
Génération aléatoire de donjons

Projet de spécialité

Encadré par Julien Moncel

Arthur
Huillet

Aloïs
Jobard

Philippe
Mattei

Loïc
Ripert

Juin 2009

Table des matières

1	Introduction	3
1.1	Problématique	3
2	Organisation du projet	3
3	Génération géométrique et topologique	4
3.1	Grammaire de pièces	4
3.2	Portes et connexité	6
4	Évaluateurs et opérations sur le graphe	8
4.1	Les évaluateurs de taille	8
4.2	Difficulté et placement des monstres	8
4.2.1	Évaluateurs de Adams	8
4.2.2	Évaluateur de difficulté final	9
4.2.3	Placement des ennemis	9
4.3	Choisir des sorties	9
4.3.1	Première approche	9
4.3.2	Deuxième approche - bulles dynamiques	10
4.4	Zones inutiles et cadeaux	11
4.4.1	Première passe : parcours de l'arbre couvrant	12
4.4.2	Seconde passe : détection des faux positifs	12
4.4.3	Troisième passe : calculs de profondeur	14
4.4.4	Placement des cadeaux	14
5	Conclusion - suite du projet	14

1 Introduction

Dans un jeu de rôle sur ordinateur, on trouve souvent des bâtiments ou grottes dans lesquels le joueur doit s'aventurer, afin par exemple d'éliminer tous les ennemis ou de ramener un objet. Ces niveaux sont couramment appelés donjons.

La génération aléatoire de donjons est une fonctionnalité courante dans les jeux vidéo. Cela permet de fournir au joueur une infinité de niveaux dans lesquels évoluer. On obtient ainsi des aventures variées et différentes à chaque utilisation.

Notre projet consiste à écrire, évaluer et améliorer un générateur aléatoire de donjons, destiné à être utilisé par le jeu libre FreedroidRPG. Nous générerons un plan de bâtiment.

Nos lectures nous ont menés sur les documents de David Adams¹ et nous avons décidé de réutiliser son travail qui consiste à utiliser une représentation sous forme de graphe pour effectuer divers traitements d'amélioration et d'évaluation sur un donjon.

1.1 Problématique

Adams présente un moyen de générer la topologie d'un donjon, basé sur des grammaires de graphes, mais explique lui-même que cette solution n'est pas idéale car transformer le graphe en un ensemble de pièces n'est pas une chose aisée. Nous avons décidé d'expérimenter un algorithme basé sur une grammaire de pièces, qui découpe récursivement un bâtiment. Cet algorithme construit en plus un graphe de représentation topologique du donjon. De cette manière nous pouvons appliquer les différents algorithmes basés sur les graphes, proposés par Adams ou conçus par nous-mêmes.

Le projet s'est donc divisé en plusieurs étapes distinctes :

1. écrire un algorithme de génération de donjon basé sur une grammaire de pièces
2. comprendre, écrire et interpréter les évaluateurs de donjons proposés par Adams
3. implémenter les algorithmes d'amélioration du donjon, en se basant sur les idées d'Adams

2 Organisation du projet

Nous avons utilisé le langage C et un dépôt Subversion (SVN) pour le code source, et un wiki pour héberger la documentation que nous avons écrit au fur et à mesure de l'avancement du projet. Notre travail est disponible à l'adresse <http://code.google.com/p/dungeonfd>. Ce rapport est en partie basé sur la documentation présente sur notre wiki.

Nous avons réparti les tâches de telle manière que le travail se fasse en binôme ou individuellement. Nous communiquons entre nous chaque jour (physiquement), de manière à ce que chacun soit au courant de l'avancement des autres, et que les problèmes "difficiles" qui

¹David Adams "Automatic Generation of Dungeons for Computer Games" - <http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2002/pdf/u9da.pdf>

demandent réflexion soient étudiés par tout le groupe. Cela a bien fonctionné : nous avons eu, pratiquement en permanence, deux à trois tâches en parallèle, avec une communication efficace dans l'équipe.

