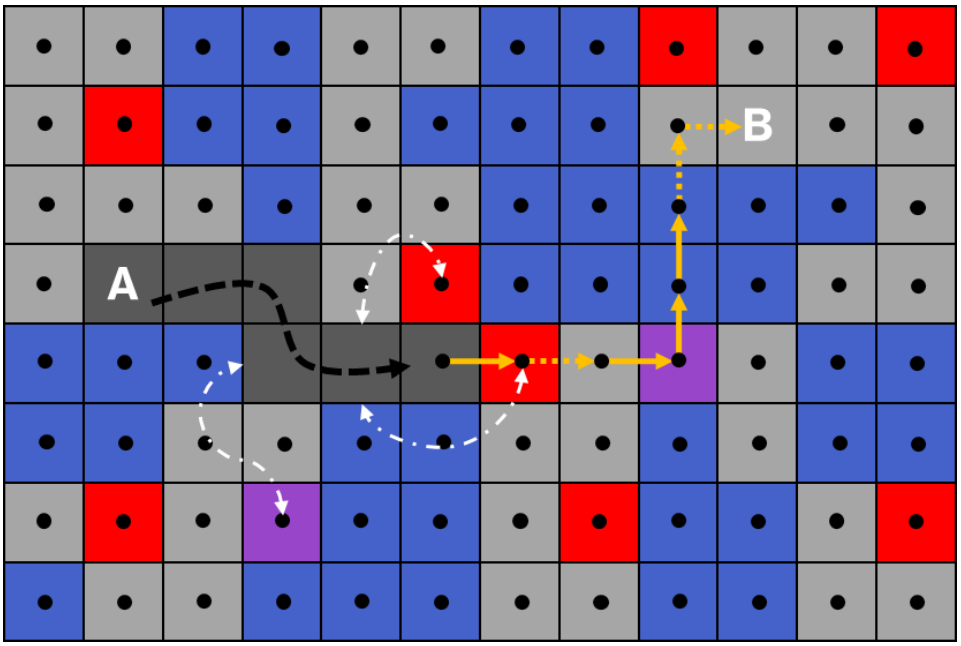
UQAC

Université du Québec
à Chicoutimi

PLATFORMER 2D & GÉNÉRATEUR DE NIVEAU



Nom : Léonard Lemoosy

Durée : 18/09/23 à 17/12/23

Module : 8INF206

Table des matières

Description du projet

Le but du projet est de réaliser un platformer 2D sur Unity et de créer un algorithme capable de générer des niveaux de toute difficulté.

Description du jeu

Le jeu s'appelle Mulpa, nous sommes un personnage dans un donjon et nous devons atteindre la sortie sans mourir.

Il y a en tout une dizaine de niveaux, il faut terminer tous les niveaux pour finir le jeu.

Dans un niveau, il y a des blocs, des pièges, des pièces, des leviers, des portes et une sortie.

Si le joueur touche un piège, il meurt (il recommence le niveau).

Exemple d'un niveau généré par le générateur de niveau.

Description du générateur de niveau

Pour la création des 10 niveaux, nous allons utiliser les Algorithmes Génétiques, nous allons faire des croisements de niveau et évaluer chaque individu par une IA qui sera déjà entraînée sur plusieurs niveaux.

Conception d'un niveau

Pour la génération des niveaux, on va se concentrer sur comment générer un niveau avec une difficulté spécifique.

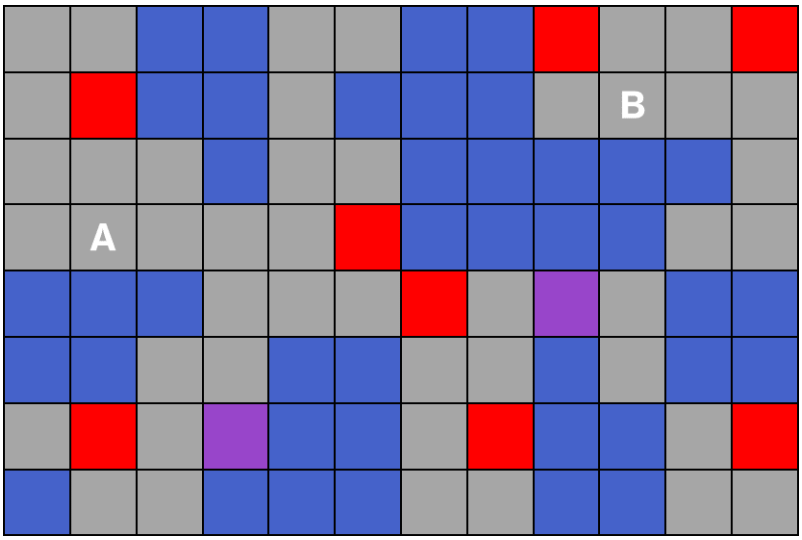
Génération

La génération d'un niveau est assez simple :

1. Création d'une matrice vide M x N (pour le projet, 24 x 14).
2. L'algorithme place un départ et une sortie aléatoire dans la matrice, avec une distance minimum (pour le projet, c'est l'utilisateur qui place le départ et la sortie).
3. L'algorithme place des blocs aléatoires dans la matrice.
4. L'algorithme place des piques aléatoires dans la matrice.
5. L'algorithme place des monstres aléatoires dans la matrice.
6. Exportation de la matrice vers le jeu.

Pour éviter d'avoir des incohérences dans le niveau (comme des monstres dans le vide), on va ajouter des conditions :

1. Création d'une matrice vide M x N (pour le projet, 24 x 14).
2. L'algorithme place un départ et une sortie aléatoire dans la matrice, avec une distance minimum (pour le projet, c'est l'utilisateur qui place le départ et la sortie).
3. L'algorithme place des blocs 2x2 aléatoires dans la matrice (pour éviter d'avoir des couloirs étroits).
4. L'algorithme place des blocs 1x1 aléatoires dans la matrice (collés à des blocs 2x2, 1x1 ou bordures).
5. L'algorithme place des piques aléatoires dans la matrice (collés à des blocs 2x2, 1x1 ou bordures).
6. L'algorithme place des monstres aléatoires dans la matrice (au-dessus des blocs 2x2 ou 1x1).
7. Exportation de la matrice vers le jeu.



Exemple d'un niveau généré aléatoirement.

