Investigating the Solution Space of an Open-Ended Educational Game Using Conceptual Feature Extraction

Erik Harpstead, Christopher J. MacLellan, Kenneth R. Koedinger,
Vincent Aleven, Steven P. Dow, Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA
{eharpste, cmaclell, koedinger, aleven, spdow, bam}@cs.cmu.edu

ABSTRACT

The rich interaction space of many educational games presents a challenge for designers and researchers who strive to help players achieve specific learning outcomes. Giving players a large amount of freedom over how they perform a complex game task makes it difficult to anticipate what they will do. In order to address this issue designers must ask: what are students doing in my game? And does it embody what I intended them to learn? To answer these questions, designers need methods to expose the details of student play. We describe our approach for automatic extraction of conceptual features from logs of student play sessions within an open educational game utilizing a two-dimensional context-free grammar. We demonstrate how these features can be used to cluster student solutions in the educational game *RumbleBlocks*. Using these clusters, we explore the range of solutions and measure how many students use the designers' envisioned solution. Equipped with this information, designers and researchers can focus redesign efforts to areas in the game where discrepancies exist between the designers' intentions and player experiences.

Keywords

Educational Games, Representation Learning, Context-Free Grammars, Clustering

1. INTRODUCTION

Educational games are a growing sub-field of instructional technology. Researchers see video games as a compelling medium for instruction because they can offer students the ability to practice new skills within an authentic context that poses little personal risk [7]. These promising aspects of games have led many educational game designers to create "open games", which allow students to exercise creativity in how they solve problems. [12,25]. Open educational games are a form of exploratory learning environment and commonly use ill-defined problems as part of their designs [11,19]. While the tendency toward open experiences is compelling for educational game designers, it presents problems when analyzing student learning, a necessary part of designing activities to foster robust learning.

When designing an open game experience, the designer surrenders a degree of control over the nature and progression of the experience to the player [10]. This openness can be problematic to the designers of educational game experiences who are concerned that students receive some type of intended instruction and achieve a desired learning outcome. Educational game designers require a detailed picture of how students are playing a game in order to know if disparities exist between the designers' intentions and player experiences; and, if such disparities do exist, designers need to know where to focus redesign efforts.

To facilitate designers' and researchers' analysis of open educational games we propose a methodology for extracting conceptual features from student log data. We demonstrate our methodology in *RumbleBlocks*, an educational game designed to teach basic concepts of structural stability to young children [5]. The method takes as input logs of student gameplay and yields a set of conceptual features which describe student solutions. While some aspects of our approach are specific to *RumbleBlocks*, the general concept should be applicable to open educational games.

To automatically generate features in *RumbleBlocks*, we use a four-step process that converts the log data from student play into feature vectors. This process, which is the primary contribution of this paper, consists of discretizing the log data containing student solutions; generating a grammar from the discretized logs; using the grammar to parse each solution; and converting the resultant parse trees into vectors that concisely represent the structural components of the solutions. In the following sections, we first describe the game *RumbleBlocks* and then provide the details of the four-step process to extract features. Afterwards, we show the results of using the extracted features to cluster student solutions, which enables the identification of misalignment between designer intentions and student actions.

1.1 RumbleBlocks

RumbleBlocks is an educational game designed to teach basic structural stability and balance concepts to children in kindergarten through grade 3 (5-8 years old) [5]. It focuses primarily on three basic principles of stability: objects with wider bases are more stable, objects that are symmetrical are more stable, and objects with lower centers of mass are more stable. These principles are derived from the National Research Council's Framework for New Science Educational Standards [21] and other science education curricula for the target age group.

The game follows a sci-fi narrative where the player is helping a group of aliens who become stranded when their mother ship is damaged. Each level (see Figure 1 for an example level) consists of an alien stranded on a cliff with their deactivated space ship lying on the ground. The player must use an inventory of blocks to build a structure that is tall enough to reach the alien. In Figure 1, the player is dragging a third block (the highlighted square block) from the inventory (top left) to the tower-under-construction (bottom, center). Additionally, the player's structure must also cover a series of blue "energy balls" floating in space which are narratively used to power the space ship, but serve to both guide and constrain the players' designs. Once the student is confident in their design, they can place the spaceship on top of their tower triggering an earthquake that serves as a test of the tower's stability. If, at the end of the quake, the tower is still standing and the spaceship is still on top, the student passes the level and proceeds to the next level; otherwise they start the level over again.

Beyond the limits imposed by the energy ball mechanic and the types of available blocks, students are not very constrained in the

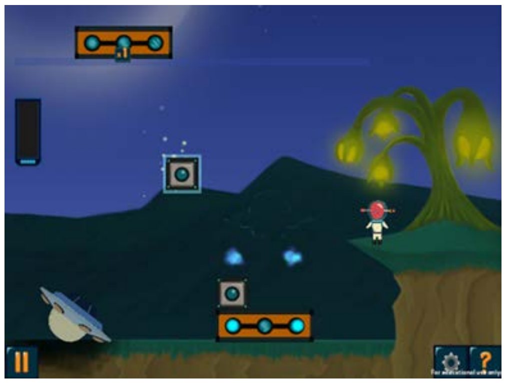


Figure 1. An example level from *RumbleBlocks*. The alien is stranded on the cliff and players must build a tower which is tall enough to reach him while also covering all blue "energy balls" to power his spaceship.

designs they can create. Each level in *RumbleBlocks* is designed to emphasize a particular principle of structural stability, and thus has a particular solution that was envisioned by the designers. However, students are not required to use it, and it is even possible that students may find a solution that is better than the one that the designer envisioned. Throughout development, the designers formed an intuition for the different groups of answer types being used by students, but they lacked methods for understanding how similar two answers were, and how many different answers were possible for each level. While it would have been possible to render all student solutions into screenshots, it would have been infeasible to manually comb through the thousands of towers generated by students.

