# An Agent-Based Approach for Procedural Puzzle Generation in Graph-Based Maps

Francesco Venco
Politecnico di Milano - DEIB, Milano, Italy
francesco.venco@polimi.it

Pier Luca Lanzi
Politecnico di Milano - DEIB, Milano, Italy
pierluca.lanzi@polimi.it
ORCID: 0000-0002-1933-7717

***Abstract*—We present an algorithm to add puzzles to maps represented as graphs. The algorithm starts from an empty map, represented as a graph, with at least one entry area and one exit area. It runs several specialized agents responsible for adding puzzles (e.g., locked doors, keys, switches). It generates a map with at least one acceptable solution (path) whose difficulty depends on the type of agents used (that is, the variety of puzzles added) and the number of puzzles added by each agent. Most importantly, no sequence of actions can leave the players stuck in a dead-end situation with no way to reach the goal. We include two examples of agents specialized in (i) switch mechanics (e.g., a lever that opens a passage and closes another one, the lighting of a fire that shows an inscription needed to solve another puzzle), and (ii) element collection mechanics (e.g., collecting keys or other puzzle elements to open a passage).**

## I. Introduction

Puzzles are vital components of many game genres. They are an effective way to enrich gameplay by asking players to stop and think about solving interesting problems. Sometimes, puzzles are visible and explicitly presented to players, like in The Witness,[1] or so much entangled into the gameplay to become almost invisible, like in Tomb Raider[2] or Uncharted[3] series. Puzzles are challenging to design both in terms of mechanics and level design. In fact, once players understand how to solve a puzzle, this most likely ceases to be fun [1]. Procedural content generation has been applied to create almost any type of game content, such as music, levels, narrative, rules [2], even entire universes as in *No Man's Sky*.[4] Its application to puzzles however, remains limited, mainly focused on games based on puzzle mechanics, and rarely on creating puzzles as accessories to other genres [3].

Mazes are puzzles that require players to reach an exit in environments that can be hazardous or just very complex to navigate. This paper presents an agent-based algorithm to populate maps, with at least one entry and one exit point, with different puzzle types. Our algorithm is inspired by agent-based procedural content generation approaches [2] that are often applied to generate terrains using agents specialized in different content types (e.g., forests, rivers, roads). Our algorithm starts from an empty map, represented as a connected graph, and runs a set of specialized agents, each one responsible for adding a specific type of puzzle to the map. The algorithm guarantees that there is one feasible solution: one sequence of actions to take the players to the exit. Most importantly, it guarantees that no sequence of players' actions will leave players unable to solve the maze. We show examples using agents specialized in (i) switch mechanics (e.g., a lever that opens/closes a passage) and (ii) element collection mechanics (e.g., collecting keys or other puzzle elements to open a passage). We present the results of an experimental evaluation showing that our algorithm can create quite complex mazes in a small amount of time.

The paper is organized as follows. In Section II, we preview a brief review of the relevant literature. In Section III, we give an overview of our algorithm using simple examples while, in Section IV, we give a detailed description including pseudo-code of the main procedures. In Section V, we present the results of an experimental evaluation we performed focused on the time required to generate puzzles and the complexity of the puzzles generated. Finally, in Section VI, we delineate the future research directions.

## II. Related Work

Procedural content generation (PCG) has been widely applied since the late 1970s for creating several types of game assets (worlds, levels, characters, narrative, rules). Togelius et al. [2] provide a thorough overview of the field up to 2016. Summerville et al. [4] surveys the PCG approaches based on machine learning models trained on existing content. Liu et al. [5] discuss the recent approaches based on deep learning.

Puzzles are popular game genres and critical ingredients in many games that improve gameplay by asking players to stop and think about solving interesting problems. Compared to other game content types, the application of procedural content generation to puzzles is quite limited. It typically focuses on creating content for actual puzzle games rather than puzzles viewed as accessories to other game genres. De Kegel and Haahr [3] provide an excellent overview of the procedural generation of puzzles covering 32 techniques and 11 puzzle categories. It includes Sokoban-like puzzles, sliding puzzles, tile-matching puzzles, assembly puzzles, path-

[1]https://en.wikipedia.org/wiki/The_Witness_(2016_video_game)
[2]https://en.wikipedia.org/wiki/Tomb_Raider
[3]https://en.wikipedia.org/wiki/Uncharted
[4]https://www.nomanssky.com

978-1-6654-3886-5/21/$31.00 

building puzzles, narrative puzzles, physics puzzles, logic puzzles, word puzzles, and, of course, mazes.

