Projet IA02 : HIMAN

Contribution :

- Souhail El Baamrani
- Julien Pillis

Vous trouverez dans ce rapport les différents éléments d'élaboration de notre programme pour le projet HITMAN. Celui-ci est découpé en trois parties : la première dédiée à la phase 1, la deuxième à la phase 2 et la dernière partie décrit comment faire fonctionner notre programme.

Chaque partie est dédiée à une phase, et décrit les modélisations associées (SAT/STRIPS), le fonctionnement du programme ainsi que les qualités et défauts associés.

Pour visionner correctement ce rapport (symboles), cliquez **ici** !

Phase 1 | phase1.py :

Modélisation SAT

Pour la génération des contraintes, nous allons grandement utiliser les fonctions :

- **at_least_n(n: int, vars: List[int]) -> List[Clause]** : retourne l'ensemble de clause traitant la contrainte : "au moins n variables vraies dans la liste"
- **at_most_n(n: int, vars: List[int]) -> List[Clause]** : retourne l'ensemble de clause traitant la contrainte : "au plus n variables vraies dans la liste"
- **exactly_n(n: int, vars: List[int]) -> List[Clause]** : retourne l'ensemble de clause traitant la contrainte : "exactement n variables vraies dans la liste"

Nous allons également considérer un monde rectangulaire $m \times n$ $(\text{lignes} \times \text{colonnes})$ soit $m \times n$ cases.

Unicité des éléments sur une case

Une case contient un élément parmi:

- un invité $I_{i,j}$
- un garde $G_{i,j}$
- une cible $CI_{i,j}$
- un mur $M_{i,j}$
- un déguisement $D_{i,j}$
- une corde $CO_{i,j}$
- case vide $V_{i,j}$

Chaque case n'a qu'un seul élément. Il faut donc utiliser la contrainte *unique* :

$CO_{i,j} \rightarrow \neg D_{i,j} \wedge \neg CI_{i,j} \wedge \neg G_{i,j} \wedge \neg I_{i,j} \wedge \neg M_{i,j} \wedge \neg V_{i,j}$

$D_{i,j} \rightarrow \neg CO_{i,j} \wedge \neg CI_{i,j} \wedge \neg G_{i,j} \wedge \neg I_{i,j} \wedge \neg M_{i,j} \neg V_{i,j}$

$CI_{i,j} \rightarrow \neg D_{i,j} \wedge \neg CO_{i,j} \wedge \neg G_{i,j} \wedge \neg I_{i,j} \wedge \neg M_{i,j} \neg V_{i,j}$

$G_{i,j} \rightarrow \neg D_{i,j} \wedge \neg CI_{i,j} \wedge \neg CO_{i,j} \wedge \neg CI_{i,j} \wedge \neg M_{i,j} \neg V_{i,j}$

$I_{i,j} \rightarrow \neg D_{i,j} \wedge \neg CI_{i,j} \wedge \neg G_{i,j} \wedge \neg CO_{i,j} \wedge \neg M_{i,j} \neg V_{i,j}$

$M_{i,j} \rightarrow \neg D_{i,j} \wedge \neg CI_{i,j} \wedge \neg G_{i,j} \wedge \neg I_{i,j} \wedge \neg CO_{i,j} \neg V_{i,j}$

$V_{i,j} \rightarrow \neg D_{i,j} \wedge \neg CI_{i,j} \wedge \neg G_{i,j} \wedge \neg I_{i,j} \wedge \neg CO_{i,j} \neg M_{i,j}$

Ces contraintes sont générées par la fonction : ***create_cell_constraints() -> ClauseBase***.

Remarque :

Toutes les personnes (gardes et invités) regardent vers une direction. Ce regarde définit donc des cases à "éviter", dans la mesure où passer devant un garde ou se faire voir en train de tuer la cible rapporter des malus.
Ce regard est donné par l'arbitre du jeu. Cet élément ne fait pas partie du contenu des cases à déterminer pour la phase 1, mais il nous aide dans le choix des déplacements de Hitman. Il n'est donc pas nécessaire de l'implémenter en SAT (car il n'aide en rien à déduction du plateau, le solveur ne peut pas déduire un regard), mais s'avère utile pour le choix des déplacements de Hitman.

