



Rapport de projet - Semestre 6 - Groupe 15528

mai 2022

---

# Robot Programming

---

Filière Informatique - ENSEIRB-MATMECA

Auteurs :

KHIARI Skander  
KABBOU Meryem  
MEDGHALI Majid  
OUALI Mohammed

Encadrants :

M.MORANDAT Floréal  
M.RENAULT David

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Contexte et présentation . . . . .	3
1.2	Organisation et outils de travail . . . . .	3
<b>2</b>	<b>Problématique</b>	<b>3</b>
2.1	Description du puzzle . . . . .	3
2.2	Récupération du puzzle et de la solution . . . . .	3
2.3	Vérification de la solution et du puzzle . . . . .	3
2.4	Implémentation d'un interpréteur . . . . .	4
2.5	Vérification de la fin d'exécution . . . . .	4
<b>3</b>	<b>Cadre du travail</b>	<b>4</b>
3.1	Graphe de dépendance . . . . .	4
3.2	Architecture des fichiers et détails techniques . . . . .	5
<b>4</b>	<b>Mécanisme du jeu</b>	<b>5</b>
4.1	Implémentation du programme et les instructions . . . . .	5
4.2	Implémentation de la boucle principale . . . . .	5
4.3	Visualisateur . . . . .	6
<b>5</b>	<b>Problèmes intéressants rencontrés et solutions algorithmiques</b>	<b>7</b>
5.1	Utilisation du context et récupération des erreurs . . . . .	7
5.2	Implémentation de la pile d'appel . . . . .	7
5.3	Affichage terminal . . . . .	8
<b>6</b>	<b>Description des tests du projet</b>	<b>8</b>
6.1	Les tests du projet . . . . .	8
6.2	Les tests d'exécutions . . . . .	9
<b>7</b>	<b>Discussion</b>	<b>10</b>
7.1	Limites et améliorations possibles . . . . .	10
7.2	Problèmes rencontrés . . . . .	10
<b>8</b>	<b>Conclusion</b>	<b>11</b>

## 1 Introduction

### 1.1 Contexte et présentation

Ce projet s'est déroulé dans le cadre d'une évaluation pour le semestre 6. Le sujet repose sur un petit robot chargé de ramasser des étoiles disséminées sur une grille colorée. L'attendu principal est d'implémenter des fonctions qui permettent d'atteindre des objectifs sur des puzzles différents et d'afficher le trajet d'un robot lors de sa recherche.

### 1.2 Organisation et outils de travail

Pour pouvoir développer un programme fonctionnel dans le temps imparti, nous avons à chaque instant, attribué à un membre de groupe une tâche précise. et pour l'échange des fichiers, on a utilisé un dépôt git. Cet outil nous permet par ailleurs en cas de problème avec la dernière version de revenir à la précédente.

LaTeX permet une mise en page claire et une facilité d'écriture de documents scientifiques. Nous avons ainsi choisi de l'utiliser pour la rédaction de notre rapport.

## 2 Problématique

### 2.1 Description du puzzle

Un puzzle est un objet dont les champs représentent des informations utiles sur le puzzle : id, title, robotRow, robotCol, robotDir, subs et board ces champs sont donnés par le sujet et on a ajouté un champ **stars** qui correspond au nombre des étoiles du puzzle. Pour pouvoir récupérer les informations d'un puzzle, on a implémenté des fonctions getPuzzleFrmId et getPuzzleFrmName qui nous permettent de récupérer un puzzle à partir de son nom ou son id.

### 2.2 Récupération du puzzle et de la solution

L'utilisateur peut entrer un puzzle et une solution de son choix à partir de la ligne de commande, deux cas sont possibles :

- Soit l'utilisateur fait entrer, dans la ligne de commande, le nom d'un puzzle déjà défini dans le fichier **puzzles.js** et puis un fichier **.json** contenant la solution dans un champ sol d'un objet. Comme cela on aura juste à récupérer le puzzle choisi à l'aide de la fonction getPuzzleFrmName mentionnée dans la partie ci-dessus.
- Soit la solution et le puzzle sont donnés dans un fichier **.json**, passé en ligne de commande, qui contient un objet avec deux champs : sol pour la solution et puzzle pour le puzzle.

