GRAFTRANSFORMATION

siman16, madsn17

1. JUNI 2021

SIMON VEILING VANGSØ ANDERSEN
MADS NICOLAJ NIELSEN

Abstract

Chemical reactions can be generated and modelled using graph transformation rules, where these rules model the react pattern. When modelling with chemistry, we use undirected, labelled graphs to model the molecules. This is done to properly represent bonds and atoms as nodes and edges in a graph, so it is possible to represent large chemical compounds. This is all since molecules and reactions often involve multiple molecules, so the transformation must be able to perform on multisets of graphs. Using the graph rewriting formalism called the Double Pushout Approach, which is rooted in category theory, we can formulate the transition rules of these chemical reactions. Furthermore, we are describing several definitions in category theory to understand the Double Pushout Approach fully. This includes the production of graphs including graph-morphisms in a chemical intuitive manner. For the generation of graphs, we use the VF2 algorithm for the testing of morphisms between graphs, which is essential in the construction of the graphs and graph transformation.

This study aims towards being able to perform the transformation for these types of graphs to model chemical reactions. To present this, we have made a small program, which contains multiple algorithms for working with transformation rules the Double Pushout way. This includes the generation of graphs and rules as well as a morphism-testing tool. Furthermore, the program can make the actual graph transformation with focus on the generation of chemical reactions. Labels are added to the nodes to properly represent the nodes as atoms, however as of now, labels are not present on the edges. Unfortunately, this means that the actual transformation most likely will not perform the expected way when looking at the labelled edges. The program does not only make the actual graph transformation, but it also features a graph- and transformation rule visualizing tool.

The program has been used on simple graphs, but also on compound chemical systems, such as the formose process, all with success (considering the edges are not with labels). However, it does not feature an automatic transformation algorithm, which is able to use multiple graphs and rules as input and plot the result. Rather, it only takes a single rule and a single (compound) graph as an input and performs the transformation on that input.

Indholdsfortegnelse

Liste af figurer

Kapitel 1: Introduktion

Konceptet bag graftransformation har rødder helt tilbage i 1960'erne og 1970'erne, hvor man allerede dengang begyndte at udvikle modeller for transformation af grafer.[1] Idéen var, at man begyndte at generalisere velkendte omskrivningsteknikker fra blandt andet strenge til grafer for at få graftransformationer. Den fundamentale anvendelse for brugen af denne teknik var inden for regel-baseret billede genkendelse[2] og oversættelsen af diagramsprog[3]. Det er dog først inden for de seneste årtier, at det virkelig har haft en stor betydning inden for mange fag. Graftransformation spiller en betydelig rolle inden for teoretisk datalogi, men har også en central rolle i anvendte områder inden for software engineering, visualisering af modeller, model transformation osv.[4] Det er dog ikke kun inden for datalogi, at graftransformation har en stor rolle. Også inden for områder som biologi og kemi er graftransformation inden for nyere tid blevet anvendt.[5] Når man snakker om graftransformation inden for kemi, snakker man blandt andet om brugen af regel-baserede omskrivnings teknikker til at generere molekyler. Det er almen praksis at modellere disse molekyler som ikke-orienterede grafer, hvor atomerne og forbindelserne mellem atomer bliver repræsenteret som knuderne og kanterne i en graf.[6]

Det er altså tydeligt, at brugen af graftransformation spiller en stor rolle inden for mange fag i dag. I denne rapport, vil vi primært fokusere på at snakke om graftransformation generelt set, men med retning i mod, hvordan graftransformation bliver brugt inden for kemi. Dette betyder blandt andet, at når vi f.eks. omtaler en graf, har vi henblik på at snakke om ikke-orienterede grafer med labels, frem for orienterede grafer, da det, som sagt, er normaliteten for at beskrive molekyler inden for grafer.

Denne rapport er inddelt i flere forskellige dele, hvor vi vil komme ind på forskellige aspekter. Vi vil først og fremmest komme ind på mere formelt set hvad en graf er, og herfra bevæge os over imod graf morfier. Dette inkluderer forskellige typer af graf morfier, da en central del inden for graftransformation handler om at finde sammenhænge mellem forskellige grafer, for at se om to grafer f.eks. repræsenterer de samme molekyler. Vi vil ydermere komme ind på hvordan vi finder disse sammenhænge (graf-morfier), hvor vi blandt andet kort kommer ind på VF2 algoritmen, som er den algoritme vi bruger for at finde morfierne.

[1] [Ehrig et al. 2006, forord]
[2] [PR69]
[3] [Pra71]
[4] [Ehrig et al. 2006, forord]
[5] [Ehrig et al. 2006, side 19]
[6] [Dittrich et al. 2001]

I forhold til selve graftransformationen, gør vi brug af fremgangsmåden, som hedder *Double Pushout* (DPO), som er baseret på kategoriteori. Vi vil komme ind på forskellige aspekter inden for kategoriteori, samt komme ind på hvad et *pushout* inden for graftransformation er, med retning mod at fortælle om hvordan DPO konkret fungerer. Vi vil ydermere komme ind på, hvordan transformationsregler inden for DPO fungerer.

I den sidste halvdel af rapporten vil vi komme ind på vores implementering af graftransformation inden for ikke-orienterede grafer. Det er her vi vil komme ind på, hvordan vi har repræsenteret grafer, morfier, regler osv. Vi vil også komme ind på, hvordan vi har opsat vores algoritme til at lave graftransformationen, samt komme ind på, hvordan vi har gjort brug af VF2-algoritmen til at finde morfier, som er nødvendig for at vi kan lave graftransformationen. I forhold til hvordan vi tester vores program, vil vi have fokus på at anvende programmet formosereaktionen[7]. Det betyder, at generelt set, når vi snakker om forskellige molekyler, snakker vi primært kun om molekyler, der indgår i den proces.

[7] Kapitel 5

Kapitel 2: Grafer og graf-morfier

En essentiel del, når vi snakker om graftransformation, er at finde såkaldte 'delgrafer', som vi vil transformere, altså at finde den delmængde i grafen, som vi gerne vil lave mønstergenkendelse ved. I forhold til kemiske molekyler, svarer dette til at finde substrukturerer af molekylerne. En anden vigtig del er at finde ud af, om disse kemiske molekyler/grafer deler information, for blandt andet at finde ud af, om der er duplikationer af osv. I dette kapitel, vil vi snakke om og give en kort introduktion til grafer, hvad er en graf, og hvad består den af. Vi vil desuden komme ind på graf-morfier, som altså er dem, der beskriver hvordan grafer er relateret til hinanden. Der er flere forskellige typer af graf-morfier, og vi vil nævne nogle stykker af dem i dette kapitel. I forbindelse med, at vi primært fokuserer på molekyler, modellering og forbindelsen mellem visse molekyler, samt transformation af molekyler, betyder det, at når vi snakker om grafer, vil som sagt ikke have noget med direkte grafer at gøre. Vi beskriver dem derfor således:

2.1 Grafer

En graf består af knuder (V) og kanter (E). En kant er hvad forbinder to knuder. I takt med, at vi ikke fokuserer på orienterede grafer, har vi hverken start knude eller slut knude. Vi tillader desuden ikke parallelle kanter, da vi vil fokusere på at modellere mod kemiske molekyler, hvor bindingerne mellem atomerne er enkle enheder. Selvom, at der dog eksisterer dobbeltbindinger inden for kemi, som indeholder det dobbelte antal elektroner end en enkeltbinding, agerer den ikke på samme måde, som to enkeltbindinger, og vi har derfor valgt ikke at tillade parallelle kanter.[8] Figur 1 er et eksempel på en graf med parallelle kanter.

Figur 1: en graf med to knuder og parallelle kanter imellem dem

Med ovenstående information taget i betragtning, beskriver vi grafer således:

Definition 2.1.1 *(grafer):* En simpel ikke-orienteret graf, $G(V, E)$ er en graf, som består af en mængde af knuder og en mængde af kanter, eventuelt med labels. Den består hverken af en *source (kilde)* eller *target (mål)* funktion.[9] Grunden til, at den ikke består af disse to funktioner, er pga. at hvis vi har et par af to knuder $(v, u) \in V$, som er associeret med hver kant $e \in E$, så svarer v til kilden af e og u svarer til målet af e. Vi arbejder dog imidlertid med ikke-orienterede grafer, hvilket vil sige, at kanterne er bidirektionelle, og vi har hverken en kilde- eller en målknude.

[8] [Andersen et al. 2016, side 4]

[9] [Ehrig et al. 2006, side 21]

2.2 Graf-morfier

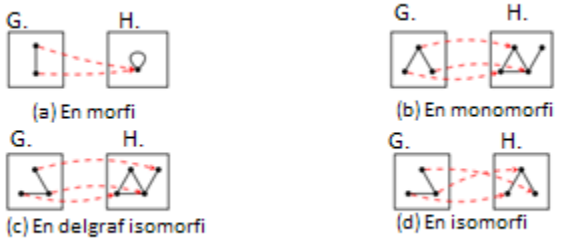
Grafer er relateret til hinanden gennem graf-morfier. Disse morfier mapper knuder og kanterne til hinanden mellem graferne, hvor man stadig bevarer strukturen af knuderne og kanterne.

Definition 2.2.1 (graf-morfi): Givent to grafer $G(V,E)$ og $H(V,E)$ er en graf homomorfi m fra G til H skrevet som: $m: G \to H$. Det betyder, at hvis $e = (u,v) \in E(G)$, så har vi $m(e) = (m(u), m(v)) \in E(H)$, for alle parrene af knuderne v, u i $V(G)$.

Det vil sige, at m er en funktion fra $V(G)$ til $V(H)$, der mapper (u,v) fra hver kant i G til (u,v) til en kant i H.[10]

Definition 2.2.1 er en bestemt type af graf-morfi, som hedder homomorfi. Der er andre typer graf-morfier, som blandt andet:

- En morfi *m* er en *monomorfi*, hvis m er en injektiv homomorfi. Det vil sige, at for alle knuderne i G mappes til maksimalt et element i H og $m(u) \neq m(v)$
- En morfi *m* er en *delgraf isomorfi*, hvis m er en monomorfi, samt endepunkterne for kanterne i G korresponderer til mappingen af endepunkterne i H.
- En morfi *m* er en *isomorfi*, hvis m er en *delgraf isomorfi*, samt er en bijektion af knuderne. Det vil sige, at hvis vi har mappingen $m: G \to H$, sådan at (u,v) er to sammenhængende knuder i G, hvis og kun hvis $m(u)$ og $m(v)$ er sammenhængende i H.



Figur 2: Eksempler på de omtalte morfier i definition 2.2.1. (a) En morfi, som ikke er en monomorfi (b) en monomorfi, som ikke er en delgraf isomorfi, (c) en delgraf isomorfi, som ikke er en isomorfi, (d) en isomorfi. Figur fundet i [Andersen 2016, Thesis]

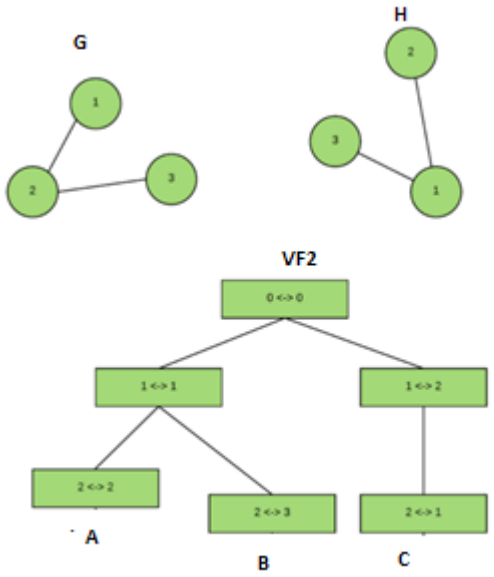
Det er imidlertid muligt, at der er flere morfier mellem objekterne, f.eks. hvis en graf G bliver matchet som en delgraf af H på forskellige måder, hvilket korresponderer til flere morfier $m: G \to H$

[10] [Ehrig et al. 2006, side 22]

2.3 Morfi testning

For at lave selve graftransformationen, er det som sagt essentielt først at finde ud af, om graferne deler information med hinanden, altså at se om der er en morfi mellem to grafer, svarende til to molekyler. Nu vi har defineret hvad en morfi er, skal vi selvfølgelig også vide, hvordan vi finder morfierne. Den algoritme, som vi gør brug af for at finde morfierne, er VF2-algoritmen. Grunden til, at vi bruger den algoritme, er pga. kvaliteten samt tilgængeligheden af den.[11] VF2-algoritmen er overordnet set en slags 'search tree algorithm', der forsøger at matche elementerne i graferne, altså den søger efter morfier. Den gør det ved først at starte med at matche de tomme grafer med hinanden og herefter gradvist udvide sig til den fulde morfi.

