Математичка гимназија

МАТУРСКИ РАД

- из програмирања -

Алгоритам за слагање Рубикове коцке

Ученик:
Стефан Миленковић, IVe

Ментор:
Јовица Милисављевић

Београд, јун 2020.

Садржај

1 Увод **1**

2 Особине Рубикове коцке **3**

2.1 Елементи Рубикове коцке . . . 3
2.2 Потези . . . 3
2.3 Пермутације и оријентације коцкица . . . 4
2.4 Теорија група . . . 8

3 Алгоритам **11**

3.1 Основна идеја . . . 11
3.1.1 Тислтвејтов алгоритам . . . 11
3.1.2 Косиембин алгоритам . . . 12
3.2 Представљање коцке . . . 13
3.2.1 Мапирање варијација са понављањем . . . 14
3.2.2 Мапирање комбинација без понављања . . . 15
3.2.3 Мапирање пермутација без понављања . . . 16
3.2.4 Табеле потеза . . . 16
3.3 Претрага . . . 17
3.3.1 Претрага у дубину . . . 17
3.3.2 Претрага у ширину . . . 18
3.3.3 IDA* и Корфов алгоритам . . . 19
3.3.4 Табеле одсецања . . . 21
3.4 Комплетирање . . . 22
3.5 Резултати . . . 24

4 Закључак **27**

Литература **29**

1 Увод

У другој половини прошлог века, тачније 1974. године, мађарски архитекта Ерне Рубик[1] осмислио је дрвену структуру у облику коцке са идејом да направи механизам у коме би његови делови могли независно да се померају али да као целина остану састављени. У почетку ни сам није увидео да је направио слагалицу, док је није растурио и покушао да је врати у почетно стање. Када је разумео шта је направио, изум је патентирао и комерцијализовао, и тако настаје прва „Магична коцка” која одлази у продају и полако освајање целог света. Од тада па до данас Рубикова коцка постала је готово неизоставна играчка сваког детета о чему и сведоче подаци да је једна од најпродаванијих играчака на свету. Пошто је од самог појављивања изазвала велику пажњу настала су бројна такмичења где је циљ што брже сложити ову коцку што је чини још занимљивијом великом броју људи због чињеница да је неко у стању да је сложи за свега неколико секунди. Међутим, иако играчка, главна особина коцке је та да је није лако вратити у почетни положај. Након свега пар потеза постаје изузетно тешко сложити је и управо зато је, осим деци, ова играчка постала занимљива и математичарима.

Поставило се питање како сложити ову коцку и који је то минималан број потеза за који је могуће сложити сваку њену могућу позицију. Пошто проблем делује као превише сложен да би било који човек могао да га реши, такав број потеза назван је *Божји број*, а потенцијални алгоритам који би нашао такво најбоље решење *Божји алгоритам*[2]. Управо у то време долази и до наглог развоја рачунара, па заиста, тражење најбољег решења није радио човек, већ рачунар. Наравно, рачунар ради оно што му се зада и једна од идеја овог рада је да покаже како су то ти људи успели да натерају рачунаре да пронађу оно што их је све занимало.

Потрага је почела. Проблем коцке је што има превише позиција да би их било који рачунар обрадио у реалном времену, и зато је било неопходно да се претрага скрати на неки погодан начин. Значајне помаке ка постизању тог циља направили су Морвен Тислтвејт[3], Херберт Косиемба[4] и Ричард Корф[5] чији су алгоритми променили начин сагледавања коцке и начине на који ће се решење тражити.

Прва процена Божјег броја јавља се врло брзо када се налази да он мора да буде бар 18, а већ 1995. године се показује да је доња граница 20 што се горње границе тиче, она је доста дуже испитивана. Прва њена процена јавља се 1979. године и била је 277 потеза. Касније се значајно смањује, а пресудну улогу имала су већ пометута три алгоритма, Твислтвејтов 1981., Косиембин 1992. и Корфов 1997. године. Крајем 20. века показано је да је горња граница не већа од 30.

[1] Ernő Rubik (1944 -)

[2] термин Божји алгоритам се односи и на друге математичке слагалице (нпр. Ханојска кула), али је првобитно настао услед Рубикове коцке

[3] Morwen Thistlethwaite, британски математичар

[4] Herbert Kociemba (1954 -), немачки наставник математике

[5] Richard Korf, амерички програмер и професор

У 21. веку акценат је био на што бољој оптимизацији програма и скраћивању што већег броја случајева и у јулу 2010. године група људи у чијем су саставу Томас Рокики[6] и Херберт Косиемба добија да је Божји број *тачно* 20. Рачунање је трајало неколико недеља и било је извршено на Гугловом серверу (еквивалентно време које би било потребно неком данашњем процесору је око 35 година за такав рачун)[7].

Број потеза је пронађен, али сам алгоритам за проналажење оптималног решења је направедовао како је напредовала снага рачунара и тренутно најбољи програми за оптимално слагање Рубикове коцке могу за одређене позиције коцке да захевају до неколико сати рада. Зато се овај рад бави Косиембиним алгоритмом који је много заступљенији и који се базира на идејама већ поменута три алгоритма са краја 20. века и који не налази оптимално решење, али зато у јако кратком времену избацује решења која су доста блиска оптималном, у просеку око 20 потеза (видети одељак Резултати у трећем поглављу).

Рад је замишљен као пројекат и више је окренут практичној имплементацији, али је за то неопходно дефинисати основне појмове и дати неки теоријски увод што је и урађено у другом поглављу. Треће поглавље садржи објашњење како алгоритам функционише и паралелно су дати и начини конкретне имплементације таквог програма који је у потпуности написан.

[6]Tomas Rokicki, амерички програмер

[7]целокупан код као и детаљи везани за то како су успели да дођу до резултата могу се наћи овде [1]

2 Особине Рубикове коцке

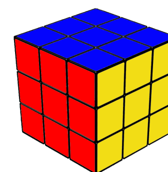
У овом поглављу представљене су дефиниције основних појмова и дат је теоријски увод неопходан за разумевање функционисања алгоритма.

2.1 Елементи Рубикове коцке

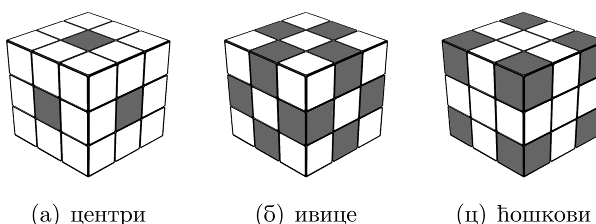
Рубикова коцка је структура у облику коцке којој је свака од страна подељена на 9 мањих квадрата, што ствара слику као да се састоји од $3^3 = 27$ мањих коцки (слика 2.1). Свака од страна коцке је обојена једном бојом и циљ је растурену коцку вратити у почетни положај.

У самом центру коцке се не налази ниједна коцкица што оставља $27 - 1 = 26$ мањих коцки које чине целину. Даље ћемо сваку од тих мањих коцки звати *коцкица* (енг. *cubie*).

Међу њима постоје оне које су обојене само једном бојом. Оне се налазе у центру сваке стране коцке и на њих дозвољени потези не утичу тј. остављају их на истом месту (дозвољени потези су описани у наредном одељку). Тих коцкица има 6 и зовемо их *центри* (слика 2.2(а)) и неће бити толико излагани касније јер нису толико од значаја зато што су код стандардне Рубикове коцке практично фиксни[1].



Слика 2.1



(а) центри (б) ивице (ц) ћошкови

Слика 2.2

Друга, значајнија, група коцкица је она која има по две своје стране обојене. Таквих коцкица има 12 и даље их зовемо *ивице* (енг. *edges*) (слика 2.2(б)).

Последња, трећа, такође значајна, група је она са по три обојене стране којих има 8 и њих зовемо *ћошкови* (енг. *corners*) (слика 2.2(ц)).

2.2 Потези

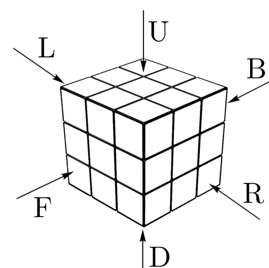
Рубикова коцка има 6 страна и свака од њих може да се ротира за произвољан угао. Ротација неке стране за 360° враћа ту страну у почетни положај и ротације за угао који није

[1] постоје варијације Рубикове коцке где се на странама, уместо једнобојних, налазе шарене слике и онда центри добијају на значају јер је за слагање битна њихова оријентација

дељив са 90° ће нарушити форму коцке. Зато има смисла посматрати само она окретања за углове који су мањи од пуног угла и који су дељиви правим углом, тј. само окретања за 90°, 180° и 270°. Дакле, једна страна има 3 различита потеза што значи да укупно постоји 18 различитих потеза који се могу применити на коцку.

Како су потези условљени странама, потребно је прво увести ознаке страна. Стандардан начин је да се страна која је окренута према нама означава са F (енг. *front*), страна наспрам ње са B (енг. *back*), страна која је окренуто десно од нас са R (енг. *right*), лево са L (енг. *left*), она која је окренута ка горе са U (енг. *up*) и она која је окренута ка доле са D (енг. *down*) (слика 2.3).

Сада са S, S2 и S' редом означавамо потезе који одговарају ротацији стране S за 90°, 180° и 270° у *смеру казаљке на сату*, дакле, математички *негативном* смеру. На пример потез F' значи да је страна F ротирана за 270°, односно, 90° у смеру супронтом од казаљке на сату, а потез F2 значи да је страна F ротирана за 180°.



