The Puzzle Forecast: Tutorial Analytics Predict Trial and Error

Dennis Vet
dennis.vet@student.uva.nl
University of Amsterdam
Amsterdam, The Netherlands

Riemer van Rozen
rozen@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

ABSTRACT

Puzzle tutorials are designed to teach puzzle-solving skills. For game designers, the difficulty is predicting if puzzle challenges will present players with opportunities for learning with trial and error. We aim to empower designers with tools and techniques for making those predictions by analyzing the goal chains inherent to good designs. We study PuzzleScript, an online game engine that has made the source code of high-quality puzzle tutorials available.

Research on puzzles has yielded algorithms that can generate playtraces of solutions. However, until now the importance of failure traces has been mostly overlooked. As a result, there is a lack of tools with analytics that can help assess challenge. To deliver them, we propose a novel approach that enriches playtraces with verbs.

We introduce TutoScript, a language for expressing goal chains in terms of verbs. By combining TutoScript with well-known search algorithms, and by mapping rules to verbs, TutoMate can enrich, analyze and visualize generated playtraces of solutions, failures and dead ends. Two case studies on Lime Rick and Block Faker demonstrate how it helps to analyze simple goal chains, and can also detect broken tutorials. Our solution takes a promising step towards generic techniques for analyzing and generating tutorials.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Integrated and visual development environments**; • **Applied computing → Computer games**.

KEYWORDS

automated game design, domain-specific languages, puzzle tutorials, verbs, skill atoms, analytics, learning, trial and error

ACM Reference Format:
Dennis Vet and Riemer van Rozen. 2024. The Puzzle Forecast: Tutorial Analytics Predict Trial and Error. In *Proceedings of the 19th International Conference on the Foundations of Digital Games (FDG 2024), May 21–24, 2024, Worcester, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3649921.3659854

1 INTRODUCTION

Puzzle-solving implies taking a series of actions, e.g., connecting puzzle pieces, navigating mazes, or solving riddles. Solving a puzzle requires skills, knowledge and insights about the rules.

Puzzle tutorials are designed to teach these skills. Each step, players progress towards a chain of goals that leads to identifying a solution [25]. Through structured progression and practice, players learn how puzzle mechanisms work. For game designers, the challenge is predicting if a puzzle tutorial will present players with the intended opportunities for learning through trial and error.

We aim to empower game designers with languages, techniques and tools that help automate these predictions. In particular, we study how Domain-Specific Languages (DSLs) can give such tools expressive power [28], e.g., for improving the predictive accuracy of analyses and enabling procedural content generation [29].

We study PuzzleScript, an online programming language and game engine created by Lavelle [19]. PuzzleScript expresses a wide variety of high-quality puzzle games and tutorials whose sources are publicly available [18]. We seize this rare research opportunity to empirically study the source code of well-designed puzzle tutorials.

Research on puzzles has yielded algorithms and techniques that can solve puzzles automatically, e.g., using SMT solvers [3], Answer Set Programming (ASP) [26], and Heuristics Search [20]. These can help to generate *playtraces*, or sequences of player actions. However, existing tools and techniques have largely overlooked failure traces, which are of critical importance for learning from mistakes.

Gameplay analytics offers metrics and visualizations to help gain insight into player data, e.g., to assess gameplay and learning [8]. In line with Koster's Theory of Fun [15], verbs have been identified as a key data point [4]. However, despite the availability of playtraces, these techniques have not yet been applied to tutorial design.

Unfortunately, no tool exist for verifying goal chains (or Skill Atoms) inherent to good tutorial designs [2, 5, 25]. As a result, designers lack an automated means for identifying ways players can learn from failure. They need analytics for measuring if tutorials timely introduce challenges, gradually increase the difficulty, and little by little present opportunities for learning. To deliver them, we propose a novel approach that enriches playtraces with verbs.

We introduce TutoScript, a textual and visual DSL for expressing goal chains in terms of verbs. We automate the analysis of goal chains for PuzzleScript by extending an existing framework for static analysis [12]. TutoMate adds an engine for generating playtraces, and analytics for comparing them against goal chains. By combining TutoScript with well-known search algorithms, and by mapping rules to verbs, TutoMate can enrich, analyze and visualize generated playtraces of solutions, failures and dead ends.

Two case studies on Lime Rick and Block Faker demonstrate how TutoMate recognizes simple goal chains. Of course, as a proof of concept, its analyses are still limited. Specifically, not every goal can be expressed using a simple 1-to-1 mapping between verbs and rules. Combinatorial explosions and sub-optimal algorithms also turn out to be bottlenecks. Despite these shortcomings, our results illustrate the approach is feasible for relatively small search spaces.

FDG 2024, May 21–24, 2024, Worcester, MA, USA

ACM ISBN 979-8-4007-0955-5/24/05
https://doi.org/10.1145/3649921.3659854

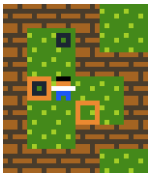
```
Background
lightgreen green
11111
01111
11101
11111
10111
```

(a) Object definition (b) Rendered object

Figure 1: A background object showing grassy patches

```
. = Background
# = Wall
P = Player
* = Crate
@ = Crate and Target
O = Target
```

```
####..
#.O#..
#..###
#@P..#
#..*.#
#..###
####..
```



(a) Legend (b) Level (c) Rendered level

Figure 2: Simple Block Pusher legend and level

Ultimately, work on language-parametric solutions can help analyzing and generating tutorials in general. The contributions of this paper are: 1) TutoScript, a DSL for expressing goal chains using verbs; and 2) TutoMate, a tutorial design tool for PuzzleScript for identifying ways players can learn through trial and error.

2 BACKGROUND

Automating the analysis and design of puzzle tutorials requires a thorough understanding of PuzzleScript. We perform design research [31], and we apply the Extract, Analyze, Synthesize (EASy) metaprogramming approach to study and create DSLs [14].

To begin, we analyze the problem space by performing a domain analysis [17]. Specifically, we analyze and reverse engineer the designs of high-quality puzzle tutorials. We formalize the extracted domain knowledge and create solutions in Sections 3 and 4. The case studies of Section 5 further detail and validate our approach. Here, we introduce PuzzleScript and summarize our problem analysis.

