BORDEAUX INP Enseirb-Matmeca

Rapport de projet Robozzle

Semestre 6 – Année universitaire 2021-2022

Table des matières

1 Introduction

1.1 Présentation du sujet

Dans le cadre de ce projet, il est demandé de réaliser un programme informatique basé sur le jeu Robozzle. Il s'agit d'un jeu dans lequel l'utilisateur choisit une liste d'instructions pour déplacer un robot au sein d'une grille rectangulaire, afin d'accomplir des tâches précises. Par exemple, sur la page Web de Robozzle, le but principal du robot est de ramasser des étoiles tout en restant sur les cases non vides de la grille comme on peut le voir sur la figure 1 ci-dessous.

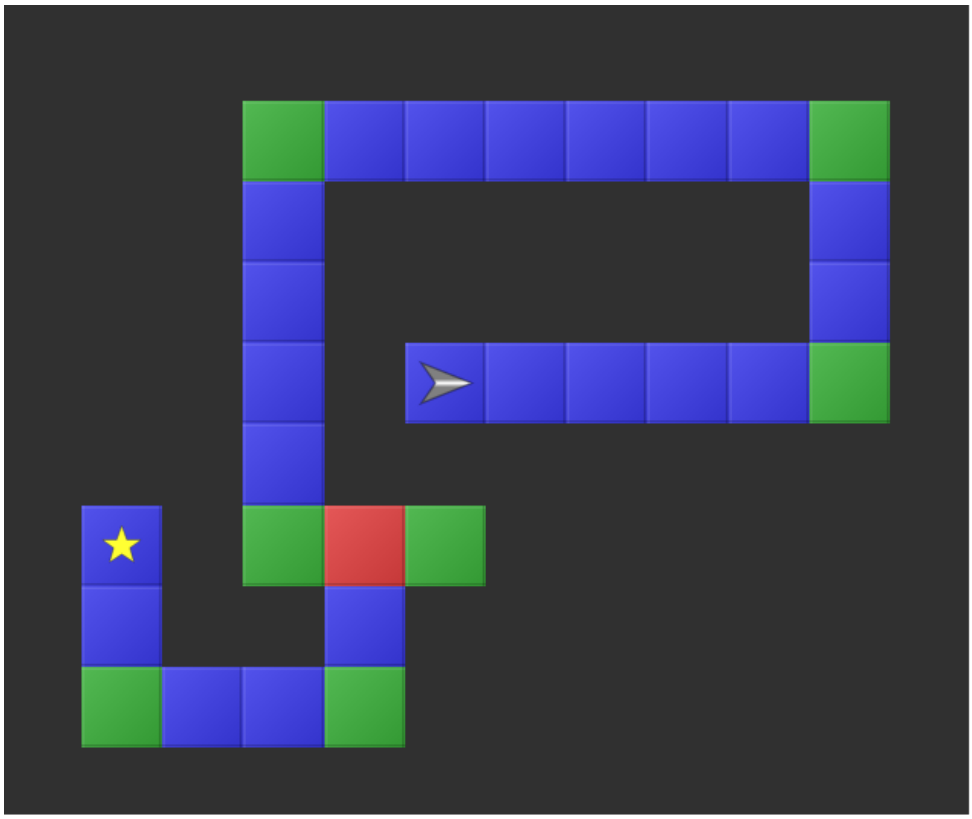

Figure 1: Exemple d'un plateau Robozzle

1.2 Définition du cadre

Lors de ce projet, le langage de programmation imposé est l'Ecmascript. L'exécution du programme peut se faire avec NodeJS ou un navigateur de pages web. La conception et l'implémentation doivent suivre le style de la programmation fonctionnelle : éviter les effets de bord, objets locaux et immutables, pureté et modularité du code.

1.3 Définition des différentes parties

- Le monde correspond au plateau de jeu, c'est une grille rectangulaire.

- Le robot connaît les données qui le concernent et agit à l'aide d'actions.
- Le puzzle ou niveau est une structure contenant un monde et des paramètres initiaux
- L'interpréteur est un programme qui à partir d'un puzzle et d'un jeu d'instruction va créer un robot et exécuter le jeu d'instructions sur ceux-ci. Il doit aussi vérifier que le robot respecte les conditions imposées par le niveau.

2 Analyse et conception algorithmique

2.1 Analyse des problèmes

2.1.1 Problèmes liés au monde

Dans ce projet, nous devons représenter les niveaux de Robozzle et les interactions entre les différents objets au cours de la partie. Il est important de distinguer la notion de niveau et la notion de monde :

- Le niveau est une structure globale, contenant le plateau de jeu ainsi que l'état général du niveau (position et direction du robot, fonctions autorisées, instructions fournies par l'utilisateur...).
- Le monde correspond au plateau de jeu. Il peut être sujet à modifications au cours de la partie. Pour représenter le monde, on doit pouvoir représenter chaque case du plateau de manière indépendante.

