MOGPL : Projet

La ballade du robot

Guillaume Hivert && Darius Mercadier

Introduction

Ce rapport présente les réponses aux questions du Projet de MOGPL : La ballade du robot.

Il contient également certaines indications quant à notre travail, et aux choix que nous avons fait lors de l'implémentation des algorithmes.

Cette introduction est également l'occasion de parler du code que nous avons écrit :

Il est écrit dans le langage de programmation Perl. Nous avons inséré un certain nombre de commentaires sur le comportement des fonctions, sur les algorithmes, et sur certaines constructions du langage. Ainsi, même sans être familier avec le langage, vous pourrez vous y retrouver.

Afin de pouvoir lancer le programme, vous devez disposer d'un certain nombre de modules Perl. Les procédures d'installation sont expliquées en commentaire au début du code.

Une fois le programme lancé, un REPL vous guidera et vous proposera différentes opérations.

A\ Formulation du problème sous forme d'un graphe orienté

Rappel : On dispose d'un robot en coordonnées (xi, yi) sur une grille de taille M par N, et on doit trouver le chemin le plus court pour le déplacer en (xf, yf).

On va transformer ce problème en un problème de graphe orienté de la manière suivante :

Nœuds :

Le robot circulant entre les cases, on peut se représenter le labyrinthe comme un ensemble de lignes horizontales et verticales positionnées entre les lignes et colonnes de cases, et le robot se déplace d'une intersection à une autre.
On crée 4 nœuds (un par direction : nord, est, sud, ouest) pour chaque intersection, à condition qu'aucune cause du labyrinthe touchant l'intersection ne soit encombrée par un obstacle.

Arcs:

Les arcs sont les suivant :

- pour les quatre nœuds correspondant à une intersection, on met des arcs suivant :
  Nord <--> Est
  Est <--> Sud
  Sud <--> Ouest
  Ouest <--> Nord
  (ces arcs sont dans les 2 sens)
  En effet, depuis un nœud, si le robot est orienté au nord, l'Est et l'Ouest de ce même nœud sont atteignables. De même pour les autres directions.

- d'autre part, on met de la manière suivant: A partir d'un nœud P de coordonnées (x, y) :
  - Si P est un nœud Nord : un arc de P vers P'(x, y-1), un arc vers P''(x, y-2), et un troisième vers P'''(x, y-3). (A condition de ne pas déborder du labyrinthe bien entendu)
  - Si P est un nœud Est : un arc de P vers P'(x-1, y), un arc vers P''(x-2, y), et un troisième vers P'''(x-3, y). (Sans dépasser du labyrinthe une fois de plus)
  - De même pour Ouest et Sud.

  Ces arcs représentent le fait que le robot ne peut se déplacer qu'en ligne droite, dans le sens où il est orienté.

Tous les mouvements (tourner, ou avancer de 1, 2 ou 3 cases) ont un cout de 1, par conséquent, on associe un cout de 1 à tous les arcs.

La figure 2 (page suivante) montre une partie du graphe obtenu à l'aide de ce procédé. Chaque nœud est étiqueté par ses coordonnées et son orientation. Les coûts des arcs ne sont pas représentés, mais ils sont tous à 1. Ce graphe est le graphe obtenu à partir du labyrinthe suivant :