2.1 PuzzleScript

We introduce PuzzleScript using an example called Simple Block Pusher, one of many Sokoban-like games created by David Skinner.

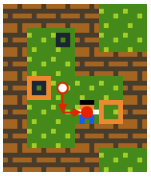
As the first game programmers see, its sources serve as a tutorial to learn the language. PuzzleScript programs consist of sequence of sections. This overview describes the essential ones. For more detailed explanations we refer to Lavelle [19] and Anthropy [1].

2.1.1 Objects. The **OBJECTS** section defines a series of game assets called objects, sprites of 5x5 pixels that can move and collide. The objects of the example are background, wall, player, crate and target. These objects appear in levels and are manipulated using rules.

Figure 1 shows its Background object. When the engine renders the object, this results in the visual sprite displayed in Figure 1b.

2.1.2 Collision layers. The game has three collision layers. This section specifies that Player, Wall and Crate objects can collide.

```
Background          (background layer)
Target              (support layer)
Player, Wall, Crate (foreground layer)
```



(a) F1. Get stuck (b) F2. Block path (c) S1. Store B and win

Figure 3: Two example failures and final steps of a solution

Table 1: Playtraces of failures and successes of one level

Name	Playtrace (w = walk, p = push)	Analysis
F1. Get stuck	$[w_\downarrow, p_\rightarrow]$	Trivial failure.
F2. Block path	$[w_\rightarrow, w_\rightarrow, w_\downarrow, p_\leftarrow]$	Simple blocked path.
S1. Solution	$[w_\downarrow, w_\leftarrow, p_\uparrow,$	Stash A to access B.
	$w_\rightarrow, w_\rightarrow, w_\downarrow, p_\leftarrow,$	Move B toward goal.
	$w_\uparrow, w_\leftarrow, w_\leftarrow, w_\downarrow, w_\downarrow, w_\rightarrow, p_\uparrow,$	Move B toward goal.
	$w_\leftarrow, w_\uparrow, w_\uparrow, p_\rightarrow$	Stash B to access A.
	$w_\uparrow, w_\uparrow, w_\leftarrow, p_\downarrow$	Store A.
	$w_\leftarrow, w_\downarrow, w_\downarrow, w_\rightarrow, w_\rightarrow, w_\uparrow, p_\leftarrow,$	Move B toward goal.
	$w_\downarrow, w_\leftarrow, p_\uparrow, p_\uparrow]$	Store B and win.

2.1.3 Legend. The **LEGEND** section of Figure 2a defines how symbols used in level descriptions can refer to objects to create levels.

2.1.4 Rules. PuzzleScript expresses game mechanics and run-time behaviors using rewrite rules. Players can interact with these rules using the arrow keys and the action key **x**. Simple Block Pusher has just one rule, which expresses pushing.

```
[ > Player | Crate ] -> [ > Player | > Crate ]
```

The rule's left hand side is a pattern describing the condition that must hold before applying the rule. We can read: *"if the player moves in the direction of a crate"*. Of course, a collision would normally prevent this movement. However, the rule's right hand side describes different result of the collision. We can read: *"then the player and the crate both move directionally"*. The omnidirectional **>** operator is short-hand for applying the rule in every direction.

2.1.5 Winconditions. The game has one win condition, which expresses that all targets must have a crate directly on top.

```
all Target on Crate
```

2.1.6 Example level. We will take a closer look at the game's first level, shown in Figure 2b. There are two crates, where crate A is on its target (**@**) and crate B is not (*****). When we run the game, the engine renders the level's start state as shown in Figure 2c.

2.2 Problem Analysis

Our problem analysis can be summarized as follows. Figure 4 illustrates the design process. The designer creates tutorials by formulating gameplay goals and programming in PuzzleScript.

2.2.1 Verbs. Designers formulate hypotheses about gameplay. For the example, a designer may have the following goals in mind.

G1 *Perceive setting.* The player occupies a top down maze.
G2 *Walk.* The player moves around using the arrow keys.
G3 *Push.* Crates can be pushed by colliding into them.
G4 *Store.* Pushing a crate onto a target stores it.
G5 *Get stuck.* Crates pushed into corners become stuck.
G6 *Block path.* Unreachable crates cannot be pushed.

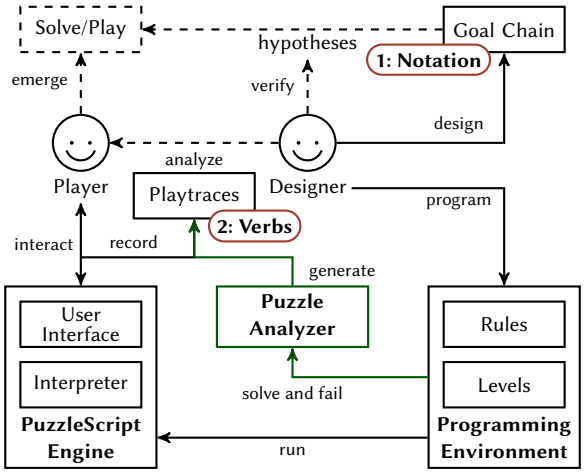


Figure 4: Tutorial design process in PuzzleScript

G7 *Stash.* Giving crates temporary spots helps create paths.
G8 *Win.* Every crate must be stored in order to win.
G9 *Strategize.* Solving a puzzle involves ordered stashing.

2.2.2 Playtesting. To assess if gameplay hypotheses hold, designers make observations and analyze how behaviors affect player experiences [24, 25]. Playtesting the example level reveals the following. Pressing the arrow keys moves the avatar (G2). Players can deduce they occupy a top-down dungeon (G1). When first pressing down and then right, the player observes movement (G2) and pushing (G3). However, Crate B is now stuck in a corner (G5), as shown in Figure 3a. Clearly, pushing a crate into a corner is a bad idea. Instead, the player could press right twice and then down (G2), to push Crate B to the left (G3). However, the path to push it north is now blocked (G6), as shown in in Figure 3b. Planning ahead (G9), the player could first stash Crate A (G7) before retrieving Crate B.

