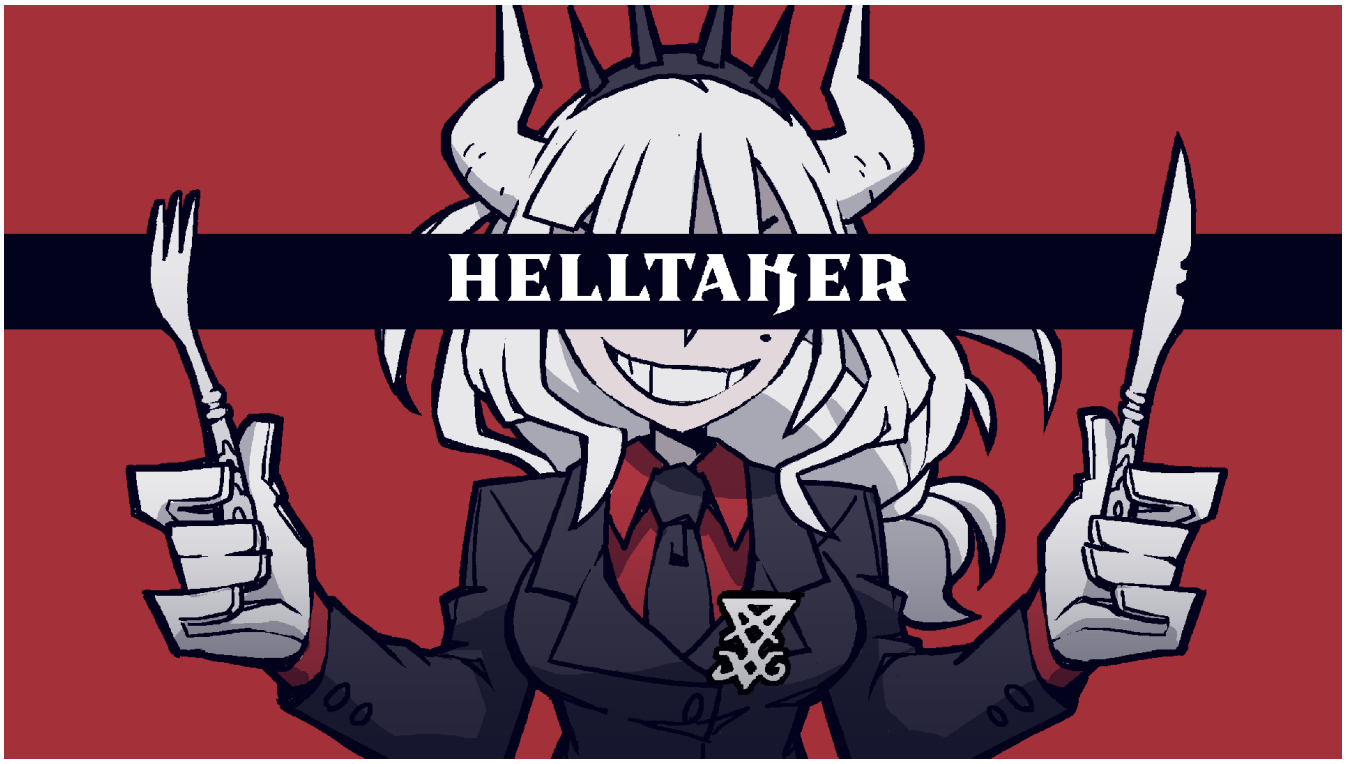
Rapport AI27 - Helltaker



Pierre Gibertini - Anne-Soline Guilbert–Ly - Romane Dauge

Table des matières

1. Introduction

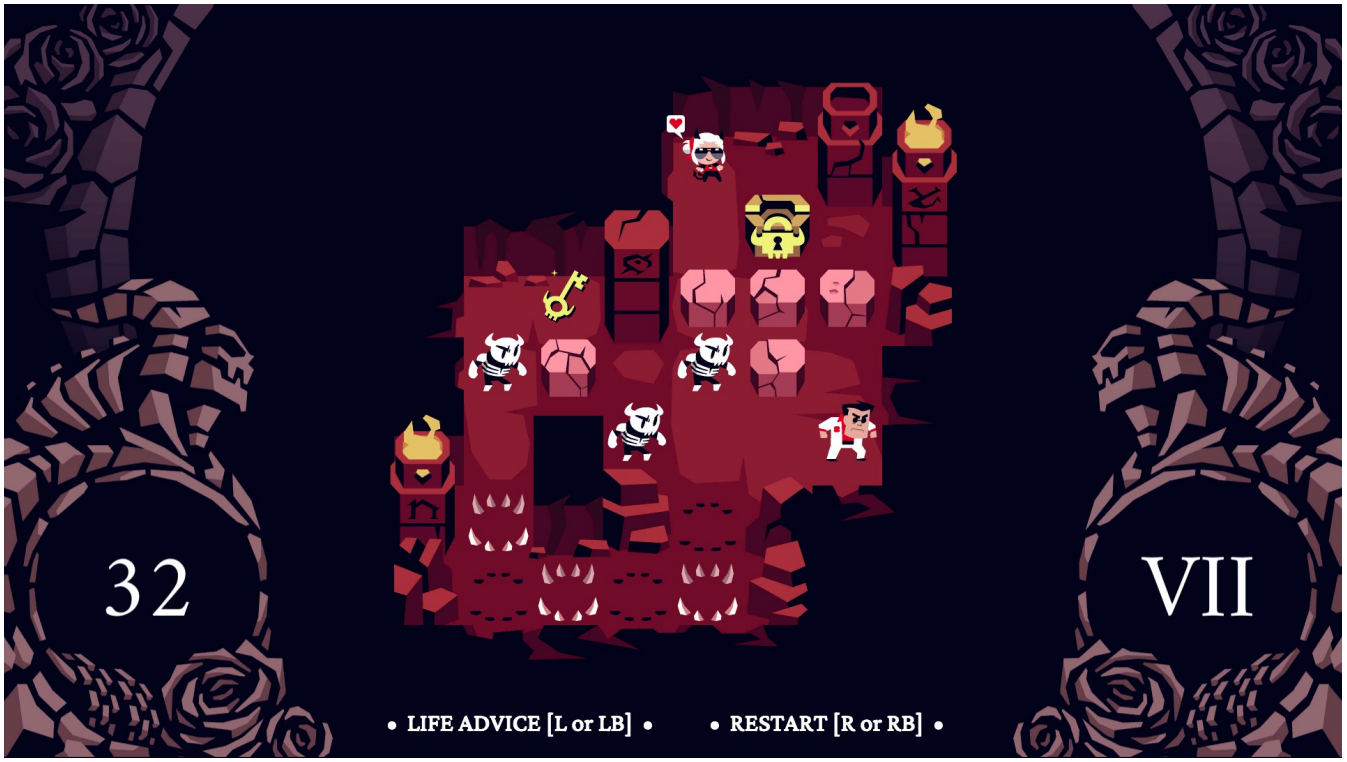
Ce rapport a pour but d'expliquer et de résumer notre projet de AI27. Nous avons adopté deux méthodes pour résoudre les problèmes du jeu Helltaker : SATPLAN et ASPPLAN.

Helltaker est un jeu où un héros cherche à se construire un harem de femmes démons. Pour cela, les joueurs doivent réussir à déplacer le héros à travers différents puzzles semés de monstres et de pièges en tout genre.

2. Préliminaires

2.1 Présentation des règles du jeu

Le but du jeu est de résoudre des labyrinthes dans lesquels le héros se déplace sur une grille case par case. Le nombre de mouvements est limité, il y a pour chaque niveau le nombre juste de déplacement nécessaire pour rejoindre les femmes démons, qui marque la réussite du niveau.



Sur son chemin, le héros rencontre plusieurs obstacles qui rendent le jeu plus complexe et stimulant.

