

Programmation évolutive et application

TIPE - 2012

Maxime Niankouri

L'objectif de ce TIPE est de réaliser un programme évolutif général et de l'étudier sur un exemple particulier. Le résultat sera ensuite réexploité pour la conception d'un circuit électronique donnant la primalité d'un entier jusqu'à 255. Dans un second temps, on s'intéressera à l'efficacité d'un tel programme sur des problèmes plus simples et on en déduira la manière d'optimiser les paramètres du problème.

Table des matières

1	Programmation évolutive	2
1.1	Principe	2
1.2	Algorithme	2
2	Premier exemple : primalité jusqu'à $2^8 - 1$	2
2.1	Application du programme	3
2.2	Traiter les données brutes	3
2.3	Concrétisation	3
3	Approche statistique	4
4	Références	4
5	Annexes	5
5.1	Implémentation en C du programme	5
5.2	Programme auxiliaire	20
5.3	Circuit électronique	24

1 Programmation évolutive

1.1 Principe

Selon Darwin et la théorie de l'évolution, dans un environnement donné, une population d'individus évolue afin de s'adapter au mieux aux conditions qui l'entourent. On peut alors imaginer simuler un environnement et une population de fonctions à l'intérieur même d'un programme et laisser, tout comme dans les hypothèses des biologistes, le pseudo-hasard résoudre un problème. Sur cette idée, on pourra alors construire un algorithme en continuant l'analogie avec la biologie et l'évolution.

1.2 Algorithme

Pour commencer, il faut déterminer une structure de données qui permet de modifier aisément une fonction et de l'évaluer rapidement. Pour cela, on peut penser aux arbres et à l'écriture postfixée. En effet, cette écriture permet, après s'être placé aléatoirement sur un noeud, de récupérer tout le sous-arbre correspondant et d'ainsi le remplacer par le sous-arbre choisi.

L'utilisateur doit alors ensuite définir, en fonction du problème posé, les valeurs des étiquettes des noeuds internes notées A_i , les fonctions élémentaires, ainsi que les valeurs des feuilles notées x_i , les variables. Il doit, de même, définir une loi, faisant office d'environnement, permettant d'affixer à chaque fonction un score permettant de classer les fonctions selon leur niveau d'adaptation, leur degré d'approche d'une solution au problème.

Pour modifier une fonction, on procèdera par remplacement de sous-arbre par un sous-arbre élémentaire, ce qui est analogue à une mutation, ou par un sous-arbre d'une autre fonction dans la génération, analogue à un crossover (deux chromosomes qui se lient pour s'échanger des fragments de matériel génétique) qui permet d'inciter les arbres à s'étoffer au lieu de rester limité à quelques noeuds.

Au lancement de l'algorithme, soit on génère une génération première à partir de mutations d'arbres vides, soit on prend en source un fichier préexistant issu d'une ancienne utilisation du programme ou contenant des idées de résolution faites main. À partir de là, on peut commencer le processus d'évolution. Pour passer de la génération n , contenant kp fonctions, à la génération $n + 1$, on se propose de copier les p fonctions au meilleur score et d'abandonner les $(k - 1)p$ suivantes. On est alors assuré d'avoir (S_n) , suite associant à la génération n le score maximal de ses fonctions, croissante. Les fonctions manquantes peuvent ensuite être créées à partir des p fonctions copiées en leur appliquant des mutations ou des crossovers selon un taux de mutation/crossover = r .

Tout le temps de calcul se situe, en général, dans l'attribution du score. Il s'agit donc, dans un souci de gain de temps, de limiter les tailles des fonctions. Néanmoins, limiter la taille limite aussi les opportunités d'évolution. On peut alors imaginer deux phases. La première lâchant du lest aux fonctions mais leur imposant tout de même une limite L , supposée suffisamment grande. La seconde, classant les fonctions dont le score est minimal pour le passage à la génération suivante par rapport à leur taille. Cette dernière phase s'active quand le temps pour passer d'une génération à une autre est supérieur au temps t_1 sur plusieurs itérations. On pourra ensuite repasser à la première phase dès qu'on repassera sous un temps t_2 .

On poursuit ainsi jusqu'à ce que (S_n) atteigne le score désiré, auquel cas, soit on retourne la fonction solution, soit on entre en phase de réduction de taille jusqu'à l'arrêt du programme par l'utilisateur.

Une implémentation en langage C est proposée en annexe, appliquée à l'exemple qui va suivre et avec les valeurs $k = 2, p = 100, r = \frac{1}{9}, t_1 = 600ms, t_2 = 150ms$, et comportant diverses améliorations d'ordres techniques.

2 Premier exemple : primalité jusqu'à $2^8 - 1$

On cherche à réaliser un circuit électrique à base de puces CMOS et TTL, de 8 interrupteurs en entrée et d'une LED en sortie, indiquant si l'entier donné en écriture binaire par l'entrée est premier ou non.

On lance donc le programme décrit précédemment avec, pour fonctions élémentaires, les portes logiques ET, OU et NON usuelles, et pour loi régissant le score : $\text{Card}\{i \in [2; 2^8 - 1] / f(i) = \text{isprime}(i)\} + 2$.

2.1 Application du programme

La résolution du problème a pris une centaine d'heures à la machine, interrompue par diverses améliorations apportées au fil de l'exécution. Outre la réduction de taille, comme les processeurs possèdent aujourd'hui plus d'un cœur, pour gagner du temps, mieux vaut utiliser toutes les ressources disponibles. On peut alors chercher à répartir le calcul du score des kp fonctions en, par exemple, k blocs, chacun étant traité par un cpu, n'en laissant aucun au repos.

Le problème étant résolu, le score maximal atteint (256), on peut récupérer la fonction solution sous forme postfixée :

$$\left[\begin{array}{l} wuEsENtxOyNvOEoysuOxtENwEOONxvEtOENEztxvOONoxtOvENwsxtEOO \\ EtwsOxEEENEyuENOEsxtENwEEEoEsvEntEwxvsOEEEowvOstyOONEextE \\ OvsOEEEnvyuOsEEwENetswuOONxOOEwsOswuEvOENExtvOENEysxEouOEN \\ EwsuOEyOzstvwEEyExEENyuENTxOvOEEENEEOE \end{array} \right]$$

Avec $n = \overline{stuvwxyz}$, l'entrée.

2.2 Traiter les données brutes

Le résultat précédent est évidemment intraitable tel quel. Il est donc raisonnable de chercher à transformer cette chaîne de caractères en arbre au format image afin de mieux visualiser les choses.

