Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Időzített rendszerek tanulás alapú analízise

Szakdolgozat

Készítette
Gujgiczer Anna

Konzulens
Farkas Rebeka

2017. december 8.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2017. december 8.

Gujgiczer Anna
hallgató

Kivonat

Biztonságkritikus rendszerek esetén kiemelten fontos a hibamentes működést ellenőrizni, például modell-alapú technológiák segítségével. Ezen rendszerek jellemzően valósidejű rendszerek, amelyekben különösen fontos az időzítési paraméterek hatásának vizsgálata, hiszen sokszor a bonyolult időzítési viszonyokból fakadnak a tervezési hibák. Előfordul azonban, hogy a rendszer egyes komponenseinek belső megvalósítása nem ismert (például külső fél által készített komponensek esetén), így működéséhez nem létezik formális modell vagy specifikáció.

Automatatanulás segítségével a megfigyelt működések alapján automatikusan származtatható a rendszernek egy véges automata-alapú viselkedésmodellje. A valósidejű rendszerek azonban új kihívásokat állítanak az automatatanulás elé, mivel kiemelten fontossá válik az időzítés-függő viselkedések felismerése, tanulása.

Munkám során időzített rendszerek tanulásával foglalkozom. Létrehoztam egy gráfadatbázison alapuló szoftvert, amely lehetővé teszi időzített rendszerek működésének analízisét egy valósidejű automatatanuló algoritmus segítségével, illetve megvizsgáltam ezen algoritmus módosításait, továbbfejlesztési lehetőségeit.

Az általam megvalósított szoftver segítségével lehetővé válik olyan szoftverkomponensek viselkedésének megtanulása, amelyekre eddigiekben nem volt lehetőség, időfüggő viselkedésük miatt. Az így létrejövő specifikáció alapján lehetővé válik a komponens viselkedésének megértése, ellenőrzése, anomáliadetektáló monitor származtatása, valamint dokumentáció, illetve tesztek generálása.

Dolgozatom célja bemutatni az általam implementált szoftvert és a felhasznált algoritmust, valamint a javasolt fejlesztéseket. Dolgozatomban az implementáció is részletesen bemutatásra kerül. Fejlesztéseim hatásosságát mérésekkel demonstrálom.

Abstract

In case of safety critical systems, it is extremely important to ensure fault-free operation, for example, by using model-based technologies. These systems are typically real-time systems where it is particularly important to examine the effect of timing parameters, as design problems often arise from complex timing conditions. However, in many cases the internal implementation of some components of the system is unknown (for example, in case of third-party components) and therefore there is no formal model or specification of its operation.

Automaton learning algorithms can be used to automatically derive a finite automaton-based behavioral model of the system, based on the observed operations. Real-time systems, however, raise new challenges to automaton learning, as the recognition and learning of time-dependent behaviors become rather important.

In my work I study learning algorithms for timed systems. I have implemented a software, based on a graph database, that allows the analysis of the time-dependent behavior of systems using a real-time automaton learning algorithm. I have also examined possible modifications and improvements of the algorithm.

The software I have implemented, supports the analysis of system components that was previously impossible due to their time-dependent behavior. Based on the resulting specification, the behavior of the system can be understood, analyzed, anomaly-detecting monitors can be derived, and documentation and tests can be generated.

The goal of my thesis is to present the software I have implemented and the algorithm it uses as well as the proposed improvements. The implementation details are also presented. The efficiency of the improvements is demonstrated by measurements.

1. fejezet

Bevezetés

Kontextus Biztonságkritikus kiberfizikai rendszerek fejlesztésére széles körben használnak modellvezérelt tervezést, amely során már a fejlesztés korai fázisaiban is lehetőség nyílik a formális módszerek eszköztárát felhasználni a rendszertervek ellenőrzésére. Emellett a formális modellekből monitorok is előállíthatók, melyek segítségével a futásidőben bekövetkező anomáliák is kiszűrhetőek.

Problémafelvetés A modellezés, precíz modellek alkotása nehéz feladat, különösen kiberfizikai rendszerek esetén. Sokszor a rendszer, vagy egyes komponensek belső viselkedése nem ismert, mert nem áll rendelkezésre dokumentáció, vagy akár a forráskód sem (egy külső beszállítótól érkező komponens esetén). Az is problémát okoz, amikor létezik valamiféle dokumentáció, specifikáció, de az hibás, vagy nem tartalmaz elég információt a megfelelő ellenőrzéshez: a kiberfizikai rendszereket gyakran bonyolult időzítések, időkényszerek vezérlik – az időfüggő viselkedések kellő ismerete kritikus lehet a megbízható verifikációhoz.

Célkitűzés Munkám célja egy olyan módszer megalkotása, mellyel kritikus, időzített kiberfizikai rendszerek viselkedésmodelljét tudom automatikusan előállítani tanuló algoritmus segítségével. A módszer megalkotása során kiemelten fontos szempont az időfüggő paraméterek felismerése a rendszerben.

Kontribúció Dolgozatomban bemutatok egy időzített automata alapú viselkedési modelleket tanuló megközelítést. Időzített automatatanulást használok formális modellek szintézisére. Megvizsgálom ezen algoritmus kiterjesztésének a lehetőségeit, annak fejlesztésére javaslatokat teszek.

Hozzáadott érték Az általam megvalósított szoftver segítségével a rendszerek szélesebb köre válik megismerhetővé. Az algoritmus segítségével rendszerek időfüggő viselkedései is feltérképezhetők. Ez pedig elősegíti ezen rendszerek analízisét, verifikációját.

Kapcsolódó munkák Bár az automatatanulás erősen kutatott téma, az időzített rendszerek tanulása új terület. A hasonló algoritmusok közül kiemelendő [17], amely az általam felhasznált algoritmusnak egy továbbfejlesztését írja le, illetve [7], amely egy nagyobb kifejezőerejű formalizmust képes megtanulni, de ezek közül egyikhez sincsen még implementáció. Tanulókeretrendszerek közül kiemelendő a LearnLib [15], illetve a Tomte [1], de ezek közül egyik sem támogatja időzítés-függő viselkedések tanulását. Azon megközelítések, amelyhez implementáció is elérhető, általában valószínűségi modelleket tanulnak meg, például [11], azonban az ilyen modellekben az idő csak az események

valószínűségét befolyásolja. Kevés olyan algoritmushoz létezik implementáció, ami képes időfüggő viselkedések felismerésére – ilyen például [10], ami egy online algoritmus az általam használt offline algoritmussal szemben.

Dolgozat felépítése A dolgozat további részének felépítése a következő. A 2. fejezetben bemutatom a dolgozat megértéséhez szükséges előismereteket, így az automataelmélet és az automatatanulás alapjait, valamint a tulajdonsággráfokat és a gráflekérdezéseket. A 3. fejezetben az általam megvalósított tanulóalgoritmust mutatom be elméleti szinten. A 4. fejezetben ismertetem az algoritmus megvalósítását, valamint a tanuló komponens egy felhasználását. A 5. fejezetben bemutatom az algoritmus továbbfejlesztési lehetőségeit. A 6. fejezetben pedig összefoglalom a munkámat és kifejtem jövőbeli terveimet.

2. fejezet

Előismeretek

Ebben a fejezetben a dolgozat további részeinek megértéséhez szükséges elméleti előismereteket mutatom be. Először az általam használt automataelméleti alapfogalmakat, jelöléseket sorolom fel, majd bemutatom az automatatanulás alapjait, végül ismertetem a tulajdonsággráfokkal, gráfmintákkal kapcsolatos alapfogalmakat.

2.1. Alapfogalmak, jelölések

A bemutatott fogalmak nagyrészben megegyeznek a formális nyelvek elméletéből ismert jelölésekkel [4], azonban az automatatanulás miatt további fogalmak is megjelennek.

Az automaták témakörében egyértelműen definiálva van, hogy az *ábécé*, *karakter*, *szó* és *nyelv* fogalmak mit jelentenek. Mindig fontos megállapítani, hogy egy automata milyen ábécé felett van értelmezve. Ez az ábécé tetszőleges karakterek halmaza lehet.

Definíció 1 (Ábécé, karakter). Egy tetszőleges, nem üres, véges halmazt ábécének nevezünk. Jelölése: Σ. A Σ ábécé elemeit betűknek, avagy karaktereknek nevezzük. ▪

Definíció 2 (Bemeneti esemény). Dolgozatomban a bemeneti esemény egy karaktert jelent, hiszen az automatatanulás során bemeneti eseményekkel kommunikál a tanuló algoritmus a rendszerrel. ▪

A karakterekből képezhető sorozatokat szavaknak nevezzük.

Definíció 3 (Szó). Egy szó a Σ ábécé elemeiből, karaktereiből képezhető véges sorozat. Az ω szó hosszát $|\omega|$ jelöli. A Σ ábécén képezhető összes szó halmazát Σ^* jelöli. ▪

Definíció 4 (Üres szó). Üres szónak nevezzük azt a szót, mely nem tartalmaz egyetlen karaktert sem. Jelölése: ϵ. ▪

Definíció 5 (Lefutás). Dolgozatomban a lefutás és szó szavakat szinonimaként használjuk. A lefutás a bemeneti események sorozata a rendszer egy adott végrehajtása során. ▪

Valós idejű automaták esetén a karakterekhez időpontok is tartoznak, így a szavak is máshogyan definiálhatók.

Definíció 6 (Időzített szó). Időzített szónak nevezünk egy olyan $\omega_t = (a_1, t_1)(a_2, t_2)...(a_n, t_n)$ sorozatot, ahol $a_1, a_2, ..., a_n \in \Sigma$ és $t_1, t_2, ..., t_n \in \mathbb{N}$. Az ω_t időzített szó hosszát $|\omega_t|$ jelöli. A Σ ábécén képezhető összes szó halmazát Σ_t^* jelöli. ▪

Időzítetlen és időzített esetben is az ábécéből képezhető szavak egy részhalmazára nyelvként hivatkozunk.

Definíció 7 (Nyelv). Egy Σ ábécé feletti L nyelvnek nevezzük az ábécé elemein képezhető összes szó egy (nem feltétlenül véges) részhalmazát ($L \subseteq \Sigma^*$). ▪

Definíció 8 (Időzített nyelv). Egy Σ ábécé feletti időzített nyelvnek nevezzük az ábécé elemein képezhető összes időzített szó (Σ_t^*) egy (nem feltétlenül véges) részhalmazát. A nyelvet L_t-lel jelöljük. Azaz $L_t \subseteq \Sigma_t^*$. ▪

2.2. Véges automata

A véges automata[4][8] az egyik legegyszerűbb számítási modell, amely annak eldöntésére használható, hogy az egyes szavak az adott nyelv elemei-e. Ezek a nyelvek akár rendszerek biztonságos működését jelentő lehetséges lefutások halmazai is lehetnek. A véges automatát általában DFA-val rövidítjük (Deterministic Finite Automaton).

Definíció 9 (Véges automata). A véges automatát a $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F \rangle$ struktúra írja le, ahol

- Q az automata állapotainak véges, nem üres halmaza,
- Σ az automata ábécéjének véges, nem üres halmaza,
- $\delta : Q \times \Sigma \to Q$ az automata állapotátmeneti függvénye,
- $q_0 \in Q$ a kezdőállapot és
- $F \subseteq Q$ az elfogadó állapotok halmaza. ▪

Egy véges automata működése a következőképpen írható le egy adott $\omega = a_1 a_2 ... a_n \in \Sigma^*$ szón:

Definíció 10. Az $r_0, r_1, r_2, ..., r_n (r_i \in Q)$ állapotsorozat az $a_1 a_2 a_3 ... a_n$ szóhoz tartozó számítás, ha $r_0 = q_0$ és $r_i = \delta(r_{i-1}, a_i)$ minden $i = 1, 2, ..., n$ esetén. ▪

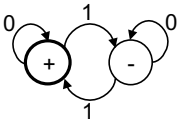
Tehát az automatát a $q_0 = r_0$ állapotból indítva rendre beolvassuk a karaktereket, és a következő állapotba a beolvasott karakter és a jelenlegi állapot által meghatározott átmeneti függvény szerint lépünk.

Egy szót elfogad az automata, ha a legutolsó beolvasott karaktere után az automata egy elfogadó állapotba ér. Ellenkező esetben az automata nem fogadja el, vagyis elutasítja a szót. Egy szó eleme az automata által leírt nyelvnek, ha az automata elfogadja azt. Az alábbiakban a vonatkozó precíz definíciókat tekintjük át.

Definíció 11. Az $\mathcal{M}$ automata elfogadja $\omega \in \Sigma^*$ szót, amennyiben $|\omega| = n$ és az ω szóhoz tartozó számítás végén r_n állapot egy elfogadó állapota az automatának. Vagyis $r_n \in F$. Egyébként $\mathcal{M}$ nem fogadja el, más néven elutasítja ω szót. ▪

Definíció 12 (Véges automata nyelve). Az $\mathcal{M}$ véges automata nyelve azon ω szavak összessége, melyeket $\mathcal{M}$ elfogadja. Jelölése: $L(\mathcal{M})$. A korábbiakból következik, hogy $L(\mathcal{M}) \subseteq \Sigma^*$. ▪

Példa 1. *Legyen $\Sigma = \{0, 1\}$, vagyis az ábécé karakterei legyenek 0 és 1. Legyen M egy olyan automata, melynek $L(\mathcal{M})$ nyelvébe azok a szavak tartoznak, amikben páros darab 1-es szerepel.*



2.1. ábra. Az 1. példában leírt automata ábrázolása

A 1. példában leírt automatát a 2.1. ábrán látható módon ábrázolhatjuk grafikusan. A dolgozatunkban ezt az ábrázolásmódot fogjuk a későbbiekben is alkalmazni. Ennek a következő elemei vannak:

Állapot: Az $\mathcal{M}$ automata állapotait körök jelölik.

Állapotátmenet: Az $\mathcal{M}$ automata állapotátmeneteit nyilak jelölik, melyek az átmenet kezdőállapotából a végállapotába mutatnak. Az állapotátmenet karakterét a nyílra írjuk rá.

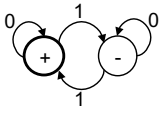
Elfogadó állapot: Az $\mathcal{M}$ automata elfogadó állapotait a $+$ jelzés jelöli.

Nem elfogadó állapot: Az $\mathcal{M}$ automata nem elfogadó állapotait a $-$ jelzés jelöli.

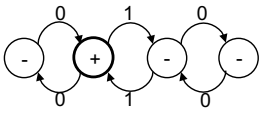
Kezdőállapot: Az $\mathcal{M}$ automata kezdőállapotát, r_0-t vastagított körvonal jelöli.

2.2.1. Minimális véges automata

Belátható az, hogy egy adott L nyelvre létezhet több automata is, melyek az L nyelv szavait fogadják el. Mint ahogy a 1. példában szereplő nyelvvel az 2.2. ábra mindkét automatájának nyelve is megegyezik. Ezeknek az automatáknak nem feltétlenül ugyanakkora az állapottere, vagyis nem ugyanannyi állapotból állnak. Ezen automaták közül a legkevesebb állapottal rendelkezőt minimális automatának nevezzük.



(a) Minimális véges automata



(b) Nem minimális véges automata

2.2. ábra. Véges automata állapottere

Definíció 13 (Állapottér). $\mathcal{M}$ véges automata állapottere alatt az állapotainak számát értjük, vagyis Q elemeinek a számát. ▪

Definíció 14 (Minimális véges automata). Egy L nyelvhez tartozó minimális automata egy olyan $\mathcal{M}$ automata, melyre igaz, hogy $L(\mathcal{M}) = L$ és az ilyen automaták közül $\mathcal{M}$ állapottere a legkisebb. ▪

2.2.2. Teljes és hiányos véges automata

Sokszor, főként kényelmi és átláthatósági szempontokból, nem szoktak minden lehetséges állapotátmeneti függvényt ábrázolni (pl. ha egy adott karakterre ugyanabban az állapotban fog maradni az automata). Abban az esetben, ha hangsúlyozni akarjuk, hogy nem hiányos automatáról van szó, akkor a 9. definícióban szereplő automatát teljes véges automatának szokták nevezni.

A 2.3. ábrán látható, hogy a 1. példában leírthoz hasonló, páros számú 1-est elfogadó nyelvet hogyan lehet egy teljes, illetve egy hiányos automatán ábrázolni. Az ábra azt is mutatja, hogy ebben az esetben indokolt is lehet hiányos automatát alkalmazni, hiszen a páros számú 1-est tartalmazó szavak elfogadásához a 0-ás karakterekkel nem kell foglalkozni.

(a) Teljes véges automata **(b)** Hiányos véges automata

2.3. ábra. Véges automata teljessége

Definíció 15 (Teljes véges automata). Teljes véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre δ állapotátmenet definiálva van. ▪

Definíció 16 (Hiányos véges automata). Hiányos véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben nincs minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre δ állapotátmenet definiálva. ▪

Hiányos véges automaták esetén többféle megoldás is lehetséges, amikor egy ω szó számítása során olyan karakter következne, amire az automata adott állapotában nincsen állapotátmenet értelmezve. Egyik lehetőség, hogy a számítás elakad. Ebben az esetben az automata nem fogadja el az ω a szót. Másik lehetséges megoldás, hogy az automata figyelmen kívül hagyja (hurokélként értelmezi) a nem definiált átmenetet. A két megoldás különbözik, pl. a 2.3. ábrán látható automata az $1, 0, 1$ szót az előbbi értelmezés szerint nem fogadná el, az utóbbi értelmezés szerint igen.

2.2.3. Determinisztikus és nemdeterminisztikus véges automata

A nyelvek leírását sokszor megkönnyíti, ha nem csak elhagyhatunk átmeneteket, de azt is megengedjük, hogy azok ne legyenek egyértelműek. Ez azt jelenti, hogy egy állapotból egy karakterre nem csak egy lehetséges állapotba juthat az automata. Az ilyen, nem egyértelmű állapotátmeneteket tartalmazó automatákat nemdeterminisztikus automatáknak nevezzük, míg a csak egyértelmű állapotátmenettel rendelkezőket determinisztikus automatáknak.

A 2.4. ábra ábrázol egy az 1.. példában definiált nyelvre adott determinisztikus és nemdeterminisztikus automatát.

(a) Determinisztikus véges automata **(b)** Nemdeterminisztikus véges automata

2.4. ábra. Véges automata determinisztikussága

Definíció 17 (Determinisztikus véges automata). Determinisztikus véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre maximum egy δ állapotátmenet van definiálva. ▪

Definíció 18 (Nemdeterminisztikus véges automata). Nemdeterminisztikus véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben van olyan $q \in Q$ kezdőállapot és $a \in \Sigma$ karakter, melyre több δ állapotátmenet is definiálva van ▪

Egy nemdeterminisztikus véges automata akkor fogad el egy szót, ha létezik benne olyan lefutás, amelyben a szó utolsó karaktere elfogadó állapotba juttatja.

Általában amikor véges automatáról beszélünk, akkor determinisztikus véges automatára (deterministic finite automaton) gondolunk. Ezért is használjuk a DFA rövidítést.

2.3. Valósidejű automata

Az időzített automaták[2] közé tartozik a valósidejű automata[17][6], mely a valós idejű rendszerek leírásához használt formalizmus. Az automata élein nem csak a bemeneti események szerepelnek, hanem az azokhoz tartozó időkorlát is. Egy időkorlát azt határozza meg, hogy egy adott állapotban legalább és legfeljebb mennyi időt kell töltenie az automatának, hogy egy adott bemeneti eseményre egy adott állapotátmenet engedélyezetté váljon. Az valósidejű automata, röviden RTA (Real-Time Automaton) így tulajdonképpen a véges automata egy időzítéssel kiterjesztett változata.

Definíció 19 (Valósidejű automata). A valósidejű automatát a $\mathcal{A} = \langle Q, \Sigma, D, q_0, F \rangle$ struktúra írja le, ahol

- Q az automata állapotainak véges, nem üres halmaza,
- Σ az automata ábécéjének véges, nem üres halmaza,
- $D : Q \times \Sigma \times \Phi \to Q$ az automata állapotátmeneti függvényeinek véges halmaza, ahol Φ az $\mathbb{N}$ feletti intervallumok halmazát jelöli
- $q_0 \in Q$ a kezdőállapot és
- $F \subseteq Q$ az elfogadó állapotok halmaza.

