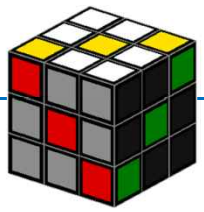


ROBOTER MIT FÄHIGKEIT, RUBIK'S CUBE ZU LÖSEN:

Konstruktion

Kornel Eggerschwiler
St. Ottilienstrasse 51
6018 Buttisholz

Programmierung

Manuel Schürch
Ed.-Huberstr. 20
6022 Grosswangen

betreut von:

Abdelhakim Ghezal
Oberdorfstrasse 22
6207 Nottwil

Vorwort

Vor kurzem war ich in einen Spielwarenladen gegangen mit der Absicht, einen 4x4 Rubik's Master Cube zu kaufen. Ich nahm ihn aus dem Regal und begab mich auf den Weg zur Kasse. Da sprach mich ein etwa sechs jähriger Junge erstaunt an: „Chasch du dä löse?" Ich erklärte ihm, dass ich es noch nie versucht hatte, dieser sei eine grössere Version und ich nur im Stande sei, die kleinere Version zu lösen. „Wie chasch du das?", wollte er von mir wissen, worauf ich ihm ganz kurz mein Vorgehen beim Lösen erklärte. Ich versuchte mich ganz simpel zu fassen. Ich wusste nicht, ob er Vieles verstand, er machte aber keinerlei Anstalten, sich mit diesen Ausführungen zufrieden zu geben. In seinen Augen sah man seine Faszination an diesem Würfel. „Mami, ech wott ou so eine!", rief er seiner Mutter durch den Laden zu. Der kleine Junge erblickte in meiner Tasche den 3x3 Rubik's Cube und setzte sich in den Kopf mich dazu zu drängen, dieser im Laden zu lösen. „Dosch dä einisch löse?", fragte er mich mit einem anmutigen Blick aus Begeisterung und kindlicher Unverfrorenheit, der verunmöglichte, ihm seinen Wunsch abzuschlagen. Als ich schlussendlich vor diesem Junge kurz den 3x3 Würfel gelöst hatte und seine Augen vor Begeisterung noch mehr glänzten, erlebte ich wieder einmal hautnah, wie solch ein kleines Spielzeug eine tiefe Faszination auslösen kann. Auch mir ging es einmal so. Ich war zwar um einiges älter als dieser Junge, als ich in einem Film den Protagonisten sah diesen Zauberwürfel zu lösen, und ich wusste, ich muss mir diesen Würfel kaufen. Kurz darauf beschäftigte mich dieser Würfel praktisch Tag und Nacht. Ich war in dieser Zeit vornehmlich damit beschäftigt, den Würfel möglichst schnell zu lösen.

Die Faszination an diesem Würfel rückte dann bei mir wieder vermehrter in den Hintergrund, doch als es darum ging, ein Thema für die Maturaarbeit zu finden, tauchte dieser kleine Gegenstand plötzlich wieder auf. Kornel Eggerschwiler und ich hatten in einem Informatikprojekt für das Ergänzungsfach einige simplere Versuche mit den Lego Roboter vornehmen wollen. Als wir über Möglichkeiten und Ideen diskutierten, stiessen wir auf den Rubik's Cube. Wir überlegten uns plötzlich, einen Roboter zu konstruieren, der fähig ist, diesen lösen zu können. Nach weiteren verifizierenden Schritten versuchten Kornel Eggerschwiler und ich dieses Projekt in Angriff zu nehmen.

An dieser Stelle möchten wir allen Personen danken, die zur erfolgreichen Durchführung dieses Projekts dazu beigetragen haben. Speziell möchte ich Manuela Fischer danken, die mich oft bei der Fehlersuche tatkräftig unterstützte.

Grosswangen, Oktober 2010

Inhalt

Einleitung

Nachdem Kornel Eggerschwiler und ich uns entschieden hatten, einen Roboter mit den Legobaukästen zu konstruieren, der im Stande ist, den 3x3 Rubik's Cube möglichst optimal zu lösen, teilten wir die Aufgabengebiete auf. Kornel war eher verantwortlich für die Mechanik, ich dagegen für die Strategie und Programmierung. Diese Aufteilung bedeutet allerdings nicht, dass wir getrennt voneinander arbeiteten, sondern es war eine Hilfestellung für uns, dass wir zweispurig arbeiten konnten.

In einem ersten Teil wird die Suche nach einer optimalen Zugfolge dokumentiert. Das Grundproblem dabei war die Quantifizierung des Würfels. Man muss die Unterschiede bzw. Gemeinsamkeiten der gut 43 Trillionen verschiedenen Stellungen analysieren, um damit arbeiten zu können. Sind diese Beschreibungen einmal getätigt worden, schrieben wir nach einer Idee von Herbert Kociemba einen Algorithmus, dessen Implementierung einen riesen Arbeitsaufwand bedeutete. Die Generierung von riesigen Datenmengen war ebenso ein spannender Teil unserer Arbeit, wie die Suche nach der eigentlichen Zugfolge. Als Abschluss dieses Teils entstand der *Cube Transformer*, ein kleines Programm, das jede beliebige Stellung des Würfels in eine andere beliebige Stellung in maximal 29 Zügen überführen kann.

Von der anderen Seite dem gemeinsamen Ziel näher gekommen sind wir mit der mechanischen Umsetzung des Lego Roboters. Dabei versuchten wir uns an mehreren Modellen und kämpften vor allem gegen die Instabilität der Legoteilchen an. Die Überlegungen, wie wir die Motoren dazu bringen können, die Seiten auf dem Würfel zu drehen, war nebst Ausprobieren von verschiedenen Kombinationen von Armen eine kreative Arbeit, bei der wir unzählige Ideen durchdiskutierten und die meisten wieder verworfen haben. Am Schluss konnten wir durch die Zusammenführung der beiden Bereiche und das Einlesen per Kamera ein Modell erstellen, das einen Rubik's Cube lösen kann. Als Produkte entstanden also einerseits das Programm Cube Transformer, anderseits ein funktionstüchtiges Modell.

Im Verlaufe der Dokumentation ist an manchen Stellen auf die beiliegende CD hingewiesen, um Zwischenschritte bei der Programmierung bzw. deren Ausführung ausprobiert werden können. Dazu verwenden Sie bitte das Programm Processing, welches ebenfalls auf der CD beiliegt.

Wir hoffen, dass ihr, geschätzte Leser und Leserinnen, die riesige Begeisterung, die das Arbeiten und Untersuchen am drehbaren Würfel bei uns geweckt hat, zu spüren bekommt und ein kleiner Funken auch auf euch rüber springt.

1 Beschreibung und Quantifizierung des Würfels

1.1 Ernö Rubik

Abbildung 1

Ernö Rubik, Bildhauer, Architekt und Designer, ist der Erfinder des nach ihm benannten Rubik's Cube, der oft auch Zauberwürfel genannt wird. Das Patent auf sein Geduldsspiel meldete er 1974 an und 1975 ging sein Spielzeug in die Massenproduktion. Dieser wurde auf der ganzen Welt verkauft und verbarg lange ungelöste Rätsel, das letzte wurde erst kürzlich gelöst. Es ist die Frage nach der höchsten Anzahl Züge, die man benötigt, um den Würfel aus jeder beliebigen Position zu lösen. Die Lösung wurde vom deutschen Mathematiker Herbert Kociemba und dem amerikanischen Informatiker Tomas Rokicki in Erfahrung gebracht und ist die „Gotteszahl“ 20. Um dies zu beweisen brauchte es enorme Rechenleistung, welche freundlicherweise von Sony Pictures und Google bereit gestellt wurde.

Der Rubik's Cube erfreut sich noch heute grosser Beliebtheit, es werden sogar Wettbewerbe in verschiedenen Disziplinen durchgeführt. Dabei ist das Ziel, den Würfel auf irgendeine spezielle Art zu lösen. Die herkömmliche Variante ist hierbei das normale Lösen mit beiden Händen, mittlerweile gibt es aber viele andere ausgefallene Disziplinen, wie zum Beispiel das Blindlösen, das Lösen mit nur einer Hand oder mit den Füssen. Unsere Disziplin ist hingegen das Lösen mit Einsatz von Computer und eines Lego-Roboters.

Im Verlaufe der Jahre wurden verschiedene Lösungsstrategien entwickelt, die es jedem erlauben, den Würfel zu lösen. Angefangen hat es mit einfachen Methoden, die zwar lange brauchen, aber einfacher zu verstehen sind. Mit der Zeit wurden immer aufwendigere Algorithmen entworfen, welche alle das Ziel haben, den Würfel in möglichst kurzer Zeit zu lösen, aber den Nachteil haben, dass sie schwerer nachzuvollziehen und deshalb schwierig oder nur mit hohem Aufwand für einen Menschen erlernbar sind.

Der erste und auch einfachste Lösungsweg wird heute der „Anfänger-Algorithmus“ genannt, jener wird auch allen empfohlen, die den Würfel zum ersten Mal lösen wollen. Bei dieser Variante wird der Würfel Ebene für Ebene gelöst. Mit dieser Methode sind aber keine Spitzenzeiten zu erwarten, denn die Speedcuber arbeiten mit komplexeren Algorithmen, meist mit dem Friedrich-Algorithmus. Dabei werden die ersten zwei Ebenen gleichzeitig gelöst. Der Zwei-Phasen-Algorithmus von Herbert Kociemba gehört in die Kategorie Algorithmen, die nur für Computer nutzbar sind. Der Algorithmusteil dieser Arbeit basiert auf den Ideen von Herrn Kociemba.

1.2 Aufbau

Rein die Mechanik und der Aufbau des Rubik's Cube zeugen schon von grosser Genialität. Jeder, der den Würfel schon Mal in den Händen hielt, fragte sich bestimmt, wie es möglich ist, dass man alle Ebenen drehen kann.

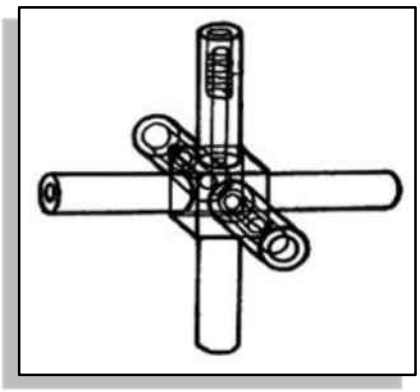


Abbildung 2: statische Achse

Wie man auf der *Abbildung 2* erkennen kann, gibt es drei fixe Achsen. Die Kanten- und Eckwürfel sind nicht fest miteinander verbunden, sondern lediglich zusammen gesteckt. Dadurch kann ein Stück an eine andere Seitenebene weiter gegeben werden. Die Fortsätze an jedem Kanten- und Eckwürfel führen dazu, dass sich im Würfel eine Art Kugel formt.

Der 3x3 Rubik's Cube besteht aus acht Ecken mit je drei farbigen sichtbaren Flächen, zwölf Kanten mit je zwei sichtbaren Flächen und sechs Mittelteile mit je einer unterschiedlichen Farbfläche. Durch diese Mittelteile wird eine Seite des ganzen Würfels auch repräsentiert, denn die Bezüge dieser Mittelteile untereinander verändern sich nie, sie sind statisch fixiert.

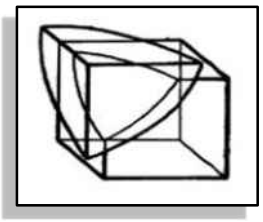


Abbildung 3: Ecke

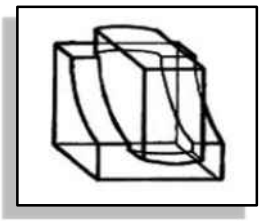


Abbildung 4: Kante

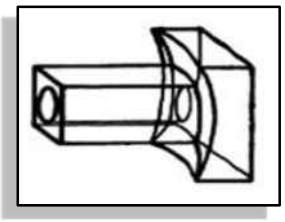


Abbildung 5: Mitte

Durch diesen Umstand der statischen Achse der Mittelteile kann man die sechs Seiten des Würfels immer identifizieren. Durch die Positionen der sechs Mittelteile wird die finale Position von jedem Einzelteil determiniert. Zum Beispiel die Eckposition zwischen dem grünen, gelben und roten Mittelteil ist für die Ecke mit der grünen, gelben und roten Fläche reserviert. Dito verhält es sich mit den finalen Positionen der Kanten, die durch die anliegenden Mittelteile bestimmt sind.

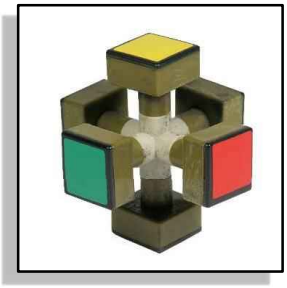


Abbildung 6

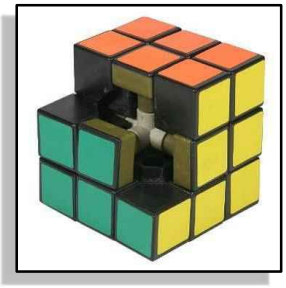


Abbildung 7

Um den Zustand eines Würfels zu beschreiben, wird eine Konvention über die Bezeichnungen benötigt. Eine Möglichkeit besteht darin, den Flächen den aktuellen Farbwert zuzuordnen. Bei einer zweiten Möglichkeit werden nicht mehr die Flächenpositionen beschrieben, sondern die einzelnen Würfelchen, also Ecken- und Kantenteile, als ein Objekt angesehen. Geht man noch ein Abstraktionsniveau höher, so kann man den Würfel mit einer Gesamtkoordinate beschreiben.

1.3 Flächenmodus

Die Flächen kann man entweder fortlaufend durchnummerieren oder mit einem Buchstaben der jeweiligen Seite versehen. Ich entschied mich für letztere Variante, da ich bei dieser den grösseren Überblick habe und Analogien für die einzelnen Seiten festzustellen sind, die mit dieser Art der Beschriftung einfacher anzusprechen sind. Wie erwähnt, sind die Mittelteile zueinander statisch fixiert, somit kann man sich einmal für eine Seitenkonfiguration entscheiden, welche Mittelteilefarben welche Seitenbezeichnungen definieren.

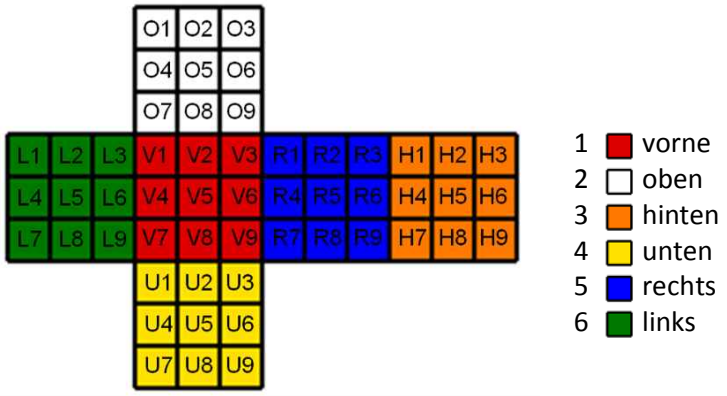


Abbildung 8: Bezeichnungen der Flächenzuordnungen

Diese Zuordnung in *Abbildung 8* ist willkürlich und zufällig, sie könnte genauso gut auch anders sein. Im weiteren Teil meiner Arbeit gelten diese Zuordnungen.

Um nun den Zustand des Würfels mit dieser Art zu beschreiben, ist es möglich, auf jeder Position den Farbwert zu vermerken. Es ist also eine simple Zuordnung, die jeder dieser 54 Flächenpositionen den aktuellen Farbwert bzw. eine Zahl, repräsentativ für den Farbwert, zuordnet.

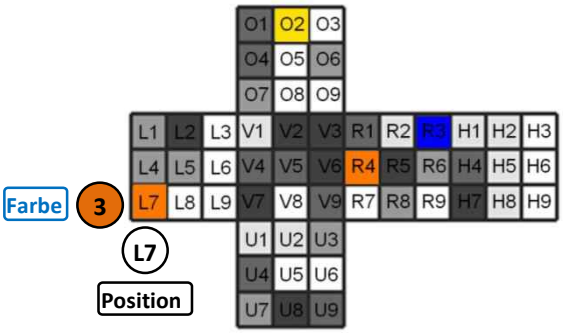
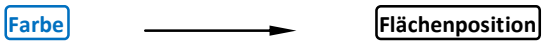


Abbildung 9: Beispiel für Farbzuordnung auf Flächenpositionen



Um dies zu verdeutlichen, können wir konkrete Beispiele betrachten. In *Abbildung 9* sieht man diese Zuordnungen: Auf der Position Oben2 wird der Farbwert gelb = 4 vermerkt oder auf L7=3, R3=5, R4=3, usw. Bei 54 Zuordnungen ist der Würfel eindeutig beschrieben.

1.4 Ecken- und Kantenmodus

Diese Zuordnung der Flächen hat aber einen grossen Nachteil. Während man den Würfel zwar sehr einfach und schnell beschreiben kann, so fehlen viele Informationen, die man zwar daraus ableiten kann. Aber um weitere Schlüsse daraus ziehen zu können, würde dies einen beträchtlichen immer wiederkehrenden Aufwand bedeuten. Es ist angenehm, zu wissen auf welcher Fläche welche Farbe ist, doch es sind 54 Einzelinformationen. Beim Beispiel oben weiss ich z.B. dass der Position R3 die Farbe Blau zugeordnet ist, doch ich weiss nicht, wo diese blaue Fläche hingehört. Nur mit den zusätzlichen Informationen von den Positionen H1 und O3 kann ich entscheiden, wo diese Fläche hingehört. Da man also immer mindestens noch Informationen über eine andere Fläche (Kanten) oder sogar zwei (Ecken) bedarf, liegt die Idee nahe, den Würfel nicht mit den 54 Einzelflächen zu beschreiben, sondern in Gruppen, nämlich den Objekten, konkret mit den Kanten und Ecken. Mit dieser Methode hat man weniger Objekte (12 Kanten, 8 Ecken) mit mehr Informationen.

Wenn wir den Würfel nun mit dieser oben beschriebenen Ansicht betrachten, sodass das rote Mittelstück vorne, das weisse oben usw. ist, so ist man nun wieder vor der Aufgabe, die Ecken und Kanten durchzunummerieren. Es sind prinzipiell keine Ecken und Kanten anderen übergeordnet. Wie wir aber später noch sehen werden, ist es sinnvoll, die ersten vier Kanten in der mittlere Scheibe mit den ersten Nummern zu versehen, den Rest kann willkürlich nummeriert werden. Im weiteren Teil meiner Arbeit gelten diese Konventionen.

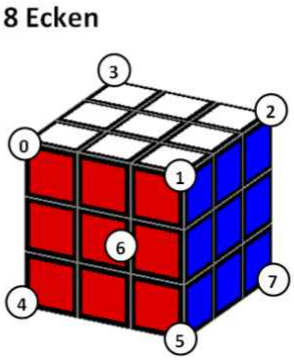


Abbildung 10: Eckpositionen

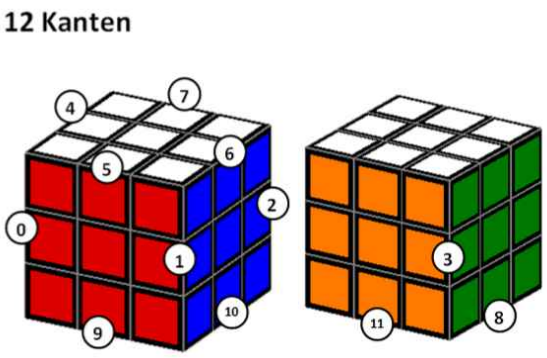


Abbildung 11: Kantenpositionen

Positionen der Objekte

Mit dieser Art den Würfel zu beschreiben ist es uns nun erlaubt, das Objekt (Ecken und Kanten) den Positionen zu zuordnen. Diese nummerierten Ecken und Kanten beschreiben einerseits die Position auf dem Würfel, anderseits aber auch die Nummer, die dem konkreten, finalen Objekt (Kanten oder Ecken) zugehört.

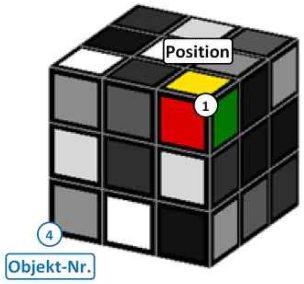
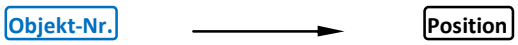


Abbildung 12: Beispiel für Eckzuordnung auf Eckposition



Die obige *Grafik 12* sollte dies verdeutlichen. Es zeigt einen zufällig angeordneten Würfel, wo eine Ecke beleuchtet wird. Die Position dieser Ecke ist die Eckposition Nummer 1. Nach kurzem Betrachten kann man auch feststellen, dass dieser Eckwürfel an die finale Position vier gehören würde. Wir ordnen also dieser aktuellen Konstellation auf der Position 1 den Wert 4 zu. Mit den weiteren Ecken und Kanten kann analog verfahren werden. Um den Würfel zu beschreiben, benötigen wir also eine Zuordnung der acht Ecken und zwölf Kanten auf die Eck- und Kantenpositionen.

Diese Zuordnung scheint auf den ersten Blick sehr effektiv, doch diese Methode ist keinesfalls eindeutig, denn jede Ecke bzw. Kante kann eine andere Orientierung haben. So könnte im obigen Beispiel z.B. die gelbe Fläche rechts, die rote oben und die grüne vorne sein. Mit diesem wichtigen Umstand befassen wir uns im nächsten Abschnitt.

Orientierungen der Objekte

Wie oben erwähnt, sind die Objekte mit der Information über die Position nicht eindeutig beschrieben. Die Ecken können auf drei und die Kanten auf zwei verschiedene Arten orientiert sein.

Abbildung 13: Orientierung 0

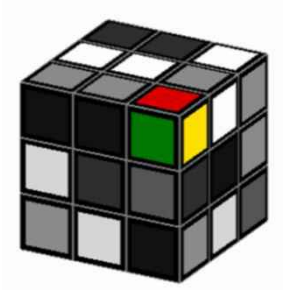


Abbildung 14: Orientierung 1

Abbildung 15: Orientierung 2

Diese Orientierung muss auch quantitativ beschrieben werden können. Um dies bewerkstelligen, ist es nötig, Bezugsflächen einzuführen, auf welchen die Orientierung ablesbar wird.

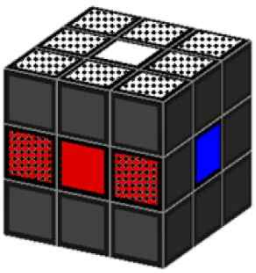


Abbildung 16: Referenzflächen Ansicht 1

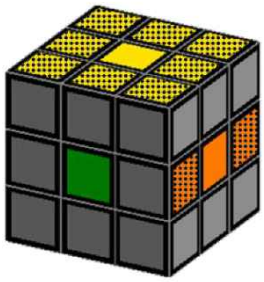


Abbildung 17: Referenzflächen Ansicht 2

Die schraffierten Flächen zeigen die Flächenpositionen auf, wenn die Ecken und Kanten die Orientierung Null haben. Es ist wieder eine willkürliche Zuordnung. Doch es ist wiederum für die weiteren Schritte sinnvoll, wenn die untere und obere Scheibe als Null-Referenz definiert werden. Wenn die Kanten also auf den Bezugsflächen keine schraffierte Kante haben, so hat diese Kante die Orientierung 1. Wenn die Ecke einmal im Uhrzeigersinn rotiert ist, so ist die Orientierung von dieser 1, bei einer doppelten Drehung im Uhrzeigersinn ist sie 2. Bei der *Abbildung 13* hat die Ecke also keine Orientierung (0), bei *Abbildung 14* eine einfache Drehung (Orientierung 1) und bei *Abbildun 15* eine doppelte Drehung (Orientierung 2).

Die Kanten und Ecken werden von nun an mit zwei Zahlen beschrieben. Den Positionen werden einerseits die Objekt-Nummern zugeordnet, anderseits die Orientierung. Mit diesen Informationen ist der Würfel eindeutig beschrieben.

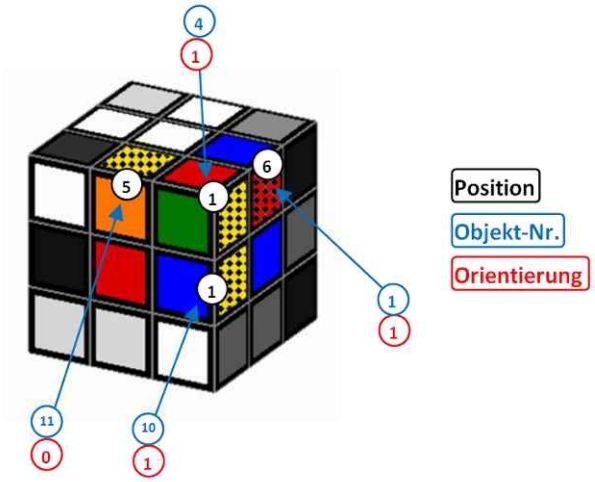
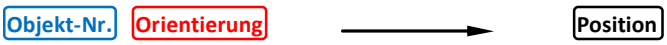


Abbildung 18: Beispiel der Zuordnung von Objekt-Nr. und Orientierung auf Position



2 Auswirkung der Seitenzüge

Nachdem nun die Flächen- bzw. Ecken-/Kantenmodi erläutert sind, kann man sich fragen, wie denn die einzelnen Anordnungen zu Stande kommen. Wenn man den Würfel in die Hand nimmt, so kann man feststellen, dass man allerhand Drehungen anstellen kann. Wenn man allerdings den Würfel im Raum einmal fixiert und wie oben beschrieben, immer darauf achtet, dass die Mittelteile wie definiert anordnet (rot = vorne, weiss = oben, …), so können diese Züge auf sechs reduzieren werden. Nämlich eine Drehung der vorderen (roten) Scheibe um 90° im Uhrzeigersinn, eine derselben Art mit der oberen (weissen) usw. Eine Drehung gegen den Uhrzeigersinn um 90° bedeutet nur die dreimalige Ausführung der zugehörigen Seitenscheibendrehung (bzw. eine Drehung im Uhrzeigersinn um 270°). Ebenso sind die Mittelscheibenzüge überflüssig geworden, denn diese sind gleichzusetzen mit der Drehung der einen äusseren Scheibe im Uhrzeigersinn um 90° und der anderen äusseren gegen den Uhrzeigersinn um 90°. Fazit: Wir fixieren den Würfel im Raum und erlauben nur noch die sechs Scheibenzüge. Diese Konvention vereinfacht die Handhabung mit dem Würfel, ermöglicht aber trotzdem noch jede Drehung auszuführen und jede Konstellation des Würfels zu erreichen.

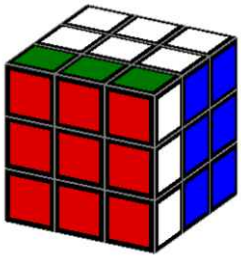


Abbildung 19: Vorne (V)

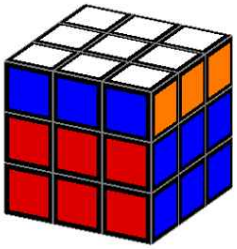


Abbildung 20: Oben(O)

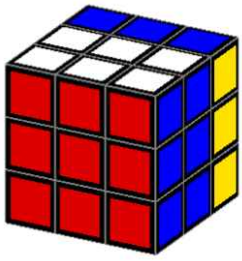


Abbildung 21: Hinten(H)

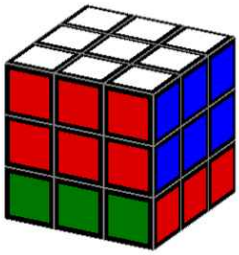


Abbildung 22: Unten (U)

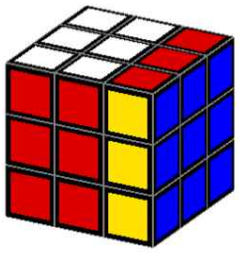


Abbildung 23: Rechts (R)

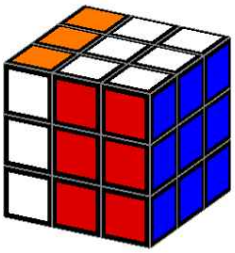


Abbildung 24: Links (L)

Im Weiteren sind die Seitenzüge mit den zugehörigen Grossbuchstaben wie in den *Abbildungen 19-24* abgekürzt. Wie wir festgestellt haben, können zwar alle Seitenzüge auf diese sechs Grundzüge reduziert werden, doch es ist üblich, eine Dreifachdrehung als eine Gegenuhrzeigersinndrehung zu kennzeichnen. Dafür führen wir ein neues Symbol ein. Der zugehörige Grossbuchstabe wird einfach mit einem Apostroph versehen. Ebenso ist es üblich,

eine doppelte Seitenscheibendrehung als einen Zug zu verstehen, dieser kennzeichnen wir mit einem hochgestellten oder angehängten zwei. Damit erhalten wir 18 Züge, nämlich folgende:

V,V2,V‘, O,O2,O‘,H,H2,H‘,U,U2,U‘,R,R2,R‘,L,L2,L‘

Die Anwendung einer der sechs Basisseitenscheibenzüge ergeben immer wieder dieselben analogen Muster in den Veränderungen der Zuordnungen. Diese Muster sind sowohl beim Flächenmodus wie auch beim Ecken-/Kantenmodus feststellbar. Diese Auswirkungen der Züge werden im nächsten Kapitel besprochen.

