The Maximum Common Subgraph Problem in Systems Chemistry

Bachelor Thesis

Kasper Halkjær Beider & Tobias Klink Lehn

Supervisors: Daniel Merkle & Akbar Davoodi

Department of Mathematics and Computer Science
University of Southern Denmark
Denmark
1st of June, 2023

GIT: https://github.com/TKL00/mcs_bachelor

Illustration by Tobias Klink Lehn

Resumé

Metabolske netværk kan modelleres som hypergrafer, hvor enhver hyperkant svarer til en reaktionstype for en given mængde af molekyler, som alle reagerer med et bestemt enzym. Fælles for disse molekyler i en given reaktion er, at de alle har en fælles mængde bindinger, som ændrer sig undervejs i reaktionen - i denne sammenhæng kaldt et *anker*. Et molekyle i sig selv kan repræsenteres som en uorienteret graf med dekorerede kanter og knuder. En reaktion af et molekyle kan derfor ses som en graftransformation, som er resultatet af en anvendt graftransformationsregel. Venstresiden af disse graftransformationsregler svarer da til reaktantsiden, og højresiden svarer til produktsiden. Ændringen mellem vesntre- og højresiden i sådanne regler kendes som ankeret, som er fælles for alle molekylerne i den givne reaktion - at udvidde dette anker svarer da til at løse største fælles delgraf-problemet. I denne rapport beskriver og implementerer vi to forskellige algoritmer, som kan løse største fælles delgraf-problemet, og diskuterer deres anvendelse på udviddelsen af ankeret. Den første algoritme er præsenteret af J. McGregor, som benytter en søgetræsmetode til at bygge en fælles delgraf op mellem to grafer. Den anden algoritme er formuleret af A. Davoodi, som benytter sig af linjegrafer, produktgrafer og kliker til at finde alle maksimale delgrafer blandt en mængde af grafer. Det viser sig, at McGregors metode ikke overføres særligt nemt til udviddelsen af ankeret, fordi det ikke er alle maksimale løsninger, der findes, hvilket giver problemer når der er mere end to molekylegrafer. Til gengæld viser klikemetoden sig at have god virkning på tre til fire grafer, men at den for nogle input stadigvæk har udfordringer, og vi præsenterer mulige forbedringer, som kan afhjælpe dette.

Abstract

Metabolic networks can be modelled as hypergraphs in which every hyper edge corresponds to a reaction type. Each reaction type consists of a given set of molecules, all of which react with a specific enzyme. All molecules in a given reaction share a common set of chemical bonds that change during the reaction - this set of edges is denoted as the anchor. A molecule in itself can be represented as a undirected graph with decorated edges and vertices. A reaction of a molecule can thus be seen as a graph transformation, namely the result of an applied graph transformation rule. The left-hand side of these graph transformation rules corresponds to the reactants and the right-hand side corresponds to the products. The change between the reactants and products is known as the anchor common to all molecules in a given reaction, and extending this anchor is known as finding the maximum common anchor extension. In this thesis, we describe and implement two different algorithms meant to solve the maximum common subgraph problem and discuss their application on extending the anchor. The first algorithm is described by J. McGregor which utilizes a backtracking search tree approach to build a common subgraph of two graphs. The second algorithm is described by A. Davoodi and utilizes line graphs, modular products and cliques to find all maximal common subgraphs of a set of graphs. It turns out that McGregor's approach does not transfer well to the problem of extending the anchor as not all maximal solutions are found which is problematic when the input consists of more than two molecule graphs. On the other hand, the clique method proves to be usable on three and four graphs although some input graphs still present issues. We discuss several optimization choices that might alleviate this.

Contents

1 Introduction

Mathematical modelling of life sciences has seen an increase in interest during the last half-century, especially in regard to computational biology and biomedicine [FKW22]. Perhaps the most relatable one for most students with experience in discrete mathematics will be the modeling of chemical molecules, viewed as undirected graphs with decorations on the vertices and edges. A reaction of molecules can be seen as a graph transformation. That is, given a graph transformation *rule* (pattern), one can apply such a rule to a molecule, rewriting parts of the molecule's underlying graph which produces a new molecule. A graph transformation rule is much like a reaction template but requires an atom mapping from reactants to products and strict mass conservation. Conceptually this is similar to a generalisation of formal language theory, where production rules rewrite parts of a graph. The graph transformation formalism was utilized in the development of the software package MØD [And+16].

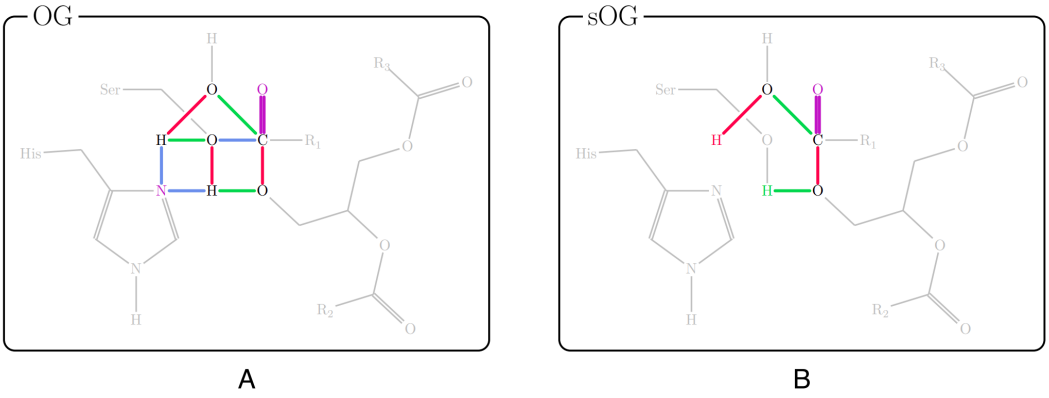


Figure 1: An example of an overlay graph depicting reaction mechanisms. Red edges depict bonds that a broken, green ones depict bonds that are created. If red overlays with green, the bond is blue. On the other hand, if green overlaps with red, the bond is purple. For detailed explanation of the reaction mechanisms, see [And+22].

In order to better illustrate the mechanisms used in graph transformation, the term "overlay graph" was established in [And+22]. The overlay graph is a colored "overlay" of the underlying, changing molecule and nicely illustrates

the change of state (change of bonds) of the molecules during a mechanism step. For an example of such a representation, see Fig. 1 from [And+22]. This type of diagram will not be used in the scope of this thesis but is highlighted to indicate the progress in mechanism modelling. On top of this, the idea behind illustrating the change of state in molecules is similar to the idea behind unioning the changing edges in the DPOs which we describe shortly.

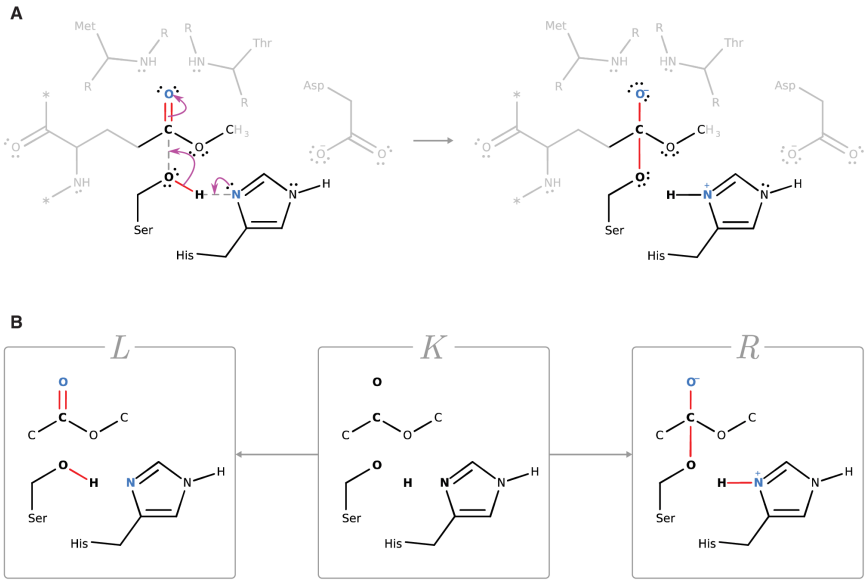


Figure 2: An example of the extraction of a double pushout diagram from a reaction step. The A-part shows an enzymatic reaction happening inside an enzymatic binding site. The B-part shows the double pushout diagram, illustrating the transformation from educt to product of the compound binding to the enzyme.

In recent years, modelling of metabolic networks - collections of enzymatic reactions[Con23] - have been modelled as hypergraphs, each hyperedge corresponding to one reaction type with several molecules into their designated products[AMR19]. That is, within metabolic networks, multiple graph transformation rules are applied to many graphs simultaneously. Applications of graph transformation rules can be modelled through the use of double pushout diagrams (DPOs) see [HMP01]. An example of a DPO can be seen on the B-part of Fig. 2 taken directly from [And+21]. The red edges in the DPO are the edges that change throughout the reaction step, and the union of these

edges compose the *anchor*. Given several of these DPOs for several molecules that react in the same way with an enzyme, we wish to find the common *context* of each of the molecules, namely the part of all molecules that *must be* present in the enzymatic reaction. In Fig. 2, we see grayed out molecules - amino acids - of the binding site and we know that these must always be present. They are therefore trivially common and can be ignored. Instead, we focus on the compounds reacting with the enzymes and find the connected extension of their known anchor. When viewing molecules as graphs, this raises the computational question of finding the maximum common subgraph of all molecules within a specific reaction type, some of which can contain more than 50 molecules. This sparks the purpose of this thesis. The maximum common subgraph (MCS) problem is known to be NP-complete [GJ79] and the runtime of any MCS algorithm is thus expected to increase exponentially as the size of the input graphs increases. In this thesis, we will consider two approaches for finding the maximum common subgraph extension of metabolic reactions. The first one is a naïve approach developed by James J. McGregor in 1982 [McG82] which utilizes a backtracking search tree methodology. The second approach is based on clique finding in modular products [BB76]. Both approaches will be implemented in Python3, and will be compared empirically. McGregor's approach will be used to illustrate the problem of finding a common subgraph, but the main focus will be on presenting the modular product approach and its usability in real life applications. As the very first thing, we introduce the necessary graph theory to understand MCS (Sect. 2). The algorithmic description immediately follows for McGregor (Sect. 3) and our own approach (Sect. 4). Following the implementation details, we present the results of the tests on both algorithms (Sect. 5) which is followed up by a discussion (Sect. 6) ended by a final conclusion (Sect. 7).

2 Graph Theory

In this section, we present the relevant concepts from graph theory used in this project, eventually introducing the heart of the matter, namely the maximum common subgraph problem. The definitions and theorems are inspired by [Wes+01].

2.1 Graphs

Definition 2.1 (undirected graph) *An undirected graph $G = (V, E)$ is a 2-tuple consisting of V, the set of vertices*[1] *in G, and E, the set of edges connecting the vertices in G. An edge $e = uv \in E$ is an unordered vertex pair for $u, v \in V$.*

Note that in our examples, we will only consider *simple* undirected graphs, i.e. graphs without loops (edges exiting and entering the same vertex) and parallel edges (edges with the same endpoints). For an example of a simple undirected graph, see Fig. 3.
If $uv \in E$, we say that u and v are connected or, equivalently, u is adjacent to v. Note that for a *directed graph*, E consists of ordered pairs. (u, v) - such graphs, however, will not be considered further in this project. An object that we will consider, however, is the adjacency matrix for a given graph as it provides useful information about the structure of a graph which will prove beneficial in the algorithms that we present later in this project.

Definition 2.2 (adjacency matrix) *Let $G = (V, E)$ be a graph with n vertices. The adjacency matrix, A_G, is an $n \times n$ matrix. If $ij \in E$, we have $A(i, j) = 1$, otherwise $A(i, j) = 0$ where $A(i, j)$ denotes the i-th row and j-th column in A.*

A graph as defined above, even without additional information, is indeed a powerful theoretical mathematical tool. However, for practical uses such as in chemical modelling where nodes can represent atoms and edges can represent

[1] Vertices are also referred to as nodes or points and the terms are used interchangeably throughout this text.

covalent bonds, having the ability to "decorate" graphs can be quite useful. For this purpose, we define labelled graphs.

Definition 2.3 (labelled graph) *A labelled graph $G = (V, E, l_V, l_E)$ is a 4-tuple consisting of V, the set of vertices in G, E, the set of edges connecting the vertices in G and two functions $l_V : V \to L_1$ and $l_E : E \to L_2$ that map vertices and edges respectively to labels from the arbitrary label sets L_1 and L_2.*

In graph theory, a labelled graph usually refers to a graph where node labels are *unique* (e.g. numbers from 1, 2, 3 and so on). We will abuse the notation of a labelled graph such that vertices/edges may share labels, i.e. l_V and l_E are not injective functions. Additionally, unless l_V and l_E are specified, one can safely assume that the graphs at hand are unlabelled. In this thesis, figures will sometimes include indices on nodes and/or small letters on edges for pedagogical purposes, but these do not necessarily correspond to labels in this regard, see Fig. 3b.

(a) (b)

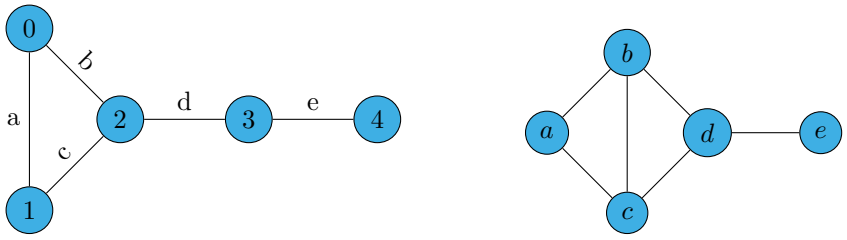
Figure 3: (a) An example of an simple undirected graph, G, with four vertices and five edges. (b) The same graph with pedagogical indices on the nodes.

These are the basic definitions of graphs used in this project. We will later focus on finding edge correspondences between several graphs, and for that we need to introduce the concept of line graphs, cliques and product graphs. It will be clear how these graph structures can be useful tools for finding edge correspondences once we establish the differences between vertex induced common subgraphs and edge induced common subgraphs, both of which will be described in detail later in this section.

Definition 2.4 (line graph) *The line graph of a graph $G = (V, E)$ is a graph $L(G) = (E, F)$ where $e_1e_2 \in F$ if and only if $e_1 = uv \in E$ and $e_2 = vw \in E$*

share an endpoint v.

An example of a graph and its line graph is illustrated in Fig. 4. When constructing the edges in $L(G)$, one must look at incidence of edges in G. Namely, if two edges $e_1, e_2 \in E$ are incident - that is, e_1 and e_2 have one node in common - the corresponding edge e_1e_2 exists in F of $L(G)$.



(a) G (b) $L(G)$

Figure 4: Example of a graph, G, and its corresponding line graph $L(G)$. Notice how the edges in G correspond to the nodes in $L(G)$. Also consider the fact that node a in $L(G)$ is adjacent to node b and c in $L(G)$ because edge a in G is incident to edges b and c in G.

Definition 2.5 (clique) *A clique of size k in a graph $G = (V, E)$ is a set of k nodes $\{v_1, v_2, \ldots, v_k\} \subseteq V$ such that v_i is adjacent to v_j, for every i, j; $1 \leq i < j \leq k$.*

Following the definition of a clique, we say that a graph G contains a k-clique if there exists a subset of nodes $S \subseteq V$ where $|S| = k$ such that all nodes in S are mutually adjacent. We say a clique is *maximal* if the clique is not a proper subset of any other clique. As an example, observe that G in Fig. 4a contains a 3-clique consisting of the nodes 0, 1 and 2.

Definition 2.6 (modular product graph) *Considering two graphs $G = (V, E)$ and $H = (V', E')$, the modular product of G and H denoted by $G \times H$ consists of the vertex set $V \times V'$. Two vertices (u, u') and (v, v') in the modular product are adjacent if and only if the following two conditions hold:*

1. $u \neq v \wedge u' \neq v'$
2. $uv \in E \wedge u'v' \in E'$ ***or*** $uv \notin E \wedge u'v' \notin E'$.

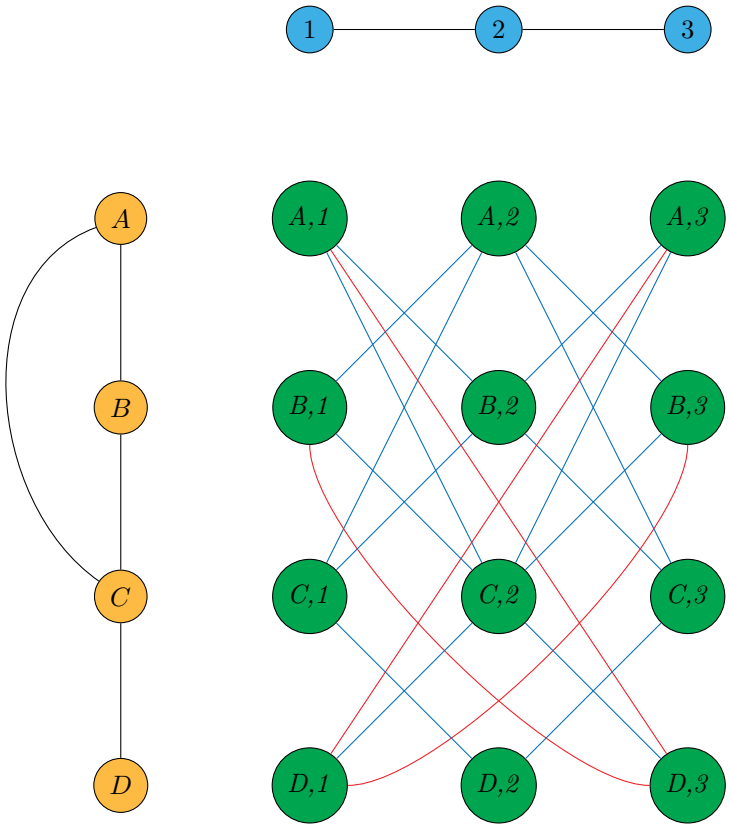


Figure 5: An example of a modular product graph, where the edge colors are purely pedagogical and not actual labels. Notice that edge $(A,1)(B,2)$ is blue because A is adjacent to B and 1 is adjacent to 2. Similarly, the edge $(B,1)(D,3)$ is red because B and D are non-adjacent, and the same goes for 1 and 3. Lastly, there is no edge $(C,1)(D,3)$ because C and D are adjacent, but 1 and 3 are not adjacent.

As is given by its name, the modular product graph contains a node for each element in the cartesian product of the two vertex sets of G and H. We will be using both *modular product* and *product graph* to refer to this construction.

To illustrate the meaning of condition *2* in Def. 2.6, we draw a blue edge if the edge was introduced due to existing edges in both G and H. If the edge was introduced due to non-existing edges in G and H, said edge will be colored red - note that these colors are purely pedagogical and not actual labels. The construction of a product graph is illustrated in Fig. 5. It is worth noting that modular product graphs are not limited to only two graphs, but can be

constructed from an arbitrary number of graphs as both conditions for edges in the modular product can easily be extended to additional graphs.

2.2 Subgraphs and subgraph isomorphisms

Definition 2.7 (subgraph) *A subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ and we then write $G' \subseteq G$.*

Definition 2.8 (node induced subgraph) *A node induced subgraph of a graph $G = (V, E)$ is a subgraph $G' = (V', E')$ such that $uv \in E'$ if and only if $u, v \in V'$ and $uv \in E$.*

That is, if two vertices u, v from G are present in the subgraph G' and u, v are adjacent in G, then these vertices must also be adjacent in G', see Fig. 6b. Informally, a node induced subgraph selects vertices from V and then requires that all edges connecting those vertices must also be included.

Definition 2.9 (edge induced subgraph) *An edge induced subgraph of a graph $G = (V, E)$ is a subgraph $G' = (V', E')$ such that for any edge $uv \in E'$, we have $u, v \in V'$.*

For an edge induced subgraph the edges are selected from E and then by extension the nodes incident on those edges are included in the subgraph, see Fig. 6c.

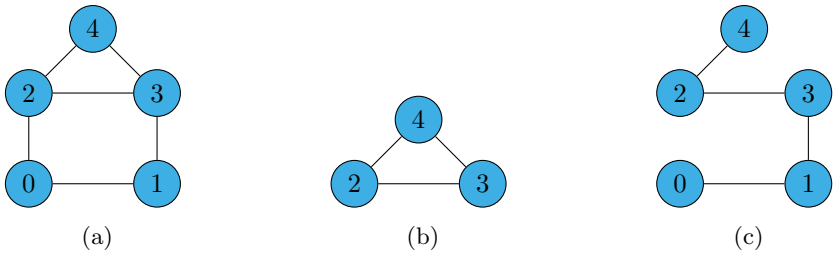


Figure 6: An example of node induced subgraph (b) and edge induced subgrah (c) of a graph (a).

