A Constraint-Based Graph Grammar Approach Unifying Level and Playthrough Generation

Seth Cooper[1], Mahsa Bazzaz[1]

[1]Northeastern University

Abstract

Graph grammars are an approach to procedurally generating content for games. However, grammars must often be carefully designed so that generated levels are completable, e.g. keys do not end up behind locks. In this work, we present a graph grammar approach that uses a single unified grammar containing the rules for both generating and playing levels. Such grammars generate not just a level, but an example playthrough of the level. Using a constraint-based application of the grammars can ensure that generated playthrough solves the level. The constraint-based approach is also highly controllable, and can constrain aspects such as the number of times rules are applied, label counts in the generated graph, and matching partial subgraphs. To scale this approach up to larger graphs, we introduce hints into the grammar rules that limit the ranges of variables that can be used for certain nodes and edges. As a motivating example, we reproduce a grammar for generating adventure game missions from Dormans' previous work, extend it with gameplay rules, and use these for flexible item placement while ensuring completability regarding keys and locks.

Keywords

procedural content generation, graph grammars, constraint solving

1. Introduction

Graph grammars are a popular approach to procedural generation of levels for video games [1]. In a graph grammar-based approach, levels are represented as graphs; the grammar provides rules for replacing—or rewriting—certain subgraphs (i.e. arrangements of nodes and edges) with new subgraphs. By repeatedly applying these rules, a graph (and thus level) of the desired complexity can be generated.

As is often the case with level generation, it is critical that levels generated by graph grammars are completable—that is, it is possible for a player to complete the level. This means that, e.g. keys should not be behind their associated locks, special items need to defeat a boss should be collectible before getting to that boss, and so on. A number of techniques have been developed to ensure this, including grammars that prevent such situations [2, 3, 4], generate-and-test [5], and constraint-based approaches [6]. However, these previous approaches require either careful manual design of grammars, or implementing a separate system to check or constrain graphs.

Recent work has explored representing the game mechanics themselves as rewrites of subgraphs, essentially a graph grammar [7]. By using a constraint-based framework, this grammar representation of the game mechanics could ensure that the generated graphs were completable. The level itself was generated by constraints learned from example graphs [8], and simultaneously, the game mechanic grammar generated a step-by-step playthrough demonstrating how the level could be completed. This was done by creating a sequence of graphs, with each corresponding to an application of a mechanic rule, and ensuring that the level is solved by the last step.

In this work, we look at unifying the graph grammar for generating levels and the graph grammar for game mechanics into a single grammar. By unifying these into a single grammar, we can use the previous constraint-based framework to generate graphs that are ensured to be playable in a single constraint problem. We can also use the same representation to describe the rules for generating the graph and ensuring that it is completable.

Joint AIIDE Workshop on Experimental Artificial Intelligence in Games and Intelligent Narrative Technologies, November 10-11, 2025, Edmonton, Canada.

se.cooper@northeastern.edu (S. Cooper); bazzaz.ma@northeastern.edu (M. Bazzaz)

0000-0003-4504-0877 (S. Cooper); 0009-0004-0022-9611 (M. Bazzaz)

For motivation, we looked to Dormans' work on graph grammar generation of adventure game missions [2], recreating it in our system, and augmenting it with mechanic rules. Our constraint-based system allows fine-grained control over generated node labels, rule applications, and even entire subgraphs. While the grammar in Dormans' work was manually constructed to ensure that keys did not end up behind their locks, our system allows flexibility in key and item placement by the generator rules, while relying on the mechanic rules to ensure completability.

At it core, the system we present is an extension to the previous work [7] in which we include both generator rules and mechanic rules, such that some steps in the graph sequence are generating the level and some are of a playthrough ensuring its completability. However, in order to implement Dormans' grammar, several pragmatic improvements to the system were needed. Perhaps the most important of these was for performance, to scale up to larger graphs with more steps in their generation. We call these *grammar hints*, which are additional information about the grammar and individual rules to limit their application and thus the constraint problem to be solved. As one example, since the performance of the constraint satisfaction problem is greatly impacted by the number of potential edges in the graph, each grammar rule can have associated with it an *edge hint*: information about which edges it is allowed to consider based on node IDs; e.g. instead of a rule having to consider any pair of nodes being connected, it may only have to consider situations where the two nodes have relative ids within a certain range, greatly reducing the space of possibilities. Whereas previous work [7] could set such limitations across the entire graph—referred to as *edge setup*—in this work it can be set per-rule. We have found *grammar hints* require additional effort to author, but greatly speed up the generation process.

