Learning Behavior from Demonstration in Minecraft via Symbolic Similarity Measures

Brandon Packard
Drexel University
Philadelphia, PA 19104, USA
btp36@drexel.edu

Santiago Ontañón
Drexel University
Philadelphia, PA 19104, USA
santi@cs.drexel.edu

ABSTRACT

This paper focuses on the challenging problem of learning behavior in a complex environment purely form observation of human performance. Specifically, we explore the performance of a collection of symbolic similarity measures in modeling the behavior of a human performing tasks in the Minecraft video game using learning from demonstration. We also analyze the performance of these measures using four different symbolic representations for the training data.

Categories and Subject Descriptors

I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems: Games

General Terms

Theory, Algorithms

1. INTRODUCTION

This paper focuses on *Learning from Demonstration* (LfD) [23, 4, 18], also known as Learning from Observation, Behavioral Cloning, or Apprenticeship Learning. Specifically, the goal of the work presented in this paper is: given a set of traces of the behavior of a human in several scenarios in a given domain (a computer game, in our case), learn how to predict her behavior in a similar scenario. This task is interesting, since it has the potential to enable generating AI for games by demonstration.

A significant amount of work exists in LfD (as discussed later in Section 7). In this paper, we focus on a Case-based reasoning (CBR) [1] approach to LfD, and specifically, on evaluating the performance of symbolic similarity measures for this task. Case based reasoning (CBR) is a problem solving methodology based on solving new problems by reusing past solutions. CBR is closely related to lazy supervised machine learning techniques such as the nearest-neighbor rule [7]. Specifically, a CBR system learns by storing problem/solution pairs (cases) in a *case base*, and solves new problems by identifying closely related problems in the case base, and adapting the stored solutions. One of the key underlying reasoning mechanisms of CBR is *similarity assessment*: CBR systems usually employ a similarity measure to determine which cases from the case base are most similar to the problem at hand. CBR has been applied to a number of game AI tasks such as playing RTS games [3, 26, 17], FPS games [5], or Tetris [11, 20], among others.

In this paper, we focus on learning complex single-player behaviors, and study symbolic representations of the world state and the performance of a collection of state-of-the-art symbolic similarity measures. We study the results of using these representations and measures to learn complex player behaviors in the *Minecraft* computer game. Our goal is to study the strengths and weaknesses of using a symbolic representation in this application domain, which will aid in the development of robust LfD methods.

To evaluate the similarity measures of interest, we recorded traces of human behavior in Minecraft, containing both the evolution of the world state and all the player actions. Four different similarity measures were employed: Jaccard similarity, Levensthein similiarity, Weighted Levenshtein similarity, and propositionalization followed by a Hamming distance. Additionally, we compared the performance of each similarity measure using four different types of world representations, which we call: *Original*, *Filtered* (removing data points where the human was not performing any action), *Discretized* (where numerical values were discretized into discrete categories), and traces that are both filtered and discretized. We describe each of them below.

The remainder of this paper is organized as follows. After we briefly discuss our problem statement, Section 2 introduces our application domain. Section 3 describes the representation formalism used to describe world states, actions and traces. After that, Section 4 presents the similarity measures employed in our study. Sections 5 and 6 describe our experimental setup and results, respectively. Finally, the paper concludes with related work and conclusions.

1.1 Problem Statement

The problem that we are trying to solve is learning to perform complex tasks by observing human behavior, with the long term goal of allowing to define AI behaviors by mere demonstration. Specifically:

Figure 1: A screenshot of Minecraft.

Given: A collection $\mathcal{R} = \{R_1, ..., R_n\}$ of previous traces of human behavior, where each trace contains the behavior of a human in a given scenario, in order to achieve a given goal g.

Predict: The behavior of the human in a new, similar, scenario when the human is trying to achieve the same goal g.

In order to address this problem, we used the *Minecraft* video game as our application domain, and recorded traces of a human trying to achieve a specific goal in Minecraft ("obtain a piece of cobblestone").