Az automata egy $d \in D$ átmeneti függvénye $\langle q, q', s, \phi \rangle$ négyesből áll, ahol $q, q' \in Q$ a kezdő és végállapotok, $s \in \Sigma$ egy karakter, $\phi \in \Phi$ pedig az időkorlát, ami egy intervallum $\mathbb{N}$ felett. ▪

Egy valósidejű automata működése a következőképpen írható le egy adott $\omega_t = (a_1, t_1)(a_2, t_2)...(a_n, t_n)$ időzített szón, ahol $a_1, a_2, ..., a_n \in \Sigma$ és $t_1, t_2, ..., t_n \in \mathbb{N}$:

Definíció 20. Az $r_0, r_1, r_2, ..., r_n (r_i \in Q)$ állapotsorozat az $(a_1, t_1)(a_2, t_2)...(a_n, t_n)$ időzített szóhoz tartozó számítás, ha $r_0 = q_0$ és $r_i = d(r_{i-1}, a_i, \phi_i)$, ahol $(t_i - t_{i-1}) \in \phi_i$ minden $i = 1, 2, ..., n$ esetén. ▪

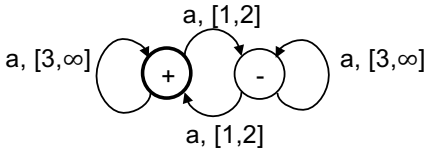
Tehát az automatát a $q_0 = r_0$ állapotból indítva rendre beolvassuk a karaktereket és a karakterek között eltelt időket $(t_i - t_{i-1})$, és a következő állapotba a beolvasott karakter, az eltelt idő és a jelenlegi állapot által meghatározott átmeneti függvénnyel lépünk.

Egy időzített szót elfogad az automata, ha a legutolsó beolvasott karaktere és időköze után az automata egy elfogadó állapotba tér. Ellenkező esetben az automata nem fogadja el, vagyis elutasítja azt az időzített szót. Egy időzített szó eleme a valósidejű automata által leírt időzített nyelvnek, ha az automata elfogadja azt. Pontosabban:

Definíció 21. Az $\mathcal{A}$ valósidejű automata elfogadja $\omega_t \in \Sigma_t^*$ szót, amennyiben $|\omega| = n$ és a ω hoz tartozó számítás végén r_n állapot egy elfogadó állapota az automatának. Vagyis $r_n \in F$. Egyébként A nem fogadja el, más néven elutasítja ω_t szót. ▪

Definíció 22 (Valósidejű automata nyelve). Az $\mathcal{A}$ valósidejű automata időzített nyelve azon ω_t szavak összessége, melyeket $\mathcal{A}$ elfogad. Jelölése: $L_t(\mathcal{A})$. A korábbiakból következik, hogy $L_t(\mathcal{A}) \subseteq \Sigma_t^*$. ▪

Példa 2. *Legyen $\Sigma = \{a\}$, vagyis az ábécé egyetlen karaktere legyen a. Legyen $\mathcal{A}$ egy olyan valósidejű automata, melynek $L_t(\mathcal{A})$ nyelvébe azok a szavak tartoznak, amelyekben páros számú a szerepel és a karakterek között legalább 1, legfeljebb 2 időegység telik el.*



2.5. ábra. Az 2. példában leírt valósidejű automata ábrázolása

2.3.1. Minimális valósidejű automata

Valósidejű automata esetén is értelmezett a minimális automata fogalma. Ebben az esetben a következők szerint módosul a definíció:

Definíció 23 (Minimális valósidejű automata). Egy L_t nyelvhez tartozó minimális valósidejű automata egy olyan $\mathcal{A}$ automata, melyre igaz, hogy $L_t(\mathcal{A}) = L_t$, és az ilyen automaták közül $\mathcal{A}$ állapottere a legkisebb. ▪

2.3.2. Teljes és hiányos valósidejű automata

Valósidejű automata esetében a teljesség fogalma annyiban módosul, hogy nem elég csupán minden állapotban minden karakterre definiálni állapotátmenetet, hanem minden időipillanatra is kell.

Definíció 24 (Teljes valósidejű automata). Teljes valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre és $t \in \phi$ időkorlátra d állapotátmenet definiálva van. ▪

Definíció 25 (Hiányos valósidejű automata). Hiányos valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben nincs minden $q \in Q$ kezdőállapotból, $a \in \Sigma$ karakterre és $t \in \phi$ időkorlátra d állapotátmenet definiálva. ▪

2.3.3. Determinisztikus és nemdeterminisztikus valósidejű automata

Valósidejű automata esetében előfordulhat, hogy egy adott állapotból egy adott karakter olvasását követően a rendszer többféle állapotba kerülhet. Azonban nem megengedett, hogy ezeknek az átmeneteknek az időkorlátjai átlapolódjanak. Vagyis egy adott állapotban egy adott karakter hatására egy adott idő elteltével már nem megengedett, hogy többféle állapotba kerüljön a rendszer.

Definíció 26 (Determinisztikus valósidejű automata). Determinisztikus valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben nincs olyan $q \in Q$ kezdőállapot és $a \in \Sigma$ karakter, melyre több d állapotátmenetnek t időkorlátjai rendelkeznének közös résszel. ▪

Definíció 27 (Nemdeterminisztikus valósidejű automata). Nemdeterminisztikus valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben van olyan $q \in Q$ kezdőállapot és $a \in \Sigma$ karakter, melyre több d állapotátmenetnek t időkorlátjai rendelkeznek közös résszel. ▪

2.4. Automatatanulás

Az automatatanulás célja egy ún. *black box* rendszer működésének feltérképezése az elérhető információk alapján. Black box rendszernek nevezzük azt a rendszert, melynek pontos belső működése nem ismert, csupán a külvilággal való kommunikációja alapján lehet következtetni rá. Az ilyen rendszerek belső működésének megismeréséhez alkalmazható az automatatanulás, amely a rendszer lefutásait vizsgálva hoz létre egy automatát (hipotézismodell), amely viselkedése jól közelíti a rendszer belső működését. Általában egy tanuló algoritmustól elvárjuk, hogy az általa megtanult hipotézismodellre a következők igazak legyenek:

Determinisztikus: Az automata későbbi felhasználása (tesztesetek generálása, anomáliadetektálás, stb.) során előnyös, ha az automata determinisztikus. Ezt a kritériumot általában képes teljesíteni egy tanuló algoritmus, amennyiben a rendszer viselkedése valóban determinisztikus.

Teljes: A teljesség azért fontos, hogy minden állapotban tudjuk, hogy egy bemenetre hogyan viselkedne a rendszer. Ez a kritérium nem tud mindig teljesülni. Főként passzív tanulás (2.4.2. fejezet) esetén fordulhat elő, hogy az algoritmusnak kevés információ áll rendelkezésére.

Minimális: Szemléletesség miatt fontos, illetve kisebb memória elég hozzá. Általában nem tudja garantálni a minimális automata megtanulását egy algoritmus, de törekszik rá.

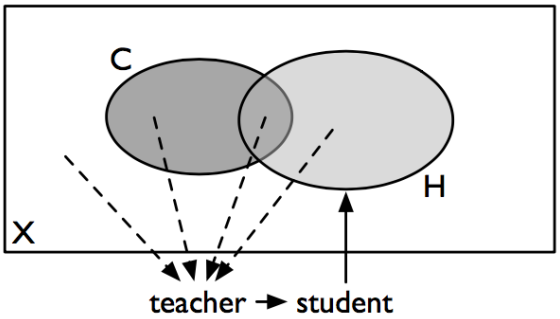
Ezen kívül gyakorlati szempontból, nem csak a megtanult automatára, de az algoritmusra nézve is vannak elvárások:

Minimális futásidő: Egy algoritmus esetében elvárás, hogy minél kevesebb idő alatt, minél kevesebb kérdéssel, az automatának minél kevesebbszeri fölösleges módosításával tudna tanulni. Ezt sem tudja általában garantálni egy tanuló algoritmus, de különféle optimalizációkkal lehet törekedni rá.

Többféle algoritmus is létezik változatos formalizmusokat és rendszereket támogatva. A különböző algoritmusok az előállított automata típusában, illetve a tanulás módjában térhetnek el.

A tanuló algoritmusok a vizsgálat során a rendszernek adott input események sorozatát és az azokra adott output reakciókat vizsgálja. A legegyszerűbb algoritmusok esetében a kimenet csupán azt az információt hordozza, hogy az adott bemeneti sorozat érvényes lefutást jelent-e. Ezáltal, a tanulás eredményeként, egy véges automata alapú modellt állít elő, mely jól modellezi a rendszer működését. Minden tanulás egy tanulóból, studentből (s) és egy tanárból, teacher (t) áll. A tanuló célja, hogy kikövetkeztessen egy hipotézist (H) a rendszer (C) működéséről.

Legyenek a rendszernek küldhető üzenetek, vagyis a rendszer lehetséges inputjai a tanulás ábécéjének karakterei. Nevezzük az ábécéből előállítható szavak halmazát (Σ^*) X-nek. A tanár a tanulót X elemeivel tanítja, elárulva az arra adott válaszát is a rendszernek is (2.6. ábra).



2.6. ábra. A tanulásban résztvevő komponensek

Definíció 28 (Rendszer). Egy C rendszer által definiált nyelv részhalmaza X-nek, amennyiben C bemeneti ábécéjéből előállítható szavak halmaza X.

Egy $x \in X$ szó egy pozitív (lefutási) példa amennyiben $x \in C$, ellenkező esetben egy negatív példa. ▪

Definíció 29 (Hipotézis). H hipotézis egy hipotézismodellje C rendszernek X fölött.

Egy $x \in X$ szó igaznak értékelődik ki H szerint, amennyiben $x \in H$, ellenkező esetben hamis. Egy H hipotézismodell helyes C-re nézve, amennyiben $x \in X$ akkor és csak akkor igaz H-ban, ha pozitív példa C-ben. Vagyis ha H és C nyelve megegyezik. ▪

Definíció 30 (Tanár). C rendszer tanára egy olyan orákulum, mely megmondja egy $x \in X$ elemről, hogy C szerint az egy pozitív, vagy negatív példa. Vagyis visszatérési értéke egy címkézett (x, b), ahol b egy boolean típus, mely akkor *igaz*, ha $x \in C$, ellenkező esetben *hamis*. ▪

Definíció 31 (Tanuló). Egy s tanuló maga a tanuló algoritmus, ami megtanulja a H hipotézist, t tanár segítségével, akinek van tudása a C rendszerről. A tanuló célja, hogy találjon egy helyes H hipotézist, melyre igaz, hogy $x \in H$ akkor és csak akkor igaz, ha $x \in C$ is igaz. ▪

Véges automatát tanuló algoritmusból is sokféle létezik. A különböző algoritmusokat[10] az alábbi szempontok alapján szokás megkülönböztetni:

Aktív vagy passzív tanulás

Aktív tanulás során az algoritmus, ha szükségesnek érzi, feltehet további kérdéseket a rendszer működését illetően (pl. megkérdezheti egy lefutásról, hogy pozitív-e), ezzel bővíteni tudja a tudását. Ezzel ellentétben a passzív tanulás során az algoritmusnak egy előre megadott lefutáshalmazzal kell dolgoznia, melyet később nincs lehetősége bővíteni.

Online vagy offline tanulás

Online tanulás során minden információhoz csak egyszer tud hozzáférni az algoritmus, ekkor kell beépítenie a modellbe. Így ezen algoritmusok működése igen időkritikus. Offline tanulás során az információ a rendszerről tárolva van. Emiatt az algoritmus többször is lekérdezheti őket, illetve akár előfeldolgozáson is áteshetnek.

Pozitív és negatív elemek

Néhány algoritmus csak pozitív elemekkel dolgozik, vagyis csak olyan $x \in X$ példákat kap a tanárától, melyekre (x, b) b része *igaz*. Azonban általában egy tanulás során pozitív és negatív elemek is rendelkezésre állnak.

Állapotok hiányos információval

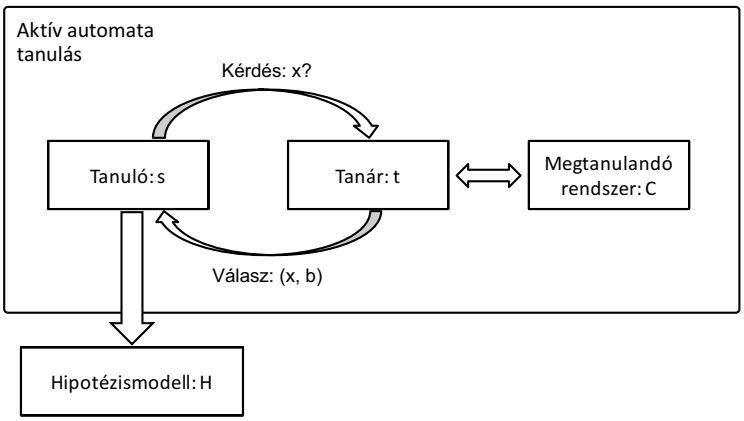
Néhány tanuló algoritmusnál nem követelmény, hogy a kapott H hipotézisautomata minden állapotról eldönthető legyen, hogy elfogadó vagy elutasító. Ilyen például akkor lehetséges, ha kevés információ áll rendelkezésre a lefutásokról, kevés az elemek száma.

Az irodalomban gyakran összekeverik az aktív és az online, valamint a passzív és az offline tanulást. Ez azért van, mert a gyakorlatban ezek a tulajdonságok gyakran egyszerre jelennek meg a tanulóalgoritmusokban.

Az alábbiakban még jobban kifejtem az aktív és passzív tanulást. A tanulás mindkét típusát szemléltetem egy-egy egyszerű példán keresztül (3. és 4. példa).

2.4.1. Aktív automatatanulás

Az aktív tanulóalgoritmus eleinte semennyi információval nem rendelkezik a megtanulandó rendszerről. Ilyenkor a kezdeti hipotézismodellje egy nagyon általános automata (általában X minden elemét elfogadó automata). A tanulás során az algoritmus kérdezgeti a megfigyelt rendszert. A kérdéseire kapott válaszok alapján tudja pontosítani a rendszert és felépíteni arról az új hipotézismodellt (2.7. ábra).



2.7. ábra. Aktív automatatanulás

Az algoritmus kétféle lekérdezést tud megfogalmazni a tanulandó rendszer számára:

Membership query avagy tagsági kérdés

A membership query-k segítségével a rendszert meghajtja annak lehetséges bemeneteivel, majd az ezekre kapott válaszok alapján, a bemenet-kimeneti párokat építve tanulja meg a viselkedéseket. A kérdéseit az alapján választja meg, hogy miből tudja meg a legtöbb információt. Az aktuális bementet az algoritmus mindig a jelenlegi tudása alapján határozza meg.

Equivalence query avagy ekvivalencialekérdezés

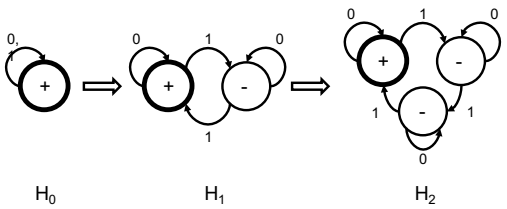
Ha a tagsági kérdések alapján az algoritmus úgy véli, hogy megtanulta az automatát, akkor egy ekvivalenciatesztet hajt végre az automata és a rendszer működése között. Ennek kimenete kétféle lehet:

Talál ellenpéldát: Ha a teszt során ellenpéldára talál, annak alapján folytatja a tanulást a tagsági kérdésekkel.

Nem talál ellenpéldát: Ha azonban nem talál ellenpéldát, akkor a tanulás befejeződik, a megtanult automata jól reprezentálja a rendszert.

Példa 3. *Legyen* $\Sigma = \{0, 1\}$*, vagyis az ábécé karakterei legyenek 0 és 1.* X *legyen továbbra is* Σ^**, vagyis a karakterekből képezhető összes szó halmaza. Legyen a* C *által elfogadott nyelv azon* x*- ek halmaza, ahol* $x \in X$ *és amely* x*-ben az* 1*-esek száma hárommal osztható.*

Ha megfigyelnénk egy aktív tanuló algoritmust, mely a 3. példában definiált rendszert tanulja és a tanulás különböző fázisaiban lekérdeznénk az aktuális hipotézismodelljét, akkor a 2.8. ábrán láthatóhoz hasonló eredményt kapnánk.



2.8. ábra. Aktív tanulás során felépített hipotézismodellek

Az aktív automatanulás széles körben használt módszer. Számos különböző hatékonyságú változata van, illetve többféle automata megtanulására is léteznek változatok.

Az algoritmus alapötlete még 1987-ből, Angluintól[3] származik. Az algoritmus a kezdeti hipotézismodellt úgy határozza meg, hogy lekérdezi a rendszer üres szóra adott reakcióját. Ezt követően ekvivalenciakérdést fogalmaz meg a tanárának. Ha ellenpéldát kap válaszul, akkor az ellenpélda alapján tagsági kérdéseket fogalmaz meg. Miután az összes tagsági kérdésére kapott választ, újból megvizsgálja az ekvivalenciát. Ezt mindaddig folytatja, míg nem kap ellenpéldát. A 1. pszeudokód írja le az algoritmus működését.

Algoritmus 1: Angluin féle aktív automatatanulás

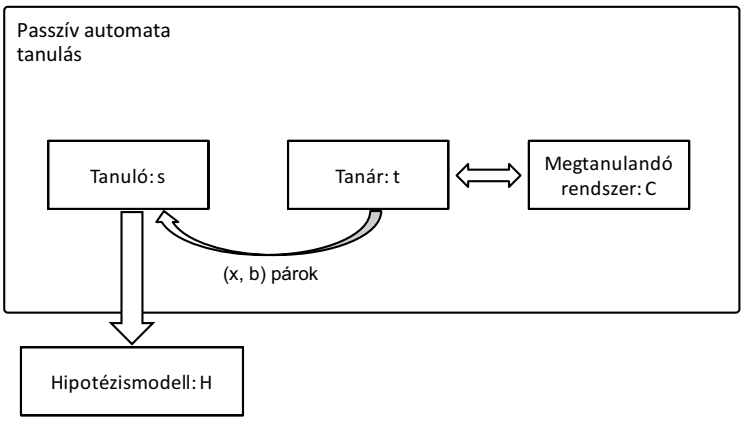
Input: C rendszer és t tanára, aki választ ad a mem() tagsági- és az eq() ekvivalencia kérdésekre
Output: H hipotézismodell C rendszer viselkedéséről

```
Kérdezze meg mem(ϵ) tagsági kérdést, hogy a kezdőállapotról kiderüljön, hogy elfogadó, vagy nem elfogadó
Állítsa elő egy kezdeti H hipotézismodellt, mely ezt az egy állapotot tartalmazza
while eq(H) ellenpéldával tér vissza do
    while Van kérdés a rendszer felé do
        Kérdezze mem(seq)-t a tanártól, ahol seq a kérdéses szó
    end
    Vizsgálja meg az ekvivalenciát: eq(H)
end
return H
```

2.4.2. Passzív automatatanulás

A passzív tanulás főként abban különbözik az aktív tanulástól, hogy annak nincs lehetősége kommunikálni a megtanulandó rendszerrel (2.9. ábra). Egy előre megkapott információhalmazból kell dolgoznia. Nem tehet fel további tagsági kérdéseket, amelyekkel több tudást tudna szerezni.

Passzív tanulás során az algoritmus bemenetként kap számos, eddig már előfordult, megfigyelt lefutást. Ezeket a lefutásokat a tanulás tanárától kapja, így azzal az



2.9. ábra. Passzív automatatanulás

információval is rendelkezik, hogy igazak, vagy sem a megtanulandó rendszerre nézve. Az algoritmus belőlük eleinte egy lefutási fát(32. definíció) épít, majd azt megpróbálja determinisztikusan összevonni, minimalizálni[16].

