ESEMÉNYSOROK GYAKORI MINTÁINAK FELTÁRÁSA GRÁFALAPÚ MÓDSZERREL

DR. RADELECZKI SÁNDOR

A projekt keretében az én feladatom általában a matematikai modellezés, ezen belül, a gráf/hálózatalapú folyamatfeltárás területén:

- *diszkrét matematikai módszerek elemzése és rendszerezése, az eseménysorok gyakori mintáinak feltárása;*
- *módszerek kidolgozása az eseménysorok gyakori mintáinak a feltárására;*
- *gráfokon alapuló modellek leírásának dokumentálása és ezeknek a módszereknek az elemzése;*
- *algoritmusok megfogalmazása, és az elvégzett elemzések dokumentálása.*

1. A kutatás célja és lépései

Első lépésként elvégeztem a témakörben fellelhető jelentősebb cikkek elemzését. Megvizsgáltam és rendszereztem az ezekben a publikációkban szereplő fogalmakat és eszközöket/módszereket. A szakirodalomban több, (egymással is kapcsolódó) módszerrel, megközelítéssel találkozhatunk, amelyeket egymással összehasonlítottam és kiemeltem közülük az alábbi csoportot:

Gráfalapú módszerek a leggyakoribb minták felismerésére – ezeknek több változata is ismert, az egyik legkorszerűbb módszer közülük a maximális méretű gyakori mintákat tárja fel egy véges parciális automata állapotgráfjának a megvalósítása által (Maximal Pattern Mining). Itt kiindulópontom a Liesaputra, V., Yongchareon, S. and Chaisiri, S.: (2016, Sept.). Efficient process model discovery using maximal pattern mining. In *International Conference on Business Process Management*, pp. 441–456), Springer, Cham. című publikáció volt.

Második lépésként rámutattam ennek az ún. MPM- (maximal pattern mining) módszernek az előnyeire a többi gráfokon alapuló módszer viszonylatában. Elemeztem az MPM-módszer szubrutinjait és implementálásra alkalmas verzióit. Ennek alapján vázoltam, hogy egy adott tevékenységfolyamnál hogyan határozható meg a fő tevékenységsor, a lehetséges elágazások, a gyakorisági paraméterek és az esetleges kivételek, illetve azt, hogyan becsülhető meg a módszer „pontossága". Megvizsgáltam, hogy a megadott algoritmus hogyan finomítható tovább és kiegészítettem az utóbbi néhány év erre vonatkozó irodalmával.

Részletesen megvizsgáltam a módszer lehetséges értékelési kritériumait (beleértve a pontosságot) és ezeknek a mérőszámait. Megvizsgáltam, hogy hogyan finomítható tovább úgy (MPM) algoritmus (lásd [2]), hogy a hurkok hosszát is nyilvántartsa és egyszerűsítse az eredményként kapott tranzíciós gráfot.

Elvégeztem az MPM-algoritmus néhány (az irodalomban nem kellően részletezett) része implementációjának az előkészítését, az eljárások leírásával és pontosításával. Megadtam az eljárás egyes lépéseihez tartozó szubrutinok pszeudokódjait és a keretalgoritmus pszeudokódját. Módszert fogalmaztam meg optimális megoldások megkeresésére. Ezt követően implementáltam a teljes algoritmust.

Kipróbáltam az MPM-algoritmus szubrutinjainak és magának a teljes algoritmusnak is a működését, lefuttatva egyszerűsítek példák egy kisebb halmazán. Az egyes szubrutinokat külön-külön elemeztem.

2. Kutatási eredmények összesítése

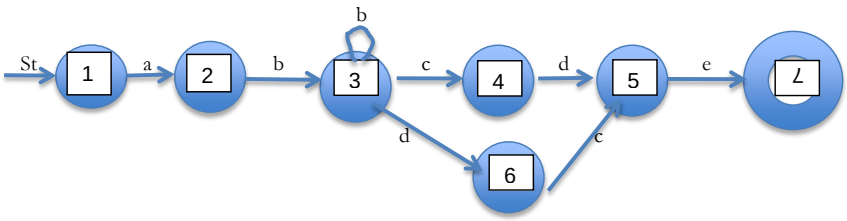
2.1. A bemutatott szakirodalom elemzése, értékelése, alapfogalmak és módszerek kiemelése

Egy vállalat belső ügyfélszolgálati rendszerének egy cselekményét *nyomnak (trace)* vagy *akciónak (action)* nevezzük. Egy t *nyom (trace) események (events)* egy véges és koherens $z_0, z_1, \ldots, z_m$ sorozata, amit a $t = (z_0, z_1, \ldots, z_m)$ formában jelölünk. Az egyes z_i eseményeket vektorként ábrázoljuk, ami kötelező komponensként kell tartalmazza az esemény *típusát (type)*, valamint *időbélyegét (timestap),* más egyéb adatok mellett. A t nyomon belül az egyes eseményeket időbélyegük szerint rendezzük sorba. Egy t nyomban csak az őt alkotó események típusait ábrázoljuk, pl. t = (a, b, b, c, e, d). Így egy véges ábécé feletti szavakat kapunk (ahol az „ábécé" az eseménytípusok halmaza), egy T = {(a, b, c, b, b, c, d, e), (a, b, b, c, b, c, d, e), (a, b, b, c, e, d)} eseménynaplót pedig egy (véges ábécé feletti) formális nyelvnek (vagy egy ilyen nyelv egy részletének) tekinthetünk. Egy t nyom *hossza*, amit |t|-vel jelölünk, nem haladhat meg egy előre megadott korlátot.

Feladatunk most úgy fogalmazható meg, hogy egy lehetséges eseménynaplóban egyrészt ellenőrizni akarjuk a cselekmények nyomait, megkülönböztetve őket a *zajtól* (ami itt egy nyom eseményeinek hibás vagy hiányos rögzítéséből adódhat), illetve olyan nyomokat szeretnénk automatikus módon generálni, amik valódi ügyfélszolgálati eseményeknek felelnek meg.

Az irodalomból itt a feladat megoldására két lehetséges módszert (megközelítést) emeltem ki:

1) Az első megközelítés esetén (α++- algoritmus, MPM [maximal pattern mining] algoritmus, OSTIA) a nyomok hasonló szerkezetű csoportjaihoz egy tranzíciós gráfot rendelünk – ezt egy címkézett és irányított gráfnak tekinthetjük és *tranzakciós mintának (transaction pattern)* nevezzük. A MPM-algoritmus esetén először maximális mintákat választunk ki, amik részgráfként tartalmazzák a többi mintát és ehhez rendeljük egy véges determinisztikus parciális automata tranzíciós gráfját



2.1. ábra. *$T = \{(a, b, c, d, e), (a, b, d, c, e), (a, b, b, c, d, e)\}$ mintához tartozó véges automata*

2) A második megközelítés esetén megerősítő tanuláson alapuló szövegfelismerést és szöveggenerálást alkalmaznak, ami egy összetett, rekurens neuronhálózat segítségével valósul meg. A tanulás során egy előre kiválasztott tanítóhalmaz alapján először a leggyakoribb nyomokkal megegyező szerkezetű nyomokat állítjuk elő, majd fokozatosan a ritkább előfordulású nyomokat is „reprodukáljuk".