Les monstres

Sur le passage du héros se trouvent des monstres. Le héros peut pousser les monstres et peut les tuer dans certaines conditions. Cela lui coûte des déplacements mais est souvent nécessaire pour rejoindre les femmes démons.

Les pics

Sur certaines cases du puzzle se trouvent des pics. Il en existe deux types, des pics ouverts tout le temps et des pics qui se ferment ou s'ouvrent. Si le héros se déplace sur une case avec un pic ouvert, il perd un déplacement.

Les blocs

Des blocs de murs se trouvent sur la carte et gênent le passage du héro. Il peut les déplacer pour libérer le passage.

Les portes et les clés

Dans certains niveaux, on trouve sur la carte des clés et des portes. Pour franchir la porte, le héros a besoin de récupérer la clé.

Si le héros ne rejoint pas les femmes démons avant la fin du nombre de déplacement, la partie est perdue ! Et il faut recommencer jusqu'à trouver la combinaison parfaite.

2.2 Le problème en STRIPS

Afin de toujours trouver la combinaison de déplacement parfaite, quoi de mieux que de modéliser toutes les règles du jeu pour demander à une IA de résoudre ces casse-têtes à notre place ?

Pour modéliser ce jeu, nous avons réalisé une analyse détaillée du problème en STRIPS.

Prédicats

Voici la liste des prédicats :

```
at(X,Y)
```

> Représente la position du héro.

```
wall(X,Y)
```

> Représente la position d'un mur.

```
block(X,Y)
```

> Représente la position d'un bloc.

`lock(X,Y)`

> Représente la position d'une porte fermée.

`mob(X,Y)`

> Représente la position d'un monstre.

`spikes(X,Y)`

> Représente la position d'un pic
>
> > **Remarque :** nous n'avons pas fait la différence entre les pics qui s'ouvrent et se ferment, et les pics statiques dans la représentation en STRIPS.

`key(X,Y)`

> Représente la position d'une clé.

`demoness(X,Y)`

> Représente la position d'une démonne.

`cell(X,Y)`

> Représente la position d'une cellule vide.

Etat initiaux

Voici plusieurs états initiaux possibles :

```
cell(1, 1)
cell(2, 1)
cell(3, 1)
cell(4, 1)
cell(5, 1)
cell(6, 1)
demoness(6, 1)
spike(4, 1)
at(2, 1)
```

```
cell(1, 1)
cell(2, 1)
cell(3, 1)
cell(4, 1)
cell(5, 1)
cell(6, 1)
demoness(6, 1)
at(2, 1)
```

But

Pour gagner le joueur doit se trouver sur une case à côté d'une démonne. Pour cela, il y a 4 buts possibles (être à droite, à gauche, en haut, ou en bas).

```
Droite
at(X, Y) ∧ demoness(X, Y + 1)

Gauche
at(X,Y) ∧ demoness(X, Y - 1)

Haut
at(X,Y) ∧ demoness(X + 1, Y)

Bas
at(X,Y) ∧ demoness(X - 1, Y)
```

Actions

Pour définir les actions, nous avons choisi de ne pas inclure la variable de temps et de détailler chacune des petites actions et leurs conséquences.

Voici la liste des actions :

- se déplacer
- pousser un monstre
- pousser un bloc
- tuer un monstre
    - contre un mur
    - un pic
    - contre un autre monstre
- récupérer une clé
- ouvrir une porte
- avancer dans une porte sans clé

- avancer dans un bloc
  - contre un mur
  - contre un autre bloc
- marcher sur un pic
- être sur un pic

> Pour chaque action, il existe quatres versions (droite, gauche, haut, bas).

Le STRIPS complet se trouve en annexe.

Cependant nous avons trouvé important d'expliquer certains choix de modélisation.

Les déplacements dans les blocs

Il est possible que dans certains niveaux, le joueur soit obligé de perdre un déplacement volontairement. Pour cela il doit avancer dans un bloc contre un mur ou dans un bloc contre un bloc.

Voici l'action associée à un déplacement sur un bloc qui est contre un mur :

```
Action(move_right_into_block_wall(X,Y))
PRECOND :
at (X, Y) ∧ block(X, Y + 1) ∧ wall (X, Y + 2)
EFFECT :
at(X, Y)
```

Les pics

Lorsque le joueur se retrouve sur un pic, il perd un déplacement. Pour modéliser cela, nous avons fait le choix de diviser cet effet en plusieurs étapes.

Premièrement, le joueur peut se déplacer sur un pic :

```
Action(move_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y + 1)
EFFECT :
at (X, Y + 1) ∧ ¬ at(X, Y) ∧ ¬ move(H)
```

> Les autres actions (gauche, haut et bas) se trouvent en annexe.

Nous avons créé une action "move(H)" qui autorise ou non le déplacement du héro.

Lorsque le héros arrive sur le pic, il ne peut plus se déplacer mais une nouvelle action,

“on_spike” se déclenche automatiquement.
Cette action autorise à nouveau les déplacements.

```
Action(on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ ¬ move(X,Y)
EFFECT :
at (X, Y) ∧ move(X,Y)
```

Il est possible que le joueur effectue différentes actions qui ont pour effet que le joueur reste sur place.
Il existe certains pics qui s’ouvrent et se ferment en fonction du déplacement du héro.

Si le héros s’avance dans un bloc et qu’un pic s’ouvre, il se retrouve donc dans état bloquant.

Pour cela nous avons rajouté différentes actions, comme celle d’avancer dans un bloc et d’arriver sur un pic, ce qui a pour effet de ne pas autoriser de déplacement au héro.

```
Action(move_block_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block(X, Y + 1) ∧ wall (X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)
```

Ou encore lorsque le héros pousse un monstre.

```
Action(push_monster_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y + 1)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)
```

Toutes les autres actions se trouvent en annexe.

Il est aussi important de noter que si un monstre est déplacé sur un pic, il meurt.
Nous avons donc créé la même action que “on_spikes” mais pour les montres, qui a pour effet de faire disparaître le monstre.

```
Action(monster_on_spikes(X,Y))
PRECOND :
mob (X, Y) ∧ spikes (X, Y)
EFFECT :
¬ mob(X,Y)
```