2. MINECRAFT

Minecraft (shown in Figure 1) is an expansive, open ended game that focuses on exploration and building. Although there are enemies to fight and items to collect, the game imposes no true goal on the player, instead letting them set their own. The players move about in a 3-dimensional world, which is divided into blocks that (with a few exceptions) the player can pick up and put down as they please. Players are able to kill enemies, build homes, tear down structures, and collect items to craft new tools. Due to these attributes, Minecraft requires both short term, almost reflexive decisions as well as long-term planning. For the purposes of this paper however, the goal pursued is very specific, and only requires a small set of reactive actions. Specifically, Minecraft exhibits the following properties:

- Almost deterministic: Most of Minecraft is deterministic, but it also boasts some stochastic features, such as the spawning of enemies or what items drop when certain blocks are broken.
- Partially Observable: How far a player can be seen depends on their settings, but even the farthest vision range is a very small fraction of the world.
- Dynamic: The environment is rich with other agents, in the form of passive and aggressive entities. Passive entities, such as cows or pigs, tend to be hunted by the player for food and items. Aggressive entities, however, will actively pursue the player and try to destroy him. These entities, along with the passage of time, comprise the dynamic parts of the environment, and act independently of the player.
- Continuous: The environment of Minecraft is continuous, with the position and rotation of the player and other entities being measured as real values.

$$\begin{aligned}
X_{162} = {} & \mathit{holding}(17,1,0) \wedge \mathit{position}(-108.0, 63.0, 155.0) \wedge \\
& \mathit{rotation}(-29.7, 51.0) \wedge \mathit{selectedBlock}(17, -109, 66, 156) \wedge \\
& \mathit{target}(\mathit{none}, 0, -1, 0) \wedge \mathit{block}(1, 16563) \wedge \\
& \mathit{block}(2, 5720) \wedge \mathit{block}(3, 1543) \wedge \mathit{block}(4, 83) \wedge \\
& \mathit{block}(17, 185) \wedge \mathit{block}(18, 3962) \wedge \\
& \mathit{level}(0) \wedge \mathit{health}(20) \wedge \mathit{food}(20) \wedge \\
& \mathit{pMob}(\mathit{pig}, -94, 64, 143) \wedge \mathit{pMob}(\mathit{pig}, -110, 64, 141) \wedge \\
& \mathit{pMob}(\mathit{pig}, -103, 63, 137) \wedge \mathit{pMob}(\mathit{pig}, -99, 63, 137) \wedge \\
& \mathit{pMob}(\mathit{sheep}, -139, 65, 145) \wedge \mathit{pMob}(\mathit{sheep}, -137, 68, 151) \wedge \\
& \mathit{pMob}(\mathit{sheep}, -137, 65, 147) \wedge \mathit{pMob}(\mathit{sheep}, -137, 67, 149) \wedge \\
& \mathit{pMob}(\mathit{pig}, -91, 68, 120) \wedge \mathit{groundItem}(17, -109, 64, 157) \wedge \\
& \mathit{groundItem}(17, -108, 64, 157) \\
Y_{162} = {} & \mathit{rotating}(0.15, 0.0), \mathit{choppingWood}(-1)
\end{aligned}$$

Figure 2: An example world state, with its corresponding action from one of the *original* traces.

All of these factors taken together make Minecraft an excellent domain in which to study the modeling of complex behaviors. Additionally, although in this paper we focus on reactive behaviors, Minecraft affords an opportunity to model both short and long term plans.

3. TRACES

To accomplish the goal of learning to perform complex tasks from observing a human playing Minecraft, data was gathered from the game in the form of *traces*. A *trace* $R = [(X_0, Y_0), ..., (X_T, Y_T)]$ is a sequence of *entries*, where each entry (X_t, Y_t) contains the world state X_t at a given time t, and the actions Y_t the player executed at time t.

As the player moves throughout the Minecraft environment, the actions Y_t capture what the player is actively doing at that time – such as walking, running, and mining. The world state X_t captures information about the state of the world at that time, including such data points as position, rotation, location of nearby entities, inventory contents, and health. A normal game of Minecraft runs at 20 ticks (the in-game unit of time) per second. Data for the traces is sampled once a tick.

The world state is represented as logical clauses of the form $X_t = x_0 \wedge x_1, \wedge ... \wedge x_m$. Each predicate x_i takes the form of a Prolog term, and represents some aspect of the world. For example, the term *health*(20) would mean that the player has 20 health. Figure 2 shows an actual world state with its corresponding action. Although world states may have an arbitrarily large number of predicates, those predicates belong to 15 different types (*position*, *rotation*, *block*, etc.), some of which can appear multiple times per entry.

Similarly, actions are also represented as logical clauses $Y_t =$

$[y_0 \wedge y_1 \wedge ... \wedge y_m]$. Each predicate y_i represents an action the player is taking at that time, also in the form of Prolog terms. Notice that we need to represent actions as clauses, since players might execute more than one action at a time (walking while turning and swinging a weapon), or even no actions. For instance, in the example in Figure 2, the player was rotating at the same time as chopping wood ($choppingWood(-1)$ means the player is chopping wood with its bare hands).

