10/06/2022 Printemps 2022

COMPTE RENDU FINAL

UTC

UV IA02

- Projet HellTaker-

Sixtine Lorphelin et Elise Maistre

Table des matières

Préambule

Il faut tenir compte que dans ce projet, nous n'étions que deux à travailler dessus. En effet, la troisième personne du trinôme n'a pas pu participer au projet en raison de problèmes médicaux. Nous vous prions donc d'être plus indulgents envers notre travail en raison de cette pénalité qui nous a rendu le travail beaucoup plus long et difficile.

Introduction

Le jeu HellTaker est un jeu similaire en termes de fonctionnement du jeu Sokoban. Le but final est de trouver un chemin dans une carte avec des contraintes pour les déplacements. Notre but pour ce projet est de trouver LE chemin optimal, tout en prenant en compte ces contraintes. Il existe plusieurs méthodes que nous avons vues en cours pour déterminer ce chemin. La première méthode est la recherche d'état pour déterminer le meilleur chemin, la seconde est l'utilisation d'ASP et la troisième est en SAT. Il existe plusieurs commandes simples pour y arriver comme pousser une pierre ou se déplacer. Notre code utilise les deux premières méthodes. Dans un premier temps, nous verrons les règles détaillées du jeu pour mieux le comprendre et comprendre ses enjeux ainsi que sa conception en STRIPS. Nous verrons ensuite les deux méthodes et ce qu'elles font et enfin, nous verrons leur comparaison et la meilleure méthode à utiliser pour le jeu.

I) Présentation des règles du jeu

Le but de HellTaker est d'arriver sur une case proche de celle de la/des démone(s). Selon les niveaux, il est souvent demandé de récupérer une clé pour pouvoir ouvrir un trésor qui permet de passer avant d'arriver au but. cLes éléments suivants peuvent apparaître dans chaque niveau : robot/joueur, piques changeantes (actives ou rétractées), pique constante, trésor, zombies, clé, pierres, solutions (avec les cases contiguës à la case de la/des démone(s)) et démone(s). Il y a toujours des murs et les pierres sont très souvent présentes et en nombre très variable. Les coups/pas dans chaque niveau sont comptés et un nombre maximal de coups est déterminé à chaque niveau. Si le nombre de coups arrive à zéro, la partie est perdue et on doit recommencer. Certains pas ont plus de poids que d'autres. Par exemple, se déplacer simplement a un poids de 1. Si on se déplace sur une pique active, le poids est de 2 et notre variable "gagne" (variable pour le nombre de pas maximum), diminue donc de 2. Même s'il y a un compteur de pas, il n'y a en revanche pas de limite de temps. Ensuite, les actions possibles sont les suivantes: se déplacer, pousser une pierre, pousser un zombie, tuer un zombie, prendre la clé et ouvrir le coffre. Ses actions peuvent se faire dans les quatres directions : en haut, en bas, à gauche et à droite. Concernant la map, celle-ci est délimitée par des murs. Il existe des maps "statiques"

et d'autres dynamiques. Les statiques permettent de s'y déplacer dedans sans avoir de changement profond de la carte, même si les modifications sont prises en compte après avoir poussé une pierre, par exemple. Les maps dynamiques ont un terrain changeant, indépendamment des mouvements du joueur. On peut y voir des lasers ou une map qui bouge comme un tapis roulant etc. Nous allons dans ce projet prendre uniquement en compte les maps dites statiques.