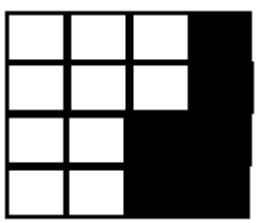


Figure 1. Un labyrinthe de taille 4 x 4

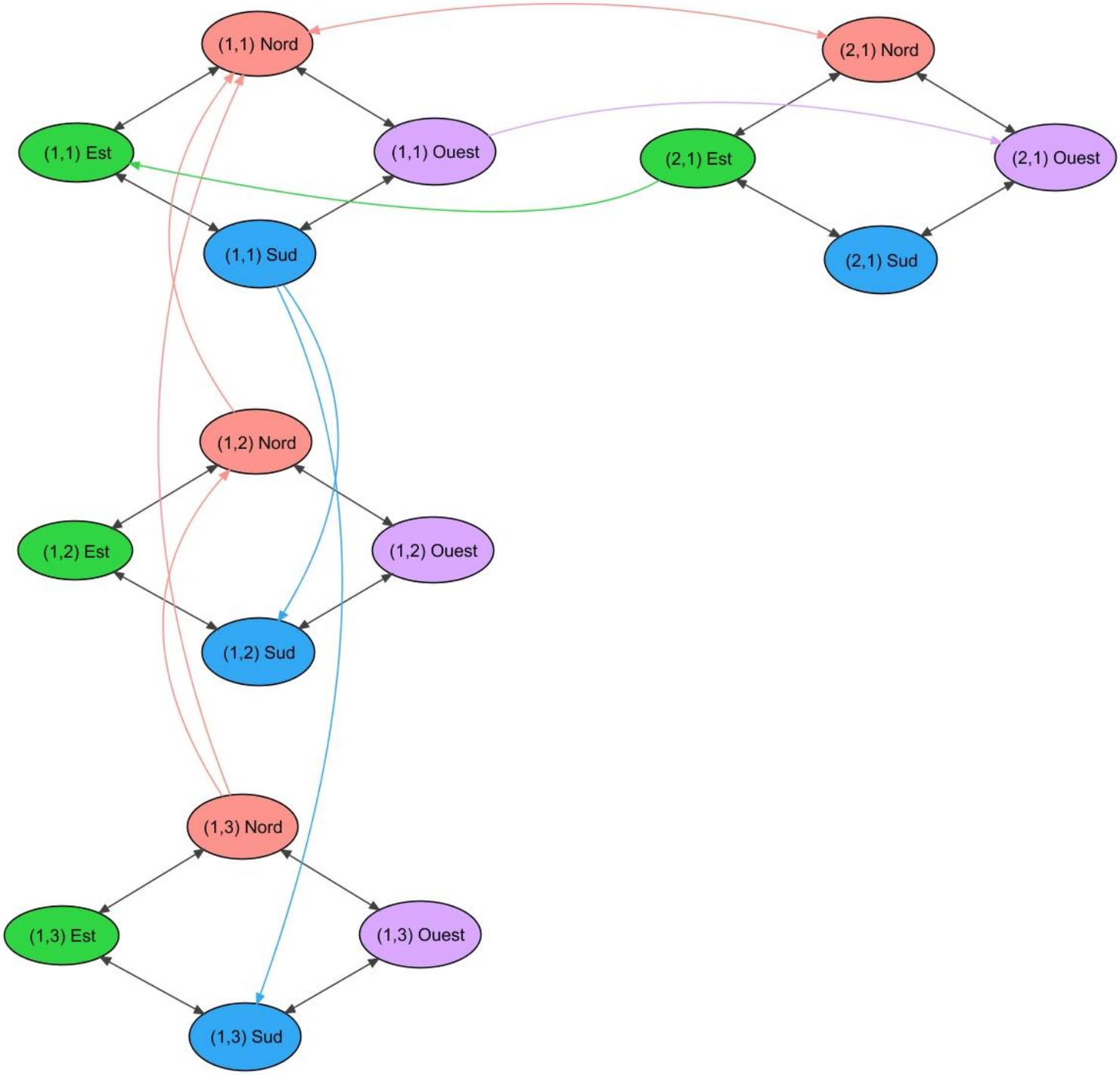


Figure 2. Représentation sous forme de graphe du labyrinthe en Figure 1.

En pratique

Nous avons choisis de représenter ce graphe à l'aide de 4 tableaux à 2 dimensions : un par direction (nord, est, sud, ouest). Les nœuds qui ne devraient pas être présents car inaccessibles le sont donc quand même, mais sont initialisée avec une valeur particulière.

Plus d'explications sur cette structure seront fournies dans la partie suivante.

B\ Résolution du problème

Algorithme

L'étape A nous a permis de transformer le problème initial en un problème de recherche de chemin le plus court dans un graphe orienté. Nous avons donc décidé de le résoudre à l'aide de l'algorithme de Dijkstra de recherche de plus court chemin dans un graphe orienté.

Nous avons donc utilisé 2 structures de données : les 4 tableaux présentés précédemment, et un tas où stocker les nœuds atteignables mais pas encore visités.

Elles sont initialisées comme suit :

- les tableaux sont initialisés à -1 partout. Les cases inatteignables (à cause des obstacles dans le labyrinthe) reçoivent la valeur -2.
- le tas est initialisé avec le nœud de départ (coordonnées et orientation du départ du robot).

A chaque itération, on effectue les opérations suivantes :

- retirer le minimum du tas (opération en O(log(M)) avec M le nombre d'éléments dans le tas), que l'on appelle dans la suite le nœud courant.
- si ce nœud a une valeur différente de -1 dans les tableaux, c'est qu'il a déjà été visité, or dans l'algorithme de Dijkstra, une fois qu'un nœud a été visité, il a sa valeur optimale. Par conséquent, on 'oublie' ce nœud et on passe à l'itération suivante.
- si ce nœud est le nœud sur lequel le robot doit arriver à la fin, on arrête l'algorithme et on récupère le chemin qui a mené jusqu'à ce nœud.
- sinon, à partir des tableaux, on récupère les nœuds accessibles depuis le nœud courant (ce qui est aisément faisable à partir de ses coordonnées et de son orientation). Pour chacun, si sa valeur est -1 dans les tableaux, c'est qu'il n'a pas été visité, alors on l'ajoute au tas. Sinon, soit sa valeur est -2, auquel cas il n'est pas accessible, soit sa valeur est autre chose, auquel cas il a déjà été visité, et on dispose donc déjà de sa valeur optimale.

