Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Pogány László

MULTIATTRIBÚTUMOS RENDSZEREK STATIKUS ÉS DINAMIKUS MODELLEZÉSE FUZZY SZIGNATÚRÁKKAL

KONZULENS

Dr. Kóczy T. László

BUDAPEST, 2017

HALLGATÓI NYILATKOZAT

Alulírott **Pogány László**, 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 hitelesített 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.

Kelt: Budapest, 2017. 05. 18.

...…….

Pogány László

Köszönetnyilvánítás

Ezúton szeretném köszönetemet és hálámat kifejezni témavezetőm, Dr. Kóczy T. László professzor úr felé, aki tanácsaival, intelmeivel és szakmai tapasztalatával járult hozzá a diplomatervem megírásához.

Absztrakt

A lágy számítási rendszerek, mint az evolúciós algoritmusok, fuzzy rendszerek, neurális hálózatok etc. megjelenésével lehetővé vált a nagy komplexitású döntéshozási és optimaliációs problémák optimum közeli megoldásainak gyorsabb megtalálása.

E diplomaterv a fuzzy alapú rendszerek alkalmazásának lehetőségeit vizsgálja a valós életből származó gyakorlati feladatokon.

A dolgozat elsőként vázolja az épületfelújításkor felmerülő alapvető számítási problémákat.

Majd főként fuzzy szignatúrák, véges állapotú determinisztikus állapotgépek és bakteriális algoritmusok együttes alkalmazásának lehetőségét tárja fel, miközben betekintést nyújt az épületfelújítási munkálatok kapcsán felmerülő folyamatok modellezésébe és az e terület folyamataihoz kapcsolódó költségek többcélú optimalizációjába.

A dolgozat továbbá javaslatokat ad a problémakör megoldásához tartozó üzleti minőségű szoftver elkészítésére is. Meghatározza egy ilyen szoftver egy lehetséges követelményrendszerét, majd utóbbi mentén a megvalósított szoftver alapjainak főbb tervezési alapelvei, valamint helyenként az implementáció részletei is bemutatásra kerülnek.

A konkrét megvalósítással a fuzzy és bakteriális alapú számítási modellek különböző szakterületeket lefedő megvalósításokban is felhasználhatók.

Kulcsszavak: *fuzzy rendszerek, fuzzy halmazok, fuzzy állapotgépek, determinisztikus véges állapotgépek, fuzzy szignatúrák, optimalizáció, evolúciós algoritmusok, bakteriális-memetikus algoritmus, épületfelújítás, épületmodellezés, folyamatoptimalizáció, aggregációs operátorok, szoftvertervezés*

Abstract

The appearance of soft computing methods - like evolutionory algorithms, fuzzy based systems, neural networks, etc. - has made it possible to give a near optimal solution much faster to highly complex decision making and optimalization problems.

This thesis firstly outlines the basic computing problems occuring in building refurbishment projects.

Next I examine the practical usage and possible applications of fuzzy based systems on real life problems.

I examine the combined application of fuzzy signatures, finite state machines and bacterial algorithms in building refurbishment projects, specifically in the process modelling and cost optimizing parts of the workflow.

In addition I give suggestions for a business quality software regarding these problems. I define a possible set of requirements and based on these I show the main design patterns and parts of the implementation.

The implemented solution makes the fuzzy and bacterial algorithm based computational models usable in various application areas.

Keywords: *fuzzy systems, fuzzy sets, fuzzy state machines, deterministic finite state machine, fuzzy signatures, optimization, evolutionary algorithms, bacterial memetic algorithm, building renovation, building refurbishment protocol, building condition evaluation, process optimization, aggregation operators, software design*

Tartalomjegyzék

1 Bevezetés

A gyakorlati életben felmerülő problémák egy jelentős része számítási értelemben véve a nehezen megoldható problémák közé sorolható, mivel általában nagyon nagy számításigényük. E problémák megoldása során a tulajdonságaikból (például a problématér méretének növekedése a bemenet függvényében) és a jelenlegi számítási megoldásaink korlátaiból adódóan ugyan törekszünk a lehető legoptimálisabb megoldás megtalálására, azonban véges számítási időn belül azokat csak nagyon kis valószínűséggel tudjuk megtalálni.

Léteznek azonban olyan számítási módszerek, amelyek a nagy bonyolultságú problémák megoldása során a megvizsgálandó problématér méretét képesek hatékonyan redukálni, miközben akár az emberi gondolkodási folyamatokhoz közel álló, vagy azt másoló egyszerűsítéseket hajtanak végre.

Ennek következménye, hogy egy kellően jó végeredményhez való eljutás folyamata sokkal gyorsabb, valamint a számítások illeszkednek az emberi gondolkodásmód által végrehajtott egyszerűsítésekre, ezáltal viszonylag könnyen áttekinthetők a megoldáshoz való eljutás lépései.

Hátránya azonban, hogy az ilyen egyszerűsítések használatával a problématérben nem kimerítő keresés kerül végrehajtásra (az ugyanis a feladat komplexitása miatt ez tipikusan kivitelezhetetlen lenne valós időben), ezért eredményként általában csak egy optimum-közeli megoldás áll elő, mely azonban már gyakorlati szempontból jónak mondható.

Ebben a dolgozatban főként olyan modellek vizsgálatával foglalkoztam, amelyek komponensei jól struktúráltak, a változóik hierarchikus rendszerbe foglalhatók, melyek egyes szinteken és részfákon belül összefüggésben állnak egymással. Továbbá megvizsgálásra kerül e módszerek alkalmazásának lehetősége is egy konkrét gyakorlati feladaton. Egy ilyen önkényesen kiválasztott feladaton, amely elsősorban az épületfelújítási munkálatok segítésére irányul megkísérlem egy, az üzleti szempontoknak is megfelelő alkalmazás megtervezését és egyes részeinek kidolgozását is. A további fejezetek iránymutatást tartalmaznak arra nézve, hogyan képzelhető el egy ilyen szoftver

kivitelezése, figyelembe véve az üzleti szektorban a minőségi alkalmazásokkal szemben támasztott követelményeket.

A második fejezetben bemutatásra kerülnek a feladat modellezésére szánt matematikai eszközök (fuzzy szignatúrák, fuzzy állapotgépek) az alkalmazhatóságuk megértéséhez szükséges mértékben, valamint áttekintésre kerülnek az alkalmazásuk következtében felmerülő problémákra irányuló optimalizációs lehetőségek (melyek elsősorban az evolúciós alapú optimalizációs megoldásokra építenek).

A harmadik fejezetben behatárolásra kerülnek a szakterülethez tartozó főbb elvárások, valamint összefoglalásra kerül egy, a feladat megoldását megvalósító szoftver követelményei az alkalmazni kívánt technológiák listájával együtt.

A negyedik fejezet foglalkozik a tervezés azon részleteivel, melyek egy üzleti minőségű és könnyen karbantartható alkalmazás fejlesztésének alapfeltételei, továbbá kitér a számítási és szakterületi modellek alapján hozott az alkalmazás felépítésére vonatkozó döntések ismertetésére is.

Az ötödik fejezet mutatja be az implementáció fontosabb részleteit elsősorban a fuzzy és evolúciós rendszerek vonatkozásában, másodsorban pedig a korábbi fejezetekben támasztott követelmények értelmezésében.

A hatodik fejezet kiértékeli az elvégzett munkát, továbbá információkkal szolgál az alkalmazás jövőbeli kiterjesztésének lehetőségeiről is.

2 Irodalomkutatás

Az alábbiakban részletesebben is bemutatásra kerülnek azon számítási modellek és megoldások, amelyek a komplex struktúrájú, nyelvi jellemzőkkel megadott modellek feldolgozása és kiértékelése céljából lettek kidolgozva.

Bár e matematikai modellek az épületfelújítási munkálatok segítésén kívül akár más, hasonló felépítésű és bonyolultságú problémák kezelésére is alkalmazhatók (például villamos energetikai hálózatok állapotmodellezése), a szemléletesség végett e modellek ebben a fejezetben már az épületfelújítási munkálatokra vonatkozó háttérrel együtt kerülnek bemutatásra.

2.1 Háttérinformáció

Az európai városok történelmi kerületeiben jelentős gondot okoz a nagy tömegben előforduló városi típusú épületek (lakóépületek, régi irodák stb.) műszaki állapotának folyamatos szinten tartása, mely költséges karbantartással, felújítási munkálatok útján kerül megvalósításra. Ez a probléma jelentős mértéket ölt Budapesten, ahol nagy számban fordulnak elő a II. világháború előtt épült épületek, melyek állapotának minősége az elmúlt időkben jelentősen csökkent. Bár ezen épületek élettartamára vonatkozó információk elég korlátozottak (mivel építésük idején tipikusan nem végeztek efféle kalkulációkat, így csak becslésekre lehet alapozni), de egy-egy megfelelő időben ütemezett felújítás akár évtizedekkel is kitolhatja az épületek használati időtartamát.

Az épületfelújítások szükségességének meghatározása viszont már komoly problémákba ütközik, amikor több száz, ezer, esetleg tízezer épület állapot információit kell kiértékelni és összehasonlítani, melyek során szakmailag is megalapozott döntést kell hozni. Az állapotfelmérések kézi összehasonlítása ilyen esetekben már jelentős nehézségekbe ütközik (főként mivel kevés a szakértő, akik munkaideje drága).

A döntéshozatalt tovább nehezítik olyan tényezők, mint például, ha az állapot információk a szakértő által, nem matematikai eszközökkel, hanem többnyire lingvisztikai címkék formájában (pl. „megfelelő”, „átlagos”, „elfogadhatatlan”) kerülnek meghatározásra. Mindemellett az állapotfelmérések különböző időszakokból származhatnak (akár több tíz éves különbség is lehet az egyes felmérések között) és nem ritka, hogy a szakértők véleményei eltérnek egymástól.

A rendelkezésre álló inputokat érintő további nehezítő tényező lehet, ha az állapotfelmérést tartalmazó űrlapoknak eltérő felépítésük (ugyanaz az információ két különböző típusú űrlapon másmilyen bontásban máshol szerepelhet), valamint az azonos tulajdonságot leíró információk két űrlapon eltérő részletességgel is szerepelhetnek.

Nagy problémát okoz, hogy jelenleg nem létezik olyan megfelelő eszköz, amely hatékonyan tudná segíteni, esetleg részben automatizálni a szakmai döntéshozatalt.

Az épületek felújításáról szóló nagy mennyiségű, szakmailag lehető legnagyobb mértékben megalapozott, nagy költségű döntések alapos meghozatalához figyelembe kell venni az épületfelújítási folyamatokban jelen levő lehetséges alternatívákat, optimalizálni kell a jelentkező költségeket (a műszaki és jogi előírások függvényében), a munkálatok során be kell tartani szakma által ellenőrzött és igazolt megoldásokhoz vezető lépéseket és a felújítás folyamán fontosabbnak ítélt struktúrák minőségének javítására kell fókuszálni (szakértői tudás bevitele súlyozás útján).

A fenti problémára egyfajta megoldásként szolgálhatnak a fuzzy állapotgépek (Fuzzy State Machines) [1][2][3][4][5] és a fuzzy szignatúrák (Fuzzy Signatures) [6][7][15][16] kombinációjával kapott fuzzy szignatúra állapotgépek (FSSM – Fuzzy Signature State Machines), valamint populáció alapú evolúciós algoritmusok [9][10][11][17][18], amelyek alkalmazásának részletei a következő alfejezetekben kerülnek ismertetésre.

2.2 Matematikai modellek a szakterület vonatkozásában

Az alábbiakban bemutatásra kerülnek azok a matematikai modellek, amelyek a fentebb összefoglalt problémakörre illeszkedően feltehetőleg képesek lehetnek a szakterületi probléma lényegének megragadására, a feladatok megfelelő reprezentációjára és a felmerülő problémák megoldására.

A fuzzy szignatúrák az épületek struktúrájának leírására, az aggregációs operátorok az információk különböző szinteken történő egyszerűsítésére, a fuzzy állapotgépek a folyamatok lépéseinek leírására és azok költségeinek reprezentálására, a fuzzy halmazok a nyelvi címkék matematikai ábrázolása céljából kerültek felhasználásra. E komplex rendszer fölé épülve az evolúciós megoldások pedig a folyamatok közel-optimális megoldásainak megtalálását hivatottak megtalálni.

2.2.1 Fuzzy halmazok

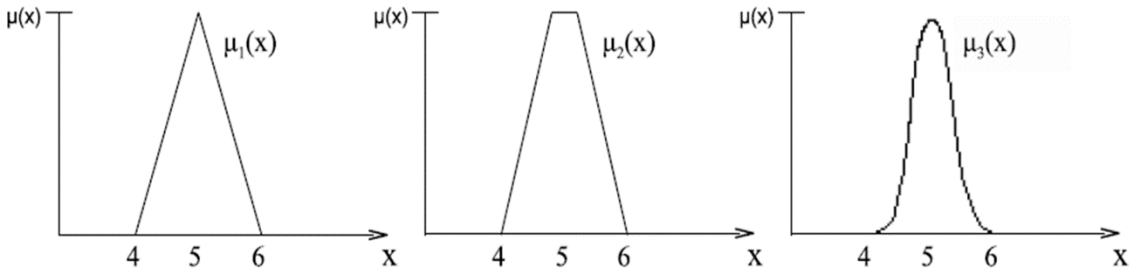
A hagyományos halmazelméletben *X* alaphalmaz *A* részhalmaza esetén bármely $x \in X$ elemre egy karakterisztikus függvény segítségével egyértelműen meghatározható az *A* halmazba tartozás ténye.

$$\kappa_A(x) = \begin{cases} 1, & ha\ x \in A \\ 0, & ha\ x \notin A \end{cases} \quad (1)$$

A fuzzy halmazelméletben azonban a halmazba tartozás fogalma nem ennyire egyértelmű (crisp). Az *A* halmaz ugyanis egy tagsági függvénnyel adható meg (μ_A), mely minden $x \in X$ értékhez egy [0,1] intervallum közötti értéket rendel annak függvényében, hogy az *x* mekkora mértékben tagja az *A* halmaznak.

$$\mu_A(x) = x \rightarrow [0,1], \forall x \in X \quad (2)$$

A tagsági függvény egyértelműen meghatározza a fuzzy halmazt, alakjától függően eltérő mértékű bizonytalanság vihető be a megoldásba.



1. ábra - Példák tagsági függvényekre

Az 1. ábrán látható a három legelterjedtebben használt tagsági függvény: a háromszög, a trapéz és a Gauss-görbe alakú tagsági függvények, melyek az $x=5$ halmazba tartozás fogalmát írják le eltérő módon. [16]

2.2.2 Fuzzy szignatúrák

Mivel az épületek komponensei, részstruktúrái, hierarchikus faszerű szerkezetbe rendezhetők, ahol az egész épület a fa gyökere által van képviselve és a nagyobb komponensek a gyökér első szintű leszármazottjai, a nagyobb komponensek alkomponensei a gyökér második szintű leszármazottjai stb., ezért egy hierarchikus, nyelvi címkékkel operáló leíró struktúra párosítható az épület egyes elemeihez tartozó állapotok leírásához. Ezek a leírók a fuzzy halmazokon alapulnak.

Egy X univerzum fuzzy halmazai a (3) képlet szerint definiálható, ahol X az alaphalmaz, μ_A az X bármelyik elemét egy [0,1] intervallumra leképező függvény.

$$A = \{X, \mu_A\}, ahol\ \mu_A: X \rightarrow [0,1] \tag{3}$$

A vektor értékű fuzzy halmazok a fuzzy halmazok egyszerű kiterjesztései:

$$A_N = \{X, \mu_A\}, ahol\ \mu_A n: X \rightarrow [0,1]^n$$

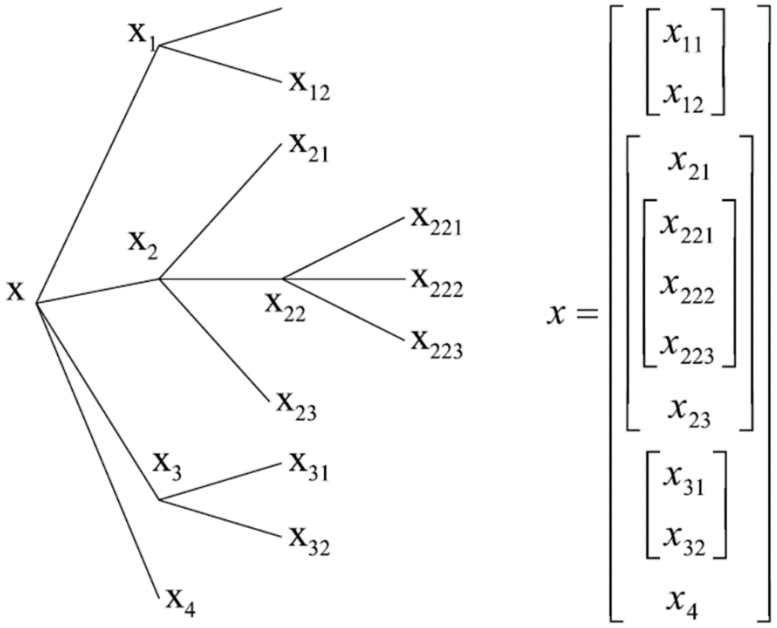
A fuzzy szignatúrák a vektor értékű fuzzy halmazok további kiterjesztései, ahol bármely komponens egy további beágyazott vektor lehet (4).

$$A_{fs} = \left\{X, \mu_{A_{fs}}\right\}, \tag{4}$$

$$ahol\ \mu_{A_{fs}}: X \rightarrow M_1 \times M_2 \times \ldots \times M_n\ ,$$

$$ahol\ M_i = [0,1]\ vagy\ [M_{i1} \times M_{i2} \times \ldots \times M_{in}]^T$$

A 2. ábrán látható a (4) összefüggésnek megfelelő fa-szerű struktúrára illeszkedő példa grafikus kibontása. [16]



2. ábra - Fuzzy szignatúra fa-szerű grafikus ábrázolása és vektorformája

A szignatúrákban így különböző szinteken tárolt fuzzy értékek jelenhetnek meg, ahogy az (5) összefüggésen is látszik:

$$\mu_{A_{fs}} = \left[\mu_1, \mu_2, \left[\mu_{3_1}, \mu_{3_2}, \left[\mu_{3_{3_1}}, \mu_{3_{3_2}}, \mu_{3_{3_3}}\right]\right], \mu_4, \left[\mu_{5_1}, \mu_{5_2}\right], \mu_6\right]^T \tag{5}$$

Az (5) összefüggés értékeikből az egyes szinteken aggregációs operátorok [8] segítségével a hierarchikus elrendezésben található fuzzy értékek akár egyetlen fuzzy értékre is leképezhetők.

A (6) összefüggés bemutatja, hogyan egyszerűsíthető a gyökérig a fenti szignatúra.

$$\{a_0\{a_3\{a_{3_3}\}\{a_5\}\}\} \tag{6}$$

A fentiek alapján az operátorok használata a következő:

$$\mu_{A_fS} \Rightarrow \left[\mu_1, \mu_2, \left[\mu_{3_1}, \mu_{3_2}, \mu_{3_3} = a_{3_3}\left(\mu_{3_{3_1}}, \mu_{3_{3_2}}, \mu_{3_{3_3}}\right)\right], \mu_4, \mu_5 = a_5\left(\mu_{5_1}, \mu_{5_2}\right), \mu_6\right]^T \tag{7}$$

$$\Rightarrow \left[\mu_1, \mu_2, \mu_3 = a_5\left(\mu_{3_1}, \mu_{3_2}, \mu_{3_3}\right), \mu_4, \mu_5, \mu_6\right]^T$$

$$\Rightarrow \mu_0 = a_0(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6)$$

Az (7) összefüggés bemutatja a példaként megadott szignatúra gyökérig történő egyszerűsítését aggregátor operátorok felhasználásával.

A fentiekben említett aggregációs operátorokra egy példa a *WRAO*[1] operátor, mely képes a súlyok alapján történő aggregálásra (8)[5] (ahol *p* az aggregációs faktor).

$$@(x_1, x_2, \ldots x_n, w_1, w_2, \ldots w_n) = \left(\frac{1}{n}\sum_{l=1}^{n}(w_l x_l)^p\right)^{\frac{1}{p}} \tag{8}$$

Mivel az űrlapokon szereplő értékek lingvisztikai címkék segítségével kerültek leírásra, azonban eltérő strukturáltsággal (mélységeiben és részleteiben eltérő űrlapok, épület típusok eltérő felépítései), ezért ilyen címkék halmazát egy-egy fuzzy szignatúrának megfeleltetve bármelyik csúcsra igaz lesz, hogy (közelített) értéke meghatározható a csúcshoz tartozó aggregációs operátorok, valamint a gyerekek értékei alapján.

[1] WRAO – Weighted Relevance Aggregation Operator

2.2.3 Állapotgépek

Az épületfelújítás folyamatának, valamint az egyes lépések költségeinek leírására állapotgépek definiálhatók. Ilyen például a (9) definíció által meghatározott állapotgép, ahol *X* az inputok halmaza, *Q* a köztes állapotok halmaza, *f* pedig az átmeneti függvény, amely meghatározza azt az átmenetet, ahol egy adott input állapot változás előidéz egy állapotátmenetet.

$$A = \langle X, Q, f \rangle \tag{9}$$

$$f: X \times Q \to Q, ahol\ X = \{x_i\}\ és\ Q = \{q_i\}$$

Az új, átmeneti függvény által meghatározott köztes állapot a következő:

$$q_{i+1} = f(x_i, q_i) \tag{10}$$

Melynek mátrixos formája:

$$F = \begin{bmatrix} f(x_1, q_1) & f(x_2, q_1) & \cdots & f(x_n, q_1) \\ f(x_1, q_2) & f(x_2, q_2) & \cdots & f(x_i, q_2) \\ \vdots & \cdots & \ddots & \vdots \\ f(x_1, q_m) & f(x_2, q_m) & \cdots & f(x_n, q_m) \end{bmatrix} \tag{11}$$

2.2.4 Fuzzy állapotgépek

Az állapotgép állapotai *Q* elemei. A jelenlegi alkalmazásban az állapotgépek kiterjesztései fuzzy állapotgépekké a következő értelmezés szerint értendő. A modellezni kívánt jelenség vetületei egy állapot univerzummal vannak reprezentálva, ezen állapotok Q_i alállapotai. Az állapotok maguk (fuzzy) részhalmazai a tárgyalt univerzum állapothalmazának úgy, hogy Q_i-n belül az értelmezés kerete meghatározott (melynek finomsága az alkalmazás kontextusától és az optimalizációs algoritmussal szemben támasztatott követelményektől függ), ezért az állapotok fuzzy halmazokhoz tartozó lingvisztikai címkék által értelmezettek. [2]

Így a kisebb állapotgépek külön definiálhatók az épület különböző komponenseinek felújítási folyamatához, melyek egy nagyobb tárgyalási szinten (szignatúra két részkomponensét összefogó csúcsa) taglalva már nem külön állapotgépként, hanem a két állapotgép együtteseként (állapotgépek szorzataként) jelennek meg.

Az állapotgépek csúcsai fuzzy halmazokhoz rendelhetők (pl. „Teljesen érintetlen”, „Enyhén sérült”, „Közepes állapot” stb.), melyek között az egyik állapotból

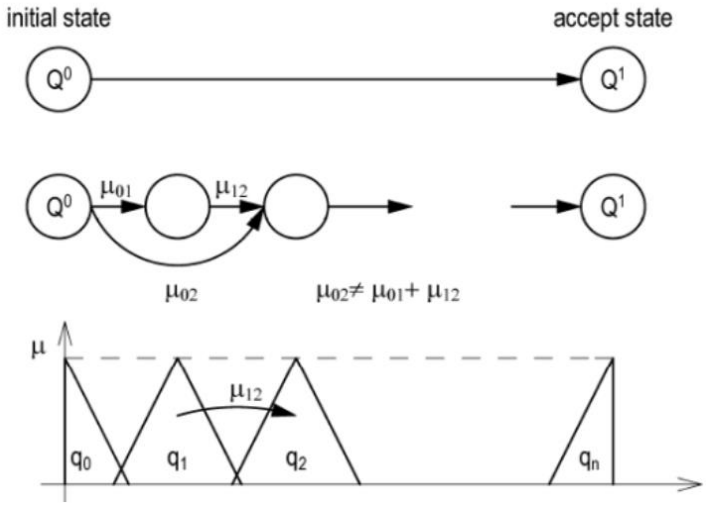
egy másik állapotba mutató tranzíciók[2] állapotjavulást definiálnak, melynek c költsége van.

$$\mu_{ij} = c(q_i, q_j) \tag{12}$$

Egy q_i-ből q_j-be irányuló átmenet egy tagsági értékkel kifejezhető, így a modellben egy út mentén haladva a költségek összegét kapjuk.

$$q_{i_i} \rightarrow q_{i_2} \rightarrow \cdots \rightarrow q_{i_n} \tag{13}$$

Ez a $\sum \mu_{ij}$ összeg nem feltétlenül egyezik meg az egy átmenet mentén q_i-ből q_j-be irányuló költséggel (nem additív tulajdonság), ahogy a 3. ábrán is látszik.



3. ábra - Fuzzy állapotgép állapotátmeneteinek és a fuzzy halmazok viszonya [1]

2.2.5 Fuzzy szignatúra állapotgépek (FSSM[3])

Mivel a szignatúra különböző szintjein a gyerek szinteken levő fuzzy állapotgépek szorzatai biztosítják a teljes felújítási folyamat egy modellben történő reprezentálását, ezért nagy szignatúra esetén a szorzat állapotgép hatalmas méretűvé duzzadhat a gyerek állapotgépek szorzatainak előállítása által. Egy szülő csúcshoz tartozó állapotgép a következő:

$$A^i = A^{i_1} \times A^{i_2} \times \cdots \times A^{i_m} \tag{14}$$

[2] Tranzíció – két csúcs közötti él, átmenet, melynek ez esetben lehet költsége, hasznossága

[3] FSSM – Fuzzy Signature State Machines

ahol A^i-hez tartozó állapotok:

$$Q^i = Q^{i_1} \times Q^{i_2} \times \cdots \times Q^{i_n} \tag{15}$$

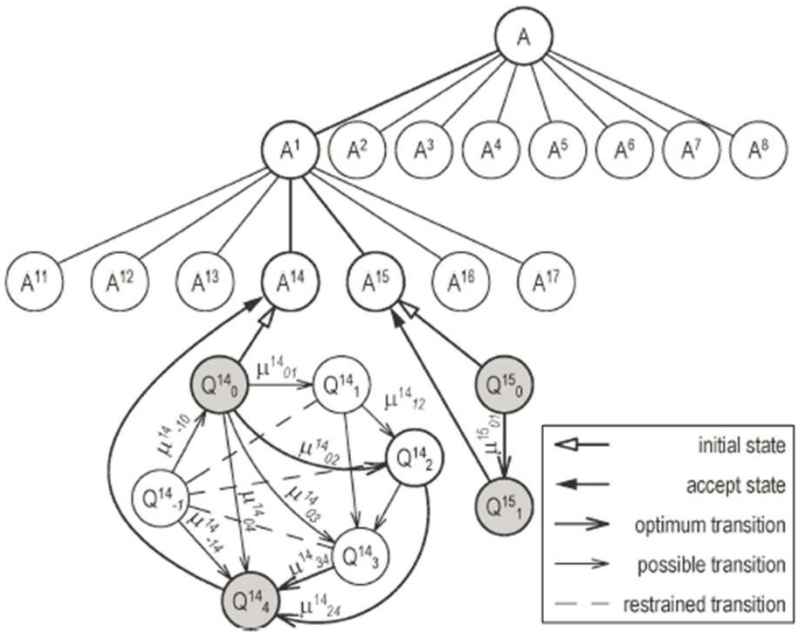
ezért a $Q^{j_1} \rightarrow Q^{j_2}$ átmenet a rész-állapotgépek feletti átmeneteket határoz meg:

$$(q_{j_{1_1}} \rightarrow q_{j_{1_2}}) \times (q_{j_{2_1}} \rightarrow q_{j_{2_2}}) \times \cdots \times (q_{j_{n_1}} \rightarrow q_{j_{n_2}}) \tag{16}$$

Minden levélhez egy speciális aggregációs operátor van hozzárendelve, amely meghatározza a $q_{j_1} \rightarrow q_{j_2}$ átmenethez tartozó $\mu_{j_{1_2}}$ költséget.

$$\mu_{j_{1_2}} = c(q_{j_1}, q_{j_2}) = a_j \left(c\left(q_{j_{1_1}}, q_{j_{1_2}}\right), c\left(q_{j_{2_1}}, q_{j_{2_2}}\right), \ldots, c\left(q_{j_{n_1}}, q_{j_{n_2}}\right) \right) \tag{17}$$

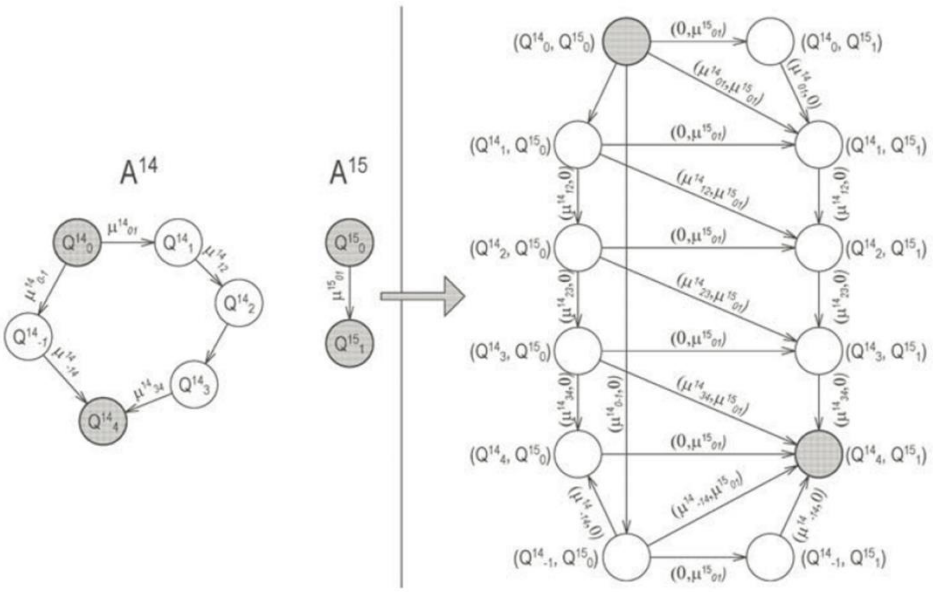
ahol a egy viszonylagos aggregáció, amely nem feltétlenül bír szimmetrikus tulajdonságokkal.