II) Le problème en STRIPS

Le problème en STRIPS montre les prédicats, les fluents, l'état initial du jeu, le(s) but(s) et enfin, les actions. Le sujet sera en annexe au dossier du rendu final.
Remarques concernant le STRIPS et les limites du jeu:
-On peut pousser un zombie sur une pique ou un croc. Cette action va tuer le zombie pour toutes piques fixes. Pour les crocs, si on pousse au temps t1 et que les crocs sont rétractés, comme ils se rétractent pour un pas sur deux, au temps t2, les crocs seront actifs et le zombie qui sera poussé, mourra. Dans le cas contraire, le zombie ne mourra pas mais il mourra au temps t3 puisque les crocs s'activeront à nouveau.
-On peut pousser sur une clé un zombie et la clé ne disparaîtra pas. Il y a coexistence des deux éléments sur la même case.
-Nous n'avons pas les moyens de vérifier si l'on peut pousser une pierre sur une pique ou un croc. Nous supposerons que oui par logique car on peut voir à l'initialisation de certaines parties que des pierres sont posées sur des crocs.
Cf annexe

IV) Méthode Espace d'états

1. Représentation du problème

La méthode 1 utilisée ici est la recherche dans un espace d'état en langage Python. J'ai implémenté plusieurs types de recherche afin de trouver le chemin optimal jusqu'au but : recherche en largeur, recherche en profondeur et recherche A*. D'après les règles du jeu, il faut trouver un chemin en prenant en compte la variable qui compte les pas et en prenant aussi en compte parfois la prise de la clé.

2. Choix d'implémentation et structures de données

J'ai trouvé que l'algorithme de recherche en largeur était le plus efficace en parlant uniquement code. Aussi, l'algorithme de recherche A* est très utile pour prendre en compte le poids des pas. En effet, nous avions vu en cours l'algorithme A* avec pour nous aider dans une recherche optimale, la distance de Manhattan. Cette distance correspond à la distance en termes de cases entre la case de départ et celle courante, puis entre la case courante et celle du but. Or dans le jeu HellTaker, chaque pas a un

poids différent et donc non unifié. Par exemple, si je veux aller en haut et que la case en haut est vide, le poids de mon déplacement sera de 1. En revanche, si je veux me déplacer en haut et qu'il y a en haut des piques, mon déplacement sera affecté de 1 et donc le poids de mon déplacement sera donc de 1+1=2. Ainsi, pour optimiser les déplacements, je fais un premier algorithme de recherche entre la case de départ et la clé puis entre la case de la clé et une des cases solutions ou la case solution, s'il faut prendre la clé bien sûr. La recherche en largeur serait plus efficace dans le cas où la clé n'est pas obligatoire à prendre. J'ai donc décidé de faire l'algorithme A* en prenant le poids en compte au lieu de la distance de Manhattan.
Pour pouvoir résoudre ce problème, j'ai décidé d'utiliser un dictionnaire pour pouvoir faire les algorithmes de recherche. Ce dictionnaire a pour clés les différentes cases de la matrice et leurs valeurs correspondantes sont toutes les actions légales depuis cette case. Par exemple, si je veux aller en haut et qu'il y a une pierre au-dessus de moi et un mur au-dessus de la pierre, l'action "aller en haut" n'est pas légale. Elle n'apparaîtra donc pas dans le dictionnaire. Pour faciliter l'algorithme de recherche, j'ai également mis dans le dictionnaire les poids correspondants à chaque action faite à partir d'une case. Le dictionnaire aura donc la forme d'une liste de tuple pour les valeurs. Ex : {"01": [("h",1], ["g",2]}
Cet exemple se lit : "pour la case i=0 et j=1, l'action d'aller en haut a un poids de 1 et l'action d'aller à gauche a un poids de 2. Il s'agit en réalité d'une sorte de dictionnaire des voisins de la case considérée. La fonction correspondante est dico_voisins_satisfiables().
Attention, en fonction des déplacements, la matrice représentant l'état courant du jeu change, le dictionnaire des voisins changera donc aussi.

3. Expérimentations pratiques

Les fonctions fonctionnent indépendamment mais il y a un problème au niveau du code en python. Cela est sans doute lié à un problème de mise à jour régulière dans mes fonctions de la matrice de l'état courant. La fonction A* renvoie deux listes. Une avec les différentes actions à réaliser pour atteindre le but final sous la forme ["h","h","g","d"]. L'autre liste renvoie les cases sur lesquelles aller correspondants à ces actions sous le forme : ["56", "45", "78"] (c'est à dire i=5 et j=6 etc).
Ainsi, comme nous n'avions que peu de temps car nous n'étions que deux sur ce projet, nous vous demandons de bien vouloir lire la structure de nos codes et les comprendre.