Слика 2.3

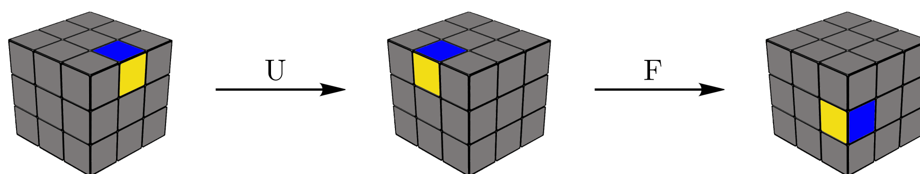
Решење за слагање неке коцке ће бити низ потеза и овде је важно напоменути како се одређује дужина решења. Наиме, природно се намеће приступ где би потезе облика S и S' гледали као дужине 1 (јер се не могу раставити на једноставније), а потезе облика S2 гледали као да су дужине 2, јер представљају композицију два потеза облика S. Међутим, *сваки* од могућих 18 потеза посматра се као дужине 1, дакле, ако би два решења садржала потезе R и R2 она би се гледала као исте дужине.

Чему овакав приступ? И ако неприродан нама, овакав начин олакшава конструкцију алгоритма за тражење решења и зато је и практично узет као стандардан. Такав начин мерења потеза назива се *метрика половне окрета* (енг. *half turn metric - HTM*), а начин мерења само са ротацијама од 90° се назива *метрика четвртине окрета* (енг. *quarter turn metric - QTM*). Божји број износи 20, али у метици половине окрета. Још један показатељ колико је она погоднија је и тај да је за проналажење Божјег броја у метрици четвртине окрета било потребно још додатне четири године и да је он пронађен у августу 2014. и износи 26 потеза[2].

2.3 Пермутације и оријентације коцкица

Пермутације и оријентације коцкица су особине које одређују различите позиције коцке, и зато је неопходно да се прецизно уведу.

Посматрајмо сада само ивице и фиксирајмо само једну од њих (слика 2.4). Било који низ потеза ту посматрану ивицу само пребацује на место неке друге ивице, дакле, применом потеза ивице не могу да пређу у неку другу групу коцкица, нпр. у ћошкове, а аналогно важи и обруно, ћошови не могу да пређу у ивице ма какве потезе примењивали.



Слика 2.4

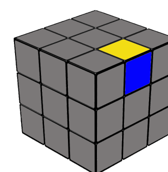
[2] Томас Рокики и Морли Дејвидсон су то показали, више детаља може се наћи овде [1]

Стога ивице имају 12 слободних места на којима могу да се нађу и једна позиција коцке садржи само једну пермутацију тих ивица. Тако се и дефинише пермутација коцкица. С обзиром да се ивице и ћошкови посматрају одвојено, то се разликују пермутације ивица и пермутације ћошоква.

Ивице се обележавају у облику XY, где су X и Y стране које та ивица дели са сложеном коцком. Тако је нпр. означена ивица на првој слици слике 2.4 ивица UR, али је она исто UR и на остале две слике, само је њена локација промењена, односно пермутована је са неким другим ивицама.

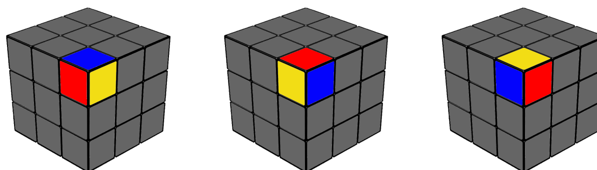
Ћошкови се, аналогно ивицама, означавају у облику XYZ где су X, Y и Z три стране које ћошак дели са целом коцком у сложеном облику. На пример ћошак који се налази у пресеку U, R и F стране обележава се са URF.

Међутим, сам положај коцкица није довољан да би се у потпуности одредило стање коцке. Наиме, ако се на коцку са слике 2.4 после последњег положаја (након потеза U и F) примени потез R добија се коцка са слике 2.5. Посматрана ивица се налази на истом положају као на првом делу слике 2.4, али су те две позиције коцке очиглено различите. Те две позиције разликују се по *оријентацији* те ивице.



Слика 2.5

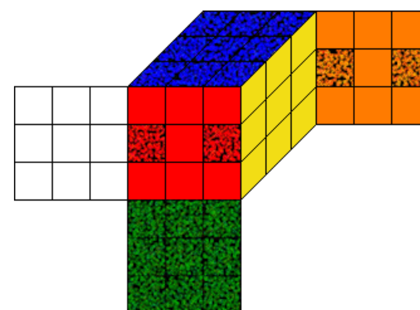
Аналогна ситуација је могућа и са ћошковима (слика 2.6), ћошак је на истом месту али другачије оријентисан.



Слика 2.6

Свака ивица има две могуће оријентације (јер има две стране), а сваки ћошак три могуће оријентације (има три стране). Зато се свакој од тих оријентација додељује број 0 или 1, у случају ивица, односно 0, 1 или 2 у случају ћошкова. То се постиже на следећи начин.

Целој коцки се додељују карактеристичне странице коцкица (осенчени делови на слици 2.7). На тај начин се и свакој коцкици одређује карактеристична страна као она страна коцкице која припада карактеристичом (осенченом) делу коцке. На пример, за ивицу UR (плаво-жута боја) ће карактеристична страна бити плава. За ћошак URF (плаво-жуто-црвени) ће такође карактерична страна бити плава.



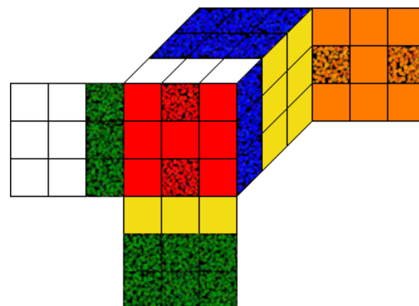
Слика 2.7

Сада се оријентација коцкица дефинише на следећи начин:

- Оријентација ивице је 0 ако карактеристична страна те ивице припада карактеристичом делу коцке. У супротном оријентација те ивице је 1.

- Оријентација ћошка је 0 ако каракретистична страна тог ћошка припада карактеристичној страни коцке. У супротном, оријентација тог ћошка је 1 (односно 2), ако је његова карактеристична страна окренута у смеру (односно супротно од смера) казаљке на сату у односу на страну ћошка који припада каракреристичној страни коцке.

Разјаснимо ово на примеру коцке где је примењен само F потез (слика 2.8). Ивица FL (бело-црвена) променила је свој положај и сада се њена осенчена (црвена) страна не поклапа ни са једном осенченом страном на слици 2.7 па је њена оријентација 1. Са друге стране, оријентација ивице UR (плаво-жута) није се променила и како њена осенчена (плава) страна припада карактеристичој страни коцке то је њена оријентација 0.



Слика 2.8

Ћошак UFL (бело-плаво-црвени) има своју карактеристичну (осенчену) страну плаву. Да би његова оријентација била 0, било би потребно да та плава страна буде окренута ка горе (да би припадала карактеристичој страни коцке тј. осенченом делу на слици 2.7). Ипак, на том месту се налази бела страна, а плава страна је окренута у смеру казаљке на сату у односу на њу па је оријентација тог ћошка једнака 1.

Пример ћошка са оријентацијом 2 би био URF (жуто-црвено-плави). Његова карактеристична страна је плава, а она је окренута супротно од казаљке на сату у односу на жуту страну (која припада карактеристичном делу целе коцке).

За ћошак са оријентацијом 0 пример може да буде UBR (плаво-наранџасто-жути) јер његова, плава, карактеристична страна припада осенченом делу целе коцке.

На слици 2.6 имао примере ћошка са оријентацијом редом 0, 1 и 2.

На овако дефинисан начин оријентације свих коцкица у сложеној коцки су 0. Сада излажемо три битне теореме.

Теорема 2.1. Збир оријентација свих ивица је увек паран.

Доказ. Потези на странама U, D, R и L уопште не утичу на оријентацију ивица, па самим тим ни на укупан збир. То исто важи и за потезе F2 и B2. Остаје да се испитају још и потези F, F' и B, B'. Међутим, они свакој од 4 ивица промене оријентацију на супротну, а како је број тих промена паран, то се укупна парност збира свих оријентација не мења.

Како је збир оријентација код сложене коцке 0, то се парност збира свих оријентација код сваке позиције коцке неће мењати, па ће бити дељив са 2. □

Теорема 2.2. Збир оријентација свих ћошкова је увек дељив са 3.

Доказ. Ниједан од потеза U и D стране не утиче на оријентацију ћошкова на њима. Краћом анализом може се закључити да ни потези R2, F2, L2 и B2 такође не утичу на оријентацију ћошкова, па тако ни на укупан збир. Остају потези страна R, F, L и B за 90° у неком смеру. Сваки од њих утиче на 4 ћошка, од којих ће за два (међусобно дијагонално постављена) од њих повећати оријентацију за 1 по модулу 3, а за друга два ће је смањити за 1 по модулу 3. Дакле, укупан збир по модулу 3 се не мења, а како сложена коцка има збир оријентације ћошкова 0, то је збир оријентације ћошова сваке позиције коцке дељив са 3.

□

Теорема 2.3. Пермутације свих коцкица су парне.

Доказ. Посматрајмо како један потез за 90° мења пермутацију коцке.

Користићемо лему да је парност пермутације којом се од једне пермутације елемената долази до друге увек иста.

Зато је довољно наћи само један начин размене коцкица и одредити његову парност да би се одредила парност пермутације након једног таквог потеза.

