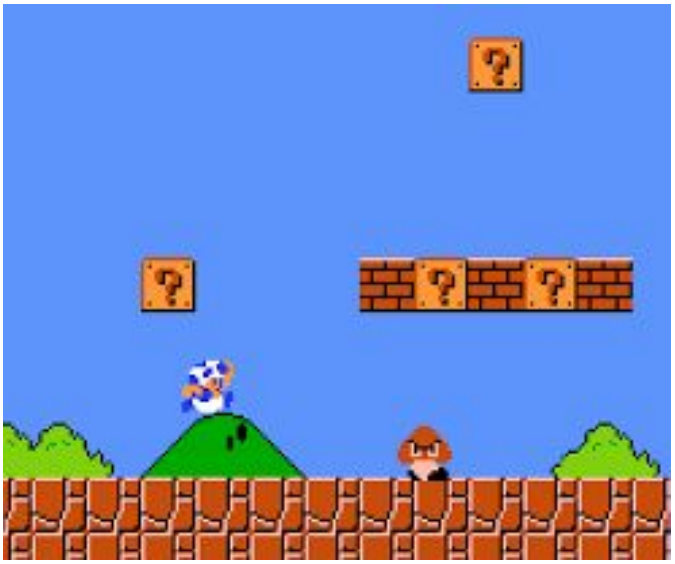
Projet Informatique Individuel (PII)
GRENIER Célestin

Rapport final

PII - Mario Agent



2ème Année, Semestre 8 — 2019 / 2020
Tuteur : M. SARAMITO Jean-Michel

Context

Le jeu support

Super Mario Bros. (SMB) est un jeu vidéo développé par Nintendo et sorti en 1985 sur Nintendo Entertainment System (NES). Il est considéré comme un grand classique des jeux de plates-formes, introduisant au genre plusieurs standards de design tels que le défilement horizontal, les raccourcis secrets.

Comme tous jeux vidéo, SMB est discrétisé en “frames” (un peu plus de 60 par secondes pour la NES). Ce terme désigne généralement l’image du jeu rendue à l’écran, mais son sense plus large d'un “cycle de calcul interne du jeu” est celui qui est utilisé par la suite (mise à jours des personnages, de la musique, des entrées du joueur, …).

Ainsi, à chaque frame, le joueur contrôle les mouvements d’un personnage vus de profil dans un monde en 2 dimension constitué de cases tangibles ou traversables ; les bouton qu’il peut alors utiliser sont :

- Les flèches de directions droite et gauche pour les déplacement horizontaux ;
- Un bouton pour sauter, qui peut être maintenu pour sauter plus haut ;
- Un bouton pour courir, permettant de se déplacer plus vite et sauter plus haut.

Selon la durée d'appuis du bouton de saut, Mario (le personnage contrôlé par le joueur) peut sauter entre 1.25 blocs et 4 blocs en hauteur. Mario peut franchir un trous de 5 blocs de large à vitesse normale et jusqu'à 10 blocs en courant.

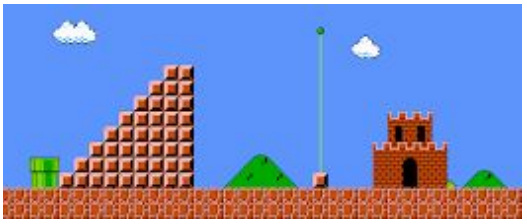
Mario commence tout à gauche du niveau. L'objectif du joueur est alors parcourire le niveaux et d’atteindre le drapeaux de fin situé tout à droite.

L’existant

Étant un jeu de plates-formes, SMB exige du joueur improvisation, réactivité et parfois instinct (par exemple : juger la longueur d’un saut). D’autre part, la (relative) simplicité de ses mécaniques principales en font un bon terrain d’expérimentation pour le développement d’agents artificiels (voir, par exemple, “*The 2009 Mario AI Competition*”). De ce fait, le jeu SMB a déjà été résolus de par l'utilisation d’A*, machine learning ou algorithmes génétiques. Ce projet s'appuis d'ailleur sur un framework très utilisé dans les domaines des la recherches tournant autour de ce style de jeu (génération de niveau, apprentissage automatisé, game design)

Objectif du projet

Ce projet a pour objectif de finir une version simplifié du premier niveau du jeu. On utilisera pour cela une approche de programmation génétique. Contrairement au niveau originel du jeu, le celui-ci est dépouillé de tout ennemis.


…


Le même algorithme, avec peu de variation dans ses paramétrages, pourrait être utilisé pour finir d’autre niveau du jeu.

Le fonctionnement de cet agent s’inspire vaguement de la mémoire procédurale que développe les joueurs de haut niveau. En effet, un joueur très entraîné se montre capable de réagir rapidement à des situations connues. Leurs réflexes leur permet de réagir sans réfléchir distinctement à chaque boutons qu’il doit appuyer, mais en réalisant une action continue.

On appelle alors par la suite “action” est une suite de mouvements atomiques effectués automatiquement et avec consistance jusqu'à en oublier les mouvements individuels. L’écriture en est un exemple plus commun (on ne réfléchit pas au mouvement de chaque doigts pour écrire la lettre 'e’ : c’est une seule “action” ininterrompue).

Algorithmes

Description / Action

Pour imiter le mécanisme de réflex, l'agent fonctionne en cherchant à reconnaître des structures dans l'environnement auxquelles il a associer des action. Il utilisa alors une base de données contenant des descriptions pour "décrire" l'environnement. Un élément de cette base de donnée doit de ce fait conteneurs les information nécessaire pour représenter l'environnement perçu par l'agent, ou une partie de celui-ci.

À ces éléments sont associés :

- un nombre d'occurrences (utilisés durant l'entraînement) ;
- un poids permettant de comparer l'importance des éléments ;
- des coordonnées relatives "préférées" (où on le trouve généralement) ;
- une action à réaliser lors de la rencontre de cet élément.

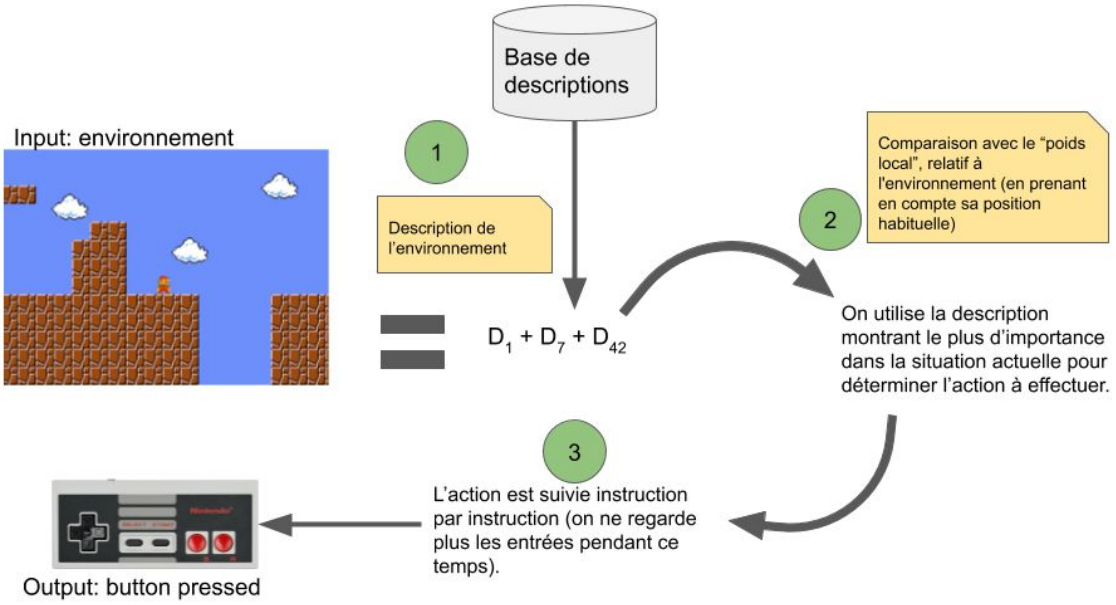
Les description possèdent un poids de base (weight) représentant leur importance hors context. Lorsque l'agent utilisera ces éléments en conditions réelles, il sera amené à calculer le poids local des description qui dépend de sa position relatives "préférées", qu'on désignera par la suite comme "habituelle".

Une action doit correspondre, comme décrit précédemment, à plusieurs mouvements atomiques.