Mazes are puzzles that ask players to find a valid path from a starting position to a goal position while avoiding hazards and learning how to overcome implicit boundaries and obstacles that might obstruct the way. Procedural generation of mazes dates back to the late 1970s when *Beneath the Apple Manor*[5] asked players to traverse ten procedural generated dungeons (with enemies, secret doors, hidden traps, and treasures) to get a golden apple. Two years later, *Rogue*[6] created the genres of rogue-like games. Buck [6] is an excellent reference for most constructive maze generation algorithms. Shaker [7] identifies four categories of maze generation algorithms: (i) space-partitioning (the same used to generate the mazes in this paper); (ii) agent-based algorithms (the same approach we use to inject puzzles in maze graphs); (iii) cellular automata; and (iv) grammars. van der Linden et al [8] focus on controlling the procedural generation of mazes created using cellular automata and generative grammars; Nelson and Smith [9] applied answer set programming solvers for generating different types of mazes, based on a set of constraints expressed in a Prolog-like language. Baron [10] analyzed several algorithms for map generation separating the placement of the rooms and their connection. Aversa et al. [11] proposed an "inventory driven" pathfinding approach based on the grid based Jump-Point-Search [12] algorithm that preserves the optimality guarantees of the original algorithm. Although, [11] does not deal with the building of mazes, the proposed approach may be used as a solver for the maps created by the algorithm presented in this paper. Pereira et al. [13] presented an evolutionary algorithm to generate dungeon maps with locked door missions. Their algorithm works on a tree based representation of the maps and aims at evolving a map that is as close as possible to an configuration provided by a designer. The map has information about the rooms, connections between them, position in a 2D grid, and semantic information for generating narrative. They evaluated the algorithm experimentally showing that it can create maps fitting the designer desiderata, which are also perceived as human-designed. Picariello et al. [14] applied procedural generation for a collaborative maze-solving game for team building that used asymmetric interactions and combined a digital maze exploration game with paper-based tools. Gutierrez and Schrum [15] combined Generative Adversarial Networks (GANs) and graph grammars to generate Zelda-like dungeons using the data collected from The Legend of Zelda to train the neural network. In this case, the focus was on generating rich environment are players engagement which was validated with a user-study.

## III. Algorithm Overview

The algorithm starts from a graph representing the connected areas of a map, with at least one starting area and one goal area. It runs a series of specialized agents that add various types of puzzles to the map (e.g., locked doors, keys, switches) on the paths connecting the starting areas to the goal areas. The specialized agents explore the underlying map graph and create a separate puzzle graph in which nodes represent states in the space of feasible puzzle configurations. Edges identify high-level actions that players can do on the puzzles.

### A. Simple Running Example using Maps

Figure 1 shows an example run using a basic map consisting of four rooms (Figure 1a); the blue marker identifies the starting area, and the green one identifies the goal area; all the rooms are connected. At first, the algorithm runs an agent specialized in building puzzles on collection mechanics (e.g. locks and keys). This adds (i) a collectible item (the blue circle labeled A) to be found in the upper-right room and (ii) a door that blocks the entrance to the bottom-left corner, the red square labeled A (Figure 1b). Next, the algorithm runs an agent specialized in switch mechanics that adds a switch in the bottom-left corner (the blue circle labeled 1) that activates the door leading to the goal area. To reach the goal position, the player should first collect the element A in the upper-right corner, use it to enter the room in the bottom-left corner, find the switch in the room to open the passage leading to the goal, and finally exit the map.

The puzzle elements in Figure 1 are placeholders to indicate the position of the puzzle elements in the underlying graph and their effect on the same graph. For example, collectible item A in Figure 1b might be hidden inside furniture (a treasure chest or a desk drawer) or a non-player character (NPC) might have it. Accordingly, players might need to take some kind of action to get it (thoroughly search the area, negotiate or fight an NPC). Switch 1 in Figure 1c might be a lever positioned in plain sight, a mechanism hidden in the walls, or simply identify the need to turn on some kind of light to see the passage leading to the exit. How the puzzle elements and the connected mechanics are actually added to the in-game map and the level of challenge required to collect/activate them are issues of game design. Our algorithm aims at adding puzzles to map represented as graphs so that there is at least one feasible solution. Most importantly, no sequence of players actions can leave the player stuck with no way of progressing in the map. Their specific in-game implementation is left to the level designers. Thus, Figure 1c is just a graphical representation of the connections between the map areas and to the puzzle elements in the map; it shows what switches and collectibles are available in each area, what mechanisms connect two areas, possibly what activates them, and their effect.

### B. Simple Running Example using Graph Representation

Our algorithm works using two main data structures (i) a Map Graph $MG$, representing the connections between areas; and (i) a Puzzle Graph $PG$, representing how players' actions modify the state of the puzzles and the connections between areas. Accordingly, in the previous example, the algorithm actually started from the map graph $MG$, shown in Figure 2a

[5]https://en.wikipedia.org/wiki/Beneath_Apple_Manor
[6]https://en.wikipedia.org/wiki/Rogue_(video_game)

(top graph), in which node $r_0$ corresponds to the starting area and node $r_3$ to the goal area.

At first, the map has no puzzles and players can move around the map until they reach the goal position. Thus, the algorithm creates the *puzzle graph* ($PG$) depicted in Figure 2a (bottom) with just two (puzzle) states: $ps_*$ for players exploring the map; $ps_g$ for players reaching the goal area. The transition between them is triggered when the player enters the goal area (action $a_g$). A state of $PG$ is characterized by a vector of integers identifying the current state of the puzzles. Initially, there are no puzzles and players can be either exploring the map (state $ps_*$ characterized by vector $\langle 0 \rangle$) or reaching the goal (state $ps_g$ characterized by vector $\langle 1 \rangle$). In this case, none of the puzzle states modifies $MG$.