This work contributes a unified graph grammar approach to both generating levels and ensuring their solvability using constraint solving, a demonstration of performance improvements provided by *grammar hints*, and a demonstration of the controllability of generated graphs afforded by constraint solving.

2. Related Work

Generating completable levels. To ensure that generated levels are completable, there are multiple approaches that are commonly used. One common method is agent-based testing, in which completability is assured by agents attempting to play the levels [9]. Another approach is to add constraints during generation, so that levels always meet some requirements. For example, Nelson and Smith [10] used answer set programming to encode rules about valid paths or item placement. Other systems like Sturgeon combine the rules specified by the designer and learned pattern constraints to guarantee reachability [6]. On the other hand, generate-and-repair techniques have a different approach, in which they ensure completability after the generation process is complete. Autoencoders have been used to automatically fix uncompletable structures by reconstructing levels [11]. More recent methods use explainable AI to detect problematic regions and guide targeted repairs to minimally edit levels until completability is achieved [12].

Generating levels with graph grammars. Graph grammars have been widely explored for level generation. The early work of Adams [13] was one of the earliest works that used a graph grammar (instead of phrase structure grammars) to generate FPS-style dungeons to utilize the power of graphs to represent the complex patterns of connectivity that are found in levels. Following this work, Dormans [2] viewed level design as a sequence of model transformations, and created mission grammars (that included players' missions and tasks) that get mapped to space graphs (that included spatial layout of levels). Font et al. [4] created a two-stage evolutionary method in which a high-level graph of areas is first evolved, and then expanded into detailed room graphs. They created examples of lock-and-key puzzles that were guaranteed to be completable. Londoño and Missura [14] introduced stochastic graph grammars to learn structural properties from Super Mario Bros levels and generate new levels with similar qualities to expert design. Later review works on graph grammars highlighted that grammars naturally capture high-level structural design patterns [15] and explored expansive examples of level

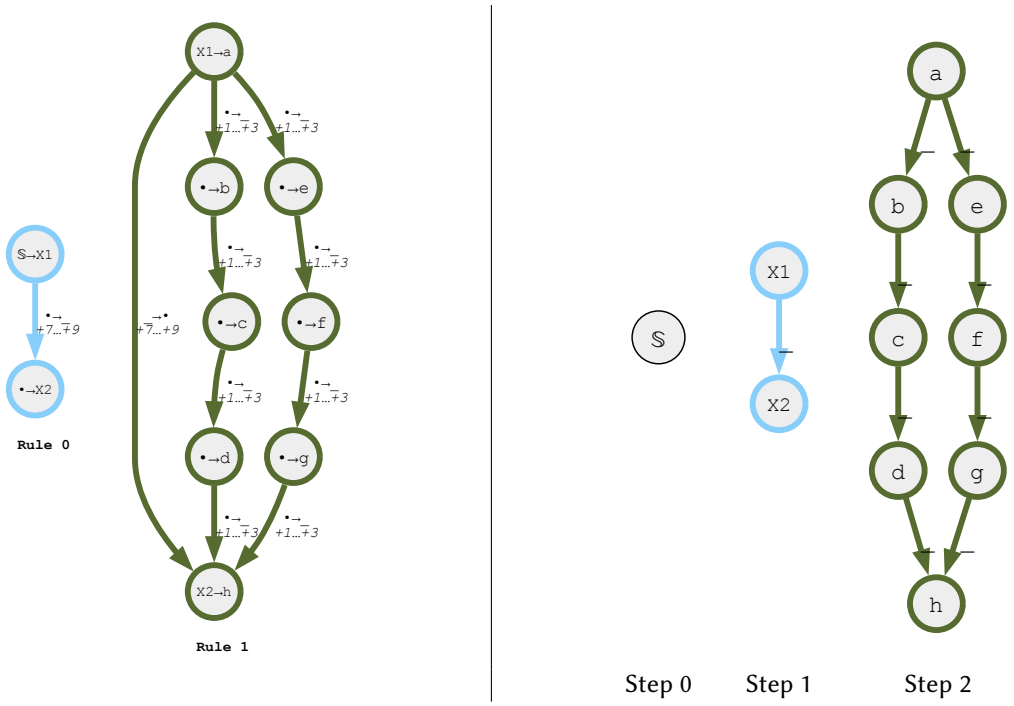

Figure 1: On the left, rules for generating comparison graphs, with edge hints shown on edges; each node and edge has both the left- and right-hand side of the rewrite. On the right, steps created as the generation a comparison graph.

structures and quest generation [16]. Other work has looked at using graph grammars to generate puzzles in educational games [17]. The sophisticated LudoScope level generation system [18, 19] allows a great degree of control over levels generated by graph grammars, for example recipes that can specify the number of times to apply a rule.