Afin de pouvoir jouer à Robozzle, il faut avoir accès à un niveau initial pour démarrer la partie. Pour cela, deux options s'offrent à nous :

- Créer manuellement des niveaux : cela nécessite de stocker les niveaux (pour le programmeur) et de pouvoir les extraire (au moment de l'exécution).
- Générer automatiquement des niveaux : méthode apportant de l'originalité aux niveaux donc plus intéressante pour le joueur, à condition que les niveaux créés soient stimulants. L'implémentation d'un tel générateur "de qualité" n'est cependant pas triviale.

2.1.2 Problèmes liés au robot

Dans ce projet, l'un des points importants est le robot qui doit résoudre le puzzle. Pour réussir le projet, il est important de bien cerner les problèmes amenés par le robot et les choix que nous avions à notre disposition.

Le robot se décompose en deux problèmes majeurs, la représentation du robot et celle de ses fonctions.

En ce qui concerne la représentation du robot, deux choix s'offrent à nous. Nous pouvons choisir d'écrire le robot dans la même structure que le puzzle et le considérer comme une donnée à part entière pour chaque problème, en le considérant comme un élément supplémentaire dans les données des tuiles. Nous pouvons également choisir de le séparer totalement du puzzle en le mettant dans sa propre structure pour le rendre indépendant des puzzles, en considérant sa position par rapport au puzzle dans un second temps.

Le premier choix qui vient à l'esprit quant il s'agit des actions du robot serait de créer des fonctions pour chaque action agissant sur le robot et pouvant être appelées par la suite. Cependant un autre choix serait de laisser les actions à l'interpréteur indépendemment du robot, ainsi, le robot n'est qu'un objet indépendant du reste qui est déplacé par le jeu.

Nous avons opté pour un robot indépendant du puzzle avec ses propres fonctions afin d'augmenter la modularité du code et de bien distinguer le puzzle et le robot comme étant deux parties distinctes ensuite utilisés lors de l'exécution.

L'un des problèmes principaux que l'on doit prendre en compte pour le robot est qu'il ne connaît pas le monde dans lequel il évolue, les données du puzzle et le monde peuvent changer mais le robot reste le même et obéit aux instructions exécutées sur lui qui varient d'un puzzle à un autre. Comme on peut le voir dans la figure 2 ci-dessous, les connaissances du robot sont utilisées pour le placer sur le monde, ce qui signifie qu'il ne peut pas avoir connaissance des données du puzzle.

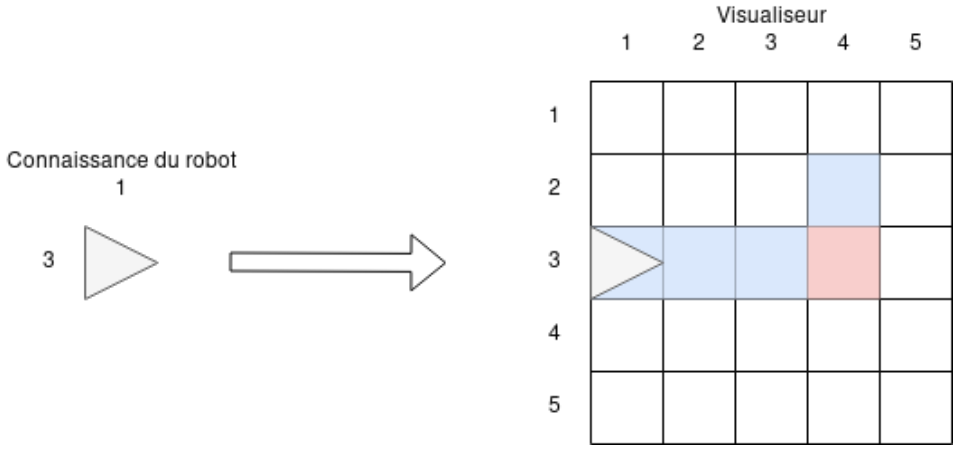

Figure 2: Connaissance du robot

Il faut donc prendre en compte son indépendance par rapport au monde mais aussi par rapport aux instructions qu'on lui applique.

Les actions à appliquer sur le robot doivent également respecter certains points et peuvent entraîner des problèmes qu'il faut également prendre en considération. La plupart des actions élémentaires du robot demandent de modifier un paramètre du robot ce qui est un problème assez simple à considérer.

Cependant d'autres actions demandent d'interagir avec les données du puzzle ce qui n'est pas quelque chose que le robot peut faire d'après les problèmes que nous avons vu précédemment. Une action qui pose problème est l'action permettant de rappeler une suite d'action du robot ce qui est une donnée du puzzle dont le robot n'a pas accès d'après nos choix précédents. Les fonctionnalités que nous avons décidé d'ajouter au robot n'ont pas créé plus de problèmes car il s'agit de modificateurs directement liés au robot et non à son environnement.

2.1.3 Problèmes liés à l'interpréteur

Lorsqu'on exécute d'un jeu d'instructions, on simule un appel de fonctions récursives. Pour simuler une exécution il existe donc plusieurs possibilités.

