[Külső fekete borítólap formátuma]

Széchenyi István Egyetem

Gépészmérnöki, Informatikai és Villamosmérnöki Kar

Informatika Tanszék

**SZAKDOLGOZAT**

**Herenkovics András**

**Mérnök Informatikus BSc szak**

[2018]

|  |
| --- |
| [Gerincen:] Herenkovics András, 2018 |



**SZAKDOLGOZ****AT**

**Csomagok kiszállításának optimalizálása**

**Herenkovics András**

**Mérnök Informatikus BSc szak**

**2019**

# Nyilatkozat

Alulírott, Herenkovics András (AQLRX3), Mérnökinformatika, BSc szakos hallgató kijelentem, hogy a Csomagok kiszállításának optimalizálása című szakdolgozat feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

Győr, [2019. március 9.]

hallgató

# Kivonat

Csomagok kiszállításának optimalizálása

[1 oldalas, magyar nyelvű tartalmi kivonat]

# Abstract

Optimalization of package delivering

[1 oldalas, angol nyelvű kivonat]

# Tartalomjegyzék

[1. Genetikus algoritmusok 2](#_Toc3124859)

[1.1. Történeti áttekintés 2](#_Toc3124860)

[1.2. Működése 3](#_Toc3124861)

[1.2.1. Reprezentáció 4](#_Toc3124862)

[1.2.2. Inicializálás 6](#_Toc3124863)

[1.2.3. Kiértékelési-, cél- és jósági függvény 6](#_Toc3124864)

[1.2.4. Szelekció 7](#_Toc3124865)

[1.2.5. Keresztezés 8](#_Toc3124866)

[1.2.6. Mutáció 12](#_Toc3124867)

[1.2.7. Reprodukció 12](#_Toc3124868)

[1.2.8. Kilépési feltétel 13](#_Toc3124869)

[1.3. A dolgozatban használt módszer 13](#_Toc3124870)

[1.3.1. Reprezentáció 13](#_Toc3124871)

[1.3.2. Kiértékelési- és jósági függvény 14](#_Toc3124872)

[1.3.3. Szelekció 14](#_Toc3124873)

[1.3.4. Keresztezés 14](#_Toc3124874)

[1.3.5. Mutáció 15](#_Toc3124875)

[1.3.6. Reprodukció 16](#_Toc3124876)

[1.3.7. Kilépési feltétel 16](#_Toc3124877)

[2. Tervezési dokumentáció 17](#_Toc3124878)

[2.1. Programmal szemben támasztott követelmények 17](#_Toc3124879)

[2.2. UML diagrammok 17](#_Toc3124880)

[2.2.1. Aktorok – funkciók 17](#_Toc3124881)

[2.2.2. ER diagram 18](#_Toc3124882)

[2.2.3. 19](#_Toc3124883)

[2.3. Felhasznált technológiák ismertetése 19](#_Toc3124884)

[2.3.1. Verziókezelő rendszer 19](#_Toc3124885)

[2.3.1.1. Git 20](#_Toc3124886)

[2.3.2. Frontend technológiák 20](#_Toc3124887)

[2.3.2.1. HTML 20](#_Toc3124888)

[2.3.2.2. CSS 21](#_Toc3124889)

[2.3.2.3. JavaScript 21](#_Toc3124890)

[2.3.3. Backend technológiák 22](#_Toc3124891)

[2.3.3.1. Java 22](#_Toc3124892)

[2.3.3.2. Spring Boot 22](#_Toc3124893)

[2.3.3.3. Thymeleaf 22](#_Toc3124894)

[2.3.3.4. Hibernate 23](#_Toc3124895)

[2.3.3.5. PostgreSQL 24](#_Toc3124896)

[2.3.3.6. Maven 24](#_Toc3124897)

[2.3.4. Fejlesztői eszközök 25](#_Toc3124898)

[2.3.4.1. IDE 25](#_Toc3124899)

[2.3.4.2. pgAdmin 25](#_Toc3124900)

# Ábrajegyzék

1. ábra: Genetikus algoritmus működésének folyamatábrája 4

2. ábra: Sorrendi reprezentáció bemutatása 5

3. ábra: A rulettkerék kiválasztás 7

4. ábra: Egy pontos keresztezés 9

5. ábra: Több (4) pontos keresztezés 9

6. ábra: Egyenletes keresztezés egy utóddal 9

7. ábra: PMX működése 10

8. ábra: Az OX működése 11

9. ábra: A CX működése 11

10. ábra: Gén értékének vátozása 12

11. ábra: Gén értékek cseréje és az inverzió 12

12. ábra: HGrex működése 15

13. ábra: Aktorok - funkciók 18

# Forráskódjegyzék

1. Forráskód: Thymeleaf 22

2. Forráskód: pom.xml 24

# Bevezetés

Témaválasztásomnál olyan feladatot próbáltam keresni, mellyel nemcsak a programozási készségeimet tudom fejleszteni, hanem elméleti kihívást is biztosít. Így kerültem szembe a csomagok kiszállításának optimalizálásával, aminek elvégzésével mindkét célomat elérhetem.

A mai társadalomban szinte senkinek sincs ideje semmire, de ha van, akkor se szeretnék bevásárlással tölteni, ami igen csak időigényes. Webshop-ok és áruszállító vállalatok ezrei épülnek ezen igények kiszolgálására. Mivel mindenki szeretné, hogy az általa megrendelt termékeket minél hamarabb megkapja, egy komoly logisztikai feladatot állítanak a szállító cégek felé. A cégek, mint mindig, szeretnék csökkenteni a működési költségeiket. Ennek az egyik módja lehet, ha a sofőrök nem tesznek meg felesleges kilómétereket a célpontok között. Ezért szükségük van olyan szoftverekre, ami nem csak két pont között adja meg a legrövidebb utat, hanem több cél közötti összes lehetséges útvonal közül kiválasztja az optimális körutat, ezzel jelentősen redukálva a megtett kilóméterek számát.

Szakdolgozatom egyik fő célja, ha nem a legfontosabb, hogy a futárok számára meghatározza az optimális útvonalat a csomagok kiszállításához. Vagyis adott *n* pont (cím) a köztük lévő távolságokkal és ezek között kell megtalálni a legrövidebb utat úgy, hogy a kiinduló- és végpont (telephely) ugyanaz, valamint minden pontot csak egyszer érintünk. Ez a feladat az utazó ügynök probléma néven lett híres a tudomány világában. Ennek megoldása első hallásra talán egyszerűnek tűnhet (és kevés célpont esetén az is), de a célpontok számának növekedésével a megoldás bonyolultsága rendkívül megnőhet, hiszen ez egy NP teljes probléma (nem determinisztikusan polinomiális). Az ilyen NP teljes feladatok megoldásához jól használhatók a meglévő, jó- és kevésbé jó algoritmusok [0a,0b], heurisztikák [5a,0c] mellett a genetikus algoritmusok [5].

A genetikus algoritmusok működési elve az elővilágban tapasztalható evolúciós folyamatok utánzásán alapul, számítógépes környezetben különböző optimalizálási problémák megoldására alkalmazható. A feladat megoldása során én is genetikus algoritmust használok, melyet részletesen be is fogok mutatni a szakdolgozat későbbi részében.

A dolgozat első felében bemutatom a feladat megoldásához használható technológiákat, technikákat, illetve azt, hogy melyiket és miért választottam ezek közül. A második felében ismertetem az elkészült szoftver részeit, működését, használatát.