4. ábra - Szignatúrák leveleiben található állapotgépek [1]

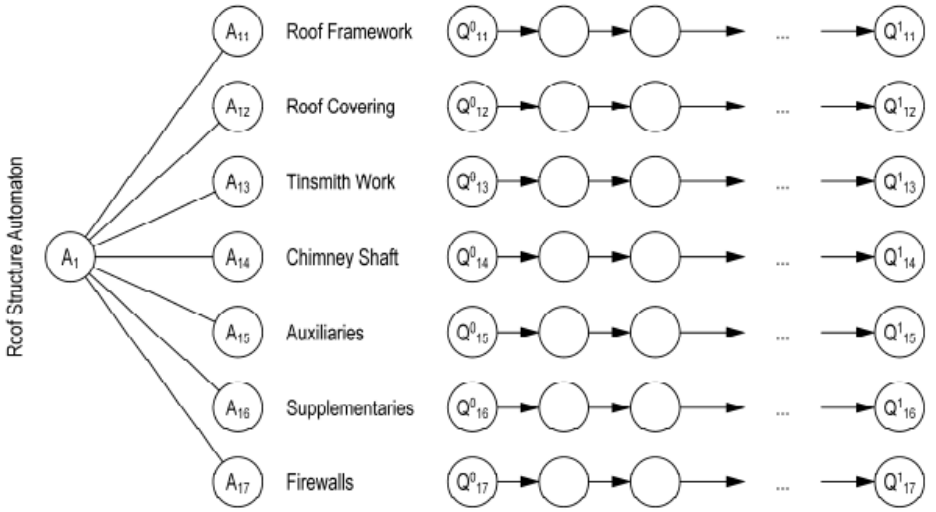
A 4. ábra illusztrálja, hogyan épül fel egy épület tulajdonságait leíró szignatúra, levél csúcsain egy-egy felújítási folyamatot definiáló állapotgéppel és azok átmeneteihez tartozó költségekkel.

A felsőbb szinteken a problémát az alsóbb szintekről származó állapotgépek szorzatai írják le, melyek már kevés állapot esetén is hatalmasra nőhetnek az összes lehetséges állapotpár és a közöttük levő átmenetpárok mennyisége miatt. Ennek illusztrálása látható az 5. ábrán.



5. ábra – Állapotgépek szorzásával előálló szorzat-állapogtép [3]

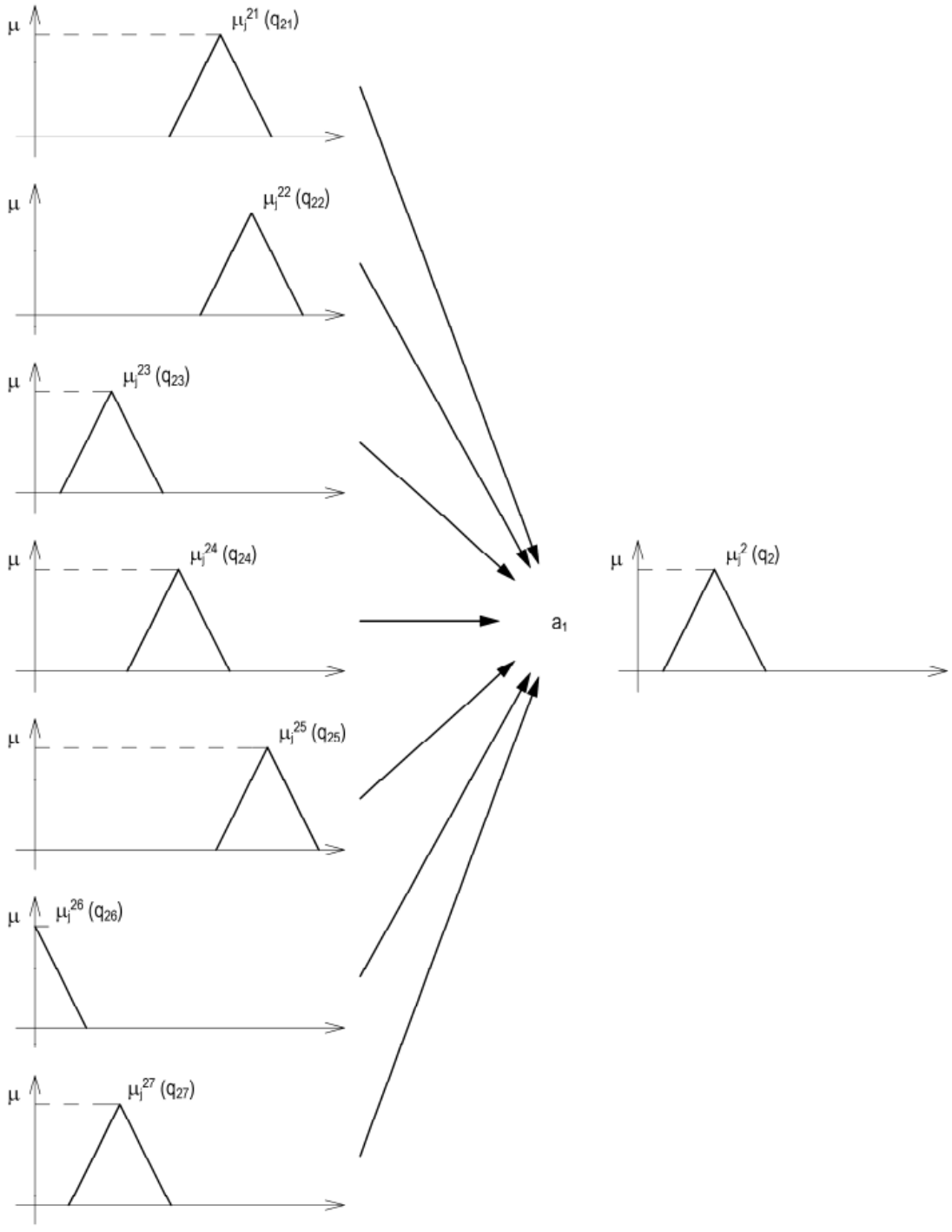
Ebben az eredményül kapott állapotgépen a költségek szerint optimális megoldás megtalálása már sok időt vehet igénybe. Ennek kivitelezésére további eszközök vonhatók be a modellbe, ilyen lehet például heurisztikák vagy populáció alapú optimalizálási algoritmusok [9] használata, mely jelentősen meggyorsíthatja az optimális (vagy kvázi optimális) útvonal megtalálásának futását.



6. ábra – Tetőfelújítás folyamatának modellezése egy szignatúra több állapotgépével [4]

A 6. ábra szemlélteti egy részkomponens felújítási folyamatának struktúráit a szignatúra egy szintjén és a folyamatok lépéseinek sorozatát a különböző fuzzy állapotgépeken belül. Mivel a fuzzy állapotgépek állapotai fuzzy halmazokhoz rendeltek, ezért a szignatúra leveleiben egy állapotgépbeli állapot meghatározza a halmazt is

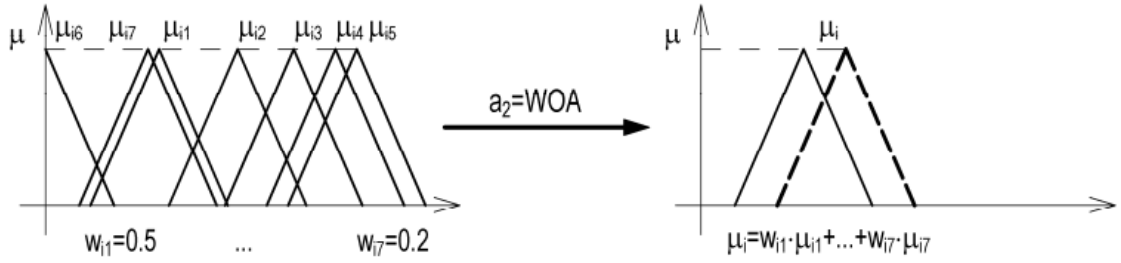
egyben. A 7. ábra szemlélteti a fuzzy állapotgép csúcsai alapján kiválasztott fuzzy halmazok aggregációját, mely így meghatározza a szülő értékét.



7. ábra – Aggregációs operátor alkalmazása fuzzy halmazok felett [4]

A szignatúra felépítése csúcsonként engedi a különböző aggregációs eljárások megválasztását. Megemlítendő, hogy az aggregációs eljárás megválasztása nagy hatással van a végső eredményre. Javasolt a súlyozott aggregációs megoldások használata (pl.

WOA[4]), melyek lehetőséget adnak arra, hogy a súlyok módosításával beállítható legyen a lingvisztikai érték megfelelő közelítése (8. ábra).



8. ábra – Súlyokkal történő aggergáció [4]

2.3 Optimális megoldás keresése

Az NP-nehéz[5] problémák megoldásainak (vagy közelítő megoldásainak) minél gyorsabb és hatékonyabb módszerekkel történő megtalálásának hatalmas szerepe van a gyakorlati problémák költséghatékony megoldásainak előállítása szempontjából.

A gyakorlatban felmerülő problémák egy része bár hagyományos matematikai eszközökkel ugyan leírható, de optimális megoldásaik megtalálásához (a nagyon speciális esetektől eltekintve) a bemenet függvényében gyakran exponenciális mértékű lépés vagy tárkapacitás szükséges. Azaz e problémák gyakran nem oldhatók meg polinom időben. A problémát leginkább az Utazóügynök probléma (TSP)[6] szemlélteti, mely jelenleg is az aktívan kutatott kombinatorikus optimalizálási problémák közé tartozik.

2.3.1 Az utazóügynök probléma

A TSP-n értelmezett feladat célja egy optimális útvonal/élsorozat/kör megtalálása egy olyan (irányított, vagy irányítatlan) gráfban, amelynek élei súlyozottak és ahol a gráf összes csúcsát érinti a bejárás (Hamilton-út/kör keresése). Az optimum meghatározása többféle szempontból értelmezhető. Amennyiben a súlyok valamiféle költségeket reprezentálnak, úgy minimalizálási probléma megoldását keressük, amennyiben pedig a

[4] WOA – Weighted Ordered Aggregation

[5] NP-hard: Non deterministic Polynomial-time hard – egy probléma NP-nehéz ha minden NP-beli probléma redukálható az adott problémára, amely alapján az eredeti probléma legalább annyira nehéz, mint bármelyik NP-beli probléma. Jelen ismeretek szerint egzakt algoritmusokkal nem oldhatók meg polinom időben.

[6] TSP – Travelling Salesman Problem

súlyok hasznosságokat írnak le, akkor a maximalizálási probléma megoldását érdemes felderíteni.

A fentiek alapján TSP definiálható úgy, mint egy gráfban történő keresés problémája súlyozott élekkel:

$$G_{TSP} = (V_{cities}, E_{conn}) \tag{18}$$

$$V_{cities} = \{v_1, v_2, \dots, v_n\}, E_{conn} \subseteq \{(v_i, v_j) \mid i \neq j\}$$

$$C: V_{cities} \times V_{cities} \to R, C = (c_{ij})_{n \times n}$$

ahol *C* a költség mátrix és c_{ij} reprezentálja az *i* csúcsból *j* csúcsba történő eljutás költségét. A cél a legrövidebb költségű kör megtalálása a gráfban, amely minden csúcsot tartalmaz, melyre igaz, hogy leírása felírható csúcsok valamilyen permutációjával: *(p1, p2, ..., pN)* és minimalizálja a teljes költséget:

$$C(i) = \left(\sum_{i=1}^{n-1} c_{p_i, p_{i+1}} \right) + c_{p_n, p_1} \tag{19}$$

A költségmátrixtól függően a TSP probléma lehet szimmetrikus (amennyiben $c_i = c_j$) és aszimmetrikus (ha $c_i \neq c_j$).

A TSP probléma NP-nehéz, így az optimális megoldás megtalálása (az összes lehetséges megoldás figyelembevételével) a bemenet méretének exponenciális függvénye. Azaz nagyon kis bemenetekre még könnyen, egyszerű algoritmussal is megtalálható a megoldás, azonban a bemenetek számának fokozatos növekedésével (már viszonylag kis növekedés után is) az exponenciális robbanásnak köszönhetően könnyen kezelhetetlenné válhat a feladat optimális megoldásának belátható időn belül történő megtalálása.

Egy szimmetrikus TSP probléma is $(n-1)!/2$ lehetséges megoldást tartalmaz, míg az asszimmetrikus TSP probléma $(n-1)!$-et. Már 15 csúcs esetén is a lehetséges megoldások száma eléri a 10^{10}-t, így az összes megoldás kiértékelésével végzett keresés nem kivitelezhető valós időben.

A TSP-nek számos lehetséges alkalmazási területe létezik, többek között a logisztika, a tervezés, a mikrochipek gyártása, de számos területen merül fel részproblémaként is, pl. génszekvenálás stb. Utóbbi esetén a csúcsok DNS töredékeket reprezentálnak és a távolság pedig a DNS töredékek közötti hasonlóság mértéke. [10][11]

2.3.2 Evolúciós optimalizálás

Számos gyakorlatban felmerülő diszkrét optimalizási probléma NP-nehéz. A számítástechnika fejlődésével a meglevő hatékony algoritmusok mellett (pl. Lin-Keringhan heurisztika, Concorde algoritmus stb.) számos evolúciós vagy más populáció alapú megoldás is kidolgozásra került, melyek képesek közel-optimális megoldás megtalálására rövid időn belül viszonylag nagy problématerek felett.

Az **evolúciós algoritmusok** sztochasztikus kereső eljárások körébe tartoznak, melyek a Darwini alapelvekre építve az optimalizációs probléma lehetséges megoldásainak egy populációját tartják nyilván és iterációnként módosítják azokat. A populáció egyedei az eredeti probléma egy-egy megoldását kódolják, melyek mindegyikéhez a probléma optimális megoldásához konvergáló „jóság érték" társítható. Az evolúciós algoritmusok iteratívan változtatják a populáció összetételét, miközben speciális operátorokkal biztosítják, hogy populáció egyedei iterációról iterációra fejlődjenek, és környezetükhöz igazodva az optimális megoldás egyre jobb közelítését adják. Az evolúciós algoritmusokra jellemző, hogy hatékony alkalmazásukhoz kritikus lépés az egyedek kódolásának helyes megválasztása, valamint az egyedekhez a probléma optimális megoldása szempontjából jóság mértéket rendelő fitnesz függvény jó megválasztása. [18] A fitnesz függvény fontos tulajdonsága, hogy olyan kimeneteket rendeljenek az egyes egyedekhez, amelyek alapján azok jóság szerint egyértelműen sorbarendezhetőek lesznek.

A **memetikus algoritmusok** a hagyományos populáció alapú algoritmusok kiterjesztései, melyek a populáció minőségének javítása érdekében minden iterációban heurisztikákat vagy lokális kereső eljárásokat alkalmaznak.

Mivel az evolúciós optimalizálási módszerek skálája nagyon széles, korábbi eredményekre alapozva leginkább a **Bakteriális Memetikus Algoritmusok** (BMA[7]) diszkrét problémák felett értelmezett változata került tanulmányozásra, amely a Bakteriális Evolúciós Algoritmusok (BEA[8]) családjába tartozik, de megjegyzendő, hogy más típusú evolúciós algoritmusok (pl. evolúciós stratégiák, genetikus algoritmusok, evolúciós programozás, genetikus programozás stb.), vagy részecske-raj alapú

[7] BMA – Bacterial Memetic Algorithm

[8] BEA – Bacterial Evolutionary Algorithm

algoritmusok is kisebb módosításokkal a fenti problémára illeszthetők lettek volna. [17][18] A kiválasztás oka az, hogy egyes cikkek szerint a bakteriális evolúciós algoritmusoknál tapasztalt konvergencia gyorsasága felülmúlja a többi felsorolt megoldását. [11][10]

Bár mind az evolúciós, mind pedig a memetikus algoritmusok kereső eljárások, eltérő tulajdonságokkal rendelkeznek. Az evolúciós algoritmusok globális kereső eljárások, melyek általában lassabb konvergencia sebesség mellett optimumhoz közeli megoldásokat adnak. A lokális kereső eljárások a lokális információt felhasználva gyorsan képesek elérni a legközelebbi lokális optimumot, azonban a feladat egészére nézve nem hatékonyak. Mivel a memetikus algoritmusok az evolúciós algoritmusok globális kereső eljárásait lokális kereső eljárásokkal egészítik ki, így ezek kombinációját megvalósítva nagyobb konvergencia sebességet képesek elérni.

2.3.3 Bakteriális Memetikus Algoritmusok (BMA)

A Bakteriális Memetikus Algoritmusok [10][11] fontos tulajdonsága, hogy speciális operátorokkal iterációnként javítja a populáció összetételének minőségét, ezzel előállítva olyan egyedeket, amelyek minőségükben egyre inkább közelednek az optimális megoldáshoz.

A populáció összetételének javítása végett az egyedeken (melyek a feladat egy lehetséges megoldásait kódolják) előbb speciális, az evolúciós szempontból értelmezett operátorok kerülnek végrehajtásra, majd az operátorok végrehajtása után lokális kereső eljárásokkal az egyedek minőségének további javítása történik.

Ilyen speciális operátorok például a **bakteriális mutáció** és a **gén transzfer**. A **lokális kereső eljárások** típusa nagyban függ a probléma jellegétől. Folytonos típusú problémák esetén gradiens alapú módszerekkel kombinálható a lokális minimum keresése (Levenberg-Marquardt eljárás). Diszkrét típusú problémák esetén a hibafüggvény nem folytonos tulajdonsága miatt lokális keresőeljárásként nem alkalmazhatóak a gradiens alapú megoldások, ezért helyettük a későbbiekben bemutatandó 2-opt és 3-opt lokális keresések kerülnek alkalmazásra.

Mivel az előzőekben bemutatott épületfelújítási munkálatok folyamatának modellezése állapotgépek felhasználásával történik, ezért egy olyan evolúciós optimalizációs eljárás megválasztása szükséges, amely képes operálni a diszkrét elemekkel (vagyis a fuzzy- állapotgép csúcsaival). Ennek a követelménynek megfelelő

algoritmus a **Diszkrét Bakteriális Memetikus Evolúciós Algoritmus** (DBMEA[9]), amely a Bakteriális Evolúciós Algoritmuson (BEA) alapul, kiegészítve azt a lokális kereső eljárással annak minden egyes iterációjában. A lokális keresési lépéssel történő kiegészítés a tapasztalatok szerint jelentősen gyorsíthatja az optimális megoldás megtalálásának konvergencia sebességét.

2.3.4 Bakteriális Evolúciós Algoritmus (BEA)

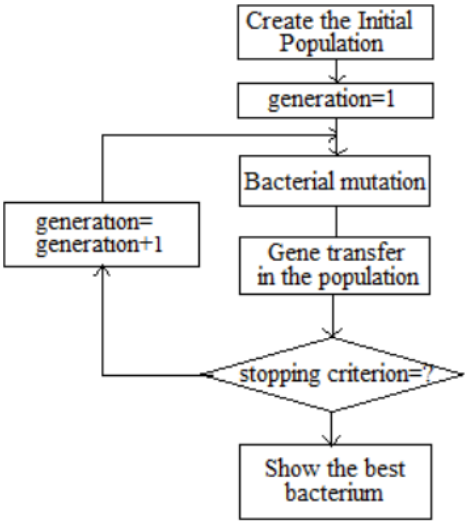
Ezen algoritmusok alapötlete azon a biológiai jelenségen alapul, mely szerint egy „hím" baktérium DNS információkat adhat át egy „nőstény" baktériumnak a párosodás során, ezáltal a nőstény baktériumok új tulajdonságokra tehetnek szert. Amennyiben a kapott génszekvenciák által kódolt információk hasznosak a nőstény számára, az továbbadhatja a megszerzett tulajdonságokat egy következő iterációban, így egy baktérium jellemző tulajdonságai könnyen elterjedhetnek a populáción belül.

Elsőként egy kezdeti populáció kerül létrehozásra N egyedszámmal. Ezen egyedek bár a feladat egy-egy lehetséges megoldásait reprezentálják, azonban csak nagyon kis valószínűséggel oldják meg optimálisan a problémát.

Ezt követően a bakteriális evolúciós algoritmus két operátort felhasználva (bakteriális mutáció és gén transzfer) iterációról iterációra javítja az egyedek minőségét (szemben a genetikus algoritmusoknál alkalmazott három operátorral, melyek a mutáció, kereszteződés és szelekció). Míg a bakteriális mutáció az egyénenkénti optimalizációt hivatott megvalósítani, addig a gén transzfer a baktériumok közötti információcserét valósítja meg.

Ez a folyamat addig ismétlődik, amíg egy leállási feltétel nem teljesül. Lehetséges leállási feltétel lehet, hogy az algoritmus átlépte az előre megadott maximális iterációszámot, vagy az, ha a populáció legjobb egyedei által biztosított legjobb megoldások fitneszeinek különbsége az utolsó N lépésben valamilyen ε mérték alá csökkent. [18]

[9] DBMEA – Discrete Bacterial Memetic Evolutionary Algorithm

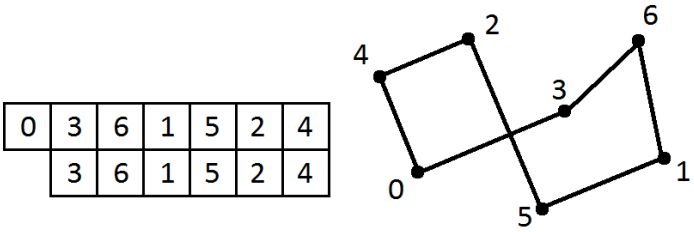


9. ábra - BEA operátorainak folyamatábrája [10]

A 9. ábrán látható a BEA algoritmus vázlatos flowchartja[10] és műveleteinek végrehajtási sorrendje.

2.3.4.1 Egyedek kódolása

Az egyedek kódolásának megválasztása kritikus jelentőséggel bír a feladat megoldásának szempontjából. Egy jó elkódolás korlátozhatja a keresési teret, kizárva a keresés szempontjából nem releváns megoldásokat, míg egy rossz kódolás akár el is lehetetlenítheti az optimumpont megtalálását.



10. ábra - Bejárás kódolása a kromoszómában

A TSP problémáknál DBMEA esetén alkalmazható legegyszerűbb kódolás a csúcsok és azok indexei között (*0…n-1*) megfeleltetett bijektív[11] leképezés megvalósítása, ahol *n* a gráf csúcsainak a száma. Az egyedet leíró kromoszóma[12] szekvencia ezáltal az

[10] Flowchart – vezérlést bemutató folyamatábra

[11] Bijektív leképezés – kölcsönösen egyértelmű leképezés, egyben injektív és szürjektív is

[12] Kromoszóma – a feladatot az egyedben ténylegesen kódoló „génszekvencia”

indexek egy permutációját határozza meg, mely sorrend egyértelműen megfeleltethető a csúcsok bejárás szerinti sorrendjének a gráfban. Ha minden meglátogatott csúcs egyszer kerül bejárásra, akkor szintén csak egyszer jelenik meg a génszekvenciában. Ha a bejárás kezdőpontja ismert, akkor elhagyható, ezzel csökkentve a problématér méretét. (10. ábra)

2.3.4.2 Kezdőpopuláció létrehozása

A kezdeti populáció kialakítása alapvetően befolyásolhatja a közel-optimális megoldás megtalálásának sebességét. A populáció kialakításakor érdemes törekedni a problématér uniform[13] lefedésére, valamint az egyedek heterogenitásának[14] maximalizálására. Lehetséges stratégiák az egyedek kialakításakor:

- **Random** – A kromoszómák véletlenszerűen sorsolt csúcsok indexeivel kerülnek feltöltésre.
- **Nearest Neighbour (NN)** – Egy olyan bejárási sorrendet kódol, melynél mindig a legkisebb költséggel elérhető csúcs kerül kiválasztásra.
- **Secondary Nearest Neighbour (SNN)** – Egy olyan bejárási sorrendet kódol, melynél mindig a második legkisebb költséggel elérhető csúcs kerül kiválasztásra.
- **Alternating Nearest Neighbour (ANN)** – Az NN és az SNN ötvözése, mely egy olyan bejárást kódol, melynél a legkisebb költséggel és a második legkisebb költséggel elérhető csúcsok váltakozó sorrendben kerülnek kiválasztásra.

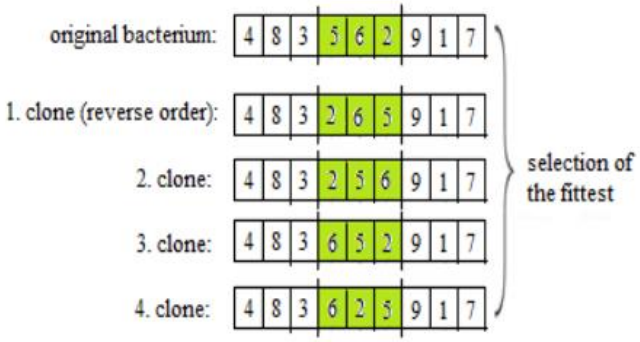
A fenti stratégiák mindegyikében a kromoszómák hossza megegyezik a gráf csúcsainak számával. A véletlenszerű csúcs kiválasztás biztosítja a populáció egyedeinek egyenletes eloszlását a keresési térben. A kezdőpopuláció egyedeit a következő stratégia alapján javasolt kialakítani: K-3 darab egyed véletlenszerűen létrehozott kromoszómával és 1-1-1 darab egyed az NN, SN és ANN stratégiák segítségével.

[13] Problématér uniform lefedése – minél egyenletesebben van lefedve a problématér a populáció által, általában annál gyorsabb konvergencia sebességet várunk el

[14] Heterogenitás maximalizálása – minél különbözőbbek az egyedek, általában annál gyorsabb konvergencia sebességet várunk el

2.3.4.3 Bakteriális mutáció operátor

A bakteriális mutáció operátor az egyedeket kizárólag önmaguk felhasználásával kísérli meg javítani azáltal, hogy a populáció minden egyedéhez N darab másolatot hoz létre, majd egy adott géncsoportot véletlenszerűen kiválasztva a másolt egyedeket a kiválasztott géncsoport elemein belül véletlenszerűen módosítja, miközben az eredeti egyed nem kerül módosításra. E kiválasztott géncsoportok kromoszómán belül lehetnek szomszédosak, vagy lazán kapcsolódóak, ez alapján megkülönböztethető a bakteriális mutáció operátor két típusa: **coherent segment mutation** vagy **loose segment mutation**.



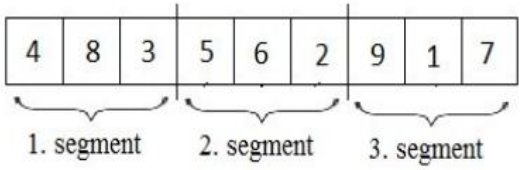
11. ábra - Mutáció operátor működése [10]

A véletlenszerűen mutált egyedek mellett egy másolat determinisztikusan kerül létrehozásra, amely az adott iterációban kiválasztott géncsoport által lefedett értékek egyszerű megfordítását kódolja (11. ábra).

Ezt követően az egy egyedből készített összes másolat kiértékelésre kerül és amennyiben valamelyik másolat esetében jobb eredményhez jutunk, mint ami az eredeti egyed értékelése volt, úgy az eredeti, valamint az összes másolt egyed ugyanezen kiválasztott géncsoport elemeibe a fitneszt növelő előnyös változások visszaírásra kerülnek a legjobb példányból. Az algoritmus mindezt addig ismétli, amíg egy egyed összes kiválasztható géncsoportja át nem esett az optimalizáción, ekkor az egyedhez tartozó másolatok eldobásra kerülnek. Az eredményként előálló egyed legalább olyan jó megoldása a problémának, mint amilyen az eredeti egyed volt. A bakteriális mutáció operátor ezután a populáció összes egyedén végrehajtásra kerül.

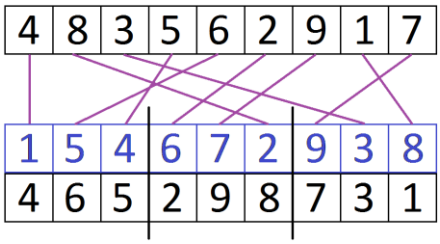
A kromoszómák tipikusan fix méretű géncsoportokra oszthatók fel, a fentebb leírt véletlenszerű módosítások egy géncsoportra vonatkozó iterációban e szegmenseken belül kerülnek végrehajtásra.

Amennyiben a kiválasztott géncsoportok a kromoszómán belül szomszédosak, koherens szegmenseknek nevezzük azokat (12. ábra). Ezek előnye, hogy a szegmensek tartalma egyszerűen kiválasztható a szegmenshatárok alapján.



12. ábra - Koherens szegmensek [10]

Amennyiben a kiválasztott géncsoportok a kromoszómán belül egymással nem szomszédosak, akkor azokat laza szegmenseknek nevezzük. Ezek előnye, hogy egyszerűen képezhető egy olyan indexpermutáció, amely mentén az eredeti kromoszóma elemei egy új sorrendbe állíthatók. A 13. ábrán látható, ahogy egy új indexpermutáció véletlenszerű létrehozásával (kék színű számok) az eredeti kromoszóma elemeinek felhasználásával egy új példány kerül létrehozásra.