À l'aide de Graphviz, outil générant un graphe à partir d'un code source, ainsi que d'un programme intermédiaire (cf. code en annexe) convertissant l'écriture postfixée en fichier lisible par ce dernier et en profitant pour donner diverses informations, on obtient l'arbre suivant :

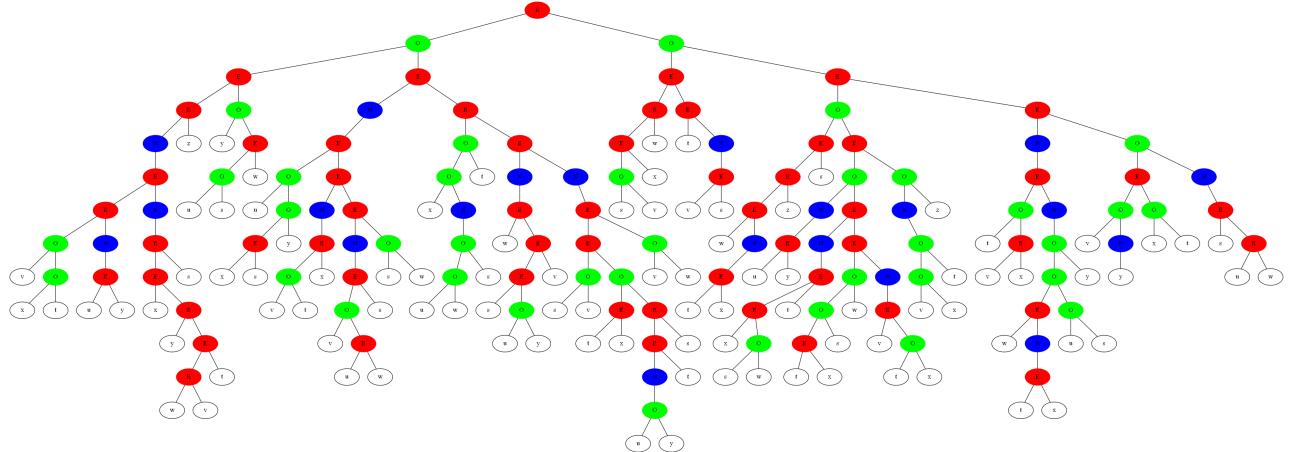


FIGURE 1 – Arbre solution du problème

Celui-ci peut être interprété comme la représentation d'un circuit logique composé uniquement de portes ET (E, rouge), OU (O, vert) et NON (N, bleu).

2.3 Concrétisation

La composition étant établie, il faut maintenant procéder au montage. Celui-ci s'est fait sur une plaque adaptée (cf. photos en annexe) à l'aide de portes CMOS AND et de portes TTL OR et NOT, ces dernières ayant été récupérées gracieusement. Une alimentation de 5V étant rigoureusement nécessaire pour les TTL, une alimentation par l'intermédiaire d'un câble USB convient.

Avant même de commencer les centaines de soudures nécessaires, il convient de convertir le plan-arbre ci-dessus en un plan adapté à l'architecture de la plaque et des composants, ainsi que de l'optimiser en réduisant le nombre de fils (un maximum de ponts d'étain) et en limitant leur longueur. Ici encore, l'utilisation de l'algorithme évolutif semble adapté, en travaillant sur une matrice (correspondant aux trous de la plaque) et en établissant la loi selon les critères précédents.

Le fer rangé, on se retrouve avec le circuit tant convoité (cf. photos en annexe).

3 Approche statistique

Les courbes ci-dessous, réalisées sur 1000 recherches de solution au problème de la primalité sur 2^4 préconisent, en tout cas pour un problème court, une valeur de r de $\frac{1}{2}$.

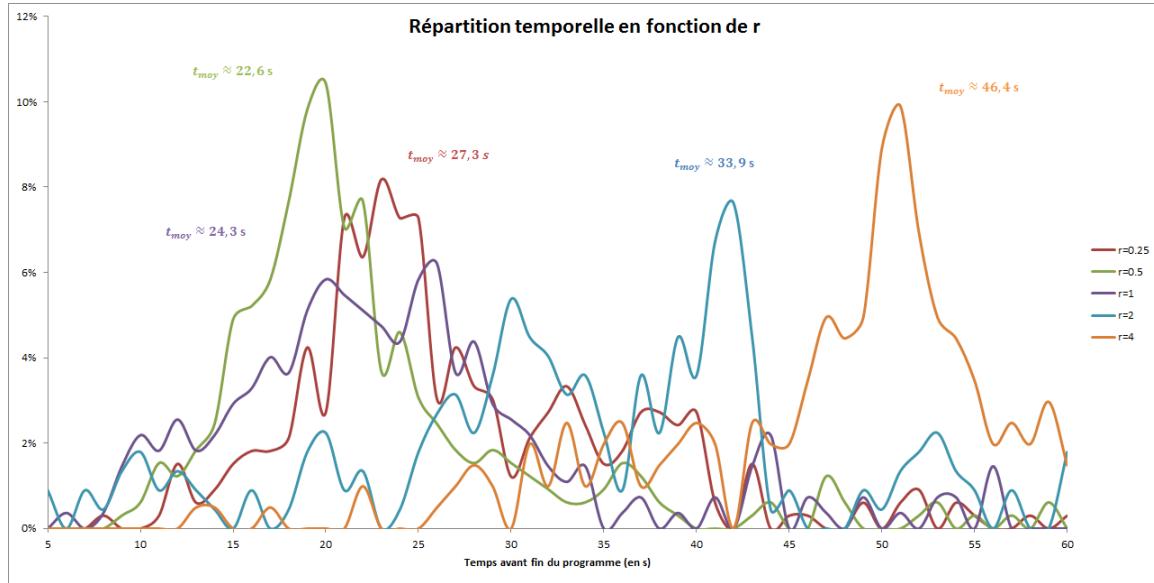


FIGURE 2 – Répartition temporelle en fonction de $r = \frac{\text{mutation}}{\text{crossover}}$

4 Références

Graphviz : <http://www.graphviz.org>

Documentation du langage C : <http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html>

Documentation sur les threads : <https://computing.llnl.gov/tutorials/pthreads/>

Ainsi que mes connaissances personnelles et mes cours en CPGE.