Такав потез утиче на 4 ивице и на 4 ћошка. Пошто ће се понашати независно једни од других тако ћемо их и посматрати. Нека су A, B, C и D неки ћошкови и нека су они у редоследу као на слици.

A B $\xrightarrow{+90^\circ}$ D A
D C C B

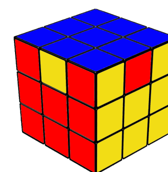
Такав резултат се може постићи тако што мењамо редом A и B, затим B и D и на крају B и C. Дакле, такав потез прави три размене, а како аналогну ситуацију имамо за ивице, то један такав потез прави 6 размена, тј. пермутација је парна.

Композиције парних пермутација су исто парне пермутације, па ће било која позиција коцке у ствари представљати парну пермутацију сложене што је и требало доказати. □

Последица теореме 2.1 је да је оријентација 12. ивице одређена оријентацијом осталих 11 јер ако су оријентације 11 ивица познате, онда ће оријентација 12. бити допуна тако да је укупан збир паран, што је једнозначно одређено.

Последица теорема 2.2 је да је оријентација 8. ћошка одређена оријентацијом осталих 7, јер тај последњи ћошак допуњава збир осталих по модулу 3 тако да он буде 0, што је такође једнозначно одређено.

Последица теореме 2.3 је искључивање случајева попут оног са слике 2.9 јер у њему имамо само једну размену (између ивица UR и UF), односно да нису могуће све пермутације коцкица.



Слика 2.9

Свака позиција коцке јединствено је одређена са 4 координате и то су **пермутација ивица, оријентација ивица, пермутација ћошкова и оријентација ћошкова**.

Како ивица има 12, то је број пермутација ивица једнак $12! = 479,001,600$. За сваку од њих постоје две оријентације, али како је оријентација последње одређена са осталима, то је број различитих оријентација ивица $2^{11} = 2,048$.

Ћошкова има 8, па је број њихових пермутација $8! = 40,320$. Сваки од њих има по 3 начина да се оријентише, али је опет, последњи одређен осталима, па је број оријентација ћошкова једнак $3^7 = 2,187$.

Дакле, свака позиција коцке може да се представи као уређена четворка целих бројева (x, y, z, t), где је $0 \leqslant x < 479001600$, $0 \leqslant y < 2048$, $0 \leqslant z < 40320$ и $0 \leqslant t < 2187$, ако се свака пермутација, односно оријентација нумерише на одговарајући начин.

Због потребе за чувањем великог броја позиција, овакав начин представљања позиција коцке је погодан за употребу у програму јер се коцка представља само као четири цела броја[3].

Још једно битно питање је колико има позиција коцке. Укупан број различитих пермутација је $12! \cdot 8!$. Међутим, од свих њих могуће су само парне пермутације према теореми 2.3, а како је број парних и непарних пермутација исти, то је број могућих пермутација једнак $\frac{12! \cdot 8!}{2}$.

За сваку од тих пермутација постоји $2^{11} \cdot 3^7$ различитих оријентација (јер су оријентације одређене са 11 ивица и 7 ћошкова), па је укупан број различитих позиција једнак $\frac{1}{2} \cdot 12! \cdot 8! \cdot 2^{11} \cdot 3^7 = 43,252,003,274,489,856,000$. Ред величине је 10^{19} и ово је главни разлог

[3] позиције у програму се ипак мало другачије представљају (одељак Представљање коцке), али је основна идеја иста

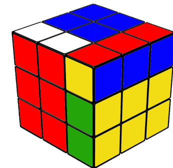
зашто је било потребно више од 30 година да би се закључило колико је то потеза потребно да би се генерисале све те позиције, а још један проблем како направити алгоритам за оптимално решавање.

2.4 Теорија група

Рубикова коцка представља једну групу и њено посматрање као такву довело је до најзначајнијих напретка ка проналаску оптималног решења.

Означимо сложену коцку са C, а са $*$ операцију композиције више потеза. На пример, онда би се примена U, па R потеза на неку коцку записивала као U$*$R. На исти начин би се примена низа потеза U$*$F$*$R, а затим примена неког другог низа потеза као што је B$*$D, записивала као (U$*$F$*$R)$*$(B$*$D).

На овај начин свака позиција коцке може да се напише као неки низ потеза примењен на сложену коцку C. На пример, коцки на слици 2.10 одговара ознака $C$$*F*$R јер је настала применом потеза F, па R на сложену коцку.



Слика 2.10

Нека је скуп свих позиција коцке које се представљају на овај начин G. Тај скуп је настао применом потеза из скупа $\{\mathrm{R}, \mathrm{F}, \mathrm{L}, \mathrm{B}, \mathrm{U}, \mathrm{D}\}$ и то се означава као $G = \langle \mathrm{R}, \mathrm{F}, \mathrm{L}, \mathrm{B}, \mathrm{U}, \mathrm{D} \rangle$.

Такав скуп G заједно са операцијом $*$ чини групу $(G, *)$. Покажимо то:

1. **затвореност** - Ако су M_1 и M_2 елементи скупа G, онда ће и $M_1 * M_2$ такође представљати низ неких потеза примењен на C, па је онда и $M_1 * M_2 \in G$

2. **асоцијативност** - Ово својство овде неће бити детаљно доказано, али се може проверити на једноставнијим примерима да примена прво потеза M_1 па онда потеза $M_2 * M_3$ исто утиче на коцку као примена прво потеза $M_1 * M_2$, па потеза M_3. Дакле, важи $M_1 * (M_2 * M_3) = (M_1 * M_2) * M_3$.

3. **неутрал** - Неутрал у скупу G је сложена коцка C, тј. коцка на коју није примењен ниједан потез.

4. **инверз** - Означимо са M^{-1} инверз потеза M. Он постоји за сваки потез, нпр. за F то је F', док је F2 сам себи инверз. Онда ће за сваки низ потеза $M_1 * M_2 * \cdots * M_n$ постојати инверз $M_n^{-1} * M_{n-1}^{-1} * \cdots * M_1^{-1}$, па ће за сваки елемент g скупа G да постоји инверз $g^{-1} \in G$ за који важи $g * g^{-1} = C$

Дакле, $(G, *)$ је група. Комутативност овде не важи, па ова група није Абелова.

Дефиниција 2.1. Ако подскуп H скупа G формира групу $(H, *)$ онда такву групу називамо подгрупом групе $(G, *)$.

Пример за подгрупу може бити група над скупом поциција генерисаним над потезима R2, F2, L2, B2, U2, D2. Скуп $G' = \langle \mathrm{R2}, \mathrm{F2}, \mathrm{L2}, \mathrm{B2}, \mathrm{U2}, \mathrm{D2} \rangle$ је очигледно подскуп скупа G, а аналогно се проверава да је и сам група.

Дефиниција 2.2. Ако је H подгрупа групе G, онда за неки елемент g скупа G дефинишемо десни косет[4] као скуп $Hg = \{a * g \mid a \in H\}$.

[4]постоји и леви косет који се дефинише аналогно

На примеру већ поменуте подгрупе $G' = \langle \mathrm{R2}, \mathrm{F2}, \mathrm{L2}, \mathrm{B2}, \mathrm{U2}, \mathrm{D2} \rangle$ групе G и неке позиције коцке $g \in G$ десни косет би био скуп свих позиција које се добијају применом низа потеза g на све могуће позиције из G', тј. скуп позиција $\{C * \mathrm{R2} * g, C * \mathrm{F2} * g, \dots\}$.

Како је десни косет по дефиницији везан за један елемент, то није једнозначно одређен скупом G, већ постоји скуп свих десних косета неког скупа G и означава се са $H \backslash G$. Значај косета ће овде, а и уопштено, бити што скуп свих десних косета дели посматрану групу на класе, односно подскупове са одређеним особинама.

Постоји још елемената теорије група који су везани за Рубиков коцку попут симетричних група које су део већине стандардних слагача[5], али су овде изложени само елементарни појмови неопходни за даљи рад.

[5]погледати одељак Закључак за више детаља о примени симетрија коцке

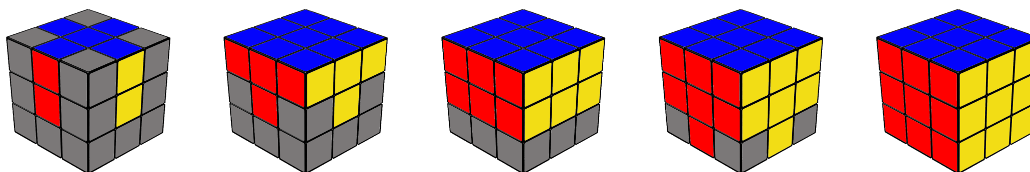
3 Алгоритам

У овом поглављу описан је концепт и имплеметација самог програма.

3.1 Основна идеја

Као што је већ речено главни помак у решавању направила су три алгоритма, Тислтвејтов, Косиембин и Корфов.

Генерална идеја је да се решавање Рубикове коцке подели на више фаза и да се свака од њих појединачно решава, док се не дође до сложене коцке. Прве идеју су, попут већине алгоритама којим се користе људи, те фазе делили по неким карактеристичним патернима или особинама по питању позиција коцкица. На пример, један од најпознатијих алгоритама за људско решавање прво слаже крст на једној страни, затим, ту страну, па онда наредни „слој” и на крају последњи слој, односно страну (слика 3.1).



