PROJET AI27 : HELLTAKER

Encadré par M Sylvain LAGRUE

Et M Khaled BELAHCENE

Réalisé par :

- Mostafa #####
- Tristan #####
- Edouard #####

ALLIANCE SORBONNE UNIVERSITÉ | utc

SOMMAIRE

1.1. Table des matières

2. Introduction

Helltaker est un jeu vidéo de type stratégie, centré sur la réflexion. Nous devons résoudre des casse-têtes tels que dans le jeu Sokoban.

L'objectif de ce projet est de créer différentes « IA » capables de résoudre ces puzzles. Pour cela nous avons donné une écriture du problème en STRIPS avant de l'implémenter en SATPLAN puis en ASPPLAN.

Nous proposerons à la fin un comparatif de ces 2 méthodes par rapport aux difficultés rencontrées et les résultats obtenus.

3. Préliminaires

3.1. Présentation des règles du jeu

Objectifs :

L'objectif de chaque puzzle est d'atteindre la fille démon, puis de répondre correctement à ses questions (sinon le joueur doit recommencer le niveau).

Déplacement sur la carte :

Les déplacements pour arriver à la fille sont limités par un compteur, il faut donc arriver avant la fin du nombre de déplacements possibles. Parfois il faudra ouvrir des cadenas en récupérant des clés en premier lieu. Les déplacements se font sur un quadrillage où peuvent être présents monstres, piques, rochers.

Rocher et monstre :

Les rochers ainsi que les monstres peuvent être déplacés par le joueur, lui coûtant 1 tour. Si le monstre se retrouve poussé dans un mur, il disparaît, le rocher ne bougera pas.

Piques :

Il faut savoir qu'à chaque déplacement les piques alternent entre l'état sorti et rentré. À l'exception de certaines piques qui restent toujours sorties. Quand le joueur se retrouve sur des piques sorties, il perd automatiquement un 1 tour de plus.

3.2. Le problème en STRIPS

Nous avons réalisé une représentation en STRIPS du problème, dans un premier temps nous initialisons tous les objets fluents (dynamique) et non fluents (statique). Tous les objets étant présents sur une carte comportant des cases, nous avons choisi un système de coordonnées en 2D. Parfois nous rajoutons un troisième argument quand l'état est susceptible de changer. De plus nous avons fixé un temps qui décrémente à chaque action. Ainsi tant que le temps n'a pas atteint 0, la partie peut continuer. Pour le

déplacement de certains objets, il fallait exprimer la notion de successeur pour situer la coordonnée limitrophe à une autre, et donc d'un objet proche d'un autre. Nous avons aussi implémenté une fonction clear pour supprimer un élément.