Wenn man einen Seitenzug anwendet, verändern sich optisch die Farben und die Positionen der Ecken und Kanten. Diese Veränderungen sind spezifisch, je nach dem welcher Zug angewendet wird. In einem ersten Teil werden die Veränderungen der Züge beim Flächenmodus illustriert, in einem zweiten Abschnitt die Auswirkungen beim Ecken- und Kantenmodus.

2.1 Flächenmodus

Bei jedem Seitenscheibenzug werden nur eine begrenzte Anzahl Flächen verändert. Auf der Seitenfläche werden acht Flächen verändert (Mittelstück bleibt bekanntlich gleich) und auf den angrenzenden Seiten je drei zusätzliche. Pro Seitenzug werden also 20 von 54 Flächen verändert. Die Art der Veränderung der Flächen in Korrelation mit den Zügen kann wie folgt gezeigt werden:

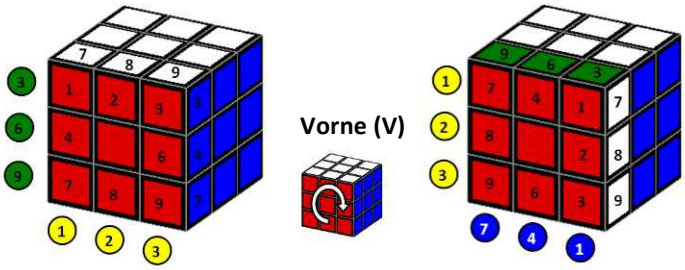


Abbildung 25: Veränderung der Flächen durch Vorne-Drehung

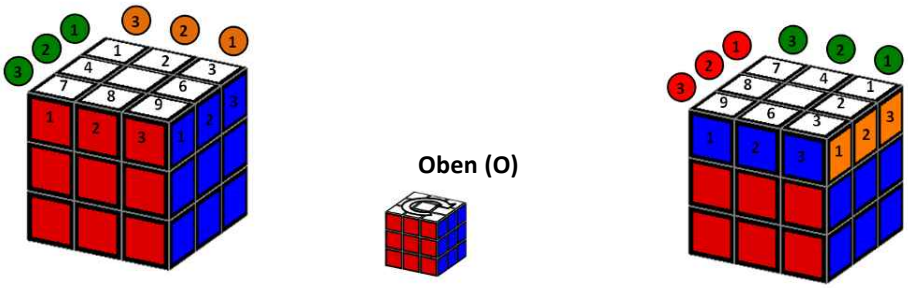


Abbildung 26: Veränderung der Flächen durch Oben-Drehung

Dieser Basisseitenscheibenzug Vorne angewendet auf den „gelösten“ Würfel verändert die acht Flächen auf der vorderen Seite sowie die benachbarten drei Flächen der anliegenden Seiten, analog werden 20 Flächen bei der Drehung der oberen Scheibe (wiederum angewendet auf den „gelösten“ Würfel) verändert.

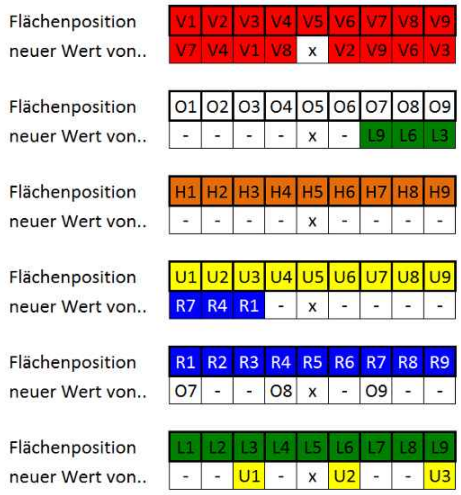


Abbildung 27: Veränderung der Flächen nach {V}

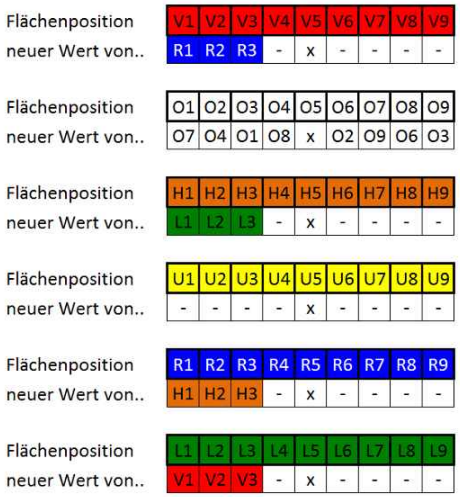


Abbildung 28: Veränderung der Flächen nach {O}

Diese Auflistungen der Zuordnungen ist simpel zu implementieren. Man kann ein Array definieren, welche Werte bei einer z.B. Vorne-Drehung von welchen Positionen übernommen werden.

```
vorne={V7,V4,V1,V8,V5,V2,V9,V6,V3,O1,O2,O3,O4,O5,O6,L9,L6,L3,H1,H2,H3,H4,H5,H6,H7,H8,H9,
       R7,R4,R1,U4,U5,U6,U7,U8,U9,O7,R2,R3,O8,R5,R6,O9,R8,R9,L1,L2,U1,L4,L5,U2,L7,L8,U3};
```

Code 1: Array mit Flächenbezügen für V

Der Blick auf diese *Grafiken 27 und 28* verleitet vielleicht zu einer Annahme, die sich als Scheinkorrelation entpuppen würde. Wenn wir z.B. die Position L9 (V) anschauen, so können wir ablesen, dass die Bezugsfläche U3 ist. Die zugehörige finale Farbe der Position U3 ist gelb, hier allerdings auch der Wert, da wir ja von einem „gelösten“ Würfel ausgegangen sind. Die farbliche Einfärbung in den *Grafiken 27 und 28* veranschaulicht den gespeicherten Farbwert dieser Positionen. Wäre die Fläche U3 zuvor z.B. rot gefärbt gewesen, so würde an dieser Stelle natürlich eine rote Fläche vorzufinden sein. Dieser Sachverhalt wird beim nächsten Überlegungsschritt noch näher erklärt.

Kombination von mehreren Zügen

Wir wenden auf den gelösten Würfel zuerst eine Drehung der vorderen Scheibe an, auf diesen einen Seitenzug Oben an. Die Reihenfolge der Züge darf nicht verändert werden, das Kommutativgesetz ist nicht gültig. Der erste Schritt ist derselbe wie oben, allerdings unter Berücksichtigung von weiteren Flächen. Bei der Kombination von diesen beiden Seitenzügen werden 32 von 54 Flächen verändert.

Abbildung 29: Veränderung der Flächen durch Vorne-Drehung

Nach diesem Schritt werden die Flächenpositionen natürlich nicht verändert, sondern nur der Wert von den Bezugsflächen übergeben.

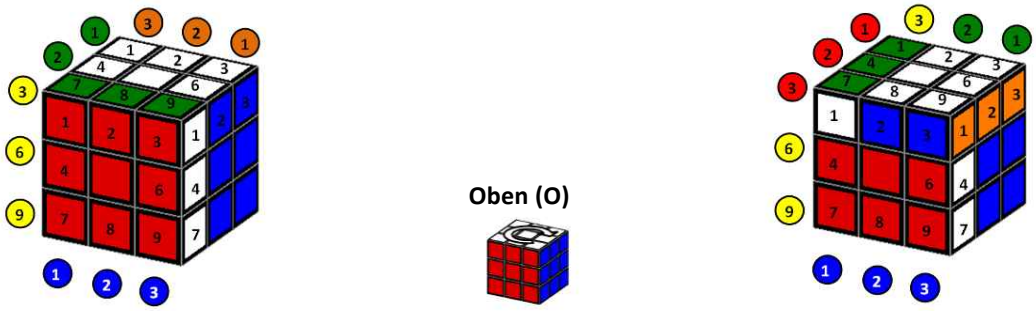


Abbildung 30: Veränderung der Flächen durch Oben-Drehung, nach {V}

Flächenposition	V1	V2	V3	V4	V5	V6	V7	V8	V9
neuer Wert von..	V7	V4	V1	V8	x	V2	V9	V6	V3

Flächenposition	O1	O2	O3	O4	O5	O6	O7	O8	O9
neuer Wert von..	-	-	-	-	x	-	L9	L6	L3

Flächenposition	H1	H2	H3	H4	H5	H6	H7	H8	H9
neuer Wert von..	-	-	-	-	x	-	-	-	-

Flächenposition	U1	U2	U3	U4	U5	U6	U7	U8	U9
neuer Wert von..	R7	R4	R1	-	x	-	-	-	-

Flächenposition	R1	R2	R3	R4	R5	R6	R7	R8	R9
neuer Wert von..	O7	-	-	O8	x	-	O9	-	-

Flächenposition	L1	L2	L3	L4	L5	L6	L7	L8	L9
neuer Wert von..	-	-	U1	-	x	U2	-	-	U3

Abbildung 31: Veränderung der Flächen nach {V}

Flächenposition	V1	V2	V3	V4	V5	V6	V7	V8	V9
neuer Wert von..	R1	R2	R3	-	x	-	-	-	-

Flächenposition	O1	O2	O3	O4	O5	O6	O7	O8	O9
neuer Wert von..	O7	O4	O1	O8	x	O2	O9	O6	O3

Flächenposition	H1	H2	H3	H4	H5	H6	H7	H8	H9
neuer Wert von..	L1	L2	L3	-	x	-	-	-	-

Flächenposition	U1	U2	U3	U4	U5	U6	U7	U8	U9
neuer Wert von..	-	-	-	-	x	-	-	-	-

Flächenposition	R1	R2	R3	R4	R5	R6	R7	R8	R9
neuer Wert von..	H1	H2	H3	-	x	-	-	-	-

Flächenposition	L1	L2	L3	L4	L5	L6	L7	L8	L9
neuer Wert von..	V1	V2	V3	-	x	-	-	-	-

Abbildung 32: Veränderung der Flächen nach {V,O}

Diese Zuordnungen in Korrelation mit den Seitenzügen kann auf jeden beliebigen Seitenzug analog übertragen werden.

2.2 Ecken- und Kantenmodus

Die Überlegungen zu den Auswirkungen auf die Ecken bzw. Kanten sind für die Positionsbezüge analog zu denjenigen der Flächen anzugehen. Auch hier wird nur eine begrenzte Anzahl an Objekten pro Seitenzug verändert. Pro Zug werden sowohl vier Ecken wie auch vier Kanten vertauscht. Diese geringe Anzahl an Veränderungen ist ein grosser Vorteil dieser Methode.

Zur Veranschaulichung werden kurz die Auswirkungen von den zwei Basisseitenzügen Vorne und Rechts beschrieben, die restlichen Züge sind wiederum im Anhang vollständig dokumentiert.

Ecken

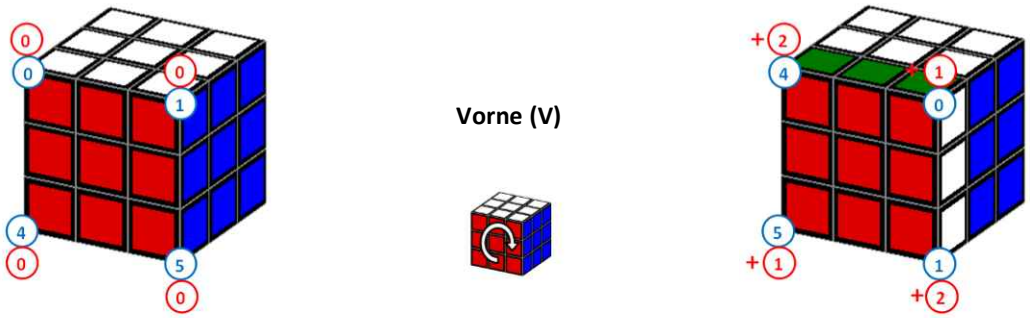


Abbildung 33: Veränderung der Ecken durch Vorne-Drehung

Position	0	1	2	3	4	5	6	7
Orientierung	0	0	0	0	0	0	0	0
Objekt-Nr.	4	0	-	-	5	1	-	-
neue Ori.	0 + 2	0 + 1	-	-	0 + 1	0 + 2	-	-

Abbildung 34: Veränderung der Ecken durch Vorne-Drehung

Die Veränderungen der Orientierungen sind etwas komplexer. Jeder Seitenzug verleiht derselben Ecke auf einer Position stets die gleiche Veränderung der Orientierung, welche nicht abhängig vom Objekt (und Orientierung) ist, das sich verändert. Für jeden Seitenzug und jede einzelne Ecke gibt es eine spezifische Vergrösserung der Orientierung entweder um 0, 1 oder 2.

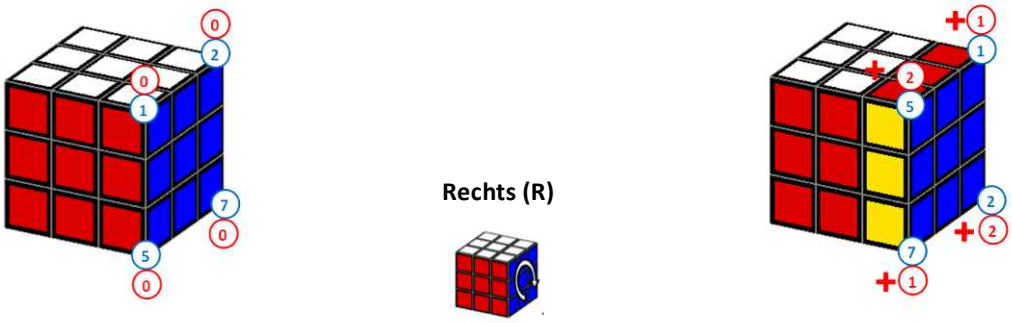


Abbildung 35: Veränderung der Ecken durch Rechts-Drehung

Position	0	1	2	3	4	5	6	7
Orientierung	0	0	0	0	0	0	0	0
Objekt-Nr.	-	5	1	-	-	7	-	2
neue Ori.	-	0 + 2	0 + 1	-	-	0 + 1	-	0 + 2

Abbildung 36: Veränderung der Ecken durch Rechts-Drehung

Die Veränderung der Orientierung ist zwar nicht abhängig von dem zu verändernden Objekt, die tatsächlich resultierende Orientierung jedoch sehr wohl. Die Vergrösserung addiert sich zu der Orientierung der neuen Ecke auf dem alten Platz. Dabei ist z.B. eine Orientierung von 4 gleichbedeutend wie eine Orientierung von 1, 6 gleich 0 usw. Diese zirkuläre Reduzierung der Orientierung ist mit der Modulodivision durch 3 zu erreichen. Bei der Kombination von mehreren Zügen wird die Methodik vielleicht noch klarer, da diese Basiszüge auf den finalen Würfel angewendet sind und so die Grundorientierungen alle 0 sind.

Diese Veränderungen der Ecken lassen sich wieder in einer Tabelle bzw. in einem Array darstellen. Bemerkenswert bei dieser Methode ist, dass mit wenigen Beschreibungen viele Informationen enthalten sind.

```
//Transformationsarray für Eckpositionen[i][j]
//i: Vorne=0  Oben=1  Hinten=2  Unten=3  Rechts=4  Links=5
//j: Eck-Nr.
int move[][]={{4,0,2,3,5,1,6,7},{1,2,3,0,4,5,6,7},
{0,1,7,2,4,5,3,6},{0,1,2,3,6,4,7,5},
{0,5,1,3,4,7,6,2},{3,1,2,6,0,5,4,7}};

//Transformationsarray für Eckorientierung[i][j]
//i: Vorne=0  Oben=1  Hinten=2  Unten=3  Rechts=4  Links=5
//Werte für 0:keine Rotation  1:rechts Rotation  2: links Rotation
int move2[][]={{2,1,0,0,1,2,0,0},{0,0,0,0,0,0,0,0},
{0,0,2,1,0,0,2,1},{0,0,0,0,0,0,0,0},
{0,2,1,0,0,1,0,2},{1,0,0,2,2,0,1,0}};
```

Code 2: Array mit Eckpositions- und Eckorientierungsbezügen

Kombination von mehreren Zügen

Wir wenden auf den gelösten Würfel zuerst wieder eine Drehung der vorderen Scheibe an, auf diese veränderte Stellung einen Seitenzug rechts. Das Kommutativgesetz ist selbstverständlich auch hier nicht gültig, die Reihenfolge muss also beachtet werden.

Der erste Schritt ist derselbe wie oben, allerdings unter Berücksichtigung von weiteren Ecken. Bei der Kombination von diesen beiden Seitenzügen werden sechs Ecken (und sieben Kanten verändert).

Nach der Ausführung der Vorne-Drehung sind folgende Zuordnungen geltend:

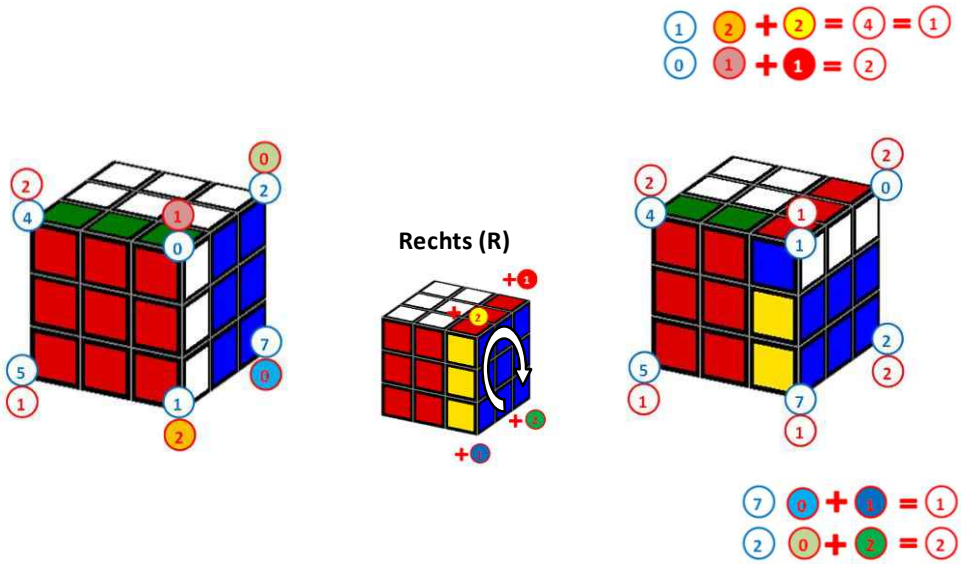


Abbildung 37: Veränderung der Eckpositionen und –orientierungen nach {V,R}

Durch die Anwendung des Seitenscheibenzug Rechts werden die Veränderungen der Orientierungen auf vier Ecken addiert. Die Ecke Nummer 1 z.B. hat nach dem Zug Vorne die Orientierung 2 und liegt auf der Position 5. Durch eine R-Drehung wird die Position zu 1 und die Orientierung von 2 zu 4 addiert, was äquivalent zu einer Orientierung von 1 ist.

Position	0	1	2	3	4	5	6	7
Orientierung	0	0	0	0	0	0	0	0
Vorne Drehung								
Objekt-Nr.	4	0	-	-	5	1	-	-
neue Ori.	0 + 2	0 + 1	-	-	0 + 1	0 + 2	-	-

Position	0	1	2	3	4	5	6	7
Objekt-Nr.	4	0	2	3	5	1	6	7
Orientierung	2	1	0	0	1	2	0	0
Rechts-Drehung								
Objekt-Nr.	-	1	0	-	-	7	-	2
neue Ori.	-	2 + 2	1 + 1	-	-	0 + 1	-	0 + 2

Abbildung 38: Veränderung der Eckpositionen und –orientierungen nach {V,R}

Kanten

Die Auswirkungen der Seitenzüge auf die Kanten sind analog zu denen der Ecken. Insgesamt gibt es zwölf Kantenstücke die untereinander vertauscht werden können, pro Zug werden vier zyklisch vertauscht. Die Orientierung verändert sich jeweils um null oder eins. Die Illustration dieser Auswirkungen befindet sich im Anhang.

Hier ist noch die Definitionen der Transformationsarrays für Kanten und deren Orientierung.

```
//Transformationsarray für Kantenpositionen[i][j]
//i: Vorne=0  Oben=1  Hinten=2  Unten=3  Rechts=4  Links=5
//Kanten-Nr./Werte für 0-11
int movek[][]={{9,5,2,3,4,0,6,7,8,1,10,11},{0,1,2,3,5,6,7,4,8,9,10,11},
{0,1,11,7,4,5,6,2,8,9,10,3},{0,1,2,3,4,5,6,7,11,8,9,10},
{0,10,6,3,4,5,1,7,8,9,2,11},{4,1,2,8,3,5,6,7,0,9,10,11}};

//Transformationsarray für Kantenorientierung[i][j]
//i: Vorne=0  Oben=1  Hinten=2  Unten=3  Rechts=4  Links=5
//Werte für 0:keine Rotation  1: Rotation
int movek2[][]={{1,1,0,0,0,1,0,0,0,1,0,0},{0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,1,1,0,0,0,1,0,0,0,1},{0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0,0,0}};
```

Code 3: Array mit Kantenpositions- und Kantenorientierungsbezügen

2.3 Implementierung von Zugkombination

Wenn die Transformationsarrays für die Eckpositionen und Eckorientierungen sowie Kantenpositionen und Kantenorientierungen definiert sind, kann eine erste Implementierung geschrieben werden.

```
int[] EckKoord={0,1,2,3,4,5,6,7};
int[] EckKoordOri={0,0,0,0,0,0,0,0};

int[] KantKoord={0,1,2,3,4,5,6,7,8,9,10,11};
int[] KantKoordOri={0,0,0,0,0,0,0,0,0,0,0,0};
```

Code 4: Initialisierung der Felder

Als erstes müssen noch Felder initialisiert werden, in denen die aktuellen Objekt-Nummern gespeichert werden können.

```
void addMove(int a) {
  ...
  ...
}
```

Code 5: Funktion *addMove()*

Eine Funktion mit einem Eingangsparameter für die Art des Seitenzuges wird definiert. Die weiteren Schritte sind in dieser Funktion enthalten.

```
int temp[]=new int[8];
int temp2[]=new int[8];

for(int i=0;i<=7;i++) {
  temp[i]=EckKoord[i];
  temp2[i]=EckKoordOri[i];
}
```

Code 6: Eckwerte werden kopiert

```
int tempk[]=new int[12];
int tempk2[]=new int[12];

for(int i=0;i<=11;i++) {
  tempk[i]=KantKoord[i];
  tempk2[i]=KantKoordOri[i];
}
```

Code 7: Kantenwerte werden kopiert

Danach werden die Felder von den Ecken und Kanten temporär gespeichert, ansonsten entstünde eine zirkuläre Definition, d.h. es würde Bezug genommen werden auf schon neu gefüllte Felder.

```
for(int i=0;i<=7;i++) {
  EckKoord[i]=temp[move[a][i]];
}

for(int i=0;i<=7;i++) {
  EckKoordOri[i]=(temp2[move[a][i]]+move2[a][i])%3;
}
```

Code 8: Objektbezüge der Ecken werden ausgewertet

Code 8 ist die Implementation der theoretischen Erkenntnisse der letzten Abschnitte dieser Arbeit. Die Werte der Eckpositionen werden von den Positionen übernommen, an denen im Feld *move[ ]* die Positionsveränderungen gespeichert wurden.

```
//Transformationsarray für Eckpositionen[
//i: Vorne=0  Oben=1  Hinten=2  Unten=3
//j: Eck-Nr.
int move[][]={{4,0,2,3,5,1,6,7},{1,2,3,0,
{0,1,7,2,4,5,3,6},{0,1,2,3,6,4,7,5},
{0,5,1,3,4,7,6,2},{3,1,2,6,0,5,4,7}};

//Transformationsarray für Eckorientierung[i][j]
//i: Vorne=0  Oben=1  Hinten=2  Unten=3  Rechts=4  Links=5
//Werte für 0:keine Rotation  1:rechts Ro
int move2[][]={{2,1,0,0,1,2,0,0},{0,0,0,0
{0,0,2,1,0,0,2,1},{0,0,0,0,0,0,0,0},
{0,2,1,0,0,1,0,2},{1,0,0,2,2,0,1,0}};
```

Zugart: a = 0
Position: i = 5

Zugart: a = 4
Position: i = 7

Code 9: Beispiel für zwei verschiedene Parameterpaare

Wenn wir ein konkretes Beispiel anschauen, so sieht man, dass bei einem gewählten Eingangsparameter a = 0, also eine Vorne-Drehung, auf der Eckposition 5 den neuen Wert von der Position 1 (Eck-Nr. 1) übergeben wird.

Auch die Veränderung der Orientierung ist wie besprochen implementiert. Konkret wird beim obigen Beispiel ein Seitenzug Rechts (a = 4) angewendet. Wenn wir z.B. schauen, wie bei der Eckposition 7 die Orientierung zu Stande kommt, so sehen wir, dass zu der aktuellen

Orientierung an der Position der potentiellen Ecke die spezifischen Veränderung der Orientierung (abhängig von der Zugart und Eckposition) addiert wird.

```
for(int i=0;i<=11;i++) {
  KantKoord[i]=tempk[movek[a][i]];
}

for(int i=0;i<=11;i++) {
  KantKoordOri[i]=(tempk2[movek[a][i]]+movek2[a][i])%2;
}
```

Code 10: Objektbezüge der Kanten werden ausgewertet

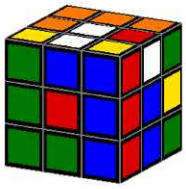
Zur Vollständigkeit ist die analoge Implementierung in *Code 10* der Kanten aufgeführt. Diese Implementierung ist auf der CD beigelegt unter dem Programm 1. Es kann variiert werden, welche Seitenzüge angewendet werden, Kombinationen sind möglich. Als Ausgabe werden die neuen Positionen und Orientierungen der Ecken und Kanten aufgeführt.

3 Gesamtkoordinaten

Die Kenntnisse bis anhin erlauben uns also, jeden beliebigen Würfel mit vier unterschiedlichen Informationen zu beschreiben.

I.	**Positionen der Ecken**
II.	**Orientierung der Ecken**
III.	**Positionen der Kanten**
IV.	**Orientierung der Kanten**

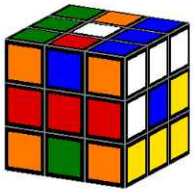
Um dies als letztes Mal zu veranschaulichen, wenden wir auf den ersten Würfel die Zugfolge *{V,O,U,R,L,H}* an, auf den zweiten die Folge *{R,L,H,V,O,U}*. Folgende Beschreibungen erhält man:



```
EckKoord={4,5,3,2,0,1,7,6};
EckKoordOri={0,0,1,2,0,0,1,2};

KantKoord={0,1,10,4,3,6,5,7,9,8,2,11};
KantKoordOri={1,1,1,1,0,0,1,1,1,0,0,1};
```

Abbildung 39: Beispiel 1 nach {V,O,U,R,L,H}



```
EckKoord={3,2,1,0,6,7,4,5};
EckKoordOri={2,1,2,1,1,2,2,1};

KantKoord={9,5,11,7,4,1,6,3,8,0,10,2};
KantKoordOri={1,1,1,1,1,0,1,0,1,0,1,0};
```

Abbildung 40: Beispiel 2 nach {R,L,H,V,O,U}

Mit diesen Zuordnungen kann man aber immer noch nicht viel anfangen. Man weiss zwar, wie der Würfel nach diesen Zügen aufgebaut ist. Doch das Ziel ist ja, eine optimale Zugfolge zu finden, die den Würfel in eine bestimmte Konstellation überführt. Um dies zu erreichen, kann man diese Zuordnungen in eine Reihenfolge einordnen. In eine Reihenfolge, die aussagt, welchen Würfel ich mit den Zahlen vor mir habe. Das heisst also, ich muss zuerst die Anzahl aller möglichen Stellungen kennen, die der Würfel annehmen kann und kann dann den konkreten Würfel in diese Reihe einordnen.