5 Annexes

5.1 Implémentation en C du programme

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <pthread.h>
5
6 #define SCOREMAX 256
7 #define LONGUEURLIGNE 2000
8 #define NOMBRELIGNE 200
9
10 /* _____
11
12             Maxime Niankouri
13             21.06.2011
14             TIPE
15
16     But : Creer, par l'intermediaire d'un algorithme genetique
17     une fonction isprime se basant sur l'ecriture binaire d'un
18     entier, composee de portes logiques et d'ainsi developper,
19     par la suite, un circuit electronique ayant la meme fonction.
20
21 _____*/
22
23 int nextgen(FILE* parents , FILE* generation , int n);
24 int calcul_score(FILE* parents);
25 int lecture(FILE *generation , int position , int s , int t , int u , int v , int w ,
26             int x , int y , int z);
27 void mutation(FILE* parents , FILE* generation , char *insert);
28 void genere_fichier_aleatoire();
29 void mutation_aleatoire(FILE* parents , FILE *generation);
30 char variable_aleatoire();
31 void crossover(FILE* parents , FILE* generation);
32 FILE* genere_fichier(int n);
33 FILE* ouvre_fichier(int n);
34 int isprime(int n);
35 void binarise(int n , int *t , int p);
36 void Convertisseur_longueur_ligne(int avant); // Convertit generation0001 dans
37 int calcul_taille(FILE* gen);
38 int evaluer(char* fonction , int s , int t , int u , int v , int w , int x , int y ,
39               int z);
40 void get_fonction(FILE* gen , char *fonction);
41 void* thread_score1(void* arg);
42 void* thread_score2(void* arg);
43 int prime[SCOREMAX+1]; // liste des nombres premiers
44 pthread_t thread1 , thread2;
```

```

45 int t_phase = 0; // nombre de generations dans les limites pour le changement
   de phase
46 int phase=1; // 0: severe // 1: lache
47
48
49
50 int main()
51 {
52     srand(time(NULL));
53
54     int t1,t2;
55     t1=clock(); // Donnee temporelle
56
57     // On remplit la liste des nombres premiers
58     int p=0;
59     while(p<=SCOREMAX)
60     {
61         prime[p]=isprime(p);
62         p++;
63     }
64
65
66     FILE *generation[10000];
67     generation[1]=NULL;
68     generation[1]=fopen("generation0001.txt","r"); // Si le fichier existe
69           deja
70
71     if(generation[1]==NULL) // Si on part d'un fichier frais
72     {
73         genere_fichier_aleatoire();
74         generation[1]=fopen("generation0001.txt","r");
75     }
76
77     int i=2, j=2;
78     int final=-1;
79     while(i<10000 && final== -1)
80     {
81         printf("%d\n",j);
82         fseek(generation[i-1],0,SEEK_SET);
83         generation[i]=genere_fichier(i);
84         final = nextgen(generation[i-1], generation[i], i);
85         fclose(generation[i-1]);
86         i++;
87         j++;
88         if(i==9901) // Si on veut une boucle infinie.
89         {
90             rename("generation9900.txt","generation0100.txt");
91             i=101;
92         }
93     }

```

```

94     fclose(generation[i-1]);
95
96     printf("La ligne finale : %d", final);
97     t2=clock();
98     printf("\n\nTemps d'execution : %ds\n\n", (t2-t1)/CLOCKS_PER_SEC);
99
100    system("speaker-test-t-sine"); // Sous linux : alerte sonore
101
102    //getchar();
103    return 0;
104 }
105
106
107 // Fonction qui passe d'une generation a la suivante :
108 int nextgen(FILE* parents, FILE* generation, int n)
109 {
110     if(t_phase==10) phase=1-phase;
111
112
113     int t1,t5; // Pour les tests de temps
114     t1=clock();
115     int score[200];
116     int i=0;
117     while(i<200)
118     {
119         fseek(parents, LONGUEUR_LIGNE*i, SEEK_SET);
120         score[i]=calcul_score(parents);
121         if(!i) // Deux calculs de la premiere ligne
122         {
123             fseek(parents, LONGUEUR_LIGNE*i, SEEK_SET);
124             score[i]=calcul_score(parents);
125         }
126         //if(score[i]==SCOREMAX) return i+1; // On a fini le prog, la
127         //solution se trouvant a la ieme ligne
128         // Ligne precedente retiree : on cherche aussi a avoir la
129         //ligne la plus courte.
130         i++;
131     }
132
133     // Ici il faut trier score[]. Ou, en tout cas, recuperer les lignes des
134     //100 meilleures.
135     int repartition[SCOREMAX+1];
136     i=0;
137     while(i<=SCOREMAX) // On commence par tout mettre a 0 ...
138     {
139         repartition[i]=0;
140         i++;
141     }
142     i=0;
143     while(i<200) // On remplit le tableau
144     {

```

```

142     repartition [ score [ i ]]++;
143     i++;
144 }
145
146 int min=SCOREMAX; // Score minimum transferable
147 int tot=0;
148 while( tot<100)
149 {
150     tot+=repartition [ min ];
151     min--;
152 }
153 min++;
154
155
156 // Affichage en console
157 i=min-10;
158 if(i<0) i=0;
159 printf ("\nEtat de la Repartition :\n");
160 while(i<=min+10 && i<=SCOREMAX)
161 {
162     printf ("Score : %d---%d\n", i , repartition [ i ]);
163     i++;
164 }
165
166
167 // Le second critere de repartition : la taille
168 char s[LONGUEUR_LIGNE];
169 i=0;
170 int j=0;
171 int k=0, taille [200], max_taille=0, repartition_taille [LONGUEUR_LIGNE+1];
172
173 while(i<=LONGUEUR_LIGNE) // On commence par tout mettre a 0 ...
174 {
175     repartition_taille [ i ]=0;
176     i++;
177 }
178 i=0;
179
180 /// On pourrait commencer par prendre ceux de la fin du fichier , afin
181      de privilegier les nouveaux.
182      /// => Augmentation rapide la longueur d'une ligne ...
183 while(j<100)
184 {
185     if (i==200) // On compare par la taille
186     {
187         while(k<200)
188         {
189             if (score [ k]==min)
190             {
191                 fseek ( parents ,k*LONGUEUR_LIGNE,SEEK_SET );
192                 taille [ k]=calcul_taille ( parents );

```