Definíció 32 (Lefutási fa). A lefutási fa, röviden APTA (Augmented Prefix Tree Acceptor) egy fa automata reprezentációja a tanulás során felhasznált pozitív és negatív elemeknek. Minden egyes elem, lefutás megtalálható a fában, annak egy útvonalaként, mely a fa gyökeréből indul. A pozitív és a negatív lefutásokhoz tartozó útvonalak végét jelző állapotok rendre elfogadóak vagy nem elfogadóak.
Az APTA így tulajdonképpen egy hiányos véges állapotgépnek felel meg, melynek nem minden állapotáról dönthető el, hogy elfogadó-e. ▪

Példa 4. *Legyen* $\Sigma = \{0, 1\}$*, vagyis az ábécé karakterei legyenek 0 és 1. X legyen továbbra is* Σ^**, vagyis a karakterekből képezhető összes szó halmaza. Legyen a C által elfogadott nyelv azon x- ek halmaza, ahol* $x \in X$ *és amely x-ben az* 1*-esek száma kettővel osztható.*

A 4. példában leírt rendszernek a megtanulási folyamatát a 2.10. ábra szemlélteti. A tanulás első fázisában a tanuló megkapja a tanártól a pozitív és negatív lefutásokat. Ezek a pozitív és negatív lefutások most rendre legyenek a következők:

- $S_+ = \epsilon, 11$
- $S_- = 0, 10, 01$

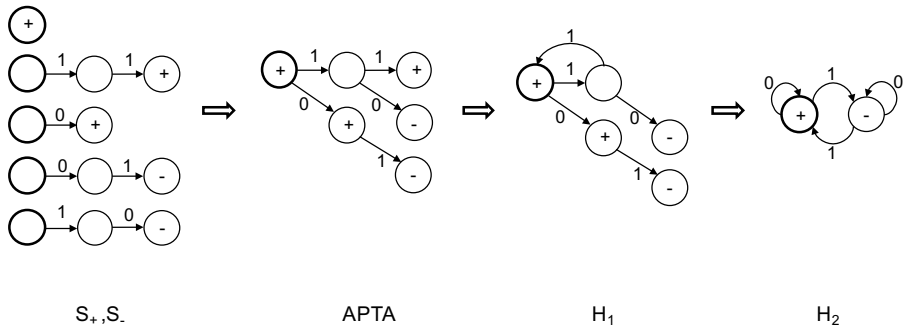
Ezek után az algoritmus felépíti ezeknek megfelelően a lefutási fát. Majd ameddig tud, kiválaszt az automatában olyan állapotokat, amelyeket összevonva nem jut ellentmondásra és csökkentheti az automata méretét.

A passzív algoritmusoknak is sokféle változata létezik, különböző típusú automaták megtanulására. A 2. pszeudokód írja le általánosan egy passzív tanuló algoritmus működését. Ennél konkrétabb megvalósításokról lesz még szó az 3. fejezetben.

2.5. Tulajdonsággráf

A gráf alapú tudásbázis reprezentálására tulajdonsággráfot használok, amely egy irányított gráf, címkézett csúcsokkal és típusos élekkel, továbbá ezek elláthatóak tetszőleges tulajdonságokkal, így alkalmas komplex rendszerek állapotának ábrázolására.

Definíció 33 (Tulajdonsággráf). A *tulajdonsággráfot* (*property graph*) a következő $G = (V, E, \mathsf{src\_trg}, L_v, L_e, l_v, l_e, P_v, P_e)$ struktúra írja le, ahol V a csúcsok halmaza, E az élek



2.10. ábra. Passzív tanulás során felépített hipotézismodellek

Algoritmus 2: Passzív automatatanulás

Input: S pozitív és negatív lefutások halmaza
Output: H DFA, ami kicsi és konzisztens S-sel

```
1 H = apta(S)
2 while van lehetőség összevonni do
3 |  Összevon két állapotot
4 end
5 return H
```

halmaza és src_trg : $E \to V \times V$ az élekhez a kezdő- és a végpontjukat rendeli hozzá. A csúcsok címkével, az éleket típussal látjuk el:

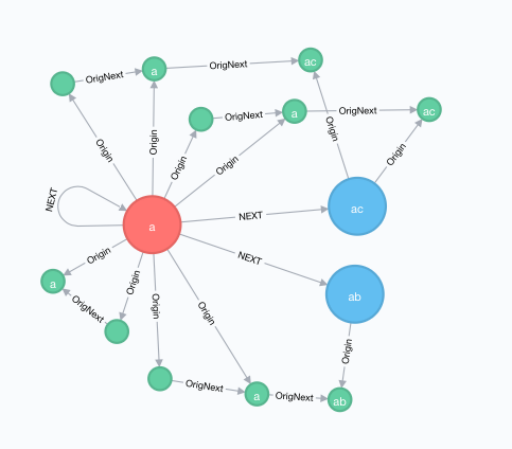
- L_v a csúcsok címkéinek halmaza, $l_v : V \to 2^{L_v}$ minden csúcshoz *címkék egy halmazát* rendeli.
- L_e az élek típusának halmaza, $l_e : E \to L_e$ minden élhez *egyetlen típust* rendel.

A tulajdonságok definiálásához $D = \cup_i D_i$ a D_i elemi tartományok uniója és NULL jelöli a NULL, üres értéket.

- P_v a csúcsok tulajdonságainak halmaza. A $p_i \in P_v$ csúcstulajdonság egy olyan $p_i : V \to D_i \cup \{\text{NULL}\}$ függvény, amely egy tulajdonság értéket rendel a $D_i \in D$ tartományból a $v \in V$ csúcshoz, ha v rendelkezik a p_i tulajdonsággal, egyébként $p_i(v)$ értéke NULL.
- P_e az élek tulajdonságainak halmaza. A $p_j \in P_e$ éltulajdonság egy olyan $p_j : E \to D_j \cup \{\text{NULL}\}$ függvény, amely egy tulajdonság értéket rendel a $D_j \in D$ tartományból a $e \in E$ élhez, ha e rendelkezik a p_j tulajdonsággal, egyébként $p_j(e)$ értéke NULL. [12] ▪

Az automatatanulás során például az automatát és az eredeti időzített lefutásokat is tulajdonsággráfban tárolom.

Példa 5 (Tulajdonsággráf). *Legyen egy tulajdonsággráfban négy féle címkéjű csúcs: State, Blue, Red, Origstate. Az OrigState címke jelölje a megfigyelt lefutások állapotait és ábrázoljuk kis zöld körrel, a State jelölje az automata állapotait és ábrázoljuk nagy körrel, a Red és a Blue jelöljék az automata piros és kék állapotait, jelöljük ezeket a mefelelő színezéssel. Legyen a State és OrigState típusú csúcsoknak is egy-egy trace tulajdonsága (pl.: "a", "ab"). Ezen kívül a tulajdonsággráfnak legyen háromféle éltípusa: OrigNext, Origin és NEXT. Az OrigNext típusú él mutasson OrigState-ből OrigState-be és jelölje*



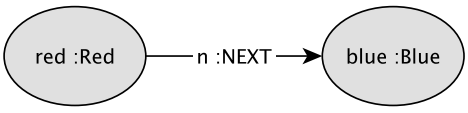
2.11. ábra. Tulajdonsággráf

egy megfigyelt lefutásban a két állapot egymásutániságát. Az Origin egy State címkéjű állapotból mutasson arra az OrigState címkéjű állapotra, melyből keletkezett, a NEXT pedig State csúcsok egymásutániságát jelölje.

A 2.11. ábra szemlélteti a 5. példának megfelelő tulajdonsággráfot. Mint látható a gráfnak vannak csúcsai, amelyekhez különböző címkék társulnak $(State, Blue, Red, Origstate)$. A csúcsok között irányított élek szerepelnek, ezeknek az éleknek típusa van, pl. a-val jelölt $State$-ből a-val jelölt $OrigState$-be mutató él típusa $Origin$, amely azt jelenti, hogy az automata a-val jelölt állapota egy eredeti időzített lefutás a-val jelölt állapotából keletkezett.

2.6. Gráfminta

Gráfadatbázisban tárolt adatok esetében lehetőség van mintaillesztéssel elemek kikeresésére és azokon műveletek végrehajtására gráflekérdezések segítségével. Munkám során gráfmintákat alkalmazok, hogy a gráfadatbázisban tárolt automatáról lekérdezésekkel információt szerezzek, illetve, hogy módosítsam azt.



2.12. ábra. Gráfminta

```
MATCH (red:Red)-[n:NEXT]->(blue:Blue)
```

2.1. kód. Gráflekérdezés

A 2.12. ábrán látható egy példa gráflekérdezés. Ez a példa olyan gráfmintára keres rá, ahol egy Red címkéjű csúcsból mutat $NEXT$ típusú él egy $Blue$ címkéjű csúcsba. A 2.11. ábrán ábrázolt tulajdonsággráfon ez a gráfminta az a és ab, illetve az a és ac-vel jelölt csúcspárokra illeszkedne.

2.7. Lekérdezőnyelv gráfadatbázisokhoz

Munkám során Cypher[13] gráflekérdező nyelvet használtam. A 2.12. ábrán látható gráfminta ezen a nyelven a 2.1. kódnak megfelelően implementálható.

A gráflekérdező nyelv néhány fontosabb, a szokásos lekérdező nyelvektől eltérő, későbbiekben használt jelöléseit, kulcsszavait szeretném bemutatni.

(red:Red)
: A kód ezen részében görbe zárójelek között tudunk egy csomópontra, állapotra hivatkozni. A *red* a csomópont nevét jelöli a lekérdezésben, míg a *Red* annak a címkéjét.

-[n:NEXT]->
: Szögletes zárójellel lehet jelölni az éleket és a nyíl irányával meghatározni annak irányát. A kódban az n az él, lekérdezésben használt, nevét jelenti, a *NEXT* az él típusát.

-[n:NEXT0..*]->
: Ez a lekérdezés az előzőtől annyiban tér el, hogy tranzitív lezártat keres, az él kiinduló állapotából 0, vagy több *NEXT* típusú élen keresztül.

DETLETE és DETACH DELETE
: A *DELETE* paranccsal törölhető az adatbázisból egy csomópont, míg a *DETACH DELETE* paranccsal egy csomópont, annak összes kapcsolatával együtt.

3. fejezet

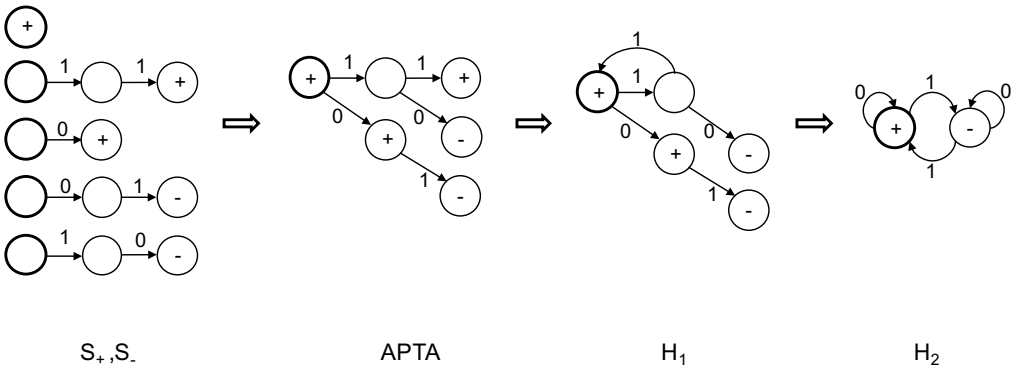
Automatatanuló algoritmus

Ebben a fejezetben bemutatom az általam használt tanulóalgoritmus működését először az időzítés nélküli, majd az időzítéseket is figyelembe vevő esetre.

Munkám során egy pozitív és negatív elemeket is felhasználó, passzív, offline tanulóalgoritmust használok, amely képes előállítani valósidejű automatát, amelynek minden állapota vagy elfogadó vagy elutasító.

3.1. Algoritmus véges automatára

Ahogy arról már a 2.4.2. fejezetben szó volt, a passzív tanulás már meglévő lefutások halmazával dolgozik, azokból először egy APTA-t (32. definíció) készítve, majd azt összevonva azt egy véges állapotgéppé (9. definíció). A 3.1. ábrán látható egy áttekintő folyamatábra a 4. példában leírt rendszernek a megtanulásáról. Ebben a példában a cél az volt, hogy a páros számú 1-es karaktert tartalmazó szavakat fogadja el a megtanult automata.



3.1. ábra. Passzív tanulás folyamata

3.1.1. APTA előállítása

A tanulási folyamat első lépése tehát az APTA előállítása, vagyis a lefutások összevonása egy fává. Az előállítás során feltételezzük azt, hogy a lefutásaink nem ellentmondásosak, vagyis nem létezhet két olyan lefutás, melyekben a bemeneti események és azok sorrendje azonos, azonban az egyik elfogadó, míg a másik nem elfogadó állapotba vinné az automatát.

Az APTA előállításának a menetét a 3. pszeudokód szemlélteti. A folyamat első lépéseként az összes lefutás kezdőállapotát összevonjuk, ez lesz a fa gyökere. Ezek után

a gyökértől távolodva, ameddig találunk olyan állapotokat, melyek a gyökértől egyenlő távolságra vannak és onnan megegyező karaktersorozatra érhetőek el, összevonjuk. Ha nincsenek már ilyen állapotok, készen vagyunk. A pszeudokódban $s_{i,j}i$ az i. lefutás j. állapotát jelöli, a az ábécé egy karakterét, q_i pedig az APTA i. állapotát.

Algoritmus 3: APTA előállítása: *apta*

Input: $S = \{S_+, S_-\}$ pozitív és negatív lefutások halmaza (összesen k db)
Output: H lefutási fa, amely tartalmazza S összes lefutását

$q_0 = osszevon(\sum_{i=0}^{k} s_{i,0})$
while $\exists q_i, s_{a,j}, s_{b,j}, ..., s_{n,j}$, *amire* $s_{a,j} = \delta(q_i, a)$, $s_{b,j} = \delta(q_i, a)$, ..., $s_{n,j} = \delta(q_i, a)$ **do**
    $osszevon(s_{a,j}, s_{b,j}, ..., s_{n,j})$
end
return H

Az állapotok összevonása (4. algoritmus) a következőképpen zajlik: létrehozzuk az új állapotot, belemásoljuk a régi állapotok bemenő és kimenő éleit (ha voltak), majd beállítjuk a jellemzőjét. Itt szintén feltételezve azt, hogy a lefutásainkban nincs ellentmondás. Ezt követően töröljük a már összevont állapotokat.

Algoritmus 4: Állapotok összevonása: *osszevon*

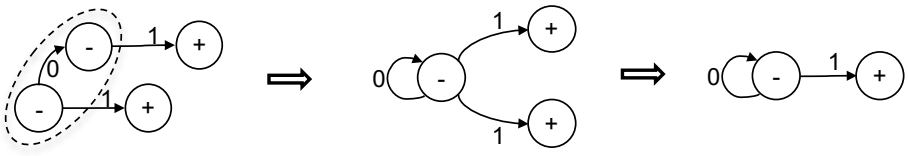
Input: $s_1, s_2, ...s_n$ összevonandó állapotok
Output: q állapot

q állapot létrehozása
for $i \in s_1, s_2, ..., s_n$ **do**
    if $\exists \delta(r, a) = s_i$ **then**
        $\delta(r, a) = q$ állapotátmenet létrehozása
    end
    if $\exists \delta(s_i, a) = r$ **then**
        $\delta(q, a) = r$ állapotátmenet létrehozása
    end
    if s_i *elfogadó* **then**
        q beállítása elfogadóra
    end
    if s_i *nem elfogadó* **then**
        q beállítása nem elfogadóra
    end
    s_i állapot és állapotátmenetei törlése
end
return q

3.1.2. Merge

A passzív tanuló algoritmus következő lépése, hogy az APTA-nak, ameddig csak tudja, csökkentse a méretét. Illetve, hogy általánosítsa azt. Ezt úgynevezett *merge* műveletekkel tudja megtenni. Ez a művelet két lépésből áll, először a kijelölt két állapotot összevonja, majd az így okozott inkonzisztenciákat, nemdeterminizmusokat feloldja. Egy *merge* csak akkor végrehajtható, hogyha sem az összevonás, sem az azt követő inkonzisztencia feloldás

során nem vonunk össze ellentétes állapotokat. Vagyis a művelet során egy elfogadó és egy nem elfogadó állapot összevonása nem megengedett.



3.2. ábra. A *merge* művelet

A a 5. pszeudokód bemutatja a *merge* művelet működését, a 3.2. ábra pedig egy példán személelti a végrehajtást.

Algoritmus 5: Merge DFA tanulás során: *merge*

Input: H DFA két állapota: q q'
Output: *igaz* ha a merge lehetséges, *hamis* egyébként

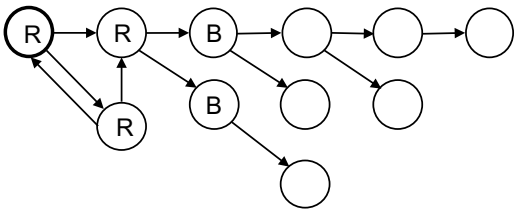
```
1  if q elfogadó és q' nem elfogadó, vagy fordítva then
2  |  return hamis
3  end
4  összevon(q, q')
5  while H automatának van nemdeterminisztikus átmenete q_n és q'_n állapotokba do
6  |  boolean b = merge(q_n, q'_n)
7  |  if b hamis then
8  |  |  eddigi merge műveletek visszavonása
9  |  |  return hamis
10 |  end
11 end
12 return igaz
```

3.1.3. Red-Blue eljárás

Annak érdekében, hogy a tanulás gyorsabb legyen, az algoritmus a Red-Blue eljárást[16] használja. Ez azáltal gyorsítja a passzív automatatanulást, hogy *merge* esetén nem vizsgálja meg az összes lehetséges q és q' párost, nem próbál meg minden állapotpárt összevonni, hanem a fa gyökere felől indulva, mindig az ahhoz közelebb eső új csúcsokat próbálja összevonni a gyökérhez közelebb esőkkel. Ennek a megvalósításához a csúcsokat két színnel színezi: pirossal jelöli azokat az állapotokat, amelyek egymással már biztosan nem összevonhatók, kékkel pedig a következő jelölteket. Futás során csak az általa kékre színezett állapotokat vizsgálja meg és párosítja az összes pirosra színezett állapottal. A színezéseket a következőképpen alakul.

Az algoritmus kezdetben az automata kezdőállapotát pirosra színezi, majd ennek az állapotnak a közvetlen gyerekeit kékre. Innentől kezdve minden lépést követően a új piros állapotok közvetlen gyerekei kékek lesznek. Csak kék állapot színeződhet át pirosra – vagy azért, mert végrehajtunk egy *merge* műveletet és ilyenkor az összevont állapot piros lesz, vagy mert nincs megengedett *merge*, így a kék állapotot át kell színeznünk pirosra.

A 3.3. ábra szemlélteti az algoritmust. A pirosra színezett állapotokat R jelöli, a kékeket B. Ez a színezés tehát úgy értelmezhető, hogy a piros állapotokat már nem fogja módosítani az algoritmus, azok a végső automatában is benne lesznek. A kék állapotok azok, melyeket az algoritmus az adott iterációban vizsgál, a nem színezettekkel pedig csak



3.3. ábra. A Red-Blue algoritmus

a későbbiekben fog foglalkozni. Ennek megfelelően az automatatanulás akkor fejeződik be, ha az automata minden állapota piros.

3.1.4. Color

Az előző fejezetben megjelent az átszínezés művelete. Ez akkor kerül végrehajtásra, ha nincsen olyan kék-piros állapotpár, melyet össze lehetne vonni. Ilyenkor egy kék állapot egyszerűen pirossá válik. Ennek a műveletnek a neve *color*.

3.1.5. Evidence driven state merging

Az automatatanuló algoritmustól azt is elvárjuk, hogy az általa megtanult automatának az állapottere minél kisebb legyen. Erre az evidence driven state merging[9][5] (EDSM) eljárás megoldást ad. Ennek az eljárásnak, a nevének megfelelően az a lényege, hogy a lehetséges *merge* műveletekhez valamilyen metrikát rendel, majd azt a műveletet hajtja végre, amelyik a legnagyobb metrikával rendelkezik. Ezt a metrikát pedig az alapján számolja, hogy milyen jónak, hasznosnak, biztosnak ítéli meg a *merge* műveletet.

Az algoritmus működése a következőképpen alakul: megnézi, hogy egy $merge(q, q')$ során mely állapotpárok lesznek majd összevonva. A metrikát eleinte $m = 0$ inicializálja, majd a q_i, q'_i állapotpárok alapján a következőképpen módosítja azt:

- ha q_i elfogadó és q'_i nem elfogadó, vagy fordítva, akkor $m = -\infty$
- ha q_i és q'_i is elfogadó, vagy mindkettő nem elfogadó, akkor m értékét növeli 1-gyel
- q_i vagy q'_i nem ismert, hogy elfogadó-e, akkor nem változtat m értékén.