In the remainder of this paper we will use the term *original traces* to refer to traces where all the information above is recorded exactly as it is collected from the game. In addition to these original traces, we also constructed three additional types of traces in the following way:

- *Discretized* traces: which take some of the numerical values and replace them with discretized values, and turns absolute coordinates into relative coordinates. For example, $rotation(45, 75)$ might become $rotation(NorthWest, Up)$. Four different discretizations are performed, as seen in Table 1.
- *Filtered* traces: which have any entries where the player was doing nothing removed.
- *Discretized-Filtered* traces: combine both these post-processing steps.

Our current dataset consists of 2 sets of 5 traces each, for a total of 10 traces. Each set was obtained by creating 5 exact duplicates of a freshly generated world state, and then attempting to carry out the actions in as similar a way as possible for each instance. Both sets had the same goal of obtaining a piece of cobblestone, and all 10 traces were taken by the same person within the research group. As obtaining a piece of cobblestone is a very early-game goal, the sequence of tasks that the player performed was the same for all 10 traces. The only difference between sets was the starting state of the world. Specifically, to obtain a piece of cobblestone, the player needs to: 1) first, collect logs, 2) then craft some planks, 3) craft a workbench, 4) craft some sticks, 5) place the workbench, 6) craft a wooden pick, 7) and finally mine some stone (which automatically turns into cobblestone). Some of these steps involve navigating around the map trying to find the appropriate blocks or locations for the different tasks.

4. SYMBOLIC SIMILARITY MEASURES

In order to test the collection of similarity measures in our study, and to study the performance of symbolic representations of the world, we set up a CBR system based on a nearest neighbor retrieval algorithm [7]. Basically, given a trace R^{test} (the *test trace*) for which we would like to predict behavior by learning from a collection of training traces $\mathcal{R} = \{R_1, ..., R_n\}$. Given an entry $(x_t, y_t) \in R^{test}$ in the test trace, the CBR system finds which is the entry in the training traces that has the most similar world state to x_t, and the corresponding action in such entry is used to predict the behavior of the human. To determine the prediction error of our method, we then compare our prediction against y_t.

We evaluated the following similarity measures:

- *Jaccard*: Given two clauses (represented as sets of terms), we define the Jaccard similarity as the size of their intersection over the size of their union, which can be characterized by the following equation:

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

  where $X \cap Y$ is a clause that contains only those predicates present both in X and in Y, and $X \cup Y$ is a clause that contains the union of predicates in X and in Y. $|\cdot|$ represents the number of predicates in a clause. Notice that for a predicate to be in the intersection, it must be a perfect match. For example, $\{pMob(pig, -94, 64, 143)\} \cup \{pMob(pig, -94, 64, 142)\} = \emptyset$, since the last parameter does not match.

- *Levenshtein*: Given two clauses, the Levenshtein distance counts the number of edit operations that we need to perform to one of the clauses, in order to convert it into the other. The *edit* operations that we consider are adding a value, removing a value, and substituting a value by another. Our similarity measure between clauses is based on first computing the edit distance between the individual terms that compose the clauses, and the aggregating them to generate the final distance. To measure distance between two terms, we first represent each of them as a tree, and then use Pawlik and Augsten's tree edit distance measure [19], we denote this distance as $\delta_L(x_1, x_2)$.

  Given that a clause can be seen as a set of terms, computing the similarity between two clauses X_1 and X_2 with a large number of terms using the edit distance might have a prohibitive cost. For that reason, we employ an approximation algorithm (Algorithm 1). This algorithm works by first computing a matrix with as many rows as terms in X_1 and as many columns as terms in X_2. Each position of this matrix contains the similarity between the corresponding row and column terms. This similarity is computed as $1 - \delta_L(x_i, x_j)/max(|x_i|, |x_j|)$, i.e., by computing the edit distance, normalizing it, and then turning it into a similarity. Here, $|x_i|$ represents the size of a term, measured as the number of edit operations needed to construct this term from scratch from the empty term. Then, the similarity between the two clauses is computed by adding the maximum similarity value in each column of this matrix, and dividing by the number of terms in X_1 (the largest of the two clauses). This approximation algorithm thus has polynomial cost with respect to the number of terms in the clauses, instead of the exponential cost required to compute the exact edit distance.