2.2.3 Goal chains. We reverse engineer the tutorial design by analyzing the dependencies between verbs. The goal chain of Simple Block Pusher is surprisingly complex. In Figure 5, arrows signify prerequisites. Strategizing about complex puzzles first requires understanding how to overcome obstacles by stashing crates. Players cannot make meaningful decisions without knowing how to win.

Automation is not yet possible because goal chains are not explicitly represented. Challenge 1 is creating a formal notation.

2.2.4 Playtraces. Designers can examine interaction patterns, or *playtraces*, sequences of player actions resulting in successes and failures. Table 1 shows three traces. Solving the puzzle (S1) entails combining the walk, push, store, stash and win mechanisms. Because failing is possible too, this requires a degree of strategizing.

Automation is not yet possible because (generated) playtraces consists of key presses only. Challenge 2 is adding the verbs.

2.2.5 Automation. Given the rules and a sequence of levels, the question is if the tutorial presents challenges in the intended order, gradually increases the difficulty, and offers opportunities for learning. For assessing the tutorial quality, designers need to investigate if the playtraces of successes and failures realize the goal chains.

Automating this analysis requires tackling two more challenges. Challenge 3 is leveraging existing algorithms for puzzle-solving

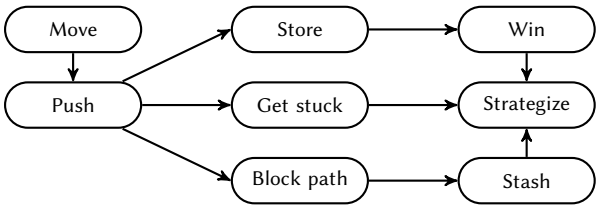


Figure 5: Simple Block Pusher: Goal Chain of Verbs

```
1  tutorial SimpleBlockPusher {
2    lesson 1: Storing {
3      "Winning requires storing crates."
4      introduce Move, Push, Store }
5
6    lesson 2: Stashing {
7      "Crates can be stashed to avoid blocking paths."
8      require Move, Push, Store
9      introduce Stash
10     avoid BlockPath }}
```

Figure 6: Simple Block Pusher: Tutorial design in TutoScript

to obtain generated playtraces. Challenge 4 is comparing the goal chains against generated playtraces enriched with verbs.

We propose a novel approach that tackles these challenges. Our solution consists of two parts. The first, TutoScript, addresses Challenge 1 by formalizing goal chains in Section 3. The second, TutoMate, addresses Challenges 2–4 by creating a tutorial design tool for PuzzleScript in Section 4. For assessing the quality of puzzle tutorials it offers analytics that work on generated playtraces.

3 TUTOSCRIPT

We introduce TutoScript, a DSL for expressing goal chains. We have already applied its visual notation in Figure 5. Here, we introduce a textual notation, which is specifically designed for puzzle game tutorials. Figure 6 shows an illustrative tutorial design.

Each design begins with the **tutorial** keyword. Two named lessons, indicated by the **lesson** keyword, describe verbs of successive levels. The first lesson newly introduces the verbs Store, Push and Move (lines 2–6). The win scenarios of the associated levels must present evidence for these verbs. The second lesson builds on prior knowledge and introduces challenge. The verbs Move, Push and Store are required (lines 8–14). However, players now have to Stash crates to solve the puzzle. For assessing challenge, we also need evidence of failure scenarios. The **avoid** keyword indicates that inadvertently blocking a path is a "good mistake". Table 1 shows example scenarios, each representing opportunities for learning. Next, we will explain how TutoScript can be used for analytics.

4 TUTOMATE

We present TutoMate, a tutorial design tool for PuzzleScript that leverges TutoScript and existing search algorithms to offer analytics that work on generated playtraces [30]. Unlike many other tools, TutoMate is not based on PuzzleScript's online engine [18].

Instead, we use the Rascal language workbench, which is especially suitable for developing DSLs and analyzing source code [14]. We use Rascal to extend ScriptButler, an existing framework specifically designed for analyzing PuzzleScript [12]. We can reuse its

PuzzleScript grammar for parsing every game in the open repository of remakes and demakes we use in this study [18]. Already capable of static analysis, the framework still lacks an interpreter and engine that is fast enough to perform dynamic analyses.

TutoMate adds the necessary components. We summarize how they work. Vet gives a more detailed account [30].

4.1 Enriching Playtraces with Verbs

To enrich playtraces with verbs, we introduce a simple mapping between rules and verbs. The verbs, defined in comments directly behind each rule, express how TutoScript can refer to them.

```
(player is defined implicitly)          (verb Walk)
[>Player | Crate] -> [>Player | >Crate] (verb Push)
```

TutoMate can enrich playtraces with these verbs. First, it collects the mapping from the sources. When applying a rule transformation, TutoMate adds the associated verb in the trace.

In addition, the win verb is based on PuzzleScript's win conditions. Verbs "get stuck" or "become trapped" are expressed as dead ends, occurrences where no rule exists that can be activated.

Of course, not every verb can be expressed this way. In particular complex verbs, such as "block path", cannot yet be expressed. In Section 5 we discuss case studies and show examples.

4.2 TutoMate Engine

TutoMate performs its language-parametric analyses as follows.

4.2.1 Engine. A new PuzzleScript engine adds an interpreter for analyzing and playing games. This interpreter processes simulated or real user actions. Given a button-press, it recursively loops through rules to identify which ones should trigger. For activated rules it performs the effects, updating the game state. To speed up the engine, rules that cannot trigger are filtered out beforehand.

4.2.2 TutoScript. Another extension adds TutoScript and a playtrace analyzer. This component compares the verbs of each lesson with playtraces of levels. In particular, these are sets of verbs. The analysis checks required and introduced verbs are in traces of success scenarios, flagging omissions. Verbs to avoid on the other hand, are compared against failure traces to obtain evidence for challenge.

4.2.3 Dynamic Analyzer. The core is a dynamic analyzer that uses the engine to generate playtraces. Initially, we have applied a brute force approach. To find the shortest traces, we have selected a simple breadth first search algorithm. However, due to the unavoidable combinatorial explosion problem, this approach does not scale well.

