

Researcher-friendly Java framework for testing conjectures in chemical graph theory[☆]

Mohammad Ghebleh^{*}, Ali Kanso, Dragan Stevanović

Department of Mathematics, Faculty of Science, Kuwait University, Safat 13060, Kuwait
mghebleh@gmail.com, akanso@hotmail.com, dragan_stevanovic@mi.sanu.ac.rs

(Received June 8, 2018)

Abstract

An important part of a chemical graph theorist’s research work is concerned with making observations and developing intuition about a particular research problem through extensive numerical testing. Research questions in chemical graph theory are often restricted to specific graph classes in which one is either looking for extremal values and extremal graphs of graph invariants, or graphs satisfying certain constraints, or inequalities between different invariants. Many graphs from such classes can nowadays be easily generated or readily downloaded from the web in nauty’s graph6 format. We describe here a Java framework for answering the above research questions among sets of graphs given in graph6 format, which represents unification of testing programs that we had used over the years. The framework consists of templates that can be easily customised so that the researcher’s initial work should reduce just to rephrasing a question in hand within a specific template. This way one can quickly prepare numerical calculations to be performed over large sets of graphs and shift focus to more creative research work instead. The use of templates is described in detail and illustrated on several conjectures from chemical graph theory.

[☆]This work was supported and funded by Kuwait University Research Grant No. SM03/17.

^{*}Corresponding author

1 Introduction

Computers are undisputably a useful tool in mathematical research as they enable complex calculations to be performed in a short period of time. For graph theory in particular, there is a long history of software packages aimed to help researchers in their studies. Certainly one of the oldest such standalone programs is GRAPH, written by Dragoš Cvetković and Laszlo Kraus from 1980–1984 [1–3], which provided closed environment for visually editing graphs and calculating their invariants, and even implemented certain artificial intelligence methods for automatic proving of theorems in graph theory. Another early example is Graffiti, written by Siemion Fajtlowicz in 1986 [4–8], which was geared more toward automatic conjecture making than enabling researcher to test his own conjectures. A slightly more recent example is AutoGraphiX, written by Gilles Caporossi and Pierre Hansen in 2000 [9–12], which considered inequalities among graph invariants as instances of optimisation problems and then applied variable neighborhood search metaheuristic to search for optimal, i.e., extremal graphs. Further examples of standalone programs supporting graph theory research include newGRAPH [13–15], which represents a modernised version of GRAPH, and Grinvin [16, 17], which represents a modernised version of Graffiti. Besides these programs, there are also several libraries of data structures and methods that may be used to write programs working with graphs in well-known programming languages, such as Leda for C++ [18, 19], Combinatorica for Mathematica [20], GraphTheory for Maple [21] or NetworkX and MathChem for Python [22–25].

However, despite all this available software it happens often enough in chemical graph theory literature that conjectures are published without being thoroughly tested (a few such conjectures may be found, for example, in [26–28] for which counterexamples are exhibited in [29–31]). A possible explanation for failing to thoroughly test a conjecture prior to its publication may lie in the fact that most of the above mentioned packages do require to invest considerable amount of time in learning how to use them.

Graph theorists usually study properties of graphs from certain classes. Brendan McKay’s package nauty [32, 33] provides a generally accepted way of generating sets of smaller graphs constrained for connectedness, number of vertices or ranges of the numbers of edges or vertex degrees. Programs are available for generating other classes of graphs as well, most widely known of which are certainly Buckygen [34–36] and fullgen [37] for generating fullerenes, and plantri [37–40] for generating other planar graphs. Actually,

many sets of graphs are already generated and ready for download in nauty's graph6 format either from the House of Graphs [41] or the web pages of mathematicians like Brendan McKay [42] and Gordon Royle [43]. Research questions that are often studied on such sets of graphs involve calculation of invariants and selection of particular graphs or pair of graphs and may have some of the following forms:

- What are the values of certain invariants (*such as energy and nullity*) for graphs in a given set?
- Which graphs in a given set satisfy certain constraints (*such as having the Laplacian energy equal to that of a complete graph*)?
- Which graphs attain the maximum or the minimum value of a given invariant expression (*such as the distance-sum heterogeneity index*) under certain constraints (*such as fixed number of edges*)?
- Which pairs of graphs have the same value of an invariant expression (*such as equal energies*)?
- Which pairs of graphs have similar values of one invariant expression, but dissimilar values of another invariant expression (*such that the difference of their Wiener indices is smaller than the difference of their Randić indices*)?

To answer questions like these it is beneficial to be able to quickly set up numerical tests to be performed over sets of graphs. We describe here a Java framework that we have developed for this purpose. The framework is based on our existing experience with graph computations and it provides general templates for dealing with the above research questions. Templates are organised so as to enable graph theorists with little to no programming experience to easily adapt them to their own question variants, run them for a given set of graphs in graph6 format, visualise the selected graphs and, eventually, develop deeper intuition about the behavior of studied graph invariants.

Before starting with descriptions of framework and its use, let us briefly explain a few main reasons for our choice of Java over other programming languages: its speed, a large number of useful libraries and existence of a simple development environment. First, while Java may not be as fast as C or C++, its speed is still comparable to them, and on the other hand, it is substantially faster than interpreted languages such as Python,

Matlab (Octave) or Mathematica. Next, Java has a large number of useful libraries, some already included in its distribution and some freely downloadable from the internet. Its collection library, for example, enables to quickly sort graphs by invariant values or to use invariant values as keys in maps to instantly discover graphs with the same or similar invariant values, while the graph6 archive is being processed. The Colt library [44], on the other hand, offers data structures optimized for eigenvalue calculations with both dense and sparse matrices. Above all, Java offers a simple integrated development environment BlueJ [45], stripped of overwhelming user interfaces of professional development environments such as NetBeans or Eclipse, enabling its users to get started more quickly, as can be evidenced from its rather short manual [46]. BlueJ is specifically designed for teaching first programming course to undergraduate students (and non-computer science researchers as well), and is followed by an excellent introductory book to programming in Java [47], which is gladly recommended for further reading, as certain knowledge of Java programming is needed for more creative uses of the framework.

The paper is organised as follows. In the next section we describe the software and setup necessary to use the framework. Different parts of the framework and its templates are explained in detail in Section 3, while examples of its use on conjectures from the literature are given in Section 4.

2 Preliminary setup for using the framework

In order to use the framework as intended, installation of several pieces of software is necessary.

BlueJ, Colt library and the framework source files

BlueJ is needed to edit and run the framework source files. Download the appropriate installer from <https://www.bluej.org/> and install it. If you have downloaded BlueJ for Windows or Mac OS X, the installer already contains Java development kit (JDK), necessary for compilation of the framework source files. If you have downloaded BlueJ for another operating system, you need to ensure that you have JDK installed on your system as well. You may quickly check this by opening up a terminal window on your system and typing `javac` in it. If the terminal window replies with a lengthy help message on how to use `javac`, you have Java compiler installed. Otherwise, download and install JDK from

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

The framework files may be downloaded as a zip archive from

<https://doi.org/10.5281/zenodo.1244001>

When unzipped, you get a folder with the framework source files and a copy of `colt.jar`, a Java library for eigenvalue calculations. BlueJ has to be instructed to load `colt.jar`, which is done by selecting **Preferences** command from either **Tools** menu under Windows or **BlueJ** menu under OS X, selecting **Libraries** tab in the newly opened window, clicking on **Add File** and choosing `colt.jar` from the standard file open dialog.

Since the framework consists of source files that need to be edited for each conjecture separately, it may be a good idea to leave the folder with original source files intact and make a new copy of that folder for each new conjecture to be tested. Once that is done, you may open a copy of the framework in BlueJ by selecting **Open Project** command from **Project** menu and then opening the folder containing the framework copy from the standard file open dialog (to avoid confusion, note that you have to select and open the actual folder and not any individual file).

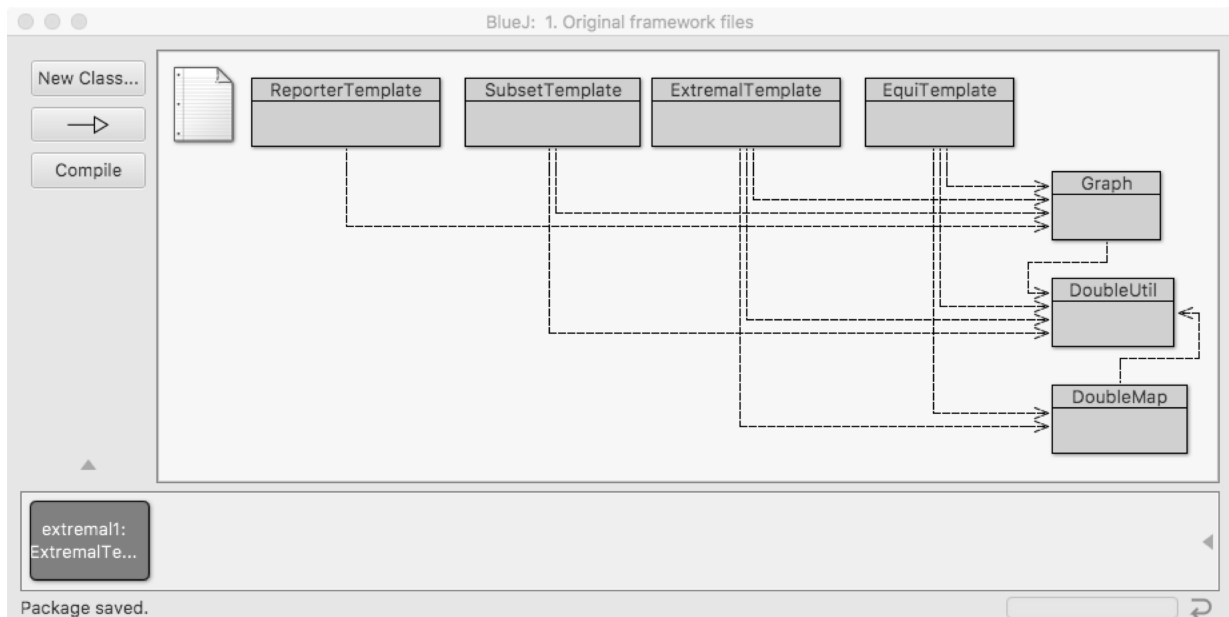


Figure 1: BlueJ window: Java classes containing program code are shown in the main part, with arrows depicting dependence of one class on another class. The compile button is situated on the left side, while the object bench is placed at the bottom of the window.