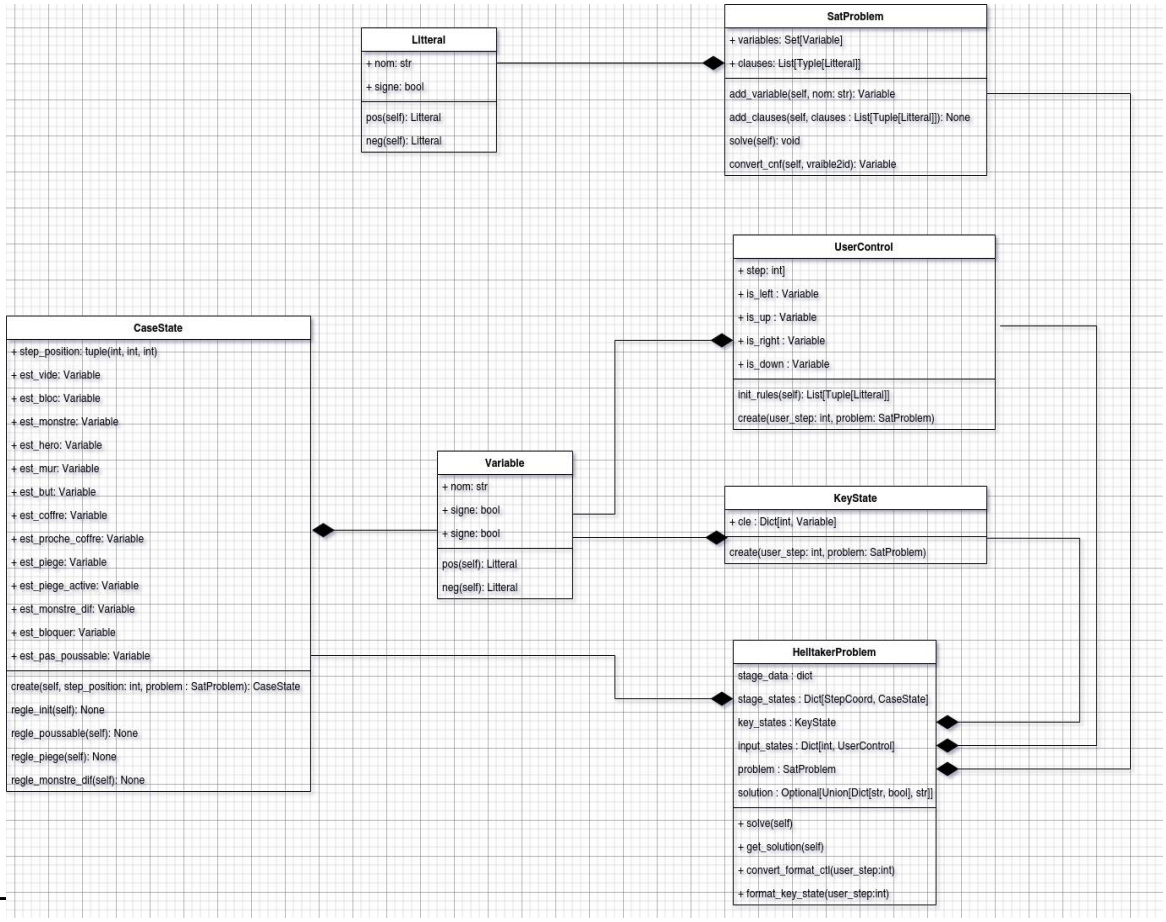
4. Méthode SATPLAN

4.1. Représentation du problème

Dans un premier temps, une modélisation a été faite en se basant sur des dictionnaires pour représenter les states et des booléens pour les variables. Il devenait de plus en plus compliqué de gérer ces dictionnaires quand bien même avec une documentation d'où l'élaboration d'une nouvelle représentation.

On a décidé de passer sous une modélisation objet permettant une meilleure gestion de toutes les states et variables. En ayant une couche d'abstraction grâce aux classes, il est nettement plus simple

Par exemple d'inverser des littéraux avec une méthode de classe ou attribuer des contraintes à une state.



4.2. Choix d'implémentation et structures de données

Les variables et les littéraux sont représentés par des classes implémenter les fonctions pos() et neg() renvoyant un littéral avec le signe correspondant. Tous les états sont des classes et possèdent un pas (horizon). À savoir les cases, les actions utilisateurs et les clés qui diffèrent au cours du temps.

Tous les states sont incluses dans l'objet HelltakerProblem centralisant tous les problèmes pour les transformer en CNF.

Pour représenter les cas possibles à chaque case, une structure crée se nommant StepCoord est un tuple associant le pas (ou horizon) et les coordonnées X, Y.

Ainsi un dictionnaire contenant en clef un StepCoord et en valeur tous les statuts associés à celui-ci permet d'avoir une modélisation du problème.

La fonction generate_problem va ajouter toutes les contraintes pour chaque catégorie d'entités, transformer en clause et ajouter les clauses dans l'objet SatProblem.

Quand toutes les contraintes ont été ajoutées au problème (via les fonctions add_x_rules), une table de correspondance est créée pour associer à chaque variable un nombre.

Puis convertis les clauses en nombres pour l'envoyer au solveur.

4.3. Expérimentations pratiques

Représenter tous les états possibles est coûteux en temps, à titre d'exemple le niveau 6 possède 51379 variables. De plus et surtout, utiliser gophersat implique d'écrire les clauses dans un fichier DIMACS.

Par conséquent, écrire 939388 clauses pour ce même niveau puis le lire avec le solveur donne des résultats extrêmement long et très dépendant des performances de l'ordinateur.

(Environ 2 min avec un Ryzen 7, 12 Go de ram, SSD)

Pour pallier à ce problème, il faudrait outrepasser la phase d'écriture / lecture. Cela est possible en utilisant un module python SAT à la place (voir ligne 319 de helltaker_plan.py). On se retrouve avec des temps nettement plus acceptables et autour de 10 secondes.