Definition 2.10 (graph isomorphism) *An isomorphism from a graph $G = (V, E)$ to a graph $G' = (V', E')$ is a bijection $f : V \to V'$ such that $uv \in E$ if*

and only if $f(u)f(v) \in E'$. *We then say that* G *is isomorphic to* G' *and denote this formally by* $G \cong G'$.

For an example of a graph isomorphism, see Fig. 7.

(a) G (b) H

Figure 7: An example of two isomorphic graphs defined by the function f when $f(0) = a$, $f(1) = b$, $f(2) = d$ and $f(3) = c$. Notice carefully how $f(0)$ is adjacent to $f(1)$ and $f(2)$ as is required by the isomorphism.

Definition 2.11 (common subgraph) *A common subgraph of graphs* G_1 *and* G_2 *is a graph* G' *such that* G' *is isomorphic to a subgraph of* G_1 *and a subgraph of* G_2.

Informally, one might say that a common subgraph of two graphs G_1 and G_2 is a graph G' that is "included" in both G_1 and G_2. That is, you can "find a structure" identical to G' in both G_1 and G_2.

In relation to this project a **maximal common subgraph** is a common subgraph whose vertex set is not a proper subset of the vertex set of another common subgraph. One might also say that a maximal common subgraph is a subgraph that cannot be extended any further. We then denote a **maximum common subgraph (MCS)** as being the largest maximal common subgraph of the two graphs. Based on Defs. 2.8 and 2.9 we talk about **Maximum Common Induced Subgraph (MCIS)** and **Maximum Common Edge Subgraph (MCES)** for node and edge induced subgraphs, respectively. See Fig. 8 for examples of common subgraphs. In the application of chemical anchor extensions we wish to find the MCES.

The decision problem of finding a common subgraph has been proven to be NP-complete by a reduction from the clique finding problem, so no polynomial algorithm for finding the maximum common subgraph exists unless P = NP.

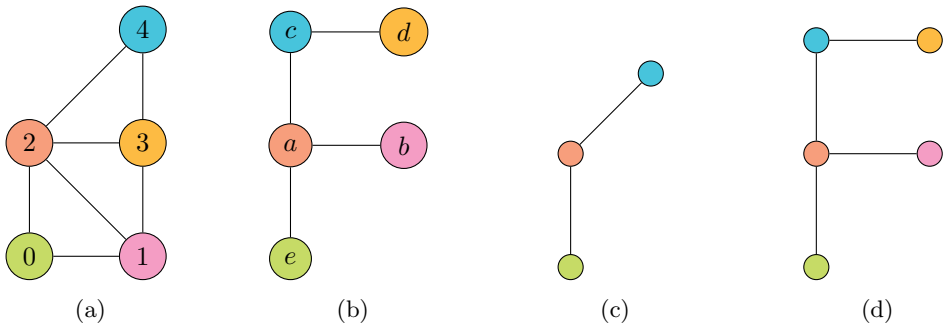


Figure 8: Two graphs (a) and (b) along with MCIS (c) and MCES (d). The colored nodes highlight the considered isomorphism of the chosen subgraphs in (a) and (b). Note how the MCES is larger than the MCIS as the MCIS cannot be extended due to the 3-cliques in (a) that do not appear in (b).

Surprisingly, clique finding algorithms can assist in finding common induced subgraphs as is evident by the following important theorem.

Theorem 2.1 *Every clique in the modular product $G_1 \times G_2$ corresponds to a common node induced subgraph of G_1 and G_2.*

Before moving onto the proof, we want to outline a few necessary observations. Recall that the common node induced subgraph between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a graph $G' = (V', E')$ such that G' is isomorphic to a subgraph of G_1 and a subgraph of G_2. That is for two subsets $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$ there exist isomorphisms $f : V_1' \to V'$ and $h : V_2' \to V'$. Since f and h are bijections, the inverses of f and h exist. The function $h^{-1} \circ f$ thus maps the subgraph of G_1 to the subgraph of G_2. This isomorpism can easily be represented by pairs (u, v) meaning that $u \in V_1$ is mapped to $v \in V_2$. Thus, the MCIS must be the subgraph isomorphism that maximizes this number of pairs. We then observe that the vertex set of the product graph $G_1 \times G_2$ consists only of such pairs. We now must show that (1) a clique in $G_1 \times G_2$ can be used to construct a valid common node induced subgraph of G_1 and G_2 and (2) that a node induced subgraph of G_1 and G_2 also corresponds to a clique in $G_1 \times G_2$. Note that it has previously been proven that cliques and modular products can be used to find a common subgraph (e.g. see [BB76], [Lev73]). Hence, we do not prove new knowledge. We instead present our take

on a proof that, hopefully, provides an intuitive explanation of how a clique in the modular product can be transformed into a subgraph isomorphism and vice versa.

Proof

(1) Let $C = \{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$ be an arbitrary clique of size k in $G_1 \times G_2$. Since there is no edge in a column or in a row of $G_1 \times G_2$ (when referring to rows/columns, see Fig. 5), we have $u_i \neq u_j$ and $v_i \neq v_j$, for every i, j; $1 \leq i < j \leq k$. i.e. all u_i's are different and the same goes for all v_j's. Also, since C is a clique, every pair of vertices, $(u_i, v_i), (u_j, v_j) \in C$ are adjacent in $G_1 \times G_2$ which, by Def. 2.6, means that u_i and v_i agree on the adjacency with u_j and v_j, respectively. This intuitively means that C defines a common subgraph. To make this intuition clear, we will define the isomorphism precisely. To do this, let $L = \{u_1, u_2, \ldots, u_k\}$ and $S = \{v_1, v_2, \ldots, v_k\}$. We then define the bijection $I : L \to S$ such that $I(u_i) = v_i$. This implies that I is a valid isomorphism between two subgraphs of G_1 and G_2, not only because it is bijective, but also because it respects the adjacency between nodes in V_1' and V_2'. This concludes the first part of the proof.

(2) Let G' be a common node induced subgraph of G_1 and G_2. That is for $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$ there exists an isomorphism $I' : V_1' \to V_2'$. We can easily represent each $I'(u) = v$ as a pair (u, v) which corresponds to nodes in the modular product $G_1 \times G_2$. This means that I' implicitly defines a subset of $V(G_1 \times G_2)$, let us call it T, where $T = \{(u, v) \in V(G_1 \times G_2) \mid I'(u) = v\}$. It now suffices to show that every pair $(u, v), (u', v') \in T$ are adjacent in $G_1 \times G_2$. Since $(u, v), (u', v') \in T$, we have $I'(u) = v$ and $I'(u') = v'$. As I' is an isomorphism, we have either $uu' \in E(G_1) \wedge vv' \in E(G_2)$ or $uu' \notin E(G_1) \wedge vv' \notin E(G_2)$. By Def. 2.6, both cases result in (u, v) being adjacent to (u', v') in $G_1 \times G_2$. This proves that T forms a clique in $G_1 \times G_2$.

□

Corollary 2.1.1 *Every clique in the modular product $L(G_1) \times L(G_2)$ corresponds to a common edge induced subgraph of the original graphs G_1 and G_2.*

Proof

This naturally follows from the fact that nodes in the line graphs correspond to edges in the original graphs G_1 and G_2. The respect to adjacency of nodes in the common subgraph of $L(G_1)$ and $L(G_2)$ implies a respect to incidence between edges in the corresponding common subgraph of G_1 and G_2 which is thus edge induced.

□

From this, we see that the MCIS can be found by finding the maximum clique in the product graph. If one instead wishes to find the MCES of two graphs G_1 and G_2, one should look for the maximum clique in $L(G_1) \times L(G_2)$. The clique finding problem itself is NP-complete, so why bother with this? The number of nodes and edges in a modular product graph clearly grows drastically as the number of graphs increases. Fortunately, the area of clique finding is very well studied (see [LP49], [MU04], [Pel09]). Because of this there exists well established algorithms and tools for solving the clique finding problem. Hence, our hope is that a proper clique finding algorithm will not be the bottleneck when used on a modular product graph, especially if one can limit the graph in such a way that the number of nodes to consider is not as vast as feared. These observations will prove useful later on.

3 McGregor's backtracking algorithm

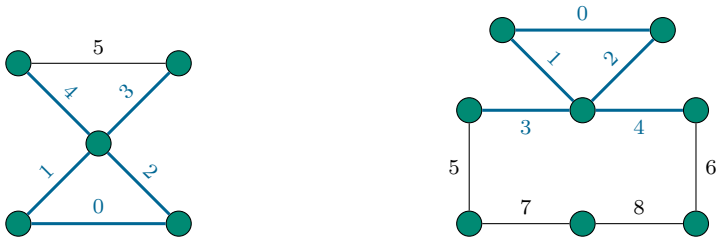
An algorithm for finding the MCES of two graphs, $G = (V_G, E_G)$ and $H = (V_H, E_H)$, was presented by James J. McGregor in 1982 [McG82]. McGregor's algorithm uses a backtracking search tree approach, building up a solution by trying all possible node correspondences whilst discarding search tree branches that will not result in "better" common subgraphs. This search tree pruning method is a common methodology used for exact algorithms on problems not known to be solvable in polynomial time [MP11]. In this section, we will describe the algorithmic approach of McGregor and subsequently discuss our implementation and our need for customization of the algorithm. Results gathered from tests of this implementation of McGregor's algorithm will be described in Sect. 5.1 and they will be discussed in Sect 6.1.

3.1 Algorithm

The basic version of McGregor's algorithm works under the assumption that $|V_G| \leq |V_H|$ and will only report a solution once all nodes in V_G have been mapped to unique nodes in V_H. When mapping a node $u \in V_G$ to a node $v \in V_H$, the algorithm keeps track of possibly mapped edges between G and H. That is, if u is tentatively mapped to v, only edges incident to u can be mapped to incident edges of v. After each tentative mapping the current state of the total mapping is saved in a workspace associated with the last mapped node. This allows for backtracking in order to try a different mapping at a later time in the execution of the algorithm. For each complete node mapping, the algorithm reports said mapping as well as a set of possible edge correspondences as a solution, taking note of the number of edges in E_G that have a possible mapping in E_H (later known as *arcsleft*). The first found solution, S, is naturally marked as the currently "best" solution. Following this, the algorithm considers a different solution S' if and only if the number of edges with a possible mapping in S' is strictly larger than the highest number of mapped edges so far. This way, at most one mapping of each possible size is reported by the algorithm. Now that an informal overview of the behavior

has been presented, we are ready to dig into the core of the algorithm.

The algorithm takes in two graphs as input. For simplicity's sake, we now denote the two input graphs as $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. The edge correspondence resulting from the performed node correspondence is constructed by a $|E_1| \times |E_2|$ matrix known as $MARCS$. Similar to an adjacency matrix of a single graph, $MARCS$ contains a 1 on position (r, s) for $r \in E_1$ and $s \in E_2$ if r can be mapped to s and a 0 otherwise. In the beginning (when no nodes have been mapped yet), $MARCS$ consists only of 1s as all edges in E_1 can be mapped to all edges in E_2. When nodes are tentatively mapped, say $u \in V_1$ is mapped to $v \in V_2$, $MARCS$ must be adjusted accordingly such that it reflects that edges incident to u can be mapped only to edges incident of v. This is done by changing all (r, s) entries in $MARCS$ to 0 when r is incident to u but s is not incident to v. From this we see that if row i in $MARCS$ is non-zero it means that edge $i \in E_1$ still has possible edge correspondences with edges of E_2. We let $arcsleft$ denote the number of rows in $MARCS$ that are not all



$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

MARCS

Figure 9: An example of the value of $MARCS$ with $arcsleft = 5$ after a run of McGregor's algorithm on graphs (a) and (b). The colored edges of (a) and (b) highlight the mapped edges in the resulting subgraph described by $MARCS$.

zero and decrement this value by 1 every time a row in $MARCS$ has its last 1 removed due to an update after a tentative node correspondence. If $arcsleft$

reaches 0, the mapping is not suitable as no edges can be mapped with the given tentative node mapping and the algorithm backtracks. Similar to *arcsleft*, the algorithm keeps track of a variable $bestarcsleft$ corresponding to the highest $arcsleft$ value discovered so far, starting off at 0. If a mapping of size $|V_1|$ is reached, the current $arcsleft$ is set as the new value of $bestarcsleft$ and the mapping is saved as a potential solution. After each refinement of $MARCS$ based on a tentative mapping the $arcsleft$ value is compared to $bestarcsleft$. If $arcsleft$ is smaller than or equal to $bestarcsleft$ then the current branch cannot be better than the current best mapping and backtracking kicks in to kill this branch. The algorithm then continues execution from a previously saved state. In the end, all found mappings along with their $MARCS$ are returned, where the largest mapping is the most relevant result. The behavior of this algorithm has been expressed in pseudocode which is illustrated in Fig. 10. An example of a found result of the algorithm can be seen in Fig. 9.

Line 7 in the pseudocode ensures that node i in G_1 is tentatively mapped to all untried nodes in G_2. "Untried nodes" refers to the nodes that i has not been mapped to previously (and thus those branches have yet to be explored). Additionally, the check for compatibility[2] ensures that any to-be-tried nodes in G_2 have not already been mapped to a node in the current branch. The purpose is to ensure that all branches in the search tree can be considered for each node in G_1. The backtracking, however, may kill certain branches without trying some mappings, but it is already clear at that point that those branches will not provide an optimal solution. The else statement starting at line 21 is executed if there are no more untried nodes in G_2 that node i can be mapped to. This allows for the backtracking where the workspace associated with node $i - 1$ is thus restored.

3.2 Implementation

The source code can be found in `/src/mcgregor.py`. The basic version of McGregor's algorithm has been implemented (albeit naïve) with additional

[2] Among other cases, which we discuss once we reach molecule graphs.

Algorithm 1: McGregor's algorithm for finding maximal common edge induced subgraphs of two graphs.

Input : Two undirected graphs G_1 and G_2.
Output: Maximal Common Subgraphs of G_1 and G_2.

Initialize $MARCS$ such that all entries are 1s;
$arcsleft \leftarrow |V_1|$;
$bestarcesleft \leftarrow 0$;
$i \leftarrow 0$;
mark all nodes of G_2 as untried for node 0;
while $i \geq 0$ **do**
    if there is any untried nodes in G_2 to which node i of G_1 may correspond **then**
        $x_i \leftarrow$ one of these untried nodes;
        mark x_i tried for node i;
        refine $MARCS$ on the basis of this tentative correspondence for node i;
        if $arcsleft > bestarcsleft$ **then**
            if $i = |V_1|$ **then**
                save mapping of nodes $x_0, x_1, \ldots, x_{|V_1|-1}$, $MARCS$;
                $bestarcsleft \leftarrow arcsleft$;
            else
                store a copy of $MARCS$, $arcsleft$ in the workspace associated to node i;
                $i \leftarrow i + 1$;
                mark all nodes of G_2 as untried for node i;
            end
        end
    else
        $i \leftarrow i - 1$;
        restore $MARCS$, $arcsleft$ from the workspace associated with node i;
    end
end
return saved mappings and their associated $MARCS$;

Figure 10: Pseudocode for McGregor. Note that 0-indexing has been used.

optional features. The library `NetworkX`'s([AS05]) `Graph` class is used for the representation of the graphs. The algorithm has been implemented as a simple tree traversal of the mappings between the input graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, each node in the search tree representing a sequence of tentative mappings of nodes from V_1 to V_2. The root node consists of the null-mapping where no vertices in G_1 have been mapped yet. The i'th layer in the search three thus consists of nodes where vertices $0, 1, \ldots, i-1$ of G_1 have received

mappings to unique vertices in G_2, the remaining nodes in G_1 have yet to be mapped. The $MARCS$ matrix is initialized, and later refined, as described in the previous section. For optimization purposes, we keep track of the number of 1s in each row in a list of size $|E_1|$, each cell to be decremented every time $MARCS$ is refined. To keep track of previous attempts at nodes in V_2, a boolean array is stored of size $|V_2|$ for each individual search tree node such that if node $i \in V_1$ is the next node to be mapped, it can instantly check if node $j \in V_2$ has already been tried. An identical array has been created to check for already "occupied" nodes in V_2. For workspaces, we created a class called $Workspace$ (in `/src/Workspace.py`). A workspace is associated with every search tree node and is created whenever a new vertex mapping has been established. The workspace contains attributes for $MARCS$, the number of arcs left, the number of 1s left in each row of $MARCS$, and the edges "killed" during the last refinement of $MARCS$. The value of the latter ensures that when backtracking and a different node is selected in V_2, $MARCS$ can be restored to its previous state[3]. This completes the description of the more ambigous parts of the pseudocode. We now briefly describe the added functionalities to the implementation with respect to the applications for rule inference in chemical reactions.

3.2.1 Anchor

The function allows the user to pass anchored edges as an optional parameter. The anchored edges are stored in a list of lists, where each inner list contains two tuples. The first tuple is the edge in G_1 and the second tuple is the edge that it is mapped to in G_2. $MARCS$ is refined based on these anchored edges as a preprocessing step, as this just needs to be done initially. This refinement is done for each anchor individually, so if edge i from G_1 is mapped to edge j then row i and column j in $MARCS$ is set to 0 except for the position (i, j) which will stay as a 1 to indicate that this is a good edge mapping.

[3]Note that this set of "killed edges" is a result of the order of the implementation which, in this case, is solely based on the order provided in the pseudocode. In theory, the non-refined $MARCS$ could easily be kept for the parent node before a new tentative mapping occurs. The refined $MARCS$ could then be passed down to the child node.

Furthermore, it is necessary to ensure that edges that are incident to edge i can only be mapped to edges incident to edge j. This is handled by setting entries to 0 where edge uv is incident to edge i in G_1 but $u'v'$ in G_2 is not incident to edge j.

3.2.2 Molecule graphs

Users can indicate labelling on the graphs via an optional boolean flag called molecule, which defaults to false. When using molecule graphs there are labels for both the vertices and the edges, where the vertices indicate atom type and the edges have a bond type. The bond type can be used to refine $MARCS$ as a preprocessing step in a straightforward way where any entry (i, j) will be set to 0 if the bond type of the two edges i and j are not the same. The vertex labelling becomes a part of the compatibility condition for line 7 in the pseudocode in Fig. 10. Vertex i from G_1 can only be tentatively mapped to vertex j in G_2 if i and j are of the same atom type. In order to simplify the code the algorithm acts as if labelling is always present, where we add empty labels if the molecule flag is false. The current implementation refers to the vertex labelling as "atom_type" and the edge labelling as "bond_type". Note that it would be rather straightforward to modify the implementation in such a way that the user could pass other strings to use for the vertex and edge labelling, respectively. Additionally, one could provide options to have several attributes that should match on the vertices and/or edges. These have not been implemented but the code would not need to be modified significantly to allow for it.

4 The LMC algorithm

A different approach to computing the maximum common subgraph of two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ was suggested by G. Levi in 1973 [Lev73]. He proposed using a table, N, called the *Node Correspondence Table*, that is, a table of size $|V_G| \times |V_H|$ where entry $N(i, j)$ corresponds to the mapping of node $i \in V_G$ to node $j \in V_H$. In other words, $N(i, j) = 1$ if the two nodes can be mapped to each other and 0 otherwise. Levi thus illustrates the k common subgraph problem as finding a square subtable T of N with k rows (called a k-cover), i.e. a set S of k non-zero cells, such that all rows and columns of T include only one element from S. Such a k-cover corresponds to a node mapping of k nodes of G to k nodes of H[4]. The constraint on the k-cover's columns and rows ensures that no two nodes from G are mapped to the same nodes in H and vice versa. All that remains for a k-cover to be a valid isomorphism is that it should respect adjacency between nodes of G and H. The connection between finding a k-cover in N, finding a k-clique in $G \times H$ and finding a common subgraph of k nodes should stand out at this point. As previously mentioned, we aim our focus on finding a *connected* extension of an already known "smaller" common edge induced subgraph between n nodes called the *anchor* denoted by A. When an anchored common subgraph of the input graphs is given, a potential algorithm for computing the extension is immediately given a theoretical (even if small) advantage because, among other aspects, the algorithm should not consider mappings of edges inside the anchor. In this section, we will describe our suggestion for an algorithmic approach of combining the anchor with the modular product to find a maximum common anchor extension of a set of input graphs. The description of the algorithmic approach will be followed by the implementation details. This algorithm will be referred to as LMC[5]. The results for the tests of the implementation will be presented in Sect. 5.2 and is followed by a discussion in Sect. 6.2.

