A new way to enumerate cycles in graph

Hongbo Liu, Jiaxin Wang
State Key Lab of Intelligent Technology and System
Department of Computer Science and Technology, Tsinghua University
Beijing 100084, China
lhb00@mails.tsinghua.edu.cn

Abstract

In many cases, the topology of communcation systems can be abstracted and represented as graph. Graph theories and algorithms are useful in these situations. In this paper, we introduced an algorithm to enumerate all cycles in a graph. It can be applied on digraph or undirected graph. Multigraph can also be used on for this purpose. It can be used to enumerate given length cycles without enumerating all cycles. This algorithm is simple and easy to be implemented.

1. Introduction

Connection is a significant feature in communication systems. Thus they can be modeled and and described as networks when dealing with some problems. Graph theories and algorithms are useful tools in the study of communication. Cycle is one of the distinct features to characterize a graph. The analysis of cycles in networks have different applications in the design and development of communicaiton systems, such as the investigation of topological features [6], and consideration of reliability and fault tolerance [4], etc. There are various problems related to the analysis of cycles in networks. Enumerating or counting all cycles or given length cycles are among them. Of course, counting problem can be easily solved if corresponding enumeration method can be achieved.

To enumerate all cycles in a graph, generally two kinds of methods have been introduced. One method is based on the cycle vector space [3, 5]. All cycles and edge-disjoint unions of cycles in an undirected graph form a vector space over $GF(2)$. The dimension of the space is $\mu = e - n + 1$, where e is the number of edges and n is the number of the vertices. The basis of the vector space can be constructed from a spanning tree. You can obtain all the cycles with ring-sum operations in the space.

However, the implemented algorithm of this method is slow, since generally only a very small fraction of the $2^\mu - 1$ vectors could be cycles. The ratio of cycles in the vector space tends to zero for numerous classes of graphs. Digraphs can not simply use the properties of linear space of undirected graphs.

The other is search method. In this class, Johnson [2] and Tarjan [7] use backtracking to find the cycles in graphs. In their algorithms, pruning method was used to improve the efficiency of computation. Johnson's algorithm has an upper time bound of $O((n+e)(c+1))$, where c is the number of cycles. It is much faster than the vector space method.

Perhaps because of the good results achieved by Johnson and Tarjan, we found little discussions on this topic in recent years. However, in their algorithm, the process of pruning seems complicated. It can not be used to find given length cycles without completing a thorough computation on the graph. And the algorithm is not easy to be realized in parallel directly. In this paper, we present an algorithm to find cycles in graph. It can be applied on digraph or undirected graph. It can be used to find given length cycles without enumerating all cycles in the graph and it can also be realized in a natural and simple way.

2. Preliminaries

For the convenience of description, some terms used in this manuscript are given in the following.

A graph $G = (V, E)$ comprises a set of vertices V and a set of edges E which are 2-element subsets of V. A subgraph G' of a graph G is a graph whose vertices and edges are subsets of those of G. A path is a alternating vertices and edges, beginning and ending with a vertex, $v_1 e_1 v_2 e_2 ... e_{n-1} v_n$ such that every consecutive pair of vertices v_x and v_{x+1} are adjacent and incident with e_x. The length of a path is the number of edges in the path. The start vertex of a path is called head. The end vertex of a path is tail. A simple path is a path such that all the vertices except head and tail are distinct. When the head and tail are identical, the simple path are called cycle. k-cycle denotes the cycle with length k. If a simple path is not a cycle, it is an open path, which means that all vertices in an open path are distinct. If the edge in the graph is ordered pair of vertices written as $\langle v_i, v_j \rangle$, we call the graph as digraph. If the vertices of an edge denoted with $\{v_i, v_j\}$ do not have ordered orientation, the graph is undirected graph.

In principle, the algorithm enumerating cycles of a digraph can be used to enumerate cycles of an undirected graph. So we focus our attention on digraph to describe the algorithm.

3. Our Algorithm

To acquire all cycles in a digraph, a straight way is to enumerate all combinations of different edges and verify if the set of edges can construct a cycle. However, in this way, the amount of computation seems awful, because there are so many combinations most of which could not construct a cycle.

