18/12 2020

Rubik's terning

Asbjørn Brummer Birkelund

3.xy NEXT Vibenshus Gymnasium HTX

Vejledere:

Sarah Bredgaard Stampe Hjorth (matematik)

Jacob Debel (programmering)

Anslag: 47961

Forside:

Rubik’s Terning

Asbjørn Brummer Birkelund

3.xy

NEXT Vibenshus Gymnasium HTX

tegn: x

Vejledere: Sarah Bredgaard Stampe Hjorth (matematik), Jacob Debel (programmering)

Resumé

I dette projekt bliver det undersøgt, hvordan gruppeteori kan anvendes i praksis ved at løse Rubik's terning programmatisk. Først undersøges terningen. Terningen består af 6 midterbrikker, 12 kantbrikker og 8 hjørnebrikker. Der er i alt $4{,}3 \cdot 10^{19}$ konfigurationer af terningen. Derefter redegøres det for matematiske grupper. Der redegøres for, hvad en gruppe er, samt permutationsgrupper og herunder symmetriske grupper. Permutationsgrupper er en gruppe af måder, et hvis antal elementer kan permuteres. En matematisk model konstrueres af Rubik's terning som en permutationsgruppe. Permutationsgruppen hedder $(G,\circ)$ og er en undergruppe til den symmetriske gruppe S_{48}. Permutationerne permuterer de 48 farver på terningen, der faktisk bevæger sig. Efterfølgende diskuteres de forskelige løsningsmuligheder af terningen, samt hvordan menneskers og computeres løsningsmetoder varierer. Computermetoderne anvender gruppeteorien, og de reducerer terningen til mindre undergrupper, som er lettere at løse.
I den anden halvdel udvikles der et program til at løse terningen. Programmet anvender Thistlethwaites algoritme, som reducerer terningen i 4 trin. For at finde løsningerne, anvendes der en træsøgealgoritme, der kan søge igennem de mulige kombinationer af drej. Den mest effektive algoritme viste sig at være IDDFS (Iterative Deepening Depth-First Search). Til sidst diskuteres mennesker og computeres styrker og svagheder i forhold til komplekse problemer. Det konkluderedes, at computeren er et redskab til at løse problemer, og kan ikke løse problemer alene, men samtidig er der problemer, mennesket kan løse uden hjælp.

Indholdsfortegnelse

Indledning

Dette projekt anvender gruppeteori og den algebraiske struktur permationsgrupper til at analysere Rubrik's terning matematisk. Forskellige metoder til løsning af terningen gennemgås, herunder hvorledes menneskers metoder afviger metoder anvendt af computere. På baggrund af dette skrives et computerprogram, der kan løse Rubriks terning fra en vilkårlig konfiguration. I forbindelse med programudviklingen gennemgås forskellige træsøgealgoritmer. Projektet ser også på optimerings muligheder i det udviklede program. Slutteligt diskuteres mennesker og computeres forskellige måder at løse problemer.

Metode

I dette projekt anvendes matematik og programmering til at belyse emnet og besvare problemformuleringen.

Fra matematikken benyttes der hovedsageligt modellering. Matematikkens rolle i projektet er overordnet at undersøge, hvordan Rubik's terning kan modellereres matematisk ved anvendelse af gruppeteori. Dette supplerer programmeringen, da det giver et grundlag for, hvordan problemet kan anskues og hjælper med at designe en løsning, der implementeres i et computerprogram.

Til at skrive programmet anvendes den iterative proces. På baggrund af matematikken kan programmet planlægges til en vis grad. Den iterative proces giver fleksibilitet til at forsøge flere løsningsmuligheder, hvilket kan være nødvendigt, når der arbejdes med et komplekst problem som Rubik's terning, der ikke nødvendigvis kun har én løsning.

Mens gruppeteorien anvendes I udviklingen af den model, der skal programmers, anvendes lidt deskriptiv statistik til at evaluere programmets ydeevne. Dette hjælper med at finde måder at videreudvikle programmet gennem den iterative proces.

Introduktion til terningen

Figur 1 Rubik's terning, til venstre: løst. Til højre: blandet

Rubik's terning, også kendt som professorterningen, er et mekanisk puslespil opfundet af den ungarske arkitekt og professor Erno Rubik i 1974 (Alter, 2020). Han opfandt den oprindeligt som et redskab til at undervise sine elever. Det var først senere, efter at have malet hver side i hver sin farve, at han opdagede, at det var et udfordrende puslespil.

Terningen er delt op i 3x3x3 felter, der alle kan bevæges rundt på hele terningen. Den har 54 felter med 6 forskellige farver. For at terningen er løst, skal der kun være én farve på hver side af terningen. Hver af terningens seks sider kan dreje uafhængig af hinanden. Umiddelbart ville man tro, at man skal holde styr på alle 54 felter for at løse terningen, hvilket ville være utroligt uoverskueligt. Ved at kigge på selve mekanismen ser man dog, at terningens struktur er en mere simpel.

Terningen består af 3 forskellige typer brikker; midter, kanter og hjørner.

Figur 2 De 3 typer brikker. hhv. midterne, en kant og et hjørne

Midterbrikkerne har kun én farve hver. De er alle forbundet i midten af terningen. De kan derfor aldrig flytte sig i forhold til hinanden, lige meget, hvordan man drejer siderne, og de kan derfor ses som én brik.. Dermed kan midterne bruges som en reference til, hvilken vej terningen vender, samt hvilken side, man drejer.

Kanterne har to farver hver. De rører to midterbrikker, og kan derfor påvirkes med et drej på én af de to sider. Brikken kan vende på to forskellige måder i samme position.

Figur 3 De to måder, en kant kan vende i samme position

Hjørnerne har tre farver hver. På samme måde som kanterne rører hjørnerne tre midterbrikker, og kan påvirkes med et drej på én af de tre sider, den rører. Den kan vende på tre forskellige måder i samme position.

Figur 4 De tre måder, et hjørne kan vende i samme position

Grundet at kanter og hjørner har forskellige former, ville en kant aldrig kunne flytte til et hjørnes placering og vice verca. Farverne på en enkel brik kan heller ikke bevæge sig uafhængig af hinanden. Hvis man vil beregne, hvor mange forskellige kombinationer, der er på terningen, skal man altså tage udgangspunkt i brikkerne og ikke de farvede sider.

Der er 6 midter, 12 kanter og 8 hjørner. Siden midterbrikkerne aldrig bevæger sig, kan de ignoreres. Hvis man splittede terningen ad og satte brikkerne ind én efter én, ville den første kant, man satte i, kunne være 12 steder, den næste 11, og så videre. Antallet af forskellige måder siderne kan sættes i på skrives som 12 fakultet:

$$12! = 479\,001\,600$$

Ligeledes gælder det samme for de 8 hjørner:

$$8! = 40320$$

Som nævnt kan hver kant vende på 2 måder og hjørner på 3. Antallet af kombinationer af orienteringer er altså hhv. 2^{12} og 3^8. Den samlede antal konfigurationer, brikkerne på Rubik's terning have, er da:

$$(8! \cdot 2^{12}) \cdot (12! \cdot 3^8) = 519\,024\,039\,293\,878\,272\,000 \approx 5{,}19024 \cdot 10^{20} \qquad (1)$$

Desværre er det ikke helt så simpelt. Hvis man forsøgte at løse en terning, hvor alle brikkerne sidder helt tilfældigt, er der kun en $\frac{1}{12}$ chance for, at det rent faktisk er muligt. Dette skyldes, at der er visse konfigurationer, man ikke kan opnå ved normale drej. For det første kan et enkelt hjørne ikke vende forkert, hvis alle andre vender rigtigt. Siden hjørnet kan vende på 3 forskellige måder, er der kun $\frac{1}{3}$ af tilfældene, der er gyldige. Ligeledes kan man heller ikke have en enkel kant, der vender forkert, når alle andre vender rigtigt. Kanten kan vende på 2 måder, og det er derfor kun $\frac{1}{2}$ af tilfældene, der er gyldige. Til sidst kan to brikker ikke være byttede om, når alle andre er placeret rigtigt. Dette er igen $\frac{1}{2}$ af tilfældene, der er ugyldige. Der er derfor kun $\frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{12}$ forskellige konfigurationer, der faktisk er gyldige. Dette skyldes terningens paritet (Wikimedia Foundation). Det betyder overordnet, at antallet af byttede hjørner og antallet af byttede kanter begge skal være lige eller begge være ulige. Tager man det samlede antal tænkelige konfigurationer og ganger med de $\frac{1}{12}$ (fraktionen af gyldige konfigurationer), finder man det samlede antal mulige konfigurationer til at være:

$$\frac{(8! \cdot 2^{12}) \cdot (12! \cdot 3^{8})}{3 \cdot 2 \cdot 2} = 43\,252\,003\,274\,489\,856\,000 \approx 4{,}3252 \cdot 10^{19} \qquad (2)$$

Et astronomisk stort tal, og det gør det tydeligt, at det ikke er muligt bare at lave en brute force løsning, hvor man prøver alle kombinationer. Et eksempel kunne være et computerprogram, der kan prøve 1 million unikke kombinationer i sekundet. Deler man antallet af konfigurationer med 1 million finder man, at det vil tage:

$$\frac{43\,252\,003\,274\,489\,856\,000}{1\,000\,000\,00} \approx 4{,}3252 \cdot 10^{11}\ sekunder \qquad (3)$$

at søge igennem alle kombinationerne. Dette svarer til 13705 år:

$$\frac{4{,}3252 \cdot 10^{11}}{60 \cdot 60 \cdot 24} \approx 5{,}00602 \cdot 10^{6}\ dage \qquad (4)$$

$$\frac{5{,}00602 \cdot 10^{6}}{365{,}25} \approx 13705{,}7\ år \qquad (5)$$

Man er derfor nødt til at have en metode, der ikke behøver at søge alle 43 trillioner konfigurationer igennem. Med brug af gruppeteori kan man nedbringe antallet af nødvendige permutationer.

Gruppeteori.

Gruppeteori beskæftiger sig med samlinger af elementer, kendt som mængder. Disse elementer kan kombineres med en binær operator. Mængden og operatoren skal sammen overholde en række specifikke aksiomer (regler). Tilsammen danner mængden og operatoren en gruppe (Wellens, 2015).

Mængde

En mængde er en simpel datastruktur. En mængde er bare en samling af elementer (Dalen, Doets, & swart, 2014). Elementerne kan være hvad som helst. Det kan være tal, bogstaver, farver, dyr i junglen, eller noget som helst andet, man kan finde på. Der er ingen restriktioner for, hvilke elementer en mængde kan indeholde. Man kan f.eks. have en mængde, der består af heltallene 1, 2 og 3. Dette skrives som:

$$G = \{1, 2, 3\} \tag{6}$$

Samlingen af alle heltal er også en gruppe, og er betegnet $\mathbb{Z}$ (Nipissing University).

