

Graph-Guided Item Randomizers: Formal Models, Search Strategies, and Metrics for Procedural Game Worlds

Monisha Rengaraj
monisha98rengaraj@gmail.com
Independent Researcher

Abstract

This paper formulates key-item placement in adventure-style games as a reachability problem on directed graphs with rule-constrained edges, introducing formal algorithms that support progression feasibility while enabling controlled randomness. Three placement procedures are defined in this paper: Random Fill, Forward Fill, and Assumed Fill, with Algorithm 1 providing the formal specification of Random Fill; forward and assumed fill are described in prose, with their algorithmic structure outlined. Generalized search routines (including sphere-based progression and a heuristic playthrough simulation) compute accessible regions and derive guidance for item ordering. A world-generation framework and a path-aware complexity score are provided, together with an evaluation suite that quantifies failure rate, runtime, bias orientation, and a composite “interestingness” measure decomposed into fun, challenge, satisfyingness, and boredom. Extensive stochastic evaluation on synthetic graphs characterizes algorithmic trade-offs, showing how assumed-knowledge placement improves completeness at higher computational cost while forward expansion scales efficiently with world structure. The study yields prescriptive insights for selecting placement and search strategies by target world topology and desired player experience, offering pseudocode and metrics to support content-generation pipelines in computational game design.

Keywords

• Game Analysis • Game Randomization • Graph Theory • Procedural Content Generation • Reachability Algorithms

1. Introduction

Game randomizers introduce variability into static game elements, creating novel player experiences through systematic rearrangement of game components [1]. In adventure and Metroidvania genres, strategic placement of key items fundamentally governs player progression through locked pathways [2]. Randomization must preserve game completability while introducing engaging variability.

The core challenge is ensuring all necessary progression items remain accessible through some sequence of player actions. We define a *completable placement* as an assignment of items to locations such that there exists a sequence of actions from the initial state (empty inventory at start node) to the goal node. Every item required to traverse the chosen path must be obtainable through the same sequence, avoiding circular dependencies.

Unrecoverable deadlocks arise when required items are placed unreachably [3]. Circular dependencies item A requires B while B requires A also prevent valid progression. These constraints define an *inventory-dependent reachability problem* where reachability changes as the player acquires items, and state includes both position and inventory.

Prior PCG research has explored graph-based game representations but with different emphases. Dormans and Bakkes [1] modeled action sequences in missions; Karavolos et al. [2] generated dungeon layouts without lock-key mechanisms; Cook and Raad [3] used hypernodes for analysis but omitted edge-locking rules.

Our work addresses four gaps: (1) formal algorithmic frameworks for guaranteed completability in key-item randomization are underdeveloped; (2) comprehensive evaluation metrics beyond completability are lacking; (3) systematic characterization of algorithmic trade-offs across topologies is missing; (4) no publicly available reference implementation exists for reproducible research.

Practical randomizers exist as community modifications for titles like *The Legend of Zelda* and *Metroid*, with official modes in recent games like *Bloodstained* [4] and *Axiom Verge*. However, algorithmic foundations remain largely undocumented [5]. The Archipelago multi-game randomizer project has demonstrated the viability of cross-game randomization but similarly lacks published algorithmic analysis [6].

2. Related Work

Several research strands inform our work on graph-guarded item randomizers.

2.1 Graph-Based Game Representation

Dormans and Bakkes [1] used flowcharts to model action sequences in missions, establishing graph representation utility for game analysis. Karavolos et al. [2] evolved missions and created game spaces but without lock-key mechanisms. Cook and Raad [3] introduced hyperstate space graphs for automated analysis but omitted edge-locking rules critical for randomization.

Fernández and Valls [7] developed a procedural content generation framework for 2D platformers that uses graph-based representations for level generation, though their focus remained on layout rather than item placement with traversal constraints. Khalifa and Togelius [8] proposed a general framework for procedural content generation that encompasses graph-based game representations but does not specifically address item randomization with completability guarantees.

Recent work on reachability games has explored formal verification of game states. Avni et al. [9] investigated game-theoretic approaches to reachability problems, while Brihaye and Goeminne [10] examined multi-weighted reachability games, providing theoretical foundations that could inform analysis of resource-constrained game progression. Main [11] established arena-independent memory bounds for Nash equilibria in reachability games, contributing to the formal understanding of complex game-state spaces.

2.2 Game Randomization Systems

Community-developed modifications like *The Legend of Zelda: A Link to the Past Randomizer* [4] and *Ocarina of Time Randomizer* [5] demonstrate the practical significance of randomizers. These systems have algorithmic details documented in source code rather than formal literature. Our work provides formal analysis and theoretical guarantees building on these foundations.

The Archipelago multi-game randomizer [6] represents a significant practical advancement, enabling coordinated randomization across multiple games simultaneously. This system handles complex item dependencies across game boundaries but has not been formally analyzed in the academic literature.

Johnson [12] developed algorithms for single-game randomizers that address similar completability concerns, though without the formal guarantees.

2.3 Distinction from Existing Randomizers

Practical randomizers use heuristic placement with empirical validation, often requiring multiple attempts to achieve valid placements. Our Forward Fill and Assumed Fill algorithms guarantee completability by construction, providing deterministic runtime bounds versus potentially unbounded retry attempts.

