Procedural Content Generation using Behavior Trees (PCGBT)

Anurag Sarkar and Seth Cooper
Northeastern University
sarkar.an@northeastern.edu, se.cooper@northeastern.edu

CEUR-WS.org/Vol-3217/paper11.pdf

Abstract

Behavior trees (BTs) are a popular method for modeling NPC and enemy AI behavior and have been widely used in commercial games. In this work, rather than use BTs to model game *playing* agents, we use them for modeling game *design* agents, defining behaviors as content generation tasks rather than in-game actions. Similar to how traditional BTs enable modeling behaviors in a modular and dynamic manner, BTs for PCG enable simple subtrees for generating parts of levels to be combined modularly to form complex trees for generating whole levels as well as generators that can dynamically vary the generated content. We refer to this approach as Procedural Content Generation using Behavior Trees, or PCGBT, and demonstrate it by using BTs to model generators for *Super Mario Bros.*, *Mega Man* and *Metroid* levels as well as dungeon layouts and discuss several ways in which this paradigm could be applied and extended in the future.

Introduction

Behavior trees (BTs) (Isla 2005; Champandard 2007) are a commonly used technique and framework for modeling agent behaviors in games and have seen widespread use in a large number of commercial games for defining NPC and enemy AI. The popularity of BTs stems from them enabling such behaviors to be implemented in a modular and reactive manner. That is, simpler actions and behaviors can be combined to define more complex agent behaviors and different BT branches corresponding to different behaviors can be selected for execution based on various runtime conditions, thus enabling agents to react dynamically during gameplay. In addition to game AI, in recent years, BTs have also been widely employed in robotics for defining behaviors of robot controllers (Colledanchise and Ögren 2017), thus demonstrating the general usefulness of BT-based methods.

Hence in this paper, with a view to bring these benefits to bear on generative methods, we propose the use of BTs for procedural content generation (PCG). More specifically, we repurpose BTs to model the behavior of game *design* agents rather than game *playing* agents. By replacing NPC actions (e.g. Cover and Shoot) with design actions (e.g. Generate Segment and Connect Rooms), we obtain BTs that are procedural level generators and are also modular, reactive and interpretable like their traditional BT counterparts. We refer to this framework as Procedural Content Generation using BTs (PCGBT) and BTs capable of doing so as PCG-BTs. We demonstrate this approach by developing PCG-BTs for generating *Super Mario Bros.* and *Mega Man* levels as well as dungeon layouts. We also show that BTs can be used to describe a generic level generator which can then be instantiated for different games such as *Mega Man* and *Metroid*. Further, we show that subtrees corresponding to different games could be combined to form a super-tree capable of generating levels that blend portions of different games. Finally, we conclude with an extensive discussion on future applications and implications of this PCGBT framework. Our work thus contributes, to our knowledge, the first application of BTs for explicitly defining game design agents, and thereby enabling procedural content generation.

Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Background

Most prior BT-related research has focused on using BTs for modeling game playing agents and controllers rather than PCG. Evolving BTs for controlling agents has seen much work with Lim, Baumgarten, and Colton (2010) focusing on the real-time strategy game DEFCON and Perez et al. (2011) and Nicolau et al. (2016) evolving controllers for the Mario AI framework. Instead of evolution, Robertson and Watson (2015) identified patterns in action sequences for producing BTs for *StarCraft* while Glavin and Madden (2014) used a reinforcement learning-based approach for building BTs for bots in *Unreal Tournament 2004*. In recent years, robotics has arguably seen more BT research than games. Comprehensive surveys of BTs in robotics have been given by Iovino et al. (2020) and Colledanchise and Ögren (2017).

In using tree-like structures to model generators, our work overlaps with graph and grammar-based generative approaches. Shaker et al. (2012) used grammatical evolution to evolve graphs of Mario levels while Dormans (2010) and Karavolos, Bouwer, and Bidarra (2015) both utilized mission and space graphs for dungeon generation. Relatedly, graph grammars have been utilized for generating levels for Mario (Hauck and Aranha 2020), puzzle games (Valls-Vargas, Zhu, and Ontañón 2017), dungeon crawlers (Linden, Lopes, and Bidarra 2013) and educational games (Jemmali et al. 2020). Rather than graphs, we use an explicit BT formulation for defining level generators. The recently intro-

duced field of PCG via Reinforcement Learning (PCGRL) (Khalifa et al. 2020; Nam and Ikeda 2019) also seeks to produce game design agents which perform generative actions. Instead of RL, we use BTs to model design agents.