En mængde har i sig selv ikke nogen funktion, og er bare en struktur, der samler elementer.

Operatoren

En binær operator er en funktion, der tager to elementer og giver ét element tilbage. Et eksempel på en operator kunne være addition. Addition tager to tal, lægger sammen, og giver som resultat summen af de to tal tilbage. Når man kombinerer to elementer med en operator, skrives de to elementer ved siden af hinanden med operatorens tegn imellem dem. Hvis man f.eks. vil addere x og y, skrives det:

$$x + y \tag{7}$$

Med nogle operatorer, f.eks. multiplikation, kan operatorens tegn undlades, og man ville da bare skrive xy. Ligeledes kan $x \circ x$ i visse sammenhænge skrives som x^2 (Wellens, 2015).

Gruppe

En gruppe består af en mængde og en binær operator, der påvirker mængden. Gruppen bestående af alle hele tal og addition som binær operator skrives f.eks. som:

$$(\mathbb{Z}, +) \tag{8}$$

For at mængden og den binære operator sammen kan danne en gruppe, skal de overholde 4 aksiomer (Davis, 2006):

1. Lukket mængde

Det første aksiom omhandler lukkethed. Det er krævet, at to elementer fra mængden, der kombineres med operatoren, sammen giver et andet element fra mængden.

$$\forall x, y \in S, \;\; x \circ y \in S \tag{9}$$

Denne linje betyder, at hvis man med den binære operator kombinerer to elementer, x og y, fra mængden S, vil resultatet også eksistere inde i mængden S.

Dette viser, at mængden $G = \{1, 2, 3\}$ fra tidligere ikke kan gå sammen med additionsoperatoren og danne gruppen $(G, +)$, da den ikke overholder aksiomet om lukkethed. Dette kan vises med:

$$2 + 3 = 5 \tag{10}$$

$$5 \notin G \tag{11}$$

Produktet af det to elementet er altså ikke inde i gruppen. Gruppen $(\mathbb{Z}, +)$ overholder til gengæld lukkethed, da to heltal adderet giver endnu et heltal.

2. Associativitet

Det andet aksiom, associativitet, siger, hvis operatoren bruges flere gange, er det ligegyldigt, hvilken operation, man laver først. Det kan skrives som:

$$(a \circ b) \circ c = a \circ (b \circ c) \tag{12}$$

Dette viser f.eks, at man kan danne en gruppe med addition, da addition er associativ (Bertram):

$$(a + b) + c = a + (b + c) \tag{13}$$

Men det samme gør sig ikke gældende for subtraktion. Subtraktion kan altså ikke danne en gruppe:

$$(a - b) - c \neq a - (b - c)$$

3. Neutralt element

Det tredje aksiom er, at mængden skal indeholde et neutralt element. Det neutrale element skal gøre 'ingenting', når det påvirkes af operatoren.

$$\exists e \in S,\ \forall x \in S,\ x \circ e = e \circ g = x \tag{14}$$

Der eksisterer altså et element således at når det kombineres med et vilkårligt element x, bliver resultatet x. En gruppe bestående udelukkende af det neutrale element kaldes den trivielle gruppe og betegnes ofte som e.

I eksemplet $(\mathbb{Z}, +)$ er det neutrale element 0, da addition mellem et vilkårligt heltal og 0 bare giver heltallet igen.

$$x + 0 = x \tag{15}$$

4. Inverst element

Det sidste aksiom er, at der for alle elementer i mængden eksisterer et inverst element. Hvis et element kombineres med sit inverse element, giver de tilsammen det neutrale element.

$$\forall x \in S,\ \exists x^{-1} \in S,\ x \circ x^{-1} = e \tag{16}$$

Dette betyder, at for alle elementer x i mængden S eksisterer et element i S, som er det inverse element, således at et element og dets invers til sammen giver det neutrale element.

I eksemplet $(\mathbb{Z}, +)$ vil det inverse element til en vilkårlig værdi x være $-x$. De vil tilsammen give det neutrale element 0.

$$x - x = 0 \tag{17}$$

Hvis alle 4 aksiomer er overholdt, kan mængden og operatoren sammen danne en gruppe.

Symmetrisk gruppe og permutationsgruppe

En symmetrisk gruppe er en form for gruppe, der omhandler kombinationer af en samling af elementer (Smith, 2018). Der tages udgangspunkt i en samling af 3 elementer. Hvad disse elementer faktisk er, har ingen relevans. I dette eksempel er det tal:

$$S = \{1, 2, 3\} \tag{18}$$

Disse elementer kan ses som at have en bestemt rækkefølge. I dette tilfælde er de i numerisk rækkefølge. Dog er dette ikke den eneste måde at arrangere tallene. Man kunne forestille sig, at man byttede om på to af dem, f.eks. 2 og 3, og så ville man ende med en ny rækkefølge:

$$1, 3, 2 \tag{19}$$

For 3 elementer findes der $3! = 6$ forskellige rækkefølger, elementerne kan arrangeres på:

$$\begin{matrix} 1, 2, 3 \\ 1, 3, 2 \\ 2, 1, 3 \\ 2, 3, 1 \\ 3, 1, 2 \\ 3, 2, 1 \end{matrix} \tag{20}$$

Disse 6 måder at permutere listen af tal kan sammen ses som en mængde og indgå i en gruppe. En sådanne gruppe kaldes en permutationsgruppe.

En permutation er en funktion, der tager elementerne i en mængde og giver dem en ny placering. Der tages udgangspunkt i eksemplet fra tidligere, hvor 2 og 3 byttes rundt. En permutation, der bytter om på 2 og 3, kunne skrives som:

$$\rho(1) = 1$$
$$\rho(2) = 3$$
$$\rho(3) = 2$$

Elementet på plads nummer 2 i mængden ender på plads nummer 3, og elementet på plads 3 ender på plads 2. En permutation kan også skrives som:

$$\rho = \begin{pmatrix} 1 & 2 & 3 \\ \sigma(1) & \sigma(2) & \sigma(3) \end{pmatrix} \tag{21}$$

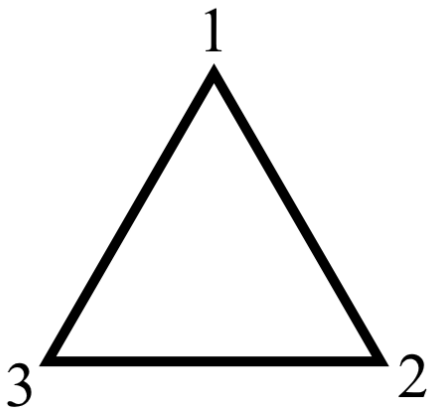
Hvilket i dette eksempel ville være:

$$\rho = \begin{pmatrix} 1\ 2\ 3 \\ 1\ 3\ 2 \end{pmatrix} \tag{22}$$

En permutationsgruppe, der indeholder alle permutationer af n elementer, kaldes en symmetrisk gruppe, S_n (Smith, 2018). Siden en samling af 3 elementer kan have 6 forskellige rækkefølger, har S_3 også 6 forskellige permutationer (navngivningen er arbitrer):

$$\rho_1 = \begin{pmatrix} 1\ 2\ 3 \\ 1\ 2\ 3 \end{pmatrix}, \quad \rho_2 = \begin{pmatrix} 1\ 2\ 3 \\ 1\ 3\ 2 \end{pmatrix}, \quad \rho_3 = \begin{pmatrix} 1\ 2\ 3 \\ 2\ 1\ 3 \end{pmatrix}, \quad \rho_4 = \begin{pmatrix} 1\ 2\ 3 \\ 2\ 3\ 1 \end{pmatrix}, \quad \rho_5 = \begin{pmatrix} 1\ 2\ 3 \\ 3\ 1\ 2 \end{pmatrix}, \quad \rho_6 = \begin{pmatrix} 1\ 2\ 3 \\ 3\ 2\ 1 \end{pmatrix}$$

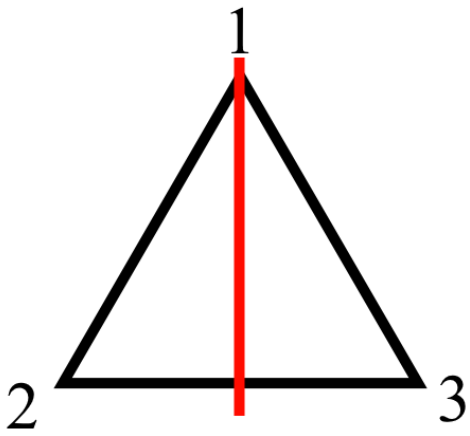
Disse 6 permutationer kan også beskrive alle transformationer af en ligesidet trekant. I den sammenhæng er det trekantens spidser, som permutationerne omrokerer (McQuarrie). Trekantens spidser kan nummereres således:



Figur 5 det neutrale element af S_3

Permutationen ρ_2 beskriver en vandret spejlvending af trekanten:

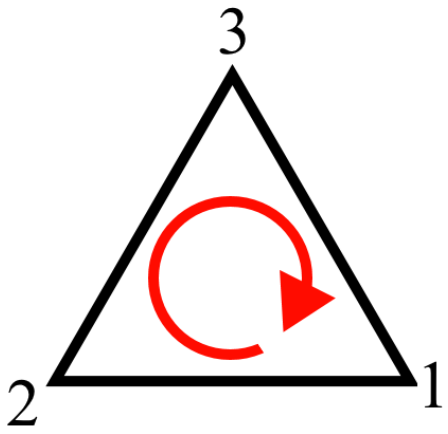
$$\rho_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$$



Figur 6 spejling af trekanten på y-aksen

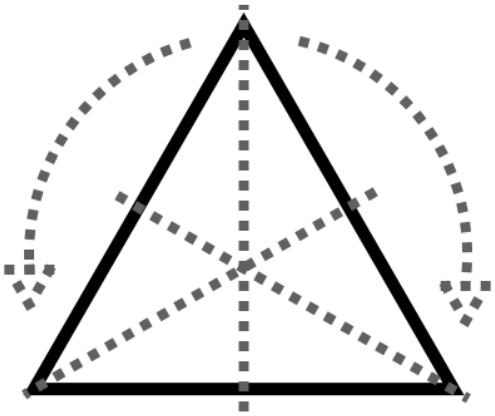
Ligeledes beskriver ρ_5 en rotation af trekanten med uret:

$$\rho_5 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$



Figur 7 rotation af trekanten med uret

Og så videre. de 6 permutationer er: rotation med uret, rotation mod uret, samt spejlvending på hver at de tre akser. Der er også en permutation, ρ_1, der ikke ændrer noget.