Description des cases :

Case	**Objet**
Gris	Vide
Bleu	Bloc
Rouge	Pique
Violet	Monstre

Valeurs que j'utilise pour le projet :

Objet	**Nombre**
Bloc 2x2	30
Bloc 1x1	30
Pique	20
Monstre	5

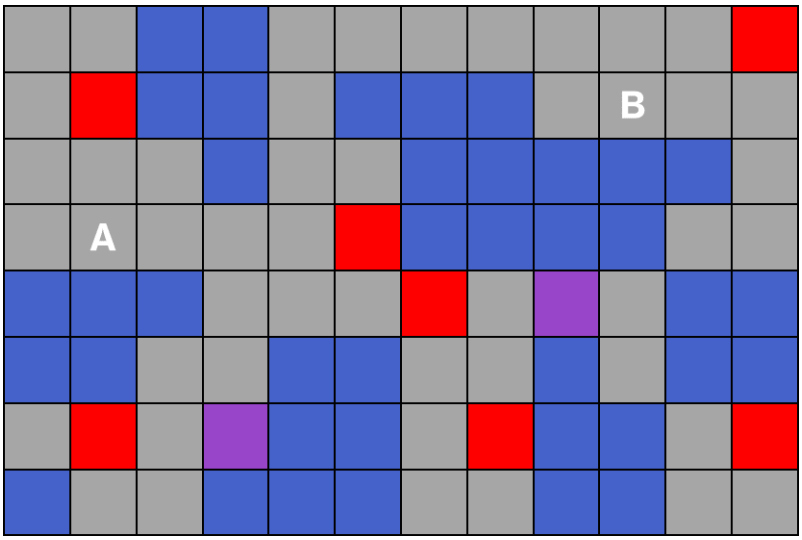
Validation

Problème : Le niveau est-il réalisable ?

Problème 2 : Le niveau est-il facile ? Moyen ? Ou Difficile ?

On va d'abord se focaliser sur : si le niveau est réalisable ou pas.

Pour vérifier si le niveau est réalisable, on pourrait vérifier s'il existe un Plus Court Chemin (PCC) entre A et B avec l'algorithme de Dijkstra.



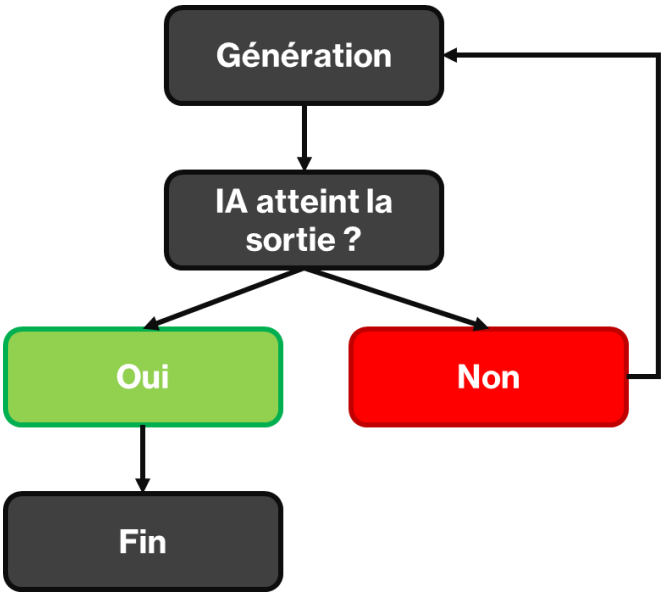
Exemple d'un niveau généré aléatoirement avec un PCC entre A et B.

Problème : Le PCC entre A et B inaccessible pour le joueur.

Une autre méthode qui permettrait de vérifier si le niveau est réalisable : une IA vérifie à notre place.

On part du principe que l'IA est entraînée à l'avance et est capable de faire n'importe quel niveau qu'on lui propose (s'il est réalisable).

Si l'IA réussit le niveau, alors le niveau est réalisable, sinon on recommence la génération.



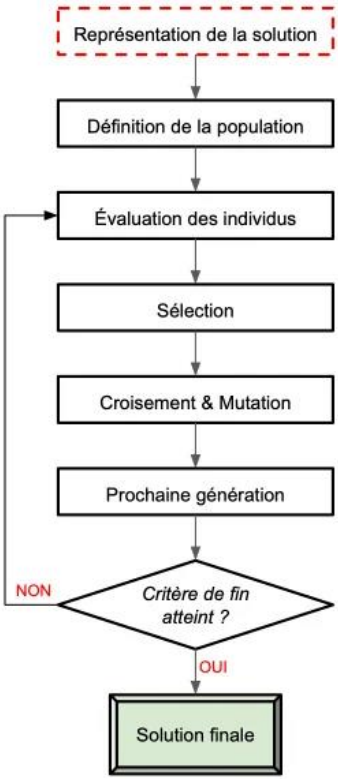
Problème : On peut très bien générer une infinité de niveaux sans que l'IA réussisse (par exemple un mur qui sépare le niveau en 2 à chaque génération).

Évaluation

Le problème, c'est que la fonction qui permet d'évaluer le niveau est binaire, soit l'IA réussie, soit l'IA échoue.

Pour avoir une solution qui converge et éviter d'avoir une infinité de niveaux irréalisable, il faudrait avoir une fonction linéaire, une fonction qui soit capable d'évaluer le niveau malgré qu'il soit impossible.

On pourrait utiliser cette fonction pour les algorithmes génétiques.



Supposons qu'il existe une fonction qui permet de calculer le taux de faisabilité d'un niveau :

1. On génère P niveaux aléatoires.
2. On évalue le taux de faisabilité de chaque niveau.
3. Si un niveau a un taux de faisabilité de 100%, on a trouvé notre niveau.
4. Sinon, on sélectionne les S meilleurs niveaux.
5. On supprime les P – S pires niveaux.
6. On crée E enfants (mélange de 2 niveaux aléatoires).
7. On complète la population (on régénère des niveaux aléatoires).
8. On réévalue toute la population et puis on recommence…

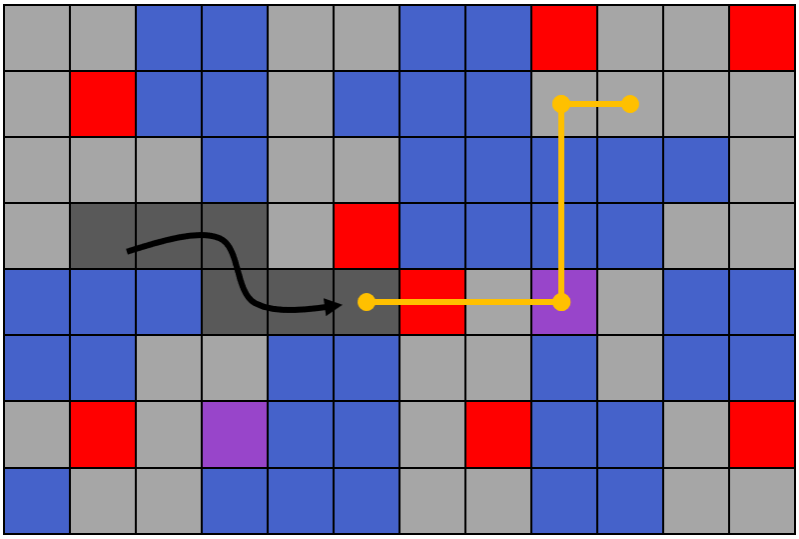
Pour le projet :

Variable	Valeur
P	200
S	50
E	100

Pour la partie crossover (mélange de 2 niveaux aléatoires), l'enfant hérite de la moitié des cases aléatoire du père et la moitié des cases aléatoire de la mère.

Attention à bien conserver le nombre de blocs 2x2, blocs 1x1, piques et monstres ! Sinon, vous risquez d'avoir des niveaux avec aucun pique ou aucun monstre.

