http://mmg.tversu.ru ISSN 2311-1275

Mathematical Modelling and Geometry

Volume 7, No 3, pp. 1 – 8 (2019) doi:10.26456/mmg/2019-731

Algorithm of the Cyclic-Order Graph Program (Implementation and Usage)

Štefan Berežný [1,a] **and Ján Buša Jr.** [2,3,b]

[1] Departmet of Mathematics and Theoretical Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Němcovej 32, 042 00 Košice, Slovakia

[2] Laboratory of Information Technologies, Joint Institute for Nuclear Research, Joliot-Curie 6, 141980 Dubna, Moscow region, Russia

[3] Institute of Experimental Physics, Slovak Academy of Sciences, Watsonova 47, 040 01 Košice, Slovakia

e-mail: [a] stefan.berezny@tuke.sk, [b] busa@jinr.ru

Received 21 December 2019. Published 30 December 2019.

Abstract. C++ Implementation of the Cyclic-Order Graph Algorithm is presented together with the source code. This program helps in determination of crossing numbers for join products of graphs with paths of given numbers of vertices necessary for the proofs in the Graph Theory. Comparison with MATLAB (GNU Octave) is provided together with outlooks for the future development.

Keywords: crossing number, cyclic permutation, join product, paths

MSC numbers: 05C12, 05C62, 05C85, 68W40

This work was partially supported by JINR LIT in Dubna, Russia, which allowed us access to the computing resources of the HybriLIT cluster [1].

1. Introduction

One of important topics in the field of Graph Theory is a question of graphs crossing number, i.e., the minimal number of crossings needed, when constructing a graph with given properties. Such problem can arise for instance when designing motherboard where we connect electronic components (graphs vertices) by conductors (graphs edges). Naturally, we want to avoid crossing of individual conductors and to do this effectively, we need to know minimal number of crossings in such situation.

From the mathematical point of view we solve the problem of determining of crossing number of join product of the graph G with the discrete graph D_n and the path P_n on n vertices. One of most straightforward approaches when trying to prove some estimate of the crossing number was presented in [2–4]. The proof of crossing numbers of such graphs with respect of their ideas is useful identify the permutations whose distance from given two permutations in the graph have some prescribed value. While this is rather easy to do by hand for a small size of the input graph (for $n = 4$ vertices we get $(n-1)! = 3! = 6$ cyclic permutations or for $n = 5$ vertices we get 30 cyclic permutations), this task gets tiresome for larger graphs as the number of tested permutations grows factorialy (see [5, 6]).

In [7] we have provided detailed description of the Cyclic-Order Graph Algorithm, which can help people trying to make proofs in the field of Graph Theory by determining the number of permutations whose distance from two given permutations is lower than some required number (and identify those permutations). As a side product this algorithm also gives user information about the maximal distance in some given graph. During testing this algorithm in MATLAB (GNU Octave), as a most important downside was slow execution, especially in the beginning when matrix of distances between individual permutations needed to be calculated. Therefore we have decided to implement this algorithm in C++ to speed up the calculation and also to allow the batch processing of request, which could eventually lead to automatic proofs of crossing number theorems or parts of them.

In Section 2 we briefly describe the details of implementation of the program together with necessary steps to execute it. Section 3 describes interactive mode when working with program. Section 4 introduces batch processing of data. In Section 5 we show some performance results when compared to the original MATLAB (GNU Octave) implementation and outline possibilities for the future development of this program.

2. Program Overview

Whole C++ implementation of the Cyclic-Order Graph Algorithm (short COGA or `coga` as a program name) closely follows [7]. Starting from `main.cpp`, command line parameters are parsed first and it is decided, whether to provide user with simple help or create demo batch program (both leading to termination of the

application) or to start the `coga` itself in manual (see Section 3) or batch/automatic (see Section 4) mode.

To speed-up and save memory several approaches have been used. Most significantly we save list of possible cyclic permutations in files called `InputPermutations-`n`.txt`, n being the size of the problem. If not provided, the file is created. This allows for faster program start-up and also to ensure, that results obtained between launches of `coga` will remain consistent. User also has possibility to save the matrices of adjacency (`AdjMat-`n`.txt`) and distances (`DistMat-`n`.txt`), but this is not done automatically as the size of these matrices grows very fast with increasing value of n. Also we have used bit-set arrays to store adjacency matrix and for calculation of distances between permutations. This led to faster calculation time and 8-fold decrease of memory used for storing these large matrices. Also using `dynamic_bitset` we were able to allow user to select size of problem interactively on the run-time and we were not forced to set fixed size of problem on the compile-time.

To make the compilation of the program easier and to allow for easy addition of tools, s.a. MPI or CUDA later we have decided to use CMake tool-chain. Prerequisites: C++ compiler supporting C++11, CMake (minimal version 2.8.12), and part of the boost library (`trim.hpp`, `trim_all.hpp`, `dynamic_bitset.hpp`). All these have to be installed and paths to them have to be set.

