppose at some point  $R_2$  is lost, the original  $A$  is no longer available, and we only have  $B_1$  left. Now we want to reconstruct  $R_2$  from  $B_1$ .

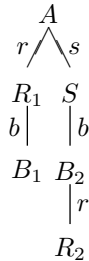


Figure 6: Production Tree

In this case, the origin production is  $\mathbb{P}_O = Arb$  and the destination production is  $\mathbb{P}_D = Asbr$ , resulting in four possible reconstruction sequences of interest:

1.  $q = b^{-1}r^{-1}sbr$  with derivation  $Arbb^{-1}r^{-1}sbr \xrightarrow{\alpha(b)}$   
 $Arr^{-1}sbr \xrightarrow{\alpha(r)} Asbr$
2.  $q = s$  with derivation  $Arbs \xrightarrow{\gamma(b,s)} Arsb \xrightarrow{\gamma(r,s)} Asrb \xrightarrow{\gamma(r,b)} Asbr$
3.  $q = b^{-1}sb$  with derivation  $Arbb^{-1}sb \xrightarrow{\alpha(b)} Arsb \xrightarrow{\gamma(r,s)} Asrb \xrightarrow{\gamma(r,b)} Asbr$
4.  $q = r^{-1}sr$  with derivation  $Arbr^{-1}sr \xrightarrow{\gamma(r,b)} Abr r^{-1}sr \xrightarrow{\alpha(r)} Absr \xrightarrow{\gamma(b,s)} Asbr$

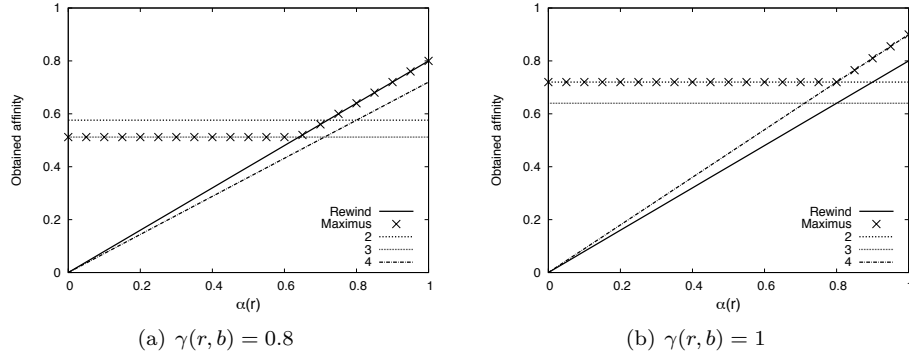


Figure 7: Reconstruction strategies for  $\mathbb{P}_O = Arb$  and  $\mathbb{P}_D = Asbr$  and different  $\gamma(r, b)$  as we vary  $\alpha(r)$  and fix  $\alpha(b) = 0.8, \gamma(r, s) = 0.8, \gamma(b, s) = 0.9$

The Rewind Algorithm always yields the first reconstruction sequence. The Maximus Algorithm selects its reconstruction based on the affinity values. To illustrate how Maximus makes its choice, we set  $\alpha(b) = 0.8, \gamma(r, s) = 0.8, \gamma(b, s) = 0.9$  and vary  $\gamma(r, b)$  and  $\alpha(r)$ . (We have experimented with many other parameter values, but we only show results for these values.) Figure 7(a) shows the  $\gamma(r, b) = 0.8$  case, whereas Figure 7(b) shows  $\gamma(r, b) = 1$ . In each figure we plot the resulting affinity for the four strategies above as a function of  $\alpha(r)$ . The strategy selected by Maximus is shown by the X marks. For example, affinity obtained by Strategy 2 is independent (constant at 0.576 in Figure 7(a) and 0.72 in Figure 7(b)) of  $\alpha(r)$ , because Strategy 2 does not reverse  $r$ . One can see how Maximus in Figure 7(a) selects Strategy 3 for  $\alpha(r) \in [0, 0.64]$  and strategy 1 for  $\alpha(r) \in (0.64, 1]$ , while in Figure 7(b) it selects Strategy 2 for  $\alpha(r) \in [0, 0.8]$  and Strategy 4 for  $\alpha(r) \in (0.8, 1]$ .