1. Genetikus algoritmusok

Nagyon sok olyan feladat van, amelyre nincs ismert algoritmus, vagy ismert ugyan, de az nem hatékony, nem gyors, vagy éppen nem lehet megkeresni az optimális megoldást. Sok ilyen feladattal találkozhatunk a keresés illetve az optimalizálás témakörében. A genetikus algoritmusok egyik fő előnye, hogy a problémák nagyon széles osztályára alkalmazható, mert általában nem használ területfüggő tudást, így akkor is működik, ha a feladat struktúrája kevéssé ismert. Az evolúciós algoritmusokhoz tartozó számítási modellnek tekinthető genetikus algoritmusok optimalizálási és keresési problémákra alkalmazhatók, melyet a fejezet megírásához használt [3],[5],[8],[9] irodalmak alapján mutatok be.

* 1. Történeti áttekintés

Az 1950-es években merült fel el az a gondolat több tudósban is, hogy a biológiai evolúció a mérnöki problémákban is felhasználható lehet optimalizációs eszközként. Az alap ötlet az volt, hogy lehetséges megoldások halmazából a legjobb megoldásokat különböző, természetes kiválasztódás és genetikus változások által ihletett operátorok használatával keressék ki.

Ingo Rechenberg publikálta 1965-ben az evolúciós stratégiák alapjait [6][7], mely alapvető ma használt fogalmakat nem használt még, mint a keresztezés, de a populációk már megjelentek benne. Fogel, Owens és Walsh 1966-ban kifejlesztették az evolúciós programozás technikáját [2], amiben egy adott feladat lehetséges megoldásait véges állapotú automatákként ábrázolták, melyek fejlődését az állapotátmeneti diagrammjuk véletlenszerű mutációjával és a legjobbak szelektálásával oldották meg. Számos más kutató dolgozott még evolúció inspirálta algoritmusokon (Box, Friedman, Bledsoe, …), de egyikük munkája se kapott annyi figyelmet, mint az evolúciós stratégiák, az evolúciós programozás vagy a genetikus algoritmus.

A darwini evolúciós elméletet [1] és a genetika alapjait magába építő szigorúan vett genetikus algoritmusok első változatát, az egyszerű genetikus algoritmusokat John Holland, a Michigani Egyetem professzora javasolta 1975-ben [4]. Ez a publikáció tekinthető az első mérföldkőnek a genetikus algoritmusok történetében és ez az eredmény nevezhető még ma is a genetikus algoritmusok elméleti alapjának. Holland genetikus algoritmusa a populációban lévő kromoszómákból hozza létre a populáció következő generációját szelekció, mutáció, keresztezés és inverzió műveletekkel. Minden kromoszóma génekből áll, melyek lehetséges értékeit génváltozatoknak vagy alléloknak nevezünk. Az utazó ügynök problémában például, egy útvonalat reprezentálhat egy kromoszóma, míg egy várost egy gén. Minél jobb egy kromoszóma rátermettsége, annál nagyobb eséllyel választják ki a szelekció művelete során, vagyis annál nagyobb eséllyel adhatja tovább génjeit a következő generáció számára. A mutáció megváltoztatja véletlenszerűen kiválasztott gének értékét, a keresztezés (rekombináció) két kromoszómát vág két- vagy több részre, majd ezeket egymásközt felcserélve új kromoszómát alkot. Az inverzió a kromoszómán belül kiválasztott szakaszt fordítja meg, ezzel változtatva annak felépítését. Az azóta eltelt évek alatt az evolúciós módszerek komoly fejlődésen mentek keresztül és a kutatók egymásra hatása egyre jobban elmosta a határokat a genetikus algoritmusok, az evolúciós stratégiák, az evolúciós programozás és a többi evolúciós megközelítés között. Mai kutatásokban gyakran használják a „genetikus algoritmus” kifejezést olyan dolgokra, amik messze állnak Holland eredeti leírásától.

* 1. Működése

Az algoritmus működése többféle módon mehet végbe, de az alábbi fő részek szinte mindegyik formában megtalálhatók: inicializálás, kiértékelés, kiválasztás, keresztezés, mutáció, reprodukció. Az 1. ábra az előbb felsorolt lépéseket szemlélteti egy folyamatábrán.

Mutáció

Reprodukció

Kilépési felétel

Kiválasztás

Keresztezés

Kiértékelés

Inicializálás

Nem

Igen

1. ábra: Genetikus algoritmus működésének folyamatábrája

* + 1. Reprezentáció

A természetes kiválasztódás nagyban függ az egyedek reprezentációjától, hiszen a szülő kromoszómákon végrehajtott műveletek által tovább vitt génekkel jönnek létre az újabb generációk. A kódolásra többféle lehetőség van. A klasszikus módszer szerint egy kromoszómát egy adott hosszúságú bitsztringben ábrázoljuk. Itt a legnagyobb kérdés a leképezés módszere. Van, ahol ez egyszerűen megoldható, de ahol nem, ott használható a Gray-kódolás, ami a szomszédos egészekhez 1 Hamming-távolságú kódot rendel (vagyis 1 bitben térnek csak el az egymás melletti értékek). Bináris kód mellett használhatók még a lebegőpontos számok vagy a gráf alapú reprezentációk.

TSP megoldására használt módszer többek között a szomszédsági vektor, sorrendi reprezentáció [2y] és az útvonal-vektor.

Szomszédsági vektornál egy vektor segítségével írjuk le az utakat. Ha a j. város az i. pozícióban van, akkor és csak akkor vezet az út az i. városból a j. városba. Például a (2 4 8 3 9 7 1 5 6) vektor az 1 - 2 - 4 - 3 - 8 - 5 - 9 - 6 - 7 utat reprezentálja. Az első pozíción a 2. város van, tehát 1.-ből megyünk 2.-ba. A 2. helyen a 4., tehát a 2.-ból megyünk a 4. városba. A 4. pozícióban a 3., vagyis a 4. város után a 3.-ba érkezünk... Előfordulhat, hogy a vektor illegális utat reprezentál, ilyen például a (2 4 8 1 9 3 5 7 6) vektor, mely 1 - 2 - 4 - 1 körutat jelentené. Ennél az ábrázolásnál ezért nem használhatók egyes keresztezések, mint például az egy pontos keresztezés (lásd később).

Sorrendi reprezentációnál is egy vektor segítségével írjuk le az utat. A kódolás egy referencia túra alapján készül. *N* város esetén ez egy *N* hosszú sztring, a lenti példában ez 12345678 lesz. A kódolás városoként halad, az aktuális város helyét a referencia túrában a kapott kromoszóma megfelelő pozíciójában tároljuk. A referencia túrából minden lépés után eltávolítjuk az utoljára használt várost. A módszert a 2-es ábrán láthatják, melyben az első oszlopban az aláhúzott szám jelzi az aktuális várost.

Aktuális túra:

Referencia túra

Sorrendi reprezentáció

1 2 5 6 4 3 8 7

1 2 5 6 4 3 8 7

1 2 5 6 4 3 8 7

1 2 5 6 4 3 8 7

1 2 5 6 4 3 8 7

1 2 5 6 4 3 8 7

1 2 5 6 4 3 8 7

1 2 5 6 4 3 8 7

2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

3 4 5 6 7 8

3 4 6 7 8

3 4 7 8

7 8

3 7 8

7

1 1

1

1 1 3

1 1 3 3

1 1 3 3 2

1 1 3 3 2 1 2

1 1 3 3 2 1

1 1 3 3 2 1 2 1

2. ábra: Sorrendi reprezentáció bemutatása

A módszer megfordításával könnyen dekódolható a reprezentáció túrává. Kissé meglepő módon, de ha ezen a módszeren alkalmazzuk az egy pontos keresztezést, mindig valid utódot fog létrehozni.