Figure 3: Exemple d'un jeu d'instruction et la représentation utilisée sur le site Robozzle

L'une d'entre elles est l'utilisation de fonctions récursives, le langage JavaScript nous permet de gérer des fonctions comme des objets avec aisance. Cependant, il serait difficile d'obtenir un comportement pas à pas durant l'exécution et de savoir à tout moment si les contraintes d'exécution sont valides.

Une autre possibilité consiste à gérer une structure de pile image de la pile d'appel, dont chaque élément est une instruction. Ainsi il est facile de connaître l'état de l'exécution en utilisant les propriétés de la pile (la taille de la pile par exemple), et chaque étape correspondrait simplement à une lecture de l'élément en tête suivi d'un dépilage. C'est donc cette possibilité qui est retenue.

2.2 Détails de la conception algorithmique

2.2.1 Conception algorithmique du monde

Nous avons vu que pour représenter le monde, nous avons besoin d'une structure représentant les cases de ce monde. On introduit donc le concept de tuile. Le monde est une "matrice" de tuiles (un tableau de tableaux de tuiles), chacune de ces tuiles contenant diverses informations utiles au cours de la partie. Parmi ces informations, on a :

- la couleur de la tuile : un des concepts de base de Robozzle, qui permet à l'utilisateur de sélectionner des instructions conditionnelles
- la condition de la tuile : définit quelle propriété le robot doit vérifier pour pouvoir traverser cette tuile
- l'effet de la tuile : permet d'inclure une grande variété d'actions pour chaque tuile du monde

Pour que notre code soit plus modulaire, on définit le type abstrait de données "tuile" dans le fichier `tile.js`, avec les primitives suivantes :

- `tile__empty` : → tuile

  crée un dictionnaire correspondant à la définition d'une tuile vide. Par convention, une tuile vide est noire, de condition non-vérifiable (le robot ne peut jamais la traverser), et sans effet.

- `tile__copy` : tuile → tuile

  duplique la tuile passée en argument. Cette fonction sera très utile par la suite pour éviter les effets de bords.

- `tile__get_color` : tuile → couleur (chaîne de caractères)
  `tile__get_cond` : tuile → condition (chaîne de caractères)
  `tile__get_effect` : tuile → effet (chaîne de caractères)

  Les trois fonctions ci-dessus permettent d'accéder aux informations contenues dans la tuile sans en connaître la structure.

- `tile__set_color` : tuile×couleur → tuile
  `tile__set_cond` : tuile×condition → tuile
  `tile__set_effect` : tuile×effet → tuile

  Les trois fonctions ci-dessus permettent de "modifier" la tuile passée en paramètre. En réalité, on fait un appel à `tile__copy` dans ces fonctions pour éviter les effets de bord, la tuile initiale n'est donc pas modifiée.

Ainsi, on peut interagir avec le monde en utilisant le type tuile dans les primitives suivantes du type "monde" :

- `world__empty` : → monde

  crée un monde vide. Par convention, un monde vide est un tableau de tuiles vides.

- `world__copy` : monde → monde

  duplique le monde passé en argument. Cette fonction permet d'éviter les effets de bord.

- `world__get_tile` : monde×entier×entier → tuile

  retourne la tuile aux coordonnées indiquées par les arguments de la fonction.

- `world__set_tile` : monde×tuile×entier×entier → monde

  "remplace" la tuile aux coordonnées indiquées par les arguments de la fonction par une autre tuile. En réalité, on fait un appel à `world__copy` dans ces fonctions pour éviter les effets de bord, le monde initial n'est donc pas modifié. On utilise également `tile__copy` pour la même raison.

En s'aidant de la définition des types de données ci-dessus, on peut désormais définir le type "niveau", qui représente à tout moment l'état de la partie :

- `puzzle__empty` : → niveau

  crée un niveau vide. Par convention, un niveau vide contient un monde vide, ainsi que des coordonnées par défaut pour le robot ($x = 0$, $y = 0$ et orienté vers la droite).

- `puzzle__copy` : niveau → niveau

  crée une copie de l'état actuel du niveau. Cette fonction sera utile par la suite pour éviter les effets de bord.

Pour accéder aux niveaux, on choisit de les créer manuellement et de les stocker avant l'exécution. Il faudra donc également implémenter une méthode capable de lire les niveaux ainsi créés et d'en extraire les informations utiles à l'initialisation d'une partie.

2.2.2 Conception algorithmique du robot

Pour répondre aux problèmes amenés par le robot, nous allons rassembler les éléments concernant le robot dans une même structure. La structure du robot contiendra les caractéristiques élémentaires du robot, ses coordonnées x et y et sa direction. Mais également des caractéristiques plus spécifiques à certaines actions comme la couleur, l'essence ou encore la vitesse du robot.

Étant donné que nous avons choisi d'écrire les actions du robot comme des fonctions, nous devons également prendre en compte le problème de pureté. C'est-à-dire que chaque action retournera un robot différent avec un élément de la structure qui aura été modifié à l'exception de quelques fonctions qui agissent sur des éléments externes au robot. Les actions liées au robot sont les suivantes :

- `move` : robot → robot
  `jump` : robot → robot

  crée un robot avec des coordonnées modifiées selon la direction vers laquelle il pointe par rapport au robot actuel.