La récupération des éléments d'un fichier **Json** a été faite à l'aide de **readFileSync()**, du module **fs** qui permet de lire le fichier et renvoyer son contenu, et de la fonction **Json.parse**, qui construit l'objet décrit par la chaîne de caractère passée en paramètre.

### 2.3 Vérification de la solution et du puzzle

nous avons implémenté plusieurs fonctions pour vérifier les erreurs statiques, c'est-à-dire si le puzzle et la solution sont valides. On vérifie la validité de la solution à partir de sa forme donnée par le champ **subs** du puzzle et la validité du puzzle à partir de sa forme générale.

## 2.4 Implémentation d'un interpréteur

Après avoir récupéré le puzzle et la solution, nous devions appliquer cette dernière sur le puzzle en parcourant le programme de la solution et en appliquant chaque instruction sur le puzzle dans le cas où le prédictat "c'est-à-dire la couleur de la case du puzzle" est vérifié, sinon on passe à l'instruction suivante si elle existe.

## 2.5 Vérification de la fin d'exécution

Le contrôle des itérations de la boucle principale nécessite de coder la fonction gameOver qui nous permet à la fin de chaque itération de savoir si la partie est terminée ou non. Cette fonction traite toutes les erreurs dynamiques possibles et vérifie si le robot a atteint son objectif qui est la collection de toutes les étoiles présentes dans le puzzle, d'où la nécessité d'ajouter dans le puzzle le champ **stars** représentant le nombre des étoiles.

Les erreurs dynamiques traitées pour savoir la fin de la partie sont les suivantes : si le robot sort de la trajectoire ou sort carrément du puzzle donné, si le robot arrive à gagner toutes les étoiles du puzzle ou si on a une boucle infinie qu'on définit par un nombre d'itérations supérieur ou égal à 500.