**Collectible Mechanics.** Next, the algorithm adds a collectible item A that is needed to move between two areas (Figure 1b). The initial map graph $MG$ is modified by disabling the edge between $r_0$ and $r_1$ that can be enabled if when players collect item A. The puzzle state must now take into account whether players have the collectible item and so it will be described by a vector of two integers $\langle p_A, p_g \rangle$: one identifying whether players have collected item A, one for players reaching the exit. The new state $ps_A$ is added to $PG$ to take into account the possession of collectible A (Figure 2b, bottom). The puzzle graphs states are now characterized as follows: $ps_*$ by vector $\langle 0, 0 \rangle$ since when staying in $r_0$ players have no item nor reached the goal; $ps_A$ by vector $\langle 1, 0 \rangle$ since when entering $r_2$ players can collect item A; $ps_A$ also enables the edge between $r_0$ and $r_1$ since when players have item A, they can move between the two areas; $ps_g$ by vector $\langle 1, 1 \rangle$ since to reach the goal, players must have both item A and enter $r_3$. Note that, there is no state corresponding to $\langle 0, 1 \rangle$ in $PG$, since it is impossible to reach the exit before collecting item A, and it is supposed to be not possible to loose the key without using it to open the passage. As before, it is important to note that $PG$ models the map in terms of feasible solutions not gameplay. Accordingly, the transition to state $ps_A$ has only one direction since in term of solution, when reaching $r_2$ players have access to item A allowing them to access $r_1$ from $r_0$. Thus, there is a feasible solution to the puzzle. However, how players get A is an issue of game design, it might require exploration, fighting, negotiation.

**Switch Mechanics.** Finally, the algorithm adds a switch 1 that is needed to move between $r_1$ and $r_3$ so that the edge between the two areas is initially disabled (Figure 2c, top left graph). A new state $ps_1$ is added to $PG$ to model the possible transitions due to the switch status. The vector describing the state of puzzles now takes into account also the state of switch 1 using a vector of three values $\langle p_1, p_A, p_g \rangle$. $ps_*$ corresponds to $\langle 0, 0, 0 \rangle$, $ps_A$ to $\langle 0, 1, 0 \rangle$, $ps_1$ to $\langle 1, 1, 0 \rangle$, and $ps_g$ to $\langle 1, 1, 1 \rangle$; $ps_A$ enables the edge between $r_0$ and $r_1$ (Figure 2c, top right graph); $ps_1$ can enable or disable the edge between $r_1$ and $r_3$. Note that the transition between $ps_1$ and $ps_A$ is bidirectional since players can use the switch by enabling and disabling the edge between $r_1$ and $r_3$, so as to transition in the puzzle space between $ps_1$ ($\langle 1, 1, 0 \rangle$) and $ps_A$ ($\langle 0, 1, 0 \rangle$).

### C. Advanced Example

Figure 3 shows a run for a slightly more complex map. Starting from an empty map (Figure 3a) represented as a connected graph (Figure 3b), first the algorithm adds a collectible item A that is needed to enter the goal area (Figure 3c). Next, it adds the switch 1 in $r_1$ that controls the access to $r_2$ and $r_3$ from $r_0$ and $r_1$ respectively (Figure 3d); initially the switch enables moving between $r_1$ and $r_3$ (the corresponding boxed 1 is green), but disables the edge between $r_0$ and $r_2$ (the corresponding boxed 1 is red); when the switch is activated, the state of the edges swaps. Finally, as shown in Figure 3e, the algorithm adds switch 2 in $r_2$ that controls access to $r_1$ from $r_0$ and to $r_4$ from $r_2$). Figure 3f shows the initial map graph—dashed edges identify the connections that are initially disabled. Figure 3g shows the final puzzle graph. As before, $ps_*$ and $ps_g$ identify the starting and goal areas respectively; edges $a_1$ and $a_2$ are triggered by the state change of switch 1 and 2; edge $a_A$ is triggered when players collect item A. Puzzle states are described by a vector of four variables $\langle p_2, p_1, p_A, p_g \rangle$: $p_2$, $p_1$, $p_A$ describe the status of the corresponding puzzle elements; $p_g$ identify whether players entered the goal area.

### D. Discussion

In the previous examples, we used collectible and switch mechanics to activate connections between areas mainly because their are more intuitive; however, our algorithm can use the same mechanics to build more general scenarios like the one shown in Figure 4, which highlights three rooms of a more complex map. In this case, players must find a way to open a connection (square 1 in the middle-right area). For this purpose, players need to find a key (collectible item A in the top-left area) to open a treasure chest (square A in the bottom-left area) and get a torch, identified by circle B that will remain inactive until players access square A. The torch can be lit when fire (square B) is found in another area (to the right) that will make lever visible (circle 1 will remain inactive until players access square B) that open the connection to another area (square 1). Actions can be even more abstract, like acquiring a certain skill or completing a story quest (a typical scenario in many role-playing games). Furthermore, while our examples show visual map-based environments, our algorithm can also be applied to any graph-based representation of environments like the ones used in text-based adventure/narrative games which typically involve explorations of maps represented as connected graphs (see for example [16]). Our approach is also not limited to the simplified model of players interactions we discussed in the above examples, which assumes players collect items as soon as they enter the area containing it. For example, suppose players must pick up a torch, light it up and use it to burn an obstacle. We can model the collectible item torch using three values in the configuration vector: 0 for the initial state