Et kort eksempel på hvordan VF2-algoritmen fungerer kan ses i figur 3.



Figur 3: VF2-algoritme der går igennem graferne G og H. Den går først igennem, hvor den tjekker knuder 1 i begge grafer. Disse bliver begge accepteret, samt gør knuderne 2 også, da der er en kant mellem 1 og 2 i begge. Der er dog ikke nogen kant mellem 3 og 2 i H, og derfor går den et skridt tilbage. Den går tilbage, indtil alle knuder passer, og vi får en morfi, der mapper 1 til 2, 2 til 1 og 3 til 3. [Strukturen af figuren er fundet på Stackoverflow, men vi modificeret den].

[11] [Cordella et al. 2001, Cordella et al. 2004]

Kapitel 3: Graftransformation og Double Pushout

En graftransformation er baseret på, at man har en graf og tillægger den graf en regel. Hver enkelt regel man anvender på en graf, fører videre til et graftransformationsskridt. Man kan tolke en graftransformation som en regel-baseret modifikation af en graf.

Der er flere forskellige tilgange til at lave en graftransformation. Der findes blandt andet de tre algebraiske tilgange: *Single Pushout(SPO), Sesqui Pushout(SQPO)* og *Double Pushout(DPO)*[12]. Det er værd at notere sig at der er en betydelig forskel mellem de tre algebraiske tilgange. Vi vil ikke gå komme yderligere ind på SQPO, men vil senere komme ind på forskellen mellem SPO og DPO.

I vores program og overordnede emne fokuserer vi på DPO og med fokus på kemisk modellering. Dette kommer også til udtryk i de kommende afsnit. Det er værd at notere sig, at når man taler om kemisk modelleringen, bliver grafen også tolket i et kemisk perspektiv. Knuder og kanter, er ikke længere bare knuder og kanter. Knuderne repræsenterer molekylerne og kanterne er de kemiske bindinger mellem disse molekyler, som vi har været inde på i kapitel 2.

DPO er det vi kalder en algebraisk graftransformation. Algebraiske tilgange til graftransformationer har haft en indflydelse for graftransformationens udvikling, og er i dag stadig relevant.[13] Vi vil kort komme ind på hvad et pushout er, generelt set, for derefter at bevæge os imod DPO-metoden.

3.1 Pushout

I kategoriteori bruger man kommutative diagrammer ved mange illustrationer, hvilket der også bliver gjort i denne rapport. Et kommutativt diagram er en orienteret graf, hvor knuder er objekter og kanter er morfier.

For at forstå hvornår et diagram er kommutativt, er det vigtigt at introducere en ny definition. For at et diagram kan kommutere er det vigtige at vide hvordan vi kombinerer morfier. Vi kombinerer morfier ved kompositionen af morfier langs stierne som ender med at blive den samme morfi.

Komposition af graf-morfier definition 3.1[14]: Givet to graf morfier $f = (f_V, f_E): G_1 \to G_2$ *og* $g = (g_V, g_E): G_2 \to G_3$, *kompositionen af* $g \circ f = (g_V \circ f_V, g_E \circ f_E): G_1 \to G_3$ er igen en graf morfi.

Et bevis for definitionen kan findes i samme afsnit.

[12] [Gritter 2018, Intro i artiklen]

[13] [Ehrig et al. 2006, Preface]

[14] [Ehrig et al. 2006, s.22, Definition 2.4]

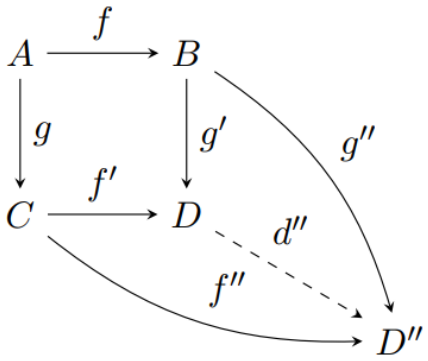
Et diagram siges at kommutere når kompositionen af *morfier* langs forskellige stier, med den samme start- og endepunkter, ender ud i den samme morfi. Hvilket figur 5 og 6 er eksempler på.

Pushout definition 3.2: Givet to morfier: $f: A \to B$ og $g: A \to C$ i en kategori, danner morfierne $f': C \to D$ og $g': B \to D$ et pushout af f og g, hvis og kun hvis:[15]

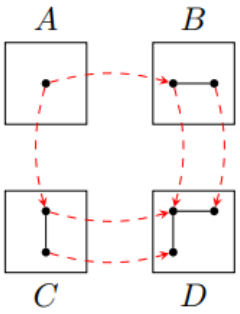
i) $g' \circ f = f' \circ g$, at det kommutative diagram er kommutativt.

ii) For alle par af morfier gælder $d'': D \to D''$ med $g'' \circ f = f'' \circ g$, der eksisterer en unik morfi $d'': D \to D''$ således at $f'' = d'' \circ f'$ og $g'' = d'' \circ g'$

Disse to regler for at danne vores pushout objekt fortæller at i) morfierne skal være kommutative og ii) for all morfier f'', g'', der er kommutative med f, g, at der eksisterer en unik morfi d'', der er kommutativ med de andre morfier. Hvis dette er opfyldt, er det muligt at danne et pushout objekt D.



Figur 5: Illustration af definitionen af et Pushout(Definition 3.2). Parret af morfier f' og g' er et Pushout af f og g hvis i). Figuren er fra [Andersen 2016, Thesis, figur 6.1, s.54]



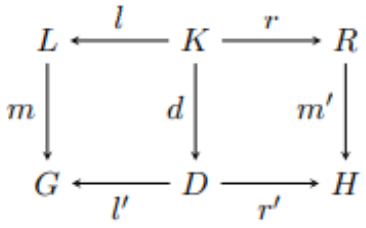
Figur 4: Viser et pushout af en simple pushout kandidat. Denne figur viser et pushout af A og B der bliver sat sammen langs A, hvor vi ender med D som vores pushout objekt. Figuren er fra [Andersen 2016, Thesis, figur 6.2(a) s.55]

[15] [Ehrig et al. 2006, Definition A.17]

3.2 Double Pushout

I DPO har vi en input-graf G. Vi har vores venstreside, L, som repræsenterer vores *pre-condition* af reglen, K, som er gluing-/ kontekstgraf. R repræsenterer vores *post-condition* af reglen. Gluing-grafen beskriver en del af grafen som skal eksistere for at vi kan bruge reglen. K binder $L\ og\ R$ sammen, da K er en delgraf af L, R og D. Regler fortæller om en proces der kommer til at ske med grafen. $L\backslash K$ er en notation, der beskriver om de knuder og kanter der er i L, men som ikke i K, senere skal fjernes fra G. $R\backslash K$ beskriver de knuder og kanter som er i R, men ikke i K, og de skal tilføjes til D. Det betyder at L som vores pre-condition, fortæller hvad der skal fjernes, og R som vores post-condition fortæller om hvad der skal tilføjes. Vi bruger $L\backslash K$ til at fjerne elementer fra G, og resultatet er den resterende struktur, der repræsenterer D. Den resterende struktur skal stadig være en lovlig graf, hvilket vil sige at vores match m, ikke må have nogen dangling edges. Det betyder morfien m skal opfylde, det man kalder en *gluing condition.* En gluing condition sørger for at 'gluingen' af $L\backslash K$ og D er det samme som G(se figur 6). (Man opnår gluing condition, når både dangling condition og identification condition er opfyldt, som vi snakker om i definition 3.3 og 3.4[16]) Vi har D, og skal nu have gluingen af $R\backslash K$ på D for at få vores udledte graf H. En graf-transformation udføres ved gluing konstruktioner, som er defineret ved $G = L+_k D$, og $H = R+_k D$[17]. Mere præcist så bruger vi graf-morfierne $l, r\ og\ d$, til at beskrive hvordan K er inkluderet i L, R og D[18]. Morfierne gør at vi kan definere vores gluing konstruktioner $G = L+_k D$, og $H = R+_k D$ som vores Pushout objekter $(PO1)$ og $(PO2)$ i figur 7, som er et DPO. Den resulterende morfi m' kaldes *co-match* af graf-transformationen $G \overset{p,m}{\Longrightarrow} H$. Processen kan beskrives at for at lave en graf-transformationen på vores graf, G, gør vi brug af reglen $p = (L \leftarrow K \rightarrow R)$. Det foregår således:

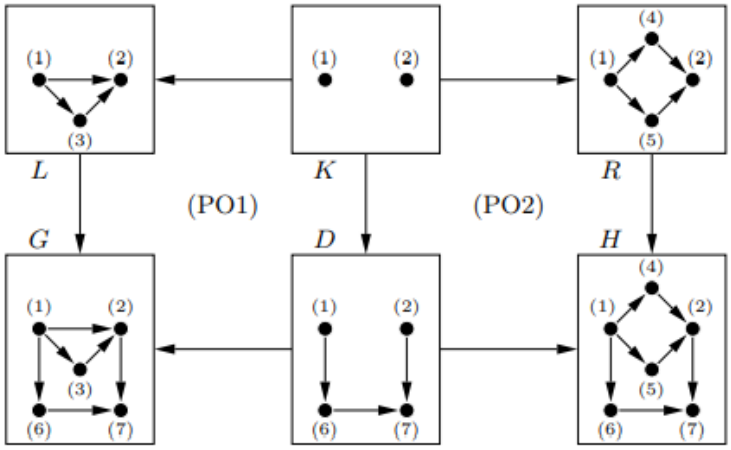
1. Find en match-morfi, m: $L \rightarrow G$, hvis den eksisterer
2. Konstruer D som et pushout af morfierne l, m, hvis D eksisterer
3. Konstruer H som det sidste pushout objekt fra d og r, hvis H eksisterer.



Figur 6: Et diagram af en udledning af $G \overset{p,m}{\Longrightarrow} H$ som Double Pushout, af G til H, ved brug af reglen p og morfien m. Figuren er taget fra [Andersen 2016 figur 6.6 s.58]

[16] [Ehrig et al. 2006, side 44,45,46,47]
[17] [Ehrig et al. 2006, side 11,12,13]
[18] [Ehrig et al. 2006, side 11,12,13]

Figur 6, er et double pushout diagram, som er kaldt en udledning af H fra G ved brug af transformationsreglen, p, og den matchende morfi, m. Det kan annoteres: $G \overset{p,m}{\Longrightarrow} H$
Efter match-morfien, m, er fundet, kan det beregnes om D eksisterer ved hjælp af den omtalte gluing condition, som vi nævnte tidligere.



Figur 7: Eksempel på en DPO-graf transformation. [Ehrig et al. Fig.1.4 s.13]

Definition 3.3 - Dangling condition:
Der er ingen kanter i $E_G \backslash m(E_L)$ tilstødende til en knude i $m\big(V_L \backslash l(V_K)\big)$. Det vil altså sige, hvis en regel p siger en knude skal slettes, kan dette kun lade sig gøre hvis den specificerer at få slettet alle kanter i forbindelse med den knude. Hvilket vil sige, at der må ikke eksistere nogle kanter, hvor der ikke er knuder i begge ender.

Definition 3.4 - Identification condition:
Der er ingen par af knuder $u, v \in V_L$ med $m(u) = m(v)$, $u \in l(V_K), u \in l(V_K), v \notin \mathrm{l}(\mathrm{V_K})$. Dette gælder hvis p specificerer sletningen af en knude eller kant, men bevarelsen af en anden knude eller kant, så må match-morfien, m, ikke identificerer disse knuder eller kanter med hinanden.

Identification points er de knuder og kanter der er identificeret af m. Dangling points er de knuder i L som afbilleder m er en kant i G, som ikke tilhører m(L). Gluing points er de knuder og kanter i L som ikke er slettet af p[19]. Gluing condition er opfyldt for p og m hvis IP $\cup$ DP $\subseteq$ GP. Hvor IP er identification points, DP er dangling points og GP er gluing points. Det betyder at hvis alle identification points

[19] [Ehrig et al. 2006, side 44,45,46,47]

og dangling points er gluing points, er gluing condition opfyldt. Det vil sige at når gluing condition er opfyldt eksisterer D, og når D eksisterer, så er den unik op til isomorfi. [20] I figur 7, hvis vi sletter knude (2) fra kontekstgrafen K så vil gluing condition ikke længere være opfyldt. Gluing condition er ikke opfyldt, da den slettede knude (2) vil føre til en dangling edge.

DPO og SPO adskiller sig specielt et sted, hvilket omfatter sletningen af elementer under et graftransformationskridt[21]. Hvis vores match $m : L \to G$ ikke tilfredsstiller vores gluing condition med henseende på vores produktion $p = (L \leftarrow K \to R)$, så kan vi ikke anvende produktionen p i DPO. Den kan derimod bruges i SPO. Vi går ikke mere i dybden med SPO, eller forskellene mellem dem i denne rapport, men der kan læses mere om SPO her[22].