3. System Overview

For the most part, the core graph grammar system used in this work is similar to that presented in previous work [7], except that some rules conceptually represent graph generation and some represent gameplay mechanics. We briefly describe the previous system, in which rules represented only gameplay mechanics, and were used to ensure that a generated graph could be completed by those mechanics by generating a playthrough. The rules could rewrite the labels on subgraphs: at each step, a subgraph matching the labels on the left-hand side of the rules could be rewritten as the labels on the right-hand side; each node and edge has a static label, which cannot change from step to step, and a dynamic label, which can change from step to step. Visually, each rule is represented as a graph with the left- and right-hand side labels on each node and edge. These rules are incorporated into a constraint satisfaction problem that creates a sequence of graphs, each representing a step in the solution. Roughly speaking, the constraint problem is set up with a (Boolean) variable for each possible label for each possible node and edge, and for dynamic labels, these are for every step. The graphs in the sequence are constrained such that the labels can only change according to the mechanic rules, and the final graph is constrained to be solved; thus, the created sequence of graphs show how the initial graph can be solved by the mechanics as a playthrough. The problem setup must also be provided with a maximum graph size (in terms of nodes) and maximum number of steps, although steps can be marked as terminal, after which no more changes are made.

As the constraint problem is essentially solving for a label for each node and edge in each timestep, practical considerations about the number of edges that need to be considered can impact performance, as this number grows rapidly as the number of nodes increases. Previously, the number of edges was

reduced by an approach called the *edge setup*. Each node has an ID, from $1 \ldots N$, where N is the maximum number of nodes in the generated graph. The edge setup globally limited which pairs of nodes could potentially have an edge between them based on ID. Though this is configurable, a straightforward edge setup simply only allows edges from a node ID i to nodes up to ID $i + n$.

Here now we describe some of the most salient changes from the previous system, updated for this work. As an illustration, we refer to Figure 1, which shows a small set of generator rules and a resulting graph sequence generated by them. In this simple example, the steps only generate a graph in the final step and do not create a playthrough (we use the term *graph sequence* in general as the steps may be part of generating the graph or part of the playthrough). On the right, there are two generator rules: one that replaces the 𝕊 node with two others, and then another rule that replaces those two with several other nodes. Constraints are added such that final graph must not have any 𝕊, X1 or X2 nodes. On the left the resulting generated graph sequence is shown. Note that the output of the solver is not just the graph from the last step, but the entire sequence of graphs at once.

- *Labels.* In this work we introduced two special labels: the nil label NIL (shown in figures as •), which means that a node or edge is not present, and the start label, 𝕊, which represents the start node. To use the existing system, which learned from example training graphs to create the initial graph (the graph in step 0), an example training graph is supplied with a single 𝕊 node connected to a single NIL node by a NIL edge. The initial graph is constrained to have a single 𝕊 node, with the rest of the nodes and all edges labeled NIL. Only dynamic labels, that can change from step to step, are used—static labels, though present from the previous system, are set to a fixed value and ignored. So, for example, the first rule in the top-left of Figure 1 can rewrite a 𝕊 node, along with a non-present NIL edge to a non-present NIL node (the left-hand side of the rule), with a node labeled X1 to a node labeled X2 by and edge labeled labeled _ (the right-hand side of the rule). This happens in the generated graph sequence in step 2.
- *Edge hints.* Edge hints can be used to specify on a per-rule basis information about which node IDs should be considered for an edge to connect. These can be *relative*, meaning they are deltas to the source node ID (shown in figures as $+n$), or *absolute*, meaning only a specific node ID can be considered (shown in figures as $@n$). For example, in the first rule in the top-left of Figure 1, the $+7 \ldots +9$ means that ID of the node rewritten to X2 must be 7 to 9 greater than the node ID rewritten to X1. This can be used to greatly limit the number of edges in consideration for this rule.
- *Node hints.* Node hints can be used to specify on which node IDs and in which steps node labels can be present.
- *Rule hints.* Rule hints can be used to specify in which steps a rule can be applied.
- *Skipped rule application.* Whereas in the previous system one rule had to apply each step (until the terminal step), a specific step can allow no rules to be applied.
- *Simultaneous rule application.* Multiple rules can be allowed to be applied in the same step, as long as they do not involve any of the same nodes or edges. This can allow the number of steps to be reduced, making the problem size smaller.
- *Required rule application.* In conjunction with skipped and simultaneous rule application, a step can require all rules that can be applied to be applied.

