ПЛОВДИВСКИ УНИВЕРСИТЕТ "ПАИСИЙ ХИЛЕНДАРСКИ", БЪЛГАРИЯ
НАУЧНИ ТРУДОВЕ, ТОМ 35, КН. 3, 2007 – МАТЕМАТИКА
PLOVDIV UNIVERSITY "PAISSII HILENDARSKI", BULGARIA
SCIENTIFIC WORKS, VOL. 35, BOOK 3, 2007 – MATHEMATICS

ALGORITHMS FOR GENERATION OF CIRCUITS AND DRAFTS IN DISTRIBUTED E-TESTING CLUSTER (DeTC)

Asen Rahnev, Olga Rahneva

Abstract. This paper describes author's algorithms for automatic generation of circuits, which are used in the tools of the Distributed e-Testing Cluster (DeTC). These algorithms contribute to solve the problem with insufficient testing electronic questions in various areas.

The paper describes algorithms for generation of bit-masks, which define possible connections between the composing elements of circuits; for generation of circuits; for checking the connections between elements; for checking the equality of circuits.

There are also presented modified algorithms for generation of bit masks and circuits, when the composing elements can be connected by double, triple and more complex connections.

Key words: DeTC, web-based testing, automatic generation, circuitries

1. Introduction

This paper describes author's algorithms for automatic generation of circuits. They are applied in the Distributed e-Testing Cluster (DeTC), which significantly facilitates authors in creating new testing questions, in parameterization of testing questions and in creating classes of testing questions, which assess same areas of knowledge. Due to their volume, initialization routines are not included in the code of the algorithms.

This approach is experimented and is being successfully applied in preparation of tests for the education in Physics, Electronics, Electrical Engineering, Chemistry, etc.

DeTC [1–3] is being developed as a joint project of the ECE Department at the University of Limerik — Ireland, the Humboldt University in Germany, the Laboratory for Electronic Trade (ECL), the departments of Computer Technologies and Computer Systems at the University of Plovdiv, Bulgaria, and the department of Informatics and Statistics at the University of Food Technologies, Plovdiv, Bulgaria.

2. Supplementary algorithm for genarating bit masks

The purpose of this algorithm is to generate bit masks, which will determine for a selected element which its neighbor elements it will connect to. This algorithm is used by the algorithms which generate circuits.

In the realization the following variables are used:
bitscount – number of bits in the mask;
tmin – minimum number of raised bits;
tmax – maximum number of raised bits;
BM – working array of bits;
tc – number of raised bits.

Code of the algorithm:

```
void Generate()
{
1.  int tc = tmin;
2.  Init&retNumInitEl(BM, 0, tmin-1, false);
3.  while (true)
4.    if (GenNextBitMask(BM,bitscount,tc,tmin,tmax) == false)
        break;
5.  done
}

bool GenNextBitMask(Array BM, int bcount, tc, tmin, tmax)
{
1.   if (BM[0] == true)
2.     i = 1; while(BM[i]) i++;
3.     if (i >= bcount) return false;
4.     BM[i] = true; tc++;
5.     tc -= Init&retNumInitEl(BM, tmin-tc+i, i-1, false);
6.   else
```

```
6.   else
7.     if (tc < tmax)
8.       if (bcount > 0)
9.         M[0] = true; tc++;
10.      else
11.        return false;
12.    else
13.      for (i=1; !BM[i] && i<bcount; i++);
14.      j = i+1;
15.      while (BM[j]) j++;
16.      if (j >= bcount) return false;
17.      BM[j] = true; tc++;
18.      tc -= Init&retNumInitEl(BM, i, j-1, false);
19.      tc += Init&retNumInitEl(BM, 0, tmin-tc-1, true);
20.    fi
21.  fi
22.  return true;
}
```

Figure 1. Supplementary Algorithm for Generating Bit Masks

3. Algorithm for generating circuits

The purpose of this algorithm is to generate circuits according to predefined elements and rules for connecting them. The algorithm uses heuristic methods to improve performance by removing repeated combinations. The process is improved when elements, selected for connecting with element "level", are sorted by a certain criteria and thus defining priorities. For example, if on every level of recursion the algorithm calculates a coefficient for every element, which element "level" can be connected to, the elements can be sorted by this coefficient and the algorithm can select the best next candidates.