Method

In this section we describe our approach. We first describe how BTs work and then detail how we adapt them for PCG.

Behavior Tree (BT)

A BT is a directed tree structure consisting of a root node where execution starts, a set of internal (or control flow) nodes which control the flow of execution and a set of leaf nodes which define the actions to be executed. The root controls execution by propagating signals called ticks at a predetermined frequency. Each node receives ticks from its parent node and propagates it to one or more of its child nodes. A node executes only if it receives a tick from its parent. Each child, after finishing execution, returns one of 3 statuses to its parent—*Running*, *Success* or *Failure*—which in turn determines the status of the parent. There are typically 4 types of control flow nodes:

- *Sequence* - execute all their children in order from left to right until one fails. They return *Success* only if all of their children also succeed.
- *Selector* - execute all their children in order from left to right until one of them succeeds. They return *Failure* only if all of their children fail.
- *Parallel* - execute their children simultaneously and succeed if a predetermined number of children succeed.
- *Decorator* - modify their child behavior node.

Leaf nodes are typically of 2 types:

- *Action* - execute commands corresponding to the most low-level behaviour being modeled.
- *Condition* - used to check conditions and return success or failure accordingly, corresponding to true and false.

More working and implementation details about BTs can be found in Champandard and Dunstan (2019) and Colledanchise and Ögren (2017). Our PCGBT approach is enabled by having *Action* nodes perform level design tasks rather than NPC/enemy behaviors. In this work, we only used *Sequence* and *Selector* control flow nodes since they were sufficient to demonstrate the application of PCG-BTs but *Parallel* and *Decorator* nodes could also be incorporated in the future. Further, all nodes in this work returned either *Success* or *Failure* though in the future these could return *Running*, while waiting for playability checks to complete, for example. In general, all features of traditional BTs would apply to PCG-BTs but due to limited space and wanting to demonstrate PCG-BTs for multiple games, we focused on a subset of BT features. In all cases, unless indicated, *Condition* nodes returned *Success* or *Failure* at random, i.e. if a randomly generated float was less than 0.5, we returned *Success*, else *Failure*. The result was then propagated back up and branches chosen accordingly, as per typical BT rules. In all figures in this work, *Sequence*, *Selector*, *Action* and *Condition* nodes are represented as orange rectangles, blue octagons, grey ellipses and grey diamonds respectively.

BTs may optionally also use a blackboard. In its simplest form, a blackboard is a key-value store that is globally accessible by all nodes in the tree and used to store information that may be useful when making decisions. We make extensive use of blackboards in our BT implementations.

Level Generation using BTs

Analogous to how traditional BTs work with a library of scripted actions which are then combined in different ways to define different behaviors, the PCG-BTs that we use work with a library of level segments, which are combined to generate different levels. That is, executing an action node in a PCG-BT places a level segment of a certain type as defined by that node with different action nodes corresponding to placing different types of segments. We also make use of global blackboard variables to keep track of the position in the level where the next segment should be placed. During execution, each action node updates this position after placing its node, with updates varying from game to game, e.g., in Mario, updates increment the x-coordinate by 1 whereas in Mega Man, this depends on if the next segment should be placed above, below or to the right of the current segment. The blackboard is particularly useful for PCGBT as it can store globally accessible information useful for aiding the generative process such as designer preferences for selecting specific node types, player data to determine which branches to execute, the state of the current game world, etc.

To illustrate how we use BTs to model design agents, we go through a step-by-step worked example for generating a Mario level, as shown in Figure 1. In this work, we use Mario design patterns as defined by Dahlskog and Togelius (2012) as the building blocks of levels with different action nodes corresponding to the placement of segments with different design patterns. We start with a very simple tree with 1 leaf node that places a *Stair Up* segment. This is then combined with another leaf that generates a *Stair Valley* to give us a tree that generates a section consisting of 2 stair segments. We define a separate tree that generates a section of 3 *Pipe Valleys*. Next, we combine these 2 separate BTs to obtain a larger tree that can now generate a stair section followed by a section of pipe valleys. This larger BT could be further expanded in either direction as shown in the next examples where we add an enemy section to the right and then a section of multiple paths to the left. This example demonstrates how we can build up BTs for generating full levels by combining smaller BTs that generate level sections, thus yielding modular level generators, similar to how traditional BTs enable modeling agent behavior in a modular fashion. In the next section, we demonstrate applications of BTs for developing level generators for a number of different games.

