Un labyrinthe, un robot, un algorithme

2017-2018

Élèves : Assane Diouf (1ère S), Baptiste Franchini (T S), Chloé Gabarren (1ère S), Bastien Girault (1ère S), Nicolas Grzes (2nde), William Parmantier–Cislo (1ère S), Maxime Ponty-Bento (1ère S)

Établissement : Lycées Marguerite de Navarre et Alain Fournier, Bourges

Enseignant(s) : Olivier Créchet et Nathalie Herminier

Chercheur : Benjamin Nguyen (INSA, Centre Val de Loire).

Notre sujet

Durant cette année, nous avons étudié la question de la résolution de labyrinthe dans nos ateliers MATh.en.JEANS.

Notre chercheur nous a donné le sujet suivant : :

On considère un robot pouvant se déplacer dans 4 directions (N, S, E, O) et un labyrinthe quelconque composé de cases vides et de murs (un mur a la même dimension qu'une case vide).

Peut-on proposer un algorithme permettant au robot de sortir de n'importe quel labyrinthe ?

Peut-on quantifier le temps moyen qu'il va falloir au robot pour sortir ?

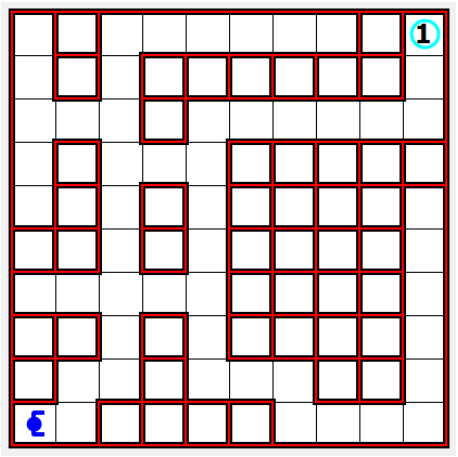
Nous allons démontrer qu'il est possible de sortir de n'importe quel labyrinthe avec certains types de programme, et nous vous exposerons nos méthodes pour quantifier le temps nécessaire afin de trouver la sortie.

1 Introduction

Imaginez que vous êtes dans un labyrinthe, vous ne voyez pas les différents couloirs que vous empruntez, vous ne pouvez pas vous repérer mais vous devez absolument trouver la

sortie. Comment feriez-vous pour en sortir ?
Pour notre part, afin de pouvoir sortir de n'importe quel labyrinthe, nous avons choisi de trouver des solutions avec un robot informatisé commandé par un algorithme.

Tout d'abord, rappelons les notions de labyrinthe, de robot et d'algorithme.
Un **labyrinthe** est un quadrillage composé de cases noires et blanches, les cases noires correspondant aux murs et les cases blanches aux chemins que peut emprunter le robot.
On considère le **robot** comme étant un automate, c'est-à-dire qu'il n'est pas muni de mémoire et qu'il ne suit que les instructions que l'on lui donne, à travers un **algorithme.** De plus, ce robot a une vision limitée : il est seulement capable de voir ses cases environnantes (c'est-à-dire sa case de droite, celle à sa gauche, celles face à lui et derrière lui).
Nos algorithmes étaient dans différents formats : Python, Java, Scratch 2 ou un logiciel dérivé de Python appelé Rurple.



Voici la présentation d'un labyrinthe sous Rurple où les murs sont des carrés rouges. Nous pouvons voir en bas à gauche le robot bleu reposant au point de départ et en haut à droite la bille numérotée représentant la sortie.

2 Notre démarche

Afin de répondre à la problématique, nous avons d'abord dû créer des labyrinthes, tout d'abord manuellement sur Rurple, logiciel utilisant Python, puis automatiquement avec un programme que nous avons créé sous Java permettant de générer un labyrinthe aléatoirement (voir les annexes).

Ensuite, nous avons trouvé des stratégies, que nous avons testées afin d'évaluer leur efficacité. Chaque échec nous permettait de rebondir sur de nouveaux algorithmes.

