IPI - Rapport de projet.

Théo Lavigne

Table des matières

1 Préambule

On cherche à lire et exécuter des fichiers représentants des automates $\mathcal{LR}(1)$ ayant un format donné. Le projet est écrit exclusivement en langage C. Les entêtes écrites dans le projets sont incluses dans le `main.c` de la sorte : `#include "nom_entete.h"`.

2 Données du fichier

2.1 Structure des données

J'ai choisi de stocker les données des fichiers `.aut` par des matrices. Ce faisant, on choisit d'implémenter les matrices, d'où l'entête `matrix.h` où sont déclarées les fonctions (leur signatures) et la forme que prennent les matrices : `typedef unsigned char** matrix` soit un tableau de `unsigned char`. Le choix de ce type a été fait au début du projet. En effet, comme vu ci-après, pour récupérer l'état n de l'automate il fallait utiliser des caractères ASCII. C'est pour cela que le choix de `unsigned char` me paraissait plus judicieux. Le fait qu'ils soient non signés étant à but préventif quant à la lecture des futures données, celle-ci étant toutes positives.
On alloue alors de la mémoire lors de la création d'une matrice de $\mathcal{M}_{N,P}(\mathbb{R})$ dans la fonction : `matrix create_matrix` :

```
matrix create_matrix(int N, int P){
    matrix Mat;
    Mat = (unsigned char **) malloc(N * sizeof(unsigned char*));
    int i;
    for (i=0; i<P; i++){
        Mat[i] = (unsigned char*) malloc(P * sizeof(unsigned char));
    }
    return Mat;
}
```

2.2 Récupération des données

La récupération des données est la partie principale du projet. En effet, avoir une structure de données rigoureuse facilite la phase d'exécution de l'automate. C'est d'ailleurs pour cela que la structure matricielle va nous être utile quant à l'obtention des données tout comme l'exécution de l'automate.

On utilise généralement la fonction `fgetc` pour récupérer les données (un caractère à la fois) du fichier donné en argument. Le fichier est ouvert dans la fonction `int main` où l'on passe en argument `(int argc, char *argv[])`. On écrit alors `FILE* fp = fopen(argv[1],"rb");` pour ouvrir le fichier passé en argument. Un message d'erreur est prévu au cas où l'utilisateur ne passerait pas de fichier en argument.

La première chose est d'obtenir le nombre d'états n de l'automate avec la fonction : `int state_number`. Pour cela, on sait que la première ligne du fichier est de la forme : "`a n`" suivi d'un retour à la ligne. Une fois le fichier ouvert en lecture, on parcourt la première ligne jusqu'à lire le caractère de retour à la ligne. On récupère alors les décimales de l'entier n qu'on rend entier avec la fonction `atoi`.

```
int state_number(FILE* fp){
    int n,i=0;
    unsigned char carac;
    char s[10];
    //The cursor is set in the position two, so as to avoid to read : "a_"
    fseek(fp, 2, SEEK_SET);
```

```
    carac = fgetc(fp);
    while(carac != '\n'){//terminating : Reading the entire first line
        s[i] = carac;
        carac = fgetc(fp);
        i++;
    }
    s[i] = '\0';
    n = atoi(s); //ascii to integer
    return n;
}
```

On souhaite ensuite récupérer la valeur des actions `action(s,c)` de l'automate. Pour cela on initialise une matrice de $\mathcal{M}_{n,128}(\mathbb{R})$. Une fois le curseur placé à `action(0,0)` grâce aux fonctions `int len_first_line` et `fseek`, on rempli les n lignes de la matrice des 128 actions correspondantes avec la fonction `fread`.

```
matrix action_matrix(FILE* fp, int n){
    int i=0, N=n, P=128; //128 is the number of octets per lines
    matrix M = create_matrix(N,P);
    for (int i=0; i<N; i++)
        fread(M[i], sizeof(char), P, fp);
    return M;
}
```

On récupère ensuite de la même manière l'ensemble des valeurs correspondantes aux réductions. On initialise une matrice de $\mathcal{M}_{2,n}(\mathbb{R})$. On rempli la première avec la première composante de réduction puis la deuxième ligne avec la deuxième composante de réduction. Cela est fait au sein de la fonction `matrix reducted_matrix`.

On désire ensuite récupérer les octets correspondant à "décale" qui seront eux aussi stockés dans une matrice. Pour cela on place le curseur au début de cette section. Pour connaître la position à prendre on fait un "saut" de plusieurs octets. On passe la première ligne, les $action(s, c)$ au nombre de $n * 128$, la phase de réduction de taille $2 * n$ ainsi que les trois retours à la ligne (1 pour pour les actions, 2 pour la réduction.) Une fois cela fait, on récupère toutes les données jusqu'à ce que l'on lise 3 fois '\255'.