Maintenant la question qu'on se pose, quelle est la fonction qui permet d'évaluer le taux de faisabilité du niveau ?



$$X = 3 \times SP - 0.1 \times nCases - 1 \times nSpades - 1 \times nMonsters$$

Supposons qu'on possède une IA capable de terminer n'importe quel niveau réalisable, qu'il est capable d'aller d'un point A à un point B dans n'importe quel niveau.

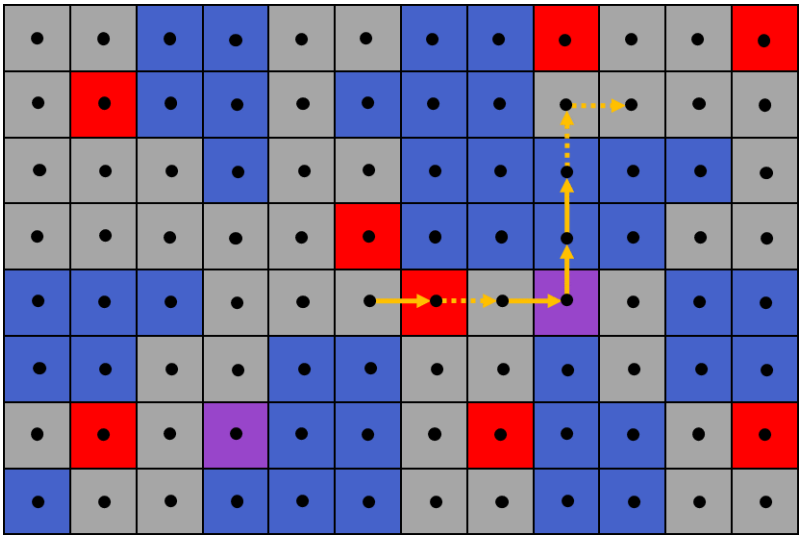
Si je lance l'IA sur un niveau irréalisable, il va essayer de se rapprocher du plus possible de la sortie en théorie.

On peut voir sur le schéma au-dessus, que l'IA est morte par un pique (ou a pris trop de temps à jouer) à la position (5, 3).

Lorsque l'IA est morte, nous calculons directement sa fonction X.

Cette fonction permet d'évaluer le taux de faisabilité d'un niveau, et permet aussi d'évaluer la difficulté d'un niveau, explication de chaque variable :

- **SP** correspond à la distance du PCC entre la mort de l'IA et la sortie en prenant compte des blocs, des piques et des monstres.

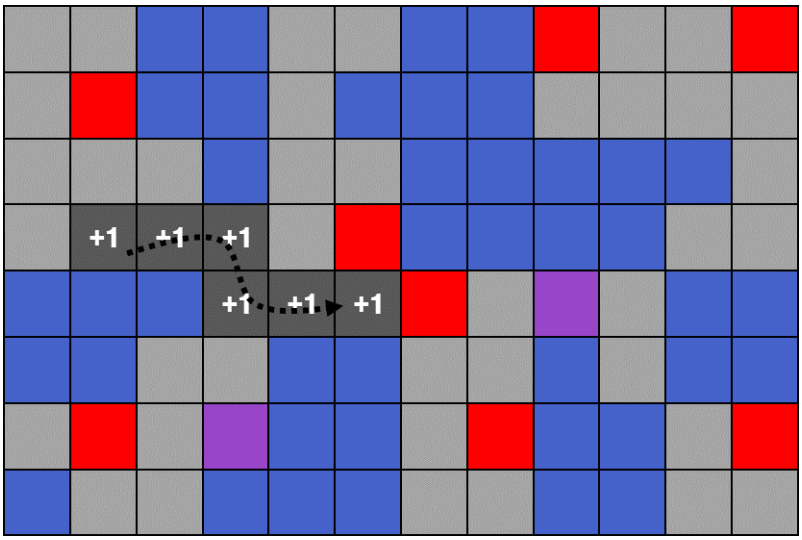


SP = 10 + 1 + 10 + 10 + 10 + 1 + 1

SP → +∞ : L'IA sera loin de la sortie

SP → 0 : L'IA sera proche de la sortie

- **nCases** correspond au nombre de cases parcourues par l'IA.

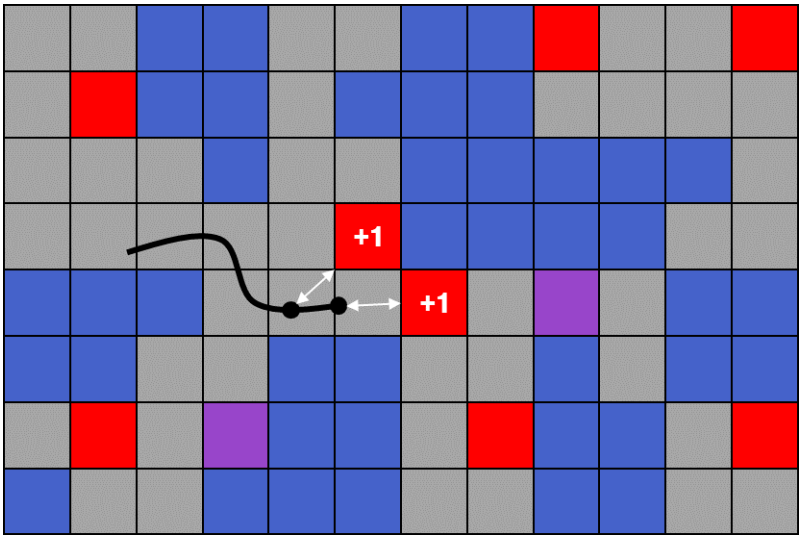


nCases = 1 + 1 + 1 + 1 + 1 + 1

nCases → +∞ : L'IA parcourra beaucoup de cases

nCases → 0 : L'IA parcourra peu de cases

- **nSpades** correspond au nombre de piques que l'IA a frôlé.

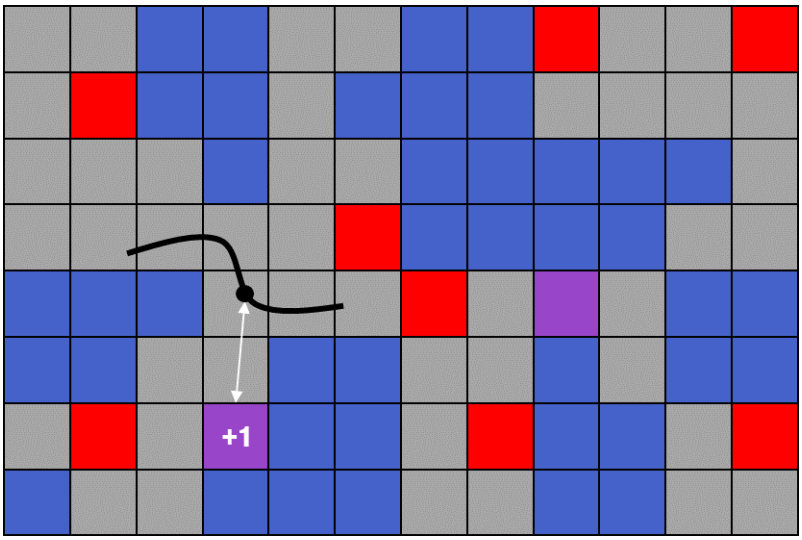


nSpades = 1 + 1

nSpades → +∞ : Le joueur frôlera beaucoup de piques

nSpades → 0 : Le joueur frôlera peu de piques

- **nMonsters** correspond au nombre de monstres que l'IA a frôlé.



nMonsters = 1

nMonsters → +∞ : Le joueur frôlera beaucoup de monstres.

nMonsters → 0 : Le joueur frôlera peu de monstres.

