

Projet MIPS : labyrinthe

Ce document décrit le projet à réaliser dans le cadre de l'UE architecture des ordinateurs. L'objectif de ce projet est d'implémenter les algorithmes permettant de générer, sauver et afficher un labyrinthe. Le programme sera à réaliser en assembleur MIPS (code source `.s`) et utilisera l'émulateur *Mars* pour l'exécution. Le projet est à réaliser en binôme¹ et devra être rendu sur la plateforme *Moodle* avant le dimanche 6 novembre à 23h55. Il devra être accompagné d'un rapport de 2 pages maximum où vous indiquerez les parties qui fonctionnent et celles qui ne fonctionnent pas dans votre projet. Vous présenterez également les choix d'implémentation que vous avez faits. Ce projet fera l'objet d'une soutenance de 10 minutes pendant la dernière séance de TP pendant la semaine du 7 novembre. Pendant cette soutenance, vous devrez montrer que vous maîtrisez et connaissez votre programme et êtes capables de tester ou modifier certaines parties au besoin.

1 Spécifications

1.1 Labyrinthe carré et parfait

On suppose qu'un labyrinthe est composé d'un ensemble de cellules organisées en N lignes et N colonnes. Chaque cellule a 2, 3 ou 4 cellules voisines. Entre deux cellules voisines, il peut y avoir un mur ou non. Sur la figure 1, les cellules C_0 et C_1 sont voisines, mais ne sont séparées par aucun mur. Par contre les cellules C_0 et C_5 sont des voisines séparées par un mur. Dans un labyrinthe, on appelle *chemin* une suite de cellules où chaque cellule est voisine de la cellule suivante et n'est séparée de celle-ci par aucun mur. Dans ce projet, nous voulons créer des labyrinthes dits *parfaits*, c'est à dire des labyrinthes où il existe un chemin et un seul reliant deux cellules quelconques.

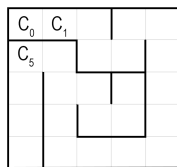


FIGURE 1 – Un labyrinthe parfait de 5 colonnes et 5 lignes



FIGURE 2 – Le même labyrinthe visualisé sous la forme d'un texte. C'est ce qu'on souhaite obtenir.

```
05
29 05 03 09 03
09 03 12 06 10
10 08 03 11 10
10 10 12 06 10
14 12 05 05 38
```

FIGURE 3 – Encore le même labyrinthe sous forme d'entiers. On peut passer de cette forme à celle de la figure 2 au moyen de l'utilitaire `print_maze.sh`.

1.2 Interface utilisateur ? ou ligne de commande ?

Étant donné un entier N , choisi par l'utilisateur, votre programme devra pouvoir générer un labyrinthe parfait et carré de N lignes et N colonnes comme celui de la figure 2 où les murs ont été représentés par des caractères `@`. Le point d'entrée du labyrinthe est représenté par `XX` et la sortie par `OO`.

Idéalement, l'entier N devrait être entré dans la ligne de commande de lancement de *Mars* (celle qui commence par `java -jar ...`) et le labyrinthe devrait être affiché sur la sortie standard que vous pourrez rediriger dans un fichier. Pour en savoir plus, consulter <http://courses.missouristate.edu/kenvollmar/mars/Help/MarsHelpCommand.html>. Comme expliqué à la section 4, ceci vous rapporterait 2 points.

À défaut, l'entier N pourra être saisi par l'utilisateur suite à un appel système (`syscall 5`), comme nous l'avons fait en TP. Et, de même, le résultat pourra être affiché par des appels système (`syscall 1` et `4`) que vous pourrez ensuite copier-coller dans un fichier.

1. Dès que vous avez trouvé quelqu'un pour constituer un binôme, créer un groupe portant vos deux noms (cf première section de *Moodle*)

1.3 Représentations de chaque cellule par un entier

Pour représenter un labyrinthe, il est nécessaire d'avoir la position des $N \times N$ cellules et de savoir si chaque cellule a un mur à gauche, à droite, en haut, en bas et si cette cellule est la cellule d'entrée ou de sortie du labyrinthe. On impose que, chaque cellule soit représentée par un entier et que les informations relatives à cette cellule soient codées dans les 6 bits de poids faible de cet entier (cf le tableau ci-dessous). Vous pourrez utiliser les deux bits de poids fort pour d'autres usages si vous le jugez utile (à mentionner dans le rapport le cas échéant).

