Travail n°1
Création d'un agent aspirateur

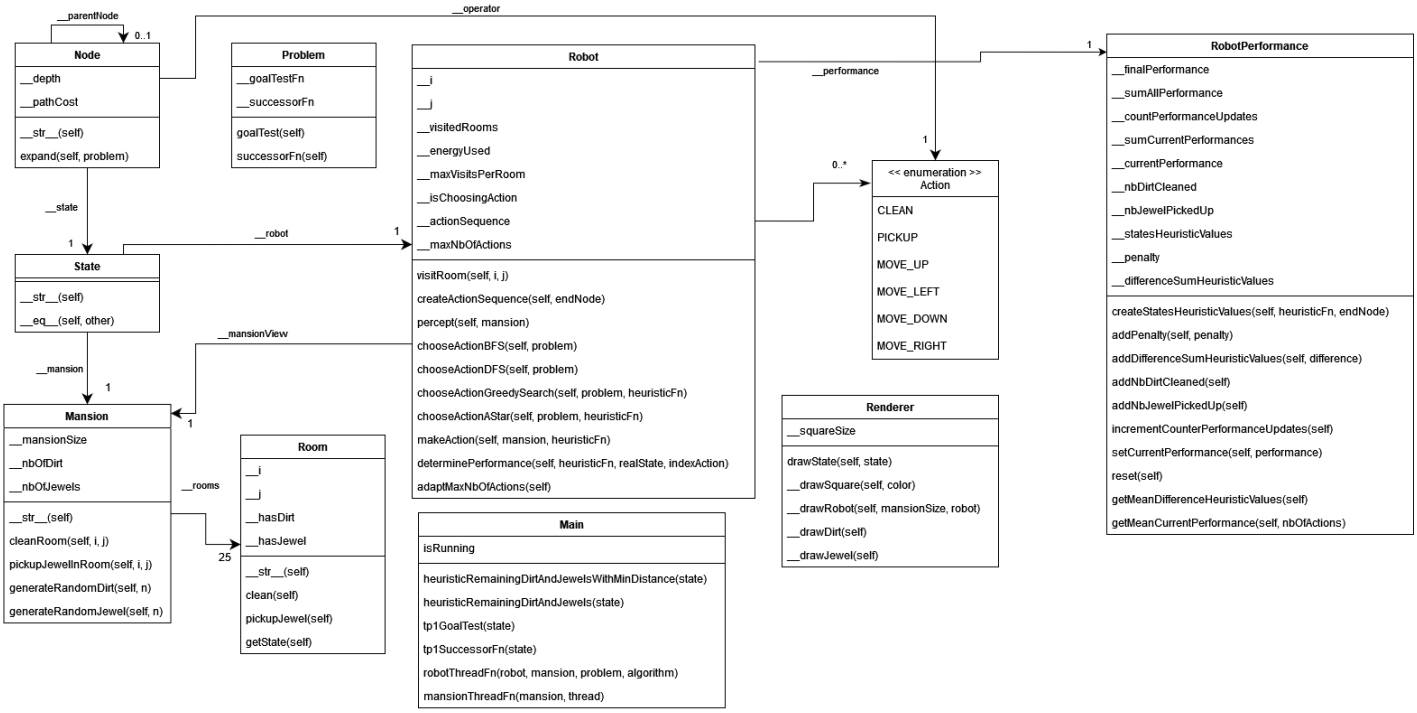
Groupe :
CAVANI Nicolas (CAVN06029909)
GRILLON Théo (GRIT04010107)
OECHSLIN Killian (OECK17039908)

I - Instructions pour le lancement de l'application

/! La seule chose à faire attention pour que l'application marche correctement est de la lancer (fichier src/main.py) à partir du dossier ***src*** de notre arborescence de projet.

