# Génération de donjons à l'aide de la programmation par contraintes *

**Gaël Glorian**[1,2] **Adrien Debesson**[2] **Sylvain Yvon-Paliot**[2] **Laurent Simon**[1]

[1] LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France
[2]Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

{prenom.nom}@labri.fr {prenom.nom}@ubisoft.com

**Résumé**

L'industrie du jeu vidéo est l'une des plus importantes industries du secteur du loisir, générant des milliards de dollars de chiffre d'affaire chaque année. Les jeux vidéos se doivent donc d'offrir des expériences de jeu de plus en plus complexes, impliquant des équipes de développeurs et d'artistes de plus en plus importantes. Dans cet article, nous proposons une approche basée sur la programmation par contraintes pour la génération procédurale de donjons dans un contexte de monde/univers ouvert, dans le but d'offrir aux joueurs des mondes ouverts mais offrant une excellente cohérence et une narration de qualité. Grâce à une description globale capturant toutes les salles et situations possibles d'un donjon donné, notre approche propose d'énumérer des variations de ce schéma global, pouvant alors être présentées au joueur pour plus de diversité. Nous formalisons ce problème à l'aide de la programmation par contraintes en exploitant une abstraction sous forme de graphe de la structure du donjon, sur laquelle chaque chemin intéressant représente une variation possible correspondant aux contraintes désirées. Pour ce faire, nous introduisons un nouveau propagateur étendant la contrainte de graphe *connected*, qui permet de considérer des graphes dirigés avec cycles. Nous montrons que, grâce à cette modélisation et le nouveau propagateur proposé, il est possible de prendre en compte des scénarios réalistes pouvant être utilisés dans les jeux AAA. Nous démontrons son efficacité en la comparant à une solution de base consistant à filtrer uniquement les solutions pertinentes a posteriori.

**Abstract**

The video games industry generates billions of dollars in sales every year. However, videos games are more and more complex, involving larger and larger teams of developers and artists to offer gigantic (but consistent) open worlds to players. In this paper, we propose a constraint-based approach to the procedural content generation of dungeons in an open world context. Thanks to a global description capturing all the possible rooms and items of a given dungeon, our approach allows enumerating variations of this global pattern that can be presented to the player for more diversity. We formalize this problem in constraint programming by exploiting a graph abstraction of the dungeon pattern structure on which each interesting path 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 new propagator, it is possible to handle realistic scenarios used in the game industry. We also demonstrate its efficiency by comparing it with a more basic solution consisting of only filtering relevant solution a posteriori. We then conclude and offer several interesting perspectives raised by this approach to the *Dungeon Variations Problem*.

## 1 Introduction

L'industrie du jeu vidéo est l'une des plus importantes industries du loisir, toujours à la pointe des innovations, générant des milliards de dollars de chiffre d'affaire chaque année. Depuis l'apparition des premiers jeux dans les années 1970, le paysage de cette industrie a radicalement changé. Les jeux vidéos sont désormais produits par de grandes équipes d'artistes et de développeurs, offrant des graphismes photo-réalistes et un degré de simulation constamment amélioré.

L'un des plus grands défis de l'industrie du jeu est de pouvoir construire des mondes ouverts, dans lesquels les utilisateurs doivent se sentir libres et où un nombre colossal d'actions leur sont possibles. Générer

*Ce travail a été soutenu par le projet « KIWI » de la région Nouvelle-Aquitaine.

un tel monde sans intervention finale d'un Level Designer (*LD*) pour valider le niveau est toujours un rêve : même si le jeu est très bien conçu, un seul niveau de jeu incohérent peut rapidement briser la réputation d'un jeu, même ayant coûté des millions de dollars. La vérification formelle des niveaux générés peut jouer un rôle essentiel dans l'avenir de l'industrie du jeu. Cependant, s'il est encore trop difficile de vérifier formellement les propriétés des niveaux générés *a posteriori*, nous proposons, dans cet article, de prouver les propriétés des niveaux générés par construction. Intuitivement, dans l'approche proposée, un *LD* produit ce que nous appelons un *donjon source* qui contient un ensemble de salles reliées par des couloirs. Chaque salle a ses propriétés et ses zones identifiées pour des situations possibles (combats, trésors, ...). Le *problème des variantes de donjons* peut être formulé comme le problème de générer une variation appropriée du *donjon source* [1] en désactivant un sous-ensemble de salles et couloirs initiaux de telle sorte que l'ensemble de salles restant corresponde à un ensemble donné de contraintes (par exemple, au moins une entrée, au moins un nombre donné de monstres sur tout chemin).