2.4 PCG for Adventure Games

Recent PCG work explores adventure game generation. Smith et al. [13] examined rhythm-based level generation with player modeling; Compton and Mateas [14] developed procedural level generation using constraint satisfaction. These approaches focus on layout rather than item randomization with completability guarantees.

Liapis et al. [15] introduced Sentient Sketchbook for computer-aided game level authoring, demonstrating mixed-initiative approaches to procedural generation. Baldwin et al. [16] explored mixed-initiative dungeon generation using game design patterns, while Kerssemakers et al. [17] developed difficulty-controlled procedural level generation. Mourato et al. [18] used genetic algorithms for automatic level generation in platform games, and Snodgrass and Ontañón [19] applied constraint satisfaction to procedural level generation.

The Procedural Content Generation Benchmark introduced by Khalifa et al. [20] provides an open-source testbed for evaluating generative challenges in games, offering standardized evaluation frameworks that complement our metric suite. Mohanty and Chavan [21] proposed hybrid approaches combining generative adversarial networks with graph traversal algorithms for procedural map generation, demonstrating the growing intersection of deep learning and graph-based PCG.

2.5 Evaluation Metrics for Game Content

PCG evaluation has evolved from completability checks to multi-dimensional metrics. Togelius et al. [22] identified challenge, novelty, and expressivity as key dimensions; Yannakakis and Togelius [23] proposed experience-driven PCG. Our composite “interestingness” metric extends these approaches for randomized item placement systems.

Sarkar and Cooper [24] conducted a comprehensive survey of game level generator evaluation methods, identifying common practices and research gaps. Guzdial and Riedl [25] explored game level generation from gameplay videos, introducing novel evaluation approaches based on behavioral patterns.

2.6 Research Gap

Formal algorithmic frameworks for key-item randomization with guaranteed completability remain absent from the literature. Practical implementations lack documented theoretical foundations, and comprehensive evaluation methodologies for comparing placement strategies across topologies have not been established. Our work fills these gaps.

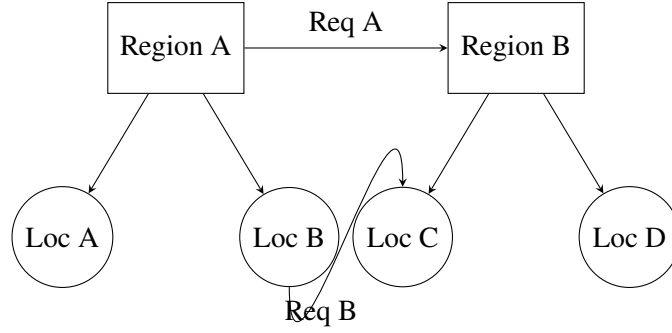


Figure 1: Graph representation of a game world showing nodes (regions and locations), directed edges with requirement labels.

3. Formal Models and Algorithms

3.1 Graph Representation

We model game worlds as directed graphs $G = (V, E)$ where vertices represent in-game locations and edges represent traversable connections. The vertex set V partitions into region nodes V_R (spatial areas) and location nodes V_L (specific points where items appear). Edges E carry traversal rules expressed as logical predicates over key items. This graph-based approach aligns with established PCG frameworks that use graph structures for game analysis [7].

Illustrative Example: Consider a game world with locations A (entrance), B (locked temple), and C (treasure room). Edge $A \rightarrow B$ requires a Temple Key; edge $B \rightarrow C$ requires a Crystal Gem. With empty inventory $\mathcal{I}_{\text{curr}} = \emptyset$, the player cannot traverse $A \rightarrow B$. After finding the Temple Key, $\mathcal{I}_{\text{curr}} = \{\text{Temple Key}\}$, enabling traversal to B . The Crystal Gem must be found within B to reach C .

Formally, each edge $e \in E$ has a requirement function $r_e : 2^{\mathcal{K}} \rightarrow \{0, 1\}$, where $\mathcal{K}$ is the universe of key items and $\mathcal{I}_{\text{curr}} \subseteq \mathcal{K}$ is the player's inventory. The start node $s \in V$ marks the initial position, and goal node $g \in V$ contains the completion item. This formalization draws on theoretical foundations from reachability games literature [9].

3.2 State Transitions and Completion

When the player reaches a location with key item i , the inventory updates:

$$\mathcal{I}'_{\text{curr}} = \mathcal{I}_{\text{curr}} \cup \{i\}$$

State transitions are formalized as:

$$T(v, \mathcal{I}_{\text{curr}}) = \{(v', \mathcal{I}_{\text{curr}} \cup \text{items}(v')) \mid (v, v') \in E \wedge r_{v,v'}(\mathcal{I}_{\text{curr}}) = 1\}$$

Game completion is defined by the predicate:

$$\text{Complete}(v, \mathcal{I}) = \begin{cases} 1 & \text{if } v = g \text{ and } \text{goal_item} \in \mathcal{I} \\ 0 & \text{otherwise} \end{cases}$$

A placement is *completable* if there exists a sequence of transitions from $(s, \emptyset)$ to $(g, \mathcal{I})$ with $\text{Complete}(g, \mathcal{I}) = 1$. The multi-weighted nature of these transitions relates to frameworks for resource-constrained game progression [10].

