Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Rubik kocka kirakásának szimulációja szakterületi nyelv segítségével

Diplomaterv

Készítette
Taska Tamás

Konzulens
Somogyi Ferenc Attila

2023. május 24.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Taska Tamás*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2023. május 24.

Taska Tamás
hallgató

Kivonat

A Rubik kocka az utóbbi években egyre nagyobb népszerűségnek örvend, annak ellenére, hogy már majdnem 50 év eltel a feltalálása óta. Bár szinte mindenki látott már ilyet, viszonylag kevesen tudják az összekevert kockát kirakni. A kocka kirakására sokféle algoritmus létezeik, vannak egyszerűbbek, kezdők számára, vannak bonyolultabbak haladók számára, és vannak kifejezetten számítógépek számára kifejlesztett algoritmusok is. Ezen algoritmusok leírására létezik egy általánosan elfogadott jelölésrendszer, noha ez a jelölérsrendszer nem alkalmas teljes algoritmusok leírására, hanem csak rövid mozgáskombinációkat szokás megadni ezzel. A megadott mozgáskombináció előfeltétele, valamint eredménye nem írható le ezzel a jelölésrendszerrel, hanem azt természetes nyelvvel (pl. angol) szokás megadni.

Az ehhez kapcsolódóan felmerülő másik kérdés, hogy a kocka állapotát milyen módon lehet leírni. Sajnos általános megoldás nem terjedt el erre, talán a legközelebbi megoldás, amit hivatalos versenyeken alkalmaznak. Ott a kocka állapotát egy mozgáskombinációval adják meg, ahol a kiindulási állapot egy kirakott kocka. Ez a módszer viszont nem alkalmazható abban az esetben, ha az ember nem tudja, hogy hogyan jutott el az adott állapothoz.

Ezekre a problémára nyújt megoldást az általam fejlesztett 2 szakterületi nyelv. Az egyik a Rubik kockák állapotának leírására alkalmas, míg a másik az algoritmusok teljes leírását teszi lehetővé, tehát az előfeltételek és eredmény is leírható vele.

A két nyelv kipróbálására egy egyszerű webalkalmazást készítettem, ami HTTP hívásokon keresztül csatlakozik a nyelv feldolgozást végző szerverekhez. A webalkalmazás tartalmaz egy szövegszerkesztőt, melyen keresztül a két nyelvhez tartozó fájlokat szerkeszthetjük, valamint egy egyszerű szimulátort, ami a kocka állapotát mutatja, illetve az algoritmus által elvégzett forgatásokat is vizualizálja.

Abstract

The Rubik's cube has grown in popularity in recent years, despite the fact that it has been almost 50 years since its invention. Although almost everyone has seen one, relatively few people know how to solve the mixed cube. There are many different algorithms to solve it, some simple for beginners, some more complex for advanced cubers, and some algorithms designed specifically for computers. There is a generally accepted notation system for describing these algorithms, although this notation system is not suitable for describing complete algorithms, since it is usually used to describe only short combinations of movements. The precondition and the result of a given combination of movements cannot be described with this notation system, but are usually given in natural language (e.g. English).

Another question that arises is how to describe the state of the cube. Unfortunately, no general solution has been found so far, but perhaps the closest solution is the one used in official competitions. There, the state of the cube is given by a combination of moves, where the initial state is a loaded cube. However, this method is not applicable if one does not know how one arrived at the state.

These problems are solved by the 2 domain specific languages I have developed. One is suitable for describing the state of Rubik's cubes, while the other allows a complete description of the algorithm, so that it can be used to describe both the preconditions and the result.

To test the two languages, I created a simple web application that connects to the language servers via HTTP calls. The web application includes a text editor to edit the files for the two languages, and a simple simulator to show the state of the cube and visualize the rotations performed by the algorithm.

