Technical Report CSTN-013

Enumerating Circuits and Loops in Graphs with Self-Arcs and Multiple-Arcs

K.A. Hawick and H.A. James*
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
k.a.hawick@massey.ac.nz; heath.james@sapac.edu.au
Tel: +64 9 414 0800 Fax: +64 9 441 8181

Abstract

The problems of detecting and enumerating circuits in graphs and networks are still of fundamental importance. We extend the circuit enumeration algorithm of Johnson for graphs with directed-arcs, multiple-arcs and self-arcs and present a memory efficient and high-performance implementation in the D programming language. We also discuss other circuit applications including how the code could be adapted as a cycle detection algorithm.

Keywords: graph; circuit; enumeration; algorithm.

1 Introduction

Recent applications involving complex network analysis [1–3] and studies of workflow task graphs have rekindled interest in the fundamental problems of detecting, counting and enumerating circuits in graphs.

A circuit or loop as we discuss here is defined as a path in which the first and last vertices are identical. A path is said to be elementary if no vertex but the first and last appears twice. We are only interested in unique circuits which are said to be distinct if they are not merely cyclic permutations of one another. In a growing number of application we are e not merely interested in counting the circuits but also need to enumerate them to evaluate other properties such as their length distribution.

Various algorithms have been formulated to count the circuits in a graph but these either use infeasible amounts of memory or are time exponential [4, 5] with a time bound of $O(N.e(c+1))$, where N is the number of nodes, e the number of edges and c the number of circuits.

In practical applications involving complex networks or work-flow task graphs, the graph or network may in fact dynamically change and can fragment into a number of disconnected clusters. Clusters may also merge and so the number of clusters N_C can also vary. A useful algorithm needs to be used in tandem with a suitable data structure and a cluster component tracking algorithm. Many applications graphs are either generated by simulation processes or arise e with both multiple arcs and self arcs. A circuit analysis algorithm and implementation needs to be able to retain these independent arcs and interpret them appropriately.

We develop a variation of Johnson's algorithm [6] implemented in the D programming language [7] and which caters for directed graphs and which does not need to treat each of the possible $N_c > 1$ components separately.

By treating each arc as a track-able entity, regardless of whether it is a self-arc, a multiple-arc or an ordinary arc, we have had to make modifications to the assumptions underpinning Johnson's algorithm.

For graphs of N vertices, e edges, c circuits and 1 fully connected component, Johnson's algorithm is time bounded in time by $O\left((N+e)(c+1)\right)$ and space bounded by $O(N+e)$. This is still a highly expensive process since the number of circuits c itself grows very rapidly with (N, e).

*Present Address: South Australian Partnership for Advanced Computing (SAPAC), Adelaide, South Australia

In this article we describe the algorithm in section 2 and our implementation in section 3. We present some results and some worked examples in section 4. We report on some performance timing values and suggest some areas for future applications in section 5.

2 Data Structures and Algorithm

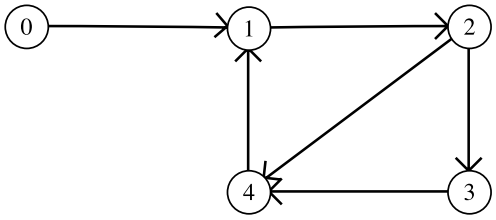

Figure 1: Test graph of 5 nodes and 6 arcs in a single component with 2 circuits (1-2-4-1 and 1-2-3-4-1)

Johnson's original algorithm was given in a Pascal-like pseudo code and used a $1..N$ integer labeling scheme. That work used an adjacency matrix structure that was incapable to supporting self-arcs and multiple arcs. We employ a set of arcs lists, where each to-arc is specified precisely by its source and destination node indices.

Figure 1 shows a small test graph of 5 nodes and 6 arcs, connected so as to yield just two circuits (1-2-4-1 and 1-2-3-4-1). Our storage structure for the adjacency list $Ak[i][j]$, where the i'th vertex has $m = Ak[i][0]$ arcs originating from it, with their destination vertices stored in $Ak[i][1..m]$. The D list structures can be manipulated as dynamical arrays and there size adjusted even for applications that may have multiple- and self-arcs and hence $m \geq N$.