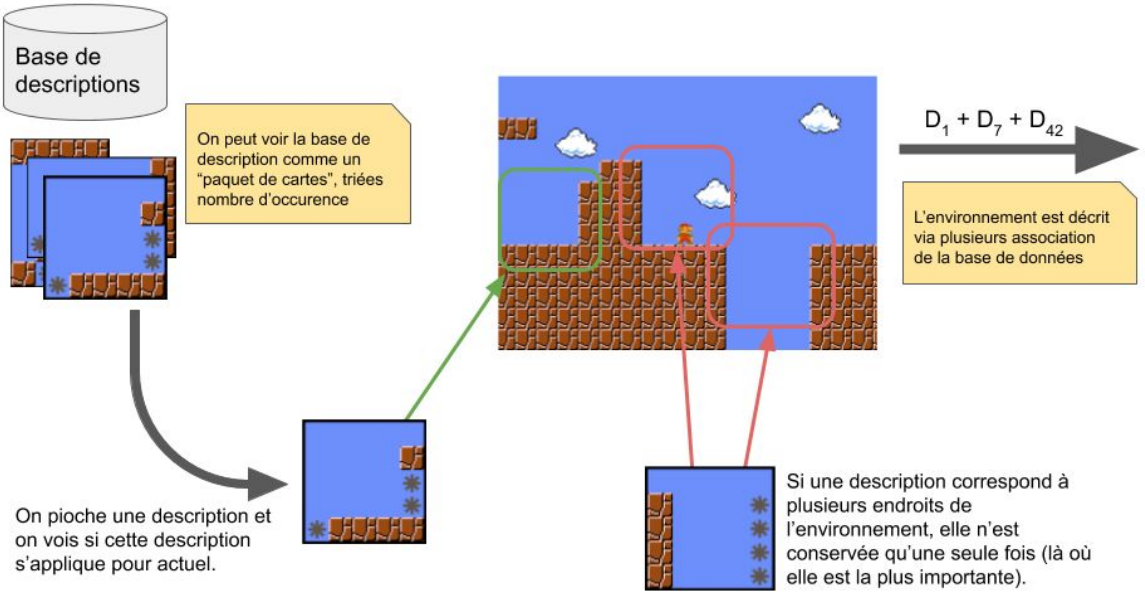
Agent utilisateur

On appel par la suite "agent utilisateur", l'agent utilisant une base de données pour suivre l'algorithme suivant :

1. Comprendre son environnement en "le décrivant en 3 mots" ;
2. Identifier l'élément le plus important parmis ces 3 ;
3. Effectuer l'action associé ;
4. Si le niveau n'est pas fini, retourner à l'étape 1 avec le nouvel environnement.



La description de l'environnement consiste alors à piocher les éléments de la base de données (ce sont des descriptions) par ordre de nombre d'occurrence décroissant et voir si cette description s'applique pour l'environnement actuel de l'agent. Le choix de l'ordre de tris des association permet de mettre en avant celles qui ont le plus de chance d'être utile à l'agent utilisateur en condition réelles (à condition que l'entraînement soit écologique).



Agents expérimentateurs

Pour cet algorithme, L'entraînement se repose sur une dualité entre une utilisation à l'échelle global et des expérimentations à l'échelle locale.

0) Initialisation

Lors de l'initialisation, la base de données est vide. Il s'agit alors de la remplie d'associations description / action initiales. Ces éléments peuvent être généré aléatoirement à condition qu'il soient "crédibles" (i.e. les descriptions doivent correspondre à des situations que l'agent utilisateur pourrait être amené à rencontrer dans le niveau).

1) Utilisation

Cette partie correspond à l'utilisation à l'échelle globale (sur l'intégrale du niveau). Un agent utilisateur est utilisé pour évaluer les associations de la base de données. Des statistiques sont alors récupéré permettant d'estimer :

- Quelle descriptions ont été les plus souvent utilisés ?
- Es-ce que l'action associé a aidé l'agent ?
- Où, dans le niveau, est-ce qu'une description était manquante ?
- Y a-t-il une difficultée majeure que l'agent n'a pas pu passer ?

Ces réponse sont alors utilisées pour déterminer quels éléments de la base de données (description et action) peuvent être améliorées (lors de la partie suivante, équivalente aux mutations d'un algorithme génétique).

2) Expérmientation

Cette partie correspond aux expérimentations à l'échelle local (en certains points du niveau). À partire des statistiques de la partie utilisation, on cherche à augmenter et adapter les éléments de la base de données ; pour ce faire, des agents expérimentateurs sont utilisé. Par opposition aux agent utilisateur, ceux-ci commence à des endroits stratégiques du niveau et n'ont pas pour objectif de finir le niveau, mais d'expérimenter avec les action et description qui leurs ont été fournis.

- Variation dans l'action :
  pour chercher à rendre une action plus efficace (ou peut-être moins efficace, mais plus stratégique), on place l'agent dans une situation qui correspond exactement à la description sur laquelle on travaille et on modifie légèrement l'action consommée ;

- Variation dans l'environnement :
  pour tester l'adaptabilité d'un action ou retravailler une description dans l'objectif d'en trouver les caractéristiques essentiels, on place l'agent dans une situation qui ne correspond pas exactement à la description (voir totalement aléatoire - mais crédible) et on consomme la même action ;

- Nouvel élément :
  pour répondre à une difficulté majeure ou un manque de description, on place un agent dans la situation critique et on expérimente en pseudo-aléatoire.]

Architecture et implémentation

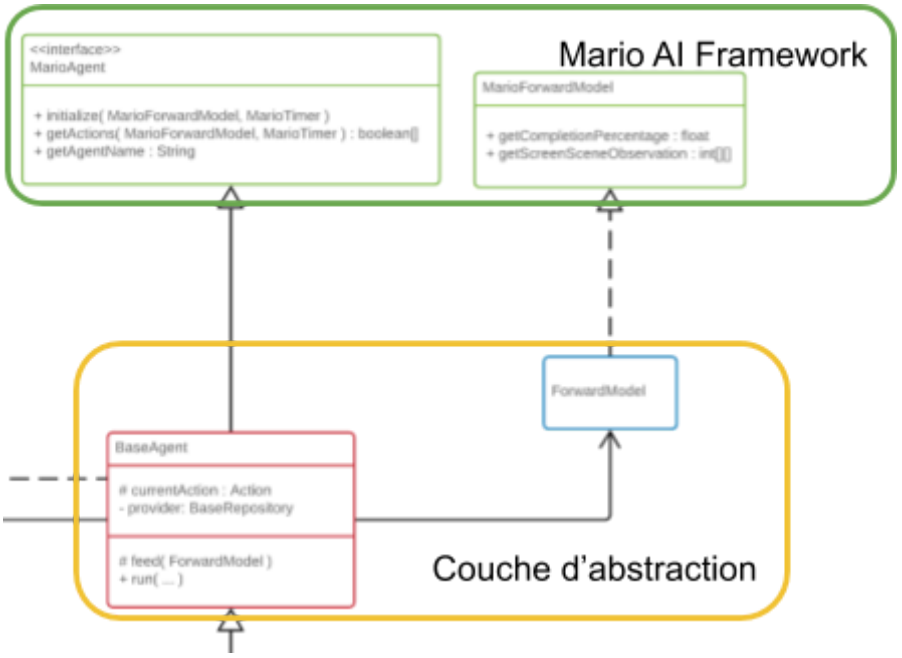
Mario AI Framework

Ce projet s'appui sur Mario AI Framework qui implémente la toute la logique de fonctionnement du jeu. Cette infrastructure est écrite en Java, c'est également la technologie choisis pour ce projet.

Le framework expose une interface de laquelle implémenter pour créer un agent.

En utilisant une instance d'une classe implémentant de cette interface lors de la création d'une nouvelle simulation de jeu, le framework fait appel à une fonction callback avec en paramètre un accès à une représentation de l'environnement pour la frame en cours. L'agent peut alors prendre une décision quant au mouvement qu'il veut effectuer pour la frame en cours.