## 3 Cadre du travail

### 3.1 Graphe de dépendance

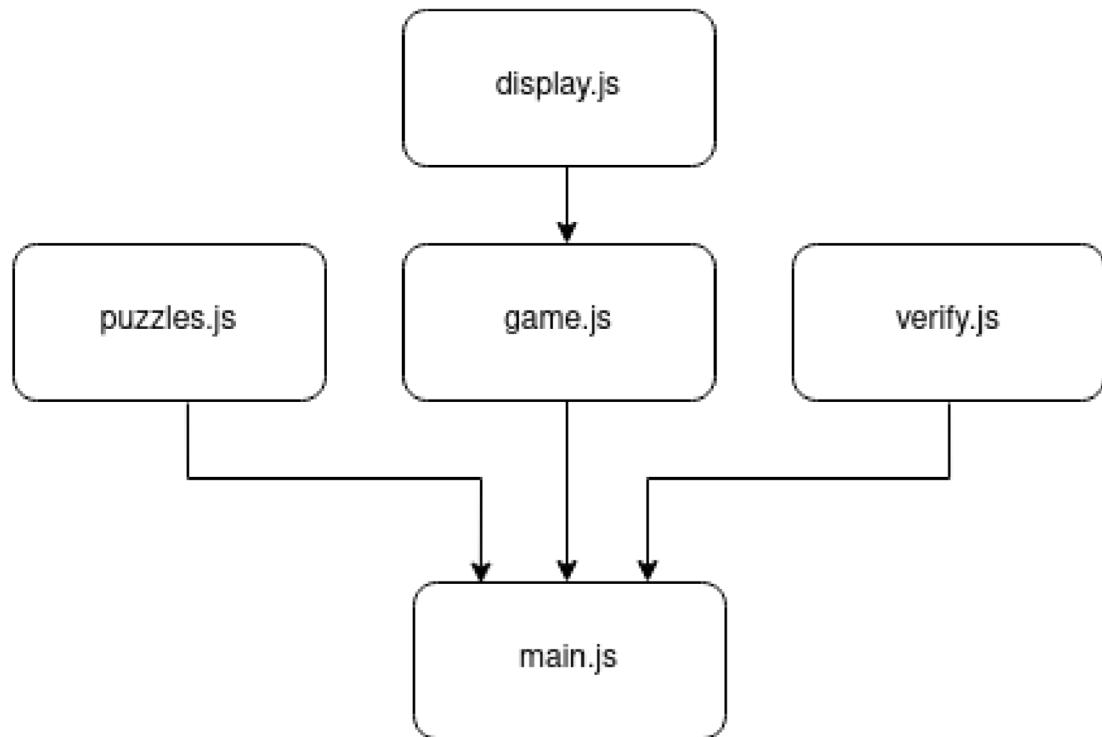


FIGURE 1 – Diagramme de dépendance

Le diagramme ci-dessus 1 fait état des dépendances des fichiers de notre code.

Le fichier `main.js` contient la boucle principale du projet, qui récupère le puzzle et la solution et l'exécute sur ce dernier.

Nous avons choisi de faire des fichiers séparés pour chaque module donc on a fait un fichier `display.js` pour l'affichage du puzzle, `verify.js` pour vérifier la solution et le puzzle, `game.js` qui contient les fonctions utiles pour exécuter un programme sur un puzzle et `puzzles` qui contient des exemples de puzzles avec leurs solutions (utiles pour les tests).

### 3.2 Architecture des fichiers et détails techniques

Les fichiers dans le dépôt sont organisés de la manière suivante :

```

/
    -- la racine du répertoire du projet
Makefile      -- Makefile global
README.md     -- fichier qui contient les instructions d'exécution
/src          -- fichiers sources et tests(.js)
/public        -- fichiers Web
/rapport       -- Rapport du projet (.tex/pdf)

```

Le fichier `Makefile` fournit :

- une règle `gen` qui génère la page web avec parcel .
- une règle `test` qui exécute les tests.

## 4 Mécanisme du jeu

### 4.1 Implémentation du programme et les instructions

Pour qu'un robot résout un puzzle, il doit exécuter un programme sur un puzzle, ce programme est implémenté sous forme d'un tableau de fonctions de 1 à N. Chaque fonction contient des instructions constituées de deux champs [ 'ACTION' , 'PRÉDICAT' ], l'action correspond soit au mouvement du robot, soit à l'appel d'une autre fonction et le prédicat correspond à la couleur nécessaire pour appliquer le premier champ.

Pour pouvoir exécuter une instruction, nous avons implémenté la fonction `executeInstOnPuzzle` qui étant donné un couplet [ 'ACTION' , 'PRÉDICAT' ] et un context C<sup>1</sup> applique l'action de cette dernière si le prédicat est vérifié. Si oui, on appelle la fonction `action` qui à son tour appelle la fonction dédiée à l'instruction qui lui est passée en paramètre.

### 4.2 Implémentation de la boucle principale

Après la récupération du puzzle et d'un programme, il fallait bien vérifier qu'ils sont valides. En effet, deux fonctions `verifyPuzzle` et `verifyProgram` assurent que le puzzle et de programme sont utilisables.

Après la vérification et pour exécuter le programme sur le puzzle, on a décidé d'implémenter une boucle de jeu à l'aide de la fonction `setInterval` qui déclenche répétitivement la même action à intervalles réguliers, la boucle de jeu appelle répétitivement une fonction `moveRobot`

---

1. le contexte est objet défini comme suit :  
`context = {"puzzle":puzzle,"prog":prog,"stack":[[0,0]],"pc":0,"num":0,"error":'NONE'};`

qui est une fonction d'ordre supérieur qui prend en paramètres le contexte et une fonction d'affichage pour visualiser les résultats, le contexte qui décrit l'état actuel du jeu sera changé après chaque itération ou bien après chaque exécution de la fonction `moveRobot`.

En plus, après chaque itération, on vérifie la fin de l'exécution à l'aide de la fonction `gameOver`.