B7	B6	B5	B4	B3	B2	B1	B0
libre	libre	Fin	Départ	mur à gauche	mur en bas	mur à droite	mur en haut

À titre d'exemple :

- la cellule en haut à gauche du labyrinthe de la figure 2 a un mur en haut, en bas et à gauche et il s'agit d'une cellule de départ. Ce qui, d'après le tableau ci-dessus, produit l'entier 00011101 qui, en décimal est l'entier 29. Et, dans la figure 3, on retrouve bien 29 pour la cellule en haut à gauche.
- la cellule en haut à droite du labyrinthe de la figure 2 a un mur en haut et à droite et il ne s'agit ni d'une cellule de départ ni d'une cellule de fin, ce qui, d'après le tableau, produit l'entier 00000011 qui, en décimal est l'entier 3. Et, dans la figure 3, on retrouve bien 03 pour la cellule en haut à droite.
- Vous pouvez faire cette vérification pour d'autres cellules.

On impose que, dans votre programme, le labyrinthe soit représenté, comme sur la figure 3, par un entier N représentant la taille du labyrinthe ($N = 5$ dans notre exemple) puis par $N \times N$ entiers représentant chacune des $N \times N$ cellules du labyrinthe. La valeur de chaque entier indique les propriétés de la cellule correspondante selon le tableau ci-dessus.

1.4 Le script `print_maze.sh` pour se faciliter la tâche

Dans les ressources sur la page *Moodle*, vous pourrez trouver le script bash `print_maze.sh`. Après l'avoir téléchargé, vous devez le rendre exécutable avec la commande :

```
chmod u+x print_maze.sh
```

Grâce à ce script, votre programme n'a pas besoin de produire un visuel comme celui de la figure 2. Il lui suffit de produire les entiers comme dans la figure 3. Si vous redirigez ou copier-collez ces entiers dans un fichier texte, comme `laby.txt` (cf *Ressources* sur Moodle), alors la commande :

```
./print_maze.sh laby.txt
```

produira automatiquement un labyrinthe du type de la figure 2. Je vous invite à faire l'essai. Ainsi, tout le travail de ce projet consiste à générer ces $N \times N$ entiers.

1.5 Structure et lisibilité

Votre programme sera évalué non seulement sur les résultats qu'il produit et son efficacité, mais aussi sur sa structuration en procédures et sur sa lisibilité, notamment au moyen de commentaires qui indiquent la signification de vos opérations en termes algorithmiques. Ci-dessous, un style à éviter absolument (figure 4) et un style correct (figure 5).

```
move $a0,$s0      # met le contenu de $s0 dans $a0
jal toto          # appelle toto
move $s2,$v0      # met le contenu de $v0 dans $s2
```

FIGURE 4 – Un code illisible et des commentaires inutiles : À PROSCRIRE!!!

```
move $a0,$s0      # l'adresse du labyrinthe -> $a0
jal rech_vois     # recherche les voisins
move $s2,$v0      # l'indice du voisin choisi -> $s2
```

FIGURE 5 – Des procédures avec des noms porteurs de sens et des commentaires qui rendent le code lisible et qui associent le programme à l'algorithme sous-jacent.

Les spécifications ci-dessus sont strictes et expresses. L'algorithme et la progression décrits ci-après sont, au contraire, des propositions.

2 Algorithme

Dans l'algorithme suivant, on commence avec un labyrinthe où chaque cellule est séparée de toutes ses voisines par 4 murs. Au fur et à mesure de l'avancée de l'algorithme, on casse certains murs pour établir des chemins.

1. Créer un labyrinthe avec des cellules qui ont initialement chacune 4 murs ;
2. Créer une pile de cellules (une pile d'entiers) ;
3. Soit C_0 la cellule en haut à gauche du labyrinthe ;
4. Faire de C_0 la cellule courante ;
5. Marquer C_0 comme visitée ;
6. Mettre C_0 dans la pile ;
7. Tant que la pile n'est pas vide, répéter :
 - Rechercher les cellules voisines de la cellule courante C qui n'aient pas encore été visitées ;
 - S'il n'y en a aucune :
 - Dépiler
 - Soit C' la cellule au sommet de la pile
 - En faire la cellule courante
 - S'il y en a au moins une :
 - choisir une de ces cellules au hasard $\rightarrow C'$;
 - Casser le mur entre C et C' ;
 - Faire de C' la cellule courante ;
 - Marquer C' comme visitée ;
 - Mettre C' au sommet de la pile
8. Faire de C_0 la cellule d'entrée et de la cellule C_N , en bas à droite, la cellule de sortie.

3 Progression

Dans cette section, nous vous proposons une progression. Cette progression permet de poser des étapes intermédiaires de façon à ce que, même si vous n'avez pas fini le projet dans sa globalité, vous puissiez tout de même présenter des résultats tangibles.

3.1 La cellule

Écrire et tester les procédures suivantes qui lisent et modifient les entiers bit à bit. Chacune prend en argument un entier n et un entier i . On numérote les bits de façon à ce que le bit 0 correspond au bit des unités.

- `cell_lecture_bit` qui retourne la valeur du bit i d'un entier n ;
- `cell_mettre_bit_a1` qui retourne un entier identique à n , mais avec le bit i mis à 1 ;
- `cell_mettre_bit_a0` qui retourne un entier identique à n , mais avec le bit i mis à 0.