To speed up the analysis, we have used the level state heuristics proposed by Lim and Harell [20]. To select which rule to apply, these heuristics use metrics for the Manhattan distance between: 1) Win Condition and Objects; and 2) Player and Win Condition Objects. The search continues until the win condition is satisfied. Though not optimal, Section 5 shows it is adequate for validating the approach on puzzle tutorials with relatively small search spaces.

To help obtain evidence for opportunities for learning, the analyzer also detects failure scenarios using **`avoid`** verbs. Failures are deviations from the win scenario. To limit the search, and find proximate playtraces that are likely encountered, the deviation depth is limited (the default is two). Finally, dead ends are traces ending with the inability to activate rules without winning.

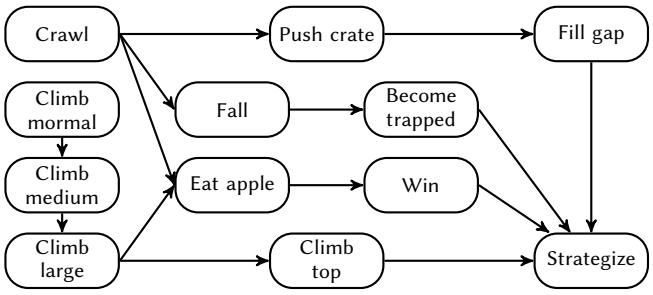


Figure 7: Lime Rick: Goal Chain of Verbs

4.2.4 User interface. Using Rascal's Salix framework, we create programming environments for PuzzleScript and TutoScript. The web-based interfaces includes a renderer, a path vizualizer and interactive controls for playing and debugging puzzle games.

5 CASE STUDIES

We validate the approach in case studies on Lime Rick and Block Faker. These are distinct puzzle games that each have high-quality tutorials with at least five levels. We explore to what extent TutoMate can automate the analysis of these tutorials. Specifically, we assess the viability of enriching playtraces with verbs and using TutoScript for analytics. The first two studies explore the expressiveness of TutoScript, and if TutoMate can provide evidence that the tutorials realize their goals. A third study, which illustrates its uses for mixed-initiative design with breaking changes, is part of Supplementary Material. Each study applies the following protocol.

5.1 Prototcol

First, we describe the game and analyze the tutorial by manually playtesting its PuzzleScript implementation. Next, we reverse engineer the tutorial design. We create a goal chain and use TutoScript to formalize a progression of lessons. We specify a mapping between PuzzleScript rules and verbs, reporting limitations.

We then apply TutoMate to analyze generated playtraces for solutions and failure scenarios for each of the levels in the tutorial. For every trace, we record: a) the user actions (button presses) and activated rules; b) its length (the number of steps); c) the verbs identified for each step; and d) the time elapsed (in seconds).

Finally, we assess if these results are as expected. We record if players: e) require verbs to solve the puzzle; f) meet challenges in the failure traces; and g) progress through puzzles as expected.

5.2 Lime Rick

5.2.1 Description. Lime Rick is a Snake-like game created by Tommi Tuovinen. In the game, the player is a snake whose objective is reach an apple in a side-view dungeon. Reaching the apple requires using a series of snaking mechanisms to crawl over obstacles. The rules ensure snaking movements are similar in structure to a limerick.

When crawling, using arrow keys, the snake grows from its head into open space. Players can only crawl upward three times before switching direction. The head color changes: green, yellow, orange, red. The snake needs solid ground to support its head. This includes its own tail. The snake can push crates with its head. Gravity pulls the head and crates own, causing both to fall.

```
UP [ UP PlayerHead4 ] -> [ PlayerHead4 ]                                  (verb ClimbTop    //cannot climb more)
UP [ UP PlayerHead3 | No Obstacle ] -> [ PlayerBodyV | PlayerHead4 ]      (verb ClimbLarge  //third climb: red head)
UP [ UP PlayerHead2 | No Obstacle ] -> [ PlayerBodyV | PlayerHead3 ]      (verb ClimbMedium //second climb: yellow head)
UP [ UP PlayerHead1 | No Obstacle ] -> [ PlayerBodyV | PlayerHead2 ]      (verb ClimbNormal //first climb: green head)
horizontal [> Player | Crate | No Obstacle] ->
      [PlayerBodyH | PlayerHead1 | Crate]                                 (verb Push        //push crates horizontally)
horizontal [ > Player | No Obstacle ] -> [PlayerBodyH | PlayerHead1]      (verb Crawl       //snake grows horizontally)
[ Player Apple ] [ PlayerBody ] -> [ Player Apple ] [ ]                   (verb Approach    //approach the apple)
[ Player Apple ] -> [ Player ]                                            (verb Eat         //eat the apple)
[ > Player ] -> [ Player ]                                                (verb Cancel      //prevent default movement)
DOWN [ Player | No Obstacle ] -> [ PlayerBodyV | PlayerHead1 ]            (verb Fall        //the head falls down)
DOWN [ Crate | No Obstacle ] ->  [ | Crate ]                              (verb FallCrate   //crates fall down)
```

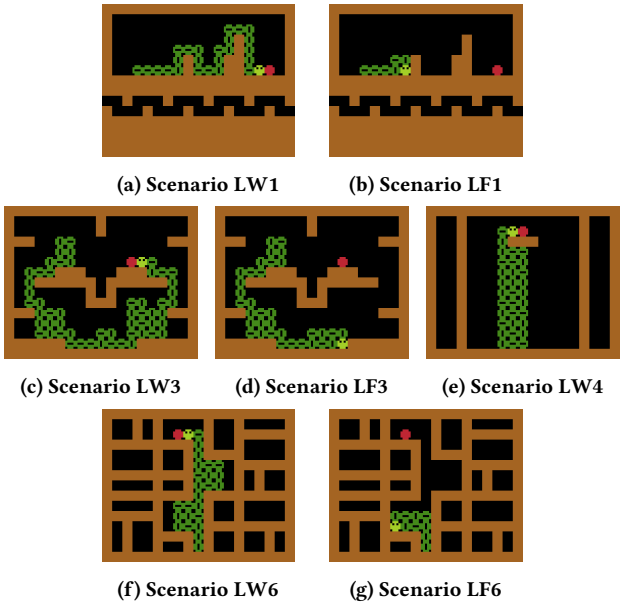
Figure 8: Lime Rick: PuzzleScript rules (on the left) annotated with Verb definitions (on the right)

```
1  tutorial LimeRick {
2    lesson 1: Crawling {
3      "The snake can crawl."
4      introduce Crawl
5      introduce ClimbNormal, ClimbMedium }
6
7    lesson 2: Snaking {
8      "The snake can climb on top of its own tail."
9      require ClimbNormal, ClimbMedium
10     introduce ClimbLarge, ClimbTop }
11
12   lesson 3: Trapped {
13     "When falling in a gap, the snake is trapped."
14     require ClimbNormal, ClimbMedium, ClimbLarge
15     avoid Fall }
16
17   lesson 4: Reach {
18     "The snake can reach great heights."
19     require ClimbNormal, ClimbMedium, ClimbLarge }
20
21   lesson 5: FillGaps {
22     "Gaps can be filled with blocks."
23     avoid Fall
24     require ClimbNormal, ClimbMedium, ClimbLarge
25     introduce Push, FallCrate }
26
27   lesson 6: Strategy {
28     "Mechanisms can be combined."
29     require ClimbNormal, ClimbMedium, ClimbLarge
30   }}
```