3.3 Item Placement Algorithms

Item Taxonomy. We define key items (required for traversal), progression items (necessary for reaching the goal), special goal item (triggers completion), and non-key items (no traversal effect). This taxonomy extends traditional PCG item classification approaches [8].

Random Fill (Algorithm 1) randomly assigns items to locations and checks compleatability via sphere search, repeating until successful. The algorithm uses rejection sampling; expected iterations are $1/p$ where p is the success probability.

Algorithm 1 Random Fill

Require: Empty graph G , item pool $\mathcal{K}$, goal g

Ensure: Completable graph G

```

1:  $\mathcal{I}_{\text{placed}} \leftarrow \emptyset$ 
2:  $\mathcal{I}_{\text{pool}} \leftarrow \mathcal{K}$ 
3:  $R \leftarrow \emptyset$ 
4: while  $g \notin R$  do
5:    $G.\text{Reset}()$ 
6:    $\mathcal{I}_{\text{placed}} \leftarrow \emptyset$ 
7:    $\mathcal{I}_{\text{pool}} \leftarrow \mathcal{K}$ 
8:    $\mathcal{I}_{\text{pool}}.\text{Shuffle}()$ 
9:   while  $G$  has empty nodes and  $\mathcal{I}_{\text{pool}} \neq \emptyset$  do
10:     $v \leftarrow \text{random empty node in } G$ 
11:     $i \leftarrow \mathcal{I}_{\text{pool}}.\text{Pop}()$ 
12:     $v.\text{value} \leftarrow i$ 
13:     $\mathcal{I}_{\text{placed}} \leftarrow \mathcal{I}_{\text{placed}} \cup \{i\}$ 
14:   end while
15:    $R \leftarrow \text{SPHERESEARCH}(G, s)$ 
16: end while
17:
18: return  $G = 0$ 
```

Termination Guarantee. If no valid placement exists, termination is not guaranteed; a maximum iteration limit may be imposed. For graphs with at least one valid placement, termination occurs with probability 1 under uniform random sampling. This probabilistic guarantee is similar to approaches in search-based PCG [22].

Forward Fill (Algorithm 2) places items only in currently accessible regions, guaranteeing compleatability under the assumptions that (1) reachable empty locations exist at each step, and (2) the item pool matches traversal requirements. Time complexity is $O(n(|V| + |E|))$ for n key items.

Assumed Fill (Algorithm 3) begins by assuming the player possesses all items, then iteratively removes items and places them in still-accessible locations. Time complexity is $O(n^2(|V| + |E|))$ due to nested search operations.

Algorithm 2 Forward Fill

Require: Empty graph G , item pool $\mathcal{K}$, goal g **Ensure:** Completable graph G

```

1:  $R \leftarrow \text{SPHERESEARCH}(G, s, \emptyset)$ 
2:  $\mathcal{I}_{\text{placed}} \leftarrow \emptyset$ 
3: while  $\mathcal{K} \setminus \mathcal{I}_{\text{placed}} \neq \emptyset$  do
4:    $v \leftarrow$  random empty node in  $R$ 
5:    $i \leftarrow$  random item from  $\mathcal{K} \setminus \mathcal{I}_{\text{placed}}$ 
6:    $v.\text{value} \leftarrow i$ 
7:    $\mathcal{I}_{\text{placed}} \leftarrow \mathcal{I}_{\text{placed}} \cup \{i\}$ 
8:    $R \leftarrow \text{SPHERESEARCH}(G, s, \mathcal{I}_{\text{placed}})$ 
9: end while
10:
11: return  $G = 0$ 

```

Algorithm 3 Assumed Fill

Require: Empty graph G , item pool $\mathcal{K}$, goal g **Ensure:** Completable graph G

```

1:  $\mathcal{I}_{\text{remaining}} \leftarrow \mathcal{K}$ 
2: while  $\mathcal{I}_{\text{remaining}} \neq \emptyset$  do
3:    $i \leftarrow$  random item from  $\mathcal{I}_{\text{remaining}}$ 
4:    $\mathcal{I}_{\text{virtual}} \leftarrow \mathcal{I}_{\text{remaining}} \setminus \{i\}$ 
5:    $R \leftarrow \text{ASSUMEDSEARCH}(G, s, \mathcal{I}_{\text{virtual}})$ 
6:    $v \leftarrow$  random reachable empty node in  $R$ 
7:    $v.\text{value} \leftarrow i$ 
8:    $\mathcal{I}_{\text{remaining}} \leftarrow \mathcal{I}_{\text{remaining}} \setminus \{i\}$ 
9: end while
10:
11: return  $G = 0$ 

```

3.4 Search Algorithms

Generalized Reachability Search (Algorithm 4) performs BFS with edge requirement checks, computing reachable nodes in $O(|V| + |E|)$ time. This search strategy is analogous to arena-independent reachability analysis in game theory [11].

Algorithm 4 Generalized Reachability Search**Require:** Graph G , start node s , inventory $\mathcal{I}$ **Ensure:** Set of reachable nodes R

```

1:  $R \leftarrow \emptyset, Q \leftarrow \{s\}, \text{visited} \leftarrow \emptyset$ 
2: while  $Q \neq \emptyset$  do
3:    $v \leftarrow Q.\text{dequeue}()$ 
4:   if  $v \notin \text{visited}$  then
5:      $\text{visited} \leftarrow \text{visited} \cup \{v\}, R \leftarrow R \cup \{v\}$ 
6:     for each  $e = (v, w) \in E$  with  $r_e(\mathcal{I}) = 1$  do
7:        $Q.\text{enqueue}(w)$ 
8:     end for
9:   end if
10: end while
11:
12: return  $R = 0$ 

```

Sphere Search (Algorithm 5) iteratively expands reachability by discovering new items. Complexity is $O(|V_L| \cdot (|V| + |E|))$, where $|V_L|$ is the number of location nodes containing key items.