A gyakorlatban az ügyfélszolgálati rendszer a beérkező valós *kérésekre (request)* kell (megoldásokat) válaszokat adjon. Egy ilyen r kérés (igénylés) maga is egy vektor formájában adható meg, ami tartalmazza a kérés típusát, az igénylő azonosítóját, a beérkezés (iktatás) idejét, valamint a kéréssel kapcsolatos szignifikáns adatokat. Az ügyfélszolgálati rendszer erre egy eseménysorral válaszol, ami az eseménynaplóban egy nyomként szerepel. Itt a legfontosabb információ a nyom „mintája", ami az első esetben egy véges automatának, az automatikus szövegfelismerés esetén pedig „rokon értelmű" szavak egy halmazának felel meg. Az ERPA öntanuló rendszer tanítóhalmazában tehát (r_i, t_i) kérés-nyom pároknak kell szerepelniük. Hogy ez elkészüljön, ezt meg kell előzze a beérkező kérések megfelelő (dimenzió-csökkentett) formában való rögzítése és különböző szempontok szerinti automatikus osztályozása.

Az első megközelítés esetén, a tanítóhalmaz alapján az (r_i, t_i) párokhoz rendelt véges automaták „finomhangolása" végezhető el, vagyis a tanítóhalmazban szereplő példák alapján, a t_i mintához tartozó véges automatának a tranzíciós diagramjában a programnak meg kell találnia a kérést megválaszoló legjellemzőbb útvonalat. A második megközelítés esetén a tanítóhalmazban szereplő (r_i, t_i) párok alapján az automatikus szöveggeneráló rendszernek elő kell állítania a leginkább jellemző „kérdés-felelet" típusú szövegrészleteket. Ezeknek a tökéletesítése megerősítő tanulás által érhető el. A továbbiakban az MPM-módszerre fókuszáltam.

2.2. Az MPM-módszer bemutatása és elemzése.

1. Az eseménynapló alapján a fő tevékenységsorokat úgy kaphatjuk meg, hogy a kibányászott maximális gyakoriságú sémákból a hurkokat töröljük. (Az ismétlődő szekvenciák nem változtatnak a folyamat természetén.) A gyakoriság figyelembevétele azért fontos, hogy a zajt és a kivételes eljárásokat kiszűrhessük. A nyomokban szereplő hurkok kiküszöbölése két lépésben történik:

a) az ismétlődő típusokat (karaktereket) egyetlen karakterrel helyettesítjük. A fenti példa esetén az (a, b, c, b, c, d, e), (a, b, c, b, c, d, e), (a, b, c, d, e) kifejezéseket kapjuk.

b) Ezután az ismétlődő karaktercsoportokat egyetlen csoporttal helyettesítjük, itt például: (a, (b, c, d, e), (a, (b, c, d, e), (a, b, c, d, e). Az utolsó kifejezés nem változik, az első kettő pedig ugyanaz lévén, elég egyszer feltüntetni, így kapjuk, hogy: (a, (b, c, d, e), (a, b, c, d, e). A * jeleket (az exponenseket) elhagyjuk a kapott kifejezésekből és megadjuk a hurkok helyét (végét) és a bennük lévő betűket. Így megkapjuk a T csoporthoz tartozó ún. *sémát* (pattern) – ez mindhárom esetben: $p = \langle a, b\langle b\rangle, c\langle bc\rangle, d, e\rangle$. A **p** séma tartója (support-ja) nem más, mint a **p**-hez tartozó nyomok száma: ez most 3.

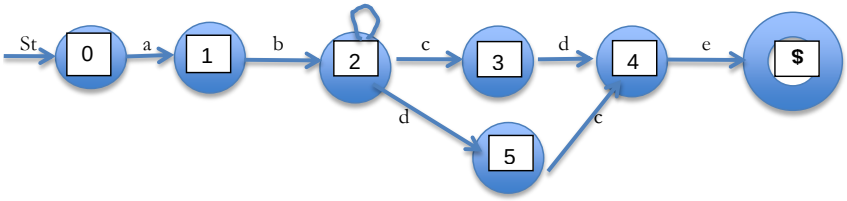
2. A gyakori műveletsorok (eseménysorok) feltárása az MPM-eljárásban két küszöbérték (trashold érték) megadásával lehetséges, úgy ahogy azt a korábbi összefoglaló jelentésben bemutattuk, felhasználva a nyomok ún. „függőleges bitmap reprezentációját". Ennek a reprezentációnak a segítségével a sémák listáján való egyszeri végighaladással leolvasható az egyes karakterek előfordulási számai és azokat a karaktereket (típusokat), amelyeknél ezek relatív előfordulása nem haladja meg a trash szám értékét, töröljük az eseménynaplóból azokkal a sémákkal és nyomokkal együtt, amelyekben előfordulnak. Ahhoz, hogy a kivételes eseménysorokat a zajtól megkülönböztessük, szükségünk van mind a bejövő kérések ismeretére, mind egy nagy elemszámú validációs halmazra. Ezután a gyakori sémák kiválasztása két különböző módon is történhet. Az egyik kézenfekvő módszer az egyes sémákhoz vezető nyomok tárolása. Végighaladva a sémák listáján, minden egyes séma esetén összeszámoljuk azokat a nyomokat, amiket az adott sémák „lefednek". Ha egy séma esetén ezeknek a relatív száma meghaladja a trash értéket, akkor a sémát megtartjuk, ellenkező esetben a sémát és a hozzátartozó nyomokat töröljük. Egy másik módszer egy az idézett cikkben megadott *Expand subroutin* ami a gyakori karakterekhez iteratív módon mindig egy-egy karaktert hozzácsatolva generálja

a gyakori sémákat. Az ebben a 2. lépésben szereplő két trash értéket egy egyszerű tanulási folyamat során tudjuk optimálisan beállítani.

3. A módszer két különböző típusú lehetséges elágazást tud felismerni. Ha közös kezdő és végső részszekvenciával rendelkező eseménysorok, például $p_1 = us_1v$, $p_2 = us_2v$, $p_3 = us_3v$, $p_4 = us_4v$ csak egy-egy részszekvenciában különböznek, azt egy formális összegként az $u(s_1 + s_2 + s_3 + s_4)v$ reguláris kifejezéssel írhatjuk le, ami annyit jelent, mint ha az $s_1, \ldots, s_4$ részszekvenciákat a logikai *vagy* (diszjunkció) kötné össze és ezt grafikusan egy (négyes) formális elágazással szemléltethetjük.

(1) Ez akkor jelent valóban opcionális helyzetet, ha a $p_1, \ldots, p_4$ sémák ugyanarra a kérésre adott válaszok (nyomok) összeségét jelentik. Ekkor a $p = (u, XOR(s_1, s_2, s_3, s_4), v)$ sémát kapjuk, ami valódi elágazást jelent.

