

Mesterségesintelligencia-kutatások a Miskolci Egyetemen

Elemző és robotizált folyamatautomatizálási rendszer fejlesztése
nagy terhelésű ügyfélszolgálatok számára

Kutatási jelentések
2024/1

Miskolci Egyetemi Kiadó
2024

Mesterségesintelligencia-kutatások
a Miskolci Egyetemen

Szerkesztette:
Prof. dr. Kovács László

Mesterségesintelligencia-kutatások a Miskolci Egyetemen

Elemző és robotizált folyamatautomatizálási rendszer fejlesztése
nagy terhelésű ügyfélszolgálatok számára

Kutatási jelentések
2024/1

Szerkesztette: Prof. dr. Kovács László



Miskolci Egyetemi Kiadó
2024

ISBN 978-615-6835-00-0

ISSN 2939-6883

TARTALOMJEGYZÉK

<i>Kovács László:</i> Előszó	7
<i>Kovács László:</i> Az ERPA-projekt kapcsán kidolgozott kísérletek, modellek bemutatása, elméleti háttér ismertetése	9
<i>Mülejtf Péter:</i> Webes gráfszerkesztő és mintageneráló alkalmazás tervezése és fejlesztése	47
<i>Baksa Attila:</i> A folyamatfeltáró algoritmusok áttekintése a PM4Py használatával	78
<i>Baksáné Varga Erika:</i> Folyamatfeltáró algoritmusok komplex vizsgálata szintetikus adat- halmazok felhasználásával	92
<i>Radeleczki Sándor:</i> Automatikus tesztelési környezet kidolgozása a létrehozott modellek összehasonlítása érdekében	116
<i>Gégény Dávid:</i> Az MPM-algoritmus továbbfejlesztése és validálása	136
<i>Csepányi-Fürjes László:</i> Ügyfélszolgálati megkeresések automatikus feldolgozásának lehetőségei emberközpontú mesterségesintelligencia-módszerekkel	146

ELŐSZÓ

A robotizált folyamatbányászat (RPM, Robotic Process Mining) és a robotizált folyamatautomatizálás (RPA, Robotic Process Automation) napjaink kiemelt kutatási területei a vállalatok ügyviteli folyamatainak intelligens informatikai megoldásokkal történő hatékonyság javítására. Jelen tanulmány a Miskolci Egyetem Informatikai Intézetének az *ERPA – Elemző és robotizált folyamatautomatizálási rendszer kifejlesztése, nagyterhelésű ügyfélszolgálatok számára* című projekt (2020-1.1.2-PIACI-KFI-2020-00165) keretében elvégzett kutatásait és az elért eredményeit mutatja be az eltelt egy évre vonatkozólag.

A kutatócsoport fő feladata az ügyviteli vagy termelési folyamatok automatizált feltárása gépi tanulási eszközökkel a meglévő eseménynaplók alapján. Az ME kutatási munkájának célja a vállalatnál végbemenő ügyviteli folyamatokban, adminisztrátori munkalépésekben azonosítani a műveletfolyamokat és vezérlési szerkezeteket (szekvenciákat, ciklusokat, elágazásokat, feltételeket) és ezek alapján elkészíteni a folyamat általános leírására szolgáló sémagráfot. A sémagráf felhasználható többek között a folyamat automatizálására és a folyamatok minőségellenőrzésére.

A projekt eredményeként olyan új algoritmusok kerültek kifejlesztésre, amelyek lehetővé teszik a komplexebb üzleti folyamatok automatizált feldolgozását is, ezáltal megnyílik az út az RPM-szoftverek új generációjának kifejlesztése előtt. A projekt eredményei között megemlíthető, hogy kidolgozásra került az ipari alkalmazások területén egy, a korábbiaknál hatékonyabb sémafeltárást végző algoritmus. A létrehozott algoritmus alkalmas komplex sémák feltárására is és alkalmazása növeli az automatizálás hatékonyságát. A kifejlesztett módszerek és eljárások lehetőséget adnak a magasabb fokú irodai folyamat automatizálásra, ezáltal költségkímélőbbé válnak az adminisztrációs eljárások; gyorsabb ügyintézés valósítható meg egy szélesebb alkalmazói körben.

A tanulmány több témakör részletes bemutatásából áll, a fejezetek önmagukban is egységet képeznek, egy-egy feladatkör módszertanát és az elért eredményeket mutatják be. Emellett a dolgozat jól hasznosítható a tématerület fontosabb fogalmainak a megismerésére is. Igény esetén a megadott elérhetőségeken keresztül további részletes információkkal is tudnak szolgálni a szerzők.

Miskolc, 2024.május 31

Prof. dr. Kovács László
szakmai projektvezető

AZ ERPA-PROJEKT KAPCSÁN KIDOLGOZOTT KÍSÉRLETEK, MODELLEK BEMUTATÁSA, ELMÉLETI HÁTTÉR ISMERTETÉSE

KOVÁCS LÁSZLÓ

Az ERPA-projekt keretében a munkacsoportunk feladata mesterséges intelligencia alapú folyamatfeltáró algoritmus(ok) tervezése és megvalósítása, ami képes nagy méretű eseménynaplók alapján az üzleti folyamatok hatékony feltárására.

1. Elvégzett kísérletek/modellezések bemutatása

1.1. Feladat célkitűzése, motiváció, elvégzendő szakmai feladat bemutatása

A kitűzött cél megvalósításhoz a projektet több részfeladatra bontottuk le.

Az időszak fő feladata a gépi tanuláson alapuló folyamat sémagráf-előállító algoritmus kidolgozása és tesztelése. A feladat aktualitását az adja, hogy

- a nagyobb méretű, komplexebb folyamatok esetén a hagyományos automata vagy illesztésalapú módszerek nem adnak kellő pontosságot;
- több funkcionális elem hiányzik a használatos módszereknél, például a ciklusok kezelése és az események ismétlődése;
- a gépi tanulási módszerek nagyobb rugalmasságot adnak a naplók feldolgozására, a modell a minták alapján határozódik meg, nem kell a programba kódolni az egyedi szabályokat.

A feladat nehézségét az adja, hogy

- nincs még hatékony gépi tanuláson alapuló módszer a sémagráfok előállításra, hiszen az NN számára is problémás a ciklusok feltárása a szekvenciákban, illetve a sémák integrálása;
- nincs egyértelmű hálóstruktúra ezen feladattípusra, eltérő induló elképzelésekkel lehet találkozni az irodalomban.

A helyzetértékelés alapján a következő feladatokat tűztük ki magunk elé:

- a minta illesztésalapú sémageneráló algoritmus tesztelése és hangolása, a módszer véglegesítése;
- az NN-alapú sémagenerálási módszerek begyűjtése, az irodalom elemzése és értékelése;
- a javasolt NN-technológiákból a leginkább reális megoldást jelentő változatok kiválasztása;

- a teszteléshez használható adathalmazok készítése; összemérhető, és az előállított folyamatmodellek értékelhetők;
- a kiválasztott technológiákhoz mintarendszer implementálása Python környezetben, a mintarendszerek tesztelése;
- benchmark tesztek elvégzése a meglévő, használatos módszerekkel történő összevetésre;
- a tesztek eredményeinek elemzése;
- a teszteredmények alapján a részletesebb vizsgálatba bevonandó változat meghatározása;
- a teszteredmények alapján annak eldöntése, hogy célszerű-e új architektúrák bevonása az elemzésbe;
- az igénynek megfelelően új architektúrák kidolgozása, a kitalált módszer elvi értékelése;
- az új módszer részletes elméleti elemzése a hatékonyság, alkalmazási korlátok, időbeli gyorsaság szempontjaiból;
- az új módszer implementációs tervének a kidolgozása;
- az új módszer implementálása;
- tesztek elvégzése a kidolgozott módszerrel;
- a módszer teljesítményének értékelése az elvégzett teszttapasztalatok alapján a pontosság és gyorsaság vonatkozásában.

1.2. Feladat módszertani háttere, irodalom rövid értékelése

A sémabányászat fő feladata az, hogy meghatározza azokat a közös mintaszegmenseket, amelyek képesek lefedni az összes trace-példányt a bemeneti naplóban. Erre a folyamatra mint egy speciális általánosítási feladatra lehet tekinteni, az eredményül kapott séma a bemeneti eseménysorozatok általánosítása. A folyamatmodell megjeleníti az események közötti vezérlési folyamatokat és azok kapcsolatait a bemeneti eseménynapló alapján. A felfedezett folyamatmodell minőségét az eseménynaplóban rögzített viselkedés pontos tükrözésének pontosságával mérik. A gyakorlatban a felfedezési módszereknek egyensúlyt kell találniuk négy minőségi kritérium között. Az felfedezett modellnek i) szimulálnia kell az eseménynaplóban található összes viselkedéstípust (alkalmasság), de ii) nem szabad más viselkedést szimulálnia (pontosság). A modellnek iii) általánosnak kell lennie, és ugyanakkor iv) a lehető legegyszerűbbnek kell lennie. Egyik legelső szabályfeltáró algoritmus az alfa-algoritmus, amely lefektette az alapokat a későbbi továbbfejlesztett algoritmusok számára is. Az alfa-algoritmus egy Petrihálót állít elő az eseménynaplóban rögzített folyamatpéldányok ábrázolására.

Az algoritmus feltételezi, hogy az eseménynaplók teljesek az összes megengedett bináris sorozat szempontjából. Érzékeny a zajra, és nem képes a hurokok, ciklusok vagy az egyedi komplex elágazási szerkezetek felfedezésére. Az elmúlt húsz évben széles körben vizsgálták az eseménynaplókból történő folyamatmodellek felfedezésének problémáját. Annak ellenére, hogy számos javaslat létezik, az automatizált folyamatfelfedezési módszerek olyan modelleket állítanak elő, amelyek (i) nagyok és spagetti-szerűek; és (ii) vagy rosszul illeszkednek az eseménynaplóhoz, vagy túláltalánosítják azt, ha valós életből származó eseménynaplókra alkalmazzák.

Az algoritmusok értékelésére több tanulmány is született. Weerdt és mtsai. a 2012 előtt közzétett technikák minőségi értékelését végezték el. Sahu és Nayak áttekintették és összehasonlították az esetazonosítót tartalmazó címkézett eseménynaplók, valamint a címkézetlen eseménynaplók feldolgozási módszereit. Augusto és mtsai. több besorolási dimenziót azonosítottak, mint például a kapott folyamatmodell típusa. Ebben a tekintetben vannak olyan módszerek, amelyek szabványos modelleket állítanak elő, mint például a Petri-hálók [HK, Aim, alfa, RegPFA, TAUMiner, ProM-D, CoMiner, Decomposed Process Miner, HybridILPMiner], a BPMN-modellek [Competition Miner, DGEM, BPMN Miner Heurisztika Miner, Split Miner, Fodina] és okhálózatok, vagy nem szabványos reprezentációk, mint például állapotgépek, folyamatfák és grafikonok.

A folyamatbányászati eszközök a felfedezett modellt közvetlenül követő grafikon (DFG) formájában biztosítják. Mokhov és mtsai. két módszert javasoltak a kompakt feltételes részleges rendelkezési grafikon (CPOG) reprezentációk bányászatára. A CPOG irányított grafikon, amelynek csúcsai és ívei Boole-függvényekkel vannak címkézve. Ezt a jelölésrendszert eredetileg a konkurens rendszerek több viselkedési forgatókönyvét tartalmazó kompakt specifikációk elemzésére vezették be. A javasolt algoritmusok a vezérlési folyamatot és az eseménynaplóból származó releváns paramétereket is kinyerik.

Vasilecas és mtsai. egy módszert mutatnak be az irányított ciklikus grafikonok (DAG) kinyerésére eseménynaplókból. Ezekből a grafikonokból bayesi hálózatokat generálnak, amelyek a valószínűségi grafikus modellekkel dolgoznak. A grafikon csomópontjai véletlen változókat képviselnek, és a csomópontokat összekötő élek a véletlen változók közötti feltételes valószínűségeket képviselik. A folyamat egy vázkinyerési eljárást használ, amely szerkezetet hoz kevésbé strukturált üzleti folyamatokhoz. Egy futásidejű monitorozó keretrendszert is kifejlesztettek naplóelemzésre, amely egyidejűleg kinyeri a folyamatpéldányokat és megfelelő metrikákat származtat.

A Celonis Process Mining jelenleg a világ legmagasabb értékelésű és legnépszerűbb folyamatbányászati eszköze, amelyet a Fortune 500 vállalatok minden

iparágban használnak. A Celonis Process Mining vannak több olyan funkciója, amelyek közvetlenül kapcsolódnak a kutatási témakörünkhöz. Idetartozik a folyamatfeltárás heurisztikus bányászata (2002) és a fuzzy bányászat (2006) által inspirált, tokenalapú megfelelőségi ellenőrzés (2005), tokenanimáció és csúszkák (2006), folyamat alapú ok-okozati elemzés (2006), és az induktív folyamatfelfedezés (2013). A folyamatfelfedezés egy tranzíciós rendszer felépítésével történik, amely közvetlenül követő kapcsolatokat használ.

Az Apromore (Advanced Process Model Repository) egy mesterséges intelligencia által vezérelt folyamatbányászati és -fejlesztési platform. Ötvözi a folyamatbányászatot a feladatbányászattal, amely a gyakran manuális feladatokra összpontosít, amelyek majdnem minden folyamat során előfordulnak. Az Apromore kiszámítja az egymást követő tevékenységpéldány-párokot, és aggregálja teljesítménymetrikáikat (pl. esetgyakoriság, teljes gyakoriság stb.), ezzel kinyerve egy teljes közvetlenül követő grafikon (DFG) a naplóból. Ezután a felhasználó szűrheti a DFG-t a kívánt szintre (ehhez az íveket egyesével eldobják, de olyan íveket nem, amelyek megszakítanák a grafikon). Az Apromore a Split Miner algoritmust használja a DFG BPMN-modellé alakításához.

Az UiPath egy globális szoftvercég, amely robotikus folyamatautomatizálási szoftvereket készít. Automatikusan felfedezik az üzleti folyamatokat, és megértik, hol nyújtja az automatizálás a legtöbb értéket. Az UiPath Process Mining egy rendkívül testre szabható folyamatbányászati platformot kínál a szabadalmaztatott, védjeggyel védett TRACY algoritmusával, amely gyorsan rendereli a folyamatgrafikonokat. Ez úgy lett tervezve, hogy egyszerűsítse a folyamatgrafikonokat és a hozzájuk kapcsolódó vizuális élményt. Ez magában foglalja a finom, de hatékony tulajdonságokat, mint például a grafikus tervezés folytonosságának fenntartása, az új adatok hozzáadása vagy eltávolítása, illetve a szűrt adatok kiválasztásának megváltoztatása során. A folyamatfelfedezéshez bevezették a valószínűségi induktív bányász (PIM) algoritmust, amely az induktív bányász keretrendszeren alapul. A PIM minden lépésben összehasonlítja a legvalószínűbb operátorokat és szerkezeteket az adatok gyakorisági információi alapján, ami blokk szerkezetű modellekben eredményez jelentősen nagyobb pontosságot.

A gráf neurális hálózatok (GNN-ek) fejlődése az 1990-es évek végére nyúlik vissza, és fontos szerepet játszottak a gépi tanulás és a mély tanulás területén, különösen olyan adatszerkezetek kezelésében, amelyek gráf formájában vannak ábrázolva, mint például közösségi hálózatok, molekuláris struktúrák és kommunikációs hálózatok.

Az első lépéseket Franco Scarselli és társai tették meg 2008-ban, amikor formalizálták a gráf neurális hálózatok alapvető modelljét. Ez a modell lehetővé tette, hogy a neurális hálózatok iteratív módon frissítsenek csúcspontokra vonatkozó

információkat a gráfon belül, amíg el nem érik a stabil fixpontot. Ez a megközeletítés új lehetőségeket nyitott meg a gráfokkal kapcsolatos prediktív feladatokra, például csúcspontok osztályozására vagy gráfok egészének regressziójára.

A 2010-es évek elején további fejlődés történt, amikor a mély tanulás rohamos előrehaladása a konvolúciós neurális hálózatok (CNN) területén inspirálta a kutatókat, hogy hasonló technikákat alkalmazzanak gráfokra. Ez vezetett a gráf konvolúciós hálózatok (GCN) kifejlesztéséhez, amelyeket 2016-ban mutattak be. A GCN-ek lehetővé tették a gráfadatokon történő mélyebb és hatékonyabb tanulást azáltal, hogy a csúcspontok környezetéből származó információkat aggregálták.

Azóta a GNN-ek területén számos innováció történt, beleértve a gráf figyelő (attention) hálózatokat, mint például a Graph Attention Networks (GAT) i. Ezek a modellek dinamikus súlyozást alkalmaznak a szomszédos csúcspontok jellemzőire, lehetővé téve a hálózat számára, hogy kiemlje a fontosabb kapcsolatokat a tanulási folyamat során.

A GNN-ek fejlődése mára eljutott odáig, hogy széles körben alkalmazzák őket különböző iparágakban és tudományos kutatásokban, ahol a gráfstruktúrák kiemelt szerepet játszanak. Ezek a hálózatok jelentősen javították a gépi tanulási modellek teljesítményét és adaptálhatóságát gráfalapú adatokon.

A GNN-architektúra több kulcsfogalmat foglal magában. A gráf mérete, amelyet egy átlós mátrix (D) ábrázol, a csomópontonkénti kapcsolatok számát jelöli. A Laplace-mátrix, amelyet gyakran $L = D - A$ (Adjacency mátrix) néven jelölnek, a gráfszerkezeten belüli egyes csomópontok hatását rögzíti. A grafikonok normalizálása a kapcsolódási számok alapján általános gyakorlat, ami lehetővé teszi az áttekinthetőbb elemzést. A gépi tanulási feladatokban a gráfok értékes bemenetként szolgálnak, megkönnyítve a csomópontok, élek vagy teljes gráfszerkezetek előrejelzését. A gráfábrázolás egyszerűsítése érdekében kulcsfontosságú a csomópontok beágyazott terekbe történő kódolása, miközben megőrizük hasonlósági kapcsolataikat.

A gráfbeágyazásoknál az elsődleges cél az eredeti gráf és a beágyazott gráf közötti eltérések minimalizálása. Különböző kódolási technikákat, például mátrixműveleteket, keresési táblázatokat és véletlenszerű lépésekből származó statisztikai dekódolást alkalmaznak ennek a célnak a hatékony elérése érdekében. A végső cél továbbra is az eredeti és a beágyazott hasonlósági struktúrák közötti különbségek csökkentése a gráf kódolási eljárásokban.

A kísérletek elvégzéséhez a Fraunhofer Institute folyamatbányászati feladatok megvalósításához összeállított pm4py Python programkönyvtárát használtuk. A pm4py könyvtárból a folyamatfeltáró algoritmusokat teszteltük. A folyamatfeltáró algoritmusok bemenete egy eseménynapló, a kimenete pedig a feltárt folyamat leírása. A bemeneti eseménynaplófájlokat TXT-formátumban

kaptuk meg, amelyeket át kellett konvertálni CSV-formátumra. Ezekre futtattuk le a 12 vizsgált algoritmust, amelyek különböző formátumban állítják elő a feltárt folyamatmodellt.

A vizsgált algoritmusok aszerint csoportosítva, hogy milyen folyamatmodellt állítanak elő:

Deklaratív modellt előállító algoritmusok:

- EFG- (Eventually-follows graph) gráfot előállító algoritmus
- Log skeletont (eseménynaplóvázlat) előállító algoritmus

Grafikus modellt előállító algoritmusok:

- DFG-(Directly-follows graph) gráfot előállító algoritmus
- Petri-háló előállító Alpha Miner, Inductive Miner, Heuristic Miner és ILP Miner algoritmusok
- Folyamatfát (process tree) előállító Inductive Miner algoritmus
- Heurisztikus háló (heuristics net) előállító Heuristic Miner algoritmus
- BPMN- (Business Process Model Notation) gráfot előállító Inductive Miner algoritmus
- Állapotátmeneti gráfot (transition system) előállító algoritmus
- Prefix fát előállító algoritmus

1.3. A kidolgozott kísérletek/modellezések módszertana

Az első feladatrészben a naplóbejegyzések, nyomok előfeldolgozását végeztük el.

Egyes problématerületeken, ahol sémaindukciót alkalmaznak, az eseménynapló heterogén lehet, ami azt jelenti, hogy különböző folyamatmodellekből származó nyomokat tartalmaz. Például egy naplószerver egyszerre több üzleti munkafolyamatot is rögzíthet, például termékrendelést és gyártási logisztikát, néha még ugyanazon eszközök használatával is. Ez azt eredményezi, hogy a nyomkövetési naplók heterogenitása miatt az indukciós motor összetettebb sémagráfot hoz létre.

Az adatkészlet egyszerűsítése és tisztázása érdekében az előfeldolgozás magában foglalja a bemeneti nyomok szétválasztását. Ez a szétválasztás homogénebb nyomvonalcsoportokat eredményez, amelyek egyszerűbb sémagráfokat eredményeznek, amelyek könnyebben érthetőek és értelmezhetőek.

A bemeneti nyomok particionálása különféle klaszterezési módszerekkel valósítható meg. A k-means módszer népszerű, de a vektorterekre korlátozódik, ahol a klaszterek centroid vektorait számítja ki. A tárgyalt kontextusban azonban, ahol az adatobjektumok nyomkövetések, nincs szabványos vagy hatékony vektoros ábrázolása ezeknek a nyomoknak.

Ehelyett egy általános metrikus teret használnak, ahol az egyetlen ismert adat az objektumok közötti távolságok távolságmátrixban ábrázolva. Ez olyan klaszterezési módszer alkalmazását teszi szükségessé, amely vektorterek helyett metrikus terekre alkalmas.

Tekintettel erre a megszorításra, a hierarchikus agglomeratív klaszterezés (HAC) használatos. A HAC azzal kezdődik, hogy minden objektumot egyszemélyes fürtként kezel. Minden következő lépésben a két legközelebbi fürt összevonásra kerül, hogy új klasztert hozzon létre.

A folyamat a befejezés két fő kritériumával folytatódik: a fürtök kívánt számával és egy előre meghatározott távolsági küszöbértékkel, amelyen túl a fürtöket már nem szabad összevonni. Ez a módszer hatékonyan rendezi a nyomokat kezelhetőbb és relevánsabb csoportokba, megkönnyítve a séma jobb indukcióját összetett heterogén naplóból.

A kidolgozott klaszterezésben a HAC-módszert alkalmazzuk és a legrövidebb elemenkénti távolságot használjuk két klaszter közötti távolság mérésére. A klaszterező algoritmus pszeudo kódja a következőben foglalható össze.

1. A C0 klaszterkészlet inicializálása.
2. A távolságok kiszámítása.
3. A klaszterek összevonása agglomeratív módon (HAC) a bemutatott felhasználásával minimális távolságú megközelítés.
4. Az összevonási távolságok viszonyát jelző szekvencia felépítése.
5. A szekvencia lokális csúcserkéinek a meghatározása.
6. A csúcsok rendezése csökkenő sr-értékek szerint.

A javasolt módszer fő előnyei:

- integrálható a HAC-algoritmusba
- alacsony többletköltség
- réteges hierarchikus klaszterezési struktúrát is képes felfedezni

A sémagenerálás során az egyik vizsgált módszer a nyomok illesztésén, hasonlóságán alapuló módszer. A módszer egyenként adja hozzá a sémához a naplóból kiolvasott nyomokat a sémagráfhoz.

A megszerkesztett sémafát az FCA által megszerkesztett fogalomrácsgráf feszítőfájának is tekinthetjük. Ebben a fában minden sorozatmintának egyetlen szülője van, és egy egyedi tárolócsomópontja van. Ez azt jelenti, hogy nincs két különböző csomópont ugyanazzal a műveletszekvenciával.

Ha például vesszük az alábbi nyomokat:

[a',b'],

[c',d'],

[a',b',c',d']

akkor a módszer az alábbi metszeteket generálja a fában:

0 : [' _ ' , * ' , # '] : [3 , 4] []

1 : [' _ ' , ' a ' , ' b ' , # '] : [] [3]

2 : [' _ ' , ' c ' , ' d ' , # '] : [] [4]

3 : [' _ ' , ' a ' , ' b ' , * ' , # '] : [1 , 5] [0]

4 : [' _ ' , * ' , ' c ' , ' d ' , # '] : [2] [0]

5 : [' _ ' , ' a ' , ' b ' , ' c ' , ' d ' , # '] : [] [3]

Az FCA módszert alkalmazva a következő fogalomkészlet jön létre:

0 : [' _ ' , * ' , # '] : [1 , 2] []

1 : [' _ ' , ' a ' , ' b ' , # '] : [3] [0]

2 : [' _ ' , ' c ' , ' d ' , # '] : [3] [0]

3 : [' _ ' , ' a ' , ' b ' , ' c ' , ' d ' , # '] : [] [1,2]

A generált sémafa a következő szempontokban tér el az FCA-rácsgráftól:

1. A sémafában az intent rész egy sorozatot jelöl, nem egy halmazt.
2. A sémafában új speciális szimbólumok/attribútumok vannak az összetett attribútumcsoportok jelölésére, míg az FCA-rács csak atomi attribútumokat használ.
3. A sémafában több csomópont van, mivel a csomópontok globális sémát jelölnek, nem csak helyi attribútumokat.

A hierarchiafelépítési algoritmus a következő lépésekben foglalható össze.

1. A bemenet S sorozatok halmaza.
2. Inicializálja az eredménykészletet R üres listára.
3. Folyamatosan dolgozd fel a $s \in S$ sorozatokat:
 - a) Végezze el az általánosítási metszéspontot a R sorozataira.
 - b) Adjon hozzá minden új metszéspontsorozatot a R -hez
4. Hozzon létre egy listát a különböző R -beli sorozathosszúságokról
5. Adjon hozzá minden $s \in R$ sorozatot a $|s|$ -hez rendelt csoporthoz
6. Folyamatosan dolgozza fel a sorozatokat növekvő vödör sorrendben (először vegye a legrövidebb sorozatokat):
 - a) hurok a már feldolgozott vödrök elemeire (kisebb méret)
 - b) teszt az általánosítási összefüggésről
 - c) tesztelje a szomszédosságot, hogy meghatározza a gráf szülőcsomópontjait.

A javasolt gráfkészítési módszer alkalmas az összes általánosított sorozat és a szomszédság meghatározására. Ha három csomópontot s , t , v és $s < v$, $t < v$ -t veszünk, akkor láthatjuk, hogy a v összes eseményeleme a s és t elemekben is előfordul és minden más elemnek más-más példánya van a s és t -ban. Így a kapott hierarchiát figyelembe véve a motor el tudja készíteni az általánosított, XOR-ágakat is tartalmazó sémagráfot.

A bemutatott általánosítási módszerekkel sorozatokat és feltételes ágakat tartalmazó sémagráfokat lehet készíteni (XOR). Egy másik pillérnél, a hurokstruktúránál, a hurokminták felderítésére egy további modult kell alkalmazni.

A hurokminták megtalálásához a motor tartalmaz egy hurokérzékelő modult is, amelyet első lépésként alkalmazunk a faépítés elvégzése előtt.

A kidolgozott sémageneráló algoritmus egy veszteségmentes módszer, amely lefedti az összes bemeneti példányt, a kapott grafikon nagyon összetett és hatalmas lehet. Másrészt nagy bemenetek esetén lehet, hogy csak a fontos, kulcsfontosságú sémamintákra van igény. A redukált sémagráfok előállításához két redukciós módszert alkalmazunk:

- Mélység alapú redukció: A sémafa felépítésénél mélység határt adunk meg, csak azokat a csomópontokat veszi be, ahol a csomópont mélysége egy küszöbérték alatt van.
- Frekvencia alapú redukció: Minden csomópont-hoz kiszámolhatunk a bemeneti adatkészletből származtatott frekvenciaértéket. Csak azokat a csomópontokat tartalmazza, ahol a csomópont frekvenciája egy küszöbérték felett van.

A nyomillesztésen alapuló modell mellett egy másik vizsgált megközelítés a gráf neurális háló alapú módszer alkalmazási lehetőségeinek a vizsgálata, egy GNN-alapú módszer kidolgozása volt. Az NN-alapú sémagráf-generálás probléma körében a célunk a mélygeneratív modellek elemzése volt. Ezek a modellek két, széles körben elismert módszert fognak beépíteni a mélygeneratív hálóba: VAE- és GAN-hálókat.

A VAE egy valószínűségi modell, amelynek célja a bemeneti adatok mögöttes eloszlásának megismerése. Ezek egy kódoló hálózattól állnak, amely a bemeneti adatokat látens térbe képezi le, és egy dekódoló hálózattól, amely rekonstruálja a látens tér bemeneti adatait. A VAE-k úgy vannak kiképezve, hogy minimalizálják a rekonstrukciós veszteséget, ugyanakkor maximalizálják a tanult látens tér és egy előre meghatározott előzetes eloszlás, tipikusan Gauss-eloszlás közötti hasonlóságot.

A GAN-architektúra két neurális hálózatból áll: egy generátorból és egy diszkriminátorból. A generátor megtanul szintetikus adatmintákat generálni, amelyek hasonlítanak a valódi adatokra, míg a diszkriminátor megtanulja megkülönböztetni a valós és a generált mintákat. A GAN-ok kontradiktórius módon vannak kiképezve, ahol a generátor arra törekszik, hogy megtévesse a diszkriminátort azáltal, hogy egyre valóságosabb mintákat generál, míg a diszkriminátor arra törekszik, hogy pontosan különbséget tegyen a valódi és a hamis minták között.

E két hatékony megközelítés kombinálásával mélygeneratív modelleket kívánunk kifejleszteni, amelyeket kifejezetten gráfstrukturált adatok generálására terveztek. Ezek a modellek lehetővé teszik számunkra, hogy megtanuljuk a gráfadatok bonyolult mintáit és összefüggéseit, és új gráfpéldányokat generáljunk, amelyek nagyon hasonlítanak a valós gráfokhoz. Egy kódoló neurális hálózat leképezi a bemeneti gráfot $G = (A, X)$ a $q\varphi(Z|X)$ utólagos eloszláshoz a Z látens változók felett. Ebből a posteriorból vett minta alapján a $p\theta(X|Z)$ dekódoló modell megpróbálja rekonstruálni a szomszédsági mátrixot.

A kezdeti cél a szignifikáns gráfok megfelelő értelmezése a kódolt látens reprezentációkból, különösen, ha tanító grafikonokkal jelenítik meg őket. Eközben a másodlagos cél egy szabályosító mechanizmusként szolgál, amely garantálja, hogy értelmes gráfokat értelmezzünk még akkor is, ha a látens reprezentációkat a korábbi eloszlásból mintavételezzük, amelyet $p(Z)$ -vel jelölünk). Ez a második cél kiemelkedő fontosságú, különösen azokban a forgatókönyvekben, ahol a képzés után új grafikonok létrehozása kívánatos. A generálási folyamat az előző mintavételen és ezeknek a látens beágyazásoknak a dekóderbe történő bevételén alapul, és csak akkor működik hatékonyan, ha a második cél teljesül.

A kidolgozott módszerek következő változata a GNN-háló olyan adaptálásán alapszik, amely lehetővé teszi a sémagráf nyomokból történő felépítését. Az irodalomban talált módszerektől eltérően ez a módszer nem tisztán GNN- vagy tisztán mintaillesztés-alapú, hanem inkább e kettő ötvözetének tekinthető. Ez alatt azt értjük, hogy mindkét motort alkalmazza az optimális séma feltáráshoz. A módszer elve a következő lépéseken alapszik.

A: tanító halmaz előállítás

1. A naplófile-ből kiemelt nyomok összevonása egyetlen gráffá, ez lesz a tanító gráf.
2. A nyomokból az események típusainak a címkézése, azaz két elem akkor kap azonos jelzőt, ha azonos üzleti tartalommal bír. Egy nyomban egy jelző több elemi lépésnél is előfordulhat.

3. Az elemi műveletek között egy ekvivalencia reláció értelmezést vezetünk be: két esemény akkor és csak akkor áll relációban egymással, ha azonos címkét viselnek, azaz azonos jelentésűek.
4. Az egyes elemi eseményekhez a környezetleíró vektor (rejtett állapot) bevezetése
5. Egy GNN felépítése, amely üzenetváltással, konvolúcióval meghatározza az egyes elemi események rejtett állapotait.
6. Az elemi eseménypárokból olyan tanító minta előállítása, melyben a bemenet a pár rejtett vektorai és a címke a reláció címkéje, amely tehát azt jelzi, hogy a két esemény azonos típusú-e vagy sem.
7. GNN betanítása a fenti tanítóhalmazon.

Az első fázis után a következő lépés a mintanapló feldolgozása

1. A nyomok egyenkénti hozzáadása a vizsgált G gráfhoz.
2. Az új nyom eseményeinek sorba vétele és a betanított GNN alapján annak eldöntése, hogy a G korábbi elemei közül melyek állnak címke- (jelentés-) azonosságban a beérkező új nyom műveleteivel.
3. Az azonos műveleti elemek egy csomópontba összevonása.
4. Az élek aktualizálása.

A megadott módszer segítségével az induló nyomok egy közös gráfba vonhatóak össze. Az algoritmus hatékonyan ki tudja emelni az elágazásokat, opcionális ágakat.

A ciklusdetektálás fő nehézsége, hogy az egzakt illesztés pontosabb elemzést tud adni egy szakasz ismétlődésére, a neurális háló csak valószínűségi modellt tud készíteni. Az NN-alapú architektúra megvalósításához bevezettünk egy ciklusgeneráló részt is.

A CNN neurális hálózatok architektúráját az 1980-as években fejlesztették ki képosztályozásra. Felépítését és funkcióját tekintve a CNN-hálózat biológiai eredetű. A fő különbség a CNN, azaz a konvolúciós neurális hálózatok és a hagyományos hálózatok között az, hogy a hagyományos neuronrétegek mellett úgynevezett konvolúciós rétegeket is tartalmaznak.

A konvolúciós rétegek a szűrőablakokat mozgatják a mátrix mentén. A szűrőablak egy $n \times n$ mátrixnak tekinthető, amely a kiszámítandó adatok mátrixára kerül. A művelet következő lépésében az egyes értékeket megszorozzuk, majd összeadjuk. Ezt követően az ablakmátrix egy egységgel eltolódik.

A műveletsor megismétlődik a teljes adatmátrixon. Ennek az ablakmátrixnak a mérete tetszőleges lehet. A betanítási folyamat előtt a szűrőablak értékei tetszőlegesek. A betanítási folyamat során a szűrőablak értékeit úgy változtatjuk

meg, hogy megtanuljanak egy releváns információt kiemelni a mátrixból. A konvolúciós rétegeket általában egy gyűjtőréteg követi, ami csökkenti a konvolúció által előállított mátrix méretét. Hasonlóan a konvolúcióhoz, meghatározott ablakmérettel mozog a mátrixon, és valamilyen függvény segítségével leképezi az ablakban lévő értékeket, általában a CNN-eknél maximum kijelölést vagy átlagolást használnak. Manapság egyre több alkalmazási területen alkalmazzák a konvolúciós neurális hálózatokat. A legnagyobb terület a képfeldolgozás. Képfeldolgozási hatékonyságuk azonban olyan magas, hogy gyakran más problémák megoldására használják az adatok képpé alakításával.

Egy neurális hálózat betanításához nagy mennyiségű példaadatra van szükség, ezért létrehoztunk egy adatkészlet-generátort. A generátor feladata ismétléseket tartalmazó karakterláncok generálása. A tesztadatkészlet-generátor motorja egyenletes véletlenszerű eloszlást használ a karakterláncok felépítéséhez véletlenszerű tandem szakaszokkal vagy tandem részek nélkül.

Ha nincs ismétlés, akkor a karakterek úgy jönnek létre, hogy ne legyenek véletlen ismétlések. Ha ismétlődő résszel rendelkező karakterláncról van szó, akkor először meghatározzuk az ismétlés pozícióját és az ismétlődő rész hosszát, majd az előző módon generáljuk a sorozatot az ismétlés helyéig. Az ismétlődő rész elérésekor válassza ki az ismétlődő részstringben használt karaktersorozatot, és fűzze egymás után a j darab i karaktersorozatot. A karakterlánc többi része ismétlés nélkül jön létre.

Az ismétlés a létrehozott képen a főátlóval párhuzamos pixelsorként jelenik meg. Ezzel az ábrázolási formátummal a CNN könnyen felismeri az ismétlődéseket. A mátrix előállítása a főátlón lévő szimmetriájának kihasználásával optimalizálható, így a számítási költség felére csökken. Van azonban egy probléma, hogy egy m hosszúságú karakterlánc esetén a neurális hálózat bemeneti neuronjainak száma m^2 lesz. Így a CNN-képzés vagy ismétléskeresés során viszonylag gyorsan elérjük a hardverhatárainkat. A probléma megoldására a következő redukciós módszereket fejlesztettük ki:

- Bittérképalapú csökkentés
- Átlagaggregátor
- Korrigált redukció.

A korrigált redukálási módszernél hasonló szomszédokat keresünk átlóirányban, így nagyobb értéket adunk azoknak az almátrixoknak, ahol a rész mátrix főátlója 1-es.

A kidolgozott módszereknek a meglévő módszerekkel történő összevetéséhez kidolgozásra került egy benchmark környezet.

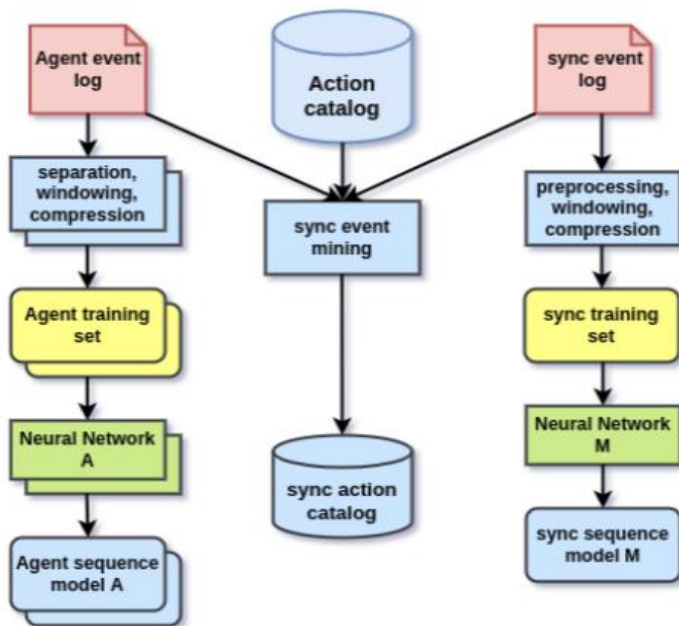
A benchmark folyamatmodellek előállításához és az azok értékelését lehetővé tevő mérések elvégzéséhez az alábbi lépések végrehajtására volt szükség:

1. Python értelmező, fejlesztőkörnyezet és a pm4py programkönyvtár telepítése
2. Tesztadathalmazok előállítás
3. Tesztadathalmazok átalakítása CSV-eseménynapló formátumra
4. CSV-állományok betöltése, statisztika készítése (esetek száma, összes események száma, eseménytípusok száma)
5. A vizsgált algoritmusok lefuttatása a tesztadathalmazokra, a feltárt modellek elmentése
6. A futási eredmények kiértékelése, a feltárt modellek összehasonlítása

1.4. A módszer lépéseinek, a referencia folyamatok részletezése

Neurális háló alapú tanulóalgoritmus kidolgozása

Az esemény sémagráf gépi tanuláson alapuló generáláshoz egy egyedi neurális háló modellt dolgoztunk ki. A javasolt új modell több különböző neurális hálózati egységet épít be a különböző ágensek modellezésére. A szereplőszintű modellek mellett egy fő szintű modell, a szinkronizációs események szintje is szerepel a keretben. A javasolt architektúra az ábrán látható.



Rendszermodell ábrája

Az architektúrában a fő bemenetek az ügynökszintű eseménynaplók és a műveletkatalógus, amely leírja a műveletek célobjektum szintű függőségét. Ezen adatok alapján a rendszer először a szinkronizálási események bányászatát végzi el, létrehozva a szinkronizációs események katalógusát, amely többek között tartalmazza a bemeneti és kimeneti szereplőlistákat. A motor eseménynaplót készít a szinkronizálási eseményekhez és egy sorozatmodellt, vagy a szinkronizálási eseményeket egy neurális hálózati egység segítségével építi fel.

A következő fázisban a motor szétválasztja az eseménynaplót az ágensek szerint, és a javasolt neurális hálózati modell segítségével, jellemzővektor-tömörítéssel létrehozza az ügynökszintű sorozatmodellt. A predikció bemenetét az előtagsorozat határozza meg, amelyet szintén eseményláncnak tekintünk.

A beviteli előtag formátuma one-hot kódolást használ. Így minden pozícióhoz van egy fix méretű vektorunk, ahol a hossza megegyezik a különböző eseménytípusok számával. Egy szakaszban az e esemény adott pozíciójának értéke 1, ha e szerepel a listában az adott pozícióhoz az előtagsorozatban. Az aggregációs MLP-modulok kimenetei szintén egy-hot kódolású vektorok. Ezek a vektorok egyetlen vektorba vannak összefűzve. Így a fő predikciós MLP-modul kimenete egy one-hot kódolású vektor, ahol minden eseménytípusnak van egy markerpozíciója, amelyet az aggregációs hálózatok számítanak ki.

A fő szint előrejelzése a következő algoritmuson alapul:

Algoritmus. Először veszünk egy üres előtagsorozatot, és megjósoljuk a fő sorozat elemeit. Minden iterációban:

1. az előtag kibővül az előző lépésben megjósolt eseménnyel,
2. a motor megjósolja a következő eseményt az új előtag alapján.

Ha a vége szimbólum a nyerő, vagy a sorozat hossza túllép egy adott határt, akkor az előrejelzési folyamat leáll.

A javasolt módszer másik sajátossága, hogy a hagyományos megközelítésekkel ellentétben a motor több jelöltet generál kimenetként, nem csak a legjobbbat. A standard megközelítésben a motor a kimeneti vektorban a legmagasabb rangú kategóriát választja ki. Megközelítésünkben minőségi küszöböt használunk a legjobb jelöltek meghatározására. Csak azok a jelölt események kerülnek kiválasztásra, amelyeknél (a neurális hálózat által kimenetként generált) alkalmassági érték megfelel a szűrőfeltételnek.

Az előzőben ismertetett modell mellett kidolgozásra került egy másik, a nyomok hasonlóságán, illesztésén alapuló módszer is. Ebben a megközelítésünkben két sorozat általánosítása a közös szakaszok kinyerésén alapul. Az alkalmazott szek-

venciametszés-művelet a Levenstein szerkesztési távolság algoritmusadaptációján alapul. Az algoritmus először meghatározza az átalakítási utat a minimális szerkesztési költséggel. Ezután azon lépésekben, ahol az illesztett karakterek eltérőek, létrehozunk egy „*” szimbólumot. Az illesztési függvény kódja:

```
def s_generalize (s1,s2):
    path = Levenstein(s1,s2)
    seq = ""
    for p in path:
        if s1(p_1) == s2(p_2):
            seq = seq + s1(p_1)
        else:
            if seq[-1] != '*':
                seq = seq + '*'
    return seq
```

Az általánosítási hierarchia létrehozásának módszere a Close by One (CbO) algoritmuson alapul, amely a fogalmakat a koncepció attribútumvektorának a mérete szerint rendezetten rekeszekbe képezi le. Ha rekeszrendben vesszük az elemeket, akkor csak egy irányt kell vizsgálatot végezni a kettő helyett a szomszédos elemek keresésénél. Így a rácsépítés költsége jelentős mértékben csökkenthető.

A hierarchia felépítési algoritmus a következő lépésekben foglalható össze.

1. A bemenet S a nyomok, sorozatok halmaza.
2. Inicializálja az eredménykészletet egy üres listára.
3. Folyamatosan feldolgozza az egyedi nyomokat:
 - (a) az általánosítás elvégzése, az eddigi részeredmények eredményeivel képzett metszetek meghatározása;
 - (b) az új fogalmak hozzáadása az eddigi részeredmények halmazához.
4. A kapott sorozatok letárolása a hosszuk alapján egy külön listába.
5. A lista elemeinek feldolgozása a sorozatok növekvő mérete alapján.
 - (a) ciklus a már feldolgozott rekeszek elemeire (kisebb méret)
 - (b) az általánosítási képesség ellenőrzése
 - (c) a szomszédossági teszt elvégzése, a gráf éleinek a meghatározása.

A bemutatott általánosítási módszerekkel sorozatokat és feltételes ágakat tartalmazó sémagráfokat lehet készíteni (XOR). Egy másik pillérenél, a hurokstruktúrájánál, a hurokminták felderítésére egy további modult kell alkalmazni.

A hurokminták megtalálásához a motor tartalmaz egy hurokérzékelő modult is, amelyet első lépésként alkalmazunk a faépítés elvégzése előtt. A módszer lépései:

- Ciklus a különböző jelölt ciklusmagokon:
 - (a) Hurok a lehetséges iterációkon.
 - (b) Számítsa ki a kernel súlyát!
 - (c) Mentse el a legjobb jelöltet!
- Végezze el a ciklusmag detektálást rekurzív módon a komplementer szekvencia szakaszokban!
- Adja vissza a legjobb ciklusmag készletét a kapcsolódó feltárt iterációk darabszámával együtt!

Mivel a bemutatott sémageneráló algoritmus egy veszteségmentes módszer, amely minden bemeneti példányt lefed, az eredménygráf nagyon összetett és hatalmas lehet. Másrészt nagy bemenetek esetén lehet, hogy csak a fontos, kulcsfontosságú sémamintákra van szükség. A redukált sémagráfok előállításához két redukációs módszert alkalmazunk:

- Mélységalapú redukció: A sémafa felépítésénél mélységhatárt adunk meg, csak azok a csomópontok tartoznak ide, ahol a csomópont mélysége egy küszöbérték alatt van.
- Frekvencia alapú redukció: Minden csomóponthoz kiszámoljuk a bemeneti adatkészletből származtatott frekvenciaértéket. Csak azokat a csomópontokat tartalmazza, ahol a csomópont frekvenciája egy küszöbérték felett van.

Mint már említettük, a ciklusok feltárása külön modul alkalmazását igényli. A rendszerünkben egy konvolúciós háló alkalmazását céloztuk meg. A nyomadatok előfeldolgozásával, mátrixos formátumra alakítással kapcsolatos követelmények:

- A tandem részsstringek jellegzetesen jelenjenek meg a mátrixon.
- A bemenetet minél kisebb dimenziójúra csökkentjük.
- A neurális háló egy hosszúságú sstringben minden lehetséges tandem részsstringet megtalálhasson.