Applications

In this section, we demonstrate applications of PCG-BTs for several games. In all cases, we used levels from the Video Game Level Corpus (Summerville et al. 2016). BTs were implemented using py_trees[1].

[1] https://github.com/splintered-reality/py_trees

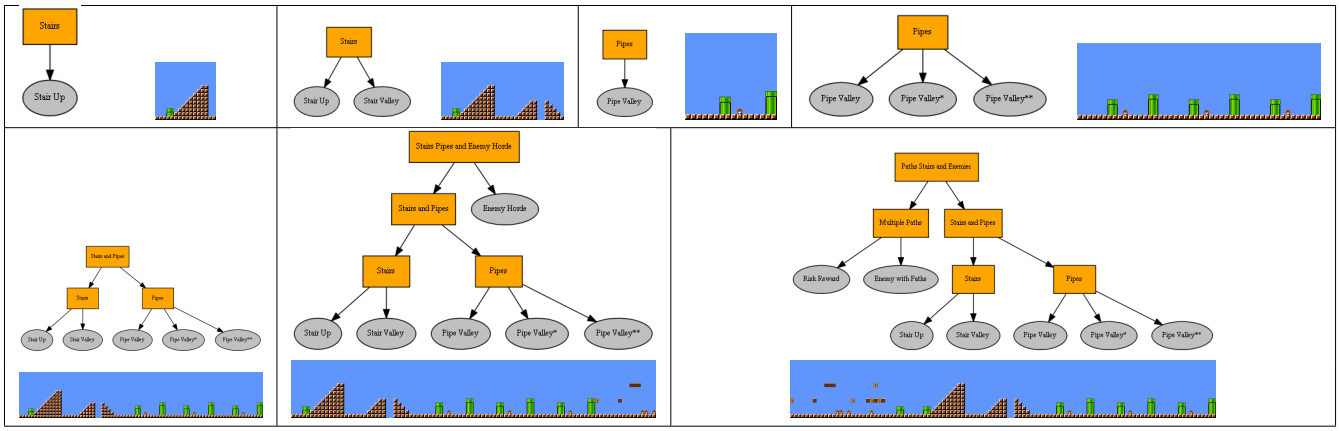

Figure 1: Worked example depicting a behavior tree modeling the process for generating a Mario level.

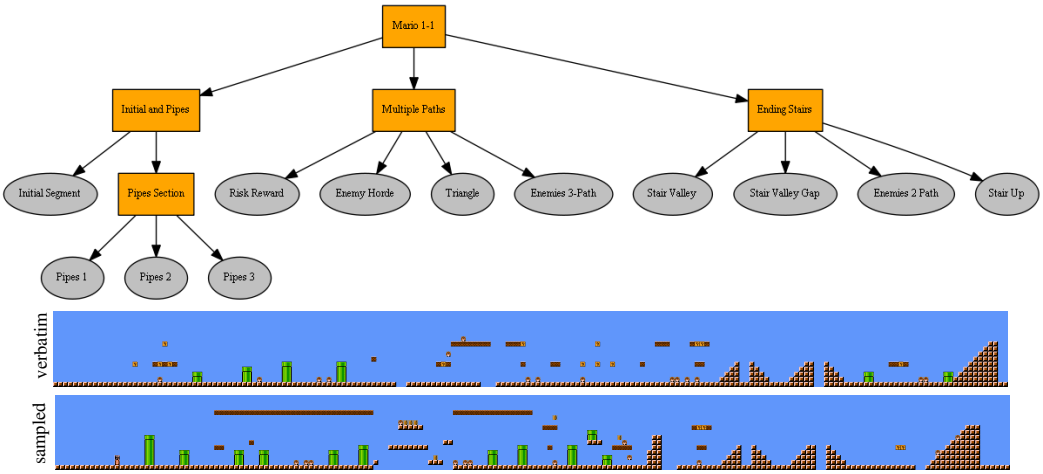

Figure 2: SMB 1-1 PCG-BT along with a recreation of the original level (verbatim) and a variant (sampled).

