Travaux pratiques

Résolution de labyrinthe (v.2)

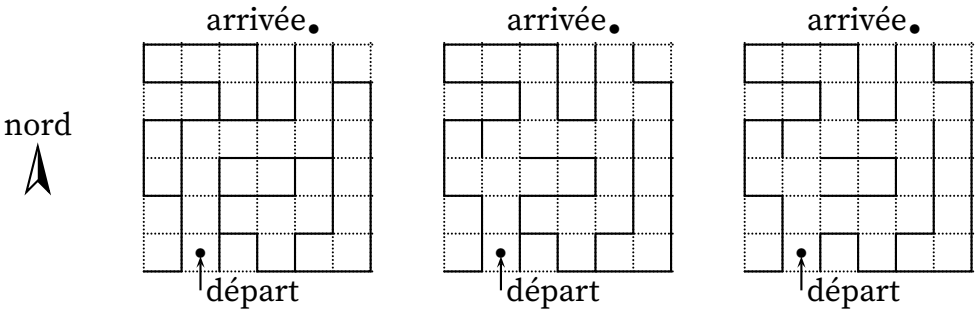
Contexte

On s'intéresse à la navigation d'un robot dans un espace bidimensionnel. Pour simplifier le problème, on discrétise l'espace d'évolution le long d'une grille : on peut repérer chaque *position* par une paire de coordonnées entières `(x, y)`. Aussi, on ne considère que quatre *directions* possibles ; on dote notre pseudo-langage des valeurs correspondantes `nord`, `sud`, `est`, `ouest`.

Quand la voie est libre, on peut se déplacer d'une case à une case adjacente. Un déplacement est donc entièrement défini par la donnée d'une position de départ et d'une direction.

Pour simuler la présence de capteurs, on utilise la routine `est_voie_libre`, qui accepte en entrée une position et une direction, et indique en sortie si le déplacement peut être effectué (la voie est libre) ou non (présence d'un obstacle).

L'objectif est d'écrire un algorithme qui résout un labyrinthe :
les entrées sont une position de départ, la direction initiale menant à cette position de départ, et une position d'arrivée ;
la sortie est une séquence de directions qui permet de passer de la position de départ à celle d'arrivée.



(a) Sans bifurcation (b) Sans boucle (c) Cas général

FIGURE 1 – Exemples de labyrinthes

1 Première séance : groupes, premières solutions

1.1 Groupes de TPiX, rédaction, dépôt

Au sein de chaque groupe de TP*i*, constituer des petits groupes de deux ou trois étudiant·e·s, et les faire valider et enregistrer par l'encadrant.

Cela donne accès, sur la plateforme pédagogique, à la *page collaborative TP*iX *Labyrinthe*. Chaque petit groupe a accès en collaboration à sa propre page, et pas celles des autres.

Il faudra utiliser ces pages pour rédiger les analyses et les algorithmes en pseudo-langage, et déposer le code source en langage C.

La mise en page est importante ; consulter le modèle de page collaborative dans la section correspondante de la plateforme pédagogique.

1.2 Étude algorithmique : labyrinthe sans bifurcation

1. Écrire un algorithme `avancer`, qui accepte en entrée une position et une direction, et qui modifie la position en la position adjacente correspondante.
2. Les trois exemples sur la figure 1 ont une solution en commun. Donner la séquence de directions correspondante.

On suppose dans un premier temps que le labyrinthe ne présente pas de bifurcation, c'est-à-dire qu'à chaque position, il n'existe qu'une seule direction possible, si l'on exclut le fait de revenir en arrière, comme illustrée sur la figure 1(a).

3. Écrire un algorithme `déterminer_prochaine_direction`, qui accepte en entrée une position et la direction empruntée pour arriver en cette position, et calcule la seule autre direction possible.
4. Pour le moment, nous ne connaissons qu'une seule structure de données pour enregistrer la solution : le tableau. Concevoir un algorithme qui permet d'enregistrer une direction supplémentaire dans la séquence de solution, en soulignant les avantages et les inconvénients d'un tableau.
5. Écrire l'algorithme `résoudre_labyrinthe_sans_bifurcation` qui résout le problème.
6. Étudier la complexité spatiale et temporelle de l'algorithme proposé au point 5., en fonction de la longueur de la solution du labyrinthe. Pour la complexité spatiale, on ne comptera pas la structure de donnée finale qui constitue la solution.