Figure 9: Lime Rick Tutorial Design in TutoScript

5.2.2 Tutorial. The Lime Rick tutorial consists of ten levels. We discuss and analyze the first six. The first level shows the player they can crawl, climb and fall. To eat the apple, players have to traverse two walls and experience growing, climbing, and falling in the process. In the second level, the player has to climb to a platform four blocks high. They learn the snake cannot climb that high unless it uses its own tail for support. The third level contains a gap. If the player moves towards the apple in a straight line, they find out they become trapped when falling inside. Eating the apple requires a detour. In the fourth level, the player has to reach a platform ten blocks high. Reaching it requires snaking multiple times. In the fifth level, reaching the apple involves filling a gap to avoid getting trapped. Finally, the sixth level requires a combination of mechanisms to avoid getting trapped and reaching the apple.

5.2.3 Tutorial Design. We describe its goal chain in Figure 7. Next, we create a mapping between verbs and rules, shown in Figure 8. Because in Lime Rick, all snake movements are explicitly defined, every verb can be mapped, except Strategize, which may be inferred.



(a) Scenario LW1 (b) Scenario LF1

(c) Scenario LW3 (d) Scenario LF3 (e) Scenario LW4

(f) Scenario LW6 (g) Scenario LF6

Figure 10: Lime Rick: Win and failure scenarios

We formalize the tutorial design using TutoScript. Figure 9 describes six lessons, one for each level. Lesson 1 introduces crawling and climbing (lines 4–5). Lesson 2 applies those verbs and adds climbing as far as possible (line 10). Lesson 3 teaches to avoid falling in gaps (line 15). Lesson 4 four again applies climbing (line 19). Lessons 5 and 6 introduces pushing and filling gaps (line 25). Lesson 6 again applies climbing (line 29).

5.2.4 Analysis. We apply TutoMate to the PuzzleScript sources, and obtain the enriched playtraces for levels 1, 3, 4 and 6 for win and failure scenarios. For levels 2 and 5, we obtain no results. The search times out due to failing heuristics. We discuss the results using the traces shown in Table 2 and Figure 10.

Crawling is required in level 1. The win scenario (LW1) is a trace of 19 steps computed in 15.8 seconds. It also shows players have to eat the apple to win. The failure scenario (LF1) is a trivial scenario of 5 steps. The snake becomes trapped behind a ledge.

Reaching the apple in level 3 requires crawling and climbing. The win scenario (LW3) is a trace of 40 steps computed in 116.7 seconds. By deviating slightly from the solution, the snake becomes trapped. The trace LF3 is an example failure scenario of 29 steps computed in 29.6 seconds. The trace shows verbs and actions it shares with the win scenario in gray. Deviations are shown in black.

Table 2: Lime Rick: Enriched playtraces and checks

Name	Playtrace + verbs	L.	D. (s)	Pass
LW1	[c→*4, c_n↑, c_m↑, c→*2, f, c→*2, c_n↑, c_m↑, c→, c_n↑, c_m↑, c→*2, f, c→, e→w]	19	15.8	✓
LF1	[c→*3, c_n↑, c→, f t]	5	6.3	×
LW3	[c_n↑, c_m↑, c←, f, c←, f, c←*2, f, c→, f, c→, c_n↑, c_m↑, c→, f, c→, f, c→*3, c_n↑, c→, f, c→*2, c_n↑, c→, c_n↑, c←, c_n↑, c_m↑, c→, f, c→, f, c→, c_n↑, c_m↑, c_l↑, c→, c_n↑, c_m↑, c_l↑, c→*2, c_n↑, c←, e←w]	40	116.7	✓
LF3	[c_n↑, c_m↑, c←, f, c←, f, c←*2, f, c→, f, c→, c_n↑, c_m↑, c→, f, c→, f, c→*3, c_n↑, c→, f, c→, c_n↑, c→, f t]	20	29.6	×
LW4	[c←*2, c_n↑, c_m↑, c→, f, c→, c_n↑, c_m↑, c←*2, c_n↑, c→*2, c_n↑, c_m↑, c_l↑, c←, f, c←, c_n↑, c_m↑, c_l↑, c→*2, c_n↑, c←*2, c_n↑, c_m↑, c→, e→]	30	10.4	✓
LW6	[c_n↑, c_m↑, c_l↑, c←, f, c←, c_n↑, c_m↑, c→*2, c_n↑, c_m↑, c_l↑, c→, f, c→, c_n↑, c_m↑, c←*2, c_n↑, c_m↑, c_l↑, c←, f, c←, c_n↑, c_m↑, c←, e←w]	23	49.6	✓
LF6	[c_n↑, c_m↑, c_l↑, c←, f, c←, c_n↑, c←, f t]	7	8.7	×

c = crawl, c_n = climb normal, c_m = climb medium, c_l = climb large
f = fall, e = eat apple, t = become trapped, w = win

Reaching the goal indeed requires crawling and climbing. The win scenario for lesson 4 (LW4) is a trace of 30 steps computed in 10.4 seconds. No proximate failure scenarios were identified.

