

Computing a Maximum Common Edge Subgraph of Two Molecular Graphs

Adrian Prinz

Bachelor thesis

Submission:	11.03.2020
Supervisor:	Prof. Dr. Gunnar Klau
Second Accessor:	Prof. Dr. Egon Wanke
Advisor:	Eline van Mantgem

Declaration

I hereby confirm that this thesis is my own work. I have documented all sources and tools used. Any direct or indirect quote has been marked as such clearly with specification of the source.

Düsseldorf, March 11, 2020

Adrian Prinz

Abstract

The comparison of two molecules is very relevant in different areas of chemistry and biology. In this thesis we compare two graphs that represent molecules by finding their Maximum Common Edge Subgraph (MCES). Our algorithm is based on an already known algorithm, which determines the MCES by finding the maximum clique in the common modular product graph. We use the Bron-Kerbosh-Algorithm to find these cliques. We then implement the algorithm and apply it to some example instances, which show that our algorithm works well, especially on small instances.

Contents

1	Introduction	1
2	Preliminaries	1
2.1	Molecules and Simplified Molecular Input Line Entry Specification	1
2.2	Undirected Labeled Graph	2
2.3	Linegraph	3
2.4	Modular Product Graph	3
2.5	Subgraph, Maximum Common Induced Subgraph and Maximum Common Edge Subgraph	4
3	MCES Algorithm	5
3.1	Explanation	5
3.2	Procedure	6
3.3	Maximum Clique Detection	7
4	Java Implementation	11
4.1	Chemistry Development Kit	11
4.2	Input	11
4.3	Representation of Molecules	11
4.4	Representation of Linegraphs	11
4.5	Representation of the Modular Product Graph	11
4.6	Bron-Kerbosh-Algorithm	12
4.7	Output	12
5	Results	12
5.1	General Information	12
5.2	Worst Bogus Smiles	14
5.3	Top Bogus Smiles	14
6	Discussion	17
6.1	Relationship between Inputgraphs and Modular Product Graph	17
6.2	Relationship between Modular Product Graph and Runtime	18
6.3	BKA2 versus BKA3	19
6.4	ΔY Exchange	19

<i>CONTENTS</i>	ii
6.5 Timeout	19
7 Conclusions	20
8 Future work	20
9 Acknowledgements	20
List of Figures	23
List of Tables	23
A Appendix	24
A.1 Output Worst Bogus Smiles	24
A.2 Output Top Bogus Smiles	28

1 Introduction

For many chemical and biological applications, it is essential to find out to what extent two given molecules match. These applications include, among others, the design of combinatorial syntheses [1], database searching [2], the prediction of biological activity [3] and the interpretation of molecular spectra [4]. In these contexts, chemical structures are often represented as graphs so that various operations can be performed on them.

In order to compare two given molecules the method of finding a so called Maximum Common Edge Subgraph (MCES) is often used. The MCES gives us the largest substructure related to the edges of two input graphs as a result. This thesis focuses on the derivation and subsequent implementation of an algorithm for detecting the MCES.

To find the MCES we use a common method, which has been proposed by Levi [5] and further developed by Koch [6]. As input we receive strings which represent molecules. We transform them to labeled graphs. Next, we generate corresponding linegraphs and create the modular product graph of both linegraphs. Subsequently, the maximum clique of that modular product graph has to be found.

This thesis provides a new, simpler way of finding the MCES by combining the creation of the modular product graph with subsequent maximum clique detection by the Bron-Kerbosh-Algorithm. There are many algorithms for finding the maximum clique, however, the Bron-Kerbosh-Algorithm provides a very simple and intuitive way to find the maximum clique [7]. This creates a new synergy which results in a simple algorithm for finding the MCES.

In Section 2 we explain the basic concepts of chemistry and graph theory that are necessary to understand the work. In Section 3 we explain how our algorithm and the Bron-Kerbosh-Algorithm work as an important part of our algorithm. Since we have implemented our derived algorithm, we explain the basic structure of our Java code in Section 4. In Section 5 we apply our algorithm to 18 pairs of molecules and subsequently discuss our results in Section 6. Section 7 concludes.

2 Preliminaries

2.1 Molecules and Simplified Molecular Input Line Entry Specification

In this thesis we focus on chemical structures called molecules. A molecule consists of atoms held together by bonds. For a more detailed description we refer to [8]. There are different ways to display molecules. We receive simple strings as input, which we convert into an undirected labeled graph to perform our arithmetic operations on.

These input strings, called Simplified Molecular Input Line Entry Specification (Smiles), consist of the element symbols of the associated atoms and a few special characters that represent the bonds between the atoms.

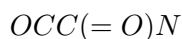


Figure 1: Smile

A simple example can be seen in Figure 1. The string is read from left to right. The Smile consists of the letters O, C, C, O and N, each of which specifies an atom. If there is no special character between two letters in the Smile, we are dealing with a simple covalent bond. An equal sign means that the two adjacent atoms are connected by a double covalent bond. If parts of the string are in parenthesis, it is a branch. For those who want to continue working on the topic, we refer to [9].

2.2 Undirected Labeled Graph

As mentioned above, we convert the given Smiles to undirected labeled graphs, on which we can run our algorithms.

Definition 1. A graph $G = (V, E)$ consists of a finite set of vertices V and a finite set of edges $E = \{\{v_i, v_j\} | (v_i, v_j) \in V, v_i \neq v_j\}$ connecting vertices of G . In an undirected graph the pairs of different vertices forming an edge are unordered, i.e., the tuples (v_i, v_j) and (v_j, v_i) represent the same edge.

By assigning the element of each atom to their corresponding vertex as a label we show which vertex refers to which atom. We describe each covalent bond by assigning its type to the corresponding edge as a label too.

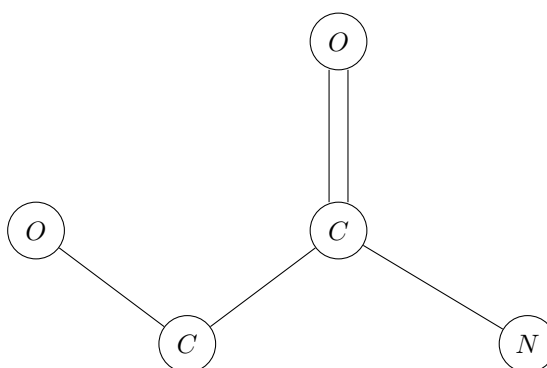


Figure 2: Undirected Labeled Graph

In Figure 2 we can see the graph which is created from the Smile in Figure 1. The letters of the Smile, which represent the atoms, become the vertices of the corresponding undirected labeled graph. If there is an equal sign in the Smile (in this example, between the

letters C and O), it is a double covalent bond, which is represented in the graph by two dashes and functions as an edge label.

2.3 Linegraph

In the course of our algorithm we will use linegraphs to convert edges to vertices.

Definition 2. In a linegraph $L(G) = (V', E')$ of graph $G = (V, E)$ the vertex set of $L(G)$ consists of all the edges of G , i.e., $V' = E$. Two vertices of $L(G)$ are adjacent, if the two corresponding edges are incident in G .

If G is a labeled graph, the vertices in V' receive the label of the corresponding edges in E and vice versa.

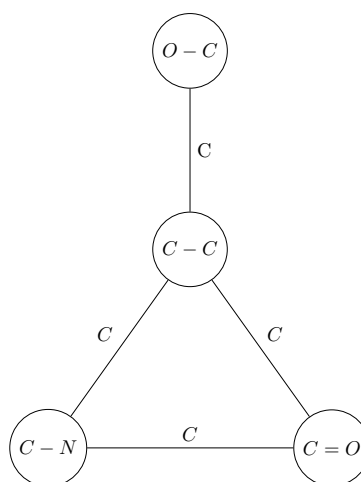


Figure 3: Linegraph

Figure 3 represents the linegraph belonging to Figure 2. It can be seen that each edge of the undirected labeled graph is now a vertex in the linegraph.