Die Gesamtzahl aller möglichen Stellungen erhält man durch kombinatorische Überlegungen. Auf der ersten Eckposition können alle acht Ecken angeordnet sein, auf der zweiten noch sieben, bis auf der letzten Position nur noch eine sein kann. Durch die Multiplikation dieser Möglichkeiten, also $8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 8! = 40'320$ erhält man die Anzahl der Möglichkeiten der Anordnungen der acht Ecken.

Dieselben Überlegungen kann man für die Anordnungen der Kanten anstellen. Auf der ersten Position können zwölf verschiedene angeordnet werden, auf der zweiten nur noch elf, bis

wieder auf der zwölften Position nur noch eine Kante sein kann. Wiederum multipliziert erhält man *12·11·10·9·8·7·6·5·4·3·2·1=12!=479'001'600*. Es ist aber unmöglich, zwei Ecken miteinander zu vertauschen ohne dass dabei keine Kanten vertauscht werden. Wenn also zwei Eckwürfel in ihrer Stelle vertauscht, aber nicht verdreht sind, dann sind auch zwei Kanten miteinander vertauscht. Daraus folgt, dass die tatsächlich erreichbaren Möglichkeiten noch halbiert werden müssen.

Zugleich können die Ecken und Kanten in ihrer Orientierung verändert sein. Auf jeder Eckposition können die Ecken drei verschiedene Orientierungen einnehmen, jede Kante zwei verschiedene. Doch wenn sieben Ecken mit ihrer Orientierung fixiert sind, so ist die letzte Ecke bestimmt. Denn durch jede Anwendung eines Zuges wird die gesamte Orientierung um eine Zahl erhöht, die bei einer Modulodivision durch drei null ergibt. Das heisst, die Orientierung der letzten Ecke ist nicht frei wählbar. Durch Verrechnung dieser Möglichkeiten erhält man für die Eckenorientierungen *3·3·3·3·3·3·3 = 3^7 = 2187* bzw. *2·2·2·2·2·2·2·2·2·2·2·2=2^{11} = 2048* für die Kantenorientierungen. Diese fünf kombinatorischen Überlegungen führen zur Gesamtzahl:

$$\frac{8! \cdot 12! \cdot 3^7 \cdot 2^{11}}{2} = 4'325'200'327'448'9856'000 = 4.3 \cdot 10^{19}$$

Jetzt kann man jede konkrete Stellung durchnummerieren. Dazu benötigt man die lexikografische Schreibweise jeder einzelnen der vier Informationskomponenten.

3.1 Orientierung der Ecken

Um eine konkrete Stellung in die Reihe einordnen zu können, muss man die lexikografische Auflistung aller möglichen Orientierungen der Ecken erstellen. Die letzte Eckposition besitzt keine lexikografische Bedeutung, sondern nimmt die Orientierung an, die aus den ersten sieben folgen. Danach kann man die konkrete Position darin einordnen.

```
0: EckKoordOri={0,0,0,0,0,0,0,0};
1: EckKoordOri={1,0,0,0,0,0,0,2};
2: EckKoordOri={2,0,0,0,0,0,0,1};
3: EckKoordOri={0,1,0,0,0,0,0,2};
4: EckKoordOri={1,1,0,0,0,0,0,1};
5: EckKoordOri={2,1,0,0,0,0,0,0};
6: EckKoordOri={0,2,0,0,0,0,0,1};
7: EckKoordOri={1,2,0,0,0,0,0,0};
8: EckKoordOri={2,2,0,0,0,0,0,2};
9: EckKoordOri={0,0,1,0,0,0,0,2};
10: EckKoordOri={1,0,1,0,0,0,0,1};
11: EckKoordOri={2,0,1,0,0,0,0,0};
12: EckKoordOri={0,1,1,0,0,0,0,1};
                .
                .
                .
```

```
                .
                .
                .
2174: EckKoordOri={2,1,1,2,2,2,2,0};
2175: EckKoordOri={0,2,1,2,2,2,2,1};
2176: EckKoordOri={1,2,1,2,2,2,2,0};
2177: EckKoordOri={2,2,1,2,2,2,2,2};
2178: EckKoordOri={0,0,2,2,2,2,2,2};
2179: EckKoordOri={1,0,2,2,2,2,2,1};
2180: EckKoordOri={2,0,2,2,2,2,2,0};
2181: EckKoordOri={0,1,2,2,2,2,2,1};
2182: EckKoordOri={1,1,2,2,2,2,2,0};
2183: EckKoordOri={2,1,2,2,2,2,2,2};
2184: EckKoordOri={0,2,2,2,2,2,2,0};
2185: EckKoordOri={1,2,2,2,2,2,2,2};
2186: EckKoordOri={2,2,2,2,2,2,2,1};
```

Code 11: Lexikografische Auflistung der möglichen Orientierungen der Ecken

Man versucht also, die verschiedenen Anordnungen der Eckorientierungen auf den Positionen in eine Reihenfolge zu bringen. Um dies zu erreichen, werden den Eckpositionen „Wertigkeiten“ verliehen. Also der Wert auf der nullten Position bekommt die Wertigkeit 1, der Wert auf der ersten den eins grösseren Wert als das mögliche Maximum (2), also 3. Durch die Multiplikation dieser Wertigkeit für die erste Position (3) mit dem möglichen Maximum (2) und der Addition mit dem Maximum auf der nullten Position (2) erhält man (3·2+2) 8. 8 ist das mögliche Maximum an der ersten Position. Durch die Vergrösserung um 1 erhält man 9, die Wertigkeit für die zweite Position. Führt man dies weiter bis zur sechsten Position (die siebte ist durch die ersten sieben bereits determiniert), erhält man jeweils die Wertigkeiten 3^i für die i-te Position. Mit dieser Methode erhält man Zahlen von 0 bis 3^7-1 (2186).

Wenn man die Beispiele aus den *Abbildungen 39 und 40* nimmt, so bekommt man die Zahl für die Orientierung der Ecken wie folgt:

```
EckKoordOri={0,0,1,2,0,0,1,2};
```

Position	i	0	1	2	3	4	5	6	7
Orientierung	ori(i)	0	0	1	2	0	0	1	2
Wertigkeit	3^i	$3^0 = 1$	$3^1 = 3$	$3^2 = 9$	$3^3 = 27$	$3^4 = 81$	$3^5 = 243$	$3^6 = 729$	
Wert	3^i * ori(i)	1 * 0 = 0	3 * 0 = 0	9 * 1 = 9	27 * 2 = 54	81 * 0 = 0	243 * 0 = 0	729 * 1 = 729	792

Abbildung 41: Beispiel 1; Herleitung der gesamten Eckorientierungen

```
EckKoordOri={2,1,2,1,1,2,2,1};
```

Position	i	0	1	2	3	4	5	6	7
Orientierung	ori(i)	2	1	2	1	1	2	2	1
Wertigkeit	3^i	$3^0 = 1$	$3^1 = 3$	$3^2 = 9$	$3^3 = 27$	$3^4 = 81$	$3^5 = 243$	$3^6 = 729$	
Wert	3^i * ori(i)	1 * 2 = 2	3 * 1 = 3	9 * 2 = 18	27 * 1 = 27	81 * 1 = 81	243 * 2 = 486	729 * 2 = 1458	2075

Abbildung 42: Beispiel 2; Herleitung der gesamten Eckorientierungen

Zur Kontrolle kann man die maximalen Orientierungen der Ecken annehmen, also die ersten sieben Ecken mit einer 2 versehen, die letzte mit einer 1. Dabei sollte der maximale Wert von 3^7*-1 = 2186* herauskommen. Sind alle Ecken dagegen richtig orientiert, das heisst, jede Ecke hat eine Orientierung von 0, so ist es einleuchtend, dass die Gesamtorientierung der Ecken ebenfalls 0 ergibt.

Position	i	0	1	2	3	4	5	6	7
Orientierung	ori(i)	2	2	2	2	2	2	2	1
Wertigkeit	3^i	$3^0 = 1$	$3^1 = 3$	$3^2 = 9$	$3^3 = 27$	$3^4 = 81$	$3^5 = 243$	$3^6 = 729$	
Wert	$3^i * ori(i)$	1 * 2 = 2	3 * 2 = 6	9 * 2 = 18	27 * 2 = 54	81 * 2 = 162	243 * 2 = 486	729 * 2 = 1458	2186

Abbildung 43: maximale Orientierung der Ecken

3.2 Orientierung der Kanten

Zuerst benötigt man wieder die lexikografische Auflistung aller möglichen Orientierungen der ersten elf Kanten. Darin wird die konkrete Stellung eingeordnet.

```
0: KantKoordOri={0,0,0,0,0,0,0,0,0,0,0};
1: KantKoordOri={1,0,0,0,0,0,0,0,0,0,1};
2: KantKoordOri={0,1,0,0,0,0,0,0,0,0,1};
3: KantKoordOri={1,1,0,0,0,0,0,0,0,0,0};
4: KantKoordOri={0,0,1,0,0,0,0,0,0,0,1};
5: KantKoordOri={1,0,1,0,0,0,0,0,0,0,0};
6: KantKoordOri={0,1,1,0,0,0,0,0,0,0,0};
7: KantKoordOri={1,1,1,0,0,0,0,0,0,0,1};
8: KantKoordOri={0,0,0,1,0,0,0,0,0,0,1};
9: KantKoordOri={1,0,0,1,0,0,0,0,0,0,0};
10: KantKoordOri={0,1,0,1,0,0,0,0,0,0,0};
11: KantKoordOri={1,1,0,1,0,0,0,0,0,0,1};
12: KantKoordOri={0,0,1,1,0,0,0,0,0,0,0};
                .
                .
                .
```

```
                .
                .
                .
2035: KantKoordOri={1,1,0,0,1,1,1,1,1,1,1};
2036: KantKoordOri={0,0,1,0,1,1,1,1,1,1,0};
2037: KantKoordOri={1,0,1,0,1,1,1,1,1,1,1};
2038: KantKoordOri={0,1,1,0,1,1,1,1,1,1,1};
2039: KantKoordOri={1,1,1,0,1,1,1,1,1,1,0};
2040: KantKoordOri={0,0,0,1,1,1,1,1,1,1,0};
2041: KantKoordOri={1,0,0,1,1,1,1,1,1,1,1};
2042: KantKoordOri={0,1,0,1,1,1,1,1,1,1,1};
2043: KantKoordOri={1,1,0,1,1,1,1,1,1,1,0};
2044: KantKoordOri={0,0,1,1,1,1,1,1,1,1,1};
2045: KantKoordOri={1,0,1,1,1,1,1,1,1,1,0};
2046: KantKoordOri={0,1,1,1,1,1,1,1,1,1,0};
2047: KantKoordOri={1,1,1,1,1,1,1,1,1,1,1};
```

Code 12: Lexikografische Auflistung der möglichen Orientierungen der Kanten

Die Hochrechnung der gesamten Orientierung für die Kanten ist wiederum analog zu deren der Ecken. Der Unterschied ist eine Wertigkeit von 2^i anstatt 3^i, da die Kanten nur maximal einfach orientiert sein können. Die letzte Kante ist wiederum für diese Berechnung nicht von Bedeutung.

KantKoordOri={1,1,1,1,0,0,1,1,1,0,0,1};

Position	i	0	1	2	3	4	5	6	7	8	9	10	11
Orientierung	ori(i)	1	1	1	1	0	0	1	1	1	0	0	1
Wertigkeit	2^i	$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$	$2^7 = 128$	$2^8 = 256$	$2^9 = 512$	$2^{10} = 1024$	
Wert	$2^i * ori(i)$	1 * 1 = 1	2 * 1 = 2	4 * 1 = 4	8 * 1 = 8	16 * 0 = 0	32 * 0 = 0	64 * 1 = 64	128 * 1 = 128	256 * 1 = 256	512 * 0 = 0	1024 * 0 = 0	463

Abbildung 44: Beispiel 1; Herleitung der gesamten Kantenorientierungen

```
KantKoordOri={1,1,1,1,1,0,1,0,1,0,1,0};
```

Position	i	0	1	2	3	4	5	6	7	8	9	10	11
Orientierung	ori(i)	1	1	1	1	1	0	1	0	1	0	1	0
Wertigkeit	2^i	$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$	$2^7 = 128$	$2^8 = 256$	$2^9 = 512$	$2^{10} = 1024$	
Wert	$2^i * ori(i)$	1 * 1	2 * 1	4 * 1	8 * 1	16 * 1	32 * 0	64 * 1	128 * 0	256 * 1	512 * 0	1024 * 1	
		1	2	4	8	16	0	64	0	256	0	1024	1375

Abbildung 45: Beispiel 2; Herleitung der gesamten Kantenorientierungen

Zur Kontrolle kann man wieder die maximalen Orientierungen der Kanten annehmen, also alle Kanten mit einer eins versehen. Dabei sollte der maximale Wert von $2^{11}-1 = 2047$ herauskommen. Auch bei den Kanten ergibt die gesamte Orientierung null, wenn alle Kanten richtig orientiert sind.

Position	i	0	1	2	3	4	5	6	7	8	9	10	11
Orientierung	ori(i)	1	1	1	1	1	1	1	1	1	1	1	1
Wertigkeit	2^i	$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$	$2^7 = 128$	$2^8 = 256$	$2^9 = 512$	$2^{10} = 1024$	
Wert	$2^i * ori(i)$	1 * 1	2 * 1	4 * 1	8 * 1	16 * 1	32 * 1	64 * 1	128 * 1	256 * 1	512 * 1	1024 * 1	
		1	2	4	8	16	32	64	128	256	512	1024	2047

Abbildung 46: maximale Orientierung der Kanten

3.3 Positionen der Ecken

Um eine Reihenfolge für die Eckpositionen erstellen zu können, benötigt man wiederum die Auflistung aller möglichen Permutationen. Aus implementationstechnischen vereinfachenden Gründen verwende ich nicht die lexikografische Auflistung, sondern man könnte sie als inverse oder arabische Lexikografie beschreiben. Am Prinzip ändert sich durch diese Abweichung nichts.

```
40319: EckKoord={7 6 5 4 3 2 1 0};
40318: EckKoord={6 7 5 4 3 2 1 0};
40317: EckKoord={7 5 6 4 3 2 1 0};
40316: EckKoord={5 7 6 4 3 2 1 0};
40315: EckKoord={6 5 7 4 3 2 1 0};
40314: EckKoord={5 6 7 4 3 2 1 0};
40313: EckKoord={7 6 4 5 3 2 1 0};
40312: EckKoord={6 7 4 5 3 2 1 0};
40311: EckKoord={7 4 6 5 3 2 1 0};
40310: EckKoord={4 7 6 5 3 2 1 0};
40309: EckKoord={6 4 7 5 3 2 1 0};
40308: EckKoord={4 6 7 5 3 2 1 0};
40307: EckKoord={7 5 4 6 3 2 1 0};
                .
                .
                .
```

```
                .
                .
                .
12: EckKoord={0 2 3 1 4 5 6 7};
11: EckKoord={3 1 0 2 4 5 6 7};
10: EckKoord={1 3 0 2 4 5 6 7};
9: EckKoord={3 0 1 2 4 5 6 7};
8: EckKoord={0 3 1 2 4 5 6 7};
7: EckKoord={1 0 3 2 4 5 6 7};
6: EckKoord={0 1 3 2 4 5 6 7};
5: EckKoord={2 1 0 3 4 5 6 7};
4: EckKoord={1 2 0 3 4 5 6 7};
3: EckKoord={2 0 1 3 4 5 6 7};
2: EckKoord={0 2 1 3 4 5 6 7};
1: EckKoord={1 0 2 3 4 5 6 7};
0: EckKoord={0 1 2 3 4 5 6 7};
```

Code 13: Lexikografische Auflistung der möglichen Positionen der Ecken

Um Rückschlüsse von einer gegebenen Stellung zur Position in dieser Reihe zu ziehen, unterscheidet sich die Methodik zur der vorherigen. Man beachtet bei jeder Position die Anzahl grösseren Ecken auf niedrigeren Positionen. Man kennzeichnet quasi diejenigen, die aus der „Reihe“ tanzen.

```
EckKoord={4,5,3,2,0,1,7,6};
```

Position	i	0	1	2	3	4	5	6	7
Eck-Nummer	eck(i)	4	5	3	2	0	1	7	6
Links_grösser	li_g(i)		0	2	3	4	4	0	1
Wertigkeit	i!		1! = 1	2! = 2	3! = 6	4! = 24	5! = 120	6! = 720	7! = 5040

Wert	i! * li_g(i)		1 * 0	+	2 * 2	+	6 * 3	+	24 * 4	+	120 * 4	+	720 * 0	+	5040 * 1
		5638	= 0	+	4	+	18	+	96	+	480	+	0	+	5040

Abbildung 47: Beispiel 1; Herleitung der gesamten Eckpositionskoordinaten

```
EckKoord={3,2,1,0,6,7,4,5};
```

Position	i	0	1	2	3	4	5	6	7
Eck-Nummer	eck(i)	3	2	1	0	6	7	4	5
Links_grösser	li_g(i)		1	2	3	0	0	2	2
Wertigkeit	i!		1! = 1	2! = 2	3! = 6	4! = 24	5! = 120	6! = 720	7! = 5040

Wert	i! * li_g(i)		1 * 1	+	2 * 2	+	6 * 3	+	24 * 0	+	120 * 0	+	720 * 2	+	5040 * 2
		11543	= 1	+	4	+	18	+	0	+	0	+	1440	+	10080

Abbildung 48: Beispiel 2; Herleitung der gesamten Eckpositionskoordinaten

Gibt man auf den Positionen die Eck-Nummern rückwärts ein, also die sieben auf der null bzw. die null auf der sieben usw., so erhält man wiederum die maximale Anzahl der Möglichkeiten, *8!-1=40319.*

3.4 Positionen der Kanten

Die Kanten auf die Positionen zu verteilen ist wieder vollkommen analog zu der Methode der Ecken. Diese Koordinaten steigen rasch ins unermessliche, es sind Zahlen von *0 bis 12!-1=479'001'599.*

```
KantKoord={0,1,10,4,3,6,5,7,9,8,2,11};
```

Position	i	0	1	2	3	4	5	6	7	8	9	10	11
Kanten_Nr.	kant(i)	0	1	10	4	3	6	5	7	9	8	2	11
Links_grösser	li_g(i)		0	0	1	2	1	2	1	1	2	8	0
Wertigkeit	i!		1! = 1	2! = 2	3! = 6	4! = 24	5! = 120	6! = 720	7! = 5040	8! = 40320	9! = 362880	10! = 3628800	11! = 39916800

Wert	i! * li_g(i)		1 * 0	+	2 * 0	+	6 * 1	+	24 * 2	+	120 * 1	+	720 * 2	+	5040 * 1	+	40320 * 1	+	362880 * 2	+	3628800 * 8	+	39916800 * 0
		29803134	= 0	+	0	+	6	+	48	+	120	+	1440	+	5040	+	40320	+	725760	+	29030400	+	0

Abbildung 49: Beispiel 1; Herleitung der gesamten Kantenpositionskoordinaten

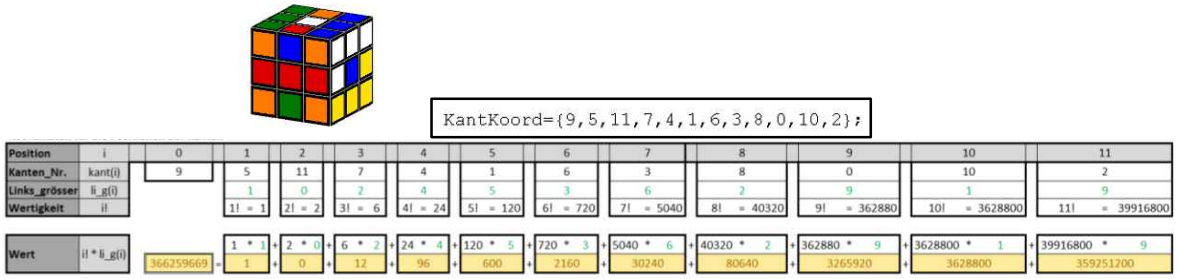


Abbildung 50: Beispiel 2; Herleitung der gesamten Kantenpositionskoordinaten

Doch diese Koordinaten für die Positionen der Kanten werden nicht vollständig gebraucht. Zu einem späteren Zeitpunkt werden zwar die Informationen über Positionen der Kanten benötigt, doch dann nur noch von acht Kanten. Die Gründe dafür werden zu einem späteren Zeitpunkt diskutiert. Hier wird die Herleitung für die Koordinaten der Positionen der acht Kanten aufgezeigt. Diese kommen unter der Annahme zu Stande, dass die vier ersten Kantenpositionen, also diese in der mittleren Scheibe nicht besetzt werden können. Es muss zusätzlich angenommen werden, dass die Kantenobjekte aus diesen mittleren Scheiben auch nicht auf den anderen Positionen auftauchen können. Durch diesen Umstand ist die Analogie zu den Eckpositionen noch grösser, denn aus kombinatorischer Sicht ist es wieder eine Permutation mit k Elementen und n Plätze.

```
40319: KantKoord={p,q,r,s,11 10 9 8 7 6 5 4};
40318: KantKoord={p,q,r,s,10 11 9 8 7 6 5 4};
40317: KantKoord={p,q,r,s,11 9 10 8 7 6 5 4};
40316: KantKoord={p,q,r,s,9 11 10 8 7 6 5 4};
40315: KantKoord={p,q,r,s,10 9 11 8 7 6 5 4};
40314: KantKoord={p,q,r,s,9 10 11 8 7 6 5 4};
40313: KantKoord={p,q,r,s,11 10 8 9 7 6 5 4};
40312: KantKoord={p,q,r,s,10 11 8 9 7 6 5 4};
40311: KantKoord={p,q,r,s,11 8 10 9 7 6 5 4};
40310: KantKoord={p,q,r,s,8 11 10 9 7 6 5 4};
40309: KantKoord={p,q,r,s,10 8 11 9 7 6 5 4};
40308: KantKoord={p,q,r,s,8 10 11 9 7 6 5 4};
40307: KantKoord={p,q,r,s,11 9 8 10 7 6 5 4};
                    .
                    .
                    .
```

```
                    .
                    .
                    .
12: KantKoord={p,q,r,s,4 6 7 5 8 9 10 11};
11: KantKoord={p,q,r,s,7 5 4 6 8 9 10 11};
10: KantKoord={p,q,r,s,5 7 4 6 8 9 10 11};
9: KantKoord={p,q,r,s,7 4 5 6 8 9 10 11};
8: KantKoord={p,q,r,s,4 7 5 6 8 9 10 11};
7: KantKoord={p,q,r,s,5 4 7 6 8 9 10 11};
6: KantKoord={p,q,r,s,4 5 7 6 8 9 10 11};
5: KantKoord={p,q,r,s,6 5 4 7 8 9 10 11};
4: KantKoord={p,q,r,s,5 6 4 7 8 9 10 11};
3: KantKoord={p,q,r,s,6 4 5 7 8 9 10 11};
2: KantKoord={p,q,r,s,4 6 5 7 8 9 10 11};
1: KantKoord={p,q,r,s,5 4 6 7 8 9 10 11};
0: KantKoord={p,q,r,s,4 5 6 7 8 9 10 11};
```

Code 14: Lexikografische Auflistung der möglichen Positionen von sieben Kanten

Die beiden Beispiele von den *Abbildungen 39 und 40* sind unter diesen beiden oben genannten Annahmen auch nicht mehr zu überprüfen, denn die vier ersten Kanten sind mit anderen Kanten besetzt, wie auch die vier Objekte auf anderen Positionen auftauchen. Dafür wird unten ein fiktives Beispiel aufgeführt, sodass eine Veranschaulichung für die spätere Implementierung vorhanden ist. Die Bedeutung dieser Koordinate wird zu einem späteren Zeitpunkt geklärt.

Position	i	0…3	4	5	6	7	8	9	10	11
Kanten_Nr.	kant(i)		8	4	6	5	7	10	11	9
Links_grösser	li_g(i)			1	1	2	1	0	0	2
Wertigkeit	i!			1! = 1	2! = 2	3! = 6	4! = 24	5! = 120	6! = 720	7! = 5040
Wert	i! * li_g(i)		10119	= 1 * 1 + 2 * 1 + 6 * 2 + 24 * 1 + 120 * 0 + 720 * 0 + 5040 * 2						
				= 1 + 2 + 12 + 24 + 0 + 0 + 10080						

Abbildung 51: fiktives Beispiel; Herleitung der gesamten Kantenpositionskoordinaten für sieben Kanten

Nun kann jeder Würfel mit einem Quartett von Zahlen beschrieben werden, das Aufschluss über die Position in der Reihe der möglichen Stellungen des Würfels gibt. Bei unseren beiden Beispielen ist dies:

I.	**Positionen der Ecken**	**5'638**
II.	**Orientierung der Ecken**	**792**
III.	**Positionen der Kanten**	**29'803'134**
IV.	**Orientierung der Kanten**	**463**
		$6.161595436243832 \cdot 10^{15}$

Abbildung 52: Beispiel 1, Hochrechnung der vier einzelnen Gesamtkoordinaten

I.	**Positionen der Ecken**	**11'543**
II.	**Orientierung der Ecken**	**2'075**
III.	**Positionen der Kanten**	**366'259'669**
IV.	**Orientierung der Kanten**	**1'375**
		$1.20622574469086593 75 \cdot 10^{19}$

Abbildung 53: Beispiel 1, Hochrechnung der vier einzelnen Gesamtkoordinaten

3.5 *Implementierung der vier Koordinaten*

Die Implementierung von diesem letzten Kapitel ist einfacher als das Erlangen des Verständnisses. Diese Erneuerung ist unter der zweiten Datei auf der CD abrufbar.

```
int EckKoordOritot;
int KantKoordOritot;
int EckKoordtot;
int KantKoordtot;
```

Code 15: vier Variablen werden erstellt

Im bestehenden Programm werden ausserhalb der Funktion *addMove()* vier Variabeln für die gesamten Koordinaten definiert.

```
int[] KantKoord2={0,0,0,0,0,0,0,0,0,0,0,0};
int[] EckKoord2={0,0,0,0,0,0,0,0};
```

Code 16: Hilfsarrays werden erstellt und initialisiert

Es braucht zudem je ein zusätzliches Feld für die Eck- bzw. Kantenpositionen. Diese beiden werden auch ausserhalb der Funktion *addMove()* definiert.

```
int [] fak={0,1,2,6,24,120,720,5040,40320,362880,3628800,39916800};
```

Code 17: Fakultäten werden in ein Array gespeichert

Ebenfalls ausserhalb wird ein Feld *fak{}* definiert, indem die Fakultäten von i an i-ter Position beinhaltet sind. Da wir die Fakultäten nur bis elf benötigen, ist es einfacher ein Feld zu definieren als eine Funktion für das Errechnen der Fakultät zu schreiben.

```
EckKoordOritot=0;
for(int i=0;i<=6;i++){
  EckKoordOritot+=EckKoordOri[i]*pow(3,i);
}
```

Code 18: Berechnung der Gesamtkoordinate der Eckorientierungen

```
KantKoordOritot=0;
for(int i=0;i<=10;i++){
  KantKoordOritot+=KantKoordOri[i]*pow(2,i);
}
```

Code 19: Berechnung der Gesamtkoordinate der Kantenorientierungen

Die vier Berechnungen der neuen gesamten Koordinaten sind in die Funktion *addMove()* implementiert. Sie werden also nach jeder Anwendung eines Zuges wieder neu berrechnet. Die Implementierung der Koordinaten für die Orientierung der Ecken und Kanten sind leicht nachvollziehbar und in diesem Kapitel bereits ausführlich erläutert. Zuerst wird die Gesamtzahl auf 0 gesetzt, danach wird in einer for-Schleife diese Gesamtzahl um die Wertigkeit, multipliziert mit der Orientierung auf der jeweiligen Position, erhöht.

```
for(int i=1;i<=7;i++){
  int grossals=0;
  for(int j=0;j<i;j++){
    if(EckKoord[i]<EckKoord[j]){
      grossals++;
    }
    EckKoord2[i]=grossals;
  }
}
EckKoordtot=0;
for(int j=0;j<=7;j++){
  EckKoordtot+=EckKoord2[j]*fak[j];
}
```

```
for(int i=1;i<=11;i++){
  int grossals=0;
  for(int j=0;j<i;j++){
    if(KantKoord[i]<KantKoord[j]){
      grossals++;
    }
    KantKoord2[i]=grossals;
  }
}
KantKoordtot=0;
for(int j=0;j<=11;j++){
  KantKoordtot+=KantKoord2[j]*fak[j];
}
```

Code 20: Berechnung der Gesamtkoordinaten der Eck- sowie Kantenpositionskoordinaten

In einem ersten Schritt werden die Objekte, welche grösser sind als das Objekt auf Position i, auf niedrigeren Positionen als i, gezählt. Dieses Zählen wird mit einer doppelt verschachtelten *for-Schleife* bewerkstelligt. Der Zähler *grossals* wird bei jedem Zutreffen der Bedingung um

eins inkrementiert. Ist die innere Abbruchbedingung *(j≥i)* erreicht, wird die gefüllte Variabel *grossals* in das Feld i gespeichert. Damit wäre der erste Teil abgeschlossen. Der zweite ist analog zu den Koordinaten für die Orientierungen. Es wird die Summe von der Multiplikation von der Wertigkeit *(fak[j])* und dem ebengerade errechneten Wert *grossals* auf jeder Position berechnet.