The reconstruction sequence obtained by Maximus is suboptimal in Figure 7(a) for values of  $\alpha(r) \in [0, 0.72]$  because Strategy 2 yields a higher affinity value. However, Maximus is always optimal in Figure 7(b). This scenario demonstrates the “dynamic” nature of the Maximus algorithm. It does not depend on just one affinity parameter, but rather dynamically selects what it thinks is the best strategy given the affinity factors.

## 4.2 Performance of Algorithms

In the second set of experiments, we evaluate in which cases and for which affinity values our algorithms yield an optimal reconstruction sequence. We assume a destination production  $\mathbb{P}_D = Xy$  and some origin production  $\mathbb{P}_O = Xz$  for some base object  $X$  such that  $\bar{z} \subseteq \bar{y}$ . Our goal is to evaluate performance as we vary  $N$ , the number of common transformations (i.e., the size of  $\bar{z}$ ).

To be comprehensive, for a given  $y$ , we generate all possible  $z$ 's, i.e., all possible permutations of  $y$  subsets. For example, for  $y = abcdefghi$ , among the many possible  $z$ 's are  $a$ ,  $ib$ ,  $bi$ , and  $hdea$ . We set the length of  $y$  to 8 to give us a rich but manageable number of subsets. In this case, we get a total of 109,600 possible  $z$ 's. Potentially, we could have  $8 * 7$  different  $\gamma$  factors (all possible pairs of transformations) and 8  $\alpha$  factors (one for each transformation). However, to keep our experiment tractable, we set the  $\gamma$  and  $\alpha$  parameters constant across all transformations.

We measure performance by comparing affinity obtained by an algorithm to the affinity of the Optimus Algorithm. As discussed earlier, we are interested in the percentage of times an algorithm is optimal, how closely an algorithm achieves optimal affinity, and for which affinity values an algorithm is optimal.

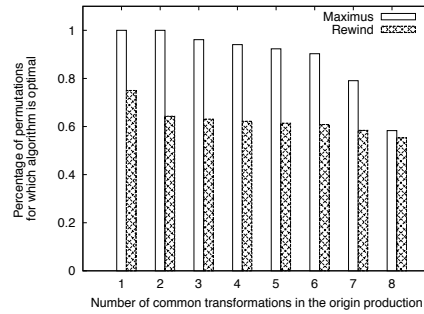
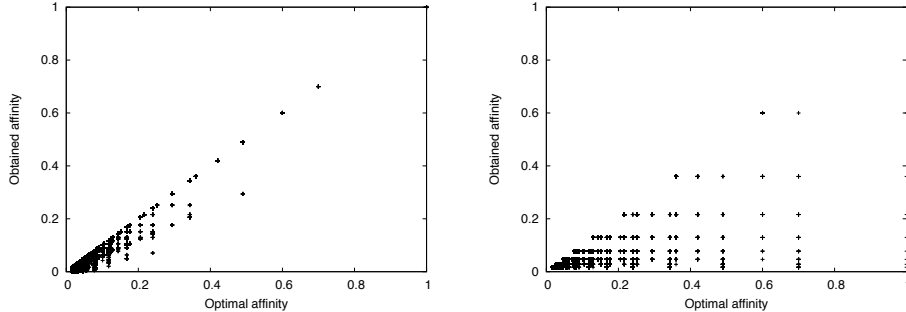


Figure 8: Percentage of optimal  $z$ 's given  $\gamma = 0.7$ ,  $\alpha = 0.6$



(a) Affinity scatter plot for Maximus Algorithm (b) Affinity scatter plot for Rewind Algorithm

Figure 9: Affinity scatter plots for Maximus and Rewind algorithms given  $\gamma = 0.7$ ,  $\alpha = 0.6$