Az így kapott legmagasabb értékű *merge* műveletet fogja végrehajtani az algoritmus. Ha a legmagasabb metrika a $-\infty$ lenne (vagyis nincs lehetséges *merge*), akkor *color* műveletet hajt végre. Ezzel tulajdonképpen 0 metrikát, értéket tulajdonítva a *color* műveletnek.

3.1.6. Az algoritmus

A fent leírt műveletek és eljárások alapján az általam megvalósított passzív automatatanuló algoritmus a 6. pszeudokód alapján működik.

3.2. Algoritmus valósidejű automatára

A tanuló algoritmus valósidejű automatákra a véges automatákat megtanuló algoritmus kiegészítése. Különbség, hogy az algoritmus időzített lefutási fát használ, időzítetlen helyett. Ebben az időzített lefutási fában lehetnek inkonzisztens állapotok, vagyis olyan állapotok, melyek egyszerre elfogadóak és nem elfogadóak is. Ez akkor lehetséges, ha ugyan az a karaktersorozat más-más időzítéssel is szerepel az eredeti lefutásokban. Emiatt

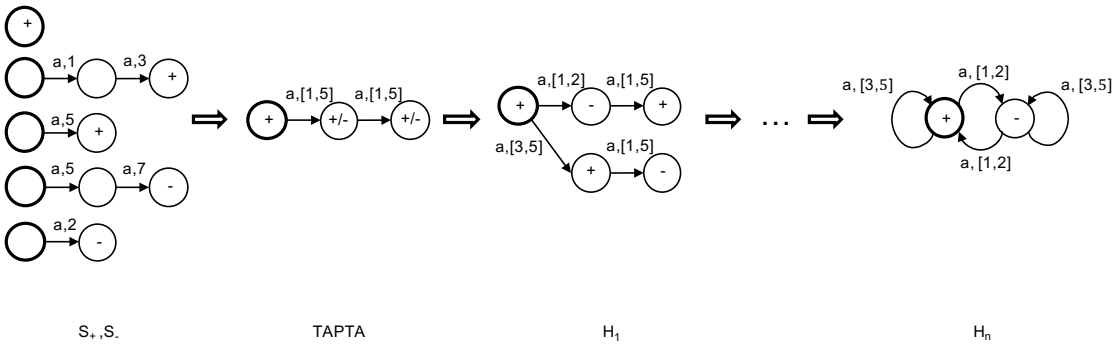
Algoritmus 6: DFA tanulóalgoritmus

Input: $S = \{S_+, S_-\}$ pozitív és negatív lefutások halmaza
Output: H kis állapotterű DFA, melyre igazak S elemei

```
H = apta(S)
q_0, a kezdőállapot pirosra színezése
q_0 közvetlen gyerekeinek kékre színezése
while H automatában van még nem piros állapot do
    Lehetséges merge műveletek metrikájának a kiszámítása
    if merge(r, b) rendelkezik a legmagasabb pozitív metrikával then
        merge(r, b)
    else
        color()
    end
    Új piros állapotok gyerekeinek kékre színezése
end
return H
```

az inkonzisztencia miatt időzített tanulás esetén szükséges bevezetni még egy műveletet. Ez az új művelet a *split*, amely fel tudja oldani az inkonzisztenciákat azáltal, hogy egy időzített élet felbont két újabb időzített élre (más időkorlátokkal). Az új művelet és az inkonzisztenciák miatt a metrikák számítása is módosul.

A 3.4. ábra szemléltet egy passzív valósidejű automatatanuló folyamatot. Az ábrában lévő algoritmus a 2. példában szereplő nyelvet tanulja. Ebbe a nyelvbe azok a szavak tartoznak, melyekben páros számú a karakter szerepel, legalább 1, legfeljebb 2 időközzel. A tanulás bemenete néhány pozitív és néhány negatív lefutás (S_+, S_-). Ezekből megépíti az időzített lefutási fát, majd *merge*, *split* és *color* műveleteket követően előállítja H_n végleges hipotézismodellt.



3.4. ábra. Valósidejű automata passzív tanulásának a folyamata

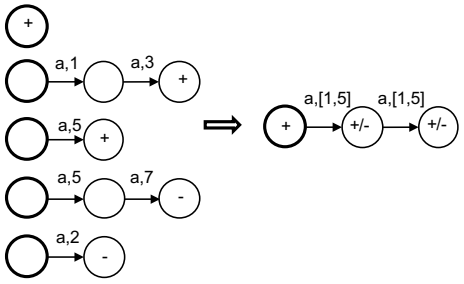
A következőkben az eddig definiált *merge* és *color* műveletek felül lesznek definiálva az időzített megfelelőikre, így a továbbiakban mikor ezekre a műveletekre hivatkozom, az időzített megfelelőjét értem alatta.

3.2.1. TAPTA előállítása

Időzített tanulás esetén olyan lefutásaink vannak, melyek a bemeneti események idejéről is hordoznak információt. Éppen ezért ezekből a lefutásokból nem elég időzítetlen lefutási fát generálnunk, hiszen így információt veszítenénk. Emiatt az APTA egy módosított változatát, időzített lefutási fát generálunk.

Definíció 34 (Időzített lefutási fa). Az időzített lefutási fa, röviden TAPTA (Timed Augmented Prefix Tree Acceptor) egy időzített fa automata reprezentációja a tanulás során felhasznált pozitív és negatív elemeknek.
Minden egyes elem, lefutás megtalálható a fában, annak egy útvonalaként, mely a fa gyökeréből indul. A pozitív és a negatív lefutásokhoz tartozó útvonalak végét jelző állapotok rendre elfogadóak vagy nem elfogadóak.
A fa állapotátmenetein az időkorlátok mind ugyan azok lesznek. Az időkorlát alsó határa a lefutásokban, azaz a $(a_1, t_1)(a_2, t_2)...(a_n, t_n)$ időzített szavakban előforduló legkisebb relatív időeltérés, a felső határa pedig a legnagyobb.
A TAPTA így tulajdonképpen egy hiányos valósidejű automatának felel meg, melynek nem minden állapotáról dönthető el, hogy elfogadó-e.

A 3.5. ábra az $S_+ = \{(\epsilon), (a,1)(a,2), (a,5)\}, S_- = \{(a,5)(a,2), (a,2)\}$ lefutásokból képzett TAPTA-t szemlélteti. Az időzített lefutási fa minden élén $[1,5]$ időkorlát van, hiszen a lefutásokban a legkisebb előforduló relatív időeltérés 1, míg a legnagyobb 5 volt.



3.5. ábra. Időzített lefutási fa

A TAPTA tehát annyiban különbözik az APTA-tól, hogy állapotátmenetein időkorlátok is vannak. A TAPTA előállítását a 7. pszeudokód írja le.

Algoritmus 7: TAPTA előállítása: *tapta*

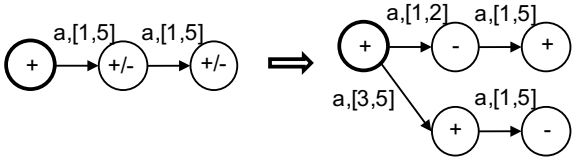
Input: $S = \{S_+, S_-\}$ pozitív és negatív időzített lefutások halmaza (összesen k db)
Output: H_t időzített lefutási fa, amely tartalmazza S összes lefutását

```
1 q_0 = osszevon(∑_{i=0}^{k} s_{i,0})
2 while ∃q_i, s_{a,j}, s_{b,j}, ..., s_{n,j} , amire s_{a,j} = δ(q_i, a), s_{b,j} = δ(q_i, a), ..., s_{n,j} = δ(q_i, a)
  do
3 |  osszevon(s_{a,j}, s_{b,j}, ..., s_{n,j})
4 end
5 T_min = S halmazban előforduló legkisebb időköz
6 T_max = S halmazban előforduló legnagyobb időköz
7 ∀d állapotátmenet időkorlátjának beállítása [T_min, T_max]-ra
8 return H_t
```

3.2.2. Split

A *split* művelet célja, hogy az automatát konzisztenssé tegye az inkonzisztens állapotokba vezető élek kettévágásával. A 3.6. ábrán látható egy példa a műveletre. A valósidejű automata eleinte két inkonzisztens állapotot is tartalmaz, majd a balról első élet $t = 2$ időpillanatban kettévágva megszűnik mindkét inkonzisztencia.

A 8. pszeudokód leírja a *split* művelet működését t időpillanatban. A művelet törli a kettévágandó állapotátmenetet és annak végállapotából kiinduló részfát. Majd két új



3.6. ábra. A *split* művelet

részfát hoz létre az eredeti lefutások felhasználásával. Végül pedig összeköti a megfelelő részfákat az eredeti él kiinduló állapotával. Ezeknek az összekötő éleknek az időkorlátai $[n, t]$ és $[t + 1, n']$ lesznek (amennyiben a kettévágott él időkorlátja $[n, n']$ volt).

Algoritmus 8: Split RTA tanulás során: *split*

Input: H_t RTA $d = \langle q, q', a, [n, n'] \rangle$ időzített állapotátmenete és $t \in [n, n']$ idő

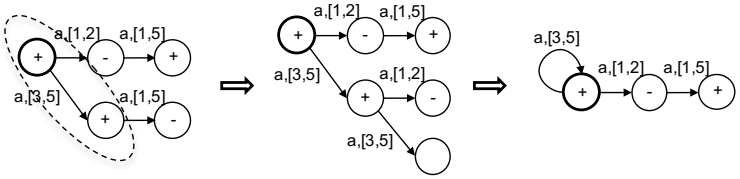
1 d állapotátmenet törlése
2 q' állapotból induló részfa törlése
3 q_1 és q_2 új állapotok létrehozása
4 $d_1\langle q, q_1, a, [n, t] \rangle$ állapotátmenet létrehozása
5 $d_2\langle q, q_2, a, [t + 1, n'] \rangle$ állapotátmenet létrehozása
6 q_1 kezdőállapottal $tapta(S^{d_1})$
7 q_2 kezdőállapottal $tapta(S^{d_2})$

3.2.3. Merge

A *merge* művelet lényege ugyan az, mint időzítetlen esetben. Két állapot összevonására szolgál, illetve az ezáltal okozott nemdeterminizmusok megszüntetésére. Azonban a *split* művelet bevezetésével számos változtatást be kell vezetni a *merge*-ben.

- Mivel a *split* művelet lehetővé teszi az inkonzisztens állapotok konzisztensé alakítását, így *merge* esetén csak akkor kell feltétlen megtiltani az elfogadó és nem elfogadó állapotok összevonását, ha abból az egyik piros állapot (3.2.4. fejezet). Ez a változás az EDSM metrika számításának módosításaként valósul meg.
- A *split* művelet bevezetésével előfordulhat olyan eset, hogy *merge* során két, q és q' összevonandó állapot közül q-ból a karakterrel több állapotátmenet indul, különböző időkorlátokkal, míg q'-ből csupán egy. Ilyenkor a q'-ből a karakterrel induló állapotátmeneten is végre kell hajtani egy *split*-et.

Az 3.7. ábra szemléltet egy *merge* műveletet időzített esetben. A két összevonandó állapotból az időkorlátok szempontjából nem azonos állapotátmenetek indulnak. Így a *merge* végrehajtása előtt még végre kell hajtani egy *split* műveletet.



3.7. ábra. *merge* művelet során *split* végrehajtása

A 9. pszeudokód leírja a módosult *merge* működését. Az algoritmusban kikötés, hogy a q' állapot nem piros – ez a Red-Blue eljárás módosulásából következik (3.2.4. fejezet).

Algoritmus 9: Merge RTA tanulás során: *merge*

Input: H_t RTA két állapota: q, q'
Output: *igaz*, ha a *merge* lehetséges, *hamis* egyébként

```
if q vagy q' piros then
    if q elfogadó és q' nem elfogadó, vagy fordítva then
        return hamis
    end
end
if ∃a, amire d1 = ⟨q, q1, a, [n, t]⟩ és d2 = ⟨q', q1, a, [n, n']⟩, ahol t < n' then
    split(d2, t)
end
osszevon(q, q')
while Ht automatának van nemdeterminisztikus átmenete qn és q'n állapotokba do
    boolean b = merge(qn, q'n)
    if b hamis then
        eddigi merge műveletek visszavonása
        return hamis
    end
end
return igaz
```

3.2.4. Red-Blue eljárás

A Red-Blue eljárásnak az lényege időzített esetben is az algoritmus futásidejének csökkentése. Az eljárás garantálja azt, hogy a korábbi lépésekben csak a piros állapotok voltak módosítva, illetve hogy a már pirosra színezett állapotok nem fognak változni a tanulás további lépéseiben.

Éppen ezért nincs megengedve a *merge* művelet esetén a piros állapotok inkonzisztens állapotba hozása. Illetve az eljárást garantálja, hogy csak piros élekből kimenő éleken lehetett korábban *split* művelet végrehajtva. Így például mikor *merge* esetén vizsgáljuk, hogy szükséges-e *split* művelet, csak a piros állapotokból induló állapotátmeneteket kell figyelni.

3.2.5. Color

A *color* művelet megegyezik az időzítetlen változatával. Ha a metrika értéke erre a műveletre a legnagyobb, akkor egy kék állapotot pirosra színez az algoritmus.

3.2.6. Evidence driven state merging

A *split* művelet sok változást hoz a többi művelethez tartozó metrika kiszámításában is. Időzített esetben a következőképpen alakul az m metrika kiszámítása. Minden esetben m értékét 0-ra inicializálja az algoritmus, majd az alábbiaknak megfelelően módosítja azt.

Merge: A *merge* művelet során összevonandó q_i, q'_i állapotpároktól függően

- ha q_i elfogadó és q'_i nem elfogadó, vagy fordítva és q_i vagy q'_i piros, akkor a metrika $m = -\infty$ lesz
- ha q_i elfogadó és q'_i nem elfogadó, vagy fordítva, akkor m értéki csökkenti 1-gyel
- ha q_i és q'_i is elfogadó, vagy mindkettő nem elfogadó, akkor m értékét növeli 1-gyel

- q_i vagy q_i' nem ismert, hogy elfogadó-e, akkor nem változtat m értékén.

Split: A *split* művelet során kettébontott q_i, q_i' állapotpároktól függően

- ha q_i elfogadó és q_i' nem elfogadó, vagy fordítva, akkor m értéki növeli 1-gyel
- ha q_i és q_i' is elfogadó, vagy mindkettő nem elfogadó, akkor m értékét csökkenti 1-gyel
- q_i vagy q_i' nem ismert, hogy elfogadó-e, akkor nem változtat m értékén
- ha q_i vagy q inkonzisztens marad, akkor nem változtat m értékén.

Color: A *color* művelet minden esetben $m = 0$ értékű.

3.2.7. Az algoritmus

Az általam megvalósított időzített rendszerekre alkalmazható passzív automatatanuló algoritmust a 10. pszeudokód írja le.

Algoritmus 10: RTA tanulóalgoritmus

Input: $S = \{S_+, S_-\}$ pozitív és negatív időzített lefutások halmaza
Output: H_t kis állapotterű RTA, melyre igazak S elemei

```
1  H_t = tapta(S)
2  q_0, a kezdőállapot pirosra színezése
3  q_0 közvetlen gyerekeinek kékre színezése
4  while H_t automatában van még nem piros állapot do
5  |  Lehetséges merge műveletek metrikájának a kiszámítása
6  |  Lehetséges split műveletek metrikájának a kiszámítása
7  |  Lehetséges color műveletek metrikájának a kiszámítása
8  |  if merge(r, b) rendelkezik a legmagasabb pozitív metrikával then
9  |  |  merge(r, b)
10 |  end
11 |  if split(d, t) rendelkezik a legmagasabb pozitív metrikával then
12 |  |  split(d, t)
13 |  end
14 |  if color() rendelkezik a legmagasabb pozitív metrikával then
15 |  |  color()
16 |  end
17 |  Új piros állapotok gyerekeinek kékre színezése
18 end
19 return H_t
```

4. fejezet

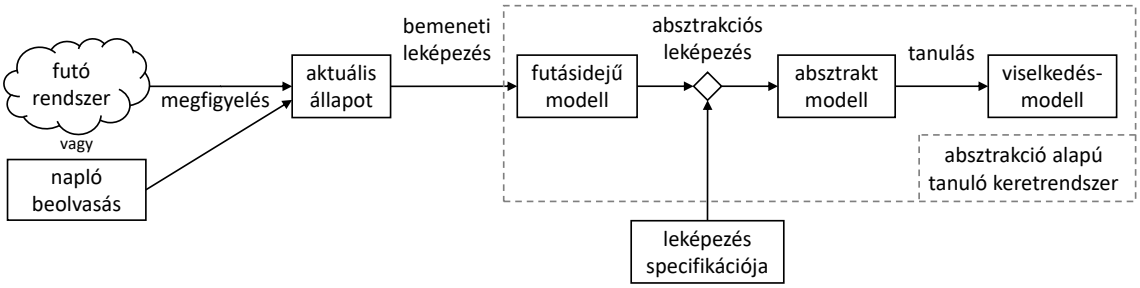
Megvalósítás

Ebben a fejezetben bemutatom az automatatanuló algoritmus implementálását. Ez az algoritmus korábban nem volt megvalósítva, csak elméleti szinten volt végiggondolva, így implementálása során számos kihívással szembesültem.

Az algoritmus használhatóságát egy évfolyamtársam által készített keretrendszerbe belehelyezve vizsgáltam meg. Ez a keretrendszer komplex rendszerek gráfminta alapú absztrakcióját valósítja meg. Az absztrakciós komponens és az időzített automatatanuló algoritmus segítségével komplex, időfüggő rendszerek viselkedését tudjuk modellezni.

4.1. A keretrendszer bemutatása

Ebben a fejezetben bemutatom a keretrendszer megvalósítása során felhasznált technológiákat, a megvalósított komponenseket, illetve a fontosabb implementációs részleteket.

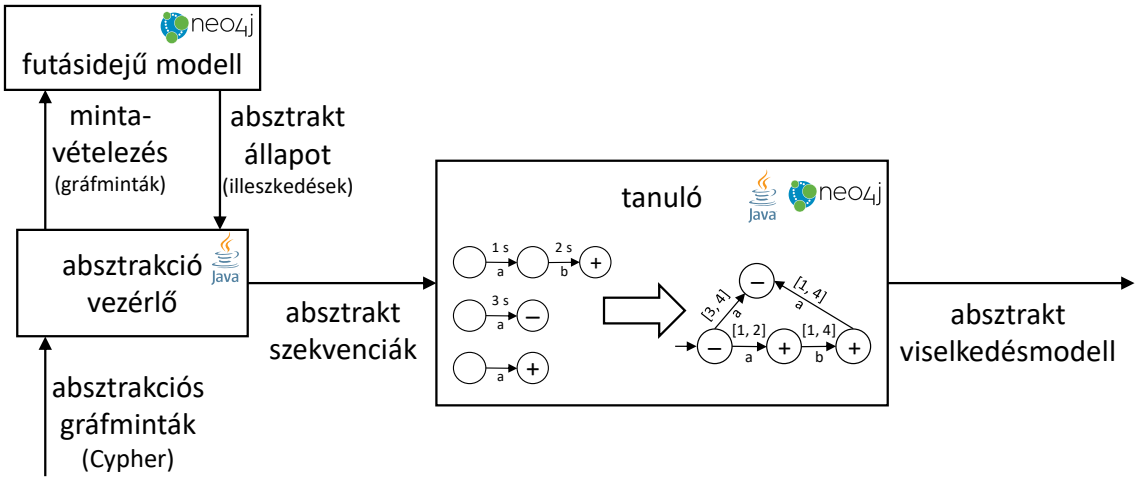


4.1. ábra. Absztrakció alapú tanuló keretrendszer

Az általunk megvalósított keretrendszer egy időzített rendszer működése során annak aktuális állapotából származtatott futásidejű modell alapján képes viselkedésmodellt alkotni (4.1. ábra). A bemenet egy gráf alapú futásidejű modell, amelyet célzott tanuláshoz használunk fel. A felhasználó gráfmintákkal specifikálhat egy absztrakciós leképezést. Az így képzett absztrakt modell felhasználható automatatanulásra, amely egy időzített viselkedésmodellt állít elő. Így lehetővé válik komplex, parametrikus rendszerek egy kiemelt részletének, az időfüggő viselkedést is figyelembe vevő vizsgálata.