Az útvonal-vektor ábrázolás a legtermészetesebb reprezentálási módszere a problémának. Mint a neve is utal rá, egy vektorban tároljuk a városokat, ahol ha *j* város az *i.* elem, akkor *j* várost *i*.-ként látogatjuk meg a sorban. Például ABCDEF körút 4. eleme D, amit csak A,B és C után érünk el.

* + 1. Inicializálás

Ahogy az 1.-es ábrán látható, az új generáció egy iteráció során, több lépésen keresztül automatikusan jön létre, de a kezdeti populációnál más a helyzet. Ezt az inicializálás során hozzuk létre. Mivel az esetek nagy részénél nem ismerjük a viszonylag jó megoldásokat, ezért véletlenszerűen válogatunk az értékkészletből, hogy megalkossuk a kiindulási kromoszómákat. Természetesen léteznek olyan esetek, amikor vannak előismereteink, ilyenkor a jól megválasztott kezdeti értékek tudják gyorsítani az algoritmus futását. Azonban ezek a heurisztikák plusz munkát jelentenek és mivel a genetikus algoritmus gyorsan tudja javítani a kezdeti rossz egyedeket ezek használta nem feltétlen éri meg a ráfordított erőforrásokat.

* + 1. Kiértékelési-, cél- és jósági függvény

A kiértékelési függvény (evaluation function), másnéven célfüggvény (objective function) és az általuk a kromoszómáknak adott értéket felhasználó jósági vagy rátermettségi függvény (fitness function) megválasztása kritikus fontosságú a genetikus algoritmus kidolgozásánál, mivel ezek sokszor fognak lefutni az iteráció során, ezért fontos, hogy ha lehet, ne legyen nagy az időigénye, könnyen elvégezhető számításokból álljon. A célfüggvény adta értékek megadják, hogy egy megoldás mértéke mennyire jó vagy rossz. Ezeket az értékeket használja a fitneszfüggvény és mondja meg a skálázás során az egyes egyedeknek a többihez viszonyított jóságát, rátermettségét. Ezért a jósági függvényt úgy kell kialakítani, hogy az megfelelően tükrözze az elemek közötti különbséget és a probléma megoldása szerinti jóságát, azaz jobb megoldáshoz jobb értéket rendeljen. Alkalmazható módszer például, hogy a jó megoldásoknál a jóságukat adjuk meg, míg a rosszakénál a hozzájuk legközelebb álló jó megoldástól való távolságukat. Másik módszer a büntető függvény, melynek során az egyedek értékét, annak hibáinak elfogadhatatlanságától függően különböző súlyokkal számoljuk, majd ezeken a hibapontokon használjuk a fitneszfüggvényt. Ha ismert a rátermettségi függvény, de túlságosan időigényes vagy túl kevés információ áll rendelkezésünkre érdemes approximációs függvényt használni, melynek eredménye ugyan nem lesz annyira pontos, de időt lehet vele nyerni és több egyedet megvizsgálni, ami ellensúlyozhatja a pontatlanságát.

* + 1. Szelekció

A szelekció során kiválasztásra kerülnek azok az egyedek, amelyek tovább adhatják a génjeiket a következő generáció számára a reprodukció során. Ez a kiválasztás a kromoszómák fitnesz értékén múlik. Általában minél jobb a rátermettség, annál nagyobb eséllyel lesz kiválasztva, így a következő generáció a nagyobb jóságú egyedekből lesz létrehozva, vagyis nagyobb eséllyel jobb lesz a rátermettsége, mint az előző generációnak.

Az egyik legismertebb módszer a rulettkerék-kiválasztás vagy fitneszarányos szelekció (roulette wheel-, fitness based selection). Ennek során a kiválasztás esélye arányos a jóság mértékével, valamint egy egyedet többször is ki lehet választani, hiszen a választási műveletek függetlenek egymástól. Ha az egyed jósága *ffi*, akkor kiválasztásának esélye , ahol *N* a populáció elemszáma. A módszer az egyedek rátermettségéhez arányosan rendel szeleteket a „rulettkerékből”, így a „golyó” nagyobb eséllyel áll meg a jobb kromoszómáknál. A rulettkerék-kiválasztást jól szemlélteti a 3. ábra.

Krom. 4 (52)  
47%

Krom. 3 (6)  
5%

Krom. 2 (25)  
24%

Krom. 1 (17)  
14%

Krom. 5 (11)  
10%

Véletlenül generált szám: 40

Kiválasztott kromoszóma: Krom.3

3. ábra: A rulettkerék kiválasztás

Populáció

Krom. 1

Krom. 2

Krom. 3

Krom. 4

Krom. 5

Fitnesz érték

17

25

6

52

11

Ezt a módszert könnyű implementálni és sokszor jó eredményt is ad, azonban vannak esetek, amikor más módszer jobb eredményt szolgáltat. Ilyen lehet a versengő kiválasztás vagy a sorrend alapú kiválasztások.

A versengő kiválasztás (tournament selection) a rulettkerék módszer továbbfejlesztése. Kiválasztunk két egyedet a rulettel vagy egy véletlen szám generátoral, majd ezeket „versenyeztetjük” és a jobb jóság értékű kerül ki győztesként, lehet szülő.

Az utóbbi csoporthoz tartozik a csonkolásos kiválasztás (truncation selection), ahol csak egy adott jósági szint feletti kromoszómákból választunk, de itt viszont a szelekció valószínűsége egyenlő minden egyednek. Szintén sorrend alapú kiválasztás a lineáris sorrendezésen alapuló kiválasztás (linear ranking selection), ahol fitnesz szerint sorba rendezzük az egyedeket. A kiválasztási valószínűségük csak a sorban elfoglalt helyüktől függ, a valószínűség növekmény állandó.

* + 1. Keresztezés

A keresztezés a genetikus algoritmusok egyik alapvető művelete, mely során általában két szülő egyed különböző részeit összetéve új egyede(ke)t hozunk létre. A legegyszerűbb ilyen módszer az egypontos keresztezés (one point crossover). A művelet során a két szülő kromoszómát ugyanazon pozíciójú gén mellett kettévágjuk, és a részeket felcseréljük, ezáltal létrejön két új egyed. Jellemző hibája a módszernek, hogy a pozíciókkal nem egyenlően bánik, például a végpontok minden esetben felcserélődnek, míg a közbülső gének megmaradására jóval nagyobb esély van. A másik elég feltűnő hibája a pozíciós hiba. A vágásoknál ugyanis semmi garancia nincs arra, hogy egy jó génsorozat tovább öröklődik, hiszen arra, hogy a vágás ne a jó sorozaton belül legyen semmi befolyásunk nincs és ez a probléma lehetőség annál nagyobb, minél hosszabb a jó génsorozat. A kisebb géncsoportok viszont könnyen együtt maradhatnak, így létre jöhetnek autóstoppos bitcsoportok, melyeknek semmi funkcionális közük nincs egymáshoz, csak közel helyezkednek el.

Az egypontos keresztezés könnyen tovább fejleszthető, ilyen például a kétpontos- vagy a több pontos keresztezés, melyeket már nem érintik az előbb említett hibák, de még messze nem tökéletesek. A kétpontos során nem egy, hanem két helyen kerül elvágásra az egyed. A több pontos keresztezés is hasonló elven működik, de itt három vagy több helyen lesz vágás. A vágáspontok itt is, mint az egypontosnál, teljesen véletlenszerűen lesznek elhelyezve. Ezeknél a kiválasztásoknál azonban nem lehet tetszőleges csoportokat ötvözni.