Niveau	1	2	3	4	5	6	7	8	9
Temps (en s)	4.45	4.08	7.50	4.48	4.48	15.92	7.84	6.43	9.73

Relève des temps de calcul de chaque niveau.

La méthode SAT est connue pour être très rapide dans l'exécution. Néanmoins, cela nécessite en contrepartie une modélisation importante et une quantité importante de code.

5. Méthode ASPPLAN

5.1. Représentation du problème

Pour représenter le problème en ASP, il a fallu que nous choisissions une méthode parmi 2 que nous avons ciblée :
La première correspond à modéliser la map/carte via les murs qui l'entourent. La seconde est de la modéliser par les cases sur lesquels le héros peut se déplacer. Nous avons décidé d'utiliser avec la seconde méthode. Dans la continuité de cette modélisation, nous avons défini tous les éléments sur la map comme les block, les mobs, etc.. selon le dictionnaire suivant :

- H: hero
- D: goal
- #: wall
- " ": case
- B: block
- K: key
- L: lock
- M: mob (skeleton)
- S: spikes
- O: spike and block

Ensuite, il a fallu définir toutes les actions qu'il est possible d'effectuer dans le jeu que l'on a stocké dans la liste direction.

Pour terminer la représentation du problème, nous avons défini par 'time' le nombre d'actions maximum à effectuer par le héros avant d'arriver son objectif.

5.2. Choix d'implémentation et structures de données

Par la suite, nous avons décidé de modéliser tous les éléments déplaçables par des fluents, ce qui nous permettra de les associer à leurs états à un instant T.
Puis nous avons généré et associé à chaque instant une direction possible.

Désormais, nous avons terminé de modéliser tout le problème. Il ne reste plus qu'à définir les règles de déplacement lors de chaque action. Par exemple, nous pouvons nous déplacer vers un couple (X,Y) que si une case existe à ces coordonnées.

Nous avons pu modéliser les maps avec des spikes, des clés et coffres, mais nous n'avons pas réussi à faire celle avec les spikes dynamiques.

Néanmoins, nous avons couvert un certain nombre de problème et permet donc de résoudre plusieurs niveaux.

5.3. Expérimentations pratiques

Dans les faits, l'exécution et la résolution des problèmes ne dépassent pas les 5 secondes. Ceci est tout à fait raisonnable. Ces tests ont été effectués avec un processeur Ryzen 7 3700X, 32Go de RAM.

Ces résultats sont tout à fait raisonnables, et si un utilisateur vient à utiliser ce programme, 5 secondes ne paraissent pas excessives.

Nous savons que des milliers de tests sont exécutés durant ces quelques secondes pour définir la solution. Néanmoins, cela nous parait assez élevé en termes de temps pour ce que c'est. Je pense que l'ASP est un langage très efficace, mais il doit très certainement exister d'autres langages permettant une résolution plus rapide et optimisée de problèmes similaires ou plus complexes.

6. Comparaison expérimentale des 2 méthodes

Les deux méthodes ont des temps de réponse assez compétitifs l'un envers l'autre. Néanmoins, nous pouvons effectuer la comparaison sur plusieurs points qui rend chaque programme gagnant dans certaines conditions :

- Temps d'exécution par rapport au nombre de requêtes :

Le programme SAT généré par python est très clairement vainqueur là-dessus, car concrètement, il génère plus de 300k clauses avant de l'envoyer au solveur SAT et de récupérer la réponse. Tandis que pour ASP, en manipulant un peu les conditions et en créant une boucle infinie, nous remarquons qu'il ne génère qu'environ 50k clauses en 3 secondes. Évidemment ces données ne sont pas produites sur les mêmes machines, mais la différence est assez flagrante pour désigner un vainqueur.

- Flexibilités et temps de réponse :

Nous avons vu que les deux programmes génèrent des réponses en quelques secondes. Mais le programme SAT lorsqu'il utilise le solveur GopherSAT il devient beaucoup plus lent et généré les réponses en 70 à 80 secondes ce qui est une différence considérable par rapport à l'ASP qui lui ne dépend que de la machine et dont le temps de réponse peut varier de 5 à 10 secondes.

Nous pouvons donc conclure en disant que ASP est très flexible et est la meilleure solution s'il doit être utilisé par des utilisateurs qui n'ont pas forcément les prérequis pour les solveurs, et à l'inverse, le SAT est la meilleure solution pour générer et chercher des solutions sur un problème complexe.