La planification du projet globale a été effectuée au démarrage du projet, car nous avions une idée relativement précise des tâches à accomplir. Leur durée exacte n'a pas été prévue car trop variable au vu du temps total disponible. Chaque semaine, et lors des réunions avec notre encadrant Julien Moncel, nous faisons le point sur le travail effectué (objectifs accomplis, travail restant), et préparions une liste de tâches à effectuer la semaine suivante. Les tâches étaient réparties par affinité et compétence. Cette méthode de planification a donné des résultats très satisfaisants car nous n'avons jamais pris de retard ni trop d'avance par rapport à nos attentes.

3 Génération géométrique et topologique

3.1 Grammaire de pièces

Nous avons commencé par chercher une grammaire de pièces. La première idée a été de créer une grammaire de pièces garantissant la connexité. Cela induit que la grammaire devait gérer les portes. Les règles de cette grammaire ont été cherchées de manière empirique. Nous avons finalement cherché l'ensemble des terminaux de la grammaire.

```
P
-----
|       |
|       |
|       |
-----
```

(P représente la ou les portes dans la salle)

P1 :	P2 :	P3 :	P4 :
-----	-----	---P---	-----
	P		P
---P---	-----	-----	-----

P5 :	P6 :	P7 :	P8 :
-----	---P---	-----	---P---
P		P	P
---P---	---P---	---P---	-----

P9 :	P10 :	P11 :	P12 :
-----	---P---	---P---	-----
P P	P	P	P P
-----	-----	---P---	---P---

P13 :	P14 :	P15 :
---P---	---P---	---P---
P	P P	P P
---P---	-----	---P---

Dérivation:

Grammaire parenthésée car l'ordre de dérivation est important ((PvP)hP) est différent de (P h (P v P))

Par convention lors d'une dérivation on nomme de haut en bas et de gauche à droite

(P v (P h P))	((P v P) h P)	(P h (P v P))
-----	-----	-----
-----	-----	-----
-----	-----	-----
-----	-----	-----

P -> (P1 h P3) | (P4 v P2)

P1 -> (P1 h P6) | (P1 v P1) | (P7 v P2) | (P4 v P5) | (P7 v P5) | p1 (terminal)

P2 -> (P2 h P2) | (P5 h P3) | (P1 h P8) | (P5 h P8) | (P9 v P2) | p2

P3 -> (P6 h P1) | (P3 v P3) | (P10 v P2) | (P4 v P8) | (P10 v P8) | p3

P4 -> (P4 h P4) | (P7 h P3) | (P1 h P10) | (P7 h P10) | (P4 v P9) | p4

etc.

Cette approche a été abandonnée, à cause de sa trop grande complexité. Nous avons alors choisi d'implémenter une grammaire de pièces plus simple avec 2 règles sans gérer les portes.

Les règles de dérivation sont :

$S \rightarrow P$

$P \rightarrow P \text{ h } P \mid P \text{ v } P$ (découpe horizontale ou découpe verticale)

$P \rightarrow p$ (terminal)

Les découpes se font de manière aléatoire, à une position aléatoire respectant certaines conditions (la coupure n'est pas toujours au milieu de la pièce). Les règles de découpe sont soumises à deux contraintes : une surface minimale par pièce, et un rapport de longueurs borné. Cela permet d'obtenir des pièces de forme "probable" (pas de pièce trente fois plus longue que large) et de dimensions acceptables (pas de pièces minuscules du fait de la contrainte de surface). Tout ceci permet de générer un plan de bâtiment sans zone vide, réaliste.

3.2 Portes et connexité

La grammaire de pièces, comme indiqué précédemment, ne crée pas de portes : les pièces ne sont pas reliées entre elles. Il faut donc exécuter un algorithme de création de portes. La contrainte mathématique fondamentale à respecter est la connexité : toute pièce doit pouvoir être atteinte à partir de n'importe quelle autre. Nous utilisons une technique très simple que nous avons appelée "bulldozer", qui ne garantit pas en elle même la connexité. Une deuxième étape de l'algorithme reliera les composantes connexes entre elles.

