A Game-Independent Play Trace Dissimilarity Metric

Joseph C. Osborn, Michael Mateas
Center for Games and Playable Media, UC Santa Cruz
Santa Cruz, CA 95064, USA
jcosborn,mmateas@soe.ucsc.edu

ABSTRACT

This paper defines a metric for comparing play traces (sequences of user decisions) in a game-independent way. The properties of this metric are determined by a proposed tool we call the *Gamalyzer*, an exploratory visualization of arbitrary games which clusters together similar play traces. Our *Gamalyzer metric* is based on refinements to edit distance and has broad uses outside of visualization and, indeed, outside of games (e.g. player and opponent modeling, goal recognition, player mimicry, user testing, and so on). We validate our metric against one synthetic and one real-world data set, finding that Gamalyzer discerns designer-relevant differences between play traces nearly as well as hand-tuned feature selection while remaining game- and genre-agnostic.

The main problem with conventional game visualizations is that judging the similarity of two game states is an underspecified problem requiring knowledge of game rules and of the purpose for which the states are being compared. Gamalyzer, in contrast, directly compares sequences of actions: it considers *strategies* instead of *states*. Existing game visualizations require significant development effort for individual genres, games, and even specific queries. Our proposed visualization (based on the Gamalyzer metric) could quickly and inexpensively show designers the strategic landscape of their game without requiring specialized knowledge of statistics or machine learning.

Categories and Subject Descriptors

G.3 [**Probability and Statistics**]: Time Series Analysis; I.5.3 [**Clustering**]: Similarity Measures; K.8.0 [**General**]: Games

General Terms

Algorithms, Design, Measurement

Keywords

edit distance, sequence clustering, visualization

1. INTRODUCTION

Both digital and analog games exhibit highly emergent behavior which is difficult for designers to predict in advance. This trait is shared with general computer programs, but in the special case of games we can often assume that players are working towards goals such as victory over an opponent or over the game itself. Designers can manage this emergent complexity by means of useful exploratory visualizations that enumerate the strategies players use to achieve those goals, detect degenerate play, and summarize a game's emergent behaviors.

While many game visualizations have been developed, most of them apply only to specific games or to constrained genres. Even the most general visualizations tend to require that play traces are naturally represented as paths through a 2D map; this encodes an assumption that movement is the most important action players perform. This kind of visualization does reveal information about how players navigate through space, but designers must choose between easily viewing a small number of traces (as paths) and easily viewing aggregate behavior (as a heatmap); moreover, annotations that supplement this movement data with other records of game state or player choices make these diagrams increasingly hard to read.

While the problems with traditional game visualizations are not insurmountable, resolving them requires genre- or game-specific knowledge. We would like to develop a tool which serves equally well for real-time 3D games as it does for board and card games, with performance that scales acceptably both in game length and in the number of traces under analysis. Such a tool—let us call it a *Gamalyzer* to emphasize its genre-independence—must satisfy four requirements:

1. Gamalyzer's only input is play traces in some machine-readable format. Specifically, it has no knowledge of the game's rules.

2. Similar play traces should be grouped together for easy appraisal, selection, and so on.

3. Gamalyzer should not require designer-authored similarity measures, distance metrics, or layout constraints.

4. Gamalyzer must scale to support real-world play trace data sets, both in terms of its performance and its visual complexity.

In order to achieve these aims with a visualization, we need a way to compare play traces that affords the same characteristics: game-independence, a meaningful metric, minimal required human intervention, and scalability. To this end, we developed a metric named for the tool: the Gamalyzer metric.

2. RELATED WORK

Our basic approach has been to take straightforward ideas from other fields and apply them to game visualization, combining them in a novel way. We have tried to tackle problems emerging from game analytics and metrics by looking at how other fields visualize and group sequences of events.

2.1 Play Trace Visualization

Game metrics and analysis comprise a relatively new field, but one with significant motivation from and participation by the games industry. This emerges somewhat naturally from practices like playtesting and it is given urgency by business concerns such as profitability and user retention. Representative examples of play trace visualization from the academy and the games industry include histograms of game event counts and BioWare's overlaying of player actions (including meta-game actions like asking other players for help) onto a game's map [4].