Following commands have been tested on Fedora 27 based platform but should work without problem on MacOS or Windows based configurations too. Maybe steps needed to configure CMake will differ. Unpack directory with source files. Enter the directory created and create inside the directory `build` and enter it. Write the command:

```
cmake -DCMAKE_INSTALL_PREFIX=../run -DCMAKE_BUILD_TYPE=Release ..
```

This will create directory `run` one level higher (same level as directory `build`) to which programs executable will be copied after successful compilation with compilation flags set to optimize for speed (release version of program). You can modify this path to suit your needs. Now execute command: `make install`. This creates executable `coga` and copies it to the directory `../run/`. Now enter the newly created directory `run/` (starting from the level where you have created the `build` directory) and you can launch the program. There are 4 different ways to launch the program depending on the required outcome:

1. `coga` — starts manual (interactive) mode, described in Section 3.

2. `coga --help` or `-h/-help` — displays help message and terminates.

3. `coga fileName1 fileName2 ...` — starts batch mode, where `fileName*` are file names of files describing what the program has to do. All files provided from command-line are parsed in order they were provided on the command-line. Usage of this mode is described in Section 4.

4. `coga -demo` — produces example (demo) file for batch mode (Section 4) called `batchDemo.txt` and terminates.

3. Manual Mode

Manual mode is implemented in `RunProgramInteractively.cpp`. User is first asked to provide size n of the problem he wants to work with. This size has to be inside the bounds (inclusively) `nLower` and `nUpper` defined in `ListOfFunctions.hpp`, currently 3 and 9. After this step, file with all possible cyclic permutations called `InputPermutations-`n`.txt` is loaded (or created if missing), adjacency (neighbors) matrix is created and matrix of distances between each permutations is calculated (or loaded if any of matrices is provided). User is then brought to interactive menu, from which he can review all this data — list of cyclic permutations, adjacency matrix, distances matrix, identify all permutations with given maximal distance from two given permutations and save requested information to output files. If user wants to change size n, he needs to terminate the application and start again.

4. Batch Mode

Batch mode implemented in `RunProgramInBatch.cpp` is intended for automatic obtaining of larger quantity of information and for simplifying of repetitive tasks. Program reads files whose names were provided on the command line when starting program `coga` line by line and executes commands from these lines. Each line may contain only one command, starting from the first column of the line. Spaces in the beginning of the line are not allowed. Lines starting with the sign `#` and lines not starting with command are considered comments. In the case, user provided invalid command, either with wrong arguments or in incorrect order (e.g. asking to print data when no problem size was set), warning is issued with the file name of batch file and line number on which the error occurred. To obtain demo batch file, one needs to issue on the command-line command: `coga -demo`. List of currently implemented commands is in the Table 1.

5. Performance and Conclusions

The first implementation of the algorithm using MATLAB (GNU Octave) seemed sufficient for our needs and it worked fast enough for small dimensions of problem ($n = 3, 4, 5$). With growing n the calculation time (calculation of adjacency matrix and matrix of distances) increased rapidly. We were able to speed-up the calculation of matrix of distances (containing matrix multiplication) by using bit arrays instead, which helped little but the calculation was still slow. Therefore we decided to switch to C++. Comparison of calculation times depending on the size n of the problem and on the implementation itself is in the Table 2.

Table 1: List of commands implemented in the `coga` for batch processing.

`0 probSize`	Set size of problem (n) to value given by `probSize`. It has to be inside the given boundaries.
`1 fileName`	Set name of output file to `fileName` and open it. If any other file was already opened, it will be closed before this command is executed. Output of all subsequent commands will be written to this file, except for the adjacency matrix (written to the file `AdjMat-`n`.txt`) and matrix of distances (written to the file `DistMat-`n`.txt`). The content will be appended to the file, if such file already exists.
`2 fileName`	Same as in command '`1 fileName`' with the exception, that file will be opened as new (older information will be overwritten).
`3`	Write adjacency matrix to file. Each row of the file holds one row of the lower sub-diagonal triangle of adjacency matrix.
`4`	Write maximum distance to file.
`5`	Write matrix of distances to file. Each row represents one row of the lower sub-diagonal triangle of distances matrix.
`6 p1 p2 DIST`	Writes to file indices of all permutations whose distance from permutations `p1` and `p2` is less than or equal to `DIST`.
`7 p1 p2 DIST`	Same as `6 p1 p2 DIST` but with more detailed information. Not only indices of permutations are written to output but also the permutations themselves are printed.
`8 text`	`text` is written to the output file. All spaces before and after `text` are removed, newline character at the end of line is added automatically.