Unicité de la cible, de la corde et du déguisement

Ces éléments sont uniques sur la carte. Il faut donc implémenter la contrainte unique pour toutes les cases de la carte :

Pour le déguisement :

$D_{0,0} \leftrightarrow \neg D_{0,1} \wedge \neg D_{0,2} \wedge \ldots \wedge \neg D_{m,n-1} \wedge \neg D_{m,n}$

$D_{0,1} \leftrightarrow \neg D_{0,0} \wedge \neg D_{0,2} \wedge \ldots \wedge \neg D_{m,n-1} \wedge \neg D_{m,n}$

…

$D_{m,n} \leftrightarrow \neg D_{0,0} \wedge \neg D_{0,1} \wedge \ldots \wedge \neg D_{m,n-2} \wedge \neg D_{m,n-1}$

Pour la cible :

$CI_{0,0} \leftrightarrow \neg CI_{0,1} \wedge \neg CI_{0,2} \wedge \ldots \wedge \neg CI_{m,n-1} \wedge \neg CI_{m,n}$

$CI_{0,1} \leftrightarrow \neg CI_{0,0} \wedge \neg CI_{0,2} \wedge \ldots \wedge \neg CI_{m,n-1} \wedge \neg CI_{m,n}$

… \

$CI_{m,n} \leftrightarrow \neg CI_{0,0} \wedge \neg CI_{0,1} \wedge \ldots \wedge \neg CI_{m,n-2} \wedge \neg CI_{m,n-1}$

Pour la corde :

$CO_{0,0} \leftrightarrow \neg CO_{0,1} \wedge \neg CO_{0,2} \wedge \ldots \wedge \neg CO_{m,n-1} \wedge \neg CO_{m,n}$

$CO_{0,1} \leftrightarrow \neg CO_{0,0} \wedge \neg CO_{0,2} \wedge \ldots \wedge \neg CO_{m,n-1} \wedge \neg CO_{m,n}$

… \

$CO_{m,n} \leftrightarrow \neg CO_{0,0} \wedge \neg CO_{0,1} \wedge \ldots \wedge \neg CO_{m,n-2} \wedge \neg CO_{m,n-1}$

Ces contraintes sont générées par la fonction : ***create_objects_constraints() -> ClauseBase***

Nombre exacte de gardes et de civils

Le nombre de gardes est connu. Si l'on suppose qu'il vaut n, nous devons introduire les contraintes suivantes :

$G_{0,0} \wedge G_{0,1} \wedge \ldots \wedge G_{0,n} \leftrightarrow \neg G_{0,n+1} \wedge \neg G_{0,n+2} \wedge \ldots \wedge \neg G_{m,n-1} \wedge \neg G_{m,n}$

On ajoute cette contrainte pour toutes les combinaisons possibles des positions des n gardes.

La logique est identique pour les civils.

Ces contraintes sont générées par la fonction : ***create_npc_constraints(nbGuards: int, nbCivils: int) -> ClauseBase***

Contraintes dynamiques

Ecoute

L'écoute permet d'obtenir de précieuses informations sur la localisation des personnes présentes sur le lieu. Bien que nous ne connaissons pas leur localisation exacte, nous savons qu'elles se situent sur l'une des neuf cases à proximité directe de Hitman.

Combinée avec les autres informations perçues, la localisation ajoutera de nouvelles contraintes qui permettront d'enrichir la base de clauses, et par conséquent, l'obtention de potentiels déductions sur les positions des personnes (ces contraintes sont ajoutées au fur et à mesure du jeu).

Le sujet précise qu'Hitman n'est pas compris dans les personnes entendues. Cependant, il peut se trouver sur la case d'un invité ou même de la cible, et l'entendre. Il faut donc prendre en compte la case où il se trouve.

Ces contraintes sont générées par la fonction : ***constraints_listener(position: Tuple[int, int], heard: int)->ClauseBase***

Vision

En fonction de ce que l'arbitre retourne, on peut générer des clauses unitaires. Par exemple, si l'arbitre retourne Garde en (i,j) on intègre à la base de clause : $G_{i,j}$. Cela facilitera la déduction pour le solveur puisque ça base de connaissance augmente dynamiquement, au fur et à mesure des déplacements de Hitman.

Ces contraintes sont générées par la fonction : ***create_npc_constraints(nbGuards: int, nbCivils: int) -> ClauseBase***

Notre Programme

Exploration

La fonction réalisant l'exploration est la suivante : ***explore****(hr: HitmanReferee, status: dict[str, Union[str, int, tuple[int, int], HC, list[tuple[tuple[int, int], HC]]]])*

L'objectif est de cartographier la carte en trouvant un chemin menant à la case la plus proche. Ce chemin ne doit passer que pas des cases que l'on connait. Nous avons implémenter cette recherche de chemin avec la fonction findPaths() qui calcul l'ensemble des chemins possibles pour aller d'un point à un autre, et respectant les conditions ci-dessus. Lors du calcul de ces chemins, notre algorithme établi une estimation des coûts pour chaque chemin (nombre de rotations/pas en avant, entrée et mouvement dans le champ de vision d'un garde). Le chemin le moins coûteux est celui que nous récupérons.
Ainsi, tant que l'entièreté de la carte n'est pas connue (existence de None dans le dictionnaire known_cells: Dict[Tuple[int, int], HC]), nous explorons de nouvelles cases.