3. Méthode 1 : ASPPLAN

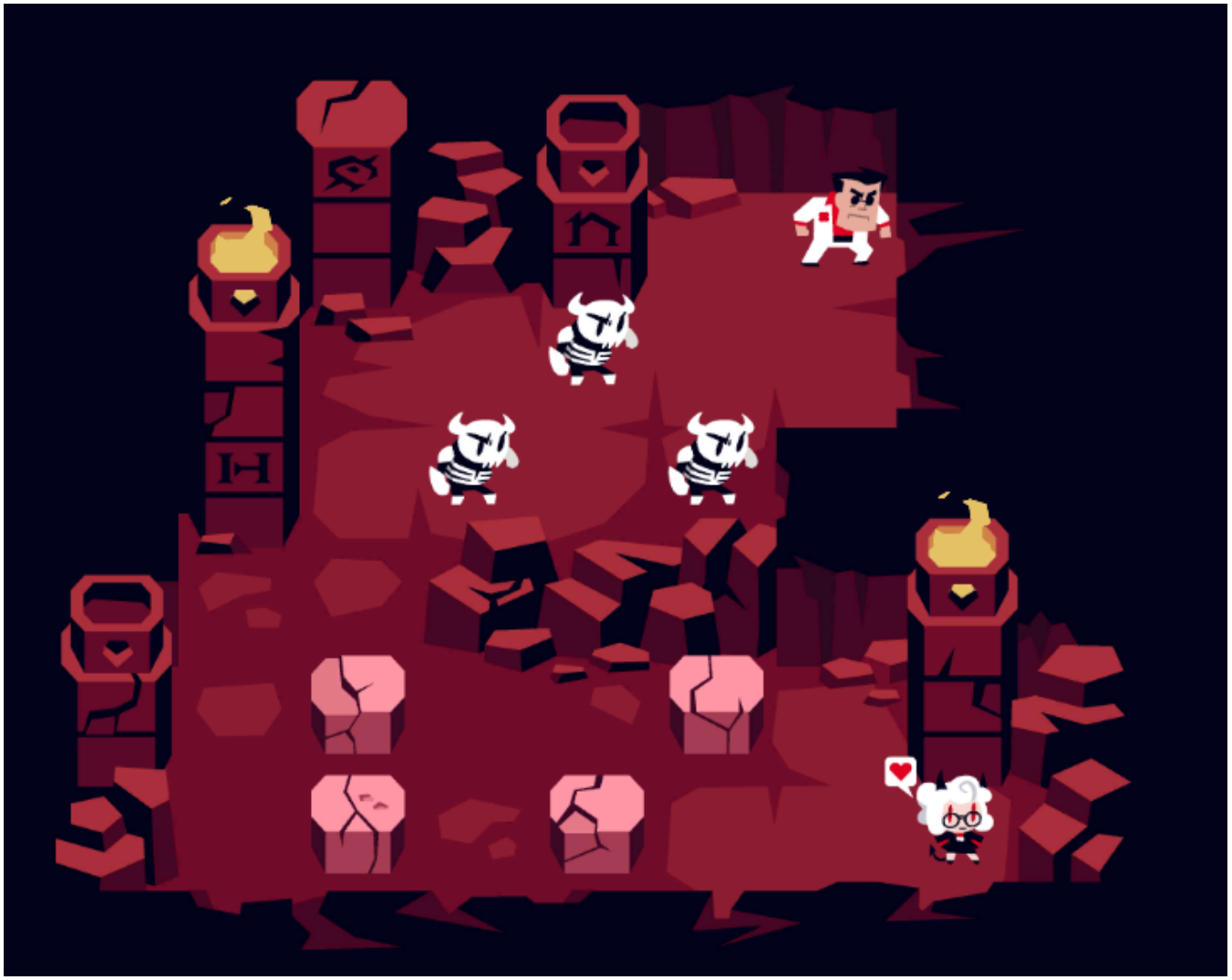
Cette section présente la stratégie adoptée pour la résolution de niveaux de Helltaker en utilisant la programmation en ASP (Answer Set Programming).

3.1 Représentation du problème

3.1.2 Représentation des puzzles

Exemple du niveau 1

Niveau à modéliser :



Ecriture du niveau :

```
Level 1
23
#########
#### H###
##  M ###
## M M###
#  ######
# B  B ##
# B B  D#
#########
```

Description du niveau en ASP :

```
%%% MAP DESCRIPTION
#const horizon=23.
cell(1, 5..6).
cell(2, 2..6).
cell(3, 2..5).
cell(4, 1..2).
cell(5, 1..6).
cell(6, 1..7).
demoness(6, 7).
fluent(block(5, 2), 0).
fluent(block(5, 5), 0).
fluent(block(6, 2), 0).
fluent(block(6, 4), 0).
fluent(mob(2, 4), 0).
fluent(mob(3, 3), 0).
fluent(mob(3, 5), 0).
fluent(at(1, 6), 0).

step(0..horizon-1).
```

3.1.2 Les prédicats

Nous avons utilisé des prédicats binaires pour connaître la position.

Nom	Représentation
at(X, Y)	position du joueur
cell(X, Y)	position d'un bloc
demoness(X, Y)	position d'une femme démon
mob(X, Y)	position des monstre
lock(X, Y)	position du cadenas
key(X, Y)	position de la clé
spike(X, Y)	position d'un pic
trap(X, Y)	position d'un piège

3.1.2 Les fluents

Nous avons utilisé des fluents pour les objets dont l'état peut changer au cours du temps.

Nom	Représentation
fluent(at(X, Y), T)	la position du joueur sur le puzzle à un moment T
fluent(block(X, Y), T)	la position des blocs sur le puzzle à un moment T
fluent(mob(X, Y), T)	la position des monstres sur le puzzle à un moment T
fluent(spike(X, Y), T)	la position d'un pic sur le puzzle à un moment T
fluent(trap(X, Y), T, E)	la position d'un piège sur le puzzle à un moment T et son état E (*safe* ou *unsafe*)
fluent(lock(X, Y), T)	la position d'un cadenas sur la puzzle à un moment T

> Nous avons utilisé des fluents pour le **joueur**, les **blocs** et les **monstres** car ces derniers peuvent se déplacer entre chaque pas de temps, et disparaître dans le cas des **monstres**.

> Nous avons utilisé des fluents pour les **pics** car nous avons considéré que les pièges *unsafe* à l'instant T induisent un pic sur la même cellule à l'instant T.

> Nous avons utilisé des fluents ternaires pour les **pièges** afin d'écrire leur état à chaque pas de temps.

> Nous avons utilisé des fluents pour les **cadenas** car ces derniers disparaissent une fois le joueur passé sur la cellule correspondante, avec la clé.

3.1.3 L'objectif

L'objectif est identique, le joueur doit rejoindre les femmes démons.
Ce but est en réalité atteint lorsque le joueur se trouve à côté d'une cellule des femmes démons.

> N'importe quelle femme démon, afin de pouvoir résoudre le **niveau 3** composé de 3 femmes démons.

Voici la modélisation que nous avons choisi :

```
%%% OBJECTIVE
% test
achieved(T) :- meet(demoness(_, _), T), demoness(_, _).
:- not achieved(_).
% conditions
meet(demoness(X+1, Y), T) :- fluent(at(X, Y), T), demoness(X+1, Y).
meet(demoness(X-1, Y), T) :- fluent(at(X, Y), T), demoness(X-1, Y).
meet(demoness(X, Y+1), T) :- fluent(at(X, Y), T), demoness(X, Y+1).
meet(demoness(X, Y-1), T) :- fluent(at(X, Y), T), demoness(X, Y-1).
```

Lorsque le joueur se trouve à une cellule de différence, l'objectif achieved(T) est atteint.

3.2 Choix d'implémentation et structures de données

Pour l'implémentation en ASP des différentes actions, nous les avons classées en différentes catégories.
Voici les différentes actions que nous avons modélisées :

```
%%% GENERATION OF ACTIONS
% list of possible actions
action(up; down; left; right).
action(hurt);
action(push_block_up; push_block_down; push_block_left; push_block_right).
action(push_mob_up; push_mob_down; push_mob_left; push_mob_right).
action(nop).
% generation
{do(A, T): action(A)}=1 :- step(T).
```

3.2.1 Frame problem

Les fluents évoluent au fur et à mesure du temps. Certains changent de place mais la majorité demeurent au même endroit entre deux pas de temps. Nous avons donc

implémenté une fonction *removed* :

```
%%% FRAME PROBLEM
fluent(F, T+1) :- fluent(F, T), T+1 < horizon, not removed(F, T).
```

3.2.2 Les déplacements du joueur

Le joueur peut se déplacer de gauche à droite et de bas en haut. Il peut aussi pousser des monstres ou des blocs qui se trouvent sur son passage.

> Exemple : Aller à droite

```
%%% ACTION: right
% precondition
:- do(right, T), fluent(at(X, Y), T), not cell(X, Y+1).
:- do(right, T), fluent(at(X, Y), T), fluent(block(X, Y+1), T).
:- do(right, T), fluent(at(X, Y), T), fluent(mob(X, Y+1), T).
% effect
fluent(at(X, Y+1), T+1) :- do(right, T), fluent(at(X, Y), T).
removed(at(X, Y), T) :- do(right, T), fluent(at(X, Y), T).
```

3.2.3 Se blesser (déplacement sur un pic)

Sur son chemin le joueur peut rencontrer des pics, il en existe deux types :

- Les pics statiques
  Ces pics n'évoluent pas avec le temps, lorsque le joueur avance sur un pic de ce type, il perd un déplacement.
- Les pics ouverts/fermés (pièges)
  Ces pics s'ouvrent et se ferment en fonction du nombre de déplacements du joueur. A chaque déplacement leurs états sont modifiés, ils passent de **safe** (cellule classique) à **unsafe** (pic statique).

