CR TEA 2

INTRODUCTION

Ce compte-rendu présente la résolution du jeu “Le compte est bon”. Pour résoudre ce problème, nous avons utilisé la méthodologie de recherche à la volée en profondeur d’abord.

D’une part, nous avons révisé des fichiers présentés dans le cours comme ajouter un nouveau type de T_elt dans le fichier elt.h. D’autre part, nous avons rédigé quelques nouveaux fichiers pour stocker nos propres fonctions comme rpn.c et explor.c.

Pour exécuter le programme, il faut saisir la commande dans la ligne de commande dans l’ordre : ./non_du _dossier (6 cartons) (1 nombre à trouver).

DEVELOPPEMENT

Nous avons d’abord ajouté au module de gestion de T_elt un type ELT_RPN pour représenter les éléments extraits d’une expression RPN et les résultats d’une expression RPN. Nous avons utilisé quelques macros comme PLUS et NON_ENTIER pour simplifier les expressions des variables.

Après la définition du T_elt, nous avons implémenté deux fonctions demandées : T_list s2list(char * exp) et T_elt rpn_eval(char * exp). Mais nous trouvons que la variable de retour de s2list, T_list, et les fonctions pour l’implémentation static contiguous sont incompatibles. Donc pour simplifier le programme, nous avons choisi le type T_stack comme la variable de retour de s2list. De plus, pour afficher les opérations à réaliser, nous avons implémenté une nouvelle fonction void show_rpn(char * exp) dans le fichier rpn.c.

Nous avons terminé la partie 1 avec une constante symbolique située dans rpn.h permettant de sélectionner le type d’implémentation. Après avoir fini ce TEA, nous avons testé la vitesse du programme par 3 implémentations différentes en saisissant l’exemple donné dans la fiche. Voici nos résultats :

IMPLEMENTATION_STATIC_CONTIGUOUS

IMPLEMENTATION_DYNAMIC_CONTIGUOUS

```
lcl@HW:~/Desktop/cwork/seance_2/compteestbon$ make
Production de compteestbon.exe à partir des fichiers : main.c elt.c list.c stack
_cs.c stack_cd.c stack_cld.c rpn.c explor.c
gcc -Wall main.c elt.c list.c stack_cs.c stack_cd.c stack_cld.c rpn.c explor.c -
o compteestbon.exe
Le programme compteestbon.exe a été produit dans le répertoire compteestbon
lcl@HW:~/Desktop/cwork/seance_2/compteestbon$ ./compteestbon.exe 3 5 7 9 25 50 8
59
7 + 25 = 32
32 * 9 = 288
3 * 288 = 864
864 - 5 = 859
La duree d'execution du programme est : 1.163238 secondes.
```

IMPLEMENTATION_DYNAMIC_LINKED

```
lcl@HW:~/Desktop/cwork/seance_2/compteestbon$ make
Production de compteestbon.exe à partir des fichiers : main.c elt.c list.c stack
_cs.c stack_cd.c stack_cld.c rpn.c explor.c
gcc -Wall main.c elt.c list.c stack_cs.c stack_cd.c stack_cld.c rpn.c explor.c -
o compteestbon.exe
Le programme compteestbon.exe a été produit dans le répertoire compteestbon
lcl@HW:~/Desktop/cwork/seance_2/compteestbon$ ./compteestbon.exe 3 5 7 9 25 50 8
59
7 + 25 = 32
32 * 9 = 288
3 * 288 = 864
864 - 5 = 859
La duree d'execution du programme est : 1.458390 secondes.
```

D’après les résultats obtenus, nous trouvons que le programme implémenté par implémentation contiguous est le plus performant.

Pour la partie 2, nous pensons que c’est beaucoup plus difficile que la partie 1 et nous avons rencontré beaucoup de problèmes. Par exemple, nous appelons récursivement la procédure de recherche mais nous trouvons que c’est difficile de déboguer quand nous rencontrons une erreur.

Afin de représenter les cartons restant à utiliser dans les appels, nous avons choisi d' utiliser un tableau d’entier et nous écrivons aussi les macros des opérateurs dans ce tableau. Si un élément est utilisé, nous utilisons 0 pour représenter la place vacante et nous trouvons que c’est très efficace.