0

0

1

0

1

1

1

1

0

1

0

1

1

1

0

1

1

0

1

0

0

0

1

0

1

1

1

1

0

1

0

1

1

1

0

1

1

0

1

0

4. ábra: Egy pontos keresztezés

0

0

1

0

1

1

1

1

0

1

0

1

1

1

0

1

1

0

1

0

1

1

1

1

1

1

0

0

1

1

0

0

0

1

0

1

1

0

0

1

5. ábra: Több (4) pontos keresztezés

Említésre méltó még az egyenletes keresztezés (uniform crossover), ahol az új kromoszóma minden génje egyenlő, 50-50 %-os eséllyel kerül át a szülőktől. Ezt a műveletet már nem érinti a vágásokat jellemző hibák. Ugyanakkor a pozíciós hiba hiánya miatt az adaptálódó génértékek nehezebben tudnak kialakulni, mivel ez a fajta keresztezés könnyen szét is darabolhat bármilyen géncsoportot

0

0

0

1

0

1

1

1

1

0

1

0

1

1

1

0

1

1

0

1

0

1

1

1

1

1

1

0

1

1

6. ábra: Egyenletes keresztezés egy utóddal

A pozíciós hiba kiküszöbölésére alkalmas módszer még a csúsztatott keresztezés, itt a csere előtt mindkét szülőben eltoljuk a biteket egy véletlenszerűen meghatározott értékkel, majd utána fordított irányba is elvégezzük a tolást.

Az említett módszerek olyan esetekben használhatók, ha a gének értékei megegyezhetnek, viszont ha az allélok egyediek más műveleteket kell használni, melyeket átrendező operátoroknak (reordering operators) hívunk.

A részlegesen megfeleltetett keresztezés (partially matched crossover, PMX) során a szülőkben két véletlenszerű vágással egy megfeleltetési szakaszt hozunk létre, melynek elemeit a szülőkön belül megcseréljük a szakaszon belüli párjukkal, a 7. ábra egy ilyen keresztezést mutat be. Az ábrán a megfeleltetési szakasz az első szülőben EAF, a másik szülőben CDE, így a párok E-C, A-D, F-E. A felső szülőn a keresztezés menete:

C

D

B

H

E

G

A

F

E

A

B

D

C

F

G

H

1.

3.

1.

2.

2.

3.

E

A

B

H

F

G

D

C

C

D

B

A

E

F

G

H

7. ábra: PMX működése

1. E – C: megkeressük a pár két tagját az első szülőben, majd megcseréljük őket. (részállás: HBCAFGDE)
2. A – D: itt is megkeressük a géneket és megcseréljük őket. (részállás HBCDFGAE)
3. F – E: lépés ugyan az, mint előbb. Végeredmény: HBCDEGAF.

Az átrendezési keresztezés (order crossover OX) is hasonló képen indul, mint a PMX, létrehozunk egy megfeleltetési szakaszt. Először az egyik egyedből kitöröljük a másik megfeleltetési szakaszában lévő elemeit, majd ezeket az üres géneket úgy tesszük a két vágáspont közé, hogy a kromoszómát két végénél fogva összefűzöttnek tekintjük, és a 2. vágásponttól balra lévő elemeket addig toljuk balra, amíg az üresek középre nem tolódnak. Végül az első elem megfeleltetési szakaszát a második helyére másoljuk. Az OX műveletét a 8. ábrán szemléltetem egy utód létrehozásával, a másik gyerek kromoszóma értéke „AFCDEGHB” lenne.

Az utolsó általam szemléltetett operátor a ciklikus keresztezés (cycle crossover CX) lesz. Itt nem használnak vágópontokat. Az első lépés, hogy az egyik szülő első génjét átmásoljuk a leszármazottba. Majd a másik szülő azonos pozíciójában lévő allélját megkeressük az első szülőben és azt is átmásoljuk az új kromoszómába, de abba a pozícióba, ahol az egyes szülőben helyet foglalt. Ez után megnézzük, hogy az előbb átmásolt allél álláspontjában a második szülőben milyen értéke szerepel, amit megkeresünk az első szülőben és átmásoljuk azonos helyre az utódba. Ez így folytatódik, amíg nem jutunk el olyan allélhoz a második szülőben, ami már szerepel a leszármazottban. Ekkor a maradék helyekre a második szülő értékeit másoljuk át a pozíciókat megtartva. Működését a 9. ábrán lehet megtekinteni.

E

A

B

H

F

G

D

C

C

D

B

A

E

F

G

H

C

D

B

G

H

1. Eltávolítjuk a megfeleltetési szakasz elemeit

D

C

G

H

B

2.Addig toljuk balra az elemeket amíg az üres gének középre nem kerülnek

E

A

D

C

F

G

H

B

3. Végül beillesszük a megfeleltetési szakaszt a másik kromoszómából

8. ábra: Az OX működése

9. ábra: A CX működése

E

A

B

H

F

G

D

C

C

D

B

A

E

H

G

F

A

H

G

D

3. …és ezt addig folytatjuk amíg el nem érünk egy olyan allélig ami már szerepel az új egyedben, ami most a negyedik lépés után következett be.

A

H

2. Megkeressük az előbb átmásolt gén helyén a második egyedben lévő allélját az első egyedben és áthelyezzük az új kromoszómában

H

1. Átmásoljuk az első allélt az első szülőből

C

A

B

H

E

G

D

F

4. Végül a második kromoszóma génjeit átmásoljuk az új egyed üres helyeire.

Másik utód értéke a művelet elvégzése után ABEDFHGC lenne.

* + 1. Mutáció

A mutáció lényege, hogy a meglévő populációból kis valószínűséggel módosítsa a kromoszómák értékét, ezáltal segítve a keresési tér eddig fel nem fedezett területeit felkutatni. Mutáció lehet például bitsztringek esetén egy kiválasztott gén értékének véletlenszerű megváltoztatása, vagy minden gén értékének kis valószínűséggel való módosítása.

10. ábra: Gén értékének vátozása

0

0

1

0

1

1

1

1

0

1

0

0

1

0

0

1

1

1

0

1

Sorrendi kódolás esetén az egyik megoldás lehet, ha egy kromoszómán belül két véletlenszerűen kiválasztott gén értékét megcseréljük, vagy használjuk az inverziót és egy szakasz sorrendjét megfordítjuk.

11. ábra: Gén értékek cseréje és az inverzió

C

D

A

B

E

F

I

J

G

H

C

D

A

B

I

F

E

J

G

H

C

D

A

B

E

F

I

J

G

H

C

D

A

B

I

H

E

J

G

F

* + 1. Reprodukció

A szelekció, keresztezés, mutáció során létrejött köztes populációból és az előző generációból a reprodukció során jön létre az új generáció. Legegyszerűbb módja ennek az ha az előző populációt teljes egészében leváltja a köztes állomány. Ezt egyszerű vagy generációs reprodukciónak hívják (simple, generational repoduction). Mivel semmi garancia nincs arra, hogy a köztes generáció tagjai jobbak, mint az előzők érdemes lehet a legjobba(ka)t megőrizni és ezáltal biztosítani a konvergenciát. Ezt hívják elitizmusnak (elitism). A megtartott tagokat kétféle módon vihetjük tovább: ha kisebb a köztes generáció, mint az előző, akkor csak hozzáadjuk az elit tagokat. Amikor az ideiglenes populáció mérete megegyezik az előzővel, akkor vagy a legrosszabb fitnesz értékű vagy véletlenszerűen kiválasztott kromoszómák kerülnek leváltásra.