4.2. Architektúra

A megvalósított keretrendszer két fő részből, egy absztrakciós és egy tanuló komponensből áll. A keretrendszer felépítése a 4.2. ábrán látható.



4.2. ábra. A keretrendszer felépítése

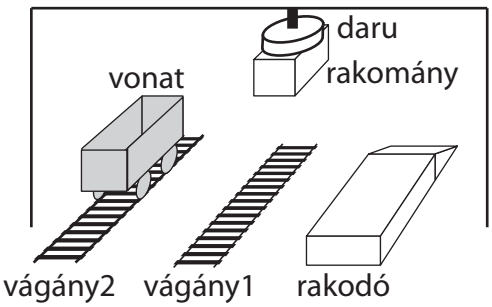
Az absztrakciós komponens a gráf formájában rendelkezésre álló futásidejű modell mintavételezésével képez absztrakt szekvenciákat. Ehhez a felhasználó a célzott tanulást vezérlő gráfmintákat definiál, amelyeket aztán a rendszer mintavételezett állapotain gráfminta-illesztéssel lehet detektálni. A kimeneti szekvenciákat alkotó absztrakt állapotokat a minták illeszkedései határozzák meg. Erre a célra évfolyamtársam Neo4j gráfadatbázist alkalmazott, amelyhez az absztrakciós gráfmintákat a Neo4j Cypher nyelvén lehet megfogalmazni.

Az absztrakt szekvenciák alapján a tanuló komponens képes absztrakt viselkedésmodellt alkotni. A tanulás során használt algoritmus gráf alapú modellen végez átalakításokat, amelyek hatékonyan fogalmazhatóak meg Cypher nyelv segítségével, így ennél a lépésnél én is a Neo4j gráfadatbázist alkalmazom. A komponens a bemenetként kapott absztrakt szekvenciákat a gráfadatbázisba tölti, ezen lekérdezések segítségével módosításokat hajt végre, majd az előálló viselkedésmodellt visszatölti az adatbázisból.

A Neo4j [14] egy népszerű NoSQL tulajdonsággráf-adatbázis, amelyben a Cypher [13] lekérdező nyelven lehet gráfmintákat leírni. A Cypher egy magas szintű deklaratív lekérdező nyelv, amelyben ASCII art-hoz hasonló szintaxissal lehet gráfokat leírni.

4.3. Esettanulmány

Esettanulmányként egy vasúti állomást használunk, ahol vonatok érkeznek rakománnyal, amelyet egy daru segítségével tud a kezelő áthelyezni a kocsik között (4.3. ábra).



4.3. ábra. Vasúti esettanulmány áttekintő ábrája

Működés során különböző szenzorok segítségével figyeljük meg, hogy az egyes komponensek milyen állapotban, milyen helyzetben vannak. A működés során megfigyelünk veszélyes helyzeteket, például ha egy rakományt nagy magasságban engedtek el. Ezekben a helyzetekben vészleállást és helyreállítást követően folytatódik a munka.

A megfigyelt információkat rögzítjük, azokon absztrakciót alkalmazunk. A rendszer lefutásait megkülönböztetjük aszerint, hogy történt-e baleset a lefutás során vagy nem, így el lehet különíteni a rendszer szabályos és szabálytalan viselkedéseit. Illetve a lefutásokhoz időbélyeget is rendelünk, hogy a rendszer modellezése során az időbeliséget is figyelembe tudja venni a tanuló algoritmus.

A megfigyelést és az absztrakciót követően a lefutások, amelyekkel a tanuló algoritmus dolgozik sztring és integer párok sorozata lesz (pl.: $(\text{"}a\text{"}, 1)(\text{"}b\text{"}, 2)$).

A továbbiakban egy olyan esettanulmányt fogok végigvezetni, amelyben az absztrakciós komponens egy rakomány mozgását figyeli meg. A rakományt mozgató daru specifikációjában a daru sebességére vonatkozóan is szerepelnek megkötések. A rakomány felfele irányuló mozgását az absztrakciós komponens "a", oldalirányú mozgását "b", lefele mozgását pedig "c" karakterré alakítja. A daru a megfigyelés során a specifikációnak megfelelően működik, ennek ellenére egyes esetekben a rakomány megsérül. Időzített automatatanulás segítségével szeretném a hiba okát felfedni.

4.4. Tanuló komponens

A tanulóalgoritmus megvalósítása során a Neo4j gráfadatbázison dolgoztam. Cypher lekérdezések segítségével hajtok végre az adatbázison lekérdezéseket, illetve módosítom is azt. Az algoritmus (10. pszeudokódnak megfelelő) logikája Java nyelven van megvalósítva. Azonban ennek egyes részletei gráflekérdezéseket takarnak. Így tehát a tanulóalgoritmus megvalósítása két nagyobb részre bontható.

Algoritmus logikája

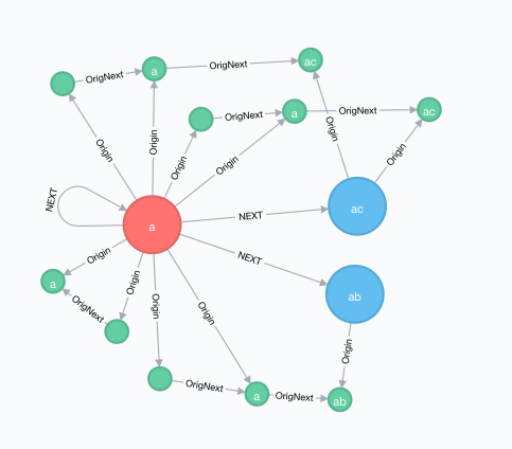
A java nyelven megírt kód felelős azért, hogy az adatmodellen a lekérdezések jó sorrendben fussanak le. Először a *tapta* előállításához szükséges lekérdezések kerülnek végrehajtásra, majd a piros és kék állapotokat beszínező lekérdezések, ezt követően pedig az egyes *merge*, *split* és *color* műveletek metrikái folyamatosan számítódnak, és a legmagasabb metrikával rendelkező művelet kerül végrehajtásra módosító adatbázislekérdezések segítségével.

Adatmodell

A tanulás során mind az aktuális hipotézismodell, mind a rendelkezésre álló lefutások gráfadatbázisban vannak eltárolva.

A 4.4. ábra szemléltet egy általam használt adatmodellt. Az adatmodellen kétféle csúcs és háromféle él látható. Az ábrán a H hipotézismodell állapotait piros illetve kék csúcsok jelölik,a zöld csúcsok pedig az eredeti S szekvenciák állapotainak felelnek meg. A H hipotézismodell állapotátmeneteinek megfelelő élekre $NEXT$ van írva, a H állapotait a megfelelő szekvenciabeli állapotokhoz kötő élek, pedig *Origin* címkével vannak ellátva. Emellett az ábrán szerepelnek *OrigNext* élek is, melyek a szekvenciák állapotátmeneteit jelölik.

Az implementáció kihasználja a Neo4j által kínált lehetőségeket: a gráf csomópontokon és az éleken is szerepelnek címkék, illetve tulajdonságok. A piros és kék állapotok például címkékkel vannak megkülönböztetve. Az állapotokhoz ezen kívül tartoznak *trace* tulajdonságok, melyek azt tartalmazzák, hogy az eredeti lefutásokban az adott állapotig



4.4. ábra. Egy Neo4j-ben tárolt egyszerű H hipotézismodell és S eredeti lefutások ábrázolása

milyen karakterek szerepeltek. Emellett egy állapotnak tulajdonsága még egy *id* mező is, mely a gráfadatbázis lementésénél és újratöltésénél játszik szerepet.

Egy-egy művelet megvalósítása általában több lekérdezés hívásából épül fel. A továbbiakban bemutatom az automatatanulás lépéseinek megvalósítását.

Megjegyzés: A kódrészletek egy részét részletesen bemutatom ebben a fejezetben, a többit függelékként csatolom.

4.4.1. Időzített lefutási fa előállítása

A tanuló algoritmus első lépése az időzített lefutási fa előállítása. Ehhez először időzített lefutásokra van szükség. A következőkben a tanuló algoritmus műveletei az esettanulmány egy egyszerű példáján keresztül kerülnek bemutatásra.

Lefutások

A keretrendszerben lehetőségünk van a tanuló algoritmus bemenetét megadni. A 4.1. kódban leírtaknak megfelelően tudjuk megtenni. Ebben a kódrészletben négy időzített lefutást adunk meg. A lefutások az 4.3. fejezetben megadott háromféle absztrakt eseményt jelentő karaktert tartalmaznak: ”a”, ”b” és ”c”-t. Ezen események között 1, illetve 2 egységnyi idő telik el.

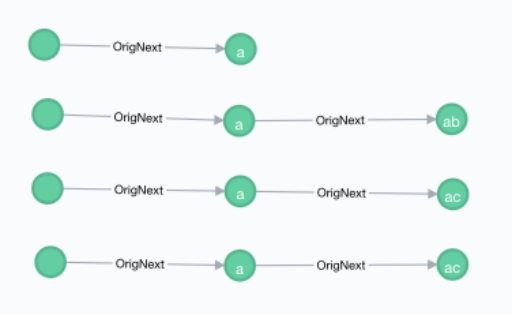
Megjegyzés: Az alábbi példabemenet a könnyebb érthetőség miatt lett leegyszerűsítve, a valódi bemenet több, hosszabb szekvenciát tartalmaz.

```
sequences.addNewRelative(Accept).add("a", 1);
sequences.addNewRelative(Accept).add("a", 1).add("b", 1);
sequences.addNewRelative(Accept).add("a", 1).add("c", 1);
sequences.addNewRelative(Reject).add("a", 1).add("c", 2);
```

4.1. kód. S lefutások megadása Java kódban

A Neo4j ezeket a lefutásokat a 4.5. ábrán látható módon ábrázolja. A kis méretű zöld körök *OrigState* címkéjű csúcsok, közöttük *OrigNext* típusú élek jelzik a sorrendiséget. Ezeknek az éleknek tulajdonságai közé tartozik a bemeneti esemény, vagyis karakter, illetve a két állapot között eltelt relatív idő.

A lefutásokból ezek után létrehozzuk a leendő automata állapotait (F.1.1. kód), hogy majd ezeken tudjuk a különböző műveleteket végrehajtani, az eredeti lefutásokat módosítatlanul hagyva. A leendő automata állapotai között már *NEXT* típusú élek szerepelnek, melyen nem relatív idő, hanem minimum és maximum idővel megadott



4.5. ábra. A Neo4j-ben tárolt S eredeti lefutások ábrázolása

időkorlát szerepel. Ezt az időkorlátot a lefutásokban előforduló legkisebb és legnagyobb relatív idő alapján határozza meg és állítja be a F.1.2. kódrészlet.

Időzített lefutási fa

Az időzített lefutási fa, úgynevezett *tapta* előállítását a 4.2. kód valósítja meg a 7. pszeudokódnak megfelelően. Először kijelöli az összes lefutás első állapotát, majd ezek összevonásából elkészíti a fa gyökerét. Ezt követően az azonos kezdetű lefutások elejét összevonja.

```
driver.run(markMerge1);

boolean doMerge = true;
while (doMerge) {
  driver.run(selectVertexToKeep);
  driver.run(treeMerge);
  doMerge = driver.runWReturn(markMerge2Result.class, iterator -> iterator.next().markCount.longValue
     () != 0);
}

driver.run(removeRegenerateLabels);
driver.run(setRedLabelToRoot);
```

4.2. kód. A *tapta* előállításának algoritmusa Java nyelven

```
// markMerge1
MATCH (s1:State:Regenerate)
WHERE NOT (:State:Regenerate)-[:NEXT]->(s1)
SET s1:Merge
WITH 1 AS dummy MATCH (n) RETURN n
```

4.3. kód. A lefutási fa gyökerének kiválasztása.

A 4.3. kód a lefutások első állapotának lekérdezésére szolgál. Olyan *Regenerate* címkével rendelkező állapotokat keres, melyekbe nem vezet $NEXT$ típusú él. Ezeket az állapotokat megjelöli *Merge* címkével. Ezt követően a 4.4. kód az összevonandó állapotokból kiválaszt egyet, amit megtart, beállítja rajta a *Keep* címkét. A 4.5. kódban ez előbb megjelölt állapoton kívül törli a többi összevonásra megjelöltet és azok éleit és tulajdonságait a *Keep* címkével megjelöltnek örökíti.

```
// selectVertexToKeep
MATCH (v:Merge)
WHERE NOT (v)-[:NEXT]->()
WITH v
LIMIT 1
SET v:Keep
REMOVE v:Merge
WITH count(v) AS cv
// if there was no vertex to keep, we pick a random one
MATCH (r:Merge)
WHERE cv = 0
WITH r
LIMIT 1
SET r:Keep
REMOVE r:Merge
WITH count(*) AS dummy MATCH (n) RETURN n
```

4.4. kód. Összevonandó állapotokból a megmaradó kiválasztása.

```
// treeMerge
MATCH (k:Keep), (m:Merge)
OPTIONAL MATCH (m)-[n1]->(s)
REMOVE k:Keep
SET k += m
WITH k, s, n1, type(n1) as edgeType
WHERE s IS NOT NULL
CALL apoc.create.relationship(k, edgeType, {}, s) yield rel
SET rel = n1
WITH count(*) AS dummy
// delete merge candidates
MATCH (m:Merge)
DETACH DELETE m
// remove keep if not before
WITH count(*) AS dummy
MATCH (k:Keep)
REMOVE k:Keep
// return graph
WITH count(*) AS dummy MATCH (n) RETURN n
```

4.5. kód. Az összevonás.

A F.1.6. *markMerge2* kód a 4.3. kódhoz hasonlóan összevonandó állapotokat jelöl meg, azonban nem azt vizsgálja, hogy az állapot egy lefutás kezdete-e. Az alapján jelöl be állapotokat összevonásra, hogy a karaktersorozat addig az állapotig megegyezik-e valahány lefutásban. Ezután pedig az előzőeknek megfelelően összevonja ezeket az állapotokat.

Az időzített lefutási fa generálásának a feladata az algoritmus során később is előfordul. A *split* műveletnél is újra kell generálni a részfákat a kettévágást követően. A *Regenerate* címke ezért lett bevezetve, hiszen így a *tapta* generálásánál minden állapotra ráfűzve lehet használni, mikor pedig a *split* során csak egyes részfákat kell újragenerálni, ennek segítségével könnyen jelölhető a releváns rész.

Egy fa (vagy akár részfa) generálásának azzal kell végződnie, hogy ezeket a *Regenerate* címkéket eltávolítjuk az állapotokról. Ezt a 4.6. kód végzi.

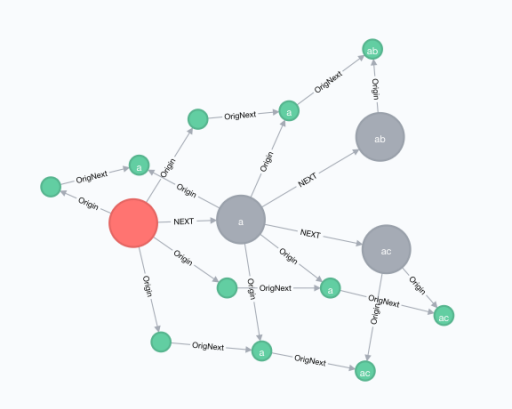
```
// removeRegenerateLabels
MATCH (s:Regenerate)
REMOVE s:Regenerate
```

4.6. kód. Eltávolítjuk a generálás bélyegét.

Utolsó lépésként pedig a Red-Blue algoritmus használata miatt, a fa gyökerét be kell színezni pirosra. Ezt a 4.7. kód végzi el. Ez a kódrészlet megkeresi azt az állapotot, amibe nem megy él, majd a *Red* címke beállításával pirosnak jelöli, az *InitialState* címke beállításával pedig kezdőállapotnak. Végül a 4.6. ábrán látható időzített lefutási fát kapjuk.

```
// setRedLabelToRoot
MATCH (s:State)
WHERE NOT ()-[:NEXT]->(s)
SET s:Red
SET s:InitialState
WITH 1 AS dummy MATCH (n) RETURN n
```

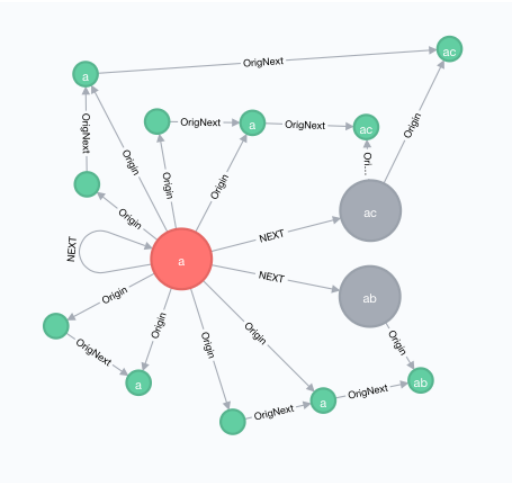
4.7. kód. A fa gyökerének pirosra színezése.



4.6. ábra. A Neo4j-ben tárolt H_0 hipotézismodell, a lefutási fa

4.4.2. Red-Blue eljárás megvalósítása

Az előbbi példa ebben a fejezetben a végrehajtás azon pontján folytatódik, amikor a tanuló algoritmus már egy iteráción túl van, amiben egy *merge* műveletet végrehajtott és a rendelkezésre álló adatmodell a 4.7. ábrának megfelelő.



4.7. ábra. Színezés előtt a hipotézismodell

A Red-Blue eljárás megvalósításához szükséges, hogy minden iterációban megfelelően karbantartsuk a színezést. Ehhez a 4.8. kódrészlet eltávolítja az esetlegesen már nem érvényes kék címkéket, a 4.9. kódrészlet pedig beszínezi a mostani állapotban érvényes kék állapotokat. Kék állapot lesz minden olyan állapot, amely piros állapotnak a gyereke, de ő maga nem piros.

```
MATCH (b:Blue)
REMOVE b:Blue
```

4.8. kód. Eddigi (előző iterációban érvényes) kék színezés eltávolítása.

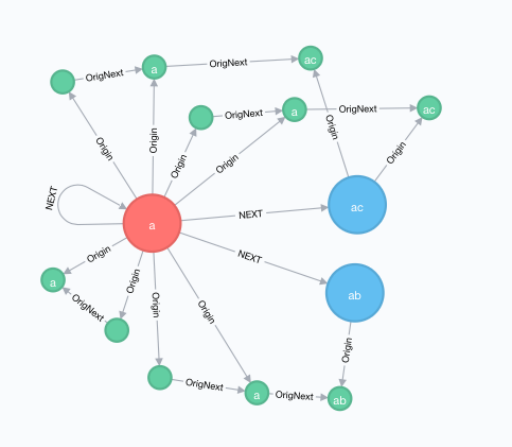
```
MATCH (r:Red)-[:NEXT]->(s)
WHERE NOT s:Red
SET s:Blue
RETURN *
```

4.9. kód. A mostani iterációban érvényes kék színezés.

r	s
{trace: a}	{trace: ac}
{trace: a}	{trace: ab}

4.1. táblázat. Az r piros állapotok és azok s kékre színezendő gyerekei

A 4.9. lekérdezés eredményét a 4.1. táblázat szemlélteti[1]. A példánkban egyetlen piros állapot van, melynek mindkét gyerekét pirosra kell színezni. A színezést követően a 4.8. ábrán látható automatát kapjuk.



4.8. ábra. Színezés után a hipotézismodell

4.4.3. Műveletek megvalósítása

A színezést követően a tanuló algoritmus minden iterációban végrehajt egy műveletet, ameddig el nem készül a végleges automata. Ennek eldöntésére a 4.10. lekérdezést használja, ami eredményeképp visszaadja, hogy van-e még nem piros állapot.

```
// notFinished
MATCH (s:State)
WHERE NOT s:Red
RETURN s
```

4.10. kód. Nem piros állapotok keresése.

Ebben a fejezetben a három művelet metrikájának a kiszámítását, illetve a végrehajtását szeretném szemléltetni.

[1] A Neo4j áz állapotok leírásához nem csak a *trace* tulajdonságot használja (*id*, *acceptance*, stb), a dolgozatban az egyszerűség és érthetőség kedvéért szerepel így.

