The Dungeon Variations Problem Using Constraint Programming

Gaël Glorian
LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France
Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

Adrien Debesson
Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

Sylvain Yvon-Paliot
Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

Laurent Simon
LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France

Abstract

The video games industry generates billions of dollars in sales every year. Video games can offer increasingly complex gaming experiences, with gigantic (but consistent) open worlds, thanks to larger and larger teams of developers and artists. In this paper, we propose a constraint-based approach for procedural dungeon generation in an open world/universe context, in order to provide players with consistent, open worlds with an excellent quality of storytelling. Thanks to a global description capturing all the possible rooms and situations of a given dungeon, our approach allows enumerating variations of this global pattern, which can then be presented to the player for more diversity. We formalise this problem in constraint programming by exploiting a graph abstraction of the dungeon pattern structure. Every path of the graph represents a possible variation matching a given set of constraints. We introduce a new propagator extending the "connected" graph constraint, which allows considering directed graphs with cycles. We show that thanks to this model and the proposed new propagator, it is possible to handle scenarios at the forefront of the game industry (AAA+ games). We demonstrate that our approach outperforms non-specialised solutions consisting of filtering only the relevant solutions *a posteriori*. We then conclude and offer several exciting perspectives raised by this approach to the *Dungeon Variations Problem*.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases constraint programming, video games, modelization, procedural generation

Digital Object Identifier 10.4230/LIPIcs.CP.2021.27

Supplementary Material *Software (Source Code)*: `https://bit.ly/3xSXK2B`

Funding This work was supported by the "KIWI" project of the Nouvelle-Aquitaine region.

1 Introduction

The video game industry is an important economic sector, generating billions of dollars each year, at the forefront of innovation. Since the appearance of the first games in the 1970s, the landscape of this industry has changed dramatically. Video games are now produced by large teams of artists and developers, offering photo-realistic graphics, exciting and complex scenarios, with a constantly improved degree of simulation.

One of the biggest challenges of the gaming industry is to be able to build open worlds, in which users must feel free and where a huge number of actions are possible for them. Generating such a world without the final intervention of a *Level Designer* (*LD*) to validate the level is still a dream: a single inconsistent game level can quickly shatter the reputation of a game, possibly costing millions of dollars. The role of *LD* is to ensure the consistency of all playable levels, and the interest of every level w.r.t the overall scenario. The formal

Figure 1 A source dungeon.

Figure 2 Some variations of the source dungeon (Figure 1). In green, the rooms and corridors preserved in the variations.

verification of generated levels will probably play an important role in the future of the gaming industry, but, for now, it is still impossible. Instead of formally verify the properties of the generated levels *a posteriori*, we propose, in this article, to enforce the desired properties of generated levels by construction. This work is developed in collaboration with an AAA+ video game studio.

Intuitively, in the proposed approach, an *LD* produces what we call a *source dungeon*, which contains a set of rooms connected by corridors. Each room has its properties and areas identified for possible situations (fights, treasures, ...). The *Dungeon Variations Problem* is the problem of generating an appropriate variation of the source dungeon by disabling a subset of initial rooms and corridors so that the remaining set of rooms matches a given set of constraints (e.g. at least one entrance, at least a given number of monsters on any path). The consistency of a source dungeon can also be validated by generating a variation taking into account all rooms as well as corridors.

While this approach does not yet fully address the global problem of procedural content generation [19], it opens up a challenging new field for constraint programming. Additionally, it allows level designers to check the consistency and quality of dungeon variations before delivering the game. We formally specify the *Dungeon Variations Problem* in Section 2 and describe it as a constraint programming problem using the graph constraint *connected* in Section 3. We then introduce, in Section 4, a new propagator extending this constraint (*connected+*) which takes into account directed graphs with cycles. In the experimental part (Section 5), we show that this propagator offers a spectacular improvement over a more naive *filtering* approach. Before concluding, we propose a list of interesting research topics, which could extend our work and have an important impact for the video game industry, providing novel challenges for constraint programming.

2 The *Dungeon Variations Problem*

2.1 Motivations

Our goal is to provide a game creation assistant tool for level designers (*LD*). This tool should provide *LD* an efficient way of building level variations to fill the open world at distinct places. This is a restricted view of the more general problem of procedural game generation, the aim of which is to guarantee high-quality automatic content generations. In these games, each level must be consistent with the narrative and the user's progression, which is not yet possible with fully autonomous level generation solutions. Delivering an AAA+ game with an unplayable level, or an inconsistent story can have dramatic effects in terms of images and costs for the responsible game studio. In order to reach the best possible quality, each level is handcrafted and validated by a *LD* guaranteeing a precise purpose of this level in the overall story. Our approach is thus not to generate levels from scratch but to propose variations (with guaranteed properties) of a dungeon pattern designed by an artist. All the variations have, by design, a strong story consistency (monsters, possible quests, ...).

We call *dungeon source* the original *design* (see for instance a very simple example of such a design Figure 1). This dungeon contains *rooms* with properties (in practice, the concept of *rooms* can be misleading: a room can be composed of a set of areas which will be considered as a whole block). A room can be an *entry* (a connection from the outside), an *exit* (a connection to the outside). Rooms can also have a set of *tags* to encode gameplay situations, design, or any other optional set up. The set of rooms that the user can explore and the situations (e.g. rewards, fights) encountered is called the *flow*. It is a common practice in level designing to add *final* rooms to dungeons to allow side quests or optional explorations. These final rooms, which are *dead-end* rooms (other than the hallway leading to them), usually contain treasures, keys, special items or monsters, but do not block the player. They are important for players because, in addition to optional rewards, *final* rooms offers a non-linear way of exploring the dungeon, which is a crucial aspect for the positive perception of levels by players. It should be noted that the *connections* between the rooms are directed (a door can be one-way access, the player can fall from one room to another, ...).