* + 1. Kilépési feltétel

A kilépési függvény különféle feltételek teljesülése esetén leállíthatja az iterációt. A legegyszerűbb megoldás, ha adott számú iteráció után leáll az algoritmus. Ehhez előismeretek szükségesek a problémával kapcsolatban. A leállás függhet az aktuális populáció állapotától is. Ha egy egyed jósága nagyon eltér az átlagos fitneszértéktől, akkor feltételezhetjük, hogy van egy kiemelkedő megoldás. Hasonló megoldás ha a szórást vizsgáljuk és az egy adott értéken belül esik, ekkor a kromoszómák a keresési tér egy pontja köré csoportosulnak. Ezeken kívül még kilépési ok lehet, ha a generáció átlagos fitnesz értéke nagyon hasonló vagy megegyezik az előző generációk rátermettségével. A legjobb egyedek konvergenciája is kilépést okozhat. A bináris kromoszómáknál egy gén konvergál, ha például 95%-ban azonos az értéke a populációban, egy populáció konvergál, ha minden gén konvergál.

* 1. A dolgozatban használt módszer
     1. Reprezentáció

A dolgozatban az útvonal kódolást használtam, mely az utazó ügynök probléma genetikus algoritmussal való megoldásánál a legkézenfekvőbb megoldás. Itt a kromoszómákat alkotó gének egész számokat tárolnak, így a kromoszóma egy egész számokból álló tömb. Az allélok a városokat reprezentálják és a tömbben elfoglalt pozíciójuk a körútban megfelelő sorszámukat jelenti. Ha négy város között keressük a legrövidebb körutat: Győr (0), Budapest (1), Szeged (2), Pécs (3), akkor egy kromoszóma a következőkép néz ki, ha Budapest, Pécs, Győr, Szeged az útirend: *[1,3,0,2]*.

* + 1. Kiértékelési- és jósági függvény

A két függvény kiválasztásánál elsődleges szempontom az volt, hogy a megfelelő érték reprezentáció mellett, könnyen és gyorsan elvégezhető legyen. Hiszen, ahogy fentebb is említettem, ezek a függvények sokszor le fognak futni.

A dolgozatban használt kiértékelési függvény kiszámolja a kromoszóma által képviselt út hosszát. Ezt a gén tömbön végig iterálva, a városokat ábrázoló számokat használva, a távolság mátrixból kiolvasott értékek összeadásával éri el.

A jósági függvény a fitnesz értéket a célfüggvény által előállított teljes távolság reciprokát véve adja meg.

* + 1. Szelekció

A szelekciós operátor kiválasztása nagyon fontos lépése volt a genetikus algoritmus elkészítésének. Több operátor (lineáris sorrendezés, versengő, rulettkerék) tesztelése után a döntésem a rulettkerék-kiválasztásra esett, mely nemcsak gyorsabb eredmény mutatott, hanem pontosabbat is. A kiválasztás menetét megtekinthetik a 1.2.4–es fejezetben.

* + 1. Keresztezés

A keresztezési operátor kiválasztása is nagyon fontos lépés volt. Szakirodalmak olvasása [0][4a] és különböző módszerek letesztelése után döntöttem a Heurisztikus Mohó Keresztezés (*heuristic greedy crossover HGreX*) mellett, melyet Grefenstette ajánlott [2a].

A keresztezés a kromoszómát egy irányított körként kezeli. A művelet lépései:

1. Véletlenszerűen kiválasztunk egy várost kezdő pontnak
2. Összehasonlítjuk a kiválasztott város és az őt követő távolságát a másik szülőben lévő ugyanazon város és azt követő távolságával. Majd a kisebb értékűt tesszük a gyerek kromoszóma következő helyére.
3. Ha a rövidebb távolságot jelentő város egy kört eredményezne, akkor a hosszabbat válaszuk. Ha ez is egy kört eredményezne, véletlenszerűen választunk egy várost, ami nem okoz kört.
4. 2. és 3. lépés ismétlése, amíg minden gént felhasználunk
5. Mivel a városok közötti távolságokat is használjuk a keresztezés során így szükségünk a távolságmátrixra is. Az operátor működését a 12. ábrán mutatom be.

12. ábra: HGrex működése

A

1. Kiválasztunk egy kezdő várost:

F

A

2. A 🡪 F (3) és A 🡪 H (5) összehasonlítása. Mivel F közelebb van ezért ő lesz a következő város

G

A

C

D

F

H

B

E

A

H

D

F

B

C

E

G

Szülő kromoszómák:

Távolság mátrix:

D

F

A

3. F 🡪 H (4) és F 🡪 D (3) összehasonlítása.

D

C

F

A

4. D 🡪 A (3) és D 🡪 C (4) összehasonlítása. DA rövidebb út, de túl hamar kört okozna, ezért C-t választjuk

D

C

F

A

G

B

E

H

5. Előzőek alapján következő C 🡪 G. Mivel a két lehetőség G 🡪 A és G 🡪 F kört okozna, ezért egy véletlenszerű, eddig nem felhasznált városba megyünk, és onnan folytatjuk. Ha ez B, akkor a végeredmény:

A

B

C

D

E

F

G

A

0

7

3

2

8

3

4

B

7

0

5

4

3

7

1

C

2

5

0

3

6

9

3

D

3

4

4

0

6

3

1

E

9

2

6

6

0

2

5

F

3

7

9

3

2

0

6

G

4

1

2

3

7

6

0

H

5

8

7

9

3

4

1

H

5

8

7

8

4

3

1

0

* + 1. Mutáció

Mint feljebb említettem, egyedi allélokat tartalmazó kromoszómáknál lehetséges opció az inverzió vagy két véletlenszerű elem felcserélése. Több lehetőség megfontolása után, az inverziós mutáció mellett döntöttem. A [0] és [1a] forrásokban felvetett operátorok között voltak olyanok, amelyek jobb eredmény produkáltak (IRGIBNNM, SBM), de ez a különbség nagy mennyiségű célpont (több száz) esetén érződött leginkább. Az általam elkészített webshop-ban az egyszerre kiszállítható rendelések száma limitált, hiszen a fuvarozó gépjármű rakodó tere se végtelen. Így a célpontok száma nem lesz annyira jelentős, hogy érezhető legyen a különbség. Másik szempont, ami miatt az inverzió mellett döntöttem, a sebesség. Mindkét jobb fitneszt eredményező mutáció több mutáció részeredményeire épít, ami plusz számításokat igényelne.

* + 1. Reprodukció

A konvergencia megőrzése érdekében elitizmust használtam a reprodukció során. A köztes populáció szülőpopuláció – n kromoszómát tartalmaz, ehhez a szülőkből az n legjobb (elit) bekerül a köztes populációba, ami leváltja a szülő populációt.

* + 1. Kilépési feltétel

Az algoritmus kilépési feltétele egy adott számú iteráció lefutása lett, mivel a probléma és a szükséges paraméterek is ismertek, így meg lehet választani azt a mennyiséget, ami után nagy esélyjel egy közel optimális eredményt szolgáltat.

1. Tervezési dokumentáció
   1. Programmal szemben támasztott követelmények

A tervezés első lépése az volt, hogy meghatároztam mit várok el a kész programtól, merre induljon el a fejlesztés. Ezek a következők:

* Könnyen kezelhető, letisztult, minimalista felhasználói felület.
* Megfelelő felhasználói szerephez, megfelelő jogosultságok biztosítása.
* Gyors interakció a backend és a frontend között.
* A kiszállítóknak optimális csomagszállítási útvonal nyújtása.
* Egyszerű telepítés.