The algorithm proceeds as Johnson's from a starting vertex from which it is assumed nodes that are involved in some circuit are at least connected. The algorithm maintains a stack of nodes from which it has tried a recursive tree search for circuits. The procedure circuit is called recursively to search a circuit and the procedure unblock which manages the elementary paths to avoid duplicating circuits. The algorithm therefore exactly enumerates the elementary circuits of the graph according to the order of the vertices.

The termination conditions for the recursive calls to the circuit function are therefor e important. Johnson's algorithm was expressed for a fully connected and fully reachable graph - effectively all arcs were two-way edges and it choosing which vertex to start from would not affect the total number of circuits found.

In our graphs, which can be fragmented but also can contain islands of unreachability even in a single component graph, the starting vertex does matter. Our algorithm obtains all elementary circuits involving nodes that are reachable from the starting 0'th vertex. An elementary circuit having eh usual definition of a circuit that contains no duplicate arcs. We can successfully count circuits that would fail in Johnson's algorithm due to multiple-arc effects. Self arcs are less trouble since e they will give only give rise to unit length elementary circuits.

3 Implementation Issues

In this section we list and comment on some D code fragments that implement our algorithm. We include e some auxiliary data structures and cod e for tracking: the vertex popularity – which is a frequency count of how many circuits a vertex appears in; the length and composition of the longest circuit found so far; and a frequency count of circuit lengths found.

The D code fragment 2 lists the global variables used in the algorithmic code fragments which follow:

```
int nVertices = 0;          // number of vertices
int start = 0;              // starting vertex index
int[][] Ak;
// integer array size n of lists
                            //
ie the arcs from the vertex
int[][] B;
// integer array size n of lists
bool[] blocked;
// logical array indexed by vertex
ulong nCircuits = 0;
// total number of circuits found;
ulong []lengthHistogram; // histogram of circuit lengths
ulong[][] vertexPopularity;
    // adjacency table of occurrences of
    //   vertices in circuits of each length

int[] longestCircuit;
// the (first) longest circuit found
int lenLongest = 0;         // its length

bool enumeration = false;// explicitly enumerate circuits
```

Figure 2: Global Variables

The short code fragment 3 shows the use of leading row elements in Ak to specify vertex out lists:

```
int countAkArcs(){   // return number of Arcs in graph
  int nArcs = 0;
  for(int i=0;i<nVertices;i++){
    nArcs += Ak[i][0];   // zero'th element gives nArcs for i
  }
  return nArcs;
}
```

Figure 3: Counting Arcs

The `unblock` routine 4 shows how vertices are recursively blocked and unblocked from elementary paths to avoid double counting.

```
void unblock( int u ){
  blocked[u] = false;
  for(int wPos = 1; wPos <= B[u][0]; wPos++){
                          // for each w in B[u]
    int w = B[u][wPos];
    wPos -= removeFromList(B[u], w);
    if( blocked[w] )
      unblock(w);
  }
}
```

Figure 4: recursive unblock

The `circuit` routine 5 is recursively called to chase down a circuit We employ some control flags to determine whether we wish to merely count circuits or to enumerate them and print them out. We also track the longest circuit found and the length frequencies and number of occurrences of particular vertices in circuits.

The code shown in figure 6 is used to initialise the global variables:

We make use of some fairly simple stack and list procedures. The main issue for performance concerns lists and iteration over lists that change length mid iteration. We list these for completeness, since they have some performance optimisations.

The key routines for managing our **stack** - based on a D dynamic array are shown in figure 7. This explicit integer array code was found considerably faster than using a template library class for a generalised stack.

Similar code is used to manage a **list** of integers as shown in figure 8.

