Rasende Roboter

Projet IA41

Noa FOUICH
Corentin HAUTEFAYE
William LE GALLOU
Kenneth SOARES

Janvier 2026

Introduction

Dans le cadre de l'UE IA41 (*Semestre A25*), nous avons réalisé une implémentation du jeu de décision *Rasende Roboter*. L'objectif de ce projet est de programmer un système capable de fournir une solution aux configurations proposées, autrement dit de permettre à un utilisateur de jouer des parties contre l'ordinateur.

Table des matières

1 Présentation générale

1.1 Rappel de l'énoncé du sujet

Permettre à l'utilisateur de jouer des parties de Rasende Roboter contre l'ordinateur. L'utilisateur devrait pouvoir choisir au début de la partie la force de l'ordinateur.

1.2 Conventions

Nous avions utilisé plusieurs conventions de code, afin de le clarifier pour tous ses utilisateurs et de le rendre professionnel :

- le nommage de variables et de constantes formalisé (sens des noms, minuscules, majuscules, tirets du bas...)
- l'ajout de commentaires en anglais pour de nombreux blocs de code
- l'organisation structurée à l'aide de classes et de fichiers appropriés
- les éléments supplémentaires pour la compréhension du code comme le `README`

1.3 Outils utilisés

Pour réaliser ce projet, les outils suivants ont été utilisés :

- **Python 3**
- **PyGame** pour l'interface graphique
- **Git** et **GitHub** pour le contrôle des versions
- **LaTeX** pour la rédaction du rapport

2 Spécification du problème

Dans cette section, on cherche à généraliser et formaliser une partie de jeu.

2.1 Notations

Soit $k \in \mathbb{N}\backslash\{0;1\}$. Soit $n \in \mathbb{N}$ tel que $n \geq k+1$. Afin de représenter le plateau de jeu, on définit les notations suivantes :

- $B = [\![1;n]\!]^2$ une grille de $n \times n$ cases.
- Pour tout $i \in [\![1;k]\!], r_i = (x_i, y_i) \in B$ représente le i-ème robot.
- $C = \left\{ \{(x,y),(x',y')\} \;\middle|\; \begin{cases} (x,y) & \in B \\ (x',y') & \in B \\ |x-x'|+|y-y'| & = 1 \end{cases} \right\}$ l'ensemble des collisions entre les cases de B, deux à deux.
- $D = \{(1,0),(-1,0),(0,1),(0,-1)\}$ l'ensemble des directions cardinales.

2.2 Espace d'états

Définition 1. On définit un *état du jeu* par tout k-uplet de B. L'ensemble de tous les états possibles, noté $\mathcal{S}$, correspond à l'ensemble des positions occupées par les k robots, d'où $\mathcal{S} = B^k$.

Il est important de noter que de ce fait, on obtient $\text{Card}(\mathcal{S}) = n^{2k}$. Or, la définition proposée n'impose pas à un état d'être valide. En effet, il est impossible que deux robots occupent la même case, d'où :

Définition 2. On définit *l'ensemble des états valides*, ou *espace d'états*, et on note $\mathcal{S}_v = \left\{ (r_1,\ldots,r_k) \in B^k \;\middle|\; \forall (i,j) \in [\![1;k]\!]^2, (i \neq j) \implies (r_i \neq r_j) \right\}$. Un élément de $\mathcal{S}_v$ est dit *valide*.

L'inclusion $\mathcal{S}_v \subset \mathcal{S}$ est ainsi triviale.

Proposition 1. *On a* $\text{Card}\,(\mathcal{S}_v) = \frac{(n^2)!}{(n^2-k)!}$.

Démonstration. On cherche le nombre de k-uplets injectifs $(r_1,\ldots,r_k) : [\![1;k]\!] \to B$. Formellement, chaque fonction $f : [\![1;k]\!] \to B$ qui est injective correspond à un état valide. Or, le nombre de fonctions injectives de $[\![1;k]\!]$ dans B est exactement le nombre d'arrangements de k parmi n^2 éléments.

D'où, la conclusion. □

Remarque 1. On obtient ainsi le fait que l'espace d'états croît très rapidement en fonction de n et k.