IV) Méthode ASP

1. Représentation du problème

Afin de comprendre au mieux les termes utilisés pour décrire les composants, voici le dictionnaire qui permet la transcription du vocabulaire utilisé pour décrire le niveau vers le vocabulaire asp choisi.

```
dico={
    "H": ["init(robot(_))", "case(_)"],
    "D": ["init(fille(_))", "case(_)"],
    " ": ["case(_)"],
    "B": ["init(caisse(_))", "case(_)"],
    "K": ["init(cle(_))", "case(_)"],
    "L": ["init(tresor(_))", "case(_)"],
    "M": ["init(zombie(_))", "case(_)"],
    "S": ["init(croc(_))", "case(_)"],
    "U": ["init(crocimpair(_))", "case(_)"],
    "T": ["init(crocpair(_))", "case(_)"],
    "O": ["init(croc(_))", "init(caisse(_))", "case(_)"],
    "Q": ["init(crocimpair(_))", "init(caisse(_))", "case(_)"],
    "P": ["init(crocpair(_))", "init(caisse(_))", "case(_)"],
}
```

La représentation choisie pour cette méthode a été de considérer les cases de jeu contrairement au choix des murs. En effet, chacune des cases de jeu (cases qui ne sont pas des murs) sont des cases où le robot pourra potentiellement aller. Ainsi la condition nécessaire pour un déplacement ou une condition sera la présence d'une case à cet endroit. Les cases sont donc implémentées suivant le dictionnaire ci-dessus. Comme on peut le voir les murs n'y sont pas représentés alors chacun des caractères correspondent à une case.

Par ailleurs, les autres éléments du jeu sont initialisés par un init(), qui sont les fluents du jeu et seront donc transformés en fluents, gardés ou supprimés selon les actions réalisés et souvent pris en compte pour ces mêmes actions.

```
fluent(F, 0) :- init(F).
```

Le niveau est représenté comme une matrice en partant d'en haut à gauche à (0,0). L'identification par ligne et par colonne est ensuite réalisée en parcourant le fichier du niveau.

```
etape(0..n-1).
action(h;b;d;g;nop;p).
```

Pour paramétrer le jeu nous avons ensuite besoin d'étapes qui seront la suite d'action, au total il est possible de réaliser jusqu'à n-1 étapes, n correspondant au palier maximal donné pour gagner.

Les différentes actions possibles sont h pour haut, b pour bas, d pour droite, g pour gauche, 2 actions supplémentaires ont été rajoutées qui sont nop : lorsque l'on termine un niveau, les autres actions ne sont plus réalisables, on ne peut alors que faire nop et p : qui correspond à un moyen d'attendre pour le robot lorsqu'il se fait taper par un pic.

```
{ do(Act, T): action(Act) } = 1 :- etape(T).
```

On utilise alors un générateur d'action pour avoir une action à chaque étape.

Finalement, il existe un objectif particulier à atteindre, celui de se retrouver sur une case adjacente à la fille. Le but est accompli si le robot se trouve sur la case à coté de

la case où se situe la fille. L'objectif est de finir : on ne peut pas ne pas finir, on ne peut pas finir dans un temps supérieur à n soit à la limite fixée par le jeu, on ne peut pas avoir fini et faire autre chose que nop en action et on ne peut pas faire nop et ne pas avoir fini.

```
fluent(F, T + 1) :-
    fluent(F, T),
    T + 1 < n,
    not supprime(F, T).
```

Pour finir, la manière dont tourne le jeu dépends de suppressions réalisées au cours des étapes, si au temps T un fluent est supprimé, avec supprime(), il ne sera pas de nouveau présent au temps T+1. En effet, si un fluent était là au temps T, n'a pas été supprimé et l'on se trouve toujours en cours de partie le fluent sera là au temps T+1. Le cas est similaire lorsque l'on a atteint un but, seulement plus rien n'est alors supprimé.