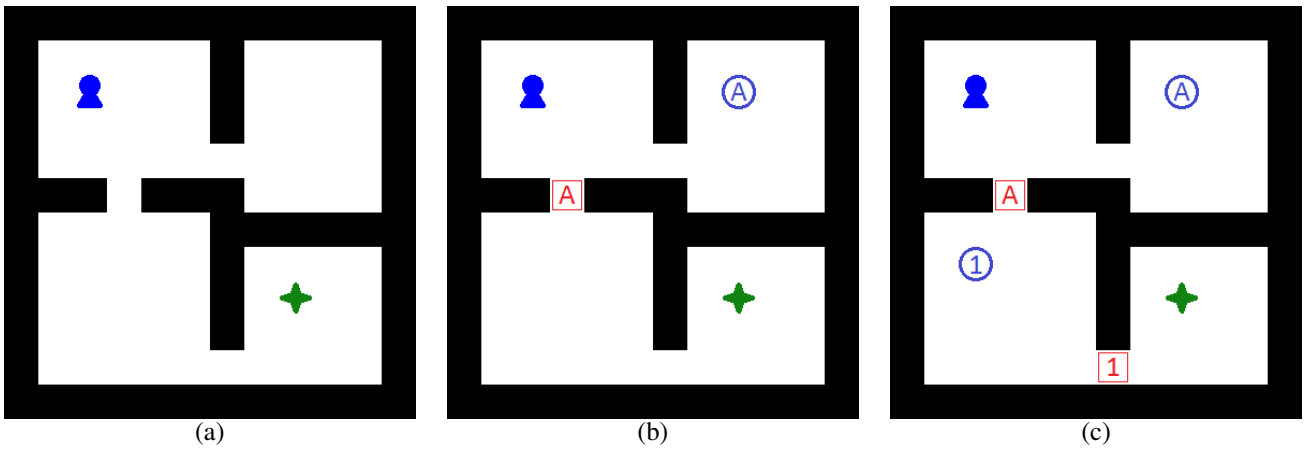


Fig. 1: Example of simple run using a map with four connected areas. The blue marker identifies the starting area. The green marker identifies the goal area. The circled A symbol is a collectible needed to enable the connection blocked by the square A symbol. The circled 1 symbol is a switch that can activate/deactivate the passage marked with the square 1 symbol.

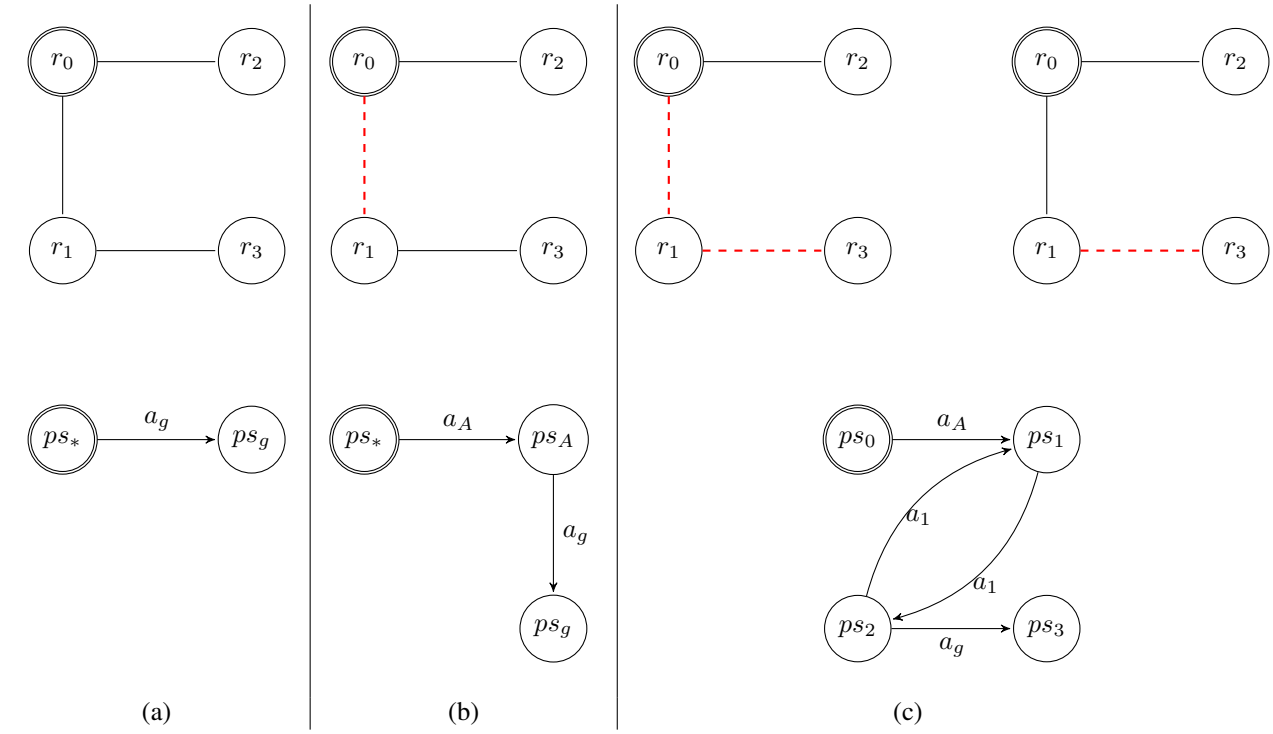


Fig. 2: The graph representation of the maps in Figure 1: (a) the initial map graph (top) and puzzle graph (bottom); (b) the map graph and puzzle graph after the collectible A is added; (c) the final graphs after the switch 1 is added, the top graphs show the initial graph map and the same map after players have collected item A, the bottom graph shows the final puzzle graph. Action $a_g$ is triggered when players enter the goal; action $a_1$ when the player interact with the switch; action $a_A$ when the player collects item A. Dashed lines identify disabled edges.

(players do not have it), 1 when players picked up the torch, and 2 when the torch is lit.