Level 6 combines crawling and climbing. The win scenario for lesson 6 is a trace of 23 steps computed in 49.6 seconds (LW6). By deviating slightly from the solution, the snake can get trapped. LF6 is an illustrative failure scenario computed in 8.7 seconds.

Even though we cannot express the verb Strategize or fill gap directly, we can still use the analytics to learn more about the tutorial design. Several failure scenarios indicate challenge.

5.3 Block Faker

5.3.1 Description. Block Faker is a puzzle game by Droqen that centers around push and vanish mechanisms. In a top-down view, the player has to create a path to reach the exit (a green square) by pushing crates that collide with walls (using arrow keys). Whenever pushing results in three equal adjacent crates, they each disappear.

5.3.2 Tutorial. The Block Faker tutorial is a demo, which consists of five levels that encourage players to play the full game.

Pushing is the goal of the first level. Deviating from the path is not possible, forcing the player to push the crate. As a result, the player observes how pushing works. *Vanishing* is a new goal of the second level. The player has to push a crate once again. This time, pushing results in three adjacent purple crates. The player observes the alignment causes the crates to disappear.

Obstacles are introduced in the third level. Passing requires applying the vanishing mechanism. When the player pushes a purple crate too far, an alignment causes it to vanish. However, as a result, the exit becomes unreachable. Solving the puzzle requires realizing this crate can instead be used to clear the path ahead. *Strategizing* happens in the more complex puzzles of the fourth and fifth levels. Solving these requires applying the push and vanish mechanisms, and dealing with an increasing number of obstacles to clear a path.

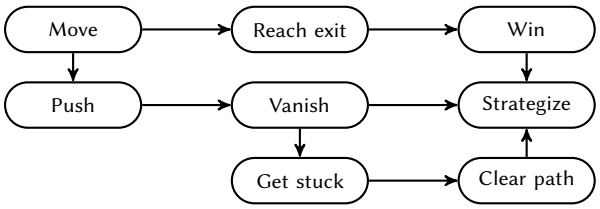


Figure 11: Block Faker: Goal Chain of Verbs

```
1  tutorial BlockFaker {
2    lesson 1: Pushing {
3      "Blocks can be pushed."
4      introduce Push }
5
6    lesson 2: Vanishing {
7      "Three adjacent blocks vanish."
8      require Push
9      introduce VanishPurple VanishOrange }
10
11   lesson 3: Obstacle {
12     "Vanishing can be used to pass obstacles."
13     require Push, VanishPurple
14     avoid VanishPink }
15
16   lesson 4: Combinations {
17     "Techniques can be combined."
18     require Push
19     introduce VanishGreen, VanishOrange }
20
21   lesson 5: Strategy {
22     "A strategy that requires moving all blocks."
23     require Push, VanishPink, VanishPurple,
           VanishOrange, VanishBlue, VanishGreen } }
```

Figure 12: Block Faker Tutorial Design in TutoScript

5.3.3 Tutorial design. We extract the goal chain in Figure 11. We then map its verbs to rules in Figure 13. Because movement is defined, we add Walk verb before the first rule. We cannot map "Get stuck" to a rule, or express the derived verb Strategize.

Figure 12 shows a an implementation of five lessons, one for each level. Lesson 1 introduces the Push mechanism (line 4). Pushing is necessary to solve the level. Lesson 2 applies pushing, a known mechanism, and adds the Vanish mechanism (lines 9). Players are required to align purple and orange crates. Lesson 3 introduces an obstacle. The shortest trace to VanishPink is a failure scenario. Players should avoid this obstacle (line 14). The fourth and fifth lessons require combining vanish mechanisms to solve the puzzle.

5.3.4 Analysis. We apply TutoMate to the PuzzleScript sources, and obtain the enriched playtraces shown in Table 3. Because the algorithm times out, we do not obtain win scenarios for levels 3 and 4. As a result, no failure scenario can be identified either.

As expected, *pushing* is required in the win scenario for lesson 1. A trace (BW1) of 26 steps is computed in 15.7 seconds. The trace shows players have to reach the exit to win. Furthermore, *vanishing* is part of the win scenario for lesson 2. The trace (BW2) has 24 steps that show purple and orange crates have to vanish in this solution.

Strategy is harder to quantify and gauge. The win scenario for lesson 5 is a trace (BW5) of 26 steps computed in 15.7 seconds.

```
(the movement of the player is defined implicitly)          (verb Walk          //move left, right, up, down)
[ > Moveable | Moveable ] -> [ > Moveable | > Moveable ]    (verb Push          //push a crate)
[ > Block | Grille ] -> [ Block | Grille ]                  (verb Collide       //collisions prevent movement)
late [ PinkBlock | PinkBlock | PinkBlock ] -> [ | | ]       (verb VanishPink    //three pink blocks vanish)
late [ BlueBlock | BlueBlock | BlueBlock ] -> [ | | ]       (verb VanishBlue    //three blue blocks vanish)
late [ PurpleBlock | PurpleBlock | PurpleBlock ] -> [ | | ] (verb VanishPurple  //three purple blocks vanish)
late [ OrangeBlock | OrangeBlock | OrangeBlock ] -> [ | | ] (verb VanishOrange  //three orange blocks vanish)
late [ GreenBlock | GreenBlock | GreenBlock ] -> [ | | ]    (verb VanishGreen   //three green blocks vanish)
```

Figure 13: Block Faker PuzzleScript rules (on the left) annotated with Verb definitions (on the right)

Table 3: Block Faker: Enriched playtraces and checks

Name	Playtrace + verbs	L.	D. (s)	Pass
BW1	[w↑, w←, p←, w↑, w←, p↓*2, w←*2, w↑*3, w→, w↑, w→, p→*4, w↑, w→ p↓*2, w→, w↓, w→, r]	26	15.7	✓
BW2	[p↑*4, v_{pl}, w←*5, w↓, w→, p↑, w←, w↑, p→, w↑, w→, p↓, v_o, w↓, w→*3, r]	24	10.8	✓
BW5	[p↓, p→, p→ v_g, w→*3, w↓*2, p↓, v_o, w↓, r]	26	15.7	×
MW2	[p↑*2, v_o, w↑, w→*3, r]	6	2.3	×

w = walk, p = push, r = reach exit / win
v_o = vanish orange, v_g = vanish green, v_{pk} = vanish pink, v_{pl} = vanish purple

TutoMate identifies verbs that are not in the trace. Initially puzzled by trace WB5, upon closer inspection we find this is intended behavior. Blue, purple and pink crates need not vanish.