On place un "bulldozer" dans une pièce du donjon. Il va se déplacer dans une direction aléatoire en cassant un mur, ce qui crée une connexion avec une autre pièce. Le bulldozer s'arrête et disparaît lorsqu'il tente de créer une connexion qui existe déjà, c'est-à-dire par exemple lorsqu'il tente de revenir sur ses pas. Pour chacune des pièces, on note si elle a été visitée par un bulldozer ou pas : tant que toutes les pièces n'ont pas été visitées par un bulldozer, on en lance un dans la première pièce non visitée. Ceci ne nous garantit pas la connexité, car deux bulldozers différents peuvent avoir créé deux composantes connexes séparées. Par contre, empiriquement, nous avons constaté que nous avions au maximum six ou sept composantes connexes dans nos donjons d'environ 130 pièces.

La deuxième étape de l'algorithme calcule les composantes connexes du graphe, puis parcourt chaque pièce. Elle en regarde les voisines. Si une des voisines appartient à une composante connexe différente, alors l'algorithme va créer une connexion entre les deux pièces avec une certaine probabilité.

Au final, on obtient un plan de bâtiment connexe.

Nous avons en revanche rapidement remarqué une certaine monotonie dans le plan généré : toutes les pièces sont rectangulaires. De plus les découpes sont visibles très facilement sur le plan. Nous avons alors implémenté une alternative aux portes, que nous appelons la fusion. Il s'agit, en connectant deux pièces, non pas de créer une porte, mais de supprimer complètement le mur qui sépare les deux pièces, créant d'une certaine façon une nouvelle pièce avec une forme souvent intéressante visuellement. On peut voir sur l'exemple des pièces qui ont fusionné entre elles. Ceci nous permet de générer une grande variété de

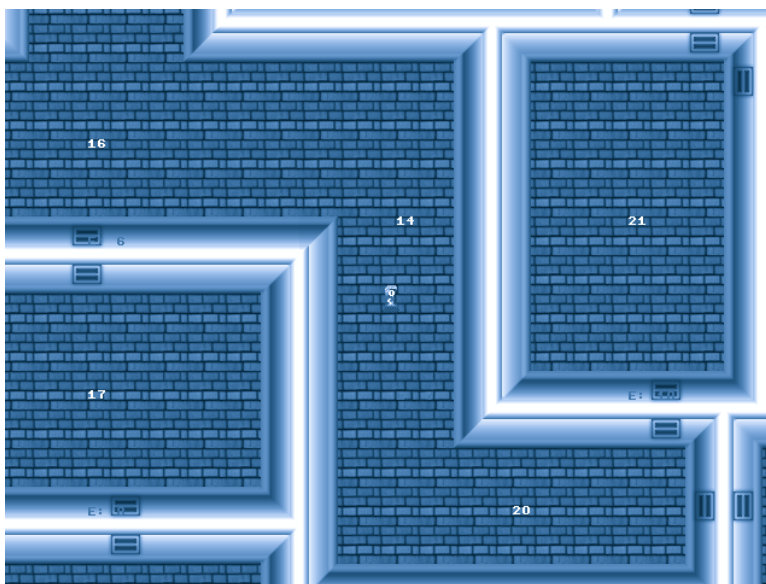


FIG. 1 – Fusion des salles 14, 16 et 20

formes de pièces, ce qui améliore grandement la qualité visuelle de nos donjons.

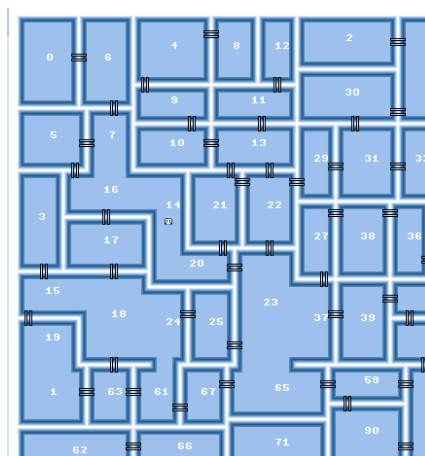


FIG. 2 – Plan fusionné du donjon

4 Évaluateurs et opérations sur le graphe

4.1 Les évaluateurs de taille

David Adams propose plusieurs manières d'évaluer la taille d'un donjon. Ces évaluations rendent compte de la taille réelle du donjon, du nombre de pièces ainsi que du nombre de connexions entre les pièces. Chaque évaluateur propose une mesure de l'une ou l'autre de ces propriétés.