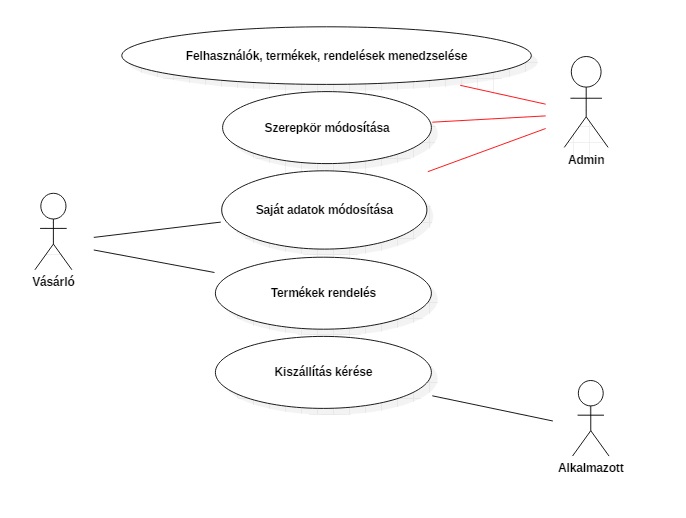
Látható, hogy ezek kezdetleges elvárások, melyek elsősorban iránymutatóként szolgáltak és a fejlesztés során bővülhetnek.

* 1. UML diagrammok

A követelmények specifikációja után UML (Unified Modeling Language) módszer és leíró nyelvet használtam, amellyel grafikus modelleket készítettem, a rendszer specifikálására. Ezek a diagrammok nagyban segítettek jobban átlátni a követelményeket.

* + 1. Aktorok – funkciók

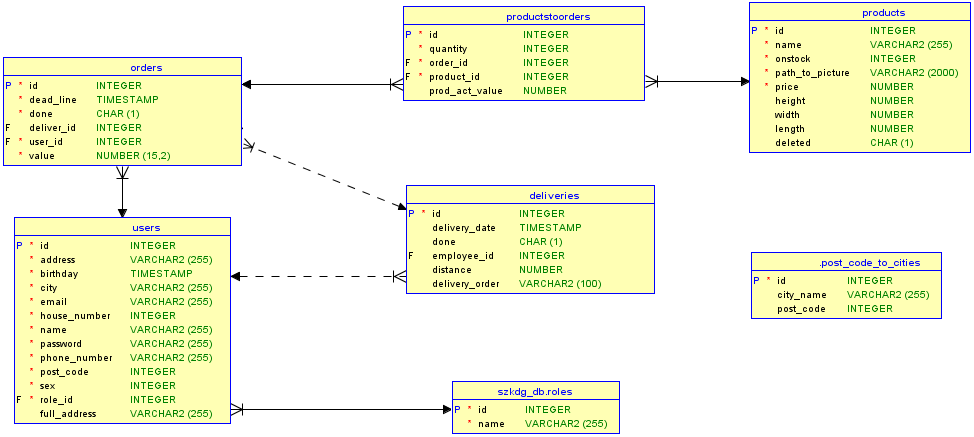
A webshop háromtípusú felhasználót különböztet meg (vásárló, alkalmazott, admin). Mind a három típus más jogokkal rendelkezik és a bejelentkezés után más oldalra érkeznek meg, más lehetőségeket látnak az oldalon. A felhasználó típusokat és az általuk használatba vehető lehetőségek a 13. ábrán láthatók.



13. ábra: Aktorok - funkciók

* + 1. ER diagram

Miután eldöntöttem, hogy milyen információkat kell hosszú távon megőriznem, már csak azt kellett kitalálni milyen formában legyen ez megvalósítva az adatbázisban. Ugyan a Hibernate-nek hála nem kellett a táblákat manuálisan SQL szkirpteket futtatva létrehoznom, de az osztály attribútumokhoz megfelelő annotáció és típus kiválasztásában segítségemre volt a készített ER diagram.



14. ábra: ER diagram

* 1. Felhasznált technológiák ismertetése

Az alkalmazásban több különböző technológiát, fejlesztőeszközt használtam, melyeket a következő fejezetekben ismertetek röviden.

* + 1. Verziókezelő rendszer

Az adatvesztés elkerülése és egy esetleges hiba egyszerű kijavítása érdekében verziókezelőrendszert használtam a fejlesztés során. Főbb előnyei a verziókezelő rendszereknek:

* Backup és visszaállítás
* Szinkronizáció
* Változások követése
* Felelősség követése
* Elágazás (branching) és összefésülés (merging)

A verziókezelőknek két fő fajtájavan:

* Centralizált. A hagyományos verziókezelők központosított modellel dolgoznak, ahol minden verziókezelési művelet egy közösen használt szerveren történik. Ha többen egyszerre akarnak módosítani egy fájlt, akkor valahogy le kell kezelni a konkurens viselkedést. Kétféleképpen oldják meg ezt a problémát: zárolással és összefésüléssel.
* Decentralizált. Itt egy központi tároló (angolul repository) helyett minden felhasználó gépe egy-egy külön tárolóként jelenik meg. A szinkronizáció az egyes gépek között küldött patch-ek (módosításcsomagok) által valósul meg. Ez a megközelítés jelentős változásokat okoz (nincs nagy központi adatbázis, gyakori műveletek gyorsabbak, minden munkamásolat egy-egy backup). Két fajta elosztott verziókezelő létezik, a nyitott és a zárt. [10]

Egy személyes projekteknél, mint a szakdolgozatírás, csökken a jelentősége a két típus közötti választásnak. Munkám során egy decentralizált verziókezelőt használtam, melyet már előző projektjeim során megismertem, a Git-et. Választásom azért a erre esett, mert így szükség esetén internetelérés nélkül is tudok verziókezelő műveleteket végrehajtani.

* + - 1. Git

A Git-et eredetileg Linus Torvalds fejlesztette ki a Linux kernel fejlesztéséhez. Minden Git munkamásolat egy teljes értékű repository teljes verziótörténettel és revíziókövetési lehetőséggel, amely nem függ a hálózat elérésétől vagy központi szervertől. [https://hu.wikipedia.org/wiki/Git]

Legfontosabb git parancsok:

* git init: lokális repository létrehozása
* git status: megmutatja azokat a fájlokat, amik különböznek az index-től
* git add: fájlok hozzáadása a következő commit-hoz
* git commit: változások mentése a helyi repository-ba
* git push: távoli referenciák frissítése, hogy megegyezzen a lokálissal (fájlok feltöltése, ha szükséges).
* git pull: távoli repository lekérése, és merge-ölése

Híres Git használó cégek és projektek: Google, Facebook, Linux, Netflix, PostgreSQL, android, ecplise, Rails, twitter, LinkedIn, Perl.

Remote szervernek a GitHub-ot választottam, melyen megtekinthető és könnyen visszakereshető a verziói a szoftvernek a következő linken: <https://github.com/HerenkovicsA/szkdga>.

* + 1. Frontend technológiák

A frontend technológiák felelősek a felhasználóval közvetlenül interakcióba kerülő részekért. A weboldal kinézetéért és a közvetlen felhasználói tevékenyégek kiszolgálására három technológiát használtam, ezek:

* HTML
* CSS
* JavaScript
  + - 1. HTML

HyperText Markup Language (HTML) a weboldalak és más böngészőben megjelenő dokumentumok elkészítéséhez használt nyelv. Pontosabban az a nyelv, amely a dokumentum struktúráját és a szemantikus tartalmát jellemezi. HTML egy nemzetközi szabvány, amelynek a leírása a [World Wide Web Konzorcium](http://www.w3.org/) és a [WHATWG](http://www.whatwg.org/) által karbantartott. A HTML egy élő szabvány, amely technikailag folyamatos fejlesztés alatt áll. A jelenlegi HTML szabvány verziójára HTML5 ként hivatkozunk. [4b]

* + - 1. CSS

A Cascading Style Sheet (CSS) egy stílusleíró nyelv, ami egy HTML oldal formázására használható. Egy CSS stíluslap szabályokból épül fel, melyek tartalmaznak egy szelektort és egy deklarációs részt. A szelektor adja meg, hogy mely elemkere kell használni a szabályt. Ez lehet egy HTML elem vagy egy elem attribútuma (class, id). A deklarációk szerkezete a következő: „tulajdonság: érték;”. A tulajdonságok angol kulcsszavak, míg az értékek szintén angol szavak, számok és mértékegységek lehetnek (pt, px, %, stb.).