Az általunk használt megoldás egy egyértelmű leképezést jelent. A teljes sstringet, amely m karakter hosszú, egy $m * m$ -es mátrixra alakítjuk. A létrehozott egyezős mátrixban egy elem értéke vagy 1 vagy 0. Az 1 érték esetén a pozíció index szerint kijelölt elemek azonosak. A 0 érték eltérő értékeket jelöl.

	a	l	a	l
a	1	0	1	0
l	0	1	0	1
a	1	0	1	0
l	0	1	0	1

	a	l	m	a
a	1	0	0	1
l	0	1	0	0
m	0	0	1	0
a	1	0	0	1

1. táblázat. Egyezés mátrix

A javaslatban kidolgozott módszer által használt konvolúciós neurális háló egyszerű szerkezetű.

A háló elkészítéséhez az egyszerűség és a számunkra szükséges könyvtárak elérhetősége miatt a Python nyelvet választottuk. A keras keretrendszer segítségével sikerült könnyedén felépíteni a neurális hálót és tanítani azt.

Benchmark környezet kifejlesztése

A szintetikus előállított TXT-fájlok sorai vesszővel elválasztott események sorozatát tartalmazzák. Egy eseményt a szimbolikus nevével (a tevékenység megnevezését helyettesítő betűjellel) adunk meg. A folyamatpéldány azonosítóját (case_id) és az időbélyeget automatikusan adjuk hozzá az átalakítás során.

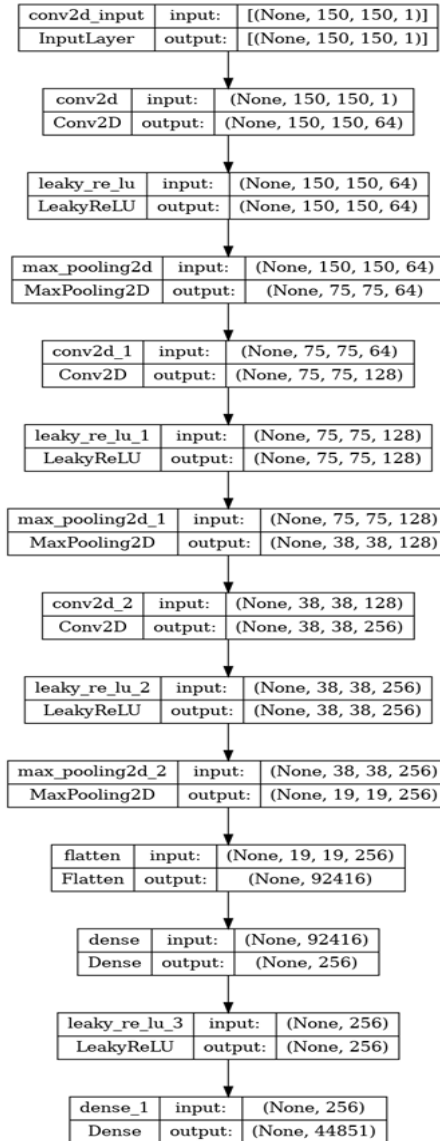
A vizsgált folyamatfeltáró algoritmusok implementációja egységes a pm4py programkönyvtárban. Ez azt jelenti, hogy minden függvény ugyanazokkal az argumentumokkal hívható. Mégpedig, a bemeneti eseménynaplót, az esetazonosítót (case_id) tartalmazó adatszlop nevét, a tevékenység/esemény megnevezését tartalmazó adatszlop nevét, valamint az időbélyeget tartalmazó adatszlop nevét kell megadnunk. A függvények visszatérési értéke az előállított folyamatmodell-leírás, amit grafikus modell esetén a GraphViz segítségével tudunk megjeleníteni, és a megfelelő pm4py függvénnyel PNG-formátumban elmenteni. A deklaratív modelleket előállító algoritmusok kimenetét TXT-formátumban tudjuk eltárolni.

A pm4py csomagban implementált folyamatfeltáró algoritmusok leírása itt található: <https://pm4py.fit.fraunhofer.de/static/assets/api/2.7.11/api.html#process-discovery-pm4py-discovery>

A vizualizációhoz felhasználható függvények leírása ezen a linken keresztül érhető el: <https://pm4py.fit.fraunhofer.de/static/assets/api/2.7.11/api.html#visualization-pm4py-vis>

1.5. A módszert bemutató folyamatábrák, diagramok

Elsőként a ciklusok felismerésére szolgáló CNN-architektúra leírását mutatjuk be.



A ciklus feltárás során alkalmazott neurális háló szerkezete

A GNN alapú sémagráf generáláshoz kapcsolódóan két rutint veszünk példaként. Az egyik egy nyomot, a másik egy rész-sémát állít elő.

```
def build_subschema(self, sch, prev):
    p = 0
    #ni = len(self.INodes)
    ni = self.IN

    while p < len(sch):
        if sch[p] != 'X':
            self.INodes[ni] = {'t':self.LabDict[sch[p][0]], 'c':sch[p][1] }
            self.Edges['P'][ni] = [{}, {}]
            self.IN = self.IN + 1
            for nn in prev:
                self.Edges['P'][ni][1][nn] = {'t':0}
                self.Edges['P'][nn][0][ni] = {'t':0}
            prev = {ni}

            ni += 1
            p += 1
        else:
            prevnew = set()
            for subsch in sch[p+1]:
                sprev = self.build_subschema (subsch,prev)
                prevnew = prevnew.union(sprev)
            prev = prevnew

            p += 2

    return prev

def build_trace(self, tr):
    #tr = tr[1:-1]

    print (tr)
    p = 0
    #ni = len(self.INodes)
    ni = self.IN
```

```

for ee in tr:
    self.Nodes[ni] = {'l':self.LabDict[ee[0]], 'c':ee[1] }
    self.N = self.N + 1
    if p == 0:
        self.Edges['P'][ni] = [{ni+1: {'t':0}}, {}]
    else:
        if p < len(tr)-1:
            self.Edges['P'][ni] = [{ni+1: {'t':0}}, {ni-1: {'t':0}}]
        else:
            self.Edges['P'][ni] = [{} , {ni-1: {'t':0}}]
    p += 1
    ni += 1

```

A sémagraf-felépítés főbb, magas szintű lépései:

```

S2 = ['#','h','e','b','e','@']
S3 = ['#','h','e','b','e','@']
S4 = ['#','h','e','b','e','g','j','@']

```

```

gg = graph_C(LabDict,RLabDict)
gg.build_graph_2P(S2,S3)
gg.predict (NN, scaler)
gg.draw_graph()
gg.merge_subgraphs()
gg.draw_graph_O()

```

```

gg.build_graph_2R(S4)
gg.predict (NN, scaler)
gg.draw_graph()
gg.merge_subgraphs()
gg.draw_graph_O()

```

Az S2, S3, S4 listák az aktuális, tanításhoz használt nyomokat, eseménysorokat jelentik. Az eseménysorban külön foglalt jele van a kezdő- és zárólépésnek.

A sémák összevonását szolgáló lépés az alábbi metóduson alapul:

```
def merge_subgraphs(self):
```

```
    dels = []
```

```
    nn = self.N1
```

```
    if len(self.Edges['P'][nn][1]) > 0:
```

```
        print ("Edge error for node ",nn)
```

```
        return
```

```
    if len(self.Edges['S'][nn]) > 0:
```

```
        dels.append(nn)
```

```
    while len( self.Edges['P'][nn][0] ) > 0:
```

```
        nnn = [kk for kk in self.Edges['P'][nn][0].keys()][0]
```

```
        print ("Process ",nn, ":",nnn)
```

```
        if len(self.Edges['S'][nnn]) > 0:
```

```
            dels.append(nnn)
```

```
            nnr = self.Edges['S'][nnn][0]
```

```
        if nn in dels:
```

```
            print ("A")
```

```
            self.Edges['P'][nn][0] = dict()
```

```
            self.Edges['P'][nnn][1] = dict()
```

```
            nr = self.Edges['S'][nn][0]
```

```
            if nnr not in self.Edges['P'][nr][0].keys():
```

```
                self.Edges['P'][nr][0][nnr] = {'t':0}
```

```
                self.Edges['P'][nnr][1][nr] = {'t':0}
```

```
        else:
```

```
            print ("B")
```

```
            del self.Edges['P'][nn][0][nnn]
```

```
            self.Edges['P'][nn][0][nnr] = {'t':0}
```

```
            del self.Edges['P'][nnn][1][nn]
```

```
            self.Edges['P'][nnr][1][nn] = {'t':0}
```

```
        else:
```

```
            if nn in dels:
```

```
                print ("C")
```

```

        nr = self.Edges['S'][nn][0]
        del self.Edges['P'][nn][0][nnn]
        del self.Edges['P'][nnn][1][nn]
        self.Edges['P'][nr][0][nnn] = {'t':0}
        self.Edges['P'][nnn][1][nr] = {'t':0}
    else:
        print ("D")

    nn = nnn

for nn in dels:

    del self.Nodes[nn]

    del self.Edges['P'][nn]

self.Edges['S'] = dict()

def build_trace(self, tr):
    #tr = tr[1:-1]

    print (tr)
    p = 0
    #ni = len(self.Nodes)
    ni = self.N

    for ee in tr:
        self.Nodes[ni] = {'l':self.LabDict[ee[0]], 'c':ee[1] }
        self.N = self.N + 1
        if p == 0:
            self.Edges['P'][ni] = [{ni+1: {'t':0}}, {}]
        else:
            if p < len(tr)-1:
                self.Edges['P'][ni] = [{ni+1: {'t':0}}, {ni-1: {'t':0}}]
            else:
                self.Edges['P'][ni] = [{}, {ni-1: {'t':0}}]
        p += 1
        ni += 1

```


1.6. Mérföldkő során megvalósított munkafolyamatok rövid, összegző összefoglalása

A projektfázis fő feladata az üzleti és technológiai folyamatok megértését, elemzését és automatizálását támogató folyamat sémagráffeltárási algoritmus neurális hálón alapuló módszerének kidolgozása volt. A tervezett algoritmusok lehetővé teszik olyan komplex, nagy méretű folyamatok pontosabb reprezentációját és kezelését is, amelyek a hagyományos módszerekkel nem érhetők el. Az ilyen típusú algoritmusok fejlesztésének egy fő kihívása, hogy jelenleg még nem áll rendelkezésre hatékony neurális hálón alapuló módszer a sémagráfok automatikus generálására. A generálási folyamatban különösen a ciklusok és a mélyen egymásba ágyazott sémák kezelése okoz nehézséget.

A projekt első szakaszában a mintaillesztésen alapuló sémageneráló algoritmus tesztelését és finomhangolását végeztük el. Erre a fázisra azért volt szükség, mert általa jobban megérthetőek a komplex sémák kritikus elemei és szükség esetén lehetővé teszi egy kombinált módszer megvalósítását is, azaz amikor a neurális hálós motor együttműködik egy hagyományos módszerrel, ahol mindegyik modul egy-egy olyan feladatrészre koncentrálna, amelyben erősebb a másiknál.

A mintaillesztésen alapuló módszer fejlesztése és implementálása Python környezetben történt. Megvalósításra került egy olyan módszer, amely a nyomok egyenkénti bevonásánál megkeresi a gráf legjobban illeszkedő részét és ezt a részt integrálja a bejövő nyom elemeivel. A módszerben külön modul a ciklusok felismerését végző rész, amely egy valószínűségszámítási alapokon nyugvó eszközökkel határozza meg a ciklusmagok jelenlétét. Míg a korábbi, az alfa-algoritmus által létrehozott Petri-hálóakra épülő módszerek nem voltak képesek kezelni a zajokat vagy komplex hurokstruktúrákat, a javasolt módszer ezek kezelésére is alkalmas.

A nyomillesztésen alapuló modell mellett egy másik vizsgált megközelítés a gráf neurális háló alapú módszer alkalmazási lehetőségeinek a vizsgálata, egy GNN-alapú módszer kidolgozása volt. Az NN-alapú sémagráfgenerálás probléma körében a célunk a mélygeneratív modellek elemzése volt. Ezen modell kidolgozása keretében előállt egy alapalgoritmus, amely egy a gráf neurális háló alapú megközelítést vezetett be a sémagráfok bányászatára. A kidolgozott módszer az eseménycsomópontokhoz kontextusleíró vektorokat rendel és ezen vektorok tanítása révén lehetővé válik a csomópontok közötti hasonlóság feltárása. Ezáltal meghatározhatóak a bejövő lánc alakú nyomokat leíró gráfok integrálása, összevonása egy tömörebb sémagráfba.

A kutatási eredmények bemutatására több folyóiratcikket is készítettünk, melyek nagy része már megjelentetésre is került.

Az adott kutatási időszakban benchmark tesztekét készítettünk, amelyek eredményeivel a munkacsoport által elkészített algoritmusok hatékonysága összemérhető, és az előállított folyamatmodellek értékelhetők.

A pm4py programkönyvtárban implementált 12 folyamatfeltáró algoritmust lefuttattuk 19 szintetikusan előállított tesztadathalmazra. Az eredményül kapott folyamatmodellek benchmark modellként használhatók. A modellek értékeléséhez az alábbi mérőszámokat számítottuk ki: illeszkedés, pontosság, általánosság és egyszerűség. Ezekhez a számításokhoz a pm4py programkönyvtár kínál függvényeket, amelyek az alábbi argumentumokat várják: eseménynapló, Petri-hálóformátumban előállított folyamatmodell, a háló kiindulópontja, a háló végpontja.

A folyamatmodellek értékelhetősége érdekében végül csak azokat az algoritmusokat tartottuk meg a benchmark teszt halmazban, amelyek Petri-hálót állítanak elő és komplex szerkezetű (többszörös elágazást és ciklust is tartalmazó) eseménynaplók esetén is alkalmazhatók. Ezek a Heuristic Miner és az Inductive Miner algoritmusok.

2. Az elért eredmények bemutatása

2.1. A tesztek, kísérletek megvalósítási környezetének ismertetése

Az adott kutatási időszakban a Python folyamatbányászati feladatok megvalósításához összeállított pm4py programkönyvtárát használtuk a kísérletek elvégzéséhez. A pm4py a Fraunhofer Institute terméke, nyílt forráskódú, akadémiai és ipari projekteken egyaránt használható. A programcsomag dokumentációja a <https://pm4py.fit.fraunhofer.de/> linken érhető el.

A pm4py telepítéséhez és használatához legalább 3.9.x verziójú Python értelmező szükséges. A telepítés utasítása: `pip install -U pm4py`. Mi 64 bites Windows operációs rendszeren, Python 3.10.6 értelmező használata mellett telepítettük a programcsomagot. A pm4py függvényei egyéb Python-csomagoktól függenek, eltérő fontossági szinteken:

- Alapvető követelmények: numpy, pandas, deprecation, networkx
- Normál követelmények (alapértelmezés szerint a pm4py csomaggal együtt telepítve, fontos az általános használathoz): graphviz, intervaltree, lxml, matplotlib, pydotplus, pytz, scipy, stringdist, tqdm
- Opcionális követelmények (alapértelmezés szerint nincs telepítve): scikitlearn, pyemd, pyvis, jsonschema, polars, openai, pywin32, python-dateutil, requests, workalendar, pyngetputwindow, pyngetwindow

A pm4py könyvtárból mi a folyamatfeltáró algoritmusokat használtuk fel a tesztekhez. A folyamatfeltáró algoritmusok bemenete egy eseménynapló, a kimenete pedig a feltárt folyamat leírása.

A bemeneti eseménynaplófájlokat TXT-formátumban kaptuk meg, ahol a fájl egy sora egy folyamatpéldány (trace) leírása. Ennek átalakítására írtunk Python programot. Ez a program a megadott könyvtárban tárolt TXT-fájlokat CSV- és XES-eseménynaplókká alakítja, amelyeket a pm4py könyvtár folyamatfeltáró algoritmusaival dolgozunk fel.

A szintetikusan előállított TXT-fájlok sorai vesszővel elválasztott események sorozatát tartalmazzák. Egy eseményt a szimbolikus nevével (a tevékenység megnevezését helyettesítő betűjellel) adunk meg. A folyamatpéldány azonosítóját (Case_id) és az időbélyeget automatikusan adjuk hozzá az átalakítás során.

A pm4py könyvtárban elérhető algoritmusok különböző módon írják le a feltárt folyamatot. Mi az alábbi algoritmusokat vizsgáltuk.

Deklaratív modellel előállító algoritmusok:

- EFG (Eventually-follows graph) gráfot előállító algoritmus `efg = pm4py.discover_eventually_follows_graph(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
- Log skeleton-t (eseménynaplóvázlat) előállító algoritmus `log_skeleton = pm4py.discover_log_skeleton(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`

Grafikus modellel előállító algoritmusok:

- DFG- (Directly-follows graph) gráfot előállító algoritmus
- `dfg, start, end = pm4py.discovery.discover_dfg_typed(log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
- Petri-hálót előállító algoritmusok (Alpha Miner, Inductive Miner, Heuristic Miner, ILP Miner)
`net, im, fm = pm4py.discover_petri_net_alpha(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
`net, im, fm = pm4py.discover_petri_net_inductive(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
`net, im, fm = pm4py.discover_petri_net_heuristics(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
`net, im, fm = pm4py.discover_petri_net_ilp(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`

- Folyamatfát (process tree) előállító Inductive Miner algoritmus
`process_tree = pm4py.discover_process_tree_inductive(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
- Heurisztikus hálót (heuristics net) előállító Heuristic Miner algoritmus
`heu_net = pm4py.discover_heuristics_net(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
- BPMN- (Business Process Model Notation) gráfot előállító Inductive Miner algoritmus
`bpmn_graph = pm4py.discover_bpmn_inductive(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
- Állapotátmeneti gráfot (transition system) előállító algoritmus
`transition_system = pm4py.discover_transition_system(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`
- Prefix fát előállító algoritmus
`prefix_tree = pm4py.discover_prefix_tree(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')`

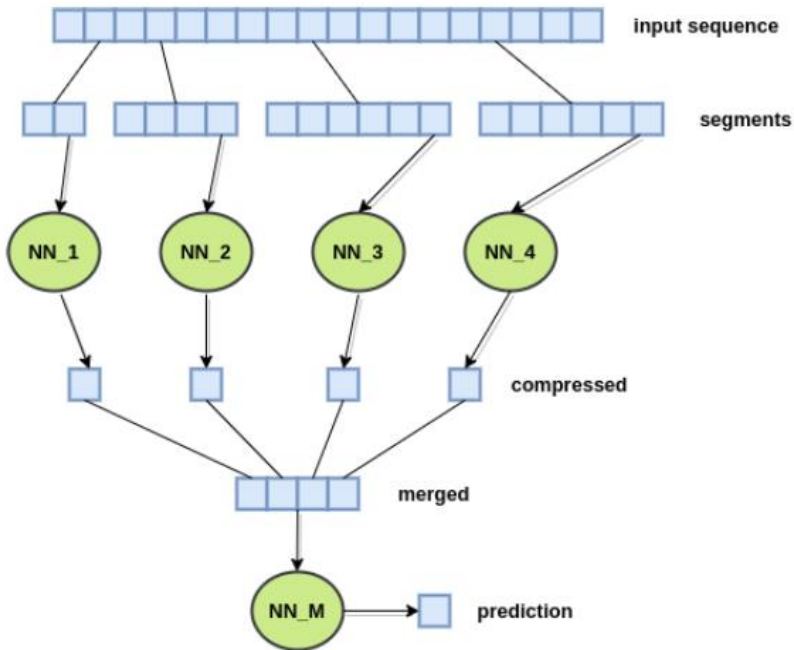
Az algoritmusok által előállított modellek értékelésekor 4 szempontot veszünk figyelembe:

- Illeszkedés: a feltárt modell mennyire illeszkedik az eseménynaplóban rögzített esetekre (azok hány százalékát fedt le). Vagyis az eseménynaplóban lévő esetek hány százaléka írható le a feltárt modellel.
- Pontosság: a feltárt modellből nem szabad olyan esetre következtetni, ami nem szerepel az eseménynaplóban. Azaz, ha a feltárt modellből legeneráljuk az összes lehetséges esetet, akkor ezek hány százaléka szerepel az eredeti eseménynaplóban.
- Általánosítás: a feltárt modell milyen mértékben általánosítja a valós eseteket.
- Egyszerűség: a feltárt modell mennyire egyszerű.

A komplex eseménysémák feltárására kidolgozott neurális háló architektúráját a következő oldalon látható ábra szemlélteti:

A háló bemenetét a legfelső sor szemlélteti. A bemeneti vektor megfelel egy hosszú eseménysornak. Első lépésben a bejövő vektort szegmensekre bontjuk, minden szegmens egy-egy időszakaszt jelöl ki. Ezt követően az egyes szegmensek külön-külön feldolgozásra kerülnek. A hozzárendelt neurális háló a szegmens idősorából egy hash értéket állít elő. A feldolgozás utolsó lépéséhez előbb ezen hash értékek egy eredő vektorba kerülnek összefűzésre, majd ezt az eredő

vektort átadjuk egy osztályozó neurális hálónak, ami becslést ad a sorozat következő elemére.



2.2. Tesztadathalmazok ismertetése

A tesztadathalmazokat a munkacsoport által készített, grafikus eseménynapló generáló szoftver segítségével állítottuk elő TXT-formátumban, az alábbi szisztematikus szerkezeti felépítést követve:

- XOR elágazást tartalmazó szekvenciális folyamatok:
 - o Egyszeres mélységű elágazásokat tartalmazó folyamatok (X) (3 fájl)
 - o Kétszeres mélységű elágazásokat tartalmazó folyamatok (XX) (2 fájl)
 - o Háromszoros mélységű elágazásokat tartalmazó folyamatok (XXX) (2 fájl)
- Ciklust tartalmazó folyamatok:
 - o Egy cikluson belül egyszeres mélységű elágazásokat tartalmazó folyamatok (LX) (3 fájl)

- o Egy cikluson belül kétszeres mélységű elágazásokat tartalmazó folyamatok (L2X) (1 fájl)
- o Egyszeres mélységű elágazásokba ágyazott ciklust tartalmazó folyamatok (XL) (1 fájl)
- o Egy cikluson belül egyszeres mélységű elágazásokba ágyazott ciklust tartalmazó folyamatok (LXL) (1 fájl)
- o Egy cikluson belül egyszeres mélységű elágazásokba ágyazott két-szintű ciklusbeágyazást tartalmazó folyamatok (LXLL) (1 fájl)
- o Kétszintű ciklusbeágyazást tartalmazó folyamatok (LL) (1 fájl)
- o Kétszintű ciklusbeágyazáson belül egyszeres mélységű elágazást tartalmazó folyamatok (2LX) (1 fájl)
- o Négy szintű ciklusbeágyazáson belül egyszeres mélységű elágazást tartalmazó folyamatok (LLLLX) (1 fájl)

Random folyamatok:

- o Azonos eseményszámú, ismétlődő tevékenységeket nem tartalmazó folyamatok (FRNOREP)
- o Azonos eseményszámú, ismétlődő tevékenységeket tartalmazó folyamatok
- o Eltérő eseményszámú, ismétlődő tevékenységeket nem tartalmazó folyamatok
- o Eltérő eseményszámú, ismétlődő tevékenységeket tartalmazó folyamatok (IR)

A tesztadathalmazok statisztikai adatait az alábbi táblázatban ismertetjük:

Adathalmaz neve	Folyamatpéldányok száma (sorok száma)	Összes események száma	Eseménytípusok száma
S_X_01	1000	5000	7
S_X_02	1000	7000	10
S_X_04	1000	11000	16
S_XX_01	1000	3932	7
S_XX_10	1000	4534	6
S_XXX_01	1000	4026	9
S_XXX_10	1000	3911	9
S_LX_01	1000	6576	4
S_LX_02	1000	8378	4
S_LX_05	1000	10922	8

Adathalmaz neve	Folyamatpéldányok száma (sorok száma)	Összes események száma	Eseménytípusok száma
S_I2X_01	1000	24725	9
S_XL_01	1000	4834	5
S_LXL_01	1000	23879	8
S_LXLL_02	1000	50798	9
S_LL_02	1000	38167	5
S_2LX_02	1000	25469	10
S_LLLLX_01	1000	237032	12
S_FRNOREP	100	500	14
S_IR	5	54	5

A kísérlethez a saját, szintetikus generálású adatsorok mellett szabvány, versenyekben is használt adatsorokat vettünk alapul. A Process Discovery Contest (PDC) célja az új felfedezési technikák fejlesztésének és a meglévő technikák teljesítményének előmozdítása. Arra is igyekszik ösztönözni a közösséget, hogy vegye fontolóra olyan új típusú eseményadatokat, amelyeket még nem kellően feltártak.

A bevont minták listája:

- pdc 2016 1.xes (Process Discovery Contest 2016)
- pdc 2016 9.xes (Process Discovery Contest 2016)
- pdc 2017 5.xes (Process Discovery Contest 2017)
- pdc 2019 2.xes (Process Discovery Contest 2019)

A XES-formátum esetén egy XML-állományban adják meg a napló tartalmát, a nyomokban megjelenő eseményeket. A XES-formátum mellett a versenyen megadják a globális paramétereket is, amelyeket egy külön konfigurációs állomány tartalmaz.

```
<lifecycle_model>standard</lifecycle_model>
<number_of_traces>150370</number_of_traces>
<number_of_events>561470</number_of_events>
<events_per_trace>3.734</events_per_trace>
<min_events_per_trace>2</min_events_per_trace>
<max_events_per_trace>20</max_events_per_trace>
```

Az XES-állomány elején is az eseménynapló fontosabb paramétereit tárolódnak:

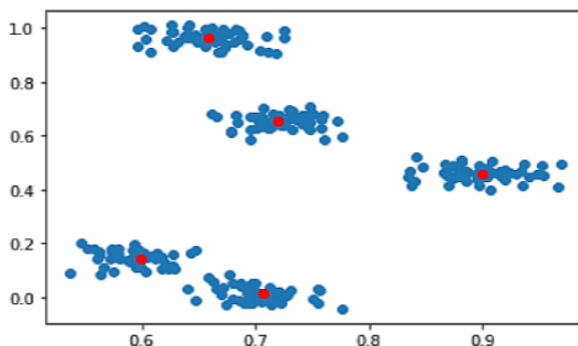
```
<int key="meta_org:different_resources_min" value="2"/>
<string key="meta_3TU:language" value="eng"/>
<int key="meta_concept:named_events_total" value="561470">
  <int key="Payment" value="77601"/>
  <int key="Insert Date Appeal to Prefecture" value="4188"/>
  <int key="Receive Result Appeal from Prefecture" value="999"/>
  <int key="Send Appeal to Prefecture" value="4141"/>
  <int key="Notify Result Appeal to Offender" value="896"/>
  <int key="Appeal to Judge" value="555"/>
  <int key="Add penalty" value="79860"/>
  <int key="Insert Fine Notification" value="79860"/>
  <int key="Send for Credit Collection" value="59013"/>
  <int key="Create Fine" value="150370"/>
  <int key="Send Fine" value="103987"/>
</int>
```

Az állomány törzsében a naplózott események jelennek meg:

```
<trace>
  <string key="concept:name" value="A25922"/>
  <event>
    <float key="amount" value="22.0"/>
    <string key="org:resource" value="561"/>
    <string key="dismissal" value="NIL"/>
    <string key="concept:name" value="Create Fine"/>
    <string key="vehicleClass" value="A"/>
    <float key="totalPaymentAmount" value="0.0"/>
    <string key="lifecycle:transition" value="complete"/>
    <date key="time:timestamp" value="2008-02-13T00:00:00+01:00"/>
    <int key="article" value="7"/>
    <int key="points" value="0"/>
  </event>
  <event>
    <string key="concept:name" value="Send Fine"/>
    <string key="lifecycle:transition" value="complete"/>
    <float key="expense" value="13.0"/>
    <date key="time:timestamp" value="2008-04-17T00:00:00+02:00"/>
  </event>
```


2.3. Kiértékelések eredményeinek bemutatása

Az első munkafázisban a beérkező nyomok (trace) szeparálására kidolgozott módszert teszteltük. A kidolgozott módszer a klaszterezés elvén működve válogatja szét a heterogén, eltérő folyamatokhoz tartozó nyomokat.



A klaszterezés során az aktuális napló nyomai mint egy-egy pont jelennek meg a sokdimenziós térben. A kidolgozott módszer ezen pontokat klaszterekbe szervezi. Ezt követően az egy klaszterekhez külön-külön állítjuk elő a hozzá tartozó sémagráfot. Így egyetlen közös (és valószínűleg kevésbé átlátható) sémagráf helyett több különböző sémagráf jön létre. Ezáltal a kapott sémagráfok:

- egyszerűbbek
- jobban átláthatóak
- pontosabbak
- jobban hasznosíthatóak

A kísérletek során teszteltük a kapott nyomlista felbontás jóságát. Ehhez összevetettük a javasolt módszer (sr-módszer) és a hagyományos módszerek eredményeit, hasonlóságát. A hasonlóságot jelző korrelációs táblázat jól mutatja, hogy a javasolt módszer áll legközelebb az emberi sémaalkotási folyamathoz.

method	items	value
similarity of predictions with highest weight	s_r -based and Silhouette	0.6
similarity of predictions with highest weight	s_r and human	1.0
similarity of predictions with highest weight	Silhouette and human	0.6
similarity of all predictions	s_r -based and Silhouette	0.57
similarity of all predictions	s_r and human	0.81
similarity of all predictions	Silhouette and human	0.61

A sémagráf-generáló algoritmus működésének szemléltetésére vegyük az alábbi példát. A napló az alábbi nyomokat tartalmazza:

EventI(1,1,'A',1)
 EventI(2,2,'A',3)
 EventI(3,3,'A',5)
 EventI(4,4,'B',10)
 EventI(5,5,'C',11)
 EventI(6,6,'B',15)
 EventI(7,7,'C',18)
 EventI(8,8,'D',21)
 EventI(9,9,'D',31)

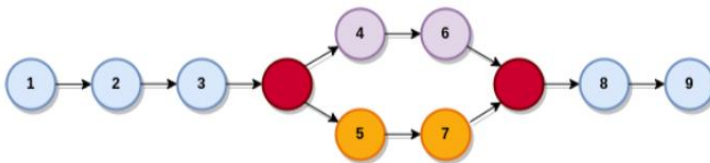
A kapcsolódó műveletkészlet a következő bejegyzéseket tartalmazza:

ActionI('1', {}, {'o1'})
 ActionI('2', {'o1'}, {'o1'})
 ActionI('3', {'o1'}, {'o2'})
 ActionI('4', {'o2'}, {'o3'})
 ActionI('5', {'o2'}, {'o4'})
 ActionI('6', {'o3'}, {'o5'})
 ActionI('7', {'o4'}, {'o6'})
 ActionI('8', {'o5', 'o6'}, {'o7'})
 ActionI('9', {'o7'}, {'o8'})

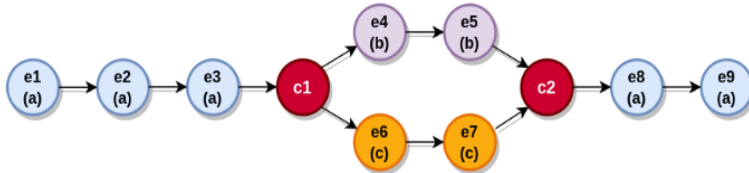
A leírásban az első argumentum a műveletazonosító, a második a bejövő termékek halmaza, a harmadik paraméter pedig a kimenő termékek halmazát jelöli. A szinkronizációs csomópont-felderítés algoritmusának kimenete két csomópontot generál:

{3} {4, 5} : 5 10
 {6, 7} {8} : 18 21

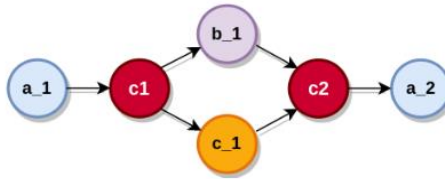
Az első szinkronizálási csomópont a 3. bemeneti eseményhez, valamint a 4. és 5. kimeneti eseményekhez kapcsolódik. Az időablaka az 5 és 10 időpontok közötti intervallum. A második szinkronizálási csomópontnál két bejövő esemény (6, 7) és egy kimenő esemény (8) található. Az ide vonatkozó időablak (18, 21).



A feldolgozás későbbi fázisában a kibővített sémából előállítunk egy felsőréteg-sémát, amelyben az egyes dolgozók saját, lokális sémái nem lesznek részletezve, azok egy-egy egységként jelennek meg. Erre egy jó példa az alábbi átalakítás. A részletes sémagráf:



illetve a felső réteg sémája:



Az elkészített modell hatékonyságának ellenőrzésére összehasonlító tesztek végeztünk el. A kísérletben az alábbi módszerek kerültek elemzésre:

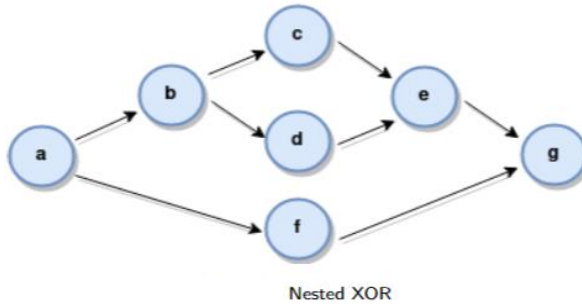
- MLP : alap MLP
- LSTM : alap LSTM
- BE-MLP : MLP with union-based reduction
- BE-LSTM : LSTM with union-based reduction
- HN-MLP : MLP with NN-based reduction

A tesztek fő célja az egyes módszerek előrejelzési pontosságának a mérése. A pontosságot az accuracy mérőszámmal vizsgáltuk. Az eredményeket az alábbi táblázat foglalja össze:

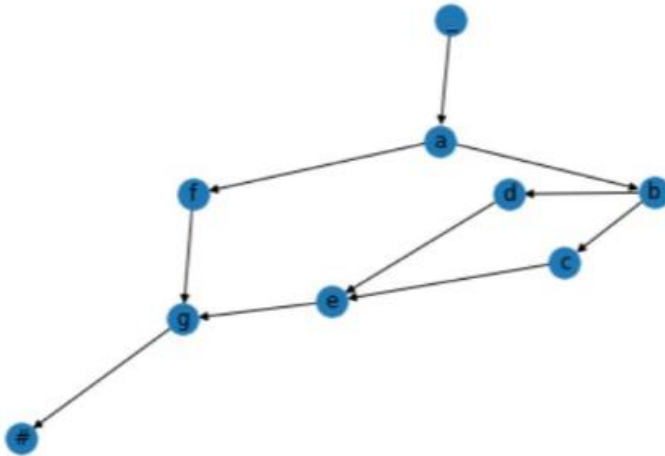
Dataset	LSTM	MLP	NH-MLP	BE-MLP	BE-LSTM
pd_c_2016_1.xes	63.5	63.4	63.4	63.9	64.1
pd_c_2016_9.xes	82.0	82.5	81.2	81.8	82.6
pd_c_2017_5.xes	62.4	62.8	63.7	64.8	64.2
pd_c_2019_2.xes	64.6	65.2	63.9	66.3	65.6
load_random	58.2	57.5	56.6	58.7	58.0

Az elért eredmények szemléltetésére bemutatunk néhány esetet a kapott és elvárt sémák összevezetésére.

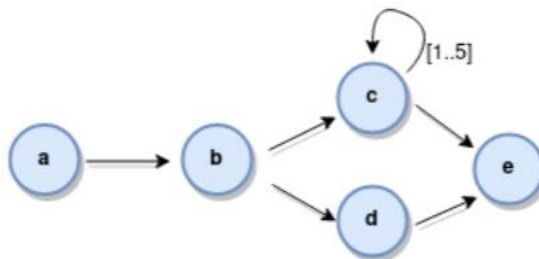
Elvárt séma:



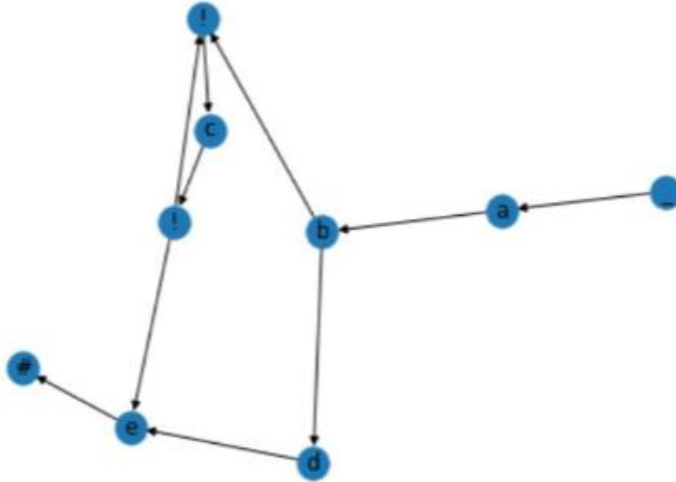
Kiszámított séma.



Elvárt séma:

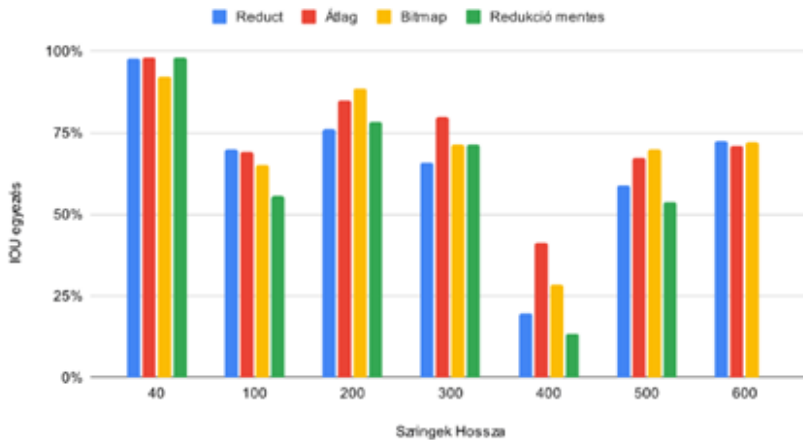


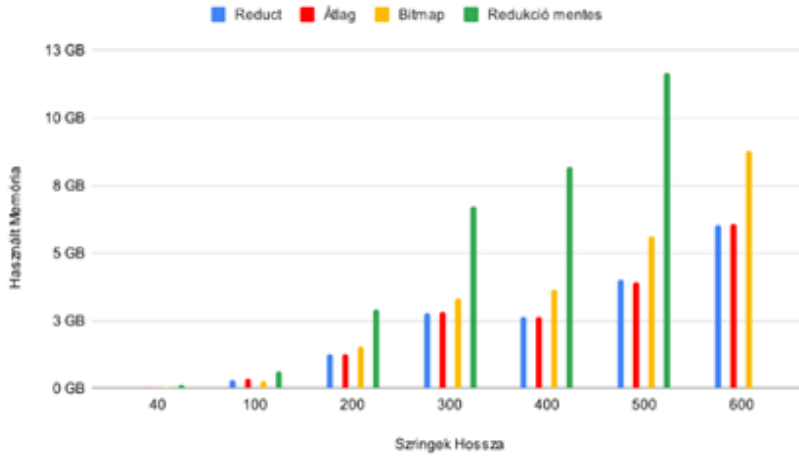
Kiszámított séma:



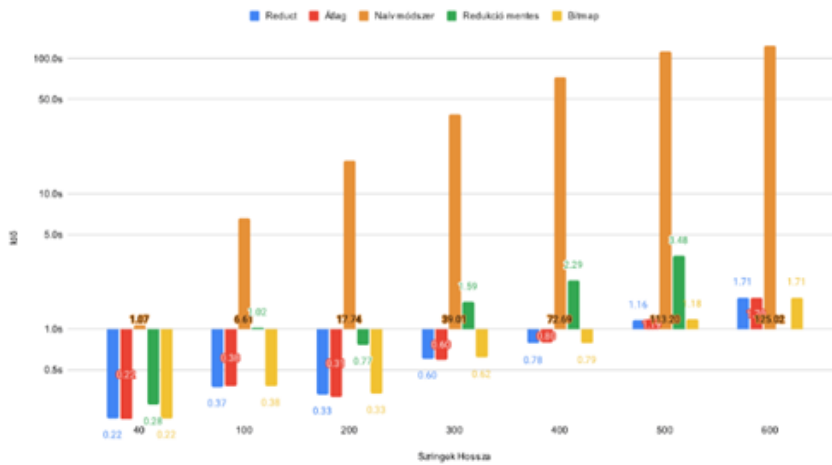
2.4. Eredményeket szemléltető képernyőképek, adatsorok

A következő ábrák a ciklusdetektáló algoritmus hatékonysági benchmark-tesztjeinek az eredményeit szemléltetik.





Megoldás idő



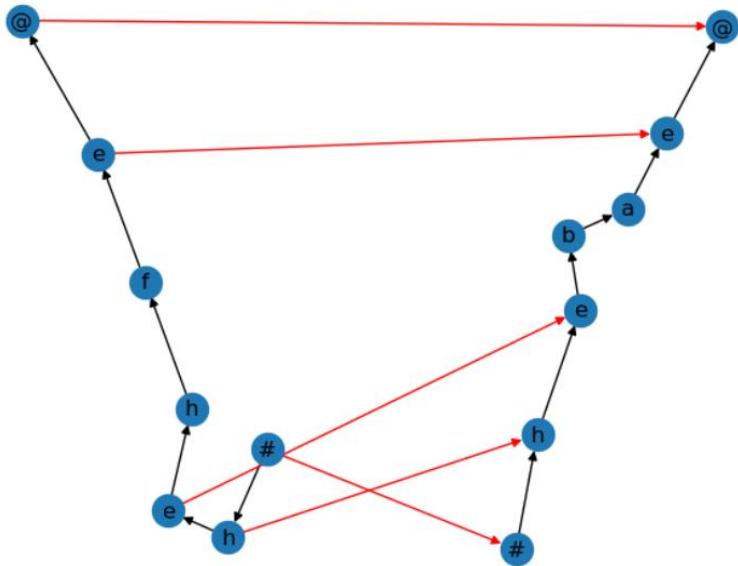
A sémagráfot előállító neurális háló alapú megoldás hatékonyságát az alábbi tesztesetek szemléltetik.

A bemenő eseménysorok:

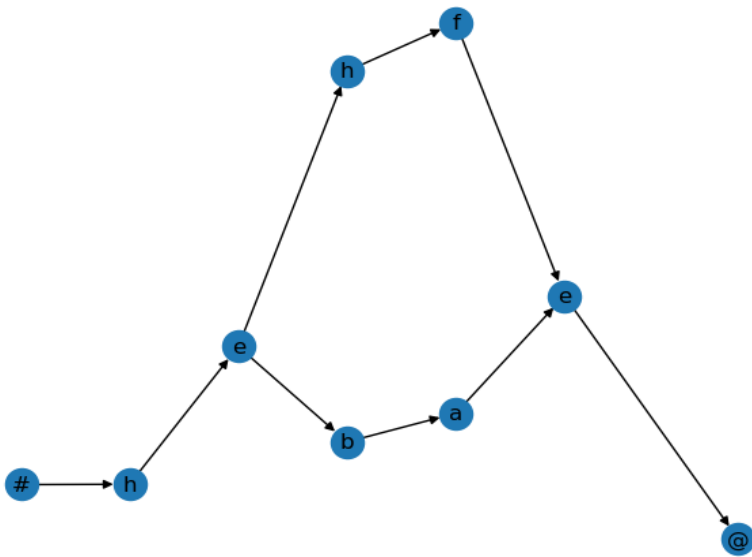
S2 = ['#', 'h', 'e', 'b', 'a', 'e', '@']

S3 = ['#', 'h', 'e', 'h', 'f', 'e', '@']

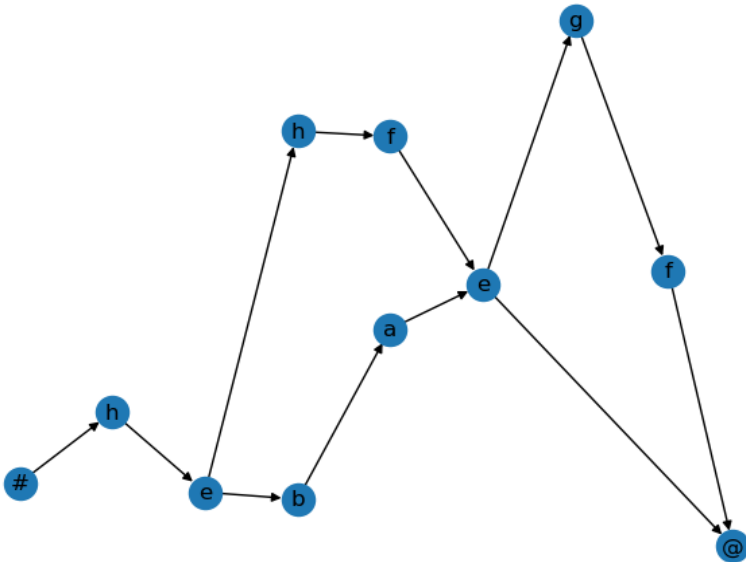
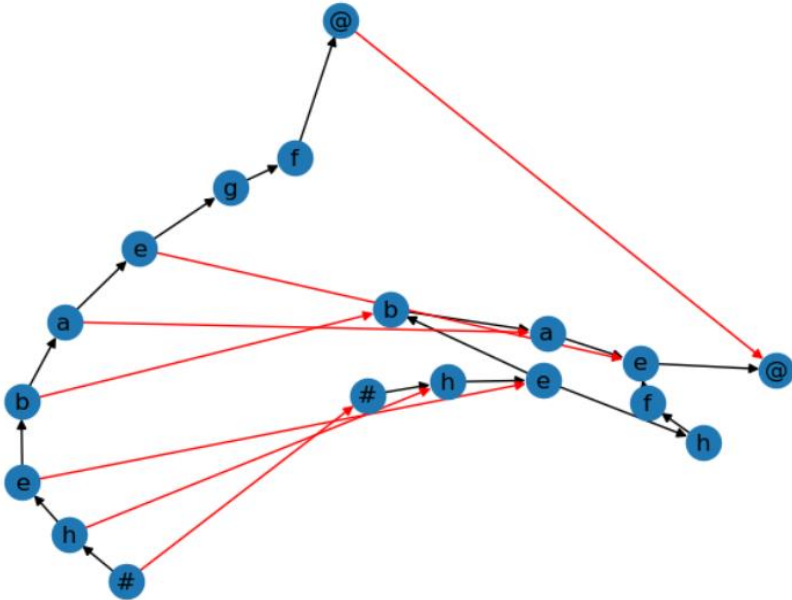
Két eseménysor illesztésének eredménye:



Az összevonás eredménye:



Egy további eseménysor hozzáadása révén a séma tovább növekedik



WEBES GRÁFSZERKESZTŐ ÉS MINTAGENERÁLÓ ALKALMAZÁS TERVEZÉSE ÉS FEJLESZTÉSE

MILEFF PÉTER

A kutatómunka ezen részfeladata egy nagyon érdekes területtel, az eseményleíró gráfok szerkesztésével és azok szimulációjával, azaz a mintaadatok generálásával foglalkozott. Munkánk során megtervezésre került egy webes környezetben működő szoftver, amely segítségével egy tetszőleges böngészővel elkészíthetjük a szükséges gráfot. A szoftver a gráf alapján képes adathalmazt generálni. Az elkészült szoftver ebben a formában tehát képes az alapvető feladatokat elvégezni, működőképes. Azonban számos területen további potenciális fejlesztési lehetőségek merülnek fel, melyek megvalósításával egy még hatékonyabb tesztelő környezet lenne kiépíthető.