2.3 Fonction de transition

Définition 3. On définit la *fonction de transition* pour passer d'un état valide à un autre, d'où :

$$\begin{array}{rll} \delta : & \mathcal{S}_v \times [\![1;k]\!] \times D & \to \mathcal{S}_v \\ & ((r_1,\ldots,r_k),i,d) & \mapsto (r_1,\ldots,r_i+dt,\ldots,r_k) \end{array}$$

$$\text{avec } t = \min\left\{ s \in \mathbb{N}^* \;\middle|\; \begin{array}{ll} & \exists j \in [\![1;k]\!],\ r_i + d(s+1) = r_j \\ \text{ou} & \{r_i + ds, r_i + d(s+1)\} \in C \end{array} \right\}$$

Remarque 2. Comme pour le passage d'un état à un autre, il est facile de trouver les variables i et d, il est recevable de simplifier l'écriture de $\delta(S,i,d)$ par $\delta(S)$.

Proposition 2. *À partir d'un état, δ peut engendrer* $\mathrm{Card}(D) \times k$ *états valides.*

Définition 4. Soit $t \in [\![1;k]\!]$ et $S_0 = (r_1,\ldots,r_k) \in \mathcal{S}_v$ l'état initial. Soit $g \in B$ une case. On dit que g est *atteignable* depuis S_0 par r_t ssi il existe une suite finie de $m+1 \in \mathbb{N}$ états $(S_i)_{i\in[\![0;m]\!]}$ telle que pour tout $i \in [\![1;m]\!], \delta(S_{i-1}) = S_i$ et $(S_m)_t = g$ où $(S_m)_t$ désigne la position du t-ième robot dans l'état considéré.

On définit ainsi l'ensemble G des cases objectifs jouables par :

$$G = \left\{ g \in B \mid \exists m \in \mathbb{N}, \exists (S_i)_{i\in[\![0;m]\!]}, \left\{ \begin{array}{ll} \forall i \in [\![1;m]\!], & \delta(S_{i-1}) = S_i \\ (S_m)_t = g & \end{array} \right. \right\}$$

Définition 5. On appelle *configuration de jeu* tout sextuplet

$$(k,n,t,C,R,g)$$

où :
- k est le nombre de robots,
- n est la taille du plateau,
- $t \in [\![1;k]\!]$ est l'indice du robot cible,
- R est l'ensemble des positions initiales des robots,
- C est l'ensemble des obstacles entre cases adjacentes,
- $g \in G$ est la case objectif du robot cible.

Définition 6. Soit $(t,g,R) \in [\![1;k]\!] \times G \times B^k$. Soit $\mathcal{C} = (k,n,t,C,R,g)$ une configuration de jeu et $S \in \mathcal{S}_v$. S est dit *terminal* ssi $(S)_t = g$. On note l'ensemble des états terminaux par $\mathcal{T}$.

Il est ainsi clair que $\mathcal{T} \subseteq \mathcal{S}_v$.

3 Analyse du problème

À partir de la formalisation précédente, nous étudions maintenant la nature du problème de recherche induit, ses contraintes en termes de calculs, ainsi que les approches algorithmiques envisageables pour le résoudre.

3.1 Nature et caractéristiques du problème

Le jeu *Rasende Roboter* est modélisable par un problème de recherche dans un espace d'états, certes fini mais très vaste. Chaque état valide correspond à un sommet du graphe orienté engendré, et chaque application de la fonction de transition δ définit alors une arête entre deux nœuds.

L'objectif est de trouver pour une configuration de jeu donnée, une suite de transitions menant d'un état initial $S_0 \in \mathcal{S}_v$ à un état terminal $S_G \in \mathcal{T}$, tout en minimisant le nombre de coups effectués par des déplacements de robots. En sus d'être un problème de décision, il s'agit également d'un problème d'optimisation.

À partir de ce paragraphe, on considère $n = 16$ et $k = 4$. Les autres variables restent quantifiées de la même manière.