Playtracer [1] is one of the few visualizations that neither maps state onto a game's navigational space nor is genre- or game-specific. Unfortunately, incorporating Playtracer into a design process is challenging, requiring that designers both identify relevant state features and define a state distance metric using those features; these problems are difficult even for experts. Wallner and Kriglstein specialize Playtracer by integrating domain knowledge [17], using a player's location in space as a shorthand for a state and rendering other information about that state with e.g. overlays and pie charts. Their work allows for other genre- and game-specific techniques to provide Playtracer-style results with less designer awareness of the underlying algorithms, but these still require human intervention.

These state-centric visualizations encounter three issues: first, exclusively representing states loses temporal information; second, comparing game states in a general way is not possible; and third, effective visualizations need substantial designer and developer time for every new game (and potentially every new query) to be visualized. We avoid these issues (which are inherent in directly visualizing game states) by focusing on the visualization of player inputs rather than game states, satisfying the first requirement of a Gamalyzer tool. While game-specific state-centric visualizations have significant value, we think they should be subordinate to a game-independent interface for displaying and selecting groups of play traces.

Game trees are traditionally used to represent both game states and the actions needed to create them. Unfortunately, these trees become very large very quickly, representing as they do the entire possibility space of a game. It is also difficult to compare distinct paths through a game tree. For example, two games of Chess might be very similar, but if they happened to begin with different moves they would reside in completely different parts of the game tree.

2.2 Sequence Clustering

We would like to aggregate similar traces together to meet our second requirement, but we need a more sophisticated notion of similarity than sharing a common game tree prefix. We therefore needed a similarity metric for arbitrary play traces. Gamalyzer also has to scale to support the hundreds of discrete event types that can occur even in a relatively simple game like Chess (32 pieces and 64 destinations produce a very large set of possible events). To attack these challenges, we leverage the body of knowledge on sequence clustering, a familiar (but open) problem in the speech recognition and bioinformatics communities (among others).

Briefly, the problem of sequence clustering is to take sequences of symbols (e.g. component molecules) and determine whether they belong to one group (e.g. a protein family) or another; or, in the unsupervised case, to determine how many distinct groups there are and how closely a given sequence of symbols matches each such group. Two approaches are generally used: *discriminative* methods locate shared structure and interpret the degree of sharing as a feature for use with a traditional clustering algorithm [8]; and *generative* methods learn statistical models (often hidden Markov models) from data, measuring similarity as the likelihood that a given statistical model would generate a candidate sequence [13].

Gamalyzer uses a discriminative technique, applying constraint continuous edit distance [3] (CCED) to sequences of game inputs. For our purposes, discriminative methods are easier to explain and encode fewer assumptions about the underlying game and player models than generative methods do. Our method is also reasonably efficient, scaling only linearly in the lengths of the sequences and in the number of sequences (due to an optimization described in Sec. 3.2).

Edit distance is a measure of how dissimilar two sequences are based on how many *edit operations*—insertions, deletions, and matches—would be required to change the first sequence into the second. We borrow three chief insights from CCED and related techniques: first, we can consider *changes* instead of matches; second, not all changes are equal (e.g. changing a 9 to an 8 may be easier than changing it to a 6, and changing it to a 1 may be impossible); and third, we can constrain the edit operations to improve efficiency and avoid degenerate cases such as deleting the entire first sequence and inserting the entire second sequence.

Our implementation takes the traditional dynamic programming approach to calculating edit distance [11]. A useful consequence of this algorithm is that we obtain not just the cost of changing one play trace to another, but also the entire sequence of edit operations and the partial costs up to each step in that sequence. This means that we could show users of Gamalyzer not just how similar entire play traces are, but also how these traces become incrementally more or less similar over time.

2.3 Player Modeling

The games community has devoted substantial effort to classifying players based on their actions (for an introduction, see [18]). These are often statistical analyses used to predict player behavior for adaptive game design or business intelligence [14], but generative player models have also been developed [10]. Most extant approaches are based on counting and classifying actions [14, 5], n-gram analysis or other forms of motif detection and counting [6, 4], or learning mappings from states to predicted player actions [15, 10].