Puis, nous avons essayé de trouver dans quels cas ces stratégies fonctionnaient, et inversement, dans quels cas elles ne fonctionnaient pas. Nous avons alors effectué des simulations sur des labyrinthes générés aléatoirement afin de déterminer dans chaque cas le nombre de réussites et d'échecs. Ainsi, nous avons pu établir avec un intervalle de confiance la probabilité de réussite d'un algorithme.

Après cela, nous avons essayé de généraliser les cas étudiés.

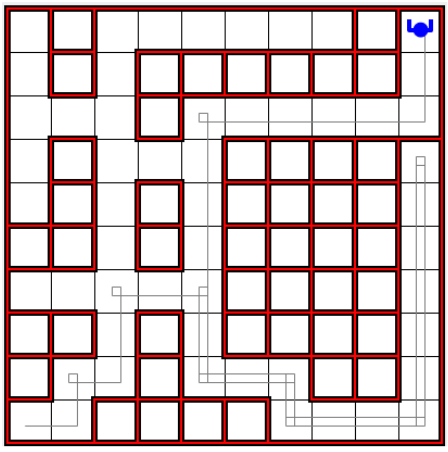
Et enfin, nous avons pu quantifier le temps nécessaire pour les méthodes aléatoires à l'aide de matrices et de graphes.

3 Nos stratégies

Nous avons d'abord utilisé une stratégie dont l'on se sert souvent, nous, humains, confrontés à un labyrinthe, que l'on nomme **Main droite**. Cette méthode semble rapide et efficace.

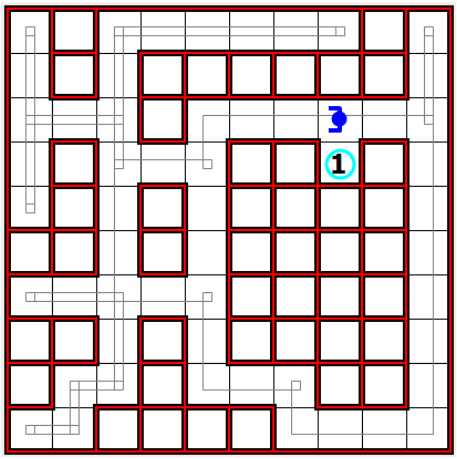
En quoi cela consiste-t-il ?

Le robot longe un mur, dans ce cas précis le mur à sa droite, et avance jusqu'à trouver la sortie. Le robot est dépendant de ce mur et ne le lâche pas.



Nous avons repris le même labyrinthe qu'utilisé précédemment, nous pouvons observer par un trait le chemin emprunté par le robot depuis l'entrée, et nous voyons donc que le robot est arrivé à la sortie.

Malheureusement, cette méthode ne fonctionne pas à tous les coups. En effet, lorsque le robot est confronté à une configuration dite **îlot**, c'est-à-dire lorsque le chemin que doit emprunter le robot nécessite qu'il se détache du mur, le robot se voit dans l'incapacité de trouver la sortie.

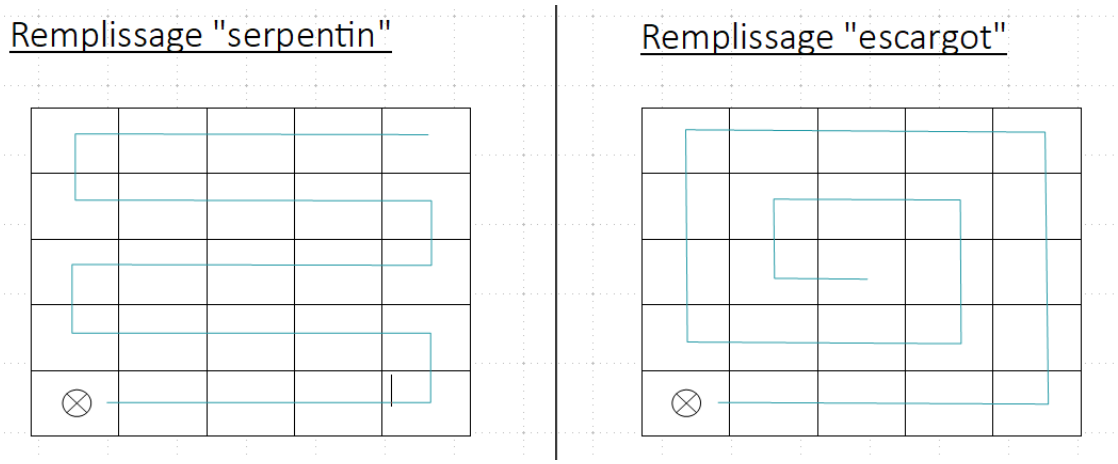


Ici, nous avons déplacé la sortie sur un îlot, nous pouvons voir le trajet du robot. Le robot passe à côté sans la trouver.