Si cette approche ne répond pas encore totalement au problème global de la génération procédurale [14], elle ouvre un nouveau champ passionnant pour la PPC. De plus, elle permet aux concepteurs de niveaux de vérifier la cohérence et la qualité des variations de donjons avant de livrer le jeu. Nous spécifions formellement le *problème des variantes de donjons* dans la section 2 et le décrivons comme un problème de programmation par contraintes à l'aide de la contrainte de graphe *connected* présentée dans la section 3. Nous introduisons ensuite, dans la section 4, un nouveau propagateur étendant cette contrainte (*connected+*) qui prend en compte les graphes orientés avec cycles. Dans la partie expérimentale (Section 5), nous montrons que ce propagateur offre une amélioration spectaculaire par rapport à une approche plus naïve de *filtrage*. Avant de conclure, nous énumérons un certain nombre de nouveaux et passionnants travaux supplémentaires rendus possibles par notre approche, offrant de nouveaux défis à la fois pour la programmation par contraintes et l'industrie du jeu.

FIGURE 1 – Un donjon source.

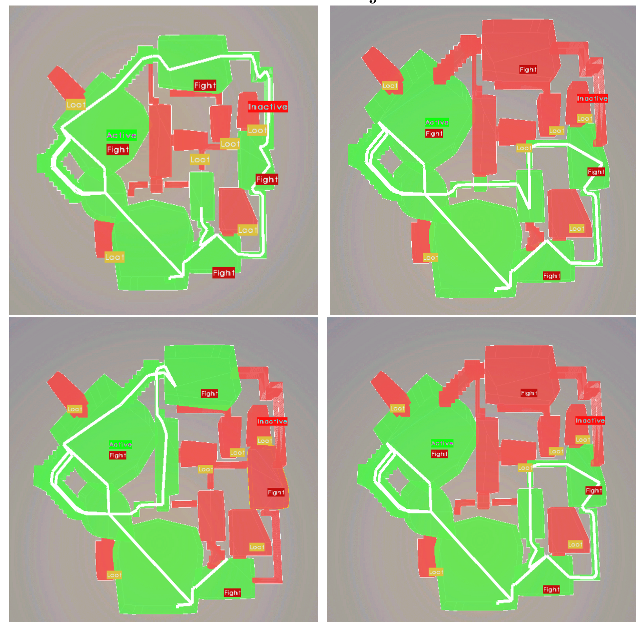

FIGURE 2 – Quelques variations du donjon source (en vert, les salles et couloirs conservés dans la variations).

## 2 Définition du problème

Notre objectif est de proposer un assistant de création de niveaux de jeux pour l'aide à la génération d'ensembles de variations de niveaux. Il s'agit d'une vision restreinte de la problématique de génération procédurale de jeux, dont le but est de garantir des générations automatiques de grande qualité. Dans les jeux que nous considérons, chaque niveau doit être cohérent avec la narration, avec la progression de l'utilisateur et surtout validé par un *LD*. Livrer un jeu AAA ayant un niveau injouable peut en effet avoir des effets dramatiques en termes d'images et de coûts pour le studio de jeux responsable. Pour atteindre la meilleure qualité possible, chaque niveau est donc fabriqué à la

1. La cohérence du donjon source peut aussi être validée en générant une variation prenant en compte toutes les salles ainsi que les couloirs.

main par un designer et sert un objectif précis dans l'histoire globale. Par conséquent, notre approche n'est pas de générer des niveaux à partir de zéro mais de proposer des variations d'un donjon source, conçu par un artiste.