Two vertices are connected to each other by an edge with label C, since in the outgoing undirected labeled graph the two associated edges were incident by a C atom.

2.4 Modular Product Graph

We connect different graphs by creating a common modular product graph from both graphs. This step is necessary in order to subsequently recognize how the two associated graphs are similar.

Definition 3. A graph $G_{12} = (V', E')$ is the modular product graph of the two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. The vertex set V' is defined as the cartesian product of V_1 and V_2 . Two vertices (u_1, v_1) and (u_2, v_2) of G_{12} are adjacent if

- u_1 and u_2 are adjacent in G_1 and v_1 and v_2 are adjacent in G_2 and $w(u_1, u_2) = w(v_1, v_2)$, or
- neither u_1 and u_2 are adjacent in G_1 , nor v_1 and v_2 are adjacent in G_2

, where $w(u_1, u_2) = w(v_1, v_2)$ indicates that u_1 and v_1 have the same label, u_2 and v_2 have the same label and the edge $(u_1, v_1) \in E_1$ has the same label as $(u_2, v_2) \in E_2$.

Note that a vertex is not adjacent to itself.

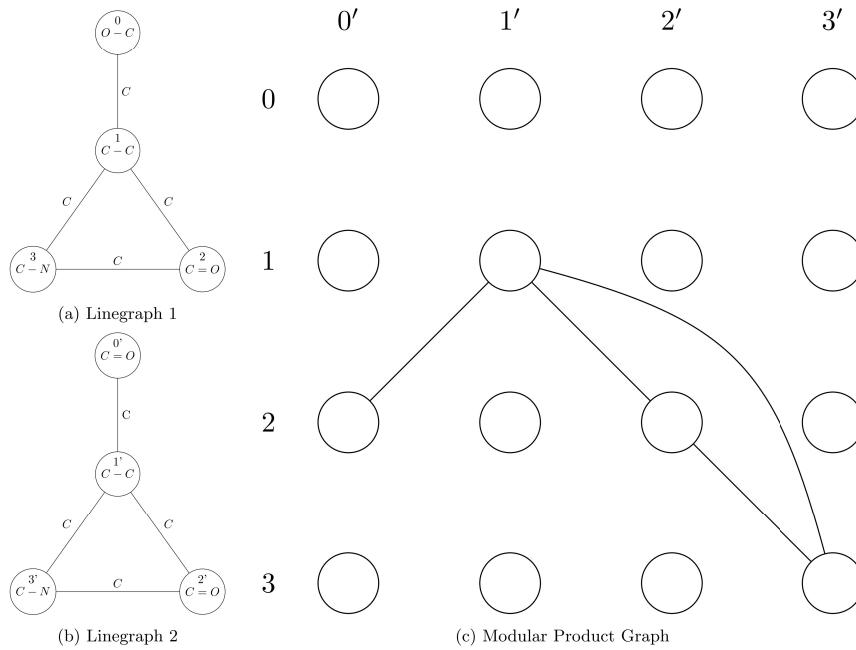


Figure 4: Modular Product Graph

In the example in Figure 4 you can see two linegraphs and their associated modular product graphs. Because of the fact that both linegraphs consist of four vertices, the modular product graph consists of 16 vertices. Since the vertex pairs $\{1, 1'\}$ and $\{2, 2'\}$ have the same labels and 1 is adjacent with 2 in $L(G_1)$ and $1'$ is connected to $2'$ in graph $L(G_2)$, the vertices $\{1_1, 2_2\}$ of the modular product graph are adjacent to each other. The same applies to the vertex pairs $\{2_2, 3_3\}$, $\{1_1, 3_3\}$ and $\{2_0, 1_1\}$.

2.5 Subgraph, Maximum Common Induced Subgraph and Maximum Common Edge Subgraph

Our goal of this work is to find an MCES, which is a special form of subgraphs.

Definition 4. A subgraph $G' = (V', E')$ of the graph $G = (V, E)$ is a graph in which $V' \subseteq V$ and $E' \subseteq E$.

Two types of subgraphs are important for our application: Firstly the Maximum Common Induced Subgraph (MCIS) and secondly the Maximum Common Edge Subgraph (MCES).

Definition 5. An MCIS between two graphs G_1 and G_2 is a subgraph containing the largest number of vertices G_1 and G_2 have in common.

Related to this is the MCES, with the difference that it considers the edges and not the vertices of the underlying graphs as a distinguishing feature.

Definition 6. An MCES between two graphs G_1 and G_2 is a subgraph containing the largest number of edges G_1 and G_2 have in common.

In this thesis we focus on disconnected MCES, i.e., the vertices of the resulting subgraph do not have to be connected by one path. The MCES is not unique which means there can exist more than just one solution of the problem.

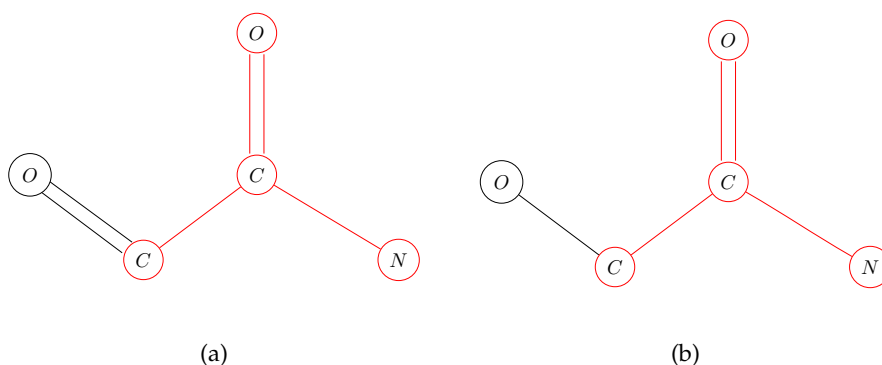


Figure 5: Maximum Common Edge Subgraph

In the example above in Figure 5, the MCES of the two molecules is marked in red in the respective undirected labeled graph.

3 MCES Algorithm

3.1 Explanation

In order to find the MCIS of the two input molecules, we have to find the maximum clique of their modular product graph as stated in [10], [11] and [5]. One speaks of a ΔY exchange if two linegraphs are isomorphic, although their original graphs are not isomorphic. Such a ΔY exchange can be seen in Figure 6. Although the two graphs in

a) are not isomorphic, their respective linegraphs in b) are isomorphic. Whitney [12] has proven, assuming that no ΔY exchange occurs, that an isomorphism of two linegraphs means that an edge isomorphism exists between the corresponding original graphs.

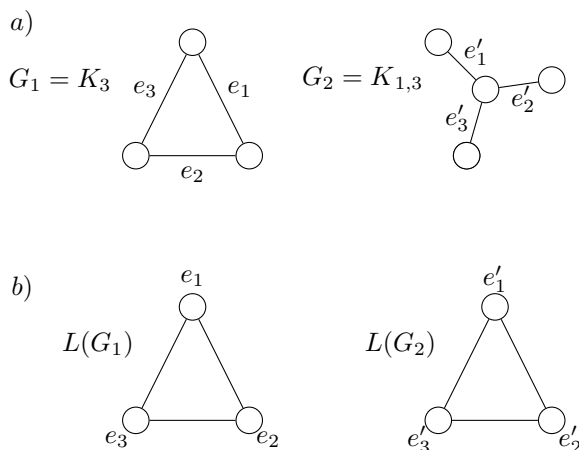


Figure 6: ΔY Exchange

However, since we are interested in an MCES instead of an MCIS, we would like to induce such an isomorphism. It follows from Whitney’s paper that if we first convert the input graphs to their respective edge graphs and then continue with the usual procedure (creating the modular product graph and finding the maximum clique), we will get an MCES[13][14].