Nous avons donc exploré une autre stratégie que l'on nomme **Courbe de remplissage**

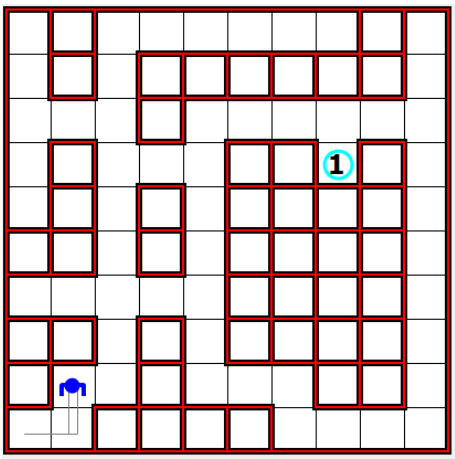
En quoi cela consiste-t-il ?

Cette méthode a pour objectif de parcourir toutes les cases d'un labyrinthe. On utilise donc un schéma répété (par exemple le serpentin ou l'escargot) pour pouvoir remplir cet objectif et ainsi trouver la sortie.



Nous montrons ici cette stratégie sans mur pour visualiser le motif. A chaque fois que le robot rencontre un mur en face de lui, le robot change de direction.

Seulement, dans certaines configurations, cette stratégie n'est pas capable de trouver la sortie. Le robot se voit bloqué. Nous avons constaté par des simulations réalisées sur des labyrinthes 10×10 générés aléatoirement que la méthode de remplissage, pour chaque motif, ne permettait pas de trouver la sortie dans environ 30% des cas.



Le robot essaye ici le motif escargot, il tourne à gauche face au mur une première fois, puis une deuxième fois contre l'autre mur, et répète le processus indéfiniment. Or, dans ce labyrinthe-ci, cette méthode ne fonctionne pas, du fait de la configuration de ce labyrinthe, le robot tourne en rond.

Nous avons ensuite essayé de mélanger les stratégies « Main droite » et « Courbe de remplissage » en alternant au bout d'un certain nombre de pas ces méthodes. Grâce à des simulations nous avons relevé un taux d'échec de 2 %, soit un taux inférieur à ceux des deux méthodes séparées. En revanche, ce taux d'échec est toujours présent, cette combinaison ne permet donc pas de répondre à notre problème

Finalement, on remarque que dans le cas où le robot est un automate et qu'il n'a que les instructions qu'on lui fournit au départ, **on trouvera toujours un labyrinthe qui ne pourra pas être résolu par une méthode déterministe, c'est-à-dire une méthode où l'on connaît à l'avance la trajectoire du robot.**

En connaissance de cela, nous nous sommes donc penchés sur des méthodes non déterministes permettant de sortir de n'importe quel labyrinthe. Nous avons donc esquissé de nouveaux algorithmes utilisant cette fois-ci **l'aléatoire.**

Comment fonctionne notre nouvelle stratégie ? [1]

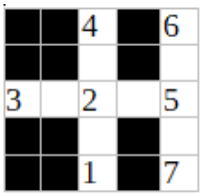
On considère dans ce programme qu'à chaque intersection, le robot choisit aléatoirement une direction en fonction des chemins qui lui sont disponibles. Le robot peut donc tourner à gauche, à droite ou bien simplement aller tout droit.
En revanche, en cas d'impasse, le robot est capable de faire demi-tour afin d'emprunter un autre chemin. Si on ne prend pas en compte le temps, le robot a la certitude de trouver la sortie. Dans le cas contraire, ce programme n'est pas le plus optimisé puisque comme le robot n'a pas de mémoire, il peut souvent se tromper de chemin, tourner en rond ou bien passer un grand nombre de fois par les mêmes chemins.