- L'évaluateur 1 est très simple : on compte le nombre de salles du donjon. Cet évaluateur ne permet pas en lui-même d'apporter un réel jugement sur la qualité d'un donjon.
- L'évaluateur 2 mesure la taille totale du donjon, en estimant la longueur de chaque pièce. Couplé au premier évaluateur, on peut obtenir la taille moyenne des pièces, ce qui est un premier critère d'évaluation. Des salles trop petites ou trop grandes dégradent la qualité du jeu.
- Le troisième évaluateur mesure le nombre de choix par pièce, c'est intrinsèquement un indicateur de la difficulté du donjon. Un nombre de choix faible (proche de 1) rend le donjon très linéaire. Par contre, un nombre de choix trop élevé induit un trop grand nombre de connexions et on trouvera un chemin quasiment direct (en ligne droite) entre l'entrée et la sortie. De plus s'il y a trop de choix par pièces, le joueur peut avoir la sensation d'être perdu.
- Le dernier évaluateur proposé par Adams calcule la distance moyenne parcourue par un joueur dans le donjon. Nous avons transformé cette surface moyenne en ratio par rapport à la surface totale. L'interprétation en devient d'autant plus facile. Cet évaluateur nous donne une idée de la réelle "longueur" du donjon.

4.2 Difficulté et placement des monstres

Adams dans sa thèse définit trois évaluateurs de difficulté pour les donjons. Nous les avons implémentés dans un premier temps tels qu'ils étaient définis, puis nous les avons adaptés à nos besoins.

4.2.1 Évaluateurs de Adams

Le premier évaluateur de difficulté proposé fait simplement la somme des difficultés des pièces du donjon. Cet évaluateur donne une information très difficilement utilisable puisqu'un grand donjon donnera toujours une grande valeur de difficulté, ce qui ne reflète pas la réalité.

Le second évaluateur marche comme le premier, à ceci près qu'il inclut un facteur correspondant au nombre de portes dans chacune des pièces. L'utilité de cet évaluateur est elle aussi limitée en pratique.

Le troisième évaluateur proposé est de loin le plus intéressant. Il calcule la moyenne de la difficulté de chaque pièce du donjon et de la difficulté du plus court chemin entre l'entrée et la sortie. C'est ce dernier évaluateur qui nous a inspirés pour écrire notre propre évaluateur, qui est celui que nous avons utilisé pour nos calculs.

4.2.2 Évaluateur de difficulté final

À partir des différentes idées introduites par les évaluateurs de Adams, nous en avons conçu un qui réponde mieux à nos besoins. Cet évaluateur calcule une difficulté moyenne par pièce, de la manière suivante :

- On calcule la difficulté moyenne des pièces se trouvant sur les plus courts chemins de l'entrée aux sorties.
- On calcule la difficulté moyenne de toutes les pièces du donjon en utilisant le premier évaluateur défini par Adams.
- Enfin, on fait une moyenne entre les deux valeurs obtenues précédemment.

Cette méthode nous permet d'obtenir une valeur moyenne de difficulté par pièce avec un poids plus fort pour les noeuds se trouvant sur le plus court chemin.

4.2.3 Placement des ennemis

Le placement des ennemis dans le donjon se fait en utilisant notre évaluateur de difficulté. L'utilisateur demande via notre interface de programmation une valeur moyenne de difficulté par pièce.

Pour répartir la difficulté entre les différentes pièces de manière aléatoire, on commence par placer dans chaque salle une difficulté égale à une fraction de la difficulté demandée, ceci pour favoriser la vitesse de convergence de l'algorithme.