3.2 Contraintes et difficultés

D'après la proposition 1, l'espace d'états $\mathcal{S}_v$ est de cardinal $\frac{256!}{252!} = 253 \times 254 \times 255 \times 256 \approx 4,2 \times 10^9$, d'où un nombre d'états presque impossible à gérer.

De plus, le facteur de branchement est important : à chaque état, il est possible de déplacer chacun des k robots dans les quatre directions cardinales D, tant que le déplacement est valide. Ainsi, chaque nœud a au plus 16 enfants (*d'après la proposition 2*).

Ces caractéristiques rendent toute exploration exhaustive de l'espace d'états rapidement in-tractable sans stratégie de recherche adaptée.

3.3 Graphe implicite

Le graphe n'est jamais construit explicitement en raison de sa taille combinatoire. Les successeurs d'un état sont générés dynamiquement à l'aide de la fonction de transition δ lors de l'exploration. C'est pour cela que l'on parle de *graphe implicite*.

Ce dernier est ainsi :

— Fini
— Orienté
— Non pondéré (*on considère qu'une transition a un coût unitaire*)
— Fortement connexe par composantes
— De facteur de branchement borné par $\text{Card}(D) \times k$ (*ici par* 16)

3.4 Approches algorithmiques

La résolution du problème peut être abordée par des algorithmes de parcours de graphes orientés. On distingue ainsi deux familles :

3.4.1 Recherche non informée

Les algorithmes de recherche non informée, tels que le parcours en largeur (*BFS*), explorent l'espace d'états sans utiliser d'information spécifique au problème. BFS garantit la découverte d'une solution optimale en nombre de coups, au prix d'une consommation mémoire importante.

3.4.2 Recherche informée

Les algorithmes de recherche informée, comme A*, exploitent une heuristique estimant la distance restante jusqu'à l'objectif. Cette approche permet de réduire considérablement le nombre d'états explorés, tout en conservant l'optimalité pourvu que l'heuristique soit admissible.

3.4.3 Critères de choix d'une approche

Le choix de l'algorithme dépend essentiellement de :
— La taille de l'espace d'états
— L'optimalité de la solution à trouver
— Des complexités spatiales et temporelles

4 Traitement du problème

4.1 Algorithmes étudiés

Compte tenu de la formalisation précédente du problème comme un parcours de graphe implicite d'états, plusieurs algorithmes de recherche peuvent être envisagés. Parmi eux, nous avons retenu le parcours en largeur et l'algorithme A*, qui présentent des garanties d'optimalité adaptées dans ce contexte.

Dans cette section, nous justifions l'admissibilité de ces deux algorithmes pour la résolution du problème considéré. Nous analysons ensuite leurs propriétés théoriques, notamment en termes d'optimalité, ainsi que de complexité temporelle et spatiale, afin de mettre en évidence leurs avantages et leurs limites respectives.

4.1.1 Parcours en largeur

Proposition 3. *Soit G un graphe. Un parcours en largeur retourne une solution optimale ssi*

1. *G est non pondéré*
2. *Chaque transition correspond à un coût unitaire*
3. *G est exploré sans revisiter les états déjà rencontrés*

Notons que les hypothèses énoncées dans la proposition 3 sont rencontrées pour ce problème.

Proposition 4. *Le parcours en largeur trouve une solution optimale en nombre de coups lorsqu'elle existe.*

Démonstration. Considérons le graphe orienté implicite $G = (\mathcal{S}_v, E)$, où pour tout $(S, S') \in \mathcal{S}_v^2$, $(S, S') \in E \iff S' = \delta(S)$ pour une action valide.

Soit $S_0 \in \mathcal{S}_v$ l'état initial. Le parcours en largeur explore les sommets de G par couches successives, où la couche $d \in \mathbb{N}^*$ contient exactement les états atteignables depuis S_0 par une suite de d transitions.

Soit $S_G \in \mathcal{T}$ un état terminal tel que la distance minimale entre S_0 et S_G soit $d^\star$. Par construction, BFS explore tous les états à distance strictement inférieure à $d^\star$ avant d'explorer ceux à distance $d^\star$.

