Special issue: Theoretical computer science and discrete mathematics

Research Paper

Implementation issues of clique enumeration algorithm

Takeaki UNO[1]
[1]*National Institute of Informatics*

ABSTRACT

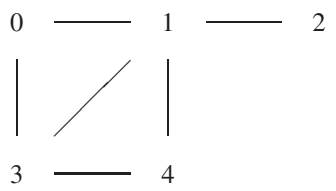
A clique is a subgraph in which any two vertices are connected. Clique represents a densely connected structure in the graph, thus used to capture the local related elements such as clustering, frequent patterns, community mining, and so on. In these applications, enumeration of cliques rather than optimization is frequently used. Recent applications have large scale very sparse graphs, thus efficient implementations for clique enumeration is necessary. In this paper, we describe the algorithm techniques (not coding techniques) for obtaining efficient clique enumeration implementations.

KEYWORDS

maximal clique, enumeration, community mining, cluster mining, polynomial time, reverse search

1 Introduction

A graph is an object, composed of vertices, and edges such that each edge connects two vertices. The following is an example of a graph, composed of vertices 0, 1, 2, 3, 4, 5 and edges connecting 0 and 1, 1 and 2, 0 and 3, 1 and 3, 1 and 4, 3 and 4.



For a graph $G = (V, E)$ of vertex set V and edge set E, a clique is a subset of V such that any two its vertices are connected by an edge of E. For example, the vertices 0, 1, and 3 compose a clique. A vertex itself is a clique, and two vertices connected by an edge is also a clique. A clique is called maximal if it is included in no other clique. For example, vertices 0 and 1 compose a clique, but it is not maximal clique. vertices 0, 1, and 3 compose a clique, and it is also a maximal clique.

Received October 10, 2011; Revised December 6, 2011; Accepted December 6, 2011.

[1)] *uno@nii.ac.jp*

DOI: 10.2201/NiiPi.2012.9.5

A maximum clique is hard to find in the sense of time complexity, since the problem is NP-hard. Finding a maximal clique is easy, and can be done in $O(\Delta^2)$ time where Δ is the maximum degree of the graph. Cliques can be considered as representatives of dense substructures of the graphs to be analyzed, the clique enumeration has many applications in real-world problems, such as cluster enumeration, community mining, classification, and so on. Real-world graph data often has huge sizes but they are usually sparse. The clique sizes are usually bigger than the expected size in the random graphs of the same sizes. It means that the graphs usually have locally dense structures. On the other hand, the number of cliques is usually not so huge, and is often tractable even for huge size graphs. Applications having graphs of big sizes but sparse structures are listed below.

Finding web communities

Consider a web network graph, in which vertices are web sites, and two vertices are connected by an edge if the corresponding sites are linked. Then, a clique of this web network graph can be considered to compose a community, since they have link in any pair of its members. Finding web communities by hand is not an easy task. By enumerating the cliques in the web network graph, we can automatically find candidates and seeds

of communities.

Application; finding clusters

Suppose that we have data of collection of objects, and we want to find groups which we can consider that the members of the group are similar, or have common features. Such a group is called a cluster. Consider a relation graph such that two objects similar to each other is connected. Then, any pair of members of a clique are similar, thus a clique should be totally included in a cluster. Therefore, cliques are used as seeds of clusters. By joining cliques, or appending a clique iteratively, clusters can be found. For the purpose maximal cliques are good to append, and maximal clique enumeration does a big help.

Existing algorithms

Clique enumeration is actually easy, since the family of cliques satisfies the monotone property. It admits a simple hill climbing algorithm. The iterations can be fastened by updating the candidates to be added to the currently operating clique, as described below. Conversely, maximal clique enumeration is not easy; maximal cliques can not be traversed by small changes, such as constant number of addition/deletion of vertices. There are mainly two algorithms for enumerating maximal cliques; one is Makino-Uno algorithm [2] and the other is Tomita algorithm [3]. The Makino-Uno algorithm is based on reverse search, and is a polynomial time delay algorithm. The Tomita algorithm is not polynomial time, and uses an efficient pruning methods. By computational experiments, Tomita algorithm is faster for relatively dense graphs, and Makino-Uno algorithm is faster for relatively sparse graphs. Makino-Uno algorithm is based on the algorithm proposed by Tsukiyama et al. [1].