[4] A common subgraph consisting of k nodes is called a common subgraph of order k in [Lev73].
[5] Linegraph-Modularproduct-Clique

4.1 Algorithm

The naïve method for finding a common node induced subgraph of size k for a sequence of input graphs $G_1, G_2, \ldots, G_n$ is to compute the modular product $G_\times = G_1 \times G_2 \times \cdots \times G_n$ and subsequently find a k-clique in $G_\times$ which we proved in Thm. 2.1. This approach does not guarantee that the resulting subgraph is connected as one could consider a clique in the modular product consisting of nodes connected only by red edges. For the purpose of finding the extension of an anchor, this approach clearly does not work as desired for numerous reasons:

i. The application of chemical context finding requires a connected, edge induced extension of the anchor.

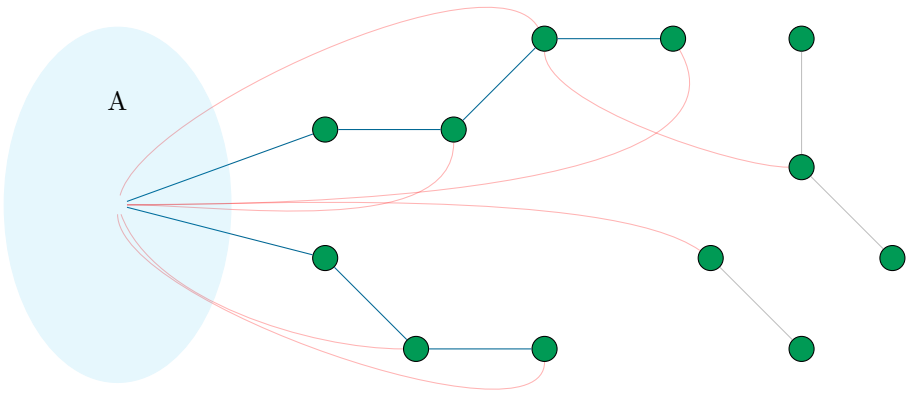
ii. Looking at arbitrary k-cliques in $G_\times$ may or may not include the anchor either entirely or in parts.

iii. The number of nodes and edges in $G_\times$ creates an unwinnable fight for any clique finding algorithm as n grows.

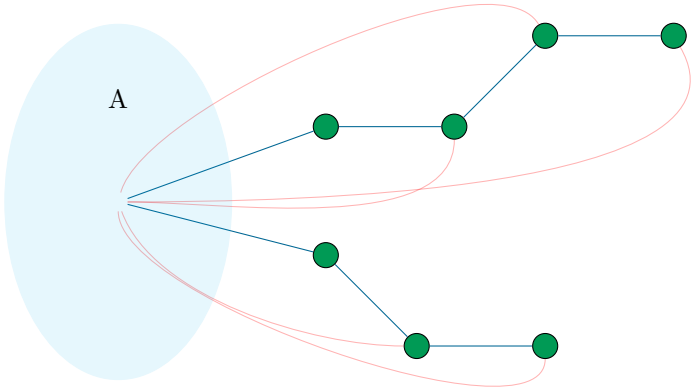
The problem of finding edge induced subgraphs as opposed to node induced subgraphs is "easily" solved following Corollary 2.1.1. Because of this, we now consider a graph $LG_\times$ which is the product graph of the respective line graphs of the input graphs. Nodes in $LG_\times$ hence now correspond to edge mappings in the input graphs. Note that the anchored nodes in $LG_\times$ are of the type $(a_1, a_2, \ldots, a_n)$ where a_1 is an anchored edge in G_1, a_2 is an anchored edge in G_2 and so on.

A suitable algorithm must find only the cliques in $LG_\times$ that result in a connected extension of A and do this without having to consider nodes who *a priori* are irrelevant. This is where A. Davoodi's proposal (see [Dav23]) kicks in. Given a set of graphs and an anchor as already described, the number of nodes in $LG_\times$ can be reduced significantly. Firstly, we discard all "conflicting nodes". For instance, we can easily exclude product graph nodes that contain both anchored and non-anchored vertices as these clearly cannot agree on whether they are anchored or not. As an example, suppose a node in the

product graph is (e_1, e_2). If e_1 is anchored in G_1 and e_2 is not anchored in G_2, then that product graph node is in a conflict with itself and can thus be discarded. When also considering the direct application of context extension in molecules, the number of conflicting nodes in $LG_\times$ will be more vast. We can consider molecules as decorated graphs with l_V being a function mapping nodes in the input graphs to atom types (e.g. C for carbon, O for oxygen, N for nitrogen) and l_E being a function that maps edges in the input graphs to their bond type (e.g. single, double, aromatic). When converting the input graphs to line graphs we recall that edges in the original graphs are transformed into vertices. Let $e = uv$ be such an edge in one of the input graphs. In the corresponding line graph, we can now define $l_V(e) = (l_E(e), \{l_V(u), l_V(v)\})$ such that a vertex in a line graph is both identified by 1) the original bond type it had and 2) the atom types of its two endpoints. Clearly, e cannot be mapped to another edge e' if $l_V(e)$ is not exactly equal to $l_V(e')$ on all parameters. Because of this, we can exclude a "product node" $v_\times = (v_1, v_2, \ldots, v_n)$ in $LG_\times$ if there exists two vertices $v_i, v_j \in v_\times$ such that $l_V(v_i) \neq l_V(v_j)$, these two vertices clearly do not agree on what kind of edges they should map to, so we can leave them out. Lastly, we discard nodes if they do not match up with our "expectation to connectivity" to the anchor. That is, we also discard a node in $LG_\times$ if it is not in the neighbourhood of *all* anchored nodes (both red and blue edges count here). Lastly, we divide the set of nodes into so called *blue connected components*. A blue connected component is just a subset of nodes on a blue path to A. All nodes not on a blue path to A are therefore also excluded. During this "polishing process" of $LG_\times$, the product graph will go from a graph similar to the one in Fig. 11a to a smaller graph similar to the graph shown in Fig. 11b.



(a)



(b)

Figure 11: An illustration of the removal of connected components not connected to the anchor by blue edges. Conflicting nodes have been removed prior to step a). The anchor is depicted as a collection of anchored vertices, that is, an edge entering the anchor could be entering any anchor node. Furthermore, each of the nodes needs to be connected to all anchor nodes in A, these extra edges have not been drawn to avoid cluttering the figure.

Now that we have limited the number of nodes in $LG_\times$, it is time to dig into how cliques in the product graph are transformed into extensions of the anchor. The idea is as follows: Given $LG_\times$ now consisting only of the anchor

and the blue connected components, we want to find all maximal cliques in the graph. For each clique, we discard all nodes not connected to the anchor via a blue path in the graph induced by the anchor and this clique. The remaining nodes and the anchor then result in the given extension. Such an extension is illustrated in Fig. 12. On the figure, one should observe that if all the gray edges are red, only the green nodes are reported as an extension of the anchor which means that the same extension is reported twice for two separate cliques. Because of this, one must also filter away isomorphic anchor extensions when considering the final result. Pseudocode describing the entire procedure of creating the product graph and finding cliques is found in Figs. 13 and 14.

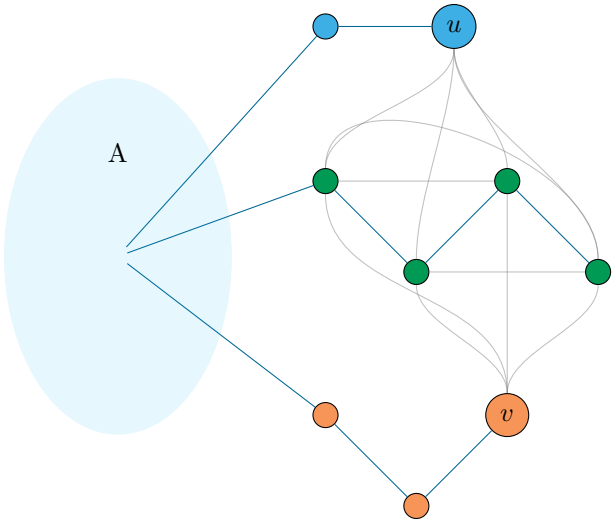


Figure 12: An example two cliques in $LG_\times$. One clique consists of all green nodes and the node u, another consists of all green nodes and the node v. Gray edges could be of any color. If all gray edges are red, only the green nodes are reported as an extension of the anchor as u will be discarded when unioned with the green vertices, and the same would happen to v. If at least one of the gray edges to, say, u is blue, u will have a blue path to the anchor when unioned with green vertices, and the union of u and the green vertices would therefore be reported as an extension.

Now, this procedure clearly works for an arbitrary number of graphs as one can compute $L(G_1) \times L(G_2) \times \cdots \times L(G_n)$ for any positive value of n. However, despite the filtering of nodes in $LG_\times$, our hypothesis is that the number of

nodes and edges still grows to an unworkable number if one computes a product graph for a large number of graphs, the number of cliques in the product graph will thus naturally also increase.

Algorithm 2: Our algorithm for finding all maximal common edge induced subgraphs of two graphs.

Input : A set of n undirected graphs, L. A set of anchored edges, A_E.
Output: All maximal common subgraph extensions of all members L w.r.t A_E.

$L_{LG} \leftarrow \{LG(G) \mid G \in L\}$;
$LG_\times \leftarrow L_1 \times L_2 \times \cdots \times L_n$ for $L_i \in L_{LG}$; `// possible size limitation`
$N \leftarrow \bigcap_{x \in A} N_{LG_\times}(x)$;
$B \leftarrow \{v \mid v \in V(LG_\times)$ and v is not reachable from A via blue edges$\}$; `// BFS`
$N \leftarrow N \setminus B$;
if $|N| = 0$ **then**
    return A_E as MCES;
end
$MCSs \leftarrow connected\_MCS(LG_\times, A_E, N)$;
return MCSs;

Figure 13: The pseudocode for the algorithm `LMC_MCS`

Algorithm 3: The algorithm for finding the anchor extensions based on the blue neighbourhood of the anchor.

Input : A modular product graph, PG. A set of anchored vertices in PG, A. The "blue" neighbourhood of A, N_b.
Output: All maximal extensions of A.

$X_A \leftarrow \{\}$;
$BG \leftarrow$ the graph induced by A and all nodes of N_b in PG;
$C \leftarrow \{c \mid c$ is a maximal clique in $BG\}$;
foreach $c \in C$ **do**
    $G_c \leftarrow$ the graph induced by A_E and c;
    $X \leftarrow \{v \mid v \in C$ is reachable from A via blue edges in $E_{G_c}\}$; `// BFS`
    if $|X| \neq 0$ **then**
        $X_A \leftarrow X_A \cup \{A \cup X\}$;
    end
end
return X_A;

Figure 14: The pseudocode for the algorithm `connected_MCS`

To minimize the product graph and the amount of work needed to han-

dle the product graph and cliques, we consider an *iterative* approach for finding the global maximum common anchor extension. Given input graphs $G_1, G_2, \ldots, G_n$ and the anchored edges in these graphs, the iterative approach first computes all maximal anchor extensions of G_1 and G_2. For every maximal extension m, we consider the maximal extension of G_m (m seen as the graph induced by m in G_1) and G_3 and so forth until we reach G_n. The resulting maximal extensions with G_n are thus the global common maximal extensions of the anchor. From these we can easily find the maximum (if needed be). The reason why we pursue *all maximal* extensions instead of only the *largest maximal* extensions is because we would run into the same issue that impacted McGregor's algorithm which will be discussed in Sect. 6.1.2. The iterative approach is described in pseudocode in Figs. 15 and 16. We will primarily focus on this iterative approach in our discussion but will also put it in perspective to using the method of computing the product graph for all input graphs (we will refer to this method as all_product approach). Now, however, we move onto to the implementation part of the presented algorithms.

Algorithm 4: The algorithm for the iterative approach of finding the maximal common anchor extensions for a set of graphs.

Input : A set of graphs $L = \{G_1, G_2, \ldots, G_n\}$. A set of anchored edges, A_E.
Output: All maximal common extensions of A_E of the graphs in L.

```
1 EXT ← {};
2 L' ← L \ {G_1, G_2};
3 iterative_aux(G_1, G_2, L', A_E, EXT);
4 return EXT;
```

Figure 15: The pseudocode for the algorithm `iterative_MCS`, namely the algorithm who's purpose is to utilize `LMC_MCS` in a step-by-step fashion.

4.2 Implementation

The source code can be found in `/src/cliques.py` with auxiliary libraries being `/src/productgraph.py` and `/src/linegraph.py`. The remaining libraries not mentioned here or in Sect. 3.2 are libraries meant to ease creation of input or visualize results and will therefore not be covered. The function that im-

Algorithm 5: The auxiliary algorithm for the iterative approach of finding the maximal common anchor extensions for a set of graphs.

Input : Two graphs, G_i, G_{i+1}. A set of remaining graphs $L = \{G_{i+2}, G_{i+3}, \ldots, G_n\}$. A set of anchored edges, A_E. The set of current found anchor extensions.
Output: Extends EXT whenever a new anchor extension is found.

```
1  R ← LMC_MCS({G_i, G_{i+1}}, A_E);
2  foreach r ∈ R do
3      if |r| > |A_E| then
4          if L = ∅ ∧ ¬∃m ∈ EXT : r ≅ m then // all graphs reached in this
           branch
5              EXT ← EXT ∪ r;
6          end
7          else
8              G' ← the graph induced by r in G_i;
9              L' ← L \ {G_{i+2}};
10             iterative_aux(G', G_{i+2}, L', A_E, EXT);
11         end
12     end
13 end
```

Figure 16: The pseudocode for the algorithm `iterative_aux` meant to control the recursive step in `iterative_MCS`.

plements the algorithm in Fig. 13 is called `mcs_list_leviBarrowBurstall`[6]. The function takes four parameters:

- `L`: A list of n graphs.
- `edge_anchor`: A list of anchored edges in all graphs. (see documentation for details on structure)
- `limit_pg`: A boolean variable indicating whether the product graph should be limited to the neighbourhood of anchors or not. Defaults to true.
- `molecule`: A boolean variable indicating whether the product graph has molecule labelling, i.e. atom type for the nodes and bond type for the edges. Defaults to false.

[6]The naming comes from G. Levi, H.G. Barrow and R.M. Burstall (see [BB76]) who all released articles on subgraph isomorphisms, product graphs and maximal cliques.

The algorithm has been implemented more or less in the order of the lines in the pseudocode, using `NetworkX`'s `Graph` class to control the structure of graphs. The graphs are initially transformed into line graphs using the algorithm that follows from Def. 2.4, keeping in mind that the edge anchors should be transformed into vertex anchors in the line graphs (i.e. if $e \in G(E)$ is an anchored edge in G, then $e \in V(L(G))$ is an anchored node in $L(G)$). The product graph is then computed using `itertools`'s function for cartesian product on the vertex sets of the graphs in L, constraining the number of nodes and edges depending on the values of `limit_pg` and `molecule` - edges are colored either red and blue to follow Def. 2.6. From there, the blue connected components are found using a simple breadth-first search[7] from the anchor, using only blue edges for traversal. The set R denotes all marked nodes found during this search. To find individual components, we extract a random node, $v \in R$, find the set of nodes, S, reachable by a blue path from v and denote $\{v\} \cup S$ as a blue connected component. For every blue connected component found, we subtract its vertex set from R and continue until R is empty. Once all blue connected components are found, the function `connected_MCS` which is implemented based on the pseudocode in Fig. 14 is called on the anchor and these blue connected components. `connected_MCS` utilizes the `NetworkX` function `find_cliques` to find all maximal cliques[8] and then uses another breadth-first search to exclude nodes that are not reachable from the anchor via blue edges. The anchor extensions are reported as already discussed.

The `mcs_list_leviBarrowBurstall` function takes an arbitrary number of graphs as input which makes it easy to use as a tool for the iterative approach which only takes on two graphs at a time - this is what we did. The function `iterative_approach` is implemented based on the pseudocode in Fig. 15. Its auxiliary function based in Fig. 16. `iterative_approach` initializes an empty list of mappings and calls `_iterative_approach_aux` with the first two input graphs and this list as input, which in turn calls itself recursively on the found

[7] Using Python's own `queue` library.

[8] We could have spent time implementing our own clique-finding algorithm, but we would have no guarantee that it would be optimal in any way and it could thus possibly interfere with our results later on.

unique[9] maximal extensions and the next input graph in line, building up an anchor extension throughout. Whenever a branch reaches an extension the size of the anchor (i.e. it is exactly the anchor), the branch terminates. This is due to the fact that all built extensions converge towards global common maximum extensions - such a global maximum cannot grow after the maximal extensions have been found for the first two input graphs G_1 and G_2 because if a maximum extension is common for *all* the input graphs, it must also be common in G_1 and G_2 and cannot "grow" afterwards! Whenever all input graphs have been processed, the currently built anchor extension is added to the initial list of mappings. If no anchor extensions were found at all, the anchor extension itself will be the maximal common anchor extension (i.e. the extension is of size 0).

[9]That is, all maximal extensions are checked up for isomorphism using the `NetworkX is_isomorphic` function, such that only one extension per isomorphism class is recursed on.

5 Results

In this section, we present the results of testing McGregor's algorithm and the LMC algorithm on various input graphs.

5.1 Testing McGregor's algorithm

The implementation of McGregor's algorithm has been tested on several smaller instances where $(|V_1|, |V_2|) \in \{(5, 10), (5, 20), (10, 10)\}$ with a timeout limit of 10 minutes. An overview of the results can be seen in Figs. 17, 18 and 19. The description of the input graphs are given on each figure. Note that on all figures, the presented results are shown in an arbitrary order - that is, there is no connection between traversing along the x-axis and the number of edges in the given input graphs. For more detailed results, we refer the reader to Appx. A where tables containing the complete results can be found. We also tested the implementation of McGregor on our randomly generated graphs in Appx. B (details on these can be found in Sect. 5.2.1). These unlabelled graphs have a cyclic anchor which means that there is no ambiguity within the mapping of edges, see Sect. 6.1.1 for details on this problem. This is due to the fact that ambiguity in McGregor's algorithm only occurs when, if you observe the anchor as a subgraph, the anchor is a tree - (the ambiguity is visible in leaf nodes, not in internal ones) and cyclic anchors are not trees. Results from these tests can be seen in Table 1.

Graphs	Without anchor time(s)	With anchor time(s)
0 1	10.88908	0.05719
0 1 2	11.51020	0.05599
0 1 2 3	16.39181	0.11060
0 1 2 3 4	18.30438	0.11020

Table 1: The runtime of McGregor's algorithm with unlabelled graphs, both with and without anchored edges.

We want to emphasize that illustrating a clear (or perhaps just reasonable) connection between graphs of varying sizes and the resulting runtime of the algorithm has been difficult. We have not included instances where $|V_1| > 10$

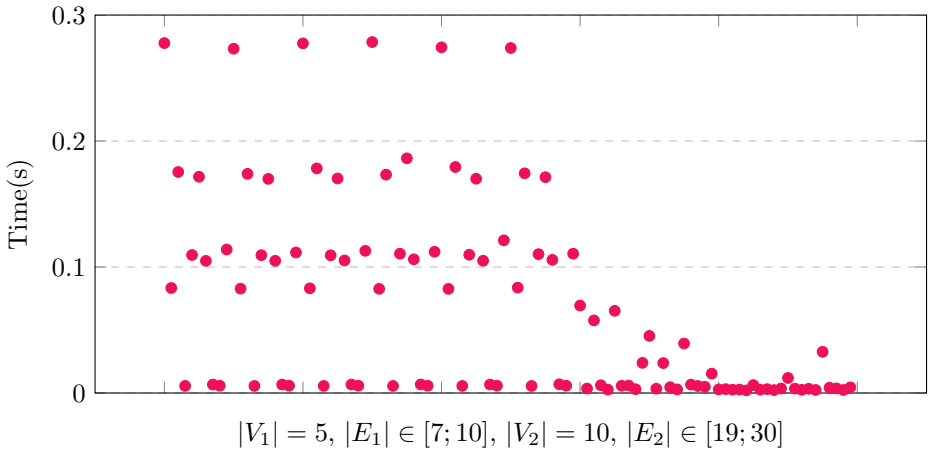


Figure 17

and $|V_2| > 20$ because, as it turns out, the algorithm simply does not terminate within 10 minutes for most graphs of these sizes. Based on this, and the fact that McGregor's algorithm does not find all maximal common edge induced subgraphs, we did not deem it relevant to test on much larger (or decorated for that matter) graphs. We will elaborate on the details of this in Sect. 6.1.