```

192         repartition_taille [ taille [k]]++;
193     }
194     k++;
195   }
196   k=0;
197   while (k+j < 100)
198   {
199     k+=repartition_taille [ max_taille ];
200     max_taille++;
201   }
202   printf ("MAX_TAILLE=%d", max_taille-1); // Affichage a l 'ecran
203
204
205 }
206 if (score [ i]>min && i<200) // Les scores > min
207 {
208   // On copie simplement la ligne
209   fseek (parents ,LONGUEUR_LIGNE*i ,SEEK_SET);
210   fscanf (parents ,"%s" ,s);
211   fseek (generation ,LONGUEUR_LIGNE*j ,SEEK_SET);
212   fprintf (generation ,"%s" ,s);
213   j++;
214 }
215   // Lorsqu'on a le choix ( a min ) :
216   // On commence par la fin et on prend les plus courts.
217   // OU On commence par la fin et on laisse la taille libre
218   // de maniere a laisser respirer les fonctions
219 else if (score[399-i]==min && (taille[399-i]<max_taille-1 || (taille
220 [399-i]<1000 && phase)) && i>=200 && i<400)
221 {
222   fseek (parents ,LONGUEUR_LIGNE*(399-i) ,SEEK_SET);
223   fscanf (parents ,"%s" ,s);
224   fseek (generation ,LONGUEUR_LIGNE*j ,SEEK_SET);
225   fprintf (generation ,"%s" ,s);
226   j++;
227 }
228 else if (score[599-i]==min && taille[599-i]==max_taille-1 && i>=400)
229   // On complete jusqu'a 100
230 {
231   fseek (parents ,LONGUEUR_LIGNE*(599-i) ,SEEK_SET);
232   fscanf (parents ,"%s" ,s);
233   fseek (generation ,LONGUEUR_LIGNE*j ,SEEK_SET);
234   fprintf (generation ,"%s" ,s);
235   j++;
236 }
237
238
239 // Vient l 'etape de mutation/crossover
240

```

```

241 // On ouvre un deuxième stream sur l'enfant.
242 int r=0;
243 FILE* enfants=NULL;
244 enfants=ouvre_fichier(n);
245
246 i=0;
247 while(i<100)
248 {
249     fseek(génération, LONGUEUR_LIGNE*i, SEEK_SET); // La zone des 100
250     premiers
251     fseek(enfants, LONGUEUR_LIGNE*(i+100), SEEK_SET); // La zone des
252     transformes
253
254     r=rand()%10;
255     if(r) crossover(génération, enfants);
256     else mutation_aleatoire(génération, enfants);
257
258     i++;
259 }
260 fclose(enfants);
261
262 t5=clock();
263 printf("\n-->%d ms\n", 1000*(t5-t1)/CLOCKS_PER_SEC);
264
265 // Pour la gestion de changement de phase
266 if(1000*(t5-t1) > 600*CLOCKS_PER_SEC || 1000*(t5-t1) < 150) t_phase++;
267 else t_phase=0;
268
269 return -1; // Cette génération n'apporte pas encore de solutions
270 convenables.
271
272 // Calcul du score d'une fonction
273 int calcul_score(FILE* parents)
274 {
275     int score=2;
276     int i=2, t[26], j=2;
277     char fonction[LONGUEUR_LIGNE+3];
278     get_fonction(parents, fonction); // recuperer le string-fonction
279
280     // cf les deux threads
281     pthread_create(&thread1, NULL, thread_score1, (void*)fonction);
282     pthread_create(&thread2, NULL, thread_score2, (void*)fonction);
283     int v1=0, v2=0;
284     int *a1, *a2;
285     pthread_join(thread1, (void**)&a1);
286     pthread_join(thread2, (void**)&a2);
287     v1= *a1;
288     v2= *a2;

```

```

289     return v1+v2;
290 }
291
292
293 // Fonction de lecture de fonctions pour le score
294 int lecture(FILE *generation, int position, int s, int t, int u, int v, int w,
295             int x, int y, int z)
296 {
297     fseek(generation, position, SEEK_SET);
298
299     int pile[LONGUEUR_LIGNE/2]; // Le /2 est pris pour diminuer un peu la
300     // taille de la pile.
301     char caractere='0';
302     pile[0]=0;
303
304     while(caractere!='\n' && caractere!=EOF' && caractere!='F')
305     {
306         caractere=fgetc(generation);
307         switch(caractere)
308         {
309             case 'N': // NON
310                 if(pile[pile[0]]) pile[pile[0]]=0;
311                 else pile[pile[0]]=1;
312                 break;
313             case 'E': // ET
314                 if(pile[pile[0]]&&pile[pile[0]-1]) pile[pile[0]-1]=1;
315                 else pile[pile[0]-1]=0;
316                 pile[0]--;
317                 break;
318             case 'O': // OU
319                 if(pile[pile[0]]||pile[pile[0]-1]) pile[pile[0]-1]=1;
320                 else pile[pile[0]-1]=0;
321                 pile[0]--;
322                 break;
323             // VARIABLES
324             case 's':
325                 pile[0]++;
326                 pile[pile[0]]=s;
327                 break;
328             case 't':
329                 pile[0]++;
330                 pile[pile[0]]=t;
331                 break;
332             case 'u':
333                 pile[0]++;
334                 pile[pile[0]]=u;
335                 break;
336             case 'v':
337                 pile[0]++;
338                 pile[pile[0]]=v;
339                 break;

```

```

338     case 'w':
339         pile[0]++;
340         pile[pile[0]]=w;
341         break;
342     case 'x':
343         pile[0]++;
344         pile[pile[0]]=x;
345         break;
346     case 'y':
347         pile[0]++;
348         pile[pile[0]]=y;
349         break;
350     case 'z':
351         pile[0]++;
352         pile[pile[0]]=z;
353         break;
354     }
355 }
356 if(caractere=='EOF') return -1;
357 else return pile[1];
358 }
359
360
361 // Fonction de mutation
362 void mutation(FILE* parents , FILE* generation , char *insert)
363 {
364     int taille=-1;
365     char c='0';
366     while(c!='\n' && c!=EOF' && c!=F')
367     {
368         c=fgetc(parents);
369         taille++;
370     }
371     fseek(parents,-taille-1,SEEK_CUR);
372     // Apres avoir releve la taille de l'engin , on choisit un noeud a modifier
373
374     int lieu=rand()%taille;
375     fseek(parents,lieu,SEEK_CUR);
376     // On regarde ensuite jusqu'o s'estend son evaluation .
377     int eval=1;
378     int longueur=0;
379     char lu='0';
380     while(eval) // en fonction des arites des fonctions elementaires
381     {
382         longueur++;
383         lu=fgetc(parents);
384         if(lu=='O' || lu=='E') eval++;
385         else if(lu=='N') eval--;
386         if(eval)fseek(parents,-2,SEEK_CUR);
387         else fseek(parents,-1,SEEK_CUR);
388     }

```