Pour le projet, la distance de validation pour le frôlement des piques et des monstres est ≤ 1. Sachant que chaque case possède une taille de 1 x 1.

X = 3 x SP – 0.1 x nCases – 1 x nSpades – 1 x nMonsters

X	SP	nCases	nSpades	nMonsters
+∞	+∞	0	0	0
−∞	0	+∞	+∞	+∞

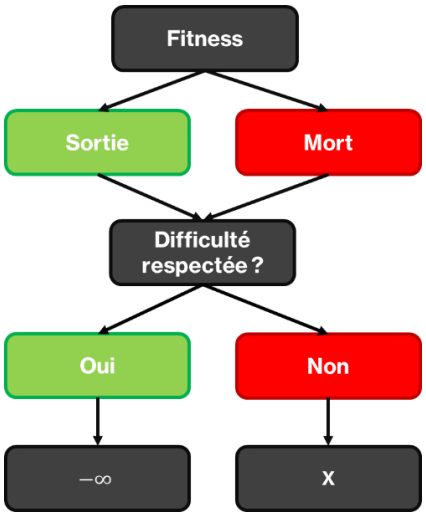
On peut en conclure que si X tend vers +∞, alors le niveau sera de plus en plus irréalisable. Alors que si X tend vers −∞, alors le niveau sera de plus en plus difficile.

Il est important de rajouter des poids derrière chaque variable, il faut qu'on puisse donner plus d'importance à SP par exemple, pour que la PG converge plus rapidement vers un SP = 0.

On peut déterminer la difficulté du niveau avec cette fonction, voici une convention que j'ai utilisé pour le projet :

Variable	Facile	Moyen	Difficile
SP	0	0	0
nCases	10	15	20
nSpades	2	3	4
nMonsters	0	1	2

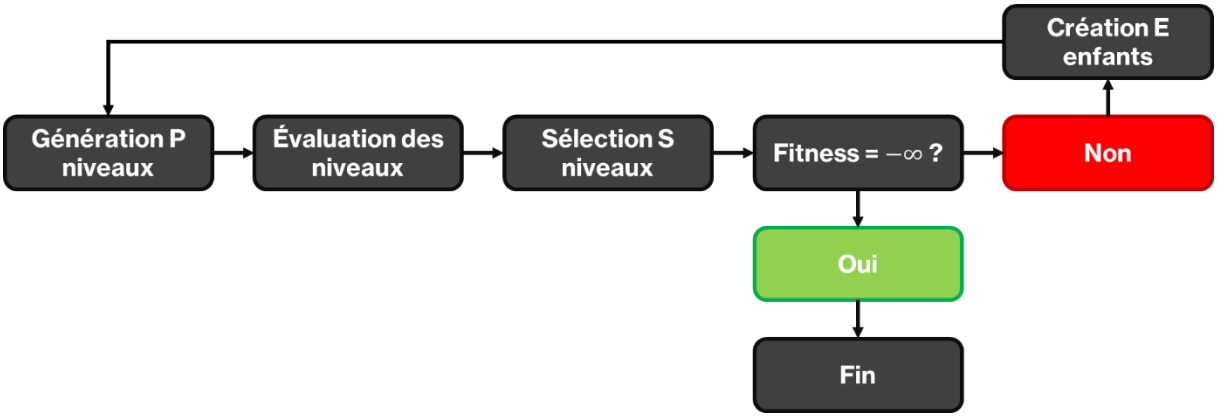
Voici un schéma qui représente l'algorithme d'évaluation d'un niveau :



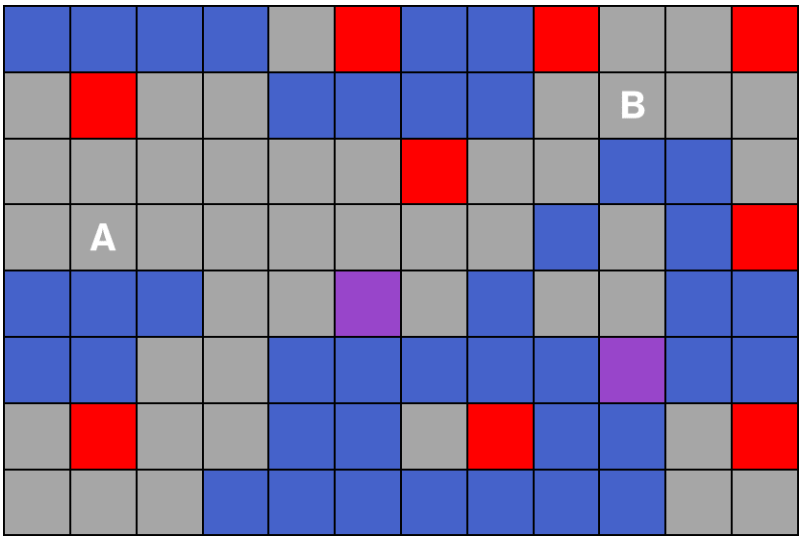
Le bloc « Difficulté respectée ? » inclus aussi, si le niveau est plus difficile que le niveau que l'on souhaite.

Imaginons que l'on souhaite générer un niveau facile, et que l'IA frôle plus de piques ou de monstres, on valide le niveau malgré qu'il soit plus difficile, car nous pourrons enlever le surplus à la complétion (je n'ai pas eu le temps de l'implémenter sur le projet, c'est à l'utilisateur de retirer manuellement le surplus).

Voici l'algorithme pour la génération d'un niveau :