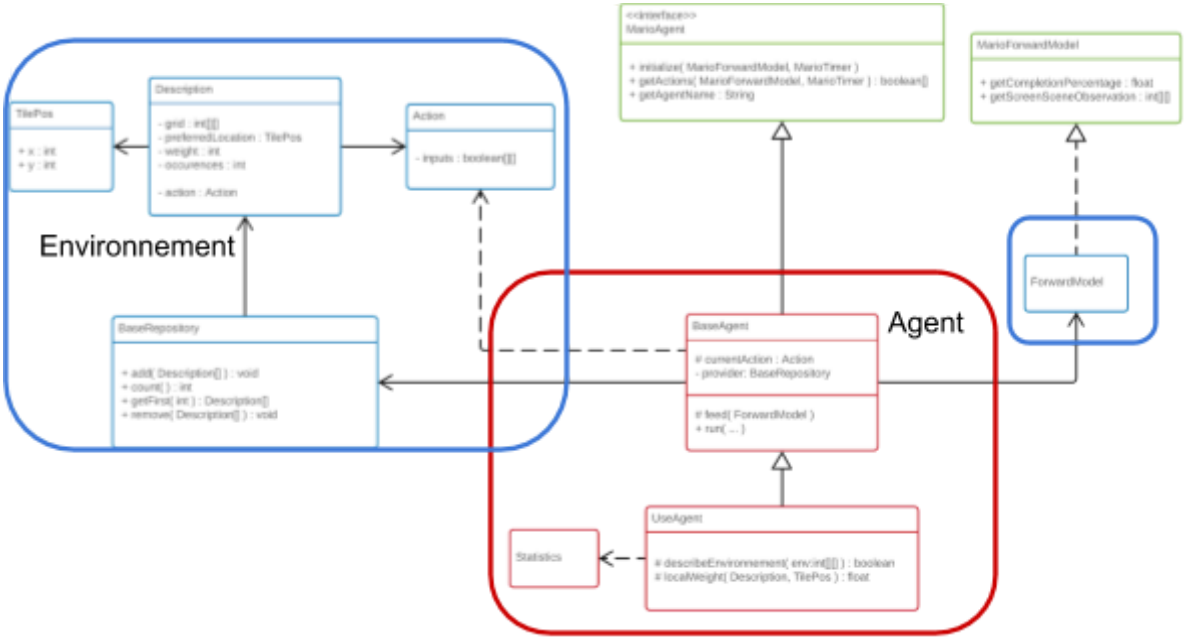
Pour simplifier la suite et avoir plus de contrôle sur les flux de données avec le framework, on implémente une couche d'abstraction qui joue un rôle d'interface entre Mario AI Framework et le reste du projet.



Cette couche consiste en deux classes couvrant les rôle suivant :

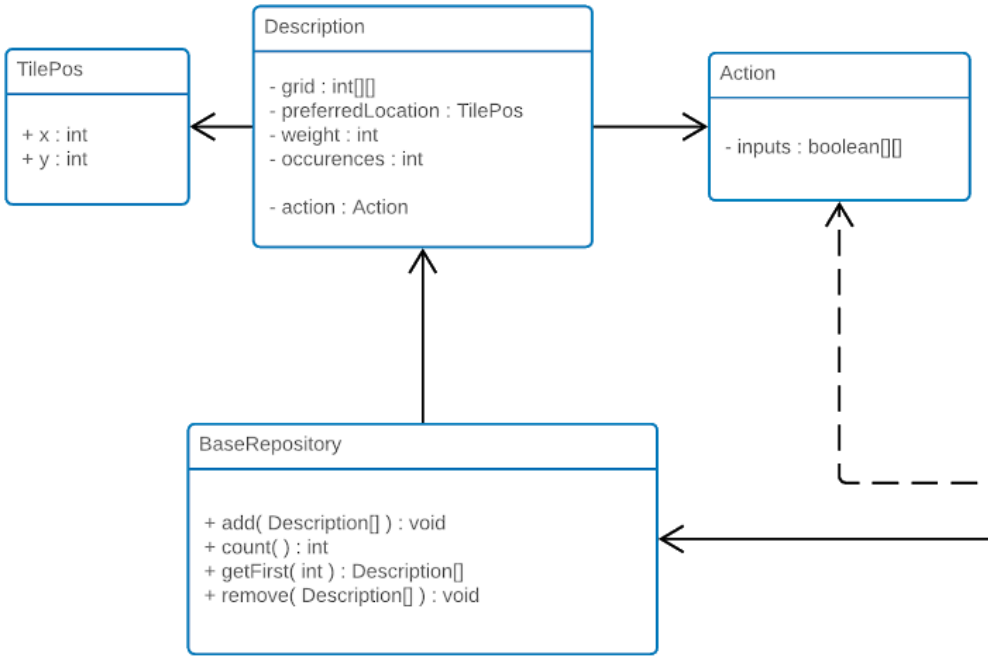
- `BaseAgent` : classe abstraite dérivant de l'interface `MarioAgent` re-définissant une nouvelle base pour les agents spécifiques au projet ;
- `ForwardModel` : encapsulation de l'objet `MarioForwardModel` qui est utilisé au scein du framework pour observer l'environnement.

Classes pour le fonctionnement de l'agent



Environnement

Comme le niveau ne contient pas d'entités mobiles, une représentation suffisante de l'environnement est sous la forme d'une grille d'entiers. On choisis de représenter par un 1 les cases correspondant à un bloc physique dans le niveau et par 0 les case ne contenant rien.



La structure de donnée `TilePos` permet de stocker 2 entiers correspondant aux coordonnées x et y d'une position dans l'environnement.

La classe `BasedRepository` est une classe abstraite qui permet d'accéder à une base de donnée contenant des `Description`s. L'usage d'une classe abstraite permet un fonctionnement indépendant de l'implémentation.

Actions

Lorsque que le framework attend une réponse de l'agent, il fait appel à l'implémentation de sa méthode `getAction` et attend en réponse une liste de booléens indiquant l'étant dans lequel chaque bouton doit être. L'ordre dans lequel les bouton son dans cette liste est notamment données par l'énumération `MarioAction` du framework.

```
public enum MarioActions {
    LEFT(0, "Left"),
    RIGHT(1, "Right"),
    DOWN(2, "Down"),
    SPEED(3, "Speed"),
    JUMP(4, "Jump");
```

L’idée “d’actions” décrite précédemment est implémenté par une classe contenant une suite de liste de booléen. Une `Action` s'étend alors sur plusieurs frames, indiquant l’états des boutons pour chacune.

	gauche	droite	bas	courir	sauter
frame 1	true	false	true	false	true
frame 2	true	false	false	true	true
frame++	…	…	…	…	…

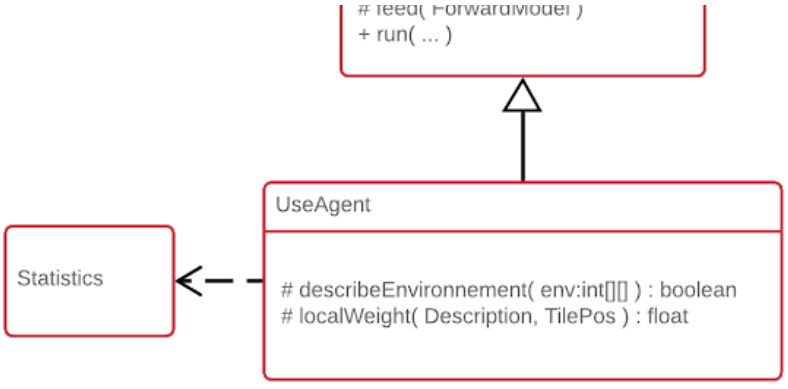
Lorsque l’agent est en train de suivre une `Action`, il suffit alors d’itérer sur la suite d’éléments de l’`Action` en incrémentant un compteur interne. L’état pour les boutons actuel peut alors être mis en retour de `getAction` pour que le framework avance la simulation.

```
// consume the current action
return this.getCurrent().consume();
```

→

```
public boolean[] consume() {
    return this.inputs[this.current++];
}
```

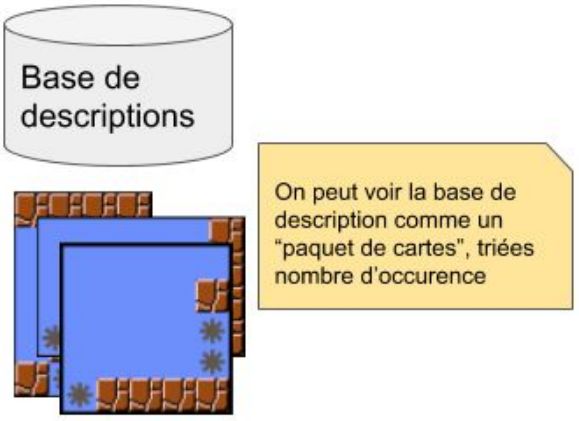
Implémentation de l’agent utilisateur



`UserAgent` dérive de la classe abstraite `BaseAgent` et implémente l’algorithme de l’agent utilisateur décrit plus haut ; cet agent choisi ses `Action`s en décrivant son environnement en parcourant une base de données implémentée comme un `BaseRepository`.