The main part of the BlueJ window contains classes of the current project, represented by named rectangles, which can be opened for editing by double-clicking. When-

ever source code has been edited, it needs to be recompiled by clicking **Compile** button (see Fig. 1). Afterwards, you can right-click on a class rectangle and choose **new <Class_name>()** to create a new object of a given class, which will then be shown as a rounded rectangle at the bottom of the window, in the so-called object bench. Right-clicking an object in the object bench gives the option to call object methods, including `void run(String inputFileName, ...)`, the main method in each framework template. BlueJ then opens a dialog, as shown in Fig. 2, that asks for values of the method arguments, where `inputFileName` denotes the file with a set of graphs in graph6 format. Note that `inputFileName`, as a Java string, has to be entered with enclosing double quotation marks, and that the program expects the file to be found in the framework directory (unless the full path to it is typed in `inputFileName` as well).

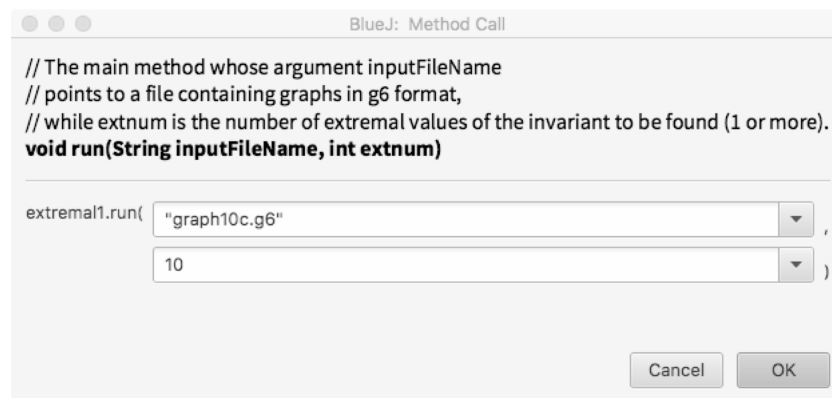


Figure 2: Entering arguments for an object method.

Sets of graphs in graph6 format and nauty

Graph6, devised by Brendan McKay, is a format for describing graphs that originated in pre-WWW times when data had to be written using printable ASCII characters in order to be sent efficiently through e-mail. Basically, assuming that you deal with a simple graph, it starts with the upper half of its adjacency matrix (without a zero diagonal), lists the columns consecutively to obtain an array of bits and then divides this array into chunks of 6 bits each (hence the name graph6). The 6-bit numbers obtained in this way are added to 63 in order to produce visible ASCII characters (going from ? to ~ through capital and small letters). The resulting file then consists of lines, one for each graph, where the first character represents the number of vertices, also added to 63, and the remaining characters in the line encode the adjacency matrix as described.

Quite a few sets of graphs in graph6 format are available online from several reliable web pages:

- Brendan McKay at <http://cs.anu.edu.au/~bdm/data/graphs.html> has posted, among others, sets of small simple, Eulerian and planar graphs;
- Gordon Royle at <http://www.maths.uwa.edu.au/~gordon/data.html> has posted, among others, sets of small trees, bipartite graphs, cubic graphs;
- the House of Graphs [41] at <https://hog.grinvin.org/MetaDirectory.action> has a metadirectory with access to several further sets of graphs.

In cases when you need a set of graphs that is not available online, new sets can be created using `geng` or `genbg` tools from `nauty` package. It is foreseen that the source code of `nauty` is downloaded from <http://pallini.di.uniroma1.it/> and compiled locally. This is usually not an issue for non-Windows users—just use short instructions provided at this site. If you happen to work on a Windows machine, a combination of Code::Blocks IDE (<http://www.codeblocks.org/>) and GCC compiler such as mingw-w64 (<http://mingw-w64.org/doku.php/download>) should get you started.

`nauty` tools are used from the command line. General format for `geng` command is

```
geng [-options] n [mine[:maxe]] [file]
```

where brackets `[]` denote optional arguments, `n` denotes the number of vertices, `mine` and `maxe` the minimum and maximum number of edges, while `file` denotes the name of the output file. The most often used options are `-c` to generate connected graphs, `-d#` for the minimum vertex degree, where `#` denotes a number, and `-D#` for the maximum vertex degree. Here are a few examples:

```
geng -c 9 graph9c.g6      to generate connected graphs on 9 vertices
```

```
geng -c 19 18:18 trees19.g6  to generate trees on 19 vertices
```

```
geng -cd3D3 16 cubic16.g6    for connected cubic graphs on 16 vertices
```

```
geng -cD4 11 chem11.g6      for connected chemical graphs on 11 vertices
```

See `geng -help` for list of other options. `genbg` is used similarly as it can be evidenced from `genbg -help`.

You can also automate generation of graph sets. For example, if you wish to generate connected 10-vertex graphs classified in files by their number of edges, you may use

```
for i in {9..45}; do geng -c 10 ${i}:${i} graph10e${i}.g6; done
```

in Unix-based terminal (Mac OS X, Linux), and

```
for /L %i in (9,1,45) do geng -c 10 %i:%i graph10e%i.g6
```

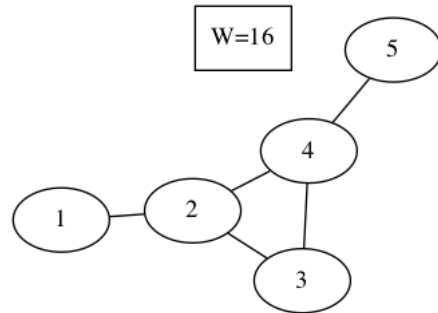
in Windows command line.

Visualisation of graphs with Graphviz

Graphviz is a well developed software package for visualisation of graphs. It can be downloaded from www.graphviz.org and, after successful installation, it offers a number of command-line tools that produce an image of a graph from a text file with its description. Such description mainly consists of the list of edges, with various options to additionally describe visual properties of vertices and edges. An example of a description of a small 5-vertex graph is shown in Fig. 3(a), which uses a small trick to force Graphviz to show further information about the graph (in this case its Wiener index) within the resulting image by specifying an isolated vertex with that information as a label.

```
Graph {
  1 -- 2
  2 -- 3
  2 -- 4
  3 -- 4
  4 -- 5
  v [shape=box, label="W=16"]
}
```

(a)



(b)

Figure 3: (a) Textual representation of a graph in the Graphviz format. (b) Visual representation of the same graph produced by `neato`.

Graphviz tools implement several well known graph drawing algorithms. For small undirected graphs perhaps the most useful among them is `neato`, based on minimisation of energy of the graph spring model (see [48]). General format of `neato` command is

```
neato [-options] dotfile > outputfile
```

where `dotfile` denotes the text file containing description of a graph (usually with extension `.dot`), and `outputfile` denotes the resulting image file. Among numerous options the most useful appear to be `-Goverlap=false`, which implies that vertices should not

overlap each other, `-Gsplines=true`, which allows curved edges and `-Tpng`, `-Tpdf`, `-Tgif` or `-Tjpg`, which specify the format of the output image file. For detailed description of other available options of `neato` and further tools contained in Graphviz, the reader is referred to www.graphviz.org/documentation/.

Calls to `neato` can be automated using `for` command. For example, to generate `png` images for all graph descriptions in a current directory, you may use (typing in a single line)

```
for file in *.dot; do neato -Goverlap=false -Gsplines=true  
-Tpng "${file}" > "${file}.png"; done
```

in Unix-based terminal and

```
for %f in (*.dot) do neato -Goverlap=false -Gsplines=true  
-Tpng %f > %f.png
```

in Windows command line.

3 Framework description

There are five main classes in our framework: `Graph` class contains methods to construct adjacency matrix and calculate invariants, while the classes `ReporterTemplate`, `SubsetTemplate`, `ExtremalTemplate` and `EquiTemplate` contain worked out examples that, respectively, report invariant values, find a subset of graphs, find extremal graphs and find pairs of graphs with (approximately) the same invariant values. Our aim was that the templates need minimal changes in order to adapt to the researcher's particular need. Structure and methods of these classes are explained in subsequent sections, and the interested reader is advised to read their actual Java code in parallel.

3.1 Graph class

`Graph` class starts with the main constructor `public Graph(String s)` that creates a `Graph` object from its description in graph6 format. Provided the graph6 code is contained in `String g6code`, the corresponding `Graph` object may be constructed by the command

```
g = new Graph(g6code);
```

Table 1: List of methods currently implemented in `Graph` class. In the method call column, it is assumed that `g` and `h` represent constructed `Graph` objects.

Method call	Return type	Description
<code>g.n()</code>	<code>int</code>	Number of vertices
<code>g.m()</code>	<code>int</code>	Number of edges
<code>g.degrees()</code>	<code>int []</code>	Array of vertex degrees
<code>g.Amatrix()</code> , <code>g.Lmatrix()</code> , <code>g.Qmatrix()</code> , <code>g.Dmatrix()</code> , <code>g.Mmatrix()</code>	<code>int [] []</code>	Adjacency, Laplacian, signless Laplacian, distance and modularity matrix
<code>g.Aspectrum()</code> , <code>g.Lspectrum()</code> , <code>g.Qspectrum()</code> , <code>g.Dspectrum()</code> , <code>g.Mspectrum()</code>	<code>double []</code>	Spectrum of adjacency, Laplacian, signless Laplacian, distance and modularity matrix
<code>g.Aeigenvectors()</code> , <code>g.Leigenvectors()</code> , <code>g.Qeigenvectors()</code> , <code>g.Deigenvectors()</code> , <code>g.Meigenvectors()</code>	<code>double [] []</code>	Eigenvectors of adjacency, Laplacian, signless Laplacian, distance and modularity matrix
<code>g.Acospectral(h)</code> , <code>g.Lcospectral(h)</code> , <code>g.Qcospectral(h)</code> , <code>g.Dcospectral(h)</code> , <code>g.Mcospectral(h)</code>	<code>boolean</code>	Checks A-cospectrality, L-cospectrality, Q-cospectrality, D-cospectrality and M-cospectrality of <code>g</code> and <code>h</code>
<code>g.Aintegral()</code> , <code>g.Lintegral()</code> , <code>g.Qintegral()</code> , <code>g.Dintegral()</code> , <code>g.Mintegral()</code>	<code>boolean</code>	Checks whether A-spectrum, L-spectrum, Q-spectrum, D-spectrum and M-spectrum consists of integers
<code>g.Aenergy()</code> , <code>g.Lenergy()</code> , <code>g.Qenergy()</code> , <code>g.Denergy()</code> , <code>g.Menergy()</code>	<code>double</code>	Energy of adjacency, Laplacian, signless Laplacian, distance and modularity matrix
<code>g.LEL()</code>	<code>double</code>	Laplacian-like energy
<code>g.estrada()</code>	<code>double</code>	Estrada index
<code>g.Lestrada()</code>	<code>double</code>	Laplacian Estrada index
<code>g.diameter()</code>	<code>int</code>	Diameter
<code>g.radius()</code>	<code>int</code>	Radius
<code>g.Wiener()</code>	<code>int</code>	Wiener index
<code>g.randic()</code>	<code>double</code>	Randić index
<code>g.zagreb1()</code>	<code>int</code>	The first Zagreb index
<code>g.zagreb2()</code>	<code>int</code>	The second Zagreb index
<code>g.dshi()</code>	<code>double</code>	Distance-sum heterogeneity index [49]
<code>g.printAmatrix()</code>	<code>String</code>	String representing adjacency matrix
<code>g.printLmatrix()</code>	<code>String</code>	String with Laplacian matrix
<code>g.printQmatrix()</code>	<code>String</code>	String with signless Laplacian matrix
<code>g.printDmatrix()</code>	<code>String</code>	String with distance matrix
<code>g.printMmatrix()</code>	<code>String</code>	String with modularity matrix
<code>g.printEdgeList()</code>	<code>String</code>	String representing edge list
<code>g.printDotFormat()</code>	<code>String</code>	Graph description in dot format
<code>g.printDotFormat(data)</code>	<code>String</code>	Dot format description with added isolated vertex showing String data
<code>g.saveDotFormat(filename)</code>	<code>none</code>	Saves graph description in dot format to the named file for later visualization
<code>g.saveDotFormat(filename, data)</code>	<code>none</code>	Saves dot format description to file with added isolated vertex showing data