1. fejezet

Bevezetés

A Rubik kocka az utóbbi években egyre nagyobb népszerűségnek örvend, annak ellenére, hogy már majdnem 50 év eltelt a feltalálása óta. Bár szinte mindenki látott már ilyet, viszonylag kevesen tudják az összekevert kockát kirakni. A kocka kirakására sokféle algoritmus létezeik, vannak egyszerűbbek, kezdők számára, vannak bonyolultabbak haladók számára, és vannak kifejezetten számítógépek számára kifejlesztett algoritmusok is. Ezen algoritmusok leírására létezik egy általánosan elfogadott jelölésrendszer, noha ez a jelölérsrendszer nem alkalmas teljes algoritmusok leírására, hanem csak rövid mozgáskombinációkat szokás megadni ezzel. A megadott mozgáskombináció előfeltétele, valamint eredménye nem írható le ezzel a jelölésrendszerrel, hanem azt természetes nyelvvel (pl angol) szokás megadni.

Valamint az ehhez felmerülő másik kérdés, hogy a kocka állapotát milyen módon lehet leírni. Sajnos általános megoldás nem terjedt el erre, talán a legközelebbi megoldás, amit hivatalos versenyeken alkalmaznak. Ott a kocka állapotát egy mozgáskombinációval adják meg, ahol a kiindulási állapot egy kirakott kocka. Ez a módszer viszont nem alkalmazható abban az esetben, ha az ember nem tudja, hogy hogyan jutott el az adott állapothoz.

Ezekre a problémára nyújt megoldást az általam fejlesztett 2 szakterületi nyelv. Az egyik a Rubik kockák állapotának leírására alkalmas, míg a másik az algoritmusok teljes leírását teszi lehetővé, tehát az előfeltételek és eredmény is leírható vele.

A szakterületi nyelveket az ANTLR keretrendszer segítségével készítettem el, a szemantikus ellenőrzést, és a kimenet előállítását Go nyelven írtam meg. Az így elkészítette language szerver mellé egy egyszerű HTTP szervert húztam fel, mely lehetőséget nyújtott arra, hogy a bemenetként kapott szövegeket feldolgozza és válaszként visszaadja a nyelvi szerverek kimeneteit. A jobb felhasználói élmény miatt egy egyszerű webalkalmazást is elkészítettem, mely lehetőséget ad szövegek szerkesztésére, melyet beküld a HTTP szervernek, és megjeleníti az eredményt. A megjelenítés egyfelől szöveges (a kimeneti forgatások, és a hibaüzenetek), másfelől pedig vizuális (szimulátor ami mutatja a kocka állapotát és a forgatásokat).

2. fejezet

Felhasznált eszközök és technológiák ismertetése

Ebben a fejezetben a munka során felhasznált technológiákat és eszközöket ismertetem.