Accès aux Descriptions

Un `BaseRepository` peut être vu comme une pile de cartes dont les éléments, des objets de type `Description`s, sont trié par ordre décroissant d’un critère. Le critère de tris choisis est le nombre d'occurrences de la `Description`, qui correspond au nombre de fois que cette `Description` à été vue / utilisée lors de la phase d’entraînement.



`BaseRepository` définit une méthode qui permet d’accéder au *n* premiers éléments de la pile, en ignorant optionnellement les *p* premiers.

Lorsque l’agent utilisateur parcourt les description de la base de donnée, il le fait en utilisant un tampon (buffer) d’éléments locaux. Pour essayer de limiter le nombre de requêtes effectuées auprès de l’instance de `BaseRepository` (comme c’est une classe abstraite, on ne sait pas quelle calculs coûteux l'implémentation doit faire), il n’effectue une nouvelle requête que si les éléments du buffer n'ont pas suffi à décrire l’environnement.

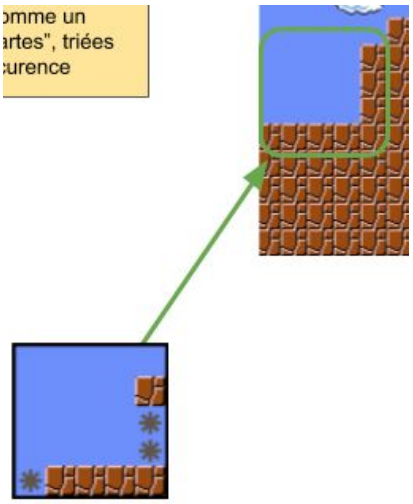
```
itted = 0;
kip = 0;
iption next = null;
<Description> buffer = new LinkedList<>();


/ keep a buff of `to.length` descriptions at hand
f (buffer.isEmpty() && this.prov != null) {
    for (Description item : this.prov.getFirst(to.lengt
        buffer.add(item);
    skip+= to.length;


/ query the next description element to use
ext = buffer.poll();
/ if none is left, break and return (@see `BaseReposit
f (next == null) break;
```

Description dans l’environnement

Pour décrire son environnement, l’agent récupère depuis la base de données les grille des `Description`s. Ces grilles sont assurées d’être de même taille ou plus petite que la grille représentant l’environnement lui-même. On peut donc parcourire l'environnement et essayer de trouver la `Description` comme étant une sous grille de l'environnement en certaines coordonnées.



Pour rappel, les `Description`s sont constituées d'une grille d'entiers valant 1 s'il y a un bloc à cet endrois, 0 sinon. A cela s'ajoute une règle selon laquelle si la valeur de la case est -1, ça correspondre à "l'un OU l'autre" (représentées par des "*" dans l'image). L'objectif de cette règle est de permettre à l'agent de reconnaître des situations qui varient peu par rapport à ce qu'il connaît.

Comme expliqué dans l'algorithme, les `Description`s possède un poids de base (importance hors context). On s'intéresse ici à leurs importance relative à l'environnement et la position de l'agent dans le niveau, calculé par la fonction `reWeight`. Cette fonction de réévaluation du poids local d'une `Description` favorise les positions les plus proches de l'habituelle en divisant le poids par 1+ la distance à la position habituelle :

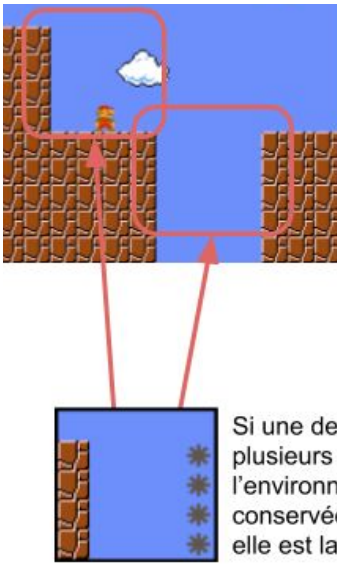
$$\left(\frac{weight}{1+dist^2} \right)$$

```java
/**
 * Local weight for a found description in the environnement (at fit) accounting for its preferred location.
 * @param d
 * @param fit
 * @return
 */
protected static float localWeight(Description d, TilePos fit) {
    return d.getWeight() / (TilePos.distanceSq(d.getPreferredLocation(), fit) + 1);
}
```

Les `Description`s trouvées au plus proche de leur position habituelle sont donc valorisées comme étant les plus importantes dans le contexte de l'environnement.

Plusieurs possibilitées

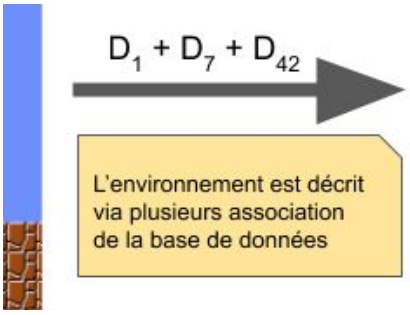
Lorsqu'une `Description` de la base de données à plusieurs possibilitées de correspondance à l'environnement, on utilise encore la fonction `reWeight` pour évaluer à quel endroit cette `Description` est la plus importante.



Arrêt de l'algorithme

Les conditions d'arrêt de l'algorithme de description de l'environnement sont :

- Un nombre de `Description`s jugé suffisant à été trouvé ;
- La base de description de contient plus d'autre `Description`s.



Pour le projet, le nombre de `Description`s suffisantes pour satisfaire la première condition à été fixé à 3. Après quoi l'agent sélectionne l'`Action` qu'il souhait effectuer en réponse à l'environnement. Cette `Action` est celle associé à la `Description` dont l'importance relativement à l'environnement est la plus grande.

Suivre l'Action sélectionnée

Lorsque l'agent utilisateur choisit d'effectuer une `Action`, celle-ci est retenu par l'agent et stocké dans une propriété protégée de `BaseAgent`. Pour chaque frame suivantes, l'agent suis alors les instructions fournies par cette propriété.

```
// consume the current action
return this.getCurrent().consume();
```

→

```
public boolean[] consume() {
    return this.inputs[this.current++];
}
```

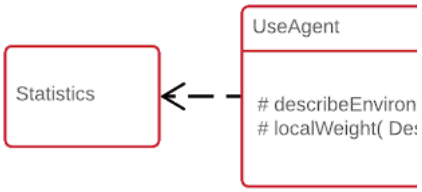
Finalement, une fois que toutes les instructions de l'`Action` ont été suivies, l'agent utilisateur fait à nouveau appel à l'algorithme précédent pour déterminer une nouvelle `Action` à suivre et ainsi de suite.

```
// if no current action or previous a
if (!this.hasCurrent())
    this.findNewAction(model);
```

Ainsi, l'agent n'a pas besoins d'analyse son environnement en continu lorsqu'il est en train de suivre une `Action`, et ce jusqu'à ce qu'il arrive au bout de son action. (Cette partie pourrait être retravaillée pour permettre d'associer "harmonieusement" plusieurs `Action`s.)