(2) Párhuzamos helyzet. Egy másik sajátos eset az, ha $s_1, \ldots, s_4$ olyan szekvenciák, amelyek egy karaktercsoport permutációi, méghozzá úgy, hogy bármely két karakter fordított sorrendben is megjelenik a példákban. Ilyen például az $s_1 = (a, b, c)$, $s_2 = (a, c, b)$, $s_3 = (b, a, c)$, $s_4 = (c, b, a)$ eset. Ekkor a mintákból arra következtetünk, hogy az a, b, c események sorrendje irreleváns és azok párhuzamosan is végrehajthatók. Ekkor az $(u, \{a, b, c\}, v)$ sémát kapjuk eredményül, ami rajzban szintén elágazással szemléltethető. Például az (a, b, c, d, e) és az (a, b, d, c, e) sémák esetén a c és d események tetszés szerinti sorrendben elvégezhetők. Ezért mind a kettő valójában egyetlen folyamathoz tartozik, aminek a sémáját így jelöljük: (a, b, {c, d}, e). A {c, d} halmazjelölés arra utal, hogy a c és d események sorrendje mellékes. Más jelölés: (a, b, ∧c, d), e). A sémának megfelelő véges parciális automata pedig az alábbi.



2.2. ábra. *Az (a, b, b, c, d, e) és (a, b, d, c, e) nyomokhoz tartozó véges automata*

Az alapcikkben a párhuzamosságok feltárását egy Solve Concurrency nevű algoritmus végzi. Az algoritmus csoportosítja azokat a sémákat, amelyek egy közös

prefixszel, sufixszel, vagy mindkettővel rendelkeznek és kiemeli a nem közös részt, amennyiben azon belül nincs ismétlődő csoport

4. Negyedik lépésként az algoritmus az ún. maximális sémákat keresi ki és listázza. Az mondjuk, hogy a p_1 sémát *lefedi* a p_2 séma, ha minden, a p_1 sémával előállítható nyom, a p_2 sémával is megkapható. Így például az (a, b, c, d, e) és az (a, b, d, c, e) sémák lefedhetők az (a, b, {c, d}, e) sémával, de az (a, b, c) séma nem (noha a hozzátartozó gráf részgráfja lenne az (a, b, {c, d}, e) sémához tartozó tranzíciós gráfnak). Egy T csoporton belül egy p séma *maximális*, ha nincs olyan tőle különböző q séma T-ben, ami p-t lefedi. (Tehát nincs olyan q séma, ami mindazokat a nyomokat előállítaná, amit p – és esetleg még azon felül mást is.) A Resolve szubrutin ebben a lépésben kettesével összehasonlítja a tanítóhalmazból kinyert, egy IdList nevű listán szereplő p_i és p_j sémákat egy kettős iteráció során. Ha a p_j séma lefedi a p_i-t akkor p_i helyére p_j-t ír és p_i-t törli. Ezután folytatja p_j összehasonlítását az eddig nem vizsgált sémákkal. Az eljárás végén az algoritmus listázza a megmaradt sémákat, amelyek már mind maximálisak.

5. Az utolsó lépésben az algoritmus megrajzolja az eseménynaplóban adott munkafolyamathoz tartozó *tranzíciós gráfot (folyamatgráfot)*, amit a (kibányászott) maximális sémákhoz tartozó parciális véges automaták tranzíciós gráfjainak az uniójaként kap meg. A kapott véges automaták kezdőállapotait egyetlen kezdő (Start) állapotban, végállapotait egyetlen, $-ral jelölt végállapotban egyesíti.

2.2.1. A zaj *kiküszöbölése tanulással és az* „eredményesség" mérése

Azokat az eseménytípusokat, illetve azokat a sémákat, amelyek gyakorisága a megfelelő trash értéket nem haladja meg, *zajnak* tekintjük és kiküszöböljük. A trash paraméterek „optimális" beállítása úgy történik, hogy a megadott nyomok M összeségét két diszjunkt (nem üres) halmazra, egy ún. T *tanítóhalmazra* és egy V *validációs* vagy *teszthalmazra* osztjuk. Az algoritmus tesztelése során előállított nyomok halmazát R-rel jelöljük. Az ún. eredményesség mérésére az algoritmus több mérőszámot is használ. Itt a legfontosabbat ismertetjük:

Az egyik az ún. *Fitness* vagy *Recall* mérőszám (*Érvényesség, Visszaidézés*) – ez nem más, mint a helyesen előállított nyomok számának (tehát az $R \cap V$ halmaz elemszámának) és az eseménynaplóban szereplő nyomok M számának az aránya, vagyis

$$\Re := \frac{|R \cap V|}{|M|}$$

A *Precision („Pontosság")* mérőszám a helyesen előállított nyomok $(R \cap Vhalmaz)$ elemszámának és az összes előállított nyom (R) elemszámának az arányát jelenti:

$$P := \frac{|R \cap V|}{|R|}$$

Az ún. *F-mérték (F-measure)* egy, a két előbbi mutatót összesítő globális mutató, ami tulajdonképpen *Re* és *P* harmonikus középarányosa. Ezért

$$F_m := \frac{2 * P * \Re}{P + \Re}$$

Egy ritkábban használt mérőszám az ún. Eltérés vagy Különbség (Difference), ami az *R* és *V* szimmetrikus különbségének és (R) elemszámának a hányadosa.

$$D := \frac{|R \Delta V|}{|R|}$$

Természetesen ezek a mérőszámok nem függetlenek egymástól. Egy másik mérőszám az a **t** *idő*, ami alatt az (MPM) algoritmus felállít egy folyamatmodellt (valójában az egyszerűsített tranzíciós gráfot). A trash paraméterek „optimális" beállítása úgy történik, hogy a megadott nyomok M összeségét két diszjunkt (nem üres) halmazra, egy ún. T *tanítóhalmazra* és egy V *validációs* vagy *teszthalmazra* osztjuk. Az MPM-algoritmus a tanítóhalmaz alapján előállítja a maximális sémákat. Aztán megvizsgálja, hogy az így kapott sémák által előállított nyomok R halmaza milyen mértékben egyezik meg a V teszthalmazzal. Ha a $V - R$ halmazkülönbség relatív elemszáma nagy, akkor a trash értékeket csökkentjük. Ha az $R - V$ halmazkülönbség relatív elemszáma nagy, akkor a trash értékeket növeljük. Ezt mindaddig folytatjuk ameddig az R és V halmazok a lehető legkisebb számú elemben különböznek. Valójában az MPM-eljárás a trashold értékek beállítására és a fenti mérőszámok kiszámítására, az ún. *k-szoros validációt* használja. Ez azt jelenti, hogy a rendelkezésünkre álló nyomhalmazt k drb. diszjunkt (és nagyjából azonos elemszámú) részhalmazra bontjuk (Itt $k \geq 3$). Először az 1. számú részhalmazt jelöljük ki tanítóhalmaznak, a többiek uniója pedig a teszthalmaz lesz. Ezután a 2. részhalmaz lesz a tanítóhalmaz és a többiek uniója a teszthalmaz, …, ezt

mindaddig folytatjuk ameddig mind a k részhalmaz sorra kerül, mint tanítóhalmaz. Minden egyes lépésnél meghatározzuk az ott adott felosztásra vonatkozó mérőszámokat. Ezek számtani középarányosaiként kapjuk meg az egész eljárásra vonatkozó mérőszámokat (jelen esetben a Fitness és a Precision mérőszámokat).