- *Propositional*: finally, we implemented a propositionalization approach [14], where we converted the world state into a feature vector of fixed length, where each feature is either numerical or categorical. In order to achieve that, we only considered the closest aggressive mob (enemy) and the closest non-aggressive mob (e.g., an animal), in order to make the feature vector fixed length. We also only considered the most common block types in the traces in our dataset. The resulting feature vector had 90 features. After this conversion

Original Predicate	Original Values	New Predicate(s)	New Values
$rotating(\alpha, \beta)$	$\alpha \in [-180, 180]$ $\beta \in [-180, 180]$	$xRotation(X), yRotation(Y)$	Where: $X \in \{Left, Center, Right\}$ $Y \in \{Up, Center, Down\}$
$rotation(\alpha, \beta)$	$\alpha \in [0, 360]$ $\beta \in [-90, 90]$	$rotation(D, H)$	Where: $D \in \{S, SW, W, NW, N, NE, E, SE\}$ $H \in \{Down, Center, Up\}$
$amob(type, x_a, y_a, z_a)$ $pmob(type, x_a, y_a, z_a)$	Where (x_a, y_a, z_a) are the absolute coordinates of an enemy/creature, and *type* is the specific type of enemy/creature.	$amob(type, x_r, y_r, z_r)$ $pmob(type, x_r, y_r, z_r)$	Where (x_r, y_r, z_r) are the relative coordinates of an enemy/creature to the player, and *type* is the specific type of enemy/creature.
$groundItem(type, x_a, y_a, z_a)$ $projectile(type, x_a, y_a, z_a)$	Where (x_a, y_a, z_a) are integers representing the absolute position of the dropped item/projectile, and *type* identifies the specific type of item/projectile.	$groundItem(type, x_r, y_r, z_r)$	Where (x_r, y_r, z_r) are integers representing the position of the dropped item/projectile relative to the player, and *type* identifies the specific type of item/projectile

Table 1: Predicates that are discretized (or transformed from absolute to relative) and the resulting predicates. Note that there are many other predicates that are not discretized and therefore are not included in this table.

Algorithm 1 clauseSimilarity(X_1,X_2)

```
1: if $|X_1| < |X_2|$ then
2:    return $clauseSimilarity(X_2, X_1)$
3: end if
4: $n_1 = |X_1|, n_2 = |X_2|$
5: $M = n_1 \times n_2$ matrix of zeroes.
6: for $x_i \in X_1$ do
7:    for $x_j \in X_2$ do
8:       $M(i,j) = 1 - \delta_L(x_i, x_j)/max(|x_i|, |x_j|)$
9:    end for
10: end for
11: return $\frac{1}{n_2} \sum_{j=1 \dots n_2} (max_{i=1 \dots n_1} M(i,j))$
```

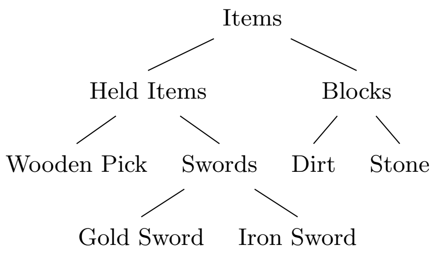

Figure 3: An excerpt of the taxonomy of concepts used in the domain of Minecraft.

was done, we employed a Hamming distance (counting the number of features that are an exact match between the feature vectors of two world states).

The Levenshtein distance is a more fine-grained measure than Jaccard (which operates at the granularity of predicates), but it also has a higher computational cost. In order to cull the runtime to a reasonable level, a technique called MAC/FAC [12] was used for retrieval in the implemented CBR system. MAC/FAC uses a two stage retrieval process: given that we want to find the nearest neighbor from a large collection of entries, we first use a computationally cheap similarity measure to filter among those entries, and then use a more expensive method to compute structural matches on the entries returned from the first stage. Specifically, we used a Jaccard Similarity to find the 10 nearest neighbors, and then the Levenshtein distance to choose which among those is the closest.

4.1 Levenshtein Weights

In the basic definition of Levenshtein Distance each of the edit operations (adding, removing, or substituting any element or subelement of a tree) is given a cost of 1. To increase the accuracy of this technique, we provided background knowledge about how similar or different the different constants appearing in the terms are. This is done by creating a taxonomy of the different concepts appearing in Minecraft, grouping them by our intuitions about their similarity (an excerpt of this taxonomy is shown in Figure 3). Then, we used this taxonomy to calculate edge weights between each of the concepts in this taxonomy according to the following formula (adapted slightly from [13]):

$$w(c,p) = \left(\beta + \frac{(1-\beta)\bar{E}}{E(p)}\right)\left(\frac{d(p)+1}{d(p)}\right)^{\alpha}[IC(c) - IC(p)]$$

where w and c are concepts, α and β are constants, $E(p)$ is the entropy of p, $\bar{E}$ is the average entropy, $d(p)$ is the depth of p, and $IC(x)$ is the information content of x, defined as $log^{-1}P(x)$ (where $P(x)$ is the probability that a given game entity is an instance of concept x). The distance between any two concepts is then the sum of all edge weights on the shortest path between them. Once these distances are calculated, they are used as the costs for the substitution operation in the Levenshtein distance.