Ainsi, lors de la première rencontre d'un état terminal S_G, le chemin construit comporte exactement $d^\star$ transitions, ce qui est minimal.

Par conséquent, BFS retourne une solution optimale en nombre de coups. □

Complexité temporelle Un parcours en largeur dans un graphe $G = (V, E)$ a une complexité en $\mathcal{O}(|V| + |E|)$. On admet tout d'abord que $\text{Card}(\mathcal{S}_v) \approx (n^2)^k$. Ensuite, on sait que le facteur de branchement est majoré par $4k$. Ainsi, pour ce problème, BFS est en $\mathcal{O}(k(n^2)^k)$. On remarque ainsi que le jeu est en complexité exponentielle en k et polynomiale en n ; on en conclut que plus il y a de robots, plus il devient difficile de trouver une solution en un temps raisonnable.

Complexité spatiale Un parcours en largeur dans un graphe $G = (V, E)$ a une complexité en mémoire en $\mathcal{O}(|V|)$. On a donc ici une occupation de la mémoire en $\mathcal{O}((n^2)^k)$. Les mêmes remarques s'appliquent que pour le paragraphe précédent.

4.1.2 Algorithme A*

Proposition 5. *A* est optimal ssi*

1. *Tous les coûts d'arêtes sont strictement positifs*
2. *L'heuristique h est admissible, i.e $\forall S \in \mathcal{S}_v, h(S) \leq h^\star(S)$ où $h^\star$ représente le coût minimal réel restant pour aller à un état terminal*

Démonstration. Soit $S_0 \in \mathcal{S}_v$ l'état initial. Soit $S_G \in \mathcal{T}$ un état terminal atteignable depuis S_0, et soit $c^\star$ le coût minimal pour aller de S_0 à S_G.

Soit $(g, h) \in (\mathcal{F}(\mathcal{S}_v, \mathbb{N}))^2$, avec pour tout $S \in \mathcal{S}_v$, $g(S)$ et $h(S)$ représente respectivement le coût exact de S_0 vers S et l'heuristique évaluée en S, supposée admissible. On peut ainsi définir la fonction $f : \mathcal{S}_v \to \mathbb{N}$ telle que $f = g + h$.

Pour tout état $S \in \mathcal{S}_v$, on a donc :

$$f(S) \leq g(S) + h^\star(S)$$

Supposons par l'absurde que A* retourne une solution $S \in \mathcal{S}_v$ de coût $c \in \mathbb{N}$ tel que $c > c^\star$ avant d'explorer un chemin optimal. Alors, il existe un état $S' \in \mathcal{S}_v$ sur un chemin optimal tel que

$$g(S') + h(S') \leq g(S') + h^\star(S') < c$$

D'où :

$$f(S') \leq c^\star < c$$

ce qui implique que S' aurait dû être extrait de la file de priorité avant S, ce qui est absurde.

Ainsi, le premier état terminal extrait par A* correspond nécessairement à un chemin de coût minimal. □

Complexité temporelle A* ici est en $\mathcal{O}(k(n^2)^k)$. Les mêmes remarques qu'au paragraphe 4.1.1 s'appliquent. Néanmoins, notons qu'une bonne heuristique permet d'évincer l'exploration d'états peu intéressants.

Complexité spatiale A* occupe ici un espace en $\mathcal{O}((n^2)^k)$. Les mêmes remarques qu'au paragraphe 4.1.1 s'appliquent.

4.1.3 Heuristiques étudiées

Dans cette partie, on présente plusieurs heuristiques auxquelles on pourrait penser afin de guider notre recherche dans $\mathcal{S}_v$.

Heuristique nulle Prenons pour tout $S \in \mathcal{S}_v, h(S) = 0$. Cette heuristique est bien admissible car elle ne surestime pas le coût réel pour passer d'un état initial à l'état final. Néanmoins, on note qu'elle n'est pas informative et de ce fait, A* dégénère et parcours l'ensemble de l'espace d'états jusqu'à trouver la solution optimale. **Inutilisable en pratique**.

Remarque 3. Notons toutefois qu'un parcours en largeur est un cas particulier de A* avec une heuristique nulle, et des coûts unitaires.