2.2.2. A kivételek meghatározása

A kivételek olyan kis gyakorisággal előforduló minták, amik megkülönböztethetők a zajtól. Jó eséllyel a beérkezett kérések (requests) ismeretében különíthetőek el, amelyek jól megkülönböztethetőek kell legyenek a többi kéréstől. A tanulási folyamat során ellenőrizzük, hogy a validációs halmazban a kitüntetett requesteknek megfelelő nyomokat megkapjuk-e, ha nem, akkor a tanulási folyamatot tovább folytatjuk.

2.3. Az elemzések eredményeinek bemutatása

2.3.1. Az ismétlődések reprodukálása

Amint azt már korábban mondtuk, az MPM-eljárás első lépéseként kiküszöböljük a nyomokban szereplő hurkokat. A [2] cikkben az ismétlődő típusok vagy típuscsoportok jelölésére zárójeleket javasolnak. Például az $a, b\langle b\rangle, c, d$ jelölés azt jelenti, hogy a b típusú esemény többször is ismétlődhet, de legalább egyszer elő kell forduljon. Ha $a, \langle b\rangle, c, d$ -t írunk, akkor b hiányozhat is. A $p = \langle a, b\langle b\rangle, c\langle bc\rangle, d, e\rangle$ minta esetén a bc csoport tetszőleges sokszor fordulhat elő. A formális nyelvek jelölését használva az ismétlődés pontos számát is jelölni és a memóriában tárolni is tudjuk, pl. (a, b^2, c, d, e). Általában ez az ismétlődési szám egy ügyviteli folyamatban nem haladhat meg egy előre megadott felsőkorlát-számot, ami legtöbbször egy kis szám. Egy ismétlődő eseménytípus (pl. $\langle b\rangle$) esetén ez viszonylag könnyen megoldható valamilyen „klaszterezési" eljárással a következőképpen: Minden (a treshold értéket meghaladó gyakoriságú) t_i nyom esetén tároljuk az ismétlődő eseményhez (itt pl. a b előfordulásaihoz) tartozó számszerű paramétereket egy $f_i = \langle p_1, .., p_{k(i)}\rangle$ vektorban. Ez pontosabban azt jelenti, hogy minden olyan (a, b, c, d, e), (a, b^2, c, d, e), (a, b^3, c, d, e) nyom esetén (aminek a gyakorisága nagyobb a treshold értéknél) annyi $\langle p_1, .., p_{k(i)}\rangle$ alakú vektort tárolunk amennyiszer az adott nyom (pl. (a, b, c, d, e)) előfordul, majd valamilyen ismert osztályozási eljárással, pl. K-mean, Support vector machine, az 1, 2, 3,… előfordulásoknak megfelelő diszjunkt csoportokra osztjuk őket. Így egy újabb beolvasott nyom esetén már megvizsgálhatjuk, hogy a b-hez tartozó paramétervektor besorolható-e valamelyik osztályba – a válasz akkor lesz igen, ha az új paramétervektor távolsága az adott osztály átlagától nem haladja meg az

osztályon belül az átlaghoz (ami valójában a klaszter középpontja) viszonyított távolságok közül a legnagyobbikat. Egy ismétlődő csoport (pl. $\langle bcd \rangle$) esetén, annyival bonyolultabb a helyzet, hogy itt mind a b, mind a c, és mind a d eseményekhez tartozó paraméter-vektorokat ismernünk kellene és belefoglalni őket egyetlen paraméter-vektorba; természetesen nem minden paraméter fogja az ismétlődést befolyásolni, ezért a paraméterek számának túlzott növekedését a nem releváns paraméterek kiküszöbölésével csökkenthetjük. (Az algoritmus futási ideje ugyanis a vizsgált vektorok hosszával hatványozottan nő!) Erre, valamilyen egyszerű statisztikai eljárás lehetne alkalmas, például a bcd csoport előfordulásainál megvizsgálnák az egyes paramétereknek a várható értékét és szórását. Ahol a teljes ismétlődéssorozatra kapott szórás „nagy" lenne – vagyis a várható értékhez hasonló nagyságú lenne, azokat a paramétereket nem vennénk figyelembe és nem írnánk be a paramétervektorba. Ez az eljárás természetesen továbbfinomítható, ha már az „alapprogram" elkészül és működik.

Egy másik, itt most csak megemlített lehetőség, az a helyes paraméter-vektorok neurális hálókon alapuló „megtanulása" lenne, vagyis annak a megerősített tanulással való behatárolása, hogy egy új paramétervektor az esemény, vagy az esemény-csoport hány ismétlődéséhez kapcsolható hozzá. Itt eredményesen alkalmazhatnánk a belső jelentésekben bemutatott ún. bitmapes reprezentációját az egyes nyomoknak.

2.3.2. Optimális útvonalak keresése

Ahhoz, hogy egy tranzíciós gráfon belül ún. optimális útvonalakat jelöljünk ki, tehát döntsünk az esetleges ismétlődések számáról és az opcionális részszekvenciák kiválasztásáról, szükséges lenne a kérések és a kibányászott minták között kapcsolatot ismernünk. Ez elérhető akkor, ha a tanítóhalmazban kérés-nyom párok szerepelnek – ahol a nyom a kérésre adott válaszhoz tartozik. Ekkor egy adott kéréshez egy célfüggvényt rendelve, rendezhetjük a hozzá tartozó (válaszként kapott) nyomokat – és kiválaszthatnánk az optimális megoldást jelentőt közülük.

2.3.3. A folyamatgráf megrajzolása és egyszerűsítése

Az eljárás végén kapott tranzíciós gráfot (az opcionális és párhuzamos szekvenciák meghatározásával és a kivételes minták elkülönítésével) egyszerűsíthetjük. Az eljárás végén a tranzíciós gráfot (folyamatgráfot) úgy rajzoljuk meg lépésről lépésre, hogy a kinyert maximális mintákhoz (sémákhoz) parciális véges determinisztikus automaták tranzíciós diagramjait rendeljük, majd az eljárás végén ezeket egyetlen gráfban vonjuk össze, azaz egyesítjük. Ez a gyakorlatban a Python programhoz tartozó könyvtárak közül a „pytranzition" könyvtárban található rajzprogram

segítségével történik, ami alkalmas automaták tranzíciós diagramjainak és Petri-hálók diagramjainak a megrajzolására is: Pytransitions függvénykönyvtár és útmutató: https://github.com/pytransitions/transitions/blob/master/README.md

2.3.4. Eredményesség mérése

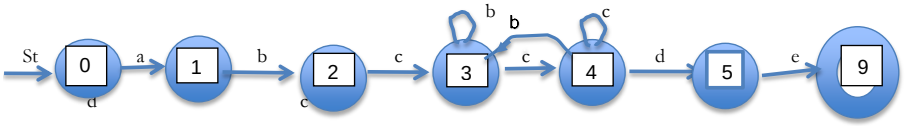
Az algoritmus „eredményességének" mérésére több (egymástól nem független) mutató is alkalmazható. Itt négy ilyen mérőszámnak (Érvényesség, Pontosság, F-mérték, Különbség) is megadtuk a pontos matematikai alakját. Itt is elmondható, hogy az eredmények helyes kiértékelése a valódi ügyviteli folyamatot megadó tanítóhalmazok alapján végezhető el hatékonyan.