Generally, these techniques require up-front work in selecting features, whereas the Gamalyzer metric can be applied with minimal effort or machine learning expertise. Our application of discriminative whole-sequence analysis and clustering to the domain of play trace analysis seems to be a novelty, although it bears repeating that we are trying to group and identify strategies rather than players or even player types. We believe that classifying players is a substantially harder task than recognizing strategies or goals. Still, it stands to reason that players might be distinguished by their preferences for different strategies.

3. THE GAMALYZER METRIC

The metric required by the Gamalyzer visualization is an adaptation of CCED [3] to play traces, i.e. sequences of inputs. The dissimilarity value we find is the (normalized) cost of changing one sequence into another using only insertion, deletion, and replacement operations. We assume that all such sequences share a common clock rate, but it may be possible to automatically insert "no-op" events to synchronize two sequences; at any rate, we leave that for future work. Taking insertion and deletion costs to be equal, we calculate the cost of changing one specific input into another using an *input* (as opposed to *input sequence*) dissimilarity metric; if such a change is not possible, then the input must be deleted or an input from the other sequence must be inserted. Given these costs, CCED finds a globally optimal edit path.

Inputs are comprised of two primary terms: a *determinant* and a *value*. Determinants and values are compound data which may consist of numbers, tokens, square lists (delimited by square brackets), and round lists (delimited by parentheses). If two inputs have distinct determinants, they are different kinds of decisions; changing one to the other is not possible. To compare two inputs' values, we must rigorously define what values are and how their similarity is evaluated. Partial value matches are given a dissimilarity δ based on the following rules:

1. Numbers are normalized according to their observed domain and their normalized distance gives a dissimilarity value. We assume here a uniform distribution, but other distributions (e.g. Bernoulli, Gaussian) are possible.

2. Non-numeric tokens are completely distinct ($\delta(\mathtt{a}, \mathtt{b}) = 1.0$). There is room here to add sophistication by considering probabilistic distributions of atoms.

3. The difference between two square lists a and b of length N is the sum
$$\sum_{i=0}^{N-1} \frac{\delta(a_i, b_i)}{N}.$$

4. The difference between two round lists a and b of length N is the sum
$$\sum_{i=0}^{N-1} \frac{(N-i)\delta(a_i, b_i)}{\sum_{j=1}^{N} j}.$$
Each successive term in the list is discounted linearly and normalized so that the maximum value of the sum is one.

By these rules, the ordering of an input's values amounts to a judgment on which parts of the input are *most important* from the designer's perspective. Since Gamalyzer is game-agnostic, designers must have some way to inform the metric about the game's notion of similarity. These encodings will be game-specific, but this language requires minimal overhead on top of traditional game telemetry reporting that may already be implemented in the game being examined. We illustrate the use of these four data types with examples:

In *Refraction* (or Chess), which piece is being placed (a token) is more significant than the specific X and Y coordinates (numbers) of its destination, but neither coordinate is more important than the other. These `move` inputs should therefore use a square list inside of a round list: (Piece, [X, Y]). Other inputs of Chess or *Refraction* might have determinants and values like "`castle`" and "`left`" or "`discard_piece`" and some piece identifier.

For a real-time game like *Super Mario Bros.*, we are interested in frame-by-frame inputs. If we were to create one Gamalyzer input event for each button (jump, move right, move left, or run), then some frames would yield several inputs and some frames would yield no inputs; this would make it hard to represent simultaneous actions and, more importantly, it violates Gamalyzer's assumption of a common clock rate among input sequences. Since simultaneous actions are possible, we must represent player input as the set of buttons held on a given frame; we will give all such inputs the determinant `move`. Among the buttons running is the most significant predictor of style [7], so we use a round list of numbers $(\mathrm{Btn}_{\mathrm{Run}}, \mathrm{XDistance}, \mathrm{Btn}_{\mathrm{Jump}}, \mathrm{Btn}_{\mathrm{Duck}})$, prioritizing the four types of input in that order. Each value represents whether the corresponding button was held (0 or 1) or the direction in which the player wanted to move (-1, 0, or 1).