1.3 Mise en œuvre en langage C, bibliothèque statique

Sur la plateforme pédagogique sont donnés les fichiers suivants.
Un fichier d'en-tête en langage C `modelisation_robot_labyrinthe.h`, qui reprend la modélisation rudimentaire exposée en introduction.
Un fichier d'en-tête en langage C `interface_simulation_labyrinthe.h`, qui déclare des fonctions permettant de simuler et afficher la navigation dans le labyrinthe.
Une bibliothèque `libsimulation_labyrinthe_linux.a` ou `libsimulation_labyrinthe_windows.a`, qui met en œuvre ces fonctionnalités.

Il s'agit d'une bibliothèque statique. Pour compiler un programme avec une telle bibliothèque avec GCC, il suffit de donner son nom ôté du préfixe `lib` avec l'option `-l` (pour l'anglais *library*), et de préciser à quelle endroit elle se situe avec l'option `-L`. Par exemple sous Linux, si la bibliothèque se trouve dans le répertoire courant

```
nom@ordi .$ gcc autres.c fichiers.c sources.c
   -lsimulation_labyrinthe_linux.a -L. -o nom_du_programme
```

Notons que la bibliothèque a été compilée avec GCC sous Linux et TDM-GCC sous Windows ; les utilisateurs de macOS ou d'autres architectures doivent envisager d'utiliser un système libre, ou le système mis à disposition par l'école.

Écrire un module qui permet de résoudre un labyrinthe sans bifurcation, et un programme qui illustre l'utilisation de ce module.

2 Seconde séance : exploration des bifurcations

2.1 Étude algorithmique : labyrinthe sans boucle

On suppose maintenant que le labyrinthe présente des bifurcations, mais pas de boucle, c'est-à-dire qu'on ne peut pas revenir sur une position déjà visitée sans revenir sur ses pas, comme illustré sur la figure 1(b).

7. Écrire un algorithme `lister_directions_possibles`, qui accepte en entrée une position et la direction empruntée pour arriver en cette position, et calcule la liste des autres directions possibles.

Cela signifie aussi qu'on peut emprunter des chemins sans issue, et il faut envisager de pouvoir revenir en arrière.

8. Écrire un algorithme `direction_opposée`, qui accepte en entrée une direction, et calcule en sortie la direction opposée.

Les chemins sans issue ne doivent pas faire partie de la solution. Il faut donc enregistrer les directions successives quand on explore, mais supprimer ces directions quand on revient en arrière.

9. Déterminer la structure de données la plus adaptée pour enregistrer les directions successives suivies par le robot.

Remarquer qu'on peut utiliser une structure de données similaire pour enregistrer les listes de directions restant à explorer pour chaque position visitée.

10. Écrire un algorithme `explorer`, qui accepte en entrée une position, une liste de directions, et les structures de données pour enregistrer les directions successives et les listes de directions possibles, et qui effectue les opérations suivantes.
    - Extraire une direction de la liste de directions ;
    - enregistrer cette direction au bon endroit ;
    - enregistrer la liste restante au bon endroit ;
    - remplacer la position donnée en entrée par la position obtenue en se déplaçant dans la direction extraite ;
    - remplacer la liste donnée en entrée par la liste des directions possibles depuis la nouvelle position.

11. Déterminer la valeur particulière de la liste des directions possibles, qui signale un chemin sans issue.
12. Écrire un algorithme `revenir`, qui accepte les mêmes entrées que l'algorithme `explorer`, et qui effectue les opérations suivantes.
    — Extraire la dernière direction empruntée ;
    — remplacer la position donnée en entrée par la position obtenue en revenant en arrière ;
    — récupèrer la liste des directions possibles de cette position ;
    — remplacer la liste de directions données en entrée par cette liste extraite.
13. Écrire l'algorithme `résoudre_labyrinthe_sans_boucle` qui résout le problème.
14. Préciser comment la solution de l'algorithme proposé au point 13. peut être utilisée par un autre robot pour suivre le chemin.
15. Étudier la complexité spatiale et temporelle de l'algorithme proposé au point 13., en fonction de la longueur de la solution du labyrinthe, et de la longueur des chemins sans issue.

2.2 Mise en œuvre en langage C ; listes chaînées, piles, files

La mise en œuvre des solutions proposées en § 2.1 nécessite des listes chaînée, des piles ou des files. Il est donc nécessaire de terminer les travaux pratiques sur les listes chaînées ; au moins les fonctionnalités de base `inserer_tete()` et `extraire_tete()`.

Écrire les modules permettant de mettre en œuvre les structures de données nécessaires à la résolution d'un labyrinthe qui présente des bifurcations, un module qui permet de résoudre un labyrinthe sans boucle, et un programme qui illustre l'utilisation de ce module.