GraphDating - a Tool for Graph Alignments

Christina Kuhn, Jens-Tilman Rau, Aljoscha Rydzyk and Johann Wurz

University of Leipzig - Department of Computer Science

Härtelstraße 16-18, 04107 Leipzig - Germany

2019
Oktober

Abstract

Comparing the similarity of two or more graphs due to graph alignments has a broad spectra of applications. GraphDating is a software to generate multiple graph alignments, but is also able to find cliques of single graphs. The program has many additional functions for further specifications of special use-cases. GraphDating is written in Python 3 in a modular structure, to maintain the opportunity to specify or expand certain functions. It is available on GitHub (/tick-trick-and-clique/BestesRepository). GraphDating is described in the following manual. The first part characterizes the utilized main functions and algorithms, but also the applied graph format. The second part describes the application of GraphDating by explaining parameters and functions with command-line examples.

Contents

1 Functions and implemented algorithms

A description of the most important functions and implemented algorithms is given in the following section, to create an overview of the program‘s features and how they are connected to each other.

1.1 Graph format

The presented graph format will be needed as input (besides *json* format) and is produced as output (if an output is desired). The file format compromises plain text and is induced with *.graph* at its end (e.g *example.graph*). The presented graph format will be called standard graph format in the following for simplicity. The required format is organised in three simple blocks - header, vertices and edges - separated from each other by an empty line (newline). The single parts are described in the following section. Specific examples are located in the appendix B. The presented example is *graph6.graph*.

Header

The header consists of five lines, each line has a parameter and a attribute separated by a semicolon. The parameters and attributes in correct order are:

1. number of nodes;Integer
2. number of edges;Integer
3. Information if nodes are labelled;Boolean
4. Information if edges are labelled;Boolean
5. Information if graph is directed;Boolean

Here is an example for a properly created header:

```
#nodes;3
#edges;3
nodes labelled;True
edges labelled;True
directed graph;True
```

Vertices

The vertex section consists of as many lines as vertices the graph consists of. The lines begin with ascending integers, the first vertex is named **1**. If the vertices are labelled, the integer is followed by a semicolon and the desired label. The label must not include a hash character ("#"). The reason therefore is stated in the last subsection of the header description.
Here is an example for a labelled vertex section appropriate to the already described header:

```
1;a
2;b
3;c
```

Edges

The edge section consists of as many lines as edges the graph consists of. The lines begin with the integers of vertices, the edges connects, seperated by a semicolon. If the edges are labelled, the two integers are followed by a semicolon and the desired label. The label must not include a hash character ("#"). The reason therefore is stated in the next subsection of the header description.
Here is an example for a labelled edge section appropriate to the already described header and vertex section:

```
1;2;a
2;1;b
2;3;c
```

Please remember that in the full format, the single sections are seperated by an empty line (newline). See appendix B for examples in full format.

Additional information about labels

If the modular product or a graph alignment of labeled graphs is applied, one can see labels separated by "#" to account for both labels of the input graphs. Therefore the label in standard graph format must not include "#"! If you align a labeled graph with an unlabeled one, the output (sub-)graph will get the "$None$" label at positions of unlabeled graphs.
Here is an example output of the vertex section in standard graph format after the application of the modular product on two labelled graphs:

```
1;a#c
2;b#g
3;c#i
```

1.2 Parser

There are different parsers implemented in GraphDating to maintain integrity. The presented graph format is usually transfered into an internal structure of several classes. The graph class consists of the vertex and the edge class. To generate an output, the internal graph structure is in most cases again transfered in the readable graph format. This basic parser also intercepts many small issues and calls exceptions. It does work as a control feature for inputs and outputs too, e.g. checking the number of vertices in the header with the number of lines of vertices.

1.3 Construction of random graphs

The construction of random graphs was implemented to test GraphDatings performance and to play around with its functions. It was also integrated for users, who do not bring their own graphs or who are to convenient to transfer their own graphs in the advised graph format. All randomly created graphs are without any labels. There are two possibilities to create your own random graphs.

Standard

To create a standard graph you need to supply three attributes. You can choose the number of vertices (integer), the mean relative connectivity (float, $0.0 < p \leq 1.0$) and the parameter, if your randomly created graph should be directed or undirected (boolean).

Cluster

To create a clustered graph you need to supply four attributes. You can choose the number of vertices (interger), the degree at each vertex (integer), the number of deleted vertices (integer) and the number of deleted edges (integer). It is an interesting task to align clustered graphs, because the runtime of many graph algorithms (e.g. Bron-Kerbosch algorithm) is suppossed to expand enormously.

1.4 Guide trees

The application of guide trees is important for progressive alignments. They define the chronological sequence of pairwise alignments to gain a multiple alignment. Therefore GraphDating is able to calculate an guide tree from the supplied input graphs. The program calculates it by comparing the densities from the supplied graphs or by the greatest subgraph isomorphism. The densest graphs or graphs with maximal number of vertices in subgraphs will be aligned first. The user can also implement an own comparison function and pass it to GraphDating (see 2.3

Guide trees for details). The resulting guide tree is saved in the *newick* format and can be output too. The user can also submit an own guide tree in *newick* format for a graph alignment.