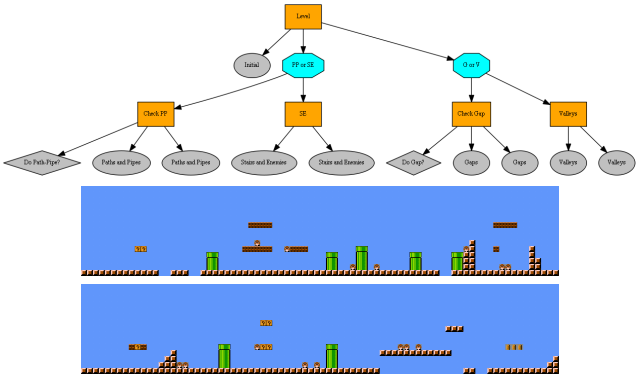

Figure 3: SMB PCG-BT and example generated levels.

Super Mario Bros. (SMB)

For SMB, we created the library of level segments for the BTs by extracting non-overlapping 14x16 segments from VGLC levels. Segments were then padded with 1 row at the top for consistency in combining with segments from other games as described later. We manually categorized the segments based on the design patterns within them. Action nodes were defined to take one or more design patterns as parameters and sample a segment from the set of segments that contain at least one of the desired pattern(s). Different action nodes could then be combined to form subtrees capable of generating level sections with different patterns, which in turn could be combined to form larger trees capable of generating entire levels. To keep track of the level during generation, we maintained a globally accessible dictionary on the blackboard, mapping (x,y) coordinates to the segment generated for that location. At the end of execution, each action node incremented x by 1 and added the sampled segment to this dictionary. We show two example BTs with the first recreating and also generating variations of level 1-1. We model this level as a BT with 3 top-level sequence nodes—the first generates the initial section of pipes, the second generates the middle section of enemies and multiple paths and the third generates the final stair-filled section. The BT and 2 sample levels are shown in Figure 2. The first 'ver-

batim' sample reproduces the original level by simply copying the corresponding segment rather than sampling, i.e. we hard-coded action nodes to pick the exact segment that appears in the corresponding position in the original level. In the second 'sampled' example, the action node samples a segment with the same design pattern in the corresponding position in the original level. These examples show that BTs can be used to sufficiently model generators capable of producing existing levels as well as their variations. The second example BT and the resulting sampled levels generated using it are shown in Figure 3. This example demonstrates the use of *Selector* nodes for picking between two branches of execution. The first decides whether to generate a section of paths and pipes or stairs and enemies while the second decides between a section with gaps or valleys. The decision to branch is made in the *Do Path-Pipe?* and *Do Gap?* condition leaf nodes. For these examples, these choices were made randomly but one could have selection probabilities be weighted by designer preferences to guide selection. Also, in a dynamic setting, player data could be used to make the choice. For example, in *Do Gap?*, we could check if the player has already lost a number of lives and generate valleys instead of gaps to reduce difficulty. We note that these BTs generated the entire level in one tick from the root rather than a repeating loop of ticks like BTs employed at runtime.

Mega Man (MM)

For MM, we constructed a similar library of 15x16 segments from the VGLC but grouped segments based on the directions in which they were open rather than design patterns, since this information is crucial for combining segments such that the overall level is traversable, e.g. a segment that is open above needs the next segment to be open below. Each segment was tagged with U, D, L and/or R to indicate it being open above, below, to the left and/or to the right respectively. Similar to SMB action nodes, MM action nodes sampled segments based on the desired opening orientations. Also, since MM levels do not exclusively progress to the right like SMB, each action node also performed a check to see if there was an opening between the current segment being sampled and the previously sampled segment along the edge adjoining them. This ensured generating segments such that the resulting level has a traversable path through it. Similar to the SMB BT, we maintained a level dictionary on the blackboard in addition to a reference to the previous segment for performing the aforementioned check. After sampling its segment, each action node updated both the x and y coordinates based on the position of the next segment. For space, we only show one MM BT in Figure 4. This generates an initial horizontal section then decides between generating a horizontal or a vertical section followed by a middle horizontal section, a vertical section and a final horizontal section. Note also that each vertical section is generated by a *Selector* that decides whether the section will progress upward or downward. Similar to SMB, these decisions were made at random but could be made based on designer preferences or at runtime based on player behavior. Also like SMB, the BT here used a single root tick to generate the entire level.

