Generation of Graphs for Invariants of the Vector Fields to the Total Affine Transformation

1 Introduction

For the generation of the invariants of the vector fields, we need special graphs with a few types of edges. Such graphs are called multi-layer graphs. Here, the edges of the first type correspond to coordinates (vector positions) and the edges of the second type correspond to the vector values. The edges of the third type have two different ends, one is connected with the coordinates and the other with the values. So, these edges are oriented. The main idea is to begin with a graph that have the node labels as low as possible and then successively increase the node labels until the last possible graph has been reached.

1.1 Graphs for Independent Total Affine Transformation

Here we need only first two types of edges, therefore we generate bi-layer graphs. Each node must be connected with at least two edges of the first type and with just one edge of the second type; self-loops are not allowed. The graph is represented by a list of edges in a matrix $m \times 2$, where each column is one edge. The elements of the matrix are labels of the nodes.

First, we generate the edges of the first type. The input is the number of edges m, we successively increase it from 3 to the value implied by the power of our computer. The current maximal value is 9. The first graph is

$$\begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ 2 & 2 & \dots & 2 & 2 \end{pmatrix}. \tag{1}$$

In C language, the graph is represented by the array j of length $2m$, where $j[2*k]$ is the first node of the kth edge and $j[2*k+1]$ is the second node of the kth edge. The first graph is made

```
for(k=0;k<m;k++)   //the edges are labeled from zero
{
  j[2*k]=0;      //the nodes are also labeled from zero
  j[2*k+1]=1;
}
```

The "last" graph, on which the algorithm should stop, is

$$\begin{pmatrix} 1 & 1 & 3 & 4 & \dots & m-2 & m-1 & m-1 \\ 2 & 3 & 4 & 5 & \dots & m-1 & m & m \end{pmatrix}. \tag{2}$$

It is generated to the array jm of the same length as j

```
for(k=0;k<m;k++)
{
  jm[2*k]=k;
  jm[2*k+1]=k+1;
}
jm[2]=0;              //irregular values are assigned separately
jm[2*(m-1)]=m-2;
jm[2*(m-1)+1]=m-1;
```

The algorithm for generation of the next graph is then

```
//algorithm 1: A not equals B, edges E1
  l1=m-1;     //l1 is the current node in the first row
  while(l1>0)
  {
      l2=m-1;     //the search starts from the second node of the last edge
      while (l2>0 && j[2*l2+1]>=jm[2*l2+1])
        l2--;
      if(l2>0)    //a node that can be increased was found
      {
        j[2*l2+1]++;  //increasing
//the nodes behind the found one that would be lower, are increased, too
        for(k=l2+1;k<m;k++)
          j[2*k+1]=max(j[2*k]+1,j[2*l2+1]);
      }
      else    //no node was found in the second row
      {
        l1=m-1;    //search from the first node of the last edge
        while (l1>0 && j[2*l1]>=jm[2*l1])
          l1--;
        if(l1>0)    //a node that can be increased was found
        {
          j[2*l1]++;  //increasing
          for(k=l1+1;k<m;k++)
            j[2*k]=j[2*l1]; //the nodes behind are the same
          //the nodes in the second row will be bigger by one
          for(k=0;k<m;k++)
            j[2*k+1]=j[2*k]+1;
        }
      }
    }   //when l1==0, no other graph can be generated and it stops
```

To generate the second layer, we proceed analogically with some modifications. The first graph is now

$$\begin{pmatrix} 1 & 3 & \dots & n-3 & n-1 \\ 2 & 4 & \dots & n-2 & n \end{pmatrix}. \tag{3}$$

The number of nodes n must be even and the representation matrix has $n/2$ columns. The criterion, if a matrix element can be increased, is not its comparison with the final graph, but the test, if there is a non-used node. The inner loop of the algorithm must be modified, too. The second layer of the graph is stored in the array g of the length n. The list of nodes in an array pg of the length n has ones for the "empty" nodes and zeros for the nodes with an assigned edge of the second type.

```
//algorithm 2: A not equals B, edges E2
  k=n-2; //search from the last but one edge
  flag=0;
  while(k>=0 && flag==0)
  {
    memset(pg,1,n); //array pg is filled by ones
    for(k1=0;k1<2*k+1;k1++)
      pg[g[k1]]=0; //the nodes before k-th edge are occupied
    for(k1=g[2*k+1]+1;k1<2*n && flag==0;k1++)
      if(pg[k1])    //free node is found
      {
        g[2*k+1]=k1;     //it is used for the new edge
```

```
          pg[k1]=0;
          flag=1;
        }
      if(flag)
      {
        k2=2*k+2;
        for(k1=0;k1<2*n && k2<2*n;k1++)
          if(pg[k1])  //free nodes must be occupied by the remaining edges
          {
            g[k2]=k1;
            pg[k1]=0;
            k2++;
          }
      }
      k--;
    }
//if flag remains zero, no edge can be moved and no new second layer is
    possible
```