2.1. Rubik kocka

2.1.1. Algoritmusok részleges leírása

2.2. Fordítóprogramok általános működése

2.3. Backend technológiák

2.3.1. ANTLR

2.3.2. Go programozási nyelv

2.4. Typescript

2.5. Docker

3. fejezet

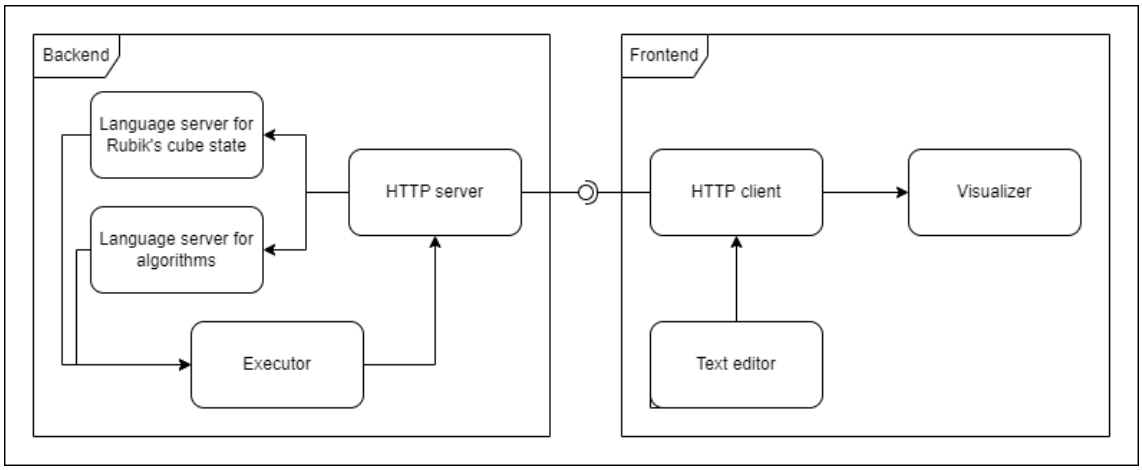
Architektúra tervezése

Ebben a fejezetben az architektúra tervezésének a folyamatát, valamint a meghozott tervezési döntéseket ismertetem. Ezen kívül az architektúra implementációja során felmerülő lehetőségekről is ejtek pár szót.

3.1. Architektúrális alapok

Az architektúra alapvetően kétszintű architektúra, egy frontend és egy backend részből áll. A frontend felelős a szövegek szerkesztéséért valamint a szervertől kapott adatok megjelenítéséért, míg a backend az állapot- és algoritmusleírások ellenőrzéséért, valamint a leírt algoritmus futtatásáért felel.

A két rész HTTP-n keresztül kommunikál egymással, így a frontend részen van HTTP kliens, míg a backend tartalmaz egy HTTP szervert. A frontend ezen kívül egy szövegszerkesztőt, valamint egy szimulátort is tartalmaz. A backend a két szakterületi nyelv feldolgozó szerverét, valamint egy executort tartalmaz, ami az algoritmus konkrét futtatásáért felel. Ezt a nézetet a 3.1. ábra mutatja be.



3.1. ábra. Magas szintű architektúrális nézet

3.2. Frontend komponensei

3.3. Backend komponensei

A két nyelvfeldolgozó szerver a HTTP szervertől kapja a feldolgozandó adatokat, vagyis a szövegfájlokat. Ennek oka, hogy így a későbbiekben könnyebben lehet kiegészíteni a szervereket, például az LSP kompatibilitáshoz szükséges funkciókkal. Ezek után a szerverek elvégzik a fájlok ellenőrzését és feldolgozását, majd a kimenetüket továbbítják az Executor felé. Az Executor feladata a bemenetként kapott állapotleírásból és algoritmusleírásból meghatározni a konkrét forgatásokat, amik a Rubik kocka kirakásához szükségesek.

3.4. Kapcsolat a komponensek között

Mivel a két komponens HTTP-n keresztül kommmunikál egymással, ezért meg kellett határoznom a kommunikációs interfészt. Ehhez következő HTTP endpointokat hoztam létre.

3.5. Fejlesztői mód

A fejlesztés közben gyakran kellett újraindítanom a backend illetve a frontend komponenst, hogy a módosított verzió fusson. Erre a frontend esetén az npm ad támogatást, de backendhez nem nagyon van ilyesmi, így készítettem egy saját programot, ami kezelte ezt.

3.6. Futtatási környezet

A tervezés során fontos eldönteni, hogy milyen környezetben fog futni a végleges rendszer. Mivel egy webalkalmazásról és a mögötte álló backend szerverről van szó, ezért kapásból felmerül a kérdés, hogy egy helyen szeretnénk-e futtatni a két szervert, vagy valamilyen szinten szeretnénk-e elkülöníteni. A közös környezet előnye, hogy a kommunikáció egyszerűbb és gyorsabb, ugyanakkor nehezebben kiegészíthető a rendszer, illetve skálázni is nehezebb. Ha viszont külöün futtatjuk, például különálló konténerekben, akkor valamennyi kommunikációs overheaddel számolnunk kell, viszont a kiegészítés és skálázás is egyszerűbb. Ezen felül a konténeres megoldás előnye, hogy sokkal platformfüggetlenebb, mert a konténerek eltérő operációs rendszereken is ugyanúgy viselkednek.