2. Choix d'implémentation et structures de données

Afin de réaliser chacun des niveaux la méthode réalisée a été de modifier chacun des cas étapes par étapes jusqu'à arriver au point final.
Tout d'abord j'ai implémenté les différentes actions haut, bas, gauche et droite en modifiant à chaque fois les coordonnées. Nous nous focaliserons donc sur le cas gauche qui est le premier réalisé.

- Les préconditions

Il n'est pas possible d'aller à gauche s'il n'y a pas de case à gauche
Il n'est pas possible d'aller à gauche si à gauche il y a un trésor mais que la clé n'a pas été trouvée.

- Les effets

Beaucoup d'effets ont été implémentés qui pourraient être remplacés par des préconditions mais il a été plus simple de représenter ainsi le problème pour moi.

a) Action du robot

Tout d'abord, le robot va à gauche, si le robot est bien sur la case, qu'il réalise l'action et qu'à gauche il n'y a rien qui obstrue le passage (comme une caisse, un zombie ou une fille, le cas du trésor ayant été ajouté dans la précondition, tout comme les murs ayant été remplacé par la présence de case), dans ce cas là le robot va à gauche à la prochaine étape et est supprimé du lieu où il était.

b) Pousser une caisse

Il est possible de pousser une caisse, pour cela à gauche du robot doit se trouver une caisse, il doit vouloir aller à gauche et rien ne doit obstruer la future place de la caisse, aucun des fluents caisse, fille, zombie et trésor ne doivent s'y trouver, de plus il doit y avoir une case pour accueillir cette caisse (pas de mur à son

emplacement). Dans ce cas on supprime la caisse de son ancien emplacement au temps T et on ajoute une nouvelle caisse, deux pas à gauche à la prochaine étape.

c) Cas des zombies

Les zombies ressemblent sur un certain point à des caisses avec la condition supplémentaire qu'ils peuvent être réduits en poussière s'ils sont cognés contre un objet comme une fille, un autre zombie, une caisse, un trésor ou encore un mur, soit l'absence de case. Le cas de pousser un zombie est le même que la caisse ci-dessus. Par contre pour supprimer un zombie lorsqu'il se cogne contre autre chose a été plus difficile et j'ai implémenté chacune des conditions. Le zombie disparait s'il est à gauche du robot, que l'action gauche est réalisée et qu'à gauche de lui il y a un fluent (liste ci-dessus), ou encore pas de case.

Ensuite est venu le moment d'implémenter les cas plus difficiles, qui m'ont posé plus de problèmes.

a) Le trésor

Le trésor est tout d'abord présent dans les effets comme un objet bloquant (voir ci-dessus). Pour débloquer le trésor il est nécessaire que le personnage passe par une clef. Il fallait donc implémenter trouve clef qui est réalisé lorsque le robot et la clé se trouvent au même moment au même endroit. Si la clef est trouvée alors comme on l'a vu dans les préconditions de déplacements on ne peut se déplacer là où il y a un trésor sans que la clef n'ait été trouvée (trouve(clef) activé). Ensuite, il ne reste plus qu'a supprimer le trésor et la clé lorsque le robot se situe aux mêmes coordonnées à la même étape.