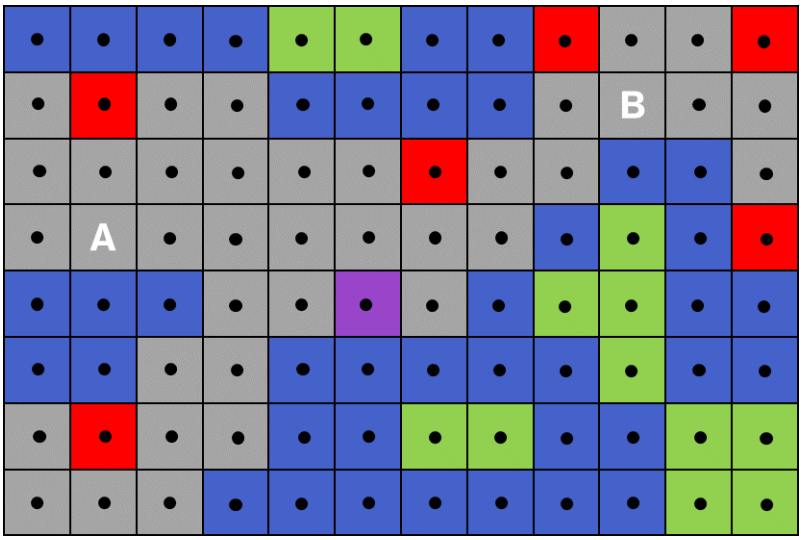
Complétion



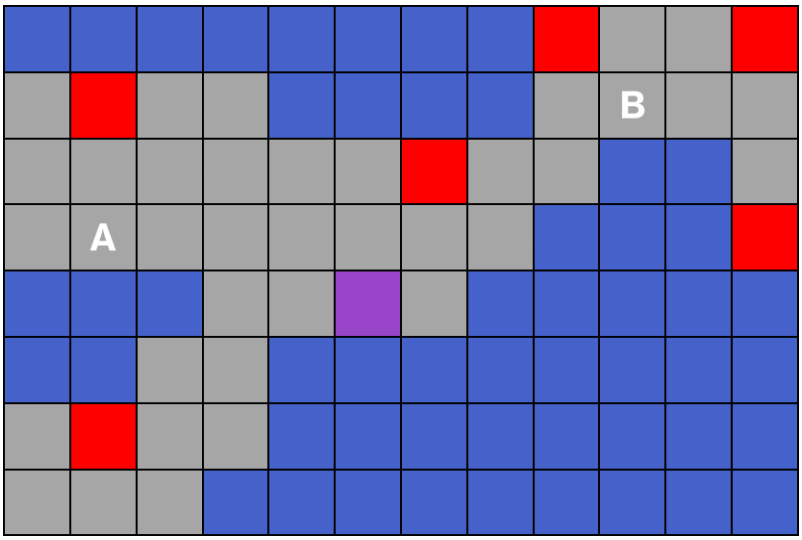
Exemple d'un niveau généré par l'algorithme génétique.

D'après le schéma ci-dessus, on peut voir qu'il y a des cases vides, des piques et des monstres inaccessibles pour le joueur, on pourrait les remplacer par des blocs pour rendre le niveau moins vide.

Pour chaque case vide, chaque pique et chaque monstre, s'il n'existe pas de PCC avec le départ (ou la sortie), on le remplace par un bloc.



Exemple d'un niveau généré par l'algorithme génétique.



Exemple d'un niveau généré par l'algorithme génétique + Complétion.

Conception d’un évaluateur

Description

Lorsqu’on génère un niveau aléatoire, il faut être capable d’évaluer la faisabilité du niveau.

Pour faire ceci, il faut créer une IA capable de tester le niveau, pour ensuite calculer la valeur de X.

Pour avoir une IA capable de résoudre un niveau, il faut l’entraîner sur plusieurs niveaux réalisables.

Je vais vous expliquer quel est le modèle que j’utilise et comment fonctionne l’IA.

Modèle

Pour la création de l’IA, j’ai utilisé le plugin ML-Agents.

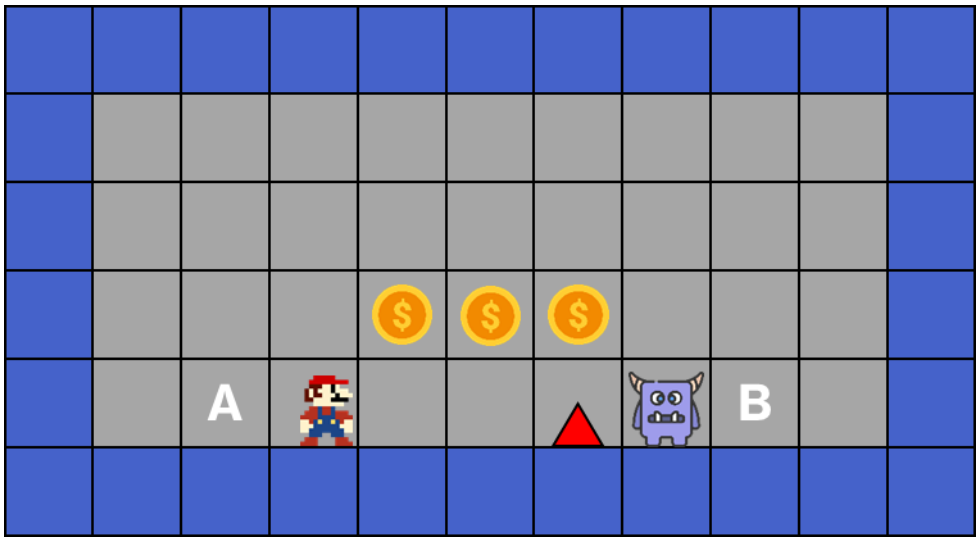
ML-Agents est un plugin sur Unity qui permet de créer des IA facilement avec TensorFlow.

J’ai utilisé pour le projet les paramètres par défaut de ML-Agents (configuration.yaml).

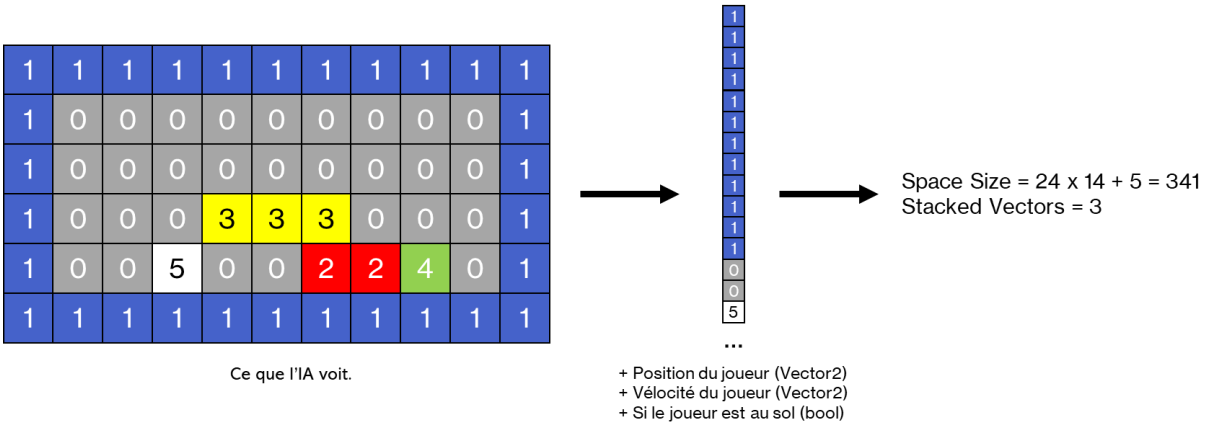
J’ai juste modifié le nombre de neurones par couche (2x128 à 2x512), car les premiers tests avec 2x128 n’étaient pas très fonctionnels.

Entrée

L’IA prend en entrée (à chaque frame) la matrice du monde, avec en plus sa vélocité, sa position et un booléen qui permet de vérifier s’il est au sol ou non.



Ce que vous voyez.



Voici la liste des ID des objets qui sont représentés dans la matrice Vision, cette matrice est lue par l'IA à chaque frame (+vélocité/position/auSol).