The constructor also populates the degree sequence and the numbers of vertices and edges, while the user has to call separate methods to calculate values of other invariants.

The class contains one more constructor `public Graph(int A[] [])` that creates a `Graph` object from the supplied adjacency matrix. This constructor may be used, for

example, if one needs to create complement of a graph or a result of another graph operation: the original graph is created from its graph6 code by the first constructor, an adjacency matrix **A** of the new graph is calculated by user code and the new **Graph** object is then constructed by `h = new Graph(A);`

Remaining methods in this class, listed in Table 1, calculate various invariants of a graph, with a good deal of them representing its spectral properties. **Graph** class also contains several auxiliary static methods (which are called with `Graph.method()` instead of `g.method()`) for calculating spectra and eigenvectors of integer and real-valued (double) matrices, checking that a matrix has integral spectrum, calculating deviation of array entries and matrix energy, etc., which may be helpful to researchers who add new invariants to this class. These are recognized in the source code by the keyword `static`.

Note that calculations of spectral properties depend on numerical routines which, in general, return approximate results. When checking mutual equality of such quantities (as in `g.Acospectral(h)` or `g.Aintegral()`), one has to allow a certain degree of freedom by checking that, actually, absolute value of the difference of two quantities is sufficiently small. This is enabled by methods implemented in classes **DoubleUtil** and **DoubleMap**. This means that the use of approximate results may also return either false positives or false negatives. While we have not yet come at an example of a false negative, false positives do appear from time to time, so that the examples obtained with the use of this framework should be checked with a symbolic computation software (such as Mathematica, Maple or Sage) prior to publication.

3.2 ReporterTemplate class

Class **ReporterTemplate** simply serves to list values of selected invariants for all graphs in a given set. Its main method is `run(String inputFileName, int createDotFiles)`, where `inputFileName` contains the name of the file (i.e., the path to the file) with a set of graphs in graph6 format, and `createDotFiles` is a flag that signals whether the method should also output dot files for each graph in the set: it should be set to nonzero to output dot files, and to zero otherwise. Beware that setting this flag to nonzero for a set with a large number of graphs will create that many dot files in the folder and may significantly slow down the operating system until the method finishes its work. Pseudo-code of the `run` method is shown in Algorithm 1.

Algorithm 1 run method of ReporterTemplate class

```
1: procedure run(inputFileName, createDotFiles)
2:   Open inputFileName for reading
3:   Open new file named inputFileName+".results.csv" for writing
4:   while line with g6code read from inputFileName is not empty do
5:     Construct Graph g from its g6code
6:     Calculate necessary invariants of g
7:     Output g6code and invariant values to inputFileName+".results.csv"
8:     if createDotFiles  $\neq$  0 then
9:       Save dot format of g to a separate file
10:  Close input and output files
```

When you download the framework from <https://doi.org/10.5281/zenodo.1244001>, the `run` method of `ReporterTemplate` class is set to report values of energy and nullity for graphs in the set. To report other invariants, one needs to customize parts of the `run` method corresponding to steps 6 and 7 in Algorithm 1. These steps correspond to the following snippet in the source code:

```
// Calculate necessary invariants here:
double energy = g.energy();

double[] eigs = g.Aspectrum();
int nullity = 0;
for (int i=0; i<g.n(); i++)
    if (DoubleUtil.equals(eigs[i], 0.0))
        nullity++;

// Output g6code and invariant values here:
outResults.println(g6code + ", " + energy + ", " + nullity);
```

Let us briefly explain this code snippet. First, each variable in a Java program must be defined with its type when used for the first time: `int` is needed to define the variable `nullity` that will keep the value of nullity (initially set to 0), and similarly, `double` is needed to define `energy` and `double[]` is needed for eigenvalues `eigs`. (Return types of `Graph` methods are listed in Table 1.) However, type is not needed when using the variables afterwards: thus we write just `nullity++` (which increases the value of `nullity` by 1), and not `int nullity++`. This snippet also illustrates the use of the static `equals` method

in `DoubleUtil` class: we will increase nullity whenever we come across an eigenvalue that is close to 0, which is checked by the command `DoubleUtil.equals(eigs[i], 0.0)`. To output values of invariants, one needs to print a line (`println`) to the output file which is kept in the object `outResults` (hence `outResults.println(string)`). The string to be output is created with the string concatenation operator `+`: if the first argument of `+` is a string (and `g6code` is), then all the remaining arguments will be treated as strings as well. Hence the result of `g6code + ", " + energy + ", " + nullity` will be a comma-separated string containing the values of graph's `g6code`, `energy` and `nullity`, that is written in the output file.

Consult source code of the `ReporterTemplate` class for implementation of the remaining steps of Algorithm 1.

3.3 SubsetTemplate class

Class `SubsetTemplate` serves to select a subset of graphs in a given set which satisfy a given condition and output the subset and further data to a new file. Its main method is `run(String inputFileName, int createDotFiles)` where `inputFileName` gives the name of the graph6 file with the set of graphs and `createDotFiles` is a zero-nonzero flag of whether the method should also output dot files for each graph that satisfies the condition (nonzero to output dot files, and zero otherwise). As in the case of `ReporterTemplate` class, nonzero value of `createDotFiles` should only be used if you expect a handful of graphs in the subset (and not thousands). Pseudo-code of the `run` method is shown in Algorithm 2.

Algorithm 2 run method of `SubsetTemplate` class

```

1: procedure run(inputFileName, createDotFiles)
2:   Open inputFileName for reading
3:   Open new file named inputFileName+".results.tex" for writing
4:   while line with g6code read from inputFileName is not empty do
5:     Construct Graph g from its g6code
6:     Calculate necessary invariants of g
7:     Check whether the given condition holds for g
8:     if condition holds then
9:       Output g6code and invariant values to inputFileName+".results.tex"
10:      if createDotFiles  $\neq$  0 then
11:        Save dot format of g to a separate file
12:   Close input and output files

```

The `run` method in the downloaded framework files is set to select integral graphs from a set of graphs. To change it to select different types of graphs, one needs to update parts of the `run` method corresponding to steps 6 and 7 of Algorithm 2. These steps initially correspond to the following code snippet:

```
// Calculate necessary invariants here:
double[] eigs = g.Aspectrum();

// Write a criterion to select a graph into the subset here:
int integral = 1;
for (int i=0; i<g.n(); i++)
    if (!DoubleUtil.equals(eigs[i], Math.round(eigs[i]))) {
        integral = 0;
        break;
    }

// Output selected graphs and other data to the output file here:
if (integral==1) {
    ...
}
```

After obtaining a copy of adjacency eigenvalues of `g` in `double[] eigs`, the code goes on to check their integrality. The variable `int integral` serves as a flag here: it is initially set to 1, and becomes 0 if there is an eigenvalue that is not sufficiently close (`!DoubleUtil.equals()`, where `!` denotes logical negation) to its nearest integer, as returned by `Math.round(eigs[i])`. In such case there is no need to check the remaining eigenvalues, so that the program interrupts the current loop with `break` and proceeds further with execution. Finally, output is produced if the flag `integral` had remained equal to 1 after the `for` loop. Note that `Graph` class already contains method `Aintegral()`, so that the whole previous code snippet can be replaced simply with

```
if (g.Aintegral()) {
    ...
}
```

Nevertheless, we have left it in the form above due to its instructiveness.

The reader may also consult source code of the `SubsetTemplate` class for example of translating an array of eigenvalues into a string for its addition to the dot file.

3.4 ExtremalTemplate class

Class `ExtremalTemplate` serves to select a given number of extremal values (either minimal or maximal) of a given invariant and to also report all graphs in the set with those invariant values. Its main method is `run(String inputFileName, int extnum, int lookformax)`, where `inputFileName` specifies the graph6 set of graphs, `extnum` gives the number of extremal values to be reported and `lookformax` determines whether the method is to look for maximum values ($\text{lookformax} \geq 0$) or minimal values ($\text{lookformax} < 0$). Pseudo-code of the `run` method is given in Algorithm 3.

Algorithm 3 run method of `ExtremalTemplate` class

```

1: procedure run(inputFileName, extnum, lookformax)
2:   Open inputFileName for reading
3:   Open new file named inputFileName+".results.tex" for writing
4:   Construct an empty map
5:   while line with g6code read from inputFileName is not empty do
6:     Construct Graph g from its g6code
7:     Calculate necessary invariant of g and put it into key
8:     if map has less than extnum keys or map already contains this key then
9:       Put key and g6code into map
10:    else
11:      if lookformax<0 then
12:        if key is smaller than the largest key currently in map then
13:          Remove the largest key from map
14:          Put key and g6code into map
15:      else
16:        if key is larger than the smallest key currently in map then
17:          Remove the smallest key from map
18:          Put key and g6code into map
19:    for each key in map do
20:      Output key to inputFileName+".results.tex"
21:      for each g6code corresponding to key in map do
22:        Output g6code to inputFileName+".results.tex"
23:        Construct Graph g from its g6code
24:        Save dot format of g with key as data to a separate file
25:  Close input and output files

```

This `run` method is slightly more complicated than `run` methods in previous two classes due to necessity to keep track of a dynamically changing map of keys and corresponding

g6code strings. This functionality is provided by auxiliary class `DoubleMap`, extended upon the standard class `TreeMap`, which enables one to identify keys that are sufficiently close to each other (i.e., that differ by less than `DoubleUtil.DOUBLE_EQUALITY_THRESHOLD` in absolute value, which is set to 10^{-8} in the framework files). As before, caution must be taken as this may either wrongly identify truly different values or treat essentially equal values calculated with sufficiently large numerical errors as different (the latter case could be easily dealt with by increasing the value of `DoubleUtil.DOUBLE_EQUALITY_THRESHOLD`). Hence results should be checked independently with a symbolic computation package prior to publication. Nevertheless, the speed and simplicity with which initial results may be obtained in this way warrants usefulness of the framework.