Figur 8 Alle transformationer af trekanten

Man kan da forestille sig at lave flere permutationer efter hinanden. Permutationsgruppens operator kombinerer to permutationer, en sålaldt permutationskomposition, således at produktet svarer til at lave de to permutationer efter hinanden (Chen, 2004). Operatorens symbol er $\circ$. Når man laver flere permutationer efter hinanden, betyder det, at man udfører den ene permutation først, og derefter udfører den anden permutation på resultatet fra den første permutation. Der tages udgangspunkt i permutationerne fra før:

$$\rho_2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

$$\rho_5 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$$

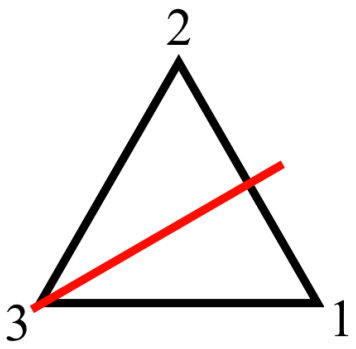
En permutationskomposition af ρ_2 og ρ_5 skrives da som:

$$\rho_2 \circ \rho_5 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \tag{23}$$

Får at finde hvilken permutation $\rho_2 \circ \rho_5$ svarer til, kan man følge elementerne én ad gangen. I ρ_2 går 1 til 3, og i ρ_5 går 3 til 2, så $\rho_2 \circ \rho_5$ af 1 er da 2. Dette gøres for alle elementerne. Man ser da, at $\rho_2 \circ \rho_5$ bliver:

$$\rho_2 \circ \rho_5 \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$$

Dette svarer til en spejling af trekanten gennem 3:



Figur 9 spejlvendning gennem 3

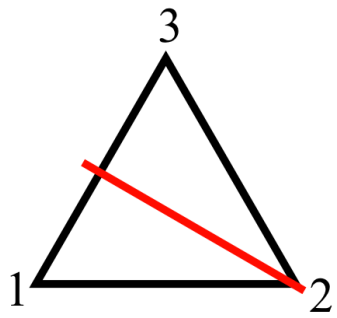
Dette er det samme som ρ_3.

Hvis man laver permutationerne i den omvendte rækkefølge, får man da i stedet:

$$\rho_5 \circ \rho_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \tag{24}$$

$$\rho_5 \circ \rho_2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Dette svarer til en spejling af trekanten gennem 2:



Figur 10 spejlvending gennem 2

Dette er det samme som ρ_6.

Man ser altså, at lige meget hvilke permutationer, man kombinerer med operatoren, ender man med en anden permutation, der også eksisterer i gruppen. ρ_1 er det neutrale element. Alle elementerne har også et inverst element. Spejlvendingerne er inverse til sig selv (to spejlvendinger på samme akse ende tilbage hvor man startede), og rotationen med uret er invers til rotaitonen mod uret. Dette overholder aksiomerne for en gruppe.

OBS: Man vil ofte se permutationskompositioner skrevet fra højre til venstre. I denne rapport skrives de fra venstre til højre, da det er standarden når det kommer til Rubik's terning og Singmasternotation (mere om dette senere). Det er vigtigt at være opmærksom på dette.

Undergrupper

Permutationerne i den symmetriske gruppe S_3 beskriver de forskellige transformationer af en ligesidet trekant. Man kunne da forestille sig, at S_4 ligeledes kunne beskrives med et kvadrat. S_4 har $4! = 24$ permutationer. Man ser dog, at der kun er 8 forskellige transformationer af kvadratet. Man kan rotere den hhv. 0°, 90°, 180° eller 270°, og man kan spejlvende den på 4 akser. Dette er kun 8 forskellige permutationer (Mathonline). Kvadratets transformationer er altså en undergruppe til S_4.

En undergruppe H til en gruppe G har den samme operator som G, og dens mængde indeholder de samme eller færre elementer end G. Dette skrives som:

$$H \subseteq G$$

Siden alle grupper har et neutralt element, og det neutrale element kan eksistere alene i sin egen gruppe, eksisterer den trivielle gruppe som undergruppe til alle andre grupper.

Rubik's terning som en permutationsgruppe

Terningen har 6 forskellige sider, som alle kan dreje uafhængig af hinanden. Midterne bevæger sig ikke i forhold til hinanden, når siderne drejes. De kan derfor bruges til at holde styr på, hvilken vej, terningen vender, samt hvilke sider, der drejes på.

Et 90° drej på én af siderne kan også ses som en handling, der tager nogle af farverne på terningen og flytter dem til en ny placering. På samme måde som transformationerne af trekanten og firkanten er permutationer, er hvert drej er altså en permutation af terningen.

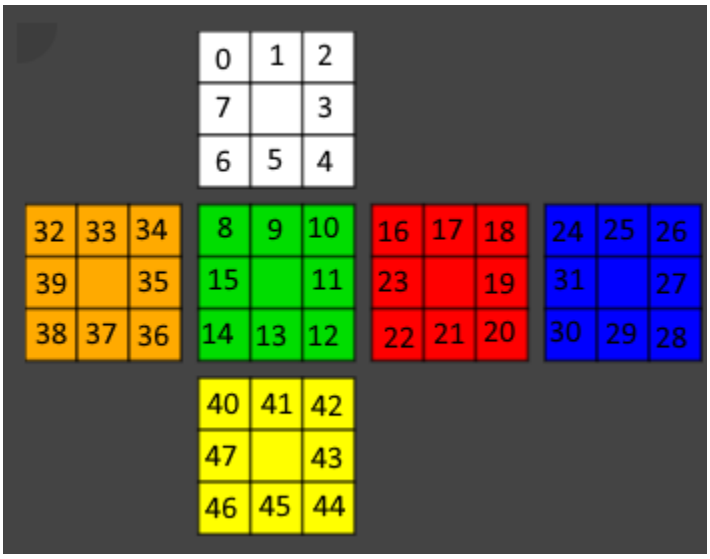
For at gøre beskrivelsen enklere, vil terningen altid vende med hvid opad og grøn hen mod brugeren. Dette er en standard opfundet af Tyson Mao (Speedsolving Wiki). Navngivningen, der benyttes til de forskellige drej, følger Singmasternotation. Singmaster giver hver side et bogstav:

bogstav	Side (engelsk)	Farve
U	Up	hvid
F	Front	grøn
R	Right	rød
B	Back	blå
L	Left	orange
D	Down	gul

Figur 11 Singmaster notation

Med denne notation kan de forskellige drej på terningen beskrives. Permutationen U svarer derfor til at dreje den hvide side 90° *med* uret, i forhold til sidens akse (Singmaster, 1981).

Terningen har 48 felter, der bevæger sig rundt (9 på hver side minus de 6 midter, der ikke bevæger sig). Hvert felt gives et tal, så de er lettere at holde styr på:



Figur 12 Alle felter af Rubik's terning nummereret. Tegning uden tal fra CSTimer.net

Hvis alle felterne kunne byttes frit rundt, ville permutationsgruppen, der holdt alle de mulige permutationer, være den symmetriske gruppe S_{48}. Dog er det ikke alle konfigurationer, der kan opnås ved kun at dreje de 6 sider. Terningens permutationsgruppe er derfor en undergruppe til den symmetriske gruppe S_{48}, og betegnes $(G,\circ)$ (Wikimedia Foundation, u.d.). Gruppens operator er derfor permutationskomposition.

Siden alle kompositioner af permutationer giver en anden permutation i gruppen, kan de mulige permutationer af Rubik's terning beskrives som en serie af drej på de 6 sider. Alle permutationer er altså en komposition af de 6 forskellige drej, og gruppens mængde kan noteres som:

$$G = \langle R, L, F, B, U, D \rangle \tag{25}$$

Dette vil sige, at $\langle R, L, F, B, U, D \rangle$ *frembringer* G. Alle elementerne i G kan findes som et produkt af de 6 elementer og deres invers.

Gruppen har selvfølgelig også et neutralt element, som er ikke at dreje nogen af siderne. Alle gruppens drej har også et inverst element. Hvert af de 6 drej drejer den tilsvarende side med uret, så det inverse element ville være at dreje det mod uret. For et vilkårligt drej X skrives dette som:

$$X \circ X^{-1} = e$$

Inverse permutationer skrives som X^{-1}, men ifølge Singmasternotation kan det også skrives som X' (Singmaster, 1981). Et 180° selvfølgeligvis til to 90° drej efter hinanden på samme side, og

skrives som X^2. Singmasternotation siger, at et dobbelt-drej som X^2 tæller som et enkelt drej. Med drejet X kan både X' og X^2 laves (X' er det samme som X^3), men med X^2 kan X og X' ikke laves.

Terningens løsningsmuligheder

Terningen har utrolig mange permutationer. Hver af disse har uendelige mange måder at komme tilbage til starten. Hvordan kan man have en tilgang, der kan løse en vilkårlig permutation hver gang? Selv opfinderen af puslespillet brugte en måned på at løse den første gang (Alter, 2020). Der er sidenhen blevet udviklet mange forskellige metoder til at løse terningen.

Menneskelige metoder

Når mennesker for første gang forsøger at løse Rubik's terning, prøver de ofte at løse én side ad gangen. Det er intuitivt, da man får sat flere ens farver sammen, og det virker som om, man laver fremskridt. Det er dog sjældent, at man kommer nogen vegne med det, fordi selv hvis alle farverne på én side er ens, kan brikkerne på den side stadig sidde forkert i forhold til hinanden, så kanten omkring siden ikke er løst. Dette vil gøre det umuligt at løse den næste side uden at lave den første side forfra igen.

Begyndermetoden

Figur 13 Hvert trin af begyndermetoden. Bemærk, at der er tydeligt, hvad der er løst.

Begyndermetoden er den metode, de fleste lærer, når de skal løse terningen for første gang. Metoden går ud på at løse ét "lag" af terningen ad gangen (SolveTheCube, 2015). Man starter med at få 4 kantbrikker omkring en midte til at være placeret rigtigt. Derefter løser man resten af "laget" ved at placere de 4 hjørner, på samme side som kanten, rigtigt. Man har da én side løst, samt kanten rundt om siden. Man kan se det som et "lag". Som det næste løser man de 4 kanter i det næste "lag" (denne gang består laget bare af 4 kantbrikker). Til sidst løser man det sidste lag. Der

er flere måder at gøre det på, men det gøres normalt i 4 trin. Begyndermetoden, ligesom de fleste andre metoder designet til mennesker, fokuserer meget på at placere brikker rigtigt, og så løse resten af brikkerne uden at ødelægge dét, man allerede har lavet. Dette er smart, fordi det er nemt at have overblik over, hvad man har løst, og hvor langt man er igennem. Begyndermetoden tager generelt over 80 drej, før terningen er løst, men det kan sagtens tage over 100 (Speedsolving Wiki).