1.5 Graph alignments

The focus of GraphDating is the calculation of graph alignments. They are done by progressive alignments - a progression of pairwise alignments in a defined order to gain a multiple alignment. The chronological sequence of pairwise alignments can be set by a guide tree, which can be calculated by GraphDating or which can be supplied by the user. There are two possibilities to create such an alignment.

Bron-Kerbosch algorithm and modular product of graphs[1]

The Bron-Kerbosch algorithm is a backtracking algorithm to find maximal cliques of an undirected graph. It can be accelerated by choosing a pivot element, but works also without pivoting. There are two options of choosing a pivot element implemented: selecting randomly a vertex as pivot or selecting the vertex with highest cardinality as pivot. It is also possible to supply a so called anchor graph, where you have to distinguish between anchors for the bron-kerbosch algorithm alone and anchors for graph alignments with the bron-kerbosch algorithm. For the first case: The anchor has to be a clique, which the algorihm would expands if possible. For the second case: The anchor does not necessarily have to be a clique. For both cases: The anchor has to be a subgraph of one of the input graphs.

For finding similarity of graphs in a graph alignment, the Bron-Kerbosch algorithm is applied to the modular product of two graphs. It is also possible to run this algorithm to a supplied graph, just for clique finding. The modular product of two graphs G and H is the cartesian product (G, H) of these two graphs. This creates a new graph with all combinations of vertices of the two given graphs. In this new graph, edges between vertices (a,b) and (a',b') are formed if and only if either a is adjacent to a' and b is adjacent to b' **or** a is not adjacent to a' and b is not adjacent to b'. By applying the Bron-Kerbosch algorithm on (G, H), you find the maximal common induced subgraph of the two original graphs. A pairwise graph alignment is done and can be developed to a multiple graph alignment by building the modular product of (G, H) and another graph. A guidetree can be used for defining the running order of the multiple graph alignment.

Cordella algorithm[2]

The Cordella algorithm implements an improved version of a graph matching algorithm and can be applied to directed or undirected graphs. It was especially

designed for matching large graphs. The algorithm implements a mapping M of vertices of graph G and graph H. $M(G,H)$ is called a graph-subgraph isomorphism if $M(G,H)$ is an isomorphism of H and a subgraph of G. Therefore it is important, that H consists of less or the same number vertices as G. The Cordella algorithm claims to be more efficient especially for large, well structured graphs (cluster).

1.6 Graph representation

GraphDating is connected to Neo4j to visualize graphs and graph alignments. See 2.3 **Graph representation with Neo4j** for the application of Neo4j with GraphDating.

2 Command-line declarations

The use-cases of GraphDating will be presented in the following section, to declare the common syntax and to show off its skill set. All positions of parameters are exchangeable and the usage of absolute paths is also possible. We recommend to use pypy3 to run GraphDating, because it shortens the runtime tremendously. **Tip:** If you dont have any graphs or graphs in the used format, you can create random graphs with GraphDating or download them from databases e.g. PubChem. See the subsection 2.3 **Random graphs** for further informations. To read the help message, use parameter **-h** or see appendix A.

2.1 Inputs

Files and formats

To supply one or several input graphs, you use the flag **-i**. You have to put a space character between submitted graph files, if you want to input more than one graph. It is also possible, to use chemical structures directly from PubChem. Therefore you need to download the 2D structure of a particular molecule in *json* format. To supply graphs in *json* format, you need to specify the input by using the flag **-if json**. You can also negelct all hydrogen atoms from *json* inputs by adding the flag **-nh**. To be more precise: by using **-nh** you neglect all atoms with the label "1" in *json* format. It is not possible to read graphs in *json* format and the displayed graph format at the same time.

If you want to read several graphs in standard format, your command-line could look like this:

```
python3 GraphDating.py -i graph2.graph graph3.graph
```

If you want to read a single graph in *json* format and neglect all found hydrogen atoms, your command-line could look like this:

```
python3 GraphDating.py -if json -i Ethane.json -nh
```

Please note, that the usage of flags **-i** and **-if** without any additional parameters will not calculate anything. It will only show some information about the inputs.

Anchors

You can supply a so called anchor graph for clique finding or graph alignenments with the bron-kerbosch algorithm to append an already known clique or an already known alignment. Therefore you use the flag **-a**. GraphDating will only return reasonable outputs, if the passed anchor is an induced subgraph of the first input file and is included structurally in all other input files. The input anchor needs to be in the standard graph format.