Contribution

In this paper, we describe the details of the implementation issues of Makino-Uno algorithm. The implementation is named MACE (MAximal Clique Enumerator), and is available on the author's Web site (`http://research.nii.ac.jp/~uno/codes.htm`). On the Web site, an implementation of the Tomita algorithm is also available. The program parts are common, thus we can fairly compare the performance.

A subgraph is called bipartite clique if it is a complete bipartite graph, i.e., the vertex set of the subgraph can be partitioned into two groups such that no edge connect two vertices from the same group, and there is always an edge connecting two vertices from one group and the other group. The bipartite clique enumeration problem can also be solved by clique enumeration, by adding edges for all pairs of vertices in the same vertex set. If the bipartite graph is sparse, the obtained graph becomes dense, thus for the practical efficiency, we should use an implementation designed for bipartite cliques. An implementation named LCM, available on the author's Web site, is designed for enumerating closed itemsets in a transaction database. A closed itemset is a maximal bipartite clique in the graph representing inclusion relation between items and transactions in the database, thus we can use LCM for enumerating bipartite maximal cliques. The performance is significantly better than MACE in the case.

The organization of this paper is as follows. In the next section, we describe Makino-Uno algorithm, and the implementation issues and techniques for fast computation. We describe the usage of the implementation in Section 3, and conclude the paper in Section 4.

2 Algorithm and implementation issue

Let $G = (V, E)$ be a graph with vertex set $V = \{1, \ldots, n\}$ and edge set defined on V. Let $tail(K)$ be the largest vertex in K. Since the graph would be large scale and very sparse, we load the graph to the memory by using adjacency list. Adjacency list is a data structure for storing graph structure in memory, that uses small amount of memory for sparse graphs. For each vertex, we assign an array of integer, and store the vertices adjacent to the vertex in the array. The size of array is thus equal to the degree of the vertex. The memory usage is one pointer for each vertex, and two integers for each edge.

Enumeration of cliques is actually easy. Since any subset of a clique is also a clique, we can find any clique by iteratively adding vertices to the emptyset. This yields a depth-first backtracking algorithm; In each iteration, for each vertex v not in the current clique K, if $K \cup \{v\}$ is a clique then recursively call the iteration to enumerate the cliques including $K \cup \{v\}$. To avoid duplications, we add only vertices larger than $tail(K)$.

Squeeze the candidates to be added

To execute the above operations quickly, we maintain the vertex set $CAND(K)$ which is the set of vertices adjacent to all the vertices in K. $K \cup \{v\}$ is a clique if and only if v is in $CAND(K)$, so we have to generate recursive calls for the vertices in $CAND(K)$. When K is the emptyset, then $CAND(K)$ is V, it means $K \cup \{v\}$ is a clique for any vertex v. Denote by $N(v)$ the set of vertices adjacent to v. Then $CAND(K \cup \{v\})$ is the intersection of $CAND(K)$ and $N(v)$. By having $CAND(K)$ and $N(v)$ in sorted order in the memory, we can take the intersection in linear time in the sizes of them. Moreover, in each iteration, we need only vertices larger than $tail(K)$, thus in the bottom of the recursion, the vertices

to be maintained are so few. The bottom of the recursion dominates the total computation time in usual enumeration algorithms, thus clique enumeration is usually done very quickly.

Maximal clique enumeration by reverse search

If we use the above algorithm to enumerate maximal cliques, by outputting only the maximal cliques among all cliques, we need very long time compared to the number of maximal cliques. Thus, we use another algorithm, search method. For a clique K, let $C(K)$ be the lexicographically minimum one among the maximal cliques including K. A set $A = \{a_1, \ldots, a_n\}$ is said to be lexicographically smaller than a set $B = \{b_1, \ldots, b_m\}$ if and only if $a_1 = b_1, \ldots, a_k = b_k$, and $a_{k+1} < b_{k+1}$ holds for some k, or $B \subseteq A$. We note that k can be 0. We denote by $K(i)$ the set of vertices in K less than i. For a maximal clique K, we define the core index $core_i(K)$ of K by the minimum vertex satisfying $C(K) = C(K(i))$. Using these terms we define the parent of K, not equal to $C(\emptyset)$, by the maximal clique $C(K(core_i(i)))$. The parent is uniquely defined, and lexicographically smaller than K. Thus, the binary relation "parent-child" induces a directed tree, rooted at the the lexicographically minimum maximal clique. Thus, starting from the lexicographically minimum maximal clique, and perform a depth first search on the tree, we can enumerate all maximal cliques. The depth first search needs an algorithm for finding all children of a given maximal clique, and it is enough, since what we have to do is recursively find children of maximal cliques.