13. ábra - Laza szegmensek képzése index permutációval

Az általam áttekintett szakirodalom azokat az eseteket szemlélteti, amelyeknél a szegmensméretek osztói a kromoszóma méretének. Azokra az esetre sajnos nem tér ki, hogy hogyan érdemes elvégezni a bakteriális mutáció operátort akkor, amikor a szegmensméret nem osztója a kromoszóma méretének, tehát a kromoszómán olyan maradék is képződik, melyekre már nem alkalmazható a bakteriális mutáció az eredeti szegmensmérettel.

A bakteriális mutáció operátor egy iteráción belüli alkalmazásának időbeli komplexitása (ahol N_{pop} a populáció méretét jelöli, N_{clones} a mutáció során létrehozott klónok számát, n pedig a kromoszóma hosszát) $O(N_{pop}N_{clones}n^2)$, tárigénye $O(N_{pop}N_{clones}n)$.

2.3.4.4 Gén transzfer operátor

A gén transzfer operátor biztosítja a populáción belüli információáramlást a baktériumok között annak reményében, hogy a problémát kevésbé jól megoldó baktériumok kromoszómáiba a problémát jobban megoldó baktériumokból olyan információ kerül át, mely segíti a gyengébb baktériumokat a probléma megoldásában.

Mindezt úgy próbálja elérni, hogy a populáció összes egyedét kiértékeli, majd a fitnesz értékük szerint sorba rendezi azokat. Ezt követően a baktériumokat fitneszeik szerint sorolja be a jobban és a kevésbé jól teljesítő csoportok egyikébe. Ezután N-szer megismétli, hogy kiválaszt egy baktériumot a jól teljesítők közül (source) és egyet a rosszul teljesítők közül (target/destination), majd néhány véletlenszerűen kiválasztott, előre definiált hosszúságú szegmenst átmásol a jól teljesítő baktériumból a rosszul teljesítőbe.

A DBMEA gén transzfer operátora némileg eltér a BEA által alkalmazott gén transzfer operátortól (lásd: következő alfejezet).

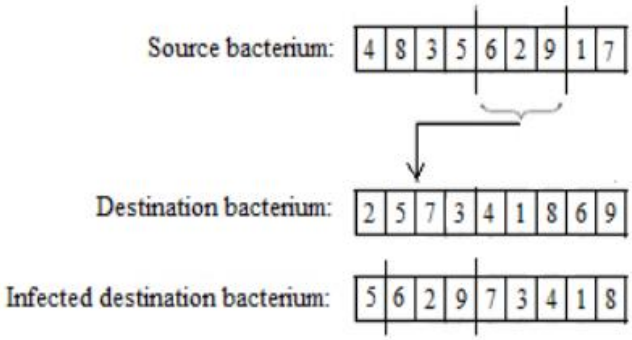
2.3.5 Diszkrét Bakteriális Memetikus Evolúciós Algoritmus (DBMEA)

Ezen algoritmus a Bakteriális Evolúciós Algoritmus lokális kereséssel történő kiterjesztése. Mivel egyben memetikus eljárásról van szó, ezért definíció szerint szükséges a globális kereső eljárást megvalósító evolúciós algoritmus megfelelő lépései közé a megfelelő lokális kereső eljárások beiktatása.

2.3.5.1 Gén transzfer operátor

Az előző alfejezetben tárgyalt BEA gén transzfer operátorához képest a DBMEA gén transzfer operátora a következő működést valósítja meg: a jobb eredményt nyújtó baktériumból véletlenszerűen kiválasztásra kerül egy előre definiált hosszúságú szegmens, amely átkerül a gyengébben teljesítő baktérium egy véletlenszerű pozíciójába. A célbaktériumban az eltolás mértéke véletlenszerűen kerül kiválasztásra és nem feltétlenül esik egybe az eredeti baktériumbeli eltolás mértékével (14. ábra).

A beszúrást követően a célkromoszóma hossza azonban szükségszerűen változatlan kell maradjon, ez ugyanis követelménye a kromoszómák TSP feladaton történő kiértékelhetőségnek. Ezért a beszúrást megelőzően a beszúrandó szegmens által tartalmazott elemek eltávolítása szükséges a célkromoszómából.



14. ábra - Gén transzfer DBMEA esetén [10]

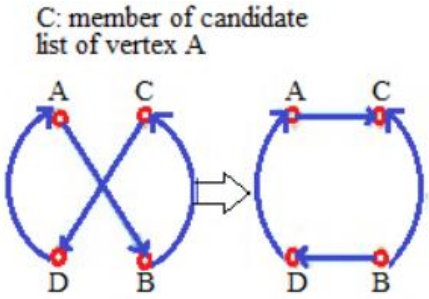
A gén transzfer operátor alkalmazásának lépésszáma a következő (ahol n jelöli a kromoszóma hosszát, N_{pop} a populáció méretét és N_{iter} a gén transzfer operátor során alkalmazott iterációk számát): az operátor alkalmazásának legelején az összes egyedre történő fitnesz meghatározása $O(N_{pop}n)$, a fitnesz szerinti csökkenő sorrendbe rendezés $O\left(N_{pop}\log\left(N_{pop}\right)\right)$, az újonnan létrehozott baktérium populációba történő beszúrását követően az új fitnesz érték kiszámításának lépésszáma $O(N_{iter}(n+N_{pop}))$. Az operátor összesített lépésszáma ezáltal $C_{GT} = O(N_{pop}(n+\log N_{pop}) + N_{iter}(n+N_{pop}))$

2.3.5.2 A 2-opt és 3-opt lokális kereső eljárások

Mivel a gradiens alapú lokális kereső eljárásoknak nincs értelme diszkrét típusú feladatok terében (mivel a hibafüggvény nem folytonos, így a gradiens sem állapítható meg), ezért a diszkrét típusú feladat jellegére jobban illeszkedő lokális keresési eljárás alkalmazása történik. Ilyen lokális kereső eljárás például a 2-opt és a 3-opt, melyek közös jellemzője, hogy egy jelöltből kiindulva determinisztikus módszerek alkalmazásával iteratívan képes eljutni egy, a szomszédossági feltételeknek jobban megfelelő megoldáshoz.

2.3.5.3 A 2-opt lokális keresés

A 2-opt egy tetszőleges kromoszómából kiindulva úgy hoz létre a szomszédossági feltételeknek jobban megfelelő kromoszómát, hogy minden a gráfból iteratívan vett (AB, CD) párra megvizsgálja az |AB|+|CD|>|AC|+|BD| egyenlőtlenséget, amely teljesülése esetén törli a gráfból az AB és CD éleket és helyettük a kisebb költségű AC és BD éleket szúrja be (15. ábra). Az algoritmus mindezt addig ismétli, amíg már nem lehetséges további javulás.



15. ábra - A gráf bejárásának sorrendje 2-opt alkalmazása előtt és után [10]

Az egyenlőtlenség teljesülésének vizsgálatakor az algoritmus nem vizsgálja meg a gráf minden csúcsát az aktuális C-be való helyettesítéshez, csupán az aktuális A csúcs kimeneteit tekinti jelöltnek.

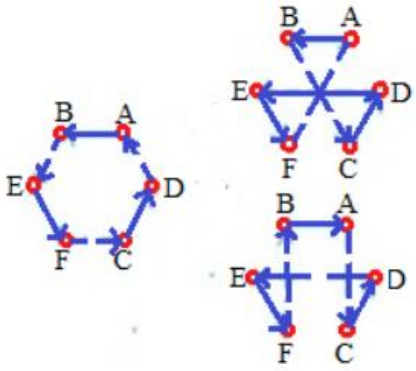
Az algoritmus metrikus TSP-n működőképes. Futásideje naiv tömb alapú implementációval $O(2!\,n^2)$, ahol n a gráf csúcsainak száma.

A gráf egy bejárását 2-optimálisnak tekintjük, ha a bejárás már nem javítható tovább a 2-opt lokális keresés segítségével.

2.3.5.4 A 3-opt lokális keresés

A 2-opt eljárással szemben a 3-opt eljárás egy hosszabb élsorozaton próbál meg javulást elérni azáltal, hogy három él törlésével és helyettük kisebb költségű élek felhasználásával kapható megoldások közül választja ki az optimálisat.

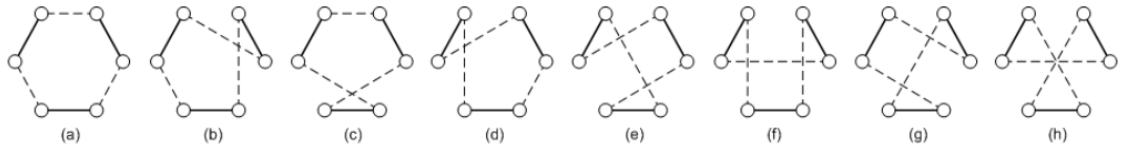
Három él törlése esetén az eredeti bejárási sorrend megtartása mellett a gráfból három különböző módon képezhető az új bejárási sorrend (melyből egy az eredeti bejárásnak felel meg) (16. ábra). [10][11]



16. ábra – A gráf bejárásának lehetséges sorrendjei 3-opt alkalmazása esetén [10]

Egy másik forrás alapján 3-opt esetén valójában 8 különböző konfiguráció keletkezik (ugyanis k él törlése esetén az élek $(k-1)!\,2^{k-1}$ lehetséges módon köthetők

újra), amelyek közül 4 (a 17. ábra *a, b, c, d* megoldásai) olyan élekből vannak képezve, amelyek kiválasztásra kerülnek a 2-opt futtatása során is, a maradék 4 konfiguráció (a 17. ábra *e, f, g, h* megoldásai) pedig a 2-opt által ki nem választott éleket is tartalmazza [20].



17. ábra - A 3-opt törölt éleihez tartozó bekötések összes lehetséges konfigurációja [20]

Az algoritmus metrikus TSP-n működőképes. Futásideje naiv tömb alapú implementációval $O(3!\,n^3)$, ahol n a gráf csúcsainak száma.

A gráf egy bejárását 3-optimálisnak tekintjük, ha a bejárás már nem javítható tovább a 3-opt lokális kereséssel. Ha egy bejárás 3-optimális, akkor egyben 2-optimális is [19].

2.3.6 Evolúciós algoritmusok problémái

Bár a evolúciós algoritmusok viszonylag gyorsan képesek optimum-közeli megoldást szolgáltatni, természetükből fakadóan sajnos érzékenyek lehetnek a kezdőfeltételekre, melyek nagyban befolyásolják működésük hatékonyságát.

Ilyen kezdőfeltételek például az egyedek heterogenitásának biztosítása, azaz a kezdő populáció uniform eloszlása a probléma megoldásainak terében, továbbá az egyedek megfelelő kódolásának megválasztása, valamint olyan hiperparaméterek beállítása, mint például az iterációszám, a leállási feltétel, az egyes operátorokban alkalmazott, szegmensméret stb. Bár az egyedek uniform eloszlása viszonylag könnyen garantálható és a megfelelő kódolás kiválasztása is többnyire jól kivitelezhető, az optimális hiperparaméterek megtalálása gyakran vagy nagyon nehéz, vagy módja egyáltalán nem ismert, ezért ezek többnyire tapasztalati úton kerülnek kikísérletezésre és megtalálásra egy konkrét probléma esetén. [17][18]

2.3.7 További evolúciós algoritmusok

Bár e dolgozatban külön nem kerül kifejtésre, azért fontos megjegyezni, hogy a korábbiakban bemutatott BEA, valamint BMA algoritmusokon túl további, e családba tartozó algoritmusok esetleg hatékonyabban oldhatják meg a felmerülő optimalizáicós feladatot.

Ezen algoritmusok például a következők lehetnek: Improved Bacterial Memetic Algorithm (IBMA), Bacterial Memetic Algorithm with the Modified Operation Execution Order (BMAM), Modified Bacterial Memetic Algorithm (MBMA) vagy Progressive Bacterial Algorithm (PBA) [9]. Ezen algoritmusok tulajdonságai, hogy egyes műveleteket más sorrendben vagy eltérő mennyiségben hajtanak végre, esetleg jobban figyelembe vesznek sorrendfüggéseket a kromoszóma elemei között, ezáltal javítva a konvergencia sebességét az optimális megoldáshoz.

2.4 Fenti modelleket megvalósító megoldások

A piacon jelenleg kapható szoftver alapú megoldások az épületfejlesztési feladatok széles skáláját fedik le, sajnos azonban nem sikerült olyan megoldást találni, amely nyilvánosan a fenti matematikai megoldások összességét együttesen modellként alkalmazná.

Mivel ezen dolgozat elsődleges célja az előzőekben bemutatott matematikai modellek kipróbálása, gyakorlati problémához történő illesztése, valamint egy „proof of concept”[15] jellegű megoldás kidolgozása, ezért a továbbiakban bemutatásra kerül, hogy hogyan képzelhető el a fenti modelleken alapuló megoldást nyújtó szoftver egy lehetséges megvalósítása.

[15] Proof of Concept – A működőképesség és koncepció igazolása, valamint annak bemutatása. Nem feltétlenül egy végleges, a működést teljességében lefedő megoldás kialakítása.

3 Követelmények összefoglalása

E fejezetben azok a felhasználói igények és gyakorlati megfontolások kerülnek összefoglalásra, amelyeket az épületfelújítások szakterületén dolgozó szakemberek támaszthatnak egy a munkájukat segítő alkalmazás irányába.

Továbbá összefoglalásra kerül néhány olyan technológia és technikai követelmény is, amelyek segítségével a modern szoftvertervezési és szoftverfejlesztési elveknek megfelelő alkalmazás fejleszthető.

3.1 Alkalmazás követelményeinek specifikálása

Mivel a kiválasztott szakterület problémáinak a korábbi fejezetben vázolt modelleken keresztül történő megoldásaira irányuló szoftver szükségszerűen a fenti modellekre épít, ezért a szoftverben oly módon kell kialakítani az épületfelújítási szakterülethez illeszthető belső adatszerkezeteket és modelleket, hogy azok kompatibilisek legyenek a fuzzy modelleket kiértékelni képes számítási rendszerrel. Emiatt a modellek szerepe az alkalmazásban kettős, egyrészt a számítási rendszer modelljeihez illeszkednek, másrészt meg kell felelniük a szakterület igényeinek. Ahogy a későbbiekben látható lesz, a modellek felületre történő kötéséhez újabb feltételek teljesülésére lesz szükség, melyet majd a grafikus keretrendszer határoz meg.

A szoftvernek lehetővé kell tenni az épületek állapotainak fuzzy modell alapú leírásainak (ide értve a fuzzy halmazok, fuzzy állapotgépek és fuzzy szignatúrák) megadását, szerkesztését, betöltését, böngészését, elmentését, valamint esetleges csoportos kiértékelését.

A szoftver lehetővé teszi a munkafolyamatok költségoptimalizációját és kiterjesztési lehetőségeket ad a következőkre:

- többcélú optimalizáció (pl. pénzbeli, munkaszervezési, jogi, technikai stb.) támogatása
- bonyolultabb heurisztikák (vagy más, pl. evolúciós algoritmusok segítségével) az optimalizáció folyamata gyorsabb és hatékonyabb legyen

A szoftver (főként kényelmi okok miatt) támogatja az újra felhasználható fuzzy modell komponensek (továbbiakban modell sablonok, vagy fuzzy modell és épület

típusok) kialakítását (melyek egy adott űrlap-típus struktúrájának vagy épület-típus felépítésének felelnek meg), és ahol lehet, elkülöníti egy épület típus leíró modelljét a konkrét épületet értékekkel leíró modelljétől.

A szoftver támogatja az azonos leíró modellel rendelkező épületek kiértékelését és összehasonlítását, valamint kiterjesztési lehetőséget biztosít arra, hogy későbbiekben az akár az egymástól nagyon eltérő leíró modellel rendelkező épületek (melyek az aggregációs operátorok használatán is túlnyúló transzformációkat igényelnek) is kiértékelhetők és összehasonlíthatók legyenek, például fatranszformációk, vagy szignatúrák felett definiált kölcsönösen egyértelmű leképezések útján.

A szoftver biztosítja a létrehozott modellek strukturális és tartalom szerinti validációját, valamint ellenőrzi az „összeszerkeszthetőségi kritériumokat” (például egy szignatúra nem tartalmaz-e olyan állapotgépeket leveleikben, amelyek eltérő dimenziójú költségvektor felett értelmezettek).

Az alkalmazás inputja/outputja (ahol lehetséges) legyen manuálisan is könnyen szerkeszthető és értelmezhető, de az alkalmazás rendelkezzen grafikus felülettel, valamint az épület struktúrát leíró szignatúra, valamint a felújítási folyamatot modellező állapotgép is magyarázó jelleggel képesek legyenek címkék, verbális megjegyzések és képek tárolására/megjelenítésére.

A fentiek célja, hogy a felhasználó definiált sablon típusokat megfelelően „összeszerkesztve”, majd az ezek felé épülő épület típusokat megfelelően „kitöltve” könnyen hozhasson létre a konkrét épületekre jellemző leírásokat (továbbiakban épület modelleket). E leírásokat azonos típusú épület modellekkel összehasonlíthassa, vagy adott célnak megfelelően tömegesen kiértékelhesse. Továbbá legyen biztosított a lehetőség a különböző állapotban levő épületek felújításainak költségbecslésére úgy, hogy paraméterezhető legyen az elvárt állapot minősége.

A különböző típusú épület modelleket (amennyiben azok valójában ugyanazt az információt reprezentálják, csak eltérő struktúrában) pedig a felhasználó egy másik típusra illeszkedő modellel összehasonlíthassa, vagy vele együtt közösen kiértékelhesse, esetleg az információkat egy másik típusra transzformálhassa.

Továbbá a szoftver képes legyen metaadatok (pl. szöveges leírások, címek stb.), valamint képi információk (pl. fotók, egyszerűbb ábrák stb.) tárolására és

megjelenítésére, melyek segítségével könnyebben beazonosítható, és áttekinthető a modellhez tartozó konkrét épület és annak részletei.

A fentiek alapján arra a következtetésre jutottam, hogy elsősorban egy olyan desktop[16] alkalmazás elkészítésére lenne szükség, amely a számítási kapacitás igénye miatt képes hatékonyan kihasználni a munkaállomások által nyújtott erőforrásokat. Amennyiben a későbbiekben felmerülne, hogy egy ilyen alkalmazás nem elég mobil, vagy a munkaállomás nem képes elegendő számítási kapacitást biztosítani a felmerülő terhelés kielégítéséhez, úgy megfontolható lenne egy kliens-szerver architektúrán alapuló rendszer kialakítása, amely mobil eszközökről használva komolyabb számítási rendszerek (grid[17] vagy cloud[18] alapú rendszerek) által nyújtott szolgáltatásokat venne igénybe.

3.2 Felhasználni kívánt technológiák áttekintése

Grafikus asztali alkalmazás fejlesztése lévén az Oracle Java platform által kínált megoldások igénybevétele mellett döntöttem. Bár itt rendelkezésre állnak már jól ismert technológiák *AWT, SWING, SWT* stb., azonban ezek a keretrendszerek régebbi paradigma alapján kerültek kialakításra, így helyenként fölöslegesen sok kódolást igényelhetnek

A Java 8-as kiadásában már (további könyvtárak telepítése nélkül) teljes mértékben elérhető **JavaFX** [21] keretrendszer lehetőséget biztosít a grafikus felület egy részének deklaratív[19] módon történő leírására FMXL nyelven (a Microsoft WPF-es XAML nyelvű megoldásához hasonlóan), amely egy az XML-re épülő felületleíró nyelv. Bár deklaratív módon a felület működése csak kis mértékben konfigurálható, megfelelő architektúrával (Model-View-Controller) és adatkötések kialakításával csökkenthető az imperatív[20] szemléletben programozandó felülettel kapcsolatos kódok mennyisége.

---

[16] Desktop alkalmazás – olyan hagyományos „asztali alkalmazás”, amely képes a legtöbb hagyományos munkaállomáson futni, annak erőforrásait kihasználni és nem feltétlenül igényli webszerver vagy más szolgáltatás meglétét

[17] Nagyobb számítási kapacitást, elosztottságot, párhuzamos számításokat biztosító rendszerek

[18] Számítási felhő által elfedett szolgáltatások gyűjteménye, melyek mögött több számítási kapacitás áll

[19] Deklaratív programozás – a „mit”-re fókuszálva követelményekben határozza meg a végrehajtást

[20] Imperatív programozás – a „hogyan”-ra fókuszálva lépésekben határozza meg a végrehajtást

A szoftverrel létrehozandó modellek letárolása során érdemes olyan adatformátumokat felhasználni, hogy segítsék a fejlesztés folyamatát, ez alapján viszonylag egyszerű eszközök igénybevételével könnyen böngészhetők és szerkeszthetők legyenek a letárolt adatok. A perzisztált feladatok betöltéséhez nem szükségszerű az adatbázisok használata, ezek telepítését sem a felhasználótól, sem az alkalmazástól elvárni nem célszerű.

A fentiek miatt a bináris vagy kódolt adatformátumok használata, valamint az objektumok tárolására képes adatbázisok használata is elvetésre került. A modellek exportálását és importálását egyszerű fájlműveletek segítségével érdemes kialakítani. Ilyen fájlformátumok például az XML (Extensible Markup Language), a JSON (Javascript Object Notation), a JPEG (Joint Photographic Experts Group), a PNG (Portable Network Graphics), az SVG (Scalable Vector Graphics) és a ZIP formátumok.

Az adatok perzisztálásához többek között felhasználhatók az **XStream** nevű osztálykönyvtár által nyújtott funkcionalitások, amelyek hasznos segítséget adhatnak a modell entitások annotálásához, ezzel megadva a modellek, valamint a XML és JSON formátumok közötti transzformációkat.

A tervezési szakaszban később bemutatott hierarchikus modellrendszeren alapuló komponensek szervezését és menedzselését az **OSGi Service Platform** [22] által nyújtott szolgáltatások kihasználásával érdemes megvalósítani abból a célból, hogy az elkészült modellek (legalábbis a fuzzy domain területhez kapcsolódóak) egy másik szakterületen való alkalmazásakor, a később bemutatandó alkalmazás keretrendszerén keresztül igény szerint dinamikusan betölthetők legyenek. Ennek következtében a létrehozott komponensek által tartalmazott típusok nemcsak jelen példaalkalmazásban, de egy másik szakterületre irányuló, esetleg ezeket átfogó összetettebb szoftverrendszer keretében is újrahasznosíthatók lennének, valamint megoldható lenne az is, hogy a fejlesztőnek csupán egyetlen szoftver fejlesztésével kelljen foglalkozni, miközben maga a szoftver egyszerre több szakterületre licenszelhető ugyanazzal a mögöttes számítási modellel (a megfelelő komponensek a licensz függvényében kerülnének szállításra, vagy betöltésre esetleg tiltásra az alkalmazáson belül). Ezáltal egyetlen kódbázis karbantartása elegendő lenne, remélhetőleg csökkentve annak fejlesztési költségeit, a minimális komplexitás növekedés mellett, miközben egyetlen kódbázis felhasználásával vagy különböző domainterülettel foglalkozó alkalmazáspéldányok jöhetnének létre, vagy egyetlen szoftver képes lenne a különböző domain területeket átfogni és licenszelési megoldáson

keresztül a felhasználók számára személyre szabottan biztosítani. Mivel az *OSGi* csupán csak szabványt definiál, ezért a fejlesztés során az elérhető implementációi közül (pl. *Apache Felix* [23], vagy *Equinox* [24]) szükséges a megfelelő kiválasztása.

Az alkalmazás fordításához és összeszerkesztéséhez valamilyen build keretrendszer használata javasolt. E platform felett elérhető megoldások például az *Ant* vagy a *Maven*. Bár már mindkettő kissé elavultnak számít, az utóbbit még széles körben alkalmazzák konfigurálhatósága miatt az XML alapú leírásának nehézségei ellenére. Emiatt azonban érdemes lehet egy újabb **Gradle** nevű build eszköz használata [26], amely amellett, hogy megfelel a modernebb kívánalmaknak, viszonylag könnyen konfigurálható és szkriptelhető *Groovy* nyelven a részben deklaratív kiterjesztési pontjainak és a Maven-re épülő konvencióinak köszönhetően.

A *JavaFX*, az *OSGi*, valamint a *Gradle* megfelelő összehangolása azonban nehézségeket állíthat a fejlesztő elé, ennek feloldására érdemes lehet egy inkompatibilitási problémákat feloldó keretrendszer használata. Erre biztosítanak lehetőséget a *e(fx)clipse [28]*, a *BndTools [25]*, valamint a viszonylag új *DromblerFX* [29] keretrendszerek.

A tesztek megírásához szükség lehet a **JUnit4** [30], valamint más keretrendszer felhasználására is. A tesztek írása során kézenfekvő lehet például a *Mockito Framework* és a *PowerMock Framework* funkcionalitásaihoz hasonló keretrendszer használata is.

A fejlesztéshez a **Git** [27] verziókezelő által nyújtott szolgáltatásokkal tehető átláthatóvá a forráskód, az erőforrás fájlok, valamint a példafájlok verziókezelése.

3.3 A megvalósítás részleteinek további finomítása

Bár az előző fejezetben és alfejezetekben összefoglaltak egy jó kiindulási alapot biztosítanak, azonban számos kérdést hagynak nyitva. Ilyen kérdések például:

- Milyen algoritmussal érdemes az állapotgépek szorzását elvégezni, hogy az nagyméretű állapotgépek esetében is hatékony legyen, valamint szerencsétlen esetben se eredményezzen több komponensből álló állapotgépeket elérhetetlen csúcsokkal
- Az evolúciós algoritmusok implementálásánál
    - Milyen értékkel legyenek az egyes konstansok inicializálva

- Mi történjen, ha a kromoszóma nem osztható fel ugyanolyan méretű szegmensekre
- Hogyan legyen megvalósítva az összes csúcsszámnál kevesebb csúcsot felhasználó, rövidebb utak leírása a kromoszómákban, valamint az evolúciós algoritmusok műveleteiben
- Hogyan kerüljenek kiértékelésre azok a kromoszómák, amelyek olyan utat írnak le, amely „szakadást” tartalmaz (azaz a szekvencia sorrendben két nem szomszédos csúcsot), vagy hogyan kerülhető el ezek kialakulása az operátorok használatánál

- Hogyan definiálható és értékelhető ki egy többdimenziós költségvektor az állapotgépek szorzataként előálló komplex állapotgépen
- Milyen tervezési mintákkal valósíthatók meg legegyszerűbben az egyes funkcionalitások
- stb.

A további finomítások a tervezési lépésben kerülnek majd elemzésre és kidolgozásra.

4 Tervezés részletei

Ebben a fejezetben néhány, a tervezéssel kapcsolatos megfontolás kerül kifejtésre. Mivel a grafikus felülettel rendelkező alkalmazások fejlesztését gyakran a felhasználói felület és funkcionalitások megtervezésével érdemes kezdeni, majd az alkalmazás modelljének megtervezésekor a felületi elemek által támasztott korlátokat is érdemes figyelembe venni, ezért ezen alkalmazás megtervezése is e módszerek követése alapján történik.

4.1 Funkcionális követelmények

A felhasználók számára a szoftver a következő lehetőségeket biztosítja:

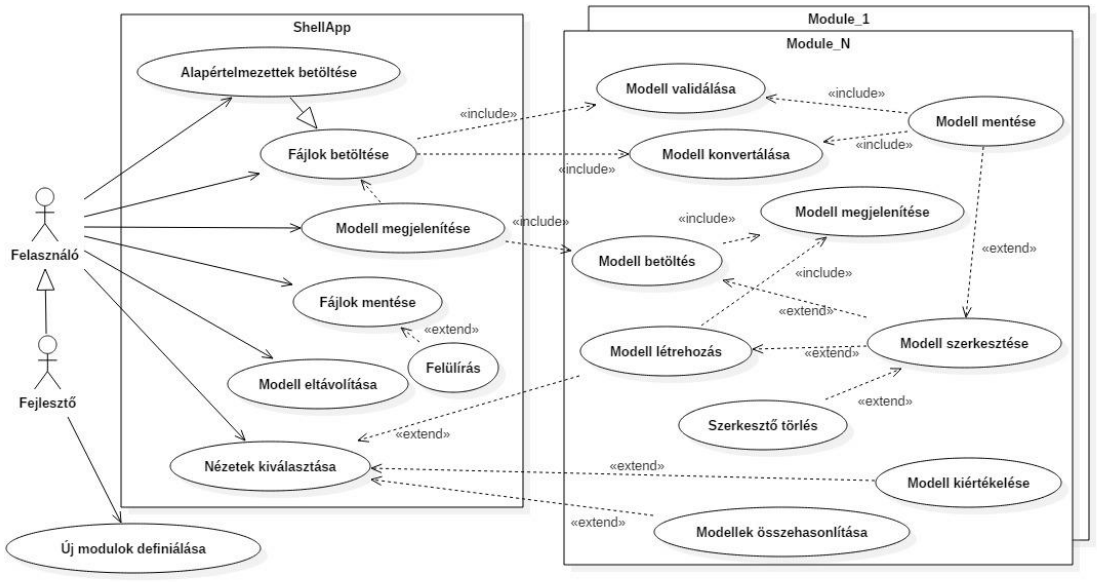
- Modellek (típus sablonok és épület modellek) megtervezése és böngészése
    - Típus sablonok (fuzzy halmazok, fuzzy állapotgépek, fuzzy szignatúrák) kialakítása
    - Típus sablonok hierarchikus típusokba történő összeszerkesztése
    - Típus sablonok adatokkal és metaadatokkal történő feltöltése, konkrét épület modellek elkészítése
- Elkészült épület modellek összehasonlítása és/vagy kiértékelése
- Elkészült épület modellek mentése, visszatöltése

A 18. ábra bemutatja a fenti követelményekből eredő főbb funkcionalitásokat. Az ábrán megfigyelhető, hogy e funkcionalitások részben a modellek különböző szakterületen történő újrahasználhatósága miatt két főbb csoportba különíthetők el, a grafikus keretalkalmazásra (**shell application**) és a modelleket definiáló, kiértékelő beépülőkre (**modules**, **plugins**).