There are various choices concerning bit precisions of various quantities. they can be `int`, `uint` or `ulong`. Generally in our applications there may be more circuits than a 32-bit `uint` can hold, but anything held in an array will typically only need an `int`. The code will typically hit factorial time limitations before it runs out of word/address space.

```
int[] stack = null; // stack of integers
static int stackTop = 0;
// the number of elements on the stack
                                    //
also the index "to put the next one"

// initialise  the stack to some size max
void stackInit(int max){
  stack.length = max;
  assert( stack != null );
  stackTop = 0;
}

// push an int onto the stack, extending if necessary
void stackPush( int val ){
  if( stackTop >= stack.length )
    stack.length = stack.length + 1;
  stack[stackTop++] = val;
}

int stackSize(){
  return stackTop;
}

int stackPop(){   // pop an int off the stack
  assert( stackTop > 0 );
  return stack[--stackTop];
}

void stackClear(){    // clear the stack
  stackTop = 0;
}
```

Figure 7: Stack management code

4 Examples Graphs with Circuits

Figure 9 shows a network with $N = 16, K = 2$. This graph has been generated from complex networks study using Kauffman NK networks [1]. The construction algorithm has allowed self-arcs – in other words the inputs for each node have been chosen according to a flat uniform distribution so they can connect to themselves.

The 22 circuits in the graph of figure 9 are shown in figure 10. There are repeated circuits due to the multiple arcs connecting nodes 12 and 11. Johnson's original algorithm would therefore give an incorrectly low number of elementary circuits for this graph.

Note that in figure 10 the circuits are found (and enumerated) in a vertex order. Starting with vertex 0, the vertices are searched in a stack/unblock order giving rise to the exact list shown.

A number of test graphs can be investigated. A interesting base case, from which modifications such as small-world connections or other rewirings can be studied is the simple square mesh. Table 1 shows some circuit measurements for periodic square mesh graphs. This emphasises how rapidly the number of circuits grows in well-connected graphs.

```
bool circuit( int v ){  // based on Johnson's logical procedure CIRCUIT
  bool f = false;
  stackPush(v);
  blocked[v] = true;

  for(int wPos = 1; wPos <= Ak[v][0]; wPos++){  // for each w in list Ak[v]:
    int w = Ak[v][wPos];
    if( w < start ) continue;  // ignore relevant parts of Ak

    if( w == start ){   // we have a circuit,
      if( enumeration ){
        stackPrint3d(); // print out the stack to record the circuit
        fwritef(stdout, "\%3d\n", start);   // and "start" which completes the printout of the circuit
      }

      assert( stackTop <= nVertices );
      ++lengthHistogram[ stackTop  ]; // add this circuit's length to the length histogram

      nCircuits++;                    // and increment count of circuits found

      if( stackTop > lenLongest ){    // keep a copy of the longest circuit found
        lenLongest = stackTop;
        longestCircuit = stack.dup;
      }

      for(int i=0;i<stackTop;i++)     // increment [circuit-length][vertex] for all vertices in this circuit
        ++vertexPopularity[stackTop ][ stack[i] ];

      f = true;
    }else if( !blocked[w] ){
      if( circuit(w) ) f = true;
    }
  }

  if( f ){
    unblock(v);
  }else{
    for(int wPos=1; wPos <= Ak[v][0]; wPos++){  // for each w in list Ak[v]:
      int w = Ak[v][wPos];
      if( w < start ) continue;  // ignore relevant parts of Ak

      if( notInList(B[w], v) ) addToList(B[w], v);
    }
  }
  v = stackPop();
  return f;
}
```

Figure 5: Recursive circuit enumeration

```
void setupGlobals(){            // presupposes nVertices is set up
  Ak.length = nVertices;        // Ak[i][0] is the number of members, Ak[i][1]..Ak[i][n] ARE the members, i>0
  B.length = nVertices;         // B[i][0]  is the number of members, B[i][1]..B[i][n] ARE the members, i>0
  blocked.length = nVertices;   // we use blocked[0] .. blocked[n-1], i>=0

  for(int i=0; i<nVertices; i++){
    Ak[i] = newList(nVertices);
    B[i]  = newList(nVertices);
    blocked[i] = false;
  }

  lengthHistogram.length = nVertices+1; // will use as [1]...[n] to histogram circuits by length
                                        // [0] for zero length circuits, which are impossible
  for(int len=0;len<lengthHistogram.length;len++) // initialise histogram bins to empty
    lengthHistogram[len] = 0;

  stackInit(nVertices);

  vertexPopularity.length = nVertices+1;  // max elementary circuit length is exactly nVertices
  for(int len=0;len<=nVertices; len++){
    vertexPopularity[len].length = nVertices;
    for(int j=0;j<nVertices; j++){
      vertexPopularity[len][j] = 0;
    }
  }
}

main(){
    setupGlobals();
    stackClear();
    start = 0;
    while( start < nVertices ){
      if( verbose && enumeration )fwritef(stderr,"Starting s=\%d\n", start );
      for(int i=0; i<nVertices; i++ ){  //for all i in Vk
        blocked[i] = false;
        emptyList(B[i]);
      }
      circuit(start);
      start = start + 1;
    }
}
```

Figure 6: Main calling code and initialisation of globals and measurement variables.