The goal of the *Dungeon Variations Problem* is to generate a coherent subset of rooms from the source dungeon satisfying some constraints (see for instance Figure 2). The first set of constraints is structural: each set of rooms must be playable (all rooms are connected, we have at least one entry and one exit), and the final rooms must be tagged as such (the role of the *LD* is typically to be able to check the interest of final rooms). The second set of constraints are *configuration* constraints, which are optional and can be defined by the user: some special rooms can be forced to be *active*, *entry*, *exit* or *final* in the solution. Our tool will have to offer, within a reasonable time (a few seconds), interesting variants to the *LD*, as illustrated, for example, Figure 2. The *LD* will set some tags defining the desired solutions to explore and will be able to navigate through a number of variations.

Procedural Generation of levels and flows is not new [6, 17, 18]. However, as pointed out in these references, systems are not yet mature to be autonomous and AAA+ studios are only using them in very restricted subareas of game creations. Moreover, none of these works has the potential to reach the level of quality expected in our context. To the best of our knowledge, no previous work is able to handle both the dungeon generation (structure of the dungeon) and the story itself (which is a cornerstone of our work, and will be ensured by the definition of the source dungeon). As described in the next section, we propose to tackle this problem by defining it as a graph problem. The connectivity property will be guaranteed by

a new extension of the graph constraint [5, 14] adapted to our problem. To illustrate this formalisation, Figures 3 and 4 show simplified versions of a graph-encoding of two generated variations used in production.

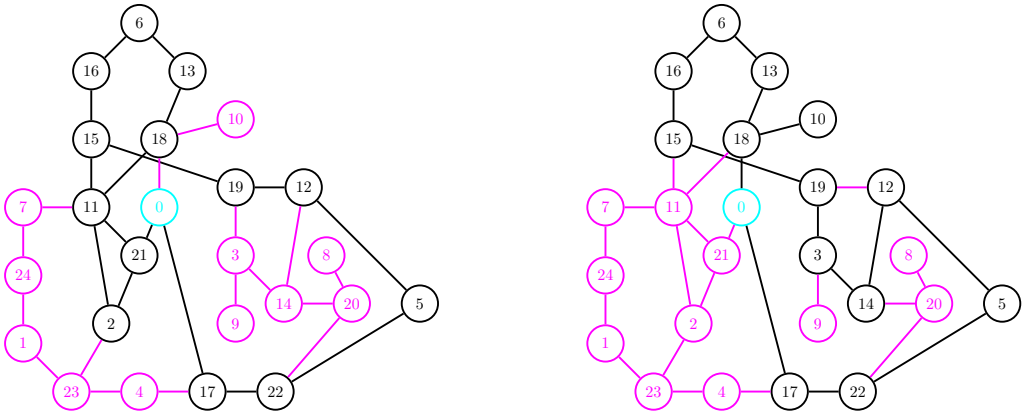

Figure 3 Two generated variations of a source dungeon used in production for a AAA+ video game (for clarity sake, we did not represent oriented edges in this example). We can see the differences between the two variations on the use of final rooms and alternative paths. The blue node 0 is both the entry and the exit of this level. Nodes and edges in red are not kept in each variation.

2.2 Related Works

Most work on constraints and graphs (e.g. [3, 4, 8, 11, 15]) generally consider one source node and one objective node with path/circuit problem. Unfortunately, we cannot rely directly on these works. They are either not applicable or too constrained: by definition, the path (or circuit) problem force the edges to be used only once, which is not relevant in our case. In addition, we have to consider the fact that the graphs are directed, and each variation can have a lot of source nodes (inputs) as well as several objective nodes (outputs). Of course, in our case, a node can be an entry and an exit at the same time[1], which is another argument to develop a novel approach. It is more classical, in state-of-the-art approaches, to suppose that entries and exit nodes are distinct (e.g., in path constraint [7]). Using the *reachable* constraint from every entry will unfortunately introduce a prohibitive overhead. It is also important to note, at this point, that we are not yet trying to optimise the paths in any way. We plan to do this as part of a further work by, for example, considering path optimisation of optimal objects placements [11]. It may also be important to optimise the flow in many additional ways, for example, by optimising a constraint-based formalisation of the fun as perceived by the player, or by optimising the hardware cost of rendering the scenes during the flow in order to guarantee a graphical performances.

3 CP model of the *Dungeon Variations Problem*

In this section, we formulate the problem as a constraint satisfaction problem. The constraints are presented in natural language at first, then in clausal and set form (Table 1).

[1] In practice, in real-world problems, this often happens.