Distance de Manhattan Soit $i \in [\![1; k]\!]$ l'indice du robot cible. Prenons pour tout $(S, g) \in \mathcal{S}_v \times G, h(S) = d_1((S)_i, g)$, avec $\begin{array}{rccc} d_1 : & \mathbb{R}^2 \times \mathbb{R}^2 & \to & \mathbb{R}^+ \\ & ((x,y),(x',y')) & \mapsto & |x-x'|+|y-y'| \end{array}$. Cette application semble de prime abord assez naturel, cependant on rencontre rapidement un problème. Supposons que j'ai un pion à 4 cases de l'objectif, ainsi on aura $h(S) = 4$ pour cet état S. Or, d'après δ, il ne suffit que d'un coup pour arriver à l'objectif. Ainsi, elle surestime le coût réel. Donc **cette heuristique n'est pas admissible**.

Heuristique naïve Soit $i \in [\![1; k]\!]$ l'indice du robot cible. Prenons pour tout $(S = ((x_j, y_j))_{j\in[\![1;k]\!]}, (x_g, y_g)) \in \mathcal{S}_v \times G, h(S) = \begin{cases} 0 & \text{si } S \in \mathcal{T} \\ 1 & \text{si } x_i = x_g \text{ ou } y_i = y_g \\ 2 & \text{sinon} \end{cases}$. Cette heuristique est bien admissible, mais reste peu informative. Elles reste malgré tout meilleure que BFS. **Utilisable, mais peu informative**.

Heuristique par "glissades" Soit $i \in [\![1; k]\!]$ l'indice du robot cible. Soit $g \in G$ la case d'arrivée. On construit une heuristique h au début de chaque nouvelle partie à partir des règles suivantes :

1. On considère un plateau dénué de tous robots.
2. On définit la matrice $H = (\tilde{h}_{i,j})_{(i,j)\in[\![1;n]\!]^2} \in \mathcal{M}(\mathbb{N})$ tel que pour tout $(i, j) \in [\![1; n]\!]^2, \tilde{h}_{i,j} = \begin{cases} 0 & \text{si } (i,j) = g \\ +\infty & \text{sinon} \end{cases}$.
3. On note $p = (x_p, y_p)$ la case parent. On applique la fonction $\tilde{\delta}$ sur p pour obtenir l'ensemble E des cases atteignables depuis celle-ci.

$$\begin{array}{rccc} \tilde{\delta} : & \mathcal{S}_v \times [\![1; k]\!] \times D & \to & \mathcal{S}_v \\ & ((r_1, \ldots, r_k), i, d) & \mapsto & (r_1, \ldots, r_i + dt, \ldots, r_k) \end{array}$$

   avec $t = \min \left\{ s \in \mathbb{N}^* \;\middle|\; \{r_i + ds, r_i + d(s+1)\} \in C \right\}$
4. Pour tout $c = (x, y) \in E$, $\tilde{h}_{x,y} = \tilde{h}_{x_p,y_p} + 1$.
5. On applique les opérations 3 et 4 sur chacun des nœuds engendrés jusqu'à avoir traité tout le plateau.

On a ainsi pour tout $S = ((x_j, y_j))_{j \in [\![1;k]\!]} \in \mathcal{S}_v,\ h(S) = \tilde{h}_{x_i,y_i}$. Autrement dit, on a effectué un parcours en largeur depuis un état terminal afin d'évaluer à combien de déplacements une case se trouve de la case finale. Cette heuristique est bien sûr admissible, et n'est calculée qu'une seule en début de partie ($\mathcal{O}(n^2)$ *en création et* $\theta(1)$ *en accès*). **Utilisable et informative**.

4.1.4 Comparaisons

Critère	**BFS**	**A***
Optimalité	Oui	Si heuristique admissible
Temps	$\mathcal{O}(k(n^2)^k)$	Meilleure que BFS si h bien trouvé
Mémoire	Très élevée	Très élevée
Usage	Petits graphes	Espaces larges avec coûts variables

4.2 Organisation du programme

Fichiers Tous les scripts Python sont localisés dans le dossier `src`. Chaque classe se trouve dans son propre fichier. Le fichier `consts.py` contient les constantes utilisées à travers le projet. Le fichier `board.txt` contient une copie des collisions (*opérations bit à bit*) des cases du plateau, avec la première case en haut à gauche. Enfin le dossier `docs` contient les sources de ce rapport.