1. Elvégzett kísérletek/modellezések bemutatása

A mai modern vállalatok legtöbb esetben már komplex informatikai rendszereket használnak az ügyviteli folyamataik támogatására. Idővel a vállalat fejlődése a folyamatok komplexitásának növekedését eredményezi. Csökken az átláthatóság, az érthetőség és idővel sokszor észrevehetetlen szűk keresztmetszetek alakulhatnak ki. Ilyen komplex rendszerek irányítása egyre több problémát vet fel mind biztonságtechnikailag, mint pedig a folyamatok követhetősége szempontjából. Az utóbbi idők eredményeképpen létrejött felhasználói aktivitásfigyelés (User Activity Monitoring – UAM) és az RPA-megoldások ezekre a problémákra próbálnak megoldást találni. Az ERPA-projekt által megfogalmazott egyik fontos cél, hogy a rendelkezésre álló, valamilyen információs rendszerből kinyert aktivitásnaplók alapján bizonyos események előre megjósolhatók legyenek valamilyen hatékonyan konfigurálható eljárás, mesterséges neurális hálózatot alkalmazó modell segítségével. A kutatás és a megoldás egyik kulcskérdése a megfelelő adathalmaz, hiszen ez teszi lehetővé a ráépülő folyamatok, a tanulóalgoritmusok tervezését, a megfelelő betanulást.

1.1. Feladat célkitűzése, motiváció, elvégzendő szakmai feladat bemutatása

A projektben végzett kutatómunkánk során sajnos kijelenthető, hogy nem könnyű megfelelő minőségű adathoz jutni. A vállalatok általában nem tudnak adatokat közzétenni, mert folyamataik szenzitív adatokat tartalmaznak. Azonban ahhoz, hogy egy mintarendszerben a predikciót végző neurális hálózatot tanítani lehessen, hogy egyáltalán kísérleteket/méréseket lehessen végezni a segítségével, mindenféleképpen szükség van valamilyen adathalmazra. Mindazonáltal az adathalmaznak tetemes mennyiségű és minőségű mintát kell tartalmaznia ahhoz, hogy a háló hatékonyan tanítható legyen.

A projekt keretében ezért magától értetődően felmerült az adatok generálásának kérdésköre. Bár a generált adatok általában sosem érik el egy valós adathalmaz minőségét, de ettől függetlenül mégis a legjobb módja annak, hogy különböző algoritmikus tervezési alternatívákat, különböző neurális hálózati modelleket lehessen kidolgozni és tesztelni. Mivel egy ilyen adathalmaz volumene nagy kell, hogy legyen sok-sok tranzakcióval, így a kézzel készítés nem megvalósítható. Mindenféleképpen adatok generálására van szükség.

Az eddigi vonalon továbbhaladva, a projekt keretein belül e témakörhöz kapcsolódó egyik fontos célunk volt, hogy egy olyan rendszert hozzunk létre, amely segítségével képesek legyünk a valósághoz közeli adathalmazok generálására.

A mintarendszer készítésének fontosabb elvárásai:

- **Online működés, hatékony infrastruktúra:** egy mai modern keretrendszer esetében elvárás, hogy az online is tudjon működni. Jelen esetben az online működés azt jelenti, hogy a felhasználó egy böngésző segítségével képes legyen az események leírására, majd azok „beküldésére” a feldolgozó motorba. Erdemes valamilyen előremutató infrastruktúrában gondolkodni, amely a későbbiekben könnyen továbbfejleszhető.
- **Eseményleírás:** egy további fontos elvárás a rendszertől, hogy támogassa az eseményfolyam sémaleírását. A felhasználónak valamilyen formában képesnek kell lennie arra, hogy definiálja és megalkossa azt a folyamatot, amelyből majd adatokat lehet generálni. A folyamatleírás valamilyen online megoldással, módon kell, hogy történjen.
- **Vizualizáció és validáció:** mivel komplex eseménygráfokról beszélünk, célszerű valamilyen vizualizációs lehetőséget beépíteni a szoftverbe. Ezzel a folyamatok jobban áttekinthetőek, hatékonyabb eseményleírás válik valóra. Mindezek mellett szükség van olyan validációs megoldásokra, amelyek ellenőrzik a gráf megfelelőségét, azt, hogy a jelenlegi feldolgozó motor képes-e értelmezni és végrehajtani.
- **Feldolgozó motor:** a szoftver központi eleme, amelynek a feladata az online érkező eseményleíró fogadása, értelmezése és a szimuláció elvégzése. Lényegében minden más eddig felsorolt elvárás ennek a modulnak a munkáját segíti. Megoldható lenne a modul kézi paramétereztetősége is, de ez olyan mértékben rontaná a használhatóságot, hogy csak nagyon körülményesen válna lehetővé a predikációs algoritmus(ok) fejlesztése.
- **Adatexport:** ahhoz, hogy az elvégzett számítások, szimulációk eredménye kiértékelhető legyen, szükséges az adatok megfelelő formátumba való export-

tálása. Célszerű több formátum (pl.: CSV, XES) támogatása, valamint a végeredmény kliensoldalra való visszaadása, hogy a felhasználó letölthesse azt a saját gépére.

Természetesen mivel a projekt keretein belül nem volt lehetőség a teljes rendszer maradéktalan kidolgozására, így a legnagyobb hangsúlyt a feldolgozó motor és az adatok megfelelő exportálása kapta.

1.2. Feladat módszertani háttere, irodalom rövid értékelése

A tervezett szoftver egyik kulcseleme a folyamat leírása és megjelenítése. A felhasználó valamilyen interaktív módszer segítségével megadhat egy folyamatgráfot. Célszerű két szakaszra bontással megközelíteni a problémát. Első szakaszban a felhasználó külön létre tudja hozni a szükséges csomópontokat. Ezekhez társítani tudja a megfelelő adatokat, tulajdonságokat. Majd a csomópontok közötti kapcsolatokat egy leíró nyelv segítségével elvégzi. Végül az elkészült gráf vizualizálva megjelenik a képernyőn, majd igény esetén, amennyiben végleges, elküldhető a szerveroldalnak feldolgozásra.

Az alábbiakban több megvalósítási lehetőséget mutatunk be:

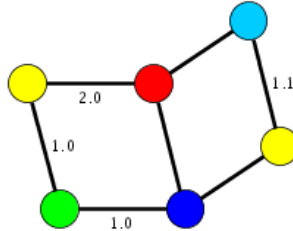
GraphML (Graph Markup Language)

A GraphML egy átfogó és könnyen használható fájlformátum a gráfok leírásához. Egy nyelvi magból áll, amely leírja a gráf szerkezeti tulajdonságait, és egy rugalmas kiterjesztési mechanizmusból az alkalmazásspecifikus adatok hozzáadásához. Fő jellemzői közé tartozik az alábbiak támogatása:

- irányított, irányítatlan és vegyes gráfok
- hipergráfok,
- hierarchikus gráfok,
- grafikus ábrázolások,
- hivatkozások külső adatokra,
- alkalmazásspecifikus attribútumadatok, és
- pehelysúlyú feldolgozók

Más gráfformátumoktól eltérően GraphML nem használ egyéni szintaxist. Ehelyett XML-en alapul, és ezért ideális „közös nevezőként” mindenféle szolgáltatáshoz, amely gráfokat készít, archivál vagy dolgoz fel.

Mintaként nézzünk egy egyszerű gráfot:



Ennek GraphML leírása pedig a következő:

```

<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
    http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="d0" for="node" attr.name="color" attr.type="string">
    <default>yellow</default>
  </key>
  <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
  <graph id="G" edgedefault="undirected">
    <node id="n0">
      <data key="d0">green</data>
    </node>
    <node id="n1"/>
    <node id="n2">
      <data key="d0">blue</data>
    </node>
    <node id="n3">
      <data key="d0">red</data>
    </node>
    <node id="n4"/>
    <node id="n5">
      <data key="d0">turquoise</data>
    </node>
    <edge id="e0" source="n0" target="n2">
      <data key="d1">1.0</data>
    </edge>
    <edge id="e1" source="n0" target="n1">

```

```
<data key="d1">1.0</data>
</edge>
<edge id="e2" source="n1" target="n3">
  <data key="d1">2.0</data>
</edge>
<edge id="e3" source="n3" target="n2"/>
<edge id="e4" source="n2" target="n4"/>
<edge id="e5" source="n3" target="n5"/>
<edge id="e6" source="n5" target="n4">
  <data key="d1">1.1</data>
</edge>
</graph>
</graphml>
```

JSON Graph format

Ez a JSON Graph Format az alapvető gráfszerkezetek kényelmesen használható formátumban történő rögzítésére összpontosít. Lehetővé teszi metaadat-objektumok használatát a gráfban, csomópontokban és élekben, amelyek felhasználhatók bármely más gráfadathoz, amelyet a gráf adatfájlaiban kell kezelni (pl.: gráf elrendezése, stílusa, algoritmus eredményei stb.). Az elmúlt néhány évben számos erőfeszítés történt a JSON Graph specifikációk létrehozására. A JSON Graph specifikációt a Githubon tették elérhetővé. A JSON-séma elérhető itt, valamint alapvető eszközöket biztosítanak hozzá, hogy szabványossá tegyék.

A JSON Graph a JSON-sémát használja a megfelelően formázott JSON-fájlok specifikálásához és ellenőrzéséhez. A JSON Graph fájl mindaddig nem érvényes, amíg át nem megy a JSON-séma specifikációval történő ellenőrzésen. A fő specifikációt a lehető legtömörebbre készítették, hogy az alkalmazások legszélesebb körében használható legyen. AI-specifikációk használhatók, amelyek érvényesítik a fő specifikációt, valamint a JSON Graph adatfájl alkalmazásspecifikus igényeihez tartozó részspecifikációt.

A következő példa egy JSON alapú mintagráfot mutat be:

```
{
  "graph": {
    "id": "1",
    "type": "weighted network",
    "directed": false,
    "label": "None",
    "nodes": { "a": {}, "b": {}, "c": {}, "d": {}, "e": {}, "x": {} },
```

```

"hyperedges": [
  { "nodes": ["a", "b", "x"], "metadata": { "weight": 17 } },
  { "nodes": ["a", "b"], "metadata": { "weight": 123 } },
  { "nodes": ["c", "d"], "metadata": { "weight": 45 } },
  { "nodes": ["d", "e"], "metadata": { "weight": 46 } }
]
}
}

```

DOT leíró nyelv

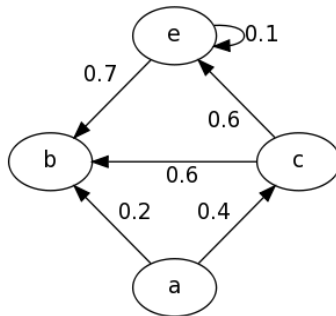
A DOT egy szöveges gráfleíró nyelv. Gráfoknak olyan egyszerű leírását adja, amely emberek és számítógépek által egyaránt olvasható. A DOT-nyelvű fájlok általában .gv (vagy .dot) kiterjesztésűek. Mivel a .dot kiterjesztést a Microsoft Office is használja, szerencsésebb a .gv kiterjesztés választása. A DOT-fájlokat számos programmal feldolgozhatjuk. Egy részük – például dot, neato, twopi, circo, fdp és sfdp – a beolvasott DOT-fájlt grafikus formában jeleníti meg. Mások – például gvpr, gc, accyclic, ccomps, scemap és tred – a beolvasott DOT-fájlon számításokat végeznek. Megint mások – például GVedit, KGraphEditor, lefty, dotty és grappa – interaktív felhasználói felületet biztosítanak. A felsorolt programok legtöbbje a Graphviz szoftvercsomag része, vagy azt használja a háttérben.

A DOT-formátum egy emberközelí formátum, amely nagyon „szimpatikus” jelölésrendszerrel rendelkezik. Leírását megtalálhatjuk az alábbi oldalon:

<https://graphviz.org/doc/info/lang.html>

A formátum egy egyszerű szöveges állományt használ, gyakorlatilag akár kézzel is könnyen szerkeszthető. Támogatja a legfontosabb sémákat, úgy mint az irányított gráfok, részgráfok, klaszterek, attribútumok stb.

Példaként nézzük meg az alábbi gráfot:



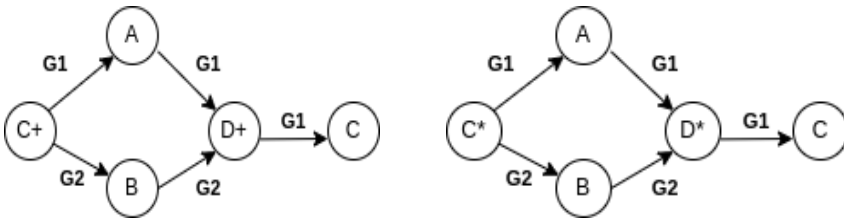
A gráf megfogalmazása DOT-nyelven az alábbi:

```
digraph {
  a -> b[label="0.2",weight="0.2"];
  a -> c[label="0.4",weight="0.4"];
  c -> b[label="0.6",weight="0.6"];
  c -> e[label="0.6",weight="0.6"];
  e -> e[label="0.1",weight="0.1"];
  e -> b[label="0.7",weight="0.7"];
}
```

A DOT-formátum meglehetősen népszerű a gyakorlatban. Több szoftver támogatja megjelenítésüket, valamint a programozási nyelvek is előszeretettel támogatják valamilyen csomag segítségével.

A projekt szempontjából első körben kiváló választásnak tűnik a könnyű érthetősége és feldolgozhatósága miatt. Egyedül a párhuzamos AND csomópontok okozhatnak fejtörést. Egy lehetséges megoldás lehet az alábbi elképzelés.

A következő ábrák két különböző gráfműködési típust ábrázolnak:



Míg az első esetben egy szekvenciális, úgynevezett „vagy” típusú hálót látunk, úgy második esetben egy párhuzamos „és” típusút. Ahhoz, hogy ezt a két típust meg tudjuk különböztetni, a jelölésrendszerbe bevezettünk úgynevezett speciális csomópontokat. A rajzon ezeket a C+, D+ és a C* és D* jelölések jelentik. Ezeket vezérlő csomópontoknak hívhatjuk. A meghozott szabály a gráf ábrázolásánál ezek alapján a következő: bármely olyan csomópont, amely mögött „+” jel szerepel, az egy „vagy” elágazás, vagy „vagy” szinkronizációs csomópont lesz. Ugyanígy, amennyiben a csomópont mögött „*” jelet teszünk, úgy pedig egy „és” csomópont és egy „és” szinkronizációs csomópont lesz. Ezzel a két elemmel a fent vázolt komplex események gráfként modellezhetők.

A szoftver egyik funkciója, hogy a felhasználó legyen képes gráfokat definiálni. Szintén egy fontos funkció a neurális hálózat felépítésének tanulmányozása és különböző alternatívák kidolgozása számára.

A gráf leírására az alábbi, DOT-szerű pszeudonyelvet lehet használni:
„VAGY” kapcsolat alapú gráf:

```
graph {
    C+ -> A (G1)
    C+ -> B (G2)
    A -> D+ (G1)
    B -> D+ (G2)
    D+ -> C (G1)
}
```

„ÉS”, párhuzamos kapcsolat alapú gráf:

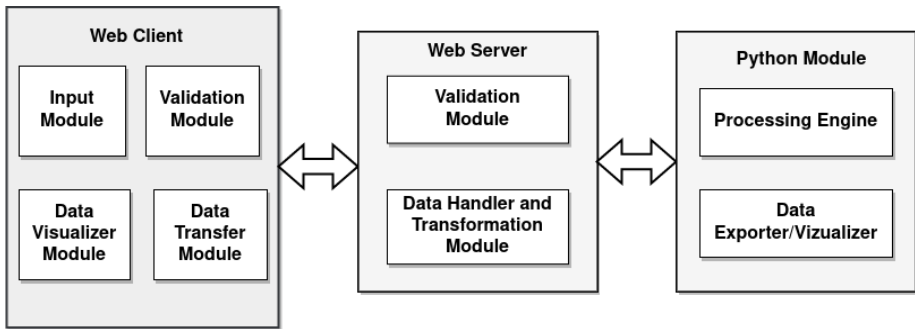
```
graph {
    C* -> A (G1)
    C* -> B (G2)
    A -> D* (G1)
    B -> D* (G2)
    D* -> C (G1)
}
```

A leírás egyértelműen definiálja melyik csomópontból melyik irányába történik az átmenet, valamint tartalmazza azt is, hogy melyik ágens az, aki a lépést végrehajtotta. G1, G2 jelenti azt, hogy ki hajtja végre az adott átmenetet. Tehát G1 például Pisti, G2 pedig Géza.

1.3. A kidolgozott kísérletek/modellezések módszertana

A megfelelő kutatási architektúra/infrastruktúra kialakítása kulcsfontosságú a projekt sikere, a kutatás „gördülékenysége” szempontjából. Általános célként megfogalmazható, hogy egy olyan kutatási szoftveres architektúrára kialakítására kell törekedni, amely képes hatékonyan kiszolgálni a projekt és a kutatás követelményeit.

Általános elvárás a célként kitűzött szoftverrel kapcsolatban, hogy webes környezetben működjön. Ennek megfelelő tipikus megvalósítás alapja általában a *kliens-szerver* architektúra. A projekt szempontjából ez a megoldás megfelel az elvártaknak. Ez alapján a rendszert alkotó modulok sematikus leírása a következő:



1. ábra. Mintageneráló rendszer architektúra

Az ábrán jól látszik, hogy három fő részből tevődik össze az architektúra: kliensoldal, szerveroldal és a processing engine.

A szoftver működése röviden a következő: A felhasználó megnyitja a weboldalt (kliensoldal), majd azon egy új eseménygráf-leírásba kezd. Ezt iteratív úton tudja megtenni. A gráflétrehozást elemi lépésekre bontjuk fel, a háttérben minden egyes kliensoldalon elvégzett gráfmódosítás a szerveroldalra küldi a szükséges adatokat (pl. Új csomópont létrehozása), ahol pedig a gráf memóriabeli (és szöveges leíró) modellje épül fel. Ez a modell fog folyamatosan módosulni a szerkesztés során. Végül, miután a felhasználó elkészült, elküldheti a gráfot a feldolgozó motornak (processing engine) eseményhalmaz-generálás céljából. A feldolgozó motor a felépített gráf modellt értelmezi, majd a paramétereknek megfelelően egy adott eseményhalmazt generál, melyet XES- és CSV-fájlokban is elment. Az elkészült eredményfájlok a mesterséges neurális hálózati modell bemeneti adathalmazaként közvetlenül használhatók.

1.3.1. Kliensoldal

A kliensoldalon jelenik meg az események interaktív megadása, amely vizualizálható és elküldhető a szervernek feldolgozásra. Technológia szempontjából javasolt valamilyen modern Javascript, Dart-alapú keretrendszer alkalmazása. Tipikusan ilyenek: Angular, ReactJS, Vue.js, Flutter. Segítségükkel a kliensoldal szükséges moduljai hatékonyan kialakíthatók.

Négy fontosabb modul emelhető ki:

Input modul: A beviteli modul lehetővé teszi a felhasználó számára egy adott folyamat elemi eseményeinek leírását a grafikus felhasználói felület segítségével.

Fontos elvárás, hogy valamilyen step-by-step úton lehessen elvégezni a folyamat elemeinek megadását, mert bizonyos események paraméterezése és egyéb adatainak megadása is elvárt. Két speciális eseménytípus is modellezésre kerül: az úgynevezett AND és OR típusú elágazások.

Fontos elvárás, hogy a szerveroldalra motorba beérkező adatok valamilyen jó áttekinthetőséget adó szöveges formátumban is elkészüljenek. Az input modul lényegében ezt az „állományt” készíti el különböző input boxok és egyéb elemek segítségével. Az elkészült szöveges leíró előnye, hogy állományként jól tárolható, később pedig újra feldolgozható. A későbbiekben több ilyen formátum is bemutatásra kerül.

Az eseményleíró gráf készítésének alapvető igényei:

- Új gráf létrehozása
- Meglévő gráf szerkesztése
- Elemek iteratív hozzáadása és elvétele
- Teljes gráf törlése
- Csomópontok attribútumainak menedzselése
- Alapvető hibakezelés

Validációs modul: a folyamat eseményeinek megadása egy gráfot fog eredményezni. Mivel célunk a komplexebb folyamatok modellezése is, így nagyobb gráfok esetében már nem elegendő az emberi szemmel történő validáció. Ebben a folyamatban segít ez a modul, hogy ne a szerveroldalon derüljenek ki olyan alapvető problémák, amely miatt az eseménygenerálás nem hajtható végre, vagy értelmetlen gráf alapján történik a generálás.

Alapvető ellenőrzések:

- Szintaktikai helyesség ellenőrzés
- Gráf kezdő- és végállapotának megléte
- Körkörös utak ellenőrzése
- Megadott paraméterek valósága

Adatvizualizációs modul: bármely hasonló jellegű szoftver esetében természetes igényként merül fel a vizualizáció. Az ember vizuális alkat, valamint a komplex folyamatok átlátásában/megértésében nagy szerepet játszik, ha vizuálisan is megtekinthetjük az elkészült eseményhalmazt.

A javasolt Javascript-alapú technológiák központi repositoryjában rendelkezésre állnak olyan csomagok (pl. d3), amelyek alkalmasak lehetnek ezeknek a

gráfoknak a megjelenítésére. Egyetlen probléma ott jelentkezhet, hogy olyan eseményeket is modellezhetünk, amelyek egymással párhuzamosan is végrehajthatódnak. Ezeket nevezzük AND elágazásoknak. Meg kell vizsgálni, hogy egyáltalán van-e arra mód a rendelkezésre álló csomagok esetében, amelyek képesek valamilyen customizált gráf létrehozására is.

Adatkezelő modul: Végül az adatkezelő modul feladata, hogy továbbítsa és fogadja az adatokat a szerver irányába. Kapcsolatot tart a szerverrel, szükség esetén olyan formátumra transzformálja a létrehozott gráfot, amely megfelel a Python feldolgozó motornak. Javasolt kommunikációs forma a REST API, JSON-alapú adatátvitel.

1.3.2. Az Angular keretrendszer

Az AngularJS egy Google által fejlesztett, nyílt forráskódú JavaScript keretrendszer dinamikus webes alkalmazásokhoz. Segítségével nagyban egyszerűsödik a webes alkalmazások frontend fejlesztése. Használatával a HTML eszköztára kibővül és az alkalmazások komponensei még egyértelműen elkülönülnek. Az Angular adatkapcsolásának, illetve függőséginjektálásának köszönhetően, rengeteg felesleges boilerplate kód elhagyható.

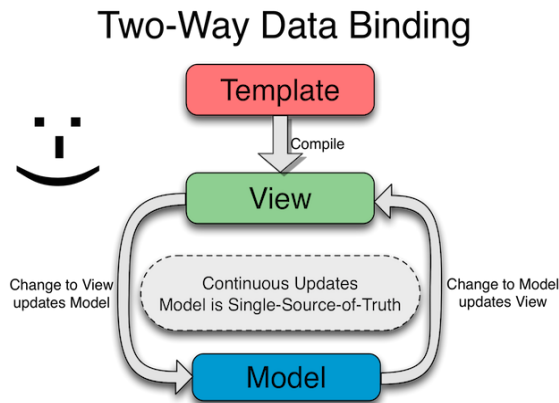
A keretrendszer főbb célkitűzései:

- Felület definiálására ideális egy deklaratív leírás (HTML), míg az imperatív programozás kiváló, hogy kifejezze az üzleti logikát.
- Válasszuk le a DOM (Dokumentum Objektum Modell) manipulációt az alkalmazás logikáról.
- A program tesztelése legalább olyan kritikus, mint annak írása.
- Az alkalmazás kliens, illetve szerveroldal teljes szétválasztása.

Az Angular keretrendszer egyik legnagyobb előnye az, hogy viszonylag kötött felépítéséből adódóan a kódolás meghatározott keretek között zajlik és ebből adódóan kevesebbet enged hibázni. Fő alkotóelemei a komponensek és a modulok. A komponensek lesznek azok az elemek, amik a végfelhasználó számára láthatóak és gyakran újra felhasználhatóak (például a fejléc és a lábléc rendszerint minden oldalon ugyanaz). A modulok pedig csoportosítják az összetartozó komponenseket és deklarálják, hogy mely komponenseket használhatnak fel más modulok. Mint minden nagyobb keretrendszerhez az Angularhoz is léteznek már kész kiegészítő modulok. Ezek segítségével egyszerűen tudjuk az alap keretrendszer eszköztárát bővíteni.

1.3.2.1. Kétirányú adatkapcsolat

A legtöbb template rendszer csak egyirányú adatkapcsolatot (data binding) tesz lehetővé, egyesítik a template és modell komponenseket egy nézetté. Az egyesítést követően a modellben történt változások nem tükröződnek automatikusan a nézetben. Ennél sokkal problémásabb, hogy a nézeten történő változások nem lesznek visszavezetve a modellbe. Az inkonzisztencia elkerülése érdekében a fejlesztőnek gondoskodnia kell a folyamatos szinkronizálásról. Ezzel szemben az Angular más megközelítést választott. Elsőként a template lefordul a böngészőben (nyers HTML kiegészítve a speciális Angular direktívákkal). A fordítás eredménye ellentétben a fenti esettel nem statikus lesz. Ennek köszönhetően bármilyen változás a nézetben azonnal tükröződik a modellben, illetve a modell változásai is továbbterjednek a nézetbe. Ennek köszönhetően az állapot egyedül a modellben tárolódik (single-source-of-truth), ami nagyban könnyíti a fejlesztő életét. Lényegében a nézet tekinthető a modell egy projekciójának. Ennek köszönhetően a controller teljesen izolált, így a tesztelhetőséget is elősegíti.



2. ábra. Kétirányú kötést megvalósító architektúra

1.3.3. Adatvizualizáció

A beviteli adatok, a megadott folyamatmodell grafikus megjelenítése kulcsfontosságú. A vizualizációt célszerű folyamatosan elérhetővé tenni, mutatni. Ennek oka, hogy a folyamat modellje általában iteratív úton készül, érdemes tehát a köztes, félkész modelleket is megjeleníteni. Így a készítő azonnal visszacsatolást kap, amellyel növelhető a készülő modell érthetősége, mert a szöveges alapokon leírt modellt nem feltétlenül könnyű átlátni. A világhálón számos függvénykönyvtár áll rendelkezésre, melyből a javasolt megoldás a D3 JS (<https://d3js.org/>).

A D3 egy Javascript nyelven írt függvénykönyvtár dokumentumok adatokon alapuló manipulálására. Nevét is innen kapta (Data-Driven Documents). A D3.js egy dinamikus, interaktív, online adatvizualizációs keretrendszer, amelyet számos webhelyen használnak. Elsősorban különböző típusú interaktív grafikonok létrehozására lett kifejlesztve. Különlegessége, hogy a legösszetettebb grafikonokat is viszonylag könnyen lehet elkészíteni vele. További erőssége a jelen célok kapcsán, hogy rendelkezik olyan kiterjesztéssel, amellyel a DOT-formátumok is megjeleníthetők (<https://github.com/magjac/d3-graphviz>).

1.3.4. Szerveroldal

A szerveroldal feladata, hogy egy köztes réteget lásson el a kliens és a feldolgozó motor között. A gráf tényleges modellje itt készül el. Fogadja a kliensoldalról érkező gráfépítő utasításokat: csomópont létrehozása, törlése, kapcsolatok létrehozása stb. Minden funkcióra külön végpont készül. Ezekkel a gráf modellt felépíthető.

A szerveroldalon épített gráf mellett az alábbi érvek szólnak: Mivel a szerveroldali modul, valamint a feldolgozó modul is mind Python nyelven készülnek, a kettő közötti „átjárás”, az adatok továbbítása egyikből a másikba egyszerűen megvalósítható. Szerveroldalon akár már rögtön Python adatstruktúrában (is) lehet a modellt tárolni, így az már közvetlenül átadható a feldolgozó motor számára.

A szerveroldali implementációt tipikusan valamilyen pehelysúlyú környezetként célszerű megvalósítani, hiszen kevésbé fog bonyolult feladatokat elvégezni. Végpontokat kínál mindkét oldalnak, illetve adatot transzformál és továbbít a szükséges irányba.

Két főbb területet célszerű megkülönböztetni:

- **Validációs modul:** természetesen a fogadott adatok minőségének ellenőrzését végzi el. Más a feladat a kliensoldalról kapott adatok esetében, és más a feldolgozó motortól kapott adatok esetében.
- **Adatátviteli és transzformációs modul:** szükség esetén a megadott formára alakítja a továbbított adatot. Amennyiben a két oldal mindegyike megfelelően előkészíti az adatokat, úgy csak minimális feladata van.

Fontos megjegyezni, hogy amennyiben sok adat generálása a cél, úgy célszerű megfontolni az aszinkron működés lehetőségét. Mivel a Python feldolgozó motor valószínűleg hosszabb ideig fog dolgozni, így a kliensoldal nem biztos, hogy tud olyan sokáig várakozni a szinkron HTTP-kérés miatt. A helyes implementáció mindenféleképpen az aszinkron megoldás, azonban ez az út valamivel bonyolul-

tabb is. Ekkor ugyanis a kliensoldalon szükség van egy olyan felületre, ahol megtekinthetők a már elkészült/generált fájlok. Ehhez viszont a szerveroldalon a fájlok és azok státuszának menedzselését kell megvalósítani. Nyilvántartást kell vezetni a generált log fájlokról és arról, hogy mi volt a kiinduló folyamatleíró gráf.

Az első implementáció során mindenféleképpen a szinkron HTTP-kérés megvalósítása a cél. Mivel egy harmadik modul, a feldolgozó is jelen van, célszerű a szerveroldali implementációt ugyanabban a környezetben elkészíteni, mint a feldolgozó motor. Jelen esetben ez a Python.

1.3.4.1. Gráfok tárolása

Bár a felhasználó a kliensoldalon végzi a szerkesztést, a háttérben a szerveroldalon áll össze a gráf. Ehhez célszerű valamilyen tároló mechanizmust biztosítani, amely jól használható és hatékony. Mivel jelen szoftver egy prototípus, ezért jelen fázisban még nem célszerű adatbázis kezelőt alkalmazni. Helyette valamilyen NoSQL-megoldás lehet hatékony. Számos elérhető NoSQL-szoftvert megvizsgálva az érvek a Redis programcsomag mellett szólnak.

A Redis egy nyílt forráskódú (BSD-licenccel rendelkező) szerveroldali alkalmazás, memórián belüli adatstruktúra tároló, amelyet adatbázisként, gyorsítótárként, üzenetközvetítőként és streamingmotorként használnak. A legjobb teljesítmény elérése érdekében a Redis a memórián belüli adatkészlettel dolgozik key-value párokban. A felhasználási esettől függően a Redis megőrizheti az adatokat úgy, hogy rendszeres időközönként kiírja az adatkészletet a lemezre, vagy minden parancsot hozzáfűz egy disk alapú naplóhoz. A perzisztencia kikapcsolható, ha csak funkciókban gazdag, hálózatba kapcsolt, memórián belüli gyorsítótárra van szükség. A Redis támogatja az aszinkron replikációt, gyors, nem blokkoló szinkronizálással és automatikus újra csatlakozással, részleges újra szinkronizálással a hálózati felosztásnál.

A Redis ANSI C nyelven íródott, és külső függőségek nélkül működik a legtöbb POSIX-rendszeren, például Linuxon, *BSD-n és Mac OS X-en. A Linux és az OS X az a két operációs rendszer, ahol a Redist a legtöbbször fejlesztik és tesztelik, és javasoljuk a Linux használatát a telepítéshez.

1.3.4.2. Python feldolgozó motor

A mesterségesintelligencia-számítások legnépszerűbb implementációs környezete jelenleg a Python. A projekt során alkalmazandó az elsődleges programozási nyelv és környezet szempontjából a választás így a Pythonra esett. A döntés

könnyen indokolható, hiszen a Python egy interpretált, általános célú programozási szkriptnyelv, több évtizedes fejlődési múltra tekint vissza. Mesterséges intelligencia, neurális hálózatok és egyéb matematikai számítások esetében előszeretettel alkalmazzák rugalmassága és hatékonysága miatt. A Python könnyű és gyors kipróbálhatóságon alapuló fejlesztést tesz lehetővé széles spektrumon mozgó moduljai segítségével. Gyakorlatilag szinte mindenféle jellegű problémára találunk olyan csomagot, amely nagymértékben segít a megoldásban. A neurális hálózatok esetében is kimondottan gazdag a kínálat. Ezért jelen projekt esetében nem kérdés, hogy a Python környezetben végezzük el a szükséges implementációkat.

A Python előnye azonban önmagában egy megkötés is. Mivel a feldolgozó motor implementációs nyelve kötött, így célszerű a szerveroldali kiszolgáló környezetet is ebben megvalósítani. Ez azonban nem szokványos, más nyelvek inkább alkalmasak erre a feladatra. Más implementációs nyelv választása azonban egy újabb nehézségi faktort illeszt a jelenleg folyamatba. Ekkor ugyanis az adott szerveroldali kiszolgálónak meg kell tudni valahogy hívni a Python feldolgozó motort, mindezt megfelelően paraméterezve, helyes adatokkal.

Általános elvárások a feldolgozó motorral szemben:

- **Input validáció:** bár a korábbi lépésekben is jelen volt a validáció, sajnos itt sem mehetünk el mellette. Valamilyen minimális validáció beépítése itt is elvárt és célszerű.
- **Parsolás, objektummodell-építés:** a szöveges formátumban elküldött gráf értelmezése és parsolása. A szöveges tartalom alapján a motor egy memóriabeli struktúrát fog felépíteni, amelyben már valós objektumok fognak szerepelni. Lényegében a szöveges gráf tartalmát objektummodellre képezi le.
- **Eseménygenerálás:** gyakorlatilag ez az a funkció, amelyre szükségünk van. A felépített objektum modell alapján a feldolgozó motor feladata, hogy a megadott paramétereket és attribútumokat figyelembe véve adatsorokat generáljon több száz, vagy akár több ezer méretekből.
- **Adatexport:** a generált adatok először mindenféleképpen a memóriában kell legyenek. Ezeket azonban valamilyen módon vissza kell juttatni a szerveroldalnak, majd pedig a kliens oldalnak. Ehhez az adatok fizikai mentésére van szükség. Az export során elvárt funkció, hogy legalább két formátumot támogasson. Ezek: CSV, XES.

A feldolgozó motor a szoftver magja. Amennyiben hibásan működik, úgy értelmetlenné válik a többi komponens is. Fontos szempont tehát a körültekintő implementáció és a robosztus viselkedés. Köztudott, hogy a Python megvalósí-

tások nem tartoznak a leggyorsabb futásidejű programokhoz, azonban jelen projekt keretében ez nem cél. Vélhetően nincs szükség akkora számítási teljesítményre, valamint fontosabb a Python által kínált rugalmas környezet a beépített számtalan funkcióval.

1.4. Docker alapú virtuális környezet

A projekt során a kutatási környezetet és a prototípus-alkalmazás fejlesztését célszerű valamilyen izolált környezetben elvégezni. Ez lehetővé fogja tenni a rugalmas konfigurációt és igény esetén a gyors migrációt is. Ma már több ilyen jellegű környezet-, úgynevezett konténertechnológia érhető el. Jelen fejlesztésben a Dockert tekintjük megfelelő választásnak.

A Docker egy olyan szoftverplatform, amelyet a konténerek koncepcióján alapuló alkalmazások telepítésének megkönnyítésére fejlesztettek ki. A platform felhasználható alkalmazások létrehozására, tesztelésére és terjesztésére, szabványos egységekben, úgynevezett konténerekben történő gyűjtésével.

A Docker konténerek elkülönített környezetek, ahol az alkalmazások más folyamatok beavatkozása nélkül futtathatók. Hasonlóan a virtuális gépek körülményéhez, az egyes konténerekhez specifikus számítási erőforrásokat allokálhatnak. A virtuális gépekkel ellentétben a Docker nem igényel hardveres emulációt, hanem minden gazdagép fizikai hardverét használja. A Docker platform lényegében becsomagolja az alkalmazásunkat egy gyengén izolált környezetbe, amit konténernek hívunk. Ezt a konténert könnyen el tudjuk szállítani a velünk dolgozó mérnököknek, vagy akár éles környezetbe is felküldhetjük annak a veszélye nélkül, hogy az alkalmazás környezete megváltozik és ezáltal nem funkcionál rendesen.

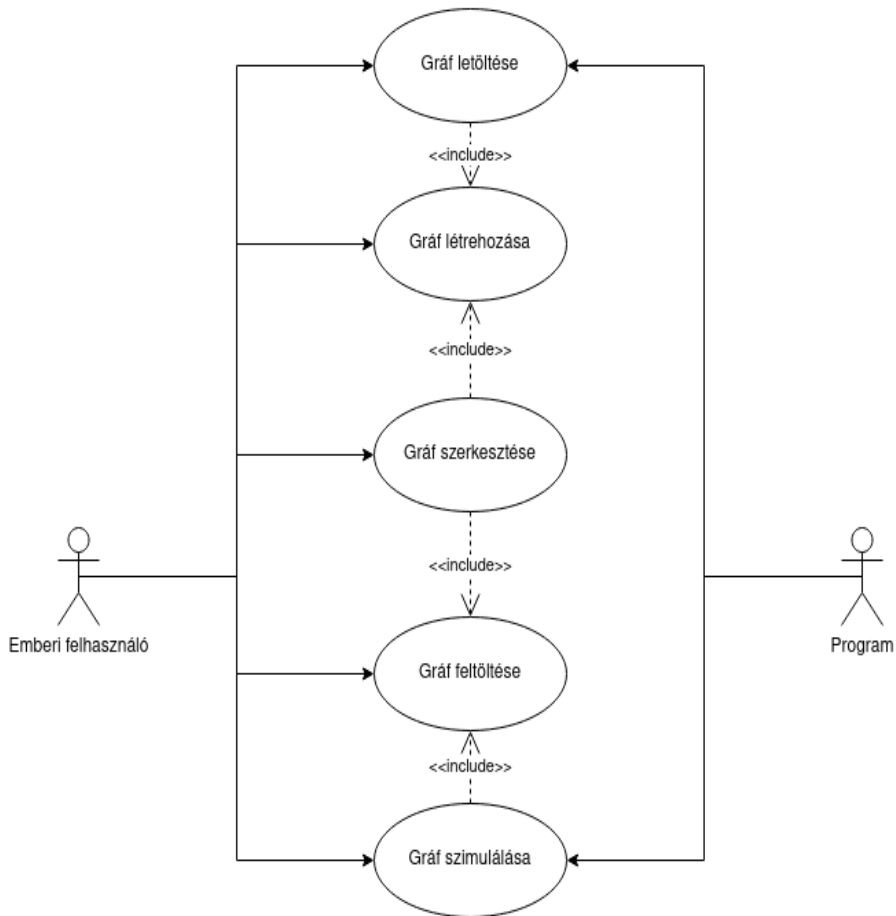
Az erőforrás-hatékonyság és a továbbfejlesztett rendszerteljesítmény előállítása érdekében a tárolást a virtualizáció természetes fejlődésének tekinthetjük. Sőt, mivel rendelkeznek az alkalmazások – beleértve a könyvtárakat, a rendszereszközöket és a kódokat is – helyes végrehajtásához szükséges eszközökkel, a konténerek egyszerűsítik az alkalmazások telepítését anélkül, hogy aggódnunk kellene a futási környezet konfigurációja miatt.

A Docker jelenleg az egyik legelterjedtebb container framework, amelynek implementációja erősen támaszkodik a Linux kernel nyújtotta szeparációs lehetőségekre. A Docker sikere három alappillérré épül:

- **Könnyű használhatóság:** A Docker fejlesztők és sysadminok számára egyaránt egy nagyon egyszerű lehetőséget biztosít hordozható alkalmazások létrehozására és menedzselésére. Segítségével bárki fejleszthet és becsomagolhat egy alkalmazást például a saját laptopján, és biztos lehet benne, hogy az egy teljesen másik környezetben, tipikusan egy cloud szerveren futtatva is ugyanúgy működni fog.

- **Sebesség:** a technológiából fakadóan a Docker containerek kicsik és gyorsak, hiszen tulajdonképpen csak egy, a kernelen futó sandbox környezetről van szó, amely kevés erőforrást emészt fel.
- **Modularitás és a skálázhatóság:** a Dockerrel nagyon egyszerűen „feldarabolhatjuk” az alkalmazásunkat különböző containerekbe, például egy containerben a MongoDB adatbázisunkat helyezzük el, egy másikban egy Redis szerver, és egy harmadik containerben pedig az ezekhez kapcsolódó Node.js app.

1.5. A módszert bemutató folyamatábrák, diagramok



3. ábra. Általános funkciók



4. ábra. Mintageneráló rendszer részletes Use-case diagram

2. Az elért eredmények bemutatása

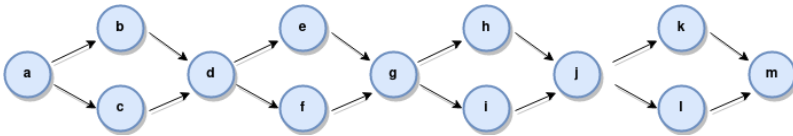
A kutatómunka során a tervezett alkalmazás prototípus-változata implementálásra került. Az alkalmazás backend és a frontendje is egy-egy Docker containerben érhető el. A kettő közötti kommunikáció egy docker network közvetítésével történik, HTTP kérés-válaszok formájában. A Docker lehetővé teszi, hogy könnyen adjunk a rendszerhez újabb elemeket, legyen az adatbázis, cache adatbázis, egyéb mikroszerviz.

A Docker használatának további előnye, hogy egy gépen, amin Docker van telepítve, a rendszer egy *docker compose up* parancssori utasítás kiadásával indítható.

2.1. Tesztadathalmazok ismertetése

Az alkalmazás teszteléséhez az alábbi adathalmazokat alkalmaztuk:

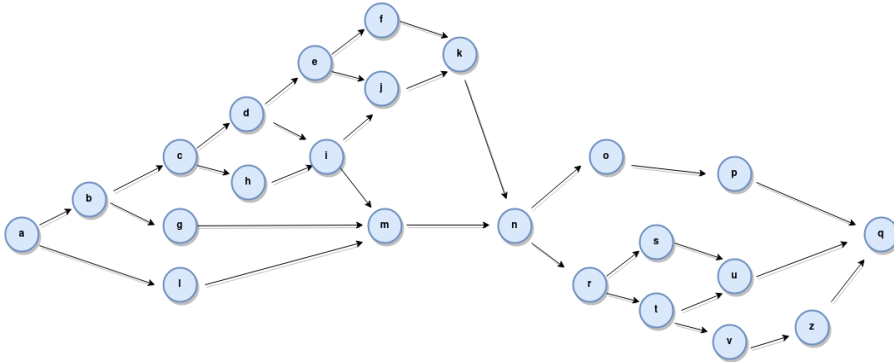
Négyszeres XOR gráf:



DOT-leírás:

```
digraph { a -> b; a -> c; b -> d; c -> d; d -> e; d -> f; e -> g; f -> g; g -> h; g -> i; h -> j; i -> j; j -> k; j -> l; k -> m; l -> m; }
```

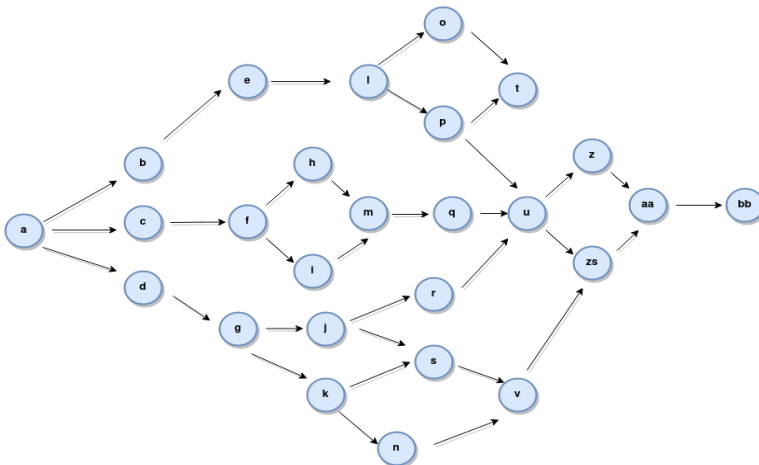
Komplex beágyazott gráf 1:



DOT-leírás:

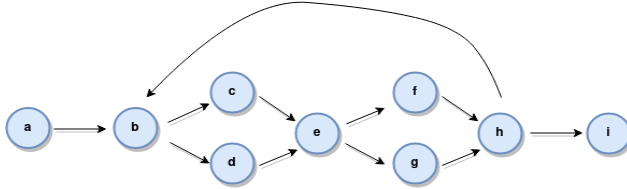
```
digraph { a -> b; a -> l; b -> c; b -> g; c -> d; c -> h; d -> e; d -> i; h -> i; e -> f; e -> j; i -> j; i -> m; f -> k; j -> k; k -> n; l -> m; g -> m; m -> n; n -> o; n -> r; o -> p; p -> q; r -> s; r -> t; s -> u; t -> u; t -> v; v -> z; z -> q; u -> q}
```

Komplex beágyazott gráf 2:



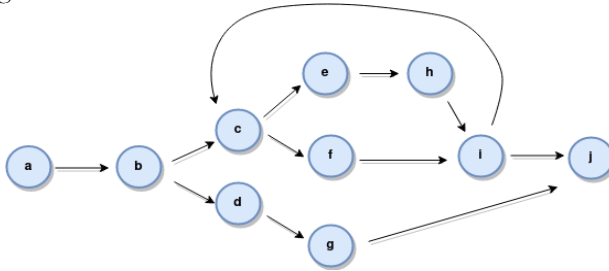
DOT-leírás:

```
digraph { a -> b; a -> c; a -> d; b -> e; c -> f; d -> g; e -> l; l -> o; l -> p; o -> t; p -> t; f -> h; f -> i; h -> m; i -> m; m -> q; q -> u; p -> u; g -> j; g -> k; j -> r; j -> s; r -> u; k -> s; k -> n; s -> v; v -> zs; u -> z; u -> zs; z -> aa; zs -> aa; aa -> bb;}
```

Loop példa gráf 1:**DOT-leírás:**

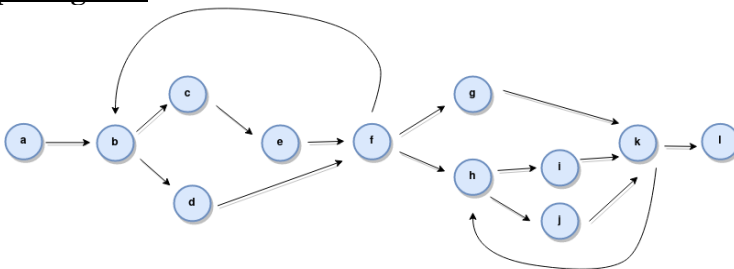
```

digraph { a -> b; b -> c; b -> d; b -> c; c -> e; d -> e; e -> f; e -> g; f -> h; g -> h; h -> b; h -> i; }
  
```

Loop példa gráf 2:**DOT-leírás:**

```

digraph { a -> b; b -> c; b -> d; c -> e; d -> g; g -> j; c -> f; e -> h; f -> i; h -> i; i -> c; i -> j; }
  
```

Loop példa gráf 3:**DOT-leírás:**

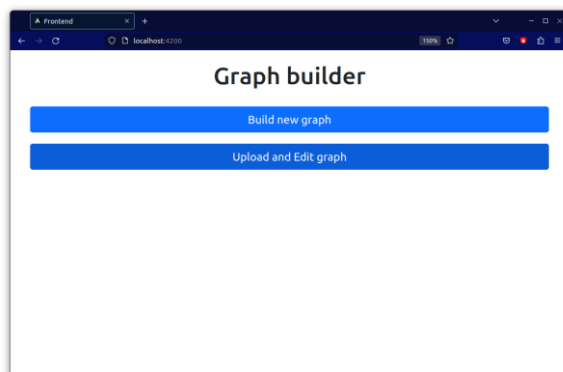
```

digraph { a -> b; b -> c; b -> d; c -> e; e -> f; d -> f; f -> b; f -> g; f -> h; g -> k; k -> l; k -> h; h -> i; h -> j; i -> k; j -> k; }
  
```

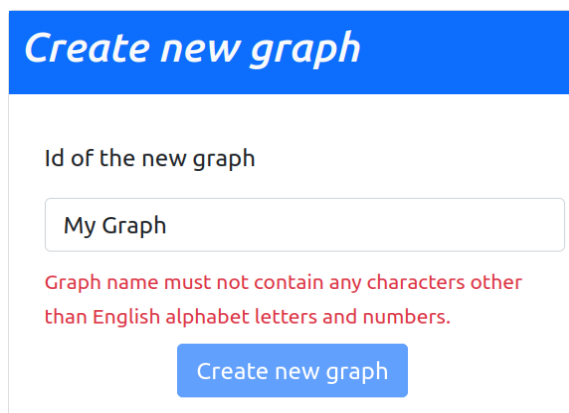
2.2. Eredményeket szemléltető képernyőképek, adatsorok

Indítás

A szoftver a böngészőben az alábbi képernyőfotón látható felületként jelenik meg.