Auf der Datei_2 finden Sie diese Implementierung. Verändern Sie wiederum die Anwendungen der einzelnen Züge und deren Auswirkungen auf die einzelnen Koordinaten. Ebenfalls auf der CD beigelegt sind die EXCEL-Dateien, in denen die Ecken- und Kantenpositionen bzw. deren Orientierungen manuell eingegeben werden können. Darin hat man mehr Kontrolle über die Koordinaten, ebenso zeigen sie Schritt für Schritt auf, wie diese Koordinaten zu Stande kommen.

4 Äquivalente Stellungen

Diese riesige Anzahl an möglichen Konstellationen ($4.3 \cdot 10^{19}$) verunmöglicht es, auch mit Hochleistungscomputer innert nutzbarer Zeit durch Kombinationen von Zügen die Zugfolge zu finden, die einen beliebigen Würfel in eine andere Stellung bzw. in die finale Stellung zu bringen. Die Aufgabe besteht eigentlich darin, eine Stellung S_1 aus der Menge der möglichen Stellungen M_0 in eine andere Stellung S_2 zu bringen. Dies ist mit den Basisdrehzüge D_0 = *{V,O,H,U,R,L}* zu erreichen. Die unbekannte, endliche Zugfolge $d_1, d_2, d_3, .. d_n$, die S_1 in S_2 überführt, ist also gesucht. Diese Zugfolge $d_1, d_2, d_3, .. d_n$ ist nicht eindeutig, es gibt viele Lösungen, allerdings mit unterschiedlicher Anzahl von Zügen. Lange war die maximale Zahl n, mit der jede Stellung in eine andere überführt werden kann, unbekannt. Im Juli 2010 gelang es einer Gruppe von Forschern mit enormem Rechenaufwand diese kleinstmögliche Zahl auf 20 festzulegen. Es ist also prinzipiell möglich, jede Stellung in maximal 20 Zügen in eine andere zu überführen. Diese Zugfolge zu finden ist aber extrem schwierig und dieses Projekt stellt auch sicherlich nicht den Anspruch, diese maximale Anzahl nicht zu überschreiten.

Ein Ansatz, um die riesige Anzahl möglicher Stellungen in M_0 zu umgehen, besteht darin, Untergruppen zu definieren, die Teilmengen von M_0 sind. Wenn man z.B. M_1 definiert als alle möglichen Stellungen, die durch die Anwendung von $D_1 = \{O,U,V^2,H^2,R^2,L^2\}$ auf den finalen Würfel erreichbar sind, so erreicht man eine Zahl von ($8!^2 \cdot 24/2 =$) $1.95 \cdot 10^{10}$. Um eine beliebige Stellung aus M_0 mit der Anwendung von D_0 in eine Stellung von M_1 zu bringen, gibt es noch ($2^{11} \cdot 3^7 \cdot 495 =$) $2.21 \cdot 10^9$ verschiedene mögliche Stellungen.

4.1 Menge M_1

In dieser Menge sind alle möglichen Stellungen, die durch eine beliebige Zugfolge von $D_1 = \{O,U,V^2,H^2,R^2,L^2\}$ auf den finalen Würfel angewendet, erreicht werden können. Das heisst also, die Flächen auf der oberen und unteren Schicht sind immer entweder gelb oder weiss gefärbt. Diesen Umstand kann man auch anders beschreiben, denn alle Orientierungen, sowohl bei den Ecken, wie auch bei den Kanten, sind 0.

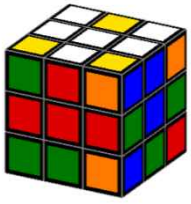


Abbildung 54: Stellung aus M_1

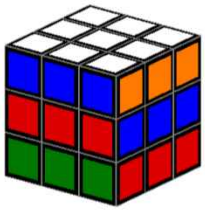


Abbildung 55: Stellung aus M_1

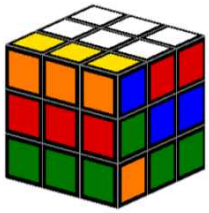


Abbildung 56: Stellung aus M_1

Ausserdem werden die vier Kanten in der mittleren Scheibe (unsere Kanten von 0 bis 3) nur innerhalb dieser Scheibe verschoben. Auch ihre Orientierung bleibt konstant bei null, doch ihre Reihenfolge relativ zueinander muss noch nicht korrekt sein.

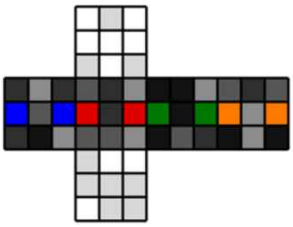


Abbildung 57: vier mittlere Kanten

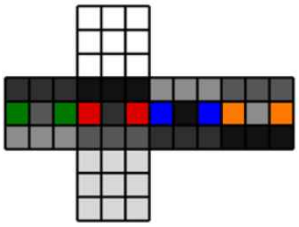


Abbildung 58: vier mittlere Kanten

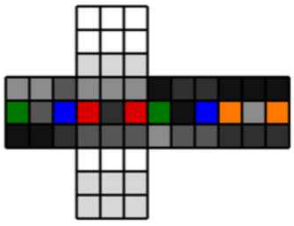


Abbildung 59: vier mittlere Kanten

Eine Stellung aus M_1 ist also erreicht, wenn alle Ecken und Kanten richtig orientiert sind und alle vier mittleren Kantenstücke sich in der mittleren Scheibe befinden.

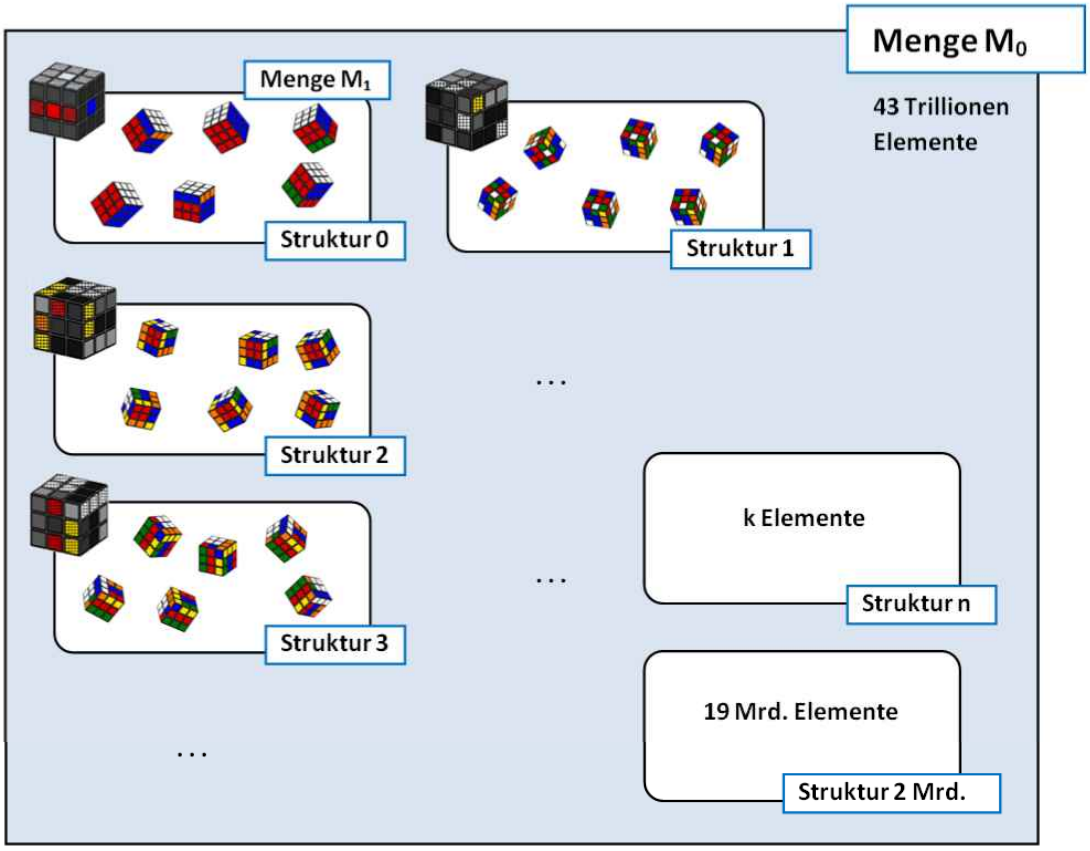


Abbildung 60: Schema für Aufteilung der Menge M_0 in Strukturtypen

Die *Abbildung 60* zeigt auf, was die Problemstellung ist, bzw. wie sie angegangen wird. Die Menge M_0 kann in ungefähr 2 Milliarden Teilmengen aufgespalten werden, die alle Stellungen vom selben Strukturtyp enthalten. In jeder dieser Teilmenge sind ungefähr je 19 Milliarden Stellungen vorhanden. Als erster Schritt wird das Ziel sein, den Strukturtyp zu finden. Einen grossen Teil haben wir mit dem Errechnen der Koordinaten bereits erledigt. Danach müssen

wir eine Zugfolge finden, die diesen Strukturtypen in den Strukturtyp 0 überführt. Dieser ist gerade die Menge M_1. In dieser Menge haben wird dann in der zweiten Phase noch 19 Milliarden verschiedene Stellungen zu untersuchen, wobei dann die Strukturtypen wieder aufgelöst werden.

Für das weitere Vorgehen bedeutet dies also, dass wir vorerst „nur" noch gut 2 Milliarden verschiedene möglichst optimale Zugfolgen suchen, die eine beliebige Stellung in M_1 überführt.

4.2 Äquivalente Strukturtypen

Um einen beliebigen Würfel in die Menge M_1 zu überführen, kann man Stellungen in der Menge M_0 also als äquivalent betrachten. Sie sind natürlich nicht dieselben Stellungen, sondern haben dieselbe Struktur von den Orientierungen. Es existieren theoretisch, im Sinne der Strukturäquivalenz, viel weniger mögliche Stellungen, da auf viele verschiedene Stellungen dieselbe Zugfolge angewendet, den Würfel in M_1 überführt.

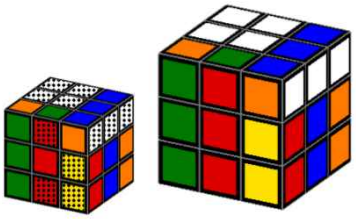


Abbildung 61: nach Anwendung von {U,O,V,R}

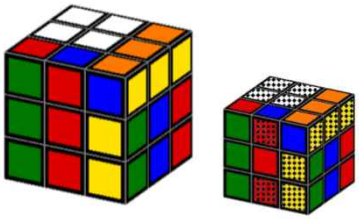


Abbildung 62: nach Anwendung von $\{U^2,O',V',R\}$

Betrachten wir zwei Beispiele. Auf einen finalen Würfel wird die Zugfolge {U,O,V,R} angewendet, auf einen zweiten $\{U^2,O',V',R\}$.

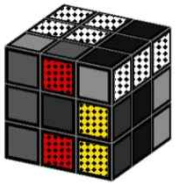


Abbildung 63: Struktur bezüglich Orientierung

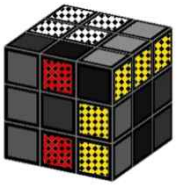


Abbildung 64: Struktur bezüglich Orientierung

Bei Betrachtung der Orientierungen der Ecken und Kanten fällt auf, dass diese beiden Würfel die gleiche Struktur haben. Wenn wir einen Zug auf diese beiden Stellungen anwenden, so bleibt die Äquivalenz der Struktur vorhanden:

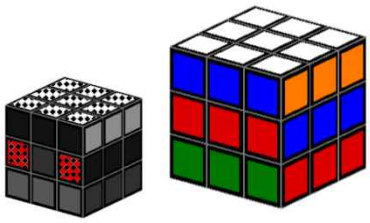


Abbildung 65: nach {R‘,V‘}

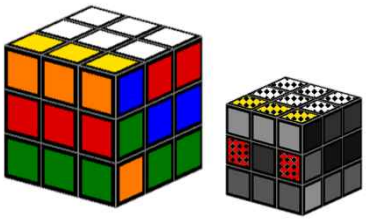


Abbildung 66: nach {R‘,V‘}

Bei der Anwendung von $\{R',V'\}$ auf beide bleibt nicht nur die Struktur erhalten, sonder es werden Stellungen aus M_1 erreicht. Diese beiden Beispiele sind konstruiert und nicht zufällig ausgewählt worden, doch es gibt ungefähr 19 Milliarden äquivalente Stellungen dieses Strukturtyps.

Man kann eine Strukturäquivalenz von Stellungen bezüglich verschiedener Koordinaten und Kriterien ausdrücken. Je nachdem welche Kriterien bei der Struktur zutreffen müssen, variiert die Mächtigkeit und die Anzahl der Gruppen. Wenn man den Würfel sehr detailliert beschreibt, so gibt es viele verschiedene Äquivalenzstrukturgruppen mit wenigen enthaltenen Stellungen. Auf der anderen Seite kann man den Würfel sehr ungenau beschreiben. Dann wächst die Anzahl an Stellungen, die in einer der wenigen Strukturgruppen sind.

5 Auswirkungs-Tabellen x_1, y_1 und z_1

5.1 Absicht der Generierung

Im letzten Abschnitt waren „Informationen“ genannt worden. Welche Daten werden nun benötigt? Für welchen Zweck braucht man diese? Da der Abstraktheitsgrad doch schon höhere Sphären erreicht hat, sollte ein Überblick für das weitere Vorgehen gegeben werden.

Was wir bis jetzt besprochen haben, dient lediglich als Grundlage um den Würfel zu beschreiben. Wir müssen uns vor Augen halten, was das Ziel ist. Dies ist die Suche nach einer möglichst optimalen Zugfolge, die eine beliebige Stellung des Würfels in die finale 0-Position überführt. Um diese Suche (1b) starten zu können, ist es, wie bereits erwähnt, sinnvoll, externe Daten zu speichern, die Aussagen darüber machen, wie die spezifischen Seitenzüge Auswirkungen auf die gesamten Koordinaten ausüben.

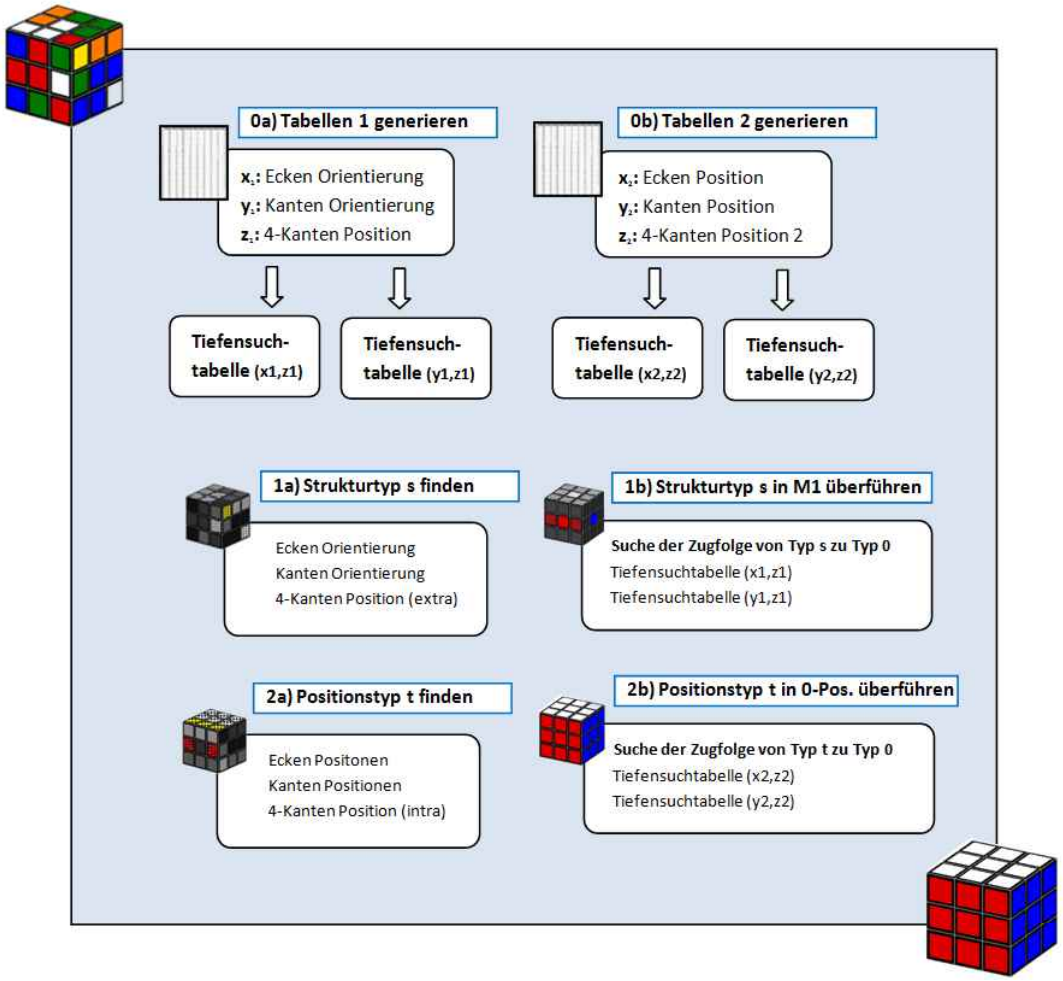


Abbildung 67: Ablaufschema für die Überführung eines beliebigen Würfels in die finale Stellung

Um diese gut 2 Milliarden Zugfolgen in der ersten Phase zu finden, benötigt man eine Menge von Informationen. Diese Informationen immer wieder zu berechnen benötigt viel Zeit, darum

bedient man sich externer Tabellen. Diese muss man einmal generieren, sodass man die spezifischen Daten bei Gebrauch nur noch herauslesen kann.

Wie in *Kapitel 2* gesehen, verändert jede Zugfolge die Koordinaten der Ecken und Kanten auf eine zugspezifische Art. Diese Veränderung kann man auch für die gesamten Koordinaten herausfinden. Wenn man also z.B. einen Strukturtyp s hat und man wendet einen Seitenzug Vorne an, so werden alle 19 Milliarden äquivalenten Stellungen zum selben Strukturtyp s'. Diese Veränderungen versuchen wir nun zu eruieren.

5.2 Auswirkungen auf die gesamten Koordinaten

Jeder Seitenzug übt z.B. eine konstante Veränderung der Orientierung auf die Ecken und Kanten aus. Die resultierende Orientierung ist zwar abhängig von der vorhandenen Orientierung, doch die Veränderung ist konstant. Bei den Auswirkungen auf die gesamten Koordinaten ist diese Veränderung nicht mehr konstant. Das bedeutet, es sind keine Regelmässigkeiten feststellbar bei Anwendungen der verschiedenen Seitenzüge. Um trotzdem diese Informationen zu erhalten, speichern wir ganze Tabellen ab, die die Daten der gesamten Koordinaten aufzeigen in Abhängigkeit der Startgesamtkoordinate und dem Seitenzug.

Tabelle für Gesamtorientierung der Ecken (x_1)

Die Auswirkungen auf die gesamten Koordinaten (0 bis 43 Trillionen) können zwar eruiert werden, jedoch sind diese nicht sinnvoll zu berechnen, da sie viel zu grosse und zu viele Zahlen beinhaltet. Die Auswirkungen werden auf dem Niveau der verschiedenen spezifischen Gesamtkoordinaten für die Orientierung der Ecken und Kanten sowie deren Positionen erstellt. Die Multiplikation kann im Nachhinein immer noch ausgeübt werden.

	V	V2	V'	O	O2	O'	H	H2	H'	U	U2	U'	R	R2	R'	L	L2	L'
0	572	0	572	0	0	0	1503	0	1503	0	0	0	258	0	258	946	0	946
1	575	243	653	27	9	3	1495	55	775	1459	163	487	16	7	250	784	729	892
2	569	486	491	54	18	6	1487	29	47	731	83	245	503	5	269	865	1458	919
3	86	81	570	1	27	9	1497	57	777	1461	165	489	24	6	492	949	3	949
4	89	324	651	28	36	12	1489	31	49	733	85	247	511	4	511	787	732	895
5	83	567	489	55	45	15	1508	5	1508	5	5	5	269	2	503	868	1461	922
6	329	162	571	2	54	18	1491	33	51	735	87	249	492	3	24	952	6	952
7	332	405	652	29	63	21	1510	7	1510	7	7	7	250	1	16	790	735	898
8	326	648	490	56	72	24	1502	62	782	1466	170	494	8	8	8	871	1464	925
9	581	9	581	3	1	27	1521	783	774	1467	171	495	15	249	243	955	9	955
10	584	252	662	30	10	30	1513	757	46	739	91	253	502	247	262	793	738	901
11	578	495	500	57	19	33	1532	731	1505	11	11	11	260	245	254	874	1467	928
12	95	90	579	4	28	36	1515	759	48	741	93	255	510	246	504	958	12	958
13	98	333	660	31	37	39	1534	733	1507	13	13	13	268	244	496	796	741	904
14	92	576	498	58	46	42	1526	788	779	1472	176	500	26	251	488	877	1470	931
15	338	171	580	5	55	45	1536	735	1509	15	15	15	249	243	9	961	15	961
16	341	414	661	32	64	48	1528	790	781	1474	178	502	7	250	1	799	744	907
17	335	657	499	59	73	51	1520	764	53	746	98	260	494	248	20	880	1473	934
18	590	18	590	6	2	54	1458	1485	45	747	99	261	501	489	264	964	18	964
19	593	261	671	33	11	57	1477	1459	1504	19	19	19	259	487	256	802	747	910
20	587	504	509	60	20	60	1469	1514	776	1478	182	506	17	494	248	883	1476	937
21	104	99	588	7	29	63	1479	1461	1506	21	21	21	267	486	498	967	21	967
22	107	342	669	34	38	66	1471	1516	778	1480	184	508	25	493	490	805	750	913
23	101	585	507	61	47	69	1463	1490	50	752	104	266	512	491	509	886	1479	940
24	347	180	589	8	56	72	1473	1518	780	1482	186	510	6	492	3	970	24	970
25	350	423	670	35	65	75	1465	1492	52	754	106	268	493	490	22	808	753	916
26	344	666	508	62	74	78	1484	1466	1511	26	26	26	251	488	14	889	1482	943
27	599	27	599	9	3	1	36	54	756	1485	189	513	42	33	276	947	81	1675
28	602	270	680	36	12	4	28	28	28	757	109	271	529	31	295	785	810	1621
29	596	513	518	63	21	7	47	2	1487	29	29	29	287	29	287	866	1539	1648
30	113	108	597	10	30	10	30	30	30	759	111	273	537	30	537	950	84	1678
31	116	351	678	37	39	13	49	4	1489	31	31	31	295	28	529	788	813	1624

Abbildung 68: Ausschnitt aus Tabelle für Orientierung der Ecken

In *Abbildung 68* ist ein kleiner Ausschnitt der Tabelle für die Orientierung der Ecken dargestellt. In der ersten Spalte sind die Startorientierungskoordinaten. Bei der Ausübung

einer der 18 Seitenzüge wird die Orientierungskoordinate für die Ecken in eine neue überführt, welche abgespeichert wird. Um konkreter zu werden, betrachten wir die Stellung, die nach der Zugfolge {R,U,V} auf die 0-Stellung resultiert:

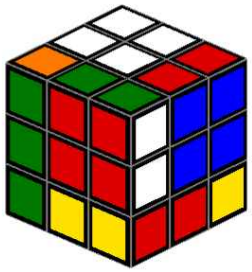


Abbildung 69: Ausgangslage; Orientierung der Ecken 1796

	V	V2	V'	O	O2	O'	H	H2	H'	U	U2	U'	R	R2	R'	L	L2	L'
1795	1635	1795	1635	1813	1819	1821	1849	1129	1075	1876	1390	1309	1558	1798	2026	1741	1013	1067
1796	1629	2038	1473	1840	1828	1824	1841	1103	347	1148	1310	1067	2045	1796	2045	1822	1742	1094
1797	1875	1633	1555	1787	1837	1827	1851	1131	1077	1878	1392	1311	1539	1797	1539	1906	287	1124

Abbildung 70: Ausgangslage; Ausschnitt aus Tabelle der Eckorientierungen

Nach dieser Anwendung hat man eine Gesamtorientierung der Ecken von 1796. Wenn wir nun in der Tabelle unter dieser Orientierung nachschauen, so sehen wir alle potentiellen Gesamtorientierungen der Ecken, je nachdem, welcher Seitenzug wir als nächstes ausführen werden.

Wir entscheiden uns nun für die Anwendung einer Inversen-Vorne-Drehung und erhalten eine neue Gesamtorientierung der Ecken (1473). Wiederum springen wir zu der Zeile mit dieser neuen Gesamtzahl.

Abbildung 71: nach Schritt 1 {V'}; Orientierung der Ecken 1473

	V	V2	V'	O	O2	O'	H	H2	H'	U	U2	U'	R	R2	R'	L	L2	L'
1472	1550	2034	1956	1516	1504	1500	1535	752	1481	176	500	14	1727	1703	1955	850	1472	850
1473	1796	1629	2038	1463	1513	1503	1518	780	24	906	582	258	1950	1704	1476	934	17	880
1474	1799	1872	2119	1490	1522	1506	1537	754	1483	178	502	16	1708	1702	1468	772	746	826

Abbildung 72: nach Schritt 1; Ausschnitt aus Tabelle der Eckorientierungen

Nach der Anwendung von U' erhalten wir wieder eine neue Stellung (258).

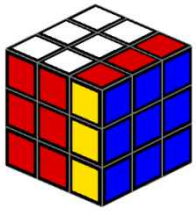


Abbildung 73: nach Schritt 2 {U'}; Orientierung der Ecken: 258

	V	V2	V'	O	O2	O'	H	H2	H'	U	U2	U'	R	R2	R'	L	L2	L'
257	173	577	501	301	289	285	1760	1004	293	743	824	338	506	257	506	1120	1713	1174
258	419	172	583	248	298	288	1770	1032	1023	1473	906	582	0	258	0	1204	258	1204
259	422	415	664	275	307	291	1762	1006	295	745	826	340	487	256	19	1042	987	1150

Abbildung 74: nach Schritt 2; Ausschnitt aus Tabelle der Eckorientierungen

Auch hier sehen wir alle folgenden Stellungen, in welche die aktuelle Stellung mit einem Seitenscheibenzug überführt werden kann. Wenn wir in der *Abbildung 74* in der Spalte des inversen Rechts-Zuges die designierte Stellung betrachten, können wir feststellen, dass bei dieser genannten Anwendung der Würfel in eine 0-Position bezüglich der Orientierungen der Ecken überführt wird.

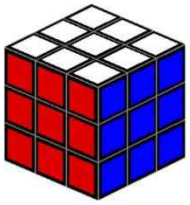


Abbildung 75: nach Schritt 3 {R'}; Orientierung der Ecken: 0

Nach der Anwendung von R' sehen wir nun etwas ganz Erfreuliches. Wir haben die 0-Position erreicht. (Nach den Tabellen müsste eigentlich nur die Eckenorientierung eine 0-Position sein.)

Dieses Beispiel soll einerseits illustrieren, welche Daten in der Tabelle gespeichert werden, anderseits gibt es einen Vorgeschmack auf den nächsten Schritt. Wir wendeten zuerst einen V'-Zug bei der Orientierung von 1796 an. Warum dieser den anderen 17 Zügen vorgezogen wurde, kommt daher zu Stande, dass wir die Zugfolge {R,U,V} kennen, und als Beispiel die inverse Zugfolge anwenden konnten.