To address this issue of understanding the kinds of solutions students are using, we have developed a method for extracting the conceptual features of game states in *RumbleBlocks* utilizing a two-dimensional context-free grammar. These features allow the designers and researchers of *RumbleBlocks* to examine the different sub-patterns that players are using in building their towers. The conceptual features can be used as a way of comparing different towers and evaluating how often students produce the answer which designers expected. It also enables us to zero in on the solutions they did not expect. To demonstrate the utility of these features we perform a clustering analysis, which assigns towers to groups, which correspond to the different unique solution that are possible on each level of the game. Designers can use this analysis to better understand the space of student solutions.

2. CONCEPTUAL FEATURE EXTRACTION

The first challenge in using *RumbleBlocks* data, or any educational game data, is to convert it into a form that is amenable to analysis. This task is not easy because a single state, or tower, in *RumbleBlocks* is both continuous and two-dimensional. Previous work has used an empirical measure of symmetry, width-of-base, and center-of-mass (human selected features) to describe a tower and has shown that these features can be predictive of student outcomes [9]. These features give a useful abstract evaluation of students' solutions; however, they are not descriptive enough to provide insight regarding specific patterns in student solutions. Without a more detailed description, it is hard for a designer to understand where new interventions need to be implemented to better facilitate student learning. In this work, we seek to remedy this problem by automatically extracting fine-grained conceptual features using unsupervised learning directly from two-dimensional descriptions of towers. These features allow us to investigate the solution space at a higher level of detail.

Our conceptual feature extraction process takes as input log files from all student play sessions and outputs all student towers as feature vectors that represent the towers' structural components. The process consists of the four steps illustrated in Figure 2 and discussed in turn in the next sections. First, we discretize the representations of all towers in the raw log files using a two-dimensional grid. Second, we generate grammatical rules based on the discretized representations using a novel algorithm of our own design: the Exhaustive Rule Generator (ERG), which induces a two-dimensional grammar returning an exhaustive set of rules capable of parsing the entire set. Third, the discrete representations are parsed using the rules generated by ERG, which returns a set of parse trees describing each tower in a hierarchical fashion. Finally, we process the parse trees to generate a set of feature vectors that denote which concepts from the grammar are present within each tower.

2.1 Discretization

The first step in the conceptual feature extraction process is discretization, or gathering meaningful data from the logs and converting it from a continuous two-dimensional space into a discrete two-dimensional space. The input to this step is the raw student log data, which contains action-by-action traces of student play sessions at replay fidelity. The logs generated by *RumbleBlocks* are intended to be post-processed through a replay analysis engine [9] which allows researchers to play logs back through an active instance of the game engine in order to extract information from live game states. Using this approach we are able to access information on individual game objects, such as collision information or bounding box dimensions, without having to log everything at the time of play. Since the logs are being replayed within the same game engine, the replayed game states are consistent with what students experienced.

To convert the continuous data from *RumbleBlocks* into discrete data we utilized a binning process. To bin a tower, the coordinates of the extents of each block's bounding box (the smallest rectangle which can be drawn around the block, a property accessible in the active game state) are translated such that the bottom left corner of the tower is at position (0,0). After translation, all of the edge coordinates of each block are divided by the size of the

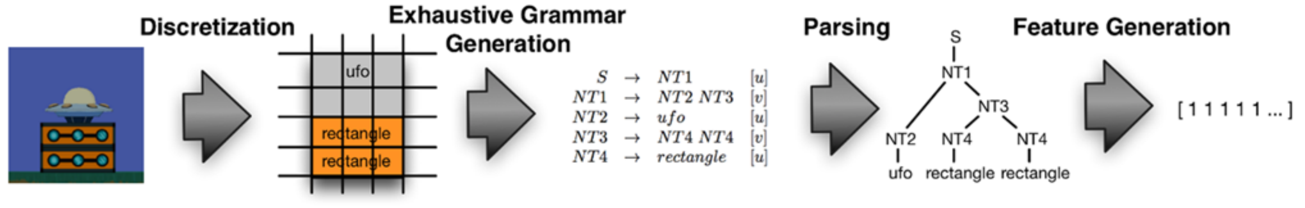


Figure 2. The Conceptual Feature Extraction Process.

smallest block (a square), creating a unit grid. Finally, the edges of blocks are rounded to their nearest integer positions (e.g., an x position of 1.6 would be rounded to 2), in effect "snapping" blocks to grid positions, which helps to ensure that clear divisions can be drawn between blocks because some blocks are slightly out of alignment. After binning, we output the discretized towers as a set of blocks described by their type and converted left, right, top and bottom values. The block type is the concatenation of the original block's shape (cube, rectangle etc.) and its rotation about the z-axis rounded to the nearest 15 degrees (for example the "rectangle" block with a 90 degree angle would now have "rectangle90" as its type). Thus, the final tower is discrete and comprised of blocks binned to a unit grid.

2.2 Exhaustive Rule Generation (ERG)

Once all of the student log data has been converted into discretized towers, we can automatically generate features describing the spatial aspects of these towers using two-dimensional context-free grammars. These grammars, which have been used to perceive structure in pictures, are an extension of 1D grammars for strings [4]. The grammar used in our approach are simplification of probabilistic two-dimensional context-free grammars, which have been used in previous work to teach an artificial agent to learn to perceive tutor interfaces [16]. Our approach is slightly different than this previous work in that we do not need to choose a single best parse of a tower but instead want to extract all of the spatial features present in the tower. This makes the rule probabilities from [16] unnecessary and so we omit them. Additionally, the towers in the *RumbleBlocks* task are much more complicated than the grid layout of the tutoring system interfaces explored in the previous work. Despite these differences, the spirit of our work is the same. We are using context-free grammars to perform representation learning.

