Error-Tolerant Version Space Algebra *

Eugene R. Creswick
Stottler Henke Associates, Inc.
1107 NE 45th, Suite 310, Seattle, WA, USA
rcreswick@stottlerhenke.com

Aaron M. Novstrup
Stottler Henke Associates, Inc.
1107 NE 45th, Suite 310, Seattle, WA, USA
anovstrup@stottlerhenke.com

ABSTRACT

Application customization has been extensively researched in the field of Programming by Demonstration (PBD), and Version Space Algebra has proven itself to be a viable means of quickly learning precise action sequences from user demonstrations. However, this technique is not capable of handling user error in domains with actions that depend on parameters that accept myriad values. Activities such as image, audio and video editing require user actions that are difficult for users to precisely replicate in different circumstances. Demonstrations that are off by a single pixel or a split-second cause traditional composite Version Spaces to collapse.

We present a method of incorporating error tolerance into Version Space algebra. This approach, termed Error-Tolerant Version Spaces, adapts Version Space Algebra to domains where the tactile capabilities of the user have a much greater chance of prematurely collapsing the hypothesis space that is being learned. The resulting framework is capable of quickly learning in domains where perfectly consistent user input can not be expected. We have successfully applied our technique in the domain of image redaction, allowing our users to quickly specify redactions that can be reliably applied to many images without the entry of explicit parameters.

Author Keywords

Smart Environments, Error Tolerance, Version Spaces, Programming by Demonstration

ACM Classification Keywords

D.2.2 Design Tools and Techniques: User Interfaces; H.1.2 Models and principles: User/Machine Systems

General Terms

Algorithms, Experimentation, Human Factors

*This work was supported by the Air Force Research Laboratory's Information Directorate (AFRL/RI), Rome, NY, under contract FA8750-09-C-0099.

IUI'10, February 7–10, 2010, Hong Kong, China.

INTRODUCTION

The field of Programming by Demonstration (PBD) endeavors to enable non-programmers to augment computer interfaces by teaching autonomous agents to perform complex actions that have been demonstrated by users. In its simplest form, programming by demonstration exists in rote recording and playback features in many different environments. Microsoft Office's macro recorder, Emacs macros, and myriad other applications provide recording and playback features that watch a carefully-crafted sequence of user actions and then allow that precise sequence to be repeated at a later time. This approach is definitely of value; however, this form of rote learning does not account for the context of the user actions, nor does it generalize to new circumstances. Therefore, it is possible to do a great deal of damage with recorded macros if they are not created with great care. PBD aims to mitigate this risk by learning "programs" that can run robustly in new execution contexts by generalizing from multiple demonstrations [2, 5].

Many approaches have been proposed for different forms of PBD—we focus on one technique in particular: Version Space Algebra [3].

VERSION SPACE ALGEBRA

Version Space Algebra maintains a set of all possible hypotheses for a given problem domain. These hypotheses are represented in a decomposed form, spread across a hierarchy of version spaces. "Leaf" version spaces are generally trivially simple, consisting of hypotheses about singular values (such as integers, ratios, or strings). These axiomatic version spaces are combined to create more complex composite version spaces by using the following operations:

Union A union of version spaces contains the full set of all valid hypotheses in each of the component version spaces.

Join A join of version spaces contains the cross-product of the valid hypotheses in the component version spaces.

Transform Transforms are used to convert the input and output types of version spaces to match those of other unioned version spaces, or to provide semantics to the tuples that result from a join operation.

The complete description of these operators is out of scope for this paper. Such a description can be found in Tessa Lau's seminal work [3].

Learning is achieved by training the top-level composite version space on a demonstrated (input, output) pair, where the input contains the context of the demonstration, and the output is the desired output state. The top-level version space decomposes the example into sub-input and sub-outputs as necessary, and trains its composite version spaces on the decomposed inputs and outputs. This process continues to the leaves, where the axiomatic version spaces are either narrowed (if the new example is consistent, but different from past examples), collapsed (if the new example is *not* consistent with past examples) or unchanged (if the new example exactly matches past examples). A version space collapse indicates that a hypothesis was discarded and that collapse is propagated up the version space hierarchy, possibly causing other composite version spaces to collapse as well, if they no longer contain valid hypotheses. Lau et. al have shown that this hierarchical structure enables extremely fast learning. In one case, they demonstrated the ability to learn text editing tasks in as little as 1-2 demonstrations [4].