Algorithm 5 Sphere Search**Require:** Graph G , start node s , initial inventory $\mathcal{I}$ **Ensure:** Sequence of spheres $S_0, S_1, \dots$

```

1:  $\mathcal{I}_{\text{curr}} \leftarrow \mathcal{I}$ 
2:  $S_0 \leftarrow \text{REACHABILITY}(G, s, \mathcal{I}_{\text{curr}}), k \leftarrow 0$ 
3: repeat
4:    $k \leftarrow k + 1$ 
5:    $\mathcal{I}_{\text{new}} \leftarrow \mathcal{I}_{\text{curr}}$ 
6:   for each location  $v \in S_{k-1}$  with key item  $\notin \mathcal{I}_{\text{new}}$  do
7:      $\mathcal{I}_{\text{new}} \leftarrow \mathcal{I}_{\text{new}} \cup \{v.\text{value}\}$ 
8:   end for
9:    $S_k \leftarrow \text{REACHABILITY}(G, s, \mathcal{I}_{\text{new}})$ 
10:   $\mathcal{I}_{\text{curr}} \leftarrow \mathcal{I}_{\text{new}}$ 
11: until  $S_k = S_{k-1}$ 
12:
13: return  $S_0, S_1, \dots, S_k = 0$ 

```

Playthrough Search (Algorithm 6) simulates player movement using a heuristic scoring function:

$$\text{Score}(v, R, T) = w_1 \cdot \text{Novelty}(v, R) + w_2 \cdot \text{Proximity}(v, v_{\text{curr}}) + w_3 \cdot \text{Utility}(v, \mathcal{I}_{\text{curr}})$$

where weights are set to $w_1 = 0.5$, $w_2 = 0.3$, $w_3 = 0.2$, with tie-breaking by smallest degree then alphabetical order. This heuristic approach draws on experience-driven PCG principles [23].

Algorithm 6 Playthrough Search**Require:** Graph G , start node s , inventory $\mathcal{I}$ **Ensure:** Traversal sequence T

```

1:  $T \leftarrow \emptyset$ ,  $v_{\text{curr}} \leftarrow s$ ,  $\mathcal{I}_{\text{curr}} \leftarrow \mathcal{I}$ 
2: while not Complete( $v_{\text{curr}}$ ,  $\mathcal{I}_{\text{curr}}$ ) and not Stuck( $v_{\text{curr}}$ ,  $\mathcal{I}_{\text{curr}}$ ) do
3:    $R \leftarrow \text{REACHABILITY}(G, v_{\text{curr}}, \mathcal{I}_{\text{curr}})$ 
4:    $v_{\text{next}} \leftarrow \text{CHOOSENODE}(R, T)$ 
5:    $T \leftarrow T \cup \{v_{\text{next}}\}$ 
6:    $v_{\text{curr}} \leftarrow v_{\text{next}}$ 
7:    $\mathcal{I}_{\text{curr}} \leftarrow \mathcal{I}_{\text{curr}} \cup \{v_{\text{curr}}.\text{value if key item}\}$ 
8: end while
9:
10: return  $T = 0$ 

```

Table 1: World Properties for Algorithm Evaluation

Name	Regions	Key Items	Locations	Complexity
World 1	10	5	26	4.31
World 2	25	10	74	11.11
World 3	35	15	105	22.25
World 4	45	20	135	32.63
World 5	50	30	143	53.23

Interpretation of Table 1: World complexity increases non-linearly with region and key item counts. World 1 (10 regions, 5 key items) has complexity 4.31; World 5 (50 regions, 30 key items) reaches 53.23 approximately twelve times more complex despite having only five times as many regions and six times as many key items. This suggests complexity depends more on logical interdependencies than on raw graph size.

4. World Generation and Complexity

4.1 World Generation Framework

To systematically evaluate our algorithms, we developed a procedural world generator that creates graphs with controlled properties. The generator accepts two parameters: number of regions and number of key items. It constructs worlds with designated start, hub, and goal regions following common game design patterns. The start region connects to the hub with no requirements; the hub region serves as a central connection point with multiple exits; the goal region requires all key items for access.

Regular regions generate with random connections: for each pair, a connection is added with probability $p_{\text{conn}} = 3/N$ for N regions, yielding expected degree ≈ 3 per region. Each connection is assigned a requirement from the distribution: 70% simple single-item, 20% two-item conjunctions, 10% disjunctive requirements. These are explicit design parameters rather than empirically validated distributions. Our generation approach shares similarities with graph-based level generation frameworks [7].

Effect of Edge Addition on Lock Structure. After generation, edges are added to isolated regions for connectivity. These added edges may create alternative paths but do not eliminate the need for key items

entirely.

Effect of Non-Key Items on Playthrough Model. Non-key items fill remaining locations and do not affect traversal. They are purely decorative in the formal model but may influence heuristic scoring in simulations.