Til modellering af kemi er det smart at kunne invertere en kemisk proces, da reaktioner af molekyler generelt er invertible[23]. Vi kan definere dette ud fra viden omkring kommutative diagrammer, hvor der er en form for symmetri i en transformations-regel. Der er ingen betydelig forskel mellem venstre og højresiderne, og uanset hvilken sti vi tager i det kommutative diagram, ender vi med det samme morfi.

Ud fra det kan vi finde den inverse udledning: $H \overset{p^{-1},m'}{\Longrightarrow} G$, ved at vi bruger den inverse transformationsregel $p^{-1} = (R \overset{r}{\leftarrow} K \overset{l}{\rightarrow} L)$ og co-matchet m'. Det er kun muligt at finde den inverse forudsat $l, r \; og \; m$ er monomorfier[24].
Det er bestemt praktisk for vores kemiske modellering, da vi skal arbejde med fire regler, hvor to af dem er de inverse. Dette komme vi nærmere ind på i afsnit 5.1.1.

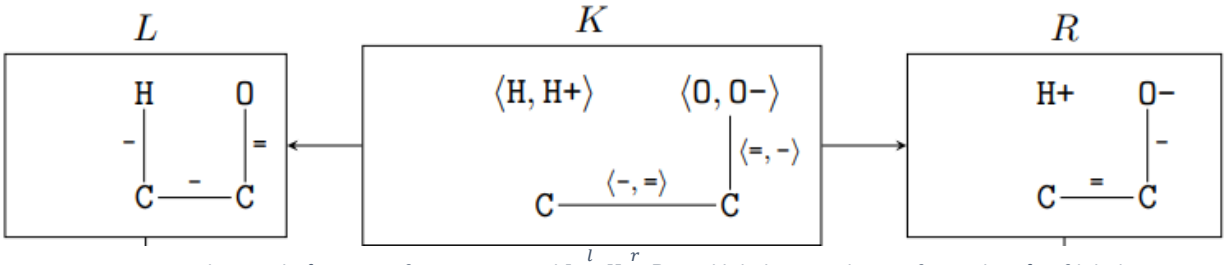
3.3 Grafer med labels

Nu vi har beskrevet hele processen bag graftransformation, graf-morfier, DPO osv., er det imidlertid vigtigt at forstå, at vi, som sagt, gerne vil arbejde os henad mod at transformere molekyler. Dette betyder, at trods førnævnte processer og definitioner virker for ikke-orienterede grafer, har vi brug for noget ekstra data for at arbejde med molekyler. Dette er grunden til, at vi nævner labels i definition 2.1.1. Det er til hvorpå, at vi vil skelne imellem de forskellige molekyler. Dvs., vi mærker hver kant og knude med noget ekstra data, for at f.eks. beskrive diverse atomer. I takt med, at der dog er en begrænset definition af labels inden for grafer med fokus på kemi, vælger vi at definere grafer med labels tæt ud fra, hvordan det er gjort i [Andersen 2016][25]. Vi definerede grafen som $G(V, E)$, men med labels, beskriver vi grafen som en tupel $G(V, E, a_V, a_E)$. Hvor a_V, a_E er funktioner der mærker hhv. knuder og kanter ud fra nogle mængder A_V og A_E. Disse mængder svarer til vores label-

[20] [Ehrig et al. 2006, side 44,45,46,47]
[21] [Ehrig et al. 2006 side 13,14]
[22] [M. Löwe.1993]
[23] [Andersen 2016, Thesis, side 58]
[24] [Annegret Habel 2001]
[25] [Andersen 2016, Thesis, side 18]

alfabet $A = (A_V, A_E)$, hvor A_V består af en mængde af knude-labels og A_E af en mængde af kant-labels. Det betyder at grafer med labels består af en graf, $G(V, E)$, samt disse labelfunktioner. Dette indebærer yderligere, at vi har et ekstra tjek, hvis vi gerne vil finde morfier imellem to grafer. Nu skal vi ikke blot kigge på knuderne og kanterne, men også de underliggende labels. En graf-morfi med labels $m: G \rightarrow H$ er altså en morfi mellem de to grafer, samt de er kompatible med vores labelfunktioner. I forhold til vores label-alfabet, fokuserer vi primært på labels som strenge, som vi kommer mere ind på i kapitel 4 og 5. Dette betyder, at for at vi skal være i stand til at finde morfier imellem graferne, kræver det at alle associerede labels er strenge, når vi prøver at finde morfier.

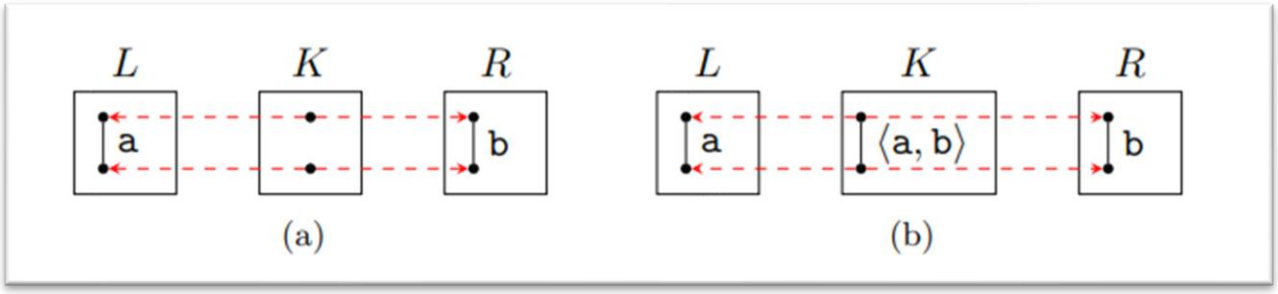
I forhold til selve graftransformationen er der også ændringer, når vi arbejder med labels. I dette kapitel har vi lært, hvordan en transformationsregel specificerer hvordan grafen ændres strukturelt. Forudsat, at vi vil transformere med labels derimod, tillader vi også muligheden for ændring af labels under denne transformation. Vi nævnte før, at A var vores alfabet for vores labels. Vi kan derfor definere en transformationsregel $\left(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\right)$ med labels således, at L og R er grafer med labels over A, hvorimod K er graf med labels over alfabetet $A \times A$. Morfierne l og r returnerer hhv. første og anden komponent i labelfunktionen: $A \times A \rightarrow A$. Det vil sige, at vores K består af alle disse par af labels, hvor den definerer om dens label skal ændres eller ej. I figur 8 kan vi se, hvordan dette er sat op i forhold til transformationsreglerne. K består som sagt af både af $A \times A$, hvor L og K består af hhv. første element af disse par.



Figur 8: Et eksempel på en transformationsregel $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ med labels. Kontekst-grafen, K, består af labels som et par af to strenge, hvorimod L og R blot består af strenge. Figur fundet i [Andersen 2016, Thesis]

I figur 9, kan vi se, hvordan de i [Andersen 2016][26] ikke tillader (a) men (b), da K i (a) ikke specificerer tilladelsen af at ændre labels, som den gør i (b). Vi tillader derimod dog denne repræsentation af labels, da, som vi kommer ind på i kapitel 5, kun gør brug af transformationsregler, hvor første og andet komponent af et par $A \times A$, består af det samme.

[26] [Andersen 2016, Thesis, side 18]



Figur 9: I reglen (a) ser vi hvordan K specificerer fjernelsen af en kant, men R tilføjer samme kant med en ny label. I reglen (b) specificerer kontekst-grafen, K, at denne kant forbliver, men den skifter label fra a til b. Figur fundet i [Andersen 2016, Thesis]

I denne sektion beskriver vi labels på kanterne, som om de er implementeret. Det er på baggrund af, at dette afsnit er rent teknisk, og vi på senere sigt gerne vil tilføje labels til vores kanter. Figur 9 er altså hypotetisk set.

Kapitel 4: Design og implementering

For at løse problemet, som vi har gennemgået gennem hele denne rapport med at graftransformere, hvor vi har bevæget os over i en retning, hvor vi vil graf-transformere molekyler, har vi lavet en prototype for, hvordan denne graftransformation kunne fungere. For at implementere dette, har vi opdelt vores program i to filer, hhv. en regel-fil og en main-fil. Vi har desuden brugt hjælp fra MØD[27], som er en software pakke, der netop er udviklet til graf-baseret teknologi inden for kemi. MØD-pakken er designet til netop denne problematik, og indeholder derfor også et helt generelt graf-transformationssystem, der er i stand til at generere kemiske reaktionsnetværk.[28] Vi bruger dog imidlertid MØD til hjælp af generering af de forskellige kemiske molekyler, samt regler. Det sidste vi benytter os af, er til vores repræsentation af de forskellige grafer for molekylerne. Her benytter vi os af Python pakken, NetworkX, der bliver brugt til manipulation og strukturering af komplekse netværkssystemer.[29]

4.1 Design

I det overordnede design af vores program, har vi, som sagt, opdelt det i to filer. Dette er blandt andet for at opnå konceptet bag *low coupling*, men samtidig opretholde *high cohesion*. Low coupling, betyder kort sagt, at vores filer skal være i stand til at blive ændret i, uden at det vil ødelægge de andre filer. Dette er en af grundene til, at vi har opdelt det i to filer, så vi er i stand til at ændre i vores regel-fil, uden at det vil have et stort indtryk på den anden fil. Når det så er sagt, er begge filer naturligvis essentielle for, at vores program fungerer, og koden er direkte

[27] [Andersen 2016]
[28] [Andersen 2016]
[29] [Aric Hagberg 2005]

relateret til hinanden i forhold til funktionaliteten af programmet, hvilket gør vores kode nemt at opretholde og kan i sidste ende hjælpe os med at tilføje nye features til programmet i fremtiden. Dette er konceptet bag high cohesion.

Det er dog imidlertid vigtigt at huske på, at idéen bag graftransformation er baseret på disse regel-baserede modifikationer af grafer.[30]I forhold til valg af datastruktur til disse regler, har vi først og fremmest valgt at lave en klasse til at gemme dem. Det er det, som hoveddelen af vores regel-fil består af, så vi nemt kan gemme og tilgå de forskellige regler. Selve graferne som reglerne indeholder dog, kræver naturligvis også en datastruktur i sig selv. Det er her, vi netop gør brug af NetworkX, som vi kom ind på tidligere. NetworkX gør det nemt for os at manipulere grafer uden at skulle lave en separat graf-klasse. Udover graferne består reglerne også af mappingen mellem disse grafer. Dem har vi valgt at repræsentere som *dictionaries*, da MØD bruger dictionaries, når den genererer dem for os. Det giver dog også god mening, i den forstand, at en dictionary består af en nøgle og en værdi, som er svarende til knude-ID'erne mellem de to grafer.

Vi har desuden valgt at tilføje en visualiseringsfunktion af graferne i begge filer, for nemt at kunne teste og modificere i den ene fil, uden at det vil ødelægge den anden, igen for opretholdelsen af low coupling. Vi har valgt at benytte os af Pythons 'matplotlib' bibliotek, da det rent grafisk set er hurtigt, nemt og tilgængeligt for visualisering af objekter[31]. Vi benytter os af denne visualiseringsteknik for både vores dictionaries og graferne.

Udover vores regel-fil, har vi naturligvis, som sagt, vores main-fil. Det er her selve graftransformationen sker. Inden vi kommer ind på selve graftransformationen dog, definerer vi først nogle små funktioner, som alt i alt, gør programmet nemmere at bruge og giver det bedre funktionalitet. Som vi nævnte, består denne fil også af en visualiseringsfunktion: 'my_show', som bliver brugt til visualisering af graferne, ikke blot før de er blevet transformeret, men også under selve transformationen og efter transformationen er færdig. Filen består desuden af funktionerne: 'my_combine' og 'my_inv'. Combine-funktionen bliver brugt til kombinering af forskellige grafer/molekyler, hvor 'my_inv' er en funktion, som bliver brugt til at lave et inverst map. Dette kan give mening i forhold til, hvis vi ikke blot vil scanne efter knuder den ene vej mellem to grafer, men også den anden. Det sidste der sker i denne fil, er selve graftransformationen, hvor vi benytter os af en NetworkX's funktioner til at finde morfier mellem to grafer, samt for hver af de matches vi finder med denne funktion, laver selve transformationen.

4.2 Implementering

I vores overordnede design, har vi som sagt gjort både brug af Pythons 'matplotlibs' bibliotek til visualisering, samt MØD til genereringen af grafer og regler, hvilket alt sammen er essentielt for at vores program fungerer. Noget af det vigtigste vi dog

[30] [Ehrig et al. 2006, side 6]
[31] [John D. Hunter 2003]

benytter os af, er NetworkX, som er fundamental for at vores program fungerer, da det er det vi bruger som datastruktur, og uden dette, ville vi slet ikke kunne generere grafer. Først og fremmest, vil vi dog gå lidt mere i dybden med MØD, og hvordan vi har brugt hjælp fra MØD til vores implementering af molekyler.

