

Rapport de projet : Machine de Turing en C

Membres du projet : LEGOUEIX Nicolas, DOS SANTOS Katy

Ecrit par : LEGOUEIX Nicolas

Contents

1	Idée générale et procédures de mise en oeuvre	2
1.1	Pourquoi ce projet	2
1.2	Organisation globale du programme	2
2	Listing du programme	4
2.1	Squelette des fonctions et répartition des roles	4
2.1.1	Fonction init	4
2.1.2	Fonction rule_generator	5
2.1.3	Fonction turing_machine	6
2.1.4	Fonction main	8
2.2	Répartition des roles	8
3	Mode d'emploi	10
4	Traces d'utilisation	11
4.1	Sans verbose	11
4.2	Avec verbose	11
5	Code complet	13

Chapter 1

Idée générale et procédures de mise en oeuvre

1.1 Pourquoi ce projet

Nous avons choisi ce projet car nous trouvons le concept de la Machine de Turing très intéressante et pleine de potentiel : une machine virtuellement capable de résoudre n'importe quel problème pour peu qu'on lui fournisse les règles à suivre pour y parvenir ne peut mériter que notre attention, non ?

Pour rappel, une machine de Turing est une machine qui manipule un ruban (infini dans les deux directions, selon le concept théorique). Une tête de lecture va parcourir ce ruban, et pour chaque symbole (ici représenté par le contenu des cases d'un tableau) qu'elle trouvera, la machine mettra en application les règles que l'utilisateur lui aura fourni.

1.2 Organisation globale du programme

Pour mettre en oeuvre une machine de Turing, voici comment nous procédons :

- Une machine de Turing doit avoir un ruban sur lequel elle pourra travailler, on commencera donc avec une fonction *init* qui va ouvrir et analyser le fichier qu'elle reçoit en argument. Chaque caractère trouvé est inséré dans un tableau *init_tape* qui sera ensuite retourné au moyen d'un vecteur en vue des manipulations ultérieures.
- Notre machine est maintenant en possession de son ruban, mais elle ne sait pour le moment pas quoi en faire, il faut donc lui donner des règles à suivre. Pour ce faire, nous utilisons une deuxième fonction

rule_generator. Cette dernière scanne le fichier passé en argument à la recherche d'une suite de 5 *int*. Ces derniers sont placés dans une structure *rule* composée de ces 5 *int* correspondant respectivement à :

- *cur_state*, l'état dans lequel doit se trouver la machine pour accéder à cette règle.
- *symbol*, le symbole que doit lire la tête de lecture pour accéder à cette règle.
- *new_symbol*, le symbole que la tête de lecture doit écrire a cet emplacement du ruban si on exécute cette règle.
- *direction*, la direction dans laquelle la tête de lecture doit se déplacer après avoir mis en oeuvre la règle...
- et *new_state*, le nouvel état dans lequel la machine sera après avoir appliqué la règle.

On scanne autant de fois qu'il y a de lignes dans le fichier (disons n lignes), pour ainsi avoir n structures *rule* que l'on stocke dans un tableau, renvoyé lui aussi au moyen d'un vecteur.

- Enfin, la fonction *turing_machine* récupère le résultat des deux précédentes fonctions et met en oeuvre le traitement du ruban. A chaque position de la tête de lecture, on comparera l'état courant de la machine aux états tarités par les règles. Si on ne trouve aucune règle, la machine a fini son travail et s'arrête, sinon on compare le symbole pointé par la tête de lecture aux symboles traitables pour cet état. Si cette comparaison aboutit, on a trouvé une règle utilisable pour la situation actuelle. Il ne reste alors qu'à imprimer le nouveau caractère correspondant à la règle à la position actuelle, à se déplacer dans la direction indiquée par la règle et à se mettre dans le nouvel état.

On recommence ensuite jusqu'à ce qu'on ne trouve plus de règles pour l'état et le symbole courant.