Our approach (a lexicographically weighted product order) is lightweight compared to asking designers to define an input distance function, simple to explain by example, usable without perfect knowledge of the four rules, and easy to integrate into existing game metric instrumentation—either by altering the game's metrics output or through a post-processing step. We thereby achieve our third objective, requiring very little in the way of designer knowledge of the underlying statistical processes. Gamalyzer does not require the use of this specific dissimilarity function, but we provide it as a generally useful default; replacing it with a different function is straightforward but might reduce generality.

Given this input-against-input dissimilarity function δ, the continuous edit distance (without constraints) for changing the input sequence (play trace) A into the input sequence B can be defined by the recurrence [3]:

$$\Delta(A, B) = \min \begin{cases} \Delta(A, B_{\mathrm{Rest}}) + \mathrm{cost}_{\mathrm{ins}}(B_{\mathrm{First}}) \\ \Delta(A_{\mathrm{Rest}}, B) + \mathrm{cost}_{\mathrm{del}}(A_{\mathrm{First}}) \\ \Delta(A_{\mathrm{Rest}}, B_{\mathrm{Rest}}) + \delta(A_{\mathrm{First}}, B_{\mathrm{First}}) \end{cases}$$

If A (respectively B) is empty we insert (respectively delete) inputs until both sequences are exhausted. In this way every pair of inputs (A_i, B_j) from both traces is compared to find the minimum value of $\Delta(A, B)$. This value is then normalized by the sum of the lengths of the two sequences (multiplied by the deletion and insertion costs, respectively) to give a value between zero and one: the Gamalyzer metric. Being a dissimilarity metric, this is larger for less similar traces and smaller for more similar ones.

Since many of the intermediate minima found during this recurrence will be reused, it is traditional to use a dynamic programming implementation. We fill out an $|A| \times |B|$ array starting from the initial pair so that the value at index (i, j) is the edit distance $\Delta(A_{0..i}, B_{0..j})$ for the corresponding subsequences of A and B. We then find the optimal path by tracing backwards from $\Delta(A_{0..|A|}, B_{0..|B|})$ towards the origin. We chose this bottom-up approach because it afforded easier implementation of global constraints on the search as described in the next section: these constraints amount to avoiding certain regions of that array.

3.1 Constraint Continuous Edit Distance

Constraint continuous edit distance is so called because it adds two constraints to continuous edit distance [3]: the Sakoe-Chiba Band [11] and the Itakura Parallelogram. These are both envelopes (or *warp windows*) that constrain the permissible area of the $|A| \times |B|$ edit matrix. Intuitively, both prevent degenerate matches of early parts of one sequence against late parts of the other sequence. The Sakoe-Chiba Band ensures that no time points which are further apart than the window can be matched against each other, and the Itakura Parallelogram gives a window which is small at the beginning and end of the sequences but larger in the middle. In other words, the former constraint prevents traces which are too different overall from being comparable (i.e. having a dissimilarity less than one), and the latter constraint requires that comparable traces begin and end in similar ways. In a speech recognition context, the Sakoe-Chiba Band ensures that the fricative "th" in "thick" is not a good match for the one in "bath," and the Itakura Parallelogram makes the pronunciation of "crawfish" comparable to "crayfish" but not to "dolphin."

Both constraints make sense for speech recognition, but game play traces invalidate the assumptions of the Itakura Parallelogram: even given similar goals, not every trace will begin or end with similar actions. Under the Itakura Parallelogram, two traces which are similar up to a certain point but then diverge (or traces which begin differently but then converge) would be punished heavily. We would be forced to ignore their common prefix (resp. suffix) because their suffix (resp. prefix) was too different. We therefore use only the Sakoe-Chiba Band to constrain matches. Adopting either constraint with envelope width ω means that the performance of the dynamic programming approach can be improved from $O(|A||B|)$ to $O(|A|\omega)$—linear in the length of the first sequence. We assume that ω is provided by the designer, but it is straightforward to imagine reasonable defaults based on median play trace lengths.