3.2 Procedure

Our algorithm can be summarized as follows:

We receive two Smiles as our input, from which we create the corresponding undirected labeled graphs. Subsequently we generate the associated linegraphs which we combine by creating the modular product graph. As a result, we receive an undirected unlabeled graph representing the neighbourly relationship of the given molecules. Two nodes of the modular product graph are only adjacent if the corresponding vertices were adjacent either in both or in neither linegraph. It follows that the two vertices in the modular product graph are only adjacent if the neighbourhood relationship of the corresponding vertices in the two linegraphs is the same. Afterwards, we have to find the maximum clique in this modular product graph. For this we use the Bron-Kerbosh-Algorithm (BKA) because it is easy and intuitive to understand and implement. After we have found the maximum clique, we project it back onto the edges of the input molecules and get our MCES.

3.3 Maximum Clique Detection

3.3.1 Maximum Clique

As mentioned above, we use maximum clique detection to find the MCES.

Definition 7. A clique of graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$, so that every vertex of V' is adjacent to every other vertex in V' . A maximum clique is a clique to which no other vertex can be added without violating the clique property.

To find the MCES, we search for the maximum clique with the largest possible subset of vertices.

A maximum clique does not have to be unique, it is possible to find more than one maximum clique and therefore more than one MCES.

3.3.2 Complexity

Finding the maximum clique is a NP-complete problem. As early as 1972 it was one of Richard Karp's 21 original NP-complete problems. He proved this by reducing the NP-complete 3-SAT problem to the maximum clique problem in [15].

3.3.3 Bron-Kerbosh-Algorithm

The BKA is a recursive backtracking algorithm enumerating all possible maximum cliques of an undirected graph $G = (V, E)$. There are three forms of the BKA, which in increasing order become more complex but also faster. The basic version of the BKA reports all maximum cliques, since we are only looking for the maximum clique with the most vertices, we do not report all maximum cliques, but save the best result found so far.

Without pivoting

The algorithm receives three sets P , R and X as input. P contains all vertices of the graph that could possibly still be added to the maximum clique. R contains all vertices that belong to the clique under observation. X is the set of all vertices that cannot be added to the clique under consideration without violating the clique property. It follows that the initial method call is $\text{BKA1}(V, \emptyset, \emptyset)$.

The method calls itself recursively until $P = \emptyset$. If X is also empty at this point, R is a maximum clique. Otherwise the clique found is not maximal and is discarded.

The basic variant of the BKA finds maximum cliques of an undirected graph with n vertices in $\mathcal{O}(4^n)$ [16].

Algorithm 1: BKA1(P, R, X)

Input: Three sets of vertices: P, R and X
Output: Reports all maximum cliques

```

1 if  $P \cup X = \emptyset$  then
2   | Report R;
3 else
4   | foreach  $v \in P$  do
5     | BKA1( $P \cap \text{neighbours}(v), R \cup \{v\}, X \cap \text{neighbours}(v)$ );
6     |  $P = P \setminus \{v\}$ ;
7     |  $X = X \cup \{v\}$ ;
8   | end
9 end

```

Figure 7: Bron-Kerbosh-Algorithm without pivoting

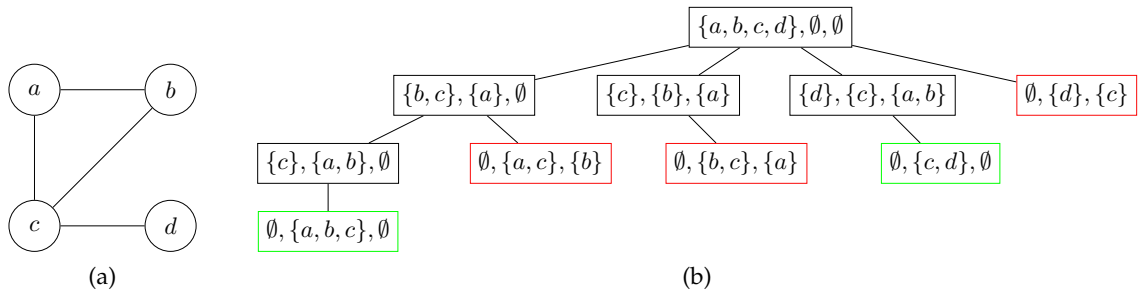


Figure 8: Recursion tree of BKA1 (Based on [16])

The recursion tree for calling BKA1 for the input graph in Figure 8a is shown in Figure 8b. The maximum cliques $\{a, b, c\}$ and $\{c, d\}$ are successfully found, as it can be seen in the green marked results. Nevertheless, there are many unsuccessful calls, which are rejected because they are non-maximum cliques (marked as red boxes). To reduce the number of these redundant calls, Bron and Kerbosh have already improved the BKA by introducing pivoting [17]. Tomita then optimized this strategy, which brings us to the BKA with pivoting [18].

With pivoting

In the first form of the BKA there are a lot of unnecessary recursive calls, in which P is the empty set at the end, but X still contains vertices, therefore R is not a maximum clique. It follows that the first form of the BKA is not suitable for graphs with many non-maximum cliques. Pivot selection is a way to reduce the number of redundant calls.

The vertex $u \in P \cup X$ in Figure 9 in line four is the "pivot vertex", which gives this method

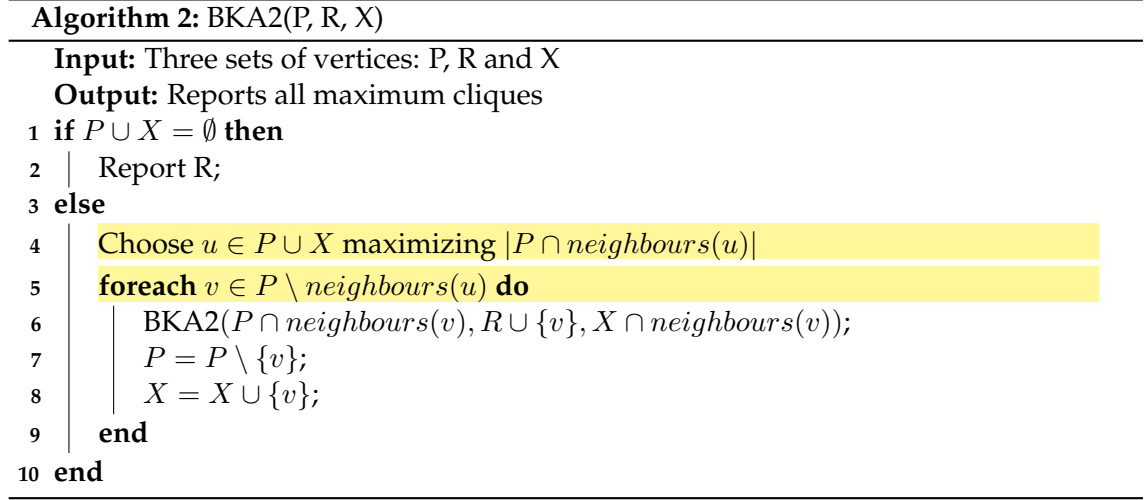


Figure 9: Bron-Kerbosh-Algorithm with pivoting

its name. It is not necessary to consider any neighbours v' of u in the for-loop because any cliques containing v' will be reported:

- Either through line five if $v = u$ and the resulting method call:
 $BKA2(P \cap \text{neighbours}(u), R \cup \text{neighbours}(u), X \cap \text{neighbours}(u))$
 Since v' is a neighbour of u , v' is considered in this method call.
- Or through line five if $v = w$, where $w \notin \text{neighbours}(u)$, but $w \in \text{neighbours}(v')$ and the resulting method call:
 $BKA2(P \cap \text{neighbours}(w), R \cup \text{neighbours}(w), X \cap \text{neighbours}(w))$
 Since v' is a neighbour of w , v' is considered in this method call.