We can prove that K is obtained from the parent $P(K)$ of K by adding $core_i(K)$ to $P(K)$, remove vertices not adjacent to $core_i(K)$, i.e., $K = (C(P(K) \cap N(core_i(K))) \cup core_i(K))$. Conversely, for a maximal clique K', K is a child of K' only if $K = C(K' \cap N(v) \cup \{v\})$. To check whether K is a child of K' or not, we compute the parent of K, and if its parent is K', then K is a child of K'.

Characterization of children

For a maximal clique $K = C(K' \cap N(v) \cup \{v\})$ obtained from K', the condition that K is a child of K' can be characterized in the following way so that we can check it efficiently.

(a) $K(v) = K'(v) \cap N(v)$
(b) for any u in $K(u)$, there is no vertex $w < u$ not in $K'(v)$ such that
u is adjacent to all vertices in $K(v) \cup K'(u)$.

We can see that if either condition is violated, the parent of K is not K'.

Sweep pointer method

To check this conditions, we use sweep pointer method. (i) is violated when there is a vertex $u < v$ adjacent to all vertices in $K'(v) \cap N(v) \cup \{v\}$. We trance $N(v)$ in the increasing order of vertices, and when we meet a vertex u not in $K'(v)$, check whether u is adjacent to all vertices in $K'(v) \cap N(v)$ or not. This can be done by tracing $N(w)$ for each vertex w in $K'(v) \cap N(v)$. Notice that to check the next vertex in $N(v)$, we can start the trance of the $N(w)$'s from the positions which we terminated the trace for checking it for u. This tracing method, called sweep pointer does not increase the computation time much since the time complexities are the same, and usually we check only few items and conclude that the maximal clique is not a child. Thus, we can check it very quickly. The second condition can be checked in the same way, but notice that we have to check the vertices one-by-one, such that the next vertex is adjacent to all of them. The two checks can be done simultaneously so that we do not have to scan the $N(w)$'s twice.

Moreover, when the size of clique is small, we can use bit operations (AND and OR) so that we can operate 32 or 64 bits at once. Suppose that the size of the current maximal clique K is less than 32. We give ID's $0, 1, 2, 4, \ldots 2^{32}$ to each vertex. Let the bit-mark $b(S)$ of a subset S of K be the sum of ID's of the vertices in S. For each vertex u adjacent to at least one vertex in K, we compute $b(N(u) \cap K)$. Then, a vertex u is adjacent to all vertices in $K \cap N(v)$, if and only if $b(N(u)) \cap b(N(v)) = b(N(v))$. Here $\cap$ means the and operation for bit string, and two numbers are regarded as bit-strings. This operation can be done in very few steps, thus we can accelerate the computation for the checking of the two conditions. When we go to a child of K and check the conditions for finding the children of the child, we re-use the bit-mark by replacing the ID's of removed vertices and newly added vertices, and updates the bit-marks. In practice it accelerates the speed of the algorithm 2 or 3 times when the data set is sparse and the maximal clique sizes are small on average, say 10.

2.1 Performance

This algorithm is polynomial delay, taking at most $O(|V|)$ time for each clique in the case of clique enumeration, and $O(|V||E|)$ time for each maximal clique in the case of maximal clique enumeration, where $|V|$ and $|E|$ are the number of vertices, and the number of edges in the input graph. Practical computation time is constant for each clique/maximal clique, and finds about 1,000,000 cliques per second in the case of clique enumeration, and fine about 100,000 maximal cliques per second in the case of maximal clique enumeration,

if the graph is sparse. As increase the density of the graph, the computation time increases almost linearly.