Our unoptimized, single-threaded implementation in the Clojure Lisp dialect (which targets the Java virtual machine) takes five seconds to compute a 100×100 dissimilarity matrix for synthetic traces averaging 44 inputs long on a 2.6GHz Intel Core i7 laptop (and uses very little RAM). In practice, the metric would not be calculated $N \times N$ times, but $k \times N$ times, where k is the number of pivots (see Sec. 3.2) and N is the number of traces. This gives a scaling behavior which is linear instead of quadratic in the number of traces (and also linear in the length of traces), which gives us the performance side of our fourth requirement. In fact, these distance calculations can be done incrementally or offline, since traces are assumed to be immutable, so performance currently poses no barriers to interactive use.

Gamalyzer penalizes differences in play trace length using global constraints, insertion costs, and deletion costs. When the window is narrow relative to differences in length between sequences, sometimes no edit path can be found; in this case, it is not only fair but desirable to assume complete dissimilarity. We believe this is justified if the time ordering of moves is important; if earlier moves substantially influence the game state, similar moves that occur later on are taken in a different context and their seeming similarity is therefore only superficial.

Consider a player jumping over an obstacle: we would not want one player's hop over the level's first obstacle to match against another player hopping over its last obstacle. Those are similar actions, and if we were doing N-gram analysis or motif detection we might be interested in that similarity, but in terms of analyzing the entire play trace this is a nonsensical matching.

Reducing the length of traces will also reduce the time required to calculate the metric. The Mario AI Benchmark receives 24 inputs per second and games last up to one minute; this could be downsampled to one summary action per second to simplify the calculations (and, incidentally, the visual display). This downsampling operation is not strictly necessary for the short games being played in the Mario AI Benchmark, but would be important to consider for longer games. A dense representation of play traces—one input per frame—may have scaling issues (even in terms of storage) as game lengths increase. Because it is not possible in general to resample input sequences automatically, the easiest solution is one that game developers may already be using to minimize the storage requirements of their game metrics: aggregation. This can be implemented when gathering metrics, in a post-processing step on the raw data, or as a custom input format for Gamalyzer.

These aggregates do not have to be sufficient to reconstruct the whole game state, but they serve as a way to smooth over small differences in event order and to reduce the calculation time required by Gamalyzer. Aggregation could also theoretically improve the semantic value of the metric if it captured certain features or context that would be hard to recover from individual traces; but this is necessarily a game- or genre-specific operation. It would be better to avoid using aggregation for that purpose and find ways to parameterize the metric to account for those kinds of concerns.

3.2 Trace Selection

From the algorithm above, we see that Gamalyzer can usefully compare small numbers of traces, but how might the metric—or the visualization, for that matter—handle hundreds, thousands, or hundreds of thousands of traces? The key insight here is that while nearly every trace is unique, most are unique in relatively uninteresting ways. It should suffice to show exemplars (individual traces which capture interesting strategies) and merely summarize the rest of the population, offering special interactions to more closely examine subgroups. In this way, the user can opportunistically explore families of traces without being overwhelmed by huge quantities of data.

This problem also emerges in the automatic layout of large graphs. Brandes and Pich developed a linear-time approximation to classical multidimensional scaling based on selecting a small number (relative to the total number of nodes) of "pivots" and doing layout relative to those pivots rather than to the whole set of nodes, with the intuition that a small number of nodes suffices to determine the overall shape of the graph [2]. They compare two approaches to pivot selection: random choice, which requires large numbers of pivots to give good layouts; and the *maxmin* strategy, which picks the first pivot arbitrarily and selects each subsequent pivot to be the datum which has the greatest (max) least (min) distance from all currently-selected pivots. In other words, maxmin always chooses a trace which is most unlike any of the previous pivots. They (and we) prefer the maxmin strategy, because it reaches good layouts with a much smaller number of pivots than the random strategy: it captures more significant samples more quickly.

Pivot selection strategies like maxmin give us a way to efficiently show users the contours of the strategy space: our pivots are the play traces which are most usefully distinct from each other. Since pivot selection requires knowing the dissimilarity between every pivot and every other trace (even the ones we do not elect to show users), Gamalyzer can also provide statistics such as how many of the elided traces were most similar to each selected pivot or the overall distribution of traces among the pivots. The number of exemplar traces could be determined automatically (by continuing until the maxmin value reaches a threshold) or given by a parameter.