3. Az MPM-eljárás technológiája: a szubrutinok leírása és pszeudokódjaik megadása

3.1. Az ismétlődések (hurkok) feltárása

Ez az eljárás a tevékenységnaplóban megadott nyomokban meghatározza az ismétlődéseket. Bemenete a vizsgálandó nyom, kimenete pedig a keletkező minta, **(pattern)**, amelyben az ismétlődések (hurkok) jelölve vannak. A ciklusok feltárását a SOLVE_LOOP névvel hivatkozott eljárás végzi.

Például az $\langle a, b\langle b\rangle, c\langle bc\rangle, d, e\rangle$ minta esetén a *bc* csoport többször is előfordulhat, vagy egyszer sem. Egy nyomot események egy sorozatának tekintünk, amelyeket 0-val kezdve indexelünk. A nyom egy elemére az indexével hivatkozunk. Például a $t = \langle a, b, c, b, b, b, c, b, c, b, c, c, d, e\rangle$ nyom esetén $t[4] = b$. A nyom egy szakaszát elejének és végének az indexét megadva jelöljük és közéjük két pontot teszünk, pl. $t[1..4] = \langle b, c, b, b\rangle$. Először az ismétlődő karaktereket szűrjük ki, $\langle a, b, c, \langle b\rangle, c, b, c, b, \langle c\rangle, d, e\rangle$**,** majd az ismétlődő csoportokat is megjelölve az ismétlődés helyét $\langle a, b, c, \langle\langle b\rangle, \langle c\rangle\rangle, d, e\rangle$**.** Most az ismétlődéseket, vagyis a $\langle \quad \rangle$ zárójeleket elhagyva, megkapjuk a mintát $p = \langle a, b, c, b, c, d, e\rangle$ ami a nyomhoz tartozik. Ahhoz, hogy ennek alapján elkészítsük a nyomnak megfelelő tranzíciós gráfot – azaz egy véges automatát, az ismétlődések pontos helyét is tárolnunk kell. Itt ezek helyét az eredményül kapott *p* mintára vonatkozó indexeléssel adjuk meg. Ezt egy intervallumhalmazzal adjuk meg, ahol egy intervallum kezdő- és végpontja azt mutatja, hogy hányadik elemtől hányadik elemig ismétlődnek az elemek a mintában (sémában). Az előző példa esetén ez az intervallumhalmaz a következő: $\{(3,3), (4,4), (3,4)\}$. Az ennek megfelelő automata az alábbi parciális véges determinisztikus automata lenne.



Az ismétlődések kijelölésekor arra kell ügyelnünk, hogy elkerüljük azt ún. *ütköző* intervallumokat. Az ütközésmentes intervallumokat úgy definiáljuk, hogy egy intervallum vagy teljesen benne van egy másikban, vagy egyáltalán nincs közös elemük. Például, az $\{(1,4),(2,4),(5,7)\}$ halmaz ütközésmentes, viszont az $\{(1,4),(3,5),(5,9)\}$ intervallumhalmazban két ütközés is található.

Az algoritmus először az 1-es hosszúságú ismétlődéseket keresi, tehát amikor egy karakter ismétlődik, majd ennek eredményén a 2-es hosszúságúakat, és így tovább, egészen addig, amíg meg nem haladja a hossza a minta méretét. A fenti példában először az $\langle a,b,c,\langle b\rangle,c,b,c,b,\langle c\rangle,d,e\rangle$ eredményt kapjuk, majd a 2-es hosszúságú ismétlődésekre az $\langle a,b,c,\langle\langle b\rangle,\langle c\rangle\rangle,d,e\rangle$ eredményt. Az eljárás egy ún. *„ablakot”* mozgat balról- jobbra az adott nyomon (azaz előző iterációban kapott mintán), és azt vizsgálja, hogy az ablak tartalma ismétlődik-e közvetlenül utána. Ha igen, az ismétlődő részt csak egyszer veszi fel az eredménymintába, illetve felveszi az ablakot az ismétlődések halmazába. Itt az $a,b,cprefixbőlabc$ csoport azért nem került be az ismétlődő csoportba, mert az eredeti nyomban az ismétlődés nem közvetlenül a c karakter után kezdődött, hanem eggyel később (a 3-as index után).

SOLVE_LOOP(trace)

```
pattern = []
loops = ∅
FOR len in {1, …, trace.length}
        newPattern = []
        newLoops = ∅
        FOR i in {0, …, trace.length - 1}
                window = trace[i .. i+len-1]
                currentLoop = NULL
                firstIter = true;
                ni = i + length
                WHILE true
                        next = trace[ni .. ni+len-1]
                        checkInterval = (ni, ni+len-1)
                        tloops = ∅
                        FOR loop in loops
                                IF loop ⊆ checkInterval AND currentLoop ≠ NULL
                                        tloop = translate loop from
```

```
                                                    checkInterval into currentLoop
                                                IF tloop does not collide with any "subloops" of
                                                currentLoop
                                                        tloops = tloops ∪ {tloop}
                                    IF window = next
                                            IF firstIter
                                                    newPattern.add(window)
                                                    newLoop = (newPattern.length – len,
newPattern.len - 1)

                                                    newLoops = newLoops ∪ {newLoop} ∪
                                                    {tloops}
                                            currentLoop = newLoop
                                            nextIndex = nextIndex + len
                                            firstIter = false
                                    ELSE
                                            IF firstIter
                                                    newPattern.add(current[0])
                                                    i = i + 1
                                            ELSE
                                                    i = nextIndex
                                            EXIT WHILE
        pattern = newPattern
        loops = newLoops
    RETURN pattern, loops
```

3.2. A zaj(ok) kiküszöbölése

Az eljárás az eseménynaplóban található nyomok közül eltávolítja a küszöbértéket meg nem haladó relatív gyakoriságú eseményeket, illetve ezután a ritka nyomokat (tehát a zajnak tekintett karaktereket és nyomokat). A ritka események eltávolítása után végighaladunk a nyomok listáján és minden egyes mintát minden nyommal összehasonlítunk, megvizsgálva, hogy illeszkedik-e a nyom az adott mintára. Minden lépésnél, tehát minden minta esetén tároljuk azokat a nyomokat, amik erre a mintára illeszkednek (tehát előállíthatók a mintához tartozó véges automatával) és a számukat is. Megvizsgáljuk, hogy melyek azok a minták, ahol ennek a számnak és az összes nyom számának az aránya meghalad egy előre megadott α küszöbszámot, azaz trash értéket (például $\alpha = 0{,}01$ - et). A küszöbszám értékének a beállítása valójában a tanulási folyamat során a validációs halmaz nyomait felhasználva történik meg. A trash értéket nem elérő relatív gyakoriságú mintákat zajként kezeljük és töröljük a minták listájáról.

```
1.  REMOVE_NOISE(eventlog, thresh)
2.          occurences = []
3.          result = copy of eventlog
4.          FOR each trace in eventlog
5.                  FOR each event in trace
6.                          occurences[event] = occurences[event] ∪ {trace}
7.          FOR each event in occurences.keys
8.                  IF occurences[event].count / eventlog.count < thresh
9.                          result = result \ { trace in eventlog | trace contains event }
10.         RETURN result
```