II - Explication de la modélisation

Avant de passer aux explications détaillées, nous pouvons déjà observer le diagramme UML de notre application :



Tout d'abord, nous notons la présence de 2 classes assez génériques : ***Node*** et ***Problem***. La classe *Problem* possède 2 attributs qui sont des fonctions : *__goalTestFn* qui permet de définir la fonction de but de l'agent pour le problème actuel, et *__successorFn* qui permet de générer les nœuds enfants d'un nœud. Ces 2 fonctions peuvent donc être changées pour s'adapter à un autre problème que celui de ce TP. Ensuite, la classe *Node* permet (comme son nom l'indique) de modéliser les noeuds de l'arbre de recherche, et possède pour cela comme attributs l'état courant, le noeud parent, l'opération qui a permis d'arriver à ce noeud, la profondeur et le coût actuel du chemin. *Node* dispose d'une méthode ***expand*** qui sert à générer les nœuds enfants du nœud

actuel, grâce à la fonction de succession définie dans *Problem* (le fonctionnement des principales méthodes sera expliqué en troisième partie).

Ensuite, pour modéliser l'état courant de chaque nœud nous avons la classe ***State***. Cette dernière possède un attribut *__mansion* qui représente l'état global du manoir et un attribut *__robot* qui représente l'état du robot.

Plus spécifiquement au problème posé, nous avons créé 4 classes :

- ***Room*** modélise une pièce du manoir, avec sa position (attributs *__i* et *__j*) et son état de propreté (*__hasDirt* et *__hasJewel* qui sont des booléens). Ensuite, pour ce qui est des méthodes, ***clean*** permet de nettoyer complètement la pièce (qu'il y ait de la poussière et/ou des bijoux), ***pickUpJewel*** permet simplement de ramasser un bijou dans la pièce, et ***getState*** renvoie l'état de propreté de la pièce (0 : vide ; 1 : poussière ; 2 : bijou ; 3 : poussière + bijou).
- ***Mansion*** représente l'environnement dans lequel évolue le robot. Il a pour attributs les 25 pièces (de type *Room*) du manoir, la taille du côté du manoir ainsi que le nombre total de poussières et de bijoux actuellement présents dans celui-ci. La méthode ***cleanRoom*** permet de nettoyer une pièce du manoir (en mettant à jour les variables de *Mansion* et de la *Room* ciblée), et ***pickUpJewelInRoom*** permet de ramasser un bijou dans une des pièces.
- ***Robot*** est la classe modélisant notre agent intelligent basé sur les buts. Comme attributs, il possède une position (*__i* et *__j*), sa propre vision du manoir (*__mansionView*) qu'il met à jour lorsqu'il n'a plus d'actions à effectuer, un compteur du nombre de visites de chaque pièces (*__visitedRooms* qui est un tableau 2D ayant la même taille que le manoir) qui, lié à l'attribut *__maxVisitsPerRoom* (indiquant le nombre maximal de visites possibles d'une même pièce au cours de l'exploration), nous permet de limiter les possibilités du robot en termes de parcours, par exemple en interdisant de repasser plusieurs fois par la même pièce au cours d'une même exploration. Par la suite, *__energyUsed* représente l'électricité dépensée par le robot suite aux actions effectuées ; *__performance* est de type *RobotPerformance* (que nous détaillerons juste après), et nous sert à gérer la mesure de performance du robot, notamment dans son apprentissage de la meilleure fréquence d'exploration ; *__isChoosingAction* est un attribut simplement utilisé dans l'affichage graphique et n'a pas d'impact dans les algorithmes ; *__actionSequence* stocke les actions que le robot a choisi afin d'atteindre un état propre (plus de poussières ni de bijoux dans notre cas). Ces actions sont modélisées grâce à l'énumération ***Action***, visible sur le diagramme UML. Le dernier attribut est *__maxNbOfActions* et il est utilisé dans l'apprentissage de la meilleure fréquence d'exploration. Il sert à limiter le nombre d'actions que peut faire le robot dans un même plan d'action.

  Par la suite, *Robot* possède de nombreuses méthodes qui permettent d'implémenter le fonctionnement de cet agent intelligent : ***visitRoom*** incrémente le compteur de visite d'une pièce ; ***createActionSequence*** remplit la séquence d'action du robot à partir du noeud final et en remontant ensuite jusqu'au noeud initial ; ***percept*** permet au robot d'observer l'environnement (le manoir) et de stocker l'état actuel de ce dernier dans sa mémoire ; ***chooseActionBFS***, ***chooseActionDFS***, ***chooseActionGreedySearch et chooseActionAStar*** sont 4 méthodes qui vont permettre à l'agent de rechercher le moyen d'atteindre son but (manoir propre). Le fonctionnement de ces méthodes sera détaillé plus tard dans le rapport. Il y a ensuite la méthode ***makeAction*** qui permet au robot d'agir sur le "vrai" manoir à partir de la séquence d'actions qu'il a déterminée. Nous voyons donc que le robot possède bien un état mental "Beliefs-Desires-Intentions" : la vision que le robot a du manoir correspond au "Beliefs" ; les méthodes de choix des actions (*chooseActionXXX*) correspondent au "Desires" et la séquence d'actions à effectuer correspond au "Intentions".