Au cours de l'exploration, nous complétons la base de clauses ainsi que le dictionnaires known_cells des éléments que nous connaissons par l'arbitre (vision et écoute).

Utilisation de la déduction du solveur

Le solveur n'est utile que pour la détection d'emplacement des gardes.
Dans de très rares cas, il peut déduire la position d'une cible, d'une corde ou d'un déguisement s'il ne reste qu'une case inconnue et que l'élément n'a pas été trouvé.

Nous nous concentrerons ainsi sur l'écoute.

La fonction qui gère cela est : ***detectGuards() -> NoReturn***:
Cette fonction n'est appelée que lorsque la position de Hitman change. Elle introduit à la base de clauses la négation de la clause unitaire pour chaque emplacement de la zone d'écoute, pouvant potentiellement contenir un garde (cases inconnues) : $\neg G_{i,j}$.

La base de clauses est alors transmise au solveur qui cherche une solution. Dans le cas où le solveur ne renvoie pas de solution, c'est qu'il y a forcément un garde à la position que nous avons testée ! $G_{i,j}$ est alors ajouté à notre base de clauses "sûres" et nous avons connaissances de la position d'un nouveau garde. Cependant, nous ne connaissons pas la direction de son regard. Nous définissons donc une zone de cases vers lesquelles le garde pourrait regarder (*define_danger_zone((col, row)) -> NoReturn*). Les cases de cette zone sont alors prises en compte lors du calcul du chemin vers une case inconnue, et apporte un malus lors du calcul du coût d'un chemin.

Qualités et défauts

Le principal problème de notre algorithme est la génération des contraintes SAT, et en particulier celles liées aux nombres de gardes et de civils. Au moment de générer les contraintes de bases, nous avons voulu les contraintes permettant de représenter exactement le nombres de gardes et de civils sur le plateau. Cependant, le plateau pouvant être de taille relativement grande, la combinatoire fait exploser le nombre de contraintes (elles sont générées par la fonction : **create_npc_constraints(nbGuards: int, nbCivils: int) -> ClauseBase**). Le temps de recherche de solution par le solveur ainsi que le temps d'écriture des clauses sur un fichier dimacs augmentaient de façon exponentielle. (Exemple : 7min30 sur une matrice 6x7 avec ces contraintes, contre 30 secondes sans). Il a donc fallu trouver un compris. Nous avons fait le choix de ne pas introduire ces contraintes puisqu'elles ne sont pas indispensables, et les gains en temps et en énergie sont nettement plus favorables. Elles ne sont pas indispensables puisque la déduction d'un garde reste toujours possible grâce aux contraintes initiales, à la vision et à l'écoute. De plus, une fois que tous les gardes ont été vus ou déduits, le solveur n'a pas besoin d'être utilisé pour la détection de gardes.

Notre algorithme n'effectue également pas de test de déduction sur la cible, la corde et le déguisement. En effet, ces éléments ne sont déductibles que si l'ensemble des cases sont connues, sauf une seule, celle qui contient l'éléments. Cependant, il est rare que ces éléments soient les derniers trouvés, et généralement, la dernière case est à proximité (selon notre algorithme). Ainsi, nous avons préféré la rapidité et l'utilisation minime de ressources, face à la gestion de cas exceptionnellement rare, au risque de prendre quelques pénalités (déplacement supplémentaire).

En revanche, un point fort de notre algorithme est la recherche du chemin vers une case inconnue. En effet, nous effectuons une estimation du coût du trajet (en fonction des éléments déjà connus, du nombre de mouvements requis…) et récupérons le trajet le moins coûteux. Ainsi, nous sommes capables de trouver le meilleur chemin, en traversant une zone connue, jusqu'à une zone inconnue.

Phase 2 | phase2.py :

Modélisation STRIPS

Pour pouvoir implémenter la phase 2, nous devons d'abord effectuer une modélisation STRIPS qui facilitera l'implémentation de l'algorithme A*.