Nous appelons *donjon source* le *design* original (Fig. 1). Ce donjon contient des *salles* avec des propriétés (en pratique, le concept de *salles* peut être trompeur : une salle peut être composée, dans le jeu, par un ensemble de zones qui seront considérées comme un bloc entier). Une salle peut être une *entrée* (une connexion depuis l'extérieur), une *sortie* (une connexion vers l'extérieur). Les salles peuvent également avoir un ensemble d'*étiquettes* (pour gérer les situations de jeu, sa conception, ...). L'ensemble des salles que l'utilisateur pourra explorer et les situations (récompenses, combats, ...) rencontrées s'appelle le *flow*. Il est fréquent d'ajouter des salles *terminales* (*i.e.* des impasses) aux donjons pour permettre des quêtes secondaires ou des explorations facultatives. Ces dernières, qui sont des salles *sans issues* (autre que le couloir qui y mène), contiennent généralement des trésors, des clés, des objets spéciaux ou des monstres, mais ne bloquent pas le joueur. Il faut noter que les *connections* entre les pièces sont dirigées (une porte peut être un accès à sens unique, le joueur peut sauter d'une pièce à une autre, ...).

Le but du *problème des variantes de donjons* est de générer un sous-ensemble cohérent de salles à partir du donjon source satisfaisant certaines contraintes (Fig. 2). Le premier ensemble de contraintes est structurel : chaque ensemble de pièces doit être jouable (toutes les pièces sont connectées, il y a au moins une entrée et une sortie), et les pièces terminales doivent être étiquetées ainsi (pour que le *LD* vérifie leur intérêt). Les contraintes du deuxième ensemble sont des contraintes de *configuration* qui sont facultatives et peuvent être définies par l'utilisateur : certaines salles spéciales peuvent être forcées d'être *active*, *entrée*, *sortie* ou *terminale* dans la solution. Notre outil devra proposer, dans un délai raisonnable (quelques secondes), des variantes intéressantes au *LD*.

La génération procédurale de niveaux et de *flow* n'est pas nouvelle [5,12,13]. Cependant, comme indiqué dans ces références, les systèmes ne sont pas encore mûrs pour être autonomes. De plus, aucun de ces travaux n'a le potentiel d'atteindre le niveau de qualité attendu dans notre contexte. De plus, à notre connaissance, aucun travail précédent ne traite à la fois de la génération des donjons et de l'histoire elle-même (qui est une pierre angulaire de notre travail, assurée par la définition du donjon source). Comme décrit dans la section suivante, nous proposons de traiter le problème comme un problème de graphe. La propriété de connexité sera garantie par une nouvelle extension de la contrainte de graphe *connected* [4,9] adaptée à notre problème.

# 3 Modélisation du *problème des variantes de donjons* en PPC

Dans cette section, nous formulons le problème en problème de satisfaction de contraintes. Les contraintes sont présentées en langage naturel dans un premier temps puis sous forme clausale et ensembliste (Tableau 3.1).

## 3.1 Variables du modèle

Soit $n$ le nombre de nœuds du donjon source (numéroté de 0 à $n-1$). Les connexions (e.g. *couloirs*) entre les nœuds sont identifiés par les numéros de nœuds ($C_{ij}$ est une connexion du nœud $i$ au nœud $j$). Nous avons besoin de six tableaux de taille $n$ pour modéliser le problème : Quatre tableaux de *booléens* `entries`, `exits`, `actives` et `finals` qui indiquent, respectivement, si un nœud est une entrée, une sortie, s'il est actif et s'il est terminal. Deux tableaux `sumToNode` et `sumFromNode` qui comptent respectivement le nombre de connexions vers chaque nœud et depuis chaque nœud. Pour encoder les connexions entre les salles, nous utilisons une matrice d'adjacence $C$ de taille $n \times n$ qui indique les arêtes actives (connexions) du graphe (les nœuds peuvent avoir plusieurs successeurs). Les tableaux `sumToNode` et `sumFromNode` sont définis à l'aide de contraintes de somme. Nous évitons également, à ce niveau, de multiples couloirs entre les mêmes salles. Une telle variation, si nécessaire, peut être gérée à un autre niveau. Pour éviter les boucles triviales, notre modèle doit également bloquer la diagonale de la matrice d'adjacence $C$.