If you want to read several graphs in standard format and also supply an anchor, your command-line could look like this:

```
python3 GraphDating.py -i graph3.graph graph6.graph \
-a graph2.graph
```

Please note, that it is not possible to use anchor graphs in *json* format or to apply anchors for the cordella algorithm.

2.2 Outputs

Calculated graphs and subgraphs

To output calculated graphs and subgraphs, you use the flags **-go** and **-sgo**. You want to save the entire graph (**-go**) from the calculation of random graphs and the modular product. But you want to save an overall alignment graph (**-sgo**) from graph alignments. You can also call for all subgraphs of an operation by providing the additional flag **-s** to the **-sgo** flag. If you want to output all found cliques from Bron-Kerbosch-algorithm for example, you use **-sgo -s**.

You can specify a file name for graph outputs and subgraph outputs. By default, subgraph outputs will have "_Subgraph_" followed by an ascending number in its file name. You can also choose the number of subgraph outputs by supplying an integer behind the **-sgo** flag. You can also combine input and output functions to reformat *json* files in *graph* files. You will find further information to the output of a particular function in the description of these function in 2.3.

If you want to format a *json* file to a standard graph file, your command-line could look like this:

```
python3 GraphDating.py -i Ethane.json -if json \
-go ethane_pubchem.graph
```

Please note, that you can not save graphs or subgraphs in *json* format. Also note, that the labels of vertices of reformatted files from PubChem represent the particular atomic number. The lables from the associated edges represent the bonding type (e.g. 1 = single bond, 2 = double bond,3 = triple bond, 7 = ionic bond, ...).

Newick format

GraphDating can calculate guide trees and save them in *newick* format for later usage. Therefore you need to use the flag **-no**. You find further information in 2.3 in the section about guide trees.

2.3 Operations

Random graphs

To calculate random graphs, you use the flags **-rg** or **-rc**. You also need to supply several attributes. For standard graphs you can choose the number of vertices, the relative connectivity and if your randomly created graph should be directed or undirected. For clustered graphs you can choose the number of vertices, the desired degree at each vertex, how many vertices you want to delete from the cluster and how many edges you want to delete from the cluster. Clustered graphs are undirected by default.

If you want to create a random standard graph, your command-line could look like this:

```
python3 GraphDating.py -rg 10 0.5 False \
-go my_random_graph1.graph
```

If you want to create a random clustered graph, your command-line could look like this:

```
python3 GraphDating.py -rc 20 3 2 2 -go my_random_graph1.graph
```

To create a lot of random graphs simoultaneously at once, you can use the provided Nextflow file **create_random_graphs.nf**. The application and changeable parameters are described in the source code of the Nextflow file.

Clique finding

To find cliques of a graph, you use the flag **-bk**. GraphDating uses the algorithm of Bron and Kerbosch for clique finding. Therefore you want to supply an input file in the *graph* or *json* format. You can supply the maximum number of found cliques after the **-bk** flag (descending by the number of vertices of the clique). If you supply the number one, GraphDating will return only the largest found clique. You can also fasten the clique finding algorithm by applying a pivot element for the candidate choice. Therefore you use the flag **-p** and decide between the modes for maximum cardinality (**max**) or a random candidate (**random**).

If you want to find a clique from a graph and apply a pivot element of maximum cardinality, your command-line could look like this:

```
python3 GraphDating.py -i graph3.graph -bk -p max
```

If you want to find a clique from a graph in standard format and save the three largest found cliques with an own file name, your command-line could look like this:

```
python3 GraphDating.py -i graph6.graph -bk \
-sgo my_subgraph_name.graph -s 3
```

Modular product

To calculate the modular product of two graphs, you use the flag **-mp**. Therefore you want to supply two input files in the *graph* or *json* format and pass the argument for the calculation of the modular product. If you calculate the modular product of two identical graphs and afterwards run the bron-kerbosch function for clique finding, one of the output cliques will be the initial graph. But please note, that you can not deduce which part from the modular product was which part from the initial graph.

If you want to calculate the modular product of two graphs in *graph* format and save the output, your command-line could look like this:

```
python3 GraphDating.py -i graph2.graph graph3.graph \
-go mp_g2g3.graph -mp
```

For label comparison of vertices and edges you use the flags **-vlc** (vertices) and **-elc** (edges). You can use both vertex and label-comparison functions along or separately. If you do not use either of the flags, the labels will not be compared at all. If you just enter a flag, the labels will be checked for identity by default. You can also compare labels by your own comparison function.
For the application of a custom comparison function you also have to supply a *python* file (e.g. functions.py) and the name of the function. These functions take two labels as input parameters and returns a boolean value. By this means edges which labels do not comply (meaning one of the functions returns "False") are excluded from the result of the modular product. This is important in regard to graph alignments with Bron-Kerbosch algorithm, which uses the result to construct alignments.