Új gráf létrehozása



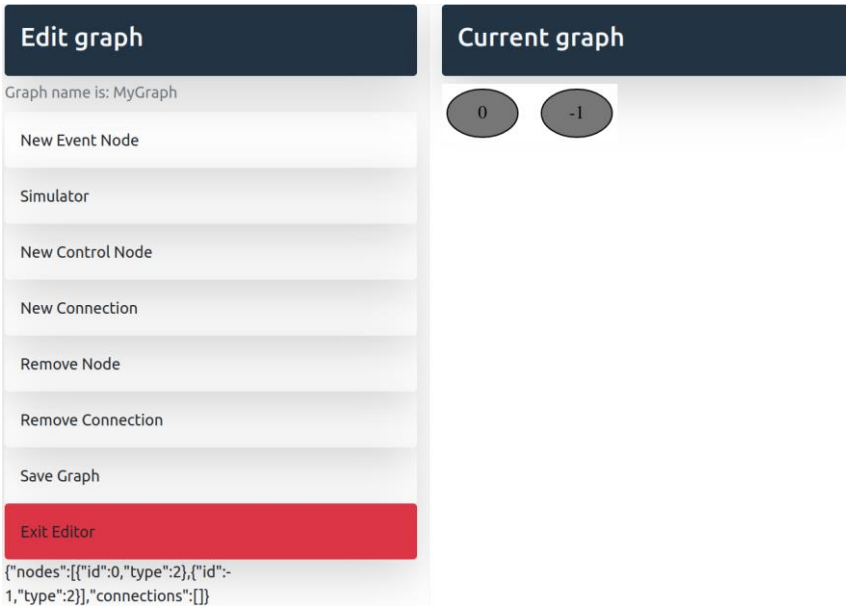
Gráf építése

A felület funkcionalitásának zöme a gráf építése során érhető el. Ezek a funkciók az alábbiak:

- Új eseménycsomópont hozzáadása (New Event Node).
- Gráf szimulálása (Simulator).
- Új kontrolcsomópont hozzáadása (New Control Node).

- Új kapcsolat hozzáadása (New Connection).
- Meglévő csomópont törlése (Remove Node).
- Meglévő kapcsolat törlése (Remove Connection).
- Gráf elmentése (Save graph).
- Szerkesztés befejezése (Exit Editor).

Mind egyik funkció egy menüpontként elérhető, mint az az alábbi képernyőfotón is látszik.



Új Event Node

Az új eseménycsomópont hozzáadásához kattintsunk a *New Event Node* felirat dobozára! Ez egy lenyíló menü, és a kattintás után egy form jelenik meg előttünk. A form minden mezőjét a címkéknek megfelelően kell kitölteni. Az azonosítónak nem negatív egész számoknak kell lenniük. A csomópont azonosítójának egyedisége ellenőrzésre kerül, és nem egyedi azonosító esetén hibüzenetet kapunk. Az ágens és az esemény nevére ugyanaz a korlátozás érvényes, mint a gráf nevére, tehát az angol ABC betűit és számokat tartalmazhatnak.

Az esemény időtartama a szimuláció során véletlenszerűen állapítódik meg, és erre a véletlen időtartamra állíthatunk alsó és felső korlátot a *Minimum time* és *Maximum time* mezők beállításával. Ezek a mezők csak pozitív vagy nulla értéket tartalmazhatnak.

Az alábbi képernyőképen látható egy kitöltött form.

Edit graph

Graph name is: MyGraph

New Event Node

id
1

Agent
Worker1

Event id
1

Event name
FormCreation

Minimum time
1

Maximum time
10

Add Node

Simulator

Current graph

0 -1

Amennyiben megfelelően kitöltöttük a form mezőit (egyik lokális ellenőrzés sem bukik el), az *Add Node* gombra kattintva felvihetünk egy új csomópontot a gráfba.

Ha a gráf azonosítójának egyedisége nem teljesül, akkor hibaüzenetet kapunk. A hibaüzenet egy felugró keretben jelenik meg, mint az a lenti képernyőfotón is látható.

Edit graph

Graph name is: MyGraph

New Event Node

id
1

Agent
Worker1

Event id
1

Event name
FormCreation

Minimum time
1

Maximum time
10

Add Node

Simulator

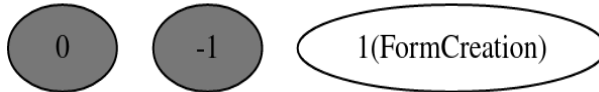
Error

Error reason: Node id 1 is already in use.

Close

1 (FormCreation)

Amennyiben a csomópont hozzáadása sikeres, a jobb oldali gráfon is meg fog jelenni az új csomópont. A felhasználó által hozzáadott eseménycsomópontok fehér színűek, a közepükben az azonosítójukkal, és zárójelben a nevükkel.



A csomópontok csak egyesével adhatók hozzá a gráfhoz.

Tipp: Amennyiben le akarjuk zárni a formok bármelyikét, a nyitott form fejlécére kattintva (ugyanoda, ahova a megnyitáskor kattintottunk) a form lezárul. A form akkor is lezárul, ha megnyitunk egy másik form elemet. Egyszerre csak egy form található nyitva.

New Control Node

Az új kontrollcsomópont hozzáadása a *New Control Node* menüpont alatt érhető el. A kontrollcsomópontok nem fognak a szimulációban megjelenni, helyette a szimuláció működését módosítják. Egyedi azonosítóval (id) ugyanúgy rendelkezniük kell, mint az eseménycsomópontoknak, azonban ezen kívül csak a típusát kell megadnunk. A típus lehet kétféle:

- XOR,
- Control.

A XOR csomópont valójában ugyanolyan, mint az eseménycsomópont, mindössze nem jelenik meg a szimulációban. Ilyenek a kezdő- és a végpontok is.

A Control típus a tényleges kontrollcsomópont, amely lehetővé teszi a párhuzamos eseményfutást a szimulációban. Minden egyes olyan esemény csomópont, amelyhez a Control csomópontból vezet él, párhuzamosan végrehajtható, és egészen addig párhuzamos ágakon futnak, amíg egy másik Control csomópontban össze nem futnak az élek.

Az alábbi képernyőkép mutatja a kitöltött kontrollcsomópont-formot, és a paraméterekkel hozzáadott csomópontot az ábrán. A Control csomópontokat kék háttérrel ábrázoljuk, míg a XOR kontrollcsomópontokat szürkével, mint az a kezdő- és végpont is látszik.

Graph name is: MyGraph

New Event Node

Simulator

New Control Node

id

2

Type

Control

Add Node

New Connection

Remove Node

Remove Connection

Save Graph

Exit Editor

```

{"nodes":[{"id":0,"type":2},{"id":1,"type":2},
{"id":1,"type":0,"agent":1,"event_id":1,"event_name":"FormCreation","min_time":1,"max_time":10},{"id":2,"type":1}],connections:[]}

```

Current graph

0 -1 1(FormCreation) 2

New Connection

Az új él (vagy új kapcsolat) hozzáadásához nyissuk le a *New Connection* menüt. Itt két lenyíló listában adhatjuk meg, hogy melyik csomópontból melyik csomópontba szeretnénk élt küldeni. Ezen felül itt adhatjuk meg az él súlyát is. A súly megadása a szimulációnál lesz lényeges, ugyanis az egy csomópontból kiinduló élek (amennyiben a csomópont nem Control csomópont) közül csak egy fog érvényesülni. Tehát egy esemény vagy XOR csomópontból egy szimulációban csak egy irányba haladunk tovább, és amennyiben több él vezet az élből, akkor az élek közül véletlenszerűen fog választani a szimulátor. A súlyok azt adják meg, hogy mekkora valószínűséggel fog érvényesülni az adott él a többihez képest. Amennyiben egy él 10-es súllyal rendelkezik, egy másik pedig 1-essel, akkor a 10-es él 10-szer akkora valószínűséggel kerül kiválasztásra.

Megjegyzendő, hogy adott számú ismétlődések jelenleg nem alakíthatók ki az alkalmazás segítségével. Ezt a viselkedést jelenleg csak közelíteni lehet a súlyokkal. Például az új kapcsolatok kialakításában nagy a felhasználó szabadsága. Egy él mutathat más csomópontra, de akár önmagára is (ilyenkor ismétlés alakul ki). Az alkalmazás jelenleg kevés ellenőrzést végez az élekkel kapcsolatban, építés során pedig gyakorlatilag csak a beírt adatok szintaktikai helyességét vizsgálja: a súlynak pozitív lebegőpontos számnak kell lennie. Itt a felhasználónak kell ügyelnie arra, hogy logikailag helyes gráfot adjon meg.

Az alábbi képernyőképen látható a nyitott form és néhány hozzáadott él a gráfon.

Edit graph

Graph name is: MyGraph

New Event Node

Simulator

New Control Node

New Connection

Start Node Id

End Node Id

Weight

[Create connection](#)

Remove Node

Remove Connection

Save Graph

[Exit Editor](#)

```
[{"nodes":[{"id":0,"type":2},{"id":1,"type":2},
{"id":1,"type":0,"agent":1,"event_id":1,"event_name":"FormCreation","min_time":1,"max_time":10},{"id":2,"type":1}],
"connections":[{"start_node_id":0,"end_node_id":1,"weight":1},
{"start_node_id":1,"end_node_id":2,"weight":10}]}
```

Current graph

```
graph TD
  0((0)) -- 1.0 --> 1([1(FormCreation)])
  1 -- 10.0 --> 2((2))
```

Remove Node

A csomópontok törlését a *Remove Node* menüpont alatt vitelezhetjük ki. Egy csomópont törlése szükséges lehet, amennyiben elrontottuk a gráf szerkesztését, és vissza akarunk törölni valamit, vagy pedig ha egy már meglévő gráfot akarunk át szerkeszteni valamilyen módon.

A csomópont törlésének formjában mindössze egy lenyíló lista található, amelyben az összes (a kezdő- és a végponton kívül) létező csomópont azonosítója lesz felsorolva. A törlés végrehajtásához válasszuk ki a törlendő csomópont azonosítóját a listából, majd nyomjuk meg a *Remove Node* gombot.

Megjegyzendő, hogy egy csomópont törlésekor az összes hozzá tartozó él is törlődik (tehát akkor is, ha belőle indul, és akkor is, ha benne ér véget). Például az alább látható képernyőfotó az idáig épített gráfról készült a Control csomópont törlése után; a törlés során a rá mutató él is automatikusan törlődött.

Edit graph

Graph name is: MyGraph

New Event Node

Simulator

New Control Node

New Connection

Remove Node

Node Id

Remove node

Remove Connection

Save Graph

Exit Editor

```
{
  "nodes": [
    { "id": 0, "type": 2 },
    { "id": 1, "type": 2 }
  ],
  "connections": [
    { "start_node_id": 0, "end_node_id": 1, "weight": 1 }
  ]
}
```

Current graph

```

graph TD
    0((0)) -- 1.0 --> 1([1(FormCreation)])
    -1((-1))
  
```

Remove Connection

Egy él törléséhez nyissuk meg a *Remove Connection* menüpontot! Ebben a pontban egy két lenyíló listából álló form található. Mind a két lenyíló listában az összes létező csomópont megtalálható. Az elsőben a törlendő él kezdőpontját, a másodikban a végpontját kell megadni. Ezután a *Remove Connection* gomb lenyomásával törölhetjük az adott élet.

Megjegyzendő, hogy amennyiben az adott él nem létezik, nem történik semmi; nem kapunk hibaüzenetet, és a gráf sem változik.

Remove Connection

Start Node Id

End Node Id

Remove connection

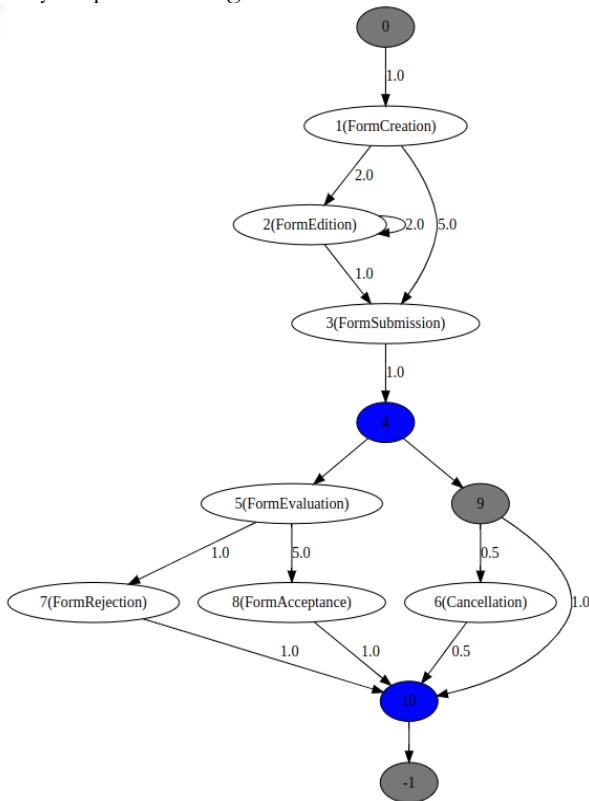
Save Graph

A gráf elmentéséhez használjuk a *Save Graph* menüpontot. Ez a pont csak egy gomb, amelyre ha rányomunk, a böngészőben azonnal letöltésre kerül a gráf leírója, amelyet az *Editor* oldal aljában is olvashatunk, kiegészítve a gráf azonosítójával. A gráf leírója egy JSON String, amely tartalmazza a formokban megadott adatokat. A letöltött fájl neve a gráf neve, plusz a .json kiterjesztés lesz.

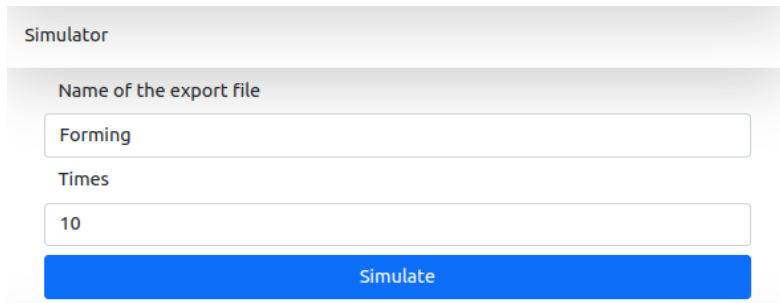
Megjegyzendő, hogy a letöltött gráfnak nem kell helyesnek lennie. A letöltés egyik célja, hogy elmentsük a gráfot a kliensgépen fájlként, és később ismét megnyissuk szerkeszteni.

Simulator

A letöltött gráf szimulálására a *Simulator* menüpontban van lehetőség. Fontos, hogy a gráf szimulálását csak akkor kíséreljük meg, amikor a gráf már valid. Tegyük fel, hogy elkészítettünk egy gráfot a szerkesztővel. Legyen ez a gráf például az alábbi képernyőképen látható gráf.



Ezt a gráfot tudjuk szimulálni, mikor meg van nyitva a szerkesztőben. Nyomjuk meg a *Simulator* menüpont téglalapját! Ekkor megnyílik az alább látható form, melyen a mentendő szimulációs fájl neve (kiterjesztés nélkül) írandó be, ezen felül az, hogy hányszor szeretnénk lefuttatni a szimulációt (hány futási esetet írjunk bele a fájlba).



Simulator

Name of the export file

Forming

Times

10

Simulate

Nyomjuk meg a *Simulate* gombot a futások generálásához! A lenyomás után (amennyiben a gráf helyes) egy .txt fájl kerül automatikusan letöltésre. A fájlban a különböző esemény csomópontjainak eseménynevei vannak a szimulációnak megfelelő mennyiségben és sorrendben. Egy lehetséges szimulációja a fenti gráfnak a következőképpen néz ki:

```
FormCreation, FormEdition, FormSubmission  
FormEvaluation, FormAcceptance  
Cancellation  
  
FormCreation, FormSubmission  
FormEvaluation, FormAcceptance  
  
FormCreation, FormSubmission  
FormEvaluation, FormAcceptance  
  
FormCreation, FormSubmission  
FormEvaluation, FormAcceptance  
  
FormCreation, FormEdition, FormEdition, FormEdition, FormEdition, FormSubmission  
FormEvaluation, FormAcceptance
```

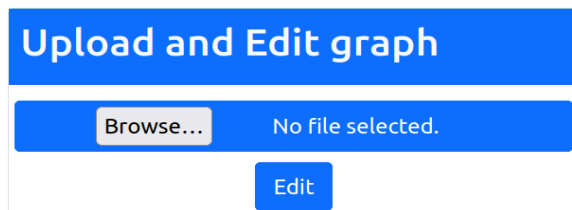
A fenti képernyőképen mindössze 5 eseménysor található, de a *Times* form elem állításával ez változtatható.

Exit Editor

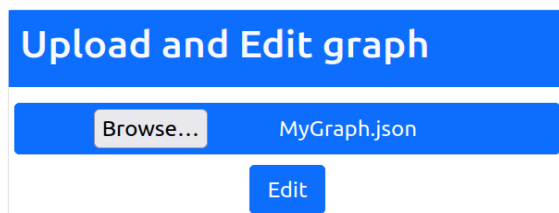
A szerkesztés és szimulálás befejezéséhez az *Exit Editor* gombbal léphetünk ki a felületről. Megjegyzendő, hogy a kilépés során nem mentődik el a gráf, amit készítettünk, ezért a kilépés előtt javasolt a gráf elmentésére szolgáló gombra rányomni.

Gráf betöltése

Az alkalmazásban lehetőség van egy korábban elmentett gráf feltöltésére, és a szerkesztés folytatására. Ez a művelet feltételezi, hogy a *Save Graph* szekcióban leírtak szerint mentettünk egy gráfot, vagy legalábbis rendelkezünk egy JSON-fájllal, ami valid módon leír egy gráfot. Amennyiben szerkeszteni akarunk egy gráfot, először az alkalmazás megnyitásakor nem a *Build new graph*, hanem az *Upload and Edit graph* gombra kell nyomnunk. Ekkor betöltődik az alább látható weblap.



Itt először fel kell töltenünk egy meglévő JSON-fájlt. Ehhez nyomjunk rá a fájl-feltöltő gombra a *Browse...* gomb vagy a teljes kék tárolója is megfelel. Amennyiben megnyomjuk, felugrik egy fájl kiválasztó ablak, amelyben értelemszerűen kiválasztjuk a megfelelő fájlt, és ezzel feltöltjük azt. Onnan tudjuk, hogy a fájl sikeresen feltöltődött, hogy a „No file selected.” szöveg kicserélődik a fájl nevére, mint az alábbi esetben.



Ha meggyőződünk, hogy a megfelelő fájl feltöltődött, nyomjuk meg az *Edit* gombot. Ennek következtében betöltődik a gráfépítő felület, aminek a működését a korábbiakban részletesen kifejtettük; a használata innentől megfelel a már leírtaknak.

2.3. Kutatási eredményeken alapuló javaslatok megfogalmazása

A kutatómunka egy nagyon érdekes területtel, az eseményleíró gráfok szerkesztésével és azok szimulációjával, azaz mintaadatok generálásával foglalkozott. Munkánk során megtervezésre került egy webes környezetben működő szoftver, amely segítségével egy tetszőleges böngészővel elkészíthetjük a szükséges gráfot. A szoftver a gráf alapján képes adathalmazt generálni. Az elkészült szoftver ebben a formában tehát képes az alapvető feladatokat elvégezni, működőképes. Azonban számos területen további potenciális fejlesztési lehetőségek merülnek fel, melyek megvalósításával egy még hatékonyabb tesztelő környezet lenne kiépíthető.

A FOLYAMATFELTÁRÓ ALGORITMUSOK ÁTTEKINTÉSE A PM4PY HASZNÁLATÁVAL

BAKSA ATTILA

A folyamatbányászati technikákat gyakran akkor alkalmazzák, amikor más megközelítésekkel nem lehet formális leírást kapni a folyamatról, vagy például, amikor a meglévő dokumentáció minősége kérdéses. Így találkozhatunk folyamatbányászattal például egy cég munkafolyamat-kezelő rendszerének auditálásakor, egy vállalatirányítási rendszer tranzakciónaplóinak elemzésekor, vagy egy kórház elektronikus betegnaplójának modellezésekor. A folyamatbányászati algoritmusok alkotják a folyamatbányászati szoftverek alapját. Az öt legnépszerűbb folyamatbányászati algoritmus közé tartozik az Alpha Miner, a Heuristic Miner, a Fuzzy Miner, az Inductive Miner és a Genetic Miner.

1. A folyamatbányászat

A folyamatbányászat olyan technikák összessége, melyeket eseményadatok elemzésére használnak annak érdekében, hogy megértsék és javítsák az operatív folyamatokat. Az adatvizsgálat mint tudomány és a folyamatkezelés részét képező folyamatbányászat alapját általában naplók adják, melyek tartalmazzák a tranzakciók azonosítóját, egy egyedi kulcsot az adott folyamathoz; egy tevékenységet, egy esemény leírását, amely lezajlott; egy időbélyeget; és néha további információt is, mint erőforrások, költségek, és így tovább.

Három fő folyamatbányászati cél létezik:

- *folyamatfeltárás (elemzés),*
- *illeszkedésvizsgálat,*
- *illetve folyamat-teljesítmény javítás.*

Ezekre korábban olyan kifejezéseket használta, mint *munkafolyamat-bányászat*, vagy *automatikus üzleti folyamatfeltárás*.

A folyamatbányászati technikákat gyakran akkor alkalmazzák, amikor más megközelítésekkel nem lehet formális leírást kapni a folyamatról, vagy például amikor a meglévő dokumentáció minősége kérdéses. Így találkozhatunk folyamatbányászattal például egy cég munkafolyamat-kezelő rendszerének auditálásakor, egy vállalatirányítási rendszer tranzakciónaplóinak elemzésekor, vagy egy kórház elektronikus betegnaplójának modellezésekor. Az esemény/folyamatnaplók felhasználhatók korábbi modell(ek)kel való összehasonlításra annak érdekében, hogy az észlelések megfelelnek-e egy előírt, vagy elvárt viselkedésnek. Fontos, hogy a naplóadatokat összekössék konkrét üzgyazonosítóval, tevékenységgel és időbélyeggel lássák el.

A célok szerinti folyamatbányászati technikák rövid bemutatása:

1. **Folyamatfeltárás** (elemzés): Ez a folyamatbányászat első lépése. A folyamatfeltárás fő célja az eseménynapló átalakítása folyamattá. Az eseménynapló bármely adattároló rendszerből származhat, amely rögzíti a szervezet tevékenységeit, valamint ezek időbélyegét. Az ilyen eseménynaplónak tartalmaznia kell az *ügyazonosítót* (egyedi kulcs azon eset felismerésére, melyhez az adott tevékenység tartozik), és a *tevékenység leírását* (a végrehajtott feladat szöveges ismertetése) és az ehhez köthető *időbélyeg*et.

A *folyamatfeltárás* eredménye általában egy olyan *folyamatmodell*, mely reprezentatív az eseménynaplóhoz. Ilyen folyamatmodellt fedezhetünk fel az *alpha* (mely egy didaktikusan vezérelt megközelítés), a heurisztikus bányász, illetve az induktív bányász algoritmusokkal.

Sok kialakult technika létezik a folyamatmodellek automatikus felépítésére az eseménynaplók alapján: például a *Petri-hálók*, *BPNM diagramok*, *tevékenységdiagramok*, *állapotsor-diagramok* és *EPC-k*.

A folyamatbányászati kutatások mostanában már más nézőpontokra is irányulnak, úgy mint az adatok, erőforrások, vagy időtényező összekapcsolása. Lehet olyan cél is például, hogy a modell segít egy társadalmi hálózatot létrehozni, valamilyen szempontrendszer szerint. Ma már olyan technikák fejlesztése folyik, mint az „áramló folyamatbányászat”, mely folyamatos online adatok alapján dolgozik, amit nyilván azonnal fel kell dolgozni. A lassú feldolgozás már nem érdekes. Itt tehát kulcsfontosságú az időtényező.

2. **Illeszkedésvizsgálat** segít összehasonlítani egy eseménynaplót egy már létező, vagy korábban felépített folyamatmodellel, azzal a céllal, hogy eldöntsük, mi változott, mi a különbség a korábban kapott eredményekhez képest. Egy ilyen folyamatmodellt kézzel vagy egy ún. feltáró algoritmussal lehet felépíteni. Például egy folyamatmodell jelezheti, hogy az egymillió forintnál nagyobb vásárlási megrendeléseknél két ellenőrzés van szükség. Egy másik feltétel lehet bizonyos banki tranzakciók, folyamatok ellenőrzésénél, a két személy jóváhagyása (négy szem) elv.

Az illeszkedésvizsgálatot használhatják eltérések észlelésére (megfelelőség-ellenőrzés), vagy értékelhetik a feltáró algoritmusokat, vagy kibővíthetnek egy meglévő folyamatmodellt. A folyamatmodell kiterjesztése teljesítményadatokkal azt jelenti, amikor valamilyen a priori folyamatmodellt használnak a potenciális zsákutcák előrejelzésére. Az eseménynaplót minden lehetőség esetén megvizsgálják annak meghatározása érdekében, hogy milyen információ áll rendelkezésre általában a döntés meghozatalakor. Az illeszkedésvizsgálatnak többféle technikája van, például „*token*

alapú visszajátzás”, „*áramló illeszkedésvizsgálat*”, amelyeket a rendszer igényeinek megfelelően használnak. Ezután a klasszikus adatbányászati technikákat használják annak megállapítására, hogy mely adatelemek befolyásolják a döntést. Ennek eredményeként minden döntésre a folyamatban egy döntési fa generálódik.

3. A **folyamat-teljesítmény javítás** akkor használatos, amikor van már egy előre ismert modell. A modellt továbbfejlesztik az azonosítható teljesítményinformációkkal – ilyenek pl.: a feldolgozási idő, ciklusidő, várakozási idő, költség stb. adatok – azért, hogy a cél ne csak az illeszkedésvizsgálat legyen, hanem az, hogy javítsák a meglévő modell teljesítményét bizonyos folyamat-teljesítmény mutatóknak megfelelően. Lehetőség van a folyamatmodellek kiterjesztésére további ellenőrzések beiktatásával, mint pl. a döntési szabályok és szervezeti szerepek figyelése.

2. A folyamatbányászat rövid bemutatása

A következőkben röviden áttekintjük a folyamatbányászat alapvető lépéseit néhány egyszerű példa segítségével. Természetesen nem célunk a minden részletre kiterjedő ismertetés, itt csak az alapok áttekintését célozzuk meg. Ha bővebben kíváncsiak vagyunk erre, akkor a folyamatbányászat irodalmának áttekintését javasoljuk, kiindulásként használható pl. a következő online kurzus a Coursera MOOC honlapjáról (<https://www.coursera.org/learn/process-mining>) vagy Wil van der Aalst: *Process Mining, Data Science Action* című könyve (<https://doi.org/10.1007/978-3-662-49851-4>).

Mindenekelőtt telepítsük fel a **Python** programozási környezetünkbe a **PM4Py** csomagot is, ehhez bőven találunk online leírást (<https://pm4py.fit.fraunhofer.de/static/assets/api/2.7.8/install.html>).

3. Üzleti folyamatok a mai világunkban

A cégek nagy többsége, tevékenységüktől függetlenül állandóan folyamatokat hajt végre, legyen az pl. egy ételkiszállító cég, vagy egy járműgyártó, vagy egy bank, esetleg biztosítótársaság. A „folyamat” tulajdonképpen többféleképpen is definiálható, de egy sokak által elfogadható változat azt mondja, hogy „*a folyamat tevékenységek gyűjteményét jelenti, melyeket egy bizonyos cél elérése érdekében hajtunk végre*”.

Tekintsünk pl. egy pizza-házhozszállítással foglalkozó céget. Először felhívjuk az éttermet, hogy megrendeljük a kívánt pizzát; az első lépés, amit a cégnél dolgozó (Péter) megtesz, hogy felírja a megrendelésünket. Tegyük fel, hogy egy

sonka-gombás pizzát kértünk. Ezután Péter elkéri a címünket és a telefonszámunkat. Végül megkérdezi, hogy milyen módon szeretnénk fizetni, és közli velünk, hogy kb. mikorra várható a pizza házhoz szállítása. A cégnél ekkor kezdődik a munka a megrendelésünkkel kapcsolatban: a szakács (Sanyi) megkapja az igényünket és azonnal elkezd elkészíteni a kért pizzát. Ezalatt természetesen további hívások is befutnak a céghez, Péter felveszi a soron következő megrendeléseket is. Amikor kész az általunk megrendelt pizza, azt becsomagolja Péter és továbbadja a pizzafutárnak Katinak, aki kiszállítja a pizzát.

Ez tehát tömören felírva a következő folyamatokat jelenti egymás után.

1. Péter felveszi a megrendelést.
2. Péter feljegyzi a címet és a telefonszámot.
3. Péter beállítja a fizetési módot.
4. Sanyi elkészíti a pizzát.
5. Péter becsomagolja a pizzát.
6. Péter kinyomtatja a számlát.
7. Péter átadja a csomagot Katinak.
8. Kati kiszállítja a megrendelést.

1. lépés: Az események megléte a PM4Py számára olvasható állományokba. Mint azt előzőleg írtuk, a folyamatbányászat eseménynaplókat használ a folyamat ismereteinek generálására. Számos információs rendszer, mint pl. SAP, Oracle lehetővé teszi számunkra, hogy valamilyen módon kinyerjük az eseménynaplókat, hasonlóan az előző résznél olvasható példanaplóhoz.

A következő példánál feltételezzük, hogy rendelkezésre állnak az eseményadatok egy megfelelő eseménynapló-formátumban.

Alapvetően kétféle fájlformátumot használunk a log txt állományok vizsgálata során: CSV- és XES-állományokat. Ez természetesen két szabványos adatformátum, a CSV – a vesszővel elválasztott értékek soraiból áll, míg a XES – az eXtensible Event Stream fájl, mely egy XML állományt takar, mely lehetővé teszi a folyamat viselkedésének leírását (bővebben lásd: <https://www.xes-standard.org>)

Egy ilyen fájlban minden sor egy bekövetkezett eseményt ír le. A mezők ugyanolyan típusú adatot képviselnek, pl.: azonosító, tevékenység, időbélyeg, költség, erőforrás. A fenti példánál megadott eseményeket listázzuk ki a CSV-fájlban, pl. az alábbiakat látjuk (lásd 1. ábra):

```
1 case_id;activity;timestamp;costs;resource
2 1;kérés rögzítése;2023-10-15 14:39;100;Péter
3 1;elérhetőség ellenőrzése;2023-10-15 14:40;100;Péter
4 1;cím felvétele;2023-10-15 14:41;100;Péter
5 1;fizetési mód;2023-10-15 14:42;100;Péter
6 1;pizzakészítés;2023-10-15 14:57;3000;Sanyi
7 1;pizzacsomagolás;2023-10-15 15:02;100;Péter
8 1;számlanyomtatás;2023-10-15 15:03;100;Péter
9 1;csomagátadás;2023-10-15 15:04;100;Kati
10 1;pizzakiszállítás;2023-10-15 15:25;500;Kati
11 2;kérés rögzítése;2023-10-15 15:10;100;Péter
12 2;elérhetőség ellenőrzése;2023-10-15 15:11;100;Péter
13 2;döntés;2023-10-15 15:12;100;Péter
14 2;visszautasítás;2023-10-15 15:13;100;Péter
```

1. ábra. Események CSV-fájlban

Kiemelendő, hogy az 1. ábrán bemutatott adatok egy táblázatot adnak, szöveges formátumban. Minden fájlbeli sor megfelel egy-egy táblázatbeli sornak, a sorban szereplő pontosvessző, jelenti a mezőelválasztó jelet, azaz a különböző oszlopokat/mezőket választja el. A fájl legelső sorában a mezők/oszlopok neve látszik. Tehát látható, hogy 5 oszlopa van a táblázatnak. Az első oszlop az eset azonosítóját reprezentálja, azaz lehetővé teszi számunkra, hogy azonosítsuk, milyen tevékenységet rögzítettek annak a folyamatpéldánynak a kontextusában. A második oszlop azt jelzi, hogy melyik tevékenységet végezték el. A harmadik oszlop azt mutatja, hogy az adott tevékenységet mikor rögzítették. Ebben a példában további információ is nyilván van tartva. A negyedik oszlop nyomon követi a tevékenység költségeit, míg az ötödik sor adja, hogy ki (a cég melyik alkalmazottja) végezte el a tevékenységet.

A fájl az egyszerűség kedvéért, csak két folyamatot rögzített az elsőt a 2–10. sorok, míg a másodikat a 11–14 sorok jelzik. A folyamatok mindig a rögzítéssel kezdődnek, és a pizza elérhetőségétől függően kiszállítással, vagy visszautasítással zárulnak.

2a. lépés: *A CSV-állomány betöltése* a PM4Py környezetbe, a további feldolgozás érdekében

Az eseménynaplókkal és az eseménynaplók CSV-fájlban való reprezentációjával megismerkedve itt elkezdünk némi folyamatbányászatot. Betöltjük az eseményadatokat, és megszámloljuk, hogy hány eset/esemény található az eseménynaplóban, valamint hány folyamat. Mindezt egy **pandas** nevű Python programmal végezzük. Ezt azért tesszük, mert a **pandas** a de facto szabvány a CSV-alapú adatok betöltésére/kezelésére. Így bármely **PM4Py**-ban megvalósított folyamatbányászati algoritmus, amely eseménynaplót használ bemenetként, közvetlenül működhet egy **pandas** fájlal.

```
1 import pandas
2 import pm4py
3
4 |
5 def import_csv(file_path):
6     event_log = pandas.read_csv(file_path, sep=';')
7     num_events = len(event_log)
8     num_cases = len(event_log.case_id.unique())
9     print("Események száma: {}".format(num_events, num_cases))
10
11
12 if __name__ == "__main__":
13     import_csv("process-list.csv")
14
```

2. ábra. Az eseménynapló CSV-állományának beolvasását elvégző programkód, mely megadja az események számát, és az ügyek/folyamatok számát is

Áttekintve a fentebb látható példa kódját: az első sorban importáljuk a **pandas** könyvtárat. Az utolsó sorok (amelyek az **if** utasítást tartalmazzák) biztosítják, hogy a kód önállóan is fusson. A szkript magja az **import_csv** függvény. Bemeneti paraméterként megköveteli a CSV-fájl elérési útját. A szkript a **pandas read_csv** függvényt használja az eseményadatok betöltésére. Az események számának kiszámításához egyszerűen lekérdezzük az adatkeret hosszát, azaz **len(event_log)** meghívásával. Az esetek számának kiszámításához beépített **pandas** funkciót használunk, hogy visszaadja a **case_id** oszlop egyedi értékeinek számát, azaz **event_log.case_id.unique()**. Mivel ez a funkció egy **pandas** beépített tömbobjektumot ad vissza, amely tartalmazza az oszlop összes értékét, újra lekérdezzük annak hosszát. Vegyük észre, hogy számos mód van már a fent említett példa kapcsán is statisztikák számítására. A példa futásának eredményét a 3. ábra mutatja be.

```
abaksa@HomePC:~/work/Projects/ERPA$ python3 -W "ignore" test.py
Események száma: 13
Ügyek száma: 2
```

3. ábra. Az 1. program kimenete

Most, hogy betöltöttük az eseménynaplót, használható tovább a **PM4Py**, ha nemcsak az események és esetek száma érdekel minket, hanem azt is szeretnénk megtudni, hogy mely tevékenységek történnek először, és melyek történnek utoljára az eseménynapló által leírt események során. A **PM4Py**-nak erre specifikus beépített funkciói vannak, **get_start_activities()** illetve a **get_end_activities()**. Tekintsük a 2. példát, amelyben a megfelelő szkriptet mutatjuk be, lásd 4. ábra.

```

1 import pandas
2 import pm4py
3
4
5 def import_csv(file_path):
6     event_log = pandas.read_csv(file_path, sep=';')
7     event_log = pm4py.format_dataframe(event_log, case_id='case_id',
8                                     timestamp_key='timestamp', activity_key='activity')
9     start_activities = pm4py.get_start_activities(event_log)
10    end_activities = pm4py.get_end_activities(event_log)
11    print("Indító tevékenység: {}".format(start_activities,
12                                         end_activities))
13
14 if __name__ == "__main__":
15    import_csv("process-list.csv")

```

4. ábra. A 2. feladat a kezdő és a záró feladat kiírása az eseményekre

Ebben a 2. példában látható, hogy mind a **pandas**, mind a **pm4py** könyvtárakat importáljuk. A szkript első sora ismét betölti az eseménynaplót, amelyet CSV-formátumban tároltunk. A második sor átalakítja az eseményadatokat táblázatát olyan formátumra, amelyet bármely folyamatbányászati algoritmus használhat a **pm4py**-ban. Azaz a **format_dataframe()**-függvény egy másolatot készít a bemeneti eseménynaplóról, és átnevezi a hozzárendelt oszlopokat a **pm4py**-ban használt standardizált oszlopnevekre. A standard nevek használatának alapvető oka elsősorban az XES- (amely a következő fájlformátum, amelyet hamarosan megvizsgálunk) örökséghez kapcsolódik.

Fontos megjegyezni, hogy ebben a példában az argumentumok értékei, azaz a **sep**, **case_id**, **activity_key** és **timestamp_key** az input adatoktól függenek. Azoknak az aktivitásoknak az előfordulását, amelyek először és utoljára fordulnak elő bármely nyomon az eseménynaplóban, az **pm4py.get_start_activities(event_log)** és az **pm4py.get_end_activities(event_log)** függvényekkel kérjük le. A függvények egy szótárat adnak vissza, amely tartalmazza az aktivitásokat kulcsként, és az eseménynaplóban előforduló megfigyelések számát (azaz azoknak a soroknak a számát, amelyekben először, illetve utoljára előfordulnak).

A **PM4Py** beépített **pandas** funkciót használ az input adatok időbélyeg-formátumának automatikus észlelésére. Azonban a **pandas** minden egyes sorban külön vizsgálja az időbélyegző értékeit. Néhány esetben ez azonban gondot okozhat. Például ha a dátum év-hónap-nap, vagy nap-hónap-év formátumot használ. Ennek a problémának enyhítésére további paramétert adhatunk meg a **format_dataframe()** módszernek, azaz a **timest_format** paramétert. A Python alapértelmezett időbélyegző-formátum kódjait használhatjuk az időbélyegző-formátum megadására. Ebben a példában az időbélyegző formátuma `%Y-%m-%d %H:%M`. Általában javasolt az időbélyegző formátum megadása.

```

/home/abaksa/PycharmProjects/PSAIE2024/.venv/bin/python /home/abaksa/work/Proje
Indító tevékenység: {'kérés rögzítése': 2}
Záró tevékenység: {'pizzakiszállítás': 1, 'visszaütasítás': 1}

```

5. ábra. A 2. program kimenete

2b. lépés: A CSV-fájl exportálása XES-állományba

Egy pandas által beolvasott adatkeretet lehetséges XES-fájlba tárolni. Ez egyszerűen megtehető a `pm4py.write_xes()` függvény meghívásával. Átadhatja az adatkeretet bemeneti paraméterként a függvénynek, azaz a `pm4py` adatkeretet belső konverzióját végzi el az eseménynapló objektummá, mielőtt azt leírná a háttértárra. Ez a szerkezet csak akkor működik, ha az valóban egy adatkeretet jelent, azaz ahogyan azt korábban is jeleztük a CSV importálás szakaszában.

```
1 import pandas as pd
2 import pm4py
3
4 if __name__ == "__main__":
5     event_log = pm4py.format_dataframe(pd.read_csv('process-list.csv', sep=';'),
6                                     case_id='case_id',
7                                     activity_key='activity', timestamp_key='timestamp')
8     pm4py.write_xes(event_log, 'process-list-exported.xes')
9
```

6. ábra. A 3. program forráskódja

A 3. program kódrészlete az 6. ábrán látható, mely sikeres import esetén a 7. ábrán látható üzenet jelzi ki.

```
abaksa@homePC:~/work/Projects/ERPAS/python3 -W "ignore" test.py
exporting log, completed traces :: 100% | 2/2 [00:00<00:00, 5817.34it/s]
```

7. ábra. A 3. program kimenete

2c. lépés: XES-fájlban tárolt eseménynapló beolvasása

A CSV-fájlok mellett az eseményadatok XML-alapú formátumban, azaz XES-fájlokban is tárolhatók. Az XES-fájlban egy tartalmazási viszonyt írunk le, azaz egy napló tartalmaz egy sorozatot, amelyek további eseményeket tartalmaznak. Továbbá, egy objektum, azaz egy napló, sorozat vagy esemény attribútumokkal rendelkezhet. Az előny az, hogy bizonyos adatattribútumok, amelyek konstansok egy napló vagy egy sorozat esetén, azon a szinten tárolhatók. Például vegyük azt, hogy csak egy eset teljes költségeit ismerjük, nem pedig az egyes események költségeit. Ha ezt az információt egy CSV-fájlban szeretnénk tárolni, akkor ezt az információt meg kell ismételni (azaz csak olyan sorokban tárolhatunk adatokat, amelyek közvetlenül hivatkoznak eseményekre), vagy pedig egyértelműen meg kell határoznunk, hogy bizonyos oszlopok csak egyszer kapnak értéket, azaz esetszintű attribútumokra hivatkoznak. Az XES-szabvány egyszerűbben támogatja ezen típusú információ tárolását.