Formellement, le problème peut s'exprimer sous forme de graphe de manière assez simple : soit $G(V,E)$ un graphe où $V \subset \mathbb{N}$ représente les salles et $E \subset \{(i,j) \mid i,j \in V \wedge i \neq j\}$ les couloirs. Le but du *problème des variantes de donjons* est de générer un graphe $G'(V',E')$ avec $V' \subset V$ et $E' \subset E$. Il est à remarquer que `actives` $\equiv V'$. Les ensembles `entries`, `exists` et `finals` sont des sous ensembles de $V'$.

## 3.2 Contraintes du modèle

Les contraintes du modèle assurent la cohérence de chaque variation, donnée en langage naturel dans la liste suivante. Ces contraintes sont exprimées comme contraintes d'intention dans le modèle XCSP3 [1] :

1. Si un nœud est une entrée, il doit être actif
2. Si un nœud est une sortie, il doit être actif
3. Si un nœud est terminal, il doit être actif
4. Si un nœud est une entrée, il ne peut pas être terminal

| | Forme Clausale | Forme ensembliste |
|---|---|---|
| 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 | $finals_i = 0 \vee sumFromNode_i = 1$ | Couvert par 8. |
| 10 | $finals_i = 0 \vee C_{ij} = 0 \vee C_{ji} = 1$ | Couvert par 8. |
| 11 | $finals_i = 0 \vee C_{ij} = 1 \vee C_{ji} = 0$ | Couvert par 8. |
| 12 | $C_{ij} = 0 \vee C_{ji} = 0 \vee sumToNode_i \neq 1$ $\vee sumFromNode_i \neq 1 \vee finals_i = 1$ | $\{(i, j), (j, i)\} \subset E' \wedge \ \forall k \ \lvert\{(i, k)\} \in E'\rvert = 1$ $\wedge \ \forall k \ \lvert\{(k, i)\} \in E'\rvert = 1 \Rightarrow i \in finals$ |

Table 1 – Expression des contraintes sous forme clausale et ensembliste.

5. Si un nœud est une sortie, alors il ne peut pas être terminal
6. Si une connexion est utilisée, alors les nœuds associés doivent être actifs
7. Si un nœud est actif, au moins une connexion doit aller vers ou partir de ce nœud
8. Si un nœud est terminal, alors une et une seule connexion doit aller vers ce nœud
9. Si un nœud est terminal, alors une et une seule connexion doit partir de ce nœud
10. Si un nœud $i$ est terminal et qu'une connexion va de $i$ à $j$ alors une connexion doit aller de $j$ à $i$
11. Si un nœud $i$ est terminal et qu'une connexion va de $j$ à $i$ alors une connexion doit aller de $i$ à $j$
12. S'il y a un aller-retour entre deux nœuds $i$ et $j$ et il n'y a qu'une seule connexion allant et partant de $i$, alors le nœud $i$ doit être terminal

L'ensemble de contraintes ci-dessus permet de spécifier la structure des variations souhaitées (le problème des solutions non connectées sera abordé dans la section suivante). On notera que la gestion des salles *terminales* (appelées *finals* dans le tableau), est couverte par les lignes 8 à 11. Il s'agit de s'assurer que le joueur pourra revenir dans le chemin principal depuis une impasse correspondant éventuellement à une quête secondaire. Certaines autres contraintes, appelées contraintes de *configuration*, sont également autorisées dans l'outil que nous proposons : bornes sur le nombre d'entrées, de sorties, de salles actives et terminales. Il faut également que notre modèle permette à l'utilisateur de forcer des pièces spécifiques à être active (ou non) dans toutes les variantes générées. L'utilisateur peut également désactiver un sous-ensemble de couloirs pour explorer les variations correspondantes. De même, il est possible de donner le contrôle sur les situations (et par conséquent sur le *flow*) en bornant le nombre de salles ayant des étiquettes données. Par exemple, si nous avons 8 salles marquées comme salle de combat dans un donjon source, un *LD* peut vouloir générer une variation avec seulement 3 d'entre elles. Pour gérer ces limites facilement, nous pouvons créer une contrainte de somme sur la propriété *actives* des salles étiquetées. Nous ne décrivons pas plus en détail cette partie du modèle, puisqu'elle est couverte par une approche classique par contraintes.