Par défaut, la machine n'affichera que l'état initial et l'état terminal, mais il est possible d'afficher tout le détail des calculs.

Au besoin, le code du projet est consultable [sur mon dépôt GitHub](https://github.com/Projet-p8-turing/Turing-In-C) : <https://github.com/Projet-p8-turing/Turing-In-C>.

Chapter 2

Listing du programme

2.1 Squelette des fonctions et répartition des roles

2.1.1 Fonction init

```
vect_tape init (char * file_tape)
```

La fonction init prend en argument *char * file_tape*, c'est à dire le premier argument passé au main, c'est à dire le fichier contenant le ruban initial.

```
file = fopen(file_tape, "r");
if (file == NULL){
    printf("Error loading the file: %s. Either it
    ↪ was misspelled or it does not exist.\n",
    ↪ file_tape);
    exit(0);
```

Ce fichier est ensuite ouvert. Si l'ouverture se passe mal, on en informe l'utilisateur et on l'invite à vérifier le nom du fichier. On vérifie ensuite si il est vide.

```
fseek(file, 0L, SEEK_END);
taille = ftell(file);
if (taille == 0){
    printf("This tape file is empty, aborting
    ↪ ... \n");
    exit(0);
```

```

    }
    fseek(file, 0L, SEEK_SET);
    for (i =0; i < taille; i ++){
        fscanf(file, "%c", &c);
        fseek(file, 0L, SEEK_CUR);
        init_tape[i] = atoi(&c);
    }
    output_tape.nb_elem = i+1;
    output_tape.p = malloc(output_tape.nb_elem *
        ↪ sizeof(char));

    for(i = 0; i < output_tape.nb_elem; i++){
        output_tape.p[i+5] = init_tape[i];
    }

```

Ensuite, on parcourt le fichier à la recherche de tout caractère, que l'on convertit en int avant de l'insérer dans un tableau.

2.1.2 Fonction `rule_generator`

```
vect_rule rule_generator (char * file_rule)
```

La fonction `rule_generator` prend en argument `char * file_rule`, c'est à dire le fichier règle passé lors de l'appel au programme en deuxième argument.

```

for(tmp = getc(file); tmp != EOF; tmp = getc(file)){
    if ( tmp == '\n')
        line_number++;
}

```

Le format étant une règle = une ligne, on parcourt le fichier une première fois pour compter le nombre de lignes dans le fichier pour pouvoir allouer la mémoire nécessaire pour stocker les règles. Ni plus, ni moins.

```

fseek(file, 0, SEEK_SET);
for (ligne = 0; ligne < line_number; ligne++ ){
    fscanf(file, "%d_%d_%d_%d", &output_rules.p[
        ↪ ligne].cur_state, &output_rules.p[ligne].
        ↪ symbol, &output_rules.p[ligne].new_symbol,
        ↪ &output_rules.p[ligne].direction, &
        ↪ output_rules.p[ligne].new_state);
}

```

On lit alors le fichier une seconde fois, à la recherche d'une suite de 5 int (soit une règle) autant de fois qu'on a compté de lignes.

Le vecteur *output_rules* est finalement retourné.

2.1.3 Fonction *turing_machine*

```
int turing_machine (vect_tape init_tape, vect_rule rule_list,
    ↪ int cur_state, int verbose){
```

La fonction *turing_machine* prend 4 choses en arguments :