In the realization the following variables are used:

Global variables:
M – Matrix of admissible connections;
C – Counter of connections in which elements take place;
Emin – The minimum number of connections an element can take place;
Emax – The maximum number of connections an element can take place;
R – Matrix storing the temporary results.

Local variables:
level – level of recursion;
BM – The bit masks of the current level of recursion;
BMM – Provides pointers to map the consequent elements of the bit mask to their corresponding real elements;
tmax – The minimum number of elements, which element "level" must connection to;
tmin – The maximum number of elements, which element "level" can connection to;
bitscount – Number of bits in the bit mask;
U – An array storing the elements which need connecting, with the first index of the array being the number of connections the elements can take part from the current moment on, subtracted from the number of connections they already take part in, subtracted from the minimum number of connections needed;
Z – An array, which stores the elements of **U** with coefficient equal to zero.

Code of the algorithm:

```
void InitBitsCount(int bitscount, Array Emin, Array EMax,
    Array M, int N)
{
1.  for (i=level+1; i<N; i++)
2.    if (M[level][i] && EMax[i] > C[i])
3.      U[(N-level-1)-(Emin[i]-C[i])][i] = true;
4.      bitscount++;
5.    fi
}

void FindMostUrgentElements()
{
1.  if (U.count(0))
2.    map<int,bool>::iterator uj = U[0].begin();
3.    while (uj != U[0].end())
4.      Z[uj->first] = true; uj++;
5.    done
6.    U.erase(0);
7.  fi
}
```

```
void InitBitscount&BMP(int bitscount, int tmin, int tmax,
    Array BMP, Array Z)
{
1.   bitscount = 0;
2.   map<int,map<int,bool>>::iterator ui = U.begin();
3.   while (ui!=U.end())
4.     map<int,bool>::iterator uj = ui->second.begin();
5.     while (uj != ui->second.end())
6.       BMM[bitscount++] = uj->first;
7.       uj++;
8.     done
9.   ui++;
10.  done
11.  tmin = Emin[level]-C[level]-(int)Z.size();
12.  if (tmin < 0) tmin = 0;
13.  tmax = EMax[level]-C[level]-(int)Z.size();
}

void GenCombs(int level=0)
{
1.   if (level >= N)
2.     PrintResult(); return;
3.   fi
4.   if (level == 0) RC=0;
5.   InitBitsCount(bitscount, Emin, EMax, M, N);
6.   FindMostUrgentElements();
7.   if ( C[level]+(int)Z.size()-EMax[level] > 0 ||
8.        Emin[level]-C[level]-bitscount > 0 ) return;
9.   UpdateResultZ(true);
10.  InitBitscount&BMP(bitscount, tmin, tmax, BMP, Z);
11.  Init&retNumInitEl(BM, 0, tmin-1, true);
12.  int tc = tmin;
13.  while (true)
14.    ApplyBitMaskSec();
15.    GenCombs(level+1);
16.    UndoBitMaskSec();
17.    if (GenNextBitMask(BM,bitscount,tc,tmin,tmax)==false) break;
18.  done
19.  UpdateResultZ(false);
}
```

```
void UpdateResultZ(bool flag)
{
1.  if (flag == true) cval = 1; else cval = -1; fi
2.  mi = Z.begin();
3.  while (mi != Z.end())
4.    C[mi->first] += cval;
5.    R[level][mi->first] = flag;
6.    R[mi->first][level] = flag;
7.    mi++;
8.  done
}

void ApplyBitMaskSec()
{
1.  SetBitMask(true);
}

void UndoBitMaskSec()
{
1.  SetBitMask(false);
}

void SetBitMask(bool flag)
{
1.  if (flag == true) cval = 1; else cval = -1; fi
2.  mi = BM.begin();
3.  while (mi != BM.end())
4.    if (mi->second)
5.      C[BMM[mi->first]] += cval;
6.      R[level][BMM[mi->first]] = flag;
7.      R[BMM[mi->first]][level] = flag;
8.    fi
9.    mi++;
10. done
}
```

Figure 2. Algorithm for Generating Circuities

4. Algorithm for circuit connection check

Some of the generated circuits consist of separate blocks, which have no connections with one another and are completely separated from the circuit. The current algorithm filters out such circuits.