4.2.1 MØD

Man kan dog undre sig over, hvorfor vi har valgt netop MØD til at generere molekyler. For at forstå dette, er det dog imidlertid vigtigt at forstå formatet SMILES. SMILES er nemlig en specifikation i form af en notation til at beskrive strukturen af kemiske molekyler ved at bruge ASCII-strenge, som stammer helt tilbage fra 1980'erne[32]. Eksempler på SMILES-formater kan ses i figur 10. Det er ikke ligetil at importere molekyler som grafer, men hvis man snakker om SMILES i forhold til graf-baseret procedurer, så er en SMILES streng opnået ved at printe knuderne man møder i en *depth-first tree traversal* af en kemisk graf. I vores program er vi, som sagt, gået henad i mod at være i stand til at indlæse molekyler og først og fremmest visualisere dem som grafer. Det er her MØD viser sig at være nyttigt, da MØD er i stand til at indlæse de fleste SMILE-strenge og konvertere dem indtil grafer med labels[33]. Selvom aflæsningen af SMILES, er baseret på OpenSMILES specifikationen, som nok er den meste udbredte SMILES-format[34], er der dog få ændringer, som I kan læse om på MØDs dokumentation af dataformater[35].

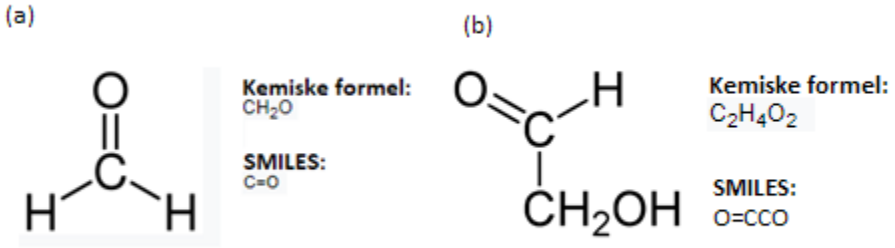
Måden vi har valgt at bruge MØD på, er dog ikke direkte at implementere deres funktion i forhold til at læse SMILES. I stedet udnytter vi MØDs *Live Playground*[36] til at lave et script, der indlæser et molekyle vha. SMILES og udskriver dens knuder og kanter i et sådant Python format, at vi blot skal indskrive dette i vores eget program. Et eksempel på hvordan et sådant script ser ud, kan ses i bilag 1.[37] Her kan det ses, hvordan vi bruger deres *smiles* funktion til at indlæse molekylet, hvor vi herefter indskriver hvordan terminalen skal printe knuderne og kanterne ud i et sådant format, at vi direkte kan kopiere udfaldet i terminalen til vores kode. På den måde kan vi direkte tilføje de forskellige knuder, kanter og labels som det molekyle består af til en graf. Dette har vi ligeledes gjort ved implementeringen af regler, som også kan ses i bilag 1[38]. Her har vi, i stedet for at konstruere et molekyle vha. SMILE-strenge, kaldt på en allerede givet regel, og på samme måde printes der i terminalen knuderne og kanterne for alle delene af reglerne, dvs. både L, K og R, samt også mappingen mellem dem. Vi kan herfra implementere det i vores program, og på samme måde har vi altså nu en regel. Reglen, som vi ser på bilaget, er keto-enol[39], som allerede i MØD er givet ved kode, som er meget lignende:

[32] [Eyben et al. 2008]
[33] [Andersen 2016, 8.6. SMILES]
[34] [Ehrig et al. 2006, side 37]
[35] [Andersen 2016, 8.6. SMILES]
[36] [Andersen 2016]
[37] Bilag 1
[38] Bilag 1
[39] Figur 13

'p = keteEnolGML("keteEnolFile.GML")', hvor i dette tilfælde, keteEnolFile.GML altså er en fil med specifikationerne af transformationsreglen. Der er således tydeligt at se, at vha. MØD og vores scripts vi har lavet via deres *playground*, har vi en hurtig, enkel og tilgængelig måde at indlæse molekyler og regler på.

Det kan dog virke underligt at vælge at bruge en ekstern hjemmeside til at indlæse disse ting på. OpenSMILES-specifikationen er som sagt velkendt, og ifølge OpenSMILES egen hjemmeside[40], er der allerede biblioteker i de fleste programmeringssprog, også Python, som allerede er i stand til at generere molekyler vha. SMILES. Disse biblioteker, som f.eks. *pysmiles*[41], virker dog imidlertid usikre, og der står sågar på specifikationen af pysmiles: *"should be usable"*. Af denne årsag har vi valgt at gå med MØD, og da MØD endda har specifikationer af regler og nem indlæsning af regler også, var det logisk at vælge.

Det sidste der er et bide mærke i, i forhold til de scripts, som vi har brugt fra MØD, er at begge scripts gør brug af 'nx.Graph()'. Dette er naturligvis en reference til NetworkX, der, som vi tidligere har oplyst, er væsentligt for vores program.



Figur 10: Eksempel hvor vi i (a) ser molekylen, formaldehyde, med dens kemiske formel, graf visualisering, samt SMILES specifikationen af molekylen, og i (b) ser det samme for glykolaldehyd-molekylet.

4.2.2 NetworkX

Vores to scripts vi benytter os af, bliver brugt i hhv. vores regel-fil og vores main-fil, og bruge NetworkX som datastruktur til graferne. De scripts vi nævnte i forrige kapitel, gør hver især brug af hver sit script, men ud fra udkommet af scriptsene, er det tydeligt at se, at de ligner hinanden. Dette er fordi, at de molekyler vi indlæser, bliver repræsenteret som en NetworkX graf, hvilket er det f.eks. 'L = nx.Graph()', betyder. Denne sætning betyder, kort sagt, at vi nu har lavet en graf, vha. NetworkX, som hedder *L*. Den består hverken af knuder eller kanter endnu, men

[40] [Eyben et al. 2008]
[41] [Kroon 2020]

dette er meget ligetil i NetworkX, da vi blot skal skrive hhv. 'add_edge' og 'add_node' for at tilføje knuder og kanter til graferne, som forklarer udkommet af resten af scriptsene. Det vil sige, at hvis vi laver f.eks. formaldehyde-molekylet, kan vi nemt tilføje knuderne, ergo lave en graf med fire knuder. (Se kapitel 5 for, hvorfor vi har valgt blandt andet at benytte os af formaldehyd-molekylet, samt de omtalte regler, vi kommer ind på i dette kapitel). Det er endda indbygget i NetworkX, så vi ikke blot kan tilføje knuderne men også de forskellige *labels* til de forskellige knuder, som kan ses i figur 14 på bilag 1, samt visualiseringen af dette i bilag 2[42]. Labels inden for kemi er, som sagt, ret essentielt, da de bliver brugt til at beskrive de forskellige atomer og bindinger mellem atomerne.

NetworkX kommer dog ikke kun med muligheden for at lave en graf samt knuderne og kanterne til den graf. Der er nemlig indbygget en del funktioner i NetworkX, som spiller en stor relevans for vores program, hvilket vi vil gå i dybden med lidt senere. NetworkX gør det blandt andet muligt at scanne igennem elementer i sin graf; muliggørelsen af at man kan visualisere sin grafer; eller helt at direkte videregive alle ens informationer i den ene graf til den anden graf.[43] Alt dette og mere benytter vi os af i vores program, hvilket gør NetworkX til en utroligt vigtig komponent i vores program. Af disse åbenlyse grunde og mere til er årsagen til, at vi benytter os af NetworkX. Vi nævnte tidligere i kapitel 2.3, at vi gør brug af VF2-algoritmen for at finde morfier mellem grafer og hvordan den fungerer. Dette er yderligere muligt med NetworkX. Alt dette, samt mere, vil vi gå i dybde med i næste afsnit, hvor vi graver os dybere ned i implementeringen af koden, og hvordan vi benytter os af alle disse funktioner fra blandt andet NetworkX.

4.2.3 rule.py

I rule.py filen, som er vores regel-fil, benytter vi os først og fremmest af disse scripts, som vi snakkede om i de forrige kapitler. Vi har valgt at importere fire regler, som hhv. er *ketonolF(keto-enol)*[44]*, aldoladdF (aldol addition)* samt de inverse af begge de regler. Vi nævnte som sagt under Design-kapitlet, at vi har valgt at gemme disse regler under denne *Rule* klasse. Det vil sige, at når vi initialiserer en ny regel, altså et Rule objekt, så skal vi give alle aspekterne af reglen, hvilket er grunden til, at vores *__init__* funktion ser ud som den gør. Til slut, når vi har initialiseret alle vores regler, gemmer vi dem i en liste af regler, så de nemt kan blive tilgået, og så brugeren kan få et overblik over alle tilgængelige regler. Udover selve Rule-klassen samt initialiseringen af de molekylære regler, består rule.py også af to funktioner. Vi nævnte kort under kapitel 3.1, at en af funktionerne vi benytter os af, er en visualiseringsfunktion. Det er dog faktisk begge funktioner i rule.py, som er visualiseringsfunktioner. Én til visualisering af grafer, den anden til visualisering af vores mappings. De to funktioner hedder hhv. 'my_show' og -'show_map'. Funktionen 'my_show' tager en graf som argument, og returnerer visualiseringen af den graf, hvorimod 'show_map' tager en dictionary som

[42] Bilag 1, figur 18
[43][Hagberg 2005, tutorial]
[44] Figur 13

argument og returnerer visualiseringen af den dictionary, svarende til vores mappings imellem graferne. Allerede i vores 'my_show' funktion, kan vi se benyttelsen af indbyggede NetworkX funktioner. Vi bruger først og fremmest 'get_node_attributes'[45], da vi gerne vil have vores labels fra vores grafer visualiseret også, da målet, som sagt, er at kunne fremvise molekyler. Derudover benytter vi os af funktionen 'draw'[46], som er en simpel NetworkX funktion, der tillader os at tegne og visualisere grafen med *matplotlib*, hvilket er grunden til, at vi bruger *matplotlibs* 'show' funktion til sidst. I vores 'show_map' funktion, benytter vi os ikke af nogle NetworkX funktioner, af den simple årsag, at de i ikke har noget med NetworkX at gøre. Vi har dog alligevel fundet det relevant at være i stand til at kunne visualisere vores mappings, og besluttede derfor at implementere en funktion, der var i stand til det.

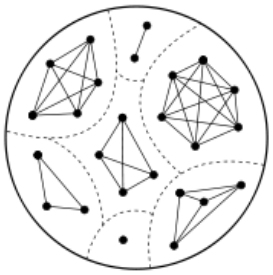
Funktionen tager, som sagt, et map som input og konverterer det map til en liste af elementerne i mappingen. Den bruger herefter *zip* til at opdele elementerne i hhv. en y og x-akse. Til sidst fremstiller funktionen disse elementer grafisk, så vi får en graf, der viser hhv. nøgle-elementerne som værende x-aksen og værdi-elementerne som værende y-aksen. Vi kan se på bilag 2[47], hvordan visualiseringen af mappingen ser ud, og hvordan vi nemt kan afbilde hvordan nøgle- og værdi-elementerne hænger sammen.

4.2.4 main.py

Det er i vores main.py fil at selve transformationen sker. Med rule.py, kan vi initialisere de forskellige regler, og ved hjælp af den fil, samt vores script for at implementere de forskellige molekylære grafer, er det nu muligt at lave selve transformationen. Udover selve oprettelsen af graferne, består vores main-fil først og fremmest af tre funktioner. Den første nævnte vi under Design-kapitlet, hvilket er 'my_show' funktionen. Den fungerer på samme måde som i rule.py. Den næste funktion i main.py er 'my_combine'. Det er en funktion, der tager en liste af grafer som input og returnerer den kombinerede graf. Et eksempel på en sådan graf kan ses i figur 18 på bilag 2.[48] Vi benytter os her af NetworkX's funktion 'disjoint_union_all'. Det er en funktion, som tager en liste af grafer som input, og altså returnerer den disjunkte fællesmængde af disse grafer. Det vil sige, at hvis vi kigger på graferne som mængder, så kombinerer den alle de forskellige parvise elementer i sættene. I forhold til grafer, så er det parallelt til, når vi kigger på det som mængder. For at give en analogi til, hvordan man finder den disjunkte foreningsmængde, kan man kan sige, at den kombinerede graf er konstrueret ved, at man tegner to grafer ned på et papir uden at forbinde dem med hinanden. Hvis vi kigger på denne graf som en enkelt graf, svarer dette til den disjunkte foreningsmængde. Figur 11 er eksempler på den disjunkte foreningsmængde ved hhv. grafer og mængder.

