Towards Procedural Generation of Narrative Puzzles for Adventure Games

Barbara De Kegel[1][0000−0002−4194−234X] and Mads Haahr[2][0000−0002−9273−6458]

[1] University College Dublin
[2] Trinity College Dublin

Abstract. Narrative puzzles involve exploration, logical thinking and progressing a story. This paper presents a narrative design innovation in the form of a system for the procedural generation of such puzzles for use in story-rich games or games with large open worlds. The approach uses an extended type of context-free grammar as the basis for both the generation algorithm and the puzzle solving. Each designer-defined rule in the grammar defines a possible behavior of item types in the game world. Puzzles are generated at runtime on a per area basis, through recursive generation of inputs for outputs. Given a valid grammar, the system guarantees that its puzzles are solvable.

Keywords: procedural content generation · puzzles · interactive narrative · authoring tools

1 Introduction

Narrative puzzles can be defined as puzzles that form part of the progression of a narrative, whose solutions involve exploration and logical as well as creative thinking. They are a key component of adventure and story-driven games, and often feature in large open world games, including RPGs. Narrative puzzles can be viewed as temporary obstacles to the story's advancement; though they do not always have to be solved in a precise order, certain puzzle sequences generally need to be solved before proceeding to others. Typically, good narrative puzzles involve making logical connections, which may not be immediately obvious, but which ultimately comprise a satisfying solution. Puzzlers typically find solutions by exploring the environment and investigating ways in which objects can be manipulated. Examples of narrative puzzle patterns identified by Fernández-Vara *et al.* [3] are: (a) Figuring out which item a character desires, usually leading to a reward in exchange; (b) Logically combining two objects to change their properties, or to create a new object; (c) Disassembling an object into useful components; (d) Saying 'the right thing' to convince a character to provide aid; and (e) Acquiring a key to open a new area.

Due to space constraints, we are not able to present a detailed review of related work here, but we refer the reader to our recent survey of procedural generation of puzzles [2], which contains a section on narrative puzzles, including Puzzle-Dice [3], as well as work by Dart and Nelson [1] and van der Linden *et al.* [4].

2 Design

Our system aims to improve replayability of smaller story-driven games as well as offer way to improve the narrative engagement of games with large open worlds and a high degree of procedural content. Our approach is inspired by (and improves upon) Puzzle-Dice [3], specifically in terms of expressivity, usability and scalability, while maintaining the guarantee of solvability.

2.1 Core Concepts

The approach is based on a context-free grammar that defines possible behaviors of game items. The puzzle generator integrates with a game world to create puzzles on the fly based on the current state of the world. There are three components that feed into the generator: a database of all items that can be used in puzzles, a set of grammar rules that describe the space of all possible puzzles, and a list of the game areas. Several core concepts form the basis of these components:

- **Items**: Conceptual game objects which are defined by their type(s) and properties.
- **Properties**: Named characteristics of Items, which have a value of specific value type.
- **Rules**: Possible in-game actions, composed of an output Term, a set of by-product Terms, an Action and a set of input Terms (see figure 1).
- **Terms**: The main units out of which Rules are composed, each is defined by a single type and an optional list of properties.
- **Action**: The unit of a Rule that described the player's action in carrying out a rule.
- **Area**: A single connected space that forms part of the game world; used to compartmentalize the puzzle generation.

The defintion of the components is flexible in terms of the designer-defined content they can support, allowing the generator to be applied to a range of different types of games.

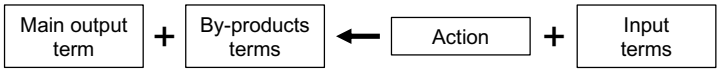

Fig. 1. Abstract representation of the general structure of a rule.

The generator uses the set of production rules that constitute the grammar in a left to right direction to generate a puzzle backwards from an end goal. The backwards process ensures the puzzle is solvable. In a game that incorporates the generated puzzles, the same rules—but used in the right to left direction—function as game logic.

2.2 The Puzzle Items

A puzzle item is a conceptual representation of a tangible object that can be used as part of a generated puzzle. Each puzzle item has a unique name, an optional list of properties, and an associated visual representation, e.g., a Unity prefab. There may be more than one puzzle item for an object that has multiple states, e.g., a tree in summer and that same tree in autumn. These specificities in item definitions are left open to the game designer.

Items' properties are defined by their name and type; the type—string, boolean or integer—determines the legal values for the property. Properties are freely defined by the game designer and can be tailored to the needs of the puzzle game. There are no required properties; if a property is not defined for an item, the generator assumes it does not have this property, or for boolean properties, assumes the value is *False*. For example, not specifying the *carryable* property is equivalent to marking an item as 'not carryable.'