## 4 Évolution de la contrainte *connected*

Comme mentionné précédemment, nous devons nous assurer que nos variantes soient connectées pour répondre à toutes les contraintes. Nous avons deux choix pour cela. (1) vérifier chaque solution avec un algorithme *ad-hoc* (c'est une approche classique dans les jeux vidéos : les niveaux générés peuvent être testés par la suite par des algorithmes, des bots et/ou des humains) ou (2) forcer toute variation proposée à être correcte par construction, en filtrant les solutions partielles dès que possible lors de la recherche de variations. Il faut garder à l'esprit ici que le but de notre approche est d'aider le *LD* à naviguer dans les alternatives possibles le plus aisément possible. Arriver à garantir que chaque solution proposée aient certaines propriétés par construction est donc un élément déterminant.

La plupart des travaux sur les contraintes et les graphes (par exemple [3, 6, 10]) considèrent en général un nœud source et un nœud objectif, au travers du problème de chemin/circuit. On ne peut pas malheureusement pas s'appuyer directement sur ces travaux. Ils sont soit non applicables, soit trop contraints. De plus, nous devons considérer le fait que les graphes

**Algorithme 1 :** connected() : Boolean

```
1  seen ← {∅};
2  stack ← selectActiveNode();
3  if stack = ∅ then
4      if {node | 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 | actives_x ≠ 0 ∧ (C_current,x ≠ 0 ∨ C_x,current ≠ 0)} ;
13 return filterNodes(seen);
```

sont dirigés, et chaque variation peut avoir beaucoup de nœuds sources (entrées) ainsi que plusieurs nœuds objectifs (sorties) [2]. Il est également important de noter, à ce stade, que nous n'essayons pas encore d'optimiser les chemins, de quelque manière que ce soit. Nous prévoyons de le faire dans le cadre de travaux ultérieurs.

L'approche que nous proposons est un propagateur à deux niveaux qui repose d'abord sur une implémentation simple de la contrainte `connected` [4] sur la version non dirigée du graphe (Alg. 1). Ensuite, un autre niveau de filtrage est ajouté (Alg. 3 et 4) pour garantir la validité des chemins.

La contrainte `connected` garantit que nous n'avons pas de donjons disjoints, mais ne considère pas les graphes dirigés. Nous n'avons donc aucune garantie que le donjon soit jouable (*i.e.* tous les nœuds sont accessibles depuis les entrées, en particulier les nœuds de sortie). Pour surmonter ce problème, nous introduisons un algorithme pour rechercher des chemins à partir de chaque entrée [3]. Pour cette raison, nous avons besoin de connaître les valeurs des tableaux `entries` et `exits` dans le propagateur.

L'algorithme 1 part d'un nœud actif (ligne 2, la fonction selectActiveNode() sélectionne un nœud actif dans le graphe, *i.e.* un nœud $x$ tel que $actives_x = 1$, ou retourne un ensemble vide si aucun nœud actif n'est trouvé). Ensuite, il suit chaque connexion en considérant la version non dirigée du graphe (ligne 12). De la ligne 3 à 7, si nous ne trouvons aucun nœud actif,

2. Un nœud peut bien sûr être une entrée et une sortie en même temps. En pratique, dans les problèmes du monde réel, cela arrive souvent.

3. Cet algorithme peut être adapté pour gérer la taille des chemins, qu'ils soient contraints ou pour obtenir des informations (longueur moyenne, etc.).