> Il existe une subtilité : si le joueur se blesse, alors l'état du joueur et des pièges restent inchangés bien qu'il s'écoule un pas de temps.

> Important : le joueur ne peut pas se blesser deux tours de suite.

Pour modéliser cela, nous avons créé une action **hurt** regroupant les deux types de pics :

```
%%% ACTION: hurt
% generation
fluent(spike(X, Y), T) :- fluent(trap(X, Y), T, unsafe). % unsafe trap is equ
removed(spike(X, Y), T) :- fluent(trap(X, Y), T, unsafe). % prevent non-desir
fluent(trap(X, Y), T+1, safe) :- fluent(trap(X, Y), T, unsafe), do(A, T), A !
fluent(trap(X, Y), T+1, unsafe) :- fluent(trap(X, Y), T, safe), do(A, T), A !
fluent(trap(X, Y), T+1, unsafe) :- fluent(trap(X, Y), T, unsafe), do(A, T), A
fluent(trap(X, Y), T+1, safe) :- fluent(trap(X, Y), T, safe), do(A, T), A = h
% precondition
:- fluent(at(X, Y), T), fluent(spike(X, Y), T), not do(hurt, T-1), not do(hur
:- do(hurt, T-1), do(hurt, T). % can't hurt two turn in a row
:- do(hurt, T), fluent(at(X, Y), T), not fluent(spike(X, Y), T). % can't hurt
```

3.2.4 Récupération de clé

Dans certains niveaux, le joueur doit récupérer une clé pour ouvrir une porte est atteindre son objectif.

Pour modéliser cela,

```
%%% LOCK AND KEY
% condition of lock
:- fluent(at(X, Y), T), fluent(lock(X, Y), T), not fluent(have(key), T).
% effect of key
fluent(have(key), T) :- fluent(at(X, Y), T), key(X, Y).
removed(lock(X, Y), T) :- fluent(at(X, Y), T), fluent(lock(X, Y), T).
```

3.2.5 Pousser un monstre

Nous avons regroupé différentes actions de la modélisation en STRIPS en une seule action.
En effet, le fait de tuer ou non un monstre est géré en fonction de la position du monstre une fois poussé. Le monstre change donc de la place dans tous les cas, mais peut ne pas survivre à ce déplacement.

```
%%% ACTION: push_mob_up
% precondition
:- do(push_mob_up, T), fluent(at(X, Y), T), not fluent(mob(X-1, Y), T).
% effect
removed(mob(X-1, Y), T) :- do(push_mob_up, T), fluent(at(X, Y), T).
fluent(mob(X-2, Y), T) :- do(push_mob_up, T), fluent(at(X, Y), T).
```

3.2.6 Mort d’un monstre

Comme expliqué dans le STRIPS, il existe différentes manières de tuer un monstre.

L'implémentation est simple à l'aide de la fonction removed :

```
%%% DEATH OF MOBS
removed(mob(X, Y), T) :- fluent(mob(X, Y), T), fluent(spike(X, Y), T+1).
removed(mob(X, Y), T) :- fluent(mob(X, Y), T), fluent(block(X, Y), T).
removed(mob(X, Y), T) :- fluent(mob(X, Y), T), not cell(X, Y).
```

3.2.6 Pousser un block

On peut pousser si on est à côté d'un bloc, mais l'évolution de sa position dépend de ce qu'il y a derrière.

```
%%% ACTION: push_block_down
% precondition
:- do(push_block_down, T), fluent(at(X, Y), T), not fluent(block(X+1, Y), T).

% effects
removed(block(X+1, Y), T) :-
    do(push_block_down, T),
    fluent(at(X, Y), T),
    cell(X+2, Y),
    not fluent(block(X+2, Y), T),
    not fluent(mob(X+2, Y), T),
    not fluent(lock(X+2, Y), T),
    not demoness(X+2, Y).

fluent(block(X+2, Y), T+1) :-
    do(push_block_down, T),
    fluent(at(X, Y), T),
    removed(block(X+1, Y), T).
```

3.2.7 Ne rien faire…

Une fois l'objectif atteint, le héros n'a plus rien à faire (si ce n'est répondre aux questions de la femme démon dans le vrai jeu).

```
%%% ACTION: nop
% only possible action when finished is nop
:- achieved(T), do(A, T), A != nop.
% but 'nop' can only be done after finishing
:- do(nop, T), not achieved(T).
```

3.3 Expérimentations pratiques

En utilisant **python**, nous avons implémenté la génération aumatique des prédicats et

fluents à partir d'une carte décrite en fichier texte.
Les régles sont stockées dans un simple *string*, et le tout est confié au module clingo assurant le grounding et la résolution.

3.3.1 Tests des solutions

Le modèle donné par clingo est converti en suite d'instructions (haut, bas, gauche, droite) correspondant aux touches du clavier à presser afin de compléter le niveau.
Nous avons donc pu tester la suite d'instructions donnée par le solveur et elle s'est avérée correcte à chaque fois.

Nombre de solution par niveau

Niveau	Nombre de solutions
level 1	8
level 2	3
level 3	1
level 4	2
level 5	26
level 6	39
level 7	2
level 8	4
level 9	2

> Les niveaux 5 et 6 comportent autant de solution car ils sont résolvables en une action de moins que demandé, et qu'il existe beaucoup de manières différentes de perdre un pas de temps.

> Le niveau 9 est le plus difficile notamment car il n'a que 2 deux solutions pour 43 pas de temps

3.3.2 Temps de résolution

Nous avons mesuré les temps de résolution des différents niveaux, pour une seule solution.

> Les mesures ont réalisées sur un laptop Lenovo Yoga Slim 7 (branché) doté d'un processeur AMD Ryzen 7 4700U.

Niveau	Temps d'exécution
level 1	0m0,717s
level 2	0m0,750s
level 3	0m2,082s
level 4	0m0,771s
level 5	0m0,662s
level 6	0m15,219s
level 7	0m7,317s
level 8	0m0,121s
level 9	0m4,100s

4. Méthode 2 : SATPLAN

4.1 Représentation du problème

4.1.1 Le vocabulaire et les actions

Pour traduire la carte de caractères en variables compréhensibles par un solver SAT, nous avons établi le vocabulaire suivant :

Nom	Représentation
at((X,Y),T)	la position du joueur sur le puzzle à l'instant T
block((X,Y),T)	la position des blocs sur le puzzle à l'instant T
mob((X,Y),T)	la position des monstres sur le puzzle à l'instant T
spike((X,Y),T)	la position des pics
trap((X,Y),T)	la position des pièges à l'instant T : vrai si inactivé, faux sinon.
do(T, action)	les actions du héros à l'instant T
have_key(T)	la possession de la clé : vrai si le héros possède la clé à l'instant T, faux sinon
empty((X,Y),T)	l'indication de si une case est "vide" : faux si non vide (bloc, mob, cadenas ou femme démon sur la case))

où :

- T : Instant de temps compris entre 0 et T_MAX-1.
- action : Action du joueur parmi :

```
ACTIONS = {"left", "right", "up", "down", "hurt",
           "push_block_left", "push_block_right", "push_block_up", "push_bloc
           "push_mob_right", "push_mob_up", "push_mob_down"}
```

> Ce sont les mêmes actions qu'en ASP excépté que nous avons cette fois pas implémenté l'action "nop" de rien faire, de toute façon peu utile dans le jeu Helltaker.

Les coordonnées des cases en fonction de leur contenu sont préparées par la fonction *grid_to_coords_dict()*.