There are several special properties which have explicit logic attached to them. One is the *carryable* property; an item is queried in-game for this specific property to determine whether it can be added to a player's inventory.

Another special property is the *isa* property, which can be used to define all the categories (i.e., super classes) a certain item belongs to; e.g., a *PineTree* might have the *isa* properties *Tree* and *Plant*. The value associated with an *isa* property may or may not be the name of another item in the puzzle items database. The name of an item is automatically considered an *isa* property of that item—it defines the most specific category the item belongs to. In addition, every puzzle item is automatically considered to be a sub-type of the type *Item*. The *isa* property allows for hierarchies among the types of puzzle items, which is central to the functioning of the grammar rules discussed in section 2.3.

The *contains* property is also a special case—though it is a string property, its value is interpreted as a puzzle item. As will be discussed in section 2.3, this property is particularly important in the definition of rules, which can refer to transient item states.

Finally there are two special properties that can be used to restrict the possible locations of puzzle items. The *notSpawnable* property, indicates that a puzzle item can only be used if it is already part of the game world, and will not be instantiated as a rule output, e.g., a large lake. The *area* property can be used to specify the legal areas for spawning and/or using an item, allowing the game designer to control which items may be included as part of a puzzle on a per-area basis.

2.3 The Grammar

The grammar, which comprises of a set of production rules, describes the space of all possible puzzles. Each rule describes a relationship between a set of inputs and a set of outputs, in a format that is loosely based on the format of rules that make up a context-free grammar. The rules serve a dual purpose: they are

used by the generator to create puzzles and as game logic. The general format of a rule is as follows:

$$itemType[properties_{0...n}]_{1...n} ::= action\ itemType[properties_{0...n}]_{1...n} \quad (1)$$

In a context-free grammar, all the productions are one-to-one, one-to-many or one-to-none. The rules that comprise the puzzle grammar fall under the first two categories. Production rules are read from left to right and can be interpreted as breaking down an output into its input(s), or replacing an output with one or more inputs. A puzzle, in the form of a tree structure, is created by iteratively (recursively) decomposing outputs, starting from an end goal.

In practice, the rules can (and often do) have multiple outputs because the right and left hand sides of the rule describe which items exist in the gameworld, and in what state, before and after the rule is applied. For generating a puzzle only the first output is important, and the others are considered by-products. For example, in rule 2, which expresses chopping down a tree, the axe is not an outcome, but it is important to account for the fact that it was not consumed as part of the execution of the rule. Each input (right-hand side term), is considered to be destroyed if it does not appear as an output (left-hand side term). The exception to this is an input that appear as the value of the *contains* property for an output—these are also not considered destroyed. Rule 3 shows an example of this type of behavior.

$$TreeStump\ Axe ::= ChopDown\ Tree\ Axe \quad (2)$$

$$Container[contains : Eggs] ::= Gather\ Eggs\ Container \quad (3)$$

The output of each rule is thus one or more terms, while the input is composed of at least one term, as shown in figure 1. Terms represent the non-terminals of the grammar while the puzzle items represent the terminals. There are implicit rules for replacing terms with specific puzzle items—terms can be seen as boxes with descriptions of what kind of puzzle item could be placed inside.

The terminals (puzzle items) are not directly used in the authoring of the grammar rules; a designer only looks at linking terms (non-terminals) to other terms. Internally, the puzzle generation system contains logic for determing which non-terminals could be replaced with terminals from the item database. The grammar is only valid if each input term can be matched to at least one output term in a different rule, or at least one puzzle item. Designers should be conscious of this when authoring the puzzle rules.

Terms have an item type and an optional list of properties. The item type corresponds to the previously described *isa* property and can be specific (e.g., *PineTree*) or general (e.g., *Plant*). The more general the type, the more puzzle items have the potential to be matched to a term. The special type *Item* can be used for terms that are allowed to be replaced by any puzzle item.

The properties associated with a term are fundamentally the same as those for a puzzle item. For a puzzle item to match a term it must be of the same type or a sub-type as the term's type, and it must include all properties of the term (though it can have many more properties than those required by the term).

Besides inputs and outputs, each rule must also have an action, which can be considered a terminal. This action is only used as part of the second purpose of the rules, i.e., as game logic, and has no bearing on the puzzle generation. The action is associated with the first input term, and as such, it is important to consider the order of the input terms; for example in rule 2, the action *ChopDown* should appear attached to the *Tree* term, rather than the *Axe* term.

2.4 The Puzzle Areas

Each puzzle area corresponds to a connected area in the game world and must have an associated goal. The goal is used by the generator as the starting point for generating a puzzle for that area. A designer can associate multiple possible goals with each area in order to increase the possibility space of puzzles that can be generated for that area. The format of an area goal is the same as that of a single term in a rule of the grammar. Each goal specifies a type of puzzle item that must be obtained, and an optional list of properties that must be fulfilled for that item. The generator checks that the goal cannot be satisfied by any intermediate items that are chosen as part of the puzzle, as this would result in a player completing a puzzle prematurely.