It should be noted that the pairs kept in a `DoubleMap` object consist of a `Double` object `key` and a collection of strings (`Vector<String>`), each of which represents g6 code of a graph with that value of the key. The key is calculated in step 7 of Algorithm 3, and in the framework version this step corresponds to the line:

```
// Calculate necessary invariant here and make it the key:
key = new Double(g.dshi());
```

The invariant used here is the distance-sum heterogeneity index, defined by Estrada and Vargas-Estrada in [49] and implemented as a method of `Graph` class. Note that `g.dshi()` returns `double` value, an ordinary real number, while `key` is defined to be of type `Double`, which represents a Java object holding a `double` value inside itself. This inconsistency is a peculiarity of Java, as collections (such as maps) are meant to keep objects (such as `Double`) and not primitive number types (such as `double`). As a consequence, when changing the above code, one needs to pay attention that the `key` has to be constructed as a `Double` object from the provided value of the invariant: if the value is calculated as `val`, then the corresponding code will be `key = new Double(val)`; On the other hand, it does not matter if `val` is of type `double` or `int`—constructor of `Double` will correctly deal with both cases.

In addition, `run` method assumes that the number of extremal graphs found will be relatively small, so that at the end it outputs each extremal graph with key added as data to a separate dot file for later visualization with Graphviz. For easier identification of dot files, their names include number of vertices, value of the key and ordinal number of graph with that key, interspersed by user defined strings. The reader may further consult

source code for details of working with maps and collections and naming output files.

3.5 EquiTemplate class

Class `EquiTemplate` serves to find subsets of graphs having (approximately) equal values of a selected invariant in a given set of graphs. Its main method is `run(String inputFileName)`, where `inputFileName` specifies the set of graphs. Pseudo-code of the `run` method is shown in Algorithm 4.

Algorithm 4 `run` method of `EquiTemplate` class

```

1: procedure run(inputFileName)
2:   Open inputFileName for reading
3:   Open new file named inputFileName+".results.tex" for writing
4:   Construct an empty map
5:   while line with g6code read from inputFileName is not empty do
6:     Construct Graph g from its g6code
7:     Calculate necessary invariant of g and put it into key
8:     Put key and g6code into map
9:   for each key in map do
10:    if collection of strings corresponding to key has at least two entries then
11:      Output key to inputFileName+".results.tex"
12:      for each g6code corresponding to key in map do
13:        Output g6code to inputFileName+".results.tex"
14:        Construct Graph g from its g6code
15:        Save dot format of g with key as data to a separate file
16:   Close input and output files

```

This `run` method also relies on `DoubleMap` class for its operation. For each graph in a set it simply puts the `key` (=calculated invariant value) and `g6code` into `DoubleMap map`, while `DoubleMap` internally takes care of checking whether `map` already contains another key `key2` that is sufficiently close to the provided `key`, in which case `g6code` is added to the collection of strings classified under `key2` (otherwise, `key` is added as a new key in `map` with the corresponding collection consisting solely of `g6code`). After `map` is fully populated, it is enough to traverse it: all graphs that have sufficiently equal invariant values will be classified under the same key, so that one has to report each key whose collection contains at least two entries, together with the list of corresponding graph6 codes and dot files for visualization with Graphviz.

This simplicity, however, is hampered by the fact that the whole set of graphs together with keys has to be kept in internal memory. Although both graph6 codes and Double-valued keys should be rather small in size, it appears that Java virtual machine is too

generous in its memory management, so that we were not able to run this method on the set of 11,716,571 connected graphs on 10 vertices on computers available to us (although it works without problems on the set of 261,080 connected graphs on 9 vertices).

As in the case of `ExtremalTemplate` class, `key` is `Double` object constructed from a supplied integer (`int`) or float (`double`) value. For example, in the downloaded version of the framework

```
key = new Double(g.energy());
```

constructs `key` from the `double` value returned by `energy()` method of `Graph` class. As it is expected that the number of graphs sharing invariant values will be relatively small, `run` method for each `key` shared by at least two graphs output each graph (with `key` added as data) to a separate dot file for later visualization with Graphviz. Dot filenames include numbers of vertices, keys and ordinal numbers (within the group sharing the `key`) for easier identification. The reader is invited to consult source code for remaining implementation details.

4 Examples of use

In this section we showcase the use of the framework described in previous section on conjectures and results recently published in literature on mathematical chemistry. Each of the four classes: `ReporterTemplate`, `SubsetTemplate`, `ExtremalTemplate` and `EquiTemplate` is illustrated in a separate subsection.

4.1 Using ReporterTemplate

The `ReporterTemplate` is the most basic part of the proposed framework. For an input list of graphs and a selected list of graph invariants, this template generates a csv file of graph6 codes of graphs and their computed invariants. The resulting csv file may be further processed in a spreadsheet software, such as Excel or OpenOffice Calc, to generate charts and diagrams. For example, reporting invariant values of trees of order 12 was one of the steps necessary to obtain plots presented in Figs. 6 and 9 in forthcoming subsections.

For another example of this type of study, we may look into dependence of graph energy on nullity. The nullity of a graph G , denoted by $n_0(G)$, is the multiplicity of the eigenvalue 0 in its adjacency spectrum. The *energy* [50] of a graph G is defined as

$\mathcal{E}(G) = \sum_{i=1}^n |\lambda_i|$, where $\lambda_1, \dots, \lambda_n$ are adjacency eigenvalues of G . Quantum chemical arguments suggest that \mathcal{E} should be a decreasing function of n_0 [51–53]. However, from a mathematical point of view the naive statement

$$n_0(G) > n_0(H) \Rightarrow \mathcal{E}(G) < \mathcal{E}(H) \quad (1)$$

is not generally true for arbitrary graphs G and H , and as stated in [53], it must be assumed that except for nullity, all other structural features of the graphs G and H that influence their energy, are equal or differ negligibly. This vague description is quite difficult to quantify since the dependence of energy on graph structure is not yet completely understood.

In Fig. 4 we present two plots of energy vs. nullity for trees of order 20, and connected graphs of order 10 and size 20. In these plots, each vertical bar at a given value of n_0 represents the interval from the smallest to the largest value of graph energy. If the naive statement (1) would be true, we would expect the top of each bar to be lower than the bottom of the bar to its left. Of course this does not hold, but at the same time we can observe that the general behavior of bars is that as we move from left to right to increase nullity, the bars move lower, which gives some support for the conjectured behavior of \mathcal{E} with respect to n_0 .

4.2 Using SubsetTemplate

The `SubsetTemplate` is used to filter out a subset of graphs according to a given property. For illustrating the use of template, we will focus on *borderenergetic* graphs [54] and *L-borderenergetic* graphs [55].

4.2.1 Borderenergetic graphs

A non-complete graph G of order n is borderenergetic if $\mathcal{E}(G) = \mathcal{E}(K_n) = 2n - 2$. Borderenergetic graphs were introduced by Gong *et al.* [54], while further results on borderenergetic graphs can be found in [56–62]. It was shown in [54] that there are no borderenergetic graphs of order less than 7, and all such graphs of orders 7, 8 and 9 were listed. This list includes one borderenergetic graph of order 7, six of order 8, and seventeen of order 9. Further, Li *et al.* [56] reported the list of borderenergetic graphs of order 10 consisting of 47 graphs. This latter number was corrected in [57] to be 49. We tested these results to confirm the lists of borderenergetic graphs of order up to 10.

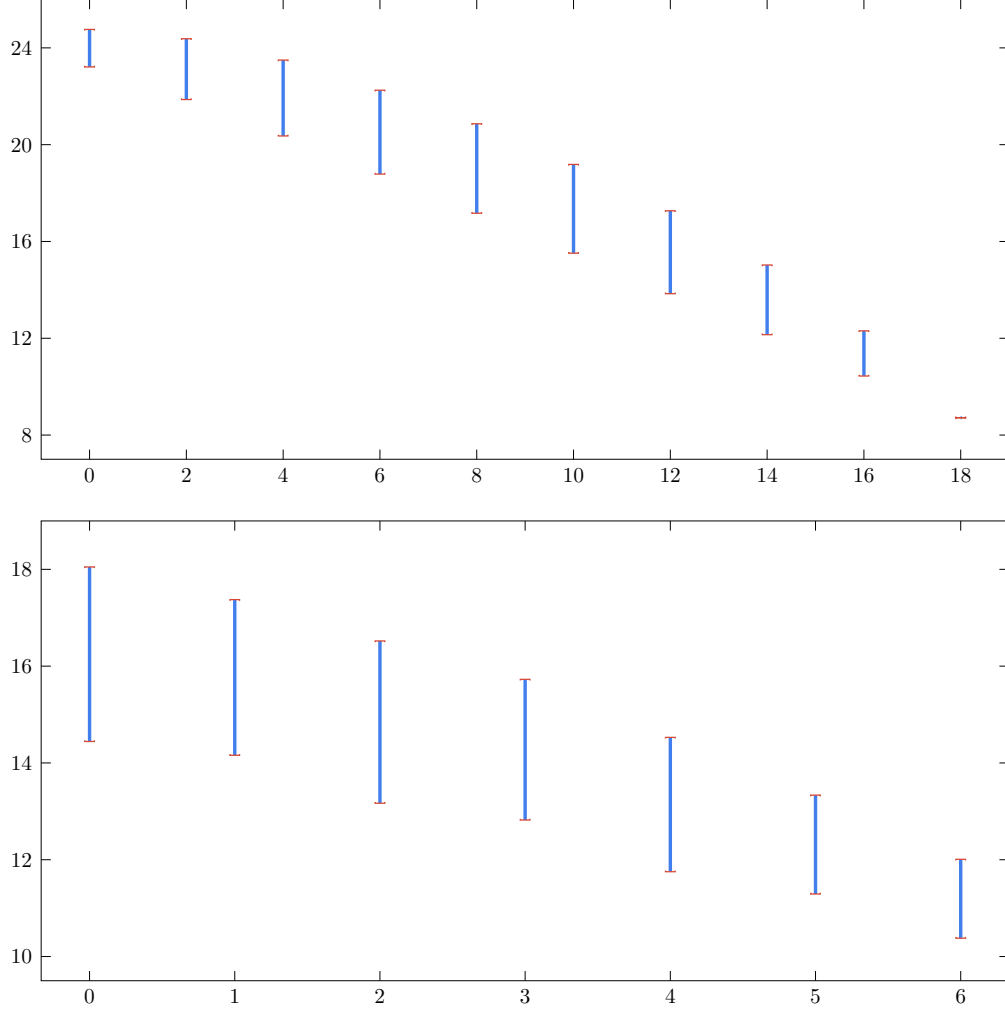


Figure 4: Plots of graph energy (on the vertical axis) vs. nullity (on the horizontal axis). Top: for all trees of order 20. Bottom: for all connected graphs of order 10 and size 20.