These two cases cover all possible cases since $\text{neighbours}(v') = u + \sum w$.

U is chosen in such a way that $P \cap \text{neighbours}(u)$ is maximized, since through this as many nodes as possible do not have to be taken into account. The second variant of the BKA finds maximum cliques of an undirected graph with n vertices in $\mathcal{O}(3^{\frac{n}{3}})$. Depending on n , this is the best runtime to achieve, since there are a maximum of $3^{\frac{n}{3}}$ cliques in a graph [18].

It is shown in [6] that BKA2 comes to the same results as BKA1.

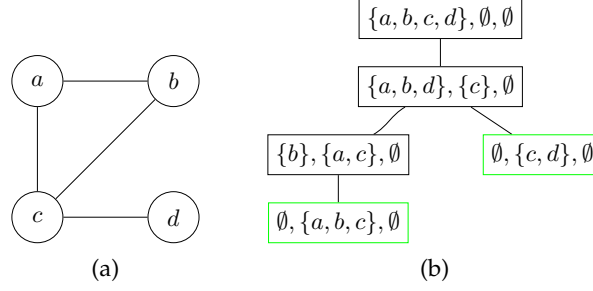


Figure 10: Recursion tree of BKA2 (Based on [16])

In Figure 10 you can see that the number of redundant calls has been significantly reduced compared to BKA1. In this example, the algorithm only calls relevant recursions that lead to a maximum clique. Instead of the previous ten calls in BKA1, the method is now only called five times, which represents a significant improvement. Despite this, there is an approach to sort the nodes according to their degree, which in some cases represents a further improvement of the BKA.

With vertex ordering

This form of the BKA is a special form of the BKA2, whereby all nodes are sorted in ascending order according to the size of the respective node degree before calling the BKA2.

Algorithm 3: BKA3(P, R, X)

Input: Three sets of vertices: P, R and X

Output: Reports all maximum cliques

- 1 Let $v_1, v_2, \dots, v_n \in V$ be a degeneracy ordering.
 - 2 **for** $i \leftarrow 1$ **to** n **by** 1 **do**
 - 3 $P = \text{neighbours}(v_i) \cap \{v_{i+1}, \dots, v_n\}$
 - 4 $X = \text{neighbours}(v_i) \cap \{v_1, \dots, v_{i-1}\}$
 - 5 BKA2($P, \{v_i\}, X$)
 - 6 **end**
-

Figure 11: Bron-Kerbosh-Algorithm with vertex ordering

The degeneracy of a graph G is the smallest number d such that every subgraph of G has a vertex with degree d or less.

The ordering of the vertices is called degeneracy ordering, where each node has d or fewer neighbours than all vertices after it. A degeneracy ordering is obtained in linear time by iteratively adding the vertex with the lowest degree to the degeneracy ordering and removing it from the graph [19]. Eppstein has also shown in [19] that the algorithm BKA3 has a runtime of $O(d3^{\frac{d}{3}})$. BKA3 is optimal for graphs with small degeneracy, i.e. for sparse graphs [20].

4 Java Implementation

4.1 Chemistry Development Kit

We use the Chemistry Development Kit (CDK) version 2.1.1 in our implementation. CDK is an open-source collection of modular Java libraries for processing chemical information. It can be found on Github¹.

4.2 Input

The main method can be found in the MoleculeMCES class. The program receives two arguments that serve as input. Firstly, the path of the CSV file that contains the Smiles is transferred. The second argument is an integer, indicating which line of the CSV file should be read as input.

4.3 Representation of Molecules

We use CDK to transform the Smiles into the CDK internal structure IAtomContainer. These containers preserve all information about the given molecule. We only filter out the information that is important to us in the classes Molecule, MyAtom and MyBond. MyAtom consists of its element symbol, a list of MyBonds it is incident with and an id in order to identify it easily. MyBond has its two incident MyAtoms, a string holding information of the bondtype and an id as class variables. The class Molecule holds two lists of all contained MyAtoms and MyBonds. The whole transformation from Smiles to our intern classes can be found in the class SmileSolver.

We chose this implementation because the three separate classes give us great flexibility to access individual molecular components. The respective classes only hold the information relevant to them, so that we can concentrate specifically on this molecular component.

4.4 Representation of Linegraphs

Since the vertices of $L(G)$ are equal to the edges of G , the vertices of our implementation are represented as a list of MyBonds. The bonds of the linegraph are represented by a list of LineBonds holding information which MyBonds they are incident with and the element symbol of the corresponding atom. In order to be able to easily find the neighbours of a vertex of the linegraph, we use a HashMap that has the vertex itself as a key and all its incident edges as value.

4.5 Representation of the Modular Product Graph

We represent our modular product graph using only a HashMap with a string as a key and a list of strings as value. Each vertex of the modular product graph is represented

¹<https://cdk.github.io/>

by its key. The string is created by concatenating the ids of the two nodes in the original molecules. All neighbours of a vertex are saved as a list in the corresponding value. We decide to represent the modular product graph in this way because of the following two reasons:

- Java is able to perform arithmetic operations very quickly on a HashMap. This is necessary because the following NP-complete problem is very computation-intensive and can take a lot of time.
- With the help of a HashMap it is very easy to see, which vertices are adjacent to which other vertices. This is the only important information for the subsequent execution of the BKA.

4.6 Bron-Kerbosh-Algorithm

We use the second and third form of the BKA because they are faster than the first form as explained in Chapter 9. Instead of reporting the maximum clique as described above, we save the largest clique found so far. We return a list of strings where each string is a member of the maximum clique of the modular product graph.

4.7 Output

Our output is created with the help of CDK. For this we map from the strings of the maximum clique to the original vertices of the linegraph (i.e. to MyBonds). Since it is a maximum clique, all of these vertices must be connected to each other, from which we derive the original MyAtoms. We form the CDK internal structure IAtomContainer out of these MyAtoms and MyBonds, with which we are able to illustrate our output in a graphic.

5 Results

5.1 General Information

To test our algorithm, we wrote a Java application as described in Chapter 4, which is available on Github².

We applied our algorithm to 18 different Smile pairs, nine of which should represent similar molecules (Top Bogus Smiles) and the other nine very different molecules (Worst Bogus Smiles). The almost similar ones can be found in the file called `top_bogus_smiles.csv`, the other ones in the file `worst_bogus_smiles.csv` on Github³ too. The algorithm ran on a computer with a 3.40 GHz Intel Core processor with 4 cores and

²<https://gitlab.cs.uni-duesseldorf.de/van.mantgem/molecule-comparison/tree/master/src/main/java>

³<https://gitlab.cs.uni-duesseldorf.de/van.mantgem/molecule-comparison/tree/master/data>