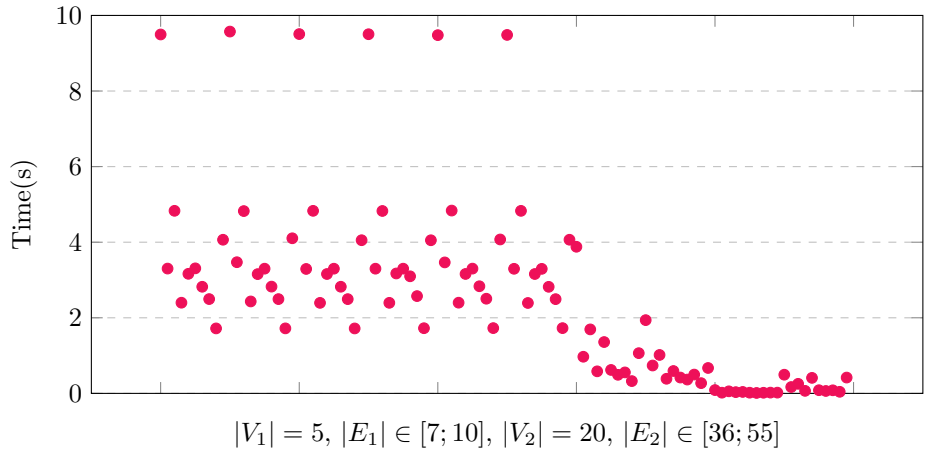


Figure 18

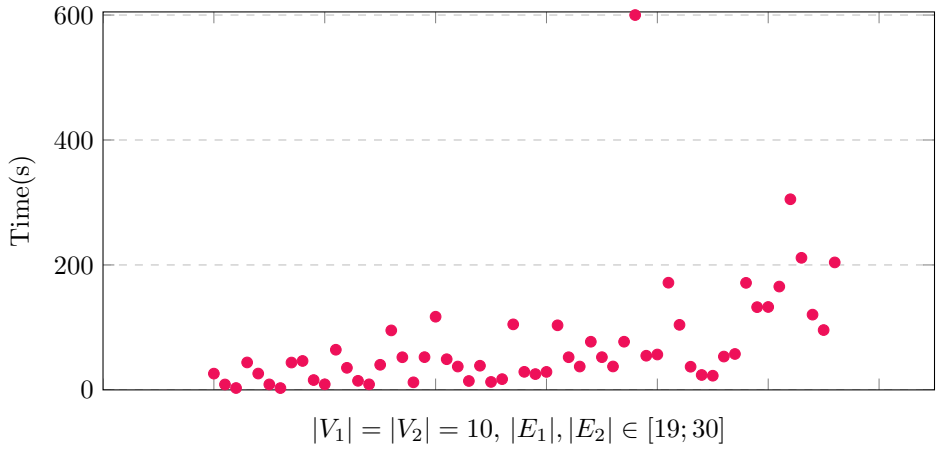


Figure 19

5.2 Testing the LMC algorithm

For the LMC algorithm, we have chosen two different (labelled, unlabelled graphs) test suites for the iterative approach and did a single test of the all_product approach with unlabelled input graphs. The reason for only doing a single test of the all_product approach should become clear from the test and subsequent discussion. Lastly, we have performed and illustrative test for the connection between size of the product graph and the number of cliques for one of the labelled graphs.

5.2.1 Unlabelled graphs

The unlabelled graphs are "randomly" generated with a limited size both in terms of nodes (8 to 10 nodes) and edges (15 to 18 edges). Visualization of these graphs can be found in Appx. B, though a single example can be found in Fig. 20. Anchors have been chosen in the graphs by visual inspection and are highlighted in pink. The tests performed on these graphs are primarily meant to highlight the impact of the chosen sequence of the graphs given to the iterative approach, e.g. $[G_1, G_2, G_3]$ or $[G_2, G_3, G_1]$. Fig. 21 shows the runtime for all the possible sequences including of 2, 3, 4, and 5 graphs, respectively. Lastly, the all_product approach with all 5 graphs did not terminate within an

hour and the test computer ran out of memory.

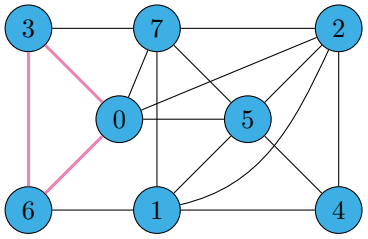


Figure 20: An example of one of the unlabelled input graphs (Graph 2) used in the test.

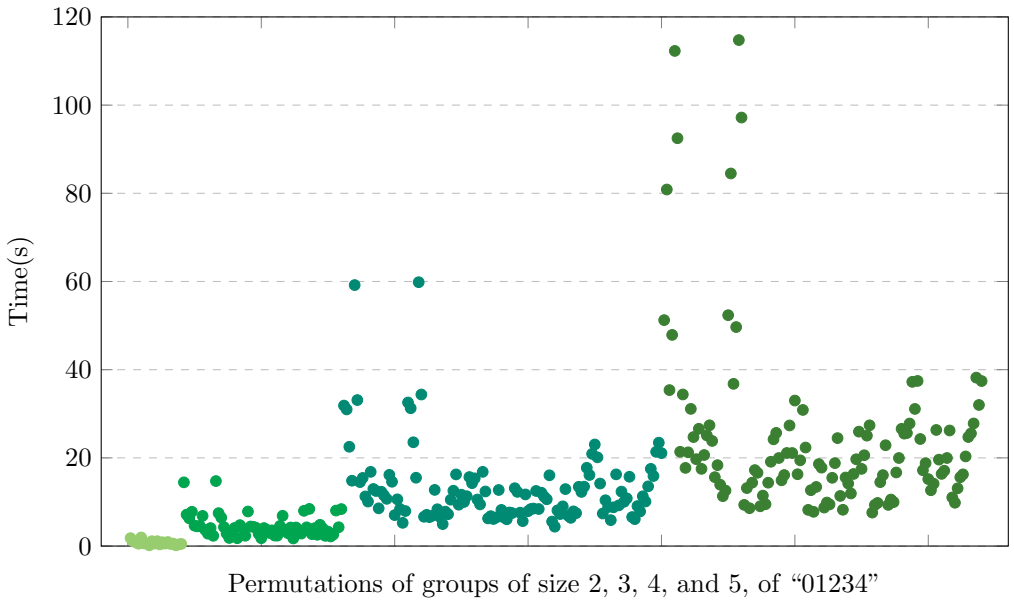


Figure 21

5.2.2 Labelled graphs

The labelled graphs have been provided by Daniel Merkle. These graphs have been extracted from DPOs and model molecules. As DPOs include both educt (G) and product (H) molecules, the graphs used in these tests correspond to the G and H molecules "stacked on top" of each other, such that anchor edges in both G and H are included in the final graphs - note again that the anchor edges are the edges that change type (e.g. from not existing to becoming

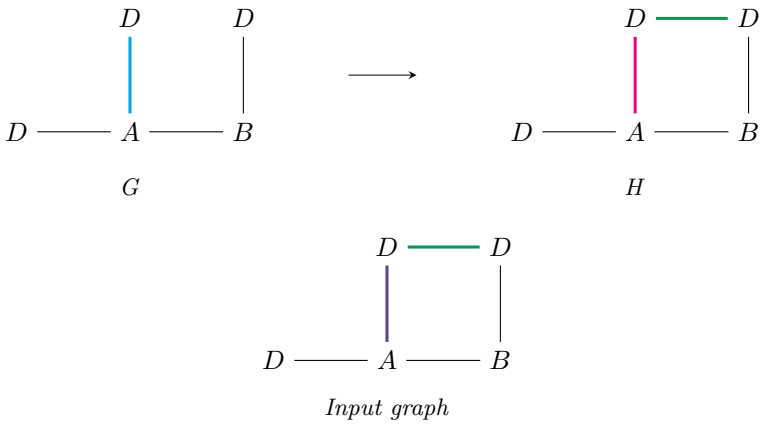


Figure 22: An illustration of how anchored edges are discovered between graphs G and H. The blue edge in G becomes a pink edge in H. The green edge does not exist in G but appears in H. Therefore, both edges (the violet edge illustrates the mix of blue and pink) are included in the input graph.

a single bond or from a single bond to a double bond) during the reaction. Because of this, only information regarding the change in the bond type for the anchor edges between G and H of the DPO must be maintained (see Fig. 22 for an example of the configuration of the input graphs).

Since these graphs model molecules they have both node labels (atom type) and edge labels (bond type). For the input given, the anchor edges for each graph are included. However, in its current form, it is not possible to directly infer the exact mapping between anchored edges in G_i and anchored edges in G_j. Because of this, all possible anchor edge-mappings have been calculated (see `/src/graph_format.py`)[10]. From these anchors, we chose the first one as it generally seemed like a good one (though not the best for all graphs as the MCS was not found for all graphs with this mapping). Furthermore, a "good" sequence of graphs given to the algorithm has been chosen. For an example of a "good" sequence, consider the sequence $0, 2, 1, 3$ for Fructose Bisphosphatase seen in Appx. D. Table 20 shows a runtime of ≈ 0.14 seconds while a different test (not presented here) showed that the sequence $0, 1, 2, 3$ has a runtime of

[10]It has been discussed that the anchor mapping would later be possible to infer given the reaction patterns, but this has not been developed yet.

$\approx$ 385 seconds which is a *remarkable* difference.

Because we did not expect to see anchor extensions of more than 5-10 edges, we decided to "shrink" the input graphs based on the distance from the anchor nodes in a breadth-first search. For each test instance, we started of by removing all edges further than one edge away from the anchor, two edges away from the anchor and so on up until nine edges in all of the input graphs. It is important to note that the result from each iteration is not used in the next, however, this is something we will discuss in Sect. 6.2.2. All test results can be found in Appx. D, but an excerpt of these can be found in Figs. 23, 24 and 25. On each figure, the left diagram shows the change in number of extensions found as we allow more and more edges to appear in each graph. The right diagram shows the change in run time. The tests were run for 30 minutes after which they would timeout if not finished. Extension diagrams does not include results for instances that timed out. For visualization of a found extension, see Appx. E in which AHACHB's solution is shown.

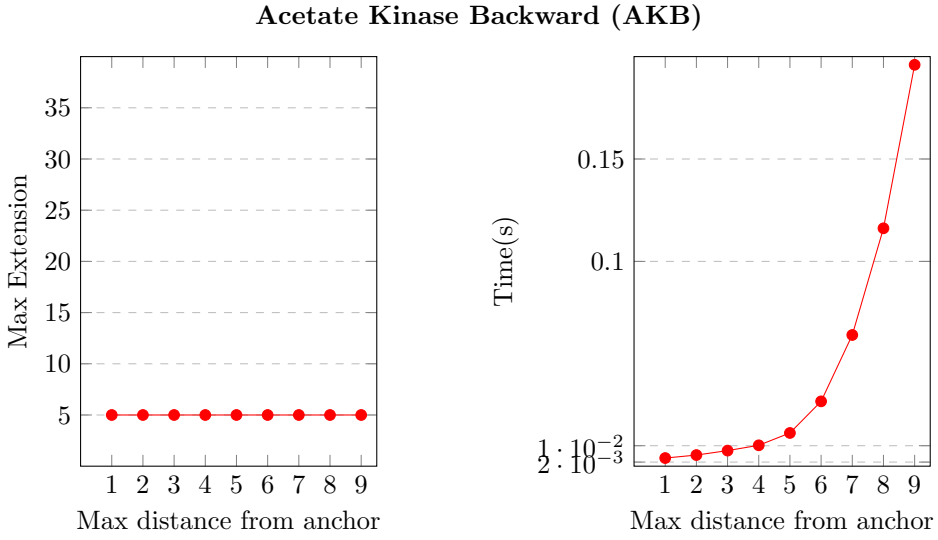


Figure 23

Alcohol Dehydrogenase Ethanol Backward (ADEB)

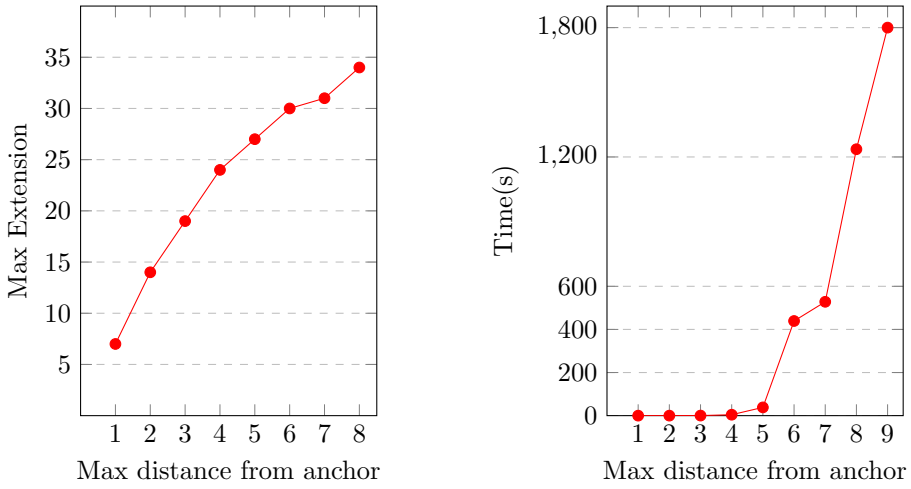


Figure 24

Aconitase Half-reaction A Citrate Hydro-lyase Backward (AHACHB)

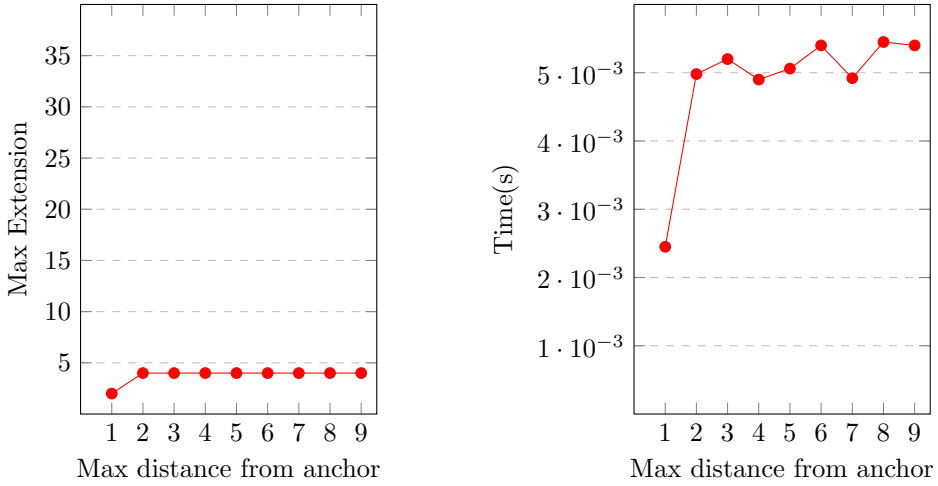


Figure 25

5.2.3 Glucose Dehydrogenase Forward clique experiment

As mentioned, we have performed a test on one of the labelled graphs to show the difference between distance from the anchor and the number of cliques found as well as the runtime of this. We decided to perform the test on one

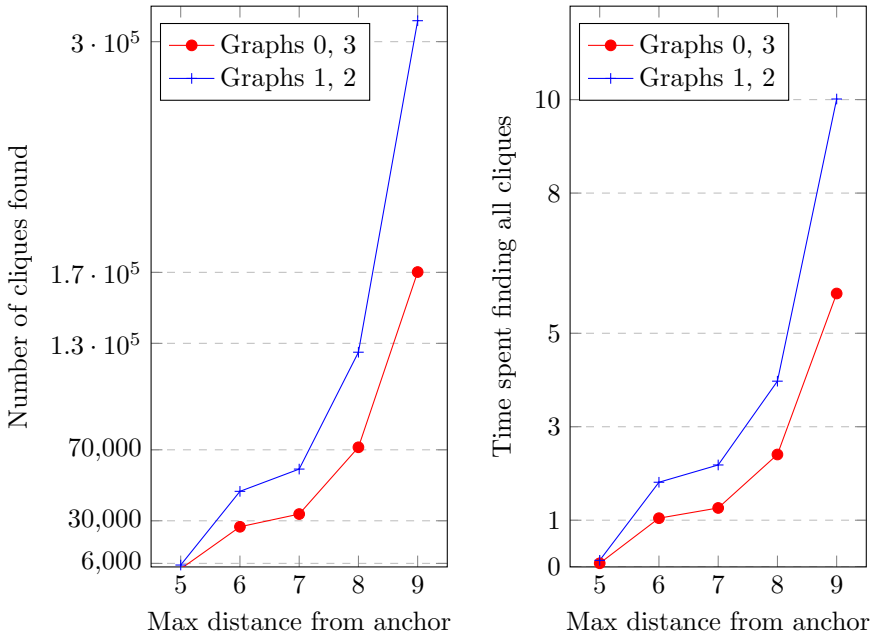


Figure 26

of the graphs with a challenging runtime, the chosen one being Glucose-6 Phosphate Dehydrogenase Forward. The results from these tests can be seen in Tables 2 and 3. An excerpt of the data set is illustrated in Fig. 26.

Tables 2 and 3 show how the product graph increases in size and the number of cliques found in the product graph as the max distance grows. The cliques are the ones found in the blue connected components. The time for both the creation of the product graph and how long it takes to find the cliques among the blue connected components is listed. Additionally, the table lists the average time it takes for the algorithm to perform BFS on 500 cliques. Entries with a dash indicate that the chosen distance did not finish execution within a reasonable time.

Distance	PG nodes	PG edges	PG creation (s)	Cliques	Find Cliques (s)	BFS per 500 Cliques (s)
5	252	20465	0.094	2896	0.078	2.5
6	289	33373	0.156	26624	1.043	3
7	291	33931	0.156	33792	1.262	3
8	300	36534	0.172	71424	2.406	4
9	306	38278	0.172	170176	5.851	3.4
infinity	862	330316	1.516	-	-	-

Table 2: Glucose-6 Phosphate Dehydrogenase Forward (Graph 0 and 3 from Table 34 as the first two graphs)

Distance	PG nodes	PG edges	PG creation (s)	Cliques	Find Cliques (s)	BFS per 500 Cliques (s)
5	210	17254	0.078	5068	0.141	3
6	265	27862	0.156	46592	1.812	3.7
7	266	28119	0.141	59136	2.182	3.2
8	271	29445	0.156	124992	3.975	3.5
9	276	30770	0.141	311808	10.013	4
infinity	804	285883	1.266	-	-	-

Table 3: Glucose-6 Phosphate Dehydrogenase Forward (Graph 1 and 2 from Table 34 as the first two graphs)

6 Discussion

We begin this section with a discussion of the results obtained from the tests of McGregor's algorithm, followed by a similar discussion of the LMC algorithm. Lastly, we will discuss the two algorithms in relation to each other.

6.1 Discussion of McGregor implementation

As can be seen in Figs. 17, 19, and 18 from Sect. 5.1 and the Tables 4 and 5 from Appx. A, it is clear that when both G_1 and G_2 become larger, the runtime increases exponentially. The tables show that the runtime increases (on non-anchored input) from less than a millisecond for some graphs with 5 nodes to in the worst case timing out after 600 seconds for graphs with 10 nodes. This implies that the runtime is highly dependent on the number of nodes in the two graphs given to McGregor's algorithm, which was expected. On the other hand, based on the referenced tables and the figures in Sect.

5.1, it can also be seen that the effect that the number of edges in each graph has on the runtime is not clear. As an example from Table 5, a run of the algorithm on two graphs with respectively 25 and 19 edges has a runtime of $\approx$ 105 seconds while an instance of two graphs with 24 and 28 edges has a runtime of $\approx$ 9 seconds. From this it appears that the structure of the two graphs may have a much higher influence on the runtime than simply the number of edges. Furthermore, the order in which the search tree is created for a specific instance can have a significant impact. If a "good" solution is found in the first branch it can help to skip most of the remaining branches, whereas if a "good" solution is not found until late in the execution a lot of "bad" branches will be explored. The current implementation does not do anything to try and force good solutions early on so it stands to reason that it will have an important impact on the runtime. J. McGregor proposes optimizations to find "good" branches quicker in [McG82].