The python script "functions.py" containing the compatibility functions for both vertex- and edge-labels could look like this, if you would allow solely edges, which have the same edge-label and same vertex-label for their start- and end-vertex respectively.

```
def vertex_compatibility(vertex_label1, vertex_label2):
    if vertex_label1 == vertex_label2:
        return True
    else:
        return False

def edge_compatibility(edge_label1, edge_label2):
    if edge_label1 == edge_label2:
        return True
    else:
        return False
```

If you want to apply your custom functions named "vertex-compatibility" and "edge-compatibility" within the python script "functions.py", your command-line could look like this:

```
python3 GraphDating.py -i graph2.graph graph3.graph -mp \
-go mp_g2g3.graph -elc functions.py vertex-compatibility \
-vlc functions.py edge-compatibility
```

Guide trees

To calculate a guide tree, you use the flag **-gt**. To save the result in *newick* format, you need to apply the flag **-no**. For guide tree construction you need to supply a list of input graphs and your choice of a comparison function, which are explained above in section 1.4 ("density", "pairwise_align", "custom"). The implemented default parameter for guide tree calculation is the graph density, but you can

also choose an alignment by greatest subgraph isomorphism or supply your own comparison function and pass it to GraphDating. Please note: It is not possible to pass an anchor for the calculation of a guide tree via "pairwise_align".

For the application of a custom comparison function you also have to supply a *python* file (e.g. example.py). The function has to take two graphs as input parameters and has to return a float number, which represents the calculated distance of the two graphs. A small float number constitutes a small distance (consequently a high similarity) and vice versa. The pseudocode for the comparison function in *python* is given below.

```
def my_comparion_function(graph1, graph2):
    calculate a float distance between graph1 and graph2
    return distance
```

If you want to calculate a guide tree with graph density as the comparison parameter and save the output, your command-line could look like this:

```
python3 GraphDating.py -i graph2.graph graph3.graph graph6.graph \
-gt density -no res_g2g3g6.newick
```

If you want to calculate a guide tree with your own comparison function, your command-line could look like this:

```
python3 GraphDating.py -i graph2.graph graph3.graph graph6.graph \
-gt custom my_comparison_function.py comparison_function_name \
-no res_g2g3g6.newick
```

Graph alignments

To make a graph alignment, you use the flag **-ga**. You can choose between the matching algorithms, which are explained above in section 1.5 ("bk" for matching by the modular product and clique finding, "mb" for matching by cordella algorithm). Therefore you want to supply a list of input files in the standard graph or *json* format. You also need to decide, if you want GraphDating to use your own guide tree, supplied in *newick* format or to calculate its own guide tree as described in 2.3 Guide trees above. Without any specifications GraphDating uses the density to calculate guide trees. The default value of the number of matchings per alignment is 1. Optionally, you can provide compatibility functions for both vertex and edge labels, analogous along the lines of the section **Modular product**.

If you want to make a graph alignment via the modular product and the Bron-Kerbosch algorithm by your own guide tree, your command-line could look like this:

```
python3 GraphDating.py -i graph2.graph graph3.graph graph6.graph \
-sgo -ga bk -gt my_guide_tree.newick
```

If you want to make a graph alignment via the Cordella algorithm and the guide tree is supposed to be calculated by your own comparison function, your command-line could look like this:

```
python3 GraphDating.py -i graph2.graph graph3.graph graph6.graph \
-gt custom my_comparison_function.py comparison_function_name \
-go res_g2g3g6.graph -ga mb
```

Unfortunately, there is no option regarding label-compatibility of both vertices and edges implemented yet for the Cordella algorithm.

Graph representation with Neo4j

To visualize graphs, you can do that via Neo4j by using the flag **-n**. Please note, that (strictly spoken) there are no undirected graphs in Neo4j. Undirected graphs

are represented by Neo4j by multi-di-graphs, which means that an undirected edge is represented as two reverse directed edges. There are two possibilities for the visualization of graphs via Neo4j. You can either download the Neo4j desktop version or use a Neo4J sandbox online.

The first option is to download Neo4j Dektop. Both here and in the online version you have to create a user profile. You can now create a project in the user interface and create a new local database using "Add Graph" and "Add local Graph" (we use version 3.5.9 for this). You choose a name and a password to start the database. Under "Manage - Details" you can now read the HTTP port you need to connect to. This information is passed to the program together with the password. The function call then looks like this: -n "http://localhost:7474" "neo4j" "1234" (Default user name is neo4j). (The easy way is to copy the infomation about the HTTP port from Figure 1). To see your uploaded data in the database open the Neo4j Browser.

Figure 1: Information about the HTTP port.

If you want to visualize graphs with the Neo4j desktop version, your command-line could look like this:

```
python3 GraphDating.py -i graph1.graph \
-n "http://localhost:7474" "neo4j" "1234"
```

The second possibility is to use Neo4j online. For this you have to log in on the website https://neo4j.com/sandbox-v2/. Now you launch a new "Blank Sandbook", because we want to upload our own data. You find all necessary connection data (Direct Neo4j HTTP, Username, Password) under "Details" (see Figure 2). This informtion have to be given to the function call. Then you can Launch the Browser under "Get Started".