3.2 La pile

Dans le cours, nous avons vu la pile, marquée par le registre `$sp` (*stack pointer*) et qui sert à stocker les variables locales à une fonction. Mais, comme vous l'avez sans doute vu en SDA1, la notion de pile est plus générale et sert à stocker des structures de données (pour nous, des entiers) de façon à ce que l'élément qu'on retire de la pile (le sommet) est toujours l'élément qui y a été mis le plus récemment. Cette page vous donnera peut-être plus d'explications. Pour la génération du labyrinthe, nous avons besoin d'une pile d'entiers. Pour cela, je vous demande de ne pas utiliser la pile du système (marquée par `$sp`) mais de stocker ces entiers sous la forme d'un tableau dans le tas.

Écrire et tester les procédures suivantes qui permettent de gérer une pile d'entiers. Voici les spécifications indiquant l'état du programme avant et après l'appel à chaque fonction.

- **st_creer** : **Avant** : `$a0` contient le nombre maximal d'entiers que la pile pourra contenir. **Après** : un espace pour au moins `$a0` entiers a été alloué dans le tas (*heap*) et `$v0` contient l'adresse de la pile.
- **st_est_vide** : **Avant** : `$a0` contient l'adresse de la pile. **Après** : `$v0` contient 1 si la pile est effectivement vide et 0 si elle contient au moins un élément.
- **st_est_pleine** : **Avant** : `$a0` contient l'adresse de la pile. **Après** : `$v0` contient 1 si la pile contient le nombre maximal d'éléments et 0 si elle en contient moins.
- **st_sommet** : **Avant** : `$a0` contient l'adresse de la pile. **Précondition** : la pile n'est pas vide. **Après** : `$v0` contient la valeur de l'élément le plus récent de la pile.
- **st_empiler** : **Avant** : `$a0` contient l'adresse de la pile et `$a1` contient un entier à ajouter dans la pile. **Précondition** : la pile n'est pas pleine. **Après** : la pile contient un nouvel élément égal à `$a1` et qui devient, de ce fait, le sommet de la pile.
- **st_depiler** : **Avant** : `$a0` contient l'adresse de la pile. **Précondition** : la pile n'est pas vide. **Après** : le sommet de la pile est supprimée de la pile.

Ces fonctions sont celles qui seraient nécessaires pour le projet labyrinthe. Mais, comme pour toute structure de données, et ne serait-ce qu'à des fins de déboguage, il est toujours utile d'avoir une fonction qui affiche le contenu de la structure de données (en l'occurrence le contenu de la pile).

3.3 Le labyrinthe

Établir un moyen de déterminer si une cellule a été visitée ou non. Écrire et tester des procédures qui puissent :

- créer un labyrinthe de $N \times N$ cellules avec des murs autour de toutes les cellules ;
- à partir de l'adresse du labyrinthe, afficher le contenu de ce labyrinthe sous la forme d'entiers (comme sur la figure 3) ;
- à partir de l'adresse du labyrinthe et de l'indice d'une cellule, obtenir la valeur de cette cellule ou modifier sa valeur (deux procédures) ;
- à partir de l'adresse du labyrinthe, de l'indice d'une cellule c et d'une direction (haut, bas, gauche, droite) d fournir l'indice de la cellule à laquelle on parvient en partant de c et en avançant d'une cellule dans la direction d ;
- trouver toutes les cellules voisines d'une cellule ;
- trouver toutes les cellules voisines d'une cellule et qui n'ont pas encore été visitées ;
- tirer au hasard une cellule parmi ces voisins (cf le programme `Alea.s` dans les ressources).
- mettre en place un algorithme comme celui de la section 2 pour générer un labyrinthe.

Si vous sentez le besoin d'écrire d'autres procédures non décrites ici, sentez-vous libres.

4 Barème indicatif

Ce barème est *indicatif*. Cela signifie qu'il est susceptible de subir des modifications mineures.

- 4 points Traitement des cellules (section 3.1)
- 4 points Traitement des piles (section 3.2)
- 6 points Traitement du labyrinthe (section 3.3)
- 2 points Lisibilité et structuration
- 2 points Rendu du rapport et contenu attendu : état d'avancement, fonctionnalités réussies et non réussies, algorithme (si différent de celui de la section 2) et choix d'implémentation et de structures de données.
- 2 points Utilisation de la ligne de commande pour entrer la taille du labyrinthe et de la redirection pour le sauver dans un fichier texte.

L'obtention de ces points est soumise au fait que, pendant la soutenance, vous puissiez, à la demande de l'examineur :

- localiser l'endroit, dans votre programme, où vous effectuez un traitement donné ;
- modifier ou tester un traitement donné ;
- expliquer la façon dont vous avez mis en œuvre un traitement donné.