Használatának számos előnye van. Csökkenti a redundanciát a HTML kódban, nem kell minden elemhez ugyanazt a stílust többször leírni, csak létre kell hozni egy szabályt egy adott szelektorhoz és a böngésző érvényesíteni fogja azt. Így a karbantartás is leegyszerűsödik, hiszen ha több elem kinézetét akarjuk módosítani elég csak egy helyen elvégezni a műveletet. Csökkentheti az adatforgalmat, mivel a böngészők gyakran tárolják a CSS fájlokat a gyorsítótárban, így azt nem kell többször lekérni.

* + - 1. JavaScript

„A JavaScript (JS) egy kis erőforrás-igényű, értelmezett vagy JIT-fordított programozási nyelv elsőrendű függvényekkel[[1]](#footnote-1). Bár legtöbben weboldalak parancsnyelveként ismerik, sok webböngészőn kívüli környezetben is használják. Ilyen a node.js, az Apache CouchDB és az Adobe Acrobat. A JavaScript egy prototípus-alapú, többparadigmás, dinamikus nyelv, ami támogatja az objektumorientált, imperatív és deklaratív (pl. funkcionális) programozási stílusokat.” [4c]

JavaScript legtöbbször kliens oldalon fut és segítségével megadható az oldal működése különböző események bekövetkeztében (oldal betöltésénél, kattintásnál, stb.). JavasScript lehet HTML dokumentum *<script>* elemén belül, vagy külön .js kiterjesztésű fájlban.

* + 1. Backend technológiák

A backend egy adott rendszer alsóbb, a tényleges feldolgozást végző rétege. A back-end réteg feladata a front-end réteg felől érkező adatok feldolgozása, ill. a keletkezett eredmény a front-end számára történő visszajuttatása.

* + - 1. Java

A Java egy általános célú, objektumorientált, szerver oldali programozási nyelv. A nyelvet a Sun Microsystems fejlesztette, majd 2009-ben az Oracle felvásárolta a céget. A nyelv elterjedésének fő oka a „write once, run anywhere” mottó (írj egyszer, futtass bárhol). A Java fordító a kódot bájtkódra fordítja, melyet a JVM (*Java Virtual Machine*) fordít natív kódra, amit az operációsrendszer értelmez.

A szakdolgozat forráskódjának nagy részét java nyelven írtam, a JDK (Java Developent Kit) 1.8.0\_101 verzióját használva. Azért használtam ezt a nyelvet, mert széles körben elterjedt és rengetek dokumentum, könyv íródott róla, keretrendszerek készültek hozzá, melyeket munkám során fel tudtam használni. Választásom oka még, hogy a kód könnyen többszálasítható, platform függetlenségnek köszönhetően szinte akármilyen környezetben lehet használni. Ezen túl tanulmányaim és munkám során ezzel ismerkedtem meg a legjobban, így célszerű választás volt.

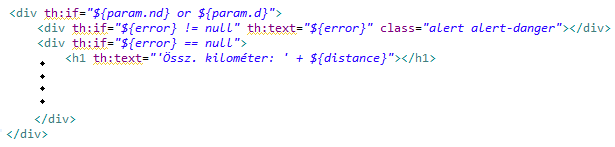
* + - 1. Spring Boot

A Spring (Boot) majdnem minden álláshirdetésben követelmény vagy előny Java programozók számára, ami nem is csoda, hiszen ez a legnépszerűbb keretrendszer Java fejlesztők körében. Nagymértékben egyszerűsíti egy projekt indulását és fejlesztését is. A Spring Boot egy Spring-re épült projekt, ami tovább egyszerűsíti a fejlesztők dolgát. Kiválasztja a függőségeket, automatikusan konfigurálja a használni kívánt funkciókat, segíti a telepítést.

Annak ellenére, hogy egyetemi tanulmányaim keretén belül sajnos nem tudtam találkozni ezzel a technológiával, fontosnak tartottam használatát a projektben a fentebb említett okok miatt..

* + - 1. Thymeleaf

A felhasználói felület elkészítéséhez használni akartam egy olyan keretrendszert, amely jól együtt működik a Spring-gel és nagyban segíti a munkám hatékonyságát. Így találtam rá a Thymeleaf-re, ami egy modern szerver oldali Java template engine. Segítségével a HTML dokumentumban is könnyen kezelhetők a Java objektumok, ciklusok és feltételes elágazások is írhatók. Használatát egy példával szemléltetem, mely az 1. Forráskódban láthatnak.



1. Forráskód: Thymeleaf

Az első sorban lévő *th:if* attribútum arra szolgál, hogy csak akkor jelenítse meg a *div* tartalmát, ha az url paraméterei között szerepel a „d” vagy az „nd”. A második sorban lévő *th:if* szerepe is hasonló. Ha a szerver által küldött objektumok között szerepel *error*  nevezetű (vagyis értéke nem null), megjeleníti a HTML elemet, melynek szövege a *th:text* attribútum értékében lévő *error* objektum tartalma lesz. Mivel *else* ág nincs a Thymeleaf-ben, ezért azt is külön meg kell vizsgálni, hogy az *error* objektumnak nincs értéke. A *h1* elemben lévő rész, azt szemlélteti, hogy statikus szöveggel is össze lehet fűzni a kapott objektumok értékeit.

* + - 1. Hibernate

Objektum-orientált szoftverrel és relációs adatbázissal dolgozni fárasztó tud lenni. Az adatok ábrázolásánál felmerülő különbségek miatti plusz munka jelentősen növeli a fejlesztési időt. Ennek az időnek a csökkentésére lett készítve a Hibernate, ami objektum-relációs leképzést (Object/Relational Mapping, ORM) biztosít java platfromra. Az ORM az adat objektum reprezentációból relációs adat reprezentációra való át- és visszaváltását jelenti.

A Hibernate nem csak ebben a reprezentációváltásban segít, hanem az adat lekérdezésekben is. Alapvető célja, hogy megszabadítsa a fejlesztőt a JDBC és SQL segítségével készített adattárolási programrészek nagy részétől. [3b]

Az osztályok és adattáblák közötti leképzés annotációk vagy XML fájlok segítségével történik. Ezen metainformációk megadásával már képes létrehozni a megfelelő táblákat az adatbázisban. Ezeken kívül még fontos, hogy a leképzendő osztálynak legyen egy argumentum nélküli konstruktora és lehetőség szerint definiálja felül az equals() és hashCode() metódusokat.

* + - 1. PostgreSQL

Adatbázis választásánál sok lehetőség volt, mert szinte minden adatbázis található driver, amivel egyszerűen megvalósítható a kapcsolat és végezhetők műveletek. Azért döntöttem a PostgreSQL mellett, mert platfrom független, nyíltforráskódú, megbízható, rengeteg nagy cég/projekt használja (Bitbucket, Instagram, Netflix, Spotify, MIT…).

* + - 1. Maven

A szoftverem külső modulokat használ (spring boot, thymeleaf, spring security, stb.), melyek megint más modulokat használnak. Ezeket összefoglaló néven *dependencies-*nek (függőségeknek) hívunk. A dependency-k megfelelő verzióját egyenként meg kéne keresni, letölteni és a projekthez csatolni. Ez rengeteg értékes időt elvenne a tényleges fejlesztésből (természetesen újra megírni a külső modulokat még több idő lenne). Ezt a problémát oldja meg az Apache Maven, mely egy build automatizáló eszköz.