b) Les pics/crocs

Il existe deux types de crocs, les crocs stables au cours du temps, c'est-à-dire fixe et les crocs alternants. J'ai commencé par les crocs fixes, trouvé une solution qui fonctionnait pour les crocs fixes, que l'on a sur le côté. Cette méthode considère que le robot reste à sa place au prochain temps s'il se situe au même emplacement qu'un croc à cet instant et qu'on réalise l'action p, l'action signifiant attendre. Dans ce cas, il fallait empêcher une boucle infinie d'apparaître, pour cela on a l'impossibilité de faire deux fois de suite l'action p lorsque le robot se trouve sur le croc, ainsi une fois que l'action p a été réalisée, elle ne le sera pas au temps suivant.

```
%action pic non variable
fluent(robot(X,Y),T+1) :-
    do(p, T),
    fluent(robot(X,Y),T),
    fluent(croc(X,Y),T).

:-  fluent(robot(X,Y),T),
    fluent(croc(X,Y),T),
    not do(p,T-1),
    not do(p,T).

supprime(zombie(X,Y),T):-
    fluent(zombie(X,Y),T),
    fluent(croc(X,Y),T).
```

Le cas de suppression d'un zombie se réalise quand il se situe sur des crocs.

```
%action pic variables
%impairs

fluent(crocimpair(X,Y),T+1):-
    fluent(crocpair(X,Y),T),
    T+1<n,
    not do(p,T).
supprime(crocimpair(X,Y),T):-
    fluent(crocimpair(X,Y),T),
    not do(p,T).

:-  fluent(robot(X,Y),T),
    fluent(crocimpair(X,Y),T),
    not do(p,T-1),
    not do(p,T).

supprime(zombie(X,Y),T):-
    fluent(zombie(X,Y),T),
    fluent(crocimpair(X,Y),T).


%pairs
fluent(crocpair(X,Y),T+1):-
    fluent(crocimpair(X,Y),T),
    T+1<n,
    not do(p,T).
supprime(crocpair(X,Y),T):-
    fluent(crocpair(X,Y),T),
    not do(p,T).
```

J'ai ensuite tenté d'implémenter cette version pour les crocs variables, or les résultats donnés étaient les bons si on ne regardait pas comment fonctionnait p, mais p ne fonctionnait alors pas normalement. On voit à gauche l'alternance des crocimpair et crocpair au cours du temps. Cette alternance étant réalisée au temps suivant si au temps précédent on avait l'inverse et si le temps global n'était pas dépassé. Par ailleurs le not do(p,T) correspond au fait que l'on ne fait pas cette alternance si l'action p est réalisée, en effet, alors les pics ne tournent plus. On fait alors de même pour les crocpair. Et également pour les zombies.
Mais cette méthode fonctionnait d'une manière originale au niveau des p qui apparaissaient lorsque le robot ne se prenait pas de pics.

```
fluent(crocimpair(X,Y),T+1):-
    fluent(crocimpair(X,Y),T),
    T+1<n, do(p,T).
```

J'ai donc rajouté plusieurs indications, tout d'abord j'ai forcé la condition de garder le même fluent lorsque l'on réalise l'action p, ce qui a permis d'enlever ces incohérences d'ajout de p que je suppose qu'il utilisait pour palier à cette variation lors de passages sur un p précédent (ex au niveau 5).

```
supprime(zombie(X,Y-1),T):-
    fluent(zombie(X,Y-1),T),
    fluent(robot(X,Y),T),
    do(g,T),
    fluent(crocimpair(X,Y-2),T), case(X,Y-2).
supprime(zombie(X,Y-2),T):-
    fluent(zombie(X,Y-1),T),
    fluent(robot(X,Y),T),
    do(g,T),
    fluent(croc(X,Y-2),T), case(X,Y-2).
```

Ensuite j'ai modifié la suppression des zombies en ajoutant le cas des crocs et des crocimpair lors de la suppression de zombie, à la fois dans le cas du déplacement du zombie lorsqu'il n'y a rien derrière pour pouvoir l'implémenter à ma manière ensuite. J'ai donc préparé une suppression en amont afin de ne pas avoir un problème de décalage comme j'avais dans une autre méthode (voir 3)).
Cette méthode suppose un zombie à gauche du robot et des crocs coupants donc soit impair ou croc à sa gauche, le robot va à gauche ainsi le zombie sera supprimé au prochain temps et on a pas de décalage.
Il faut également bien remarquer qu'il a été décidé que les crocs levés correspondaient aux crocs impairs.