Pour quantifier le temps de sortie du robot, nous avons représenté le labyrinthe sous forme de graphe. Ce dernier est donc représenté par des sommets, correspondant aux intersections du labyrinthe, et par des arêtes, représentant les chemins. Nous posons donc

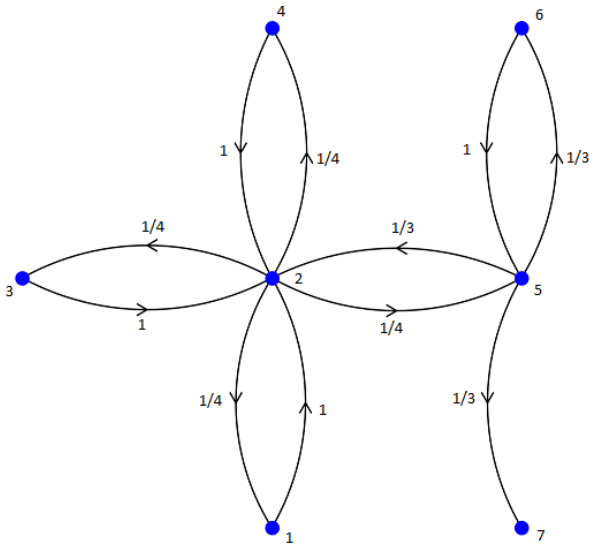
dans ce graphe les probabilités d'emprunter chaque chemin à partir des intersections. Nous utilisons ensuite ces probabilités dans des matrices afin d'obtenir, pour un nombre de pas donné, les probabilités d'atteindre la sortie. Pour pouvoir utiliser une unique matrice dite de « transition », nous avons accordé au robot la possibilité de retourner en arrière.

Voici un exemple de l'utilisation de matrice : on prendra ici un petit labyrinthe.

On considère que le point 1 correspond au point de départ et le point 7 à la sortie
On considère également que lorsque le robot se trouve en 7, il sort du labyrinthe.



Lorsque l'on transpose ce labyrinthe en graphe, on obtient :



On peut voir dans ce graphe les différentes probabilités correspondant à la position du robot au bout d'un pas (un pas est symbolisé par une flèche). Par exemple, la probabilité d'aller de 1 en 2 est de 1 car il n'y a qu'un seul chemin. En revanche, pour aller de 2 en 5, la probabilité est de 1/4 *car on peut également aller en 3, en 4 ou bien retourner en 1.*

Avec ce graphe, nous pouvons obtenir des *états probabilistes* que l'on entre dans des matrices. On note S_i le sommet pour tout entier i tel que $1 \leq i \leq 7$. Soit $P_{i,n}$ la probabilité de se trouver au sommet i au bout de n déplacements pour tout $i \in [\![1\ ;\ 7]\!]$.

La matrice de transition du graphe est :

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Le nombre à la i^{ème} ligne et j^{ème} colonne dans la matrice est appelée coefficient de matrice et est noté $m_{i,j}$. Il correspond à la probabilité d'aller du sommet i au sommet j.

Au coefficient $m_{2,3}$ on obtient une probabilité de 1/4 car elle symbolise la probabilité d'aller de 2 en 3. Au coefficient $m_{1,1}$, on obtient une probabilité de 0, de même pour le coefficient $m_{2,2}$ ou le coefficient $m_{3,3}$ car c'est une matrice de transition symbolisant un pas. Le robot est donc obligé de faire un pas et ne peut donc pas rester sur la même case.
Également, au coefficient $m_{7,5}$, nous avons mis la probabilité 0 car la position 7 représente la sortie, nous voulons donc que le robot sorte du labyrinthe lorsqu'il l'a trouvée.

