Projet Labyrinthe MIPS

Efe ERKEN et Sajjad DAVOUDI

Le projet est réalisé après lecture et compréhension très complet du sujet. Toutes les notions appris en cours ont été intégrées dans le rendu final. La progression conseillé dans le sujet a été suivi avec le plus de précision et de cohérence possible même si elle était très souvent ouverte à interprétation et un peu flou dans ce qui est attendu de l'étudiant.

On s'est rendu compte vers la toute fin du projet que l'utilisation d'une structure de pile dynamiquement alloué n'était pas la solution la plus adapté en vue du problème à résoudre. À cause de la nature très manuelle de l'assembleur MIPS, tout devrait être fait à main et cela prend beaucoup de temps. On a fait la réflexion que l'utilisation d'un tableau alloué toujours dynamiquement pourrait être plus adapté, plus compréhensible/logique et plus facile à gerer. Cela peut aussi réduire beaucoup la complexité et améliorer l'efficacité du programme car on n'a plus a se soucier du défaut d'une pile qui est qu'on ne peut accéder qu'au sommet sans manipuler la pile.

En cela, le point faible du sujet est qu'il essaye de faire deux choses à la fois : suggérer une progression pour ceux qui ont du mal et baser un barème fortement attaché là dessus, pourtant laisser les étudiants libres en leur disant qu'il ne faut pas forcement suivre la progression décrite. Cela ne pose pas de problème tant que la progression décrite reste très claire et précise mais ce n'était pas le cas. Il faut soit bien proposer une progression, soit pas du tout et laisser les esprits libres dans l'implémentation. Le pire c'était d'arriver au bout et voir des problèmes de complexité et l'algorithme qui ne marchait pas car la progression décrite était problématique.

Dernièrement, on s'est senti plus dans un projet de l'UE Algorithmique qu'un projet de l'UE Architecture. Le but c'est de montrer aux professeurs qu'on a compris le bas niveau, comment le processeur marche et qu'on sait l'utiliser pourtant avec la complexité du problème de labyrinthe c'était plutôt un énigme à résoudre que de montrer nos capacités MIPS. Cela ne pose pas problème avec un langage d'un peu plus haut niveau comme le C mais avec MIPS c'était vraiment douloureux. Notre conseil est de baisser la difficulté de l'algorithme dans le sujet ou améliorer la qualité de la progression décrite pour qu'on ne se perd pas dans l'algorithme et qu'on peut se concentrer sur la partie MIPS et le bas niveau qui est la plus importante.

Etat d'avancement

Le projet est complet du début du point 3.1 jusqu'à la fin du point 3.3 avec toutes les fonctions demandées ainsi que les structures de données, la structuration et lisibilité du code avec les spécifications et commentaires à chaque fonction, et l'exécution depuis la ligne de commande.

La progression conseillée a été suivi méticuleusement pourtant il y reste des fonctionnalités non réussis.

Fonctionnalités réussi et non réussi

Toutes les fonctionnalités demandées sont réussi sauf la génération d'un labyrinthe aléatoire. Le générateur de nombre aléatoire du fichier "Alea.s" fourni comme ressource ne génère pas de nombre aléatoire mais au lieu donne toujours le même nombre dans un intervalle donné peu importe si vous recompilez ou reexécuter le programme. "Alea.s" reste toujours dans le rendu final mais puisqu'il génère pas de nombre aléatoire, le labyrinthe est toujours le même.

Si vous utilisez la ligne de commande pour exécuter le programme, sachez que la taille du labyrinthe ne peut pas dépasser les nombres à un chiffre (donc jusqu'à 9) car le parsing d'arguments donné dans la ligne de commande n'est conçu que pour cela. On a implémenté une conversion de caractère ASCII en entier et cela ne marche que pour un chiffre. Le reste on n'a pas eu le temps pour implémenter mais vous pouvez exécuter le programme dans le simulateur Mars graphique et mettre la taille que vous voulez dans le registre $a0.

Dernièrement, sauf pour les labyrinthes de taille 2 et 3 qui ont une unique solution, les autres allant jusqu'à 9 n'ont aucune solution. Cela est pensé dû à l'algorithme utilisé ainsi que le fichier "Alea.s".

Algorithme

On a écrit l'algorithme conseillé pourtant c'était très complexe et inefficace car pour traverser la pile il y avait beaucoup de fonctions récursives et il fallait traverser la pile une énorme nombre de fois pour générer un labyrinthe aléatoire en cassant les murs entre cellules. Le labyrinthe de taille 5 prenait 2,5 minutes pour exécuter dans le simulateur Mars graphique. Par conséquent on l'a changé par un algorithme itérative moins compréhensive pourtant beaucoup plus rapide qui fait la meme chose que l'algorithme proposé mais en traversant la pile une fois du début à la fin en choisissant un voisin à chaque fois. Quand on atteint la fin de la pile on s'arrête. Puisqu'on marque les cellules visité par le bit numéro 6, on a écrit une fonction de nettoyage pour à la fin parcourir le labyrinthe et effacer ce bit dans chaque cellule avant d'afficher le labyrinthe.

Juste après cela on a implémenté l'exécution depuis la ligne de commande et on a finalisé le projet. En testant l'exécution dans la ligne de commande on s'est rendu compte que tout avait la vitesse d'éclair contrairement au Mars graphique. On était surpris par le fait que même notre premier algorithme du sujet s'exécutait maintenant dans quelques secondes. Si on avait plus de temps on aurait pu inclure la premiere version pour avoir suivi jusqu'au bout la progression conseillé. Sauf l'algorithme final, on n'a pas dévié de la progression du sujet. Et cette déviation à la fin est dû au simulateur Mars graphique qui a fait qu'on s'est inquiété du temps d'exécution pour rien.

Choix d'implémentation et structures de données

Pour simuler une pile sans utiliser la pile du système accédé par le registre $sp, on a utilisé le syscall 9 de taille de pile voulu plus 2. C'est une structure de donnée avec 3 champs, la taille maximale de la pile, le nombre d'éléments actuels dans la pile et la pile elle même. Comme cela on a pu écrire les fonctions "st_est_pleine", "st_est_vide" ainsi que toute autre fonction de manipulation de pile. On a bien respecté la simulation de pile en n'accédant que le sommet à la fois au lieu de se tromper et l'utiliser comme un tableau. Dans le programme une pile est utilisé pour le labyrinthe et une autre de taille 4 pour stocker les voisins non visité des cellules.

La pile système $sp est utilisé dans les fonctions pour sauvegarder et restaurer les registres $aX pour les arguments de fonction, $sX pour les registres de sauvegarde et $ra pour le saut à l'endroit d'appel de la fonction appelante en sauvegardant le compteur ordinal.

Pour les cellules, le bit numéro 6 est utilisé pour marquer une cellule visité.

Le bit numéro 7 reste inutilisé.