[45][Hagberg 2005, get_node_attributes sektion]
[46][Hagberg 2005, draw sektion]
[47] Bilag 2
[48] Bilag 2

Den sidste funktion før selve transformationen i main.py er ’my_inv’, hvilket er en funktion, der tager en dictionary som input og returnerer det inverse af den dictionary. Som vi vil se senere, har det relevans i forhold til at finde det inverse af de forskellige mappings mellem graferne.



$A^* = \{(1, 0), (2, 0), (3, 0), (4, 0), (5, 0)\}$
$B^* = \{(1, 1), (2, 1), (3, 1), (4, 1)\},$

$A \bigcup^* B = A^* \bigcup B^*$
$= \{(1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (1, 1), (2, 1), (3, 1), (4, 1)\}.$

Figur 11: Hhv. et eksempel på en graf, der er en disjunkt foreningsmængde af en masser grafer, samt et eksempel på den disjunkte foreningsmængde af to mængder A og B. (grafen og mængderne har intet med hinanden at gøre). Graferne er taget fra hhv. [Rinke 2008] og [Armstrong 1997].

Efter vores molekylære graf er konstrueret, samt vores transformationsregler er på plads, kan selve transformationen påbegyndes. Vi har valgt at kalde denne funktion ’transformation’. Denne funktion tager 2 argumenter; vores oprindelige graf, samt den regel vi vil bruge. (Den kan yderligere tage to argumenter, hvis man vil have verbose mode på eller vil visualisere graferne. De er som default sat til ’False’.)Vi nævnte i kapitel 2, at det først og fremmest var essentielt at finde morfierne af knuderne mellem den graf, som skal transformeres, samt den transformationsregel, der skal benyttes til transformationen. Det er netop det, som bliver gjort vha. NetworkX med sætningen: ’GM=nx.algorithms.isomorphism.GraphMatcher(graph,rule.left).subgraph_isomorphisms_iter()[49]’. NetworkX har netop implementeret VF2 algoritmen for morfi testning, og vha. deres bibliotek, kan vi nemt finde ud af, om to grafer er morfier. Med det første step i selve denne funktion, altså ’algorithms.isomorphism.GraphMatcher’ finder man først og fremmest ud af, om de to grafer er morfier. Vi kunne tilføje en ’is_isomorphic’ funktion efterfølgende, for at verificere, om de f.eks. er isomorfier. Det sidste step i funktionen er en simpel generator over alle de isomorfier, der er blevet matchet, som er fundet mellem en delmængde af G og L. I kapitel 2, beskriver vi hvordan det første step i transformationen er at finde delgrafer og morfierne mellem dem, hvilket er grunden til, at vi gør brug af ’subgraph_isomorphisms_iter()’ og ikke blot ’isomorphisms_iter()’. Efter morfierne mellem G og L er fundet, er det at selve

[49] [Hagberg 2005, subgraph_isomorphisms_iter sektion]

implementeringen af vores egen algoritme påbegyndes. Det er her selve transformationen sker, som den forklares, når vi snakker om Double Push Out i kapitel 3.

Efter vi har fundet samtlige morfier mellem G og L, går vi igennem et loop af alle de matches, som vi fandt, da det er for alle de matches, at vi vil lave transformationen. Som vi ved, når vi snakker om DPO, består den af flere forskellige komponenter. Disse komponenter er nogle vi allerede har, i form af de regler, som vi allerede har implementeret, samt vores graf, som vi giver som input. Det er dog også de grafer, som vi selv skal konstruere, altså H og D. Dette er grunden til, at vi initialiserer dem først. Vi initialiserer herefter også de inverse mappings af de forskellige relevante mappings, som vi kommer til at bruge senere i programmet.

Efter de forskellige initialiseringer påbegyndes konstruktionen af D. Den tilføjer de forskellige knuder, som er i G, og hvis den knude er i L, men ikke i K, så fjernes knuden. Det er vigtigt, at vi tjekker i *mapLK*, som svarer til mappingen af fra L til K, for ikke at lave rod med knuderne, samt huske at køre de knuder, som vi har tilføjet til D, igennem *mapM*, når vi igen fjerner den. Dette er vigtigt, da vi skal huske, at f.eks. knude 1 i L ikke nødvendigvis svarer til knude 1 i G, så det er essentielt, at vi fjerner og tilføjer de korrekte knuder og kanter. Dette er gældende resten af programmet også.

Herefter skal kanterne tilføjes. Den tilføjer også først og fremmest de forskellige kanter, som er i G i et for-loop. Inden vi fortsætter med at snakke om kanterne, er der dog noget terminologi, som er vigtigt at forstå. I takt med, at et map kun kigger på knuderne, så er det vigtigt at for, at vi får fat i de rigtige kanter, skal endepunkterne fra kanterne også køres igennem de relevante mappings. F.eks. hvis vi har en kant i L, eL, så svarer endepunkterne fra den kant til $eL = (vL, uL)$, og for så at tjekke om den *samme* kant findes i K, skal vi tjekke: $eK = (mapLK[vL], mapLK[uL])$. Dette er hovedpræmissen i vores loop for at tilføje kanter. Efter vi har tilføjet kanterne fra G til D vil vi finde kanterne i K. Vi finder først kanterne i L, hvor den kører igennem mappingen af G og L. Hvis sådan en kant findes, så fjerner vi den. Efter vi har fjernet kanten, tjekker vi om den samme kant er i K, ved netop at køre den igennem mapLK, som forklaret tidligere. I tilfældet, hvor eK eksisterer, tilføjer vi kanten igen. I de tilfælde, hvor vi ikke kan finde eK, printer vi en *error*. Grunden til, at den i første omgang er under en *'try block'*, er fordi vi prøver at tilgå knuder, som i nogle tilfælde ikke eksisterer. Det vil sige, når vi prøver at tilgå en nøgle i et af de her mappings, som ikke er i dem. Vi printer en *error* i alle tilfælde som denne.

Efterfølgende bliver der tjekket for *dangling edges*, som vi ved er en stor del af DPO. Måden vi har valgt at håndtere dem på, er først og fremmest at lave to *if not* statements. I det første statement, bliver der tjekket om L har en kant, som også er i G. Hvis ikke dette er tilfældet, men L har endepunkterne for de kanter, og disse

endepunkter bliver fjernet i K, har vi en *dangling edge* og vi stopper programmet. Transformationen kan ikke fuldføres. Grunden til, at vi får en *dangling edge* her, er fordi alle de kanter/knuder i G, som ikke er i L, skal tilføjes. Dog, hvis de tilhørende knuder til de kanter, som bliver fjernet i K, har vi en kant uden knuder, som ikke er en mulighed i DPO. I næste håndtering af *dangling edges,* kigger vi ikke på, om L har begge endepunkter, men blot én af dem og denne knude fjernes i K. De håndteres her på samme måde som før. Transformationen kan ikke fuldføres.

I vores valg af brug af NetworkX som datastruktur, er der dog også nogle ulemper. En af disse ulemper er, at NetworkX automatisk tilføjer en knude til de kanter, hvis knuder ikke allerede er i grafen[50]. Dette kan gå hen at skabe problemer i vores program. F.eks. i det tilfælde, hvor L har en kant, som ikke er i K, samt knuderne for denne kant ikke er i K. Her ville vores program tilføje disse knuder til D, hvilket er forkert. Vi håndterer denne problemstilling i den sidste del af konstruktionen af D, altså efter vi har tilføjet alle kanterne til D. Vi håndterer i denne del af programmet også et sidste tilfælde af *dangling edges.* Efter vi har tilføjet alle knuder og kanter til D, kan vi opnå et tilfælde, hvor K fjerner en knude, for en kant, som hører til i D. I dette tilfælde kan transformationen heller ikke fuldføres.

Efter konstruktionen af D er færdiggjort, er det som følge af DPO, H, som skal opbygges. Inden vi gør det, skal vi, af samme grund som ved opbygningen af D, have lavet mappingen mellem K og D, således, at vi får *mapKD*. Vi har alle informationer til at lave det, da vi både har K og D, og bruger hhv. *zip* og *dict* for at repræsentere denne mapping som en *dictionary,* som vi også gør i de andre tilfælde. H består først og fremmest af alle knuder i D, hvorfor de tilføjes først. Det er dog vigtigt, som det også fremstår af figur 12, at de knuder og kanter, som der er i R, de tilføjes, hvis ikke de er i K. Der kan opstå et problem ved dette scenarie, hvis navnene på de knuder i R er de samme, som de allerede er i D. For at komme udenom denne problemstilling, har vi valgt at lave endnu en *dictionary,* som vi kalder *mapRH*. Meningen med den, er at den skal mappe de knuder i R til knuderne i H, så de knuder der er i D, ikke overlapper med de knuder, som er i R. Det første vi gjorde, var som sagt, at tilføje alle knuderne fra D over i H. De knuder, der dog eksisterer side om side i R og D, altså de knuder i R, som også er i K, mappes først og fremmest. I tilfælde af figur 12, svarer dette til knude 1. Det er vigtigt, at når vi tjekker efter disse, at vi først tjekker i mappingen mellem K og R, og dernæst mappingen mellem K og D, så den ikke tror, at knude 1, 2 og 3 i R svarer til det samme i D, men kun for dem, som også er i K. Af samme årsag, som da vi tilføjede knuderne til D, kører vi dog også disse knuder i R, som svarer til dem i hhv. K, D, L og G, igennem *mapM*, for at være sikker på, at de knuder i R, svarer til de korrekte knuder i G.

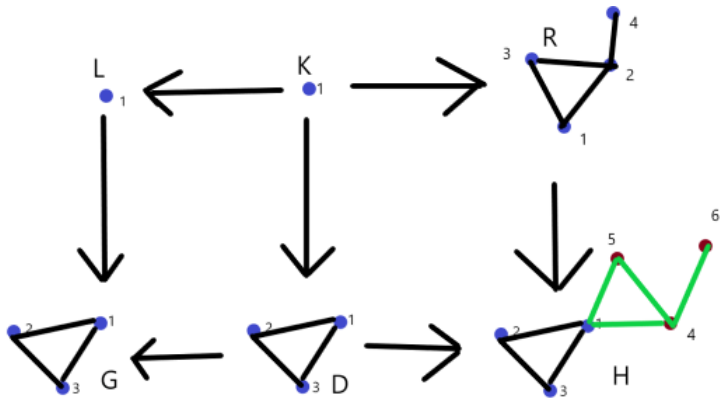
Efter, at vi har tilføjet knuderne fra D over i H og har styr på mappingen, skal de knuder i R, som ikke eksisterer i K, tilføjes. Her tjekker programmet først og fremmest om knuderne eksisterer i K. I tilfældet, hvor de gør der, gør programmet

[50] [Hagberg 2005, add_edge sektion]

intet, knuderne er allerede tilføjet. I det tilfælde, hvor de ikke gør det dog, skal vi have tilføjet knuderne. For dog at komme uden om problemet med, at der måske allerede eksisterer knuder i H med samme navn, tjekker vi i H, om de har det samme navn. Hvis dette er tilfældet, forøger vi knuden med én indtil, at den knude ikke eksisterer i H, og så tilføjes knuden. Denne knude skal også mappes i *mapRH*, så denne mapping nu består af alle de knuder i R uden, at den overlapper med navnene på knuderne i D. Knuderne for H er nu færdigkonstrueret.

At tilføje kanterne til H er lidt mere simpelt, end da vi tilføjede kanterne til D. Først og fremmest skal alle de kanter i D tilføjes, og hernæst skal alle kanterne i R tilføjes. Vi behøves ikke tænke på *dangling edges* i dette tilfælde, da der ikke er kanter der skal fjernes her, men blot tilføjes. Rent teknisk set, skal vi kun tilføje de kanter i R, som ikke er i K. I og med, at vi dog kører vores kanter igennem vores *mapRH*, før vi tilføjer dem, som vi lavede tidligere, kan vi blot tilføje alle de kanter der er i R, efter de er kørt igennem det map. I de tilfælde, hvor der er kanter i både R og K, eksisterer de kanter allerede, og selvom vi tilføjer kanterne, har det ingen effekt, som følge af NetworkX. Grunden til, at vi har en del *try-blocks* i dette segment, er at vi tjekker for parallelle kanter. Måden vi her gør det på, er at vi først og fremmest kører vores kanter igennem diverse mappings for at være sikre på, at de korresponderer med hinanden. Det vil sige, at hvis vi har en kant, $E(1,2)$ i R, tjekker vi om den svarer til en kant i K og D. I det tilfælde, hvor den kant er i D, men ikke K, men vi stadig skal tilføje kanten, da den er i R, prøver den at tilføje en parallel kant, hvilket ikke er tilladt i DPO, og derfor stopper programmet. Ellers bliver kanten blot tilføjet. H er nu færdigbygget, transformationen er fuldført.

Til slut i funktionen, printer vi knuderne og kanterne ud fra H, så man evt. kan tjekke manuelt i hånden, om transformationen er fuldført korrekt. Vi bruger desuden også vores 'my_show' funktion til at visualisere H.