3.3. Vizsgálatok

3.3.1. Párhuzamosságok feltárása (a minták között)

Az eljárás bemenete két minta (p_1 és p_2), amelyek között az algoritmus azonosítja a párhuzamosságot. Amennyiben nincs köztük lehetőség párhuzamosság jelölésére, akkor az algoritmus visszatér mindkét mintával. Ha a két minta között van párhuzamosság, akkor egyetlen egyesített mintával, mint eredménnyel tér vissza. Az eljárás a következő fő lépéseket tartalmazza:

- Megállapítjuk, hogy a minták milyen indexig egyeznek meg az elejéről nézve.
- Megállapítjuk, hogy a minták milyen indexig egyeznek meg a végéről nézve.
- Ellenőrizzük, hogy a két index között ugyanazok az elemek szerepelnek-e, vagy az egyik elemhalmaz része-e a másiknak.
- H nincs a két index közt elem, vagy az ott szereplő elemekre a fenti feltétel nem teljesül, akkor nem beszélhetünk párhuzamosságról.
- Ha ugyanazok az elemek szerepelnek a két index között (más sorrendben), vagy az egyik elemhalmaz része-e másiknak, akkor megvizsgájuk, hogy bármelyik két karakter, illetve a két csoport sorrendje felcserélhető-e. Ha nem, mert nincs minden inverzióra példa, akkor nincs párhuzamosság sem.
- Ha a válasz igen, akkor megállapítjuk a párhuzamosság tényét és egy kombinált mintát hozunk létre, amellyel visszatérünk az eljárásba.

A mintában a párhuzamos elemeket egy halmazzal jelöljük – ezzel kifejezve, hogy a sorrendjük nem számít. A már említett $p = \langle a, b, c, d, e \rangle$ és $q = \langle a, b, d, c, e \rangle$ minták esetén van, a kapott eredmény pedig $\langle a, b, \{c, d\}, e \rangle$ lenne.

```
1.  SOLVE_CONCURRENCY(p, q)
2.      IF p.length ≠ q.length
3.          RETURN p, q
4.      start = -1
5.      end = p.length
6.      FOR i in {0, 1, … p.length - 1}
7.          IF p[i] ≠ q[i]
8.              start = i – 1
9.              EXIT FOR
10.     FOR i in {p.length-1, p.length-2, …, 0}
11.         IF p[i] ≠ q[i]
12.             end = i + 1
13.             EXIT FOR
14.     IF start > end
15.         RETURN p, q
16.     FOR i in {start+1 .. end-1}
17.         cp = 0
18.         cq = 0
19.         FOR j in {start+1 .. end-1}
20.             IF p[i] = p[j]
21.                 cp = cp + 1
22.             IF p[i] = q[j]
23.                 cq = cq + 1
24.         IF cp ≠ cq
25.             RETURN p, q
26.     RETURN ⟨p[0], …, p[start-1], {p[start], …, p[end]}, p[end+1], …, p[p.length-1]⟩
27.
```

3.3.2. Opcionális vizsgálata (a minták között)

Az eljárás bemenete az előzőleg kapott minták listája. Az eljárás először minden minta-párra megállapítja, hogy hány egyezés van az elejükön (prefixükön) és a végükön (szufixükön). Ahol az egyezések száma a legnagyobb, azt a két mintát egyesíti úgy, hogy az egyező részeket az elején és a végén meghagyja, a köztes részt pedig összekapcsolja XOR-ral. A két mintát eltávolítja a listából, és az egyesített változatot hozzáadja, majd rekurzívan az algoritmus újra indul a kapott mintalistára. Ha a minták listája egyelemű, nincs mit egyesíteni, az algoritmus megáll. Ha nincs egyezés a listában szereplő minták között sehol, az összeset összekapcsolja egy XOR-ral (kijelölve egyetlen kezdő- és végállapotot). Így az algoritmus eredménye mindenképpen egyetlen minta lesz.

```
1.  SOLVE_OPTIONALITY(pList)
2.      IF pList.length = 1
3.          RETURN pList
4.      starts = [][]
5.      ends = [][]
6.      FOR i in {0, 1, … p.length - 1}
7.          FOR j in {i + 1, I + 2, …, p.length - 2}
8.              maxindex = min(pList[i].length, pList[j].length)
9.              FOR k in {0, …, maxindex - 1}
10.                 IF pList[i][k] ≠ pList[j][k]
11.                     starts[i][j] = k
12.                     EXIT FOR
13.             FOR k in {maxindex - 1, …, 0}
14.                 IF pList[i][k] ≠ pList[j][ k]
15.                     ends[i][j] = maxindex - k
16.                     EXIT FOR
17.     u, v = argmax{starts[u][v] + ends[u][v] | u,v in {0, …, pList.length}, u < v}
18.     IF starts[u][v] + ends[u][v] = 0
19.         RETURN XOR(pList)
20.     ELSE
21.         result = take first starts[u][v] elements of pList[u].
22.         x = pList[u]
23.         Remove first starts[u][v] elements from x.
24.         Remove last ends[u][v] elements from x.
25.         y = pList[v]
26.         Remove first starts[u][v] elements from x.
27.         Remove last ends[u][v] elements from x.
28.         Append XOR(x, y) to result.
29.         Append last ends[u][v] elements of pList[u] to result.
30.         Remove pList[u] and pList[v] from pList.
31.         Add result to pList.
32.         RETURN SOLVE_OPTIONALITY(pList)
```

3.4. Az MPM „fő" algoritmus: a maximális minták kinyerése

Az eljárás először az eseménynaplóban tárolt nyomokból kiküszöböli a zajt. A megmaradó nyomokhoz mintát szerkeszt az ismétlődések vizsgálatával. A keletkező mintákból eltávolítja azokat, amelyek részmintái valamely másik mintának (nem maximálisak). A fennmaradt mintákban beazonosítja a párhuzamosságokat és elvégzi az opcionalitás vizsgálatát, melynek eredményeképpen egyetlen minta keletkezik, ami „magában foglalja" a többit. Végül ehhez a mintához elkészít egy automatát, amellyel visszatér és tovább folytatja a nyomok vizsgálatát.

Megjegyezzük, hogy itt a „thresh" kifejezés a „treshvalue" (küszöbszám) rövidítése.

```
1.  MPM(eventlog, thresh)
2.          traces = REMOVE_NOISE(eventlog)
3.          patterns = {}
4.
5.          FOR each trace in traces
6.                  patterns. Add(SOLVE_LOOP(trace))
7.
8.          FOR each pattern p in patterns
9.                  FOR each pattern q in patterns
10.                         IF p is a subpattern of q
11.                                 patterns.Remove(p)
12.                         IF q is a subpattern of p
13.                                 patterns.Remove(q)
14.
15.         FOR each pattern p in patterns
16.                 FOR each pattern q in patterns
17.                         r = SOLVE_CONCURRENCY(p, q)
18.                         patterns. Remove(p)
19.                         patterns. Remove(q)
20.                         patterns. Add(r)
21.
22.         result = SOLVE_OPTIONALITY(patterns)
23.         RETURN CREATE_NFA(result)
```