On procède de la même manière pour récupérer les informations relatives aux branchements. On place le curseur à la même position que précédemment et on le décale la taille de la section "décale" en rajoutant les trois octets '\255'.

On possède alors toutes les données de l'automate. Dans le `main` on choisit la notation M_partie pour chaque matrice.

3 Application

3.1 Les piles

Les automates $\mathcal{LR}(1)$ présentés dans ce projet sont des automates à pile. On implémente donc la structure de pile au sein du projet. On crée deux fichiers, un d'en-tête `stack.h` ainsi que `stack.c`. On défini les piles comme suit :

```
struct stack{
    int top;
```

```
    int size;
    int* content;
};
```

Les fonctions nécessaire à l'utilisation des piles sont alors implémentées. On initialise notamment la pile dans la fonction : `stack stack_init`.

```
stack stack_init(){
stack r = {
    .top = -1,
    .size = 1,
    .content = malloc(sizeof(int))
};
return r;
}
```

L'entier `top` est initialisé à -1 ce qui permet de savoir lorsque la pile est vide.

3.2 Boucle de jeu

On peut désormais gérer l'input de l'utilisateur. Une fois toutes les données récupérées on affiche que le programme est prêt à recevoir les entrées qui seront stockées au sein d'un `buf[256]`. On rentre alors dans la fonction `game_loop` . Cette fonction prend en argument toutes les données que l'on a établi précédemment ainsi que le buffer, le nombre d'états n et la pile `state_stack` initialisée ensuite avec l'état 0.

Le corps de la fonction se fait principalement sur un `switch` portant sur `action = M_action[s][c];` avec `[s]` l'état courant et `[c]` le caractère courant. Si l'action vaut 0 alors l'action l'entrée de l'utilisateur est rejetée. Elle est acceptée dans le cas où l'action vaut 1. Dans les deux cas on sort de la fonction grâce au `return`.

Dans le cas où l'action vaut 2, on empile `decale(M_decale,s,c)` et on passe au caractère suivant.

```
case 2 : //décale
    dec = decale(M_decale,s,c);
    push(dec,state_stack);
    i++; //next charac of the buffer
    break;
```

La fonction `decale` associe à un état s et un caractère c, l'état correspondant s' conformément au format du fichier: triplet (s, c, s').

```
char decale(matrix M_decale, char s, char c){
    for(int i=0; i<n_decale; i++)
        if (M_decale[i][0] == s && M_decale[i][1] == c) return M_decale[i][2];
}
```

Dans le cas où l'action vaut 3 et avec la notation $reduit(s) = (n_r, A)$ on récupère A et n_r. Par la suite, on dépile n_r états de la pile. L'état courant est désormais l'état en haut de la pile. On empile `branchement(M_branchement,s,A)`.

```
case 3 : //Reduit(s) = (n,A)
    A = M_reduc[1][s];
    for (int i=0; i<M_reduc[0][s]; i++)
        pop(state_stack); //n states are popped
```

```
    s = top_elem(*state_stack);//new current state
    branch = branchement(M_branchement,s,A);
    push(branch,state_stack); //On empile branchement(s',A)
break;
```

La fonction `branchement` associe à un couple (s, A) un état s' conformément au format du fichier: triplet (s, A, s'). Avec s un état et A un symbole non terminal.

```
char branchement(matrix M_branchement, char s, char A){
    for(int i=0; i<n_decale; i++)
        if (M_branchement[i][0] == s && M_branchement[i][1] == A)
            return M_branchement[i][2];
}
```

4 Conclusion

4.1 Problèmes rencontrés

Lors de la "boucle de jeu", j'ai souvent eu des erreurs de segmentation lors de mes essais. De plus, les procédures d'affichage au sein du programme n'étaient pas toujours exécutées et je n'avais que le `segfault` comme erreur. Pour résoudre ce problème j'ai utilisé la fonction `fflush` appliquée au flux `stdout`. De la sorte, le programme assurait l'affichage de mes `printf` dans le flux `stdout`. Cela m'a permis de trouver la source de mes `segfault` ; dans le cas où l'action vaut 3, je dépilais n_r fois avant d'attribuer à A la deuxième valeur de $reduit(s)$[1]. Ce faisant j'avais un A tel que le couple (s, A) avec s l'état courant ne corresponde à aucun couple de branchement. Alors l'entier `branchement(M_branchement, s, A)` n'existait pas.

4.2 Finalités

Le programme fonctionne pour les tests que effectués, ceux-ci ne sont cependant pas exhaustifs. Il se compile grâce à la commande `make` et au makefile. L'exécutable est nommé comme dans le sujet: `automaton`. On aurait pu rendre l'écriture plus rigoureuse en gérant mieux les types des entiers. En effet on empile parfois des entiers ayant le type `unsigned char` dans des piles pourtant définies comme contenant des entiers de type `int`.

[1] Alors qu'il fallait d'abord attribuer la valeur de A