On définit également $X_n = (P_{1,n}\ P_{2,n}\ \ldots\ P_{7,n})$ l'état probabiliste au bout de n déplacements. Au départ le robot est en S_1 donc :
$X_0 = (1,0,0,0,0,0,0)$

Pour tout entier naturel n, on a alors :
$X_{n+1} = X_n \times M$

On en déduit que pour tout entier naturel n, on a alors :
$X_n = X_0 \times M^n$

Nous pouvons donc obtenir le tableau suivant :

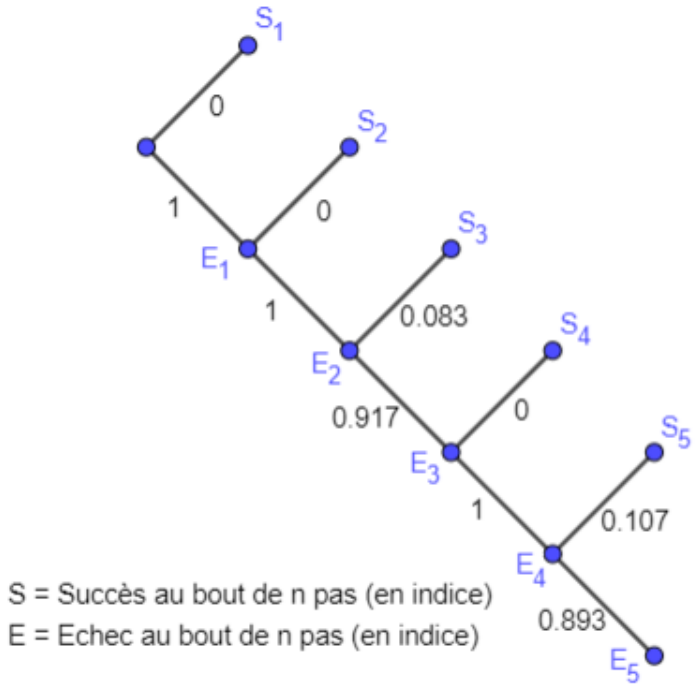
Nombre de pas	**Probabilités d'atteindre la sortie en n pas exactement**	**Probabilités d'atteindre la sortie en n pas maximum**
1	0	0
2	0	0
3	0,083	0,083
4	0*	0,083
5	0,097	0,181
...	...	...
10	0	0,357
15	0,058	0,556
20	0	0,653
29	0,025	0,813
...	...	...
100	0	0,998

Les valeurs sont arrondies à 10^{-3}.

**On observe que la probabilité est nulle d'atteindre la sortie en 4 pas exactement car, en partant du point 1, si l'on fait 4 pas, on ne peut se trouver qu'en 3, 4, 1 ou 5. C'est pourquoi on s'intéresse à la troisième colonne où l'on cumule les probabilités.*

Lorsque l'on observe les probabilités d'atteindre la sortie en n pas maximum, on se rend compte que la probabilité tend vers 1, même si elle ne l'atteindra jamais. En effet, malgré l'augmentation du nombre de pas, il restera toujours une infime possibilité que le robot ne trouve pas la sortie. Si l'on réalise un arbre pondéré avec, à chaque nœud, deux issues possibles au bout de n pas, l'une où le robot trouve la sortie (succès) et l'autre où il ne la trouve pas (échec), il y aura toujours un unique chemin symbolisant l'échec. Même si cette probabilité est non-nulle, il devient de plus en plus improbable de ne pas trouver la sortie lorsque le nombre de pas augmente.

Arbre pondéré représentant les issues possibles en 5 pas au maximum



On voit alors que les chemins «succès» deviennent de plus en plus nombreux, et le chemin «échec» possède une probabilité de plus en plus faible en parallèle (on aura dans un premier temps une probabilité d'échec de 1 pour les deux premiers pas, puis une probabilité de 0.917 pour le troisième pas, etc...)

Grâce au tableau, nous pouvons calculer l'espérance symbolisant le temps moyen nécessaire pour trouver la sortie, de formule :
$E(X) = \sum_{i=1}^{+\infty} x_i P(X = x_i)$.