If you want to visualize graphs with the Neo4j browser version, your command-line could look like this:

```
python3 GraphDating.py -i graph1.graph \
-n "HTTP" "USERNAME" "PASSWORD"
```

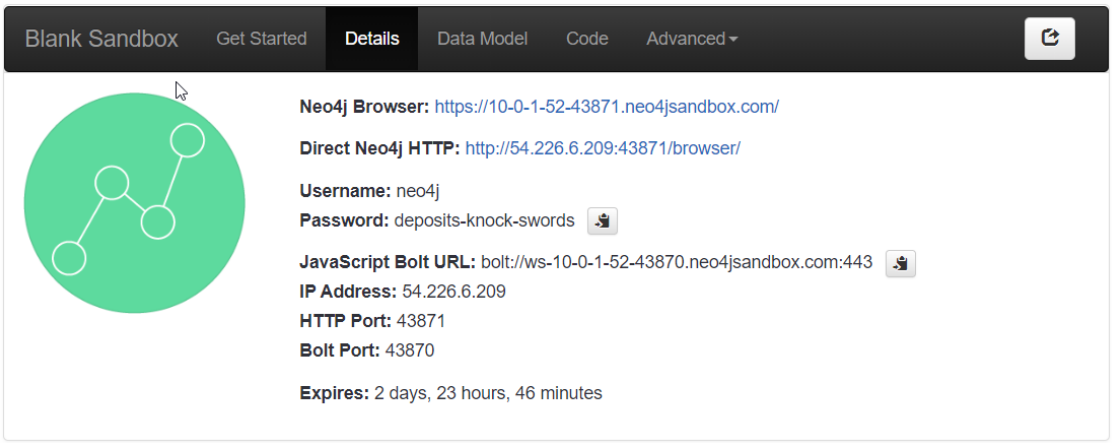

Figure 2: Example of the details description, where you find the Direct Neo4j HTTP, the username and the password, which you need to use the Neo4j browser version.

Other

There are some other useful functions implemented in GraphDating. They are described in the following section.

You can sort your output of graph alignments by supplying a custom sort function (similar to the custom function for the guide tree construction). Therefore you use the flag **-ms**. The function has to take a matching graph as input parameter and has to return a float number. A high float number constitutes a high priority and vice versa. The pseudocode for the comparison function in *python* is given below.

```
def my_sorting_function(matching_graph):
    calculate a float priority from matching_graph
    return priority
```

If you want to sort your output from clique finding, your command-line could look like this:

```
python3 GraphDating.py -i graph1.graph -bk -sgo \
-ms my_sorting_function.py
```

You can cut off matchings, which are redundant relating to stereo isomerism. Therefore you use the flag **-nsi**. If you align methane to ethane for example and use **-nsi**, GraphDating will return two matchings instead of 18 whereby 17 of them would be redundant to each other.

You can cut off matchings, which are not connected, by using the flag **-cc**.

You can use an own label comparison function by using the flags **-vlc** (vertices) and/or **-elc** (edges). Function takes two strings, return boolean

```
def my_label_function(label1, label2):
    decision if label1 and label2 match (True) or not (False)
    return decision
```

3 Profiling

In general, the vertical axis always is the time the commandline call needed. The relative number of calls, which run through without an error (e.g. MemoryError), are given in red above the bars in the diagrams. The error bars represent the standard deviation.

3.1 Comparison of pivoting methods of Bron-Kerbosch algorithm

Calculations were run on 30 randomly created graphs with 20 vertices and a connectivity of 0.5. The averaged results for random pivoting and pivoting with maximal cardinality are shown in Figure 3. The random pivoting approach is faster with the chosen graph composition.

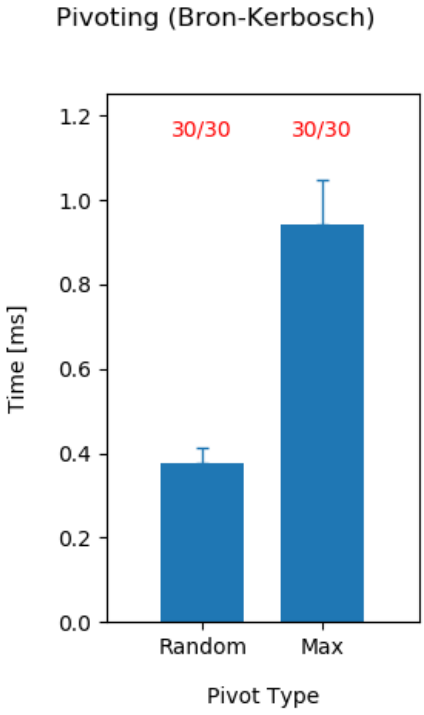

Figure 3: Comparison of pivoting methods of the Bron-Kerbosch algorithm for clique finding by time.