Complexité de l'algorithme

A chaque itération de la boucle principale, la valeur du chemin le plus court jusqu'à un nœud du graphe est trouvée.
Il y a donc au maximum n itérations. Avec n le nombre de nœuds, soit M x N x 4 (hauteur x largeur x directions).

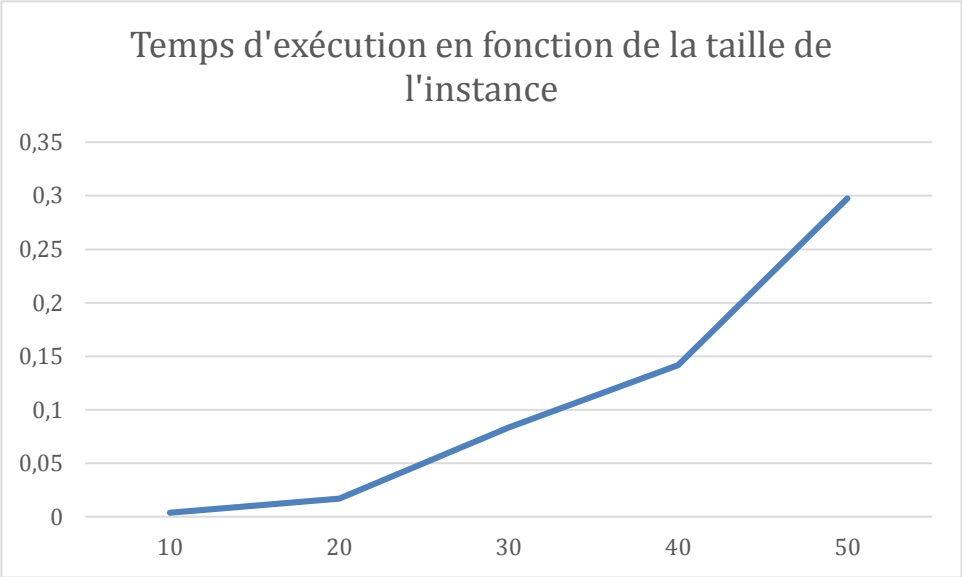
A chaque itération principale, on retire l'élément le minimum du tas. Cette opération est en $O(\log(m))$, avec m le nombre d'éléments dans le tas, majoré donc par n x 5 (car chaque nœud peut accéder au plus à 5 autres nœuds : G, D, a1, a2, a3). Donc c'est une opération en $O(\log(5n)) = O(\log(n)$.

Récupérer les nœuds accessibles depuis un nœud se fait en $O(5) = O(1)$ car seuls 5 nœuds sont accessibles au maximum depuis un nœud, et on y a accès instantanément par les tableaux.

La complexité temporelle de l'algorithme est donc $\mathbf{O(n\log(n))}$, avec n = M * N.

Sa complexité spatiale est la place des quatres tableaux et du tas, soit $4n + O(n)$,
soit $O(5n) = \mathbf{O(n)}$

C\ Temps d'exécution selon la taille



Pour commencer, les éxecutions sur des instances de tailles de cet ordre de grandeur sont très rapides, et nous avons donc décidé, afin d'avoir plus de précision sur nos résultats, d'effectuer les 100 tests pour chaque taille d'instance et non 2.

Cela permet, notamment, de limiter l'impec qu'ont la répartition aléatoire du point de départ et du point d'arrivé. En effet, il peut arriver que ces deux soit très proches, comme il peut arriver qu'ils soient très éloignés, ce qui peut faire que très peu (voir aucun) nœud du graphe n'est exploré, tout comme tous peuvent l'être.