Id	Input		Runtime	
	Input 1	Input 2	BKA2	BKA3
1	<chem>CCCCCCCCCCCCC(=O)OC(=O)C1CCCN1</chem>	<chem>CCCCCCCCCCCCC(=O)NC1CCOC1=O</chem>	2046796	2300129
2	<chem>C1C2CN(CC1C3=CC=C(C(=O)N3C2)C4=CN=CC=C4)CC5=CC=C(C=C5)F</chem>	<chem>C1=CC=C2C(=C1)C=CC=C2NC(=O)C3=NN(C4=CC=CC=C43)CCCCCF</chem>	Timeout	Timeout
3	<chem>C1CC(N(C1)C(=O)C2CCCN2C(=O)CCCC3=CC=CC=C3)C(N)O</chem>	<chem>CC(CCCN1C=C(C2=CC=CC=C21)C(=O)NC(C(=O)N)C(C)(C)O</chem>	Timeout	Timeout
4	<chem>CC1CC(=O)CC2C1(C3CCC4(C(C3CC2)CCC4O)C)C</chem>	<chem>CC12CCC3CC(C(CC3(C1CCC(C2=C)O)C)O)(C)C=C</chem>	Timeout	Timeout
5	<chem>CC1=CC2=C(C=C1)NC(=C(C2=O)CN(C)C3CCN(CC3)C)C</chem>	<chem>CCCCCN1C=C(C2=CC=C(C=C21)C(=O)N3CCN(C3)C</chem>	12318462	13050463
6	<chem>CC1CC2CC(C3(C2(CCCN(CCC3)C(=O)OC(C)(C)C(C1)OCOC)O)O</chem>	<chem>CCCCCC(C=CC1C(CC(C1CC=CCCCC(=O)NC(CO)CO)O)O)O</chem>	Timeout	Timeout
7	<chem>CC(=O)NC1C(C(C(OC1NOC(=O)NC2=CC=CC=C2)C=O)O)O</chem>	<chem>C1CC(N(C1)N=O)C2=C[N+](=CC=C2)C3C(C(C(C(O3)C(=O)[O-])O)O)O</chem>	217306	245260
8	<chem>CC(C)(C)C(=NO)CNCC(=NO)C(C)(C)C</chem>	<chem>CC(C)CC(C(=O)NC(CC(C)C)C(=O)N)N</chem>	3335	2007
9	<chem>C1=CON(N1)NC2C(C(C(C(O2)CO)O)O)O</chem>	<chem>C(C(C(=O)NCC(=O)NC(CO)C(=O)O)N)O</chem>	2066	1297

Table 1: Worst Bogus Smiles

8.00 GB RAM. The used operating system was Microsoft Windows 10 Home.

We tried to find an MCES for all given instances. Since the program has a long runtime for some instances, we set a timeout after five hours and let us output the previously optimal result. For a more detailed analysis of the relationship between the instance and the associated runtime, we looked more closely at the number of edges of the input and output graphs, the number of vertices of the modular product graph and the average number of neighbours of each vertex of the modular product graph.

We have not checked for a ΔY exchange. However, since the probability of an exchange in 2-dimensional graphs is very low, this is not of great importance [21]. Although we have not done so, it should be done in the future, otherwise the results may be distorted.

Id	Input		Modular Product Graph vertices		Edges Output
	Edges Input 1	Edges Input 2	Vertices	Avg. neighbours	
1	21	21	441	88	18
2	32	31	992	74	(21)
3	28	27	756	70	(20)
4	25	24	600	236	(17)
5	25	25	625	58	19
6	32	30	960	154	(21)
7	26	27	702	25	18
8	16	16	256	24	10
9	18	16	288	6	9

Table 2: Worst Bogus Smiles Properties

5.2 Worst Bogus Smiles

In Table 1 all input Smiles of the Worst Bogus Smiles are listed with their associated program runtimes in milliseconds. An MCES was found in five of the nine given instances in less than five hours. We notice that in some instances the BKA without pivoting found a faster result, while in the other instances the BKA with pivoting was better. The fastest way to find a result was at instances eight and nine, namely in less than a minute.

As one can see in Table 2, all input graphs had between 16 and 32 edges. Those modular product graphs whose input graphs had only a few edges consist of relatively few nodes. While the smallest modular product graph consists of 256 vertices, the largest modular product graph consists of 960 vertices. The vertices of small modular product graphs usually have a small number of neighbours, but there are often deviations. The MCES consists of 9 to 21 edges, whereby a number of edges in brackets means that this is not the optimal result, but only the solution found until the timeout occurred.

5.3 Top Bogus Smiles

Table 4 shows that the MCES of the Top Bogus Smiles needs to be calculated much longer than the MCES of the Worst Bogus Smiles. Of the nine instances, only one MCES was found within the five hour timeout.

The input graphs of the Top Bogus Smiles are much larger than the ones of the Worst Bogus Smiles. While an input graph of the Worst Bogus Smiles still had an average of 24.4 edges, it is now an average of 32.8 edges. This means that the modular product graphs are now also much larger and each vertex of it has on average more neighbours. The MCES now consists of an average of 24.2 edges, instead of 17 as before. The only result that could be found before the timeout occurred was that from instance seven. It is striking that this instance has the smallest number in all properties considered from Table 5.

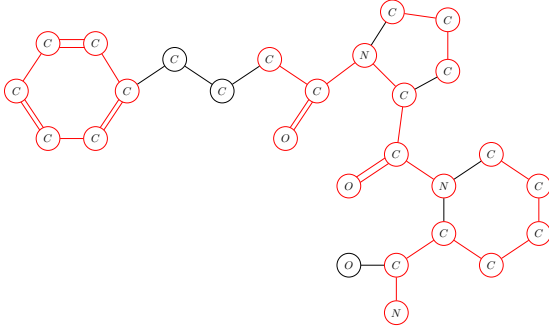
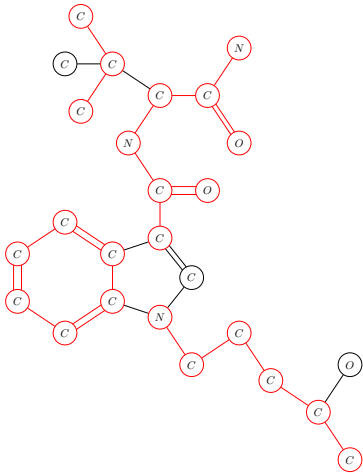
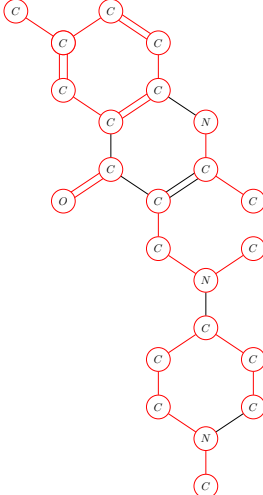
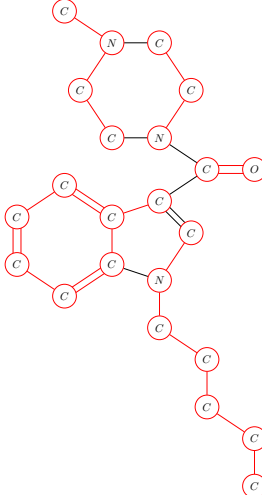
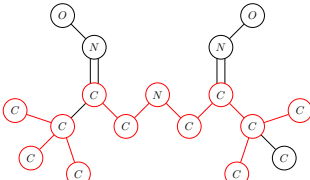
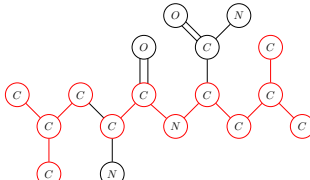
Id	Output 1	Output 2
3		
5		
8		