Nom	**ID**
Bloc	1
Porte	1
Pique	2
Monstre	2
Pièce	3
Levier	4
Sortie	4 (0 si un levier existe)
Joueur	5
Vide (le reste)	0

Sortie

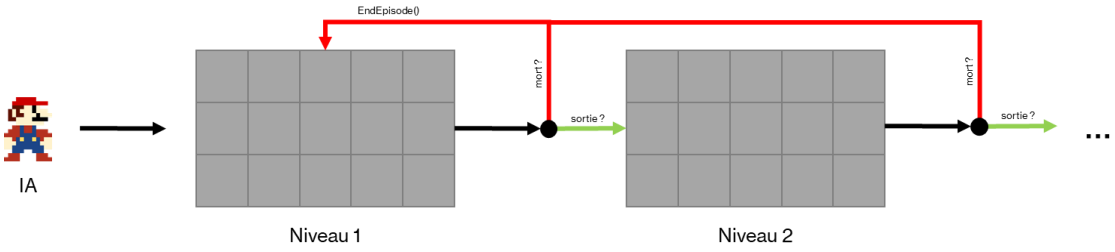
Il y a en tout 5 sorties :

- Gauche, droite ou rien
- Saut ou rien

Important de séparer les commandes en 2 parties, car l'IA (tout comme l'utilisateur) peut très bien aller à gauche et sauter en même temps.

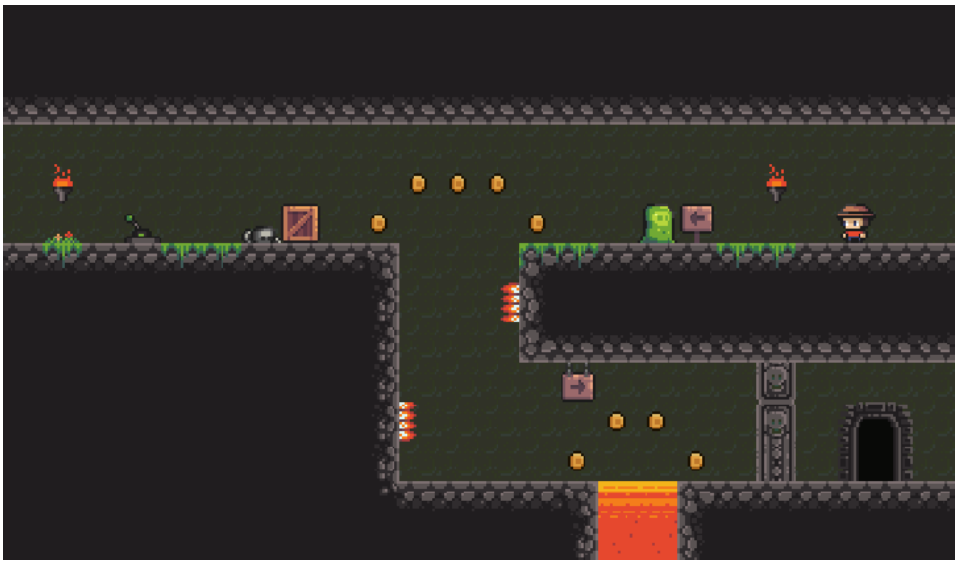
Entraînement

Dans le projet, j'ai créé 14 niveaux, du plus facile au plus dur, plus l'IA termine des niveaux, plus elle sera récompensée.



Exemple processus d'apprentissage.

Exemple niveau 1 pour l'entraînement.



Exemple niveau 7 pour l'entraînement.

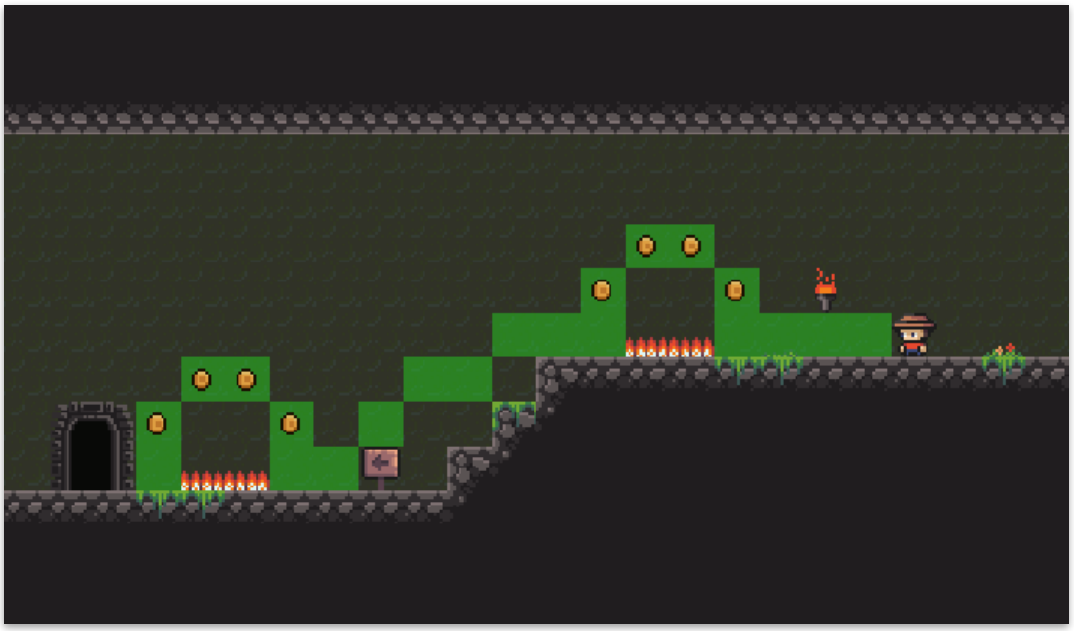
Exemple niveau 12 pour l'entraînement.

Récompense

À chaque épisode, l'IA doit être récompensé pour ses bonnes ou mauvaises actions.

Nom	**Valeur récompense**
Bonus	+1
Pièce	+10
Levier ou Sortie	+100
Step	-0.01

Lorsque l'utilisateur créé des niveaux d'entraînement, il est important de rajouter des bonus qui lui permettront de lui montrer le chemin vers la sortie, au début du projet j'avais utilisé un PCC comme récompense, plus la distance du PCC est court, mieux l'IA sera récompensée, mais le problème c'est qu'il risque de converger à des endroits inaccessibles.