The action to pick up the torch can be activated at any time once the item can be reached by the player, while players can light it up only if they have collected it; in this interaction model, players can drop the torch, going back to the initial state. The algorithm can also model actions that players might perform unintentionally; for example, falling into a water pool would extinguish the torch. This is particularly important to ensure that the player always has a path to the end of the level.

## IV. The Algorithm

Our algorithm takes as input a graph-based representation of a map and a set of specialized *puzzle agents* responsible for adding various types of puzzles to the map. It outputs (i) a modified map graph showing the initial state of the map enriched with the puzzles; and (ii) a puzzle graph specifying how the exiting puzzle mechanics modify the underlying map based on the players' actions.

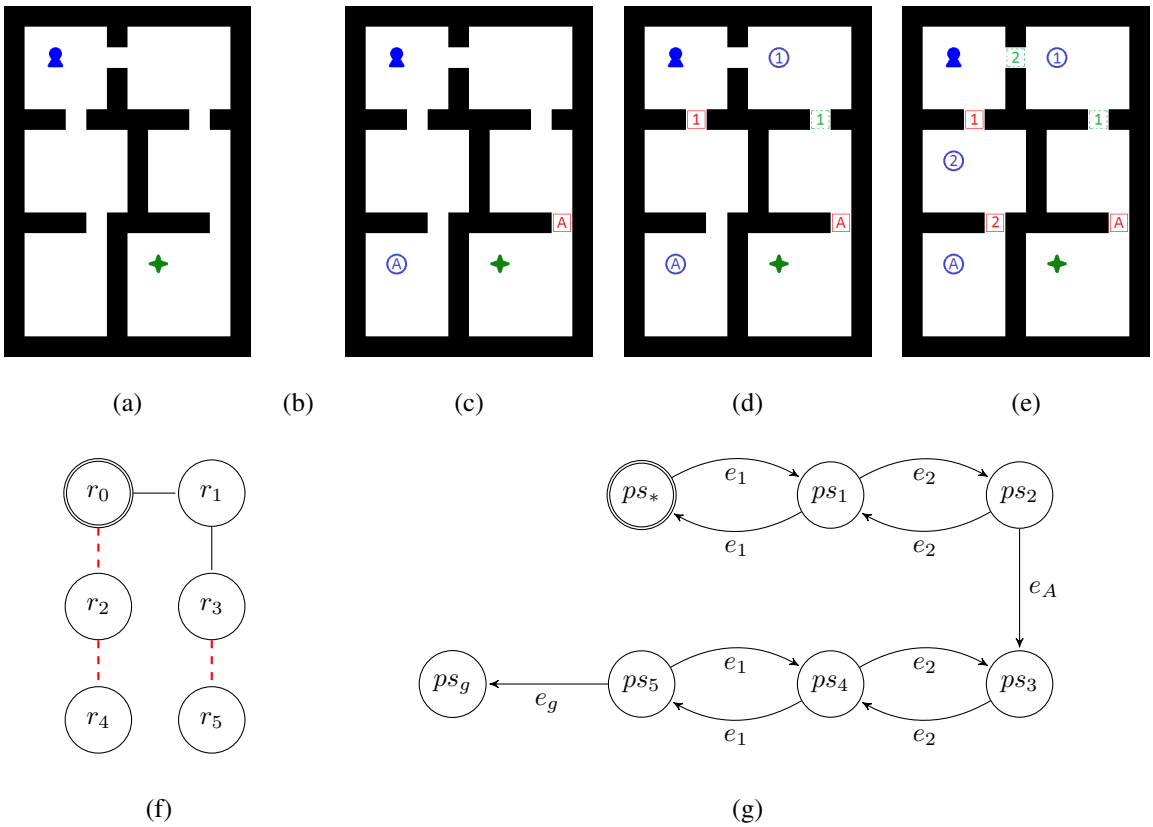


Fig. 3: Example run using a map with six connected rooms (a) represented by the map graph (b). First, a collectible item A is added (c), next two switch mechanisms (d) and (e). This results in the initial map graph (f) and puzzle graph (g). Dashed lines identify disabled edges.

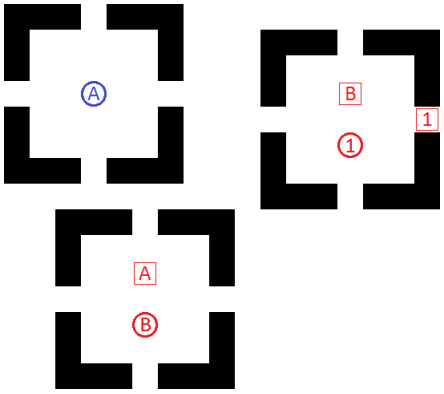


Fig. 4: Example of collectible mechanics applied to elements in the same area.

### A. Map and Puzzle Graphs

A *map graph* $MG$ is a direct graph in which nodes $r_i$ represent areas of the map (e.g., rooms, caves) and edges $(r_i, r_j)$ represent connections between areas; edges can be either enabled or disable depending on whether players can move between the areas. A *puzzle state* $ps_i$ is defined by (i) a map graph $MG_i$ representing the current state of accessibility between map areas; (ii) a set of areas $R_i$ that players can reach given the current graph map $MG_i$; and (iii) a configuration vector of integers $P_i = \langle p_0 \dots p_k \rangle$ describing the state of each puzzle in the map.