3 Usage of the implementation

The program is written in C code (gcc). It uses only the basic library, so you can compile it in any environment. To compile the program, first put all the source files in a directly, and just execute

% make

Then, you can execute "mace". The format of the input parameter is,

% mace [commands] [options] input-filename [output-filename]

[options] and [output-filename] can be abbreviated. MACE is executed with at least two parameters. The first parameter is composed of commands, given by a combination of letters. The meaning of the letters are:

- _ : do not output the reports of the execution, such as size of input graph, to the standard output
- % : show progress, by outputting "——" for each 10,000 cliques
- + : if the output file exists, append the solutions to the output file
- C : enumerate cliques of the give graph
- M : enumerate maximal cliques of the given graph
- s : terminate after finding 1,000,000 cliques

The second parameter is the name of the input file, and the third parameter is the name of output file. You can omit the output file name. In this case, the program only counts the numbers of cliques to be output, classified by their sizes. Note that the name of the input file can not start with "-". If the output file name is "-", the solutions will be output to the standard output.

Between the first parameter and the second parameter, you can give some options as follows.

- -# [num] : stop after outputting [num] solutions
- -, [char] : give the separator of the numbers in the output the numbers in the output file are separated by the given character [char].
- -Q [filename] : replace the output numbers according to the permutation table written in the file of [filename], replace the numbers in the output. The numbers in the file can be separated by any non-numeric character such as newline character.
- -l [num] : output cliques with size at least [num]
- -u [num] : output cliques with size at most [num]

For example, by executing

mace C -l 5 -u 7 g100.grh clq.out

the program outputs all cliques of sizes from 5 to 7 (including both size 5 and size 7) in the "g100.grh" to the file "clq.out".

Example of the execution

mace M% g15.grh clq.out
(output maximal cliques in "g15.grh" to "clq.out". Show the progress during the execution)

mace Cq -# 1000000 -l 5 g15.grh clq.out
(output cliques in "g15.grh" of at least size 5 to "clq.out". Stop if over 1,000,000 cliques are found.)

3.1 Output file format

The input graph, with n vertices, is considered to be composed of vertices from 0 to $n-1$. The cliques found are output to the output file specified by user, represented by a sequence of numbers corresponding to the vertices of the clique. One line of the output file is for one clique. The numbers in each line is separated by " ", and by giving "-," option we can change the separator. At the termination of the program, it outputs the number of cliques found, and the number of cliques classified by their sizes. For example, if there are cliques {0,1}, {2}, {0,1,3}, {1,2}, the output to the standard output will be

```
#vertices=3 #edges 5 ⇐ numbers of vertices and edges
4                    ⇐ total number
0                    ⇐ number of cliques of size 0
1                    ⇐ number of cliques of size 1
2                    ⇐ number of cliques of size 2
1                    ⇐ number of cliques of size 3
```

and the output file will be

```
0,1
2
0,1,3
1,2
```

If q is given in the parameter, then "#vertices, #edges"

is not printed. If output file name is not given, then no output file is generated.

The output cliques are not sorted. If you want to sort it, use the script "sortout.pl". The usage is just,

% sortout.pl ⟨ input-file ⟩ output-file

"input-file" is the name of file to which mace output, and the sorted output will be written in the file of the name "output-file". The vertices of each clique will be sorted in the increasing order of indices, and all the cliques (lines) will be also sorted, by the lexicographical order (considered as a string). (Actually, you can specify separator like sortout.pl ",").

3.2 Input file format

Each following i-th line is the list of vertices adjacent to vertex $i-1$, so each line is: (vertex), (vertex), ... Any non-numeric letter (except for newline and end-of-file) is allowed to be used for the separator. Each vertex has to be ranged from 0 to (#vertices−1).

For an edge connecting vertices u and v, we do not need to write both in the output file "v in the (u-1)th line" and "u in the (v-1)th line". We need just one, "v in the $(u-1)$th line" or "u in the $(v-1)$th line." If you write both, an edge will be counted doubly, and mace does not work correctly.

Example) a graph with edges (1,0), (2,0), (1,3), (2,3), (3,4), (0,4):

```
1,2
3

2 4
0
```

3.3 Transforming other graph file format

For the use of other formats for input graph files, we have several scripts. We explain the functions of the Perl scripts in the following.

– compgrh.pl [b] [separator] ⟨ input-file ⟩ output-file

Write to output-file the complement graph of the graph read from the input-file. If you specify "b" option, then the input-file is regarded as a bipartite graph.