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <Log xes:version="1040-2016" xes:features="nested-attributes" xmlns="http://www.xes-standard.org/">
3   <extension names="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext" />
4   <extension names="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext" />
5   <string key="origin" value="csv" />
6   <trace>
7     <string key="concept:name" value="1" />
8     <event>
9       <int key="case id" value="1" />
10      <string key="activity" value="kéres rögzítése" />
11      <date key="timestamp" value="2023-10-15T14:39:00+00:00" />
12      <int key="costs" value="100" />
13      <string key="resource" value="Péter" />
14      <string key="concept:name" value="kéres rögzítése" />
15      <date key="time:timestamp" value="2023-10-15T14:39:00+00:00" />
16      <int key="@index" value="0" />
17      <int key="@case_index" value="0" />
18    </event>
19  </event>
20  <event>
21    <int key="case id" value="1" />
22    <string key="activity" value="elérhetőség ellenőrzése" />
23    <date key="timestamp" value="2023-10-15T14:40:00+00:00" />
24    <int key="costs" value="100" />
25    <string key="resource" value="Péter" />
26    <string key="concept:name" value="elérhetőség ellenőrzése" />
27    <date key="time:timestamp" value="2023-10-15T14:40:00+00:00" />
28    <int key="@index" value="1" />
29    <int key="@case_index" value="0" />
30  </event>
31  <event>
32    <int key="case id" value="1" />
33    <string key="activity" value="cím felvétele" />
34    <date key="timestamp" value="2023-10-15T14:41:00+00:00" />
35    <int key="costs" value="100" />
36    <string key="resource" value="Péter" />
37    <string key="concept:name" value="cím felvétele" />
38    <date key="time:timestamp" value="2023-10-15T14:41:00+00:00" />
39    <int key="@index" value="2" />
40    <int key="@case_index" value="0" />
41  </event>

```

8. ábra. Események XES-fájlban

3. lépés: A folyamatmodell kirajzolása – folyamatfeltárás

Mivel megvizsgáltuk a folyamatbányászat alapvető fogalmi koncepcióit és az eseményadatok feldolgozását, most a folyamatok feltárására összpontosítunk. Ahogy jeleztük, a cél az, hogy azonosítsuk, azaz elsősorban teljesen automatizált és algoritmikus módon, egy olyan folyamatmodellt, amely pontosan leírja a folyamatot. Például, a jelen példa eseményadatainak megadásakor azt a folyamatmodellt kívánjuk feltárni, amelyet feljebb bemutattunk az ételrendelés kapcsán. Most röviden bemutatjuk, hogy milyen modellezési lehetőségek léteznek a **PM4Py**-ban különböző folyamatfeltáró algoritmusok alkalmazása során. Majd áttekintést adunk az implementált folyamatfelfeltáró algoritmusokról.

A **PM4Py** jelenleg három különböző folyamatmodellezési algoritmust támogat. Ezek a következők: a *BPMN-diagramok*, a *folyamatfák* és a *Petri-háló*k. A *Petri-háló* egy matematikaibb reprezentáció a BPMN-hez képest. Gyakran a Petri-háló viselkedése nehezebben érthető, mint a BPMN-modelleké. Azonban matematikai természetük miatt a Petri-háló általában logikusan leírják a viselkedést (azaz nem okoz félreértést az általuk leírt működés). A *folyamatfák* szigorú részalmazát képviselik a Petri-hálóknak, és a folyamatviselkedést hierarchikus módon írják le. Jelenleg elsősorban a BPMN modellekre és a folyamafákra fogunk összpontosítani.

További információért a Petri-hálókról és alkalmazásukról folyamatmodellezésben ajánljuk ezt a cikket (<https://doi.org/10.1142/S0218126698000043>).

Érdekes módon egyik algoritmus sem fedezi fel közvetlenül a BPMN-modellt a **PM4Py**-ban. Azonban bármely folyamatfa könnyen átfordítható egy BPMN modellre. Mivel a BPMN-modellek alapvető operátorai egyszerűek, így kezdjük a folyamatfa feltárással, amelyet átalakítunk majd egy BPMN-modellre. Később megvizsgáljuk az „alapvető” folyamatfát. Az algoritmus, amelyet fel fogunk használni, az „*Inductive Miner*”.

```

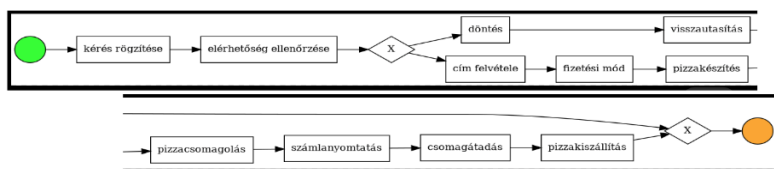
1 import pm4py
2 if __name__ == "__main__":
3     log = pm4py.read_xes('process-list-exported.xes')
4
5     process_tree = pm4py.discover_process_tree_inductive(log)
6     bpmn_model = pm4py.convert_to_bpmn(process_tree)
7     pm4py.view_bpmn(bpmn_model)
8     pm4py.view_tree(process_tree)

```

9. ábra. A BPMN-modellt létrehozó 4. program

A létrejövő folyamatmodell az alábbi 10. ábrán látható. Megfigyelhető, hogy a feltárt folyamatmodell valójában azt a viselkedést írja le, ami a pizzarendelés során ténylegesen tapasztalható.

Ez lényegében egy folyamatfa előállítását teszi meg. Az ilyen folyamatfa egy gyökérrel rendelkező fa, melyet „vezérlési/áramlási” információval látunk el.



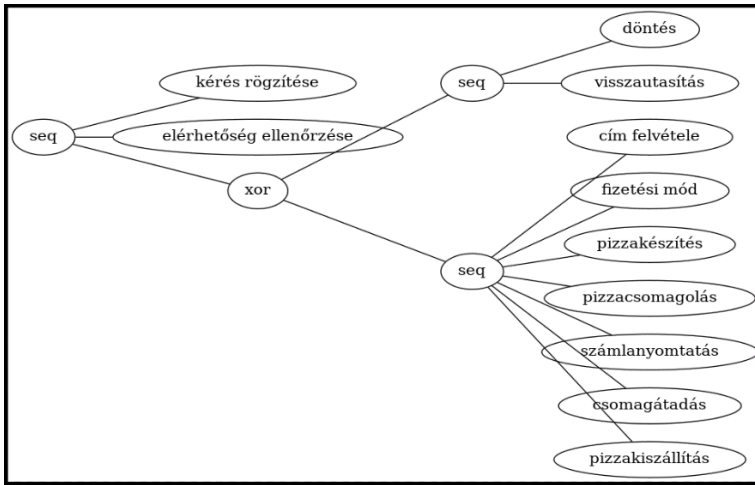
10. ábra. A BPMN folyamatmodell a pizzakészítéshez

```

1 import pm4py
2 if __name__ == "__main__":
3     log = pm4py.read_xes('process-list-exported.xes')
4
5     process_tree = pm4py.discover_process_tree_inductive(log)
6     pm4py.view_process_tree(process_tree)

```

11. ábra. A folyamatfa előállítását célzó 5. programkód



12. ábra. A folyamatfa modell, mely az „inductive miner” algoritmussal készült

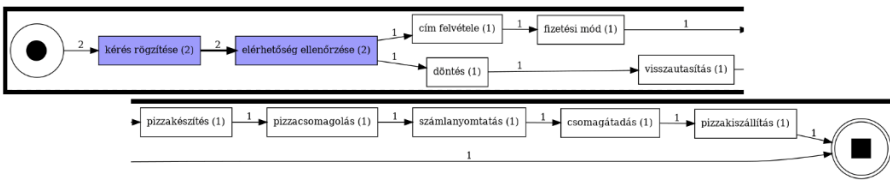
Sok kereskedelmi folyamatbányászati megoldás nem nyújt kiterjesztett támogatást a folyamatmodellek feltárásához. Gyakran, a folyamatok fő vizualizációjaként, folyamatábrák használhatóak csak. Egy folyamatábra tevékenységeket és közöttük kapcsolatokat tartalmaz (ívek formájában). Az egyik és egy másik tevékenység közötti kapcsolat általában azt jelenti, hogy valamilyen előzménykapcsolat áll fenn közöttük. A legegyszerűbb formájában ez azt jelenti, hogy a „forrás” tevékenység közvetlenül megelőzi a „cél” tevékenységet. Nézzünk egy konkrét példát! Tekintsük az alábbi kódrészletet, amelyben megmutatunk egy DFG-alapú folyamatábrát (13. és 14. ábra):

```

1 import pm4py
2
3 if __name__ == "__main__":
4     log = pm4py.read_xes('process-list-exported[xes']
5
6     dfg, start_activities, end_activities = pm4py.discover_dfg(log)
7     pm4py.view_dfg(dfg, start_activities, end_activities)
8

```

13. ábra. Folyamatkép-előállítást célzó 6. program forráskódja



14. ábra. Folyamatkép a pizzarendelés során zajló eseményekhez

4. Folyamatbányász algoritmusok áttekintése

(<https://www.workfellow.ai/learn/process-mining-algorithms-simply-explained>)
Az algoritmusok a számítógépes programozás alapjai, továbbá az adatfeldolgozás tudományának lényeges részei is egyben. Itt az öt legnépszerűbb algoritmust ismertetjük, amelyet a folyamatbányászatban használnak.

A folyamatbányászati algoritmusok alkotják a folyamatbányászati szoftverek alapját. Az öt legnépszerűbb folyamatbányászati algoritmus közé tartozik az Alpha Miner, a Heuristic Miner, a Fuzzy Miner, az Inductive Miner és a Genetic Miner.

A. Alpha Miner (α -algoritmus)

Ez az algoritmus az eseménynaplókat vagy a megfigyelt adatokat kapcsolja össze a folyamatmodell feltárással. Az *Alpha Miner* az első olyan folyamatfeltáró algoritmus volt, amelyet Will van der Aalst, Ton Weijters és Laura Maruster dolgozott ki.

Az *Alpha Miner* algoritmus eseménynaplókat használ adatforrásként. Az eseménynaplókat átalakítja közvetlenül egymást követő, sorrendi, párhuzamos és választási kapcsolatokká, majd ezeket felhasználva egy *Petri-háló*t hoz létre, amely leírja a folyamatmodellt. Egyszerűen fogalmazva, időbélyegzős üzleti folyamatáramlást hoz létre, amelyet meg lehet jeleníteni.

Ma az *Alpha Miner* algoritmust és annak változatait széles körben lehet használni a folyamatbányászati alkalmazásokban, például a folyamatfeltárásban és a megfelelés ellenőrzésében.

B. Heuristic Miner (Heurisztikus bányász algoritmus)

A második legnépszerűbb folyamatbányászati algoritmus, a *Heuristic Miner*, Ton Weijters fejlesztette ki annak érdekében, hogy kezelje az *Alpha Miner* néhány kulcsfontosságú korlátját. A heurisztika a számítástechnikában olyan technika, amelyet arra terveztek, hogy egy problémát gyorsabban oldjon meg, egy közelítő megoldást találva, amikor a klasszikus algoritmusok pontos megoldást keresnek. A heurisztikus algoritmusok népszerűek a mesterséges intelligenciában, ahol nagy mennyiségű adat és a gépi tanulás alapján meghatározott elég jó válaszok képződnek.

Ahogy az *Alpha Miner* algoritmusnál, a heurisztikus bányász is egy követő gráfot használ az eseménynaplók alapján az üzleti folyamatok sorrendjének megjelenítésére. A fő különbség az, hogy a *Heuristic Miner* alkalmaz szűrést a zaj csökkentése érdekében, vagyis a jelentéktelen vagy hiányos eseménynapló-adatokat, hogy olyan folyamatábrákat biztosítson, amelyek kevésbé pontosak, de hibátűrőbbek, mint az *Alpha Miner* által szolgáltatott modell.

A heurisztikus bányász algoritmusok egy olyan ábrázolást használnak, amit kauzális hálónak neveznek, hogy felrajzolják az összes időtörténetet a különböző tevékenységeknek. Ezután megvizsgálják, hogy milyen gyakran fordulnak elő különböző események, és olyan folyamatmodellt hoznak létre, amely kizárja a legkevésbé gyakori útvonalakat a megjelenített modelltől. Az eredmény jól működik bonyolult adatkörnyezetekben is, például folyamatbányászat esetén különösen nagy mennyiségű folyamatokra.

C. Fuzzy Miner

A *Fuzzy Miner* a harmadik alapvető folyamatbányászati algoritmus, amely alkalmas a kevésbé strukturált folyamatok modellezésére. Christian W. Günther fejlesztette ki, és a célja, hogy a folyamatbányászat segítségével a nagyobb adathalmazok elemzését is lehetővé tegye, azáltal hogy a lényegre összpontosít, tehát csak azt vizsgálja, amit a felhasználó feltárni és elemezni kíván.

A *Fuzzy Miner* jelentőség/korreláció mutatókat használ interaktív módon a folyamatmodell egyszerűsítésére egy-egy kívánt részletességi szinten. Egyszerűbben fogalmazva, csak egy bizonyos szintű adatbányászatot végez el a felhasználó keresési helyének megfelelően. Ha a felhasználó részletesebb elemzést kíván, a modell több szintet/részletet fog tartalmazni. Amikor a felhasználó egy magasabb szintű nézetet vizsgál csak, a modell csoportosítva lesz és „homályosabbá” válik.

D. Inductive Miner

Az *Inductive Miner* egy másik gyakran használt algoritmus a folyamatbányászatban, amely folyamatmodelleket fedez fel az eseménynaplókból. Ez a technika az eseménynaplókat kisebb résznaplókra vágja, amelyeket „vágásoknak” vagy „felosztásoknak” nevezünk, majd különféle vágásokat észlel a követő gráfban, amelyet az eseménynaplók felhasználásával hoztak létre. Az *Inductive Miner* fő előnye a flexibilitás és skálázhatóság.

Az *Inductive Miner* egyedi aspektusa a különböző felosztások feltáró módszere az események egymás követését leíró gráfban, és a felosztások után a kisebb komponensek használata az aktivitások végrehajtási sorrendjének reprezentálására. Az *Inductive Miner* algoritmus iteratíván feltárja a lehetséges folyamatmodelleket, és képes széles körű folyamatstruktúrák észlelésére, a lineáris modelltől a komplexebb modellekig terjedően, amelyek tartalmazhatnak konkurenciát, ciklusokat, párhuzamos és / vagy-ágakat.

E. Genetic Miner

A *Genetic Miner* a biológiából származó nevét kapta, és hasonló módon működik, mint a természetes kiválasztódás. Egy genetikus algoritmust használ, hogy keresse a lehetséges folyamatmodellek terét, és azonosítsa a legvalószínűbb folyamatmodellt. A *Genetic Miner* egy evolúciós megközelítésként értelmezhető, amely magában foglalja a folyamatmodellek mutálását és kombinálását a jobb modellek kereséséhez.

A *Genetic Miner* algoritmus minden folyamatmodellt kiértékel, és kiválasztási, kereszteződési és mutációs műveleteket használ az új folyamatmodellek létrehozásához. A folyamatmodelleket értékelik, és a legalkalmasabb modellt választják ki az utolsó folyamatmodellnek. A *Genetic Miner* képes azonosítani a többváltozatos folyamatmodelleket, és képes kimutatni a bonyolult folyamatstruktúrákat, például a ciklusokat és a konkurenciát.

FOLYAMATFELTÁRÓ ALGORITMUSOK KOMPLEX VIZSGÁLATA SZINTETIKUS ADATHALMAZOK FELHASZNÁLÁSÁVAL

BAKSÁNÉ VARGA ERIKA

A kutatási időszakban a feladatunk az volt, hogy a benchmark teszt adathalmazokra lefutassuk a PM4Py programkönyvtárban implementált folyamatfeltáró algoritmusokat és értékeljük az előállított folyamatmodelleket. A vizsgálat elején 12 algoritmust próbáltunk ki, amelyek közül több nem végzi el a folyamatmodell megfelelő mértékű általánosítását, ezért nagy méretű eseménynaplók esetén a gyakorlatban nem alkalmazható. Ezt a szempontot figyelembe véve, végül három algoritmust alkalmaztunk a teszt adathalmazokra: az Alpha Miner, a Heuristic Miner és az Inductive Miner algoritmusokat.

1. Elvégzett kísérletek bemutatása

1.1. Feladat célkitűzése, motiváció, elvégzendő szakmai feladat bemutatása

Az ERPA-projekt keretében a munkacsoportunk feladata mesterségesintelligencia-alapú folyamatfeltáró algoritmus(ok) tervezése és megvalósítása, ami képes nagy méretű eseménynaplók alapján az üzleti folyamatok hatékony feltárására. Ehhez kapcsolódóan az adott kutatási időszakban, mi azt a feladatot kaptuk, hogy készítsünk benchmark tesztek, amelyek eredményeivel a munkacsoport által elkészített algoritmusok hatékonysága összemérhető, és az előállított folyamatmodellek értékelhetők.

1.2. Feladat módszertani háttere, irodalom rövid értékelése

A kísérletek elvégzéséhez a Fraunhofer Institute folyamatbányászati feladatok megvalósításához összeállított PM4Py Python programkönyvtárát használtuk. A PM4Py könyvtárból a folyamatfeltáró algoritmusokat teszteltük.

A folyamatfeltáró algoritmusok bemenete egy eseménynapló, a kimenete pedig a feltárt folyamat leírása. A bemeneti eseménynaplófájlokat TXT-formátumban kaptuk meg, amelyeket át kellett konvertálni CSV-formátumra. Ezekre futtattuk le a 12 vizsgált algoritmust, amelyek különböző formátumban állítják elő a feltárt folyamatmodellt.

A vizsgált algoritmusok aszerint csoportosítva, hogy milyen folyamatmodellt állítanak elő:

Deklaratív modellt előállító algoritmusok:

- EFG- (Eventually-follows graph) gráfot előállító algoritmus
- Log skeleton (eseménynaplóvázat) előállító algoritmus

Grafikus modellt előállító algoritmusok:

- DFG- (Directly-follows graph) gráfot előállító algoritmus

- Petri-háló előállító Alpha Miner, Inductive Miner, Heuristic Miner, és ILP Miner algoritmusok
- Folyamatfát (process tree) előállító Inductive Miner algoritmus
- Heurisztikus háló (heuristics net) előállító Heuristic Miner algoritmus
- BPMN- (Business Process Model Notation) gráfot előállító Inductive Miner algoritmus
- Állapotátmeneti gráfot (transition system) előállító algoritmus
- Prefix fát előállító algoritmus

1.3. A kidolgozott kísérletek/modellezések módszertana

A benchmark folyamatmodellek előállításához és az azok értékelését lehető tevő mérések elvégzéséhez az alábbi lépések végrehajtására volt szükség:

1. Python értelmező, fejlesztőkörnyezet és a PM4Py programkönyvtár telepítése
2. Teszt adathalmazok előállítása
3. Teszt adathalmazok átalakítása CSV-eseménynapló formátumra
4. CSV-állományok betöltése, statisztika készítése (esetek száma, összes események száma, eseménytípusok száma)
5. A vizsgált algoritmusok lefuttatása a teszt adathalmazokra, a feltárt modellek elmentése
6. A futási eredmények kiértékelése, a feltárt modellek összehasonlítása

1.4. A módszer lépéseinek, a referencia folyamatoknak részletezése

A szintetikusan előállított TXT-fájlok sorai vesszővel elválasztott események sorozatát tartalmazzák. Egy eseményt a szimbolikus nevével (a tevékenység megnevezését helyettesítő betűjellel) adunk meg. A folyamatpéldány azonosítóját (case_id) és az időbélyeget automatikusan adjuk hozzá az átalakítás során.

A vizsgált folyamatfeltáró algoritmusok implementációja egységes a PM4Py programkönyvtárban. Ez azt jelenti, hogy minden függvény ugyanazokkal az argumentumokkal hívható. Mégpedig, a bemeneti eseménynaplót, az esetazonosítót (case_id) tartalmazó adatszlop nevét, a tevékenység/esemény megnevezését tartalmazó adatszlop nevét, valamint az időbélyeget tartalmazó adatszlop nevét kell megadnunk. A függvények visszatérési értéke az előállított folyamatmodell-leírás, amit grafikus modell esetén a GraphViz segítségével tudunk megjeleníteni, és a megfelelő PM4Py függvénnyel PNG-formátumban elmenteni. A deklaratív modelleket előállító algoritmusok kimenetét TXT-formátumban tudjuk eltárolni.

A PM4Py csomagban implementált folyamatfeltáró algoritmusok leírása itt található: <https://PM4Py.fit.fraunhofer.de/static/assets/api/2.7.11/api.html#process-discovery-PM4Py-discovery>

A vizualizációhoz felhasználható függvények leírása ezen a linken keresztül érhető el: <https://PM4Py.fit.fraunhofer.de/static/assets/api/2.7.11/api.html#visualization-PM4Py-vis>

1.5. Mérföldkő során megvalósított munkafolyamatok rövid, összegző összefoglalása

Az adott kutatási időszakban benchmark tesztek készítettünk, amelyek eredményeivel a munkacsoport által elkészített algoritmusok hatékonysága összemérhető, és az előállított folyamatmodellek értékelhetők.

A PM4Py programkönyvtárban implementált 12 folyamatfeltáró algoritmust lefuttattuk 19 szintetikus előállított teszt adathalmazra. Az eredményül kapott folyamatmodellek benchmark modellként használhatók. A modellek értékeléséhez az alábbi mérőszámokat számítottuk ki: illeszkedés, pontosság, általánosság és egyszerűség. Ezekhez a számításokhoz a PM4Py programkönyvtár kínál függvényeket, amelyek az alábbi argumentumokat várják: eseménynapló, Petri-háló-formátumban előállított folyamatmodell, a háló kiindulópontja, a háló végpontja.

A folyamatmodellek értékelhetősége érdekében végül csak azokat az algoritmusokat tartottuk meg a benchmark teszt halmazban, amelyek Petri-hálót állítanak elő és komplex szerkezetű (többszörös elágazást és ciklust is tartalmazó) eseménynaplók esetén is alkalmazhatók. Ezek a Heuristic Miner és az Inductive Miner algoritmusok.

2. Az elért eredmények bemutatása

2.1. A tesztek, kísérletek megvalósítási környezetének ismertetése

Az adott kutatási időszakban a Python folyamatbányászati feladatok megvalósításához összeállított PM4Py programkönyvtárát használtuk a kísérletek elvégzéséhez. A PM4Py a Fraunhofer Institute terméke, nyílt forráskódú, akadémiai és ipari projektekben egyaránt használható. A programcsomag dokumentációja a <https://PM4Py.fit.fraunhofer.de/> linken érhető el.

A PM4Py telepítéséhez és használatához legalább 3.9.x verziójú Python értelmező szükséges. A telepítés utasítása: `pip install -U PM4Py`. Mi 64 bites Windows operációs rendszeren, Python 3.10.6 értelmező használata mellett telepítettük a programcsomagot. A PM4Py függvényei egyéb Python csomagoktól függenek, eltérő fontossági szinteken:

- Alapvető követelmények: numpy, pandas, deprecation, networkx
- Normál követelmények (alapértelmezés szerint a PM4Py csomaggal együtt telepítve, fontos az általános használathoz): graphviz, intervaltree, lxml, matplotlib, pydotplus, pytz, scipy, stringdist, tqdm
- Opcionális követelmények (alapértelmezés szerint nincs telepítve): scikit-learn, pyemd, pyvis, jsonschema, polars, openai, pywin32, python-dateutil, requests, workalendar, pyngetputwindow, pyngetwindow

A PM4Py könyvtárból mi a folyamatfeltáró algoritmusokat használtuk fel a tesztekhez. A folyamatfeltáró algoritmusok bemenete egy eseménynapló, a kimenete pedig a feltárt folyamat leírása.

A bemeneti eseménynapló fájlokat TXT-formátumban kaptuk meg, ahol a fájl egy sora egy folyamatpéldány (trace) leírása. Ennek átalakítására írtunk Python programot. Ez a program a megadott könyvtárban tárolt TXT-fájlokat CSV- és XES-eseménynaplókká alakítja, amelyeket a PM4Py könyvtár folyamatfeltáró algoritmusaival dolgozunk fel.

A szintetikusan előállított TXT fájlok sorai vesszővel elválasztott események sorozatát tartalmazzák. Egy eseményt a szimbolikus nevével (a tevékenység megnevezését helyettesítő betűjellel) adunk meg. A folyamatpéldány azonosítóját (Case_id) és az időbélyeget automatikusan adjuk hozzá az átalakítás során.

A PM4Py könyvtárban elérhető algoritmusok különböző módon írják le a feltárt folyamatot. Mi az alábbi algoritmusokat vizsgáltuk.

Deklaratív modellt előállító algoritmusok:

- EFG- (Eventually-follows graph) gráfot előállító algoritmus
efg = **PM4Py.discover_eventually_follows_graph**(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
- Log skeletont (eseménynaplóvázat) előállító algoritmus
log_skeleton = **PM4Py.discover_log_skeleton**(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')

Grafikus modellt előállító algoritmusok:

- DFG- (Directly-follows graph) gráfot előállító algoritmus
dfg, start, end = **PM4Py.discovery.discover_dfg_typed**(log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
- Petri-hálót előállító algoritmusok (Alpha Miner, Inductive Miner, Heuristic Miner, ILP Miner)
net, im, fm = **PM4Py.discover_petri_net_alpha**(event_log, case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')

- ```

net, im, fm = PM4Py.discover_petri_net_inductive(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
net, im, fm = PM4Py.discover_petri_net_heuristics(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
net, im, fm = PM4Py.discover_petri_net_ilp(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')

```
- Folyamatfát (process tree) előállító Inductive Miner algoritmus  

```
process_tree = PM4Py.discover_process_tree_inductive(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
```
  - Heurisztikus hálót (heuristics net) előállító Heuristic Miner algoritmus  

```
heu_net = PM4Py.discover_heuristics_net(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
```
  - BPMN (Business Process Model Notation) gráfot előállító Inductive Miner algoritmus  

```
bpmn_graph = PM4Py.discover_bpmn_inductive(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
```
  - Állapotátmeneti gráfot (transition system) előállító algoritmus  

```
transition_system = PM4Py.discover_transition_system(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
```
  - Prefix fát előállító algoritmus  

```
prefix_tree = PM4Py.discover_prefix_tree(event_log,
case_id_key='case_id', activity_key='activity', timestamp_key='timestamp')
```

Az algoritmusok által előállított modellek értékelésekor 4 szempontot veszünk figyelembe:

- Illeszkedés: a feltárt modell mennyire illeszkedik az eseménynaplóban rögzített esetekre (azok hány százalékát fedi le). Vagyis az eseménynaplóban lévő esetek hány százaléka írható le a feltárt modellel.
- Pontosság: a feltárt modellből nem szabad olyan esetre következtetni, ami nem szerepel az eseménynaplóban. Azaz, ha a feltárt modellből legeneráljuk az összes lehetséges esetet, akkor ezek hány százaléka szerepel az eredeti eseménynaplóban.
- Általánosítás: a feltárt modell milyen mértékben általánosítja a valós eseteket.
- Egyszerűség: a feltárt modell mennyire egyszerű.

## 2.2. Teszt adathalmazok ismertetése

A teszt adathalmazokat a munkacsoport által készített, grafikus eseménynapló-generáló szoftver segítségével állítottuk elő TXT-formátumban, az alábbi szisztematikus szerkezeti felépítést követve:

- XOR elágazást tartalmazó szekvenciális folyamatok:
  - Egyszeres mélységű elágazásokat tartalmazó folyamatok (X) (3 fájl)
  - Kétszeres mélységű elágazásokat tartalmazó folyamatok (XX) (2 fájl)
  - Háromszoros mélységű elágazásokat tartalmazó folyamatok (XXX) (2 fájl)
- Ciklust tartalmazó folyamatok:
  - Egy cikluson belül egyszeres mélységű elágazásokat tartalmazó folyamatok (LX) (3 fájl)
  - Egy cikluson belül kétszeres mélységű elágazásokat tartalmazó folyamatok (L2X) (1 fájl)
  - Egyszeres mélységű elágazásokba ágyazott ciklust tartalmazó folyamatok (XL) (1 fájl)
  - Egy cikluson belül egyszeres mélységű elágazásokba ágyazott ciklust tartalmazó folyamatok (LXL) (1 fájl)
  - Egy cikluson belül egyszeres mélységű elágazásokba ágyazott két-szintű ciklusbeágyazást tartalmazó folyamatok (LXLL) (1 fájl)
  - Kétszintű ciklusbeágyazást tartalmazó folyamatok (LL) (1 fájl)
  - Kétszintű ciklusbeágyazáson belül egyszeres mélységű elágazást tartalmazó folyamatok (2LX) (1 fájl)
  - Négy szintű ciklusbeágyazáson belül egyszeres mélységű elágazást tartalmazó folyamatok (LLLLX) (1 fájl)
- Random folyamatok:
  - Azonos eseményszámú, ismétlődő tevékenységeket nem tartalmazó folyamatok (FRNOREP)
  - Azonos eseményszámú, ismétlődő tevékenységeket tartalmazó folyamatok
  - Eltérő eseményszámú, ismétlődő tevékenységeket nem tartalmazó folyamatok
  - Eltérő eseményszámú, ismétlődő tevékenységeket tartalmazó folyamatok (IR)

A teszt adathalmazok statisztikai adatait az alábbi táblázatban ismertetjük:

| Adathalmaz neve | Folyamat-példányok száma (sorok száma) | Összes események száma | Esemény-típusok száma |
|-----------------|----------------------------------------|------------------------|-----------------------|
| S_X_01          | 1000                                   | 5000                   | 7                     |
| S_X_02          | 1000                                   | 7000                   | 10                    |
| S_X_04          | 1000                                   | 11000                  | 16                    |
| S_XX_01         | 1000                                   | 3932                   | 7                     |
| S_XX_10         | 1000                                   | 4534                   | 6                     |
| S_XXX_01        | 1000                                   | 4026                   | 9                     |
| S_XXX_10        | 1000                                   | 3911                   | 9                     |
| S_LX_01         | 1000                                   | 6576                   | 4                     |
| S_LX_02         | 1000                                   | 8378                   | 4                     |
| S_LX_05         | 1000                                   | 10922                  | 8                     |
| S_L2X_01        | 1000                                   | 24725                  | 9                     |
| S_XL_01         | 1000                                   | 4834                   | 5                     |
| S_LXL_01        | 1000                                   | 23879                  | 8                     |
| S_LXLL_02       | 1000                                   | 50798                  | 9                     |
| S_LL_02         | 1000                                   | 38167                  | 5                     |
| S_2LX_02        | 1000                                   | 25469                  | 10                    |
| S_LLLLX_01      | 1000                                   | 237032                 | 12                    |
| S_FRNOREP       | 100                                    | 500                    | 14                    |
| S_IR            | 5                                      | 54                     | 5                     |

### 2.3. Kiértékelések eredményeinek bemutatása

Az S\_IR\_01 random generált mintaadathalmaz a legegyszerűbb szerkezetű és legrövidebb adathalmaz, ezért ezen mutatom be, hogyan kell értelmezni az egyes folyamatleíró modellek eredményét.

Az adathalmaz 5 esetet (folyamatpéldányt) tartalmaz, amelyekben 5 különféle tevékenység szerepel.

```
a,b,e,d,e
a,b,e,d,c,e,d,c,e,d,e
a,b,e,d,c,b,e,d,e
a,b,e,d,c,b,e,d,c,b,e,d,e,
a,b,e,d,c,b,e,d,c,e,d,c,b,e,d,e
```

A PM4Py függvényeivel az alábbi, szöveges formátumban előállított EFG-gráffal és eseménynaplóvázzal (Log skeleton) írható le az adathalmazban tárolt folyamatpéldányokból feltárt folyamat.

EFG-gráf:

```
{('a', 'b'): 10, ('a', 'e'): 18, ('a', 'd'): 13, ('a', 'c'): 8, ('b', 'e'): 31, ('b', 'd'): 21, ('b', 'c'): 11, ('b', 'b'): 7, ('e', 'd'): 26, ('e', 'c'): 13, ('e', 'b'): 8, ('e', 'e'): 26, ('d', 'c'): 13, ('d', 'b'): 8, ('d', 'e'): 26, ('d', 'd'): 13, ('c', 'b'): 8, ('c', 'e'): 21, ('c', 'd'): 13, ('c', 'c'): 5}
```

Ebben a leírásban azt látjuk, hogy az eseménynaplóban hányszor fordul elő, hogy egy  $x$  eseményt egy  $y$  követ. Az EFG-gráfban nemcsak a közvetlen egymásutániséget számoljuk, de az események sorrendje lényeges.

Log skeleton:

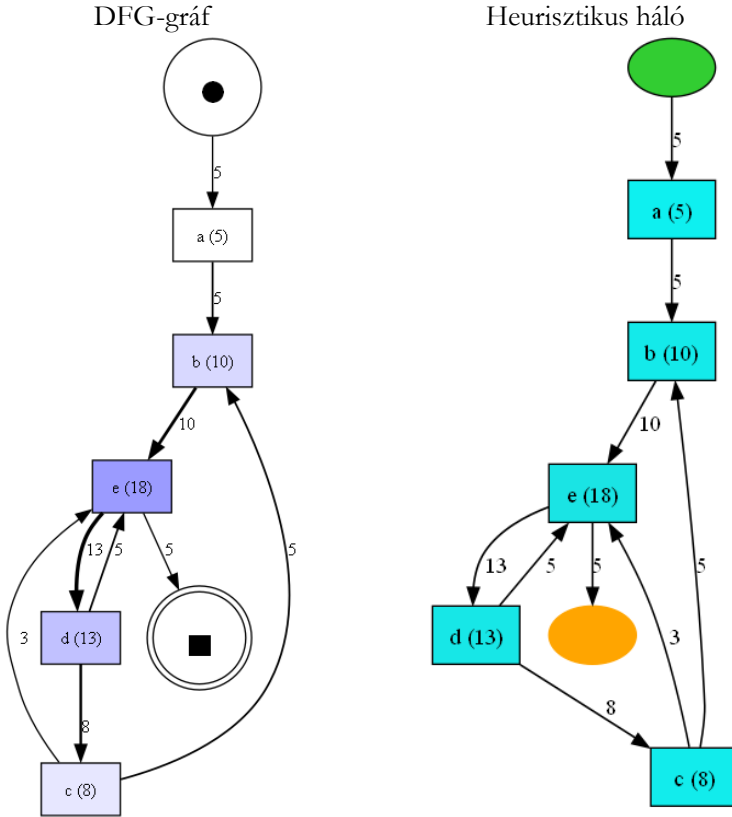
```
{'equivalence': set(),
'always_after': {('c', 'd'), ('b', 'c'), ('d', 'd'), ('c', 'e'), ('b', 'd'), ('a', 'e'), ('d', 'e'), ('e', 'd'), ('c', 'b'), ('a', 'c'), ('b', 'e'), ('a', 'b'), ('e', 'e'), ('d', 'c'), ('a', 'd')},
'always_before': {('c', 'd'), ('e', 'c'), ('d', 'a'), ('d', 'd'), ('c', 'e'), ('b', 'a'), ('d', 'e'), ('e', 'a'), ('c', 'b'), ('e', 'd'), ('e', 'e'), ('d', 'c'), ('c', 'a'), ('d', 'b'), ('e', 'b')},
'never_together': set(),
'directly_follows': {('b', 'e'), ('a', 'b')},
'activ_freq': {'a': {1}, 'b': {1, 2, 3}, 'e': {2, 3, 4, 5}, 'd': {1, 2, 3, 4}, 'c': {0, 1, 2, 3}}}
```

Az eseménynaplóból készített naplóváz az események egymásutániségát gyűjti össze a fent látható csoportosításban:

- **Equivalence:** a két esemény (tevékenység) sorrendje felcserélhető.
- **Always\_after:** azon eseménypárok, amelyekre igaz, hogy az első eseményt követi a második.
- **Always\_before:** azon eseménypárok, amelyekre igaz, hogy az első eseményt megelőzi a második.
- **Never\_together:** olyan eseménypárok, amelyek egyik folyamatpéldányban sem fordulnak elő egyidejűleg.
- **Directly\_follows:** olyan eseménypárok, amelyekre igaz, hogy az első eseményt közvetlenül követi a második.
- **Activ\_freq:** az adott esemény (tevékenység) előfordulási gyakoriságai a folyamatpéldányokban.

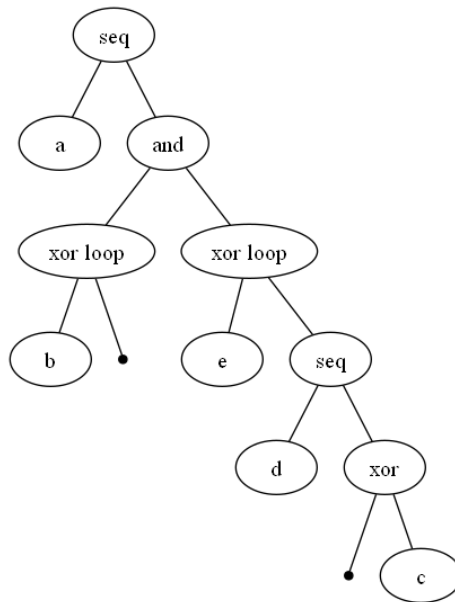
A deklaratív modellek tulajdonképpen az események egymást követésének leszámítását tartalmazzák, tehát nem általánosítanak és ezért nem a lehető legegyszerűbbek.

A grafikus modelleket előállító algoritmusok közül a DFG-gráfot és a heurisztikus hálót előállító algoritmust nézzük meg először, mert ezek az ábrák értelmezhetőek legkönnyebben a deklaratív leírások alapján.



Mindkét algoritmusnak sikerült feltárnia a folyamat kezdő- és végpontját (az első és az utolsó eseményt), és ugyanazt a folyamatmodellt állították elő az események egymást követési gyakoriságai alapján. Mindkét algoritmusra igaz, hogy nem általánosít eléggé, mert az összes előforduló eseménysorrendet ábrázolja, ami nagy méretű és komplex szerkezetű eseménynapló feltárásakor bonyolult modellt eredményez.





A folyamatfa és a BPMN olyan modellek, amelyek az eseményeken kívül a felismert vezérlési szerkezet kiinduló csomópontját is ábrázolják. Ezek a modellek már tartalmazznak általánosítást és van értelme vizsgálni, hogy mennyire egyszerűek. Ugyanezen okból viszont nehezebb nyomon követni a feltárt modellt a minta alapján.

A folyamatfát előállító algoritmus az alábbi szerkezeteket képes feltárni:

- Seq: szekvenciális végrehajtás
- And: párhuzamos ágak összefűzése (join)
- Xor: elágazás egyik ágának végrehajtása (kizáró vagy)
- Xor loop: ciklus (a folyamat egyik ágának ismétlése). Az ismétlődő szakasz végét a fekete pont jelöli.

A modellből ez olvasható ki:

A folyamat első eseménye az 'a'. Ezután 'b' következik (seq) legalább egyszer (xor loop), és utána (and) legalább egyszer (xor loop) 'e', majd (seq) 'd' és utána visszatérünk a ciklus elejére vagy (xor) 'c'.

Formálisan megfogalmazva:  $a \rightarrow b^* \rightarrow [e \rightarrow d \rightarrow (c \text{ vagy Vége})]^*$

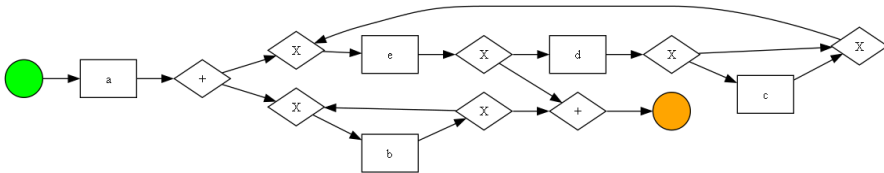


A valóságban az adathalmaz eseményeinek ez a rögzített sorrendje:  
 $a \rightarrow b$  ;  $b \rightarrow e$  ;  $e \rightarrow d$  vagy  $Vége$  ;  $d \rightarrow e$  vagy  $c$  ;  $c \rightarrow b$  vagy  $e$

A modell általánosításra és egyszerűsége törekszik, ezért nem tökéletesen illeszkedik az adathalmazra, mert például azt nem írja le, hogy a minta alapján minden eset az 'e' eseménnyel zárul. Nem is pontos, mert sok olyan eseménysor generálható a modell alapján, ami valójában nem szerepel a mintában. Például: a, b, b, b, e, d.

Az Inductive Miner algoritmus által előállított BPMN-folyamatmodell ugyanezeket a vezérlési szerkezeteket képes megmutatni. A folyamat kezdő- és végállapotát kiemeli, a szekvenciát nem jelöli külön, az elágazásoknak viszont az elejét és a végét is jelöli. A '+' az AND, míg az 'x' az XOR megfelelője. A ciklust olyan XOR elágazásként ábrázolja, ahol a végállapotból visszatérünk a kezdő állapotba.

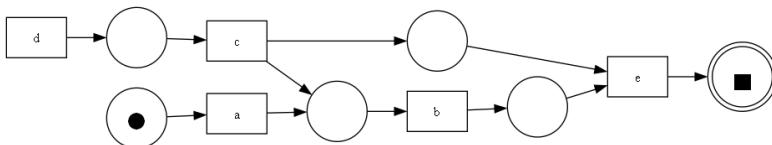
A random mintahalmazból feltárt BPMN-folyamatmodell formális leírása:  
 $a \rightarrow [b^* \rightarrow [e \rightarrow [d \rightarrow c \text{ vagy vissza} ] \text{ vagy } Vége]^*]$



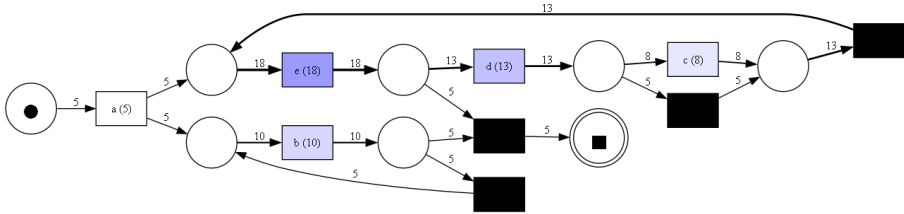
Ez a modell is elég általános és leegyszerűsített (tömör), de jobban illeszkedik a mintaadathalmazra, mint a folyamatfa, mert csak az olyan folyamatokat írja le, amelyek az 'a' eseménnyel kezdődnek és az 'e' eseménnyel végződnek. Az általánosság miatt ez sem tökéletes pontosságú, mert több olyan eset írható le, ami a mintaadathalmazban nem szerepel.

A vizsgálatunk szempontjából érdekesek még azok az algoritmusok, amik egy-egy esetben Petri-hálót állítanak elő kimenetként, ezzel elősegítve az algoritmusok összehasonlíthatóságát.

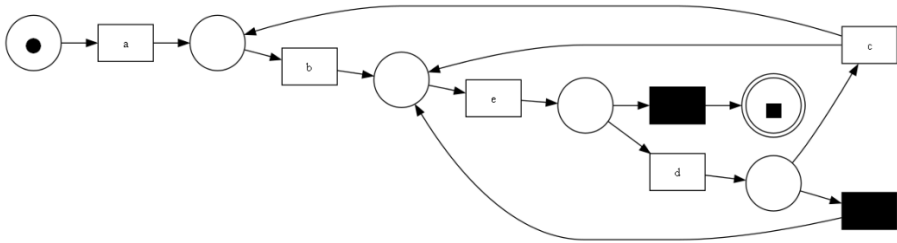
Az Alpha Miner algoritmus által előállított Petri-háló így néz ki:



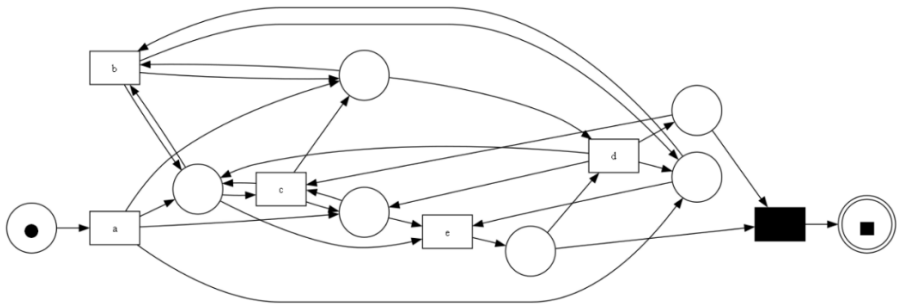
Az Inductive Miner algoritmus által előállított Petri-háló esetén az események előfordulási gyakoriságai is megjeleníthetők. Látható, hogy az algoritmus által készített BPMN-modell és Petri-háló szerkezete megegyezik.



A Heuristic Miner algoritmus által előállított Petri-háló szerkezete hasonlít a heurisztikus háléhoz:



Az ILP Miner algoritmus által előállított Petri-háló a következő:



Az előállított folyamatmodelleket összehasonlítva elmondható, hogy az ILP Miner algoritmus nem általánosít és nem egyszerűsít megfelelően, az összes előforduló esetet magában foglalja.

Mivel a többi teszt adathalmaz nagy méretű, a továbbiakban csak azon algoritmusok által feltárt modelleket mutatjuk be, amelyek tudnak általánosítani, és

az összehasonlíthatóság miatt csak a Petri-háló modelleket. Azaz az Alpha Miner, az Inductive Miner és a Heuristic Miner algoritmusok kimenetét használjuk benchmark eredményként. Ezen modellekhez fogjuk hasonlítani a saját algoritmusaink által előállított folyamatmodelleket.

Az Inductive Miner algoritmus által feltárt modellek értékeléséhez a PM4Py programkönyvtár biztosít segédfüggvényeket. Ezeket felhasználva a következő adatokat kaptuk az értékelési szempontjainkhoz.

Illeszkedés: 1,0

A modell 100%-ban illeszkedik az adathalmazra, azaz a minta összes esete leírható.

Pontosság: 0,71

A modell 71%-os pontosságú, azaz a modell visszajátszásakor a generált esetek 29%-a nincs a minta adathalmazban.

Általánosítás: 0,64

Egyszerűség: 0,74

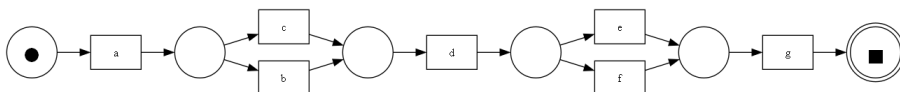
Ezek az eredmények azt mutatják, hogy a random generált, alacsony esetszámú eseménynaplóból feltárt folyamatmodell jól illeszkedik a mintára, nem teljesen pontos és nem elég általános.

#### 2.4. Eredményeket szemléltető képernyőképek, adatsorok

A munkacsoportunk által kidolgozott neurális háló és véges automata alapú algoritmusok értékeléséhez az alábbi benchmark teszt eredmények használhatók.

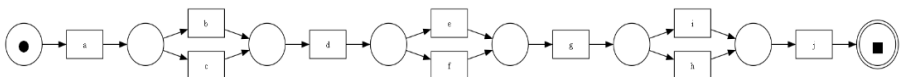
##### S\_X\_01 teszt adathalmaz

Erre az adathalmazra az Alpha Miner, az Inductive Miner és a Heuristic Miner is az alábbi folyamatmodellt generálta.



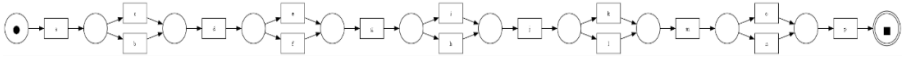
##### S\_X\_02 teszt adathalmaz

Erre az adathalmazra az Alpha Miner, az Inductive Miner és a Heuristic Miner is az alábbi folyamatmodellt generálta.

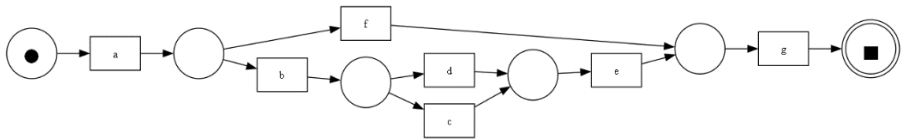


S\_X 04 teszt adathalmaz

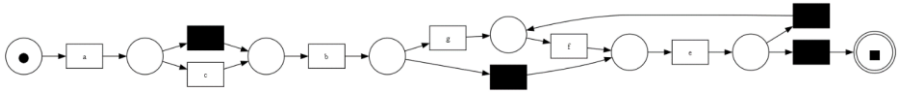
Erre az adathalmazra az Alpha Miner, az Inductive Miner és a Heuristic Miner is az alábbi folyamatmodellt generálta.

S\_XX 01 teszt adathalmaz

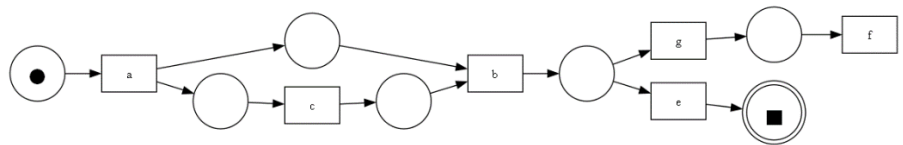
Erre az adathalmazra az Alpha Miner, az Inductive Miner és a Heuristic Miner is az alábbi folyamatmodellt generálta.

S\_XX 10 teszt adathalmaz

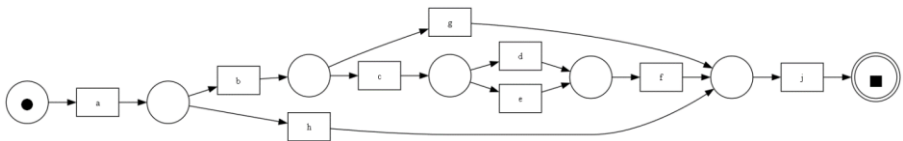
Ehhez az adathalmazhoz a Heuristic Miner és az Inductive Miner ezt a modellt állította elő.



Az Alpha Miner algoritmus viszont egy egyszerűbb modellt készített.

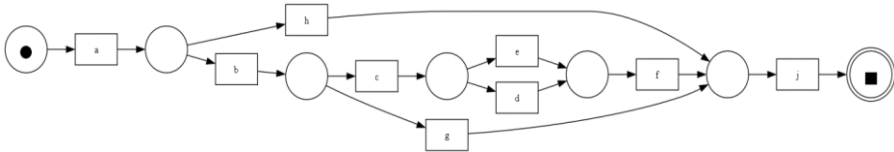
S\_XXX 01 teszt adathalmaz

Erre az adathalmazra is mindhárom algoritmus ugyanazt a modellt generálta.



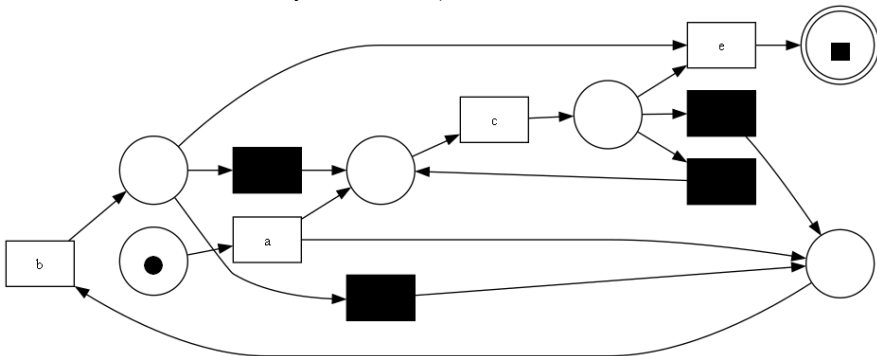
S\_XXX\_10 teszt adathalmaz

Erre az adathalmazra mindhárom algoritmus ugyanazt a modellt állította elő, ami ráadásul megegyezik az előző, hasonló bonyolultságú és esetszámú eseménynapló modelljével.

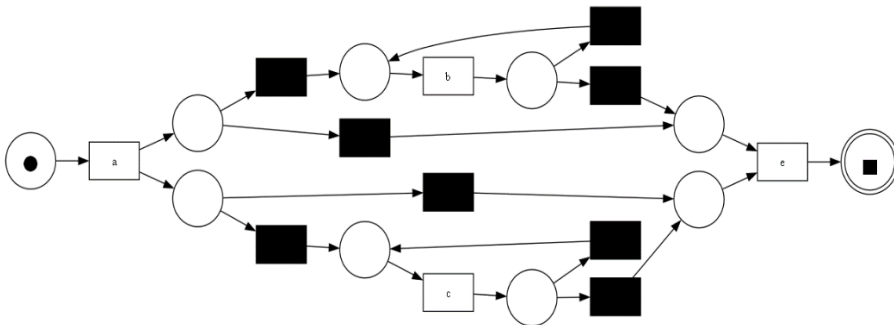


S\_LX\_01 teszt adathalmaz

Az Alpha Miner nem tudott ehhez az adathalmazhoz használható modellt készíteni. A Heuristic Miner folyamatmodellje az alábbi:



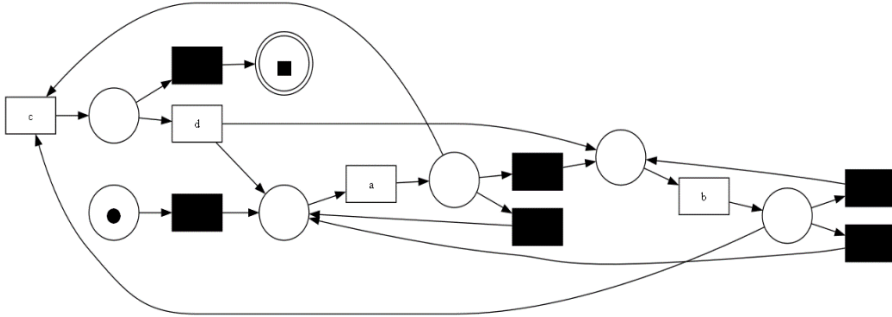
Az Inductive Miner által előállított modell pedig így néz ki:



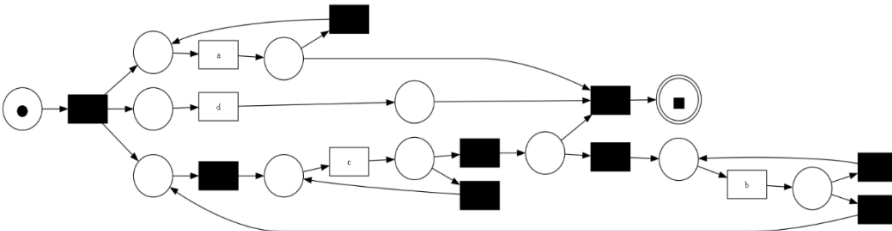
S\_LX\_02 teszt adathalmaz

Az Alpha Miner nem tudott ehhez az adathalmazhoz használható modellt készíteni.

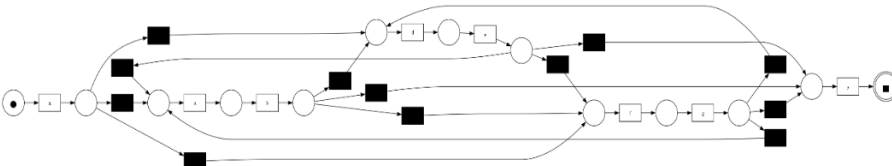
A Heuristic Miner folyamatmodellje az alábbi:



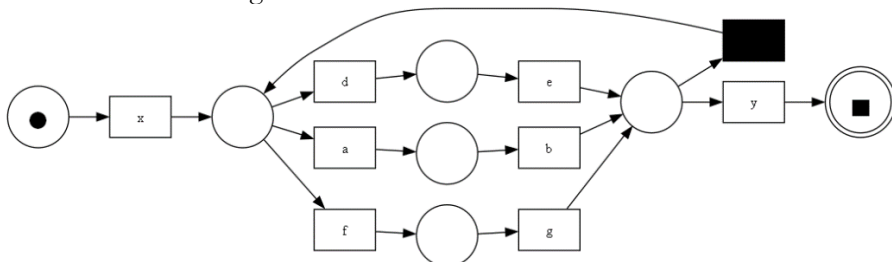
Az Inductive Miner által előállított modell:

S\_LX\_05 teszt adathalmaz

Az Alpha Miner nem tudott ehhez az adathalmazhoz használható modellt készíteni. A Heuristic Miner folyamatmodellje így néz ki:

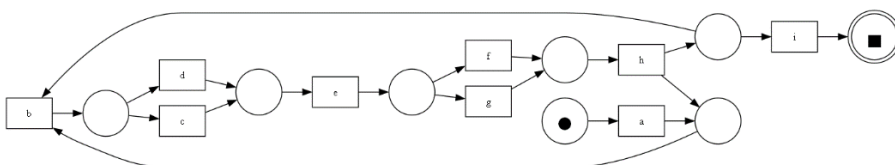


Az Inductive Miner algoritmus kimenete:

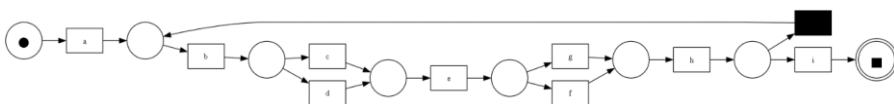


S\_L2X\_01 teszt adathalmaz

Erre az adathalmazra az Alpha Miner által adott modell az alábbi, amin látszik, hogy az algoritmus nem jól kezeli a ciklust:

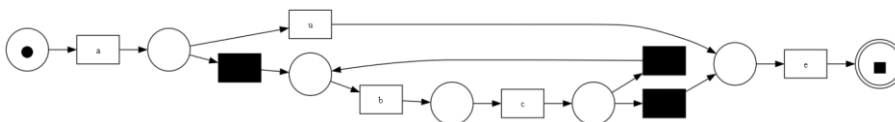


A Heuristic Miner és az Inductive Miner ugyanazt a modellt adja:



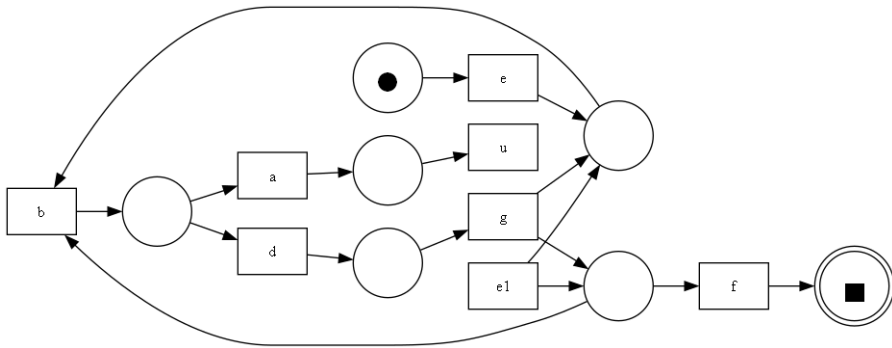
S\_XL\_01teszt adathalmaz

Erre az adathalmazra az Alpha Miner nem ad használható modellt. A Heuristic Miner és az Inductive Miner viszont ugyanazt a folyamatmodellt tárja fel:

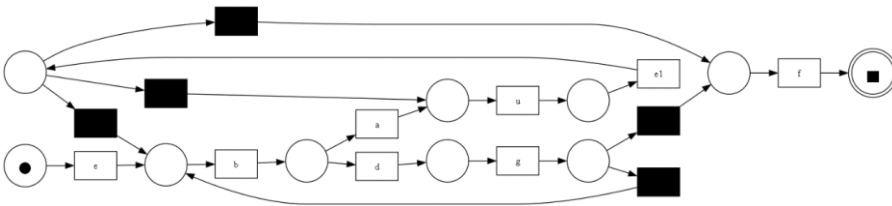


S\_LXL\_01 teszt adathalmaz

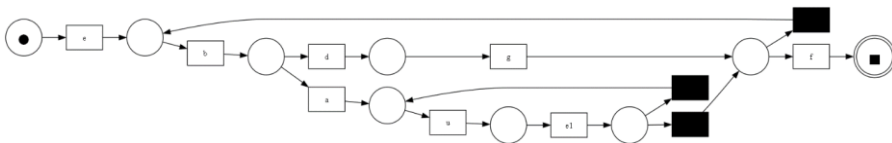
Az Alpha Miner modellje erre az adathalmazra az alábbi, amin látszik, hogy az algoritmus nem ismerte fel az eseménysorokban elrejtett beágyazott szerkezetet.



A Heuristic Miner algoritmus kimenete:

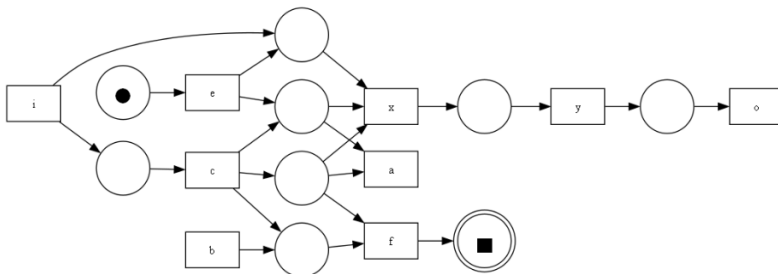


Az Inductive Miner modellje:



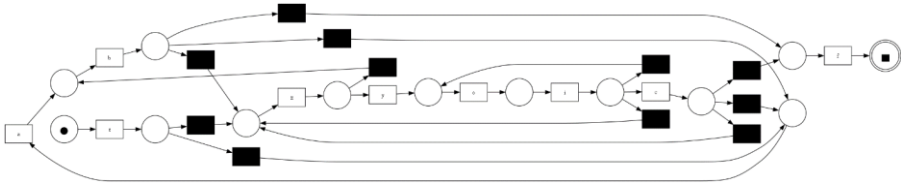
S\_LXLL\_02 teszt adathalmaz

Az Alpha Miner modellje erre az adathalmazra az alábbi, amin látszik, hogy az algoritmus nem ismerte fel az eseménysorokban elrejtett beágyazott szerkezetet.

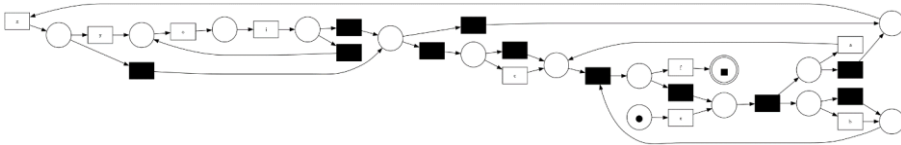




A Heuristic Miner folyamatmodellje:

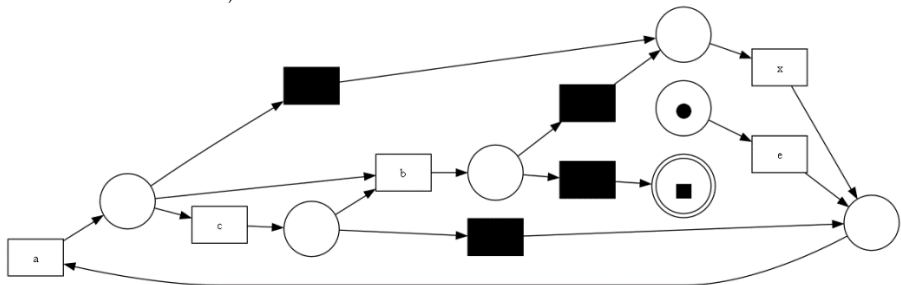


Az Inductive Miner folyamatmodellje:

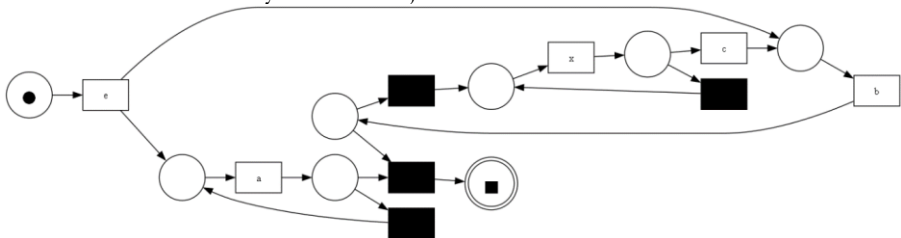


S LL 02 teszt adathalmaz

Az Alpha Miner erre az adathalmazra sem ad használható folyamatmodellt. A Heuristic Miner modellje az alábbi:

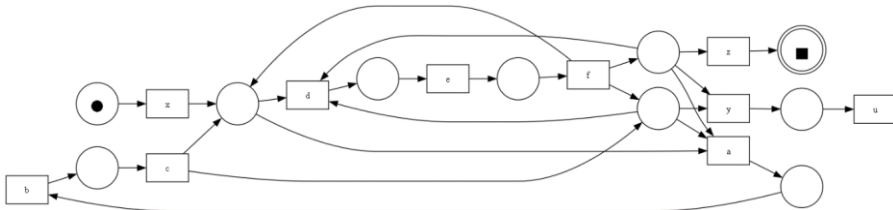


Az Inductive Miner folyamatmodellje:

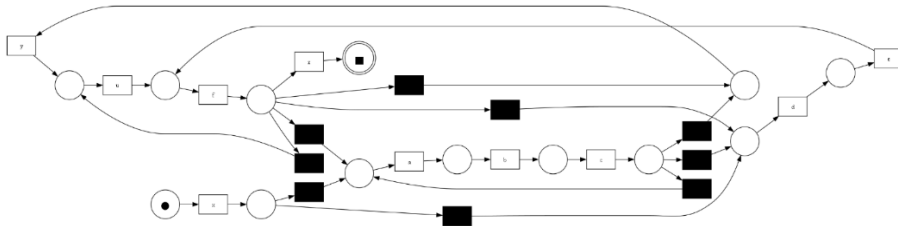


S 2LX 02 teszt adathalmaz

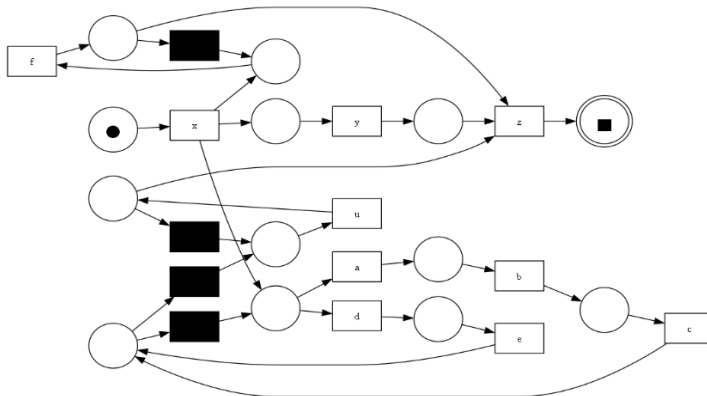
Az Alpha Miner folyamatmodellje nem tükrözi az egymásba ágyazott ciklusokban lévő elágazás szerkezetet.



A Heuristic Miner modellje:

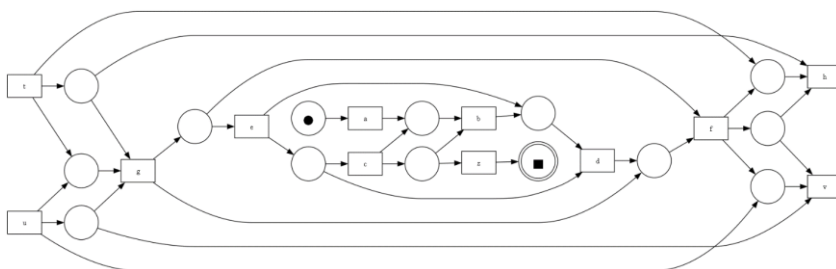


Az Inductive Miner folyamatmodellje:

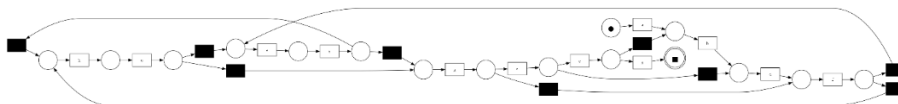


S LLLX 01 teszt adathalmaz

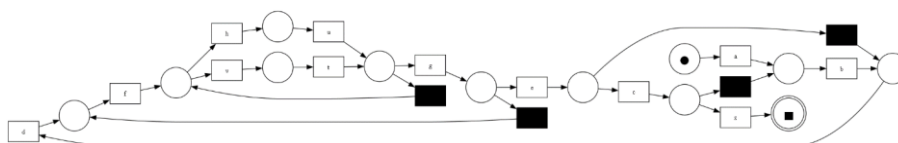
Az Alpha Miner az alábbi modellt adja a négyszeresen beágyazott ciklust tartalmazó folyamatokhoz:



A Heuristic Miner folyamatmodellje:

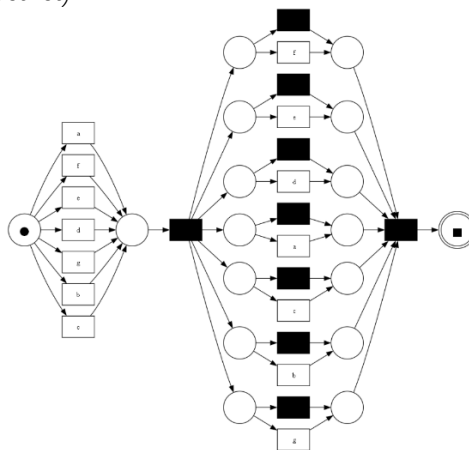


Az Inductive Miner folyamatmodellje:



S ERNOREP teszt adathalmaz

Erre a random adathalmazra az Alpha Miner és a Heuristic Miner is hasonló, áttekinthetetlen folyamatmodellt produkált. Az Inductive Miner modellje áttekinthető, és jól látható rajta az adathalmaz véletlenszerűsége (bármely eseményt bármely másik követhet).



Az Inductive Miner algoritmus által előállított folyamatmodellek értékelését a táblázat tartalmazza.

| Adathalmaz | Illeszkedés | Pontosság   | Általánosítás | Egyszerűség |
|------------|-------------|-------------|---------------|-------------|
| S_X_01     | 1,0         | 1,0         | 0,96          | 0,87        |
| S_X_02     | 1,0         | 1,0         | 0,96          | 0,82        |
| S_X_04     | 1,0         | 1,0         | 0,96          | 0,78        |
| S_XX_01    | 1,0         | 1,0         | 0,95          | 0,87        |
| S_XX_10    | 1,0         | <b>0,83</b> | 0,96          | 0,77        |
| S_XXX_01   | 1,0         | 1,0         | 0,94          | 0,80        |
| S_XXX_10   | 1,0         | 1,0         | 0,94          | 0,80        |
| S_LX_01    | 1,0         | 0,93        | 0,96          | 0,73        |
| S_LX_02    | 1,0         | <b>0,52</b> | 0,97          | 0,71        |
| S_LX_05    | 1,0         | 0,91        | 0,97          | 0,80        |
| S_L2X_01   | 1,0         | 0,94        | 0,98          | 0,82        |
| S_XL_01    | 1,0         | 0,99        | 0,96          | 0,88        |
| S_LXL_01   | 1,0         | 0,91        | 0,98          | 0,83        |
| S_LXLL_02  | 1,0         | <b>0,45</b> | 0,98          | 0,71        |
| S_LL_02    | 1,0         | <b>0,64</b> | 0,98          | 0,74        |
| S_2LX_02   | 1,0         | <b>0,54</b> | 0,98          | 0,73        |
| S_LLLLX_01 | Nincs adat  | Nincs adat  | Nincs adat    | Nincs adat  |
| S_FRNOREP  | 1,0         | 0,80        | 0,82          | 0,53        |
| S_IR       | 1,0         | 0,71        | 0,64          | 0,74        |

A modellek kiértékelése alapján elmondható, hogy az Inductive Miner algoritmus az adathalmazra illeszkedő folyamatmodelleket állít elő, amelyeknek az általánossága és egyszerűsége között igyekszik egyensúlyt tartani. Ennek sokszor a pontosság csökkenése az ára, azaz a modell alapján olyan esetek is előállíthatók, amelyekre nincs példa az adathalmazban.

A teljesen véletlenszerű eseménysorokra az algoritmus nem ad jó eredményeket sem pontosság, sem általánosság, sem egyszerűség tekintetében.

A nagy bonyolultságú folyamatok modellezésekor korlátokba ütközünk mind a grafikus modell ábrázolása, mind pedig az értékelés mérőszámainak kiszámítása kapcsán.

## **2.5. Kutatási eredményeken alapuló javaslatok megfogalmazása**

A kutatási időszakban a feladatunk az volt, hogy a benchmark teszt adathalmazokra lefuttassuk a PM4Py programkönyvtárban implementált folyamatfeltáró algoritmusokat és értékeljük az előállított folyamatmodelleket.

A vizsgálat elején 12 algoritmust próbáltunk ki, amelyek közül több nem végzi el a folyamatmodell megfelelő mértékű általánosítását, ezért nagy méretű eseménynaplók esetén a gyakorlatban nem alkalmazható. Ezt a szempontot figyelembe véve, végül három algoritmust alkalmaztunk a teszt adathalmazokra: az Alpha Miner, a Heuristic Miner és az Inductive Miner algoritmusokat.

A futási eredmények igazolják, hogy az Alpha Miner algoritmus a ciklust is tartalmazó, komplex szerkezetű eseménysorokat nem tudja megfelelően feldolgozni. Emiatt nem fogjuk tudni a benchmark összehasonlításoknál használni. A Heuristic Miner és az Inductive Miner hasonló bonyolultságú folyamatmodelleket generál. Az Inductive Miner algoritmus alkalmazása mellett szól, hogy BPMN- és Petri-háló modellt is tud generálni, és ki tudjuk számítani a modell értékeléséhez szükséges mérőszámokat. Ezek a jellemzők alkalmassá teszik az algoritmust arra, hogy benchmarkként szolgáljon a munkacsoport által kidolgozott algoritmusok futási eredményeinek értékelésekor.

# AUTOMATIKUS TESZTELÉSI KÖRNYEZET KIDOLGOZÁSA, A LÉTREHOZOTT MODELLEK ÖSSZEHASONLÍTÁSA ÉRDEKÉBEN

RADELECZKI SÁNDOR

*Jelen célkitűzéseket az motiválta, hogy ellenőrizni lehessen az ERPA-csoport által kidolgozott folyamatfeltárási módszerek hatékonyságát, ezeket össze tudjuk egymással hasonlítani és megvizsgáljuk, hogy hogyan lehet őket kiegészíteni/tökéletesíteni, valamint az eredményeket felhasználóbarát módon vizuálisan ábrázolni.*

## **1. Elvégzett kísérletek/modellezések (releváns szakasz) bemutatása**

Az elmúlt évben végzett kutatás az ERPA-projektben megadott kutatási célok közül az alábbiakhoz kapcsolódik:

A kutatóknak modellezniük kell az AIM-modul alapját képező, mesterséges intelligenciával is támogatott, tanuló algoritmusokat. A kutatóknak ki kell fejleszteniük az NN alapú továbbfejlesztett algoritmust és az AIM mesterséges intelligenciával támogatott tanuló modult.

### ***1.1. Feladat célkitűzése, motiváció, elvégzendő szakmai feladat bemutatása***

**Az én jelenlegi feladatom** a gráf/hálózat alapú folyamatfeltárás területén az alábbi volt:

Kutatási feladatok ellátása a gráf/háló és automata alapú folyamatfeltárás és folyamatvalidáció területen.

A kidolgozott automata alapú módszer adaptálása a generált tesztek befogadására. Az automatikus tesztelési környezet kialakítása és ellenőrzése, tesztelesek végzése.

Az automata alapú módszer eredményeinek összehasonlítása az ERPA-csoport által kidolgozott más elveken alapuló (pl. NN) módszerek eredményeivel.

A kialakított automatikus tesztelési környezetben összehasonlító és problémafeltáró tesztelesek végzése és dokumentálása, az algoritmus tökéletesítése.

Rendszeres beszámolás az elvégzett feladatokról és tevékenységéről.

Ezeket a célkitűzéseket az motiválta, hogy ellenőrizni lehessen az ERPA-csoport által kidolgozott folyamatfeltárási módszerek hatékonyságát, ezeket össze tudjuk egymással hasonlítani és megvizsgáljuk, hogy hogyan lehet őket kiegészíteni/tökéletesíteni, valamint az eredményeket felhasználóbarát módon vizuálisan ábrázolni.

## **1.2. Feladat módszertani háttere, irodalom rövid értékelése**

Ezeknek a céloknak a megvalósításához több módszert is alkalmaznunk kellett: Eljárást dolgoztunk ki egy folyamatgráf duálisának a létrehozására. Tesztelésein során felhasználtuk az eseménynaplók statikus és dinamikus tesztelési módszerét, amit az előző évben dolgoztunk ki. Az egyes eseménynaplók futtatási eredményeinek a vizsgálatára felhasználtuk az ún. keresztvalidációs módszert. Módszert dolgoztunk ki a folyamatgráf nevezetes elemeinek, vagyis többszörös vagy egymásba ágyazott LOOP, XOR, AND elemek vizsgálatára, reguláris kifejezések bonyolultságalapú osztályozására, valamint arra, hogy egy mintának egy reguláris kifejezést feleltessünk meg. Módszert fogalmaztunk meg a módosított MPM-algoritmusnak a Chat-GPT hozzáférhető változatával való kiegészítésére, összekapcsolására is.

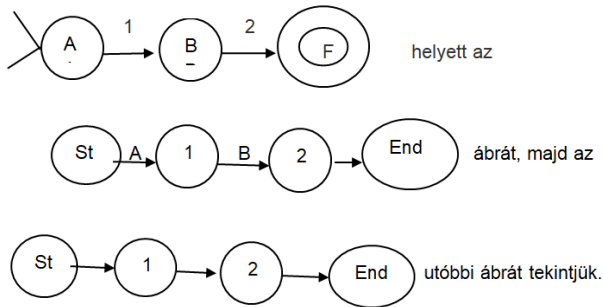
## **1.3. A kidolgozott kísérletek/modellezések módszertana**

- (a) A kapott automata gráfokat a felhasználó számára jól áttekinthető, vizuális formába kellett hoznunk, ami lényegében a folyamatgráf duálisának az áttekinthető megrajzolását jelentette.
- (b) A kapott gráfok egyszerűsítése, az eljárás tesztelése és tökéletesítése a minták elemeinek (LOOP, XOR, AND) a vizsgálata révén.
- (c) Az ERPA-csoport által létrehozott könyvtár segítségével a kapott mintákat összevetettük a csoport által kidolgozott más módszerek által kapott mintákkal.
- (d) Keresztvalidációs módszert alkalmazva kétfajta teszt sorozatot: ún. dinamikus és statikus tesztet is végeztünk abból a célból, hogy a módosított MPM-algoritmus hatékonyságát megvizsgáljuk.
- (e) Tökéletesítettük a kiindulási, vagyis az automataalapú algoritmust és megvizsgáltuk azt, hogy a kapott eredmények jóságát hogyan befolyásolja a nyomok, illetve a kapott minták bonyolultsága, ehhez bevezettünk egy egyszerű módszert a nyomok bonyolultságalapú osztályozására.
- (f) A megmaradt hibák struktúrájának az elemzése, az ezzel kapcsolatos tesztek kiértékelése. Itt konkrétan a többszörös LOOP és XOR elemeket, illetve egymásba ágyazott, többszörösen összetett folyamatábra-elemeket vizsgáltunk.
- (g) Javaslatot fogalmaztunk meg a módosított MPM-algoritmus NN-alapú kiegészítésére, illetve, hogy erre a célra hogyan használható fel a Chat-GPT elérhető változata. Ennek a módszertana a kinyert és az ERPA-könyvtárban talált maximális mintáknak a Chat-GPT segítségével való összehasonlításán alapszik.
- (h) Teszteltük a Chat-GPT-vel összekapcsolt módszer hatékonyságát felhasználva a könyvtárunkban tárolt eseménynaplókat.

Ezeknek a módszereknek a részleteit alább ismertetjük:

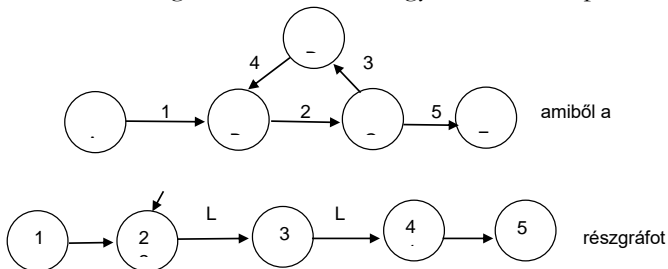
### 1.4. A módszerek lépéseinek, a referenciafolyamatoknak részletezése a módszert bemutató folyamatábrák, diagramok segítségével

- (a) Az eljárás első lépéseként az egyes részgráfok duálisait hozzuk létre, ami azt jelenti, hogy az események (amiket pozitív egész számok jelöltek), amik eddig az irányított élek címkei voltak, az új gráf csúcspontjai lesznek, az új gráf élei pedig az eddigi csúcspontoknak felelnek majd meg – ezeket kezdetben nagybetűkkel címkézzük meg, majd ezeket a címkéket, néhány a továbbiakban részletezett részlet kivételével töröljük. Ilyen lesz például a kezdő és a végállapotra vonatkozó címke, amit Starttal vagy röviden St-vel, illetve Enddel jelölünk és a csomópont belsejébe írunk. Például, az alábbi folyamatgráf:



- (b) Ott, ahol ez a fenti séma valamelyest módosulhat, tehát a speciális szakaszokat, például hurkokat, alternatív elágazásokat és párhuzamosságokat tartalmazó részgráfok, alább részletezzük:

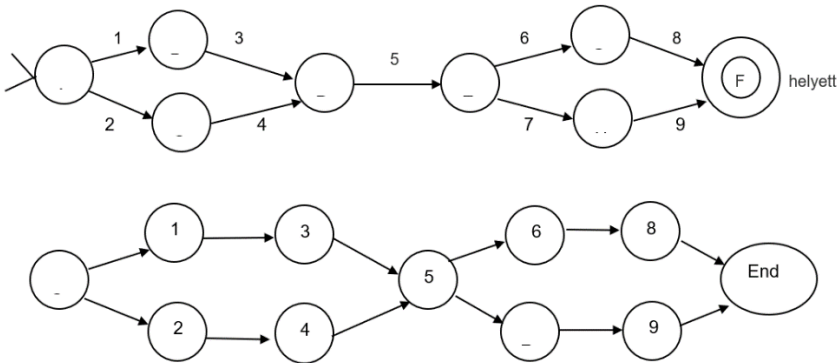
**Hurkok.** Kicsit módosul a gráf egy 1-nél hosszabb hurok, tehát egy 1-nél hosszabb ismétlődő eseményszakasz esetén, ahol a hurokba magába is csúcspontokat szükséges illeszteni, ahogy azt az alábbi példa szemlélteti:



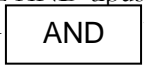
kapjuk, ahol az L betű a hurok éleit jelöli.



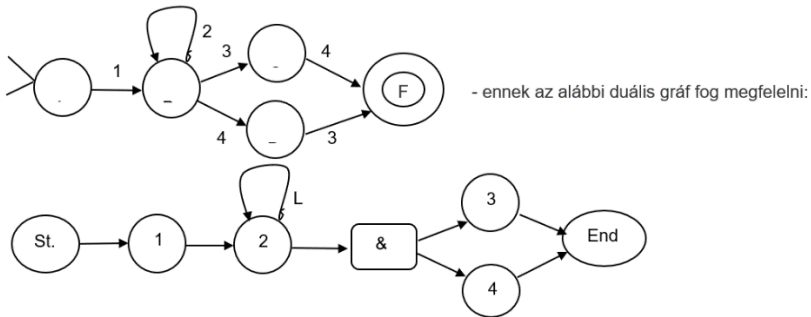
**XOR elágazás.** Ennél a lépésnél csak arra kell ügyelnünk, hogy amennyiben a XOR elágazás a kezdőállapotból indul, vagy egy végállapotban végződik, akkor egy-egy közös új állapottal kell nyitnunk, illetve zárnunk az alternatív útszakaszt, úgy, ahogy azt az alábbi példa mutatja:



**AND elágazás.** Ez az egyetlen olyan részlet a tranzíciós gráfban, ahol a duális gráf létrehozásakor nem automatikusan a bemutatott alapelv szerint járunk el, hanem külön jelölést alkalmazunk. Ennek oka az, hogy a felhasználó számára nem látható azonnal a duális gráfban a XOR és az AND típusú elágazás közötti eltérés. Ezért külön jelölést vezetünk be, ami **AND** vagy



Ha egy eseménycsoport tagjaiba mutató nyilak egy ilyen téglalapról indulnak ki, akkor ez azt jelzi, hogy az egyes események párhuzamosak. Az ábra alapján megállapítható, hogy a 3 és 4 események tetszés szerinti sorrendben elvégezhetők, ezért 3, 4 és 4, 3 valójában párhuzamos szakaszok:



## A konvertáló algoritmus pszeudokódja

A kimeneti folyamatgráfot konvertáló algoritmus úgy épül fel, hogy az alapalgoritmus a bemutatott módon létrehozza a duális gráfot majd kezeli a hurkokat, az alternatív útszakaszokat (XOR) és a párhuzamosságokat. A továbbiakban mellékeltem a pszeudokódban található rövidített jelölések listáját és azok magyarázatát is, majd magát a pszeudokódot:

- $old\_nfa.Q$  – Az eredeti automata állapotainak halmaza.
- $new\_nfa.Q$  – A szerkesztett (duális) automata állapotainak halmaza.
- $old\_nfa.T / new\_nfa.T$  – Az automata tranzícióinak (éleinek) halmaza. Az eredeti automatánál egy elemhármass: kezdőpont, végpont, input (esemény). A duális automatánál egy elempár: kezdőpont, végpont (mindkettő esemény). Opcionálisan tartalmazhat ez is inputot, de nincs jelentése.
- $t.input_{end}$  – Az új automata állapotai az eseményeknek felelnek meg, azonban egy esemény több ágon, vagy akár ciklusban is előfordulhat. Ilyenkor nem mindegy, hogy pl. melyik 3-as állapotba kell menni éppen. Ha hozzárendeljük szerkesztéskor azt is, hogy az eredeti automatában melyik (vég)pont tartozik hozzá, akkor egyértelművé lehet tenni. Ezt a pszeudokódban alsó indexszel jelöljük. Ábrázolásakor (az eljárás végén) ezek az alsó indexek eltávolíthatók.
- $old\_nfa.F / new\_nfa.F$  – Az automata végállapotainak halmaza. Az MPM-algoritmus kimeneteként általában olyan automatát kapunk, ahol F egyelemű, ez a duálisban is így marad. A duálisban egy speciális állapot jelöli: END.
- $old\_nfa.q_0 / new\_nfa.q_0$  – Az automata kezdőállapota. A duálisnál ezt az St nevű speciális állapot jelöli.
- 10–11. sor – azt az esetet kezeli, amikor az eredeti automatában egyetlen állapot van, ahonnan és amibe nem húzódik él (tranzíció). Így ez egyben kezdő- és végállapot, ami ezáltal csak az üres szót fogadja el.
- 12–14. sor – azt az esetet kezeli, amikor az eredeti automatában az éppen vizsgált állapotnak nincs bemenő éle (tranzíciója), tehát a kezdőállapotot.
- 15–17. sor – azt az esetet kezeli, amikor az eredeti automatában az éppen vizsgált állapotnak nincs kimenő éle (tranzíciója), tehát a végállapotot.
- 18–21. sor – az általános esetet kezeli, amikor egy állapot szerepel kezdőpontként is, végpontként is valamely tranzíciókban.

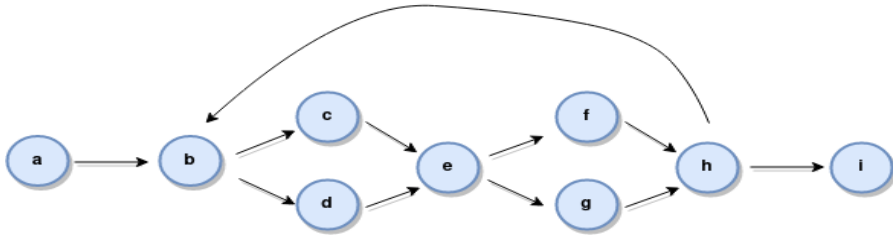
```

CONSTRUCT_DUAL(old_nfa)
new_nfa = new NFA()
new_nfa.Q = U{ t.inputt,end | t ∈ old_nfa.T }
new_nfa.q0 = {St}
new_nfa.F = {END}
new_nfa.T = ∅
FOR state IN old_nfa.Q
 incoming = { t ∈ old_nfa.T | t.end = state }
 outgoing = { t ∈ old_nfa.T | t.start = state }
 IF incoming = ∅ AND outgoing = ∅
 new_nfa.T = new_nfa.T ∪ {(St, END)}
 ELSE IF incoming = ∅
 FOR t IN outgoing
 new_nfa.transitions = new_nfa.T ∪ {(St, t.inputt,end)}
 ELSE IF outgoing = ∅
 FOR t IN incoming
 new_nfa.transitions = new_nfa.T ∪ {(t.inputt,end, END)}
 ELSE
 FOR t_in IN incoming
 FOR t_out IN outgoing
 new_nfa.T = new_nfa.T ∪ {(t_in.inputt_in,end, t_out.inputt_out,end)}