Az ábrán szereplő keretalkalmazás (*ShellApp*) főként olyan közös funkciókat határol be, amelyek szinte bármilyen szakterületi modell esetén függetlenek a számítási modell konkrét típusától, céljától, kiértékelésétől, felületi logikájától stb., azonban a modulok működése és számítási modellek felhasználása során az egyszerűség végett érdemes lehet kiemelni, ezáltal közös csoportba gyűjteni e funkciókat. Ezért a keretalkalmazás főként szolgáltatásokat valósít meg és nyújt a beépülő modulok számára. Ilyen szolgáltatások például a fájlok egységes kezelése, modellek fájlokból történő

betöltése a munkatérbe, modellek fájlokba történő mentése a munkatérből, modellek munkatérből való törlése, a nézetek betöltésének biztosítása stb.

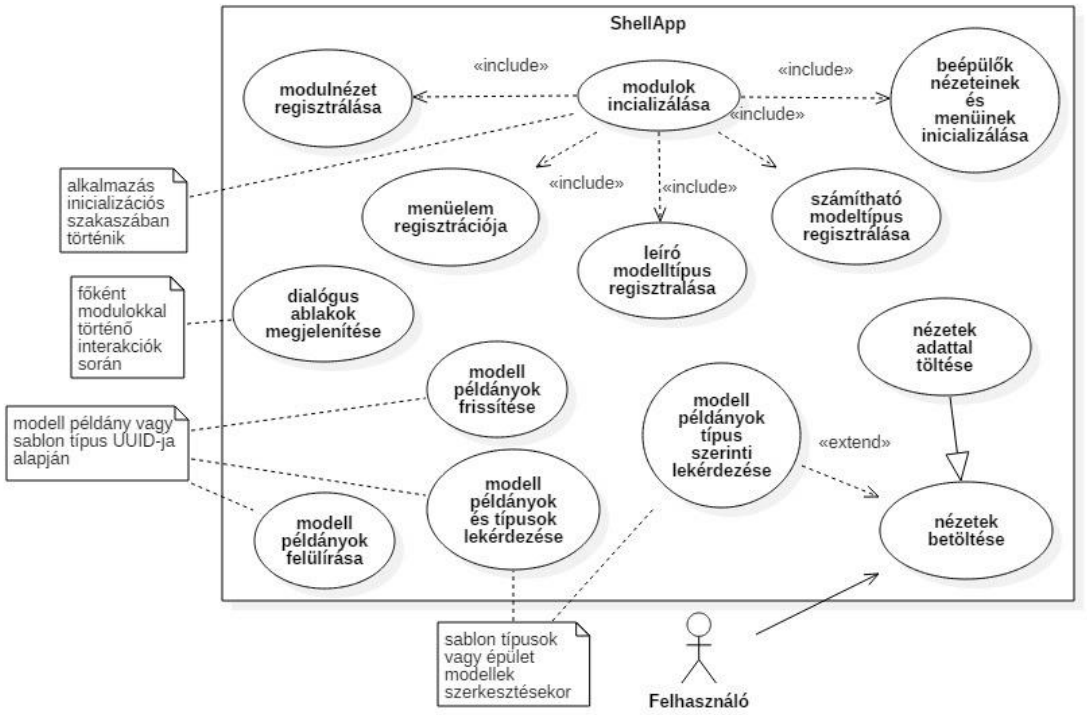
A funkcionalitások másik csoportját képezik a szakterületi modellekkel kapcsolatos funkciók. A modelleknél (fuzzy halmaz típus, fuzzy állapotgép típus, fuzzy szignatúra típus, épület modell típus vagy egy konkrét épület leírása) szükséges a modellek érvényességének ellenőrzése, a végrehajtható modellek (továbbiakban *fx model*) leíró modellekké (továbbiakban *descriptor model*) történő konverziója a modellek mentéséhez (és ellenkező irányú átalakítása a modellek betöltéséhez), a modellek megjelenítése, betöltése stb. A szerkesztő felületet nyújtó modulok lehetőséget kell, hogy biztosítsanak a modellek létrehozására, szerkesztésére, betöltésére, mentésére és törlésére, miközben igénybe vehetik a keretalkalmazás által biztosított szolgáltatásokat (export, import, munkatérbe helyezés, munkatérből törlés stb.). A kiértékelő felületet nyújtó modulok pedig a modellek összehasonlítását és numerikus kiértékelhetőségét teszik lehetővé. A 18. ábra modul rendszere alatt szereplő funkciók mindegyike nem feltétlenül van egyszerre jelen minden egyes modulban (az ábrán csak összefoglaló jelleggel kerültek egy modul alá), e funkcionalitások akár több modul között is elosztásra kerülhetnek, vagy egy modul akár az összest is megvalósíthatja.



18. ábra - A keret alkalmazás és a beépülő modulok tipikus Use Case diagrammja

A fenti Use Case diagrammon a keret alkalmazásnak csak a bemenő követelményekből adódó főbb funkcionalitásai kerültek szemléltetésre. Ezek kiegészülhetnek egyéb járulékos funkcionalitásokkal is, mint például a modulok

nézeteinek és menüelemeinek regisztrációja a keret alkalmazásban, a modulok leíró modell típusainak regisztrációja, a modulok üzleti logikát definiáló számítható modell típusainak regisztrációja, elkészített modell példányok lekérdezése típus vagy azonosítójuk szerint, modell példányok frissítése, modell példányok érvényességének ellenőrzésének kezdeményezése a modellt tartalmazó komponensnél, nézetek adatokkal történő feltöltésének kezdeményezése a megfelelő komponenseknél/moduloknál, dialógus ablakok megjelenítése (pl. hibajelzések, fájl megnyitások/mentések, modulok üzleti logikájából adódó visszajelzések megjelenítése) stb.



19. ábra – A keret alkalmazás járulékos funkcióinak Use Case diagrammja

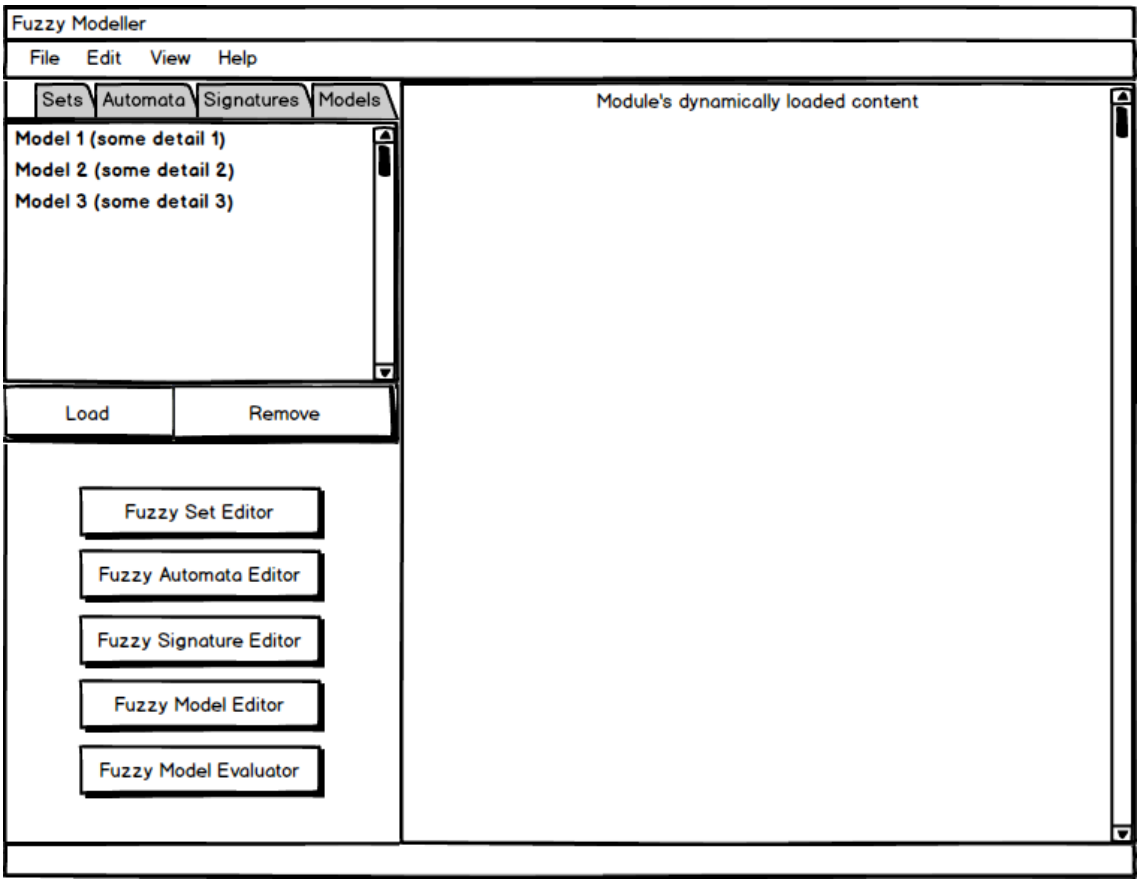
A fenti ábrán a járulékos funkciók egy egyszerűsített összefoglalása látható.

Az 18. ábra és a 19. ábra összegzéséből kikövetkeztethető, hogy a funkciókat úgy érdemes elosztani a komponensek között, hogy a keret alkalmazás egy általánosabb, generikusok segítségével megvalósított megoldás legyen, amely közös őstípusok felhasználásával és publikus interfészek nyújtásával valósítja meg a funkcionalitásait. E funkcionalitásokat veszik igénybe a publikus őstípusokat specializáló beépülő modulok, melyek a keret alkalmazás által megvalósított interfészeken és az általuk nyújtott szolgáltatásokon keresztül képesek beépülni a keret alkalmazásba és kapcsolatba lépni vele, vagy akár egymással is.

4.2 Felhasználói felület

Mivel a grafikus felülettel rendelkező alkalmazások felépítésére és működésére nézve nagy hatást gyakorol a felhasználói felület kialakítása, ezért már a tervezés kezdeti szakaszától ajánlott nagy figyelmet fordítani a felület megtervezésére. A felhasználói felületnek elsősorban mindig a felhasználói igényekhez ajánlott igazodnia. Emiatt egy könnyen áttekinthető és intuitíven használható felhasználói felület kialakítása célszerű. Ehhez a hagyományos asztali alkalmazások egyablakos elrendezése lett alapul véve.

A felület alapkoncepciójának bemutatásához, valamint egy egyes funkciók grafikus elemekhez történő párosításához néhány mockup[21] került elkészítésre, melyek közül a keretalkalmazás véglegesnek tekintett vázlata tekinthető meg lent (20. ábra).



20. ábra - Keret alkalmazás felületének wireframe[22] modellje

[21] Mockup – egy felület kialakításának modellje, főbb felületi elemek elhelyezkedésének terve, amely akár még kattintható és bemutatható prototípusként is szolgálhat a termék fejlesztésének kezdeti szakaszában.

[22] Wireframe modell – egy felület drótvázas ábrázolása.

Az ábrán látható, hogy a grafikus felület három lényegi területre különül el (a menüsort és a státuszsort leszámítva). Ezek közül mindhárom terület tartalmát a beépülők határozzák meg.

A bal alsó részen látható gombokat a keretalkalmazás generálja a modulok által beregisztrált nézetek alapján. Egy modul akár nulla, vagy több nézetet is regisztrálhat. A gombok segítségével megnyithatók a modulok által regisztrált nézetek, melyek között előfordulnak a szerkesztéshez és a kiértékeléshez használt felületek.

A bal felső részen a modulok segítségével létrehozott, vagy korábbról betöltött sablon típuspéldányok és modell példányok listaszerű megjelenítései láthatók a modulok által beregisztrált adattípusok szerint. A példányokat tartalmazó listák a típusaik szerint vannak elkülönítve egymástól és egy betöltés, valamint eltávolítás gombbal vannak ellátva. Egy modul akár nulla, vagy több típust is beregisztrálhat, melynek tárolóiba (munkatér típus szerinti felosztásai) példányokat hozhat létre vagy tölthet be a fájlból. A típusok szerint itt elhelyezett modelleket (pl. épületmodellek) és újra felhasználható sablon típusokat (pl. fuzzy halmaz, állapotgép, szignatúra típus stb.) egy modul más modulok számára is elérhetővé teszi a keretrendszerben történő modellpéldány regisztrációja során. Egy modulnak nem kötelező saját típusait regisztrálni és ezáltal a modell vagy sablon típus példányaikat publikussá tenni, használhatja akár a modul ősei által már regisztráltakat is, vagy akár semmilyen publikus típust sem deklarálhat. Azonban csak a közös munkatérben elhelyezett sablon típus és modell példányok lesznek közös szolgáltatások által elérhetőek más modulok számára (mivel a háttérben ez a keretrendszer modell megosztási funkciójának alapja is egyben). Ez a megközelítés viszonylag jól illeszkedik a hierarchikus modellek szerkesztésének követelményéhez (még ha a különböző típusok különböző modulokban is kerülnek kialakításra), miszerint a listából kiválasztott típusok egy magasabb szintű típusban vagy modell példányban könnyen felhasználhatóvá, vagy hivatkozhatóvá válnak.

A jobb oldalra a modulok által regisztrált nézetek tartalmai kerülnek betöltésre, melyek betöltése a két bal oldali nézetből kezdeményezhető, vagy a nézet egyszerű megnyitásával (egy bal alsó gombra történő kattintással), vagy egy meglevő modellpéldány betöltésével (bal felső nézetben egy konkrét modell leírás vagy modell sablon típus példányának kiválasztása után a betöltésre kattintva).

A modulok egyéb területekre is regisztrálhatnak tartalmakat. Ilyen például a menürendszer bizonyos kiterjesztési pontjai.

A tervezés során a modulok által megvalósítandó nézetekről is készültek a fentihez hasonló wireframe modellek, azonban terjedelmi okok miatt ezek végül e dokumentumba nem kerültek be.

4.3 Modellek

Az alkalmazásban és annak moduljaiban használt modell entitások többféle szempontrendszer szerint csoportosíthatók, melyek egy praktikus implementációban néha átfedésben lehetnek egymással, ezért az egyes modellcsoportok könnyen összekeverhetők.

A tervezés során használt legfontosabb csoportosítási szempont a modell alkalmazásának célja szerinti feloszlása, mely az alábbi kategóriákat különbözteti meg:

- **Számítási modellek** – Ide értendő minden olyan modell, amely az alapprobléma megoldása és kiszámítása szempontjából lehet érdekes (pl. fuzzy szignatúrák, fuzzy állapotgépek, fuzzy halmazok, bakteriális modell, kromoszóma modellje stb.).
- **Szakterületi modellek** – Ez alatt értendők azok a modellek, amelyek a konkrét domain terület szerint lehetnek érdekesek (pl. épületfelújítási vagy villamos energetikai probléma leírása, ehhez tartozó egyéb információk stb.). Ezek a modellek tipikusan a számítási modellek felett helyezkednek el.
- **Felületi modellek** – Ezek vagy a választott grafikus keretrendszerhez illeszthető (modulok által megvalósított) modellek, amelyek adatkötések útján automatikusan értesülhetnek a felületen történt változásokról, vagy pedig azok a (keretrendszer által) generikusan megvalósított modellek, amelyek számára a keretrendszer bizonyos szolgáltatásokat képes nyújtani, vagy rajtuk eljárásokat végrehajtani.

A számítási és szakterületi modellek a szerkesztés során történő újrahasznosíthatóságuk miatt a szóhasználatban az alábbiak szerint különböztethetők meg:

- **Sablon modell típusok** – Azok a számítási és szakterületi típusok, amelyek példányai a modellek összeszerkesztése során újrahasználhatók (ilyen például egy fuzzy állapotgép, amely akár azonos, akár különböző módon felparaméterezve felhasználható vagy egy szignatúra két különböző levelében, vagy két különböző szignatúrában).
- **Feladatot leíró modell típusok** – Azok a szakterületi típusok, amelyek a modell egészét konkrét adatokkal feltöltve tartalmazzák és kiértékelhetők egy adott probléma felett (például mennyi a legkisebb összköltsége *x* állapotból *y* állapotba történő felújításnak, vagy *z1* épület esetleg *z2* épület felújítása a költséghatékonyabb).

Újabb csoportosítási lehetőséget képez a modellek végrehajtása szerinti felosztás, melyekre már a fejezet elején történt hivatkozás:

- **Végrehajtható modellek (fx model)** / Dinamikus modellek – A modellek olyan csoportját fedik, amelyeken elvégezhetők bizonyos műveletek pl. számítási feladatok kiértékelése, felületi változások átvezetése stb. (Megj.: ez utóbbiból ered az *fx* prefixum, ugyanis ez az elkészült implementációban a JavaFX speciális elvárásaihoz kapcsolható tulajdonság.)
- **Leíró modellek (descriptor model)** / Statikus modelek – Az entitások olyan csoportját fedik, amelyek pusztán a modellek leírására szolgálnak. Rajtuk műveletek (a leírt modellek validációját leszámítva) nem végezhetők el. Ide tartoznak pl. a perzisztáláshoz használt osztályok. (Megj.: ez a kifejezés az elkészült implementációban Java POJO-ihoz hasonló, megfelelő tulajdonságaikon annotált osztályokat takar.)

Az alkalmazás konkrét implementációjában elhanyagolódhat a modellek felosztásának és csoportosításának szerepe, ugyanis akár több kategóriát is átfedhetnek a fenti felosztásból. A fenti felosztások leginkább a főbb tervezési szempontok megértése szempontjából célszerűek.

4.4 Architektúra

Az alkalmazásban a JavaFX-hez leginkább illeszkedő **Model-View-Controller** (**MVC**) (a továbbiakban hivatkozva vagy angol nevén, vagy mint modell-nézet-vezérlő) architektúra kialakítása javasolt. E megközelítést használva a rendszer valamennyi

osztálya e három csoport (modell, nézet és vezérlő) valamelyikének megvalósítása. A gyakorlatban több variációja is elterjedt ennek az architektúrális megközelítésnek, de általánosságban igaz az, hogy a modellek tipikusan az adatok tárolásáért, kiterjesztett esetben a rajtuk végezhető műveletek végrehajtásáért is (pl. szakterületi algoritmusok futtatásáért, számítások elvégzéséért) felelősek. A nézetek feladata a felület megjelenítésének és a megjelenítés logikájának definiálása, tipikusan passzív elemek segítségével. A vezérlők felelnek a modellek és a nézetek megfelelő összeköttetéséért és a rajtuk végezhető műveletek végrehajtásának menedzseléséért. Tipikusan az ő feladatuk a nézetek, vagy legalább a modellek példányosítása, szükség esetén a modellek belső állapotának megváltoztatása, valamint a felületi elemek állapotváltozása esetén tipikusan általuk kerülnek átvezetésre a nézeteken végrehajtott változások a modellek irányába és fordítva (hacsak nem közvetlen adatkötés van megvalósítva a nézetek és a modellek között).

A JavaFX MVC alapú megvalósításban a nézetek példányosítják a hozzájuk tartozó vezérlőket és a felület elemei adatkötéssel (databinding) kapcsolódhatnak a modelleket megfigyelhető tulajdonságként (property-ként) nyújtó vezérlőkhöz. Amennyiben a modell egyszerű és csakis az adatok „tárolásáért” felel, úgy az üzleti logika szakterület specifikus részeit a vezérlő is megvalósíthatja, ellenkező esetben ez átkerülhet a modell hatókörébe, miközben a vezérlőre csak az összehangoló szerepköre jut.

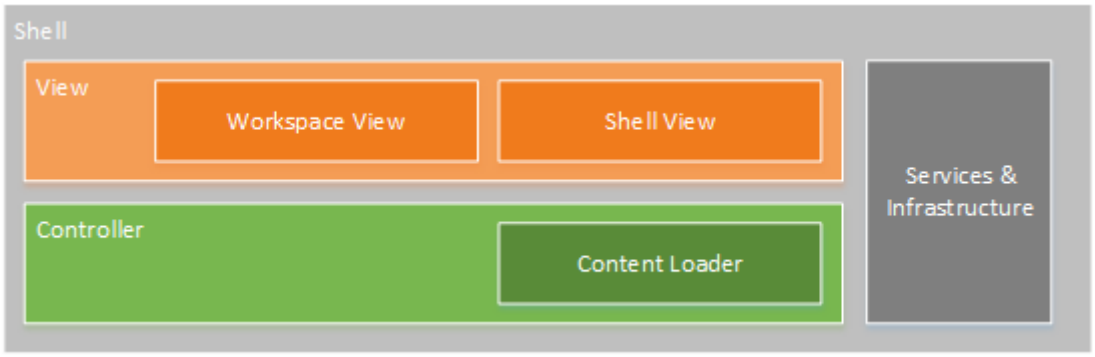
Az MVC minta alkalmazása az alkalmazás fenti igényeinek tekintetében némileg eltérő módon implementálandó a keretrendszerben, valamint a modulokban. Ezek tervezett megvalósításai az alábbiakban kerülnek összefoglalásra.

4.4.1 A keretrendszer felépítése

A keretrendszer sajátossága, hogy bár az MVC mintán alapul, azonban (alapkörülmények között) attól eltérően mégsem tartalmaz a hagyományos értelemben vett igazi modellt (csupán azok megvalósítását előíró absztrakt osztályokat és interfészeket), ahogy ez a 21. ábrán is megfigyelhető.

A keretrendszer valódi modelljei modulok formájában kerülnek definiálásra, melyek dinamikusan tölthetők be a keretrendszerbe (továbbiakban *shell*). Ehhez a keretrendszer a modulok által specializálandó kiterjesztési pontokat nyújt (feltölthető nézetek, örökölhető adatszerkezetek, interfészek és absztrakt osztályok formájában),

valamint ezek a keretrendszerrel történő együttműködéséhez igénybe vehető szolgáltatásokat kínál beépülői számára.



21. ábra - Keretrendszer megvalósításának architektúrája

A fentiek következtében a *shell controller*-einek feladata a beépülők által implementált nézetek betöltése és integrálása a keretrendszer felületébe, valamint a beépülők vezérlőinek megfelelő inicializálása. További feladatai között szerepel, hogy műveletek végrehajtását biztosítsa a beépülőkön (pl. a felület adatokkal történő feltöltése – *Content Loader*).

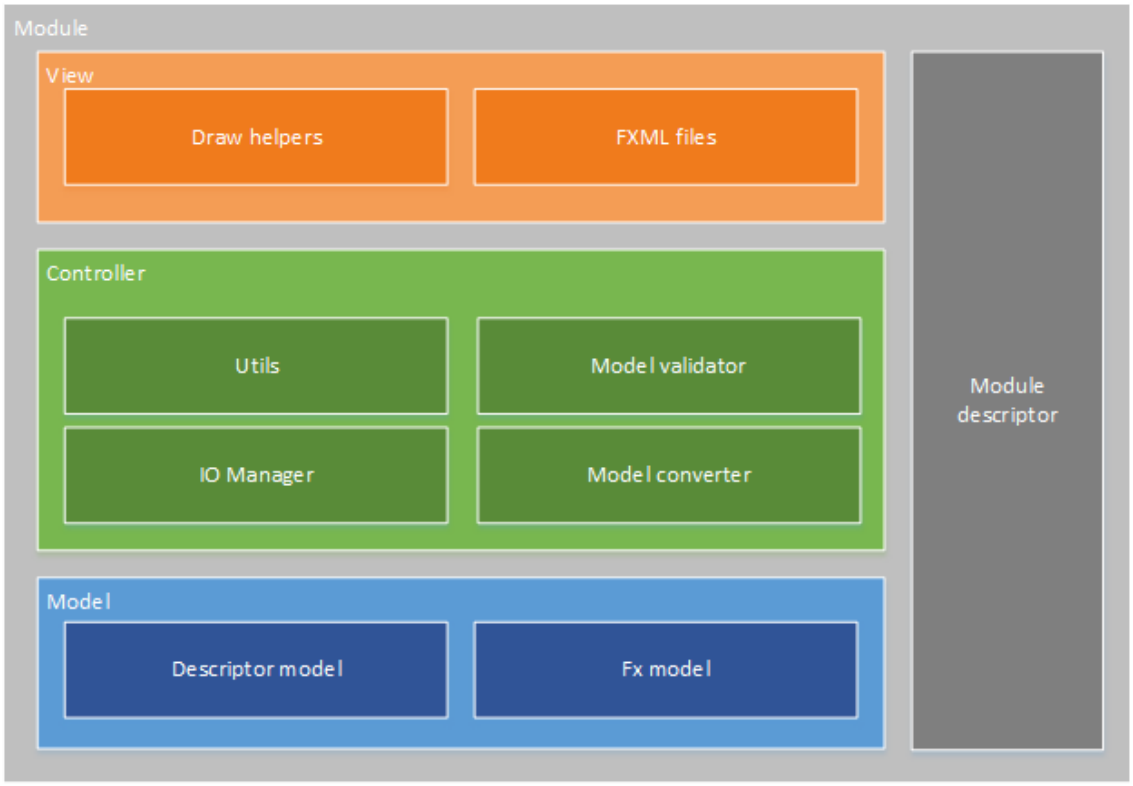
A *shell* nézeteinek feladata, hogy vizuális elemekkel (és hozzájuk *controller*-ben kapcsolható funkcionalitásokkal) egészítsék ki a modulok által definiált nézeteket (pl. *WorkspaceView*, amely az adott típusú modell példányhoz megjeleníti a betöltés, törlés gombokat - 20. ábra bal felső területe, a *MenuView*, amely a regisztrált nézetek menüpontjaihoz tartozó nyomógombokat generálna - 20. ábra bal alsó területe, vagy a *ShellView*, amely feladata a beépülők nézeteinek testreszabható megjelenítésének biztosítása - 20. ábra jobb oldala).

A *shell* által megvalósított szolgáltatások felelősek a modulok integrációjáért (amennyiben a modulok biztosítják az ezzel kompatibilis interfészek helyes megvalósítását). Továbbá több kiegészítő szolgáltatás implementációja is itt kapott helyet, melyek például megkönnyítik a leíró modell/sablon típus példányok regisztrációit, nézetek vagy modellek kereshetőségét biztosítják akár regisztrált típusok vagy egyedi azonosítók alapján stb.

4.4.2 A beépülő modulok felépítése

Ellentétben a keretrendszerrel a modulok felépítése már jobban illeszkedik a hagyományos MVC mintára. Itt a modellek már a számítási és a szakterületet leíró,

valamint azokon alkalmazható műveleteket is végrehajtani képes entitások. A beépülő modulok tipikus felépítése megtekinthető a 22. ábra.



22. ábra - Modulok tipikus megvalósításának architektúrája

A tervezett megvalósítás szerint a modellek két fő csoportra oszthatók, melyeket a modulok vezérlői ismernek és használják, továbbá a keretrendszer is képes ezen típusok kezelésére (az absztrakt ősosztályaikon keresztül).

A modellek egy része passzív, csupán leíró jellegű és primitív, abban az értelemben, hogy nem tartalmaznak semmilyen számítási, vagy felület logikát (korábbiakban *descriptor model*). Ezek csupán az adatok perzisztálásáért, valamint beolvasásáért felelnek. Tulajdonságaik az XStream annotációival lettek ellátva, így kialakítva a fájlok sémáját, mellyel az adatok a háttértárolóra menthetők XML, valamint JSON formátumokban. Minden modul saját hatókörben biztosítja ezen fájlok exportálásának, importálásának és validálásának módjait.

A modellek másik csoportja aktív olyan értelemben, hogy számításokat végző algoritmusok is használják őket és képes a megvalósított szakterületre jellemző számítások kiértékelésére az általa tárolt adatokon (korábbiakban *fx model*). E modellek az információt tipikusan olyan formában tárolják, amelyek akár a felületi elemekhez is

köthetők a vezérlőkben definiált adatkötések által, azaz változásukról automatikusan értesülnek a felületi elemek, melyek így megjelenésükben lekövethetik azokat.

A vezérlők elsődleges feladatai közé tartozik a modellek példányosítása, adatokkal történő feltöltése, felületi elemekhez történő illesztése és a nézet(ek) működésének összehangolása a modell(ek)el. Ehhez a tervek szerint további kiegészítő funkciók is megvalósításra kerülnek, mint pl. a *descriptor* és *fx modellek* egymásba történő konverziói (*Model Converter*), a létrehozott és betöltött modellek ellenőrzése (*Model Validator*), valamint a perzisztálás modul- és modellfüggő tulajdonságainak specifikálása (*IO Manager*) stb.

A modulok nézetei egyszerűen megvalósítják azokhoz a funkciókhoz tartozó felhasználói felületeket, melyeken a felhasználók összeszerkeszthetik vagy kiértékelhetik modelljeiket. Ennek támogatására alkalmanként segédosztályok kerülhetnek bevezetésre (pl. *draw helpers, utils stb.*), melyek egy-egy részfunkcióért lehetnek felelősek (például gráfrajzolás).

A fenti ábrán az MCV részegységei mellett található egy további osztálycsoport (*module descriptor*), amely nem illeszkedik az architektúrába (bár funkcionálisan lehetne tekinthető akár a vezérlők egy speciális csoportjának is). Ennek feladata, hogy megadja a modul keretrendszerbe történő integrációjának részleteit, valamint publikus információkkal szolgáljon a modulra építő gyermekmodulok számára. Ilyen leírókban történik a keretrendszer által önmagába integrálni és megjeleníteni képes nézetek (és azok megjelenítendő neveinek, menüelemeinek stb.) regisztrálása, valamint olyan modelltípusok regisztrációja, melyeken a keretrendszer képes műveletek végrehajtására (pl. perzisztálás, leíró és végrehajtható modellek közötti konverzió), valamint a konvertálást végezni képes osztályok típusainak megadása. E leírók által tartalmazott típusok használatba vétele általában a keretrendszer által definiált interfészeken, ősosztályokon, generikus típusokon, vagy a Java Reflection API-n keresztül történik. Mivel a már regisztrált típusokat a keretrendszer publikusan is elérhetővé teszi egyben, ezért a gyerek modulok a keretrendszer álltal nyújtott *common service*-ek felhasználásával többek között a saját *module descriptor*aik számára kérhetnek le leíró jellegű információkat azokról a modulokról, melyek nézeteit vagy funkcionalitásait szeretnék kibővíteni, vagy modelljeiket felhasználni.