3.2 Runtime of graph alignments via Bron-Kerbosch algorithm

Calculations were run 30 times on respectively three randomly created graphs with equal number of vertices and a connectivity of 0.5. The averaged results for graph alignments with ascending number of vertices via Bron-Kerbosch algorithm are shown in Figure 4. The runtime increases with ascending number of vertices as expected.

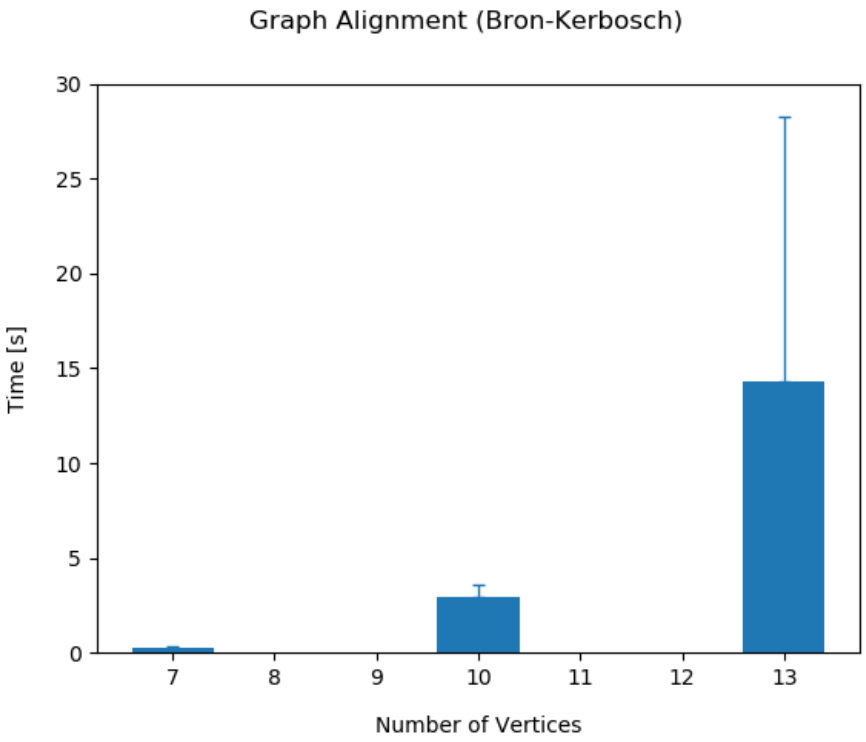

Figure 4: Comparison of runtimes of graph alignments via Bron-Kerbosch algorithm by the number of vertices of aligned graphs.

3.3 Runtime of graph alignments via VF2 algorithm

Calculations were run 30 times on respectively three randomly created graphs with equal number of vertices and a connectivity of 0.5. The averaged results for graph alignments with ascending number of vertices via VF2 algorithm are shown in Figure 5. The runtime increases with ascending number of vertices as expected. The graph alignment with VF2 algorithm also is much faster than the graph alignment with Bron-Kerbosch algorithm for a higher number of vertices. However, since the VF2 algorithm solves a different problem than the Bron-Kerbosch approach, this result has to be considered carefully.

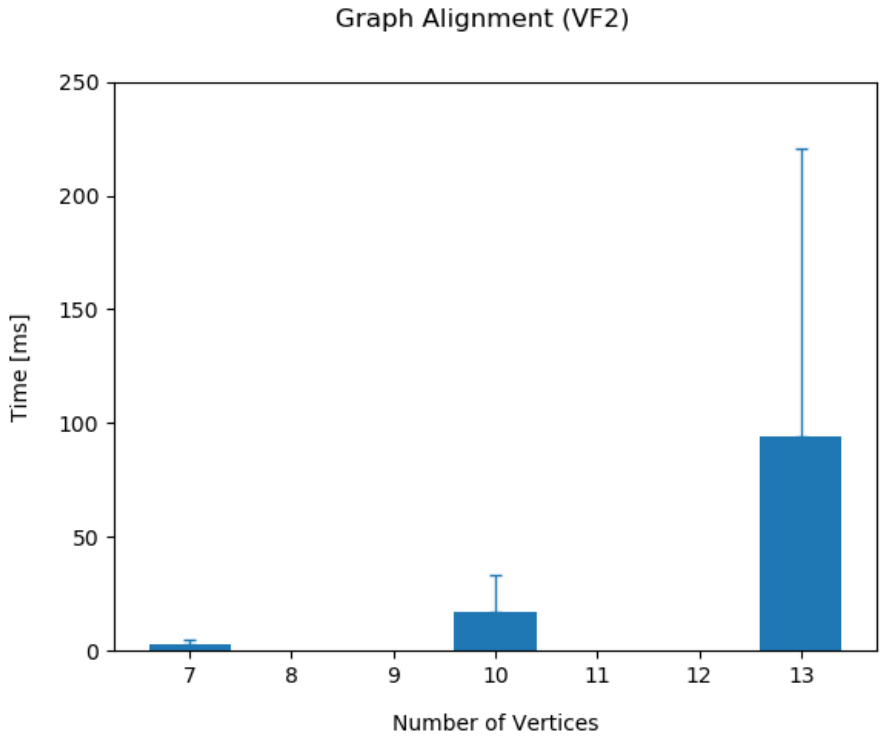

Figure 5: Comparison of runtimes of graph alignments via VF2 alogrithm by the number of vertices of aligned graphs.