As Tables 6, 7, 8, and 9 from Appx. A clearly show, the graph with the smallest vertex set must be *very* small in order for the runtime of McGregor's algorithm to be reasonable. For instances when at least one of the input graphs has 5 nodes, the runtime stays somewhat "reasonable". One might argue that a graph of 10 nodes is still small, but the instances with 10 and 20 nodes did not stop within the timeout limit of 10 minutes, once again clearly demonstrating the exponential increase in runtime as both graphs grow. In the results we also see that the size of the second input graph has an impact on the runtime. It does, however, depend on the graph instances, as the fastest of the instances with 5 and 20 nodes ($\approx$ 0.01 seconds) is still significantly faster than the slowest instance with 5 and 10 nodes ($\approx$ 0.28 seconds). This illustrates the issues with giving a good analysis of the runtime as a function of the size of the input graphs.

When considering the search tree to traverse, we know that increasing the size of G_2 (the larger graph) widens the search tree, as each node from G_1 have more possibilities in each branch of the tree. However, when increasing the size of G_1 the search tree becomes deeper instead. This results in a potential increase in the number of needed computations, which supports the markedly

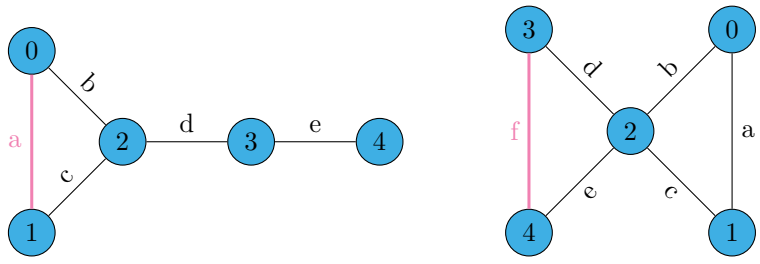
larger increase in runtime when the smaller of the two graphs become larger compared to increasing the size of the larger graph. As an example, suppose that G_1 has 5 nodes and G_2 has 10 nodes. The number of possible decisions is $10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 = 30240$. If G_2 gains an additional node, the number of possible decisions is $11 \cdot 10 \cdot 9 \cdot 8 \cdot 7 = 55440$. On the other hand, if G_1 was the graph that gained an additional node, the number of possible decisions would be $10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 = 151200$ which is five times the original number of decisions! This increase in the number of decisions is clearly dependent on the difference between the number of nodes in the two graphs. If G_1 only had one less node than G_2 the impact would be significantly lessened. Increasing the number of nodes in the smaller graph (which determines the depth) can thus be much more devastating compared to increasing the number of nodes in the larger graph (which determines the width).

6.1.1 Potential issues with anchored edges

In Sect. 3.2.1 one of the optional parameters described is the anchor, which allows the user to specify edge mappings that are already known before the execution of the code. We only present results on anchored graphs where the anchor is cyclic. If the anchor is not cyclic, we run into ambiguity issues of the result in *MARCS*. The reasons for this can be found in the ambiguity introduced by the interaction between the node mapping of McGregor's algorithm and the edge mapping of the anchor. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and an anchor mapping $[[e_1, e_2]]$ with $e_1 = uv \in E_1$ and $e_2 = u'v' \in E_2$ it is not specified whether u is mapped to u' or v' and because of this there can be situations where a later discovered edge mapping may potentially not be able to discern the exact edge mapping. The problem is illustrated in Fig. 27.

The resulting *MARCS* would potentially force the user to decide the exact mapping after the execution of the code, if anchored edges were given as input. For the reasons explained before, this is not an issue when the edge anchor is a cycle[11] as there would be only one way to map the nodes from the anchored

[11] The reader should convince themselves that this is the case when the direct mapping of edges is given.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 27: Illustration of the issue with anchors in the implementation of McGregor's algorithm. Mapping edge a from G_1 to edge f in G_2 does not specify whether node 0 in G_1 is mapped to node 3 or node 4 in G_2 and likewise for node 1 in G_1. Because of this in the MARCS matrix returned from the implementation rows 1 and 2 indicates that edges b and c from G_1 can both be mapped either to edge d or e in G_2.

edges in G_1 to the nodes in G_2. The anchors in the graphs from Appx. B that were used to create Table 1 consists of cycles. For the graphs in Fig. 27 it would not be a big problem for the user to decide the exact mapping, but with larger graphs it could be very difficult to handle. Based on this we have concluded that anchored edges cannot be guaranteed to work properly with the current implementation of McGregor's algorithm. However, there are several possible modifications that could work around or solve the problem. When discovering ambiguity as in Fig. 27 the algorithm could return two results, one with *MARCS* such that positions $(1, 3)$ and $(2, 4)$ have been set to zero and the other where positions $(1, 4)$ and $(2, 3)$ have been set to zero. Another more complicated solutions would be determining the node mapping from the edge anchor i.e., if $e_1 = uv$ and $e_2 = u'v'$ then u is mapped to u' and v is mapped to v'. This would put a lot of responsibility on the user and make the creation of the edge anchors more difficult. Another way to handle it would be taking a node mapping instead of the current edge mapping as the anchor. The reason that this has not been done instead is to be consistent with the input for the LMC algorithm.

6.1.2 Problems with labelled graphs

When dealing with unlabelled graphs McGregor's algorithm can easily be extended to work on more than two graphs, as can be seen from the result of Table 1. This is done by executing the code with G_1 and G_2 first and the resulting subgraph G', which can easily be constructed based on G_1 and $MARCS$, would be input along with G_3 for the second execution of the algorithm. It simply requires an additional execution of the algorithm for each graph after the first two. This works when the graphs are not decorated with node and/or edge labels as only the structure of edges and nodes must be present in all graphs and is not constrained by their attributes. However, it is not possible to use the current implementation for more than two decorated graphs. The problem is illustrated in Fig. 28 where the MCS of all three graphs cannot be found when finding the MCS of G_1 and G_2 first. This is because McGregor's algorithm finds the maximum common subgraph, instead of finding all maximal common subgraphs, which would be necessary to eliminate the described issue. We want to emphasize that it is not a simple task to gather maximal common subraphs instead of the maximum common subgraph with the current implementation of the algorithm. This is in large part due to the backtracking that is used in the algorithm to discard as many branches as possible, and by doing so it discards many of the maximal extensions. It would make the algorithm nonfunctional if the backtracking was removed as it would result in the algorithm going through all branches in the search tree, and it would result in a runtime that would make the program unusable.

The combinations of the inconsistencies with the edge anchors and the issues with more than two decorated graphs means that the implementation of McGregor's algorithm is not usable in the application of finding MCS in molecule graphs in order to discover context for graph transformation rules.

6.2 Discussion of the LMC implementation

As mentioned in Sect. 5.2.1 the all_product approach did not complete its execution within an hour, even with only five "relatively" small graphs (8 to

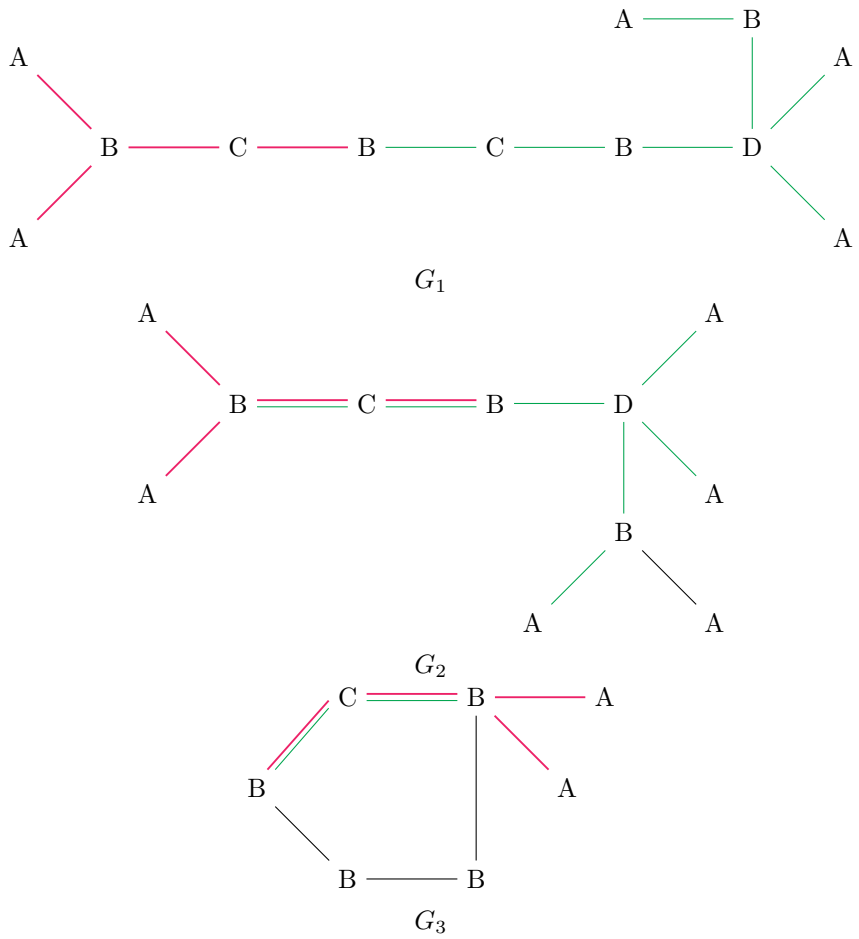


Figure 28: Parallel edges in G_2 and G_3 correspond to one edge, but two different mappings. By first running McGregor's algorithm on G_1 and G_2 the MCES would be the green edges. Unfortunately, the resulting MCES would only have three nodes and 2 edges in common with G_3 namely the green $B - C - B$ edges, whereas the MCES for all three graphs would be the red edges in all three graphs.

10 nodes, and 15 to 18 edges). The primary reason for this excessive runtime is the size of the product graph created from all five test graphs and the number of cliques found in the product graph. This problem becomes worse as the graphs grow, both in size and in the number of graphs. Because of this, we did not test the all product approach on the labelled graphs as they are on average a lot larger than the tested unlabelled graphs. This implies that, even if the number of nodes in the product graph could be reduced due to conflicting nodes etc., the product graph would get too large and would result in a never ending computation when finding cliques. Therefore, we will not consider the all product approach any further. Instead, we move on to the discussion of the results from testing the iterative approach.

6.2.1 Sequence of unlabelled graphs

First we bring up the peculiarities of Fig. 21 which illustrates the tests on the unlabelled graphs. As is evident by the diagram, there is a huge difference between the runtime within almost every class of unlabelled graphs, each class representing the number of graphs to consider. The runtime clearly depends on the sequence of the input graphs. Common for all classes is that all outliers include Graph 0 and 1 as the first two graphs whose maximal anchor extensions must be branched out on. On the other hand, the fastest runtimes were achieved when Graph 1 and 3 were the first ones. It is not obvious what the exact reason is for this outcome, but it turns out that the issue persists for labelled graphs as well. This sparked the motivation for the test described in Sect. 5.2.3 and this will be discussed in the following section.

6.2.2 Enzymatic reaction graphs

As can be seen on Table 2 and 3 in Sect. 5.2.3, it is clear that creating the product graph itself does not take a long time. Even creating the product graph when the input graphs are not limited does not take more than 2 seconds. However, finding cliques in this product graph does not complete within a reasonable time frame. This is supported by the growth in the number of

cliques and the time it takes to compute them as the product graph increases in size. For Graph 0 and 3 (Table 2) when increasing the distance from 8 to 9 the number of cliques grows from $\approx 7 \cdot 10^4$ to $\approx 17 \cdot 10^4$, while the time to find the cliques increases from ≈ 2.4 to ≈ 5.9 seconds. The difference in number of nodes and edges between these two graphs is 6 and 1744, respectively. On the other hand, the complete product graph had 556 more nodes and 292038 more edges than the graph with distance 9. From this it follows that the number of cliques would be astronomical when considering the entire product graph. Furthermore, even though the time it takes to do the BFS on each clique is somewhat stable, the serious increase in the number of cliques results in a significant increase in the overall runtime. In Table 2, we see that a distance of 9 created 170176 cliques. If it takes 3.4 seconds on average to process 500 cliques, the algorithm would not terminate for at least 19 minutes for the *first two graphs*. It is clear that this trend continues as the distance grows, so the number of cliques found clearly plays a huge part in the runtime.

Comparing the data of Table 2 and 3 highlights the importance of the sequence of the graphs. The number of cliques for Graph 1 and 2 is always higher than for Graph 0 and 3, even with the product graph being smaller for Graph 1 and 2. This illustrates the correlation between the number of cliques and a "good" sequence.

Another aspect is that, as the number of cliques grows, so does the number of potential branches to explore in the iterative approach. The more extensions found between the first two input graphs, the wider the recursion tree will become[12]. This will also have a natural impact on the runtime of the algorithm as more branches must be explored. There is no clear way of determining which graphs should be given as the first two graphs as that would require knowledge about the number of pairwise maximal extensions between the graphs beforehand, which is a circular argument.

The approach that we used to determine "good" sequences for the *labelled graphs* (which was doable as the number of input graphs for each instance

[12]Recall that the depth of the search tree is exactly the number of input graphs, so this cannot be optimized.

we were given was limited to five) was to run the algorithm on the first two input graphs and determine the number of cliques found in their product graph, selecting the sequence with the least number of cliques. Even though as described in Sect. 4.1 and illustrated in Fig. 12 several cliques may be discarded, a large amount of cliques is still very likely to also result in a lot of extensions, which we have now illustrated plays an important part in the runtime of the algorithm.

Using this approach of "pairing up the first two graphs" with n graphs, one would have to try out $\binom{n}{2} = O(n^2)$ different pairs of the first two input graphs. This is a polynomial amount of pairs, but still not optimal if the reaction network is large. This approach may be used for unlabelled graphs as well, but the problem of running all possible pairs of input graphs still persists.

In general, one can see that the number of cliques impacts the runtime in different ways. Firstly, if the number of found cliques is large, the time it takes to run BFS on the induced subgraphs with each clique becomes significant. Secondly, if a lot of cliques turn into extensions, the recursion tree becomes wider. Finally, even finding the cliques is an issue as the product graph grows in size. Everything thus comes down to the size of the product graph. This underlines the importance of limiting the product graph as much as possible. For most cases one will find that the smaller the product graph is, the better runtime can be achieved. Although, as the comparison between Table 2 and 3 highlighted, it is not always the case.

Moving on to the discussion of the actual extension tests presented in Sect. 5.2.2 and in Appx. D, it is clear that the graphs with larger maximal extensions have a higher runtime which, based on the above discussion, is expected. For the instances whose maximal extension of the anchor grows as the max distance from the anchor increases, the runtime increases exponentially (this can be seen from Tables 18, 19, 22, and 23 from Appx. D). It is worth pointing out that the maximal extension of the anchor has not been discovered for graphs with a still-increasing extension as one cannot be sure that it would not increase in size with the next max distance. However, for the graphs where the max extension stops growing, the max extension has definitely been discovered. In

fact, for many of these graphs it was already discovered very early on, see Fig. 23. In the case where the max extension is discovered earlier, it would not be necessary to consider the entire graph, or any further max distances. In the instances in this thesis it is not relevant, since for those instances where the MCS was discovered early, the subsequent tests ended quickly - it may be relevant for later use, however.

6.2.3 Challenges introduced by the implementation

As can be seen from the presented results and the following discussion there are several challenges in regard to solving this problem completely. For one, if the exact anchor mapping is not known, it will require a lot of computations to find the mapping that provides the best result. However, this may be an issue that can be handled elsewhere.

Another challenge is handling instances with large graphs, such as **GPDB**, where the runtime is significantly higher than for other instances. A proposed solutions for this problem would be to update the anchor after each iteration of increasing the max distance from the anchor. Meaning for the first iteration the graphs would be limited to a max distance of one from the anchor, then the anchor would be updated based on that result and the limit would be expanded for the next iteration. It is believed that this would be an improvement compared to the current implementation, as there would be less nodes and edges that would have to be considered as the graphs are limited less and less. There are two main reasons for this hypothesis. The first is that the edges extending the anchor would not need to be considered, as these are already known at this point to be a part of the MCS. The second is the belief that the neighbourhood, and therefore the size of the product graph, would be positively affected by the larger anchor.

6.3 Comparison of both algorithms

Because of its misalignment with the practical aspect of this thesis, we have not implemented a non-anchored version of the LMC algorithm. As a result, it is

very difficult to compare it with McGregor's algorithm in a meaningful sense. Even if we had implemented a non-anchored LMC approach, the algorithm would still find *all* maximal common subgraphs which McGregor wouldn't - this naturally implies that, on average, the LMC algorithm will spend more time computing its solutions. This, however, is also what makes the LMC algorithm much more suitable for global MCS for multiple decorated graphs. As discussed, McGregor's naïve approach does not work well in this regard, so comparing them like this would not be realistic.

With that in mind, we are still able to compare the anchored version of McGregor's algorithm on our randomly generated undecorated graphs. Comparing the results from Fig. 21 and the results from Table 1 from Sect. 5.1 over the same graphs, it appears that the implementation of McGregor's algorithm is significantly faster than the LMC implementation. The implementation of McGregor's algorithm for all five graphs has a runtime of ≈ 0.11 seconds, while the fastest sequence for the LMC algorithm has a runtime of ≈ 7.8 seconds. The LMC algorithm will, similarly to McGregor's algorithm, have an exponential increase in runtime as the input graphs grows in size, especially so when dealing with unlabelled graphs. The reasoning comes from the previous discussion of the impact that the size of the product graph (which cannot be limited as much for undecorated graphs, see 4.1) has on the runtime. While the implementation of McGregor's algorithm does perform better for these *unlabelled* graphs, it is important to keep in mind that the algorithm is not directly usable for finding MCS between more than two *labelled* graphs, as discussed in Sect. 6.1.2. Fortunately, the performance of the LMC algorithm improves significantly when dealing with labelled graphs as a result of the limitation of the product graph based on the node and edge labels. The fact that the LMC algorithm does not perform as well for unlabelled graphs is not as important because the application for anchor extensions (in molecules) will always be working on decorated graphs.

7 Conclusion

When it comes to finding the maximum common subgraph extension of more than two labelled input graphs with an attached anchor, it was immediately clear that a direct implementation of McGregor's solution is unfeasible. Not only because of the time complexity, but because McGregor's approach simply does not find all maximal extensions needed for more than two labelled graphs. The LMC algorithm, however, clearly works for labelled graphs and even very well for some instances. There are still instances where the runtime increases to a point where the current implementation is unable to solve the problem in reasonable time. We presented possible solutions to this problem in Sect. 6.2.3. In relation to runtime, the implementation may not be feasible if the anchor mapping is not given by the input graphs because computing the possible edge mapping combinations quickly gets out of hand when the number of edges in the anchor is large. On the other hand, should the edge mapping be given as input, the modular product approach seems feasible for most of the tested graphs and shows potential given an optimal implementation in a more efficient programming language like `C++`. A peculiar observation that we made was the fact that the sequence in which the input graphs were given had a remarkable impact on the runtime of the iterative approach, and we brought up possible explanations for this pattern. We also suggested a method for detecting a good sequence of the first two input graphs rather quickly.

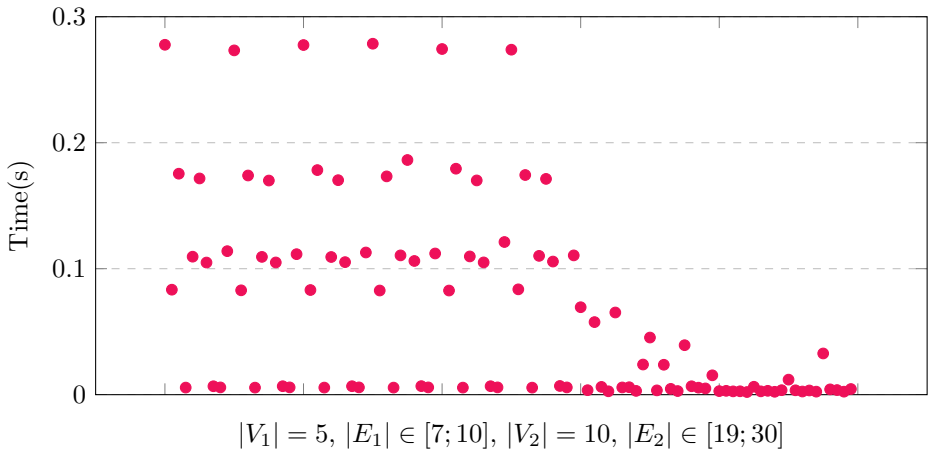
In conclusion, our prototype implementation shows that the problem of finding the maximum common subgraph extensions of anchored molecules is possible to solve for instances with three to four graphs when a "good" order of graphs is given. With further optimiziations, it may be possible to increase the number of graphs and still reach a reasonable runtime - reasonable being relative to the fact that we expect users to run this algorithm once on each set of graphs and save the results in a database for later use.