Megemlítendő, hogy a 22. ábra segítségével felvázolt architektúra, nem egy kötelezően előírt és kikényszerített minta. A készítendő modulok akár a fentitől eltérő

módon is megvalósíthatók, bár a fent részletezett megvalósítás a tapasztaltak alapján egy ésszerű és hatékony felépítésnek bizonyult (így akár egy ajánlott mintának is tekinthető). A modulok sikeres integrációjának kritériuma a későbbiekben bemutatott interfészek, absztrakt osztályok megfelelő használata, valamint a keretrendszer regisztrációt végző metódusainak meghívása.

4.4.3 A keretrendszer és modulok együttműködése

Az alkalmazás indulásakor a keretrendszer inicializációs fázisában egyebek mellett megtörténik a keretrendszer beépülőinek felderítése és inicializálása is.

Miután a keretrendszer induláskor sikeresen azonosítja a rendelkezésre álló modulokat (melyek a tervek szerint az alkalmazás végleges változatában akár már egyszerű másolás útján is telepíthetők), feltérképezi azok függőségeinek rendszerét, majd a függőségek hierarchiája szerint haladva meghívja azok leíróiban levő inicializációt végző metódusokat.

Miután a modulok a leírókban elhelyezett inicializációs szakaszaikban regisztrálták nézeteiket és (a keretrendszerrel, vagy más modulokkal közös) típusaikat a keretrendszerben, mind a modulok, mind pedig a keretrendszer passzív várakozó állapotba kerülnek.

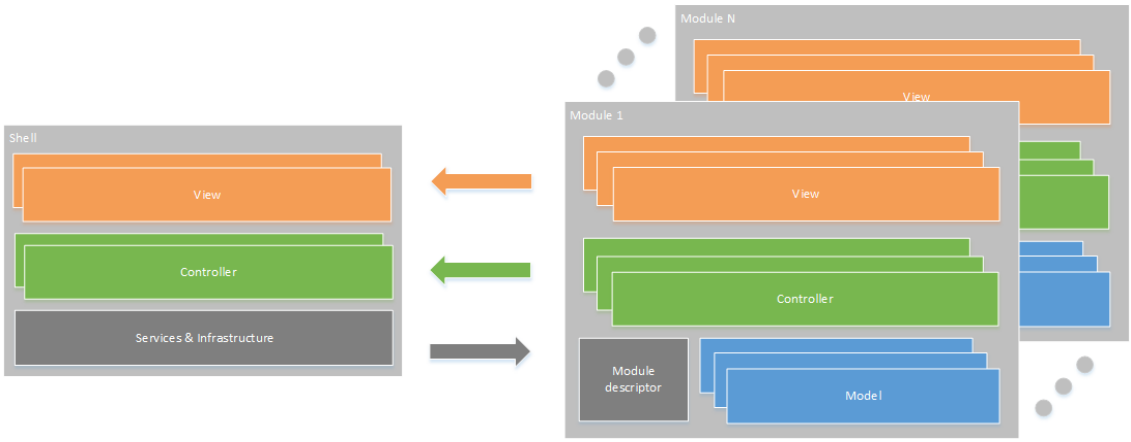
A keretrendszer a felhasználói interakciók csakis azon csoportjának végrehajtásával foglalkozik, melyek a modulok által a keretrendszerben regisztrált információkra eredeztethetők vissza. Ezek többnyire az alkalmazás által biztosított infrastruktúrával kapcsolatos műveletek (pl. modulok nézetek betöltése, modellek exportálása és importálása stb.).

A felhasználói interakciók azon csoportja, melyek a modul által közétett nézeteken és modelleken történnek, a modul nézeteinek és vezérlőinek hatókörében kerülnek feldolgozásra. Ezek többnyire szakterület specifikus műveletek (pl. fuzzy halmaz létrehozása, szerkesztése, épület modell kiértékelése stb.). Amennyiben ezek a műveletek mégis a keretrendszer szolgáltatásainak igénybe vételét igényelik, ezen szolgáltatások igénybe vehetők bármely modulból a *Service Factory* által visszaadott publikus interfészt megvalósító objektumon keresztül.

A 23. ábrán látható ahogy a keretrendszer az infrastruktúráját, valamint szolgáltatásait elérhetővé teszi a modulok számára, melyek ennek felhasználásával még

a saját inicializációs szakaszaikban a saját publikus részegységeiket regisztrálják a keretrendszerben.

A modulok, valamint az azokon belül elhelyezett főbb részegységek száma nem limitált, egy modulon belül tetszőleges számú (akár nem regisztrált) nézet, vezérlő és modell is elhelyezhető. Modul leíróból azonban a keretrendszer sajátosságai miatt érdemes egyetlen típust definiálni.



23. ábra - A keretrendszer (shell) és a beépülők (modules) viszonya

4.4.4 Beépülő modulok és modellek rendszere

Mivel az alkalmazás szétválik egy keretrendszerre és abba beépülő több (akár egymásra épülő) modulra, így a modulok függőségi rendszere nem lehet DAG (Directed Acyclic Graph), ezáltal a kertrendszer képes a függőséggel nem rendelkező modul(ok)tól kiindulva az ők által exportált függőségeket használó modulokig egyenként mindet betölteni, amennyiben a függőségek rendelkezésre állnak.

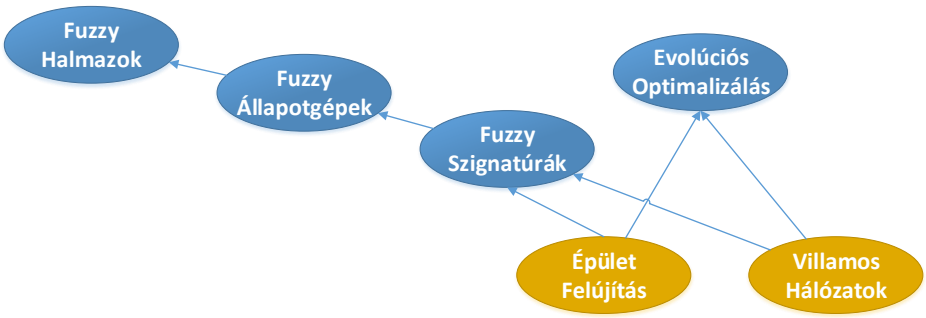
A keretrendszer egyik legfontosabb feladata, hogy dokkolja a beépülő modulokat, egységes felületbe foglalja azok tartalmát, valamint közös szolgáltatásokat is biztosítson számukra. Az előbbihez a grafikus felületen elhelyezkedő közös területek kínálnak lehetőséget, melyekre akár a modulok által definiált felületek kerülhetnek betöltésre. Egy modul akár több ilyen betölthető felületet is regisztrálhat a keretrendszerben, azonban egyszerre csak egyikük lehet megjelenítve. A regisztrált felületek oldalválasztó gombok segítségével választhatók ki, melyeket a *shell* hoz létre a regisztrált modulok által nyújtott információkból. A közös szolgáltatások egyike, hogy egy modul publikusan hozzáférhető adattípusokat is regisztrálhat a keretrendszerben, melyek így más modulok számára is lekérdezhetők és felhasználhatók. Ezek a keretrendszer által meghatározott formában listázásra kerülnek, így kiválaszthatók és betölthetők, ezáltal böngészhetők vagy

szerkeszthetők a megfelelő komponens valamelyik felületének segítségével. A fentiek miatt egy komponensre ráépülő modul számára nem szükséges a komponens belső működésének ismerete, elég csupán a keretrendszertől elkérni a regisztrált modellek listáját.

A regisztrálható nézeteken és adatszerkezeteken keresztül ezáltal több lépésben összeszerkeszthető egy olyan hierarchikus adatszerkezet, melynek részei több különböző, egymásra épülő modulok használatát is igényelhetik. A közös együttműködéshez a keretrendszer kötelezően implementálandó interfészeket ír elő a modulok számára, valamint ő maga is publikus interfészeket implementál, mellyel vállalja, hogy a szolgáltatásokat képes nyújtani a beépülői számára. Ilyen közös szolgáltatás lehet például az adatok egységes perzisztálása, betöltése stb.

4.5 Szakterületi és számítási modulok viszonya

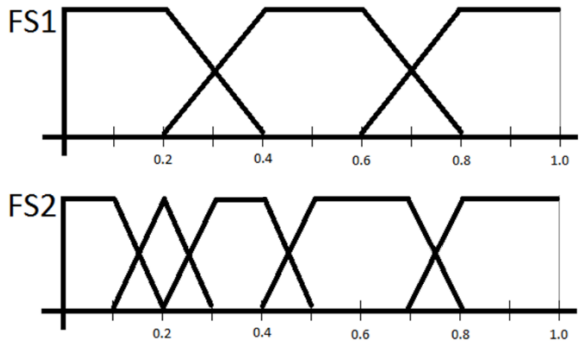
Az előző alfejezetekben tárgyaltak szerint az épületfelújítást (vagy más, szignatúra alapú számítási modellt és evolúció alapú optimalizálási modellt használó szakterületi rendszerek konkrét megvalósításait) az alábbi módon javasolt modulokba szervezni a fent vázolt keretrendszer és modulrendszer sajátosságait figyelembe véve.



24. ábra - Számítási és szakterületi alrendszerek viszonya

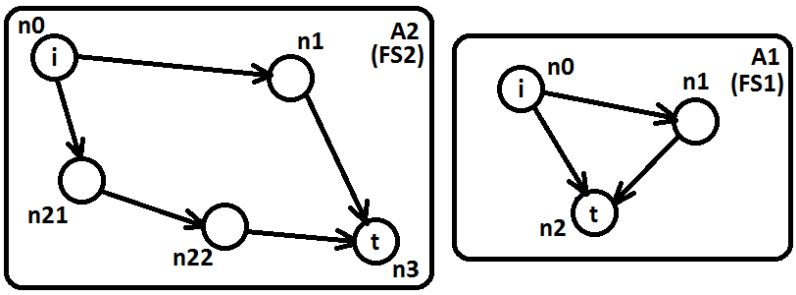
A 24. ábrán megfigyelhető, hogy a kékkel jelölt számítást végző komponensek függőségeik alapján két klaszterbe csoportosíthatók, melyek egymástól függetlenül is implementálhatók. Mind az evolúciós optimalizálással foglalkozó, mind pedig a fuzzy domainbe tartozó számítási modellek szerves részeit alkotják az ábrán sárgával jelölt szakterülethez tartozó alrendszereknek, de a fentebb felvázolt keretrendszerbe olyan szakterületi modulok is fejleszthetők, amelyek ezek közül csak az egyiket, vagy akár egyiket sem használják fel.

A fent javasolt modulrendszer kialakítása teljesíti a használandó számítási modellek felépítését, valamint a típusok újrahasználhatóságával kapcsolatos kritériumot is. Elsőként fuzzy halmazrendszer típusdefinícióinak megadását szükséges biztosítani (25. ábra).



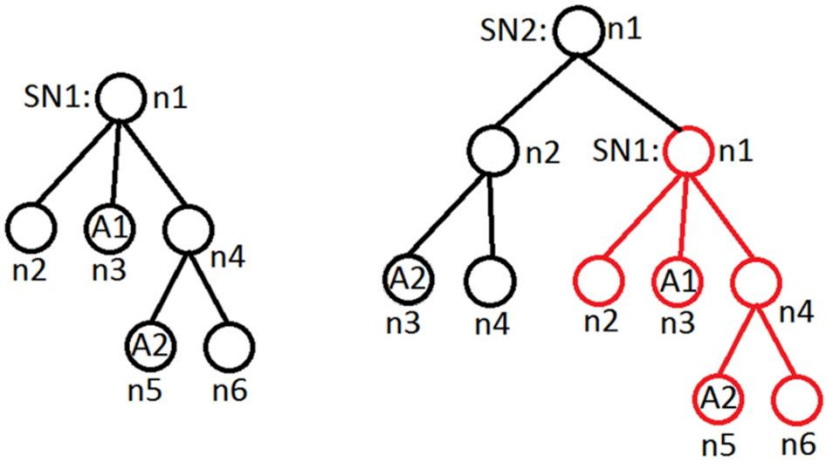
25. ábra - Fuzzy halmazok megadásának grafikus szemléltetése

E halmazrendszer típusok felhasználhatók az állapotgépek típusdefinícióinak megadásához (26. ábra), ahol egy átmenet a fuzzy halmazok feletti átmenetet valósít meg. Egy állapotgép nem szükségszerűen rendel minden fuzzy halmazhoz egy-egy csúcsot, vagy minden fuzzy halmaz párhoz egy-egy átmenetet. Az állapotgép definíciójának meghatározásánál megadható, hogy melyik csúcs tekinthető alapértelmezett kezdő állapotnak és végállapotoknak, valamint melyek az egyes átmenetekhez tartozó alapértelmezett költségek (vagy költségvektorok többcélú optimalizáció esetén). Az állapotgépek élein szereplő költségek a konkrét épületek állapotainak függvényében, csúcsainak típusa az optimalizáció céljának függvényében változhat, így ezek értékei az épületfelújítást megvalósító modulokban felülírhatók vagy tovább pontosíthatók. Az állapotgépeknél felvett költségek abszolút/relatív viszonya értelmezhető a sablon modellek és épületmodellek között.



26. ábra - Fuzzy állapotgépek megadásának grafikus szemléltetése

Az állapotgépek típusdefiníciói felhasználhatók a szignatúrák típusdefinícióinak megadásához. A szignatúrák esetében az újrahasználhatóság többszörösen is értelmezhető. Egyrészt egy szignatúra típus levél csúcsaiban többször is felhasználhatók a korábbiakban megadott állapotgép típusok. Nem szükséges tehát új, különböző elnevezésű, de tartalmilag azonos állapotgép típusokat felvenni ugyanolyan fogalmak átmeneteinek leírására (így kivédhető pl. a „jó”, „közepes”, „rossz” fogalmak redundáns értelmezése). Másrészt a már elkészült szignatúra típusok is újrahasznosíthatók egy másik szignatúra típusdefiníciójában, annak egy részfájaként (27. ábra). Ez a fajta újrafelhasználhatóság leegyszerűsíti az alapvetően különböző típusú, de részben hasonló struktúrájú adatok kezelését, valamint a későbbiekben is szerepet játszhatnak a különböző épület típusokra illeszkedő konkrét feladatleírások összehasonlítása során használandó fatranszformációknál is.



27. ábra - Szignatúrák újrahasználhatóságának értelmezése

A fenti ábra jobb oldalán látható, ahogy egy adott állapotgép típus a szignatúra típus két különböző levelében is felvételre kerül. Továbbá látható a bal oldali szignatúra típus részfaként történő felhasználása a jobb oldali szignatúra típusában.

A szignatúrák által előállított szorzat állapotgépeken történő optimalizációnak kétféle megvalósítása lehetséges. Egyik lehetőség szerint az evolúciós algoritmus a szignatúrák modulja felé épülve ismeri és használja annak adatszerkezeteit, így végezve rajtuk el az optimalizációt. Második lehetőség szerint az optimalizációs modul interfészek definiálásával és generikus implementációjával teszi elérhetővé a szolgáltatásait és a szolgáltatásokat igénybe vevő fél feladata ezek megfelelő kitöltése és implementációja. Ez utóbbi kialakítással a bakteriális algoritmus működése elfüggetleníthető a szignatúra modelltől, az optimalizációhoz szükséges feltételek garantálása akár szakterületi modulba

is eltolhatók, miközben a modulokhoz egy lazábban csatlakozó és más területen is újrafelhasználható keretrendszer kerül kialakításra. Emiatt ez utóbbi megközelítés használata javasolt.

5 A megvalósítás részletei

Az alábbiakban körvonalakban összefoglalásra kerülnek az alkalmazás implementációja során hozott döntések, melyek kihatással voltak az alkalmazás felépítésére, működésére és kritikus szerepük lehet az alkalmazás továbbfejlesztése, vagy további moduljainak megalkotása során.

5.1 A szakterületre és a számítási modellre vonatkozó implementációs részletek

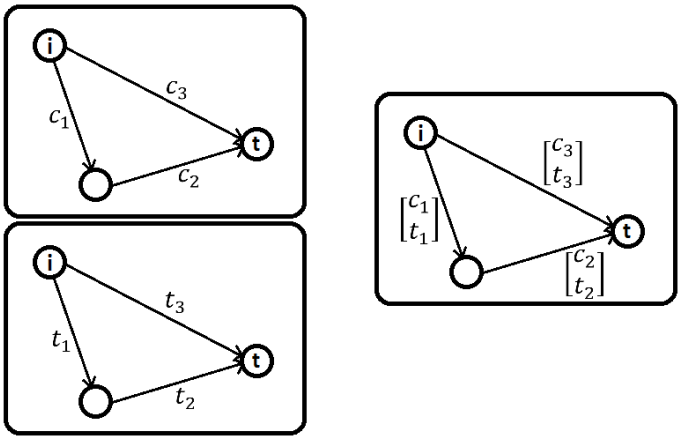
Ebben az alfejezetben áttekintésre kerül néhány a szakterületi modellre vonatkozó implementáció szempontjából fontosabb részlet és tervezői döntés.

5.1.1 Többdimenziós költségek fogalma a fuzzy állapotgépek felett

Korábbiakban már esett szó arról, hogy a szignatúrák levelein definiált állapotgépek az egyes részfolyamatok kivitelezésének leírásai. Az állapotgép állapotai fuzzy halmazokhoz vannak rendelve, egy állapotátmenet pedig a fuzzy halmazok közötti állapotátmenet leírásai. A fuzzy állapotgép minden állapota egy-egy fuzzy halmazhoz kötelezően hozzárendelt értéket takar, azonban nem kötelező egy univerzum összes halmazához állapotot rendelni, ekkor az adott fuzzy halmaz által leírt állapot elérése nem lehetséges az állapotgép segítségével (pl. a felújítás során kimarad a „közepes” állapot és „rossz” állapotról egy felújítási lépésben kerülünk a „jó” állapotra).

A többcélú optimalizációhoz olyan állapotgép használata javasolt, amelynek élein a költség nem skalárként, hanem vektorként kerül definiálásra. Ezáltal a vektor különböző komponensei eltérő fogalmakat kódolnak (pl. vektor első komponense a financiális költséget, második komponense az időt, harmadik jogi, esetleg technikai megvalósítással kapcsolatos paramétereket kódol stb.). A vektorok komponensei nem szükségszerűen ordinális változók, lehetnek akár kategorikusak is, sőt egy vektoron belül eltérő típusúak is. Ennek oka, hogy egy vektorban szereplő értéknek önmagában kicsi a jelentősége, csakis más vektorok viszonyában értelmezett és hasonlítható ugyanazon komponenssel. Az összehasonlításhoz viszont szükséges az összehasonlítás módjának definiálása az egyes komponensek által kódolt információk segítségével (például melyik komponenst szeretnénk jobban súlyozni az optimalizációban, pl. összköltség vagy idő

minimalizálására, esetleg jogi/technikai megfelelőség és minőségi kritériumok maximalizálására törekedjünk).



28. ábra - Költségek megadása állapotgépek felett

Ahogy az a 28. ábrán is látható, különböző típusú költségeket skalár értékekkel csak két ugyanolyan típusú állapotgép felhasználásával lehetne megadni, azonban a szignatúra egy tetszőleges leveléhez csak egyetlen állapotgép csatolása ésszerű. Ennek következménye a különböző típusú értékek vektorban történő reprezentálása még akkor is, ha azok osztályozásuk típusa szerint (nominális, ordinális stb.) eltérnek. A költségvektorok aggregációjának megadása a felhasználó felelőssége.

5.1.2 Állapotgépek szorzatát előállító algoritmus

Az állapotgépek[23] szorzatának előállításához többféle algoritmus használható, amelyek eltérnek megvalósításukban, komplexitásukban, egyéb tulajdonságaikban. A továbbiakban egy egyszerűbb és egy bonyolultabb változat összehasonlításán keresztül kerül részletezésre az implementált algoritmus.

Az egyszerűbb algoritmus helyesen állítja elő a szorzat állapotgépet, azonban előfordulhatnak benne nem elérhető állapotok és átmenetek. A bonyolultabb algoritmus is funkcionálisan helyes állapotgépet eredményez, azonban ezt elérhetetlen állapotok felvétele nélkül éri el úgy, hogy a kezdőállapotpárból kiindulva minden lépésben egy adott csúcs lehetséges kimeneteit vizsgálva egyszerre mindkét állapotgépen csak a két állapotgép egyidejű működésében elérhető állapotokra korlátozza a végrehajtást [12].

[23] Determinisztikus Véges Állapotú Gép - Deterministic Finite State Machine (DFSM)

Jelölje egy állapotgép állapotait Q, ábécéjét Σ, átmeneti függvényét δ: $Q \times \Sigma \rightarrow Q$, kezdőállapotát $s \in Q$, elfogadó állapotait $F \subset Q$, azaz formálisan egy determinisztikus állapotgép megadása a $M = <Q, \Sigma, \delta, s, F>$ ötös megadással történik. Jelölje T táblázat $T[q,r]$ elemének értékét a δ függvény értéke a $\delta(q,r)$ helyen, M_a és M_b a két összeszorzandó állapotgépet, M_p a két állapotgép szorzatát, R pedig egy olyan halmazt, amely kezdetben a két M_a és M_b állapotgép kezdőpontjaiból adódó állapotpár kombinációkat tartalmazza. A két algoritmus pszeudokódja ekkor a következő:

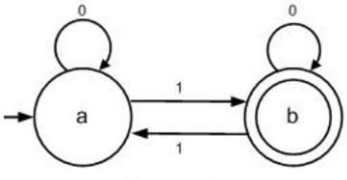
```
for each state a in Qa
   for each state b in Qb
      for each symbol r in Σ
         Tp[(a, b), r]=(Ta[a, r], Tb[b, r]);
```

```
repeat the following
{
 remove a state (a,b) from R;
 for each symbol r in Σ do the following
   {
    if (Ta[a,r], Tb[b,r]) is not in Mp
        add (Ta[a, r],Tb[b, r]) to Mp and to R; //end if
    add to Mp a transition on r from (a,b) to (Ta[a,r],Tb[b,r]);
   } //end for-each
}
until R is empty; //end repeat-until
```

29. ábra - Szorzat állapotgép előállítása egyszerű (balra) és összetett (jobbra) algoritmussal [12]

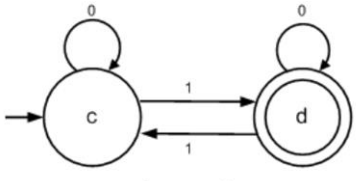
Megjegyzendő, hogy az állapotgépek szorzatának előállítása alatt tipikusan két művelet valamelyike, az *állapotgépek metszetének* és *állapotgépek uniójának* előállítása értendő. Az állapotgépek metszete esetében az elfogadó állapotok azok, amelyek mindkét állapotgépen elfogadó állapotok, az állapotgépek szorzatának esetében pedig azok, amelyek a kettő közül valamelyik állapotgépen elfogadó állapotok. Formálisan az $M_a = \{Q_a, \Sigma, \delta_a, s_a, F_a\}$ és $M_b = \{Q_b, \Sigma, \delta_b, s_b, F_b\}$ állapotgépek esetén az unió művelet eredménye $M_p = \{Q_a \times Q_b, \Sigma, \delta_p, (s_a, s_b), F_a \times \{F_b \cup \emptyset\} \cup \{F_a \cup \emptyset\} \times F_b\}$, míg a metszet művelet eredménye $M_p = \{Q_a \times Q_b, \Sigma, \delta_p, (s_a, s_b), F_a \times F_b\}$.

Legyenek M_1, M_2 és M_3 állapotgépek a 30. ábra szerint adottak, melyek közül M_1és M_2 azonos nyelvet fogadnak el (csupán állapotaik megnevezésében térnek el).



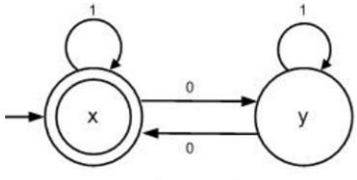
	0	1
a	a	b
b	b	a

$M_1 = (\{a, b\}, \{0,1\}, \delta_1, a, \{b\})$



	0	1
c	c	d
d	d	c

$M_2 = (\{c, d\}, \{0,1\}, \delta_2, c, \{d\})$

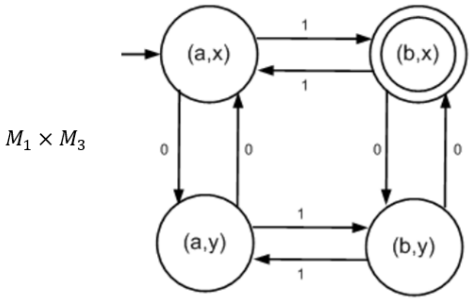


	0	1
x	y	x
y	x	y

$M_3 = (\{x, y\}, \{0,1\}, \delta_3, x, \{x\})$

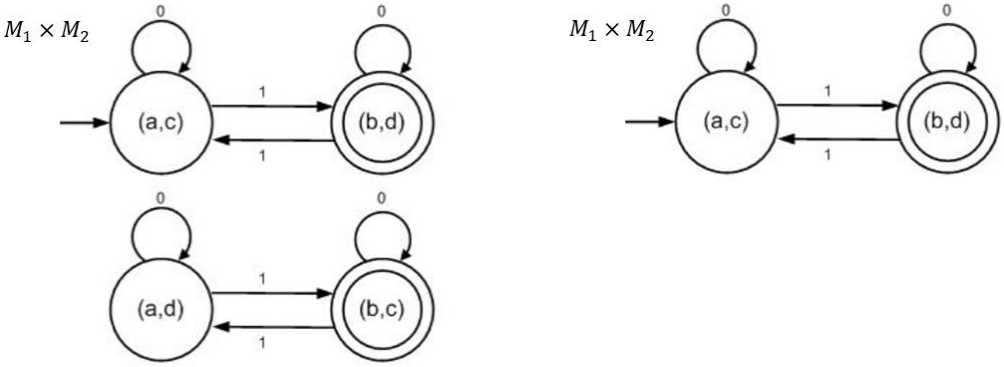
30. ábra - Példa állapotgépek [12]

Ekkor bármelyik algoritmus használatával az $M_1 \times M_3$ helyesen és fölösleges csúcsok nélkül előáll (lásd 31. ábra). $M_1 \times M_2$ előállításakor azonban az egyszerűbb algoritmus használatakor elérhetetlen csúcsok jelennek meg, melyek az összetettebb algoritmus esetében nem (lásd 32. ábra).



	0	1
(a,x)	(a,y)	(b,x)
(a,y)	(a,x)	(b,y)
(b,x)	(b,y)	(a,x)
(b,y)	(b,x)	(a,y)

31. ábra – Mindkét algoritmus esetén keletkező helyes szorzat állapotgép [12]



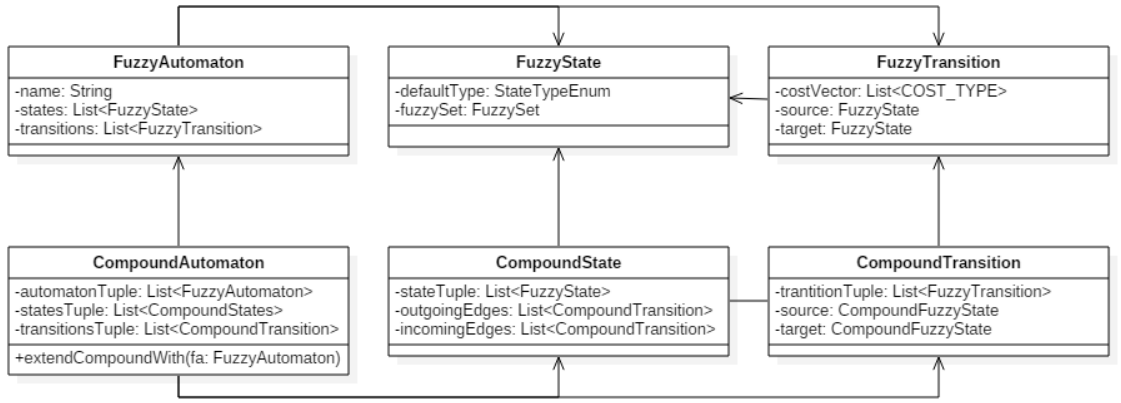
32. ábra - Egyszerűbb algoritmus esetén elromló állapotgép (balra), melyet az összetettebb algoritmus nem eredményez [12]

A szakterületen használt állapotgépek bár matematikailag ekvivalensek az itt bemutatott állapotgépekkel, rendelkeznek néhány speciális tulajdonsággal, melyek a fenti állapotgépek esetében a szokásostól némileg másképp formalizálandók. Ilyen eltérések:

- a fuzzy állapotgépek felett cselekvések értelmezettek, nem pedig egy ábécé, azaz nincsenek „inputok” a fenti állapotgép értelmében, egy cselekvés pedig egyértelműen meghatározza a kezdő és célcsúcsot, szemben az ábécé egy karakterével, mely több csúcs közötti átmenethez is tartozhat
- egy fuzzy állapotgépek típuson belül eltérhetnek a kiinduló és a végállapotok a feladat inputjainak függvényében (minden épület más-más kiinduló állapotban van és más lehet a felújítás célja is), tehát nincs dedikált kiinduló állapot, sem célállapotok, ezek a feladat megadása során kerülnek kijelölésre egy állapotgép sablon típus felett

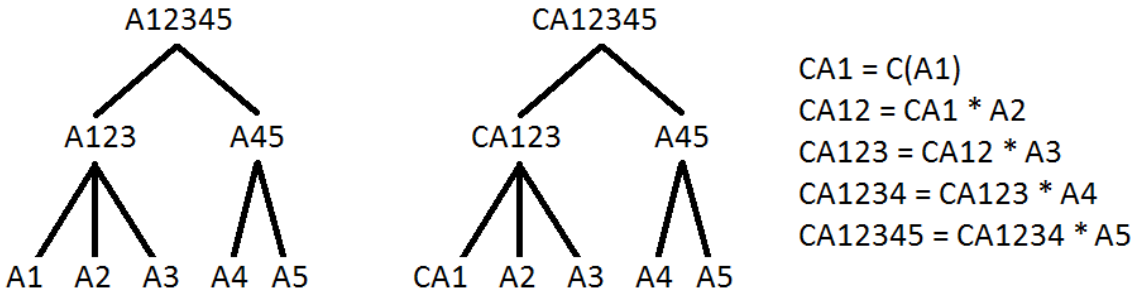
- az átmeneteken az inputok helyett „akciók”, valamint hozzájuk tartozó többdimenziós költségvektorok vannak

Az általam implementált algoritmus a 29. ábra jobb oldalán levő összetettebb algoritmusnak egy módosított változata, mivel illeszkednie kell a szorzat állapotgép általam definiált adatszerkezetére, amely a következő:



33. ábra - Fuzzy állapotgép adatszerkezetének burkolása szorzat eredményének

A 33. ábra látható ahogy az egyszerű fuzzy állapotgépet leíró (amely a *fuzzy automaton* modulban található) adatszerkezetből a szorzat eredményét (és többszörös szorzás részeredményeit) tárolni képes összetett adatszerkezeté bővül (melyet már a *fuzzy signature* modul tartalmaz). A *Compound*-al jelölt adatszerkezetek több *Fuzzy*-val jelölt adatszerkezet átfogására képesek úgy, hogy közben megőrzik és kibővítik az egyszerű állapotgép tulajdonságait. Egy *CompoundAutomaton* például *automatonTuple* adatszerkezetében tárolja azokat az egyszerű állapotgépeket, amelyek a szorzás során előállították azt. Az *extendCompundWith()* metódusa segítségével képes egy N darab állapotgép szorzatából álló állapotgépet a paraméterül kapott állapotgéppel egy N+1 szorzatból álló állapotgépre kiterjeszteni (34. ábra).