In Figure 8 we consider a scenario where  $\gamma(x, y) = 0.7$  and  $\alpha(x) = 0.6$  for all  $x, y \in \bar{y}$ . Each pair of bars shows performance for origin productions with  $N$  transformations. For example, if  $N = 3$ , the origin has 3 transformations, all shared with the destination. The vertical axis shows the percentage of permutations for which an algorithm is optimal (i.e., yields the same affinity as Optimus). For instance, for  $N = 8$ , both Rewind and Maximus match Optimus' performance in about half of the cases. *Both Rewind and Maximus seem to do better for smaller size permutations*, i.e. cases where there are few common transformations between the origin and destination productions. This seems reasonable since for shorter  $z$ , there are fewer transformations to reverse or swap and thus fewer reconstructions to consider.

Figure 9(a) shows a scatter plot of obtained affinity vs. optimal affinity for the Maximus

Algorithm, for the same scenario as the previous figure. Points along the diagonal represent instances where Maximus yielded the same affinity as Optimus. Point below the diagonal represent cases where Maximus did not do as well. Note that *many of the suboptimal cases occur when the resulting affinity is quite low*, and these are the cases that are probably not of interest since even the optimal reconstruction has poor affinity. Contrast the behavior of Maximus with the behavior of Rewind, shown in Figure 9(b). In this case, Rewind is suboptimal across the spectrum, even in cases where the optimal affinity is high.

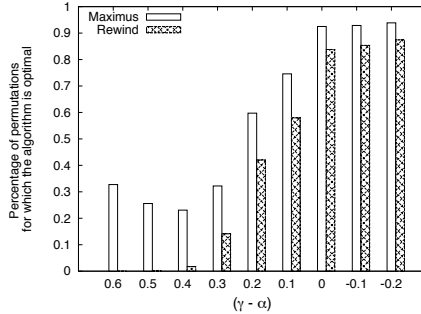


Figure 10: Percentage of optimal z’s, given different ( $\gamma - \alpha$ ) values ( $\gamma = 0.7$ )

Do these trends continue to hold as we vary affinity factors? Generally yes, but the value of the parameters with respect to each other can change the relative performance. For example, Figure 10 shows the performance of Maximus and Rewind as we vary  $\alpha$  and keep  $\gamma$  constant at 0.7. The horizontal axis is labeled with the *difference* between  $\gamma$  and  $\alpha$ . For instance, for the 0.3 value we have  $\alpha = \gamma - 0.3 = 0.4$ . The vertical axis shows the fraction of permutations (over all  $N$ ) for which Maximus (or Rewind) yields the best affinity.

From the figure, it is clear that *both algorithms perform substantially better as the difference between  $\gamma$  and  $\alpha$  decreases*. For instance, for large differences, Maximus only does well in about 30% of the cases, while Rewind never finds the best solution. For small differences, and if  $\alpha$  is larger than  $\gamma$ , the algorithms perform in the 90% range.

## 5 Related Work

Lineage, or provenance, has been extensively studied in the context of relational databases and scientific computation. References [11, 12] survey data provenance characteristics and recent research in provenance in databases.

Lineage is generally classified into *coarse-grained* and *fine-grained* lineage. Coarse-grained lineage records the workflow of the data processing tasks, i.e. the steps required to transform the data. Fine-grained lineage, on the other hand, is more specific to the data product and records derivation of data items, generated in the result of transformation steps. Furthermore, reference [4] defines *why* lineage, which describes which sources contribute to the existence of data, and *where* lineage, which describes the location where the data is extracted from. Finally, many research efforts in lineage focus on how lineage is recorded and stored (or computed).

There has been growing interest in recording lineage for non-relational transformations. Systems like Chimera [7], myGrid [18], and Trio [15] all support lineage for non-relational transformations. Moreover, Chimera supports reconstructions by relying on an object’s workflow metadata to re-execute transformations [7]. However, such reconstructions are only limited to forward reproductions so no properties of transformations are exploited.