Enfin, Robot possède 2 dernières méthodes qui sont ***determinePerformance*** et ***adaptMaxNbOfActions***. *determinePerformance* sert à calculer et mettre à jour la performance du robot lors de l'exécution de ses actions, tandis que *adaptMaxNbOfActions* permet d'ajuster la limite maximale d'actions que le robot peut faire dans un même plan d'action, en se basant justement sur la mesure de performance du plan d'action précédent. Par cela, le robot peut adapter sa fréquence d'exploration pour maintenir le manoir propre en fonction de l'environnement et ne pas faire d'erreurs (aspirer un bijou).

- ***RobotPerformance*** est une classe qui va permettre de stocker et calculer les données nécessaires à l'apprentissage de la meilleure fréquence d'exploration (dont nous venons juste de parler). L'explication de la méthode utilisée sera détaillée en 3ème partie, mais globalement *RobotPerformance* stocke la performance calculée suite à la dernière action effectuée (*__currentPerformance*), la somme des performances d'un même plan d'action (*__sumCurrentPerformances*), la somme des performances depuis le lancement du robot (*__sumAllPerformances*), la performance totale depuis le lancement (*__finalPerformance*) qui est simplement la moyenne des performances, calculée avec *__sumAllPerformances* et un compteur du nombre de performances qui ont été sommées (*__countPerformanceUpdates*). Ensuite, *RobotPerformance* stocke le nombre de poussières nettoyées (*__nbDirtCleaned*), le nombre de bijoux ramassés (*__nbJewelPickedUp*) et les pénalités du robot (*__penalty*), et ce pour le plan d'action actuel.
  Pour ce qui est des méthodes de *RobotPerformance*, la plupart servent à mettre à jour les attributs utiles au calcul de performance. Nous pouvons quand même faire un petit commentaire sur la méthode ***reset*** qui permet de remettre à 0 tous les attributs liés au calcul de performance d'un plan d'action, et ***getMeanCurrentPerformance*** permet de calculer la moyenne des performances du plan d'action actuel, afin qu'on puisse utiliser cette valeur dans l'adaptation du nombre maximal d'actions du robot.

Nous voyons sur le diagramme UML le ***Main***, qui n'est pas vraiment une classe en tant que telle, mais qui contient la définition de nos 2 heuristiques, de la fonction de but (***tp1GoalTest***) et de la fonction de succession (***tp1SuccessorFn***), qui sont ensuite transmises à la classe *Problem*. C'est également dans *Main* que sont définies les fonctions de notre robot (***robotThreadFn***) et de notre manoir (***mansionThreadFn***), qui sont ensuite exécutées dans 2 threads indépendants.

Finalement, nous avons une classe ***Renderer*** qui sert à l'affichage graphique. Il n'est donc pas vraiment utile de détailler ici ses attributs et méthodes.

Remarque : Au niveau des méthodes de chaque classe, nous n'avons pas indiqué sur le diagramme UML les constructeurs, getter et setter car cela n'ajoute pas vraiment d'information.

III - Explication de l'implémentation

a) Explication du fonctionnement global

Pour l'implémentation, tout le programme a été codé en Python, au moyen de classes. De plus, 2 threads tournent en parallèle : le thread du robot et le thread du manoir. Chacun des threads tourne en boucle jusqu'à ce que la variable ***isRunning*** devienne *False* (c'est le cas lorsque l'utilisateur quitte l'application). De plus, à la fin de chaque boucle nous mettons en pause chaque thread pendant 1 seconde, afin que le déroulement du programme soit correctement visible pour un humain. Les threads sont également créés en tant que *daemons*, ce qui permet une exécution continue et totalement en arrière-plan.

Dans le **thread du manoir**, nous appelons simplement les méthodes de génération de poussières et de bijoux qui, avec une certaine probabilité donnée, vont faire évoluer l'état de propreté du manoir.