- `speedUp` : robot → robot
  `speedDown` : robot → robot

  crée un robot avec une vitesse augmentée ou diminuée par rapport au robot actuel.

- `turnLeft` : robot → robot
  `turnRight` : robot → robot
  `turnTwice` : robot → robot

  crée un robot avec une direction modifiée vers la droite, la gauche ou dans l'autre sens par rapport au robot actuel.

- `refill` : robot×entier → robot

  crée un robot qui regagne une certaine quantité d'essence par rapport au robot actuel.

- `doIfIsOnRed` : robot×monde×couleur → robot
  `doIfIsOnBlue` : robot×monde×couleur → robot
  `doIfIsOnGreen` : robot×monde×couleur → robot

  crée un robot que l'on modifie suivant une action par rapport au robot actuel si le robot se trouve sur une case d'une certaine couleur.

- `paintInColor` : robot×monde×couleur → monde

  crée un monde avec une couleur différente pour la tuile sous le robot par rapport au monde actuel.

- `hasRightColor` : robot×couleur → booléen

  vérifie si le robot a la bonne couleur.

On peut voir qu'il y a des fonctions qui n'ont pas le choix que de communiquer avec le monde pour obtenir des informations qui pourront être obtenues uniquement lors de l'exécution d'une partie, ceci est donc un problème laissé et réglé par l'interpréteur.

2.2.3 Conception algorithmique de l'interpréteur

La solution retenue pour simuler une exécution d'un jeu d'instruction est la gestion d'une pile. Pour cela il est possible de concevoir un type abstrait de données nommé "stack". Les primitives du type pile sont les suivantes:

`stack__empty` :		→	pile
`stack__isEmpty` :	pile	→	booléen
`stack__get_head` :	pile	→	élément
`stack__get_size` :	pile	→	entier positif
`stack__add_head` :	pile×élément	→	pile
`stack__remove_head` :	pile	→	pile

Pour l'exécution d'un jeu d'instructions, il est important de gérer intelligemment la pile. L'application d'une instruction équivaut à un dépilage et l'appel d'une fonction d'instructions correspond à l'empilement dans l'ordre des instructions de la fonction.

L'interpréteur s'occupe aussi de lier un puzzle/niveau à un robot. Pour cela il possède une fonction d'initialisation qui à partir d'un puzzle et d'un jeu d'instructions, crée un robot, une pile, et des paramètres de contrôles permettant la vérification du bon déroulement du jeu.

Plutôt qu'avoir une exécution directe jusqu'à validation des objectifs du robot ou non validation des contraintes, on préférera une exécution étape par étape. C'est pourquoi l'interpréteur possède une fonction calculant les nouveaux états du robot, puzzle et paramètres, après l'interprétation de l'instruction suivante (en tête dans la pile). Pour obtenir une exécution directe, il faut simplement itérer sur la fonction calculant l'état suivant.

Enfin, l'interpréteur a pour rôle de vérifier à chaque état si le robot a rempli ses objectifs ou si il y a non validation des contraintes. Des fonctions de vérification (booléennes) incluses dans l'interpréteur ont pour rôle de stopper l'interpréteur et de renvoyer un message/code de fin à l'utilisateur.

3 Implémentation et Validation

3.1 Réalisation du projet

3.1.1 Réalisation du monde

Afin de représenter le type tuile en JavaScript, une des méthodes les plus triviales est d'utiliser un dictionnaire. On pourrait également les mettre sous forme de chaînes de caractères. Comparons ces deux méthodes :

- exemple de représentation par une chaîne de caractères : `"r10"`
  - `'r'` signifie que la tuile est rouge
  - `'1'` signifie que la condition de la tuile est `true`, autrement dit le robot peut toujours la traverser
  - `'0'` signifie que la tuile est sans effet
- exemple de dictionnaire : `{color:"red", cond:"any", effect:"none"}`
  - `"red"` signifie que la tuile est rouge
  - `"any"` signifie que le robot peut traverser la tuile quel que soit son état.
  - `"none"` signifie que la tuile est sans effet

La tuile décrite par ce dictionnaire est la même que celle représentée plus haut par la chaîne de caractères `"r10"`. Les champs `color`, `cond` et `effect` représente respectivement la couleur, la condition et l'effet de la tuile.

Ces deux méthodes peuvent fonctionner, bien qu'elles soient sensiblement différentes. La représentation par chaînes de caractères occupe un espace mémoire beaucoup plus réduit pour la même quantité d'informations, et semble donc à privilégier du point de vue du programme. Cependant, la lecture de ces chaînes de caractères peut être laborieuse : une personne découvrant le code pour la première aurait besoin d'une explication pour comprendre à quoi les caractères correspondent. La représentation sous forme dictionnaire est facilement lisible, et semble donc à privilégier du point de vue du programmeur. Dans tous les fichiers de code, les tuiles manipulées seront donc des dictionnaires.