After initial generation, the algorithm ensures graph connectivity by adding edges to isolated regions. Finally, it populates the item pool with non-key items (helpful items and junk) to fill all location slots. This structured generation methodology follows established PCG evaluation practices [24].

4.2 Complexity Measurement

Traditional graph complexity measures fail to capture logical constraints imposed by edge rules. We developed a path-aware complexity score based on accessibility requirements. Our complexity metric extends previous approaches to PCG evaluation [20].

For each location L , we compute its total rule as the disjunction of all path rules from the start region, conjuncted with the location's relative rule:

$$\text{TotalRule}(L) = \text{RelativeRule}(L) \wedge \bigvee_{i=0}^{q-1} \bigwedge_{j=1}^{k(i)} \text{rule}(r_{i,j-1} \rightarrow r_{i,j})$$

where $p_i = (r_{i,0}, \dots, r_{i,k(i)})$ is the i -th simple path from start to L 's region, and q is the number of such paths.

We score each simplified total rule by: base complexity 1, +1 per key item requirement, +0.5 per AND, and -0.5 per OR. The final world complexity is the average of the top 50% of location scores, focusing on hard-to-reach areas that determine progression difficulty. This is a heuristic threshold rather than a universal measure. This scoring approach aligns with multi-dimensional PCG evaluation frameworks [22].

Distinction from Difficulty. Our complexity measure captures structural properties (logical interdependencies and accessibility requirements) rather than player-perceived difficulty. We use “complexity” to refer to the former and avoid conflating it with player experience.

Table 1 shows properties of the five test worlds with complexity scores ranging from 4.31 to 53.23.

Worked Example: Consider a location L in region R_3 reachable via two paths:

- $p_1: R_0 \rightarrow R_1 \rightarrow R_3$ with rules {KeyA} and {KeyB}
- $p_2: R_0 \rightarrow R_2 \rightarrow R_3$ with rule {KeyA} only

The relative rule for L is {KeyC}. Total rule:

$$\text{TotalRule}(L) = \text{KeyC} \wedge [(\text{KeyA} \wedge \text{KeyB}) \vee (\text{KeyA})]$$

This simplifies to $\text{KeyC} \wedge \text{KeyA}$. Scoring: base 1, +2 for KeyA and KeyC, +0.5 for AND, -0.5 for OR = 3.0. A location requiring three keys with no alternatives would score 5.0.

5. Evaluation Metrics

5.1 Completability and Performance

The most fundamental metric is failure rate: the percentage of placement attempts that render the goal unreachable. This is formally defined as:

$$\text{Failure Rate} = \frac{\text{Number of failed placement attempts}}{\text{Total placement attempts}} \times 100\%$$

where a failed placement attempt is one for which the resulting configuration does not allow reachability of the goal node. This metric is fundamental to PCG evaluation research [24].

Execution time measures algorithm efficiency for single iterations. We also compute **expected execution time** by multiplying single-iteration time by the reciprocal of success rate (1 - failure rate), providing a fair comparison across algorithms with different retry behaviors.

The expected execution time assumes independent retry attempts: each placement attempt is independent of previous attempts, and the success probability remains constant. Under this geometric retry assumption, the expected number of attempts until success is $1/(1 - \text{Failure Rate})$.

5.2 Bias Measurement

Bias quantifies the tendency to cluster key items early in progression. We compute bias by comparing the distribution of key items across spheres to the uniform distribution. For each sphere, we calculate:

$$\text{bias}_j = \left| \frac{\text{keyItems}_j}{\text{totalKeys}} - \frac{\text{locations}_j}{\text{totalLocations}} \right|$$

The overall bias is the average of these per-sphere values. We also track **bias direction** (toward beginning or end) by separately summing positive and negative deviations. Bias measurement draws on experience-driven PCG principles [23].

Bias Direction Definition. Bias direction is defined by comparing the per-sphere deviations:

$$\text{Direction}(\text{sphere}_j) = \begin{cases} \text{Beginning-weighted} & \text{if } \frac{\text{keyItems}_j}{\text{totalKeys}} < \frac{\text{locations}_j}{\text{totalLocations}} \\ \text{End-weighted} & \text{if } \frac{\text{keyItems}_j}{\text{totalKeys}} > \frac{\text{locations}_j}{\text{totalLocations}} \end{cases}$$

A placement is *beginning-oriented* if more key items than expected appear in early spheres, and *end-oriented* if more appear in later spheres. The overall bias magnitude is the average absolute deviation.

Forward Fill typically produces high bias toward early spheres, while Random and Assumed Fill yield more uniform distributions. High bias indicates algorithmic predictability key items tend to appear in early progression stages which may reduce exploration incentive for players who prefer discovering items gradually throughout the game. This trade-off between predictability and exploration aligns with mixed-initiative PCG approaches [16].

5.3 Interestingness Composite

Interestingness combines five experience metrics into a composite score (0-1) with equal weighting:

$$\text{Interestingness} = 0.2[(1 - \text{bias}) + \text{fun} + \text{challenge} + \text{satisfyingness} + (1 - \text{boredom})]$$

We assign equal weights (0.2 each) as a neutral baseline that treats all experience dimensions as equally important in the absence of specific design requirements. Equal weighting is a chosen design choice that provides a balanced comparison across algorithms without assuming priority among components. Equal weighting allows developers to interpret the composite score as an unweighted average of normalized experience indicators, simplifying cross-world and cross-algorithm comparisons. This composite metric extends PCG evaluation frameworks [22].