(L'espérance est égale à la somme des probabilités multipliées par les nombres de pas correspondant)

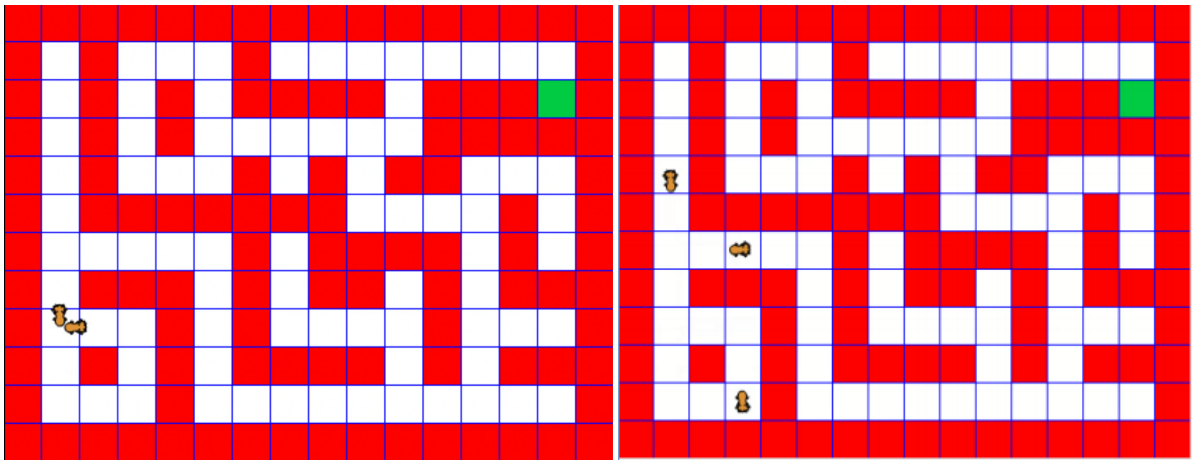
On obtient donc un temps moyen égal à 19 pas. Pour calculer cette espérance, nous avons fait la somme des 100 premiers termes car on considère que le restant, donc quand $n \in]100;+\infty[$, est négligeable. [2]

Nous avons pris l'exemple d'un petit labyrinthe afin d'alléger les calculs, mais il faut imaginer que dans un labyrinthe plus conséquent, le nombre de pas augmente grandement, et donc le temps de sortie aussi.
Pour diminuer ce temps, nous avons pensé à une stratégie que l'on a nommé « **Parallélisme**

En quoi cela consiste-t-il ?

A chaque intersection, le robot se duplique un certain nombre de fois selon le nombre de chemins possibles afin d'emprunter toutes les cases blanches d'un labyrinthe. Chaque robot qui atteint une impasse «meurt» et disparaît. Cela permet de faire plusieurs tâches à la fois et donc de réduire considérablement le temps. Mais cette méthode est très coûteuse en ressources et nécessiterait plusieurs ordinateurs.

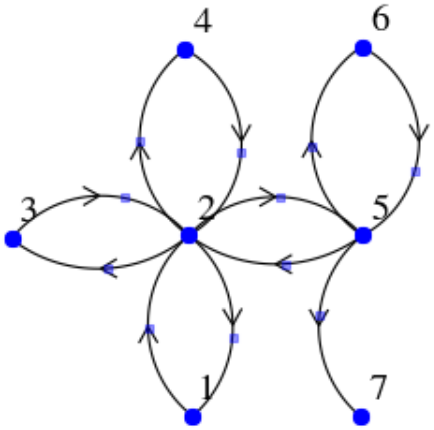
Même si nous n'avons pas pu mettre cette méthode en place, nous pouvons quand même en avoir un aperçu grâce à Scratch 2, on obtient quelque chose sous cette forme :



On voit dans la première image que le robot se duplique car il voit deux chemins. Chaque robot parcourra un chemin jusqu'à une prochaine intersection où il se dupliquera, et ainsi de suite, comme on le voit par exemple dans la seconde image.

Dans ce cas-là, pour calculer le temps de sortie du robot, il est préférable de calculer le nombre de robots nécessaires à cette stratégie, puisque le robot se dédouble à chaque intersection et ne fait donc plus appel à l'aléatoire. Nous avons donc aussi utilisé des graphes dans ce cas-ci pour mathématiser le labyrinthe. Nous considérons la matrice N telle que $N_{i,j}$ est égale au nombre d'arêtes allant du sommet i au sommet j.