Besides a goal, a puzzle area has a unique name, a list of connected areas, and maximum puzzle depth. The maximum depth refers to the depth of the tree structure representation of the puzzle that is created by the generator. Puzzle areas can be predefined, or in the case of a procedurally generated game, they could also be automatically defined at run-time based on environmental attributes. The player's current in-game area is tracked by the generator and used to spawn puzzle items pick area appropriate rules.

2.5 Puzzle Generation

The puzzle generator works by recursively generating inputs for outputs using the set of rules that make up the puzzle grammar. The rules are used in the left to right direction as production rules and do not take into account the by-product terms. Puzzle generation is done live, i.e., while the game is being played, on the basis of currently accessible areas and items. At a high level (between areas), generation is running forwards throughout the game, but at a low level (within each area), generation runs backwards. This forward-backwards combination ensures solvability, quality and lack of repetition for the generated puzzles.

At the start of the game, a puzzle is generated for the area that has been designated as the start area. Finishing a puzzle for one area, (i.e., achieving the area's goal), causes all its connected areas to become unlocked, and triggers the generation of puzzles for those newly available areas. This forwards part of the algorithm can branch off into different tracks depending on the specified connections between areas. The system maintains each of the available areas independently, so multiple puzzles can be in progress at the same time. The overall forward direction of the algorithm allows for scenarios in which an item

that is needed to solve a puzzle for one area must be retrieved from another area.

When generating a puzzle for an area, the algorithm begins by finding a rule with a left hand side term that matches the current area's goal. The area goal is analogous to the grammar's start symbol. From that starting rule, the generator continues trying to substitute right hand side terms for other terms until no suitable rule can be found to perform such a substitution, or the area's depth limit is reached. At that point, the generator adds the puzzle item (terminal) that matches the last term to the game world. The rules used for the substitutions are recursively chained together into a tree structure that defines the entirety of the created puzzle. The items spawned in the world correspond to the input terms for the rules that make up the leaves of that tree.

An example of a generated puzzle is shown in figure 2, followed by the rules that would be chained together to create that puzzle. In reality, it is the rules that make up the nodes of the tree, rather than the terms, but the terms make for a clearer representation of the structure. The narrative solution to this puzzle is as follows: first the player must assemble a disguise out of glasses and a fake moustache and set of a car alarm to distract the security guard; these events can happen in either order. Then the player can steal the distracted security guard's badge, and proceed to unlock the safe with it. Finally, once the safe is unlocked, the player can open it and access the gold (the goal of the puzzle).

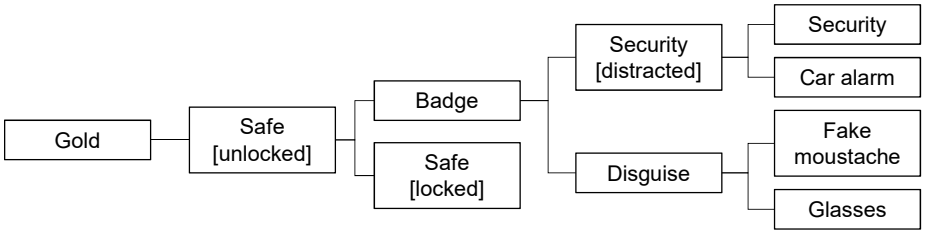

Fig. 2. An example puzzle tree.

$$Gold\ Safe ::= Open\ Safe[locked : False] \tag{4}$$

$$Safe[locked : False]\ Badge ::= Unlock\ Safe[locked : True]\ Badge \tag{5}$$

$$Badge\ Security ::= Steal\ Security[distracted : True]\ Disguise \tag{6}$$

$$Security[distracted : True] ::= Trigger\ CarAlarm\ Security[distracted : False] \tag{7}$$

$$Disguise ::= CreateDisguise\ Glasses\ FakeMoustache \tag{8}$$

Matching Terms Terms can be matched to other terms according to their types and properties. The properties must be an exact match, but the type of the output term can be the same or more general than the type of the input

term. For example, an input term of type *Tree* could be replaced by a rule with an output term of type *Tree* or *Plant* but not by one of type *PineTree*.

Notably, the generation algorithm does not wait until it reaches a terminal to pick a matching puzzle items for a term but rather attempts to find one as early as possible. The reason is that this allows for the use of more specific rules, widening the scope of possible puzzles. Terms become more specific as a result of an associated puzzle item, and can then be matched to a wider variety of output terms in other rules. For example, a rule with an input term with type *Tree*, as in the previous example, might pick a *PineTree* item as the matching puzzle item and change its type accordingly.