Figur 12: Et eksempel på en DPO-transformation. D bliver konstrueret som følge af G, da L kun består af knuden 1, og K fjerne ikke denne knude. Der tilføjes imidlertid tre ekstra knuder samt fire ekstra kanter fra R.

Vi har yderligere implementeret nogle ekstra funktioner, hvilket vi kommer ind på i kapitel 5.1.1, som vi bruger til tests.

Kapitel 5: Anvendelse og evaluering

Inden vi går i dybden med de tests, som vi har gennemgået, er det imidlertid vigtigt at forstå, at programmet består af flere komponenter, som alle sammen skal fungere, for at vi kan udføre relevante tests. De tests, som vi kommer ind på i kapitel 5.1, er lavet på baggrund af, at alle disse komponenter virker, og vi vil ikke gå i dybden med detaljer om de basale tests, som vi har gennemgået for at tjekke for korrekthed. Vi vil i stedet her kort nævne de små tests, som vi har gået igennem for at komme frem til det endelige program. Det er dog ikke desto mindre muligt at følge med i disse små tests som følge af figurerne på bilag 3[51], men vi går som sagt ikke i detaljer med dem. I stedet vil vores tests, som vi går i detaljer med, være fokuseret på brugen af vores algoritmer på mere interessante og større eksempler, som formaldehyd og glykolaldehyd m.m. Vi vil desuden teste hvor hurtigt vores program kører for givne cases, og hvor mange grafer, der bliver genereret for disse cases.

De små tests, som vi har kørt for at komme frem til at programmet fungerer som helhed, har været at tjekke hver del af programmet enkeltvis. Dette inkluderer først og fremmest tilføjelsen og fjernelsen af knuderne i D, samt hernæst tilføjelsen og fjernelsen af kanterne i D. Vi valgte helt at konstruere D først, før vi begyndte at kigge på H, selvom man måske kunne have tilføjet alle knuderne til hhv. D og H, og dernæst kigget på kanterne. Vi har gjort meget brug af vores 'my_show' funktion til tests af korrekt visualisering, samt Pythons 'print' funktion til at tjekke om de rigtige knuder og kanter blev visualiseret. På samme måde testede vi H, hvor vi først kørte tests med knuderne og dernæst kanterne.

5.1 Anvendelse

I dette afsnit, vil vi gå i dybden med større molekylære grafer, for at se, hvordan den rent faktisk vil agere i forhold til virkelige molekyler. Vi vil komme ind på, om det reelt virker for større eksempler, samt hastigheden af vores program, hvor vi vil kigge på køretiderne af forskellige testtilfælde. Vi vil desuden komme ind på hvor mange grafer, der bliver genereret i forskellige testtilfælde. I den første test vil vi blot transformere et enkelt glykolaldehyd-molekyle, og se på, hvordan vores program håndterer dette. Vi benytter os af *enol-keto*[52] reglen *(*inverse af ketonolF) til denne transformation. Det første vi er interesseret i, er at se om transformationen bliver lavet korrekt. I takt med, at dette ikke er et specielt stort eksempel, kan vi nemt følge DPO-reglerne og tjekke det i hånden. Som følge af bilag 3 figur 24, kan vi ved at printe alle delene ud nemt følge med i transformationen, og vi kan her se, at selve transformationen er udført korrekt. K fjerner den korrekte kant i

[51] Bilag 3
[52] Figur 13

konstruktionen af D, samt R tilføjer den korrekte kant i konstruktionen af H. Figur 25 på bilag 3 visualiserer omtalte graf.

Vi har desuden tilføjet en køretidsfunktion, der viser hvor hurtigt vores algoritme kører. I dette tilfælde, kan vi gætte os til, at algoritmen vil køre forholdsvis hurtigt, da grafen ikke er så stor. Efter at have kørt algoritmen igennem et par gange, kan vi aflæse køretiden til at være $\sim 0{,}005$ sekunder. Det lyder måske ikke af meget, men da algoritmen kører i polynomisk tid, dvs. den er bundet af størrelsen af input, kan vi allerede nu tænke os til, at jo større grafer og regler som input, kan vi hurtigt komme op i meget højere køretidstilfælde.

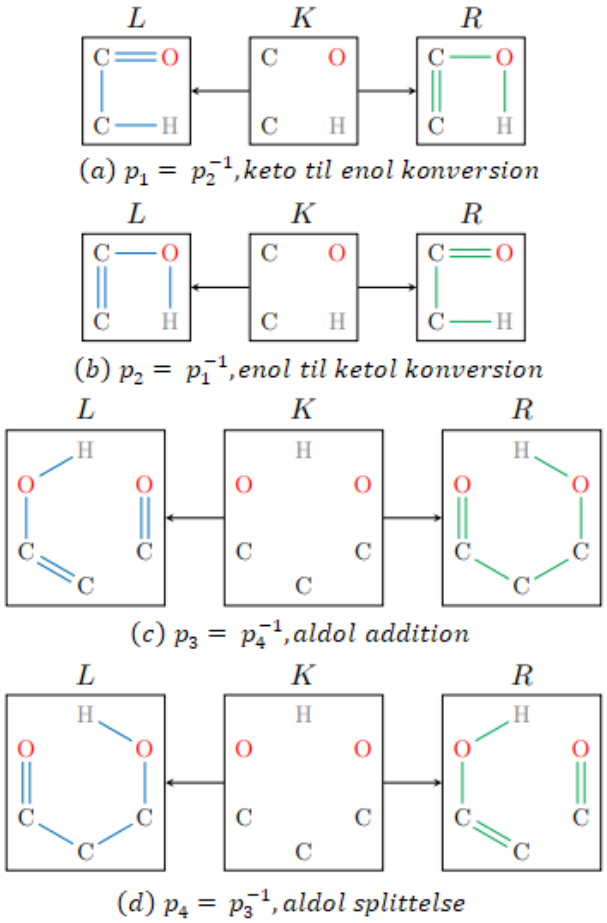
En anden ting, som er meget relevant for køretiden, er antallet af *double pushouts* vi reelt set får lavet i algoritmen. Dette er nemlig også altafgørende alt efter de grafer, som vi giver som input. Grafen i eksemplet, som vi snakker om pt, er som sagt den kemiske reaktion på molekylet glykolaldehyd. Vi har indsat en simpel *count* funktion, og kan her se, at vi kommer op på ét tilfælde af forskellige outputs. Hvis ikke vi havde labels på knuderne, ville denne transformation have set helt anderledes ud, og vi ville komme op på 18 transformationer, hvilket ville have øget køretiden til næsten $\sim 0{,}031$ sekunder. Dette støtter argumentet om, at den altså stiger i forhold til antallet af produkter, H. I takt med, at vi tester med molekyler, er der dog kun én mulig reaktion med denne kombination af reaktant og regel. Det er derfor essentielt at have labels på graferne, når vi arbejder med kemi, hvilket er ærgerligt i den forstand, at vi ikke har fået labels på vores kanter endnu, da det også ville have en indflydelse på antallet af transformationer.

Vi nævnte i kapitel 4, at vi havde taget grafer og regler fra MØD, da det gav mening i den forstand, at de allerede var givet. Kemisk set, er der dog også en grund til, at vi har valgt netop de regler og de grafer. Når vi arbejder med DPO ved kemiske reaktioner, er det vigtigt at bide mærke i, at vi på den ene side har vores reaktanter, dvs. vores G, samt produkterne af reaktionen, H. På den anden side har vi reglen, der siger hvilken kemisk transformation der skal ske, altså hvordan elektroderne de kan flytte rundt. Det betyder, at hvert af de matches vi får mellem G og L, når vi i starten af algoritmen søger efter morfier, er en måde elektronerne kan flytte rundt på. De følger alle det samme mønster, de er blot forskellige steder. Vi nævnte kort i eksemplet før, at vi fik én H-graf som output. Det betyder, at ud fra ét glykolaldehyd-molekyle, samt keto-enol reglen, så får vi altså ét muligt produkt. I virkeligheden er det tilfældigt hvad man får ud af det, men antallet af H-grafer er potentielt mulige måder, som de kan reagere på kemisk set. Et mere interessant eksempel, hvor vi rent faktisk kan benytte det i det "virkelige" liv, er formosereaktionen (the formose reaction).

5.1.1 Formosereaktionen

Vi vil ikke gå i dybe detaljer med formosereaktionen, da det er et stort emne for sig selv, og man vil kunne gå meget dybere ind i emnet rent kemisk set, end vi er i stand til. Vi vil dog forklare det i en sådan grad, at det kan forstås hvorfor det er

vigtigt, samt hvorfor vi har valgt lige netop denne reaktion, som vores hovedanvendelse af vores program. Formosereaktionen er først og fremmest en kompleks proces af kemiske reaktioner, hvor formaldehyd bliver konverteret til sukker. Denne proces består af en masse mellemliggende molekyler, som f.eks. glykolaldehyd. Noget af det meste interessante ved formosereaktionen er, at den er autokatalytisk. Autokatalytiske reaktioner er dem, hvor et af produkterne af reaktionen, agerer som katalyst til reaktionen. Det vil sige, at selvom processen i starten er langsom, stiger hastigheden af konversioner i en hastig udvikling. Grunden til, at den gør det, er at en af katalysatorerne, glykolaldehyd, er med til at starte reaktionen, samtidig med, at den producerer endnu et glykolaldehyd-molekyle. Måden hvorpå dette foregår, er at vi med to formaldehyd-molekyler, samt et glykolaldehyd-molekyle kan frembringe endnu et glykolaldehyd-molekyle (fremadrettet vil vi omtale denne kombination af molekyler som $G_g^{f_2}$). Det kræver desuden en kombination af de fire transformationsregler, som vi har implementeret: keto-enol, aldol addition, enol-keto (inverse af keto-enol) og aldol splittelse (den inverse aldol addition), som kan ses i figur 13.



Figur 13: Transformationsreglerne i formosereaktionen. I (a) og (b) ser vi reglerne, der modellerer hhv. keto-enol og enol-keto processen. I (c) og (d) ser vi modeller for reglerne aldol addition og aldol splittelse.[53]

Med den "korrekte" sekvens af disse regler, kan vi altså komme hen til klassiske formose cyklus og frembringe endnu et glykolaldehyd-molekyle. I takt med, at vores program ikke har labels på kanter, vil vi aldrig komme frem til det korrekte resultat. Vi prøver dog at komme så tæt på som muligt, og vi har imidlertid opstillet en tabel, hvor vi har brugt reglerne på $G_g^{f_2}$. Vi vil herfra benytte reglerne igen på vores resultatgrafer og kommentere på resultatet af dette.

[53] [Andersen 2016, Thesis]

Tabel 1: Transformation hvor reglerne bliver brugt på kombinationen af to formaldehyd- og ét glykolaldehyd-molekyle.

	Transformationsregel	**Tid (i sekunder)**	**Antal resultatgrafer**	
	Aldol addition	0,001	2	
	Aldol splittelse	0	0	
	Keto-enol	0,001	3	
	Enol-keto	0,000	1	

I forhold til tiden i tabel 1, kan det virke som om, at vores transformation udføres ekstremt hurtigt. I takt med, at antallet af resultatgrafer er lavt – eller ikke eksisterer – er dette dog forventeligt. Vi har yderligere ikke taget til overvejelse den "korrekte" sekvens af brugen af disse regler, da vi alligevel ikke vil komme frem til det korrekte resultat. I stedet benytter vi os blot af alle fire regler på graferne for at se, hvordan de agerer.

Tabel 2: Transformationen hvor reglerne bliver brugt på resultatgraferne fra tabel 1.

	Transformationsregel	**Tid (i sekunder)**	**Antal resultatgrafer**	
	Aldol addition	0,052	14	
	Aldol splittelse	0,019	2	
	Keto-enol	0,080	22	
	Enol-keto	0,039	8	

Vi har yderligere tilføjet en funktion 'saving', der gemmer alle resultatgraferne fra vores 'transformation'-funktion. Dette har vi gjort, så vi er i stand til at bruge vores transformationsregler på disse resultatgrafer. Det bliver gjort med funktionen 'transformation2'. Ud fra tabel 2 kan vi se, at i takt med at graferne bliver større, får vi flere resultatgrafer. Dette kan yderligere ses i forhold til tiden, som stiger gevaldigt. Allerede her kan man forestille sig, hvordan den autokatalytiske proces fungerer, da antallet af resultatgrafer stiger gevaldigt i takt med, at graferne bliver større. Ud fra tabel 2 kan vi yderligere se, at aldol splittelse reagerer med resultatgraferne, selvom den ikke gjorde ved begyndelsen. Dette viser, at selvom aldol splittelse ikke reagerede direkte med $G_g^{f_2}$, kan den godt reagere "senere" inde med nogle af de reaktioner vi frembragte ud fra tabel 1.