Слика 3.1: Поједностављени приказ тзв. почетничог метода

Слагање једне фазе много је лакше од слагања целе коцке, односно, у случају рачунара, има мање позиција кроз које треба проћи. За рачунар код таквог начина поделе има неколико проблема. Први је што је број фаза генерално велики па чак и ако би сваку од њих решили оптимално, коначно решење би могло да испадне доста дуже него очекивано. Осим тога, таква подела је лакша само људима за слагање, неке од тих фаза су по броју случајева готово исто захтевне као и сама коцка.

3.1.1 Тислтвејтов алгоритам

Погодну поделу налази Тислтвејт 1981. године када коцку посматра као групу генерисану потезима L, R, F, B, U и D и сваку наредну фазу као подргупу претходне. Те групе означавао је редом G_0, G_1, G_2, G_3 и G_4 где група G_0 представља све могуће позиције Рубикове коцке, а група G_4 има само један елемент - сложену коцку тј. C.

Идеја је, као и код осталих алгоритама за поделу на фазе, да се почетна, произвољна, коцка доведе прво у подгрупу G_1, када се то уради онда се тражи начин да се доведе у G_2, затим у G_3 и, када се доведе у G_4, коцка ће бити сложена.

Групе је изабрао на следећи начин:

- $G_0 = \langle \mathrm{L, R, F, B, U, D} \rangle$

- $G_1 = \langle \mathrm{L}, \mathrm{R}, \mathrm{F}, \mathrm{B}, \mathrm{U2}, \mathrm{D2} \rangle$
- $G_2 = \langle \mathrm{L}, \mathrm{R}, \mathrm{F2}, \mathrm{B2}, \mathrm{U2}, \mathrm{D2} \rangle$
- $G_3 = \langle \mathrm{L2}, \mathrm{R2}, \mathrm{F2}, \mathrm{B2}, \mathrm{U2}, \mathrm{D2} \rangle$
- $G_4 = \{C\}$

Групе су биране по потезима којима су те позиције генерисане. На пример, у групи G_1 су све позиције које се могу добити потезима L, R, F, B, U2, D2, па ће решавање коцке која припада тој подгупи бити могуће употребом само тих потеза. Дакле, када се коцка доведе у G_1 за даље решавање неће бити потребни потези U, U', D, D' што значајно смањује број случајева кроз које треба проћи.

Теоретски, претрага се одвија на следећи начин. Нека је тренутна позиција коцке p у групи G_i. Претражује се прво скуп свих десних косета $G_{i+1} \backslash G_i$. Тај скуп дели скуп G_i на подскупове са карактеристичним особинама од којих је један једнак G_{i+1}. Проналази се којој од тих група припада полазна позиција p тј. проналази се десни косет облика $G_{i+1}g$ коме она припада. Онда позиција $p * g^{-1}$ припада G_{i+1} и узимама се за почетну вредност следеће итерације са $i+1$. Поступак се понавља док се не добије елемент скупа G_4 тј. сложена коцка.

На овај начин је број случајева које претражујемо у фази i једнак броју елемената скупа десних косета $G_{i+1} \backslash G_i$. Ти бројеви дати су у следећој табели.

Скуп косета	Број елемената
$G_1 \backslash G_0$	2,048
$G_2 \backslash G_1$	1,082,565
$G_3 \backslash G_2$	29,400
$G_4 \backslash G_3$	663,552

Највећи од њих је реда величине 10^6 што свакако није проблем за данашње рачунаре и то је управо разлог зашто је овај алгоритам направио први велики корак ка тражењу оптималног решења. Осим тога, највећи број потеза који захтева нека позиција коцке да се доведе у G_1 је 7, да се из G_1 доведе у G_2 је 10, из G_2 у G_3 је 13 и из G_3 до сложене коцке је 15. Тако да овај алгоритам у најгорем случају налази решења дужине 45 што није нимало лош резултат и свакао доста бољи од свих претходних, али знајући сада оптималан број, видимо да има простора за побољшање.

3.1.2 Косиембин алгоритам

Тислтвејтов алгоритам унапређује Херберт Косиемба 1992. године тако што задржава основну идеју али смањује број група на следећи начин:

- $G_0 = \langle \mathrm{R}, \mathrm{F}, \mathrm{L}, \mathrm{B}, \mathrm{U}, \mathrm{D} \rangle$
- $G_1 = \langle \mathrm{R2}, \mathrm{F2}, \mathrm{L2}, \mathrm{B2}, \mathrm{U}, \mathrm{D} \rangle$
- $G_2 = \{C\}$

Главна разлика је што овај алгоритам превазилаци проблем свог претходника јер мањи број група даје решења ближа оптималном. Друга разлика је што је због повећаног броја елемата скупа косета начин претраживања морао да се промени у односу на Тислтвејтов алгоритам где су се неопходне вредности чувале у табели.

Овај алгоритам је унапређиван кроз године и узимао разне друге идеје (највише по питању претраге), тако да је он данас један од најкоришћенијих алгоритама за слагање Рубикове коцке јер даје решења врло блиска оптималном (ако не и оптимална) за јако кратко време.

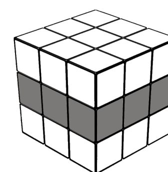
Претрага се у теорији одвија на исти начин као код Тислтвејтовог алгоритма, али је начин претраге другачији и састоји се од два дела. Довођење коцке у G_1 зваћемо прва фаза, а довођење коцке из G_1 у G_2 односно сложено стање, зваћемо друга фаза. Зато се овај алгоритам често и назива агоритам две фазе (енг. *two phase algorithm*)

Тражење одговарајућег десног косета почетног елемента одређене фазе не може лако да се претражи преко потеза који генеришу то стање и зато се уочавају правилности које важе код групе G_1. Нека је скуп потеза који генеришу G_1 означен са $M = \{\mathrm{R2}, \mathrm{F2}, \mathrm{L2}, \mathrm{B2}, \mathrm{U}, \mathrm{D}\}$, онда важи:

Теорема 3.1. Ако је оријентација свих ивица неке позиције Рубикове коцке једнака 0, онда потези скупа M не мењају њихову оријентацију.

Теорема 3.2. Ако је оријентација свих ћошкова неке позиције Рубикове коцке једнака 0, онда потези скупа M не мењају њихову оријентацију.

Скуп коцкица које се налазе у средњем слоју (слика 3.2) слободно преводимо као исечак (енг. *slice*). Постоје три различита исечка коцке UD, FB и RL и сваки од њих представља онај скуп коцкица које не припадају ниједној од одговарајућих страна. На пример, на слици 3.2 приказан је UD исечак јер га чине све коцкице које не припадају U и D страни.



Слика 3.2

Теорема 3.3. Потези скупа M не мењају скуп коцкица UD исечка.

Свака од ових теорема се може једноставно показати анализом потеза и зато је њихов доказ овде изостављен. Њихов значај је у наредној последици.

Последица. Свака коцка из групе G_1 има оријентацију свих ивица и ћошкова 0 и све ивице UD исечка су у њему.

Она следи из чињенице да сложена коцка има оријентације свих ивица и ћошкова 0, као и да су ивице UD исечка у UD исечку, а како потези скупа M не утичу на те три особине, онда ће их сви елементи скупа G_1 задржати.

Важи и обрнуто, да свака позиција коцке која има те три особине припада скупу G_1, али се тај доказ овде изоставља.

Дакле, коцка је у G_1 ако и само ако је оријентација свих њених ивица 0, оријентација свих њених ћошкова 0 и ако су све ивице UD исечка у њему.

Ово је основа алгоритма, односно превођење почетне коцке у стање са оријентисаним ивицама и ћошковима и одређеним ивицама на свом месту, а онда применом потеза који не нарушавају те већ успостављене особине, вршење претраге до сложене коцке.

3.2 Представљање коцке

Свака позиција је одређења са четири координате, тј. уређеном четворком (x, y, z, t), где су x, y, z и t редом пермутација ивица, оријентација ивица, пермутација ћошокв и оријентација ћошкова. Због специфичности скупа G_1 коцка се представља са 6 координата, и то су:

1. оријентација ивица
2. оријентација ћошкова
3. позиција ивица UD исечка

4. пермутација ивица UD исечка
5. пермутација ивица ван UD исечка
6. пермутација ћошкова

Прве три координате су специфичне за прву фазу и само се у њој и посматрају јер су током друге фазе оне 0, када се посматрају само последње три координате. Како је идеја да се коцка представи као уређена шесторка целих бројева потребно је да се свака од ових координата мапира на одговарајући интервал целих бројева.

Оријентације ивица представљају варијације са понављањем, и има их, $2^{11} = 2048$.

Оријентација ћошкова исто представаљају варијације са понављањем и има $3^7 = 2187$

Позиција ивица UD исечка је нова координата која означава на којим се све местима налазе четири ивице UD исечка од 12 могућих позиција за ивице. Оне представљају комбинације без понављања те четири ивице и има их $\binom{12}{4} = 495$.

Пермутација ивица UD исечка се јавља у другој фази када су UD ивице у свом исечку, па представља пермутацију без понављања 4 елемента, тј. има их $4! = 24$.

Пермутација ивица ван UD исечка се исто јавља у другој фази када се ивице које не припадају UD исечку налазе на преосталих 8 места за ивице, па такође представља пермутацију без понављања и има их $8! = 40320$

Пермутација ћошкова је пермутација без понављања и њих је $8! = 40320$.