4. Comparison with Prior Approach

First, we compare the previous graph-global *edge setup* method the new per-rule *edge hints* in terms of performance. Using the simple comparison grammar from Figure 1(left), we generated graphs of increasing maximum size allowing 3 steps. Note that the maximum size just sets an upper limit on the number of nodes and edges that can be in the graph and therefore must be considered; however, this simple grammar does not allow anything beyond what is shown in Figure 1(right) to be generated. When generating with *edge hints*, those in the figure were used. For *edge setup*, the hints in the figure were not used and a *band-9* edge setup was used, where a node ID i could connect to node ID up to

Max Size	Edge Setup	Edge Hints
10	1.67s	1.39s
12	2.57s	1.58s
14	4.64s	1.67s
16	9.30s	1.74s
18	18.62s	1.81s
Increase between sizes	$\sim 1.85\times$	$\sim 1.06\times$

Table 1
Times to generate comparison graphs with different maximum sizes using either edge setup or edge hints.

$i + 9$. Each method was run 25 times, and the time taken to generate graph sequences was averaged. For the constraint solver, we use a portfolio of three instances of PySAT's [20] gluecard41 SAT solver, in turn based on Glucose [21], MiniCARD [22], and MiniSAT [23].

The resulting times as shown in Table 1 show how *edge hints* have a greatly reduced increase in time as the maximum size of the graph grows. In particular, the rough multiplier of times between each increase is much smaller.

5. Generating Missions

As our motivating example, we sought to recreate Dormans' [2] adventure game mission grammar in our constraint-based system, and augment it with mechanic rules. As it it quite large, we start with a simplified version, referred to as mini-Dormans, to explain how the new features of the system were used to implement it.

The grammar is shown in Figure 2. *Edge hints* are used throughout; *rule hints* are used to limit the steps in which rules can apply. *Node hints* are used to make sure all non-terminal labels are gone by the step the generate rules are done applying. *Node hints* are used, in conjunction with *edge hints* to implement an inventory system. *Simultaneous rule application* is allowed, and *skipped/required rule application* is used to force the application of rules related to edge traversability. By the end of the graph sequence, the goal node label must be gone (due to the player moving onto it).

The first generate rule creates two corridors *C* each followed by a lock *l*; it can only be used between steps 0 and 1. The next two generate rules can replace these corridors *C* with slots *SL* in sequence or parallel, and the next two replace a slot *SL* with either a key *k* or nothing *n*; these can only apply between steps 1 and 3. At this point in the steps the level has been generated. The next rules are meant as a proxy for spatial information in Dormans' grammar to set up for the mechanic rules: in the original grammar, double edges indicate part of the mission should be spatially behind another when turned into a space. Since we are not generating the space, we instead make all the other edges leading to a node that should be behind a node not travesible. This is accomplished by a required rule application rewriting all incoming single dash - edges to a node with an incoming equal = edge to be an untraversible x between steps 3 and 4, and then marking all the remaining edges as a traversible period . between steps 4 and 5 (the box □ in a node means it that node is ignored in the rule). At this point the playthrough begins: the next rule sets up the player @, with an inventory, which uses *node hints* and *edge hints* to use the label at node ID 0 (k : _, k : 1, k : 2) to count the number of keys k the player has collected. The remaining mechanic rules allow the player to move through the graph, collecting keys and using them to open locks, and reach the goal.

Top control the generated level, constraints are added so that sequence and parallel corridor rules are used once each, and step 3 has 2 keys. The maximum size is 15 nodes and maximum steps is 20. The same solver setup is used as above, and 25 graph sequences were generated.

Figure 3 shows a generated graph sequence using this grammar and hints. The first several steps use generate rules. Step 0 is the initial graph, with just a 𝕊 node. The next three steps create the graph structure and place the keys. Step 4 marks two edges incoming to the first lock as not traversible, requiring the player to use the remaining edge later, and step 5 marks the rest of the edges as traversible.