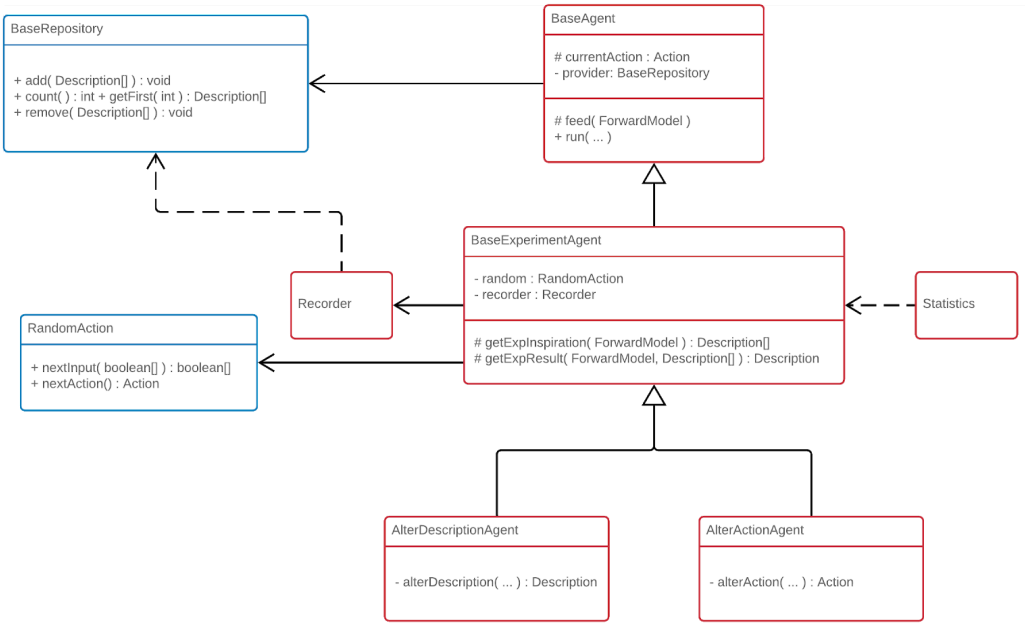
Relever des statistiques

La description de l'environnement est fixé pour relève 3 `Description`s valides. L'objectif est de nourrir un objet de type `Statistics` qui aura pour rôle de permettre de sélectionner les `Description`s / `Action`s lors de l'entraînement.

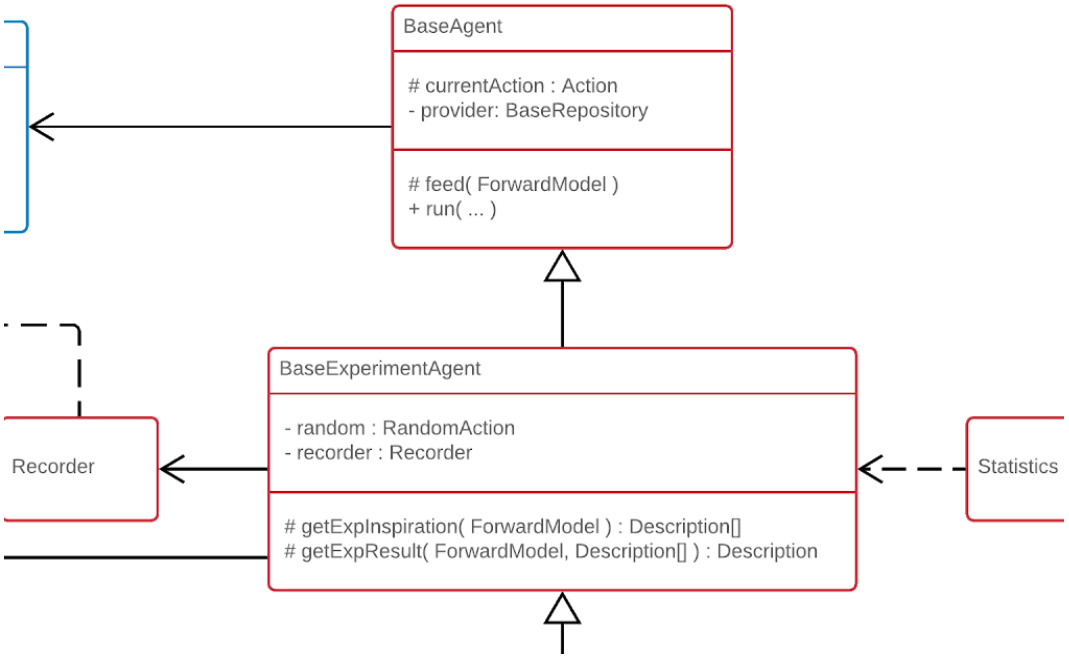


Une instance de cette classe s'occupe alors d'enregistrer tous les faits et gests de l'agent et d'établir quelles `Description`s reviennent le plus souvent et quelle `Action`s ont permis ou non à l'agent de progresser.

Implémentation de l'entraînement



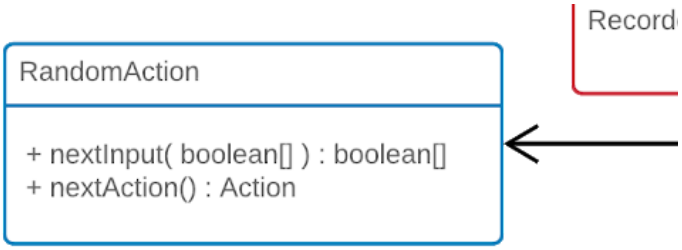
Pour implémenter l'algorithme d'entraînement de l'agent tel que décrit en première partie, plusieurs classes ont dû être ajoutées.



La classe abstraite `BaseExperimentAgent` définit tous les points commun des différent types d'agents expérimentateurs qui seront utilisées.

Initialisation

L'initialisation de la base de données emploie une classe de génération aléatoire procédurale d'`Action` dérivant de la classe Java `Random`.



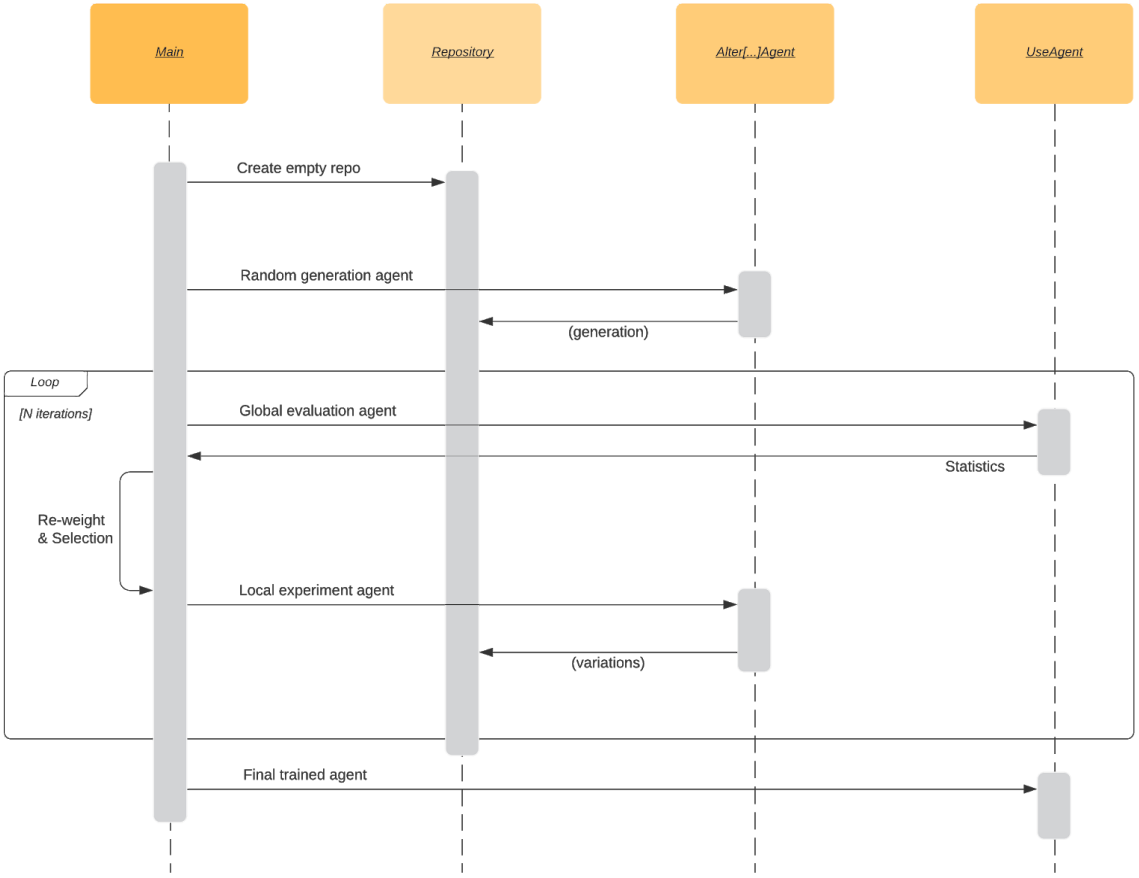
Cette classe permet de générer des `Action`s aléatoires qui ne soient pas simplement du bruit pour autant. En effet, les boutons enfoncé à la frame *k* influence ceux qui le seront à la frame *k+1*, et ainsi de suite. De même, les combinaisons de touches impossibles telles que droite et gauche ne peuvent pas apparaître.