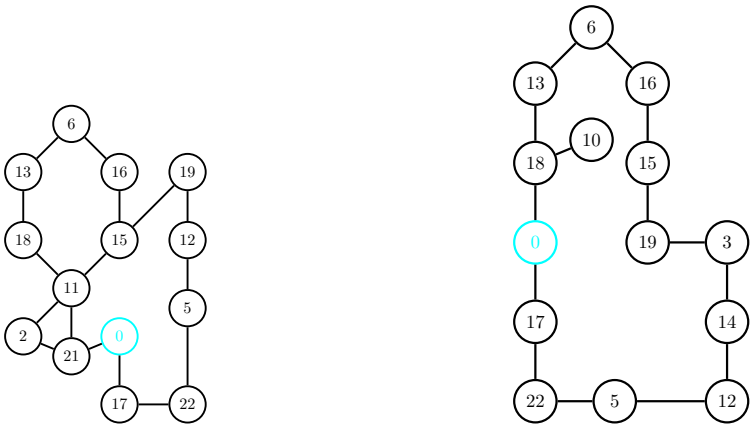

Figure 4 The two variations of Figure 3 with the remaining edges and nodes, reported for simplicity. The blue node 0 is both the entry and the exit of this level.

3.1 Model Variables

Let n be the number of nodes in the pattern (numbered from 0 to $n-1$). The connections (e.g. *corridors*) between the nodes are identified by node numbers (e.g. C_{ij} is a connection from node i to node j). We use six arrays of size n to model the problem: Four arrays of *booleans* `entries`, `exits`, `actives` and `finals` which indicates, respectively, if a node is an entry, an exit, if it is active and if it is final. Two arrays `sumToNode` and `sumFromNode` which respectively counts the number of connections entering each node and leaving each node. To encode the connections between rooms, we use an $n \times n$ adjacency matrix C that can indicate the active edges (connections) of the graph since the nodes can have multiple successors. Arrays `sumToNode` and `sumFromNode` are defined using sum constraints. We also avoid multiple corridors between the same rooms. Such a tricky variation, when needed, can be handled at another level (e.g. on top of a variation that admits two connected rooms allowing multiple connections). To prevent trivial loops, our CP model must also block the diagonal of the adjacency matrix C.

Formally, the problem can be expressed as a graph problem in a fairly simple way: let $G(V, E)$ be a graph with $V \subset \mathbb{N}$ represents the rooms and $E \subset \{(i, j) \mid i, j \in V \wedge i \neq j\}$ the corridors. The goal of the *Dungeon Variations Problem* is to generate a graph $G'(V', E')$ with $V' \subset V$ and $E' \subset E$ with additional constraints encoding the fact that the variation is playable. Note that `actives` $\equiv V'$. The sets `entries`, `exists` and `finals` are subsets of V'.

3.2 Model Constraints

The constraints of the model ensure the consistency of each variation, given in natural language in the following list (see Table 1 for the clausal and set form). These constraints are expressed as intention constraints in the *XCSP3* model [1]:

(1) If a node is an entry, it must be active.
(2) If a node is an exit, it must be active.
(3) If a node is final, it must be active.
(4) If a node is an entry, it cannot be final.
(5) If a node is an exit, it cannot be final.
(6) If a connection is used, then the associated nodes must be active.
(7) If a node is active, at least one connection must go to (or leave from) this node.

Table 1 Constraints in clausal and set form. Note that the clausal form, lines 8-11, can be encoded as a single set constraint.

	Clausal form	Set form
1	$entries_i = 0 \vee actives_i = 1$	$i \in entries \Rightarrow i \in V'$
2	$exits_i = 0 \vee actives_i = 1$	$i \in exits \Rightarrow i \in V'$
3	$finals_i = 0 \vee actives_i = 1$	$i \in finals \Rightarrow i \in V'$
4	$entries_i = 0 \vee finals_i = 0$	$entries \cap finals = \emptyset$
5	$exits_i = 0 \vee finals_i = 0$	$exits \cap finals = \emptyset$
6	$C_{ij} = 0 \vee (actives_i = 1 \wedge actives_j = 1)$	$(i,j) \in E' \Rightarrow \{i,j\} \in V'$
7	$actives_i = 0 \vee sumFromNode_i > 0$ $\vee sumToNode_i > 0$	$i \in V' \Rightarrow \exists j \mid ((i,j) \in E' \vee (j,i) \in E')$
8	$finals_i = 0 \vee sumToNode_i = 1$	$i \in finals \Leftrightarrow \exists! j \mid (\{(i,j),(j,i)\} \subset E')$
9	$finals_i = 0 \vee sumFromNode_i = 1$	See 8.
10	$finals_i = 0 \vee C_{ij} = 0 \vee C_{ji} = 1$	See 8.
11	$finals_i = 0 \vee C_{ij} = 1 \vee C_{ji} = 0$	See 8.
12	$C_{ij} = 0 \vee C_{ji} = 0 \vee sumToNode_i \neq 1$ $\vee sumFromNode_i \neq 1 \vee finals_i = 1$	$\{(i,j),(j,i)\} \subset E' \wedge \ \forall k \ \lvert\{(i,k)\} \in E'\rvert = 1$ $\wedge \ \forall k \ \lvert\{(k,i)\} \in E'\rvert = 1 \Rightarrow i \in finals$

(8) If a node is final, then one and only one connection must go to this node.

(9) If a node is final, then one and only one connection must leave from this node.

(10) If a node i is final and one connection goes from i to j then a must go from j to i.

(11) If a node i is final and one connection goes from j to i then a must go from i to j.

(12) If there is a round trip between two nodes i and j and there is only one connection to i and from i, then the node i must be final.