34. ábra – Az állapotgépek szorzatának előállítása az implementációban, balra a szignatúra szerinti, jobbra a megvalósítás szerinti sorrendben,
ahol a * az *extendCompundWith()* metódus hívását jelöli a * után következő paraméterrel

A fentiek miatt egy szignatúra adott részfájához tartozó összes állapotgép szorzata úgy kapható meg, hogy a legbaloldalibb levelet *CompoundAutomaton*-ná alakítva (*Compund* adatszerkezet konstruktor hívása), majd lépésenként mindig egy korábban nem vett legbaloldalibb levél *FuzzyAutomaton*-jával a meglevő *compound* adatszerkezetet beszorozva (az *extendCompundWith()* metódussal) végül a fuzzy szignatúra adott részfájához tartozó szorzat állapotgep áll elő.

```
// lehetséges kezdőpontpárok meghatározása
toProcessPairList = []
for (node1 in aut1)
    for (node2 in aut2)
        if (node1 == INITIAL and node2 == INITIAL)
            toProcessPairList.add(<node1, node2>)

transitionsBuffer = []
alreadyProcessedCompoundStates = []
while (toProcessPairList != empty)

    <n1, n2> = toProcessPairList.getAndRemoveFirst()
    compoundTransitionList = {null} + n1.getOutgoingTransitions()
    fuzzyTransitionList    = {null} + n2.getOutgoingTransitions()

    sourceCompoundState = alreadyProcessedCompoundStates.search(<n1, n2>)
    if (sourceCompoundState == null)
        // <n1,n2> not in alreadyProcessedCompoundStates
        alreadyProcessedCompoundStates.add(<n1,n2>)

    for (cTrans in compoundTransitionList)
        for (fTrans in fuzzyTransitionList)

            if((cTrans,fTrans) == (null, null))
                continue

            cTarget1 = ( cTrans == null ? n1 : cTrans.getToState())
            fTarget2 = ( fTrans == null ? n2 : fTrans.getToState())

            targetCompoundState = alreadyProcessedCompoundStates.find(
                <cTarget1, fTarget2>)
            if (targetCompoundState == null)
                targetCompoundState = new(<cTarget1, fTarget2>)
                toProcessPairList.add(targetCompoundState)
                alreadyProcessedCompoundStates.add(targetCompoundState)

            transToCreate =
                <sourceCompoundState, targetCompoundState, cTrans, fTrans>
            if (transToCreate not in transitionsBuffer)
                transitionsBuffer.add(transToCreate)

processedCompoundTransitions = []
for (<sourceCompoundState, targetCompoundState, cTrans, fTrans>
        in transitionsBuffer)

    newTransition = combineToCompoundState(
        <sourceCompoundState, targetCompoundState, cTrans, fTrans>)
    sourceCompoundState.setOutgoingEdge(newTransition)
    targetCompoundState.setIncomingEdge(newTransition)
    processedCompoundTransitions.add(newTransition)
```

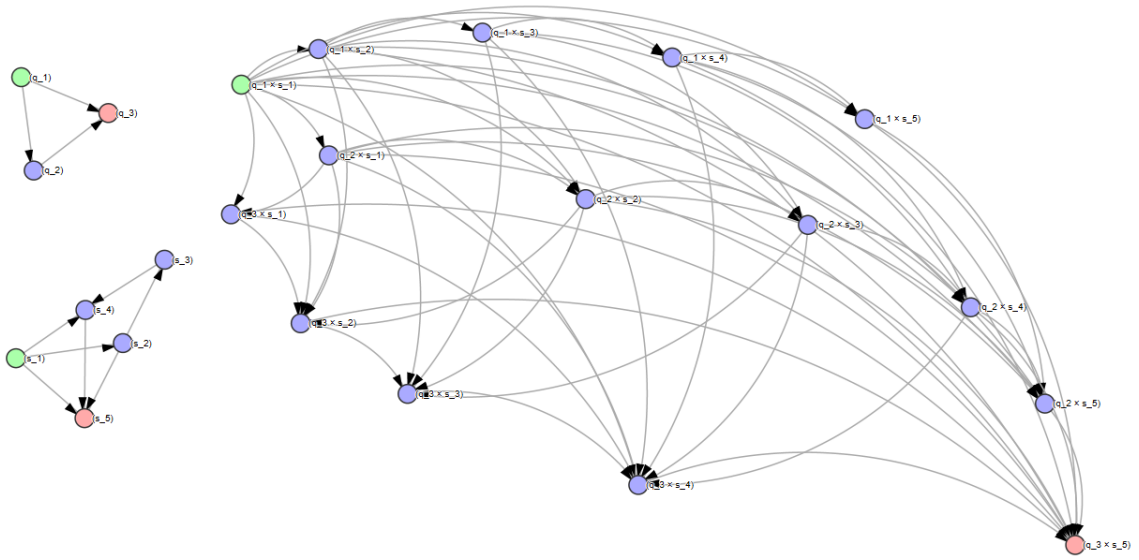
35. ábra - A szorzást végző algoritmus pszeudokódja

A fenti adatszerkezetekre és az összetettebb szorzást végző finomított algoritmust alapul vevő implementáció pszeudokódja a 35. ábrán látható. A kódban a lehetséges összetett kezdőállapot párok hozzáadásra kerülnek egy listához. Majd a lista elemeit egyenként feldolgozva egy állapotpár összes kimenet kombinációja megvizsgálásra kerül. Amennyiben a kimenet kombinációhoz tartozó összetett célcsúcs még nem került feldolgozásra, akkor hozzáadásra kerül a feldolgozandók listájához. Ezt követően az aktuálisan vizsgált élhez tartozó átmenet csúcsai és átmenet információi is hozzáadásra kerülnek egy másik feldolgozási listához, majd végül új átmenet létrehozásával az átmenetek és csúcsok adatszerkezetei is kitöltésre kerülnek.

5.1.3 Állapotgépek szorzatainak tulajdonságai

A számítási modell lényegi részét képezi az állapotgépek szignatúrák leveleiben történő felhasználása. Ennek oka az a törekvés, hogy az egyes folyamatok lehetőleg önmagukban, más folyamatoktól függetlenül kerüljenek definiálásra a különböző részstruktúrákon, de kiértékelésük és az egész folyamat optimalizációja már a probléma teljes komplexitásában történjen meg. A teljes komplexitáshoz a levél csúcsoktól felfelé haladva szükséges az alsóbb szinten levő állapotgépek szorzatának előállítása.

Ez a művelet amellett, hogy az állapotgép (és az állapottér) méretét jelentősen megnöveli, sajnos még az állapotgépek felett definiált költségekre és azok értelmezésére is komoly hatást gyakorol.



36. ábra - Méretnövekedés szemléltetése
(balra a két összeszorzandó állapotgép, jobbra a szorzás eredménye)

Az állapotgépek felett értelmezett szorzat művelet már két kisméretű állapotgépen alkalmazva is a problématér robbanását eredményezi (ennek szemléltetése látható a 36. ábrán).

A szorzás eredményeként kialakuló költségvektorok pedig a következők szerint értelmezhetők a szorzat állapotgépen: ha az él egy olyan él, amely

- mindkét szorzott állapotgépen egy-egy átmenetet határoznak meg, akkor az új költség: $\langle\left[c_1^{(1)}, \dots, c_k^{(1)}\right], \left[c_1^{(2)}, \dots, c_k^{(2)}\right]\rangle$
- csak az első állapotgépen határoznak meg átmenetet: $\langle\left[c_1^{(1)}, \dots, c_k^{(1)}\right], null\rangle$
- csak a második állapotgépen határoznak meg átmentet: $\langle null, \left[c_1^{(2)}, \dots, c_k^{(2)}\right]\rangle$

ahol a költségvektor komponensének az alsó indexe *k* darab komponens valamelyikét jelöli, felső indexe pedig arra az állapotgéprera utal, amelyből a költségvektor származik.

A szorzás művelet *n*-szer történő alkalmazásával a szorzat állapotgépen az alábbi adatszerkezetekhez juthatunk $\langle\underline{c^{(1)}}, \dots, \underline{c^{(n)}}\rangle$, ahol bármelyik $\underline{c^{(i)}} = \left[c_1^{(i)}, \dots, c_k^{(i)}\right]$ költségvektor lehet üres, de legalább egy költségvektor adott.

Az implementációban fenti adatszerkezetre leginkább illeszkedő *List<List<COST_TYPE>>* típus került alkalmazásra.

5.1.4 Szorzat állapotgépek szakterülethez illesztése

A korábbiak értelmezésében a szignatúra gyökerén levő állapotgép olyan költségvektorokkal definiált, melynek élein $\langle\underline{c^{(1)}}, \dots, \underline{c^{(n)}}\rangle$ felépítésű költségvektor sorozatok találhatóak, amelyek egyetlen összetett lépést írnak le az szorzat állapotgépen. Ez a szorzat állapotgép egy átmenetén értelmezett egyetlen lépés különböző lépéseket kódolhat egy vagy több részállapotgépen, melyeknek így egyidejű (párhuzamos) végrehajtását írja elő.

Ilyen átmenetek esetén definiálható kétféle aggregáció, méghozzá az **élen belüli aggregáció** $\langle\underline{c^{(1)}}, \dots, \underline{c^{(n)}}\rangle$ felett, amely egy összetett lépéshez rendel valamiféle aggregált költséget, valamint **élsorozatokon belüli aggregáció** $[\langle\underline{c^{(1)}}_1, \dots, \underline{c^{(n)}}_1\rangle, \dots, \langle\underline{c^{(1)}}_l, \dots, \underline{c^{(n)}}_l\rangle]$ felett, ahol $\underline{c^{(j)}}_i$ határozza meg az *l* hosszú

szekvencia *i*-edik élének *j*-edik állapotgépből származó, adott tranzícióhoz tartozó költségvektorát.

Az implementációban az éleken belüli aggregációs függvény definiálásához bemenő paraméterként egy *List<List<COST_TYPE>>* típusú adatszerkezet a bemenő paramétervektor, míg élsorozatokon belüli aggregációs függvény definiálásához már *List<List<List<COST_TYPE>>>* típusú bemenő paramétervektor feldolgozása szükséges.

Jelen implementációban néhány lehetséges függvény hardcode[24]-olása történt meg, de a továbbiakban érdemes lehetőséget biztosítani a felhasználó által tetszőlegesen definiálható aggregációs függvények megadására is, mind éleken belüli, mind pedig élsorozatokon belüli kiértékeléshez.

Itt megjegyzendő, hogy az akciók párhuzamos ($pl. \langle \underline{c^1}, \underline{c^2} \rangle$ költséget tartalmazó él) és szekvenciális (pl. $\langle \underline{c^1}, null \rangle$ majd $\langle null, \underline{c^2} \rangle$ költségeket tartalmazó élsorozat) végrehajtása között szakterülettől függően nem feltétlenül áll fenn egyenlőség (tehát nem feltétlenül érvényes az $f\left(\langle \underline{c^1}, \underline{c^2} \rangle\right) = f\left(\langle \underline{c^1}, null \rangle\right) + f(\langle null, \underline{c^2} \rangle)$ additív tulajdonság).

A fentiek egy szemléletes példája, ha az első komponensként a pénzügyi költséget, második komponensként a ráfordításra szükséges időt kódoljuk. A gyakorlatban ez a szorzat állapotgépben egy 2 felújítási lépést (nem *null* költséggel) tartalmazó összetett lépés párhuzamos feldolgozását jelenti, tehát az élen belüli aggregáció ekkor *c = sum(c1, c2)* és *t = max(t1, t2)*. Azonban ha a szorzat állapotgépben csak két átmenet felhasználásával kivitelezhető ugyanez a folyamat, akkor *c = sum(c1, c2)* és *t = sum(max(t1, null), max (null, t2)) = sum(t1, t2)*. A fentiekből látható, hogy miért nem alkalmazható ugyanaz az aggregációs függvény az élsorozatokon belüli (azaz az élek közötti), valamint az élen belüli esetre. Az implementációban csak néhány függvény lett a költségszámításokhoz hardcode-olva (pl. *max*, *min*, *sum*, *prod* stb.) demonstrációs célból, de természetesen bármilyen típusú költségszámítás értelmezhető a fenti adatok felett.

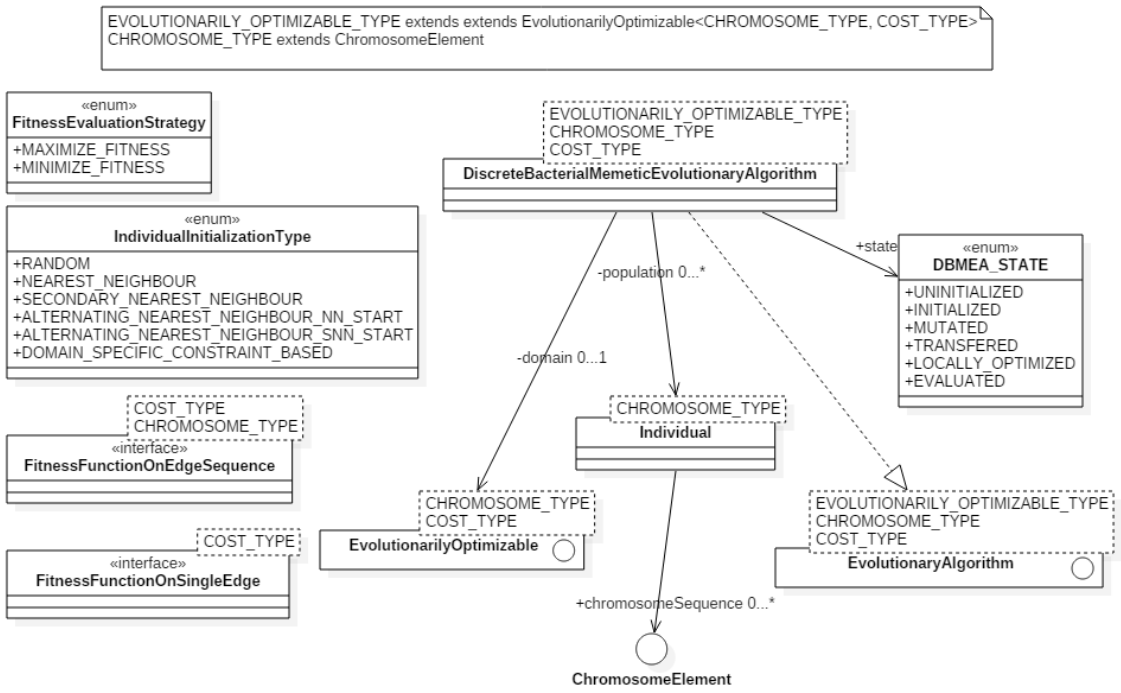
Továbbá szintén megjegyzendő, hogy a gyakorlatban a szorzat állapotgépben előfordulhat tiltott párhuzamosítás is, tehát amikor két folyamat egyszerre nem

[24] hardcoded – kódban rögzített, nem változtatható működés, érték stb.

kivitelezhető. Ezért a továbbiakban érdemes lehet szabály alapú szűrést biztosító logikát alkalmazni a szorzat állapotgép éleire, melyeket alacsony szintű állapotgépek csúcsai felett lehet értelmezni.

5.1.5 Bakteriális algoritmus keretrendszerként való megvalósítása

Korábbi fejezetekben már volt szó arról, hogy az evolúciós algoritmus fuzzy számítási modellhez történő illesztése többféleképp is megtörténhetett volna. Egyik lehetőség szerint az evolúciós optimalizálást tartalmazó modul a fuzzy szignatúra modul által tartalmazott *Compound Automaton* adatszerkezetét közvetlenül ismerte és használta volna, ezáltal az evolúciós optimalizációt végző modulnak közvetlen függősége lett volna a fuzzy szignatúra modulja. Mivel generikusabb kialakítással az optimalizációs modulnak nem kell ismernie az optimalizálandó adatszerkezetet, ezért végül az a tervezői döntés került megavalósításra, mely szerint a bakteriális algoritmus a saját moduljában egy független keretrendszer formájában kerüljön megvalósításra. Az optimalizációhoz szükséges feltételek, a keretrendszer igénybevétele így még csak nem is feltétlenül a szignatúrákat tartalmazó modulból, hanem a rá épülő szakterületi modulból is kielégíthetők, ezáltal csökkentve a számítási modulok közötti függőségeket.



37. ábra - Optimalizációs keretrendszer felépítése generikusokkal (részlet)

A 37. ábra demonstrálja, hogy a típuskényszereken keresztül hogyan alakítható ki egy evolúciós algoritmussal optimalizálható számítási modell. Az ábrán csak a főbb osztályok kerültek feltüntetésre és a továbbiakban csak körvonalakban kerül összegzésre.

Az *EvolutionaryAlgorithm* interfész előír néhány evolúciós optimalizálásra jellemző viselkedést, amelyeket a *DiscreteBacterialMemeticEvolutionaryAlgorithm* (DBMEA) osztály megvalósít és továbbiakkal ki is terjeszt. Ez az osztály képes az *EvolutionarilyOptimizable* interfész bármely megvalósítását felhasználni optimalizációhoz, melyek felett értelmezi a populációt *Individual* példányok összességeként. Ez utóbbiak jelölik az egy kromoszómával kódolható probléma (rész)megoldások leírásait, melyeket a *ChromosomeElement*-ek szekvenciáin keresztül a domain területet megvalósító *EvolutionarilyOptimizable* szolgáltat az adott probléma konkrét leírására.

Megjegyzendő, hogy a 37. ábrán látható interfészek és osztályok terjedelmi korlátok miatt csak elnagyolva kerültek ábrázolásra, valós implementációjuk ennél mélyebb. Az *EvolutionarilyOptimizable* implementációi biztosítják, hogy egy adott domain terület megvalósítson olyan műveleteket, amelyeket egy *EvolutionaryAlgorithm* interfészt megvalósító osztály az optimalizáció során fel tud használni (pl. populáció inicializálása, egy élhez, vagy élsorozathoz a domain terület szerinti költségvektor definiálása, probléma leírásának kódolása és dekódolása stb.).

Tehát a fuzzy szignatúrák korábban ismertetett *CompoundAutomaton*-jai az *EvolutionarilyOptimizable* megvalósításaként kódolhatják az adott probléma leírását (maga az összeszorzott állapotgép modell, tranzícióin költségvektorok sorozatával, kiinduló és célállapot stb.), a *CompoundState*-ek pedig a *ChromosomeElement* megvalósításiként sorba állítva kódolják a probléma egy adott (rész)megoldásának leírásait, melyek sorrendje egyben meghatározza az állapotok közötti átmenetekhez tartozó költségvektor sorozatok (mely egy összetett él költsége) sorozatát is (mely ilyen élek sorozatának költsége).

Mivel a korábbi alfejezetekben vázoltak szerint a költségvektor sorozatokhoz (*List<List<COST_TYPE>>* egy él esetén), valamint a költségvektor sorozatok sorozataihoz (*List<List<List<COST_TYPE>>>* élsorozat esetén) tartozó fitnesz érték kiszámítása erősen szakterület specifikus, ezért a keretrendszer ilyen adatszerkezetek fitneszének meghatározását, azaz *double* típusú skalár értékre való transzformációk megadásának módját várja el az paraméterként. A Java 8 erre lehetőséget biztosít a

paraméterként átadható **metódus referencia** által. Ez alapján *FitnessFunctionOnSingleEdge* és *FitnessFunctionOnEdgeSequence* interfészek egyetlen metódusainak szignatúráira, valamint a DBMEA konstruktorában megadott COST_TYPE típusára illeszkedő szignatúrájú metódus referenciájának átadása szükséges a DBMEA konstruktorának, hogy populáció egyedeinek kiértékelése megtörténhessen.

Megjegyzendő, hogy annak az oka, hogy a fitnesz kiértékelésének kétfajta megvalósítása létezik (élek kiértékelésén és élsorozatok kiértékelésén keresztül) arra is visszavezethető, hogy számos, a bakteriális algoritmusban alkalmazott részfeladat megoldásához nem csupán a teljes kromoszóma (esetünkben N hosszú állapotsorozatok, ahol N az összetett állapotgép összes állapotának száma), hanem egy-egy él kiértékelésére is szükséges lehet. Ilyen például a Nearest Neighbour stratégia alkalmazása a populáció inicializációja során, ahol a szekvencia kialakításakor a következő kiválasztott csúcs a költségtől függ, azaz a szekvencia meghatározásának folyamatában egy élre és nem a még el nem készült teljes szekvenciára szükséges a fitnesz meghatározása. További műveletek (pl. bakteriális mutáció) esetén azonban nem értelmezett az egyetlen él kiértékelésének művelete, hanem ekkor a fitnesz meghatározása a teljes kromoszómán kell megtörténjen. További észrevétel, hogy ennek a kettős kialakításnak köszönhetően az élsorozat kiértékelésére szánt függvényben (bár nem kötelezően, de) felhasználható a fitnesz élenkénti kiértékelésének megvalósítása is, vagy akár jóval komplexebb kiértékelő függvények is definiálhatók, mint ami az egyszerű élenkénti kiértékelések sorrendszerű végrehajtásával kapható lenne. A 38. ábra egy sablont mutat a kiértékelő függvények szignatúrájára.

```
public interface FitnessFunctionOnSingleEdge<COST_TYPE> {

    Double calculateOnSingleEdge(COST_TYPE cost);
}

public interface FitnessFunctionOnEdgeSequence<
    COST_TYPE, CHROMOSOME_TYPE extends ChromosomeElement> {

    double calculateOnEdgeSequence(Individual<CHROMOSOME_TYPE> individual,
       FitnessFunctionOnSingleEdge<COST_TYPE> chromosomeElementCostFunction,
       EvolutionarilyOptimizable<CHROMOSOME_TYPE, COST_TYPE> evolutionarilyOptimizable);

}
```

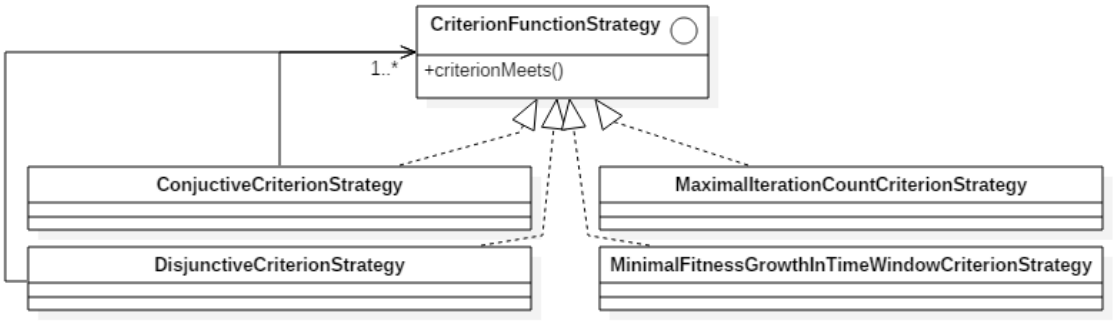
38. ábra – Kétfajta fitneszfüggvény elvárt megvalósításainak szignatúra sablonja

Elengedhetetlen továbbá a fitnesz stratégia megválasztása a *FitnessEvaluationStrategy* enum értéke által, mellyel kiválasztható, hogy a nagyobb vagy a kisebb fitneszű példányok legyenek az optimalizáció szempontból a kedvezőbbek.

A populáció inicializációjához egy „inicializációs terv” adható meg a keretrendszernek, mely szerint egy *Map<IndividualInitializationType, Integer>* típus kitöltésével előírható, hogy hány példányt milyen stratégia szerint készítsen el az *EvolutionarilyOptimizable* egy példánya. Megemlítendő, hogy az irodalomkutatás során megjelölt stratégiákon túl (*Random, NearestNeighbour, SecondaryNearestNeighbour, AlternatingNearestNeighbour*) további két stratégia használata lehetséges. Az egyik egy olyan *AlternatingNearestNeighbour* amely az első él kiválasztásakor nem a legjobb, hanem a második legjobb fitneszűvel kezd, azaz az eredetileg definiáltal pont fordított fázisban alternál, a másik pedig a *DomainSpecificConstraintBased*, amelyet a szakterületi modell szabadon definiálhat. A *CompoundAutomaton* esetében például érdemes lehet olyan állapot sorozatokat előállítani, amelyek között garantáltan vannak élek, azaz a véletlenszerűvel ellentétben nincs bennük szakadás, vagyis olyan két egymást követő csúcs a kormoszómában, amely a szorzat állapotgépben vagy egyáltalán nincsenek irányított éllel összekötve, vagy a két csúcs között nem létezik a kromoszómabeli sorrendjük szerinti irányítottságnak megfelelő irányú átmenet.

A DBMEA példányán hívható publikus metódusok egy része megfelel a bakteriális algoritmus lépésenként végrehajtott műveleteinek (populáció inicializálása, koherens bakteriális mutáció, laza bakteriális mutáció, gén transzfer, lokális optimalizáció, populáció kiértékelése, iteráció léptetése stb.). A fenti műveletek egy része (mutáció, gén transzfer, lokális optimalizáció) amellett, hogy alapesetben a teljes populáción kerül végrehajtásra, elérhető olyan formában is, amely csak egy paraméterül kapott egyeden végzi el a műveletet, ezáltal rugalmasabbá téve a keretrendszer használatát. Az populáción végzett műveletek lépésenkénti végrehajtása megváltoztatja az objektum *DBMEA_State* állapotát, továbbá minden lépés végrehajtása előtt ezen az állapoton keresztül ellenőrzésre kerül, hogy a végrehajtáshoz szükséges lépések megtörténtek-e már korábban. A fenti műveleteket nem feltétlenül szükséges lépésenként hívni, ugyanis lehetséges a teljes optimalizáció automatizált futtatása alapértelmezett beállítások mellett, amennyiben a keretrendszer használója nem kívánja konfigurálni az evolúciós algoritmus finombeállításait.

Amennyiben a keretrendszer automatikus optimalizálással kerül futtatásra, tehát a felhasználó nem konfigurálja egyenként a végrehajtás minden lépését, hanem az alapműködést szeretné minimális mértékben befolyásolni, akkor megadható egy leállási feltétel is. A leállási feltétel kiértékelése a **Strategy**[25] **tervezési minta** megvalósítására épül. A leállási feltétel lehet akár egy összetettebb feltétel is, mely logikai és/vagy (konjunkció és diszjunkció) műveletek felhasználásával kialakítható a *CriterionFunctionStrategy* interfészből történő leszármaztatással. Az automatizáláshoz két tipikusan használt kiértékelési függvény került elkészítésre, a *MaximalIterationCountCriterionStrategy*, amely egy adott számú iteráció után leállítja az optimalizációt, valamint a *MinimalFitnessGrowthInTimeWindowCriterionStrategy,* amely akkor teljesül, ha egy adott iterációszám alatt a fitnesz értéke egy minimálisnál nagyobb mértékben nem képes javulni.



39. ábra - Strategy design pattern alkalmazása leállási feltétel vizsgálatához

A 39. ábrán látható, hogy a konjunkciót és diszjunkciót megvalósító osztályok hogyan képesek több kiértékelési feltételt is egymáshoz fűzni.

5.1.6 Kromoszómák kódolása

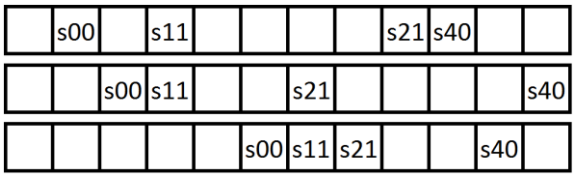
Az irodalomkutatás fejezetében összefoglaltak szerint egy probléma kromoszómába történő kódolása TSP problémák esetén viszonylag egyszerű kódolással történik, ott ugyanis pont ugyanannyi állapotot kell hivatkozni a kromoszómában, mint amennyi elem található a gráfban és a kromoszóma hossza is megegyezik az állapotok számával (esetleg eggyel kevesebb, ha a kiindulási pont adott). Ennek oka, hogy a kromoszóma az állapotok bejárási sorrendjét határozza meg Hamilton út/kör keresése

[25] Strategy Design Pattern – Egy olyan viselkedési típusú tervezési minta, ahol akár futás időben is kicserélhető a végrehajtó logikát megvalósító algoritmus.

esetén, amely szintén pontosan ugyanannyi csúcsot kell kódoljon, mint amennyi a gráf csúcsainak a száma.