	Original			Filtered			Discretized			Filtered/Discretized		
	0-1	JL	LL	0-1	JL	LL	0-1	JL	LL	0-1	JL	LL
Random	0.841	0.823	0.823	0.973	0.919	0.919	0.973	0.919	0.919	0.949	0.907	0.907
Most-likely	0.630	0.630	0.630	0.794	0.765	0.765	0.630	0.630	0.630	0.776	0.759	0.759
Jaccard	0.634	0.576	0.542	0.710	0.606	0.534	0.680	0.625	0.601	0.707	0.604	0.560
Levenshtein	**0.619**	**0.556**	0.550	0.698	0.580	0.568	0.667	0.606	0.601	0.683	0.568	0.558
W. Levenshtein	**0.619**	0.557	0.551	**0.696**	**0.578**	0.566	0.662	0.606	0.601	0.682	0.566	0.557
Propositional	0.663	0.577	**0.540**	0.722	0.583	**0.512**	**0.649**	**0.563**	**0.541**	**0.672**	**0.524**	**0.482**

Table 2: Average loss for various algorithms and loss functions, evaluated using a one-vs-one procedure (lower is better). Results which are statistically significantly better than the rest are underlined.

Finally, even if some of the values in the terms in our dataset are numeric, in the experiments reported in this paper we consider them as if they were categorical constants (i.e., 1.0 is equally distant from 2.0 than from 3.0).

5. EXPERIMENTAL EVALUATION

In order to evaluate our approach, we used "obtaining a piece of cobblestone" as the goal g to achieve. We collected traces using two different worlds, collecting 5 different traces in each world, for a total of 10 traces, all taken by one person. Although the world is procedurally generated, the same seed was used for each set of 5, and traces were taken from the moment the world was first generated, for consistency. The average trace length is 723 entries, which is 40 seconds of gameplay.

We evaluated the performance of our approach using a number of loss functions:

- 0-1 Loss: returns 1 if the predicted actions match the actual actions exactly, and 0 otherwise.
- Jaccard Loss (JL): we define the Jaccard Loss as one minus the Jaccard similarity between the predicted actions and the ground truth.
- Levenshtein Loss (LL): Levenshtein distance between the predicted action set and the actual action set.

We report average loss for all the entries of the traces in the test set. We note that 0-1 loss is always lower or equal to the Jaccard loss, which is lower or equal to Levenshtein.

6. RESULTS

Table 2 shows the average loss for all 4 trace types and all 3 loss functions, evaluated using a one-vs-one procedure (for each pair of traces in our dataset, using one as the test set and the other as the training set). The loss values are calculated by using the respective loss function of the predicted actions compared to the actual actions in the test trace. For the 0-1 loss columns ("0-1"), the higher the number in the table, the less entries that were correctly predicted by that method. For example, a value of 0.000 represents that every entry was correctly predicted, and an entry of 0.500 represents 50% of actions were correctly predicted. Statistical significance was tested using a paired t-test.

We compared the results against a series of baseline predictors ("Random", "Most-likely"). The first row ("Random") in the table shows the results of retrieving one of the entries in the training set at random, and using it as the prediction. The second row ("Most-likely") shows the results from always predicting the set of actions that was the most common in the training set. The other 4 rows ("Jaccard", "Levenshtein", "Weighted Levenshtein", and "Propositional") show results for generating a prediction using a nearest neighbor algorithm with the corresponding similarity measure. As expected, the Levenshtein similarities achieve the best results when using the original traces. However, surprisingly, the simple propositional distance outperforms Levenshtein when the traces are discretized. Although further experiments are needed to determine the cause, we hypothesize that this is caused by the two-step retrieval process (MAC/FAC) used for Levenshtein, where the first pass uses the Jaccard similarity, thus limiting the performance of Levenshtein. One interesting bit of data is that the original traces have much lower error than both filtered and filtered/discretized when using 0-1 loss, but only a little lower when using Jaccard or Levenshtein loss.