Dans le labyrinthe utilisé précédemment, nous obtenons le graphe suivant :



C'est le même graphe que celui utilisé pour la méthode aléatoire sauf qu'ici on n'indique pas les différentes probabilités.

Ainsi que la matrice N suivante :

$$N=\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Nous obtenons la même matrice que la matrice de transition sauf qu'ici, au lieu d'implanter les différentes probabilités d'accéder à chaque chemin, on indique la présence ou non d'un robot empruntant un chemin par le nombre 1 ou 0.
Par exemple, si le coefficient $n_{2,1}$ est égal à 1, cela signifie qu'il y a un robot qui part de 2 et va en 1.

À partir de cette matrice, nous obtenons les résultats suivants :
$X_0 = (1,0,0,0,0,0,0)$ Le robot est à l'entrée sur le sommet 1.
$X_1 = (0,1,0,0,0,0,0)$ Le robot est sur le sommet 2.
$X_2 = (1,0,1,1,1,0,0)$ Le robot s'est divisé en quatre robots.
$X_3 = (0,4,0,0,0,1,1)$ Un robot a atteint la sortie située sur le sommet 7 et 6 robots ont été nécéssaires (4+1+1).

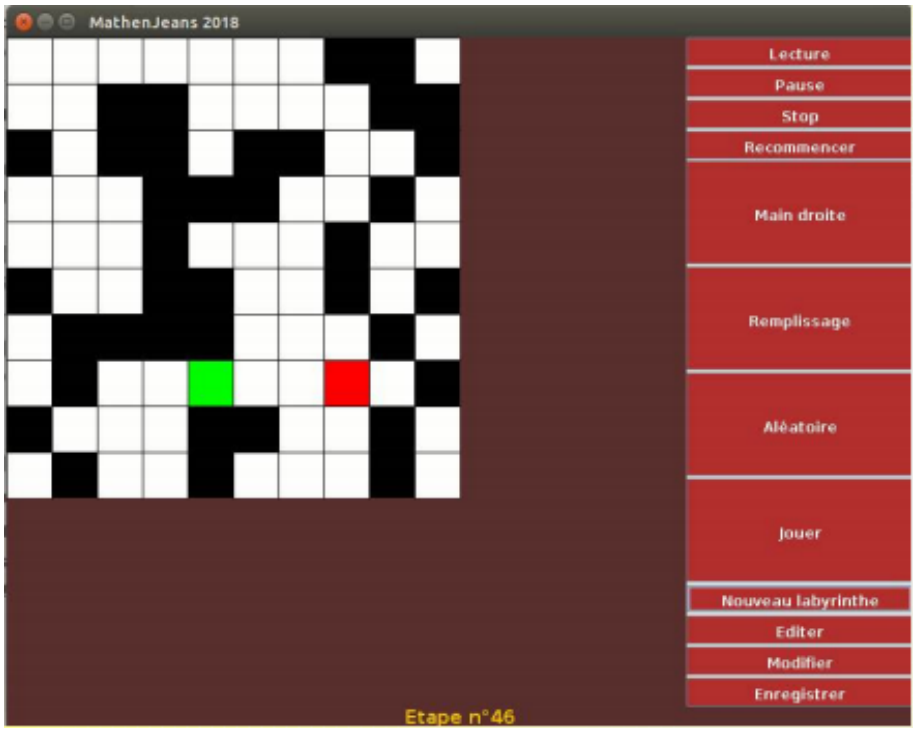
Pour ce labyrinthe, 6 robots seraient nécessaires pour atteindre la sortie.

Nos recherches ont abouti à plusieurs stratégies. Nous avons vu que les stratégies « Main droite » et « Courbe de remplissage », qui étaient des méthodes déterministes, ne permettaient pas au robot de trouver la sortie à tous les coups. Grâce à la stratégie comportant de

l'aléatoire, et aussi, grâce au « Parallélisme », le robot est capable de sortir de n'importe quel labyrinthe. Même si, en ce qui concerne l'aléatoire, il reste toujours une infime possibilité qu'il ne trouve pas la sortie, cette issue est fortement improbable et est donc négligeable. [2] Nous avons pu quantifier le temps que passait le robot dans le labyrinthe grâce à des calculs matriciels où l'on obtient un nombre de pas moyen (ou un nombre exact de robots nécessaires) pour un labyritnhe donné. De meilleurs résultats pourraient être trouvés à l'avenir, mais actuellement, c'est ainsi que nous avons abordé le problème.