The above set of constraints allows to specify the structure of the desired variations (the problem of unconnected solutions will be discussed in the next section). Note that the management of *final* rooms is covered by lines 8 to 11. It ensures that the player can return to the main path from a dead-end, possibly corresponding to a secondary quest. This set of constraints (lines 8–11) is encoded as a single constraint when using the set-based form, Table 1.

Some other constraints, called *configuration* constraints, are also allowed in the proposed tool, but not reported here for simplicity (for instance, the tool allows the *LD* to limit the number of inputs, outputs, active and final rooms). It is also necessary that our tool allows the user to force specific parts to be active (or not) in all the generated variants (for instance, the *LD* may activate or deactivate certain parts of the source dungeon depending on where the variation will be placed on the map, for instance). The user can also deactivate a subset of corridors to explore the corresponding variations (for instance, some corridors may require the user to use some items/capacities that may not be available at all steps of the overall story). Likewise, it is possible to give control over situations (and therefore over the dungeon *flow*) by limiting the number of rooms with some given tags. For example, if we have 8 rooms marked as a combat room in a source dungeon, an *LD* may want to generate a variation with only 3 of them. To handle these limits easily, we can create a sum constraint on the *active* property of the tagged rooms. We do not further describe this part of the model since it is covered by a classical constraint-based approach and can be handled as soon as the overall problem is tackled by CP.

Figure 5 The graph displayed on the left represents a source dungeon (or a variation) with two entries (nodes A and G) and one exit (node B). We can see that the node H is not accessible and will be filtered by the constraint introduced in Section 3.2. A *cul-de-sac* is also present in this dungeon (nodes C and D). Indeed, when entering node C, a player would be stuck inside the two rooms without any possibility to escape this "trap". This kind of *cul-de-sac* is not filtered by the structural constraints and are not welcome in the resulting variations. The goal of the *connected+* constraint is to filter these cases and obtain the dungeon displayed on the right.

4 The Connected Constraint Extended

As mentioned before, we need to make sure that our variations are connected to satisfy all the constraints. We can ensure this in two different ways. (1) We may verify each solution, afterwards, with an *ad hoc* algorithm (this is a classic approach in video games: the levels generated can be tested subsequently by algorithms, bots and/or humans) or, (2) we may force any proposed variation to be correct by construction, filtering out partial solutions as soon as possible when looking for variations. Let us recall here that the goal of our approach is to help the *LD* navigating through the possible alternatives as easily as possible. Being able to guarantee that each proposed solution has certain properties by construction is, therefore, a crucial element. Figure 5 shows a very simple example of the connected+ constraint we propose, in practice.

The approach we propose is a two-level propagator that first relies on a simple implementation of the constraint `connected` [5] on the undirected version of the graph (Algorithm 1). Then, another level of filtering is added (Algorithm 4 and 3) to guarantee the validity of the paths. The `connected` constraint guarantees that we do not have disjoint dungeons but does not consider directed graphs. We, therefore, have no guarantee that the dungeon is playable (i.e. all nodes are accessible from the entrances, in particular, the exit nodes). To overcome this problem, we introduce an algorithm to find paths from each entry to at least one exit (this algorithm can be extended in many ways, for instance to handle the size of the paths, whether they are constrained or not, or to obtain information (average length, etc.)). For this reason, the propagator need to know the values of the `entries` and `exits` arrays.

Algorithm 1 starts from an active node (line 2, the selectActiveNode() function selects an active node in the graph, i.e. a node x such that $actives_x = 1$, or returns an empty set if no active node is found). Then, it follows each connection by considering the undirected version of the graph (line 12). From line 3 to 7, if we do not find any active node, we need to check the consistency of the graph. Indeed, if some of the nodes are not yet decided (i.e. their Boolean domain size is 2), the constraint is consistent since we cannot filter anything. Otherwise, an inconsistency is detected if all the nodes are marked inactive (lines 4–5). Until the stack is empty, we mark the current node (line 11) and add its neighbours to the stack (line 12). When the stack is empty, Algorithm 2 is called to filter the nodes and to check

Algorithm 1 connected(): Boolean.

```
1  seen ← {∅};
2  stack ← selectActiveNode();
3  if stack = ∅ then
4      if {node s.t. |dom(actives_node)| = 2} = ∅ then
5          return false ;                                  // conflict detected
6      else
7          return true;
8  while stack ≠ ∅ do
9      current ← pop(stack);
10     if current ∉ seen then
11         seen ← seen ∪ current;
12         stack ← stack ∪ {x s.t. actives_x ≠ 0 ∧ (C_current,x ≠ 0 ∨ C_x,current ≠ 0)};
13 return removeUnseen(seen);
```

Algorithm 2 removeUnseen($seen$: Set of nodes) : Boolean.

```
1  foreach node ∉ seen do
2      if |dom(actives_node)| = 1 ∧ actives_node = 1 then
3          return false;                                   // conflict detected
4      actives_node ← 0;
5  return true;
```

the consistency of the constraints. In fact, for each node that we did not see in the search of Algorithm 1 (therefore not connected to the considered graph), we must set their active value to 0. A conflict is identified when an already decided active node is processed.

After running Algorithm 2, the undirected graph is guaranteed to be connected. Now, we need to check the directed paths from each entry node and identify the unreachable nodes to disable them. We formally introduce Algorithms 4 and 3 to manage the directed graph.