La génération procédurale d'`Action`s implémentée dans la méthode `nextAction` consiste alors à créer une première liste de bouton appuyés (frame 1), puis la méthode `nextInput` implémente la logique de génération de la frame suivante.

Dans le cadre de ce projet, les `Action`s initiales sont générées avec une durée de 30 frames, mais le nombre de frames qu'une `Action` couvre peut être paramétré pour être aléatoire.

Comme indiqué précédemment, les association initiales de la base de donnée doivent respecter un critère de "crédibilité" selon lequel les `Description`s doivent correspondre à des situations que l'agent utilisateur pourrait être amené à rencontrer dans le niveau.

Pour ce faire, les `Action`s générées aléatoirements sont associé à des `Description`s correspondant exactement à un endroit du niveau (la grille de la `Description` est exactement la même que l'environnement). Cela est réalisé en plaçant un agent temporaire à un endroit aléatoire du niveau.

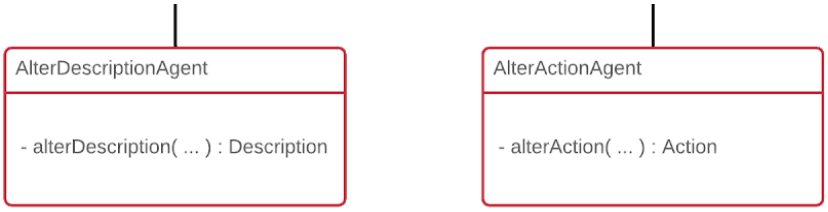


Utilisation

Un agent utilisateur classique (tel que vue précédemment) est utilisé pour évaluer les associations de la base de données. On obtient donc un objet de type `Statistics`.

Expérmientation

Deux types d'agents expérimentateurs sont dérivé de `BaseExperimentAgent` pour répondre aux deux proposition (altération d'action et altération de description). Ces agents implémentent alors les méthode `alterDescription` et `alterAction`.



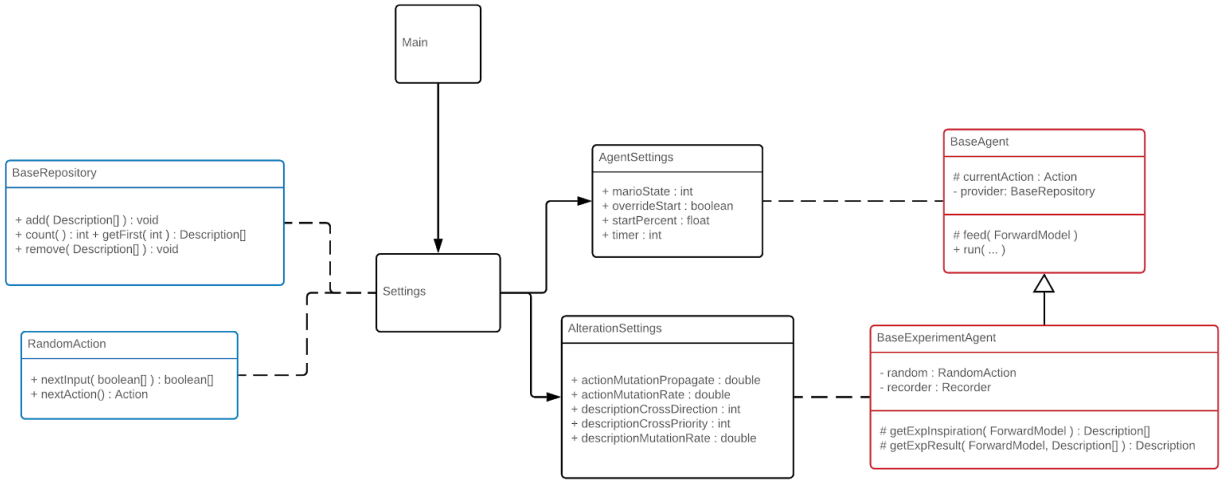
L'agent `AlterActionAgent` est instancié avec une association de la base de données et pour rôle de retravailler l'`Action` de l'association à l'environnement où elle est utilisée. On applique alors une mutation aléatoire sur les boutons appuyés pour chaque frames de l'`Action`. Cette mutation suis un taux et une règle de propagation qui indique si une mutation dans une frame donnée doit affecté la frame suivante, auquel cas on se rabat également sur la méthode `nextInput` de `RandomAction` pour tenter de réduire l'apparition potentiel de bruit dans la nouvelle Action.

Les `Descriptions` sont modifiées par une instance de `AlterDescriptionAgent` qui se contente d'appliquer l'`Action` associée mais à un autre endroit du niveau que là où l'agent utilisateur s'en est servi. La nouvelle `Desctiption` résulte à la fois de la grille de l'ancienne mais également de la grille de l'environnement et d'un procédé de mutation aléatoire. À travers ce procédé, la grille peut être recoupé sur ses bord de une ou plusieurs rangées (autant à l'horizontal qu'à la vertical) et peut également subire des modification des valeurs :

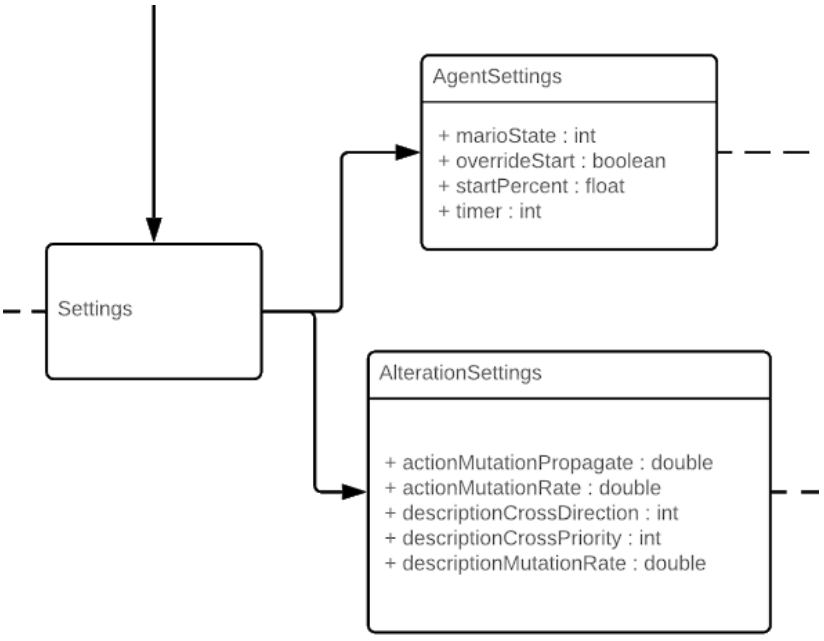
- une case peut prendre la valeur -1 si elle est en conflits avec l'environnement
- une case contenant -1 peut devenir un 1 ou un 0 avec une chance équiprobable

La particularité de ces agents les distinguant de l'agent utilisateur est qu'il ne sont pas positionné au début du niveau lors de leur tours de simulation. Idéalement, le point de départ de ces agents doit être déduits à partire des `Statistics` qui permettrait alors de déterminer s'il y a un endroit où l'agent utilisateur n'a pas pu progressé (par exemple par manque d'association dans la base de données ou pas mauvaise valorisation des `Description`s et leur `Action`). Cependant, pour garder l'entraînement suffisamment simple durant le développement, ils sont placé de manière aléatoire homogène entre le début et la fin du niveau.

Paramétrage



Comme l'entraînement introduit de nombreux paramètre modifiables dans le projet, il est préférable, voir indispensable, de les réunir sous une seule classe `Settings`.



D’autre structures de données sont également définis pour réduire et centraliser les paramètre de certains éléments du projet plus particulièrement.

Tests, exploration et observations

Les paramètres

Les paramètres mis à disposition par le biais de la classe `Settings` et les autre structures de données sont les suivants :

- le nombre d'itérations du cycle utilisation / expérimentation lors de l'entraînement ;
- le nombre de frames pour les `Action`s générées ;
- la méthode de réévaluation du poids de base (hors context) des `Description`s ;
- les différents paramètres de la phase d'altération :
    - le taux de mutation des `Action`s,
    - la propagation des mutation à travers les frames d'une action,
    - le taux de mutation des grilles des `Description`s,
    - la façon dont les `Description`s sont recoupées avec l'environnement ;
- les différents paramètres de simulation pour chaque types d'agent :
    - le temps accordé,
    - l'état de Mario,
    - le point de départ dans le niveau ;
- la façon de déterminer avec les statistiques quelles associations sont intéressantes.