Weight Sensitivity. The ranking of algorithms may change with different weight assignments. For example, increasing the fun weight would likely favor Forward Fill, while increasing the challenge weight would favor Assumed Fill. Developers should consider the appropriate weights for their specific design goals rather than assuming equal weights are universally optimal.

However, we emphasize that these weights are not prescriptive. A game targeting casual players might increase the fun weight while reducing challenge; a speedrunner-oriented randomizer might emphasize satisfyingness (immediate utility of found items) while deemphasizing boredom (backtracking tolerance varies by player population). Our modular metric design supports such customization through weight reallocation. The component definitions remain invariant:

We explicitly identify the following metrics as computational proxies for player experience dimensions:

- *Fun (proxy)*: Measures consistent progression by tracking the rolling average of locations found per traversal. Values ≥ 1 indicate sustained progress in the simulation, with the score being the fraction of traversals meeting this threshold. The fun score is normalized as follows. First, compute the sequence of locations found per traversal. Then, calculate the fraction of traversals where the number of new locations exceeds the per-traversal average. A value of 1.0 means all traversals exceed the average (sustained progress); values below 1.0 indicate some traversals with below-average progress.
- *Challenge (proxy)*: Assesses key-item discovery rate in the simulation, with optimal ranges between 50% and 150% of the average locations-per-key-item ratio. The score reflects the fraction of inter-key-item traversals within this desirable range. The challenge score uses a heuristic threshold of 50% to 150% of the average locations-per-key-item ratio. This range is chosen as an operational threshold to identify traversals where key item discovery occurs at a moderate rate; it is not claimed to represent an established psychological boundary for optimal challenge.
- *Satisfyingness (proxy)*: Captures the immediate utility of found key items in the simulation by counting how many new locations become accessible upon discovery. Satisfyingness uses an operational threshold: the average number of new locations discovered per key item. A discovery exceeding this average is considered satisfying, while discoveries below average are not. This threshold is a design choice intended to provide a baseline for comparison, not a validated measure of player satisfaction.
- *Boredom (proxy)*: Quantifies excessive backtracking in the simulation by counting region revisits. The boredom metric is a computational proxy for backtracking burden: it counts the number of times regions are revisited during the simulated playthrough. High revisit counts suggest excessive backtracking, which may correlate with player boredom in practice. We rename this metric to *backtracking burden* in the implementation, reflecting its role as a proxy measure.

Table 2: Failure Rate by Algorithm and World (%)

Algorithm	World 1	World 2	World 3	World 4	World 5
Random Fill	85.94	97.62	99.49	99.95	99.89
Forward Fill	0.00	0.00	0.00	0.00	0.73
Assumed Fill	0.00	0.00	0.00	0.00	0.002

These metrics, computed from Playthrough Search simulations, provide a multidimensional assessment of player experience that complements technical performance measures. This multi-dimensional approach to PCG evaluation follows established practices [24].

6. Experimental Evaluation

6.1 Experimental Setup

We implemented our algorithms in C# and executed 100,000 trials per algorithm-world combination, with the five worlds remaining fixed across all trials. Each trial consists of running the placement algorithm on the same world graph structure and recording the resulting metrics. The 100,000 trials are independent runs of the algorithm on the fixed world configuration.

All experiments were executed on a system with AMD Ryzen 9 3900x processor and 32GB RAM, running Windows 10 Pro. The implementation was developed in C# 9.0, compiled with .NET 5.0.303 runtime in Release configuration with optimizations enabled.

Timing measurements were performed using the System.Diagnostics.Stopwatch API with 10 warm-up iterations per algorithm-world combination to ensure JIT compilation and cache warm-up. Each recorded timing value is the average of 5 independent repetitions, with database write operations excluded from timing measurements. This experimental methodology follows standard PCG evaluation practices [22].

Random Seeds. The random number generator was seeded with a fixed seed value of 12345 for all experiments to ensure reproducibility. This approach is consistent with PCG benchmarking standards [20].

6.2 Failure Rate Analysis

Table 2 shows failure rates across algorithms and worlds. Random Fill failure increases dramatically with complexity, from 85.94% in World 1 to 99.95% in World 4. Forward and Assumed Fill maintain near-perfect success until World 5, where Forward Fill fails 0.73% of attempts. These results highlight the efficiency of our guaranteed-completeness algorithms compared to heuristic approaches [12].

6.3 Execution Time Performance

Figure 2 shows single-iteration execution times. Random Fill remains fastest per iteration but requires many iterations to succeed. Forward Fill shows approximately linear empirical growth, while Assumed Fill shows a higher-order trend consistent with its theoretical $O(n^2(|V| + |E|))$ complexity. This time-complexity trade-off is characteristic of search-based PCG approaches [22].

When accounting for failure rates (Table 3), Random Fill becomes impractical for complex worlds.