Before explaining how we automatically generate a grammar we give a description of how they are structured. A two-dimensional context-free grammar is represented by a 4-tuple $G = <S,V,E,R>$. *S* is the start symbol, which in our case represents the concept of a complete tower. *V* represents the set of nonterminal symbols, which represent the structural components of a tower. In the trivial case these nonterminals represent terminals, i.e. individual blocks or space, but more complicated nonterminals represent intermediate structures, e.g. a pair of blocks stacked on one another, or even entire towers. *E* is the set of terminal symbols, which in our task represent the blocks and filler space. Finally, *R* is the set of rules, which describe how nonterminal symbols can decompose into other terminal and nonterminal symbols, as well as the direction (vertical, horizontal, or unary) in which they decompose. Because our rules capture the relative positions between blocks and the spaces between them (vertically and horizontally adjacent), we do not need to store position information. To clarify, our rules have the following form:

NT → *BOTTOM TOP* [*vertical*]

NT → *LEFT RIGHT* [*horizontal*]

NT → *block* [*unary*]

Where *NT*, *BOTTOM*, *TOP*, *LEFT*, and *RIGHT* are nonterminal symbols, i.e. $\in V$, and *block* is a terminal symbol, i.e. $\in E$. The *vertical* rule can be used to parse the two structures, *BOTTOM* and *TOP*, into the *NT* structure if they are vertically adjacent, horizontally aligned (the values of their left extents and right extents are equal), have equal width, and if the *BOTTOM* structure is below the *TOP* structure. Similarly, the *horizontal* rule can be used to parse the two structures, *LEFT* and *RIGHT*, into the *NT* structure if they are horizontally adjacent, vertically aligned, have equal height, and if the *LEFT* structure is to the left of the *RIGHT* structure. Finally, the *unary* rule allows the *block* symbol to be parsed into *NT*; no additional constraints apply for unary rules. We utilize Chomsky Normal Form to represent our rules because it allows for polynomial time parsing using the CKY algorithm [6], so every nonterminal decomposes into a pair of nonterminals or a single terminal symbol. Note that for convenience, we also have unary start rules that point to nonterminals representing entire towers. Even though this is in violation of Chomsky Normal Form, we only have these special rules at the top-most level so it does not have an effect on parsing complexity. See Figure 3 for an example of how a grammar can be used to parse a tower.

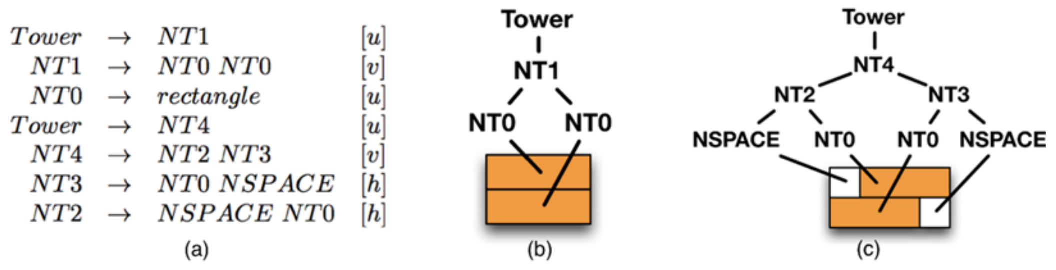
Before we can parse discretized *RumbleBlocks* towers we need to generate a grammar capable of parsing the set of towers. One difficulty is that most towers are not initially parsable because their blocks don't align cleanly, which is needed for matching vertical and horizontal grammar rules. To deal with the problem that not all towers are completely rectangular in shape, we introduce a new '*space*' terminal symbol that has unit size, i.e. takes up one grid cell, and fill in all of the negative space in a tower with these symbols.

While introducing '*space*' symbols enables us to parse towers that have space in them, it also causes an additional problem. First, we plan on automatically generating new nonterminals for blocks that are adjacent to one another. Because there are so many ways to pair up '*space*' symbols we end up bloating the grammar with unnecessary nonterminals that all reduce to space. Furthermore, this explosive number of nonterminal symbols also pair up with meaningful block symbols causing the grammar to grow even larger. To prevent grammar bloat we seed our initial grammar with the following recursive space rules:

NSPACE → *space* [*unary*]

NSPACE → *NSPACE NSPACE* [*vertical*]

NSPACE → *NSPACE NSPACE* [*horizontal*]



$$\begin{array}{lcll} Tower & \to & NT1 & [u] \\ NT1 & \to & NT0\ NT0 & [v] \\ NT0 & \to & rectangle & [u] \\ Tower & \to & NT4 & [u] \\ NT4 & \to & NT2\ NT3 & [v] \\ NT3 & \to & NT0\ NSPACE & [h] \\ NT2 & \to & NSPACE\ NT0 & [h] \end{array}$$

Figure 3. An example of how grammar (a) can be used to describe towers (b and c). The extra space and alignment rules of the grammar are omitted for clarity.

We also ensure that no additional nonterminals that reduce solely to space are introduced during automatic grammar generation.