Version Space Algebra has one Achilles' heel that prevents this technique from being applied to many domains: user error during demonstrations can cause version spaces to collapse prematurely. For example, consider the task of drawing rectangles on images of various sizes. This task is particularly important in the area of information redaction. Drawing subsequent rectangles to redact similar content on a series of images at varying resolutions is both time consuming and difficult to do consistently. A user can easily specify the first example by manually dragging a selection tool to specify a rectangular region. The first example will narrow the set of possibilities significantly, but not sufficiently to learn a usable "program" for drawing rectangles. At least one subsequent example must be provided in a different context. In this domain, that means that the user must provide a consistent example on an image of a different size. To do so, the user must draw precisely the same rectangle on a new image. This is extremely difficult to do reliably in all but the simplest cases. Errors of one pixel are sufficient to cause the version space to collapse prematurely, resulting in no benefit at all.

In this paper we present an approach to Version Space Algebra that is tolerant of these types of user error. We start by discussing the limited related work in this area, then we present our initial approach to error tolerant version space algebra, we briefly discuss our specific use cases, and finally we conclude and discuss our goals for subsequent research.

RELATED WORK

User demonstration error can occur in any programming by demonstration domain; however, different approaches are necessary for different domains. Chen and Weld developed CHINLE, a learning system that incorporates techniques for handling errors in demonstrations within widget-based interfaces [1]. CHINLE allows users to detect and fix errors during training by providing the user with a view of the sequence of demonstrations. This view allows the user to retract or correct incorrect demonstrations. This approach works well when the user can readily identify these incorrect demonstrations, but that is not always the case. We are concerned with demonstration errors that are the result of tactile difficulty in precisely specifying an action. In such a situation, it is unlikely that the user will be able to identify an incorrect example from a list of recorded demonstrations.

Numerous PBD techniques have focused on reducing the error of inference engines. Opsis [6] presents users with the choice to create new examples based on old examples, but no generalization is done at all. Rather, "imitation" is used to directly generate programs from user input. Peridot [7, 8] helps users generate interface widgets, including shapes of various sizes and purposes. Peridot uses inferred graphical constraints to allow sloppier user actions, but this is greatly helped because Peridot only needs to deal with the relationships that are typical in user interfaces. In contrast, Error-Tolerant Version Spaces have no such restrictions.

ERROR-TOLERANT VERSION SPACE ALGEBRA

We focus on the domain of image manipulations, and particularly drawing or selecting shapes on an image. This task (shape drawing) arises any time a user needs to specify a selection area, perform a crop, resize, or place an image or other shape within an existing image repeatedly in a predictable way, such as when redacting content. This approach is also applicable to any task where the set of valid user inputs is very high and the parameters to user actions are ordered—such as pixel locations or color selection. In our sample domain the set of inputs is represented by combinations of pixel locations with respect to the loaded image dimensions. Other example use cases include adjusting volume controls, specifying temporal coordinates in video or audio editing, and any application that involves input mechanisms that acquire ordered inputs. The rest of this paper focuses on the task of training an agent to redact rectangular regions of a given input image. To further simplify the text, we only consider selections in one dimension[1].

Error-Tolerant Version Spaces are grounded on the idea that each demonstration is based on a single true hypothesis but the demonstrated values may be perturbed from the true hypothesis by a certain error for the given context of the example. Therefore, error-tolerant version spaces narrow the set of valid hypotheses more conservatively than Lau et. al's Version Spaces [3]. This is accomplished by the addition of two version space functions:

similar $similar(D, H)$ encodes the assumption that for some distance metric d, $d(D, H) < \epsilon$ for any demonstration D, a true hypothesis H, and error tolerance ϵ. If $d < \epsilon$ is true, then $similar(D, H)$ returns true. The purpose of $similar$ is to conservatively narrow the set of valid hypotheses—during training—to the intersection of those hypotheses that are $similar$ to the user's demonstrations.

filter the filter function transforms a set of hypotheses into a (generally smaller, or unit) set of values that represent the most likely valid hypotheses when a version space is

[1]Two-dimensional selections are simply a combination of two one-dimensional selections.