Acknowledgement

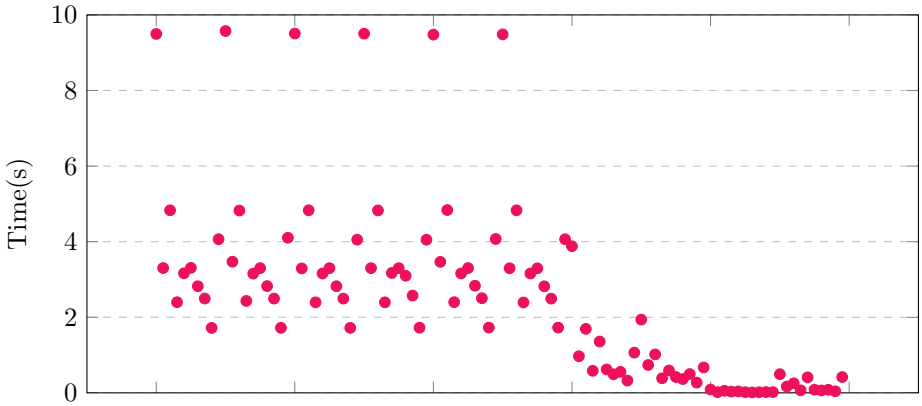
We would like to thank our supervisors Daniel Merkle and Akbar Davoodi for healthy discussions and constructive feedback. Their inputs were paramount to the further development of the algorithm and this paper and we could not have done it without them.

Appendix A McGregor test data

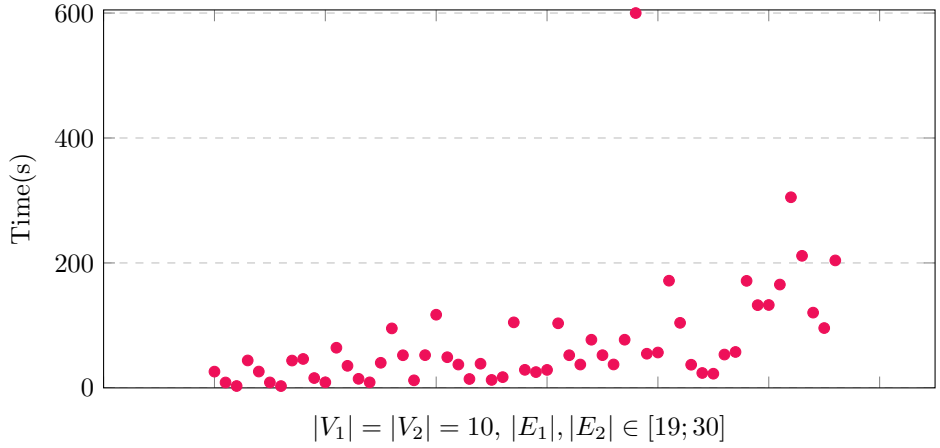
This data set represents tests of the McGregor algorithm. The set of graphs consists of three size classes - graphs with 5 nodes, 10 nodes and 20 nodes. The number of edges varies through the graphs, though all graphs are connected. First there are three figures plotting the run time of the tests. Afterwards 4 tables with the exact test results are displayed.



$|V_1| = 5$, $|E_1| \in [7; 10]$, $|V_2| = 10$, $|E_2| \in [19; 30]$



$|V_1| = 5$, $|E_1| \in [7; 10]$, $|V_2| = 20$, $|E_2| \in [36; 55]$


600
400
200
0
Time(s)
$|V_1| = |V_2| = 10,\ |E_1|, |E_2| \in [19; 30]$

G_1 nodes/edges	G_2 nodes/edges	Time (s)	G_1 nodes/edges	G_2 nodes/edges	Time (s)
5/10	5/10	0.00046	5/10	5/8	0.00439
5/10	5/10	0.00038	5/10	5/10	0.00046
5/10	5/10	0.00037	5/10	5/10	0.00045
5/10	5/10	0.0004	5/10	5/9	0.00383
5/10	5/10	0.00037	5/10	5/9	0.00359
5/10	5/9	0.00393	5/10	5/7	0.00432
5/10	5/9	0.00384	5/10	5/8	0.00447
5/10	5/7	0.00454	5/10	5/10	0.00044
5/10	5/8	0.00446	5/10	5/9	0.0038
5/10	5/10	0.00045	5/10	5/9	0.00405
5/10	5/10	0.00044	5/10	5/7	0.00437
5/10	5/10	0.00037	5/10	5/8	0.00478
5/10	5/10	0.00055	5/10	5/9	0.00389
5/10	5/9	0.00363	5/10	5/9	0.00392
5/10	5/9	0.00407	5/10	5/7	0.00456
5/10	5/7	0.00449	5/10	5/8	0.0042
5/10	5/8	0.00446	5/9	5/9	0.00292
5/10	5/10	0.00045	5/9	5/7	0.00394
5/10	5/10	0.00045	5/9	5/8	0.00262
5/10	5/10	0.00036	5/9	5/7	0.00303
5/10	5/9	0.00361	5/9	5/8	0.00273
5/10	5/9	0.00394	5/7	5/8	0.0005
5/10	5/7	0.0046			

Table 4: McGregor on graphs with 5 nodes

G_1 nodes/edges	G_2 nodes/edges	Time (s)	G_1 nodes/edges	G_2 nodes/edges	Time (s)
10/19	10/26	26.02879	10/25	10/28	25.30191
10/19	10/28	8.66097	10/25	10/29	28.79291
10/19	10/29	2.99406	10/25	10/25	103.38305
10/19	10/25	43.92559	10/26	10/28	52.28709
10/19	10/26	26.14767	10/26	10/29	37.37093
10/19	10/28	8.76622	10/26	10/25	77.08232
10/19	10/29	2.90403	10/26	10/28	52.37045
10/19	10/25	43.74742	10/26	10/29	37.48603
10/24	10/25	46.30886	10/26	10/25	77.10551
10/24	10/30	15.74858	10/28	10/29	timed out
10/24	10/28	8.9341	10/28	10/25	54.68681
10/24	10/19	64.26798	10/28	10/29	56.62926
10/24	10/26	35.31754	10/28	10/25	171.56876
10/24	10/28	14.51149	10/28	10/19	104.11856
10/24	10/29	8.87568	10/28	10/26	37.06321
10/24	10/25	40.17854	10/28	10/28	23.8604
10/25	10/24	95.22253	10/28	10/29	22.69586
10/25	10/25	52.25699	10/28	10/25	53.35599
10/25	10/30	12.11052	10/28	10/29	57.49223
10/25	10/28	52.41866	10/28	10/25	171.31848
10/25	10/19	117.1208	10/29	10/25	132.45440
10/25	10/26	49.09435	10/29	10/25	132.69938
10/25	10/28	37.37419	10/30	10/28	165.39116
10/25	10/29	14.23601	10/30	10/19	305.1515
10/25	10/25	38.67678	10/30	10/26	211.39996
10/25	10/30	12.74075	10/30	10/28	120.44785
10/25	10/28	17.22705	10/30	10/29	95.77612
10/25	10/19	104.88667	10/30	10/25	204.02821
10/25	10/26	28.89528			

Table 5: McGregor on graphs with 10 nodes. Note timeout happens after 600 seconds.

G_1 nodes/edges	G_2 nodes/edges	Time (s)	G_1 nodes/edges	G_2 nodes/edges	Time (s)
5/10	10/25	0.27773	5/10	10/25	0.27383
5/10	10/24	0.08333	5/10	10/24	0.08361
5/10	10/25	0.1754	5/10	10/25	0.17436
5/10	10/30	0.00563	5/10	10/30	0.00559
5/10	10/28	0.10953	5/10	10/28	0.11018
5/10	10/19	0.17163	5/10	10/19	0.17128
5/10	10/26	0.10488	5/10	10/26	0.10567
5/10	10/28	0.00669	5/10	10/28	0.00692
5/10	10/29	0.00577	5/10	10/29	0.00577
5/10	10/25	0.11392	5/10	10/25	0.11056
5/10	10/25	0.27328	5/9	10/25	0.06942
5/10	10/24	0.08284	5/9	10/24	0.00357
5/10	10/25	0.17398	5/9	10/25	0.05761
5/10	10/30	0.0056	5/9	10/30	0.00618
5/10	10/28	0.10937	5/9	10/28	0.00269
5/10	10/19	0.16997	5/9	10/19	0.06523
5/10	10/26	0.10495	5/9	10/26	0.00574
5/10	10/28	0.00672	5/9	10/28	0.00592
5/10	10/29	0.00579	5/9	10/29	0.00301
5/10	10/25	0.11153	5/9	10/25	0.02393
5/10	10/25	0.27753	5/9	10/25	0.04535
5/10	10/24	0.08307	5/9	10/24	0.00342
5/10	10/25	0.17827	5/9	10/25	0.02375
5/10	10/30	0.00563	5/9	10/30	0.00462
5/10	10/28	0.10928	5/9	10/28	0.00288
5/10	10/19	0.17027	5/9	10/19	0.03928
5/10	10/26	0.10521	5/9	10/26	0.00668
5/10	10/28	0.00672	5/9	10/28	0.00556
5/10	10/29	0.00579	5/9	10/29	0.00496
5/10	10/25	0.11286	5/9	10/25	0.01537
5/10	10/25	0.27857	5/7	10/25	0.00283
5/10	10/24	0.08268	5/7	10/24	0.00306
5/10	10/25	0.17328	5/7	10/25	0.00263
5/10	10/30	0.00561	5/7	10/30	0.0027
5/10	10/28	0.11056	5/7	10/28	0.0021
5/10	10/19	0.18628	5/7	10/19	0.00629
5/10	10/26	0.10613	5/7	10/26	0.00265
5/10	10/28	0.00679	5/7	10/28	0.00307

Table 6: McGregor on graphs with 5 and 10 nodes.

G_1 nodes/edges	G_2 nodes/edges	Time (s)	G_1 nodes/edges	G_2 nodes/edges	Time (s)
5/10	10/29	0.00578	5/7	10/29	0.0023
5/10	10/25	0.11207	5/7	10/25	0.00357
5/10	10/25	0.27437	5/8	10/25	0.01194
5/10	10/24	0.08266	5/8	10/24	0.00348
5/10	10/25	0.17937	5/8	10/25	0.00251
5/10	10/30	0.00561	5/8	10/30	0.00334
5/10	10/28	0.10973	5/8	10/28	0.00241
5/10	10/19	0.17006	5/8	10/19	0.03271
5/10	10/26	0.10499	5/8	10/26	0.00422
5/10	10/28	0.00672	5/8	10/28	0.00367
5/10	10/29	0.00579	5/8	10/29	0.00239
5/10	10/25	0.12118	5/8	10/25	0.00449

Table 7: McGregor on graphs with 5 and 10 nodes continued.

G_1 nodes/edges	G_2 nodes/edges	Time (s)	G_1 nodes/edges	G_2 nodes/edges	Time (s)
5/10	20/47	9.49598	5/10	20/47	9.48423
5/10	20/42	3.30214	5/10	20/42	3.29642
5/10	20/36	4.8306	5/10	20/36	4.82945
5/10	20/42	2.39904	5/10	20/42	2.39215
5/10	20/40	3.16369	5/10	20/40	3.15848
5/10	20/55	3.3082	5/10	20/55	3.29496
5/10	20/52	2.82093	5/10	20/52	2.81965
5/10	20/48	2.49535	5/10	20/48	2.4935
5/10	20/42	1.71912	5/10	20/42	1.72669
5/10	20/49	4.06498	5/10	20/49	4.0652
5/10	20/47	9.57288	5/9	20/47	3.87927
5/10	20/42	3.47059	5/9	20/42	0.96943
5/10	20/36	4.82355	5/9	20/36	1.69311
5/10	20/42	2.43258	5/9	20/42	0.58358
5/10	20/40	3.15565	5/9	20/40	1.35845
5/10	20/55	3.29954	5/9	20/55	0.62026
5/10	20/52	2.82496	5/9	20/52	0.49159
5/10	20/48	2.4946	5/9	20/48	0.55511
5/10	20/42	1.7202	5/9	20/42	0.32515
5/10	20/49	4.10385	5/9	20/49	1.06441
5/10	20/47	9.50666	5/9	20/47	1.93818
5/10	20/42	3.29232	5/9	20/42	0.73771
5/10	20/36	4.82993	5/9	20/36	1.0171
5/10	20/42	2.39494	5/9	20/42	0.38676
5/10	20/40	3.15848	5/9	20/40	0.59216
5/10	20/55	3.29907	5/9	20/55	0.41914
5/10	20/52	2.82155	5/9	20/52	0.36576
5/10	20/48	2.49394	5/9	20/48	0.49767
5/10	20/42	1.71828	5/9	20/42	0.27133
5/10	20/49	4.05254	5/9	20/49	0.67223
5/10	20/47	9.50277	5/7	20/47	0.08752
5/10	20/42	3.29923	5/7	20/42	0.01847
5/10	20/36	4.82647	5/7	20/36	0.05464
5/10	20/42	2.3961	5/7	20/42	0.03304
5/10	20/40	3.17261	5/7	20/40	0.03845
5/10	20/55	3.29973	5/7	20/55	0.0182
5/10	20/52	3.09788	5/7	20/52	0.01167
5/10	20/48	2.5741	5/7	20/48	0.01672

Table 8: McGregor on graphs with 5 and 20 nodes.

G_1 nodes/edges	G_2 nodes/edges	Time (s)	G_1 nodes/edges	G_2 nodes/edges	Time (s)
5/10	20/42	1.72449	5/7	20/42	0.02036
5/10	20/49	4.0515	5/7	20/49	0.01992
5/10	20/47	9.47856	5/8	20/47	0.49435
5/10	20/42	3.46557	5/8	20/42	0.16805
5/10	20/36	4.83699	5/8	20/36	0.25297
5/10	20/42	2.39967	5/8	20/42	0.06693
5/10	20/40	3.15988	5/8	20/40	0.41064
5/10	20/55	3.3011	5/8	20/55	0.0842
5/10	20/52	2.83458	5/8	20/52	0.06394
5/10	20/48	2.50464	5/8	20/48	0.08155
5/10	20/42	1.72719	5/8	20/42	0.04201
5/10	20/49	4.07225	5/8	20/49	0.41861

Table 9: McGregor on graphs with 5 and 20 nodes continued.

Appendix B Unlabelled graphs

The graphs presented here are five unlabelled graphs with anchored edges. Each anchor consists of three edges. All figures illustrate the randomly generated graphs. The numbers in the vertices are not labels, they are just used indices that separate the vertices.

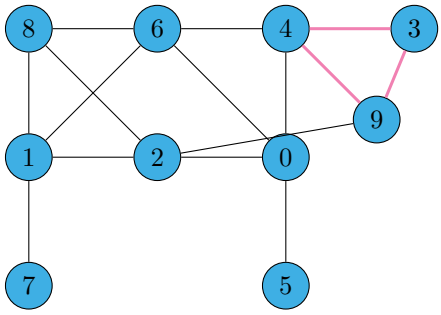


Figure 29: Graph 0

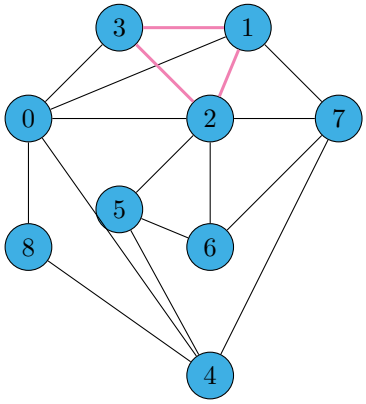


Figure 30: Graph 1

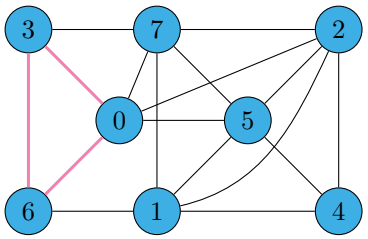


Figure 31: Graph 2

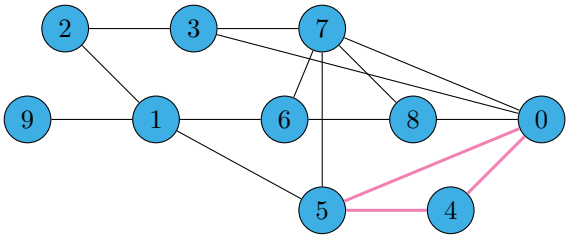


Figure 32: Graph 3

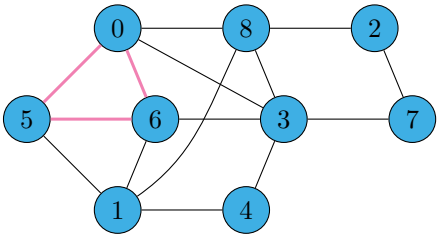
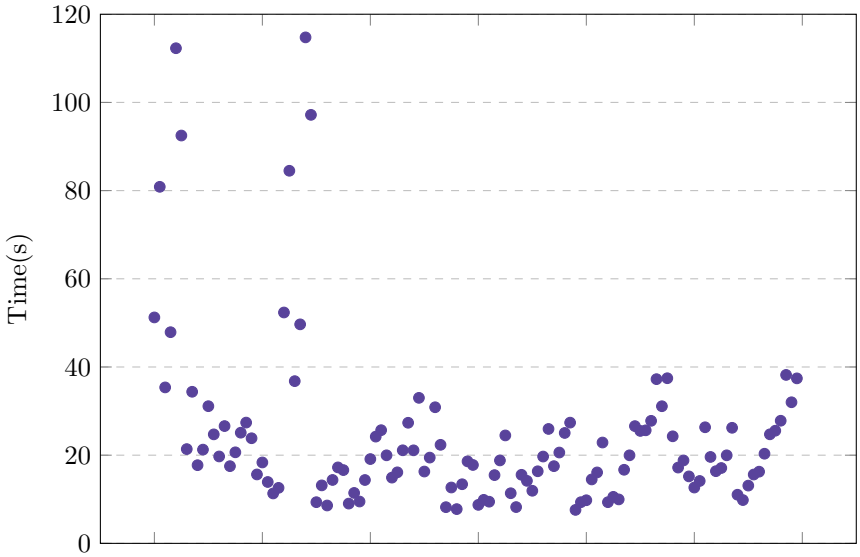


Figure 33: Graph 4

Appendix C Unlabelled graphs test data

The first figure illustrates the runtime of the algorithm when all five graphs are given to the iterative approach but in different order. The remaining tables represent the resulting test data for 2, 3, 4, and all 5 of the randomly generated graphs.