```

388
389 // On va ensuite lire la fonction en omettant la partie ainsi definie.
390 fseek(parents, longueur-lieu-1, SEEK_CUR);
391 char s1[LONGUEUR_LIGNE], s2[LONGUEUR_LIGNE];
392 if(lieu-longueur+2>0) fgets(s1, lieu-longueur+2, parents);
393 fseek(parents, longueur, SEEK_CUR);
394 if(taille-lieu>0) fgets(s2, taille-lieu, parents);
395
396 // Puis on printf ca, intercale par le string desire dans la generation
397 // enfant.
398 fprintf(generation, "%s%s%sF", s1, insert, s2);
399 // A la fin, on se situe au debut de la prochaine fonction, dans les deux
400 // fichiers.
401 }
402
403 // Pour avoir un generation0001.txt frais
404 void genere_fichier_aleatoire()
405 {
406     FILE *gen=NULL;
407     gen=genere_fichier(1);
408     int i=0;
409     while(i<200) // On remplit de fonctions elementaires
410     {
411         fseek(gen, i*LONGUEUR_LIGNE, SEEK_SET);
412         switch(rand()%5)
413         {
414             case 0:
415                 fprintf(gen, "%cF", variable_aleatoire());
416                 break;
417             case 1:
418                 fprintf(gen, "%cF", variable_aleatoire());
419                 break;
420             case 2:
421                 fprintf(gen, "%cNF", variable_aleatoire());
422                 break;
423             case 3:
424                 fprintf(gen, "%c%cEF", variable_aleatoire(), variable_aleatoire());
425                 break;
426             case 4:
427                 fprintf(gen, "%c%cOF", variable_aleatoire(), variable_aleatoire());
428                 break;
429         }
430         i++;
431     }
432     fclose(gen);
433 }
434
435 void mutation_aleatoire(FILE* parents, FILE *generation)
436 {

```

```

437     char insert [] = "000";
438     // J'ai choisi 2/5 chance d'avoir une simple variable
439     switch (rand() % 5)
440     {
441         case 0:
442             insert [0] = variable_aleatoire ();
443             insert [1] = '\0';
444             break;
445         case 1:
446             insert [0] = variable_aleatoire ();
447             insert [1] = '\0';
448             break;
449         case 2:
450             insert [0] = variable_aleatoire ();
451             insert [1] = 'N';
452             insert [2] = '\0';
453             break;
454         case 3:
455             insert [0] = variable_aleatoire ();
456             insert [1] = variable_aleatoire ();
457             insert [2] = 'E';
458             insert [3] = '\0';
459             break;
460         case 4:
461             insert [0] = variable_aleatoire ();
462             insert [1] = variable_aleatoire ();
463             insert [2] = 'O';
464             insert [3] = '\0';
465             break;
466     }
467     mutation (parents, generation, insert);
468 }
469
470
471 char variable_aleatoire ()
472 {
473     switch (rand () % 8)
474     {
475         case 0:
476             return 's';
477             break;
478         case 1:
479             return 't';
480             break;
481         case 2:
482             return 'u';
483             break;
484         case 3:
485             return 'v';
486             break;
487         case 4:

```

```

488         return 'w';
489         break;
490     case 5:
491         return 'x';
492         break;
493     case 6:
494         return 'y';
495         break;
496     case 7:
497         return 'z';
498         break;
499     }
500 }
501
502
503 // Fonction de crossover
504 void crossover(FILE* parents , FILE* generation)
505 {
506     char insert [LONGUEUR_LIGNE];
507     // On prend l'insert aleatoirement
508
509     int taille=-1;
510     char c='0';
511     while(c!='\n' && c!=EOF && c!='F')
512     {
513         c=fgetc(parents);
514         taille++;
515     }
516     fseek(parents,-taille-1,SEEK_CUR);
517     // Apres avoir releve la taille de l'engin , on choisit un noeud a modifier
518
519     int lieu=rand()%taille;
520     fseek(parents,lieu,SEEK_CUR);
521     // On regarde ensuite jusqu'où s'estend son evaluation .
522     int eval=1;
523     int longueur=0;
524     char lu='0';
525     while(eval)
526     {
527         longueur++;
528         lu=fgetc(parents);
529         if(lu=='O' || lu=='E') eval++;
530         else if(lu=='N') eval--;
531         if(eval)fseek(parents,-2,SEEK_CUR);
532         else fseek(parents,-1,SEEK_CUR);
533     }
534     // Et on balance ce qu'on veut dans l'insert
535     fgets(insert,longueur+1,parents);
536 }
```

```

537 // On se place sur le deuxième parent (avec i entre 0 et 99) de l'enfant
538 // et on mute.
539 fseek( parents ,LONGUEUR_LIGNE*(rand()%100),SEEK_SET);
540 mutation( parents ,generation ,insert );
541 }
542 // S'occupe de generer un generationXXXX.txt vierge , dans le bon format
543 // cad : LONGUEUR_LIGNE*NOMBRE_LIGNE espaces
544 FILE* genere_fichier(int n)
545 {
546     char s []="generation0000.txt";
547     // On supprime le fichier inutilise pour economiser de l'espace disque .
548     if(n>3)
549     {
550         sprintf(s , "generation%d%d%d%d.txt", (n-2)/1000,((n-2)%1000)/100, ((n-2)
551             %100)/10, (n-2)%10);
552         remove(s );
553     }
554     sprintf(s , "generation%d%d%d%d.txt", n/1000,(n%1000)/100, (n%100)/10, n%10);
555
556     FILE* out=NULL;
557     out=fopen (s , "w");
558     int i=0,j=0;
559     while(i<200)
560     {
561         while(j<LONGUEUR_LIGNE-1)
562         {
563             fputc ( '_',out );
564             j++;
565         }
566         fputc ( '\n',out );
567         j=0;
568         i++;
569     }
570     fclose (out );
571     return fopen (s , "r+");
572 }
573 //Simple raccourci pour ouvrir un fichier
574 FILE* ouvre_fichier(int n)
575 {
576     char s []="generation0000.txt";
577     sprintf(s , "generation%d%d%d%d.txt", n/1000,(n%1000)/100, (n%100)/10, n%10);
578     return fopen (s , "r+");
579 }
580
581
582 int isprime(int n) // Fonction isprime() classique
583 {
584     if(n<2) return 0;
585     if(n==2) return 1;

```