Once we augment the towers with '*space*' symbols we use the novel Exhaustive Rule Generator (ERG) Algorithm (see Algorithm 1) to recursively generate a nonterminal for every pair of adjacent structures. The input to the algorithm is a set of towers and a start symbol. The algorithm starts by creating an empty grammar (seeded with recursive space rules), adds terminals for all of the blocks and the space symbol, creates an empty collection of remembered structures (used to ensure multiple nonterminals are not generated for the same structure), and iterates through the set of towers adding rules for each tower using the recursive *Rule-Gen* procedure.

The *Rule-Gen* procedure takes a single tower, a grammar, and a collection of remembered structures as inputs. It starts by checking if there is already a nonterminal that describes the tower, if such a nonterminal exists it is returned. Next, the algorithm checks if the tower only contains space, if so the algorithm returns the special *NSPACE* symbol (so any generated grammar integrates with the recursive space rules). If neither condition is met then a new nonterminal is generated with a unique name and added to the grammar. If the structure consists of a single terminal then a unary rule is added decomposing the new nonterminal into the terminal symbol. An entry in the hash table is created for that tower and the nonterminal is returned. If the structure contains more than one terminal, it is divided at each location (both horizontal and vertical) where the structure can be divided into two sub-structures (without splitting a terminal). For each division, the *Rule-Gen* procedure is called on the sub-structures and a rule is added mapping the new nonterminal to the nonterminals representing each sub-structure. The direction of this rule is determined by the direction of the division. After adding rules for all divisions, an entry is added to the collection mapping the structure to the new nonterminal and the nonterminal is returned.

```
Algorithm 1: EXHAUSTIVE-RULE-GENERATOR($Towers, S$)

procedure RULE-GEN($Structure, Grammar, Remembered$)
  if $Structure \in Remembered$
    then return ($Remembered[Structure]$)

  if IS-ONLY-SPACE($Structure$)
    then return ($'NSPACE'$)

  $Head \leftarrow$ UNIQUE-NONTERMINAL-NAME()
  ADD-NONTERMINAL($Head, Grammar$)

  if IS-TERMINAL($Structure, Grammar$)
    then  $Terminal \leftarrow$ GET-TERMINAL($Structure, Grammar$)
          ADD-RULE(($Head \rightarrow Terminal[unary]$), $Grammar$)
          $Remembered[Structure] \leftarrow Head$
          return ($Head$)

  for each ($SubA, SubB, Direction$) $\in$ ALL-DIVISIONS($Structure$)
    do  $NtA \leftarrow$ RULE-GEN($SubA, Grammar, Remembered$)
        $NtB \leftarrow$ RULE-GEN($SubB, Grammar, Remembered$)
        ADD-RULE($Grammar$, ($Head \rightarrow NtA, NtB[Direction]$))
  $Remembered[Structure] \leftarrow Head$
  return ($Head$)

main
  $Grammar \leftarrow$ EMPTY-GRAMMAR()
  ADD-TERMINALS($Towers, Grammar$)
  $Remembered \leftarrow$ EMPTY-COLLECTION()
  for $Tower \in Towers$
    do  $Nt \leftarrow$ RULE-GEN($Tower, Grammar, Remembered$)
        ADD-RULE(($S \rightarrow Nt[unary]$), $Grammar$)
        return ($Grammar$)
```

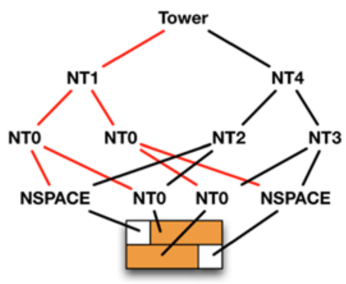


Figure 4. The two possible parses of tower *(c)* after alignment rules are added. Notice that the rules in the red tree are now similar to the rules in tower *(b)*'s parse tree.

The result of the ERG algorithm is a grammar that contains a nonterminal for every structure present in the set of towers. However, one subtle problem remains. Two towers that are nearly similar, but are unaligned and consequently have an additional '*space*' somewhere in the tower end up sharing no intermediate nonterminal symbols in their parses, see the differences between towers *(b)* and *(c)* in Figure 3. This is a problem because we are using nonterminals to model spatial features common across towers. To counter this effect, we introduce a set of "alignment rules" for every nonterminal *NT* in our grammar:

$$NT \rightarrow NT\ NSPACE\ [vertical]$$

$$NT \rightarrow NSPACE\ NT\ [horizontal]$$

$$NT \rightarrow NT\ NSPACE\ [horizontal]$$

These rules triple the number of grammar rules, but add additional parses to towers so that they share common structure with other similar but differently aligned towers, see Figure 4. We have two horizontal rules so that we can have additional space on the left and right of a symbol, but we only have one vertical rule because we can have additional negative space on the top of a block, but not on the bottom, because blocks in *RumbleBlocks* are subject to gravity and any space below a block would be filled by the block falling into a new position. It is important to note that while these rules enable the towers to share similar structure, it does not give them identical parses. This enables us to relate similar structures using their parse trees without having to worry about truly different towers being lumped together.

2.3 Parsing

After generating a grammar, we can use it to parse the towers and determine all of the nonterminal symbols that can be derived from each tower. We use a modified version of the CKY algorithm [6] that functions over two dimensions instead of one. This algorithm, which utilizes dynamic programming, is an approach to bottom-up parsing in polynomial time. One feature of the CKY algorithm is that the amount of time required to compute all parses of a tower is the same as the amount of time required to compute one parse. Using this approach, we produce all of the parses for every tower in our set.