Ensuite, et tant que notre évaluateur de difficulté donne une valeur inférieure à celle demandée par l'utilisateur, nous augmentons la difficulté dans des pièces choisies aléatoirement.

Cette méthode nous donne une bonne répartition des ennemis avec une difficulté variable d'une pièce à l'autre.

4.3 Choisir des sorties

L'entrée de notre donjon étant sélectionnée au hasard parmi toutes les salles du donjon, il nous fallait concevoir un algorithme² capable de placer les sorties de manière intelligente, c'est à dire respectant les quelques propriétés suivantes :

- Une sortie est toujours située raisonnablement loin de l'entrée.
- S'il y a plusieurs sorties, celles-ci sont suffisamment éloignées les unes des autres.

4.3.1 Première approche

Notre première approche fût de placer des sorties les plus éloignées possibles de l'entrée, et à chaque candidat, on vérifiait que l'on se trouvait assez loin d'une sortie déjà sélectionnée. Cette distance entre deux sorties était évaluée grâce à l'évaluateur de taille 4 décrit par Adams. Cependant, nous avons pu remarquer certains points négatifs vis-à-vis de cet algorithme :

²Voir <http://code.google.com/p/dungeonfd/wiki/HowToFindExits> pour voir les algorithmes détaillés

- Les sorties se retrouvaient souvent aux bords du donjon, limitant ainsi la variabilité de nos donjons.
- Suivant la valeur de l'évaluateur de taille 3, nous avons remarqué que le critère choisi pour la distance entre les sorties n'était pas pertinent.
- Il arrivait que notre algorithme ne donne pas le nombre désiré de sorties, car la distance minimale entre les sorties n'était pas dynamique.

Au vu de ces différentes contraintes, nous avons abandonné cet algorithme, et opté pour un algorithme introduisant une plus grande variabilité dans le placement des sorties ainsi qu'une dimension dynamique pour éviter le cas d'erreur de sorties non trouvées. En effet, lorsque l'utilisateur demande deux sorties, notre générateur doit retourner exactement deux sorties.

4.3.2 Deuxième approche - bulles dynamiques

L'idée de cet algorithme est simple : lorsque l'on place une entrée ou une sortie, on crée autour d'elle une "bulle". Toutes les salles comprises à l'intérieur de la bulle seront alors enlevées des salles dites "candidates". Puis, tant que l'on a pas trouvé le nombre demandé de sorties, on en sélectionne une au hasard parmi les candidates et on crée une nouvelle bulle autour de celle-ci.

Si l'on arrive à ne plus avoir de candidates alors que l'on doit encore générer des sorties, on réduit la taille des bulles afin d'adapter notre algorithme et augmenter nos chances de trouver des sorties. On recommence alors la procédure de recherche des sorties avec cette taille de bulle plus faible.

Algorithme 4.1 ³

```
bubble_size = 0.8;
bubble_reduce = 0.5;

while(i < nbexits){
    reset_candidates();
    candidate[start] = 0;

    CreateBubble(start, candidate, bubble_size);

    for(i = 0; i < nbexits){
        if(nomoreexits()) break;
        exit = PickRandomExit(candidate);
        CreateBubble(exit, candidate, bubble_size);
    }

    bubble_size = bubble_size * bubble_reduce;
}
```

Cet algorithme nous garantit les éléments suivants :

1. les sorties sont à la fois “assez loin” de l’entrée et des autres sorties, et ce de manière homogène.
2. les sorties étant choisies aléatoirement, on a une plus grande variabilité que l’algorithme précédent.
3. vu que l’on réduit à chaque itération la taille des bulles, le cas bloquant serait que l’on demande un nombre de sorties supérieur au nombre de salles dans le donjon. Dans tous les autres cas, on assure de retourner le nombre adéquat de résultats.