Annexes

lAfin de générer aléatoirement nos labyrinthes, et aussi, de tester nos stratégies, nous avons créé un programme qui se présente sous cette forme :

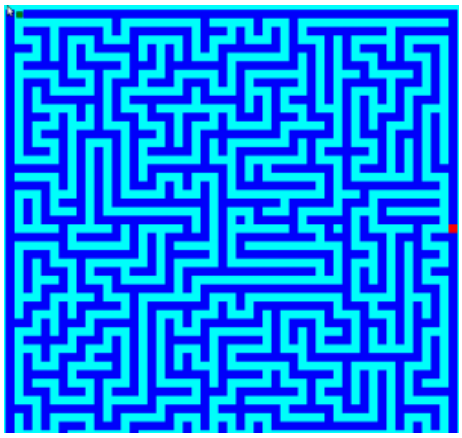


L'interface est assez explicite, on choisit la méthode que l'on souhaite à droite de l'écran. Pour obtenir un autre labyrinthe, on clique sur «nouveau labyrinthe» et pour lancer le programme on appuie sur «lecture». En bas de la page est indiqué le nombre d'étapes, donc le nombre de pas, que le robot a besoin pour trouver la sortie.

Ce programme, utilisant Java, utilise une méthode particulière pour générer aléatoirement des labyrinthes :
On génère d'abord aléatoirement la taille du labyrinthe.

Puis, on place aléatoirement des cases qui sont définies par leurs coordonnées dans ce labyrinthe, remplissant ainsi le labyrinthe avec entre 50 et 75 % de murs.
On pourrait penser que le programme s'arrête ici, mais il faut s'assurer qu'il y a bien un chemin possible entre le point de départ et la sortie. Le programme teste alors ceci en parcourant le labyrinthe de l'entrée vers la sortie, et s'il n'y arrive pas, il supprime suffisamment de murs pour qu'il y ait au moins un chemin permettant d'atteindre la sortie depuis le point de départ.

Lors des stands, nous proposions un jeu où le joueur devait sortir d'un labyrinthe avec une vue en hauteur (donc on changeait les règles de départ). L'objectif était de sortir le plus rapidement possible du labyrinthe. Ce jeu comportait plusieurs niveaux de difficulté qu'il fallait adapter selon l'utilisateur. Il se présentait sous cette forme :



Ce labyrinthe a été réalisé sous Python. Son programme consistait également à créer des murs aléatoirement et si nécessaire en détruire certains pour garantir un chemin menant à la sortie. Le point vert symbolise le personnage et le point rouge la sortie.

Également, nous avons tenté de mettre en place nos méthodes sur le robot «Thymio», sous le programme Aseba, mais par manque de temps, nous sommes encore sur le projet. Le robot «Thymio» est doté de capteurs devant lui et de moteurs réglables. Il se présente sous cette forme :

Notes d'édition

[1] Il n'y a pas de mur dans les deux types de remplissage, donc le robot ne devrait pas changer de direction, ou alors il faut expliquer que le robot fait « comme si »afin d'éviter les cases déjà parcourues.
On « voit »bien les stratégies de remplissage de type serpentin et escargot sur des labyrinthes rectangles vides, mais ce n'est pas évident de décrire ces stratégies sur des labyrinthes « aléatoires ». Il faudrait expliciter en particulier ce qui se passe lorsque le robot est forcé de revenir sur ces pas.
[2] L'explication indiquant que des termes sont négligeables ne suffit pas pour être certain que $x_i P(X = x_i)$ tend vers 0 lorsque i tend vers l'infini (cela revient à dire que « $x_i P(X = x_i)$ tend vers 0 »se déduit de « $P(X = x_i)$ tend vers 0 » ;