Split művelet metrikájának kiszámítása

A *split* művelet végrehajtásához először le kell kérdezni, hogy melyik piros állapotból melyik kék állapotba menő állapotátmeneten, milyen időkorláttal kell azt végrehajtani. Ezt a lekérdezést a split metrikaszámító lekérdezése hajtja végre, amely emellett azt is eredményül adja, hogy a szekvenciákban szereplő állapotok közül melyikből melyik részfát kell generálni.

A split metrikájának kiszámításához szükséges teljes lekérdezést a függelékben lévő F.1.12. kód szemlélteti.

Ez a lekérdezés a valósidejű automatában a legmagasabb értékű *split* művelet paramétereit adja vissza. Ezáltal a későbbiekben, ha ehhez a művelethez tartozik a legmagasabb értékű metrika, akkor felhasználva a paramétereket végre lehet hajtani a *split* műveletet.

Az alábbiakban kifejtem a lekérdezés egyes részeit.

A lekérdezés felső három sora (4.11. kódrészlet) arra szolgál, hogy kiválogassuk a lehetséges d állapotátmeneteket a $split(d, t)$ művelethez.

```
MATCH (r:Red)-[n:NEXT]->(b:Blue)
WITH r, b, n.Tmax as Tmax, n.Tmin as Tmin, n.symbol as symbol
UNWIND range(Tmin, Tmax-1) AS t
```

4.11. kód. d állapotátmenet és t idő kiválasztása.

Ezek az állapotátmenetek egy *red* piros állapotból egy *blue* kék állapotba kell, hogy mutassanak, ennek lekérdezésére szolgál az 1. sor. A továbbiakban szükségünk lesz mindkét állapotra, illetve a közöttük lévő állapotátmenet minden tulajdonságára: a *symbol* karakterre, illetve az időkorlátra (2. sor).

Ha a lekérdezésünk csupán ez a két sor lenne és az állapotokkal, valamint az állapotátmenet tulajdonságaival rögtön visszatérne, akkor a 4.2. táblázathoz hasonló eredményt kapnánk (aktuális automatától függően).

red	blue	Tmax	Tmin	symbol
{trace: a}	{trace: ab}	1	2	b
{trace: a}	{trace: ac}	1	2	c

4.2. táblázat. A maxSplitScore részlekérdezésének eredményeként kapott reláció

A lekérdezés azonban folytatódik, végül meg kell vizsgálni a $split(d, t)$ művelet értékét adott d állapotátmenet esetén minden lehetséges t-re. Az alsó és a felső időkorlát közé eső t idők lehetségesek *split* szempontjából, ahogy az a 3. sorban látható.

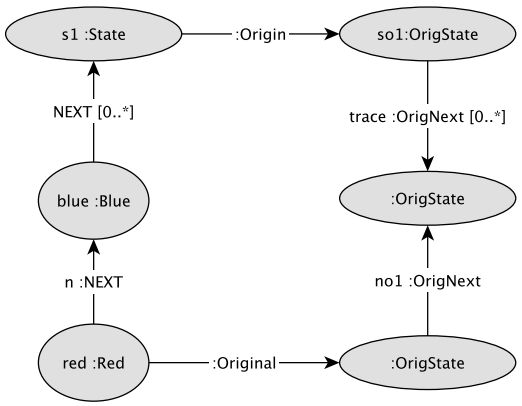
```
MATCH (b)-[:NEXT*0..]->(s1:State)-[:Origin]->(so1:OrigState), trace1=(so1)<-[OrigNext*0..]-(
    redOrigNext:OrigState), (redOrigNext)<-[no1:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace1) WHERE (x)<-[:Origin]-(r))
WITH r, b, t, Tmin, Tmax, symbol, s1, no1, so1
MATCH (b)-[:NEXT*0..]->(s2:State)-[:Origin]->(so2:OrigState), trace2=(so2)<-[OrigNext*0..]-(
    redOrigNext:OrigState), (redOrigNext)<-[no2:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace2) WHERE (x)<-[:Origin]-(r))
```

4.12. kód. Az RTA egy részének az eredeti S lefutásainak megkeresése.

A lekérdezés következő pár sora (4.12. kód) arra szolgál, hogy a kettévágott élet követő részfának az állapotaihoz megkeressük az eredeti S lefutásokban az ősét. Vagyis a lefutások azon részeit, amiből az adott részfa generálódott. Fontos megjegyezni itt, hogy egy-egy állapot a részfában több különböző lefutásnak is lehetett az állapota, melyek az APTA generálása során összevonódtak. A *split* műveletnek éppen az a lényege, hogy az ilyen állapotokat két külön kategóriára bontsa. Ezek közül az egyik kategória a *split*-et követően az egyik, míg a másik kategória a másik részfába fog kerülni.

A 4.9. ábra szemlélteti a gráfmintának ezt a két sorát. A *red* piros állapotból a *blue* kék állapotba mutató kettévágandó n állapotátmenet alapján keressük az illeszkedéseket olyan $s1$ állapotokra, melyek *blue* állapotból induló részfának az állapotai (*blue*-t is beleértve). Ezen $s1$ állapotok valamelyik (esetleg több) lefutás egy $so1$ állapotából generálódtak. Ezekbe az $so1$ állapotokba pedig az eredeti lefutásokban vezetett út a *red* piros állapot őséből, amely útnak az első állapotátmenete $no1$ volt. Vagyis az $so1$ állapotot tartalmazó lefutás egy korábbi állapotából *red* generálódott.

Az illeszkedés sora a 4.12. kódban azért van megduplázva, hogy ha egy $s1$ állapothoz tartozóan több $so1$ is tartozna (vagyis $s1 = s2$ de $so1 \neq so2$), akkor azoknak a párba állításával ki lehet számítani a metrikát.



4.9. ábra. Eredeti S lefutások megkeresése

A 4.13. kódban lévő sorok felelősek azért, hogy az állapotok ősét ($so1$ és $so2$) kategóriákba osszuk. A kategóriák lényege az, hogy a *split* során melyik keletkező részfa generálásában fog részt venni az adott állapot. A kategóriákat pedig az alapján számítjuk, hogy a korábban lekérdezett $no1$ és $no2$ állapotátmeneten szereplő *time* idő nagyobb-e a *split* művelet t időparaméterénél. Ez a *time* idő az előző állapotba lépéstől számított eltelt időt jelenti. A metrika kiszámításához olyan $no1$ és $no2$ párokra van szükségünk, amelyekből ugyanaz az állapot generálódott, vagyis $s1 = s2$. Ezen kívül, $no1$ és $no2$ állapotoknak külön részfába kell kerülniük ahhoz, hogy a *split* metrikáját módosítsák, tehát más kategóriába kell tartozzanak. Az ezt követő sorban $no1$ és $no2$ egyedi azonosítóját (id) csak azért hasonlítjuk össze, hogy egy állapotpárt csak egyszer adjon eredményül a lekérdezés.

```
WITH t, r, b, s1, s2, toInteger(no1.time)>t as cat1, toInteger(no2.time)>t as cat2, Tmin, Tmax,
     symbol, so1, so2
WHERE s1 = s2
    AND cat1 <> cat2
    AND id(so1) < id(so2)
```

4.13. kód. Az eredeti lefutások t szerint kategóriákba rendezése.

A lekérdezés következő részlete (4.14. kód) végzi az állapotpárokra a metrika módosításának a kiszámítását. A 3.2.6. fejezetben leírt elvek alapján.

```
WITH r, b, t, Tmin, Tmax, symbol,
    [coalesce(so1.accepting, false), coalesce(so1.rejecting, false)] AS so1ar,
    [coalesce(so2.accepting, false), coalesce(so2.rejecting, false)] AS so2ar
WITH
    r, b, t, Tmin, Tmax, symbol,
    CASE
        WHEN so1ar[0] <> so2ar[0] AND so1ar[1] <> so2ar[1] THEN 1
        WHEN so1ar =  so2ar AND (so1ar = [false, true] OR so1ar = [true, false]) THEN -1 // there is
     at least one true
        ELSE 0
    END AS score
```

4.14. kód. A *split* során szétszedett q és q' állapotokhoz tartozó metrika változtatásának a kiszámítása.

Végül pedig a lekérdezés utolsó pár sora rendezi a lehetséges *split* műveleteket a metrikájuk alapján. Rendezés után az egyetlen legmagasabb értékű *split* művelet paramétereivel visszatér. Így a későbbiekben a paraméterek felhasználásával meg tud történni a *split* végrehajtása.

```
WITH r, b, t, sum(score) AS metric, Tmin, Tmax, symbol
ORDER BY metric DESC
LIMIT 1
WITH r, b, t, metric, Tmin, Tmax, symbol
MATCH (b)-[:NEXT*0..]->(s1:State)-[:Origin]->(so1:OrigState), trace1=(so1)<-[OrigNext*0..]-(
     redOrigNext:OrigState), (redOrigNext)<-[no1:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace1) WHERE (x)<-[:Origin]-(r)) AND toInteger(no1.time)>t
WITH r, b, t, metric, Tmin, Tmax, symbol, collect(so1) as so1s
MATCH (b)-[:NEXT*0..]->(s2:State)-[:Origin]->(so2:OrigState), trace2=(so2)<-[OrigNext*0..]-(
     redOrigNext:OrigState), (redOrigNext)<-[no2:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace2) WHERE (x)<-[:Origin]-(r)) AND  toInteger(no2.time)<=t
RETURN r, b, t, metric as m, Tmin, Tmax, symbol as sym, so1s, collect(so2) as so2s
```

4.15. kód. m metrikák kiszámítása és a legnagyobb kiválasztása.

A példában ennek a lekérdezésnek az eredménye a 4.3. táblázatban látható[2].

r	b	t	m	Tmin	Tmax	sym	so1s	so2s
{trace: a}	{trace: ac}	1	1	1	2	c	{trace: ac, id: n8}	{trace: ac, id: n11}

4.3. táblázat. A maxSplitScore lekérdezésének eredményeként kapott reláció

Split művelet végrehajtása

A *split* művelet végrehajtásához három lépésre van szükség. Egy módosító lekérdezéssel törölni kell a kék állapotból induló részfát, egy másikkal legenerálni az új részfákat, egy utolsóval pedig állapotátmeneteket generálni a piros állapotból, megfelelő időkorlátokkal, a részfákba. A kék állapotból induló részfa törlését a 4.16. kód valósítja meg.

```
// deleteBlueSubtree
WITH $b as b
MATCH (b)-[:NEXT*0..]->(s:State)
DETACH DELETE s
```

4.16. kód. A kék állapotból induló részfa törlése.

Ezután a *tapta* generálásával megegyező módon az algoritmus először megjelöli a részfák újragenerálásához szükséges állapotokat *Regenerate* címkével. Ezeket az

[2]Az so1s és so2s oszlopnál most az *id* tulajdonsága is megjelenik az eredeti állapotoknak, hiszen *trace* tulajdonságuk alapján nem lennének megkülönböztethetők.

állapotokat a *split* metrikaszámító lekérdezése adja meg $so1s$ és $so2s$ értékekként. Ezt követően az algoritmus csak ezen állapotokból időzített lefutási fákat gyárt.

Utolsó lépésként a részfákat össze kell kötni az automata többi részével, ügyelve arra, hogy az új élek jó időkorlátokat kapjanak. Ezt a műveletet a F.1.14. kód valósítja meg.

Merge művelet metrikájának kiszámítása

A legnagyobb metrikájú *merge* művelet a kiválasztásához először sorban meg kell vizsgálni, hogy mely piros és kék állapotpárok alkalmasak összevonásra. Erre szolgál a 4.17. kódrészlet. Ebben a kódban a program páronként vizsgálja a piros és kék állapotokat, ha egy állapotpárt már megvizsgált, akkor közöttük egy *Checked* típusú élt hoz létre.

```
// mergeCandidates
MATCH (r:Red), (b:Blue)
WHERE NOT (r)-[:Checked]-(b)
WITH r, b
LIMIT 1
CREATE (r)-[:Checked]->(b)
RETURN r, b
```

4.17. kód. Még nem vizsgált piros és kék állapotpár keresése *merge* művelethez.

Az időzítés bevezetésével lehetségessé vált az, hogy két összevonandó állapotból ugyan olyan karakterrel, de más időkorláttal mennek ki élek. Ilyenkor még a *merge* végrehajtása, vagy annak metrikájának a kiszámítása előtt végre kell hajtani egy *split* műveletet. A F.1.16. lekérdezés a szükséges *split* műveletek paramétereivel tér vissza.

Miután már minden szükséges *split* műveletet elvégzett az algoritmus, bejelöli, hogy mely állapotokat vonná össze (F.1.17. kód). A F.1.18. kód azért felelős, hogy azon állapotpárok is be legyenek jelölve összevonásra, melyek, az eredeti jelölés szerinti, több összevonás végeredményeképp kerülnének csak összevonásra. Ez a metrika számítás szempontjából fontos, illetve segít eldönteni, hogy egy adott *merge* művelet egyáltalán lehetséges-e (nem okoz piros állapotban inkonzisztenciát).

Miután az összes összevonandó pár meg van jelölve, a F.1.19. kód kiszámolja a *merge* művelethez tartozó metrikát. Végül az eddig létrehozott összevonás-jelöléseket eltávolítjuk az automatáról. Erre szolgál a F.1.20. kód.

A példánkban a *merge* egy helyen lenne lehetséges (4.4. táblázat), hiszen a *trace: ac* tulajdonsággal rendelkező állapotot összevonva a piros állapottal egy inkonzisztens piros állapotot kapnánk.

r	b	dpairs	metric
{trace: a}	{trace: ab}	{trace: a}, {trace: ab}	1

4.4. táblázat. A maxMergeScore lekérdezésének eredményeként kapott reláció

Merge művelet végrehajtása

A *merge* művelet végrehajtása az összes szükséges *split* végrehajtásáig ugyan az, mint a metrikájának a kiszámítása esetében. Utána a metrika kiszámítás során elmentett összevonandó állapotpárokat újból megjelöli összevonása (F.1.21. kód).

Ezt követően a 4.18. java kódnak megfelelően további műveleteket hajt végre.

```
while (driver.run(createSingleNewState)) {
  driver.run(copyEdgesToNewState);
  driver.run(deleteMergedStates_RemoveNewStateLabel);   driver.run(deleteParallelEdges);
}

driver.run(removeIndexedMergeLabel);
driver.run(joinEdgeTimeguards);
```

4.18. kód. A *merge* művelet végrehajtásának algoritmusa

Először, amíg van, minden összevonandó állapotpár helyett egy új állapotot hoz létre (F.1.22. kód). Majd ebbe az állapotba átmásolja az összevonandó állapotok tulajdonságait és éleit (F.1.23. kód). Ezek után törli a már összevont állapotokat (F.1.24. kód). Végül pedig, ha esetleg kialakultak volna párhuzamos élek, azokat megszünteti (F.1.25. kód).

Miután összevont minden összevonandó állapotpárt, eltávolítja az összevonásra létrehozott jelzőket az automatáról. Majd ha esetleg kialakultak volna átlapolódó állapotátmenetek (melyek azonos állapotból, azonos állapotba mutatnak, azonos karakterrel és átlapolódó időkorláttal), akkor azokat összevonja (F.1.27. kód).

Color művelet metrikájának kiszámítása

A *color* művelet megvalósítása és metrikaszámítása is egyszerű. A 4.19. lekérdezéssel, ha van, megkapunk egy olyan kék állapotot, amelyet pirosra lehet színezni. A kód első sorában fogalmazza meg azt a megkötést, hogy az állapot rendelkezzen *Blue* címkével, vagyis kék legyen, a második sorában pedig az inkonzisztenciát vizsgálja. Az ilyen állapotok közül kiválaszt egyet és 0 metrikával visszaadja. A 4.5. táblázat ennek a lekérdezésnek a visszatérési értékét mutatja a példánkban. (Az *trace: ac* tulajdonsággal rendelkező állapotot nem engedné az algoritmus átszínezni, mert így egy inkonzisztens piros állapotot kapnánk.)

```
// maxColorScore
MATCH (b:Blue)
WHERE NOT (coalesce(b.accepting, false) AND coalesce(b.rejecting, false))
WITH b
ORDER BY b.trace DESC
LIMIT 1
WITH b, 0 as metric
RETURN b, metric
```

4.19. kód. A color lekérdezés.

b	metric
{trace: ab}	0

4.5. táblázat. A maxColorScore lekérdezésének eredményeként kapott reláció

Color művelet végrehajtása

A *color* végrehajtása egyszerűen a 4.20. kódnak megfelelően a paraméterként kapott kék állapot átszínezése.

```
// color
WITH $b as b
SET b:Red
```

4.20. kód. Egy megfelelő kék állapot átszínezése pirosra.

4.4.4. EDSM megvalósítása

Az EDSM algoritmus lényege, hogy kiszámolja minden művelet metrikáját, majd a legjobbnak bizonyulót hajtja végre. A megvalósítás során először a legjobb *merge* műveletet és a hozzá tartozó metrikát számoljuk ki, majd a *split* műveletek közül a legjobbét, végül a *color* műveletét. A metrikák kiszámításának a vezérlését a 4.21. kód végzi.

```
while (driver.run(notFinished)) {
  driver.run(removeAndSetBlueLabel);

  // ...

  // if merge scores the highest
  calculateMaxMergeMetric(true);

  // if split scores the highest
  if (getMaxScore(maxSplitScoreResult.class, null))
    operation = Operation.split;

  // if color scores the highest
  if (getMaxScore(maxColorScoreResult.class, null))
    operation = Operation.color;

  // ...
```

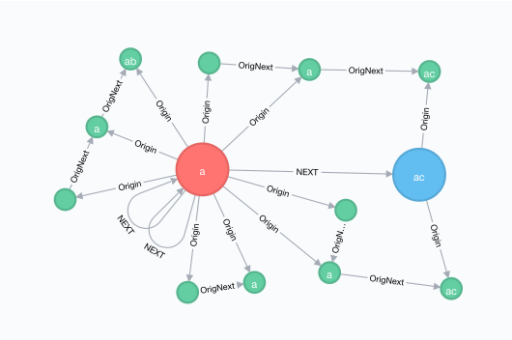
4.21. kód. Legjobb metrikájú művelet kiválasztása

Miután a tanulóalgoritmus kiszámolta, hogy melyik a legjobb művelet, a 4.22. java kódban végrehajtja azt.

```
  switch (operation) {
    case merge:
      merge();
      break;
    case split:
      split(map.get());
      break;
    default:
      color();
      break;
  }
```

4.22. kód. Legjobb metrikájú művelet végrehajtása

A példában egy *merge* és egy *split* műveletnek volt 1 metrikája, egy *colornak* pedig 0. Mivel az algoritmus a *merge* metrikákat hamarabb számolja ki, mint a *split* műveletekét, így a végrehajtandó műveletnek a *merge* lesz beállítva (*split* művelet metrikája nem nagyobb, így nem írja felül). Emiatt a példánkban a *a* és *ab trace* tulajdonságú állapotokat fogja összevonni az algoritmus ebben az iterációban. Az összevonás után a 4.10. ábrán látható automatát kapjuk.

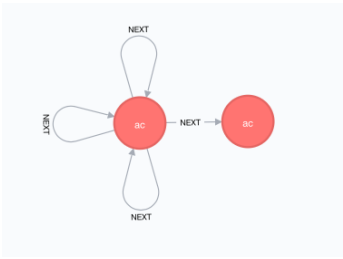


4.10. ábra. Színezés előtt a hipotézismodell

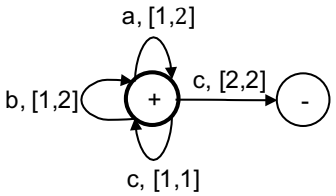
4.4.5. Végeredmény

Az algoritmust végig futtatva, a példa lefutásokból a 4.11a. ábrán látható automata fog létrejönni. Ez az automata két állapotból áll, egy elfogadóból és egy nem elfogadóból. Nem elfogadó állapotba "c" karakternek megfelelő absztrakt bemeneti eseményre kerül, amennyiben az 2 időegység elteltével következik be.

A Neo4j az 4.11a. ábrának megfelelő módon ábrázolja az automatát – sajnos a megjelenítés nem ábrázol minden szükséges információt. Azonban az automata egyes komponenseire kattintva ki lehet íratni azok címkéit, tulajdonságait. Ezek alapján a megtanult automatáról a 4.11b. ábrát készítettem.