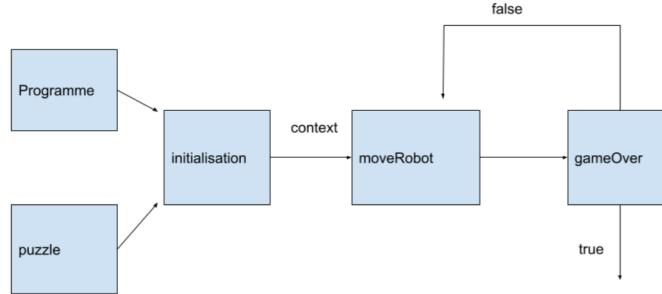


FIGURE 2 – La représentation de la boucle principale

### 4.3 Visualisateur

Pour visualiser correctement l'exécution de notre programme, on a eu l'idée d'implémenter un visualiseur en mode terminal qui affiche le puzzle et le robot à chaque itération. En effet, une fonction `display` dans le fichier.

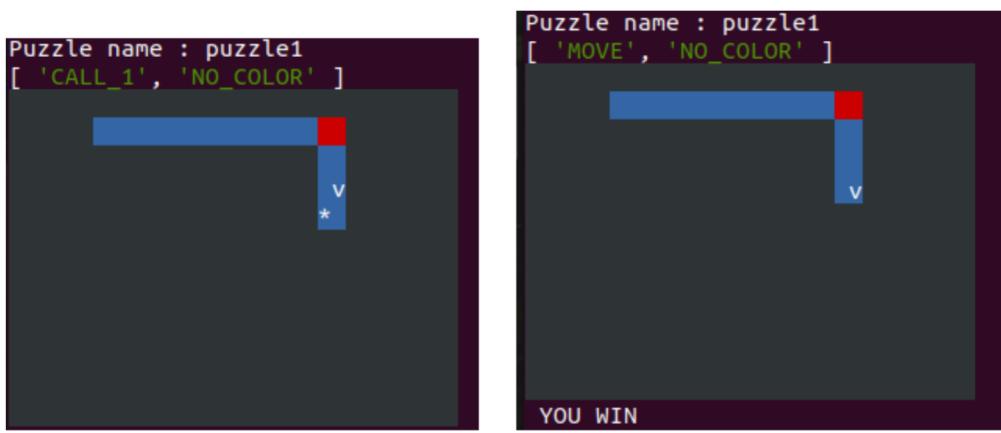


FIGURE 3 – L'affichage de l'évolution du robot en mode terminal

`display.js`. En plus de ça une fonction `term` qui sert à afficher le puzzle en plus de résultat d'exécution et le message d'erreur.

Une partie de sujet propose de créer une page web pour visualiser l'exécution de programme sur les puzzles. Sur notre interface, on a implémenté un ensemble de puzzles et l'utilisateur doit insérer les instructions et les prédictats nécessaires pour gagner la partie.