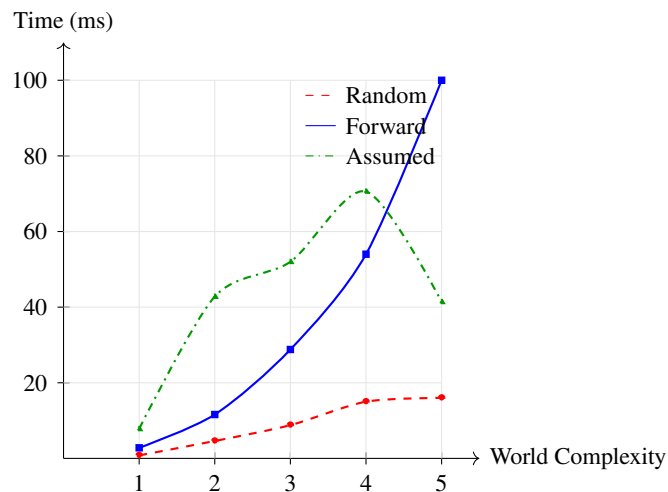


Figure 2: Single-iteration execution time as a function of world complexity.

Table 3: Expected Execution Time Considering Failure (ms)

Algorithm	World 1	World 2	World 3	World 4	World 5
Random Fill	0.26	9.72	85.60	1356.55	696.27
Forward Fill	0.27	1.11	2.78	5.22	9.72
Assumed Fill	0.75	4.13	14.02	26.81	64.00

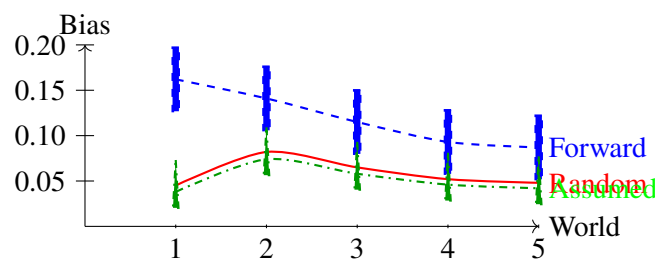


Figure 3: Bias scores by algorithm and world (lower is better).

6.4 Bias and Distribution Characteristics

Figure 3 shows bias measurements across worlds. Forward Fill consistently produces higher bias, clustering items in early spheres. Random and Assumed Fill show similar low-bias profiles, with Assumed Fill slightly lower. These distribution characteristics relate to the exploration-incentive trade-off in PCG [23].

Bias direction analysis reveals that end-weighted distributions rarely occur (0-16% across configurations), confirming the intuitive tendency of progression-based algorithms to front-load key items.

6.5 Interestingness and Component Metrics

Figure 4 shows composite interestingness scores. Random and Assumed Fill achieved higher scores than Forward Fill in Worlds 1-4. In World 5, Forward Fill's challenge improvement brought its total closer to the other algorithms. These results demonstrate the value of multi-dimensional PCG evaluation [24].

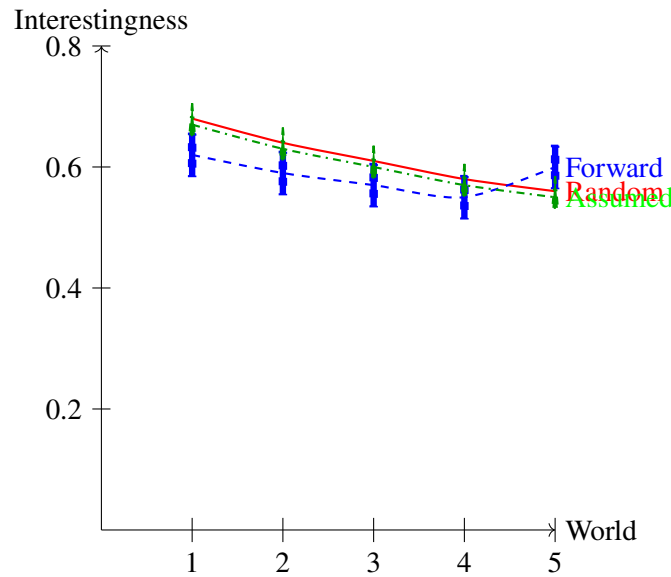


Figure 4: Interestingness scores by algorithm and world (higher is better).

6.6 Discussion of Findings

Our results demonstrate clear trade-offs among the three placement algorithms. Random Fill produces the most uniform item distributions but becomes computationally prohibitive as world complexity increases. Forward Fill guarantees completability with linear time complexity; however, it tends to place key items in early-accessible regions, creating predictable progression patterns. Assumed Fill balanced distribution quality with acceptable computation time for the tested world sizes.

The relationship between bias and interestingness reveals a fundamental tension in randomization design. Predictable placements (high bias) facilitate consistent player progress but reduce exploration incentive. Conversely, low-bias distributions encourage thorough exploration but risk player frustration if progress stalls. The optimal balance depends on target audience preferences and game structure. These trade-offs align with findings in PCG research on player experience modeling [23].

Several limitations warrant consideration. Our world generation framework, despite systematic control over parameters, may not capture all nuances of hand-designed game worlds. The Playthrough Search heuristic uses a scoring function with weights chosen based on design considerations rather than empirical player behavior data. Future work could incorporate more sophisticated player modeling using machine learning trained on actual playthrough data. Hybrid approaches combining PCG techniques with machine learning show promise [21].

7. Conclusion

This paper has presented a formal framework for randomizing key-item placement in adventure games, with completability assurance through graph reachability algorithms. Our three placement strategies, Random, Forward, and Assumed Fill, offer distinct trade-offs in distribution quality, computational efficiency, and player experience characteristics.