A Maven használatához elengedhetetlen, hogy a projektben legyen egy pom.xml fájl (**P**roject **O**bject **M**odel), mely többek között a build-eléshez és a dependency-k letöltéséhez tartalmaz fontos információkat [4e]. A 2. Forráskódban bemutatok egy egyszerű pom.xml-t.



2. Forráskód: pom.xml

Konfigurációk magyarázata:

* modelVersion: a POM verziója
* groupId: csoport vagy szervezet, amihez a projekt tartozik
* artifactId: a projekt neve
* version: a build-elt alkalmazás verziószáma
* packaging: hogyan legyen a projekt csomagolva (ha nincs megadva, akkor jar)
* dependencies: itt vannak felsorolva a projekt közvetlen függőségei

A Maven mellett más csomagoló rendszereket is fontolóba vettem, mielőtt döntöttem. Ezek voltak az Apache Ant és a Gradle. Az előbbi egy régebbi rendszer és épp ezért kicsit kezdetleges. A Gradle újabb, gyorsabban buildel és flexibilisebb, mint a Maven. De a build gyorsasága nagy projektek esetén látszik csak meg és mivel a szakdolgozatom nem számít annak, illetve a Maven-t már használtam korábban, így célszerűbb választásnak tűnt.

* + 1. Fejlesztői eszközök
       1. IDE

Az IDE (Integrated development environment) kiválasztása inkább személyes preferencia kérdése. Fejlesztés során az Eclipse Oxygen.3a verzióját használtam, ami a legújabb verzió volt, amikor elkezdtem a munkát. Döntésem azért lett az Eclipse, mert sok plugin érhető el hozzá könnyen (hála a nyílt forráskódnak), régebbi munkáim során már megismertem, az egyik legnépszerűbb IDE a Java fejlesztők körében és ingyenes.

* + - 1. pgAdmin

Fejlesztés során szükségem volt egy grafikus felületre, amin könnyen szemmel tudtam követni az adatbázisban lévő adatok változását. Erre a célra a pgAdmin 4. verzióját használtam. A pgAdmin egy nyíltforráskódú adminisztrációs és fejlesztési platform, ami direkt a PostgerSQL számára készült, így célszerű volt mellette döntenem.

# Irodalomjegyzék

[0] Alain Petrowski, Sana Ben-Hamida: *Evolutionary Algorithms*, John Wiley and Sons, Inc., Hoboken, New Jersey, USA, (2017)

[0c] Andrew B. Kahng, Sherief Reda: *Match twice and stitch: a new TSP tour construction heuristic*, Operations Research Letters. 32: 499–509, 2004

[0a] David L. Applegate, Robert E. Bixby, Vašek Chvátal, William Cook, Daniel G. Espinoza, Marcos Goycoolea, Keld Helsgaun: *Certification of an optimal TSP tour through 85,900 cities*, Operations Research Letters, 11-15, 2009.

[0b] Held–Karp algorithm: *A dynamic programming approach to sequencing problems,*

Journal for the Society for Industrial and Applied Mathematics 1:10, (1962)

[1] Darwin, C. R.: *The Origin of Species*, John Murray, London (1859).

[1a] Esra'a Alkafaween, Ahmad B. A. Hassanat: *Improving TSP Solutions Using GA with a New Hybrid Mutation Based on Knowledge and Randomness*, ArXiv, (2018)

[2] Fogel, L. J., Owens, A. J., Whals, M. J.: *Artificial Intelligence through Simulated*

*Evolution*, Wiley, (1966).

[2y] Jean-Yves Potvin, *Genetic algorithms for the traveling salesman problem,* Annals of Operations Research, 63., 337–370, (1996)

[2a] J. J. Grefenstette, R. Gopal, B. J. Rosmaita, D. Van Gucht: *Genetic algorithms*

*for the traveling salesman problem*, Proceedings of the 1st International Conference on Genetic Algorithms, Mahwah NJ, 160–168, (1985)

[3] Hatwágner, F. M.: *Nagy számításidejű, folytonos változójú célfüggvények optimalizációja evolúciós számítások segítségével,* PhD disszertáció, Széchenyi István Egyetem, Műszaki Tudományi Kar, (2012).

[3b] Hibernate

<https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>, 2019.03.08.

[4] Holland, J. H.: *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, (1975).

[4a] Krunoslav Puljic´ and Robert Manger, *Comparison of eight evolutionary crossover operators for the vehicle routing problem, Mathematical Communications*, 18, (2013)

[4b] HTML

<https://developer.mozilla.org/hu/docs/Web/HTML>, 2019.02.16.

[4c] JavaScript

<https://developer.mozilla.org/hu/docs/Web/JavaScript>, 2019.02.16.

[4e] Maven

<https://maven.apache.org/pom.html>, 2019.03.09.

[5] Mitchell, M.: *An introduction to Genetic Alogrithms*, A Bradford Book The MIT Press (1998)

[5a] Nicos Christofides: *Worst-case analysis of a new heuristic for the travelling salesman problem, Report 388, Graduate School of Industrial Administration, 1976*

[6] Rechenberg I.: *Cybernetic solution path of an experimental problem*, Royal Aircraft Establishment (1965)

[7] Rechenberg, I.: *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien*

*der Biologische Evolution*, Fromman-Holzboog, Stuttgart, (1973).

[8] Sivanandam, S.N., Deepa, S. N.: *Introduction to Genetic Algorithms,* Springer-Verlag Berlin Heidelberg, (2008)

[9] Várkonyiné, K. A. : *Genetikus algoritmusok*, Typotex kiadó, Bp., (2002).

[10] Verziókezelés

[*https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s*](https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s)*,* 2019.02.15.

[A felhasznált szakirodalom megadása]

[A szerzők nevét mindenütt “Családnév, X.” formában kell megadni, ahol X. a szerző keresztnevének (keresztneveinek) kezdőbetűje. Magyar cikk esetén a vessző a családnév és a keresztnév kezdőbetűje közt elhagyható. Ha az egyértelműség megkívánja, a keresztnév kiírható teljesen is.

A dolgozat szerzője szabadon választhat, az A vagy a B típust használja.]

**[A-típus:**

A cikkekre való hivatkozás egy []-be írt sorszámmal történik. A sorszámozást folytonosan kell megtenni, a sorba rendezés alapja az első szerző családneve.]

[1] Szerző1 (, Szerző2 ...): *Cikk címe*

Folyóirat neve, sorszám, kezdőoldal-végoldal, év.

[2] Szerző1 (, Szerző2 ...): *Konferencia-kiadvány-beli cikk címe*

„Konferenciakiadvány:” Konferencia neve, hely, kezdőoldal-végoldal, év.

[3] Szerző1 (, Szerző2 ...): *Könyvcím*

„Könyv:” Kiadó, hely, oldalszám, év.

[4] Szerző1 (, Szerző2 ...): *Kutatási jelentés címe (csak publikus elérhető jelentés!)*

„Kutatási jelentés”: Kutatási projekt neve, intézet, oldal, év.

[5] Szerző: *Disszertáció címe*

„PhD/kandidátusi/stb. disszertáció”: Egyetem, kar neve, év.

[6] *Internetes oldal elnevezése:*

*URL, letöltés ideje (csak konkrét dokumentumra mutató URL adható meg!)*

# Mellékletek

1. [A dolgozat mellékletei, ha vannak]

1. Egy programnyelvnek **elsőrendű függvényei** vannak, ha a nyelv függvényeit úgy kezeljük mint a többi változót. Egy ilyen nyelveben függvényeket adhatunk meg más függvények paramátereként, függvények más függvényeket adhatnak vissza és váltózókhoz is hozzárendelhetjük őket. [↑](#footnote-ref-1)