Pivot selection makes Gamalyzer scale better in terms of performance, but more importantly it reduces the required *visual* complexity by foregrounding the traces that add the most substantial information about the game's strategic landscape. We have therefore extended our metric to completely satisfy the fourth and final requirement of a Gamalyzer visualization, and it remains only to produce the graphical visualization. This is left for future work, but we will outline the basic idea below.

3.3 Towards a Gamalyzer Visualization

Spatiotemporal visualizations face their greatest challenges in environments where the space and its contents respond to player actions over time. It is unclear how to extend existing graphical representations like heatmaps to support branching outcomes based on player activity, especially given the time-sensitivity of these interactions: overlaying all possible paths and level states onto a single map would be difficult to read.

If a designer had a straightforward way of selecting specific *approaches* that players used to get through the level, then each strategy could be viewed and appraised separately or in relation to other strategies. This is the use for which the Gamalyzer metric was originally intended: the Gamalyzer metric identifies the distinct player policies for solving the level, and a Gamalyzer visualization shows them to the designer. A designer could use this visualization to select traces for display on an auxiliary spatiotemporal or statistical visualization, and in this way she could explore the game's play.

We imagine that all of the exemplar traces are laid out side by side, as poly-lines connecting easily distinguishable glyphs representing player inputs. Similar traces are nearer to each other than they are to dissimilar traces, and as traces become more and less similar over time they converge and diverge. Users could select traces or inputs to reveal extra information, either by dragging out selections over the Gamalyzer visualization or, ideally, over the auxiliary state visualization (to answer e.g. "How did the character get to this position?" if the state visualization is a heatmap, or "How did this quantity exceed this amount?" if the state visualization is a plot of values over time).

Noise	$k = 3$	$k = 4$	$k = 5$
0%	100%	100%	100%
15%	90%	97%	98%
30%	66%	83%	89%
45%	38%	58%	73%

Table 1: Gamalyzer's rate of successful archetype identification in synthetic data (drawn uniformly from three models) within five pivots, out of 1000 trials.

Such a visualization would give deep insights into the strategy space of a given game level. This kind of information is difficult to obtain except by watching individual playthroughs and recording sequences of interest, but this kind of observation does not scale and requires carefully controlled conditions.

3.4 Other Applications

The Gamalyzer metric gives a series of edit operations and dissimilarity measures between two play traces, along with an overall dissimilarity value. This metric has many applications outside of visualization; for example, differences with respect to canonical play traces could be used as features for machine learning classifiers. Given a set of traces, the Gamalyzer metric could locate a candidate trace for player modeling, goal recognition, or AI agent evaluation.

Another notable feature of Gamalyzer's assumptions is that they can be satisfied by activities that are not games at all. Any goal-directed sequence of actions is a candidate for analysis with the Gamalyzer metric (and, indeed, the Gamalyzer visualization). This includes task-oriented computer usage (such as game designer modeling [9]), medical treatment history, and conversational AI.

4. EVALUATION

We applied the Gamalyzer metric to two data sets: our own synthetic play traces for an abstract game (generated from a suite of player models) and the subset of the Mario AI Benchmark data [10, 12] used by Holmgård et al [7].

The synthetic data experiment (Table 1) evaluated Gamalyzer against 1000 sets of 60 traces each, where each trace was drawn from one of three models of an abstract game with three similar (but distinct) moves. Each model preferentially selects one among these three inputs, modulated by a noise percentage by which inputs are pulled from a different (randomly selected) model. We increased this noise to understand how Gamalyzer's discriminative performance degraded as noise increased.

Table 1 shows Gamalyzer's success rate in picking out the three models (i.e. selecting at least one instance of each of the three models as a pivot) from increasingly noisy data with three, four, and five pivots. Each row uses the same 1000 sets of synthetic traces. Even when all three models are not recovered, two distinct models always appear among the traces, and the redundantly-selected pivots are at the fringes where models act like each other. The synthetic data experiments suggest that Gamalyzer's distance metric successfully identifies distinct player strategies when applied to a data set with noisy play traces from a finite set of player models.