Table 3: Worst Bogus Smiles Output Examples

Id	Input		Runtime	
	Input 1	Input 2	BKA2	BKA3
1	<chem>CCCCCN1C=C(C2=CC=C(C=C21)C(=O)NC3=CC=CC4=CC=CC=C43</chem>	<chem>CCCCCN1C=C(C2=CC=C(C=C21)C(=O)NC3=CC4=CC=CC=C4C=C3</chem>	Timeout	Timeout
2	<chem>CCCCC=CCC=CCCCC CCC(=O)NCC(=O)C1=C NC2=CC=CC=C21</chem>	<chem>CCCCC=CCC=CCCCC CCC(=O)NCC(=O)C1=CC 2=CC=CC=C2N1</chem>	Timeout	Timeout
3	<chem>CN1CCCCC1CN2C=C(C3 =CC=CC=C32)C(=O)C4= CC=CC5=CC=CC=C54</chem>	<chem>CN1CCCCC(C1)N2C=C(C 3=CC=CC=C32)C(=O)C4= CC=CC5=CC=CC=C54</chem>	Timeout	Timeout
4	<chem>CCCCC(C=CC1C(CC(C1 CC=CCCCC(=O)NCCO)O)O)O</chem>	<chem>CCCCC(C=CC1C(CC(=O) C1CCCCCCC(=O)NCC O)O)O</chem>	Timeout	Timeout
5	<chem>CC1=CCC2=CC=C(N2)C 3=NC(C(O3)COC(=O)C (CC(=O)OC4C1OC(=O)C 4(C)CO)CC(C)C)C=C5 C(OCC5=C(C)C=CCOC) CC(C)C</chem>	<chem>CC1=CCC2=CC=C(N2)C 3=NC(C(O3)COC(=O)C (=CC(=O)OC4C1OC(=O))C4(C)CO)CC(C)C=C 5C(OCC5=C(C)CCCOC)CC(C)C</chem>	Timeout	Timeout
6	<chem>CC(=O)OC1C=CC2C3CC4 =C5C2(C1OC5=C(C=C4)OC(=O)C)CCN3C</chem>	<chem>CC(=O)N1CCC23C4C1C C5=C2C(=C(C=C5)OC) OC3C(C=C4)OC(=O)C</chem>	Timeout	Timeout
7	<chem>CC(=O)N(C1CCN(CC1) CCC2=CC=CS2)C3=CC= CC=C3</chem>	<chem>CCC(=O)N(C1CCN(CC1) CC2=CC=CS2)C3=CC=C C=C3</chem>	2823666	3236020
8	<chem>CCC(=O)N(C1CCN(CC1 C)CCC2=CC=CS2)C3=C C=CC=C3</chem>	<chem>CCC(=O)N(C1CCN(CC1)C(C)CC2=CC=CS2)C3= CC=CC=C3</chem>	Timeout	Timeout
9	<chem>C1=CC=C2C(=C1)C=CC= C2OC(=O)C3=CN(C4=C C=CC=C43)CCCCCF</chem>	<chem>C1=CC=C2C(=C1)C=CC= C2C(=O)C3=CN(C4=CC =CC=C43)CCCC(CF)O</chem>	Timeout	Timeout

Table 4: Top Bogus Smiles

Id	Input		Modular Product Graph vertices		Edges Output
	Edges Input 1	Edges Input 2	Vertices	Avg. neighbours	
1	30	30	900	94	(22)
2	33	33	1089	181	(22)
3	33	33	1089	119	(26)
4	28	28	784	151	(22)
5	57	57	3249	369	(28)
6	31	31	961	78	(25)
7	25	25	625	41	23
8	27	27	729	57	(25)
9	31	31	961	88	(25)

Table 5: Top Bogus Smiles Properties

6 Discussion

In the following chapter, we will focus on the Worst Bogus Smiles, since we received most of the relevant results before the timeout occurred. However, this discussion can be generalized to any input graph, especially to the Top Bogus Smiles.

6.1 Relationship between Inputgraphs and Modular Product Graph

In the following, we will consider to what extent there is a connection between the two input graphs and the resulting modular product graph.

From the definition of the modular product graph it follows that the number of vertices corresponds to the product of the number of edges of the two input graphs. From this follows: The larger the respective input graphs, the larger the resulting modular product graph.

A basic requirement for two adjacent vertices in the modular product graph is that the corresponding four vertices in the linegraph have the same label. For this reason, we have examined which edges occur most frequently in the two input graphs. We have found that in all instances the most common type of connection is a single covalent bond between two C atoms. If one now counts the cumulative number of edges of this type in the two input graphs, this gives a guideline with how many vertices a vertex in the modular product graph is adjacent on average.

Id	Number of single covalent C-Bonds	Avg. neighbours of the mpg
1	30	88
2	30	74
3	29	70
4	43	236
5	23	58
6	40	154
7	20	25
8	20	24
9	10	6

Table 6: Relationship between the number of C bonds in the input graph and adjacent neighbours in the modular product graph

This relationship is illustrated in Table 6. For those input graphs that have few C bonds, the resulting modular product graph is sparse and for those input graphs that have many C bonds, the resulting modular product graph is dense.

6.2 Relationship between Modular Product Graph and Runtime

In Table 2 you can easily see that the runtime depends both on the number of nodes and on the average number of neighbours per vertex. In general, the denser and larger the modular product graph, the longer it takes the program to find an MCES.

In order to be able to estimate the runtime only on the basis of the modular product graph, we divide the two properties into two groups:

	Small	Medium
Number of vertices	0 – 600	601 – ∞
Avg. neighbours	0 – 60	61 – ∞

Table 7: Categories for the properties of the modular product graph

We have derived the rule of thumb from this classification that if the modular product graph is small in one of the two properties, the MCES can be calculated successfully before the timeout. How fast the runtime is within these five hours depends on how close the respective values are to the limits. If one of the two values is very low and the other is high, this still results in a low runtime, as can be seen in instance seven.

However, if the value of a property is far higher than its limit, the runtime may lie outside the timeout, although the other property would have to be classified in the small category. This can be shown using the example of instance four: Although the number of vertices could be assigned to the category "Small", no result could be found within the timeout, since the number of average neighbours of each vertex corresponds to 236, which is four times the limit.

6.3 BKA2 versus BKA3

As can be seen in the Tables 1 and 4, the BKA with vertex ordering is only faster in instances 8 and 9 of the Worst Bogus Smiles than the BKA without vertex ordering. As already mentioned in Chapter 10, Eppstein has shown that the BKA with vertex ordering is only effective in graphs with a small degeneracy. This is also shown in Table 3. BKA3 was faster than BKA2 only in instance 8. One can see that the associated graphs are much thinner due to the lack of circles and thus also have a smaller degeneracy than the ones of instances 3 and 5.

6.4 $\triangle Y$ Exchange

We have not checked in our implementation whether a $\triangle Y$ exchange occurs. As a result, an incorrect MCES may be detected. As soon as we find an isomorphism between the linegraphs without checking, it is not guaranteed that this also results in an edge isomorphism between the two input graphs [12]. It follows that we think we have found a match between two edges of the input graph, even though they are not isomorphic at all. The detected MCES thus becomes larger than it actually is.

6.5 Timeout

Due to the frequent occurrence of timeouts, we switched off the timeout for some instances and ran it with the BKA with vertex ordering to see how long the instances take to output the optimal result. These experiments were performed on the HPC-System of the Heinrich Heine Universität Düsseldorf using 2 Intel(R) Xeon(R) E5-2667 v4 cores (3.20 GHz) as well as 4 GB RAM. We therefore tested the smallest instances where we did not have a result before the timeout. These were instance 3 of the Worst Bogus Smiles and instance 4 of the Top Bogus Smiles.

Instance	Runtime	Edges MCES before Timeout	Edges MCES without Timeout
3 (Worst Bogus)	92542714	(20)	20
4 (Top Bogus)	> 1 week	(22)	(22)

Table 8: Runtime of Timeout instances

We stopped calculating instance 4 of the Top Bogus Smiles after still not getting a result after a week. However, Table 8 shows that, at least for these medium-sized instances, the result on reaching of the timeout serves as a good heuristic, since the number of edges of the optimal MCES and that of the MCES found before the timeout are the same. However, due to the high runtime of 25.7 hours on the first and over a week on the second example, we decided not to consider any other instances without a timeout.