Скуп свих десних косета $G_1 \backslash G_0$ дели скуп G на одређене подскупове који се разликују по прве три координате, док скуп $G_2 \backslash G_1$ дели G_1 на подскупове који се разликују по последње три координате. Зато је број њихових елемената следећи:

Скуп косета	Број елемената
$G_1 \backslash G_0$	$2^{11} \cdot 3^7 \cdot \binom{12}{4} = 2,217,093,120$
$G_2 \backslash G_1$	$4! \cdot 8! \cdot 8! = 39,016,857,600$

Маскималан број потеза потребан да се коцка доведе у G_1 је 12, а да се из G_1 доведе до G_2 је 18. Ово значи да у најгорем случају алгоритам даје решење дужине 30, што је већ унапређење у односу на претходника, али се такав резултат може још унапредити о чему ће бити речи касније.

Свака од наведених координата представља или варијацију или пермутацију или комбинацију и зато је неопходно направити функције које ће моћи да мапирају те три врсте пребројавања на одговарајуће интервале, као и њима инверзне функције које од датог броја из интервала проналазе њему одговарајуће пребројавање.

3.2.1 Мапирање варијација са понављањем

Сваки низ од k елемената који узима вредности $0, 1, \ldots, m$ може да се нумерише неким скупом бројева са m^k елемената. Како сваки такав низ представља један број између 0 и $m^k - 1$ (укључујући и њих) у бројевном систему са основом m, једна функција тражи вредности броја у систему са основом m, а друга претвара цели број на одговарајућем интервалу у број са основом m.

Ако је дат низ $a_0, a_1, \ldots, a_{k-1}$ такав да је $0 \leqslant a_i < m$ за $i = 0, 1, \ldots, k-1$, онда се такав низ мапира тако што му се додељује вредност x таква да је

$$x = a_0 m^{k-1} + a_1 m^{k-2} + \cdots + a_{k-2} m + a_{k-1} \tag{1}$$

Обрнуто, за неки цео број x за који важи $0 \leqslant x < m^k$ постоје јединствени бројеви a_i који задовољавају једнакост (1). То се постиже на следећи начин:

Алгоритам 1 Мапирање варијације

Улаз: Низ $(a_i)_{i=0}^{k-1}$, цео број m
Излаз: Цео број x

1: $x \leftarrow 0$
2: $d \leftarrow 1$
3: **за** $i \leftarrow k-1$ **до** 0 **ради**
4: $\quad x \leftarrow x + da_i$
5: $\quad d \leftarrow md$
6: **врати** x

Алгоритам 2 Тражење варијације

Улаз: Цели бројеви x, m, k
Излаз: Низ $(a_i)_{i=0}^{k-1}$

1: **за** $i \leftarrow k-1$ **до** 0 **ради**
2: $\quad a_i \leftarrow x \bmod m$
3: $\quad x \leftarrow \lfloor \frac{x}{3} \rfloor$
4: **врати** $(a_i)_{i=0}^{k-1}$

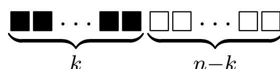
3.2.2 Мапирање комбинација без понављања

У овом случају дат је низ a од n елемената од којих је k њих означено као `true`, а остали као `false`. Позиције тих k елемената су редом $c_1, c_2, \ldots, c_k$ и нумеришу се с лева на десно почевши од 0.

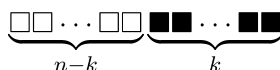
Мапирање се врши тако да свака могућа комбинација добије одговарaћи цео број од 0 до $\binom{n}{k} - 1$. То се постиже тако што се комбинацији са позицијама $c_1, c_2, \ldots, c_k$ додели вредност x која је једнака

$$x = \binom{c_k}{k} + \binom{c_{k-1}}{k-1} + \cdots + \binom{c_1}{1} \tag{2}$$

Положај испод узима вредност 0 (`true` вредности су означене црном бојом).



Док ситуација испод узима вредност $\binom{n}{k} - 1$.



Поступак добијања комбинације од датог целог броја x за који важи $0 \leqslant x < \binom{n}{k}$ је могућ зато што су бројеви c_i који задовољавају (2) јединствени, а састоји се у томе да се нађе највећа могућа вредност $\binom{p}{k}$ која није већа од x. Такво p ће бити c_k. Онда се x смањи за ту пронађену вредност и тражи маскималан број облика $\binom{p}{k-1}$ који није већи од нове вредности x. Тако се добија c_{k-1}. Поступак се наставља док се не добију сви елементи низа c. Алгортми који се користе су следећи:

Алгоритам 3 Мапирање комбинације

Улаз: Низ $(a_i)_{i=0}^{n-1}$
Излаз: Цео број x

1: $k \leftarrow 0$
2: **за** $i \leftarrow 0$ **до** $n-1$ **ради**
3: $\quad$ **ако** $a_i = \text{true}$ **онда**
4: $\quad\quad k \leftarrow k+1$
5: $\quad\quad c_k \leftarrow i$
6: $x \leftarrow 0$
7: **за** $i \leftarrow 1$ **до** k **ради**
8: $\quad x \leftarrow x + \binom{c_i}{i}$
9: **врати** x

Алгоритам 4 Тражење комбинације

Улаз: Цели бројеви x, k
Излаз: Низ $(a_i)_{i=0}^{n-1}$

1: **за све** a_i **ради**
2: $\quad a_i \leftarrow \text{false}$
3: **за** $i \leftarrow k$ **до** 1 **ради**
4: $\quad p \leftarrow i-1$
5: $\quad$ **док** $\binom{p+1}{i} \leqslant x$ **ради**
6: $\quad\quad p \leftarrow p+1$
7: $\quad a_p \leftarrow \text{true}$
8: $\quad x \leftarrow x - \binom{p}{i}$
9: **врати** $(a_i)_{i=0}^{n-1}$

3.2.3 Мапирање пермутација без понављања

Дат је низ a са n елемената и сваки од њих узима вредност из скупа $\{0, 1, \ldots, n-1\}$ али тако да не постоје два једнака елемента. сваком таквом низу може се једнозначно придружити један цео број између 0 и $n! - 1$ на следећи начин. Ако се број i налази на позицији k, односно ако је $a_k = i$, дефинишемо коефицијент уз i као цео број c_i који је једнак броју елемената низа чија је позиција већа од k, а вредност мања од i. Онда таквој пермутацији додељујемо број x такав да је

$$x = 0! \cdot c_i + 1! \cdot c_1 + \cdots + (n-1)! \cdot c_{n-1} \tag{3}$$

На примеру испод

i	0	1	2	3	4	5	6	7
a_i	3	5	0	1	4	7	2	6

Број 0 се налази на позицији 3, а са његове десне стране су сви бројеви већи од њега и има их 5, па је $c_0 = 5$. Добија се да је $c_1 = 4, c_2 = 1, c_3 = 4, c_4 = 2, c_5 = 2, c_6 = 0$ и $c_7 = 0$. Ова пермутација ће бити нумерисана као

$$5 \cdot 0! + 4 \cdot 1! + 1 \cdot 2! + 4 \cdot 3! + 2 \cdot 4! + 2 \cdot 5! + 0 \cdot 6! + 0 \cdot 7! = 323$$

Обрнути процес је опет могућ јер на овај начин број x за који је $0 \leqslant x < n!$ постоје јединствени бројеви c_i за које важи (3). Алгоритми се реализују на следећи начин:

Алгоритам 5 Мапирање пермутација

Улаз: Низ $(a_i)_{i=0}^{n-1}$
Излаз: Цео број x

1: $f_0 \leftarrow 0$
2: **за** $i \leftarrow 1$ **до** k **ради**
3: $\quad f_i \leftarrow i f_{i-1}$
4: **за** $i \leftarrow 0$ **до** $k-1$ **ради**
5: $\quad c_i \leftarrow 0$
6: $\quad j \leftarrow k-1$
7: $\quad$**док** $a_j \neq i$ **ради**
8: $\quad\quad$**ако** $a_j < i$ **онда**
9: $\quad\quad\quad c_i \leftarrow c_i + 1$
10: $\quad\quad j \leftarrow j-1$
11: $x \leftarrow 0$
12: **за** $i \leftarrow 0$ **до** $k-1$ **ради**
13: $\quad x \leftarrow x + c_i f_i$
14: **врати** x

Алгоритам 6 Тражење пермутације

Улаз: Цели бројеви x, n
Излаз: Низ $(a_i)_{i=0}^{n-1}$

1: $f_0, a_0 \leftarrow 0$
2: **за** $i \leftarrow 0$ **до** $n-1$ **ради**
3: $\quad f_i = i f_{i-1}$
4: $\quad a_i \leftarrow 0$
5: **за** $i \leftarrow n-1$ **до** 0 **ради**
6: $\quad c_i \leftarrow \lfloor \frac{x}{f_i} \rfloor$
7: $\quad x \leftarrow x \bmod f_i$
8: **за** $i \leftarrow n-1$ **до** 0 **ради**
9: $\quad t \leftarrow 0$
10: $\quad j \leftarrow n$
11: $\quad$**док** $t \leqslant c_i$ **ради**
12: $\quad\quad j \leftarrow j-1$
13: $\quad\quad$**ако** $a_j < i$ **онда**
14: $\quad\quad\quad t \leftarrow t+1$
15: $\quad a_j \leftarrow i$
16: **врати** $(a_i)_{i=0}^{n-1}$

3.2.4 Табеле потеза

Претходно објашњено представљање коцке погодно је за чување, али није толико подесно за директно примењивање потеза. Та мана се решава табелама потеза (енг. *moving tables*). Табеле постоје за сваку координату и таквог су облика да i-ти ред одређене табеле представља i-ту позицију те координате. Колоне представљају 18 различитих потеза, тако

да се у пољу (i, j) налази вредност одређене координате када се потез j примени на коцку са том координатом која је једнака i.