- *vect_tape init_tape* est le vecteur retourné par la fonction *init* et qui permet d'accéder au ruban.
- *vect_rule rule_list* est le vecteur retourné par la fonction *rule_generator* et qui permet d'accéder au tableau contenant les règles.
- *int cur_state*. Etant donné que l'état de départ dépendra seulement de la manière dont l'utilisateur a décidé d'écrire ses règles, celui doit être renseigné par l'utilisateur lui-même.
- *int verbose* permet à la fonction de savoir si l'utilisateur a demandé la mode verbose ou non. Ceci sera déterminé lors de l'affichage des résultats.

```
while(1){
    rule_found = 0;
    for (i = 0; i < rule_list.nb_elem; i++){
        if (cur_state == rule_list.p[i].cur_state
            ↪ && init_tape.p[head_pos] ==
            ↪ rule_list.p[i].symbol){
            rule_found = 1;
            init_tape.p[head_pos] = rule_list.
                ↪ p[i].new_symbol;
            if (rule_list.p[i].direction)
                head_pos++;
            else
                head_pos--;
            cur_state = rule_list.p[i].
                ↪ new_state;
            break;
        }
    }
}
```

La boucle principale va tester autant de fois qu'on connaît de règles. Tout d'abord, on regarde si l'état courant correspond à un état décrit par un règle, et si un des symboles tarités par cette règle correspond au symbole pointé par la tête de lecture.

Si c'est le cas, on a trouvé une règle applicable pour la situation actuelle. On peut donc remplacer le symbole actuel par le nouveau symbole fournis par la règle.

Ensuite, on regarde dans quelle direction la tête de lecture doit se déplacer : à droite pour 1 (*head_pos++*) et a gauche pour 0 (*head_pos--*).

Enfin, l'état courant est modifié pour devenir l'état décrit par la règle. Cette boucle sera répétée jusqu'à ce que *rule_found* soit égal à 0 à la fin de la boucle : cela signifie qu'on a parcouru l'intégralité des règles sans en trouver une applicable, c'est notre cas d'arrêt.

```
if(!rule_found){
    printf("\nNo rule found for the current state: %d,
        ↪ job is done.", cur_state);
    printf("\nExited with:\n");
    for(int n = 5; n > 0; n--){
        if (init_tape.p[n-1] == 1)
            printf("%d", init_tape.p[n-1]);
    }
    for (n = 0; n < init_tape.nb_elem-1; n ++){
        printf("%d", init_tape.p[n+5]);
    };
    printf("\n");
    return 0;
}
```

On informe l'utilisateur qu'aucune règle n'a été trouvée, et on imprime le ruban.

```
if (verbose == 1){
    printf("state is: %d. Head is at position %d\n",
        ↪ cur_state, head_pos);
    printf("current tape is:");
    for(int n = 5; n > 0; n--){
        if (init_tape.p[n-1] == 1)
            printf("%d", init_tape.p[n-1]);
    }
    for (int n = 0; n < init_tape.nb_elem-1; n ++){
```



```

        printf("%d_", init_tape.p[n+5]);
    };
    printf("\n");

```

Enfin, si *verbose* vaut 1, c'est à dire que le programme a été appelé avec *-v*, pour chaque tour de boucle, donc à chaque étape du traitement, on affichera l'état du ruban, l'état courant et la position de la tête de lecture.

2.1.4 Fonction main

```

int main (int argc, char *argv[]){

if(argv[4] && !strcmp(argv[4], "-v"))
    verbose = 1;
else{
    printf("Note_:call_with_v_for_detailed_processing.\n\
        ↪ n");
    verbose = 0;
}
if (argc < 4 || argc > 5 || !strcmp(argv[1], "-help")){
    printf("//mode d'emploi en detail");
    exit(0);
}

```

La fonction main sert essentiellement à appeler les autres fonctions dans l'ordre, mais aussi à initialiser la variable *verbose* et à afficher le mode d'emploi du programme si il n'est pas appelé correctement.

```

init_tape = init(argv[1]);
printf("initial_call_done_with_:");
for (n = 0; n <= init_tape.nb_elem-2; n ++){
    printf("%d_", init_tape.p[n+5]);
};
printf("\n");

```

Après avoir récupéré le ruban initial en appelant *init()*, on informe l'utilisateur du ruban sur lequel la machine va travailler.