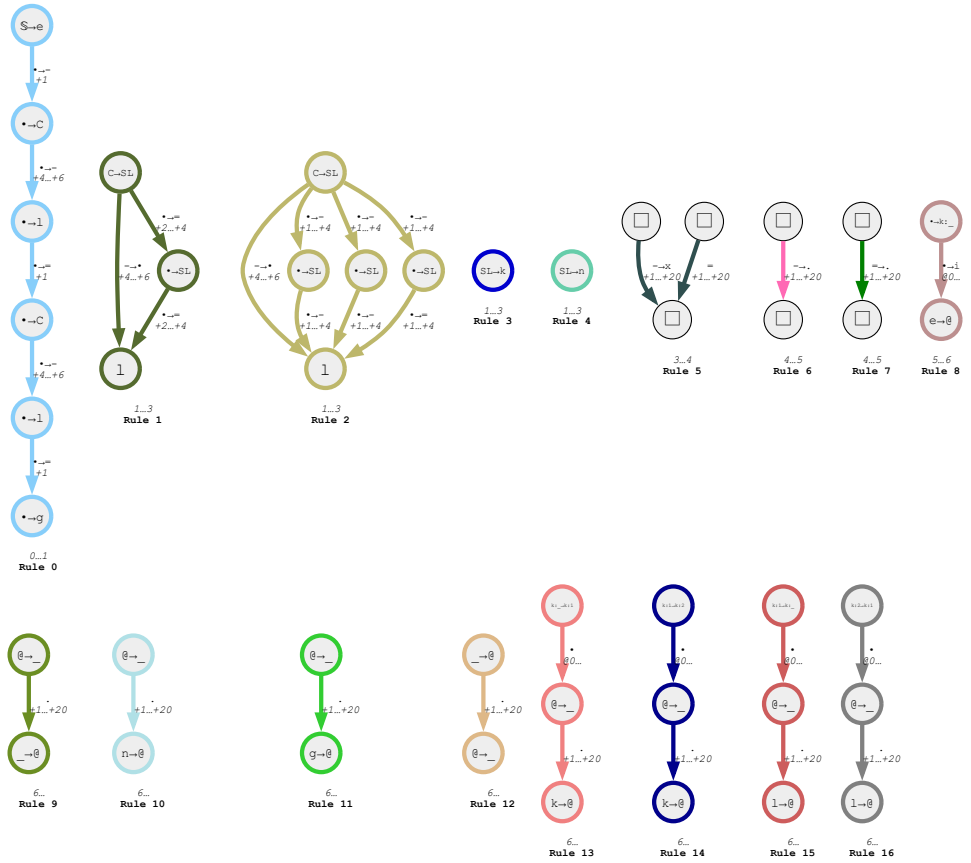

Figure 2: Rules for a mini-Dormans mission graph with slots and gameplay rules.

Step 6 begins the application of the mechanic rules, with the player appearing with an empty inventory. From there the player moves through the graph, picking keys and using them to open locks, and eventually reaching the goal. The player notably does not take the most direct path. Due to unifying the generate rules and mechanic rules into a single grammar, keys are placed in the right locations for the player to open the locks.

We applied the same conceptual approach to Dormans' full mission grammar [2] to explore the controllability and applications of the unified grammar approach in various configurations. As the grammars and generated graph sequences are quite large, here we describe their setup and summarize their properties, while making more information available at https://osf.io/zxpky/. A partial graph sequence for the most complex, *Dormans-slots*, is shown in Figure 4.

- *Dormans*. This configuration is just generating a Dormans mission graph, with maximum size 45 and maximum 8 steps.
- *Dormans-lock-multi*. Same as *Dormans*, but there must be exactly one multi-lock `lm` node. This could be useful if a designer would like specific nodes present in the generated graph.
- *Dormans-rule-count*. Same as *Dormans*, but the rules for creating a 3 and 5 long linear chain must be used exactly once each. This could be useful if a designer would like specific rules to be used to generate the graph.
- *Dormans-match*. Same as *Dormans*, but a given substructure consisting of 6 nodes and 4 edges must be matched in the final graph. This could be useful if a designer likes part of a generated graph but would like to try variations of the rest of it.
- *Dormans-slots*. Similar to *mini-Dormans* but with the complete Dormans grammar. Replaces the specific key and item placement rules of the original generate grammar with slot nodes, which can then be filled in with items and keys; augments the generate grammar with a mechanic grammar and uses this to ensure the generated graph is completable. Uses a maximum size of 45