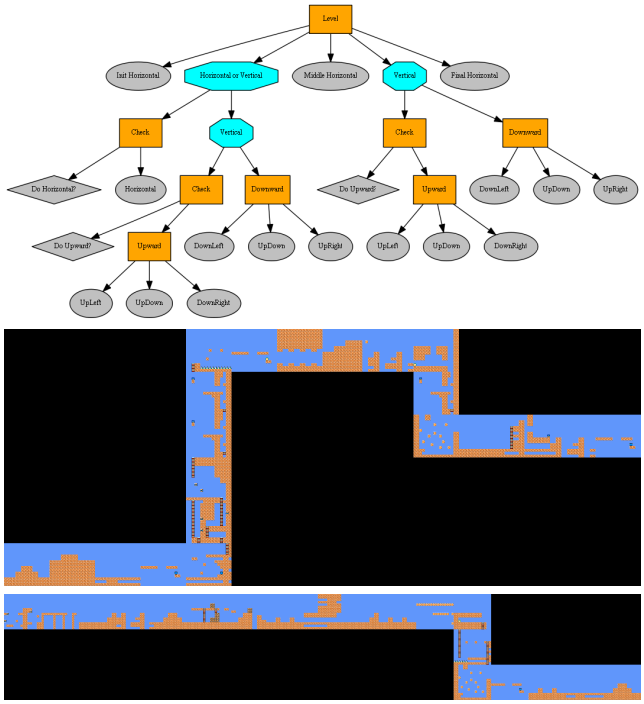

Figure 4: MM PCG-BT and example generated levels.

The SMB and MM BTs both produced levels in only a linear manner due to the original games lacking loops and progressing linearly, necessitating a linear generation approach. PCG-BTs can also generate levels that progress in multiple directions and have paths that loop around, as we'll see next.

Dungeons

Having demonstrated BTs for platformer level generation using a segment-by-segment generative approach, we wanted to test BTs for generating levels for other genres. Thus, we implemented BTs for generating layouts for dungeon crawlers such as *The Legend of Zelda*. We developed a BT to model a simple layout generation algorithm which begins by placing a starting room that is closed on all sides. Then in each iteration, a random closed side is chosen. If there is no room next to it, a room is placed there with all its sides closed. The two rooms are then connected by setting the adjoining edge to be open. This is repeated until the desired number of rooms has been generated. To visualize the generated dungeons, we extracted the 11x16 rooms from Zelda levels in the VGLC. Similar to MM, we tagged each room with the direction in which they had doors. For each room generated by the BT, we placed a sampled Zelda room with doors in the corresponding directions. The BT and sample dungeons are shown in Figure 5. Unlike previously, the BT here generates one room per tick and executes a loop of ticks until all rooms have been generated. For this, we store a flag on the blackboard to indicate if the start room has been generated, initialized to false. During the first tick, the start *Selector* executes, generates the starting room and disables the flag so that on subsequent ticks this check fails. Note that

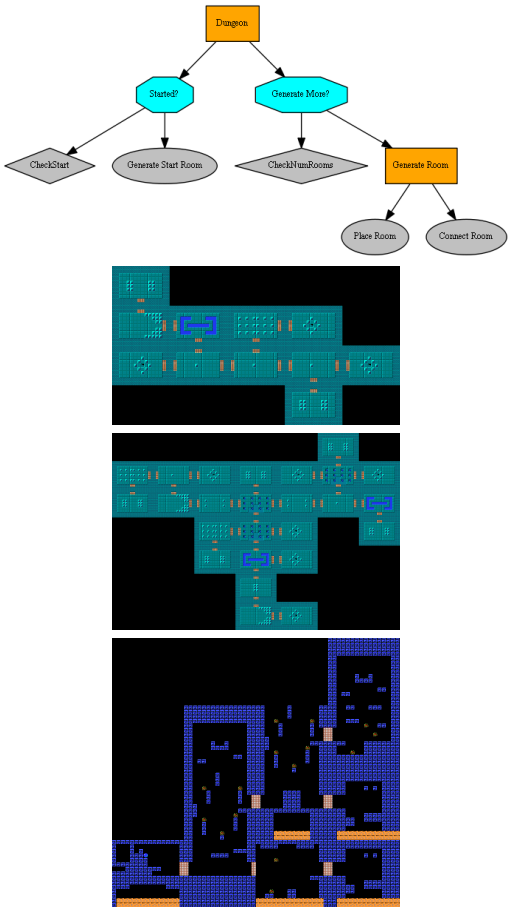

Figure 5: Dungeon Layout PCG-BT and example generated dungeons and Metroid level

the SMB and MM BTs could also have been implemented using a loop of ticks and the dungeon could have been implemented using a single tick, demonstrating the different ways in which BTs could be used for level generation.