En variation af denne metode hedder Fridrich metode (Fridrich, u.d.). Den er ligesom begynder-metoden, men løser de første to lag i ét trin, samt det sidste lag i to trin. I det værste tilfælde vil det tage 69 at løse terningen med Fridrich metode (Wang, 2020). I gennemsnit tager det 56 drej (Fridrich, u.d.). Hvis man er lige så heldig med blandingen, som Yusheng Du var, da han fik sin verdensrekordtid på 3.47 sekunder, vil det kun tage 27 drej (Speedsolving Wiki).

ZZ-metoden

Figur 14 terningen efter EOLine. Bemærk, at den forreste og bagerste kant på den gule side er løst

ZZ, opfundet af Zbigniew Zborowski (Speedsolving Wiki), har en unik tilgang til at løse terningen. Det første trin af ZZ hedder EOLine (Edge Orientation Line). Målet er at få alle kanter til at ”vende rigtigt”. En kant kan vende på to forskellige måder. Det kræver at drej på alle tre akser (dvs. U/D, F/B og L/R) for at kunne placere en kant i sin løste position mens den vender forkert. Ved at vælge to modstående lag, man ikke vil dreje, vil kanterne kun kunne løses med én

orientering, når de placeres på deres rigtige plads. Under EOLine får man alle kanter til at vende rigtigt således, at drej på F og B ikke længere er nødvendige for at løse terningen. Samtidig løser man DF- og DB-kanterne. Hermed behøves der ikke længere F, B og D drej til at løse terningen, da alle de uløste brikker kan påvirkes udelukkende med drej på L, U, og R. Resten af løsningen minder meget om Fridrich, bare udelukkende med L, U og R drej. Denne metode tager i gennemsnit 45 drej, hvilket er betydelig bedre end Fridrich.

Computeralgoritmer

Metoderne, som mennesker bruger, kan en computer også bruge. Der er en række regler og sekvenser af drej, man skal huske, for at kunne løse terningen. Disse regler og sekvenser kan en computer også godt følge. Computeren har dog ikke behov for at løse enkelte brikker ad gangen, der findes en anden meget mere effektiv tilgang. Princippet er trinvist at reducere terningens permutationsgruppe, indtil den er løst. præcis hvordan dette opnås, kommer senere. Indtil videre er det vigtige, hvilke trin, løsningsmetoden indebærer. Tidligere blev Rubik's ternings permutationer skrevet op som:

$$G = \langle R, L, F, B, U, D \rangle \tag{26}$$

Alle permutationer kan konstrueres som kombinationer af de 6 drej (som også selv er permutationer i mængden). Hvis man kan få terningen hen til en konfiguration, hvor der er nogle typer af drej (f.eks. drej på en specifik side), der ikke længere er behov for, kan resten af løsningen findes med en kombination af færre forskellige drej. Dette er hvad ZZ-metoden gør i EOLine. Som sagt reducerer den terningen til kun at bruge R, U og L drej. Denne gruppe kan skrives som:

$$G_{EOLine} = \langle R, U, L \rangle \tag{27}$$

Hvis man vil finde ud af, hvor mange forskellige permutationer, der eksisterer i denne gruppe, kan man tage udgangspunkt i beregningen for alle konfigurationen på terningen. Beregningen for antallet af mulige konfigurationer af kanter var $(12! \cdot 2^{12})$. $12!$ er alle de forskellige placeringer, kanterne til sammen kan have, og 2^{12} er alle de måder, kanterne tilsammen kan vende på. Siden to kanter skal være løst og derfor kun kan have 1 placering hver, er det kun de resterende 10 kanter, der kan være placeret tilfældigt. Der skrives altså $10!$ i stedet for $12!$. Siden EOLine sørger for, at alle kanterne kun kan vende på én måde, skal der ikke længere ganges med 2^{12}. Desuden

skal der heller ikke ganges med $\frac{1}{2}$ længere, fordi der ikke kan være en enkel kant, der vender forkert (Hvis det var tilfældet, ville terningen ikke være reduceret til G_{EOLine}).

kan antallet af mulige konfigurationer beregnes altså således:

$$\frac{(8! \cdot 3^8) \cdot 10!}{3 \cdot 2} = 159\,993\,501\,696\,000 \approx 1{,}59994 \cdot 10^{14} \quad (28)$$

Terningen er altså blevet reduceret til en mindre undergruppe, der har færre permutationer end de $4{,}3 \cdot 10^{19}$, terningen normalt har. For at terningen løses, skal den til sidst reduceres til den trivielle gruppe e, der ikke består af nogle drej eller permutationer. Terningen er da løst. Dette er grundlaget, som de fleste computerløsninger bygger på.

Kociembas 2-fase algoritme

Herbert Kociemba's 2-fase algoritme består af 2 trin (antydet ved navnet). I den første fase hedder Domino reduktion. Terningen bliver reduceret til følgende gruppe (Kociemba):

$$G_{DR} = \langle U, D, R2, L2, F2, B2 \rangle \quad (29)$$

Navngivningen kommer af, at gruppen har de samme drej som en 3x3x2 terning, også kendt som Rubik's Domino. En 3x3x3 terning har dog stadig 4 kantbrikker mere, så det er ikke den samme mængde, de permuterer.

Figur 15 Rubik's Domino. Billede hentet fra Ruwix.com

Ved kun at lave disse drej på en terning, ser man, hvad der skal være opfyldt, for at terningen er reduceret til G_{DR}. Alle kanter og hjørner skal vende rigtigt, og alle kanterne, der ikke hører til i U eller D laget skal være i laget mellem de to. Antallet af konfigurationer kan beregnes således:

$$\frac{(8!) \cdot (4! \cdot 8!)}{2} = 19\,508\,428\,800 \approx 1{,}95084 \cdot 10^{10}$$

Denne reduktion har omtrent $2 \cdot 10^9$ færre konfigurationer end en normal terning.

Efter domino reduktion løses resten af terningen.

Kociembas 2-fase algoritme bruger i gennemsnit under 20 drej (Kociemba), og hvis den forsøger at finde forskellige løsninger på samme blanding, vil algoritmen altid kunne finde en optimal løsning. Dette er algoritmen, der blev brugt til at bevise, at alle konfigurationer kan løses med 20 drej eller færre (Rokicki, Kociemba, Davidsom, & Dethridge). Det er også den algoritme, der benyttes til at generere blandinger til officielle Rubik's terning konkurrencer.

Thistlethwaites algoritme

Figur 16 Hvert trin Thistlethwaite's metode. Billederne viser hhv. terningen i G0, G1, G2, G3 og den trivielle gruppe e. Bemærk, at det er svært at tyde, hvor tæt terningen er på at være løst I de forskellige trin.

Morwen B. Thistlethwaite opfandt i 1981 en computeralgoritme til at løse Rubik's terning (Thistlethwaite, 1981). Ligesom Kociembas algoritme tager metoden udgangspunkt i at reducere terningens permutationsgruppe. Thisethwaite adskiller sig ved at dele løsningen op i 4 trin i stedet for 2. Det første trin er EO (Edge Orientation). Her skal alle kanterne vende rigtigt. Det er ligesom EOLine fra ZZ, men de to kanter på bunden behøves ikke at løses. Efter EO er der ikke behov for F, F', B eller B' drej, og gruppen bliver da:

$$G_1 = \langle R, L, F2, B2, U, D \rangle$$

Det næste trin er Domino Reduktion. Dette trin er præcis det samme som i Kociemba, forskellen er bare, at kanterne allerede vender rigtigt, så det er én ting mindre, der skal tages højde for:

$$G_2 = \langle R2, L2, F2, B2, U, D \rangle$$

Den sidste reduktion er HTR (Half Turn Reduction). Her reduceres terningen således, at alle felter enten har den rigtige eller modsatte farve. Dette betyder, at terningen kan løses udelukkende med dobbelt-drej. Gruppen bliver altså:

$$G_3 = \langle R2, L2, F2, B2, U2, D2 \rangle$$

Herfra løses terningen helt. Følgende er et skema over, hvor mange permutationer, der er i hver gruppe. Forklaringer af udregningerne nedenfor:

gruppe		beregning	Antal permutationer
G_0	$\langle R, L, F, B, U, D \rangle$	$\frac{(8! \cdot 3^8) \cdot (12! \cdot 2^{12})}{3 \cdot 2 \cdot 2}$	$43\,252\,003\,274\,489\,856\,000 \approx 4{,}3252 \cdot 10^{19}$
G_1	$\langle R, L, F2, B2, U, D \rangle$	$\frac{(8! \cdot 3^8) \cdot (12!)}{3 \cdot 2}$	$21\,119\,142\,223\,872\,000 \approx 2{,}11191 \cdot 10^{16}$
G_2	$\langle R2, L2, F2, B2, U, D \rangle$	$\frac{(8!) \cdot (4! \cdot 8!)}{2}$	$19\,508\,428\,800 \approx 1{,}95084 \cdot 10^{10}$
G_3	$\langle R2, L2, F2, B2, U2, D2 \rangle$	$\frac{(4!^2) \cdot (4!^3)}{6 \cdot 2}$	$663\,552 \approx 6{,}64 \cdot 10^5$
e	$\langle\ \rangle$	1	1

Figur 17 Udregninger af antal permutationer i de forskellige grupper. Udregningerne er selv udledt, resultaterne kan også findes ved (Thistlethwaite, 1981)

For at forstå beregningerne skal man vide, hvilke restriktioner hver gruppe lægger på brikkernes bevægelighed. Nedenstående kan testes med en fysisk terning. Hvis man kun laver drejene, der er tilladte i en given gruppe, vil de følgende regler være sande for alle konfigurationer, man støder på. Figur 17 har et eksempel på en konfiguration for hvert trin af løsningen.

I G_1 gælder det samme som under EOLine med at alle kanter vender rigtigt. De to kanter i bunden skal dog ikke løses, og der ganges derfor ikke med 2^{12}, og der deles med 6 i stedet for 12.

Beregningen af G_2 er allerede beskrevet under Domino Reduction i Kociembas algoritme.