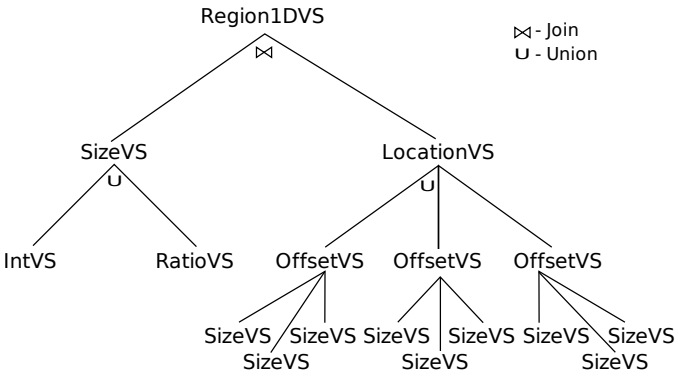

Figure 1. Version Space for one-dimensional regions. IntVS and RatioVS are axiomatic version spaces, the repeated SizeVS instances have been elided for clarity.

evaluated on a real input. We expect that $filter$ will often either be the identity function (in which case no filtering is done) or it will be a strict aggregator (for example, to select the most conservative of, or the average of the incoming hypotheses).

Figure 1 shows a simple version space for learning one-dimensional regions. Regions may have a fixed size or a size relative to the size of the image, and they may be located either a fixed or relative offset from the front (left/top) edge of the image, the back (right/bottom) edge of the image, or the center of the image. The region may be anchored to the location based on the front (left/top) edge, back (right/bottom) edge, or center of the region. This version space is made error tolerant by using a definition of $similar$ to narrow the hypotheses spaces of each of the axiomatic version spaces ($IntVS$ and $RatioVS$). These particular axiomatic version spaces typically only exist in one of three states: (a) Any value, (b) a specific value, or (c) nothing (collapsed). The "Error-Tolerant" versions contain a boundary-set representable region that is determined by intersecting the regions around each example based on the evaluation of $similar$ (and thus on the distance metric d and error tolerance ϵ).

DISCUSSION

Our solution is motivated by a simple model of the errors that users are expected to make, which is based upon our experience implementing and using Error-Tolerant Version Space Algebra for image redaction tasks.

We assume that the size and location of a demonstration are within a fixed, known error bound. Specifically, with an error bound of 2 pixels, we can make some assumptions about the true demonstration that the user meant to provide. For example, if a demonstrated region has a width of 10 pixels, and is centered at 15 pixels from the edge of the image, we can conclude that the true size is no less than 8 and no more than 12. We can also conclude that the left offset of this demonstration was meant to be between 8 and 12 pixels from the left edge of the image, because the demonstration showed a left offset of 10 pixels. We make no assumptions about the distribution of errors. In particular, we do not

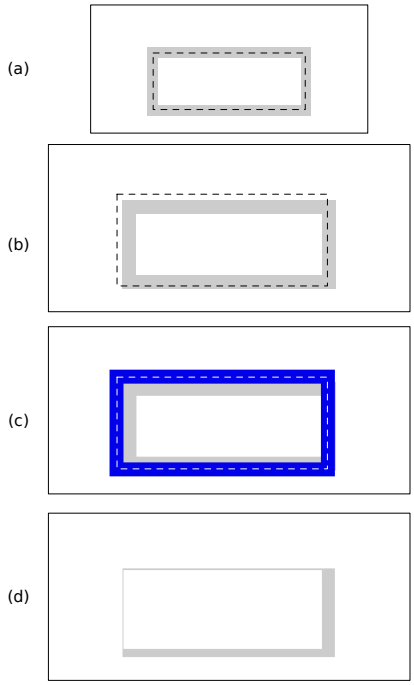

Figure 2. A sequence of demonstrated selections (dashed rectangles), and the valid regions learned from the examples (shaded regions). The outer rectangles represent image borders.

assume that small errors are more likely than large errors, nor do we assume that errors are more likely to be biased in one direction or the other (e.g. making the selection slightly larger than necessary, rather than slightly smaller).

It is important to note that this error model leads to different conclusions than other (also reasonable) models. For example, perhaps users tend to make larger errors with larger images, in which case we would require a relative rather than fixed error tolerance. Alternatively, the error tolerance could apply to the individual end points. In that case, we would assume that the true left endpoint is at least 8 and no more than 12 and the true right endpoint is at least 18 and no more than 22 when the target is the fixed region (10,20) and the error tolerance is 2 units. Note that the region covered by the union of the accepted hypotheses in this second error model is smaller (8 to 22) than the error model we chose to use (8 to 24). This is a point of consideration for certain use cases and demonstrates how the choice of error model can affect the learning properties of the algorithm.