```

586     if (n%2==0) return 0;
587     int i=3;
588     while( i*i<=n)
589     {
590         if (n%i==0) return 0;
591         i+=2;
592     }
593     return 1;
594 }
595
596
597 void binarise(int n, int *t, int p) // binarise un entier dans un tableau a 8
598 // cases
599 {
600     t [p]=n%2;
601     if(p<8) binarise(n/2,t,p+1);
602 }
603
604 // En cas de changement de format
605 void Convertisseur_longueur_ligne(int avant)
606 {
607     FILE *genbis=NULL, *gen=NULL;
608     genbis=genere_fichier(0);
609     gen=fopen("generation0001.txt", "r");
610     int i=0;
611     char s[avant];
612     while(i<200)
613     {
614         fseek(gen, avant*i,SEEK_SET);
615         fscanf(gen,"%s",s);
616         fseek(genbis,LONGUEUR_LIGNE*i,SEEK_SET);
617         fprintf(genbis,"%s",s);
618         i++;
619     }
620     fclose(gen);
621     fclose(genbis);
622 }
623
624 // 'F' est le symbole que j'ai pris pour 'Fin de fonction'
625 int calcul_taille(FILE* gen)
626 {
627     int taille=0;
628     while(fgetc(gen)!= 'F') taille++;
629     return taille;
630 }
631
632
633 // On se place en debut de ligne
634 // Cette fonction lis la fonction et la met en string.
635 void get_fonction(FILE* gen, char *fonction)

```

```

636 {
637     int i=0;
638     fonction[0]=fgetc(gen);
639     while(fonction[i]!='F')
640     {
641         i++;
642         fonction[i]=fgetc(gen);
643     }
644 }
645
646 // Contrairement à lecture, cette fonction prend un string en argument
647 int evaluer(char* fonction, int s, int t, int u, int v, int w, int x, int y,
648             int z)
649 {
650     int pile[LONGUEUR_LIGNE/2]; // Le /2 est pris pour diminuer un peu la
651     // taille. Peut être est-ce trop ...
652     int i0=0;
653     pile[0]=0;
654
655     while(fonction[i0]!='F')
656     {
657         switch(fonction[i0])
658         {
659             case 'N': // NON
660                 if(pile[pile[0]]) pile[pile[0]]=0;
661                 else pile[pile[0]]=1;
662                 break;
663             case 'E': // ET
664                 if(pile[pile[0]]&&pile[pile[0]-1]) pile[pile[0]-1]=1;
665                 else pile[pile[0]-1]=0;
666                 pile[0]--;
667                 break;
668             case 'O': // OU
669                 if(pile[pile[0]]||pile[pile[0]-1]) pile[pile[0]-1]=1;
670                 else pile[pile[0]-1]=0;
671                 pile[0]--;
672                 break;
673             // VARIABLES
674             case 's':
675                 pile[0]++;
676                 pile[pile[0]]=s;
677                 break;
678             case 't':
679                 pile[0]++;
680                 pile[pile[0]]=t;
681                 break;
682             case 'u':
683                 pile[0]++;
684                 pile[pile[0]]=u;
685                 break;
686             case 'v':

```

```

685         pile[0]++;
686         pile[pile[0]]=v;
687         break;
688     case 'w':
689         pile[0]++;
690         pile[pile[0]]=w;
691         break;
692     case 'x':
693         pile[0]++;
694         pile[pile[0]]=x;
695         break;
696     case 'y':
697         pile[0]++;
698         pile[pile[0]]=y;
699         break;
700     case 'z':
701         pile[0]++;
702         pile[pile[0]]=z;
703         break;
704     }
705     i0++;
706 }
707 return pile[1];
708 }

710

711 // Le calcul de score, chaque coeur faisant la moitie du boulot
712

713 void* thread_score1(void* arg) // en arg, on a le string de la fonction.
714 {
715     int score=0;
716     int i=0, j=0, t[8];
717     while(i<SCOREMAX/2)
718     {
719         binarise(i,t,0);
720         j=i;
721         if(prime[i]==evaluate(arg,t[7],t[6],t[5],t[4],t[3],t[2],t[1],t[0]))
722             score++;
723         i=j;
724         i++;
725     }
726     pthread_exit(&score);
727 }

728 void* thread_score2(void* arg) // en arg, on a le string de la fonction.
729 {
730     int score=0;
731     int i=SCOREMAX/2, j=0, t[8];
732     while(i<SCOREMAX)
733     {
734         binarise(i,t,0);

```

```

735     j=i ;
736     if(prime[i]==evaluate(arg,t[7],t[6],t[5],t[4],t[3],t[2],t[1],t[0]))
737         score++;
738     i=j;
739     i++;
740 }
741 pthread_exit(&score);
}

```

5.2 Programme auxiliaire

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define LONGUEUR_LIGNE 2000
5
6 /// Convertir convenablement ce qui sort du main en un joli petit arbre
7 /// Maxime Niankouri - 11.07.11
8
9
10
11 int noeud(FILE *output, char* s, int i);
12 void binarise(int n, int *t, int p);
13 void get_fonction(FILE* gen, char *fonction);
14 int evaluate(char* fonction, int s, int t, int u, int v, int w, int x, int y,
15             int z);
16 int isprime(int n);
17
18 int main()
19 {
20     FILE* input=NULL, *output=NULL;
21     output=fopen("tree.gv","w");
22     input=fopen("generation0001.txt", "r");
23
24     // On commence par recuperer la fonction a modeliser dans un string.
25     char s[LONGUEUR_LIGNE];
26     char c=fgetc(input);
27     int i=0;
28     while(c!= 'F')
29     {
30         s[i]=c;
31         i++;
32         c=fgetc(input);
33     }
34     i--; // i prend la position du noeud racine
35
36     fclose(input);
37
38     // On balance le header
39     fprintf(output,"graph{\n");