A *puzzle graph* $PG$ is a direct graph in which nodes are puzzle states $ps_i$ edges between states are labeled with the players' actions $a_j$ that can cause such a transition between puzzle states. Actions $A$ in $PG$ can thus be viewed as functions defined over the space of feasible puzzle states. These actions comprise all those events that are (intentionally or unintentionally) triggered by players and modify status of the map by enabling or disabling edges. For example, burning a barrier with a torch to open a passage, fighting or negotiating an NPC to get a key are all examples of actions that modify the puzzle graph. In contrast, interactions that do not modify the status of the puzzles are not actions in the puzzle graph, like for example, fighting a random enemy or talking to an NPC, are not encoded in the puzzle graph. The configuration vector $P_i$ of $ps_i$ keep track of the outcome of the actions that must be performed to reach a state. For example, $P_i$ would include a 0/1 value for every collectible item and any switch mechanic needed to reach the goal area. Its values are not limited to 0 and 1 since we can have mechanics that require more values, like for example, a lever mechanic with more positions, a container that must be filled with a certain amount of water to activate another element.

In a puzzle graph, we label the initial puzzle state as $ps_*$ and the goal puzzle state as $ps_g$. The map graph $MG_*$ connected to the initial puzzle state $ps_*$ represents the initial situation that players will find when entering the map (e.g., the top

graph in Figure 2b); $R_*$ for the same state includes the areas that are initially accessible to the players when entering in the starting position (e.g., $r_0$ and $r_2$ in Figure 2b); $P_*$ have all the values set to zero that, by definition, represents the initial state of every puzzle in the map (e.g., $\langle p_A, p_g \rangle$) in Figure 2b).

### B. Puzzle Agents

They are the core of our algorithm that, given a puzzle graph $PG_i$, create $PG_{i+1}$ by adding a new puzzle to the original graph. As a result, the agent increases the number of actions available to players (since an additional puzzle means more opportunity for players' interaction) and it will modify the number of variables required to describe the current puzzle configuration in each state of the graph (since there is one more puzzle to keep track of). In the previous example (Figure 2), starting from the original puzzle graph (Figure 2a), an agent blocked the connection between $r_0$ and $r_1$ that could be activated by the collectible item A, this added a new action $a_A$ that the players needed to perform to reach $r_1$; at the same time, the set of areas that players could reach from the starting area was reduce to $r_0, r_2$; finally, the size of the configuration vector of each puzzle state was increased by one variable ($p_A$) that was needed to keep track of the status of the new puzzle (Figure 2b). Agents can apply various strategies to add a puzzle to the current graph. They can be deterministic or stochastic; they can add a puzzle in any random position of the map or apply a strategy (e.g. they can first check the areas near the goal or those near the start position).

### C. The Main Loop

Algorithm 1 shows the pseudo-code of our algorithm main loop. It takes an initial map graph $MG$ and a list of available puzzle agents $MG$. Initially, line 1, it creates a puzzle graph $PG$ from $MG$, with two states $ps_*$ and $ps_g$ and one action $a_g$; the state configuration consists of just one variable $\langle p_g \rangle$; $R_*$ contains all the areas reachable from the start position. It inits to zero the counter, *failures*, for the infeasible puzzle graphs generated (line 2). Then, it repeats the following loop until an end condition is met (lines 3-12): it selects a puzzle agent from the available ones and (line 4) applies the agent to add a puzzle to PG to generate a new candidate puzzle graph PG$_{new}$ (line 5); this will have new states, new actions, and a richer state configuration vector. If the new graph has at least one feasible solution (line 6), it becomes the current puzzle graph and its complexity is computed (line 8); if PG$_{new}$ has no feasible solution, then the failure counter is updated; finally, it checks whether the process should stop (line 11). Agents can be selected using various policies; they could be selected completely at random or using some constraints about the number of puzzles of each type (for example, by requiring a certain number of puzzles of each type); they could even be selected deterministically by imposing a sequence of puzzle types to generate. Agents can select the position of puzzles completely at random or a specific strategy (for example, by requiring that the new puzzle is added as near as possible to the goal area). We currently compute the feasibility and complexity of a puzzle graph using Dijkstra's algorithm [7]. A puzzle graph is feasible if there is a path from the starting puzzle state $ps_*$ to a state in which the configuration vector has $p_g$ set to one (which signals that the goal area can be reached); its complexity is computed as the length of the shortest path between the same two nodes. More importantly, our algorithm also guarantees that the goal state $p_g$ can be reached from any other state, thus players will always be able to reach the goal no matter what sequence of actions they perform. This is an important feature to avoid players frustration and increase engagement.

There are a number of conditions that can be used to end the process: it can be stopped if the number of failures exceeds a certain threshold (this usually happens when the map is too simple with respect to the number of puzzles we wish to add), when the number of puzzles added and their type satisfy our objective (for instance, when we set a fixed number of puzzles for each type), or when we reached a target complexity level (the minimum number of actions on the puzzles in above a certain threshold).