In the realization the following variables are used:

Global variables:
M – Matrix of admissible connections;
C – Counter of connections in which elements take place;
Emin – The minimum number of connections an element can take place;
Emax – The maximum number of connections an element can take place;
R – Matrix storing the temporary results.

Local variables:
level – level of recursion;
BM – The bit masks of the current level of recursion;
BMM – Provides pointers to map the consequent elements of the bit mask to their corresponding real elements;
tmax – The minimum number of elements, which element "level" must connection to;
tmin – The maximum number of elements, which element "level" can connection to;
bitscount – Number of bits in the bit mask;
U – An array storing the elements which need connecting, with the first index of the array being the number of connections the elements can take part from the current moment on, subtracted from the number of connections they already take part in, subtracted from the minimum number of connections needed;
Z – An array, which stores the elements of **U** with coefficient equal to zero.

Code of the algorithm:

```
void trace(int level=0)
{
1.  for (int i=0; i<N; i++)
2.    if (R[level][i] && !V[i])
3.      V[i] = true;
4.      trace(i);
5.    fi
6.  done
}
```

```
bool validate()
{
1.  V.clear();
2.  V[0] = true;
3.  trace();
4.  for (int i=0; i<N; i++)
5.    if (!V[i]) return false;
6.  return true;
}

Lines 1-3 of routine GenCombs are changes as follows:
1.  if (level >= N)
2.    if (validate())
3.      // Go through R and report the result
4.    return;
5.  fi
```

Figure 3. Algorithm for Circuity Connection Check

5. Algorithm for identification of linear structures in circuits

The purpose of the algorithm is to identify whether two neighbor elements of the circuit n1 and n2 create a linear structure. The algorithm checks the number of connections of the two elements, excluding the connection between them. If there is only one connection, then the algorithm assumes that there exists a liner structure.

In the realization the following variables are used:

Global variables:
N – Number of elements;
M – Matrix of connections, representing the circuit;
V – A supplementary array for marking processed elements;
C – Array with number of connections each element takes part in.

Local variables:
L – An array with the linear structures of the circuit;
N – The current element;
n1 and **n2** – Elements, for which the algorithm have to identify a linear structure.

Code of the algorithm:

```
void LNext(int n, Array V, Array M)
{
1.  for (i=0; i<N; i++)
2.    if ((M[n][i] != 0) && (!V[i]))
3.      V[i] = true;
4.      LNext(i, V, M);
5.    fi
}

bool Linear(int n1, int n2, Array C, Array M)
{
1.  if ((M[n1][n2]!=0)&&(C[n1]<=2)&&(C[n2]<=2)) return true;
2.  else
3.    V[n1] = true;
4.    for (int i=0; i<N; i++)
5.      if ((M[n1][i] != 0) && (!V[i]) && (i!=n2))
6.        V[i] = true;
7.        LNext(i, V, M);
8.      fi
9.  fi
10. if (V[n2]) return false;
11. else return true;
}
```

Figure 4. Algorithm for Identification of Linear Structures in Circuits

6. Algorithm for checking equality of circuits

Two circuits are considered equal if they contain equal linear structures and connections between them. This algorithm uses the algorithm for identification of linear structures. It filters out equal circuits.

Routine **FindLinear** identifies all linear structures of the circuits. Those are compared and if they are equal, routine **BuildCompareMatr** is used to create arrays of connections between the linear structures of both circuits. If those arrays are equal, the circuits are equal as well.

In the realization the following variables are used:

Global variables:
N – Number of elements.

Local variables:
M1 – Matrix of connections, representing the first circuit;
M2 – Matrix of connections, representing the second circuit;
L1 – Array with the linear structures of first circuit;
L2 – Array with the linear structures of second circuit;
C1 – Arrays with connections between the linear structures of first circuit;
C2 – Array with connections between the linear structures of second circuit;
C – Array with connections between the linear structures in M;
M – Matrix of connections;
L – An array with linear structures in M.