2.4 Feature Vector Generation

Once we have all of the parse trees, we convert them into feature vectors. This converted format is useful because the vector representation is more concise and easier to manipulate when doing analyses. To create a feature vector we create a one-dimensional vector with an integer value for every nonterminal in the grammar. These values are initialized to be 0 but are set to 1 for every nonterminal that appears in at least one of a given tower's parse trees, similar to previous work [17]. Thus, a feature vector is a concise description of all the structures that are present in all of the parses of a given tower. Once we have generated these feature vectors, we can use them to perform a variety of analyses as we will demonstrate next.

3. Data

The data we present here comes from a large formative evaluation of *RumbleBlocks*, which was performed in two local area elementary schools. The sample includes play sessions from 174 students from grades K-3 (5-8 years old) who played the game for a total of 40 min across 2 sessions. The game contained 39 different levels, each intended to target a specific principle of stability through the use of the energy balls as scaffolding. Players played an average of 17.8 unique levels (σ =7.2), as not all students completed the entire game. Additionally, because students are allowed to retry levels in which they fail, the data can contain multiple attempts by a student on each level (μ =1.24, σ =.68). In total, the dataset contains 6317 unique structures created by students.

Due to constraints of the conceptual feature extraction process some data had to be excluded from analysis. The parsing process requires that blocks be aligned to a grid such that clear separations can be drawn between them—because of this it was necessary to omit any structures where the binning process caused blocks to overlap the same grid cell (less than 0.2% of data). Additionally, rotating a block will sometimes cause its bounding box to intersect with adjacent grid cells, because the bounding box expands to encompass the maximum left, right, top, and bottom values of the block's geometry rather than rotating with it. To address these issues of grid overlap we exclude any record that contained blocks whose dimensions intersected or any blocks whose z-axis rotation was not a multiple of 90, after rounding to the nearest 15 degrees. Overall these constraints exclude ~3.5% of our sample.

The final grammar generated from the dataset by the ERG algorithm contains 13 terminals, 6,010 nonterminals, and 30,923 rules. Each nonterminal was used an average of 50.59 times (σ =240.2) across all towers. The average number of levels in which a given nonterminal was used was 3.09 (σ =4.14). The average number of nonterminals per towers was 49.96 (σ =40.23). Reporting statistics on the number of nonterminals within an average parse or number of parses within an average tower is complicated by the inclusion of alignment rules which add some arbitrary number of parses to each tower.

4. CLUSTER ANALYSIS

In order to demonstrate the utility of these conceptual features to guide the design process in educational games, we performed a clustering analysis of student solutions in *RumbleBlocks*, to discern how many solutions students were demonstrating. Clustering takes a series of data points, in our cases represented by conceptual feature vectors, and assigns them to groups based on how similar the points are. Clustering similar to ours has been used by Andersen and Liu et al. to group game states as a way of exploring common paths that players take through a game [18]. Our approach differs from theirs in that our features are machine learned rather than defined by designers. This allows us to observe emergent patterns in play without biasing the results with human input.

4.1 Method

As we were interested in what kinds of solutions students were using on each level, we performed clustering of solutions on a level-by-level basis, which will yield groups of similar student solutions. Within each level we utilized the k-means clustering algorithm (we use the scikit-learn implementation [22]). This algorithm takes as input a set of data and a parameter k, where each datum is described by an n-dimensional vector and k specifies the number of desired clusters. The output is a set of labels assigning each datum to a particular cluster. The algorithm works by using the k-means++ approach [3] to select initial centroids for the clusters such that they are generally distant from one another. This initialization algorithm guarantees that the solution found will be $O(log\ k)$ competitive to the optimal solution. Given the initial centroid positions, the data points are then assigned to the clusters based on which centroid they are nearest to, as measured by the Euclidian distance between the n-dimensional vectors of the point and the centroid. Once the points are assigned, the positions of the centroids are updated relative to the points they encompass. This process (also called hard expectation-maximization) is then repeated until quiescence. Although the worst-case running time is known to be super polynomial in the size of the input, in practice the algorithm finds solutions reasonably quickly [2]. For a given run of k-means we repeat this process 10 times and select the model that has the best fit to the data, which is measured by the within cluster sum squared distance from every point to its centroid. Running the algorithm multiple times helps to avoid local maximums and accounts for the inherent non-deterministic nature of the algorithm.

As we are also interested in how many solutions are present in the data, not just which solutions are similar, we therefore must determine the correct number of clusters to use, in essence choosing a good value for k. To identify the number of clusters present in the data, we use the G-means algorithm, which acts as a wrapper around the k-means algorithm [8]. This approach starts by running k-means on the entire dataset with k initialized to 1. The algorithm then takes the clusters of points returned by the previous k-means and attempts to divide each of them into two further sub-clusters, again using k-means with k=2. A vector is then drawn between the two new sub-clusters' centroids, which represents the dimension over which the two clusters are separated. The algorithm then projects all the points from both sub-clusters onto this single dimension of separation and checks to see if they have a Gaussian distribution using the Anderson-Darling statistic (with $p < 0.01$). If the distribution is found to not be Gaussian, the original value of k is incremented and the process is repeated for all clusters. Once all of the clusters are found to have a Gaussian distribution, the final k value is returned, representing a good number of groups in the dataset. This approach has been shown to be more effective than BIC at deciding the correct value for k [8]. Because k-means returns different clusters on different runs, we run the G-means algorithm 10 times and return the mode k value as the most likely value for k.

Before using the machine clustering to conduct analyses, we must first ensure that it is creating reasonable clusters. As a test of the validity of the clusters, we had two independent coders hand cluster three levels to generate a gold standard with which to compare the machine clustering results ($\kappa = 0.88$). Additionally, we want to evaluate the effectiveness of our approach by comparing it to a naïve method of automatic grouping. The naïve method we used was to group the towers by direct equability, i.e. assigning all towers that have identical discrete representations to the same group. This allows us to see how much closer our approach gets to human results than a naïve machine approach.