```

```

39
40 // Le corps
41 noeud(output,s,i);
42
43 // Et le finish
44 fprintf(output,"}\"); 
45
46 fclose(output);
47
48 // Avant de quitter, on en profite pour afficher a la console le
49 // nombre de composants.
50 int E=0, O=0, N=0, autre=0;
51 while(i>=0)
52 {
53     if(s[i]=='E') E++;
54     else if(s[i]=='O') O++;
55     else if(s[i]=='N') N++;
56     else autre++;
57     i--;
58 }
59 printf("La nombre de chaque composant est :\n\n\tE\t%d\n\tO\t%d\n\tN\t%d\n\tAutres\t%d\n\tTotal\t%d\n", E, O, N, autre, E+N+autre);
60
61 printf("-----\n\n");
62 // On passe ensuite au test d'efficacite.
63
64 i=0;
65 int t[8], j=2;
66 while(i<256)
67 {
68     binarise(i,t,0);
69     j=i;
70     if(isprime(i)!=evaluate(s,t[7],t[6],t[5],t[4],t[3],t[2],t[1],t[0]))
71         printf("On devrait avoir %d pour %d,\n\tas %d%d%d%d%d%d.\n",
72             isprime(j),j,t[7],t[6],t[5],t[4],t[3],t[2],t[1],t[0]);
73     i=j;
74     i++;
75 }
76
77 getchar(); // On attend pour quitter
78
79 return 0;
80 }
81
82
83 int noeud(FILE *output, char* s, int i) // s etant le string a lire et i la
84 // position dans le string.
{ 
```

```

85     int j=i ;
86
87     if( s [ i]== 'E' )
88     {
89         fprintf( output , "A%d-[ label=\\"E\\" , -color=red , -style=filled ];\n" , j )
90         ;
91
92         fprintf( output , "A%d---A%d;\n" , j , i-1 );
93         i=noeud( output , s , i-1 );
94
95         fprintf( output , "A%d---A%d;\n" , j , i-1 );
96         i=noeud( output , s , i-1 );
97     }
98     else if( s [ i]== 'O' )
99     {
100        fprintf( output , "A%d-[ label=\\"O\\" , -color=green , -style=filled ];\n"
101           n" , j );
102
103        fprintf( output , "A%d---A%d;\n" , j , i-1 );
104        i=noeud( output , s , i-1 );
105    }
106
107    else if( s [ i]== 'N' )
108    {
109        fprintf( output , "A%d-[ label=\\"N\\" , -color=blue , -style=filled ];\n" , j
110           );
111
112        fprintf( output , "A%d---A%d;\n" , j , i-1 );
113        i=noeud( output , s , i-1 );
114    }
115    else
116    {
117        fprintf( output , "A%d-[ label=\\"%c\\" ];\n" , j , s [ i ] );
118    }
119
120    return i ;
121
122
123
124
125 void get_fonction(FILE* gen , char *fonction) // doit a la base etre de taille
126   LONGUEUR_LIGNE
127 {
128     int i=0;
129     fonction[0]=fgetc(gen);
130     while(fonction[i]!='F')
131     {
132         i++;

```

```

132     fonction [ i ]=fgetc ( gen ) ;
133 }
134 }
135
136 int evaluer ( char* fonction , int s , int t , int u , int v , int w , int x , int y ,
137 int z )
138 {
139     int pile [ LONGUEUR_LIGNE / 2 ]; // Le /2 est pris pour diminuer un peu la
140     // taille .
141     int i0=0;
142     pile [ 0 ]=0;
143
144     while ( fonction [ i0 ]!= 'F' )
145     {
146         switch ( fonction [ i0 ] )
147         {
148             case 'N': // NON
149                 if ( pile [ pile [ 0 ] ] ) pile [ pile [ 0 ] ]=0;
150                 else pile [ pile [ 0 ] ]=1;
151                 break;
152             case 'E': // ET
153                 if ( pile [ pile [ 0 ] ] && pile [ pile [ 0 ] -1 ] ) pile [ pile [ 0 ] -1 ]=1;
154                 else pile [ pile [ 0 ] -1 ]=0;
155                 pile [ 0 ]--;
156                 break;
157             case 'O': // OU
158                 if ( pile [ pile [ 0 ] ] || pile [ pile [ 0 ] -1 ] ) pile [ pile [ 0 ] -1 ]=1;
159                 else pile [ pile [ 0 ] -1 ]=0;
160                 pile [ 0 ]--;
161                 break;
162             // VARIABLES
163             case 's':
164                 pile [ 0 ]++;
165                 pile [ pile [ 0 ] ]=s;
166                 break;
167             case 't':
168                 pile [ 0 ]++;
169                 pile [ pile [ 0 ] ]=t;
170                 break;
171             case 'u':
172                 pile [ 0 ]++;
173                 pile [ pile [ 0 ] ]=u;
174                 break;
175             case 'v':
176                 pile [ 0 ]++;
177                 pile [ pile [ 0 ] ]=v;
178                 break;
179             case 'w':
180                 pile [ 0 ]++;
181                 pile [ pile [ 0 ] ]=w;
182                 break;

```

```

181     case 'x':
182         pile[0]++;
183         pile[pile[0]]=x;
184         break;
185     case 'y':
186         pile[0]++;
187         pile[pile[0]]=y;
188         break;
189     case 'z':
190         pile[0]++;
191         pile[pile[0]]=z;
192         break;
193     }
194     i0++;
195 }
196 return pile[1];
197 }

198
199 int isprime(int n) // Check si un nombre est premier ou non.
200 {
201     if(n<2) return 0;
202     if(n==2) return 1;
203     if(n%2==0) return 0;
204     int i=3;
205     while(i*i<=n)
206     {
207         if(n%i==0) return 0;
208         i+=2;
209     }
210     return 1;
211 }
212
213 void binarise(int n, int *t, int p) // binarise un entier dans un tableau a 8
214 // cases;
215 {
216     t[p]=n%2;
217     if(p<8) binarise(n/2,t,p+1);
218 }
```

