IA02

Rapport de projet

Solveur de niveaux du jeu Helltaker

Introduction

Dans le cadre de l'UV IA02 : Logique et Résolution de problèmes par la recherche, nous avons développé plusieurs méthodes permettant de résoudre les puzzles logique du jeu Helltaker à l'aide des outils d'intelligence artificielle découverts tout au long du semestre P22. Ce rapport vise à expliquer nos différentes méthodes et choix d'implémentation afin de les comparer

Notre groupe est composé de Adrien Simon (GI01), Léo Perron (GI02) et Julie Pichon (GI02) .

Sommaire

Introduction 2

Sommaire 2

1. Préliminaires 3

    Présentation des règles du jeu 3

    Le problème en STRIPS 3

    Les différences majeures avec le sokoban 4

2. Recherche dans un espace d'état (Python) 5

    Représentation du problème 5

    Choix d'implémentation et structures de données 5

    Expérimentations pratiques 6

3. ASP PLAN (réécriture en ASP du problème de planification) 7

    Représentation du problème 7

    Choix d'implémentation et structures de données 7

    Expérimentations pratiques 8

Comparaison expérimentale des 2 méthodes 8

Annexes 9

1. Préliminaires

Présentation des règles du jeu

Helltaker est un jeu indépendant freeware développé par Łukasz Piskorz en 2020. Le jeu se compose d'une série de puzzle et de questions d'une simulation de drague.

Synopsis :

Un homme veut atteindre des femmes démons pour qu'elle rejoigne son harem, pour cela, le personnage est placé à chaque niveau sur un espace avec des blocs, des squelettes, des pièges… Il doit atteindre le démon en exécutant un nombre d'action limité. Nous ne nous intéresserons pas dans ce projet aux questions de drague qui se déroulent à la fin de chaque niveau.

Les règles du jeu sont simples, chaque niveau possède un nombre de mouvements maximum, le joueur doit déplacer son personnage dans la carte en poussant les blocs, squelettes etc pour se trouver dans une des cases adjacentes au démon. dans certains niveaux, le démon se cache derrière un cadena, le joueur doit donc aussi récupérer une clé sur la carte avant d'atteindre le démon.

- R1 : on ne peut que pousser les objets (on ne peut pas tirer)
- R2 : on ne peut pas passer au dessus d'un objet, ni le traverser
- R3 : une action coûte au moins 1 (au nombre de mouvement total
- R4 : On peut tuer un squelette en le poussant sur un obstacle (de type bloc, mur, porte …)
- R5 : on ne peut pousser qu'un élément a chaque fois (si un obstacle se trouve derrière on perd simplement une action mais la carte ne change pas)
- R6 : il faut une clé pour ouvrir une porte
- R7 : marcher sur un piège fait perdre 2 unités de temps

Le problème en STRIPS

Prédicats

- *empty/2* : prédicat qui décrit toute les cases valide du plateau (ou le personnage peut aller)
- *demoness/2* : décrit la position du démon

Fluents

- man/2 : la position du personnage principal durant la partie
- block/2 :position d'un bloc
- skeleton/2 : position d'un squelette

- trap/2 : position d'un piège (noton que nous ne ferons pas la différence entre les piège actif et inactif dans ce strips pour simplifier la visualisation du problème)
- lock/2 : position de la porte
- key/2 : position de la clé
- time/1 : nombre d'action autorisée au cours de la partie
- haskey/1 : booléen qui décrit si le personnage possède la clé ou non

Remarques :

Notons que la porte et la clé ne se déplace pas réellement de case, mais nous avons tout de même décidé d'en faire des fluent car elles peuvent disparaître du plateau

Fonctions utiles : *next/2* : next(x,x') => x = x'+ 1

Etat initial :

- *is(X,Y)* : position du personnage initiale
- *demoness(...)* : position du demon

Objectif :

Se situer sur une case adjacente au démon avec t >= 0 :

time(t) ∧ t>=0 ∧ man(x,y) ∧ (demoness(x',y) ∧ next(x, x'))

Actions :

On peut noter différentes actions comme avancer d'une case (en bas, haut,...), pousser un squelette, pousser un bloc, récupérer une clef …

Les actions sont detaillées dans le ficher annexe "helltaker_strips.md"

Les différences majeures avec le sokoban

a rediger mieux (julie)

Helltaker est un problème comparable au sokoban, certaines règles sont identique et Helltaker est en fait dérivé des problèmes comme le sokoban. Cependant les deux jeu présentent des différences majeures.