Tabel 3: Transformationen hvor reglerne bliver brugt på resultatgraferne fra tabel 2.

	Transformationsregel	**Tid (i sekunder)**	**Antal resultatgrafer**	
	Aldol addition	0,208	32	
	Aldol splittelse	0,000	0	
	Keto-enol	0,386	74	
	Enol-keto	0,014	6	

I vores sidste testcase benyttede vi os af alle resultatgraferne fra tabel 2 og brugte endnu engang reglerne på disse resultater. Vi tilføjede endnu en funktion 'transformation3' til at udføre denne proces. Ud fra tabel 3 kan vi tydeligt se, hvordan hhv. aldol addition og keto-enol kan lave mange reaktioner. Antallet af resultatgrafer er steget drastisk, og tiden støtter dette. Vi kan tænke os til, at dette resultat kun vil stige løbende, og i forhold til formosereaktionen understøtter dette yderligere den autokatalytiske proces. Det mest iøjefaldende derimod, er at antallet af reaktioner er faldet både ved aldol splittelse og enol-keto, hvilket vi fandt meget interessant. Vi er dog imidlertid usikre på, om dette er normalt i forhold til kemiske reaktioner, eller om det blot er vores programs mangel på labels på kanter, der fører til dette resultat.

Kapitel 6: Konklusion og fremtidigt arbejde

I denne rapport har vi præsenteret forskellige definitioner, algoritmer og teori, der i sidste ende har givet en bedre forståelse for, hvordan graftransformation virker. Vi har introduceret kategori-teori, hvor vi kom ind på forskellige relationer og morfier mellem grafer, samt hvorfor de her morfier er vigtige i forhold til graftransformation. Vi er yderligere kommet ind på Double Pushout, da det var den form for graftransformation, som vi fokuserede på. Her snakkede vi tilmed om de forskellige graftransformationsregler, og hvordan de virker i DPO. Vi har desuden implementeret vores eget graftransformationsprogram, der er i stand til at transformere grafer, som det foregår i DPO. Dette program kan desuden transformere grafer som værende molekyler. Det vil sige, at det kan indlæse molekyler som grafer med labels og artificielt kreere kemiske reaktioner. Vores program er dog imidlertid ikke i stand til at transformere grafer med labels på kanterne, hvorfor vi ikke med 100% tilfredsstillelse kan sige, at en graftransformation er afbildet ud fra, hvordan den ville se ud i virkeligheden. At få labels på kanter såvel som vi også har på knuderne, er derfor en funktion, der vil blive tilføjet i fremtiden, og med denne tilføjelse, vil vi helt være i stand til at transformere molekyler. Hvis vi dog blot kigger på graferne som ikke-orienterede grafer uden labels på kanterne, men blot på knuder, virker den. Vi har desuden brugt vores program til at analysere kemiske reaktioner med fokus på formosereaktionen. Derfra kunne vi først og fremmest konkludere, at programmet virkede som det skulle, med henblik på, at kanterne ingen labels har. Desuden kunne vi med overbevisning se, at jo flere iterationer vi lavede af graftransformationerne, jo flere resultatgrafer fik vi, samt jo længere tid tog køretiden for disse funktioner. Det er dog ydermere pga. hvordan vores funktioner er sat op. Vil vi f.eks. køre funktionen 'transformation3', som kører transformationen efter tre iterationer, kører den først de andre transformationsfunktioner, før at den kommer til den tredje iteration. Den ville køre væsentligt hurtigere, hvis den blot kiggede på antallet af resultatgrafer fra forrige funktion. Vi var dog imidlertid usikre på, hvordan vi skulle opsætte det anderledes, og da vi kun har tre iterationer, er det ikke noget problem.

I forhold til vores implementering af funktionerne 'transformation2' og 'transformation'3, er det desuden ærgerligt, at vi ikke formåede at få lavet en mere generel funktion. Dette gør, at vi kun er i stand til at køre tre iterationer. Hvis vi ville køre flere iterationer, skulle vi indsætte flere af samme slags funktioner, hvilket på længere sigt ikke er ideelt. Denne problemstilling gør yderligere, at vi ikke formår at komme op på så mange iterationer og testcases, at vores program ville tage for lang tid at køre. Dette kunne være interessant, for at se hvor mange resultatgrafer, den reelt set ville være i stand til at kreere, før køretiden ville være for lang. Vi mangler dog disse generelle meta-algoritmer for at gøre dette, som på sigt nok ville være en af de næste funktioner, som vi ville implementere.

Bilag

Bilag 1: Scripts

```
g = smiles("C=O", name="Formaldehyde")
p = GraphPrinter()
p.edgesAsBonds = False
g.print(p)

print("g = nx.Graph()")
for v in g.vertices:
    print("g.add_node({}, attribute={{label: \"{}\"}})".format(v.id, v.stringLabel))
for e in g.edges:
    print("g.add_edge({}, {})".format(e.source.id, e.target.id))
```

Figur 14: Script på MØDs Live Playground, der blev brugt til indlæsning af de forskellige molekyler ud fra SMILES-format. Figur 15 er resultatet af dette script, som herefter kan kopieres direkte ind i en python-fil, og på den måde er molekylet blevet genereret

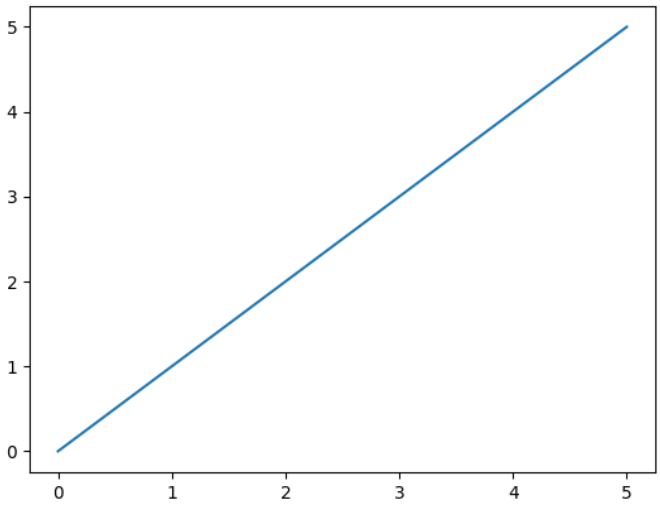
```
Executing code from 'input.py'
g = nx.Graph()
g.add_node(0, attribute={label: "C"})
g.add_node(1, attribute={label: "O"})
g.add_node(2, attribute={label: "H"})
g.add_node(3, attribute={label: "H"})
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(0, 3)
End of code from 'input.py'
```

Figur 15: Resultatet af scriptet fra figur 14..

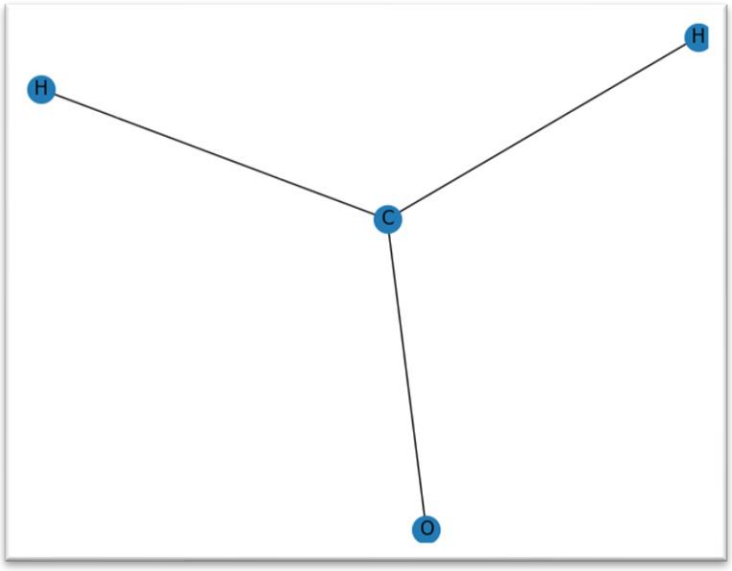
```
rule = ketoEnol_F

g = rule.left
print("L = nx.Graph()")
for v in g.vertices:
    print("L.add_node({}, attribute={{label: \"{}\"}})".format(v.id, v.stringLabel))
for e in g.edges:
    print("L.add_edge({}, {})".format(e.source.id, e.target.id))

g = rule.context
print("K = nx.Graph()")
for v in g.vertices:
    print("K.add_node({}, attribute={{label: \"{}, {}\"}})".format(v.id, v.core.left.stringLabel, v.core.right.stringLabel))
for e in g.edges:
    print("K.add_edge({}, {})".format(e.source.id, e.target.id))

g = rule.right
print("R = nx.Graph()")
for v in g.vertices:
    print("R.add_node({}, attribute={{label: \"{}\"}})".format(v.id, v.stringLabel))
for e in g.edges:
    print("R.add_edge({}, {})".format(e.source.id, e.target.id))

print("mapL = {}")
for v in rule.vertices:
    print("mapL[{}] = {}".format(v.context.id, v.left.id))

print("mapR = {}")
for v in rule.vertices:
    print("mapL[{}] = {}".format(v.context.id, v.right.id))
```

Figur 16: På samme made som scriptet i figur 14, bliver reglerne genereret. Alle komponenterne af reglen bliver genereret, og outputtet af dette script, giver et lignende resultat som figur 14, blot for alle delene af reglen.

Bilag 2: Visualiseringer

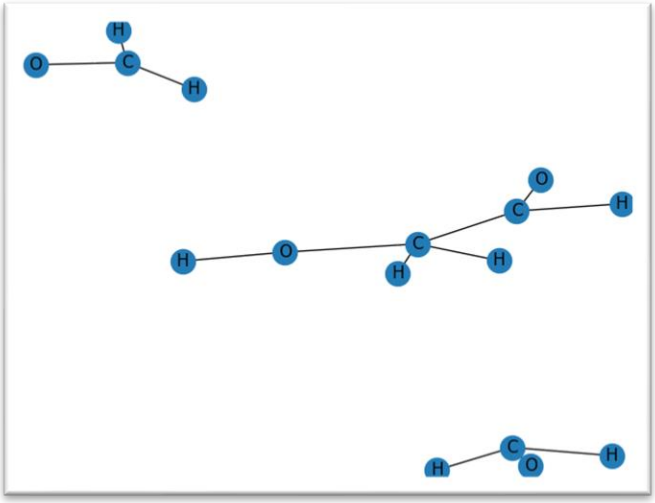


Figur 17: Visualisering af mappingen af K og R af reglen, Aldol addition. I form af en dictionary ser reglen således ud som {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}, og når vi kigger på visualiseringen, kan vi nemt se hvordan de forskellige nøgler og værdier spiller op mod hinanden i form af at nøglerne svarer til y-aksen og værdierne x-aksen.



Figur 18: Visualisering af molekylet, formaldehyd, som blev genereret som følge af scriptet i figur 14. Vha. vores funktion 'my_show', kan vi nemt visualisere vores grafer, og her kan vi se hvordan molekylet består af atomerne: karbon, hydrogen og oxygen.

Bilag

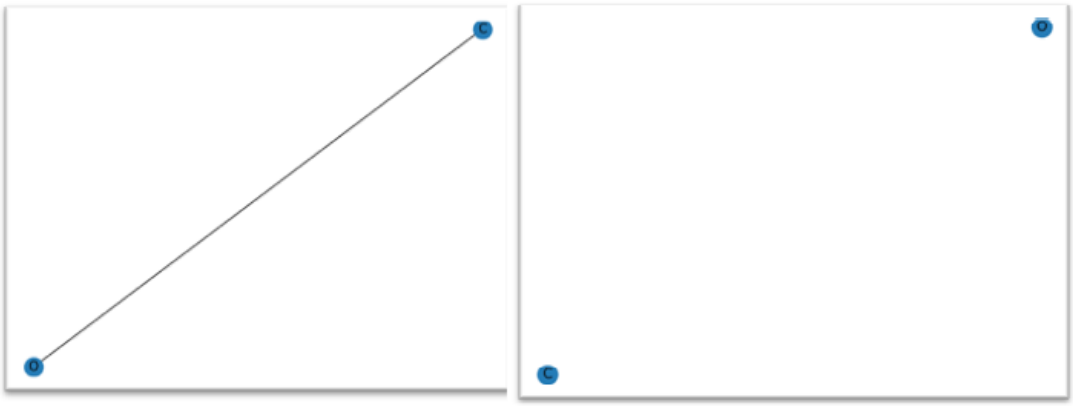


Figur 19: Visualisering af den disjunkte foreningsmængde af to formaldehyd-molekyler, samt et glykolaldehyd-molekyle. Dette er et eksempel på, hvordan vi har brugt vores combine-funktion til at repræsentere de tre molekyler som én enkelt graf.

.