1.2 Graphs for Special Total Affine Transformation

We need tri-layer graph for this geometric transformation. The generation of three different layers would be too complicated, therefore we first generate all edges together and we assign some of them to the second or third type. The main difference from the first algorithm is the possibility of self-loops here, therefore the algorithm must be modified. The first graph is now

$$\begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & \dots & 1 & 1 \end{pmatrix}. \tag{4}$$

and the last one is

$$\begin{pmatrix} m & m & \dots & m & m \\ m & m & \dots & m & m \end{pmatrix}. \tag{5}$$

```
//algorithm 3: A equals B, edges E1 + E2 + E3
  l1=m-1;     //l1 is the current node in the first row
  while(l1>=0)
  {
    l2=m-1;     //the search starts from the second node of the last edge
    while (l2>=0 && j[2*l2+1]>=jm[2*l2+1])
      l2--;
    if(l2>=0)    //a node that can be increased was found
    {
      j[2*l2+1]++;  //increasing
      for(k=l2+1;k<m;k++)
        j[2*k+1]=max(j[2*k],j[2*l2+1]);
    }
    else    //no node was found in the second row
    {
      l1=m-1;    //search from the first node of the last edge
      while (l1>=0 && j[2*l1]>=jm[2*l1])
        l1--;
      if(l1>=0)    //an increasable node was found, l1==0 is sufficient here
      {
        j[2*l1]++;  //increasing
```

```
        for(k=l1+1;k<m;k++)
          j[2*k]=j[2*l1]; //the nodes behind are the same
          //the nodes in the second row will be also the same
        for(k=0;k<m;k++)
          j[2*k+1]=j[2*k];
      }
    }
  }   //when l1==0, no other graph can be generated and it stops
```

Now, we must assign the type to each edge in each graph. We use an array jf for it. It is the list of nodes, where each node has the following data: the number of connected edges, the label of the edge of the second or third type (it must be just one) and the list of nodes connected by an edge with the current node. The array jf has size $n(2m+2)$, i.e. $2m+2$ elements for each node. An ith edge of the kth node finishes at the node $jf[k*(2*m+2)+2+i]$. The array is created by transcription from the array j

```
  for (k = 0; k < m; k++)
  {
    phu = jf[j[2 * k] * (2 * m + 2)];    //the current number of edges
//at the first node of the k-th edge
    jf[j[2 * k] * (2 * m + 2) + phu + 2] = j[2 * k + 1] + 1;
    jf[j[2 * k] * (2 * m + 2)]++;
    phu = jf[j[2 * k + 1] * (2 * m + 2)];    //the current number of edges
//at the second node of the k-th edge
    jf[j[2 * k + 1] * (2 * m + 2) + phu + 2] = j[2 * k] + 1;
    jf[j[2 * k + 1] * (2 * m + 2)]++;
//the first connection node-edge of each node is assigned
//to the second type
    for (k = 0; k < n; k++)
    {
      jf[k*(2 * m + 2) + 1] = 1;
    }
  }
```

Now, we need to generate next assignment from the current one

```
//algorithm 4: A equals B, assign edges E2 + E3
  flag = 0;
  k = n - 1;    //n is the number of nodes, so, k is now the label of
//the last node
  while (k >= 0 && flag==0)
  {
    adr = jf[k * (2 * m + 2) + 1];  //address of the second-type connection
//node-edge at the k-th node
    k1 = jf[k * (2 * m + 2) + 1 + adr];
//If two nodes are connected by more than two edges, then only first
//two edges are relevant
    if (adr > 1 && jf[k * (2 * m + 2) + 1 + adr] == jf[k * (2 * m + 2) +
    adr] || k1<k)
    {
      while (jf[k * (2 * m + 2) + 1 + adr] == jf[k * (2 * m + 2) + adr] &&
    adr<jf[k*(2 * m + 2)])
        adr++;
    }
    if (adr >= jf[k*(2 * m + 2)])    //No edge at the k-th node can be
//assigned, test next node.
    {
```

```
20        jf[k*(2 * m + 2) + 1] = 1;
21        k--;
22      }
23      else  //Assign the next edge at the k-th node.
24      {
25        jf[k*(2 * m + 2) + 1]++;
26        flag = 1;
27      }
28    }
29  //If flag remains zero, there is no other node, we have reached
30  //all the graphs.
```

If two nodes are connected by more than two edges, there are only five different possibilities, how to assign the connection of the second type:

(a) all edges are of the first type,

(b) one edge of the second type, rest first type,

(c) one edge of the third type with orientation from the first to the second node, rest first type,

(d) one edge of the third type with orientation from the second to the first node, rest first type,

(e) two edges of the third type with opposite orientations, rest first type.

See also Fig. 1. All other possibilities are just permutations and it is useless to test them separately. The algorithm tests the case (d) twice, it is trade-off between simplicity and speed.

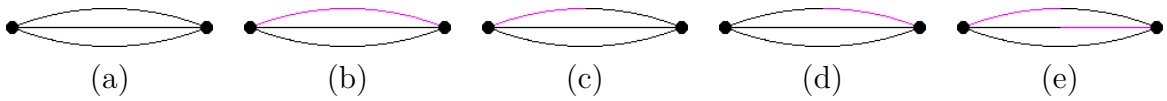

Figure 1: The only five relevant possibilities of the assignment of the second-type connection node-edge to three edges connecting the same two nodes. All other possibilities are just permutations.