Doch bei der Suche nach einer 0-Position bezüglich den Koordinaten x_1, y_1 und z_1 müssen wir sozusagen im Heuhaufen stochern, um eine geeignete Zugfolge zu finden. Das *Kapitel 6* wird ein Ansatz dazu aufzeigen, um die Suche starten zu können.

Tabelle für Gesamtorientierung der Kanten (y_1)

Zunächst werden aber noch die Tabellen für die Gesamtorientierungen der Kanten benötigt. Die Analogie zu dem ebengerade beschriebenen ist 1:1, daher wird hier darauf verzichtet, noch einmal auf das Prinzip einzugehen. Auch hier werden einfach wieder zu jeder Orientierungsmöglichkeit alle 18 Züge angewendet und die neue Orientierung der Kanten in eine Tabelle gespeichert.

Tabelle für Gesamtposition der mittleren Kanten (z_1)

Bis jetzt kennen wir vier verschiedene Koordinaten. Eine weitere wurde schon mehrmals erwähnt: die Positionen für die vier mittleren Kanten. Um eine Stellung aus der Menge M_1 zu erreichen, müssen sich alle vier mittleren Kanten in dieser mittleren Scheibe befinden. Wie bei den anderen vier Koordinaten benötigen wir die Kombinatorik. Anders als bei den Positionen der Ecken und Kanten sind die Positionen, auf denen sich die vier mittleren Kanten befinden unwesentlich, das heisst, es ist egal, in welcher Reihenfolge die vier mittleren Kanten auf ihren vier Plätzen angeordnet werden, ebenso die restlichen acht Kanten auf ihren acht Positionen. Die totale Anzahl an Möglichkeiten, wie vier Kanten auf diesen Plätzen zu liegen kommen kann man mit einer Menge n Kanten vergleichen, von denen k ausgewählt werden. Es ist also im kombinatorischen Sinne eine Kombination ohne Wiederholung. Es gibt n = 12 Kanten, davon werden k = 4 ausgewählt. Mit Hilfe der Binomialkoeffizienten kann die Anzahl von Kombinationen errechnet werden.

$$K_n{}^{(k)} = \binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

$$K_{12}{}^{(4)} = \binom{12}{4} = \frac{12!}{4!\,(12-4)!} = \frac{12!}{4! \cdot 8!} = 495$$

Um eine konkrete Stellung dieser vier Kanten in eine Reihe einordnen zu können, kann man sich wiederum die lexikografische Auflistung dieser 495 Möglichkeiten zu Gemüte führen.

```
0: KantKoord={0,1,2,3,4,5,6,7,8,9,10,11};
1: KantKoord={0,1,2,4,3,5,6,7,8,9,10,11};
2: KantKoord={0,1,2,4,5,3,6,7,8,9,10,11};
3: KantKoord={0,1,2,4,5,6,3,7,8,9,10,11};
4: KantKoord={0,1,2,4,5,6,7,3,8,9,10,11};
5: KantKoord={0,1,2,4,5,6,7,8,3,9,10,11};
6: KantKoord={0,1,2,4,5,6,7,8,9,3,10,11};
7: KantKoord={0,1,2,4,5,6,7,8,9,10,3,11};
8: KantKoord={0,1,2,4,5,6,7,8,9,10,11,3};
9: KantKoord={0,1,4,2,3,5,6,7,8,9,10,11};
10: KantKoord={0,1,4,2,5,3,6,7,8,9,10,11};
11: KantKoord={0,1,4,2,5,6,3,7,8,9,10,11};
12: KantKoord={0,1,4,2,5,6,7,3,8,9,10,11};
                    .
                    .
                    .
```

```
                    .
                    .
                    .
482: KantKoord={4,5,6,7,8,9,0,1,2,10,11,3};
483: KantKoord={4,5,6,7,8,9,0,1,10,2,3,11};
484: KantKoord={4,5,6,7,8,9,0,1,10,2,11,3};
485: KantKoord={4,5,6,7,8,9,0,1,10,11,2,3};
486: KantKoord={4,5,6,7,8,9,0,10,1,2,3,11};
487: KantKoord={4,5,6,7,8,9,0,10,1,2,11,3};
488: KantKoord={4,5,6,7,8,9,0,10,1,11,2,3};
489: KantKoord={4,5,6,7,8,9,0,10,11,1,2,3};
490: KantKoord={4,5,6,7,8,9,10,0,1,2,3,11};
491: KantKoord={4,5,6,7,8,9,10,0,1,2,11,3};
492: KantKoord={4,5,6,7,8,9,10,0,1,11,2,3};
493: KantKoord={4,5,6,7,8,9,10,0,11,1,2,3};
494: KantKoord={4,5,6,7,8,9,10,11,0,1,2,3};
```

Code 21: Lexikografische Auflistung der möglichen Positionen der ersten vier Kanten

Durch das Einordnen einer konkreten Stellung in diese Reihenfolge bekommt jede Kombinationen von den Konstellationen der vier mittleren Kantenstücke eine gesamte Positionszahl der 4-Kantenmittelstücke. Auf eine Erklärung der Implementierung wird an dieser Stelle verzichtet. Unter der Datei_3 ist diese fünfte Koordinate auch berechnet bzw. implementiert.

5.3 Implementierung der Generierung der Tabellen

Um diese drei Tabellen zu generieren, läuft es eigentlich bei allen drei analog ab. Es wird eine Stellung p bezüglich einer der drei Gesamtkoordinaten (x1,y1 oder z1) erstellt. Auf diese Stellung werden alle 18 Seitenzüge angewendet und die neue resultierende Gesamtkoordinate in der Tabelle gespeichert. Das Prinzip dieser Implementierung wird am Beispiel der Orientierungskoordinaten der Ecken erläutert.

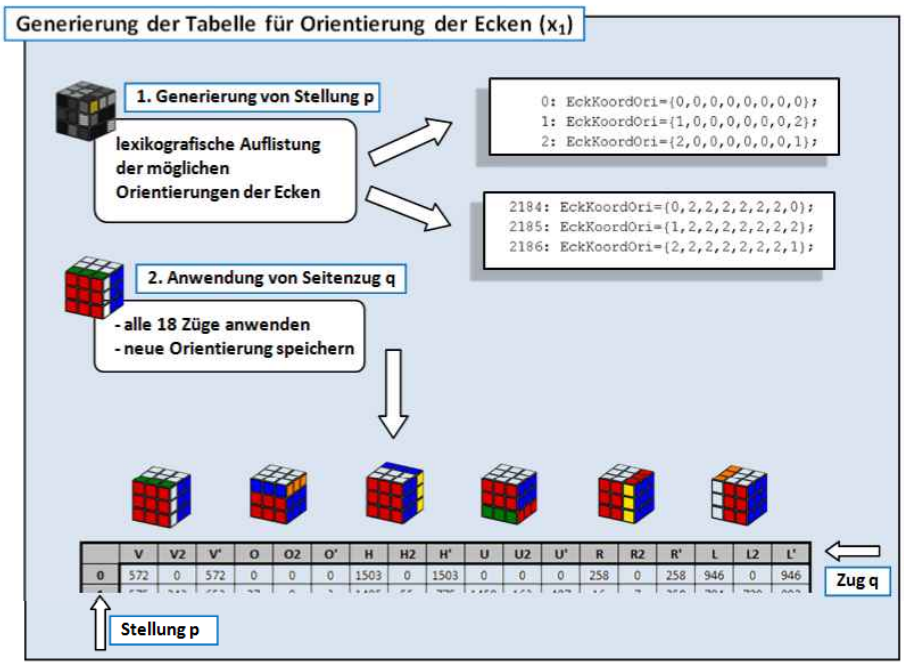


Abbildung 76: Ablaufschema für Generierung der Tabellen

Generierung der gesuchten Stellung p

Um eine lexikografische Auflistung der möglichen Orientierungen zu erstellen kann man eine siebenfache verschachtelte for-Schleife schreiben, wobei jeder Zähler von null bis zwei läuft.

```
void moveTableEckOri(){
  for(int i=0;i<=2;i++){
    for(int j=0;j<=2;j++){
      for(int k=0;k<=2;k++){
        for(int l=0;l<=2;l++){
          for(int m=0;m<=2;m++){
            for(int n=0;n<=2;n++){
              for(int o=0;o<=2;o++){
                int oritott=o*1+n*3+m*9+l*27+k*81+j*243+i*729;
                int p=(3-((o+n+m+l+k+j+i)%3))%3;
                              .
                              .
                              .
              }
            }
          }
        }
      }
    }
  }
}
```

Code 22: Generierung der Stellung p

Die Orientierung der achten Ecke ist durch die Modulodivision durch drei determiniert, muss also nur noch berechnet werden. Ebenfalls kann die Gesamtorientierung über die Wertigkeiten-Multiplikation mit den einzelnen Orientierungen auf jeder Eckposition berechnet werden.

```
EckKoordOri[0]=o;
EckKoordOri[1]=n;
EckKoordOri[2]=m;
EckKoordOri[3]=l;
EckKoordOri[4]=k;
EckKoordOri[5]=j;
EckKoordOri[6]=i;
EckKoordOri[7]=p;
```

Code 23: Stellung p wird gespeichert

```
oritott: EckKoordOri={o,n,m,l,k,j,i,p};
```

Code 24: resultierendes Array von Stellung p

In diese verschachtelte for-Schleife wird nun die konkrete Stellung in das Feld *EckKoordOri[]* übergeben.

Anwendung der Seitenzüge q

In diese iterative Verkettung und den konstruierten Orientierungen werden nun der Reihe nach die Seitenzüge angewendet.

```
for(int q=0;q<=5;q++){
  addMove(q);
  moveTableEckKoord[oritott][3*q]=EckKoordOritot;
  addMove(q);
  moveTableEckKoord[oritott][3*q+1]=EckKoordOritot;
  addMove(q);
  moveTableEckKoord[oritott][3*q+2]=EckKoordOritot;
  addMove(q);
}
```

Code 25: Seitenzüge q werden auf Stellung p angewendet

Die Funktion *addMove()* wird durch eine weitere *for-Schleife* mit dem *Laufparameter q* aufgerufen. Der *Parameter q* bestimmt über die Art des Zuges, also 0 = Vorne, 1 = Oben, etc. Die Variabel q beginnt bei 0, also mit einem Vorne-Zug. Es wird zuerst ein einfacher Zug ausgeführt und die neue Gesamtorientierungskoordinate von der Funktion *addMove()* zurückgegeben und in das Feld *moveTableEckKoord[p=oritott][0]* gespeichert. Auf diese Stellung wird ein zweiter einfacher Zug der gleichen Art angewendet, sodass relativ zur konstruierten *Stellung p* ein zweifacher Zug der *Art q* angewendet wurde. Diese neue Stellung wird nun in der zweiten Spalte des Arrays *moveTableEckKoord[p][1]* gespeichert. Nachdem noch ein weiterer einfacher Zug der *Art q* vollführt wurde und der zurückgegebene Wert in das *Feld[2]* gespeichert wurde, wird noch ein vierter einfacher Zug angewendet, um die konstruierte *Stellung p* wieder zu erreichen. Diese neue Koordinate wird selbstverständlicherweise nicht in das Array gespeichert. Danach wird der *Zähler q* um eins inkrementiert, sodass die Seitenzüge Oben, Oben 2 usw. angewendet werden können.

Diese Anwendung der 18 Seitenscheibenzüge wird auf alle 2187 mögliche Stellungen gemacht. Ist dies erreicht, wird das Array extern in einer *.txt* Datei abgespeichert und kann bei Gebrauch wieder in ein neues Array geladen werden.

Das ist das Prinzip für die Implementierung zur Generierung der Tabellen. Dies muss nicht nur für die Orientierungen der Ecken geschrieben werden, sondern auch für die Orientierungen der Kanten sowie für die Koordinaten für die vier mittleren Kantenpositionen.

	mögliche Stellungen		Anzahl Einträge
x_1: Ecken Orientierung	3^7	2187	39‘366
y_1: Kanten Orientierung	2^{11}	2048	36‘864
z_1: 4-Kanten Position	$\binom{12}{4}$	495	8‘910

Sind alle drei Tabellen erstellt, kann man zum nächsten Schritt übergehen. Dieser Schritt beruht auf einer simplen Idee. Mit den Informationen, die wir erstellt haben, können wir nun aussagen, welche Gesamtkoordinaten nach den Anwendungen von verschiedenen Seitenzügen resultieren. Im nächsten Kapitel wird der umgekehrte Weg verfolgt. Wir beginnen bei der Ursprungsstellung und wenden alle 18 Seitenscheibenzüge an und kennzeichnen den Weg, den wir genommen haben.

Die Generierung dieser drei Tabellen ist auf der CD unter den Dateien_4 aufgeführt. Es ist nötig, dass in der obersten Zeile jeweils der Pfad für den Ort der externen Tabellen eingegeben wird.

6 Tiefensuchtabellen generieren

Die Auswirkungen durch Seitenscheibenzüge sind nun in den drei Tabellen für die Koordinaten x_1, y_1 und z_1 einzeln gespeichert. Der Strukturtyp (also einer dieser gut 2 Mrd. verschiedenen Gruppierungen von äquivalenten Strukturen) ist mit diesen drei Koordinaten eindeutig beschrieben. Das Ziel bleibt immer noch, diesen konkreten Strukturtypen in eine Stellung aus der Menge M_1 überführen zu können.

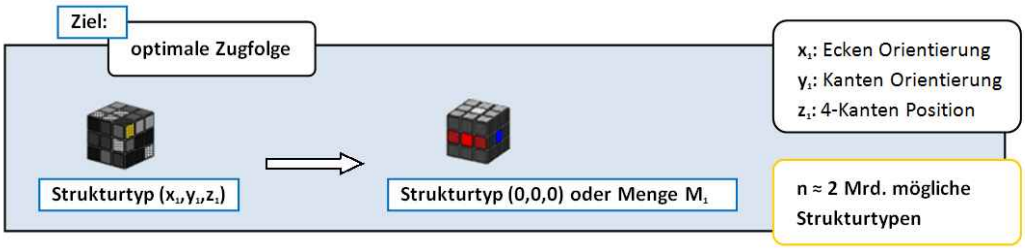


Abbildung 77: Aufgabenstellung

Doch um diese Zugfolge finden zu können, generieren wir noch eine Tabelle, die in jeder Zeile die verschiedenen Strukturtypen auflistet und in den Spalten die Anzahl Züge, die benötigt werden um den Strukturtyp (0,0,0) zu erreichen.

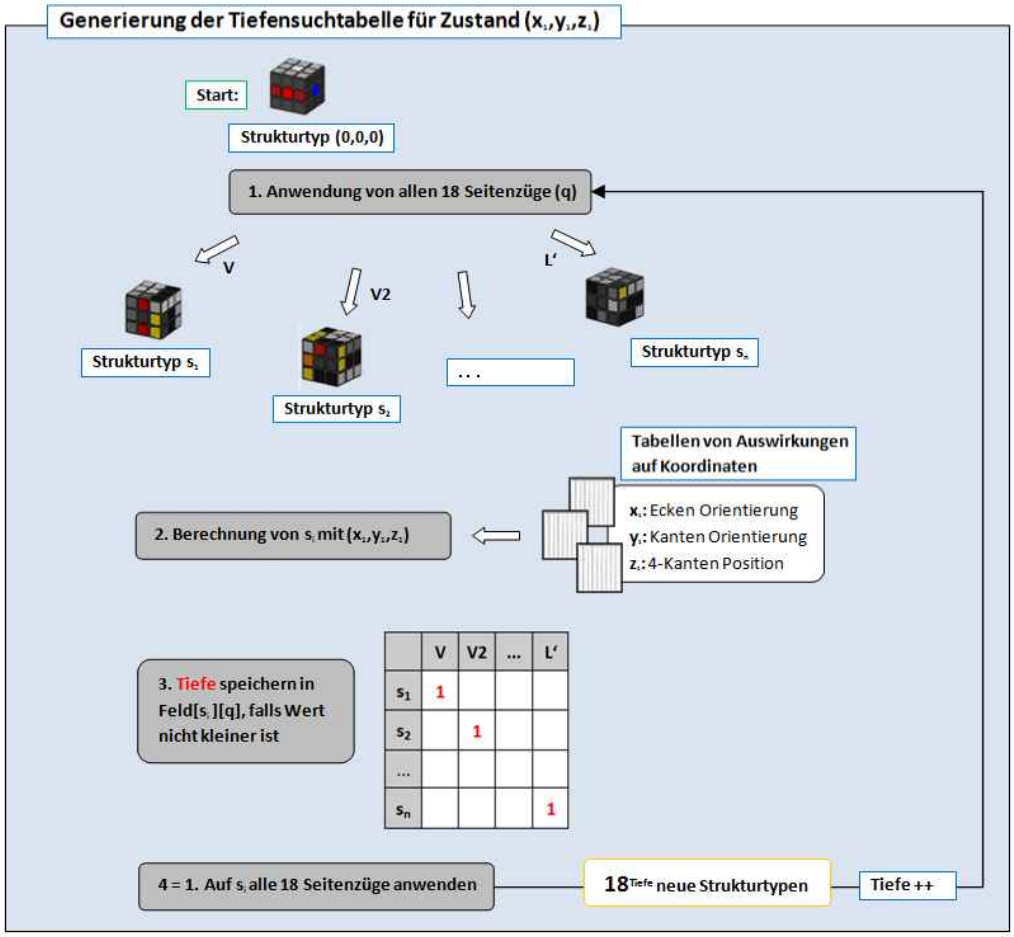


Abbildung 78: Ablaufschema für Generierung der Tiefensuchtabelle

Diese Tabelle wird rückwärts generiert und der gegangene Weg wird gekennzeichnet. Bei der späteren Suche kann dann ein optimaler inverser Weg herausgesucht werden.

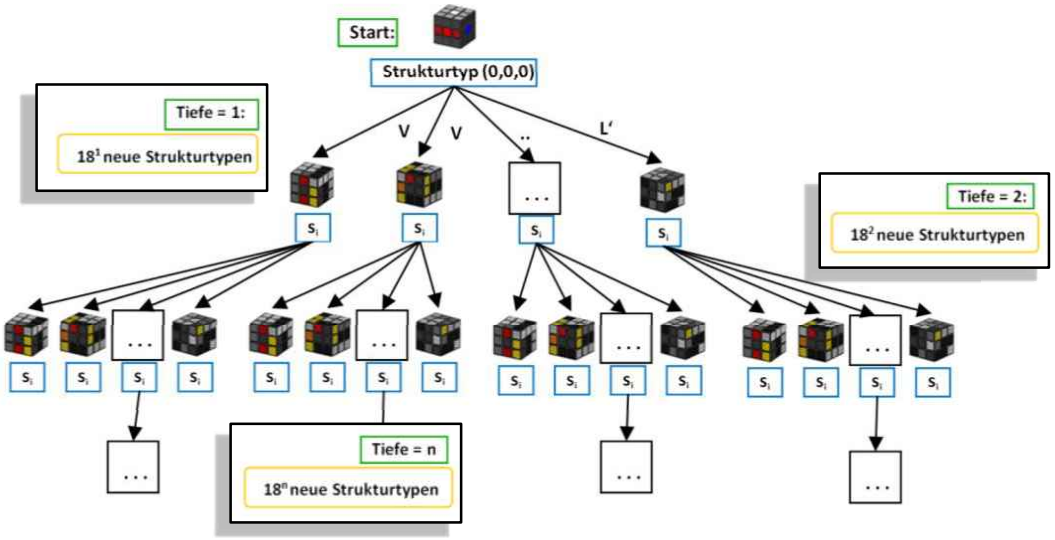


Abbildung 79: Baumstruktur bei Generierung der Tiefensuche

Um die Tabelle zu generieren, geht man vom Strukturtyp (0,0,0) aus, führt darauf alle 18 möglichen Züge aus und berechnet den neuen Strukturtyp mit Hilfe der Tabellen x_1,y_1 und z_1. In der Zeile des neuen Strukturtyps trägt man in der Tabelle eine 1 ein und zwar in die Spalte, die für den angewandten Seitenzug, mit dem man diese Stellung erreicht hat, reserviert ist.

Im zweiten Schritt wendet man auf alle im ersten Durchgang erreichten Strukturtypen wiederum alle Seitenzüge an und trägt in die Zeile von der neuen erreichten Stellung und Zeile vom angewandten Zug eine 2 ein. Dieser Vorgang wird solange wiederholt, bis in jedem Feld eine Zahl vermerkt ist. Am Schluss hat man eine Tabelle, die einem sagt, wie viele Züge man braucht, um von diesem Typ die Stellung (0,0,0) zu finden.

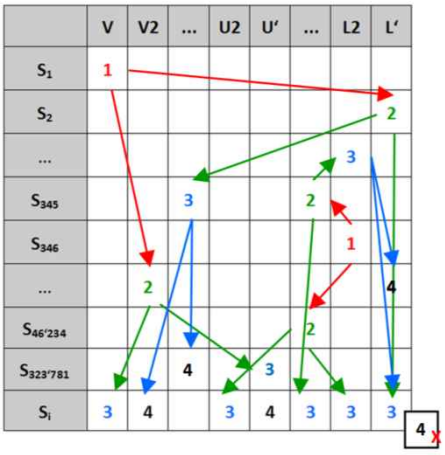


Abbildung 80: Schema für Generierung

In dieser obigen *Abbildung 80* ist ein schematischer Überblick gegeben, wie diese Tabelle zu Stande kommt. Die Daten sind fiktiv und dienen nur dem Verständnis des Prinzips. Bei jedem Durchgang werden hier nur zwei (anstatt 18) neue Strukturtypen erreicht. Das heisst z.B., dass bei der Anwendung von einem V-Zug die fiktive Stellung S_1 erreicht wird, bei einem L2-Zug die Stellung S_{346}. Es werden also je eine 1 in die Spalte 0 und in die Spalte 17 der jeweiligen Stellungszeile geschrieben. Auf die Stellungen S_1 wird danach ein L‘- und V2-Zug vollführt und es werden wiederum zwei neue Stellungen erreicht. Nach jedem Erreichen einer neuen Stellung wird der Zähler um eins erhöht und die erreichbaren Stellungen gekennzeichnet. Eine Stellung kann über jeden Seitenzug erreicht werden, so z.B. in unserem fiktiven Beispiel wird die Stellung S_{345}.einmal in drei und ein anderes Mal in zwei Zügen erreicht. Diese Daten werden auch gespeichert, obwohl eigentlich nur die kleinsten Zahlen in einer Zeile von Bedeutung wären. (Bei der konkreten Implementierung werden aber alle diese Zahlen benötigt, die Gründe werden später erläutert). Wird dieselbe Stellung allerdings durch mehrere Wege erreicht, kommen aber über denselben Zug auf diesen Strukturtyp, so wird nur die kleinere Zahl gespeichert, so im Beispiel Zeile S_i, Spalte L‘.

6.1 Inverse Zugsuche

Bei einer gegebenen fiktiven Stellung von $S_{323‘781}$ kann also z.B. die gesamte Zeile durchgeschaut werden und die kleinste Zahl gefunden werden. Bei unserem Beispiel hat es nur zwei Zahlen, in der realen Tabelle ist jede Spalte gefüllt. Bei uns ist die 3 das Minimum. Diese 3 ist in der Spalte U‘. Durch den inversen Zug von U‘, also U, wird eine fiktive unbekannte Stellung S erreicht. Durch Aussuchen des Minimums in dieser Zeile, 2 < 4, wird also der inverse Zug von V2, also V2, ausgeführt und die Stellung S_1 wird erreicht. Bei uns ist die 1 die kleinste Zahl, also wird noch der inverse Zug von V, V‘ ausgeführt und der Würfel wird in eine 0-Position überführt. Das heisst also, die drei Koordinaten x_1, y_1 und z_1. sind alle 0 oder mit Worten, alle Ecken und Kanten sind richtig orientiert und die vier mittleren Kanten sind in der mittleren Scheibe. Aber bisher klappt diese Suche nur im Fiktiven und bei einer bescheidenen Tiefe von 4.

6.2 Modifikation mit zwei Kombi-Tabellen

Die Hochrechnung dieser drei Koordinaten x_1, y_1 und z_1 auf die gut 2 Milliarden Strukturtypen und der anschliessenden Wegkennzeichnung klingt einleuchtend und stellt theoretisch ein funktionierendes Prinzip dar. Doch die Ausführung bzw. Implementierung führt nicht zu ertragreichen Versuchen, denn der Speicherplatz und die benötigte Zeit sind bei dieser Datenmenge immer noch erheblich. So wird man wiederum dazu gedrängt, die Tiefensuche in zwei Teilschritte zu unterteilen. Um dies zu bewerkstelligen kann man die analogen

Überlegungen anstellen mit je zwei kombinierten Koordinaten. Man sucht sozusagen zwei halbe 0-Positionen. Man versucht dieses Prinzip mit der Rückwärtswegkennzeichnung getrennt auf Stellungen der Form $(x_1,?,z_1)$ und $(?,y_1,z_1)$ anstatt (x_1,y_1,z_1) anzuwenden. Das heisst also, dass wir eine modifizierte Teilzielstellung haben.

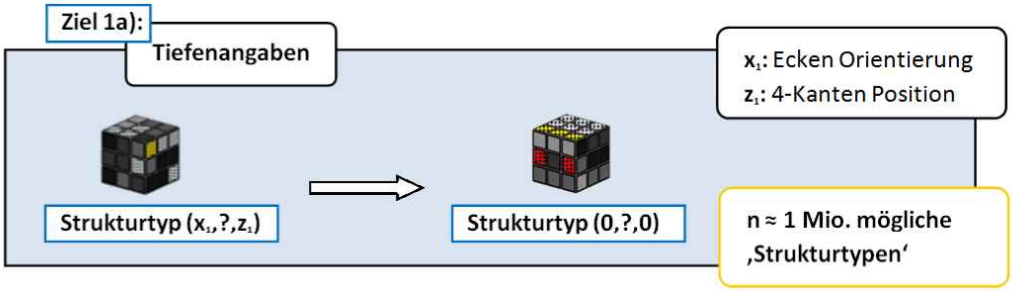


Abbildung 81: modifizierte Teilaufgabenstellung a)

Wenn wir die Koordinaten für die Orientierung der Ecken und die vier mittleren Kanten Positionen miteinander kombinieren, schauen wir ebenfalls einen Strukturtypen an, aber bezüglich anderer Kriterien. Durch das Weglassen der Kanten Orientierungen werden die äquivalenten Stellungen bezüglich der Struktur von x_1 und z_1 viel grösser, folglich gibt es weniger Gruppierungen von diesen modifizierten Strukturtypen. Dies ist genau das, was wir wollen. So haben wird nur noch $(3^7 \cdot 495 =)$ 1'082'565 Strukturstellungen mit je 39'953 Milliarden äquivalenten Stellungen der Form $(x_1,?,z_1)$. Die in diesem Abschnitt gesuchte Stellung (0,?,0) garantiert eine korrekte Orientierung der Ecken und korrekte Platzierung der vier mittleren Kanten, nicht aber eine korrekte Orientierung aller Kanten.

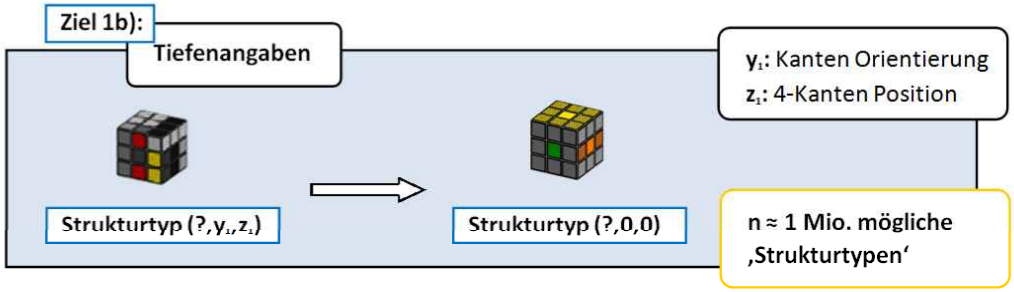


Abbildung 82: modifizierte Teilaufgabenstellung b)

Ebenfalls durch das Weglassen der x_1 Koordinate wird die Anzahl von diesen Strukturtypen vermindert. Wir suchen nun also nur noch je zweimal die Überführung von 1 Million möglichen Stellungen in eine 0-Stellung anstatt von 2 Milliarden.