The selected three levels were chosen because they were part of an in-game counterbalanced pre-posttest, which did not use the

Table 1. Clustering measures (completeness, homogeneity, v-measure, and adjusted rand index) means and standard deviations after 10 iterations of clustering. Note that equality clustering is constant and so has no standard deviation.

Level	Comparison	Completeness (SD)	Homogeneity (SD)	V-Measure (SD)	Adj. Rand Index (SD)
com_11_noCheck (n=251)	k-means	.74 (.06)	.57 (.10)	.63 (.04)	.51 (.08)
	equality	.55 (NA)	.99 (NA)	.71 (NA)	.23 (NA)
s_13_noCheck (n=249)	k-means	.83 (.02)	.63 (.04)	.72 (.02)	.47 (.04)
	equality	.60 (NA)	.99 (NA)	.75 (NA)	.16 (NA)
wb_03_noCheck (n=254)	k-means	.63 (.02)	.80 (.02)	.71 (.02)	.42 (.02)
	equality	.53 (NA)	.99 (NA)	.69 (NA)	.28 (NA)

energy ball mechanic, making them less constrained and likely to have more variable answers, and therefore pose a greater challenge in terms of accurate clustering. Additionally, because all students were required to play them as part of the pre-post design, these levels have some of the largest sample sizes of all levels.

In comparing different clusterings we report the completeness, homogeneity, V-Measure [24], and Adjusted Rand Index (ARI) [23] on these three levels for machine clustering and direct equality using human clustering as a gold standard. These measures each evaluate different aspects of clustering results and are standard metrics of clustering quality. The Completeness score measures how well records in the same class are clustered together, i.e., how well the clustering put items that should be together in the same group. The Homogeneity score measures how well records that are different are separated, i.e. when the elements within a given cluster are all the same. Because these measures are in opposition to each other, we report the V-measure, which gives a harmonic balance between the completeness and homogeneity scores. Finally, ARI is a measure of clustering accuracy adjusted for chance. The measure has a range of [-1, 1] and approaches 0 when guessing.

After testing the clustering on a subset of hand-coded levels, we also wanted to gauge the validity of the approach on all levels. To measure validity we make the assumption that if two towers are highly similar they are also likely to both stand or fall in the earthquake, though some noise is to be expected due to indeterminacies in the game's physics engine. Taking this assumption, we can again use homogeneity as a way of calculating how consistent the success/failure designation is within a cluster. Comparing the homogeneity scores of the machine clustering and the random clustering of the towers (using the same number of clusters as determined by G-means) can tell us if the machine clustering is significantly better than that expected by chance. This metric can be interpreted as a sanity check to ensure that the clustering is actually working on levels that have not been hand labeled.

After evaluating clustering validity, we can use clustering to get a sense of how often players are using designer envisioned solutions. In designing the levels, the game designers tried to make each level focus on one of the three targeted principles of stability (low center of mass, wide base, symmetry). That is, the designers' intention is that on each level, the configuration of the energy dots and the block inventory, are such that the student is led to a solution that exemplifies the particular principle targeted at that level. It is fine, and probably desirable, if levels allows for multiple (and unforeseen) solutions. However, what we hope to avoid is levels that have a large number of unforeseen solutions that do *not* address the particular principle that the level is intended to target.

To perform this alignment analysis we had one of the designers of *RumbleBlocks* generate a play session log that represented the "answer key" for each level. We then determined which of the clusters the intended solution would be grouped into on each level and compared the number of towers in that group to the total number of towers for that level. This information can help us get a sense of the alignment between what designer expectant students to do and what players actually do. Having this information can help the designers know where to focus future redesign efforts to best target discrepancies.

4.2 Results

When looking at the measures of clustering effectiveness in Table 1 we see that the k-means algorithm was able to outperform straight equality grouping in ARI and completeness. This can be interpreted to mean that k-means clustering is making a higher percentage of correct decisions in grouping structures, suggesting that the results of clustering can be validly used in further analysis. In all instances, the equality grouping performs better than k-means clustering in homogeneity score because if direct equality is used to assign group labels the resulting groups will be, by definition, perfectly homogeneous. In many instances, this causes the V-measure to also be better because V-measure evenly weights for completeness and homogeneity. Overall these results can be interpreted to mean that clustering along conceptual features of towers provides reasonable grouping accuracy when compared to human clustering.

When clustering was performed across all levels, the mean homogeneity of the k-means clusters was found to be significantly greater than the homogeneity from random grouping of student solutions using a two-sample t-test (p < .001). Assuming that similar towers would stand or fall together, this further supports the idea that the clustering algorithm is not separating similar student solutions.

Overall the clustering algorithm generated an average of 8 clusters per level (σ = 3.98), compared to the average number of groups as determined by equality grouping 56 (σ = 45.77). The smallest number of clusters (2) was seen in the tutorial level, which contains only 1 block and the spaceship allowing for very little difference between solutions. The highest number of clusters (17) was found in a later level (centerOfMass_07) which contains 5 larger blocks and 6 energy balls allowing for nuanced differences in solution styles.