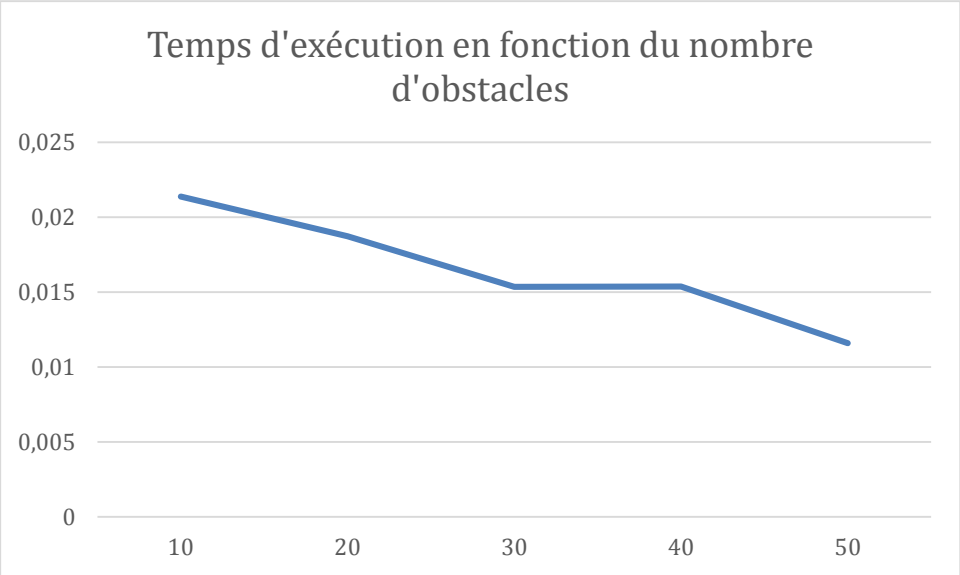
Comme il est possible de le voir sur ce graphique, nous avons pu mesurer le temps d'exécution selon la taille de l'instance. Nous avons effectué de nombreux tests sur des tailles de 10, 20, 30, 40 et 50. Nous avons ensuite fait la moyenne de ces valeurs, afin d'obtenir la courbe ci-dessus. Nous pouvons voir que le temps augmente de façon parfaitement quadratique.

En effet, le labyrinthe étant carré dans nos exemples, et comme expliqué dans la partie précédente, le nombre de nœuds du graphe le représentant est donc de n^2, n étant la taille du labyrinthe. Puisque lorsqu'un nœud est atteint, il l'est de façon optimale (en étant passé par le plus court chemin vers ce nœud), il est donc inutile de passer dessus à nouveau. Chaque nœud est donc au plus visité une seule et unique fois, et l'algorithme effectuera donc au plus n^2 itérations.

Cela corrobore donc parfaitement les résultats pratiques et la théorie, avec une croissance quadratique. Selon la théorie de la seconde question, l'algorithme est en $O(n \log(n))$, avec $n = N * M$. Or, dans un labyrinthe carré, N = M. Donc $O(n \log(n)) = O(N^2 \log(N))$. Soit une croissance quadratique.

Notez que les instances testées, les résultats trouvés, et les temps d'éxecutions sont disponibles dans le dossier Instances.

D\ Temps d'exécution selon le nombre d'obstacles



Après avoir mesuré le temps d'exécution en fonction de la taille de l'instance, nous avons mesuré le temps d'exécution en fonction du nombre d'obstacles.

Nous avons donc, dans une instance de taille 20, effectué la mesure du temps d'exécution de l'algorithme avec 10, 20, 30, 40 et 50 obstacles. Comme dans la partie précédente, nous avons à chaque fois effectué 100 tests.

De la même manière, nous avons effectué la moyenne des résultats des nombreux essais, et nous avons obtenu cette courbe décroissante.

Nous constatons donc que plus il y a d'obstacles, plus le temps de calcul diminue. Ce résultat est tout à fait cohérente avec l'algorithme utilisé – là aussi, Dijkstra :

En augmentant le nombre d'obstacles, on diminue le nombre d'arcs possibles entre les différents nœuds du graphe. En effet, un obstacle empêche le passage du robot. Un obstacle sur une case diminue donc potentiellement de 4 nœuds le graphe. En augmentant le nombre d'obstacles, on diminue le nombre de nœuds du graphe. L'algorithme de résolution effectuera donc moins d'itérations, et s'interrompt plus rapidement.

Notez que les instances testées, les résultats trouvés, et les temps d'éxecutions sont disponibles dans le dossier Instances.

Conclusion

Nous sommes donc partis d'un problème qui était de trouver un chemin le plus court sur un labyrinthe, nous l'avons transformé en un problème de plus court chemin dans un graphe orienté, et l'avons résolu à l'aide de l'algorithme de Dijkstra.

La complexité théorique temporelle de l'algorithme que nous avons proposé était en $O(n\log(n))$ avec n = M x N, où M et N sont la largeur et la longueur du labyrinthe.
Sa complexité spatiale est, quant à elle, en $O(n)$.

Nous avons observé expérimentalement son temps d'exécution sur des instances de différentes tailles, et les résultats constatés sont en harmonie avec la complexité théorique de $O(n\log(n))$.