Table 2: Comparison of calculation times and complexity for different implementations of cyclic permutation algorithm (multiple runs averaged).

		matrix of adjacency				matrix of distances			
n	perm.	comp.	MAT	C++	lda	mul.	MAT	C++	ldd
3	2	15	0.002	0.00008	0.000099	1	0.001	0.00003	0.000053
4	6	276	0.011	0.00010	0.000112	2	0.003	0.00004	0.000068
5	24	7 140	0.165	0.00032	0.000145	4	0.066	0.00016	0.000090
6	120	258 840	5.688	0.00521	0.001138	6	2.730	0.00173	0.000972
7	720	12 698 280	302.832	0.08817	0.013004	9	166.168	0.08733	0.009630
8	5040	812 831 040	20 988.000	5.46466	0.438866	12	18 512.000	13.59132	0.434627

n – problem size
perm. – number of cyclic permutations for given problem size is $(n-1)!$
comp. – number of permutations comparisons during creation of adjacency matrix
MAT – adjacency or distances matrix creation time (in seconds) in MATLAB
C++ – adjacency or distances matrix creation time (in seconds) in C++
lda – time of adjacency matrix load from file in C++ (in seconds)
mul – number of $(n-1)! \times (n-1)!$ matrices squarings needed to calculate distances
ldd – time of distances matrix load from file in C++ (in seconds)

One can see, that the order of speed-up when comparing C++ and MATLAB (GNU Octave) implementation is from about 1000 up to 3800, depending on the size of the problem. Obviously, even better results are obtained when using adjacency and distances matrices stored on the computer (skipping the calculations) but this comes at the cost of large files stored ($2\times$24MB for $n = 8$ and about $2\times$1.8GB for $n = 9$).

For now we consider $n = 9$ limit for C++ as the computational time and memory necessary to store all information grow fast. Calculation of the adjacency matrix takes about 550 s and matrix of distances needs 16 multiplications and 9742 s on our testing configuration using Intel Core i7-6700 CPU running at 3.4GHz. Load of saved configurations takes 28 s in the case of adjacency matrix and 40 s for matrix of distances. With such large files, data access starts to play role and we were using only traditional rotational HDD. In the future we would like to implement improved version of the program which would use some of the parallel techniques (MPI and/or CUDA or OpenCL) to speed-up the calculation even further and we would like to store some information in local files in some kind of compressed form to be able to work with larger number of permutations.

The Cyclic-Order Graph Algorithm and the presented software (in C++ and MATLAB) can be seen used in the following articles [8–11]. In these articles, the authors use the software to prove theorems about crossing numbers of graphs.

Using the presented software, they generated the distances between appropriate permutations (and inverse permutations), from which they created tables for the lower bounds of the crossing numbers of graphs. These tables form the basis for further progress on the proofs of the theorems in mentioned articles.

The software we presented in this article can be downloaded from the web site of the Department of Mathematics and Theoretical Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice can be downloaded from the following link:

http://web.tuke.sk/fei-km/en/coga

This website also describes how to use and install this software in the Slovak and English languages.

References

[1] Adam Gh. et al *IT-ecosystem of the HybriLIT heterogeneous platform for high-performance computing and training of IT-specialists*. In International Conference "Distributed Computing and Grid-technologies in Science and Education" 2018, Edited by Korenkov V., Nechaevskiy A., Zaikina T., and Mazhitova E., CEUR Workshop Proceedings, Volume **2267** (2018), pp. 638–644

[2] Kleitman. D.J. *The crossing number of* $K_{5,n}$. Journal of Combinatorial Theory 1971, **9**, pp. 315–323

[3] Woodall D.R.. *Cyclic-order graphs and Zarankiewicz's crossing-number conjecture* Journal of Graph Theory 1993, **17** (6), pp. 657–671

[4] Zarankiewicz K. *On a problem of P. Turán concerning graphs* Fundamenta Mathematicae 1954, **41**, pp. 137–145

[5] Li B., Wang J., and Huang Y. *On the crossing number of the join of some* 5*-vertex graphs and* P_n, International Journal of Mathematical Combinatorics 2008, **2**, pp. 70–77

[6] Hernández-Vélez C., Medina C., and Salazar G. *The optimal drawings of* $K_{5,n}$. The Electronic Journal of Combinatorics 2014, **21** (4), 29., Paper 4.1.

[7] Berežný Š., Buša J. Jr., and Staš M. *Software solution of the algorithm of the cyclic-order graph.* Acta Electrotechnica et Informatica 2018, **18** (1), pp. 3–10

[8] Berežný Š. and Staš M. *On the crossing number of the join of five vertex graph* G *with the discrete graph* D_n. Acta Electrotechnica et Informatica 2017, **17** (3), pp. 27–32

[9] Berežný Š. and Staš M. *Cyclic permutations and crossing numbers of join products of symmetric graph of order six.* Carpathian Journal of Mathematics 2018, **34** (2), pp. 143–155

[10] Staš M. *On the crossing number of the join of the discrete graph with one graph of order five*, Mathematical Modelling and Geometry 2017, **5** (2), pp. 12–19

[11] Staš M. *Cyclic permutations: Crossing numbers of the join products of graphs*, In 17$^{\text{th}}$ Conference on Applied Mathematics (Aplimat 2018), Proceedings, pp. 979–987