À chaque variable correpond un nombre unique afin de pouvoir écrire le dichier DIMACS.

> Les fonctions *grid_to_coords_dict()* et *vocabulary()* construisant le vocabulaire sont données en annexe.

4.2 Choix d'implémentation et structures de données

4.2.1 Structures des données

Nous avons créé des alias de type afin de faciliter la compréhension du code python.

```
# alias de type
Grid = List[List[str]]
Variable = int
Literal = int
Clause = List[Literal]
Map = List[List[Literal]]
Coord = Tuple[int, int]
```

4.2.2 Choix d'implémentation

La modélisation adopté se rapproche largement de celle implémentée en ASP. Cependant, il est nécessaire d'être plus explicite en SAT.

La difficulté de l'implémentation résidait surtout dans les clauses des pics et des pièges. Nous allons donc expliquer nos choix spécifiques d'implémentation pour ceux-là.

Pics

Pour représenter le fait que le héros perde un temps quand il tombe sur un pic, nous avons créé une action "hurt" :

> Dans le même ordre que dans la fonction python *clauses_spikes()* donnée dans l'annexe

- **1ère clause :**
  Le héros ne peut pas se blesser si il n'est pas sur un pic à l'instant T.

```
at((X, Y), T) and not spike((X, Y), T) -> not do(hurt, T)
```

- **2nde clause :**
  Le héros ne peut pas se blesser deux fois de suite

```
do(hurt, T) -> not do(hurt, T+1)
```

- **3ème clause :**
  Si le héros est sur un pic et qu'il ne s'est pas blessé au tour précédent, alors il se blesse.

```
at((X, Y), T) and spike((X, Y), T) and not do(hurt, T-1) -> do(hurt, T)
```

Comme nous avions une clause pour l'unicité d'action parmi {"left", "right", …, "hurt", …, "push_mob_down"}, le héros ne peut alors rien faire quand "hurt" est vrai.

Pièges

Les clauses des pièges ne modélisent pas directement les dégâts infligés au héros. Ces dégâts sont modélisés par les clauses des pics.
Comme les pièges sont soit activés soit désactivés, il faut modéliser le fait qu'ils agissent comme des piques ou non.
Il faut en plus modéliser le basculement activé / désactivé des pics. En effet, quand le héros est blessé, il perd un temps et donc il s'assure que les pics restent comme ils sont. Quand le héros n'est pas blessé, les pics passent d'activé à inactivé et vice-versa.

> Dans le même ordre que dans la fonction python *clauses_traps()* donnée dans l'annexe

- **1ère clause :**
  Si un piège est activé, alors c'est comme si nous avions un pic.
- **2nde clause :**
  Si un piège est inactivé, alors ce n'est pas un pic.
- **3ème clause :**
  Si "hurt" est vrai, alors un piège inactivé reste inactivé au temps suivant.
- **4ème clause :**
  Si "hurt" est vrai, alors un piège activé reste activé au temps suivant.

- **5ème clause :**
  Si “hurt” est faux, alors un piège inactivé devient activé au temps suivant.
- **6ème clause :**
  Si “hurt” est faux, alors un piège activé devient inactivé au temps suivant.

> **Un certain nombre d’autres choix modélisation en SAT sont données en annexe, ou directement dans le code python *helltaker_sat_utils.py***

4.3 Expérimentations pratiques

Nous avons pu mesurer les temps de résulotion en SAT pour différents solvers.

Niveau	Temps d’exécution GOPHERSAT	Temps d’exécution GLUCOSE4
level 1	0m3,737s	0m3,621s
level 2	0m2,348s	0m2,108s
level 3	0m4,564s	0m2,731s
level 4	0m7,264s	0m2,384s
level 5	0m3,116s	0m1,675s
level 6	3m53,543s	0m11,511s
level 7	0m41,325s	0m3,467s
level 8	0m5,307s	0m6,221s
level 9	5m23,017s	0m11,176s

Nous constatons que les temps d’exécution sont plus courts avec le solver SAT GLUCOSE4 de la librairie python pySAT. Plus le niveau est complexe, plus la différence de temps est importante (niveaux 6 et 9 en particulier).

4.4 Perspectives d’amélioration

La modélisation n’est pas optimale et pourrait être améliorée. On pourrait par exemple essayer de rajouter des variables pour que le solveur n’ait pas à faire la même déduction plusieurs fois de suite dans différentes clauses.

> Pour la modélisation du niveau 9 en SAT, on dénombre 6753 variables et 611702 clauses.

De plus, certaines règles ou actions spécifiques n’ont pas été modélisées, comme par exemple :

- Le fait de pouvoir pousser un bloc sur une case qui était un cadenas qu'on a ouvert
- La gestion de plusieurs cadenas et clés
- Le fait qu'on ne peut plus rien faire si on finit en un coup de moins

et d'autres tout aussi spécifiques.
La modélisation adoptée permet tout de même de résoudre tous les problèmes du jeu (excépté le niveau 10).

5. Comparaison expérimentale des 2 méthodes

Pour comparer les méthodes SAT et ASP, nous avons pris les temps du Glucose4 pour SAT.

> Les mesures ont réalisées sur un laptop Lenovo Yoga Slim 7 (branché) doté d'un processeur AMD Ryzen 7 4700U.

Niveau	Temps d'exécution SAT	Temps d'exécution ASP
level 1	0m3,621s	0m0,717s
level 2	0m2,108s	0m0,750s
level 3	0m2,731s	0m2,082s
level 4	0m2,384s	0m0,771s
level 5	0m1,675s	0m0,662s
level 6	0m11,511s	0m15,219s
level 7	0m3,467s	0m7,317s
level 8	0m6,221s	0m0,121s
level 9	0m11,176s	0m4,100s

De manière générale, il est plus efficace de résoudre le jeu Helltaker en ASP. En outre, le langage ASP permet un programme plus court et plus lisible que le SAT, et est bien plus simple à appréhender.

On peut tout de même noter que sur le niveau avec le plus de pas de temps (43 pour le niveau 6), Glucose4 fait mieux que Clingo. Il semble donc meilleur pour les problèmes de plus grande taille.

6. Annexes

6.1 STRIPS complet

Action de déplacement

Simple action de déplacement d'une case à une autre vers la droite, la gauche, le haut, le bas.

```
Action(move_right(X,Y))
PRECOND :
at (X, Y) ∧ ¬ wall (X, Y + 1) ∧ ¬ block (X, Y + 1) ∧ ¬ mob (X, Y + 1) ∧ ¬ loc
EFFECT :
at (X, Y + 1) ∧ ¬ at(X, Y)

Action(move_left(X,Y))
PRECOND :
at(X, Y) ∧ ¬ wall(X, Y - 1) ∧ ¬ block (X, Y - 1) ∧ ¬ mob (X, Y - 1) ∧ ¬ lock
EFFECT :
at(X, Y - 1) ∧ ¬ at(X, Y)

Action(move_top(X,Y))
PRECOND :
at(X, Y) ∧ ¬ wall(X + 1, Y) ∧ ¬ block (X + 1, Y) ∧ ¬ mob (X + 1, Y) ∧ ¬ lock
EFFECT :
at(X + 1, Y) ∧ ¬ at(X, Y)

Action(move_bottom(X,Y))
PRECOND :
at(X, Y) ∧ ¬ wall(X - 1, Y) ∧ ¬ block (X - 1, Y) ∧ ¬ mob (X - 1, Y) ∧ ¬ lock
EFFECT :
at(X - 1, Y) ∧ ¬ at(X, Y)
```