Table 3 shows the average loss for all 4 trace types and all 3 loss functions, but evaluated using a leave-one-out procedure. Again, we observe that the Levenshtein distances achieve the best results for the non-discretized traces, but that once discretized, a propositional similarity achieves better results. Surprisingly, discretizing the traces made the Levenshtein distances perform worse in some scenarios. However, this does not seem to affect the propositional similarity. Our hypothesis is that discretizing made some of the entries in our dataset become too similar, specially when using a Levenshtein similarity, where two predicates with the same arguments but different head might have a higher similarity than two predicates with the same head but different arguments. Thus, it seems that given our world representation, the weighted approach should be taken one step further, and also increase the penalties for matching predicates with different heads.

Additionally, it is interesting to note that, although the discretized traces have much higher error on the one-vs-one tests, their error on the leave-one-out tests is equivalent to or a little lower than the original traces. This is likely due to the discretization increasing the amount of nearly identical world states, but the extra amount of training data available in the leave-one-out tests allowing for better predictions, reversing the negative effects. It is also notable that the traces which have been both filtered and discretized have lower error than the original traces when using Jaccard or Levenshtein loss with a leave-one-out testing scheme, due to

	Original			Filtered			Discretized			Filtered/Discretized		
	0-1	JL	LL	0-1	JL	LL	0-1	JL	LL	0-1	JL	LL
Random	0.841	0.823	0.823	0.973	0.919	0.919	0.973	0.919	0.919	0.949	0.907	0.907
Most-likely	0.630	0.630	0.630	0.794	0.765	0.765	0.630	0.630	0.630	0.776	0.759	0.759
Jaccard	0.589	0.480	0.467	0.679	0.505	0.484	0.572	0.462	0.455	0.621	0.444	0.386
Levenshtein	0.619	0.508	0.496	**0.661**	0.480	0.395	0.560	0.476	0.425	0.600	0.429	0.374
W. Levenshtein	**<u>0.558</u>**	**0.438**	**0.389**	**0.661**	0.480	0.395	0.560	0.458	0.425	0.600	0.429	0.374
Propositional	0.601	0.476	0.424	0.671	**0.472**	**0.382**	**0.555**	**0.441**	**0.414**	**0.587**	**0.398**	**0.350**

Table 3: Average loss for various algorithms and loss functions, evaluated using a leave-one-out procedure (lower is better). Results which are statistically significantly better than the rest are underlined.

	1	2	3	4	5	6	7	8	9	10
1	0.000	0.631	0.772	0.453	0.838	0.853	0.866	0.770	0.854	0.946
2	0.583	0.000	0.563	0.543	0.648	0.834	0.849	0.804	0.849	0.922
3	0.505	0.515	0.000	0.603	0.687	0.829	0.826	0.792	0.815	0.906
4	0.672	0.500	0.489	0.000	0.676	0.816	0.874	0.801	0.817	0.898
5	0.651	0.478	0.518	0.480	0.003	0.890	0.854	0.805	0.855	0.975
6	0.677	0.734	0.714	0.596	0.725	0.000	0.736	0.646	0.769	0.700
7	0.625	0.638	0.560	0.563	0.617	0.535	0.028	0.705	0.641	0.645
8	0.682	0.748	0.727	0.596	0.733	0.694	0.736	0.024	0.787	0.703
9	0.596	0.598	0.602	0.531	0.667	0.633	0.612	0.644	0.001	0.637
10	0.714	0.743	0.728	0.682	0.746	0.642	0.673	0.671	0.834	0.029

Table 4: 0-1 loss for Jaccard similarity in a one-vs-one procedure (split into quadrants).

the discretization allowing for better predictions.

Table 4 shows the individual results for using each trace to predict every other trace (one-vs-one procedure), with Jaccard similarity and 1-0 loss. This table has at least two notable features. First, as can be seen by the top-left to bottom-right diagonal, running a trace on itself does not always result in correctly predicting the action entries, as might be expected. This is because consecutive entries in a trace might sometimes have an identical world state, but have different actions.

Second, if the table is split into quadrants as shown, an interesting phenomenon occurs, which is easiest to see with the Jaccard results, but present in the other results as well. As previously noted, this data was generated from 2 sets of 5 similar traces. Considering the top left quadrant to be quadrant one and labeling them clockwise, quadrant 1 shows traces from set 1 predicting other traces from set 1, quadrant 2 shows traces from set 2 predicting traces from set 1, quadrant 3 shows traces from set 2 predicting traces from set 2, and quadrant 4 shows traces from set 1 predicting traces from set 2. As can be derived visually from the table, quadrant 2 has the lowest values, with an average value of only 0.148. As can be predicted due to the traces being from homogenous sets, quadrants 1 and 3 have much higher values of 0.410 and 0.318, respectively (after removing the values of running each trace on itself, for fairness). However, quadrant 4 has an average value of 0.338, which means that using traces from set 1 to predict traces from set 2 was more effective than using the ones from set 2.