Más a helyzet azonban a legkisebb költségű útvonal megkeresésénél, főként abban az esetben, ha az adott gráfban a kiinduló és a célállapot is könnyen változhat (szakterületi optimalizációs céltól függően). Általános esetben ugyanis egy kromoszómában nem feltétlenül kell annyi csúcsot hivatkozni, mint amennyi pont létezik a gráfban. A csúcsok sorrendjének kódolására ilyenkor két lehetőség is kínálkozik, a **változó hosszúságú kromoszómák**, valamint a **maximális fix hosszúságú kromoszóma kitöltetlen helyekkel**, majd fitnesz kiértékelés előtti szűréssel. [13][14]

Mivel a változó hosszúságú kromoszómák használata több olyan kérdést is felvet, amelyek módosíthatják az evolúciós algoritmus műveleteinek implementációit, ezért az implementációban a fix hosszú kromoszómák üres helyekkel történő feltöltésének változata került megvalósításra, mivel ez esetben nem szükséges az optimalizációs műveletek helyességének garantálása vagy a lefutásuk eredményeinek ellenőrzése.



40. ábra - Néhány példa azonos fitneszű kromoszómára

Megjegyzendő, hogy ez esetben a kitöltetlen üres hely nem egy új állapotot kódol, ezek kiértékelés előtt kiszűrendők, így egy konkrét feladatleírásnak általános esetben több azonos fitneszű, de különböző leírású reprezentációja is létezhet (40. ábra), a kiértékeléskor csupán a gének sorrendje számít a kromoszómán belül. Ez a fajta reprezentáció csak minimális mértékben befolyásolja a korábbiakban leírt optimalizációs műveleteket megvalósításait. Ilyen minimális változtatás például, hogy a RANDOM típusú egyed inicializáció során egy valószínűségi érték alapján üres érték is generálható, nemcsak a fel nem használt csúcsok közül lehet egyet választani.

5.1.7 Kromoszómák inicializálása

A kromoszómák hasznos tartalmának kialakítása kétfajta megközelítésben történhet attól függően, hogy véletlenszerűen (*Random*), vagy útvonal alapon (*Nearest Neighbour*, *Secondary Nearest Neighbour*, *Alternating Nearest Neighbour*) kellett az adott kromoszómát inicializálni.

Véletlenszerű inicializálásnál a kromoszóma elemeinek meghatározásakor egy paraméterként megadható valószínűség alapján egy adott pozícióhoz véletlen sorsolással kerül eldöntésre, hogy üres állapot, vagy a még fel nem használt csúcsok valamelyikéből kerüljön kiválasztásra a kromoszóma adott eleme, majd ez addig ismétlődik, amíg az összes elem meghatározásra nem kerül.

A kromoszómák útvonal szerinti inicializálása a szakterületi modellben két lépcsőben történik. Első lépésben rekurzív algoritmusok segítségével a kromoszómák hasznos tartalmai kerülnek meghatározása, második lépésben pedig a szükséges üres helyek számával egyenlő, véletlen pozíción történő üres érték beszúrása történt meg.

5.1.8 Kromoszómák kiértékelése

A fentiek alapján a kromoszómák a szorzat állapotgép állapotai mellett üres elemeket is tartalmazhatnak. Ez utóbbiaknak nincs jelentőségük a feladat szempontjából, csupán a kromoszóma kitöltésére szolgálnak, hogy az optimalizációs algoritmusokat ne kelljen nagyobb mértékben módosítani, így tehát ezeknek a fitnesz kiértékelésekor sincs különösebb jelentőségük.

A kromoszómák azonban a *null* elemek szűrése után is tartalmazhatnak „ugrásokat”, vagy „szakadásokat” abban az értelemben, hogy a kromoszóma két szomszédos eleme által jelölt csúcsok között nem fut él, vagy a köztük futó él irányítottsága nem illeszkedik a kromoszómabeli sorrendre. Azonban a fitneszt kiértékelő metódusoknak (egy él vagy egy élsorozat fitneszeit meghatározó metódusoknak) ekkor is költségeket tartalmazó vektorok listáját, vagy vektorok listájának listáját kell kiértékelniük, amely ez utóbbi esetben egy N+1 pontból álló N hosszú élsorozat esetén (mely a kromoszómában N darab nem üres csúccsal kódolt) esetén N hosszú listát jelent.

A legrövidebb út optimalizációjára vonatkozó cikkek [13][14] szerint a szakadásokat ebben a költségvektor sorozatban érdemes egy kimagaslóan nagy költséggel számolni. Azonban feltehető, hogy minél jobb „felbontású” a fitnesz függvény által szolgáltatott információ és minél diverzebbek a populáción belüli fitneszek értékek, annál gyorsabb lesz a konvergencia a keresett optimumhoz, emiatt valószínűleg nem érdemes minden szakadást kódoló kromoszómához ugyanazt a költséget rendelni és talán érdemes lehet a „szakadások hosszát” is értelmezni két csúcs között.

A fitnesz felbontása úgy finomítható tovább, valamint a populáció fitneszei úgy diverzifikálhatók, hogy két csúcs közötti „**szakadás hossza**” kerül beszorzásra az előre

definiált költséggel, melyek a kromoszóma mentén akkumulálódnak, így különbséget téve a szakadásokat kódoló kromoszómák fitneszei között. Amennyiben két állapot nem közvetlenül szomszédos, megkereshető annak a legrövidebb élsorozatnak a hossza, amelyen keresztül az első állapotból el lehet jutni a második állapotba. Amennyiben nincs ilyen élsorozat, akkor az élsorozat hossza helyett alapul vehető az állapotgép éleinek a száma, mint maximális korlát.

A szakadások hosszához szükséges legrövidebb utak hossza több módszerrel is megállapítható. A következő két egymással ekvivalens eredményt előállító algoritmus került implementálásra:

- **Dijkstra algoritmusa**, mely egy adott csúcshoz képes meghatározni az összes többi csúcshoz vezető legrövidebb költségű utat (nemnegatív élsúlyok esetén). Mivel ez esetben az élek számában mért távolság jelenti a „szakadás hosszát”, ezért minden él súlya lehet konstans egy, ezáltal az algoritmus helyes eredményt ad.
- **Módosított szélességi bejárás** (BFS[26]) algoritmus, amely két listát kezel, egy a már feldolgozottak és egy még a feldolgozandó állapotok listáját. Kezdetben a feldolgozandó listához hozzáadásra kerül a kiindulópont. Az algoritmus a feldolgozandó csúcsok listájából veszi ki mindig a legelső elemet, majd annak szomszédjait megvizsgálva a még egyik listában sem levő elemet hozzáadja a feldolgozandókhoz. A feldolgozás során az elemeket kiveszi a feldolgozandók listájából, majd hozzáadja a feldolgozottak listájához. Ezt kiegészítve egy olyan lépéssel, amely egy állapot szomszédainak feldolgozása során egy adatszerkezetből kiolvassa a szomszéd állapothoz tartozó eddig talált legrövidebb utat, majd visszaírja az éppen feldolgozandó állapothoz tartozó legrövidebb út hosszának eggyel növelt értékének és a szomszéd állapothoz tartozó kiolvasott értékének minimumát. Végül az összes állapot feldolgozott listába kerülése után visszaadja az adatszerkezetben a keresett állapothoz tartozó eltárolt minimális értéket.

[26] BFS – Breadth First Search Algorithm – fabejárási algoritmus

```
toProcessList = []
processedList = []
resultMap = []

toProcessList.add(INIT_STATE)
resultMap.add(<INIT_STATE, 0>)

while (toProcessList != empty)

    elementToProcess = toProcessList.pop()
    shortestPathToProcessedElement = resultMap.get(elementToProcess)

    for (neighbour in elementToProcess.getNeighbours())

        if (not toProcessList.contains(neighbour)
            and not processedList.contains(neighbour))
            toProcessList.add(neighbour)

        shortestPathToNeighbour = resultMap.get(neighbour)

        if (shortestPathToNeighbour == null)
            resultMap.add(<neighbour, shortestPathToProcessedElement + 1>)
        else
            minDistance = min(
                shortestPathToProcessedElement + 1, shortestPathToNeighbour)
            resultMap.add(<neighbour, minDistance>)

    processedList.add(elementToProcess)

return resultMap.get(TARGET_STATE)
```

41. ábra - Módosított BFS pszeudokódja

A 41. ábrán látható a BFS-en alapuló megoldás pszeudokódja. Mivel a kezdőállapotból kiinduló állapotgép szorzásai során nem alakulhat ki független komponenseket tartalmazó erdő alakú állapotgép, ezért az a módosított BFS az összes kiindulópontból bejárható állapotot felderíti.

A BFS-en alapuló megoldás gyorsabb, mivel lépésszáma $O(|V| + |E|)$, míg a Dijkstra algoritmusé $O(|V|^2)$ naiv implementációk esetén, ahol V a csúcsokat, E pedig az éleket jelöli.

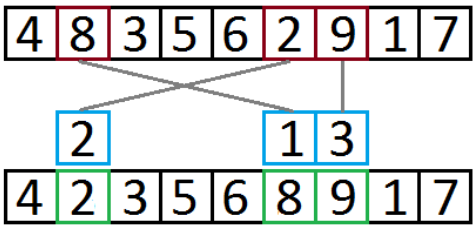
5.1.9 A standard DBMEA kiterjesztései

Az irodalomkutatás részben foglaltakhoz képest helyenként új ötletek is implementálásra kerültek. Ilyen ötlet például a már fentebb említett inicializációs stratégia, valamint egy újfajta mutációs operátor.

Mivel a populáció inicializációjának az *EvolutionaryAlgorithm* megvalósítása által történő meghívása az *EvolutionarilyOptimizable*-t megvalósító osztály segítségével történik, mely a domain feladat optimalizációjának felelőseként számos a szakterületre

specifikus részletet is el kell fedjen, ezért tőle elvárható követelmény, miszerint a szokásos inicializációs stratégián túl egy olyat is megvalósítson, amely a szakterület szempontjából a véletlenszerűnél némileg jobb fitneszekkel indul (*DOMAIN_SPECIFIC_CONSTRAINT_BASED*). Egy ilyen megvalósítás lehet például, hogy a kromoszóma leírásban az egymást követő csúcsok között szükségszerűen fusson él, így kapva egy összekapcsolt véletlenszerű útvonalat az állapotgépen, melyben nincs szakadás. Kellően nagy gráf esetén ezek az elkódolt útvonalak nagy valósínűséggel eltérnek egymástól és az ilyen módszerrel előállított kromoszómák részszekvenciái hasznosak lehetnek más egyedek számára.

További kiterjesztésként a *DiscreteBacterialMemeticEvolutionaryAlgorithm* műveleteként implementálva lett a meglevő mutációs operátorokon túl egy újfajta mutációs operátor (*RandomColumnGroupMutation*), amely a koherens és laza bakteriális mutációk ötvözeteként működik. Ennek az új operátornak a **véletlen oszlopcsoport kiválasztás** az alapja, mely szerint addig történik véletlenszerűen még korábban ki nem választott oszlopok kiválasztása a kromoszómán, amíg egy szegmensméretet elérve már elvégezhető az optimalizáció a kiválasztott oszlopok értékei felett. Az algoritmus addig optimalizál véletlenszerűen kiválasztott oszlopcsoportokat a kromoszóma felett, amíg el nem fogynak a még ki nem választott oszlopok. Míg a koherens bakteriális algoritmusban ezek az „oszlopok” egymás mellettiek így alkotva egy összefüggő szegmenst, a véletlen oszlopkiválasztásnál nincs ilyen kritérium. A laza bakteriális mutációhoz képest az eltérés pedig az, hogy a lehetséges sorrendpermutáció nem a kromoszóma minden értéke felett kerül meghatározásra, hanem csak a kiválasztott oszlopok értékei felett.



42. ábra - Random Column Group mutációs operátor egy lépésének szemléltetése, 3 szegmensméret mellett

A 42. ábrán látható, hogy egy adott iterációban kiválasztásra kerülnek a korábbi iterációkban még ki nem választott oszlopok (2., 6. és 7. oszlopok a 8, 2, 9 értékekkel a felső kromoszómából), majd az ezeken a pozíciókon található értékek új sorrendje kerül meghatározásra (kékkel jelölt indexsorrend). A művelet kimenete az a kromoszóma,

melyben a kijelölt pozíciókon az indexsorrend szerinti véletlen értékek találhatók. A mutációs operátor a korábbiakhoz hasonlóan csak akkor cseréli le az eredeti kromoszómát a populációban, amennyiben annak fitnesze jobb, mint a bemeneti kromoszómáé.

-	s00	-	s11	s21	-	-	s40	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12
-	s00	-	s11	s21	-	-	s40	-	-	-	-
-	s11	-	s00	s21	-	-	s40	-	-	-	-
s11	s00	-	-	s21	-	-	s40	-	-	-	-
s11	-	-	-	s21	-	-	s40	s00	-	-	-
-	s11	-	-	s21	-	-	s40	s00	-	-	-
s00	s11	-	-	s21	-	-	s40	-	-	-	-
-	s00	-	-	s21	-	-	s40	s11	-	-	-
-	-	-	s11	s21	-	-	s40	s00	-	-	-
s00	-	-	s11	s21	-	-	s40	-	-	-	-
s11	-	-	s00	s21	-	-	s40	-	-	-	-
-	-	-	s00	s21	-	-	s40	s11	-	-	-
s00	-	-	-	s21	-	-	s40	s11	-	-	-

1. táblázat – Egy kromoszómához (fent) egy kiválasztási lépésben 4-es oszlopcsoport méret mellett generálható lehetséges klónok

Az 1. táblázaton megfigyelhető, hogy az adott kiválasztási iterációban csak a kiválasztott, zölddel jelölt oszlopokon belüli értékek lehetséges permutációival előálló kromoszómák keletkezhetnek. A sárgával jelölt oszlopok az adott kiválasztási iterációban változatlanok maradnak. Megfigyelhető az is, hogy az üres gének oszlopainak permutációi nem képeznek új lehetőséget, ezek sorrendje ugyanis irreleváns a kromoszóma fitneszének szempontjából.

5.2 Alkalmazás működésének főbb irányelvei

A kertrendszer a moduloktól egy *ModuleDescriptor* interfész implementációját kívánja meg. Ennek legfontosabb eleme az *initializeModule()* metódus, melyben a beépülők inicializációs időben nyilatkozhatnak a regisztrálandó nézeteikről és adatszerkezetikről, valamint saját inicializációs lépéseket hajthatnak végre, melyek szükségesek lehetnek a nézeteik és vezérlőik példányosítása előtt.

A beépülőknek lehetőségük van igénybe venni a keretrendszer által nyújtott szolgáltatásokat. Ilyen szolgáltatás a már korábban említett nézet és adattípus regisztráció

vagy a megszerkesztett modellek közös tárolóba való betöltése, adatok exportálása, importálása stb. Ennek megvalósításához szükséges valamiféle **service loader** funkcionalitás (amely lazán csatolt, egymás osztályait nem ismerő komponensek esetében is lehetővé teszi a szolgáltatások elérését regisztrált típusokon keresztül), vagy valamilyen tipikusan **Inversion of Control** (**IoC**) megvalósítás, például **Dependency Injection** (**DI**) keretrendszer segítségével.

Az OSGi által nyújtott, láthatóságokkal történő operációk által behozott új absztrakciós szint miatt még lazán csatolt, komponens orientált környezetben is szükségtelen olyan eszközökhöz nyúlni, mint DI keretrendszerek.

Bár az OSGi által kínált lehetőségek szükségtelenné teszik a fenti megoldások használatát, sajnos az OSGi keretrendszer alkalmazása során felmerülő nehézségek egyelőre ellehetetlenítették az ilyen lehetőségek kihasználását, ezért egy ideiglenes *ServiceFactory* megoldás került kialakításra az efféle szolgáltatások elérésére. Ennek implementációja a keretrendszer egy publikus névtérben (*common*) került elhelyezésre. A modulok ezáltal könnyen elérhetik az általuk ismert interfészek konkrét implementációit az implementációs osztályok ismerete nélkül.

A *ServiceFactory* egy a szolgáltatásokat implementáló és szálbiztos hozzáférést biztosító singleton[27] osztály (pontosabban annak egy statikus) példányát adja vissza (*CommonServicesImplSingleton*). Ez a megvalósítás illeszkedik a későbbiekben alkalmazni kívánt OSGi-os alapelvekhez, mely elvei szerint elégséges a package láthatóságokkal operálni ahhoz, hogy egy új absztrakciós szinten kezeljük az elrejtendő, vagy exportálandó komponenseinket.

A *shell* által nyújtott közös szolgáltatások közös adatszerkezetek ismeretét és használatát követelik meg a moduloktól. Ilyen adatszerkezetek kitöltésével egy modul deklarálhatja, hogy milyen nézeteket kíván elérhetővé tenni (*FxModulesViewInfo* példány kitöltésével) a keretrendszerben, hogy milyen típusú modelleket szeretne közös térben regisztrálni (*WorkspaceElement*-ből való modell leszármaztatással) vagy, hogy a regisztrált modelltípusokhoz milyen címkék és egyéb információk tartozzanak a felületen és adatainak betöltésénél mely nézetek jelenítsék meg azokat (*WorkspaceInfo* példány),

[27] Singleton / Egyke - tervezési minta, amely garantálja, hogy az osztálynak csakis egyetlen példányához történjen mindig a hozzáférés (pattern és antipattern is egyben)

valamint mely vezérlő támogassa nézetbe történő adatbetöltést (*LoadableDataController* interfész implementáció) a keretrendszer kezdeményezésére.

5.3 Jelentősebb interfészek és absztrakt osztályok

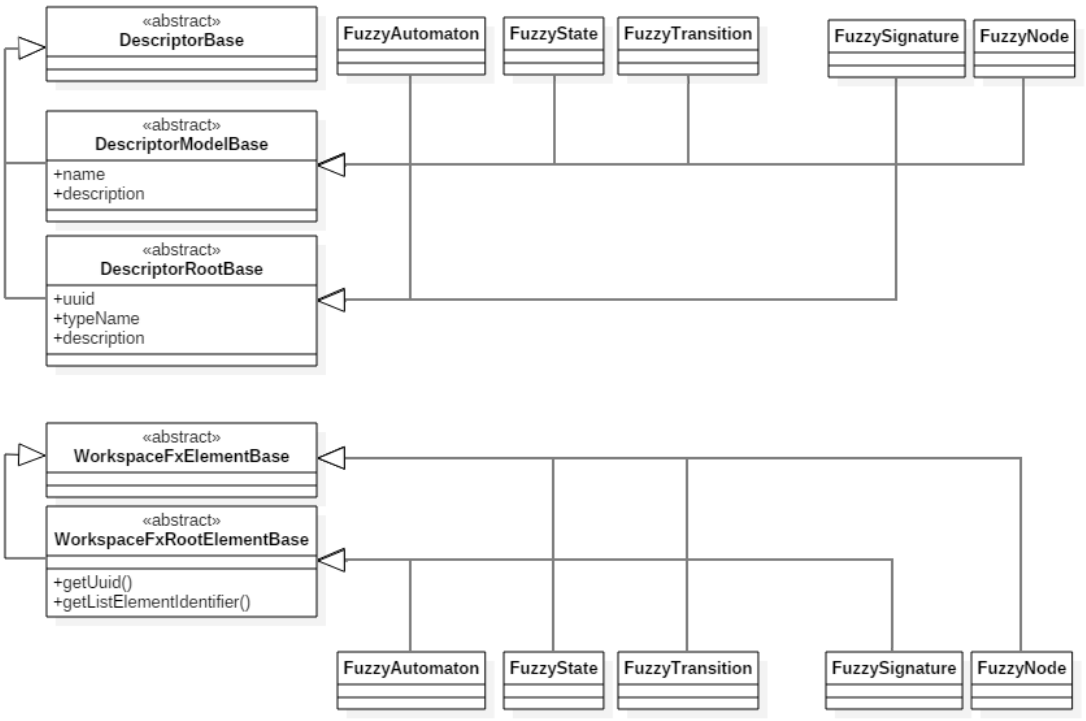
A keretrendszer és modulok együttesének összehangolása főként absztrakt osztályok, interfészek és generikusok felhasználásával történik. Bár Java nyelvű generikusok a type erasure alapú megvalósításának köszönhetően bytecode szinten már csak nyers objektumok formájában állnak rendelkezésre, a fordítás idejű ellenőrzések miatt nagy jelentőséget kapott az alkalmazás moduljainak helyességének implementációs időben történő ellenőrzésében. Az említett komponensek összehangolása az alábbiak szerint kerültek megvalósításra.

5.3.1 A modulok részére megvalósításra kínált adatszerkezetek

Az keretrendszer publikus API-ja a modulrendszere számára könnyen használható absztrakt típusokat tesz közé a fejlesztéshez. Ezek a típusok általában olyan absztrakt osztályok, melyek megjelennek a service-k interface definícióiban. A szolgáltatásokat megvalósító keretrendszer e típusokon keresztül biztosítja a modulok számára elérhető szolgáltatások helyes működését.

A 43. ábra bal oldalán látható az API-ban definiált meta szintű osztályok (*DescriptorBase, DescriptorModelBase, DescriptorRootBase, valamint WorkspaceFxElementBase, WorkspaceFxRootElementBase*) konkrét megvalósításait az egyes modulok tartalmazzák.

A modulok a jelenlegi megvalósításban tipikusan egy kiinduló (gyökér) elemre építik fel adatszerkezeteiket (ez alkotja pl. az XML/JSON fájlok gyökér csomópontját, vagy a modellszerkesztő listában megjelenítendő modelleket), melyekben meghivatkozzák a további (tipikusan hierarchikus) adatszerkezeteket (melyek *descriptor model* esetben leírják pl. az állapotgép átmeneteit, szignatúra felépítését, univerzum fuzzy halmazait, *fx model* esetben pedig felülethez kötik azokat).

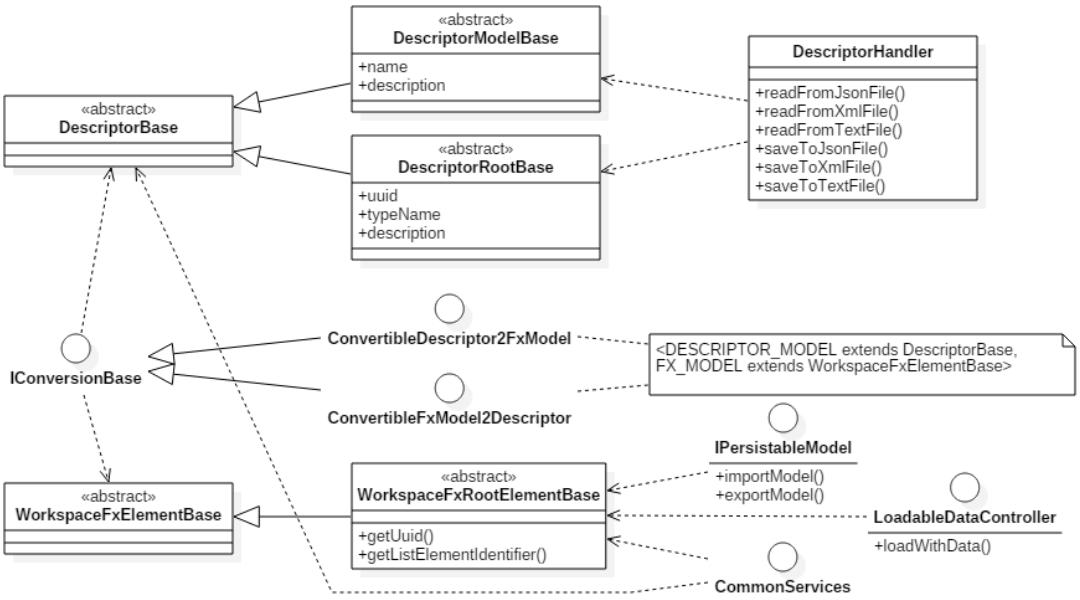


43. ábra - Metaosztályok és modulok domain osztályainak viszonya

A fenti ábrán látható *fx model* esetén a *WorkspaceFxElementBase* és *WorkspaceFxRootElementBase*, valamint *descriptor model* esetén a *DescriptorBase*, *DescriptorModelBase* és *DescriptorRootBase* megkülönböztetésére a modellek keretrendszerben történő felhasználásának szerepei miatt volt szükség. Mindkét esetben az öröklő absztrakt osztály egy speciálisabb helyzetben van felhasználva a keretrendszer által, ezért további tulajdonságokat/műveleteket is meg kell valósítania, melyekre egy átlag modellnél nem feltétlenül van szükség.

5.3.2 Szolgáltatások a descriptor model és fx model osztályaira építve

Az előző pontban leírt modellek absztrakcióira építi a keretrendszer az általa kínált szolgáltatásokat. Ennek egy egyszerűsített összefoglalója áttekinthető a 44. ábrán. Az ábrán látható ahogy a keretrendszerben kikényszerítésre kerül a *descriptor model* és az *fx model* típusok közötti konverzió megvalósítása, melyet a típusokból öröklő modell osztályok minden modulban szükségszerűen megvalósítanak.



44. ábra – A keretrendszer által nyújtott funkciók megvalósítása

Az ábrán látható, hogy az API-ban definiált osztályokra építenek a keretrendszer azon szolgáltatásai is, amelyek a modelleken keresztül a modulok vezérlői számára előírt működés meglétére építenek (például adatok betölthetők legyenek egy modul adott nézetébe, vagy a konkrét modellek validálhatósága).

A *DescriptorHandler* osztály a keretrendszer szintjén definiál bizonyos alapműködéseket, melyeket a modulok tovább specializálhatnak.

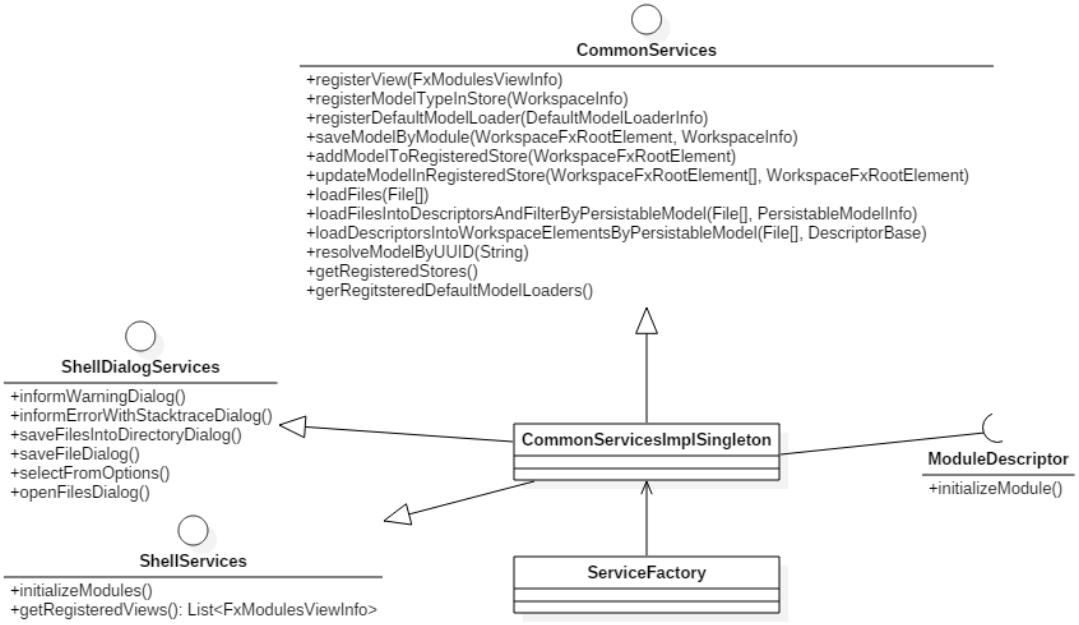
Az *IPersistabeModel*, valamint a *LoadebleDataController* az *api* névtérben kapott helyet, ezáltal a modulok számára kötelezően implementálandók. A *common* névtérben található *CommonServices* megvalósítása a keretrendszer feladata, a modulok ennek meglétére és helyes működésére aktívan építenek.

5.3.3 A keretrendszer által megvalósított interfészek

A keretrendszer által számos szolgáltatás került implementálásra. Ezek közül néhány fontosabb megtekinthető a 45. ábrán.

A keretrendszer több célból is szolgáltatásokat kínál. A modulok számára a *ShellService*-ek funkcionalitása nem, csupán a *ShellDialogServices*, valamint a *CommonServices* által összefogott funkciók érhetők el.

A *ShellDialogServices* a modulok számára nyújt az alkalmazás ablakkezelési megoldásával kompatibilis eszköztárt. Ablakkezelési funkciók használatát egy modul ezen az interfészen keresztül kezdeményezheti a keretrendszernél.



45. ábra - A keretrendszer által megvalósított interfészek

A *CommonServices* alatt találhatók azok a szolgáltatások, melyek segítségével a modulok elérhetik, hogy nézeteikről a keretrendszer tudomás szerezzen, típusaikat fel tudja használni a betöltésekhez, konverziókhoz, mentésekhez stb. Továbbá ezen szolgáltatásokon keresztül történik a létrehozott modellek mentése, frissítése. A modulok más modulok által regisztrált modellpéldányaihoz is az ezeken a metódusokon keresztül megvalósított lekérdezéseket felhasználva juthatnak. Egyik további nagyon fontos megvalósított művelet a típus sablon példányok UUID[28]-vel történő lekérdezése (ennek oka a következők alfejezetben kerül kifejtésre). A *CommonServices* biztosítja a leíró modell típusok (*descriptor model*), valamint a végrehajtható típusok (*fx model*) keretrendszer funkcionalitásaihoz történő illesztését is.