Action de déplacement sur un pic

Lorsque le héros se déplace sur un pic.

```
Action(move_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y + 1)
EFFECT :
at (X, Y + 1) ∧ ¬ at(X, Y) ∧ ¬ move(H)

Action(move_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y - 1)
EFFECT :
at (X, Y - 1) ∧ ¬ at(X, Y) ∧ ¬ move(H)

Action(move_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X + 1, Y)
EFFECT :
at (X + 1, Y) ∧ ¬ at(X, Y) ∧ ¬ move(X,Y)

Action(move_bottom_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X - 1, Y)
EFFECT :
at (X - 1, Y) ∧ ¬ at(X, Y) ∧ ¬ move(X,Y)
```

Action d’être sur un pic

Effet lorsque le héros se trouve une un pic.

```
Action(on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ ¬ move(X,Y)
EFFECT :
at (X, Y) ∧ move(X,Y)
```

Action de déplacement dans un bloc contre un mur

Action lorsque le héros se déplace dans un bloc qui est contre un mur vers la droite, la gauche, le haut ou le bas.

```
Action(move_right_into_block_wall(X,Y))
PRECOND :
at (X, Y )∧ block(X, Y + 1) ∧ wall (X, Y + 2)
EFFECT :
at(X, Y)

Action(move_left_into_block_wall(X,Y))
PRECOND :
at (X, Y )∧ block(X, Y - 1) ∧ wall (X, Y - 2)
EFFECT :
at(X, Y)

Action(move_top_into_block_wall(X,Y))
PRECOND :
at (X, Y )∧ block(X + 1, Y) ∧ wall (X + 2, Y)
EFFECT :
at(X, Y)

Action(move_bottom_into_block_wall(X,Y))
PRECOND :
at (X, Y )∧ block(X - 1, Y) ∧ wall (X - 2, Y)
EFFECT :
at(X, Y)
```

Action de déplacement dans un bloc contre un bloc

Action lorsque le héros se déplace dans un bloc qui est contre un autre bloc vers la droite, la gauche, le haut ou le bas.

```
Action(move_right_into_block_block(X,Y))
PRECOND :
at (X, Y) ∧ block(X, Y + 1) ∧ block (X, Y + 2)
EFFECT :
at(X, Y)

Action(move_left_into_block_block(X,Y))
PRECOND :
at (X, Y) ∧ block(X, Y - 1) ∧ block (X, Y - 2)
EFFECT :
at(X, Y)

Action(move_top_into_block_block(X,Y))
PRECOND :
at (X, Y) ∧ block(X + 1, Y) ∧ block (X + 2, Y)
EFFECT :
at(X, Y)

Action(move_bottom_into_block_block(X,Y))
PRECOND :
at (X, Y) ∧ block(X - 1, Y) ∧ block (X - 2, Y)
EFFECT :
at(X, Y)
```

Action de pousser un montre

Action lorsque le héros se déplace dans un monstre. Le monstre avance d'une case et le héros reste sur la même case.

```
Action(push_monster_right(X,Y))
PRECOND :
at(X, Y) ∧ mob (X, Y + 1) ∧ ¬ wall(X, Y + 2) ∧ ¬ block (X, Y + 2) ∧ ¬ lock (X
EFFECT :
at(X, Y) ∧ mob(X, Y + 2) ∧ ¬ mob(X, Y + 1)

Action(push_monster_left(X,Y))
PRECOND :
at(X, Y) ∧ mob (X, Y - 1) ∧ ¬ wall(X, Y - 2) ∧ ¬ block (X, Y - 2) ∧ ¬ lock (X
EFFECT :
at(X, Y) ∧ mob(X, Y - 2) ∧ ¬ mob(X, Y - 1)

Action(push_monster_top(X,Y))
PRECOND :
at(X, Y) ∧ mob (X + 1, Y) ∧ ¬ wall(X + 2, Y) ∧ ¬ block (X + 2, Y) ∧ ¬ lock (X
EFFECT :
at(X, Y) ∧ mob(X + 2, Y) ∧ ¬ mob(X + 1, Y)

Action(push_monster_bottom(X,Y))
PRECOND :
at(X, Y) ∧ mob (X - 1, Y) ∧ ¬ wall(X - 2, Y) ∧ ¬ block (X - 2, Y) ∧ ¬ lock (X
EFFECT :
at(X, Y) ∧ mob(X - 2, Y) ∧ ¬ mob(X - 1, Y)
```

Action de pousser un bloc

Action lorsque le héros se déplace dans un bloc. Le bloc avance d'une case et le héros reste sur la même case.

```
Action(push_block_right(X,Y))
PRECOND :
at (X, Y) ∧ block (X, Y + 1) ∧ ¬ wall (X, Y + 2) ∧ ¬ mob (X, Y + 2) ∧ ¬ lock
EFFECT :
at (X, Y) ∧ block (X, Y + 2) ∧ ¬ block (X, Y + 1)

Action(push_block_left(X,Y))
PRECOND :
at (X, Y) ∧ block (X, Y - 1) ∧ ¬ wall (X, Y - 2) ∧ ¬ mob (X, Y - 2) ∧ ¬ lock
EFFECT :
at (X, Y) ∧ block (X, Y - 2) ∧ ¬ block (X, Y - 1)

Action(push_block_top(X,Y))
PRECOND :
at (X, Y) ∧ block (X + 1, Y) ∧ ¬ wall (X + 2, Y) ∧ ¬ mob (X + 2, Y) ∧ ¬ lock
EFFECT :
at (X, Y) ∧ block (X + 2, Y) ∧ ¬ block (X + 1, Y)

Action(push_block_bottom(X,Y))
PRECOND :
at (X, Y) ∧ block (X - 1, Y) ∧ ¬ wall (X - 2, Y) ∧ ¬ mob (X - 2, Y) ∧ ¬ lock
EFFECT :
at (X, Y) ∧ block (X - 2, Y) ∧ ¬ block (X - 1, Y)
```

Action de tuer un monstre dans un bloc

Action lorsque le héros se déplace vers un monstre qui est à côté d'un bloc. Cette action ne fait pas bouger le héros mais détruit le monstre.

```
Action(kill_right_block(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y + 1) ∧ block (X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y + 1)

Action(kill_left_block(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y - 1) ∧ block (X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y - 1)

Action(kill_bottom_block(X,Y))
PRECOND :
at (X, Y) ∧ mob (X - 1, Y) ∧ block (X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X - 1, Y)

Action(kill_top_block(X,Y))
PRECOND :
at (X, Y) ∧ mob (X + 1, Y) ∧ block (X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X + 1, Y)
```

Action de tuer un monstre dans un mur

Action lorsque le héros se déplace vers un monstre qui est à côté d'un mur. Cette action ne fait pas bouger le héros mais détruit le monstre.

```
Action(kill_right_wall(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y + 1) ∧ wall (X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y + 1)

Action(kill_left_wall(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y - 1) ∧ wall (X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y - 1)

Action(kill_bottom_wall(X,Y))
PRECOND :
at (X, Y) ∧ mob (X - 1, Y) ∧ wall (X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X - 1, Y)

Action(kill_top_wall(X,Y))
PRECOND :
at (X, Y) ∧ mob (X + 1, Y) ∧ wall (X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X + 1, Y)
```

Action de tuer un monstre dans un pic

Action lorsque le héros se déplace vers un monstre qui est à côté d'un pic. Cette action ne fait pas bouger le héros mais détruit le monstre.

```
Action(kill_right_spikes(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y + 1) ∧ spikes (X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y + 1)

Action(kill_left_spikes(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y - 1) ∧ spikes (X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y - 1)

Action(kill_bottom_spikes(X,Y))
PRECOND :
at (X, Y) ∧ mob (X - 1, Y) ∧ spikes (X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X - 1, Y)

Action(kill_top_spikes(X,Y))
PRECOND :
at (X, Y) ∧ mob (X + 1, Y) ∧ spikes (X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X + 1, Y)
```