Voici le prototype de notre fonction :

```c
#include "explor.h"

//Chercher la solution le plus proche
char* explor(int n1, int n2 , int n3, int n4, int n5, int n6, int num_goal){
        int rest[10] = {n1, n2, n3, n4, n5, n6, PLUS, MOIN, MULT, DIVI};
        int resolu[11] = {0,0,0,0,0,0,0,0,0,0,0};
        T_elt t = {0,ENTIER};
        char best[30];
        char * explor;
```

```
11          if((n1==num_goal)||(n2==num_goal)||(n3==num_goal)||(n4==num_goal)||(n5==num_goal)||(n6==num_goal)){
12                  memset(best, 0, sizeof(best));
13                  sprintf(best,"%d ",num_goal);
14                  explor = best;
15                  printf("%s\n",explor);
16                  return explor;
17          }
18          develop(rest, num_goal, resolu, t, ECART, best);
19          explor = best;
20          return explor;
21  }
22
23
24  //La fonction develop() est pour
25  int develop(int * rest, int num_goal, int * resolu, T_elt t, int min_ecart, char * best){
26          char * s;
27          if(t.status != ENTIER) return 0; //Eviter des expressions inutiles
28          if(min_ecart == 0) return 1; //Si la meilleure solution est trouvee, return 1
29          if( (*(resolu+10)) != FIN){
30                  s = tab2s(resolu);
31                          if(strlen(s) != 0){
32                                  t = rpn_eval(s);
33                          }
34                  T_elt reponse = rpn_eval(s);
35                  free(s);
36                  if( abs(reponse.value - num_goal) < min_ecart){
37                          s = tab2s(resolu);
38                          if(strlen(s) != 0){
39                                  t = rpn_eval(s);
40                          }
41                          strcpy(best,s);
42                          free(s);
43                          min_ecart = reponse.value;
44                          if(min_ecart == 0) return 1;
45                  }
46                  return 0;
47          }
48
49          s = tab2s(resolu);

50
51          if(strlen(s) != 0){
52                  T_elt reponse = rpn_eval(s);
53                  free(s);
54                  if( abs(reponse.value - num_goal) < min_ecart){
55                          s = tab2s(resolu);
56                          if(strlen(s) != 0){
57                                  t = rpn_eval(s);
58                          }
59                          strcpy(best, s);
60                          free(s);
61                          min_ecart = reponse.value - num_goal;
62                  }
63                  if( abs(reponse.value - num_goal) == 0){
64                          s = tab2s(resolu);
65                          if(strlen(s) != 0){
66                                  t = rpn_eval(s);
67                          }
68                          strcpy(best, s);
69                          free(s);
70                          min_ecart = 0;
71                          return 1;
72                  }
73          }
74
75          int i,j;
76          int num_next;
77          *(rest+6) = PLUS;
78          *(rest+7) = MOIN;
79          *(rest+8) = MULT;
80          *(rest+9) = DIVI;
81          for(i=0;i<10;i++){
82                  if( (*(rest+i)) != 0){
83                          num_next = *(rest+i);
84                          *(rest+i) = 0;
85                          j=0;
86                          while((resolu[j]!=0)&&(j<10)) j++;
87                          resolu[j] = num_next;
88                          s = tab2s(resolu);
89                          t = rpn_eval(s);
90                          free(s);
91                          int eval = develop(rest, num_goal, resolu, t, min_ecart, best);
92                          if(eval == 1) return 1; //Si la meilleure solution est trouvee, return 1 et la programme sera arretee.
93                          *(rest+i) = num_next;
94                          resolu[j]=0;
95                  }
96          }
97          return 0;
98  }
99
```

```c
//Transformer un tableau, par exemple resolu[], en une expression RPN sous forme de chaîne de caractères
char * tab2s(int * tab){
        int i;
        char * s;
        s = malloc(30*sizeof(char));
        memset(s, 0, 30);
        char int2s[5];
        for(i=0;i<11;i++){
                if(tab[i]!=0){
                        if(tab[i]>0){
                                memset(int2s, 0, sizeof(int2s));
                                sprintf(int2s,"%d ",tab[i]);
                                strcat(s,int2s);
                        }else{
                                switch(tab[i]){
                                        case PLUS :
                                                strcat(s,"+ ");
                                                break;
                                        case MOIN :
                                                strcat(s,"- ");
                                                break;
                                        case MULT :
                                                strcat(s,"* ");
                                                break;
                                        case DIVI :
                                                strcat(s,"/ ");
                                                break;
                                }
                        }
                }
        }
        return s;
}
```