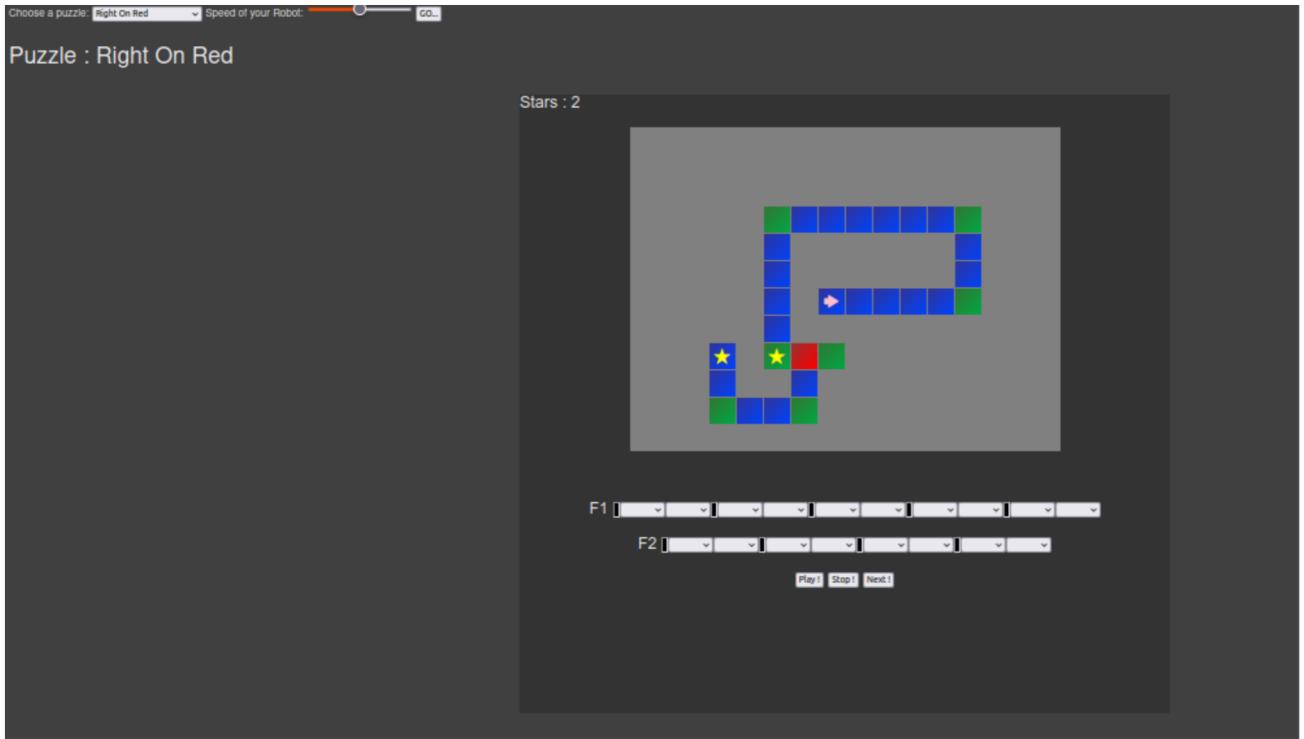


FIGURE 4 – La représentation en mode HTML

## 5 Problèmes intéressants rencontrés et solutions algorithmiques

### 5.1 Utilisation du context et récupération des erreurs

Pendant le déroulement du projet, nous nous sommes trouvés avec des fonctions dont les paramètres sont nombreux. D'où l'idée de créer une structure **context** qui regroupe les éléments suivants : "puzzle" correspondant au puzzle concerné, "prog" correspondant à la solution, "stack" correspondant à la pile, "num" correspond au nombre d'itérations de la boucle principale.

Pour tester si les erreurs dynamiques et statiques sont bien révélées par nos fonctions, nous avions besoin de récupérer le types d'erreur détecter pour chaque cas et le comparer avec l'erreur attendue. Pour récupérer le type d'erreur révélé, nous avons choisi d'ajouter un champ **error** où en met "NONE" au début et si une erreur est détectée on met dans ce champ le message d'erreur qui doit être affiché.

### 5.2 Implémentation de la pile d'appel

Pendant l'exécution de la solution sur le robot, si la solution contient un appel de fonction, alors toutes les instructions suivant cet appel de fonction ne vont pas être exécutées.

Pour palier à ce problème, nous avons utilisé une pile où on stocke la variable `pc=['ID_FONCTION', 'ID_INSTRUCTION_DANS_LA_FONCTION']` représentant les indices d'une instruction.

On commence par empiler la première instruction de la première fonction. Pour gérer cette pile, on a implémenté la fonction `step` qui met à jour la pile et retourne le nouveau pc à exécuter.

Plusieurs cas de figure peuvent se présenter :

- Si l'instruction est une simple instruction (pas d'appel de fonction) `step` change le dernier pc stocké dans la pile par le pc suivant en incrémentant l'indice de l'instruction dans la fonction et renvoie ce nouveau PC
- Si l'instruction est un CALL `step` empile le PC correspondant à la première instruction de la fonction appelée et renvoie ce PC.
- Si l'instruction correspond à une instruction simple en fin de fonction, `step` dépile le dernier PC, et puis retourne le dernier PC restant dans la pile après l'avoir incrémenté.

On appelle cette fonction jusqu'à ce que la pile soit vide et aucune autre instruction à appliquer, dans ce cas, on retourne `PC=[-1,-1]`.