7 Conclusions

In this thesis, we combined two already known methods to develop an algorithm that finds the MCES of a chemical molecule. Then we implemented this algorithm.

The algorithm works by creating a common modular product graph from two input strings and finding the maximum clique in it.

The algorithm works well on relatively small input graphs, but takes a long time to find an MCES in large graphs. This makes sense since clique finding is an NP-complete problem. We have found that molecules that have few identical bonds can be calculated faster. Unfortunately, the calculation of large molecules took too long, which is why we set a timeout and used the best result so far. In the future, one could have the algorithm terminated without a timeout and compare these results with ours to see whether timeout is a good heuristic for finding the MCES.

8 Future work

In Chapter 6 we analyzed which properties of the input graphs lead to a short runtime. As can be seen in Tables 1 and 4, these properties unfortunately only apply to seven of the eighteen instances, and only in six of these we could find an MCES within a reasonable time.

The main component of the calculations is the BKA with pivoting or with vertex ordering. We chose this algorithm to solve the NP-complete problem, because on the one hand it is a new synergy of two already known algorithms and on the other hand the BKA is easy to understand and implement.

In the future, one can try to solve the NP-complete clique problem in a different way and maybe even use the molecular properties of the input. We particularly recommend the work of Gardiner and Willet [22], who use a Branch-and-Bound algorithm to find the cliques. They use upper and lower limits, which are derived from the properties of the respective molecules and are therefore optimally tailored to this problem. In addition, the authors have developed two pruning techniques that also work with the properties of the underlying molecules.

Another possibility to improve the runtime could be an iterative solution, which is processed in several threads at the same time. Since the BKA is a recursive algorithm, we were only able to run the program in a single thread.

Furthermore, it would also be a good extension of the present program to check whether there is a $\triangle Y$ exchange.

9 Acknowledgements

First of all, I would like to thank Eline van Mantgem, who has been at my side with advice all the time and has therefore made understanding, implementing and writing much easier for me. I would also like to thank Prof. Dr. Gunnar Klau, who made the thesis possible and always took the time to help me with major problems. I thank all of my fellow students who discussed with me and thus revealed further aspects to me.

References

- [1] Richard A Lewis, Jonathan S Mason, and Iain M McLay. "Similarity measures for rational set selection and analysis of combinatorial libraries: the diverse property-derived (DPD) approach". In: *Journal of chemical information and computer sciences* 37.3 (1997), pp. 599–614.
- [2] Peter Willett. "Matching of chemical and biological structures using subgraph and maximal common subgraph isomorphism algorithms". In: *Rational Drug Design*. Springer, 1999, pp. 11–38.
- [3] E Gifford et al. "Structure-reactivity maps as a tool for visualizing xenobiotic structure-reactivity". In: *Network Science* 2 (1996), pp. 1–33.
- [4] Lingran Chen and Wolfgang Robien. "Application of the maximal common substructure algorithm to automatic interpretation of ¹³C-NMR spectra". In: *Journal of Chemical Information and Computer Sciences* 34.4 (1994), pp. 934–941.
- [5] Giorgio Levi. "A note on the derivation of maximal common subgraphs of two directed or undirected graphs". In: *Calcolo* 9.4 (1973), p. 341.
- [6] Ina Koch. "Enumerating all connected maximal common subgraphs in two graphs". In: *Theoretical Computer Science* 250.1-2 (2001), pp. 1–30.
- [7] HC Johnston. "Cliques of a graph-variations on the Bron-Kerbosch algorithm". In: *International Journal of Computer & Information Sciences* 5.3 (1976), pp. 209–238.
- [8] Nenad Trinajstić. *Chemical graph theory*. Routledge, 2018.
- [9] David Weininger. "SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules". In: *Journal of chemical information and computer sciences* 28.1 (1988), pp. 31–36.
- [10] VG Vizing. "Reduction of the problem of isomorphism and isomorphic entrance to the task of finding the nondensity of a graph". In: *Proc. Third All-Union Conference on Problems of Theoretical Cybernetics*. 1974, p. 124.
- [11] Harry G Barrow and BURSTALL RM. "Subgraph isomorphism, matching relational structures and maximal cliques." In: (1976).
- [12] Hassler Whitney. "Congruent graphs and the connectivity of graphs". In: *Hassler Whitney Collected Papers*. Springer, 1992, pp. 61–79.
- [13] Victor Nicholson et al. "A subgraph isomorphism theorem for molecular graphs". In: *Graph Theory and Topology in Chemistry* 51 (1987), pp. 226–230.
- [14] V Kvasnicka and J Pospichal. "Maximal common subgraphs of molecular graphs". In: *Reports Molecular Theory* 1 (1990), pp. 99–106.
- [15] Richard M Karp et al. "Complexity of computer computations". In: *Reducibility among combinatorial problems* 23.1 (1972), pp. 85–103.
- [16] Jelmer Mulder. "Local Network Alignment by Enumerating Common Subgraphs".
- [17] Coen Bron and Joep Kerbosch. "Algorithm 457: finding all cliques of an undirected graph". In: *Communications of the ACM* 16.9 (1973), pp. 575–577.

- [18] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. "The worst-case time complexity for generating all maximal cliques and computational experiments". In: *Theoretical computer science* 363.1 (2006), pp. 28–42.
- [19] David Eppstein, Maarten Löffler, and Darren Strash. "Listing all maximal cliques in sparse graphs in near-optimal time". In: *International Symposium on Algorithms and Computation*. Springer. 2010, pp. 403–414.
- [20] David Eppstein and Darren Strash. "Listing all maximal cliques in large sparse real-world graphs". In: *International Symposium on Experimental Algorithms*. Springer. 2011, pp. 364–375.
- [21] John W Raymond and Peter Willett. "Maximum common subgraph isomorphism algorithms for the matching of chemical structures". In: *Journal of computer-aided molecular design* 16.7 (2002), pp. 521–533.
- [22] John W Raymond, Eleanor J Gardiner, and Peter Willett. "Rascal: Calculation of graph similarity using maximum common edge subgraphs". In: *The Computer Journal* 45.6 (2002), pp. 631–644.

List of Figures

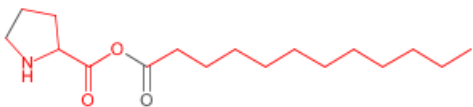

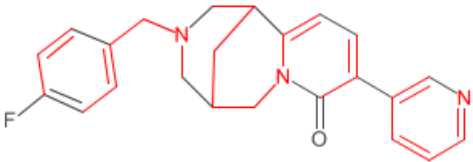
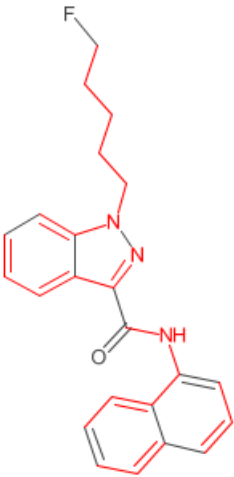
1	Smile	2
2	Undirected Labeled Graph	2
3	Linegraph	3
4	Modular Product Graph	4
5	Maximum Common Edge Subgraph	5
6	ΔY Exchange	6
7	Bron-Kerbosh-Algorithm without pivoting	8
8	Recursion tree of BKA1 (Based on [16])	8
9	Bron-Kerbosh-Algorithm with pivoting	9
10	Recursion tree of BKA2 (Based on [16])	10
11	Bron-Kerbosh-Algorithm with vertex ordering	10