Our analysis of what percentage of solutions appear similar to the designers' intended solutions shows a high degree of variability, see Figure 5. Some levels, like the tutorial and other earlier levels, are found near the higher end of the spectrum because as introductory levels they do not allow for a large number of solutions. However, the levels on the lower end of the spectrum indicate that few students actually created the towers envisioned by the designers. These levels warrant a closer investigation to ascertain what other kinds of solutions students are producing. For example, upon further inspection of the solutions to centerOfMass_07, designed to target the principle of low center of mass, we discovered

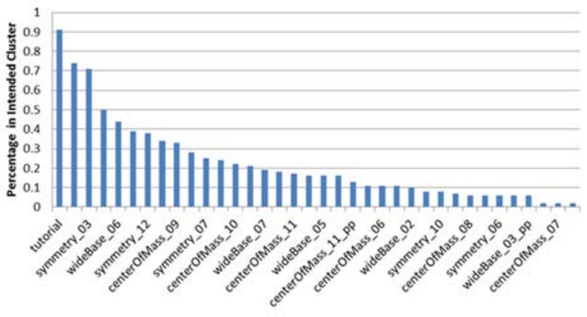


Figure 5. Percentage of use of the envisioned solution on a level for each level.

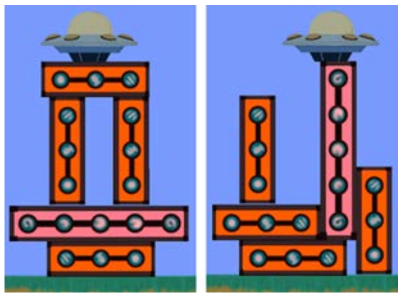


Figure 6. An example of mismatch with designer expectation and student solution from the centerOfMass_07 level. The designer's answer is on the left.

that a large number of student solutions that did not typify the level's key principle (See Figure 6). While a number of these solutions did not actually survive the earthquake the variety of atypical solutions points to the need for more guidance. In further iterations designers should focus their efforts on these levels to consider whether students need more scaffolding.

5. DISCUSSION

In this paper we have described a process for conceptual feature extraction from logs of gameplay in an educational game. The process follows four steps starting with the raw student log files. The files are discretized and then used to generate a two-dimensional context-free grammar that can be used to parse the towers and yield a vector of features present in the tower. We demonstrated how conceptual features could be used to perform a clustering analysis of common student solutions.

While the results we discussed are specific to *RumbleBlocks* aspects of our approach could be generalized to other games or educational technology environments by altering some of the steps in the process. One example of another game this approach would work for is *Refraction,* which has players redirecting laser beams around a grid based board by placing laser splitters to make proper fractions [1,18]. This game already takes place on a grid and so would not require a discretization step, but the other steps would be applicable. In this game, our approach would learn features corresponding to patterns of laser splitters on the grid, which could be used to generate feature vectors for each student solution and to cluster these feature vectors. These clusters would be similar to those generated by Liu *et al.* [18] but the features would be automatically generated rather than human tagged.

When applying our approach more generally, the discretization step will always be specific to a particular game or interface, as it requires an intimate knowledge of the context. Employing a replay analysis engine can assist with discretization by providing a standard format [9]. The ERG algorithm is applicable to any discrete two-dimensional representation of structure in which adjacency relations are meaningful. Converting parses into feature vectors for analysis is a technique that should be applicable to most situations.

The features generated with this method can be used by many different kinds of analyses beyond what we present here. For instance, the feature vectors could be used as a way to represent game data in a format suitable for DataShop [13], a large open repository of educational technology interaction data. A feature vector is analogous to the state of a tutoring system interface and the changes in the feature vector from step to step correspond to the student actions. Additionally, virtual agents, such as SimStudent [20], could use this data representation as a way of understanding and interacting with educational games, enabling us to model student learning in these contexts.

While the grammars extracted by our method have proven to be useful, they still have some limitations, such as an inability to represent towers that cannot be cleanly mapped to a grid or which contain overlapping or angled substructures. Making the grammar more descriptive would require the relaxing of constraints concerning how nonterminals can be parsed, e.g., not requiring strict alignment. Another issue has to do with how many different nonterminals map to nearly equivalent structures. Even though we attempt to minimize this by introducing the alignment and space rules, there are still cases where further reductions could be implemented. One potential solution, to address this problem in general, is to implement model merging to condense pairs of nonterminals that represent similar concepts into single nonterminals [15]. The ability to merge similar nonterminals is a promising direction for future work.

In addition to being able to describe more towers, model merging would also allow the generalization of grammars to cases we have not seen. Because context-free grammars can be used generatively, the generalized grammar could be used to produce novel towers, similar to the work of Talton *et al.* [26]. In our case, these novel towers would give insight into the as-yet-unseen portions of the solution space. Furthermore, the novel towers could be used as templates in creating new levels. In future work we will be exploring ways to feed this information, and information from clustering, directly back into the game development environment.

The clustering results not only provide the designers of *RumbleBlocks* with a picture of how students are playing their game, they also possess further uses beyond assisting design iteration, such as exploring research questions. One potential use of the clustering is as an empirical measure of how "open" a particular level is, by counting how many different clusters, i.e. different solutions, that level affords. Using this measure allows researchers to explore the interactions of openness with learning and engagement. Exploring this interpretation of the clustering results will be a part of our ongoing analysis of *RumbleBlocks.*

Another intriguing direction for future work would be to explore the relationship between the conceptual features and the knowledge components [14] used in building towers in *RumbleBlocks*. There may exist a mapping between the substructures used in towers and the conceptual knowledge components related to stable structures. Exploring this would require measurements of how a student's use of particular structures changed over time and how it relates to task performance. If such a mapping exists, then our approach would not only be useful for automated feature extraction, but also for automatically building models of conceptual knowledge components.