**Algorithme 2 :** filterNodes(*seen* : Set of nodes) : Boolean

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

**Algorithme 3 :** handleNode(*node* : a node, *seen* : a set, *possible* : a set, *kept* : a set) : 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 |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 handleNode(a, seen, possible, kept) then
13             valid ← true;
14 possible ← possible \ {node};
15 return valid;
```

nous devons vérifier la cohérence du graphe. En effet, si certains des nœuds ne sont pas décidés (leur taille de domaine booléen est de 2), la contrainte est cohérente puisque nous ne pouvons rien filtrer. Sinon, on détecte une incohérence si tous les nœuds sont marqués inactifs (ligne 4-5). Jusqu'à ce que la pile soit vide, nous marquons le nœud courant (ligne 11) et ajoutons ses voisins à la pile (ligne 12). Lorsque la pile est vide, l'algorithme 2 est appelé pour filtrer les nœuds et vérifier la cohérence des contraintes. En effet, pour chaque nœud que nous n'avons pas vu dans la recherche de l'algorithme 1 (donc non connecté au graphe considéré), nous devons mettre leur valeur active à 0. Un conflit est identifié lorsque un nœud déjà décidé actif est traité.

Après l'exécution de l'algorithme 2, le graphe non orienté est assuré d'être connecté. Nous devons maintenant vérifier les chemins dirigés depuis chaque nœud d'entrée et identifier les nœuds non accessibles pour les désactiver. Nous introduisons formellement les algorithmes 3 et 4 pour gérer le graphe dirigé.

**Algorithme 4 :** checkPaths($G$ : the graph) : Boolean

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

L'algorithme 4 vérifie qu'il existe un chemin entre chaque entrée et au moins une sortie. Pour cela, trois ensembles sont utilisés : `seen`, `possible` et `kept`. Respectivement, ces ensembles contiennent tous les nœuds vus, les nœuds dans un chemin possiblement valide (*i.e.* qui atteint une sortie) depuis l'entrée considérée, ainsi que les nœuds qui sont conservés si aucun conflit n'est détecté avant la fin de l'algorithme. Le graphe est ensuite parcouru à partir de chaque entrée potentielle grâce à l'algorithme 3 (ligne 6). Si l'entrée considérée est valide, elle est comptabilisées (ligne 7) ; sinon, elle est marquée comme invalide suite à un test de cohérence (ligne 9 à 11). Si aucune entrée valide n'est trouvée (lignes 13 et 14), un conflit est soulevé. Nous appelons ensuite l'algorithme 2 (ligne 15) pour filtrer les nœuds qui n'ont pas été conservés (et éventuellement détecter un conflit).

L'algorithme 3 (appelée à la ligne 6 de l'algorithme 4) permet de tester la validité d'un nœud fourni en paramètre (`node`). En premier lieu, le nœud à tester est ajouté aux ensembles `seen` et `possible` (lignes 2 et 3). Ensuite, si le nœud est déjà dans l'ensemble des nœuds à conserver `kept` ou que ce nœud est une sortie possible, alors il est marqué comme valide et conservé (lignes 4 à 6). Il est à noter que la recherche n'est pas arrêtée lorsqu'un noeud (ou plus particulièrement une entrée) est trouvée valide puisque nous voulons marquer tous les nœuds atteignables à partir du nœud actuellement considéré. La boucle principale (lignes 7

| # variations | | W/o GC | Connected+ |
|---|---|---|---|
| | Time | 2,7 | 2,78 |
| 100 | # conflicts | 15 | 173 |
| | % Valid | 0 | 100 |
| | Time | 4,33 | 5,42 |
| 1 000 | # conflicts | 130 | 1 462 |
| | % Valid | 0 | 1 000 |
| | Time | 21,72 | 32,23 |
| 10 000 | # conflicts | 820 | 11 781 |
| | % Valid | 248 | 10 000 |

Table 2 – Données des expériences sur l'instance réelle considérée.

à 13) permet de considérer les fils du nœud considéré (`node`). En effet, d'une manière semblable à la ligne 12 de l'algorithme 1, nous ajoutons les voisins du nœud courant, sans considérer le cas non orienté cette fois-ci. Pour chacun des fils `a` deux cas sont possibles : (1) `a` est déjà dans l'ensemble des nœuds conservés (`kept`) (lignes 8 à 10). Cela implique que `node` est valide car il rejoint un chemin déjà validé précédemment. Le chemin des nœuds possibles est par conséquent conservé (ligne 10). (2) `a` n'a jamais été exploré (lignes 11 à 13). L'algorithme 3 est donc appelé récursivement pour connaître la validité de `a` (ainsi que celle de `node` si `a` est valide [4]). Enfin, avant de retourner la validité du nœud original (`node`), celui-ci est retiré de l'ensemble `possible` (ligne 14).