Lors de la conception algorithmique du monde, nous avons décidé de stocker les niveaux au préalable, afin que l'utilisateur puisse ensuite y jouer en les sélectionnant dans une base de données. Nous choisissons de représenter ces niveaux par des dictionnaires. Puisque ces niveaux pré-enregistrés sont immuables, le programmeur n'a pas besoin de les modifier fréquemment. Nous choisissons donc d'y représenter les tuiles par un unique caractère, commun à toute les tuiles du même type. Ce choix peut sembler contradictoire avec l'argument de lisibilité évoqué plus tôt, mais il nous permet au contraire de représenter le monde d'une manière très intuitive, comme nous pouvons le voir en figure 4.

```
["                ",
 "                ",
 "                ",
 "     gbbbbbbg   ",
 "     b      b   ",
 "     b      b   ",
 "     b bbbbbg   ",
 "     b          ",
 "   B grg        ",
 "   b  b         ",
 "   gbbg         ",
 "                " ]
```

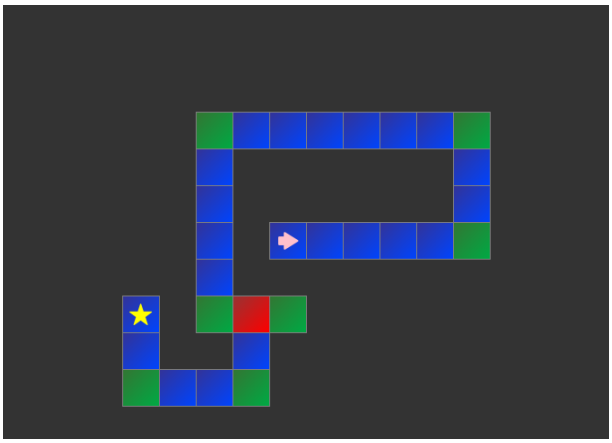

Figure 4: Représentation d'un monde par un tableau de chaînes de caractères

Notons que le robot n'est pas inclus dans cette représentation du monde. En effet, le robot et le monde sont deux structures indépendantes et sont stockées séparément dans le modèle de niveau. Le modèle de niveau est un dictionnaire comprenant quatre catégories de champs :

- informations générales :
  - identifiant du niveau (`id`)
  - nom du niveau (`title`)
- informations sur le robot :
  - position initiale (`initial_x` et `initial_y`)
  - direction initiale (`initial_dir`)

    Ce champ contient un dictionnaire de champs `x` et `y`, en accord avec la convention choisie pour décrire la direction du robot.
- informations sur les commandes :
  - types d'instructions autorisées (`instruction_types`)

    Ce champ contient un dictionnaire qui associe à chaque commande autorisée un unique caractère. Par convention, le champ `" "` contient `null` et correspond à l'absence d'instruction. Les champs "i" contiennent l'entier i et représentent un appel à la fonction `F`i dans l'interpréteur. Les autres champs peuvent varier d'un niveau à l'autre mais sont de la forme `"x" : "instruction"`, avec `"instruction"` le nom d'une fonction du robot définie dans `robot.js`, et `"x"` le caractère symbolisant cette fonction dans le cadre de ce niveau.
  - nombre de fonctions disponibles pour le joueur (`nb_functions`)
  - nombre d'instructions par fonction (`functions`)

    Ce champ contient un dictionnaire de champs `F1`...`F`n avec $n =$ `nb_functions`. Le champ i contient un entier définissant le nombre maximal d'instructions pour `F`i.
- informations sur le monde :
  - liste des tuiles présentes dans le niveau (`tile_types`)

    Ce champ contient un dictionnaire qui attribue un unique caractère à chaque type de tuile présente dans le monde. Il constitue une légende du champ suivant.
  - "carte" du monde sous forme de liste de chaînes de caractères (`world`)

3.1.2 Réalisation du robot

Le robot est donc défini comme une fonction prenant autant de paramètres que de données présentes dans le robot et retourne un dictionnaire contenant chacune de ses

données. Des fonctions sont présentes pour pouvoir manipuler chacune des données du dictionnaire.

Les paramètres du robot sont les suivants :

- `x` : Le paramètre `x` correspond à la position du robot par rapport aux colonnes, en partant de 0 jusqu'à $longueur - 1$.
- `y` : Le paramètre `y` correspond à la position du robot par rapport aux lignes, en partant de 0 jusqu'à $largeur - 1$.
- `dir` : Le paramètre `dir` correspond à la direction vers laquelle le robot est tourné. Elle est donc définie comme un dictionnaire avec un paramètre `x` et un paramètre `y` valant -1, 0 ou 1. -1 signifie que le robot est tourné dans le sens décroissant du paramètre, 1 dans le sens croissant et 0 qu'il n'est pas orienté selon ce paramètre.
- `gas` : Le paramètre `gas` correspond à l'essence que possède le robot. Ce paramètre est considéré comme désactivé s'il vaut -1. S'il a une valeur définie, chacune de ses actions lui coûte une unité et le puzzle est considéré comme perdu s'il atteint 0.
- `speed` : Le paramètre `speed` correspond à la vitesse actuelle du robot. Ce paramètre est réglé à 1 par défaut et est compris entre 1 et 5 inclus.
- `color` : Le paramètre `color` correspond à la couleur du robot.