2.2 Répartition des rôles

- *init* a été codée par Nicolas LEGOUEIX

- *rule_generator* a été codée par Nicolas LEGOUEIX
- *turing_machine* a été codée par Nicolas LEGOUEIX
- *main* a été codée par Nicolas LEGOUEIX

Chapter 3

Mode d'emploi

Une fois compilé, le programme s'appelle avec 3 ou 4 arguments :

- le fichier contenant le ruban. Notre machine étant une machine binaire, celui-ci doit être une suite ininterrompue de 0 et 1.¹
- le fichier contenant l'ensemble des règles. Chaque membre de la règle doit être séparé par un espace. De plus, chaque ligne ne doit contenir qu'une seule règle. Exemple :

0 0 0 0 1
1 0 0 0 0
(...)

- L'état dans lequel la machine doit commencer le traitement.
- Enfin, il est possible (mais pas obligatoire) d'utiliser l'option -v (pour verbose) qui permet d'afficher le détail, étape par étape, de la progression de la machine.

Note : Toutes ces informations vous sont rappelées si vous appelez le programme de manière incorrecte ou avec l'option *-help*

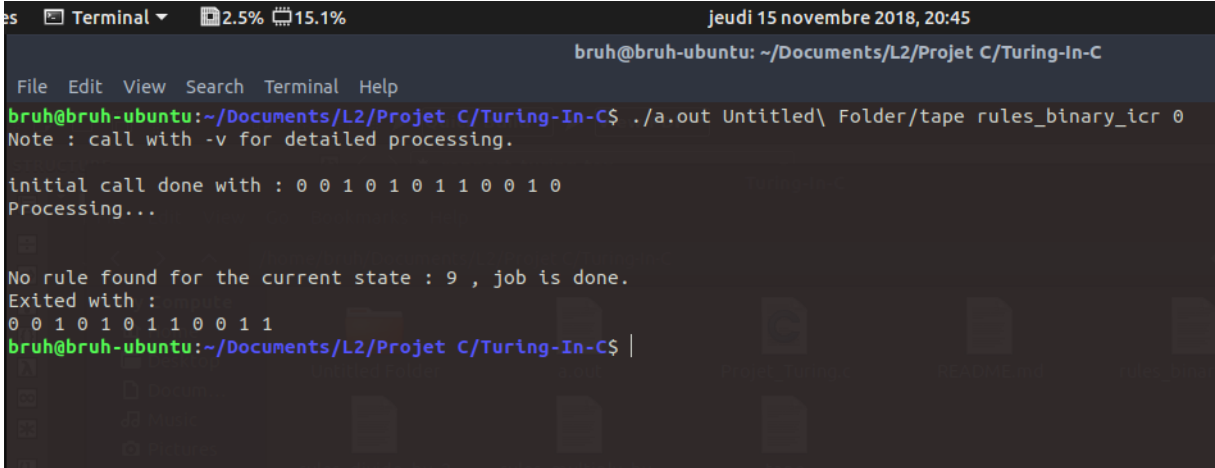
¹Ceci dit, si vous y mettez d'autres chiffres et que vos règles sont adaptées, la machine devrait fonctionner...

Chapter 4

Traces d'utilisation

4.1 Sans verbose

Voici un exemple d'utilisation sans verbose, pour une incrémentation binaire :



```
es  Terminal 2.5% 15.1% jeudi 15 novembre 2018, 20:45
bru@bru-ubuntu: ~/Documents/L2/Projet C/Turing-In-C
File Edit View Search Terminal Help
bru@bru-ubuntu:~/Documents/L2/Projet C/Turing-In-C$ ./a.out Untitled\ Folder\tape rules_binary_icr 0
Note : call with -v for detailed processing.

initial call done with : 0 0 1 0 1 0 1 1 0 0 1 0
Processing...

No rule found for the current state : 9 , job is done.
Exited with :
0 0 1 0 1 0 1 1 0 0 1 1
bru@bru-ubuntu:~/Documents/L2/Projet C/Turing-In-C$ |
```