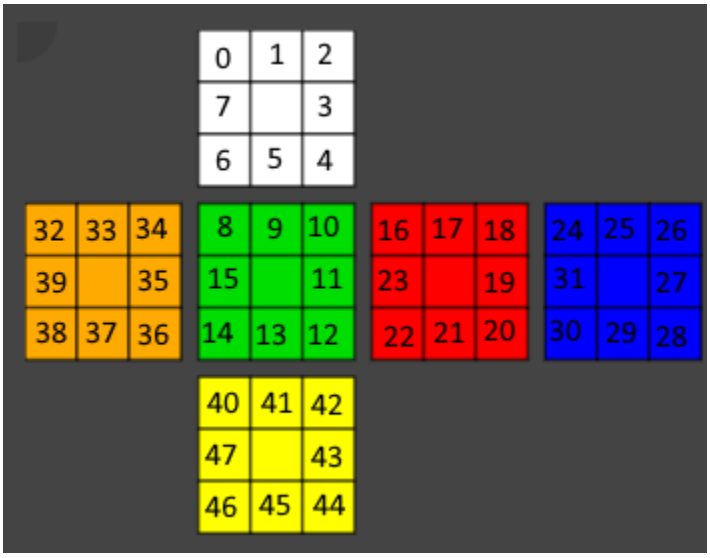
I G_3 er kun dobbelt-drej tilladt. Ligesom 4 kanter vil være placeret i samme lag i G_2, vil alle kanter nu være begrænset til én af de tre akser omkring terningen. Antallet af konfigurationer af kanterne er da $4! \cdot 4! \cdot 4! = 4!^3$. Ligeledes er hjørnerne begrænset 2 til grupperinger af 4 mulige placeringer hver, antallet af mulige hjørnekonfigurationer falder til $4!^2$. Her deles dog med 12, da der er 6 forskellige hjørnekonfigurationer, hvor alle hjørner kan løses enkeltvis, men kun 1 hvor alle kan løses sammen. To kanter kan stadig være byttede, så der deles med 2.

Løsning af Rubik's terning i et computerprogram

Gruppeteorien kan ikke alene løse en Rubik's terning. Den redegør for eksempel ikke for, hvordan man skal komme fra den ene gruppe til den næste. Programmet skal altså have en metode til at finde en sekvens af drej, der kan bringe terningen til den næste gruppe. Dette kræver selvfølgelig også, at man har programmeret en model af terningen. Programmet skal bestå af følgende dele:

- En model, der kan afbilde en konfiguration af terningen.
- En søgealgoritme, der kan finde en "vej" (en serie af drej), der kan få terningen til den næste gruppe og til sidst løst. Søgealgoritmen skal kun kunne lave de tilladte drej fra gruppen, som den søger i.
- En liste af drej, der er tilladte i hver gruppe samt en måde at undersøge, hvorvidt en given konfiguration kan løses under gruppen.

Model af terningen



Figur 18 Rubik's terning med alle felter nummereret Tegningen uden tal er fra CSTimer.net

Til mit program har jeg valgt at holde alle felterne samlet i et array med 48 elementer. Hvert felt har en farve, som jeg repræsenterer med en byte, der har en værdi mellem 0 og 5, som hver svarer til én af de 6 farver. Dette er den mængde, som permutationerne manipulerer. Tallene starter i ét af hjørnerne og går så med uret rundt om hver side. Dette gør det simpelt at flytte elementerne på samme side rundt, når den side drejes.

Permutationer

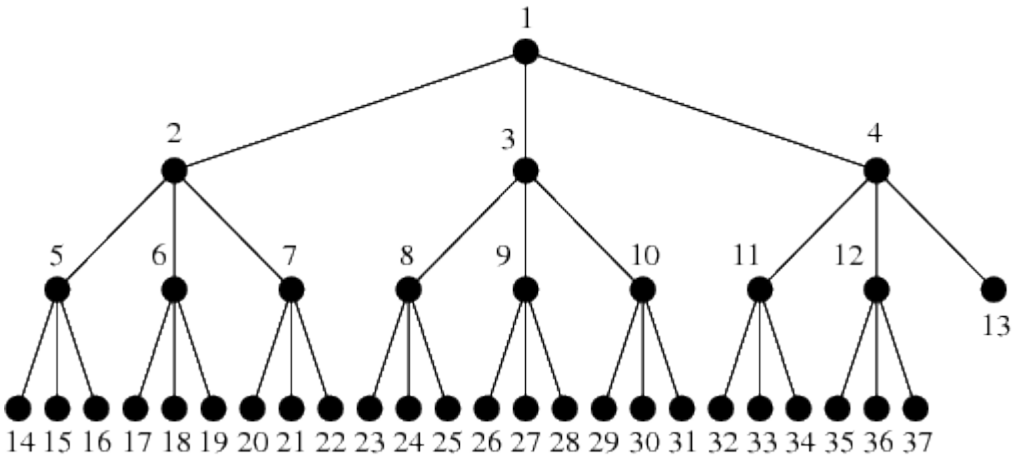
Siden modellen er et array af farver, har hver farve en placering i arrayet. Man kan derfor tage elementet fra én placering og flytte den til en anden placering. Det er præcis den handling, en permutationsfunktion beskriver.

Søgealgoritme

Søgealgoritmen er ”hjernen” af programmet. Den står for at kunne finde en sekvens af drej, der vil få terningen til en konfiguration, der kan løses med en mindre gruppe af unikke drej.

Tager man udgangspunkt i en tilfældig konfiguration på terningen, har man 18 forskellige drej, man kan lave. Der er 18, da der er 6 sider, der kan drejes, som også hver har et inverst og dobbelt-drej. Fra en vilkårlig konfiguration er der da 18 nye konfigurationer, man kan opnå. Vælger man at lave ét af drejene, ender man med en ny konfiguration hvor man igen har 18 drej at vælge imellem. Antallet af forskellige konfigurationer, man kan komme til efter n drej, stiger

eksponentielt. Man kan forestille sig, at måden de forskellige konfigurationer forgrener sig kan modelleres som et søgetræ:



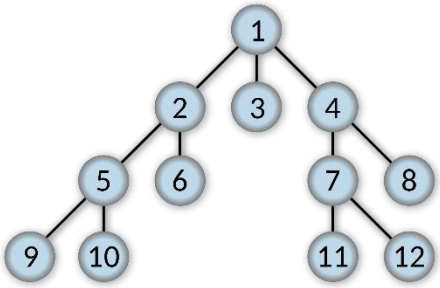
Figur 19 Eksempel på et søgetræ. I dette tilfælde er det et ternært søgetræ, hvilket betyder, at forgreningsfaktor er 3. Kilde: https://mathworld.wolfram.com/CompleteTernaryTree.html

Søgealgoritmen skal kunne finde en vej gennem dette søgetræ hen til en løsning, der kan reducere terningen til det næste trin.

Da jeg udviklede programmet, forsøgte jeg med flere forskellige søgealgoritmer, før jeg fandt én, der virkede ordentligt. Følgende er de søgealgoritmer, jeg forsøgte, og hvorfor de virker eller ej.

Breadth-First Search (BFS)

Breadth-first search, bredde-først søgning på dansk, er én af de simpleste søgealgoritmer, man kan forestille sig til at udforske et søgetræ. BFS besøger knudepunkter i den rækkefølge, de bliver opdaget. Dette opnås ved at have en kø af knudepunkter, der ikke er besøgt. Knudepunktet forrest i køen besøges, og nye knudepunkter tilføjes for enden af køen.



Figur 20 Tegning af rækkefølgen, BFS besøger knudepunkterne i. Kilde: https://en.wikipedia.org/wiki/Breadth-first_search

En bredde-først søgning vil altid finde den korteste rute til sit mål, og er derfor oplagt at bruge til at finde den korteste løsning. Algoritmens store ulempe er til gengæld, at alle knudepunkter skal hele tiden holdes i hukommelsen. Dette gør ikke så meget på mindre søgetræer, men Rubik's terning har trillioner af konfigurationer, som ikke kan holdes i hukommelsen på én gang. Da jeg implementerede denne algoritme, løb jeg ofte tør for hukommelse.

A* (AStar) algoritmen

For at undgå at besøge alle knudepunkter og løbe tør for plads, forsøgte jeg at implementere A*, som er en heuristik-baseret søgealgoritme. A* giver hvert knudepunkt en værdi. Værdien er afstanden til startnoden og den estimerede afstand til slutnoden, lagt sammen.

$$f = g + h \tag{30}$$

A* vil besøge det knudepunkt, der har den mindste f værdi. A* kan derfor ofte tage en lige vej til målet, uden at have behov for at forgrene sig. Dette virker som en oplagt algoritme at benytte, men der er et enkelt problem: h værdien. Heuristikken, A* bruger, er som sagt et estimat på, hvor mange drej, der er til målet. Dette er ikke nemt at gøre på Rubik's terning.

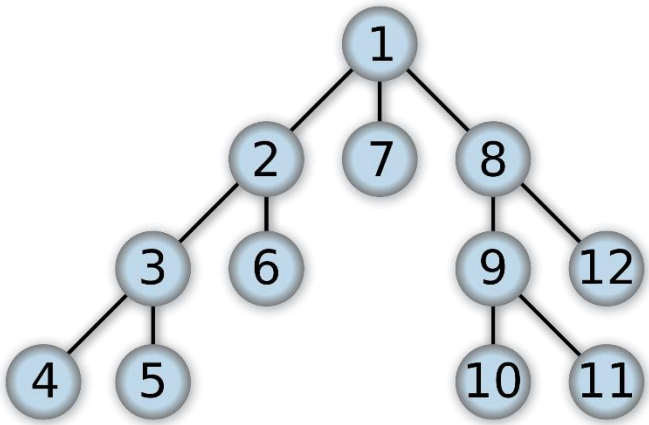
Man kunne estimere, hvor mange træk hver brik er væk fra at være løst, og dele med antallet af brikker, der flyttes per træk. Dette kaldes Manhattan-metrik (Blair, 2018). Da jeg forsøgte at implementere dette opdagede jeg dog, at det højeste estimat, den kunne give, var, at terningen var 3 drej fra at blive løst. Dette var sjældent sandt, og den endte med at søge alt igennem ligesom BFS.

En anden løsning, og den metode Morwen Thistlethwaite brugte, da han opfandt sin algoritme (Thistlethwaite, 1981), var at have en mønsterdatabase (patterndatabase, PDB), der var genereret i forvejen (Blair, 2018). Mønsterdatabasen holder f.eks. alle konfigurationer af hjørner. Når A* skulle bruge et estimat på, kunne den bare slå op i databasen (dette ville ikke altid give et perfekt estimat, men det ville være tæt på). Det ville til gengæld kræve, at man først konstruerede databasen, hvilket i sig selv er en udfordring. Jeg valgte derfor at finde en anden søgealgoritme.

Depth-First Search (DFS)

Hvor bredde-først besøger knudepunkter i rækkefølgen, den opdager dem, besøger dybde-først det seneste knudepunkt, den har fundet. Dette opnås ved at bruge en stak i stedet for en kø. Nye

knudepunkter tilføjes til toppen af stakken, og det øverste knudepunkt er det næste, algoritmen besøger. På den måde vil den besøge det seneste knudepunkt, den har besøgt. Den forgrener sig derfor først, når et knudepunkt, den besøger, ikke har nogen naboer. Fordelen ved DFS er, at den ikke behøver holde alle elementerne i hukommelsen. Dens plads-kompleksitet er derfor $O(d)$ (Wikimedia Foundation). Problemet ved at bruge DFS i dette tilfælde er, at man altid kan lave drej på en terning, og alle knudepunkter har derfor naboer. Den ville derfor aldrig forgrene sig og (højst sandsynligt) aldrig finde en løsning.