Maintenant que nous avons expliqué comment est construite la structure du robot, nous pouvons expliquer plus en détail la réalisation de ses fonctions.

Les différentes fonctions élémentaires du robot sont construites avec la même structure, on regarde tout d'abord si le paramètre `gas` est actif, si c'est le cas on regarde s'il en a assez ou pas pour effectuer l'action. Dans le cas où il en a assez ou si le paramètre est désactivé, il effectue l'action élémentaire en appelant la fonction pour créer un nouveau robot où il rappelle chaque paramètre du robot sauf un qu'il modifie. Ce nouveau robot est ensuite renvoyé par la fonction.

La fonction `refill` est particulière car elle n'est pas liée à une action du robot mais lié à l'effet d'une tuile lui permettant de reprendre une certaine quantité d'essence (sans excéder la limite fixée arbitrairement, 20 unités). Cependant, étant donné que l'action concerne une donnée du robot, elle doit se trouver parmi ses actions.

Les fonctions `doIfIsOnColor` sont des fonctions élémentaires améliorées. Elles appliquent la fonction élémentaire du robot seulement si le robot se trouve sur une certaine couleur. Il faut donc communiquer la position du robot pour avoir la couleur de la tuile correspondante dans le monde du puzzle actuel.

La fonction `paintInColor` permet au robot de modifier le monde en modifiant la couleur de la tuile sur laquelle il se trouve en utilisant le même procédé que la fonction

précédente. Cette fonction renvoie ensuite un nouveau monde avec le changement de couleur effectué sur la tuile.

3.1.3 Réalisation de l'interpréteur

L'état d'une exécution définie dans la partie conception est implémenté sous la forme d'un dictionnaire nommé `program_data`. Il contient l'objet robot, l'objet pile, une chaîne de caractère représentant la valeur de validation du programme et un entier compteur d'étape. Cette structure "d'état" est attendue en entrée de la fonction de calcul de l'état suivant.

La fonction de calcul d'état suivant dépend du niveau fixé à l'initialisation. L'idée est donc de la générer de manière à ce qu'elle retienne le niveau en mémoire mais uniquement dans la fonction. Pour cela le JavaScript permet de créer des fonctions génériques et de les retourner. Une fonction `generic_step` est donc réalisée pour retourner une fonction de calcul d'état suivant nommée `step`. Elle est appelée à l'initialisation de l'exécution.

La fonction `step` crée une copie de `program_data` ne fait rien la valeur de validation n'est pas la chaîne vide, sinon elle appelle une fonction qui interprète l'instruction en tête de pile et renvoie une nouvelle pile sans interruption. Après la réception de la nouvelle pile, la fonction `step` utilise les différentes fonctions de validation pour adapter la valeur de vérification et finalament renvoyer la nouvelle structure `program_data`.

Les instructions sont initialement sous forme de double tableau dans les commandes du puzzle, et sont ajoutées au fur et à mesure de l'exécution dans la pile. Elles sont implémentées par un dictionnaire contenant une chaîne de caractère correspondant à la couleur de la case requise pour appliquer l'instruction et un champ `f` soit une fonction du robot, soit un entier pour appeler les instructions de la ième fonction du jeu d'instructions. L'interprétation d'une instruction se fait donc par analyse du type du champ `f` (par utilisation de la syntaxe `typeof`).

3.2 Complexités des implémentations

Toutes les fonctions des types robot, tuile, monde, puzzle et pile étant réalisées en fonctionnel, chacune d'elles renvoient un nouvel objet (une copie modifiée), il n'y a pas d'effets de bord. Il faut donc prendre en compte ce détail pour calculer les complexités.

D'abord, Le robot et la tuile sont des dictionnaires dont les champs sont à dimensions spatiales constantes. Donc créer une copie est réalisé avec une complexité temporelle en constante, de même la complexité spatiale est aussi constante. De plus seules des opérations élémentaires comme la lecture d'un champ, la copie sont réalisées, donc la complexité temporelle et spatiale de toutes ces fonctions est en $O(1)$.

Ensuite, le puzzle possède des champs de dimensions spatiales constantes et un champ contenant un monde. Le monde est considéré comme un double tableau mais pour le projet

la taille de celui-ci est considérée comme constante (16 de largeur et 12 de hauteur). Ainsi comme précédemment les complexités temporelles et spatiales de toutes ces fonctions sont en $O(1)$.

Enfin, l'implémentation utilisée pour la pile ne réalise pas de copie d'elle même mais déplace les objets en mémoire sans les modifier. La complexité spatiale de la pile est évidement en $O(n)$ avec n la taille de la pile. Mais la méthode d'implémentation induit une complexité temporelle de toutes les fonctions en $O(1)$.

Dans l'ensemble toutes les implémentations possèdent donc une complexité en temps constant, ce qui est excellent et dans le plupart des cas pas facile à réaliser avec la programmation fonctionnelle.