7. RELATED WORK

The problem of learning behavior from traces has received significant attention in the literature. For example, Ross, Godron, and Bagnell, study learning from demonstration in the context of the Super Tux Cart and Super Mario Bros games [21]. However, they use an online learning approach, where we focus on batch learning. Their system, called DAgger, creates a policy iteratively, it uses the current policy to gather more data, which is added into the dataset. The new policy is then derived by attempting to mimic the expert on the entire set, not just the new portion. For Super Tux Cart, they were able to prevent the car from ever falling off the track after 15 training iterations. It also averaged a distance of around 3000 on Super Mario Bros, where stages are 4200-4300 on average.

Other approaches to learning from demonstration include *inverse reinforcement learning* [2, 25], Dynamic Bayesian Networks (such as Hidden Markov Models) [8, 18], supervised learning techniques [22] (including relational approaches [16]). The reader is referred to [4] and [18], for recent surveys of learning from demonstration.

Concerning the use of Case-Based Reasoning for LfD, the Darmok system [17], combined learning from traces and a case-based planning approach. Specifically, it learned plan snippets from traces, and these snippets where then reused to create a plan to play the real-time strategy game Wargus. Darmok required humans to annotate the learning traces with the goals they were pursuing, but was able to win 41% of its games after learning from just one learning trace. Compared to the approach presented in this paper, Darmok was designed for games requiring long-term planning instead of reactive control, and thus would not be suitable for domains such as Minecraft.

Floyd and Esfandiari [10] studied case based learning in robot soccer, simulated space combat, and physical robots with varying degrees of success. The robot soccer player had decent prediction accuracy, but was difficult to visually distinguish from an actual player - similar results were obtained for the space combat agent. Finally, with the allowance of considering right and left turns identical, their physical robot learning agents were able to achieve perfect accuracy by choosing the correct action and executing it for the correct duration. In their approach, Floyd and Esfandiari studied an approach based on representing the world state as a feature vector, and also considered domains with a fixed set of actions. In our domain, players can execute more than one action at a time (e.g., walking, while turning and attacking), making prediction more complex.

A different take on learning human-like behavior in computer games comes from Bauckhage, Thurau, and Sagerer [6]. Therein, they use neural networks to perform pattern recognition and perform actions in hope of imitating human behavior. Their system was able to fairly successfully emulate various human-like behaviors in Quake 2, such as learning efficient paths (moving through a map in a deliberate way to collect all items efficiently instead of arbitrarily moving around the map). Their results show that learning human-like behavior by training agents on data generated by humans is indeed feasible. However, their work was applied primarily to reactive behaviors, whereas our focus is on more complex behaviors that might require many steps.

Another related approach to learn behavior in games is to use reinforcement learning. For example, Smith, Lee-Urban, and Muñoz-Avila [24] present an approach, called RETALIATE, to learn behavior for first-person shooters. RETALIATE was tested against 3 different opponent profiles: opportunistic, possessive, and greedy, with very different play styles. Which agent was being fought against was changed after each game instance, and RETALIATE was able to adapt quickly and win 47 of the 50 games. They also ran RETALIATE against HTNbots, which changed their strategies dynamically within a single game, and it was able to adapt and defeat this agent as well. Although it is very good at adapting to dynamically changing circumstances, they focus on learning to adapt to an opponent's strategy, which is different from the goal we pursue in this paper.

The reader is referred to [15] for a more general description of open challenges in applying machine learning to games.

8. CONCLUSIONS

In this paper, we presented a case-based reasoning approach to learning behavior from demonstration in Minecraft, focusing on the evaluation of symbolic similarity measures and relational representations of the world state. We detailed the 4 similarity measures, 3 loss functions, and 4 trace types that were employed in our study.

It is clear from observing the data that our CBR approach using some of the distance measures performs much better than the baseline of predicting a random action, or just predicting the most common action, especially when using the Jaccard and Levenshtein loss functions (which provide a more fine-grained performance measure). However, performance is still not good enough as for using this technique for automatically learning behavior from demonstration.