Action de tuer un monstre dans un monstre

Action lorsque le héros se déplace vers un monstre qui est à côté d'un monstre. Cette action ne fait pas bouger le héros mais détruit le monstre poussé par le héro.

```
Action(kill_right_monster(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y + 1) ∧ monster (X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y + 1)

Action(kill_left_monster(X,Y))
PRECOND :
at (X, Y) ∧ mob (X, Y - 1) ∧ monster (X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ mob (X, Y - 1)

Action(kill_bottom_monster(X,Y))
PRECOND :
at (X, Y) ∧ mob (X - 1, Y) ∧ monster (X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X - 1, Y)

Action(kill_top_monster(X,Y))
PRECOND :
at (X, Y) ∧ mob (X + 1, Y) ∧ monster (X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ mob (X + 1, Y)
```

Action d’aller dans un bloc contre mur et qu’un pic s’ouvre

Action lorsque le héros se prend un bloc contre un mur et qu’il reste donc sur place mais qu’un pic s’ouvre.

```
Action(move_block_wall_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ wall (X, Y + 1)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(move_block_wall_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ wall (X, Y - 1)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(move_block_wall_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ wall (X + 1, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(move_block_wall_bottom_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ wall (X - 1, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)
```

Action d’aller dans un bloc contre bloc et qu’un pic s’ouvre

Action lorsque le héros se prend un bloc contre un bloc et qu’il reste donc sur place mais qu’un pic s’ouvre.

```
Action(move_block_block_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X, Y + 1) ∧ block (X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(move_block_block_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X, Y - 1) ∧ block (X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(move_block_block_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X + 1, Y) ∧ block (X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(move_block_block_bottom_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X - 1, Y) ∧ block (X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)
```

Action de pousser un bloc et qu'un pic s'ouvre

Action lorsque le héros pousse un bloc alors qu'il se trouve sur un pic qui s'ouvre.

```
Action(push_block_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X, Y + 1)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(push_block_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X, Y - 1)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(push_block_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X + 1, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(push_block_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ block (X - 1, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)
```

Action de pousser un monstre et qu'un pic s'ouvre

Action lorsque le héros se prends un bloc contre un mur et qu'il reste donc sur place mais qu'un pic s'ouvre.

```
Action(push_monster_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y + 1)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(push_monster_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y - 1)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(push_monster_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X + 1, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)

Action(push_monster_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X - 1, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y)
```

Action de tuer un monstre contre un mur et qu'un pic s'ouvre

Action lorsque le héros tue un monstre contre un mur alors qu'il se trouve sur un pic qui s'ouvre.

```
Action(kill_monster_wall_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y + 1) ∧ wall(X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X,Y + 1)

Action(kill_monster_wall_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y - 1) ∧ wall(X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X,Y - 1)

Action(kill_monster_wall_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X + 1, Y) ∧ wall(X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X + 1,Y)

Action(kill_monster_wall_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X - 1, Y) ∧ wall(X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X - 1,Y)
```

Action de tuer un monstre contre un bloc et qu'un pic s'ouvre

Action lorsque le héros tue un monstre contre un bloc alors qu'il se trouve sur un pic qui s'ouvre.

```
Action(kill_monster_block_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y + 1) ∧ block(X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X,Y + 1)

Action(kill_monster_block_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y - 1) ∧ block(X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X,Y - 1)

Action(kill_monster_block_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X + 1, Y) ∧ block(X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X + 1,Y)

Action(kill_monster_block_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X - 1, Y) ∧ block(X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X - 1,Y)
```

Action de tuer un monstre contre un monstre et qu'un pic s'ouvre

Action lorsque le héros tue un monstre contre un monstre alors qu'il se trouve sur un pic qui s'ouvre.

```
Action(kill_monster_monster_right_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y + 1) ∧ mob(X, Y + 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X,Y + 1)

Action(kill_monster_monster_left_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X, Y - 1) ∧ mob(X, Y - 2)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X,Y - 1)

Action(kill_monster_monster_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X + 1, Y) ∧ mob(X + 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X + 1,Y)

Action(kill_monster_monster_top_on_spikes(X,Y))
PRECOND :
at (X, Y) ∧ spikes (X, Y) ∧ mob (X - 1, Y) ∧ mob(X - 2, Y)
EFFECT :
at (X, Y) ∧ ¬ move(X,Y) ∧ ¬ mob(X - 1,Y)
```

Action de tuer un montre s'il se trouve sur un pic qui s'ouvre

Si un monstre se trouve sur un pic, il disparaît.

```
Action(monster_on_spikes(X,Y))
PRECOND :
mob (X, Y) ∧ spikes (X, Y)
EFFECT :
¬ mob(X,Y)
```

Action de marcher sur une clé

Action lorsque le héros se trouve sur la même cellule où se trouve une clé.

```
Action(havekey())
PRECOND :
at (X, Y) ∧ key (X, Y)
EFFECT :
at (X, Y) ∧ have(key)
```

Action d'ouvrir une porte

Action lorsque le héros se déplace sur une porte et qu'il a une clé.
La porte disparaît par la suite et le joueur avance.

```
Action(unlock_right(X,Y))
PRECOND :
at (X, Y) ∧ door(X,Y+1) ∧ have(key)
EFFECT :
at (X, Y+1) ∧ ¬ have(key) ∧ ¬ door(X,Y+1)

Action(unlock_left(X,Y))
PRECOND :
at (X,Y) ∧ door(X,Y-1) ∧ have(key)
EFFECT :
at (X,Y-1) ∧ ¬ have(key) ∧ ¬ door(X,Y-1)

Action(unlock_top(X,Y))
PRECOND :
at (X,Y) ∧ door(X+1,Y) ∧ have(key)
EFFECT :
at (X+1,Y) ∧ ¬ have(key) ∧ ¬ door(X+1,Y)

Action(unlock_bottom(X,Y))
PRECOND :
at (X,Y) ∧ door(X-1,Y) ∧ have(key)
EFFECT :
at (X-1,Y) ∧ ¬ have(key) ∧ ¬ door(X-1,Y)
```

Action d'avancer dans une porte sans clé

Action lorsque le héros se déplace sur une porte mais qu'il n'a pas de clé.
Il reste sur place.

```
Action(lock_right(X,Y))
PRECOND :
at (X, Y) ∧ door(X,Y+1) ∧ ¬ have(key)
EFFECT :
at (X, Y)

Action(lock_left(X,Y))
PRECOND :
at (X,Y) ∧ door(X,Y-1) ∧ ¬ have(key)
EFFECT :
at (X, Y)

Action(lock_top(X,Y))
PRECOND :
at (X,Y) ∧ door(X+1,Y) ∧ ¬ have(key)
EFFECT :
at (X, Y)

Action(lock_bottom(X,Y))
PRECOND :
at (X,Y) ∧ door(X-1,Y) ∧ ¬ have(key)
EFFECT :
at (X, Y)
```