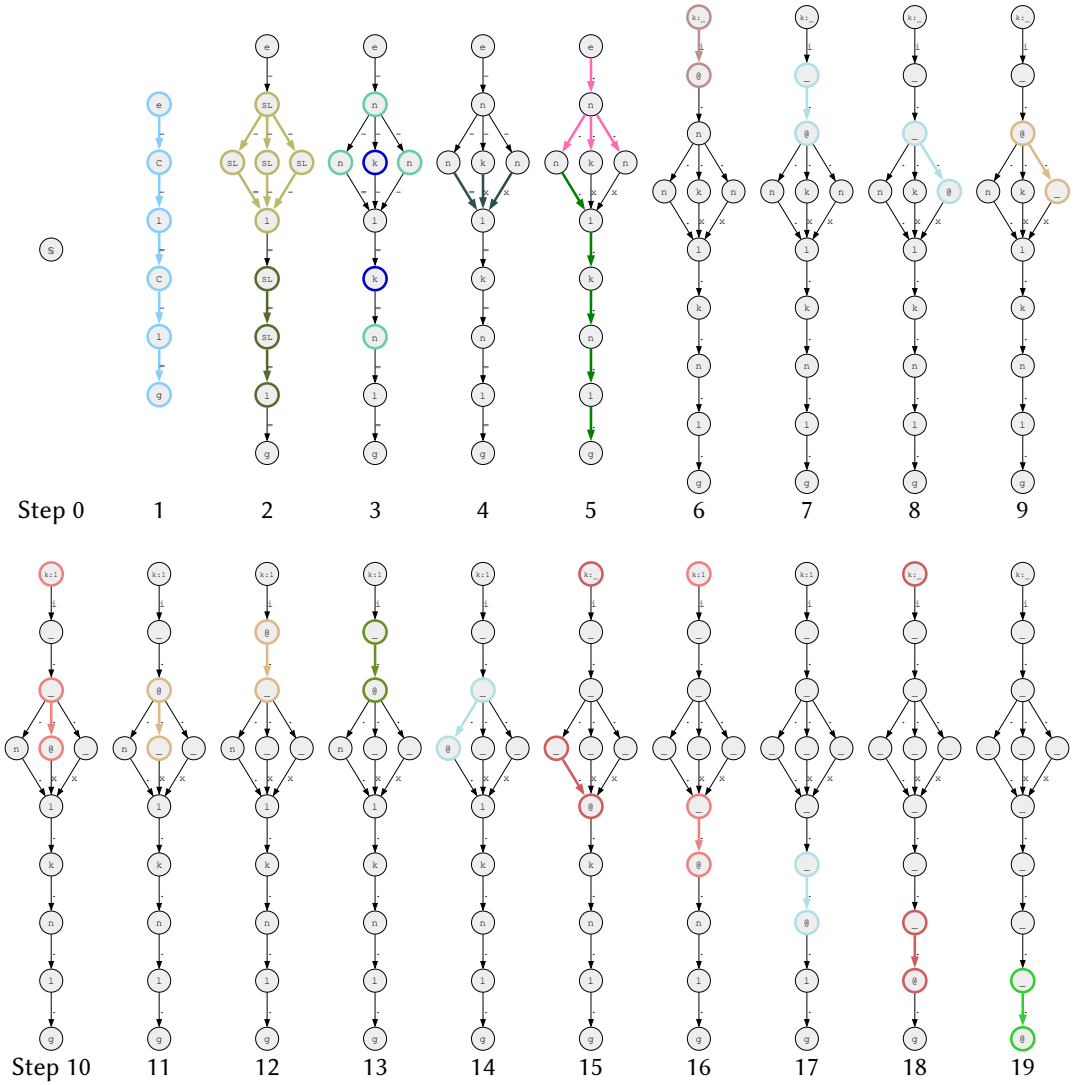

Figure 3: Generated graph sequence for mini-Dormans mission graph.

Configuration	Time (Med)	Time (Max)	Steps (Med)	Steps (Max)	Nodes (Med)	Nodes (Max)	Edges (Med)	Edges (Max)
mini-Dormans (gen./mech.)	2.82s	3.21s	20.0	20.0	11.0	11.0	12.0	12.0
Dormans (gen.)	6.07s	6.39s	8.0	8.0	26.0	30.0	28.0	32.0
Dormans-multi-lock (gen.)	6.05s	6.93s	8.0	8.0	25.0	30.0	27.0	32.0
Dormans-rule-count (gen.)	5.99s	6.61s	6.0	8.0	25.0	32.0	25.0	32.0
Dormans-match (gen.)	6.01s	6.26s	8.0	8.0	29.0	33.0	29.0	33.0
Dormans-slots (gen./mech.)	585.38s	1625.65s	35.0	35.0	29.0	35.0	31.0	37.0

Table 2
Summary of different mission generation configurations.

and maximum 35 steps. In the step where the level is generated, there must be 1 quest item, 1 item test, 4 keys, 1 multi-lock, 1 single lock, 1 final key, and 1 final lock.