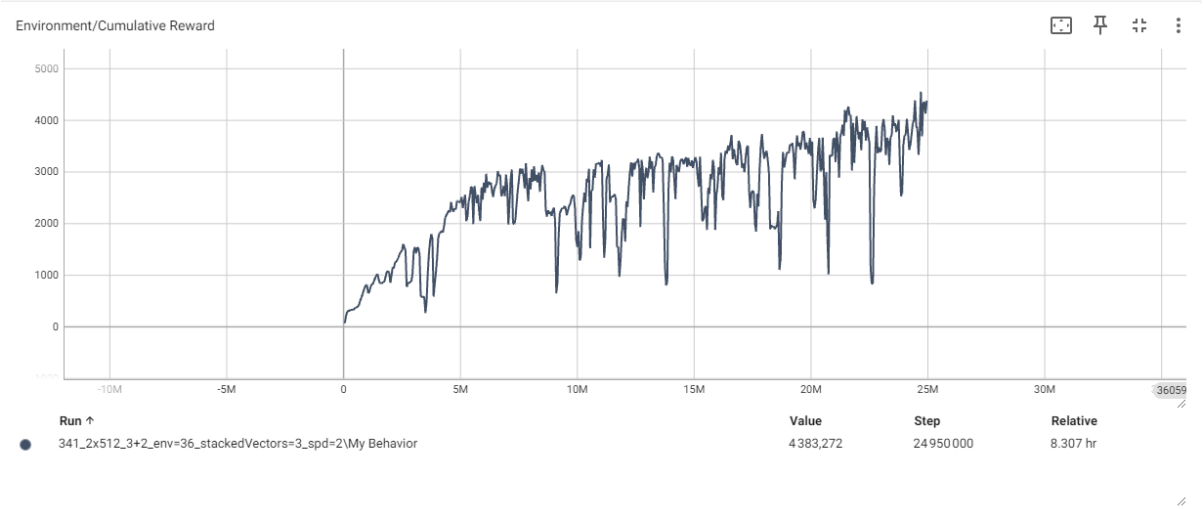
Exemple d'un niveau d'entraînement.

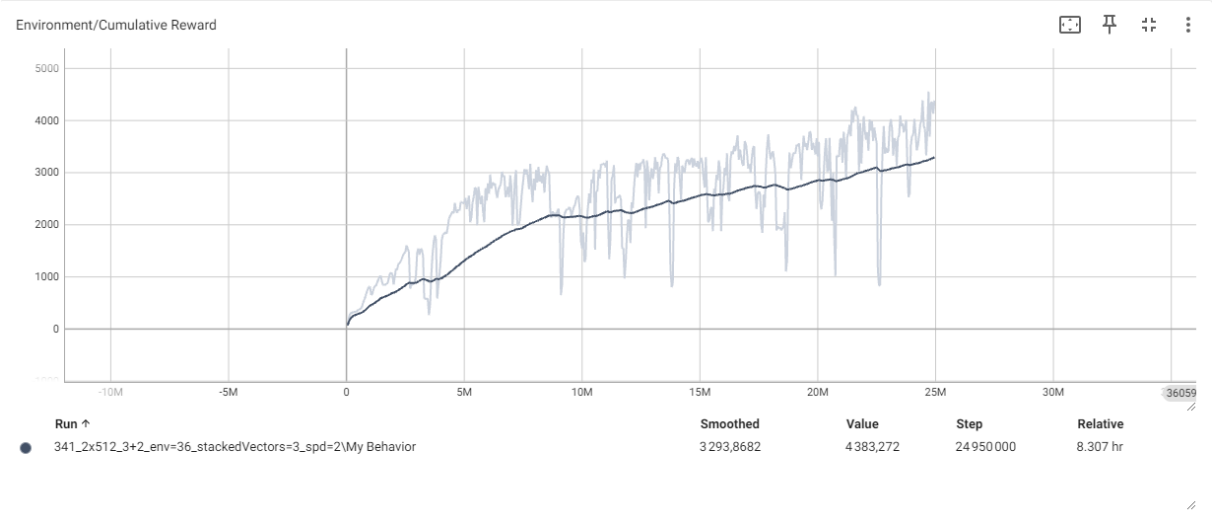
Le nom « Step » correspond à chaque prise de décision de l'IA, plus il prendra de décision, moins l'IA sera performante, je voulais une IA capable de prendre des décisions rapidement.

Résultat

Évaluateur

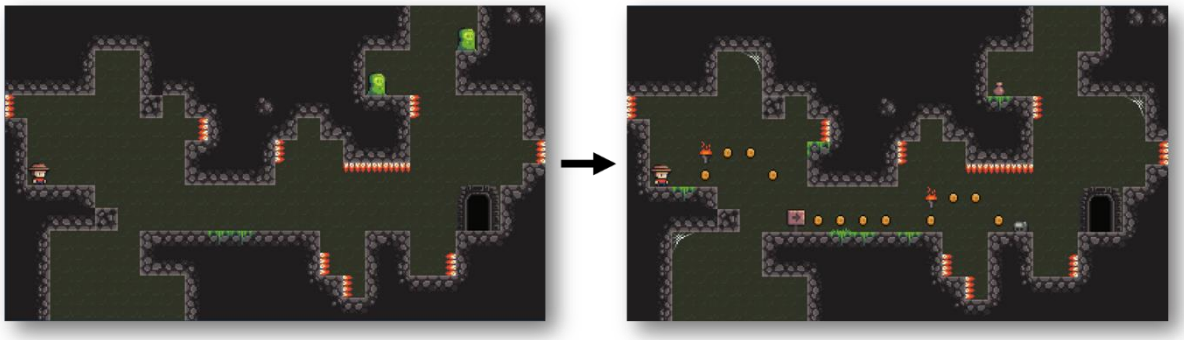
Au bout de 8 heures d'entraînement, l'IA est capable de faire 14 niveaux.



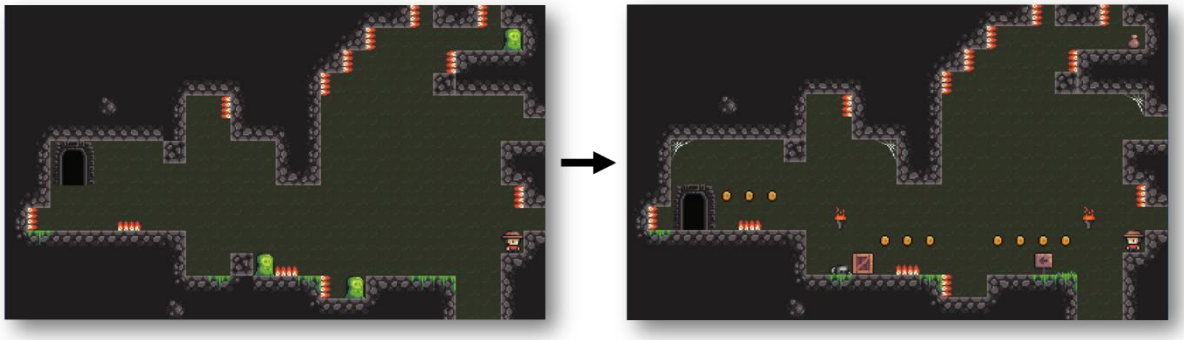


D'après le deuxième graphe, on peut voir que l'IA peut encore apprendre davantage, malheureusement, par manque de temps, je n'ai pas pu tester avec 20, voire 30 niveaux, ce qui aurait été meilleur pour la génération.