3. Expérimentations pratiques

Plusieurs difficultés se sont posées au cours du projet, pour les résoudre le langage asp n'est pas le plus simple, mais quelque chose de pratique se trouve être l'affichage des fluents pour mieux comprendre ce que fait le programme. Mais pour avoir un affichage il faut tout de même trouver un cas où le programme propose un modèle,

soit qu'il est satisfiable. Pour cela en augmentant n certaine fois on pouvait avoir un résultat.
Les points qui m'ont posés problème sont tout d'abord les buts, en effet il m'a paru difficile de les implémenter de cette manière. Je voulais utiliser la négation, les choses impossibles pour justifier les buts, comme le fait qu'il ne fallait pas résoudre tous les buts mais un seul.
Ensuite les zombies ont été un peu difficiles également car je voulais faire tous les cas en un seul et qu'utiliser la précondition me semblait difficile, mais en comprenant mieux la logique d'asp j'y suis parvenue, il suffisait de faire tous les cas.
La clef m'a également posé quelques problèmes, je n'arrivait pas à imaginer comment implémenter un booléen en asp, mais finalement ce n'était pas si compliqué il suffisait de créer un nouveau prédicat trouve().

```
%action pic
%impairs
fluent(crocimpair(X,Y),T+1):-
    fluent(crocpair(X,Y),T),
    T+1<n, not do(p,T).
supprime(crocimpair(X,Y),T):-
    fluent(crocimpair(X,Y),T), not do(p,T).
fluent(crocimpair(X,Y),T+1):-
    fluent(crocimpair(X,Y),T),
    T+1<n, do(p,T).

douleur(T):-fluent(robot(X,Y),T),fluent(crocimpair(X,Y),T).
douleur(T):-fluent(robot(X,Y),T),fluent(croc(X,Y),T).

do(p,T+1) :- douleur(T).

%pairs
fluent(crocpair(X,Y),T+1):-
    fluent(crocimpair(X,Y),T),
    T+1<n, not do(p,T).
supprime(crocpair(X,Y),T):-
    fluent(crocpair(X,Y),T), not do(p,T).
fluent(crocpair(X,Y),T+1):-
    fluent(crocpair(X,Y),T),
    T+1<n, do(p,T).
```

La dernière difficulté se trouve être les pics, qui m'ont posé beaucoup de problèmes, comme on peut le voir en 2).
J'ai également trouvé une solution différente qui ne fonctionne que lors de cas de pics variables, soit pas dans les niveau 2 et 3.
Les actions sont modifiées par un prédicat douleur(T) qui est réalisé si le robot se trouve au même endroit qu'un crocimpair au temps T, dans ce cas là l'action p est réalisé au temps suivant. Dans ce cas alors un décalage avait lieux, ce qui n'était pas optimal. La version finale fonctionne dans tous les cas et n'a plus ce problème de version. Tout de même cette version était plus rapide, lors du solving.

V) Comparaison expérimentale des deux méthodes

	Level 1	Level 2	Level 3	Level 4	Level 5	Level6	Level7	Level 8	Level 9
Solving	0.02s	0.04s	0.62s	0.06s	0.04s	4,29s	9.61s	0.00s	1.72s
Time	1.472 s	1.581 s	4.373 s	1.547 s	1.440 s	15,359 s	13.668 s	0.261 s	6.448 s

Tableau correspondant au temps de résolution des niveaux sur asp, on peut voir que le temps est beaucoup plus long que la méthode python qui avec l'algorithme de recherche A*, on trouve un temps égal à 0.019855976104736328 secondes.
Il y a une nette variation entre les différents niveaux, dépedant du nombre d'étapes ou encore de la difficulté de la carte.

On peut également observer que lorsqu'il y a un choix à faire le programme asp passera par la clé et le trésor comme on peut l'observer dans le niveau 4, alors qu'une autre méthode est également possible.