Figur 21 Tegning af rækkefølgen, BFS besøger knudepunkterne i. Kilde: https://en.wikipedia.org/wiki/Depth-first_search

Iterative Deepening Depth-First Search (IDDFS)

IDDFS bygger på dybde-først søgning (DFS). Det er en dybde-først søgning, der har en dybde-grænse. Når DFS når til et knudepunkt ved dybdegrænsen, vil den ikke undersøge knudepunktets naboer. Når alle knudepunkter i en given dybde er fundet, stiger dybden med 1. Algoritmen vil da først besøge alle knudepunkter i dybden 0, så dybden 1, så 2, osv. IDDFS finder derfor den korteste vej, ligesom BFS, men skal kun holde en lille del af træstrukturen i hukommelsen, ligesom DFS. Ulempen er til gengæld, at der er mange knudepunkter, der bliver besøgt flere gange. Siden den ikke husker noget fra forrige iteration, bliver alle knudepunkter i starten besøgt hver evig eneste gang. Dette gør den langsommere end BFS.

Grundet IDDFS evne til at finde den optimale løsning uden at bruge al den tilgængelige hukommelse, er det den algoritme, jeg valgte at implementere i mit program. Følgende er pseudokode for IDDFS-algoritmen:

```
// iterative deepening depth-first search
function IDDFS(root):
    for depth from 0 to infinity:
        hasFoundGoal, node ← DLS(root, depth)
            if hasFoundGoal is true:
                return node

    // return null if the goal wasn't found.
    // this won't every run in an infinite search space,
    // because DLS won't unwind before a goal is found
    return null


// depth limited search using DFS
function DLS(node, depth):
    // checks if the node is the goal node only if depth is 0.
    // nodes higher depth values have been checked in previous iterations of IDDFS
    if depth is 0:
        if node is goal:
            return true, node
        else:
            return false, null

    // if depth > 0, continue searching
    else if depth greater than 0:
        // check every branch
        foreach childNode of node:
            hasFoundGoal, node ← DLS(childNode, depth-1)
            if hasFoundGoal:
                return true, node
            else:
                return false, null
```

Figur 22 Pseudokode for IDDFS I et uendeligt søgerum. Modificeret fra Wikipedia (Wikimedia Foundation). for bedre at tage højde for et uendeligt søgerum

IDDFS starter på dybden 0 og kører DFS til en bestemt dybde (Depth-Limited Search) på det første knudepunkt (node). Hvis DFS ikke finder målet, stiger dybden med 1, og DLS køres igen med den nye dybde.

DLS er en rekursionsfunktion, der kalder sig selv, for hvert knudepunkt, den opdager. Den trækker 1 fra dybden, hver gang den kalder sig selv, indtil den bliver kaldt med dybden 0. Når DLS bliver kaldt med dybden 0, skal den ikke opdage nye knudepunkter, Den skal i stedet undersøge, om knudepunktet er målet. Hvis det er, vil den sende den fundne tilbage til det forrige kald, som så gør det samme, osv. Dette afvikler rekursionen og vil returnere målet til IDDFS.

I konteksten af programmet er målet en konfiguration, som kan løses udelukkende med drejene fra en mindre gruppe.

Implementering af løsningsalgoritme

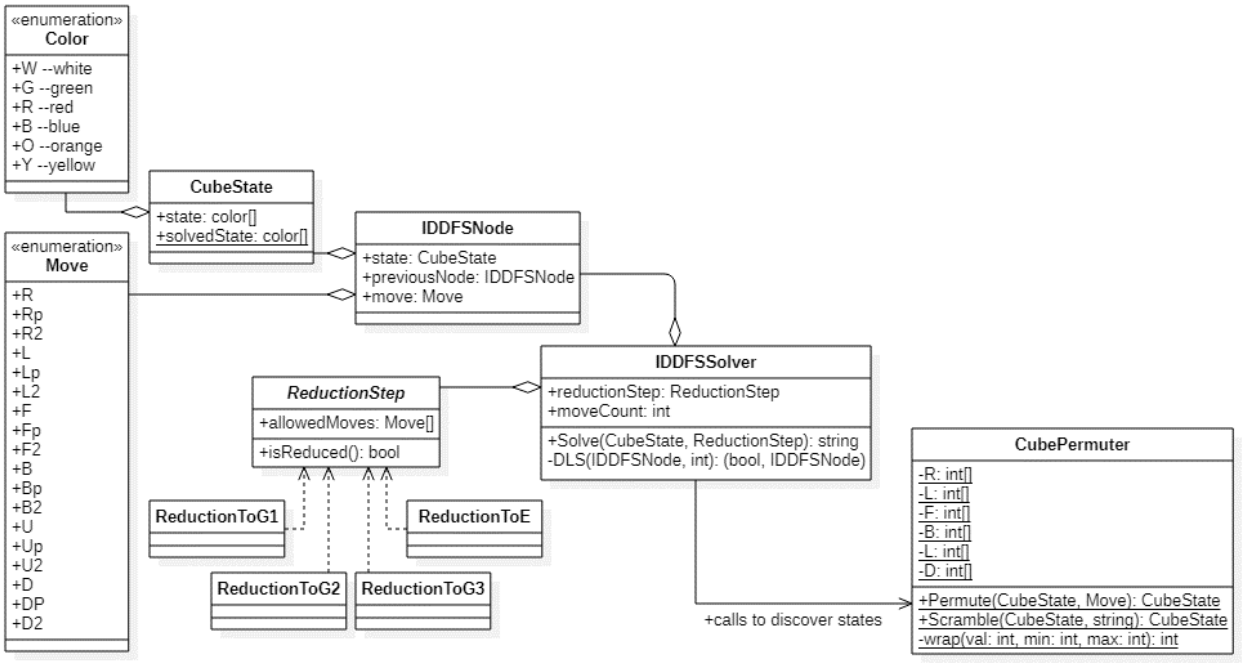
Der er nu en model af terningen, en måde at permutere denne model, samt en søgealgoritme til at finde en sekvens af drej, der løser terningen. Dette er hovedelementerne, der er nødvendige for at skrive et program, der kan løse terningen. Følgende er dokumentation på programmets komponenter og hvordan det fungerer.

Programmeringssprog

Jeg har valgt at skrive programmet i C#. C# er objekt-orienteret, hvilket gør de forskellige dele af programmet nemmere at dele op og holde styr på. Det er desuden det sprog, jeg er mest bekendt med. Objekt-orienterede sprog gør komplekse datastrukturer nemmere at arbejde med, men man risikerer også at få ellers unødvendige klasser, der ikke indeholder data og udelukkende holder funktioner.

Klassediagram

Følgende er et klassediagram over de vigtigste klasser i programmet (der er enkelte klasser og enums undladt, da de ikke har den store betydning for forståelsen af programmet):



Figur 23 Klassediagram over programmet

CubeState er en klasse, der holder modellen af terningen.

IDDFSNode er en repræsentation af knudepunkterne. Den har en CubeState samt hvilket drej, der sidst blev lavet for at nå dens CubeState. Den har også en reference til det forrige knudepunkt. Når rekursionen afvikles, kan kæden af IDDFSNodes følges til at finde sekvensen af drej, der er blevet brugt til at finde løsningen.

ReductionStep holder en liste af de drej, der er tilladte i en given gruppe, samt en funktion (isReduced), der ud fra en CubeState kan fortælle, om terningen er blevet reduceret til det næste trin. ReductionStep er en abstract klasse. Der kan ikke laves instanser af den, men kun klasser, der arver fra den.

IDDFSSolver har en offentlig Solve funktion, der tager en CubeState og et ReductionStep, og benytter IDDFS-algoritmen, til at finde en sekvens af drej, der kan bringe den originale konfiguration til en konfiguration, der er reduceret til næste trin. Den har også en privat DLS-funktion, som er en rekursionsfunktion, der laver Depth-Limited Search.

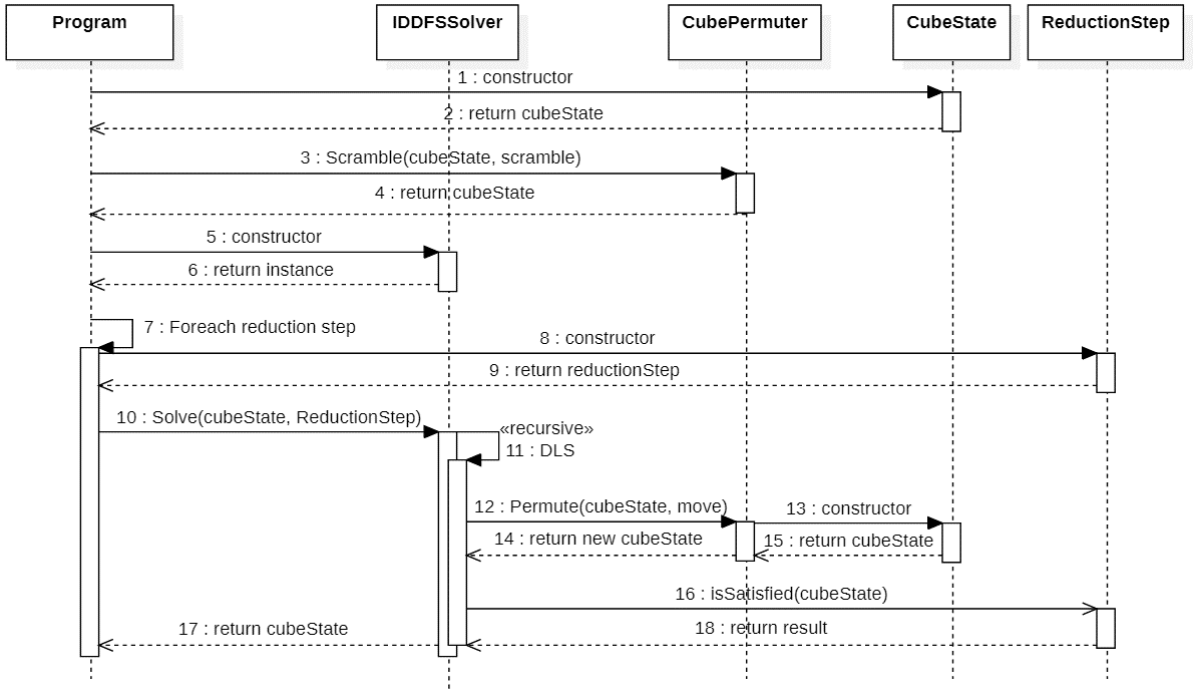
CubePermuter er en statisk klasse, hvis funktion er at permutere terningen. For hver af de 6 sider har den en liste af tal. Disse tal svarer til placeringen af farverne på siden af laget på terningen. Hvert tal er altså et index i CubeState's array. For at permutere terningen skal alle farverne på de givne pladser flyttes til indekset 3 elementer senere i CubePermuter's liste, siden et drej flytter 3 farver fra hver side. Sådanne en liste behøves ikke til at kunne flytte farverne på selve siden, da de i modellen af terningen er i rækkefølge (de 8 felter på U-siden, den hvide side, har indeksene 0-7, F-siden, den grønne side, har 8-15, osv.). Farverne kan altså bare forskydes.

```
/// <summary>
/// Defines the position of the colors around the F face
/// </summary>
private static int[] F = new int[12]
{
    16, 23, 22,
    42, 41, 40,
    36, 35, 34,
    6, 5, 4

};
```

Figur 24 listen for et drej på F i CubePermuter. Tallene svarer placeringerne rundt om F-laget (den grønne side). (se Figur 19)

Sekvensdiagram



Figur 25 Sekvensdiagram over programmet

Dette er et sekvensdiagram over hovedelementerne i programmet. Programmet starter med at lave en instans af CubeState og så blande den baseret på brugerinput. Den laver så en instans af IDDFSSolver. For hvert af de 4 trin i løsningen laver den så en instans af det tilsvarende ReductionStep og bruger den som argument sammen med cubeState til IDDFSSolver.solve(). Solveren kalder sin private DLS-funktion, der så rekursivt forsøger de forskellige drej, indtil den finder en CubeState, der er løst ifølge det givne ReductionStep.