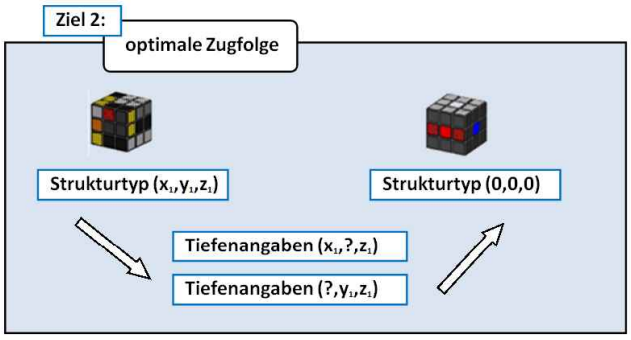


Abbildung 83: Aufgabenstellung mit Umweg

Wenn wir diese beiden einzelnen Tabellen mit den Informationen für die Anzahl Zügen, mit der eine beliebige Stellung der Form $(x_1,?,z_1)$ bzw. $(?,y_1,z_1)$ erreicht werden kann, generiert haben, so können wir bei einer konkreten Stellung den Strukturtyp bezüglich x_1 und z_1 bestimmen und denjenigen bezüglich y_1 und z_1. In einem nächsten Schritt können wir bei beiden Tabellen bei den jeweiligen Zeilen (jeweiliger Strukturtyp) die kleinsten Zahlen herauslesen. Diese Zahlen geben aber nicht mehr eine Information über die Entfernung zur Stellung (0,0,0), sondern nur noch die Distanz zu (0,?,0) bzw. (?,0,0). Die grössere Zahl der beiden kleinsten Zahlen gibt darüber Auskunft, wie gross die Distanz *mindestens* zu der Stellung (0,0,0) ist. Dieser beschriebene Umstand wird uns im Kapitel erwarten.

6.3 Implementierung der Tiefensuchtabellen

Das Prinzip für die Implementierung basiert auf den Illustrationen und Erklärungen von der *Abbildung 78*. Die Implementierung wird am Beispiel für die Generierung der Tiefensuchtabelle für die beiden kombinierten Koordinaten x_1 und z_1 erklärt.

```
void draw(){
  initialisiere_xzKoord();
  loadTable_x();
  loadTable_z();
  moveTablegesamt1(0,1,0);
  saveTable();
}
```

Code 26: Aufruf der versch. Methoden

```
void initialisiere_xzKoord(){
  for(int i=0;i<=xmax-1;i++){
    for(int j=0;j<=zmax-1;j++){
      xzkoord[xmax*j+i][0]=i;
      xzkoord[xmax*j+i][1]=j;
    }
  }
}
```

Code 27: x_1 und z_1 Strukturtyp$(x_1,?,z_1)$ zuordnen

Es werden verschiedene Methoden nacheinander aufgerufen. Als erstes wird ein Array *xzkoord* $[x_{max} \cdot z_{max} + x_{max}][2]$ gefüllt. Die Zeilennummer repräsentiert den Strukturtyp der Form $(x_1,?,z_1)$, in die erste Spalte werden die dazugehörigen x_1-Koordinaten gespeichert und in die zweite die z_1-Koordinaten. Dies ist in einem späteren Schritt sehr praktisch, um von einem gegebenen Strukturtyp der Form $(x_1,?,z_1)$ auf x_1 und z_1 Rückschlüsse ziehen zu können.

```
void loadTable_x(){
  String[] datentemp=loadStrings(pfad+"MoveTableEckOri.txt");
  String[] datensplit=split(datentemp[0],"},{");
  for(int i=0;i<xmax;i++){
    for(int j=0;j<18;j++){
      int temps[]=int(split(datensplit[i],","));
      newMTEckKoord[i][j]=temps[j];
    }
  }
}
```

Code 28: Daten über Auswirkungen für x_1 und z_1 werden geladen

Ausserdem werden die erstellten Informationen für die Auswirkungen auf die Koordinaten x_1 und z_1 benötigt, diese werden von der externen Datei geladen und in das neue Array *newM(ove)T(able)EckKoord[i][j]* gespeichert, welches zum bisherigen Array *EckKoordOri[i][j]* analog ist, wo die Veränderungen der Orientierung der Ecken abgespeichert sind. Ausserdem werden die Daten für die Veränderungen der Positionen der vier Mitte-Kanten in das Array *newM(ove)T(able)mitteKant[i][j]* in einem anderen, hier undokumentierten Schritt, geladen.

```
int tablegesamt1[][]=new int[xmax*zmax+xmax][18];
```

Code 29: Array für Tabelle

Im *Code 29* wird ein Array erstellt, wo die Entfernungen zur Position (0,?,0) nun generiert werden. Dieses Array ist also unsere Tabelle.

```
void moveTablegesamt1(int xz,int tief, int m){
  for(int i=0;i<=17;i++){
    moveTablegesamt2(xz,tief,i);
  }
}
```

Code 30: erste Funktion ruft 18 Mal zweite Funktion auf

Nachdem die beiden Tabellen mit den Informationen für die Auswirkungen auf die Koordinaten x_1 und z_1 geladen sind, wird im *void draw()*, in der Hauptmethode die Funktion *moveTablegesamt1(xz,tief,m)* mit den Parametern (0,1,0) aufgerufen (siehe *Code 26*).

Die Generierung wird in zwei Rekursionsschritten vollzogen, das heisst die erste Funktion ruft die zweite auf, diese bezieht sich wiederum auf die erste Funktion.

Dabei ist diese erste Funktion abhängig von dem Strukturtypen *xz* sowie von der Tiefenzählvariable *tief* wie auch vom Seitenzug *m*. Innerhalb dieser Funktion wird dann also die zweite Funktion *moveTablegesamt2(xz,tief,m)* 18 Mal mit den verschiedenen anzuwendenden Seitenscheibenzüge aufgerufen. Diese Aufrufe werden mit einer for-Schleife bewerkstelligt: Die Variablen *xz* und *tief* werden unberührt übergeben, der Zähler der for-Schleife *m* übergibt die Art des Seitenzuges.

```
void moveTablegesamt2(int xz,int tief,int m){
  if(tief<=anz){
    int xtemp=newMTEckKoord[xzkoord[xz][0]][m];
    int ztemp=newMTmitteKant[xzkoord[xz][1]][m];

    int xztot=xmax*ztemp+xtemp;

    if(tablegesamt1[xztot][m]>tief ||tablegesamt1[xztot][m]==0){
      tablegesamt1[xztot][m]=tief;


      moveTablegesamt1(xztot,tief+1,0);
    }
  }
}
```

Code 31: zweite Funktion errechnet neue Koordinaten

Die zweite Funktion wird also in einem ersten Schritt 18^1 und in einem n-ten Schritt 18^n Mal aufgerufen. Als Erstes wird überprüft, ob die Abbruchbedingung noch nicht erreicht ist. Danach wird aus der Variabel *xz* mit Hilfe des zuvor erstellten Array *xzkoord[][]* die dazugehörigen Komponenten x_1 und z_1 errechnet. Diese werden in dem Array newMTEckKoord[x_1][] bzw. newMTmitteKant[z_1][] in die erste Spalte eingeben, um die Zeilennummer zu bestimmen. In die zweite Spalte wird der übergebene Seitenzug übergeben, welcher als Spaltennummer fungiert. Durch diese Abfrage in den beiden Tabellen können die veränderten neuen einzelnen Koordinaten von x_1 und z_1 in die temporären Variablen *xtemp* bzw. *ztemp* gespeichert werden. Der neue Strukturtyp wird in xztot gespeichert. Zwei Bedingungen müssen noch erfüllt sein, um die Tiefenzahl in das aktuelle Feld in tablegesamt1[*Strukturtyp*][*Seitenzug*] eintragen zu können. Falls in dem Feld noch keinen Eintrag in dem Feld ist oder der Eintrag kleiner als der aktuelle Tiefeneintrag ist. Als Abschluss wird die erste Funktion mit den übergebenen Parametern des neuen Strukturtyps *xztot*, der um eins inkrementierte Tiefenzähler *tief+1* und der Seitenzug 0 aufgerufen. Und die Anwendung aller 18 Seitenzüge können an der neuen Position gestartet werden.

```
PrintWriter tabelle_xz=createWriter(pfad);
void saveTable(){
  for(int i=0;i<=xmax*zmax+xmax-1;i++){
    for(int j=0;j<=17;j++){
      if(j<17){
        tabelle_xz.print(tablegesamt1[i][j]+",");
      }
      if(j==17){
        tabelle_xz.print(tablegesamt1[i][j]+"),(");
      }
      if(i==(xmax*zmax+xmax-1) && j==17){
        tabelle_xz.close();
        println("Tabelle fertig generiert");
      }
    }
  }
}
```

Code 32: Tiefenzahlen werden extern abgespeichert

Ist die Abbruchbedingung erreicht, die vorher bestimmte Anzahl an Rekursionstiefen also erlangt, wird das Array extern wiederum in einer *.txt* Datei abgespeichert, sodass bei der Suche die Zeit nicht für die Generierung der Tiefensuchtabellen verloren geht.

Dieses ganze Verfahren wird analog angewendet um die Tiefensuchtabelle für die beiden kombinierten Koordinaten y_1 und z_1 zu generieren. Beide Processing Dateien sind auf der CD unter den Dateien_5 abgelegt und können generiert werden. Wiederum ist darauf zu achten, einen gültigen Pfad für den Speicherort einzugeben. Diese Generierung kann einige Zeit dauern, bei Fertigstellung der Tabelle wird „Tabelle fertig generiert“ ausgegeben.

7 Tiefensuche in der erste Phase

Sind auch diese beiden Tiefensuchtabellen generiert, kann man sich nun einen produktiveren Teil vornehmen, nämlich die Suche einer optimalen Zugfolge für das Erreichen des Strukturtyps (0,0,0) oder einer Stellung aus der Menge M_1.

Durch die erzwungene Tatsache mit zwei kombinierten Tabellen arbeiten zu müssen (Kap. 6.2) ist diese Suche um einiges schwieriger und ungerichteter geworden.

Durch diesen Umstand ist der Weg nicht mehr genau gekennzeichnet, sondern es sind nur noch Fragmente auf dem Weg hinterlassen worden, trügerische Wegweiser sozusagen. Die theoretische Tabelle mit allen drei Informationen kann man mit einem Navigationssystem beschreiben, das uns zwar immer in die inverse Richtung navigiert, doch bei Kenntnis dieses Umstandes, führt es uns mit genauen Kilometerangaben zu den drei Hügeln am Meer (0,0,0). Doch der Rucksack war zu klein, um das Gerät mitnehmen zu können, darum müssen wir uns auf die ungenauen Wegweiser an jeder Kreuzung und die alte Karte mit den alten Strassen verlassen. An jeder Kreuzung weiss ich nur, dass auf dem entgegengesetzten „(Spalten)wegweiser“ und der falsch herum gehaltenen „(Spalten)karte“ in der andern „Hand“ (Tabelle) beide Distanzangaben nur die minimalen Entfernungen bis zur Ankunft bei den drei Hügeln darstellt. Der Weg führt zwar oft an der Küste entlang, doch es tauchen nur immer zwei Arten von Hügeln auf einmal auf. Es wird also ein buntes Herumirren werden, bis wir (hoffentlich) bei Tageslicht und mit nicht zu müden Beinen (Anzahl Zügen) an einer Position ankommen, wo wir alle drei Arten von Hügeln auf einmal antreffen.

Um nicht noch mehr Druckerschwärze für weitere Geschichten von Wanderern und morphologischen Paradoxon zu verschwenden, wenden wir uns der Tiefensuche zu.

```
                                .
                                .
                                .
946321: tablegesamt1_xz={7,7,7,6,7,7,8,8,8,7,7,7,6,7,7,7,7,7},
946322: tablegesamt1_xz={7,7,7,7,7,8,8,7,7,7,7,7,7,8,7,8,7,7,6},
946323: tablegesamt1_xz={7,7,8,7,7,7,7,7,7,7,7,7,7,7,6,7,7,7},
946324: tablegesamt1_xz={7,7,6,7,7,7,8,7,8,7,7,7,8,7,8,6,7,7},
                                .
                                .
                                .
```

Code 33: Tabelleneintrag in Tabelle_xz an Stellung 946‘322

```
                                .
                                .
                                .
34291: tablegesamt1_yz={7,7,7,6,7,7,8,8,8,7,7,7,6,7,7,7,7,7},
34292: tablegesamt1_yz={7,7,7,7,7,8,8,7,7,7,7,7,7,8,7,8,7,7,6},
34293: tablegesamt1_yz={7,7,8,7,7,7,7,7,7,7,7,7,7,7,6,7,7,7},
34294: tablegesamt1_yz={7,7,6,7,7,7,8,7,8,7,7,7,8,7,8,6,7,7},
                                .
                                .
                                .
```

Code 34: Tabelleneintrag in Tabelle_yz an Stellung 34‘292

Jede Stellung kann nun in die Form (x_1,?,z_1) bzw. (?,y_1,z_1) gebracht werden. Die Stellung_xz ist beispielweise wie im *Code 33* 946'322, während Stellung_yz nach *Code 34* 34'292 wäre. An diesen Positionen in den jeweiligen Tabellen haben wir bekanntlich unsere Informationen, die aussagen, wie weit die 0-Position mindestens entfernt ist, wenn wir diese Richtung einschlagen würden.

Wir starten also bei einer beliebigen Stellung, beschrieben mit (x_1, y_1,z_1) oder mit (x_1, ?,z_1) bzw. (?, y_1,z_1). Durch das Errechnen der Mindestdistanz zu (0,0,0) durch das grössere Minima der beiden Tabelleneinträgen, können wir alle Spaltenwegweiser verfolgen, die mit dieser Zahl versehen sind. Unsere Gruppe von Wanderern teilt sich also auf und jeder macht sich in eine andere Richtung davon. An diesen neuen Stellungen berechnen wir wiederum die Mindestdistanz, und verfolgen wieder die erfolgversprechendsten Wege.

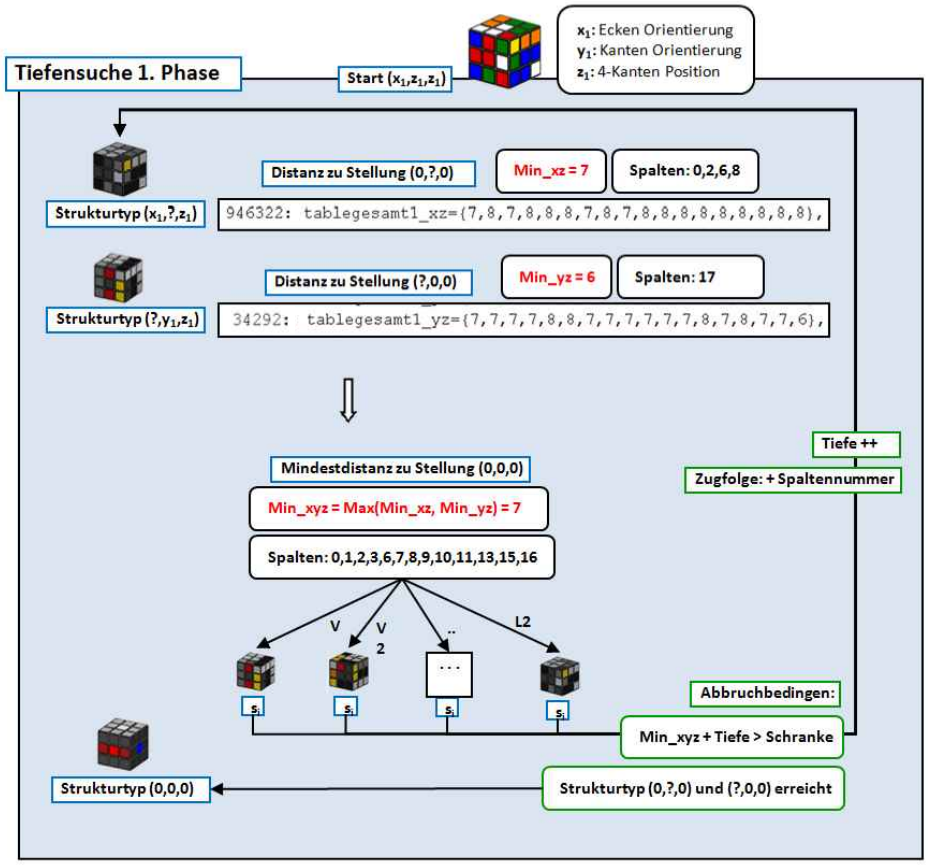


Abbildung 84: Ablaufschema für die Tiefensuche in der 1. Phase

Dieses Herumirren können wir aber nicht planlos machen. Beim Start machen wir eine Bedingung ab: Wir suchen vorerst nur Wege, die in einem Schritt erreichbar sind. Jeder von uns geht nur ein inverses Wegstück bis zum nächsten Wegweiser. Wir überprüfen, ob wir bei den drei Hügeln angekommen sind. Falls nicht, brechen wir ab und gehen einen Wegweiser zurück, also zum Ausgangspunkt. Wir erhöhen die Anzahl der erlaubten Schritte auf zwei. Wenn wir jetzt einen Teilschritt ausgeführt haben, so können wir mit den Mindestangaben auf

den Wegweisern überprüfen, ob wir mit dem eingeschlagenen Weg die Bedingung überhaupt einhalten können. Wenn also nach einem Teilschritt die Mindestanzahl grösser als 1 ist, so müssen wir den eingeschlagenen Weg nicht mehr weiter verfolgen, denn wir wissen bereits jetzt, dass wir die Bedingung nicht einhalten werden können. Haben alle abgebrochen, so wird die Zahl der maximal erlaubten Züge wieder um eins erhöht. Mit dieser Abmachung einer Schranke können wir schon bei frühen Knotenpunkten sehen, welche Wege nicht zu einem kurzen Weg führen werden. Wir wissen zwar nicht welchen Weg wir einschlagen sollen, doch wir wissen, welche Richtungen wir nicht nehmen sollen und können unsere Ressourcen für erfolgversprechendere Wege aufsparen. Jetzt ist es kein blindes Herumirren mehr sondern eine eingeschränkte Suche.

Wenn wir die Schranken immer um eins erhöhen, werden wir einmal einen Weg finden, der uns zu den drei Hügeln oder zu einer Stellung (0,0,0) führt. Wir müssen aber bedenken, für jede Ausweitung der Schranke vervielfachen sich die Verzweigungen und das Tempo wird dadurch verringert. Wenn wir einen Weg gefunden haben, so brechen alle ab und der Weg, der auf dem persönlichen Notizblock niedergeschrieben worden war, wird ausgeführt.

Wenn wir nun anspruchsvoll werden, wollen wir nicht nur einen möglichen Weg finden, sonder mehrere. Also machen wir noch eine Anzahl an Zugfolgen ab, die gefunden werden sollen. Wenn jemand eine findet, wird jedem eine Nachricht gesendet, dass jetzt bereits einer gefunden worden ist. Die Suche läuft also so lange, bis die Anzahl an definierten Zugfolgen gefunden worden ist.

Für uns ist diese erweiterte Abmachung insofern von Bedeutung, dass die Stellungen (0,0,0) alle die selbe Struktur bezüglich den Koordinaten x_1, y_1 und z_1 aufweisen, doch sind sie nicht identisch, die Positionen der Ecken und Kanten stimmen z.B. noch nicht überein. Die Suche in der zweiten Phase beginnt damit bei unterschiedlichen Stellungen, die mehr oder weniger optimal sind. Daher kann es von Vorteil sein, wenn wir mehrere mögliche Zugfolgen in der ersten Phase herausfinden können, auch wenn einige davon mehr Züge beinhalten. Die Suche von den erreichten Stellungen (0,0,0) zu den finalen Stellungen beschäftigen uns aber vorerst noch nicht, sie werden im nächsten Kapitel besprochen.

7.1 Implementierung der Tiefensuche

Diese Suche wird wieder rekursiv implementiert. Die Funktionen rufen sich in einem Zirkel immer wieder selbst auf. Bei einem einmaligen Aufruf wird sozusagen ein Suchtrupp losgeschickt um die Zugfolgen zu finden.

```
sucheweg(xmax*kaliz+kalix,ymax*kaliz+kaliy,0,startzug1);
```

Code 35: Start der Suche

Der obige Aufruf muss einmal getätigt werden um die Suche für die erste Phase zu starten.

```
void sucheweg(int xz,int yz,int rek, int[] zugtemp){
  for(int i=2;i<14;i++){
    anz1=i;
    if(anzzuege1>=grenz1){
    }
    else{
      searchwayfrom(xmax*kaliz+kalix,ymax*kaliz+kaliy,0,startzug1);
    }
  }
}
```

Code 36: Suche wird mit verschiedenen Schranken (*anz1*) gestartet

In der aufgerufenen Funktion wird die Schranke festgelegt, um zuerst nach kurzen Zügen Ausschau zu halten. Die *for-Schleife* mit dem Zähler *i* wird auf die globale Variabel *anz1* übertragen, sodass in allen Funktionen auf diese zugegriffen werden kann. Falls die Anzahl an gefundenen Zugfolgen noch nicht erreicht ist (beim ersten Durchgang irrelevant), wird die Parameter der nächsten Funktion übergeben. Auf den Weg werden dem Wanderer die aktuellen hochgerechneten Koordinaten mitgegeben sowie die Information, dass er noch keinen Schritt hinter sich hat. Ebenfalls bekommt er einen 12-seitigen Notizblock mit, wo er seine Züge notieren kann. Diese 12 Seiten repräsentieren die maximale Schranke, die wir festlegen. Die Schwierigkeit liegt in der Wahl dieser Schranke: Ist sie zu niedrig, so wird keine Lösung gefunden, wird sie zu gross gewählt, so werden unnötig viele Stellungen betrachtet. Durch das dynamische wachsen der Tiefenschranke *anz1* wird diesem Dilemma entgegengewirkt, eine maximale Schranke wird dennoch benötigt, die wir für die erste Phase auf 12 setzten.

```
void searchwayfrom(int xz,int yz,int rek, int[] zugtemp){
  int minixz=0, miniyz=0, minitot=0;

  int eightteen[]=int(split(datensplitxz1[xz],","));
  int eightteen2[]=int(split(datensplityz1[yz],","));

  for(int i=0;i<=17;i++){
    if(eightteen[i]==0){
      eightteen[i]=20;
    }
    if(eightteen2[i]==0){
      eightteen2[i]=20;
    }
  }

  minixz=min(eightteen);
  miniyz=min(eightteen2);

  if(minixz>miniyz){
    minitot=minixz;
  }
  else{
    minitot=miniyz;
  }
  welcherzug(minitot,xz,yz,rek,zugtemp);
}
```

Code 37: errechnen der Mindestdistanz *(minitot)* an Knotenpunkt

In dieser Funktion wird die Mindestdistanz *minitot* gesucht. Dafür werden in die Arrays *eightteen[]* und *eightteen2[]* die 18 Distanzangaben von den geladenen Tabellen für die Distanzangaben herausgesucht. Falls bei einem Feld keine Daten vorhanden sind, also eine 0 vermerkt ist, wird dieses mit der willkürlichen Zahl 20 gefüllt, sodass diese nicht als Minimum interpretiert wird. Es werden die beiden Minima gesucht und in die Variablen *minixz* und *miniyz* gespeichert. Die grössere der beiden wird dann unter der Variablen *minitot* gespeichert. Ist diese Mindestdistanz zu der O-Position gefunden, wird wiederum die nächste Funktion aufgerufen, die nötigen Parameter werden übergeben.

```
void welcherzug(int minitot, int xz, int yz, int rek,int [] zugtemp){
  int minitot1=0;      //welcher Zug

  if(rek+minitot<anz1){
    int eightteen[]=int(split(datensplitxz1[xz],","));
    int eightteen2[]=int(split(datensplityz1[yz],","));

    for(int i=0;i<=17;i++){
      if(eightteen[i]==minitot){
        minitot1=i;
        neuxyz(minitot1,xz,yz,rek,zugtemp);
      }
    }

    for(int i=0;i<=17;i++){
      if(eightteen2[i]==minitot){
        minitot1=i;
        neuxyz(minitot1,xz,yz,rek,zugtemp);
      }
    }
  }
}
```

Code 38: errechnen des nächsten Zuges, Richtung

Ist die Mindestdistanz gefunden, muss noch die Art des zugehörigen Richtungswegweisers, also die Art des Zuges, identifiziert werden. Dabei wird noch zuerst überprüft, ob die Schranke noch nicht überschritten ist. Die zweimal 18 Felder werden nach dem Wert von *minitot* abgeglichen, bei einem Treffer wird eine neue Funktion mit dieser Art des Zuges (*minitot1*) aufgerufen.

```
void neuxyz(int minitot1,int xz,int yz, int rek, int [] zugtemp){
  int minitot2=0;      //umgekerter Zug

  if(minitot1%3==0){
    minitot2=minitot1+2;}
  if((minitot1+1)%3==0){
    minitot2=minitot1-2;}
  if((minitot1+2)%3==0){
    minitot2=minitot1;}

  int xtemp=newMTEckKoord[xzkoord[xz][0]][minitot2];
  int ztemp=newMTmitteKant[xzkoord[xz][1]][minitot2];
  int ytemp=newMTKantKoord[yzkoord[yz][0]][minitot2];
  int ztemp2=newMTmitteKant[yzkoord[yz][1]][minitot2];
  int neuxz=xmax*ztemp+xtemp;
  int neuyz=ymax*ztemp2+ytemp;

  zugtemp[rek]=minitot2;

  if(anzzuege1<grenz1){
    if(neuxz==0 && neuyz==0){
      for(int i=0;i<12;i++){
        zuege1[anzzuege1][i]=zugtemp[i];
      }
      anzzuege1++;
    }
  }
  else{
    searchwayfrom(neuxz,neuyz,rek+1,zugtemp);
  }
}
```

Code 39: Berechnung und Aufruf der neuen Stellung

Dieser Zug muss noch umgewandelt werden, sodass der inverse Zug herauskommt (*minitot2*). In einem nächsten Schritt werden die neuen Koordinaten berechnet. Dabei wird wieder auf die Auswirkungstabellen für die Orientierungen der Ecken bzw. Kanten und den Positionen für die vier mittleren Ecken zugegriffen. Die einzelnen neuen Koordinaten müssen noch hochgerechnet werden, um die beiden neuen Strukturstellungstypen (x_1, ?,z_1) bzw. (?, y_1,z_1) zu erreichen. Zudem wird das persönliche Notizbuch *zugtemp[]* an der Stelle der Rekursionstiefe (Anzahl bereits ausgeführter Züge) mit dem ausgeführten Zug gespeichert. Falls eine 0-Position erreicht wurde (*neuxz* und *neuyz* = 0) wird die Suche abgebrochen und die Anzahl gefundener Zugfolgen global erhöht. Ebenfalls wird bei einer erfolgreichen Suche das persönliche Notizbuch in das globale Array *zuege1[][]* gespeichert. Bei einer (noch) nicht

erfolgreichen Suche werden an der neu errechneten Stellung wiederum der Wegweiser bzw. die Mindestdistanz analysiert, das heisst die Funktion *searchwayfrom()* wird mit einer inkrementierten Rekursionstiefe aufgerufen.

I.	**„Schaltungszentrale“, Anzahl gefundenen Zugfolgen überprüft, Schranke festlegen, „Losschicken“ der „Personen“**
II.	**Wegweiser Analyse, Mindestdistanz finden**
III.	**Wegweiser Analyse, welcher Zug bzw. Richtung**
IV.	**Berechnung der neuen Stellung und Eintrag ins persönliche Notizbuch, Überprüfung der Stellung, falls 0-Position abbrechen und speichern, ansonsten Überführung zu Wegweiseranalyse II**

Diese Ausführungen sind auf die simpelste Form reduziert worden. Die tatsächliche Implementierung enthält noch einigen Zwischenschritte, die für die Reibungslose Anwendung bzw. den Übergang in die zweite Phase dienen. Doch in den Grundzügen wurde in diesem Kapitel die Implementierung illustriert.

Wenn diese Suche implementiert ist, so ist die erste Phase abgeschlossen. Aus einer beliebigen Stellung (x_1, y_1,z_1) wird eine Zugfolge gesucht, die eine maximale Länge von 12 Zügen hat. Es werden nicht die kürzest möglichen Folgen herausgesucht, nur möglichst optimale. Durch die Wahl des Parameters *anzuege1* wird die Anzahl gewünschter Zugfolgen festgelegt, sodass man bei der zweiten Phase aus einer Auswahl von Stellungen eine optimalere suchen kann.