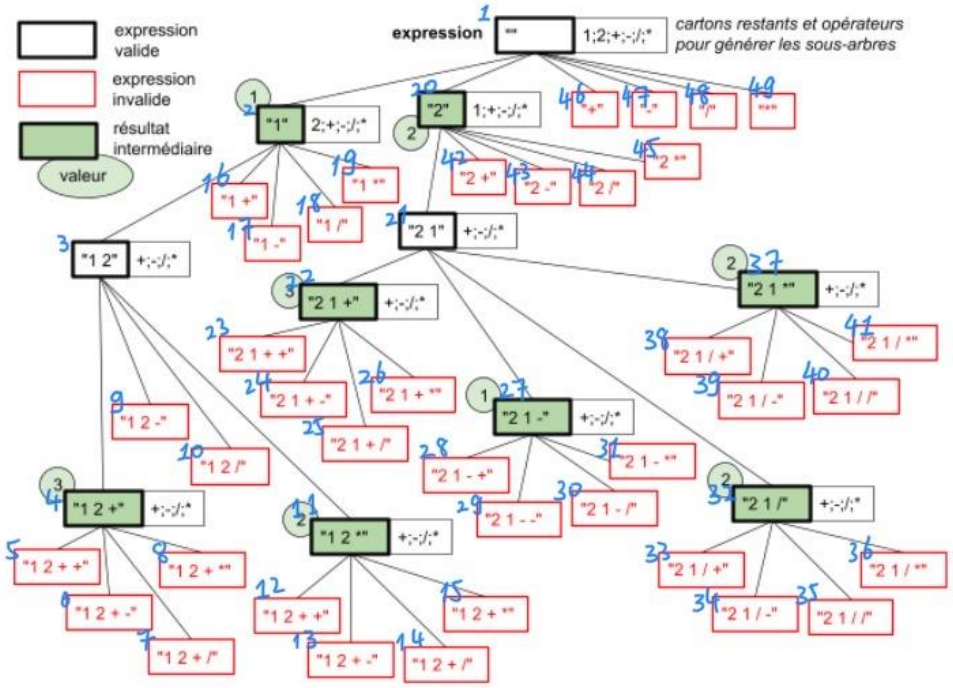
Voici l'algorithme permettant d'implémenter cette fonction :

1. Les cases restantes sont toutes les cases sauf dans les cas suivants : l'arbre suivant une valeur RPN qui n'est pas un résultat entier, l'arbre suivant le "nombre de chiffres <= nombre d'opérateurs + 1" dans l'expression RPN, l'arbre suivant le meilleur nombre trouvé, l'arbre qui a déjà été calculé…

2&3

```
int fonction developer(){

if(Expressions RPN qui n'ont pas de valeurs entières) return 0; //Ce nœud et l'arbre en dessous ne sont pas considérés;
//Eviter des expressions inutiles
if(la meilleure solution est trouvee) return 1; //Le nombre a été trouvé, et en sortant de la boucle,
//on évite d'entrer à nouveau dans le programme pour trouver une autre combinaison.
if(Déterminer si un numéro est disponible)
{
if(Les expressions RPN sont valides) {Calcul de la valeur d'une expression RPN}
Stocker la valeur de l'expression dans PILE
libérer de l'espace

if(L'expression RPN est plus proche du nombre recherché que l'expression précédente, mais pas égale.)
{Enregistrez la nouvelle expression RPN}

if(Si la valeur est exactement égale à la valeur à trouver)
{Renvoie 1, qui est utilisé pour quitter l'opération récursive.}

Mettre à jour le tableau qui indique si le numéro peut participer à l'opération.
ne change pas le nombre d'éléments de l'expression RPN, remplaçant le dernier élément
Déterminer si la fonction appelée a trouvé le numéro qu'elle cherche, si oui, sortir de la fonction.
dégager le signe que le numéro a été utilisé afin qu'il puisse être utilisé sur un autre arbre


}

}
```

D’après notre algorithme, les noeuds de l’arbre sont parcouru dans l’ordre préfixe comme le schéma ci-dessous :



CONCLUSION

D’après les exécutions du programme, nous pouvons trouver facilement que la vitesse de l’implémentation static contiguous mais elle occupe plus d’espace. Nous pouvons choisir l’implémentation dont nous avons besoin selon l’objectif du programme.

L’arbre est un type de structure efficace quand il y a beaucoup de situations pour un programme. Nous pouvons évaluer toutes les situations en parcourant l’arbre.

PERSPECTIVES

Améliorer l'efficacité des procédures du programme en libérant la mémoire en temps utile, en modifiant conception des algorithmes, etc.

Reconnaissance intelligente du nombre de chiffres et des chiffres saisis par l'utilisateur, demandant à l'utilisateur s'il doit continuer avec le prochain tour de recherche et une interface conviviale.

Optimiser l'algorithme pour enregistrer la valeur de l'expression RPN existante à chaque nœud et l'appeler directement lors de la poursuite de la recherche de branche, ce qui évite les calculs répétés et gagner le temps de calcul.

Lorsque l'expression RPN requise a été obtenue, trouvez un moyen de mettre fin directement au processus récursif, en remplaçant la méthode qui renvoie une valeur nulle pour mettre fin au processus.

L'ORGANISATION DU GROUPE

LIU Changle : la conception de l'algorithme, la rédaction de rpn.c et de CR, le débogage
LIU Yu : la conception de l'algorithme, la rédaction de explor.c et de CR, l'exécution et le test
QIU Ziyan : la conception de l'algorithme, la rédaction de explor.c, l'exécution et le test