Le propagateur `connected+` applique d'abord l'algorithme `connected` (Alg. 1), et si aucun conflit ne survient, alors les chemins sont vérifiés (Alg. 4).

## 5 Expérimentations

Nous avons implémenté les algorithmes dans le solveur *Nacre* [7]. L'équipe *XCSP3* nous a fourni une version alternative du parseur officiel pour gérer notre nouvelle contrainte globale [5]. Les expériences ont été exécutées sur un ordinateur avec un processeur 6 cœurs cadencé à 3,70 GHz (Intel i7-8700K) avec 32Go de RAM. Nous avons utilisé la méthode de recherche complète (MAC) du solveur *Nacre* sans redémarrage afin de compter les solutions trouvées [6].

Le donjon source (tiré d'un niveau de jeu réel) utilisé pour l'expérience a 52 nœuds, 58 connexions, 7 entrées possibles, 7 sorties possibles et 30 salles finales possibles. Même si ce problème semble petit, il est déjà difficile

4. Il n'est pas nécessaire de répliquer la ligne 10 après la ligne 13 si `a` et `node` sont valides, cela a déjà été fait dans l'appel à la ligne 12

5. Merci à Gilles Audemard et Christophe Lecoutre !

6. `./nacre_mini_release pattern.xml -complete -sols=X`

et il représente bien les problèmes réels (en général entre 40 et 60 nœuds, jusqu'à environ 100 nœuds pour les plus grands). L'instance *XCSP3* associée a 3019 variables et 11169 contraintes (+1 pour la contrainte de graphe). Nous évaluons les deux approches sur la génération de 100, 1000 et 10000 variations, respectivement. La première approche (*W/o GC* dans le tableau 5) n'utilise pas la nouvelle contrainte globale. La seconde approche (*Connected +* dans le tableau 5) utilise la nouvelle contrainte globale et calcule exactement le nombre de variations spécifié (puisqu'elles sont toutes valides). La recherche s'arrête lorsque le nombre spécifié de variations est trouvé (mais pas nécessairement des variantes valides pour l'approche *W/o GC*).

Dans le tableau 5, nous pouvons voir que, comme prévu, avoir le vérificateur de graphes comme propagateur intégré permet de ne produire que des variations valides pour un coût très faible. Nous avons mené d'autres expériences pour mesurer le nombre de variations que la méthode *W/o GC* devrait calculer pour obtenir 100 variations valides. Nous avons constaté que 5390 solutions devraient être calculées (en 7,63 de secondes pour 542 conflits, à comparer avec les 2,78 secondes de notre approche, pour seulement 173 conflits). Nous avons également essayé de calculer 1000 variations valides avec la méthode *W/o GC* mais, après avoir atteint le délai de 2 400 secondes, moins de 300 variations valides ont été générée pour cette instance. Cela doit être comparé à notre approche, qui permet de générer 1000 variations valides en 5,42 secondes.

Nous travaillons à l'enrichissement de la comparaison expérimentale par l'ajout de problèmes générés aléatoirement mais ayant une structure. Cela fait partie de la poursuite du travail présenté ici.

## 6 Conclusion & Perspectives

Nous avons présenté un nouveau problème d'importance industrielle pour la programmation par contraintes, le *problème des variantes de donjons*, et proposé une première approche pour y faire face. Notre solution est déjà utilisée en pré-production comme un outil interne d'assistance aux *Levels Designers*, supporté par le modèle *XCSP3* et le solveur *Nacre*. Il y a encore des possibilités d'améliorations (par exemple une meilleure structure de données au sein de l'algorithme de filtrage), mais nous pensons que notre approche a déjà démontré son utilité. C'est une solution pragmatique et efficace pour aider les *Levels Designers* dans leur travail quotidien pour l'industrie du jeu.