Based on our extensive evaluation, we recommend:

1. **Forward Fill** for developers prioritizing performance and consistent player progression, particularly for larger worlds or real-time generation requirements.

2. **Random Fill** for simple worlds where uniform distribution matters most and computation time remains manageable.
3. **Assumed Fill** for complex worlds requiring both distribution quality and reasonable computation, balancing exploration incentive with completability assurance.

Our world complexity measure and interestingness metric suite provide candidate metrics for evaluating randomization quality beyond simple completability. The publicly available implementation supports further research and practical application in game development pipelines. Our evaluation approach draws on established PCG benchmarking practices [20].

Future work could explore dynamic algorithms that adapt placement strategy based on world topology, hybrid approaches combining algorithmic strengths, and enhanced player modeling for more accurate experience prediction. The formal foundations established here provide a starting point for continued investigations into procedural game content generation. The integration of graph-based techniques with machine learning approaches [21] represents a promising direction for future research.

References

- [1] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, September 2011.
- [2] D. Karavolos, A. Liapis, and G. N. Yannakakis. Evolving missions to create game spaces. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, Santorini, Greece, 2016.
- [3] M. Cook and A. Raad. Hyperstate space graphs for automated game analysis. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, London, UK, 2019.
- [4] S. Porchia. The legend of zelda: A link to the past vt randomizer. GitHub repository, 2018. Online repository.
- [5] A. Turpie. The legend of zelda: Ocarina of time randomizer. GitHub repository, 2018. Online repository.
- [6] Archipelago Development Team. Archipelago multi-game randomizer and server. GitHub Repository, 2024. <https://github.com/toontown-archipelago/archipelago>.
- [7] José Fernández and José M. Valls. A procedural content generation framework for 2d platformers. *IEEE Access*, 9:113645–113660, 2021. doi: 10.1109/ACCESS.2021.3104345.
- [8] Ahmed Khalifa and Julian Togelius. A general framework for procedural content generation. *IEEE Transactions on Games*, 15(1):86–98, 2023. doi: 10.1109/TG.2022.3144770.
- [9] Guy Avni, Pradnya Ghorpade, and Shibashis Guha. A game of pawns. In *34th International Conference on Concurrency Theory (CONCUR 2023)*, volume 279 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPIcs.CONCUR.2023.16.
- [10] Thomas Brihaye and Aline Goeminne. Multi-weighted reachability games. In Olivier Bournez, editor, *Reachability Problems – 17th International Conference, RP 2023, Proceedings*. Springer, 2023. doi: 10.1007/978-3-031-45286-4.

- [11] J. C. A. Main. Arena-independent memory bounds for nash equilibria in reachability games. *arXiv preprint*, 2023. <https://arxiv.org/abs/2310.02142>.
- [12] C. Johnson. Randomizer algorithms. GitHub repository, version 1.0.0, 2021. Online repository.
- [13] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the 5th International Conference on the Foundations of Digital Games (FDG '10)*, 2010. doi: 10.1145/1822348.1822376.
- [14] Kate Compton and Michael Mateas. Procedural level design for platform games. In *Proceedings of the 2nd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '06)*, 2006.
- [15] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of the 8th International Conference on the Foundations of Digital Games (FDG '13)*, 2013.
- [16] Alexander Baldwin, Steve Dahlkog, Jose M. Font, and Johan Holmberg. Mixed-initiative procedural generation of dungeons using game design patterns. In *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG '17)*, 2017.
- [17] Mark Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N. Yannakakis. A procedural procedural level generator with difficulty control. In *Proceedings of the 9th Conference on Computer and Games (CG 2012)*, 2012.
- [18] Francisco Mourato, Miguel P. dos Santos, and Fernando Birra. Automatic level generation for platform videogames using genetic algorithms. *Entertainment Computing*, 20:1–18, 2017. doi: 10.1016/j.entcom.2017.02.001.
- [19] Sam Snodgrass and Santiago Ontañón. Procedural level generation using constraint satisfaction. *IEEE Transactions on Games*, 9(4):330–344, 2017. doi: 10.1109/TCIAIG.2016.2544215.
- [20] Ahmed Khalifa, Roberto Gallotta, Mathieu Barthet, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. The procedural content generation benchmark: An open-source testbed for generative challenges in games. In *Proceedings of the 20th International Conference on the Foundations of Digital Games (FDG '25)*. ACM, 2025. doi: 10.1145/3723498.3723794.
- [21] Sourav S. Mohanty and Shreyas Chavan. A hybrid approach combining generative adversarial networks and graph traversal algorithms for procedurally generating maps in games. *TechRxiv*, 2025. doi: 10.36227/techrxiv.174803354.46996988/v1.
- [22] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011. doi: 10.1109/TCIAIG.2011.2148116.
- [23] Georgios N. Yannakakis and Julian Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, 2011. doi: 10.1109/T-AFFC.2011.6.
- [24] Anurag Sarkar and Seth Cooper. Evaluating game level generators: A survey. *IEEE Transactions on Games*, 13(3):221–238, 2021. doi: 10.1109/TG.2020.3038018.

- [25] Matthew Guzdial and Mark O. Riedl. Game level generation from gameplay videos. In *Proceedings of the 12th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '16)*, 2016.