6 DISCUSSION

6.1 Costs and benefits

We have proposed a novel approach for predicting if puzzle tutorials will present opportunities for learning. The case studies show TutoMate helps assess to what extent playtraces enriched with verbs, realize the goal chains of a puzzle tutorial. TutoMate provides useful analytics and TutoScript is an expressive notation for expressing these analytics in terms of verbs. Particularly insightful are the failure scenarios, which indicate challenge.

For debugging, the interactive visualizations with replay functionality are especially helpful. A key benefit of test automation is reducing the need for manual playtesting. In mixed-initiative design settings, where designers make gradual changes to levels and rules, re-testing hypotheses is particularly helpful. At present however, the puzzle-solving algorithms are still prohibitively slow.

Of course, there are also costs. The automation shifts the division of work between engineers and designers. While it helps raise the tutorial quality, the workload is not necessarily reduced. As with every DSL, learning to program TutoScript costs time and effort, and maintaining the framework is a long term investment.

6.2 Limitations and threats to validity

Of course, our approach is a research prototype and a proof of concept. We describe its limitations and discuss threats to validity.

The approach relies on the availability of playtraces, in particular generated ones. Because no off-the-shelf solution suited our needs, we have implemented a simple breadth first search algorithm. Since tutorials are designed to be safe, manageable and simple, we have assumed they have relatively small search spaces. Our engine is not yet optimized for speed and scalability. Depending on the puzzle, generating solutions results in potentially vast state spaces. For now, applications to complex puzzles are therefore out of scope.

The one-to-one mapping between verbs and PuzzleScript rules limits which verbs can be recognized. In particular, the notation cannot yet express composite verbs or derived skills. For instance, the verb *strategize* is never a single act. In Lime Rick, players "snake" by crawling on top of the snake's tail. A more powerful TutoScript is part of future work. Despite its shortcomings, the results show the approach is viable for small search spaces. The simple mapping is sufficient for many meaningful analyses. TutoMate helps provide useful insights that are difficult to obtain by other means.

7 RELATED WORK

The contributions of this paper intersect at the areas of game analytics, automated game design and procedural content generation.

7.1 Game Analytics

Game Analytics is an area that leverages player data to perform quantitative analyses [8]. Many tools, techniques and commercial frameworks have been created that record and analyze game states, events and interactions, e.g., activities and time spent. Using metrics, visualizations and statistical analyses, these tools can provide valuable insight on a game's qualities, e.g., player preferences, and reasons to play. Playtraces are commonly used in analytics to relate observations to expectations [9, 21, 23]. Osborn et al. propose PlaySpecs, regular expressions for analyzing playtraces [23]. Playtraces are defined as sequences of facts that record observations of the game state. In accordance with Koster's Theory of Fun [15] and best practices [25], a standardization effort has identified verbs as a key data point [4]. However, despite the availability of playtraces, these techniques have not yet been applied to puzzles. We address this challenge with specialized metrics and analytics.

7.2 Automated Game Design

Automated Game Design (AGD) is a research area that develops and applies techniques for automating game design processes [6, 29]. Domain-Specific Languages (DSLs) are a particular means to give such tools expressive power [28], e.g., for enabling generative approaches and analyses with predictive accuracy. In a comprehensive survey of languages and tools for game design and development, van Rozen identifies PuzzleScript as one of 108 languages in over 1400 publications [29]. Due to a lack of empirical studies, what the merits of DSLs are for AGD is still largely unknown [16]. Our study on PuzzleScript sheds additional light on this question.

A recent survey on puzzle generation shows the state-of-the-art consists of highly focused applications that cannot be easily reused [13]. Our DSL approach represents a promising first step towards generic techniques for analyzing and generating tutorials.

7.3 Tools for Puzzles and Tutorials

7.3.1 Algorithms. Research on puzzles has yielded algorithms and techniques that can solve puzzles automatically, e.g., using SMT solvers [3], Answer Set Programming (ASP) [26, 27], and Heuristics Search [20]. Cooper describes Sturgeon-MKIII, a tool and technique that uses SAT solvers for obtaining playthroughs for rule-based transformations [7]. Green et al. propose an evolutionary approach to generate Mario game tutorials [10]. We leverage existing algorithms to obtain playtraces. Unlike many other approaches, we also focus on failures and dead-ends, which are indicative of challenge.

7.3.2 Skill Atoms. Skill Atom theory originates with Dan Cook who proposes designing challenges using skill chains [5]. Authors have proposed techniques and tools that apply this theory [2, 11].

Talin is a low-code tool for creating tutorials by visually designing a mastery model [2]. We also propose designing mastery models. However, our aim is not to generate feedback only when needed, but to obtain evidence that opportunities for learning exist.

Similar to Skill Atoms, Koster uses Verbs to design affordances [15, 16]. The visual notation of TutoScript also foregrounds verbs. However, its aim is to describe prerequisite knowledge.

7.3.3 PuzzleScript. Lim and Harell propose an approach for analyzing PuzzlesScript that generates solutions to puzzles [20]. Two sets of heuristics speed up the analysis. The first, *level state heuristics* is used for evaluating how close the state of given level is to completion during gameplay. The second, *ruleset heuristics* evaluates a videogame's mechanics and assesses them for playability. TutoMate uses the level state heuristics.

Naus and Jeuring study problem solving for rule-based systems [22]. Inspired by intelligent tutoring systems and expert systems, they propose a generic approach for creating feedback systems that provide hints, e.g., next-step hints for PuzzleScript puzzles. Their DSL expresses rule based problems and leverages generic search algorithms to provide hints. In contrast, we aim to provide feedback on the tutorial design itself.