Algorithm 3 checks that there is a path between each entry and at least one exit node. Three sets are used: `seen`, `possible` and `kept`. These sets contain, respectively, the nodes that have been seen, the nodes in a possibly valid path (i.e. which reach an exit) from the considered entry, as well as the nodes which are kept if no conflict is detected before the end of the algorithm. The graph is then traversed from each potential entry using Algorithm 4 (line 6). If the considered entry is valid, it is counted (line 7); otherwise, it is marked as invalid following a consistency test (line 9 to 11). If no valid entry is found (lines 13 and 14), a conflict is raised. We then call Algorithm 2 (line 15) to filter the nodes that have not been kept (and possibly detect a conflict).

Algorithm 4 (called on line 6 of Algorithm 3) is used to test the validity of a node provided as a parameter (`node`). First, the node to be tested is added to the `seen` and `possible` sets (lines 2 and 3). Then, if the node is already in the set of nodes to keep `kept` or if this node is a possible exit, then it is marked as valid and kept (lines 4 to 6). It should be noted that the search is not stopped when a node (or more particularly an entry) is found valid since we want to mark all the nodes reachable from the node which is currently considered. The main

Algorithm 3 checkPaths(G : the graph) : Boolean.

```
1  seen ← {∅};
2  possible ← {∅};
3  kept ← {∅};
4  nbValidEntries ← 0;
5  foreach node s.t. entries_node ≠ 0 ∧ actives_node ≠ 0 do
6      if isValidNode(node, seen, possible, kept) then
7          nbValidEntries ← nbValidEntries + 1
8      else
9          if entries_node = 1 then
10             return false;                          // conflict detected
11         entries_node ← 0
12     possible ← {∅};
13 if nbValidEntries = 0 then
14     return false;                                  // conflict detected
15 return removeUnseen(kept);
```

Algorithm 4 isValidNode($node$: a node, $seen$, $possible$, $kept$: node sets) : Boolean.

```
1  valid ← false;
2  seen ← seen ∪ {node};
3  possible ← possible ∪ {node};
4  if node ∈ kept ∨ exits_node ≠ 0 then
5      valid ← true;
6      kept ← kept ∪ possible;
7  foreach a s.t. actives_node ≠ 0 ∧ C_node,a ≠ 0 do
8      if a ∈ kept then
9          valid ← true;
10         kept ← kept ∪ possible;
11     else if a ∉ seen then
12         if isValidNode(a, seen, possible, kept) then
13             valid ← true;
14 possible ← possible \ {node};
15 return valid;
```

loop (lines 7 to 13) allows considering the children of the node considered (`node`). Indeed, like line 12 of Algorithm 1, we add the neighbours of the current node without considering the undirected case this time. For each of the children `a`, that are not deactivated, two cases are possible:

1. `a` is already in the set of kept nodes (`kept`) (lines 8 to 10). This implies that `node` is valid because it joins a path already validated previously. The path of the possible nodes is therefore kept (line 10).

Algorithm 5 propagate() : Boolean.

```
1 return connected() && checkPaths();
```

2. `a` has never been explored (lines 11 to 13). Algorithm 4 is therefore called recursively to test the validity of `a` (as well as that of `node` if `a` is valid [2]).

Finally, before returning the validity of the original node (`node`), it is removed from the `possible` set (line 14).

The `connected+` propagator (Algorithm 5) first applies the `connected` algorithm (Algorithm 1), and if no conflict occurs, then the paths are checked (Algorithm 3).

5 Experiments

We implemented the algorithms in the *Nacre* [9] solver to evaluate them. The *XCSP3* team kindly provided us with an alternative version of the official parser to manage our new global constraint.

The experiments were performed on a computer with a 6-core processor clocked at 3.70 GHz (Intel i7-8700K) with 32GB of RAM. We used the complete search method (MAC) of the *Nacre* solver without restarts to count the solutions found[3].

The experiments are divided into three parts. The first one presents a real-world use case called *the 1,000 dungeons*. It is a simple scenario that illustrate how our tool can be used in practice, on a simple use case. This "finalised" proof of concept was used to demonstrate the practical interest of our approach to the AAA+ game studio we collaborated with for this study. Indeed, generating a large number of variations from a few source patterns lets us quickly fill the world with many high-quality playgrounds. The second part shows the performances of our approach against the previously used one (the *a posteriori* method used in the AAA+ game studio) on a real-world source dungeon. Our procedure generates only valid variations for an insignificant overhead compared to the one without the global constraint. In the last part, we present a source dungeon generator, based on a structured problem random generator, to evaluate our approach to more dungeons. This generator allows us to conclude on the usefulness and performances of our procedure in different cases.

5.1 The 1,000 Dungeons Experiment

The 1,000 dungeons experiment is a proof of concept developed in the AAA+ video game studio with which we collaborated to show the practical interest of our approach. It corresponds to a finalised tool, build on top of many other internal tools. In this experiment, we generated, from three source patterns, $1,000$ variations. The structural constraints of the model are the ones presented in Section 3.2. The configuration constraints for the computed variations are set up as follows: between 3 and 12 rooms; at most 3 final rooms; one treasure in a final room; and between 3 and 8 fights.

Figure 6 shows a sneak peek of the generated dungeons. We can distinguish the different source patterns used as input. The green and golden rooms are kept in the variation; the golden ones are the final rooms. The crossed swords point out a fight room, and the cup shows a treasure room.

[2] It is not necessary to replicate line 10 after line 13 if `a` and `node` are valid, this has already been done in the call at line 12.