It was further reported that there are 158 borderenergetic graphs of order 11 in [57], and 572 borderenergetic graphs of order 12 in [58], although we did not check these results here due to time limits.

4.2.2 L -borderenergetic graphs

Let G be a graph and let A denote its adjacency matrix. The matrix $L = D - A$ where D is the diagonal matrix of vertex degrees of G is called the *Laplacian matrix* of G . The *Laplacian energy* [63] of G , denoted by $\mathcal{E}_L(G)$, is defined by

$$\mathcal{E}_L(G) = \sum_{i=1}^n |\mu_i - \bar{d}|,$$

where n is the order of G , \bar{d} is the average degree of G , and μ_1, \dots, μ_n are the eigenvalues of L . A simple property of the Laplacian energy is that for regular graphs it coincides

with graph energy. As an example, $\mathcal{E}_L(K_n) = \mathcal{E}(K_n) = 2n - 2$.

Current results on L -borderenergetic graphs can be found in [55, 64–70]. Several of these works include listings of small L -borderenergetic graphs of order at most 10. We used `SubsetTemplate` to verify these lists. This is especially of interest since results of [64] have been shown to be incomplete [65].

Order	Number of graphs
4	2
5	1
6	11
7	5
8	33
9	23
10	227

Table 2: Counts of L -borderenergetic graphs of small order.

To mitigate possible floating point errors, we instructed the framework to make a subset of graphs of a given order n whose Laplacian energy differs by at most 0.5 from $2n - 2$ by changing `DOUBLE_EQUALITY_THRESHOLD=0.5` in `DoubleUtil` class. We then carefully investigated (at a higher computational cost) each graph in the resulting subset for being L -borderenergetic. For this we used symbolic and arbitrary precision computation in Maple to reach a reliable conclusion. As our first result, we can confirm the counts reported in the literature of L -borderenergetic graphs for orders up to 9. These counts are summarized in Table 2.

However, for L -borderenergetic graphs of order 10 we obtained only 227 such graphs, while 233 were reported in [65]. We investigated the reported graphs in [65] and found six graphs listed there as being L -borderenergetic, while they are not. These false positives are the graphs H_{10}^{31} , H_{10}^{88} , H_{10}^{89} , H_{10}^{90} , H_{10}^{91} , and H_{10}^{92} in the notation of [65], shown in Table 3 along with their Laplacian energy.

4.3 Using ExtremalTemplate

The `ExtremalTemplate` finds graphs which maximize or minimize certain graph invariant, which is the focus of most problems in mathematical chemistry. Here we illustrate the use of this template on a conjecture about a recent topological index.

The *distance-sum heterogeneity index* of a graph $G = (V, E)$ is defined by Estrada and

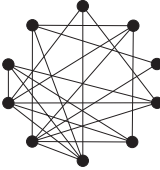
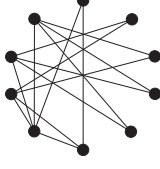
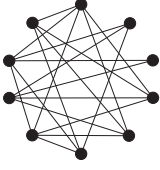
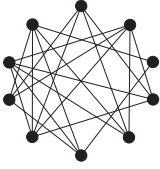
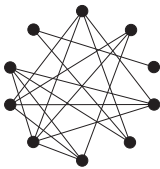
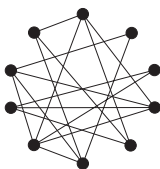
Graph	Graph6 code	Laplacian energy (approx.)
 H_{10}^{31}	ICQB}w\X_	19.3754231351000249
 H_{10}^{88}	I?ACJLYM_	18.0000048728229290
 H_{10}^{89}	IC'Czref?	17.9999954284824327
 H_{10}^{90}	IEoeeH] Jo	17.9999955203530093
 H_{10}^{91}	IC'Emg] I?	17.9999950133844081
 H_{10}^{92}	IC'DhpUb?	17.9999979727567947

Table 3: Graphs of order 10 wrongly reported as L -borderenergetic in [65].

Vargas-Estrada [49] as

$$\varphi(G) = \sum_{i \in V} \frac{d_i}{s_i} - 2 \sum_{\{i,j\} \in E} (s_i s_j)^{-1/2}$$

where d_i is the degree of vertex i and s_i is the sum of all distances from i to other vertices. Estrada and Vargas-Estrada [49] propose the problem of finding the graph of any given order which maximizes distance-sum heterogeneity index. Computational results of [49] for graphs of order at most 8 show that for $n = 3, 4, 5$, the star graph $K_{1,n-1} \cong K_1 \vee \overline{K}_{n-1}$ maximizes φ , while for $n = 6, 7, 8$ the maximum is achieved by the so-called *agave graph* $K_2 \vee \overline{K}_{n-2}$, where \vee denotes the join of two graphs. Inspired by these results, they conjecture the following.

Conjecture 1 [49] *Among the graphs of order $n \geq 6$, the agave graph has the maximum distance-sum heterogeneity index.*

Extending computational results of [49], we searched for graphs of order 9 and 10 which maximize φ and for these orders, we found graphs with distance-sum heterogeneity index larger than that of the agave graph. It turns out that these maximal graphs, like stars and agave graphs, are also joins of a complete and an empty graph.

A *split graph* is a graph whose vertex set V can be partitioned into a clique C and an independent set I . A *complete split graph* is a split graph in which every vertex in C is adjacent with every vertex in I . The complete split graph with $|V| = n$ and $|I| = \alpha$ is denoted by $\text{CS}(n, \alpha)$. Table 4 summarizes computational results on graphs with maximum distance-sum heterogeneity index.

Order	Graph
$n \in \{3, 4, 5\}$	$\text{CS}(n, n - 1)$
$n \in \{6, 7, 8\}$	$\text{CS}(n, n - 2)$
$n \in \{9, 10\}$	$\text{CS}(n, n - 3)$

Table 4: Graphs with maximum distance-sum heterogeneity index

Computational results of Table 4 suggest that for graphs G of a fixed order n , the maximum value of $\varphi(G)$ is attained by a complete split graph $\text{CS}(n, \alpha)$. Let $V = C \cup I$ be the vertex set of $\text{CS}(n, \alpha)$ as defined above. Then for every $i \in C$, $d_i = s_i = n - 1$, and for every $i \in I$, $d_i = n - \alpha$ and $s_i = n + \alpha - 2$. Therefore,

$$\begin{aligned} \varphi(\text{CS}(n, \alpha)) &= (n - \alpha) \cdot 1 + \alpha \cdot \frac{n - \alpha}{n + \alpha - 2} - 2 \left[\frac{\binom{n - \alpha}{2}}{n - 1} + \frac{\alpha(n - \alpha)}{\sqrt{(n - 1)(n + \alpha - 2)}} \right] \\ &= \frac{\alpha(n - \alpha)}{(n - 1)(n + \alpha - 2)} \left[2n + \alpha - 3 - 2\sqrt{(n - 1)(n + \alpha - 2)} \right]. \end{aligned}$$

For example, the above formula gives

$$\varphi(\text{CS}(9, 7)) = \frac{11 - \sqrt{112}}{4} \approx 0.10425 \quad \text{and} \quad \varphi(\text{CS}(9, 6)) = \frac{189 - 18\sqrt{104}}{52} \approx 0.10452,$$

which provides a counterexample to Conjecture 1.

Table 5 reports numerical values of $\varphi(\text{CS}(n, n - \omega))$ for small values of n and ω . For each fixed n , the largest such value is highlighted in bold to emphasize the pattern. The complete split graphs corresponding to these maximum values can be considered as

replacements of the agave graph in Conjecture 1. It can be easily seen that for every positive integer ω ,

$$\lim_{n \rightarrow \infty} \varphi(\text{CS}(n, n - \omega)) = \left(\frac{3}{2} - \sqrt{2}\right)\omega.$$

Assuming that the maximal graphs with respect to distance-sum heterogeneity index are complete split graphs, this implies that there is no bound on their clique number ω .

	$\omega = 1$	$\omega = 2$	$\omega = 3$	$\omega = 4$	$\omega = 5$	$\omega = 6$	$\omega = 7$	$\omega = 8$	$\omega = 9$	$\omega = 10$
$n = 2$	0									
$n = 3$	0.0337	0								
$n = 4$	0.0508	0.0239	0							
$n = 5$	0.0596	0.0505	0.0167	0						
$n = 6$	0.0648	0.0702	0.0432	0.0121	0					
$n = 7$	0.0684	0.0847	0.0673	0.0359	0.0092	0				
$n = 8$	0.0709	0.0957	0.0877	0.0610	0.0299	0.0072	0			
$n = 9$	0.0727	0.1042	0.1045	0.0842	0.0542	0.0251	0.0057	0		
$n = 10$	0.0742	0.1111	0.1185	0.1048	0.0784	0.0479	0.0213	0.0047	0	
$n = 11$	0.0754	0.1167	0.1303	0.1228	0.1010	0.0719	0.0423	0.0182	0.0039	0
$n = 12$	0.0763	0.1214	0.1403	0.1386	0.1217	0.0955	0.0657	0.0375	0.0158	0.0033
$n = 13$	0.0771	0.1253	0.1489	0.1524	0.1405	0.1179	0.0894	0.0598	0.0334	0.0138
$n = 14$	0.0778	0.1287	0.1564	0.1646	0.1573	0.1386	0.1126	0.0833	0.0545	0.0299
$n = 15$	0.0784	0.1316	0.1629	0.1754	0.1726	0.1578	0.1347	0.1067	0.0773	0.0498
$n = 16$	0.0789	0.1341	0.1686	0.1850	0.1863	0.1754	0.1555	0.1296	0.1007	0.0718
$n = 17$	0.0793	0.1364	0.1737	0.1936	0.1987	0.1916	0.1750	0.1515	0.1239	0.0948
$n = 18$	0.0797	0.1384	0.1782	0.2013	0.2100	0.2065	0.1932	0.1724	0.1466	0.1180
$n = 19$	0.0800	0.1402	0.1823	0.2083	0.2203	0.2202	0.2101	0.1921	0.1684	0.1410
$n = 20$	0.0803	0.1417	0.1860	0.2147	0.2297	0.2328	0.2258	0.2106	0.1892	0.1635