3.5. A kibányászott maximális sémákhoz tartozó gráf megrajzolása

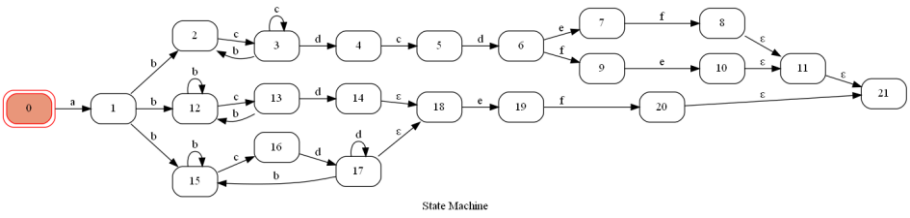
Miután az MPM-eljárás a tevékenységnaplóban megadott nyomokból kiindulva meghatározza a maximális mintákat és megállapítja a párhuzamosságokat és elágazásokat, a keretprogram meghívja a kirajzoló programot. A kirajzoltatás a Pytransitions könyvtárban található GraphMachine osztály által létrehozott gráf draw függvényével történik. Az ábra elkészítésére a „graphviz" vagy a „pygraphviz" Python-alkönyvtárat használja. Az MPM-algoritmushoz tartozó összefogó kódrész a kirajzoltatással együtt a következő:

```
traces = [[c for c in entry] for entry in eventlog]
traces = remove_noise(traces, 0.2)
patterns = list(map(lambda trace: solve_loop(trace), traces))
maximal_patterns = determine_maximal_patterns(patterns, traces)
with_concurrency = maximal_patterns.copy()

i = 0
while i < len(with_concurrency)-1:
    j = i + 1
```

```
    while j < len(with_concurrency):
        r = solve_concurrency(with_concurrency[i], with_concurrency[j])
        if len(r) == 1:
            with_concurrency[i] = r[0]
            del with_concurrency[j]
        else
            j += 1
    i += 1
result = solve_optionality(with_concurrency)
final_nfa = NFA(result.elements)
final_nfa.get_graph().draw("result.png", prog = "dot")
```

Például a $T = \langle abbcbbbbcbbcbbbcbcdef,\ abcbcbccdcdef,\ abcbcbccdcdfe,\ abbbcdbcdddef \rangle$ eseménynaplóhoz a következő automatát készíti el az algoritmus.



3.6. A „keret” algoritmus

Az MPM-eljárásnak a kapott mintákhoz egy olyan automatát kell konstruálnia, ami akceptál minden olyan szót, ami illeszkedik a mintára. Mivel a minta tartalmazhat (egyedi) eseményeket, ciklusokat, párhuzamosságokat, opcionalitást, valamint ezeket a részleteket egymásba ágyazott formában is, így a kidolgozott eljárásnak ezt rekurzívan kell kezelnie. Az itt ismertetett algoritmusban az állapotokat nemnegatív egész számok jelölik, a kezdőállapot 0. Mivel az algoritmus rekurzív, így folyamatosan figyeli, hogy a következő szakasz elejéhez tartozó állapot az előző szakasz végén melyik elemből fog kiindulni (last paraméter). A ciklusnál gondoskodni kell róla, hogy az elem utolsó állapotából vezessen az elsőbe is tranzíció, ezzel biztosítva az ismétlődés lehetőségét. Párhuzamos eseményeknél az automata elágazik többfelé, mindegyiknél az eseménysorozat valamely permutációjához tartozó tranzíciók szerepelnek. Opcionalitás esetén szintén elágazás történik, ahol egy-egy elágazás egy-egy opcionális karaktersorozatnak felel meg. Az eljárás pszeudokódja az alábbi:

```
1.  CREATE NFA (PATTERN)
2.          Q={0}
3.          Σ={e | e  is an event-type in pattern}
4.          T=ø
5.          f=GET_TRANSITIONS_FROM_PATTERN(pattern, -1, Q, T)
6.      F={f}
7.      RETURN(Q, Σ , T, 0, f)
8.
9.
10. GET_TRANSITIONS_FROM_PATTERNS(pattern, prev, Q, T)
11.         IF prev < 0
12.             last = max(Q)
13.     ELSE
14.                     last = prev
15.     FOR element IN pattern
16.         FOR start > end
17.                     IF element is single event
18.                 current = max(Q)+1
19.                         Q = Q ∪ {current}
20.                     last = current
21.                     ELSE IF element is loop
22.                             start =  max(Q)+1
23.             GET_TRANSITION_FROM_PATTERN(loop.elements, last, Q, T)
24.                         end = max(Q)
25.                             e = first event of loop
26.                 T = T ∪ {(e, end, start)}
27.                     last = end
28.                     ELSE IF element is concurent
29.                             ends  = ø
30.             FOR p IN permutations of concurent.elements
31.                 ends = ends ∪ PATTERN GET_TRANSITIONS_FROM_PAT-
32. TERN(p, last, Q, T)
33.             recombine = max(Q)+1
34.             FOR end IN ends
35.                 T = T ∪ {(Ɛ, end, recombine)}
36.             last = recombine
37.         ELSE IF element is optionaly
38.              ends  = ø
39.         FOR eList IN optionaliy.elements
40.              ends = ends ∪ PATTERN GET_TRANSITIONS_FROM_PAT-
41. TERN(eList, last, Q, T)
42.         recombine = max(Q)+1
43.         FOR end IN ends
44.              T = T ∪ {(Ɛ, end, recombine)}
45.          last = recombine
    RETURN last
```

3.7. Elvégzett ellenőrzések, kísérletek

Az előző belső kutatási jelentésekben szereplő javaslatok, valamint az ERPA-csoport tagjainak a javaslatai alapján hozzáláttunk egy ún. tanítóhalmaz összeállításához.

Az alábbi, általam kreált tevékenységnapló estén végeztük el a program részeinek és egésze futásának az ellenőrzését:

T:= {(a, b, b, c, b, b, b, b, c, b, b, c, b, b, b, c, b, c, d, e,f), (a, b, c, b, c, b, c, c, d, c, d, e, f),
(a, b, c, b, c, b, c, c, d, c, d, f, e), (a, b, b, b, c, d, b, c, d, d, d, e, f), (b, c, d, a, a, b, f, g), (b, c, a, a, a, d, b, f, g), (a, b, c, b, c, d, g, f), (a, b, c, b, c, b, c, d, f,), (a, b, c, d, c, d, f, g, g, g), (a, b, c, d, c, d, c, d, f, g, g,, k, l),
(d, a, a, b, c, d, g, f), (c, e, f, g, d, b, b, a, a)}

– ebből a 7. és 8. minta szándékosan bevitt zaj volt.

Ezenkívül, mintegy 100 nyomot tartalmazó tevékenységnaplót állítottunk elő a csoport többi tagja által is használt Process Discovery Contest (2020) mintafilejainak a felhasználásával. Elérési linkje:
https://data.4tu.nl/articles/dataset/Process_Discovery_Contest_2020/14626020

Egyelőre csak a hurkok felismerését, a zaj kiszűrését és a maximális sémák kinyerését vizsgáltuk. Egészen jó (90% feletti) Recall értékeket kaptunk. Hibákat csak a többszörösen összetett hurkok esetén találtunk, illetve a zaj felismerésénél. Ezeket a hiányosságokat szeretnénk a jövőben korrigálni.