Dans le **thread du robot**, nous appelons en premier la méthode ***percept*** qui permet au robot d'observer l'environnement (via ses capteurs) et de mettre à jour son état interne. Une fois que ce dernier a été actualisé, nous appelons la méthode ***chooseActionXXX*** (dépend du choix d'algorithme de l'utilisateur) qui va permettre de trouver une séquence d'action permettant de nettoyer le manoir. Enfin, vient la méthode ***makeAction*** dans laquelle le robot agit véritablement sur le manoir (avec ses effecteurs).

Pour l'implémentation de l'aspect graphique, la librairie ***turtle*** est utilisée. Elle permet de dessiner facilement une interface graphique au moyen de diverses fonctions préétablies.

b) Explication de la recherche de solution et du fonctionnement du robot

La **fonction de but** est définie au moyen de la recherche du nombre d'éléments (poussières et bijoux) dans les pièces du manoir. On récupère cette valeur et l'objectif est qu'elle soit égale à 0. Le but du robot est donc de nettoyer toutes les poussières et de ramasser tous les bijoux.

La **fonction de succession** commence par récupérer l'état des pièces, notamment celle où se situe le robot. Ensuite, si une poussière ou un bijou est présent dans la pièce où se trouve le robot, l'état enfant généré consistera à nettoyer la poussière ou ramasser le bijou (note : la fonction de succession privilégie en premier le ramassage du bijou). S'il n'y a pas de poussière ou de bijou dans la pièce, dépendamment de la position actuelle de l'agent, la fonction de succession indique comment il doit se déplacer, et génère les états enfants correspondants.

Les **algorithmes de recherche** utilisent 4 stratégies différentes, 2 non-informées et 2 informées :

- Non-informée : Nous avons implémenté une approche ***BFS***, qui utilise un tableau de nœuds et les consulte un à un, en ajoutant les enfants à la suite. Ensuite, nous avons aussi implémenté un algorithme ***DFS***, qui part du même principe mais ajoute les enfants du nœud visité en début de tableau et non à la fin, ce qui fait que l'algorithme explore d'abord une branche complète avant de remonter à la racine.
- Informée : Nous avons implémenté une approche qui utilise un **algorithme glouton** basé sur une heuristique (l'heuristique sera détaillée juste après). Cet algorithme utilise aussi un tableau de nœuds et cherche celui dont la valeur de l'heuristique est la plus faible. Une fois trouvé, il parcourt et ajoute ses enfants à la fin du tableau. Nous avons aussi implémenté l'algorithme ***A**** qui possède un principe très similaire à l'algorithme glouton, la seule différence sera sur la valeur attribuée à chaque noeud où *A** calcule la valeur de l'heuristique du nœud et ajoute également le coût du chemin actuel (pour arriver à ce nœud).

Pour ce qui est des heuristiques, nous en avons implémenté 2 différentes, prenant à chaque fois un état en paramètre et renvoyant la valeur associée, mais une seule est vraiment utilisée. La **première heuristique** renvoie juste le nombre de poussières et de bijoux présents dans le manoir dans l'état évalué. Ensuite, dans la **deuxième heuristique** (celle qui est utilisée) nous calculons la distance entre le robot et la poussière la plus proche, et entre le robot et le bijou le plus proche, et nous multiplions chaque distance respectivement par le nombre de poussières total et le nombre de bijoux total dans le manoir. (Note : nous additionnons 1 aux valeurs des distances les plus proches, car, sans cela lorsque le robot se trouve sur une case ayant une poussière ou un bijou cela renverrait 0).

Chaque algorithme s'exécute jusqu'à ce que le tableau de nœuds en attente soit vide ou que la solution (manoir propre) ait été trouvée. Dans cette dernière situation, nous remontons la branche solution jusqu'au départ afin de construire la séquence d'action finale du robot, constituant

son plan d'action. Une fois le plan d'action déterminé, le robot parcourt la séquence d'actions qu'il a en mémoire et les exécute une à une, mettant à jour le "vrai" manoir lorsqu'il y a des actions de collecte ou de nettoyage.

c) Explication de la mesure de performance et de l'apprentissage de la meilleure fréquence d'exploration