On peut supposer : x,y des coordonnées, et o une orientation (N,S,E,O)

Fluents :

- **at(x)** : hitman se trouve en x
- **empty(x)** : La case en x est vide
- **loot_at(x)** : Hitman regarde vers x
- **has_wire()** : si vrai, hitman a récupéré la corde
- **has_suit()** : si vrai, hitman a récupéré le déguisement
- **target_killed()** : si vrai, la cible a été tuée
- **is_invisible()** : si vrai, hitman est invisible (il porte le déguisement)
- **guard_view(x,y)** : le garde à la case x voit la case y
- **civil_view(x,y)** : le civil à la case x voit la case y
- **guard(x)** : le garde se trouve à la position x (disparait une fois tué)
- **civil(x)** : le civil se trouve à la position x (disparait une fois tué)
- **suit(x)** : le déguisement se trouve à la position x (disparait une fois récupérée)
- **wire(x)** : la corde se trouve à la position x (disparait une fois récupérée)

Prédicats :

- **target(x)** : la cible se trouve en x
- **wall(x)** : un mur se trouve à la position x
- **succView(o1,o2)** : la rotation à 90° de l'orientation o1 donne l'orientation o2
- **proximity(x,y,o)** : x est une case accessible, adjacente à la case y selon l'orientation o

But :

$$\text{target\_killed}() \wedge \text{at}(s)$$

(s : la position de départ)

Etat initial

L'état initial est déterminé à partir de l'arbitre et construit en fonction du plateau. Pour cela, nous récupérerons la carte visitée lors de la phase 1 (ou celle que renvoie l'arbitre), puis nous construirons l'état initail à partir des informations obtenues.

Actions :

- **turn_clockwise(o)** :
  - Préconditions :
  $$succView(o, w) \wedge \text{look\_at}(o)$$
  - Effets :
  $$\text{look\_at}(w) \wedge \neg \text{look\_at}(o)$$
- **turn_anticlock(x,y,o)** :
  - Préconditions :
  $$succView(w, o) \wedge \text{look\_at}(o)$$
  - Effets :
  $$\text{look\_at}(w) \wedge \neg \text{look\_at}(o)$$
- **move(x,y,o)** :
  - Préconditions :
  $$\text{at}(x) \wedge \text{empty}(y) \wedge \text{proximity}(x, y, o)$$
  - Effets :
  $$\text{at}(y) \wedge \neg \text{at}(x) \wedge \neg \text{empty}(y) \wedge \text{empty}(x)$$
- **kill_target(x,y)** : il faut avoir la corde de piano sur soi et être sur la case de la cible
  - Préconditions :
  $$\text{at}(x, y) \wedge \text{target}(x, y) \wedge \text{has\_wire}()$$
  - Effets :
  $$\text{target\_killed}()$$
- **kill_guard(x,o)** et **kill_civil(x,o)** : il faut regarder le garde ou le civil, être sur une case adjacente et qu'il ne vous regarde pas. La corde de piano n'est pas nécessaire. Une fois neutralisée, la personne n'apparaît plus dans les visions et sa case est libérée. :
  - Préconditions :
  $\text{at}(x) \wedge \text{look\_at}(o) \wedge \text{proximity}(x, y, o) \wedge \text{civil}(y) \wedge (x! = y) \wedge \text{civil\_view}(y, z) \wedge (x! = z)$
  - Effets :
  $$\neg \text{civil}(y) \wedge \neg \text{civil\_view}(y, z) \wedge \text{empty}(y)$$

  (On applique le même raisonnement pour kill_guard())
- **wear_suit()** :
  - Préconditions :
  $$\text{has\_suit}()$$
  - Effets :
  $$\text{is\_invisible}()$$
- **grab_suit()** : il faut être sur la case du costume :
  - Préconditions :
  $$\text{at}(x) \wedge \text{suit}(x)$$
  - Effets :
  $$\text{has\_suit}() \wedge \neg \text{suit}(x)$$
- **grab_weapon()** :
  - Préconditions :
  $$\text{at}(x) \wedge \text{wire}(x)$$
  - Effets :
  $$\text{has\_wire}() \wedge \neg \text{wire}(x)$$

Notre programme

Le programme commence par définir deux classes **Enum**, **Action** et **Orientation**, qui représentent respectivement les différentes actions possibles pour le personnage du jeu et les différentes orientations possibles. Les actions comprennent des mouvements tels que tourner à droite, tourner à gauche, se déplacer, ainsi que des actions spécifiques comme tuer une cible, tuer un civil, tuer un garde, attraper un costume, attraper un fil et porter un costume.

L'état initial du jeu est défini par la fonction initial_state, qui initialise l'état en fonction d'une carte donnée, d'une position de départ et d'une orientation de départ. L'état du jeu est représenté par un dictionnaire qui contient des informations sur la position actuelle du personnage, l'orientation, les objets qu'il possède, les cibles qu'il a tuées, et plus encore.