Tout d'abord les objectifs sont différents : dans le sokoban l'objectif est de pousser les caisse sur des destinations alors que pour helltaker il s'agit de se rendre à une position (ou plusieurs en cas de clé)

L'environnement est aussi différent car des squelettes et des blocs se situent sur la carte. Il y a donc plus d'actions différentes dans helltaker : pousser un squelette, récupérer une clé, tuer un squelette …

Le fait de pousser un objet est considéré comme un tour a part entière, le personnage ne se déplace pas en poussant dans helltaker, il faut pousser puis avancer.

Un certain type de notions ne sont pas présent dans le sokoban : les pièges qui font perdre 2 vies, la disparition des portes et de la clé sur le plateau…

Globalement la modélisation de Helltaker est plus complexe mais se base sur les mêmes principes de base.

2. Recherche dans un espace d'état (Python)

Représentation du problème

Nous avons séparé les états selon deux catégories vues précédemment, les non-fluents (que nous appellerons map-rules) et les fluents (que nous appellerons state). Les non fluents sont: les murs, les spikes et les démones.

Pour la représentation d'un state, nous avons décidé d'émettre l'hypothèse qu'il ne peut y avoir qu'un seul lock et key par niveau (ce n'est pas fondamentalement différent des autres états, ça nous permet surtout de faciliter de représenter l'ouverture d'un lock).

Concernant la représentation des actions, nous les avons divisés selon 4 verbes: move, push, kill, unlock/key. Selon nous, les différentes réactions dues aux mouvements ou aux push (bouger sur un spike, bouger une caisse ou un monstre) peuvent se gérer assez facilement dans l'actualisation d'un state.

Afin de résoudre les différents plans, nous avons tout d'abord réalisé un BFS puis un DFS. Par la suite, quand nous avons réussi à résoudre tous les niveaux, nous avons décidé d'implémenter un algorithme A*.

Concernant notre heuristique, nous avons tout d'abord décidé d'inclure une distance manhattan (non pas euclidienne car on travaille avec une grille). Concernant les autres éléments pouvant influer, nous avons réfléchi à faire une heuristique prenant en compte la distance avec la clé ou lock si nécessaire; En outre, ce choix semble pertinent puisque les niveaux suivent pour la majorité des cas la logique de se rapprocher de la clé puis du lock, il serait donc dommage de de ne pas utiliser ces connaissances afin d'optimiser l'heuristique. De plus, l'action "unlock/key" est également valorisée (qui ont été trouvé à tatillon, il aurait été sûrement plus pertinent de faire un algo génétique)

Choix d'implémentation et structures de données

Concernant les structures de données utilisées, nous avons décidé de reprendre la structure NamedTuple vu en TP (pour les states et les actions), elle donne l'avantage d'être hashable et permet d'accéder aux données plus facilement. Pour les ensembles d'éléments (*exemple: monstres*), nous avons décidé d'utiliser des FrozenSet car ils sont également hashable (nous avons cependant donc dû créer une méthode permettant leur modification à travers la création d'une copie).

Afin de mettre à jour un state, nous avons créé une fonction update_state qui permet de créer une copie de state qui swap automatiquement les traps, actualise les steps restants en fonction de présence de spike/trap ou non, kill les mobs sur les spikes et interagit avec le lock ou la key si nécessaire. En outre, si le state est impossible (Game Over), il ne renvoie aucun state. Actualiser le state de manière générique était un point particulièrement important pour nous puisqu'il permet de rendre le code plus flexible dans la résolution de nouveaux plans.

Pour la génération des actions, nous générons d'abord l'ensemble des actions possibles à l'aide des verboses et des directions. Ensuite, une fonction do_inplace s'occupe de vérifier la possibilité de l'action ou non; le cas échéant, il génère un nouveau state avec si besoin, les informations à modifier (*exemple: Position du héro, position des monstres…)*

Afin de faire tourner l'algorithme de recherche, nous avons dû coder une fonction successeurs qui permet de renvoyer tous les states successeurs au state passée en paramètre. Elle ne vérifie pas la possibilité du state ou non puisque nous avons déjà la fonction do_inplace conjuguée à la fonction update_state qui s'en occupent. Cette fonction renvoie un dictionnaire avec en clé le state "fils" et l'action correspondante.

Concernant les différents algorithmes de recherche, nous avons repris celui étudié en TP pour les explorations en largeur/profondeur en codant les fonctions FIFO (les fonctions LIFO étant déjà disponibles). Pour l'exploration heuristique (Glouton ou A*), nous avons décidé d'utiliser la librairie heapq permettant d'instancier des tas min/max (ce choix de structure de données nous semblait pertinent au vu de la nécessité de trier tous les successeurs à un state). Le reste de l'algorithme est très similaire aux BFS/DFS puisqu'il est basé sur un BFS avec ajout d'heuristique.