Permutations of “01234” on all five graphs.

graph seq	time (s)
0 1	1.8
0 2	1.1
0 3	0.84
0 4	0.53
1 0	1.96
1 2	0.61
1 3	0.59
1 4	0.17
2 0	1.11
2 1	0.56
2 3	1.11
2 4	0.37
3 0	0.87
3 1	0.56
3 2	0.92
3 4	0.47
4 0	0.56
4 1	0.17
4 2	0.37
4 3	0.49

Table 10: The runtime of all permutations with two of the five unlabelled graphs.

graph seq	time (s)	graph seq	time (s)
0 1 2	14.45	2 1 4	1.78
0 1 3	7.13	2 3 0	4.08
0 1 4	6.29	2 3 1	3.71
0 2 1	7.78	2 3 4	2.98
0 2 3	4.63	2 4 0	3.57
0 2 4	4.47	2 4 1	2.39
0 3 1	4.56	2 4 3	2.36
0 3 2	6.84	3 0 1	4.56
0 3 4	3.64	3 0 2	6.87
0 4 1	2.81	3 0 4	3.65
0 4 2	4.08	3 1 0	2.92
0 4 3	2.33	3 1 2	4.21
1 0 2	14.74	3 1 4	1.73
1 0 3	7.47	3 2 0	4.22
1 0 4	6.46	3 2 1	3.6
1 2 0	4.31	3 2 4	2.83
1 2 3	2.84	3 4 0	7.99
1 2 4	1.84	3 4 1	4.25
1 3 0	3.3	3 4 2	8.43
1 3 2	4.11	4 0 1	2.71
1 3 4	1.78	4 0 2	4.06
1 4 0	4.75	4 0 3	2.56
1 4 2	4.0	4 1 0	4.82
1 4 3	2.34	4 1 2	4.02
2 0 1	7.84	4 1 3	2.32
2 0 3	4.44	4 2 0	3.29
2 0 4	4.38	4 2 1	2.18
2 1 0	4.15	4 2 3	2.62
2 1 3	2.77	4 3 0	8.05
4 3 2	8.39	4 3 1	4.27

Table 11: The runtime of all permutations with three of the five unlabelled graphs.

graph seq	time (s)	graph seq	time (s)	graph seq	time (s)	graph seq	time (s)
0 1 2 3	31.86	3 2 4 1	12.28	1 2 0 4	7.08	2 3 1 0	7.62
0 1 2 4	30.99	3 4 0 1	13.47	1 2 3 0	6.56	2 3 1 4	7.48
0 1 3 2	22.52	3 4 0 2	17.75	1 2 3 4	6.85	2 3 4 0	13.12
0 1 3 4	14.82	3 4 1 0	16.12	1 2 4 0	12.74	2 3 4 1	12.29
0 1 4 2	59.19	3 4 1 2	20.91	1 2 4 3	8.35	2 4 0 1	7.58
0 1 4 3	33.11	3 4 2 0	23.04	1 3 0 2	6.58	2 4 0 3	5.65
0 2 1 3	14.51	3 4 2 1	20.12	1 3 0 4	4.98	2 4 1 0	11.71
0 2 1 4	15.45	4 0 1 2	14.18	1 3 2 0	7.81	2 4 1 3	7.91
0 2 3 1	11.26	4 0 1 3	7.38	1 3 2 4	7.23	2 4 3 0	8.39
0 2 3 4	10.09	4 0 2 1	10.39	1 3 4 0	10.51	2 4 3 1	8.51
0 2 4 1	16.83	4 0 2 3	8.86	1 3 4 2	12.55	3 0 1 2	12.5
0 2 4 3	12.95	4 0 3 1	5.88	1 4 0 2	16.25	3 0 1 4	8.4
0 3 1 2	12.49	4 0 3 2	8.79	1 4 0 3	9.38	3 0 2 1	12.1
0 3 1 4	8.56	4 1 0 2	16.26	1 4 2 0	11.65	3 0 2 4	11.22
0 3 2 1	12.32	4 1 0 3	9.23	1 4 2 3	10.01	3 0 4 1	10.61
0 3 2 4	11.5	4 1 2 0	12.35	1 4 3 0	11.31	3 0 4 2	16.05
0 3 4 1	10.71	4 1 2 3	10.05	1 4 3 2	15.65	3 1 0 2	5.56
0 3 4 2	16.15	4 1 3 0	10.82	2 0 1 3	14.27	3 1 0 4	4.4
0 4 1 2	14.58	4 1 3 2	15.72	2 0 1 4	15.36	3 1 2 0	8.09
0 4 1 3	7.02	4 2 0 1	6.58	2 0 3 1	10.56	3 1 2 4	7.26
0 4 2 1	10.6	4 2 0 3	6.11	2 0 3 4	9.45	3 1 4 0	9.01
0 4 2 3	8.42	4 2 1 0	9.07	2 0 4 1	16.83	3 1 4 2	12.92
0 4 3 1	5.27	4 2 1 3	7.85	2 0 4 3	12.38	3 2 0 1	6.9
0 4 3 2	7.94	4 2 3 0	11.29	2 1 0 3	6.27	3 2 0 4	6.4
1 0 2 3	32.56	4 2 3 1	10.1	2 1 0 4	6.78	3 2 1 0	7.87
1 0 2 4	31.26	4 3 0 1	13.54	2 1 3 0	6.14	3 2 1 4	7.47
1 0 3 2	23.54	4 3 0 2	17.51	2 1 3 4	6.48	3 2 4 0	13.44
1 0 3 4	15.5	4 3 1 0	15.89	2 1 4 0	12.72	1 2 0 3	6.64
1 0 4 2	59.84	4 3 1 2	21.37	2 1 4 3	8.2	4 3 2 1	21.08
1 0 4 3	34.37	4 3 2 0	23.45	2 3 0 1	6.47	2 3 0 4	6.08

Table 12: The runtime of all permutations with four of the five unlabelled graphs.

graph seq	time (s)	graph seq	time (s)	graph seq	time (s)
0 1 2 3 4	51.24	2 3 0 1 4	8.73	1 4 0 2 3	25.67
0 1 2 4 3	80.86	2 3 0 4 1	9.87	1 4 0 3 2	19.97
0 1 3 2 4	35.37	2 3 1 0 4	9.45	1 4 2 0 3	14.91
0 1 3 4 2	47.89	2 3 1 4 0	15.48	1 4 2 3 0	16.11
0 1 4 2 3	112.29	2 3 4 0 1	18.81	1 4 3 0 2	21.11
0 1 4 3 2	92.49	2 3 4 1 0	24.48	1 4 3 2 0	27.35
0 2 1 3 4	21.38	2 4 0 1 3	11.37	2 0 1 3 4	21.1
0 2 1 4 3	34.38	2 4 0 3 1	8.23	2 0 1 4 3	33.0
0 2 3 1 4	17.73	2 4 1 0 3	15.55	2 0 3 1 4	16.29
0 2 3 4 1	21.26	2 4 1 3 0	14.18	2 0 3 4 1	19.45
0 2 4 1 3	31.11	2 4 3 0 1	11.93	2 0 4 1 3	30.87
0 2 4 3 1	24.72	2 4 3 1 0	16.35	2 0 4 3 1	22.35
0 3 1 2 4	19.71	3 0 1 2 4	19.69	2 1 0 3 4	8.21
0 3 1 4 2	26.6	3 0 1 4 2	25.95	2 1 0 4 3	12.67
0 3 2 1 4	17.51	3 0 2 1 4	17.51	2 1 3 0 4	7.77
0 3 2 4 1	20.65	3 0 2 4 1	20.6	2 1 3 4 0	13.41
0 3 4 1 2	25.1	3 0 4 1 2	25.04	2 1 4 0 3	18.6
0 3 4 2 1	27.39	3 0 4 2 1	27.38	2 1 4 3 0	17.78
0 4 1 2 3	23.84	3 1 0 2 4	7.59	4 1 0 2 3	26.33
0 4 1 3 2	15.63	3 1 0 4 2	9.34	4 1 0 3 2	19.6
0 4 2 1 3	18.35	3 1 2 0 4	9.78	4 1 2 0 3	16.37
0 4 2 3 1	13.92	3 1 2 4 0	14.5	4 1 2 3 0	17.09
0 4 3 1 2	11.32	3 1 4 0 2	16.09	4 1 3 0 2	19.97
0 4 3 2 1	12.59	3 1 4 2 0	22.87	4 1 3 2 0	26.21
1 0 2 3 4	52.36	3 2 0 1 4	9.34	4 2 0 1 3	11.03
1 0 2 4 3	84.5	3 2 0 4 1	10.55	4 2 0 3 1	9.82
1 0 3 2 4	36.79	3 2 1 0 4	9.97	4 2 1 0 3	13.1
1 0 3 4 2	49.67	3 2 1 4 0	16.69	4 2 1 3 0	15.61
1 0 4 2 3	114.75	3 2 4 0 1	19.97	4 2 3 0 1	16.26
1 0 4 3 2	97.18	3 2 4 1 0	26.59	4 2 3 1 0	20.33
1 2 0 3 4	9.34	3 4 0 1 2	25.5	4 3 0 1 2	24.73
1 2 0 4 3	13.15	3 4 0 2 1	25.64	4 3 0 2 1	25.55
1 2 3 0 4	8.59	3 4 1 0 2	27.79	4 3 1 0 2	27.81
1 2 3 4 0	14.38	3 4 1 2 0	37.24	4 3 1 2 0	38.2
1 2 4 0 3	17.23	3 4 2 0 1	31.1	4 3 2 0 1	32.0
1 2 4 3 0	16.62	3 4 2 1 0	37.45	4 3 2 1 0	37.42
1 3 0 2 4	9.02	4 0 1 2 3	24.28	4 0 3 1 2	12.66
1 3 0 4 2	11.44	4 0 1 3 2	17.19	4 0 3 2 1	14.15
1 3 2 0 4	9.49	4 0 2 1 3	18.81	1 3 4 2 0	24.21
1 3 2 4 0	14.37	4 0 2 3 1	15.19	1 3 4 0 2	19.12

Table 13: The runtime of all permutations with the five unlabelled graphs.

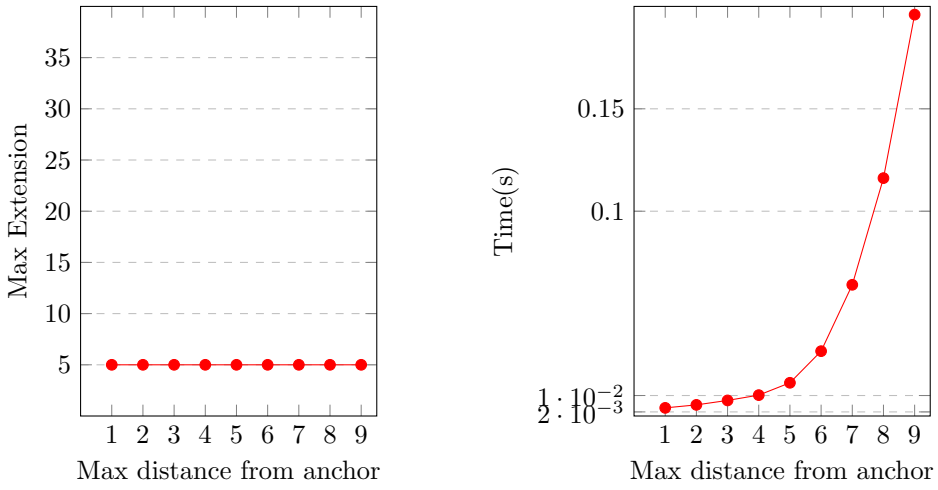
Appendix D Enzymatic reaction graphs test data

This data set represents tests of the clique algorithm on Graphs modelling molecules supplied with anchored Edges, atom and bond labels on vertices and edges respectively. For each test instance there is a figure containing the development of the runtime compared to the max distance from the anchor, as well as the corresponding table of data.

Additionally, the appendix includes tables containing descriptions of the graphs in all instances.

D.1 Graph results

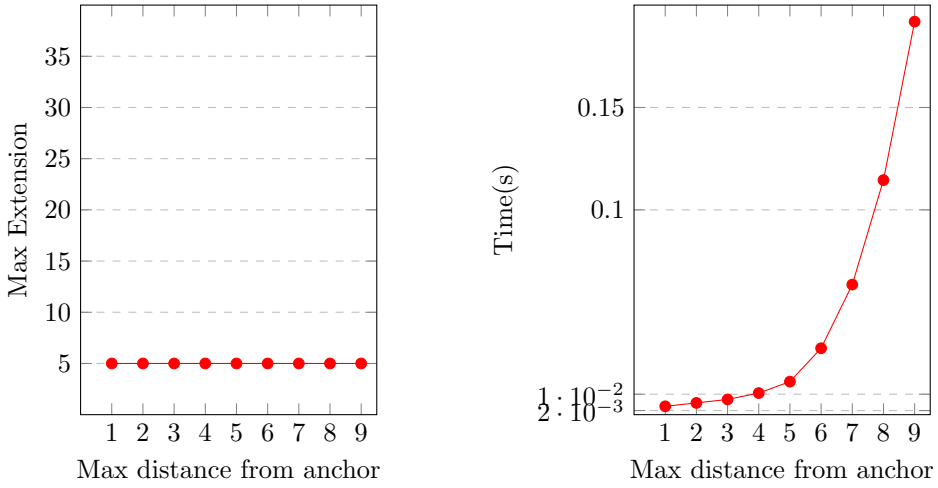
Acetate Kinase Backward (AKB)



Max distance	Max extension	Time(s)
infinity	5	0.75766
1	5	0.00397
2	5	0.00543
3	5	0.00761
4	5	0.0102
5	5	0.01622
6	5	0.03166
7	5	0.06407
8	5	0.11616
9	5	0.19599

Table 14: Acetate Kinase Backward (AKB)

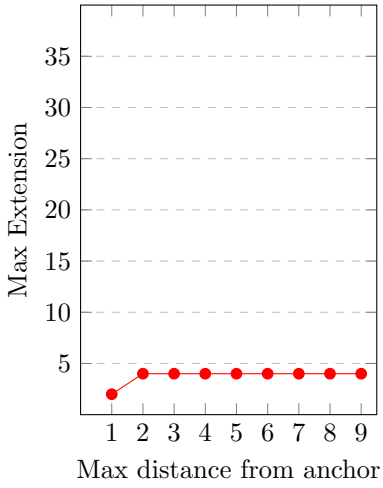
Acetate Kinase Forward (AKF)

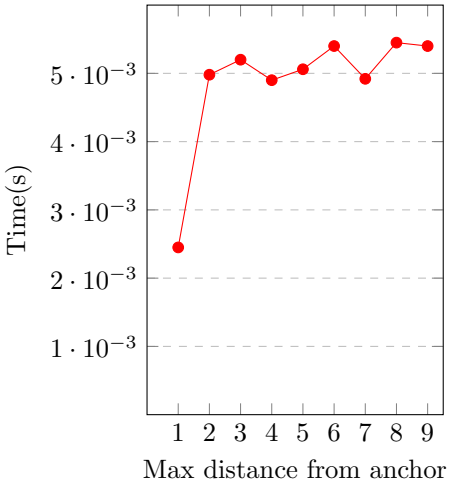


Max distance	Max extension	Time(s)
infinity	5	0.73113
1	5	0.00409
2	5	0.00574
3	5	0.00744
4	5	0.01055
5	5	0.01613
6	5	0.03244
7	5	0.06352
8	5	0.11453
9	5	0.19192

Table 15: Acetate Kinase Forward (AKF)

Aconitase Half-Reaction A Citrate Hydro-lyase Backward (AHACHB)

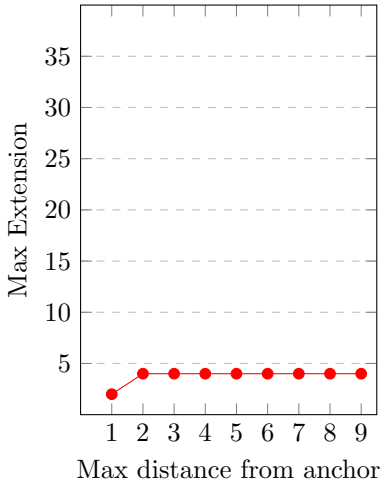


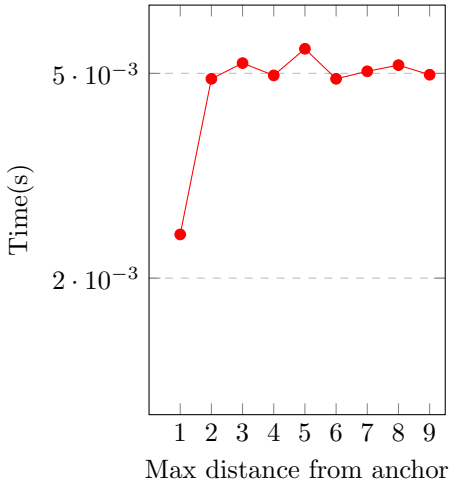


Max distance	Max extension	Time(s)
infinity	4	0.00491
1	2	0.00245
2	4	0.00498
3	4	0.0052
4	4	0.0049
5	4	0.00506
6	4	0.0054
7	4	0.00492
8	4	0.00545
9	4	0.0054

Table 16: Aconitase Half-Reaction A Citrate Hydro-lyase Backward (AHACHB)

Aconitase Half-Reaction A Citrate Hydro-lyase Forward (AHACHF)

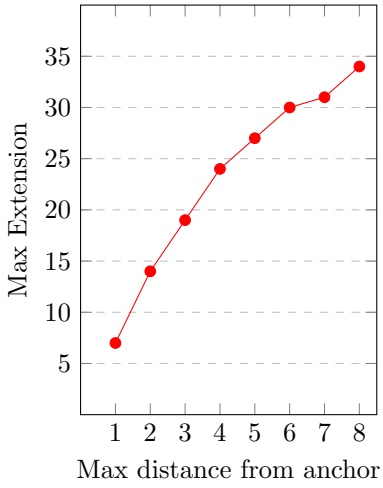


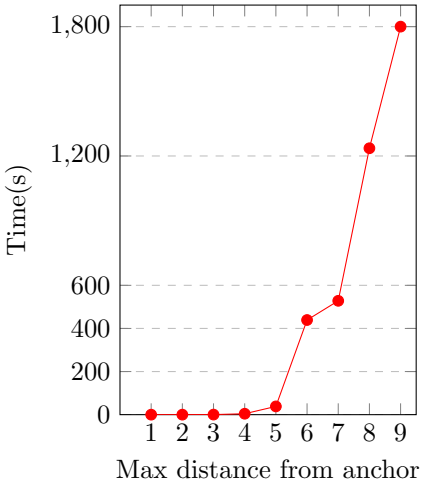


Max distance	Max extension	Time(s)
infinity	4	0.00509
1	2	0.00264
2	4	0.00492
3	4	0.00515
4	4	0.00497
5	4	0.00536
6	4	0.00492
7	4	0.00503
8	4	0.00512
9	4	0.00498

Table 17: Aconitase Half-Reaction A Citrate Hydro-lyase Forward (AHACHF)

Alcohol Dehydrogenase Ethanol Backward (ADEB)

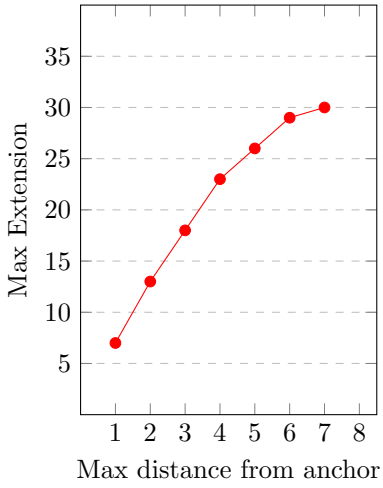


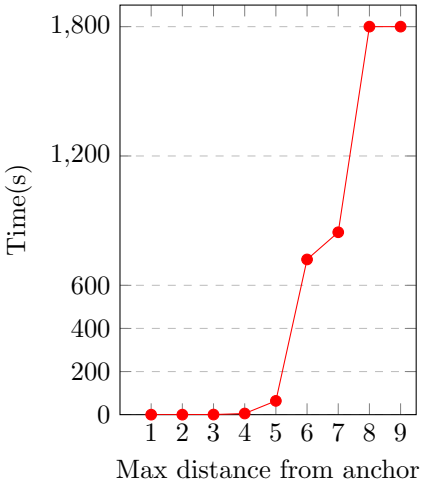


Max distance	Max extension	Time(s)
infinity	-	timed out after 1800 seconds
1	7	0.00798
2	14	0.05179
3	19	0.33891
4	24	4.32864
5	27	38.29215
6	30	439.06321
7	31	528.13756
8	34	1235.89867
9	-	timed out after 1800 seconds

Table 18: Alcohol Dehydrogenase Ethanol Backward (ADEB)

Alcohol Dehydrogenase Ethanol Forward (ADEF)

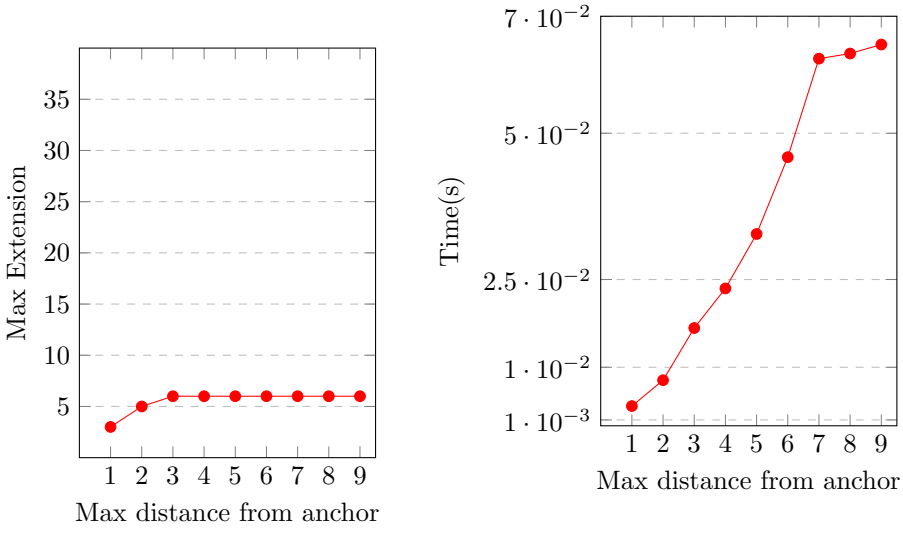




Max distance	Max extension	Time(s)
infinity	-	timed out after 1800 seconds
1	7	0.02538
2	13	0.07016
3	18	0.48703
4	23	5.34438
5	26	63.95696
6	29	720.00129
7	30	846.17031
8	-	timed out after 1800 seconds
9	-	timed out after 1800 seconds