The idea of *inverse* transformations in the context of lineage was first presented in reference [16]. They defined inverse functions in a relational sense, where an inverse function generates

source tuples given one or more output tuples. They also introduced the notion of a *weak-inverse* function that generates a subset or superset of source tuples. This idea of inverse functions was later studied in the context of data warehouse transformations in reference [6]. However, there has been no work to our knowledge that studies inverse transformations in combination with commutativity or compensation properties.

Tracking lineage and exploiting lineage to reconstruct objects has also been studied as part of version control and configuration management. References [5, 3] both provide a good overview of related research that has been done in this area. The key difference, however, between our work and version management is in that we allow “imperfect” reconstructions which are modeled by affinities.

The problem of generating a reconstruction sequence given an origin and destination sequences is similar to the approximate string matching, or edit distance, problem. More specifically, the problem as described by our model without compensation could be viewed as a special case of *extended edit distance*, defined in reference [14], that considers change, insert, delete, and swap operations. However, our case is different from what has been studied previously in that the inserts and deletes are restricted to the end of a string and in that the cost of operations is dependent on the participating symbols. Reference [8] surveys state of art algorithms available for a broad family of string matching problems.

## 6 Conclusion

In this paper, we described a model for finding object reconstructions using lineage, as applied to photographs. Our model exploits lineage beyond the typical forward reproductions. The reconstructions we study may use inverse transformations, exploit transformation properties such as commutativity and compensation, and may in general be imperfect giving “approximate” results. This flexibility increases the number of possible reconstructions and, hence, increases our chances of recovering lost data. However, in order to use such reconstructions, we need to measure “degradation.” Thus, we proposed an affinity model along with the composition rules that specify how we measure a reconstruction’s value to the end user. We then formulated the problem of finding an object’s reconstruction and proposed an efficient algorithm for it. With a number of representative scenarios, we show that this algorithm achieves a good balance between performance and quality of the results.

In this work, we considered only the quality of reconstructions. However, we could have also considered the cost of reconstructions (measured in time, resources, or monetary units). For example, we could assign costs to the individual objects in a production tree and/or each transformation invocation. A reconstruction’s cost could then be its base object’s cost and/or sum of the costs of its transformation invocations. A reconstruction’s value to the user could be a combination of its cost and affinity.

Although we used photo transformations as our driving application, our model is not limited to the domain of photographs. For instance, our model also covers video and text files. Indeed, transformations on video and text data exhibit the same properties of reproducibility, reversibility, commutativity, and compensation. Many video transformations are similar to the photo transformations, e.g., cropping, resizing, format conversion. On the other hand, the text domain is conveniently limited to only a few general transformations and conveniently offers straight-forward ways to compute affinities. Accordingly, our model can help the database community address the many challenges when managing graphic, video, and text data common in scientific and business applications.

## References

- [1] Lightroom. <http://adobe.com/products/photoshoplightroom>.
- [2] Bioact project. <http://i.stanford.edu/bioact>.
- [3] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.
- [4] P. Buneman, S. Khanna, and W. chiew Tan. Why and where: A characterization of data provenance. In *In ICDT*, pages 316–330. Springer, 2001.
- [5] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.
- [6] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1):41–58, 2003.
- [7] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, Washington, DC, USA, 2002. IEEE Computer Society.
- [8] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
- [9] Photoshop. <http://adobe.com/products/photoshop/photoshop>.
- [10] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.
- [11] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.
- [12] W. C. Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.
- [13] Transformers animated series. [http://en.wikipedia.org/wiki/The\\_Transformers\\_\(animated\\_series\).characters](http://en.wikipedia.org/wiki/The_Transformers_(animated_series).characters).
- [14] R. A. Wagner. On the complexity of the extended string-to-string correction problem. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 218–223, New York, NY, USA, 1975. ACM.
- [15] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.
- [16] A. Woodruff, M. Stonebraker, A. Woodruff, and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *In ICDE*, pages 91–102, 1997.
- [17] Xmp. <http://www.adobe.com/products/xmp>.
- [18] J. Zhao, C. A. Goble, R. Stevens, and S. Bechhofer. Semantically linking and browsing provenance logs for escience. In *ICSNW*, 2004.