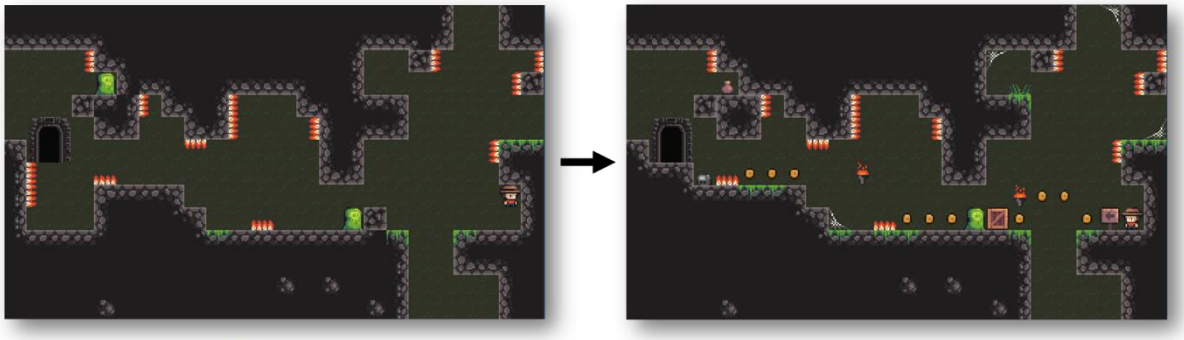
Niveau



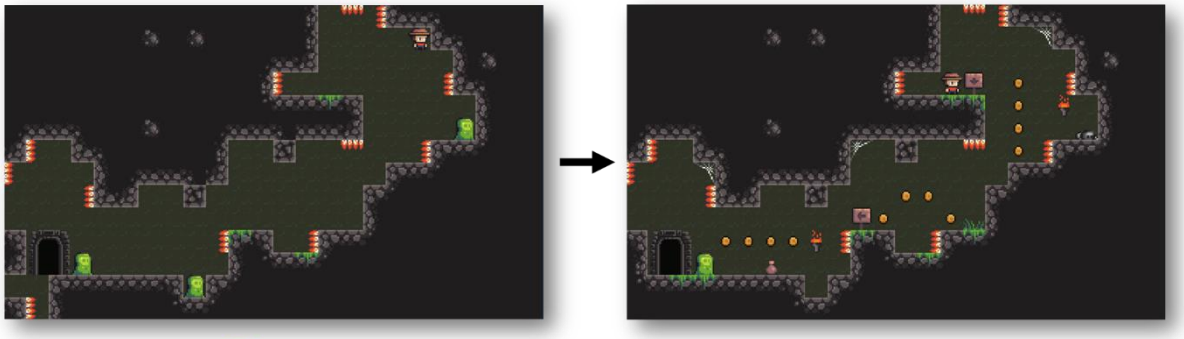
Mode facile
35 générations
-4 minutes



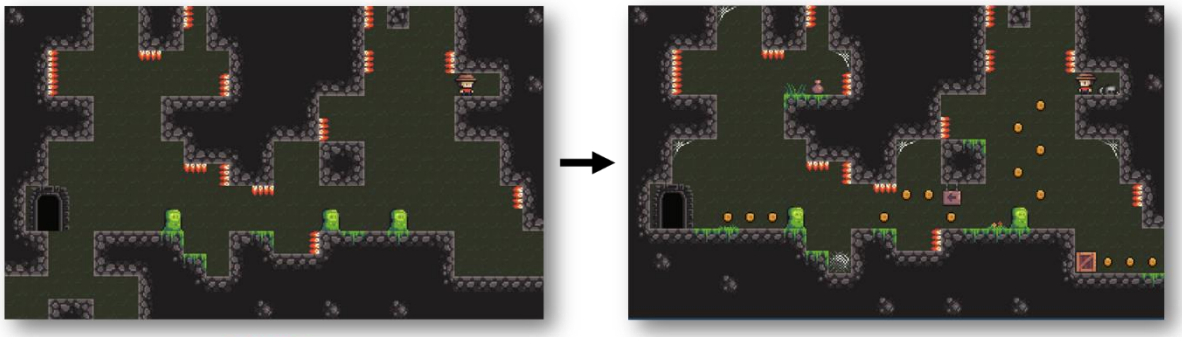
Mode facile
20 générations
-2 minutes



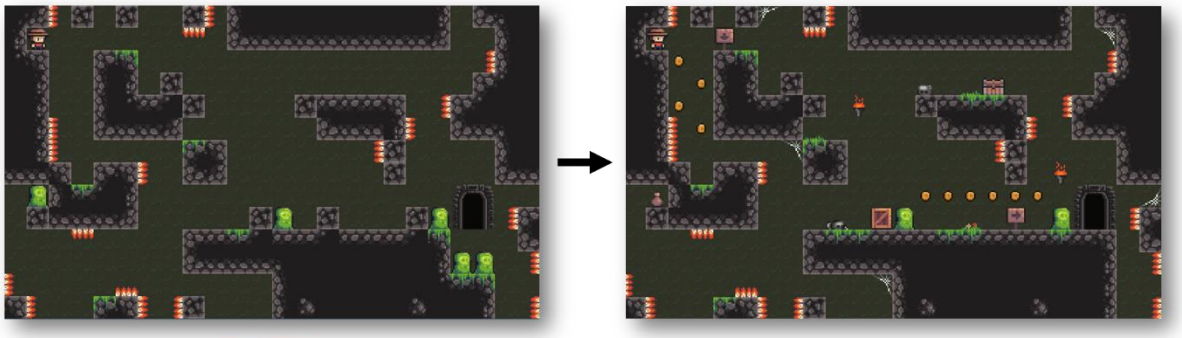
Mode moyen
54 générations
6 minutes



Mode moyen
27 générations
3 minutes



Mode difficile
104 générations
~13 minutes



Mode difficile
70 générations
~8 minutes

Comme nous pouvons le voir sur les différentes générations, on peut y voir plusieurs problèmes :

- **Départ & Sortie** : Le départ et la sortie sont parfois dans le vide, pour y remédier, on pourrait créer une plateforme dès le début de la génération.
- **Piques & Monstres** : Pour la suppression du surplus des piques et des pièges, l'utilisateur doit enlever manuellement, mais on pourrait créer une fonction pour les retirer.
- **Décoration** : Pour la disposition de la décoration, c'est à l'utilisateur de les ajouter, on pourrait créer une fonction simple avec de l'aléatoire pour que le joueur n'ait pas à interagir.
- **Pièces** : Pour la disposition des pièces, je n'ai pas eu le temps d'implémenter un algorithme capable de les placer, donc l'utilisateur doit les placer lui-même.
- **Convergence** : Ne converge pas à tout le temps. Sur certaines générations, pour le mode difficile, la création peut prendre jusqu'à 30 minutes, voire impossible.

Conclusion

Grâce à ce projet, nous pouvons réaliser des niveaux avec 3 types de difficultés en quelques minutes.

Mais il reste tout de même plusieurs améliorations :

- Amélioration de l'évaluateur, l'entraîner sur plus de niveaux (dans plusieurs cas, il ne converge pas vers la solution que l'on souhaite).
- Amélioration de la fonction fitness (par exemple de généraliser les piques et les monstres par une seule case).
- Fonction pour la décoration à ajouter.
- Fonction pour placer des pièces à ajouter.
- Fonction pour éviter d'avoir le départ et la sortie dans le vide à ajouter.
- Amélioration du jeu (ajouter d'autres objets, menus pour les niveaux…).
- Rajouter des difficultés.
- Ajouter un levier et une porte (additionner les SP pour la fonction de fitness).

Après toutes ces modifications, on pourrait avoir un jeu vidéo capable de générer des centaines de niveaux en très peu de temps.