For each, the same solver setup is used as above, and 25 graph sequences were generated. Summary statistics of timing, sizes, ans steps are given in Table 2. Some observations of note are that to just generate the graphs on this scale takes only a few seconds, and adding in a few additional constraints on the generated graph does appear to increase the time needed much. However, augmenting with mechanics rules greatly increases the time to generate graph sequences, around 2 orders of magnitude on average. This may make it prohibitive still for online or interactive generation.

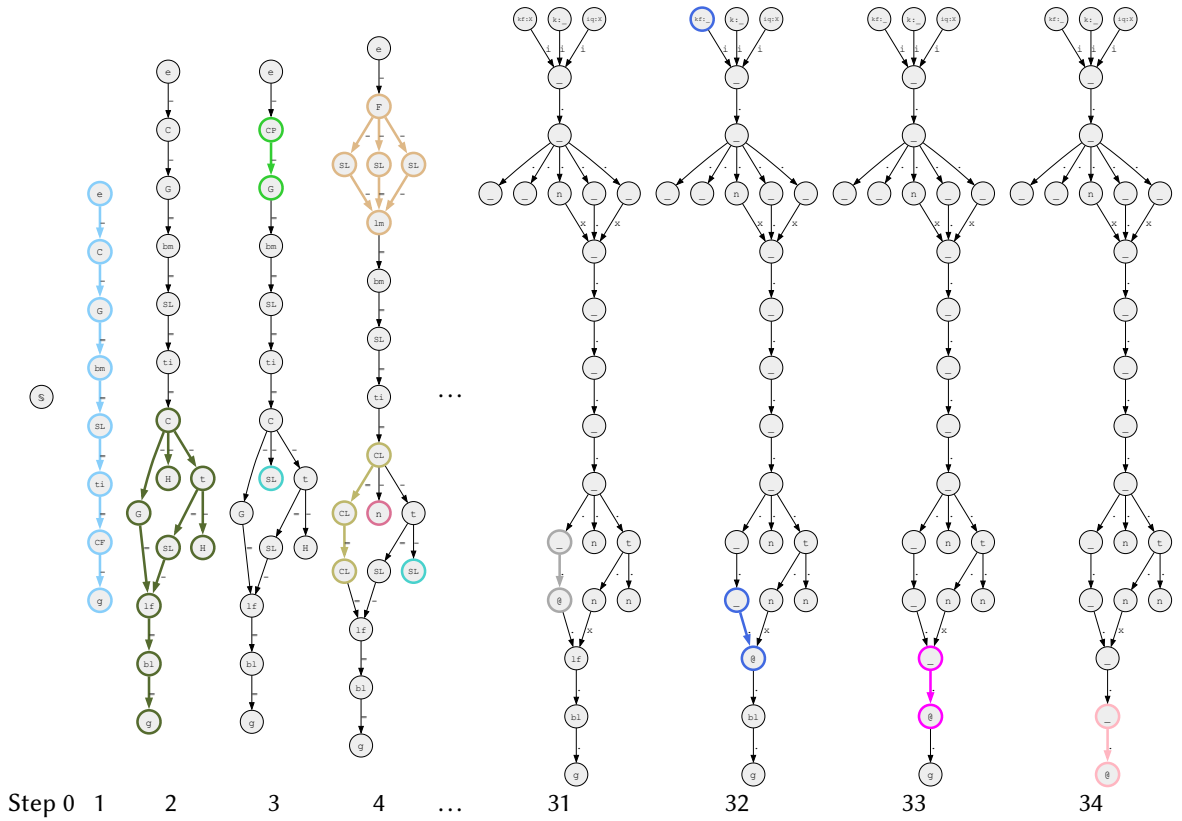

Figure 4: Selected steps of the graph sequence for a generated Dormans-slots mission graph, showing both generate and mechanic rules.

6. Conclusion

In this work we have presented an approach to generate completable levels using graph grammars. The approach combines rules for generating graphs with rules for running game mechanics on graphs into a single unified grammar. Using a constraint-based system that generates a graph sequence, where the grammar constrains changes between graphs in the sequence and the level must be solved by the final step, the mechanic rule ensure that the level generated by the generate rules can be completed. We describe some practical improvements over the previous system and show their beneficial impact on timing. We also recreate and augment the adventure game mission grammar of Dormans and show how the constraint-based approach can provide controllability of generated graphs.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools in preparing this work.