Генерисање табела се састоји у следећем. Нека је S скуп свих могућих вредности неке координате, нпр. за оријентаицју ћошова тај скуп је $\{0, 1, \ldots, 2186\}$. Онда се за сваки елемент тог скупа рачуна којој пермутацији, варијацији или комбинацији одговара (у зависности од координате која се посматра) и то раде алгоритми 2, 4 и 6. На тај начин се доста лакше примењују потези, што се и ради и добија се нека нова врста пребројавања. На крају се добијена варијација, комбинација или пермутација мапира назад уз помоћ алгоритама 1, 3 и 5.

Идејно прављење табела потеза није захтевно, тако да је овде представљена само уопштена идеја која би се могла овако описати.

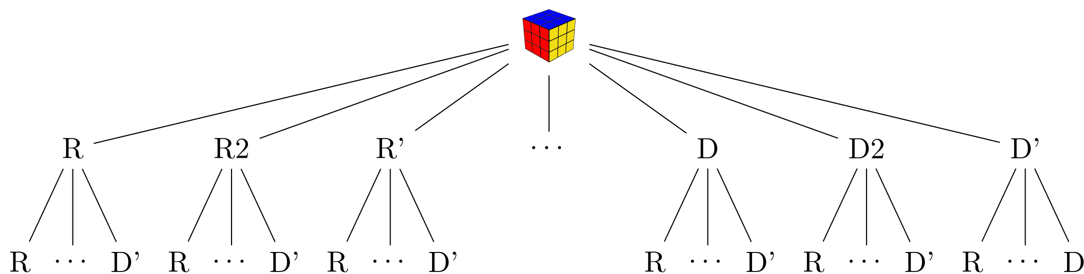
Алгоритам 7 Креирање табеле потеза

```
1: за све i ∈ S ради
2:     a_n ← TRAZI_PREBROJAVANJE(i)
3:     за све m у скупу потеза ради
4:         t_n ← m(a_n)
5:         i' ← MAPIRAJ(t_n)
6:         запиши i' у табелу
```

Највећа од табела узима око 4MB простора, док све укупно узимају око 7MB што не представља никакав проблем за савремене рачунаре.

3.3 Претрага

Овако представљена коцка са свим својим позицијама одређује један граф са чворовима облика (u, v, w, x, y, z) где су u, v, w, x, y и z одговарајуће координате поцизије. Свака позиција повезана је са тачно 18 других, јер постоји 18 различитих потеза који се могу применити. Свака веза представљена је у табелама потеза, тако да је време приступа суседним чворовима константно.



Слика 3.3

Тако посматрано, претраживање прве и друге фазе биће заправо претрага графа. Основни алгоритми за такву претрагу су претрага у дубину и претрага у ширину који се користе и који ће бити основа са мало напреднији алгоритам.

3.3.1 Претрага у дубину

Претрага у дубину (позната као DFS, од енглеског *Depth First Search*) је агортитам потпуне претраге графа у коме се полази од неког чвора и обилази се свака грана колико код може и онда се враћа на непосећене чворове.

Принцип извршавања је такав да се прво посматрају сви суседни чворови почетног чвора и узима први који није већ посећен. Онда се алгоритам рекурзивно позива за тај нови чвор. Када се дође до тренутка да нема више непосећених суседа тренутног чвора, због рекурзивног позивања долази до враћања на остале. Може се имплементирати на следећи начин:

Алгоритам 8 Претрага у дубину

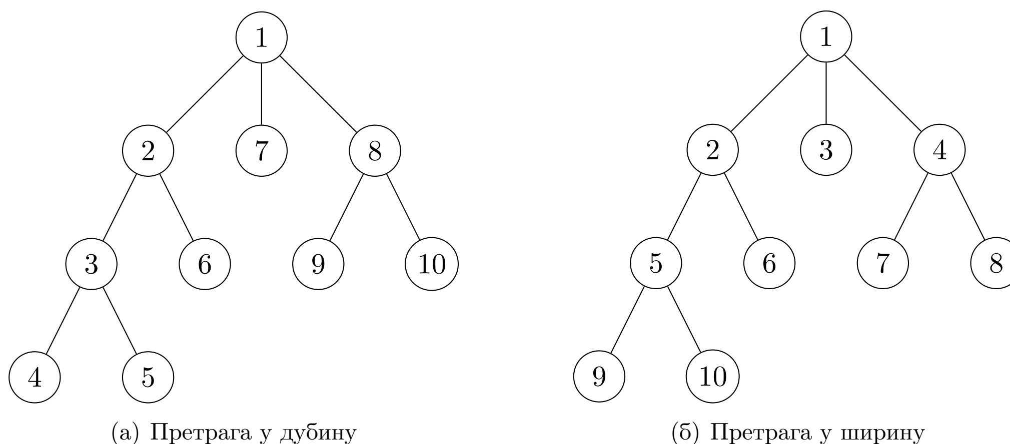
Улаз: Граф G, полазни чвор v

Излаз: Сви чворови доступни од чвора v, означени као посећени

1: означи све чворове као непосећене
2: **функција** DFS(G, v)
3: означи v као посећен
4: **за све** $w \in G$ који су суседни са v **ради**
5: **ако** w није посећен **онда**
6: DFS(G, w)

Пример такве претраге дат је на слици 3.4(а). чворови се обилазе редом којим су нумерисани и тако се прво пролази кроз дубину одакле назив и потиче.

Временска сложеност ове претраге је $O(|V| + |E|)$, а меморијска сложеност је $O(|V|)$, где је $|V|$ број свих чворова, а $|E|$ укупан број веза.



(а) Претрага у дубину

(б) Претрага у ширину

Слика 3.4

3.3.2 Претрага у ширину

Претрага у ширину (позната као BFS, од енглеског *Breadth First Search*) је алгоритам потпуне претраге графа који претражује чворове по њиховој удаљености од почетног. Прво се прође почетни чвор, затим они који су му директни суседи, па суседи тих суседа итд.

На овај начин алгоритам претражује по ширини ако би се графички представило и отуда потиче назив. Пример такве претраге дат је на слици 3.4(б) где су чворови нумерисани редом којим се обилзе.

За имплементацију се користи помоћни ред због особине да први који се убаци у њега, први и излази из њега (FIFO принцип, *First In First Out*) и у свакој итерацији се њему додаје низ чворова на новој дубини у графу, а они из претходне избацују.

Временска сложеност је $O(|V| + |E|)$, где су $|V|$ и $|E|$ редом број чворова и број ивица графа, а меморијска сложеност је $O(|V|)$.

Алгоритам 9 Претрага у ширину

Улаз: Граф G, полазни чвор v
Излаз: Сви чворови доступни од чвора v

```
1: убаци чвор v у ред q
2: док q није празан ради
3:     u ← први елемент у q
4:     избаци први елемент из q
5:     за све w ∈ G који су суседни са u ради
6:         додај w на крај реда q
```

Овако имплементиране ове две претраге дају исте временске и меморијске сложености па избор међу њима зависи од потребе проблема. Даље ће се користити овако имплементирана претрага у ширину док ће се претрага у дубину користити мало измењена о чему ће бити речи касније.

3.3.3 IDA* и Корфов алгоритам

Ричард Корф је 1997. године представио алгоритам за оптимално слагање Рубикове коцке тако што је свој већ развијени алгоритам IDA* применио на Рубикову коцку. Такав програм није могао оптимално да реши сваку позицију у неком релативно кратком времену, али теоријски увек даје најкраће решење [2]. Примену налази у Косиембином алгоритму у току претраге фаза, јер је број позиција доста мањи и овај алгоритам даје одличне резултате.

IDA* (енг. *Iterative Deepening A**) је верзија A* алгоритма за тражење најкраћег пута од полазног до задатог чвора неког графа. Функционише по принципу претраге у дубину са итеративним продубљивањем и користи хеуристичку функцију за процену и одбацивање случајева.

Претрага у дубину итеративним продубљивањем је претрага у којој се задаје максимална дубина до које се иде и онда се примењује претрага у дубину у графу ограниченом том дубином. Ако је тражени чвор пронађен претрага се зауставља и враћа резултат, у супротном се максимална дубина повећава за 1 и примењује се поново претрага у дубину на граф са новим ограничењем. При томе се не користи помоћни низ који бележи да ли су неки чворови посећени или не, тако да се може десити да се више пута уђе у исти чвор. Ово је једна неизбежна мана јер се итеративно продубљивање управо и користи у ситуацијама када је број чворова превише велики да би се чували у меморији и на овај начин се меморијска сложеност значајно смањује на $O(d)$, где је d максимална дубина до које се иде у графу.

Хеуристичка функција је функција процене удаљености неког чвора у графу од циљаног чвора. Важно је да хеуристича функција никада не прецењује тачно растојање да би алгоритам радио и означава се са $h(v)$ за неки чвор v.

Сада се принцип рада IDA* алгоритма састоји у следећем. У једној итерацији се граф претражује до одређене дубине m, у стилу претраге у дубину. За сваки чвор v се рачуна вредност $f(v) = g(v) + h(v)$ где је $g(v)$ растојање чвора v од полазног, а $h(v)$ хеуристичка функција која процењује растојање од чвора v до циљног чвора. Ако је $f(v) > m$ онда се претрага за тај чвор прекида јер се за њега не може наћи решење дужине не веће m. Уколико се дође до циљног чвора враћа се резултат и прекида даљи ради, а у супротном се максимална дубина m повећава за 1 и процес понавља док се не нађе тражени чвор. Решење ће се сигурно пронаћи ако важи већ поменути услов да хеуристичка функција

никада не прецењује право растојање тј. да је $h(v) \leqslant h^*(v)$ где је h^* право (тј. најкраће растојање) до циљаног чвора.