When an item replacement is found for a term, that item is passed up the tree to previously visited rules, and attached to corresponding terms. In this way, each term in each rule in the puzzle tree structure will have an associated puzzle item when generation completes, for use during the solving of the puzzle.

Generation per Game Area The game areas are modular but conscious of their context. New puzzles are created on a per area basis, with the generation algorithm taking into account all currently accessible areas, all items currently in the world, and all items in the player's inventory. The generator ensures that puzzle items chosen for a term are accessible and appropriate, making use of the items' *area* and *notSpawnable* properties. Additionally, generation will terminate upon reaching an intermediate puzzle item that already exists in the world to prevent recreating a puzzle that the player has already solved, or creating a puzzle that is trivial, because the player already has the goal item.

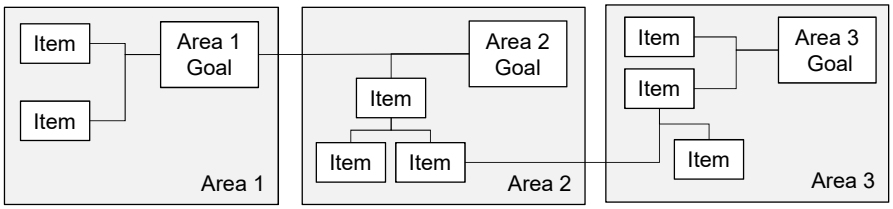

Fig. 3. A layout of how puzzles in different game areas can be interconnected.

Figure 3 shows how puzzles in each area can re-use items from previously visited areas. For example, the goal for area 1 is re-used as one of the input items needed to acquire the goal for area 2, and one of the items from area 2 can be re-used as an input to a puzzle in area 3. Puzzles are generated per area in a linear order for this example, e.g., the puzzles for area 2 are created after the goal for area 1 has been achieved.

We do not make the assumption that the world is empty at the start—existing objects in the scene can be included in the puzzles, if they are identified as puzzle items. This is an important design choice for integrating puzzles into an

environment. Puzzle items could correspond to environmental features, such as a lake, or large static structures, which are more easily placed in the game world as part of scene design, allowing for freedom in the construction of the game world. One reason for this choice is the potential use of this puzzle generator in a game with a procedurally generated environment, such as *Minecraft* or *No Man's Sky*. In these games, the puzzle generator could run as a separate layer on top of the existing generator and construct puzzles featuring already spawned game objects, environmental features and NPCs.

The puzzle generator also tracks the depth of the tree that represents the current puzzle, allowing for a designer specified puzzle length. The number of actions needed to solve a puzzle is also determined by the breadth of the tree but due to a low average branching factor (most rules will have one or two inputs), depth influences the length of the solution sequence more than breadth.

2.6 Puzzle Solving

Next to puzzle generation, the grammar rules also provide the in-game logic that allows a player to solve a generated puzzle. For this purpose the rules are used from right to left; the inputs on the right hand side must be satisfied in order to produce the output(s) on the left hand side. Inputs are satisfied when they are co-located, which could be through use of an inventory system, and have all of the required properties. When the inputs for a rule are satisfied, the action to execute that rule is provided to the player. Only when the player chooses that action is the rule actually executed, i.e., are its inputs replaced by its outputs. While the generator only looked at the first (main) output, each output is important in-game because they indicate which items should be created and/or destroyed.

3 Conclusion

This paper presented a way of procedurally generating narrative puzzles that builds onto what was achieved with the Puzzle-Dice system. The approach can be integrated into existing games, given that the game designer defines puzzle items, rules and game areas as they pertain to his/her game. The difficulty of the puzzles is determined by the designer. As a preliminary evaluation, we have developed a small proof-of-concept game in Unity using the narrative puzzle generator. The game was made with free 3D assets and set in an environment with two areas; a grass field, and a river bank. The areas contained game objects designated as puzzle items, including trees, corn stalks and a well. On a given playthrough, each of these may or may not be used in the puzzle (depending on the puzzle created), but it is always possible to interact with the items. This adds consistency to the world, and can throw the player off in terms of what items he/she needs to complete the puzzles for an area. In future work, we plan to create a bigger game and evaluate the approach through a user study.

References

1. Isaac Dart and Mark J Nelson. Smart terrain causality chains for adventure-game puzzle generation. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 328–334. IEEE, 2012.
2. Barbara De Kegel and Mads Haahr. Procedural puzzle generation: A survey. *IEEE Transactions on Games*, 2019.
3. Clara Fernández-Vara and Alec Thomson. Procedural generation of narrative puzzles in adventure games: The puzzle-dice system. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, page 12. ACM, 2012.
4. Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.