### 5.3 Affichage terminal

À part l'affichage en mode HTML, nous avions besoin de vérifier si la solution donnée s'exécuté correctement sur le puzzle ou non, pour cela on a pensé à faire un autre affichage dans le terminal sous forme de plusieurs images successives du puzzle, de la position du robot après un déplacement et de l'instruction appliquée. L'affichage des images n'est rien d'autre qu'un affichage du champ `board` du puzzle en utilisant les références des couleurs lors de l'exécution.

## 6 Description des tests du projet

### 6.1 Les tests du projet

Les tests des fichiers `move.test.js` et `puzzles.test.js` ont été faites pour assurer la validité des fonctions programmées. On a donc testé toutes les fonctions et dans chaque fonction, ou on a couvert toutes les lignes, tout cela a été réalisé en utilisant la bibliothèque `jest` qui nous permet de calculer le taux de couverture et contrôler les tests.

```

console.log
[REDACTED]

at forEach (src/display.js:28:17)
at Array.forEach (<anonymous>)

PASS  src/puzzle.test.js
File    | %Stmts | %Branch | %Funcs | %Lines | Uncovered Line #
-----+-----+-----+-----+-----+
All files | 88.62 | 87.05 | 96.77 | 87.95 |
display.js | 100 | 100 | 100 | 100 |
game.js | 82.4 | 82.05 | 94.11 | 82.69 | 45,108-111,122-123,130-131,134-135,159,178,207,211-219
puzzles.js | 100 | 100 | 100 | 100 |
verify.js | 91.52 | 90 | 100 | 90.9 | 8-9,12-13,21

Test Suites: 2 passed, 2 total
Tests:      23 passed, 23 total
Snapshots:   0 total
Time:        0.59 s, estimated 1 s
Ran all test suites.

```

FIGURE 5 – Le taux de couverture des tests

## 6.2 Les tests d'exécutions

On a testé tous les cas possibles qu'un utilisateur peut affronter lors de son expérience sur notre interface **HTML** les tests possibles sont : le robot sort de sa trajectoire, une boucle infinie, le robot ramasse toutes les étoiles et le cas de l'utilisateur ne donne pas de solution.