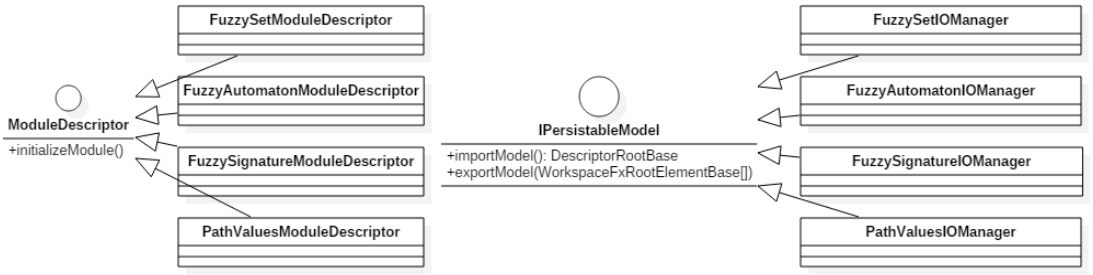
A *ShellServices* interfész csak a keretrendszer számára foglal össze néhány műveletet, a modulok számára különösebb jelentőséggel nem bír.

A keretrendszer által definiált szolgáltatások a *ServiceFactory*-n keresztül vehetők igénybe a modulokból.

[28] Universally Unique Identifier (UUID) vagy Globally Unique Identifier (GUID) – egyedileg generálható 128 bites erőforrás azonosító szám

5.3.4 Absztrakt osztályok megvalósítása a modulokban

A modulok a fent definiált típusok megvalósításával vállnak képessé a keretrendszerrel történő kommunikációra.



46. ábra - Néhány osztály megvalósítása modul szinten

Az implementált modulok a 46. ábrán látható sablon szerint végzik a számukra előírt interfészek megvalósítását. A fenti ábrán ezeknek az előírásoknak csak egy kisebb részhalmaza látható a szemléltetés végett. A fentieknél azonban jóval több előírást kell teljesítenie egy-egy modulnak, amennyiben megfelelően szeretne együttműködni a keretrendszerrel az alkalmazásban.

5.4 Típusok és modellek egyedi azonosítása

Mivel az alkalmazással szemben támasztott ésszerű követelmény, hogy konkrét adatokkal feltöltött modellek mellett újrafelhasználható típusok használatát is támogassa, (mely következménye, hogy a modulok csoportjának modelljein átívelő típusrendszereket kell tudni kezelni), valamint ezeknek még projekt szervezésű támogatását is érdemes megvalósítani (pl. egyik projektből másikba átmozgatható adat nélküli épület típus sablonok, vagy konkrét adattal kitöltött modell struktúrák), ezért ez az elvárás egy újabb dimenzionalitással egészítette ki a modellek kezelésének szintjét.

A növekvő komplexitás feloldása úgy történt, hogy közös szemantikai szinten kerültek kezelésre a konkrét modell adatok, valamint a feltöltendő üres épület sablon típusok. Így a modulok saját hatáskörben, őseik által definiált adattípusokra alapozva definiálhatják és használhatják sajátjaikat.

A *shell* keretrendszer szolgáltatásokat definiálva további lehetőséget biztosít a már létrehozott sablon-típusok/épület-típusok, valamint épület modellek elérésére és feloldására egy UUID azonosító segítségével. A moduloktól ugyanis minden esetben megkövetelt, hogy az általuk gyártott fuzzy típus sablonok és konkrét épület modelljeik

is egy egyedi azonosítóval kerüljenek kialakításra. A mögöttes szemantika szerint a típusok immutábilisak, azaz egyértelműen azonosítva vannak egy generált UUID azonosítóval, kialakításuk utáni további szerkesztésük már egy teljesen másik típust eredményez. Ennek szükségességét a hierarchikus modellek összeszerkesztése és gyakorlati használata indokolja, ugyanis nem kezelhető megoldás, ha egy már összeszerkesztett épület-modell valamelyik típusa a modell összeszerkesztését követően időközben megváltozik az alatta levő modell változása miatt. Továbbá nem elvárható az sem, hogy egy a domain terület karbantartója (nem pedig a szoftver felhasználója) által elkövetett változást a szoftver minden példányában a modellek azonnal lekövessenek.

5.5 Egyéb technikai részletek

Az alábbiakban néhány fontosabb technikai kialakítás kerül ismertetésre.

5.5.1 Gráfok megjelenítése

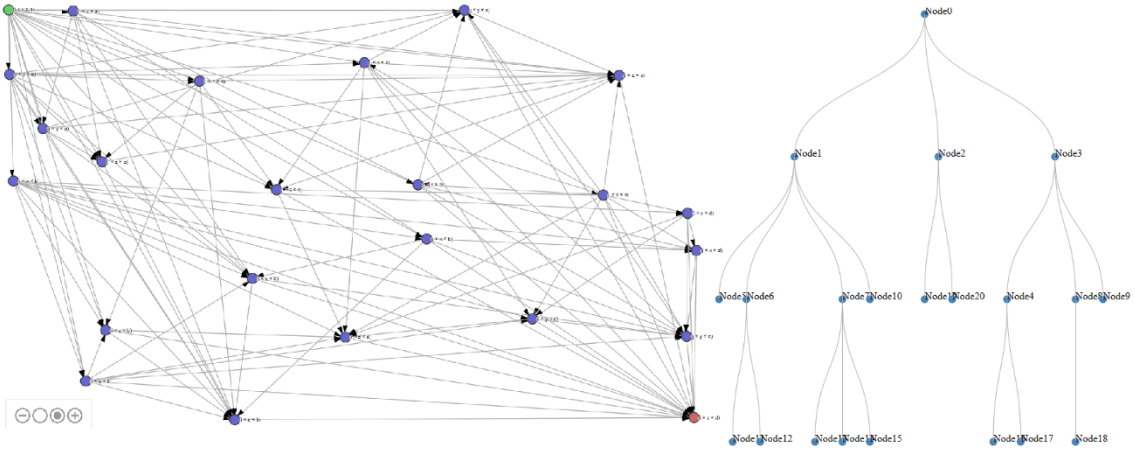
Mivel a felületen megjelenő állapotgépek megrajzolása körülményes lett volna pusztán a JavaFX grafikus vezérlő elemeire építve, ezért egyszerűbb és gyorsabb fejlesztés érdekében a rajzolásokhoz vagy egy külső osztálykönyvtár vagy valamilyen más jellegű, a JavaFX-n túlmutató megoldás használata tűnt ésszerűnek.

A gráfrajzoló osztálykönyvtárak közül a megfelelő kiválasztása kisebb nehézségekbe ütközött, mivel az ingyenes és licensz szerint fel is használható osztálykönyvtárak vagy nem széles körben támogatottak, vagy már nem állnak aktív fejlesztés alatt, vagy más grafikus keretrendszerekhez tervezték őket és a JavaFX-el való együttműködésük további nehézségekbe ütközik. Rendelkezésre álltak azonban jól használható fizetős lehetőségek, amelyek használatával nem kívántam élni.

A fentiek miatt egy web alapú megoldás használata került implementálásra, mely a Google **d3.js** keretrendszerére épülve adatvezérelt módon képes grafikus elemek vizualizálására.

A JavaFX keretrendszer oldaláról egy *WebView* típusú felületi elemen keresztül (mely a böngészők motorjaihoz hasonló) bármilyen webes tartalom megjeleníthető egy *WebEngine* által. A gráfokat megjelenítő felületi elemek ezért egy **HTML** oldal segítségével kerülnek megjelenítésre. A *WebEngine* betölti a HTML oldal tartalmát, annak minden szükséges függőségével együtt. Ezáltal a **CSS**, valamint a **Javascript** által meghatározott tartalmak is kiértékelésre kerülnek. A *WebView*-n keresztül Java kódból

string-gel parametrizált Javascript függvényhívások is kezdeményezhetők *executeScript()* metódus hívásával, ezáltal összeállítható egy olyan modellt leíró JSON karaktersorozat, amelyet a megfelelő Javascript függvénynek értékül adva a modell tartalmát a *d3.js* adatvezérelt grafikus keretrendszer segítségével végül egy animált SVG formátumú objektumot konstruál, amit a HTML oldal a CSS által meghatározott stílusok alapján képes megjeleníteni.



47. ábra - Példa gráfok WebView-n keresztül történő megjelenítésére az alkalmazásban

A 47. ábrán látható egy példa arra, ahogy a szignatúrák és a szorzat állapotgépek a fent leírt módszerrel megjelenítésre kerülnek.

A fenti megoldás egy általánosabb, gráfrajzoló osztálykönyvtártól független megoldásra épül, ezért viszonylag könnyen is karbantartható. Sajnos azonban hátránya is van a fent alkalmazott megoldásnak, ugyanis fejlesztési időben nehézkes a hibakeresés és a *d3.js* animációinak renderelésének gyorsasága is hagy némi kívánnivalót maga után *WebView*-n keresztül történő futtatás esetén, bár ez utóbbira már készültek független megoldások.

5.5.2 Algoritmusok adatszerkezetei

Az algoritmusok implementációikor gyakran volt szükséges kritérium az egyes objektumok könnyű kezelése, vagy olyan összetett adatszerkezetek használata, melyek alapból nem képezik részeit a Java nyelv környezetének.

Emiatt a *common* csomagban létrehozásra került néhány publikusan elérhető osztály, amelyek az algoritmusok implementációjakor néhol különösen hasznosnak

bizonyultak. Ilyen segéd adatszerkezet például az *ennes*[29]-eknek megfelelő generikus osztályból képzett objektum, amely sok algoritmusnál könnyítette meg az összetartozó értékek egy objektumként történő visszaadását metódusok visszatérési értékeként (48. ábra).

```
public class Tuple2<T1, T2> {

    public T1 _1;
    public T2 _2;

    public Tuple2(T1 t1, T2 t2){
        _1 = t1;
        _2 = t2;
    }

}
```

```
public class Tuple3<T1,T2,T3> {

    public T1 _1;
    public T2 _2;
    public T3 _3;

    public Tuple3(T1 t1, T2 t2, T3 t3){
        _1 = t1;
        _2 = t2;
        _3 = t3;
    }
}
```

48. ábra - Egy 2 és egy 3 hosszú ennes megvalósítása

További segéd adatszerkezetekre volt szükség például a Dijkstra algoritmus szomszédossági mátrixának implementációjakor is, ahol egy dinamikusan növekvő méretű, soraiban és oszlopaiban különböző típusú elemekkel címezhető tábla generikus megvalósítására volt szükség (ahol például az oszlopok és sorok a szorzat állapotgép összetett állapotaival címzettek, a cellák tartalma azonban egy olyan egész szám, mely akár a végtelent reprezentáló *null* értéket is felvehet). E kritérium kielégítéséhez felhasználásra került a **Google Guava** osztálykönyvtára, amely hasonló adatszerkezeteket implementál. A Dijkstra algoritmus esetében ez egy *Table* típusú adatszerkezet, mely sorain és oszlopain *CompoundState* csúcsokkal címzett, a cella értéke pedig egy *Optional<Integer>* típusú érték.

```
Table<CompoundFuzzyState, CompoundFuzzyState, Optional<Integer>>
adjacencyMatrix_C = HashBasedTable.create();
```

5.5.3 Unit tesztek

A bonyolultabb algoritmusok implementációi során (mint például véges állapotú állapotgépek szorzatainak előállítása, vagy az evolúciós algoritmus egyes operátorainak

[29] Ennes (n-tuple) – egy n hosszú rendezett lista

transzformációi) a TDD[30] elveinek megfelelően lettek megvalósítva. Elsőként a lefuttatandó tesztfeladatok és azok elvárt eredményei kerültek leírásra (unit teszt[31]eken belüli *Assert*-ek formájában), majd csak ezt követően kerültek a tényleges műveletet elvégző algoritmusok implementálásra.

A tesztek lefuttatásával így a teszteken belül létrehozott példákkal is ellenőrizhető az algoritmus futásának helyessége.

A unit tesztek a jUnit4-es keretrendszerben kerültek megvalósításra.

5.6 Projekt hierarchia

Bár nem minden modul rendelkezett egységesen az MVC összes komponensével (hiszen a keretrendszer alkalmazásban a beépülő modulok képezik a *shell* „modelljét"), azonban viszonylag egységes projekt és névtérszervezési konvenció került kialakításra az alábbiak szerint:

- *src / api* – az alkalmazás által a beépülők számára előírt feltételei, interfészek
- *src / common* – az alkalmazás által publikált, bárki által szabadon használható adatszerkezetek és már megvalósított interfészek
- *src / shell / services* – szolgáltatások implementációi (keretalkalmazásban van jelen)
- *src / modul / alap névtér* - az alkalmazások inicializációs kódjai és „main-controller"-ei
- *src / modul / model* – az MVC architektúra modelljei (csak a modulokban jelennek meg)
- *src / modul / model / descriptor* – exportot/importot segítő kódok
- *src / modul / model / fx* – adatkötést és domain logikát megvalósító modellek
- *src / modul / view* – az MVC architektúra nézetei és „view-controllerei"

[30] Test Driven Developement (TDD) – olyan fejlesztési módszertan, amely szerint elsőként a tesztek (bemenetek és elvárt kimenetek) kerülnek megírásra, majd ezután maga a logika/algoritmus, amelynek az elvárt kimeneteket elő kell tudnia helyesen állítani.

[31] Unit teszt – a rendszer legkisebb egységeinek tesztelése.

- *src / modul / util* – egyéb segéd megoldások
- *src / modul / view / drawing* – kirajzolásokat segítő kódok (modulokban jelennek meg)
- *sample* – példaadatok automatikus generálását végző kódok
- *test* – a bonyolultabb számítások eredményeinek ellenőrzéséhez előre megírt tesztjei

6 Konklúziók

A fenti fejezetekben kidolgozásra került egy választott szakterületnek a fuzzy szignatúra és evolúciós optimalizálás alapú kombinált modellre történő illesztése. Az Irodalomkutatás c. fejezetben tárgyaltak önmagukban csak a legalapvetőbb ötleteket és információkat szolgáltatták egy ilyen rendszer megalkotásához, ezek technikai részletei az átolvasott cikkekben nem kerültek még kidolgozásra.

A korábbi fejezetek bemutatták, hogy milyen alapvető tervezési és implementációs döntésekkel érdemes kiegészíteni a cikkekben tárgyaltakat ahhoz, hogy a fenti modell, mint egy önálló szoftvertermék működőképes legyen.

Az alábbiakban kerülnek bemutatásra a téma feldolgozása során szerzett eredmények, az alkalmazás fejlesztésével szerzett tapasztalatok, valamint megfogalmazásra kerül néhány jövőbeli fejlesztésre irányuló javaslat is.

6.1 A munka elkészültségi szintje

Az előzőekben bemutatott alkalmazás a megvalósítása jelen időpontig többszöri iteráción esett át. Az iterációk folyamán igyekeztem minden lehetséges szempontot figyelembe venni az alkalmazás folyamatos finomítása során, azonban ez nem történhetett meg maradéktalanul.

Bár véleményem szerint összességében egy jól megtervezett alkalmazás alapjai kerültek implementálásra, azonban ennek sajnálatos következménye, hogy az alkalmazás implementációjával szemben támasztott elvárások jelentősen megnövelték a fejlesztésre fordítandó idő mennyiségét.

Eddig a pontig az alkalmazásnak valamennyi fontosabb részegysége implementálásra került, így sok funkció már bemutatható. Azonban számos olyan részegység készült el, amelyek egyelőre még csak „proof of concept” szintjén léteznek. Ezek ugyan működőképes és számítható modellek, de az alkalmazásba történő integrációjuk egyelőre még nem teljes. Erre példa, hogy a felületen adott a szignatúrák összeállításának és kiértékelésének lehetősége, azonban a konkrét kiértékelés egyelőre csak futtatható tesztek formájában került megvalósításra, grafikus integrációja még kidolgozásra vár (bár alsóbb szintű modulok implementációi alapján ez már nem jelenthet nagyobb problémát). Emiatt az evolúciós optimalizációt, vagy az épületek

összehasonlítását végző részletek is egyelőre csak tesztek formájában elérhetők. A feladat „proof of concept” jellegéből fakadóan e funkciók teljeskörű, maximálisan részletes implementációja a feladat terjedelmére nézve nem volt a kezdeti elvárások része. A tárgyalt algoritmusok implementációja, valamint azok működőképességének igazolása azonban mindenhol megtörtént, ezáltal a koncepció helyességének és működőképességének igazolása is.

A főbb funkciók a következő mértékben kerültek megvalósításra:

- **Fuzzy halmazok** – Modell és a szerkesztő felület grafikus implementáció teljes.
- **Fuzzy állapotgépek** – Modell és a szerkesztő felület grafikus implementáció teljes.
- **Fuzzy szignatúrák** – Modell és a szerkesztő felület grafikus implementációja teljes.
- **Állapotgépek szorzata** – Modell és a vizualizáció teljes.
- **Szakterületi modellek** – Implementáció részleges, csak a számítások kiértékeléséhez szükséges mértékben megvalósított. Egyéb funkciók, mint például multimédia kezelése az egyes épületekre egyelőre még erősen hiányos.
- **Bakteriális optimalizáció –** Keretrendszer és a modell implementációja teljes, a felülethez történő integráció jelenleg nem megoldott. A probléma definiálása egyelőre tesztkörnyezetben oldható meg, akár unit tesztek formájában.
- **Többcélú optimalizáció** – Kiterjesztési pontok keresztül kódolható fitneszfüggvényeket megvalósító metódusreferenciák átadásával és költségvektor típusok meghatározásával a tesztkörnyezetben megvalósított.
- **Keretrendszer + Modulrendszer** – Teljes.

Bár az elkészült alkalmazás üzleti szempontból nem tekinthető teljesnek, azonban ez nem képezte a követelmények részét. A két fő cél csupán egy olyan szemléletes példaalkalmazás elkészítése volt, amelyik képes igazolni a fenti modellek szakterületen történő alkalmazhatóságát, valamint amely lefekteti az üzleti szempontoknak megfelelő kritériumokat. Ezek a kritérium azonban a jelenlegi félkész állapot ellenére is teljesülnek. Az alkalmazás kész termékké történő fejlesztése azonban az eddigi erőfeszítéseket is jócskán meghaladó mértékű energiabefektetést és időt igényelhet.

Érdemes megemlíteni, hogy a fejlesztés iterációi során nagy volt az a kódmennyiség, amely időközben elévült, újra írásra került vagy generikus formába történő átírása során szerepe jelentősen csökkent.

Véleményem szerint az alkalmazás technikai szempontból megfelelő lehetőségeket kínál a további modulok fejlesztéséhez, melyek a kialakítás miatt csak egy rendszer szintű áttekintést igényelnek (esetleg az eddig elkészült szakterületi kódok felületesebb ismeretét). Ezek megértésével komolyabb háttértudás nélkül egy független fejlesztő is képes a meglevő implementáció részét vagy egészét felhasználni munkájában.

6.2 Továbbfejlesztési lehetőségek

A kitűzött feladat céljai között nem szerepelt számos olyan lehetőség, amely megfontolásra érdemes a jövőbeli továbbfejlesztések szempontjából. Ilyen lehetőségek például:

- A szakterületi modell kiegészítése nevezéktannal és címkékkel (taxonómiák vagy szemantikus elemek hozzáadásával), mellyel a számítási modell egyes elemeihez a szakterület mérnökei információkat rendelhetnek (pl. a szignatúrában hol található a tetőszerkezetet leíró változók csoportja).
- Két eltérő szignatúrájú modell leírás esetén, amelyek az információkat eltérő részletezettséggel és struktúrában reprezentálják lehetőség lehet szignatúra transzformációk megadására, és ezáltal résziben is összehasonlítani két szignatúrát, nem csak a gyökércsúcsba történő aggregációval.
- Állapotgépek és épületek abszolút és relatív költségeinek adatbázisokból történő meghatározása a pontosabb becslések elérése érdekében. Korábbról származó felújítási munkálatok információinak felhasználása az felújítások modellezéséhez.
- Folyamatok modellezésének állapotgépek közötti finomítása. Például, ha két állapotgép között definiálható, hogy egy-egy tranzíciójuk szükségszerűen kövessék egymást, párhuzamosak legyenek, átfedjenek, részben függetlenek legyenek, vagy egyik használata a másik állapotgépben egy él tiltását eredményezi stb. Ezekre állapotgépek között szabályok fogalmazhatók meg.
- Szignatúrán belül az állapotgép szorzás műveletének végrehajtása akkor, ha a levelekben a felújítások folyamatai nem ismertek/leírhatók. Ekkor egy újfajta

bizonytalanság jön be a becslés során, amely a költségmátrixot min-max értékekkel korlátolt tartományba

- Két szignatúra legnagyobb közös részfájának meghatározása és a kiértékelés arra történő korlátozása.
- stb.

7 Összefoglalás

A dolgozatban részletesen bemutatásra került egy olyan számítási modell, amely képes hasonló struktúrájú, főként hierarchikusan szervezhető de akár eltérő részletezettségű, vagy helyenként hiányos információkkal leírható problémák feldolgozására és kiértékelésére. Erre épülve további modellek kerültek bevezetésre, melyek nem az információk struktúrájának megragadására, hanem az egyes folyamatok modellezésére és azok optimalizációjára hivatottak.

Érintőlegesen bemutatásra került az épületfelújítások problémája is, melyhez tartozó szakterületi problémák modellezése során az ilyen típusú számítási modellek kiértékelésének lehet valós gyakorlati alkalmazása. Feltárásra került, hogy e területen még nem létezik ilyen számítási modellt alkalmazó eszköz, ezért felmerült egy ilyen szoftver elkészítésének az igénye, melyhez egy célzott „proof of concept” jellegű alkalmazás megvalósítása lett tervbe véve.

Mivel korábbról származó implementációk vagy egyéb gyakorlati eredmények nem álltak rendelkezésre, ezért a fentiekre építve kidolgozásra kerültek azok a gyakorlati megfontolásokból származó igények, melyek egy üzleti jellegű alkalmazással szemben támaszthatók. Továbbá a fenti számítási modellek kiegészültek a gyakorlati alkalmazásból származó elvárásokkal.

A munka során megtervezésre és elkészítésre került egy olyan implementáció, amely a fenti elvárásoknak megfelelő működést demonstrál, magában foglalja a fenti számítási modelleket, modulokkal dinamikusan bővíthető grafikus keretrendszert tesz elérhetővé és kiterjesztési pontokat kínál további funkciók implementálására. Ez az implementáció főként mintákat és útmutatásokat tartalmaz egy üzleti szempontból vett termék megvalósítására nézve, valamint igazolja a fenti számítási modellek működőképességét. Ez a „proof of concept” jellegű implementáció bár példákon és teszteken keresztül mutatja be a fenti modellek alkalmazhatóságát, azonban számos helyen még nem teljes, ezért egy a piaci igényeknek is megfelelő szoftver további fejlesztéseket igényel.

A hiányzó, még nem implementált (főként a felületi integrációval kapcsolatos) funkciók megvalósításán túl további lehetséges fejlesztési irányok is felmerülnek, melyek

főként a modellek azonos alakra történő transzformálásával, valamint szakterületi modellek szemantikai alapú kiterjesztéseivel kapcsolatosak.

E diplomaterv folytatását ajánlom minden olyan informatikai háttérrel rendelkező érdeklődőnek, aki kedvet érez a fenti szakterület fuzzy alapú modelleken történő megismerésére, vagy a létrehozott platform alkalmazás és moduljainak továbbfejlesztésére.

Irodalomjegyzék

[1] László T. Kóczy, Gergely I. Molnárka, *Optimizing Complex Building Renovation Process with Fuzzy Signature State Machines*, C.K. Loo et al. (Eds.): ICONIP 2014, Part II, LNCS 8835, pp. 573-380, 2014., Springer International Publishing Switzerland 2014

[2] László T. Kóczy , Gergely I. Molnárka, *Fuzzy State Machine-Based Refurbishment Protocol for Urban Residential Buildings*, A. Laurent et al. (Eds.): IPMU, Part II, CCIS 443, pp. 375-384, 2014., Springer International Publishing Switzerland 2014

[3] László T. Kóczy, Gergely I. Molnárka, *Building Renovation Cost Optimalization with the Support of Fuzzy Signature State Machines*, Springer International Publishing Switzerland 2015, S. Phon-Amnuaisuk and T.-W. Au (eds.), Computational Intelligence in Information Systems, Advances in Intelligent Systems and Computing 331, DOI: 10.1007/978-3-319-13153-5_13

[4] László T. Kóczy, Gergely I. Molnárka, *Fuzzy Signature Structure-Based Finite-State Machines in a Residential Building Ranovation Procedure*, Paper 95, Civil-Comp Press 2014, Proceedings of the Ninth International Conference on Engineering Computational Technology, P. Iványi and B.H.V Topping, (Editors), Civil-Comp Press, Stirlingshire, Scotland

[5] László T. Kóczy, Gergely I. Molnárka, *Decision Support System for Evaluating Existing Apartment Buildings Based on Fuzzy Signatures*, Int. J, of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1842-9844, Vol. VI (2011), No. 3 (September), pp. 442-457

[6] László T. Kóczy, Ádám Bukovics, *Evaluating Condition of Buildings by Applying Fuzzy Signatures and R-Fuzzy Operations*, L. T. Kóczy et al (eds.), Issues and Challenges of Intelligent Systems, Studies in Computational Intelligence 530, DOI: 10.1007/978-3-319- 03206-1_4, Springer International Publishing Switzerland 2014

[7] László T. Kóczy, Claudiu Pozna, Nicusor Minculete, Radu-Emil Precup, Áron Ballagi, *Signatures: Definitions, operators and applications to fuzzy modelling*, 0165-0114 ©2012 Elsevier B. V., doi:10.1016/j.fss.2011.12.016

[8] László T. Kóczy, István Á. Harmati, Ádám Bukovics, *Sensitivity Analysis of the Weighted Generalized Mean Aggregation Operator and its Application to Fuzzy Signatures*, 2014 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), July 6- 11, 2014, Beijing, China, 978-1-4799-2072-3/14

[9] László T. Kóczy, László Gál, Rita Lovassy, *Progressive Bacterial Algorithm*, CINTI 2012, 13th IEEE International Symposium on Computational Intelligence and Informatics, 20-22 November, 2012, Budapest, Hungary, 978-1-4673-5206-2/12, 2014 IEEE

[10] László T. Kóczy, Boldizsár Tüű-Szabó, Péter Földesi, *Improved Discrete Bacterial Memetic Evolutionary Algorithm for the Traveling Salesman Problem*, adfa, p. 1, 2011., Springer-Verlag Berlin Hedelberg 2011

[11] László T. Kóczy, Boldizsár Tüű-Szabó, Péter Földesi, *An effective Discrette Bacterial Memetic Evolutionary Algorithm for the Traveling Salesman Problem, Intern. Journal of Intelligent Systems, Wiley, 2017 p.15 (online available)*

[12] Samuel C. Hsieh, *Product Construction of Finite-State Machines*, Proceedings of the World Congress on Engineering and Computer Science 2010 Vol I, WCECS 2010, October 20-22, 2010, San Francisco, USA

[13] Sachith Abeysundara, Baladasan Giritharan, Saluka Kodithuwakku, *A Genetic Algorithm Approach to Solve the Shortest Path Problem for Road Maps*, Proceedings of the International Conference on Information and Automation, December 15-18, 2005, Colombo, Sri Lanka

[14] Dr. Rakesh Kumar, Mahesh Kumar, *Exploring Genetic Algorithm for Shortest Path Optimization in Data Networks*, Global Journal of Computer Science and Technology, Vol. 10 Issue 11 (Ver. 1.0) October 2010

[15] Molnárka Gergely, *Lakóépületek állapotfelismerése és felújítás tervezése strukturált fuzzy deszkriptorok segítségével*, Doktori értekezés, Széchényi István Egyetem, 2015

[16] Tamás Katalin, *Következtetés fuzzy szignatúra alapú modellekben*, Diplomaterv, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2008

[17] Borgulya István, *Optimalizálás evolúciós számításokkal*, p. 387, Typotex kiadó, 2012

[18] Álmos Attila, Győri Sándor, Horváth Gábor, Várkonyiné Kóczy Annamária, *Genetikus algoritmusok*, p. 254, Typotex kiadó 2010

[19] Christian Nilsson, *Heuristics for the Traveling Salesman Problem*, Tech. Rep., Linkoping University, Link ¨ oping, Sweden, 2003.

[20] Andrius Blazinskas, Alfonsas Misevicius, *Combining 2-opt, 3-opt and 4-opt with K-Swap-Kick Perturbations for the Traveling Salesman Problem*, Conference: 17th International Conference on Information and Software Technologies, At Kaunas, 2011

[21] JavaFX dokumentáció (megtekintve: 2017-05-12) http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

[22] OSGi dokumentáció (megtekintve: 2017-05-12) https://www.osgi.org/

[23] Apache Felix OSGi implementáció (megtekintve: 2017-05-12) http://felix.apache.org

[24] Equinox OSGi implementáció (megtekintve: 2017-05-12)
http://www.eclipse.org/equinox/

[25] BndTools dokumentáció (megtekintve: 2017-05-12)
http://bndtools.org/

[26] Gradle dokumentáció (megtekintve: 2017-05-12)
http://gradle.org/

[27] Git verziókövető dokumentáció (megtekintve: 2017-05-12)
https://git-scm.com/

[28] e(fx)clipse dokumentáció (megtekintve: 2017-05-12)
http://www.eclipse.org/efxclipse/index.html

[29] DromblerFx dokumentáció (megtekintve: 2017-05-12)
http://www.drombler.org/drombler-fx/

[30] JUnit dokumentáció (megtekintve: 2017-05-12)
http://junit.org/junit4/