Next, we wanted to investigate the semantic value of the Gamalyzer metric. The play trace similarity measures used by Holmgård et al. were based on hand-selected features and seemed to be both well-motivated and effective at distinguishing how players moved through a level, so we used the clusters found in that work (along with their method of clustering) as a baseline [7]. Gamalyzer is sensitive to the level in which play traces take place, so we planned to compare a clustering based on the Gamalyzer metric for each level against the corresponding tagged player data for each level.

Unfortunately, this was not a perfect comparison—for example, many levels had no examples of a particular Holmgård et al. cluster, but Gamalyzer would still try to split those traces into four groups. Instead, we cut off clustering at each level after finding as many distinct clusters as were in the baseline for that level. We then compared the quality of each pair of clusterings (the baseline clusters and those found by Gamalyzer for a given level) using Adjusted Mutual Information—specifically, AMI_{max}. AMI_{max} is an information-theoretic measure tuned for small numbers of items relative to the number of clusters which discounts any similarities that could be due to chance [16]. Any value above zero indicates that the two clusterings are more similar than would be expected by chance, and a value of one means that the two clusterings are identical.

Across the 40 levels, we found an AMI_{max} of 0.28 with a standard deviation of 0.26. The high standard deviation was caused by levels where only one cluster was attested in the baseline data; these (trivially) perfect alignments skewed the distribution. Ignoring those levels, we found a mean of 0.22 with a standard deviation of 0.17. These values indicate that the manual feature selection and the Gamalyzer metric gave clusterings which were substantially more similar than could be expected by chance alone.

One significant difference between the Holmgård et al. feature vector and the Gamalyzer metric is that the former is *duration-insensitive*. A play trace which perfectly followed the target path for five seconds before falling into a pit would receive a high similarity score according to the feature vector even though the AI's behavior is much more diverse when considering the whole level. Gamalyzer would treat this pair of traces as similar at first, but strongly divergent overall (we suspect that large differences in trace length will be of significance to human designers as well). We performed a third experiment to investigate how much this artifact of the different approaches impacted the clusterings.

For this analysis, we dropped from each level any trace whose duration was more than one standard deviation away from that level's mean. We removed these traces from the (post-clustering) tagged Holmgård et al. data set and from the (pre-clustering) input to Gamalyzer. This difference in treatment was justifiable: a feature-based score is independent of which traces are included in the clustering, so removing individual samples post-clustering should not have a significant impact on the overall clustering; in Gamalyzer, on the other hand, all distances are relative to other traces, and removing those distances amounts to removing the pre-clustering data point. When all traces were of similar length, we saw an AMI_{max} of 0.44 with a standard deviation of 0.30 when including levels with only one cluster and an AMI_{max} of 0.32 with a standard deviation of 0.16 without those levels. This reinforced our intuition that much of the difference between the clusterings was due to time sensitivity.

To sum up, we found fairly similar clusters to those determined by the hand-selected and hand-tuned feature vectors from Holmgård et al. This was achieved solely by determining an order among the properties of player inputs—namely, considering running as the most important aspect of player intention, followed by moving right and jumping—and clustering based on the Gamalyzer distance metric under that assumption. Humans still have a role in this process, but the play trace features under examination fall naturally out of the game design and do not require substantial effort in defining or calculating time-varying features of the game state.

5. CONCLUSION

With few exceptions, game visualization tools have been constructed in an ad hoc, one-off manner. Generic approaches like Playtracer have so far required substantial awareness of the underlying mathematics of the clustering algorithm and an investment in feature selection.

While states are difficult to compare directly, sequences of actions are easier to compare. Even identical game states can carry substantially different design consequences: a player who visits a state once is very different from a player who visits that same state ten times. We can also assume that most players actively pursue goals, so that each trace captures an attempt to achieve one of a small set of objectives (relative to the number of traces). The goal of the Gamalyzer visualization is therefore to collate and present players' strategies for achieving goals so that designers can understand how their game is played in the wild.