Bilag 3: Tests af transformationen

Figur 20: Vores første tests bestod af konstruktionen af de korrekte knuder til D. I dette tilfælde, består vores graf, G, af knuderne: (1, 2, 3), der er mærket C, O, H. Vores regel K, består af knuderne (1, 2) med de samme labels. På venstre siden, ser vi grafen D, efter alle de tre knuder i G er tilføjet. På højresiden ser vi grafen D, efter den ene knude, der er mærket H, er blevet fjernet.



Figur 21: Vi begyndte, som sagt, først helt at konstruere D før H. Dvs., efter vi havde lavet tests på knuderne af D, begyndte vi på kanterne af D. I dette tilfælde, består grafen G, udover knuderne (1, 2, 3) også af en kant (1, 2). Den bliver tilføjet først, som kan ses i venstre billede. Herefter, bliver den fjernet hvis ikke den er i K, som den ikke er i dette tilfælde, og D er nu færdigkonstrueret som følge af højresiden af billedet.

På samme måde, som figur 20 og 21, har vi naturligvis kørt mange flere og større tests end blot disse eksempler for at være sikker på, at transformationen af *D* virker for alle scenarier. Dette inkluderer b.la. tests af *dangling edges*, samt tests af hvor knuderne i *L* og *K*, ikke svarer til knuderne i G, dvs. tjekke om den fjerner de *korrekte* kanter og knuder i forhold til mappingen mellem graferne osv.

```
G nodes:  [1, 2, 3, 4]
G edges:  [(1, 3)]
Mapping from L to G:  {1: 1, 2: 2, 3: 4}
Node in mapM 1 Corresponding to  1  in G
Node in mapM 2 Corresponding to  2  in G
Node in mapM 3 Corresponding to  4  in G
Dangling edge
```

Figur 22: Eksempel på en 'dangling edge' case. I dette eksempel prøver K at fjerne knude 1, men i takt med, at der er en kant mellem knude 1 og 3 (svarende til 1 og 4), vil vi have en dangling edge, hvis det blev fuldført.

Figur 23: Test af konstruktion af H. Samme eksempel som i figur 20 og 21. H består af knuderne i D, samt de ekstra knuder, der skal tilføjes fra R. R består af knuderne (1, 2, 3 og 4) med labels P og L på knude 3 og 4, samt en kant (2, 4) . Det kan ses, at selvom R består af fire knuder, tilføjer den kun de knuder, som ikke er i K, som bestod af knude 1 og 2. Herefter skal kanterne tilføjes. I dette tilfælde bestod D ikke af nogen kanter, da K fjernede dem, men R tilføjer en ekstra kant. I netop dette tilfælde, er der to morfier. I den ene, som på dette billede, tilføjer den en kant mellem C og P, men i den anden morfi, tilføjer den en kant mellem O og P.

Grunden til, at den i figur 23 tilføjer en kant imellem to forskellige knuder, handler om, hvor vores funktion finder morfier. Der kan som sagt findes mange morfier, imellem de forskellige knuder. Vi har tilføjet en kort *print*-funktion, der viser hvilke knuder i G, der svarer til hvilke knuder i *L*. Dette kan ses i figur 21.

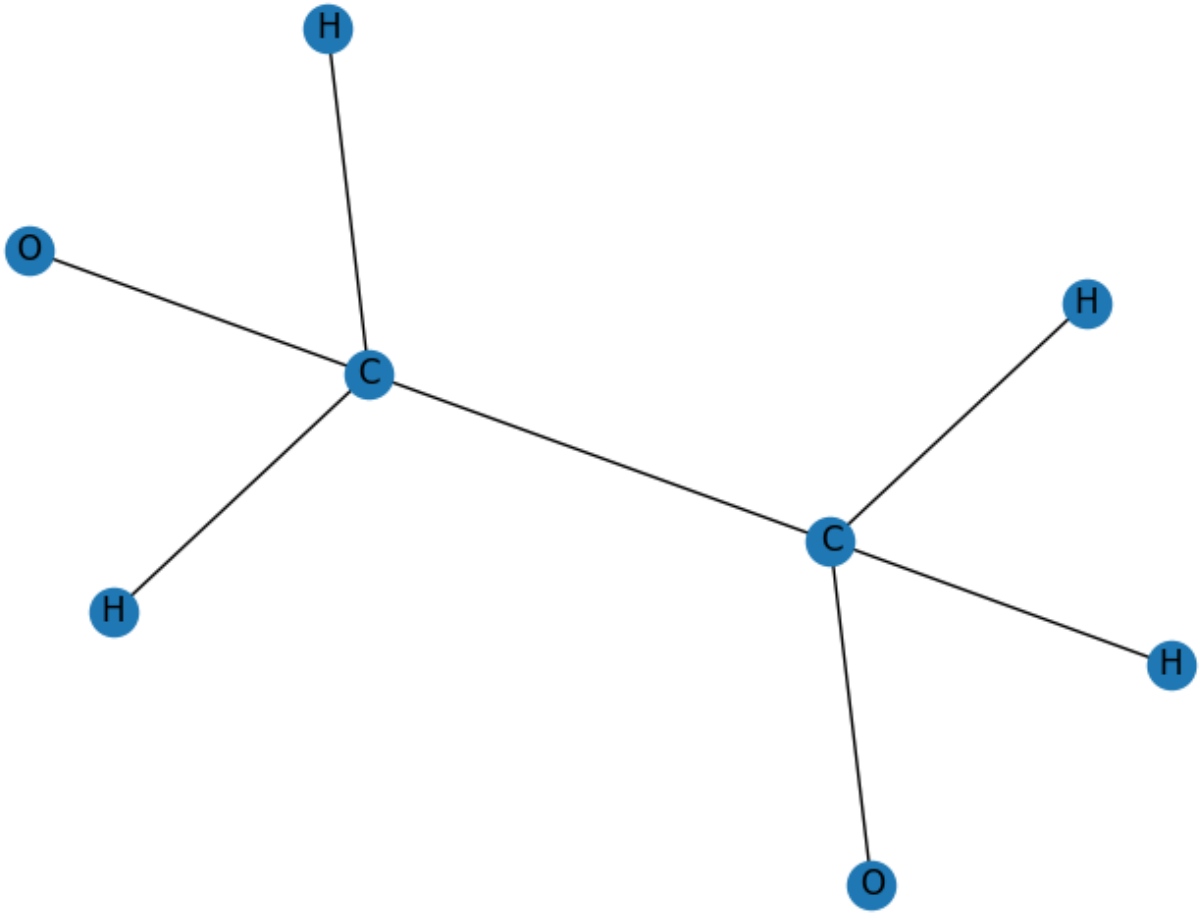
```
Node in mapM 1 Corresponding to  1  in G
Node in mapM 2 Corresponding to  2  in G
Node in mapM 3 Corresponding to  3  in G
H nodes [1, 2, 3, 4]
H edges [(2, 4)]
TRANSFORMATION DONE
```

```
NEW TRANSFORMATION STARTING
Node in mapM 1 Corresponding to  2  in G
Node in mapM 2 Corresponding to  1  in G
Node in mapM 3 Corresponding to  3  in G
H nodes [1, 2, 3, 4]
H edges [(1, 4)]
TRANSFORMATION DONE
```

Figur 24: Her kan ses hvilke knuder i G, der svarer til hvilke knuder i L (ergo også K og R) ud fra de forskellige matches. I eksemplet i figur 23, har vi at alle knuderne svarer til de samme knuder, og ergo tilføjes der en kant (2, 4) til H. I næste morfi derimod, svarer knude 2 i L til knude 1 i G, og derfor, når R tilføjer kanten (2, 4), tilføjer den faktisk en kant imellem knuderne 1 og 4 i H.

```
G nodes:  [0, 1, 2, 3, 4, 5, 6, 7]
G edges:  [(0, 1), (0, 4), (1, 2), (1, 5), (1, 6), (2, 3), (2, 7)]
Mapping from L to G:  {0: 2, 1: 1, 2: 0, 3: 4}
Node in mapM 0 Corresponding to  2  in G
Node in mapM 1 Corresponding to  1  in G
Node in mapM 2 Corresponding to  0  in G
Node in mapM 3 Corresponding to  4  in G
L edges [(0, 1), (1, 2), (2, 3)]
L nodes [0, 1, 2, 3]
K edges [(0, 1), (1, 2)]
K nodes [0, 1, 2, 3]
D edges: [(0, 1), (1, 5), (1, 6), (1, 2), (2, 3), (2, 7)]
D nodes [0, 1, 2, 3, 4, 5, 6, 7]
R nodes [0, 1, 2, 3]
R edges [(0, 1), (0, 3), (1, 2)]
H nodes [0, 1, 2, 3, 4, 5, 6, 7]
H edges [(0, 1), (1, 5), (1, 6), (1, 2), (2, 3), (2, 7), (2, 4)]
Number of transformation:  1
TRANSFORMATION DONE

NEW TRANSFORMATION STARTING
Time spent 0.0009918212890625
```

Figur 25: Transformation af molekylet glykolaldehyd med reglen ketonolB (det inverse af keto-enol) som regel-input. Denne figur viser udkommet af terminalen step by step. Den viser først og fremmest knuderne og kanterne i G. Herefter print den mappingen mellem L og G ud, og udskriver hvilke knuder i L svarer til hvilke i G, samt printer de kanter ud, som er i både L og G. Efterfølgende printer vi alle kanter/knuderne ud for alle delene af reglen, samt de konstruerede grafer, for at vi nemt kan verificere, at grafen H er lavet korrekt.



Figur 26: Visualisering af grafen, der blev konstrueret i figur 25.

Litteraturliste

[Ehrig et al. 2006] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange og Gabriele Taenthzer. Fundamentals of algebraic graph transformation. Springer-Verlag, Berlin, D, 2006. (Citeret på side 1, 3, 4, 6 og 8.)

[King 1983] R. B. King. Chemical Applications of Topology and Graph Theory. Elsevier, 1983. (Citeret på side 7)

[PR69] J.L. Pfaltz og A. Rosenfeld. Web Grammars. In D.E. Walker og L.M. Norton, editors, Proceedings of IJCAI 1969, sider 609–620. William Kaufmann, 1969. (Citeret på side 1)

[Pra71] T.W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. Journal of Computer and System Sciences, 5(6):560–595, 1971. (Citeret på side 1)

[Dittrich et al. 2001] Peter Dittrich, J. Ziegler og Wolfgang Banzhaf. Artificial chemistries — a review. Artifical Life, vol. 7, no. 3, pages 225–275, 2001. (Citeret på side 1)

[Cordella et al. 2001] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone og Mario Vento. An improved algorithm for matching large graphs. In Proc. of the 3rd IAPRTC15 Workshop on Graph-based Representations in Pattern Recognition, side 149–159, 2001. (Citeret på side 5)

[Cordella et al. 2004] L.P. Cordella, P. Foggia, C. Sansone og M. Vento. A (Sub) Graph Isomorphism Algorithm for Matching Large Graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, side 1367, 2004. (Citeret på side 5)

[Andersen 2016] Jakob L. Andersen. MedØlDatschgerl (MØD). https://cheminf.imada.sdu.dk/mod/, http://jakobandersen.github.io/mod/dataDesc/dataDesc.html, (Citeret på side 6 og 8, sidst tilgået 01/06/2021)

[Andersen 2016, Thesis] Jakob L. Andersen. Analysis of Generative Chemistries, 2016 (citeret på side 8, 9, 11, 12 og 25)

[Hagberg 2005] Aric Hagberg. NetworkX (online). http://networkx.org (citeret på side 6, 10, 12 og 14, sidst tilgået 01/06/2021)

[Hunter 2003] John D. Hunter. Matploblib (online). https://matplotlib.org/ (citeret på side 7, sidst tilgået 01/06/2021)

[Rinke 2008] Wikipedia (online): https://en.wikipedia.org/wiki/Disjoint_union_of_graphs (Figur brugt på side 12, sidst tilgået 01/06/2021)

[Armstrong 1997] Armstrong, M. A. Basic Topolopy, New York: Springer-Verlag, 1997. (Citeret på side 12)

[Eyben et al. 2008] Florian Eyben, Martin Wöllmer, and Björn Schuller. OpenSmiles (online). http://opensmiles.org/opensmiles (citeret på side 8, sidst tilgået 01/06/2021)

[Kroon 2020] P C Kroon. Pysmiles (online). https://pypi.org/project/pysmiles/ (citeret på side 9, sidst tilgået 01/06/2021)

[Andersen et al. 2016] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, og Peter F. Stadler. A Software Package for Chemically Inspired Graph Transformation (online). https://arxiv.org/abs/1603.02481 (citeret på side 3, sidst tilgået 01/06/2021)

[Gritter 2018] Mark Gritter. Double Pushouts on Graphs (online). https://steemit.com/mathematics/@markgritter/double-pushouts-on-graphs (citeret på side 6, sidst tilgået 01/06/2021)