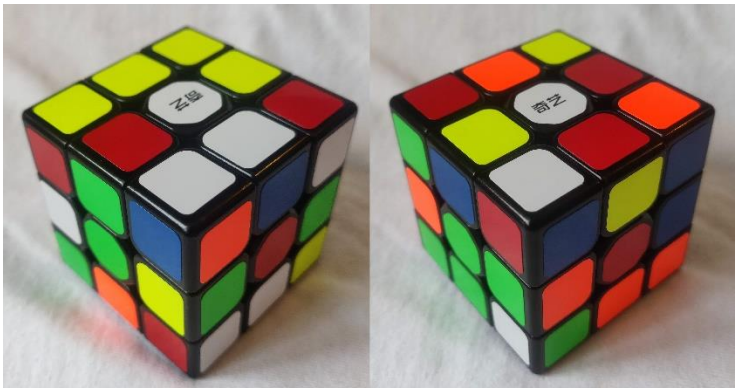
Resultater

På mit system (AMD Ryzen 7 8-core 3GHz) kan programmet søge igennem 1-4 millioner permutationer i sekundet, afhængig af, hvilket trin, den er i gang med. Det varierer dog drastisk, hvor lang tid, programmet bruger på at finde løsningen. F.eks. tog denne blanding under 10 sekunder at løse:

$$B2\ D\ U'\ F2\ U'\ B2\ U'\ L\ B2\ F'\ L\ F\ D'\ B2\ D\ B'\ D\ L2\ B'\ U$$

mens denne tog over 5 minutter:

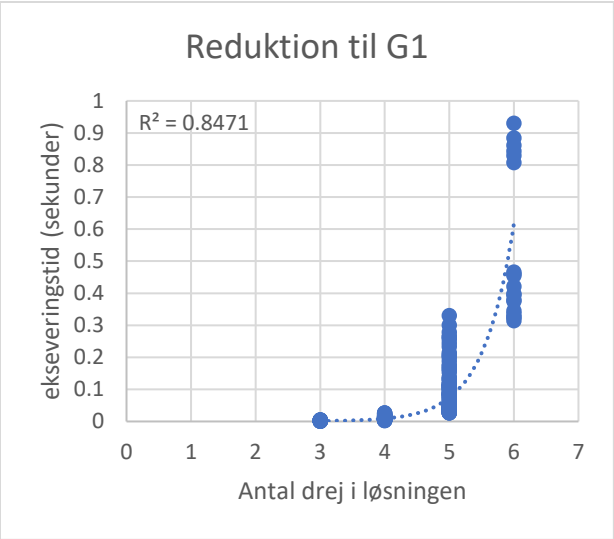
$$B2\ D\ U\ F2\ L2\ U'\ L2\ F2\ L\ D\ R2\ F2\ L'\ D'\ F2\ L\ U2\ B\ R\ U2$$

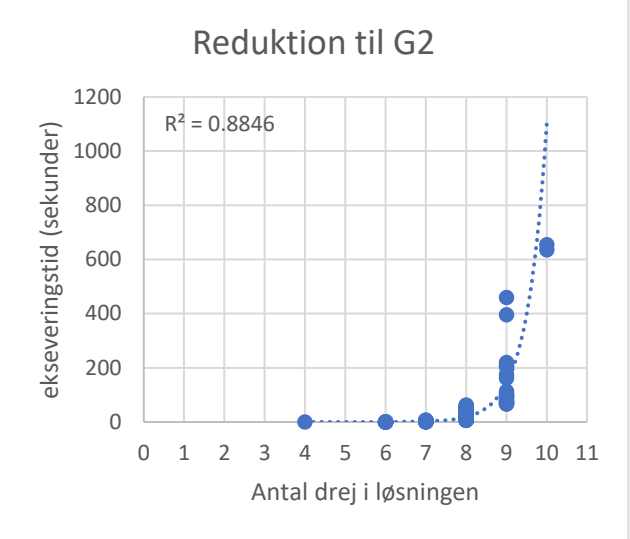


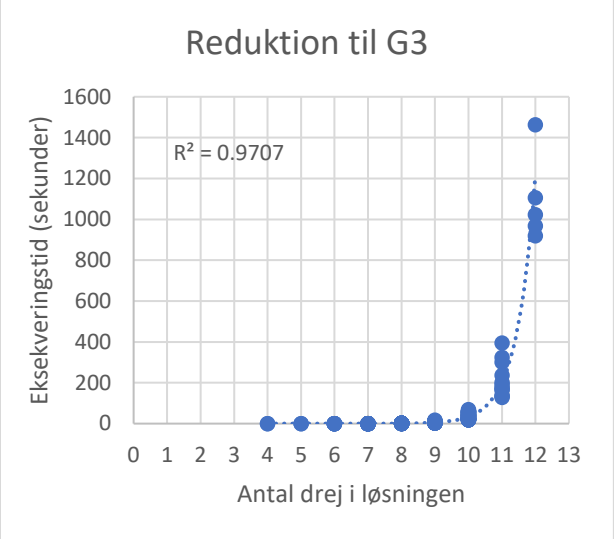
Figur 26 De to blandinger. Den første til venstre, den anden til højre.

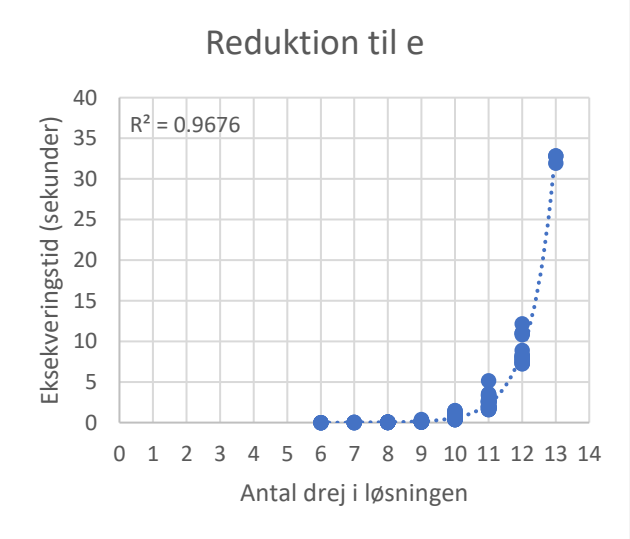
Det er stort set umuligt at sige, hvor mange drej, det vil kræve at løse terningen, eller hvor lang tid, det vil tage at finde den løsning, bare ved at kigge på den blandede terning.

Den store variation i eksekveringstiden skyldes, at IDDFS tager eksponentielt længere tid for at nå til den næste dybde. Den har kompleksiteten $O(b^d)$ hvor b er forgreningsfaktoren og d er dybden (Wikimedia Foundation). For at eftervise dette kørte jeg mit program over 170 gange på tilfældige konfigurationer og plottede eksekveringstiden for hvert af de 4 trin afhængig af antallet af drej i løsningen, altså dybden:



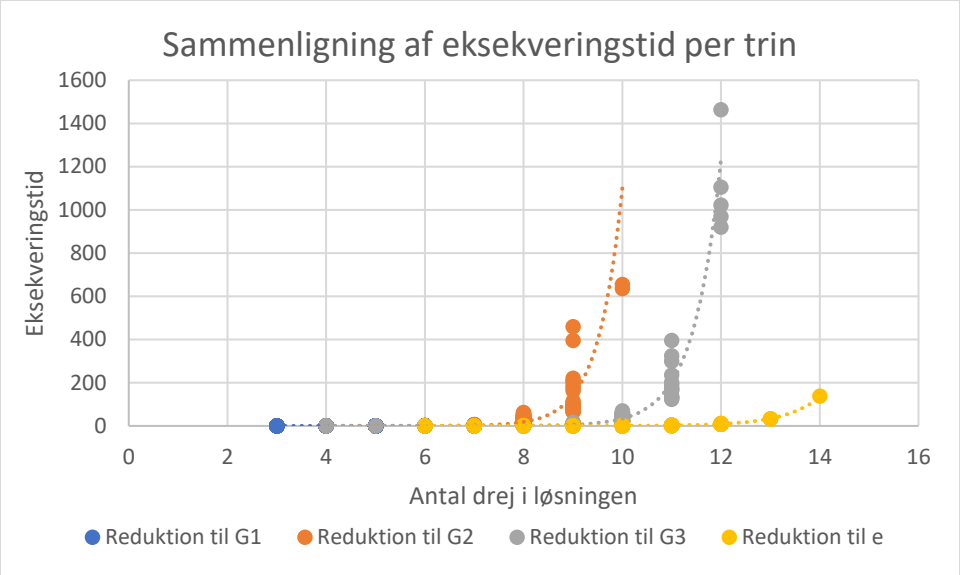






Der er en meget god overensstemmelse. Der er en del afvigelse ved det første trin. Det tager dog under et sekund at løse, så det gør ikke en stor forskel.

Her er en sammenligning mellem hvert trin:



Man kan se, at hældningen er lavere for hvert trin, hvilket giver mening, da hvert trin skal søge færre træk igennem per knudepunkt. Forgreningsfaktoren er altså faldende. Trin 1 (reduktion til G1) tager under et sekund på næsten alle bladninger. Dette skyldes, at alle løsningerne er meget korte, nemlig altid 7 træk eller færre (Thistlethwaite, 1981). Trin 4 (reduktion til e) tager også meget kort tid. Dette skyldes den meget lave forgreningsfaktor på kun 6 forskellige træk.