The combination of the edges could be cycle, open path, and subgraph which is union of cycles and open paths. As we study the combinations attentively, we can find that there is some relationship between k-cycle and open path of length $k-1$. Any $k-$cycle is composed of an open path of length $k-1$ and an edge with which the tail and head of the open path are incident. And an open path of length k is composed of an open path of length $k-1$ and an adjacent edge of tail whose end does not occur in the open path. So if we obtain all open paths of length $k-1$, we can generate all open paths of length k and all k-cycles. This leads to our algorithm.

The main process of our algorithm is to generate all k-cycles and k length open paths from $k-1$ length open paths. This procedure can be iterated until there is no open path generated. When the process goes on, k-cycles are enumerated and k increases progressively. However, using this method, duplicate cycles may be generated from different vertices of the same cycle. To avoid this, integer designators $1, 2, ..., N$ are assigned to vertices as their orders in the graph, and a constraint is added that the order of vertex adjacent to the tail should be greater than that of the head when generating a new k length open path from a $k-1$ length open path.

The flowchart of the algorithm is illustrated in Figure 1. In this figure, $order(v_i)$ returns the order of vertex v_i and $length(P)$ returns the length of path P. The procedure of the algorithm is described as follows.

First, you should initialize the graph from which you want to enumerate the cycles. The graph can be represented as an adjacent matrix or an adjacent list, which doesn't matter. Every vertex is assigned a unique integer as its order.

We use a FIFO queue to store the open paths and a register to record the length of the path. The register is not

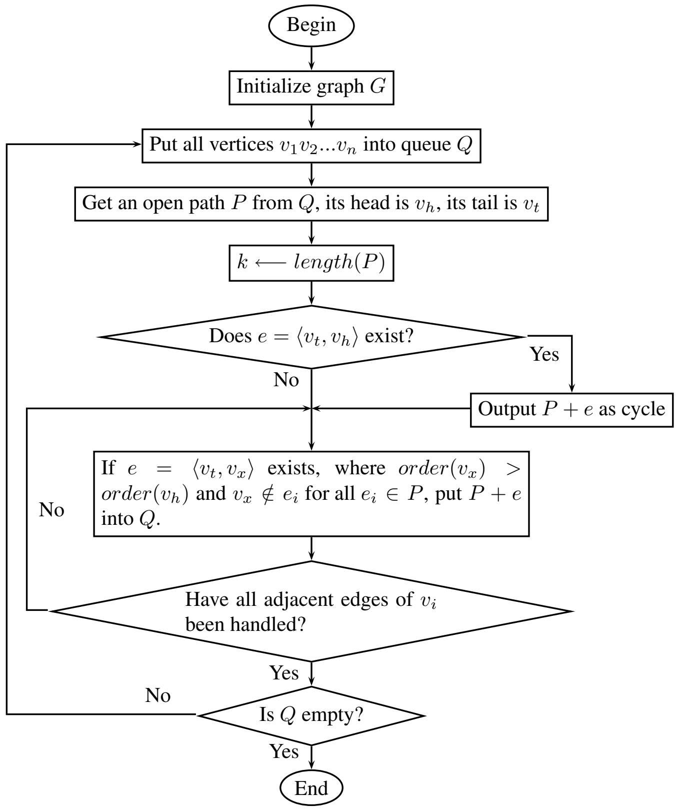
Begin
Initialize graph G
Put all vertices $v_1v_2...v_n$ into queue Q
Get an open path P from Q, its head is v_h, its tail is v_t
$k \longleftarrow length(P)$
Does $e = \langle v_t, v_h \rangle$ exist?
Yes
No
Output $P + e$ as cycle
If $e = \langle v_t, v_x \rangle$ exists, where $order(v_x) > order(v_h)$ and $v_x \notin e_i$ for all $e_i \in P$, put $P + e$ into Q.
No
Have all adjacent edges of v_i been handled?
Yes
No
Is Q empty?
Yes
End

Figure 1. Flowchart of our algorithm

necessary, because you can always get the length from the open path directly.

The algorithm starts with all vertices of the graph. First, put all vertices into the queue and set the register to 0.

Then the iteration of main loop of the algorithm starts.

Fetch an open path from the queue. Its length is k which is indicated by the register.

Verify if there is an edge which links the tail to the head of the open path. If it is true, a cycle is enumerated, and then output the cycle. When the register is 0 in such case, it means the cycle is a selfloop.

Then get an adjacent edge of the tail whose end does not occur in the open path and the order of its end is greater than the order of the head. This edge and the k length open path construct a new $k+1$ length open path. Put this new open path into the queue.