In order to improve the performance of our approach, we would like to improve the weight calculations for the weighted Levenshtein similarity measure, and perform experiments with further similarity measures. We are specially interested in exploring world representations that capture the geometry of the world, which is key for determining behavior, and is not captured in our current world representation. Additionally, we would like to further improve our evaluation methodology by letting the CBR system control the character in the game, and actually act out the actions it predicts, which would provide us a different method of evaluating our results. Additionally, adding the notion of having a goal/subgoals to the traces may help to further reduce errors (for example, the goal at the start might be to obtain 4 logs – once the traces show that the inventory contains at least 4 logs, the goal is changed to making a crafting table). Finally, some actions in the game take time to complete, sometimes up to a few seconds, and some others only make sense in the context of a specific sequence of actions. Therefore, our methods may be able to be improved by including a representation of how long an action was performed or of previous actions in the world state representation (for example incorporating temporal backtracking retrieval [9] into our algorithm).

9. REFERENCES

[1] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.

[2] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

[3] D. W. Aha, M. Molineaux, and M. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *Case-based reasoning research and development*, pages 5–20. Springer, 2005.

[4] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[5] B. Auslander, S. Lee-Urban, C. Hogg, and H. Munoz-Avila. Recognizing the enemy: Combining reinforcement learning with strategy selection using case-based reasoning. In *Advances in Case-Based Reasoning*, pages 59–73. Springer, 2008.

[6] C. Bauckhage, C. Thurau, and G. Sagerer. Learning human-like opponent behavior for interactive computer games. In *Pattern Recognition*, pages 148–155. Springer, 2003.

[7] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[8] E. W. Dereszynski, J. Hostetler, A. Fern, T. G. Dietterich, T.-T. Hoang, and M. Udarbe. Learning probabilistic behavior models in real-time strategy games. In *AIIDE*, 2011.

[9] M. W. Floyd and B. Esfandiari. Learning state-based behaviour using temporally related cases. In *16th UK Workshop on Case-Based Reasoning.*

[10] M. W. Floyd and B. Esfandiari. Toward a domain-independent case-based reasoning approach

for imitation: Three case studies in gaming. In *Proceedings of the Workshop on Case-Based Reasoning for Computer Games (held at the 18th International Conference on Case-Based Reasoning)*, pages 55–64, 2010.

[11] M. W. Floyd and B. Esfandiari. A case-based reasoning framework for developing agents using learning by observation. In *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, pages 531–538. IEEE, 2011.

[12] K. Forbus, D. Genter, and K. Law. MAC/FAC: a model of similarity-based retrieval. *Cognitive Science*, 19:141–205, 1994.

[13] J. J. Jiang and D. W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *CoRR*, cmp-lg/9709008, 1997.

[14] S. Kramer, N. Lavrač, and P. Flach. *Propositionalization approaches to relational data mining*. Springer, 2001.

[15] H. Muñoz-Avila, C. Bauckhage, M. Bida, C. B. Congdon, and G. Kendall. Learning and game ai. In *Artificial and Computational Intelligence in Games*, pages 33–43, 2013.

[16] S. Natarajan, S. Joshi, P. Tadepalli, K. Kersting, and J. Shavlik. Imitation learning in relational domains: A functional-gradient boosting approach. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1414, 2011.

[17] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram. On-line case-based planning. *Computational Intelligence*, 26(1):84–119, 2010.

[18] S. Ontañón, J. L. Montaña, and A. J. Gonzalez. A dynamic-bayesian network framework for modeling and evaluating learning from observation. *Expert Systems with Applications*, 41(11):5212–5226, 2014.

[19] M. Pawlik and N. Augsten. RTED: A robust algorithm for the tree edit distance. In *Proceedings of the VLDB Endowment)*, volume 5, pages 334–345, Instanbul, Turkey, 2009. VLDB Endowment.

[20] H. Romdhane and L. Lamontagne. Forgetting reinforced cases. In *Advances in Case-Based Reasoning*, pages 474–486. Springer, 2008.

[21] S. Ross, G. J. Gordon, and J. A. Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686, 2010.

[22] C. Sammut, S. Hurst, D. Kedzier, D. Michie, et al. Learning to fly. In *Proceedings of the ninth international workshop on Machine learning*, pages 385–393, 2014.

[23] S. Schaal et al. Learning from demonstration. *Advances in neural information processing systems*, pages 1040–1046, 1997.

[24] M. Smith, S. Lee-Urban, and H. Muñoz-Avila. Retaliate: learning winning policies in first-person shooter games. 2007.

[25] B. Tastan and G. R. Sukthankar. Learning policies for first person shooter games using inverse reinforcement learning. In *AIIDE*. Citeseer, 2011.

[26] B. G. Weber and M. Mateas. Case-based reasoning for build order in real-time strategy games. In *AIIDE*, 2009.