8 CONCLUSIONS

We have proposed a novel approach for analyzing puzzle tutorials that enriches playtraces with verbs. We have introduced TutoScript, a textual and visual DSL for expressing goal chains in terms of verbs. We have presented TutoMate, a tool for PuzzleScript that offers analytics for automating the analysis of goal chains. By combining TutoScript with well-known search algorithms, and by mapping rules to verbs, TutoMate can enrich, analyze and visualize generated playtraces of solutions, failures and dead ends. To validate our approach, we have performed case studies on Lime Rick and Block Faker. Our results demonstrate TutoMate can recognize simple goal chains, and also detects broken tutorials.

Future Work. Generating puzzle game tutorials can provide personalized learning trajectories [10]. Extending TutoScript with patterns could make the analyses more precise [23]. Finally, investigating code puzzles could help to generalize the approach.

ACKNOWLEDGMENTS

We thank the reviewers for providing insightful comments, and for giving crucial suggestions that have helped improve this paper.

REFERENCES

[1] Anna Anthropy. 2019. *Make Your Own PuzzleScript Games!* No Starch Press.
[2] Batu Aytemiz, Isaac Karth, Jesse Harder, Adam M. Smith, and Jim Whitehead. 2018. Talin: A Framework for Dynamic Tutorials Based on Skill Atoms Theory. In *Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2018.* AAAI.
[3] Eric Butler, Emina Torlak, and Zoran Popovic. 2017. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Foundations of Digital Games.* ACM.
[4] Sven Charleer, Francisco Gutiérrez, Kathrin Gerling, and Katrien Verbert. 2018. Towards an Open Standard for Gameplay Metrics. In *CHI Play.* ACM.
[5] Daniel Cook. 2007. The Chemistry Of Game Design. *Game Developer* (July 2007). https://www.gamedeveloper.com/design/the-chemistry-of-game-design
[6] Michael Cook. 2020. Software Engineering for Automated Game Design. In *2020 IEEE Conference on Games (CoG).* IEEE.
[7] Seth Cooper. 2023. Sturgeon-MKIII: Simultaneous Level and Example Playthrough Generation via Constraint Satisfaction with Tile Rewrite Rules. In *Foundations of Digital Games, FDG 2023.* ACM.
[8] Magy Seif El-Nasr, Anders Drachen, and Alessandro Canossa. 2016. *Game Analytics.* Springer.
[9] Michael Cerny Green, Ahmed Khalifa, Gabriella A. B. Barros, Tiago Machado, and Julian Togelius. 2020. Automatic Critical Mechanic Discovery Using Playtraces in Video Games. In *Foundations of Digital Games.* ACM.
[10] Michael Cerny Green, Ahmed Khalifa, Gabriella A. B. Barros, Andy Nealen, and Julian Togelius. 2018. Generating Levels that Teach Mechanics. In *Foundations of Digital Games, FDG 2018.* ACM.
[11] Britton Horn, Seth Cooper, and Sebastian Deterding. 2017. Adapting Cognitive Task Analysis to Elicit the Skill Chain of a Game. In *Proceedings of the Annual Symposium on Computer-Human Interaction in Play, CHI PLAY 2017.* ACM.
[12] Clement Julia and Riemer van Rozen. 2023. ScriptButler serves an Empirical Study of PuzzleScript. In *Foundations of Digital Games, FDG 2023.* ACM.
[13] Barbara De Kegel and Mads Haahr. 2020. Procedural Puzzle Generation: A Survey. *IEEE Trans. Games* 12, 1 (2020).
[14] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. EASY Meta-programming with Rascal. In *Generative and Transformational Techniques in Software Engineering (LNCS, Vol. 6491).* Springer.
[15] Raph Koster. 2013. *Theory of fun for game design.* O'Reilly Media, Inc.
[16] Raph Koster. 2016. The Limits of Formalism. Raph Koster's Website. *Presentation delivered at the BIRS Workshop on Computational Modeling in Games* (2016). https://www.raphkoster.com/games/presentations/the-limits-of-formalism
[17] Craig Larman. 2012. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interactive Development.* Pearson Education.
[18] Stephen Lavelle. 2013. PuzzleScript. https://github.com/increpare/PuzzleScript Last visited: November 26th 20203.
[19] Stephen Lavelle. 2013. PuzzleScript Documentation. https://www.puzzlescript.net/Documentation/documentation.html Last visited: November 26th 20203.
[20] Chong-U Lim and D. Fox Harrell. 2014. An Approach to General Videogame Evaluation and Automatic Generation using a Description Language. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014.* IEEE.
[21] Yun-En Liu, Erik Andersen, Rich Snider, Seth Cooper, and Zoran Popovic. 2011. Feature-based Projections for Effective Playtrace Analysis. In *Foundations of Digital Games.* ACM.
[22] Nico Naus and Johan Jeuring. 2019. Building a Generic Feedback System for Rule-Based Problems. In *Trends in Functional Programming.* Springer.
[23] Joseph Carter Osborn, Ben Samuel, Michael Mateas, and Noah Wardrip-Fruin. 2015. Playspecs: Regular Expressions for Game Play Traces. In *Artificial Intelligence and Interactive Digital Entertainment, AIIDE.* AAAI Press.
[24] Katie Salen and Eric Zimmerman. 2004. *Rules of Play.* MIT Press.
[25] Jesse Schell. 2008. *The Art of Game Design: A Book of Lenses.* CRC press.
[26] Adam M. Smith, Eric Butler, and Zoran Popovic. 2013. Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design. In *Foundations of Digital Games, FDG 2013.* SASDG.
[27] Adam Summerville, Chris Martens, Ben Samuel, Joseph C. Osborn, Noah Wardrip-Fruin, and Michael Mateas. 2018. Gemini: Bidirectional Generation and Analysis of Games via ASP. In *Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2018.* AAAI Press.
[28] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35, 6 (2000).
[29] Riemer van Rozen. 2021. Languages of Games and Play: A Systematic Mapping Study. *Comput. Surveys* 53, 6 (2021).
[30] Dennis Vet. 2023. *Tutomate: Relating skill atoms to playtraces for enabling automated analysis of game tutorials.* Master's thesis. University of Amsterdam. https://github.com/Hyper445/Tutomate
[31] Roel. J. Wieringa. 2014. *Design Science Methodology for Information Systems and Software Engineering.* Springer.