Nous prévoyons, bien entendu, d'étendre ce travail de plusieurs manières. Nous pouvons utiliser des métriques (mission linearity, map linearity, leniency and path redundancy [8, 11, 12] par exemple) pour noter chaque variation afin de générer un ensemble pertinent de variations de donjon à partir d'un modèle. Ces métriques pourraient être utilisées pour la version d'optimisation (COP) du *problème des variantes de donjons* permettant plus de contrôle sur les donjons générés (un *LD* joue souvent sur la linéarité du *flow*). En utilisant une approche similaire à [2], nous pourrions trouver des chemins où la structure du donjon est faite en utilisant certaines données (ou approximations) : le temps ou la difficulté pour terminer un combat ou un puzzle, par exemple. Nous pourrions alors construire des niveaux en fonction de la difficulté ou du temps.

Nous pourrions également enrichir le modèle avec différentes contraintes pour fournir plus de contrôle et d'automatisation pour le *problème des variantes de donjons* (par exemple, en considérant l'orientation de la salle, en permettant sa rotation). Nous pouvons également penser à des modèles de salles et de connexions (par exemple, une connexion peut être un couloir, une fenêtre, un mur cassable). Nous pouvons également étendre les contraintes de graphes pour gérer les contraintes de distance (par exemple entre les accès, des entrées aux salles de combat). Une dernière amélioration serait de considérer des réseaux de contraintes qualitatives (*QCN*) pour gérer les positions relatives des différentes salles et connexions, permettant au *LD* de spécifier les contraintes topologiques des connexions du donjon vers l'extérieur.

Un objectif à long terme de notre travail est de générer des niveaux à la volée, basés sur l'expérience utilisateur, avec de fortes garanties. Nous pensons que ce travail est le premier pas dans cette direction.

## Références


[1] Gilles AUDEMARD, Frédéric BOUSSEMART, Christophe LECOUTRE, Cédric PIETTE et Olivier ROUSSEL : Xcsp$^3$ and its ecosystem. *Constraints An Int. J.*, 25(1-2):47–69, 2020.

[2] Daniel Le BERRE, Pierre MARQUIS et Stéphanie ROUSSEL : Planning personalised museum visits. *In* Daniel BORRAJO, Subbarao KAMBHAMPATI, Angelo ODDI et Simone FRATINI, éditeurs : *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013.

[3] Diego de UÑA, Graeme GANGE, Peter SCHACHTE et Peter J. STUCKEY : A bounded path propagator on directed graphs. *In* Michel RUEHER, éditeur : *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 de *Lecture Notes in Computer Science*, pages 189–206. Springer, 2016.

[4] Grégoire DOOMS, Yves DEVILLE et Pierre DUPONT : Cp(graph) : Introducing a graph computation domain in constraint programming. *In* Peter van BEEK, éditeur : *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 de *Lecture Notes in Computer Science*, pages 211–225. Springer, 2005.

[5] Joris DORMANS : A handcrafted feel : Unexplored explores cyclic dungeon generation. https ://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/.

[6] Jean-Guillaume FAGES : On the use of graphs within constraint-programming. *Constraints An Int. J.*, 20(4):498–499, 2015.

[7] Gael GLORIAN, Jean-Marie LAGNIEZ et Christophe LECOUTRE : NACRE - A nogood and clause reasoning engine. *In* Elvira ALBERT et Laura KOVÁCS, éditeurs : *LPAR 2020 : 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 de *EPiC Series in Computing*, pages 249–259. EasyChair, 2020.

[8] R. LAVENDER : The zelda dungeon generator : Adopting generative grammars to create levels for action-adventure games, 2016.

[9] Patrick PROSSER et Chris UNSWORTH : A connectivity constraint using bridges. *In* Gerhard BREWKA, Silvia CORADESCHI, Anna PERINI et Paolo TRAVERSO, éditeurs : *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 de *Frontiers in Artificial Intelligence and Applications*, pages 707–708. IOS Press, 2006.

[10] Luis QUESADA, Peter Van ROY, Yves DEVILLE et Raphaël COLLET : Using dominators for solving constrained path problems. *In* Pascal Van HENTENRYCK, éditeur : *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 de *Lecture Notes in Computer Science*, pages 73–87. Springer, 2006.

[11] Gillian SMITH et 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.

[12] Thomas SMITH, Julian A. PADGET et 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 et Christoph SALGE, éditeurs : *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.

[13] Valtchan VALTCHANOV et 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.

[14] Breno M. F. VIANA et 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.