4.4 Zones inutiles et cadeaux

Les zones inutiles dans un donjon sont définies par Adams comme suit : ce sont les ensembles de noeuds connectés qui ne se trouvent pas sur un chemin de l’entrée à la sortie. Ces zones représentent donc des culs-de-sac qui obligent le joueur à tourner en rond ou à revenir sur ses pas. Détecter ces zones est important car les grandes zones inutiles peuvent être extrêmement frustrantes pour le joueur.

Nous avons donc pensé et implémenté un algorithme de détection des zones inutiles afin ensuite de pouvoir appliquer un traitement. Les plus petites seront laissées telles quelles, les plus grandes seront par exemple remplies avec des cadeaux, des monstres spéciaux, etc.

³Voir en annexe (5, page 15) pour un exemple pas à pas de placement de sorties

Notre algorithme fonctionne en trois passes : la première utilise un arbre couvrant de notre graphe pour trouver toutes les zones inutiles plus un certain nombre de faux positifs qui seront filtrés dans la deuxième passe. La troisième passe calcule la profondeur de chaque noeud inutile dans la zone inutile à laquelle il est associé. Au final, l'algorithme renvoie les différentes zones inutiles, chacune associée à son noeud d'entrée, sa profondeur et son noeud le plus profond.

4.4.1 Première passe : parcours de l'arbre couvrant

Pour détecter les culs-de-sac et l'unique chemin de l'entrée à la sortie dans l'arbre couvrant, on parcourt l'arbre en profondeur avec une fonction récursive. L'idée est de remonter une information pour chaque noeud et de les trier en deux catégories : utiles (c'est à dire sur un chemin entre l'entrée et la sortie) ou potentiellement inutiles.

Algorithme 4.2

```
G = (V,E) dungeon graph
T = (V, E') one spanning tree of G
X /
n = entry node of V

n.is_useful = explore(n);

function explore(node n)

ret = maybe_useless

if n is a EXIT
    ret = useful

if n is a leaf
    if n is not an EXIT
        return maybe_useless

for each n' neighbor of n in T
    if explore(n') == useful
        ret = useful

return ret
```

4.4.2 Seconde passe : détection des faux positifs

Comme on l'a dit plus haut, la première passe laisse un certain nombre de faux positifs (c'est-à-dire. des noeuds utiles marqués comme potentiellement inutiles). Pour régler la question nous effectuons un second parcours.

Tout d'abord, nous allons associer à chaque noeud potentiellement inutile un numéro de noeud d'entrée dans la zone inutile, qui correspond au premier noeud utile qui soit connecté avec notre zone inutile. Pour chaque zone inutile, le noeud d'entrée existe et est unique.

Algorithme 4.3

```

for each node N *in the spanning tree* marked as useful
    N.entrypoint = N
    for each node N' neighbor of N
        if N' is maybe_useless
            propagate_entry(N, N);

procedure propagate_entry(root, entrypoint)

    root.entrypoint = entrypoint

    for each node N neighbor of root *in the spanning tree*
        if N is maybe_useless
            propagate_entry(N, entrypoint)

```

Ensuite, nous itérons la propriété suivante jusqu'à ce qu'on ait atteint un point fixe : propriété : si un noeud potentiellement inutile a, dans le graphe de départ, un voisin qui ne porte pas le même point d'entrée que lui, alors ce noeud est utile. Dans ce cas, on marque le noeud comme utile et on propage le nouveau point d'entrée aux voisins inutiles de ce noeud

Algorithme 4.4

```

while (changed)
    for each maybe_useless node N *in the graph*
        usefulcount = 0

        for each node N' neighbor of N
            if N'.entrypoint != N.entrypoint
                N = useful

                for each node M neighbor of N *in the spanning tree*
                    propagate_entrypoint(M, N)

                changed = true
            elif N' is useful
                usefulcount ++

        if usefulcount >= 2
            N = useful
            changed = true

```

4.4.3 Troisième passe : calculs de profondeur

Pour chaque zone inutile on doit avoir des informations, notamment sur sa taille et son noeud le plus profond. Ces informations serviront à effectuer le traitement d'amélioration du donjon. On calcule donc lors d'une troisième passe la profondeur de chaque noeud inutile dans sa zone. Ceci se fait par un calcul de profondeur récursif sur chaque voisin inutile de chaque noeud du graphe.