**Algorithm 1:** Main loop.

```
   Data: Map graph MG; a set of Puzzle Agents PA
   Result: Puzzle graph PG
1  PG ← CreatePG(MG)
2  failures ← 0
3  repeat
4      agent ← SelectAgent(PG, PA)
5      PG_new ← agent.AddPuzzle(PG)
6      if (PG_new is not null) and SolutionExists(PG_new)
         then
7          PG ← PG_new
8          complexity ← SolutionComplexity(PG)
9      else
10         failures ← failures + 1
11     finished ← ShouldStop(PG, failures, complexity)
12 until finished;
13 return PG
```

### D. Example of Puzzle Agent Implementation

Algorithm 2 shows the pseudo-code for the puzzle agent for collective items. Initially, line 1, it selects a state $ps$ from the current puzzle graph $PG$ using a given strategy (e.g., completely at random or starting as near to the goal or start as possible). Then, it looks for a bottleneck in the map graph of $ps$ (line 2); this can is implemented with a simple heuristic but it could also be selected using the Girvan-Newman[8] method and *edge betweenness*. Next, it looks for one of the most isolated area with respect to the bottleneck position (line 5); also in this case, we implemented a simple heuristics but more advanced graph algorithms might be employed. In both

[7] https://en.wikipedia.org/wiki/Dijkstra's_algorithm

[8] https://en.wikipedia.org/wiki/Girvan\OT1\textendashNewman_algorithm

cases, if a bottleneck or an isolated area cannot be found the function returns *null* signaling that it failed. Finally, given the bottleneck connection $\langle r_i, r_j \rangle$ that will be blocked with an item in $r_t$, the puzzle graph is updated by (i) adding a new puzzle state, a new action, and required additional edges to the current PG, (ii) computing the map graph for the added state, and (iii) adding a variable that keeps track of the status of the new puzzle in the configuration vector of puzzle states. Finally, the updated puzzle graph is returned.

**Algorithm 2:** Puzzle agent for collectible items.

```
Data: Puzzle Graph PG
Result: Puzzle Graph PG
ps ← SelectPuzzleState(PG)
⟨r_i, r_j⟩ ← GetBottleneck(ps.MG)
if ⟨r_i, r_j⟩ is null) then
    return null
r_t ← GetIsolatedArea(ps.MG)
if r_t is null) then
    return null
PG_new ← UpdatePuzzleGraph(PG, r_t, ⟨r_i, r_j⟩)
return PG_new
```

The agents specialized in switch placement follow a similar approach, but they do not identify bottlenecks. Instead, they isolate two different parts of the map to make them accessible in a mutually exclusive way. Ideally, switch agents try to select areas that contain actions that are necessary to proceed in the game. Since switch and collective actions are entirely abstract, several agents can work together to create complex emergent puzzles.

## V. Experimental Evaluation

We performed a series of experiments to evaluate our algorithm in terms of computation time and complexity of the generated maps. We applied a binary space partition algorithm [17] to generate grid maps of different complexity similar to the ones used in the examples previously discussed [2], [17]. The map generator was run 10000 times, with 7×7 grids up to 22×22 grids, with a minimum room size of 2×2, generating maps with 4 up to 45 rooms (i.e., graphs with 4-45 nodes). We focused on collectible and switch mechanics and applied our algorithm to each map to add as many puzzles as possible. Starting with an empty map, the algorithm added a puzzles to the map using a puzzle agent, which was randomly selected, until the agents failed more than ten times (see Algorithm 1).

Table I reports the statistics computed over ten runs using mean and median. Column *Grid* reports the size of the grid that was used to generate the map when using rectangular shapes (like 10×15), half the runs (5000) were performed using the reported shape, half (the other 5000) were performed swapping the width and height parameters (e.g., 15×10). Column *Rooms* reports the average number of rooms generated for the corresponding grid and the median value; note that, for small grids all the runs of the runs created the same number of rooms (first three rows of Table I); the largest map created consisted of 45 rooms. Column *Complexity* reports the number of actions that players must perform to reach the goal computed as the *minimum number of transitions* needed to reach the puzzle state goal $ps_g$ from the starting node $ps_*$. For collectible items, we used the same approach used in the previous examples (Section III) and modeled with just one transaction in the puzzle graph two main events, the players entering the area *and* collecting the item. Thus, the reported complexity represents the minimum number of *puzzle actions* that players need to perform. Column *Time* reports the CPU time in number of seconds required to generate one single map on a iMac 27 3.8 GHz i7. As can be noted, the algorithm takes around one second to generate a map with around 30 rooms that players can traverse using around 11 transitions on the puzzle graph.

| Grid Size | # Rooms $\mu$ $(median)$ | Complexity $\mu$ $(median)$ | Time $\mu$ $(median)$ |
|---|---|---|---|
| $7 \times 7$ | 4.0 (4) | 2.2 (2) | 0.001 (0.001) |
| $7 \times 12$ | 6.0 (6) | 3.1 (3) | 0.002 (0.002) |
| $12 \times 12$ | 9.0 (9) | 4.7 (5) | 0.015 (0.009) |
| $17 \times 12$ | 13.2 (13) | 7.0 (7) | 0.114 (0.059) |
| $17 \times 17$ | 19.1 (19) | 8.8 (9) | 0.366 (0.199) |
| $17 \times 22$ | 25.0 (25) | 9.9 (10) | 0.710 (0.365) |
| $22 \times 22$ | 32.1 (32) | 10.7 (11) | 1.272 (0.618) |

TABLE I: Columns report the grid size used to generate the maze and the average number of rooms, puzzle complexity, and CPU time to generate one maze. Statistics are averages over 10000 runs.

## VI. Conclusions

We presented an algorithm that, starting from a graph representation of a map, runs specialized agents for adding several types of puzzles. The algorithm generates maps that have at least one acceptable solution. Most importantly, they are robust to players' actions in that there is no sequence of actions that might leave players stuck in an unsolvable situation. We evaluated the algorithm experimentally by applying it to generate 10000 maps of varying sizes and measured the average CPU time taken to generate one map and the complexity of the generated puzzles. Our results show that the algorithm can create quite complex maps in a small amount of time; in fact, it requires less than a second to generate maps with an average of 32 rooms that requires 11 actions on the puzzles to make the goal accessible to players.