Table 5: Distance-sum heterogeneity index of some complete split graphs $\text{CS}(n, n - \omega)$

Estrada *et al.* [49] also consider extremal graphs for distance-sum heterogeneity index among all graphs of a fixed order and size. They give an algorithm to generate such maximal graphs in an iterative fashion, where starting with a star, each maximal graph is obtained from a previous one by adding an edge. All resulting graphs constructed in this way are split graphs. More specifically, each of these maximal graphs is a split graph with partition $V = C \cup I$ and a distinguished vertex $x \in C$ such that C induces a clique, every $i \in C \setminus \{x\}$ is adjacent to every $j \in I$, and x is adjacent to exactly β vertices in I . If $\beta \in \{0, \alpha\}$, we have a complete split graph. Estrada *et al.* [49] conjecture that these graphs have maximum distance-sum heterogeneity index among all graphs with the same order and size, and they report computational verification of this conjecture for orders $n \leq 8$. We further checked this conjecture for $n \in \{9, 10\}$ and found that the conjecture holds for these orders.

4.4 Using EquiTemplate

The `EquiTemplate` class can be used to find subsets of graphs which have the same value of a graph invariant. To illustrate this template, we provide counterexamples to two conjectures on interrelations of graph distance measures from [71]. Note that counterexamples to these conjectures have been obtained independently by Ilić and Ilić [72] as well.

For a pair of graph invariants, Dehmer *et al.* [71] study behavior of one invariant with respect to the other. For these comparisons, the distance measure

$$d(x, y) = 1 - e^{-(x-y)^2/\sigma^2}$$

is used, where $x, y \in \mathbb{R}$ and σ is the parameter of the Gauss function. This distance measure is due to Schädler [73]. If I is a graph invariant, the above distance function defines the graph distance

$$d_I(G, H) = d(I(G), I(H)) = 1 - e^{-(I(G)-I(H))^2/\sigma^2}$$

for any two graphs G and H . Several results and conjectures in [71] have the form

$$d_I(G, H) \leq d_J(G, H) \tag{2}$$

where I and J are two graph invariants and G and H are a pair of graphs. It is easily observed that

$$d_I(G, H) \leq d_J(G, H) \iff |I(G) - I(H)| \leq |J(G) - J(H)|.$$

Hence a possible way to contradict an inequality of the form (2) is to find a pair G, H of graphs such that $J(G) = J(H)$ while $I(G) \neq I(H)$, for which `EquiTemplate` is well suited.

4.4.1 Conjecture on Wiener index vs. Randić index

For a vertex u of G let d_u denote the degree of u , and for another vertex v of G let $d(u, v)$ denote the distance between u and v in G . Wiener index [74] and Randić index [75] of G are defined, respectively, as

$$W(G) = \sum_{\{u,v\} \subseteq V} d(u, v),$$

and

$$R(G) = \sum_{uv \in E} (d_u d_v)^{-1/2}.$$

The first conjecture of Dehmer *et al.* [71] that we tackle is as follows.

Conjecture 2 [71] *Let T and T' be any two trees of order n . Then $d_W(T, T') \geq d_R(T, T')$.*

We approach this conjecture by searching for pairs of trees with the same Wiener index. Since such trees do not necessarily have equal degree sequences, one expects their Randić index to be different. We were able to find two pairs of counterexamples already at order 7, which are presented in Fig. 5. Investigating further trees of higher orders, we found out that the second pair can be generalized as described below.

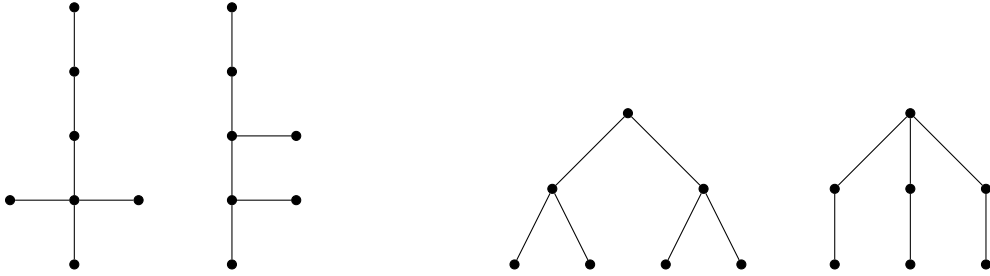


Figure 5: The two pairs of counterexamples of order 7 to Conjecture 2.

Let n and r be positive integers such that n is odd and $2 \leq r \leq n - 3$. We define two trees $B_{n,r}$ and F_n as follows. The tree $B_{n,r}$ is obtained from a path on three vertices by attaching r pendent vertices at one leaf, and $n - r - 3$ pendant vertices at the other leaf. The tree F_n is obtained by subdividing every edge of the star $K_{1,(n-1)/2}$. The pair of trees on the right in Fig. 6 consists of $B_{7,2}$ and F_7 . The simple structure of these trees easily yields the following results.

Proposition 1 *Let n and r be positive integers such that n is odd and $2 \leq r \leq n - 3$. Then*

$$\begin{aligned} W(B_{n,r}) &= (n+r)^2 - 3(r+1)^2 - n + 1, \\ R(B_{n,r}) &= \frac{r + \frac{1}{\sqrt{2}}}{\sqrt{r+1}} + \frac{n-r-3 + \frac{1}{\sqrt{2}}}{\sqrt{n-r-2}}, \\ W(F_n) &= \frac{1}{2}(n-1)(3n-5), \\ R(F_n) &= \frac{n-1}{2\sqrt{2}} + \frac{\sqrt{n-1}}{2}. \end{aligned}$$

These formulas allow us to find infinitely many pairs of counterexamples to Conjecture 2. For example, if $n = 2k + 1$ and $r = k - 1$, the pair $B_{n,r}, F_n$ is a counterexample since

$$W(B_{n,r}) = W(F_n) = 6k^2 - 2k,$$

while

$$R(B_{n,r}) - R(F_n) = \frac{k\sqrt{k} + k - 2 - \sqrt{8}(k - 1)}{\sqrt{2k}} = \Theta(k).$$

One might wonder here whether there are counterexamples to Conjecture 2 with different Wiener indices. Motivated by this question, we investigated differences $|W(T) - W(T')|$ and $|R(T) - R(T')|$ for pairs of trees T and T' of the same order n . Fig. 6 shows a plot of these differences for $n = 12$. It can be observed from this plot that most pairs of trees satisfy the assertion of Conjecture 2 (note that the axes in this plot have different scales, so that the line $y = x$ is rather steep). On the other hand, the accumulation of points on the left side of the plot suggests that there exist counterexamples with different Wiener index. Indeed, this plot includes a point with Wiener difference 1 (on the horizontal axis) and Randić difference bigger than 1. The corresponding pair of trees is shown in Fig. 7.

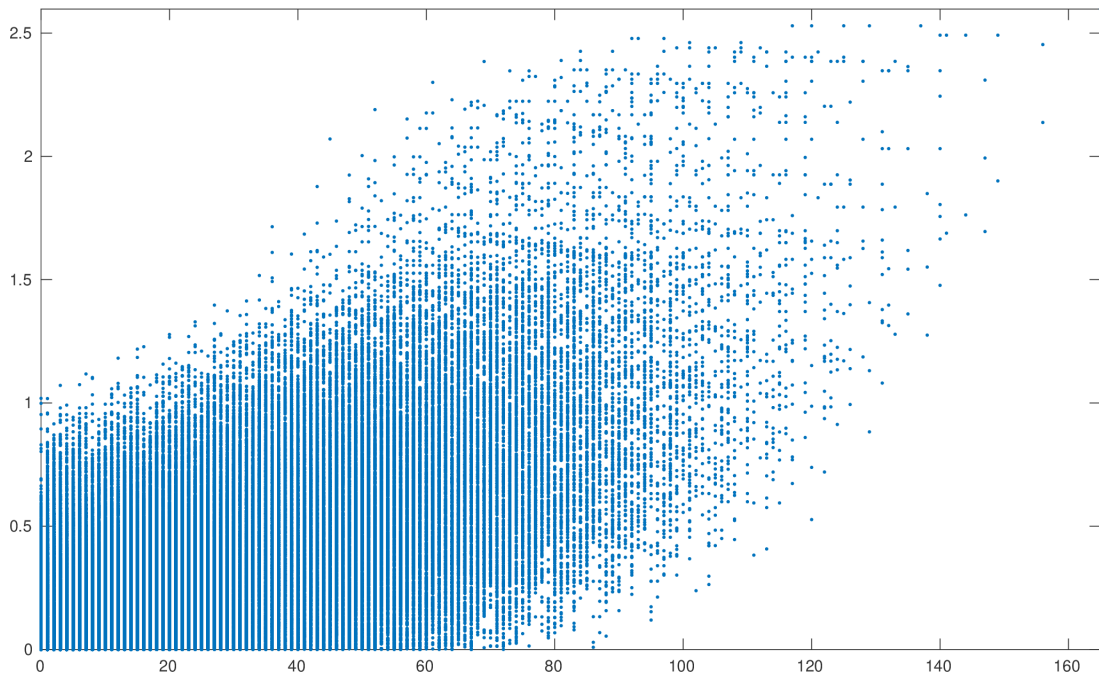


Figure 6: A plot of $|R(T) - R(T')|$ (on the vertical axis) against $|W(T) - W(T')|$ (on the horizontal axis) for all pairs T, T' of trees of order 12.

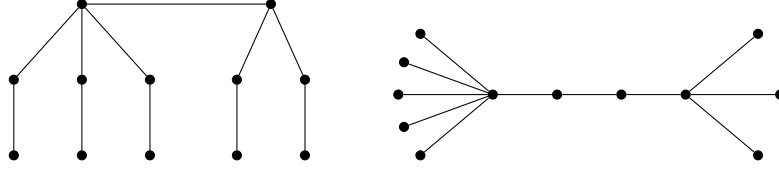


Figure 7: Counterexample to Conjecture 2 with different Wiener indices.