References

[1] N. Shaker, J. Togelius, M. J. Nelson, Procedural Content Generation in Games, Springer International Publishing, 2016.

[2] J. Dormans, Adventures in level design: Generating missions and spaces for action adventure games, in: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, 2010.

[3] J. Dormans, S. Bakkes, Generating missions and spaces for adaptable play experiences, IEEE Transactions on Computational Intelligence and AI in Games 3 (2011) 216–228.

[4] J. M. Font, R. Izquierdo, D. Manrique, J. Togelius, Constrained level generation through grammar-

based evolutionary algorithms, in: G. Squillero, P. Burelli (Eds.), Applications of Evolutionary Computation, Lecture Notes in Computer Science, 2016.
[5] S. Snodgrass, S. Ontañón, Controllable procedural content generation via constrained multi-dimensional Markov chain sampling, in: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, 2016, pp. 780–786.
[6] S. Cooper, Sturgeon: tile-based procedural level generation via learned and designed constraints, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 18 (2022) 26–36.
[7] S. Cooper, M. Bazzaz, Sturgeon-MKIV: constraint-based level and playthrough generation with graph label rewrite rules, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 20 (2024) 13–24. Number: 1.
[8] S. Cooper, Sturgeon-GRAPH: Constrained Graph Generation from Examples, in: Proceedings of the 18th International Conference on the Foundations of Digital Games, FDG '23, 2023, pp. 1–9.
[9] J. Togelius, G. N. Yannakakis, K. O. Stanley, C. Browne, Search-based procedural content generation: a taxonomy and survey, IEEE Transactions on Computational Intelligence and AI in Games 3 (2011) 172–186.
[10] M. J. Nelson, A. M. Smith, ASP with applications to mazes and levels, in: N. Shaker, J. Togelius, M. J. Nelson (Eds.), Procedural Content Generation in Games, Springer International Publishing, 2016, pp. 143–157.
[11] R. Jain, A. Isaksen, C. Holmgård, J. Togelius, Autoencoders for level generation, repair, and recognition, in: Proceedings of the ICCC workshop on computational creativity and games, volume 9, 2016.
[12] M. Bazzaz, S. Cooper, Guided game level repair via explainable AI, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 20 (2024) 139–148.
[13] D. Adams, Automatic generation of dungeons for computer games, Bachelor's thesis, University of Sheffield, UK, 2002.
[14] S. Londoño, O. Missura, Graph grammars for Super Mario Bros levels, in: Sixth FDG Workshop on Procedural Content Generation, 2015.
[15] N. Shaker, A. Liapis, J. Togelius, R. Lopes, R. Bidarra, Constructive generation methods for dungeons and levels, in: Procedural Content Generation in Games, Springer, 2016, pp. 31–55.
[16] J. Togelius, N. Shaker, J. Dormans, Grammars and l-systems with applications to vegetation and levels, in: Procedural Content Generation in Games, Springer, 2016, pp. 73–98.
[17] J. Valls-Vargas, J. Zhu, S. Ontañón, Graph grammar-based controllable generation of puzzles for a learning game about parallel programming, in: Proceedings of the 12th International Conference on the Foundations of Digital Games, 2017, pp. 7:1–7:10.
[18] J. Dormans, Engineering emergence: applied theory for game design, Research HvA, graduation external, Universiteit van Amsterdam, Amsterdam, 2012.
[19] D. Protsenko, J. Dormans, R. van Rozen, Maintenance in procedural level design: lessons from Ludoscope, in: Proceedings of the 20th International Conference on the Foundations of Digital Games, 2025, pp. 1–4.
[20] A. Ignatiev, A. Morgado, J. Marques-Silva, PySAT: a Python toolkit for prototyping with SAT oracles, in: Theory and Applications of Satisfiability Testing – SAT 2018, 2018, pp. 428–437.
[21] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: Proceedings of the 21st International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009, pp. 399–404.
[22] M. H. Liffiton, J. C. Maglalang, A cardinality solver: more expressive constraints for free, in: Theory and Applications of Satisfiability Testing – SAT 2012, 2012, pp. 485–486.
[23] N. Eén, N. Sörensson, An extensible SAT-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, number 2919 in Lecture Notes in Computer Science, 2003, pp. 502–518.