Les fonctions en lien avec ces types sont utilisées principalement par l'interpréteur. La fonction de calcul de l'état suivant (`step`) réalise des opérations en temps constant (car elle fait appel aux fonctions des types). Lors d'une exécution l'interpréteur itère un certain nombre de fois sur la fonction `step`, le sujet impose un nombre d'itérations limite car sinon la terminaison de l'algorithme ne serait pas prouvée. Ainsi une exécution complète a une complexité temporelle en $O(n)$ avec n le nombre d'itérations (borné dans le programme). La mémoire utilisée après itération évolue à cause de la pile, le sujet impose aussi une limite sur sa taille.

3.3 Visualisation

Jusqu'à présent toutes les implémentations ont été visualisées avec le terminal par l'utilisation de NodeJS mais il est complexe de visualiser et d'interagir facilement avec le robot, le monde et l'interpréteur. Une des propositions du sujet consiste à développer un code html (voir même css) dans le but d'avoir un affichage adapté et des interactions rapides avec le programme. Un code html, css, ttf et JavaScript sont mis à disposition pour prendre plus facilement en main le développement de tel outil. Cependant l'intégration des JavaScript dans la page html doit se faire avec un serveur (sinon le navigateur trouve pas les fichiers source), des modules de simulation de serveur via Node existent comme `liveserver` ou `parcel`, ils permettent de ne pas avoir à intégrer le code source dans le fichier html (on aurait une duplication du code) mais de le lire directement dans les fichiers source (comme avec Node seul).

Le navigateur et Node étant deux environnements distincts, il y a quelques différences à prendre en compte:

- Certains outils NodeJS n'existent pas sur navigateur: dans notre cas seul le module `fs` permettant de lire des fichiers externes n'existe pas, la lecture de niveau via un fichier `json` ou texte n'est donc pas possible, les niveaux sont donc copiés dans un fichier JavaScript.

- L'affichage dans la console sur Node est remplacé par un affichage sur la console du navigateur par la même commande `console.log` (mais l'utilisateur doit connaître le moyen d'avoir accès à cette console).
- Il est possible de gérer les objets html de la page sur navigateur avec des commandes JavaScript en utilisant les syntaxes `document` et `window`. Pour plus de lisibilté l'utilisation de cette optionalité est limité au fichier `displayer.js`, qui s'occupe intégralement de l'édition de la page du navigateur.
- Il est aussi possible de gérer des fonctions/partie du programme de manière asynchrone avec des promesses, mais cela n'a pas été implémenté.

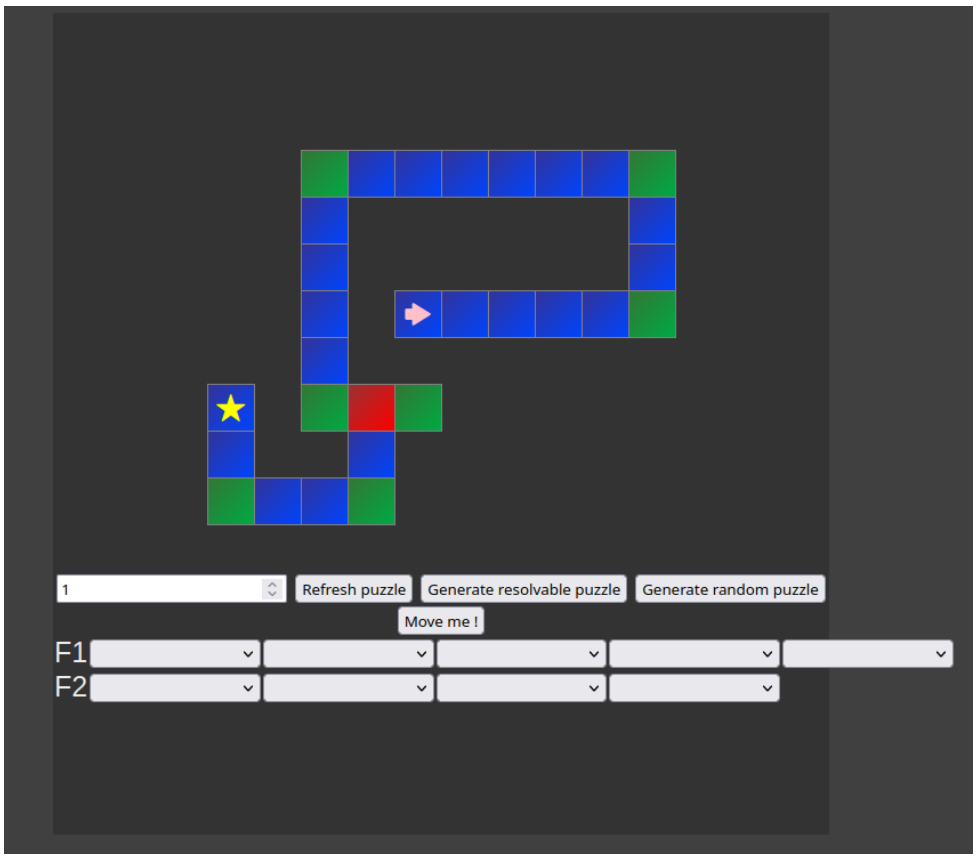

Figure 5: Exemple de visualisation d'un puzzle sur navigateur

Le visualiseur implémenté permet à l'utilisateur de sélectionner quelques puzzles et de pouvoir lancer une exécution pas à pas après la sélection d'un jeu d'instruction, il peut aussi générer des niveaux/puzzles solvable mais pas complexes (mais l'utilisateur ne peut pas lancer d'exécution).