References

[1] C. Bron and J. Kerbosch, Finding All Cliques of an Undirected Graph, Communications of the ACM, 16(9):575-577, ACM, 1973.

[2] L. P. Cordella, P. Foggia, C. Sansone and M. Vento, An Improved Algorithm for Matching Large Graphs, 2001.

A Help message

```
usage: Software for graph analysis

positional arguments:
  syntax

optional arguments:
  -h, --help            show this help message and exit

  -a , --anchor         Supply anchor graph file (path) for an anchor to the
                        first graph in the input graphs.NOTE: Anchor option is
                        not available for anchor graphs in json format!

  -bk [BRON_KERBOSCH [BRON_KERBOSCH ...]],
  --bron-kerbosch [BRON_KERBOSCH [BRON_KERBOSCH ...]]
                        Invokes maximal clique finding on input graph.
                        Optionally, you may pass a file name together clique
                        sorting function name in that file.

  -bm [BENCHMARK [BENCHMARK ...]], --benchmark [BENCHMARK [BENCHMARK ...]]
                        Writes benchmark-timestamps into file

  -cc, --check_connection
                        If selected, only connected subgraphs will be the
                        output of graph alignment!

  -ga [ [ ...]], --graph_alignment [ [ ...]]
                        Choose matching algorithm: Either 'bk' for bron-
                        kerbosch or 'mb' for matching-based. You may also
                        provide the number of matched subgraphs in previous
                        pairwise alignments on which the the alignment should
                        be expanded, i.e. the number of matchings that should
                        be forwarded to the next alignment step and/or output
                        (e.g. 'bk 5'). Default is one subgraph only. If
                        matching-based was chosen,you may provide a margin in
                        percent to what extend the smaller graph may be
                        reduced for subgraph isomorhpism search (e.g. 'mb 5
                        0.2').

  -go [ [ ...]], --graph_output [ [ ...]]
                        Saves graph as .graph file. If a path (in quotation
                        marks!) is provided, graph will be saved there. If a
                        .graph file name is provided, it will be saved with
                        that name in the current working directory. Else it
                        will be saved in the current working directory using
                        its name attribute.
```

```
-gt [ [ ...]], --guide_tree [ [ ...]]
                      Choose either a comparison function/attribute for
                      guide tree construction (Keywords for available graph
                      characteristics to base the comparison on: 'density'
                      for graph density, 'pairwise_align' for greatest
                      subgraph isomorphisms, i.e. maximal number of nodes in
                      subgraphs). Alternatively, pass a preconstructed guide
                      tree as '.newick' file or pass the keyword 'custom'
                      together with a .py file and the name of the
                      comparison function in that file. For detailed
                      requirements of the custom comparison functions see
                      the manual! If not selected, default is a heuristic
                      construction of a guide tree by graph density.NOTE:
                      For pairwise alignment add a second argument 'only'
                      ('pairwise_align' 'only'). Also, you need to pass the
                      '-ga' command line keyword and required parameters.

-i [ [ ...]], --input [ [ ...]]
                      Supply input path(s) of input file(s).

-if {graph,json}, --input_format {graph,json}
                      Specify type of input files. For .graph files pass
                      'graph' and for .json files pass 'json'.'graph' is
                      default. NOTE: Data in json files is supposed to be of
                      the structure of PubChem 2D json files!

-vlc [VERTEX_LABEL_COMPARISON [VERTEX_LABEL_COMPARISON ...]],
--vertex_label_comparison [VERTEX_LABEL_COMPARISON [VERTEX_LABEL_COMPARISON ...]]
                      For custom matching-conditions of vertex-labels, pass
                      a file name together with a function in that file that
                      will take two strings (see manual) and returns a
                      boolean value.If return is TRUE, two two distinct
                      VERTEX-objects with relative strings as labels are
                      possible matches.

-elc [EDGE_LABEL_COMPARISON [EDGE_LABEL_COMPARISON ...]],
--edge_label_comparison [EDGE_LABEL_COMPARISON [EDGE_LABEL_COMPARISON ...]]
                      For custom matching-conditions of edge-labels, pass a
                      file name together with a function in that file that
                      will take two strings (labels) and returns a boolean
                      value.If return is TRUE, two distinct EDGE-objects
                      with relative strings as labels are possible matches.

-mp, --modular_product
                      Forms the modular product of two graphs.
```

```
-ms [MATCHING_SORT [MATCHING_SORT ...]],
--matching_sort [MATCHING_SORT [MATCHING_SORT ...]]
                      For custom sorting of matching graphs, pass a file
                      name together with a function in that filethat will
                      take a matching-GRAPH object (see manual) and returns
                      a floating point value.Matching graphs will then be
                      sorted in descending order. Default is sorting by
                      descendingnumber of vertices!

-n NEO4J NEO4J NEO4J, --neo4j NEO4J NEO4J NEO4J
                      Visualize output using NEO4J!

-nh, --no_h_atoms     Specifies json format parsing. If selected, all
                      H-atoms will be neglected.

-no [], --newick_output []
                      Saves guide tree representation as Newick string to
                      .newick file. If a path (in quotation marks!) is
                      provided, it will be saved there. If a .newick file
                      name is provided, it will be saved with that name in
                      the current working directory. Else it will be saved
                      in the current working directory using a default name.

-nsi, --no_stereo_isomers
                      If selected and if input is a molecule graph, output
                      will be reduced neglecting multiple stereo isomers,
                      i.e. matchings where the subset of vertices is the
                      same for each inputgraph, respectively. Only one
                      stereoisomer will be forwarded!

-p , --pivot          Choose pivot mode: Either 'max' or 'random'.

-rg   , --random_graph
                      Create a random graph. Supply the number of vertices
                      (N, integer), the mean relative connectivity (p,
                      float) and whether it should be directed (either
                      'True' or 'False') as e.g. '10 0.8 True'.

-rc RANDOM_CLUSTER RANDOM_CLUSTER RANDOM_CLUSTER RANDOM_CLUSTER,
--random_cluster RANDOM_CLUSTER RANDOM_CLUSTER RANDOM_CLUSTER RANDOM_CLUSTER
                      Create a random clustered graph. Supply the number of
                      vertices (N, integer), the desireddegree at each node
                      (d, integer), how many vertices you want to delete
                      from the cluster(del_vert, int) and finally how mandy
                      edges you want to delete(del_edges, int)e.g. '20 3 2
                      2'.

-s, --seperate        Select if you like output to subgraphs of input
                      graphs.
```

```
-sgo [ [ ...]], --subgraph_output [ [ ...]]
                    Saves found subgraphs as .graph file. You may specifiy
                    the number of matchings in your output(e.g. -sgo 5),
                    default is all matchings. If a path (in quotation
                    marks!) is provided, subgraphs will be saved there
                    with an additional sequential number. If a .graph file
                    name is provided, they will be saved with that name in
                    the current working directory. Else they will be saved
                    in the current working directory using a default name.
```