Seulement certains de ces paramètres ont pus être testés et certains valeurs on été déterminé comme optimales malgré de faibles résultats. Le paramètre "état de Mario" n'a pas été utilisé car il n'est pas d'une réelle importance ici (il permet principalement à Mario d'encaisser un dégât supplémentaire de la part des ennemis, mais ceux ci on été retiré du niveau).

Influence de certains paramètres

Concernant la génération d'`Action`s, le nombre de frames d'une action à été fixé assez tôt à 30 soit ½ seconde. Dans les faits, ce paramètre s'est montré être négatif pour des valeurs supérieurs à environ 50 frames : l'agent utilisateur se retrouve "bloqué" à suivre une `Action` trop longtemps après le point où elle était jugée pertinente. À l'inverse, j'ai jugée qu'en dessous de 20 frames les `Action`s ne serait plus pertinentes : dans le cadre du jeu, cela correspond approximativement à la durée d'un saut de taille normal. Il est à noter que même si les `Action`s générées sont de 30 frames de durée, les versions altérées qui en découlent au cours de l'entraînement sont plus longues ou plus courtes.

Le nombre d'itérations est un paramètre qui à été modifié tout au long des démarches d'exploration de par son importance. Cependant, il ne s'est pas révélé impliquer beaucoup de changements dans le comportement de l'agent au delà d'une valeur de 100 ~ 120. Je l'ai fixé à 50 itérations, ce qui offrait un bon compromis entre la durée d'entraînement et une performance représentative de l'agent utilisateur final.

La méthode de réévaluation du poids de base des `Description`s est un élément critique de l'algorithme d'entraînement. En effet, l'objectif est de trouver un moyen d'évaluer l'importance "hors context" d'une partie de l'environnement à partir [uniquement (ou presque)] de ses statistiques d'utilisation (nombre de fois que l'agent utilisateur a aperçus cette `Description` dans l'environnement lors de ce cycle d'utilisation, quelle distance l'`Action` associée à permis de parcourir au maximum, minimum, quelles autre `Description`s apparaissent en parallèles, …). Durant une itération de tests, les association ajouté à la base de données héritait des associations sur lesquelles elles étaient basées. De par sa complexité, c'est un des paramètre sur lequel les expérimentations se sont le plus étendues. Malheureusement les résultats ne sont pas convaincants. On peut cependant faire les observations suivantes :

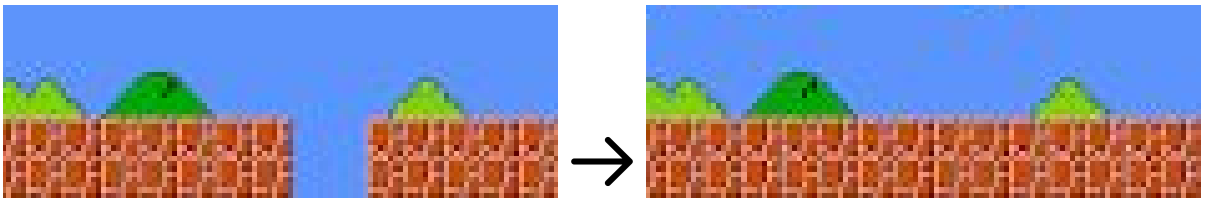
- Augmenter le poids d'une association en fonction de la distance maximale que l'`Action` a permis de parcourir et la diminuer avec la distance minimale permet d'indiquer à l'agent que les meilleurs associations sont celles qui le font progresser.
- Augmenter le poids avec la variété d'usage de l'association permet de valoriser la diversité, mais dans ce cas l'agent à tendance à ne pas avancer du tout (manque de récompense).
- Uniquement augmenter le poids des `Action`s créé immanquablement une situation "d'élitisme", ou une unique association (2 dans le meilleur des cas) est tellement valorisée qu'elle surpasse tout autres même lorsqu'elle est observée loin de sa position habituelle.
- De manière opposée, appliquer un poids "de base" (constant ou hérité) au nouvelle association empêche l'agent de se décider sur quelle `Action`s effectuer : à chaque itération l'agent utilisateur a un comportement totalement différent car les acquis sont repoussé au fond de la base de donnée (manque d'apprentissage).

Ayant été ajouté relativement tards et étant plutôt complexs, les différents paramètres de la phase d'altération n'ont pas été suffisamment testés, notamment :

- Les effet des modification sur la propagation des mutation à travers les frames d'une `Action` n'ont pas été observé comme impactant l'apprentissage. Ce paramètre a pour objectif d'éviter l'apparition de "bruits" trop rapide dans les `Action`s (tels qu'appuyer sur une touche et la relâcher instantanément) ; ainsi si une frame d'une `Action` est modifié (par mutation), ce paramètre détermine si et comment la frame suivante doit être impactée.
- Les modifications sur la façon avec laquelle les `Description`s sont recoupées avec l'environnement ont un impacte très contrasté sur le comportement de l'apprentissage. Le principe de ce paramètre est que les nouvelles associations soient influencées autant par la `Description` dont elle s'inspire que (avec un degré de variation) par l'environnement de l'expérimentation local. Ce paramètre a montré les même aspects que le manque d'apprentissage ou l'effet "d'élitisme" selon son réglage.

Influence du niveau lui-même

Finalement, même en adaptant les paramètres de l'entraînement, l'agent n'était pas toujour capable de finir le premier niveau, même simplifié de ses ennemis, à cause de la présence de trous dans le niveau. En effet, comme l'agent est conçu et entraîné pour voir la présence d'obstacles, il n'est pas capable de voir l'absence de sol. Le niveau à donc été adapté comme un paramètre en soit : plusieurs itérations ont été testées (notamment d'autres niveaux du même jeu, des niveaux générés procéduralement ou même manuellement).



Les résultats les plus convaincants ont été obtenus en couvrant les trous dans le niveau. L'agent arrive alors à finire le niveau, cependant sa stratégie consiste souvent à simplement sauter vers la droite à un rythme plus ou moins régulier. La base de données contient alors généralement un élément très utilisé (encore l'effet "d'élitisme") car c'est une `Description` très vague de l'environnement qui est toujours vraie reliée à une `Action` qui fonctionne quelque soit la situation. Le reste des éléments ne servant pas, ou n'étant que des toutes petites variations de la première association.

Description				Action associée
0	0	-1	-1	Sauter vers la droite
-1	0	0	-1	
0	-1	0	-1	

Gestion de projet et tutorat

Compte rendu d'avancement

Ce projet à été réalisé sous la tutel et les conseils de M. SARAMITO Jean-Michel. La stratégie de communication utilisée emploie un document au format tableur permettant de faire un compt rendus organisé des événements de la semaine.

Etudiant	
Projet	
Semaine	10-14 février 2020

Evènements Positifs	Evènements Négatifs
Les réalisations	Les Difficultés
Les opportunités	Les Risques

Evolution du planning	
Décisions attendues	Prochaines étapes court terme

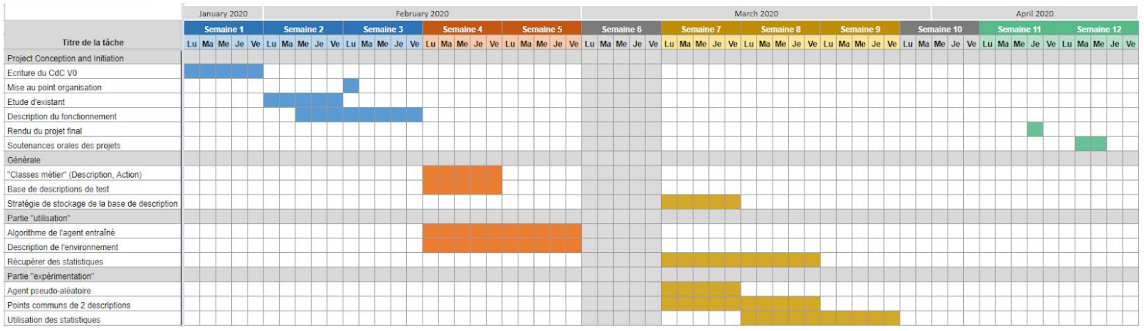
Ce document à été remplis et transmis de manière hebdomadaire et m'a permis à plusieur reprise d'évaluer mon avancement réelle sur le projet :