Ezen okokból kifolyólag döntöttem úgy, hogy az elkészült rendszert különálló konténerben futtatom. Ehhez mind a backend szerverhez, mind pedig pedig a frontendet kiszolgáló szerverhez el kell készítenem egy-egy Dockerfile-t ami tartalmazza a konténer konfigurációját, illetve egy compose fájlt, amivel egyszerűen tudom indítani a két konténert. Így könnyen futtatható a rendszer különböző környezetekben. A konténereknek célszerű Linux alapú konténereknek lenniük, hiszen azok sokkal kisebbek, mint Windowsos társaik.

4. fejezet

Frontend elkészítése

Ebben a fejezetben a szimulátor kiválasztását és testreszabását mutatom be. Mivel a szimulátor feladata kizárólag a Rubik kocka, valamint az azon elvégzett forgatások vizualizációja, ezért a frontenden kap helyett.

4.1. Szimulátor

A szimulátor elkészítése nem teljes mértékben saját munka, mivel a diplomamunka során elsősorban a nyelvek fejlesztése volt a középpontban, illetve egy szimulátor elkészítése a nulláról jelentős többletmunkát jelentett volna. Éppen ezért első körben körbenéztem, hogy milyen nyílt forráskódú, szabadon felhasználható megoldások vannak, amelyeket tudnék integrálni. A kiválasztott szimulátort természetesen módosítanom kellett, hogy megfeleljen a saját és rendszer igényeinek is.

4.1.1. Open source szimulátorok

4.1.2. A megfelelő szimulátor kiválasztása

A megfelelő szimulátor kiválasztásánál figyelembe kellett vennem, hogy milyen célokra szeretném azt használni:

- Legyen nyílt forráskódú és szabadon felhasználható, hogy fel tudjam használni
- Fusson böngészőben, hogy illeszkedjen az eddigi architektúra terveim közé
- Legyen könnyen paraméterezhető, hogy a backend szerverről érkező adatokat könnyen meg lehessen jeleníteni
- Legyen dinamikus, hogy ne csak statikusan a kockát jelenítse meg, hanem a forgatásokat is meg tudja jeleníteni
- Lehessen testre szabni a kocka méretét, hogy ne csak 3x3-as Rubik kockán működjön

Mivel igen kicsi volt az esélye annak, hogy találok egy olyan implementációt, ami ezek közül mindnek megfelel, ezért kénytelen voltam eldönteni, hogy mely elemek fontosabbak, és melyek kevésbé.

Az általam választott szimulátor [1] böngészőben futott, megjelenítette a forgatásokat is, valamint be lehetett állítani a kocka méretét (bizonyos határokon belül), ám a másik két kritériumot nem teljesítette. Nyílt forráskódú volt, de nem volt licenszelve, így nem használhattam fel semmihez. Valamint nem igazán lehetett parametrizálni sem.

A licensz probléma miatt írtam a szimulátor készítőjének, hogy tisztázzuk, szándékosan nem licenszelte-e, vagy pedig véletlenül maradt le. Miután megbeszéltük, hogy mire is szeretném használni, tett hozzá egy MIT licenszfájlt. A paraméterezhetőséget pedig teljes mértékben nekem kellett megoldanom, viszont az alkalmazás szerkezetét és a vizuális részeket meghagytam közel érintetlenül.

4.1.3. A szimulátor kódjának módosítása

A szimulátor kódján két fontosabb módosítást végeztem, egyrészről a kódminőségen javítottam, másrészről Javascriptről Typescriptre írtam át az egészet.

4.2. Szövegszerkesztő

A szakterületi nyelvek kódjának szerkesztésére a Monaco szövegszerkesztőt integráltam a webalkalmazásomba.

4.2.1. Open source szövegszerkesztők

4.2.2. Monaco integrálása

4.3. Az oldal összerakása

A szövegszerkesztőt és a szimulációt el kellett helyeznem az oldalon, valamint szükségem volt egy olyan helyre az oldalon, ahol a szöveges kimenetet tudom leírni.