Меморијска сложеност је $O(d)$ где је d највећа дубина графа, док временска сложеност зависи од хеуристичке функције h. Имплементација се реализује на следећи начин:

Алгоритам 10 IDA*

Улаз: Граф G, полазни чвор v, циљани чвор c
Излаз: Најкраћи пут од v до c

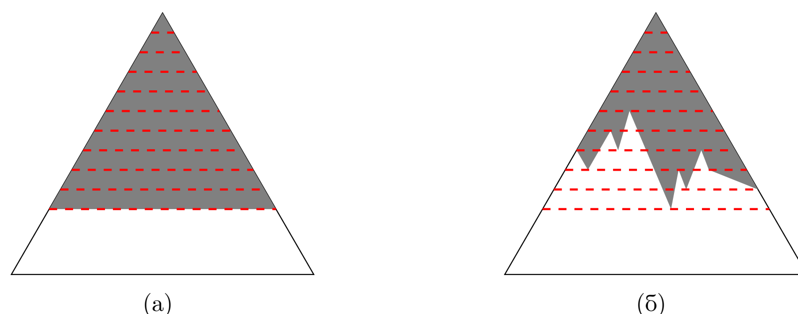
```
 1: s ← false                         ▷ параметар који говори да ли је решење пронађено
 2: празан стек p                     ▷ овде ће се чувати коначно решење

 3: функција DFS(G, u, c, t, m)       ▷ t је тренутна дубина
 4:     ако t = m онда
 5:         ако u = c онда
 6:             s ← true
 7:     иначе
 8:         ако t + h(u) ⩽ m онда
 9:             за све w ∈ G који су суседни са u ради
10:                 ако s = false онда
11:                     додај w у стек p
12:                     DFS(G, w, c, t + 1, m)
13:                     ако s = false онда
14:                         избаци w из стека p

15: d ← h(v)
16: док s = false ради
17:     DFS(G, v, c, 0, d)
18:     ако s = true онда
19:         врати p
20:     иначе
21:         d ← d + 1
```

Разјаснимо ово на примеру коцке. Нека је функција h процена за најмањи број потеза који су потребни неком стању да се сложе само ћошкови тј. да се њихова оријентација и пермутација доведе на 0, а остале координате могу бити било шта. Таква функција може да се израчуна јер је број случајева са само издвојеним ћошковима значајно мањи од случајева када се поматра цела коцка.

Ако се тренутно претражује граф до дубине 10, односно, нема решења краћих од 10 и дође се до стања v које је на дубини 6 тј. $g(v) = 6$ и потребно му је 7 потеза за слагање ћошкова тј. $h(v) = 7$, онда је за слагање целе коцке из тог стања неопходно бар $6 + 7 = 13$ потеза. Како се испитује до дубине 10, то се та грана одбацује и тражи се даље.

Ово представља Корфов алгоритам и визуелно се може приказати као на слици 3.5. Троуглови приказују граф који се грана. На врху се налази почетно стање v и свака испрекидана линија представља следећу дубину тј. гранања стања из претходне дубине, а сивом бојом је означен скуп чворова који се обилази.

На слици 3.5(а) је приказана ситуација када би се испитивали сви могући случајеви до неке максималне дубине, а на слици 3.5(б) када би се применио алгоритам IDA*. У другом случају број посећених чворова је знатно мањи јер долази до одсецања неких грана хеуристичком функцијом и решење се проналази доста брже.



(а) (б)

Слика 3.5

3.3.4 Табеле одсецања

Да би се омогућило коришћење IDA* алгоритма неопходно је да се нађе хеуристичка функција h. Како се решавање дели на две фазе, то се користе и две претраге са посебним функцијама процене. Да би те претраге биле што ефикасније потребно је да h буде што ближа тачној најкраћој удаљености и зато рачунање такве функције није могуће у току извршавања програма већ се вредности израчунају раније и сачувају у меморији у табелама одсецања (енг. *pruning tables*). Дакле, табеле одсецања су, попут табела потеза, фајлови које садрже вредности функције h за различите позиције коцке.

Прва фаза

У првој фази посматрају се прве три координате, односно оријентација ивица, оријентација ћошкова и позиција ивица UD исечка. Укупан број позиција је зато реда величине $2 \cdot 10^9$ за шта је могуће сачувати све вредности у меморији, па ће се у табелама одсецања прве фазе заправо налазити тачне вредности, а не приближне процене дужине решења ($h \equiv h^*$).

Оне се генеришу у стилу претраге у ширину тако што се крене од позиције која је већ у подгрупи G_1 тј. позиције чије су све три координате једнаке 0. Принцип је као у алгоритму 9 коме је почетни чвор $v = (0, 0, 0)$ са додатком да се за на свакој новој дубини поставља да је вредност h за један већа од претходне дубине. На пример, на сликама 3.3 и 3.5 на врху графа односно на почетном чору поставља се да је $h(v) = 0$. Онда се за све његове суседе u поставља да је $h(u) = h(v) + 1 = 1$, затим се за све суседе чворова за које је вредност h једнака 1, а није им додељена нека вредност, поставља да је вредност за њих 2 итд.

Испоставља се да је највише потребно 12 потеза да се нека позиција доведе у фазу 2. Како је временска сложеност претраге у ширину $O(|V| + |E|)$, а у овом случају је $|E| = \frac{18 \cdot |V|}{2} = 9|V|$ то ће временска сложености бити $O(|V|)$. Време које је потребно да се ово генерише се слаже уз ову рачуницу и за генерисање табела одсецања прве фазе било је потребно око 20 минута.

Међутим, проблем је меморијска сложеност која је исто $O(|V|)$, а $|V|$ је реда величине 10^9 па је потребно око 7GB меморије при њиховом креирању (јер стек p садржи елементе дужине од 4 бајта). То се превазилази при каснијем коришћењу јер се за сваку позицију чува број потеза по модулу 3[1]. На овај начин се за сваку инстанцу табеле користе 2 бита за чување па је потребна меморија при раду око 500MB.

[1] Ово је могуће зато што знамо тачан број потеза за све позиције и онда је довољно да се при претрази прве фазе нађе који од суседних чворова има за 1 мању вредност h по модулу 3, детаљи у одељку Комплетирање

Друга фаза

У другој фази прве три координате коцке су 0, па се посматрају само последње три, односно пермутација ивица UD исечка, пермутација исечка ван UD исечка и пермутација ћошкова. Разлика је и што је број веза другачији јер неки од 18 потеза мењају координате фазе 1, па их је број потеза који се посматрају у другој фази 10.

Број позиција је реда величине 10^{10} што је превише за чување (чак и ако би се чувало по модулу 3) па овде функција h даје процену за најмањи број потребних потеза h^* тако да за све чворове v важи $h(v) \leqslant h^*(v)$.

Ово се постиже на сличан начин као у примеру за Корфов алгоритам. Нека су h^*_{12}, h^*_{23} и h^*_{31} редом функције које одређују минималан број потеза за довођење коцке у позицију са првом и другом координатом 0, другом и трећом координатом 0 и трећом и првом координатом 0 (мисли се на прве три координате друге фазе). На пример, $h^*_{12}(v)$ говори колико је најмање потеза потребно да се позиција v доведе до стања у коме су пермутација ивица UD исечка и пермутација ивица ван UD исечка једнаке 0. Јасно је да су $h^*_{12}(v), h^*_{23}(v), h^*_{31}(v)$ доња ограничења за број потеза потребан да се позиција v доведе у стање са све три координате 0, тј. сложено стање, па се може узети да је

$$h(v) = \max\{h^*_{12}(v), h^*_{23}(v), h^*_{31}(v)\}.$$

Дакле, у другој фази имамо три табеле, а за хетуристичку функцију узимамо највећу од њих. Испод је дата расподела броја позиција међу њима.

Табела	Број елемената
h^*_{12}	$4! \cdot 8! = 967,680$
h^*_{23}	$8! \cdot 8! = 1,625,702,400$
h^*_{31}	$8! \cdot 4! = 967,680$

Генерисање ових табела се постиже на исти начин као и у првој фази. Користи се алгоритам 9 тј. претрага у ширину тако што се чворовима на свакој новој дубини се повећава вредност функције. Временска сложеност је $O(|V| + |E|) = O(|V| + \frac{10\cdot|V|}{2}) = O(6|V|) = O(|V|)$. Време потребно за највећу од тих табела h^*_{23} је око 15 минута док је за остале потребно мање од секунде.

Овде такође има већег коришћења меморије јер је меморијска сложеност исто $O(|V|)$ па програм узима око 6GB у току генерисања табеле h^*_{23}. Како се овде не чувају тачна растојања, у табели не могу да се користе вредности по модулу 3, па је меморија потребна за табеле одсецања друге фазе при њиховом коришћењу 1.5GB.

3.4 Комплетирање

До сада су већ описани сви главни делови програма и остаје још да се они обједине у целину.

На самом почетку, уколико не постоје, неопходно је да се генеришу табеле потеза и табеле одсецања прве и друге фазе. Када се једном генеришу биће сачуване у посебном фолдеру из кога касније постоји опција да се учитају. Затим се уноси почетно стање коцке након чега се позива функција која тражи решење што се одвија на следећи начин.