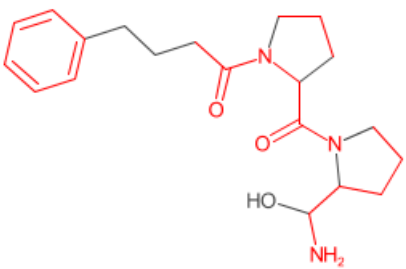
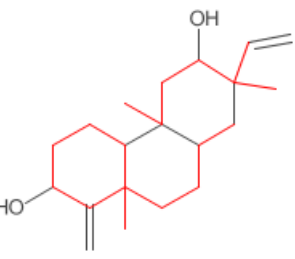
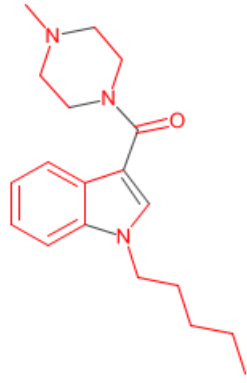
List of Tables

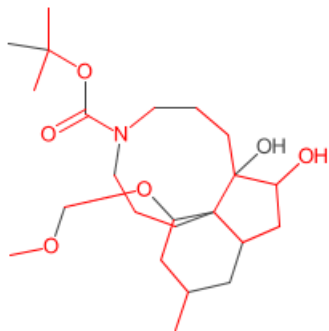
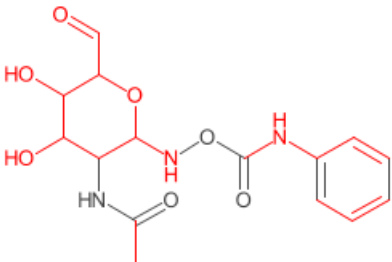
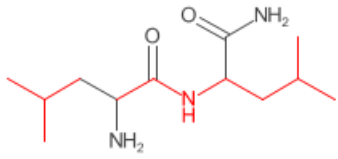
1	Worst Bogus Smiles	13
2	Worst Bogus Smiles Properties	14
3	Worst Bogus Smiles Output Examples	15
4	Top Bogus Smiles	16
5	Top Bogus Smiles Properties	17
6	Relationship between the number of C bonds in the input graph and adjacent neighbours in the modular product graph	18
7	Categories for the properties of the modular product graph	18
8	Runtime of Timeout instances	19

A Appendix

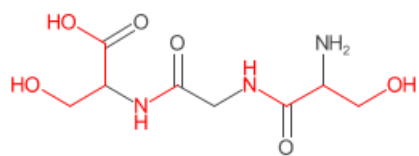
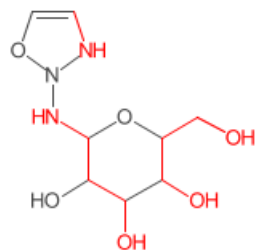
A.1 Output Worst Bogus Smiles

Id	Output 1	Output 2
1		
2		

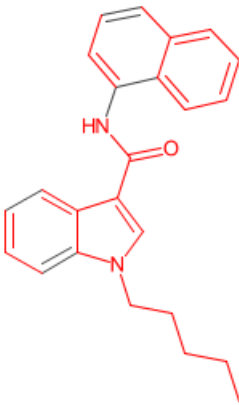
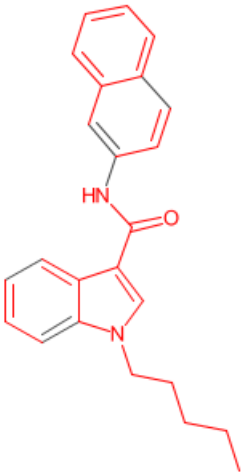
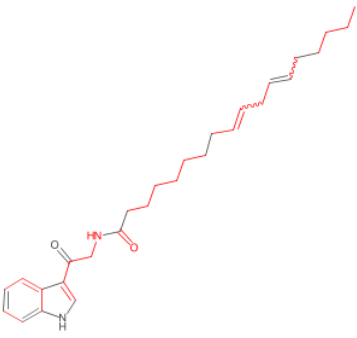
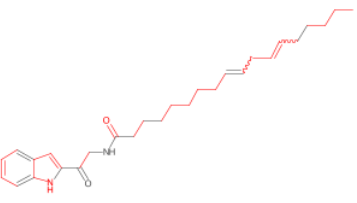
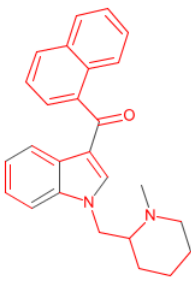
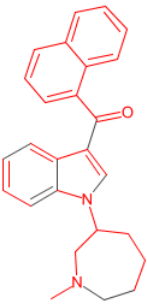
3	 <chem>O=C(CCN1CCCC1)N2CCCC2C(O)N</chem>
4	 <chem>CC(C)(O)CCN1CCCC1C(O)C2CCCC2</chem>
5	 <chem>CCN1CCCC1C(O)C2CCCC2C(O)C3CCCC3</chem>

6	
7	
8	

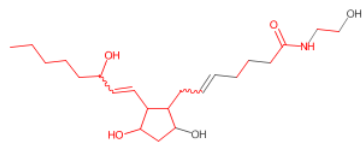
9



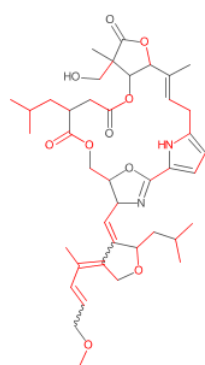
A.2 Output Top Bogus Smiles

Id	Output 1	Output 2
1		
2		
3		

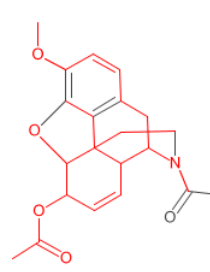
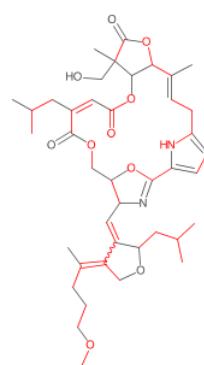
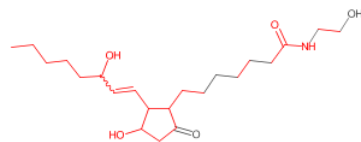
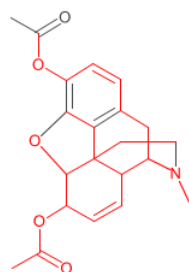
4

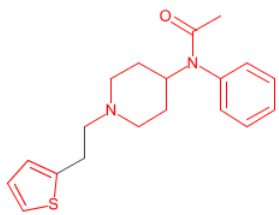


5

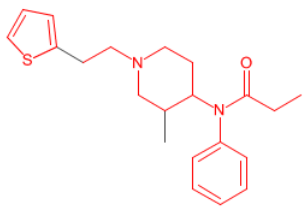
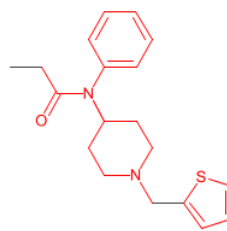


6

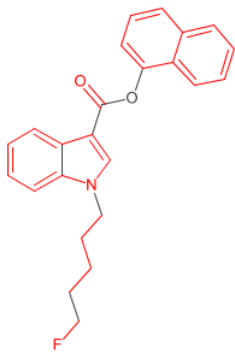
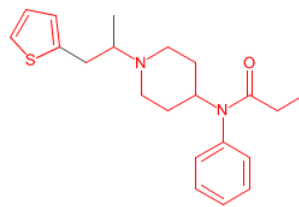




7



8



9