6.2 Fonctions Python du solveur SAT

Fonctions pour le vocabulaire

```
def grid_to_coords_dict(grid: Grid) -> dict:
    """
    :param grid: grid of the level
    :return: dict containing position of each element
    """
    coords = {
        "cells": [],
        "empty": [],
        "hero": [],
        "demonesses": [],
        "key": [],
        "lock": [],
        "spikes": [],
        "traps_safe": [],
        "traps_unsafe": [],
        "blocks": [],
        "mobs": [],
    }

    for i, line in enumerate(grid):
        for j, cell in enumerate(line):
            if cell != "#":
                coords["cells"].append((i, j))
            if cell in ["S", " ", "T", "U", "K"]:
                coords["empty"].append((i, j))
            if cell == "H":
                coords["hero"].append((i, j))
            elif cell == "D":
                coords["demonesses"].append((i, j))
            elif cell == "K":
                coords["key"].append((i, j))
            elif cell == "L":
                coords["lock"].append((i, j))
            elif cell == "S":
                coords["spikes"].append((i, j))
            elif cell == "T":
                coords["traps_safe"].append((i, j))
            elif cell == "U":
                coords["traps_unsafe"].append((i, j))
            elif cell == "B":
                coords["blocks"].append((i, j))
            elif cell == "M":
                coords["mobs"].append((i, j))
            elif cell == "O":
                coords["blocks"].append((i, j))
                coords["spikes"].append((i, j))
            elif cell == "P":
                coords["blocks"].append((i, j))
                coords["traps_safe"].append((i, j))
            elif cell == "Q":
                coords["blocks"].append((i, j))
```

```
                coords["traps_unsafe"].append((i, j))

    return coords


def vocabulary(coords: dict, t_max: int) -> dict:
    """
    :param coords: dict containing coord of each element of the map
    :param t_max: horizon
    :return: dict containing all the vocabulary
    """
    cells = coords["cells"]
    traps = coords["traps_unsafe"] + coords["traps_safe"]

    act_vars = [("do", t, a) for t in range(t_max) for a in ACTIONS]
    at_vars = [("at", t, c) for t in range(t_max + 1) for c in cells]
    spike_vars = [("spike", t, c) for t in range(t_max + 1) for c in ce
    traps_vars = [("trap", t, c) for t in range(t_max + 1) for c in tra
    have_key_vars = [("have_key", t) for t in range(t_max + 1)]
    empty_cell_vars = [("empty", t, c) for t in range(t_max + 1) for c
    block_vars = [("block", t, c) for t in range(t_max + 1) for c in ce
    mob_vars = [("mob", t, c) for t in range(t_max + 1) for c in cells]

    return {
        v: i + 1
        for i, v in enumerate(
            act_vars
            + at_vars
            + spike_vars
            + traps_vars
            + have_key_vars
            + block_vars
            + empty_cell_vars
            + mob_vars
        )
    }
```

Fonction level_data_to_clauses

```
def level_data_to_clauses(data: dict) -> Tuple[dict, List[Clause]]:
    """
    :param data: dict containing all level data
    :return: all clauses corresponding to the level
    """
    coords = grid_to_coords_dict(data["grid"])
    t_max = data["max_steps"]
    var2n = vocabulary(coords, t_max)

    clauses = clauses_exactly_one_action(var2n, t_max) + clauses_initia
        var2n, coords, t_max
    )

    for cell in coords["cells"]:
        clauses += clauses_successor_from_given_position(
            var2n, coords["cells"], t_max, cell
        )
        clauses += clauses_spikes(var2n, t_max, cell)
        clauses += clauses_empty(var2n, t_max, cell)
        clauses += clauses_blocks(var2n, t_max, coords["cells"], cell)
        clauses += clauses_mobs(var2n, t_max, coords["cells"], cell)

    for cell in coords["cells"]:
        if cell not in coords["lock"] + coords["demonesses"]:
            clauses += clauses_empty(var2n, t_max, cell)

    for cell in coords["traps_unsafe"] + coords["traps_safe"]:
        clauses += clauses_traps(var2n, t_max, cell)

    if len(coords["lock"]) > 0 and len(coords["key"]) > 0:
        clauses += clauses_lock_and_key(
            var2n, t_max, coords["lock"][0], coords["key"][0]
        )

    # on doit être à côté d'une demoness à la fin
    clauses.append(
        [
            var2n[("at", t_max, coord)]
            for demoness in coords["demonesses"]
            for coord in adjacent(demoness)
            if coord in coords["cells"]
        ]
    )

    return var2n, clauses
```

Fonction clauses_to_dimacs

```
def clauses_to_dimacs(clauses: set[Clause], numvar: int) -> str:
    """
    :param clauses: list of all clauses corresponding to the problem
    :param numvar: number of variables
    :return: string
    """
    dimacs = "c Helltaker SAT\np cnf " + str(numvar) + " " + str(len(cl
    for clause in clauses:
        for atom in clause:
            dimacs += str(atom) + " "
        dimacs += "0\n"
    return dimacs
```

Fonction clauses_spikes

```
def clauses_spikes(var2n: dict, t_max: int, position: Coord) -> List[Cl
    """
    :param var2n: dict giving the corresponding number for each variabl
    :param t_max: horizon
    :param position: cell position
    :return: clauses corresponding to spikes
    """
    clauses = []

    # interdit de hurt si pas sur spike
    clauses += [
        [
            -var2n[("at", t, position)],
            -var2n[("do", t, "hurt")],
            var2n[("spike", t, position)],
        ]
        for t in range(t_max)
    ]

    # pas hurt deux tours de suite
    clauses += [
        [-var2n[("do", t, "hurt")], -var2n[("do", t + 1, "hurt")]]
        for t in range(t_max - 1)
    ]

    # obliger de hurt si sur spike et pas hurt au dernier tour
    clauses += [
        [
            var2n[("do", t - 1, "hurt")],
            var2n[("do", t, "hurt")],
            -var2n[("at", t, position)],
            -var2n[("spike", t, position)],
        ]
        for t in range(1, t_max)
    ]

    return clauses
```

Fonction clauses_traps

```
def clauses_traps(var2n: dict, t_max: int, position: Coord) -> List[Cla
    """
    :param var2n: dict giving the corresponding number for each variabl
    :param t_max: horizon
    :param position: cell position
    :return: clauses corresponding to traps
    """
    clauses = []

    # un trap unsafe est équivalent à un spike
    clauses += [
        [var2n[("trap", t, position)], var2n[("spike", t, position)]]
        for t in range(t_max)
    ]

    # un trap safe n'est pas un spike
    clauses += [
        [-var2n[("trap", t, position)], -var2n[("spike", t, position)]]
        for t in range(t_max)
    ]

    # un trap unsafe reste unsafe si l'action est hurt
    clauses += [
        [
            -var2n[("do", t, "hurt")],
            var2n[("trap", t, position)],
            -var2n[("trap", t + 1, position)],
        ]
        for t in range(t_max)
    ]

    # un trap safe reste safe si l'action est hurt
    clauses += [
        [
            -var2n[("do", t, "hurt")],
            -var2n[("trap", t, position)],
            var2n[("trap", t + 1, position)],
        ]
        for t in range(t_max)
    ]

    # un trap unsafe devient safe si l'action n'est pas hurt
    clauses += [
        [
            var2n[("do", t, "hurt")],
            var2n[("trap", t, position)],
            var2n[("trap", t + 1, position)],
        ]
        for t in range(t_max)
    ]

    # un trap safe devient unsafe si l'action est hurt
```

```
    # un trap safe devient unsafe si l'action n'est pas hurt
    clauses += [
        [
            var2n[("do", t, "hurt")],
            -var2n[("trap", t, position)],
            -var2n[("trap", t + 1, position)],
        ]
        for t in range(t_max)
    ]

    return clauses
```

Supplément : Les règles

Pour représenter le problème, voici les règles que nous avons explicité pour pouvoir ensuite les transformer en clauses. La grande majorité des clauses sont à écrire pour tout instant t allant de 0 à la valeur maximale.

Position et déplacement du héro

- Le héros doit se trouver à un endroit précis au départ.
- Le héros doit se trouver à un seul endroit par instant t.
- Le héros peut se déplacer sur les cases voisines seulement.
- Le héros ne peut pas marcher sur un mur ou dépasser un bord.

Pics

- Le héros est blessé quand il est sur un pic et ne peut donc pas se déplacer.
- Le héros perd un temps quand il est sur un pic.

Pièges

- Un piège activé a les mêmes effets qu'un pic et un inactivé n'a pas les mêmes effets.
- Un piège s'active et se désactive à chaque fois que le héros se déplace

Verrou et clé

- Le héros ne peut pas aller sur un verrou s'il n'a pas la clé
- Le héros n'a pas la clé au départ

Arrivée

- Le héros doit arriver à côté de la démonne au temps maximum.

Nous expliquons en partie 3.2. comment nous avons écrit les clauses.
Celles-ci sont rassemblées dans la fonction *level_data_to_clauses()* disponible en Annexe.