5. fejezet

Nyelvek

Ebben a fejezetben a 2 szakterületi nyelv megtervezését, a meghozott tervezői döntéseket és a nyelvek implementációját mutatom be.

5.1. Állapotleíró nyelv

5.1.1. Hatókör

Ezzel a nyelvvel fogom leírni a Rubik kocka állapotát, valamint azt is, hogy milyen kockáról van szó. Ez kezdetben a méret leírását, később pedig akár a Rubik test típusát (kocka, dodekaéder) is jelenti. 2 módot terveztem, egyet kezdők, egyet pedig haladóbbak számára.

5.1.2. Szintaxis

5.1.3. Nyelvtani szabályok

5.1.4. Szemantikus ellenőrzés

5.1.5. Kimenet

Az állapotleíró nyelv kimenete egy olyan adatstruktúra, amely egyértelműen és hiba nélkül leírja a kocka állapotát, könnyen értelmezhető, és amelyet az Executor képes használni.

5.2. Algoritmusleíró nyelv

5.2.1. Hatókör

Az algoritmusleíró nyelvvel a teljes algoritmus le kell tudnom írni, azaz nem csak a forgatásokat, hanem az előfeltételeiket és céljukat is, valamint azt is, hogy hogyan juthatunk el valamelyik előfeltétel állapotba.

5.2.2. Szintaxis

5.2.3. Nyelvtani szabályok

5.2.4. Szemantikus ellenőrzés

5.2.5. Kimenet

Az algoritmusleíró nyelv kimenete egy olyan adatstruktúra, amely egyértelműen és hiba nélkül leír egy algoritmust, könnyen értelmezhető, és amelyet az Executor képes használni.

6. fejezet

Executor

Ebben a fejezetben az executor részt mutatom be, amely az algoritmusok konkrét végrehajtásáért felel a backend oldalon. Ez a komponens kapja meg a Rubik kocka állapotát, valamint az algoritmus absztrakt leírását, és végrehajtja azt, eredményként azokat a forgatásokat adja vissza, amiket a kockán végrehajtva a kocka kirakott állapotba kerül.

6.1. Kapott adatstruktúrák

6.1.1. Puzzle struktúrája

6.1.2. Algoritmus struktúrája

6.2. Kimenet

Az executor által adott kimenetet a webalkalmazás megjeleníti, valamint az ebben leírt forgatások lesznek láthatóak a szimulátorban is.

7. fejezet

Esettanulmány

Ebben a fejezetben az elkészített szoftvert mutatom be, egy rövid példán keresztül.

8. fejezet

Összefoglalás, továbbfejlesztési lehetőségek

A munka összefoglalása

Köszönetnyilvánítás

Ezúton szeretném megköszönni Jonathan Taylornak, aki kérésemre publikusan elérhetővé és szabadon felhasználhatóvá tette saját Rubik kocka szimulátorát [1], amelyet alapul véve készítettem el a saját szimulátoromat.

Irodalomjegyzék

[1] Jonathan Taylor: Jonathan Taylor's Rubik's Simulator. https://github.com/taylorjg/rubiks-cube. [Online; accessed 15-May-2023].