To this end, we have developed a reasonably efficient Gamalyzer metric based on constraint continuous edit distance which achieves all four requirements of such a visualization. Our metric has additional applications including goal recognition and player mimicry; in fact, it is likely to be useful for any goal-directed sequence of interactions, not just games.

Gamalyzer only requires that input sequences share the same clock and that inputs be given in a particular format, with each input carrying a determinant and a value. The value must be written in terms of numbers, tokens, round lists, and square lists according to the designer's appraisal of each parameter's importance. In fact, this requirement is soft: Gamalyzer will work with any input-to-input distance metric δ such that $0 \leq \delta(i_1, i_2) \leq 1$ for all inputs i_1, i_2. Via maxmin pivot selection, this dissimilarity metric can highlight the most interestingly different traces to sketch out a game's strategic landscape. This technique could provide a useful initial visualization for play traces and its readability could be improved by a simple auxiliary display that expresses the selected traces' corresponding states. Gamalyzer can also reveal relationships between player strategies that are very difficult to visualize using existing tools.

6. ACKNOWLEDGMENTS

Thanks to Eric Butler and Adam Smith at the University of Washington and to Prof. Alex Pang's visualization seminar at U.C. Santa Cruz for early feedback. We also thank the researchers who shared their data sets [1, 10, 12].

7. REFERENCES

[1] E. Andersen, Y.-E. Liu, E. Apter, F. Boucher-Genesse, and Z. Popović. Gameplay analysis through state projection. In *International Conference on the Foundations of Digital Games*, FDG '10, pages 1–8, New York, NY, USA, 2010. ACM.

[2] U. Brandes and C. Pich. Eigensolver methods for progressive multidimensional scaling of large data. In M. Kaufmann and D. Wagner, editors, *Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 42–53. Springer Berlin Heidelberg, 2007.

[3] V. M. Chhieng and R. K. Wong. Adaptive distance measurement for time series databases. In *Advances in Databases: Concepts, Systems and Applications*, pages 598–610. Springer, 2007.

[4] M. S. el Nasr, A. Drachen, and A. Canossa. *Game Analytics*. Maximizing the Value of Player Data. Springer, Mar. 2013.

[5] M. Etheredge, R. Lopes, and R. Bidarra. A Generic Method for Classification of Player Behavior. In *Second Workshop on Artificial Intelligence in the Game Design Process*, pages 1–7, Aug. 2013.

[6] B. Harrison and D. L. Roberts. Using sequential observations to model and predict player behavior. In *International Conference on the Foundations of Digital Games*, pages 91–98. ACM, 2011.

[7] C. Holmgård, J. Togelius, and G. N. Yannakakis. Decision making styles as deviation from rational action: A super mario case study. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[8] C. S. Leslie, E. Eskin, A. Cohen, J. Weston, and W. S. Noble. Mismatch string kernels for discriminative protein classification. *Bioinformatics*, 20(4):467–476, 2004.

[9] A. Liapis, G. N. Yannakakis, and J. Togelius. Designer modeling for personalized game content creation tools. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[10] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis. Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93 – 104, 2013.

[11] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(1):43–49, 1978.

[12] N. Shaker, G. N. Yannakakis, and J. Togelius. Crowd-sourcing the aesthetics of platform games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

[13] P. Smyth. Clustering sequences with hidden Markov models. *Advances in Neural Information Processing Systems*, pages 648–654, 1997.

[14] P. Spronck and F. den Teuling. Player modeling in civilization iv. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[15] D. Thue and V. Bulitko. Modeling goal-directed players in digital games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 86–91, 2006.

[16] N. X. Vinh, J. Epps, and J. Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 9999:2837–2854, 2010.

[17] G. Wallner and S. Kriglstein. A spatiotemporal visualization approach for the analysis of gameplay data. In *Computer-Human Interaction*, pages 1115–1124, Austin, TX, 2012. ACM.

[18] G. Yannakakis, P. Spronck, and D. Loiacono. Player modeling. In *Dagstuhl Seminar on Artificial and Computational Intelligence in Games*, 2013.