Lors d'une exploration de type **informée**, le robot va être capable d'apprendre au cours de l'exécution quelle est la meilleure fréquence d'exploration permettant de ne pas utiliser trop d'énergie, mais également de ne pas aspirer trop de bijoux par manque de connaissances de l'environnement. Pour cela, nous fixons une **limite maximale du nombre d'actions** que le robot peut effectuer dans un plan d'action (au lancement, cette limite est de **20 actions**). Après chaque exécution d'un plan d'action, nous allons récupérer la mesure de performance du robot sur ce plan, puis en fonction de sa valeur exacte, nous allons :

- **Diminuer** le nombre d'actions max (= augmentation de la fréquence d'exploration) si la performance est trop faible
- **Augmenter** le nombre d'actions max (= diminution de la fréquence d'exploration max) si la performance est trop élevée, car cela veut dire que le robot peut se permettre de scanner moins souvent
- **Conserver** le nombre d'actions max (= conservation de la fréquence d'exploration) si la performance est comprise entre les seuils fixés

Pour évaluer la mesure de performance du robot nous utilisons plusieurs choses :

- Un **ratio d'efficacité de nettoyage** : ce ratio permet de symboliser si le robot nettoie efficacement le manoir avec les actions choisies. La formule du calcul est $\frac{energieUtilisee}{nbPoussieresNettoyees + nbBijouxRamasses} + penalites$. Plus ce ratio est petit, plus cela signifie que le robot est efficace.
  Si le robot ne ramasse rien, on attribue une valeur arbitraire élevée qui représentera une mauvaise efficacité du robot. Cela aura pour conséquence que sa performance sera faible, et ainsi que sa fréquence d'exploration augmentera, permettant au robot de s'occuper des bijoux et de la poussière qui peuvent apparaître sur son chemin (car cette situation a des chances de se produire lorsque le robot est situé loin des éléments à aspirer).

- La **moyenne de la différence des valeurs d'heuristique entre les états "réels" et les états "internes" au robot**. C'est-à-dire que pour chaque état présent dans la solution que le robot a trouvé pour nettoyer le manoir, nous allons conserver leur valeur d'heuristique (dans *RobotPerformance* avec l'attribut *__statesHeuristicValues*), et ensuite lors de l'exécution des actions nous calculons la valeur d'heuristique de l'état correspond au monde réel actuel (donc en tenant compte des nouvelles poussières et des nouveaux bijoux que le robot n'a pas encore dans sa mémoire). Nous faisons ensuite, à chaque action exécutée, la différence entre la valeur "réelle" d'heuristique et la valeur calculée à partir de la mémoire du robot. Cette différence permet en fait de tenir compte de l'évolution du manoir lors de l'exécution du plan d'action du robot, et donc plus la différence augmente, plus cela signifie que la vision interne que le robot a du manoir s'éloigne du véritable état du manoir.

Avec les 2 valeurs décrites précédemment, nous calculons la mesure de performance via la formule suivante :

$$performance = \frac{100}{ratioEfficaciteNettoyage + 5 * differenceMoyenneDesHeuristiques}$$

(Note : le facteur 5 devant la différence moyenne des heuristiques a été ajouté afin d'apporter plus de poids à ce terme qui possède parfois une valeur très faible en comparaison du ratio d'efficacité) Cette formule nous donne alors une mesure **entre 0 et 100**, avec 100 étant la meilleure performance. A partir de cette mesure, nous adaptons (comme expliqué précédemment) le nombre maximal d'actions du robot pour un même plan d'action. Pour choisir les seuils permettant de faire évoluer ce nombre, nous avons observé la plage de valeur dans laquelle se trouve le plus souvent les mesures de performances et nous avons alors obtenu : **15** pour le seuil en dessous duquel il faut baisser le nombre d'actions max, et **30** pour le seuil au-dessus duquel il faut augmenter le nombre d'actions max.

d) Explication de la génération de poussières et bijoux

La génération de nouveaux éléments dans les pièces se fait au moyen de fonctions génératrices : une pour les poussières et une pour les bijoux. Ces fonctions prennent en argument la probabilité (allant de 0 à 100) que l'élément (poussière ou bijou) soit ajouté à la case. Elles parcourent toutes les pièces du manoir et, pour chacune d'elles, déterminent une valeur aléatoire. Si cette valeur est en dessous de la probabilité passée en argument, l'élément est ajouté dans la pièce, sinon rien ne se passe.