Függelék

F.1. Egy minta algoritmus leírás

```
helpers:
    upsideDown: x2
steps:
    step white_up:
        goal: white(Up 1 1)
        runs: 4
        branches:
            if white(Down 1 1):
                do: upsideDown
            if white(Front 1 1):
                do: x
            prepare: y
    step white_cross:
        goal: orientation([(Up, Front), (Up, Right), (Up, Back), (Up, Left)])
        runs: 20
        branches:
            if orientation(Up, Front):
                do: y
            if piece(Up, Front) at (Up, Front):
                do: F R' D' R F2 y
            if piece(Up, Front) at (Front, Down):
                do: F2
            if piece(Up, Front) at (Up, Right):
                do: R2 D'
            if piece(Up, Front) at (Up, Back):
                do: B2 D2
            if piece(Up, Front) at (Up, Left):
                do: L2 D
            if piece(Up, Front) at (Right, Front):
                do: F
            if piece(Up, Front) at (Front, Left):
                do: F'
            if piece(Up, Front) at (Left, Back):
                do: L'
            if piece(Up, Front) at (Back, Right):
                do: R
            prepare: D
    step white_side:
        goal: orientation([(Up, Front, Right), (Up, Right, Back), (Up, Back, Left), (Up, Left, Front)
    ])
        runs: 16
        branches:
            if orientation(Up, Front, Right):
                do: y
            if piece(Up, Front, Right) like position(Right, Up, Front):
                do: R' D' R D
            if piece(Up, Front, Right) like pos(Front, Right, Up):
                do: R' D R
            if (Up, Front, Right) like position(Down, Right, Front):
                do: R' D2 R D
            if (Up, Front, Right) like (Front, Down, Right):
                do: D' R' D R y
            if piece(Up, Front, Right) like (Right, Front, Down):
                do: D F D' F' y
            if piece(Up, Front, Right) at (Up, Back, Right):
```

```
            do: B' D' B
        if piece(Up, Front, Right) at (Up, Back, Left):
            do: L' D2 L
        if piece(Up, Front, Right) at (Up, Left, Front):
            do: L D L'
        prepare: D
step second_layer:
    goal: orientation([(Front, Right), (Right, Back), (Back, Left), (Left, Front)])
    helpers:
        right: D' R' D R D F D' F'
        left: D L D' L' D' F' D F
    runs: 16
    branches:
        if orientation(Front, Right):
            do: y
        if place(Front, Right) or piece(Front, Right) like (Front, Down):
            do: right
        if piece(Front, Right) like (Down, Front):
            do: D y left
        if piece(Front, Right) at (Front, Left):
            do: left D2
        if piece(Front, Right) at (Left, Back):
            do: y' left y D'
        if piece(Front, Right) at (Back, Right):
            do: y right y' D
        prepare: D
step yellow_up:
    do: upsideDown
step yellow_cross:
    goal: yellow([Up 0 1, Up 1 0, Up 1 2, Up 2 1])
    helpers:
        fromL: F U R U' R' F'
        fromDash: F R U R' U' F'
    runs: 4
    branches:
        if yellow([Up 0 1, Up 1 0]):
            do: fromL
        if yellow([Up 1 0, Up 1 2]):
            do: fromDash
        if none(yellow(?), [Up 0 1, Up 1 0, Up 1 2, Up 2 1]):
            do: fromDash U2 fromL
        prepare: U
step yellow_edges:
    goal: orientation([(Up, Front), (Up, Right), (Up, Back), (Up, Left)])
    helpers:
        swapTwo: R U R' U R U2 R' U
    runs: 8
    branches:
        if orientation([(Up, Back), (Up, Right)]):
            do: swapTwo
        if orientation([(Up, Back), (Up, Left)]):
            do: y
        if orientation([(Up, Front), (Up, Right)]):
            do: y'
        if orientation([(Up, Front), (Up, Left)]):
            do: y2
        prepare:
            do: U
            consecutive: 3
        prepare: swapTwo
step yellow_corners:
    goal: place([(Up, Front, Right), (Up, Right, Back), (Up, Back, Left), (Up, Left, Front)])
    helpers:
        cycleCorners: U R U' L' U R' U' L
    runs: 9
    branches:
        if place(Up, Front, Right):
            do: cycleCorners
        prepare:
            do: y
            consecutive: 3
        prepare: cycleCorners
```

```
    step yellow_corners_orient:
        goal: orientation([(Up, Front, Right), (Up, Right, Back), (Up, Back, Left), (Up, Left, Front)
    ])
        runs: 4
        branches:
            if yellow(Up 2 2):
                do: U
            if yellow(Front 0 2):
                do: 4(R' D' R D) U
            if yellow(Right 0 0):
                do: 2(R' D' R D) U
    step orient_top_layer:
        goal: orientation(Up, Front, Right)
        runs: 3
        do: U
```

F.1.1. lista. Minta algoritmus egy 3x3-as Rubik kocka kirakására