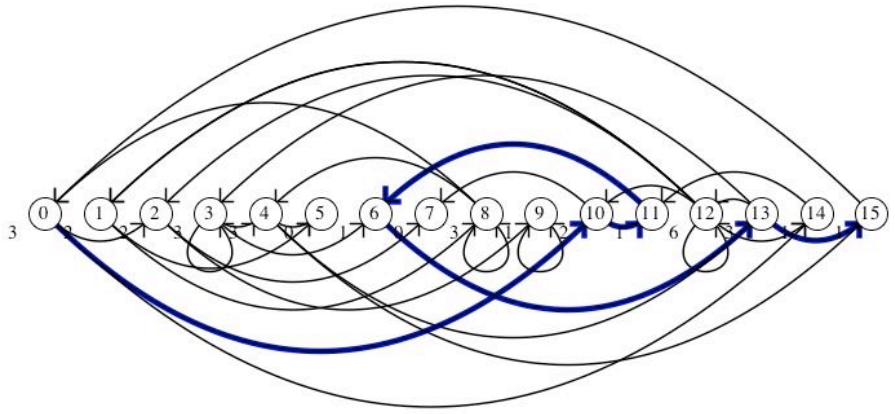

Figure 9: 16 Node Network with $K = 2$ inputs, showing the output degree of each node and one of the circuits in the graph, connecting node 0 to node 15. This network allows self-arcs.

```
0  10  11  6  13   3  4  15  0
0  10  11  6  13  12  1   8  0
0  10  11  6  13  12  1   8  4 15 0
0  10  11  6  13  12  1   8  0
0  10  11  6  13  12  1   8  4 15 0
0  10  11  6  13  15  0
0  14  11  6  13   3  4  15  0
0  14  11  6  13  12  1   8  0
0  14  11  6  13  12  1   8  4 15 0
0  14  11  6  13  12  1   8  0
0  14  11  6  13  12  1   8  4 15 0
0  14  11  6  13  15  0
```

```
 1   8   4  13 12 1
 1   8   4  13 12 1
 3   3
 3   4  13   3
 3   6  13   3
 6  13  12  10 11 6
 6  13  12  14 11 6
 8   8
 9   9
12  12
```

Figure 10: 22 Circuits found in the network shown in figure 9 which has 16 nodes and 32 arcs and allows self-arcs. Note there are repeated circuits due to the presence of a multiple-arc connecting nodes 12 and 1.

Len L	Nodes	Arcs	Circuits	Max Length	Time (secs)
2	4	16	48	4	< 0.1
3	9	36	642	9	< 0.1
4	16	64	29,440	16	≈ 0.1
5	25	100	4,367,030	25	5.5
6	36	144	1,991,637,504	36	$3,271$

Table 1: Nearest-Neighbour Periodic Square $L \times L$ Meshes – Graph and Circuit Properties

5 Discussion and Conclusions

There are a number of interesting graph applications for which it is interesting to study elementary circuits. A simple one arises in work-flow applications with task graphs where it is often important to determine if there are any loops or circuits. Such a loop can mean the task graph or work-flow is indeterminate and can therefore be flagged as an error in its specification. The algorithm and code we have presented can be readily adjusted to simple return a boolean as soon as it finds such a loop or circuit or to search all possibilities before returning a "false" as proof that there are no such loops. This sort of routine could do without with much of the measurements and support infrastructure code we have presented.