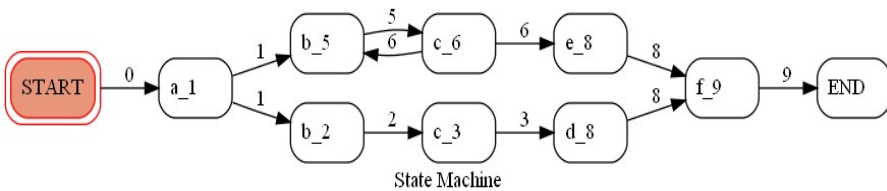
```

(c) Különböző mintákat állítottunk elő, amik a hozzájuk tartozó eseménynaplókkal együtt az ERPA-könyvtárunkba kerültek, mint például:

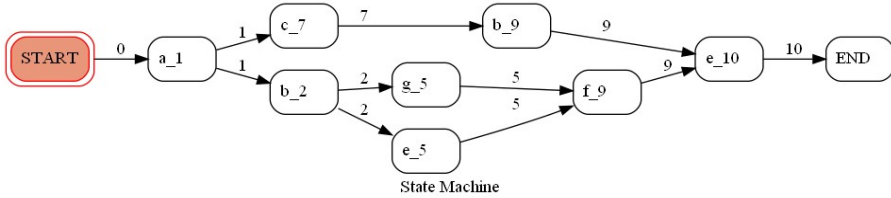
**1. Példa**



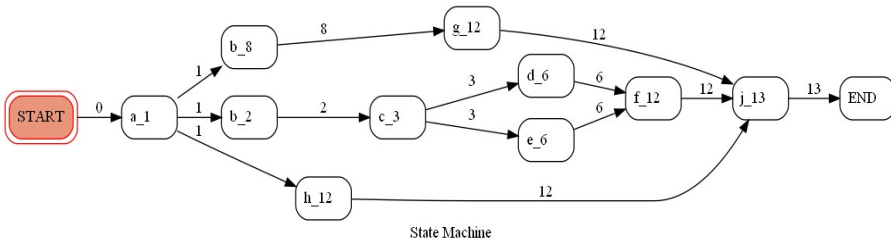
**2. Példa**



### 3. Példa



### 4. Példa



(d) Egy-egy eseménynapló feldolgozásakor a kapott teljes (globális) gráf ellenőrzését visszavezetjük a kapott maximális minták (sémák, patterns) hasonlóságának az ellenőrzésére.

A **statikus tesztelési eljárás** azt jelenti, a teszthalmaz nyomaiból az MPM-eljárás segítségével maximális mintákat gyártunk és ezeket hasonlítjuk össze a tanítóhalmazból nyert mintákkal (vagy a könyvtárunkban tárolt mintákkal).

A **dinamikus eljárás esetén** az előbbi lépésben kapott maximális mintákkal nyomokat generálunk, majd ezekből új maximális mintákat hozunk létre, és ezeknek a „megfelelését” vizsgáljuk.

A keresztvalidációs módszert már az előző évben is használtuk. Röviden összefoglalva: **k-szoros keresztvalidáció** esetén az eseménynaplót hozzávetőleg egyenlő méretű k részre osztjuk.

A kapott partíció minden halmazát pontosan egyszer használjuk fel teszthalmazként, míg a többi halmazban található adatokat tanítóhalmazként használják. Az algoritmus teljesítményét úgy számítjuk ki, hogy minden ilyen lehetséges kiosztás esetén kapott teljesítményértékeknek vesszük az átlagát. Például a 3-szoros keresztvalidációnál a naplót három, nagyjából egyenlő nagyságú részre (A, B, C) osztjuk. Először is, az algoritmus az A és B unióját tekinti tanítóhalmaznak és C-t teszthalmaznak és kiszámítja az ezekre vonatkozó P1 teljesítményt.

Majd az algoritmus B és C-unióját tekinti tanítóhalmaznak és A-t teszt-halmaznak és kiértékeli az így kapott P2 teljesítményt. Végül az A és a C uniója lesz a tanítóhalmaz és B a teszt-halmaz és P3 jelenti a hozzájuk tartozó teljesítményt. Az algoritmus általános teljesítményét a

P1, P2 és P3 teljesítmények számtani átlagaként definiáljuk.

- (e) Az előző évben kapott automata alapú MPM algoritmuson a következő javításokat/módosításokat végeztük el:
- (1) Az egyes kapott mintáknak előbb elkészítettük a duálisát, és amennyiben szükség volt az összekapcsolásukra, csak ezután kapcsoltuk össze a duális gráfokat. Az eddigi eljárásunk egy közös kezdő és egy közös végponttal összefűzött folyamatgráfokat hozott létre, majd képezte ennek a duálisát. Ezt úgy módosítottuk, hogy az MPM-eljárás végén elhagyjuk a közös kezdő- és végállapot beiktatását tartalmazó lépést. Ezeket a speciális állapotokat a duális gráfot létrehozó algoritmus utolsó lépéseként hozzuk létre.
  - (2) Az algoritmus javítása céljából a XOR elágazásnál megjelenő szakaszok hosszát 4-ben korlátoztuk.
  - (3) Az eredeti MPM-algoritmus kimeneteken kapott mintákból kiiktattuk az ún. E-os tranzakciókat, a folyamatgráf hozzájuk tartozó szakaszának a lerövidítésével. A duális gráfot az így „küigazított” kimeneti gráfból kell képezni.
  - (4) Az elágazások vizsgálata előtt az algoritmus a ciklusokat felbontja. A kezdetüket és a végeiket az **LS** és **LE** speciális szimbólumokkal jelzi. Így például a **LOOP (a, b)** és a **LOOP (a, c)** mintákhoz rendre az **LS, a, b, LE**, illetve az **LS, a, c, LE** sorozatot rendeli.
  - (5) Az algoritmus végére bekerült egy újabb lépés a maximális minták meghatározására vonatkozóan, mert a fenti módosítás miatt valamelyik korábbi minta részmintája lehet valamely újonnan keletkező összevont mintáknak. Így például **LOOP (a, b, d, a, c, d)** lehetett volna egy minta – de ez részmintája lenne a **LOOP (a, XOR (b, c), d)** mintának, ezért eltávolítható.
  - (6) A **XOR**-t tartalmazó minták előállításakor azt, hogy ezek maximálisak, oly módon ellenőrizzük, hogy a már előállított hasonló tartalmú minták esetén összevetjük az általuk generált nyomok halmazait majd megismételjük a maximális minta kiválasztására vonatkozó lépést. A kapott **XOR** elágazást tartalmazó mintákat nemcsak egymással vetjük össze, hanem olyan **XOR**-t nem tartalmazó mintákkal is, amik nagyrészt ugyanazokat az eseményeket tartalmazzák. Végül, a **XOR**-t tartalmazó mintákat előbb kettősével, valamennyi pár által előállított nyomhalmazokat megvizsgálva hasonlítjuk össze, majd csak ezután vonjuk őket össze, figyelve arra, hogy

egyetlen csoportba egyesítsük azokat a mintákat, amelyek több összevonható párban is szerepelnek.

**A nyomok bonyolultságát** a következőképpen számoltuk ki:

Minden egyes kérést (request) egy-egy reguláris kifejezéssel adunk meg. Ennek a szerkezetét egy faként ábrázoltuk, ahol az egymásba ékelt zárójelek különböző szinteket jelentenek, A fa gyökerének a mérőszáma 0 (ezt kapjuk az  $\epsilon$  üres szót reprezentáló reg. kifejezésre), az első szinten, az egyes betűk bonyolultsági mérőszáma 1, egy zárójelen belül 2, ha  $k$  számú zárójel van egymásba ágyazva akkor a legbelsőben található karakterek bonyolultsága  $2^k$ . A reguláris kifejezéshez tartozó fa bonyolultsága rekurzívan is értelmezhető, tehát egy csúcspont bonyolultsági mérőszáma a fiaihoz tartozó részfák bonyolultságainak összege. Például az  $a+b+c$  kifejezés bonyolultsága  $1+1+1=3$ , az  $a(bc(d+e)+f)$  kifejezés bonyolultsága  $1+(2+2+(4+4)+2)=15$ .

Megjegyezzük, hogy itt a hurkot is zárójelnek tekintjük, a párhuzamosság-nál(AND) és a XOR alternatívánál egyaránt a műveleteket a  $+$  operator „variánsaiként” kezeljük és úgy tekintjük, hogy használatuk egy újabb zárójel megnyitásával jár. Egy hurok esetén úgy tekintjük, hogy az más operátorokhoz hasonlóan eggyel mélyebb szintet jelent, ha a hurokhoz tartozó szekvencia többször is megismétlődik a nyomban akkor a mérőszámát 2-vel szorozzuk. Tekintsük a következő példákat:

$$a(bc)\{0,3\}(cd)\{1,3\}(ef|fe)\{0,1\}, \quad a(bc)\{1,2\}(cd)\{1,3\}\{ef|fe\}.$$

Ezek bonyolultsága:  $1+2 \times 2+2 \times 2+(2+2) \times 2=17$ , illetve

$$1+2 \times 2+2 \times 2+2+2=13.$$

(Itt hurok és XOR elágazás rögzítésére a pszeudokódban használt konvenciót használtunk, amit az előző projekt szakaszban vezettünk be.)

(f) A megmaradt hibák struktúrájának az elemzése

Az első problémát a következő példán szemléltetjük. Az eseménynaplóhoz tartozó helyes minta legyen a következő: **a, LOOP (b, c), XOR (d, e), f**. Az eseménynapló egy részletét itt láthatjuk:

1. a, b, c, b, c, e, f
2. a, b, c, e, f
3. a, b, c, b, c, b, c, e, f
4. a, b, c, d, f

Itt az algoritmus az első lépésben az 1-es nyomban helyesen felismeri az ismétlődést és „legyártja” a **a, LOOP (b, c), e, f** mintát.

A 2-es és 3-as nyomot helyesen „elnyeleti” ezzel a mintával.

A 4-esben (feltehetően) nem ismer fel hasonlóságot, így ez megmarad maximálisnak: **a, b, c, d, f**.

Ez a minta az ellenőrző lépés során sem tűnik el, mert nem része valamely megkapott maximális mintának.

Tehát a következő két maximális mintát kapjuk (helytelenül):

1. **a, b, c, d, f**.
2. **a, LOOP (b, c), e, f**.

Szerencsére, a minta végrehajtása nem okoz a gyakorlatban hibás működést, csupán redundanciát, ami nem előnyös, ha az eljárás hatékonyságot növelni szeretnénk.

Az MPM-algoritmus ugyancsak felesleges maximális mintákat gyárthat, ha többszörös XOR-on belül LOOP (hurok), vagy ha egy összetett hurkon belül többszörös XOR szerepel, pl.: **LOOP (a, (XOR (b, c), d, XOR (e, f), g).**)

Ilyenkor megtörténhet, hogy a program bizonyos lehetséges elágazásokat nem von be az előállított maximális mintába, hanem mint különálló maximális mintákat kezel (úgy, mint az az előző esetben). Ezt a jelenséget az okozza, hogy az algoritmus bizonyos sorrendet követve állít elő maximális mintákat és ezeket egymással hasonlítja össze, részmintákkal nem. A probléma tökéletes megoldása csak úgy lenne lehetséges, ha valamennyi hasonló tartalmú minta, valamennyi lehetséges csoportosítását megvizsgálánánk – ez viszont a maximális mintahalmaz részhalmazainak a számával lenne egyenesen arányos vizsgálati szám, így a lépésszám exponenciális növekedéséhez vezetne – legalábbis a program ezen részlete esetén. Ezért ez az út nem járható, vagy pontosabban, egzakt megoldás itt nem lehetséges.

### **Az ezzel kapcsolatos tesztek kiértékelése**

A tesztek során felhasznált eseménynaplók 50–250 nyomból álltak, és kis, illetve közepes bonyolultságú nyomokat (eseményszám: 4–14) tartalmaztak. Egyedül a pontossági,  $A_c$  (Accuracy) mutatót mértük és keresztvalidációs módszer használtunk  $K = 5$ , illetve  $K = 3$  paraméterrel. Az utóbbi vizsgálatok során statikus és dinamikus tesztelést is használtunk.

### **A tesztek tapasztalatai:**

- Az algoritmus teljesen jól felismerte az olyan mintákat, amikben csak egyszeres LOOP, XOR vagy AND részlet szerepelt, illetve ahol ezek, vagy ezek közül valamelyik hiányzott.
- Nem tapasztaltunk hibát akkor sem, ha ezekbe AND elágazás volt ágyazva.

Hibát tapasztaltunk akkor, ha

- egy mintában hurok és többszörös XOR elágazás szerepelt,
- egy mintában egymásba ágyazott XOR elágazások vagy hurkok szerepeltek,
- egy mintában egy hurok (LOOP) egy vagy több XOR elágazást is tartalmazott.

A jól kezelt esetekben az Ac mutató 98% feletti volt. A problémás esetekben a legjobb eredmények 82% körül alakultak (ennyi volt a helyesen felismert maximális minták aránya).

Megnyugtató viszont, hogy a hibák 98%-ban a helytelenül előállított maximális sémák valóban létező sémák, tehát nem hamisak, csupán azt nem ismerte fel a program, hogy részei egy náluknál nagyobb maximális mintának. Tulajdonképpen a hibák döntő része abban nyilvánult meg, hogy az MPM eljárás gyakran egyetlen maximális minta helyett több felesleges maximális mintát is generált, amik egy kapott kérésre (requestre) adott alternatív válaszoknak tekinthetők – így valójában, a gyakorlatban nem vezetnek hibás működéshez.

(g) Javaslat a módosított MPM-algoritmus NN-alapú kiegészítésére

Ennek a programrésznek az lenne a feladata, az MPM-program lefutása után sorra vegye a kapott maximális mintákat és megvizsgálja, hogy nem vonhatók-e össze, illetve nincs-e közöttük redundáns minta. Ez úgy történne, hogy a program egy tanulási folyamat eredményeként tárolna a már megkapott és a vizsgált folyamatokra jellemző maximális mintákat, valamint tárolná a futása során az adott eseménynaplóból kinyert mintákat.

Ezek tárolására egy olyan reprezentációt javasoltunk, ami a minták strukturáját tükrözi és nem a kapott gráfokat, ami fellelhető a szakirodalomban. A program memóriája tehát írott sémákat tartalmazna, azzal a fontos pontossággal, hogy a hurkok esetén a felhasznált jelölés jelezné mind a hurok kezdetét, pl. **Begin L**, mind a végét **End L**, például így:

**Begin L, a, XOR (b, c), d, End L,**

az egymásba ágyazottságát pedig zárójelekkel jelölnénk, pl.:

**a, Begin L, b, XOR (c, d), e, (Begin L a, f, End L), End L,** – úgy, ahogy eddig is.



A neuronhálós szubrutin a minták hasonlóság alapú összehasonlításáért (matching) lenne felelős és ezt egy megerősítéssel tanulási folyamat során végezné el. Az eljárás az MPM-algoritmus lefutása után sorra venné a kapott maximális mintákat és megvizsgálná, hogy nem vonhatók-e össze, illetve, összehasonlítaná őket a memóriájában tárolt maximális mintákkal. Ha azt találja, hogy egy újonnan kapott maximális minta hasonlít egy, a memóriájában tárolt mintára, akkor megvizsgálná, hogy az új minta nem részmintája-e a tárolt mintának, és ha igen, akkor megnézi azt, hogy a tárolt régi minta lefed-e olyan nyomokat, amit az új minta nem. Ha nincs ilyen nyom, megőrzi az új mintát, ha van ilyen nyom, akkor helyettesíti az új mintát a tárolt mintával.

Az egyes neuronok implementálása úgy kell majd történnjen, hogy ezek mindegyike valamilyen részlet felismeréséért, például **LOOP, XOR, AND**, zárójel, eseménytípusok stb. legyen felelős. Nyilvánvalóan szükség lesz olyan neuronokra is, amik ezeknek a részeknek egymáshoz viszonyított helyzetét ismerik fel. Ezért több (legalább két) neuronrétegre lesz szükség és feltehetően konvolúciós hálóra. A neuronok tanítása a legegyszerűbb mintákkal kezdődne, amikben még többszörös és beágyazott elemek nincsenek és így haladnánk az egyre bonyolultabbak felé.

- (h) A Chat-GPT felhasználása az algoritmus NN alapú kiegészítésére, a módszer tesztelése

Az általunk vizsgált kiegészítő (rész)megoldás azon alapszik, hogy a Chat-GPT hozzáférhető változatát arra használjuk fel, hogy a módosított MPM-algoritmus által megtalált maximális mintákat összehasonlítsa a csoportunk könyvtárában található maximális mintákkal. Ezt, az adott mintákhoz tartozó reguláris kifejezések összehasonlításával valósítja meg, megvizsgálva, hogy az egyik kifejezés részkifejezése-e egy másiknak, úgy, ahogy ezt az előző jelentésünkben részleteztük. Ezután, ha valamelyik minta részmintája egy, a könyvtárban lévő mintának, akkor ellenőrizzük, hogy a talált nagyobb mintával helyettesíthető-e, illetve, hogy a nagyobb minta segítségével előállított nyomok tartalmazzák-e az összes a kisebbik mintával előállított nyomot, valamint, hogy valóban kapunk-e új, az eseménynaplóban szereplő, de a kisebbik minta által nem felismert nyomot. Ebben a jelentésben csak az algoritmus helyességét vizsgáltuk. Az újonnan kinyert mintákat is bevonjuk a tanulási folyamatba. Ez azt jelenti, hogy az eljárás végén, ha megerősítést nyer az, hogy egy újonnan kapott minta helyes és nem helyettesíthető egy már előzőleg megtalált maximális mintával, akkor az újonnan kinyert mintát felvesszük a program memóriájába, azaz az ERPA-könyvtárunkba.

Ezt a módszert tovább tökéletesítettük, úgy hogy a Chat-GPT-nek egy bővebb „prompt”-ot adtunk meg, amiben a **XOR**, az **AND** és a **LOOP** elemekre vonatkozó „invariancia” szabályokat adtunk meg, például úgy hogy „megtanítottuk” a programot arra, hogy a XOR és AND kifejezések tartalma nem változik meg a változóinak a permutációja révén. A megadott szabályok a következők voltak:

$$\mathbf{XOR(a \mid b) = XOR(b \mid a)}$$

$$\mathbf{XOR(a \mid b \mid c) = XOR(a \mid c \mid b) = XOR(b \mid a \mid c) = XOR(b \mid c \mid a) = XOR(c \mid a \mid b) = XOR(a \mid b \mid c)}$$

$$\mathbf{AND(a \mid b) = AND(b \mid a)}$$

$$\mathbf{AND(a \mid b \mid c) = AND(a \mid c \mid b) = AND(b \mid a \mid c) = AND(b \mid c \mid a) = AND(c \mid a \mid b) = AND(a \mid b \mid c)}$$

$$\mathbf{LOOP(a, b) = LOOP(b, a)}$$

$$\mathbf{LOOP(a, b, c) = LOOP(b, c, a) = LOOP(c, a, b)}$$

Itt jegyezzük meg, hogy a **LOOP** kifejezés csak a karakterek cirkuláris permutációjára invariáns. (A karakterek sorrendjének az inverziója egy 2-nél hosszabb hurok esetén nem megengedett, mert az így kapott kifejezés már nem ekvivalens az eredetivel.)

Itt is csupán az algoritmus helyességét vizsgáltuk. Az újonnan kinyert mintákat bevontuk a tanulási folyamatba, úgy, hogy felvettük a program memóriájába.

Sajnos az előző tesztekben tapasztalt problémák csak egy kicsiny része küszöbölődött ki a „prompt” kiegészítésével. A helytelen eredményeket csak az emberrel való kommunikáció után javította ki a program, de így is maradt hiba. Az alapvető problémát az jelentette, hogy a Chat-GPT több lépésben (itérációban) éri el a helyes eredményt, úgy, hogy közben konzultál a felhasználóval és rendszeresen gondot okoz számára az egymásba ágyazott mintáknak reguláris kifejezésekhez illesztése. Eredményeink azt mutatják, hogy egy neuronháló-alapú osztályozó-összehasonlító szubrutin megtervezése/implementálása nem váltható ki egy a Chat-GPT-hez hasonló általános programmal.

### **1.5. Mérföldkő során megvalósított munkafolyamatok rövid, összegző összefoglalása**

Tökéletesítettük az MPM-algoritmust kimentként egy felhasználóbarát vizuális reprezentáció hozzáadásával. Ennek helyességét és hatékonyságát több teszt-sorozatban ellenőriztük, az általunk kidolgozott statikus és dinamikus módszerek-

kel. Ugyancsak javítottuk a maximális minták felismerésének a helyességét. Ki- alakítottunk egy automatikus tesztelési környezetet felhasználva az ERPA-csoport által kreált könyvtárat. Ezután algoritmusunk hatékonyságát úgy is megvizsgáltuk, hogy az általa kapott eredményeket összevetettük az ERPA-csoport által kidolgozott egyéb adatbányászati eljárások eredményeivel. Javaslatot tettünk az automataalapú MPM-algoritmus egy NN-alapú szubrutinnal való kiegészítésére, megfogalmazva a kiegészítés elvi alapjait és módszertanát. Kipróbáltuk, hogy az MPM-algoritmusunk kiegészíthető-e a Chat-GPT felhasználásával és azt az eredményt kaptuk, hogy ez a módszer úgy lehet hatékony, ha egy szakértővel való kommunikáció lehetőségét is magában foglalja.

## 2. Az elért eredmények bemutatása

Sikerült úgy módosítani/tökéletesíteni az MPM-algoritmust, hogy az a felhasználó számára vizuálisan is ellenőrizhető és legalább 98%-ban helyes mintákat állít elő. Sajnos a minták maximális voltak a felismerése csak 78-80% hatékonysággal történik. Sikerült megfogalmazni a program NN-alapú kiegészítésének az alapjait.

### 2.1. A tesztek, kísérletek megvalósítási környezetének ismertetése

Az összehasonlító teszteléshez az eseménynaplókat az ERPA csoport által kreált könyvtár: ERPA megvalósítók\_ERPA adatforrások\_tesztesetek2023 elérési file-ból vettem. Ezt előzőleg úgy készítettük el, hogy az XES-formátumban kapott magyar nyelvű szöveges adatokat, amik az eseménynapló nyomait jelentik, SPMF-formátumba alakítottuk. Eseménynaplókat hoztunk létre a „traces.txt” weboldalon fellelhető nyomhalmazok segítségével is, valamint a „string generátor” nevű, az internetről letölthető programmal. A folyamatgráf mintákat ugyancsak az ERPA-csoport által kreált könyvtárból vettük, ezeket, illetve a felhasználói kéréseket megadó ún. „requesteket” reguláris kifejezések formájában is ábrázoltuk.

### 2.2. Teszt adathalmazok ismertetése

A felhasznált eseménynaplók a minták ún. vizuális reprezentációja vizsgálatánál általában 1000 generált nyomot tartalmaztak és a minták három különböző requesthez tartoztak. Az eredmények szórását 50–100 közötti futtatások esetén vizsgáltuk meg.

Az összehasonlító vizsgálatokban és az MPM-eljárás tökéletesítése során az eseménynaplók általában kevesebb, 100–250 nyomból álltak és 5–14 eseményszámú nyomokat tartalmaztak. Ugyanilyen paraméterű eseménynaplókat használtunk fel a Chat-GPT-vel kiegészített automata alapú algoritmus tesztelésére, de itt

bonyolultabb, 15–18 eseményt tartalmazó nyomokat is megengedtünk. Az ERPA-könyvtárból kb. 12 mintát vizsgáltunk meg. A felhasznált eseménynaplók: XOR\_2, XOR\_3: XOR\_nested\_1 és a XOR\_nested\_2, loop\_1, loop\_2, a loop\_3 valamint a loop\_nested\_1. Eredményeink azt mutatják, hogy egy neuronháló alapú osztályozó-összehasonlító szubrutin megtervezése/implementálása nem váltható ki egy a Chat-GPT-hez hasonló általános programmal.

Felhasználtuk még a

**LOOP (a, XOR (b, c), d), az a, c, LOOP (d, a, b), d, az a, LOOP (b, c), XOR(d, e), f és a**

**LOOP (a, (XOR (b, c), d, XOR (e, f), g)** kifejezésekkel megadott mintákat. Továbbá Gégény Dávid munkatársam megvizsgált még hat egyéb, általa generált eseménynaplót és mintát is.

### 2.3. *Kiértékelések eredményeinek bemutatása*

**(1) Vizuális ábrázolás, a duális folyamatgráf generálásának a helyessége**

Kétféle tesztet is végeztünk a duális folyamatgráfokra vonatkozólag, statikus és dinamikus folyamatgráf ellenőrzési tesztet is, az összes bevezetett módosítást tartalmazó új MPM-algoritmussal. Mind a statikus, mind a dinamikus tesztelés során a tanítóhalmazból és a tesztalalmazból kapott (duális) folyamatgráfokat hasonlítottuk össze  $k = 5$ -szörös keresztvalidációt használva. A minták egyezőségének mérésére mindkét eljárásnál az Accuracy (Ac) mérőszámot használtuk. Ennek a meghatározásánál a kapott folyamatgráfokra az ún. empirikus hasonlósági mérőszámot használtuk, ami azt jelenti, hogy a módszer folyamatgráfok hasonlóságának a megállapításánál csak azokat a csúcspontokat veszi figyelembe, ahol a két (egymásnak megfelelő) pont kifoka és befoka legfeljebb 1-gyel különbözhet. Ennek a használata, illetve egy ilyen különbségtől való eltekintés eléggé életszerű, mert ilyen különbséget az egyes sémák utólagos összekapcsolása is okozhat. Megvizsgáltuk az egyes mérési eljárások szórását (D) is. A felhasznált eseménynaplót ún. string generátorral hoztunk létre. A vizsgált sorozatok kb. 50-100 futtatást tartalmaztak. Az egyes futtatásoknál kapott eredmények számítani átlagát tekintettük az adott sorozat eredményének, mind a statikus, mind a dinamikus eljárásnál. A felhasznált requestek a következők voltak:

- Egyszerű kérés: **(a)(bc){1,2}(cd){1,3}(ef|fe)** (kis bonyolultság)
- Komplex kérés: **(a)(bc|cb){1,2}(d(e|f)){1,3}** (nagyobb bonyolultság)

- Közepes kérés: **(a)(bc){0,3}(cd){1,3}(ef|fe){0,1}** („közepes” bonyolultság)

### A kapott mérési eredmények

|                            | Statikus m.<br>Ac. átlag | Statikus m.<br>D szórás | Dinam. m.<br>Ac. átlag | Dinam. m.<br>D. szórás |
|----------------------------|--------------------------|-------------------------|------------------------|------------------------|
| request 1.<br>(egyszerű)   | 0,82725                  | 0,07842                 | 0,83164                | 0,07800                |
| request 2.<br>(bonyolult). | 0,74285                  | 0,10155                 | 0,75971                | 0,09210                |
| request 3.<br>(közepes)    | 0,81523                  | 0,14161                 | 0,78700                | 0,13760                |

### (2) Összehasonlító vizsgálatok

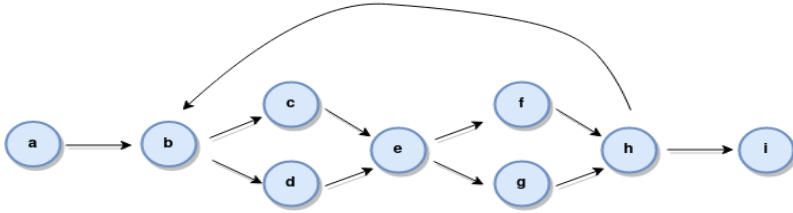
Itt az általunk tökéletesített MPM-eljárás eredményeit vetettük össze azokkal a mintákkal, amelyeket a csoportunk tagjai más eljárásokkal találtak. A XOR\_2, XOR\_3: XOR\_nested\_1 és a XOR\_nested\_2 eseménynaplók esetén kiváló eredményeket kaptunk 98% feletti Ac mutatóval. A futási idők is kedvezően alakultak (szekundumban megadva):

|               |          |
|---------------|----------|
| XOR_2:        | 0,088457 |
| XOR_3:        | 0,140376 |
| XOR_nested_1: | 0,257074 |
| XOR_nested_2: | 0,272478 |

#### 2.4. Eredményeket szemléltető adatsorok

- (a) Ha egy hurok (**LOOP**) ún. **XOR** elágazást is tartalmaz, az MPM-eljárás a XOR eljárás alternatív szakaszait – illetve ezek ismétlődését – gyakran különböző hurkokként érzékeli, így egy minta helyett több mintát generál, amik egy kapott kérésre (requestre) adott alternatív válaszoknak tekinthetők, így ezek valódi hibát nem jelentenek, viszont a tesztelések eredményességi mutatóit rontják, így kiküszöbölésük indokolt. Például az alábbi test\_loop\_1-hez tartozó gráf esetén

**Példa 1. a LOOP (b, XOR (c,d) e XOR (f,g) h) i minta**



Sablonazonosító: test\_loop\_1

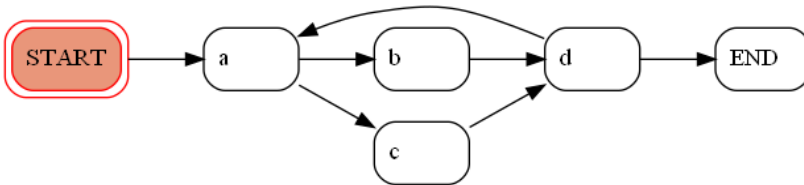
N-méret (napló trace/sorok száma): 200

eseményszám: e = 9

**Példa 2. LOOP (a, XOR (b, c), d) minta**

(eseményszám: e=4)

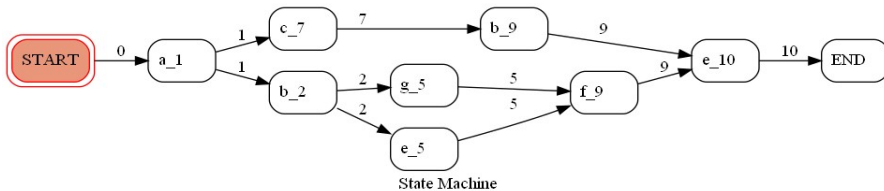
N-méret 200



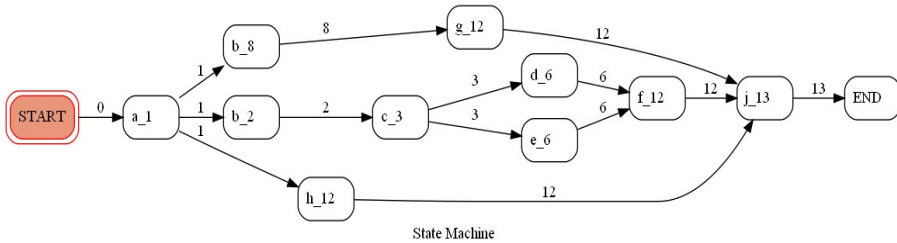
**Példa 3. a XOR (c, b, b XOR (e, g) f) e minta**

(eseményszám: e = 8).

N-méret: 150



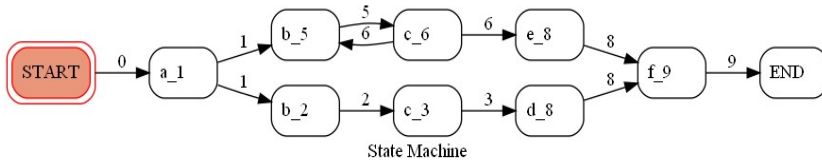
**Példa 4.** – ez az **5\_XXX\_01.png** számú minta az ERPA-csoportkönyvtárból **a, XOR ((b, g), (b, c, XOR (d, e), f), h), j** minta. eseményszám:  $e = 10$



Itt ezeknél ismét 98%-ban helyes eredményt kaptunk. Viszont helytelenül ismertete fel az alábbi mintát:

**Példa 5.**

**a, XOR (LOOP (b, c) e | b, c, d)) f** minta, eseményszám: 8  
N-méret: 200



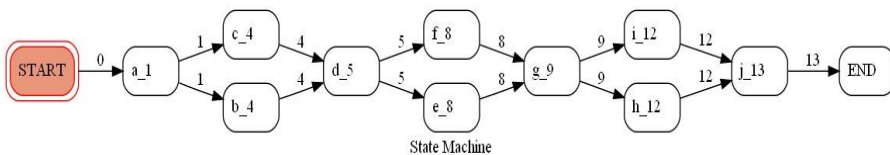
### A program Chat-GPT-vel való kiegészítése

Itt a vizsgált példák az előzőleg már felsoroltak közül voltak, azonban nem sokat javultak az eredmények: Az 1. és 5. példákat valamivel nagyobb hatékonysággal kezelte (50% helyett 78% helyes felismerés) viszont „leromlott” a felismerés jósága a 4. példánál 82%-ra).

Hibákkal talákoztunk a sokszoros **XOR**-t tartalmazó minták esetén a maximális minta képzésénél:

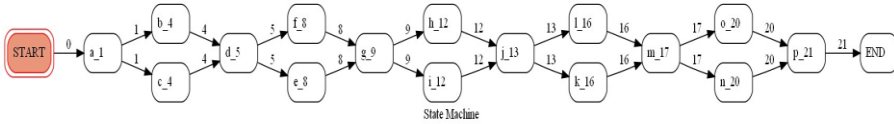
**Példa 6.**

**a, XOR (b | c) d, XOR (e | f), g, XOR (h | i), j**



**Példa 7.**

a, XOR (b | c) d, XOR (e | f) g, XOR (h | i) j, XOR (k | l) m, XOR (n | o), p



A Chat-GPT egy bővebb Prompt-ot megadva ez utóbbiaknak a felismerése megjavult, viszont a **Példa 1.** és **Példa 5.**-ben a hatékonyság csak minimálisan javult (1-2%-kal).

### 2.5. Kutatási eredményeken alapuló javaslatok megfogalmazása

- Az általunk továbbfejlesztett MPM-program jó hatásfokkal nyeri ki a naplók-ból a helyes mintákat. A hibák egy jelentős része a folyamatgráfok duálisának a létrehozásakor jelenik meg, amire azért van szükség, hogy a felhasználók számára áttekinthetőbbé tegyük az eredményeket. Ezért a hibaszázalék jelentősen csökkenthető, ha a tesztelési eljárás során nem magukat a gráfokat, hanem javaslatunknak megfelelően a mintáknak megfelelő reguláris kifejezéseket hasonlítjuk össze az ERPA-könyvtárban tárolt mintákhoz tartozó reguláris kifejezésekkel. (A minták valójában parciális véges automaták, ezért az általuk felismert nyomok reguláris kifejezésekkel leírhatók.)
- A hibák egy másik forrása az, hogy az algoritmus bizonyos esetekben nem vonja össze a kapott mintákat egy maximális mintába. Így a kapott sémák azok valóságos és helyesek, csak tárolásuk redundáns. Megmutattuk, hogy ezt a problémát nem lehet egzakt módon kiküszöbölni, csak egy tanulási folyamat keretében. Ahhoz, hogy ez a tanulási folyamat automatikus módon történjen egy NN-alapú szubrutin implementálását javasoltuk kiegészítés céljából. A teljes folyamatgráf-konvertáló algoritmus futását és hatékonyságát is ezután tesztelni kellene, felhasználva az általunk kidolgozott statikus és dinamikus módszert és a korábban megfogalmazott validációs alapelveket.
- Eredményeink azt mutatják, hogy ha jó hatásfokra törekszünk, akkor egy neuronháló-alapú osztályozó-összehasonlító szubrutin megtervezése/implementálása nem váltható ki egy, a Chat-GPT-hez hasonló, általános programmal. Ennek az a magyarázata, hogy a Chat-GPT feltételez egy szakértő – algoritmus jellegű kommunikációs folyamatot a végeredmény elérése céljából



és kevés kommunikáció mellett az eredmények nem túl biztatók. Ezért egy sajátos és a konkrét célhoz illeszkedő NN-alapú szubrutinnal való kiegészítést javasolunk.

- (d) A neuronhálós szubrutin a minták hasonlóság alapú vizsgálatáért (matching) lenne felelős és ezt egy megerősítéses tanulási folyamat során végezné el a maximális minták ellenőrzése céljából. Az eljárás az MPM-algoritmus lefutása után sorra venné a kapott maximális mintákat és megvizsgálná, hogy nem vonhatók-e össze, illetve, összehasonlítaná őket a memóriájában tárolt maximális mintákkal. Ha azt találja, hogy egy újonnan kapott maximális minta hasonlít egy, a memóriájában tárolt mintára, akkor megvizsgálná, hogy az új minta nem részmintája-e a tárolt mintának, és ha igen, akkor megnézné azt, hogy a tárolt régi minta lefed-e olyan nyomot, amit az új minta nem. Ha nincs ilyen nyom, megőrzi az új mintát, ha van ilyen nyom, akkor helyettesíti az új mintát a tárolt mintával. Az eseménynaplók nyomaiból ilyen módon megkapott minták úgy tesztelhetnének, hogy ezeket és a globális folyamatgráfokat össze kellene hasonlítani az ERPA csoport neuronhálós módszerrel kinyert mintáival és folyamatgráfjaival.
- (e) Azt találtuk, hogy a vizsgált nyomok bonyolultsága csak kismértékben rontja az eredmények helyességét viszont nagymértékben befolyásolja az algoritmus futási idejét. Ezért hasznos lehet a konkrét adatbányászati céllal összhangban a vizsgált eseménynaplók nyomainak a hosszát és bonyolultságát észszerűen korlátozni.

### 3. Elkészült publikációk, belső jelentések megadása, link

Minden egy hónapban végzett tevékenységtől havi összefoglaló jelentést, belső kutatási jelentést és angol nyelvű rövidített kutatási jelentést (Survey) is készítetem, amelyek megtalálhatóak az ERPA csoport könyvtárában a következő linken:

[ERPA Megvalósítók - Files - Nextcloud \(uni-miskolc.hu\)](#) az

ERPA Megvalósítók **alkönyvtárban** (Havi Jelentések, Havi Kutatási Jelentések, Survey mappákban)

Gégény Dávid munkatársammal közösen egy, a

*The possible improvements of the Maximal Pattern Mining algorithm* címmel cikket készülünk benyújtani a Production Systems and Information Engineering folyóirathoz, ennek a törzsanyaga már összegyűlt.

# AZ MPM-ALGORITMUS TOVÁBBFEJLESZTÉSE ÉS VALIDÁLÁSA

GÉGÉNY DÁVID

*A Pythonban implementált Maximal Pattern Mining algoritmus továbbfejlesztése, illetve hatékonyságának mérésére vonatkozó tesztek létrehozása és futtatása a feladatunk. Ezt az összehasonlítást korábban a folyamatgráfon végeztük, most azonban a duális gráfon végezzük. További teszteket végeztünk a csapat által készített példafolyamatok alapján is, melyekhez tartozó eseménynaplókra ellenőriztük, hogy az MPM-algoritmus helyes eredményt adott-e.*

## 1. Elvégzett kísérletek/modellezések bemutatása

### 1.1. A kutatás céljának és lépéseinek összefoglalása

Ebben a kutatási időszakban elsődlegesen az általunk korábban megvalósított, Pythonban implementált *Maximal Pattern Mining* (MPM) algoritmus továbbfejlesztésére, illetve a hatékonyságának mérésére vonatkozó tesztek futtatására került sor. Az algoritmus implementációja a Liesaputra és szerzőtársai által 2015-ben publikált *Efficient process model discovery using maximal pattern mining* című cikk alapján készült. Ennek a cikknek sajnálatos hátrányossága, hogy az MPM-algoritmus nem minden lépését részletezi, így a programozás során ezekhez megfelelő algoritmust kell társítanunk. A fejlesztés során korábban az eljárás véges non-determinisztikus automatát használt, amelynek gráfjában a nyom eseményei az éleken szerepelnek, nem a csomópontokon. Ehhez már készítettünk és implementáltunk egy átalakító algoritmust, amely elkészíti a gráf duálisát – egy olyan gráfot, ahol az események már a csomópontokon szerepelnek. Ennél a módszernél is merülnek fel olyan igények, amelyek azt a célt szolgálják, hogy emberi szemmel jobban átlátható legyen a keletkező folyamatgráf.

Az algoritmus tesztelése *k-szoros kereszvalidációval* történik. Az algoritmus „dinamikus” tesztelése során a létrehozott mintákból egy újabb eseménynaplót készítettünk, majd ismételtelen előállítjuk a mintákat az új eseménynaplóból is. A két mintahalmaz összevetésére a minták gráfjait kell összehasonlítanunk, melyre többféle megközelítést alkalmazunk. Ezt az összehasonlítást korábban a folyamatgráfon végeztük, most azonban a duális gráfon végezzük. További teszteket végeztünk a csapat által készített példafolyamatok alapján is, melyekhez tartozó eseménynaplókra ellenőriztük, hogy az MPM-algoritmus helyes eredményt adott-e.

Az MPM-algoritmus további részletezésére, továbbfejlesztésére is szükségletek merültek fel. Mivel az algoritmust leíró cikk a lépések közül nem mindegyiket részletezi, így könnyen előfordulhatnak olyan határesetek és különleges helyzetek, amelyekre a korábbi implementáció nem volt felkészülve. A tesztek során

például előjött, hogy nem kezelt helyesen bizonyos komplexebb eseteket (például a LOOP elemen belül található XOR elemet). Az algoritmus javítására ilyen szempontból javaslatokat tettünk, a módosításokat implementáltuk.

### **1.2. Feladat módszertani háttere, irodalom rövid áttekintése**

Az MPM-algoritmus implementációja a korábban említett Liesaputra és szerzőtársai által írt cikken alapul. A speciális esetek kezelésére azonban szükséges lehet néhol saját algoritmus kifejlesztése is. A *k*-szoros keresztvalidáció az osztályozási feladatok helyességének ellenőrzésére alkalmas statisztikai módszer, amit széles körben alkalmaznak a gépi tanulási körökben az eljárások hatékonyságának ellenőrzésére. Az MPM-algoritmust megadó Liesaputra-cikk szintén ezt a megközelítést alkalmazza a mintabányászati algoritmus validációjára. Az algoritmus helyességének megítélésére az *accuracy* mérőszámot használjuk.

Az algoritmus által a komplexebb esetek kezelésére irányuló továbbfejlesztések során a speciális elemek (AND, LOOP, XOR) elejét, végét és elválasztóit ideiglenesen helyettesítjük egy-egy speciális szimbólummal, majd soros nyomként kezelve dolgozzuk fel őket – figyelve arra, hogy ne történjenek nem megengedett összevonások, módosítások.

### **1.3. A kidolgozott kísérletek, modellezések módszertana**

A *k*-szoros keresztvalidáció során a rendelkezésünkre álló nyomhalmazt *k* egyenlő részre bontjuk. Ezek közül az egyiket kijelöljük tanítóhalmaznak, ami alapján az algoritmus felépíti az eseménynapló modelljét (mintáit). Utána megvizsgáljuk különféle mérőszámok (pl. *accuracy*) segítségével hogy ez a modell mennyire illeszkedik a többi *k* – 1 nyomhalmaz együttesére, amelyet teszt-halmaznak nevezünk. Az eljárást összesen *k* alkalommal hajtjuk végre, melyek mindegyikénél a tanítóhalmaz szerepébe más-más halmaz kerül. A kapott eredményekre a végén célszerű átlagot (vagy akár szórást is) számolni, így kaphatunk egy értéket, ami alapján megítélhetjük az MPM-algoritmus működésének helyességét.

Az algoritmus által előállított összevont mintát ellenőriztük továbbá a csapat által készített eseménynaplókra is, melyeknél az őket generáló minta ismert. Ahol eltérést tapasztaltunk, ott bizonyos esetekben az algoritmus továbbfejlesztését, kiegészítését láttuk célszerűnek. Ezek a továbbfejlesztések bizonyos szabályszerűségeken alapulnak. Korábban szóba került, hogy a komplexebb esetek kezelésére az AND, XOR és LOOP elemek elejét és végét jelző speciális szimbólumok kerülnek be, így események szekvenciájaként kezelhető a minta. Azonban figyelni kell rá, hogy például a ciklus elejét és végét ne válasszuk el egymástól az összevonások során – a beágyazásoknak ugyanazon szintjén kell lenniük, és

ugyanazon elemen belül. A ciklus elejét, illetve végét LS, illetve LE szimbólumokkal, a párhuzamos események elejét és végét AS, illetve AE szimbólumokkal jelölve például az alábbi minta helytelen: LS, a, AS, b, LE, AE.

Az elemek egyezőségének ellenőrzésekor az algoritmusnak (és a teszteknek is) figyelembe kell vennie bizonyos tulajdonságokat:

- a XOR elemekben szereplő elágazások sorrendje a megadás során nem számít
- az AND elemekben szereplő események sorrendje nem számít
- az AND elem ekvivalens XOR elemmé alakítható, pl.  $\text{AND}(a, b) = \text{XOR}(ab, ba)$
- a konkatenáció disztributív a XOR elemre nézve jobbról és balról is

#### **1.4. A módszer lépéseinek, a referencifolyamatoknak részletezése**

A k-szoros keresztvalidáció során az eseménynaplót reguláris kifejezésekből generált nyomhalmazra néztük meg. Az eseménynaplóban háromféle kéréshez tartozó nyomokat hoztunk létre, melyek mindegyikéhez 1000-1000 db nyomot generáltunk (véletlenszerű sorrendben). A generáláshoz használt reguláris kifejezések eltérő bonyolultságúak a három kérésnél, ez alapján háromféle kérés különböztethető meg: egyszerű, komplex és közepes bonyolultságú.

A tesztek kétféle módon is elvégeztük – ezekre statikus és dinamikus módszerként hivatkozunk. Mindkét esetben a tanítóhalmazból előállítjuk a kéréshez tartozó mintákat és a hozzájuk tartozó duális folyamatgráfot. Statikus esetben a teszhalmazra is elvégezzük ugyanezt, majd összevetjük a mintákhoz tartozó duális gráfokat a tanítóhalmazból képzettekkel. Dinamikus esetben a teszhalmazból képzett mintákból készítünk egy újabb eseménynaplót – ezt generált halmaznak nevezzük. A generált halmazon szintén lefuttatjuk az MPM-algoritmust, és az így kapott duális folyamatgráfokat vetjük össze a tanítóhalmaz folyamatgráfjaival.

#### **1.5. A módszert bemutató folyamatábrák, diagramok**

A duális automata előállítására használt algoritmus, a benne szereplő jelölések, és azok magyarázata:

Jelölések és magyarázat:

- $\text{old\_nfa.Q}$  – Az eredeti automata állapotainak halmaza.
- $\text{new\_nfa.Q}$  – A szerkesztett (duális) automata állapotainak halmaza
- $\text{old\_nfa.T} / \text{new\_nfa.T}$  – Az automata tranzícióinak (éleinek) halmaza. Az eredeti automatánál egy elemhármast: kezdőpont, végpont, input (esemény). A duális automatánál egy elempár: kezdőpont, végpont (mindkettő esemény). Opcionálisan tartalmazhat ez is inputot, de nincs jelentése.

- $t.\text{input}_{t.\text{end}}$  – Az új automata állapotai az eseményeknek felelnek meg, azonban egy esemény több ágon, vagy akár ciklusban is előfordulhat. Ilyenkor nem mindegy, hogy pl. melyik 3-as állapotba kell menni éppen. Ha hozzárendeljük szerkesztéskor azt is, hogy az eredeti automatában melyik (vég)pont tartozik hozzá, akkor egyértelművé lehet tenni. Ezt a pszeudokódban alsó indexszel jelöljük. Ábrázolásakor (az eljárás végén) ezek az alsó indexek eltávolíthatók.
- $\text{old\_nfa.F} / \text{new\_nfa.F}$  – Az automata végállapotainak halmaza. Az MPM algoritmus kimeneteként általában olyan automatát kapunk, ahol F egyelemű, ez a duálisban is így marad. A duálisban egy speciális állapot jelöli: END.
- $\text{old\_nfa.q}_0 / \text{new\_nfa.q}_0$  – Az automata kezdőállapota. A duálisnál ezt a START nevű speciális állapot jelöli.
- AND – egy speciális elemet jelöl a gráfban, amely a párhuzamosan kezelt események állapotainak elejét/végét jelzi.