Generic and Blending

In addition to modeling specific games, we envisioned crafting game-agnostic BTs which could be instantiated using levels from different games. This only requires that sections generated by the control flow nodes and segments produced by the action nodes are compatible across multiple games. We have seen two such BTs. While the SMB BT is not generic since it utilizes Mario-specific design patterns, the MM BT can generate levels for any platformer consisting of vertical and horizontal sections and segments with openings in 4 directions. Similarly, the dungeon layout BT could be used for any game with interconnected segments. *Metroid*, being a platformer and having a sprawling interconnected game world, satisfies both criteria. Similar to MM, we used 15x16 segments from full Metroid levels from the VGLC. Segments were similarly labeled based on directionality with slight modifications where needed (e.g. no Metroid segment had just Down and Right openings without Up also being open, thus we mapped the DR action node to sample UDR in this case). An example Metroid level generated using the dungeon BT is shown in Figure 5. A generic BT adapted from the prior MM BT is shown in Figure 6 along with an example MM level and Metroid level, both generated using it. Finally, given that subtrees corresponding to different level sections could be combined to form larger trees for generating whole levels, one could combine a tree for generating levels of one game with a tree for generating levels of another to produce a more complex tree that can generate levels from different games taken together i.e., perform game blending (Gow and Corneli 2015). Such a blending BT is shown in Figure 7. It consists of an initial *Sequence* for generating an SMB section, followed by a *Selector* for generating either a horizontal or a vertical MM section, followed by a *Sequence* for generating a horizontal and then a vertical section of Metroid. Playability is maintained similar to MM action nodes i.e. by checking adjoining edges of successive segments and resampling if no path exists.

Discussion and Future Directions

Our results suggest that BTs can be repurposed for modeling design agents and yield modular and interpretable generators—modular in that subtrees generating level sections can be recombined with other subtrees to produce a variety of BTs and interpretable in that BTs indicate why and how the generators produce the levels they do. Note that we are not proposing a specific algorithm or BT implementation but simply the use of BTs for modeling procedural level generators rather than NPCs, and demonstrating that it is feasible to do so via our examples. The implementations of the underlying condition and action nodes are agnostic to the PCGBT framework, e.g. all selection decisions in our examples are made at random. As mentioned before, these decisions could also be made based on designer preferences or player behavior at runtime. Decoupling the PCGBT framework from underlying implementations enables the approach to generalize to multiple design styles/preferences without enforcing any one. The primary utility of PCG-BTs would be to let designers combine sub-levels or handmade content into whole levels in a modular, explainable manner. Overall, we believe this initial exploratory foray into defining and using PCG-BTs holds promise for several interesting future directions.

Dynamic Level Generation Similar to traditional BTs, PCG-BTs could be reactive and produce dynamic generators where different level sections are generated by choosing different branches based on runtime conditions. Moreover, by having conditions tied to player behavior, we could generate different levels and gameplay experiences specifically tailored towards different player types and difficulty levels.

Combining with traditional BTs Modeling both design and gameplay agents via the same formalism opens up many generative possibilities. We could develop hybrid BTs with subtrees for both agent AI and level generation. Execution of a level generation branch could be controlled based on if a game playing branch succeeds or fails. Prior work (Cooper and Sarkar 2020) looked at using pathfinding agents for

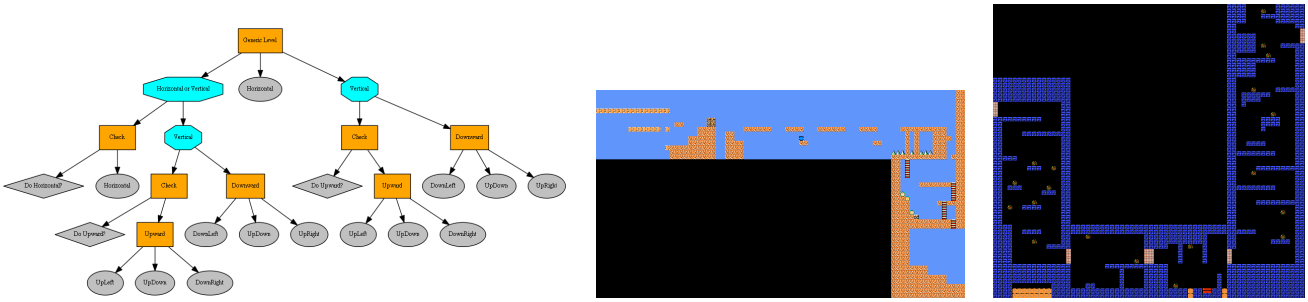

Figure 6: Generic PCG-BT and example generated MM and Metroid levels.