3.8. Optimális megoldások keresése egy tanítóhalmazban

Ez tulajdonképpen csak a kérések (requestek) ismeretében megvalósítható és a következő két lehetőség érhető el az algoritmusunkkal: A kérésekkel címkézett minták közül minden kérés esetén kiválaszthatjuk a leggyakoribb mintát(kat), illetve a legrövidebb tranzíciós útvonalat tartalmazó mintát. Ez utóbbi esetén a legrövidebb útvonal alatt egy kezdőállapotból a megoldásba (végállapotba) vezető tranzíciós útvonalat értjük – a fenti algoritmus ezek kinyerésére is alkalmas.

4. Összegzés

A kutatás egy fontos szakasza lezárult az MPM-alapprogram implementálásával. A következő nagyobb szakasz ennek a tesztelésével és tökéletesítésével kapcsolatos.

a) Ezen belül, következő lépésként egy adekvát tanítóhalmaz kialakítása lenne a cél, oly módon, hogy az ebben szereplő mintákat különböző kéréseknek feleltessük meg.
b) Ezekkel először az algoritmus hatékonyságát tesztelnénk a fentiekben már részletezett egyéb mérőszámokat és a korábban már megfogalmazott validációs alapelveket is felhasználva. Ezeket ezután egy külön szubrutin tartalmazná, amit az alapprogramhoz kapcsolnánk.
c) A kapott futtatási eredmények birtokában később javaslatokat fogalmaznánk meg az alapalgoritmus tökéletesítésére. Így például a mostani ellenőrzés fényében szükséges lenne átnéznünk az összetett hurkok képzését, valamint a zaj felismerésének a hatékonyságát.
d) A megkezdett ellenőrzést a párhuzamosságok és opcionalitások feltárásának a vizsgálatával, a folyamatgráf egyszerűsíthető voltának az ellenőrzésével folytatnunk szükséges.
e) Felmerült annak a lehetősége is, hogy az algoritmus kibővítsük NN-alapú részalgoritmusokkal, amelyek a hurkok ismétlődési számát, illetve a párhuzamosított részszekvenciák méretét szabályoznák, oly módon, hogy azok minél „valóságosabbak" legyenek.

Felhasznált irodalom

[1] van der Aalst, W. M. P.: Process Mining: Overview and Opportunities. *ACM Transactions on Management Information Systems*, 2012, vol. 3, no. 2, article 7.

[2] Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. *Proceedings of the 6th International Conference on Extending Database Technology* (EDBT'98), 1998, LNCS 1377, pp. 469–483.

[3] Cook, J., Wolf, A.: Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology,* 1998 (7), pp. 215–249.

[4] Datta, A.: Automating the discovery of AS-IS business process models: probabilistic and algorithmic approaches. *Information Systems Research*, 1998, vol. 9, pp. 275–301.

[5] Mannila, H., Meek, C.: Global partial orders from sequential data. *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '00), 2000, pp. 161–168.

[6] van der Aalst, W. M. P., Weijters, A. J. M. M., Maruster, L.: Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 2004, vol. 16, pp. 1128–1142.

[7] Alves de Medeiros, A. K., van Dongen, B. F., van der Aalst, W. M. P. and Weijters, A.J.M.M.: Process Mining: Extending the Alpha-Algorithm to Mine Short Loops. *BETA Working Paper Series*, TU Eindhoven, 2004, vol. 113.

[8] Wen, L., van der Aalst, W. M. P., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery*, 2007 (15), pp. 145–180.

[9] Wen, L., Wang, J., van der Aalst, W. M. P., Huang, B., Sun, J.: A novel approach for process mining based on event types. *Journal of Intelligent Information Systems*, 2009, vol. 32, pp. 163–190.

[10] Weijters, A. J. M. M., van der Aalst, W. M. P., Alves de Medeiros, A. K.: Process Mining with the Heuristics Miner algorithm. *BETA Working Paper Series*, 2006, TU Eindhoven, vol. 166.

[11] Graves, A. (2013). Generating sequences with recurrent neural network. *arXiv preprint arXiv:1308.0850*.

[12] Folino, F., Greco, G., Guzzo, A., Pontieri, L.: Discovering expressive process models from noised log data. *Proceedings of the 2009 International Database Engineering & Applications Symposium*, 2009, ACM, pp. 162–172.

[13] Islam, M. S., Mousumi, S. S., Abujar, S. and Hossain, S. A. (2019). Sequence-to-sequence Bangla sentence generation with LSTM recurrent neural networks. *Procedia Computer Science*, 152, pp. 51–58.

[14] Ferreira, H., Ferreira, D.: An integrated life cycle for workflow management based on learning and planning. *International Journal of Cooperative Information Systems*, 2006, vol. 15, pp. 485–505.

[15] Burattin, A. and Sperduti, A. (2010, September). PLG: a framework for the generation of business process models and their execution logs. In *International Conference on Business Process Management*, pp. 214–219, Springer, Berlin, Heidelberg.

[16] Liesaputra, V., Yongchareon, S. and Chaisiri, S.: (2016, Sept.) Efficient process model discovery using maximal pattern mining. In *International Conference on Business Process Management*, pp. 441–456, Springer, Cham.

[17] Alves de Medeiros, A. K., Weijters, A. J. M. M., van der Aalst, W. M. P.: Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery,* 2007, vol. 14, pp. 245–304.

[18] Sutskever, I., Martens, J. and Hinton, G. E.: Generating text with recurrent neural networks. *ICML* (2011, Jan.)

[19] Kuo, C. Y., & Chien, J. T. (2018, September). Markov recurrent neural networks. In *2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP)* (pp. 1–6). IEEE.

[20] Hanga, K. M., Kovalchuk, Y. and Gaber, M. M. (2020). A Graph-Based Approach to Interpreting Recurrent Neural Networks in Process Mining. *IEEE Access*, 8, pp. 172923–172938.

[21] Yunhao, T. Agrawal, S. and Faenza. S. Reinforcement learning for integer programming: Learning to "cut". *International Conference on Machine Learning*, PMLR, 2020.

[22] Agrawal, S. IEOR 8100: Reinforcement learning lectures.

[23] Ritter, G. X., and Urcid, G. (2021). Introduction to Lattice Algebra: With Applications in AI, Pattern Recognition, Image Analysis, and Biomimetic Neural Networks.

[24] Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. *Proc. 8th ACM Intern. Conf. Knowl. Discov. Data Mining*, ACM (2002), pp. 429–435.

[25] Fournier-Viger, P., Lin, J. C. W., Kiran, R. U., Koh, Y. S., & Thomas, R. (2017). A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1 (1), pp. 54–77.

[26] Pelleg, Dan; Moore, Andrew (1999). Accelerating exact k-means algorithms with geometric reasoning. *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '99*. San Diego, California, United States: ACM Press: pp. 277–281.

[27] Szilágyi Judit et al. (2014). A Support Vector Machine osztályozó eljárás alkalmazása felszínborítás vizsgálatok esetében. *Agriculture Informatics*, p. 46.

[28] Powers, David M W (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation (PDF). *Journal of Machine Learning Technologies*, 2 (1), pp. 37–63. Archived from the original (PDF) on 2019-11-14.

[29] Deza, Michel Marie, Laurent, Monique (1997). Geometry of Cuts and Metrics. *Algorithms and Combinatorics*, 15, Springer-Verlag, Berlin, p. 27. doi:10.1007/978-3-642-04295-9