Trin 3 generelt længere tid end trin 2. Selvom forgreningsfaktoren er mindre i trin 3, kræver det ofte flere træk at løse (Thistlethwaite, 1981).

Optimering

Siden programmet søger flere millioner permutationer i sekundet, kan hver eneste optimering gøre en stor forskel. Mit program har meget plads til at blive optimeret. På baggrund af data ville det bedste sted at starte nok være hvordan programmet checker, om den har fundet en løsning, da det er det langsomste trin.

Det er også tydeligt, at der er andre måder at optimere programmet, når man sammenligner med andre implementeringer af Thistlethwaites algoritme. Et godt eksempel, og muligvis det hurtigste program, der anvender Thistlethwaites algoritme, er Ben Botto's "Rubik's Cube Cracker", som han har skrevet en Medium-artikel om (Botto, 2020). Botto's model af terningen er anderledes end min. Hans har en liste af 6 64-bit tal. Hver side af terningen kan holdes i ét 64-bit tal (det

giver 8 bits til hver farve på siden). Han kan da bit-shifte dataen for at ”dreje” siderne, hvilket er betydelig hurtigere end at flytte dele af en liste, som jeg gør i mit program.

Man kunne også lave en ny model af terningen for hvert trin af løsningen. F.eks. er hjørnerne af terningen fuldstændig ligegyldige under det første trin af Thistlethswaites algoritme, Edge Orientation. Man kunne lave en model, der kun har de 12 kanter, og hvor vidt de vender rigtigt eller ej. Dette kunne f.eks. være en 12-element lang boolean array. Det ville nok være muligt at optimere alle dele af programmet for hvert trin af løsningen.

En anden optimering kunne være at forsøge med en helt anden algoritme. Hvis jeg havde valgt at konstruere en mønsterdatabase til hvert trin i Thistlethwaites algoritme, kunne jeg have brugt A* og på den måde søge igennem betydelig færre konfigurationer.

Mennesker og maskiner

Når det kommer til komplekse problemer, f.eks. Rubik’s terning, har mennesker og computere forskellige forudsætninger, som gør det muligt at gribe problemet an på forskellige måder, men som også holder dem tilbage på forskellige måder.

Mennesker kan med fornuft og logik udtænke løsninger til problemer. Det er dét, der gør os til fornuftsvæsner. Det gør det muligt for os at udtænke løsninger til komplicerede problemer og forstå komplekse sammenhænge.

Menneskers store svagheder er deres upræcished og begrænsede kapacitet. Mennesker kan ikke rumme særlig meget information ad gangen (Miller, 1956). Når man kigger på Rubik’s terning og sammenligner menneskelige metoder med computermetoder, ser man igen, at de menneskelige metoder gør det muligt at filtrere nogle brikker fra og kun fokusere på en lille del ad gangen. Kigger man f.eks. på Fridrich metode, kigger man i det første trin kun på 4 kanter. I de sidste to trin kigger man også kun på 8 brikker samlet på én side.

Computeralgoritmerne tager højde for de fleste eller alle brikkerne på én gang. Dette gøres muligt af deres perfekte hukommelse og hurtigere beregninger. Mit program kan analysere flere millioner konfigurationer i sekundet. Computere kan altså holde styr på mange flere ting ad gangen, samt lave mange flere beregninger end mennesker.

Computerens svaghed er, at den ikke kan ”tænke” på samme måder som mennesker. Den kan kun gøre præcis hvad den får at vide i form af et program. Det vil sige at den er holdt tilbage af programmerne, de kører. Hvis programmerne ikke er optimeret, eller hvis algoritmerne, der er implementeret, ikke er effektive til at løse det givne problem, hindrer det, hvor effektivt den kan løse problemet. Dette kunne jeg selv se, da jeg forsøgte forskellige søgealgoritmer til at finde en løsning til Rubik’s terning. Computeren er altså holdt tilbage af menneskers innovation og evne til at skrive gode programmer. Det kræver også, at mennesker i forvejen har udtænkt en løsning til problemet.

Computeren kan altså ikke løse problemer selv, og dens rolle er altså et redskab til at hjælpe mennesker med at løse problemer.

Konklusion

I projektet er der ved brug af matematisk modellering lavet en model af Rubik’s terning i form af en permutationsgruppe, $(G,\circ)$ som undergruppe til den symmetriske gruppe S_{48}. Projektet sammenligner menneskelige metoder til at løse terningen til computeralgoritmer. De menneskelige løsningsmetoder drejer sig om at få enkelte brikker på plads, da mennesker ikke kan rumme at holde styr på hele terningen på én gang. Computeralgoritmerne fokuserer derimod på at reducere terningen til mindre undergrupper af G.
Gennem den iterative proces er der blevet udviklet et computerprogram, der kan løse Rubik’s terning ved brug af Thistlethwaites algoritme, en computeralgoritme, der reducerer terningens permutationsgruppe 4 gange. Gennem den iterative proces er der eksperimenteret med flere forskellige søgealgoritmer, og programmet bruger IDDFS (Iterative Deepening Depth-First Search) til at finde en løsning til hver af de 4 trin. Der er gjort brug af statistisk analyse til at evaluere programmets ydeevne ved hver af de 4 trin. På baggrund af dette kan det ses, at der vil være mest tid, der kan spares ved trin 3 i Thistlethwaites algoritme.
Til sidst blev mennesker og computeres styrker og svagheder i forhold til komplekse problemer diskuteret. Selvom mennesker i sig selv er begrænsede i deres udførsel af løsninger, har de evnen til at udtænke dem, noget som en computer ikke kan. Computeren er et redskab, der kan bruges til at løse problemer, og kan altså ikke gøre det selv.

Kilder

Alter, A. (16. September 2020). He Invented the Rubik's Cube. He's Still Learning From It. *The New York Times*. Hentet fra https://www.nytimes.com/2020/09/16/books/erno-rubik-rubiks-cube-inventor-cubed.html

Bertram, A. (u.d.). The Integers and Rational Numbers. Hentet fra https://www.math.utah.edu/~bertram/courses/4030/Ints.pdf

Blair, A. (7. marts 2018). Finding Admissible Heuristics - Artificial Intelligence. Hentet fra https://www.youtube.com/watch?v=FxLJ0vfMFHQ

Botto, B. (9. maj 2020). *Implementing an Optimal Rubik's Cube Solver using Korf's Algorithm.* Hentet fra Medium: https://medium.com/@benjamin.botto/implementing-an-optimal-rubiks-cube-solver-using-korf-s-algorithm-bf750b332cf9

Chen, J. (2004). Group Theory and the Rubik's Cube. Hentet fra http://people.math.harvard.edu/~jjchen/docs/Group%20Theory%20and%20the%20Rubik%27s%20Cube.pdf

Dalen, D. V., Doets, H. C., & swart, H. D. (9. maj 2014). Naïve Set Theory. Hentet fra https://books.google.dk/books?id=PfbiBQAAQBAJ&pg=PA1

Davis, T. (6. December 2006). Group Theory via Rubik's Cube. Hentet fra http://geometer.org/rubik/group.pdf

Fridrich, J. (u.d.). *My system for solving Rubik's cube*. Hentet fra http://www.ws.binghamton.edu/fridrich/system.html

Kociemba, H. (2017). *RubiksCube-TwophaseSolver*. Hentet fra GitHub: https://github.com/hkociemba/RubiksCube-TwophaseSolver

Kociemba, H. (u.d.). *The Two-Phase-Algorithm.* Hentet fra http://kociemba.org/cube.htm

Mathonline. (u.d.). *The Group of Symmetries of the Square.* Hentet 10. december 2020 fra Wikidot: http://mathonline.wikidot.com/the-group-of-symmetries-of-the-square

McQuarrie, B. (u.d.). Symmetry and Patterns: Groups. Hentet fra http://cda.mrs.umn.edu/~mcquarrb/teachingarchive/M1001/Resources/Chapter19Groups.pdf

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Harvard* . Hentet fra http://www2.psych.utoronto.ca/users/peterson/psy430s2001/Miller%20GA%20Magical%20Seven%20Psych%20Review%201955.pdf

Nipissing University. (u.d.). *Number Sets Tutorial.* Hentet fra Nipissing University: https://calculus.nipissingu.ca/tutorials/numbers.html

Rokicki, T., Kociemba, H., Davidsom, M., & Dethridge, J. (u.d.). *God's Number is 20.* Hentet fra cube20: https://cube20.org/

Singmaster, D. (1981). Notes On Rubik's Magic Cube. Hentet fra https://maths-people.anu.edu.au/~burkej/cube/singmaster.pdf

Smith, K. E. (2018). The Symmetric Group Sn. Hentet fra http://www.math.lsa.umich.edu/~kesmith/SymmetricGroup.pdf

SolveTheCube. (2015). *Beginner's Guide*. Hentet fra SolveTheCube: https://solvethecube.com

Speedsolving Wiki. (u.d.). *Layer by layer.* Hentet 10. december 2020 fra Speedsolving Wiki: https://www.speedsolving.com/wiki/index.php/Layer_by_layer

Speedsolving Wiki. (u.d.). *Tyson Mao.* Hentet 15. december 2020 fra https://www.speedsolving.com/wiki/index.php/Tyson_Mao

Speedsolving Wiki. (u.d.). *Yusheng Du.* Hentet 10. december 2020 fra Speedsolving Wiki: https://www.speedsolving.com/wiki/index.php/Yusheng_Du

Speedsolving Wiki. (u.d.). *ZZ method.* Hentet 10. december 2020 fra Speedsolving Wiki: https://www.speedsolving.com/wiki/index.php/ZZ_method

Thistlethwaite, M. D. (13. juli 1981). *Thistlethwaite's 52-move algorithm.* Hentet fra https://www.jaapsch.net/puzzles/thistle.htm

Wang, D. (2. december 2020). The Search For God's Number. Hentet fra https://youtu.be/YsWKrxAbopk?t=425

Wellens, J. (2015). A Friendly Introduction To Group Theory. *MIT*. Hentet fra http://math.mit.edu/~jwellens/Group%20Theory%20Forum.pdf

Wikimedia Foundation. (u.d.). *Iterative Deepening Depth-first Search.* Hentet 12. december 2020 fra Wikipedia: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Wikimedia Foundation. (u.d.). *Parity (Mathematics).* Hentet 15. december 2020 fra Wikipedia: https://en.wikipedia.org/wiki/Parity_(mathematics)

Wikimedia Foundation. (u.d.). *Rubik's Cube Group*. Hentet 10. december 2020 fra Wikipedia: https://en.wikipedia.org/wiki/Rubik%27s_Cube_group

Wikimeida Foundation. (u.d.). *Depth-first search.* Hentet 13. december 2020 fra Wikipedia: https://en.wikipedia.org/wiki/Depth-first_search