[3] `./nacre_mini_release DUNGEON.xml -complete -sols=X`

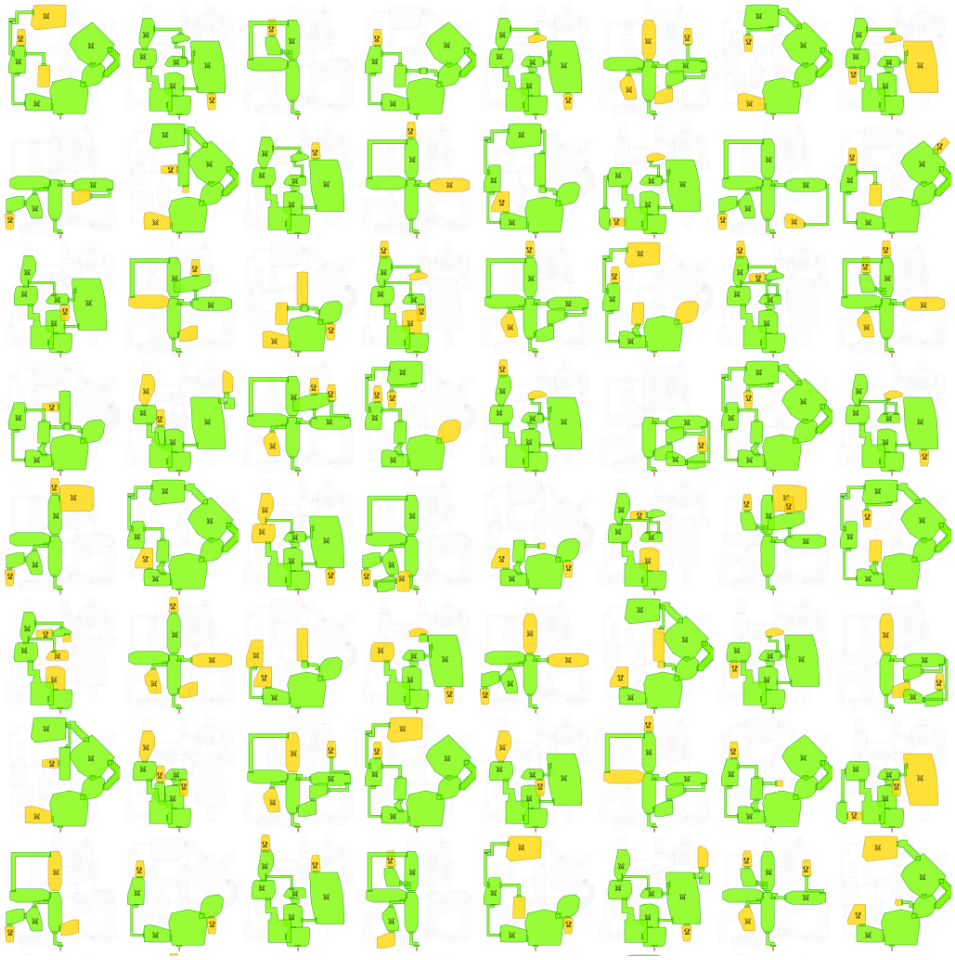

Figure 6 A sneak peek of the 1,000 dungeons experiment using 3 source dungeons that shows the integration of our CP model and propagator in a production tool for a AAA+ video game.

This proof of concept shows that our approach could be useful to quickly fill a world with a lot of different variations or to train bots (e.g., for exploration, for fights). It also illustrates how variants of the same source dungeon could be very different in practice. Most of the considered dungeons Figure 6 will be perceived as different levels by the player. To increase the player's feeling of visiting different levels, for this experiment, we adapted our tool to avoid close variations. We generated much more variations and selected distant ones in the search tree (choosing one solution every fifty solutions, during the backtrack search, allowed us to increase the chances of increasing the diversity, as shown on the random selection of some computed variations, Figure 6). To further expand this work and ensure the generation of a relevant set of variations from one (or more) source dungeon in these kinds of experiment, one could use some metrics (mission linearity, map linearity, leniency and path redundancy [12, 16, 17], for example) to score each variation. The use of additional tags (for instance by specifying the textures of the walls, the positions of furniture and items) are another way of enforcing the novelty feeling by the player. Improving our tool to take this kind of new constraints is easy with CP and will be discussed in the last section.

Table 2 Data from experiments on the real-world instance considered, with 3,019 variables and 11,169 when expressed as a XCSP3 instance. The last column shows how many variations are valid over the number of variations produced by the given method. As discussed in the text, only a small fraction of generated variations are valid when the connected+ propagator is not used.

#Variations	Method	Time (seconds)	#Conflicts	#Valid Variations
100	W/o GC	2.7	15	0
	Connected+	2.78	173	100
1,000	W/o GC	4.33	130	0
	Connected+	5.42	1,462	1,000
10,000	W/o GC	21.72	820	248
	Connected+	32.23	11,781	10,000

5.2 Study of a real-world Instance

To evaluate our approach against the standard *a posteriori* method, we used a real-world source dungeon taken from an actual game level. The source dungeon used for this experiment has 52 nodes, 58 connections, 7 possible entrances, 7 possible exits, and 30 possible final rooms. Even though this problem seems small, it is already difficult and represents real problems well (mainly between 40 and 60 nodes, up to about 100 nodes for larger ones). The *XCSP3* instance associated to this problem has 3,019 variables and 11,169 constraints (1 more for the graph constraint *connected+*).