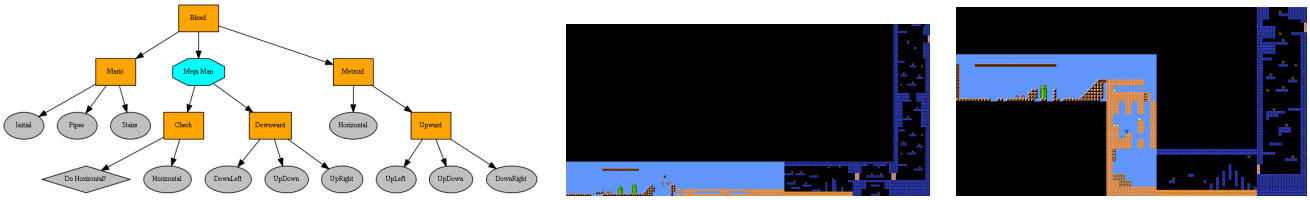

Figure 7: Blend PCG-BT and example generated levels blending Super Mario Bros., Mega Man and Metroid.

level repair. Such agents in BT form could be combined with PCG-BTs. There are also several directions using evolutionary algorithms. Could we meaningfully crossover and mutate such hybrid trees? Would we find commonalities between BT agents able to play certain PCG-BTs? Could we co-evolve game playing BTs and PCG-BTs in an open-ended manner to discover new types of agents and games?

RL and Evolution We used hand-crafted PCG-BTs but in the future they could be generated via learning and evolution. Prior work has used RL to learn BTs (Dey and Child 2013; Banerjee 2018) and could be leveraged for learning agents capable of designing games, akin to PCGRL. Evolution has also been applied for evolving BTs modeling desired behaviors (Jones et al. 2018; Neupane and Goodrich 2019), so we could evolve PCG-BTs modeling desired levels. Both RL and evolution could be used by designers for inferring BT structures from a set of exemplar levels.

General Game Design We showed that BTs can model game-agnostic level generators which can be instantiated to produce levels for different games and genres using the same tree. Thus PCG-BTs are capable of general level generation (Togelius and Yannakakis 2016), i.e. generating levels for a number of different games. This could in turn enable dynamic game generation analogous to dynamic level generation, producing different games for different players. Further, we could incorporate the blend BTs, and dynamically switch different games in and out during gameplay. In the future, it would be interesting to develop such generalized PCG-BTs. For e.g., could we cluster similar PCG-BTs into *forests* of BTs capable of generating similar types of games?

Designers and Practitioners PCG techniques, particularly those involving evolution, ML (Summerville et al. 2018) and RL have not been widely adopted in commercial games. Jacob, Devlin, and Hofmann (2020) point out that practitioners seek methods that afford authorial control, are readable and easy to interpret and do not require research expertise. Since BTs are a very well-known AI technique among designers, PCG-BTs could be a more accessible generative approach to those outside of research circles.

Conclusion

We introduced PCGBT, i.e., the use of behavior trees for procedural content generation, and demonstrated its application in several games and use-cases. While we did not perform playability evaluations on generated levels, we note that the building blocks of our levels are taken directly from the original games and thus segments themselves are playable. For dungeons, the layout algorithm guarantees that the dungeon is traversable from one room to another. Of course, if our action nodes explicitly generated the segments from scratch, playability evaluations of those would be necessitated. Since we focused on investigating the feasibility of this approach, we leave such considerations of overall playability of levels for future work. Moreover, in addition to the directions discussed previously, in the future, we want to develop a GUI/interactive application to enable designers to create and combine their own PCG-BTs as well as conduct a user study to examine PCGBT-generated levels in a dynamic, reactive context.

References

Banerjee, B. 2018. Autonomous acquisition of behavior trees for robot control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Champandard, A. 2007. Behavior trees for next-gen AI. In *Game Developers Conference Europe*.

Champandard, A. J.; and Dunstan, P. 2019. The behavior tree starter kit. In *Game AI Pro 360*, 27–46. CRC Press.