```

1. CONSTRUCT_DUAL(old_nfa, and_elements)
2. new_nfa = new NFA()
3. new_nfa.Q = U{ t.inputt.end | t ∈ old_nfa.T }
4. new_nfa.q0 = START
5. new_nfa.F = {END}
6. new_nfa.T = ∅
7. FOR state IN old_nfa.Q \ U{ a.states | a ∈ and_elements }
8. incoming = { t ∈ old_nfa.T | t.end = state }
9. outgoing = { t ∈ old_nfa.T | t.start = state }
10. IF ∃ a ∈ and_elements, a.start = state
11. new_nfa.Q = new_nfa.Q ∪ { ANDa.start, ANDa.end }
12. q = (qs ∈ new_nfa.Q | s = a)
13. new_nfa.T = new_nfa.T ∪ {(q, ANDa.start)}
14. FOR x IN { t.input | t ∈ old_nfa.T és t.end ∈ a.states }
15. new_nfa.Q = new_nfa.Q ∪ {t.input}
16. new_nfa.T = new_nfa.T ∪ {(ANDa.start, t.input), (t.input, ANDa.end)}
17. IF incoming = ∅ AND outgoing = ∅
18. new_nfa.T = new_nfa.T ∪ {(START, END)}
19. ELSE IF incoming = ∅
20. FOR t IN outgoing
21. new_nfa.T = new_nfa.T ∪ {(START, t.inputt.end)}
22. ELSE IF outgoing = ∅
23. FOR t IN incoming
24. new_nfa.T = new_nfa.T ∪ {(t.inputt.end, END)}
25. ELSE
26. FOR tin IN incoming
27. FOR tout IN outgoing
28. new_nfa.T = new_nfa.T ∪ {(tin.inputtin.end, tout.inputtout.end)}
29. IF state IN old_nfa.F
30. new_nfa.T = new_nfa.T ∪ {tin.inputtin.end, END}
31.
32.

```

## 1.6. Mérföldkő során megvalósított munkafolyamatok rövid, összegző összefoglalása

A kutatás során az automataalapú folyamatfeltáráshoz használt MPM-algoritmus implementációt továbbfejlesztettük, validáltuk, illetve különféle futási tesztek hajtottunk végre. A duális folyamatgráfhoz tartozó tesztek alapján azt tapasztaltuk, hogy az MPM viszonylag jó eredményekkel teljesít, azonban lehet még fejleszteni. A tesztelések során először többféle megközelítést alkalmaztunk a gráfok közötti hasonlósági mérték kiszámítására, de a későbbiekben már csak azt alkalmaztuk ezek közül, amelyik a valószerűbb eredményt adja (legfeljebb 1-1 eltérés lehet a kimenő és bemenő éleknél). Az eljárásn javítottunk azzal, hogy kiküszöböltük az epsilonos átmenetek közül azokat, amelyek nem voltak feltétlenül szükségesek. Bizonyos komplexebb mintán alapuló eseménynaplóknál az implementált MPM algoritmus olyan folyamatgráfot hozott létre ugyan, ami illeszkedik a naplóra, viszont a felhasználó számára nem feltétlenül áttekinthető. A probléma orvoslására az algoritmuson tovább javítottunk a komplexebb, beágyazott speciális elemek jobb kezelésével.

## 2. Elért eredmények

### 2.1. A tesztek, kísérletek megvalósítási környezetének ismertetése

Az MPM-algoritmus és a kapcsolódó tesztek implementációja *Python*-ban történt, különféle szabadon felhasználható függvénykönyvtárak segítségével. A matematikai műveletekhez a *numpy* függvénykönyvtárat használtuk. A mintákhoz tartozó automaták létrehozására, reprezentálására, adott nyom elfogadásának ellenőrzésére a *pytransitions* függvénykönyvtár szolgált. A speciális elemek (AND, XOR) esetén az egyenlőség megállapításához az *itertools* függvénykönyvtár nyújtott segítséget, amely a permutációk létrehozására és egyéb kombinatorikai műveletek elvégzésére biztosít függvényeket. A véletlenszám-generálásra és a kapcsolódó műveletekre a *random* függvénykönyvtárat használtuk. Az algoritmus futásidejének mérésére a *timeit* függvénykönyvtárat használtuk. A tesztek futtatását Windows 11 operációs rendszeren végeztük a fejlesztésre is használt Visual Studio Code fejlesztői környezet Python termináljának segítségével. A tesztelés során használt számítógép Intel i7 2.1 GHz processzorral rendelkezik, 32 GB RAM memóriával.

### 2.2. Teszt adathalmazok ismertetése

A k-szoros keresztvalidáció során egy 3000 nyomból álló eseménynaplót készítettünk, amelyet reguláris kifejezésekből generáltunk. Az eseménynaplóban háromféle kéréshez tartozó nyomok szerepelnek, egyenlő arányban. Bonyolultságuk szempontjából három különböző reguláris kifejezést használtunk:

- Egyszerű kérés:  $(a)(bc)\{1,2\}(cd)\{1,3\}(ef|fe)$
- Komplex kérés:  $(a)(bc|cb)\{1,2\}(d(e|f))\{1,3\}$
- Közepes kérés:  $(a)(bc)\{0,3\}(cd)\{1,3\}(ef|fe)\{0,1\}$

Az MPM-algoritmus továbbfejlesztése során az ellenőrzéseket a csapat többi tagja által előállított példa-eseménynaplókon végeztük. Ezekhez az eseménynaplókhöz ismert a (duális) folyamatgráf, legtöbbjük emberi szemmel is könnyen értelmezhető. Ahol az MPM-algoritmus implementációja sokkal bonyolultabb vagy eltérő eredményt szolgáltatott, ott kivizsgáltuk a problémát.

### 2.3. *Kiértékelések eredményeinek bemutatása*

Amikor az MPM-algoritmust kipróbáltuk a csapat által előállított eseménynaplókon, bizonyos esetekben bonyolult, emberi szemmel nem átlátható folyamatgráfot hozott létre, amikor a speciális elemek (LOOP, AND és XOR) többszörös beágyazása történt. Ezeket az eseteket kivizsgálva egyrészt arra jutottunk, hogy egyrészt javítanunk kell az elemek egyezőségének ellenőrzésén, másrészt pedig vannak olyan minták, amik még összevonhatók lennének. Utóbbihoz a speciális elemeket átalakítottuk úgy, hogy a kezdetüket és végüket speciális szimbólumok jelöljék:

- LS = loop start
- LE = loop end
- AS = and start
- AE = and end
- XS = xor start
- XE = xor end
- XD = xor delimiter

A XOR esetén az egyes ágak elválasztására szintén egy új szimbólumot vezettünk be. A LOOP, XOR és AND elemeket kivesszük, az elejükre/végükre/elválasztóhoz pedig beírjuk a megfelelő szimbólumot. Ezáltal egy egyszerű szekvenciát hozunk létre, ami majdnem kezelhető úgy, mintha egy sima nyom lenne. Például az egyik minta legyen: **a, LOOP(b, c, e), f**. A másik minta legyen: **a, LOOP(b, d, e), f**.

- Az összehasonlításnál az első mintából ez lesz: **a, LS, b, c, e, LE, f**
- A második mintából pedig ez lesz: **a, LS, b, d, e, LE, f**.

Köztük pedig az opcionálitáshoz lehet vizsgálni az egyezőségeket/eltéréseket és nincs szükség rekurzióra sem. Ebből a következő mintát kapjuk: **a, LS, b, XOR(c, d), e, LE, f**

Ezt visszaalakítva: **a, LOOP(b, XOR(c, d), e), f**.

Ezután szükséges még a maximális minták újbóli megállapítása, mert lehet olyan egyéb minta, amely már részmintája lesz az összevont mintának. Ezt egy újabb példával szemléltetem:

- Minta 1: **a, LOOP(b, c, e), f**
- Minta 2: **a, LOOP(b, d, e), f**
- Minta 3: **a, b, c, e, b, d, e, f**

Korábban láttuk, hogy az első két minta összevonásából keletkező minta: **a, LOOP(b, XOR(c, d), e), f**

Ennek viszont részmintája lett a harmadik minta is, így azt eltávolíthatjuk, mert nem maximális.

#### 2.4. Eredményeket szemléltető képernyőképek, adatsorok

A k-szoros keresztvalidáció futtatási eredményei az alábbiak statikus esetben:

|                    | Egyszerű kérés<br>accuracy<br>(statikus) | Komplex kérés<br>accuracy<br>(statikus) | Közepes kérés<br>accuracy<br>(statikus) |
|--------------------|------------------------------------------|-----------------------------------------|-----------------------------------------|
| Átlag              | 0,836242333                              | 0,741419832                             | 0,772967556                             |
| Szórás             | 0,09076501                               | 0,084465043                             | 0,16324092                              |
| Normalizált szórás | 0,108539124                              | 0,113923366                             | 0,211187285                             |
| Minimum            | 0,612947347                              | 0,547593504                             | 0,40034965                              |
| Maximum            | 1                                        | 0,898557529                             | 1                                       |
| Terjedelem         | 0,387052653                              | 0,350964025                             | 0,59965035                              |

Dinamikus esetben:

|                    | Egyszerű kérés<br>accuracy<br>(dinamikus) | Komplex kérés<br>accuracy<br>(dinamikus) | Közepes kérés<br>accuracy<br>(dinamikus) |
|--------------------|-------------------------------------------|------------------------------------------|------------------------------------------|
| Átlag              | 0,813366842                               | 0,737802314                              | 0,783873389                              |
| Szórás             | 0,093465197                               | 0,096132515                              | 0,138174607                              |
| Normalizált szórás | 0,114911492                               | 0,130295762                              | 0,176271588                              |
| Minimum            | 0,577289646                               | 0,403096774                              | 0,40034965                               |
| Maximum            | 1                                         | 0,998487865                              | 1                                        |
| Terjedelem         | 0,422710354                               | 0,595391091                              | 0,59965035                               |



MPM-algoritmus futásideje másodpercben a tanítóhalmaz esetén:

|                           | <b>Egyszerű kérés<br/>MPM-futásidő<br/>(tanítóhalmaz)</b> | <b>Komplex kérés<br/>MPM-futásidő<br/>(tanítóhalmaz)</b> | <b>Közepes kérés<br/>MPM-futásidő<br/>(tanítóhalmaz)</b> |
|---------------------------|-----------------------------------------------------------|----------------------------------------------------------|----------------------------------------------------------|
| <b>Átlag</b>              | 5,017774079                                               | 26,64065237                                              | 6,518885111                                              |
| <b>Szórás</b>             | 0,666641411                                               | 7,631032429                                              | 0,760867584                                              |
| <b>Normalizált szórás</b> | 0,132856004                                               | 0,286443152                                              | 0,11671744                                               |
| <b>Minimum</b>            | 4,1032081                                                 | 10,9237524                                               | 5,2754233                                                |
| <b>Maximum</b>            | 7,788368                                                  | 48,9277802                                               | 10,4449159                                               |
| <b>Terjedelem</b>         | 3,6851599                                                 | 38,0040278                                               | 5,1694926                                                |

MPM-algoritmus futásideje másodpercben a teszhalmaz esetén:

|                           | <b>Egyszerű kérés<br/>MPM-futásidő<br/>(teszhalmaz)</b> | <b>Komplex kérés<br/>MPM-futásidő<br/>(teszhalmaz)</b> | <b>Közepes kérés<br/>MPM-futásidő<br/>(teszhalmaz)</b> |
|---------------------------|---------------------------------------------------------|--------------------------------------------------------|--------------------------------------------------------|
| <b>Átlag</b>              | 28,53793914                                             | 69,46134532                                            | 36,91682876                                            |
| <b>Szórás</b>             | 3,320942286                                             | 7,571283939                                            | 3,584916672                                            |
| <b>Normalizált szórás</b> | 0,11636938                                              | 0,108999961                                            | 0,097107926                                            |
| <b>Minimum</b>            | 25,9168394                                              | 62,2058263                                             | 33,1373155                                             |
| <b>Maximum</b>            | 43,8900976                                              | 110,5834647                                            | 57,0458735                                             |
| <b>Terjedelem</b>         | 17,9732582                                              | 48,3776384                                             | 23,908558                                              |

MPM-algoritmus futásideje másodpercben a generált halmaz esetén:

|                           | <b>Egyszerű kérés<br/>MPM-futásidő<br/>(generált halmaz)</b> | <b>Komplex kérés<br/>MPM-futásidő<br/>(generált halmaz)</b> | <b>Közepes kérés<br/>MPM-futásidő<br/>(generált halmaz)</b> |
|---------------------------|--------------------------------------------------------------|-------------------------------------------------------------|-------------------------------------------------------------|
| <b>Átlag</b>              | 148,3983804                                                  | 233,921536                                                  | 197,3984879                                                 |
| <b>Szórás</b>             | 14,10400808                                                  | 21,21106883                                                 | 18,69077679                                                 |
| <b>Normalizált szórás</b> | 0,095041523                                                  | 0,090675998                                                 | 0,094685511                                                 |
| <b>Minimum</b>            | 130,7849762                                                  | 206,611172                                                  | 180,7265767                                                 |
| <b>Maximum</b>            | 222,1692401                                                  | 350,4907305                                                 | 290,7117599                                                 |
| <b>Terjedelem</b>         | 91,3842639                                                   | 143,8795585                                                 | 109,9851832                                                 |

Az MPM-algoritmus accuracy mérőszámai javítások implementálása után statikus esetben:

|                    | Egyszerű<br>kérés<br>accuracy<br>(statikus) | Komplex<br>kérés<br>accuracy<br>(statikus) | Közepes<br>kérés<br>accuracy (sta-<br>tikus) |
|--------------------|---------------------------------------------|--------------------------------------------|----------------------------------------------|
| Átlag              | 0,82725831                                  | 0,742850097                                | 0,815238095                                  |
| Szórás             | 0,07842502                                  | 0,101550018                                | 0,141617729                                  |
| Normalizált szórás | 0,094801127                                 | 0,136703244                                | 0,173713335                                  |
| Minimum            | 0,723148148                                 | 0,47800339                                 | 0,438461538                                  |
| Maximum            | 0,994354839                                 | 0,907946258                                | 1                                            |
| Terjedelem         | 0,271206691                                 | 0,429942868                                | 0,561538462                                  |

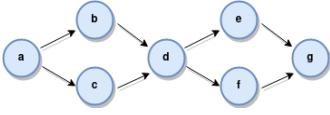
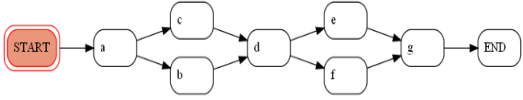
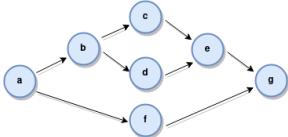
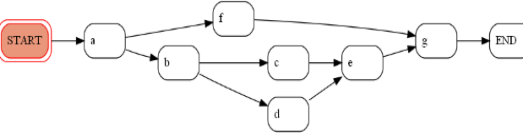
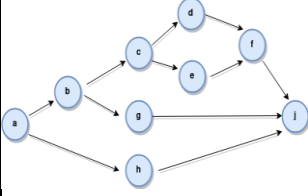
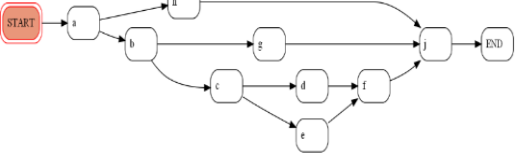
|                                          |              |             |              |
|------------------------------------------|--------------|-------------|--------------|
| Accuracy átlageltérés az előzőhöz képest | -0,008984023 | 0,001430265 | 0,042270539  |
| Szóráseltérés az előzőhöz képest         | -0,01233999  | 0,017084975 | -0,021623191 |

Az MPM-algoritmus accuracy mérőszámai javítások implementálása után dinamikus esetben:

|                    | Egyszerű<br>kérés<br>accuracy<br>(dinamikus) | Komplex<br>kérés<br>accuracy<br>(dinamikus) | Közepes<br>kérés<br>accuracy<br>(dinamikus) |
|--------------------|----------------------------------------------|---------------------------------------------|---------------------------------------------|
| Átlag              | 0,831642818                                  | 0,759718117                                 | 0,787008547                                 |
| Szórás             | 0,087002102                                  | 0,092102434                                 | 0,137601726                                 |
| Normalizált szórás | 0,104614746                                  | 0,121232378                                 | 0,174841463                                 |
| Minimum            | 0,640843621                                  | 0,576196074                                 | 0,438461538                                 |
| Maximum            | 0,99382716                                   | 1                                           | 1                                           |
| Terjedelem         | 0,352983539                                  | 0,423803926                                 | 0,561538462                                 |

|                                          |              |              |              |
|------------------------------------------|--------------|--------------|--------------|
| Accuracy átlageltérés az előzőhöz képest | 0,018275976  | 0,021915803  | 0,003135158  |
| Szóráseltérés az előzőhöz képest         | -0,006463095 | -0,004030081 | -0,000572881 |

Néhány példa a csapat által készített eseménynaplókból, illetve az MPM algoritmus által a hozzájuk előállított folyamatgráf:

| <i>Eseménynaplót generáló<br/>folyamatgráf</i>                                     | <i>MPM által előállított folyamatgráf</i>                                           |
|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|   |   |
|   |   |
|  |  |

### 2.5. Kutatási eredményeken alapuló javaslatok megfogalmazása

Az MPM-algoritmus a futtatási mérőszámok alapján jól teljesít ugyan, bizonyos esetekben olyan folyamatgráfot generál, ami – bár illeszkedik a nyomokra, mégsem elég egyszerű vagy emberi szemmel nem túl jól átlátható. Az algoritmus megfelelő implementációja az eredeti cikkben szereplő részletes specifikáció hiánya miatt nehézkes lehet, ami bizonyos határesetek, nem várt szituációk, komplex folyamatok esetén problémát okozhat. A módszer analitikus módon generál nagyrészt helyes mintát, amit a k-szoros keresztvalidációval kapott eredmények is szemléltetnek. Önmagában ezt az algoritmust azonban csak viszonylag egyszerű folyamatok mintafeltárására javaslom. Bonyolultabb folyamatok esetén az említett hátrányok előtérbe kerülnek – ilyenkor célszerű kombinálni valamilyen más módszerrel, pl. kiegészíteni mesterséges neurális hálókon vagy gépi tanuláson alapuló lépésekkel, hogy átláthatóbb eredményt adjon.

# ÜGYFÉLSZOLGÁLATI MEGKERESÉSEK AUTOMATIKUS FELDOLGOZÁSÁNAK LEHETŐSÉGEI EMBERKÖZPONTÚ MESTERSÉGESINTELLIGENCIA-MÓDSZEREKKEL

Csepányi-Fürjes László

*A korábbiakban megtervezett SynergyUnit keretrendszer tartalmi implementációjának továbbfejlesztését tűztük ki célul, mivel az ügyfélszolgálati környezetben alkalmazott mesterségesintelligencia- (MI) rendszereket meg kívánjuk feleltetni a legmagasabb emberközpontú mesterséges intelligencia (EMI) feltételrendszerének. Emellett célunk, hogy az üzenetekből minden releváns információt kinyerjünk, amelyeket a tervezett rendszer felhasználhat a további feladatok megoldásához.*

## 1. Elvégzett kísérletek bemutatása

### 1.1 Feladat célkitűzése, motiváció, elvégzendő szakmai feladat bemutatása

Az ügyfélszolgálati folyamatok optimalizálása és hatékonyabbá tétele egyre nagyobb jelentőséggel bír a vállalatok számára. Az ügyfelek által írt e-mailek vagy telefonon történt hívások tartalmának elemzése lehetővé teszi az ügyfél szándékainak és igényeinek gyors és hatékony azonosítását. A feladat célja, hogy olyan szövegekategorizáló módszereket és nyelvi modelleket vizsgáljunk, amelyek segítségével az ügyfélszolgálati szövegek automatikus elő-kategorizálása és elemzése elvégezhető.

Eddigiek mellett a beérkező dokumentumok tartalomosztályozásának továbbfejlesztését állt szándékunkban elvégezni, valamint egy e-mail-osztályozáshoz használható tesztadatbázist kívántunk felépíteni.

Ezen felül célul tűztük ki a – napjainkban nagy népszerűségnek örvendő – generatív nyelvi modellek alkalmazhatóságának ügyfélszolgálati környezetben történő áttekintését.

Végül, de nem utolsósorban, meg kívántuk vizsgálni, hogy a gráfadatbázisok hogyan használhatóak nagy nyelvi modellekkel kiegészítve ügyfélszolgálati környezetben.

### 1.2. Feladat módszertani háttere, irodalom rövid értékelése

Zhou és mtsai. tanulmányukban áttekintik az úgynevezett intelligencia felerősítés (Intelligence Augmentation) irányába tett kutatási tevékenységeket [1]. Ezen irányzat célul tűzte ki az emberi intelligencia fejlesztését, segítségét egy általános mesterséges intelligencia megalkotása és alkalmazása helyett. Csepeli könyvében szociológiai szempontból elemzi a mesterséges intelligencia gazdasági és társa-

dalmi hatásait [2]. A kutatás során megállapítható volt, hogy az EMI célkitűzéseihez leginkább alkalmazkodó megoldás a mesterséges intelligenciát alkalmazó rendszerek tudásmegosztási képességének kifejlesztése. Ez a gondolat határozza meg a kutatócsoport által kifejlesztett SynergyUnit keretrendszer alapjait.

Chen és mtsai. egy BERT-re épülő közös INTENT CLASSIFICATION és SLOT FILLING modellt javasol, amely egyszerre valósítja meg a szövegkategorizálás és adatkinyerés feladatát [5]. Tanulmányukban kitérnek az emberi szakértő által címkézett tréningadatok hiányára és annak kezelési lehetőségeire. Gangadharaiyah és Narayanaswamy felhívják a figyelmet, hogy valós helyzetben a felhasználók több szándékot is kifejezhetnek ugyanabban a szövegben, és egy TOKEN szintű SLOT címke több szándékhoz is tartozhat [3]. Ezen két problémára keresnek megoldást tanulmányukban. Előbbi tanulmányokra alapozva alkotta meg és elemezte a kutatócsoport az e-mail-kategorizáló modellt.

Tan és mtsai. értekezésükben egy olyan energia-ügyfélszolgálati szakterület-specifikus tudásgráfot javasolnak, mely tulajdonképpen a fogalmak, entitások, események és azok kapcsolatainak strukturált módú leírása [4]. A gráfmodell lehetővé teszi a hatékonyabb adatszervezést, kezelést, valamint kognitív képességeket biztosít az ügyfélszolgálati szakemberek számára.

### ***1.3. A kidolgozott kísérletek/modellezések módszertana***

Adatgyűjtés céljából a Miskolci Egyetem Számítóközpontjának központi levelezőrendszeréből e-mailek kinyerésére került sor egy meghatározott időszakra vonatkozóan. Mivel az elektronikus levelek személyes adatnak minősülnek, ezért elsődleges szempont volt, hogy a levelek tartalmi feldolgozása megfeleljen a hatályos GDPR-előírásoknak. Minden egyes levélből eltávolításra került a feladó beazonosítására használható információ. A hibás karakterkódolások javításra kerültek, valamint azok a levelek kerültek eltávolításra az adathalmazból, melyek a kutatás szempontjából nem használhatóak.

Tanító és teszt adathalmazok kialakítása céljából a nyers szövegadatokat címkézésre volt szükség. Az annotáció elvégzéséhez a már korábban elemzett Doccano applikáció került felhasználásra. A Doccano egy lokális Docker image-ként került telepítésre, ezzel könnyítve a munkát. Bemenetként a Doccano alkalmazás egy fájllistát vár, mely esetben minden egyes fájl egy önálló dokumentum, vagyis egy önálló e-mail.

Az alapmodell kiválasztása és a paraméterek beállítása során elsődleges cél volt, hogy a már korábban létrehozott SZDTROBERTA modell összehasonlításra kerüljön a Magyar Nyelvtudományi Intézet HIL\_ROBERTA modelljének teljesítményével, valamint alap HUGGING-FACE ROBERTA-BASE modellel.

A kísérletek és eredmények értékelésekor a betanított modellek a teszt adathalmaz segítségével kerültek értékelésre. Kiértékelés során az ACCURACY érték vizsgálata történt meg, valamint nyomkövetésre került a LOSS csökkenése a tanítás során.

#### **1.4. A módszer lépéseinek, a referencia folyamatoknak részletezése**

##### *Emberközpontú mesterséges intelligencia*

A SynergyUnit keretrendszer alapján felépített referenciaalkalmazás elemzése miatt kiépítésre került egy felhasználói tevékenységet nyomon követő és loggoló alrendszer, mely az egyes tevékenységelemeket és a tevékenységekhez tartozó adatokat tárolja. Az adatok tárolására relációs adatbázis került felhasználásra, amely két-táblás alrendszerként valósult meg.

A nyomkövetés teszi lehetővé, hogy a referenciaalkalmazás használata a SynergyUnit alrendszer vonatkozásában kiértékelhető legyen. Konkrétan az került elemzésre, hogy a felhasználó mennyire használja munkája során az Explain és a Teach unitokat. Ezek az egységek felelősek az MI-rendszer tudásmegosztásáért. A vizsgálat során a kísérletben résztvevő alanyok két csoportba kerültek. Az egyik csoport számára elérhető volt az ExplainSU és a TeachSU szolgáltatásai, míg a másik csoport számára ezek letiltásra kerültek. Vizsgálat alá került a két csoport hatékonysága, valamint személyes interjúk segítségével mérésre került a rendszer használatának felhasználói élménye.

##### *Dokumentumosztályozás*

A beérkező dokumentumok osztályozása esetében külön elemzésre került az az eset, amikor az osztályozást emberi ügynök végzi egyedül, valamint amikor az osztályozást nyelvi modell segítségével egy automatikus rendszer végzi el. Három nyelvi modell került összehasonlításra. A korábbiakban, a kutatócsoport által létrehozott SZDTROBERTA modell, a Magyar Nyelvtudományi Intézet HIL\_ROBERTA modellje, valamint a HUGGING-FACE ROBERTA\_BASE modellje. Mindhárom modell esetében az ACCURACY mérőszám került vizsgálatra az iterációk függvényében.

Generatív nyelvi modellek tekintetében a ChatGPT 3.5-ös verziója került tesztelésre. A megfogalmazott PROMPT segítségével kategorizálás és adatkiemelés került definiálásra, majd az eredmények összevetésre kerültek az emberi szakértő kategorizálásának eredményével.

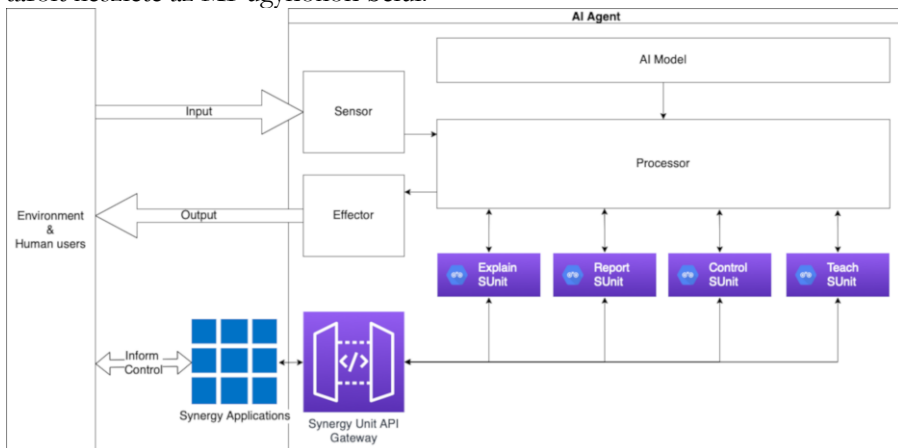
### Gráfadatbázisok

Az ügyfélszolgálati tevékenységek optimalizálása és hatékonyságának növelése érdekében a gráfadatszerkezet számos módon használható. A gráfadatszerkezet segíthet az ügyfeladatok hatékony kezelésében és szervezésében. Lehetővé teszi az ügyfélszolgálati hálózatok és kapcsolatok elemzését. Alkalmazhatók az ügyfélszolgálati támogatási útvonalak és döntési fák modellezésére. Az ügyfélpanaszokat és problémákat gráfként modellezve könnyebben azonosíthatók a gyakori problémák és azok az ügyfelek, akik gyakran hasonló problémákkal küzdenek. Az ügyfélkapcsolatokat gráfként modellezve könnyebb azonosítani azokat az ügyfeleket, akiknek hasonló igényei vannak, és személyre szabott ajánlatokat vagy megoldásokat kínálni számukra. A vizsgálat során ajánlások kerültek megfogalmazásra a gráfadatbázisok felhasználhatóságának tekintetében.

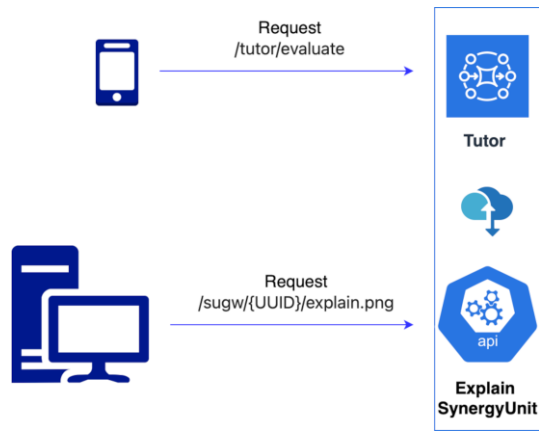
### 1.5. A módszert bemutató folyamatábrák, diagramok

#### EMI ügyfélszolgálati környezetben

A SynergyUnit egy szoftverkomponens, mely a tudásmegosztó funkciók behatárolt készlete az MI-ügynökön belül.

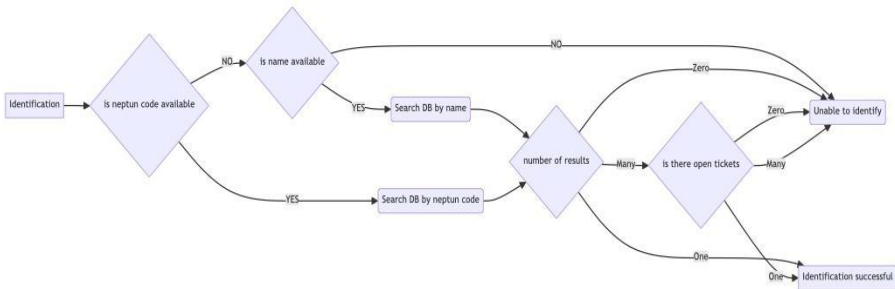


A tudásmegosztás funkciójának eléréséhez az ügyfélszolgálati alkalmazásnak kommunikálnia kell a SynergyUnit Gateway-en keresztül a TeachSU-val és az ExplainSU-val.



### Dokumentumosztályozás

Az osztályozás során adatok gyűjtésére is lehetőség nyílik. Az adatkinyerés bizonyos esetekben konkrét feladat szekvenciát indukál. Példaként tekinthető a számítógéppontba beérkező levelek feladójának azonosítási szekvenciája: van-e névalírás az e-mailben → van-e Neptun kód → keresés adatbázisban: több találat → szűkítés egyéb adatokból, pl. valamelyik a találatok közül indított-e folyamatot az elmúlt 1 hétben → van egy találat → OK.

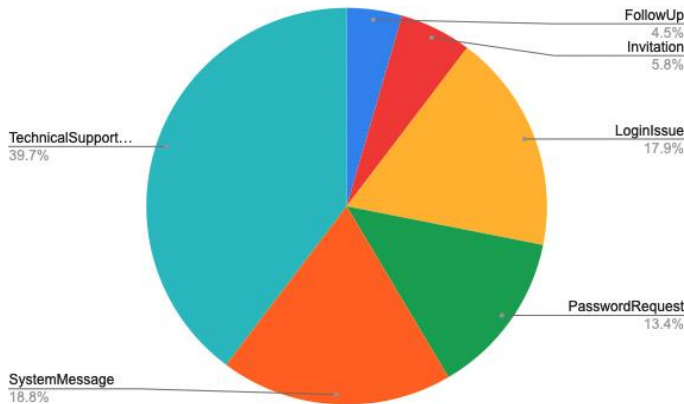


A dokumentumok emberi szakértő általi osztályozásához használt Doccano alkalmazást Docker konténerben célszerű futtatni.

|                          | Name                                 | Image                  | Status  | Port(s)   | Last started  | Actions |
|--------------------------|--------------------------------------|------------------------|---------|-----------|---------------|---------|
| <input type="checkbox"/> | uom-email-annotation<br>8a0f631e4e62 | doccano/doccano:latest | Running | 8000:8000 | 3 minutes ago | ⋮       |



Az e-mail-adathalmaz normalizálás előtti eloszlása meglehetősen egyenetlen volt. Egy normalizálási folyamat segítségével ki kellett egyenlíteni azt. A normalizálás többek között magában foglalta a TechnicalSupportRequest kategória elemeinek csökkentését, valamint a FollowUp és az Invitation kategóriák összevonását.



### 1.6. Mérföldkő során megvalósított munkafolyamatok rövid, összegző összefoglalása

Az ügyfélszolgálati környezetben megvalósítandó EMI-alkalmazások fejlesztésére vonatkozó szakirodalom áttekintése után megállapítható volt, hogy az elsődleges cél a rendszer tudásmegosztási képességének lehetővé tétele. Kidolgozásra került a SynergyUnit keretrendszer, mely a tudásmegosztó funkciók behatárolt készletének leírása az MI-ügynökön belül. Az elmélet kidolgozása után elkészült a rendszer prototípus-implementációja is. A prototípus univerzális egyedi azonosítókat (UUID) használ a kommunikációs folyamat azonosítására, amely REST-szolgáltatásként valósul meg. A modul válaszol a kérésre, miután elvégezte a kiértékelést az MI-modell segítségével.

A dokumentumosztályozás elvégzéséhez létrehozandó nyelvi modell minden esetben megköveteli a tanító adathalmaz megfelelő minőségét. Abból a célból, hogy a megfelelő minőség elérhető legyen, adattisztítást kell végrehajtani. Az adattisztítás automatizált és manuális munkafolyamatokat egyaránt tartalmaz. Ezt követően a tanító adathalmaz manuális címkézése történt meg. A címkézés eredménye egy JSON-formátumú szövegsorokat tartalmazó fájl, melyet további

konverzióknak kellett alávetni, hogy az a tanulóalgoritmus bemeneti követelményeinek megfelelően. Az algoritmus programozása után létrehozásra került egy tesztkörnyezet, a finomhangolt modellek elemzése céljából.

A generatív nyelvi modellek tanulmányozása során megállapítható volt, hogy a hagyományos értelemben vett programozási szintek (Low-Level programming, High-Level programming) mellett megjelent egy új szint, mely a kutatócsoporttól az emberi szintű programozás (Human-Level Programming) nevet kapta. Ez az elnevezés pontosabbnak tekinthető, mint a ma népszerű Prompt-Engineering meghatározás. A generatív modellek közül a ChatGPT került kiválasztásra, mivel ma ez fogadható el a legalkalmasabbnak emberi szintű programozásra. Az elemzés során a ChatGPT azt a feladatot kapta emberi szintű programként, hogy a tanító mintában lévő e-maileket kategorizálja, majd az eredmény összevetésre került egy szakértő által végzett kategorizálással.

A gráfadatbázisok tanulmányozásának kiindulópontja az úgynevezett tudásgráfok voltak. Tudás gráfok létrehozhatóak generatív nyelvi modellek segítségével. Ez a terület jelenleg kiaknázatlan egy ügyfélszolgálati környezetben megvalósítandó intelligens rendszer tekintetében. Ebből következően javaslatok kerültek megfogalmazásra a gráfadatbázisok ügyfélszolgálati környezetben történő alkalmazására.

## 2. Az elért eredmények bemutatása

### 2.1. A tesztek, kísérletek megvalósítási környezetének ismertetése

A SynergyUnit keretrendszer microservice architektúra alkalmazásával került megvalósításra. Maga a főrendszer egy REST API-n keresztül elérhető kiértékelő modul, amelyhez csatlakoznak a SynergyUnit keretrendszer egységei. Ezen egységek szintén REST API-n keresztül érhetőek el. A rendszer a Python programozási nyelv 3.10-es verziójában került implementálásra.

A dokumentumkategorizálás tesztjeinek elvégzéséhez Python 3.8-as verzió került felhasználásra Jupyter notebook környezetben. Konkrétan a következő könyvtárak kerültek beépítésre: numpy, seqeval, pytorch-transformers, pandas, torch, pytorch-crf, tqdm, transformers, simpy, jupyter, ipywidgets.

Generatív nyelvi modell tekintetében a ChatGPT 3.5-ös verziója került elemzésre a saját, pszeudonimizált szöveges adatok segítségével.

A gráfadatbázisok elemzéséhez a Neo4J Docker konténerben futó 4.4.19-community verziója került felhasználásra.

## **2.2. Teszt adathalmazok ismertetése**

A dokumentumkategorizálási feladathoz az adatok három forrásból kerültek beszerzésre. Az első tesztek elvégzéséhez a Szeged Dependency Treebank korpusza került felhasználásra. Ezen adathalmaz egy CONLLU-formátumban annotált magyar nyelvű szövegtörzset tartalmaz. Annotáció segítségével kerültek kijelölésre a szövegtörzsek szándékjelző címkéi.

Az algoritmus további finomításához adatok már nem álltak rendelkezésre, ezért a SONOS-NLU-BENCHMARK angol nyelvű adathalmazát magyar nyelvre adaptáltuk.

Ezen felül a Miskolci Egyetem Számítógéptudományi Intézetjével történt megállapodás alapján egy e-mail-tesztadatbázis került kialakításra. Az adatbázisba kerülő szövegeket alapos tisztításnak és előfeldolgozásnak kellett alávetni. A szöveg kategorizálására és annotálására a Doccano rendszer került felhasználásra. A szöveg elemzése a következő kategóriákat eredményezte: Invitation, PasswordRequest, TechnicalSupportRequest, SystemMessage, FollowUp, LoginIssue. Azonban az egyes kategóriák eloszlása meglehetősen egyenetlen volt, ezért az adathalmazt normalizálni kellett.

## **2.3. Kiértékelések eredményeinek bemutatása**

A SynergyUnit keretrendszer használói pozitív visszajelzéseket adtak a felhasználói élmény tekintetében. Minden esetben, amikor az SU szolgáltatásai elérhetőek voltak, akkor a felhasználók igénybe is vették azokat. Az eredmények a csoporton belül az SU használata következtében kiegyensúlyozottabbak lettek. Az eredmények tartóssága további vizsgálatok által tisztázható.

A generatív nyelvi modellel végzett szövegekategorizálási kísérlet kielégítő eredménnyel zárult. Megállapítható, hogy a ChatGPT 3.5-ös modell még magyar nyelven programozva is emberi szintű kategorizálásra és adatkiemelésre képes. A programozás során szabad szöveges formában ki lettek jelölve az egyes kategóriák és a feltárandó adatok, az eredményeket egy táblázatba rendezve kellett a ChatGPT-nek produkálnia.

Gráfadatbázisok tekintetében a visszajelzések nem egyértelműek, nem mindig érthető a felhasználók számára, hogy miért előnyös egy gráfstruktúra alkalmazása. Ennek oka lehet, hogy a felhasználók nem gyakran találkoznak ezzel a struktúrával a mindennapi életben.

## **2.4. Eredményeket szemléltető képernyőképek, adatsorok**

A SynergyUnit prototípus-implementáció egy automatikus válaszkértékelő modul, melyhez felhasználói felület kapcsolódik, ez teszi lehetővé a rendszer tesztelését.

Kérem, válaszoljon a kérdésre, vagy oldja meg a feladatot!

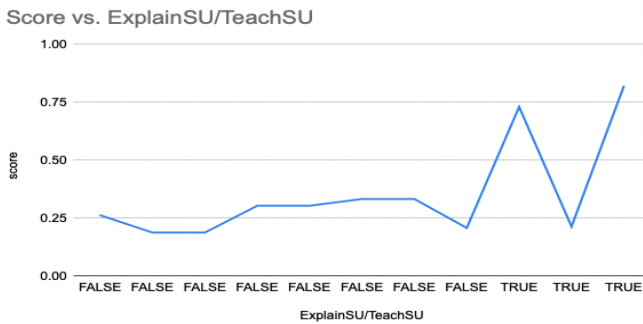
**?** How do you greet the customer?

**🗣️** Szép napot! A nevem Gee4, én fogom kiértékelni a választat...

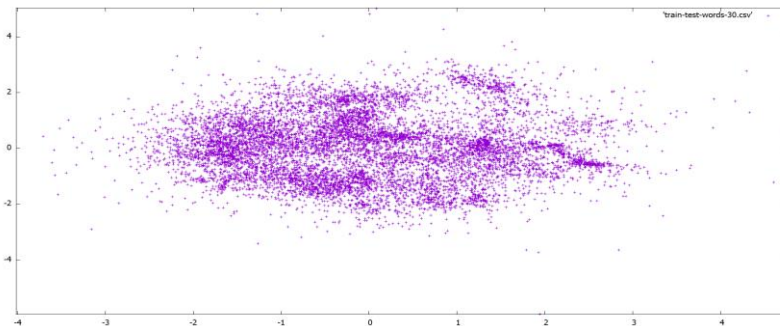
Hello, how may I help you?!

Evaluate my answer

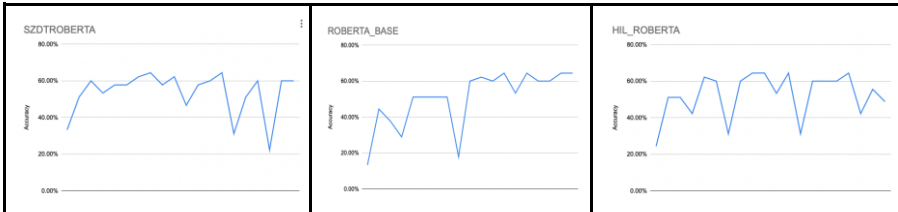
A tesztelés során megállapítást nyert, hogy az ExplainSU és a TeachSU elérhetővé tétele után (FALSE/TRUE) a tesztalanyok teljesítménye (Score) jelentősen javult. A teljesítmény gyors javulásának pontos okát azonban még további vizsgálatokkal szükséges elemezni.



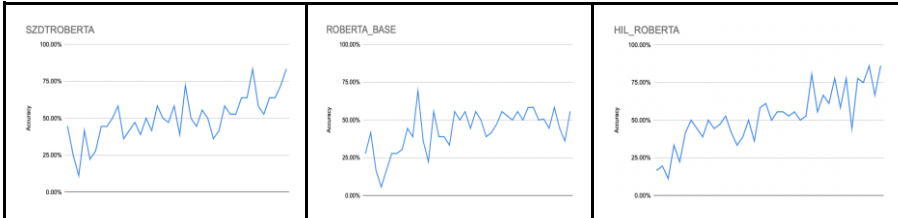
A nyelvi modell elkészítéséhez használt szóbeágyazó algoritmus (Word2Vec) eredményét kétdimenziós síkra alakítva megfigyelhető az egyes kifejezések „csoportosulása”, melyből automatikus témakijelölésre lehet következtetni. Többek között ezt a jelenséget használják ki a generatív nyelvi modellek is.



A létrehozott e-mail-adatbázison három előtanított modell (SZDTROBERTA, ROBERTA\_BASE, HIL\_ROBERTA) finomhangolása került elvégzésre. Az adatok normalizálása előtt az Accuracy az Epochok előrehaladtával 60% körülnyi maximumot ért el, viszont nem stabilizálódott.



Normalizálás után az eredmények látványosan javultak, a modell Accuracy a 80%-os szintre emelkedett és elkezdődött a stabilizálódás az SZDTROBERTA és a HIL\_ROBERTA modellek esetében. A teszt végeredményeként a legmagasabb Accuracyt a HIL\_ROBERTA érte el 86,11% értékkel.



A ChatGPT 3.5-ös verziója megfelelő prompt segítségével szövegcategorizálásra és adatkiemelésre került programozásra. Példaként tekintsük következő e-mail szövegét, majd a prompt eredményét.

*Tisztelt Cím!*

*Segítséget szeretnék kérni az egyetemi email címmel kapcsolatban.*

*Sajnos valamiért nem tudok belépni a Webmail-be miután megváltoztattam a jelszavam.*

*Ezzel kapcsolatban tudnának segíteni?*

*Neptun kódom: AA2AA3*

*Üdvözlettel:*

*Vagány Viktória*



### ChatGPT

A levél alapján a következő adatokat állapíthatjuk meg és rendezzük a táblázatba:

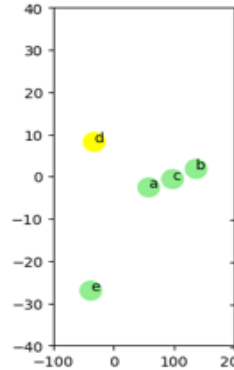
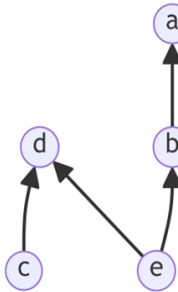
- Levél küldője: Vagány Viktória
- Neptun kódja: AA2AA3
- Levél kategóriái: PasswordRequest (mivel jelszóproblémáról tájékoztat)
- Rendszerek: Webmail
- Helyszínek: Egyetem

A táblázat tehát a következőképpen néz ki:

| Levél küldője   | Neptun kódja | Kategória       | Rendszerek | Helyszínek |
|-----------------|--------------|-----------------|------------|------------|
| Vagány Viktória | AA2AA3       | PasswordRequest | Webmail    | Egyetem    |

A teszt eredményeként megállapítható, hogy a generatív GPT-modellek, azon belül a ChatGPT 3.5 messze felülmúlja a kísérletben tesztelt BERT-modellek szövegkategorizáló teljesítményét.

A gráfadatszerkezet lehetővé teszi az ügyfélszolgálati hálózatok és kapcsolatok elemzését. Gráf beágyazási algoritmusok tesztelése során megállapítható volt, hogy a szavak témacsoportokba szerveződéséhez hasonlóan az egyes node-ok is csoportosulnak. Az ügyfelek és a hozzájuk kapcsolódó adatok segítségével kialakuló csoportok együttes kezelése javíthatja az ügyfélszolgálatok teljesítményét.



## 2.5. Kutatási eredményeken alapuló javaslatok megfogalmazása

Az EMI-kutatások további folytatása ajánlott, különös tekintettel a javasolt SynergyUnit keretrendszer tekintetében. A keretrendszer által nyújtott szolgáltatások kedvező fogadtatása a felhasználók körében azt mutatja, hogy célszerű

ezen szolgáltatásokat továbbfejleszteni, minél több rendszerbe beépíteni, valamint azok használatát népszerűsíteni.

A szövegekategorizálási feladatok elvégzése tekintetében a generatív modellek, mint a ChatGPT használata javasolt, mivel a tesztek és elemzések azt mutatják, hogy ezek eredményei felülmúlják a BERT nyelvi modellek eredményeit. Mindazonáltal kisebb célfeladatok ellátására továbbra is alkalmasak a bemutatott \*ROBERTA modellek. A ROBERTA modellek betanítása, illetve finomhangolása lényegesen költséghatékonyabb, mint a GPT modellek létrehozása. Továbbá ügyfélszolgálati környezetben a munkatársak munkáját ezek is segíthetik és gyorsíthatják, amennyiben nem cél a teljes automatizálás.

Javasolt továbbá, hogy a gráfadatbázisok és alkalmazások elterjesztését képzések és bemutatók tartásával segítsük elő a jövőben. Számos alkalmazásra nyílik lehetőség, ha sikerül a gráfadatbázisokat elfogadtatni a döntéshozókkal, illetve felhasználókkal. Néhány alkalmazási lehetőség, a teljesség igénye nélkül:

- **Ügyfeladatok kezelése:** Az ügyfelek közötti kapcsolatokat gráfként modellezve könnyen nyomon követhetők az egyes ügyfelekkel kapcsolatos adatok, például: kapcsolódó számlák, előzmények, panaszok stb.
- **Hálózati elemzések:** Az ügyfelek közötti kapcsolatokat modellezve felismerhetők azok a kapcsolatok, amelyek fontosak lehetnek az ügyfélszolgálati teljesítmény javítása érdekében, például: azok az ügyfelek, akik gyakran keresnek segítséget.
- **Támogatási útvonalak és döntési fák:** Az ügyfélszolgálati folyamatokat gráfként ábrázolva könnyebb megérteni és optimalizálni az ügyfélszolgálati munkafolyamatokat, beleértve a hívásközponti irányítást, az ügyfélkapcsolat-kezelést stb.
- **Problémamegoldás és panaszkezelés:** Az ügyfélpanaszokat és problémákat gráfként modellezve könnyebben azonosíthatók a gyakori problémák és azok az ügyfelek, akik gyakran hasonló problémákkal küzdenek.
- **Ajánlások és személyre szabott ügyfélszolgálat:** Könnyebb azonosítani azokat az ügyfeleket, akiknek hasonló igényei vannak, és személyre szabott ajánlatokat vagy megoldásokat kínálni számukra.

## Referenciák

- [1] Zhou, L., Paul, S., Demirkan, H., Yuan, L., Spohrer, J., Zhou, M., & Basu, J. (2021). Intelligence Augmentation: Towards Building Human-Machine Symbiotic Relationship. *AIS Transactions on Human-Computer Interaction*, 13 (2).
- [2] Csepeli G. (2020). *Ember 2.0 – A mesterséges intelligencia gazdasági és társadalmi hatásai*. Kossuth Kiadó Zrt, Budapest.

- [3] Gangadharaiah, R., Narayanaswamy, B., & Ai, A. (2019). Joint Multiple Intent Detection and Slot Labeling for Goal-Oriented Dialog. North American Chapter of the Association for Computational Linguistics. <https://doi.org/10.18653/v1/N19-1055>
- [4] Tan, Y., Xu, H., Wu, Y., Zhang, Z., An, Y., Xiong, Y., & Wang, F. (2021). Research on Knowledge Driven Intelligent Question Answering System for Electric Power Customer Service. *Procedia Computer Science*, 187. <https://doi.org/10.1016/j.procs.2021.04.072>
- [5] Chen, Q., Zhuo, Z., & Wang, W. (2019). *BERT for Joint Intent Classification and Slot Filling*. <http://arxiv.org/abs/1902.10909>





A kiadásért felelős:  
a Miskolci Egyetem rektora: Prof. dr. Horváth Zita  
Megjelent a Miskolci Egyetemi Kiadó gondozásában  
A kiadó felelős vezetője: Szendi Attila  
Szöveggondozás, tördelés, tipográfia, borítóterv: Gramantik Csilla  
Korrektor: Juhász Zoltán  
GÉIK, Általános Informatikai Intézeti Tanszék – 2024 – ME