4.4.4 Placement des cadeaux

Maintenant que l'on connaît la profondeur de chaque zone inutile et son noeud le plus profond, nous pouvons effectuer un traitement pour rendre un certain intérêt à cette zone du point de vue du joueur. Nous avons décidé de placer un cadeau (qui peut être un monstre spécial, de l'équipement, ou une récompense quelconque) au niveau des noeuds les plus profonds de chaque zone inutile de profondeur supérieure ou égale à 4. Ceci permet de récompenser le joueur pour avoir exploré cette zone. La profondeur de 4 noeud est bien sûr sujette à discussion puisqu'elle dépend entièrement de la taille des pièces associées aux noeuds dans le jeu.

5 Conclusion - suite du projet

Nous sommes très satisfaits des résultats que nous avons obtenus. Nous avons en effet implémenté un système de génération de donjon qui propose un plan de bâtiment compact (sans espace mort entre les pièces), qui effectue différentes mesures dessus, permet le placement de contenu (ennemis, cadeaux et sorties), qui améliore les éléments peu satisfaisants (les grandes zones inutiles), et qui peut être interfacé avec un jeu vidéo de manière relativement simple. Du point de vue de la performance, notre système génère un donjon en moins d'une seconde, ce qui satisfait pleinement nos attentes.

Annexe

Exemples de placement de sorties

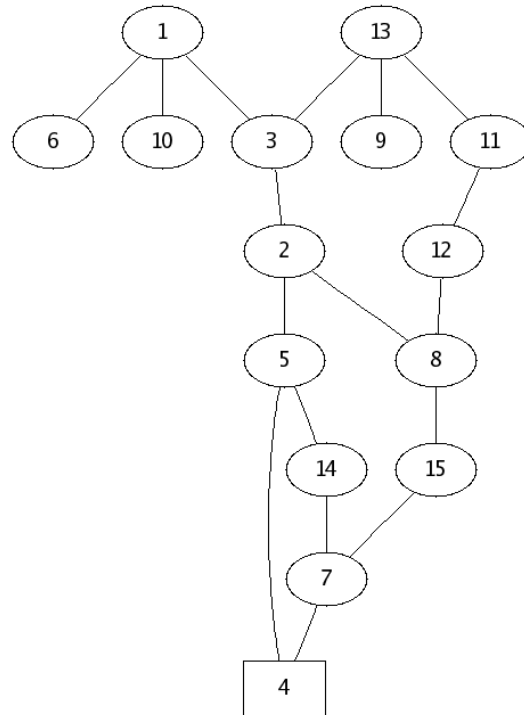


FIG. 3 – Le noeud 4 est sélectionné comme l'entrée du donjon

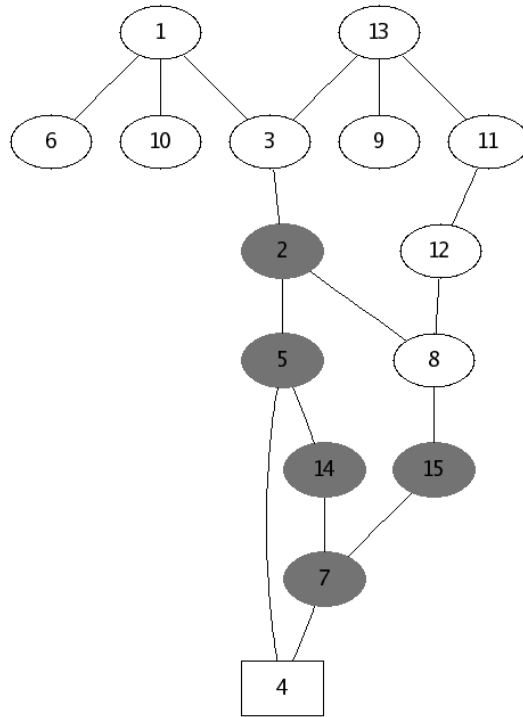


FIG. 4 – On crée un “bulle” autour du noeud 4.