Table 19: Alcohol Dehydrogenase Ethanol Forward (ADEF)

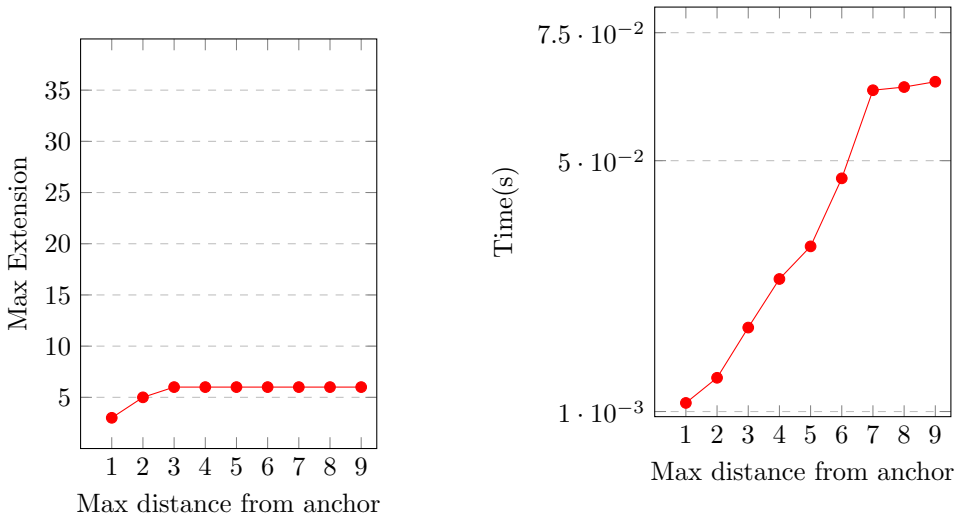
Fructose Bisphosphatase (FB)



Max distance	Max extension	Time(s)
infinity	6	0.1402
1	3	0.00337
2	5	0.00779
3	6	0.01671
4	6	0.02347
5	6	0.03278
6	6	0.04591
7	6	0.06275
8	6	0.06362
9	6	0.06516

Table 20: Fructose Bisphosphatase (FB)

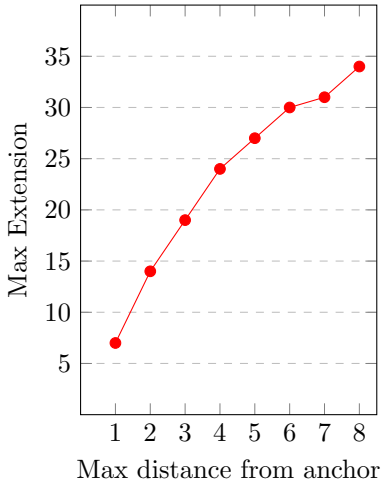
Generic Phosphatase (GP)

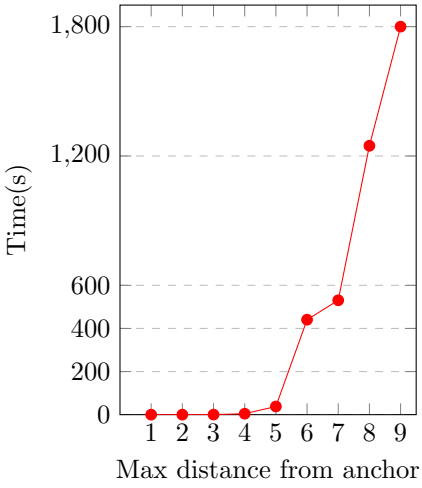


Max distance	Max extension	Time(s)
infinity	6	0.07348
1	3	0.00268
2	5	0.00759
3	6	0.0174
4	6	0.02689
5	6	0.03326
6	6	0.04655
7	6	0.06376
8	6	0.06438
9	6	0.06542

Table 21: Generic Phosphatase (GP)

Glucose 6 Phosphate Dehydrogenase Backward (GPDB)

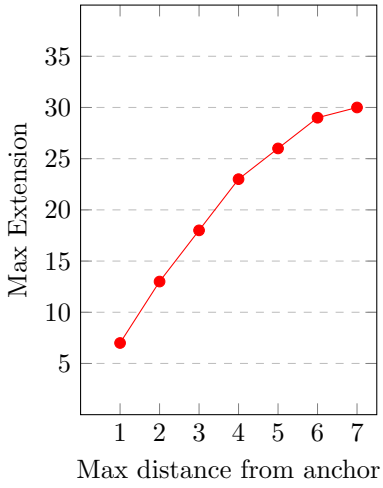


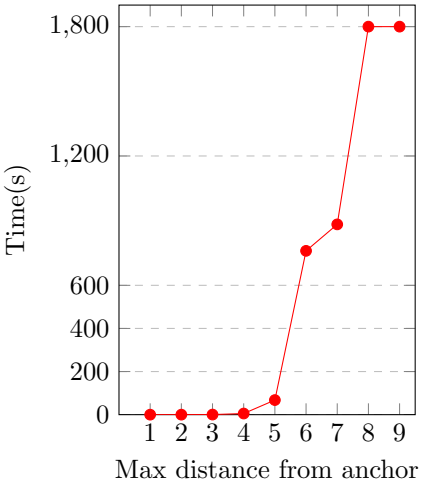


Max distance	Max extension	Time(s)
infinity	-	timed out after 1800 seconds
1	7	0.01695
2	14	0.05962
3	19	0.34715
4	24	4.43406
5	27	37.96491
6	30	440.50666
7	31	530.58584
8	34	1247.38214
9	-	timed out after 1800 seconds

Table 22: Glucose 6 Phosphate Dehydrogenase Backward (GPDB)

Glucose 6 Phosphate Dehydrogenase Forward (GPDF)

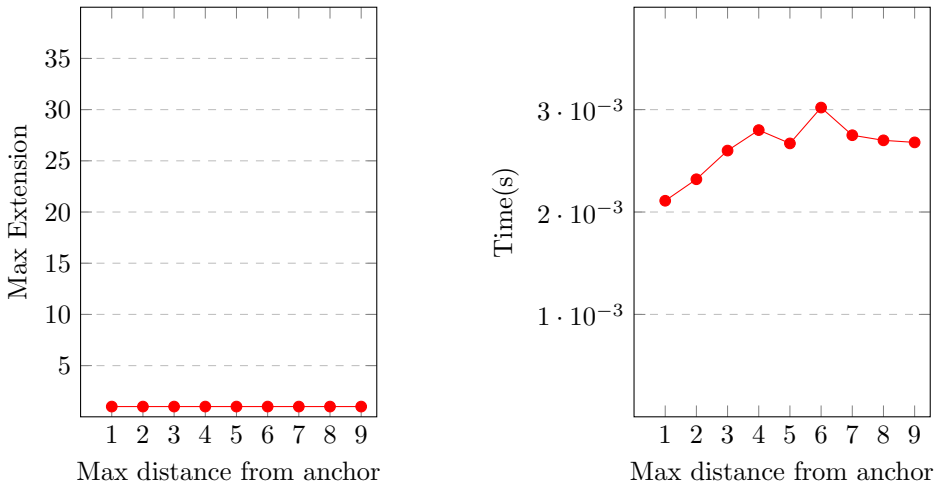




Max distance	Max extension	Time(s)
infinity	-	timed out after 1800 seconds
1	7	0.01225
2	13	0.06343
3	18	0.4396
4	23	5.2616
5	26	67.71842
6	29	759.92566
7	30	882.70626
8	-	timed out after 1800 seconds
9	-	timed out after 1800 seconds

Table 23: Glucose 6 Phosphate Dehydrogenase Forward (GPDF)

Glucose 6 Phosphate Isomerase Forward (GPIF)



Max distance	Max extension	Time(s)
infinity	1	0.00263
1	1	0.00211
2	1	0.00232
3	1	0.0026
4	1	0.0028
5	1	0.00267
6	1	0.00302
7	1	0.00275
8	1	0.0027
9	1	0.00268

Table 24: Glucose 6 Phosphate Isomerase

D.2 Graph information

The following tables for each test instance provides information regarding each of the graphs in the test instances. For each graph the number of nodes and edges, as well as the left hand side and the right hand side of the reaction is given. Lastly, a table over the "good" sequence chosen for each test instance.

Graph	# Nodes	# Edges	Reaction
0	50	52	adp → ac + atp
1	78	83	amp + atp → 2 adp
2	78	83	2 adp → atp + amp
3	58	60	adp + 13dpg → atp + 3pg

Table 25: **Acetate Kinase Backward (AKB)**

Graph	# Nodes	# Edges	Reaction
0	50	52	ac + atp → actp adp
1	78	83	atp + amp → 2 adp
2	78	83	2 adp → atp + amp
3	58	60	atp + 3pg → 13dpg + adp

Table 26: **Acetate Kinase Forward (AKF)**

Graph	# Nodes	# Edges	Reaction
0	18	18	acon-C + h2o → cit
1	18	18	acon-C + h2o → icit
2	15	15	pep + h2o → 2pg
3	13	13	fum + h2o → mal-L

Table 27: **Aconitase Half-Reaction A Citrate Hydro-lyase Backward (AHACHB)**

Graph	# Nodes	# Edges	Reaction
0	18	18	cit → acon-C + h2o
1	18	18	icit → acon-C + h2o
2	15	15	2pg → pep + h2o
3	13	13	mal-L → fum + h2o

Table 28: **Aconitase Half-Reaction A Citrate Hydro-lyase Forward (AHACHF)**

Graph	# Nodes	# Edges	Reaction
0	79	83	nadh + acald + h → nad + etoh
1	81	85	nadh + pyr + h → nad + lac-D
2	83	87	nadh + oaa + h → nad + mal-L

Table 29: **Alcohol Dehydrogenase Ethanol Backward (ADEB)**

Graph	# Nodes	# Edges	Reaction
0	79	83	nad + etoh → nadh + acald + h
1	81	85	nad + lac-D → nadh + pyr + h
2	83	87	nad + mal-L → nadh + oaa + h
3	89	93	nad + 4pe → nadh + pdxb_r + h

Table 30: **Alcohol Dehydrogenase Ethanol Forward (ADEF)**

Graph	# Nodes	# Edges	Reaction
0	33	34	fdp + h2o → f6p + pi
1	144	154	nadph + nad → nadh + nadp
2	15	16	2pg → 3pg
3	15	16	3pg → 2pg

Table 31: **Fructose Bisphosphatase (FB)**

Graph	# Nodes	# Edges	Reaction
0	33	34	h2o + fdp → pi + f6p
1	144	154	nad + nadph → nadh + nadp
2	15	16	2pg → 3pg
3	15	16	3pg → 2pg

Table 32: **Generic Phosphatase (GP)**

Graph	# Nodes	# Edges	Reaction
0	79	83	nadh + acald + h → nad + etoh
1	81	85	nadh + pyr + h → nad + lac-D
2	83	87	nadh + oaa + h → nad + mal-L

Table 33: **Glucose 6 Phosphate Dehydrogenase Backward (GPDB)**

Graph	# Nodes	# Edges	Reaction
0	79	83	nad + etoh → nadh + acald + h
1	81	85	nad + lac-D → nadh + pyr + h
2	83	87	nad + mal-L → nadh + oaa + h
3	89	93	nad + 4pe → nadh + pdxb_r + h

Table 34: **Glucose 6 Phosphate Dehydrogenase Forward (GPDF)**

Graph	# Nodes	# Edges	Reaction
0	15	16	2pg → 3pg
1	15	16	3pg → 2pg
2	15	16	dhap → g3p
3	15	16	g3p → dhap

Table 35: **Glucose 6 Phosphate Isomerase Forward (GPIF)**

Instance	Sequence used
AKB	2 1 0 3
AKF	2 1 0 3
AHACHB	2 1 0 3
AHACHF	2 1 0 3
ADEB	0 2 1
ADEF	0 3 1 2
FB	0 2 1 3
GP	0 2 1 3
GPIF	0 1 2 3
GPDB	0 2 1
GPDF	0 3 2 1
PD	—

Table 36: The "good" sequence for each of the test instances. The last test instance **Phosphogluconate Dehydrogenase (PD)** has not been tested, as we were unable to finish computing anchors.

Appendix E Aconitase Half-Reaction A Citrate Hydrolyase Backward illustration

This appendix contains the four graphs contained in AHACHB along with their anchor and their found maximum extension of four edges. The anchored edges in each graph are depicted in orange, and the found extension is depicted in pink. Parallel edges where one of the bonds is dashed illustrates the change from a double bond to a single bond.

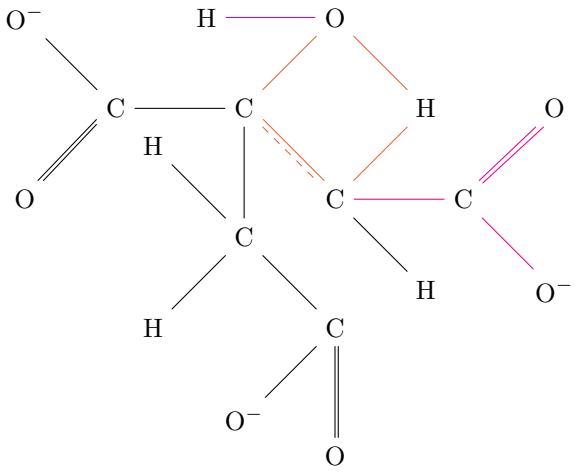


Figure 45: Graph 0

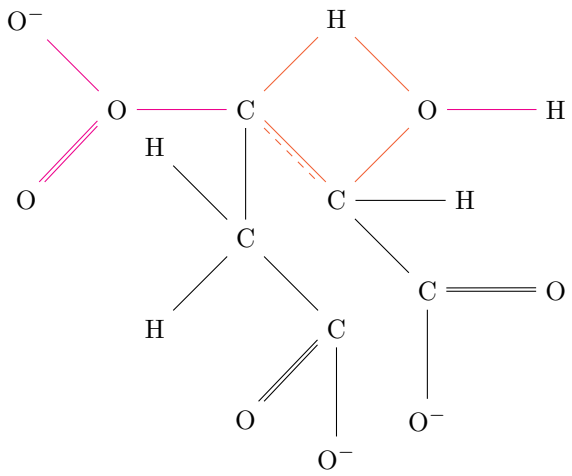


Figure 46: Graph 1

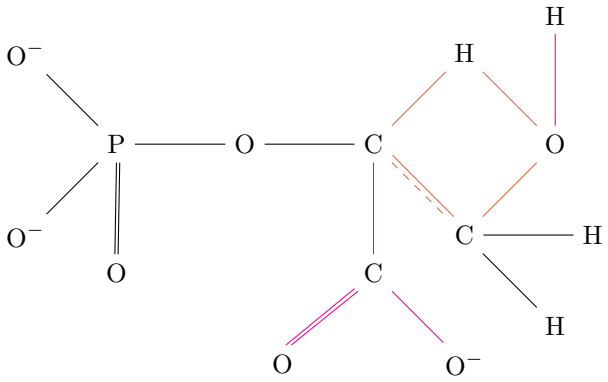


Figure 47: Graph 2

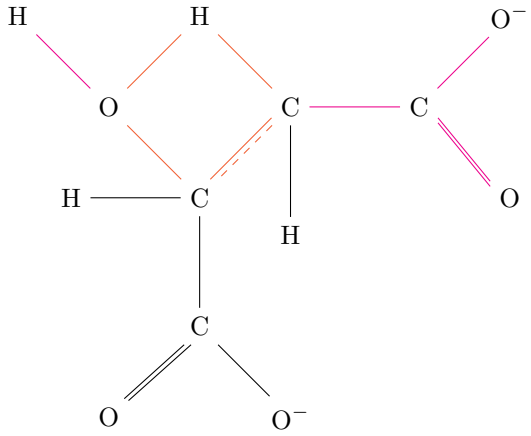


Figure 48: Graph 3

References

[AMR19] J. L. Andersen, D. Merkle, and P. S. Rasmussen. "Graph Transformations, Semigroups, and Isotopic Labeling". In: *Bioinformatics Research and Applications*. Ed. by Z. Cai, P. Skums, and M. Li. Cham: Springer International Publishing, 2019, pp. 196–207. DOI: https://doi.org/10.1007/978-3-030-20242-2_17.

[And+16] J. L. Andersen et al. "A Software Package for Chemically Inspired Graph Transformation". In: *Graph Transformation*. Ed. by R. Echahed and M. Minas. Cham: Springer International Publishing, 2016, pp. 73–88. DOI: https://doi.org/10.1007/978-3-319-40530-8_5.

[And+21] J. L. Andersen et al. "Graph transformation for enzymatic mechanisms". In: *Bioinformatics* 37.Supplement_1 (July 2021), pp. i392–i400. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btab296. eprint: https://academic.oup.com/bioinformatics/article-pdf/37/Supplement_1/i392/42208562/btab296.pdf.

[And+22] J. L. Andersen et al. "Representing Catalytic Mechanisms with Rule Composition". In: *Journal of Chemical Information and Modeling* 62.22 (2022). PMID: 36326605, pp. 5513–5524. DOI: 10.1021/acs.jcim.2c00426. eprint: https://doi.org/10.1021/acs.jcim.2c00426. URL: https://doi.org/10.1021/acs.jcim.2c00426.

[AS05] P. Swart A. Hagberg and D. Schult. *NetworkX*. 2005. URL: https://networkx.org/.

[BB76] H. G. Barrow and R. M. Burstall. "Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques". In: *Inf. Process. Lett.* 4.4 (1976), pp. 83–84. DOI: 10.1016/0020-0190(76)90049-1. URL: https://doi.org/10.1016/0020-0190(76)90049-1.

[Con23] Wikipedia Contributers. *Metabolic pathways*. 2023. URL: https://en.wikipedia.org/wiki/Metabolic_pathway.

[Dav23] A. Davoodi. "An Application of Graph Products in Rule Inference". In: *Slides for TBI Wintersemina, Slovenia* (2023).

[FKW22] P. Formanowicz, M. Kasprzak, and P. Wawrzyniak. "Labeled Graphs in Life Sciences—Two Important Applications". In: *Graph-Based Modelling in Science, Technology and Art*. Ed. by Stanisław Zawiślak and Jacek Rysiński. Cham: Springer International Publishing, 2022, pp. 201–217. DOI: 10.1007/978-3-030-76787-7_10.

[GJ79] M. R. Garey and D. S. Johnson. “Computers and intractability: A”. In: vol. 174. freeman San Francisco, 1979, p. 202.

[HMP01] A. Habel, J. Müller, and D. Plump. “Double-pushout graph transformation revisited”. In: *Mathematical Structures in Computer Science* 11.5 (2001), pp. 637–688. DOI: 10.1017/S0960129501003425.

[Lev73] G. Levi. “A note on the derivation of maximal common subgraphs of two directed or undirected graphs”. In: *Calcolo* 9.4 (1973), pp. 341–352. DOI: 10.1007/BF02575586.

[LP49] R.D. Luce and A.D. Perry. “A method of matrix analysis of group structure”. In: *Psychometrika* 14 (1949), pp. 95–116. DOI: 10.1007/BF02289146.

[McG82] J. J. McGregor. “Backtrack search algorithms and the maximal common subgraph problem”. In: *Software: Practice and Experience* 12.1 (1982), pp. 23–34. DOI: 10.1002/spe.4380120103.

[MP11] B. D. McKay and A. Piperno. *Nauty and Traces*. 2011. URL: https://pallini.di.uniroma1.it/.

[MU04] K. Makino and T. Uno. “New Algorithms for Enumerating All Maximal Cliques”. In: *Algorithm Theory - SWAT 2004*. Ed. by T. Hagerup and J. Katajainen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 260–272. DOI: 10.1007/978-3-540-27810-8_23.

[Pel09] M. Pelillo. “Heuristics for Maximum Clique and Independent Set”. In: *Encyclopedia of Optimization*. Ed. by C. A. Floudas and P. M. Pardalos. Boston, MA: Springer US, 2009, pp. 1508–1520. DOI: 10.1007/978-0-387-74759-0_264.

[Wes+01] D. B. West et al. *Introduction to graph theory*. Vol. 2. Prentice hall Upper Saddle River, 2001.