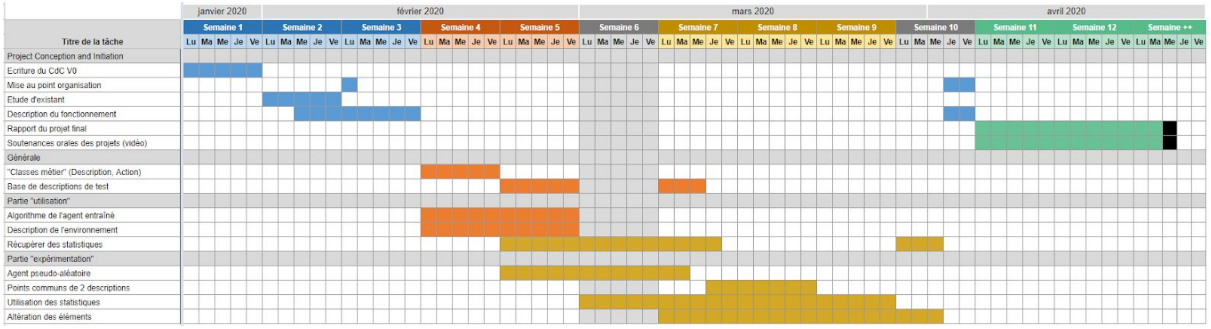
- Les réalisations de la semaine et les opportunitées permettent de remarquer comment certaines difficultées ont été surmontées et quel fonctionnalitées ont été implémentées ;
- Les difficultées peuvent être clairement listées dans la partie événements négatifs, ce qui permet en particulier de reprendre d'une semaine à l'autre les tâches qui causeraient du retard, ou ne serait pas encore réglées ;
- Les risques peuvent être communiqués facilement pour permettre d'établir les choix possibles et de prendre une décision ;
- La case "évolution du planning", en parallèle avec la case "prochaines étapes court terme", est un aperçus directe des potentiel retards ou avances prise par rapport le planning.

Le lancement du projet s'est organisé par une rencontre avec le tuteur dans les locaux de l'école. De même, l'avancement du projet à été discuté à mi-parcours afin de repérer des erreurs de gestion et recadrer l'organisation.

Planification

Ce projet s'est étendus sur 13 semaines (12 semaines était initialement prévues). La planification du projet à été détaillée à l'aide d'un diagramme de Gantt présentant les tâches réparties sur la durée du projet. Le planning final (présent ci-dessous, en bas) présente les temps qui ont effectivement été dédiés aux tâches. Il se compare au planning tel qu'initialement prévu (en haut).





De manière générale, l'organisation est restée relativement la même. La liste des tâches n'a globalement pas changée et toutes ont été réalisé à un degré bien suffisant.

Titre de la tâche
Project Conception and Initiation
Ecriture du CdC V0
Mise au point organisation
Etude d'existant
Description du fonctionnement
Rapport du projet final
Soutenances orales des projets (vidéo)
Générale
"Classes métier" (Description, Action)
Base de descriptions de test
Partie "utilisation"
Algorithme de l'agent entraîné
Description de l'environnement
Récupérer des statistiques
Partie "expérimentation"
Agent pseudo-aléatoire
Points communs de 2 descriptions
Utilisation des statistiques
Altération des éléments

Enseignements et conclusion

Le Java est un langage qui nécessite l'écriture de beaucoup de code pour réaliser peu, ce qui a tendance à ralentire le développement. Le choix de cette technologie a été faite en connaissance de cause avec pour intérêt que, ayant déjà utilisé ce langage pour des projets personnels, je n'ai pas eu de difficultées et ai pus utiliser le framework mentionné. D'autre part, les principes de programmation du Java implique d'organiser son code tôt et de concevoir une architecture structurée.

Architecture

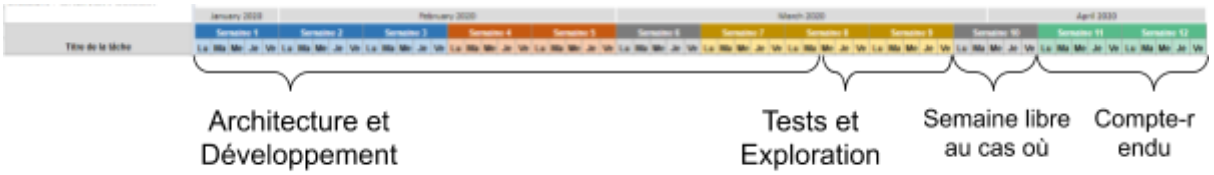
L'architecture définie pour le projet s'est révélée suffisamment souple et a permis d'atteindre l'objectif fixé. Cependant, même si son implémentation a été réfléchie dès le début du projet, la phase d'entraînement se révèle la plus sujet à erreurs. Cela est notamment lié au fait que les principales réflexions à prioris ("sur papier") ont portées presque exclusivement sur le fonctionnement de l'agent utilisateur et pas assez sur la partie entraînement.

Le choix d'utiliser des agents expérimentateurs pour effectuer des mutations aux `Description`s et leur `Action` s'est révélé superflux ; l'idée originelle était d'assurer un environnement écologie pour les expérimentation locales en les effectuants au travers d'un agent qui aurait les même restrictions et possibilitées que l'agent utilisateur. Pourtant, tels qu'ils sont employés dans le projet, ils pourraient être omis et remplacé par des fonctions de mutation et crossover plus classiques.

Centraliser le prélèvement de statistiques d'utilisation de l'agent dans une objet de type Statistics c'est révélé une bonne pratique, cependant il aurait été nécessaire de préparer avant coup quelles éléments statistiques peuvent être utilisés. Dans les fait, les statistiques récupérés se révèlent ne pas être suffisantes (en plus d'être maladroitement utilisées) pour déterminer efficacement quelles associations de la base de données sont vraiment intéressantes.

Démarche exploratoire

Le planning initiale prévoyait de se concentrer avant tout sur les tâches de développement de l'agent et de l'architecture du projet.



De ce fait, la démarche exploratoire a été repoussé et principalement réalisé en fin de projet. Ce choix d'organisation est critiquable car en plus de ne pas avoir laissé beaucoup de temps pour une partie critique du projet, cette partie à dû être réalisé en parallèle de nombreux autre projets scolaire. Cependant, il restait plus important d'implémenter l'architecture générale du projet pour supporter ces exploration.

Par exemple, l'ajout de la classes Settings centralisant les paramètres s'est fait relativement tard ; durant le développement, les paramètres de l'entraînement était répartis à travers plusieurs fichiers, ce qui rendait les tests fastidieux.

Conclusion finale

Le projet informatique individuel a été pour moi une occasion d'expérimenter avec une idée de projet personnel dont je n'avais que les bases fondamentales mais pas le temps de mettre en tests pratiques. La consigne initiale de ce projet étant de mener un projet informatique de la conception à la livraison, la proposition d'un projet exploratoire s'avère plutôt inapproprié (par exemple ce projet est, en parallèle, un support d'exercices pour d'auteur matière et une démarche exploratoir ne menant pas à un produit final utilisable, ce n'est pas toujour possible de suivre les attentes).

Mais malgré tout, les objectifs du module ont été remplis ; ce projet m'a permis de renforcer une expérience d'analyse et de réalisation d'un programme informatique complexe et vient s'ajouter à mon apprentissage en tant qu'élève ingénieur en cognitique. Ceci a également été une expérience donnant un aperçu plus probant de l'organisation professionel qu'un projet à l'échelle d'une (ou plusieurs) équipes implique.

Pour conclure quant au projet lui-même, l'algorithme employé s'inspire vaguement (initialement) de plusieurs autre algorithme connus (programmation génétique et grammatical evolution, rules-based, state machine, …) mais en dévie beaucoup. Il se voulait de proposer un agent capable de créer un "vocabulaire" (des objets porteurs de sens pour l'agent lui-même et dans son environnement) et d'introduire les même mouvement complexes qu'on observe dans l'évolution d'une langue d'un point de vue strictement sémantique. Les comportement recherché n'ont pas pu être observé et cela est principalement lié à la méthode d'entraînement, mais également à l'implémentation de l'agent. Je n'ai pas encore pu déterminer avec certitude de quels éléments sont à revoir, mais toutes autre recherches que je ferais dans cette direction partirai d'une nouvelle base.