Прво се позива функција претраге прве фазе. Како имамо тачан потребан број потеза за сваку позицију у првој фази то је ова претрага доста брза сама по себи и одвија се тако што се посматрају сви суседи почетне позиције (чвора) и узима онај чији је број потеза потребан за довођење у фазу 2 мањи од број потеза потребног за почетно стање. Ово је могуће зато што се применом потеза на неку позицију добијају само оне позиције чији је број потеза потребан за довођење у другу фазу или за један већи, или за један мањи,

или једнак броју потеза за ту позицију. Прецизније речено, за сваку позицију v и сваки потез M важи $|h^*(v) - h^*(M(v))| \leqslant 1$. Ово је и разлог зашто се у првој фази вредности h^* чувају само по модулу 3, јер је то довољна информација за закључивање који потез води ка решењу. То се реализује на следећи начин.

Алгоритам 11 Претрага прве фазе

Улаз: Почетна позиција v

Излаз: Оптимално решење које доводи v у G_1

1: празан стек p ▷ овде ће се чувати решење
2: $u \leftarrow v$
3: **док** $u \neq (0, 0, 0)$ **ради**
4: **за све** m у скупу потеза **ради**
5: $w \leftarrow m(v)$
6: **ако** $h^*(u) - 1 \equiv h^*(w) \pmod 3$ **онда**
7: додај m у стек p
8: $u \leftarrow w$
9: **прекини**
10: **врати** p

Када се коцка доведе у подгрупу G_1 започиње наредна претрага, односно друга фаза. Како у другој фази немамо тачне вредности за потребну дужину броја потеза за слагање, користимо алгоритам IDA*, односно Корфов алгоритам на подрупи G_1. Прво је потребно да се примени низ потеза p на координате друге фазе јер нису биле учитане током прве фазе и позиција са те три израчунате координате се узима за почетни чвор v у алгоритму 10. Како се свака позиција може довести из групе G_1 до сложене коцке у највише 18 потеза, то је максимална дубина до које би претрага могла да се одвија 18. Фактор гранања сваког чвора је 10 (јер се у другој фази посматра само 10 потеза), па је број чворова који треба обићи, ако би се применила само претрага у дубину, могао да буде до 10^{18}. Овај број је превелики за израчунавање на било ком рачунару и овде се види значај претраге са функцијом процене јер се, уз помоћ њега, свакој позицији из G_1 налази решење за време које је реда величине 0.1 ms. Као што је речено временска сложеност алгоритма IDA* зависи од хеуристичке функције h и испоставља се да она у другој фази даје доста добре процене.

У оваквој имплементацији алгоритам решава оптимално прву, а затим другу фазу и враћа унију та два решења. Међутим, може се догодити да за неку позицију којој оптимално решење прве фазе има 9 потеза, а друге 15 (укупно 24 потеза) постоји неоптимално решење прве фазе од 11 потеза за које је потребно 7 потеза у другој фази, што даје свеукупно краће решење од 18 потеза. Како је програм у стању да сложи више од 1000 позиција у секунди, целокупној претрази може се додати унапређење у виду додавања произвољног низа потеза на почетну позицију у циљу добијања неоптималног решења прве фазе. Прави се функција SLOZI која слаже произвољну позицију коцке оптимално по обе фазе. Она се примењује на почетку позицију и одређује дужина тог првог решења.

Затим се користи функција која прави варијације са понављањем скупа потеза одређене дужине које ће се примењивати на почетну позицију и свака од тих се убацује у функцију SLOZI. Креће се од дужине 1, и повећава се кроз сваку итерацију. Због овога се у претрази друге фазе додаје још један параметар који представља дужину решења и ако се пронађе решење краће од те дужине оно се поставај као ново најбоље решење и његова дужина поставља као нова дужина решења. Програм се овако извршава док се не пређе неко задато време извршавања t. То се постиже као у алгоритму 12.

Алгоритам 12 Унапређена претрага

Улаз: Почетна позиција v, време извршавања t

Излаз: Најкраће пронађено решење које доводи v у сложено стање за време t

```
1:  функција PROIZVOLJNI(v, d′, d, p, t, r)          ▷ d′ је тренутна дужина
2:      ако d′ = d онда
3:          v′ ← r(v)                               ▷ r је низ потеза који се додаје
4:          p′ ← SLOZI(v′)
5:          ако d+ дужина p′ < дужина p онда
6:              p ← споји r са p′
7:      иначе
8:          за све m у скупу потеза ради
9:              ако време извршавања < t онда
10:                 додај m у стек r
11:                 PROIZVOLJNI(v, d′ + 1, d, p, t, r)
12:                 избаци m из стека r

13: p ← SLOZI(v)                                    ▷ p је прво пронађено решење
14: d ← 1
15: док време извршавања < t ради
16:     PROIZVOLJNI(v, 0, d, p, t, r)
17:     d ← d + 1
18: врати p
```

Како се у току извршавања пролази кроз велики број случајева, посебно при генерисању табела, за писање је изабран C++ због брзине коју нуди и целокупан код се може наћи на GitHub-у на адреси `https://github.com/stemil01/two-phase-solver`.

3.5 Резултати

Резултати су мерени тако што се генерише низ од произвољних 100 потеза и примени на сложену коцку, и онда таква позиција убацује у програм који враћа решење. Генерисано је 100 таквих позиција и рачунато је накраће и најдуже пронађено решење као и просечан број потеза потребан за решавање генерисаних позиција. Сви тестови су рађени на истих 100 примера.

Примена само претраге која обе фазе оптимално решава без додатне оптимизације даје следеће резултате:

Најкраће решење	19
Најдуже решење	26
Просечна дужина	22.9

Како свака коцка може да захтева до 20 потеза резултати без икакве оптимизације нису уопште лоши. Међутим, применом унапређене претраге се добијају значајно бољи резултати. Ову су резултати у зависности од времена t које се зада да рад.

t	0.5 s	1 s	2 s	5 s	10 s
Најкраће решење	17	17	17	17	17
Најдуже решење	22	21	21	21	21
Просечна дужина	20.17	20.09	19.96	19.73	19.68

Испоставља се са додатне пола секунде добијају резултати са новим најдужим решењем краћим од првобитног просечног, док се већ за време од 2 s просечна дужина решења спушта испод 20 потеза. Примећујемо још и да се повећањем времена након 2 s резултати не разликују значајно, па се алгоритам у пракси најбоље показује са временом $t = 2\,\mathrm{s}$. На поновљеним тестовима за различитим уносом за 2 s увек су се добијали резултати са просеком мањим од 20 потеза.

4 Закључак

Овај рад се не бави неком научном облашћу или дисциплином већ је његова идеја да се кроз један мало обимнији пројекат повежу и покажу неки начини којима се проблем за који се не зна опште решење (односно, стратегија за решавање) и за који нема довољно рачунарске снаге ипак врло ефикасно реши свођењем на једноставније подпроблеме и примењивањем неких погодних метода на њих. Још једна интересантна ствар везана за овај пројекат је што се кроз њега може видети паралалелно развој људког размишљања са развојем и напретком рачунара јер су све ове идеје настале на рачунарима са неколико MB радне меморије и фреквенције процесора реда величине MHz. Програмери су морали да осмисле начине како да уз такве ограничене алате реше проблеме и одговор је, у овом, као и у многим случајевима, давала математика. У последње време развили су се и програми коју слажу Рубикову коцку уз помоћ појачаног учења [3], међутим, још увек нису на истом нивоу као описани алгоритам у овом раду.

На овом пројекту има доста места за напредак, и ако су коначни резултати јако добри. Пре свега по питању меморије, јер програм захтева око 2GB радне меморије за разлику од већине апликација за слагање које захтевају много мање. То се може постићи коришћењем симетричних група коцке којима се позиције деле на одређен број еквивалентних класа по броју потеза потребног за слагање чиме се број инстанци у табелама одсецања значајно смањује. Тиме се не постижу велика побољшања брзине налажење решења и резултати би били врло слични, али би учитавање програма било доста брже и мање захтевно што би омогућило његово покретање на већем броју различитих машина. Разлог зашто је то изостављено у раду је што симетричне групе доста усложњавају програм, а не доводе до већег побољшања резултата. Осим тога, унос позиција коцке није толико практичан, па би се ту могао додати унос путем камере (наравно, био би потребан одговарајући програм за обраду слике) чиме би се овај пројекат комплетирао и био спреман за покретање на већини данашњих уређаја.

Литература

[1] T. Rokicki, H. Kociemba, D. Morley, and J. Dethridge, *God's Number is 20*. https://cube20.org/.

[2] R. E. Korf, *Finding optimal solutions to Rubik's cube using pattern databases*, 1997.

[3] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi, *Solving the Rubik's Cube Without HumanKnowledge*, 2018. https://arxiv.org/pdf/1805.07470.pdf.

[4] H. Kociemba, *Cube explorer*. http://kociemba.org/cube.htm.

[5] L. T. Hoang, *Optimally Solving a Rubik's Cube Using Vision and Robotics*, 2015. https://www.doc.ic.ac.uk/teaching/distinguished-projects/2015/l.hoang.pdf.

[6] J. Chen, *Group Theory and the Rubik's Cube*. http://people.math.harvard.edu/~jjchen/docs/Group%20Theory%20and%20the%20Rubik's%20Cube.pdf.

[7] H. Kaur, *Algorithms for solving the Rubik's cube*, 2015. http://www.diva-portal.org/smash/get/diva2:816583/FULLTEXT01.pdf.

[8] https://en.wikipedia.org/wiki/Rubik's_Cube.

[9] https://en.wikipedia.org/wiki/Optimal_solutions_for_Rubik%27s_Cube.

[10] https://en.wikipedia.org/wiki/God's_algorithm.