4.2 Avec verbose

Et un autre, toujours une incrémentation binaire, avec verbose. :

```
s  Terminal 2.5% 15.4% vendredi 16 novembre 2018, 07:42
bruh@bruh-ubuntu: ~/Documents/L2/Projet C/Tur
File Edit View Search Terminal Help
bruh@bruh-ubuntu:~/Documents/L2/Projet C/Turing-In-C$ ./a.out tape rules_binary_icr 0 -v
initial call done with : 1 0 0 1 1 0 1 1
Processing...

state is : 0. Head is at position 1
current tape is : 1 0 0 1 1 0 1 1
state is : 0. Head is at position 2
current tape is : 1 0 0 1 1 0 1 1
state is : 0. Head is at position 3
current tape is : 1 0 0 1 1 0 1 1
state is : 0. Head is at position 4
current tape is : 1 0 0 1 1 0 1 1
state is : 0. Head is at position 5
current tape is : 1 0 0 1 1 0 1 1
state is : 0. Head is at position 6
current tape is : 1 0 0 1 1 0 1 1
state is : 0. Head is at position 7
current tape is : 1 0 0 1 1 0 1 1
state is : 0. Head is at position 8
current tape is : 1 0 0 1 1 0 1 1
state is : 1. Head is at position 7
current tape is : 1 0 0 1 1 0 1 1
state is : 2. Head is at position 6
current tape is : 1 0 0 1 1 0 1 0
state is : 2. Head is at position 5
current tape is : 1 0 0 1 1 0 0 0
state is : 9. Head is at position 4
current tape is : 1 0 0 1 1 1 0 0

No rule found for the current state : 9 , job is done.
Exited with :
1 0 0 1 1 1 0 0
```

Chapter 5

Code complet

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<unistd.h>
#include <string.h>
#define TAILLEMAX 2000

struct rule {
    int cur_state;
    int symbol;
    int new_symbol;
    int direction;
    int new_state;
};
typedef struct rule rule;

struct vect_rule {
    int nb_elem;
    rule *p;
};
typedef struct vect_rule vect_rule;

struct vect_tape{
    int nb_elem;
    char *p;
};
typedef struct vect_tape vect_tape;
```

```

vect_tape init (char * file_tape){
    FILE * file;
    int i, taille;
    char init_tape[TAILLEMAX], c;
    vect_tape output_tape;

    memset(init_tape, 2, TAILLEMAX);
    file = fopen(file_tape, "r");
    if (file == NULL){
        printf("Error loading the file: %s. Either it
            ↳ was misspelled or it does not exist.\n",
            ↳ file_tape);
        exit(0);
    }else {
        fseek(file, 0L, SEEK_END);
        taille =ftell(file);
        if (taille == 0){
            printf("This tape file is empty, aborting
                ↳ ... \n");
            exit(0);
        }
        fseek(file, 0L, SEEK_SET);
        for (i =0; i < taille; i ++){
            fscanf(file, "%c", &c);
            fseek(file, 0L, SEEK_CUR);
            init_tape[i] = atoi(&c);
        }
        output_tape.nb_elem = i+1;
        output_tape.p = malloc(output_tape.nb_elem *
            ↳ sizeof(char));
        for(i = 0; i < output_tape.nb_elem; i++){
            output_tape.p[i+5] = init_tape[i];
        }
    }
    fclose(file);
    return output_tape;
}

vect_rule rule_generator (char * file_rule){

```

```

    int line_number = 0;
    int ligne;
    char tmp;
    vect_rule output_rules;
    FILE * file;
    file = fopen(file_rule, "r");