Moteur de jeu Notre projet utilise `pygame` pour l'interface graphique et la gestion des entrées. Le point d'entrée est à travers la fonction `main` dans le fichier éponyme. Le programme fonctionne ainsi de la manière suivante :

1. Initialisation de la fenêtre et des événements
2. Initialisation des éléments de jeu (*joueur, plateau, ...*)
3. Boucle principale
   a) Mise à jour des objets de jeu
   b) Rendu à l'écran
   c) Pause pour maintenir 60 images par secondes
4. Fermeture de la fenêtre & Nettoyage

Partie de jeu La boucle principale agit comme un automate fini simple, selon les états suivants (*définis dans* `consts.py`) :

1. **Initialisation de la partie**
   — Variables remises à zéro
   — Choix d'une nouvelle configuration de jeu
   — Pré-calcul de l'heuristique (*valeurs stockées dans un dictionnaire*)
2. **Tour du joueur**
   — Le joueur doit fournir une solution en moins de 60 secondes

3. **Fin de tour du joueur**
   — Affichage des résultats du tour
4. **Calculs de l'IA**
   — Application de l'algorithme de recherche dans un second processus
   — Si aucune solution n'est trouvée en 60 secondes, alors on passe aux résultats
5. **Tour de l'IA**
   — L'IA donne le nombre de coups qu'elle va faire
   — Elle affiche sa solution étape par étape
   — Les scores sont mis à jour ici sauf si aucune solution n'a été trouvée
6. **Affichage des résultats de partie**
   — Affichage des scores
   — Retour au premier état

5 Résultats

Nous avons ainsi testé les algorithmes (*et heuristiques pour A**) présentés en section 4.1.

5.1 Résultats obtenus

Parcours en largeur Pour une configuration de jeu quelconque, quand bien même l'algorithme est optimal, il ne termine pas.

A* avec heuristique nulle Mêmes résultats que pour le parcours en largeur.

A* avec heuristique naïve L'algorithme ne termine pas en des temps raisonnables mais reste néanmoins meilleur que BFS.

A* avec heuristique d_1 Bien que l'heuristique ne soit pas admissible, il s'agit de celle qui donne les meilleurs résultats en termes de complexité temporelle. Néanmoins, son comportement dépend fortement de la configuration. Ainsi, la solution optimale n'est pas en moyenne pas fournie.

A* avec heuristique par glissements Pour une configuration quelconque, l'algorithme met en moyenne entre 10 et 20 secondes à fournir la solution optimale. Il arrive toutefois qu'il mette plus de temps parfois (*quelques minutes*).

5.2 Interprétation des résultats

Les résultats expérimentaux obtenus confirment les analyses théoriques menées dans les sections précédentes, en particulier concernant la taille de l'espace d'états et le rôle déterminant des heuristiques dans la réduction de la complexité effective de la recherche.

Limites des approches non informées Le parcours en largeur, bien qu'optimal par construction, souffre d'une explosion combinatoire de l'espace d'états. Pour une configuration standard du jeu, toute exploration exhaustive reste irréaliste en pratique. A* avec l'heuristique nulle étant équivalent à BFS, il hérite strictement des mêmes limitations et ne termine pas non plus dans des délais raisonnables.

Apport partiel des heuristiques naïves L'heuristique naïve introduit une première forme de guidage de la recherche, ce qui permet de réduire le nombre d'états explorés par rapport à BFS. Toutefois, cette heuristique reste trop peu informative pour compenser la taille du graphe implicite. En conséquence, l'algorithme demeure trop lent pour résoudre une instance quelconque du problème.