Figure 2 displays a series of two demonstrations on two images of different sizes. In Figure 2 (a), a region is learned around the demonstrated rectangle. In Figure 2 (b), a second demonstration is given; however, because of the changed size, the user is unable to precisely match the same region. Despite the discrepancy, the version space does not collapse. Figure 2 (c) shows that the error tolerant regions around both of the demonstrations do intersect on all sides. Because this intersection is non-empty, the version space still contains hypotheses, which are depicted by the shaded region in Figure 2 (d).

Note that the intersection in Figure 2 (d) is a region, not an individual hypothesis. Instead of collapsing, as a normal version space would, the error-tolerant version space has reduced to a set of tightly clustered hypotheses. A direct comparison between standard and error-tolerant version spaces is impossible, since standard version spaces collapse when given imprecise demonstrations. Nonetheless, it is true that because error-tolerant version spaces do not narrow as quickly, they can require more training examples. The precise number is entirely dependent on the similarity measure used, the error tolerance, and the nature of the examples. In general, however, the maximally acceptable erroneous demonstrations must be observed before the error-tolerant version space achieves perfect learning.

In practice, this has not proven to be a problem, but that may depend greatly on the details of the domain in which this technique is employed. We expect that in the error-tolerant domains where these version spaces are applicable, it may be sufficient to quickly converge to a set of hypotheses clustered around the "true" hypothesis. In preliminary experimental work, we sampled truncated Gaussian models of input region size and user error to generate training/test data. Our results indicate a sizable, but not unreasonable, increase in the number of training examples required to achieve acceptable error as compared to standard version spaces utilizing perfect demonstrations (from an average of two examples with standard version spaces to an average of thirteen with error-tolerant version spaces). These results have been confirmed on a smaller scale by our practical experience with our image redaction tool, where we have found that four or five examples are usually sufficient. We also noted that even with perfect examples, the standard version spaces collapsed prematurely in over 90% of the trials due to rounding errors in the RatioVS.

CONCLUSIONS AND FUTURE WORK

Investigations continue into techniques that will improve upon this research, and we are particularly motivated by the following issues that arose during the development of Error-Tolerant Version Space Algebra:

Redaction error models revealed dependencies When learning conservative redactions on images, we found that the necessary error model revealed dependencies between the component version spaces used to represent redaction size and redaction locations. Because each version space maintained a range of valid hypotheses during most of the learning process, the conservative result (that covered all demonstrations) was not present in the cross-product of the size and location hypotheses, rather the bounds on the location had to be used independently with the largest size possible to ensure complete coverage.

Learning improves with error In fact, learning the correct hypothesis *requires* examples that exhibit the maximum error, as mentioned above. This is undesirable, since it may take significant time for that to happen. However, the degree of error in the generated results is bounded by the error tolerance. Therefore, the executing version spaces always generate results that are within the error tolerance.

Learning does not improve with repeated demonstrations It is reasonable to expect the learning algorithm to converge on the correct hypothesis as evidence accumulates. However, because error-tolerant version spaces only consider whether a hypothesis is $similar$ to the user's demonstrations, they gain no information from demonstrations that are $similar$ to all of the remaining hypotheses. We envision a probabilistic approach to address this shortcoming. It is also worth noting that when traditional version spaces encounter similar or repeated demonstrations, they either collapse or also learn nothing, respectively.

We have presented an approach to learning with Version Space Algebra when user demonstrations are subject to a form of error that has not previously been addressed. This approach greatly increases the applicability of Version Space Algebra by maintaining version spaces that would have otherwise collapsed prematurely.

REFERENCES

1. J. H. Chen and D. S. Weld. Recovering from errors during programming by demonstration. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 159–168, New York, NY, USA, 2008. ACM.

2. A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

3. T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

4. T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 527–534, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

5. H. Lieberman. *Your Wish is My Command: Programming By Example (Interactive Technologies)*. Morgan Kaufmann, 1st edition, February 2001.

6. A. Michail. Imitation: An alternative to generalization in programming by demonstration systems. Technical report, University of Washington School of Computer Science and Engineering, TR# UW-CSE-98-08-06, 2006.

7. B. A. Myers. *Creating user interfaces by demonstration*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

8. B. A. Myers. Peridot: creating user interfaces by demonstration. pages 125–153, 1993.