The algorithm is also very modular since more puzzles can be added by implementing more specialized agents. In contrast, other approaches (e.g., [13]) might require more structural modifications (e.g., probably a different crossover/mutation operators and fitness function). Future works include the evaluation using (i) other map generation algorithms, like the ones inspired by Zelda [13], [15], or based on cellular automata [8]; (ii) more agent types. We plan to improve the evaluation of map complexity by introducing solvers that can model players' behavior, like the inventory-driven solver introduced in [11]. We also plan to extend the algorithm by adding a preprocessing phase to identify connected components in the initial map graph and then applying

a separate instance of the algorithm to each component. This will reduce the computational complexity while improving the overall level design by more modular. Finally, we plan to have a user study to compare the map complexity's evaluation using the puzzle graph and solvers against the complexity as perceived by players.

In the end, we wish to point out that, although we presented our approach using relatively simple agents in maze-like maps, we can extend the approach to model much different and more complex scenarios. Our algorithm allows several agents to work on a shared task at potentially very different levels. For example, an action can represent a simple gameplay interaction (like picking up a key) or an entire sub-puzzle area generated by another set of agents. Thus, partitioning the creation of complex maps into different phases would benefit both the computation time and the player experience. Moreover, each puzzle element can contain additional procedural generation aspects. For example, even collecting a simple key can become much more enjoyable if placed at the end of a series of jumps because placed in a room with procedurally generated platforms.

## REFERENCES


[1] J. Schell, *The Art of Game Design A Book od Lenses*, 3rd ed. CRC Press Taylor & Francis Group, 2019.

[2] J. Togelius, N. Shaker, and M. J. Nelson, *Procedural Content Generation in Games*, ser. Computational Synthesis and Creative Systems. Springer, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-42716-4

[3] B. de Kegel and M. Haahr, "Procedural puzzle generation: A survey," *IEEE Transactions on Games*, vol. 12, no. 1, pp. 21–40, 2020. [Online]. Available: https://doi.org/10.1109/TG.2019.2917792

[4] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (PCGML)," *IEEE Trans. Games*, vol. 10, no. 3, pp. 257–270, 2018. [Online]. Available: https://doi.org/10.1109/TG.2018.2846639

[5] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Comput. Appl.*, vol. 33, no. 1, pp. 19–37, 2021. [Online]. Available: https://doi.org/10.1007/s00521-020-05383-8

[6] J. Buck, *Mazes for Programmers: Code Your Own Twisty Little Passages*. Pragmatic Bookshelf, 2015.

[7] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, *Constructive generation methods for dungeons and levels*. Cham: Springer International Publishing, 2016, pp. 31–55. [Online]. Available: https://doi.org/10.1007/978-3-319-42716-4_3

[8] R. Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, pp. 78–89, 03 2014.

[9] M. J. Nelson and A. M. Smith, *ASP with Applications to Mazes and Levels*. Cham: Springer International Publishing, 2016, pp. 143–157. [Online]. Available: https://doi.org/10.1007/978-3-319-42716-4_8

[10] J. R. Baron, "Procedural dungeon generation analysis and adaptation," in *Proceedings of the 2017 ACM Southeast Regional Conference, Kennesaw, GA, USA, April 13-15, 2017*. ACM, 2017, pp. 168–171. [Online]. Available: https://doi.org/10.1145/3077286.3077566

[11] D. Aversa, S. Sardiña, and S. Vassos, "Path planning with inventory-driven jump-point-search," in *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2015, November 14-18, 2015, University of California, Santa Cruz, CA, USA*, A. Jhala and N. Sturtevant, Eds. AAAI Press, 2015, pp. 2–8. [Online]. Available: http://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11532

[12] D. D. Harabor and A. Grastien, "The JPS pathfinding system," in *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*, D. Borrajo, A. Felner, R. E. Korf, M. Likhachev, C. L. López, W. Ruml, and N. R. Sturtevant, Eds. AAAI Press, 2012. [Online]. Available: http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5396

[13] L. T. Pereira, P. V. S. Prado, and C. Toledo, "Evolving dungeon maps with locked door missions," in *2018 IEEE Congress on Evolutionary Computation, CEC 2018, Rio de Janeiro, Brazil, July 8-13, 2018*. IEEE, 2018, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CEC.2018.8477718

[14] U. Picariello, D. Loiacono, F. Mosca, and P. L. Lanzi, "A framework to create collaborative games for team building using procedural content generation," in *2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020, Canberra, Australia, December 1-4, 2020*. IEEE, 2020, pp. 2365–2372. [Online]. Available: https://doi.org/10.1109/SSCI47803.2020.9308431

[15] J. Gutierrez and J. Schrum, "Generative adversarial network rooms in generative graph grammar dungeons for the legend of zelda," in *IEEE Congress on Evolutionary Computation, CEC 2020, Glasgow, United Kingdom, July 19-24, 2020*. IEEE, 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CEC48606.2020.9185631

[16] "Zork - The Great Underground Empire." [Online]. Available: http://adventuresinzork.blogspot.com

[17] G. Uribe, "Dungeon Generation using Binary Space Trees." [Online]. Available: https://medium.com/@guribemontero/dungeon-generation-using-binary-space-trees-47d4a668e2d0