We evaluate the two approaches on generating 100, 1,000 and 10,000 variations, respectively. The number of variants seems large (a typical shipped game will never contain so many variations) but the goal of our tool is to give the *LD* as many variations as possible for them to pick up the most interesting ones. It is thus expected that our tool will have to handle such large sets of variations. We may have to score each variation (according to the metrics mentioned, for instance, in section 5.1), and to list them for the *LD*.

The first approach (*W/o GC* in Table 2) does not use the new global constraint. The second one (*Connected+* in Table 2) uses the new global constraint and computes exactly the number of variations specified (since they are all valid). The search stops when the specified number of variations is found (but not necessarily valid ones for the *W/o GC* approach).

In Table 2, we can check that, as expected, having the graph checker as a built-in propagator only produces valid variations for the price of small overhead. We conducted further experiments to measure how many variations the method *W/o GC* would have to compute in order to obtain 100 valid variations. We found that 5,390 solutions would have to be computed (in 7.63 seconds with 542 conflicts, to be compared with the 2.78 seconds of our approach, with only 173 conflicts). We also tried to compute 1,000 valid variations with the *W/o GC* method but, after hitting the 2,400 seconds timeout, it generated less than 300 valid variations for this instance. This has to be compared with our approach, that allows to generate 1,000 valid variations in 5.42 seconds.

This clearly shows the interest of our global constraint. It is now possible to produce a tool allowing a real time exploration of many variants, to rank them and change the desired tags almost on the fly. This was clearly not even possible with the previous approach.

5.3 Small-World Random Generated Instances

In order to generalise our findings, we propose a deeper experimental analyse thanks to a random instance generator (`https://bit.ly/3xSXK2B`), available for further works to compare with (our previous problems are unfortunately not publicly available). To kept the random generated graphs interesting and close to real-world problems, designed by humans, our generator is based on the *Watts–Strogatz* model [10] adapted for directed graphs [13]. This model produces graphs with small-world properties; it allows us to simulate real-world instances (e.g., with local clustering) and not random graph with uniform structure only. The model takes the number of nodes of the graph we want to generate N, the mean degree K of the graph and some special probability parameters (detailed below, these are reals between 0 and 1). The associated algorithm is composed of two parts:

1. construct a ring lattice (a graph of N nodes each connected to K neighbours) ;
2. rewire some edges following a probability p.

As mentioned before, in our case, the algorithm is modified to handle directed graphs (Section 3.2 of [13]). The first step remains the same, but the connections are set both ways. The second step uses another probability d to choose if only one way is rewired or both. We also randomly select nodes to be entries and exits. Our generator takes a few parameters:

1. n, the number of nodes of the graph;
2. k, the number of edges by nodes for the ring lattice;
3. a, the access density (percentage of entries and exits);
4. d, the probability for reciprocal edges during rewiring steps;
5. p, the probability of edge rewiring;
6. s, a seed can be specified for reproducibility.

This generator builds the structure of a source dungeon and does not propose to (randomly) generate the configuration constraints. Indeed, choosing and generating a realistic configuration is another work by itself.

We generated 100 instances[4] from 10 nodes to 100 nodes with a step of 10 (10 instances for each step). The generation tool is called with the following default parameters: 20% accesses density (a parameter; 20% of nodes are entries and 20% of nodes are exits); 20% chance for reciprocal edges (d parameter); 20% chance for rewiring (p parameter); and finally, k, the number of edges by nodes for the ring lattice, is 40% of n, the number of nodes of the graph.

The results of the two methods (*W/o GC* and *Connected+*) on the 100 generated instances are reported Table 3. As we have done in the previous experiment, we evaluate the two methods on the generation of 100, 1,000 and 10,000 variations. The data presented in the Table are the computation time (in seconds) and the number of valid variations generated. For each of the metrics, the minimum, maximum, median and mean are shown. This allows us to show the behaviour on the easier (usually fewer nodes in the graph) and harder instances.

This last experiment, based on 300 runs for each method (each run generating many variations, with a total of more than 2 millions variations generated), confirms the conclusions on the previous real-world instance:

- the overhead due to the addition of our global constraint connected+ constraint is very small;

[4] We used a script with the following parameters to generate the instances presented in Table 3: `./genDungeons.sh 10 100 10 10 0`.

Table 3 Data from experiments on the Watts-Stragatz small-world random instances. Each reported number summarises 100 points, 10 points per increasing size of 10 to 100 (see text).

#Var	Method	Time (sec)				#Valid Variations			
		Min	Max	Med.	Mean	Min	Max	Med.	Mean
100	W/o GC	0.13	23.42	6.28	7.57	0	46	24	22.7
	Connected+	0.14	25.94	6.3	7.62	100	100	100	100
1,000	W/o GC	0.27	27.48	7.7	9.07	0	380	173	166.74
	Connected+	0.39	29.15	7.91	9.37	1K	1K	1K	1K
10,000	W/o GC	1.89	57.58	20.39	22.7	0	4798	1,244	1,578.45
	Connected+	2.56	66.45	24.31	26.68	10K	10K	10K	10K

- the variation problem, and the use of classical graph constraint, generates a very large number of non valid solutions;
- the median time obtained by our approach makes it possible to use it in an on-line manner, to help *LD* navigating and sorting the solutions. This is not possible with an approach based on existing constraints.