3.4 Validation

Les tests du robot sont très importants dans ce projet car il faut vérifier que les actions du robot produisent bien le résultat attendu. Pour chaque fonction, nous avons donc testé

les différents résultats que pouvait renvoyer la fonction :

- `move/jump`

  Pour tester ces fonctions, on vérifie que les coordonnées du robot ont bien été modifiées selon la direction vers laquelle le robot était tourné. On vérifie également que le robot ne bouge pas s'il n'a plus d'essence.

- `speedUp/speedDown`

  Pour tester ces fonctions, on vérifie que le paramètre `speed` du robot a bien augmenté ou diminué et que cela augmente ou diminue le nombre de cases dont il se déplace.

- `turnLeft/turnRight/turnTwice`

  Pour tester ces fonctions, on vérifie que la nouvelle direction du robot correspond bien au sens vers lequel il a tourné. Si le robot n'a plus d'essence, on vérifie qu'il n'a pas tourné.

- `refill`

  Pour tester cette fonction, on vérifie que le robot à bien gagné `x` unités d'essence et qu'il n'a pas excédé 20 si l'addition de son essence actuelle et `x` unités est supérieure à 20.

- `doIfIsOnColor`

  Pour tester ces fonctions on vérifie que le robot a changé par rapport au précédent s'il est sur la bonne couleur et on vérifie à l'inverse que le robot n'a pas changé par rapport au précédent s'il est sur la mauvaise couleur.

- `paintInColor`

  Pour tester cette fonction, on vérifie que la case sur laquelle se trouve le robot a bien obtenu la couleur souhaitée.

- `hasRightColor`

Nous avons également implémenté une série de tests à différentes couches de la structure d'un niveau : tout d'abord les tuiles, puis les mondes.

- `tile__copy`

  Pour tester cette fonction, on crée une tuile avec `tile__empty` puis une copie de cette tuile avec `tile__copy`. Ensuite, on modifie la copie. Enfin, on vérifie que la modification a bien été prise en compte sur la copie mais que l'original est restée inchangée.

- `tile__set_color/tile__set_cond/tile__set_effect`

  Pour tester ces fonctions, on vérifie simplement que le champ `color/cond/effect` contient la bonne valeur après avoir appliqué la fonction, peu importe la couleur initiale.

- `tile__get_color/tile__get_cond/tile__get_effect`

  Pour tester ces fonctions, on donne une valeur définie au champ `color/cond/effect` grâce à la fonction `tile__set_xxx` (qui a déjà été testée et validée) et on vérifie que le résultat renvoyé par `tile__get_xxx` est correct.

- `world__set_tile/world__get_tile`

  Ces deux fonctions sont testées simultanément : on modifie une tuile du monde en utilisant la fonction `world__set_tile` puis on vérifie que ce changement est détecté par `world__get_tile`.

4 Conclusion

4.1 Résultat du projet

Lors de ce projet il a été possible de réaliser les objectifs suivants (en respectant le cadre de la programmation fonctionnelle):

- Un codage standardisé permettant de créer ainsi que visualiser rapidement un niveau/puzzle
- Un interpréteur pour simuler des exécutions
- Un visualiseur pour une simplicité d'affichage et d'interaction pour l'utilisateur
- Un générateur de puzzle modeste et simpliste
- Des ajouts d'options pour robot et le monde (gas, speed, modification de tuile)

De plus il a été possible de mettre en pratique des concepts/notions de programmation fonctionnelle, et avons pu comprendre les problèmes de réalisation d'un programme utilisant ces notions. Par exemple l'organisation des fichiers par modules Ecmascript. Il a été aussi intéressant d'apprendre directement par la pratique quelques notions de base du html et la gestion d'une page web (en local bien sûr).

Enfin on peut ajouter l'expérience de la difficulté et des problèmes d'organisation lors de la réalisation d'un projet en équipe de 4 personnes.

4.2 Amélioration du projet

Pour aller plus loin, il aurait été possible de réaliser/finaliser les objectifs non atteints et répondre à quelques remarques intéressantes:

- Faire un générateur de puzzle plus avancé et pouvant être joués (car ce n'est pas sur la version actuelle)
- Proposer des méthodes pour résoudre automatiquement un puzzle
- Ajouter plus d'optionnalités pour le robot et le monde
- Proposer plus de puzzles standards
- Afficher la pile d'appel (de l'interpreteur) durant l'exécution
- Avoir la possibilité de revenir en arrière à l'exécution