    for(tmp = getc(file); tmp != EOF; tmp = getc(file)){
        if ( tmp == '\n')
            line_number++;
    }
    output_rules.p = malloc(line_number*sizeof(rule));
    assert(output_rules.p);
    output_rules.nb_elem = line_number;
    fseek(file, 0, SEEK_SET);
    for (ligne = 0; ligne < line_number; ligne++ ){
        fscanf(file, "%d_%d_%d_%d", &output_rules.p[
            ↪ ligne].cur_state, &output_rules.p[ligne].
            ↪ symbol, &output_rules.p[ligne].new_symbol,
            ↪ &output_rules.p[ligne].direction, &
            ↪ output_rules.p[ligne].new_state);
    }
    fclose(file);
    return output_rules;
}

int turing_machine (vect_tape init_tape, vect_rule rule_list,
    ↪ int cur_state, int verbose){
    int head_pos, i, n;
    char rule_found;
    head_pos = 5;
    while(1){
        rule_found = 0;
        for (i = 0; i < rule_list.nb_elem; i++){
            if (cur_state == rule_list.p[i].cur_state
                ↪ && init_tape.p[head_pos] ==
                ↪ rule_list.p[i].symbol){
                rule_found = 1;
                init_tape.p[head_pos] = rule_list.
                    ↪ p[i].new_symbol;
                if (rule_list.p[i].direction)

```



```

        head_pos++;
    else
        head_pos--;
    cur_state = rule_list.p[i].
        ↪ new_state;
    break;
}
}
if(!rule_found){
    printf("\nNo rule found for the current
        ↪ state: %d, job is done.",
        ↪ cur_state);
    printf("\nExited with:\n");
    for(int n = 5; n > 0; n--){
        if (init_tape.p[n-1] == 1)
            printf("%d", init_tape.p[n
                ↪ -1]);
    }
    for (n = 0; n < init_tape.nb_elem-1; n
        ↪ ++){
        printf("%d", init_tape.p[n+5]);
    };
    printf("\n");
    return 0;
}
if (verbose == 1){
    printf("state is: %d. Head is at
        ↪ position %d\n", cur_state,
        ↪ head_pos);
    printf("current tape is:");
    for(int n = 5; n > 0; n--){
        if (init_tape.p[n-1] == 1)
            printf("%d", init_tape.p[n
                ↪ -1]);
    }
    for (int n = 0; n < init_tape.nb_elem-1;
        ↪ n ++){
        printf("%d", init_tape.p[n+5]);
    };
    printf("\n");
}
}

```

```

    }
}

int main (int argc, char *argv[]){
    int verbose, n;
    vect_tape init_tape;
    vect_rule rule_list;

    if(argv[4] && !strcmp(argv[4], "-v"))
        verbose = 1;
    else{
        printf("Note: call with -v for detailed
        ↪ processing.\n\n");
        verbose = 0;
    }
    if (argc < 4 || argc > 5 || !strcmp(argv[1], "-help")){
        printf("Usage: %s<tape_file><rule_file><
        ↪ initial_state><-v>(last_argument_is
        ↪ optional_and_enables_verbose_mode)\n\nRule
        ↪ pattern must be: current_state
        ↪ found_symbol new_symbol movement_direction
        ↪ (where 0 is left and 1 is right) new_state
        ↪ WITH SPACES.\nTape pattern must be a
        ↪ chain of boolean numbers not separated by
        ↪ anything. The '2' symbol will represent
        ↪ blank spaces, rules must be set
        ↪ accordingly.\n\nHead starting position is
        ↪ at the begining of tape. The machine will
        ↪ halt anytime it detects a symbol for
        ↪ which it doesn't know any rule to apply.\n
        ↪ ", argv[0]);
        exit(0);
    }
    init_tape = init(argv[1]);
    printf("initial call done with:");
    for (n = 0; n <= init_tape.nb_elem-2; n ++){
        printf("%d", init_tape.p[n+5]);
    };
    printf("\n");
    rule_list = rule_generator(argv[2]);
    printf("Processing...\n\n");

```

```
turing_machine(init_tape, rule_list, atoi(argv[3]),  
    ↪ verbose);  
return 0;  
}
```