Code of the algorithm:

```
void FLNext(int n, Array M, Array VL, Array L, Array C, int Row)
{
1.  for (int i=0; i<N; i++)
2.    if ((M[n][i] != 0))
3.      if (Linear(n, i, C, M) && (!VL[i]))
4.        L[Row][i] = true;
5.        VL[i] = true;
6.        FLNext(i, M, VL, L, C, Row);
7.      fi
}
void FindLinear(Array M, Array L)
{
1.  for (i=0; i<N; i++)
2.    for (j=0; j<N; j++)
3.      if (M[i][j]) C[i]++;
4.  Row = 0;
5.  while (true)
6.    for (i=0; i<N; i++)
7.      if (!VL[i]) break;
8.      if (i == N) break;
9.      L[Row][i] = true;
10.     VL[i] = true;
11.     FLNext(i, M, VL, L, C, Row);
12.   done
13.   Row++;
14. done
}
```

```
void BuildCompareMatr(Array C, Array M, Array L)
{
1.  for (int i=0; i<N; i++)
2.    for (int j=0; j<N; j++)
3.      if (M[i][j] != 0)
4.        C[FindKey(i,L)][FindKey(j,L)] = true;
}

bool SimilarityCheck(Array M1, Array M2)
{
1.  FindLinear(M1, L1);
2.  FindLinear(M2, L2);
3.  if (!Compare(L1, L2)) return false;
4.  BuildCompareMatr(C1, M1, L1);
5.  BuildCompareMatr(C2, M2, L2);
6.  if (!Compare(C1, C2)) return false;
7.  return true;
}
```

Figure 5. Algorithm for Checking Equality of Circuits

7. Modified algorithms

The modified versions of the algorithms overcome a limitation in the prior versions: the modified versions can work properly with elements, which allow for different types of connections – double, triple, etc. Such complex connection types exist in various areas, for example – Chemistry, where connections between chemical elements can be single, double, triple, etc, corresponding to the valence of the elements.

In order to model the connections, the matrix of admissible connections has to be updated into a *modified matrix of admissible connections*. The diagonal of the modified matrix contains zeros and ones, which specify whether the corresponding element is static or dynamic. Data above the diagonal contains integer numbers, each number defining the type of connection – single, double, etc. The connections between elements are complex, because some of them can connect to each other in more than one way. For example, chemical element C connects to chemical compound CH by a single, double or triple connection.

8. Modified algorithm for generating masks

The algorithm generates masks of integer numbers, which specify for each element the neighbor ones which it can connect to, and the type of connection (1 – single, 2 – double, 3 – triple, etc).

In the realization the following variables are used:

N – Number of elements in the mask;
Sum – The sum we would like to achieve;
BM – A supplementary array;
C – Sum achieved at the current moment.

```
void Generate()
{
1.  Init&retNumInitEl(BM, 1, n-1, 0);
2.  BM[0] = sum;
3.  c = sum;
4.  while (true)
5.    if (n == 0) break;
6.    PrintResult();
7.    if (GenNextIntMask(BM, n, sum, c) == false) break;
8.  done
}

bool GenNextIntMask(array BM, int bcount, abcount, tmax)
{
1.  int iStart;
2.  if (BM[bcount-1] == tmax) return false;
3.  iStart = bcount-2;
4.  while (BM[iStart] == 0) iStart--;
5.  BM[iStart]--;
6.  abcount--;
7.  abcount -= BM[bcount-1];
8.  BM[bcount-1] = 0;
9.  BM[iStart+1] = tmax - abcount;
10. abcount = tmax;
11. return true;
}
```

Figure 6. Modified Algorithm for Generating Masks

9. Modified algorithm for generating circuits

This version of the algorithms checks on every step whether the end element of the circuit is reached. If so, the circuit is validated.

In the realization the following variables are used:

N – Number of elements;
M – Matrix of admissible connections;
CC – Counter of connections, in which elements take part in;
Min – Stores the minimum number of connections an element has to take part in;
Max – Stores the maximum number of connections an element has to take part in;
R – Matrix, which stored the temporary results;
V – Supplementary array, used in validity check.