After having generated all the $k+1$ length open paths from the k length open path, if this open path is the last k length open path of the queue, set register to $k+1$.

If the queue is empty, the algorithm finishes, else jumps to where the main loop starts.

Before enumerating cycles with this algorithm, you'd better do some pretreatment on the digraph . An unconnected graph may be divided into connected components of the graph, so you can use the algorithm to treat each connected component separately. Selfloops can be enumerated alone. For multigraph, a set of p parallel edges can be replaced by a new edge. When enumerating a cycle containing the new edges, enumerate p cycles, each containing a parallel edge replaced. The purpose of these treatments is to use the algorithm conveniently or improve the speed of it. They are not of necessity. The algorithm can work well without such treatments.

Although we described the algorithm with digraph in this manuscript, it can be applied to an undirected graph as well.

When enumerating cycles of an undirected graph G, normally we suppose G is simple. If this is not true, you can do some pretreatment on G to eliminate the parallel edges and selfloops to make it simple. Then by replacing each undirected edge with two oppositely directed edges, we get a directed graph D which is the directed counterpart of G. For each cycle of G, there are two corresponding cycles in D in opposite direction. Thus $C_G = 2C_D$, where C_G is the number of cycles in G and C_D is the number of cycles in D. Enumerating cycles whose lengths are greater than 3 in D using our algorithm, you can obtain the corresponding cycles of G. Disregarding the computational speed, we can use this method on undirected multigraph without selfloops directly.

Since all kinds of graphs are subgraphs of complete graph, complete graph is a useful tool to test the speed of the algorithm. Here, we use the results of Johnson [2] and Tiernan [8] from their papers for comparison. As we mentioned in the introduction, they are all good search methods and are very efficient among all these algorithms. The complete graph of Tiernan includes selfloops, which do not affect the comparison. Because the results are obtained in different computing environment, only the comparison of relative running time of the same algorithm is meaningful. The results of running time are given in Table 1. From this table, we can infer that our algorithm is not as good as their algorithms in the speed of computation.

Number of vertices	Number of cycles	Running time		
		Johnson's algorithm	Tiernan's algorithm	Our algorithm
2	1	0	.13	.002
3	5	0	.15	.003
4	20	.02	.30	.008
5	84	.02	.80	.019
6	409	.07	4.20	.104
7	2365	.35	27.60	.798
8	16064	2.43	219.00	10.047
9	125664	20.17		524.95

Table 1. Running time on complete digraphs

4. Conclusion

Compared with some efficient search algorithms already known, it seems that our algorithm is not efficient enough. However, it has its own advantages.

It can enumerate $k-$cycles without enumerating all cycles in the graph. In such cases, the process of computation finishes when the register increases to $k+1$. The algorithm has good generality. It can be used on digraph and undirected graph. It can also be applied on multigraph straightly. With sharing the same queue by multiprocess, the algorithm can be implemented in parallel naturally.

The procedure of this algorithm is not very complicated. So it can be realized efficiently. You can get what you wanted directly avoiding some errors and wrong results in programming.

Thus this algorithm is suitable for the situation that the scale of the graph you deal with is not very large and you want to find a simple and elegant way to enumerate the cycles of it.

Some problems such as the improvement of the performance of our method or the study of the scalability of this algorithm will be addressed in future.

References

[1] R. W. Floyd. Nondeterministic algorithms. *J. Assoc. Comput. Mach*, 4:636–644, 1967.

[2] D. B. Johnson. Find all the elementary circuits of a directed graph. *J. SIAM*, 4:77–84, 1975.

[3] P. Mateti and N. Deo. On algorithms for enumerating all circuits of a graph. *SIAM J. Comput.*, 5:90–99, 1976.

[4] M. Médard and S. S. Lumetta. Network reliability and fault tolerance. In J. Proakis, editor, *Wiley Encyclopedia of Engineering*.

[5] V. V. B. Rao and V. G. K. Muti. Enumeration of all circuits of a graph. *Proc. IEEE*, 57:700–701, 1969.

[6] H. D. Rozenfeld et al. Statistics of cycles: how loopy is your network? *J. Phys. A: Math. Gen.*, 38:4589–4595, 2005.

[7] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *J. SIAM*, 2:211–216, 1973.

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