8 Überblick über die zweite Phase

Wir können jetzt eine beliebige Stellung (x_1,y_1,z_1) in die Stellung (0,0,0) bezüglich der Form (x_1, y_1,z_1) überführen. Das Erreichen dieses Zustandes ist aber erst die Hälfte der Arbeit. Im Prinzip müssen die *Kapitel 5, 6 und 7* für die zweite Phase noch einmal komplett durchgearbeitet werden. Durch die Analogie ist es allerdings nicht nötig, die Prinzipien im Detail noch einmal zu erläutern. An dieser Stelle wird lediglich ein Überblick verliehen. Vergleichen Sie dazu eventuell noch einmal *Abbildung 67*.

Abbildung 85: Aufgabenstellung 1. Phase

Das Ziel aus der ersten Phase ist also erreicht, nun versuchen wir die Zielsetzung der zweiten Phase zu fassen. Durch den speziellen Strukturtyp aus der Menge M_1 können wir nun eine riesige zentrale Reduzierung machen. Wir verbieten von nun an alle einfachen Züge auf die Seiten Vorne, Rechts, Hinten und Links. Es sind also nur noch die drei verschiedenen Züge für die obere und untere Seite erlaubt sowie alle Doppelzüge. Das heisst also, von nun an sind nur noch 10 anstatt 18 Seitenzüge erlaubt.

{V2,O,O2,O‘,H2,U,U2,U‘,R2,L2}

Durch das Verbot der anderen Züge überführt jeder Seitenzug jede Stellung wiederum in eine Stellung aus der Menge M_1. Die Orientierungen der Kanten und Ecken können nicht mehr verändert werden. Das sind sozusagen die Pflichten, um sich in dieser Menge aufhalten zu dürfen. Dies hat die angenehme Nebenerscheinung, dass wir die Koordinaten x_1, y_1 und z_1 vergessen können, denn wir wissen, egal welchen Zug wir anwenden, diese Koordinaten sind 0.

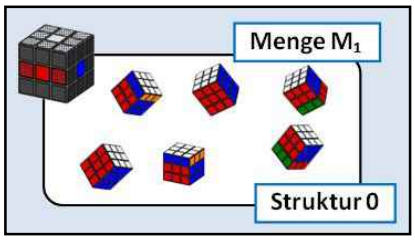


Abbildung 86: Analyse von Menge M_1

Wir haben diesen Strukturtyp erreicht, bei dem alle gelben und weissen Flächen sich auf den Seiten oben und unten verteilen. Alle diese Stellungen, die dieser Beschreibung entsprechen, haben wir in die Menge M_1 verfrachtet. Wir sprachen von äquivalenten Stellungen, doch das sind diese natürlich keineswegs. Sie sind nur gleich bezüglich der erwähnten drei Kriterien. Es ist nun gerade das Ziel, diese Anzahl an strukturähnlichen Stellungen wieder zu unterscheiden. Es ist gerade die Mächtigkeit dieser Gruppe, die uns jetzt mit den möglichen Stellungen konfrontiert. Das heisst also, diese gut 20 Milliarden Elemente, die in diesen zusammengefassten Gruppen erschienen sind, müssen auf die Auswirkungen der Seitenzüge bezüglich anderen Kriterien analysiert werden.

Abbildung 87: Aufgabenstellung 2. Phase

Es können wiederum neue Strukturtypen bezüglich Kriterien erstellt werden.

Wenn wir uns einen Würfel von der Menge M_1 vorstellen, so sehen wir, dass alle Ecken und Kanten noch an ihre ursprünglichen Positionen müssen. Durch die Auflistungen der möglichen Positionen können wir wieder drei Koordinaten jedem Strukturtyp zuordnen. Durch die Koordinaten für die Ecken- sowie Kantenpositionen und den zweiten 4-Mitte Kantenpositionen können die Strukturtypen der Form (x_2,y_2,z_2) eindeutig beschrieben werden.

Abbildung 88: Würfel aus M_1

All diese Mengen von Strukturtypen in der ersten Phase haben alle die gleichen Strukturtypen bezüglich der Koordinaten in der zweiten Phase. Wenn wir diese neuen Strukturtypen auf die Auswirkungen der Seitenzüge analysieren und die Tiefensuche darauf anwenden, kommen wir bezüglich den neuen Koordinaten zu einer 0-Position.

Wenn wir von einem Würfel aus der Menge M_1 ausgehen und das Verbot einiger Seitenzüge erlassen und diesen konkreten Strukturtyp zweiter Ordnung, also bezüglich den neuen Koordinaten x_2, y_2 und z_2 , in eine 0-Position für die Form (x_2, y_2,z_2) überführen können, so ist sichergestellt, dass sowohl die ersten Koordinaten x_1, y_1 und z_1 wie auch die zweiten Koordinaten x_2, y_2 und z_2 0 sind. Damit wäre der Würfel also gelöst.

8.1 Generierung der Auswirkungs-Tabellen x_2, y_2 und z_2

Als ersten Schritt werden die Auswirkungen der Seitenzüge auf die drei Koordinaten x_2, y_2 und z_2 benötigt. Dazu werden drei Tabellen generiert die z.B. Informationen enthalten, in welche Positionskoordinatenstellung der Ecken eine beliebige Stellung nach den jeweiligen Seitenzügen überführt wird.

Die Koordinate z_1 ist die Zahl in der Reihe der lexikografische Auflistungen der Permutationen mit 4 Elementen, die auf 4 Plätzen angeordnet sein können. Dies entspricht unseren 4 mittleren Kanten, die bereits auf ihren vier Plätzen angeordnet sind, allerdings noch nicht jede an ihrem eigenen Platz. Durch die Auflistungen der Möglichkeiten dieser vier Kanten auf vier Plätzen erhalten wir eine Zahl, die dem Platz in diesen Auflistungen entspricht. Die 0-te Auflistung wird wiederum angepeilt.

Die Implementierung der Generierung dieser drei Tabellen läuft sehr analog zu ihrem jeweiligen Pendent in der ersten Phase ab. Es gibt allerdings auch einige Abweichungen, z.B. die Generierung der Stellungen. Doch basieren sie auf denselben Überlegungen.

	mögliche Stellungen		Anzahl Einträge
x_1: Ecken Positionen	8!	40‘320	403‘200
y_1: Kanten Positionen	8!	40‘320	403‘200
z_1: 4-Kanten Position 2	4!	24	240

Die Generierung der drei Tabellen ist unter den Dateien_6 auf der CD vorhanden. Es gibt drei einzelne Dateien, bei denen jeweils der Pfad für den Speicherort eingegeben werden muss.

8.2 Tiefensuchtabelle generieren

Auch hier werden wiederum Distanzangaben benötigt. Das Prinzip dabei ist sehr analog, die Kombination von drei Koordinaten ergibt eine Mächtigkeit von 20 Milliarden verschiedener Strukturtypen, bei der Aufspaltung in 2 Kombi-Tabellen werden dagegen nur noch je eine knappe Million verschiedener möglicher Strukturtypen unterschieden, sodass die Wegkennzeichnung (bzw. Wegfragmentierung) für jeden Strukturtypen möglich wird. Die Implementierung basiert auf den Erklärungen in *Kapitel 6*.

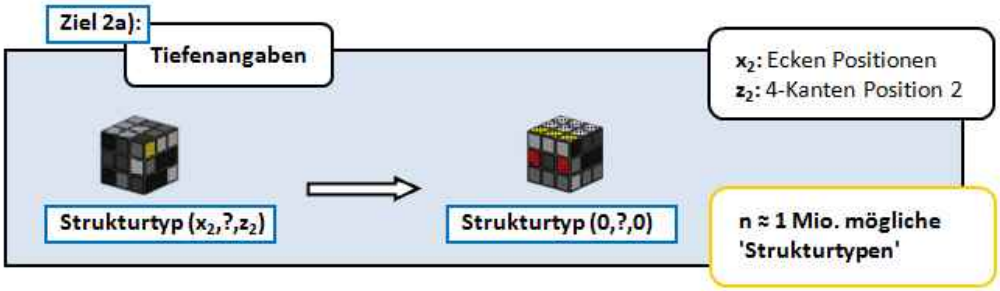


Abbildung 89: modifizierte Teilaufgabenstellung a)

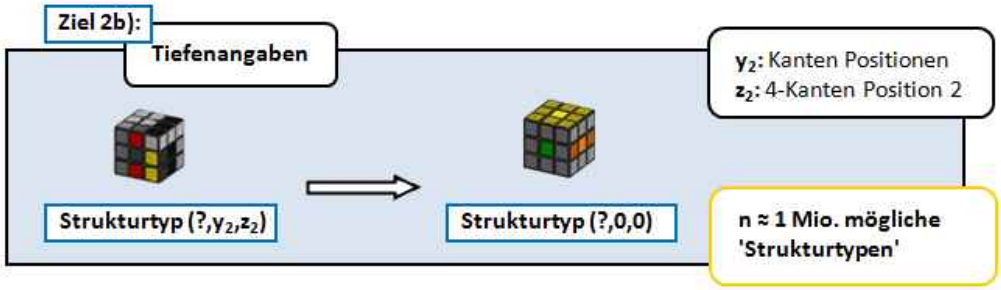


Abbildung 90: modifizierte Teilaufgabenstellung b)

Die Generierung der beiden Tabellen ist unter den Dateien_7 auf der CD vorhanden. Es gibt zwei einzelne Dateien, bei denen wiederum der Pfad für den Speicherort eingegeben werden muss.

8.3 Tiefensuche in der zweiten Phase

Die detaillierten Erklärungen entnehmen Sie bitte aus dem *Kapitel 7*, mit der nahezu analogen Illustrationen. Die Grundidee ist dieselbe. Nach der Suche in der ersten Phase muss nun eine gefundene Zugfolge angewendet werden. Der Würfel befindet sich also in einer M_1-Stellung. Wir erhalten die drei Koordinaten für die zweite Phase und können den Zustand in der Form (x_2, y_2, z_2) beschreiben.

Vom konkreten Zustand werden die beiden Zustände (x_1,?,z_1) bzw. (?,y_1,z_1) konstruiert, und es werden jeweils in den beiden Tiefentabellen die Mindestdistanzen zu den Positionen (0,?,0) bzw. (?,0,0) herausgesucht. Es ist also wiederum eine Suche mit fragmentierten Informationen zu den Richtungen, die wir einschlagen sollen.

Wir haben in dem Sinne neue Karten mit anderen Angaben für weniger Strassen sowie neue Wegweiser (mit weniger möglichen Richtungen) mit anderen Distanzangaben zu anderen Ortschaften erhalten. Das Ziel der Suche ist nun nicht nur drei Hügel auf 0-Niveau zu finden, sondern auf diesen Hügeln müssen drei Bäume mit 0 Blättern stehen. Wenn wir allerdings nur Wege einschlagen, die auch wirklich auf den Karten angezeigt sind bzw. mit einem Wegweiser versehen sind, so können wir davon ausgehen, wenn wir die drei Bäume mit 0

Blätter finden, dass sie sich auf den drei Hügeln befinden. Bei keinem Verstoss gegen das Verbot kommen wir nun sogar nur noch an Orte mit drei Hügeln am Meer entlang.

Die erlaubte Schranke von Schritten in der zweiten Phase legen wir auf unter 18 fest, sodass wir auf eine maximale Länge des Gesamtzuges von 29 bekommen. Ist die Implementierung vollständig, so kann also mit diesem Algorithmus von einer beliebigen der 43 Trillionen möglichen Stellungen eine Zugfolge mit maximal 29 Zügen generiert werden, die die konkrete Stellung in den Ursprungszustand überführt.

9 Cube Transformer

9.1 Beliebige Stellungen

Wenn man diesen Algorithmus noch ausweitet, kann sogar jeder beliebige Würfel in eine andere beliebige Stellung in 29 Züge überführt werden. Diese Ausweitung konnten wir ebenfalls in unser Programm implementieren. Unser Programm ist also in der Lage, jeden beliebigen Würfel in jeden beliebigen Würfel zu überführen und nicht nur in die finale Stellung.

9.2 Suboptimale Zugfolgen in der ersten Phase

Wendet man die Suche in der zweiten Phase auf verschiedene resultierende Stellungen aus der ersten Phase an, so bekommt man unterschiedliche Längen von Zugfolgen in der zweiten Phase. Es ist sogar erstaunlich, dass z.B. oft eine längere Zugfolge in der ersten Phase zu sehr kurzen Zugfolgen in der zweiten Phase führt. Diese erweiterte Suche kann man ebenfalls implementieren, sodass noch eine optimalere Zugfolge ausgegeben werden kann. Die Implementation dafür ist auch in unserem Programm enthalten, allerdings bei den Standardeinstellungen nicht aktiviert, denn es führt rasch zu zeitaufreibenden Suchverfahren.

9.3 Benutzeroberfläche

Um all diese Implementierungen auch visuell wahrzunehmen, konnten wir ein kleines Programm schreiben, wo der Würfel dargestellt wird. Dort können bei zwei Würfeln - Ausgangsstellung sowie die Zielstellung - die Farbflächen gefüllt werden, entweder durch Füllen mit der Farbpipette oder aber auch durch Einlesen mit einer Kamera. Je nachdem welcher Würfel auf der Seite selektiert ist, können auch die Seitenzüge per Tastendruck ausgeführt werden. Der entsprechende kleine Buchstaben *{v,o,h,u,r,l}* bedeutet einen Seitenzug im Uhrzeigersinn, wohingegen die Grossbuchstaben *{V,O,H,U,R,L}* eine Gegenuhrzeigersinndrehung zur Folge haben.

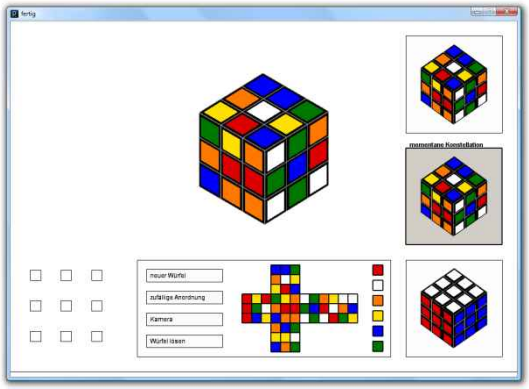


Abbildung 91: Cube Transformer

Um dieses Programm ausführen zu können, beachten Sie bitte, dass Sie den Pfad in der ersten Zeile in der Datei Cube_Transformer aktualisieren müssen, um auf die generierten Daten zugreifen zu können.

10 Mechanik/Robotik

10.1 Erster Versuch

Inspiriert durch Videos auf YouTube, wagten wir uns an einen ersten Versuch, einen einfachen Prototyp zu errichten. Wir integrierten zwei Motoren, einen um die Plattform zu drehen, den anderen als Stoss- und Greifarm. Es stellte sich schnell heraus, dass mit den vorhandenen Lego-Bauteilchen kein stabiler Roboter erbaut werden kann. Der Würfel verkantete sich oft, sodass ein weiteres Lösen verunmöglicht wurde. Wir diskutierten über weitere Möglichkeiten. Wir kamen zu zwei weiteren möglichen besseren Ideen. Eine davon war ein weiteres Modell mit Lego, allerdings sollte ein zusätzlicher Stossarm integriert werden. Bei der anderen sollte die Instabilität dadurch verbessert werden, indem wir mit anderen Materialien die Befestigung errichten sollten. Auf der beiliegenden CD ist ein kurzes Video *(Modell 1)* von diesem Modell vorhanden.

Abbildung 92: erstes Modell

10.2 Modell mit anderen Materialien

Da wir mehr als nur ein Lego NXT Roboter zur Verfügung hatten, wollten wir eine Konstruktion erstellen, die fähig ist, den Würfel mehr als nur von einer Seite anzusteuern damit wir den Würfel schneller lösen können. Um diese Idee zu verwirklichen behalfen wir uns mit anderen Materialien ausser Lego, etwa Holz oder Metall, um der Instabilität der Lego-Bauteilchen zu entgehen.

Um das Ziel, den Würfel von vier Seiten anzusteuern, zu erreichen, kam uns die Idee, den Würfel auf einer drehbaren Achse zu fixieren und ihn mit vier linearen Seitenarmen anzusteuern. Doch die Umsetzung war nicht so einfach wie der Plan. Auch hier gab es einige Probleme: Wie kann man die Seite in der eingespannten Ebenen drehen? Wie können die Arme geführt werden?

Abbildung 93: fixierte Achse, 2. Modell

In der *Abbildung 93* kann man die fixierte Achse sehen, wobei der Würfel mit einer Drehplattform (*Abb. 94*) gedreht hätte werden sollen. Wird die Drehplattform an der Achse

hinuntergelassen und über das Zahnrad durch einen Motor gedreht, so dreht sich der gesamte Würfel. Würde jedoch ein Seitenarm den Würfel festhalten, so würde ein Seitenzug ausgeführt werden. Würde anderseits die obere Plattform hinunter gelassen werden und der Seitenarm würde sich drehen, so würde der Seitenzug auf der Seite ausgeführt werden. Der Arm auf der Achse, der sich durch ein inneres und ein äusseres Zahnrad auf der Achse bewegen kann, wie auch die Seitenarme sollten auf einer Schiene bewegt werden. Wir versuchten dies zuerst mit Metallstangen zu erreichen, die mit einem Seilzug empor bzw. entlang der Schiene gezogen werden konnte. In einem zweiten Schritt versuchten wir dies mit einer Zahnstange zu erreichen. Wegen immer wieder neuen Problemen, welche teils noch ungelöst sind, und durch mangelnde Zeit konnte dieses Modell leider (noch) nicht fertig gestellt werden. Die Ansätze waren sehr interessant, doch Instabilität und Probleme bei der konkreten Umsetzung verhinderte uns dies zu ermöglichen.

Abbildung 94: Drehplattform

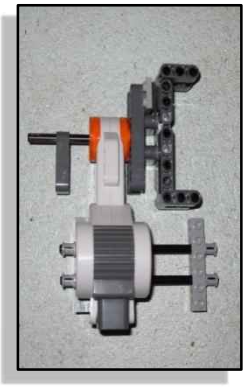


Abbildung 95: Seitenarm

10.3 Aktuelles Modell

Da die Zeit knapp wurde mit einem funktionstüchtigen Modell, sahen wir uns gezwungen, den zweiten Typ liegen zu lassen und verbesserten bzw. änderten das erste Modell. Wir wollten noch einen zusätzlichen Arm integrieren, sodass die Kippbewegung des Würfels besser funktionieren würde und der Würfel nicht mehr verkantet. Im Unterschied zur ersten Variante besteht also dieses Modell nicht nur aus der Drehplattform und dem Kipparm, sondern zusätzlich noch aus einem Stossarm, welcher das reibungslose Kippen und genaueres Drehen einer Ebene gewährleisten sollte. Das gebaute Modell sah viel versprechend aus, nun ging es ans programmieren. Nach etlichen Feinabstimmungen auf Hardware- und Softwarebasis, hatten wir unseren ersten funktionstüchtigen Roboter und wir führten erste erfolgreiche Testläufe durch.

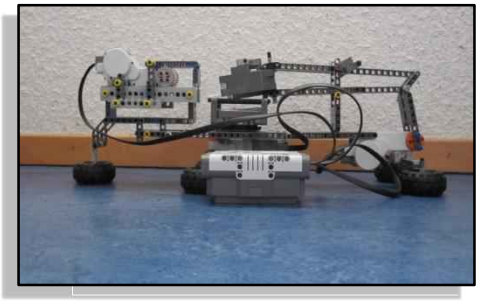


Abbildung 96: aktuelles Modell

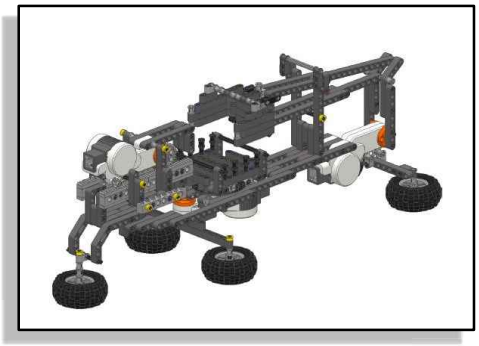


Abbildung 97: Plan vom aktuellem Modell

Da die Programmierung des Roboters und das Programm *Cube Transformer* in verschiedenen Programmiersprachen geschrieben wurden, bzw. unabhängig voneinander programmiert wurden, kamen weitere Probleme hinzu. Die Verbindung der beiden Bereiche war (und leider auch ist) das grösste Problem. Die Zugfolge, die das Programm Cube Transformer ausgibt, muss irgendwie auf den NXT geladen werden können.

10.4 Steuerung des NXT Roboters

Um den Roboter zu programmieren, mussten wir uns für eine Software entscheiden. Damit wir mehr Möglichkeiten zu haben, wollten wir den Roboter nicht mit der Software von Lego programmieren, sondern entschieden uns für einen Editor, basierend auf der Sprache Java. Nun ist es aber so, dass der Lego NXT Roboter nicht von vornherein fähig ist, Java zu „verstehen“. Es muss eine andere Firmware geladen werden, welche von begeisterten Hobbyprogrammierern geschrieben wurde. Zusätzlich mussten einige Treiber installiert werden und Einstellungen am Computer vorgenommen werden, damit die Kommunikation zwischen PC und Roboter funktioniert. Ausserdem muss man sich mit einigen spezifischen Befehlen bekannt machen, die man benötigt, um die Motoren anzusteuern. Gut dokumentiert zum Programmieren mit Java für Lego Roboter ist auf der Seite *www.legorobotik.ch*, sie beinhaltet sogar einen eigenen Online-Editor, mit dessen Hilfe es einfach gelingt den Roboter zu programmieren. Diesen Editor haben wir auch zu Beginn benutzt.

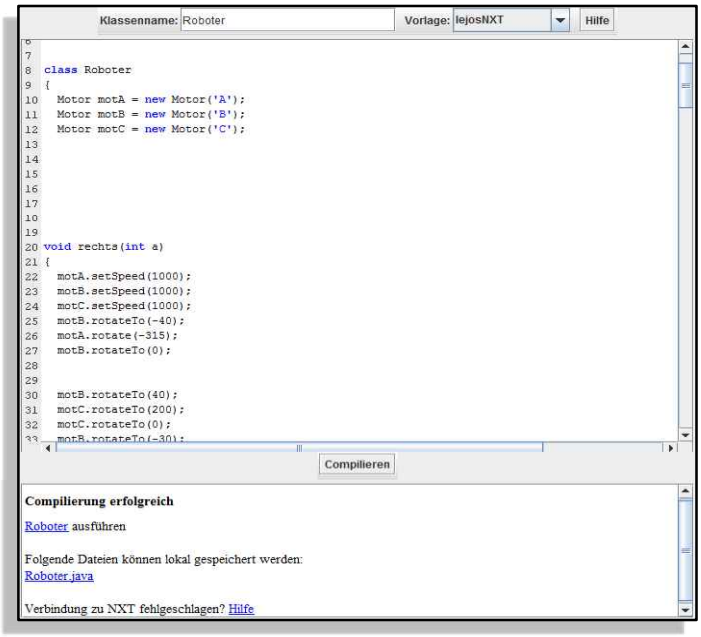


Abbildung 98: Online Editor auf leforobotik.ch

Nach erfolgter Kompilierung bietet er direkt die Option, den Code an den Roboter zu senden oder ihn abzuspeichern. Durch den Online-Editor ist es jedoch nicht möglich, eine Datei einzulesen, da er online auf einem externen Server arbeitet. Daher sind wir auf das Programm BlueJ umgestiegen, das mit Hilfe einer Erweiterung für NXT fast so einfach zu bedienen ist wie der Online-Editor.

Um den Roboter dazu zu bringen, was wir von ihm wollen, müssen wir nur die Motoren ansprechen und diese durch ihre Drehbewegung in Gang setzten. Bei unserem letzten Modell benannten wir die Motoren wir folgt:

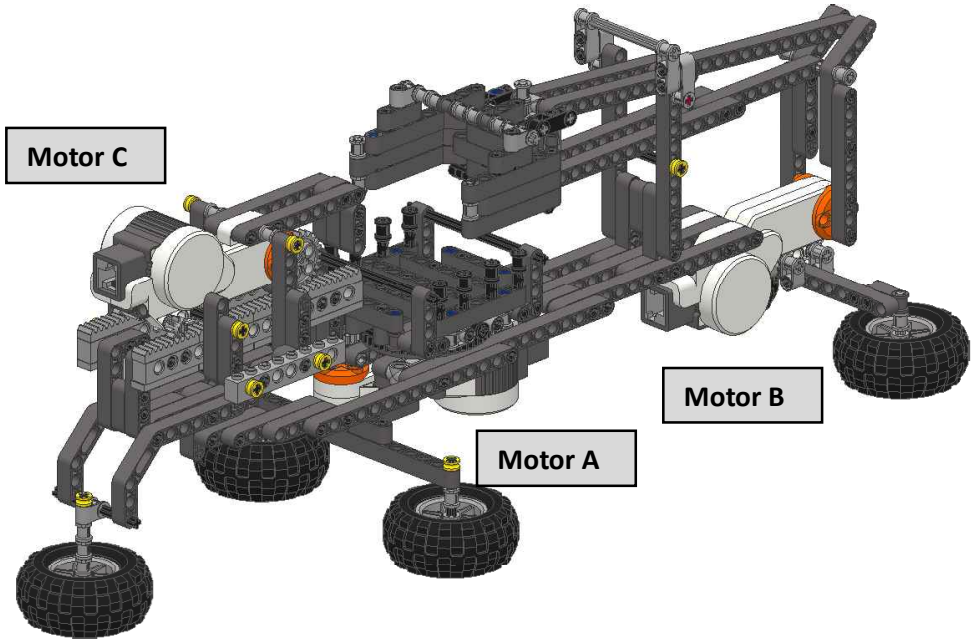


Abbildung 99: Motorenzuordnung von unserem Modell

Um diese Motoren ansprechen zu können, gibt es verschiedene Befehle. Man kann die Motoren im Uhrzeigersinn oder aber auch im Gegenuhrzeigersinn laufen lassen. Weiter kann man die Motoren stoppen, wobei die aktuelle Position fixiert wird. Um die Drehbewegung genauer zu bestimmen gibt es drei Möglichkeiten; die Geschwindigkeit der Drehung angeben oder den Drehwinkel, der ausgeführt werden soll, bzw. zu welchem Winkel der Motor gedreht werden soll.

Befehl	Erklärung
forward()	Dreht Motor vorwärts
backward()	Dreht Motor rückwärts
stop()	Stoppt den Motor
setSpeed()	Setzt Geschwindigkeit des Motors
rotate()	Dreht gewünschter Winkel in Grad (additiv)
rotateTo()	Dreht zu gewünschtem Winkel (konkret)

Angewendet werden diese Befehle stets in Verbindung mit dem jeweiligen Motor in der Form: NameDesMotors.Befehl(Parameter). Also konkret z.B.: *motA.forward();*.

Das Programm Cube Transformer gibt eigentlich Zugfolgen mit der Notation aus, welche Seitenzüge ausgeführt werden unter der Annahme, dass die Achsen fixiert sind. Somit ist zum

Beispiel ein L' immer ein Seitenscheibenzug auf der Seite mit dem grünen Mittelstück. Doch der Roboter muss wissen, welche Seite er drehen muss, ohne dass die Achse im Raum fixiert ist. Das heisst, es kommt darauf an, wie der Würfel aktuell im Raum bzw. auf der Drehplattform liegt. Zuerst lösten wir das Problem so, dass der Würfel immer wieder in die Anfangsposition zurückgeführt wurde und den Würfel im Raum immer wieder zur anvisierten Seite dreht, um einen Seitenscheibenzug auszuführen. Dies führte zu einer umständlichen Lösung. Die dabei verloren gegangene Zeit wollten wir natürlich einsparen und versuchten die Zugfolge an dieses konkrete Modell anzupassen. Wie noch in *Kapitel 11.1* beschrieben wird, werden nun also spezifisch auf dieses Robotermodell die Zugfolgen umgerechnet, sodass die konkrete Lage des Würfels im Raum berücksichtigt wird. Die Notation dieser umgerechneten Züge bedeuten also, dass die Seiten im Raum beschrieben werden. Ein L' bedeutet bei dieser umgerechneten Notation also, dass die aktuelle linke Seite nach unten gedreht werden muss und ein dreifacher Seitenscheibenzug an dieser Seite ausgeübt wird.