4.4.2 Conjecture on graph energy vs. graph entropy

The *entropy* of G [71] is defined by

$$Ig(G) = \log(\mathcal{E}(G)) - \frac{1}{\mathcal{E}(G)} \sum_{i=1}^n |\lambda_i| \log |\lambda_i|,$$

where $\mathcal{E}(G) = \sum_{i=1}^n |\lambda_i|$ is the energy of G and $\lambda_1, \dots, \lambda_n$ are adjacency eigenvalues of G . It represents the standard Shannon entropy [76] applied to the probability vector $\left(\frac{|\lambda_i|}{\mathcal{E}(G)}\right)_{i=1, \dots, n}$. Dehmer *et al.* [71] posed the following conjecture.

Conjecture 3 [71] *Let T and T' be any two trees of order n . Then $d_{\mathcal{E}}(T, T') \geq d_{Ig}(T, T')$.*

Taking the same approach as for the previous conjecture, we looked for pairs T, T' of trees of the same order, such that $\mathcal{E}(T) = \mathcal{E}(T')$ while $Ig(T) \neq Ig(T')$, which would serve as counterexamples to this conjecture. It should be noted that unlike Wiener index, graph energy is not necessarily an integer, and numerical errors might yield `EquiTemplate` to report false positives. To mitigate this effect and check the correctness of numerical calculations in Java, we used symbolic computation in Maple for graphs of small order and arbitrary-precision arithmetic of Maple for graphs of higher orders.

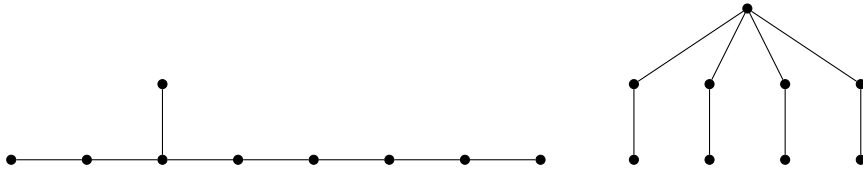


Figure 8: A counterexample to Conjecture 3.

This search produced a pair T, T' of trees of order 9 with the desired property, presented in Fig. 8. Let T be the left tree in this figure and T' be the right tree. The adjacency spectrum of T is

$$Sp(T) = \left\{ 2, \frac{\sqrt{5}+1}{2}, 1, \frac{\sqrt{5}-1}{2}, 0, \frac{1-\sqrt{5}}{2}, -1, -\frac{\sqrt{5}+1}{2}, -2 \right\},$$

while the adjacency spectrum of T' is

$$Sp(T') = \left\{ \sqrt{5}, 1, 1, 1, 0, -1, -1, -1, -\sqrt{5} \right\}.$$

From here we can see that $\mathcal{E}(T) = \mathcal{E}(T') = 6 + 2\sqrt{5}$, while $Ig(T) \approx 2.005062$ and $Ig(T') \approx 1.992056$. We had found several more counterexamples of this type: one pair of equienergetic trees of order 13, one pair of order 14, four pairs of order 15, seven pairs of order 16, and three pairs of order 17.

Certainly, counterexamples to Conjecture 3 do not have to constitute equienergetic pairs of trees. Fig. 9 shows a plot of $|Ig(T) - Ig(T')|$ against $|\mathcal{E}(T) - \mathcal{E}(T')|$ for pairs T, T' of trees of order 12. It can be seen from this plot that while most pairs of trees of order 12 satisfy Conjecture 3, there are also quite a few counterexamples in which trees have different energies.

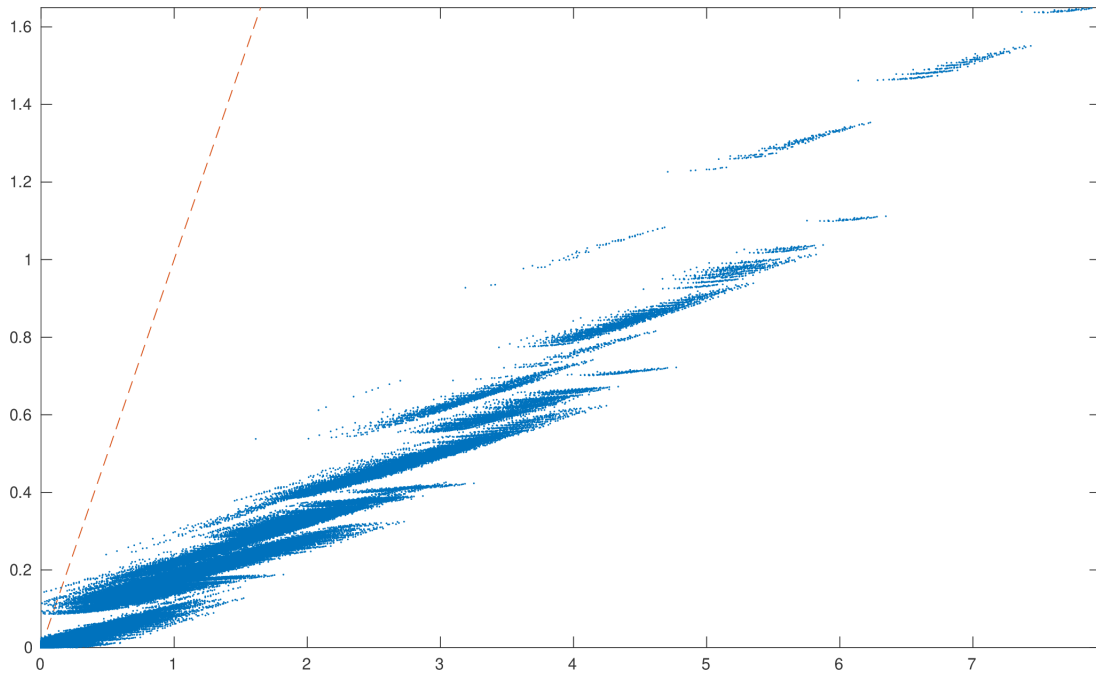


Figure 9: A plot of $|Ig(T) - Ig(T')|$ (on the vertical axis) against $|\mathcal{E}(T) - \mathcal{E}(T')|$ (on the horizontal axis) for all pairs T, T' of trees of order 12. The dashed line has slope 1.

5 Conclusions

We have presented here a Java framework that can be quickly and simply adapted for solving particular instances of a few general research problems in graph theory and mathematical chemistry. Its usefulness is demonstrated by improving understanding of several

recent results from the literature. Since the framework is based on numerical computations, it is also possible that its use can lead to either false positives or false negatives when dealing with real-valued invariants that are densely distributed in an interval, emphasizing the need to verify results obtained by this framework in another symbolic computation or arbitrary-precision arithmetic package. As a matter of fact, it would be even more useful to have a piece of software that would unify ability to perform fast numerical computations with both symbolic algebraic computation and arbitrary-precision arithmetic. At the moment it seems that Python is well suited to pack all these abilities into a coherent whole: it could compile and run this Java framework in the background when fast numerical computation is needed, while SymPy library [77] offers tools for symbolic algebraic computations, and mpmath library [78] offers arbitrary-precision arithmetic. We leave this idea for future work.

References

- [1] D. Cvetković, L. Kraus, S. Simić, Discussing graph theory with a computer I, Implementation of graph theoretic algorithms, Univ. Beograd, Publ. Elektrotehn. Fak., Ser. Mat. Fiz., No. 716–734 (1981), 100–104.
- [2] D. Cvetković, Discussing graph theory with a computer II, Theorems suggested by the computer, Publ. Inst. Math. (Beograd), 33(47) (1983), 29–33.
- [3] D. Cvetković, Discussing graph theory with a computer VI, Theorems proved by the aid of the computer, Bull. Acad. Serbe Sci. Arts, Cl. Sci. Math. Natur., Sci. Math., 47 (1988), 51–70.
- [4] S. Fajtlowicz, W.A. Waller, On two conjectures of Graffiti, Combinatorics, graph theory, and computing, Proc. 17th Southeast. Conf., Boca Raton, 1986, Congr. Numerantium 55 (1986), 51–56.
- [5] S. Fajtlowicz, On conjectures of Graffiti. II, Combinatorics, graph theory, and computing, Proc. 18th Southeast. Conf., Boca Raton, 1987, Congr. Numerantium 60 (1987), 189–197.
- [6] S. Fajtlowicz, On conjectures of Graffiti, Discrete Math. 72 (1988), 113–118.
- [7] S. Fajtlowicz, On conjectures of Graffiti. III, Combinatorics, graph theory, and computing, Proc. 19th Southeast. Conf., Boca Raton, 1988, Congr. Numerantium 66 (1988), 23–32.
- [8] S. Fajtlowicz, On conjectures of Graffiti. IV, Combinatorics, graph theory, and computing, Proc. 20th Southeast Conf., Boca Raton, 1989, Congr. Numerantium 70 (1990), 231–240.
- [9] G. Caporossi, P. Hansen, Variable neighborhood search for extremal graphs. I: The AutoGraphiX system, Discrete Math. 212 (2000), 29–44.
- [10] P. Hansen, G. Caporossi, AutoGraphiX: an automated system for finding conjectures in graph theory, Rusu, Irena (ed.), Proceedings of the 6th international conference on graph theory, Marseille-Luminy, France, August 28–September 2, 2000, Amsterdam: Elsevier, 2000, pp. 158–161.
- [11] M. Aouchiche, G. Caporossi, P. Hansen, M. Laffay, Autographix: a survey, Raspaud, André (ed.) et al., 7th international colloquium on graph theory, Hyeres, France, September 12–16, 2005, Electronic Notes in Discrete Mathematics 22 (2005), 515–520.
- [12] M. Aouchiche, J.M. Bonnefoy, A. Fidahoussen, G. Caporossi, P. Hansen, L. Hiesse, J. Lacheré, A. Monhait, Variable neighborhood search for extremal graphs. XIV: The AutoGraphiX 2 system, Liberti, Leo et al., Global optimization. From theory to implementation, Nonconvex Optimization and Its Applications 84, New York: Springer, 2006, pp. 281–310.