5.3 Circuit électronique

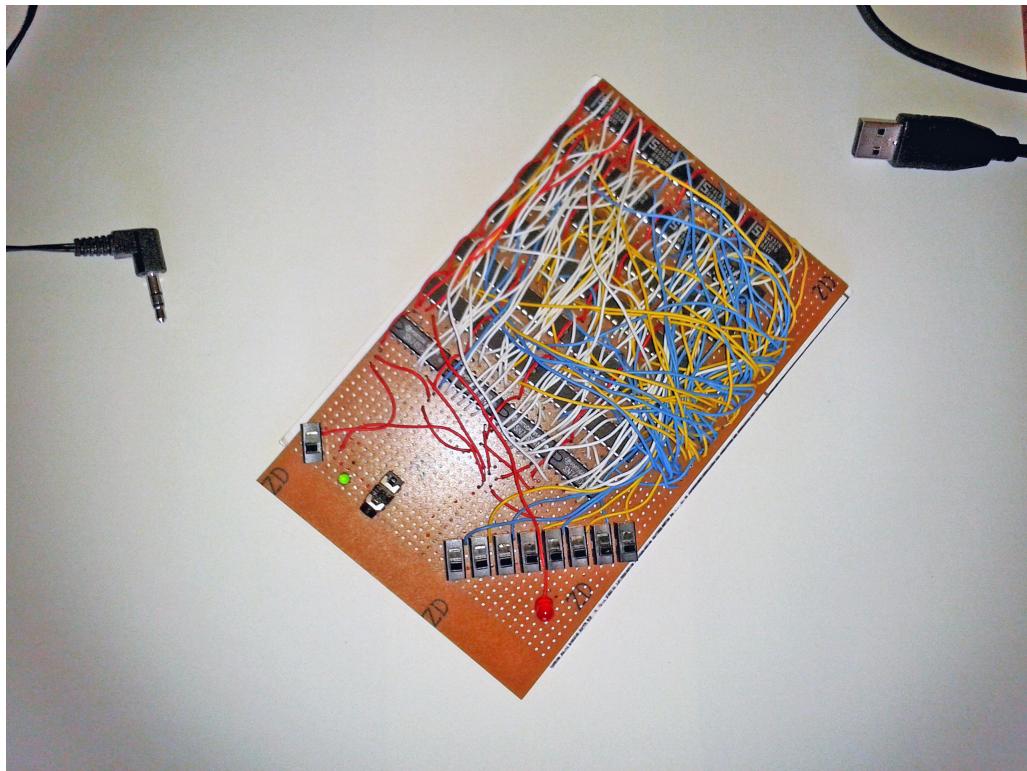


FIGURE 3 – Circuit électronique au repos, vu de dessus

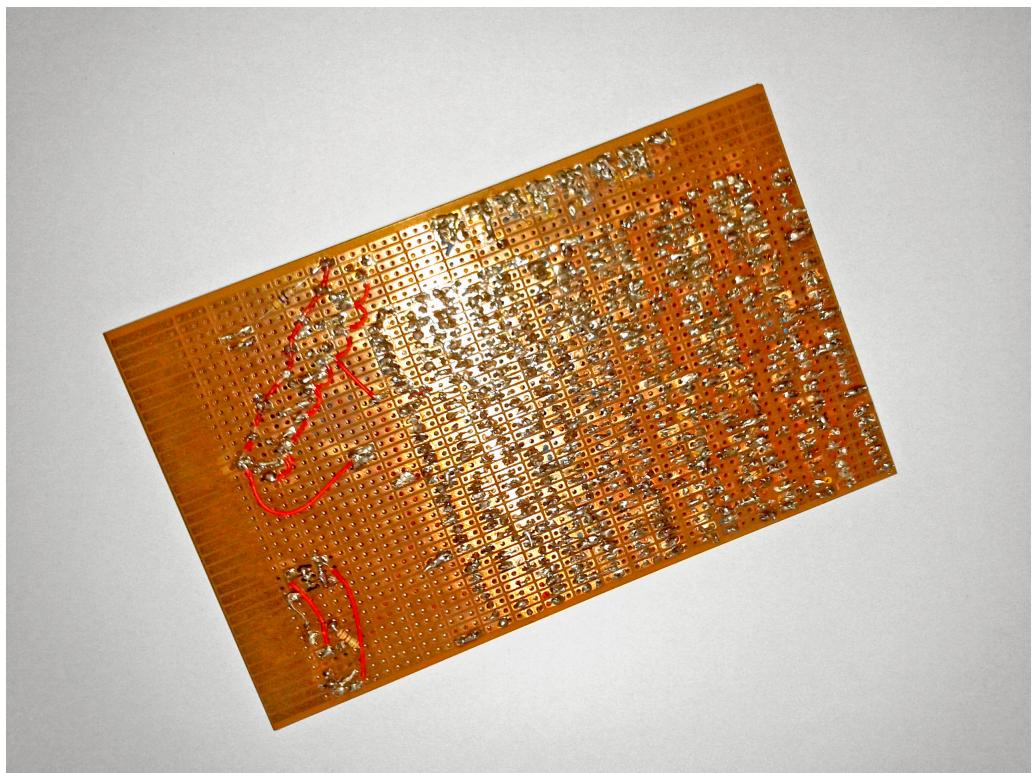


FIGURE 4 – Circuit électronique, vu de dessous

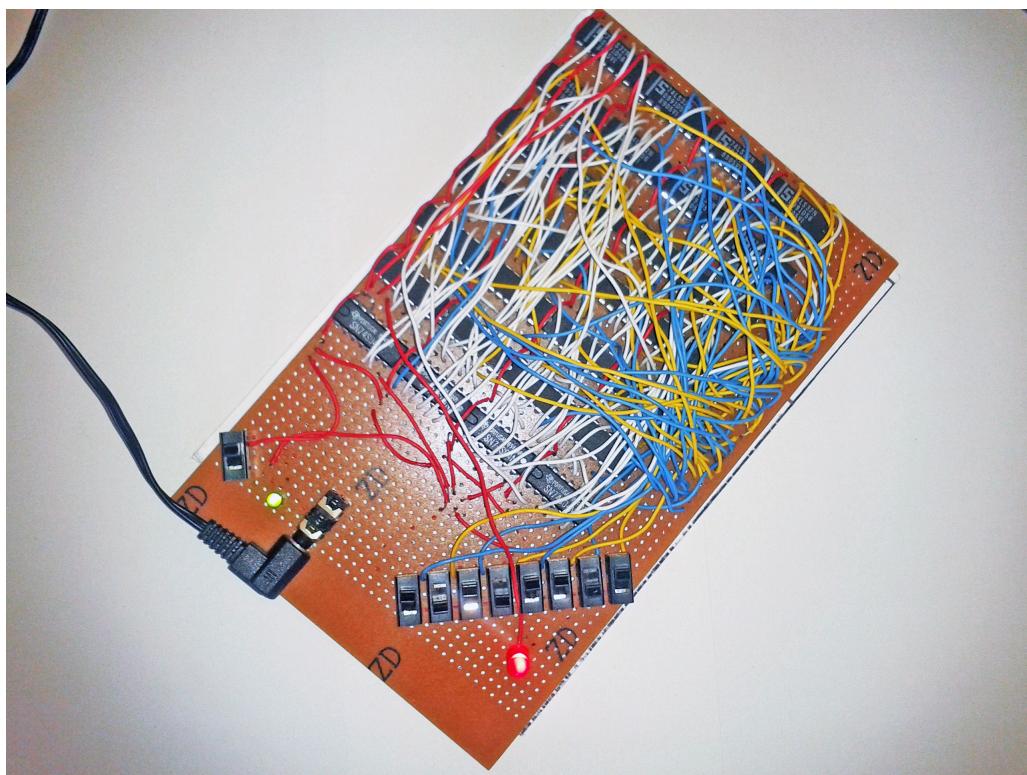


FIGURE 5 – 10101101 est premier