Colledanchise, M.; and Ögren, P. 2017. Behavior Trees in Robotics and AI: An Introduction. *arXiv preprint arXiv:1709.00084* .

Cooper, S.; and Sarkar, A. 2020. Pathfinding Agents for Platformer Level Repair. In *Proceedings of the Experimental AI in Games (EXAG) Workshop at AIIDE*.

Dahlskog, S.; and Togelius, J. 2012. Patterns and procedural content generation: revisiting Mario in world 1 level 1. In *First Workshop on Design Patterns in Games*.

Dey, R.; and Child, C. 2013. Ql-bt: Enhancing behaviour tree design and implementation with q-learning. In *IEEE Conference on Computational Inteligence in Games (CIG)*.

Dormans, J. 2010. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in games*, 1–8.

Glavin, F. G.; and Madden, M. G. 2014. Adaptive shooting for bots in first person shooter games using reinforcement learning. *IEEE Transactions on Computational Intelligence and AI in Games* 7(2): 180–192.

Gow, J.; and Corneli, J. 2015. Towards generating novel games using conceptual blending. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Hauck, E.; and Aranha, C. 2020. Automatic Generation of Super Mario Levels via Graph Grammars. In *2020 IEEE Conference on Games (CoG)*, 297–304. IEEE.

Iovino, M.; Scukins, E.; Styrud, J.; Ögren, P.; and Smith, C. 2020. A survey of behavior trees in robotics and ai. *arXiv preprint arXiv:2005.05842* .

Isla, D. 2005. Handling complexity in the Halo 2 AI. In *Game Developers Conference*. URL https://www.gamasutra.com/view/feature/130663/gdc_2005_.

Jacob, M.; Devlin, S.; and Hofmann, K. 2020. "It's Unwieldy and It Takes a Lot of Time"—Challenges and Opportunities for Creating Agents in Commercial Games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 88–94.

Jemmali, C.; Ithier, C.; Cooper, S.; El-Nasr, M. S.; et al. 2020. Grammar Based Modular Level Generator for a Programming Puzzle Game. In *Proceedings of the Experimental AI in Games (EXAG) Workshop at AIIDE*.

Jones, S.; Studley, M.; Hauert, S.; and Winfield, A. 2018. Evolving behaviour trees for swarm robotics. In *Distributed Autonomous Robotic Systems*, 487–501. Springer.

Karavolos, D.; Bouwer, A.; and Bidarra, R. 2015. Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation. In *FDG*.

Khalifa, A.; Bontrager, P.; Earle, S.; and Togelius, J. 2020. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Lim, C.-U.; Baumgarten, R.; and Colton, S. 2010. Evolving behaviour trees for the commercial game DEFCON. In *European conference on the applications of evolutionary computation*.

Linden, R.; Lopes, R.; and Bidarra, R. 2013. Designing procedurally generated levels. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Nam, S.; and Ikeda, K. 2019. Generation of Diverse Stages in Turn-Based Role-Playing Game using Reinforcement Learning. In *IEEE Conference on Games (CoG)*.

Neupane, A.; and Goodrich, M. A. 2019. Learning Swarm Behaviors using Grammatical Evolution and Behavior Trees. In *IJCAI*, 513–520.

Nicolau, M.; Perez-Liebana, D.; O'Neill, M.; and Brabazon, A. 2016. Evolutionary behavior tree approaches for navigating platform games. *IEEE Transactions on Computational Intelligence and AI in Games* 9(3): 227–238.

Perez, D.; Nicolau, M.; O'Neill, M.; and Brabazon, A. 2011. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *European Conference on the Applications of Evolutionary Computation*, 123–132. Springer.

Robertson, G.; and Watson, I. 2015. Building behavior trees from observations in real-time strategy games. In *International Symposium on Innovations in Intelligent SysTems and Applications (INISTA)*.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'neill, M. 2012. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 304–311. IEEE.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games* .

Summerville, A.; Snodgrass, S.; Mateas, M.; and Ontañón, S. 2016. The VGLC: The video game level corpus. In *PCG Workshop*.

Togelius, J.; and Yannakakis, G. N. 2016. General general game AI. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

Valls-Vargas, J.; Zhu, J.; and Ontañón, S. 2017. Graph grammar-based controllable generation of puzzles for a learning game about parallel programming. In *Foundations of Digital Games*.