6 Conclusion & Perspectives

We have presented a new problem of industrial importance for constraint programming, the *Dungeon Variations Problem*, and proposed a first approach to tackle it, based on the introduction of a new global constraint with its propagator. Our solution is already used in pre-production as an internal support tool for *Levels Designers*, supported by the *XCSP3* model and the *Nacre* solver. There is, of course, still rooms for improvements, but we believe our approach has already proven its usefulness and its practical interest for the studio. It is a pragmatic and efficient solution to help *Levels Designers* in their daily work for the gaming industry.

We are, of course, planning to expand this work in several ways. We can use metrics (mission linearity, map linearity, leniency and path redundancy [12, 16, 17], for example) to score each variation to generate a relevant set of variations from a source dungeon. These metrics could be used for the optimisation version (COP) of the *Dungeon Variations Problem*, allowing more control over the dungeons generated (an *LD* often plays on the linearity of the *flow*). Using an approach similar to [2], we could find paths where the dungeon structure is made using some data (or approximations): time or hardness to complete a fight or a puzzle, for example. We could then build levels based on difficulty or time.

We could also enrich the model with different constraints to provide more control and automation for the *Dungeon Variations Problem* (for example, considering the orientation of the room, allowing its rotation). We can also think of models of rooms and connections (for example, a connection can be a hallway, a window, a breakable wall). We can also extend graph constraints to handle distance constraints (e.g., between accesses, from entrances to combat rooms). A final improvement would be to consider qualitative constraint networks (*QCN*) to manage the relative positions of the different rooms and connections, allowing an *LD* to specify the topological constraints of the connections from the dungeon to the outside.

A long-term goal of our work is to generate levels on the fly, based on each user experience and preferences, with strong guarantees. We believe that this work is the first step in this direction.

References

1 Gilles Audemard, Frédéric Boussemart, Christophe Lecoutre, Cédric Piette, and Olivier Roussel. Xcsp^3 and its ecosystem. *Constraints An Int. J.*, 25(1-2):47–69, 2020. doi:10.1007/s10601-019-09307-9.

2 Daniel Le Berre, Pierre Marquis, and Stéphanie Roussel. Planning personalised museum visits. In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6025.

3 Diego de Uña. *Discrete optimization over graph problems*. PhD thesis, University of Melbourne, Parkville, Victoria, Australia, 2018. URL: http://hdl.handle.net/11343/217321.

4 Diego de Uña, Graeme Gange, Peter Schachte, and Peter J. Stuckey. A bounded path propagator on directed graphs. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2016. doi:10.1007/978-3-319-44953-1_13.

5 Grégoire Dooms, Yves Deville, and Pierre Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2005. doi:10.1007/11564751_18.

6 Joris Dormans. A handcrafted feel: Unexplored explores cyclic dungeon generation. URL: https://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/.

7 Jean-Guillaume Fages. *Exploitation de structures de graphe en programmation par contraintes*. Theses, Ecole des Mines de Nantes, 2014. URL: https://tel.archives-ouvertes.fr/tel-01085253.

8 Jean-Guillaume Fages. On the use of graphs within constraint-programming. *Constraints An Int. J.*, 20(4):498–499, 2015. doi:10.1007/s10601-015-9223-9.

9 Gael Glorian, Jean-Marie Lagniez, and Christophe Lecoutre. NACRE - A nogood and clause reasoning engine. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 249–259. EasyChair, 2020. URL: https://easychair.org/publications/paper/rN67.

10 Carsten Grabow, Stefan Grosskinsky, Jürgen Kurths, and Marc Timme. Collective relaxation dynamics of small-world networks. *CoRR*, abs/1507.04624, 2015. arXiv:1507.04624.

11 Ian Horswill and Leif Foged. Fast procedural level population with playability constraints. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'12, page 20–25. AAAI Press, 2012.

12 R. Lavender. The zelda dungeon generator: Adopting generative grammars to create levels for action-adventure games, 2016.

13 Brenton Prettejohn, Matthew Berryman, and Mark McDonnell. Methods for generating complex networks with selected structural properties for simulations: A review and tutorial for neuroscientists. *Frontiers in Computational Neuroscience*, 5:11, 2011. doi:10.3389/fncom.2011.00011.

14 Patrick Prosser and Chris Unsworth. A connectivity constraint using bridges. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 707–708. IOS Press, 2006.

15 Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2006. doi:10.1007/11603023_6.

16 Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1814256.1814260.

17 Thomas Smith, Julian A. Padget, and Andrew Vidler. Graph-based generation of action-adventure dungeon levels using answer set programming. In Steve Dahlskog, Sebastian Deterding, José M. Font, Mitu Khandaker, Carl Magnus Olsson, Sebastian Risi, and Christoph Salge, editors, *Proceedings of the 13th International Conference on the Foundations of Digital Games, FDG 2018, Malmö, Sweden, August 07-10, 2018*, pages 52:1–52:10. ACM, 2018. doi:10.1145/3235765.3235817.

18 Valtchan Valtchanov and Joseph Alexander Brown. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, page 27–35, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2347583.2347587.

19 Breno M. F. Viana and Selan R. dos Santos. A survey of procedural dungeon generation. In *18th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2019, Rio de Janeiro, Brazil, October 28-31, 2019*, pages 29–38. IEEE, 2019. doi:10.1109/SBGames.2019.00015.