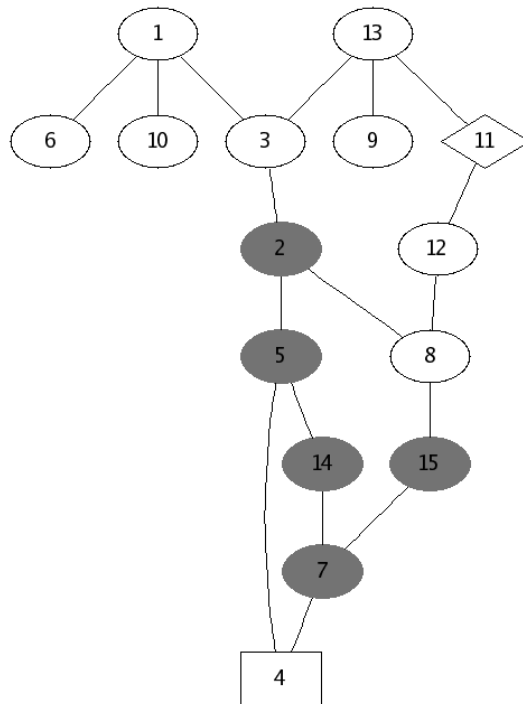


FIG. 5 – Le noeud 11 est alors notre première sortie

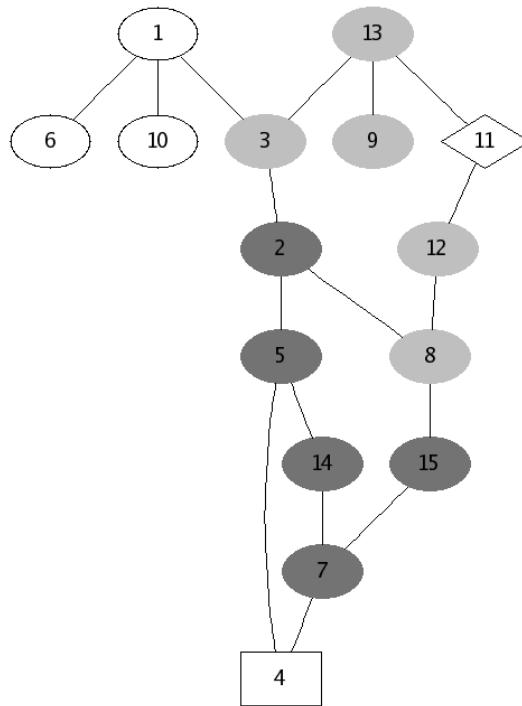


FIG. 6 – On crée une deuxième bulle autour du noeud 11

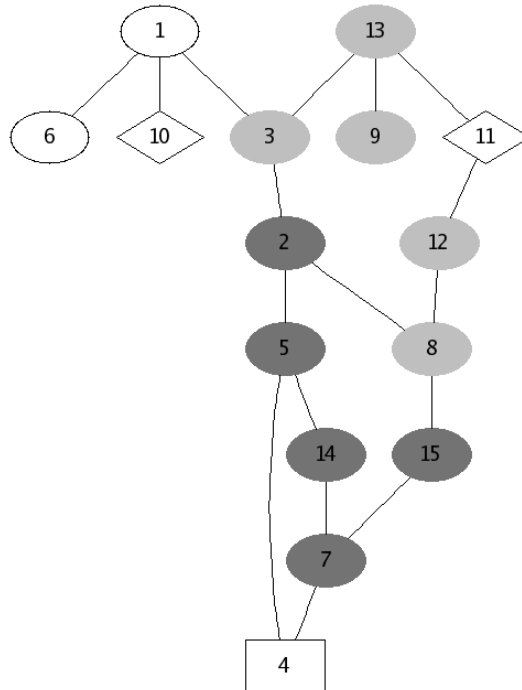


FIG. 7 – Le noeud 10 est notre deuxième sortie