(a) Neo4j-ben ábrázolva



(b) Paraméterekkel ábrázolva

4.11. ábra. A megtanult automata

A 4.11b. ábrán látható, hogy a két állapotot valóban az időbélyeg fogja megkülönböztetni. Példa lefutásaink közül kettő "a" karaktert követő "c" karakter volt, melyek közül elfogadó volt, ha 1 időegység telt el köztük, nem elfogadó volt, mikor 2. Az így megtanult automata alapján lehet arra következtetni, hogy a leengedésnek az ideje rosszul volt specifikálva – a valós rendszer esetén a leengedés tovább tart, mint az a specifikáció alapján várható, emiatt sérülhetett meg a rakomány (például, ha vonat hamarabb elindult, a rakomány érkezésekor már mozgásban volt).

5. fejezet

Fejlesztések

Ebben a fejezetben egy esettanulmányon keresztül bemutatom az eredeti algoritmus hatékonyságára, skálázhatóságára vonatkozó méréseket, az algoritmus lassú komponenseinek optimalizációjára tett javaslataimat, a főbb javasolt megoldások implementációs részleteit, valamint hatékonyságuk kiértékelését.

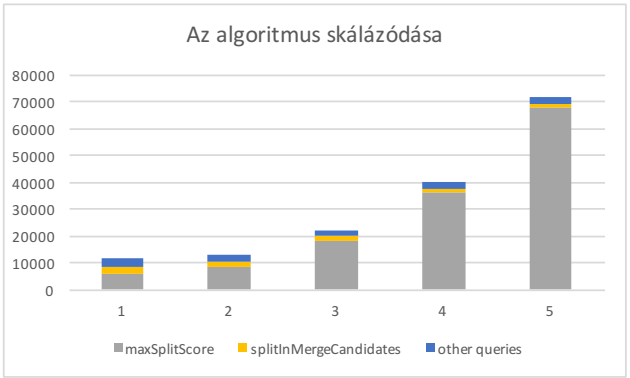
5.1. Probléma azonosítása

Az algoritmus hatékonyságának növeléséhez először meg kell határozni, hogy melyek azok a komponensek, amelyek legjobban befolyásolják az algoritmus futásidejét. Az automatatanuló algoritmus érzékenységvizsgálatára kétféle megközelítéssel került sor: először az algoritmus és egyes komponenseinek futásidejét lemérve, gyakorlati szempontból vizsgáltam az egyes műveletek hatékonyságát, majd algoritmus működését alapul véve, elméleti megfontolások alapján további tényezőket is kijelöltem.

5.1.1. Profiling

A algoritmus skálázódására méréseket végeztem, amihez a bemenetet 4.4. fejezetben leírt lefutások szolgáltatták. Lemértem, hogy mennyi ideig tart az algoritmusnak megtanulnia az automatát, abban az esetben, ha a lefutásokat n-szer kapja meg a tanulás bemeneteként, ahol $n = 1, 2, 3, 4, 5$. Minden egyes n-re három mérést végeztem és ezek mediánjával dolgoztam tovább.

Ezen kívül az algoritmus futása során az egyes gráflekérdezések összidejét is megmértem, hogy kiderüljön melyik növeli legjobban a futásidőt. A mérési eredmények a 5.1. ábrán láthatók. Az x tengelyen az n paraméter, míg az y tengelyen a lefutási idő van ábrázolva ms-ban.



5.1. ábra. Mérési eredmények

A mérések alapján, amint az a diagramon látszik, a maximális metrikájú *split* megkeresése veszi el a legtöbb időt. Emellett számottevő az a lekérdezés, amely a *merge* során szükséges *split* műveletek paramétereit adja vissza.

5.1.2. Elméleti megfontolás

Elméleti szempontból vizsgálva az algoritmust meg lehet fogalmazni néhány további okot, ami miatt az algoritmus nem elég hatékony.

- Az algoritmus a *merge*, *split* és *color* műveleteket ebben a prioritási sorrendben hajtja végre. Ez azt jelenti, hogy ha azonos a legjobb *merge* metrikája a legjobb *split* metrikájával, akkor a *merge* fog végrehajtódni. Ugyan ez igaz a *merge* és *color*, illetve a *split* és *color* párokra is. Azonban mind a *merge*, mind a *split* igen bonyolult műveletek, így valószínűleg időigényesebbek is, mint az egyszerűbb *color*. Az algoritmus ennek ellenére csak akkor fog *color* műveletet végrehajtani, ha a többi nem lehetséges.
- Az időzített lefutási fa előállítása során az algoritmus megengedi inkonzisztens állapotok az összevonását. Ezen állapotok szétbontásához a későbbiekben szükség van egy *split* műveletre, ami nem hatékony.
- A *split* metrikaszámító művelete az összes lehetséges időkorlátot megvizsgálja a szétbontáshoz, amely jelentős mennyiségű számítást von maga után. Előnyösebb volna csupán a lényegesen különböző korlátokat vizsgálni.

5.2. Továbbfejlesztési lehetőségek

A fentebb felvázolt problémákra a következő fejlesztéseket javasoltam.

5.2.1. Különböző időkorlátok különböző karakterekre

Első javasolt megoldás, hogy a minimum és maximum időkorlátok számontartása az automatában minden egyes karakterre külön történjen. Így *split* művelet, vagy annak metrikaszámítása során csak kevesebb lehetséges időkorlátot kell vizsgálnia az algoritmusnak. Ezáltal gyorsabb lenne a legtovább tartó, *split* metrikaszámító művelet is.

Ezt a módosítást még az időzített lefutási fa létrehozása előtt lehet megtenni, az algoritmus azon részében, ahol az eredeti lefutásokból az automatatanulás során módosítható lefutásokat készíti. Azaz ahol az *OrigState* címkéjű állapotoknak és *OrigNext* típusú éleknek megfelelő *State* címkéjű állapotokat és *NEXT* típusú éleket létrehozza az algoritmus.

5.2.2. Prioritási sorrend megváltoztatása

Egy másik javaslat a prioritási sorrend megváltoztatása. Ennek eredményeként a *color* művelet előrehozásával az algoritmus nem hajt végre 0 metrikájú *merge*, illetve *split* műveleteket, melyek időigényesek lennének.

Ehhez a változtatáshoz a Java kódon kell módosítani, az algoritmusban felcserélve az egyes műveletek metrikájának a kiszámítási a sorrendjét.

5.2.3. Szigorúbb időzített lefutási fa előállító algoritmus

Az időzített lefutási fa előállítása során megtiltva inkonzisztens állapotok összevonását, az automatatanuló algoritmus későbbi lépéseit lehet gyorsítani. Ezzel a javítással elérhető, hogy kevesebb *split* műveletet kelljen végrehajtani.

Megvalósítás szempontjából ez a javítás, az előzőekhez képest, nagyobb munkát venne igénybe. Először a *tapta* előállítás során kellene megfogalmazni egy feltételt arra, hogy elfogadó állapotot ne vonjon össze nem elfogadóval. Ennek hatására azonban a *tapta* nemdeterminisztikussá válna. Előfordulhatna olyan helyzet, hogy egy s_0 állapotból megy egy-egy él a karakterrel, $Tmin$ és $Tmax$ időkorlátokkal egy s_+ elfogadó és egy s_- nem elfogadó állapotba is. A nemdeterminizmus feloldására még az automatatanulás ezen fázisában módosítani kellene az időkorlátokat az eredeti lefutások segítségével.

5.2.4. Okosított vágási idő keresés

A legidőigényesebb művelet, a *split* metrikaszámítása során az algoritmus megvizsgálja az összes lehetséges időpontot, amely szerint fel lehet bontani az adott állapotátmenetet. A vizsgált időpontok számának csökkentésével gyorsulhat az algoritmus. Ilyen javítást kétféleképpen is meg lehet valósítani.

- A Java kódban bináris keresés megvalósításával. Ehhez a Cypher lekérdezést is módosítani kellen olyanra, hogy egy megadott t időparaméterrel dolgozzon.
- A Cypher lekérdezésben is meg lehetne fogalmazni olyan megkötéseket, hogy csak releváns időket próbáljon ki. Relevánsnak számít az az idő, amely azon eredeti szekvenciákban szerepel, amikből az automata szétvágandó éle keletkezett.

5.2.5. Javítások kombinációja

Természetesen a fent megfogalmazott javításokat egymással kombinálva is lehet alkalmazni, illetve vizsgálni hatékonyságukat.

5.3. Javítások kiértékelése

A fenti javaslatok közül kettőt implementáltam.

5.3.1. Különböző időkorlátok különböző karakterekre

Ehhez a javításhoz az algoritmus *initIntervals* lekérdezését kellett módosítani a 5.1. kódnak megfelelően. A javított verzióban a lekérdezés második sora a maximum és minimum időket karakter szerint gyűjti ki, a negyedik sora a megfelelő karakterre vonatkozó időkorlátokat állítja be.

```
// initIntervals
MATCH ()-[on:OrigNext]->()
WITH max(on.time) as maxTime, min(on.time) as minTime, on.symbol as symbol
MATCH (:State:Regenerate)-[n:NEXT]->(:State:Regenerate)
WHERE n.symbol = symbol
SET n.Tmin = minTime
SET n.Tmax = maxTime
```

5.1. kód. Karakterfüggő időkorlátok beállítása.

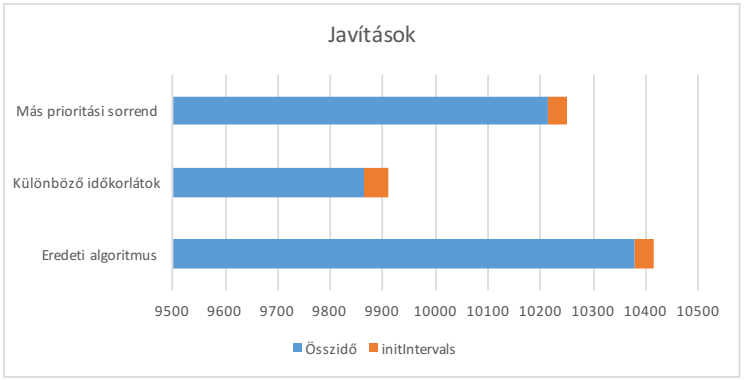
5.3.2. Prioritási sorrend megváltoztatás

A prioritás megváltoztatásához a Java forráskódon kellett módosítani (5.2. kód). A ciklusban a *color* metrikájának a kiszámítása pár sorral feljebb került, így az *operation* változó csak akkor kerül felülírásra a *merge* vagy a *split* metrikaszámítása során, ha azok metrikája szigorúan nagyobb.

```
while (driver.run(notFinished)) {
driver.run(removeAndSetBlueLabel);

// ...

// if color scores the highest
if (getMaxScore(maxColorScoreResult.class, null))
operation = Operation.color;

// if merge scores the highest
calculateMaxMergeMetric(true);

// if split scores the highest
if (getMaxScore(maxSplitScoreResult.class, null))
operation = Operation.split;

// ...
```

5.2. kód. Legjobb metrikájú művelet kiválasztásának megváltoztatása

Ezekkel a módosításokkal a 5.2. diagramon látható eredményeket értem el, ahol az x tengelyen a futásidő látható *ms*-ban. Külön ábrázoltam az időkorlátok beállításának idejét. Látható, hogy más prioritási sorrend beállításánál ez az idő nem változott jelentősen, a teljes lefutási idő viszont csökkent. Karakterekre különböző időkorlátok beállítása esetén ez a lekérdezés valamennyivel több ideig futott, azonban az egész tanulóalgoritmus lefutási ideje jelentősen csökkent.

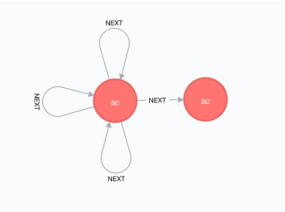


5.2. ábra. Javítások mérési eredményei

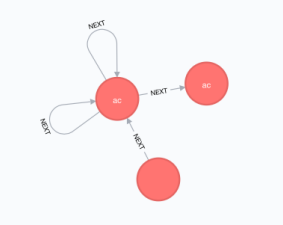
A lefutási idők mérése mellett megvizsgáltam a tanuló algoritmus által adott automatákat is a javítások során (5.3. ábra). Különböző időkorlátok esetén ugyanazt az automatát kaptam, mint a változtatások nélküli algoritmust futtatva. Más prioritási sorrend beállítása esetén azonban egy nagyobb automata lett az eredmény, mely a tanulás bemenete (a lefutások) alapján nem megkülönböztethető.

5.4. Konklúzió

Méréseimből az a következtetés vonható le, hogy a különböző időkorlátok beállítása hatékonyabb algoritmust eredményez. A prioritás megváltoztatása esetén azonban



(a) Alapesetben és különböző időkorlátok esetén



(b) Más prioritási sorrend beállítása esetén

5.3. ábra. Megtanult automaták

mérlegelni kell, hogy minimális automata megtanulása-e a cél, vagy egy rövidebb futásidő. A *color* művelet éppen azért került az utolsó helyre, mert az automata minimalizálása szempontjából ez a legkevésbé hasznos művelet (végrehajtása biztosan megnöveli a végső automata állapotterét).

6. fejezet

Összefoglalás

Ebben a fejezetben összefoglalom a dolgozat fő kontribúcióit, és bemutatom jövőbeli céljaim.

6.1. Kontribúció

Munkám célja egy olyan módszer megalkotása volt, mellyel kritikus, időzített kiberfizikai rendszerek viselkedésmodelljét tudom automatikusan előállítani tanuló algoritmus segítségével.

Dolgozatomban bemutattam számos tudományterület, többek között az automataelmélet, a valósidejű automaták, az automatanulás, és a gráflekérdezések alapjait, emellett kifejtettem egy automatanuló algoritmus működését mind időzítettlen, mind pedig időzítésekkel ellátott esetben, valamint bemutattam a munkám során elért eredményeket.

Munkám során elért legjelentősebb kutatási eredmények a következő pontokba foglalhatók össze.

- Egy létező időzített automata tanuló algoritmust leimplementáltam.
  - Az implementáció egy gráfadatbázisra épülő szoftver, az egyes műveleteket gráfminták segítségével valósítottam meg.
  - Az algoritmus logikáját Java kódban valósítottam meg.
- Az elkészült szoftvert modulként egy gráfminta-alapú tanulókeretrendszerbe illesztettem.
- Egy esttanulmányon demonstráltam a keretrendszer és a szoftver implementációját, eredményességét.
- Az algoritmus hatékonyságát megvizsgáltam, azonosítottam az algoritmus azon pontjait, ahol a hatékonyság növelhető.
  - Elméleti szempontból, az algoritmus működésének ismeretében meghatároztam azon részeit, amelyek hatékonyabban is megoldhatók lennének.
  - Gyakorlati szempontból, mérésekkel meghatároztam azon műveleteket, amelyek egy átlagos lefutás idejének jelentős hányadát adják.
- A hatékonyság növelésére javaslatokat tettem, amelyek az azonosított problémákat megoldják.

- A felsorolt javaslatok közül a legfontosabbakat leimplementáltam, eredményességüket mérésekkel ellenőriztem, hatásukat elméleti szempontból megvizsgáltam.

6.2. Jövőbeli munka

Kutatásom fő célja az algoritmus hatékonyságának növelése, hogy iparilag releváns esettanulmányokon is alkalmazni lehessen. Ehhez a jövőben tervezem a bemutatott fejlesztési lehetőségek implementálását (szigorított *tapta* előállítás, hatékonyabb *split* metrikaszámítás), hatékonyságuk elemzését, valamint további fejlesztési lehetőségek kutatását.

A jövőben szeretném megvizsgálni a tanuló algoritmus alkalmazhatóságát, korlátait. Ehhez az absztrakciós keretrendszerben alkalmazva tervezzük komplex, iparilag releváns esettanulmányok felhasználását.

Köszönetnyilvánítás

Szeretnék köszönetet mondani Elekes Márton Farkasnak, Farkas Rebekának, Semeráth Oszkárnak, Szárnyas Gábornak, Tóth Tamásnak és Vörös Andrásnak hogy segítettek a munkámban.

Az Emberi Erőforrások Minisztériuma ÚNKP-17-1-I. kódszámú Új Nemzeti Kiválóság Programjának támogatásával készült.

Irodalomjegyzék

[1] Fides Aarts – Faranak Heidarian – Harco Kuppens – Petur Olsen – Frits Vaandrager: Automata learning through counterexample guided abstraction refinement. In *International Symposium on Formal Methods* (konferenciaanyag). 2012, Springer, 10–27. p.

[2] Rajeev Alur – David L Dill: A theory of timed automata. *Theoretical computer science*, 126. évf. (1994) 2. sz., 183–235. p.

[3] Dana Angluin: Learning regular sets from queries and counterexamples. *Information and computation*, 75. évf. (1987) 2. sz., 87–106. p.

[4] Budapesti Műszaki és Gazdaságtudományi Egyetem, Számítástudományi és Információelméleti Tanszék: *Nyelvek és automaták*. `http://www.cs.bme.hu/~friedl/nya/jegyzet-13.pdf`.

[5] Miguel Bugalho – Arlindo L Oliveira: Inference of regular languages using state merging algorithms with search. *Pattern Recognition*, 38. évf. (2005) 9. sz., 1457–1467. p.

[6] Catalin Dima: Real-time automata. *Journal of Automata, Languages and Combinatorics*, 6. évf. (2001) 1. sz., 3–24. p.

[7] Olga Grinchtein – Bengt Jonsson – Martin Leucker: Learning of event-recording automata. *Theor. Comput. Sci.*, 411. évf. (2010) 47. sz., 4029–4054. p.
URL `https://doi.org/10.1016/j.tcs.2010.07.008`.

[8] John E Hopcroft – Rajeev Motwani – Jeffrey D Ullman: Automata theory, languages, and computation. *International Edition*, 24. évf. (2006).

[9] KJ Lang – BA Pearlmutter – RA Price: Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. lnai, 1433: 1–12, 1998. In *4th International Colloquium, ICGI-98* (konferenciaanyag).

[10] Alexander Maier: Online passive learning of timed automata for cyber-physical production systems. In *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on* (konferenciaanyag). 2014, IEEE, 60–66. p.

[11] Alexander Maier – Oliver Niggemann – Roman Just – Michael Jäger – Asmir Vodencarevic: Anomaly detection in production plants using timed automata - automated learning of models from observations. In Jean-Louis Ferrier – Alain Bernard – Oleg Yu. Gusikhin – Kurosh Madani (szerk.): *ICINCO 2011 - Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics, Volume 1, Noordwijkerhout, The Netherlands, 28 - 31 July, 2011* (konferenciaanyag). 2011, SciTePress, 363–369. p. ISBN 978-989-8425-74-4.

[12] József Marton – Gábor Szárnyas – Dániel Varró: Formalising openCypher graph queries in relational algebra. In *ADBIS* (konferenciaanyag). 2017, 182–196. p. URL https://doi.org/10.1007/978-3-319-66917-5_13.

[13] Neo Technology: Cypher query language. https://neo4j.com/docs/developer-manual/current/cypher/, 2016.

[14] Neo Technology: Neo4j. http://neo4j.org/, 2016.

[15] Harald Raffelt – Bernhard Steffen – Therese Berg: Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems* (konferenciaanyag). 2005, ACM, 62–71. p.

[16] SE Verwer – MM De Weerdt – Cees Witteveen: An algorithm for learning real-time automata. In *Benelearn 2007: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands, Amsterdam, The Netherlands, 14-15 May 2007* (konferenciaanyag). 2007.

[17] Sicco Ewout Verwer: *Efficient identification of timed automata: Theory and practice.* PhD értekezés (TU Delft, Delft University of Technology). 2010.