6. CONCLUSION

Framing game experiences in terms of conceptual features can help both designers and researchers better understand how students interact with their games. The main contribution of this paper is an approach for extracting conceptual features from play logs within educational games and using these features to perform clustering of student solutions. Designers can use the clusterings to better understand the space of student solutions and to know where to focus their attention to improve student learning experiences. Ultimately we envision feeding back this clustering information directly into the game design platform. This information can also enable researchers to explore important questions, such as how "openness" and difficulty relate to student engagement. While our approach was created with the specific two-dimensional world of *RumbleBlocks* in mind, it should be generalizable, and we hope others will find it useful in exploring other educational games.

7. ACKNOWLEDGMENTS

We would like to thank the designers of *RumbleBlocks* and our colleagues who conducted the formative evaluation that yielded our data. This work was supported in part by a Graduate Training Grant awarded to Carnegie Mellon University by the Department of Education #R305B090023 and the DARPA ENGAGE research program under ONR Contract Number N00014-12-C-0284.

8. REFERENCES

[1] Andersen, E., Liu, Y., Apter, E., Boucher-genesse, F., and Popovi, Z. Gameplay Analysis through State Projection. *Proc. FDG '10*, (2010).

[2] Arthur, D. and Vassilvitskii, S. How slow is the k -means method? *Proc. SCG '06*, ACM Press (2006), 144.

[3] Arthur, D. and Vassilvitskii, S. K-means++: The Advantages of Careful Seeding. *Proc. ACM-SIAM*, (2007), 1027–1035.

[4] Cherubini, A. and Pradella, M. Picture Languages: From Wang Tiles to 2D Grammars. In S. Bozapalidis and G. Rahonis, eds., *Algebraic Informatics*. Springer, Berlin, Germany, 2009, 13–46.

[5] Christel, M.G., Stevens, S.M., Maher, B.S., et al. RumbleBlocks: Teaching Science Concepts to Young Children through a Unity Game. *Proc. CGames 2012*, (2012), 162–166.

[6] Cocke, J. *Programming Languages and their Compilers: Preliminary Notes*. New York University, 1969.

[7] Gee, J.P. *What video games have to teach us about learning and literacy*. Palgrave Macmillan, New York, 2003.

[8] Hamerly, G. and Elkan, C. Learning the k in k-means. *Proc. NIPS '03*, (2003).

[9] Harpstead, E., Myers, B.A., and Aleven, V. In Search of Learning : Facilitating Data Analysis in Educational Games. *Proc. CHI '13*, (2013), 79–88.

[10] Hunicke, R., Leblanc, M., and Zubek, R. MDA : A Formal Approach to Game Design and Game Research. *Proc. of the AAAI Workshop on Challenges in Game AI*, (2004), 1–5.

[11] De Jong, T. and Van Joolingen, W.R. Scientific Discovery Learning with Computer Simulations of Conceptual Domains. *Review of Educational Research 68*, 2 (1998), 179–201.

[12] Ketelhut, D.J. The Impact of Student Self-efficacy on Scientific Inquiry Skills: An Exploratory Investigation in River City, a Multi-user Virtual Environment. *Journal of Science Education and Technology 16*, 1 (2006), 99–111.

[13] Koedinger, K.R., Baker, R.S.J. d, Cunningham, K., Skogsholm, A., Leber, B., and Stamper, J. A Data Repository for the EDM community: The PSLC DataShop. In C. Romero, S. Ventura, M. Pechenizkiy and R.S.J. d. Baker, eds., *Handbook of Educational Data Mining*. 2010, 43–55.

[14] Koedinger, K.R., Corbett, A.T., and Perfetti, C. The Knowledge-Learning-Instruction Framework: Bridging the Science-Practice Chasm to Enhance Robust Student Learning. *Cognitive Science 36*, 5 (2012), 757–98.

[15] Langley, P. *Simplicity and Representation Change in Grammar Induction*. 1995.

[16] Li, N., Cohen, W.W., and Koedinger, K.R. Learning to Perceive Two-Dimensional Displays Using Probabilistic Grammars. *LNCS 7524*, (2012), 773–788.

[17] Li, N., Schreiber, A., Cohen, W.W., and Koedinger, K.R. Creating Features from a Learned Grammar in a Simulated Student. *Proc. ECAI '12*, (2012).

[18] Liu, Y., Andersen, E., Snider, R., Cooper, S., and Popovi, Z. Feature-Based Projections for Effective Playtrace Analysis. *Proc. FDG '11*, (2011), 69–76.

[19] Lynch, C., Ashley, K.D., Pinkwart, N., and Aleven, V. Concepts , Structures , and Goals : Redefining Ill-Definedness. *International Journal of Artificial Intelligence in Education 19*, (2009), 253–266.

[20] Matsuda, N., Cohen, W.W., Sewall, J., Lacerda, G., and Koedinger, K.R. Evaluating a Simulated Student Using a Real Student's Data for Training and Testing. *LNAI 4511*, (2007), 107–116.

[21] National Research Council. *A Framework for K-12 Science Education: Practices, Crosscutting Concepts, and Core Ideas*. The National Academies Press, 2012.

[22] Pedregosa, F., Varoquaux, G., Gramfort, A., et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research 12*, (2011), 2825–2830.

[23] Rand, W. Objective Criteria for the Evaluation of Clustering Methods. *Journal of American Statistical Association 66*, 336 (1971), 846–850.

[24] Rosenburg, A. and Hirschber, J. V-Measure: A Conditional Entropy-based External Cluster Evaluation Measure. *Proc. EMNLP-CoNLL '07*, (2007), 410–420.

[25] Spring, F. and Pellegrino, J.W. The Challenge of Assessing Learning in Open Games : HORTUS as a Case Study. *Proc. GLS 8.0*, (2011), 200–208.

[26] Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N., and Měch, R. Learning design patterns with bayesian grammar induction. *Proc. UIST '12*, (2012), 63–74.