Dies bedeutet also, dass der Roboter im Stande sein muss, jede Seite nach unten zu transportieren und einen einfachen, doppelten oder dreifachen Seitenzug auszuführen. Die Bezeichnungen für vorne, oben usw. sind von der *Abbildung 100* zu entnehmen, wobei die Farbzuordnungen für die jeweiligen Seiten wie bis anhin gelten. Jedoch sind es jetzt die Positionen im Raum, welche die Seitennotationen repräsentieren. So ist zum Beispiel die Lage, welche in der *Abbildung 100* die rote Seite einnimmt, immer die vordere Seite oder die obere (aktuell weisse) Seite immer die obere Seite, egal welches Mittelstück auf dieser Seite liegt.

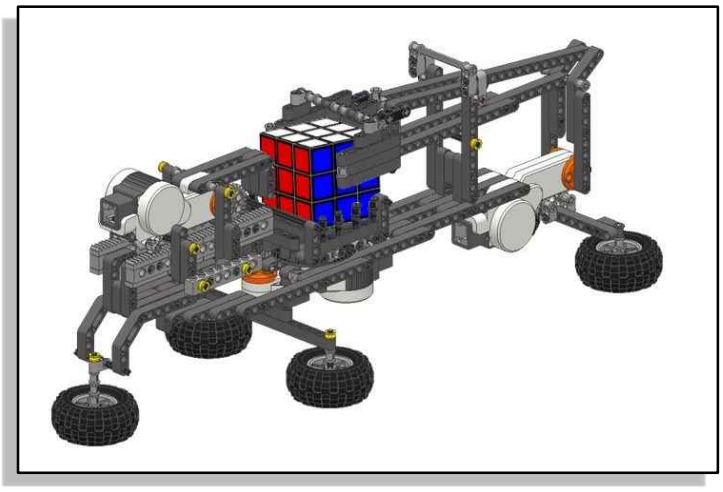


Abbildung 100: Lage des Würfels auf dem Roboter

Um diese sechs verschiedenen Seiten in die untere Ebene zu bringen, müssen die Motoren A, B und C sechs verschiedene Abläufe von Ausführungen beherrschen.

Am Beispiel der Seitenscheibendrehung der aktuellen Seite auf der Position im Raum rechts soll hier das Prinzip illustriert werden.

Es wird eine Funktion *void rechts(int a)* mit dem Parameter *a* aufgerufen, der die Art des Zuges charakterisiert, also ein einfacher, doppelter oder dreifacher Seitenscheibenzug. Zuerst werden die Geschwindigkeiten der Motoren initialisiert, dabei bedeutet der Parameter in der Klammer die Anzahl Umdrehungen pro Minute.

```
void rechts(int a)
{
  motA.setSpeed(1000);
  motB.setSpeed(1000);
  motC.setSpeed(1000);
```

Code 40

In einem weiteren Schritt wird der Motor B auf -40° gedreht, das bedeutet, dass sich der Kipparm vom Würfel wegbewegt und ihn somit frei gibt. Danach dreht sich Motor A um -315°, sodass die Drehplattform mit Würfel um 90° gedreht wird, der Wert in der Klammer entspricht nicht den 90°, da der Motor durch eine Übertragung mit Zahnrädern mit der Drehplattform verbunden ist. Am Ende dieses Teils bewegt sich der Kipparm wieder in die ursprüngliche Position zurück. Hier wird noch kein Seitenscheibenzug ausgeführt, lediglich den Würfel im Raum gedreht, nämlich so, dass durch die Kippbewegung die ursprünglich rechte Seite auf die Drehplattform zu liegen kommt.

```
motB.rotateTo(-40);
motA.rotate(-315);
motB.rotateTo(0);
```

Code 41

Bei diesem Abschnitt bewegt sich der Kipparm zum Würfel hin und bringt ihn beinahe zum Kippen. Um den Würfel ganz zu drehen benötigt es eine Stossbewegung von der anderen Seite, diese wird durch Drehen des Motors C vollbracht, welcher sich auf einer Zahnschiene linear nach vorne bewegt. Danach begibt sich Motor C wieder in die ursprüngliche Position. Der Kipparm hingegen macht zuerst einen kleinen Schlenker nach hinten, bevor er sich wieder in die ursprüngliche Position begibt, um den Würfel wieder kompakt zu umgeben.

```
motB.rotateTo(40);
motC.rotateTo(200);
motC.rotateTo(0);
motB.rotateTo(-30);
motB.rotateTo(0);
```

Code 42

In diesem Block wird nun der eigentliche Seitenzug ausgeführt. Zuerst fährt der Stossarm bis zum Würfel vor und danach drückt der Kipparm von der anderen Seite bis so stark dagegen, dass der Würfel sehr kompakt eingeklemmt wird, um so den Toleranzbereich zu minimieren, so dass die Ebene schlussendlich möglichst exakt gedreht werden kann. Während der Würfel zwischen den beiden Armen eingeklemmt ist, erfolgt die Drehung der Drehplattform und somit die Ausführung des Seitenscheibenzuges.

```
motC.rotateTo(160);
motB.forward();
motA.rotate(a*315+16);
motA.rotate(-16);
motB.stop();
motC.rotateTo(0);
motB.rotateTo(0);
```

Code 43

Der Parameter a bestimmt darüber wie viele 90°-Drehungen gemacht werden sollen. Die Plattform dreht dabei ein wenig zu weit und schlussendlich wieder zurück um den bestehenden Toleranzbereich auszugleichen. Wenn die Drehung der untersten Ebene erfolgt ist, werden der Kipp- und der Stossarm wieder an ihre ursprünglichen Positionen zurückgefahren.

Die anderen fünf Drehungen sind analog zu diesem. Es ist immer wieder eine Verbindung aus „Leerdrehen“ bzw. eine Drehung des Würfels im Raum, einer Kippbewegung um eine Achse im Raum und einer Drehung der unteren Seite, wobei die beiden Arme den Würfel einklemmen.

Die in Cube Transformer umgerechneten Zugfolgen werden in einer *.txt* Datei gespeichert und an dieser Stelle eingelesen. Mit 17 Fallunterscheidungen werden die Funktionen für die Seitenscheibenzüge mit unterschiedlichen Parametern aufgerufen. Da der Toleranzbereich bei der Umschliessung des Würfels ein paar Millimeter gross ist, wird auf Drehungen der Drehplattform im Gegenuhrzeigersinn verzichtet, dadurch wird die Gefahr der Verkantung minimiert.

Nach dem Einlesen wird das Programm entweder per Bluetooth oder USB an den NXT gesendet, wo die Motoren die Befehle ausführen.

```
if (input[b]==0){roboter.vorne(1);}
if (input[b]==1){roboter.vorne(2);}
if (input[b]==2){roboter.vorne(3);}

if (input[b]==3){roboter.oben(1);}
if (input[b]==4){roboter.oben(2);}
if (input[b]==5){roboter.oben(3);}

if (input[b]==6){roboter.hinten(1);}
if (input[b]==7){roboter.hinten(2);}
if (input[b]==8){roboter.hinten(3);}

if (input[b]==9){roboter.unten(1);}
if (input[b]==10){roboter.unten(2);}
if (input[b]==11){roboter.unten(3);}

if (input[b]==12){roboter.rechts(1);}
if (input[b]==13){roboter.rechts(2);}
if (input[b]==14){roboter.rechts(3);}

if (input[b]==15){roboter.links(1);}
if (input[b]==16){roboter.links(2);}
if (input[b]==17){roboter.links(3);}
```

Code 44: Fallunterscheidungen

11 Verbindung von Algorithmus und Roboter

11.1 Zugfolge umrechnen

Wie ein Kapitel zuvor darauf verwiesen, muss die eigentliche Zugfolge im *Cube Transformer* umgerechnet werden. Diese neue Zugfolge ist spezifisch auf dieses Modell zugeschnitten, wo nur eine Seite direkt angesprochen werden kann. Dabei wird die Lage des Würfels im Raum berücksichtigt, sodass nicht immer eine Retourdrehung des Würfels im Raum in die Ursprungslage erfolgen muss.

Im Programm *Cube Transformer* läuft im Hintergrund diese Umrechnung bei jedem Durchgang automatisch ab. Dabei werden die Züge in diese modellspezifische Zugfolge umgerechnet und in einer *.txt* Datei gespeichert, auf die in *BlueJ* zugegriffen werden kann.

```
int seiten[]={0,1,2,3,4,5};
int outputZuege[]= new int [30];

PrintWriter robot=createWriter(pfad+"robotZuege.txt");
```

Code 45: Initialisierung und Erstellung des Files

In einem ersten Schritt wird ein Array erstellt, das den Seitenpositionen im Raum die jeweilige Seitennummer zu ordnet. Ein weiteres Array wird erstellt, um die umgerechneten Züge zu speichern. Um die neue Zugfolge extern speichern zu können bedarf es der Erstellung eines Files *robot*.

```
void robotUmwandlung(){
    for(int i=0;i<LOESUNG[32];i++){
      seitenUmwandlung(i);
    }
    robot.print(LOESUNG[32]+", ");
    for(int i=0;i<LOESUNG[32];i++){
      robot.print(outputZuege[i]+", ");
    }
    robot.close();
  }
}
```

Code 46: neue Funktion wird mit versch. Zügen aufgerufen

Die eigentliche Zugfolge, also unter der Bedingung der statischen Achse, ist im Array *LOESUNG[]* gespeichert worden, im *Feld[32]* ist die Länge der Zugfolge abgespeichert worden. In der Funktion *robotUmwandlung()* wird zuerst in einer for-Schleife jeder Zug in der Notationsweise 1 mit der zweiten Funktion *seitenUmwandlung()* aufgerufen. Diese wird im nächsten Abschnitt besprochen. Nachdem diese Umwandlung bei allen Positionen der Zugfolge erfolgt ist, wird an erster Stelle in der externen Datei die Länge der Zugfolge

abgespeichert. Danach wird das neue, spezifische Array der Zugfolge in der Notationsweise 2 in die externe Datei *robot* abgespeichert.

```
void seitenUmwandlung(int nr){
  int ort=0;
  int anzDreh=LOESUNG[nr]%3;
  int wert=int(LOESUNG[nr]/3);

  for (int i=0;i<6;i++){
    if(wert==seiten[i]){
      ort=i;
    }
  }
  outputZuege[nr]=ort*3+anzDreh;
  faelle(ort);
}
```

Code 47: Seite wird gesucht

Diese Funktion *seitenUmwandlung()* wird also an jeder Position in der alten Zugfolge aufgerufen. Es werden drei Variablen definiert und gerade initialisiert. Die Variable *ort* steht für die Position der gesuchten Seite im Raum, diese wird auf 0 gesetzt. Die zweite Variable *anzDreh* ist ein Platzhalter für die Anzahl, wie oft der Zug ausgeführt wird. Das heisst also konkret, es wird untersucht, ob dieser Zug in der Notationsweise 1 für einen einfachen, einen doppelten oder einen dreifachen Seitenscheibenzug steht. Dies kann mit der Modulodivision durch drei erreicht werden, da die Werte im Array *LOESUNG[]* in Zahlen von 0 bis 17 abgespeichert sind, wobei die 3 Arten pro Seitenscheibenzug nacheinander aufgeführt werden. Mit der Variable *wert* wird die Art des Seitenscheibenzugs abgefragt und abgespeichert, indem der Quotient - bei einem Divisor von 3 und Dividenden von der Zahl für die Zugart in der Notationsweise 1 - nach unten gerundet wird.

Iterativ wird nun der *wert*, also eigentlich das entsprechende Mittelstück, an jeder Position im Array *seiten[]* gesucht. Ist eine Übereinstimmung gefunden, wird die Position in der Variable *ort* vermerkt. Nun wissen wir also, auf welcher Seitenposition im Raum diese Seite, ausgedrückt durch die „Farbe“ des Mittelstücks, konkret liegt. Die Bedeutung für den Roboter ist bereits besprochen, nämlich nun muss diese Seite in die untere Ebene gebracht werden und mit der Anzahl (*anzDreh*) gedreht werden. Um die Art des Zuges (*wert*) und die Anzahl in eine Zahl von 0 bis 17 umwandeln, wird diese hochgerechnet und in das entsprechende Feld des Arrays *outputZuege[]* gespeichert. Diese Drehung des Würfels im Raum hat Auswirkungen auf die Seitennummern in Korrelation mit den Seitenpositionen. Diese Veränderungen oder Auswirkungen müssen natürlich auch wieder nachgeführt werden, sodass bei der Analyse des nächsten Zuges die korrekte Position der Seitennummer gefunden werden kann. Dies wird mit dem Aufruf *faelle(ort)* durchgeführt.

```
void faelle(int a) {
  int tempSeiten[]=new int [6];
  for (int i=0;i<6;i++){
    tempSeiten[i]=seiten[i];
  }
      .
      .
      .
}
```

Code 48: Seitenumwandlung

In dieser Funktion *faelle()* wird zuerst eine Kopie der Seitennummern auf den Seitenpositionen erstellt, sodass keine zirkuläre Zuweisung erfolgt. Danach wird je nach Eingangsparameter *a* die Auswirkungen auf die Seitennummern verändert. Bei a = 3 wird nichts verändert, da dabei die Positionen der Seiten im Raum nicht verändert werden, denn die zu drehende Seite liegt bereits unten.

```
if (a==0){
  seiten[0]=tempSeiten[1];
  seiten[1]=tempSeiten[2];
  seiten[2]=tempSeiten[3];
  seiten[3]=tempSeiten[0];
}
```

```
if (a==1){
  seiten[0]=tempSeiten[2];
  seiten[1]=tempSeiten[3];
  seiten[2]=tempSeiten[0];
  seiten[3]=tempSeiten[1];
}
```

```
if (a==2){
  seiten[0]=tempSeiten[1];
  seiten[1]=tempSeiten[0];
  seiten[2]=tempSeiten[3];
  seiten[3]=tempSeiten[2];
  seiten[4]=tempSeiten[5];
  seiten[5]=tempSeiten[4];
}
```

```
if (a==4){
  seiten[0]=tempSeiten[1];
  seiten[1]=tempSeiten[5];
  seiten[2]=tempSeiten[3];
  seiten[3]=tempSeiten[4];
  seiten[4]=tempSeiten[2];
  seiten[5]=tempSeiten[0];
}
```

```
if (a==5){
  seiten[0]=tempSeiten[1];
  seiten[1]=tempSeiten[4];
  seiten[2]=tempSeiten[3];
  seiten[3]=tempSeiten[5];
  seiten[4]=tempSeiten[0];
  seiten[5]=tempSeiten[2];
}
```

Code 49: verschiedene Änderungen der Seiten im Raum

Je nach Parameter *a* bzw. die Art der Drehung des Würfels im Raum, werden die Seitennummern auf andere Positionen übergeben. Bei a = 0 werden zum Beispiel die Seiten so übergeben, wie wenn der Würfel gekippt wird, also eine Drehung des Würfels so, dass die vordere Seite in die untere Seitenebene gelangt. Sind diese Auswirkungen implementiert, ist die Umrechnung fertig und die neue Zugfolge in der Notationsweise 2 können vom File *robot.txt* geladen werden.

11.2 Kameraverbindung

Damit die Farben vom konkreten Würfel eingelesen werden können, erwarben wir eine Kamera. Durch Import der Bilder in das Programm Processing können die einzelnen Pixel analysiert werden. Um den Toleranzbereich der verschiedenen Lichtumgebungen zu erhöhen, versuchten wir eine dynamische Auslesung der Farbwerte zu programmieren. Das heisst konkret, dass wir die Farben aus den Mittelteilen, die bekanntlich immer fix und repräsentativ für eine Seite stehen, auslesen und mit den anderen Farbflächen vergleichen bzw. versuchen sie einzuordnen. Da die Kamera jedoch stark auf Lichtveränderungen reagiert bzw. eine automatische Lichtanpassung vornimmt, ist die Fehlerquote auch durch ausgiebige Versuche nicht auf 0 % minimierbar gewesen. So werden die Farben nicht bei allen Lichtverhältnissen gleich gut ausgelesen.

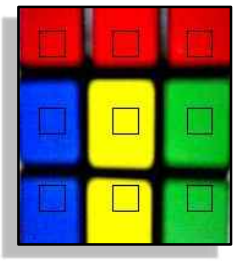


Abbildung 101

Es werden pro Fläche immer 81 Pixel ausgelesen und der Mittelwert von diesen berechnet. Da wir die Zuordnung der sechs Farben auf die Flächen relativ zu den Farbwerten der Mittelstücke machen wollen, werden die aufgenommenen Farbwerte vorerst in Arrays gespeichert, solange wir noch nicht alle Mittelstücke eingelesen haben. Sind alle Flächen eingelesen, so werden, mit unterschiedlichen Toleranzbereichen, die Farbwerte der einzelnen Flächen in die nach unten und oben erhöhten Schranken der Farbwerte der Mittelstücke eingeordnet und eine Zahl von 1 bis 6 zugeordnet. Danach werden die Farbwerte in die Arrays aus Kapitel 1.3 gespeichert.

11.3 Koordinaten initialisieren

Um aus den Flächenarrays die Arrays für die Positionen sowie Orientierungen der Ecken und Kanten zu erhalten, müssen die Flächen auf den Eck- bzw. Kantenpositionen analysiert werden.

```
void InitialisiereKoord(){
  for(int i=0;i<=7;i++){
    gibEStrich(1,i);
    EckKoord[i]=ecke;
    EckKoordOri[i]=oriEck;
  }
  for(int i=0;i<=11;i++){
    gibKStrich(1,i);
    KantKoord[i]=kante;
    KantKoordOri[i]=oriKant;
  }
}
```

Code 50: an jeder Position wird Ecken- und Kantennummern gesucht

Bei der Initialisierung der Ecken- und Kantenarrays wird für jede Position eine Funktion *gibEStrich(1,i)* bzw. *gibKStrich(1,i)* aufgerufen und die dazugehörigen Farbflächen untersucht, welches Objekt an dieser Stelle ist. Die Zahl 1 als Eingangsparameter für die beiden Funktionen bedeutet das Auslesen der Ecken auf den Flächen des Ausgangswürfels. Eine 2 würde vermerkt werden, wenn die Flächen auf dem Zielwürfel untersucht werden sollen.

In diesen beiden Funktionen werden jeweils die entsprechenden Farbflächen den untenstehenden Funktionen übergeben, wo die Zuordnungen der jeweiligen Objekten, Ecken und Kanten, zu den Positionen erfolgt. Ebenso werden die Orientierungen dieser besagten Objekten ebenfalls den Positionen zugeordnet, sodass die Gesamtkoordinaten errechnet werden können.

```
void gibEckaus3Farb(int col1, int col2, int col3){
  int colg=col1*col2*col3;

  if(colg==12){ecke=0;}
  if(colg==10){ecke=1;}
  if(colg==30){ecke=2;}
  if(colg==36){ecke=3;}
  if(colg==24){ecke=4;}
  if(colg==20){ecke=5;}
  if(colg==72){ecke=6;}
  if(colg==60){ecke=7;}

  if(col1==2 || col1==4){oriEck=0;}
  if(col2==2 || col2==4){oriEck=1;}
  if(col3==2 || col3==4){oriEck=2;}
}
```

```
void gibKantaus2Farb(int col1, int col2){
  int colg=col1*col1+col2*col2;
  if(colg==37){kante=0;}
  if(colg==26){kante=1;}
  if(colg==34){kante=2;}
  if(colg==45){kante=3;}
  if(colg==40){kante=4;}
  if(colg==5){kante=5;}
  if(colg==29){kante=6;}
  if(colg==13){kante=7;}
  if(colg==52){kante=8;}
  if(colg==17){kante=9;}
  if(colg==41){kante=10;}
  if(colg==25){kante=11;}
  if(col2==5 || col2==6){oriKant=0;}
  if(col1==2 || col1==4){oriKant=0;}
  if(col2==2 || col2==4){oriKant=1;}
  if(col1==5 || col1==6){oriKant=1;}
}
```

Code 51: Eck- bzw. Kantennummern werden durch Flächenfarben bestimmt

Sind diese Koordinaten berechnet, kann mit dieser konkreten Stellung die Tiefensuche gestartet werden. Somit würde sich also der Kreis schliessen.

Reflexion

Wir hatten uns ein ehrgeiziges Ziel gesetzt, welches mit einem grossen Arbeitsaufwand verbunden war. Wir waren uns darüber bereits im Voraus im Klaren, dass wir unzählige Stunden vor dem PC oder am Herumtüfteln verbringen werden. Meist waren wir mit ungeheurem Elan und Motivation bei der Arbeit und gaben nie auf. Wir haben auch unser Ziel nie aus den Augen verloren und bei einigen Entscheidungen siegte zum Glück der Pragmatismus vor dem Perfektionismus. Wir sind stolz, unsere Zielsetzung mehr oder weniger erreicht zu haben.

Doch gab es immer wieder Momente, die einem zum Verzweifeln brachten. Die Konstruktion eines fähigen Roboters zum Beispiel und danach die Abstimmung mit der Software zu finden, raubte viel Zeit und Geduld. Wir standen manchmal vor grossen Problemen und suchten dann gemeinsam nach brauchbaren Lösungsansätzen in intensiven Diskussionen. Die Ungewissheit, ob man es nun schaffen wird, raubte einem zeitweise die Motivation. Kleine Zwischenziele anzuvisieren war besonders wichtig, um sich in der ganzen Arbeit nicht zu verlieren.

Wo sollte man zum Beispiel bei der Roboterkonstruktion beginnen? Zuerst musste man sich den Roboter vorstellen können, wie er die Ebene ansprechen soll und wie viele Motoren benötigt werden. Als wir die ersten Versuche starteten, bemerkten wir, dass es schwierig ist mit Legobausteinen ein stabiles Modell zu erstellen, so dass die Züge exakt ausgeführt werden können. Schlussendlich war die Erleichterung gross, als das erste funktionierende Modell erstellt wurde. Wir versuchten noch ein schnelleres Modell zu entwickeln, wo der Würfel mit vier Seiten ansprechbar sein sollte. Dies wollten wir mit Hilfe von Holz und Metall konstruieren, doch es ergaben sich viele Probleme, wie zum Beispiel die Fixierung einer Achse, die stabil sein musste aber trotzdem drehbar. Nach etlichen weiteren Problemen mussten wir einsehen, dass wir bis zum Abgabetermin nicht im Stande waren, dieses Modell zu vollenden. Bei Gelegenheit wird dies eventuell noch fortgesetzt.

Bei der Programmierung raubte die Fehlersuche unzählige Stunden. Es brauchte oft außerordentliche Selbstdisziplin, um manchmal Nächte lang einen einzigen Fehler zu finden. Ich kann mich noch an eine Nacht erinnern, wo ich daran war, die Tabelle der mittleren Kanten zu generieren. Die generierten Daten auf ihre Korrektheit zu verifizieren stellte sich ziemlich schwierig an. Nach einigen Tests musste ich wirklich feststellen, dass die erstellten Tabellen falsch waren. Als ich dann endlich im Array für die Binomialkoeffizienten ein Minus vor einer 1 zu viel eruieren konnte, führte ich im wahrsten Sinne einen Freudentanz auf. Das Gefühl war herrlich und liess alle mühsamen Stunden vergessen. Den Überblick zu behalten bei den riesigen Mengen an Daten war zeitweise eine grosse Herausforderung. Doch in den

Momenten, wo plötzlich Zugfolgen ausgespuckt worden sind, sind wirklich unvergesslich. Man arbeitet auf ein Ziel hin und kann in diesen abstrakten Zahlen kaum einen Anhaltspunkt finden, doch plötzlich ist eine Zugfolge verifizierbar und führt zu einem Teilergebnis.

Die beiden Bereiche zu verknüpfen ist dagegen nicht optimal gelungen. Wir würden sagen, es lag nicht an mangelnder Harmonie bei der Zusammenarbeit von uns sondern bei der Grundkonzeption. Wir hätten zu Beginn beim Algorithmus mit Java beginnen sollen, um die Bereiche sauber vereinigen zu können. Dies ist uns nicht wirklich gut gelungen. Es sind zwei Bereiche, in denen wir unsere Erwartungen und Ziele mehrheitlich erreichen konnten, doch mit der Zusammenführung sind wir überhaupt nicht zufrieden. Es bedarf bei jedem Durchgang einige manuelle Eingriffe, um die Daten am richtigen Ort zu haben.

Durch dieses Projekt sahen wir in sehr unterschiedliche Bereiche der Informatik hinein. Grosse Datenmengen, Farberkennung, Suchverfahren und Grundeinblicke in die Robotik waren Themen, mit denen wir uns auseinandersetzten konnten. Auch das Arbeiten als Team ermöglichte uns wertvolle Erfahrungen, sind doch grössere Projekte in dieser Fachrichtung heute nur in Gruppen umsetzbar. Durch dieses Projekt wurde uns klar, dass dies unsere Studienrichtung sein wird: Kornel beabsichtigt ein Maschinenbaustudium, währenddem ich das Ziel habe, Informatik zu studieren.

Wir sahen hinter die Fassaden des Rubik's Cube und die riesige Anzahl von möglichen Stellungen. Dieses Thema an sich faszinierte uns umso mehr, desto länger wir uns damit beschäftigten. Die Begeisterung und das Interesse war ein riesiger Antrieb um so viel Zeit zu investieren. Wir sehen auf eine nervenaufreibende Zeit zurück, die glücklicherweise relativ erfolgreich abgeschlossen werden konnte.

Bibliographie

Internetquellen

2-Phasen-Algorithmus von Herbert Kociemba

http://kociemba.org/cube.htm [Stand 17.10.2010]

Wikipedia: Zauberwürfel

http://de.wikipedia.org/wiki/Zauberw%C3%BCrfel [Stand 17.10.2010]

Neue Zürcher Zeitung

http://www.nzz.ch/nachrichten/wissenschaft/20_ist_die_gotteszahl_1.7212437.html [Stand 17.10.2010]

Abbildungen

Abbildung 1

http://www.brechtel-folkers.de/magellan/magellan1/Aufloesung.html [Stand 17.10.2010]

Abbildungen 2-5

http://www.jaapsch.net/puzzles/circleman.htm [Stand 17.10.2010]

Abbildungen 6-7

http://de.wikipedia.org/wiki/Zauberw%C3%BCrfel [Stand 17.10.2010]

Die weiteren Abbildungen erstellten wir mit diversen Programmen, darunter Gimp, Cube Transformer, Processing, Word und Excel.

Anhang

Deklaration

„Wir erklären hiermit

- dass wir die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen verfasst haben,
- dass wir auf eine eventuelle Mithilfe Dritter in der Arbeit ausdrücklich hinweisen,
- dass wir die Schulleitung und die betreuende Lehrperson informieren, wenn wir diese Maturaarbeit bzw. Teile oder Zusammenfassungen davon veröffentlichen, oder Kopien dieser Arbeit an Dritte aushändigen werden.“

Ort: Grosswangen

Datum: 18. Oktober 2010

Unterschrift:

Ort: Buttisholz

Datum: 18. Oktober 2010

Unterschrift:

Erstes Programm

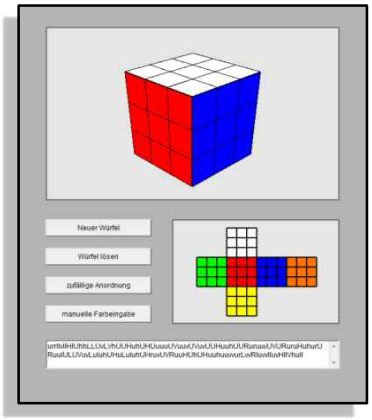


Abbildung 102: erstes Programm

Wenn man alle flächenbezogenen Auswirkungen definiert hat (Kap. 2.1), so kann man einen Würfel mit 54 Feldern erstellen, auf welche die Seitenzüge anwendbar sind. Wir erstellten vor Eingabe des Projekts zur Verifizierung der Durchführbarkeit ein kleines Programm, das in der Lage ist, jeden Würfel zu lösen. Allerdings nach einem manuellen Algorithmus. Dieser ist aus lauter Fallunterscheidungen aufgebaut, je nachdem wie der Würfel gerade aussieht, werden programmierte Teilzugfolgen ausgeführt und die nächste Stellung wiederum mit Fallunterscheidungen nach den weiteren Zugfolgen untersucht.

Dieses Programm funktioniert einwandfrei, ist aber nicht wirklich interessant. Einerseits die grosse Anzahl an Zügen die ausgegeben werden, anderseits kann das auch jeder Mensch. Auf der CD ist dieses Programm unter *Manuell_0* vorhanden.

Bauanleitung

Für das aktuelle Modell erstellten wir einen Plan, dieser ist in einem separaten Dossier beigelegt.