Compromis performance/optimalité avec la distance de Manhattan L'heuristique d_1 se révèle être la plus efficace en pratique en termes de temps de calcul. Son coût d'évaluation constant et son pouvoir discriminant élevé permettent à A* de se concentrer rapidement sur des zones pertinentes de l'espace d'états. Cependant, cette heuristique n'étant pas admissible dans le cadre des déplacements par glissements, l'optimalité des solutions produites n'est pas garantie. De plus, les performances observées varient fortement selon la configuration initiale et la position de la cible.

Heuristique par glissements L'heuristique par glissements offre le meilleur compromis entre garanties théoriques et performances pratiques. En étant admissible, elle permet à A* de conserver son optimalité tout en réduisant drastiquement l'espace de recherche (*en général, on explore au plus quelques milliers d'états au lieu de milliards*). Les temps d'exécution observés, bien que significatifs, restent compatibles avec une résolution effective du problème. Les cas où le temps de calcul augmente sensiblement correspondent à des configurations particulièrement défavorables, confirmant le caractère intrinsèquement difficile du problème.

Utilisation de la mémoire Notons que peu importe la technique employée, la mémoire utilisée augmente exponentiellement. Cela constitue ainsi le défaut majeur des algorithmes présentés.

6 Retour d'expérience

6.1 Organisation

Nous avons utilisé **GitHub** afin de centraliser le code tout en assurant un suivi régulier du projet. Cela nous a ainsi permis de travailler sur une base commune une fois l'implémentation commencée.

De plus, nous avons cherché à nous placer dans le contexte du travail collaboratif en entreprise, et de ce fait, nous avons travaillé par *Pull Requests*. La branche `main` a été ainsi protégée par défaut des téléversements non approuvés au préalable.

En terme de tâches à effectuer, nous avions réparti le travail de manière relativement équilibrée.

6.2 Difficultés rencontrées

En terme d'organisation, nous avons trouvé la synchronisation entre nous particulièrement difficile concernant les *pull requests*. Ce n'était pas trop dérangeant pendant le semestre, mais bien plus pendant les périodes de congés. Cela a engendré du retard et des différences de quantité de travail entre les membres, qui a dû être rattrapé.

En ce qui concerne le code à produire, la fonction `astar` a dû être revue, afin de prendre en compte les positions des pions du plateau et ainsi pouvoir comparer le résultat trouvé par l'ordinateur à celui trouvé par le joueur. Nous n'avions pas suffisamment conçu l'ensemble du projet pour toujours être sur la même longueur d'onde.

Il n'y a eu aucun problème technique dû à un manque de compétences en python.

6.3 Suggestions d'amélioration

Tout d'abord, un point négatif est la gestion de la mémoire. En effet, il aurait été préférable d'utiliser un algorithme avec un $\mathcal{O}(n)$ en mémoire. Par exemple, un parcours en profondeur a cette complexité, néanmoins elle ne fournit jamais la solution optimale. Ainsi, peut-être pourrions-nous essayer d'élaguer certaines branches d'une exploration en profondeur en se basant sur le principe de l'heuristique ?

Il est également à noter que nous n'avons pas permis à l'utilisateur de changer le niveau de difficulté de l'IA. En effet, comme le problème revient à trouver l'existence d'une solution, il y aurait alors un choix à faire :

— Soit on considère le temps d'exécution comme niveau de difficulté (*plus l'IA est rapide, plus elle est dure*)

— Soit on considère l'optimalité de la solution

Cela revient en somme à jouer sur l'heuristique employée dans A*. Nous avons ainsi décidé que ce point était inutile.

7 Conclusion

Les résultats obtenus mettent en évidence que, pour le jeu *Rasende Roboter*, la faisabilité algorithmique repose essentiellement sur la qualité de l'heuristique utilisée. Les approches non informées sont rapidement dépassées par la taille de l'espace d'états, tandis que des heuristiques bien conçues permettent de rendre le problème tractable en pratique, au prix éventuel de l'abandon de l'optimalité ou d'un coût de calcul supplémentaire.

Cependant, ce problème déterministe en soulève un second. Que se passe-t-il si on considère une grille générée aléatoirement ? A-t-on nécessairement l'existence d'au moins une solution pour une instance donnée ? Ainsi, cela constitue un problème très compliqué.