Local variables:
level – level of recursion;
BM – The bit masks of the current level of recursion;
BMM – Provides pointers to map the consequent elements of the bit mask to their corresponding real elements;
tmax – The minimum number of elements, which element "level" must connection to;
tmin – The maximum number of elements, which element "level" can connection to;
bcount – Number of bits in the bit mask;
abcount – The current sum of values of elements in mask.

```
void trace(int el=0)
{
1.  for (int i=0; i<N; i++)
2.    if ((R[el][i]) && (!V[i]))
3.      V[i] = true;
4.      trace(i);
5.    fi
}

bool validate()
{
1.  V.clear();
2.  V[0] = true;
3.  trace();
```

```
4.  for (int i=0; i<N; i++)
5.    if (Min[i]>CC[i] || Max[i]<CC[i] || !V[i]) return false;
6.    for (int j=i+1; j<N; j++)
7.      if (R[i][j] && (M[i][j]==0)) return false;
8.    done
9.  done
10. return true;
}

bool BMSatisfiesRestrictions(Array BM, Array BMP, Array M,
    Array CC, int bcount, int level)
{
1.  for (i=0; i<bcount; i++)
2.   if (Max[BMP[i]]-CC[BMP[i]] < BM[i] || BM[i]> M[level][BMP[i]])
3.     return false;
4.  return true;
}

void Generate(int level=0)
{
1.  if (level > N && validate()) PrintResult(); return; fi
2.  if (CC[level] > Max[level]) return;
3.  EvaluateMinMax&Init(tmin, tmax, Min, Max, CC, BMP, bcount);
4.  while (tmin <= tmax)
5.    Init&retNumInitEl(BM, 1, N-1, 0);
6.    BM[0] = tmax;
7.    abcount = tmax;
8.    while (true)
9.      ApplyMaskToResult(R, BM, BMP, CC, level);
10.     if (BMSatisfiesRestrictions(BM,BMP, M, CC, bcount,level))
11.       Generate(level+1);
12.     if (bcount == 0) break;
13.     RemoveMaskFromResult(R, BM, BMP, CC, level);
14.     if (GenNextIntMask(BM,bcount,abcount,tmax) == false) break;
15.   done
16.   tmax--;
17. done
}
```

Figure 7. Modified Algorithm for Generating Circuits

Acknoledgement

This research has been partially supported by the Bulgarian NSF under Contract No VU-MI 107/2005.

References

[1] Rahneva O., . Rahnev, N. Pavlov, *Functional Workflow and Electronic Services In a Distributed Electronic Testing Cluster – DeTC*, Proceedings 2nd International Workshop on eServices and eLearning, Otto-von-Guericke Universitaet Magdeburd, 2004, pp 147–157.

[2] Rahnev A., N. Pavlov, O. Rahneva, *Architecture & Design of Distributed Electronic Testing Cluster (DeTC) based on Microsoft .NET Framework* – IMAPS CS International Conference 2005, September 15–16, 2005, Brno, Czech Republic, pp 417–422.

[3] Rahneva O., A. Rahnev, N. Pavlov, N. Valchanov, *Authoring and Automatic Generation of Circuitries and Drafts in Distributed e-Testing Cluster (DeTC)*, ELECTRONICS'05, Sozopol, 21–23 Sept. 2005 (to appear).

Asen Rahnev
Faculty of Mathematics and Informatics
University of Plovdiv
236 Bulgaria Blvd.,
4003 Plovdiv, BULGARIA
e-mail: assen@pu.acad.bg

Olga Rahneva
University of Food Technologies
Dept. of Informatics & Statistics
26 Maritsa Str.
4000 Plovdiv, Bulgaria

Received 30 October 2005

АЛГОРИТМИ ЗА ГЕНЕРИРАНЕ НА СХЕМИ И ЧЕРТЕЖИ В РАЗПРЕДЕЛЕН КЛЪСТЕР ЗА ЕЛЕКТРОННО ТЕСТВАНЕ (DeTC)

Асен Рахнев, Олга Рахнева

Резюме. В тази работа се описват авторски алгоритми за автоматично генериране на схеми, които се използват от инструментите в разпределения клъстер за електронно тестване — DeTC (Distributed eTesting Cluster). Тези алгоритми спомагат за преодоляване на проблема, свързан с недостатъчното количество тестови електронни въпроси в редица области.

Разгледани са алгоритми за генериране на битови маски за определяне на възможности за свързване на съставящите елементи на схемите, за генериране на схеми, за проверка на връзките между елементите, за проверка за еднаквост между различни схеми.

Представени са модифицирани алгоритми за генериране на битови маски и схеми, в които съставящите елементи могат да се свързват помежду си чрез двойни, тройни и по-сложни връзки.