Le programme contient également plusieurs fonctions qui déterminent si une certaine action est possible dans l'état actuel du jeu. Par exemple, la fonction **turn_clockwise()** vérifie si le personnage peut tourner à droite, la fonction move vérifie si le personnage peut se déplacer, et ainsi de suite.

L'implémentation de l'algorithme A* est utilisée pour trouver le chemin optimal vers un objectif donné, en tenant compte de l'état actuel du jeu. L'objectif peut être d'attraper la corde, de tuer la cible, ou de fuir, et nous utilisons l'algorithme A* pour chacune de ces phases en calculant le chemin le moins couteux pour arriver jusqu'à notre objectif.

Enfin, la fonction launch_killing lance la phase 2 du jeu, qui consiste à atteindre différents objectifs en utilisant l'algorithme A* pour déterminer le meilleur chemin.

Qualités et défauts

Qualités :

- **Utilisation de l'algorithme A* :** Une des principales forces de notre programme est l'utilisation de l'algorithme de recherche informée A* pour déterminer le chemin optimal vers un objectif donné. Nous l'avons optimisé en divisant les objectifs en: Chercher la corde, tuer la cible et revenir à la case départ. Cet algorithme est largement reconnu pour sa performance et son efficacité dans la résolution de problèmes de recherche de chemin, et son utilisation dans notre programme a permis d'obtenir des résultats de haute qualité.
- **Flexibilité :** Le programme est conçu de manière modulaire, avec des fonctions distinctes pour chaque action possible et pour l'initialisation de l'état du jeu. Cela rend le code plus lisible et plus facile à maintenir. De plus, cette conception permet une grande flexibilité, car il est facile d'ajouter, de modifier ou de supprimer des actions.

Défauts :

- **Gestion des actions impossibles :** Un des défauts de notre programme est la manière dont il gère les actions qui ne sont pas possibles dans l'état actuel du jeu. Actuellement, le programme vérifie simplement si une action est possible avant de la réaliser. Cependant, il pourrait être plus efficace d'implémenter une sorte de mécanisme de replanification ou de récupération pour gérer les situations où une action prévue devient impossible.
- **Optimisation de l'algorithme A'*' :** Bien que l'algorithme A* soit généralement efficace, il y a toujours des possibilités d'optimisation. Par exemple, l'utilisation de heuristiques plus sophistiquées pourrait améliorer la performance de l'algorithme dans certains cas.

Fonctionnement du projet | main.py

Modules utilisés

Voici la liste des modules utilisés dans le projet.

```python
from itertools import combinations
from pprint import pprint
from typing import *
import subprocess
from enum import Enum
import copy
```

Après vérification sur les ordinateurs de l'école, l'ensemble des modules sont déjà installés.

Mise en route du programme

Il faut s'assurer que le solveur soit bien mis en place. Pour notre part, nous avions placé l'exécutable dans le dossier, et cela fonctionnait !

Comme démontré précédemment, le projet fonctionne en deux phases.

Phase 1

Pour lancer la phase une, il suffit simplement d'appeler la fonction ***phase1_run*** dans le fichier *main.py*. Cette fonction récupère l'état initial de l'arbitre (et donc du jeu, situé dans *hitman.py*), et lance l'exploration par la fonction ***launch_exploration(status,hr)*** (*phase1.py*). Cette fonction fait appel à la fonction ***init_exploration*** (*phase1.py*) qui génère les contraintes SAT du plateau de jeu. Ensuite, l'exploration débute en faisant appel à la fonction ***explore*** (*phase1.py*).

Phase 2

Une fois la phase une terminée, nous pouvons lancer la phase 2 en récupérant la carte. L'arbitre et la carte sont transmis à la fonction ***phase2_run*** (*main.py*) qui démarre la phase deux. Le statut initial de l'arbitre et la carte sont fournis à la méthode ***launch_killing***(*phase2.py*) qui lit les valeurs retournées et crée l'état initial par la fonction ***initial_state*** (*phase2.py*). Une fois l'état initial élaboré, on lance l'algorithme A* pour la recherche de la corde de piano et ainsi de suite.

Programme complet

Ainsi, pour faire tourner notre algorithme sur une carte, il suffit simplement de d'associer une carte à la variable ***world_example*** du fichier *hitman.py* et d'exécuter le fichier *main.py*.
Nous avons réutiliser la fonction ***main()*** fournie, puisque nous l'avons trouvée très pertinente.