B Example graphs

graph1.graph

The first example is derived from the chemical *benzoic acid*. The graph is not directed but its vertices are labeled as chemical elements and the its edges as the bond types single/double.

```
#nodes;15
#edges;15
nodes labelled;True
edges labelled;True
directed graph;False

1;O
2;C
3;O
4;C
5;C
6;C
7;C
8;C
9;C
10;H
11;H
12;H
13;H
14;H
15;H

15;1;s
1;2;s
3;2;d
2;4;s
4;5;s
5;14;s
5;6;d
6;13;s
6;7;s
7;12;s
7;8;d
8;11;s
8;9;s
9;10;s
9;4;d
```

graph2.graph

```
#nodes;6
#edges;10
nodes labelled;True
edges labelled;False
directed graph;True

1;a
2;b
3;c
4;d
5;e
6;f

1;2
2;1
2;4
4;2
4;1
1;4
4;3
3;4
1;5
1;6
```

graph3.graph

```
#nodes;10
#edges;22
nodes labelled;False
edges labelled;False
directed graph;True

1
2
3
4
5
6
7
8
9
10

1;10
10;9
9;10
9;1
1;9
9;3
3;9
3;1
1;3
1;4
3;4
4;2
1;2
2;1
1;5
3;2
2;3
5;6
8;6
6;7
7;8
7;2
```

graph4.graph

Attention! The graph was created to challenge the format parser.

```
#nodes;0
#edges;2
nodes labelled;False
edges labelled;False
directed graph;False

1;2
2;1
```

graph5.graph

Attention! The graph was created to challenge the format parser.

```
#nodes;0
#edges;0
nodes labelled;False
edges labelled;False
directed graph;True
```

graph6.graph

```
#nodes;3
#edges;3
nodes labelled;True
edges labelled;True
directed graph;True

1;a
2;b
3;c

1;2;a
2;1;b
2;3;c
```

graph7.graph

```
#nodes;4
#edges;8
Nodes labelled;True
Edges labelled;True
Directed graph;True

1;one
2;two
3;three
4;four

1;2;a
1;3;b
2;1;a
2;3;c
2;4;d
3;1;b
3;2;c
4;2;d
```