- [13] newGRAPH, a system for visualization and interactive modification of graphs and automatic recalculation of graph invariants, available at <http://www.mi.sanu.ac.rs/newgraph/>
- [14] D. Stevanović, V. Brankov, An Invitation to newGRAPH, *Rend. Semin. Mat. Messina, Ser. II* 9(25) (2003), 211–216.
- [15] V. Brankov, D. Cvetković, S. Simić, D. Stevanović, Simultaneous editing and multilabelling of graphs in system newGRAPH, *Univ. Beograd. Publ. Elektr. Fak, Ser. Mat.* 17 (2006), 112–121.
- [16] Grinvin, available at <http://www.grinvin.org/en/>
- [17] A. Peeters, K. Coolsaet, G. Brinkmann, N. Van Cleemput, V. Fack, GrInvIn in a nutshell, *J. Math. Chem.* 45 (2009), 471–477.
- [18] Algorithmic Solutions Software GmbH, LEDA, C++ class library for efficient data types and algorithms, available at <http://www.algorithmic-solutions.com/leda/index.htm>
- [19] K. Mehlhorn, S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge: Cambridge University Press, 1999.
- [20] S. Pemmaraju, S. Skiena, *Computational discrete mathematics. Combinatorics and graph theory with Mathematica*, Cambridge: Cambridge University Press, 2003.
- [21] GraphTheory Package, Maple Programming Help, <http://www.maplesoft.com/support/help/Maple/view.aspx?path=GraphTheory>
- [22] NetworkX, Software for complex networks, available at <https://networkx.github.io/>
- [23] A.A. Hagberg, D.A. Schult, P.J. Swart, Exploring network structure, dynamics, and function using NetworkX, in: (G. Varoquaux, T. Vaught, J. Millman, eds.) *Proc. 7th Python in Science Conference (SciPy2008)*, Pasadena, USA, August 2008, pp. 11–15.
- [24] D. Stevanović, A. Vasilyev, MathChem, available at <http://mathchem.iam.upr.si/>
- [25] A. Vasilyev, D. Stevanović, MathChem: a Python package for calculating topological indices, *MATCH Commun. Math. Comput. Chem.* 71 (2014), 657–680.
- [26] M. Knor, B. Lužar, R. Škrekovski, I. Gutman, On Wiener Index of Common Neighborhood Graphs, *MATCH Commun. Math. Comput. Chem.* 72 (2014), 321–332.
- [27] D. Vukičević, Note on the graphs with the greatest edge-Szeged index, *MATCH Commun. Math. Comput. Chem.* 61 (2009) 673–681.
- [28] J.R. Dias, Structural origin of specific eigenvalues in chemical graphs of planar molecules, *Molec. Phys.* 85 (1995), 1043–1060.
- [29] C. da Fonseca, M. Ghebleh, A. Kalso, D. Stevanović, Counterexamples to a Conjecture on Wiener Index of Common Neighborhood Graphs, *MATCH Commun. Math. Comput. Chem.* 72 (2014), 333–338.
- [30] D. Stevanović, Counterexamples to conjectures on graphs with greatest edge-Szeged index, *MATCH Commun. Math. Comput. Chem.* 64 (2010), 603–606.
- [31] P.W. Fowler, D. Stevanović, M. Milošević, Counterexamples to a conjecture of Dias on eigenvalues of chemical graphs, *MATCH Commun. Math. Comput. Chem.* 63 (2010), 727–736.
- [32] B.D. McKay, A. Piperno, nauty and Traces, available at <http://pallini.di.uniroma1.it/>
- [33] B.D. McKay, A. Piperno, Practical Graph Isomorphism, II, *J. Symb. Comput.* 60 (2014), 94–112.
- [34] CAAGT, Fullerenes, available at <http://caagt.ugent.be/buckygen/>
- [35] G. Brinkmann, J. Goedgebeur, B.D. McKay, The Generation of Fullerenes, *J. Chem. Inform. Modeling* 52 (2012), 2910–2918.
- [36] J. Goedgebeur, B.D. McKay, Recursive generation of IPR fullerenes, *J. Math. Chem.* 53 (2015), 1702–1724.
- [37] G. Brinkmann, B.D. McKay, plantri and fullgen, available at <https://users.cecs.anu.edu.au/~bdm/plantri/>

- [38] G. Brinkmann, B.D. McKay, Fast generation of planar graphs, MATCH Commun. Math. Comput. Chem. 58 (2007), 323–357 (expanded version available at <https://users.cecs.anu.edu.au/~bdm/papers/plantri-full.pdf>).
- [39] G. Brinkmann, S. Greenberg, C. Greenhill, B.D. McKay, R. Thomas, P. Wollan, Generation of simple quadrangulations of the sphere, Discrete Math. 305 (2005), 33–54.
- [40] G. Brinkmann, B.D. McKay, Construction of planar triangulations with minimum degree 5, Discrete Math. 301 (2005), 147–163.
- [41] G. Brinkmann, K. Coolsaet, J. Goedgebeur, H. Mélot, House of Graphs: a database of interesting graphs, Discrete Appl. Math. 161 (2013), 311–314. Available at <http://hog.grinvin.org>
- [42] B.D. McKay, Graphs, available at <http://users.cecs.anu.edu.au/~bdm/data/graphs.html>
- [43] G. Royle, Combinatorial catalogues, available at <http://staffhome.ecm.uwa.edu.au/~00013890/>
- [44] Colt, available at <http://dst.lbl.gov/ACSSoftware/colt/>
- [45] BlueJ, available at <http://www.bluej.org/>
- [46] M. Kölling, The BlueJ Tutorial, available at <https://www.bluej.org/tutorial/tutorial-v4.pdf>
- [47] D.J. Barnes, M. Kölling, Objects First with Java, A Practical Introduction using BlueJ, Pearson, 2016.
- [48] T. Kamada, S. Kawai, An algorithm for drawing general undirected graphs, Inf. Process. Lett. 31 (1989), 7–15.
- [49] E. Estrada, E. Vargas-Estrada, Distance-sum heterogeneity in graphs and complex networks, Applied Math. Comput. 218 (2012), 10393–10405.
- [50] I. Gutman, The energy of a graph, Ber. Math. Statist. Sect. Forschungsz. Graz 103 (1978), 1–22.
- [51] I. Gutman, N. Cmiljanović, S. Milosavljević, S. Radenković, Effect of non-bonding molecular orbitals on total π -electron energy, Chem. Phys. Lett. 383 (2004), 171–175.
- [52] I. Gutman, D. Stevanović, S. Radenković, S. Milosavljević, N. Cmiljanović, Dependence of the total π -electron energy on large number of non-bonding molecular orbitals, J. Serb. Chem. Soc. 69 (2004), 777–782.
- [53] I. Gutman, I. Triantafyllou, Dependence of graph energy on nullity: a case study, MATCH Commun. Math. Comput. Chem. 76 (2016), 761–769.
- [54] S. Gong, X. Li, G. Xu, I. Gutman, B. Furtula, Borderenergetic graphs, MATCH Commun. Math. Comput. Chem. 74 (2015), 321–332.
- [55] F. Tura, L -Borderenergetic graphs, MATCH Commun. Math. Comput. Chem. 77 (2017), 37–44.
- [56] X. Li, M. Wei, S. Gong, A computer search for the borderenergetic graphs of order 10, MATCH Commun. Math. Comput. Chem. 74 (2015), 333–342.
- [57] Z. Shao, F. Deng, Correcting the number of borderenergetic graphs of order 10, MATCH Commun. Math. Comput. Chem. 75 (2016), 263–266.
- [58] B. Furtula, I. Gutman, Borderenergetic graphs of order 12, Iranian J. Math. Chem. 8 (2017), 339–343.
- [59] Y. Hou, Q. Tao, Borderenergetic threshold graphs, MATCH Commun. Math. Comput. Chem. 75 (2016), 253–262.
- [60] B. Deng, X. Li, I. Gutman, More on borderenergetic graphs, Linear Algebra Appl. 497 (2016), 199–208.
- [61] X. Li, M. Wei, X. Zhu, Borderenergetic graphs with small maximum or large minimum degrees, MATCH Commun. Math. Comput. Chem. 77 (2017), 25–36.
- [62] I. Gutman, On Borderenergetic graphs, Bull. Acad. Serbe Sci. Arts (Cl. Math. Natur.) 42 (2017), 9–18.
- [63] I. Gutman, B. Zhou, Laplacian energy of a graph, Linear Algebra Appl. 414 (2006), 29–37.

- [64] Q. Tao, Y. Hou, A computer search for the L -borderenergetic graphs, MATCH Commun. Math. Comput. Chem. 77 (2017), 595–606.
- [65] S. Elumalaia, M.A. Rostamib, Correcting the Number of L -Borderenergetic Graphs of Order 9 and 10, MATCH Commun. Math. Comput. Chem. 79 (2018), 311–319.
- [66] B. Deng, X. Li, More on L -borderenergetic graphs, MATCH Commun. Math. Comput. Chem. 77 (2017), 115–127.
- [67] L. Lu, Q. Huang, On the existence of non-complete L -borderenergetic graphs, MATCH Commun. Math. Comput. Chem. 77 (2017), 625–634.
- [68] F. Tura, L -Borderenergetic Graphs and Normalized Laplacian Energy, preprint, arXiv:1611.01461, 2016.
- [69] B. Deng, X. Li, J. Wang, Further results on L -borderenergetic graphs, MATCH Commun. Math. Comput. Chem. 77 (2017), 607–616.
- [70] B. Deng, X. Li, On L -Borderenergetic Graphs with Maximum Degree at Most 4, MATCH Commun. Math. Comput. Chem. 79 (2018), 303–310.
- [71] M. Dehmer, F. Emmert-Streib, Y. Shi, Interrelations of Graph Distance Measures Based on Topological Indices, PLoS ONE 9(4) (2014), e94985.
- [72] A. Ilić, M. Ilić, Counterexamples to conjectures on graph distance measures based on topological indexes, Appl. Math. Comput. 296 (2017), 148–152.
- [73] C. Schädler, Die ermittlung struktureller ähnlichkeit und struktureller merkmale bei komplexen objekten: Ein konnektionistischer ansatz und seine anwendungen, PhD thesis, Technische Universität Berlin, 1999.
- [74] H. Wiener, Structural determination of paraffin boiling points, J. Am. Chem. Soc. 69 (1947) 17–20.
- [75] M. Randić, Characterization of molecular branching, J. Am. Chem. Soc. 97 (1975), 6609–6615.
- [76] C. Shannon, W. Weaver, The Mathematical Theory of Communication, University of Illinois Press, Urbana, USA, 1949.
- [77] A. Meurer, C.P. Smith, M. Paprocki, O. Čertík, S.B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J.K. Moore, S. Singh, T. Rathnayake, S. Vig, B.E. Granger, R.P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M.J. Curry, A.R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, SymPy: symbolic computing in Python, PeerJ Comput. Sci. 3 (2017), e103.
- [78] F. Johansson, mpmath: a Python library for arbitrary-precision floating-point arithmetic, Available at <http://mpmath.org/> [Accessed Jun 4, 2018].