Expérimentations pratiques

Ci-dessous un tableau répertoriant tous les temps réalisés pour chaque niveau. Pour le calcul du temps, nous avons fait la différence entre 2 variables étant instanciés à time.time() avant résolution et après résolution. (À noter que les temps ont été réalisés sur un ordinateur portable en charge, *processeur*: Intel Core i5 double cœur - 3,1 GHz)

Niveau	1	2	3	4	5	6	7	8	9
Temps	0.25s	0.03s	0.4s	0.4s	0.45s	7.2s	3.0s	0.01s	3,7s

Tableau des temps en fonction du niveau (originellement en ms)

En ayant comparé nos résultats avec des camarades, nous avons pu constater que les runs des premiers niveaux sont en moyenne beaucoup plus longs que les leurs mais que cependant la résolution est similaire sur les niveaux complexes (6 et 9). Notre heuristique étant similaire, nous en avons conclu que la différence principale se situait dans l'utilisation des structures de données.

On peut donc en émettre l'hypothèse que notre structure de données semble inappropriée à la résolution de plans simples (il y a sûrement un passage en non hashable à un moment que l'on a pas trouvé). En outre, l'heuristique semble bonne même si nous pensons qu'elle n'est pas optimale (il y a sûrement des améliorations à en tirer sur par exemple l'évaluation de l'action en tant que tel, le nombre de blocs ayant bougés..)

3. ASP PLAN (réécriture en ASP du problème de planification)

Représentation du problème

Concernant l'ASP, nous avons fait les choix suivants :

- Les actions ont été séparées en 14 : à savoir right, left, up, down, pour les 4 directions, un push_dir dans chaque direction pour les boxes, et un monster_dir dans chaque direction aussi pour les monstres. Nous avons aussi l'action on_spike, qui nous permet de "gâcher" un coup en étant sur ces derniers, et l'action nop si nous sommes arrivés avant la fin du nombre de coups impartis.

Les hypothèses concernant les coffres et clés uniques du python sont reprises dans l'ASP.

Les fluents et non-fluents sont représentés de la même façon que dans le TP Sokorridor.

Choix d'implémentation et structures de données

A propos du TP Sokorridor, nous avons repris les choix d'implémentations faits à propos des buts, et de la fin du niveau grâce au prédicat achieved, des fluents instanciés au temps T0 grâce à un prédicat init, les prédicats du Frame Problem, et le générateur d'action

A propos des choix d'implémentations personnels qui ont été faits, voici les plus importants.

Pour la gestion des coffres et des clés, nous avons 3 prédicats fluents. Un prédicat chest(X, Y) qui donne la position du coffre, un key(X, Y), qui donne celui de la clé, et un has_key(n), avec n pouvant varier de 0 ou 1, en fonction de si le joueur ramasse une clé ou non.

Pour les traps qui s'activent et se désactivent, la solution utilisée n'est peut-être pas la plus optimale, mais elle fonctionne. En effet nous utilisons un generate & test pour chaque trap de la façon suivante :

```
{trap(I, J, S, T) : state_trap(S)} = 1 :- step(T). :- trap(I, J, S, 0), S != on.
```

Ceci permet deux choses : d'initialiser la trap à l'état "on", et de générer tous les états de la trap pour tout T. Nous avons un deuxième test qui permet de switcher les traps à chaque step comme dans le ping-pong.

Expérimentations pratiques

Le tableau ci-dessous recense les temps d'exécution pour trouver un modèle valide pour chaque niveau. (Les temps ont été réalisés sur un ordinateur portable avec un processeur Apple M1).

Niveau	1	2	3	4	5	6	7	8	9
Temps	0.08s	0.13s	1.11s	0.25s	0.32s	4.27s	3.77s	0.06s	2.96s

Tableau des temps en fonction du niveau (originellement en ms)

Ces temps sont raisonnables pour une résolution en ASP, et les choix faits ont l'air d'être cohérents. Quelques choses pourraient sûrement encore être optimisées, cependant nous n'avons pas eu le temps de faire plus de tests afin d'avoir un programme le plus optimisé possible.

4. Comparaison expérimentale des 2 méthodes

On observe des résultats comparés assez différents entre les deux méthodes implémentées.

En effet, l'ASP semble plus efficace pour les niveaux avec beaucoup de déplacements, tels que les niveaux 6 et 9, et le Python semble plus efficace sur les plus petits niveaux, la différence est flagrante sur les niveaux 2 et 3.