Ex.) % compgrh.pl b "," ⟨ test15.grh ⟩ test2.grh

– transnum.pl table-file [separator] ⟨ input-file ⟩ output-file

Read file from standard input, and give a unique number to each name written by general strings (such as ABC, ttt), and transform every string name to a number, and output it to standard output. The mapping from string names to numbers is output to table-file. The default character for the separator of the string names is " "(space). It can be changed by giving a character for the option [separator]. For example, A, B is a string name, if the separator is space, but if we set the separator to ",", it is regarded as two names A and B. This is executed by "transnum.pl table-file ","⟨ input-file ⟩".

– untransnum.pl table-file ⟨ input-file ⟩ output-file

According to the table-file output by transnum.pl, untransform numbers to string names. The output of the program is composed of numbers, thus it is used if we want to transform to the original string names. It reads file from standard output, and output to the standard output.

– transgrh.pl [Bb] [separator] ⟨ input-file ⟩ output-file

Transform a file in the format of that every line writes two end vertices of an edge, to the format of this program. For example, the file

```
0,2
0,1
1,4
3,4
1,3
```

representing the graph with edge (0,2), edge (0,1) and ..., is transformed to

```
1,2
3,4

4
```

If the name of vertices are written in general strings, transform them to numbers by transnum.pl before the execution. When we give parameter D or d, the input graph is regarded as a directed graph. If we give d, the first number is the origin of a directed edge, and if we give D, the second number is the origin. Bipartite graph can be transformed by giving options b or B. Suppose that we have a bipartite graph with vertex sets A and B both indices start from 0. If we transform it by the above way, vertex i in A and i in B are considered to be the same vertex. By giving b, the second number in each row is added by a constant so that no two vertices have the same index. In the case of B, the first number is added. When we transform the above graph by (transgrh.pl b "," ⟨ input ⟩ output), we have a graph in the format,

```
5,6
7,8

8
```

– sortgrhid

Transform a graph file format such that each row corresponds to a vertex. The first number of each row is the ID of the vertex, and the following numbers are the vertex ID's to which the vertex adjacent. The script sorts the row according to the ID's, and remove the ID from the file. For example, the file,

```
4,2,3
1,2
0,1,3,4
3
2,3
```

is transformed to

```
1,3,4
2
3

2,3
```

If the ID's are given in general strings, use transnum.pl before the execution.

Example of the usage

When transform the file test.grh which is edge list format with general string vertex names with separator ",", and enumerate cliques:

```
transnum.pl table "," ⟨ test.grh ⟩ tmp.grh
transgrh.pl ⟨ tmp.grh ⟩ tmp2.grh
mace M tmp2.grh out
untransnum.pl table ⟨ out ⟩ out2
```

4 Conclusion

In this paper, we described the implementation issues of maximal clique enumeration; reverse search, reducing the candidates for children, and sweep pointer method for fast parent-child relation check. We also described the usage of the implementation in detail.

The implementation is available on the author's Web site

`http://research.nii.ac.jp/~uno/index.html`.

Anyone can modify this program, but he/she has to write down the change of the modification on the top of the source code. Neither contact nor appointment to Takeaki Uno is needed. If one wants to re-distribute this code, do not forget to refer the newest code, and show the link to homepage of Takeaki Uno, to notify the news about codes for the users. For the commercial use, please make a contact to Takeaki Uno.

Acknowledgments

We gratefully thank to Kazuhisa Makino of Tokyo University, the co-author of the paper of mace. We also thank to Zhiao Shi for bug reports. We also thank to Dr. Krister Swenson of University of Ottawa for giving an advice to improve the document.

References

[1] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, "A new algorithm for generating all the maximal independent sets," *SIAM Journal of Computing*, vol.6, pp.505–517, 1977.

[2] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," *In Proc. of the 9th Scandinavian Workshop on Algorithm Theory (SWAT 2004)*, pp.260–272, Springer-Verlag, 2004.

[3] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theor. Comp. Sci.*, vol.363, pp.28–42, 2006.

Takeaki UNO

Takeaki UNO received the PhD degree (Doctor of Science) from Department of Systems Science, Tokyo Institute of Technology Japan, 1998. He was an assistant professor in Department of Industrial and Management Science in Tokyo Institute of Technology from 1998 to 2001, and have been an associate professor of National Institute of Informatics Japan, from 2001. His research topic is discrete algorithms, especially enumeration algorithms, algorithms on graph classes, and data mining algorithms. In data mining area, his main research is on pattern mining algorithms. He got Young Scientists' Prize, of The Commendation for Science and Technology in Japan, in 2010.