Függelék

F.1. Cypher lekérdezések

```
// initStates
// from OrigStates
MATCH (root:OrigState:Regenerate)-[:OrigNext*0..]->(os:OrigState:Regenerate)
WHERE NOT (:Regenerate)-[:OrigNext]->(root)
CREATE (os)<-[:Origin]-(s:State:Regenerate)
SET s = os
WITH s, os
MATCH (os)-[on:OrigNext]->(nos:OrigState)<-[:Origin]-(ns:State:Regenerate)
CREATE (s)-[:NEXT { symbol: on.symbol }]->(ns)
```

F.1.1. kód. Automatabeli állapotok létrehozása a lefutás állapotaiból.

```
// initIntervals
MATCH ()-[on:OrigNext]->()
WITH max(on.time) as maxTime, min(on.time) as minTime
MATCH (:State:Regenerate)-[n:NEXT]->(:State:Regenerate)
SET n.Tmin = minTime
SET n.Tmax = maxTime
```

F.1.2. kód. Minimum és maximum idők beállítása.

```
// markMerge1
MATCH (s1:State:Regenerate)
WHERE NOT (:State:Regenerate)-[:NEXT]->(s1)
SET s1:Merge
WITH 1 AS dummy MATCH (n) RETURN n
```

F.1.3. kód. A lefutási fa gyökerének kiválasztása.

```
// selectVertexToKeep
MATCH (v:Merge)
WHERE NOT (v)-[:NEXT]->()
WITH v
LIMIT 1
SET v:Keep
REMOVE v:Merge
WITH count(v) AS cv
// if there was no vertex to keep, we pick a random one
MATCH (r:Merge)
WHERE cv = 0
WITH r
LIMIT 1
SET r:Keep
REMOVE r:Merge
WITH count(*) AS dummy MATCH (n) RETURN n
```

F.1.4. kód. Összevonandó állapotokból a megmaradó kiválasztása.

```
// treeMerge
MATCH (k:Keep), (m:Merge)
OPTIONAL MATCH (m)-[n1]->(s)
REMOVE k:Keep
SET k += m
WITH k, s, n1, type(n1) as edgeType
WHERE s IS NOT NULL
CALL apoc.create.relationship(k, edgeType, {}, s) yield rel
SET rel = n1
WITH count(*) AS dummy
// delete merge candidates
MATCH (m:Merge)
DETACH DELETE m
// remove keep if not before
WITH count(*) AS dummy
MATCH (k:Keep)
REMOVE k:Keep
// return graph
WITH count(*) AS dummy MATCH (n) RETURN n
```

F.1.5. kód. Az összevonás.

```
// markMerge2
MATCH (s1:State:Regenerate)<-[n1:NEXT]-(s0:State:Regenerate)-[n2:NEXT]->(s2:State:Regenerate)
WHERE n1.symbol = n2.symbol
WITH s0, n1.symbol AS symbol
LIMIT 1
MATCH (s0)-[n:NEXT {symbol: symbol}]->(s3:State:Regenerate)
SET s3:Merge
RETURN count(s3) as markCount
```

F.1.6. kód. Összevonandó állapotokból kiválasztása.

```
// removeRegenerateLabels
MATCH (s:Regenerate)
REMOVE s:Regenerate
```

F.1.7. kód. Eltávolítjuk a generálás bélyegét.

```
// setRedLabelToRoot
MATCH (s:State)
WHERE NOT ()-[:NEXT]->(s)
SET s:Red
SET s:InitialState
WITH 1 AS dummy MATCH (n) RETURN n
```

F.1.8. kód. A fa gyökerének pirosra színezése.

```
MATCH (b:Blue)
REMOVE b:Blue
```

F.1.9. kód. Eddigi (előző iterációban érvényes) kék színezés eltávolítása.

```
MATCH (r:Red)-[:NEXT]->(s)
WHERE NOT s:Red
SET s:Blue
RETURN *
```

F.1.10. kód. A mostani iterációban érvényes kék színezés.

```
// notFinished
MATCH (s:State)
WHERE NOT s:Red
RETURN s
```

F.1.11. kód. Nem piros állapotok keresése.

```
// maxSplitScore
MATCH (r:Red)-[n:NEXT]->(b:Blue)
WITH r, b, n.Tmax as Tmax, n.Tmin as Tmin, n.symbol as symbol
UNWIND range(Tmin, Tmax-1) AS t
MATCH (b)-[:NEXT*0..]->(s1:State)-[:Origin]->(so1:OrigState), trace1=(so1)<-[OrigNext*0..]-(
    redOrigNext:OrigState), (redOrigNext)<-[no1:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace1) WHERE (x)<-[:Origin]-(r))
WITH r, b, t, Tmin, Tmax, symbol, s1, no1, so1
MATCH (b)-[:NEXT*0..]->(s2:State)-[:Origin]->(so2:OrigState), trace2=(so2)<-[OrigNext*0..]-(
    redOrigNext:OrigState), (redOrigNext)<-[no2:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace2) WHERE (x)<-[:Origin]-(r))
WITH t, r, b, s1, s2, toInteger(no1.time)>t as cat1, toInteger(no2.time)>t as cat2, Tmin, Tmax,
    symbol, so1, so2
WHERE s1 = s2
    AND cat1 <> cat2
    AND id(so1) < id(so2)
WITH r, b, t, Tmin, Tmax, symbol,
    [coalesce(so1.accepting, false), coalesce(so1.rejecting, false)] AS so1ar,
    [coalesce(so2.accepting, false), coalesce(so2.rejecting, false)] AS so2ar
WITH
    r, b, t, Tmin, Tmax, symbol,
    CASE
        WHEN so1ar[0] <> so2ar[0] AND so1ar[1] <> so2ar[1] THEN 1
        WHEN so1ar =  so2ar AND (so1ar = [false, true] OR so1ar = [true, false]) THEN -1 // there is
    at least one true
        ELSE 0
    END AS score
WITH r, b, t, sum(score) AS metric, Tmin, Tmax, symbol
ORDER BY metric DESC
LIMIT 1
WITH r, b, t, metric, Tmin, Tmax, symbol
MATCH (b)-[:NEXT*0..]->(s1:State)-[:Origin]->(so1:OrigState), trace1=(so1)<-[OrigNext*0..]-(
    redOrigNext:OrigState), (redOrigNext)<-[no1:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace1) WHERE (x)<-[:Origin]-(r)) AND toInteger(no1.time)>t
WITH r, b, t, metric, Tmin, Tmax, symbol, collect(so1) as so1s
MATCH (b)-[:NEXT*0..]->(s2:State)-[:Origin]->(so2:OrigState), trace2=(so2)<-[OrigNext*0..]-(
    redOrigNext:OrigState), (redOrigNext)<-[no2:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace2) WHERE (x)<-[:Origin]-(r)) AND  toInteger(no2.time)<=t
RETURN r, b, t, metric as m, Tmin, Tmax, symbol as sym, so1s, collect(so2) as so2s
```

F.1.12. kód. A legnagyobb *split* metrika kiszámítása.

```
// deleteBlueSubtree
WITH $b as b
MATCH (b)-[:NEXT*0..]->(s:State)
DETACH DELETE s
```

F.1.13. kód. A kék állapotból induló részfa törlése.

```
// linkBlueSubtree
WITH $t AS t, $r AS r, $symbol AS symbol, $Tmin AS Tmin, $Tmax AS Tmax
MATCH (s:State)-[:Origin]->(:OrigState)<-[on:OrigNext]-(:OrigState)<-[:Origin]-(r)
WHERE NOT (:State)-[:NEXT]->(s)
AND NOT (s:Red)
CREATE (r)-[:NEXT {
symbol: symbol,
Tmin: CASE toInteger(on.time) <= t WHEN true THEN Tmin ELSE t + 1 END,
Tmax: CASE toInteger(on.time) <= t WHEN true THEN t    ELSE Tmax  END
}]->(s)
```

F.1.14. kód. Az új részfák megfelelő időkorláttal való bekötése az automatába.

```
// mergeCandidates
MATCH (r:Red), (b:Blue)
WHERE NOT (r)-[:Checked]-(b)
WITH r, b
LIMIT 1
CREATE (r)-[:Checked]->(b)
RETURN r, b
```

F.1.15. kód. Még nem vizsgált piros és kék állapotpár keresése *merge* művelethez.

```
// splitInMergeCandidates
WITH $r as r, $b as b
MATCH rp=(r)-[:NEXT*0..]->(:State)
MATCH bp=(b)-[:NEXT*0..]->(:State)
WHERE length(rp) = length(bp)
WITH r, b, rp, bp, [rr IN rels(rp) | rr.symbol] AS rss, [br IN rels(bp) | br.symbol] AS bss,
[rr IN rels(rp) | rr.Tmin] AS rmins, [br IN rels(bp) | br.Tmin  ] AS bmins,
[rr IN rels(rp) | rr.Tmax] AS rmaxs, [br IN rels(bp) | br.Tmax  ] AS bmaxs
WITH r, b, rss, bss, rmins, rmaxs, bmins, bmaxs, rp, bp, [ i IN range(0, length(rp)) ] AS is
UNWIND is as i
WITH r, b, rss, bss, rmins, rmaxs, bmins, bmaxs, rp, bp, i

WHERE rss[i] = bss[i]
WITH bss[i] as symbol,
    CASE
        WHEN rmaxs[i] < bmaxs[i] AND rmins[i] = bmins[i]
        THEN [nodes(bp)[i], nodes(bp)[i+1], rmaxs[i], bmins[i], bmaxs[i]]
        WHEN bmaxs[i] < rmaxs[i] AND bmins[i] = rmins[i]
        THEN [nodes(rp)[i], nodes(rp)[i+1], bmaxs[i], rmins[i], rmaxs[i]]
    END as params
WHERE params IS NOT NULL
WITH symbol, params[0] as r, params[1] as b, params[2] as t, params[3] as Tmin, params[4] as Tmax
LIMIT 1

OPTIONAL MATCH (b)-[:NEXT*0..]->(s1:State)-[:Origin]->(so1:OrigState), trace1=(so1)<-[OrigNext*0..]-(
     redOrigNext:OrigState), (redOrigNext)<-[no1:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace1) WHERE (x)<-[:Origin]-(r)) AND toInteger(no1.time)>t
WITH r, b, t, Tmin, Tmax, symbol, collect(so1) as so1s
OPTIONAL MATCH (b)-[:NEXT*0..]->(s2:State)-[:Origin]->(so2:OrigState), trace2=(so2)<-[OrigNext*0..]-(
     redOrigNext:OrigState), (redOrigNext)<-[no2:OrigNext]-(redOrig:OrigState)<-[:Origin]-(r)
WHERE none(x IN nodes(trace2) WHERE (x)<-[:Origin]-(r)) AND toInteger(no2.time)<=t
RETURN r, b, t, Tmin, Tmax, symbol, so1s, collect(so2) as so2s
```

F.1.16. kód. A szükséges *split* műveletek paramétereinek megkeresése.

```
WITH $r as r, $b as b
MATCH rp=(r)-[:NEXT*0..]->(:State)
MATCH bp=(b)-[:NEXT*0..]->(:State)
WITH r, b, rp, bp, [rr IN rels(rp) | rr.symbol] AS rss, [br IN rels(bp) | br.symbol] AS bss,
                   [rr IN rels(rp) | rr.Tmin] AS rmins, [br IN rels(bp) | br.Tmin  ] AS bmins,
                   [rr IN rels(rp) | rr.Tmax] AS rmaxs, [br IN rels(bp) | br.Tmax  ] AS bmaxs
WHERE rss = bss AND rmins = bmins AND rmaxs = bmaxs
WITH
    r, b, rp, bp,
    [ i IN range(0, length(rp)) | [nodes(rp)[i], nodes(bp)[i]] ] AS pairs
UNWIND pairs AS pair
WITH r, b, collect(DISTINCT pair) AS dpairsResult

UNWIND range(0, length(dpairsResult)-1) AS i
WITH dpairsResult[i][0] AS rpc, dpairsResult[i][1] AS bpc, i, r, b, dpairsResult
CREATE (rpc)-[:IndexPair {index: i}]->(bpc)
SET rpc:IndexedMerge, bpc:IndexedMerge
RETURN r, b
```

F.1.17. kód. Összevonandó állapotok megjelölése.

```
// createIndexPairs
MATCH (s1:IndexedMerge)-[:IndexPair*..]-(s2:IndexedMerge),
s1p = (s1)-[:NEXT*..]->(s12:State), s2p = (s2)-[:NEXT*..]->(s22:State)
WHERE id(s1) > id(s2) AND length(s1p) = length(s2p)
AND NOT (s12)-[:IndexPair*0..]-(s22)
WITH s12, s22, s1p, s2p,[s1r IN rels(s1p) | s1r.symbol] AS s1ss, [s2r IN rels(s2p) | s2r.symbol] AS
    s2ss,
[s1r IN rels(s1p) | s1r.Tmin] AS s1mins, [s2r IN rels(s2p) | s2r.Tmin  ] AS s2mins,
[s1r IN rels(s1p) | s1r.Tmax] AS s1maxs, [s2r IN rels(s2p) | s2r.Tmax  ] AS s2maxs
WHERE s1ss = s2ss AND s1mins = s2mins AND s1maxs = s2maxs
WITH collect([s12, s22]) as pairs

MATCH ()-[ip:IndexPair]->()
WITH max(ip.index) + 1 as nextIndex, pairs

UNWIND range(0, length(pairs)-1) as idx
WITH pairs[idx][0] as s12, pairs[idx][1] as s22, idx + nextIndex as edgeIndex
CREATE (s12)-[:IndexPair {index: edgeIndex}]->(s22)
SET s12:IndexedMerge, s22:IndexedMerge
```

F.1.18. kód. További összevonásra kerülő állapotok megjelölése.

```
// mergeMetric
WITH $r as r, $b as b
MATCH (rpc:IndexedMerge)-[:IndexPair*1..]-(bpc:IndexedMerge)
WHERE id(rpc)>id(bpc)
WITH r, b, collect(DISTINCT [rpc, bpc]) AS dpairs

WITH r, b, dpairs,  reduce(acc = [0,true], pair IN dpairs |(
CASE
    WHEN coalesce(pair[0].accepting, false) = coalesce(pair[1].accepting, false)
        AND coalesce(pair[0].rejecting, false) = coalesce(pair[1].rejecting, false)
        AND coalesce(pair[0].accepting, false) <> coalesce(pair[0].rejecting, false)
    THEN [acc[0] + 1, acc[1]]
    WHEN ((coalesce(pair[0].accepting, false) OR coalesce(pair[1].accepting, false))
        AND (coalesce(pair[0].rejecting, false) OR coalesce(pair[1].rejecting, false)))
        AND (pair[0]:Red OR pair[1]:Red)
    THEN [0,false]
    WHEN coalesce(pair[0].accepting, false) <> coalesce(pair[1].accepting, false)
        AND coalesce(pair[0].rejecting, false) <> coalesce(pair[1].rejecting, false)
    THEN [acc[0] - 1, acc[1]]
    ELSE [acc[0], acc[1]]
END)) AS metric
WHERE metric[1]
WITH r, b, metric[0] as metric, dpairs
ORDER BY metric DESC
LIMIT 1
RETURN r, b, dpairs, metric
```

F.1.19. kód. A *merge* metrikájának kiszámolása.

```
// cleanIndexPairs
MATCH (s:IndexedMerge)
REMOVE s:IndexedMerge
WITH count(*) as dummy
MATCH ()-[e:IndexPair]->()
DELETE e
```

F.1.20. kód. Összevonás jelölés megszűntetése.

```
// merge
WITH $r as r, $b as b, $dpairs as dpairs
UNWIND range(0, length(dpairs)-1) AS i
WITH dpairs[i][0] AS rpc, dpairs[i][1] AS bpc, i
CREATE (rpc)-[r:IndexPair {index: i}]->(bpc)
SET rpc:IndexedMerge, bpc:IndexedMerge
RETURN i
```

F.1.21. kód. Indexpárok behúzása.

```
// createSingleNewState
MATCH (rpc:IndexedMerge)-[r:IndexPair]->(bpc:IndexedMerge)
WHERE rpc <> bpc
WITH rpc, bpc, r.index as idx
ORDER BY idx DESC
LIMIT 1
CREATE (new:NewState)
WITH new, rpc, bpc, idx
CALL apoc.create.addLabels([ new ], labels(rpc)) YIELD node
WITH new, rpc, bpc, idx
CALL apoc.create.addLabels([ new ], labels(bpc)) YIELD node
SET new += bpc
SET new += rpc         // to copy acceptance properties
SET rpc:MergeSource
SET bpc:MergeSource
RETURN idx as mergeIndex
```

F.1.22. kód. Új állapot létrehozása.

```
// copyEdgesToNewState
MATCH (newState:NewState), (origState:MergeSource)-[origEdge]-(origNeighbor)
WITH origEdge, newState, startNode(origEdge) as origStart, endNode(origEdge) as origEnd
WITH origEdge, newState
, CASE WHEN origStart:MergeSource THEN newState ELSE origStart END AS newStart
, CASE WHEN origEnd:MergeSource THEN newState ELSE origEnd END AS newEnd
WITH DISTINCT newStart, newEnd, origEdge, type(origEdge) as edgeType
CALL apoc.create.relationship(newStart, edgeType, {}, newEnd) yield rel
SET rel = origEdge
```

F.1.23. kód. Új állapotba a megfelelő élek behúzása.

```
// deleteMergedStates_RemoveNewStateLabel
MATCH (newState:NewState), (origState:MergeSource)
REMOVE newState:NewState
DETACH DELETE origState
WITH count(*) as dummy
MATCH (newState:State)-[r:IndexPair]->(newState)
DELETE r
```

F.1.24. kód. Már összevont állapotok törlése és az új állapot *NewState* címkéjének eltávolítása.

```
// deleteParallelEdges
MATCH (src:State)-[r1:NEXT]->(trg:State), (src)-[r2:NEXT]->(trg)
WHERE r1.symbol = r2.symbol
AND r1.Tmin = r2.Tmin
AND r1.Tmax = r2.Tmax
WITH tail(collect(r1)) AS copies, src, trg, r1.symbol AS sym
FOREACH (value in copies | DELETE value)
```

F.1.25. kód. Párhuzamos állapotátmenetek megszüntetése.

```
// removeIndexedMergeLabel
MATCH (s:IndexedMerge)
REMOVE s:IndexedMerge
REMOVE s.redIndex
REMOVE s.blueIndex
```

F.1.26. kód. Összevonás jelzésének eltávolítása.

```
// joinEdgeTimeguards
MATCH (s:State)-[e1:NEXT]->(t:State), (s)-[e2:NEXT{symbol:e1.symbol, Tmin:e1.Tmax+1}]->(t)
WITH s, t, e1, e2
ORDER BY e2.Tmax DESC
SET e1.Tmax = e2.Tmax
DELETE e2
```

F.1.27. kód. Átlapolódó állapotátmenetek összevonása.

```
// maxColorScore
MATCH (b:Blue)
WHERE NOT (coalesce(b.accepting, false) AND coalesce(b.rejecting, false))
WITH b
ORDER BY b.trace DESC
LIMIT 1
WITH b, 0 as metric
RETURN b, metric
```

F.1.28. kód. A color lekérdezés.

```
// color
WITH $b as b
SET b:Red
```

F.1.29. kód. Egy megfelelő kék állapot átszínezése pirosra.

F.2. Java kódrészletek

```
driver.run(markMerge1);

boolean doMerge = true;
while (doMerge) {
  driver.run(selectVertexToKeep);
  driver.run(treeMerge);
  doMerge = driver.runWReturn(markMerge2Result.class, iterator ->   iterator.next().markCount.
     longValue() != 0);
}

driver.run(removeRegenerateLabels);
driver.run(setRedLabelToRoot);
```

F.2.1. kód. A *tapta* előállításának algoritmusa Java nyelven

```
while (driver.run(createSingleNewState)) {
  driver.run(copyEdgesToNewState);
  driver.run(deleteMergedStates_RemoveNewStateLabel);   driver.run(deleteParallelEdges);
}

driver.run(removeIndexedMergeLabel);
driver.run(joinEdgeTimeguards);
```

F.2.2. kód. A *merge* művelet végrehajtásának algoritmusa

```
while (driver.run(notFinished)) {
  driver.run(removeAndSetBlueLabel);

  // ...

  // if merge scores the highest
  calculateMaxMergeMetric(true);

  // if split scores the highest
  if (getMaxScore(maxSplitScoreResult.class, null))
    operation = Operation.split;

  // if color scores the highest
  if (getMaxScore(maxColorScoreResult.class, null))
    operation = Operation.color;

  // ...
```

F.2.3. kód. Legjobb metrikájú művelet kiválasztása

```
switch (operation) {
  case merge:
    merge();
    break;
  case split:
    split(map.get());
    break;
  default:
    color();
    break;
}
```

F.2.4. kód. Legjobb metrikájú művelet végrehajtása