We have studied some highly regular graphs formed from simple meshes. This gives us some heuristics and intuition about the practical limitations on the size of graphs for which we can study the number of circuits explicitly. On a typical modern 64-bit word size workstation (2.66GHz) and operating system with 4GBytes of memory we can count exactly the number of circuits in a nearest-neighbour connected mesh of $36 = 6 \times 6$ nodes in less than an hour. A mesh of $49 = 7 \times 7$ is just possible, taking 3 days execution time. Larger than this we run out or memory (and indeed patience).

Another interesting issue concerns the presence of Hamiltonian circuits. These are circuits that include all nodes in a graph. Random graphs or those generated according to a statistical distribution or arising from some application can have multiple Hamiltonian routes if there are multiple arcs present. In our study of Kauffman networks [1] we found that in highly complex networks, the presence of Hamiltonian circuits and also circuits of length greater than half the number of nodes were interesting indicators of a phase transitions.

The number of circuits grows very rapidly, particularly in highly connected graphs. We were able to study Kauffman systems of up to around 100 nodes but only for very low connectivities of around $K = 2, 3$. For higher connected systems, only graphs with a small numbers of nodes such as $30 - 40$ are possible in a practical execution time.

We have shown how Johnson's algorithm can be adjusted to cope with self- and multiple arcs and how a fast and practical implementation can be implemented using the capabilities of a systems level object-oriented language like D. Despite the practical limitations on graph size, we believe a exact study of circuits yields very interesting insights into certain network problems such as small-world systems, biological, social and Internet systems.

References

[1] Hawick, K., James, H., Scogings, C.: Structural circuits and attractors in kauffman networks. In Ab-

```
// return a pointer to a list of fixed max size
int[] newList(int max){
  int[] retval; retval.length = max + 1;
  retval[0] = 0;
  return retval;
}

// return TRUE if value is NOT in the list
bool notInList(int[] list, int val) {
  assert(list != null);
  assert(list[0] < list.length);
  for(int i=1; i<=list[0]; i++) {
    if(list[i] == val)
      return false;
  }
  return true;
}

// return TRUE if value is in the list
bool inList(int[] list, int val) {
  assert(list != null);
  assert(list[0] < list.length);
  for(int i=1; i<=list[0]; i++) {
    if(list[i] == val)
      return true;
  }
  return false;
}

// empties a list by simply zeroing its size
void emptyList(int[] list){
  assert( list != null );
  assert( list[0] < list.length );
  list[0] = 0;
}

// adds on to the end (making extra space if needed)
void addToList(inout int[] list, int val){
  assert(list != null);
  assert( list[0] < list.length );

  int newPos = list[0] + 1;
  if(newPos >= list.length )
    list.length = list.length + 1;

  list[ newPos ] = val;
  list[0] = newPos;
}

// removes all occurences of val in the list
int removeFromList(int[] list, int val){
  assert( list != null);
  assert( list[0] < list.length );
  int nOccurrences = 0;
  for(int i=1;i<=list[0];i++){
    if( list[i] == val ){
      nOccurrences++;
      for(int j=i;j<list[0];j++){
        list[j] = list[j+1];
      }
      --list[0];  // should be safe as list[0] is
                  //  re-evaluated each time around the i-loop
      -- i;
    }
  }
  return nOccurrences;
}
```

Figure 8: List management code - note use of the D **inout** qualifier which is required for `addToList` to change its list argument.

bass, H.A., Randall, M., eds.: Proc. Third Australian Conference on Artificial Life. Volume 4828 of LNCS., Springer (2007) 189–200 978-3-540-76930-9.

[2] Hawick, K.A., James, H.A., Scogings, C.J.: Circuits, attractors and reachability in mixed-k kauffman networks. Technical Report CSTN-046; arXiv:0711.2426, Massey University (2007)

[3] Hawick, K.A., James, H.A.: Node importance ranking and scaling properties of some complex road networks. Technical report, Information and Mathematical Sciences, Massey University, Albany, North Shore 102-904, Auckland, New Zealand (2005)

[4] Tiernan, J.C.: An efficient search algorithm to find the elementary circuits of a graph. Communications of the ACM **13** (1970) 722–726

[5] Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM Journal on Computing **2** (1973) 211–216

[6] Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing **4** (1975) 77–84

[7] Bright, W.: D Programming Language. Digital Mars. (2008)