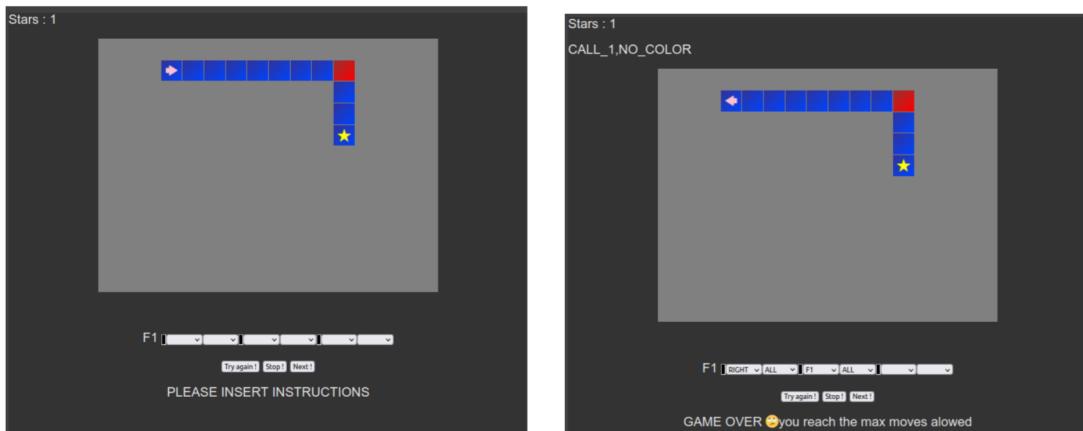


FIGURE 6 – Test d'exécution sans solution et test de boucle infinie

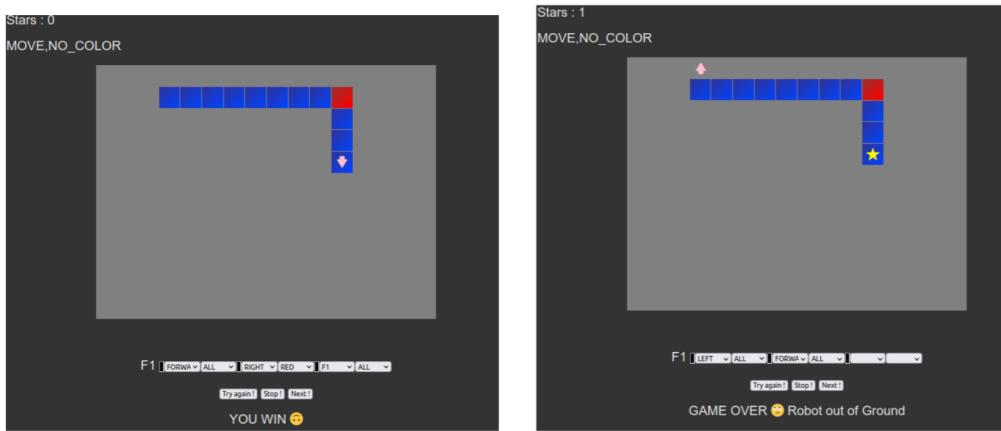


FIGURE 7 – Test d’exécution d’une solution correcte et d’un robot ”Out of ground”

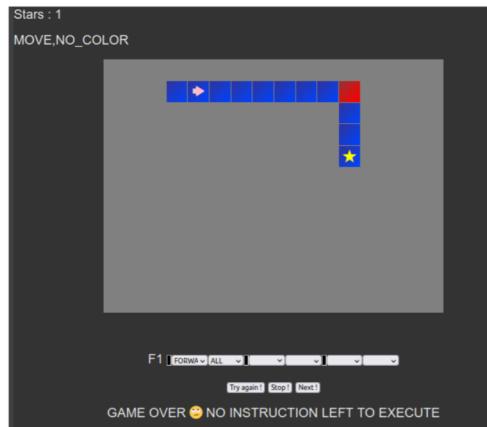


FIGURE 8 – Test d’exécution d’une solution non complète

## 7 Discussion

### 7.1 Limites et améliorations possibles

Plusieurs améliorations sont envisageables : Par exemple, une fonction qui prend en paramètre un puzzle et retourne un programme résolvant ce puzzle.

De plus, une extension du puzzle est possible en ajoutant d’autres éléments que les étoiles, ce qui permet de complexifier le puzzle.

#### **pureté et effets de bords**

Pour améliorer notre projet, on peut réécrire les fonctions programmées en les rendant des fonctions pures. Ainsi, on minimisera les effets de bords, on aura un code plus simple, plus flexible et facile à tester.

### 7.2 Problèmes rencontrés

Premièrement, nous avions une contrainte sur l’écriture du code mode HTML vu que c’est notre première expérience avec ce langage, et donc au début, on a perdu beaucoup de

temps à faire des recherches sur internet afin de comprendre ce langage et pouvoir faire des améliorations pour faciliter l'utilisation de notre interface web.

Dans le cadre de la programmation fonctionnelle et en ce qui concerne la pureté du code, on a essayé d'écrire un code pur dans le maximum des fonctions, mais cela a été difficile à réaliser tout en gardant un code valide.

On a aussi trouvé des difficultés dans l'importation des fonctions d'un fichier vers un autre en utilisant les modules du **Ecmascript** vu qu'on le faisait avant avec les modules du **commonJS** qu'on a fait pendant les séances de TD.

## 8 Conclusion

Dans le cadre des projets de programmation, nous avons donc réalisé un programme du jeu Robozzle. Ce projet a permis de mettre en œuvre les connaissances que nous avons acquises dans le cours de programmation fonctionnelle et de les améliorer. Nous avons été confrontés à de nombreux problèmes, mais nous avons pu trouver des solutions alternatives pour les résoudre.

Le travail en groupe était enrichissant pour la confrontation d'idées et a été une bonne manière de progresser, à la fois sur le code et sur la manière d'appréhender et de résoudre les problèmes.